From DS Univ Practical Question Bank

i. **Represent a Min heap with all required operations.**

```
#include<iostream.h>
#include<conio.h>
#include<ctype.h>
#include<stdlib.h>
class heap
{
	private:
		int heaparr[20];
		int last;
		void reheapup(int *,int);
		void reheapdown(int *,int,int);
		void swap(int,int);
	public:
		heap();
		void buildheap();
		int deleteheap(int);
		void insertheap(int);
		void printheap();
};
heap::heap()
{
	cout<<"\nEnter 10 numbers";
	for(int i=0;i<10;i++)
	{
		cin>>heaparr[i];
	}
	last=9;
}
void heap::reheapup(int *a,int l)
{
	int p;
	if(l!=0)
	{
		p=(l-1)/2;
		if(a[l]<a[p])
		{
			swap(l,p);
			reheapup(a,p);
		}
	}
	return;

}
void heap::swap(int x,int y)
{
```

```
        int t;
        t=heaparr[x];
        heaparr[x]=heaparr[y];
        heaparr[y]=t;
}
void heap::reheapdown(int *a,int root,int l)
{
        int leftkey,rightkey,smallchildkey,smallchildindex,lowkey=0;
        if((root*2+1)<=l)
        {
                leftkey=a[root*2+1];
                if((root*2+2)<=l)
                        rightkey=a[root*2+2];
                else
                        rightkey=lowkey;
                if(leftkey<rightkey)
                {
                        smallchildkey=leftkey;
                        smallchildindex=(root*2+1);
                }
                else
                {
                        smallchildkey=rightkey;
                        smallchildindex=root*2+2;
                }
                if(a[root]>smallchildkey)
                {
                        swap(root,smallchildindex);
                        reheapdown(a,smallchildindex,l);
                }
        }
        return;

}
void heap::buildheap()
{
        int walker;
        walker=0;
        while(walker<last)
        {
                reheapup(heaparr,walker);
                walker=walker+1;
        }
}

int heap::deleteheap(int dataout)
{
        dataout=heaparr[0];
        heaparr[0]=heaparr[last];
        last=last-1;
```

```
        reheapdown(heaparr,0,last);
        return (dataout);

}
void heap::insertheap(int data)
{

        last=last+1;
        heaparr[last]=data;
        reheapup(heaparr,last);

}
void heap::printheap()
{
        cout<<"\n";
        for(int i=0;i<10;i++)
        {
                cout<<heaparr[i]<<"\t";
        }

}
void main()
{
        clrscr();
        heap h;
        int o,k,d,b;
        char ch;
        h.buildheap();
        do
        {

                cout<<"\n1.insert";
                cout<<"\n2.delete";
                cout<<"\n3.display";
                cout<<"\n Enter your option :" ;
                cin>>o;

                switch(o)
                {
                        case 1:
                        cout<<"\nEnter the value you want to insert";
                        cin>>k;
                        h.insertheap(k);
                        break;
                        case 2:
                        b=h.deleteheap(d);
                        cout<<b<<"has ben deleted";
                        break;
                        case 3:
                        h.printheap();
```

```
                        break;
                }

                cout<<"\n\nDo u wish to continue???";
                cin>>ch;

        }while(tolower(ch)=='y');


}
```

ii. Represent a max heap with all the required operations.

```
#include<iostream.h>
#include<conio.h>
#include<ctype.h>
#include<stdlib.h>
class heap
{
private:
int heaparr[20];
int last;
void reheapup(int *,int);
void reheapdown(int *,int,int);
void swap(int,int);
public:
heap();
void buildheap();
int deleteheap(int);
void insertheap(int);
void printheap();
};
heap::heap()
{
cout<<"\nEnter 10 numbers";
for(int i=0;i<10;i++)
{
cin>>heaparr[i];
}
last=9;
}
void heap::reheapup(int *a,int l)
{
int p;
if(l!=0)
{
p=(l-1)/2;
if(a[l]>a[p])
```

```
{
swap(l,p);
reheapup(a,p);
}
}
return;
}
void heap::swap(int x,int y)
{int t;
t=heaparr[x];
heaparr[x]=heaparr[y];
heaparr[y]=t;
}
void heap::reheapdown(int *a,int root,int l)
{
int leftkey,rightkey,childkey,childindex,key=0;
if((root*2+1)<=l)
{
leftkey=a[root*2+1];
if((root*2+2)>=l)
rightkey=a[root*2+2];
else
rightkey=key;
if(leftkey>rightkey)
{
childkey=leftkey;
childindex=(root*2+1);
}
else
{
childkey=rightkey;
childindex=root*2+2;
}
if(a[root]<childkey)
{
swap(root,childindex);
reheapdown(a,childindex,l);
}
}
return;
}
void heap::buildheap()
{
int walker;
walker=0;
while(walker<last)
{
reheapup(heaparr,walker);
walker=walker+1;
}
```

```
}
int heap::deleteheap(int dataout)
{
dataout=heaparr[0];
heaparr[0]=heaparr[last];
last=last-1;
reheapdown(heaparr,0,last);
return (dataout);
}void heap::insertheap(int data)
{
last=last+1;
heaparr[last]=data;
reheapup(heaparr,last);
}
void heap::printheap()
{
cout<<"\n";
for(int i=0;i<10;i++)
{
cout<<heaparr[i]<<"\t";
}
}
void main()
{
clrscr();
heap h;
int o,k,d,b;
char ch;
h.buildheap();
do
{
cout<<"\n1.insert";
cout<<"\n2.delete";
cout<<"\n3.display";
cout<<"\n Enter your option :" ;
cin>>o;
switch(o)
{
case 1:
cout<<"\nEnter the value you want to insert";
cin>>k;
h.insertheap(k);
break;
case 2:
b=h.deleteheap(d);
cout<<b<<"has ben deleted";
break;
case 3:
h.printheap();
break;
```

```
}
cout<<"\n\nDo u wish to continue???";
cin>>ch;
}while(tolower(ch)=='y');
}
```

iii. Perform Heap Sort on a given array.

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>

#define size 7
#define lowKey 0

class heap
{
	private:
		int arr[20];
		int last;
		void reheapup(int *,int);
		void reheapdown(int *,int,int);
	public:
		heap();
		void buildheap(int *,int);
		void heapsort(int *,int);
		void printheap(int *,int);
};

void heap::heapsort(int *arr,int last)
{
	int sort=last,temp;
	while(sort>0)
	{
		temp=arr[0];
		arr[0]=arr[sort];
		arr[sort]=temp;
		sort--;
		reheapdown(arr,0,sort);
	}
	return;
}

heap::heap()
{
	int i=0,ch;
	cout<<"Enter 7 elements:";
	for(i=0;i<size;i++)
		cin>>arr[i];
```

```
        last=size-1;
        buildheap(arr,last);
        cout<<"\n\nHeap before sorting:\n";
        printheap(arr,last);
        heapsort(arr,last);
        cout<<"\n\nHeap after sorting:\n";
        printheap(arr,last);
}

void heap::buildheap(int *arr,int last)
{
        int walker=1;
        for(walker=1;walker<=last;walker++)
        {
                reheapup(arr,walker);
        }
}

void heap::reheapup(int *arr,int last)
{
        int parent,temp;
        if(last!=0)
        {
                parent=(last-1)/2;
                if(arr[last]>arr[parent])
                {
                        temp=arr[last];
                        arr[last]=arr[parent];
                        arr[parent]=temp;
                        reheapup(arr,parent);
                }
        }
}

void heap::reheapdown(int *arr,int root,int last)
{
        int temp,leftKey,rightKey,largeChildKey,largeChildIndex;
        if((root*2)+1<=last)
        {
                leftKey=arr[(root*2)+1];
                if((root*2)+2<=last)
                        rightKey=arr[(root*2)+2];
                else
                        rightKey=lowKey;
                if(leftKey>rightKey)
                {
                        largeChildKey=leftKey;
                        largeChildIndex=root*2+1;
                }
                else
```

```
            {
                    largeChildKey=rightKey;
                    largeChildIndex=root*2+2;
            }
            if(arr[root]<arr[largeChildIndex])
            {
                    temp=arr[root];
                    arr[root]=arr[largeChildIndex];
                    arr[largeChildIndex]=temp;
                    reheapdown(arr,largeChildIndex,last);
            }

    }
    else
            return;
}

void heap::printheap(int *arr,int sz)
{
    int x;
    for(x=0;x<=sz;x++)
            cout<<arr[x]<<"\t";
}

void main()
{
    clrscr();
    heap h;
    getch();
}
```

iv. Perform the select-k operation on a heap where k is the input given by the user.

```
#include<iostream.h>
#include<conio.h>
#include<ctype.h>
#include<stdlib.h>
class heap
{
    private:
    int heaparr[20];
    int last;
    void reheapup(int *,int);
    void reheapdown(int *,int,int);
    void swap(int,int);
    public:
    heap();
    void buildheap();
```

```
        int deleteheap(int);
        void insertheap(int);
        void selectk(int);
        void printheap();
};

heap::heap()
{
        cout<<"\nEnter 10 numbers";
        for(int i=0;i<10;i++)
        {
                cin>>heaparr[i];
        }
                last=9;
}

void heap::reheapup(int *a,int l)
{
        int p;
        if(l!=0)
        {
                p=(l-1)/2;
                if(a[l]<a[p])
                {
                        swap(l,p);
                        reheapup(a,p);
                }
        }
        return;
}

void heap::swap(int x,int y)
{       int t;
        t=heaparr[x];
        heaparr[x]=heaparr[y];
        heaparr[y]=t;
}

void heap::reheapdown(int *a,int root,int l)
{
        int leftkey,rightkey,childkey,childindex,key=0;
        if((root*2+1)<=l)
        {
                leftkey=a[root*2+1];
                if((root*2+2)>=l)
                rightkey=a[root*2+2];
                else
                rightkey=key;
                if(leftkey<rightkey)
                {
```

```
                    childkey=leftkey;
                    childindex=(root*2+1);
            }
            else
            {
                    childkey=rightkey;
                    childindex=root*2+2;
            }
            if(a[root]>childkey)
            {
                    swap(root,childindex);
                    reheapdown(a,childindex,l);
            }
    }
return;
}

void heap::selectk (int k)
{
int dataout,temp;
    if(k>last)
    {
            cout<<"\nheap size is less";
            return;
    }
    while(k>0)
    {

            dataout=heaparr[0];
            temp=deleteheap(dataout);
            last=last+1;
            heaparr[last]=temp;
            cout<<endl<<"Dtat deleted"<<temp;
            k--;
    }
}


void heap::buildheap()
{
    int walker;
    walker=0;
    while(walker<last)
    {
            reheapup(heaparr,walker);
            walker=walker+1;
    }
}

int heap::deleteheap(int dataout)
```

```
{
        dataout=heaparr[0];
        heaparr[0]=heaparr[last];
        last=last-1;
        reheapdown(heaparr,0,last);
        return (dataout);
}

void heap::insertheap(int data)
{
        last=last+1;
        heaparr[last]=data;
        reheapup(heaparr,last);
}

void heap::printheap()
{
cout<<"\n";
for(int i=0;i<10;i++)
{
cout<<heaparr[i]<<"\t";
}
}
void main()
{
clrscr();
heap h;
int o,k,d,b,n;
char ch;
h.buildheap();
do
{
cout<<"\n1.insert";
cout<<"\n2.delete";
cout<<"\n3.display";
cout<<"\n4.Select k operation";
cin>>o;
switch(o)
{
case 1:
cout<<"\nEnter the value you want to insert";
cin>>k;
h.insertheap(k);
break;
case 2:
b=h.deleteheap(d);
cout<<b<<"has ben deleted";
break;
case 3:
h.printheap();
```

