Priority Queue:

* Priority Queue is a collection of elements such that each element in has an associated priority.

* Here there are two types of priority Queues.
  1) Single-Ended priority Queue (SEPQ)
  2) Double-Ended priority Queue (DEPQ)

Singe Ended priority Queue

* Heaps are used to implement SEPQ (priority Queues).

In this kind of Queue the element to be deleted is the one with the highest (or) lowest priority.

* In this Queue the element to ~~be~~ ~~is the one~~ with the arbitary priority can be inserted into the Queue.

* SEPQ may further categorized as
  1) Min Priority Queues
  2) Max Priority Queues

Operations supported by a Minimum Priority Queue:

(1) Return an element with minimum priority
MinElement(); minkey();

(2) Insert an element with an arbitary priority
insert();

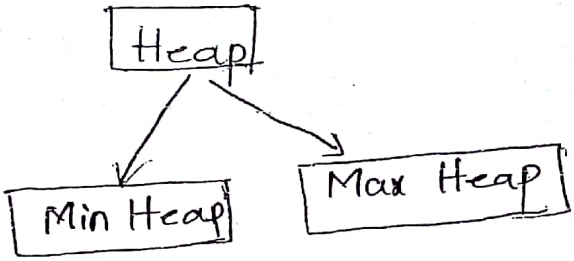
(3) Delete an element with minimum priority.
deleteMin();

Operations Supported by Max priority Queue

1. Return an element with maximum priority
2. Insert an element with an arbitary priority
3. Delete an element with maximum priority.

Heap :-

The Heap structure is a classic data structure for the representation of priority Queue.



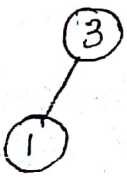
Min Heap :-

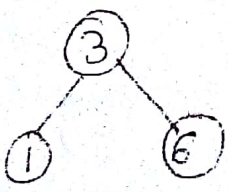
The keys of parent nodes are lessthan (or) equal to those of the children.

construction of Min Heap:

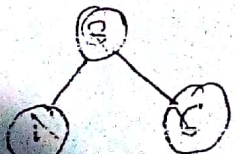
3 1 6 5 2 4 → given elements

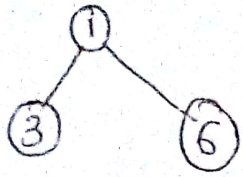
1) (3)

2) 

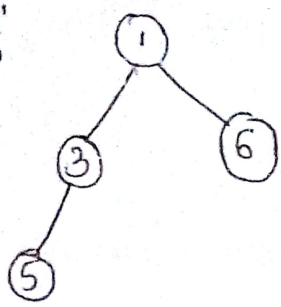
3) 

→ (3) & (1) we need to swap because root node always lessthan the children.

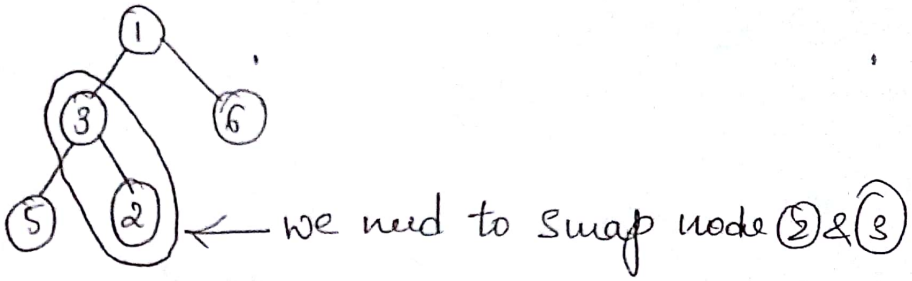
4) 

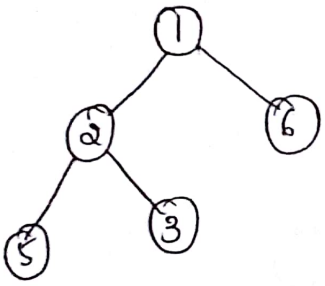


(5) insert '5'

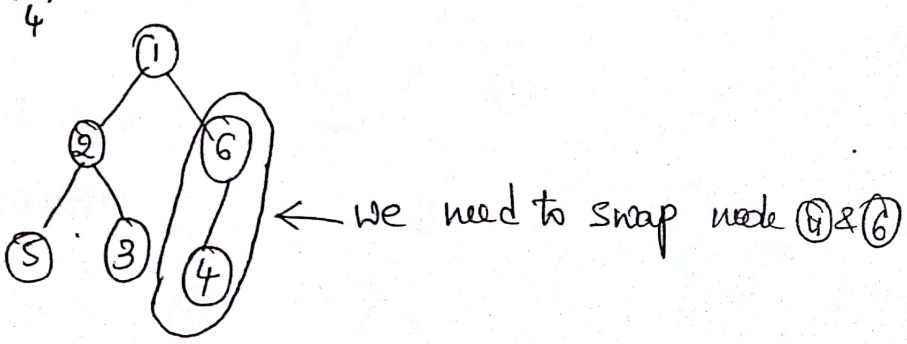


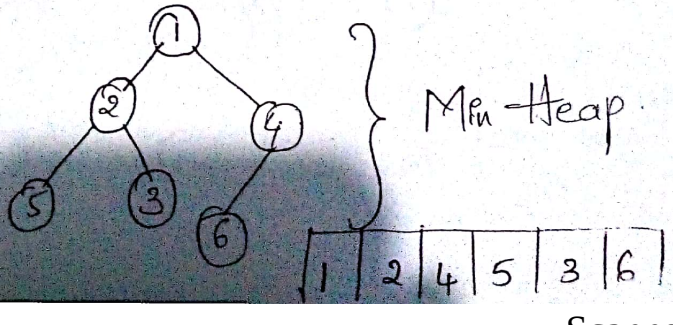
(6) insert '2'





(7) insert '4'





Max Heap:

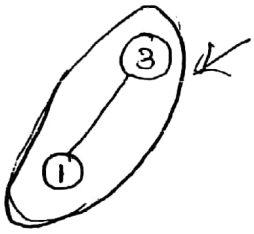
Max Heap is the keys of parent node are always greater than (or) equal to those of the children.

Construction of Max Heap:

3 1 6 5 2 4 ← given elements

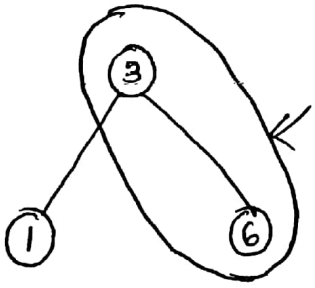
1) insert '3'

2) insert node '1'

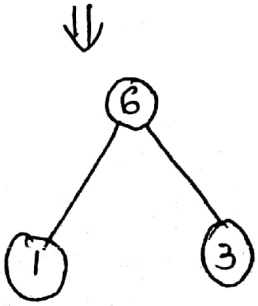


no need to apply swapping

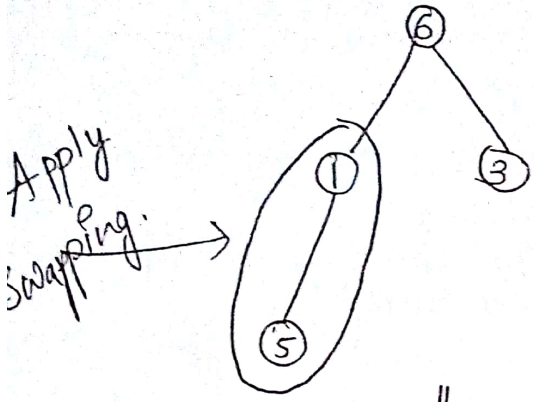
3) insert node '6'

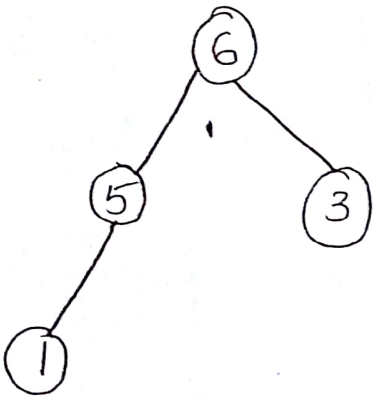


we need to apply swapping, bcz. root node always greater than the children.

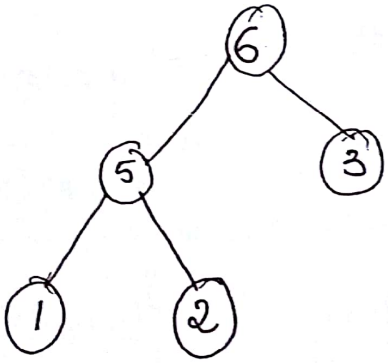


4) insert node '5'

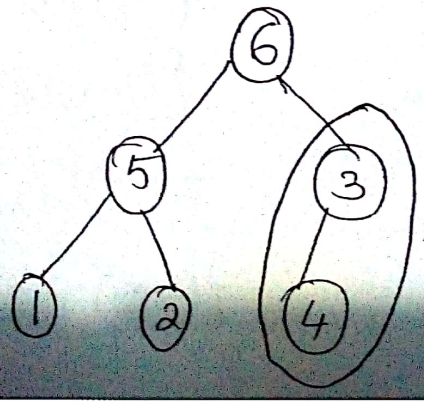


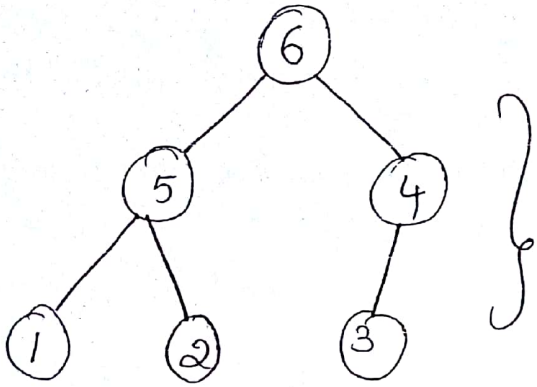


5) Insert node '2'



6) Insert node '4'.





Max Heap:

6	5	4	1	2	3

Eg:

Operation	Output	Priority Queue.
insert(5,A) (key, element)	-	{(5,A)}
insert (9,C)	-	{(5,A)} {(9,C)}
insert (3,B)	-	{(3,B), (5,A), (9,C)}
insert (7,D)	-	{(3,B), (5,A), (7,D), (9,C)}
minElement()	B	{(3,B), (5,A), (7,D), (9,C)}
minkey()	3	{(3,B), (5,A), (7,D), (9,C)}
removeMin()	-	{(5,A), (7,D), (9,C)}
size()	3	{(5,A), (7,D), (9,C)}
deleteMin()	-	{(7,D), (9,C)}
deleteMin()	-	{(9,C)}
deleteMin()	"error"	{ }
isE		{ }

Example for Min & Max priority Queue:

* Suppose that we are selling the services of a machine. Each user pays a fixed amount per use. However the time needed by user is different.

