**Coursework 2**

**Data Structures and Algorithms**

**M2I224180**

|  |  |
| --- | --- |
| Surname: | <Surname> |
| First Name: | <First Name> |
| Matriculation No: | <Matric No> |

**Please answer ALL questions in the boxes provided. Questions which ask you to provide your own numbers have space in this Answer Sheet for these numbers. Please ensure that you complete these sections clearly.**

This coursework must be submitted by **1700 hrs (BST) on Friday 17th April 2020** on GCULearn via Turnitin.

This file should be named in the format **surname\_name\_CW2.docx** before submitting.

This is an individual piece of work and your attention is drawn to the university rules on plagiarism and its repercussions:

‘The deliberate and substantial unacknowledged incorporation in your submission of material derived from work (published or unpublished) of another is prohibited. In such cases the work submitted is classified as illegal and will result in further action being taken by the University which may include recording a failure.’

**Question 1**

|  |  |
| --- | --- |
| **(a)** | Let double power (double base, int exponent) be our function.  If exponent is greater than 1, return 1  Else calculate: power(base, exponent-1)\*base |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **(b)** | |  |  |  |  | | --- | --- | --- | --- | | **Function** | **Stack before** | **Stack Afterwards** | **Top** | | Push: inserts values into stack. We’ll insert 5,4,9,8, in that order. | *Empty* | 8  9  4  5 | 8(because  Stacks are LIFO  8 is the top) | | Pop: removes a value from the stack. Working with the values from the previous example, let’s pop 8. | 8  9  4  5 | 9  4  5 | 9(because 8 was  Popped from the  Stack, 9 becomes  the new top) | | Peek: returns a specific value from the stack, without removing it | 9  4  5 | 9  4  5 | 9(because in this  Case, we’re only  Peeking, no value  Is removed. So,  9 remains the top. | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **I** | |  |  |  | | --- | --- | --- | | A | = | 6 | | B | = | 6 | | C | = | 1 | | D | = | 2 | | E | = | 4 |  |  |  |  |  | | --- | --- | --- | --- | | **Value** | **Stack Top** | **Expression(Stack)** | **Stack With Phone Number Values** | | A | A | A | 6 | | B | B | B A | 6  6 | | - | (A-B) | (A-B) | (6-6) | | C | C | C  (A-B) | 1  (6-6) | | D | D | D  C  (A-B) | 2  1  (6-6) | | / | (C/D) | (C/D)  (A-B) | (1/2)  (6-6) | | \* | (A-B)\*(C/D) | (A-B)\*(C/D) | (6-6)\*(1/2) | | E | E | E  (A-B)\*(C/D) | 4  (6-6)\*(1/2) | | + | ((A-B)\*(C/D))+E | ((A-B)\*(C/D))+E | ((6-6)\*(1/2))+4 |   ((6-6)\*(1/2))+4 = 4 |

|  |  |
| --- | --- |
| **(d)** | **d)**  There are various algorithms to use when needing to add or remove items from a stack implemented as an array:   * Push: an algorithm that’s designed to add a value into the stack. Using push, items are placed at the top of the stack. * Pop: an algorithm designed to remove an item from the stack. The pop function searches through the stack until it finds the item specified, then deletes it from the stack entirely. * isEmpty: will check to see if the stack is empty. Is used by both push and pop to see if removal and addition is possible. |

**Question 2**

|  |  |
| --- | --- |
| **(a)** | 1. When a node has no children, in other words it’s a leaf. 2. When a node has both left and right subtree. 3. When a node has a left or right subtree, but not both. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **(b)** | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | N | A | T | H | A | N | D | U | |
| **(i)** | A picture containing clock, pair, table, plate  Description automatically generated  Seeing as there are duplicates in my binary search tree, I opted to follow the format of representing the amount of duplicates of a certain letter with a number next to it. Currently, there are 2 Ns and 2 As in my binary search tree. |
| **(ii)** | Following this representation of my binary search tree, the Post Order sequence of traversal would be: DHAAUTNN. |
| **(iii)** | * First create an array with the values you would like to sort. * Once completed, add these values into a binary search tree. * Once the tree is created, proceed to do an in-order traversal of the tree. * Return the values from the in-order traversal to get your sorted values. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **(c)** | My adjacency matrix is as follows:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | 1 | 2 | 3 | 4 | 5 | | 1 | 0 | 1 | 0 | 0 | 1 | | 2 | 0 | 0 | 0 | 1 | 1 | | 3 | 0 | 0 | 0 | 0 | 0 | | 4 | 0 | 0 | 1 | 0 | 0 | | 5 | 0 | 0 | 1 | 0 | 0 | |

**Question 3**

|  |  |
| --- | --- |
| **(a)** | XX = 14  YY = 18  Our set becomes: 21, 18, 55, 14, 60, 18, 28  **Step 1**: 21,**|**18, 55,14,60,28:   * 21 is greater 18, swap the items: * 21,\_\_,55,14,60,28 (18 kept in a separate “bucket” of memory) * \_\_,21,55,14,60,28 * 18,21,55,14,60,28   **Step 2:** 18,21**|**55,14,60,28:   * 21 is less than 55, no swap   **Step 3:** 18,21**,**55,**|**14,60,28:   * 55 is greater than 14, swap   18,21,55,\_\_,60,28 (14 kept in a separate “bucket” of memory)  18,21,\_\_,55,60,28 18,21,14,55,60,28   * 21 is greater than 14, swap   18,21,\_\_,55,60,28 (14 kept in a separate “bucket” of memory)  18,\_\_,21,55,60,28 18,14, 21,55,60,28   * 18 is greater than 14, swap   18,\_\_,21,55,60,28 (14 kept in a separate “bucket” of memory)  \_\_,18,21,55,60,28 14,18, 21,55,60,28  **Step 4:** 14,18, 21,55,**|**60,28   * 55 is less than 60, no swap   **Step 5:** 14,18, 21,55, 60, **|**28   * 60 is greater than 28, swap   14,18, 21,55,60,\_\_ (28 kept in a separate “bucket” of memory)  14,18,21,55,\_\_,60 14,18, 21,55,28,60   * 55 is greater than 28, swap   14,18, 21,55,\_\_,60 (28 kept in a separate “bucket” of memory)  14,18,21,\_\_,55,60 14,18, 21,28,55,60  **ARRAY SORTED: 14,18,21,55,60** |

|  |  |
| --- | --- |
| **(b)** | In Hash tables, collisions happen when 2 values in the table are assigned the same key. This triggers a collision in the Hash table. |

|  |  |
| --- | --- |
| **(c)** | When working with linear probing, when one finds a collision, the item is simply moved along the hashtable until an empty spot is found for it to occupy. The item is then inserted in that spot. However, with quadratic probing, the next open spot is found by following a probe sequence. Both resolutions have similar issues with regards to things like deletion, clustering and wrapping around, although quadratic probing may encounter less clustering problems generally. |

|  |  |
| --- | --- |
| **(d)** | A=7  B=5  4  ∑(7i-5) i=2  **Value 1** = 7\*2-5=**9**  **Value 2**= 7\*3-5=**16**  **Value 3** = 7\*4-5 =**23**  4  ∑(7i-5)=9+16+23=**48** i=2 |

|  |  |
| --- | --- |
| **(e)** | X = 8  Y = 7  Set n = 10000  *f(10000)=(8\*10000*)^2/3 = 80 000^2/3 = 1857 (rounded up)  *g(10000)= (7\*10000) logBase2 10000=* 70 000 logBase2 10000 =930140 (rounded up)  g(n)>f(n), therefore use f(n). |