```
break;
case 4: cout<<"\n Enter k ";
        cin>>n;
        h.selectk(n);
        getch();
        break;
}
cout<<"\n\nDo u wish to continue???";
cin>>ch;
}while(tolower(ch)=='y');}
```

From DS Univ Theory Question Bank

v. **Define the properties of a heap. Depict a max heap with a simple diagram. State how the left child, right child and parent can be arithmetically derived in a heap with the examples of an heaptree converted into an array.**

Ans. 1)A heap is a binary tree structure with the following properties:-

a)The tree is complete or nearly complete.

b)The key value of each node is greater than or equal to the key value in each of its descendents.

2)A heap is generally a max heap though the properties can be reversed to create a min hep where the key value in a node is less than equal to the key values in all of its subtrees.

3)Unlike Binary Search Tree, the smaller nodes of a heap can be placed on either the right or the left subtree. Therefore, both the left and right branches of the tree have the same meaning.

4)Heaps are generally implemented as an array.

5)The relationship between a nodeand its children is fixed and can be calculated as:

(i)For a node located at index I, its children are found at :

a. leftchild=2i + 1

b. rightchild=2i + 2

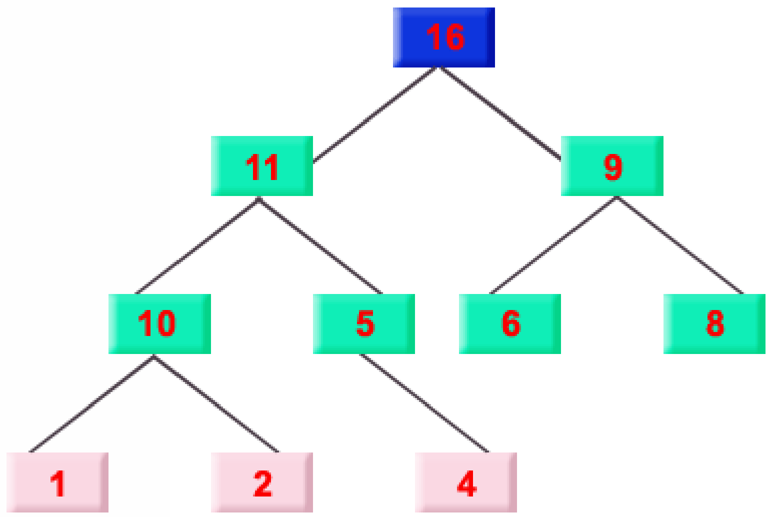
(ii)For a node located at index I, its parent is located at [(i-1)/2]

(iii)Given the index for a left child j, its right sibling if any is found at j+1. Conversely, given the index for a right child k, its left sibling which must exist, is found at k-1.

(iv)Given the size, n of a complete heap, the location of the first leaf is n/2. Given the location of the first leaf element, the location of the last non-leaf element is one less.

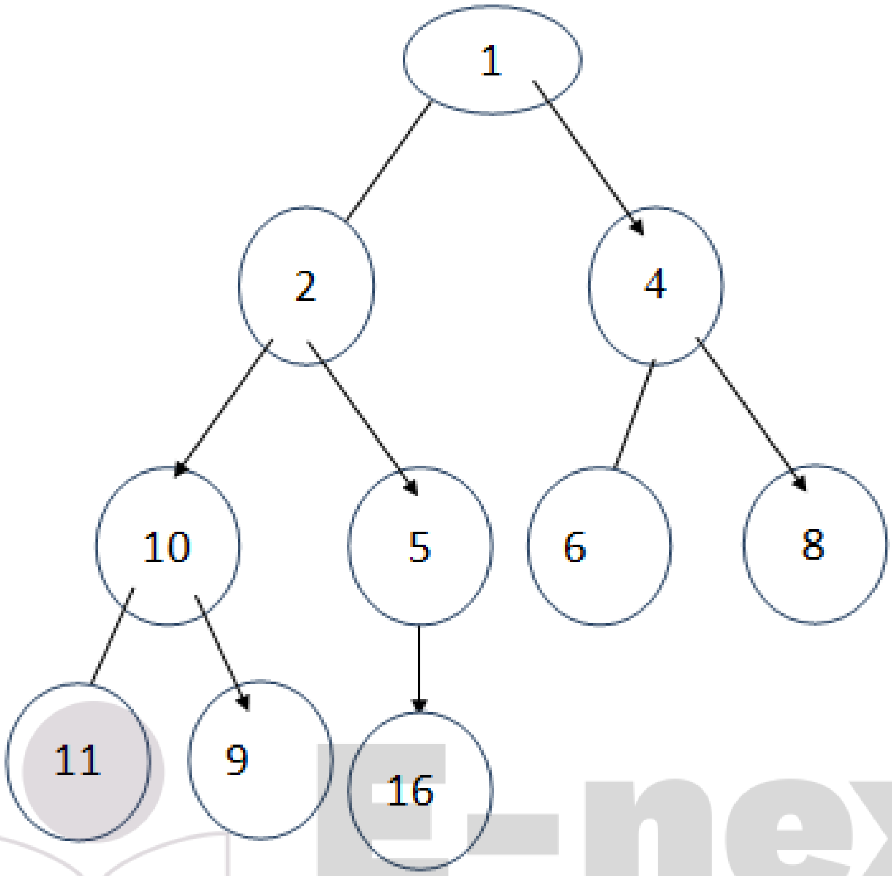
5)In short, a heap is a complete or nearly complete binary tree in which the key value in a node is greater than the key values in all of its subtrees and the subtrees are in turn heaps

Max heap with a simple diagram.



Left child and Right child and Parent node calculated in an array.

Let the array that is converted from a heaptree be 1,2,4,9,5,6,8,11,10,16
The left child calculated is by this formula:- leftkey=a[root*2+1];
where root is the index of parent node.
Similarily, right child of parent node is calculated through this formula :-
rightkey=a[root*2+2];



Now through an array

1) a[0] is the parent node. so it's left child will be :-

    root=0

    leftkey=a[root*2+1]

        it will be equal to a[1]

        so, leftchild is a[1] i.e. 2 from the above array

    through tree we can see that leftchild is 2 of 1

2) now we will calculate right child of a[0]

    root=0

    rightkey=a[root*2+2]

        it will be equal to a[2] i.e. 4 from the above array

    so, right child of a[0] is 4

    through tree we can see that rightchild is 4 of 1

3) now we will calculate for randomly a[4]

    so let's checkout what is left child

    leftkey=a[root*2+1]

    root=4

    so leftchild will be a[9]

    from above array i.e. a[9]=16

    so left child of a[4] is 16.

    through tree we can see that leftchild is 16 of 5

So, like this we can calculate the left and right child from an array.

vi. Write algorithms for the following heap operations:

a) Algorithm reHeapUp:-

```
Algorithm ReheapUp(heap, newNode)
Reestablishes heap by moving data in child up to its correct location in the heap array
Pre     heap is array containing an invalid heap
        newNode is index location to new data in heap
Post    heap has been recordered
        1  if(newNode  not the root)
              1 Set parent to parent of new node
              2 if(newNode key > parent key)
                    1 exchange newNode and parent
                    2 reheapUp(heap, parent)
              3 end if
        2  end if
end ReheapUp
```

b) Algorithm reHeapDown :-

```
Algorithm ReheapDown (heap, root, last)
Reestablishes heap by moving data in child up to its correct location in the heap array
Pre     heap is an array of data
        Root is root of heap or subheap
        Last is an index to the last element of the heap
Post    heap has been restored
Determine which child has larger key
        1 if (there is left subtree)
              1 set leftKey to left subtree key
              2 (there is right subtree)
                    1 set rightKey to right subtree key
              3 else
                    1 set rightKey to null key
              4 end if
              5 if(leftKey > rightKey)
                    1 set largeSubtree to left subtree
              6 else
                    1 set largeSubtree to right subtree
              7 end if
              8 if( root < largesubtree key)
                    1 exchange root and largeSubtree
                    2 reheapDown(heap,largeSubtree,last)
              9 end if
        2 end if
end reheapDown
```

c) Algorithm for deleteHeap:-

Algorithm DeleteHeap(heap, last, dataOut)
Delete root of heap and passes data back to caller
Pre heap is valid heap structure
Last is reference parameter to last node in the heap
dataOut is reference parameter for output area
Post root deleted and heap rebuilt
Root data placed n dataOut
Return true if successful ; false if array empty

```
        1 if (empty heap)
                1 return FALSE
        2 end if
        3 set dataOut to root data
        4 move last data to root
        5 decrement last
        6 reheapDown(heap , 0 , last)
        7 return true
end deleteHeap
```

d) Algorithm for insertHeap:-

Algorithm InsertHeap(heap, last, data)
Insert data in the heap
Pre heap is valid heap structure
Last is reference parameter to last node in the heap
Data contain data to be inserted into heap
Post data have been inserted in to the heap
Return true if successful ; false if array empty

```
        1 if (heap full)
                1 return false
        2 end if
        3 increment last
        4 move data to last node
        5 reheapUp(heap , last)
        6 return true
end InsertHeap
```

e) Algorithm for buildHeap:-

Algorithm buildHeap (heap , size)
Given an array , rearrange data so that they form a heap
Pre heap is array containing data in nonheap order
Size is number of element inarray
Post array is now a heap

```
        1 set walker to 1
        2 loop (walker < size)
                1 reheapUp(heap , walker)
                2 increment walker
        3 end loop
end buildHeap
```

f) Algorithm for select kth element in heap:-

```
Algorithm selectK (heap , k , heapLast)
Select the k-th largest element from a list
Pre     heap is an array implementation of heap
        K is the ordinal of the element desired
        heapLast is the reference parameter to last element
post    k-th largest value returned
        1 if( k > heap size)
                1 return false
        2 end if
        3 set origHeapSize to heapLast + 1
        4 loop (k time)
                1 set tempData to root data
                2  DeleteHeap(heap ,  last ,  dataOut)
                3 move tempData to heapLast + 1
        5 end loop
        6 move root data to holdout
        7 loop(while heapLast < origHeapSize)
                1 Increment heapLast
                2 reheapUp(heap , heapLast)
        8 end loop
        9 return holdout
end selectK
```

vii. Write a C++ program for priority Queue.

```
//Priority queue
#include<iostream.h>
#include<conio.h>
#include<string.h>
#include<assert.h>
#include<stdlib.h>
const int SIZE=20;
class PriorityQ
{
   private:
        char pri_Q[SIZE][6];
        int last;
        void reheapUp(char[][6], int);
        void reheapDown(char[][6],int,int);
   public:
        PriorityQ();
        void insertPriorityQ(char *);
        char *deletePriorityQ();
        void displayPriQ();
};
```

```
PriorityQ::PriorityQ()
{
   last=-1;
   for(int i=0;i<SIZE;i++)
           pri_Q[i][0]='\0';
}

void PriorityQ::insertPriorityQ(char *k)
{
   last++;
   if(last==0)
   {
           strcpy(pri_Q[last],k);
           return;
   }
   if(last<20)
   {
           strcpy(pri_Q[last],k);
           reheapUp(pri_Q,last);
   }
}

char *PriorityQ::deletePriorityQ()
{
   char *temp=new char[6];
   assert(temp);
   strcpy(temp,pri_Q[0]);
   strcpy(pri_Q[0],pri_Q[last]);
   last--;
   reheapDown(pri_Q,0,last);
   return(temp);
}

void PriorityQ::displayPriQ()
{
   for(int i=0;i<=last;i++)
           cout<<pri_Q[i]<<"\t";
}
void PriorityQ::reheapUp(char pri_Q[][6],int newNode)
{
   int pri_child_num,pri_parent_num,key_child_num,key_parent_num,parent;
   char pri_child_char[2],pri_parent_char[2],key_child_char[2],key_parent_char[2];

   parent=(newNode-1)/2;
   //initialization of priority number
   pri_child_char[0]=pri_Q[newNode][0];
   pri_child_char[1]='\0';
   pri_parent_char[0]=pri_Q[parent][0];
   pri_parent_char[1]='\0';
   pri_child_num=atoi(pri_child_char);
```

```
    pri_parent_num=atoi(pri_parent_char);

    //initialization of key numbers within the same priority
    key_child_char[0]=pri_Q[newNode][2];
    key_child_char[1]='\0';
    key_parent_char[0]=pri_Q[parent][2];
    key_parent_char[1]='\0';
    key_child_num=atoi(key_child_char);
    key_parent_num=atoi(key_parent_char);

    if(newNode!=0)
    {
            parent=(newNode-1)/2;
            if(pri_child_num > pri_parent_num)
            {
                    char temp[6];
                    strcpy(temp, pri_Q[parent]);
                    strcpy(pri_Q[parent], pri_Q[newNode]);
                    strcpy(pri_Q[newNode], temp);
                    reheapUp(pri_Q,parent);
            }
            else if(pri_child_num == pri_parent_num)
            {
                    if(key_child_num > key_parent_num)
                    {
                            char temp[6];
                            strcpy(temp, pri_Q[parent]);
                            strcpy(pri_Q[parent], pri_Q[newNode]);
                            strcpy(pri_Q[newNode], temp);
                            reheapUp(pri_Q,parent);
                    }
            }
    }
    return;
} //end reheapUp

void PriorityQ::reheapDown(char pri_Q[][6], int root, int last)
{
    char left_pri_char[2], right_pri_char[2],
             left_key_char[2], right_key_char[2],
             largeChildKey[6];
    int  left_pri_num, right_pri_num,
             left_key_num, right_key_num,
             largeChildIndex;

    if(root*2+1 <= last)
    {
            //There is atleast one child
            left_pri_char[0]=pri_Q[root*2+1][0];
            left_pri_char[1]='\0';
```

