Data structure – It is the particular way of organizing data so that it can be used effectively

using an effective data structure is the key point for designing an effective algorithm.

→ Hash tables used in compiler design/implementation

→ B-trees are used in implementation of data bases.

Priority Queue :-

It is an abstract data type in computer Programming.

Exactly like a regular queue - but here each element is associated with a Priority

Elements with highest Priority are pulled first

Priority Queue is always different from heap but it can be implemented using heap data structure.

Operations :- 1) insert_with_Priority :- adding element to queue with an associated Priority.

2) Pull_highest_Priority_element :- removing elemat with highest Priority not the one from end.

some other operations like Pull-lowest_Priority_element can also be used.

For deleting elements in stack behaving like Priority queue - while inserting increment the value because the last element must have highest Prio.

Similarly for queue which uses FIFO - decrement Priority values.

Naive Implementation :-

Unsorted implementation :- (Keep all elements unsorted)

Whenever highest-Priority element is requested, search through all elements for one with highest Priority

(O(1) for insertion O(n) - for searching the highest Priority element)

Sorted implementation :-

there O(n) insertion time & O(1) PullNext time (from front), on avg O(n*log(n)) time to initialize (using quicksort)

In order to get better Performance & reduce time Complexity Priority queues use heap data structure for their implementation.

which gives O(logn) TC for insertion & deletion
O(n) - for building.

Applications of Priority Queue :-

1) Bandwidth management (BM)
2) Discrete event simulation (DES)

Bandwidth - amount of data transmit

BM requires ensuring the handling the traffic which can be done through Priority Queue which is helpful in serving the one with more Priority first. ex: when more People are using internet at a time.

DES :- ~~When~~ simulation of Events/manage the event

Events are added to queue and based on simulation time which acts as Priority is used to Pull events.

Heap :-

It is a specialized tree-based data structure that satisfies heap property :-

If B is a child node of A, then key(A) ≥ key(B).

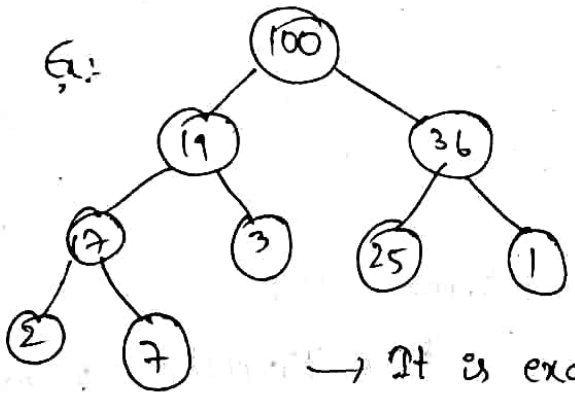
That means greatest key is always in root node

→ NO restriction as to how many children each node has in a heap.

Heaps are used in Dijkstras algorithm also.

→ Heaps are usually implemented in an array & do not requires pointers b/w elements.

Eg:-



100	19	36	17	3	25	1	2	7

→ Array form

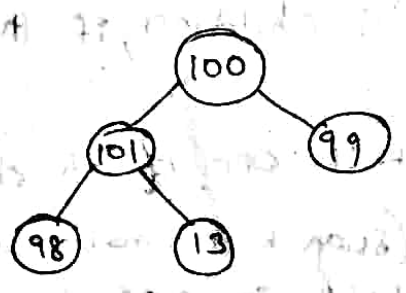
→ It is example of max heap as the highest element is the root node.

Binary heap is a special kind of binary tree. It maintains Shape Property & heap Property.

Shape Property :-

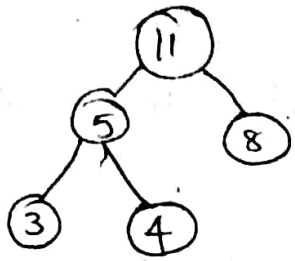
Complete binary tree :- A tree is said to be complete the nodes have atmost two children.

The nodes must be filled from left to right only.



→ Doesn't satisfy heap Property as 101 > 100

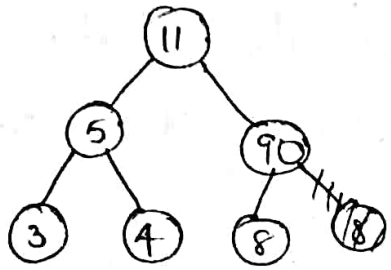
→ Satisfy shape property.



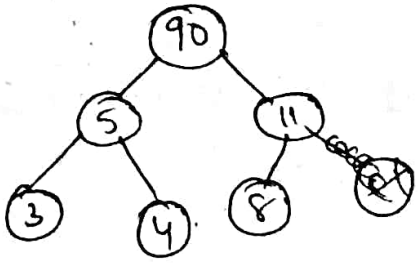
→ Max heap

→ Satisfy shape Property as nodes are filled to left to right

Insert 90 :- Operations on heap :- 1) Insert operation

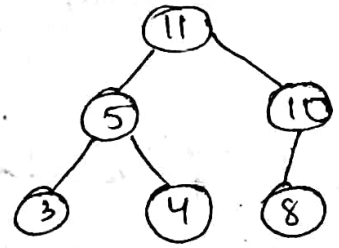


But its violating heap Property so swap 11 with 90



→ Now it is a max heap

Insert 10 not 90 :-



→ max heap

Heap Property is satisfied.

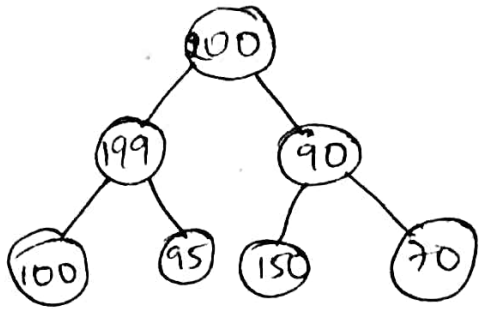
2) Remove operation :-

Procedure for deleting the root from heap.

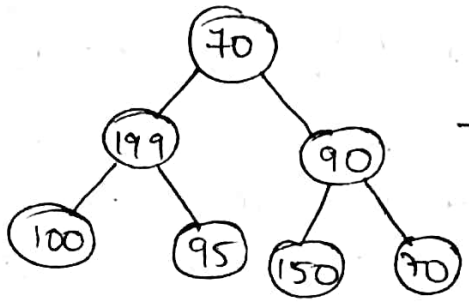
Removing root :-

1) Replace root of heap with last element on the last level.
2) Compare new root with its children, if they are in correct order stop
3) If not swap element with one of its child & return to Previous Step. (Swap with smaller child in min heap & bigger child in max heap)

Ex :-

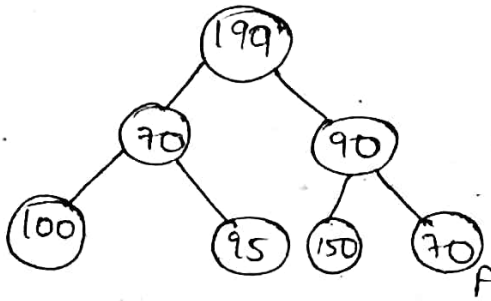


Step 1 :- Replace root with last element of last level



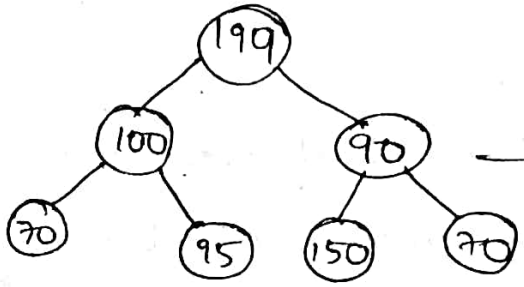
→ The ~~do~~ obtained tree is not satisfying heap Properties

Since its a max heap ~~replace~~ swap with max of its childs.



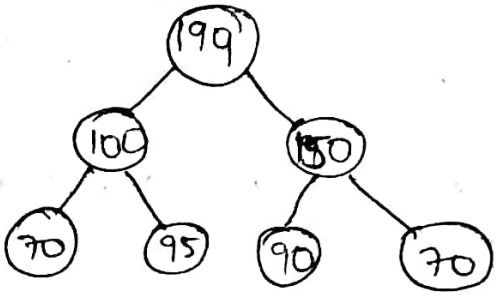
At 70 it is not satisfying heap Property

As its max heap ~~rep~~ swap 70 & 100



→ heap Property is satisfied by left subtree but not right

Swap 90 & 150



→ Now the heap Property is satisfied.

Max-heapify Algo:-

```
Max-Heapify(A,i):
left ← 2i
right ← 2i+1
largest ← i
if left ≤ heap_length[A] and A[left] > A[i] then:
largest ← left
if right ≤ heap_length[A] and A[right] > A[largest] then:
largest ← right
if largest ≠ i then:
swap A[i] ↔ A[largest]
Max-Heapify(A, largest)
```

→ last element index $2h-1$

→ A Complete binary tree is heapified using bottom up approach.

→ There is no need to heapify leaf elements

→ No need of pointers can be implemented using arrays.

Binary heap:-

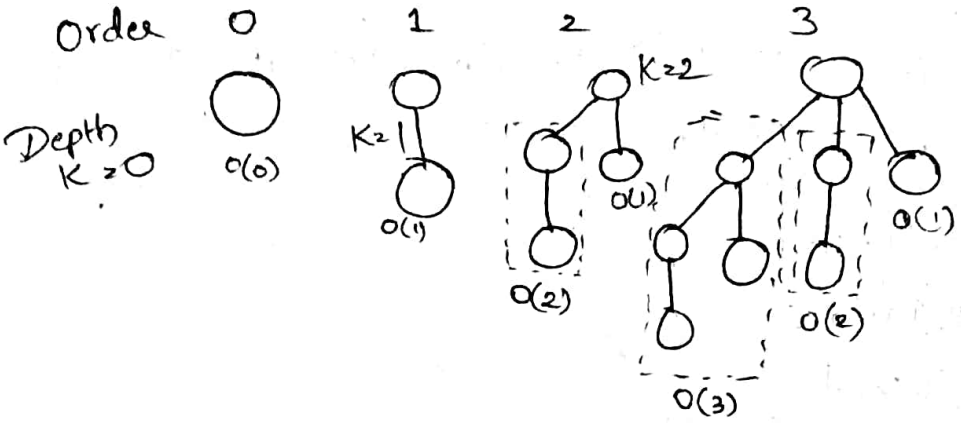
Binomial heap:- a heap which is similar to binary heap

→ Supports merging two heaps

→ special tree structure so that two can be merged.

→ implementation of mergeable heap abstract data type.

Binomial heap provides faster merge oper than Binary heap



Leaf elements are at depth 0

Binomial heap is implemented as a collection of binomial trees. Binomial tree is defined recursively :-

→ A BT of order 0 is a single node.

→ A BT of order k has a root node whose child are roots of BT's of orders $k-1, k-2, \ldots 2, 1, 0$

* A binomial tree of order k can have (exactly) 2^k nodes.

→ A BT of order k can be constructed from 2 trees BT's of order $k-1$

→ there are exactly ^kC_i nodes at depth $i = 0, 1, \ldots k$

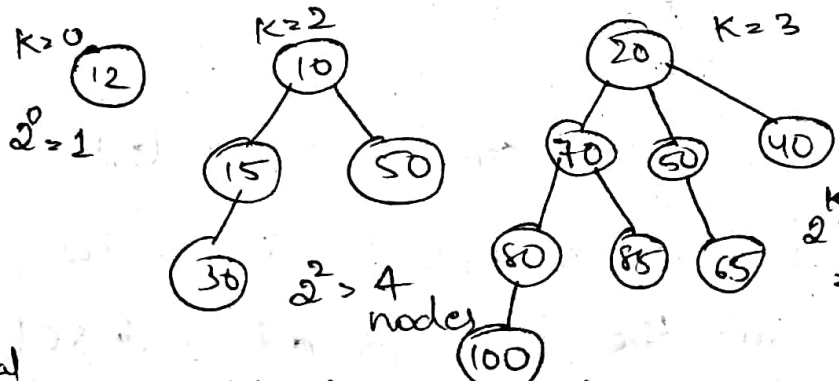
Binomial heap properties :-

→ Each BT in a heap obeys min-heap property

→ there ~~are~~ can only be either one or zero BT's for each other, including 0 order.

