UNIT-III : PRIORITY QUEUES

Model, Simple Implementation, Binary Heap-Structure Property-Heap-Order Property-Basic Heap Operations- Other Heap Operation, Applications of Priority Queues- The Selection Problem Event Simulation Problem, Binomial Queues- Binomial Queue Structure – Binomial Queue Operation- Implementation of Binomial Queues

---

A priority queue is a collection of elements such that each element has an associated priority. Priority Queue is a data structure like stack, queue where insertions, deletions and retrieval is done based on Priority.

A priority can be either Maximum or Minimum.
Priority queues are implemented using 1) Heap
2) Binomial Queue

A heap is a hierarchical data structure in which operations are done based on maximum or minimum priority.

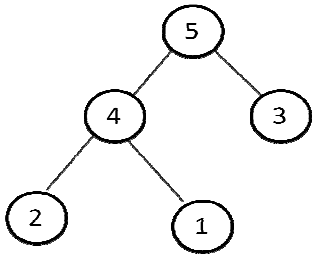
A binary heap is defined as a binary tree with two additional constraints:

- **Shape property**: a binary heap is a *complete binary tree*; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.

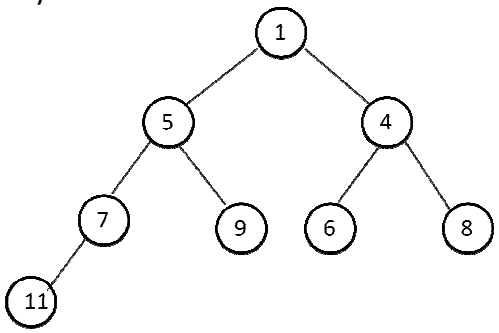
- **Heap property**: the key stored in each node is either greater than or equal to (≥) or less than or equal to (≤) the keys in the node's children, according to some total order.

Heaps where the parent key is greater than or equal to (≥) the child keys are called *max-heaps*; those where it is less than or equal to (≤) are called *min-heaps*.

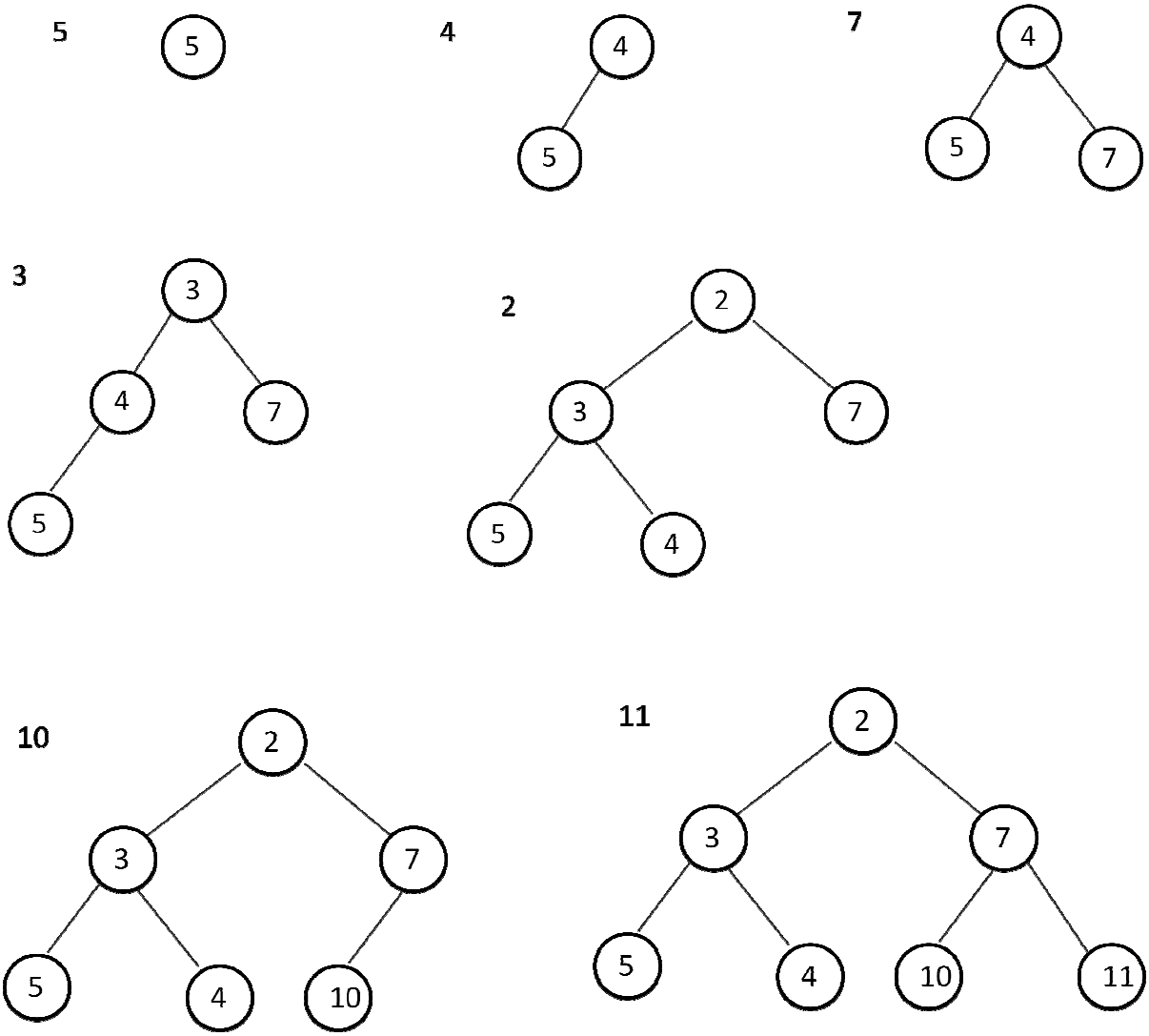
A max heap is a complete binary tree in which a root of every sub tree has larger element to the data compared to its children.



A min heap is a complete binary tree in which a root of sub tree is smaller than its children.

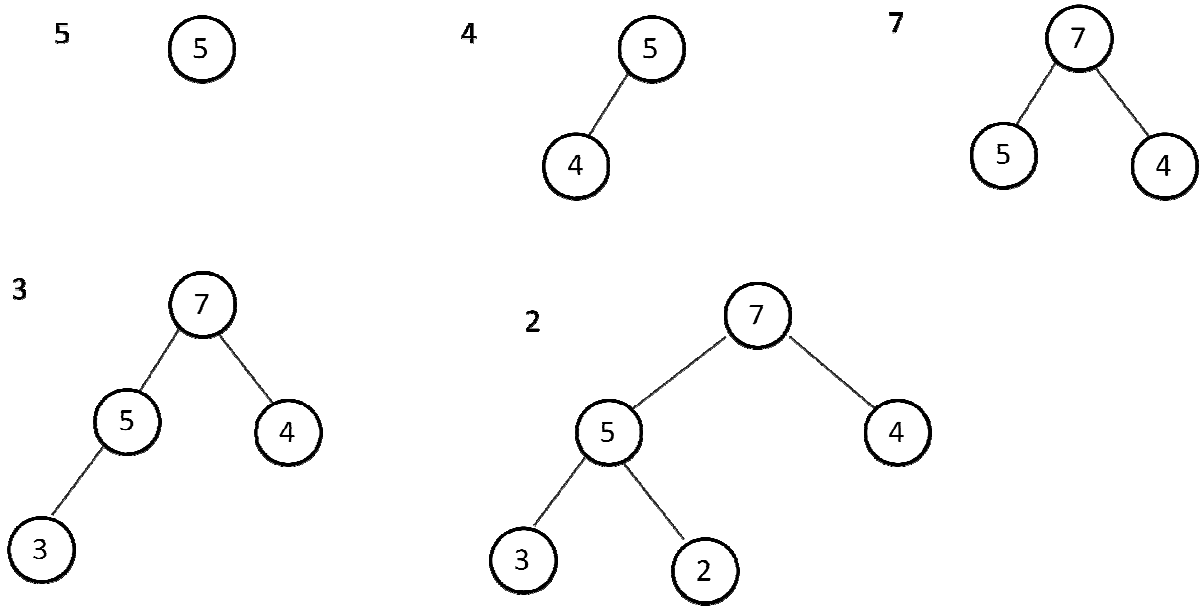


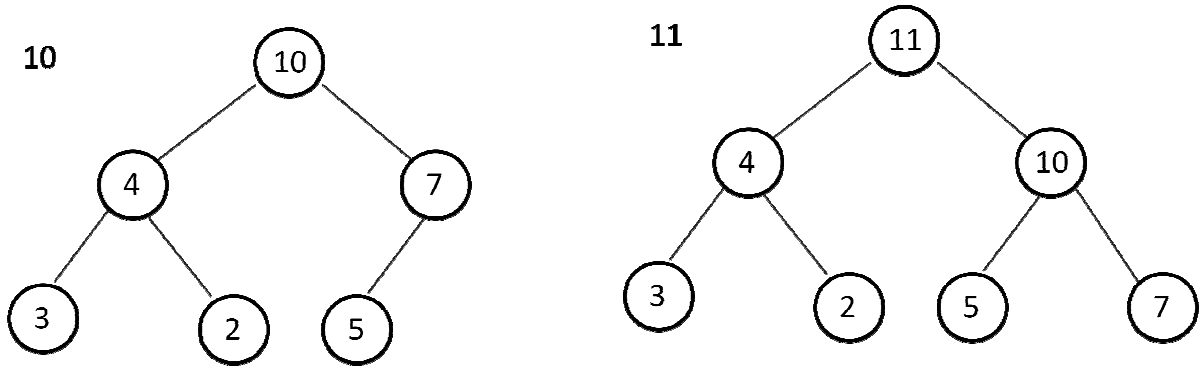
Construction of a min heap with 5, 4, 7, 3, 2, 10, 11



Array	2	3	7	5	4	10	11
index	0	1	2	3	4	5	6

Construction of a max heap with 5, 4, 7, 3, 2, 10, 11





Array	10	4	7	3	2	5
index	0	1	2	3	4	5

Array	11	4	10	3	2	5	7
index	0	1	2	3	4	5	6

A heap is represented by an array. When a node is at index 'k' where k>=0 then its left child will be at index 2k+1, right child is at 2k+2.

When a child is at index j, its parent is at index (j-1)/2.

If K is parent k>=0

Left child is at 2k+1

Right child is at 2k+2

Operations on Heap

1)Insert
2)Delete
3)Peek (Max or Min)

Insert

Let an element ‘e’ be inserted in a heap ‘T’ ----------max heap

Insertion Algorithm

Now, let us construct general algorithm to insert a new element into a max heap. 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.

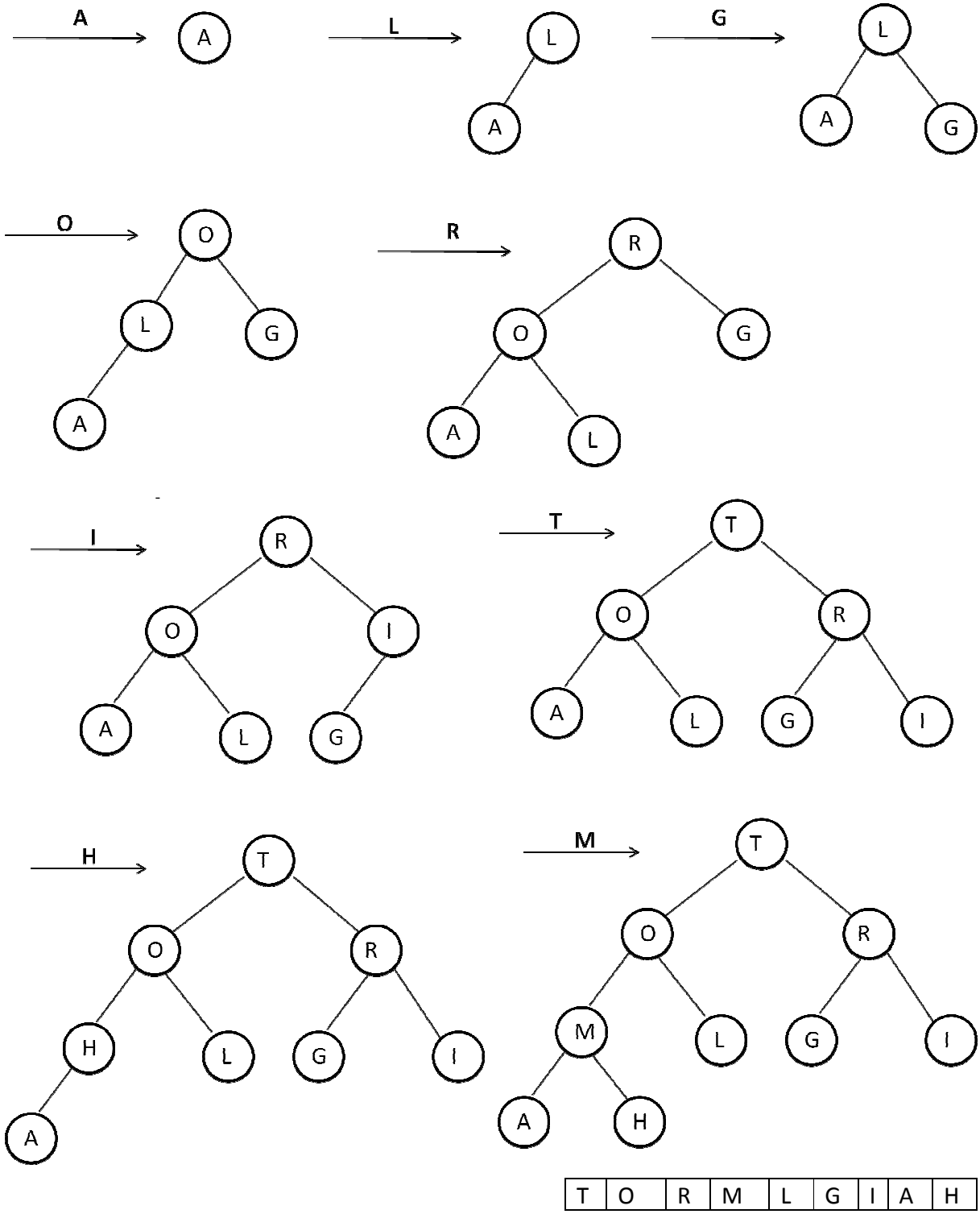
Step 1 - Create a new node at the end of heap.
Step 2 - Assign new value to the node.
Step 3 - Compare the value of this child node with its parent.
Step 4 - If value of parent is less than child, then swap them.
Step 5 - Repeat step 3 & 4 until Heap property holds.

```
insert(int num, int location)
{
   int parentnode;
   while (location > 0)
   {
         parentnode =(location - 1)/2;
         if (num <= array[parentnode])
         {
            array[location] = num;
            return;
         }
         array[location] = array[parentnode];
         location = parentnode;
   }/*End of while*/
   array[0] = num; /*assign number to the root node */
}/*End of insert()*/
```

Complexity analysis

Complexity of the insertion operation is O(h), where **h** is heap's height. Taking into account completeness of the tree, O(h) = O(log n), where **n** is number of elements in a heap.

Construct max heap for ‘A L G O R I T H M’



T	O	R	M	L	G	I	A	H

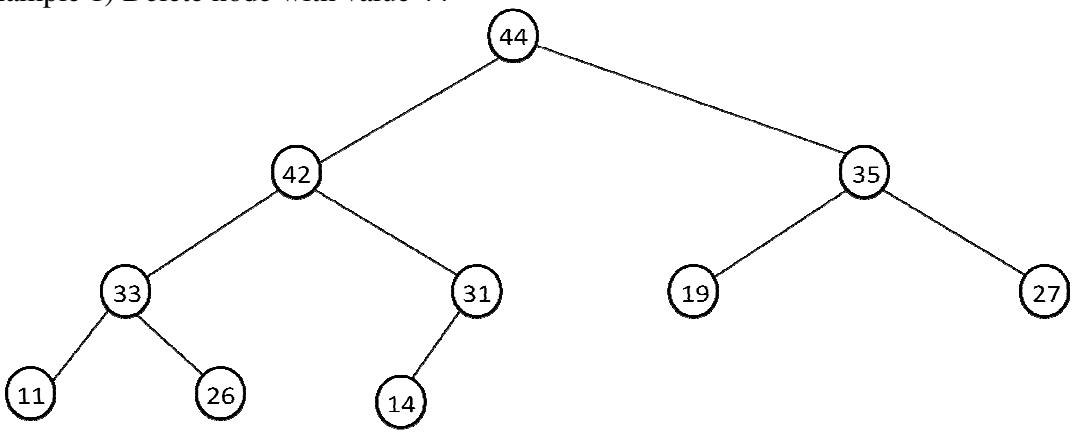
Deletion from Heap / Process of Deletion:

Since deleting an element at any intermediary position in the heap can be costly, so we can simply replace the element to be deleted by the last element and delete the last element of the Heap.

