UNIT-3

BINARY HEAP

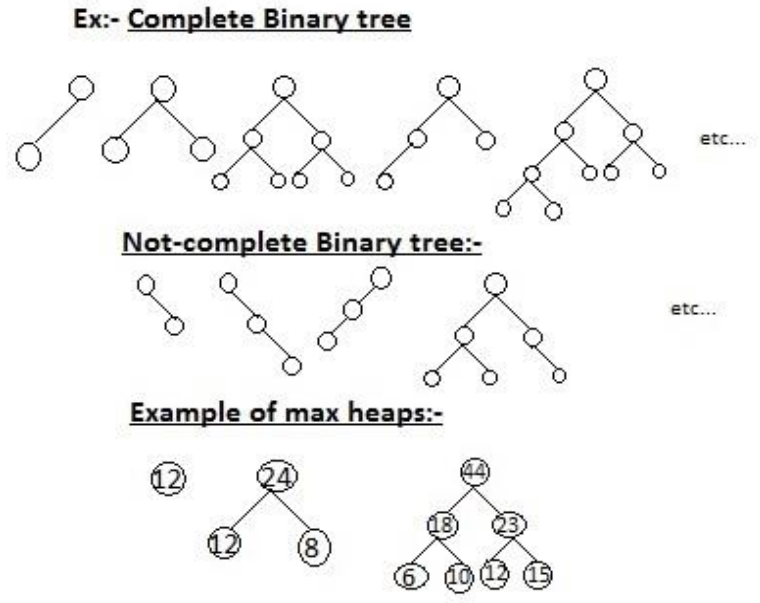
DEFINITION: A heap is a binary tree structure with the following properties

1. The tree is a complete binary tree (**Structuring Property**).
2. The key value of each node is greater than or equal to the key value in each of its descendents for a min heap (or) the key value of each node is less than or equal to the key value in each of its descendents for min heap (**Ordering Property**).

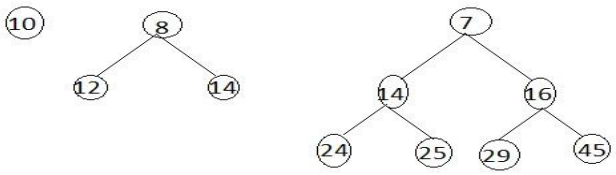
COMPLETE BINARY TREE

It is a Binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right.

Example of Min heaps**:** Here the key value of each node is less than or equal to the key value in each of its descendants.



Examples of Min Heap



HEAP IMPLEMENTATION

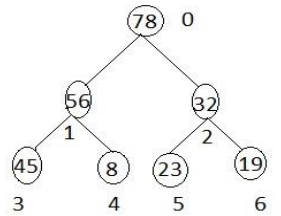
Generally a complete binary tree is so regular, so it can be represented in an array and no pointers are necessary. The relation between a node and its children is fixed and can be calculated as shown below:

(1) For a node located at index 'i' its children are found at
  a) Left child: 2i+1 b) Right child: 2i+2
(2) The parent of a node located at index 'i' is located at (i-1)/2.

(3) Given the index for a left child 'j', its right sibling is found at j+1. Given the index for a right child 'j' its left sibling is found at j-1.

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

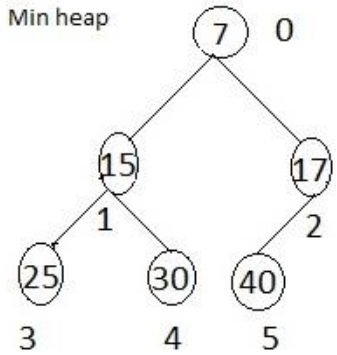
Example: Max Heap



Array representation of the above tree

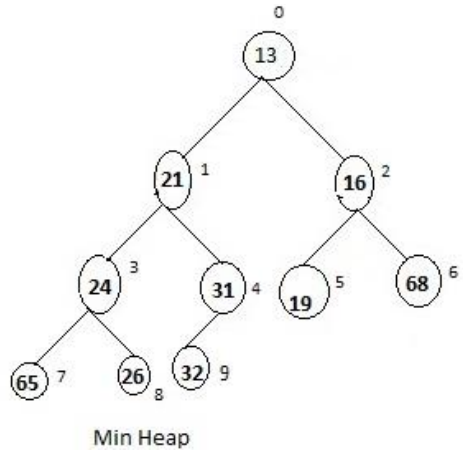
0	1	2	3	4	5	6
78	56	32	45	8	23	19

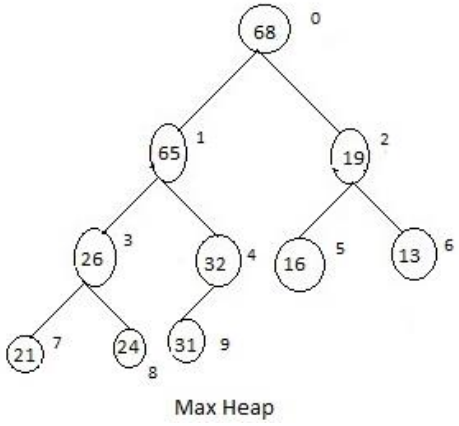
i.e., the index of 32 is 2, so the index of its left child 23 is at 2*2+1 the right child 19 is at 2*2+2(6).



0	1	2	3	4	5	
7	15	17	25	30	40	

TWO COMPLETE BINARY TREES





BASIC HEAP OPERATIONS

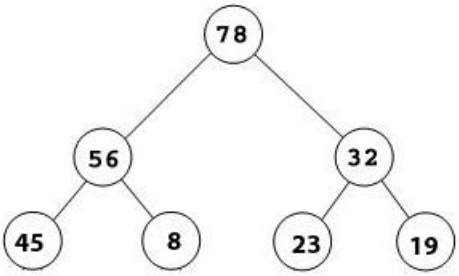
Two basic operations are performed on a heap. They are:

1) Inserting a node into Binary heap. 2) Deleting a node from Binary heap.

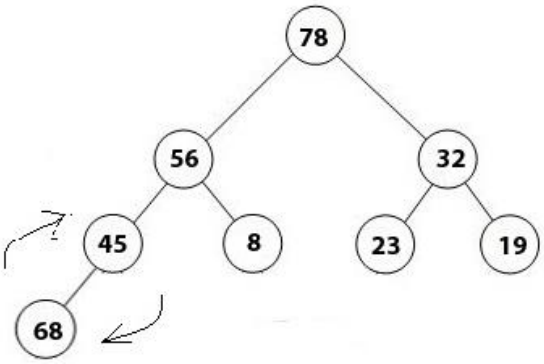
INSERT OPERATION

- Assume that we have a complete binary tree with 'N' elements whose elements satisfy the ordering property of heap.
- All these elements are stored from 0 to N-1 locations of array.
- Insert a new element 'X' at the last location. If the binary heap property is maintained then there is no need to change the position of any element.
- But if the heap property is violated then swap 'X' and its parent. Swap 'X' until the correct location of 'X' is found. Once the correct location is found then place 'X' such that the heap order property is maintained.
- This general strategy is known as a percolate up, i.e., the new element is percolated up the heap until the correct location is found.

