NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
CL218–DATA STRUCTURES LAB
Lab Session 11

Instructors: Ms. Mubashra, Ms. Aqsa

Objective:

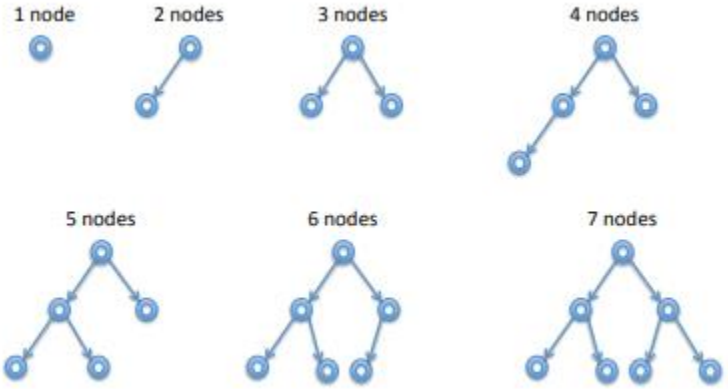
Today's lab includes heap.

What is a Heap?

- A Heap data structure is a binary tree with the following properties:

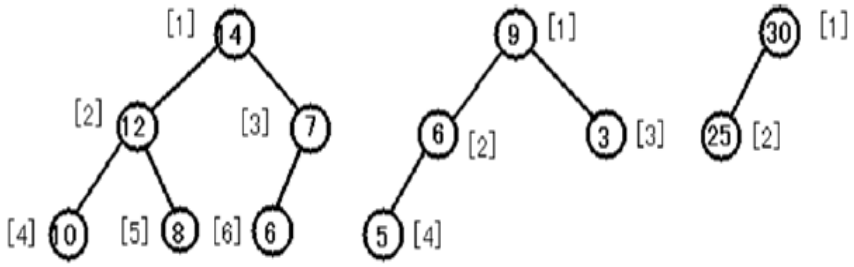
1- It is a ***complete binary tree***; that is, each level of the tree is completely filled, except possibly the bottom level. At this level, it is filled from left to right.

2- It satisfies the heap-order property: The data item stored in each node is greater than or equal to / less than or equal to the data items stored in its children.



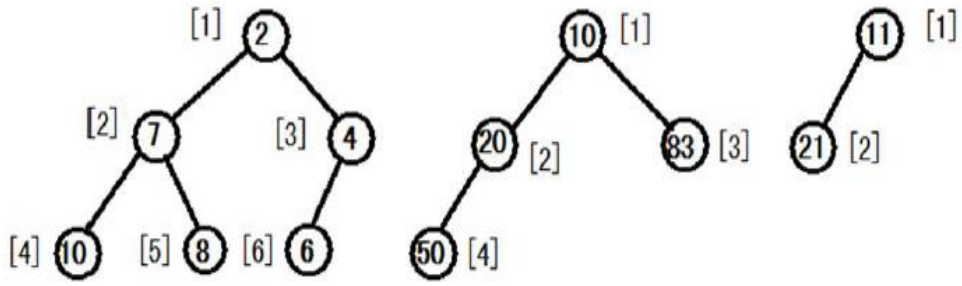
What does complete binary tree means:
A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

What does Max heap means:
If the value at the node N, is greater than or equal to the value at each of the children of N, then Heap is called a MAX-HEAP



What does Min heap means?
If the value at the node N, is less than or equal to the value at each of the children of N, then Heap is called a MIN-HEAP.

MIN HEAP

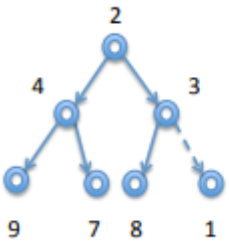


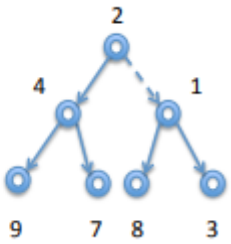
Ask me, what is invariant?