Algorithm

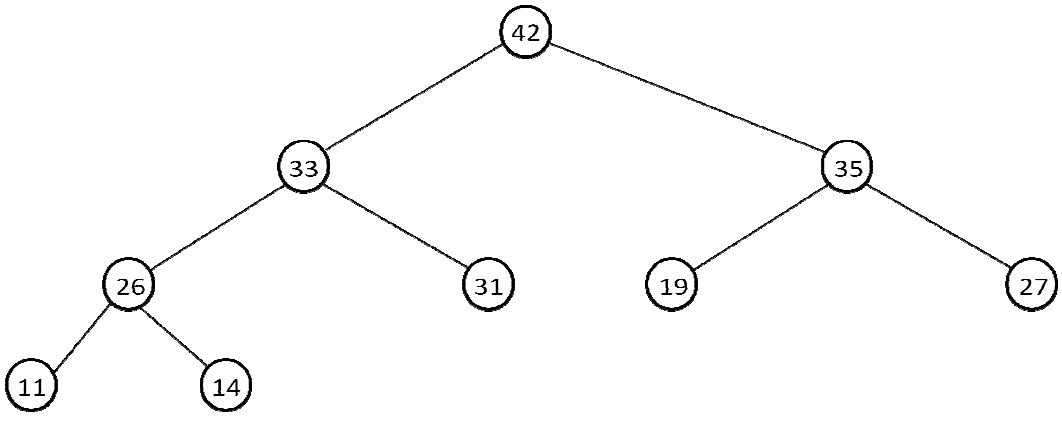
Step 1) Replace the root or element to be deleted by the last element.
Step 2) Delete the last element from the Heap.
Step 3) Since, the last element is now placed at the position of the root node. So, it may not follow the heap property. Therefore, **heapify** the last node placed at the position of root.

Example 1) Delete node with value 44



Array	44	42	35	33	31	19	27	11	26	14
Index	0	1	2	3	4	5	6	7	8	9

Heap after deleting node 44

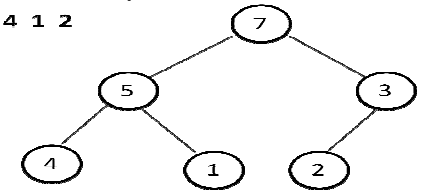


Array	42	33	35	26	31	19	27	11	14
Index	0	1	2	3	4	5	6	7	8

Time complexity of deleting a node from heap = O(log n)

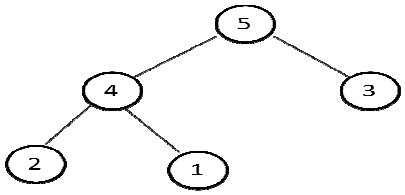
Example 2) Build heap with 5 7 3 4 1 2 and then delete one node

Insert 5 7 3 4 1 2



Array	7	5	3	4	1	2
index	0	1	2	3	4	5

delete



Array	5	4	3	2	1
index	0	1	2	3	4

```
/*************************************************************
 * C Program to Implement a Heap & provide Insertion & Deletion Operation
 *************************************************************/
#include <stdio.h>
#include <conio.h>
int array[100], n;
        display()
        {
          int i;
          if (n == 0)
          {
                printf("Heap is empty \n");
                return;
          }
          for (i = 0; i < n; i++)
                printf("%d ", array[i]);
          printf("\n");
        }/*End of display()*/

        insert(int num, int location)
        {
          int parentnode;
          while (location > 0)
          {
                parentnode =(location - 1)/2;
                if (num <= array[parentnode])
                {
                  array[location] = num;
                  return;
                }
                array[location] = array[parentnode];
                location = parentnode;
          }/*-----------------------------------------------------------------End of while*/
          array[0] = num; /*-------------------------assign number to the root node */
        }/*-----------------------------------------------------------------End of insert()*/
```

```
deletenode()
{
   int left, right, i, temp, parentnode;

   i=0;
   array[i] = array[n - 1];

   n = n - 1;
   parentnode =(i - 1) / 2; /*find parentnode of node i */
   if (array[i] > array[parentnode])
   {
         insert(array[i], i);
         return;
   }
   left = 2 * i + 1;          /* ---------------left child of i   */
   right = 2 * i + 2;         /* ---------------right child of i  */

  while (right < n)
   {
         if (array[i] >= array[left] && array[i] >= array[right])
            return;
         if (array[right] <= array[left])
         {
            temp = array[i];
            array[i] = array[left];
            array[left] = temp;
            i = left;
         }
         else
         {
            temp = array[i];
            array[i] = array[right];
            array[right] = temp;
            i = right;
         }
         left   = 2 * i + 1;
         right = 2 * i + 2;
   }/*End of while*/
   if (left == n - 1 && array[i])
   {
         temp = array[i];
         array[i] = array[left];
         array[left] = temp;
   }
}
```

```
void main()
{
   int choice, num;
   n = 0;                    /*--------------Represents number of nodes in the heap*/
   clrscr();
   while(1)
   {
         printf("1.Insert the element \n");
         printf("2.Delete the element \n");
         printf("3.Display all elements \n");
         printf("4.Quit \n");
         printf("Enter your choice : ");
         scanf("%d", &choice);
         switch(choice)
         {
             case 1:    printf("Enter the element to be inserted to the list : ");
                        scanf("%d", &num);
                        insert(num, n);
                        n = n + 1;
                        break;
             case 2:   deletenode();
                        break;
             case 3:   display();
                        break;
             case 4:   exit(0);
             default:   printf("Invalid choice \n");
          }/*-----------------------------------------------------End  of switch */
}/*-------------------------------------------------------------------End of while */
}/*-------------------------------------------------------------------End of main()*/
```

Binary Heap Time Complexity in Big O notation

Algorithm	Average case	Worst case
Insert	**O(1)**	**O(log n)**
Delete	**O(logn)**	**O(log n)**
peek	**O(1)**	**O(1)**
Search	**O(n)**	**O(n)**
Buildheap	**O(nlog(n))**	**O(n log(n))**

Building a heap from an array of n input elements can be done by starting with an empty heap, then successively inserting each element. This approach, called Williams' method after the inventor of binary heaps, is easily seen to run in $O(n \log n)$ time: it performs n insertions at $O(\log n)$ cost each.

Heap Sort : One of the Applications of Heap

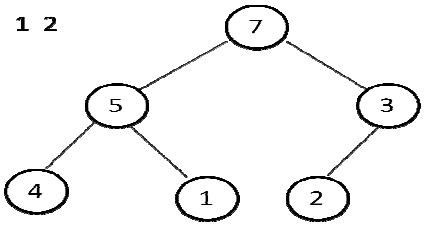
To sort the given n elements using HEAP SORT using Max HEAP the algorithm is as follows

Algorithm

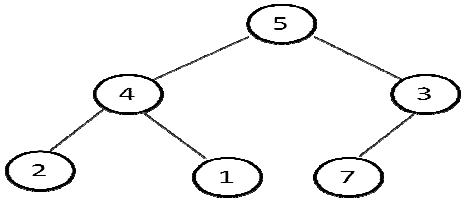
Step 1) Build max heap
Step 2) Swap last element with root
Step 3) Reduce size by 1 and reheapify the remaining elements
Step 4) Repeat step 2 and 3 until size of heap is 1 which is all elements are sorted

For example to sort the elements 5 7 3 4 1 2 first build a max heap

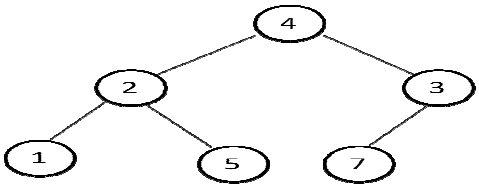
Insert 5 7 3 4 1 2



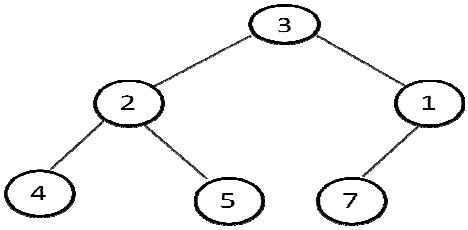
Array	7	5	3	4	1	2
index	0	1	2	3	4	5



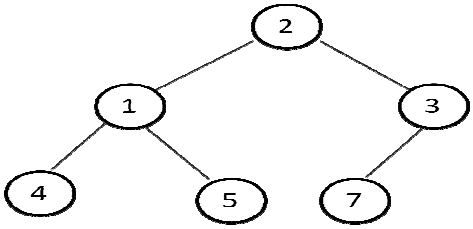
Array	5	4	3	2	1	7
index	0	1	2	3	4	5



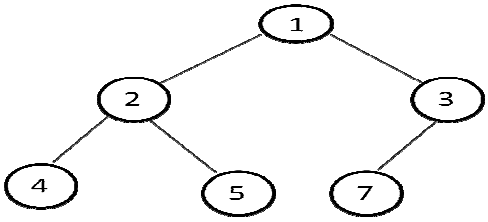
Array	4	2	3	1	5	7
index	0	1	2	3	4	5



Array	3	2	1	4	5	7
index	0	1	2	3	4	5



Array	2	1	3	4	5	7
index	0	1	2	3	4	5



Array	1	2	3	4	5	7
index	0	1	2	3	4	5

Heap Applications:

1) Heap Sort
2) Prim's Algorithm
3) Dijkstra's algorithm
4) Order Statistics

Operations on Heap
Insert ---- O(log n)
Peek ---- O(1)
Delete Min ----- O log n)
Delete Max ---- O(log n)

Heap sort is a sorting technique in which elements are sorted based on heap. It can be either internal sorting or external sorting.

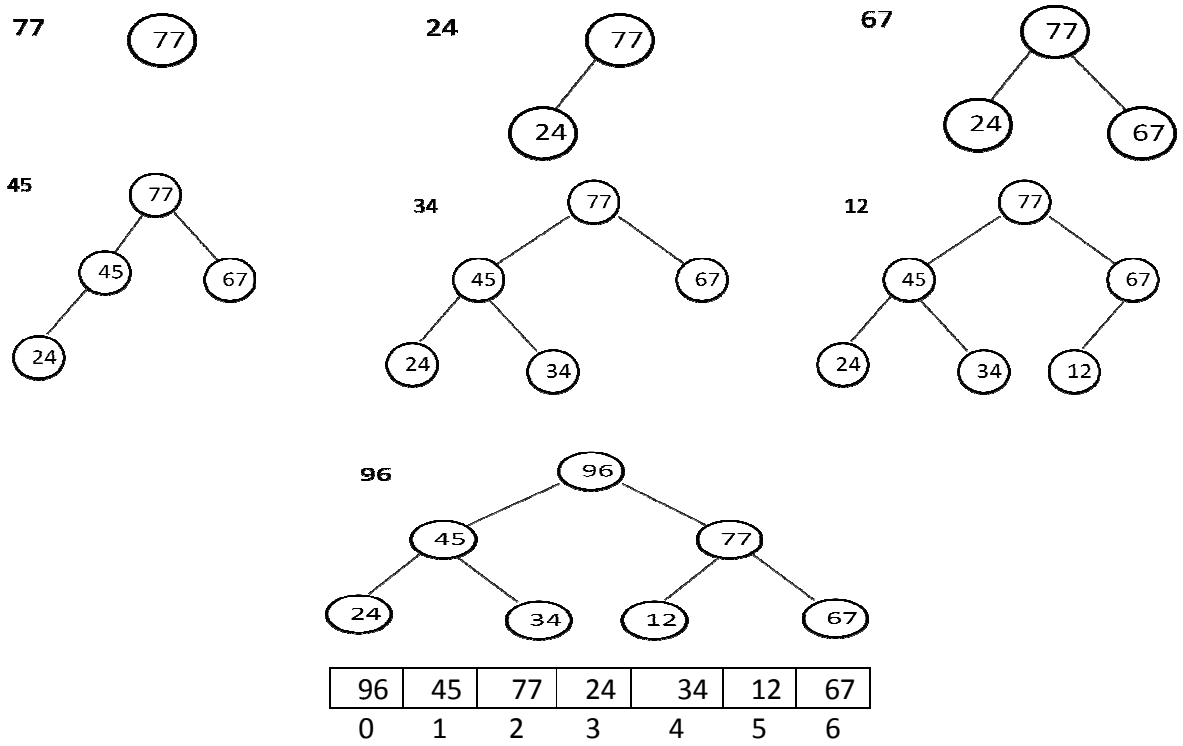
For ascending order in internal sorting a max heap is used.
For descending order in internal sorting a min heap is used.
For ascending order in External sorting a min heap is used.
For descending order in External sorting a max heap is used.

HEAP SORT :: Internal Sorting using Max Heap

Algorithm

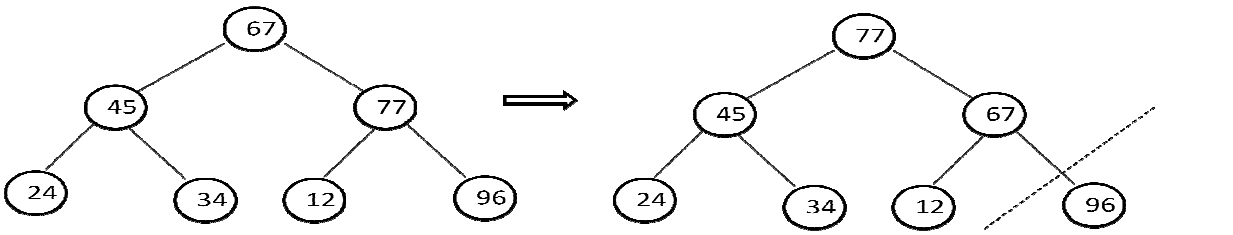
Step 1) Build a Max heap
Step 2) Swap Last element with root. (Largest element)
Step 3) size --. (reduce size of heap). Reheapify the remaining elements.
Step 4) Repeat steps 2 and 3 until heap size is 1 which is all elements are sorted.

Ex: 77 24 67 45 34 12 96



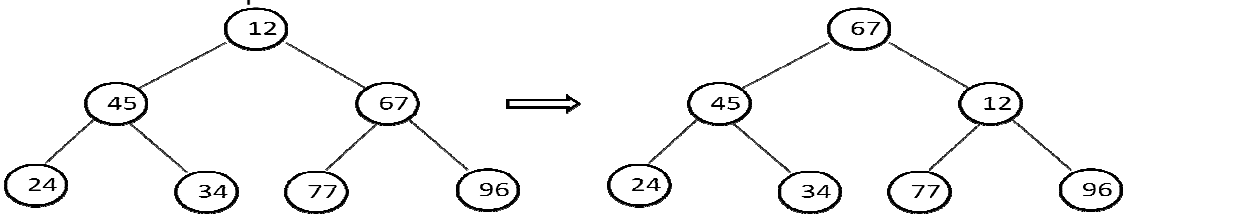
96	45	77	24	34	12	67
0	1	2	3	4	5	6

The heap is represented as an array. The root node value is 96, which is the largest element in the array. So we interchange the largest element with the last element of the array. The remaining elements excluding the last position will be reheapifyed and the process is repeated.



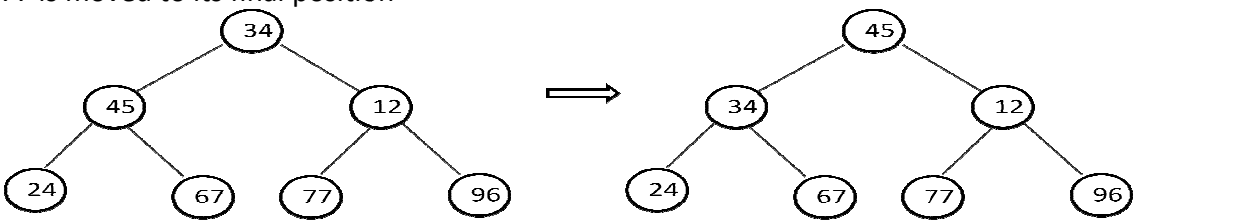
77	45	67	24	34	12	96
0	1	2	3	4	5	6

96 is moved to last position.