Eg:- A binomial heap with 13 nodes

It is collection of 3 binomial trees of orders 0, 2, 3 from left to right.



13 nodes

1 1 0 1

B_3 B_2 B_1 B_0

✓ ✓ ✓

(1 – indicates the binary trees are in collection)

Here we have trees of 0, 2, 3 order

Min heap Property is satisfied by all trees

total nodes in 3 trees = 13

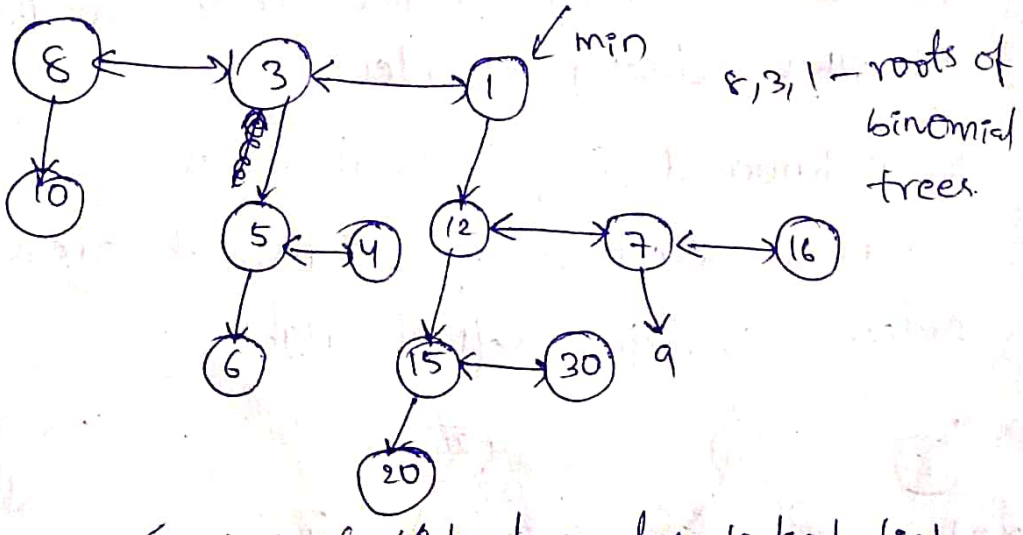
→ Binomial heap with n nodes consists of atmost $\log n + 1$ binomial trees.

Functions :- 1, Insert O(1)
2, Delete O(log n)
3, Meld (merge) O(1)

B-heaps are represented using node have following data members :

① Degree of node → no. of child it has
② Child node link → Point to any of its child
③ link of siblings – singly Circular linked list of all siblings.
(All child of a node from a singly circular list & the node Points to one of its child)
④ Data → roots of min trees that Comprises B-heap are linked to form singly linked circular list

fig-1



↔ singly ~~linked~~ circular linked list

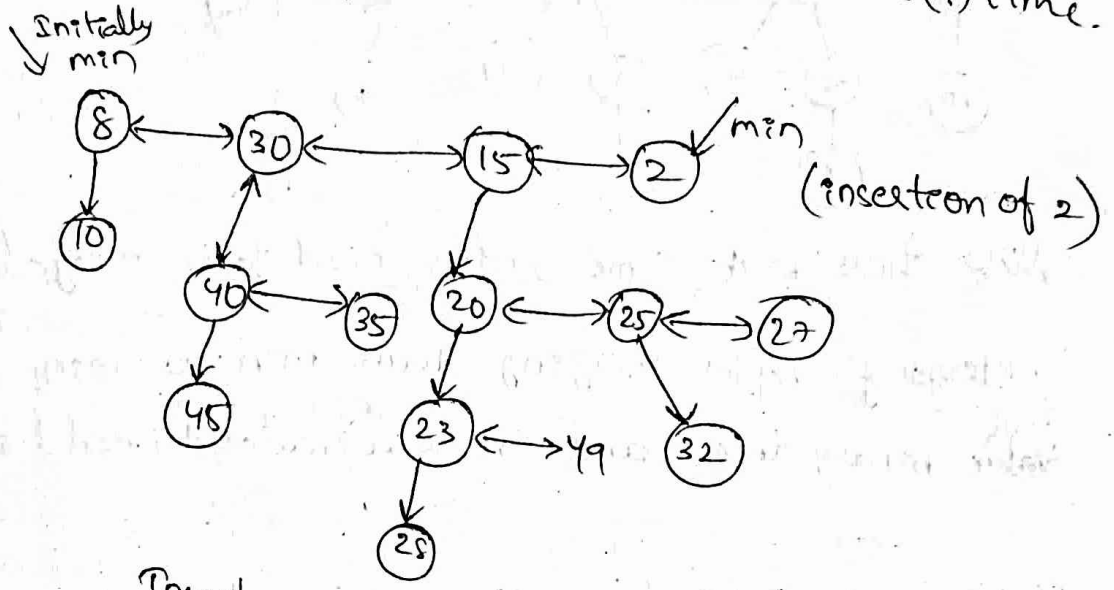
3 child are 5 & 4 and 3 Points to anyone of 5, 4 and ~~the~~ it ~~is~~ formed a ~~d~~ SCLL with other child.

Insertion:-

x - element to inserted to b-heap by first Putting x into a new node & then inserting this node into circular linked list pointed by min.

→ The pointer min is reset to this new node only if min is 0 (or) key of x is smaller than key in the node pointed at by min.

Insert operation can be done in O(1) time.



Insert x; key(x) = 2

min = 8

now 2 < 8 so change min to 2

8, 30, 15 — roots of binomial trees.

The order of binomial trees in binomial heap are always unique

i.e; No two b-trees has same order.

Now - 2 has order 0 so it can be inserted.

If we have a element of binary tree with order 0 we must check while inserting if order 0 present we must merge these two (the existed one & new one)

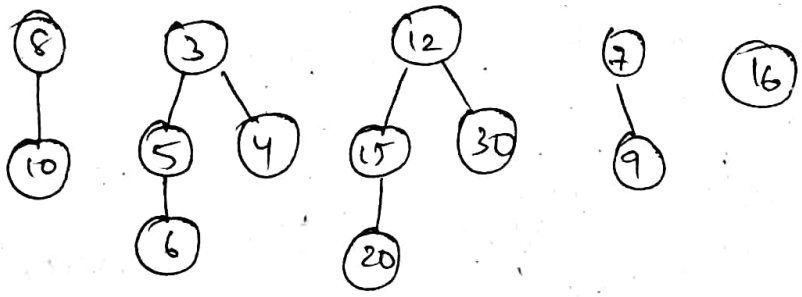
Deletion :-

If Binary heap - deletion can't be zero.

We need to delete min node from circular linked list

Consider Fig 1 Min is "1"

→ Go to child of min node, we need to create new binary trees for all child.



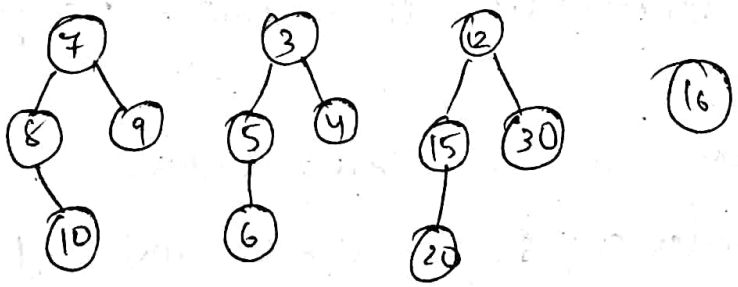
Now trees with same orders need to be merged.

Merging After merging Point min to min root ~~value~~ binary tree and create single Circular linked list

If no same order tree - then for insertion tc O(1)

if exits - then tc O(logn)

Merging :-



At a time we can merge only 2 trees.

So merge ① & ②

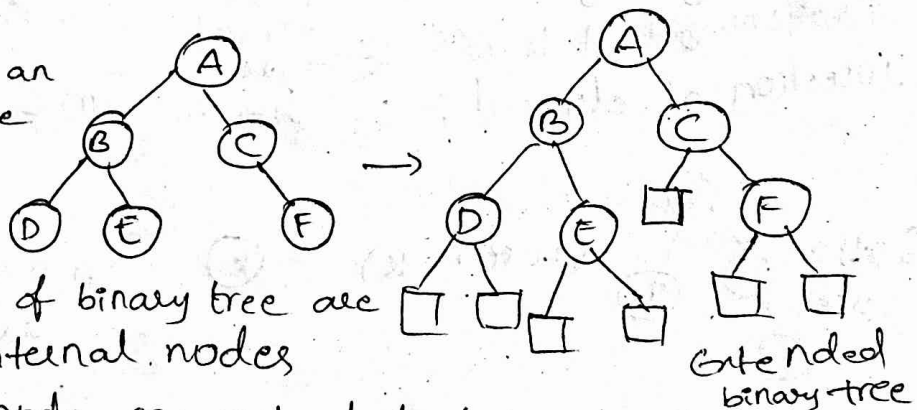
Left ist trees — Height Based — Length of a shortest Path from root to external nodes
— Weight based.
— no: of nodes is considered here.

↓

Provides different efficient implementation of meldable Priority Queues.

→ Defined using 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 an extended binary tree are called as External nodes



→ Original nodes of binary tree are called as internal nodes

Let x be a node in extended binary tree.

Leftchild(x) – Left child of internal node x
Rightchild(x) – " " " " "

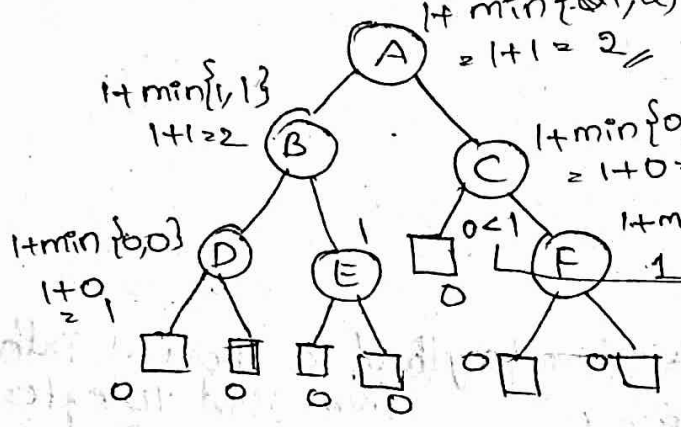
Shortest(x) – Length of a shortest Path from x to external node.

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

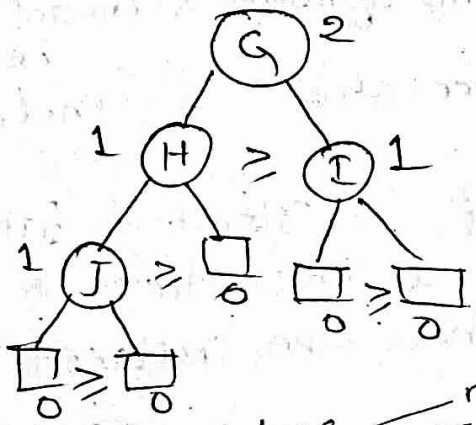
Where x is a node in an extended binary tree

Def :- A leftist tree should satisfy condition that shortest(x) of a right child must be greater than or equal to left child.

