

**CC-213L**

**Data Structures and Algorithms**

**Laboratory 11**

**Heap and AVL**

**Version: 1.0.0**

**Release Date: 04-12-2023**

**Department of Information Technology**

**University of the Punjab**

**Lahore, Pakistan**

**Contents:**

- Learning Objectives
- Required Resources
- General Instructions
- Background and Overview
  - Pointers and Dynamic Memory Allocation
  - Self-Referential Objects
    - Representation
    - Implementation
  - Non-Linear Data Structure
    - Tree
    - Graphs
    - Heaps
    - Hash Tables
  - Binary Trees
    - Binary Trees
    - Binary Search Tree
    - AVL
  - Rotations in AVL
    - Left-Left Rotation
    - Right-Right Rotation
    - Left-Right Rotation
    - Right-Left Rotation
  - Heap Data Structure
    - Heap
    - Min Heap
    - Max Heap
    - Operations on Heap
    - Common Uses of Heap
- Activities
  - Pre-Lab Activity
    - Task 01: AVL Implementation
  - In-Lab Activity
    - Task 01: Array Based Heap Implementation
    - Task 02: Insert into Heap
    - Task 03: Remove from Heap
    - Task 04
    - Task 05: Driver Program
  - Post-Lab Activity
    - Task 01: Add Member functions to Heap ADT
    - Task 02: Heap Sort
- Submissions
- References and Additional Material
- Lab Time and Activity Simulation Log

**Learning Objectives:**

- Pointers and Dynamic Memory Allocation
- Self-Referential Objects
- Non-Linear Data Structure
- Heap Data Structure
- AVL

**Resources Required:**

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

**General Instructions:**

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the followings.

Teachers:		
Course Instructor	Prof. Dr. Syed Waqar ul Qounain	<a href="mailto:swjaffry@pucit.edu.pk">swjaffry@pucit.edu.pk</a>
Lab Instructor	Madiha Khalid	<a href="mailto:madiha.khalid@pucit.edu.pk">madiha.khalid@pucit.edu.pk</a>
Teacher Assistants	Muhammad Nabeel	<a href="mailto:bitf20m009@pucit.edu.pk">bitf20m009@pucit.edu.pk</a>
	Abdul Rafay Zubairi	<a href="mailto:bcsf20a032@pucit.edu.pk">bcsf20a032@pucit.edu.pk</a>

## Background and Overview

**Pointers and Dynamic Memory Allocation:** Pointers and dynamic memory allocation are important concepts in programming, particularly in languages like C and C++. Pointers allow you to work with memory addresses, while dynamic memory allocation allows you to manage memory at runtime.

**Pointers:** A pointer is a variable that stores the memory address of another variable. It allows you to indirectly access the value of the variable stored at that address. Pointers are often used for various purposes, such as dynamically allocated memory, working with arrays, and passing functions as arguments.

```
3  int main() {
4      int x = 10;
5      int* ptr; // Declare a pointer to an int
6      ptr = &x; // Assign the memory address of x to ptr
7
8      std::cout << "The value of x: " << x << std::endl;
9      std::cout << "The value pointed to by ptr: " << *ptr << std::endl;
10
11     *ptr = 20; // Modify the value through the pointer
12     std::cout << "The updated value of x: " << x << std::endl;
13
14     return 0;
15 }
```

Figure 1(Pointers)

**Explanation:** In this example, ptr is a pointer to an integer, and it is assigned the memory address of the variable x. You can access and modify the value of x through the pointer using the dereference operator (\*ptr).

```
The value of x: 10
The value pointed to by ptr: 10
The updated value of x: 20
```

Figure 2(output)