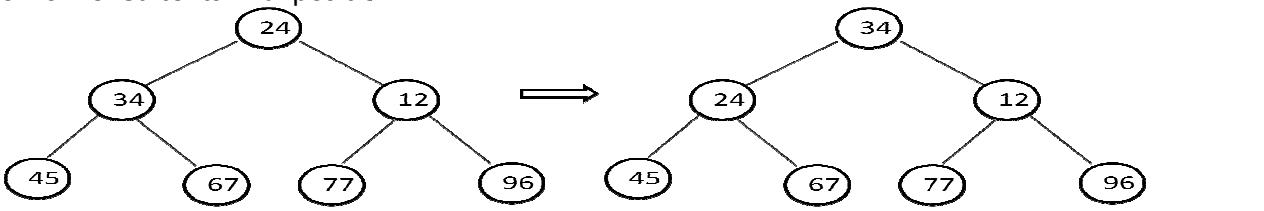
67	45	12	24	34	77	96
0	1	2	3	4	5	6

77 is moved to its final position

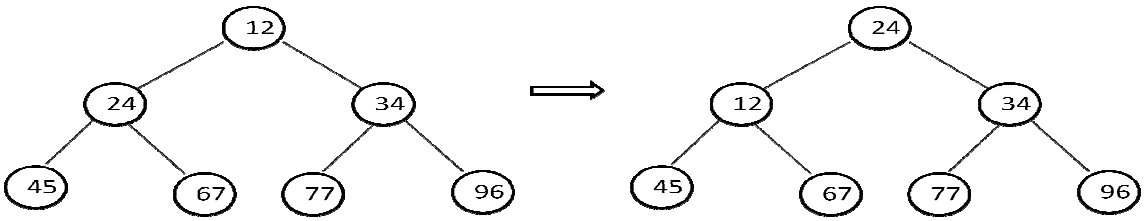


45	34	12	24	67	77	96
0	1	2	3	4	5	6

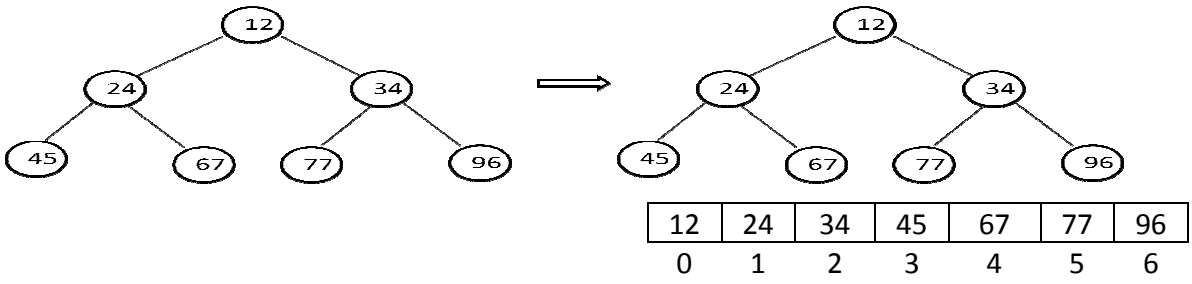
67 is moved to its final position



34	24	12	45	67	77	96
0	1	2	3	4	5	6



24	12	34	45	67	77	96
0	1	2	3	4	5	6



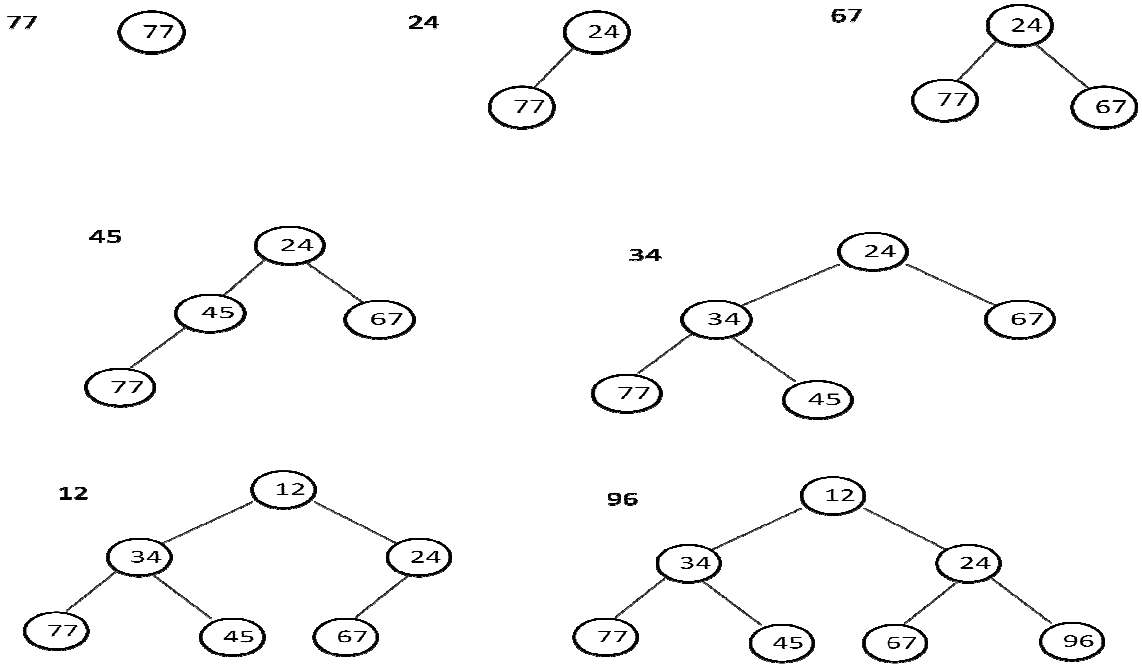
Sorted array in ascending order

HEAP SORT : External Sorting using Min Heap

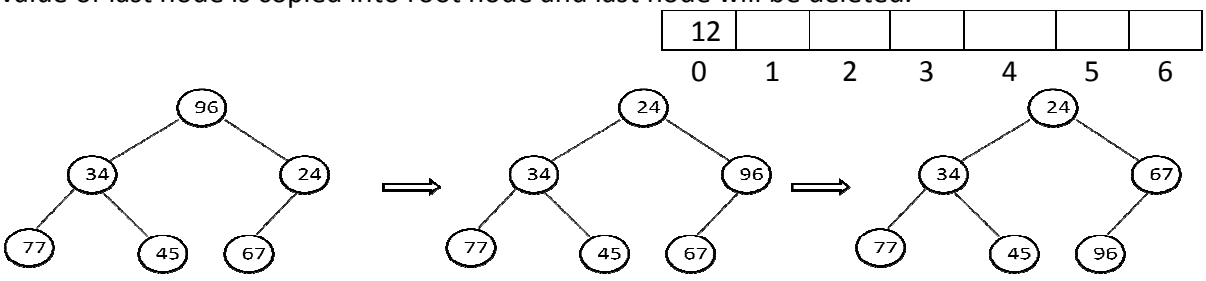
Algorithm

Step 1) Build a min heap
Step 2) Delete min element (root) in heap and store it in an external data structure.
Step 3) Reheapify
Step 4) Repeat steps 2 and 3 until heap is empty.

Example: 77 24 67 45 34 12 96

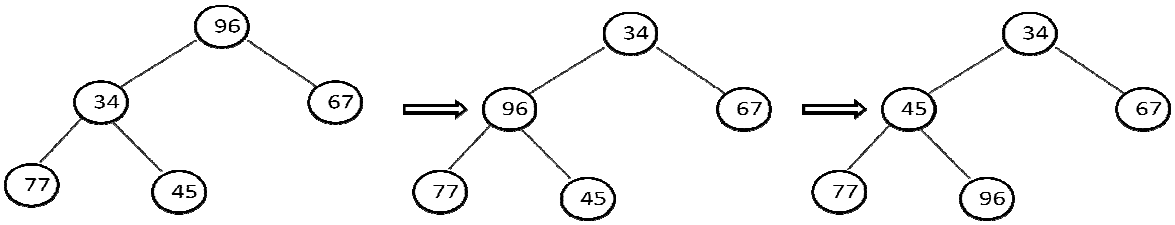


Delete root value and store in array i.e., the root value is stored in external data structure and value of last node is copied into root node and last node will be deleted.



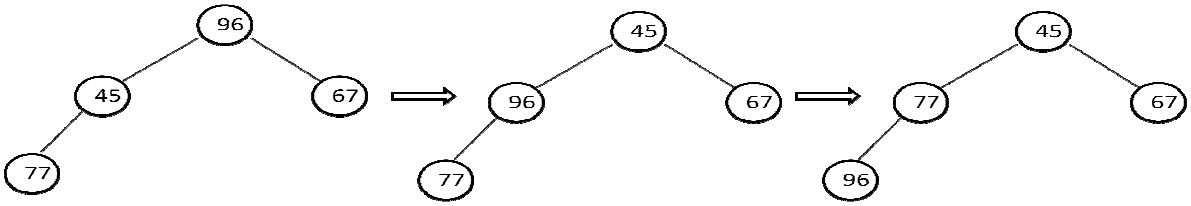
Delete root value and store in array i.e., the root value is stored in external data structure and value of last node is copied into root node and last node will be deleted.

12	24					
0	1	2	3	4	5	6



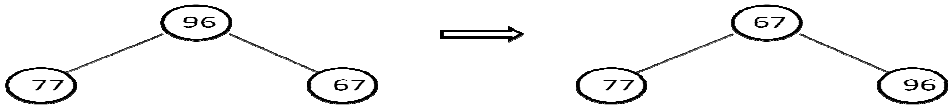
Delete root value and store in array i.e., the root value is stored in external data structure and value of last node is copied into root node and last node will be deleted.

12	24	34				
0	1	2	3	4	5	6



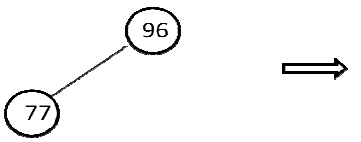
Delete root value and store in array

12	24	34	45			
0	1	2	3	4	5	6



Delete root value and store in array

12	24	34	45	67		
0	1	2	3	4	5	6



Delete root value and store in array

12	24	34	45	67	77	
0	1	2	3	4	5	6

Delete root value and store in array

12	24	34	45	67	77	96
0	1	2	3	4	5	6

```
/*******************************************************
                    Heap Sort Technique
*******************************************************/
        #include<stdio.h>
        #include<conio.h>
        display(a,n)
        int a[],n;
        {
                int i;
                for(i=0;i<n;i++)
                printf("%d ",a[i]);
                printf("\n");
        }

        /*-------------------------------------- Function to create heap */
        createheap(x,n)
        int x[],n;
        {
          int i,ele,s,f;
          for (i=1;i<n;i++)
          {
            ele = x[i];
            s = i;
            f = (s-1) / 2;
            while (s>0 && x[f]<ele)
             {
                   x[s] = x[f];
                   s = f;
                   f = (s-1) / 2;
             }
             x[s] = ele;
          }
    }



         swap(x,i,j)
         int x[],i,j;
         {
           int temp;
           temp=x[i];
           x[i]=x[j];
           x[j]=temp;
         }
```

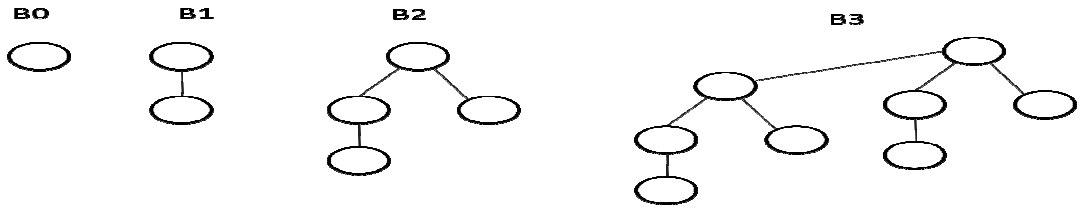
```
 /* -------------------------repeatedly remove x[0] and insert it in proper position
                                                        and readjust the remaining heap */
 heapsort(x,n)
 int x[],n;
 {
  int i;
  for (i=n-1;i>0;i--)
   {
         display(x,n);
         swap(x,0,i);
         createheap(x,i);
   }
 }

void main()
{
 static int a[20] = { 25,57,48,37,12,92,86,33 };
 int n,i;
 clrscr();
 createheap(a,8);
 heapsort(a,8); /* function call */
 printf("the sorted array is:");
 display(a,8);
 getch();
}

/*-------------output----------------------
92 37 86 33 12 48 57 25
86 33 57 25 12 37 48 92
57 33 48 25 12 37 86 92
48 33 37 25 12 57 86 92
37 25 33 12 48 57 86 92
33 12 25 37 48 57 86 92
25 12 33 37 48 57 86 92
THE SORTED ARRAY
12 25 33 37 48 57 86 92
-----------------------------------------*/
```

Binomial Queue/ Binomial Heap/ Binomial Tree

A Binomial Tree is a forest that contains a collection of B_0 to B_k trees.

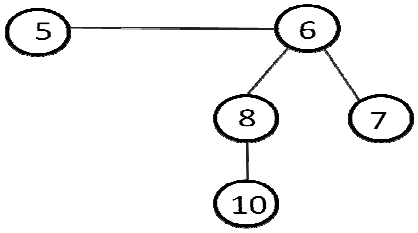
Binomial queues is a collection of heap-ordered trees. Each of the heap-ordered trees is called a *binomial tree* with the following constraints:

- There is at most one binomial tree of every height.
- A binomial tree of height 0 is a one-node tree; a binomial tree, *Bk*, of height *k* is formed by attaching a binomial tree, *Bk*−1, to the root of another binomial tree, *Bk*−1.

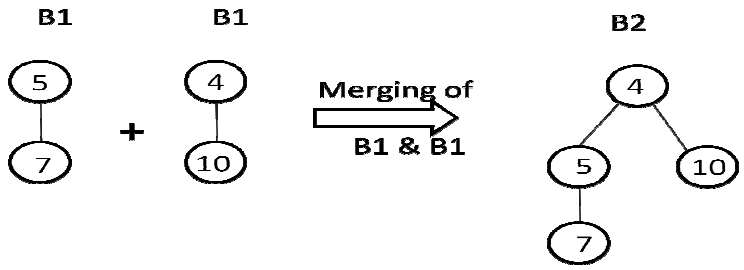


2^0 2^1 2^2 2^3 no of nodes

Binomial queue of 5 elements (combination of sub trees B_0 and B_2) binomial queue of B0 contains 1 element. Binomial queue B2 contains 4 elements. The union of B0 and B2 is as follows.



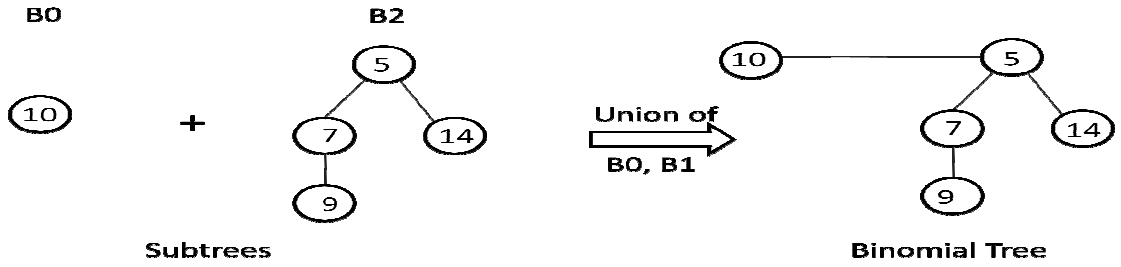
A sub tree B_k is generated by joining / merging of two B_{k-1} subtrees. For example B1 contains 2 elements. Combining two B1 trees forms B2 tree(Merging).



Given any n elements the sub trees in binomial queue are obtained by representing n in binary form. A bit in binary form may be 1 or 0. A 1 represents subtree. For example to form binomial tree of 5 elements the binary of 5 is 0 1 0 1 i.e.