* Inorder to maximize the returns, we need to maintain priority Queue of all persons waiting to use the machine.

* Whenever the machine becomes available, the user with the smallest time requirement is selected. Hence a min priority Queue is required. When a new user requests the machine, his/her request is put into the priority Queue.

* If each user needs the same amount of time on the machine but people are willing to pay different amounts for the service, then a priority Queue based on the amount of payment can be maintained. Whenever the machine becomes available, the user paying most is selected. This requires a max priority Queue.

The following code Represents ADT of a min Priority Queue. The

```
template <class T>
class minPQ  → represented by using pure virtual function.
{
Public:
    Virtual minPQ()
```

- Virtual bool IsEmpty(): ~~const=0~~.
  ↳ Return true if the priority Queue is empty.
- Virtual const T & Top() const = 0;
  ↳ Return Reference to minimum element.
- Virtual void push(const T&) = 0;
  ↳ add the elements into priority Queue.
- Virtual void pop() = 0;
  ↳ Delete element with minimum priority.

};

Extensions of a Single Ended priority Queue.

1) * The first extension is meldable (Combine) single-ended priority queue.

* Two data structures for Meldable priority Queues are
  (1) Leftist trees ✓
  (2) Binomial Heaps ✓

2) Further Extension of meldable priority Queue include Operations to delete an arbitary element and to

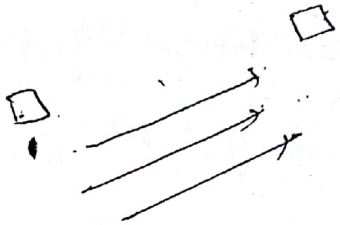
Link:

The Link data member is used to maintain simply linked circular list of siblings.

The following heap data structures are used to represent the DEPQ.

① Symmetric min-max Heaps

② Interval Heaps



Examples:

→ A DEPQ may be used to implement a network buffer. This buffer holds packets that are waiting their turn to be sent out over a network link, each packet has an associated priority.

→ When the network link becomes available, the packet with the heighest priority is transmitted. This corresponds to the DeleteMax operation.

→ When a packet arrives at the buffer from elsewhere in the network, it is added to this buffer. This is corresponds to Insert Operation.

→ If the Buffer is full, we must drop a packet with the minimum priority before we can insert one. This is corresponds to DeleteMin Operation.

incase of max priority Queue) of the element

→ Two data structures are developed for this Extensions.

(1) Fibnocci Heaps
(2) pairing Heaps

Double - Ended priority Queues (DEPQ):

. A DEPQ is a data structure that supports the following operations on a collection of elements.

(1) Return an element with minimum priority.
(2) Return an element with maximum priority
(3) Insert an element with an arbitary priority
(4) Delete an element with minimum priority
(5) Delete an element with maximum priority

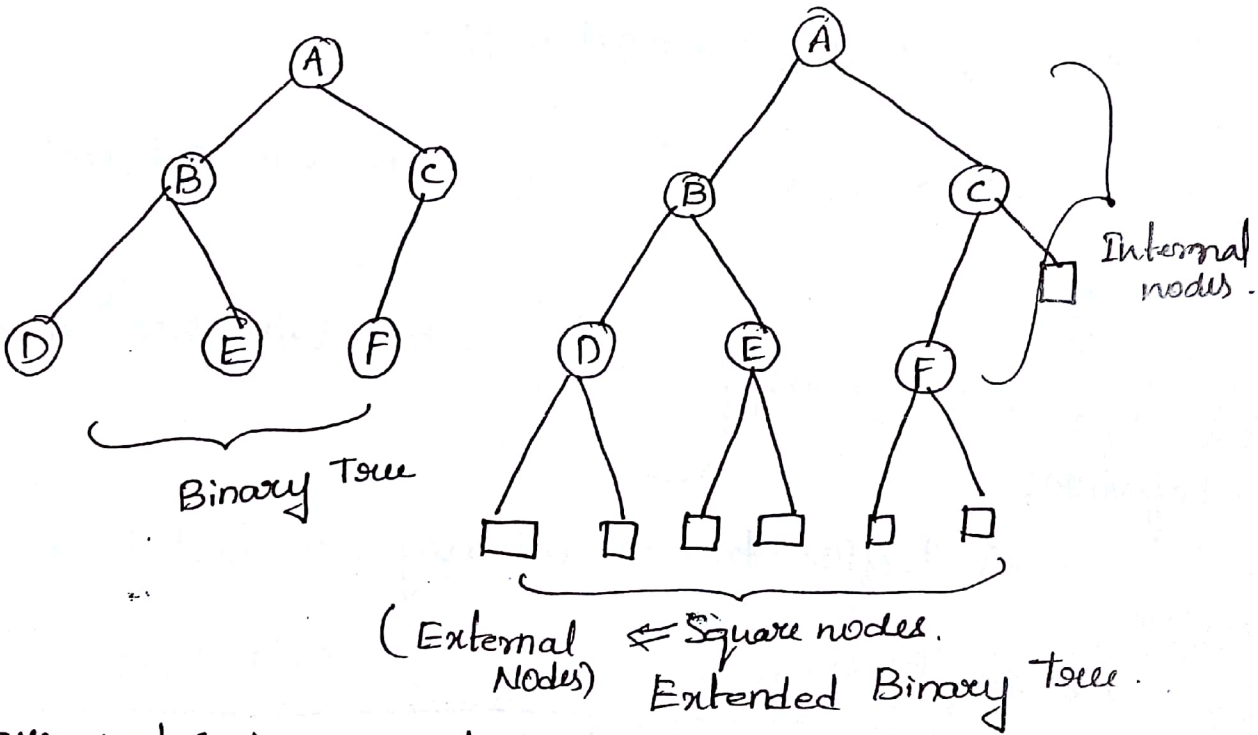
* DEPQ is a min and max priority Queue rolled into one structure.

program for double ended priority Queues.

```
template <class T>
class DEPQ
{
 Public:
virtual const T & GetMin() const=0;
virtual const T & GetMax() const=0;
virtual void Insert(const T&) =0;
virtual void DeleteMin() =0;
```

Leftist Trees :-

* Leftist trees provide an efficient implementation of meldable priority queues.
* Leftist trees are defined using the concept of an Extended binary tree.
* An Extended Binary Tree is a binary tree in which all empty binary subtrees have been replaced by a Square node.



* Square nodes in all Extended binary trees are called External nodes.
* The Original nodes of the Binary Tree are called Internal nodes.
* Leftist trees
  ↳ it is of two types
  (1) Height-Biased Leftist Trees (HBLT)
  (2) Weight-Biased Leftist Trees (WBLT)

Height-Biased Leftist Trees: (HBLT):

↳ HBLT simply called as Leftist tree.

* Let 'x' is a node in the extended binary Tree.
* Let Leftchild(x) and Rightchild(x) denote the left and right children of the internal node x.
* Find the shortest (Height) to each and every node.

↓

Shortest(x) is the length of the shortest path from 'x' to the External node.

$$\text{Shortest}(x) = \begin{cases} 0 & \text{if } x \text{ is an external node.} \\ 1 + \min\{\text{shortest}(\text{Leftchild}(x)), \text{shortest}(\text{rightchild}(x))\} \end{cases}$$

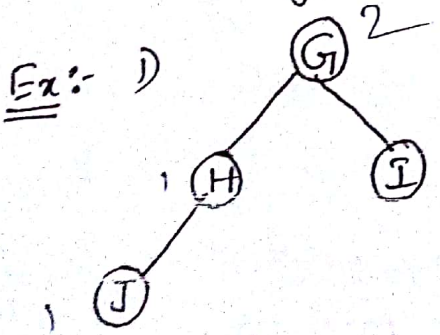
Definition:-

A Leftist tree is a binary Tree such that if it is not empty, then

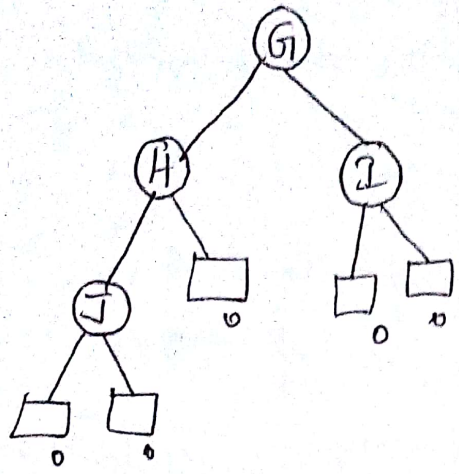
$$\boxed{\text{Shortest}(\text{Leftchild}(x)) \geq \text{shortest}(\text{rightchild}(x))}$$

for every internal node 'x'.

Ex:- 1)



2) Represent the given Binary tree in the form of



Now calculate shortest(x) $\downarrow$ node for each node.

$$\text{shortest}(J) = 1 + \min\{\text{shortest}(\text{leftchild}(J)), \text{shortest}(\text{rightchild}(J))\}$$

$$= 1 + \min\{0, 0\}$$

$$= 1 + 0$$

$$= \underline{1}$$

$$\text{shortest}(I) = 1 + \min\{\text{shortest}(\text{leftchild}(I)), \text{shortest}(\text{rightchild}(I))\}$$

$$= 1 + \min\{0, 0\}$$

$$= 1 + 0$$

$$= \underline{1}$$

$$\text{shortest}(H) = 1 + \min\{\text{shortest}(\text{leftchild}(H)), \text{shortest}(\text{rightchild}(H))\}$$

$$= 1 + \min\{1, 0\}$$

$$= 1 + 0$$

$$= \underline{1}$$

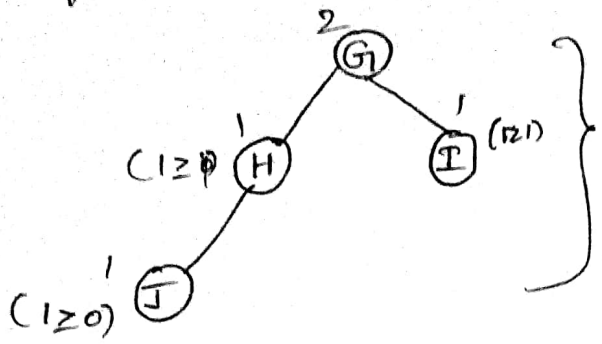
$$\text{shortest}(G) = 1 + \min\{\text{shortest}(\text{leftchild}(G)), \text{shortest}(\text{rightchild}(G))\}$$

$$= 1 + \min\{1, 1\}$$

$$= 1 + 1$$

$$= \underline{2}$$

4) After calculating Heights the tree becomes.



