Essential algorithms and data structures

Second Edition

D Hillyard

C Sargent

Published by
CRAIGNDAVE LTD
12 Tweenbrook Avenue
Gloucester
Gloucestershire
GL1 5JY
United Kingdom
admin@craigndave.co.uk
www.craigndave.org
2021

Craig 'n' Dave

Front cover artwork: The binary tree created using www.visnos.com/demos/fractal.

Text © D. Hillyard and C. Sargent 2022
Original illustrations © D. Hillyard and C. Sargent 2022
Edited by A. Fenn
Graphics and typesetting by CraignDave Ltd
First edition 2018

ISBN: 9798468205341

About the authors

Craig Sargent

Craig is a postgraduate qualified teacher with a Bachelor of Science (Honours) in Computer Science with Geography. Craig has over seventeen years' teaching experience in ICT and computer science in two Gloucestershire schools. An examiner and moderator for awarding bodies in England, Craig has authored many teaching resources for major publishers.

Formerly a Computing at School (CAS) Master Teacher, Craig has also served as regional coordinator for the Computing Network of Excellence and contributed to the development of the National Curriculum for Computer Science. His industry experience includes programming for a high street bank and the Ministry of Defence. He also wrote his first computer game in primary school.

Both Craig and David contribute to the National Centre for Computing Education.

David Hillyard

David is a postgraduate qualified teacher with a Bachelor of Science (Honours) in Computing with Business Information Technology. David has over twenty years' teaching experience in ICT and computer science in five Gloucestershire secondary schools.

Formerly subject leader for the Gloucestershire Initial Teacher Education Partnership (GITEP) and graduate teacher programme (GTP) at the University of Gloucestershire, David has led a team of trainee teacher mentors across the county in ICT and computer science.

His industry experience includes programming for the Ministry of Defence. A self-taught programmer, he wrote his first computer game at ten years old.

Preface

The aim of this book is to provide students and teachers of A Level Computer Science with a comprehensive guide to the algorithms and data structures students need to understand for examinations. Each chapter considers a data structure or algorithm in isolation, including:

- An explanation of how it works
- Real-world applications
- A step-by-step example
- Pseudocode
- Actual printed Python 3.x code listing
    - Additional code listings in Visual Basic and C# are provided via download
- A description of its space and time complexity

Additional support materials

In this book, coded solutions are provided in Python 3.x, the most popular language taught at GCSE. Solutions for Visual Basic (2015 onwards) and C# (2019 onwards) are also available to download from craigndave.org/algorithms.

Standards

Python source code conforms to the PEP-8 standard as much as possible. Where this would introduce new keywords, the preference is to show code in the format students are more likely to see in an examination.

Coding standards used in this book

- ++ += -= for incrementing and decrementing variables has been avoided in favour of x = x + 1. Except for iterations in C# source code.
- Swapping variables using approaches such as a, b = b, a has not been used. A three-way swap with a temporary variable is favoured in examinations.
- Using classes for data structures and methods for their operations.
- Using the == comparison operator with *None* instead of *is*.
- Using shadow names from outer scope in subroutines for simplification.

The result is not necessarily the most efficient code but, rather, the most suitable implementation for this level of study. There are many ways to code algorithms and data structures. For example, the depth-first traversal can be coded using iteration or recursion with a dictionary, objects or arrays – that is six different implementations, but even these are not exhaustive. Combined with a programmer's personal approach and the commands available in their chosen language, the possibilities are near-endless.

It is important that students recognise the underlying data structures, understand the way an algorithm works and can determine outputs from code listings. Therefore, the approaches and solutions presented in this book are one solution – but not the only solution.

Examiners require students to be able to trace and write many algorithms, but we strongly recommend learning the principles of the algorithm and how it operates rather than attempting to memorise code listings.

Specification mapping

	OCR GCSE (J277)	OCR GCE AS Level (H046)	OCR GCE A Level (H446)	AQA GCSE (8525)	AQA GCE AS Level (7516)	AQA GCE A Level (7517)	Pearson Edexcel GCSE (1CP2)	WJEC GCSE (C00/1157/9)	WJEC GCE AS Level (601/5391/X)	WJEC GCE A Level (601/5345/3)	Cambridge IGCSE (0984) 2023-2025	Oxford AQA IGCSE (9210)	Pearson Edexcel IGCSE (4CP0)
Array	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Binary tree			✓			✓				✓			
Dictionary						✓							
Graph			✓			✓							
Linked list			✓							✓			
List	✓	✓	✓				✓			✓			
Object			✓			✓		✓	✓	✓			
Queue		✓	✓			✓				✓			
Record	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Stack		✓	✓		✓	✓				✓			
Binary search	✓	✓	✓	✓		✓	✓	✓	✓	✓		✓	✓
Hash table search			✓			✓				✓			
Linear search	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
Bubble sort	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
Insertion sort	✓	✓	✓						✓	✓			
Merge sort	✓		✓	✓		✓	✓	✓		✓		✓	✓
Quick sort			✓							✓			
A* pathfinding			✓*										
Dijkstra's shortest path			✓*			✓				✓			

* Students are not required to write code for this algorithm – only to know how the algorithm works and be able to trace its pseudocode.

Contents

ABSTRACTIONS OF MEMORY

The way in which the computer uses memory to store and retrieve data impacts the efficiency of the algorithm using that data.

Introduction to data structures

Algorithms are a sequence of instructions that a human or computer can follow to solve a problem. All algorithms require data structures – somewhere to store data while the algorithm is executing. The simplest data structure is the variable, an area of memory holding a single item of data that can be changed when a program is running with assignment statements. Usually, multiple data items are required and related to each other in some way. Related data is stored in more complex data structures like arrays, lists and objects.

Algorithms and data structures are held in primary memory – usually random-access memory (RAM) – although during execution, they will also be held in the cache, an area of high-speed memory in the CPU for temporary storage of frequently used instructions and data.

Memory is divided into several parts. Three important memory partitions include:

1. Code
2. Call stack
3. Heap

The code section holds the instructions for the algorithm. The call stack (also known as the program stack, execution stack, control stack, run-time stack, machine stack or simply the stack) is an area of memory used by the computer to manage subroutines and their data structures.

A stack frame holds local variable values, parameters and the state of the registers when the subroutine was called, including the return address in the program counter. Beyond simple values of local variables, other data structures and global variables are stored on the heap with only the reference (address) to the structure being held in the stack frame.

Confusingly, the actual implementation of the call stack and the heap for the different data structures is largely dependent on the architecture of the computer and programming language. For example, with C++, objects can be allocated on the stack or the heap. In Java, all objects are allocated to the heap.

Did you know?

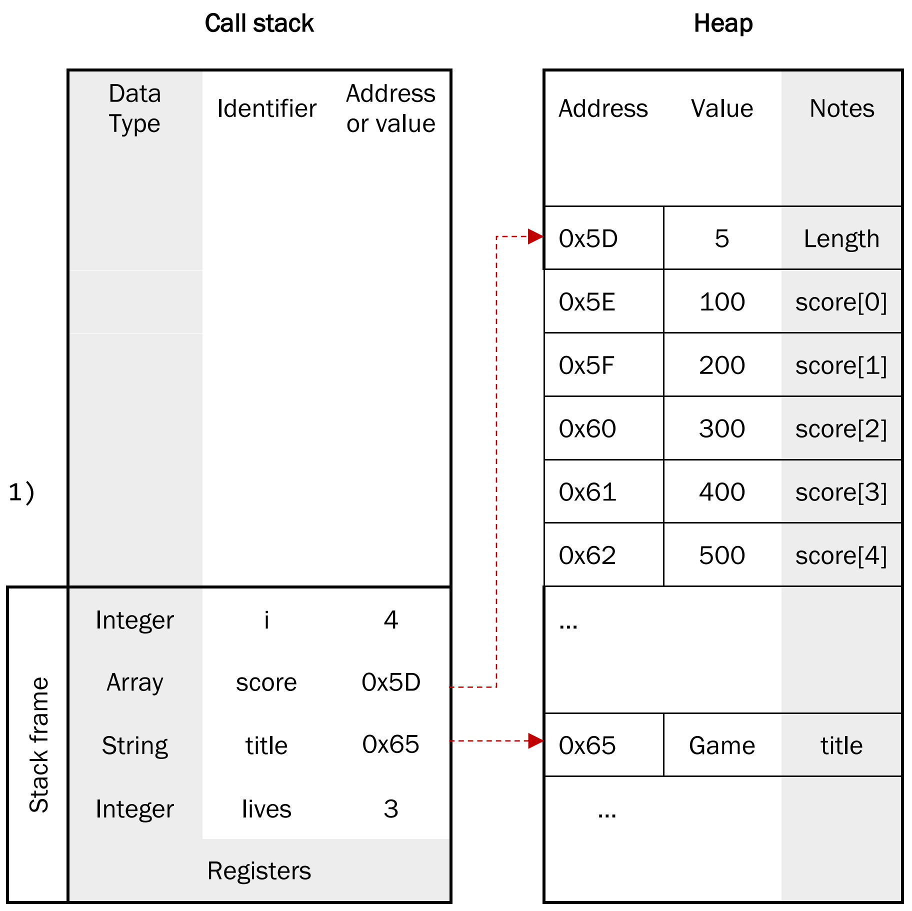
In Python, the following code will output the memory address where the global variable x is being stored on the heap. 0x in the output indicates that the address is in hexadecimal.

```
x = "Hello World"
print(hex(id(x)))
```

Code, the call stack and the heap illustrated

Code

```
Sub initialise()
  int lives = 3
  string title = "Game"
  int score[5]
  for i = 0 to 4
    score[i] = 100 * (i + 1)
  next i
End Sub
```



As data structures are created, memory on the heap is allocated to them. Live structures are tracked, and everything else in the heap is designated as unused. A process known as garbage collection reclaims unused memory to be used for future allocation.

Data structures are often referred to as static, dynamic, mutable and immutable. Before looking at each data structure in more detail, it will help you to be aware of what these terms mean.

- Data type: What a sequence of binary digits represents – e.g., integer, floating-point number, string, date. Each data type requires a specific number of bits to be stored in memory.
- Static: The data structure cannot change in size when the program is running.
- Dynamic: The data structure can change size when a program is running.
- Mutable: The data in the data structure can change when a program is running.
- Immutable: The data in the data structure cannot change when a program is running.

DATA STRUCTURES

Supported by some programming languages as primitive data types within the command set, programmers can also implement their own data structures.

Array

An array is a static collection of items of the same data type called elements. An array is typically used when the required number of data items is known in advance, to store data that would otherwise be stored in multiple variables. For example, to store the names of four people, you could have four variables – name1, name2, name3 and name4.

The problem with this approach is it would not be possible to refer to a specific name or loop through all the names with a FOR or WHILE command because the number or index is part of the identifier – the name of the variable. Instead, we need to declare name1 as name[1] and so on. Note how the index is now enclosed in brackets. Some programming languages used curved brackets while others use square brackets.

Assigning data to the array can be carried out using syntax similar to:

```
name[0] = "Craig"
name[1] = "Dave"
name[2] = "Sam"
name[3] = "Carol"
```

With four names, the maximum index is 3 because arrays are usually zero-indexed – i.e., the first item is stored at index zero. You could start at index 1, but why waste memory unnecessarily?

Now, it is possible to use an iteration to output all the data in the array:

```
For index = 0 to 3
       Print(name[index])
Next
```

This is an extremely useful algorithm that you need to know at all levels of study. Alternative code would be:

```
Print(name1)
Print(name2)
Print(name3)
Print(name4)
```

It is a misconception that using an array instead of individual variables is more memory-efficient. The same amount of data still needs to be stored, but an array makes the algorithm scalable, so we only need to change the number 3 in the iteration to change the number of names we output. You can see how implementing code with a thousand names without iteration would be time-consuming and impractical.

There are two limitations of using arrays.

1. All data elements in an array must be of the same data type. For example, an array of strings cannot contain an integer.
2. The size of an array is determined when it is declared – i.e., when the algorithm is written – and cannot usually be changed when the program is running.

Applications of an array

Arrays are used when a known number of related data items of the same data type need to be stored. They are used extensively in supercomputers and graphics processing units.

Some programming languages and hardware implementations allow a single operation to be performed on many indexes at the same time, making arrays very efficient for mathematical operations. All algorithms can use an array as their base data structure.

Storing an array in memory

Arrays are always stored in contiguous memory, meaning each element follows the previous one in memory. Their size is fixed because a sufficient block of memory needs to be allocated in advance. Since all elements must be of the same data type, it is easy to calculate where an item is in memory from the base address of the structure, the data type being stored and the index.

For example, an array of 32-bit (4-byte) integers with index 0 stored at address 0x1A would have index 1 stored at 0x1A + 4 = 0x1E – this makes arrays very efficient, as any element can be found immediately by its address (known as random-access) rather than sequentially.

To overcome the limitation of arrays being a fixed size, they can be declared larger than they need to be so data can be inserted into an empty index later. Although this is an inefficient use of memory, it is necessary to implement dynamic data structures with a static array.

Modern programming languages do allow for the resizing of an array at run-time by finding a new place in the heap for the entire data structure and moving it to that location. This is a slow O(n) operation, so if it is frequently required, the programmer should consider using a list or objects instead.

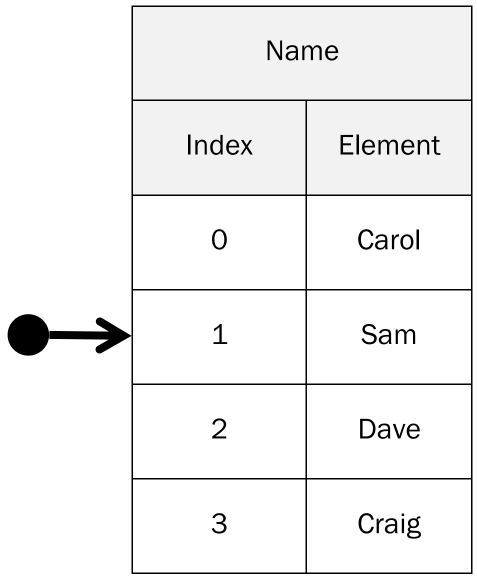
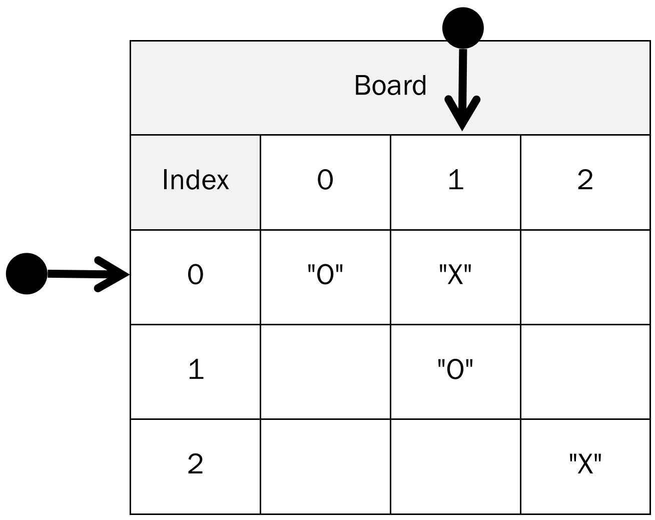
Arrays may have multiple indexes (also called indices) in what is known as a multi-dimension array. An array of two dimensions is referred to as a two-dimensional array. By using two dimensions, we can store a grid of data – for example, to represent an 8 x 8 board for a game of chess.

Did you know?

One-dimensional arrays are sometimes called vectors, and two-dimensional arrays are sometimes called matrices.

All data structures including dictionaries, linked lists, graphs, queues, stacks and trees can be constructed from arrays.

Some modern programming languages such as Java implement arrays using objects.

One-dimensional array	Two-dimensional array
 Name[2] = Dave	 Board[1,0] = “X”

Name	
Index	Element
0	Carol
1	Sam
2	Dave
3	Craig

Board			
Index	0	1	2
0	"O"	"X"	
1		"O"	
2			"X"

Notice that in the example of the two-dimensional array above, the first index was chosen to represent the x position in the table, and the second index represents the y position.

It is also acceptable for the first index to represent the y position and the second index to represent the x position. In memory, the table doesn’t exist as it is shown. Indexes are simply memory addresses, so the structure itself is abstracted. It is up to the programmer how they visualise the data structure in their own mind.

Arrays can be declared in any number of dimensions. For example, when storing a linked list using an array, one dimension is used for the data item, a second dimension is used for the pointer to the next item.

Operations on an array

Typical operations that can be performed on an array include:

- Declare: Initialise the size of an array – i.e., the number of indexes it will have – and reserve memory.
- Access: Return an element from an index.
- Assign: Set the element for a specific index.
- Resize: Change the size of the array at run-time by reallocating it to a different part of the memory.

Efficiency of operations on an array

	Time complexity			Space complexity	
	Best case	Average case	Worst case	Best case	Worst case
Access, Assign	$O(1)$ Constant	$O(1)$ Constant	$O(1)$ Constant	$O(1)$ Constant	$O(1)$ Constant
Search, Iterate	$O(1)$ Constant	$O(n)$ Linear (one-dimension)	$O(n^2)$ Polynomial (two-dimension)		
Add item	$O(1)$ Constant	$O(n)$ Linear	$O(n)$ Linear		
Delete item	$O(1)$ Constant	$O(n)$ Linear	$O(n)$ Linear		
Resize	$O(n)$ Linear (one-dimension)	$O(n)$ Linear (one-dimension)	$O(n^2)$ Polynomial (two-dimension)		

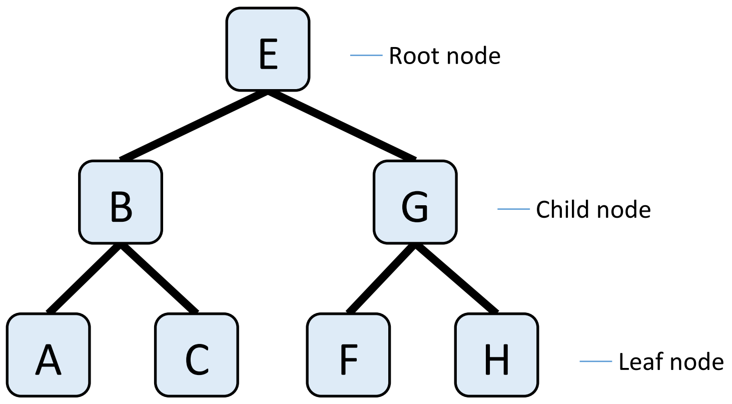
Since an array is considered a static data structure, its memory footprint is always known at run-time, so it has a constant space complexity of $O(1)$. Some languages support the resizing of an array at run-time, which can require moving the structure in memory – an $O(n)$ process for a one-dimension array.

The advantage of using an array is that it can be randomly accessed, meaning you can jump immediately to any index/memory address in the data structure, resulting in basic operations executing in a constant time $O(1)$ – for example, adding a new item to an empty index. However, if that index must be found first or the item needs to be inserted between two existing indexes to maintain an order to the data, the time complexity will degrade to $O(n)$ because other elements must be considered or moved first.

With multi-dimensional arrays, if a nested loop is required to iterate over all the elements – for example, with a bubble sort – the time complexity becomes polynomial, $O(n^2)$. The exponent represents the number of dimensions in the structure. In the case of a three-dimensional array, it would be $O(n^3)$.

Binary tree

A binary tree is a dynamic structure of nodes and pointers. A binary tree is a special case of a graph where each node can only have zero, one or two pointers, with each pointer connecting to a different node. Unlike a regular tree, the binary tree is a rooted tree. It has a first node from which all operations start, known as the root node. The connected nodes are called child nodes, while the nodes at the very bottom of the structure are referred to as leaf nodes. Unlike graphs, trees do not allow for circular links between nodes.



Since a binary tree is essentially a special type of undirected graph, the nodes and pointers can also be described as vertices and edges, respectively.

There are two types of binary trees. With a binary unordered tree, items added to the tree are not maintained in a logical order. The balance of the tree is always maintained, with new nodes simply being added to the next available leaf node. However, when discussing binary trees, we are commonly referring to a binary search tree where a logical order to the items is maintained as they are entered into the tree.

With a binary search tree, instead of new nodes being entered at the next available leaf node, items that are lower than the root or child node are placed to the left, while items that are greater than the root or child node are placed to the right.

The structure may become unbalanced, with one side of the tree containing more nodes than the other. It is possible to rebalance a binary search tree so that each leaf node is the same distance away from the root node as any other leaf node. Doing so ensures the structure is as efficient as possible but does require you to output the data and recreate the tree.

The significant advantage of a binary search tree is that by using traversal algorithms, it is possible to output nodes in order regardless of the order they were entered into the tree.

From this point onwards, we will only consider the binary search tree, as this is the data structure you will be expected to understand for examinations.

Applications of a binary tree

A binary tree has many uses in computer science. They are frequently used for storing and searching large data sets. Binary trees can be found in routing tables for packet switching within routers, where similar addresses are grouped under a single sub-tree. Decision trees are often used in supervised machine learning algorithms and expression evaluation for compilers.

Huffman coding – used in compression algorithms such as JPEG and MP3 – utilises binary trees, as does cryptography with GGM trees. Binary trees are also useful for performing arithmetic with reverse Polish notation, which negates the use of brackets to define the order of precedence for operators in an expression.

Storing a binary tree in memory

A binary tree can be represented in memory using an array or list, with node objects or in a dictionary.

Array

Index	Element	Left	Right
0	E	1	2
1	B	3	4
2	G	5	6
3	A	7	8
4	C	9	10
5	F	11	12
6	H	13	14

Object

A single node is defined as:

```
Class Node
        element = ""
        left_pointer = Node
        right_pointer = Node
End Class
```

Dictionary

```
binaryTree = {
  "E": ["B", "G"],
  "B": ["A", "C"],
  "G": ["F", "H"],
  "A": [],
  "C": [],
  "F": [],
  "H": []
}
```

Data structures are often confusing because they can be implemented using other data structures. For example, a dictionary is a hash map, and a hash map is a particular use of an array. So, storing a binary tree as a dictionary is also storing the binary tree using an array – just using it in a completely different way.

Operations on a binary tree

Typical operations that can be performed on a binary tree include:

- Add: Adds a new node to the tree.
- Delete: Removes a node from the tree.
- Binary search: Returns the data stored in a node.
- Pre-order traversal: A type of depth-first traversal.
- In-order traversal: A type of depth-first traversal.
- Post-order traversal: A type of depth-first traversal.
- Breadth-first search: Traverses the tree, starting at the root node and visiting each node at the same level before going deeper into the structure.
- Rebalance: The Day-Stout-Warren (DSW) algorithm is a method of generating a compact, balanced tree by converting the tree into a linked list using in-order traversal and performing a series of rotations to reshape it.

A traversal refers to the process of navigating through the structure and visiting each node only once. A traversal algorithm is used to find, update and output the data in a binary tree.

Did you know?

The binary tree used to be called a "bifurcating arborescence" – a name associated with graph theory – before its modern name became popular in computer science.

Array/list implementation of a binary tree

When a binary tree is stored as an array, the total number of indexes required can be determined by calculating 2^(depth + 1) – 1, the depth being the number of levels required. For example, a two-level binary tree will have 2^3 – 1 = 7 nodes or indexes.

Although a binary tree array is often depicted as having left and right index pointers, they are not necessary. However, they do make it easier to visualise the structure as a table and see how each node points to the next index. It is possible to reduce data storage requirements by using only a one-dimensional array to store a node and calculate the index of the left and right nodes when required:

- The index of the left node can be calculated as 2 * current index + 1.
- The index of the right node can be calculated as 2 * current index + 2.

Index	Element	Left node	Right node
0	E	(2 * 0) + 1 = 1	(2 * 0) + 2 = 2
1	B	(2 * 1) + 1 = 3	(2 * 1) + 2 = 4
2	G	(2 * 2) + 1 = 5	(2 * 2) + 2 = 6
3	A	(2 * 3) + 1 = 7	(2 * 3) + 2 = 8
4	C	(2 * 4) + 1 = 9	(2 * 4) + 2 = 10
5	F	(2 * 5) + 1 = 11	(2 * 5) + 2 = 12
6	H	(2 * 6) + 1 = 13	(2 * 6) + 2 = 14

One of the problems of storing a binary tree as an array in this way is there may be many unused leaf nodes or indexes in the array, resulting in the data structure having a higher memory footprint than is necessary to store the data. To negate this, it would be possible to always use the next available index when adding new items and store the values of the left and right pointers instead of calculating them.

The problem with this approach is you also need to store a pointer to the next available index or use an inefficient linear search to find it. The additional complexity arising from these operations means this implementation is rarely used but watch out for it in examinations. If a reduced memory footprint is required, an object-based approach is much easier to implement and more efficient.

When using an array, even though the underlying data structure is static with the reservation of empty space and unused indexes, it facilitates the creation of a dynamic data structure with a constant space complexity.

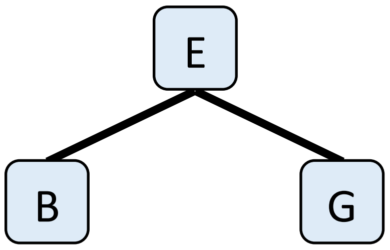
Object implementation of a binary tree

When implementing a binary tree with objects, a node class is defined. The class has the attributes:

- element: To store the value of the node.
- left_pointer: Another instance of the node class or null value if no child exists on the left edge.
- right_pointer: Another instance of the node class or null value if no child exists on the right edge.

Instances of the node class (objects) are created when required, making it a truly dynamic data structure.

Data structures created with objects are often visualised with each node/object represented as a box or circle with the data element written inside. Lines between the boxes indicate how the nodes are connected to each other using the left_pointer and right_pointer attributes – these are references to the base memory address of the connected object on the heap.



To create the structure depicted above, the root node, E, would be created first. Typically, objects are created (instantiated) with code like: `new_node = new Node`

`new_node` is a variable, storing the base memory address of a new instance of the object created on the heap. Variables used in this way are more commonly called pointers.

The root node of the binary tree can be assigned as: `root_node = new_node`

The data element of the root node can be assigned as: `root_node.element = "E"`

This does not create a new object because the keyword “new” was not used. Instead, `root_node` points to the base address of `new_node`.

Instantiating node B and connecting it to E would require the code:

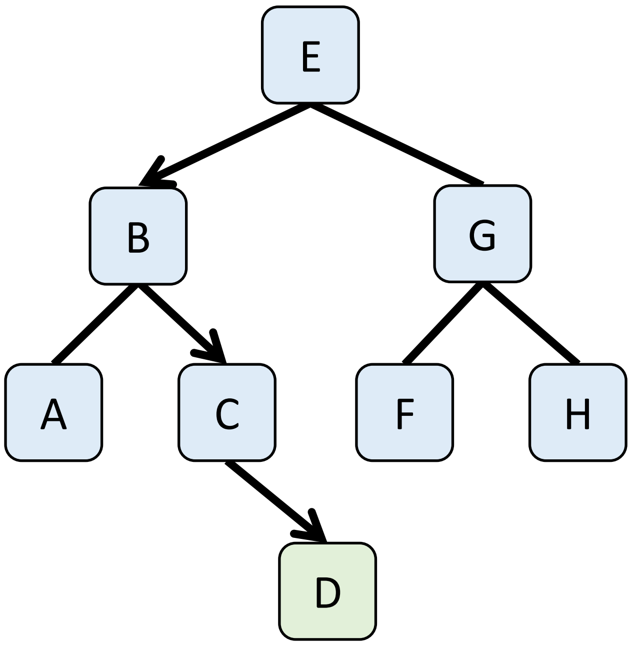
```
new_node = new Node
new_node.element = "B"
root_node.left_pointer = new_node
```

Adding an item to a binary tree

1. Check there is available memory for a new node – output an error if not.
2. Create a new node and insert data into it.
3. If the binary tree is empty:
    a. The new node becomes the first item – create a start pointer to it.
4. If the binary tree is not empty:
    a. Start at the root node.
    b. If the new node should be placed before the current node, follow the left pointer.
    c. If the new node should be placed after the current node, follow the right pointer.
    d. Repeat from step 4b until the leaf node is reached.
    e. If the new node should be placed before the current node, set the left pointer to be the new node.
    f. If the new node should be placed after the current node, set the right pointer to be the new node.

With an array implementation, step 2 is not necessary because, if the structure is not full, there will already be a vacant index. Instead, the position of the new node is found, and after step 3 or 4, the data can be inserted into the tree. Steps 4e and 4f are also unnecessary because it is possible to calculate the pointer from one node to any other node.

Adding an item to a binary tree illustrated



Start at the root node. D is less than E; follow the left pointer. D is more than B; follow the right pointer. D is more than C; create a right pointer from C to D.

Pseudocode for adding an item to a binary tree

```
If not memoryfull Then
      new_node = New Node
      new_node.left_pointer = Null
      new_node.right_pointer = Null
      current_node = start_pointer
      If current_node == Null Then
            start_pointer = new_node
      Else
            While current_node != Null
                  previous_node = current_node
                  If new_node < current_node Then
                        current_node = current_node.left_pointer
                  Else
                        current_node = current_node.right_pointer
                  End If
            End While
            If new_node < previous_node Then
                  previous_node.left_pointer = new_node
            Else
                  previous_node.rigth_pointer = new_node
            End If

      End If
End If
```

Did you know?

The maximum efficiency of a binary tree is achieved when the tree is balanced, meaning one branch is not significantly larger than another. A self-balancing binary tree automatically keeps its height small when items are added and deleted by moving nodes within the structure. A self-balancing operation decreases the efficiency of adding and deleting items but increases the efficiency of searches.

Deleting an item from a binary tree

Firstly, we need to find the node to delete in the structure.

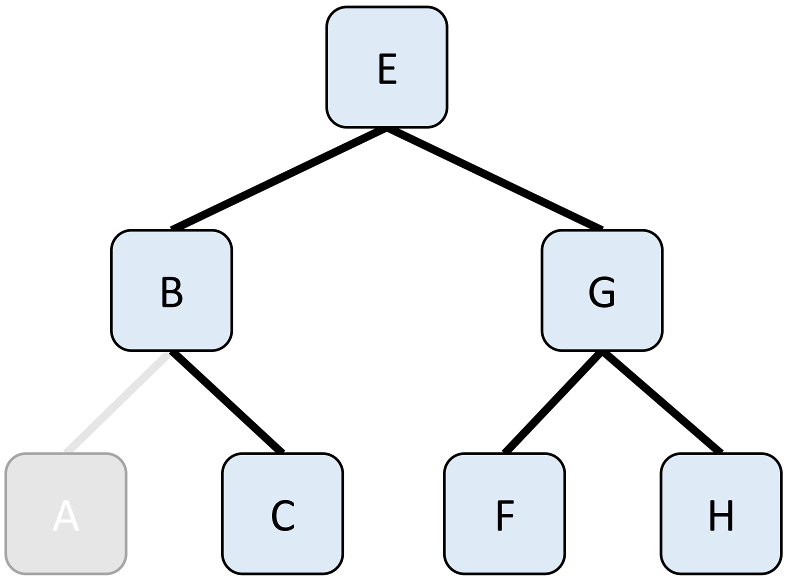
1. Start at the root node.
2. While the current node exists, and it is not the one to be deleted:
   a. Set the previous node to be the current node.
   b. If the item to be deleted is less than the current node:
      i. Follow the left pointer.
      ii. Set the node found to be the current node.
   c. If the item to be deleted is greater than the current node:
      i. Follow the right pointer.
      ii. Set the node found to be the current node.

Assuming the node to be deleted exists in the binary tree, we can now proceed to delete it. As with all algorithms, there are many ways to implement the deletion of a node. The Thomas Hibbard algorithm is one approach. There are three possibilities that need to be considered when deleting a node from a binary tree using this approach, each with a different operation:

i. The node is a leaf node and has no children. The node can be removed from the tree.
ii. The node has one child. Copy the child node to the node to be deleted and remove the child.
iii. The node has two children. The in-order successor node is copied to the node to be deleted and the in-order successor is deleted. The successor node is the leftmost/smallest node in the right sub-tree of the node to be deleted.

The node to be deleted has no children

Here, A is being deleted. The previous node's left pointer is set to null because A is to the left of B. If C were to be removed, the previous node's right pointer would be set to null.



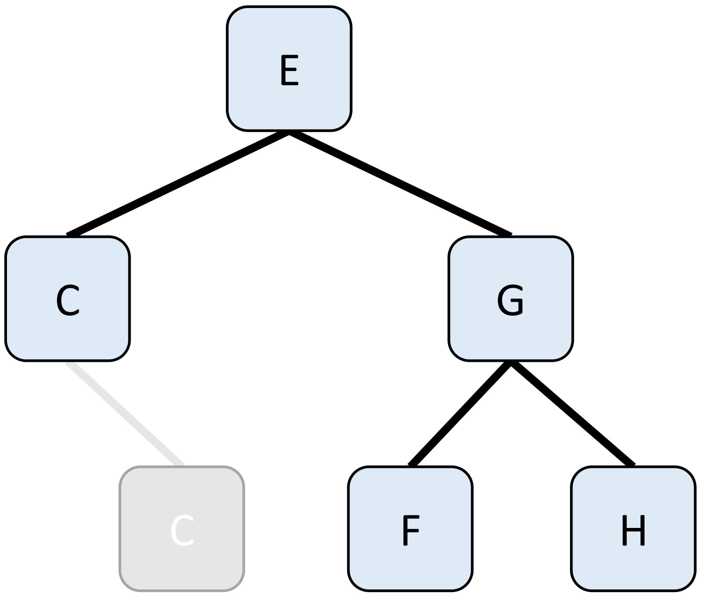
3. If the previous node is greater than the current node, the previous node's left pointer is set to null.
4. If the previous node is less than the current node, the previous node's right pointer is set to null.

The node to be deleted has one child

Here, B is being deleted. The child node C and its pointers are copied to the node previously occupied by B and the child node is removed.

This operation can also be accomplished by changing node E's left pointer to be the left pointer of the node being deleted. In this case, E's left pointer becomes C.

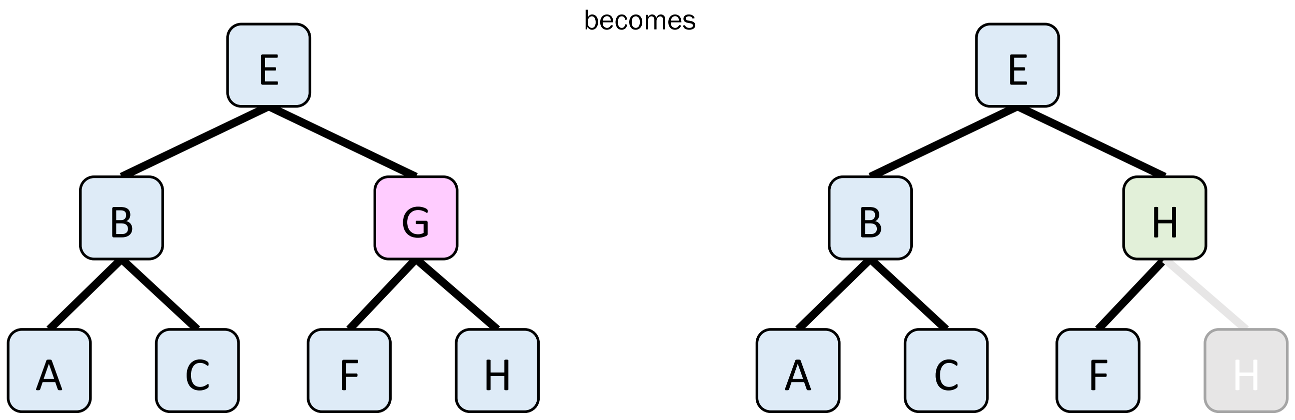
In either case, E now points to node C instead of node B.



5. If the current node is less than the previous node:
    a. Set the previous node's left pointer to the current node's left child.
6. If the current node is greater than the previous node:
    b. Set the previous node's right pointer to the current node's right child.

The node to be deleted has two children but no left sub-tree on the right pointer

Here, G is being deleted. In this case, there is no left sub-tree from H, so G is replaced with H and its right pointer is set to null.



H is the successor node and is promoted to the position previously occupied by G. Note that H was not chosen simply because it is the right child of G but because it is the leftmost node on the right sub-tree from G. If there were further left nodes to follow, they would be followed to the final leaf node, the successor node.

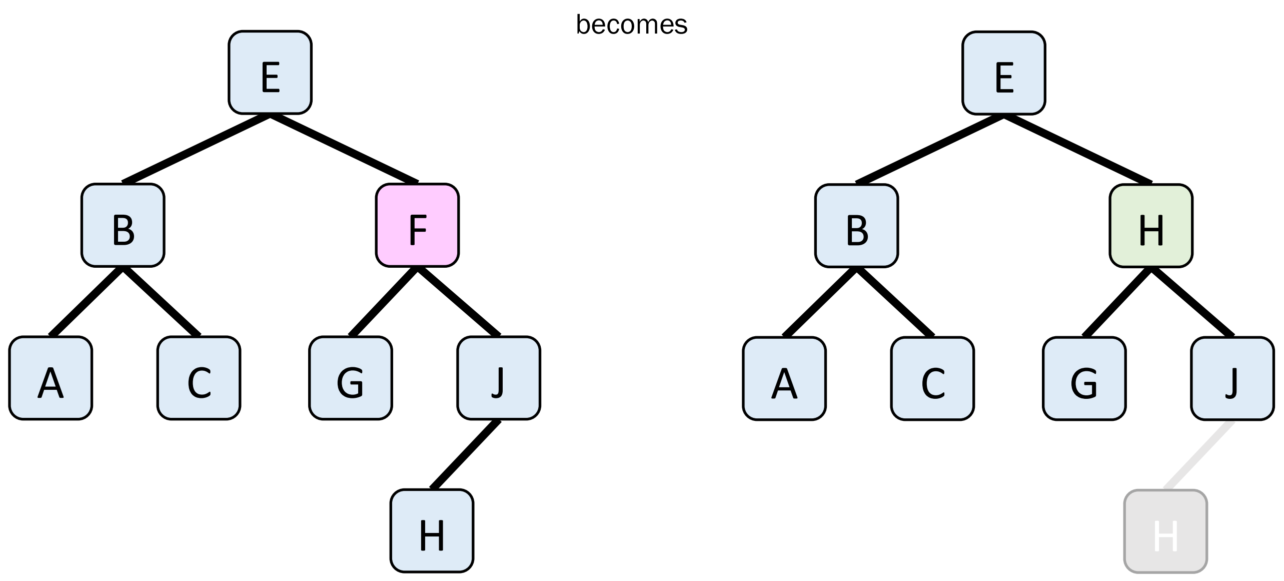
7. If a right node exists but has no left sub-tree:
    a. Set the current node to be the current node's right pointer.
    b. Set the current node's right pointer to be null.

Did you know?

In the example above, you could also replace H with F and delete node F instead.

The node to be deleted has two children and a left sub-tree on the right pointer

Here, F is being deleted. The successor node is H. F is replaced with H, and J's left pointer is set to null. However, it is a mistake to assume that it is always the first left node from the right sub-tree that is found. Instead, it is the leftmost leaf node that is swapped. Therefore, if H had a left pointer, it would be followed until the leftmost leaf node was found.



8. If a right node exists and has a left sub-tree, find the smallest leaf node in the right sub-tree.
    a. Change the current node to be the value of the smallest leaf node.
    b. Remove the leaf node.

Alternative approaches to deleting a node

One alternative approach would be to use the predecessor node (the rightmost node in the left sub-tree) instead of the successor node with the Hibbard algorithm.

Another simple approach is for each node to have an additional Boolean field that is set to true if the node exists and false if it has been deleted. When the tree is traversed, if the node has been deleted and the Boolean flag is false, the node is not output. When a new node is added to the binary tree, the deleted node is still considered to ensure the new node is added to the correct branch. With this approach, the binary tree will continue to grow over time and contain many redundant nodes. Periodically, a rebalance operation would be performed, skipping over deleted nodes and effectively deleting them forever.