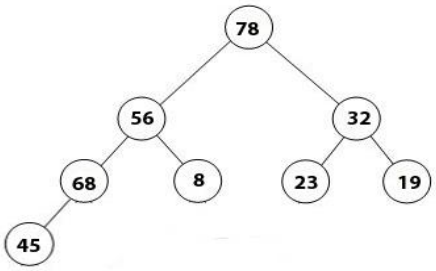
Example: Insertion into max heap



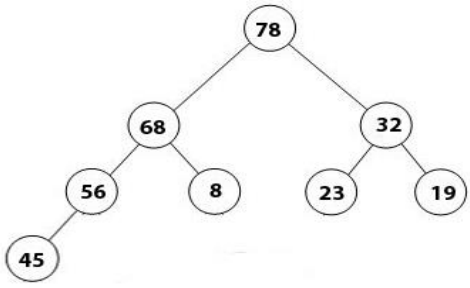
INSERT 68



Insert 68 at the last location. But by inserting '68' the heap ordered property is violated .so swap '68','45'.



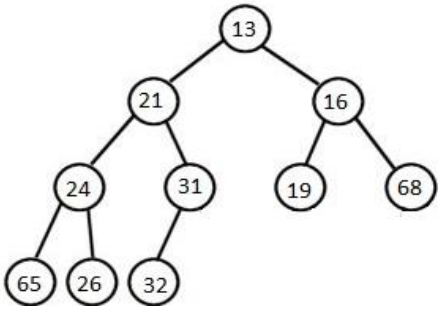
68>56. In max heap the parent should be greater. So swap 68 & 56.



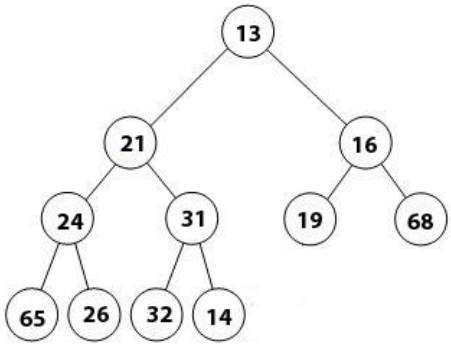
Now this is Heap.

Example: Insertion into min Heap

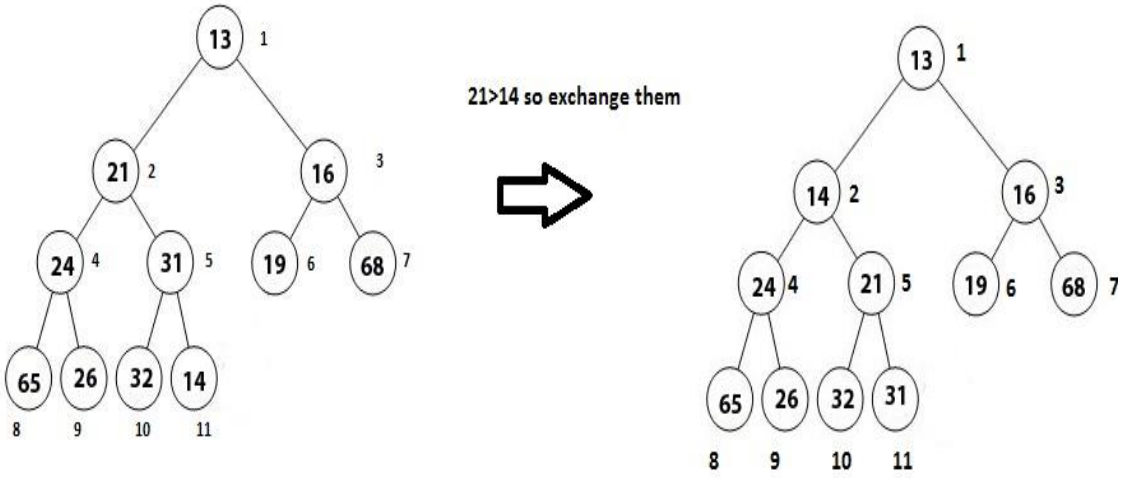
In min Heap every element at parent is less than its children.



Insert 14



Now this is not heap, because 31 > 14. So exchange 31 & 14.



This is a min heap

Note: It we store heap into array from location 1 to n instead of 0 to n-1 then for a node 'i' its left child is in position 2i, the right child is in position 2i+1 and parent of 'i' is available at [i/2].

Algorithm for inserting an element into min heap

```
void insert(x, a)
{
  if (n=max)
  {   print:"Heap is full"
      return;
  }
  for (i=n;a[i/2]>x;i=i/2)
    a[i]=a[i/2];
  a[i]=x;
}
```

Consider the above example of inserting 14

14 was inserted at position '11'.

for(i=11;a[11/2]>x;)
    a[5]=31, so
    a[11]=a[5]
    i.e., a[11]=31

now i=i/2=11/2=5

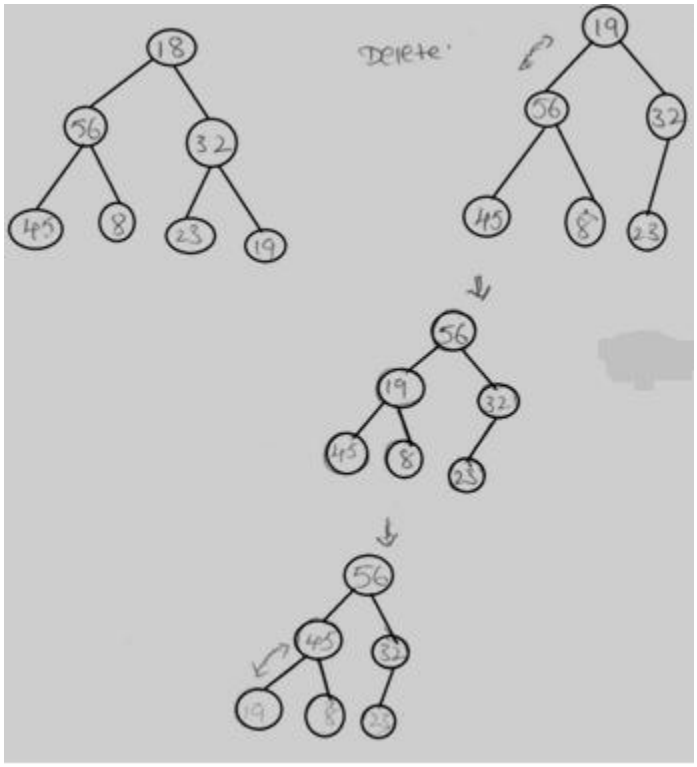
for(; a[5/2]>x;)
    a[2]>x
    21>14 true, so
    a[5]=a[5/2]
    a[5]=21
    i=5/2=2

a[1]>14 is false so control come out of for loop & a[2]=14 is assigned.

DELETE OPERATION

- In Binary Heap, if it is max heap the maximum element is removed from heap. If it is min heap the minimum element is removed from heap.
- In max heap, maximum element is available at root.
- In min heap, minimum element is available at root.
- Delete root element (i.e., max or min) from heap and place the last element 'x' in root position.
- Now compare 'x' with its children. If 'x' is greater than any one of its children then move 'x' one level down and bring its child one level up.
- We repeat this process until 'x' is placed in correct position such that the heap property is maintained.