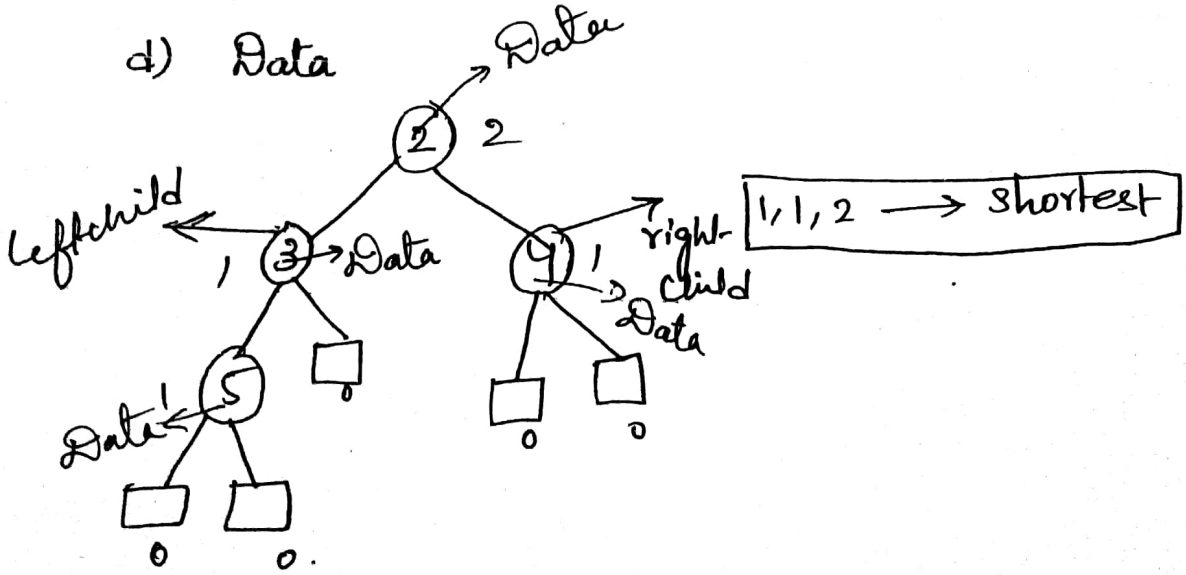
the tree is Leftist tree because

Shortest(Leftchild(x)) ≥ Shortest(rightchild(x))

Representation :-

Leftist trees are represented using following data members

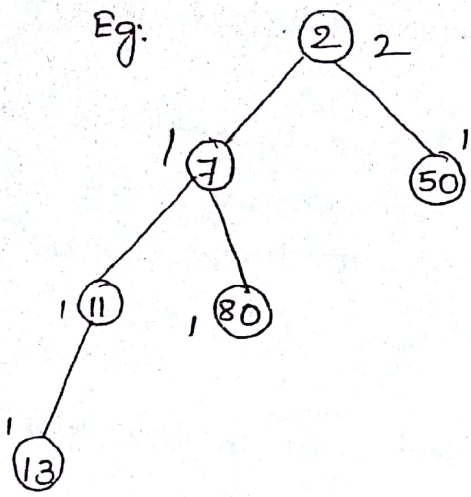
a) Leftchild
b) Rightchild
c) shortest
d) Data

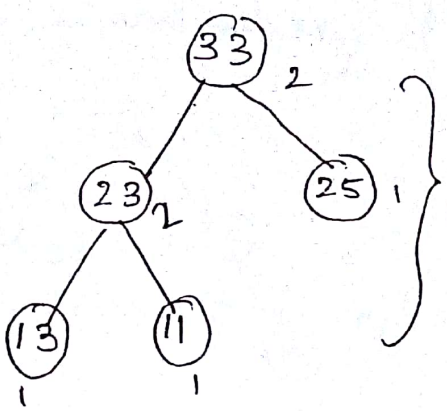


Definition for Minimum Leftist Tree:

* A min Leftist Tree is a leftist tree in which the key value in each node is smaller than the key value of its children. (i.e like Minimum Tree).

Eg:





Max-Leftist tree.

(8)

Definition for Max Leftist Tree:

A Max Leftist tree is a Leftist tree in which the key value in each node is greater than the value of its children. (i,e like Max tree).

Operations on Min Leftist Tree:

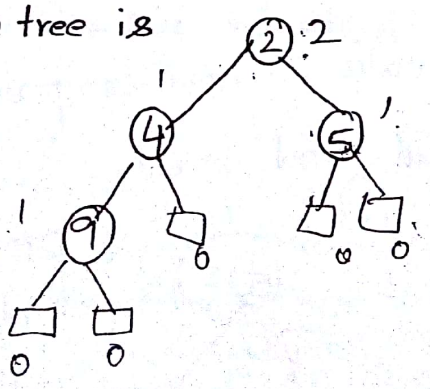
1) Insert
2) Delete Min
3) Meld

* Insert and delete-min operations both can be performed using the meld operation.

Insert:

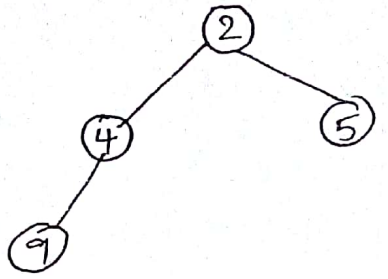
Suppose we have some elements 20, 35. insert into Leftist tree.

1) Given tree is



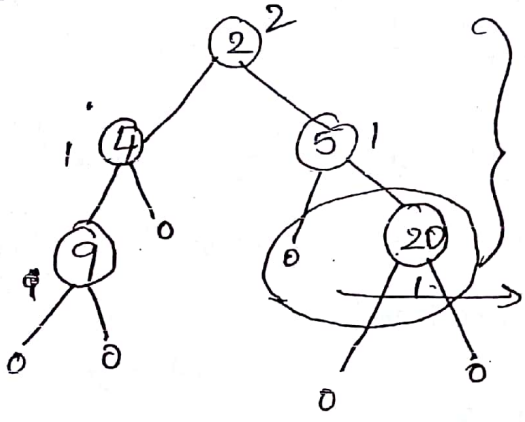
insert "20" (compare "2" with (root node) with "5".) (20 < 50)

* 20 is greater than "2" so, move to right part.



Now compare 5 with because in the right we have element called

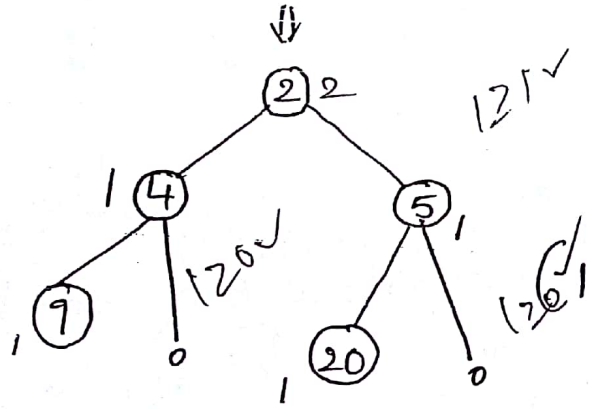
* 20 is greater than "5" insert "20" in the right child Pos



calculate shortest for every node.

Leftist tree property violated.

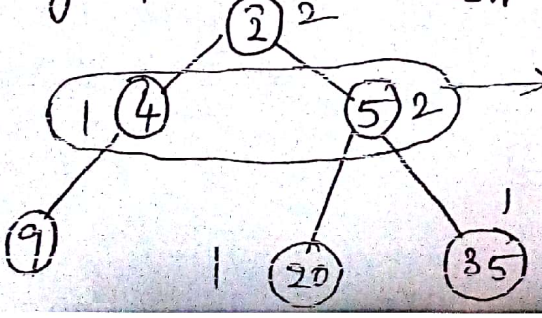
so, 20 becomes Leftchild of 5.



* Now insert "35", first compare with root node (2<35, so move to right part,

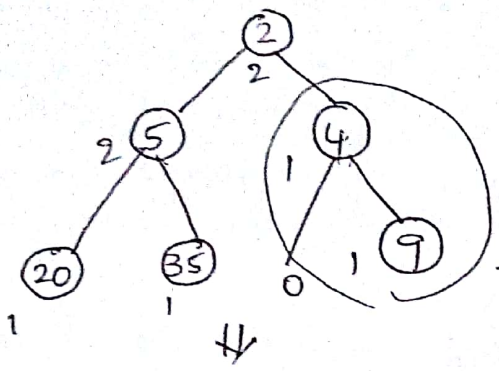
* Again we have node "5" now compare with node 5 (5<=

So move to right part. and insert.

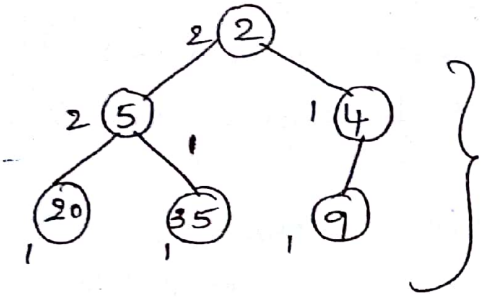


Property violated.

Now, swap Left Sub tre and right subtree.



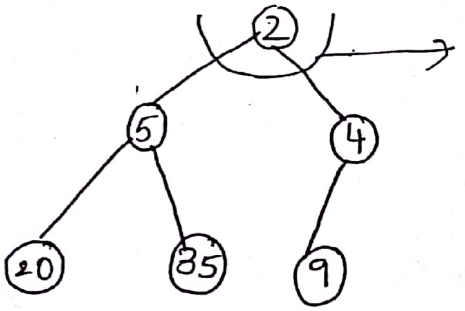
0≥1(X) → swap to Left side.



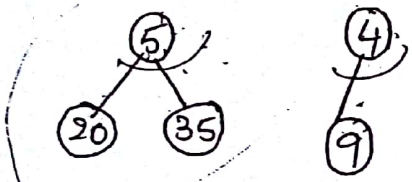
Now it is in the form of Leftist tree.

Delete Min :-

* Delete min() operation is simply deletes root node, because in minimum Leftist tree should be minimum.



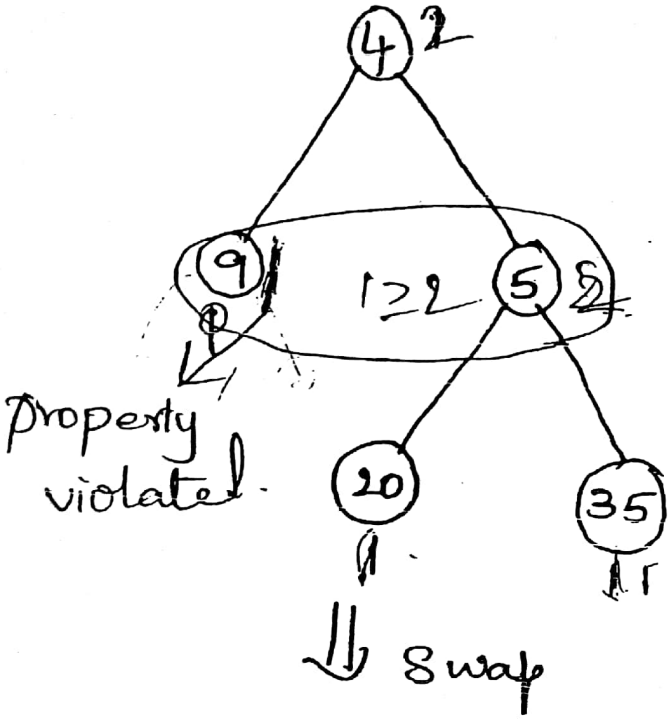
deletemin()
delete node "2"



Now combine the two min trees with the help of meld operation.