* A left ist tree is a binary tree such that if it is not empty then shortest(leftchild(x)) >= shortest(rightchild(x))



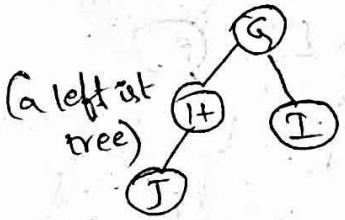
[not satisfying the condition so not a left ist tree]



[Condition is satisfied so it is a left ist. tree]

Operations on left ist tree
- meld
- insert
- delete
- find minimum

Insertion of element :-



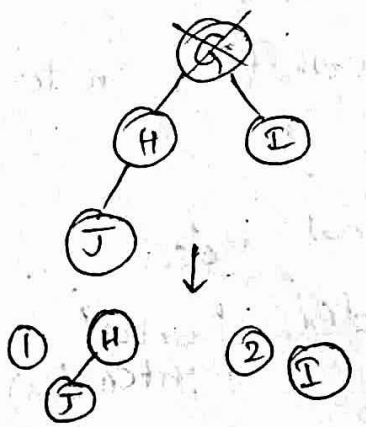
(a left ist tree)

(Insert K)

(K) → a min left ist tree as it satisfies Condition

So merge the two left ist trees.

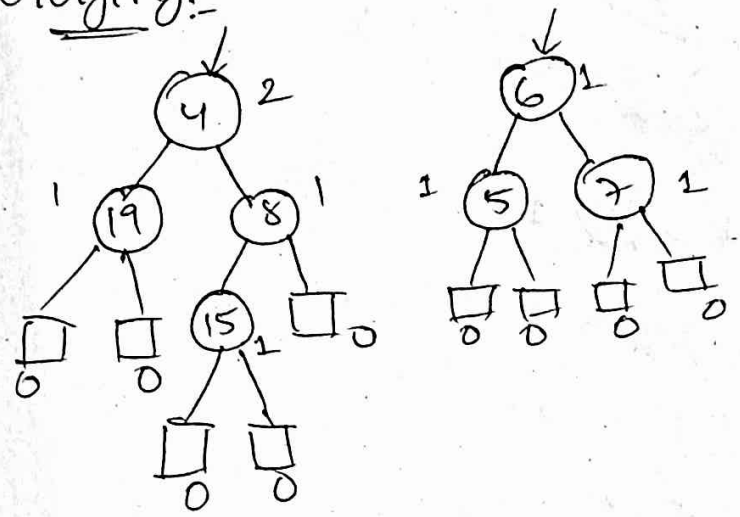
Deletion :-



'G' - min element

to remove minimum from min left ist tree we store address of right left istree & left ist tree and make them two and then merge

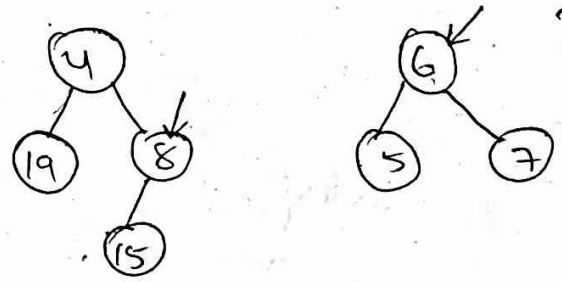
Merging:-

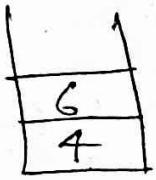


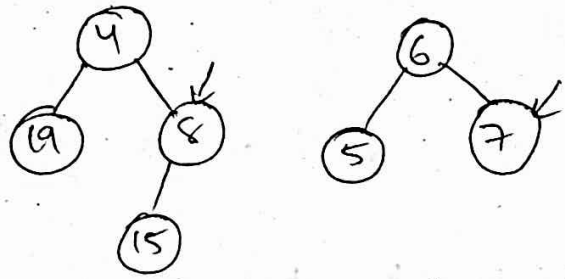
① Check whether given two are left ist tree are not

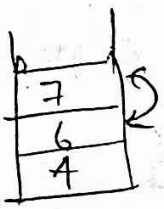
After Pushing one element Pointer moves to right child.

① →

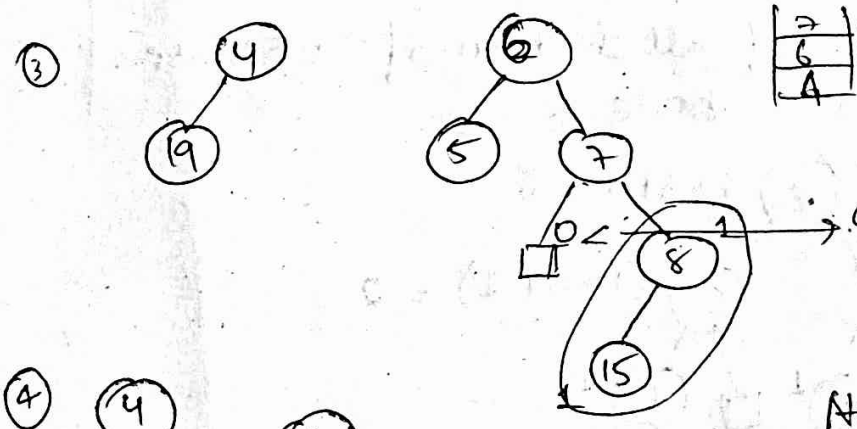




② 



Now Pointer Points to null as no right child to 7. Now move the pointer pointed subtree (i.e., & sub tree) to right of 7 and check Condition of left is tree

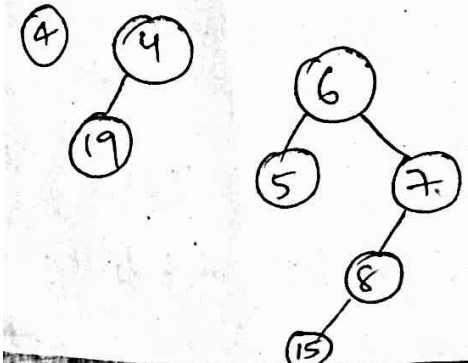
③ 

7 should be right child of 6 ✓

6 should be right child of 4

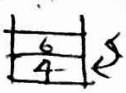
condition fails (so move & subtree as left child)

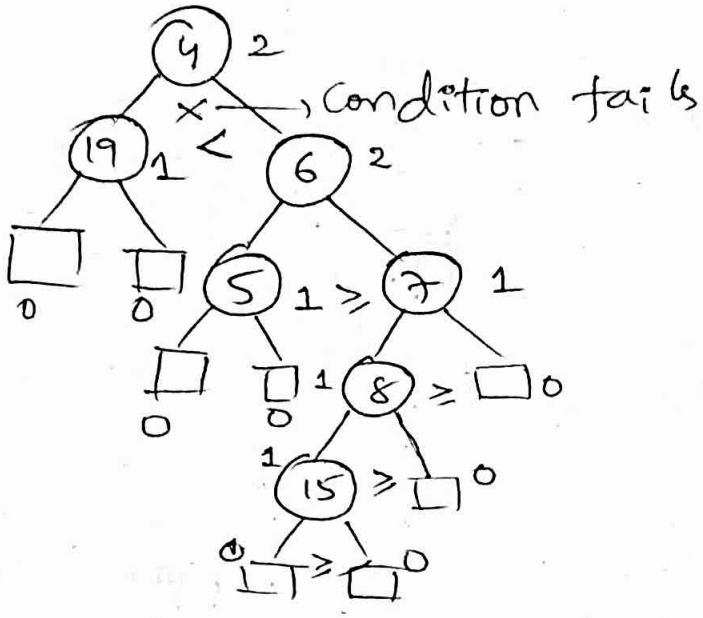
~~Now start Popping elements~~

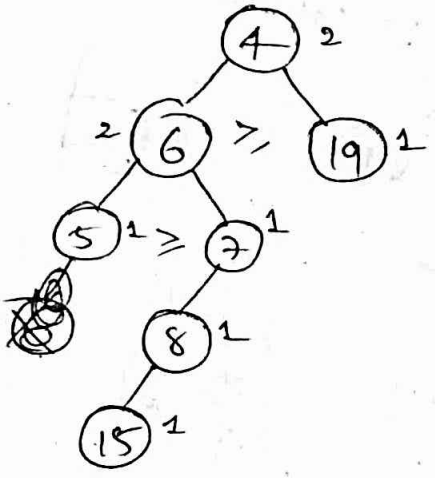
④ 

~~Here for first left ist tree Pointer Points to null so~~

6 should be right child of 4







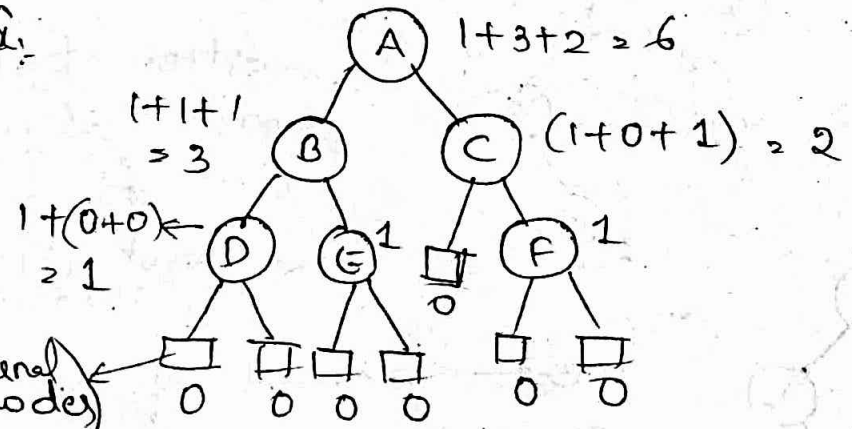
satisfied

Leftist tree :-

Weight based :- In weightbased leftist tree we are considering no: of nodes in subtree rather than length of shortest root to external node path.

$$w(x) = \begin{cases} 0 & \text{if } x \text{ is an external node} \\ 1 + \text{weight of all children of node } K & \text{otherwise.} \end{cases}$$

Q.



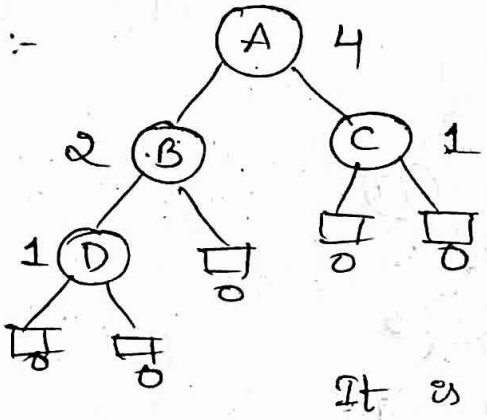
Weight = 0 * weight(x) of a node x to be no: of external nodes in subtree with root x.

Definition:- weight based left ist tree

A Binary tree is called iff for every internal node ~~W(value)~~ W[left] ≥ W[right].

In above example at C it is failing so it is not a weight based leftist tree.

Q:-

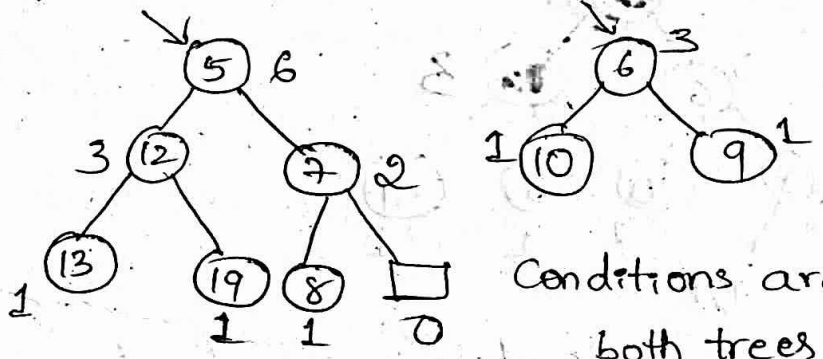


It is a weight based leftist tree as all nodes satisfy condition.