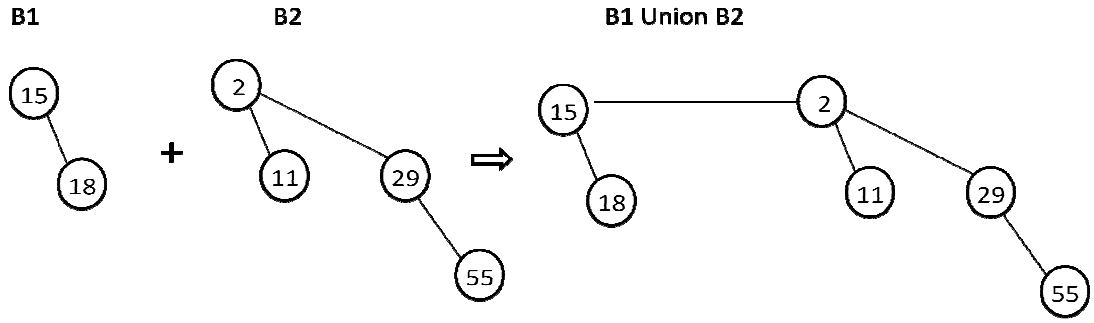
2^3 2^2 2^1 2^0

B2 B0 are subtrees of the binomial queue



Given n=6, its binary representation is 0110 hence subtrees B2 and B1combined to form binomial queue. B2 contains 4 elements and B1 contains 2 elements.

The picture below shows a binomial queue consisting of six elements with two binomial trees *B*1 and *B*2:



Operations:

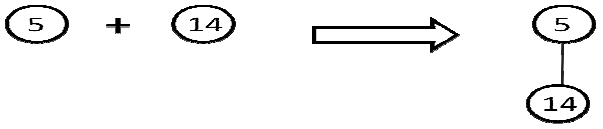
1)Insertion

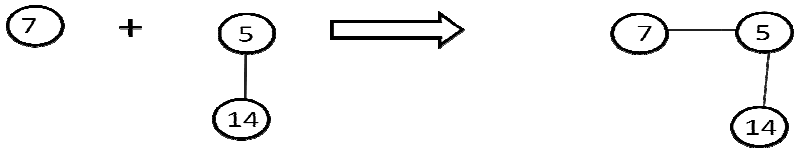
Step 1 : Given a binomial tree T, let an element e be inserted as node in subtree B0.

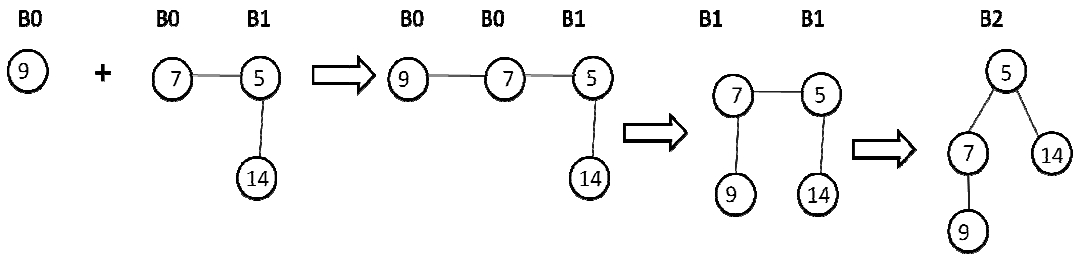
Step 2 : As a min heap with subtrees till Bk where B_k equals B_{k-1} + B_{k-1}. In other words subtrees of same kind can be merged as min heap.

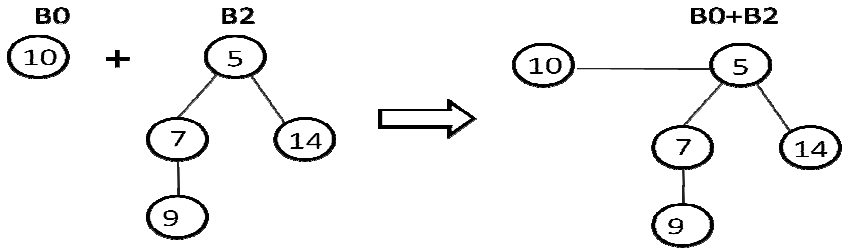
Example 1) Create a binomial tree with 5 14 7 9 10

Total 5 elements 101 B2 and B0 will be merged



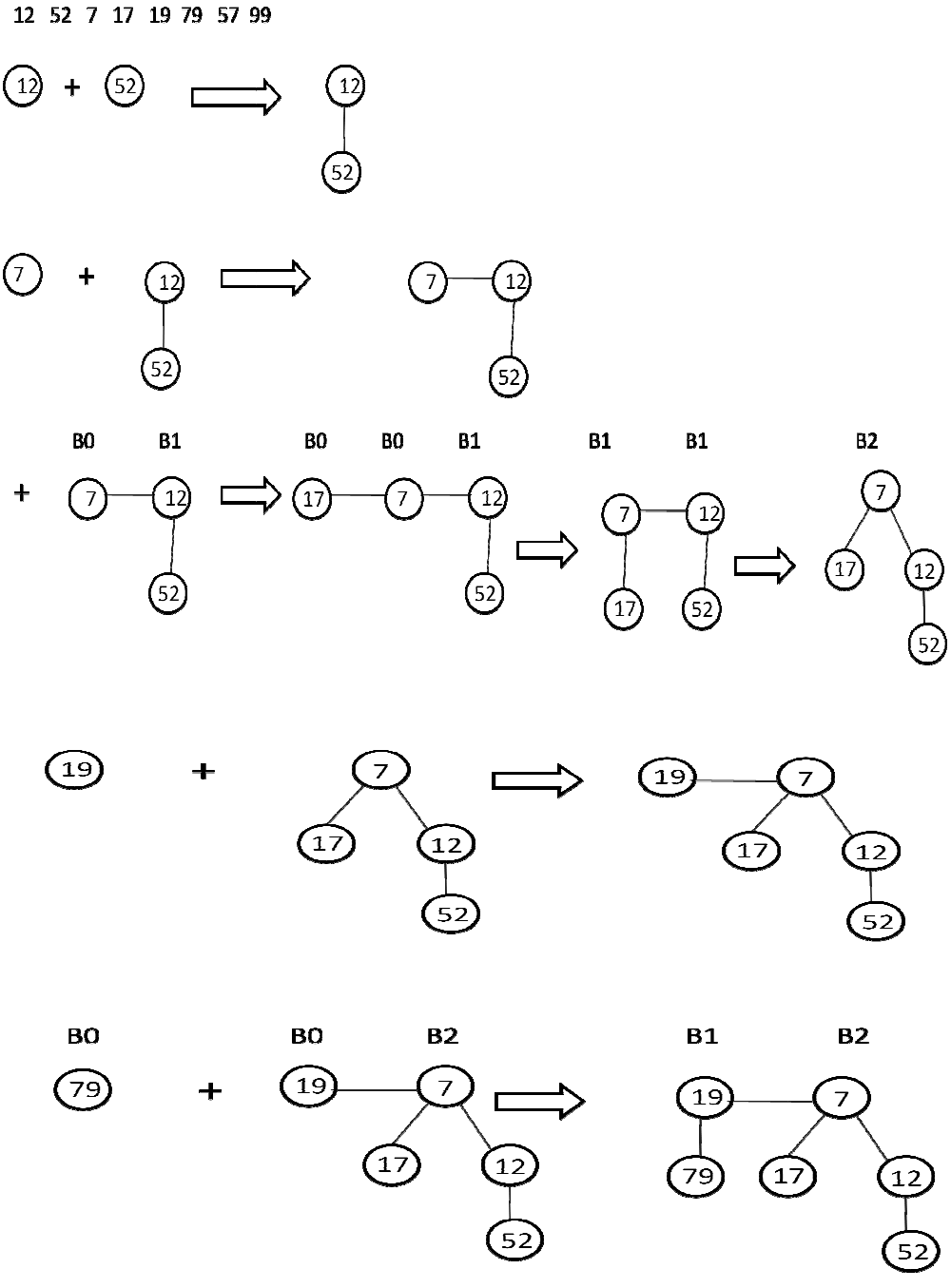


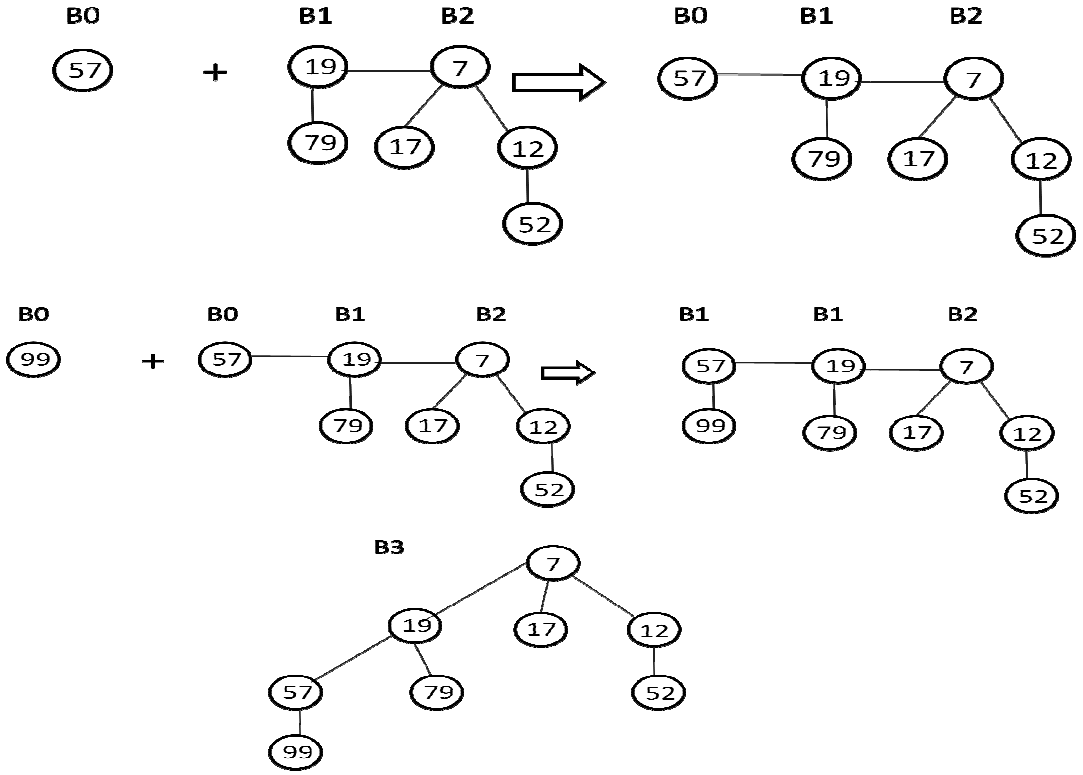




Example 2) Construct Binomial tree using 12 52 7 17 19 79 57 99

Total elements are 8 its binary value is 1000 where B3 is active.





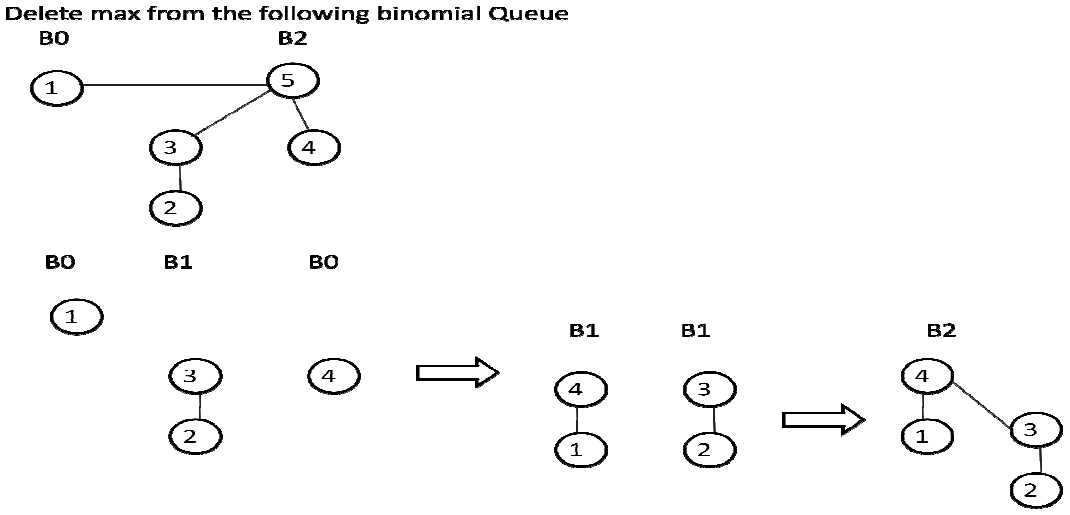
Searching

Just like in a heap roots of subtrees contain minimum or maximum element in a binomial tree.
A minimum element is searched among B_k B_{k-1} -------- B_0 subtrees

A simple comparison with roots of those subtrees is a subtree with least element or largest element.

Delete min or max from Binomial Queue

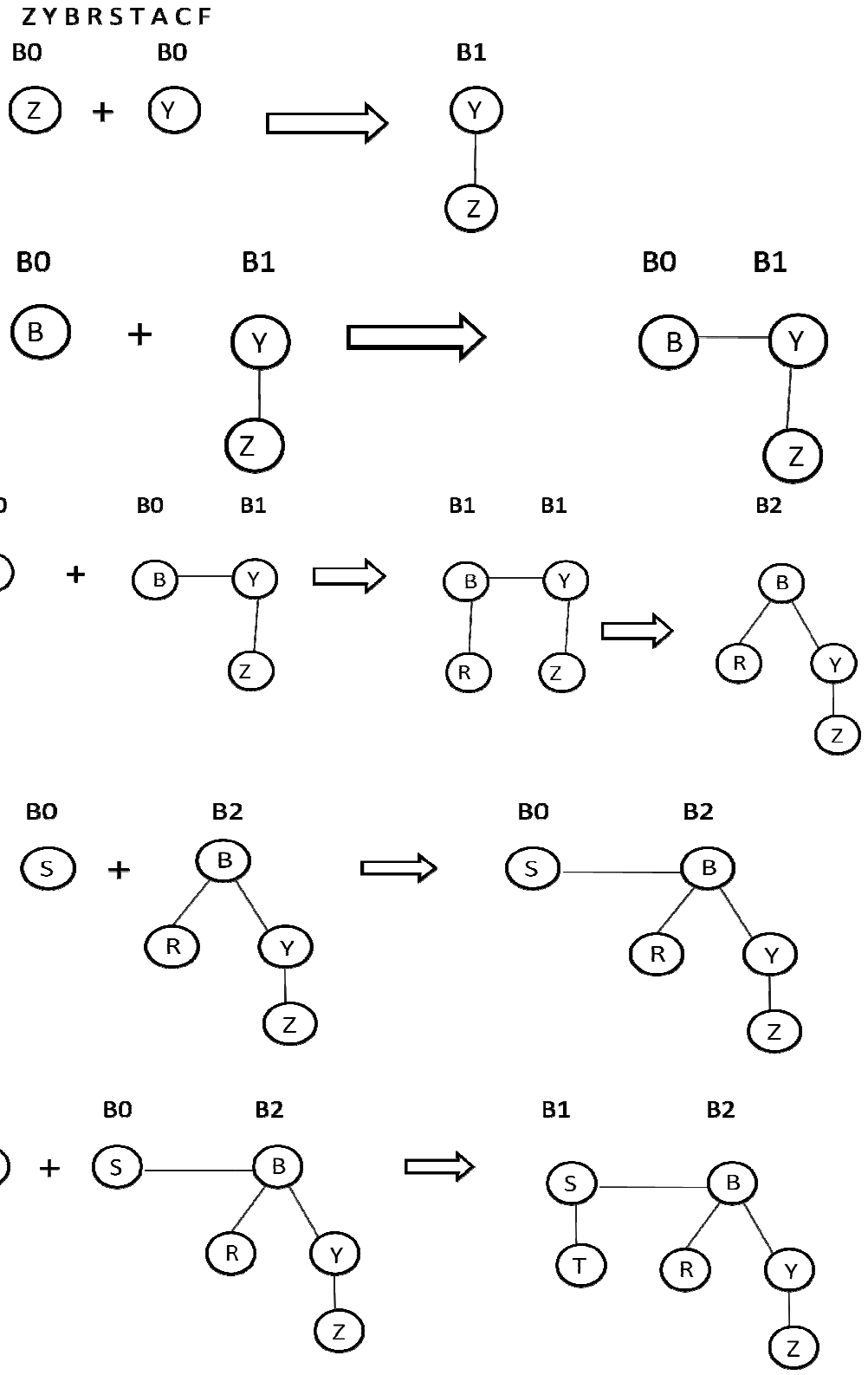
Deleting a minimum or maximum element from a binomial queue generates subtrees B_{k-1} ---- B_0.
If an element is in subtree B_k subtrees are merged.