Meld Operation:

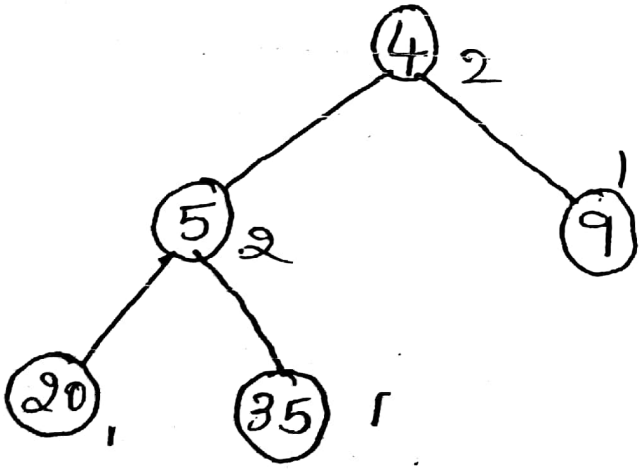
1. First compare the roots of two trees
   ie 5 > 4 (So 5 subtree will be right subtree of node (4).



ight-

{ Now, calculate shortest Height for each node

1+min{0,0}



Now calculate Heights
Now it is in the form of Leftist tre

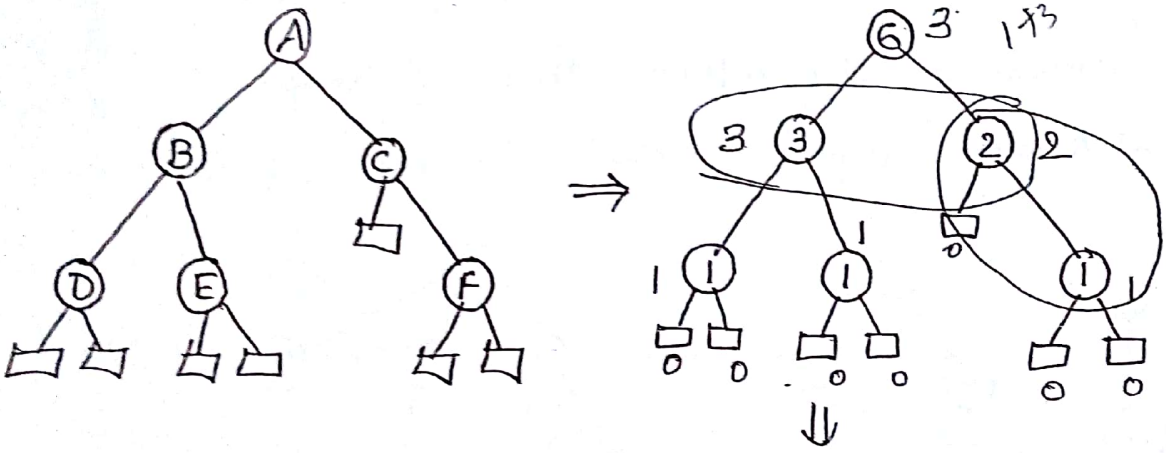
Weight- Biased Leftist Trees (WBLT):

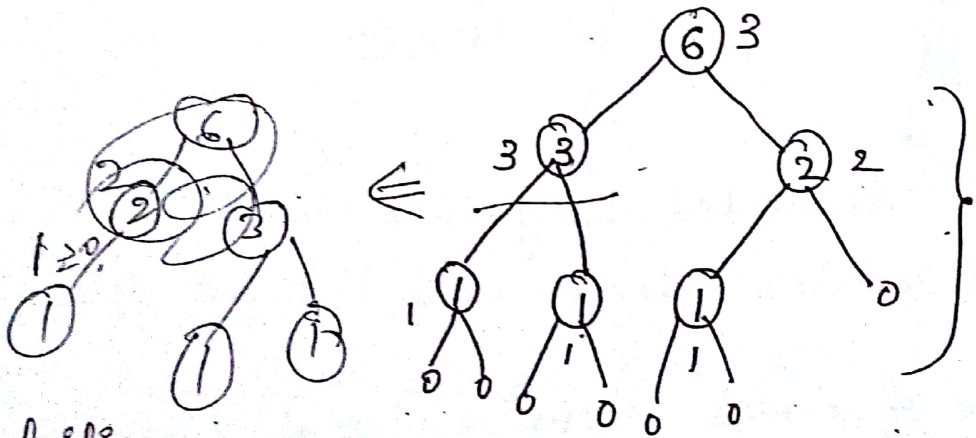
* In this we define weight $w(x)$ of node 'x'.
* Weight $w(x)$ of node 'x' is number of internal nodes in the subtree with root 'x'

Note:

1) If x is an External node, its weight $(x) = 0$.
2) If 'x' is an internal node, its weight is 1 more than the sum of the weights of its children.

Eg: consider the following Extended binary, the weights of the nodes are.





Definition:

A binary tree is a WBLT (Weight- Biased Leftist tree) if at every internal node the w(eight) value of the left child is greater than or equal to the weight (w) value of the right child.

Binomial Heaps:

A binomial Heap is a data structure that supports the following operations.

(1) Insert
(2) delete min (or) delete max
(3) Meld

The above operations are supported by leftist trees.

Definitions:

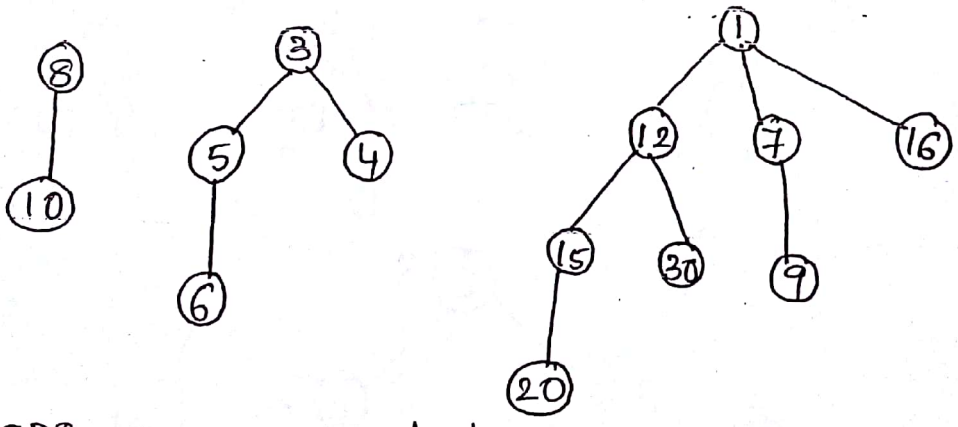
Types: There are two varities of binomial Heaps.

Min Binomial Heap → it is a collection of min trees.

Max Binomial Heap → it is a collection of max trees.

* Binomial Heaps are also referred to as B-heaps.

eg:) The following figure represent B-heap with three min trees.



* B-Heaps are represented using nodes with the following data members. degree, child, link and data.

Degree: Degree of a node is the number of childrens it has

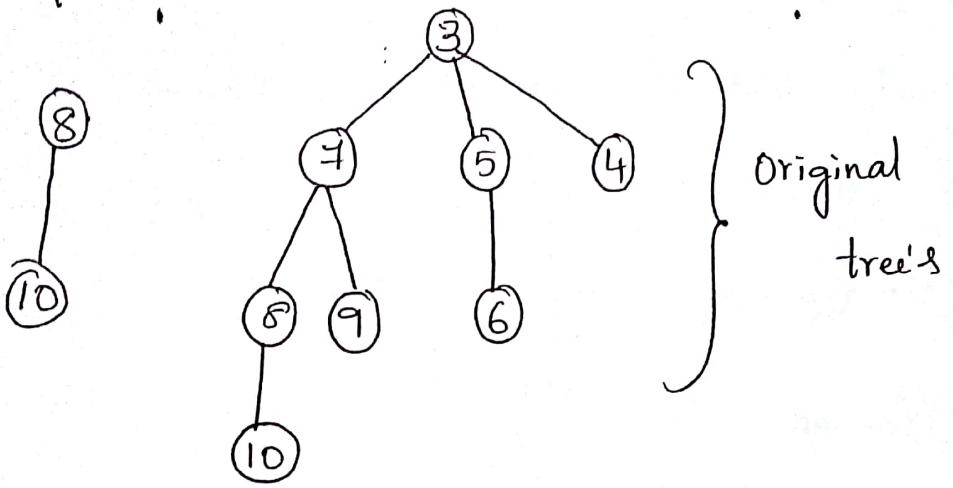
child: child data member is used to point any one of its children.

Link: The Link data member is used to maintain Singly linked circular lists of sibl

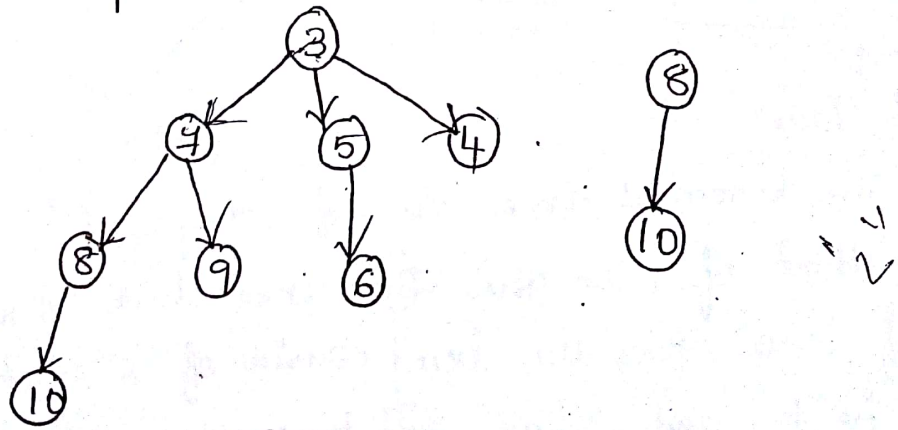
All the children of a node form a singly circular linked list, and the node points to one of these children.

* The roots of the min trees that comprise a B-heap are linked to form a singly linked circular list.
* Then the B-Heap pointer is pointed to root of the which is of minimum value.