```
        left_pri_num=atoi(left_pri_char);

        left_key_char[0]=pri_Q[root*2+1][2];
        left_key_char[1]='\0';
        left_key_num=atoi(left_key_char);

        if(root*2+2 <= last)
        {
                //There is a right child
                right_pri_char[0]=pri_Q[root*2+2][0];
                right_pri_char[1]='\0';
                right_pri_num=atoi(right_pri_char);

                right_key_char[0]=pri_Q[root*2+2][2];
                right_key_char[1]='\0';
                right_key_num=atoi(right_key_char);
        }
        else
        {
                right_pri_num = -1;
                right_key_num = -1;
        }

        if(left_pri_num > right_pri_num)
        {
                strcpy(largeChildKey, pri_Q[root*2+1]);
                largeChildIndex = root*2+1;
        }
        else if(left_pri_num == right_pri_num)
        {
                if(left_key_num > right_key_num)
                {
                        strcpy(largeChildKey, pri_Q[root*2+1]);
                        largeChildIndex = root*2+1;
                }
        }
        else
        {
                strcpy(largeChildKey, pri_Q[root*2+2]);
                largeChildIndex = root*2+2;
        }

//Test if root < larger subtree
//If yes SWAP and Call reheapDown

int pri_child_num,pri_parent_num,key_child_num,key_parent_num,parent;
char pri_child_char[2],pri_parent_char[2],key_child_char[2],key_parent_char[2];

parent=root;
//initialization of priority number
```

```
    pri_child_char[0]=pri_Q[largeChildIndex][0];
    pri_child_char[1]='\0';
    pri_parent_char[0]=pri_Q[parent][0];
    pri_parent_char[1]='\0';
    pri_child_num=atoi(pri_child_char);
    pri_parent_num=atoi(pri_parent_char);

    //initialization of key numbers within the same priority
    key_child_char[0]=pri_Q[largeChildIndex][2];
    key_child_char[1]='\0';
    key_parent_char[0]=pri_Q[parent][2];
    key_parent_char[1]='\0';
    key_child_num=atoi(key_child_char);
    key_parent_num=atoi(key_parent_char);


        if(pri_child_num > pri_parent_num)
        {
            char temp[6];
            strcpy(temp, pri_Q[parent]);
            strcpy(pri_Q[parent], pri_Q[largeChildIndex]);
            strcpy(pri_Q[largeChildIndex], temp);
            reheapDown(pri_Q, largeChildIndex, last);
        }
        else if(pri_child_num == pri_parent_num)
        {
            if(key_child_num > key_parent_num)
            {
                char temp[6];
                strcpy(temp, pri_Q[parent]);
                strcpy(pri_Q[parent], pri_Q[largeChildIndex]);
                strcpy(pri_Q[largeChildIndex], temp);
                reheapDown(pri_Q, largeChildIndex, last);
            }
        }
    }
    return;
}  //end reheapDown

void main()
{
    char arr[10][6] = {"3.9-A", "5.9-B", "3.8-C", "2.9-D", "1.9-E", "2.8-F", "3.7-G",
"2.7-H", "2.6-I", "2.5-J"};
    clrscr();

    PriorityQ PQ;