Example: Deletion from max heap



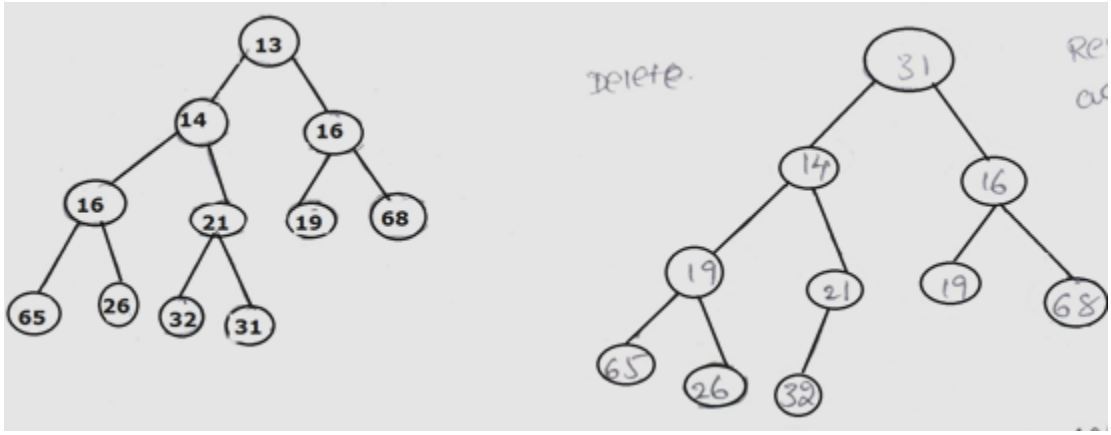
Replace root with the last element of heap.

Now this is not max heap so adjust it.

19<56 so exchange or swap them.

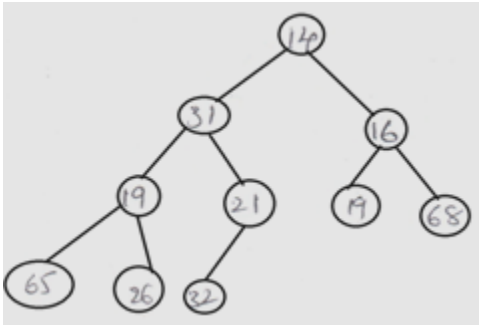
19<45 so exchange or swap them.

Example: Deletion from min heap



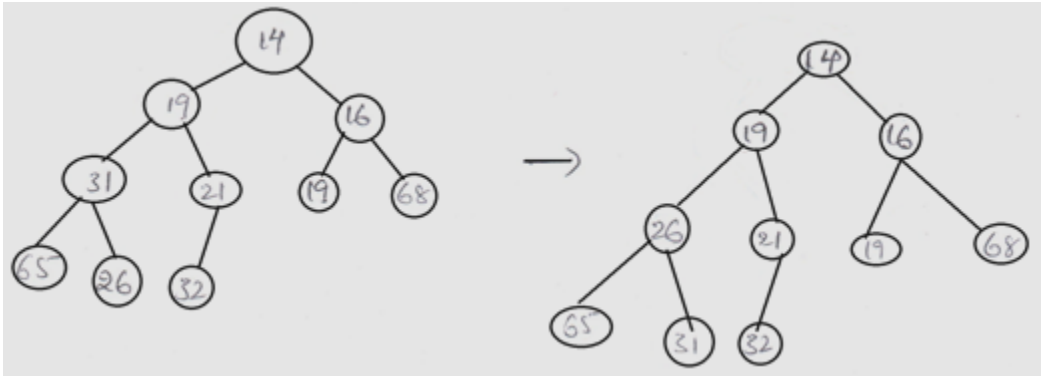
Remove 13 and place last element from heap at root

Now it is not heap, so select the least element from children of '31' and exchange 31 with that. '14' is least among 14, 16 so replace 31 with 14.



Now consider the least element among children of 31, i.e., among 19, 21. 19 is least so exchange 31 and 19.

Next consider the least element among children of 31, i.e., 65, 26. 26 is least element, so exchange 31 and 26.



Algorithm for Deleting minimum element from min heap

```
int DeleteMin(a)
{
  if(n==0)
  {
    printf:"heap is empty";
    return 0;
  }
  min=a[1];
  last=a[n];
  for(i=1;i*2<=n;i=c)
  {     c=i*2;
        if(c+1!=n  && a[c+1]<a[c])
          c++;
        if(last>a[c])
          a[i]=a[c];
        else
          break;
  }
  a[i]=last;
  return min;
}
```

BUILD A HEAP (OR) CREATE HEAP

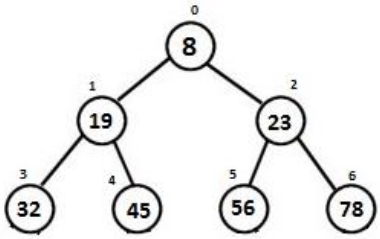
- Given a filled array of elements in random order, to build the heap we need to rearrange the data so that each node in the heap is greater than its children in max heap or each node in the heap is less than its children in min heap.
- The general algorithm is to place the N keys into the tree in any order, maintaining the structuring property.
- The percolatedown(i) algorithm percolates down from node 'i'. Perform the algorithm build heap to create a heap–ordered tree.

```
Algorithm  BuildHeap
{
  for(i=N/2;i>0;i--)
    percolateDown(i);
}
```

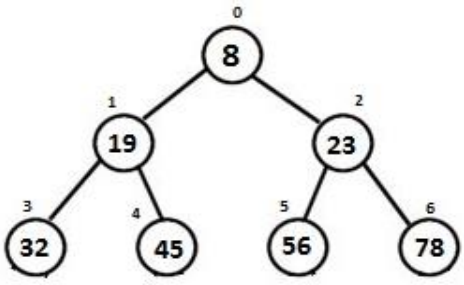
```
Algorithm percolateDown(i)
{
  for(tmp=a[i];2*i+1<n;i=c)
  {
    c=2*i+1;
    if(c+1!=n  && a[c+1]>a[c])
      c++;
    if(tmp<a[c])
      a[i]=a[c];
    else
      break;
   }
   a[i]=temp;
}
```

EXAMPLE:

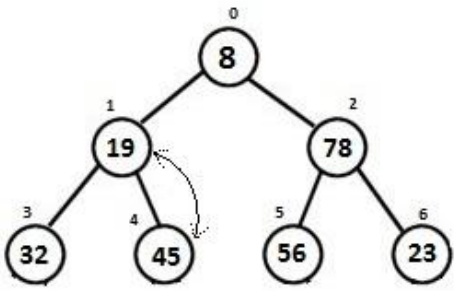
8	19	23	32	45	56	78



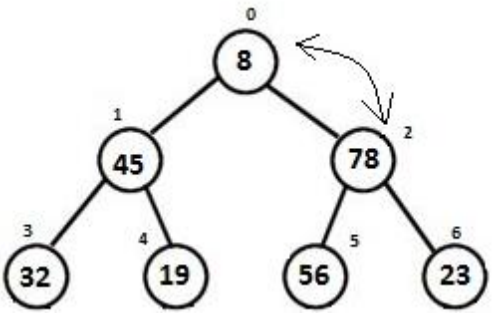
Create max Heap from these elements.



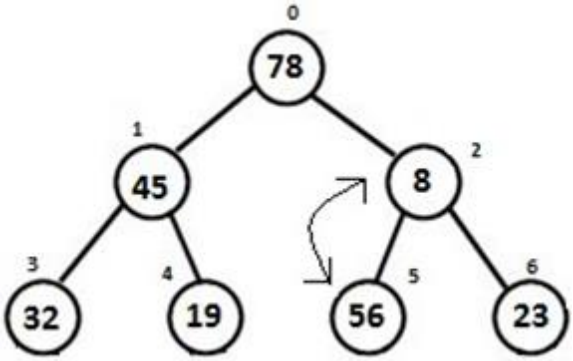
Among a[5], a[6] a[6] is large i.e., among 56,78 78 is large. Now compare a[6] with a[2]. i.e., 78 with 23. 78 is large, so swap 78 and 23.



Among a[3], a[4] a[4] is large. Now compare a[4] with a[1]. a[4] is greater so exchange with a[1].