Insert, delete, & merging in weight based leftist trees:-

Merging:-

1) check whether given two trees are weight based left ist trees or not



Conditions are satisfied by both trees. so both are weight based left ist trees

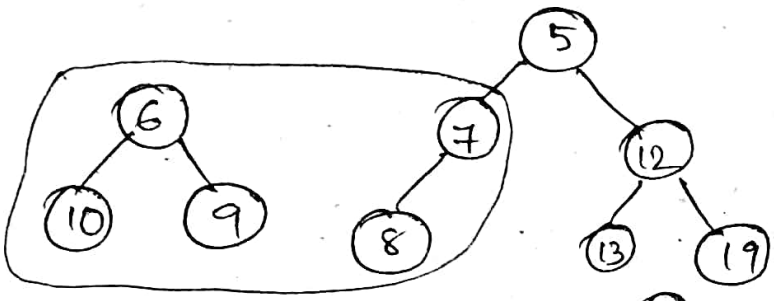
Pointer Points to (5) & (6) initially so 6 comes as child (usually right child) of 5.

~~5~~

→ whenever we add (6) as right child total ~~val~~ weight becomes 2(of 7)+3(of 6) = 5

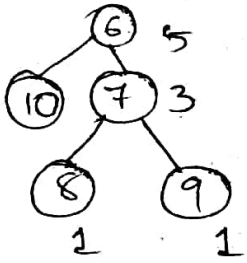
3 < 5 and condition fails so swap

~~take (6) as left child.~~

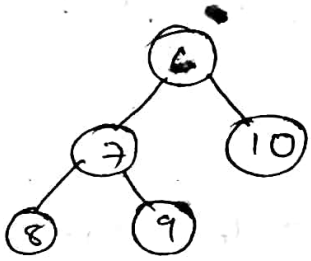


6 < 7
19

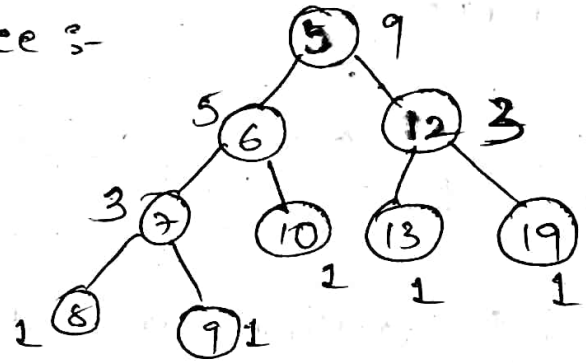
Merging :-



But condition fails so swap in this sub tree.

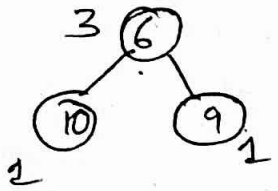


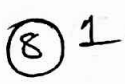
total tree :-



Condition satisfied.

Insertion :-





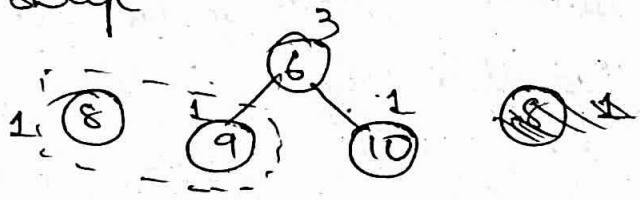
Both are weight based left ist tree

$6 < 8$

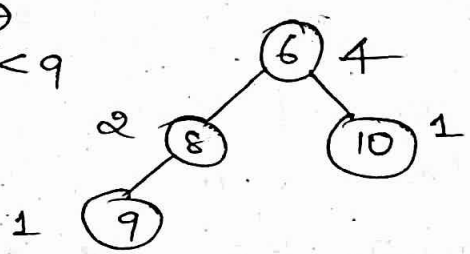
6 will be the root

if 8 comes to right child of 6 — condition fails as $6 + 2 > 1$

So swap

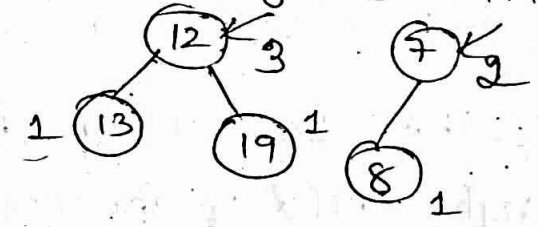


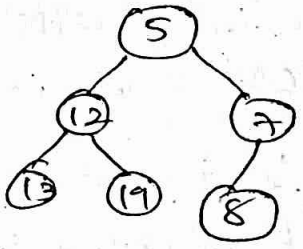
$8 < 9$



Weight based left ist tree

Deletion :- Deleting 5 in

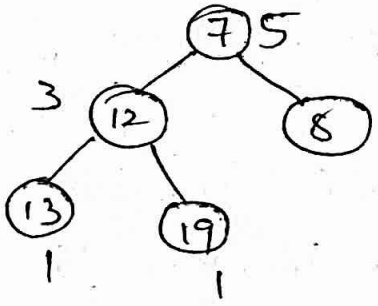




7 is smallest

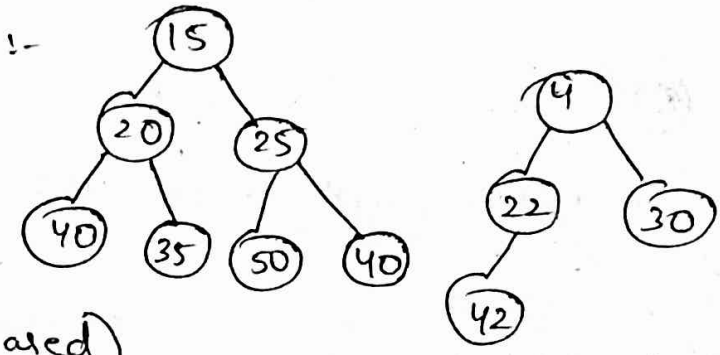
so 12 — right child of 7

Condition fails so swap



→ weight based left ist tree obtained.

HW Ex :-

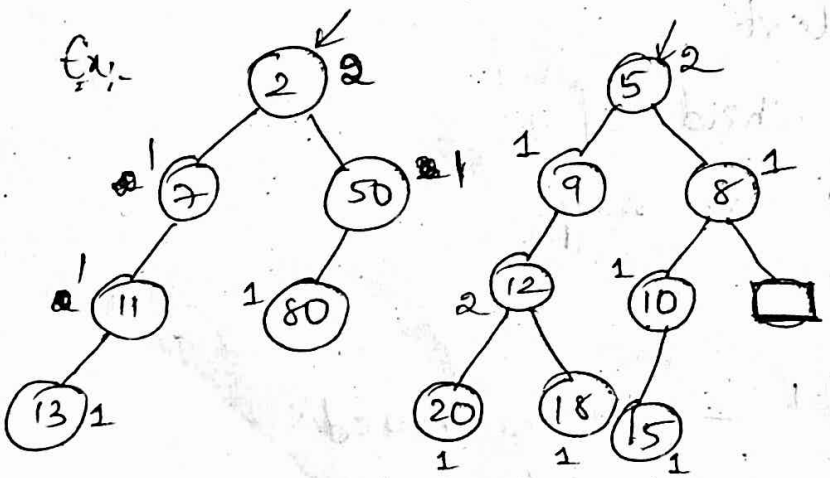


(Height based)

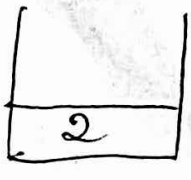
↳ Steps to Perform merging two left ist trees :-

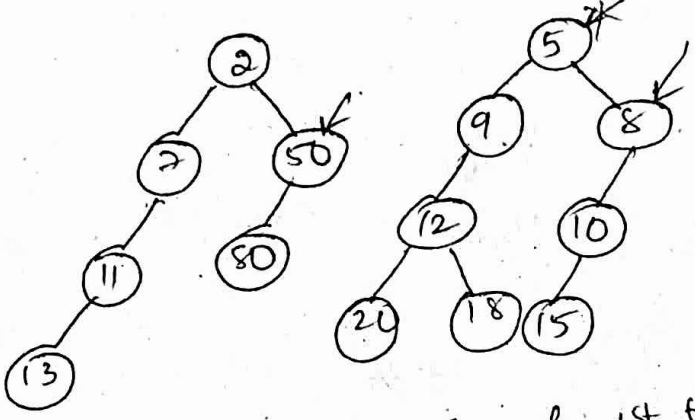
1) Compare the roots of two heaps
2) Push the smaller key into an empty stack and move to the right child of smaller key
3) Recursively compare 2 keys and move to its right child.
4) Repeat until a null node is reached.
5) Take the last Processed node and make it the right child of the node at top of the stack and Convert it into Leftist tree if the Properties of the Leftist tree is violated.
6) Recursively go on popping the elements of the stack and making them right child of the new stack.

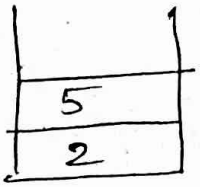
Ex :-



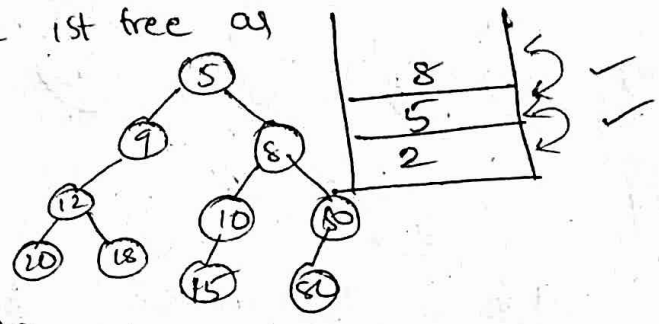
2 < 5 so push 2 into stack

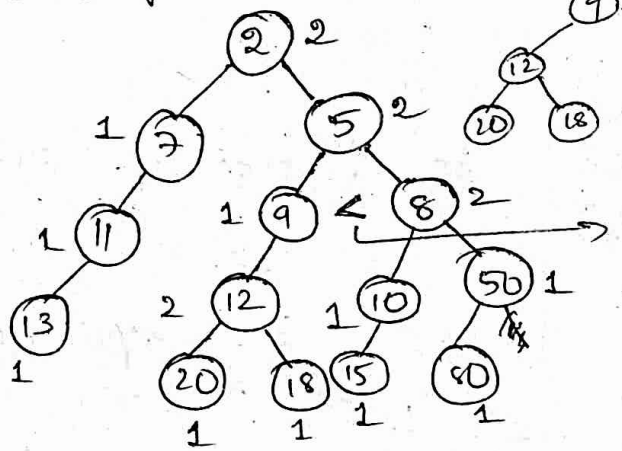


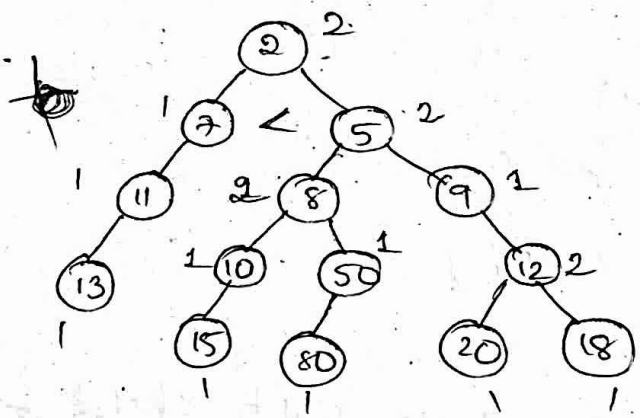




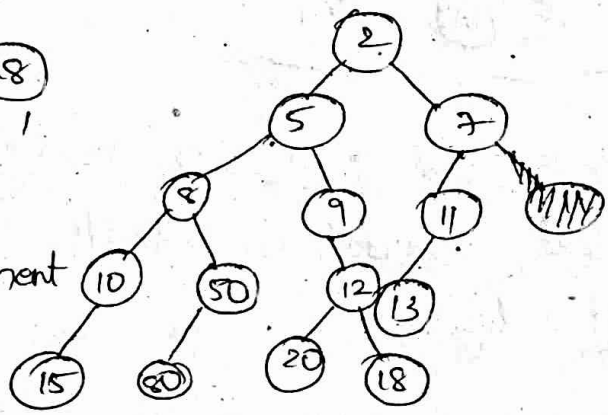
Make Right subtree of 1st tree as right child of 8





