# Homework 3 Stacks Answers

1. Describe why a stack is not a suitable data structure for holding client records at a call centre. Suggest a more suitable data structure and justify why it is suitable. [3]

* A stack is a limited access data structure. Items can only be added at the top and removed from the top. At a call centre, records would have to be accessed randomly when the client calls.
* A list is a more suitable data structure.
* A list can be searched randomly to find a particular item.

2. If a stack is implemented as a dynamic data structure, what bounds the number of items that can be pushed? [1]

* The amount of memory available to grow the stack is the only limiting factor

3. The operation peek() returns the top item of a stack without removing it from the stack. What should happen if a peek() is attempted on an empty stack? [1]

 An error message should be returned because the top pointer would be out of range, for example, -1

4. Complete the following to show the state of a stack after the indicated operations. The stack can only hold 4 items in total. [5]

 One mark for each 3 lines correct (one mark for last 4 lines)

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | **Stack** | **Front** | **Result** |
| stack 🡨 new array(4) | [] | -1 |  |
| push(rabbit) | [rabbit] | 0 |  |
| push(fox) | [rabbit, fox] | 1 |  |
| push(mouse) | [rabbit, fox, mouse] | 2 |  |
| peek() | [rabbit, fox, mouse] |  | mouse |
| pop() | [rabbit, fox] | 1 | mouse |
| pop() | [rabbit] | 0 | fox |
| push(hedgehog) | [rabbit, hedgehog] | 1 |  |
| push(magpie) | [rabbit, hedgehog, magpie] | 2 |  |
| push(badger) | [rabbit, hedgehog, magpie, badger] | 3 |  |
| isFull() | [rabbit, hedgehog, magpie, badger] |  | True |
| peek() | [rabbit, hedgehog, magpie, badger] |  | badger |
| pop() | [rabbit, hedgehog, magpie] | 2 | badger |
| pop() | [rabbit, hedgehog] | 1 | magpie |
| pop() | [rabbit] | 0 | hedgehog |
| pop() | [] | -1 | rabbit |
| isEmpty() | [] |  | True |

5. (a) Describe the role of the **call** **stack** and **stack frame** in relation to subroutine calls. [5]

 A stack frame is pushed onto the **call stack** when a subroutine call is executed. (1)

 Each stack frame holds the return address and the parameters passed to a subroutine, and the local variables used in the subroutine. (3)

 A stack may hold many stack frames when subroutines are nested, one for each subroutine currently being executed. (1)

 The contents of a stack frame are popped when a subroutine completes, and execution resumes at the line after the call statement. (1)

 (b) Figure 1 shows the skeleton of a program containing several subroutine calls.

 Figure 1 Figure 2

|  |  |
| --- | --- |
| 100 | SUB subA(p1) |
| 101 |  subB (p1, 4) |
| 102 |  … |
| 199 | ENDSUB |
|  |  |
| 200 | SUB subB(p10) |
| 201 |  x 🡨 12 |
| 202 |  subC (p10, x) |
| 203 |  … |
| 299 | ENDSUB |
|  |  |
| 300 | SUB subC (p1, p2) |
| 301 |  … |
| 399 | ENDSUB |
|  |  |
|  |  |
| 500 | main() |
| 501 |  p10 🡨 8 |
| 502 |  subA(p10) |
| 503 |  … |
|  |  |

|  |  |  |
| --- | --- | --- |
| **Line** | **Stack** | **Mark** |
| 500 |  |  |
| 501 |  |  |
| 502 | [503] |  |
| 100 |  |  |
| 101 | [503,102] | (1) |
| 200 |  |  |
| 201 |  |  |
| 202 | [503,102,203] | (1) |
| 300 |  |  |
| 301 |  |  |
| 399 | [503,102] | (1) |
| 203 |  |  |
| 299 | [503] | (1) |
| 102 |  |  |
| 199 | [] | (1) |
| 503 |  |  |

Complete the table in Figure 2 to show the state of the stack during these subroutine calls, using the notation [return address1, return address2, ..]

**Show only return addresses.**

The state of the stack at line 502 is given in the table. [5]

* Each correct line = 1 mark

[Total 20 Marks]