**Dynamic Memory Allocation:** Dynamic memory allocation allows you to allocate memory for variables at runtime. In C++, you can use new and delete operators to allocate and deallocate memory for objects on the heap.

```
3  int main() {
4      int* dynamicArray = new int[5]; // Allocate an array of 5 integers
5
6      for (int i = 0; i < 5; i++) {
7          dynamicArray[i] = i * 10;
8      }
9
10     for (int i = 0; i < 5; i++) {
11         std::cout << "dynamicArray[" << i << "] = " << dynamicArray[i] << std::endl;
12     }
13
14     delete[] dynamicArray; // Deallocate the memory
15
16     return 0;
17 }
```

Figure 3(Dynamic Memory)

**Explanation:** In this example, dynamicArray is allocated on the heap with space for 5 integers. After using it, it is essential to deallocate the memory using delete[] to prevent memory leaks. **Note:** In modern C++ (C++11 and later), it is recommended to use smart pointers like std::unique\_ptr and std::shared\_ptr for better memory management, as they automatically handle memory deallocation.

Select Microsoft Visual Studio Debug Console

```
dynamicArray[0] = 0
dynamicArray[1] = 10
dynamicArray[2] = 20
dynamicArray[3] = 30
dynamicArray[4] = 40
```

Figure 4(Output)

**Self-Referential Objects (Single Self Reference):** Classes that have capability to refer to their own types of objects are called **Self Referential Classes/Structs**. Objects of such classes are called self-referential Objects. Self-referential structure in C++ are those structure that contains one or more than one pointer as their member which will be pointing to the structure of the same type. In simple words, a structure that is pointing to the structure of the same type is known as a self-referential structure.

Example in C++

```
3 struct Node
4 {
5     int info;
6     Node* left;
7     Node* right;
8 };
```

Figure 5(Self Referencing)

**Explanation:** In Figure 5 we have declared a struct Node. It has three data members info, left and right pointers to Node.

**Info** Represents the information data part. Enables the object to store relevant information in it. There can be more than one identifier of same/different datatypes depending upon the application /situation. **left and right** Represents the link part. Enables the object to a self-referential object. There can be more than one such references used for different purposes in different applications /situations.

```
3 struct Node
4 {
5     int info;
6     Node* left;
7     Node* right;
8 };
9 int main()
10 {
11     Node a, b, c;
12     a.info = 10;
13     a.left = &b;
14     a.right = &c;
15     a.left->info = 20;
16     a.right->info = 30;
17     a.left->left = nullptr;
18     a.left->right = nullptr;
19     a.right->left = nullptr;
20     a.right->right = nullptr;
21     return 0;
22 }
23 }
```

Figure 6(Self Referential Objects)

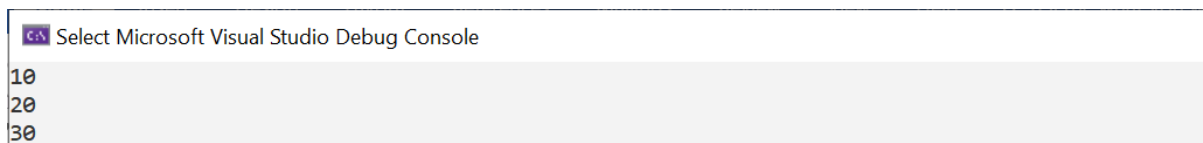
Here we have displayed the values that are stored in those variables that are self-referenced.

```

9  int main()
10 {
11     Node a, b, c;
12     a.info = 10;
13     a.left = &b;
14     a.right = &c;
15     a.left->info = 20;
16     a.right->info = 30;
17     a.left->left = nullptr;
18     a.left->right = nullptr;
19     a.right->left = nullptr;
20     a.right->right = nullptr;
21
22     cout << a.info << endl; // 10
23     cout << a.left->info << endl; // 20;
24     cout << a.right->info << endl; // 30
25     return 0;
26 }
27

```

Figure 7(Self-Referencing)



```

Select Microsoft Visual Studio Debug Console
10
20
30

```

Figure 8(Output)

**Non-Linear Data structures:** Non-linear data structures are data structures in which elements are not arranged in a sequential, linear manner. Unlike linear data structures (e.g., arrays, linked lists) where elements are stored in a linear order, non-linear data structures allow for more complex relationships among elements. Here are some examples of non-linear data structures:

### 1. Trees:

- Binary Tree: Each node has at most two children.
- Binary Search Tree (BST): A binary tree where the left subtree of a node contains only nodes with keys less than the node's key, and the right subtree only nodes with keys greater than the node's key.
- AVL Tree: A self-balancing binary search tree where the height of the two child subtrees of every node differs by at most one.