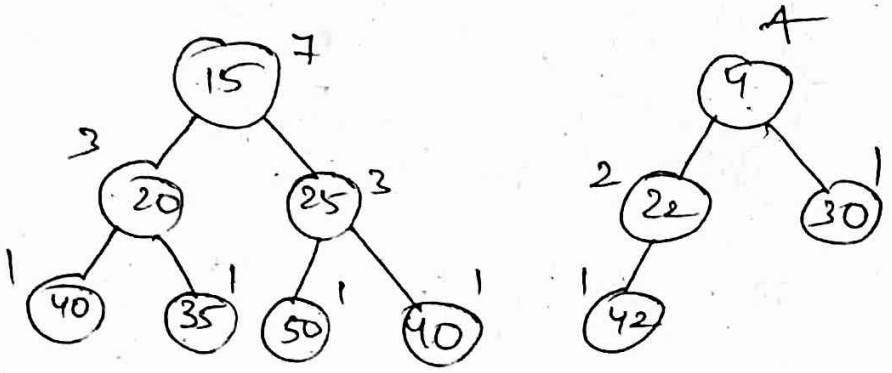


interchange to form left ist tree



Finding minimum element is - root node if it is a min heap.

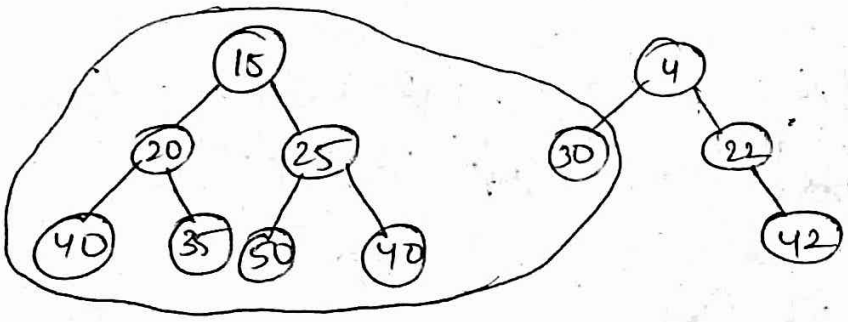
Weight based HB Problem:-



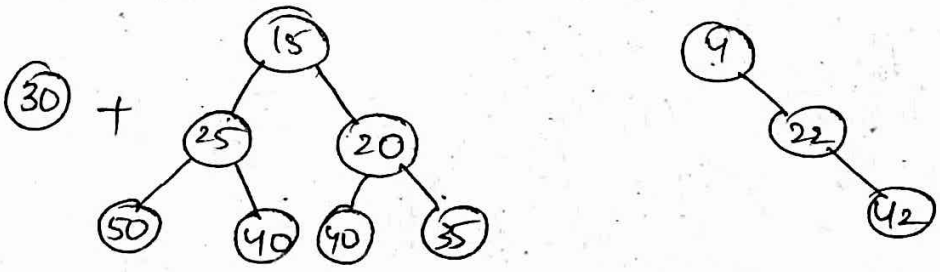
Merging :-

4 < 15 so make the 1st tree as right child of second tree

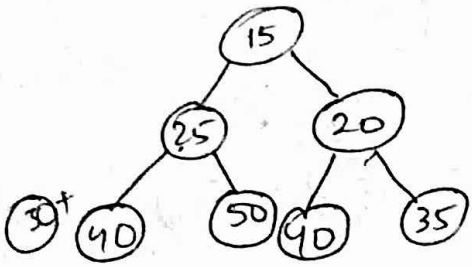
But condition is failed so swap the right & left sub-trees of 2nd tree

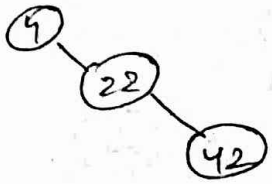


~~30 <~~ 30 > 15, so make 30 as right child of 15 but condition fails so swap the right & left subtrees

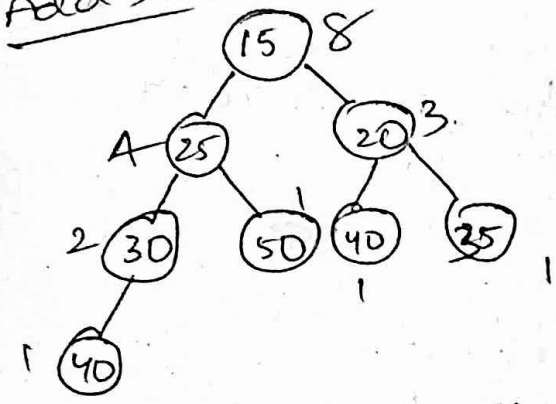


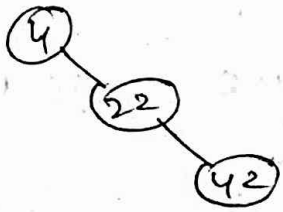
25 < 30, so right child of 25, ~~40 <~~ 40 > 30 but Condition fails so swap



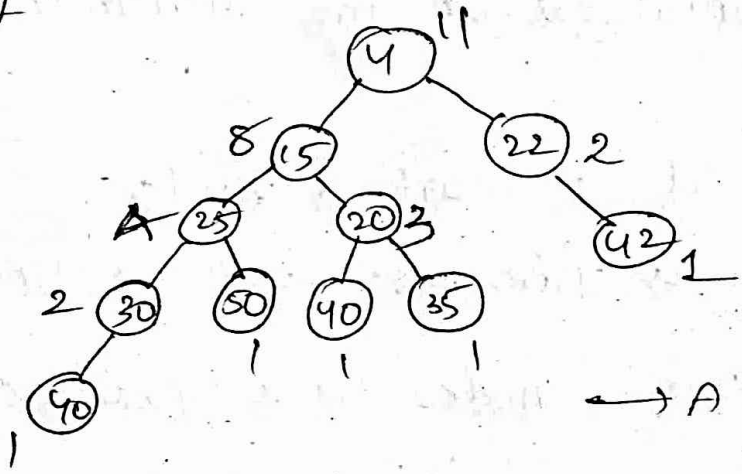


Add 30 :-





Merge :-

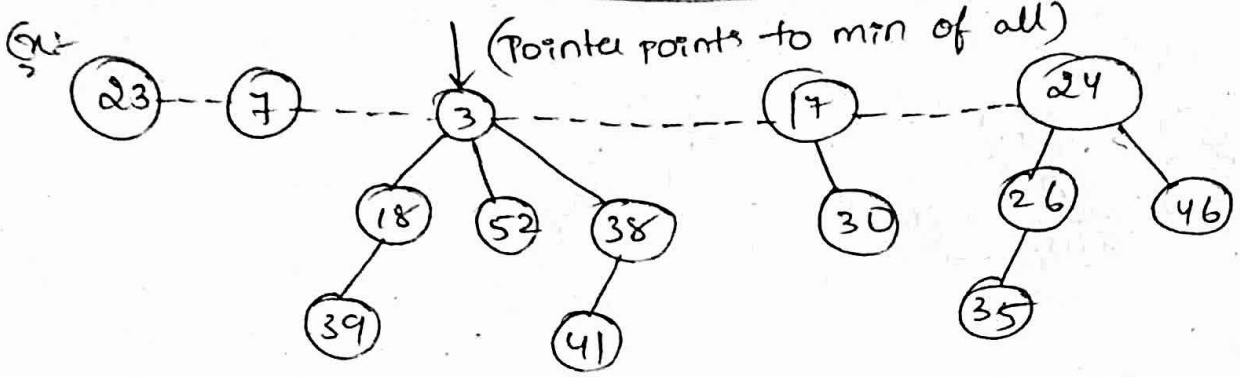


→ A weight-based leftist tree.

Fibonaci heap :-

It is a data structure ~~that~~ that consists of a collection of trees which follows either min heap or max heap properties.

→ In a Fibonaci heap a node can have more than two children or no children at all.



The trees are connected through Circular doubly linked list.

Properties of fibonaci heap :-

→ It is a set of min heap order trees.

→ A Pointer is maintained at the minimum element node.

→ It consists of a set of marked nodes

→ The trees with in a fibonaci heap are unordered

Memory representation of nodes in a fibonaci heap :-

→ Roots of all the trees are linked together for faster access.

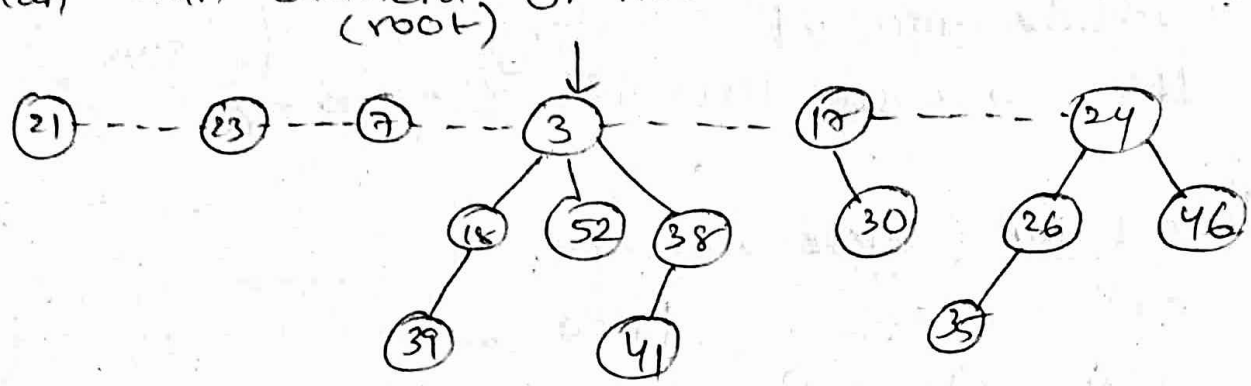
→ Child nodes of a Parent node are connected to each other through a circular doubly linked list.

(Benefit - It takes O(1) time for deleting a node if circular doubly linked list is used)

Insert Operation :-

Insert (21) to the above example

A new link is created to the new node and check whether the new node is greater than min element or not.



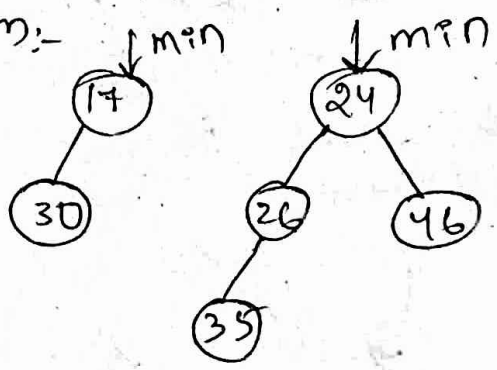
Steps for insertion :-

→ Create a new node for the element
→ check if the heap is empty or not
→ If the heap # is empty set the new node as the root node and mark it as minimum otherwise insert the node into the root list and update the min Pointer.