    for(int i=0; i<10; i++)
    {
        PQ.insertPriorityQ(arr[i]);
```

```
    }
    cout<<"\nPriority Queue at the start:\n";
    PQ.displayPriQ();
    char *temp=PQ.deletePriorityQ();
    cout<<"\nAfter deletion of "<<temp<<":\n";
    PQ.displayPriQ();
    PQ.insertPriorityQ("4.9-K");
    cout<<"\n\nAfter insertion of 4.9-K:\n";
    PQ.displayPriQ();
    getch();
}
```

viii. Solve the following University Questions:

a) Year – 2011(MAY): Q1(A) 10 Marks

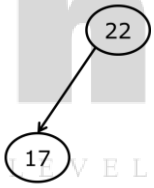
Given the following set of numbers, implement heap sort on this array. Show the resulting array after every pass.

22 17 19 15 13 14 42 23 12 91

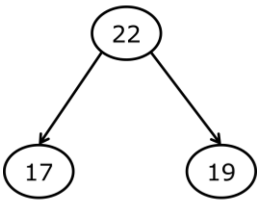
Solution:

Step 1:

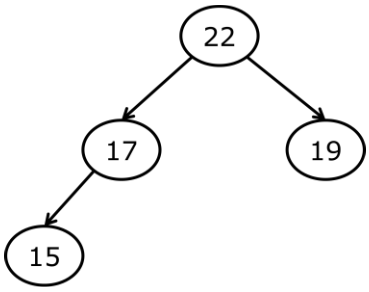
Step 2:



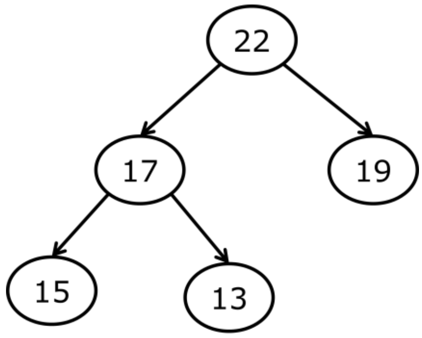
Step 3:



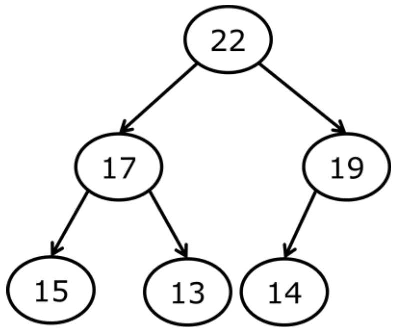
Step 4:



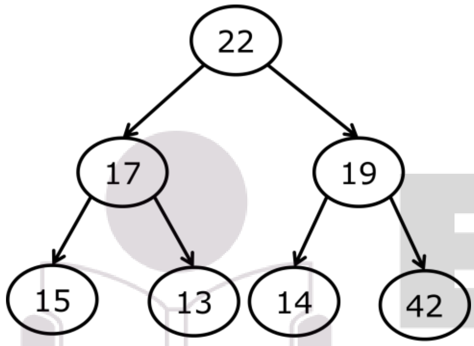
Step 5:



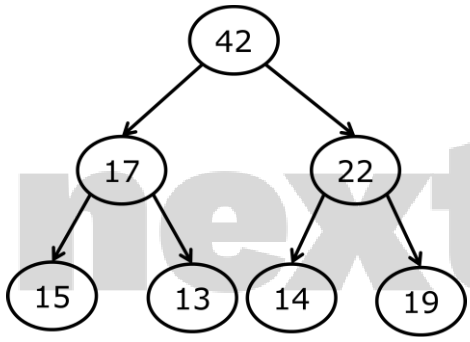
Step 6:



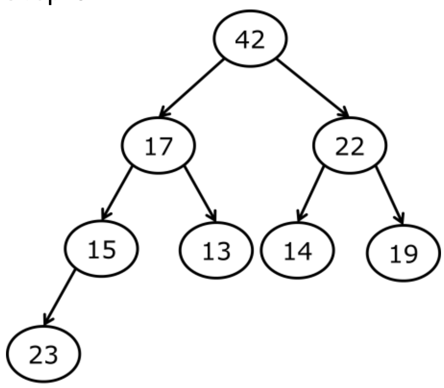
Step 7:



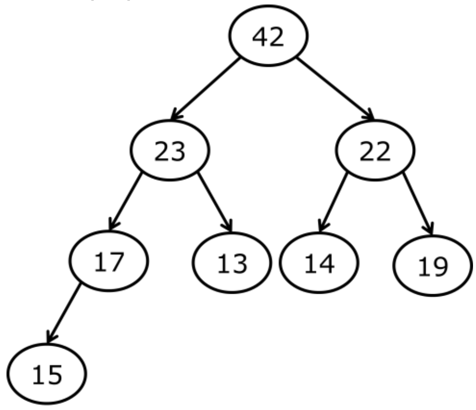
ReHeapUp:



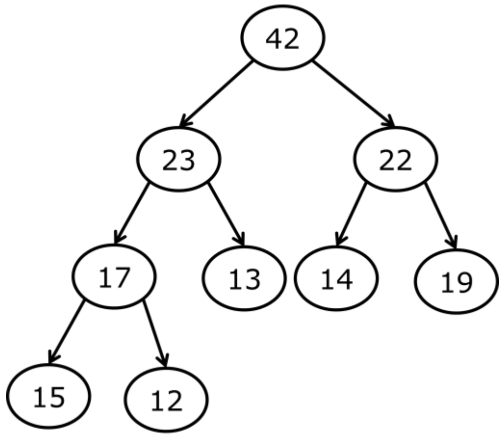
Step 8:



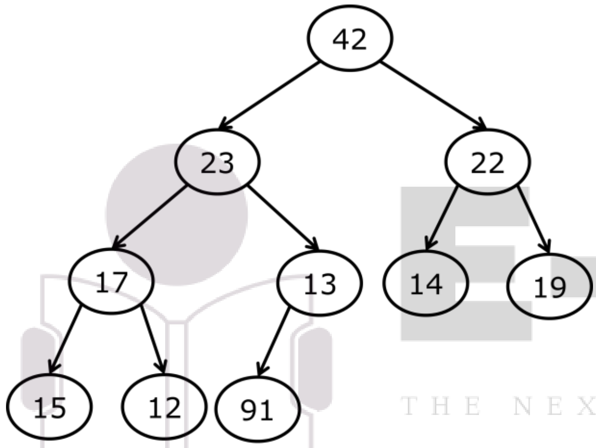
ReHeapUp:



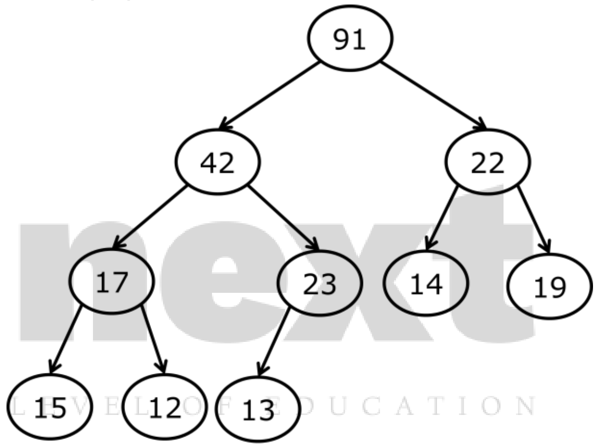
Step 9:



Step 10:



ReHeapUp:



Sorting the array:

1st pass:										
91	42	22	17	23	14	19	15	12	13	
2nd pass:										
42	23	22	17	13	14	19	15	12	91	
3rd pass:										
23	17	22	15	13	14	19	12	42	91	
4th pass:										
22	17	19	15	13	14	12	23	42	91	

5th pass

19	17	14	15	13	12	22	23	42	91

6th pass:

17	15	14	12	13	19	22	23	42	91

7th pass:

15	13	14	12	17	19	22	23	42	91

8th pass:

14	13	12	15	17	19	22	23	42	91

9th pass:

13	12	14	15	17	19	22	23	42	91

10th pass:

12	13	14	15	17	19	22	23	42	91

Year – 2010(MAY): Q1(A) 10 Marks

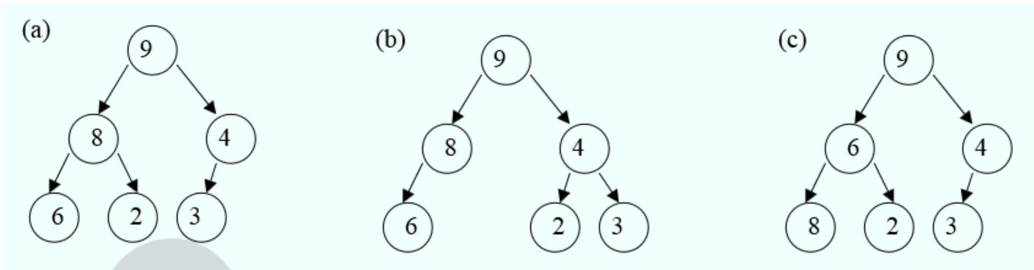
b) Explain the heap as a data structure? Build a maxheap by inserting the following values in the heap-
16 31 5 22 45 74 2 42

Solution:

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

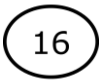
1. It is a complete binary tree; that is, each level of the tree is completely filled, except possibly the bottom level. At this level, it is filled from left to right.
2. It satisfies the heap-order property: The data item stored in each node is greater than or equal to the data items stored in its children.

Examples:

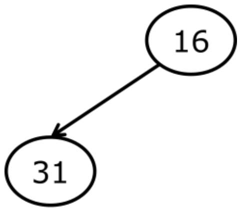


In the above examples:
(a) is a heap
(b) is not a heap as it is not complete and
(c) is complete but does not satisfy the second property defined for heaps.

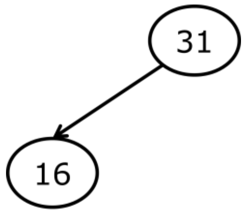
Step 1:



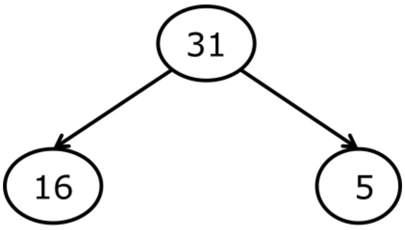
Step 2:



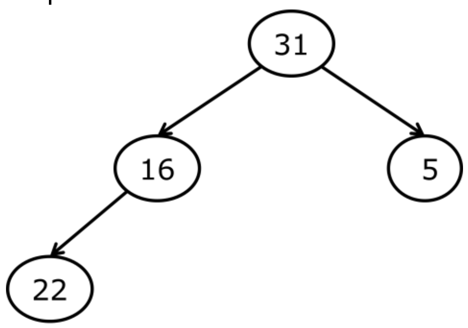
ReHeapUp



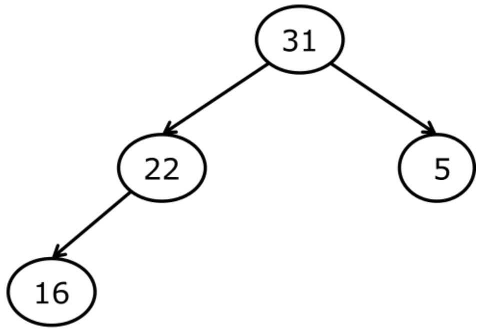