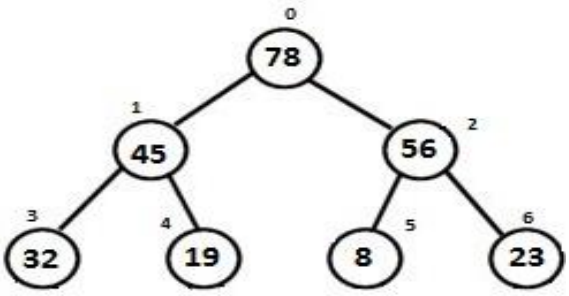


Among a[1] and a[2], a[2] is greater. Now compare a[2] with a[0]. a[2] is greater than a[0] so exchange these two.



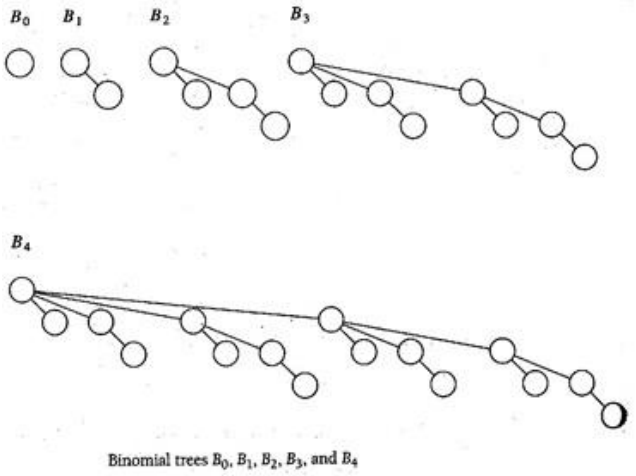
Now the root node (i.e., a[0] is in correct position).

Apply the same for every node.



BINOMIAL QUEUES

- Binomial Queue is a collection of heap-ordered Binomial trees which is also known as a forest
- Binomial trees:-
  - **a.** A Binomial tree of height 0 is a one–node tree.
  - **b.** A Binomial tree B_k of height 'k' is formed by attaching a binomial tree B_{k-1} to the root of another binomial tree B_{k-1}.



Binomial trees B_0, B_1, B_2, B_3, and B_4

B_0 is a Binomial tree of height of 0.

B_1 is formed by attaching a binomial tree **B_0** to the root of another Binomial tree **B_0**.

A Binomial tree **B_2** is formed by attaching a binomial tree **B_1** to the root of another Binomial tree **B_1**.

A Binomial tree **B_3** is formed by attaching a Binomial tree **B_2** to the root of another Binomial tree **B_2**.

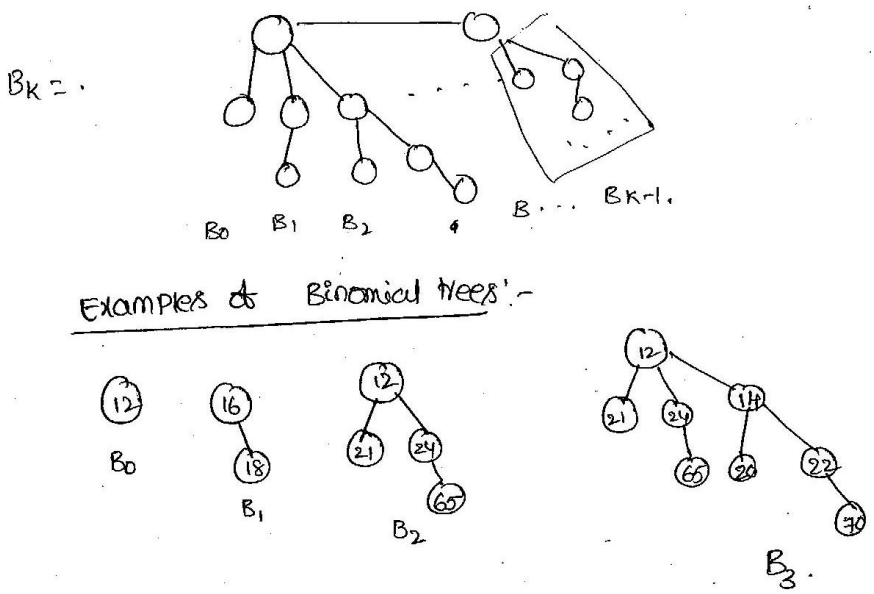
A Binomial tree **B_4** is formed by attaching a binomial tree **B_3** to the root of another Binomial tree **B_3**.

- A Binomial Queue contains at most one binomial tree of every height.
- A Binomial tree of height 'k' has exactly 2^k nodes.
  Example: A binomial tree of height 3 have exactly 2^3=8 nodes.
- The no. of nodes at depth 'd' is the binomial coefficient $\binom{k}{d}$
- The no. of nodes at depth '0' of binomial tree of height '4' is

  $\binom{4}{0}$ **=4!/0!(4-0)!= 1 node**

  At depth 1 there are $\binom{4}{1}$ nodes i.e., $\binom{4}{1}$ =4!/1!(4-1)!=4!/3!=$\frac{1x2x3x4}{1x2x3}$ =4 nodes.

  At depth 2 there are $\binom{4}{2}$ nodes i.e. $\binom{4}{2}$ =4!/2!(4-2)!=4!/(2!2!)= 6 nodes.
- A Binomial tree B_k, consists of a root with children as B_0, B_1, B_2…….. B_{k-1}.
  Ex: A Binomial tree **B_4** consists of a root with children as B_0, B_1, B_2, B_3.



Note: The degree and depth of a binomial tree with at most n nodes is at most log(n). Each tree "doubles" the previous.

BINOMIAL QUEUE OPERATIONS

Various operations that can be performed on Binomial Queue are:

1. Merging of two Binomial Queues. (Union or Meld)
2. Insertion of an element into a Binomial Queue.
3. Deletion of an element from a Binomial Queue.
4. Find Minimum element from Binomial Queue.

Merging of Two Binomial Queues

The merge is performed by essentially adding the two queues together.

Procedure

Input: Two Binomial Queue H_1 and H_2.

Output: A Binomial Queue H_3 in which the merged result is stored.

1. H_3 can contain only one B_k for each k.
2. If a Binomial tree B_k of height k does not exist either in H_1 or H_2 then don't add B_k to H_3.
3. If only one of H_1 and H_2 contain a B_k add it to H_3.
4. If H1 and H_2 both contain a B_k merge them and add B_{k+1} to H_3.
5. Now if H_1, H_2 or both contain a B_{k+1} merge until there are Zero or one of them is remained.

- Merging operation is similar to Binary Addition
  - Adding bits corresponds to merging trees.

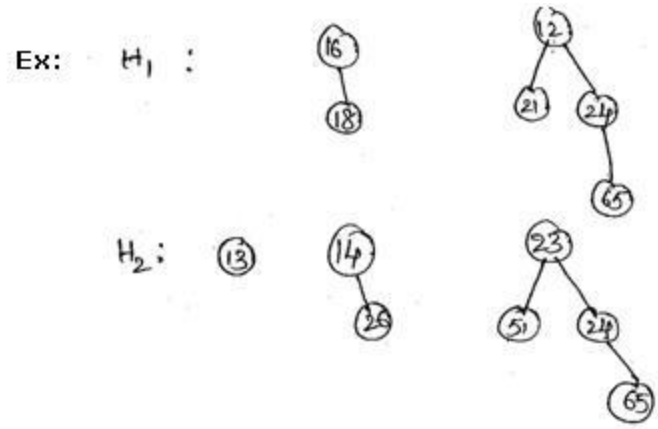
To compute value of bit 'k' for H_3

- Add bit 'k' from H_1 and H_2 and carry obtained from position k-1.
- May generate carry bit to position k+1.