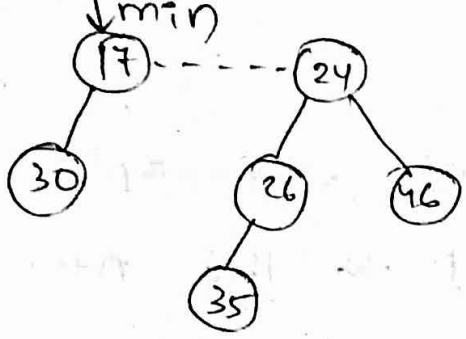
Find Min :-

Min element is always given by the minimum Pointer.

Union :-



Create circular doubly link b/w heaps and Point the Pointer to the min element & make it as root

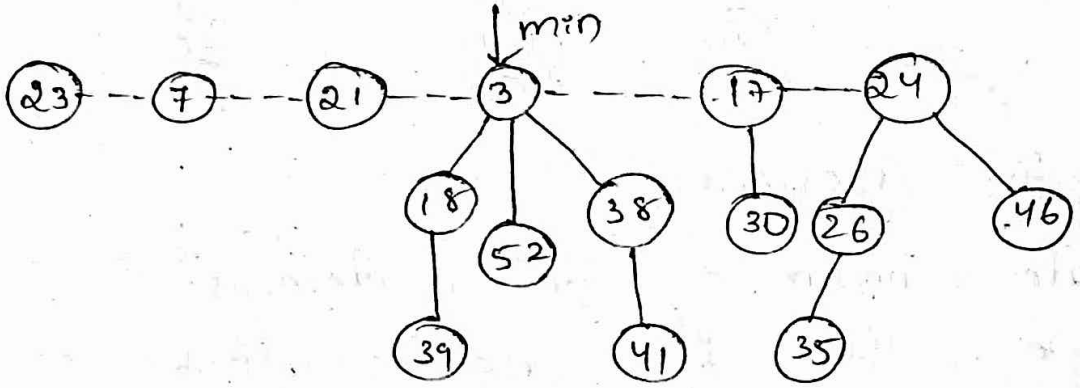


Steps:-

→ Concatenate both the roots of the heaps.

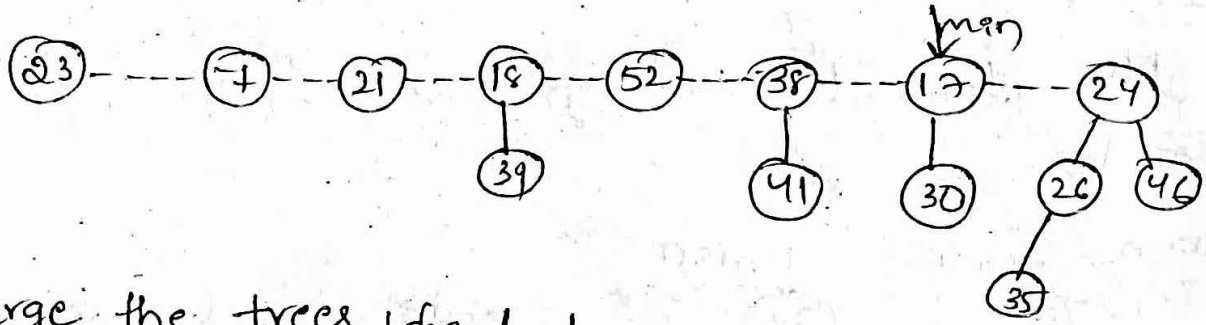
→ update min by selecting min of heap from the new root list that is formed.

Extract min (Delete a min) :-

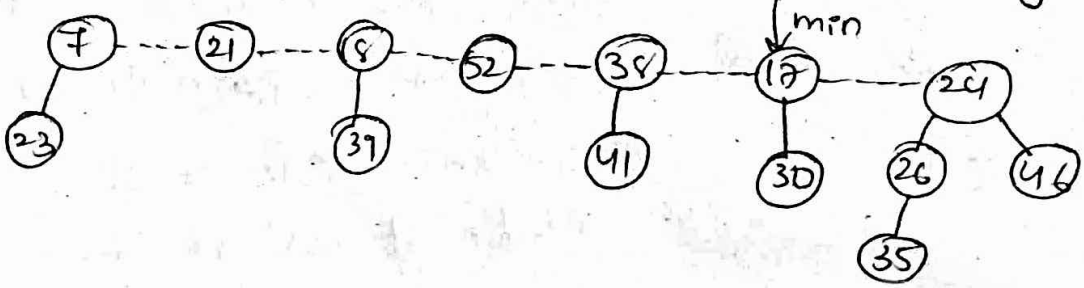


Node with minimum value is removed from the heap and the tree is readjusted.

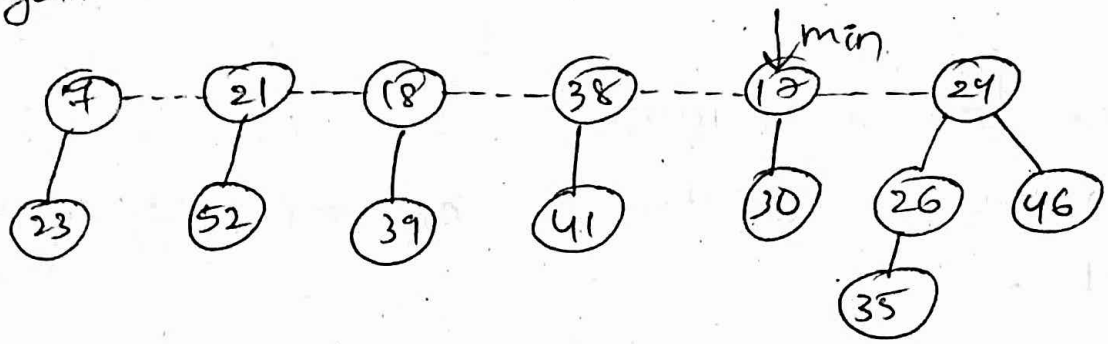
(After removing min make childrens of removed node as new heaps)



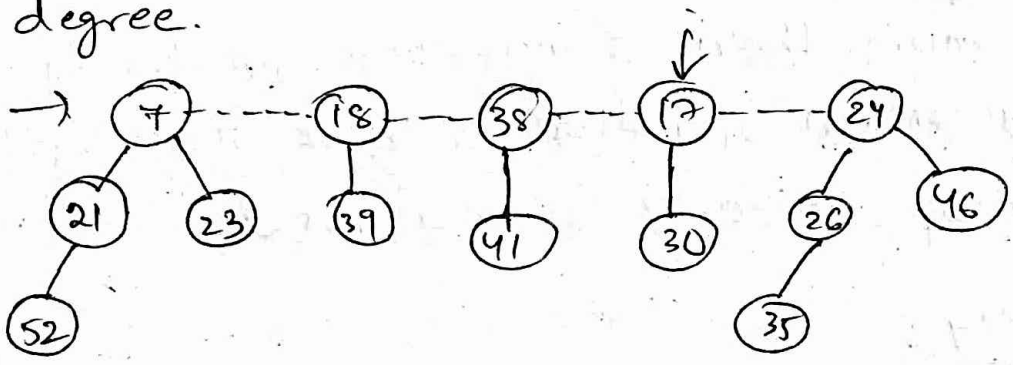
Merge the trees which have same degree

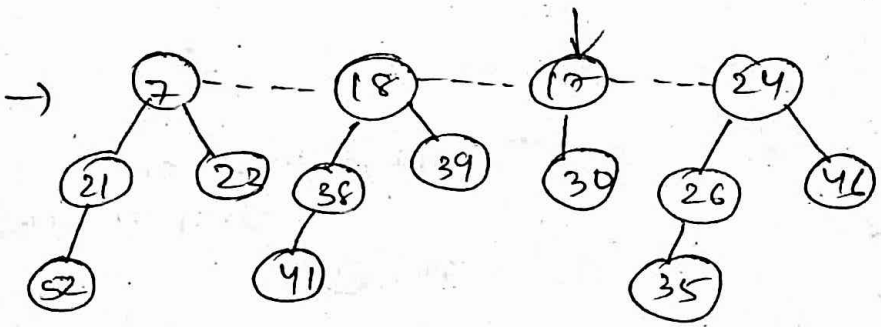


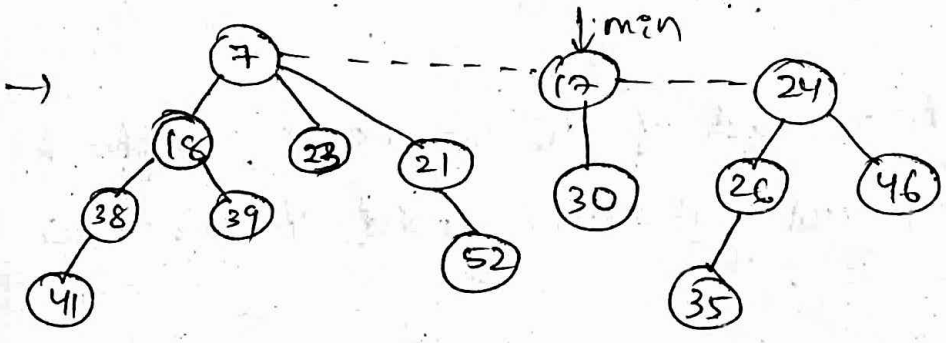
Again combine (21) & (52) which have degree 1



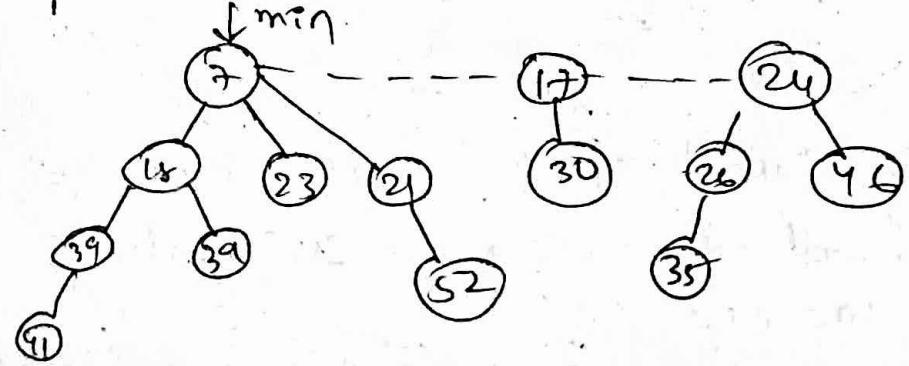
Now combine first two trees which have same degree.







Now every tree has different degree so update min pointer.



Steps :-

→ Delete the minimum node

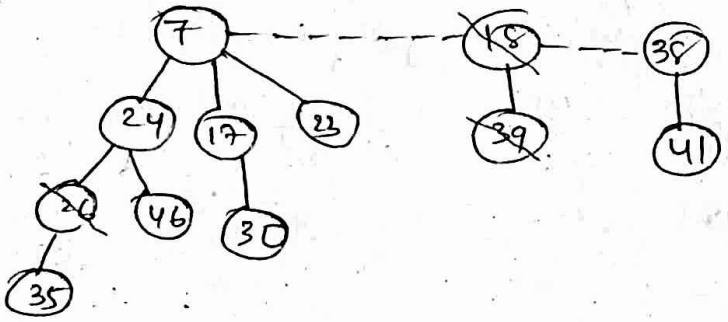
→ set the minimum pointer to the next root in the root list.