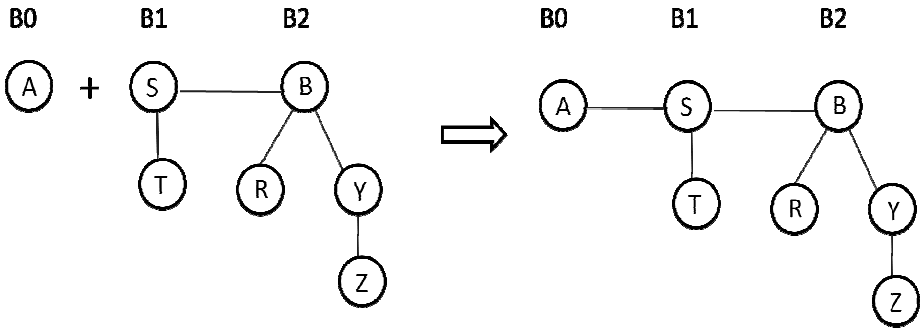


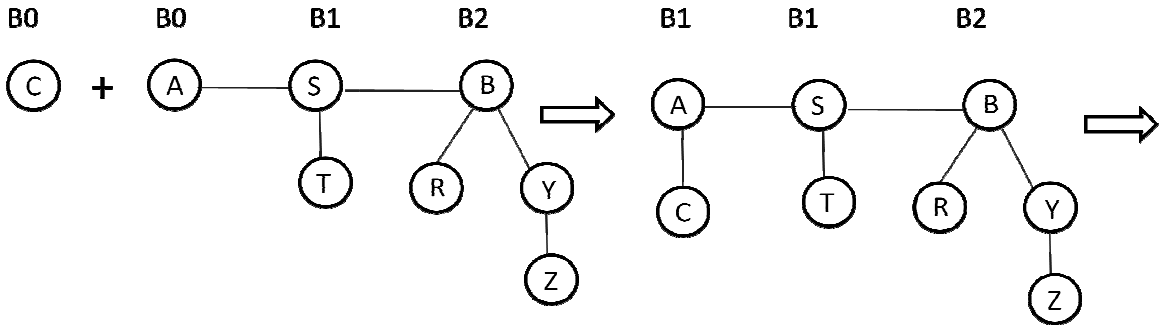
Example : Construct a binomial queue for the following nodes using min heap and perform three deletions.

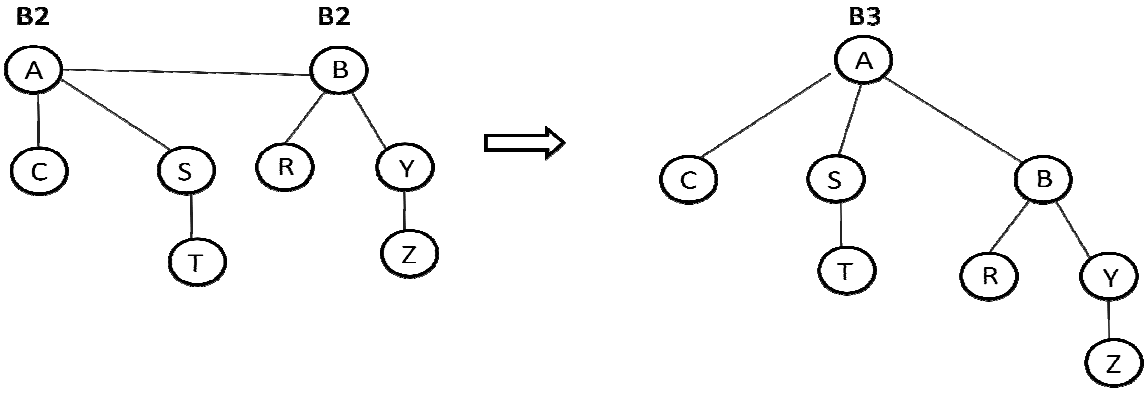
Z	Y	B	R	S	T	A	C	F
26	25	2	18	19	20	1	3	6

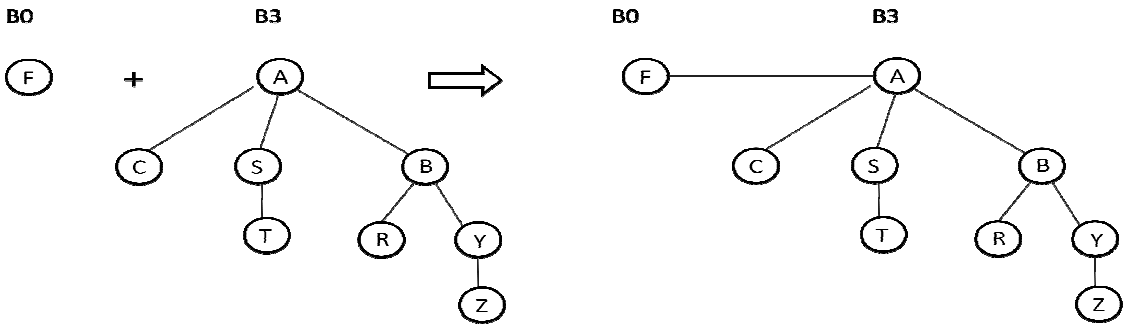
total 9 elements 1001 B_3 and B_0

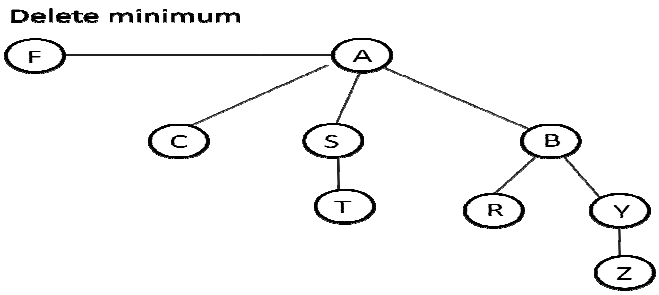


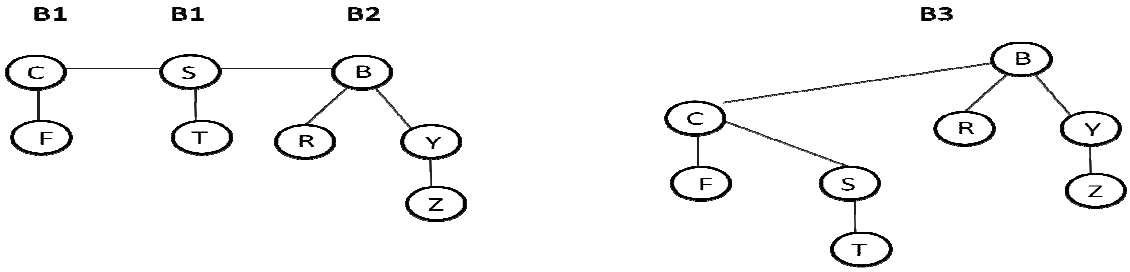

B0
B1
B2
A
+
S
B
T
R
Y
Z
B0
B1
B2
A
S
B
T
R
Y
Z


B0
B0
B1
B2
C
+
A
S
B
T
R
Y
Z
B1
B1
B2
A
S
B
C
T
R
Y
Z

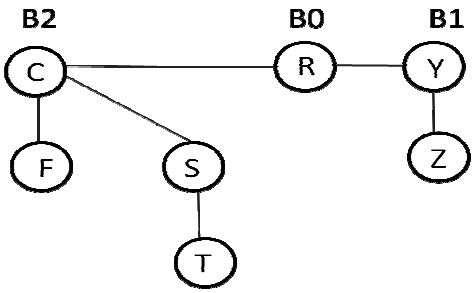

B2
B2
A
B
C
S
R
Y
T
Z
B3
A
C
S
B
T
R
Y
Z


B0
B3
F
+
A
C
S
B
T
R
Y
Z
B0
B3
F
A
C
S
B
T
R
Y
Z

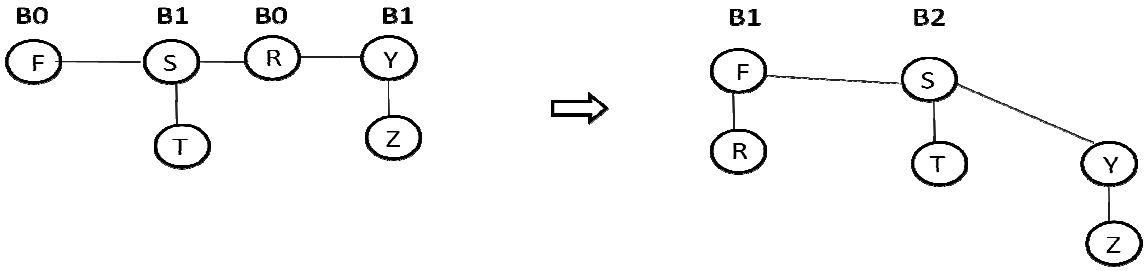




Delete minimum

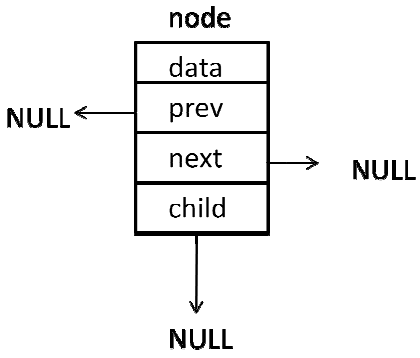


After deleting node 'c' the binomial tree will be as follows



Binomial Queue consists of different sub trees with data in them. A binomial queue is represented by using a structure as follows.

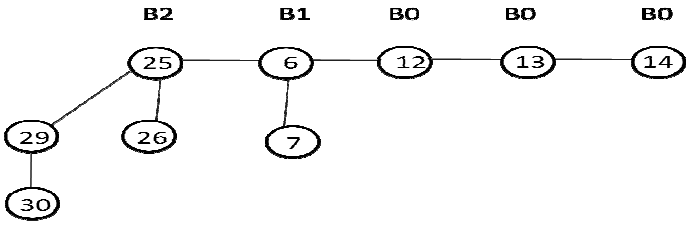
```
struct node
{
   int data;
   struct node *prev;
   struct node *next;
   struct node *child;
};
```



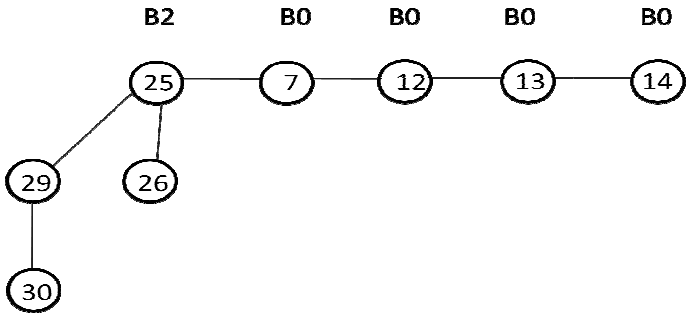
Lazy Binomial Queue

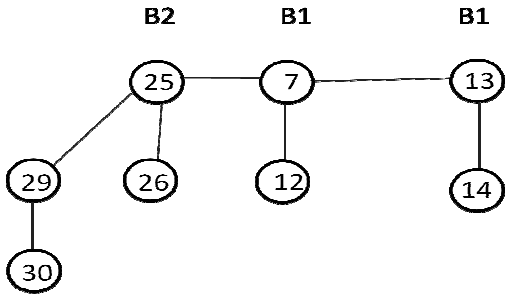
Lazy Binomial Queues are special Binomial queues in which merging is postponed until a deletion operation is performed.

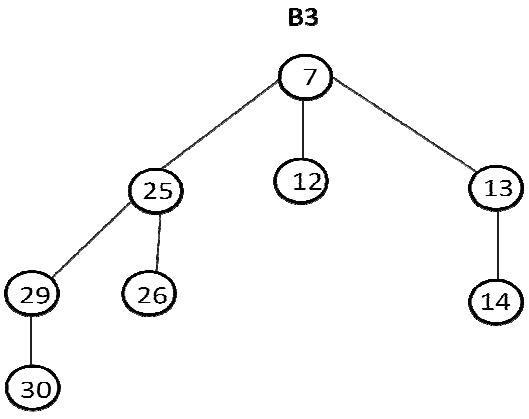
Eg:



Delete 6



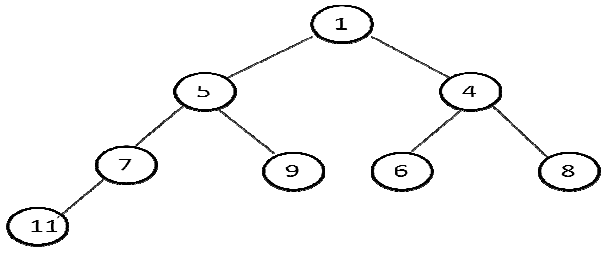




QUESTIONS FROM PREVIOUS UNIVERSITY EXAMINATIONS

1) What is the maximum height of a binary heap with n nodes?

Let the size of heap be **N** and height be **h**
The value of h in a complete binary tree is ceil($\log_2$(N+1)) – 1.



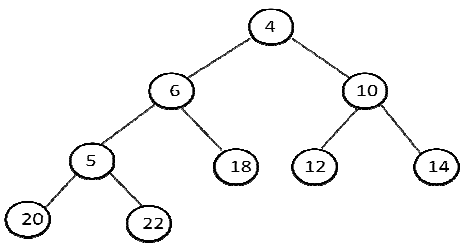
N=8
Maximum height = ceil(log(9))-1
= 4-1 = 3

2) Write and explain build heap algorithm. Give an example.
Algorithm
Step 1) Initialize heap i.e. array with no nodes
Step 2) Add a new element to the end of an array;
Step 3) Sift up the new element, while heap property is broken. Sifting is done as following: compare node's value with parent's value. If they are in wrong order, swap them.
Step 4) Repeat step 2 and 3 for all the nodes we want to insert to heap

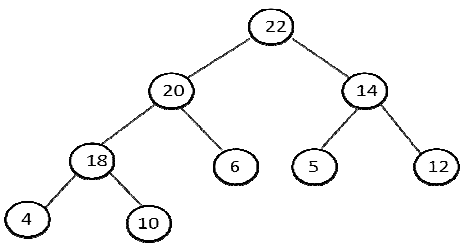
Explanation: we are going to create a heap. Initially heap is empty. Add first node. Add second node as last node in the array. Reheapify. Each time we add a node that node will be added as last node of the array and all the array is reheapified.

3) Calculate amortized insertion cost of binomial heaps.
insert is *O(logn)*
4) Explain the binary heap structure property?

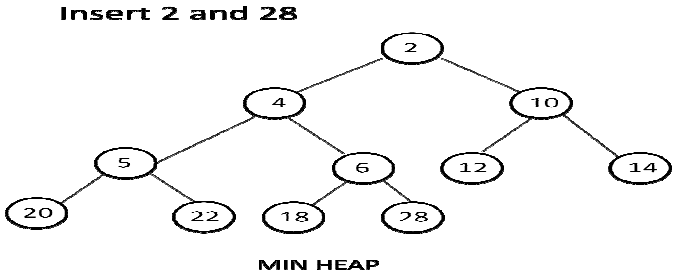
5) Do the following operations of priority queues?
   i) Construct the min and max priority queue with the following elements 20, 10, 5, 18, 6, 12, 14, 4, and 22.
   ii) Insert 2 and 28 in the min priority queue.
   iii) Perform two successive deletion operations on max priority queue.