Pseudocode for deleting an item from a binary tree

```
current_node = root node
while current_node != null and current_node != item
    previous_node = current_node
    if item < current_node.data Then
        current_node = current_node.left_pointer
    Else
        current_node = current_node.right_pointer
    End If
If current_node != null then
    If current_node.left_pointer == null and current_node.right_pointer == null Then
        If previous_node.data > current_node.data Then
            previous_node.left_pointer = null
        Else
             previous_node.right_pointer = null
        End If
    Elseif current_node.right_pointer == null Then
        If previous_node.data > current_node.data Then
            previous_node.left_pointer = current_node.left_pointer
        Else
            previous_node.right_pointer = current_node.left_pointer
        End If
    Elseif current_node.left_pointer == null Then
        If previous_node.data < current_node.data Then
            previous_node.left_pointer = current_node.right_pointer
        Else
            previous_node.right_pointer = current_node.right_pointer
        End If
    Else
        right_node = current_node.right_pointer
        If right_node.left_pointer != null Then
            smallest_node = right_node
            While smallest_node.left_pointer != null
                previous_node = smallest_node
                smallest_node = smallest_node.left_pointer
            End While
            current_node.data = smallest_node.data
            previous_node.left_pointer = null
        Else
            current_node.data = right_node.data
            current_node.right_pointer = null
        End If
    End If
End If
```

Traversal operations

Finding, updating or outputting the nodes from a binary tree requires a traversal algorithm. These algorithms visit each node in turn once, starting from the root node, until the desired node is found. Since there are many potential paths to follow in a binary tree, there are many ways to traverse the structure too, including depth-first traversal (pre-, in- and post-order) or breadth-first traversal.

Pre-order traversal (depth-first traversal)

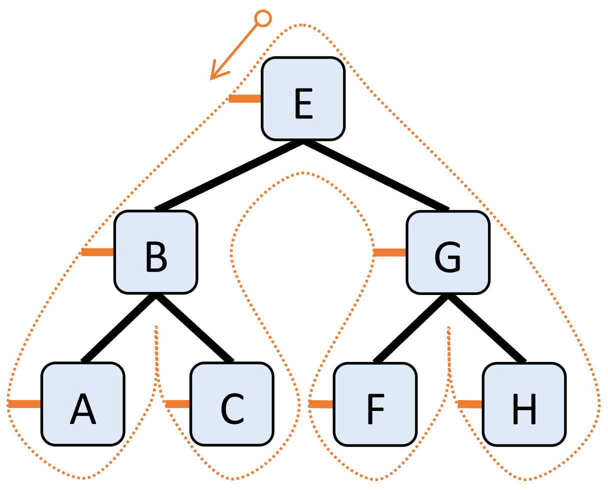
Pre-order traversal – a type of a depth-first traversal – is used to create a copy of a binary tree or return prefix expressions in Polish notation, which can be used by programming language interpreters to evaluate syntax.

The algorithm can be described as *node-left-right*:

1. Start at the root node.
2. Output the node.
3. Follow the left pointer and repeat from step 2 recursively until there is no pointer to follow.
4. Follow the right pointer and repeat from step 2 recursively until there is no pointer to follow.

Pre-order traversal illustrated

A pre-order traversal can be pictured like this:



Note the markers on the left side of each node. As you traverse the tree, starting from the root, the nodes are only output when the marker is passed: E, B, A, C, G, F, H. You can illustrate like this in exams to demonstrate your understanding of the algorithm.

Pseudocode for a pre-order traversal

```
Procedure preorder(current_node)
      If current_node != null Then
            Print(current_node.data)
            If current_node.left_pointer != null Then
                  preorder(current_node.left_pointer)
            End If
            If current_node.right_pointer != null Then
                  preorder(current_node.right_pointer)
            End If
      End If
End procedure
```

In-order traversal (depth-first traversal)

An in-order traversal – a type of a depth-first traversal – is used to output the contents of the binary tree in order. One of the significant advantages of the binary search tree is that it automatically sorts the contents of the structure without moving data, irrespective of the order in which the data arrived.

The algorithm can be described as *left-node-right*:

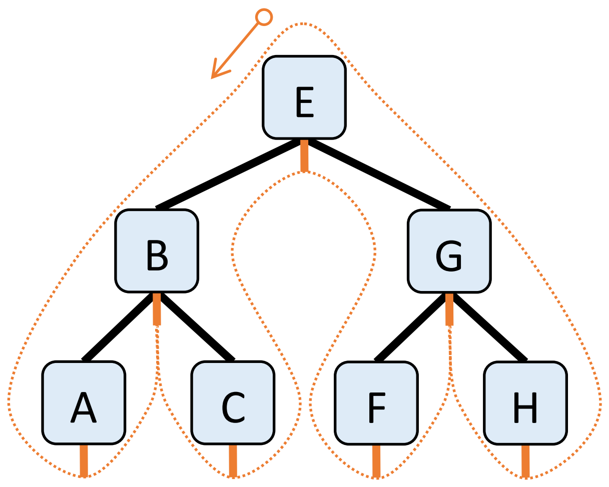
1. Start at the root node.
2. Follow the left pointer and repeat from step 2 recursively until there is no pointer to follow.
3. Output the node.
4. Follow the right pointer and repeat from step 2 recursively until there is no pointer to follow.

Did you know?

Don't forget, all traversals including pre-order, in-order and post-order always follow the left path first – unless you want to output the items in reverse order.

In-order traversal illustrated

An in-order traversal can be pictured like this:



Note the markers on the bottom of each node. As you traverse the tree, starting from the root, the nodes are only output when the marker is passed: A, B, C, E, F, G, H. You can illustrate like this in exams to demonstrate your understanding of the algorithm.

To output the nodes in reverse order, simply reverse the algorithm by following the right pointers before outputting the node and then follow the left pointers.

Pseudocode for an in-order traversal

```
Procedure inorder(current_node)
      If current_node != null Then
            If current_node.left_pointer != null Then
                  inorder(current_node.left_pointer)
            End If
            Print(current_node.data)
            If current_node.right_pointer != null Then
                  inorder(current_node.right_pointer)
            End If
      End If
End procedure
```

Post-order traversal (depth-first traversal)

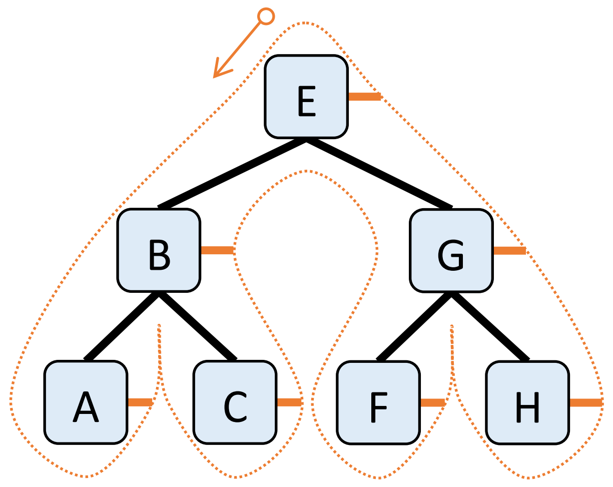
A post-order traversal – a type of a depth-first traversal – is used to delete a binary tree or output post-fix expressions that can be used to evaluate mathematical expressions without brackets. This is how arithmetic logic units work in stack-machine computers, and it was popular in pocket calculators until the early 2010s.

The algorithm can be described as *left-right-node*:

1. Start at the root node.
2. Follow the left pointer and repeat from step 2 recursively until there is no pointer to follow.
3. Follow the right pointer and repeat from step 2 recursively until there is no pointer to follow.
4. Output the node.

Post-order traversal illustrated

A post-order traversal can be pictured like this:



Note the markers on the right side of each node. As you traverse the tree, starting from the root, the nodes are only output when the marker is passed: A, C, B, F, H, G, E. You can illustrate like this in exams to demonstrate your understanding of the algorithm.

Pseudocode for a post-order traversal

```
Procedure postorder(current_node)
    If current_node != null Then
        If current_node.left_pointer != null Then
            postorder(current_node.left_pointer)
        End If
        If current_node.right_pointer != null Then
            postorder(current_node.right_pointer)
        End If
        Print(current_node.data)
    End If
End procedure
```

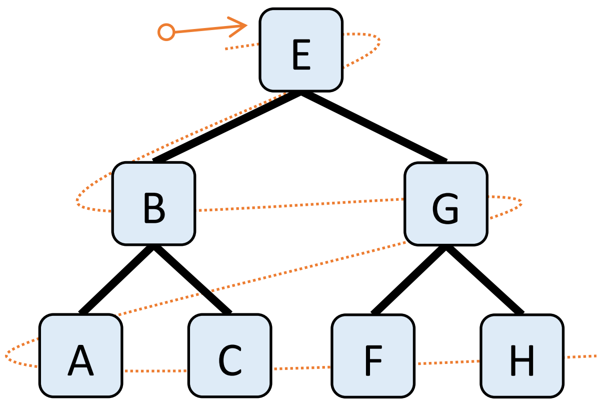
Breadth-first traversal using a binary tree

Visiting each node on the same level of a tree before going deeper is an example of a breadth-first traversal. Although commonly associated with graphs, a binary tree can also be traversed in this way – however, it does require a queue structure.

1. Start at the root node.
2. While the current node exists:
    a. Output the current node.
    b. If the current node has a left child, enqueue the left node.
    c. If the current node has a right child, enqueue the right node.
    d. Dequeue and set the current node to the dequeued node.

Breadth-first traversal on a binary tree illustrated

A breadth-first traversal can be pictured like this:



Pseudocode for a breadth-first traversal on a binary tree

```
current_node = root_node
While current_node != null
       Print(current_node)
       If current_node.left_pointer Then enqueue(current_node.left_pointer)
       If current_node.right_pointer Then enqueue(current_node.right_pointer)
       current_node = dequeue()
```

Binary tree coded in Python using an array

```python
class BinaryTree:

    depth = 5
    max = 2**(depth + 1) - 1
    btree = ["" for item in range(max)]
    root = 0

    def add(self, item):
        current_node = self.root
        # Find correct position
        while current_node < self.max and self.btree[current_node] != "":
            if item < self.btree[current_node]:
                current_node = (2 * current_node) + 1
            else:
                current_node = (2 * current_node) + 2
        # Check overflow
        if current_node < self.max:
            self.btree[current_node] = item
            return True
        else:
            return False

    def delete(self, item):
        # Using Hibbard's algorithm (leftmost node of right sub-tree is the successor)
        # Find the node to delete
        current_node = self.root
        while current_node < self.max and self.btree[current_node] != item:
            if item < self.btree[current_node]:
                current_node = (2 * current_node) + 1
            else:
                current_node = (2 * current_node) + 2
        if current_node < self.max and self.btree[current_node] == item:
            # Handle 3 cases depending on the number of child nodes
            left_node = (2 * current_node) + 1
            right_node = (2 * current_node) + 2
            if left_node < self.max and self.btree[left_node] == "" and right_node <
self.max and self.btree[right_node] == "":
                # Node has no children
                self.btree[current_node] = ""
            elif left_node < self.max and self.btree[left_node] != "" and right_node <
self.max and self.btree[right_node] != "":
                # Node has two children
                # Find the smallest value in the right sub-tree (successor node)
                smallest = right_node
                while (2 * smallest) + 1 < self.max and self.btree[(2 * smallest) + 1] !=
"":
```

DATA STRUCTURES

```
                smallest = (2 * smallest) + 1
            self.btree[current_node] = self.btree[smallest]
            self.btree[smallest] = ""
        elif left_node < self.max and self.btree[left_node] != "":
            # Node has one left child
            self.btree[current_node] = self.btree[left_node]
            self.btree[left_node] = ""
        elif right_node < self.max and self.btree[right_node] != "":
            # Node has one right child
            self.btree[current_node] = self.btree[right_node]
            self.btree[right_node] = ""
        return True
    else:
        return False

def preorder(self, current_node):
    # Visit each node: NLR
    if current_node < self.max and self.btree[current_node] != "":
        print(self.btree[current_node])
        self.preorder((2 * current_node) + 1)
        self.preorder((2 * current_node) + 2)

def inorder(self, current_node):
    # Visit each node: LNR
    if current_node < self.max and self.btree[current_node] != "":
        self.inorder((2 * current_node) + 1)
        print(self.btree[current_node])
        self.inorder((2 * current_node) + 2)

def postorder(self, current_node):
    # Visit each node: LRN
    if current_node < self.max and self.btree[current_node] != "":
        self.postorder((2 * current_node) + 1)
        self.postorder((2 * current_node) + 2)
        print(self.btree[current_node])

def bft(self):
    # Visit each node: BFT
    for current_node in range(self.max):
        if self.btree[current_node] != "":
            print(self.btree[current_node])
```

```
# Main program starts here
items = ["E", "B", "G", "A", "C", "F", "H"]
binary_tree = BinaryTree()
for index in range(0, len(items)):
    binary_tree.add(items[index])
# Traverse the binary tree
print("Breadth first traversal:")
binary_tree.bft()
print("Pre-order traversal:")
binary_tree.preorder(binary_tree.root)
print("In-order traversal:")
binary_tree.inorder(binary_tree.root)
print("Post-order traversal:")
binary_tree.postorder(binary_tree.root)
```

Binary tree coded in Python using objects

A queue data structure has been included with this implementation to facilitate the breadth-first traversal.

```python
class Queue:

    class Node:
        data = None
        pointer = None

    front_pointer = None
    back_pointer = None

    def enqueue(self, item):
        # Check queue overflow
        try:
            # Push the item
            new_node = Queue.Node()
            new_node.data = item
            # Empty queue
            if self.back_pointer == None:
                self.front_pointer = new_node
            else:
                self.back_pointer.pointer = new_node
            self.back_pointer = new_node
            return True
        except:
            return False

    def dequeue(self):
        # Check queue underflow
        if self.front_pointer != None:
            # Dequeue the item
            popped = self.front_pointer.data
            self.front_pointer = self.front_pointer.pointer
            # When the last item is dequeued reset the pointers
            if self.front_pointer == None:
                self.back_pointer = None
            return popped
        else:
            return None

    def peek(self):
        # Check queue underflow
        if self.front_pointer != None:
            # Peek the item
            return self.front_pointer.data
        else:
            return None


class Binary_tree:
    class Node:
        data = None
        left_pointer = None
        right_pointer = None

    root = None
```

```
def add(self, item):
    # Check memory overflow
    try:
        new_node = Binary_tree.Node()
        new_node.data = item
        current_node = self.root
        new_node.left_pointer = None
        new_node.right_pointer = None
        # Tree is empty
        if current_node == None:
            self.root = new_node
        else:
            # Find correct position in the tree
            while current_node != None:
                previous = current_node
                if item < current_node.data:
                    current_node = current_node.left_pointer
                else:
                    current_node = current_node.right_pointer
            if item < previous.data:
                previous.left_pointer = new_node
            else:
                previous.right_pointer = new_node
            return True
    except:
        return False

def delete(self, item):
    # Using Hibbard's algorithm (leftmost node of right sub-tree is the successor)
    # Find the node to delete
    current_node = self.root
    while current_node != None and current_node.data != item:
        previous = current_node
        if item < current_node.data:
            current_node = current_node.left_pointer
        else:
            current_node = current_node.right_pointer

    # Handle 3 cases depending on the number of child nodes
    if current_node != None:
        if current_node.left_pointer == None and current_node.right_pointer == None:
            # Node has no children
            if previous.data > current_node.data:
                previous.left_pointer = None
            else:
                previous.right_pointer = None
        elif current_node.right_pointer == None:
            # Node has one left child
            if previous.data > current_node.data:
                previous.left_pointer = current_node.left_pointer
            else:
                previous.right_pointer = current_node.left_pointer
        elif current_node.left_pointer == None:
            # Node has one right child
            if previous.data < current_node.data:
                previous.left_pointer = current_node.right_pointer
            else:
                previous.right_pointer = current_node.right_pointer
        else:
            # Node has two children
            right_node = current_node.right_pointer
            if right_node.left_pointer != None:
                # Find the smallest value in the right sub-tree (successor node)
                smallest = right_node
                while smallest.left_pointer != None:
                    previous = smallest
                    smallest = smallest.left_pointer
                # Change the deleted node value to the smallest value
                current_node.data = smallest.data
                # Remove the successor node
                previous.left_pointer = None
            else:
                # Handle special case of no left sub-tree from right node
                current_node.data = right_node.data
                current_node.right_pointer = None

    def preorder(self, current_node):
        if current_node != None:
            # Visit each node: NLR
            print(current_node.data)
            if current_node.left_pointer != None:
                self.preorder(current_node.left_pointer)
            if current_node.right_pointer != None:
                self.preorder(current_node.right_pointer)

    def inorder(self, current_node):
        if current_node != None:
            # Visit each node: LNR
            if current_node.left_pointer != None:
                self.inorder(current_node.left_pointer)
            print(current_node.data)
            if current_node.right_pointer != None:
                self.inorder(current_node.right_pointer)

    def postorder(self, current_node):
        if current_node != None:
            # Visit each node: LRN
            if current_node.left_pointer != None:
                self.postorder(current_node.left_pointer)
            if current_node.right_pointer != None:
                self.postorder(current_node.right_pointer)
            print(current_node.data)

    def bft(self, current_node):
        q = Queue()
        # Visit each node: BFT
        while current_node != None:
            print(current_node.data)
            if current_node.left_pointer != None:
                q.enqueue(current_node.left_pointer)
            if current_node.right_pointer != None:
                q.enqueue(current_node.right_pointer)
            current_node = q.dequeue()
```

```
# Main program starts here
items = ["E", "B", "G", "A", "C", "F", "H"]
# Create binary tree
binary_tree = Binary_tree()
for index in range(0, len(items)):
    binary_tree.add(items[index])
# Traverse the binary tree
print("Breadth first traversal:")
binary_tree.bft(binary_tree.root)
print("Pre-order traversal:")
binary_tree.preorder(binary_tree.root)
print("In-order traversal:")
binary_tree.inorder(binary_tree.root)
print("Post-order traversal:")
binary_tree.postorder(binary_tree.root)
```

Operations

Adding items:	binary_tree.add(*item*)
Deleting items:	binary_treet.delete(*item*)
Outputting items:	binary_tree.bft()
	binary_tree.preorder(binary_tree.root)
	bt.inorder(binary_tree.root)
	bt.postorder(binary_tree.root)

Did you know?

Breadth- and depth-first traversals are also called breadth- and depth-first searches if the algorithm is used to find a particular node and then stop before visiting every node.

Breadth- and depth-first traversals or searches can be used with any type of graph or tree structure, but a binary search can only be performed on a binary search tree.

Efficiency of operations on a binary tree

	Time complexity		
	Best case	Average case	Worst case
Access a node, binary search	O(1) Constant	O(log n) Logarithmic	O(n) Linear
Add node	O(log n) Logarithmic	O(log n) Logarithmic	O(n) Linear
Delete node	O(log n) Logarithmic	O(log n) Logarithmic	O(n) Linear
Rebalance	O(n) Linear	O(n) Linear	O(n) Linear
Traversal	O(n) Linear	O(n) Linear	O(n) Linear

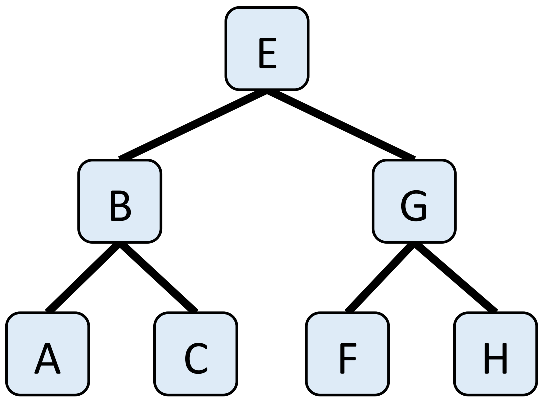
Space complexity	
Array implementation	Object implementation
O(1) Constant	O(n) Linear

If the binary tree is implemented with an array, the memory footprint will be fixed and constant, but the size of the structure will be limited and there will likely be a lot of unused space. An implementation using objects allows for dynamic memory allocation and for the size of the structure to grow in a linear fashion.

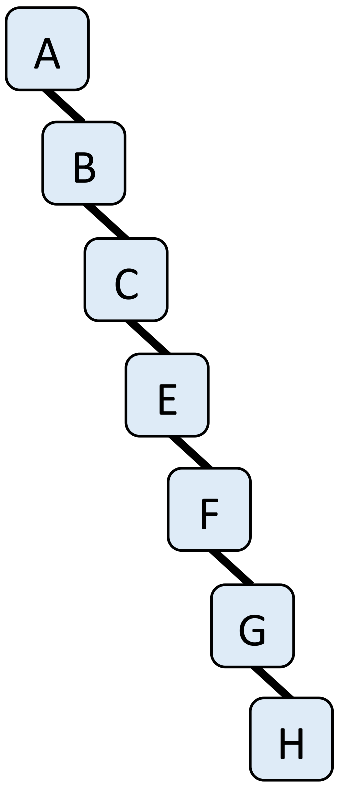
Assuming the binary tree is not pre-populated, establishing the tree from a list of data items will require each item to be added sequentially. However, the input order of the items does not matter – therefore, the number of operations depends on the number of data items, O(n).

Adding, deleting and searching nodes on a binary tree uses a technique called divide and conquer. With a balanced tree, the number of items to consider is halved each time a node is visited, providing logarithmic complexity, O(log n). If the tree is unbalanced, it becomes the same as a linear search to determine the location of an item to add or delete, demoting it to linear complexity, O(n).

Balanced tree where each level is complete:



Unbalanced tree holding the same data:



In the best-case scenario, the node to be found is the root node, so it will always be found first, O(1). Notice in the example above how the worst case becomes a linear search for item H.

All traversal algorithms require each node to be visited. As the number of nodes in the tree increases, so does the execution time, so traversals are of linear complexity, O(n).

Did you know?

Binary trees are often used to implement dictionary data structures. As a dictionary does not permit duplicate keys, it is often thought that binary trees do not allow duplicate nodes – this is not true. A binary tree can hold duplicate values as child nodes, either to the left or right of the node with the same value.

Dictionary

A dictionary is used for storing related data and can be static or dynamic. Each item in a dictionary has a key and value, known as a key-value pair. The key must be unique for all data stored in the dictionary. For example, the registration number of a car is unique because no two vehicles can have the same registration, so it makes an ideal key. The value would be associated data about the car – for example:

- Key: GB21 CWW
- Value: Vauxhall Corsa

A dictionary is an implementation of a hash table search using an array or list and is often called an associative array or hash map. Many high-level languages provide the dictionary as a primitive data structure for the programmer to use in their programs. The value in the key-value pair can be any another data structure – commonly a list – making the dictionary highly versatile.

Applications of a dictionary

A dictionary is used to hold data related to a unique key field. NoSQL – an acronym for *not only SQL* – makes extensive use of dictionaries to store databases. Dictionaries are also ideal for storing a graph as part of the implementation of depth- and breadth-first traversal algorithms. A dictionary is useful in any situation where an efficient value look-up is required.

Storing a dictionary in memory

The implementation of a dictionary is often abstracted from the programmer, who can use operations to add, remove and change key-value pairs without knowing how the dictionary works in memory. However, if the dictionary is not supported by your chosen programming language, you would use an array to implement your own dictionary structure, with the index being determined by the key.

Operations on a dictionary

Typical operations that can be performed on a dictionary include:

- Add key/value: Adding a new key-value pair to the dictionary.
- Delete key/value: Removing a key-value pair from the dictionary.
- Modify key/value: Edit a key-value pair.
- Lookup: Return the value from a key.
- Rehashing: Optimise the data structure to reduce collisions.

Array/list implementation of a dictionary

A key is changed into a number using a hashing function (also known as a hashing algorithm or hash). The number returned from the function is an index in an array or list where the value can be found.

Consider the following dictionary:

```
dictionary = {
  "E": ["B", "G"],
  "B": ["A", "C"],
  "G": ["F", "H"],
  "A": [],
  "C": [],
  "F": [],
  "H": []
}
```

Examining the first part of the structure, `"E": ["B", "G"]`, the letter E is a key and `["B", "G"]` is the value, a list holding more keys. Since the dictionary needs to be stored as an array, the key must be associated with an index in the array. Indexes are always integers, so the hashing function converts the character E into a useful integer.

Each letter can easily be converted into a number using its ASCII value. The number returned will be 65 for the letter A, so by subtracting 65, A could be stored in the first index in the array. This is a simplistic hashing algorithm because the key in this example is a single alphanumeric character. When the key is a string or large integer, a more complex algorithm will be required to determine a suitable index for each key.

		Array	
Key	Hashing function	Index	Element
A	65 – 65 =	0	[]
B	66 – 65 =	1	["A","C"]
C	67 – 65 =	2	[]
		3	
E	69 – 65 =	4	["B","G"]
F	70 – 65 =	5	[]
G	71 – 65 =	6	["F","H"]

Notice how the array may not be fully utilised, with some indexes never being generated by the hashing function. If it is possible to calculate the total number of items to be stored in advance, modulo can be used as part of the hashing function to ensure values that would result in indexes that fall out of the bounds of the array are not returned.

The data in the dictionary looks remarkably like the binary tree from the previous chapter, albeit the indexes are being used differently – this is an example of how one data structure can be implemented using another data structure. All data structures are simply memory locations that can be accessed with their address.

Adding an item to a dictionary

To add an item to a dictionary, the hashing function is applied to the item's key, which returns an index where the item can be stored in the array. However, the hashing function may return a value that is already occupied by a data item. There are several potential resolutions to this. See the *Hash table search* section of the *Searching algorithms* chapter for a detailed description of the essential properties of hashing functions and collision resolutions.

Deleting an item from a dictionary

When deleting an item from a dictionary, the hashing function will be applied to the key to determine the index of the element to delete. It may be that the item at that index is not the one to be deleted because the hashing function does not always return unique values. The solution to this problem is also described in the *Hash table search* section of the chapter on *Searching algorithms*.

Assuming the item to be deleted is found at the index returned by the hashing value, indexes are not removed – this is because the dictionary is stored as an array. Instead, the data element can be set to a null value or an empty string. Alternatively, an additional dimension in the array can be used to store a Boolean value that is set to true when an item is added and false when the item is deleted.

Did you know?

Programs will often need to store data in a file permanently or transmit it across a network. A process called serialisation, marshalling, flattening or pickling produces a plain-text or binary output of a data structure.

Looking up and item in a dictionary

1. Read a key.
2. Look up the key in the dictionary.
3. If the key exists, return the matching value; if not, return an error.

Pseudocode for looking up an item in a dictionary

```
key = input("Enter the key: ")
If key in dictionary Then
       Return dictionary[key].value
Else
       Return "Not found"
End If
```

Lookup operation coded in Python

```python
# Dictionary using primitive hash map
dictionary = {"England": "London", "France": "Paris", "Germany": "Berlin"}
key = input("Enter the key: ")
if key in dictionary:
    print(dictionary[key])
else:
    print("Not found")
```

Did you know?

Dictionary data structures can also be implemented using a self-balancing binary tree.

The advantage of this is that the hashing function traverses the tree to find the node, so collisions will not happen. Although finding an item may be slower on average, the time complexity of finding an item can also be reduced to O(log n) in the worst-case scenario because no collision resolution is necessary.

It is important that the binary tree is balanced to achieve this efficiency, so the structure of the tree must be updated regularly.

It is worth noting that there is no known order to the items in a dictionary due to the hashing function. In the example, even though “England” was entered into the dictionary first, it is not necessarily the first item in the structure. For this reason, it does not make sense to try to return an item from an index number. Some languages support such commands, but the returned value often appears to be a random item.

Efficiency of operations on a dictionary

	Time complexity		
	Best case	Average case	Worst case
Lookup, Modify	O(1) Constant	O(1) Constant	O(n) Linear
Add key/ value	O(1) Constant	O(1) Constant	O(n) Linear
Delete key/ value	O(1) Constant	O(n) Linear	O(n) Linear
Rehash	O(n) Linear	O(n) Linear	O(n) Linear

Space complexity	
Best case	Worst case
O(1) Constant	O(n) Linear

The dictionary data structure aims to be extremely efficient, returning a value without having to search for the key in the data set. Since the index of an item is determined by the hashing function, there is no order to the items in a dictionary and some indexes may not be used.

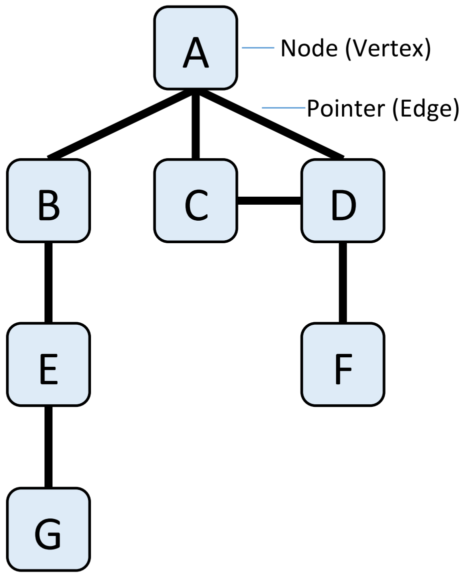
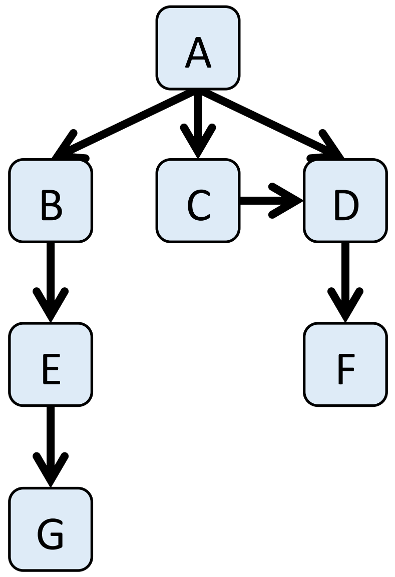
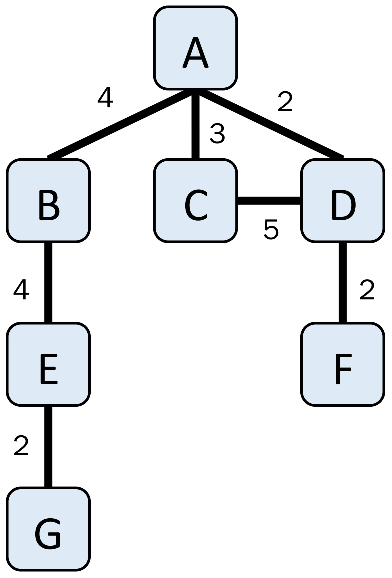
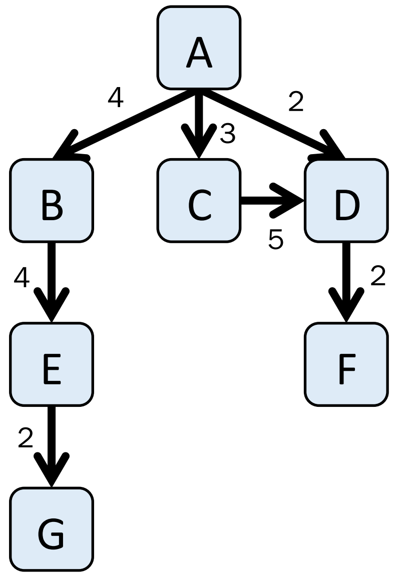
By using a key, items can still be randomly accessed. In most cases, with an appropriate hashing function, the key always delivers the index where the item is stored, O(1). New items are added instantly, and deleted items are replaced with null values. However, depending on the implementation, collisions are likely to occur and must be resolved, often degrading the lookup operation to a linear search, O(n). Rehashing to optimise the data structure is also a linear operation.

Dictionaries are often implemented using arrays as their underlying data structure. An array will be of a fixed size with an O(1) space complexity, although it may be resized, or stored as a binary tree O(n).

Graph

A graph is a dynamic structure of nodes (called vertices) and pointers (called edges). It is different from a binary tree because a root vertex is not required, and each vertex can have any number of edges. A graph is different from a regular tree because any vertex can point to any other vertex in the data structure, making loops between vertices possible.

The edges on a graph can either point in one direction, known as a directed graph, or in both directions, known as an undirected graph. Graphs can also be weighted, with each edge given a value or cost that represents a relationship between the vertices such as the distance between them.

Undirected graph	Directed graph
 A Node (Vertex) Pointer (Edge) B C D E F G	
Undirected weighted graph	**Directed weighted graph**
 A 4 3 2 B C 5 D 4 2 E F 2 G	 A 4 3 2 B C 5 D 4 2 E F 2 G

Applications of a graph

Graphs have many uses in computer science – for example, mapping road networks for navigation systems, storing flight paths for aircraft, social networking data such as friends lists, resource allocation in operating systems, representing molecular structures and geometry. They may also be used for storing knowledge graphs for expert systems, modelling business problems and pathfinding algorithms.

Storing a graph in memory

Typically, an undirected graph is stored as a dictionary. The value in the key/value pair being a list of connected vertices. This is known as an adjacency list. In Python, the code would be:

```
graph = {
  "A": ["B", "C", "D"],
  "B": ["A", "E"],
  "C": ["A", "D"],
  "D": ["A", "C", "F"],
  "E": ["B", "G"],
  "F": ["D"],
  "G": ["E"]
}
```

This may also be presented in pseudocode as:

```
edges {
(A, B), (A, C), (A, D), (B, A,), (B, E), (C, A), (C, D), (D, A), (D, C), (D, F), (E, B),
(E, G), (F, D), (G, E)
}
```

A directed graph stored as a dictionary:

```
graph = {
  "A": ["B", "C", "D"],
  "B": ["E"],
  "C": ["D"],
  "D": ["F"],
  "E": ["G"],
  "F": [],
  "G": []
}
```

In this example, note how vertex A is connected to C, but C is only connected to D and not back to A.

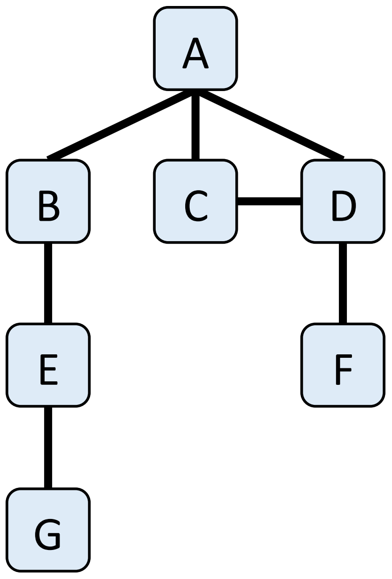
Costs on weighted graphs can be defined as:

```
graph = {
  "A": {"B": 4, "C": 3, "D": 2}
}
```

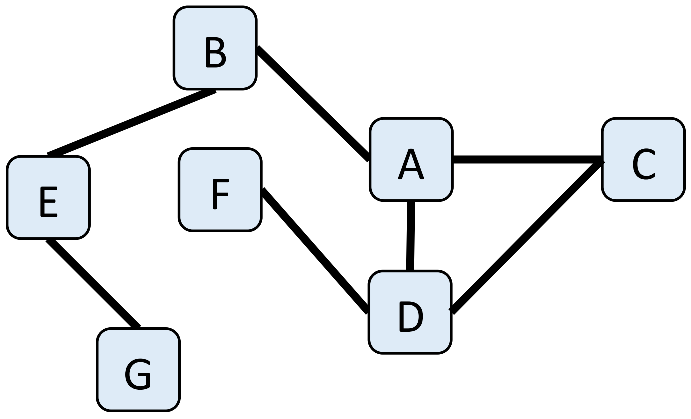
This means the cost between A and B is four, the cost between A and C is three, and the cost between A and D is two. Depending on what the data structure is being used for, costs between two vertices can represent many things including distance or bandwidth. Note how the graph becomes a dictionary of dictionaries when storing this additional data.

Abstraction of a graph data structure

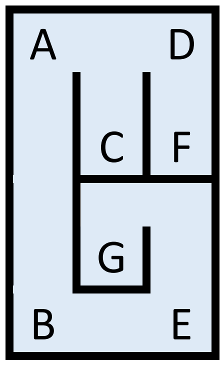
In the illustrations below, we can see how graph data can remain the same even though illustrations of it can look different. What is important is not what the graph looks like, but which vertices are connected. Using only the necessary detail and discarding unnecessary detail is known as abstraction.



is the same as



which is also the same as



An example of abstraction is the map of the London Underground. It bears no resemblance to where the stations are located; all that matters is which stations are connected in which order and on which line.

Similarly, it is important that students do not concern themselves with how a graph looks or what operation is being asked for in an examination– e.g., a pre-order traversal on a graph, which you might expect to only be relevant to a binary tree. Simply follow the algorithm on the structure provided.

Why use abstraction?

Abstraction can often help simplify a problem and make it easier for humans to communicate. Abstractions are especially useful to explain and illustrate complex relationships. Sometimes, links between seemingly unrelated problems can be seen with abstractions. Computers do not benefit from abstract illustrations.

Operations on a graph

It is worth noting that because graphs and trees are very similar – with a few notable exceptions – the operations on a binary tree such as traversal can also be performed on a graph data structure.

Typical operations that can be performed on a graph include:

- Adjacent: Returns whether there is an edge between two vertices.
- Neighbours: Returns all the vertices between two vertices.
- Add: Adds a new vertex to the graph.
- Remove: Removes a vertex from the graph.
- Add edge: Adds a new edge to a vertex from another vertex.
- Remove edge: Removes an edge between two vertices.
- Get vertex value: Returns the data stored in a vertex.
- Set vertex value: Sets the data for a vertex.
- Get edge value: Returns the value of an edge between two vertices.
- Set edge value: Sets the value of an edge between two vertices.
- Depth-first traversal/search: Traverses the graph, starting at the root vertex, visiting each vertex and exploring each branch as far as possible before backtracking.
- Breadth-first traversal/search: Traverses the graph, starting at the root vertex, visiting each neighbouring vertex before moving to the vertices at the next depth.
- Pre-order traversal: A type of depth-first traversal (see the *Binary tree* section for an example).
- In-order traversal: A type of depth-first traversal (see the *Binary tree* section for an example).
- Post-order traversal: A type of depth-first traversal (see the *Binary tree* section for an example).

Array implementation of a graph

While it is typical to store a graph as a dictionary, known as an adjacency list, it is also possible to store a graph using an array. This implementation is known as an adjacency matrix, with rows and columns representing vertices and the intersections representing edges.

An example of an adjacency matrix for an undirected graph is shown overleaf. The rows and columns are not usually labelled with the vertices with this approach because they would be indexes, but they are shown here for clarity.

With undirected graphs a 1 represents an edge existing between the two vertices, while a 0 represents no edge. With directed graphs the edge value is stored instead.

Undirected graph	Directed graph

Undirected graph

	A	B	C	D	E	F	G
A	0	1	1	1	0	0	0
B	1	0	0	0	1	0	0
C	1	0	0	1	0	0	0
D	1	0	1	0	0	1	0
E	0	1	0	0	0	0	1
F	0	0	0	1	0	0	0
G	0	0	0	0	1	0	0

Directed graph

	A	B	C	D	E	F	G
A	0	1	1	1	0	0	0
B	0	0	0	0	1	0	0
C	0	0	0	1	0	0	0
D	0	0	0	0	0	1	0
E	0	0	0	0	0	0	1
F	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0

Undirected weighted graph

	A	B	C	D	E	F	G
A	0	4	3	2	0	0	0
B	4	0	0	0	4	0	0
C	3	0	0	5	0	0	0
D	2	0	5	0	0	2	0
E	0	4	0	0	0	0	2
F	0	0	0	2	0	0	0
G	0	0	0	0	2	0	0

Directed weighted graph

	A	B	C	D	E	F	G
A	0	4	3	2	0	0	0
B	0	0	0	0	4	0	0
C	0	0	0	5	0	0	0
D	0	0	0	0	0	2	0
E	0	0	0	0	0	0	2
F	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0

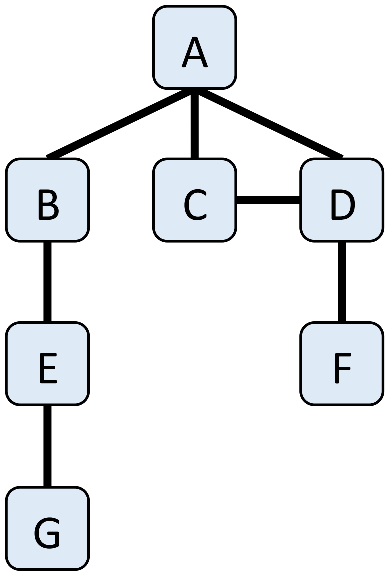
Notice an undirected graph is symmetrical and a directed graph is not.