### 2. Graphs:

- Directed Graph (Digraph): A graph in which edges have a direction.
- Undirected Graph: A graph in which edges do not have a direction.
- Weighted Graph: A graph in which each edge has an associated weight.

### 3. Heaps:

- Binary Heap: A complete binary tree where the value of each node is greater than or equal to (or less than or equal to) the values of its children.
- Max Heap: A binary heap where the value of each node is greater than or equal to the values of its children.
- Min Heap: A binary heap where the value of each node is less than or equal to the values of its children.

### 4. Hash Tables:

- Hash Map: A data structure that implements an associative array abstract data type, a structure that can map keys to values.
- Open Addressing: A technique in hash tables where collisions are resolved by finding the next open slot in the hash table.
- Separate Chaining: A technique in hash tables where each bucket (or slot) in the hash table holds a linked list of elements.

These non-linear data structures are essential in various applications and are chosen based on the specific requirements and characteristics of the data and the operations to be performed on them.

### Binary Trees:

Binary trees are a type of tree data structure in which each node has at most two children, which are referred to as the left child and the right child. These children are distinguished as being either the "left" or "right" child. The topmost node in a binary tree is called the root, and nodes with no children are called leaves. Here are some common types of binary trees:

**1. Binary Tree:** In a general binary tree, each node can have at most two children. However, there are no strict rules about how the children are organized, making it a more general form.

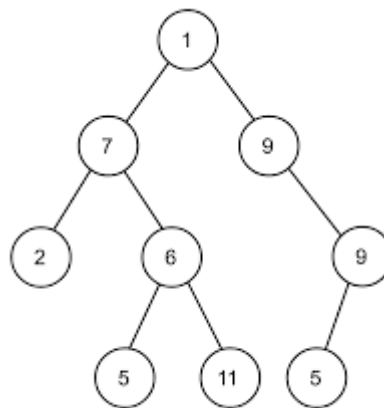


Figure 9(Binary Tree)

**2. Binary Search Tree (BST):** A binary search tree is a binary tree in which each node has a value, and the values of nodes in the left subtree are less than the value of the root, while the values in the right subtree are greater. This property makes searching for a specific value more efficient compared to a general binary tree.

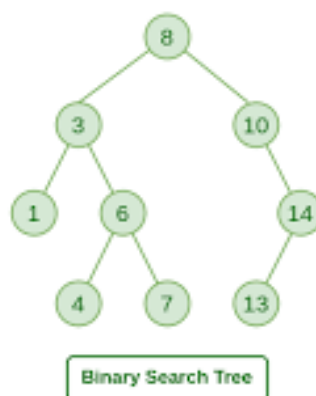


Figure 10(Binary Search Tree)

**3. AVL:** AVL refers to a type of self-balancing binary search tree named after its inventors Adelson-Velsky and Landis. AVL trees maintain balance during insertions and deletions to ensure that the tree remains relatively balanced, which helps to keep the search time for elements logarithmic.

The balance property of an AVL tree is defined by the heights of the two child subtrees of every node, which should not differ by more than one. If at any time during an insertion or deletion operation the balance factor violates this property, rotations are performed to restore balance. There are four possible rotations in AVL trees: left rotation, right rotation, left-right rotation, and right-left rotation. These rotations are applied to adjust the balance factors of nodes and maintain the overall balance of the tree. The self-balancing property of AVL trees ensures that the worst-case time complexity for basic operations such as search, insertion, and deletion remain logarithmic, making them efficient for dynamic sets and dictionaries where these operations are commonly performed.