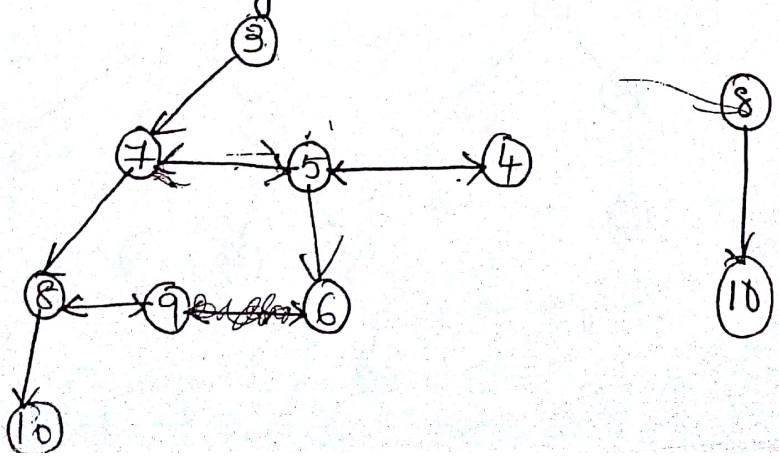
Representation:



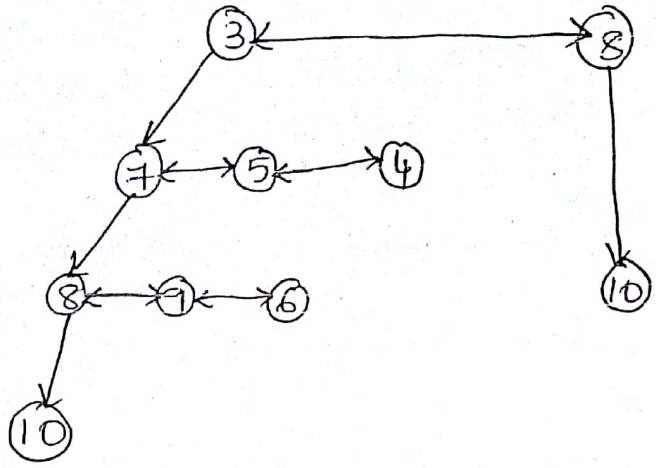
(1) first Represent children's with "⟶"



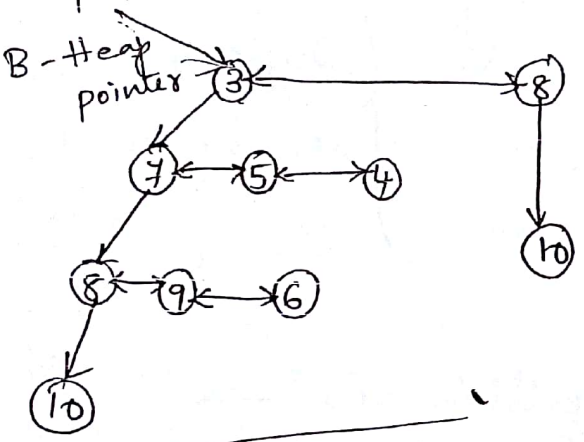
(2) Represent siblings now ⟺.



(3) Now combine roots of the trees.



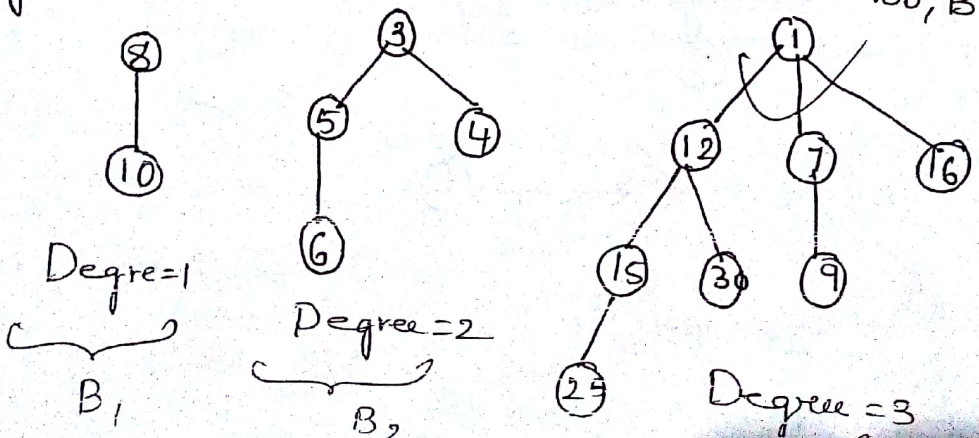
(4) B-Heap pointer points to node (3) (bcz Minimum)



Insertion:

Binomial Tree:

The binomial tree B_k of degree 'k' is a tree, such that if $k=0$ then the tree has Exactly one node and if $k>0$, then the tree consist of a root whose degree is k and whose subtrees are $B_0, B_1, \ldots B_{k-1}$.



Here B_k has exactly 2^k nodes.

B_1 has $2^1 = 2$ nodes

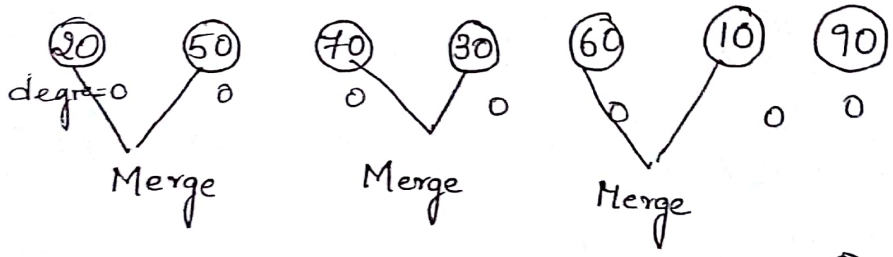
B_2 has $2^2 = 4$ nodes

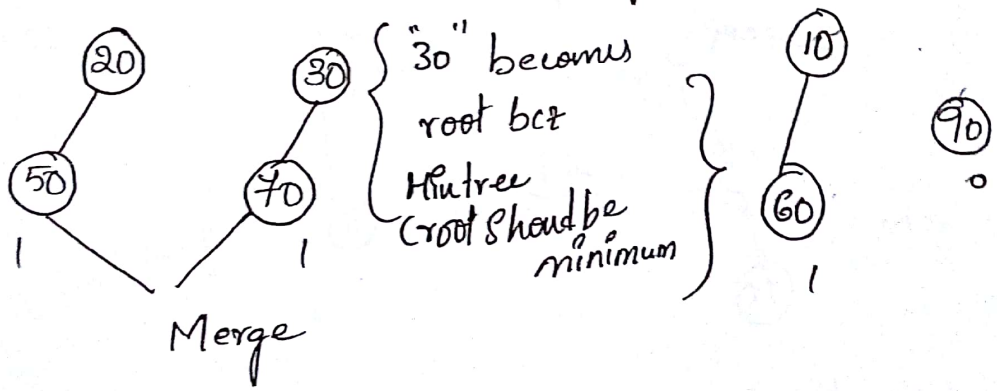
B_3 has $2^3 = 8$ nodes.

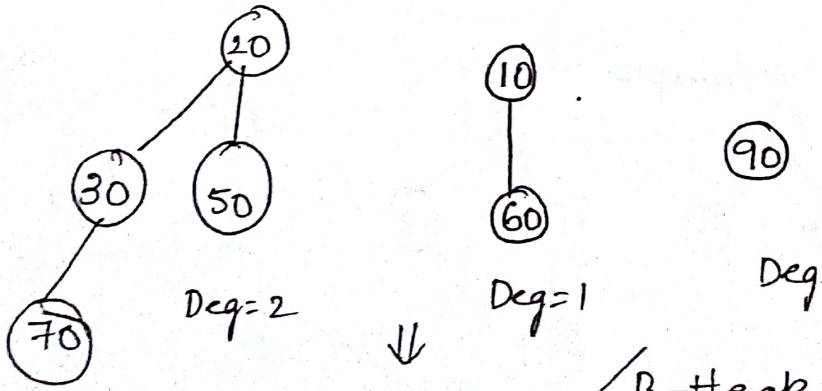
Insertion:

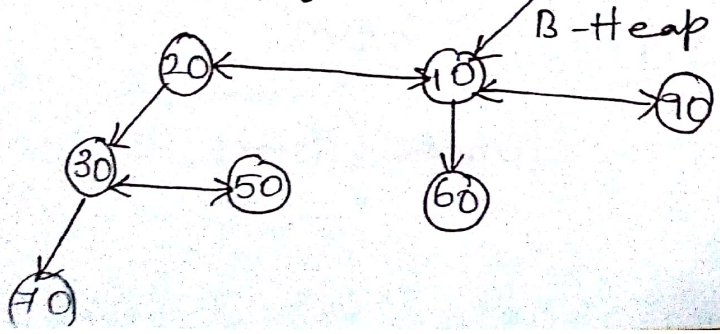
Exp: 20 50 70 30 60 10 90

Note: Melding operation can be done ;i,e the trees with the same degree.



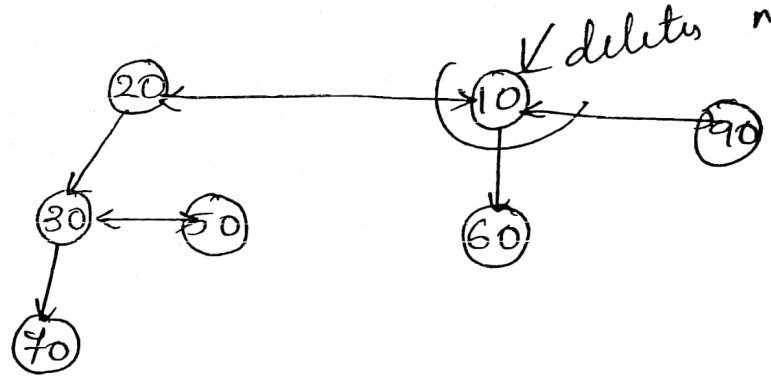


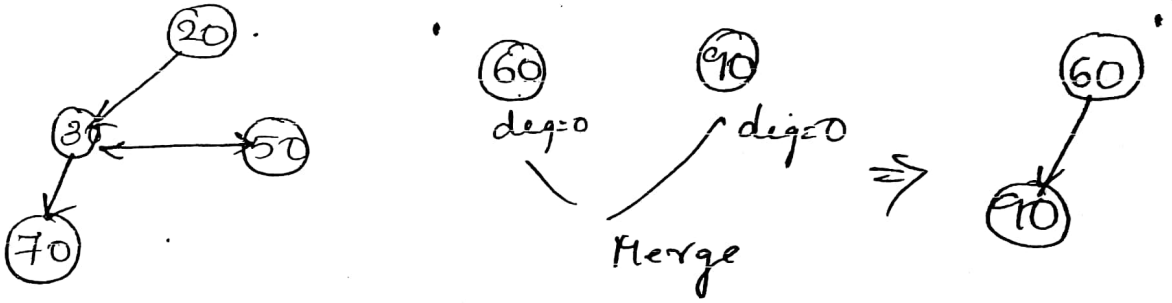


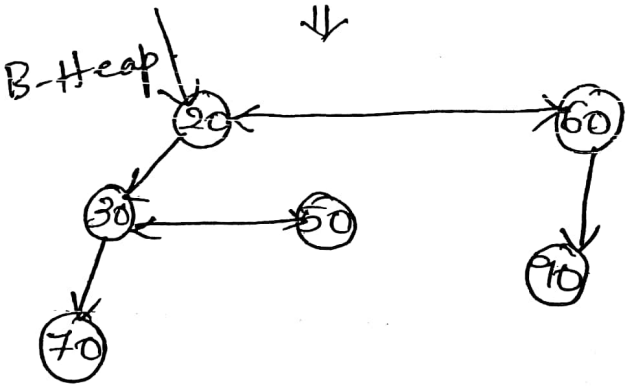


Delete Min :-

Delete min operation deletes simply minimum root node.