→ Do the following steps(a) until there are no multiple roots with same degree:

(a) If there are more than 2 mappings for the same degree apply union operation to those roots such that min heap property is mainted.

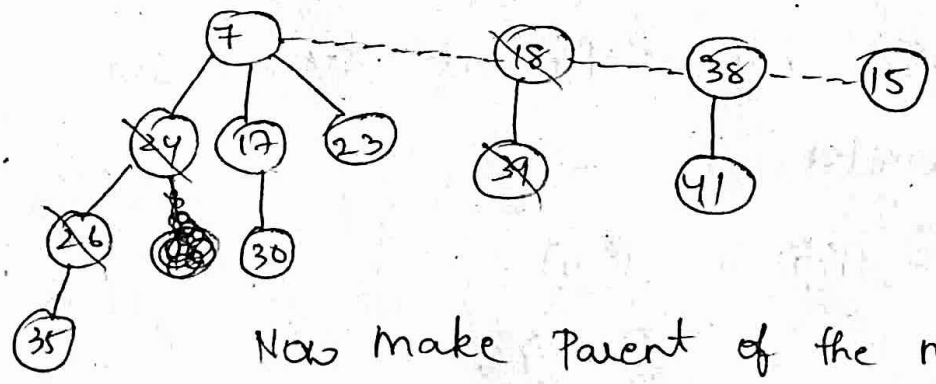
Decreasing a key :-

Decrease 46 to 15



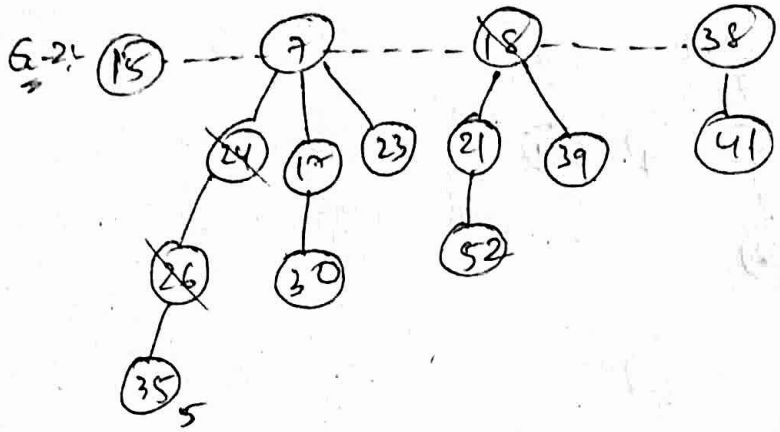
18, 39, 26 – marked nodes represented as ⊗

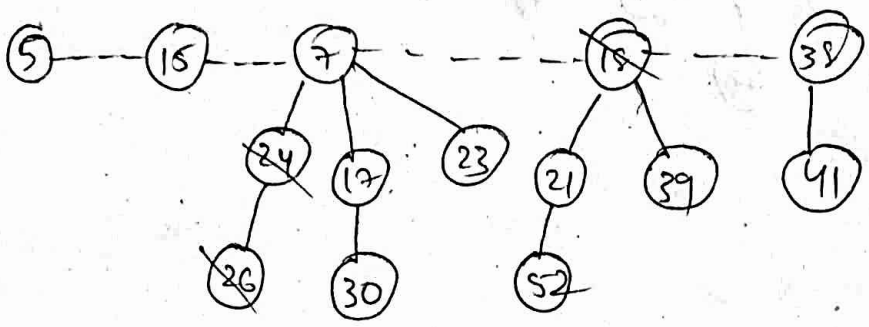
If 15 is placed instead of 46 we need to check min heap property if not satisfied add it as new node to the heap.



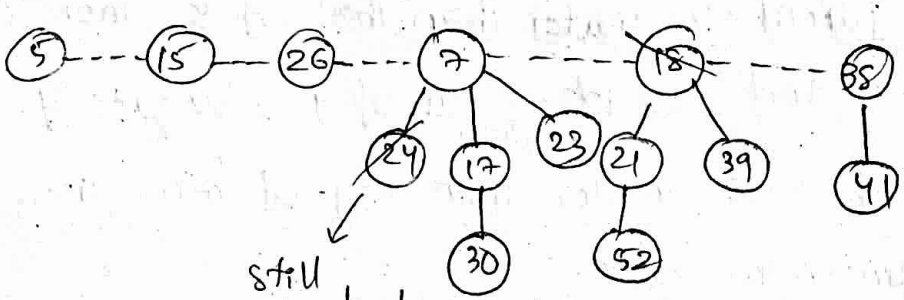
Now make Parent of the node which is removed and added as new tree as marked if it is not marked.

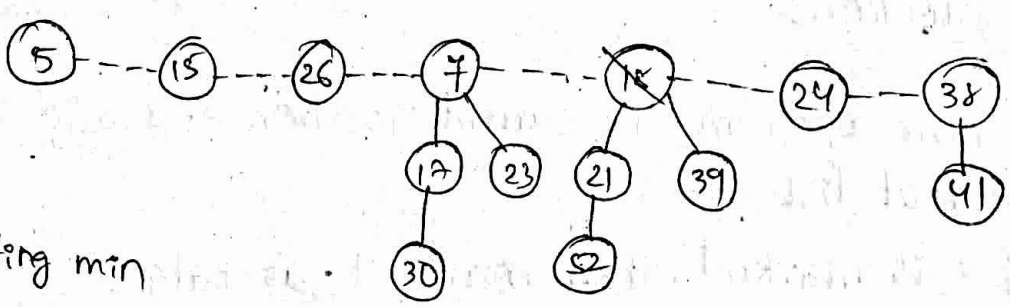
6.2: 35 to 5



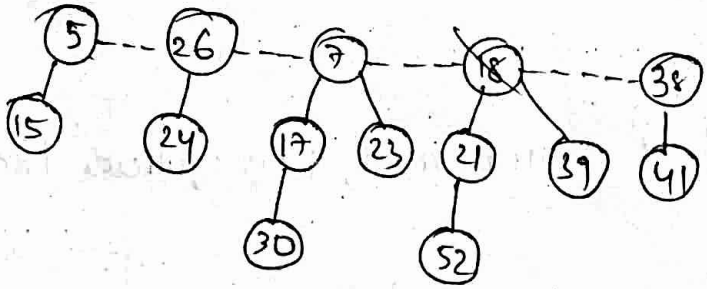


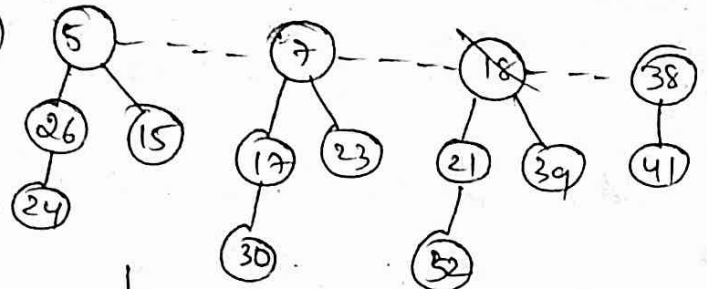
Here the parent of node which is added as new one ~~to~~ heap is marked so continue the Process until you reach unmarked node.

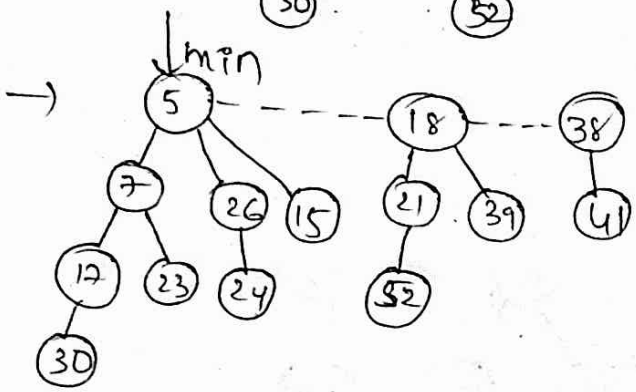




Extracting min



→ 

→ 

Steps:-

→ Select the node to be decreased (x) and change its value to the new value k.

→ If the parent of x i.e, y is not null and the key of parent is greater than that of k then call cut of x and cascading cut of y subsequently.

→ If key of x is smaller than key of min then mark 'x' as minimum value.

Cut function:-

→ Remove x from the current position and add it to the root list.

→ If x is marked, then mark it as False.

CASCADING CUT:-

→ If the Parent of y is not null, then follow the below steps.

Step 1:- If y is unmarked then mark y otherwise Call cut(y) and Cascading Cut of Parent of y.

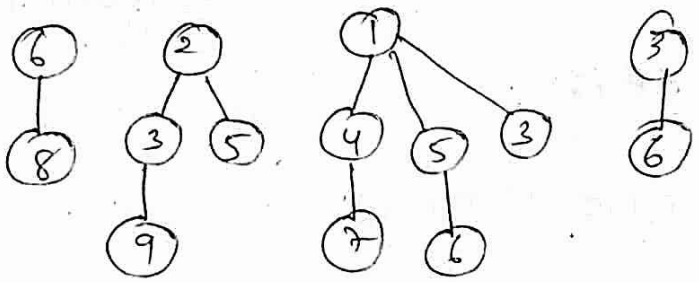
Pairing heap :-

→ It supports the same operations as supported by fibonacci heap.

→ A min Pairing heap is a min tree in which operations are performed in a sequence of steps.

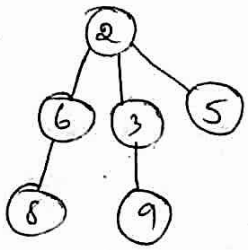
→ Pairing heap is a single tree which need not be a binary tree.

Eg :-

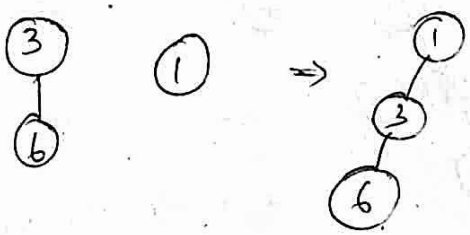


(a) (b) (c) (d)

Merging (a) and (b) :-



insert operation



Meld :-

Two min Pairing heaps are melted into a single min Pairing heap by Performing a compare link operation.

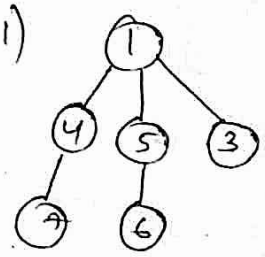
→ In Compare link, the roots of the two trees are compared and the min tree that has the largest root is made the left most subtree of other tree.

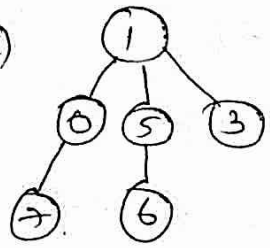
Insert:-

→ To insert an element x into a already existing Pairing heap we first create a Pairing heap q with single element x and then meld P & q.

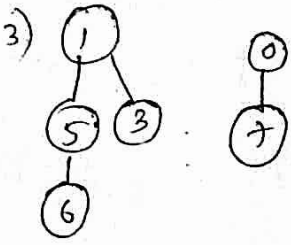
DECREASE KEY:-

Decrease 4 to 0.

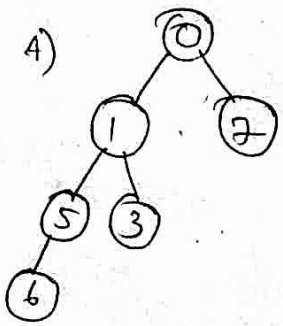
1) 

2) 

Detach the subtree  from the tree.