An adjacency matrix can also be stored as an adjacency list:

```
graph ([[0,1,1,1,0,0,0],
        [1,0,0,0,1,0,0],
        [1,0,0,1,0,0,0],
        [1,0,1,0,0,1,0],
        [0,1,0,0,0,0,1],
        [0,0,0,1,0,0,0],
        [0,0,0,0,1,0,0]])
```

This can be abstracted to a table:

Nodes	Adjacency list
A	B; C; D
B	A; E
C	A; D
D	A; C; F
E	B; G
F	D
G	E



Object implementation of a graph

Another possibility for storing the graph in memory is for each vertex to be stored as an object. The object would have an attribute that is a list of edges – since they can be variable in number – with each edge being a pointer to the connected vertex.

Traversal operations

Finding, updating or outputting the vertices from a graph requires a traversal algorithm. These algorithms visit each vertex in turn until the desired vertex is found. Since there are many potential paths to follow in a graph, there are many ways to traverse the structure too including depth- and breadth-first traversals.

If only one occurrence of an item needs to be found, the traversal algorithm can stop without evaluating any other vertices. In this situation the traversal is often referred to as a depth and breadth-first search.

Breadth-first traversal on a graph

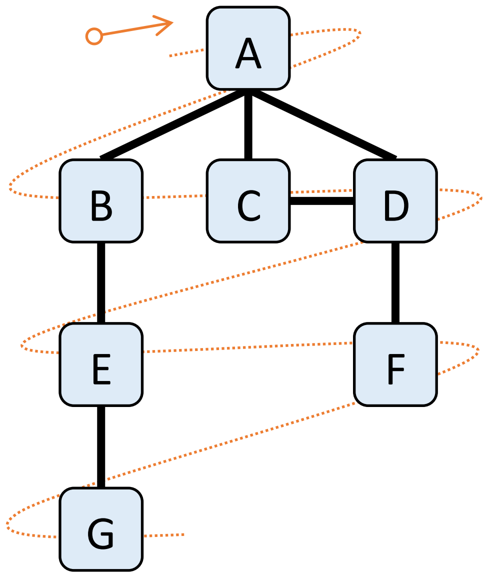
The breadth-first traversal is used to determine the shortest path between vertices on a graph. It is also used by search engines to index connected web pages using a crawler algorithm. The breadth-first search can also be used to find devices on peer-to-peer networks and to send broadcast frames or packets on a network too. GPS navigation systems, memory management using Cheney's algorithm and finding nearby people on a social network are all applications of a breadth-first traversal.

A breadth-first traversal requires the use of a queue data structure.

1. Set the root vertex as the current vertex.
2. Add the current vertex to the list of visited vertices if it is not already in the list.
3. For every edge connected to the vertex:
    a. If the linked vertex is not in the visited list:
        i. Enqueue the linked vertex.
        ii. Add the linked vertex to the visited list.
4. Dequeue and set the vertex removed as the current vertex.
5. Repeat from step 2 until the queue is empty.
6. Output all the visited vertices.

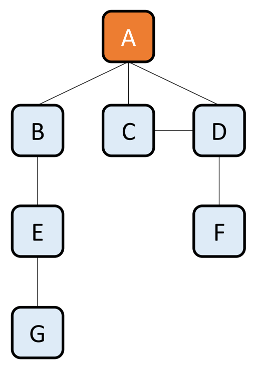
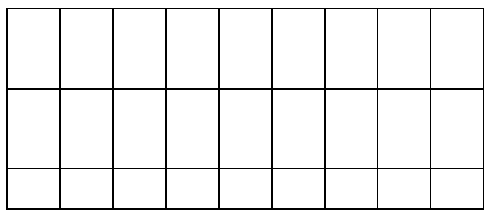
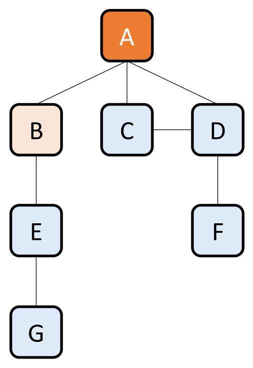
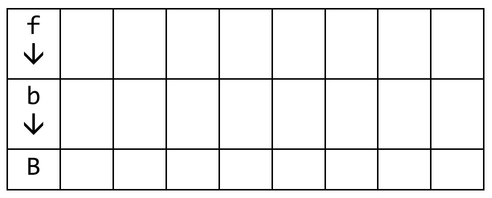
Breadth-first traversal illustrated

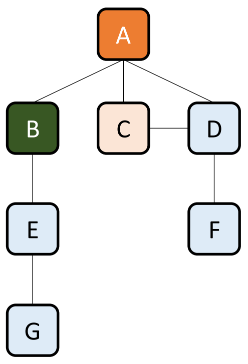
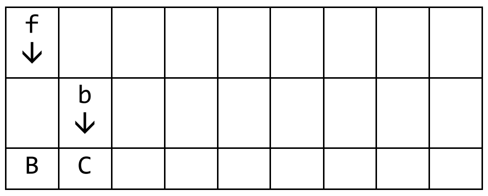
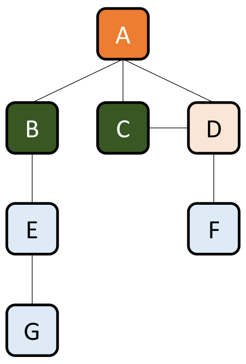
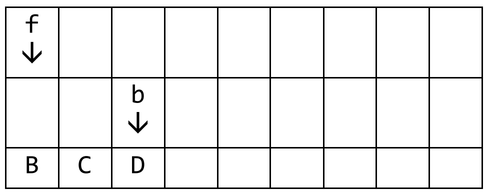
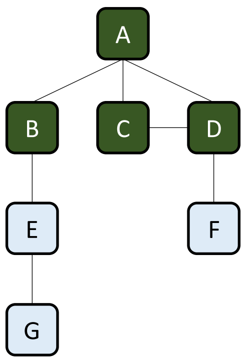
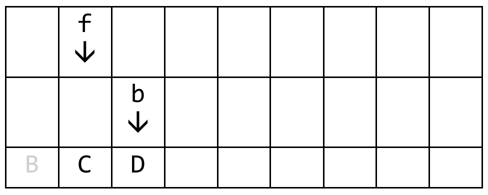
A breadth-first traversal can be pictured like this:



Stepping through the breadth-first traversal

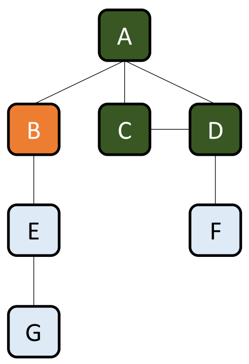
The front pointer in the queue is represented by the lowercase letter *f*. The back pointer in the queue is represented by the lowercase letter *b*.

	Graph	Queue	Visited
Step 1	Start at a root vertex – this can be any vertex, but A has been chosen. A is not in the visited list; add A to the list of visited vertices. Consider each edge of A.		
			A
Step 2	B is not in the visited list; enqueue B and add B to the visited list.		
			AB

Step 3	C is not in the visited list; enqueue C and add C to the visited list.		
			ABC
Step 4	D is not in the visited list; enqueue D and add D to the visited list.		
			ABCD
Step 5	All edges of A have been considered. Dequeue. New current vertex is B.		
			ABCD

Step 6

B is in the visited list. Consider each edge of B.

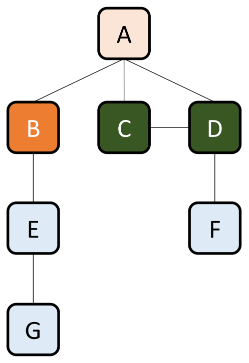


	f ↓							
		b ↓						
B	C	D						

ABCD

Step 7

A is in the visited list; ignore the vertex.

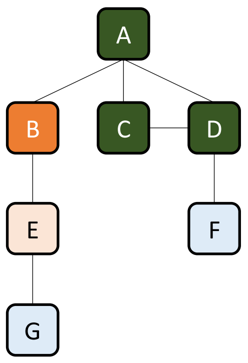


	f ↓							
		b ↓						
B	C	D						

ABCD

Step 8

E is not in the visited list; enqueue E and add E to the visited list.

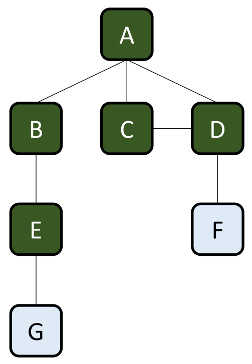


	f ↓							
			b ↓					
B	C	D	E					

ABCDE

Step 9

All edges of B have been considered. Dequeue. New current vertex is C.

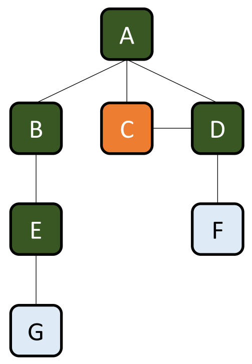


		f ↓						
			b ↓					
B	C	D	E					

ABCDE

Step 10

C is in the visited list. Consider each edge of C.

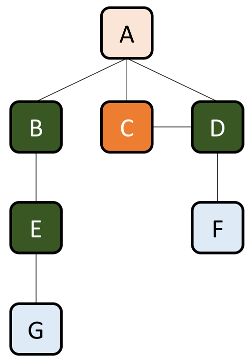


		f ↓						
			b ↓					
B	C	D	E					

ABCDE

Step 11

A is in the visited list; ignore the vertex.

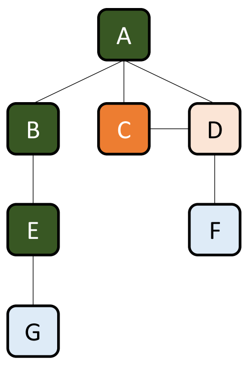


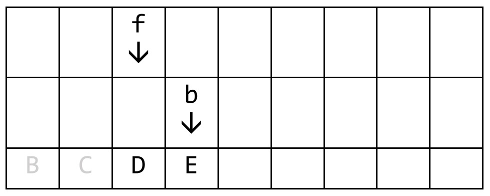
		f ↓						
			b ↓					
B	C	D	E					

ABCDE

Step 12

D is in the visited list; ignore the vertex.

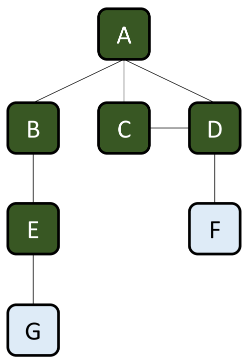


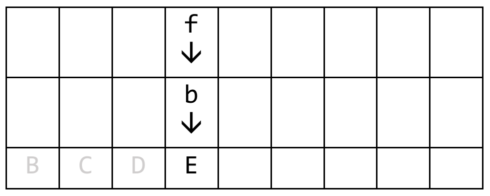


ABCDE

Step 13

All edges of C have been considered. Dequeue. New current vertex is D.

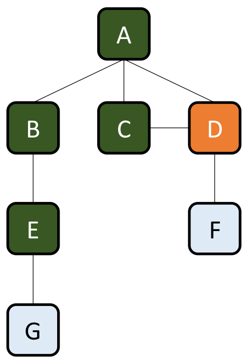


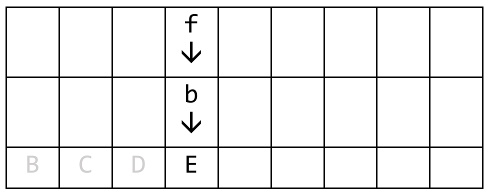


ABCDE

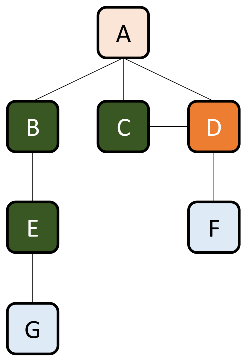
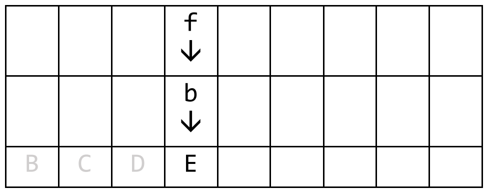
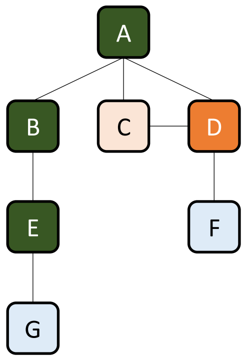
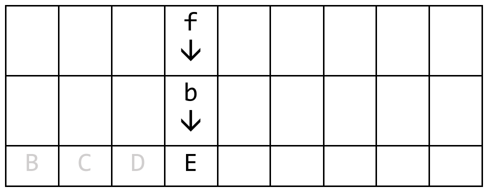
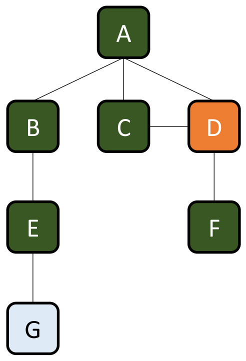
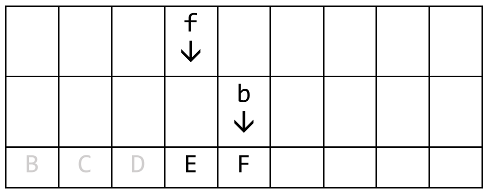
Step 14

D is in the visited list. Consider each edge of D.



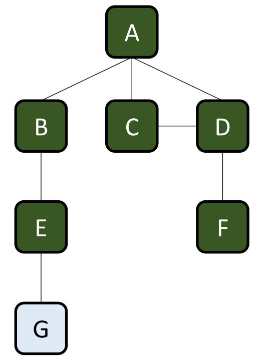


ABCDE

Step 15	A is in the visited list; ignore the vertex.		
			ABCDE
Step 16	C is in the visited list; ignore the vertex.		
			ABCDE
Step 17	F is not in the visited list; enqueue F and add F to the visited list.		
			ABCDEF

Step 18

All edges of D have been considered. Dequeue. New current vertex is E.

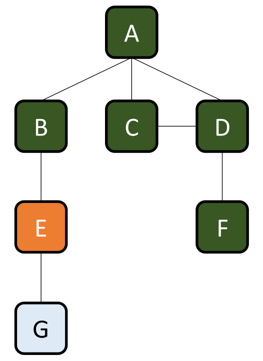


				f ↓				
				b ↓				
B	C	D	E	F				

ABCDEF

Step 19

E is in the visited list. Consider each edge of E.

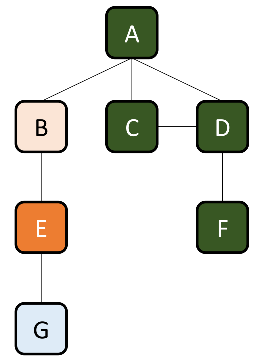


				f ↓				
				b ↓				
B	C	D	E	F				

ABCDEF

Step 20

B is in the visited list; ignore the vertex.

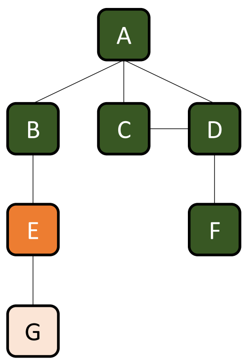


				f ↓				
				b ↓				
B	C	D	E	F				

ABCDEF

Step 21

G is not in the visited list; enqueue G and add G to the visited list.

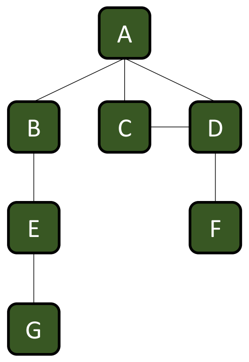


				f ↓				
					b ↓			
B	C	D	E	F	G			

ABCDEFG

Step 22

All edges of E have been considered. Dequeue. New current vertex is F.

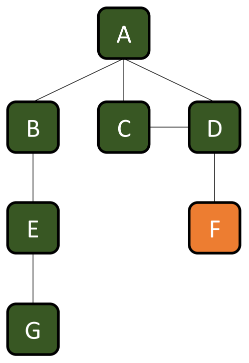


					f ↓			
					b ↓			
B	C	D	E	F	G			

ABCDEFG

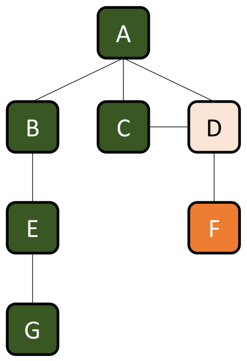
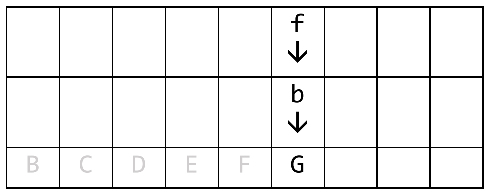
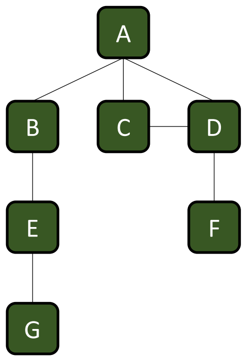
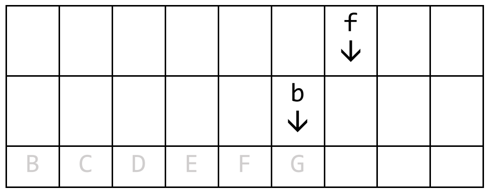
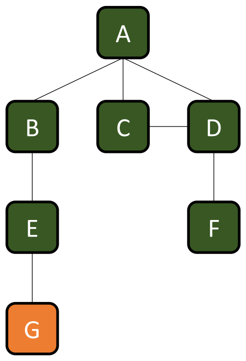
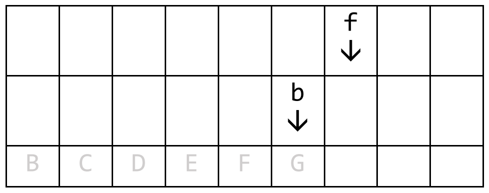
Step 23

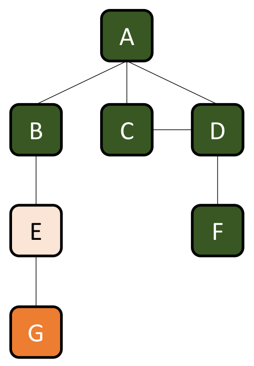
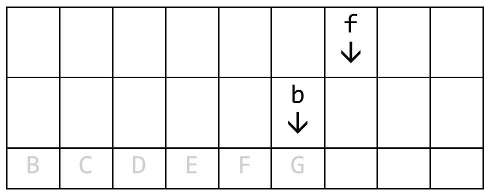
F is in the visited list. Consider each edge of F.



					f ↓			
					b ↓			
B	C	D	E	F	G			

ABCDEFG

Step 24	D is in the visited list; ignore the vertex.		
			ABCDEFG
Step 25	All edges of F have been considered. Dequeue. New current vertex is G.		
			ABCDEFG
Step 26	G is in the visited list. Consider each edge of G.		
			ABCDEFG

Step 27	E is in the visited list; ignore the vertex.		
			ABCDEFG
Step 28	All edges of G have been considered. Dequeue. The queue is empty – the algorithm is complete.		

Did you know?

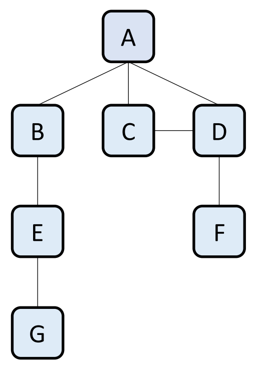
The breadth-first traversal was invented by Konrad Zuse, a pioneering computer scientist in 1945. He is best known for the creation of the world's first programmable computer, the Turing-complete Z3. Zuse is often regarded as the inventor of the modern computer.

Zuse worked entirely independently of other leading computer scientists of his day. Between 1936 and 1945 in near total isolation due to World War II.

Re-invented in 1959 by Edward Moore, the breadth-first traversal was used to find the shortest path out of a maze. The algorithm was later developed by Chester Lee into a wire routing algorithm for printed circuit boards.

Outputs from a breadth first traversal

It is worth noting that there is more than one valid output from a breadth-first traversal. This implementation examined the edges from A in the order B, C, D, from left to right. However, it would be perfectly valid to examine them from right to left. To achieve this, the edges would need to be enqueued in reverse order.

Illustrated graph	Valid output 1	Valid output 2
	A B C D E F G This is the most common output shown in examples of this algorithm. Therefore, it is the one you should illustrate in exams unless the question specifically states otherwise.	A D C B F E G Reverse traversal.

This algorithm demonstrates a full traversal of the graph. However, if the goal is to search the graph for a specific vertex, the algorithm can terminate when the vertex is found.

Also note that we did not enqueue the root vertex. There seems little point considering it would be dequeued immediately. However, many implementations of this algorithm will do that. We also checked if the current vertex was in the visited list – it always will be, so we could introduce a further optimisation here.

Pseudocode for a breadth-first traversal

```
current_vertex = root
While current_vertex != Nothing
    If not visited.Contains(current_vertex) Then
        Visited.Add(current_vertex)
    End If
    For each edge in vertex
        If not visited.Contains(edge.vertex) Then
            Queue.enqueue(edge.vertex)
            Visited.Add(edge.vertex)
        End If
    Next
    current_vertex = Queue.dequeue
End While
For each vertex in visited
    Output current_vertex
Next
```

Breadth-first traversal coded in Python

```python
# Queue data structure for breadth-first traversal
class Queue:
    class Node:
        data = None
        pointer = None

    front_pointer = None
    back_pointer = None

    def enqueue(self, item):
        # Check queue overflow
        try:
            # Enqueue the item
            new_node = Queue.Node()
            new_node.data = item
            # Empty queue
            if self.back_pointer == None:
                self.front_pointer = new_node
            else:
                self.back_pointer.pointer = new_node
            self.back_pointer = new_node
            return True
        except:
            return False

    def dequeue(self):
        # Check queue underflow
        if self.front_pointer != None:
            # Dequeue the item
            item = self.front_pointer.data
            self.front_pointer = self.front_pointer.pointer
            # When the last item is dequeued reset the pointers
            if self.front_pointer == None:
                self.back_pointer = None
            return item
        else:
            return None


# Breadth-first traversal
def bft(graph, root):
    visited = []
    q = Queue()
    current_vertex = root
    while current_vertex != None:
        if current_vertex not in visited:
            visited.append(current_vertex)
        for vertex in graph[current_vertex]:
            if vertex not in visited:
                q.enqueue(vertex)
                visited.append(vertex)
        current_vertex = q.dequeue()
    print(visited)
```

```
# Main program starts here
graph = {"A": ["B", "C", "D"], "B": ["A", "E"], "C": ["A", "D"], "D": ["A", "C", "F"], "E":
["B", "G"], "F": ["D"],
         "G": ["E"]}
bft(graph, "A")
```

Alternative implementations of the breadth-first search

In the code examples above, a queue structure was created using a class with only the enqueue and dequeue methods. There was no way to determine if there were items in the queue until a dequeue operation was attempted. With the addition of a method to check if the queue was empty, the algorithm could be rewritten to iterate only while there were items in the queue. Dequeue could then become the first operation:

```
Queue.Enqueue(Root)
While not Queue.Empty
      Current_vertex = Queue.Dequeue
      Visited.Add(Current_Vertex)
      For each edge in Current_Vertex
            If not visited.Contains(edge.vertex) Then
                  Queue.enqueue(edge.vertex)
                  Visited.Add(edge.vertex)
            End If
      Next
```

Some programming languages support queue data structures without needing them to be implemented by the programmer. When asked to write the code for a graph traversal, it is safe to assume that there is no need to write the code for the queue methods too.

Since queues are higher-order data structures, they can be implemented with arrays or lists in addition to an object-oriented approach. With lists, indexes change when items are added and deleted. Index zero will always be the front of the queue and new items will always be added to the end – so a list is already a queue.

In Python, you could use this code to implement a queue with a list:

Declare a list which will be the queue:	`Queue = []`
Enqueue an item:	`Queue.append(Vertex)`
Dequeue an item:	`Vertex = Queue.pop(0)`
Check if the queue has items:	`if len(Queue) > 0:`
Or more simply:	`if Queue:`

Did you know?

It is possible to implement a parallel breadth-first traversal to increase efficiency.

Depth-first traversal on a graph

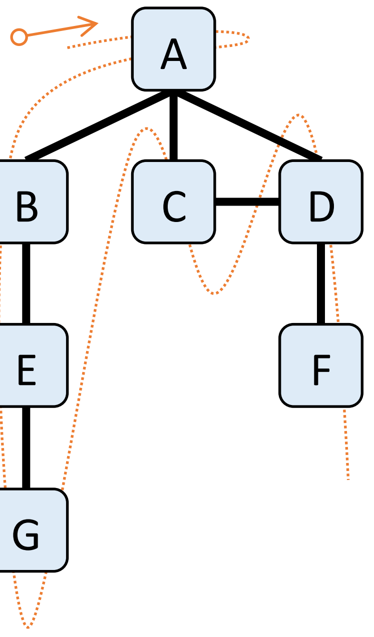
A depth-first traversal is used for pathfinding algorithms, operating system instruction scheduling, the order of formula recalculation in a spreadsheet, linkers and maze solution algorithms.

A depth-first traversal requires the use of a stack data structure.

1. Set a root vertex as the current vertex.
2. Add the current vertex to the list of visited vertices if it is not already in the list.
3. For every edge connected to the vertex:
    a. If the linked vertex is not in the visited list:
        i. Push the linked vertex to the stack.
4. Pop the stack and set the item removed as the current vertex.
5. Repeat from step 2 until the stack is empty.
6. Output all the visited vertices.

Depth-first traversal illustrated

A breadth-first traversal can be pictured like this:



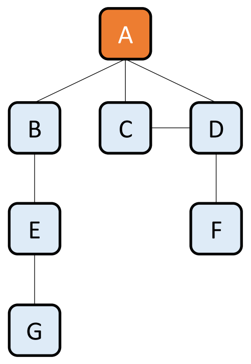
Depth-first traversal with recursion

This type of traversal assumes that a user-defined stack is being used with an iterative algorithm. However, it is also common to implement the depth-first traversal using only the call stack and recursion:

1. Set a root vertex as the current vertex.
2. Add the current vertex to the list of visited vertices.
3. For every edge connected to the vertex:
    a. If the linked vertex is not in the visited list:
        i. Recursively repeat from step 2 passing the linked vertex.

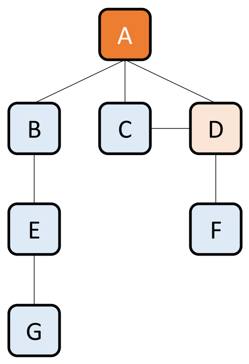
Stepping through the depth-first traversal using iteration and a user-defined stack

There are many different approaches to the depth-first traversal algorithm. Each one uses a stack in different ways to achieve an output. Using iteration as shown below, each vertex linked to an edge is added to the stack before moving to the next vertex. Later in the chapter, we will contrast this by using a recursive technique where we rely only on the call stack.

	Graph	Stack	Visited
Step 1	Start at a root vertex – this can be any vertex, but A has been chosen. A is not in the visited list; add A to the list of visited vertices. Consider each edge of A.		
			A

Step 2

D is not in the visited list; push D onto the stack.
We are exploring the vertices from right to left so they are output in the expected order, but it would be fine to push B instead and explore from left to right.

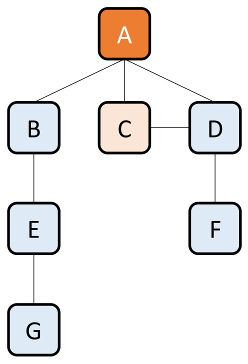


→	D

A

Step 3

C is not in the visited list; push C.

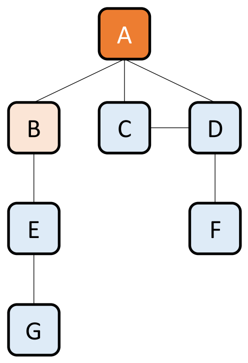


→	C
	D

A

Step 4

B is not in the visited list; push B.

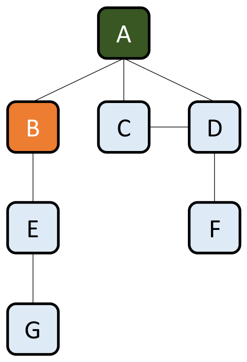


→	B
	C
	D

A

Step 5

All edges of A have been considered; pop the stack.
New current vertex is B.

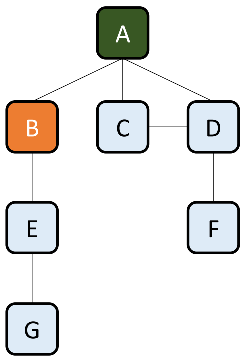


	B
→	C
	D

A

Step 6

B is not in the visited list; add B to the list of visited vertices.
Consider each edge of B.

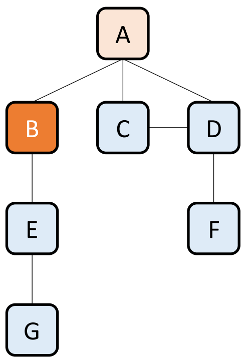


	B
→	C
	D

AB

Step 7

A is in the visited list; ignore the vertex.

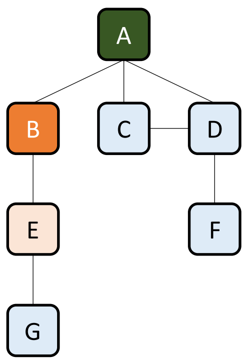


	B
→	C
	D

AB

Step 8

E is not in the visited list; push E onto the stack.

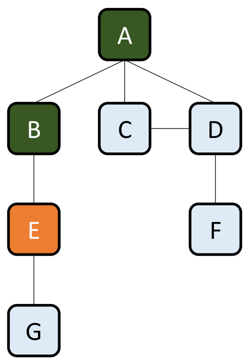


→	E
	C
	D

AB

Step 9

All edges of B have been considered; pop the stack.
New current vertex is E.

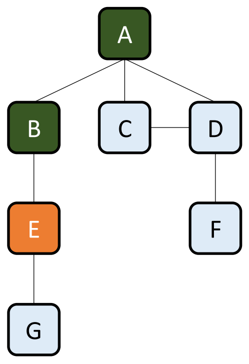


	E
→	C
	D

AB

Step 10

E is not in the visited list; add E to the list of visited vertices.
Consider each edge of E.

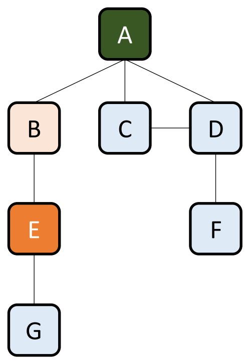


	E
→	C
	D

ABE

Step 11

B is in the visited list; ignore the vertex.

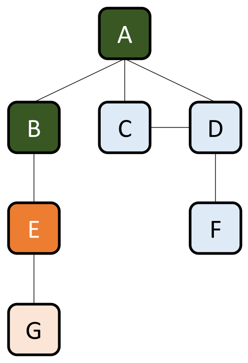


	E
→	C
	D

ABE

Step 12

G is not in the visited list; push G onto the stack.

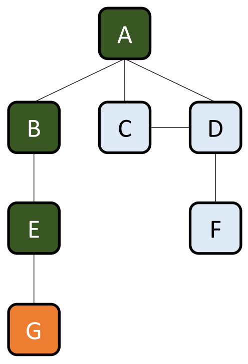


→	G
	C
	D

ABE

Step 13

All edges of E have been considered; pop the stack.
New current vertex is G.

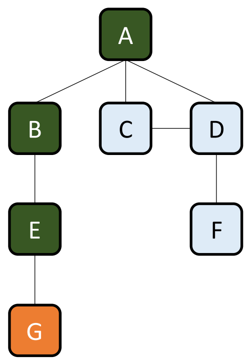


	G
→	C
	D

ABE

Step 14

G is not in the visited list; add G to the list of visited vertices.
Consider each edge of G.

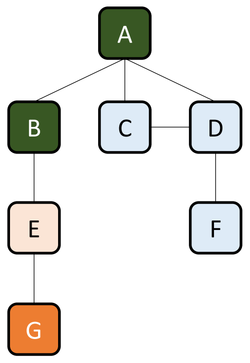


	G
→	C
	D

ABEG

Step 15

E is in the visited list; ignore the vertex.

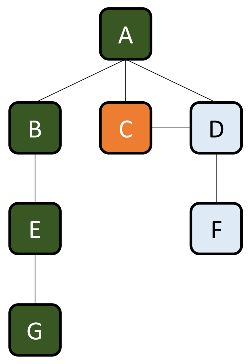


	G
→	C
	D

ABEG

Step 16

All edges of G have been considered; pop the stack.
New current vertex is C.

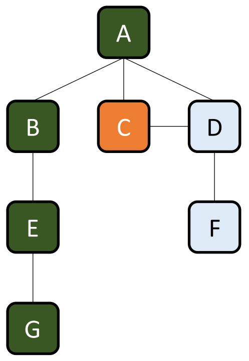


	G
	C
→	D

ABEG

Step 17

C is not in the visited list; add C to the list of visited vertices.
Consider each edge of C.

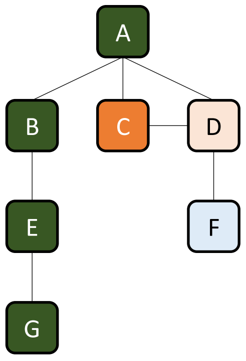


	G
	C
→	D

ABEGC

Step 18

D is not in the visited list; push D onto the stack.

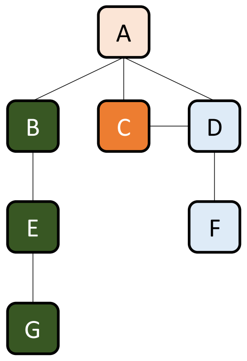


	G
→	D
	D

ABEGC

Step 19

A is in the visited list; ignore the vertex.

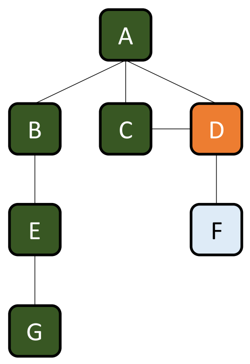


	G
→	D
	D

ABEGC

Step 20

All edges of C have been considered; pop the stack.
New current vertex is D.

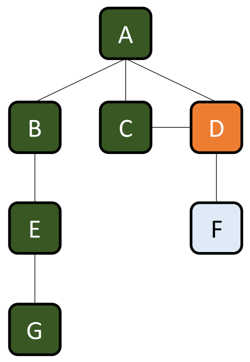


	G
	D
→	D

ABEGC

Step 21

D is not in the visited list; add D to the list of visited vertices.
Consider each edge of D.

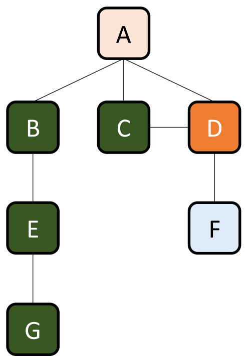


	G
	D
→	D

ABEGCD

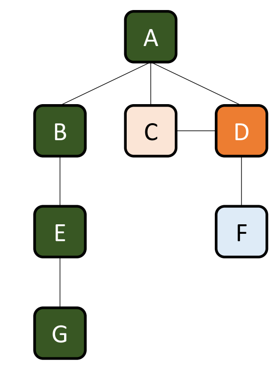
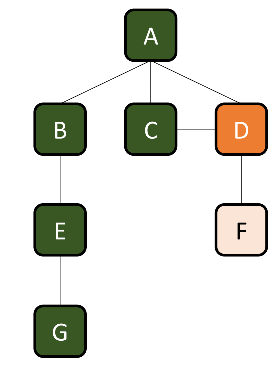
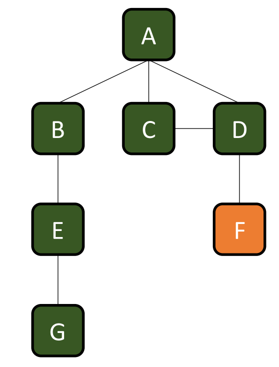
Step 22

A is in the visited list; ignore the vertex.



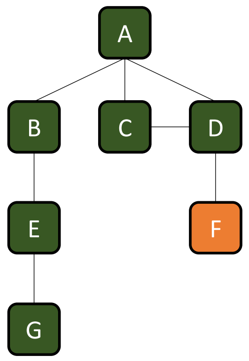
	G
	D
→	D

ABEGCD

Step 23	C is in the visited list; ignore the vertex.		
		G D → D	ABEGCD
Step 24	F is not in the visited list; push F onto the stack.		
		G → F D	ABEGCD
Step 25	All edges of D have been considered; pop the stack. New current vertex is F.		
		G F → D	ABEGCD

Step 26

F is not in the visited list; add F to the list of visited vertices.
Consider each edge of F.

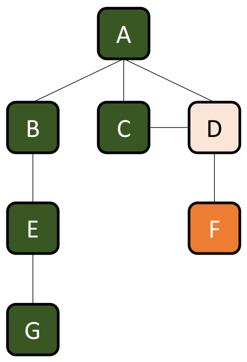


	G
	F
→	D

ABEGCDF

Step 27

D is in the visited list; ignore the vertex.

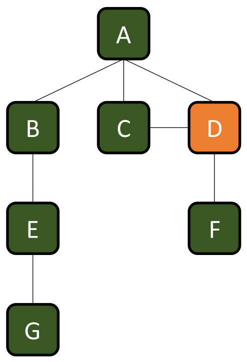


	G
	F
→	D

ABEGCDF

Step 28

All edges of F have been considered; pop the stack.
New current vertex is D.



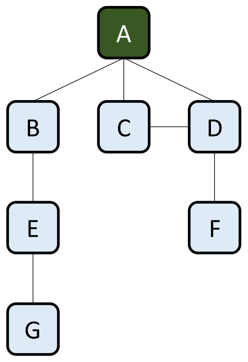
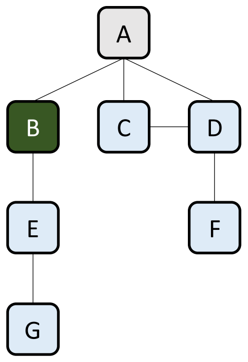
	G
	F
	D

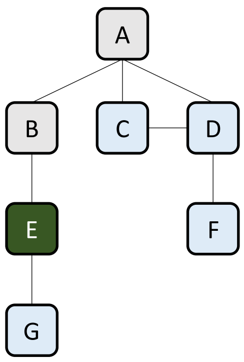
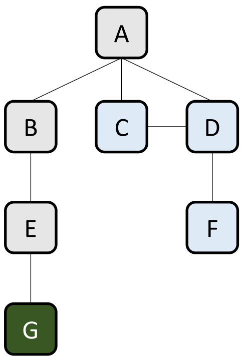
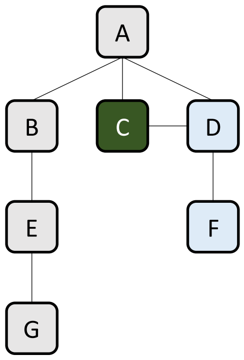
ABEGCDF

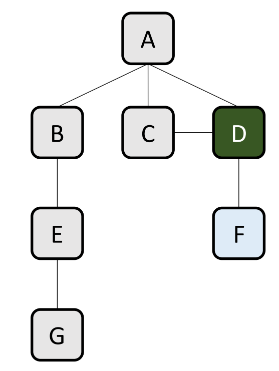
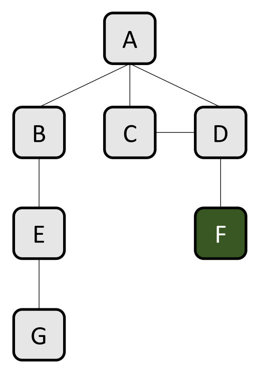
Step 29	D is in the visited list; ignore the vertex. Pop the stack. The stack is empty – the algorithm is complete.