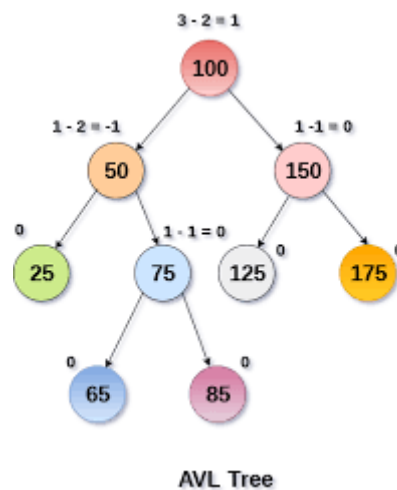
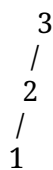


Figure 11(AVL)

## Rotations in AVL

### 1. Left-Left Rotation (LL):

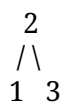
Unbalanced Tree:



**Steps for Left-Left Rotation:**

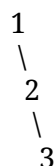
- Perform a right rotation on the unbalanced node (3 in this case).
- Update heights of the rotated nodes.

Result (Balanced Tree):



### 2. Right-Right Rotation (RR):

Unbalanced Tree:





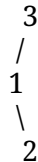
**Steps for Right-Right Rotation:**

- a) Perform a left rotation on the unbalanced node (1 in this case).
- b) Update heights of the rotated nodes.

Result (Balanced Tree):

**3. Left-Right Rotation (LR):**

Unbalanced Tree:

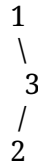
**Steps for Left-Right Rotation:**

- a) Perform a left rotation on the left child (1 in this case).
- b) Perform a right rotation on the unbalanced node (3 in this case).
- c) Update heights of the rotated nodes.

Result (Balanced Tree):

**4. Right-Left Rotation (RL):**

Unbalanced Tree:

**Steps for Right-Left Rotation:**

- a) Perform a right rotation on the right child (3 in this case).
- b) Perform a left rotation on the unbalanced node (1 in this case).
- c) Update heights of the rotated nodes.

Result (Balanced Tree):



- During each rotation, it's essential to update the heights of the nodes involved to maintain accurate height information for balancing.
- The balance factor (difference in heights of left and right subtrees) of each node should be in the range [-1, 1] after the rotations.
- Rotations are recursive; if a rotation is performed on a subtree, it might trigger further rotations in the parent or ancestor nodes.

These rotations are the key mechanisms for maintaining the AVL property, which ensures that the tree remains balanced and maintains a logarithmic height for efficient operations.

## Heap Data Structure

A heap is a specialized tree-based data structure that satisfies the heap property. There are two main types of heaps: min-heap and max-heap.

1. **Min-Heap:** In a min-heap, for every node  $i$  other than the root, the value of  $i$  is greater than or equal to the value of its parent. This means that the minimum element is at the root, and the values of child nodes are greater than or equal to the value of their parent. The heap property ensures that the minimum element is always at the top, making it efficient to extract the minimum element.