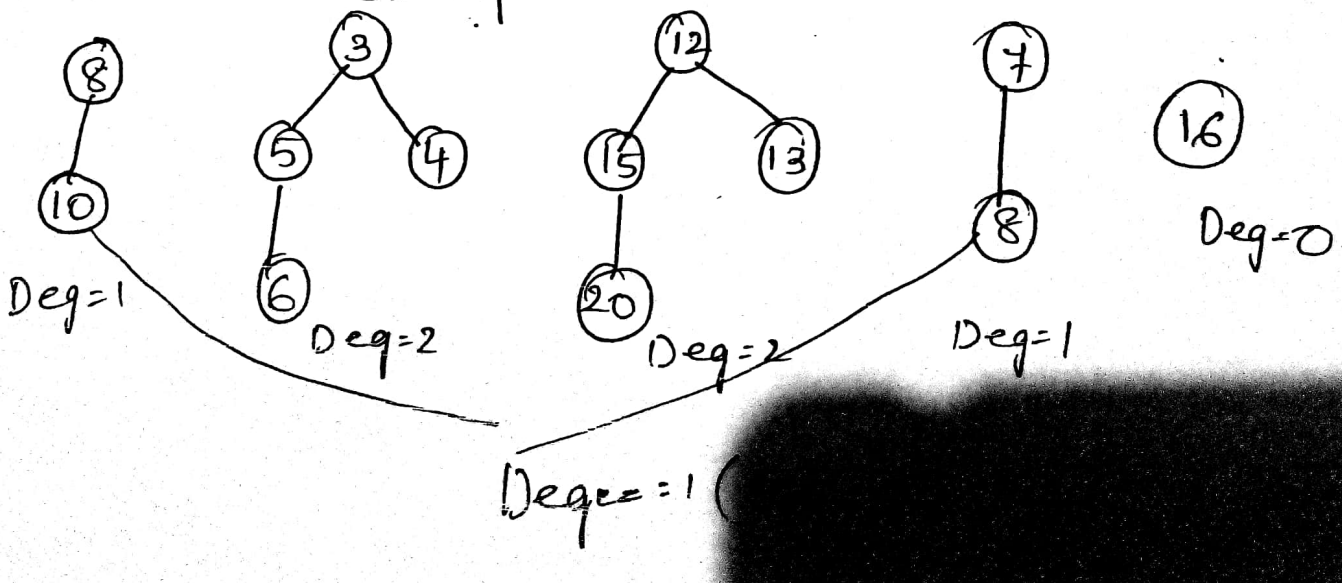


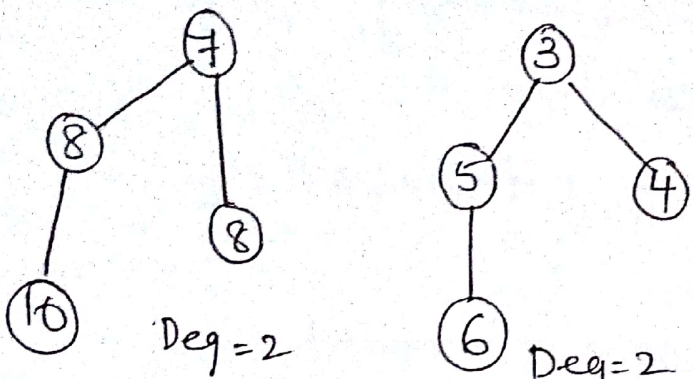




Meld Operation:

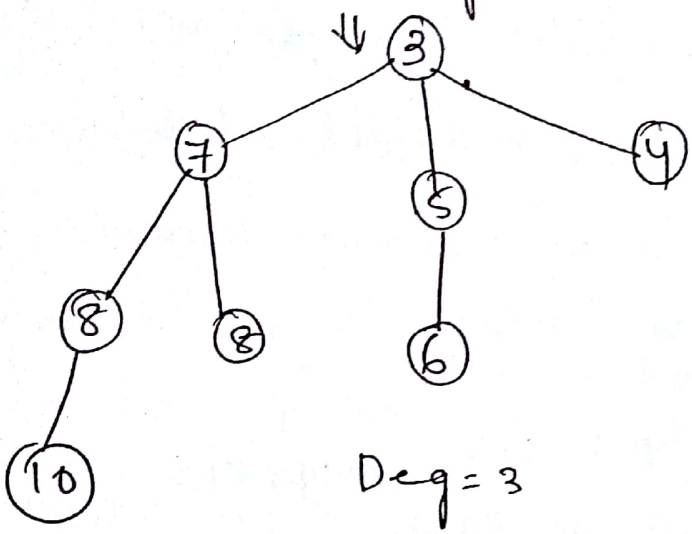
Take some Example trees.



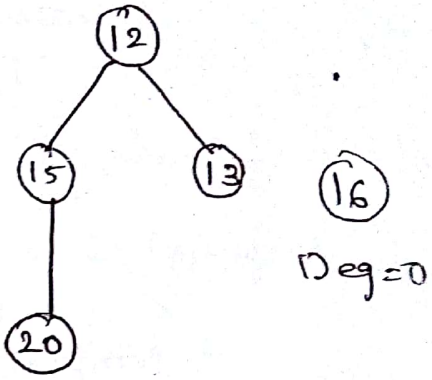


Deg = 2 Deg = 2

Merge

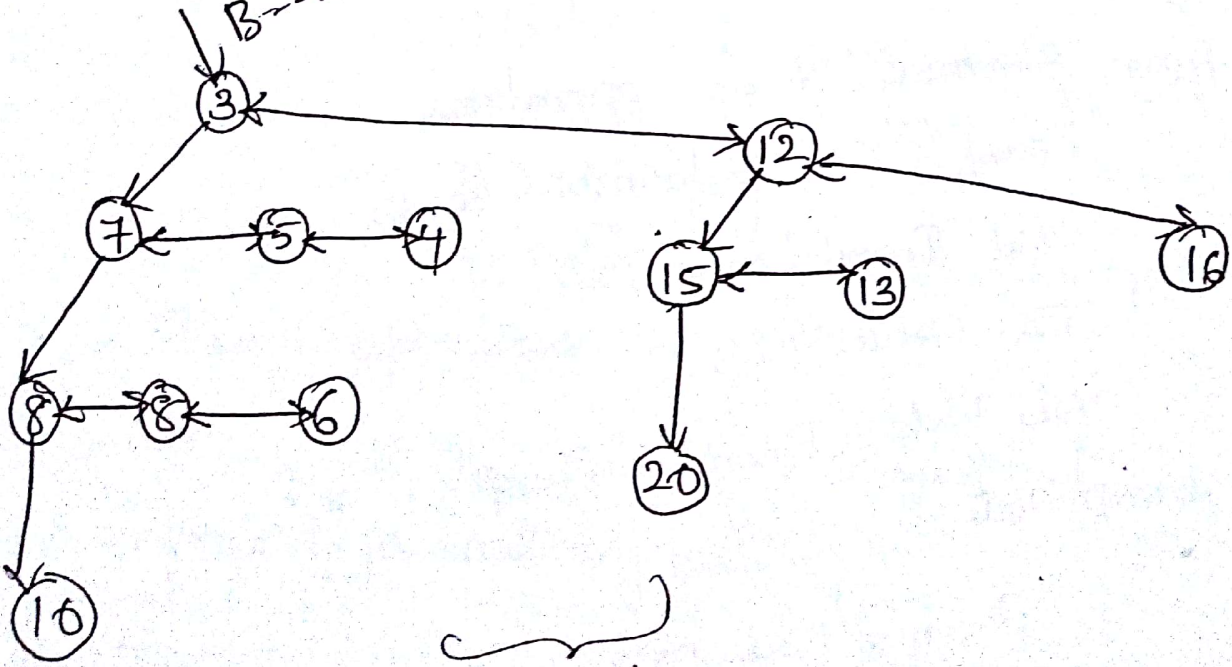


Deg = 3



Deg = 2 Deg = 0

Represent in the form of Binomial Heap.



B-Heap

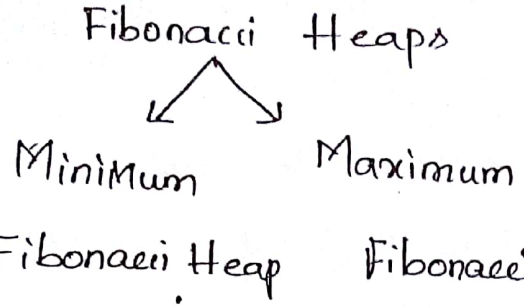
The following code contains the class definitions of Binomial node and Binomial Heap.

```
template<class T> class BinomialHeap;
template <class T>
Class BinomialNode
{
  friend class BinomialHeap<T>;
 private:
    T data;
    BinomialNode <T>* child, * link;
   int degree;
};
template <class T>
Class BinomialHeap : public minpq<T>
{
  public:
   BinomialHeap (BinomialNode<T>* int=0) min (init)
 {
 };
// Four Binomial Heap operations
      const T& GetMin() const;
      void Insert(const T&);
      T& DeleteMin();
      void Meld (BinomialHeap<T>*);
      private:
         BinomialNode <T> *min;
};
```

Fibonacci Heaps:

* Fibonacci Heaps is a data structure for priority Queue operations, consisting of a collection of heap-ordered trees.

*



* A minimum Fibonacci Heap is a collection of min trees
* A Maximum Fibonacci Heap is a collection of max trees
* Fibonacci Heap also refered to as F-Heaps.
* B-Heaps are special case of F-Heaps.

All examples of B-Heaps are also examples of F-Heaps.

* An F-Heap is a data structure that supports the four B-heap operations such as

1) GetMin
2) Insert
3) DeleteMin
4) Meld.

Same operations like Binomial Heap

as well as the following additional operations.

5) Delete: Delete the Element in a specified node. This is refered to as arbitary node.

6) Decrease key: Decrease the key/priority of a specified node by a given positive amount.

Delition from an F-Heap:

To delete an arbitary node 'b' from a F-Heap these are the following steps.