Stepping through the depth-first traversal using recursion and the call stack

Although the use of a user-defined stack delivers the correct output, using only the procedure call stack with recursion is somewhat simpler. This approach is most likely to feature in mark schemes of depth-first traversal-related questions.

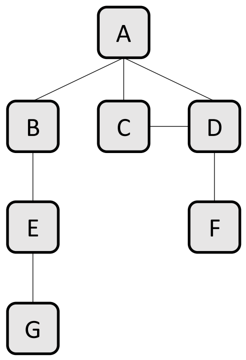
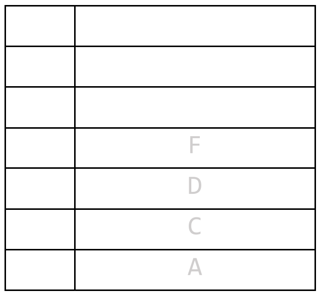
Step 1	Start at a root vertex. Push A to the stack and list of visited vertices.		
		→ A	A
Step 2	Push B to the stack and list of visited vertices.		
		→ B A	AB

Step 3	Push E to the stack and list of visited vertices.		
		→ E B A	ABE
Step 4	Push G to the stack and list of visited vertices.		
		→ G E B A	ABEG
Step 5	G has no unvisited edges. Pop the stack. E has no unvisited edges. Pop the stack. B has no unvisited edges. Pop the stack. Push C to the stack and list of visited vertices.		
		E → C A	ABEGC

Step 6	Push D to the stack and list of visited vertices.		
		→ D C A	ABEGCD
Step 7	Push F to the stack and list of visited vertices.		
		→ F D C A	ABEGCDF

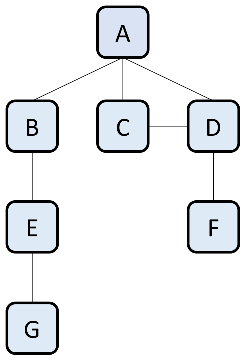
Did you know?

All algorithms with repeating instructions can be coded using iteration instead of recursion. It is usually the best choice because iterative algorithms execute within a defined memory space and are not reliant on free space in the call stack. However, it is often easier to write recursive algorithms, and more readable code is created. It may even be possible to use parallel processing to maximise efficiency.

Step 8	F has no unvisited edges. Pop the stack. D has no unvisited edges. Pop the stack. C has no unvisited edges. Pop the stack. A has no unvisited edges. Pop the stack. Algorithm complete.		
			ABEGCDF

Outputs from the depth-first traversal

It is worth noting that there can be multiple valid outputs from a depth-first traversal. You will typically see the leftmost path being traversed first in mark schemes and other sources. However, it is also valid to follow the rightmost path first or choose any random edge from a vertex to follow. In doing so, the results of the algorithm will be different – but nonetheless, it is still a depth-first traversal. Providing the algorithm follows edges to the bottom of the structure for any vertex, the output is valid as shown below.

Illustrated graph	Valid output 1	Valid output 2	Other valid outputs
	A B E G C D F This is the most common output shown in examples of this algorithm. Therefore, it is the one you should illustrate in exams unless the question specifically states otherwise.	A D F C B E G Reverse traversal.	A C D F B E G A D C F B E G A B E G D F C A B E G D C F

Pseudocode for a depth-first traversal using iteration

```
current_vertex = root
While current_vertex != Nothing
      If not visited.Contains(current_vertex) Then
            Visited.Add(current_vertex)
      End If
      For each edge in vertex
            If not visited.Contains(edge.vertex) Then
                  Stack.push(edge.vertex)
            End If
      Next
      current_vertex = Stack.pop
End While
For each vertex in visited
      Output current_vertex
Next
```

Depth-first traversal coded in Python using iteration

```python
# Stack data structure for depth-first traversal
class Stack:
    class Node:
        data = None
        pointer = None

    stack_pointer = None

    def push(self, item):
        # Check stack overflow
        try:
            # Push the item
            new_node = Stack.Node()
            new_node.data = item
            new_node.pointer = self.stack_pointer
            self.stack_pointer = new_node
            return True
        except:
            return False

    def pop(self):
        # Check stack underflow
        if self.stack_pointer != None:
            # Pop the item
            popped = self.stack_pointer.data
            self.stack_pointer = self.stack_pointer.pointer
            return popped
        else:
            return None

    def peek(self):
        # Check stack underflow
        if self.stack_pointer != None:
            # Peek the item
            return self.stack_pointer.data
        else:
            return None
```

```
# Depth-first traversal
def dft(graph, root):
    visited = []
    s = Stack()
    current_vertex = root
    while current_vertex != None:
        if current_vertex not in visited:
            visited.append(current_vertex)
        for vertex in reversed(graph[current_vertex]):
            if vertex not in visited:
                s.push(vertex)
        current_vertex = s.pop()
    print(visited)


# Main program starts here
graph = {"A": ["B", "C", "D"], "B": ["A", "E"], "C": ["A", "D"], "D": ["A", "C", "F"], "E":
["B", "G"], "F": ["D"], "G": ["E"]}
dft(graph, "A")
```

Depth-first traversal coded in Python using recursion

```
# Depth-first search using a graph
def dft(graph, current_vertex, visited):
    visited.append(current_vertex)
    for vertex in graph[current_vertex]:
        if vertex not in visited:
            dft(graph, vertex, visited)
    return visited


# Main program starts here
graph = {"A": ["B", "C", "D"], "B": ["A", "E"], "C": ["A", "D"], "D": ["A", "C", "F"], "E":
["B", "G"], "F": ["D"], "G": ["E"]}
print(dft(graph, "A", []))
```

Did you know?

Graph databases were created to overcome the limitations of relational databases. Unlike relational databases where there is a universal query language (SQL), there are a wide variety of query languages for graph databases such as Gremlin, SPARQL and Cipher. A proposal to create a universal query language known as GQL was approved by the International Organization for Standardization (ISO) in 2019.

Efficiency of operations on a graph

	Time complexity	
	Object implementation, adjacency list	Array implementation, adjacency matrix
Storing graph	$O(V+E)$ Linear	$O(V^2)$ Polynomial
Add vertex	$O(1)$ Constant	$O(V^2)$ Polynomial
Add edge	$O(1)$ Constant	$O(1)$ Constant
Remove vertex	$O(E)$ Linear	$O(V^2)$ Polynomial
Remove edge	$O(V)$ Linear	$O(1)$ Constant

Space complexity	
Best case	Worst case
$O(1)$ Constant	$O(n)$ Linear

Although the examples presented in this chapter use a static dictionary, a graph is considered a dynamic data structure. When implemented using object-oriented techniques, its memory footprint grows and shrinks as data is added and deleted from the structure. Using only the required amount of memory is the most efficient way of implementing a graph.

When implementing a graph as an array, the memory footprint remains constant but operations such as adding new vertices may require the matrix to be recreated.

When expressing the time complexity of operations on a graph, V represents the number of vertices and E represents the number of edges. Storing a graph using object-oriented techniques is far superior to using an array in most cases – the only exception being removing edges and checking the adjacency of two vertices.

Both of these operations are constant O(1) with an adjacency matrix, as you can immediately return an element of an index in an array. However, they are linear O(V) with an object-oriented approach because it is necessary to follow the edges to find a particular vertex.

Efficiency of a breadth and depth-first traversal

Best case	Average case	Worst case
O(1) Constant	O(V+E) Linear	O(V+E) Linear

The breadth-first traversal can be used to find a single vertex in the structure or output all the data stored in it. At best, the graph contains just one vertex, so it can be found immediately – O(1) – but if that were the case, there wouldn't be any point in having the data structure at all.

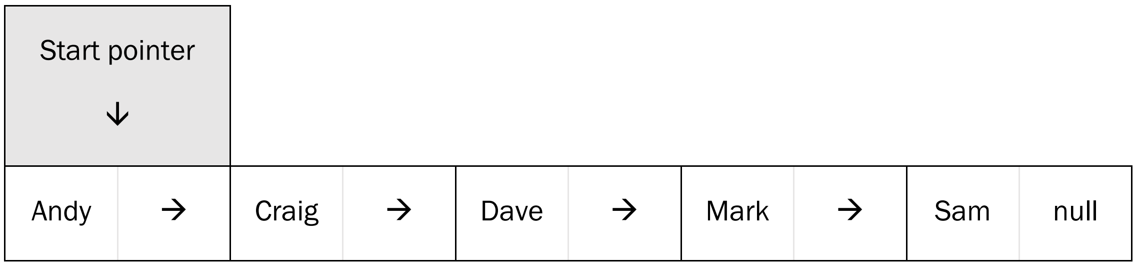
Usually, many vertices need to be visited, so it is of linear complexity O(V+E), where V represents the number of vertices and E the number of edges. Although the nested loop in an algorithm often indicates polynomial complexity, the time complexity is O(V + E) or linear at worst case because items are enqueued once.

Did you know?

One proposed optimisation of the breadth-first search is the direction-optimising algorithm described by Scott Beamer in 2016. A conventional top-down approach is combined with a new bottom-up approach. The resulting algorithm examines fewer edges and improves the efficiency of the breadth-first search in many cases.

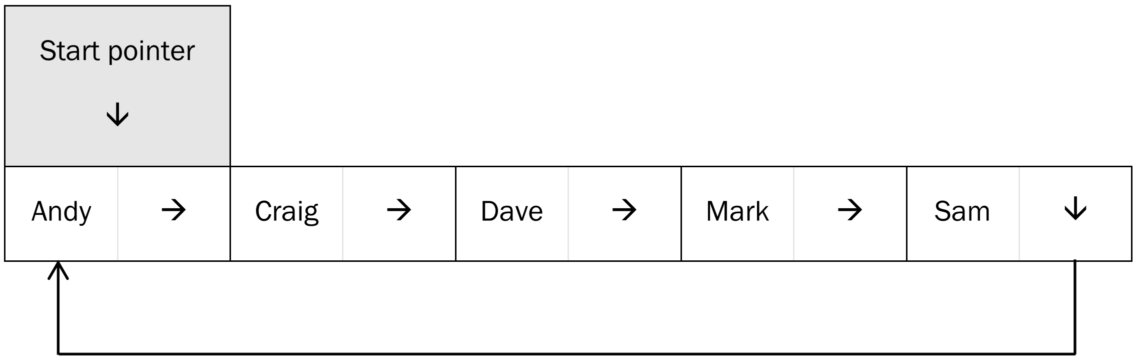
Linked list

A linked list is a dynamic, ordered structure of nodes and pointers. Unlike a regular list, items in a linked list are not necessarily stored in contiguous memory, they do not use an index and they must maintain their own references to the next item in the structure. A start pointer is used to identify the first node in a linked list, and each item points to the next.

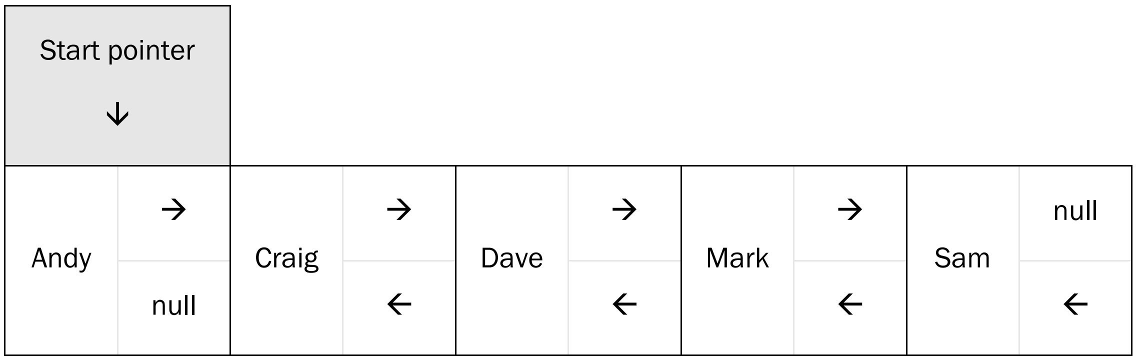


Circular and doubly linked lists

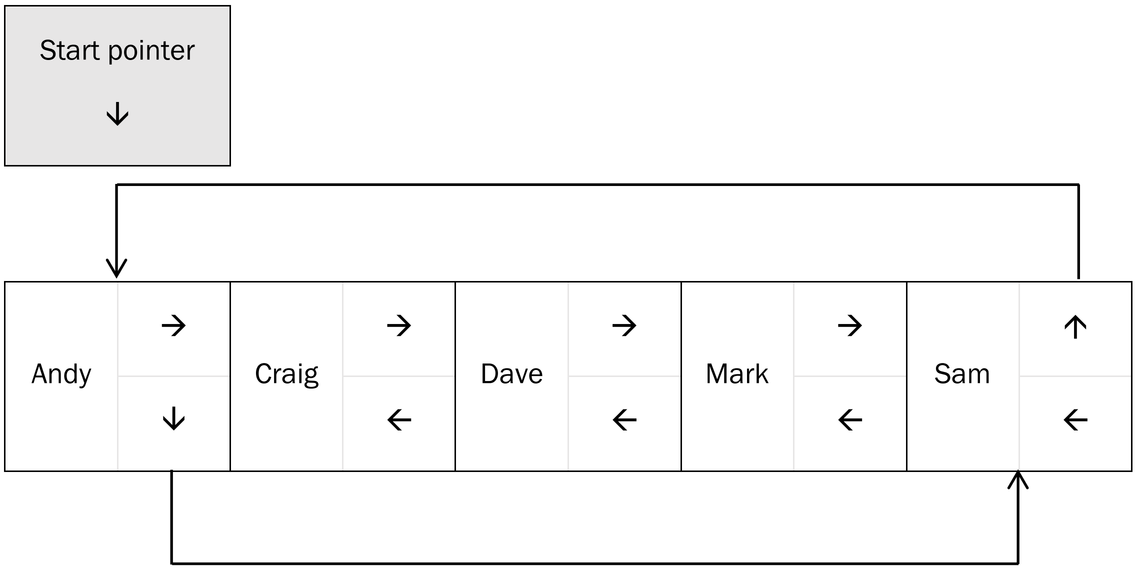
By using the pointer from the last node to point to the first node, a circular linked list can be created.



By adding an extra pointer to each element, a node can point to both the previous and next nodes – we refer to this as a **doubly linked list**.



With a circular linked list, the pointers on the first and last elements can also point to each other, creating a doubly circular linked list.



Applications of a linked list

Linked lists can be used to store process blocks in a ready state for operating systems managing a processor, image viewers moving between previous and next images, music players storing tracks in a playlist or navigating backwards and forwards in a web browser. Linked lists can also be used for hash table collision resolution as an overflow or maintaining a file allocation table of linked clusters on a secondary storage medium such as a hard disk.

Although a linked list can simply be an implementation of a list without indexes, it is common to use a linked list to maintain a logical order to the items in the structure irrespective of their physical order. New items are always added to the end of the structure, but the pointers are updated so that if they were followed from the start pointer, they would visit each node in the order of the data stored.

Did you know?

Hans Peter Luhn, the inventor of the hash map and check digit, first proposed linked lists as a structure in 1953.

Storing a linked list in memory

A linked list can be represented in memory using an array, a list or with node objects.

<table>
<tr><th>Array</th><th>Object</th></tr>
<tr><td>

```
start = 1
nextFree = 4
```

Index	Element	Pointer
0	Dave	3
1	Craig	0
2	Sam	-1
3	Mark	2
4		5
5		6
6		-1

A doubly linked list would have another dimension for a second pointer.

A circular linked list would have the pointer at index 6 assigned to zero.

</td><td>

A single node is defined as:

```
Class Node
      element = ""
      Pointer = Node
End Class
```

</td></tr>
</table>

Did you know?

Some implementations of the linked list use sentinel or dummy nodes, empty nodes at the start or end of the structure. These empty nodes can be used to ensure a linked list always has a first and last node, simplifying the operations.

Operations on a linked list

Typical operations that can be performed on a linked list include:

- Add: Adds a node to the linked list.
- Delete: Removes a node from the linked list.
- Next: Moves to the next item in the list.
- Previous: Moves to the previous item in a doubly linked list.
- Traverse: A linear search through the linked list.

Array implementation of a linked list

With an array implementation of a linked list, two pointers need to be maintained – one pointer to the index of the first node containing data and another pointer to the index of the next available free element. The value for the start pointer can be set to -1 to indicate the linked list is empty. The value of the pointer for the last element in the array can be set to -1 to indicate there is no more free space in the structure.

With a linked list, each item points to the next – this is achieved with a two-dimensional array where one dimension holds the value and another dimension holds the index to the next item. Items are added at the next available free element. When an item is deleted, it becomes the next available free element and its pointer is updated to be the previous free element.

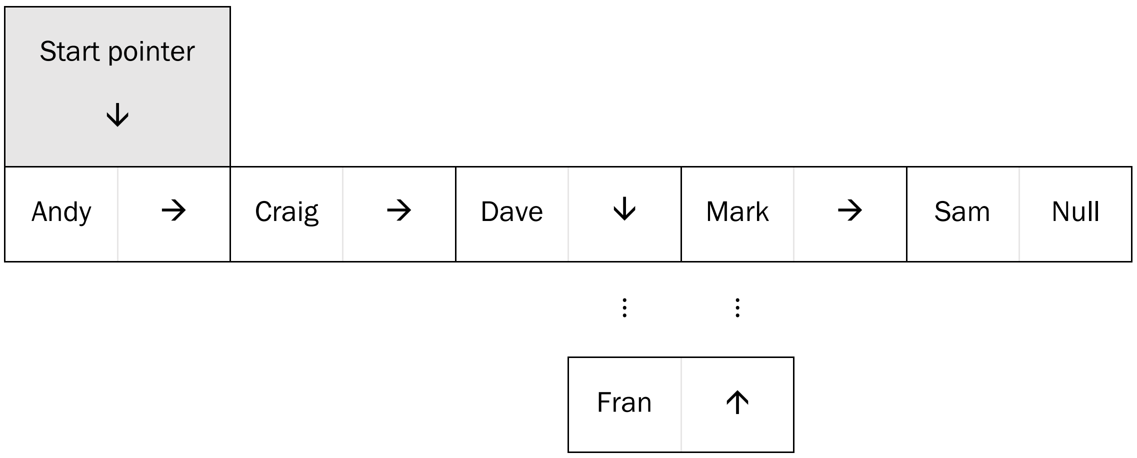
A linked list is still considered to be dynamic even though, with an array implementation, it is constrained in size by the static structure.

Object implementation of a linked list

With an object implementation, only one pointer needs to be maintained, pointing to the first node containing data. When an item is added or deleted, memory is taken or returned to the heap and the pointers of the other nodes are changed. Object implementations are truly dynamic and can grow or shrink in size.

Adding an item to a linked list

1. Check there is free memory for a new node. Output an error if not.
2. Create a new node and insert data into it.
3. If the linked list is empty:
    a. The new node becomes the first item. Create a start pointer to it.
4. If the new node should be placed before the first node:
    a. The new node becomes the first node. Change the start pointer to it.
    b. The new node points to the second node.
5. If the new node is to be placed inside the linked list:
    a. Start at the first node.
    b. If the data in the current node is less than the value of the new node:
        i. Follow the pointer to the next node.
        ii. Repeat from step 5b until the correct position is found or the end of the linked list is reached.
    c. The new node is set to point where the previous node pointed.
    d. The previous node is set to point to the new node.



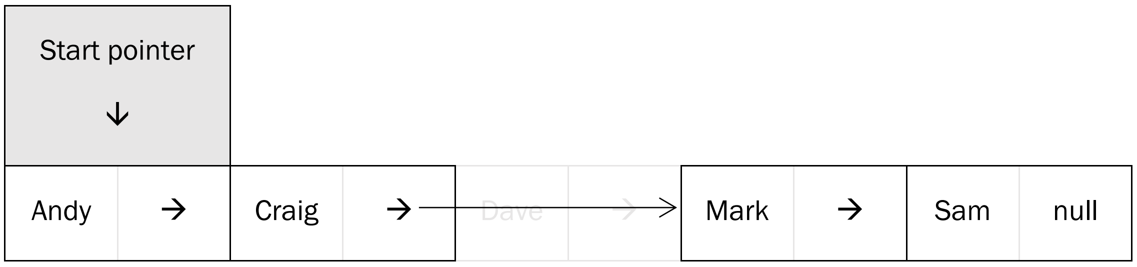
The assumption here is that we are maintaining an ordered list. If that is not necessary, a new node can simply be added to the end of the structure and the pointers updated. For the purposes of examinations, you should always assume that the linked list will be ordered.

Pseudocode for adding an item to a linked list

```
If not memoryfull Then
    new_node = New Node
    current_node = start_pointer
    If current_node == Null Then
        new_node.pointer = Null
        start_pointer = new_node
    Else
        If item < current_node.data Then
            start_pointer = new_node
            new_node.pointer = current_node
        Else
            While current_node != Null And item < current_node.data
                previous_node = current_node
                current_node = current_node.pointer
            End While
            new_node.pointer = previous_node.pointer
            previous_node.pointer = new_node
        End If
    End If
End If
```

Deleting an item from a linked list

1. Check if the linked list is empty and output an error if there is no start node.
2. If the first item is the item to delete, set the start pointer to the next node.
3. If the item to delete is inside the linked list:
    a. Start at the first node.
    b. If the current node is the item to delete:
        i. The previous node's pointer is set to point to the next node.
    c. Follow the pointer to the next node.
    d. Repeat from step 3b until the item is found or the end of the linked list is reached.



The node is not physically removed but, rather, the pointer from the previous item is updated. The garbage collection process will reclaim the unused memory.

Pseudocode for deleting an item from a linked list

```
current_node = start_pointer
If current_node != Null Then
      If item == current_node.data Then
            start_pointer = current_node.pointer
      Else
            While current_node != Null And item != current_node.data
                  previous_node = current_node
                  current_node = current_node.pointer
            End While
            previous_node.pointer = current_node.pointer
      End If
End If
```

Traversing a linked list

1. Check if the linked list is empty.
2. Start at the node that the start pointer is pointing to.
3. Output the item.
4. Follow the pointer to the next node.
5. Repeat from step 3 until the end of the linked list is reached.

Pseudocode for traversing a linked list

```
current_node = start_pointer
If current_node != Null Then
      While current_node != Null
            Output current_node
            current_node = current_node.pointer
      End While
End If
```

Did you know?

In 1955, Allen Newell, Cliff Shaw and Herbert Simon used linked lists as a primary data structure for their Information Processing Language (IPL). IPL was used to develop AI programs including Newell, Shaw and Simon's Logic Theory Machine.

Linked list coded in Python using an array

```python
class LinkedList:
    max = 10
    llist = [["" for pointer in range(2)] for item in range(max)]
    # Initialise the free items list
    for index in range(max - 2):
        llist[index][1] = index + 1
    llist[max - 1][1] = -1
    nextFree = 0
    start = -1

    def add(self, item):
        current_node = self.start
        # Check memory overflow
        if self.nextFree != -1:
            new_node = self.nextFree
            self.llist[new_node][0] = item
            self.nextFree = self.llist[self.nextFree][1]
            # List is empty
            if self.start == -1:
                self.llist[new_node][1] = -1
                self.start = new_node
            else:
                # Item becomes the new start item
                if item < self.llist[current_node][0]:
                    self.start = new_node
                    self.llist[new_node][1] = current_node
                else:
                    # Find correct position in the list
                    while current_node != -1 and self.llist[current_node][0] < item:
                        previous_node = current_node
                        current_node = self.llist[current_node][1]
                    self.llist[new_node][1] = self.llist[previous_node][1]
                    self.llist[previous_node][1] = new_node
            return True
        else:
            return False

    def delete(self, item):
        current_node = self.start
        # Check the list is not empty
        if current_node != -1:
            # Item is the start node
            if item == self.llist[current_node][0]:
                self.start = self.llist[current_node][1]
            else:
                # Find the item in the list
                while current_node != -1 and item != self.llist[current_node][0]:
                    previous_node = current_node
                    current_node = self.llist[current_node][1]
                self.llist[previous_node][1] = self.llist[current_node][1]
            # Return deleted node to the free list
            self.llist[current_node][1] = self.nextFree
            self.nextFree = current_node
    def output(self):
        items = []
        current_node = self.start
        if self.start != -1:
            # Visit each node
            while current_node != -1:
                items.append(self.llist[current_node][0])
                current_node = self.llist[current_node][1]
        return items


# Main program starts here
items = ["Florida", "Georgia", "Delaware", "Alabama", "California", "Wyoming"]
linked_list = LinkedList()
# Adding items to the linked list
for index in range(0, len(items)):
    linked_list.add(items[index])
# Deleting items from a linked list
linked_list.delete("Florida")
# Output the linked list
print(linked_list.output())
```

Linked list coded in Python using objects

```
class LinkedList:
    class Node:
        data = None
        pointer = None

    start = None

    def add(self, item):
        # Check memory overflow
        try:
            new_node = LinkedList.Node()
            new_node.data = item
            current_node = self.start
            # List is empty
            if current_node == None:
                new_node.pointer = None
                self.start = new_node
            else:
                # Item becomes the new start item
                if item < current_node.data:
                    self.start = new_node
                    new_node.pointer = current_node
                else:
                    # Find correct position in the list
                    while current_node != None and current_node.data < item:
                        previous_node = current_node
                        current_node = current_node.pointer
                    new_node.pointer = previous_node.pointer
                    previous_node.pointer = new_node
            return True
        except:
            return False
    def delete(self, item):
        current_node = self.start
        # Check the list is not empty
        if current_node != None:
            # Item is the start node
            if item == current_node.data:
                self.start = current_node.pointer
            else:
                # Find item in the list
                while current_node != None and item != current_node.data:
                    previous_node = current_node
                    current_node = current_node.pointer
                previous_node.pointer = current_node.pointer

    def output(self):
        items = []
        current_node = self.start
        if current_node != None:
            # Visit each node
            while current_node != None:
                items.append(current_node.data)
                current_node = current_node.pointer
        return items


# Main program starts here
items = ["Florida", "Georgia", "Delaware", "Alabama", "California", "Wyoming"]
linked_list = LinkedList()
# Adding items to the linked list
for index in range(0, len(items)):
    linked_list.add(items[index])
# Deleting items from a linked list
linked_list.delete("Florida")
# Output the linked list
print(linked_list.output())
```

Operations

Adding items:	linked_list.add("Idaho")
Deleting items:	linked_list.delete("Idaho")
Outputting items:	print(linked_list.output)

Did you know?

The disadvantage of a linked list is that elements cannot be randomly accessed – finding an element requires a linear search, which is slow.

Efficiency of operations on a linked list

	Time complexity		
	Best case	Average case	Worst case
Access	O(1) Constant	O(n) Linear	O(n) Linear
Next, Previous	O(1) Constant	O(1) Constant	O(1) Constant
Search, Traverse	O(1) Constant	O(n) Linear	O(n) Linear
Add item	O(1) Constant	O(n) Linear	O(n) Linear
Delete item	O(1) Constant	O(n) Linear	O(n) Linear

Space complexity	
Best case	Worst case
O(1) Constant	O(n) Linear

A linked list is a dynamic data structure. Implemented using object-oriented techniques, its memory footprint grows and shrinks as data is added and deleted. Using only the required amount of memory is the most efficient way to implement a linked list. However, a linked list can also be implemented using an array. In this case, the dynamic structure is created upon a static data structure and the memory footprint remains constant – this is inefficient, as the linked list reserves more memory than it needs unless it is full.

If the order of items is important, you can find the correct position to add or delete a node using a linear search. At best, a new node becomes the first node, O(1), or it is the first to be deleted, O(1). Typically, this will not be the case, resulting in a linear complexity, O(n).

If the order of the items in a linked list is not important, adding a new item will always have a time complexity of O(1). In this situation, a pointer to the last item would be used to prevent having to follow the pointers from node to node to find the position of the last item in the structure.

List

A list is a dynamic collection of elements of different data types. Like arrays, lists are indexed and references to items are held in contiguous memory. Unlike arrays, the size of the list can change when a program is running, and memory is allocated when it is required.

When choosing between an array and a list, programmers need to consider the purpose of the structure. If the number of items is known in advance and is of the same data type, an array is a better choice. Not only can addresses of indexes in an array be calculated, but a list may need to be moved in memory as new items are added to ensure the structure is held in contiguous memory space. However, lists are often more memory-efficient because they only use the memory they need for the data they are storing at the time.

A list declaration and assignment in Python:

```
items = ["Florida", "Georgia", "Delaware", "Alabama", "California"]
```

Two-dimensional lists can also be implemented as a list of lists.

The first item in a list is often referred to as the *head* and all other items in the list except the head are the *tail*. Some programming languages support head-and-tail operations to extract those items from the list for further processing.

Lists are implemented in different ways by different programming languages – for example, as a linked list using objects. You will often hear list and linked list being used synonymously when, in fact, the linked list is just one possible implementation of a list.

Applications of a list

Lists are used in situations where a set of related data items needs to be stored but the number of items may vary and be of different data types. Lists are typically used to implement other data structures such as stacks and queues and to hold values of a dictionary.

Storing a list in memory

Given that each element can be a different data type – and therefore use a different amount of memory – it is not possible to use both indexes and contiguous data items. To overcome this problem, references to the data items are stored contiguously instead.

Heap

	Address	Value	*Notes*
References to data items	0x5D	0x63	*item[0]*
	0x5E	0x8E	*item[1]*
	0x5F	0xA3	*item[2]*
	0x60	0xA4	*item[3]*
	…		
Data items	0x63	Sword	*string*
	…		
	0x8E	20	*integer*
	…		
	0xA3	Shield	*string*
	0xA4	100	*integer*

In the example above, a list identified as *item* has four elements at indexes 0 to 3, with the first index stored at memory address 0x5D. The address of the other indexes of the list can now be calculated because addresses are all a fixed size. What is stored in address 0x5D is not the item but the address where the item is stored. The data for item[0] is actually stored in 0x63, *Sword*. This is an example of indirect addressing.

Did you know?

Lists can be made from objects too. In this implementation, they become linked lists.

Operations on a list

Typical operations that can be performed on a list include:

- Access: Return an item by its index.
- Add: Append or insert an item in the list.
- Delete: Remove an item from the list.
- Display: Output all the items in the list.
- Head: Return the first item in the list.
- Exists: Find if an item is in the list.
- Tail: Return all the items in the list except for the first item.

Efficiency of operations on a list

	Time complexity		
	Best case	Average case	Worst case
Access, Head	$O(1)$ Constant	$O(1)$ Constant	$O(1)$ Constant
Search, Exists, Display, Tail	$O(1)$ Constant	$O(n)$ Linear (one-dimension)	$O(n^2)$ Polynomial (two-dimension)
Add item	$O(1)$ Constant	$O(n)$ Linear	$O(n)$ Linear
Delete item	$O(1)$ Constant	$O(n)$ Linear	$O(n)$ Linear

Space complexity	
Best case	Worst case
$O(1)$ Constant	$O(n)$ Linear

Since a list is a dynamic data structure, its memory footprint is not usually known in advance, so it has a linear space complexity, O(n). A tuple is immutable – therefore, in a best-case scenario, the list may have a constant space complexity, O(1).

Like arrays, list elements can be accessed using an index. While this may require a second hop to where the data is actually stored on the heap (in order to support multiple data types), items can still be accessed in a constant time, O(1).

Searching for an item in a list may return the first item checked, O(1), or require checking every item until the one required is reached, O(n). Lists of lists may also further degrade the complexity to polynomial, O(n^2), because of the requirement for a nested loop.

New items are typically added to the end of the list. If an item to be deleted is the last index, both operations can be performed in a constant time, O(1). However, the dynamic nature of lists and the fact indexes remain contiguous as items are deleted – even from the middle of the list – means these operations are frequently linear, O(n), due to the moving of other references in the structure.

Did you know?

References to items in lists are stored in contiguous memory like an array. However, as items are added and removed from the middle of a list, no empty space remains. The items in subsequent indexes fill the available space.

For example, if item[0] is deleted, item[1] will become item[0] – this is extremely useful for programming data structures like queues using a list.

As new items are added to the list, it may become necessary to move the entire structure in memory to ensure references to items remain contiguous.

Object

The first computers used arrays for storing data. By 1965, Simula had introduced a new way of programming using an object-oriented approach. Programs were built from fundamental constructs called objects.

An object has attributes (variables) and methods (subroutines). Like a record data structure, an object may contain many related attributes of different data types. Unlike records, attributes of an object can be public or private; a similar concept to the scope of a variable being global or local. Public attributes are like global variables and can be used and accessed by other objects. Private attributes can only be accessed by methods of the object – this is known as encapsulation.

Over the years, the concept of object-oriented programming (OOP) changed from its original intentions of self-contained data structures – messages being passed between objects – into a more complex model. Today, OOP includes many additional features like sub-classes, inheritance and polymorphism, which are beyond the scope of this book. Instead, we will look at how objects are used as data structures for algorithms.

An object is defined by using a class structure. For example, part of a person object might look like this expressed as a class diagram:

Class: Person
Attributes: Name: string EyeColour: string HeightInCm: integer

Methods:
SetName()
SetHeight()

Objects are also known as instances of a class. You can create (instantiate) as many copies of an object as you need from a single class structure. Since the number of objects can change at run-time, objects are dynamic data structures.

Maintaining groups of objects is usually important when programming – so you can use an iteration to call the methods in all the objects one at a time, for example. Objects can be stored in arrays, known as a collection. Alternatively, you can use an attribute of an object as a pointer to store the address of another object, effectively linking them together.

Applications of an object

Objects are used in situations where there are many related data items of different data types. Objects are useful when the number of data items to be stored in the structure cannot be known in advance and fast access to data is required. All algorithms can use an object as their base data structure.

Storing an object in memory

Like all dynamic data structures, when an object is created, memory is allocated to store it on the heap and a reference to the start of that block of memory is returned and stored on the call stack.

Attributes of an object may be stored as key-value pairs like a dictionary.

Operations on an object

Typical basic operations that can be performed on an object include:

- Construct: A method run when an object is first instantiated.
- Delete: When the last reference to an object is removed, the garbage collector makes the memory available to other processes.
- Get: A public method to retrieve the value of a private, encapsulated attribute.
- Set: A public method to store a value to a private, encapsulated attribute.
- Receive a message from another object.
- Send a message to another object.

Did you know?

In Java, every data structure is an object – including arrays.

Efficiency of operations on an object

	Time complexity		
	Best case	Average case	Worst case
Access	O(1) Constant	O(1) Constant	O(n) Linear
Construct	O(1) Constant	O(1) Constant	O(1) Constant
Get, Set	O(1) Constant	O(1) Constant	O(n) Linear
Delete	O(1) Constant	O(1) Constant	O(n) Linear

Space complexity	
Best case	Worst case
O(1) Constant	O(n) Linear

Since an object is a dynamic data structure, the number of objects required is usually not known in advance, so it has a linear space complexity, O(n). In some applications, the maximum number of objects is fixed to maintain performance. Objects are constructed in advance and referenced in a collection or pool. Take a video game, for example, where no more enemies can be spawned than the maximum number of enemies available in the pool. In this unique situation, the memory footprint is known, O(1).

Creating, deleting and accessing an object can usually be done in a constant time, O(1), because an object is an unordered structure. An object in memory can be accessed immediately from its block address, with individual attributes often stored as key-value pairs. This could potentially increase the complexity of operations to O(n) depending on how the object and its attributes are stored.

The time complexity of other operations on a collection of objects depends on what the objects are being used for. For example, if objects are being used to store a binary tree for a post-order traversal, consider the time complexity of that implementation.

Queue

A queue is a dynamic data structure. Items are *enqueued* (added) at the back of the queue and *dequeued* (removed) from the front of the queue. It is also possible to *peek* at the front item without removing it.

Imagine a queue at a checkout. The person at the front is served first and people join at the back. This strict process can also allow for people to jump the queue – when implemented in computer science, this is known as a priority queue. In special circumstances, new items can join either the front or back of the queue.

A queue is known as a first-in, first-out or FIFO structure.

A queue has a back pointer that always points to the last item in the queue, sometimes referred to as the tail or rear pointer. A queue also has a front pointer that always points to the first item in the queue, sometimes referred to as the head pointer.

Front ↓				Back ↓	
Craig	Dave	Sam	Mark	Carol	

Applications of a queue

Queues are used for process scheduling, transferring data between processors and printer spooling. They are also used as buffers between input/output devices operating at different speeds.

In programming, queues are useful for reversing the elements in an array. In game design, they can be used to manage the spawning of objects to ensure the CPU is not overloaded with operations. Queues are required to perform a breadth-first traversal on graph data structures. Simulations of real-world models such as modelling traffic congestion can also use queues.

Did you know?

With a priority queue, each element also has an additional priority attribute. When items are enqueued in a priority queue, they are inserted into the correct position from highest to lowest in the queue. Two elements with the same priority are served in the order they arrived. Priority queues are very useful for implementing operating system scheduling algorithms such as shortest remaining time.

Storing a queue in memory

Queues can be represented in memory using an array, a list or objects.

Array	List	Object

`front = 0`
`back = 4`
 	
Index | Element
0 | Craig
1 | Dave
2 | Sam
3 | Mark
4 | Carol
5 |
6 |
	`front = 0`
	`back = 4`
	
	Index | Element
	0 | Craig
	1 | Dave
	2 | Sam
	3 | Mark
	4 | Carol
	
	Note, there is no free space with a list implementation because it is truly dynamic. The front pointer is always at index 0.
		`Class Queue`
		`      frontPointer = None`
		`      backPointer = None`
		
		`  Class Node`
		`      element = ""`
		`      pointer = Node`
		`  End Class`
		
		`End Class`

Operations on a queue

Typical operations that can be performed on a queue include:

- Enqueue: Add an item to the back of the queue.
- Dequeue: Remove an item from the front of the queue.
- Peek: Return the value from the front of the queue without removing it.
- Is Empty: Return whether the queue is empty.
- Is Full: Return whether the queue is full.

An attempt to enqueue an item to a queue that is already full is called a queue overflow, while trying to dequeue an item from an empty queue is called a queue underflow. Both situations should be considered before attempting to enqueue or dequeue an item.

Array implementation of a queue

An array is declared large enough to accommodate the maximum number of items. The front and back pointers are held using variables that are indexes to elements of the array, set to -1 when they are not pointing to any elements – i.e., when the queue is empty.

When implementing a queue using an array, a unique problem arises because the array is a static structure of a fixed size. As items are enqueued and dequeued, the values of the front and back pointers will continue to increase. Eventually, they will reach the end of the structure, preventing any more items from being added – this is known as a linear queue because the pointers never decrease in value.

A potential solution would be to move all the items in the array down by one index when they are dequeued. However, moving data items is not efficient, O(n), so it is avoided. A better solution is to cycle back to the start of the array when the bounds are reached, known as a circular queue.

Linear queue

The status of a linear queue after five items have been enqueued using an array with seven indexes:

Index	0	1	2	3	4	5	6
Pointers	Front ↓				Back ↓		
Element	Craig	Dave	Sam	Mark	Carol		

The status of the linear queue after dequeuing an item using an array:

Index	0	1	2	3	4	5	6
Pointers		Front ↓			Back ↓		
Element	Craig	Dave	Sam	Mark	Carol		

Note that *Craig* was not actually removed from the structure; the front pointer moved instead.