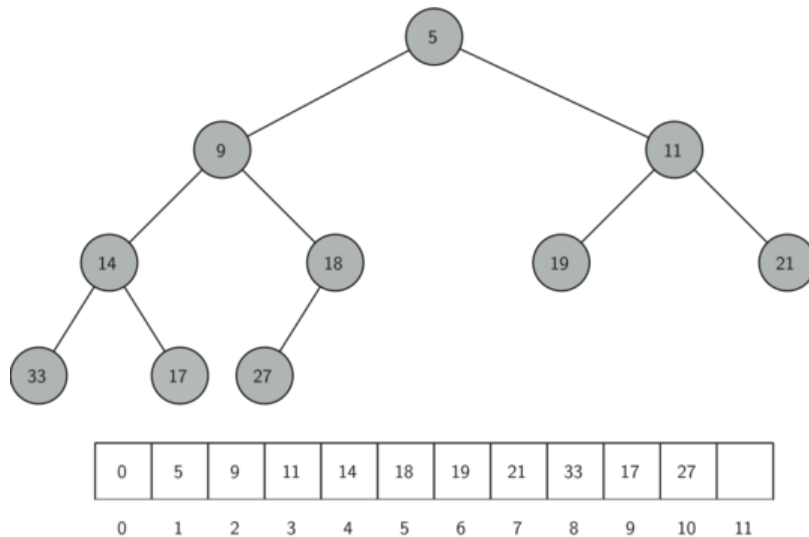
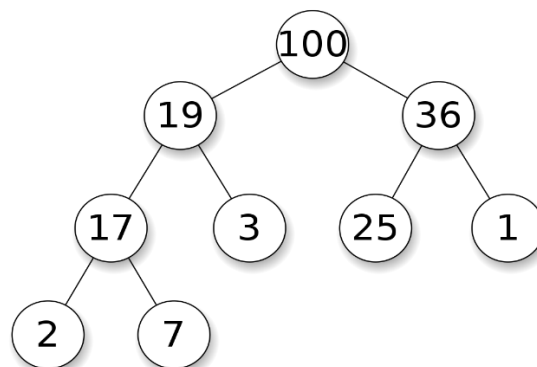


Figure 12(Min Heap)

2. **Max-Heap:** In a max-heap, for every node  $i$  other than the root, the value of  $i$  is less than or equal to the value of its parent. This means that the maximum element is at the root, and the values of child nodes are less than or equal to the value of their parent. Like the min-heap, the max-heap property makes it efficient to extract the maximum element.

### Tree representation



### Array representation

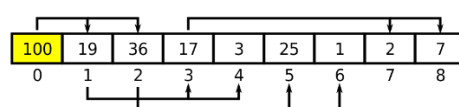


Figure 13(Max Heap)

**Operations on Heap:**

1. **Insert:** Adding a new element to the heap involves placing the new element at the bottom, maintaining the heap property, and then bubbling up or down to restore the heap property.
2. **Extract Minimum (or Maximum):** Removing the root element (minimum in a min-heap or maximum in a max-heap) involves replacing it with the last element, adjusting the heap to maintain the heap property, and then bubbling down or up as needed.
3. **Heapify:** Converting an array of elements into a heap is known as heapify. It can be done in linear time.

**Common Use Cases:**

1. **Priority Queues:** Heaps are often used to implement priority queues, where elements with higher priorities (lower values in min-heap, higher values in max-heap) are dequeued before elements with lower priorities.
2. **Heap Sort:** The heap data structure is used in the Heap Sort algorithm, which is an in-place sorting algorithm with a time complexity of  $O(n \log n)$ .
3. **Graph Algorithms:** Heaps are used in various graph algorithms, such as Dijkstra's algorithm for finding the shortest path and Prim's algorithm for finding a minimum spanning tree.

Heaps are efficient data structures for scenarios where you need quick access to the minimum (or maximum) element in a set of elements.

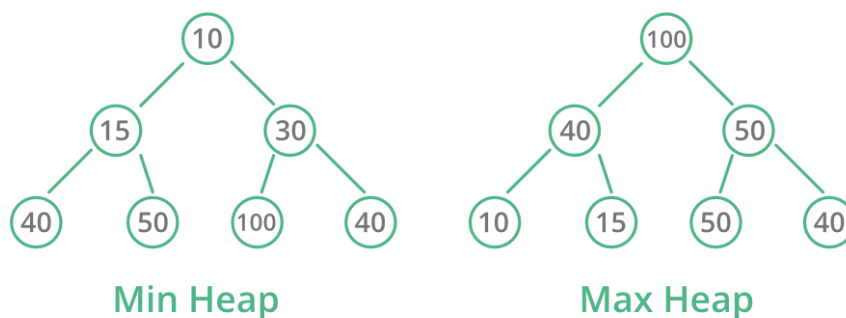


Figure 14(Heap Data Structure)