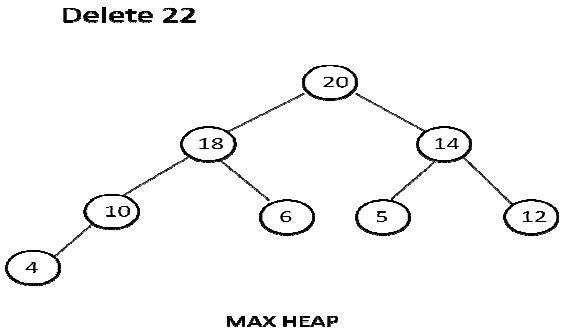


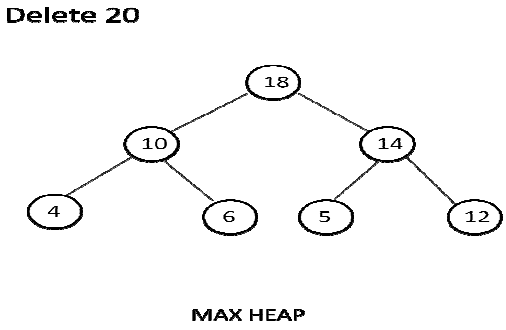
MIN HEAP



MAX HEAP







6) Explain binary heap – order -- property.

A **binary heap** is a heap data structure that takes the form of a binary tree. Binary heaps are a common way of implementing priority queues. A binary heap is defined as a binary tree with two additional constraints:

- Shape property: a binary heap is a *complete binary tree*; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.

- Heap property: the key stored in each node is either greater than or equal to (≥) or less than or equal to (≤) the keys in the node's children, according to some total order.

Heaps where the parent key is greater than or equal to (≥) the child keys are called *max-heaps*; those where it is less than or equal to (≤) are called *min-heaps*.

7) Explain the implementation methods of priority queues.

A priority queue is a collection of elements such that each element has an associated priority. Priority Queue is a data structure like stack, queue where insertions, deletions and retrieval is done based on Priority.

A priority can be either Maximum or Minimum.
Priority queues are implemented using 1) Heap
2) Binomial Queue

8) Write insertion and deletion algorithms of priority heap.

Insertion Algorithm : page 4

Deletion Algorithm : Page 6

9) Define a priority queue.

A priority queue is a collection of elements such that each element has an associated priority. Priority Queue is a data structure like stack, queue where insertions, deletions and retrieval is done based on Priority.

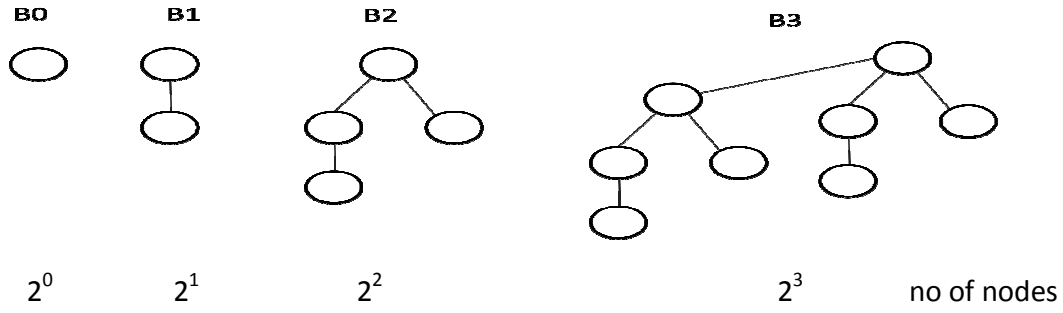
A priority can be either Maximum or Minimum.
Priority queues are implemented using 1) Heap
2) Binomial Queue

10) Define binomial queues. Explain the properties of binomial queues.

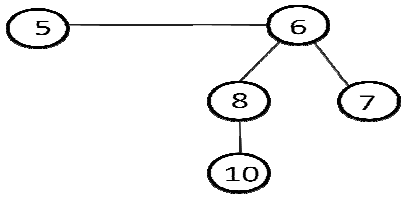
A binomial tree is a forest that contains a collection of B_0 to B_k trees.

Binomial queues is a collection of heap-ordered trees. Each of the heap-ordered trees is called a *binomial tree* with the following constraints:

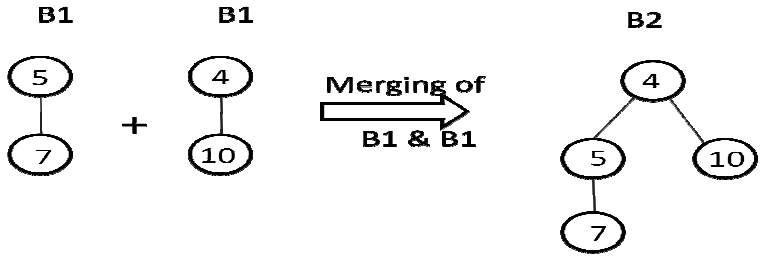
- There is at most one binomial tree of every height.
- A binomial tree of height 0 is a one-node tree; a binomial tree, *Bk*, of height *k* is formed by attaching a binomial tree, *Bk*−1, to the root of another binomial tree, *Bk*−1.



Binomial queue of 5 elements (combination of sub trees B_0 and B_2) binomial queue of B0 contains 1 element. Binomial queue B2 contains 4 elements. The union of B0 and B2 is as follows.

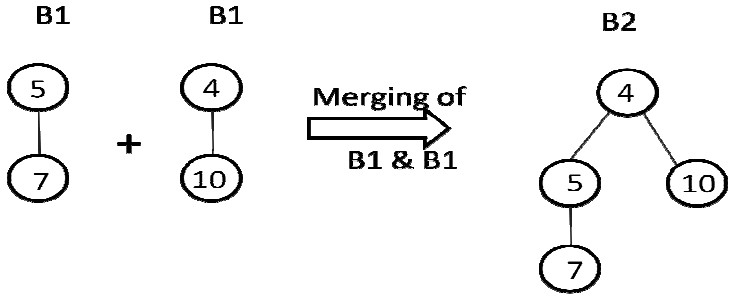


A sub tree B_k is generated by joining / merging of two B_{k-1} subtrees. For example B1 contains 2 elements. Combining two B1 trees forms B2 tree.

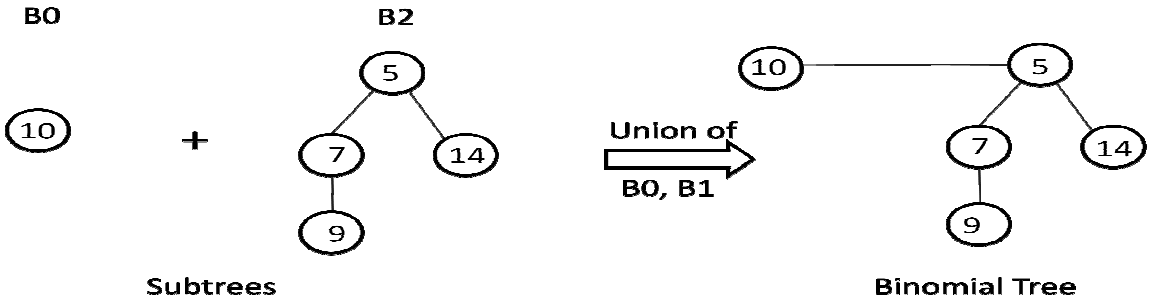


11) Explain the following operations of binomial queues with suitable example.
    i) Merging of two binomial queues
    ii) Union of two binomial queues.

Combining two Bk-1 trees to form a tree of Bk is called merging

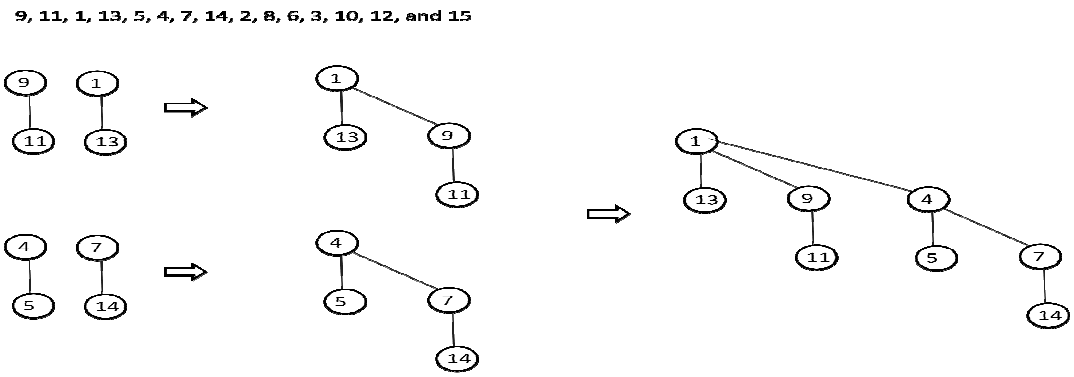


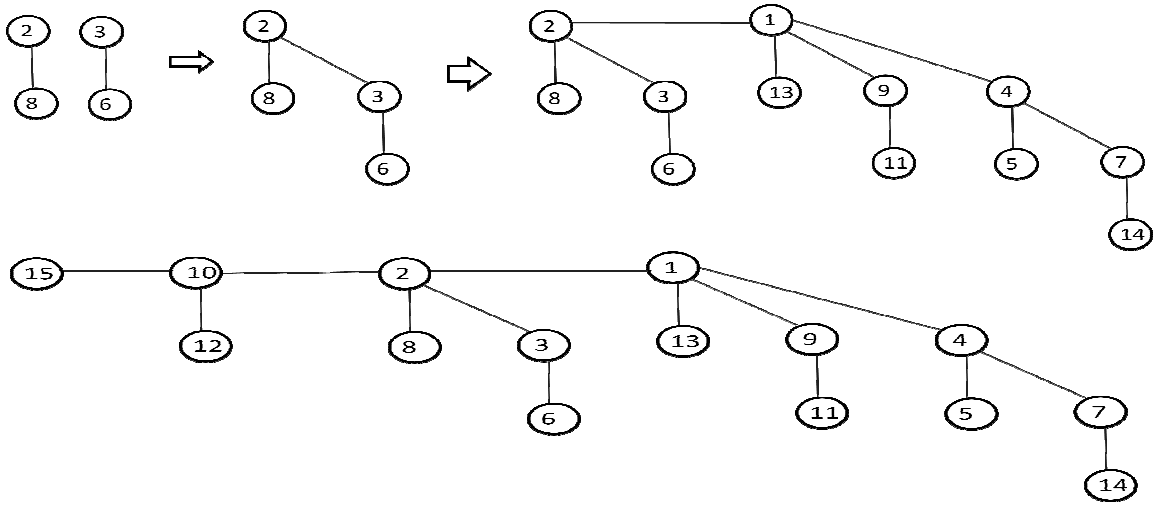
Combining any two binomial trees is called union



12) List the operations supported by a minimum priority queue.
    Insert, Delete, Find min/find max

13) Show the result of constructing a binomial heap using the following elements. 9, 11, 1, 13, 5, 4, 7, 14, 2, 8, 6, 3, 10, 12, and 15 one at a time into an initially empty binomial heap.

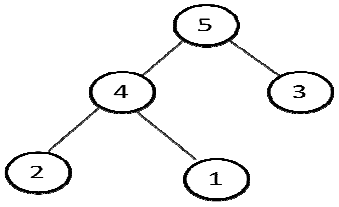




14) Show the resultant binomial heap after perform delete minimum element and reconstruct the binomial heap twice on the above constructed binomial heap.

15) What is a max heap? What are its applications?

A max heap is a complete binary tree in which a root of every sub tree has larger element to the data compared to its children.



Heap Applications:

a. Heap Sort
b. Prim's Algorithm
c. Dijkstra's algorithm
d. Order Statistics

16) Briefly discuss about different implementations of priority queues. Also compare these implementations w.r.t. time complexity for basic priority queue operations.

17) Write and explain buildheap algorithm with an example. Also analyse its time complexity.

Algorithm for Build Heap

Now, let us construct general algorithm to insert elements into an empty heap. Repeating the insert algorithm for all the nodes to be inserted becomes algorithm form build heap.

Step 1) create a hole and add a new element to the end of an array;

Step 2) Sift up the new element, while heap property is broken. Sifting is done as following: compare node's value with parent's value. If they are in wrong order, swap them.

Step 3) Repeat steps 1 and 2 for all the elements to be inserted.

```
insert(int num, int location)
{
   int parentnode;
   while (location > 0)
   {
        parentnode =(location - 1)/2;
        if (num <= array[parentnode])
        {
           array[location] = num;
           return;
        }
        array[location] = array[parentnode];
        location = parentnode;
   }/*End of while*/
   array[0] = num; /*assign number to the root node */
}/*End of insert()*/
```

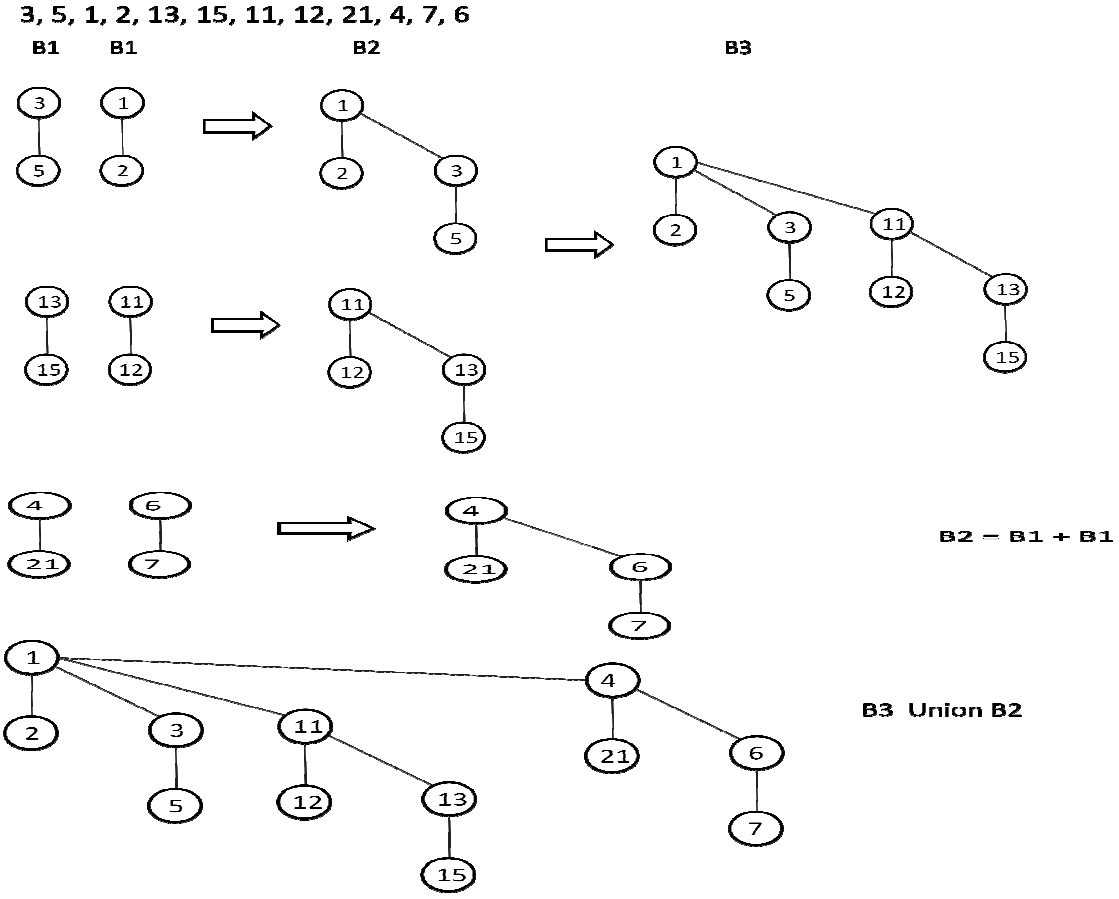
Complexity analysis

Complexity of the insertion operation is O(h), where **h** is heap's height. Taking into account completeness of the tree, O(h) = O(log n), where **n** is number of elements in a heap.

18) Insert the below list of elements into an initially empty binomial queue. 3, 5, 1, 2, 13, 15, 11, 12, 21, 4, 7, 6.