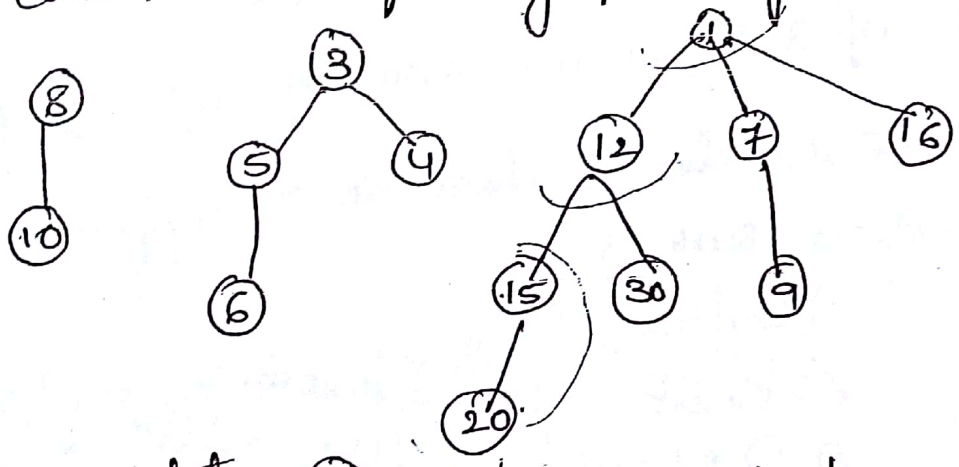
(1)* If min = b then do a delete min; Otherwise do steps 2, 3, and 4 below.

(2) Delete 'b' from its doubly linked list.

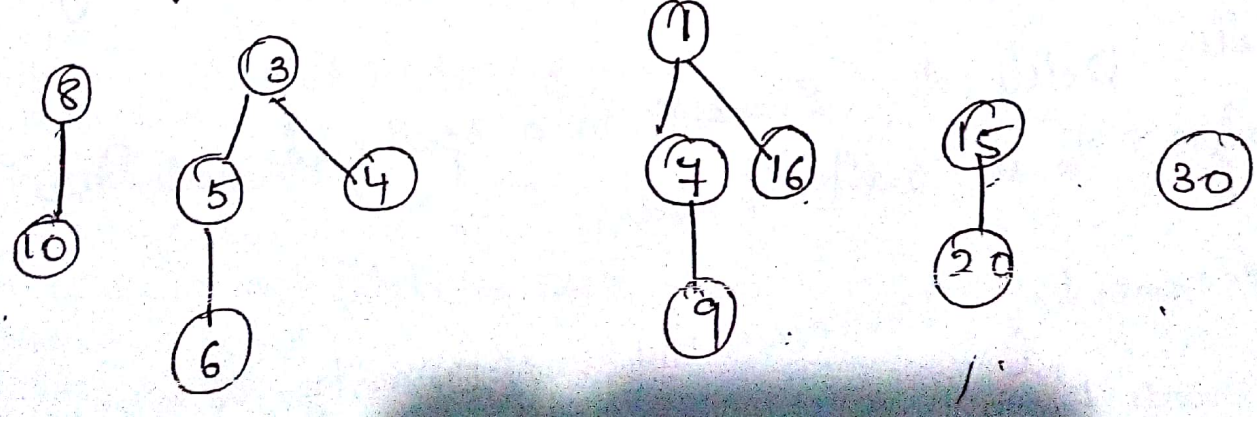
(3) Combine the doubly linked list of b's children with the doubly linked list pointed at by min into a single double linked list. Trees of equal degree are not joined as in a delete-min operation.

4) Dispose of node 'b'.

eg: consider the following F-Heap.



* Now delite (12), first check if there is any children for the given node, if it is present make the new tree



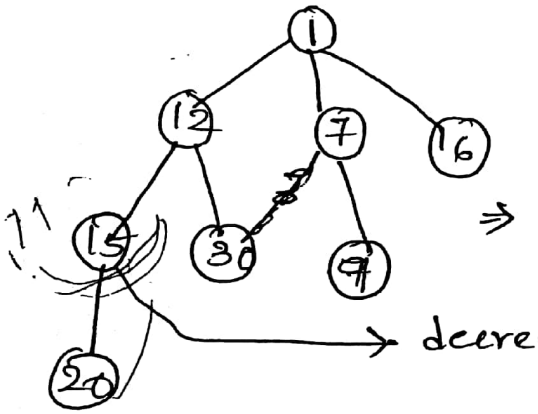
To decrease the key in node 'b' we perform the following steps.

(1) Reduce the key in 'b'

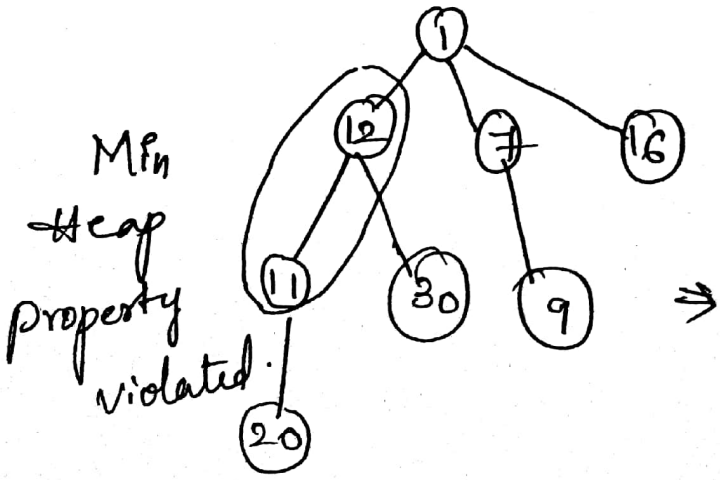
(2) If b is not a min tree root and its key is smaller than that in its parent, then delete 'b' from its doubly linked list and insert it into the doubly link list of min tree roots.

(3) change min to point to b if the key in b is smaller than that in min.

eg: Suppose we decrease the key 15 (by 4).

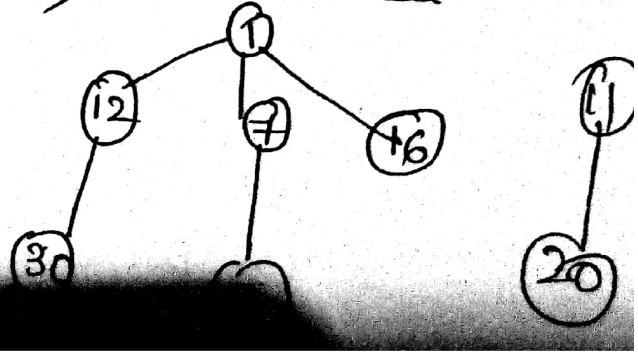


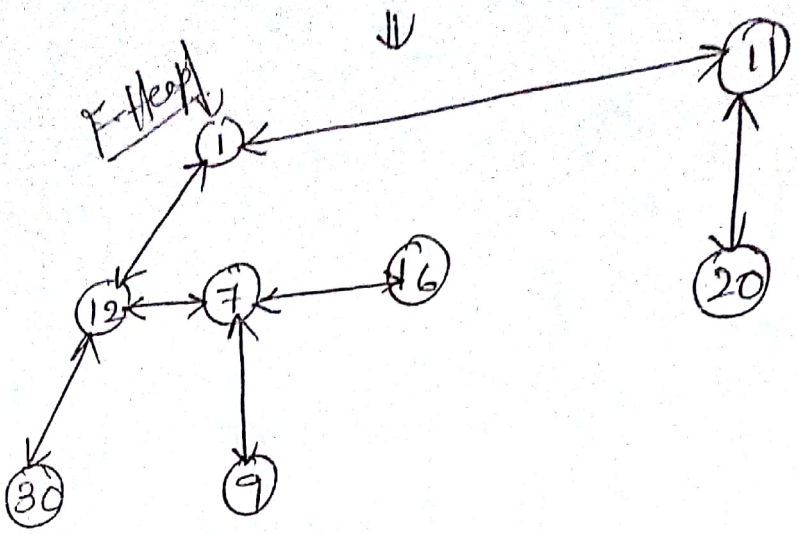
(11) Now



Min Heap property violated.

Remove (11) with(and make seperal tree.





Pairing Heaps:

* The pairing heap supports same operation supported by the Fibonacci Heap.
* Basically it used two merge Heaps.

* pairing Heaps are two types.
  1. Min pairing Heaps
  2. Max pairing Heaps.

Min pairing Heaps: min pairing heaps are used when for min priority Queues.

Max pairing Heaps: Max pairing Heaps are used for max priority Queues.

* A min pairing Heap is a min tree in which we perform the following operation.
  1. Meld
  2. Insert
  3. Decrease key
  4. Delete Min
  5. ... specified node

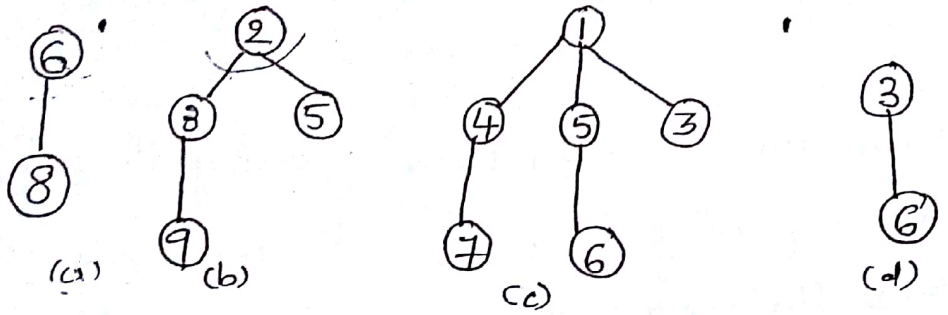
perations:

eld:

Two min pairing Heaps may be melded into a single min priority heap by performing a compare-link operation.

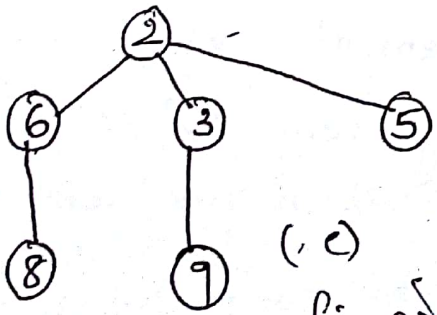
→ In a compare Link, the roots of the two min trees are compared and the min tree that has the larger root is made the leftmost subtree of the other tree.

Eg:



→ To meld the min trees of above figures (a) & (b) we compare the two roots.

Here Tree in fig(a) has the larger root, this tree becomes the leftmost subtree of tree in fig(b). ~~becomes the leftmost subtree of tree~~



(, c)

fig (e).

→ Meld C & D.