Step 3:



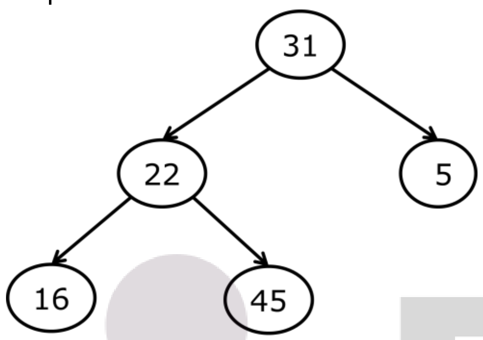
Step 4:



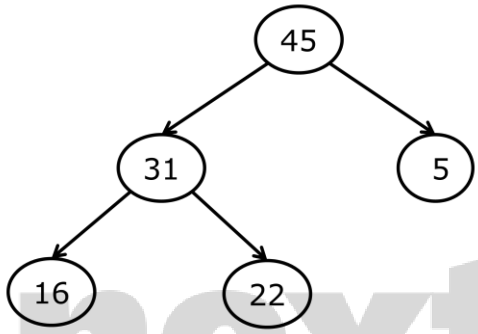
ReHeapUp:



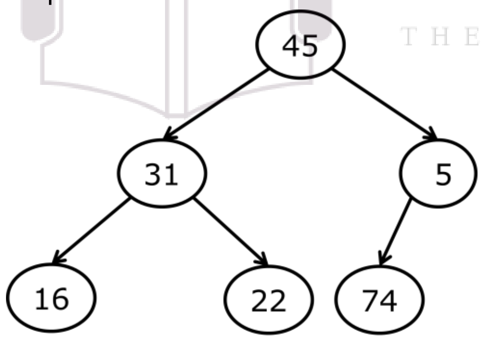
Step 5:



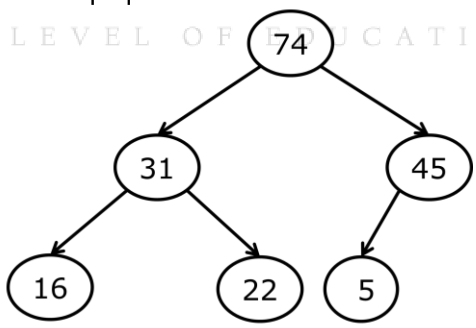
ReHeapUp:



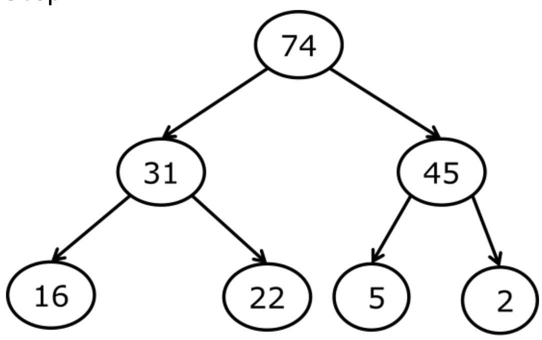
Step 6:



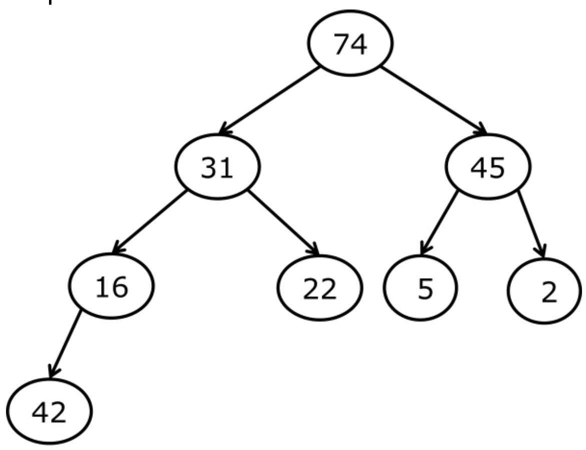
ReHeapUp:



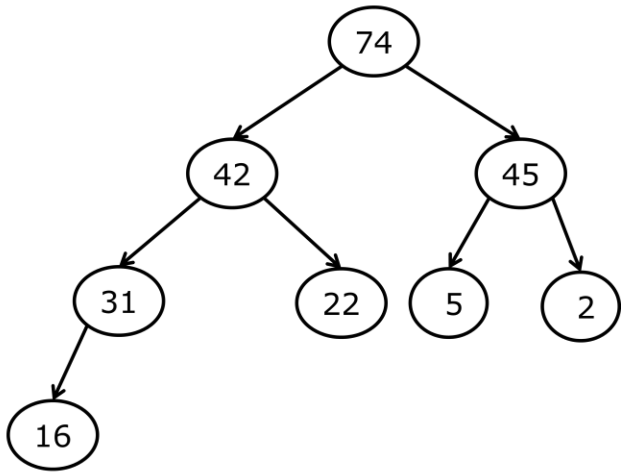
Step 7:



Step 8:



ReHeapUp:



c) **M2010-Q7 a) i) Priority Queue**

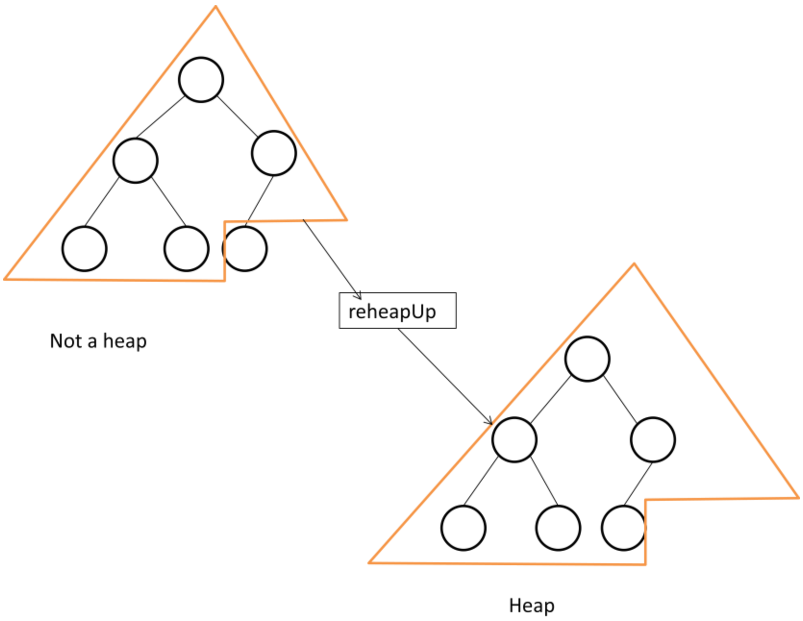
d) **Year – 2010(DEC): Q3(B) 10 Marks**

Define ReheapUp operation for a heap. Create a max heap using following:-

42 , 23 , 74 , 11 , 65 , 3 , 94 , 36 , 99 , 87.

Ans:-

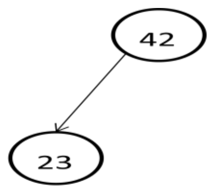
Imagine that we have a nearly complete binary tree with N elements whose first N – 1 elements satisfy the order property of heaps, but the last elements does not.
The reheap up operation repairs the structure so that it is heap by last floating the last element up the tree until that element is in its correct location in tree.



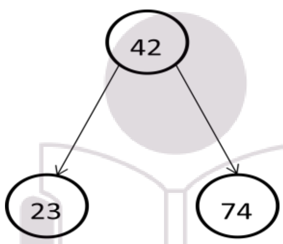
As you can see in figure before we reheap up, the last node in the heap was out of order. After the reheap, it is correct location and the heap has extended one node.

Like the binary search tree, inserts into heaps take place at a leaf, furthermore, because the heap is complete or nearly complete tree, the node must be placed in the last leaf level at the first empty position .
Reheap up operation in a heap . at the beginning we observe that 25 is greater that its parent’s key.12. because 25 is greater that 12, we also know from the definition of heap that it is greater that the parent’s left subtree keys . we therefore exchange 25 and 12 and call reheap up to test its current position in the heap . once again, 25 is greater that its parent’s key, 21 . therefore , we   again exchange the node data. This time when reheap up is called , the value of the current nodes key is less than the value of its parent key, indicating that we have located the correct position and the operation stop.

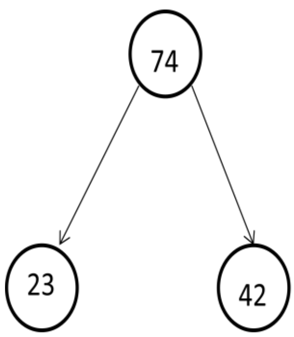
Insert 42



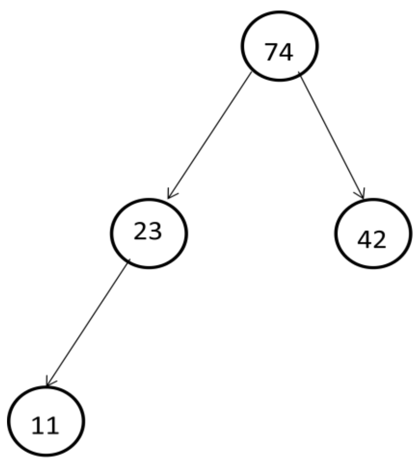
Insert 23



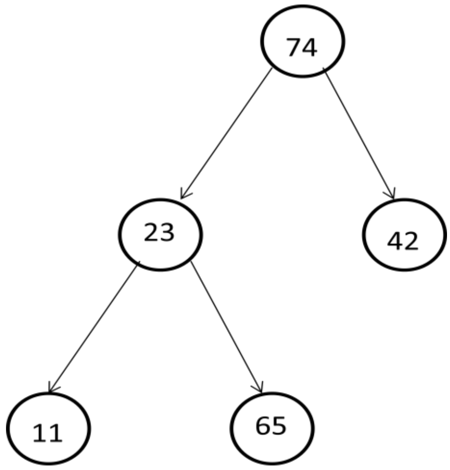
Insert 74



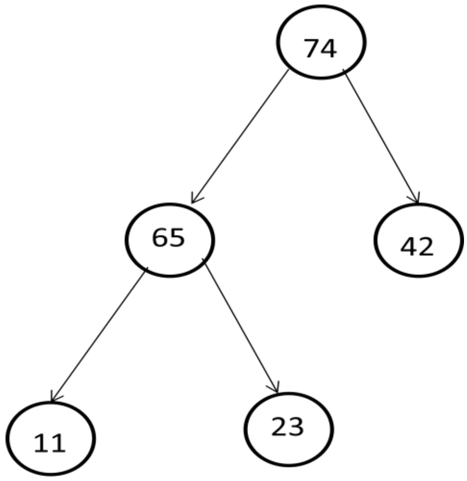
ReheapUp



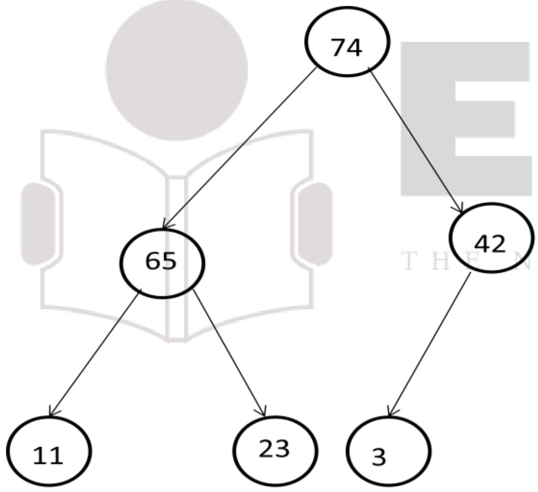
Insert 11



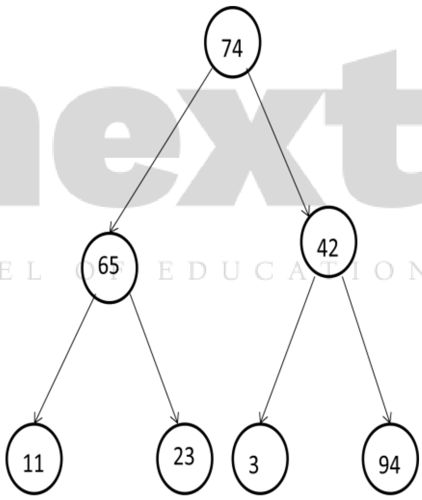
Insert 65



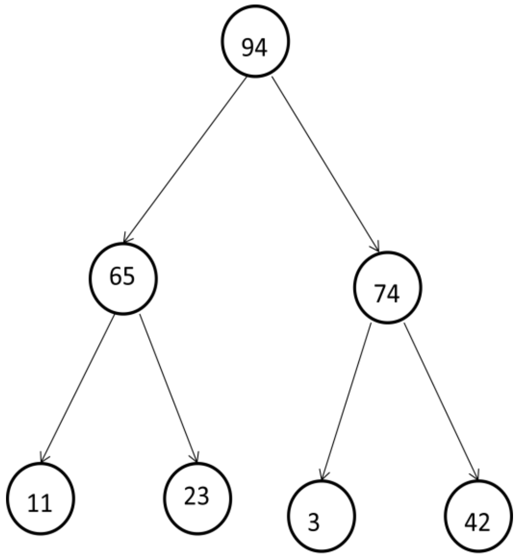
reheapUp



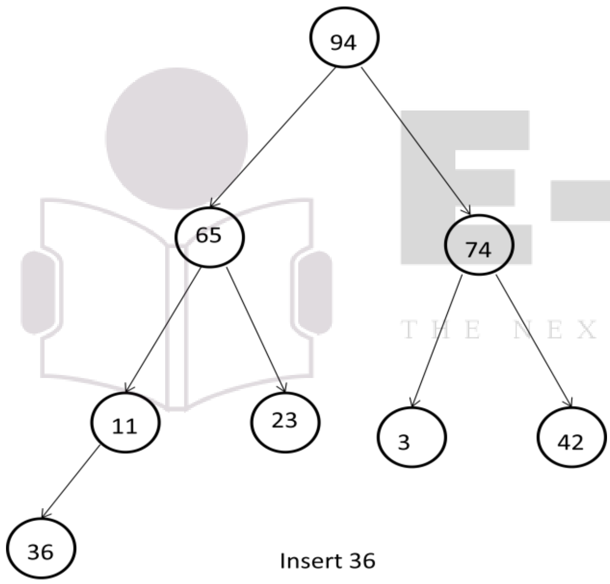
Insert 3



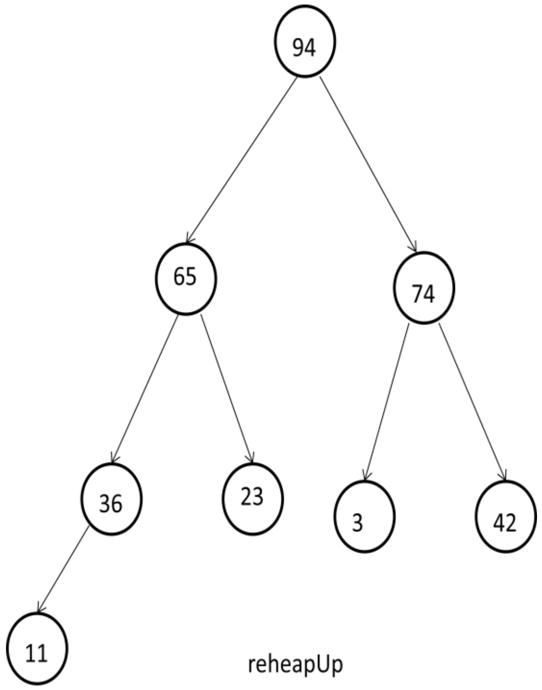
Insert 94



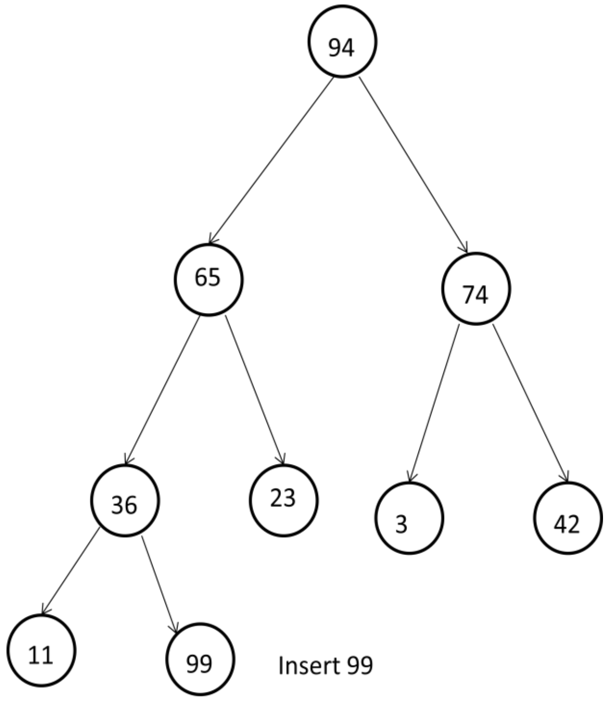
reheapUp



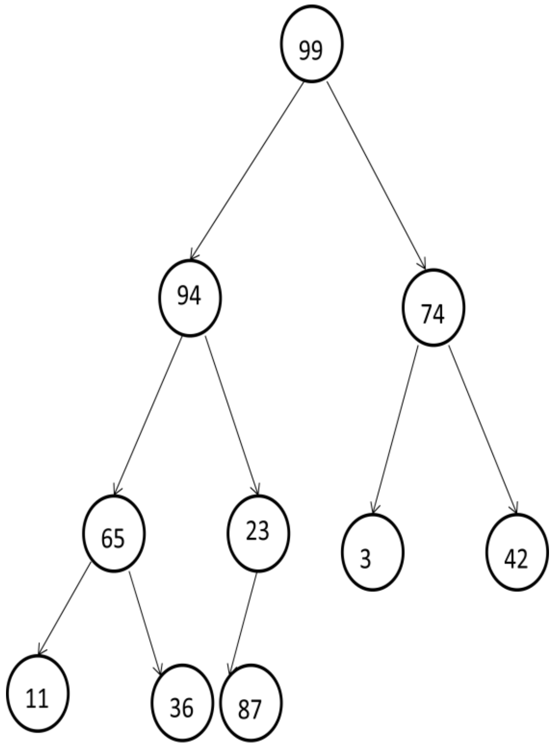
Insert 36



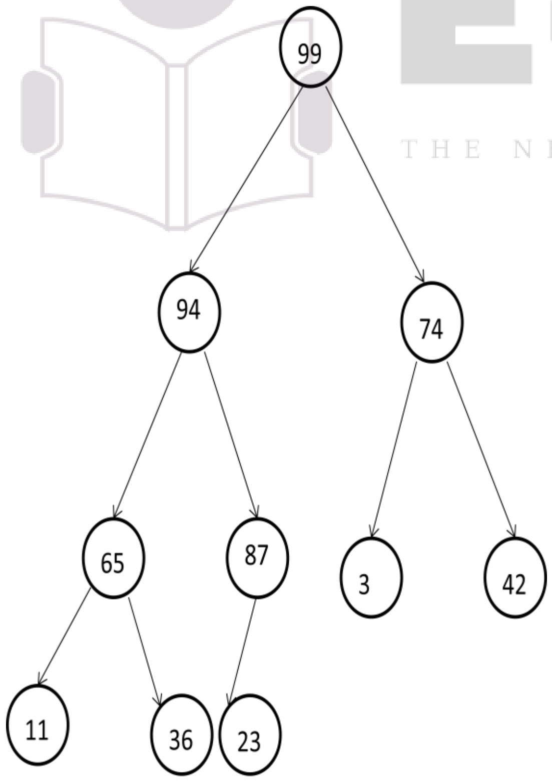
reheapUp



Insert 99



Insert 87



reheapUp

