Lab 08: Heap implementation and operations

Objective:

- Heap implementation in C++
- Various operation of heap using C++

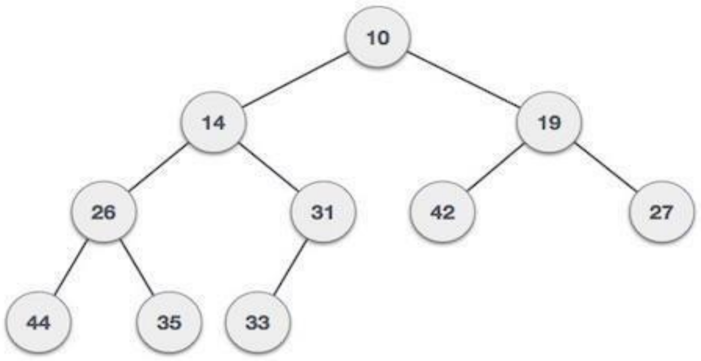
Introduction

The efficiency of the heap data structure provides itself to a surprisingly simple and very efficient sorting algorithm called heapsort. The heap sort algorithm uses a heap tree to sort an array either in ascending or descending order. It consists of two phases:

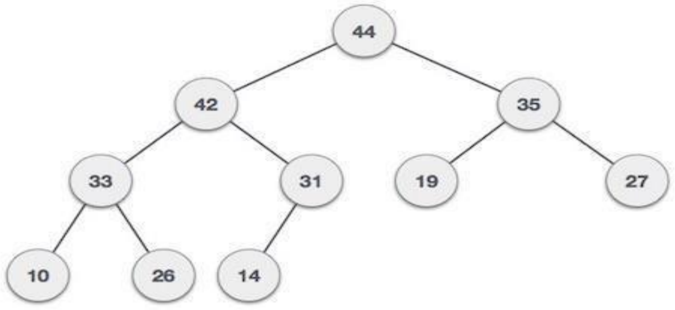
- Using unsorted data array, build the heap by repeatedly inserting each element into the heap.
- Remove the top element (item at the root) from the heap, and place it in the last location of the array. Rebuild the heap from the remaining elements of the array. This process is repeated until all items from the heap are deleted. Finally, the array is in sorted order.

A heap is a complete binary tree such that the root is the largest item (max-heap). The ordering in a heap is top-down, but not left-to-right. Each root is greater than or equal to each of its children, but some left siblings may be greater than their right siblings and some may be less. Since heaps are typically stored as arrays, we can apply the heap operations to an array of integers. We illustrate the operations as though they are being performed on binary trees, while they are really defined for the arrays that represent them by natural mapping.

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



Max-Heap − Where the value of the root node is greater 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.
```

Note − In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

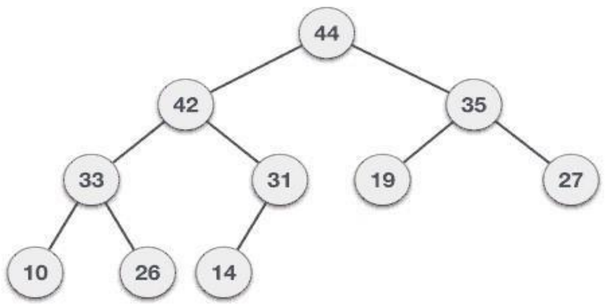
Let's understand Max Heap construction by an animated illustration. We consider the same input sample that we used earlier.

Input 35 33 42 10 14 19 27 44 26 31

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.
```