Inserting into a Heap

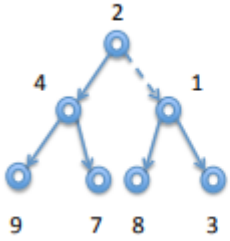
Let's consider an example. On the left is the heap before insertion of data with key 1; on the right after, but before we have restored the invariant.

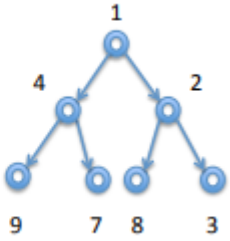
The dashed line indicates where the ordering invariant might be violated. And, indeed, 3 > 1. We can fix the invariant at the dashed edge by swapping the two nodes. The result is shown on the right.





The link from the node with key 1 to the node with key 8 will always satisfy the invariant, because we have replaced the previous key 3 with a smaller key (1). But the invariant might now be violated going up the tree to the root. And, indeed 2 > 1. We repeat the operation, swapping 1 with 2





As before, the link between the root and its left child continues to satisfy the invariant because we have replaced the key at the root with a smaller one. Furthermore, since the root node has no parent, we have fully restored the ordering invariant. In general, we swap a node with its parent if the parent has a strictly greater key. If not, or if we reach the root, we have restored the ordering invariant. The shape invariant was always satisfied since we inserted the new node into the next open place in the tree. The operation that restores the ordering invariant is called sifting up, since we take the new node and move it up the heap until the invariant has been reestablished.

Accessing the Heap Values

Given the index i of a node, the indices of its parent Parent(i), left-child LeftChild(i) and right child RightChild(i) can be computed simply :

• Parent(i)

return i/2

• LeftChild(i)

return 2i

• RightChild(i)

return 2i+1

Consider the following code:

Task1: Call its functions in main and get max heap.

Task2: With the help of max heap, now implement min heap.

```
#include<iostream>
using namespace std;
```

```
class MaxHeap{
 int *arr;        // pointer to array of elements in heap
    int capacity;    // maximum possible size of min heap
    int size;
public:

            MaxHeap(int capacity)
    {
            arr = new int[capacity]; // create an array with size capacity
  this->capacity = capacity;
  this->size = 0;    //
            }
    int getSize()
    {
            return size;
            }
            int getparent(int child)

            {
                        if(child%2==0)
                        return (child/2)-1;
                        else
                        return (child/2);
            }
            int getleft(int parent)
            {
                        return (2*parent+1);

            }
            int getright(int parent)
            {
                        return (2*parent+2);
            }
    int getMax()
    {
            for(int i = 0;i<size;i++)
  {
     cout<<arr[i]<<" ";
  }
  cout<<endl;
  return arr[0];

            }
            bool isleaf(int i)
            {

  if(i>=size)
      return true;
```

```
return false;
        }
        void siftup(int i)
        {if(i == 0)
   return;     //only one element in the array

int parent_index = getparent(i);        // get the index of the parent

if(arr[parent_index] < arr[i])
{
    int temp = arr[parent_index];   //if value at parent index is less than inserted value
    arr[parent_index] = arr[i];     // swap the values
    arr[i] = temp;
    siftup(parent_index);    // loop untill it satisfies parent child max heap relationship
}

        }
        void siftdown(int i)
        {

 int l = getleft(i);
 int r = getright(i);

 if(isleaf(l))
   return;

 int maxIndex = i;
 if(arr[l] > arr[i])
 {
  maxIndex = l;
 }

 if(!isleaf(r) && (arr[r] > arr[maxIndex]))
 {
  maxIndex = r;
 }

 if(maxIndex != i)
 {
  int temp = arr[i];
  arr[i] = arr[maxIndex];
  arr[maxIndex] = temp;
  siftdown(maxIndex);
 }
        }
 void insert(int k)
 {
        arr[size] = k;   // insert the value into array
siftup(size);
size++;     //increment the size of the array
for(int i = 0;i<size;i++)
```

```
  {
    cout<<arr[i]<<" ";
  }
  cout<<endl;
            }
   int extractMax()
   {
            int max = arr[0];
  arr[0] = arr[size - 1];

  size--;

  siftdown(0);
  return max;
            }
   int removeAt(int K)
   {
            int r = arr[K];

  arr[K] = arr[size -1];  // replace with rightmost leaf
  size-- ;
  int p = getparent(K);
  if(K == 0 || arr[K] < arr[p])
    siftdown(K);
  else
    siftup(K);
  return r;
            }
   void heapify(int *array, int len)
   {

  size = len;
  arr = array;

  for(int i=size-1; i>=0; --i)
   {
      siftdown(i);
   }
            }
};
int main()

{


            return 0;


}
```