e) (year-2009(may):Q7(B)10 marks)

(i)What is a heap? Give the algorithm for reheapdown.

(ii)Make a heap out of the following data 23,7,92,6,12,14,40,44,20,21

Ans:

1)A heap is a binary tree structure with the following properties:-

a)The tree is complete or nearly complete.

b)The key value of each node is greater than or equal to the key value in each of its descendents.

2)A heap is generally a max heap though the properties can be reversed to create a min hep where the key value in a node is less than equal to the key values in all of its subtrees.

3)Unlike Binary Search Tree, the smaller nodes of a heap can be placed on either the right or the left subtree. Therefore, both the left and right branches of the tree have the same meaning.

4)Heaps are generally implemented as an array.

5)The relationship between a nodeand its children is fixed and can be calculated as:

(i)For a node located at index I, its children are found at :

a. leftchild=2i + 1

b. rightchild=2i + 2

(ii)For a node located at index I, its parent is located at [(i-1)/2]

(iii)Given the index for a left child j, its right sibling if any is found at j+1. Conversely, given the index for a right child k, its left sibling which must exist, is found at
k-1.

(iv)Given the size, n of a complete heap, the location of the first leaf is n/2. Given the location of the first leaf element, the location of the last non-leaf element is one less.

5)In short, a heap is a complete or nearly complete binary tree in which the key value in a node is greater than the key values in all of its subtrees and the subtrees are in turn heaps.

ALGORITHM FOR REHEAPDOWN

algorithm reheapDown(ref heap<array>,
val root <index>,
val last <index>)

Restablishes heap by moving data in root down to its correct location in the heap.

Pre: heap is an array of data
Root is root of heap or subheap
Last is an index to the last element in heap

Post: heap has been restored

Determine which child has larger key

1. if(root*2+1 <= last)

There is atleast one child

1. leftkey = heap[root*2+1].data.key
2. if(root*2+2 <= last)
   1. rightkey=heap[root*2+2].data.key
3. else
   1. rightkey=lowkey
4. if(leftkey>rightkey)

1. largechildkey=leftkey
2. largechildindex=root*2+1

5. else
   1. largechildkey=rightkey
   2. largechildindex=root*2+2

Test if root is greater than larger subtree

6. if(heap[root].data.key < heap[largechildindex].data.key)
   1. swap(root,largechildindex)
   2. reheapDown(heap,largechildindex,last)

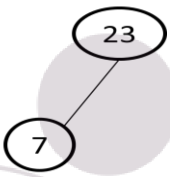
2. Return

End reheapDown

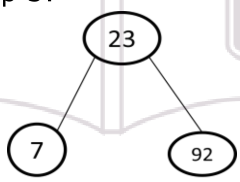
(ii)Given data: 23,7,92,6,12,14,40,44,20,21

Step 1:

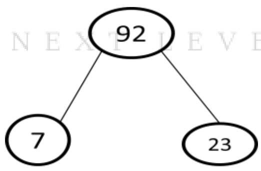
Step 2:



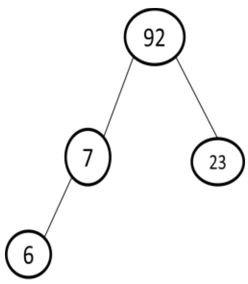