## Activities

### Pre-Lab Activities:

**Task 01: AVL implementation:** Implement **AVL ADT** inherited from **BST** and override insertion and delete class method.

```
template <class T>
class AVL;
template<class T>
class BST;
template <class T>
class BSTNode {
    friend BST<T>;
    friend AVL<T>;
    T data;
    BSTNode<T>* left;
    BSTNode<T>* right;
    // Methods...
};

template <class T>
class BST {
protected:
    BSTNode<T>* root;
public:
    virtual void insert(T value);
    virtual void deleteNode(T value);
    // methods
};

template <class T>
class AVL: public BST<T> {
public:
    void insert(T value) override;
    void deleteNode(T value) override;
};
```

All the method of the BST will be part of the AVL according to the principle of Inheritance. Just override deletion and insertion method. Will constructor and destructor and copy constructor will also be inherited too ? Think and implement them if they are not inherited.

```
BST<int>* ptr = new AVL<int>(); // think about the virtual destructor
delete ptr;
```

1. Constructor
2. Destructor
3. Copy-Constructor
4. void setRoot(T value);
5. void insert (T value);
6. BSTNode<T>\* getLeftChild(BSTNode<T>\* node);
7. BSTNode<T>\* getRightChild(BSTNode<T>\* node);
8. BSTNode<T>\* search (T value);
9. void deleteNode (T value);
10. void printNodes (T parentKey, T value) ;// use in-order traversal.

## In-Lab Activities:

### Task 01: Array Based Heap Implementation

In this task, you are going to implement a class **StudentMaxHeap**. Each node of this Max Heap will contain the Roll number, and CGPA of a student. **The heap will be organized on the basis of students' CGPAs i.e. the student having the maximum CGPA will be at the root of the heap.** The class definitions will look like:

```

Class StudentMaxHeap;
class Student {
    friend class StudentMaxHeap;
private:
    int rollNo;          // Student's roll number
    double cgpa;         // Student's CGPA
};
class StudentMaxHeap
{
private:
    Student* st;         // Array of students which will be arranged like a Max Heap
    int currSize;        // Current number of students present in the heap
    int maxSize;         // Maximum number of students that can be stored in the heap
public:
    StudentMaxHeap (int size); // Constructor
    ~StudentMaxHeap();        // Destructor
    bool isEmpty();           // Checks whether the heap is empty or not
    bool isFull();            // Checks whether the heap is full or not
};

```

First of all, implement the **constructor**, **destructor**, **isEmpty** and **isFull** functions shown above in the class declaration.

### Task 02 Insert in Heap:

Implement a public member function of the **StudentMaxHeap** class which inserts the record of a new student (with the given roll number and CGPA) in the Max Heap. The prototype of your function should be:

```
bool insert (int rollNo, double cgpa);
```

This function should return **true** if the record was successfully inserted in the heap and it should return **false** otherwise. The worst-case time complexity of this function should be  $O$

$(\lg n)$ . If two students have the same CGPA then their records should be stored in a way

such that at the time of removal if two (or more) students have the **same highest CGPA** then the student with **smaller roll number** should be removed **before** the students with larger roll number(s).

You can assume that Roll numbers of all students will be unique (different).

### Task 03 Remove from Heap:

Now, implement a public member function to remove that student's record from the Max Heap which has the **highest CGPA**. The prototype of your function should be:

```
bool removeBestStudent (int& rollNo, double& cgpa);
```

Before removing the student's record, this function will store the roll number and CGPA of the removed student in its two reference parameters. It should return **true** if the removal was successful and it should return **false** otherwise. The worst-case time complexity of this function should also be  $O(\lg n)$ .

