EC-200 Data Structures

Lab Manual 11

**Course Instructor:** Lecturer Anum Abdul Salam

**Lab Engineer:** Lab Engineer Ayesha Batool

**Student Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Degree/ Syndicate: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Trait** | **Obtained Marks** | **Maximum Marks** |
| **R1** | **Application Functionality 20%** |  | 20 |
| **R2** | **Specification & Data structure implementation**  **30%** |  | 30 |
| **R3** | **Reusability**  **10%** |  | 10 |
| **R4** | **Input Validation**  **10%** |  | 10 |
| **R5** | **Efficiency**  **20%** |  | 20 |
| **R6** | **Delivery**  **10%** |  | 10 |
| **R7** | **Plagiarism above 80%** |  | 1 |
|  | **Total** |  | 10 |

**Total Marks = O**𝒃𝒕𝒂𝒊𝒏𝒆𝒅 𝑴𝒂𝒓𝒌𝒔 (∑6𝟏 𝑹𝒊 ∗ 𝑹7)

# Lab Manual # 11: Heaps

## Lab Objective:

To implement heaps

## Description:

A Binary Heap is a Binary Tree with following properties.

1. It’s a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.
2. A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to Min Heap.

### Max Heap

Where the value of the root node is greater than or equal to either of its children.



### Min Heap

Where the value of the root node is less than or equal to either of its children.

****

Both trees are constructed using the same input and order of arrival.

### Max Heap Construction Algorithm

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

**Step 1 −** Create a new node at the end of heap.

**Step 2 −** Assign new value to the node.

**Step 3 −** Compare the value of this child node with its parent.

**Step 4 −** If value of parent is less than child, then swap them.

**Step 5 −** Repeat step 3 & 4 until Heap property holds.

### Max Heap Deletion Algorithm

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

**Step 1** − Remove root node.

**Step 2** − Move the last element of last level to root.

**Step 3** − Compare the value of this child node with its parent.

**Step 4** − If value of parent is less than child, then swap them.

**Step 5** − Repeat step 3 & 4 until Heap property holds.

### How is Binary Heap represented?

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as array.

* The root element will be at Arr[0].
* Below table shows indexes of other nodes for the ith node, i.e., Arr[i]:

|  |  |
| --- | --- |
| Arr[i/2-1] | Returns the parent node |
| Arr[(2\*i)+1] | Returns the left child node |
| Arr[(2\*i)+2] | Returns the right child node |

## Lab task:

Implement Heap data structure using arrays which has following functions

* Isempty()
* int parent\_index(int index)
* int left\_child\_index(int index)
* int right\_child\_index(int index)
* bool has\_right\_child(int index)
* bool has\_left\_child(int index)
* bool has\_parent(int index)
* int peek() // returns top element
* swap( int index1 , int index2)
* heapifyup()
* insert()
* heapifydown()
* remove()

**TEST PLAN:**

1. **Execute your test plan. If you discover mistakes in your implementation, correct them and execute your test plan again.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.** | **Operations** | **Expected Results** | **Results/Status** |
| **1.** | Create an empty heap h | Items [20] created |  |
| **2.** | Call h.remove() | Empty heap |  |
| **3.** | h.add(10); | 10 added in array |  |
| **4.** | h.peek() | Display 10 |  |
| **5.** | h.add(20); | 20 added |  |
| **6.** | h.peek() | Display 20 |  |
| **7.** | h.add(30); | 30 added |  |
| **8.** | h.add(25); | 25 added |  |
| **9.** | h.add(28); | 28 added |  |
| **10.** | h.peek() | Display 30 |  |
| **11.** | h.remove(); | Remove 30 |  |
| **12.** | h.peek() | Display 28 |  |
| **13.** | h.remove(); | Remove 28 |  |
| **14.** | h.remove(); | Remove 25 |  |
| **15.** | h.peek() | Display 20 |  |
| **16.** | h.remove(); | Remove 20 |  |
| **17.** | h.remove(); | Remove 10 |  |
| **18.** | h.remove(); | Empty Heap |  |

* Calculate time complexity for inserting and removing an element from heap.
* Write any 3 applications of heaps.