Step 3:



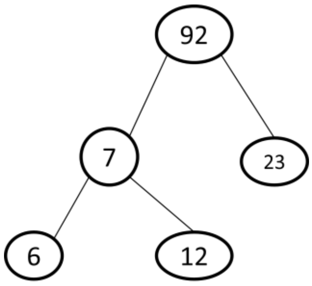
reheap ->



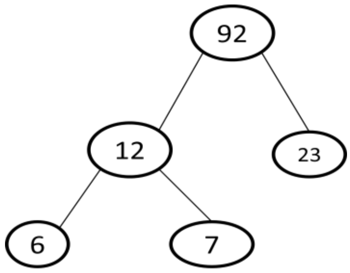
Step 4:



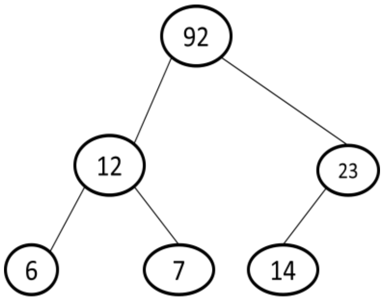
Step 5:



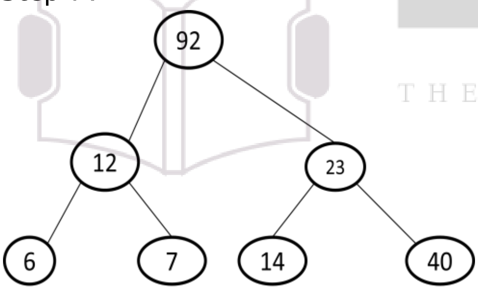
reheap->



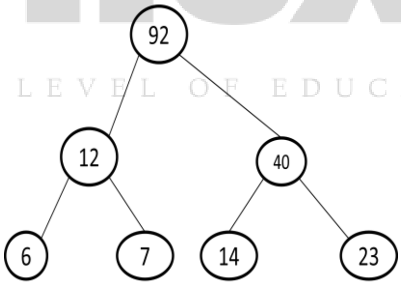
Step 6:



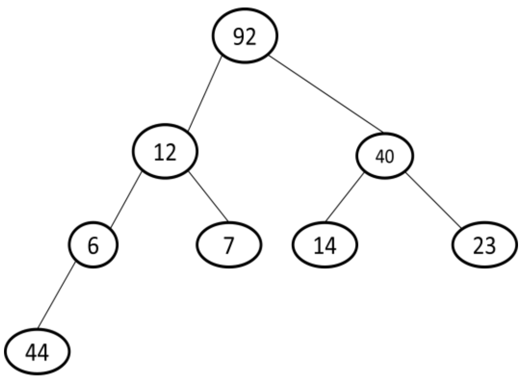
Step 7:



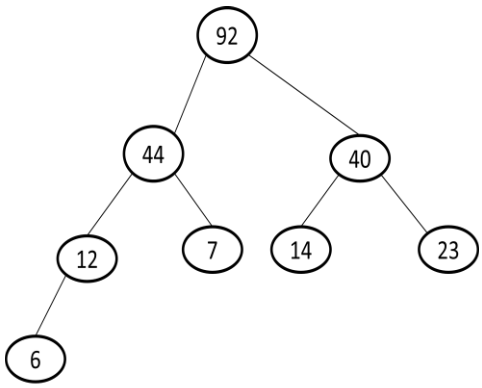
reheap ->



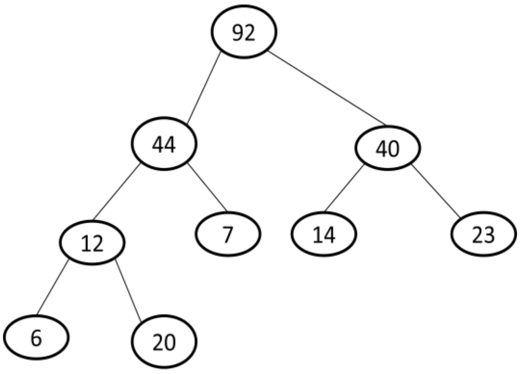
Step 8:



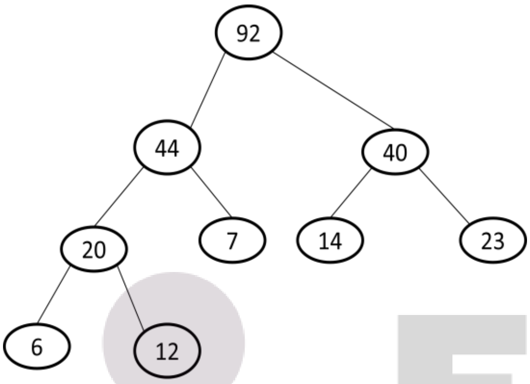
reheap ->



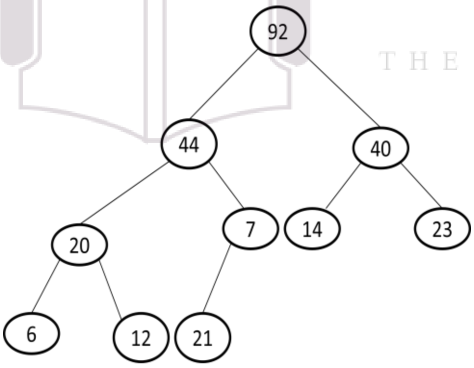
Step 9:



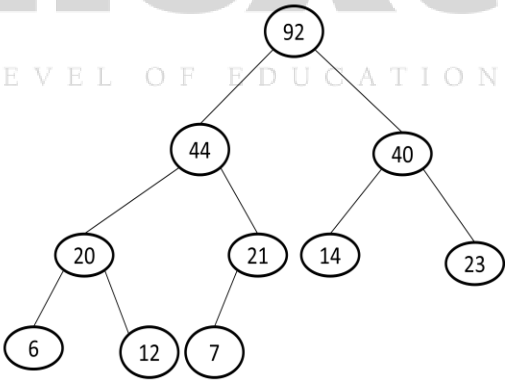
reheap->



Step 10:



reheap->



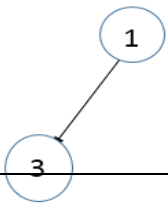
f) Year 2009 (dec): Q3(b)

Q. Define heap, construct max heap for the following data values arriving in the sequence:

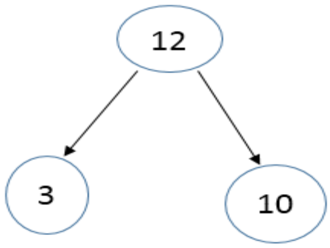
12,3,10,14,58,26,18,2,91,3

Ans:-

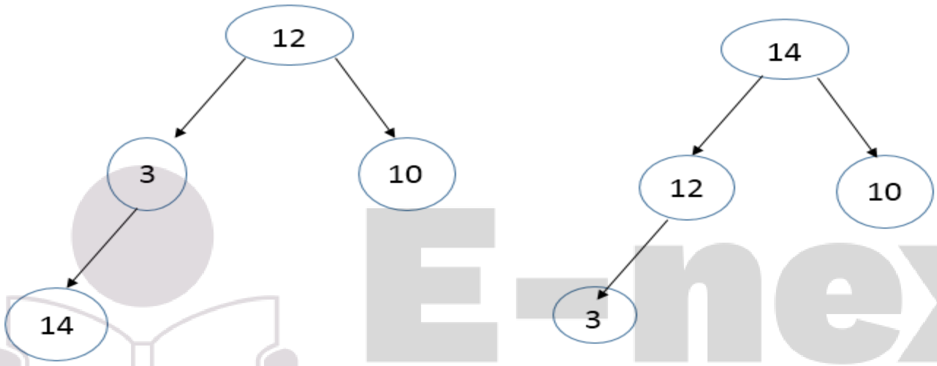
Step1:-



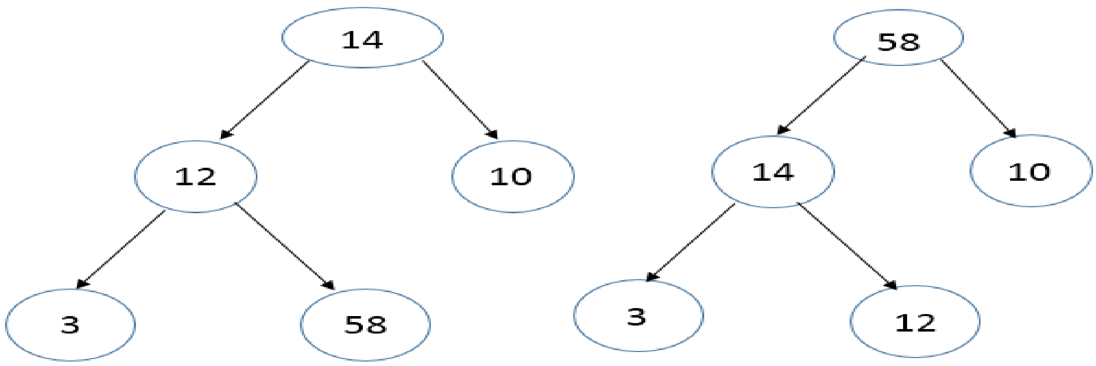
Step 2:-



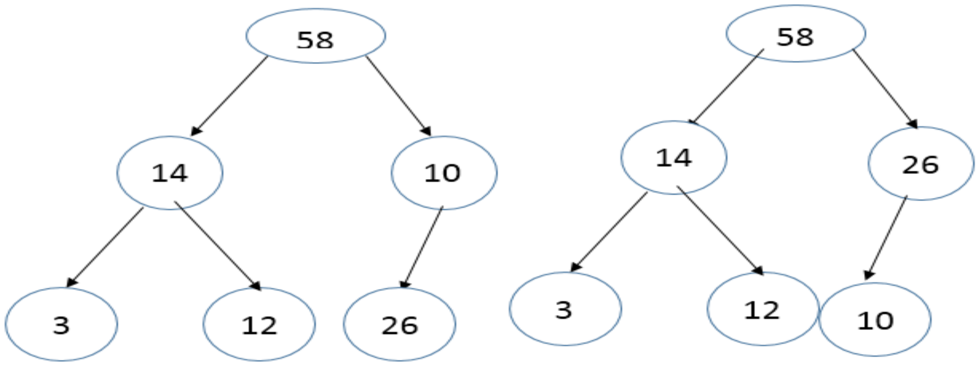
Step 3:-



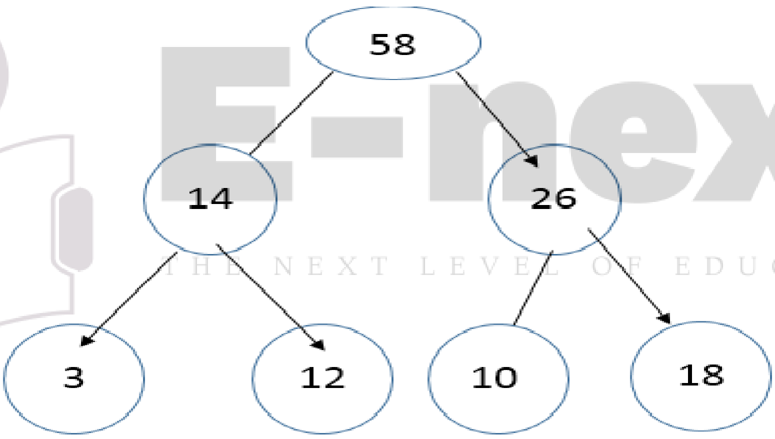
Step:-insert 58



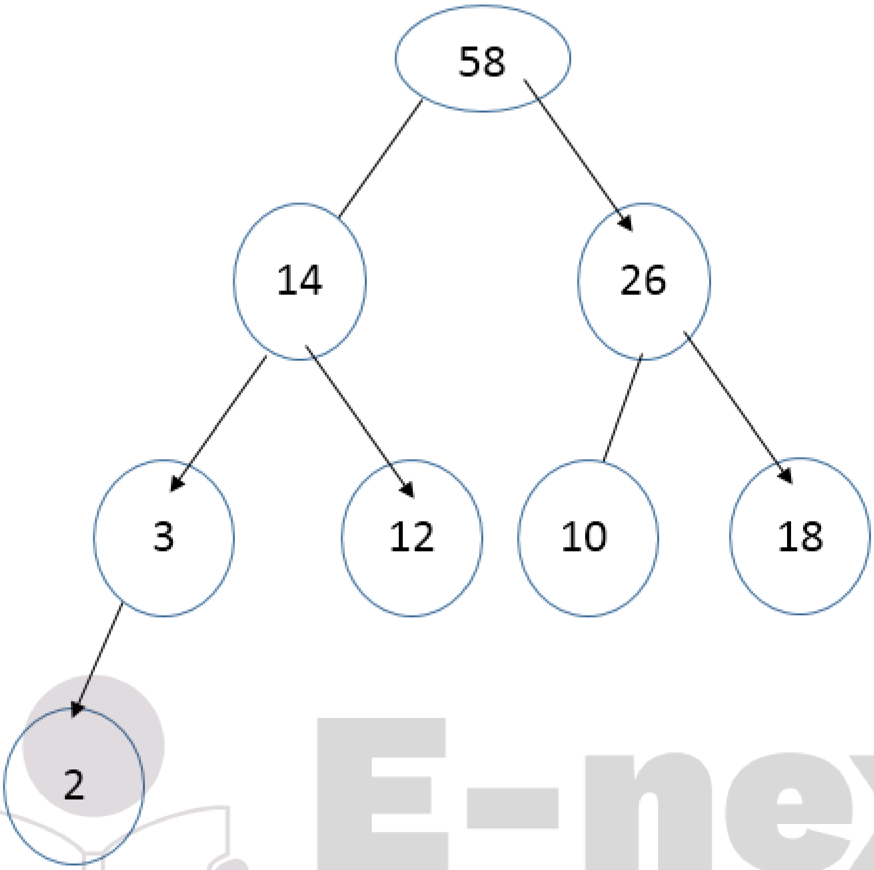
Step5:



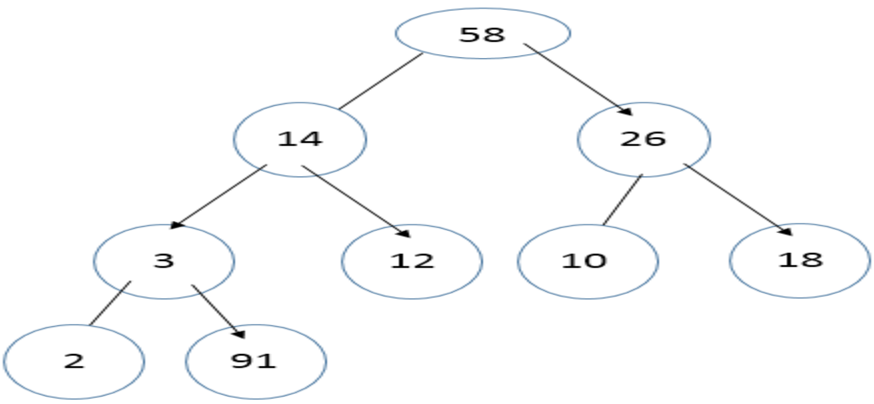
Step 6:-

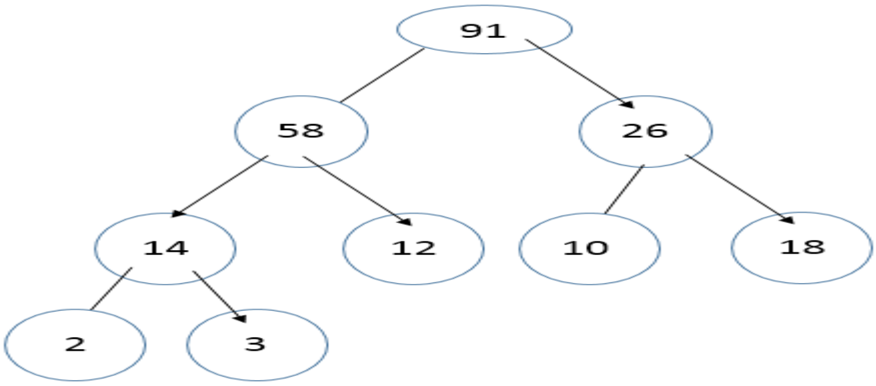


Step 7:-

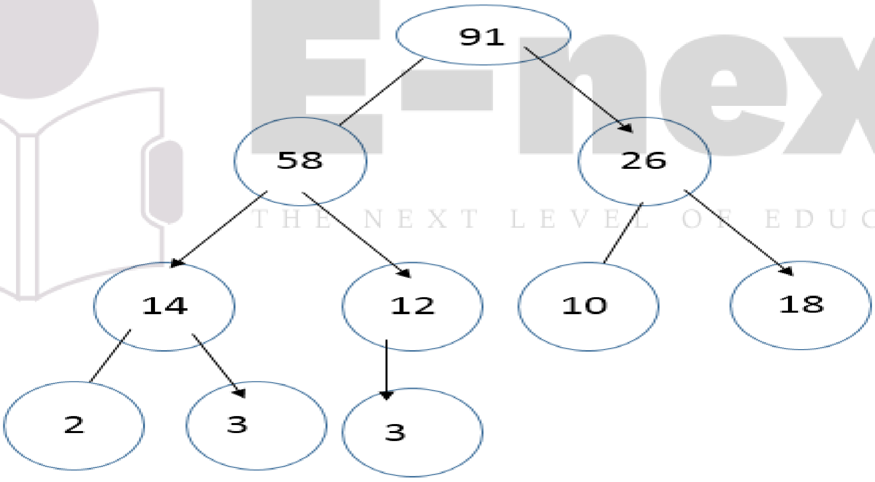


Step 9:-





Step:-10



g) year 2008 may : Q6(B)

Q. i) Define a heap. Give the algorithm for reheap up.

ii) Show the array implementation of the heap up int the following figure. Apply the delete operation to the heap on the followng figure and repair the heap. Insert 39 in the resultant heap and repair the same after insertion.

Ans: i)

1)A heap is a binary tree structure with the following properties:-

a)The tree is complete or nearly complete.

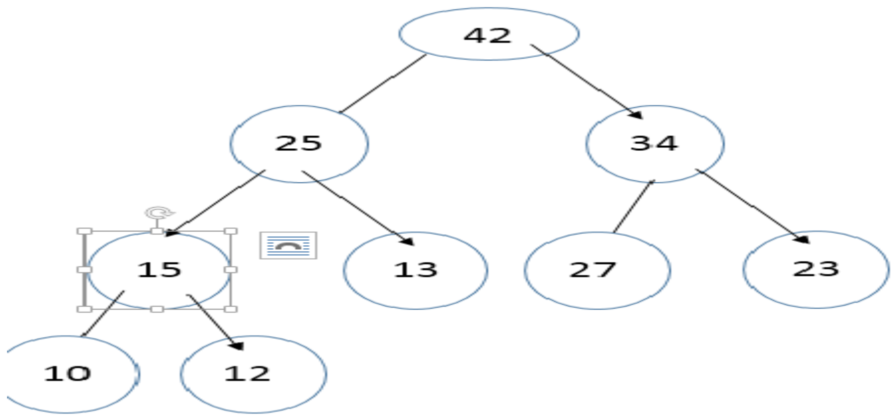
b)The key value of each node is greater than or equal to the key value in each of its descendents.

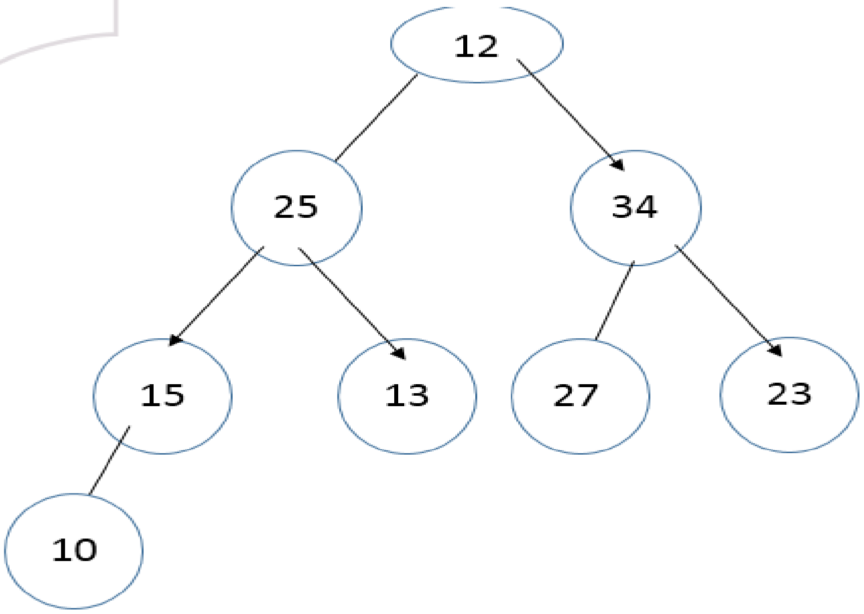
2)A heap is generally a max heap though the properties can be reversed to create a min hep where the key value in a node is less than equal to the key values in all of its subtrees.
3)Unlike Binary Search Tree, the smaller nodes of a heap can be placed on either the right or the left subtree. Therefore, both the left and right branches of the tree have the same meaning.
4)Heaps are generally implemented as an array.
5)The relationship between a nodeand its children is fixed and can be calculated as:
(i)For a node located at index I, its children are found at :
a. leftchild=2i + 1
b. rightchild=2i + 2
(ii)For a node located at index I, its parent is located at [(i-1)/2]
(iii)Given the index for a left child j, its right sibling if any is found at j+1.
Conversely, given the index for a right child k, its left sibling which must exist, is found at