Total number of elements in the binomial queue is 12

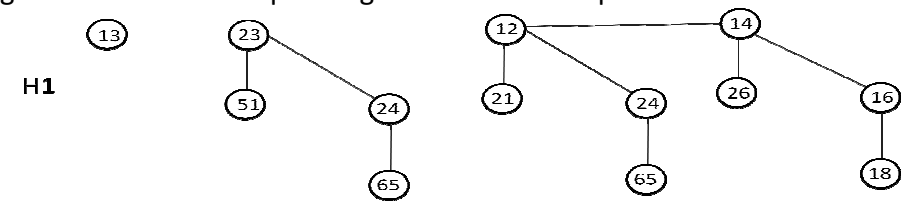
Binary value of 12 = 1100 = B3 B2 B1 B0 = B3 and B2 are active

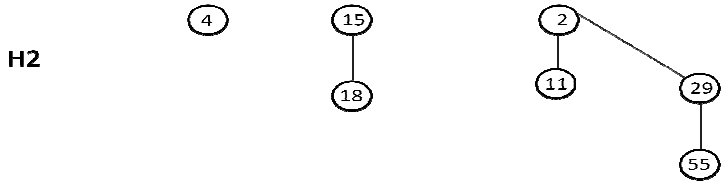


19) How a priority queue is different from normal queues?

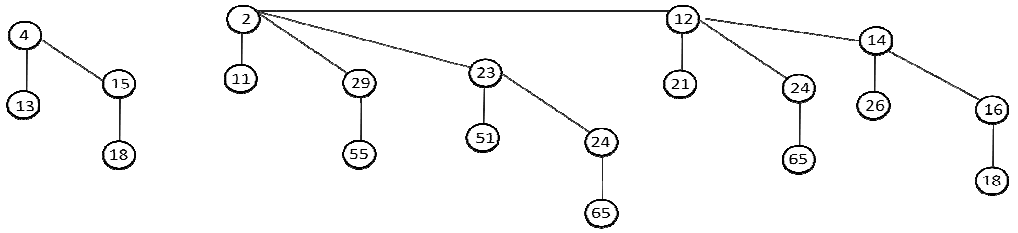
A priority queue is different from a "normal" queue, because instead of being a "first-in-first-out" data structure, values come out in order by priority.

20) Merge the two binomial queues given below. Then perform two delete min operations

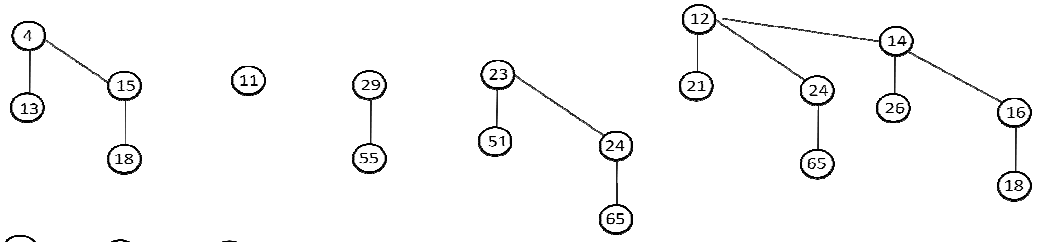


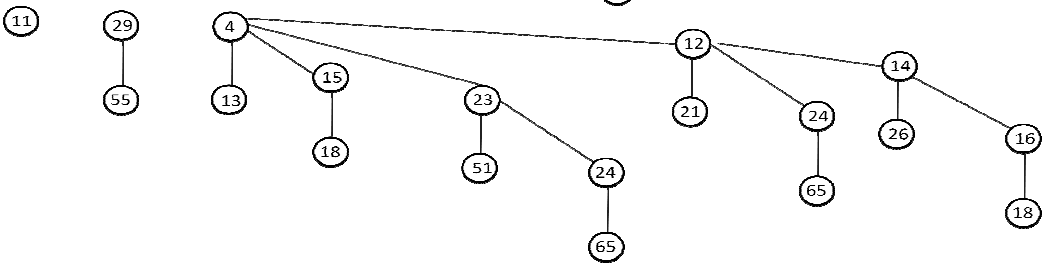


Resultant binomial queue H3 after merging H1 and H2

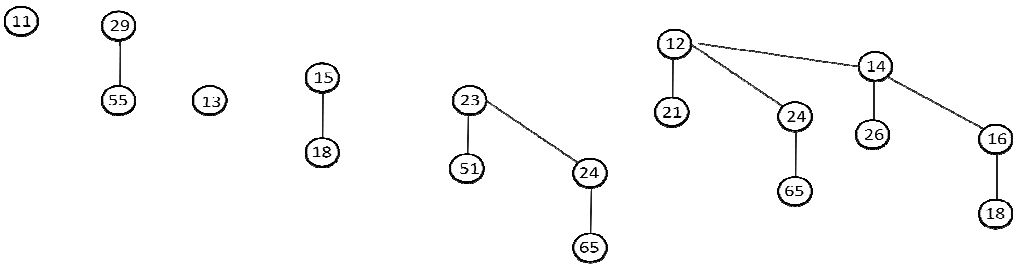


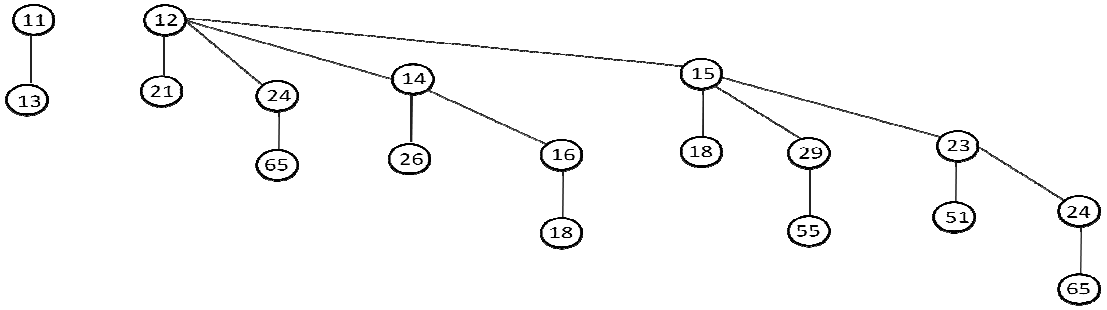
Delete min





Delete node 4





21) Present the basic model of priority queues.

Page 1

22) Explain the priority queue solution for event simulation problem

In a bank, where customers arrive and wait on a line until one of *k* tellers is available. Customer arrival is governed by a probability distribution function, as is the service time (the amount of time to be served once a teller is available). We are interested in statistics such as how long on average a customer has to wait or how long the line might be.

With certain probability distributions and values of *k*, these answers can be computed exactly. However, as *k* gets larger, the analysis becomes considerably more difficult, so it is appealing to use a computer to simulate the operation of the bank. In this way, the bank officers can determine how many tellers are needed to ensure reasonably smooth service.

A simulation consists of processing events. The two events here are (a) a customer arriving and (b) a customer departing, thus freeing up a teller.

We can use the probability functions to generate an input stream consisting of ordered pairs of arrival time and service time for each customer, sorted by arrival time. We do not need to use the exact time of day. Rather, we can use a quantum unit, which we will refer to as a *tick*.

One way to do this simulation is to start a simulation clock at zero ticks. We then advance the clock one tick at a time, checking to see if there is an event. If there is, then we process the event(s) and compile statistics. When there are no customers left in the input stream and all the tellers are free, then the simulation is over.

The problem with this simulation strategy is that its running time does not depend on the number of customers or events (there are two events per customer), but instead depends on the number of ticks, which is not really part of the input. To see why this is important, suppose we changed the clock units to milliticks and multiplied all the times in the input by 1,000. The result would be that the simulation would take 1,000 times longer!

The key to avoiding this problem is to advance the clock to the next event time at each stage. This is conceptually easy to do. At any point, the next event that can occur is either (a) the next customer in the input file arrives, or (b) one of the customers at a teller leaves. Since all the times when the events will happen are available, we just need to find the event that happens nearest in the future and process that event.

If the event is a departure, processing includes gathering statistics for the departing customer and checking the line (queue) to see whether there is another customer waiting. If so, we add that customer, process whatever statistics are required, compute the time when that customer will leave, and add that departure to the set of events waiting to happen.

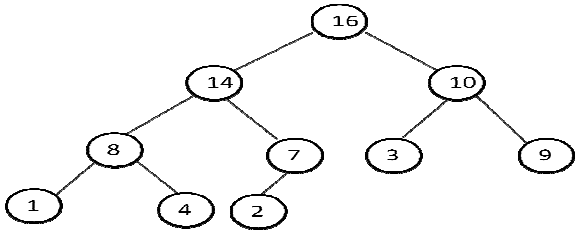
If the event is an arrival, we check for an available teller. If there is none, we place the arrival on the line (queue); otherwise we give the customer a teller, compute the customer's departure time, and add the departure to the set of events waiting to happen.

The waiting line for customers can be implemented as a queue. Since we need to find the event *nearest* in the future, it is appropriate that the set of departures waiting to happen be organized in a priority queue. The next event is thus the next arrival or next departure (whichever is sooner); both are easily available.

It is then straightforward, although possibly time-consuming, to write the simulation routines. If there are C customers (and thus $2C$ events) and k tellers, then the running time *of the simulation* would be $O(C \log(k + 1))$* because computing and processing each event takes $O(\log H)$, where $H = k + 1$ is the size of the heap.

* We use $O(C \log(k + 1))$ instead of $O(C \log k)$ to avoid confusion for the $k = 1$ case.

23) Create a heap from the following elements by inserting all of them at once using buildHeap algorithm. 4, 1, 3, 2, 16, 9, 10, 14, 8, 7
Max heap



24) How many trees will be there in a binomial queue at 30 elements?

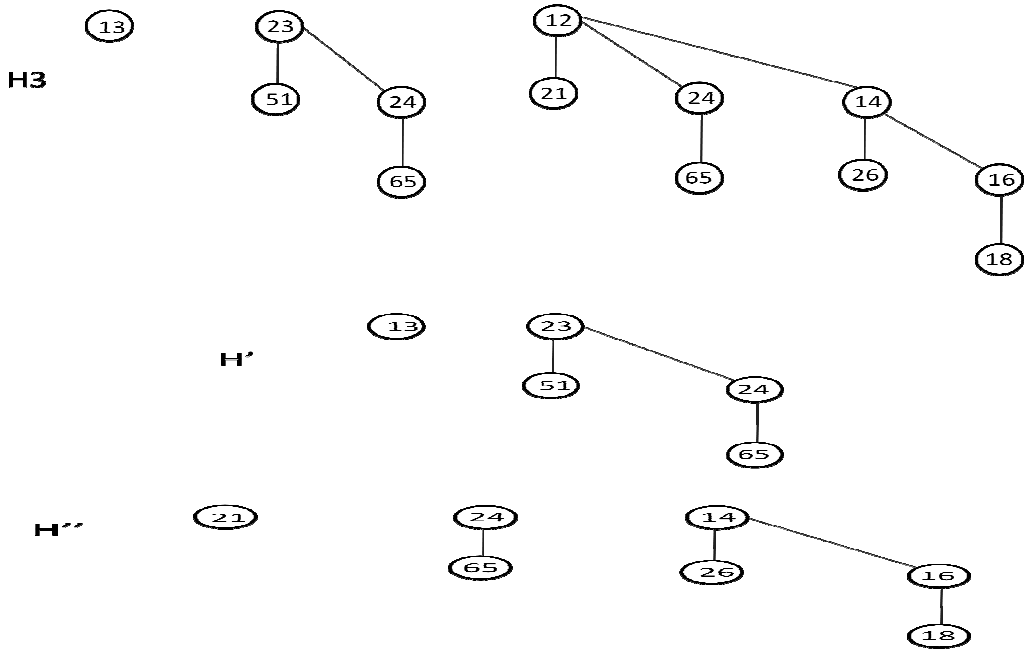
Binary value of 30 is 11110 which is B4 B3 B2 B1 B0 i.e. consider the trees corresponding to 1. So there are 4 binary trees.

25) Explain the procedure for deleteMin operation in binomial queues with an example.

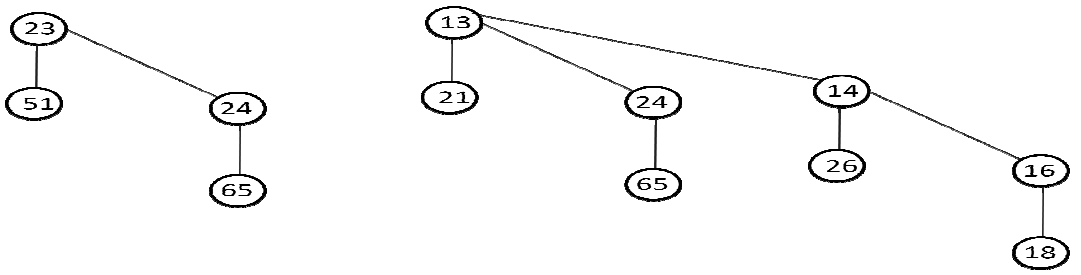
DeleteMin

a. find the binomial tree with the smallest root. Let this tree be Bk, and let the original priority queue be H.
b. Remove the binomial tree Bk from the forest of trees in H, forming the new binomial queue H′.
c. Remove the root of Bk, creating binomial trees B0,B1,…,Bk−1, which collectively form priority queue H″.
d. merge H′ and H″.

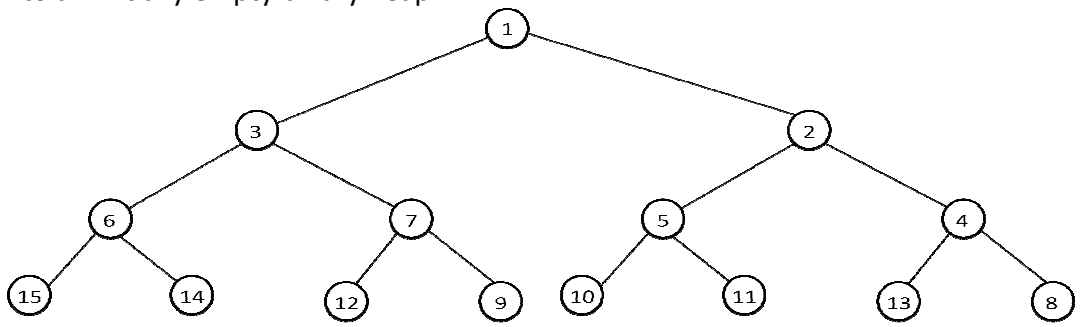
Suppose we perform a DeleteMin on H3 . The minimum root is 12, and we have H′ and H″ below:



and our final result is :



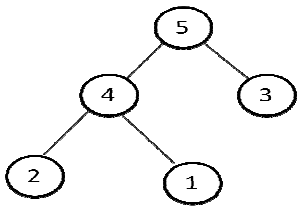
26) Show the result of inserting 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13 and 2 one at a time into an initially empty binary heap.



27) What is a binary max heap? Give an example.

A heap is a hierarchical data structure in which operations are done based on maximum or minimum priority.

A max heap is a complete binary tree in which a root of every sub tree has larger element to the data compared to its children.



28) Derive expression for asymptotic time complexity of build-heap algorithm.

Page 8

29) Explain event simulation problem with an example.

Same as Q No 22

30) Briefly discuss insertion and deletion to double ended priority queue.

Double ended priority queue

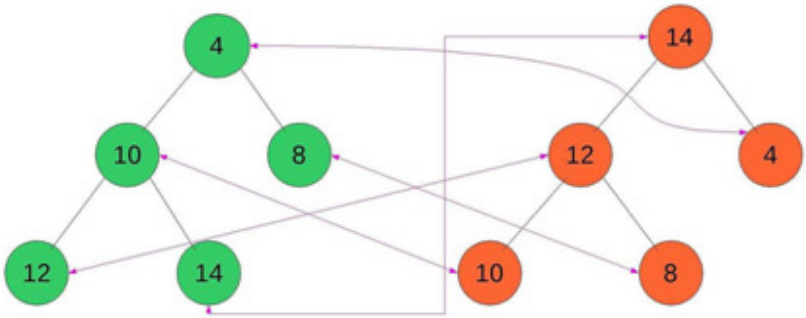
A double ended priority queue supports operations of both max heap (a max priority queue) and min heap (a min priority queue). The following operations are expected from double ended priority queue.

1. getMax() : Returns maximum element.
2. getMin() : Returns minimum element.
3. deleteMax() : Deletes maximum element.
4. deleteMin() : Deletes minimum element.
5. size() : Returns count of elements.
6. isEmpty() : Returns true if the queue is empty.