Ex.: 0+0=0 i.e., if there is no B_k in H_1 or H_2 so it is not added to H_3.

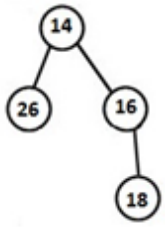
1+0=1 (or) 0+1=1 i.e., if there is only one B_k either in H_1 or H_2 then add it to H_3.

1+1=10 i.e., if there are two B_k's one in H_1 and other in H_2 then B_{k+1} is created by merging B_k and it will be considered as carry.

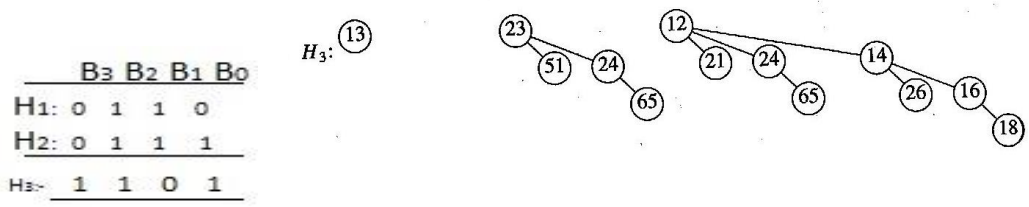


STEP 1: B_o does not exist in H_1. But it is present in H_2. So add it to H_3.

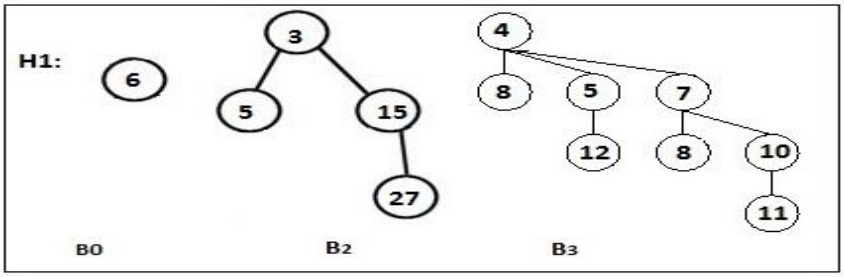
STEP 2: B_1 exists in both H_1 and H_2. So merge them, so that we will get B_2.

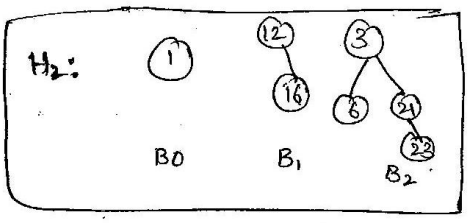


STEP 3: B_2 exists both in H_1 and H_2 and we obtained one more B_2 from step 2. Now there are total 3 B_2's. From these 3 trees we add 1 to result and other 2 are merged. Tree with root 23 is added to the result and trees with root 12 and 14 are merged.

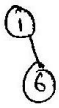


Example 2: Merge the following Binomial queues H_1 and H_2

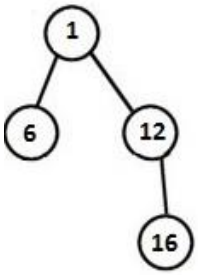




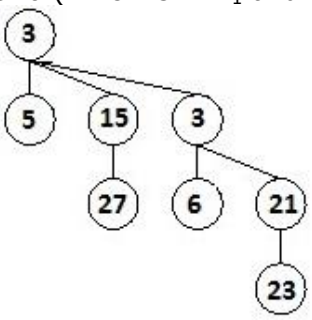
1) Two queues H_1 and H_2 both have B_0's so merge them, so that we will get B_1.



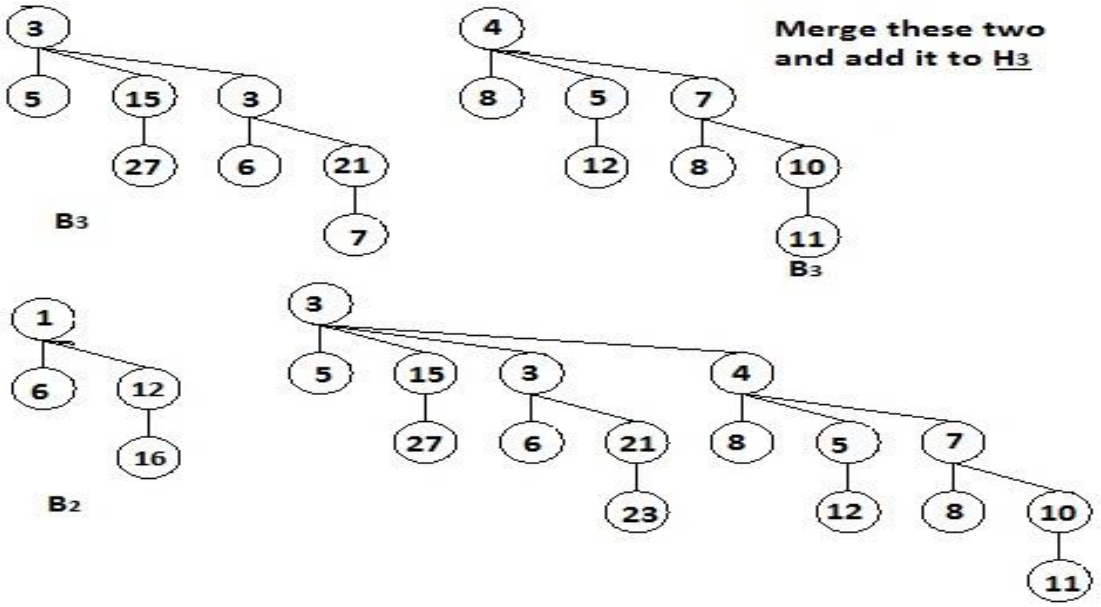
2) In H_2 one B_1 is present and in above step we obtained a B_1. So merge them.



3) Total we have 3 B_2's one in H_1 and other in H_2, and one more is obtained in previous step as carry. From these three trees add carry to merged result H_3 and remaining two (which is in H_1 and H_2) merge.



4) In H_1, B_3 is present and in above step one more B_3 is obtained, so merge the two B_3's to get a B_4