#### Task 04: Student MaxHeap

Now, implement the following two public member functions of the **StudentMaxHeap** class:

```
void levelOrder ();
```

This function will perform a level order traversal of the **StudentMaxHeap** and display the rollnumbers and CGPAs of all the students.

```
int height ();
```

This function will determine and return the height of the **StudentMaxHeap**. The worst-case time complexity of this function should be **constant** i.e.  $O(1)$ .

#### Task 05 Driver program:

Now, write a menu-based driver function to illustrate the working of different functions of the **StudentMaxHeap** class. The menu should look like:

1. **Insert** a new student
2. **Remove (and display)** the student with the Max CGPA
3. **Display the list** of students (Level-order traversal)
4. **Display the height** of the heap
5. **Exit**

**Enter your choice:**

## Post-Lab Activities

### Task 01: Add Member Functions to Heap ADT.

Implement the following two public member functions of the **StudentMaxHeap** class:

```
void heapify (int i)
```

This function will convert the subtree rooted at index **i** into a Max-Heap. This function will assume that the left-subtree and the right-subtree of index **i** are already valid Max-Heaps.

```
void buildHeap (Student* st, int n)
```

This function will take an array of students (**st**) and its size (**n**) as parameters. This function should build a Max-Heap containing the records of all **n** students using the algorithm **buildHeap**

### Task 02 Heap Sort

Implement the following **global** function:

```
void heapSort (Student* st, int n) // must be implemented as a  
global function
```

This function will take an array of students (**st**) and its size (**n**) as parameters. This function should sort the array of students (**st**) into **increasing order** (according to **CGPA**), using the **Heap sort** algorithm that we have discussed in lecture. If two (or more) students have the same **CGPA** then their records should be sorted in a way that the record of the student having **smaller Roll number** should come before the record of the student having **larger Roll number**.

Also write a driver main function to test the working of the above function.

**Submissions:**

- For In-Lab Activity:
  - Save the files on your PC.
  - TA's will evaluate the tasks offline.
- For Pre-Lab & Post-Lab Activity:
  - Submit the .cpp file on Google Classroom and name it to your roll no.

**Evaluations Metric:**

- All the lab tasks will be evaluated offline by TA's
- **Division of Pre-Lab marks:** **[30 marks]**
  - Task 01: AVL implementation [30 marks]
- **Division of In-Lab marks:** **[50 marks]**
  - Task 01: Array Based Heap Implementation [10 marks]
  - Task 02: Insert into Heap [10 marks]
  - Task 03: Remove from Heap [10 marks]
  - Task 04 [15 marks]
  - Task 05: Driver Program [05marks]
- **Division of Post-Lab marks:** **[30 marks]**
  - Task 01: Add member Functions [20 marks]
  - Task 02: Heap Sort [10 marks]

**References and Additional Material:****Heap Data Structure**

<https://www.geeksforgeeks.org/heap-data-structure/>

**AVL**

<https://www.geeksforgeeks.org/introduction-to-avl-tree/>

**Lab Time Activity Simulation Log:**

- Slot – 01 – 02:00 – 00:15: Class Settlement
- Slot – 02 – 02:15 – 02:30: In-Lab Task 01
- Slot – 03 – 02:30 – 02:45: In-Lab Task 01
- Slot – 04 – 02:45 – 03:00: In-Lab Task 02
- Slot – 05 – 03:00 – 03:15: In-Lab Task 02
- Slot – 06 – 03:15 – 03:30: In-Lab Task 03
- Slot – 07 – 03:30 – 03:45: In-Lab Task 03
- Slot – 08 – 03:45 – 04:00: In-Lab Task 03
- Slot – 09 – 04:00 – 04:15: In-Lab Task 04
- Slot – 10 – 04:15 – 04:30: In-Lab Task 04
- Slot – 11 – 4:300 – 04:45: In-Lab Task 05
- Slot – 12 – 04:45 – 05:00: Discussion on Post-Lab