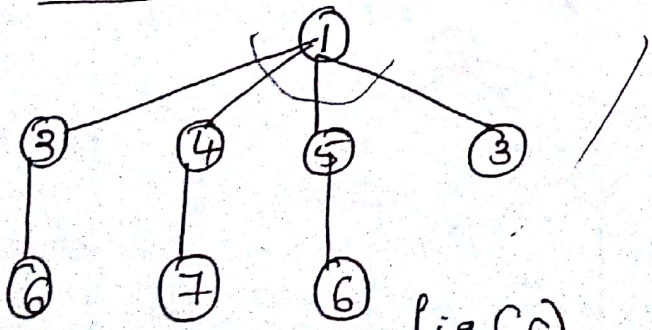


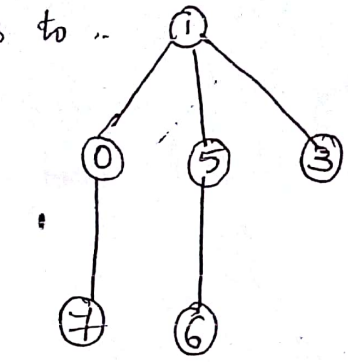
fig (f).

The corrective action consits of the following steps:

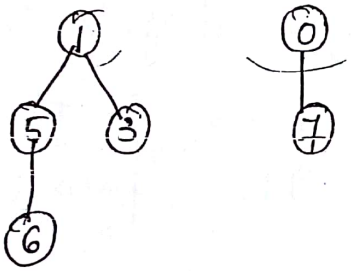
Step 1: Remove the subtree with root 'N' from the tree. This results two min trees.

Step 2: Meld the two min trees together.

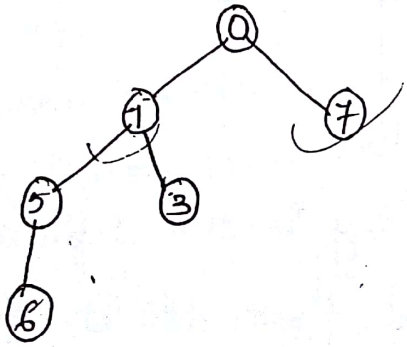
Now apply two steps to ..



After applying step(1) the two min trees are.



Now apply step (2) i,e Meld two min trees, the result is



Delete Min:

In pairing Heap, the min element is in the root of the tree. So, to delete the min element, we first delete the root node.

> When the root is deleted, we are left with zero (or)

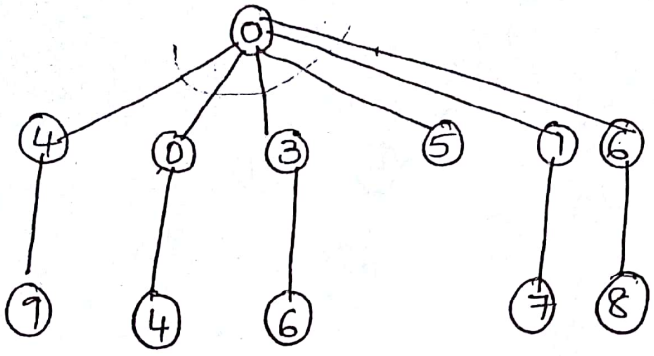
when the number of remaining min trees is two (or more) these min trees must be melded into a single min tree.

* In two pass pairing heaps, this melding is done as follows.

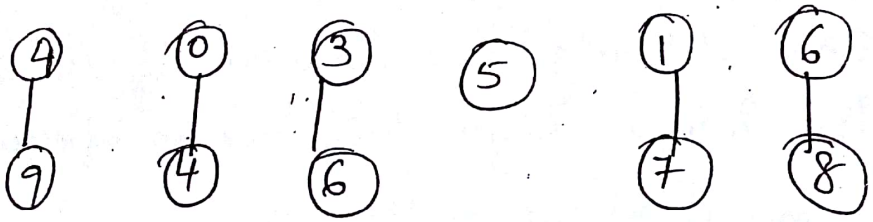
Step 1: make a left to right pass over the trees melding pairs of trees.

Step 2: Start with the rightmost tree and meld the remaining trees (right to left) into this tree one at a time.

eg:



* When the root is removed, we get the collection of 6 min trees, ..

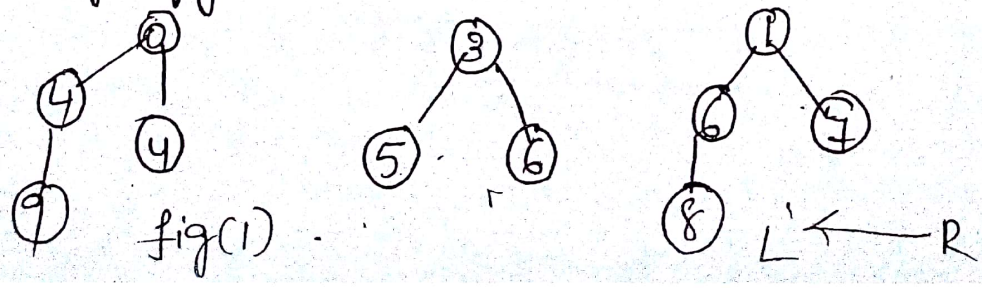


L → R.

* In the left to right pass of Step1, we first meld the trees with roots 0 & 4.

Next the trees with root ③ & ⑤ are melded, finally, the trees with root ① & ⑥ are melded

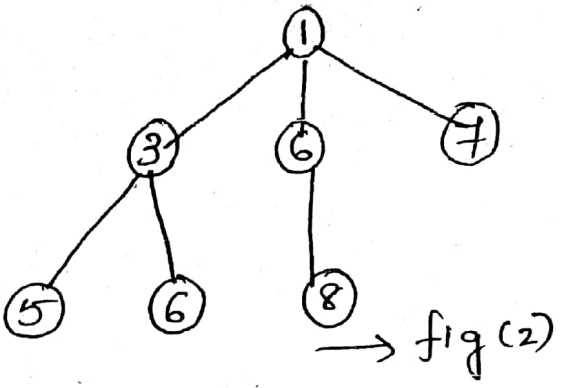
e following figure shows resultant min trees.



fig(1).

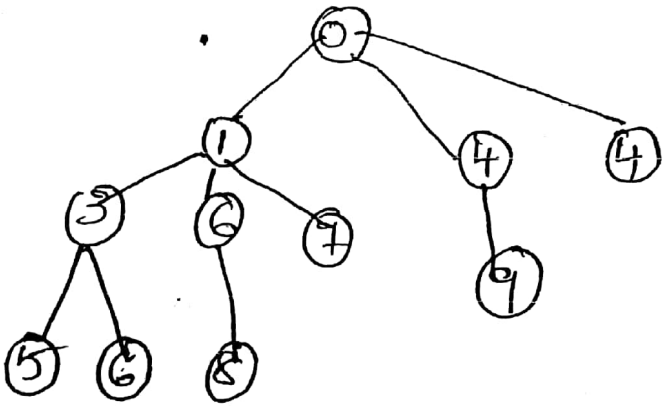
L ← R

* Now right to left.



→ fig (2)

Now Meld fig(1) & fig(2), resultant tree is



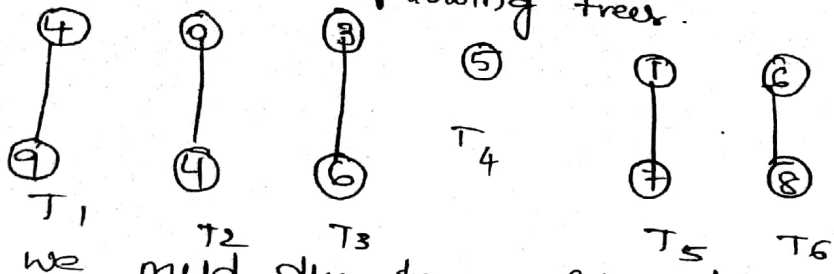
Multipass pairing Heaps.

In Multipass pairing Heaps, the min trees that remain following the removal of the root are melded into a single min tree as follows.

1) put the min trees into a FIFO Queue.
2) Extract two trees from the front of the Queue, meld them and put the resulting tree at the end of the Queue..

Repeat these steps until only one tree remains.

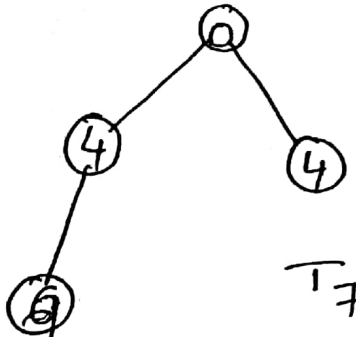
consider the following trees.



* First we meld the trees with roots 4 and 9 and put the resultant min tree at the end of the queue.

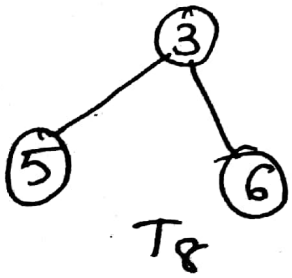
Initial Q = | T1 | T2 | T3 | T4 | T5 | T6 |

Now Q = | T3 | T4 | T5 | T6 | T7 |



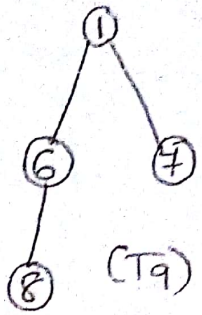
Next the trees with roots (3) & (5) are melded and the resultant min tree is put at the end of the queue.

Q = | T5 | T6 | T7 | T8 |

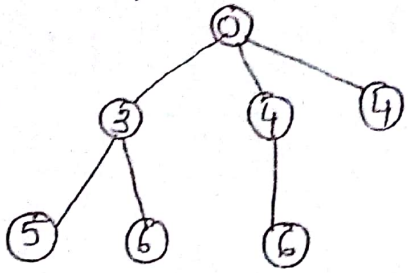


Now trees with roots (1) & (6) are melded and the resultant min tree added to the Queue at the end.

Q = | T7 | T8 | T9 |

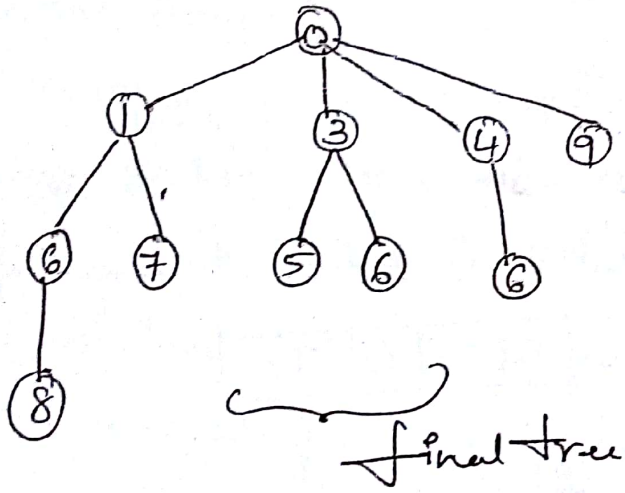


Now Held T_7 and T_8. The resultant min tree is T_{10} added to the end of the queue.



Q = | T_9 | T_{10} |

Now meld T_9 and T_{10}. The resultant tree is



Q = | T_{11} |

Arbitary Delete:

Deletion from an arbitary node N is handled as a delete-min operation, when N is the root of the pairing heap.

When N is not the tree root, the deletion done as follows.

) Detach the subtree with root N from the tree.