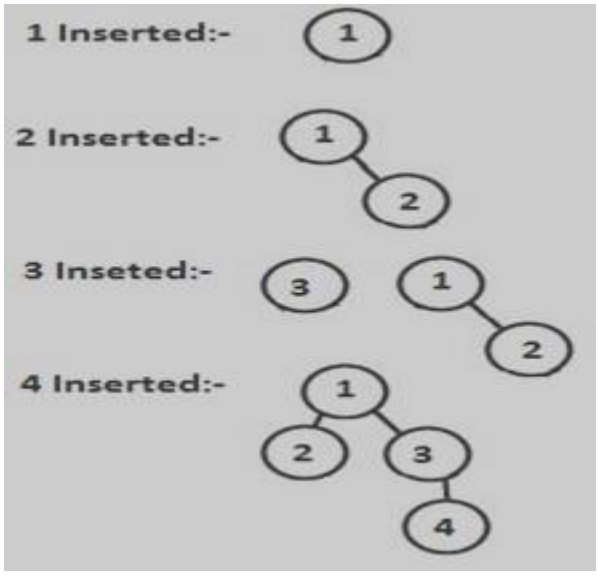


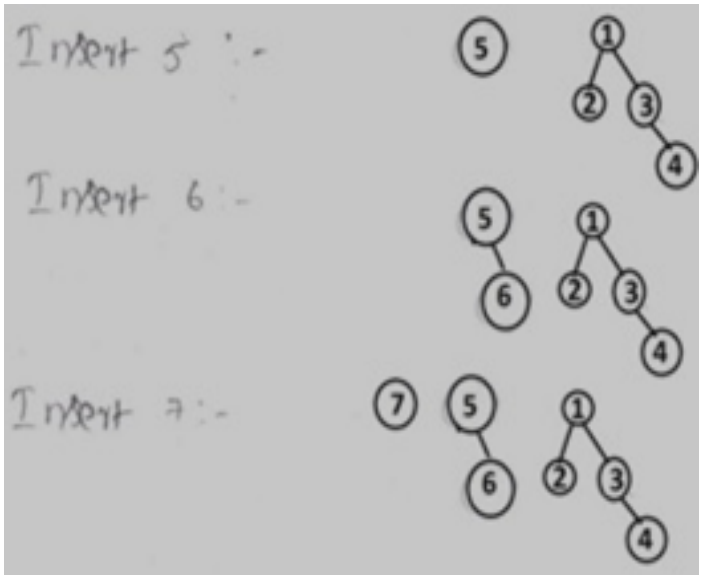
- Merging two binomial trees takes constant time.
- There are O(log n) Binomial trees, so the merge takes O(log n) time in worst case.

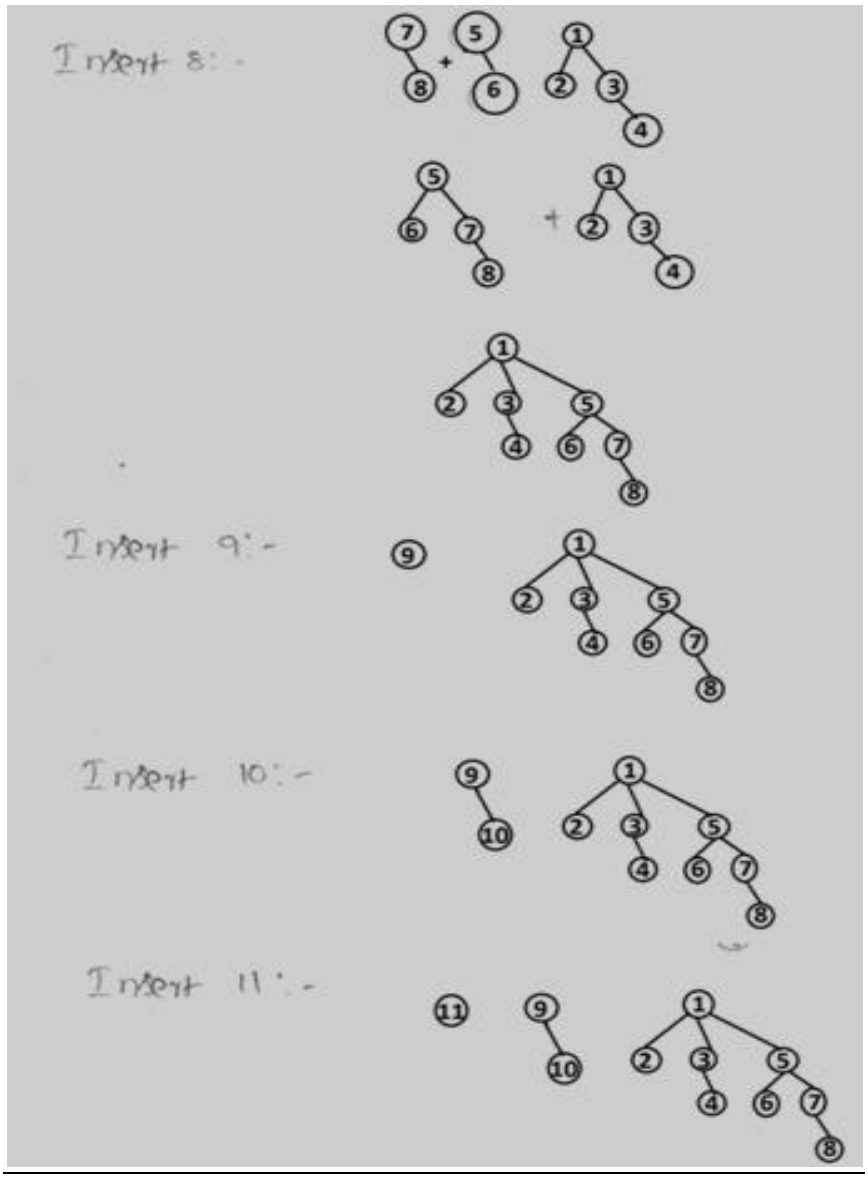
INSERTION:

- Insertion is a special case of merging, since we merely create a one–node tree and perform a merge.
- The worst case time complexity of this operation is O(log n).

➢ Create a binomial heap by inserting all the elements from 1 to 11.







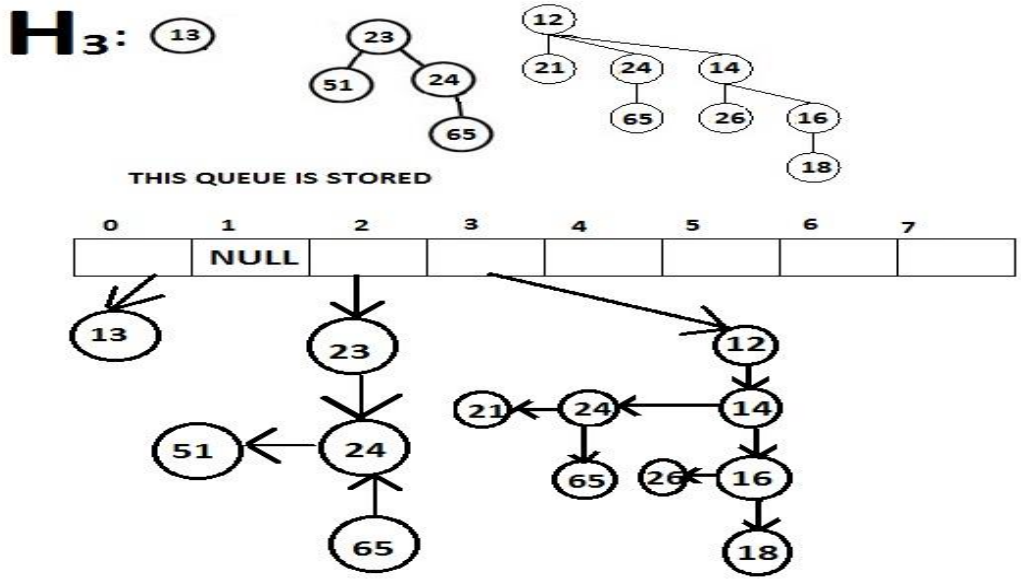
IMPLEMENTATION OF BINOMIAL QUEUES

A binomial queue can be implemented as array of linked lists. Each node in a tree is represented as a linked list with 3 fields (i.e., data field, child field, sibling field).