The status of the linear queue after enqueuing the next item using an array:

Index	0	1	2	3	4	5	6
Pointers		Front ↓				Back ↓	
Element	Craig	Dave	Sam	Mark	Carol	Andy	

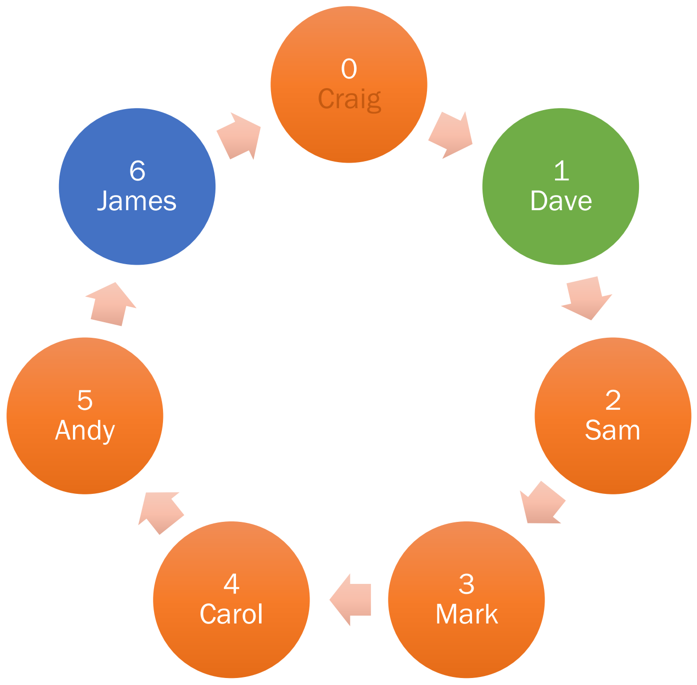
The status of the queue after enqueuing another item using an array:

Index	0	1	2	3	4	5	6
Pointers		Front ↓					Back ↓
Element	Craig	Dave	Sam	Mark	Carol	Andy	James

Note how the array is now full and no more items can be enqueued.

Circular queue

Often shown in illustrations as a cycle:



The status of a circular queue before enqueuing a new item:

Index	0	1	2	3	4	5	6
Pointers		Front ↓					Back ↓
Element	Craig	Dave	Sam	Mark	Carol	Andy	James

The status of the circular queue after enqueuing a new item:

Index	0	1	2	3	4	5	6
Pointers	Back ↓	Front ↓					
Element	Fran	Dave	Sam	Mark	Carol	Andy	James

Notice that the back pointer has looped back around to the start of the structure. There are still a finite number of items that can be stored, but it is less restrictive than a linear queue. An array with a circular queue is ideal if you know the memory footprint and want to restrict the number of items in the structure.

It is particularly useful in game design where the number of sprites on the screen affects the framerate. By only spawning sprites up to the limit of the queue, a stable framerate can be achieved. Changing the value of a pointer when an item is enqueued or dequeued can be achieved with the following code:

```
pointer = pointer + 1
if pointer = max then pointer = 0
```

However, by making use of the modulo operator, the new position of a pointer can be calculated in a single statement as:

`pointer = mod max` (In Python: `pointer = pointer % max`)

Here is an easier way to check if a queue is full before attempting to add a new item regardless of the values of the front and back pointers:

```
if (pointer + 1) mod max = front then queueFull
```

Did you know?

In computer science, modulo is the operator used to calculate modulus.

The modulus can be found by calculating the integer division of one number by another (called the quotient), multiplying the quotient by the divisor and subtracting the resulting number from the initial number – e.g., 3 modulo 2 is:

3 / 2 = 1 (rounded down)
1 * 2 = 2
2 – 1 = 1

How modulus works

Modulo is an arithmetic operator that returns the remainder from a division of two numbers. In Python, this is represented by the percent symbol – e.g., pointer + 1 % max.

pointer	pointer + 1	max	(pointer + 1) mod max
0	1	7	1
1	2	7	2
2	3	7	3
3	4	7	4
4	5	7	5
5	6	7	6
6	7	7	0

List implementation of a queue

When implementing a queue using a list, we can use the special property of list indexes. As items are removed from a list, references to subsequent items are moved down to replace the deleted element, the back pointer is decremented, and the front of the queue is always at index zero – this negates the need for an additional variable to store the value of the front pointer because it is always constant.

There is no need to implement a circular queue because a list is dynamic. New elements are allocated from the heap and indexed as they are required. The linear queue works with list and object implementations.

The status of a queue after five items have been enqueued with a list:

Index	0	1	2	3	4
Pointers	Front ↓				Back ↓
Element	Craig	Dave	Sam	Mark	Carol

The status of the queue after dequeuing an item with a list:

Index	0	1	2	3
Pointers	Front ↓			Back ↓
Element	Dave	Sam	Mark	Carol

The status of the queue after enqueuing the next item with a list:

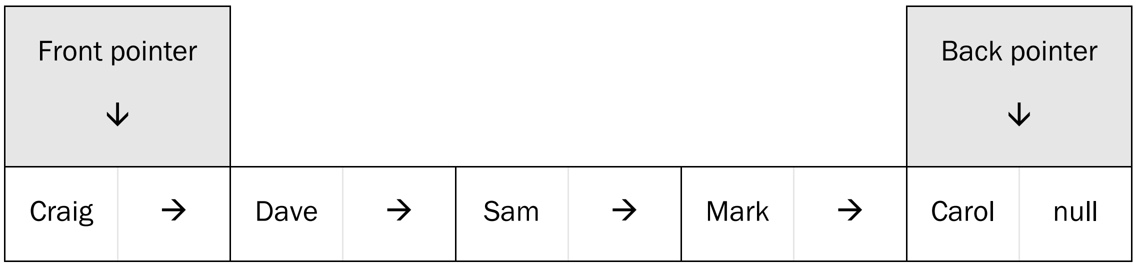
Index	0	1	2	3	4
Pointers	Front ↓				Back ↓
Element	Dave	Sam	Mark	Carol	Andy

Note how the number of indexes grows and shrinks as items are enqueued and dequeued.

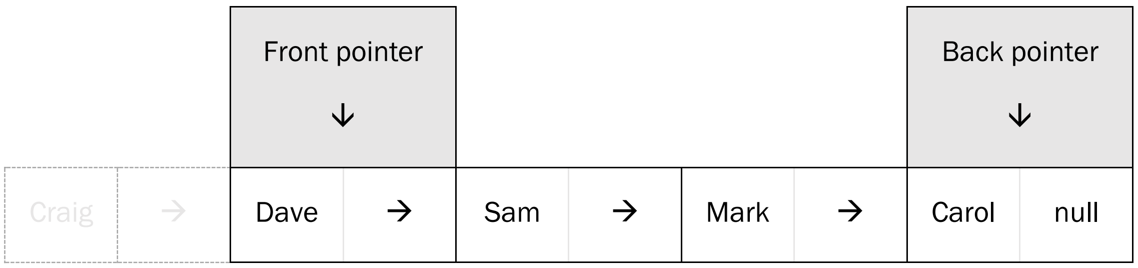
Object implementation of a queue

With an object implementation, memory is allocated from the heap to new items as necessary. Each object is a node, with each node pointing to the next. Front and back pointers are also maintained.

The status of a queue after five items have been enqueued with objects:

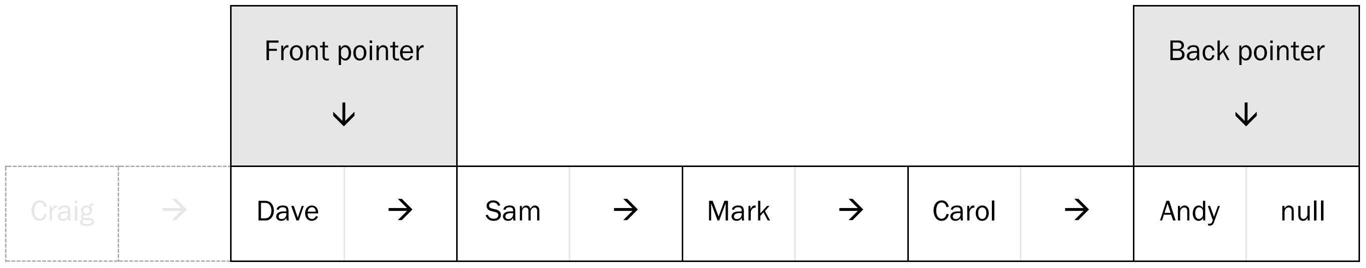


The status of the queue after dequeuing an item with objects:



The front pointer is changed to be the node pointed to by the front pointer. Although the node *Craig* will still exist in memory on the heap, it does not have an active reference, so it will be reclaimed by the garbage collection process.

The status of the queue after enqueuing an item with objects:



Available memory is taken from the heap and the back pointer is updated to point to the new item.

Enqueuing an item to a queue

1. Check for queue overflow. Output an error if no free memory is available.
2. Create a new node and insert data into it.
3. The back pointer is set to point to the new node.
4. If this is the first node in the list, the front pointer is set to point to the new node.

Pseudocode for enqueuing an item

```
If not memoryfull Then
      new_node = New Node
      If back_pointer = Null Then
            front_pointer = new_node
      Else
            Previous_back_node.pointer = new_node
      End If
      Back_pointer = new_node
End If
```

Dequeuing an item from a queue

1. Check for queue underflow. Output an error if the front pointer does not point to a node.
2. Output the node pointed to by the front pointer.
3. Set the front pointer to the previous item.

Pseudocode for dequeuing an item

```
If front_pointer != Null Then
      Output front_pointer.data
      front_pointer = front_pointer.pointer
      If front_pointer = Null Then back_pointer = Null
End If
```

Peeking an item in a queue

1. Output an error if the front pointer does not point to a node.
2. Output the node pointed to by the front pointer.

Pseudocode for peeking an item from a queue

```
If front_pointer != Null Then
      Output front_pointer.data
End If
```

Circular queue coded in Python using an array/list

```python
class Queue:
    max = 10
    items = ["" for index in range(max)]

    front_pointer = -1
    back_pointer = -1

    def enqueue(self, item):
        # Check queue overflow
        if (self.back_pointer + 1) % self.max != self.front_pointer:
            self.back_pointer = (self.back_pointer + 1) % self.max
            # Enqueue the item
            self.items[self.back_pointer] = item
            # Set first item if queue was empty
            if self.front_pointer == -1:
                self.front_pointer = 0
            return True
        else:
            return False

    def dequeue(self):
        # Check queue underflow
        if self.front_pointer != -1:
            # Dequeue the item
            item = self.items[self.front_pointer]
            # If the queue is not empty change the front pointer
            if self.front_pointer != self.back_pointer:
                self.front_pointer = (self.front_pointer + 1) % self.max
            else:
                # When the last item is dequeued reset the pointers
                self.front_pointer = -1
                self.back_pointer = -1
            return item
        else:
            return None

    def peek(self):
        # Check queue underflow
        if self.front_pointer != -1:
            # Peek the item
            return self.items[self.front_pointer]
        else:
            return None
```

```python
# Main program starts here
items = ["Florida", "Georgia", "Delaware", "Alabama", "California"]
q = Queue()
# Add items to the queue
for index in range(0, len(items)):
    q.enqueue(items[index])
# Remove items from the queue
print(q.dequeue())
# Output the next item in the queue
print(q.peek())
```

Queue coded in Python using objects

```python
class Queue:
    class Node:
        data = None
        pointer = None

    front_pointer = None
    back_pointer = None

    def enqueue(self, item):
        # Check queue overflow
        try:
            # Enqueue the item
            new_node = Queue.Node()
            new_node.data = item
            # Empty queue
            if self.back_pointer == None:
                self.front_pointer = new_node
            else:
                self.back_pointer.pointer = new_node
            self.back_pointer = new_node
            return True
        except:
            return False

    def dequeue(self):
        # Check queue underflow
        if self.front_pointer != None:
            # Dequeue the item
            item = self.front_pointer.data
            self.front_pointer = self.front_pointer.pointer
            # When the last item is dequeued reset the pointers
            if self.front_pointer == None:
                self.back_pointer = None
            return item
        else:
            return None

    def peek(self):
        # Check queue underflow
        if self.front_pointer != None:
            # Peek the item
            return self.front_pointer.data
        else:
            return None
```

```
# Main program starts here
items = ["Florida", "Georgia", "Delaware", "Alabama", "California"]
q = Queue()
# Add items to the queue
for index in range(0, len(items)):
    q.enqueue(items[index])
# Remove items from the queue
print(q.dequeue())
# Output the next item in the queue
print(q.peek())
```

Operations

Enqueue items:	q.enqueue(*item*)
Dequeue items:	print(q.dequeue())
Peeking an item:	print(q.peek())

Efficiency of operations on a queue

	Time complexity			Space complexity	
	Best case	Average case	Worst case	Best case	Worst case
Enqueue	O(1) Constant	O(1) Constant	O(1) Constant	O(1) Constant	O(n) Linear
Dequeue	O(1) Constant	O(1) Constant	O(1) Constant		
Is Empty, Is Full	O(1) Constant	O(1) Constant	O(1) Constant		
Peek	O(1) Constant	O(1) Constant	O(1) Constant		

A queue is a dynamic data structure. When implemented using object-oriented techniques, its memory footprint grows and shrinks as data is enqueued and dequeued from the structure. Using only the required amount of memory is the most efficient way of implementing a queue.

However, a queue can also be implemented using an array. In this case, you are creating a dynamic structure on top of a static data structure, so the memory footprint remains constant. This method is inefficient because, unless it is full, the queue will be reserving more memory than it needs.

Items are enqueued at the back of the queue and dequeued from the front. Peeking looks at the item at the front of the queue. Detecting if a queue is empty or full only requires examining the value of the pointers. Therefore, the time complexity of all operations on a queue is constant, O(1).

Record

A record, sometimes called a structure or compound data is a collection of related items – often of different data types – called fields. A record is like an object with attributes, although records do not require the object-oriented paradigm and do not have methods. Records are often thought of as special cases of objects, referred to as *plain old data structures* or *PODs*.

It is useful to define a fixed maximum size of each field in bytes when using string fields in records in order to minimise data storage requirements and speed up searches.

Applications of a record

The history of the record can be traced to the punched card, where columns on the card related to each field. In early programming applications, the record was an ideal structure for reading and writing data from secondary storage. The size of a record could be calculated by adding the fixed size of all the fields.

By using a unique numeric primary key field called a record number, the location of the record on the storage medium can be found by multiplying the record number by the record size. It is also possible to determine the record number using a hashing function.

Records were so common in early computing that there are special characters for delimiting fields and records (ASCII codes 28-31). If fixed-size data types are not used, these special characters can be used to easily identify where one data item ends and the next one begins.

Today, records are still used extensively as an alternative structure for groups of related variables and use the *with* statement in many programming languages. Records are also used to construct databases, although their operations are largely abstracted by database management systems. It is more common to connect a program to a DBMS if records need to be stored or use a data interchange format such as JSON or XML.

Operations on a record

Typical operations that can be performed on a record structure include:

- Declare: When using records, the structure of the record type is defined first, much like an object being declared in a class.
- Add/Construct: Make a new record using a set of specific values and possibly field names.
- Assign/Update: Attribute a value to a field.
- Hash: Compute a record number from the key.
- Search: Find a record by hashing the primary key, using a linear search or a linked list secondary key.
- Delete: Remove a record.
- Compare: Return whether two records are the same or one is greater than the other using lexicographic order.

Using a record structure

Before a record structure can be used, the fields must be defined. The syntax will vary between languages. In Visual Basic, a record declaration statement might look like this:

```
Structure Person
    <VBFixedString(6)> Public StaffCode As String
    <VBFixedString(20)> Public Forename As String
    <VBFixedString(20)> Public Surname As String
    <VBFixedString(10)> Public StaffType As String
End Structure
```

<VBFixedString(20)> means the *Forename* field will be of a fixed length of 20 characters – this is optional.

The *StaffCode* field is being used as the unique primary key.

The record structure must be declared before it can be used:

```
    Dim Employee As Person
```

However, it is typically more useful to have more than one record, so an array of records can be declared:

```
    Dim Employee(50) As Person
```

The records can then be used:

```
    Employee(1).StaffCode = "CRS"
    Employee(1).Forename = "Craig"
    Employee(1).Surname = "Sargent"
    Employee(1).StaffType = "Teacher"
    Employee(2).StaffCode = "DHI"
    Employee(2).Forename = "Dave"
    Employee(2).Surname = "Hillyard"
    Employee(2).StaffType = "Teacher"
```

Not all languages support the record data structure – Python being an example – but it can be simulated with objects or dictionaries quite easily.

For example, using a dictionary in Python, a record structure could be defined and declared as follows:

```
Employee = {
  "CRS": {"Surname": "Sargent", "Forename": "Craig", "StaffType": "Teacher"},
  "DAH": {"Surname": "Hillyard", "Forename": "Dave", "StaffType": "Teacher"},
}
```

Records can then be output...

```
    print(Employee["DAH"]["Surname"])
```

...and updated:

```
    Employee["CRS"]["StaffType"] = "Admin"
```

Efficiency of operations on a record

	Time complexity		
	Best case	Average case	Worst case
Add, Delete	O(1) Constant	O(1) Constant	O(n) Linear
Search	O(1) Constant	O(1) Constant	O(n) Linear
Compare	O(1) Constant	O(1) Constant	O(1) Constant

Space complexity	
Best case	Worst case
O(1) Constant	O(n) Linear

If there are a known number of records with a fixed field size, the memory footprint can be calculated, making the space complexity constant, O(1). However, record structures are frequently used in database applications where the number of records stored is variable. Although using fixed-length fields is more efficient for finding records in a data set, it is also possible to use variable-length fields, which require end-of-field markers – the memory footprint will then change as records are added to the data set, O(n).

Calculating the time complexity of a record depends on the algorithms being implemented. New records are usually added to the end of the data set because it is inefficient to insert them between existing records. Linked lists are often maintained as secondary keys in order to maintain a logical order of records. As records need to be appended, the time complexity is O(1).

When finding records with fixed-length fields, a hash function can be used on the key field to calculate the byte position of the required record, O(1). However, variable-length records will require a linear search; O(1) at best – if the record is the first in the data set – but usually O(n). The time complexity for both adding a record and searching the data set assumes that records are being held either in memory or on random-access secondary storage such as a hard disk or solid-state drive. In the past, records were frequently held on linear drives such as tape or punched cards, so the time complexity would always be linear, O(n).

Deleting records can be done in several ways. One approach is to flag the record as deleted but not actually remove it until the record set is maintained in a batch process. Alternatively, the record can be removed, but this is inefficient because all the other records would need to be moved to fill space in the data set. Both methods require searching for the record – O(1) with fixed-length records or O(n) with variable-length.

Stack

A stack is a dynamic data structure. Items are both *pushed* (added) onto the top of the stack and *popped* (removed) from the top of the stack. It is also possible to *peek* at the top item without removing it.

Imagine a stack of coins. A coin can be added or removed from the top but not the middle. The only way of accessing items in the stack is from the top.

A stack is known as a last-in, first-out or LIFO structure.

It has a stack pointer that always points to the item at the top – this is also called the top pointer.

Top →	Carol
	Mark
	Sam
	Dave
	Craig

Applications of a stack

Stacks are essential to the operation of a computer system. A stack frame is pushed onto a call stack when subroutines are called. The stack frame includes the values of the registers when the subroutine was called, parameters and local variables. Many compilers use a stack for parsing syntax expressions.

Stacks are used for depth-first traversal of graph data structures, undo operations that track user inputs and backtracking algorithms – for example, pathfinding maze solutions. Stacks are also used to evaluate mathematical expressions without brackets using a shunting yard algorithm and reverse Polish notation.

Did you know?

In 1946, Alan Turing first used the terms *bury* and *unbury* to describe calling and returning from subroutines. In 1955, Klaus Samelson and Friedrich Bauer filed a patent for the idea of a stack.

Storing a stack in memory

Array

```
top = 4
```

Index	Element
0	Carol
1	Mark
2	Sam
3	Dave
4	Craig
5	
6	

List

```
top = 0
```

Index	Element
0	Craig
1	Dave
2	Sam
3	Mark
4	Carol

Note that there is no free space with a list implementation because it is truly dynamic. New items are always added and removed from index 0.

Object

```
Class Stack
       stackPointer = None

  Class Node
       element = ""
       pointer = Node
  End Class

End Class
```

Operations on a stack

Typical operations that can be performed on a stack include:

- Push: adding an item to the top of the stack.
- Pop: removing an item from the top of the stack.
- Peek: return the value from the top of the stack without removing it.
- Is Empty: return whether the stack is empty.
- Is Full: return whether the stack is full.

An attempt to push an item onto a stack that is already full is called a stack overflow, while attempting to pop an item from an empty stack is called a stack underflow. Both of these situations should be considered before attempting to push or pop an item.

Array implementation of a stack

An array is declared large enough to accommodate the maximum number of items. A variable is used to hold the stack pointer, the index of the item at the top of the stack. This index is set to -1 when the stack is empty.

It feels counterintuitive to illustrate a stack with index 0 at the top, but as it is an abstraction, it doesn't matter. To avoid confusion, index 0 will be shown at the bottom of the structure.

The status of the stack after five items have been pushed using an array with seven indexes:

Pointer	Index	Element
	6	
	5	
Top →	4	Craig
	3	Dave
	2	Sam
	1	Mark
	0	Carol

The status of the stack after popping an item using an array:

Pointer	Index	Element
	6	
	5	
	4	Craig
Top →	3	Dave
	2	Sam
	1	Mark
	0	Carol

Note that *Craig* was not actually removed from the structure; instead, the pointer was moved.

The status of the stack after pushing the next item using an array:

Pointer	Index	Element
	6	
	5	
Top →	4	Andy
	3	Dave
	2	Sam
	1	Mark
	0	Carol

The status of the stack after pushing another item using an array:

Pointer	Index	Element
	6	
Top →	5	James
	4	Andy
	3	Dave
	2	Sam
	1	Mark
	0	Carol

Notice how previously popped items are overwritten when new items are pushed.

List implementation of a stack

When implementing a stack using a list, it is easier to push a new item to index 0 and move all the other elements down one index – this is not very efficient, O(n), but list methods can be used to handle the operation and make the implementation easier.

The status of a stack after five items have been pushed with a list:

Pointer	Index	Element
Top →	0	Craig
	1	Dave
	2	Sam
	3	Mark
	4	Carol

The status of the stack after popping an item with a list:

Pointer	Index	Element
Top →	0	Dave
	1	Sam
	2	Mark
	3	Carol

The status of the stack after pushing the next item with a list:

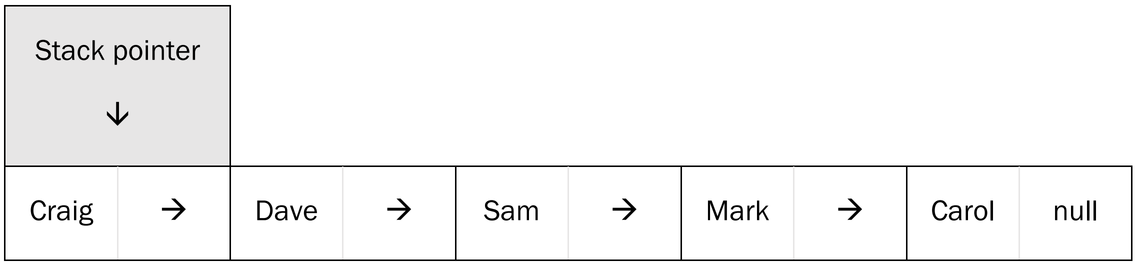
Pointer	Index	Element
Top →	0	Andy
	1	Dave
	2	Sam
	3	Mark
	4	Carol

Note how the number of indexes grows and shrinks as items are pushed and popped.

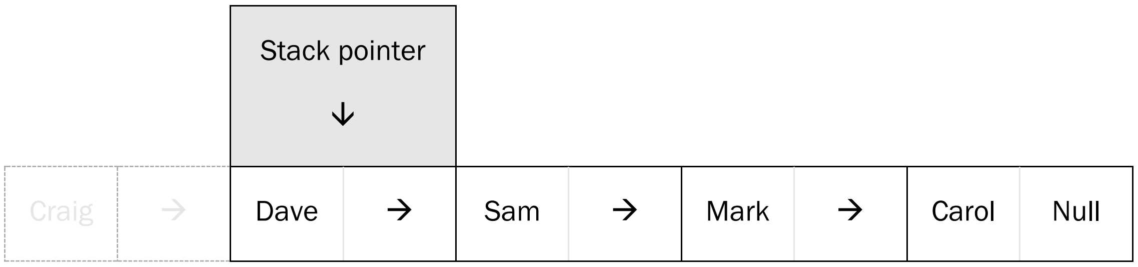
Object implementation of a stack

With an object implementation, memory is allocated to new items as necessary. Each object is a node, with each node pointing to the next. The stack pointer is still maintained.

The status of a stack after five items have been pushed with objects:

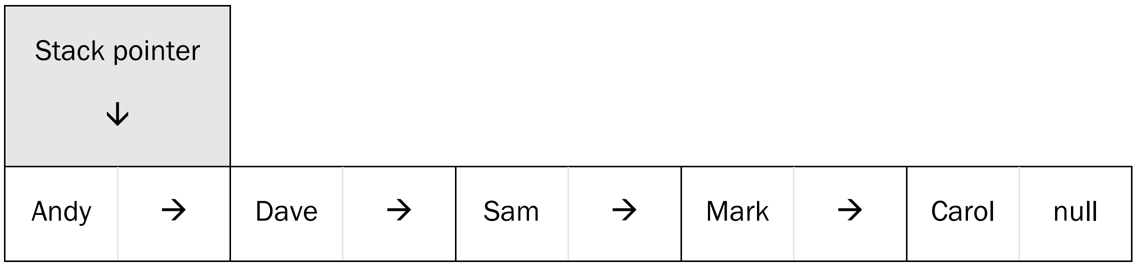


The status of the stack after popping an item with objects:



The stack pointer is changed to be the node pointed to by the top node. Although the node *Craig* will still exist in memory, it does not have an active reference, so it will be reclaimed by the garbage collection process.

The status of the stack after pushing an item with objects:



The new item is added, and the stack pointer is changed to point to the new node. The pointer on the new node points to the node previously pointed to by the stack pointer.

Pushing an item onto a stack

1. Check for stack overflow. Output an error if no free memory is available.
2. Create a new node and insert data into it.
3. The new node points to the previous node.
4. The stack pointer is set to point to the new node.

Pseudocode for pushing an item onto a stack

```
If not memoryfull Then
      new_node = New Node
      new_node.pointer = stack_pointer
      stack_pointer = new_node
End If
```

Popping an item from a stack

1. Check for stack underflow. Output an error if the stack pointer does not point to a node.
2. Output the node pointed to by the stack pointer.
3. Set the stack pointer to the previous item.

Pseudocode for popping an item from a stack

```
If stack_pointer != Null Then
      Output stack_pointer.data
      stack_pointer = stack_pointer.pointer
End If
```

Did you know?

Popping is sometimes called *pulling*.

In addition to pushing, popping and peeking items from a stack, a third operation called *rotate* or *roll* can be used to move items around a stack – e.g., A-B-C becomes B-C-A, where the first item has *rolled* around to become the last item.

Peeking an item from a stack

1. Output an error if the stack pointer does not point to a node.
2. Output the node pointed to by the stack pointer.

Pseudocode for peeking an item from a stack

```
If stack_pointer != Null Then
      Output stack_pointer.data
End If
```

Did you know?

Stacks can also be used in the architecture of a computer. The x87 floating point co-processor used by old x86 machines is one example. The registers inside the co-processor were organised as a stack, but direct access to individual registers was also possible.

Today these maths co-processors are integrated into the main CPU architecture.

Stack coded in Python using an array/list

```
class Stack:
    max = 10
    items = ["" for index in range(max)]

    stackPointer = -1

    def push(self, item):
        # Check stack overflow
        if self.stackPointer < self.max:
            self.stackPointer = self.stackPointer + 1
            # Push the item
            self.items[self.stackPointer] = item
            return True
        else:
            return False

    def pop(self):
        # Check queue underflow
        if self.stackPointer != -1:
            # Pop the item
            item = self.items[self.stackPointer]
            self.stackPointer = self.stackPointer - 1
            return item
        else:
            return None

    def peek(self):
        # Check stack underflow
        if self.stackPointer != -1:
            # Peek the item
            return self.items[self.stackPointer]
        else:
            return None


# Main program starts here
items = ["Florida", "Georgia", "Delaware", "Alabama", "California"]
s = Stack()
# Add items to the stack
for index in range(0, len(items)):
    s.push(items[index])
# Remove items from the stack
print(s.pop())
# Output the next item in the stack
print(s.peek())
```

Stack coded in Python using objects

```
class Stack:
    class Node:
        data = None
        pointer = None

    stack_pointer = None

    def push(self, item):
        # Check stack overflow
        try:
            # Push the item
            new_node = Stack.Node()
            new_node.data = item
            new_node.pointer = self.stack_pointer
            self.stack_pointer = new_node
            return True
        except:
            return False

    def pop(self):
        # Check stack underflow
        if self.stack_pointer != None:
            # Pop the item
            popped = self.stack_pointer.data
            self.stack_pointer = self.stack_pointer.pointer
            return popped
        else:
            return None

    def peek(self):
        # Check stack underflow
        if self.stack_pointer != None:
            # Peek the item
            return self.stack_pointer.data
        else:
            return None


# Main program starts here
items = ["Florida", "Georgia", "Delaware", "Alabama", "California"]
s = Stack()
# Add items to the stack
for index in range(0, len(items)):
    s.push(items[index])
# Remove items from the stack
print(s.pop())
# Output the next item in the stack
print(s.peek())
```

Operations

Adding items:	s.push("Colarado")
Deleting items:	print(s.pop())
Peeking an item:	print(s.peek())

Did you know?

The peek operation is not necessary. It can be implemented with a pop operation followed by a push operation.

Other operations on a stack can include *duplicate*, where the top item is popped and then pushed twice, and *swap* or *exchange*, where the top two items are popped and then pushed back to the stack in the opposite order.

While these additional operations do not feature in the specification, examples of them can make great examination questions to test your understanding of how a stack works.

Did you know?

Computers use a stack to store data when subroutines are called – this includes the values of the registers (so a program can resume when the subroutine ends), fixed-length local variables and memory address references to other data structures.

As the size of the stack frame is not fixed, it poses a security risk. When an oversized frame is written to the stack, the values of the registers stored in another stack frame can be overwritten, causing the program to branch unexpectedly, known as *stack smashing*. This is a technique used by viruses.

Efficiency of operations on a stack

	Time complexity		
	Best case	Average case	Worst case
Push	O(1) Constant	O(1) Constant	O(1) Constant
Pop	O(1) Constant	O(1) Constant	O(1) Constant
Is Empty, Is Full	O(1) Constant	O(1) Constant	O(1) Constant
Peek	O(1) Constant	O(1) Constant	O(1) Constant

Space complexity	
Best case	Worst case
O(1) Constant	O(n) Linear

A stack is a dynamic data structure. When implemented using object-oriented techniques, its memory footprint grows and shrinks as data is pushed onto and popped from the structure. Using only as much memory as needed is the most efficient way of implementing a stack.

However, a stack could also be implemented using an array. In this case, the dynamic structure has been created upon a static data structure, so the memory footprint remains constant. This method is inefficient because the stack will be reserving more memory than it needs unless it is full.

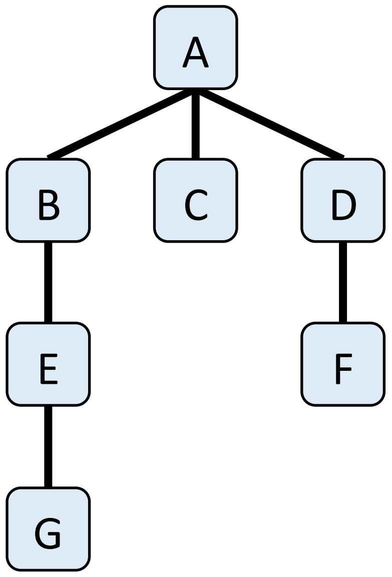
Items are always pushed onto and popped from the top of the stack. Peeking an item looks at the item at the stack pointer and detecting if a stack is full or empty only requires examining the value of the stack pointer. Therefore, the time complexity of all operations on a stack is constant, O(1).

Tree

A tree is a connected, undirected graph with no cycles. A tree may have a root node – known as a rooted tree – or no identifiable root node. Unlike a graph, a rooted tree is often used to define parent-child relationships between nodes and store hierarchical data.

There are many different types of tree structures including general trees, AVL trees, b-trees, binary trees, binary interval trees, cartesian trees, KD trees, search trees, quad trees, R-trees, red-black trees, splay trees, and treaps to name a few. For the purposes of examinations, it is sufficient for you to only understand the similarities and differences between general trees and binary search trees.

Unlike a binary tree, a general tree is not limited to a maximum of two child nodes. A node can have any number of children, but loops between nodes are not permitted and it is common for one node to have a single parent. Therefore, there is only one path to any one node.



Applications of a tree

General trees are used to model file systems, represent abstract syntax trees for program compilation, parse natural language and store geometric data. Interval trees are used in vector graphic processing to find visible elements inside three-dimensional space. Quad trees are used in image processing for compression algorithms, mesh generation and spatial partitioning for collision detection in games.

R-trees may be used to store objects on maps, allowing for queries such as, “Find all the banks within 5km of my location.” Splay trees are used to implement caching and garbage collection algorithms.

Storing a general tree in memory

Since trees are very similar to graphs, they can be stored in the same way using a dictionary or objects.

Operations on a tree

- Add: Add a new node at a certain position in the tree.
- Delete: Remove a node from the tree.
- Find ancestor: Find the common parent of one or more nodes.
- Graft: Add a whole section to a tree – e.g., copying a node and its children from another tree.
- Prune: Remove a node and all its children.
- Search: Find an item using a breadth- or depth-first search.

Did you know?

In graph theory, an undirected graph where any two vertices are connected by one edge at most is also known as a forest. In computer science, a forest is often defined as multiple disconnected trees. If you remove the root node of a binary tree that has two children, the result is a forest of two trees.

Think about the organisation of files on a computer, presented in a tree data structure with one folder connected to one or more sub-folders. The computer may also have multiple drives – e.g., C:\, D:\. This is an example of a forest, where each tree is a drive.

Efficiency of a tree

	Time complexity			Space complexity	
	Best case	Average case	Worst case	Best case	Worst case
Add, Delete	O(1) Constant	O(log n) Logarithmic	O(log n) Linear	O(1) Constant	O(n) Linear
Search	O(1) Constant	O(log n) Logarithmic	O(n) Linear		
Graft, Prune	O(1) Constant	O(n) Linear	O(n) Linear		

A tree is a dynamic data structure that grows and shrinks as necessary. If the items in a tree are fixed, the space complexity is O(1), but this is rarely the case.

When performing operations on a tree, one branch will be followed. There is only one route to a node because cycles are not permitted. Therefore, it may be possible to discount whole branches of a tree when an edge is followed, resulting in a divide-and-conquer algorithm of O(log n).

It could also be the case that the operation to be performed only requires accessing the root node of a rooted tree. In this best case, the time complexity will be constant, O(1) – but again, this is rarely the case.

Due to the nature of the tree, it may be necessary to use breadth- and depth-first algorithms – which are linear operations, O(n) – to find items.

Grafting and pruning trees may be straightforward if the new branches can be added to existing nodes, O(1). However, they may also require some restructuring of the tree, resulting in a linear operation, O(n).

In summary

Array	List	Object
Elements are all the same data type.	Elements can be different data types.	Attributes can be different data types.
Elements are stored in contiguous memory.	Elements are stored in contiguous memory.	Objects are not stored in contiguous memory.
Uses the index register.	Uses the index register.	Does not use the index register.
Static data structure.	Dynamic data structure.	Dynamic data structure.
Uses an index to set or retrieve an element at a specific address.	Uses an index to set or retrieve an element at a specific address.	Uses methods to get and set private attributes.
Does not use methods.	May use additional methods to perform operations on elements.	May use additional methods to perform operations on attributes.
Items are not added; vacant indexes are assigned with data elements.	When items are added, memory is reserved from the heap.	When new instances of objects are constructed, memory is reserved from the heap.
Indexes are not deleted; when data items are removed, the element is set to an empty string or null value.	When items are removed, other items fill the available space.	Objects that are not referenced with a pointer are reclaimed during garbage collection.
As elements are of the same data type, the address of each index can be calculated easily, making arrays efficient for arithmetic.	The address of each element cannot be calculated because they can hold data of different types, making lists inefficient.	Maintaining pointers to obsolete objects can cause memory leaks in algorithms, degrading their performance over time.

Dictionary
Static or dynamic data structure, depending on the implementation.
Uses key-value pairs. Values are stored at an index determined by a hashing function applied to the key.
Uses a hash table search or binary tree search to retrieve a value from a key.

Linked list
Dynamic data structure.
Elements can be different data types.
Elements are not stored contiguously.
Elements are accessed sequentially with a linear search.
May have special implementations, enabling it to be doubly linked or circular.

Stack	Queue
LIFO – last-in, first-out structure.	FIFO – first-in, first-out structure.
Items are inserted (pushed) onto and deleted (popped) from the top of the structure.	Items are inserted (enqueued) to the back of the structure and removed (dequeued) from the front.
One pointer at the top of the structure.	Two pointers – one at the back of the structure where new data items are added and one at the front where data items are removed.

Graph	Binary tree	Tree
No root vertex.	Has a root node.	May have a root node, but it is not essential.
Vertices can have any number of edges.	Nodes can only have zero, one or two child nodes.	Nodes can have zero or any number of child nodes.
There may be many paths to a vertex. Loops are permitted between vertices.	Only one path to a node.	Only one path to a node.
Any vertex can be a starting point for breadth- and depth-first traversal or search.	All operations start from the root node.	Operations can start from any node.
Used to represent related data.	Can be used to implement binary searches using a binary search tree and many other data structures including dictionaries.	Used to represent hierarchical related data.

SEARCHING ALGORITHMS

Routines that find data within a data structure.

Binary search

The binary search is an efficient algorithm for finding an item in a sorted data set. It repeatedly checks the middle item in the data set and discards half the data during each pass until either the item is found at the middle position, it is the only item left to check or it is not found.

The binary search can be performed on an array, list or binary search tree, implemented with either arrays, lists or objects. For the algorithm to work, the data set must be sorted – this can be achieved by inserting new items into their correct place, by applying a sorting algorithm to an array, or as a consequence of storing the data in a binary search tree.

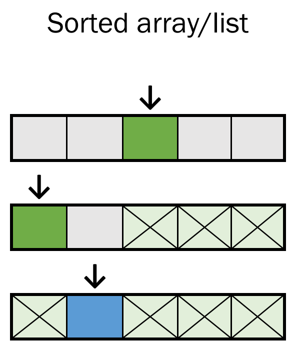
Applications of a binary search

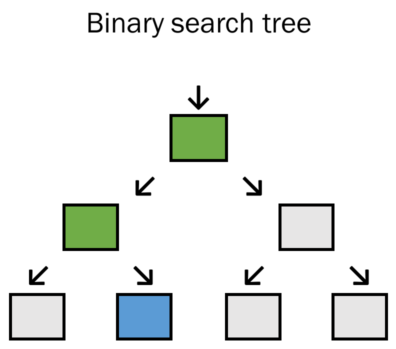
The binary search is an efficient algorithm, providing the data is already sorted. It is most typically used with binary search trees on large data sets. Compared to alternative searching algorithms, the binary search is particularly useful for range searches – for example, returning all the items greater than *x*.

Binary search in simple-structured English

1. Start at the middle item in the list.
2. If the middle item is the one to be found, the search is complete.
3. If the item to be found is lower than the middle item, discard all items to the right.
4. If the item to be found is higher than the middle item, discard all items to the left.
5. Repeat from step 2 until the item is found or there are no more items in the list.
6. If the item has been found, output its data. If it has not, output "Not found".

Visualising a binary search





Stepping through an example of a binary search on a sorted array

Searching for California:

Index:	0	1	2	3	4
Step 1	Calculate the middle as the first index (0) + the last index (4) integer division by 2 = 2: Delaware				
	Alabama	California	Delaware	Florida	Georgia
Step 2	Delaware is not the item to be found. California is lower. Discard all items to the right.				
	Alabama	California	Delaware	Florida	Georgia
Step 3	Calculate the middle as the first index (0) + the last index (1) integer division by 2: = 0: Alabama				
	Alabama	California	Delaware	Florida	Georgia
Step 4	Alabama is not the item to be found. California is higher. Discard all items to the left.				
	Alabama	California	Delaware	Florida	Georgia
Step 5	Calculate the middle as the first index (1) + the last index (1) integer division by 2 = 1: California				
	Alabama	California	Delaware	Florida	Georgia

Three comparisons were needed to find the item.