Heap (Max-Heap, Min-Heap, Insertion and Deletion)

```
#include <iostream>
using namespace std;
#define MAX 5
int heap_size=0;
int harr[MAX];
int parent(int i) { return (i-1)/2; }
// to get index of left child of node at index i
int left(int i) { return (2*i + 1); }
```

```
// to get index of right child of node at index i
int right(int i) { return (2*i + 2); }   // Inserts a new key 'k'
void insertKey_min(int k)
{     if (heap_size == MAX)
    {         cout << "\nOverflow: Could not insertKey\n";
return;
    }
      // First insert the new key at the end
         heap_size++;
         int i = heap_size - 1;
         harr[i] = k;
               // Fix the min heap property if it is violated
         while (i != 0 && harr[parent(i)] > harr[i])
         {        int temp = harr[i];
         harr[i] = harr[parent(i)];
         harr[parent(i)] = temp;
         i = parent(i);
    }
}
 // Inserts a new key 'k' void insertKey_max(int
k)
{     if (heap_size == MAX)
    {         cout << "\nOverflow: Could not insertKey\n";
return;
    }
      // First insert the new key at the end
         heap_size++;      int i = heap_size - 1;
         harr[i] = k;
```

```
   // Fix the min heap property if it is violated
while (i != 0 && harr[parent(i)] < harr[i])  {
int temp = harr[i];        harr[i] =
harr[parent(i)];        harr[parent(i)] = temp;
i = parent(i);
    }
}
// A recursive method to heapify a subtree with root at given index //
This method assumes that the subtrees are already heapified void
MinHeapify(int i)
{     int l = left(i);     int r = right(i);     int
smallest = i;     if (l < heap_size && harr[l] < harr[i])
smallest = l;     if (r < heap_size && harr[r] <
harr[smallest])         smallest = r;     if (smallest !=
i)
    {        int temp = harr[i];        harr[i] =
harr[smallest];        harr[smallest] = temp;

        MinHeapify(smallest);
    }
}
  void MaxHeapify(int i)
{     int l = left(i);     int r = right(i);     int
largest = i;     if (l < heap_size && harr[l] >
harr[i])         largest = l;     if (r < heap_size &&
harr[r] > harr[largest])         largest = r;     if
(largest != i)
    {        int temp = harr[i];
harr[i] = harr[largest];
MaxHeapify(largest);
    }
}  int delete_key()
{    if (heap_size <= 0)       return
0;      if (heap_size == 1)       {
heap_size--;         return harr[0];
    }
// Store the minimum value, and remove it from heap     int
root = harr[0];     harr[0] = harr[heap_size-1];
heap_size--;    // MinHeapify(0);
MaxHeapify(0);       return root;
```

```
}    void
display(){

      for(int i=0 ; i<heap_size ; i++){
      cout<<" "<<harr[i]<<" ,";
       }
       cout<<"\b \b";

}  int
main() {
      /*insertKey_min(3);
      insertKey_min(2);
      insertKey_min(1);
      insertKey_min(15);
      insertKey_min(5);*/
      insertKey_max(3);
      insertKey_max(2); insertKey_max(1);
      insertKey_max(15); insertKey_max(5);
      cout<<"Inserted!!"; display(); cout<<endl;
      int temp = delete_key(); if(temp == 0) {
            cout<<"\nHeap is
      Empty!!"<<endl;
      } else
      {   cout<<temp<<" Deleted!!"<<endl;
      }
      display();
      return 0;
```

Lab Tasks:

1. Revise the heap definition of above class to implement a max-heap. The member function removemin should be replaced by a new function called removemax.

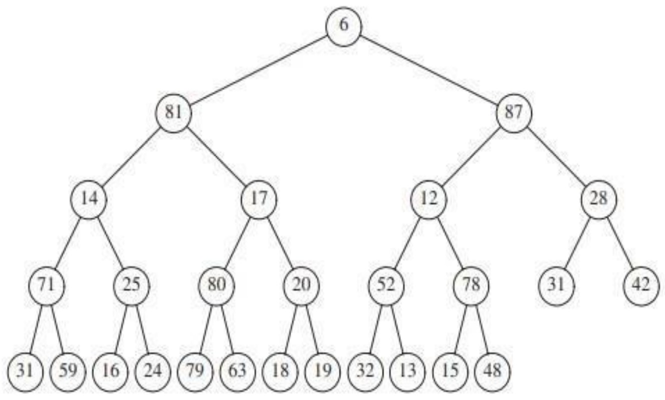
2. Build the Huffman coding tree and determine the codes for the following set of letters and weights:

Letter	A	B	C	D	E	F	G	H	I
Frequency	2	3	5	7	11	13	17	19	23

Post Lab Task

Task 2: A min-max heap is a data structure that supports both deleteMin and deleteMax in O(logN) per operation. The structure is identical to a binary heap, but the heaporder property is that for any node, X, at even depth, the element stored at X is smaller than the parent but larger than the grandparent (where this makes sense), and for any node, X, at odd depth, the element stored at X is larger than the parent but smaller than the grandparent.

a. How do we find the minimum and maximum elements?
b. Give an algorithm to insert a new node into the min-max heap.
c. Give an algorithm to perform deleteMin and deleteMax.
d. Can you build a min-max heap in linear time?
e. Suppose we would like to support deleteMin, deleteMax, and merge. Propose a data structure to support all operations in O(logN) time.


6
81
87
14
17
12
28
71
25
80
20
52
78
31
42
31
59
16
24
79
63
18
19
32
13
15
48