k-1.
(iv)Given the size, n of a complete heap, the location of the first leaf is n/2. Given the location of the first leaf element, the location of the last non-leaf element is one less.
5)In short, a heap is a complete or nearly complete binary tree in which the key value in a node is greater than the key values in all of its subtrees and the subtrees are in turn heaps.

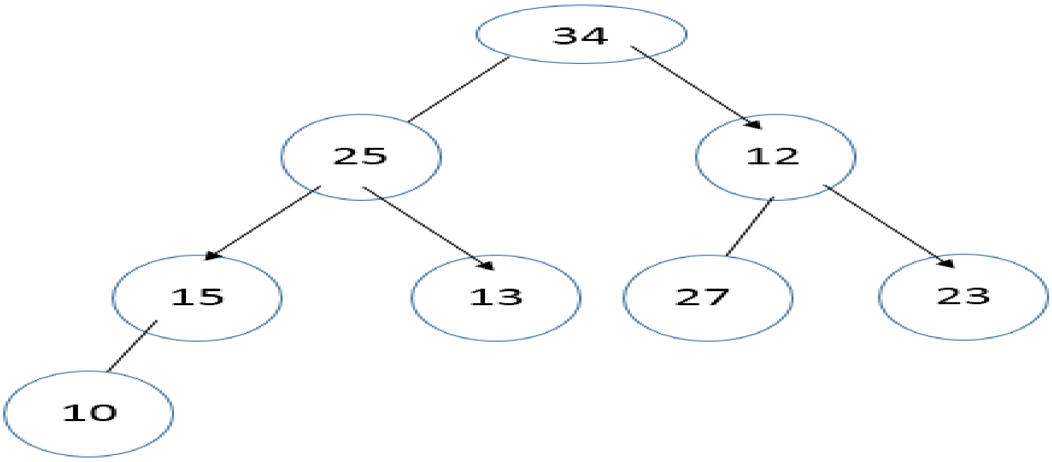
ALGORITHM FOR REHEAPDOWN

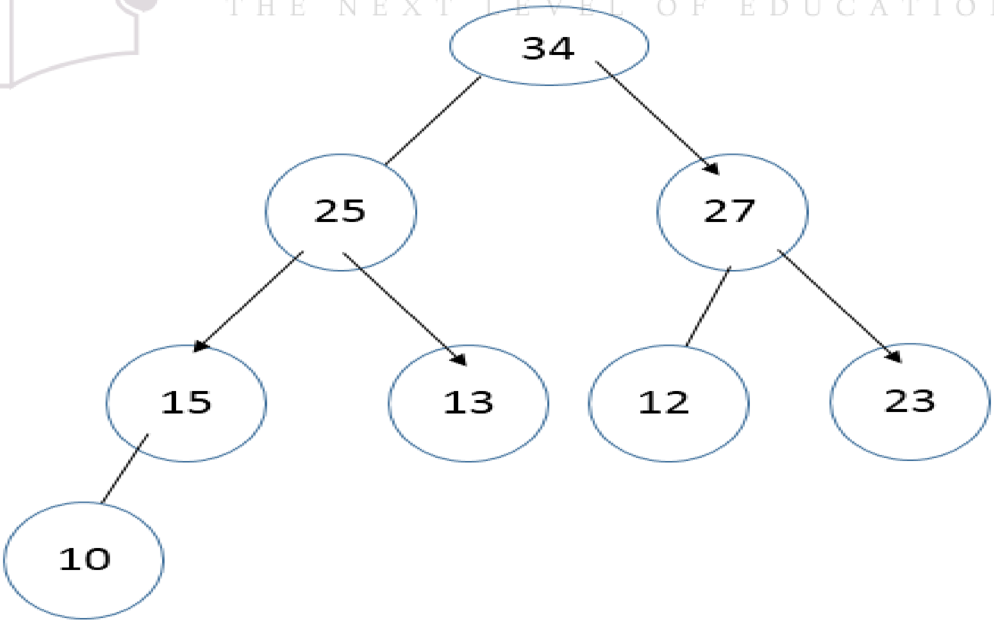
```
algorithm reheapDown(ref heap<array>,
 val root <index>,
 val last <index>)
 Restablishes heap by moving data in root down to its correct location in the heap.
Pre: heap is an array of data
 Root is root of heap or subheap
 Last is an index to the last element in heap
Post: heap has been restored
Determine which child has larger key
1. if(root*2+1 <= last)
There is atleast one child
1. leftkey = heap[root*2+1].data.key
2. if(root*2+2 <= last)
1. rightkey=heap[root*2+2].data.key
 3. else
 1. rightkey=lowkey
 4. if(leftkey>rightkey)
 1. largechildkey=leftkey
 2. largechildindex=root*2+1 5. else
 1. largechildkey=rightkey
 2. largechildindex=root*2+2
 Test if root is greater than larger subtree
 6. if(heap[root].data.key < heap[largechildindex].data.key)
 1. swap(root,largechildindex)
 2. reheapDown(heap,largechildindex,last)
2. Return
End reheapDown
```

Ans ii) :-
delete



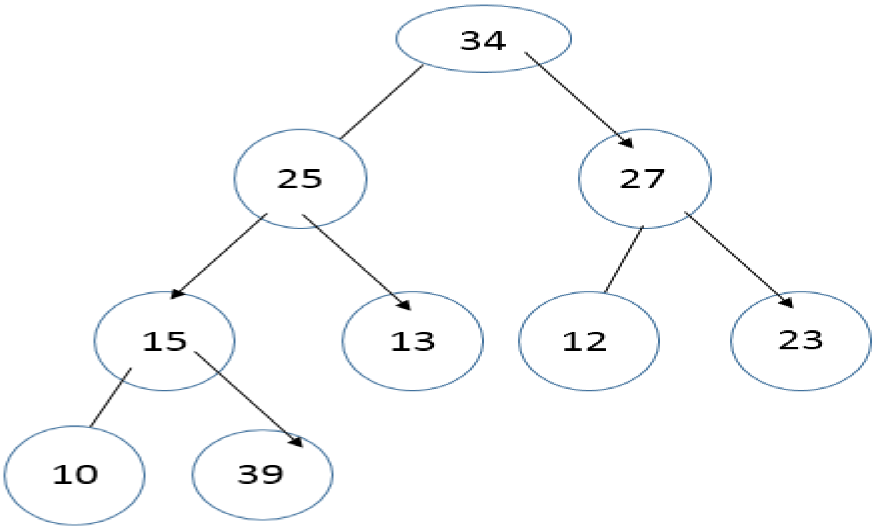


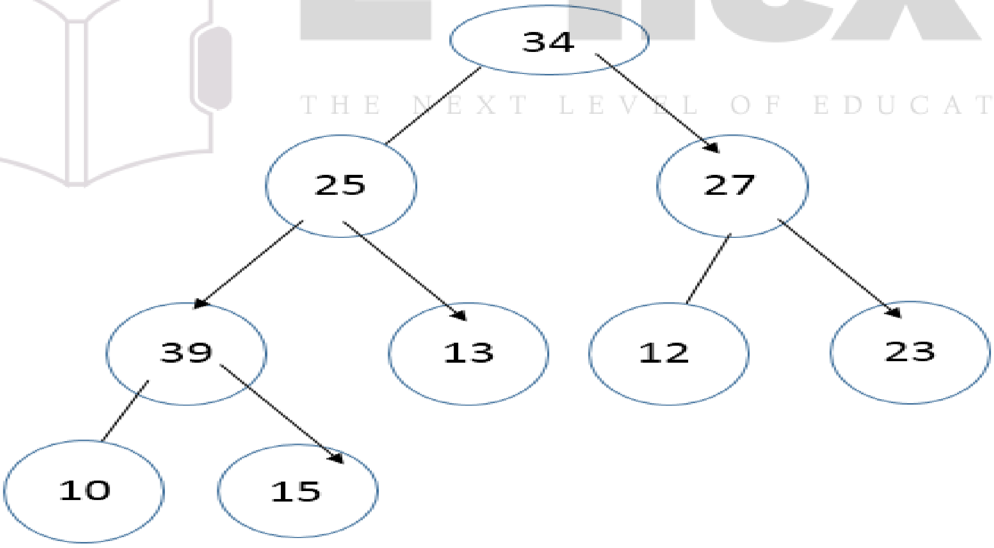


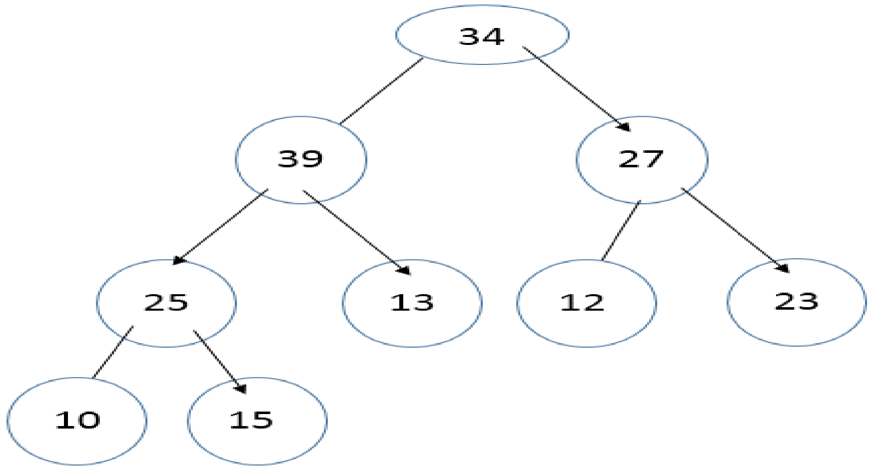


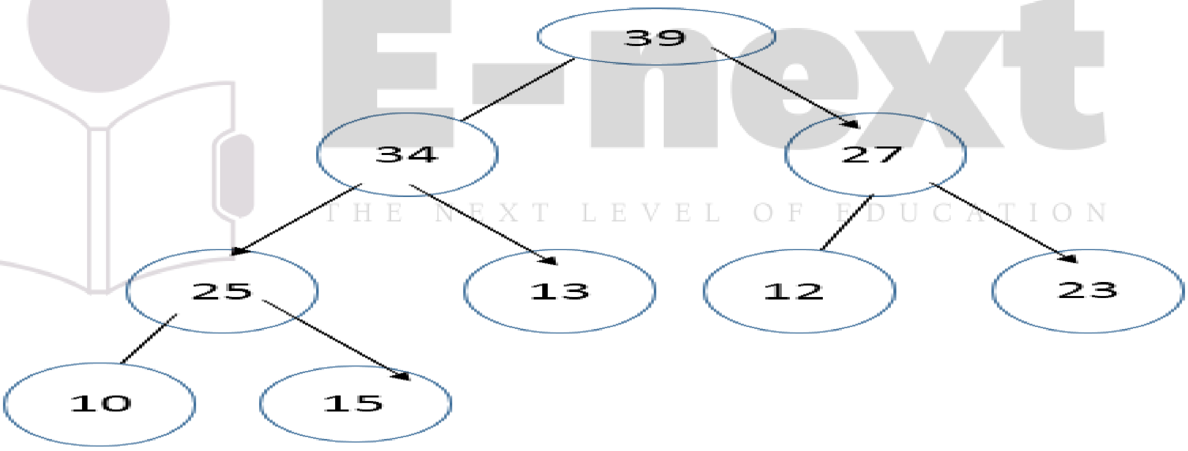
Ans:b)

Insert 39






34
39
27
25
13
12
23
10
15


39
34
27
25
13
12
23
10
15

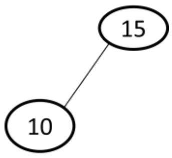
h) (year-2007(may):Q5(A)10 marks)

Given a set of numbers build a heap and sort the array using heap sort method.
15, 10, 12, 8, 6, 7, 35, 16, 5

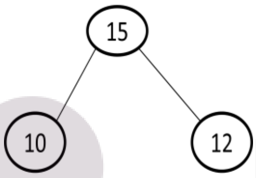
Ans:

Step 1:

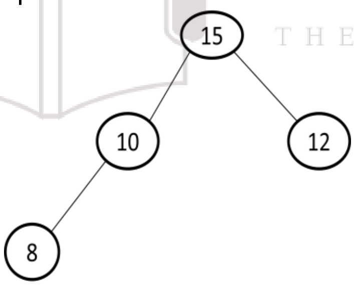
Step 2:



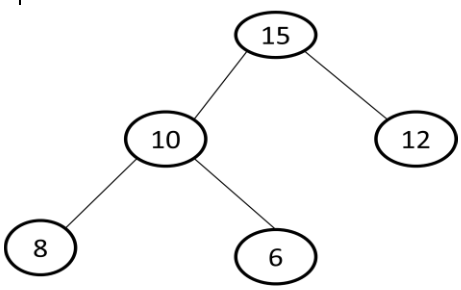
Step 3:



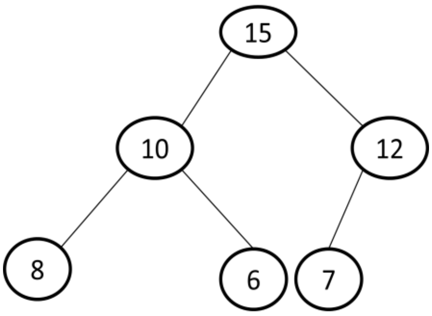
Step 4:



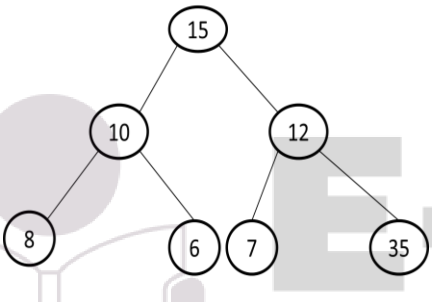
Step 5:



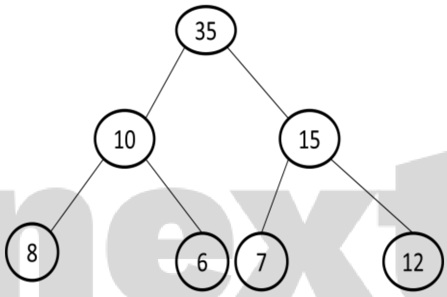
Step 6:



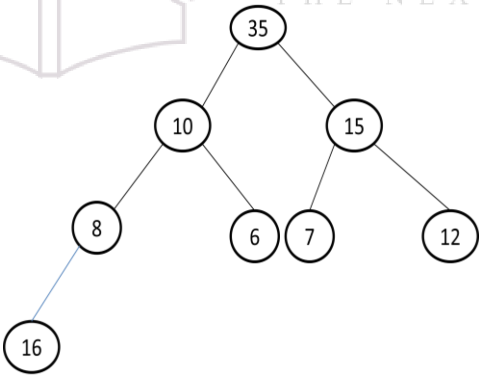
Step 7:



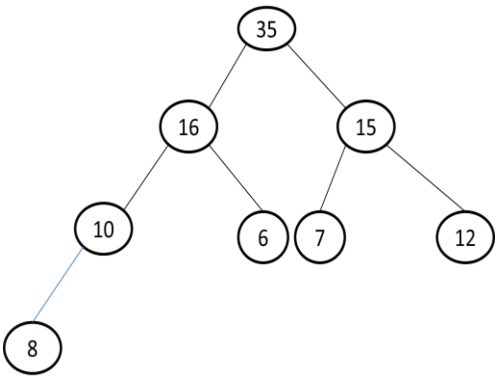
reheap->



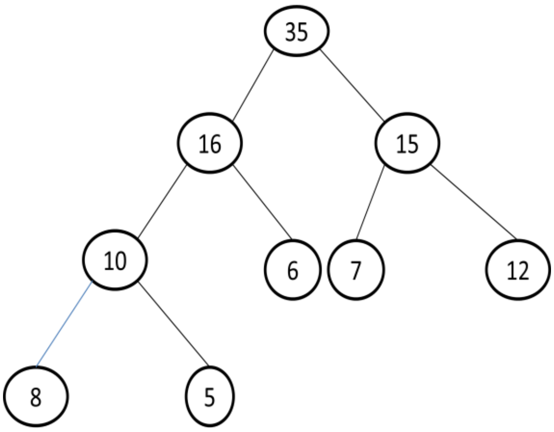
Step 8:



reheap->



Step 9:



Sorting the given heap:
After Heap : 35 16 15 10 6 7 12 8 5

After pass1
and reheap : 16 10 15 8 6 7 12 5 35

After pass2
and reheap : 15 10 12 8 6 7 5 16 35

After pass3
and reheap : 12 10 7 8 6 5 15 16 35

After pass4
and reheap : 10 8 7 5 6 12 15 16 35

After pass5
and reheap : 8 6 7 5 10 12 15 16 35

After pass6
and reheap : 7 6 5 8 10 12 15 16 35

After pass7
and reheap : 6 5 7 8 10 12 15 16 35

After pass8
and reheap : 5 6 7 8 10 12 15 16 35