Note how the number of items to be checked is halved after each comparison – this is what makes the binary search so efficient, but it also explains why the items must be in order for the algorithm to work.

Use of integer division to find the middle point

In this example, the middle position was calculated as first index + last index integer division by 2 – this ensures that the result of the division is always an integer (whole number) because dividing odd numbers by two causes a fractional component of 0.5. The number is rounded down, but this is not important. Another acceptable implementation could round the result of the division up instead. It just feels less intuitive for experienced programmers to do this because it requires an additional function.

Stepping through an example of a binary search on a binary search tree stored as an array

Searching for Alabama:

Index:	0	1	2	3	4	5	6
Step 1	Start at the root node. Index 0: Delaware						
	Delaware	California	Florida	Alabama			Georgia
Step 2	Delaware is not the item to be found. Alabama is lower. Follow the left pointer (2 * 0) + 1 = Index 1						
	Delaware	California	Florida	Alabama			Georgia
Step 3	California is not the item to be found. Alabama is lower. Follow the left pointer (2 * 1) + 1 = Index 3						
	Delaware	California	Florida	Alabama			Georgia

For an explanation of how binary search trees are stored in arrays, see the *Binary tree* section of the chapter on *Data structures*.

An object implementation

The binary tree can also be stored using objects. The algorithm is the same, but instead of calculating the index of the next item, each node has attributes for the left and right pointers. Each of those attributes stores the address of the object it is connected to. For example, imagine the items are being stored at memory addresses 0xA0, 0xB3, 0xB6, 0xC8 and 0xCF. The memory map for the binary tree would look like this:

Address:	0xA0
Item:	Delaware
Left pointer:	0xB3
Right pointer:	0xB6

Address:	0xB3
Item:	California
Left pointer:	0xC8
Right pointer:	null

Address:	0xC8
Item:	Alabama
Left pointer:	null
Right pointer:	null

Address:	0xB6
Item:	Florida
Left pointer:	null
Right pointer:	0xCF

Address:	0xCF
Item:	Georgia
Left pointer:	null
Right pointer:	null

In this case, to find Alabama, the algorithm would begin at the start pointer, which would point to the root node at address 0xA0. The item is compared, and the left pointer is followed to address 0xB3, then 0xC8.

Pseudocode for the binary search using an array/list

```
Function binarySearch(items, item_to_find)
    found = False
    first = 0
    last = items.Length -1
    While first <= last and found == False
        midpoint = (first + last) DIV 2
        If items[midpoint] == item_to_find then
            found = True
        Else
            If items[midpoint] < item_to_find then
                first = midpoint + 1
            Else
                last = midpoint – 1
            End If
        End If
    End while
    If found == True then
        Return "Item found at position ", midpoint
    Else
        Return "Item not found"
    End If
End function
```

Did you know?

The binary search is also known as the half-interval search, logarithmic search or binary chop. There are many different types of binary search including uniform, exponential, interpolation, fractional cascading, noisy and quantum.

First proposed by Bernard Chazelle and Leonidas Guibas in 1986, fractional cascading uses multiple arrays or lists to optimise the search for frequently searched items. Thankfully, you only need to know about the classic binary search for examinations.

Binary search coded in Python using an array/list

```python
def binary_search(items, item_to_find):
    found = False
    first = 0
    last = len(items) - 1
    # Repeat until the item is found or no items are left to check
    while first <= last and not found:
        # Calculate the mid point using integer division
        midpoint = (first + last) // 2
        # Item found
        if items[midpoint] == item_to_find:
            found = True
        else:
            # Recalculate the mid point
            if items[midpoint] < item_to_find:
                first = midpoint + 1
            else:
                last = midpoint - 1
    if found:
        print("Item found at position", midpoint)
    else:
        print("Item not found")


items = ["Alabama", "California", "Delaware", "Florida", "Georgia"]
item_to_find = input("Enter the state to find: ")
binary_search(items, item_to_find)
```

Did you know?

As the binary search requires you to calculate the mid-point, the algorithm can fail in certain situations. `Midpoint = (first + last) DIV 2` can result in an arithmetic overflow if the result is greater than the maximum value for an integer – this occurs when the number of items in the data structure is larger than the maximum number the computer can store. For a 32-bit signed integer, that's 2,147,483,6437.

The Java programming language failed to trap this error in a library providing a binary search function for more than nine years, causing programs to crash unexpectedly.

Efficiency of a binary search

	Time complexity			Space complexity
	Best case	Average case	Worst case	
Sorted array	O(1) Constant	O(log n) Logarithmic	O(log n) Logarithmic	O(1) Constant
Binary search tree	O(1) Constant	O(log n) Logarithmic	O(n) Linear	O(1) Constant

A binary search will usually be more efficient than a linear search and less efficient than a hash table search. The biggest disadvantage of a binary search is that data items must be sorted for the algorithm to work – this can be achieved logically when items are added to a binary search tree by maintaining an order to the items in an array/list or using a searching algorithm.

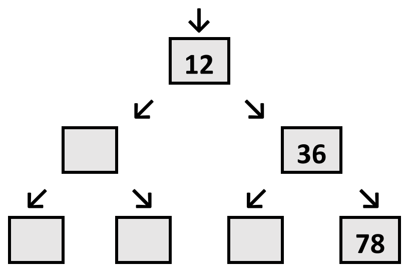
Given that maintaining the order to items in an array requires a linear algorithm O(n), this is not usually an efficient approach unless items are added infrequently and searches are performed very frequently. It is typically better to use a binary search tree implemented with an array (although that can result in a lot of unused indexes) or the most optimised object-based approach in terms of both time and space complexity.

In the best case, the item to be found is either in the middle position of an array or at the root node of a binary search tree. In this special case, the algorithm has a time complexity of O(1) since the item to be found will always be the first item checked – however, this is not usually the case.

In most cases, the time it takes to find an item increases with the size of the data set, but because half the items can be discarded at a time, the algorithm is usually logarithmic, O(log n).

Unbalanced binary tree

If a binary tree is used to store the data, it is possible that an unbalanced tree could be created:



In a worst-case scenario, finding number 78 would require checking all the items, increasing the time complexity to O(n) – this does not usually happen, and if a binary tree is to be used, it should be balanced.

Did you know?

Careful consideration of all the nuances of an algorithm and its data structure results in higher-level responses to examination questions such as, "Compare and contrast the linear and binary search." You will want to consider:

- The underlying data structure – e.g., an array or objects.
- The implementation of the data structure – e.g., a binary search tree stored as an array or objects and how such a choice affects space complexity.
- The number of items in the data set and how it affects time complexity.
- The position of the item to be found in the data set and how it affects the best-, worst- and average-case scenarios.
- Maintaining the data structure as items are added and deleted or the need for a sorting algorithm – and how this affects the usefulness of the algorithm.

Hash table search

The goal with a hash table search is to immediately find an item without needing to compare other items in the data set first – this makes it the most efficient searching algorithm in most cases. Often called a hash map, a hash table search is also how programming languages implement a dictionary data structure. A hashing function is used to calculate the position of an item in a hash table.

Applications of a hash table search

Hash tables are used in situations where items in a large data set need to be found quickly – for example, looking up vehicle details from a car registration plate using the police ANPR system. Hash tables are also used to create a symbol table during program compilation.

The Rabin-Karp algorithm uses a hash table search to find pattern matches in strings – for example, detecting plagiarism in student essays. Search engine indexes are stored and searched using a hash table and, as the name suggests, the dictionary compression algorithm also uses a hash table search.

Hashing functions

A hashing function is applied to a key to determine a hash value – the position of the item in a hash table. There are many different hashing functions in use today. A simple example might be to add up the ASCII values of all the characters in a string and calculate the modulus of that value by the size of the hash table.

Assuming a table size of 10:

Florida: F = 70, l = 108, o = 111, r = 114, i = 105, d = 100, a = 97
70 + 108 + 111 + 114 + 105 + 100 + 97 = 705
705 mod 10 = 5
The position of Florida in the table is 5.

A hash table needs to be at least large enough to store all the data items but is usually significantly larger to minimise the chance of returning the same value for more than one item, known as a collision:

Delaware: D = 68, e = 101, l = 108, a = 97, w = 119, a = 97, r = 114, e = 101
68 + 101 + 108 + 97 + 119 + 97 + 114 + 101 = 805
805 mod 10 = 5
The position of Delaware in the table is 5.

Since two items cannot occupy the same position in a hash table, a collision has occurred.

Properties of good hashing functions

A good hashing function should:

1. Be calculated quickly.
2. Result in as few collisions as possible.
3. Use as little memory as possible.

Resolving collisions

There are many strategies to resolve collisions generated from hashing functions. A simple solution is open addressing, repeatedly checking the next available space in the hash table until an empty position is found and storing the item in that location. To find the item later, the hashing function delivers the start position from which a linear search can be applied until the item is found – this is known as linear probing:

1	2	3	4	5	6	7	8	9	10
Alabama	Georgia			Florida →	Delaware →	California			

In this example, we can see that Delaware has a hash value of 5, but that position is occupied by Florida. Delaware is therefore placed at 6, the next available position. California has a hash value of 6 but cannot occupy its intended position, so it must be stored at the next available position, 7.

A disadvantage of this form of linear probing is that it prevents other items from being stored at their correct locations in the hash table. It also results in what we refer to as clustering, several positions being filled around common collision values.

Notice that with a table size of ten, two collisions occur. With a table size of five, three collisions occur, resulting in a less efficient algorithm but a reduced memory footprint. If the table size is increased to eleven, no collisions occur. With hashing algorithms, there is often a trade-off between the efficiency of the algorithm and the size of the hash table.

A potential solution to clustering is to skip several positions before storing the item, resulting in more even distribution throughout the table. A simple approach would be to skip to every third item. A variation of this known as quadratic probing increases the number of items skipped with each jump – e.g., 1, 4, 9, 16.

It may also be necessary to increase the size of the hash table in the future and recalculate new positions for the items. The process of finding an alternative position for items in a hash table is known as rehashing.

An alternative method of handling collisions is to use a two-dimensional hash table, making it possible for more than one item to be placed at the same position – this is called chaining:

	1	2	3	4	5	6	7	8	9	10
0	Alabama	Georgia			Florida	California				
1					Delaware					

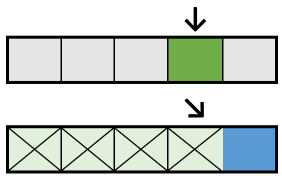
In this example, we can see that both Florida and Delaware are occupying the same position but different elements of a two-dimensional array. Another possibility would be to use a second table for collisions, referred to as an overflow table:

1	2	3	4	5	6	7	8	9	10
Alabama	Georgia			Florida	California				
0	1	2	3	4	5	6	7	8	9
Delaware									

Hash table search in simple-structured English

1. Calculate the position of the item in the hash table using a hashing function.
2. Check if the item at that position is the item to be found.
3. If it is not, move to the next item.
4. Repeat from step 2 until the item is found or there are no more items in the hash table.
5. If the item has been found, output its data. If it has not, output "Not found".

Visualising a hash table search



Stepping through an example of a hash table search

Searching for California:

Index:	1	2	3	4	5	6	7	8	9	10
Step 1	Calculate the hash value for California: C = 67, a = 97, l = 108, l = 105, f = 102, o =111, r =114, n = 110, l = 105, a = 97 67 + 97 + 108 + 105 + 102 + 111 + 114 + 110 + 105 + 97 = 1016 1016 mod 10 = 6. Compare California to Delaware. Delaware is not the item to be found – move to the next item in the hash table.									
	Alabama	Georgia			Florida	Delaware	California			
Step 2	Compare California to California. Item found.									
	Alabama	Georgia			Florida	Delaware	California			

Two comparisons were required to find the item.

Pseudocode for the hash table search

```
Function hashSearch(items, item_to_find)
      hash = calculate_hash(item_to_find)
      If hash_table(hash) != "":
            If hash_table(hash) == item_to_find:
                  Found = True
            Else
                  Do While hash < hash_table.Length and not found:
                        If hash_table(hash) != item_to_find:
                              hash = hash + 1
                        Else
                              found = True
                        End If
                  End While
            End If
      End If
      If found == True then
            Return "Item found at position ", index
      Else
            Return "Item not found"
      End If
End function
```

Did you know?

The efficiency of a hash table search is reliant on the suitability of the hashing function. If the data set is known in advance and does not change, it is possible to construct a perfect hash function that guarantees constant time complexity without any collisions.

Hash table search coded in Python using an array/list

```python
class HashTable:
    array_table = []

    def hashing_function(self, item, table_size):
        # Simple hashing algorithm adds ascii values of characters modulus table size
        total = 0
        for character in range(len(item)):
            total = total + ord(item[character])
        return total % table_size

    def create_table(self, items, table_size):
        # Reserve memory for hashing table
        for counter in range(table_size):
            self.array_table.append("")

        # Place data into hashing table
        for item_index in range(len(items)):
            table_index = self.hashing_function(items[item_index], table_size)
            if self.array_table[table_index] != "":
                print("Collision inserting", items[item_index], "at", table_index)
                # On collision insert in next available space
                while self.array_table[table_index] != "":
                    table_index = table_index + 1
            self.array_table[table_index] = items[item_index]
            print("Inserted", items[item_index], "at index", table_index)
        return self.array_table

    def search(self, item_to_find):
        found = False
        index = self.hashing_function(item_to_find, len(self.array_table))
        # Attempt to find item at hash value
        if self.array_table[index] != "":
            if self.array_table[index] == item_to_find:
                found = True
            else:
                # Degrade to linear search for collisions
                while index < len(self.array_table) and not found:
                    if self.array_table[index] != item_to_find:
                        index = index + 1
                    else:
                        found = True
        if found:
            print("Item found at position", index)
        else:
            print("Item not found")


# Main algorithm starts here
items = ["Florida", "Georgia", "Delaware", "Alabama", "California"]
# The larger the table the fewer collisions
store = HashTable()
table_size = 10
store.create_table(items, table_size)
item_to_find = input("Enter the state to find: ")
store.search(item_to_find)
```

Efficiency of a hash table search

Time complexity			Space complexity
Best case	Average case	Worst case	
O(1) Constant	O(1) Constant	O(n) Linear	O(1) Constant

On average, the hash table search outperforms a linear or binary search. It does not require the data to be sorted – instead, it finds the position of an item immediately using a hashing function, resulting in a constant time complexity of O(1) in both best and average cases.

However, if the item cannot be found immediately because of a collision, linear probing must be used instead – this results in a time complexity of O(n) because each item must be checked in turn until either the item is found or the end of the table is reached.

With hash table searching, it is important to use an efficient hashing function that will calculate quickly but also produce as few collisions as possible – this is often achieved using a larger hash table than required to store the data, increasing the number of unique positions and reducing the number of possible collisions.

Did you know?

It is also a good idea to swap data items where collisions occur so that the more frequently searched item is stored in the location determined by the hashing function. This approach is known as Robin Hood hashing.

Linear search

The linear search finds an item in a sorted or unsorted list. A linear search starts at the first item in the list and checks each item one by one. Think about searching for a card in a shuffled deck, starting with the top card and checking each one until you find the card you want.

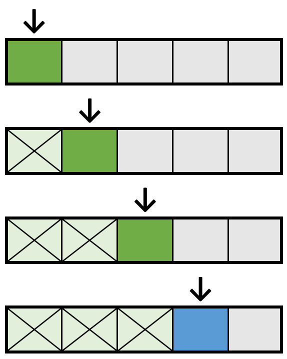
Applications of a linear search

The linear search is ideal for finding items in small data sets and performing searches with unordered data such as settings files. It is the easiest searching algorithm to implement but usually the most inefficient.

Linear search in simple-structured English

1. Start at the first item in the list.
2. If the item in the list is the one to be found, the search is complete.
3. If it is not, move to the next item.
4. Repeat from step 2 until the item is found or there are no more items in the list.
5. If the item has been found, output its data. If it has not, output “Not found”.

Visualising a linear search



Did you know?

Linear searches can also be performed on serial files. If you are asked to perform a linear search on a file, don’t forget to include a line of pseudocode to open the file before the algorithm and to close the file afterwards – it may be worth a mark.

Stepping through an example of a linear search

Searching for California:

Index:	0	1	2	3	4
Step 1	Start at the first item in the list. Compare California to Florida. Florida is not the item to be found – move to the next item in the list.				
	Florida	Georgia	Delaware	Alabama	California
Step 2	Compare California to Georgia. Georgia is not the item to be found – move to the next item in the list.				
	Florida	Georgia	Delaware	Alabama	California
Step 3	Compare California to Delaware. Delaware is not the item to be found – move to the next item in the list.				
	Florida	Georgia	Delaware	Alabama	California
Step 4	Compare California to Alabama. Alabama is not the item to be found – move to the next item in the list.				
	Florida	Georgia	Delaware	Alabama	California
Step 5	Compare California to California. Item found.				
	Florida	Georgia	Delaware	Alabama	California

Five comparisons were required to find the item.

Did you know?

It was once thought that an unstructured search could never perform better than O(n) on average. However, Grover's algorithm and later modifications make O(√n) a possibility with quantum computers.

Pseudocode for the linear search

```
Function linearSearch(items, item_to_find)
    index = 0
    found = False
    While found == False and index < items.Length
        If items[index] == item_to_find then
            found = True
        Else
            index = index + 1
        End If
    End while
    If found == True then
        Return "Item found at position ", index
    Else
        Return "Item not found"
    End If
End function
```

Linear search coded in Python using an array/list

```python
def linear_search(items, item_to_find):
    index = 0
    found = False
    # Check every item until found or until there are no more items to check
    while not found and index < len(items):
        if items[index] == item_to_find:
            found = True
        else:
            index = index + 1
    if found:
        print("Item found at position", index)
    else:
        print("Item not found")


# Main program starts here
items = ["Florida", "Georgia", "Delaware", "Alabama", "California"]
item_to_find = input("Enter the state to find: ")
linear_search(items, item_to_find)
```

Efficiency of a linear search

Time complexity			Space complexity
Best case	Average case	Worst case	
O(1) Constant	O(n) Linear	O(n) Linear	O(1) Constant

A linear search is only efficient on small, unsorted data sets. It is usually inferior to a binary or hash table search, but it does benefit from not requiring the data to be sorted and has the smallest memory footprint.

In the best case, the item to be found is the first in the data set. In this situation, the linear search performs as well as a binary search when the first item is in the middle of the list, O(1) – that means the linear search can perform better than a binary search on very small lists.

In the worst case, the item to be found is last on the list or not in the list at all, so all the items need to be checked, O(n). Typically, the item to be found will be somewhere in the data set and, as the data set grows, more searching must be performed. Therefore, the algorithm has a complexity of O(n).

Did you know?

Although inefficient, the linear search is the only option if items need to be found in an unordered data set.

If certain items are likely to be searched for more frequently, it would also be better to place them towards the beginning of the list to maximise the efficiency of the algorithm.

When programming a linear search, use a WHILE statement if only one occurrence of an item needs to be found and a FOR statement if all occurrences need to be found.

Did you know?

Cache is high speed memory located inside the CPU. If the number of items in the data set is less than 16,000, the entire data set can probably be held in the L1 cache. If the number of items is less than 64,000, the entire data set can probably be held in the L2 cache.

Even though the linear search is inefficient, it can be performed so quickly that it likely won't matter. Any data that can be processed within a couple of hundred milliseconds is considered small – and you'd be surprised how much data that is with modern processors. Benchmarks have shown it is possible to execute a linear search on over 10 million items in less than 200ms.

Many optimisations of algorithms include caching the results of calculations or frequently required data. If the number of data items is less than 100, the inefficient linear search executes so quickly that it can be more expensive to store and retrieve the result instead of searching on demand. Optimisations often only become worthwhile if the algorithm needs to be run several thousand times.

Modern CPUs have single instruction, multiple data (SIMD) capabilities, enabling parallel operations where the same instruction can be carried out on multiple memory locations simultaneously. With just four cores, a small data set for a linear search can become as many as 200 million data items.

A final consideration when optimising algorithms includes the human factor. Sometimes, optimising approaches to code can take time and are also more complex than the solutions they might replace. Premature optimisation has its own cost in terms of development time.

Even if it is not optimal, any algorithm that can execute quickly is usually good enough, providing it is not required for critical, real-time applications or maximising frame rate. The surprising conclusion is that although the linear search is considered inefficient, it is often a suitable solution.

In summary

Binary search	Hash table search	Linear search
Items must be in order for the algorithm to work.	Items do not need to be in order.	Items do not need to be in order.
Start at the middle item.	Uses a hashing function on the key to determine the item index.	Start at the first item.
Halve the item set after each comparison until the item is found or there are no more items.	Hashing function delivers the item's location unless it doesn't exist or there is a collision.	Search each item in sequence until the item is found or there are no more items to check.
Can be implemented using an array or binary search tree.	Can be implemented with an array or list. Used to implement a dictionary.	Can be implemented using an array, list or linked objects.
Suitable for a large number of items and range searches.	Suitable for a large number of items but not range searches.	Only suitable for a small number of items.
Requires $\log_2 n + 1$ comparisons.	Usually requires one comparison or $<n$ in the worst case.	Requires n comparisons.
	On average, the fastest of the searching algorithms.	On average, the slowest of the searching algorithms.

SORTING ALGORITHMS

Routines that organise data in fundamental data structures.

Bubble sort

The bubble sort orders a data set by comparing each item with the next one and swapping the items if they are out of order. Comparing each item with its neighbour until the end of the data set is reached is known as a pass. If a swap is made at any time during a pass, the algorithm must start again with a new pass until no more swaps are made. The largest (or smallest) item *bubbles up* to the end of the data set with each pass.

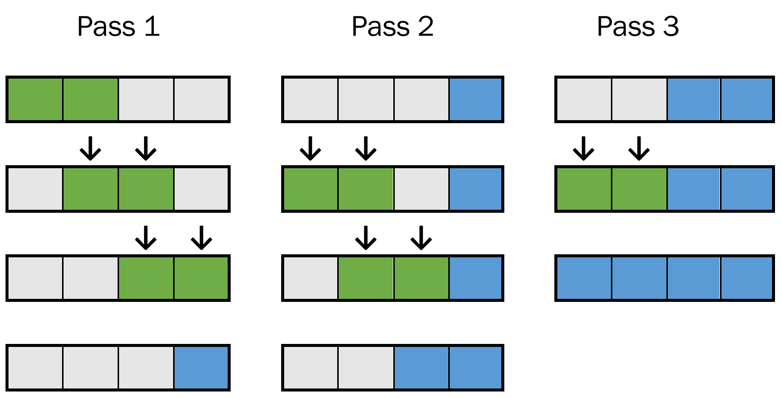
Applications of a bubble sort

The bubble sort is usually the most inefficient sorting algorithm but easy to implement, so it is a popular choice for very small data sets. It is ideal for situations that require a simple, easy-to-program sorting algorithm.

Bubble sort in simple-structured English

1. Start at the first item in the list.
2. Compare the current item with the next item.
3. If the two items are in the wrong position, swap them.
4. Move up one item to the next item in the list.
5. Repeat from step 2 until all the unsorted items have been compared.
6. If any items were swapped, repeat from step 1. Otherwise, the algorithm is complete.

Visualising a bubble sort



Stepping through an example of a bubble sort

Index:	0	1	2	3	4
Step 1	Compare Florida and Georgia – no swap required.				
	→Florida	Georgia←	Delaware	Alabama	California
Step 2	Compare Georgia and Delaware – swap the items.				
	Florida	→Georgia	Delaware←	Alabama	California
Step 3	Compare Georgia and Alabama – swap the items.				
	Florida	Delaware	→Georgia	Alabama←	California
Step 4	Compare Georgia and California – swap the items.				
	Florida	Delaware	Alabama	→Georgia	California←
Step 5	Georgia has bubbled to the end of the list. At least one swap was made, so start the algorithm again. Compare Florida with Delaware – swap the items.				
	→Florida	Delaware←	Alabama	California	Georgia
Step 6	Compare Florida with Alabama – swap the items.				
	Delaware	→Florida	Alabama←	California	Georgia
Step 7	Compare Florida with California – swap the items.				
	Delaware	Alabama	→Florida	California←	Georgia
Step 8	Florida has bubbled to the end of the list. At least one swap was made, so start the algorithm again. Compare Delaware with Alabama – swap the items.				
	→Delaware	Alabama←	California	Florida	Georgia

Step 9	Compare Delaware with California – swap the items.				
	Alabama	→Delaware	California←	Florida	Georgia
Step 10	Delaware has bubbled to the end of the list. At least one swap was made, so start the algorithm again. Compare Alabama with California – no swap required.				
	→Alabama	California←	Delaware	Florida	Georgia
Step 11	No swaps were made – the list is sorted.				
	Alabama	California	Delaware	Florida	Georgia

Pseudocode for the bubble sort

```
Function bubbleSort(items)
      n = items.Length
      swapped = True
      While n > 0 AND swapped
            swapped = False
            n = n - 1
            For index = 0 TO n - 1
                  If items[index] > items[index+1] then
                        Swap(items[index], items[index+1])
                        swapped = True
                  End If
            End For
      End while
      Return items
End function
```

Did you know?

The bubble sort is also called the sinking sort. Although it is regarded as an inefficient sorting algorithm, it is a good solution if a data set is almost sorted already. It has an advantage over the merge sort and quicksort because it can detect when the sort is complete, making it more efficient in some situations.

Bubble sort coded in Python using an array/list

```python
def bubble_sort(items):
    n = len(items)
    swapped = True
    # Start a new pass until no swap is made
    while n > 0 and swapped:
        swapped = False
        # Last item has bubbled to the top and no longer needs to be checked
        n = n - 1
        # Compare all items except those already sorted
        for index in range(0, n):
            if items[index] > items[index + 1]:
                temp = items[index]
                items[index] = items[index + 1]
                items[index + 1] = temp
                swapped = True
    return items


# Main program starts here
items = ["Florida", "Georgia", "Delaware", "Alabama", "California"]
print(bubble_sort(items))
```

Optimisations

Note that the number of items being compared is reduced by one with each pass. Once items have bubbled to their correct position, there is no need to compare them again – only items that could potentially be swapped are considered, making the bubble sort more efficient. However, it will still work if all items are compared in each pass, albeit less efficiently.

Another feature of the bubble sort is to nest a counter-controlled loop (FOR statement) inside a condition-controlled loop (WHILE statement) so the algorithm can terminate early if no swaps are made during a pass, further increasing its efficiency. It will still work with two counter-controlled loops – but again, less efficiently.

Did you know?

There have been attempts to improve the efficiency of the bubble sort – one includes reversing the direction of the algorithm after each iteration, known as a cocktail sort. The comb sort is another method that compares non-adjacent items. At best, a comb sort can perform as well as a quicksort.

Efficiency of a bubble sort

Time complexity			Space complexity
Best case	Average case	Worst case	
$O(n)$ Linear	$O(n^2)$ Polynomial	$O(n^2)$ Polynomial	$O(1)$ Constant

In the best case, the data set is already sorted, in which case, only one pass is required to check all the items in the list and no swaps will be made, so it is of linear complexity, $O(n)$ – this is not usually the case, as the purpose of the algorithm is to sort data.

As the algorithm contains a nested loop, it has polynomial time complexity, $O(n^2)$, because the time it takes to execute both iterations increases with the size of the data set.

However, it does not require any additional memory, making it an in-place sort. It can be performed on the data structure containing the data set, so the space complexity is $O(1)$.

The most efficient method of implementing a bubble sort is to stop the algorithm when no swaps are made.

The bubble sort was ideal in the early days of computing when data was stored on tape drives and computers had very little RAM. Rewinding and fast-forwarding a tape was so slow that executing a random-access algorithm was impractical. If possible, you would always want to process data sequentially with no more than two records at a time being compared.

Did you know?

It is a misconception to assume the bubble sort is always the least efficient sorting algorithm. There are certain edge-case situations where one algorithm may outperform another. In the case of the bubble sort, that situation would be when the data set is already sorted – however, this cannot be known in advance.

Big O notation considers the worst case, where the bubble sort is outperformed by other algorithms. On the other hand, Big Ω (Omega) considers the best case, where the bubble sort may have an advantage.

Insertion sort

The insertion sort inserts items into their correct position amongst previously placed items. It is a useful algorithm for small data sets.

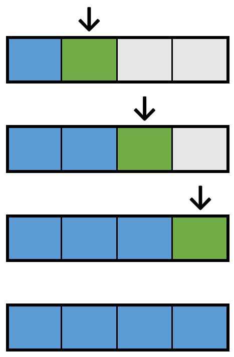
Applications of an insertion sort

The insertion sort is particularly useful for inserting items into an already sorted list. It is usually replaced by more efficient sorting algorithms for large data sets.

Insertion sort in simple-structured English

1. Start at the second item in the list. This becomes the item to be inserted.
2. Compare the item to be inserted with the next adjacent item to the left.
3. If the adjacent item is greater than the item to be inserted move the adjacent item up one place.
4. Repeat from step 2 until the position of the item to be inserted has been found.
5. Insert the current item.
6. Repeat from step 2 with the next item in the list until all the items have been inserted.

Visualising an insertion sort



The steps of an insertion sort are often visualised in examination mark schemes with a table:

Unsorted:	Florida	Georgia	Delaware	Alabama	California
Insert Georgia:	Florida	Georgia	Delaware	Alabama	California
Insert Delaware:	Delaware	Florida	Georgia	Alabama	California
Insert Alabama	Alabama	Delaware	Florida	Georgia	California
Insert California:	Alabama	California	Delaware	Florida	Georgia

Stepping through an example of an insertion sort

Index:	0	1	2	3	4
Step 1	Start at the second item (index 1): Georgia. Compare Georgia to Florida. Florida is less than Georgia, so the correct position has been found. Georgia is inserted.				
	→Florida	Georgia←	Delaware	Alabama	California
Step 2	Move to the next item: Delaware. Compare Delaware with Georgia. Georgia is greater than Delaware, so the position of Delaware has not been found.				
	Florida	→Georgia	Delaware←	Alabama	California
Step 3	Move Georgia up. Compare Delaware with Florida. Florida is greater than Delaware, so the position of Delaware has not been found.				
	→Florida	Georgia	Georgia←	Alabama	California
Step 4	Move Florida up. There are no more items to compare. Delaware is inserted.				
	→Delaware	Florida	Georgia←	Alabama	California
Step 5	Move to the next item: Alabama. Compare Alabama with Georgia. Georgia is greater than Alabama, so the position of Alabama has not been found.				
	Delaware	Florida	→Georgia	Alabama←	California
Step 6	Move Georgia up. Compare Alabama with Florida. Florida is greater than Alabama, so the position of Alabama has not been found.				
	Delaware	→Florida	Georgia	Georgia←	California
Step 7	Move Florida up. Compare Alabama with Delaware. Delaware is greater than Alabama, so the position of Alabama has not been found.				
	→Delaware	Florida	Florida	Georgia←	California

Step 8	Move Delaware up. There are no more items to compare. Alabama is inserted.				
	→Alabama	Delaware	Florida	Georgia←	California
Step 9	Move to the next item: California. Compare California with Georgia. Georgia is greater than California, so the position of California has not been found.				
	Alabama	Delaware	Florida	→Georgia	California←
Step 10	Move Georgia up. Compare California to Florida. Florida is greater than California, so the position of California has not been found.				
	Alabama	Delaware	→Florida	Georgia	Georgia←
Step 11	Move Florida up. Compare California to Delaware. Delaware is greater than California, so the position of California has not been found.				
	Alabama	→Delaware	Florida	Florida	Georgia←
Step 12	Move Delaware up. Compare California to Alabama. Alabama is less than California, so the correct position has been found. California is inserted. There are no more items to check – the list is sorted.				
	→Alabama	California	Delaware	Florida	Georgia←

Notice the algorithm started at the second item. The first item is assumed to already be inserted into the sorted list. This is one implementation of the insertion sort. It is possible to write the algorithm in several different ways. For example, instead of comparing every item with the one before it until the correct position is found, you could start at the beginning of the list (index 0) and work up to the item to be inserted instead.

Did you know?

There are often many different implementations of the same algorithm. You could also write the insertion sort to take items from a list and insert them into a new list. When algorithms move items within the same data structure and do not use additional data structures for their operation, they are called in-place algorithms.

Pseudocode for the insertion sort

```
Function insertionSort(items)
    For index = 1 TO items.Length
        current = items[index]
        index2 = index
        While index2 > 0 AND items[index2 -1] > current
            items[index2] = items[index2 -1]
            index2 = index2 -1
        End while
        items[index2] = current
    End For
    Return items
End function
```

Insertion sort coded in Python using an array/list

```python
def insertion_sort(items):
    n = len(items)
    # Consider all the items
    for index in range(1, n):
        current = items[index]
        index2 = index
        # Find position and move items down one index to make space for new item
        while index2 > 0 and items[index2 - 1] > current:
            items[index2] = items[index2 - 1]
            index2 = index2 - 1
        # Insert new item in correct position
        items[index2] = current
    return items


# Main program starts here
items = ["Florida", "Georgia", "Delaware", "Alabama", "California"]
print(insertion_sort(items))
```

Did you know?

Sorting algorithms are often combined to achieve maximum performance with a variety of different data sets.

The binary merge sort uses an insertion sort on groups of 32 elements and completes the process with a merge sort on the groups. It combines the speed of the insertion sort on small data sets with the speed of the merge sort on large data sets.

Efficiency of an insertion sort

Time complexity			Space complexity
Best case	Average case	Worst case	
$O(n)$ Linear	$O(n^2)$ Polynomial	$O(n^2)$ Polynomial	$O(1)$ Constant

In the best case, the data set is already ordered, so no data needs to be moved. The algorithm has a linear time complexity, $O(n)$, because each item must still be checked – this is not usually the case, as the purpose of the algorithm is to sort data.

The algorithm contains a nested iteration – one loop to check each item and another to move items so an item can be slotted into place. Due to these iterations, the algorithm has polynomial complexity, $O(n^2)$.

The algorithm does not require any additional memory, as it can be performed on the data structure containing the data set. Therefore, it has a space complexity of $O(1)$.

An alternative version of the algorithm puts sorted data into a new list instead of working on the original list – this does not change the time complexity of the algorithm but would increase the space complexity to $O(n)$.

Did you know?

Attempts have been made to further optimise the insertion sort. The Shellsort, invented in 1959 by Douglas Shell, was the most notable, achieving significantly better performance. Another optimisation uses a linked list instead of an array, negating the need to move items within the data structure.

In 2006, a new variant of the insertion sort was proposed, called the library sort or gapped insertion sort. This version leaves a small number of unused spaces in the array to avoid having to move large numbers of items when one is inserted.

Merge sort

A merge sort can sort a data set extremely quickly using divide and conquer. The principle of divide and conquer is to create two or more identical sub-problems from the larger problem, solve them individually and combine their solutions to solve the bigger problem. With the merge sort, the data set is repeatedly split in half until each item is in its own list. Adjacent lists are then merged back together, with each item being entered into the correct place in the new, combined list.

Applications of a merge sort

The merge sort works best with larger data sets where memory usage is not a concern. It is ideal for parallel processing environments where the concept of divide and conquer can be used to maximise efficiency.

Merge sort in simple-structured English

The merge sort has two stages, often called the split and merge steps.

Split step:

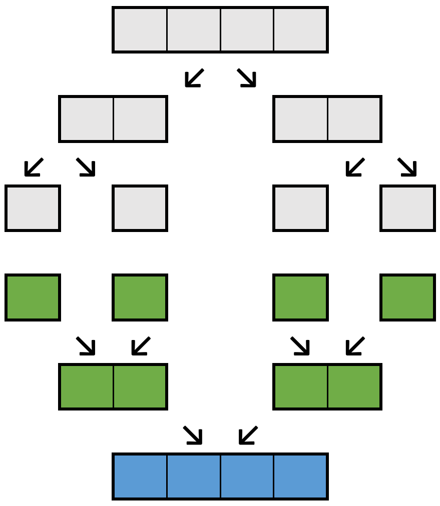
1. Repeatedly divide each list in half until each item is in its own list.

Merge step:

2. Take two adjacent lists and start with the first item in each one.
3. Compare the two items.
4. Insert the lowest item into a new list. Move to the next item in the list it was taken from.
5. Repeat steps 3 and 4 until all the items from one of the lists have been put into the new list.
6. Append all the items from the list still containing items to the new list.
7. Replace the two adjacent lists with the one new list.
8. Repeat from step 2 until only one list remains.

When programming this algorithm using iteration, step 1 can be achieved by putting each item into a new list one at a time – a simple optimisation that is worth implementing. However, in examinations, you should show that the data set is repeatedly split until each item is in its own list.

Visualising a merge sort



Unlike the bubble and insertion sort, the classic merge sort is not an in-place algorithm. It requires additional memory beyond that which is used to store the data to perform its operation. Although this is a significant disadvantage, it does enable multiple processors to work on different parts of the data set at the same time, significantly increasing the algorithm's performance and enabling it to perform better with larger data sets.

Did you know?

The merge sort is very expensive in terms of space complexity because it repeatedly creates new lists. Most optimisations have explored reducing the amount of memory the algorithm requires.

Although the merge sort usually includes the splitting and merging of new, separate lists, it is also possible to write the algorithm to execute within one structure. A smaller amount of additional memory is used as a temporary space outside of the merge routine.

Jyrki Katajainen proved it is possible to write a variation of the classic merge sort with O(1) space.

Stepping through an example of a merge sort

Index:	0	1	2	3	4
Step 1 Split	Divide the list in half to create two smaller lists.				
	Florida	Georgia	Delaware	Alabama	California
Step 2 Split	Divide the two lists in half to create four smaller lists.				
	Florida	Georgia	Delaware	Alabama	California
Step 3 Split	Divide the four lists to create five lists.				
	Florida	Georgia	Delaware	Alabama	California
Step 4 Merge	Compare the first items in two lists: Florida and Georgia are in the correct order. Copy Florida into a new list. Copy Georgia into the list.				
	→Florida	Georgia←	Delaware	Alabama	California
New list	Florida	Georgia			
Step 5 Merge	Compare the first items in two lists: Alabama and Delaware are in the correct order. Copy Alabama into a new list. Copy Delaware into the list.				
	Florida	Georgia	→Alabama	Delaware←	California
New lists	Florida	Georgia	Alabama	Delaware	
Step 6 Merge	No list to compare California with.				

Step 7 Merge	Compare the first items in two lists: Florida and Alabama. Copy Alabama into a new list. Move to the next item in that list.				
	→Florida	Georgia	Alabama←	Delaware	California
New lists	Alabama				
Step 8 Merge	Compare Florida to Delaware. Copy Delaware into the new list. There are no more items in that list.				
	→Florida	Georgia	Alabama	Delaware←	California
New list	Alabama	Delaware			
Step 9 Merge	Copy all the items from the other list into the new list.				
New list	Alabama	Delaware	Florida	Georgia	
Step 10 Merge	Compare the first item in two lists: Alabama and California. Copy Alabama into a new list. Move to the next item in that list.				
	→Alabama	Delaware	Florida	Georgia	California←
New list	Alabama				
Step 11 Merge	Compare the first item in two lists: Delaware and California. Copy California into a new list. There are no more items in that list.				
	Alabama	→Delaware	Florida	Georgia	California←
New list	Alabama	California			
Step 12 Merge	Copy all the items from the other list into the new list.				
New list	Alabama	California	Delaware	Florida	Georgia

When only one list remains, the items are sorted.