Implementation:

Double-ended priority queues can be built from balanced binary search trees (where the minimum and maximum elements are the leftmost and rightmost leaves, respectively), or using specialized data structures like min-max heap and pairing heap.

Generic methods of arriving at double-ended priority queues from normal priority queues are



minHeap maxHeap

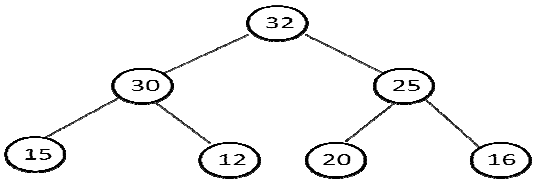
Dual structure method

In this method two different priority queues for min and max are maintained. The same elements in both the PQs are shown with the help of correspondence pointers. Here, the minimum and maximum elements are values contained in the root nodes of min heap and max heap respectively

- **Removing the min element**: Perform removemin() on the min heap and remove(*node value*) on the max heap, where *node value* is the value in the corresponding node in the max heap.

- **Removing the max element**: Perform removemax() on the max heap and remove(*node value*) on the min heap, where *node value* is the value in the corresponding node in the min heap.

31) The elements 32, 15, 20, 30, 12, 25 and 16 are inserted one by one in the given order into a max heap. What is the resultant Max-heap?



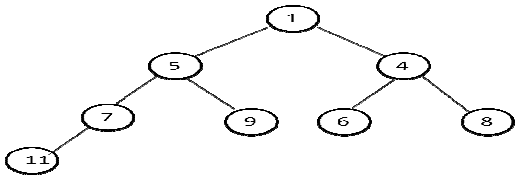
32) Explain single ended priority queue operations

A priority queue is a collection of elements such that each element has an associated priority. Priority Queue is a data structure like stack, queue where insertions, deletions and retrieval is done based on Priority.

A priority can be either Maximum or Minimum.
Priority queues are implemented using 1) Heap
2) Binomial Queue

A heap is a hierarchical data structure in which operations are done based on maximum or minimum priority.

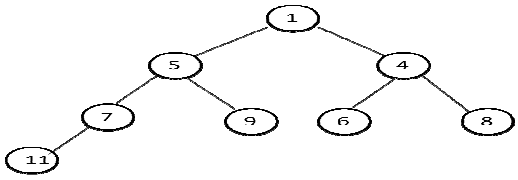
A min heap is a complete binary tree in which a root of sub tree is smaller than its children.



Operations : Insertion, deletion

33) What is ascending priority queue? Explain how to implement this using binary heap? Explain the insertion and deletion operations performed on binary heap with example.

A min heap is a complete binary tree in which a root of sub tree is smaller than its children.



Insertion Algorithm

Now, let us construct general algorithm to insert a new element into a max heap. 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.

Deletion from Heap / Process of Deletion:

Since deleting an element at any intermediary position in the heap can be costly, so we can simply replace the element to be deleted by the last element and delete the last element of the Heap.

Algorithm

Step 1) Replace the root or element to be deleted by the last element.
Step 2) Delete the last element from the Heap.
Step 3) Since, the last element is now placed at the position of the root node. So, it may not follow the heap property. Therefore, **heapify** the last node placed at the position of root.

34) What is a priority queue?

A priority queue is a collection of elements such that each element has an associated priority. Priority Queue is a data structure like stack, queue where insertions, deletions and retrieval is done based on Priority.

A priority can be either Maximum or Minimum.
Priority queues are implemented using 1) Heap
2) Binomial Queue

A heap is a hierarchical data structure in which operations are done based on maximum or minimum priority.

A binomial Queue is a forest that contains a collection of B_0 to B_k trees

35) Applications of Priority Queues

a)The Selection Problem
b)Event Simulation

The Selection Problem : The selection problem is to find the *k*th largest element.
We usually read the elements into an array and sort them, returning the appropriate element. Assuming a simple sorting algorithm, the running time is $O(n^2)$.
We give two algorithms here, both of which run in $O(n \log n)$

Algorithm A
For simplicity, we assume that we are interested in finding the *k*th *largest* element. The algorithm is simple. We read the *n* elements into an array. We then apply the *build_heap* algorithm to this array. Finally, we'll perform *k* *delete_max* operations. The last element extracted from the heap is our answer.

The correctness of the algorithm should be clear. The worst-case timing is $O(n)$ to construct the heap, if *build_heap* is used, and $O(\log n)$ for each *delete_min.* Since there are k *delete_mins*, we obtain a total running time of $O(n + k \log n)$. If $k = O(n/\log n)$, then the running time is dominated by the *build_heap* operation and is $O(n)$. For larger values of k, the running time is $O(k \log n)$. If $k = n/2$, then the running time is $\Theta(n \log n)$.
Notice that if we run this program for $k = n$ and record the values as they leave the heap, we will have essentially sorted the input file in $O(n \log n)$ time.

Algorithm B

For the second algorithm, we return to the original problem and find the kth *largest* element. At any point in time we will maintain a set S of the k largest elements. After the first k elements are read, when a new element is read, it is compared with the kth largest element, which we denote by S_k. Notice that S_k is the smallest element in S. If the new element is larger, then it replaces S_k in S. S will then have a new smallest element, which may or may not be the newly added element. At the end of the input, we find the smallest element in S and return it as the answer.

Normally we use array to represent S. Here, however, we will use a heap to implement S. The first k elements are placed into the heap in total time $O(k)$ with a call to *build_heap*. The time to process each of the remaining elements is $O(1)$, to test if the element goes into S, plus $O(\log k)$, to delete S_k and insert the new element if this is necessary. Thus, the total time is $O(k + (n - k) \log k) = O(n \log k)$. This algorithm also gives a bound of $\Theta(n \log n)$ for finding the median.

```
/*-------------------------------------------------------------------------
          C program to implement Binomial Heap tree
-------------------------------------------------------------------------*/
 #include<stdio.h>
#include<malloc.h>

struct node {
   int n;
   int degree;
   struct node* parent;
   struct node* child;
   struct node* sibling;
};

struct node* MAKE_bin_HEAP();
int bin_LINK(struct node*, struct node*);
struct node* CREATE_NODE(int);
struct node* bin_HEAP_UNION(struct node*, struct node*);
struct node* bin_HEAP_INSERT(struct node*, struct node*);
struct node* bin_HEAP_MERGE(struct node*, struct node*);
struct node* bin_HEAP_EXTRACT_MIN(struct node*);
int REVERT_LIST(struct node*);
int DISPLAY(struct node*);
struct node* FIND_NODE(struct node*, int);
int bin_HEAP_DECREASE_KEY(struct node*, int, int);
int bin_HEAP_DELETE(struct node*, int);

int count = 1;
```

```
struct node* MAKE_bin_HEAP() {
   struct node* np;
   np = NULL;
   return np;
}

struct node * H = NULL;
struct node *Hr = NULL;

int bin_LINK(struct node* y, struct node* z) {
   y->parent = z;
   y->sibling = z->child;
   z->child = y;
   z->degree = z->degree + 1;
}

struct node* CREATE_NODE(int k)
{
   struct node* p;//new node;
   p = (struct node*) malloc(sizeof(struct node));
   p->n = k;
   return p;
}

struct node* bin_HEAP_UNION(struct node* H1, struct node* H2)
{
   struct node* prev_x;
   struct node* next_x;
   struct node* x;
   struct node* H = MAKE_bin_HEAP();
   H = bin_HEAP_MERGE(H1, H2);
   if (H == NULL)
      return H;
   prev_x = NULL;
   x = H;
   next_x = x->sibling;
   while (next_x != NULL) {
      if ((x->degree != next_x->degree) || ((next_x->sibling != NULL)
            && (next_x->sibling)->degree == x->degree)) {
         prev_x = x;
         x = next_x;
      } else {
         if (x->n <= next_x->n) {
            x->sibling = next_x->sibling;
            bin_LINK(next_x, x);
         } else {
            if (prev_x == NULL)
               H = next_x;
            else
               prev_x->sibling = next_x;
            bin_LINK(x, next_x);
```

```
            x = next_x;
          }
        }
        next_x = x->sibling;
    }
    return H;
}

struct node* bin_HEAP_INSERT(struct node* H, struct node* x)
{
    struct node* H1 = MAKE_bin_HEAP();
    x->parent = NULL;
    x->child = NULL;
    x->sibling = NULL;
    x->degree = 0;
    H1 = x;
    H = bin_HEAP_UNION(H, H1);
    return H;
}

struct node* bin_HEAP_MERGE(struct node* H1, struct node* H2)
{
    struct node* H = MAKE_bin_HEAP();
    struct node* y;
    struct node* z;
    struct node* a;
    struct node* b;
    y = H1;
    z = H2;
    if (y != NULL) {
        if (z != NULL && y->degree <= z->degree)
            H = y;
        else if (z != NULL && y->degree > z->degree)
            /* need some modifications here;the first and the else conditions can be
merged together!!!! */
            H = z;
        else
            H = y;
    } else
        H = z;
    while (y != NULL && z != NULL) {
        if (y->degree < z->degree) {
            y = y->sibling;
        } else if (y->degree == z->degree) {
            a = y->sibling;
            y->sibling = z;
            y = a;
        } else {
            b = z->sibling;
            z->sibling = y;
            z = b;
        }
```

```
    }
    return H;
}

int DISPLAY(struct node* H)
{
    struct node* p;
    if (H == NULL) {
        printf("\nHEAP EMPTY");
        return 0;
    }
    printf("\nTHE ROOT NODES ARE:-\n");
    p = H;
    while (p != NULL) {
        printf("%d", p->n);
        if (p->sibling != NULL)
            printf("-->");
        p = p->sibling;
    }
    printf("\n");
}

struct node* bin_HEAP_EXTRACT_MIN(struct node* H1)
{
    int min;
    struct node* t = NULL;
    struct node* x = H1;
    struct node *Hr;
    struct node* p;
    Hr = NULL;
    if (x == NULL) {
        printf("\nNOTHING TO EXTRACT");
        return x;
    }
    //    int min=x->n;
    p = x;
    while (p->sibling != NULL) {
        if ((p->sibling)->n < min) {
            min = (p->sibling)->n;
            t = p;
            x = p->sibling;
        }
        p = p->sibling;
    }
    if (t == NULL && x->sibling == NULL)
        H1 = NULL;
    else if (t == NULL)
        H1 = x->sibling;
    else if (t->sibling == NULL)
        t = NULL;
    else
        t->sibling = x->sibling;
```

```
    if (x->child != NULL) {
        REVERT_LIST(x->child);
        (x->child)->sibling = NULL;
    }
    H = bin_HEAP_UNION(H1, Hr);
    return x;
}

int REVERT_LIST(struct node* y) {
    if (y->sibling != NULL) {
        REVERT_LIST(y->sibling);
        (y->sibling)->sibling = y;
    } else {
        Hr = y;
    }
}

struct node* FIND_NODE(struct node* H, int k)
{
    struct node* x = H;
    struct node* p = NULL;
    if (x->n == k) {
        p = x;
        return p;
    }
    if (x->child != NULL && p == NULL) {
        p = FIND_NODE(x->child, k);
    }

    if (x->sibling != NULL && p == NULL) {
        p = FIND_NODE(x->sibling, k);
    }
    return p;
}

int bin_HEAP_DECREASE_KEY(struct node* H, int i, int k)
{
    int temp;
    struct node* p;
    struct node* y;
    struct node* z;
    p = FIND_NODE(H, i);
    if (p == NULL) {
        printf("\nINVALID CHOICE OF KEY TO BE REDUCED");
        return 0;
    }
    if (k > p->n) {
        printf("\nSORY!THE NEW KEY IS GREATER THAN CURRENT ONE");
        return 0;
    }
    p->n = k;
    y = p;
```

```
    z = p->parent;
    while (z != NULL && y->n < z->n) {
        temp = y->n;
        y->n = z->n;
        z->n = temp;
        y = z;
        z = z->parent;
    }
    printf("\nKEY REDUCED SUCCESSFULLY!");
}

int bin_HEAP_DELETE(struct node* H, int k)
{
    struct node* np;
    if (H == NULL) {
        printf("\nHEAP EMPTY");
        return 0;
    }

    bin_HEAP_DECREASE_KEY(H, k, -1000);
    np = bin_HEAP_EXTRACT_MIN(H);
    if (np != NULL)
        printf("\nNODE DELETED SUCCESSFULLY");
}

int main()
{
    int i, n, m, l;
    struct node* p;
    struct node* np;
    char ch;
    printf("\nENTER THE NUMBER OF ELEMENTS:");
    scanf("%d", &n);
    printf("\nENTER THE ELEMENTS:\n");
    for (i = 1; i <= n; i++)
    {
        scanf("%d", &m);
        np = CREATE_NODE(m);
        H = bin_HEAP_INSERT(H, np);
    }
    DISPLAY(H);
    do
    {
         printf("MENU:-\n");
         printf("1.INSERT AN ELEMENT\n");
         printf("2.EXTRACT THE MINIMUM KEY NODE\n");
         printf("3.DECREASE A NODE KEY\n");
         printf("4.DELETE A NODE\n");
         printf("5.QUIT\n");
        scanf("%d", &l);
        switch (l)
            {
```

```
case 1:
   do {
      printf("\nENTER THE ELEMENT TO BE INSERTED:");
      scanf("%d", &m);
      p = CREATE_NODE(m);
      H = bin_HEAP_INSERT(H, p);
      printf("\nNOW THE HEAP IS:\n");
      DISPLAY(H);
      printf("\nINSERT MORE(y/Y)= \n");
      fflush(stdin);
      scanf("%c", &ch);
   } while (ch == 'Y' || ch == 'y');
   break;
case 2:
   do {
      printf("\nEXTRACTING THE MINIMUM KEY NODE");
      p = bin_HEAP_EXTRACT_MIN(H);
      if (p != NULL)
         printf("\nTHE EXTRACTED NODE IS %d", p->n);
      printf("\nNOW THE HEAP IS:\n");
      DISPLAY(H);
      printf("\nEXTRACT MORE(y/Y)\n");
      fflush(stdin);
      scanf("%c", &ch);
   } while (ch == 'Y' || ch == 'y');
   break;
case 3:
   do {
      printf("\nENTER THE KEY OF THE NODE TO BE DECREASED:");
      scanf("%d", &m);
      printf("\nENTER THE NEW KEY : ");
      scanf("%d", &l);
      bin_HEAP_DECREASE_KEY(H, m, l);
      printf("\nNOW THE HEAP IS:\n");
      DISPLAY(H);
      printf("\nDECREASE MORE(y/Y)\n");
      fflush(stdin);
      scanf("%c", &ch);
   } while (ch == 'Y' || ch == 'y');
   break;
case 4:
   do {
      printf("\nENTER THE KEY TO BE DELETED: ");
      scanf("%d", &m);
      bin_HEAP_DELETE(H, m);
      printf("\nDELETE MORE(y/Y)\n");
      fflush(stdin);
      scanf("%c", &ch);
   } while (ch == 'y' || ch == 'Y');
   break;
case 5:
   printf("\nTHANK U SIR\n");
```

```
            break;
        default:
            printf("\nINVALID ENTRY...TRY AGAIN....\n");
        }
    } while (l != 5);
}

/*--------------------OUTPUT-----------------------------
ENTER THE NUMBER OF ELEMENTS:5
ENTER THE ELEMENTS:12 23 34 45 56

THE ROOT NODES ARE:-
56-->12

MENU:-
 1)INSERT AN ELEMENT
2)EXTRACT THE MINIMUM KEY NODE
3)DECREASE A NODE KEY
 4)DELETE A NODE
5)QUIT
1
ENTER THE ELEMENT TO BE INSERTED:67
NOW THE HEAP IS:

THE ROOT NODES ARE:-
56-->12

INSERT MORE(y/Y)= n

MENU:-
 1)INSERT AN ELEMENT
2)EXTRACT THE MINIMUM KEY NODE
3)DECREASE A NODE KEY
 4)DELETE A NODE
5)QUIT

ENTER THE ELEMENT TO BE DELETED:67
NOW THE HEAP IS:

THE ROOT NODES ARE:-
67-->56-->12

DELETE MORE(y/Y)= n

MENU:-
 1)INSERT AN ELEMENT
2)EXTRACT THE MINIMUM KEY NODE
3)DECREASE A NODE KEY
 4)DELETE A NODE
5)QUIT
 ------------------------------------------------*/
```