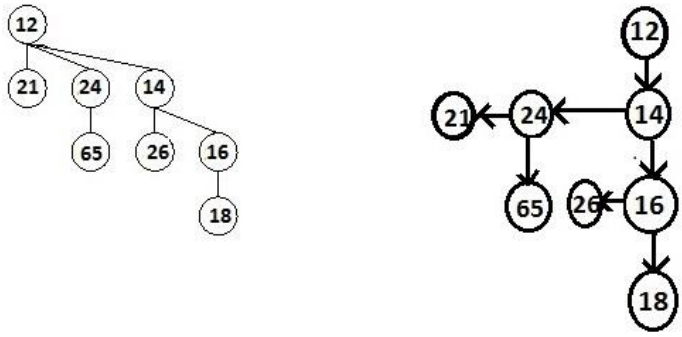
```
struct Binnode
{
    int ele;
    struct Binnode  *child;
    struct Binnode  *sibling;
};
struct collection
{
    int currentsize;
    Bintree  Tree[MAXTREE];
};

typedef struct Binnode  *positon;
typedef struct collection *BinQueue;
typedef struct Binnode  *BinTree;
```

- Binomial Queue is an array Tree [MAXTREE], which stores the addresses of roots of every Binomial tree in the Binomial Queue.
  Example:

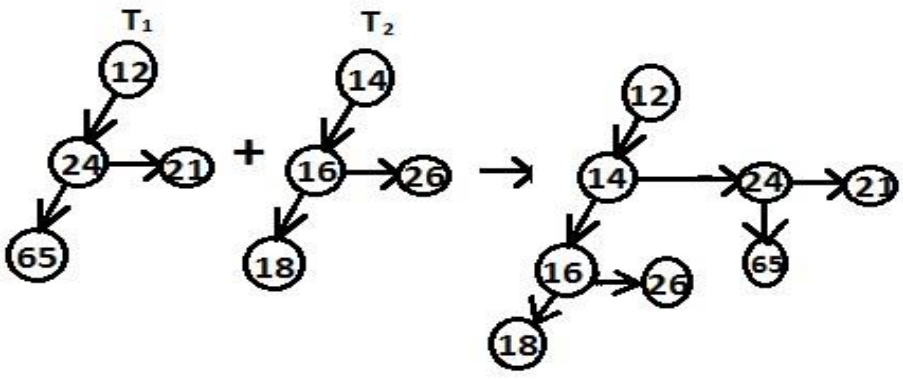


Every Binomial tree in queue is represented as follows:



From the **3rd** cell children of '12' it considers only the 14 (which has more height) as its child and '24' is considered as 14's sibling. '21' is considered as 24's sibling.

Merging Two Binomial trees of same height:



Among T_1 and T_2 select the minimum as root and other as child for root. If the root has already a child make this as the sibling to the new child i.e., among 12, 14 12 is minimum so make it as root, '14' as child of '12'. The previous child of 12 (i.e., 24) make as sibling to '14'.

Routine to Merge 2 Binomial Trees of equal size

```
BinTree combineTrees(BinTree T1,BinTree T2)
{       if(T1->ele > T2->ele)
          return combineTrees (T2,T1);
        T2->sibling=T1->child;
        T1->child=T2;
        return T1;
}
```

Algorithm to Merge two Binomial Queues or Priority Queues

- The following algorithm combines two Binomial Queues H_1 and H_2.
- It places the result in H_1 and making H_2 Empty.
- T_1 and T_2 are the trees in H_1 and H_2 respectively and carry is the tree carried from previous step.
- **!!** T_1, is 1 if T_1 exists and is 0 otherwise.
  !! T_2, is 1 if T_2 exists and is 0 otherwise.
  !! carry, is 1 if **carry** exists and is 0 otherwise.
- Depending on each of eight possible cases, the tree that results for rank 'i' and the 'carry' tree of rank i+1 is formed.
- This process proceeds from rank 0 to the last rank in the resulting Binomial Queue.
  case 0:- Executed when T_1, T_2 is NULL, carry also NULL
  case 1:- Executed when we have only T_1 (i.e., only H_1)
  case 2:- Executed when we have only T_2 (i.e., only H_2)
  case 3:- Executed when we have both trees T_1, T_2 so both will be merged. We will get a carry (B_K is disappeared and B_{k+1} is formed)
  case 4:- When we have only carry.
  case 5:- We have Both H_1 and carry.
  case 6:- We have Both H_2 and carry.
  case 7:- All three i.e., H_1, H_2 and carry.

ROUTINE TO MERGE TWO BINOMIAL QUEUES OR PRIORITY QUEUES

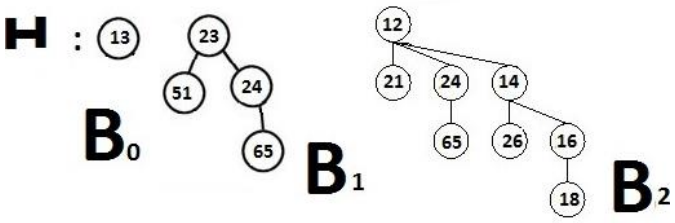
```
BinQueue Merge(BinQueue H1 , BinQueue H2)
{
   BinTree T1,T2,carry=NULL;
   int i,j;
   if (H1->currentsize+H2->currentsize > capacity)
     print "Merge would exceed capacity";
   H1->currentsize = H1->currentsize+H2->currentsize;
for(i=0,j=1;j<=H1->currentsize;i++,j=j*2)
{  T1=H1->Tree[i];
   T2=H2->Tree[i];
   switch(!!T1+2*!!T2+4*!!carry)
   {       case 0:
           case 1:break;
           case 2:H1->Tree[i]=T2;
                  H2->Tree[i]=NULL;
                  break;
           case 4:H1->Tree[i]=carry;
                  carry=NULL;
                  break;
```

```
            case 3:carry=combineTrees(T1,T2);
                        H1->Tree[i]=NULL;
                        H2->Tree[i]=NULL;
                        break;
            case 5:carry=combineTrees(T1,carry);
                        H1->Tree[i]=NULL;
                        break;
            case 6:carry=combineTrees(T2,carry);
                        H2->Tree[i]=NULL;
                        break;
            case 7:H1->Tree[i]=carry;
                        carry=combineTrees(T1,T2);
                        H2->Tree[i]=NULL;
                        break;
            }
    }
   return H1;
}
```

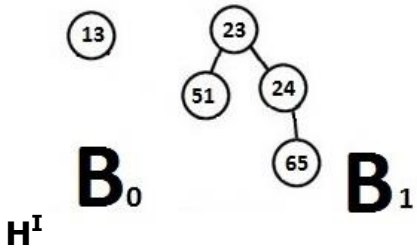
DELETION OF AN ELEMENT FROM BINOMIAL QUEUE

- In deletion operation the smallest element is deleted from Binomial queue.
- Deletion is performed by first finding the Binomial tree with the smallest root.
- Let the tree with smallest root be B_k, and let the original priority queue be H.
- Form a new Binomial queue H^1, by removing the binomial tree B_k from the forest of trees in H.
- Form a new Binomial queue H^{II}, by removing the root of B_k creating binomial trees $B_0, B_1, B_2, B_3, \ldots B_{k-1}$.
- Now merge H^1 and H^{II}. This is the final result of deleting minimum element from min heap binomial queues.

Example:

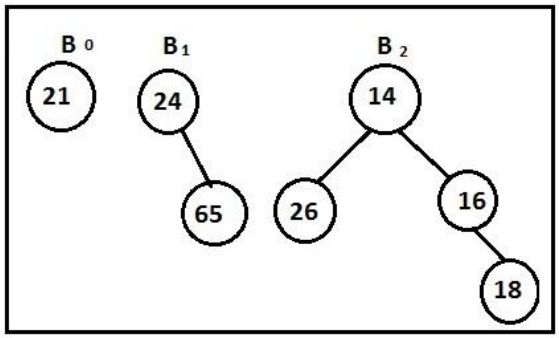


Apply Delete operation to H.
Step 1) Find the tree with smallest root i.e., 12.
Step 2) Create H^I By removing the Binomial tree with root 12 from H.