Pseudocode for the merge sort

```
Function mergeSort(items)
        listofitems = []
        item = []

        For n = 0 TO items.Length - 1
                item = items[n]
                listofitems.append(item)
        Next n
        While listofitems.Length != 1
                index = 0
                While index < listofitems.Length - 1
                        newlist = merge(listofitems[index],listofitems[index+1])
                        listofitems[index] = newlist
                        del listofitems[index+1]
                        index = index + 1
                End while
        End while
        return listofitems
End function

Function merge(list1,list2)
        newlist = []
        index1 = 0
        index2 = 0
        While index1 < list.Length and index2 < list2.Length
                If list1[index1] > list2[index2] Then
                        newlist.append list2[index2]
                        index2 = index2 + 1
                ElseIf list1[index1] < list2[index2] Then
                        newlist.append list1[index1]
                        index1 = index1 + 1
                ElseIf list1[index1] == list2[index2] Then
                        newlist.append list1[index1]
                        newlist.append list2[index2]
                        index1 = index1 + 1
                        index2 = index2 + 1
                End If
        End While
        If index1 < list1.Length Then
                For item = index1 to list1.Length
                        newlist.append list1[item]
                Next item
        ElseIf index2 < list2.Length Then
                For item = index2 to list2.Length
                        newlist.append list2[item]
                Next item
        End If
        Return newlist
End Function
```

Merge sort coded in Python using arrays/lists and iteration

```
def merge_sort(items):
    # Split step
    items_list = split(items)

    # Merge step
    while len(items_list) != 1:
        index = 0
        # Merge pairs of lists
        while index < len(items_list) - 1:
            new_list = merge(items_list[index], items_list[index + 1])
            items_list[index] = new_list
            # Once merged, delete one of the now redundant lists
            del items_list[index + 1]
            index = index + 1
    return items_list[0]


def split(items):
    # Every item is put into its own list within a container list
    list_of_items = []
    for n in range(len(items)):
        item = [items[n]]
        list_of_items.append(item)
    return list_of_items


def merge(list1, list2):
    # Merge two lists into a new list
    new_list = []
    index1 = 0
    index2 = 0
    # Check each item in each list, and add the smallest item to a new list
    while index1 < len(list1) and index2 < len(list2):
        if list1[index1] > list2[index2]:
            new_list.append(list2[index2])
            index2 = index2 + 1
        elif list1[index1] < list2[index2]:
            new_list.append(list1[index1])
            index1 = index1 + 1
        elif list1[index1] == list2[index2]:
            new_list.append(list1[index1])
            new_list.append(list2[index2])
            index1 = index1 + 1
            index2 = index2 + 1

    # Add left over items from the remaining list
    if index1 < len(list1):
        for item in range(index1, len(list1)):
            new_list.append(list1[item])
    elif index2 < len(list2):
        for item in range(index2, len(list2)):
            new_list.append(list2[item])

    return new_list


# Main algorithm starts here
items = ["Florida", "Georgia", "Delaware", "Alabama", "California"]
print(merge_sort(items))
```

An alternative implementation using recursion

While it is more memory-efficient to implement the merge sort using iteration alone, the algorithm can be optimised for parallel processing using recursion.

A merge sort coded in Python using arrays/lists and recursion

```python
def mergeSort(items):
    # Split step
    if len(items) > 1:
        mid = len(items) // 2
        list1 = items[:mid]
        list2 = items[mid:]

        mergeSort(list1)
        mergeSort(list2)

        index1 = 0
        index2 = 0
        items_index = 0

        # Merge step
        while index1 < len(list1) and index2 < len(list2):
            if list1[index1] < list2[index2]:
                items[items_index] = list1[index1]
                index1 = index1 + 1
            else:
                items[items_index] = list2[index2]
                index2 = index2 + 1
            items_index = items_index + 1

        while index1 < len(list1):
            items[items_index] = list1[index1]
            index1 = index1 + 1
            items_index = items_index + 1

        while index2 < len(list2):
            items[items_index] = list2[index2]
            index2 = index2 + 1
            items_index = items_index + 1
    return items

# Main program starts here
items = ["Florida", "Georgia", "Delaware", "Alabama", "California"]
print(mergeSort(items))
```

Efficiency of a classic merge sort

Time complexity			Space complexity
Best case	Average case	Worst case	
O(n log n) Linearithmic	O(n log n) Linearithmic	O(n log n) Linearithmic	O(n) Linear

In all cases, the data in a merge sort needs to be manipulated so each item is in its own list. The time this takes will increase with more data, as will the memory requirements, O(n). However, by using a divide-and-conquer algorithm, the data set can be repeatedly divided, O(log n).

When the lists are merged back together, it is possible to merge more than one list simultaneously, although each item in the list needs to be considered in turn to determine its position in the new list. Therefore, the algorithm has a linearithmic time complexity, O(n log n), and a space complexity of O(n).

Did you know?

The merge sort was extremely popular in the early days of computing because the algorithm could be run on multiple tape drives. Modern solid-state drives and magnetic hard disks allow for any item of data on the drive to be randomly accessed. However, with tape drives, data could only be accessed serially, one item at a time.

The linear nature of the merge sort makes it ideal for drives such as tape that can only be efficiently read and written forwards.

Quicksort

As its name suggests, the quicksort orders a data set extremely quickly using divide and conquer. The principle of divide and conquer is to create two or more identical, smaller sub-problems from the larger problem, solve them individually and combine their solutions to solve the bigger problem. There are many different implementations of the quicksort, which can be confusing, but they all share two common features identifying the algorithm as a quicksort:

1. A pivot – a single item in the data set that all other items are compared to.
2. Partitioning step – an algorithm that swaps items so that, after each pass, items less than the pivot are placed to its left and items greater than the pivot are placed to its right.

A typical data set after one pass of a quicksort:

9	12	6	20	33	25	47
Values less than the pivot			Pivot	Values greater than the pivot		

The algorithm is called recursively on the set of values to the left of the pivot first and then the values to the right until the complete data set is sorted.

The choice of pivot and partitioning step can make a significant difference to the efficiency of the algorithm. Any item in the data set can be chosen as the pivot, although typically, it is either the first item (Hoare scheme), the last item (Lomuto scheme), the middle item or a random item. Middle and random items are often swapped with the first item before the algorithm begins so the pivot is always the first item in the list.

The quicksort has been continually evolving since its publication by Tony Hoare in 1961, and many different partition schemes are used today. These include finding the median for the pivot, sub-dividing the data set into smaller data sets first, the dual pivot and three pivots. In this book, we will explore three simple approaches, but you will only need to be familiar with one for examinations.

Applications of a quicksort

The quicksort is suitable for any data set but shines with larger data sets. It is ideal for parallel processing environments where the concept of divide and conquer can be used. Typically found in real-time situations due to its efficiency, the quicksort has applications in medical monitoring, life support systems, aircraft controls and defence systems.

Hoare scheme in simple-structured English

A pivot item is chosen, against which other items will be compared – this is usually the first item in the list, but it can be any item including the last item or even a random item. Two pointers are each compared with the pivot in turn until the correct position for the pivot is determined. The algorithm is then repeated recursively on the items to the left and right of this position.

This approach can be described with the following steps:

1. Set the pivot value as the first item in the list.
2. Set a pointer to the second and last items in the list.
3. While the second pointer is greater than or equal to the first pointer:
    a. While the first pointer is less than or equal to the second pointer and the item at the first pointer is less than or equal to the pivot value, increase the first pointer by one.
    b. While the second pointer is greater than or equal to the first pointer and the item at the second pointer is greater than or equal to the pivot, decrease the second pointer by one.
    c. If the second pointer is greater than the first pointer, swap the items.
4. Swap the pivot value with the item at the second pointer.
5. Repeat from step 1 on the list of items to the left of the second pointer.
6. Repeat from step 1 on the list of items to the right of the second pointer.

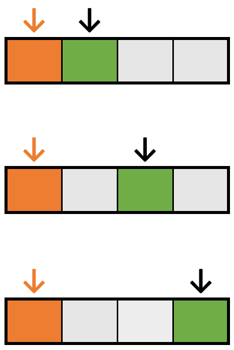
Did you know?

Many sources present the quicksort using only one of the popular algorithms. This can be confusing when you compare this to examination mark schemes that may be using an alternative quicksort scheme. Be aware there is more than one valid approach to a quicksort.

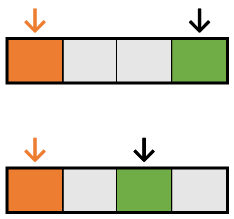
In 2009, Vladimir Yaroslavskiy proposed a new implementation of the quicksort that used two pivots instead of one. At the time, Java was using a variant of the quicksort created by Jon Bentley and Doug McIlroy in the 1990s. After extensive performance tests, Yaroslavskiy proved beyond doubt that his implementation was superior, and it is now the default sorting algorithm for Java.

Visualising one pass of the Hoare scheme

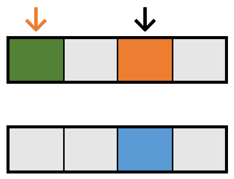
Step 3a: Position the first pointer in relation to the pivot. Move up the data set until an item greater than the pivot is found.



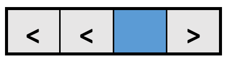
Step 3b: Position the second pointer in relation to the pivot. Move down the data set until an item less than the pivot is found:



Step 3c and 4: Set the position of the pivot by swapping the item at the pivot with the item at the position of the second pointer:



Repeat on all the items to the left of the pivot first and then all the items to the right of the pivot until all items are in the correct position.



Stepping through an example of the Hoare scheme

Index:	0	1	2	3	4
Step 1	Set the pivot to the first item. Set pointers to indexes 1 and 4.				
	↓Florida	→Georgia	Delaware	Alabama	California←
Step 2	Consider the first pointer: Georgia is greater than Florida. First pointer finished.				
	↓Florida	→Georgia	Delaware	Alabama	California←
Step 3	Consider the second pointer: California is less than Florida. Second pointer finished.				
	↓Florida	→Georgia	Delaware	Alabama	California←
Step 4	The second pointer is greater than the first pointer, so swap the items.				
	↓Florida	→California	Delaware	Alabama	Georgia←
Step 5	The second pointer is greater than the first pointer, so consider the first pointer again. Consider the first pointer: California is less than Florida. Increment the first pointer.				
	↓Florida	→California	Delaware	Alabama	Georgia←
Step 6	Consider the first pointer: Delaware is less than Florida. Increment the first pointer.				
	↓Florida	California	→Delaware	Alabama	Georgia←
Step 7	Consider the first pointer: Alabama is less than Florida. Increment the first pointer.				
	↓Florida	California	Delaware	→Alabama	Georgia←
Step 8	Consider the first pointer: Georgia is greater than Florida. First pointer finished.				
	↓Florida	California	Delaware	Alabama	→Georgia←
Step 9	Consider the second pointer: Georgia is greater than Florida. Decrement the second pointer.				
	↓Florida	California	Delaware	Alabama	→Georgia←

Step 10	Consider the second pointer: Alabama is less than Florida. Second pointer finished.				
	↓Florida	California	Delaware	Alabama←	→Georgia
Step 11	The second pointer is not greater than the first pointer. Swap the pivot item with the item at the second pointer.				
	↓Alabama	California	Delaware	Florida←	→Georgia
Step 12	Quicksort the items to the left of the second pointer. Set the pivot to the first item. Set pointers to indexes 1 and 2.				
	↓Alabama	→California	Delaware←	Florida	Georgia
Step 13	Consider the first pointer: California is greater than Alabama. First pointer finished.				
	↓Alabama	→California	Delaware←	Florida	Georgia
Step 14	Consider the second pointer: Delaware is greater than Alabama. Decrement the second pointer.				
	↓Alabama	→California	Delaware←	Florida	Georgia
Step 15	Consider the second pointer: California is greater than Alabama. Decrement the second pointer.				
	↓Alabama	→California←	Delaware	Florida	Georgia
Step 16	The second pointer is less than the first pointer. Swap the pivot item with the item at the second pointer. (No change)				
	↓Alabama←	→California	Delaware	Florida	Georgia
Step 17	Quicksort the items to the left of the second pointer. There are none. Quicksort the items to the right of the second pointer. Set the pivot to the first item. Set pointers on indexes 2 and 2.				
	Alabama	↓California	→Delaware←	Florida	Georgia
Step 18	Consider the first pointer: Delaware is greater than California. First pointer finished.				
	Alabama	↓California	→Delaware←	Florida	Georgia

Step 19	Consider the second pointer: Delaware is greater than California. Decrement the second pointer.				
	Alabama	↓California	→Delaware←	Florida	Georgia
Step 20	The second pointer is less than the first pointer. Swap the pivot item with the item at the second pointer. (No change)				
	Alabama	↓California←	→Delaware	Florida	Georgia
Step 21	Quicksort the items to the left of the second pointer – there are none. Quicksort the items to the right of the second pointer. Delaware is the only item in the list.				
	Alabama	California	Delaware	Florida	Georgia
Step 22	Continue the algorithm from step 11 before the recursive calls. Quicksort the items to the right of the second pointer. Georgia is the only item in the list.				
	Alabama	California	Delaware	Florida	Georgia

Pseudocode for the Hoare scheme

```
Function quicksort(list)
      If items.Length <= 1 Then Return items
      pointer1 = 1
      pointer2 = items.Length – 1
      pivot = items[0]
      While pointer2 >= pointer1
            While pointer1 <= pointer2 And items[pointer1] <= pivot
                  pointer1 = pointer1 + 1
            End While
            While pointer2 >= pointer1 And items[pointer2] >= pivot
                  pointer2 = pointer2 – 1
            End While
            If pointer2 > pointer1 Then Swap(items[pointer1],items[pointer2])
      End While
      Swap(pivot,items[pointer2])
      left = quicksort(items[0 to pointer2])
      right = quicksort(items[pointer2 + 1 to items.Length])
      Return left + items[pointer2] + right
End Function
```

Hoare scheme coded in Python using an array/list

```python
def quicksort(items):
    # A single item does not need sorting
    if len(items) <= 1:
        return items
    else:
        # Set the two pointer positions and pivot to be the first item
        pointer1 = 1
        pointer2 = len(items) - 1
        pivot_value = items[0]
        # Partitioning step
        while pointer2 >= pointer1:
            # Move first pointer
            while pointer1 <= pointer2 and items[pointer1] <= pivot_value:
                pointer1 = pointer1 + 1
            # Move second pointer
            while pointer2 >= pointer1 and items[pointer2] >= pivot_value:
                pointer2 = pointer2 - 1
            # Swap items
            if pointer2 > pointer1:
                temp = items[pointer1]
                items[pointer1] = items[pointer2]
                items[pointer2] = temp
        # Put pivot item in position
        temp = items[0]
        items[0] = items[pointer2]
        items[pointer2] = temp

        # Divide and conquer left and right of the pivot
        left = quicksort(items[0:pointer2])
        right = quicksort(items[pointer2 + 1:len(items)])
        return left + [items[pointer2]] + right


# Main program starts here
items = ["Florida", "Georgia", "Delaware", "Alabama", "California"]
print(quicksort(items))
```

There are many variations of this algorithm, but what identifies it as a Hoare scheme are the two condition-controlled iterations in the partitioning step. One pointer increments up the data set, while a second pointer decrements down the data set.

Did you know?

If you choose three random elements from the data set, select the median of the three and swap it with the rightmost element, you will have improved the chance of an optimal pivot to 68 percent. Take the median of five random elements, and the chance increases to 79 percent. Seven random items, and it's 90 percent! That's because the best-case scenario for a quicksort is to have half the elements on either side of the pivot.

Lomuto scheme in simple-structured English

The Lomuto scheme is a popular alternative quicksort algorithm. It does not perform as well as the Hoare scheme, as it swaps more items than necessary, which is computationally slow. However, many consider it easier to understand because it only uses a single counter-controlled iteration in the partitioning step.

This approach can be described with the following steps:

1. Set the pivot value as the last item in the list.
2. Set a pointer to the first item in the list.
3. For every item in the list:
    a. If the item is less than the pivot:
        i. swap the item with the item at the pointer.
        ii. Increment the pointer.
4. Swap the pivot with the item at the pointer.
5. Repeat from step 1 on the list of items to the left of the pointer.
6. Repeat from step 1 on the list of items to the right of the pointer.

Did you know?

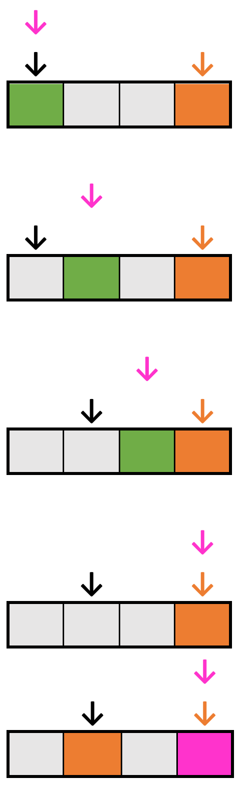
The quicksort was invented by Tony Hoare while he was a student in the Soviet Union.

He was working on a machine translation project and needed to sort Russian words before looking them up in an English-Russian dictionary. Having initially developed the insertion sort, he realised it was too slow and developed the quicksort instead.

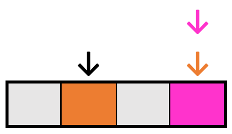
When back in England, Tony Hoare was asked to write code for a Shellsort by his boss and placed a bet that his sorting algorithm was the fastest. His boss coded the quicksort – and Tony won a sixpence!

Visualising one pass of the Lomuto scheme

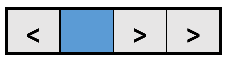
Step 3: The pivot is always the last item. The pink arrow represents the counter-controlled FOR loop. The black arrow is the pointer, which increments only after the item at the pink arrow is less than the pivot.



Step 4: At the end of the iteration, swap the pivot with the item at the pointer.



Repeat on all the items to the left of the pivot first and then all the items to the right of the pivot until all the items are in the correct position.



Did you know?

The quicksort can also be visualised as a binary tree. Each node is a pivot for the child nodes below it.

Stepping through an example of the Lomuto scheme

Index:	0	1	2	3	4
Step 1	Set the pivot to the last item. Set both the pointer and counter to index 0.				
	→Florida←	Georgia	Delaware	Alabama	↓California
Step 2	Consider the item at the counter. Florida is greater than California – no swap is required. Increment the counter.				
	→Florida←	Georgia	Delaware	Alabama	↓California
Step 3	Consider the item at the counter. Georgia is greater than California – no swap is required. Increment the counter.				
	→Florida	Georgia←	Delaware	Alabama	↓California
Step 4	Consider the item at the counter. Delaware is greater than California – no swap is required. Increment the counter.				
	→Florida	Georgia	Delaware←	Alabama	↓California
Step 5	Consider the item at the counter. Alabama is less than California – swap the items at the pointer and counter. Increment the pointer. Increment the counter.				
	→Florida	Georgia	Delaware	Alabama←	↓California
Step 6	The end of the list is reached.				
	Alabama	→Georgia	Delaware	Florida	↓California←
Step 7	Swap the pivot with the item at the pointer.				
	Alabama	→California	Delaware	Florida	↓Georgia←
Step 8	Quicksort the items to the left of the pivot. Alabama is the only item. Quicksort the items to the right of the pivot.				
	Alabama	California	→Delaware←	Florida	↓Georgia

Step 9	Consider the item at the counter. Delaware is less than Georgia – swap the items at the pointer and counter – they are the same. Increment the pointer. Increment the counter.				
	Alabama	California	→Delaware←	Florida	↓Georgia
Step 10	Consider the item at the counter. Florida is less than Georgia – swap the items at the pointer and counter – they are the same. Increment the pointer. Increment the counter.				
	Alabama	California	Delaware	→Florida←	↓Georgia
Step 11	The end of the list is reached.				
	Alabama	California	Delaware	Florida	→↓Georgia←
Step 12	Swap the pivot with the item at the pointer – they are the same.				
	Alabama	California	Delaware	Florida	→↓Georgia←
Step 13	Quicksort the items to the left of the pivot. Quicksort the items to the right of the pivot – no items.				
	Alabama	California	→Delaware←	↓Florida	Georgia
Step 14	Consider the item at the counter. Delaware is less than Florida – swap the items at the pointer and counter – they are the same. Increment the pointer. Increment the counter.				
	Alabama	California	→Delaware←	↓Florida	Georgia
Step 15	The end of the list is reached.				
	Alabama	California	Delaware	→↓Florida←	Georgia
Step 16	Swap the pivot with the item at the pointer – they are the same.				
	Alabama	California	Delaware	→↓Florida←	Georgia
Step 17	Quicksort the items to the left of the pivot. Delaware is the only item. Quicksort the items to the right of the pivot – no items.				
	Alabama	California	Delaware	Florida	Georgia

Pseudocode for the Lomuto scheme

```
Function quicksort(list)
    If items.Length <= 1 Then Return items
    pointer2 = 0
    pivot = items[items.Length -1]
    For pointer1 = 0 TO items.Length - 1
        If items[pointer1] < pivot then
            Swap(items[pointer1], items[pointer2])
            pointer2 = pointer2 + 1
        End If
    End For
    Swap(pivot, items[pointer2])
    left = quicksort(items[0 to pointer2])
    right = quicksort(items[pointer2 + 1 to items.Length])
    Return left + items[pointer2] + right
End Function
```

Lomuto scheme coded in Python using an array/list

```python
def quicksort(items):
    # A single item does not need sorting
    if len(items) <= 1:
        return items
    else:
        # Set the pointer position and pivot to be the last item
        pointer2 = 0
        pivot_value = items[len(items) - 1]
        # Partitioning step
        for pointer1 in range(0, len(items) - 1):
            if items[pointer1] < pivot_value:
                temp = items[pointer2]
                items[pointer2] = items[pointer1]
                items[pointer1] = temp
                pointer2 = pointer2 + 1
        # Put pivot item in position
        temp = items[pointer2]
        items[pointer2] = pivot_value
        items[len(items) - 1] = temp

        # Divide and conquer left and right of the pivot
        left = quicksort(items[0:pointer2])
        right = quicksort(items[pointer2 + 1:len(items)])
        return left + [items[pointer2]] + right


# Main program starts here
items = ["Florida", "Georgia", "Delaware", "Alabama", "California"]
print(quicksort(items))
```

Lomuto variant in simple-structured English

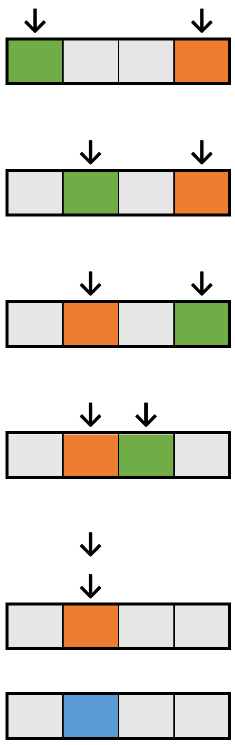
An adaptation of the Lomuto scheme was demonstrated as a Hungarian folk dance by a dance group at Romania's Sapientia University and has appeared in examination mark schemes.

Although the coded solution appears more complex, it is perhaps even easier to understand because it only uses one pointer and a pivot. Two items are compared and, if necessary, swapped. The pointer starts at the first item and always moves towards the pivot. If the pointer is less than the pivot, it moves to the right. If the pointer is greater than the pivot, it moves to the left. Once the pointer has reached the pivot, the position of the pivot has been found. This approach can be described with the following steps:

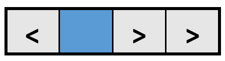
1. Set a pointer to the first item and set the pivot to be the last item in the list.
2. While the pointer is not equal to the pivot:
    a. If the items are in the wrong order, swap the item with the pivot.
    b. Move the pointer one index towards the pivot. *(Note: This could be a move left or right.)*
3. Repeat from step 1 on the list of items to the left of the pointer.
4. Repeat from step 1 on the list of items to the right of the pointer.

Visualising one pass of the Lomuto variant

Step 2: Compare the items at opposite ends and swap them if necessary:



Repeat on all the items to the left of the pivot first and then all the items to the right of the pivot until all the items are in the correct position.



Stepping through an example of the Lomuto variant

Index:	0	1	2	3	4
Step 1	Set the pointer to index 0 and set the pivot to be the last item.				
	→Florida	Georgia	Delaware	Alabama	↓California
Step 2	Compare Florida and California. Florida is greater than California – swap them over.				
	→Florida	Georgia	Delaware	Alabama	↓California
Step 3	Move the pointer towards the pivot.				
	↓California	Georgia	Delaware	→Alabama	Florida
Step 4	Compare Alabama with California. Alabama is less than California – swap them over.				
	↓California	Georgia	Delaware	→Alabama	Florida
Step 5	Move the pointer towards the pivot.				
	Alabama	→Georgia	Delaware	California↓	Florida
Step 6	Compare Georgia with California. Georgia is greater than California – swap them over.				
	Alabama	→Georgia	Delaware	California↓	Florida
Step 7	Move the pointer towards the pivot.				
	Alabama	California↓	→Delaware	Georgia	Florida
Step 8	Compare Delaware with California. Delaware is greater than California – no swap required.				
	Alabama	California↓	→Delaware	Georgia	Florida
Step 9	Move the pointer towards the pivot.				
	Alabama	→California↓	Delaware	Georgia	Florida

Step 10	The pointer is at the pivot. Quicksort items to the left of the pivot. Alabama is the only item. Quicksort items to the right of the pivot.				
	Alabama	California	Delaware	Georgia	Florida
Step 11	Set the pointer to index 2 and set the pivot to be the last item.				
	Alabama	California	→Delaware	Georgia	Florida↓
Step 12	Compare Delaware with Florida. Delaware is less than Florida – no swap required.				
	Alabama	California	→Delaware	Georgia	Florida↓
Step 13	Move the pointer towards the pivot.				
	Alabama	California	Delaware	→Georgia	Florida↓
Step 14	Compare Georgia with Florida. Georgia is greater than Florida – swap them over.				
	Alabama	California	Delaware	→Georgia	Florida↓
Step 15	Move the pointer towards the pivot.				
	Alabama	California	Delaware	→Florida↓	Georgia
Step 16	The pointer is at the pivot. Quicksort items to the left of the pivot. Delaware is the only item. Quicksort items to the right of the pivot. Georgia is the only item.				
	Alabama	California	Delaware	Florida	Georgia

Pseudocode for the Lomuto variant

```
Function quicksort(items)
       If items.Length <= 1 Then Return items
       pointer = 0
       pivot = items.Length – 1
       While pointer != pivot
              If (items[pointer] > items[pivot] and pointer < pivot)
              Or (items[pointer] < items[pivot] and pointer > pivot) Then
                     Swap(items[pointer], items[pivot])
                     Swap(pointer, pivot)
              End If
              If pointer < pivot Then
                     pointer = pointer + 1
              Else
                     pointer = pointer – 1
              End If
       End While
       left = quicksort(items[0 to pointer])
       right = quicksort(items[pointer + 1 to items.Length])
       Return left + items[pointer] + right
End Function
```

Lomuto variant coded in Python using an array/list

```python
def quicksort(items):
    # A single item does not need sorting
    if len(items) <= 1:
        return items
    else:
        # Set the pointer position and pivot to be the last item
        pointer = 0
        pivot = len(items) - 1
        # Partitioning step
        while pointer != pivot:
            # Put pivot item in position
            if (items[pointer] > items[pivot] and pointer < pivot) or (
                    items[pointer] < items[pivot] and pointer > pivot):
                temp = items[pointer]
                items[pointer] = items[pivot]
                items[pivot] = temp
                temp_pointer = pointer
                pointer = pivot
                pivot = temp_pointer
            # Move the pointer
            if pointer < pivot:
                pointer = pointer + 1
            else:
                pointer = pointer - 1

        # Divide and conquer left and right of the pivot
        left = quicksort(items[0:pointer])
        right = quicksort(items[pointer + 1:len(items)])
        return left + [items[pointer]] + right
```

```
# Main program starts here
items = ["Florida", "Georgia", "Delaware", "Alabama", "California"]
print(quicksort(items))
```

Efficiency of a quicksort

Time complexity			Space complexity
Best case	Average case	Worst case	
$O(n \log n)$ Linearithmic	$O(n \log n)$ Linearithmic	$O(n^2)$ Polynomial	$O(\log n)$ Logarithmic

Once the position of the pivot is found, the quicksort divides the list in half and recursively sorts each of the sub-lists, making it ideal for parallel processing.

Dividing the data set in this way provides a linearithmic time complexity, $O(n \log n)$. It is unlikely that the data set will be divided equally in half, but in the worst case, the pivot will always be the first or last item and the time complexity becomes polynomial, $O(n^2)$. At best, the data set will be divided equally, resulting in linearithmic complexity, $O(n \log n)$. On average, the position of the pivot item will be somewhere between the first and last item in the data set, and the time complexity will average out.

The code examples presented here are not memory-efficient because they create new lists for each recursive call. Implementing the algorithm in this way increases the space complexity but makes the algorithm easier to code and understand. An alternative approach would be to use the same data set, passing the pointers as parameters to the quicksort function, not the data structure.

However, any recursive algorithm – even if it does not create new data sets – will require a call stack, so the space complexity is still considered $O(\log n)$, not $O(1)$.

Using the first item as a pivot is often a bad choice for data that is not in a random order because the item at the beginning of a data set is often a small one. In this situation, the number of swaps is increased, so the algorithm is more likely to be $O(n^2)$. A random pivot is more likely to achieve $O(n \log n)$ performance.

One method of further optimising the quicksort is to recognise when the number of items in the data set is small and switch to an algorithm that is better suited to small data sets such as an insertion sort.

The quicksort is often considered more efficient than a merge sort due to requiring less memory than a typical recursive merge sort implementation.

One of the goals of a good sorting algorithm is to not change the order of items that are the same. Moving items unnecessarily is called instability and is a weakness of the quicksort.

Did you know?

Nico Lomuto's partition scheme is generally considered to be less efficient than Tony Hoare's algorithm because it may perform unnecessary swaps – three times as many, on average.

Did you know?

In 1987, Tony Hoare met Nico Lomuto at the Continental Club in Austin, Texas.

"Listen, Tony," Nico said as the chit chat petered off, "about that partitioning algorithm. I never meant to publish or—"

"Oh yes, yes, the partitioning algorithm", interrupted Tony.

"Yeah, that partitioning algorithm that keeps on getting mentioned together with yours. I'm not much of an algorithms theorist… the bothersome part about it is that it's not even a better algorithm. My partitioning scheme will always do the same number of comparisons and at least as many swaps as yours. In the worst case, mine does n additional swaps — n! I can't understand why they keep on mentioning the blessed thing. It's out of my hands now. I can't tell them what algorithms to teach and publish."

"I understand, Nico. Yet please consider the following. Your algorithm is simple and regular, moves in only one direction, and does at most one swap per step. That may be appropriate for some future machines that…"

"No matter the machine, more swaps can't be better than fewer swaps. It's common sense," Nico replied.

"I would not be so sure. Computers do not have common sense. Computers are surprising. It stands to reason they'll continue to be."

dlang.org/blog/2020/05/14/lomutos-comeback/

Why are there so many sorting algorithms?

It is easy to jump to the conclusion that there must be a “best” sorting algorithm. While it is true that sorting algorithms have developed over time and research is still ongoing today, there is no perfect sorting algorithm. Each algorithm’s performance depends on the data set it is being applied to.

When writing algorithms, there are four significant factors to consider:

1. The processing architecture. Parallel processors usually execute more efficiently than single processors. If a data set can be divided and the same algorithm applied to each sub-set of data, it is well-suited to parallelism. Good examples of this are the merge sort and quicksort, which take advantage of divide and conquer and work best with large, unsorted data sets.

2. The memory footprint. The amount of memory an algorithm has to work with can be a limiting factor. In computer science, there is often a balance between the efficiency of processing and the efficiency of memory. In the past, memory was a significant limiting factor and programmers would strive to save every byte they could. In the past, computers had only kilobytes of memory to store the operating system, as well as programs and data – today, we have gigabytes. Algorithms that perform well in a defined memory space include the bubble sort and insertion sort.

3. The volume of code. More efficient algorithms often require more complex code – this is difficult to write for novice programmers and requires more memory to store. Sometimes, the speed of execution is not the most important factor. A good example of this is the bubble sort, which is easy to write, requires very few lines of code and works on all storage media.

4. The state and size of the data set. Some algorithms that are often less efficient can outperform others if the data set is already partially sorted. A good example is the bubble sort, which can outperform a quicksort on a partially sorted list but is quickly beaten if the data set is large and random.

A great exercise is to take the code for each algorithm in this book and apply it to different types of data sets to see how the speed of execution is affected. Different data sets might include ordered, reverse-ordered, random, small and large volumes of data.

Did you know?

In 2001, recognising that no one algorithm is best, Tim Peters invented the Timsort, which is used by the Python sort() method. It is now the default algorithm in Java and across the Android platform. The Timsort is different from other algorithms in that it pre-processes the data to compute which algorithm would be most efficient before applying it. If the array or list has less than 64 elements, it uses an insertion sort. If the data set has more than 64 elements, it uses a variation of a merge sort.

In summary

Bubble sort	Insertion sort	Merge sort	Quicksort
Adjacent items are compared and swapped if they are out of order.	Each item is compared to every other item from the start until its place is found.	Items in one list are compared to items in another list to create a new list.	Items are compared to a pivot.
Uses a nested iteration.	Uses a nested iteration.	Uses either a nested iteration or recursion.	Uses either a nested iteration or recursion.
Slow.	Slow.	Quick.	Quick.
Suitable for a small number of items.	Suitable for a small number of items.	Suitable for a large number of items.	Suitable for a large number of items.
Easy to program.	Easy to program.	More difficult to program.	More difficult to program.
Known memory footprint. In-place algorithm.	Known memory footprint. In-place algorithm.	Memory footprint can increase as the algorithm executes. Classic implementation is not an in-place algorithm.	Memory footprint can increase as the algorithm executes. Can be an in-place algorithm depending on the implementation.

OPTIMISATION ALGORITHMS

Routines that find optimal paths between vertices on a graph.

A* pathfinding

The A* algorithm finds the shortest path between two vertices on a weighted graph using a heuristic. It performs better than Dijkstra's algorithm because not every vertex is considered. Instead, an optimal path is followed towards the goal. A* is an example of a best-first search algorithm.

To avoid the need to consider every vertex, a heuristic estimates the cost of the path between adjacent vertices and the goal. It then follows this path. It is important that the heuristic is admissible by not over-estimating the cost, thereby choosing an incorrect vertex to move to next and ultimately taking a longer path to the goal. The vertices being considered are referred to as the fringe, frontier or border. The usefulness of the A* algorithm is determined by the suitability of the heuristic.

About heuristics

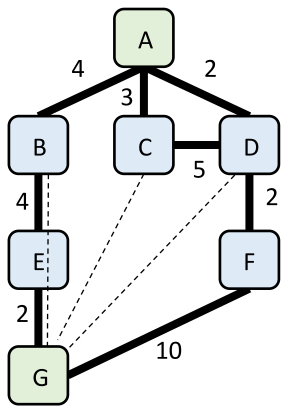


Figure 1

In addition to the cost of each edge, you might be able to calculate the distance between a vertex and the goal, represented by the dotted lines in Figure 1. This additional data, called the heuristic, allows you to determine that B is closer to the goal than C or D. Therefore, B is the vertex that should be followed, even though the cost from A to B is higher than the cost from A to C or A to D. When determining the shortest path, the cost is added to the heuristic to determine the best path. It is worth noting that the calculation of the heuristic does not need to be accurate, it just needs to deliver an admissible result.

Consider a video game where we want to maximise the frames per second by reducing the number of operations to be performed in a specific amount of time. In Figure 2, a character at position S (10,2 on the screen) is required to travel to E (8,16). A heuristic measuring the distance between S and E could be calculated using Pythagoras' theorem on the right-angle triangle: $\sqrt{2^2 + 14^2}$ = 14.142

However, simply adding the two sides (*a* and *o*) requires fewer processing cycles but still delivers a useful result (2 + 14 = 16). We have used the Manhattan distance (the sum of the opposite and adjacent) to estimate *h* instead of the accurate Euclidian distance (hypotenuse). Estimating results in this way – where only a best guess matters – is called a heuristic in computer science.

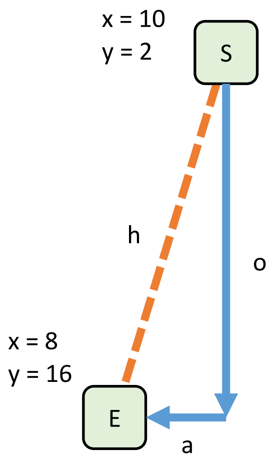


Figure 2

Although the heuristic could potentially be calculated in advance for each vertex, it is usually only calculated as needed to maximise the efficiency of the algorithm.

The A* algorithm has some notation that programmers of the algorithm will be familiar with:

- The distance from the start of a vertex plus the edge value is referred to as the *g* value.
- The heuristic is known as the *h* value.
- The distance from the start of a vertex plus the edge value plus the heuristic is called the *f* value.

Applications of A* pathfinding

Although useful in travel routing systems, A* is generally outperformed by other algorithms that pre-process the graph to enhance performance. A* was originally developed as part of the Shakey project to build a robot with artificial intelligence.

Frequently found in video games, the algorithm is used to move non-playable characters in a way that makes them appear to move intelligently. It can also be found in network packet routing, financial modelling for trading assets and goods (arbitrage opportunity), solving puzzles like word ladders and social networking analyses – calculating degrees of separation between individuals to suggest friends, for example.

A* pathfinding in simple-structured English

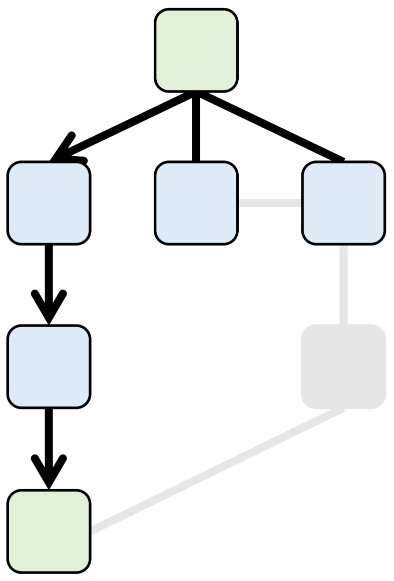
To calculate the shortest path between two nodes:

```
1. Set initial g (and optionally h and f) values for all vertices in graph.
2. Until the goal vertex has been visited:
     a. Find the vertex with the lowest f value that has not been visited.
     b. For each connected vertex that has not been visited:
          i. Calculate the distance from the start by adding the edge value and the heuristic.
         ii. If the distance is lower than the currently recorded f value:
               1. Set the f value of the connected vertex to the newly calculated distance.
               2. Set the previous vertex to be the current vertex.
     c. Set the current vertex as visited.
```

To output the shortest path:

```
3. Start from a goal vertex.
4. Add the previous vertex to the start of a list.
5. Repeat from step 4 until the start vertex is reached.
6. Output the list.
```

Visualising A* pathfinding



Blue nodes represent the fringe. Grey nodes are never considered.

Stepping through A* pathfinding

Note that many sources do not show the initial values for the distance from the start or *f* in their illustration of the A* algorithm. However, they need to be infinity for all vertices, except for the start, which has a value of zero. The heuristic for the goal should also be zero.

Representing infinity

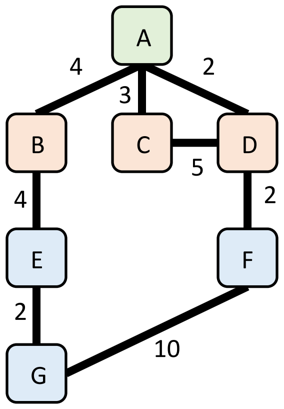
Some programming languages support infinity as a value. In Python, float("inf") uses the IEEE 754 standard which has special floating-point values for infinity. An alternative is to use sys.maxsize from the sys library. In languages that do not support infinity, you would need to set infinity as a very large number that cannot occur. For example, 999999999.

In this example, we have chosen to show the calculated heuristic for each vertex to the goal in advance. Although the heuristic is often be calculated as needed in most implementations, the algorithm can also be optimised with pre-calculations if it makes sense to do so.

Step 1

Set the distance from the start to infinity for all vertices.
Set the distance from the start to zero for vertex A. Set *f* value to the heuristic.

Select the vertex with the lowest *f* value that has not been visited: A.
Consider each edge from A that has not been visited: B, C, D.



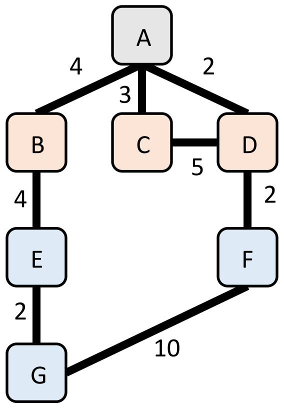
Vertex	Distance from start	Heuristic	*f*	Visited	Previous vertex
A	0	10	10	No	
B	∞	6	∞	No	
C	∞	8	∞	No	
D	∞	12	∞	No	
E	∞	2	∞	No	
F	∞	10	∞	No	
G	∞	0	∞	No	

Step 2

f = A's distance from the start + edge weight + heuristic.