Exercises

1. Dry run the heap code provided above on notebook.
2. Implement all heap functions: Insert, delete and others.
3. Given a binary min heap and a value x, print all the binary heap nodes having value less than the given value x.

Examples : Consider the below min heap as
common input two both below examples.

```
          2
        /    \
       3       15
     /   \    / \
    5     4  45  80
   / \   / \
  6  150 77 120
```

Input : x = 15
Output : 2 3 5 6 4

Input : x = 80
Output : 2 3 5 6 4 77 15 45

READ :

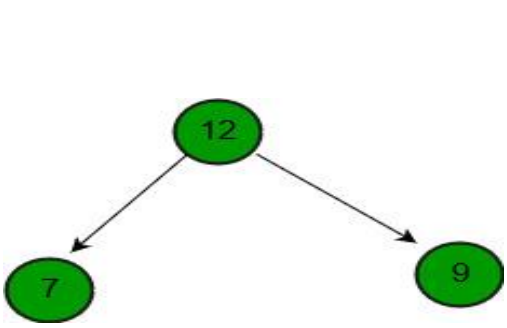
4. https://www.hackerearth.com/practice/data-structures/trees/heapspriority-queues/tutorial/
5.

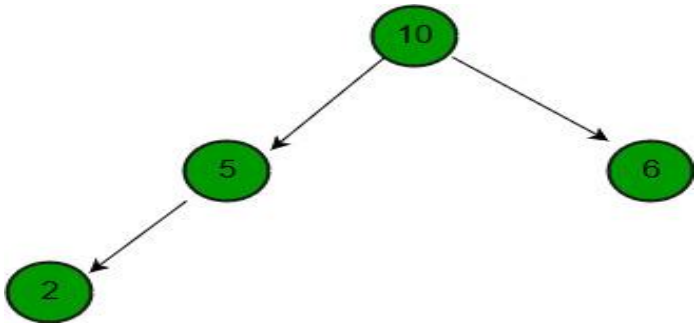
Given two binary max heaps as arrays, merge the given heaps.

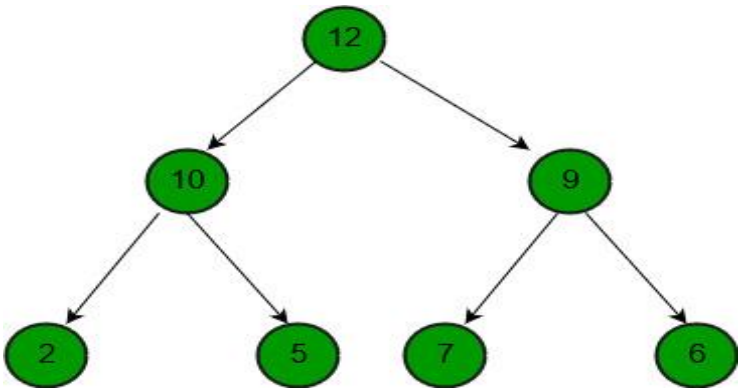
Examples :
Input : a = {10, 5, 6, 2},

b = {12, 7, 9}

Output : {12, 10, 9, 2, 5, 7, 6}






12
10
9
2
5
7
6