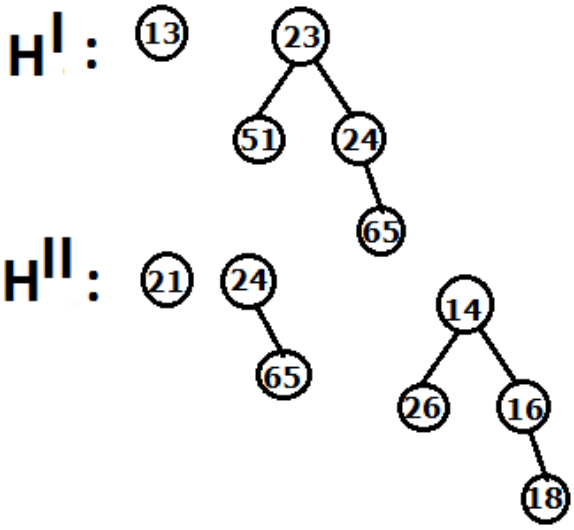


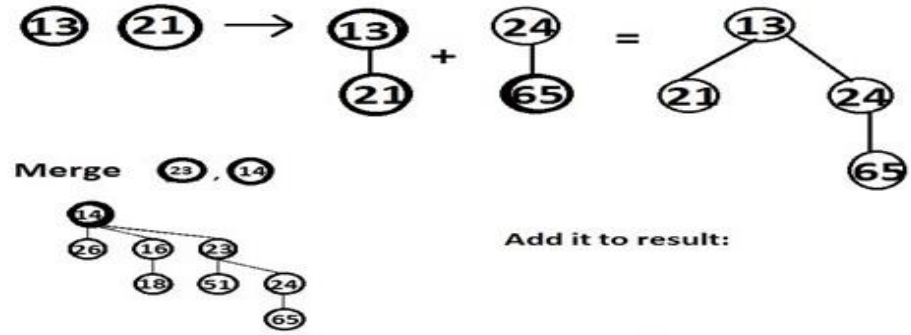
H^I

Step 3) Create H^{II} by removing the root of B_3 from H. So add B_0, B_1, B_2 to H^{II}



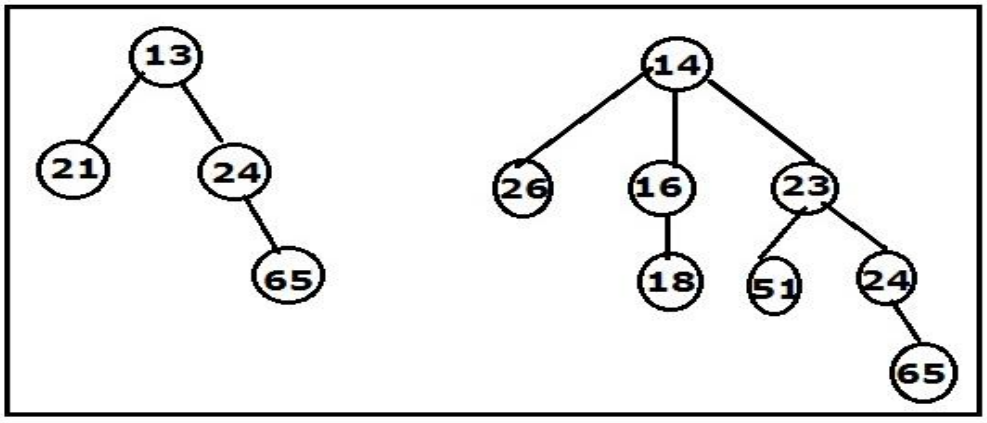
MERGE H^1 and H^{II}.





Add it to result:

The resultant Binomial Queue after deleting the minimum element:



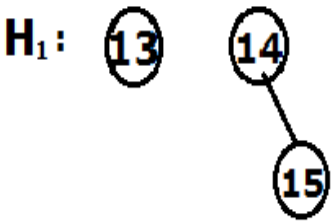
IMPLEMENTATION OF DELETION ALGORITHM

```
int DeleteMin(BinQueue  H)
{       int i,j,MinTree,MinItem;
        BinQueue  DeletedQueue;
        position DeletedTree,oldroot;
        if(Isempty(H))
        {       print:"Empty  Binomial  Queue".
                return -1;
        }
         MinItem=Infinity;
        for(i=0;i<MaxTrees;i++)
        {       if(H->Tree[i]&&  H->Tree[i]->ele<MinItem)
                {       MinItem=  H->Tree[i]->ele;
                        MinTree=i;
                }
        }
        DeletedTree=  H->Tree[MinTree];
        oldroot=DeletedTree;
        DeletedTree=DeletedTree->child;
        free(oldroot);
        DeletedQueue=Initialize();
        DeletedQueue->currentsize=(1<<MinTree)-1;
        for(j=MinTree-1;j>=0;j--)
        {
                DeletedQueue->Tree[j]=DeletedTree;
                DeletedTree=  DeletedTree->sibling;
                DeletedQueue->Tree[j]  ->sibling=NULL;
                }
        H->Tree[MinTree]=NULL;
        H->currentsize=  H->currentsize-( DeletedQueue->currentsize+1);
        Merge(H,DeletedQueue);
        return MinItem;
        }
```

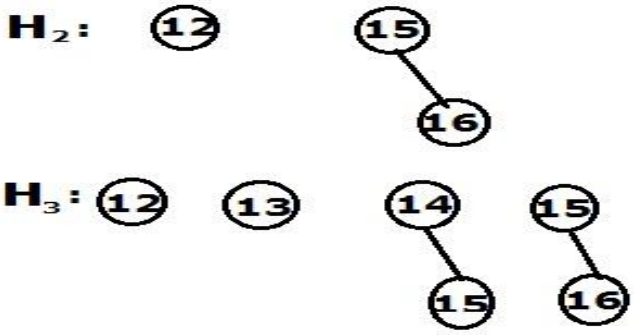
- In the above algorithm, if(Isempty(H)) statement checks whether the Binomial Queue is empty or not. If it is empty then deletion is not possible.
- for(i=0,…) loop finds the tree with minimum root.
- Next H^{II} is created by removing the root and create $B_0, B_1 \ldots . B_{k-1}$ by the for(j=1,….) loop. These are stored in DeletedQueue. (DeletedQueue->Tree[0], DeletedQueue->Tree[1],…)
- H^{I} is created by H->Tree[MinTree]=NULL; statement.
- Both H^{I} and H^{II} are merged by calling the function Merge(H,…).

LAZY Binomial Queues

- Binomial queue in which merge is done lazily.
- Here to merge two Binomial Queues, we simply concatenate the two lists of Binomial trees.
- In the resulting forest, there may be several trees of same size.
- Because of the lazy merge, merge and insert are both worst case O(1) time.



Ex:-



DeleteMin

- It converts lazy Binomial Queue into a standard Binomial Queue.
- Do DeleteMin as in standard queue.

Priority Queues Applications

1) Implementing scheduler in OS, and distributed systems.
2) Representing event lists in discrete event simulation.
3) Implementing numerous graph algorithms efficiently.
4) Selecting K^{th} largest and K^{th} smallest elements in lists.
5) Sorting applications.
6) A* Search.
7) Huffman encoding.
8) Network bandwidth management.

Time Complexity of Binomial Heap Operations

Operation	Binomial Heap	Amortized Binomial Heap
Make-Heap	O(1)	O(1)
Insert	O(log n)	O(1)
Deletemin (or max)	O(log n)	-
Merge (Union Or Meld)	O(log n)	O(log n)
Extract-Min	O(log n)	O(log n)