B: 0 + 4 + 6 = 10	(lower than ∞)	Update
C: 0 + 3 + 8 = 11	(lower than ∞)	Update
D: 0 + 2 + 12 = 14	(lower than ∞)	Update

Set A to visited.

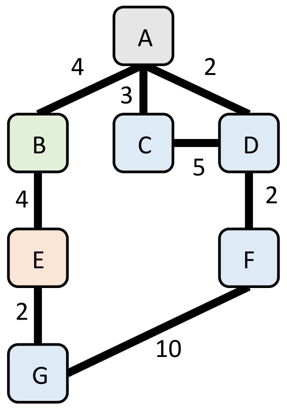


Vertex	Distance from start	Heuristic	f	Visited	Previous vertex
A	0	10	10	Yes	
B	4	6	10	No	A
C	3	8	11	No	A
D	2	12	14	No	A
E	∞	2	∞	No	
F	∞	10	∞	No	
G	∞	0	∞	No	

Step 3

Select the vertex with the lowest f value that has not been visited: B.

Consider each edge from B that has not been visited: E.



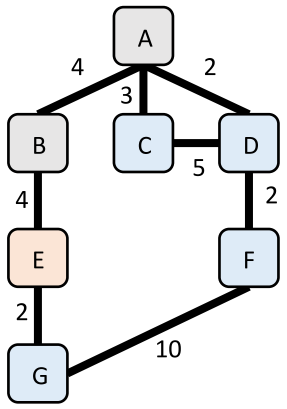
Vertex	Distance from start	Heuristic	f	Visited	Previous vertex
A	0	10	10	Yes	
B	4	6	10	No	A
C	3	8	11	No	A
D	2	12	14	No	A
E	∞	2	∞	No	
F	∞	10	∞	No	
G	∞	0	∞	No	

Step 4

f = B's distance from the start + edge weight + heuristic.

E: 4 + 4 + 2 = 10 (lower than ∞) Update

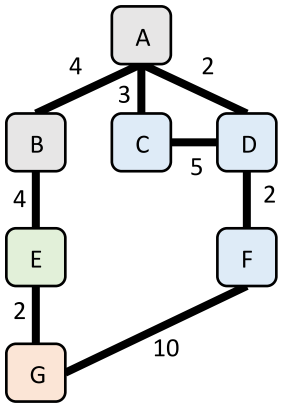
Set B to visited.



Vertex	Distance from start	Heuristic	f	Visited	Previous vertex
A	0	10	10	Yes	
B	4	6	10	Yes	A
C	3	8	11	No	A
D	2	12	14	No	A
E	8	2	10	No	B
F	∞	10	∞	No	
G	∞	0	∞	No	

Step 5

Select the vertex with the lowest *f* value that has not been visited: E.
Consider each edge from E that has not been visited: G.



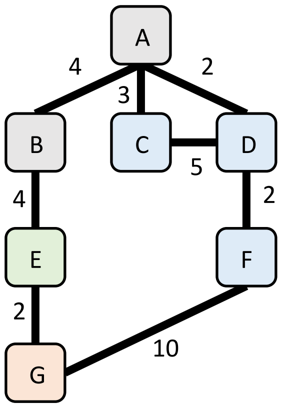
Vertex	Distance from start	Heuristic	f	Visited	Previous vertex
A	0	10	10	Yes	
B	4	6	10	Yes	A
C	3	8	11	No	A
D	2	12	14	No	A
E	8	2	10	No	B
F	∞	10	∞	No	
G	∞	0	∞	No	

Step 6

f = E's distance from the start + edge weight + heuristic.

G: 8 + 2 + 0 = 10 (lower than ∞) Update

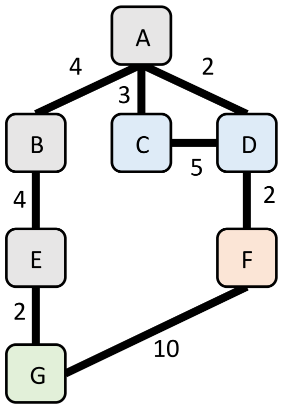
Set E to visited.



Vertex	Distance from start	Heuristic	f	Visited	Previous vertex
A	0	10	10	Yes	
B	4	6	10	Yes	A
C	3	8	11	No	A
D	2	12	14	No	A
E	8	2	10	Yes	B
F	∞	10	∞	No	
G	10	0	10	No	E

Step 7

Select the vertex with the lowest *f* value that has not been visited: G.
Consider each edge from G that has not been visited: F.



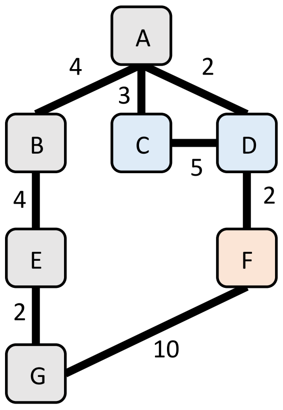
Vertex	Distance from start	Heuristic	f	Visited	Previous vertex
A	0	10	10	Yes	
B	4	6	10	Yes	A
C	3	8	11	No	A
D	2	12	14	No	A
E	8	2	10	Yes	B
F	∞	10	∞	No	
G	10	0	10	No	E

Step 8

f = G's distance from the start + edge weight + heuristic.

F: 10 + 10 + 10 = 30 (lower than ∞) Update

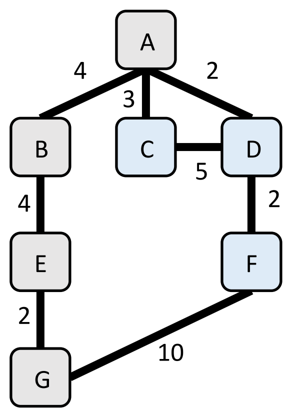
Set G to visited.



Vertex	Distance from start	Heuristic	f	Visited	Previous vertex
A	0	10	10	Yes	
B	4	6	10	Yes	A
C	3	8	11	No	A
D	2	12	14	No	A
E	8	2	10	Yes	B
F	15	10	30	No	G
G	10	0	10	Yes	E

Step 9

At this point, although vertices C, D and F have not been visited, the goal vertex G has been visited and all vertices connected to it have been considered – this means we have found the optimal path.



Vertex	Distance from start	Heuristic	f	Visited	Previous vertex
A	0	10	10	Yes	
B	4	6	10	Yes	A
C	3	8	11	No	A
D	2	12	14	No	A
E	8	2	10	Yes	B
F	15	10	30	No	G
G	10	0	10	Yes	E

Step 10

To output the optimal path, start with the goal vertex and follow the previous vertices back to the start, inserting the new vertex at the front of the list:

Optimal path from A to G is: A→B→E→G

Did you know?

A search algorithm is said to be admissible if it is guaranteed to return an optimal solution. If the heuristic function used by A* is admissible, then A* is admissible.

Pseudocode for A* pathfinding

```
Function astar(graph, start, goal)
    For each vertex in graph
        distance[vertex] = infinity : f[vertex] = infinity
    Next
    distance[start] = 0 : f[start] = 0
    While goal in graph
        shortest = null
        For each vertex in graph
            If shortest == null Then
                shortest = vertex
            ElseIf distance[vertex] < distance[shortest] Then shortest = vertex
            End If
        Next
        For each neighbour in graph[shortest]
            If neighbour in graph and cost + distance[shortest] <
    distance[neighbour] then
                distance[neighbour] = cost + distance[shortest]
                f[neighbour] = cost + g[shortest] + h[neighbour]
                previous_vertex[neighbour] = shortest
            End If
        Next
        graph.pop(shortest)
    End While
    vertex = goal
    While vertex != start
        optimal_path.insert(vertex)
        vertex = previous_vertex[vertex]
    End While
    Return optimal_path
End Function
```

Did you know?

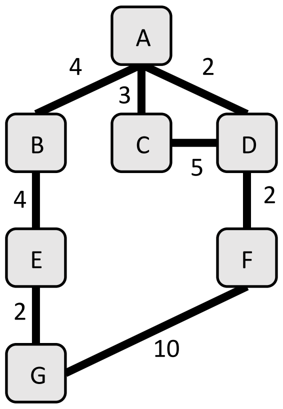
The * symbol in computer science is referred to as an asterisk. However, when speaking, we refer to the A* algorithm as the A-star algorithm.

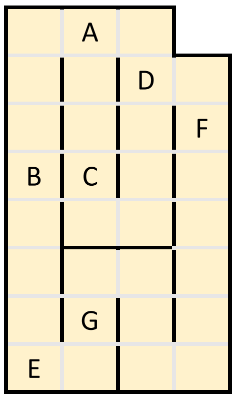
A* pathfinding coded in Python using a dictionary and graph

```python
def astar(graph, h, start, goal):
    # Initialise
    infinity = float("inf")
    g = {}
    f = {}
    previous_vertex = {}
    optimal_path = []

    # Set the g and f for all vertices to infinity
    for vertex in graph:
        g[vertex] = infinity
        f[vertex] = infinity
    # Set the g from the start vertex to 0 and the f value to the heuristic of the start
    g[start] = 0
    f[start] = h[start]

    # Consider each vertex until goal is visited
    while goal in graph:
        # Find the vertex with the shortest f from the start
        shortest = None
        for vertex in graph:
            if shortest == None:
                shortest = vertex
            elif f[vertex] < f[shortest]:
                shortest = vertex

        # Calculate g & f value for each node
        for neighbour, cost in graph[shortest].items():
            # Update f value and previous vertex if lower
            if neighbour in graph and cost + g[shortest] + h[neighbour] < f[neighbour]:
                g[neighbour] = cost + g[shortest]
                f[neighbour] = cost + g[shortest] + h[neighbour]
                previous_vertex[neighbour] = shortest

        # The vertex has now been visited, remove it from the vertices to consider
        graph.pop(shortest)

    # Generate the shortest path
    # Start from the goal, adding vertices to the front of the list
    vertex = goal
    while vertex != start:
        optimal_path.insert(0, vertex)
        vertex = previous_vertex[vertex]
    # Add the start vertex
    optimal_path.insert(0, start)

    # Return the shortest shortest_path
    return optimal_path


# Main program starts here
graph = {"A": {"B": 4, "C": 3, "D": 2}, "B": {"A": 4, "E": 4}, "C": {"A": 3, "D": 5}, "D":
{"A": 2, "C": 5, "F": 2}, "E": {"B": 4, "G": 2}, "F": {"D": 2, "G": 10}, "G": {"E": 2, "F":
10}}
h = {"A": 10, "B": 6, "C": 8, "D": 12, "E": 2, "F": 10, "G": 0}

print(astar(graph, h, "A", "G"))
```

Application of A* to a grid

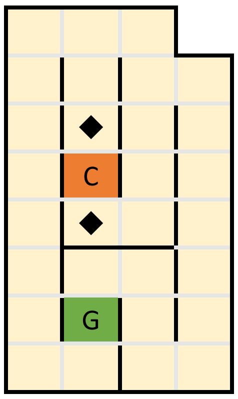
The A* algorithm is often associated with finding the shortest path on a grid – this is especially useful for video games. Consider the graph below. As this is an abstraction, it can be drawn in any number of different ways. The only important data is the vertex, its connections and their edge values.





The grid above contains the same data but visualised in a different way. The edge values are the number of squares to the next letter, location or waypoint.

If the waypoints are not important, they could be removed completely. In that case, each square becomes a vertex. Using the A* algorithm, it is possible to find the shortest path between any two squares, even around walls. A wall prevents squares from having an edge connection between them. The distance from the start (*g*) is the distance to the start square. The heuristic (*h*) is the Manhattan distance to the goal, ignoring walls. Therefore, the *f* value becomes the distance from the start plus the distance to the goal.



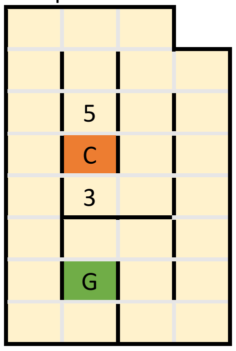
C is the start. G is the goal. The diamonds indicate the vertices connected to C for which the *g*, *h* and *f* values need to be calculated.

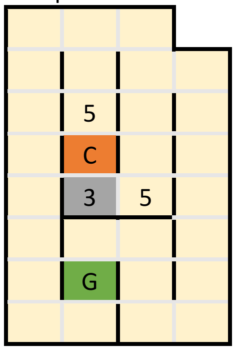
The *g* value of this square is the *g* value of the previous square plus one (0 + 1 = 1).
The *h* value is 4 (four squares from the goal using the Manhattan distance).
The *f* value is 1 + 4 = 5.

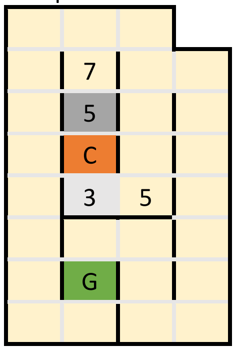
The *g* value of this square is the *g* value of the previous square plus one (0 + 1 = 1).
The *h* value is 2 (two squares from the goal using the Manhattan distance).
The *f* value is 1 + 2 = 3.

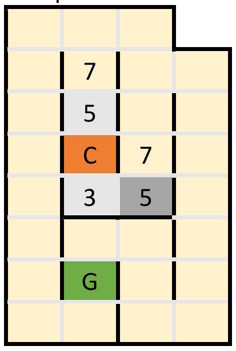
The illustration below shows the shortest path calculated using the A* algorithm from C to G. The *f* values are calculated as needed and shown inside the square to make the example easier to follow. The initial *g* values of the squares are null, which is another way of representing infinity.

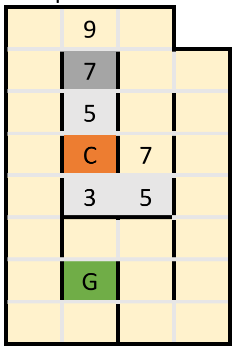
The current vertex is shaded dark grey. The visited vertices are shaded lighter grey. In this example, if more than one square has the lowest *f* value, the one closest to the top-left is considered first.

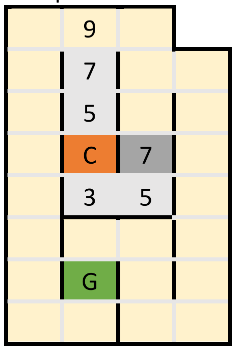
Step 1


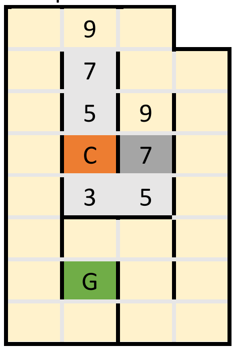
Step 2


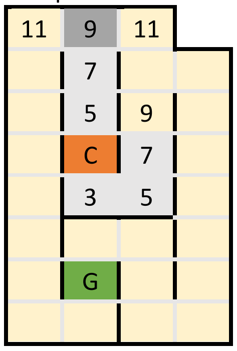
Step 3


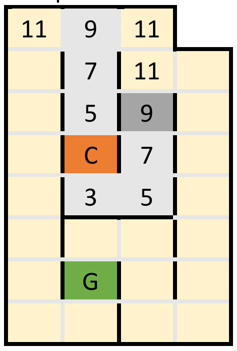
Step 4


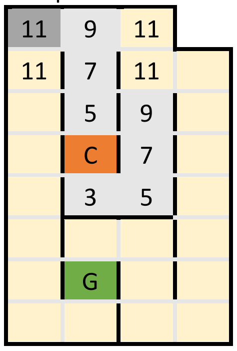
Step 5


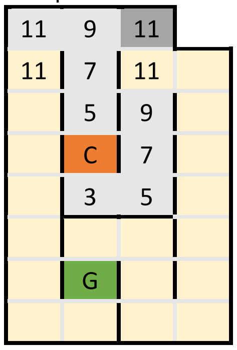
Step 6


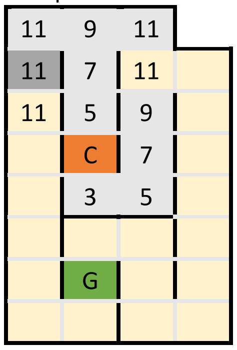
Step 7


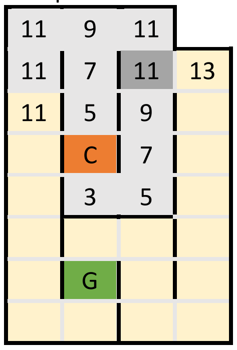
Step 8


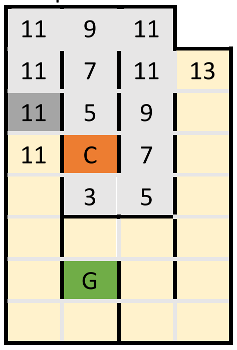
Step 9


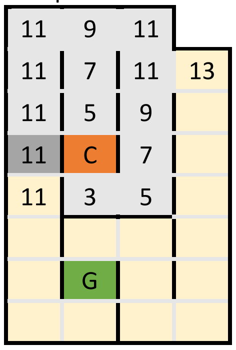
Step 10


Step 11


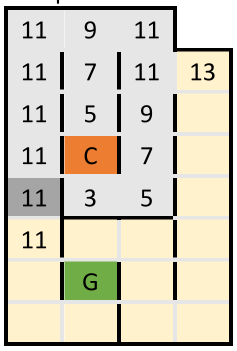
Step 12


Step 13


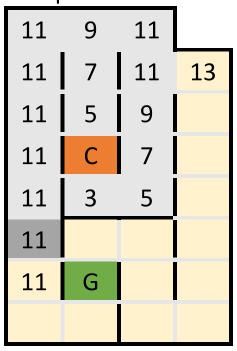
Step 14


Step 15


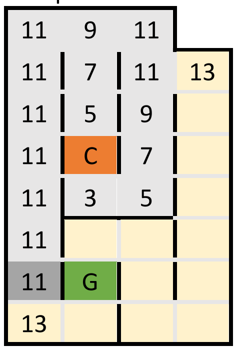
Step 16



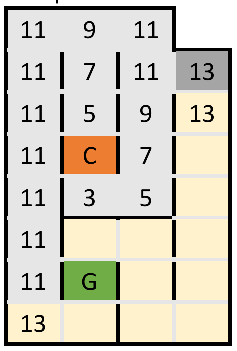
Step 17



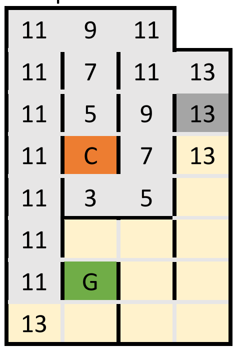
Step 18



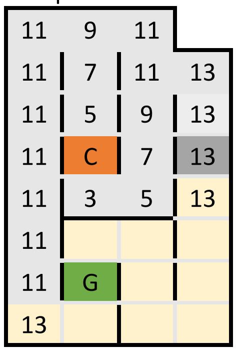
Step 19



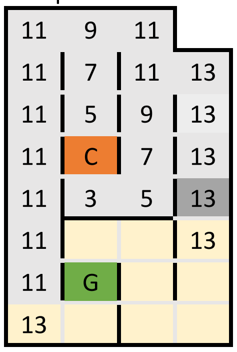
Step 20



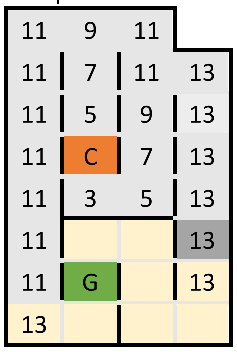
Step 21



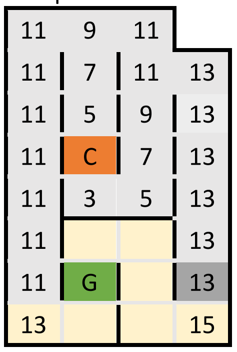
Step 22



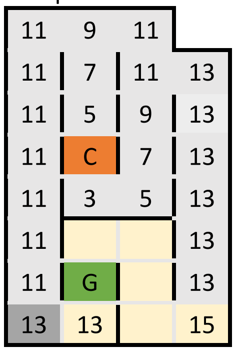
Step 23



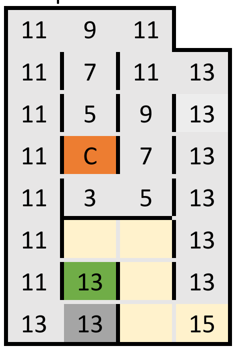
Step 24



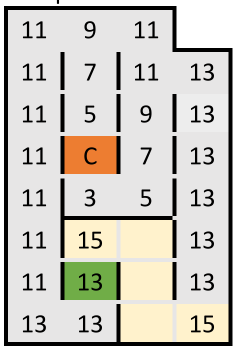
Step 25



Step 26



Step 27



Because of the configuration of the maze, almost all squares were visited. However, there were some squares (shaded yellow) that were not, demonstrating the efficiency savings of the algorithm. With a larger grid and fewer or no walls, the efficiency of the A* algorithm would be more profound.

Efficiency of A* pathfinding

Time complexity		
Best case	Average case	Worst case
$O(b^d)$ Linear	$O(b^d)$ Linear	$O(b^d)$ Polynomial

Determining the efficiency of the A* algorithm is not simple because there are several optimisations that can be implemented and different perspectives on how to calculate the time complexity.

Depending on the purpose of A* pathfinding within a larger program, it may only be necessary to compute one path, not necessarily the most optimal path. Therefore, once the goal vertex has been reached, the algorithm could stop prematurely, possibly reducing the execution time but at the cost of not backtracking to consider other, potentially more optimal routes.

A common optimisation with A* pathfinding is to pre-calculate the *f* values of some vertices and store these in another data structure such as a hash table. If a value is needed again, instead of being calculated to return the same result, it can simply be looked up – assuming the heuristic has not changed.

Having a suitable heuristic that can be calculated quickly is essential – the complexity of this is usually considered to be $O(1)$. Where it makes sense to use a Manhattan distance, the heuristic is as optimal as it can be. However, A* can be used in situations where the heuristic is just a computation to deliver a best guess.

Computer science theorists usually consider the execution time of A* as the result of the number of vertices and edges in the graph. However, those working with artificial intelligence consider what is known as the branching factor (*b*). In AI processing, the number of edges to consider can be extremely large, so an optimisation that avoids having to consider all the vertices would be used. In this case, the number of vertices and edges has less relevance, so time complexity is more a measurement of the depth to the goal vertex (*d*).

The time complexity of A* pathfinding is often calculated as polynomial, $O(b^d)$. The values of the branching factor, the goal vertex and the heuristic affect the efficiency of the algorithm so significantly that it can either be almost linear at best or polynomial at worst.

Dijkstra’s shortest path

Dijkstra’s shortest path algorithm is a special case of the A* pathfinding algorithm. It finds the shortest path between one vertex and all other vertices on a weighted graph. It does not use a heuristic and is a type of breadth-first traversal. A limitation of Dijkstra’s shortest path is that it does not work for edges with a negative weight value. The Bellman–Ford algorithm later provided a solution to that problem.

Applications of Dijkstra’s shortest path

Developed before the A* optimisation for a single path, Edsger Dijkstra developed his algorithm to find the shortest route of travel between Rotterdam and Groningen. It can be used for many purposes where the shortest path between two points needs to be established. Maps, IP routing and the telephone network make use of Dijkstra’s algorithm.

Dijkstra’s shortest path in simple-structured English

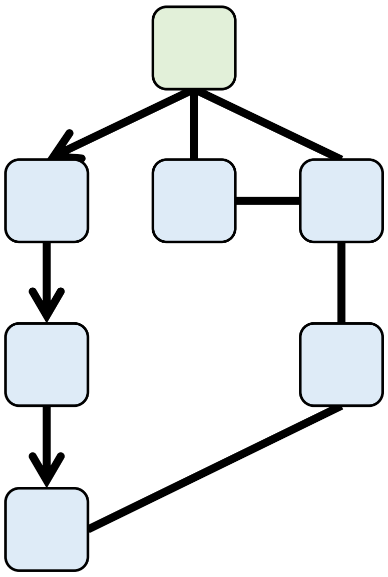
To calculate the shortest path:

```
1.  Set initial distance-from-the-start values for all vertices in graph.
2.  For each vertex in the graph:
        a.  Find the vertex with the shortest distance from the start that has not been visited.
        b.  For each connected vertex that has not been visited:
                i.  Calculate the distance from the start.
                ii. If the distance from the start is lower than the currently recorded distance from the
                    start:
                        1.  Set the shortest distance to the start of the connected vertex to the newly
                            calculated distance.
                        2.  Set the previous vertex to be the current vertex.
3.  Set the vertex as visited.
```

To output the shortest path:

```
4.  Start from a goal vertex
5.  Add the previous vertex to the start of a list.
6.  Repeat from step 5 until the start vertex is reached.
7.  Output the list.
```

Visualising Dijkstra’s shortest path

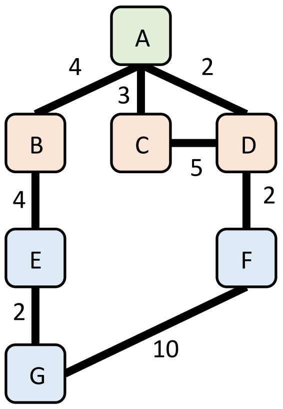


Stepping through Dijkstra’s shortest path

Step 1

Set the distance from the start for all vertices to infinity.
Set the distance from the start for vertex A to zero.

Select the vertex with the lowest distance from the start that has not been visited: A.
Consider each edge from A that has not been visited: B, C, D.



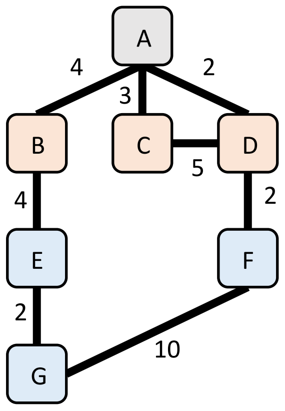
Vertex	Distance from start	Visited	Previous vertex
A	0	No	
B	∞	No	
C	∞	No	
D	∞	No	
E	∞	No	
F	∞	No	
G	∞	No	

Step 2

Distance from the start = A's distance from the start + edge weight.

B: 0 + 4 = 4	(lower than ∞)	Update
C: 0 + 3 = 3	(lower than ∞)	Update
D: 0 + 2 = 2	(lower than ∞)	Update

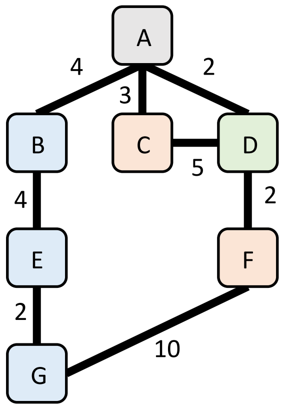
Set A to visited.



Vertex	Distance from start	Visited	Previous vertex
A	0	Yes	
B	4	No	A
C	3	No	A
D	2	No	A
E	∞	No	
F	∞	No	
G	∞	No	

Step 3

Select the vertex with the lowest distance from the start that has not been visited: D.
Consider each edge from D that has not been visited: C, F.



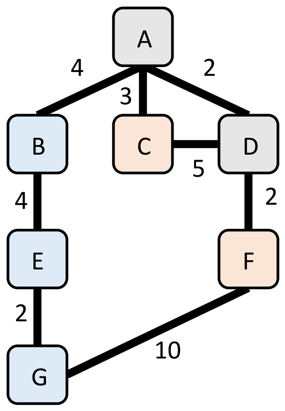
Vertex	Distance from start	Visited	Previous vertex
A	0	Yes	
B	4	No	A
C	3	No	A
D	2	No	A
E	∞	No	
F	∞	No	
G	∞	No	

Step 4

Distance from the start = D’s distance from the start + edge weight.

C: 2 + 5 = 7	(higher than 3)	No update
F: 2 + 2 = 4	(lower than ∞)	Update

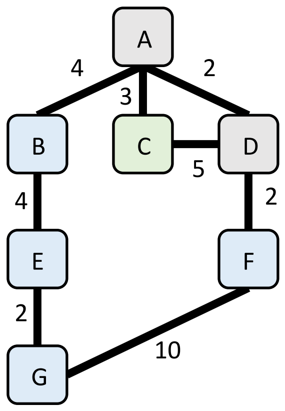
Set D to visited.



Vertex	Distance from start	Visited	Previous vertex
A	0	Yes	
B	4	No	A
C	3	No	A
D	2	Yes	A
E	∞	No	
F	4	No	D
G	∞	No	

Step 5

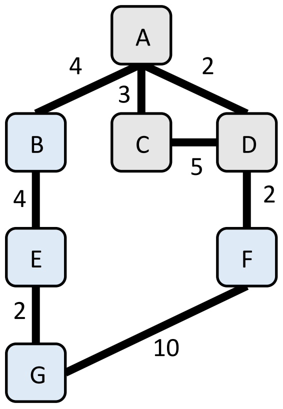
Select the vertex with the lowest distance from the start that has not been visited: C.
Consider each edge from C that has not been visited: none.



Vertex	Distance from start	Visited	Previous vertex
A	0	Yes	
B	4	No	A
C	3	No	A
D	2	Yes	A
E	∞	No	
F	4	No	D
G	∞	No	

Step 6

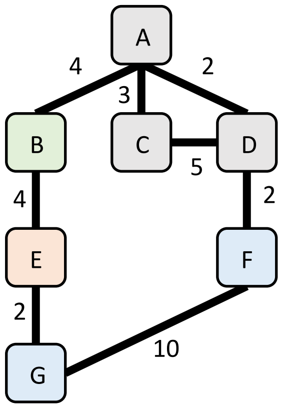
No edges to consider.
Set C to visited.



Vertex	Distance from start	Visited	Previous vertex
A	0	Yes	
B	4	No	A
C	3	Yes	A
D	2	Yes	A
E	∞	No	
F	4	No	D
G	∞	No	

Step 7

Select the vertex with the lowest distance from the start that has not been visited: B.
This could also have been F – it doesn't matter which is chosen first.
Consider each edge from B that has not been visited: E.



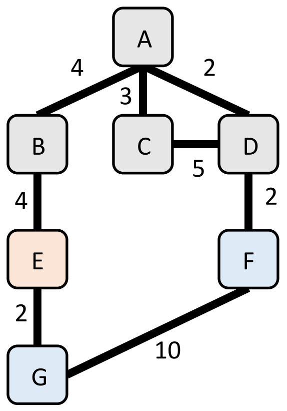
Vertex	Distance from start	Visited	Previous vertex
A	0	Yes	
B	4	No	A
C	3	Yes	A
D	2	Yes	A
E	∞	No	
F	4	No	D
G	∞	No	

Step 8

Distance from the start = B's distance from the start + edge weight.

E: 4 + 4 = 8 (lower than ∞) Update

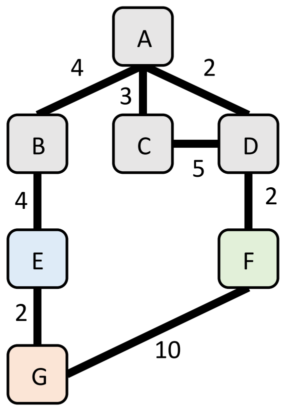
Set B to visited.



Vertex	Distance from start	Visited	Previous vertex
A	0	Yes	
B	4	Yes	A
C	3	Yes	A
D	2	Yes	A
E	8	No	B
F	4	No	D
G	∞	No	

Step 9

Select the vertex with the lowest distance from the start that has not been visited: F.
Consider each edge from F that has not been visited: G.



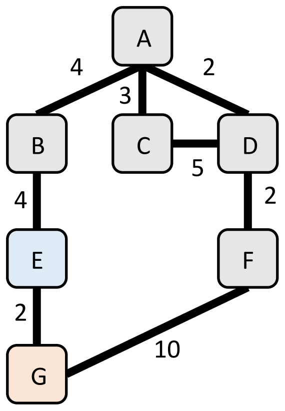
Vertex	Distance from start	Visited	Previous vertex
A	0	Yes	
B	4	Yes	A
C	3	Yes	A
D	2	Yes	A
E	8	No	B
F	4	No	D
G	∞	No	

Step 10

Distance from the start = F’s distance from the start + edge weight.

G: 4 + 10 = 14 (lower than ∞) Update

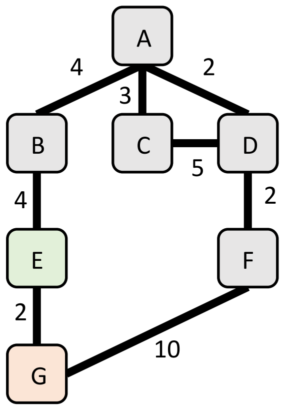
Set F to visited.



Vertex	Distance from start	Visited	Previous vertex
A	0	Yes	
B	4	Yes	A
C	3	Yes	A
D	2	Yes	A
E	8	No	B
F	4	Yes	D
G	14	No	F

Step 11

Select the vertex with the lowest distance from the start that has not been visited: E.
Consider each edge from F that has not been visited: G.



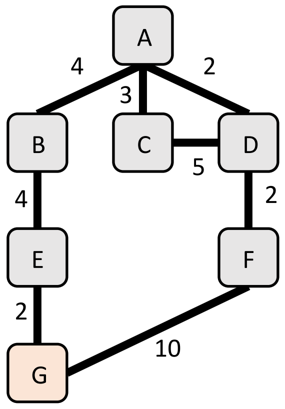
Vertex	Distance from start	Visited	Previous vertex
A	0	Yes	
B	4	Yes	A
C	3	Yes	A
D	2	Yes	A
E	8	No	B
F	4	Yes	D
G	14	No	F

Step 12

Distance from the start = E's distance from the start + edge weight.

G: 8 + 2 = 10 (lower than 14) Update

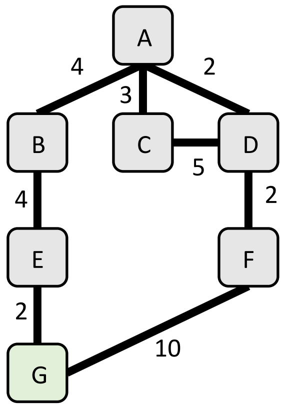
Set E to visited.



Vertex	Distance from start	Visited	Previous vertex
A	0	Yes	
B	4	Yes	A
C	3	Yes	A
D	2	Yes	A
E	8	Yes	B
F	4	Yes	D
G	10	No	E

Step 13

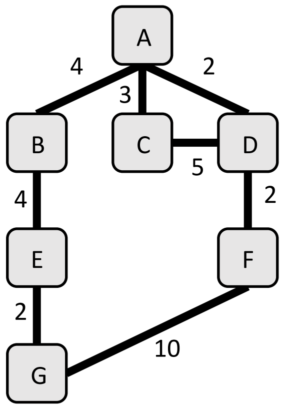
Select the vertex with the lowest distance from the start that has not been visited: G.
Consider each edge from G that has not been visited: none.



Vertex	Distance from start	Visited	Previous vertex
A	0	Yes	
B	4	Yes	A
C	3	Yes	A
D	2	Yes	A
E	8	Yes	B
F	4	Yes	D
G	10	No	E

Step 14

No edges to consider.
Set G to visited.



Vertex	Distance from start	Visited	Previous vertex
A	0	Yes	
B	4	Yes	A
C	3	Yes	A
D	2	Yes	A
E	8	Yes	B
F	4	Yes	D
G	10	Yes	E

Step 15

There are no unvisited vertices. The algorithm is complete.

To output the shortest path from the start to the goal, begin with the goal vertex and follow the previous vertices back to the start, inserting the new vertex at the front of the list:

Shortest path from A to G is: A→B→E→G.

Note how Dijkstra's algorithm finds the shortest path between every vertex to the start. You can pick any vertex and follow the previous vertices back to vertex A – a key difference between Dijkstra and the A* algorithm.

Did you know?

Dijkstra's algorithm was used to demonstrate the computing ability of the new ARMAC computer in 1956. The computer was able to find the shortest routes between 64 cities in the Netherlands on a transport map. 64 was the limit because it only required six bits (2^6) to encode a city name as a number.

Pseudocode for Dijkstra's shortest path

```
Function dijkstra(graph, start, goal)
    For each vertex in graph
        distance[vertex] = infinity
    Next
    distance[start] = 0
    While unvisited_vertices in graph
        shortest = null
        For each vertex in graph
            If shortest == null Then
                shortest = vertex
            ElseIf distance[vertex] < distance[shortest] Then
                shortest = vertex
            End If
        Next
        For each neighbour in graph
            If cost + distance[shortest] < distance[neighbour] then
                distance[neighbour] = cost + distance[shortest]
                previous_vertext[neighbour] = shortest
            End If
        Next
        vertex = visited
    End While
    vertex = goal
    While vertex != start
        shortest_path.insert(vertex)
        vertex = previous_vertex[vertex]
    End While
    Return shortest_path
End Function
```

Did you know?

Adaptations of Dijkstra's shortest path are often the algorithms behind routing protocols in packet switching networks.

Dijkstra’s shortest path coded in Python using a dictionary and graph

```
def dijkstra(graph, start, goal):
    # Initialise
    infinity = float("inf")
    distance = {}
    previous_vertex = {}
    shortest_path = []

    # Set the shortest distance from the start for all vertices to infinity
    for vertex in graph:
        distance[vertex] = infinity
    # Set the shortest distance from the start for the start vertex to 0
    distance[start] = 0

    # Loop until all the vertices have been visited
    while graph:
        # Find the vertex with the shortest distance from the start
        shortest = None
        for vertex in graph:
            if shortest == None:
                shortest = vertex
            elif distance[vertex] < distance[shortest]:
                shortest = vertex

        # Calculate shortest distance for each edge
        for neighbour, cost in graph[shortest].items():
            # Update the shortest distance for the vertex if the new value is lower
            if neighbour in graph and cost + distance[shortest] < distance[neighbour]:
                distance[neighbour] = cost + distance[shortest]
                previous_vertex[neighbour] = shortest

        # The vertex has now been visited, remove it from the vertices to consider
        graph.pop(shortest)

    # Generate the shortest path
    # Start from the goal, adding vertices to the front of the list
    vertex = goal
    while vertex != start:
        shortest_path.insert(0, vertex)
        vertex = previous_vertex[vertex]
    # Add the start vertex
    shortest_path.insert(0, start)

    # Return the shortest shortest_path
    return shortest_path


# Main program starts here
graph = {"A": {"B": 4, "C": 3, "D": 2}, "B": {"A": 4, "E": 4}, "C": {"A": 3, "D": 5}, "D":
{"A": 2, "C": 5, "F": 2},
         "E": {"B": 4, "G": 2}, "F": {"D": 2, "G": 10}, "G": {"E": 2, "F": 10}}
print(dijkstra(graph, "A", "G"))
```

Efficiency of Dijkstra’s shortest path

Time complexity		
Best case	Average case	Worst case
O(E+V log V) Linearithmic	O(E log V) Linearithmic	$O(V^2)$ Polynomial

A FOR loop is used to set the shortest distance of all the vertices. This part of the algorithm is linear, O(n). The main algorithm uses a graph stored as a dictionary, and we assume a time complexity of O(1) for looking up data about each vertex.

A FOR loop is nested in a WHILE loop when the shortest distance for each neighbouring vertex is calculated from every connected edge – this means the algorithm has a polynomial complexity at worst, $O(V^2)$, but different implementations can reduce this to O(E log V), where E is the number of edges and V the number of vertices.

Did you know?

Dijkstra’s shortest path does not work with negative edge values. The Bellman-Ford algorithm and a further development, Johnson’s algorithm, would later provide a solution to this problem.

Dijkstra’s shortest path is also an example of a breadth-first traversal.

In summary

A* pathfinding	Dijkstra's shortest path
Finds the shortest path between two vertices only.	Finds the shortest path from one vertex to all other vertices.
Uses a heuristic.	No heuristic.
Only promising vertices are expanded during the search.	All vertices are expanded during the search.
The heuristic helps find a solution more quickly.	Slower than A* pathfinding.
Fails with negative edge values and if the heuristic is over-estimated.	Fails with negative edge values.
A typical heuristic might be calculated as the distance to the goal vertex.	

Index