3) 

→

4) 

Steps:-

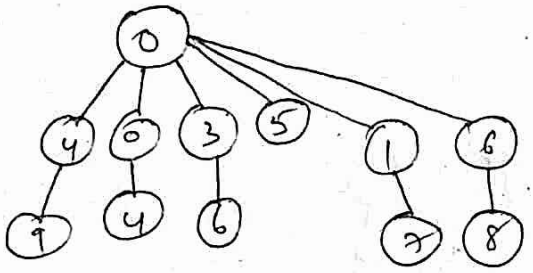
→ Remove the subtree with root n from the tree, this results in two min tree.

→ Meld the two min trees together.

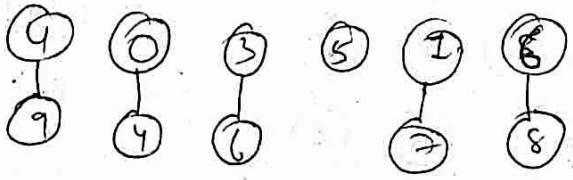
Delete minimum :-

To delete the minimum element, we first delete the root node, when the root is deleted ~~we~~ we are left with zero or more min trees.

→ When the no: of remaining min trees is 2 or more; these min trees must be melded into a single min tree by using twopass pairing heap scheme or multiple pass pairing heap scheme.

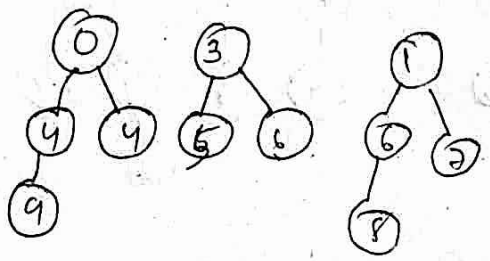


Deleting '0' results in 6 min trees.

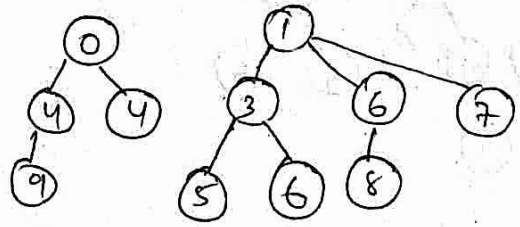


. Combine nodes ~~with same degree~~:

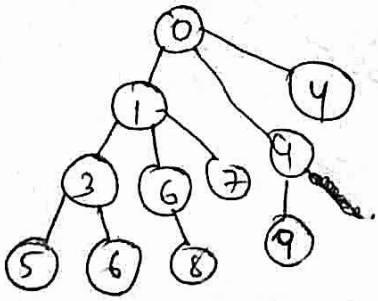
→



→



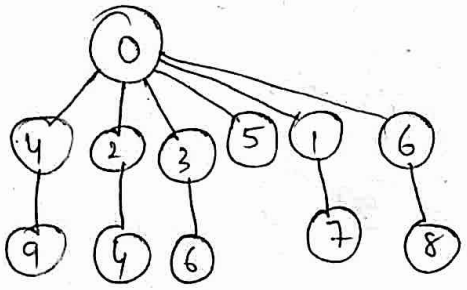
→



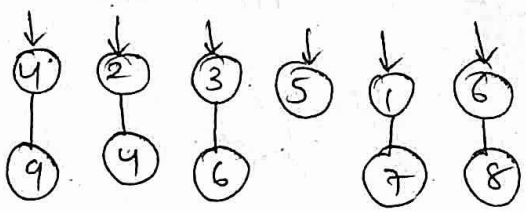
Multipass Pairing heap :-

The min trees that remain following the removal of the root are melded into a single min tree are as follows :-

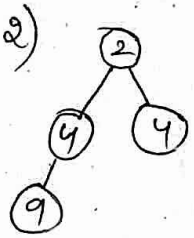
1) Put the mintrees into a FIFO Queue.
2) Extract two trees from front of the Queue, meld them and Put the resulting tree at the end of Queue. Repeat this step until only one tree remains



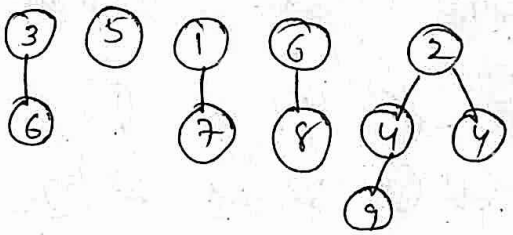
Step1 :-



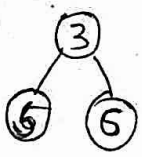
2)



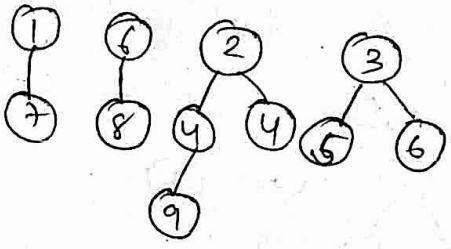
3)



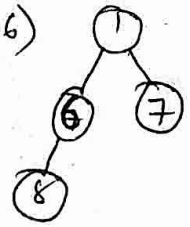
4)



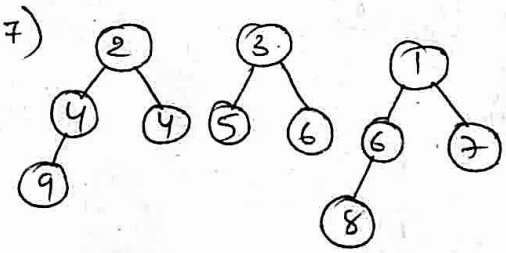
5)



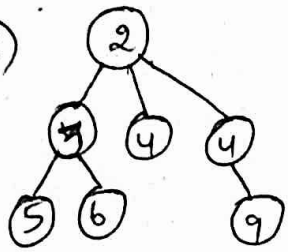
6)



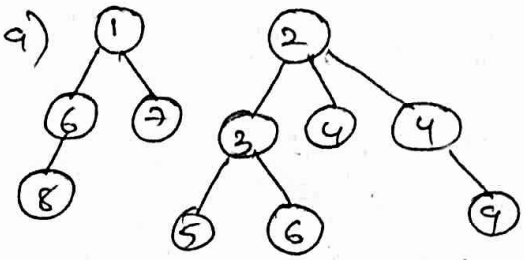
7)



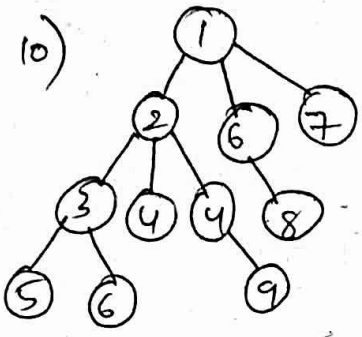
8)



9)



10)



Arbitarary delete :-

When N is not a root of the tree, deletion is done as follows :-

1) Detach subtree with root N from the tree.
2) Delete node N and melding subtrees into a single mintree using 2 Pass Pairing or multi Pass Pairing heap scheme
3) Meld trees from step1 and step2 into a single min tree.

UNIT-2

Hashing:-

It is an important data structure designed to solve the Problem of efficiently finding and storing data in an array.

→ With hashing in data structure, you can narrow down the search and find the number within seconds.

Definition of Hashing:-

Hashing is a datastructure which follows the technique of mapping a large chunk of data into a small tables using a hash function.

Hash function is also known as message digest function.

Hashing in a data structure follows a two step Process.

step 1 :- Hash function Converts the item into a small integer or hash value.

step 2 :- This integer is used as an index to store the original data.

you can use a hash key to locate the data quickly.

Hash table :-

Hashing in data structure uses hash tables to store the key-value pairs.

The hash table is partitioned into 'b' buckets

ht[0], ht[1], ht[2] . . . . . ht[b-1]

→ Each bucket is capable to hold 's' dictionary pairs.

→ Usually s=1 i.e., each bucket can hold only one pair.

→ The address or location of a pair whose key is k is determined by using a hash function 'h'

h[K] is the home address of k.

Key density of a hash table = $\frac{n}{T}$

Where 'n' is no:of pairs in the table
'T' is no:of possible keys.

Properties of hash function :-

1) Easy to compute.
2) It minimizes the no:of collisions.

Types of hash functions :-

1) Division :-

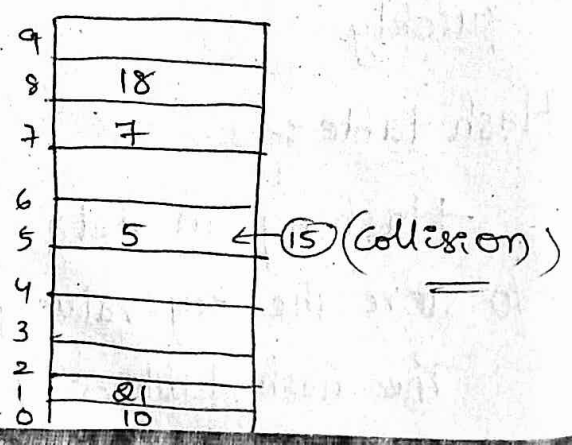
h(K) = Key % size K → key
(or)
h(K) = K mod size size → hash table size

Generally take size of table as Prime number to avoid collisions.

Ex:- 10, 5, 7, 21, 18, 15

table size = 10

10 % 10 = 0
5 % 10 = 5
7 % 10 = 7
21 % 10 = 1
18 % 10 = 8
15 % 10 = 5



Folding:-

* Key is divided into several parts, these parts are Combined or folded together and are often transmitted in a certain way to get the target or home address.

Two types :- 1) shift folding
2) Boundary folding.

Shift folding :-

Ex :- 12345789

3parts, tablesize = 1000

123 456 789 → Add these 3 Parts

123+456+789 = 1368

Apply module table size on the resultant value

1368 % 1000 = 368

∴ 12345789 can be hased to 368.

Boundary folding :-

Key is seen as being written on a piece of paper that is folded on the borders between different parts of the key, every other part ~~with~~ will be Placed in reverse order.

123456 → 2 Parts

123, 456

Revese ~~other~~ 2nd Piece - 456 to 654

123+654 = 777

777 % 1000 = 777.

1234578 12+43+56+87 = 198

198%1000 = 198.

Mid square function :-

In the midsquare method, the key is squared and the middle or mid part of the result is used as an address.

Ex :- Key = 18, table size = 10 = 10^{1} → these many no: of middle digits are to be extracted

$(18)^2$ = 324

So 2 is middle number.

~~2%10 = 2~~. 2 is address of 18.

2) $(3121)^2$ = 9740641

table size = 1000 = 10^{3}

406 is address of 3121.

If it is in binary format choose the table size as power of 2 & Extract the middle part of bit representation of square of the key.

Extraction :-

In the extraction method only a part of the key is used to compute the address. Each time only a portion of the key is used.

If this portion is carefully choosen, it can be sufficient for hashing.

Ex :- 1913A05N8 – last two numbers uniquely identify person so N8 – Key
↓
home address

Radix transformation:-

Key is transformed into another number system. the resultant value module table size is the resulting home address of the Particular key.

Ex:- 345 → 16 | 345
16 | 21 - 9
1 - 5

159 %1000 ⇒ 159
↓
home address of key

Collision-Resolution techniques:-

We have different methods ~~of~~ for reducing no: of collisions:- 1) open addressing.
2) Chaining
3) Bucket Addressing

Straight forward hashing is not without its Problem because for ~~atleast~~ almost all hash function with more than one key can be assigned to the same Collision.

This collision Problem can be solved by finding a function that distributes keys more uniformely in the table.

One more factor can contribute to avoiding conflict b/w the hashed keys is the size of the table.

1) Open Addressing:-

In the open addressing method when a key Collides with another key, the collision is

resolved by finding an available entry other than the Position to which colliding key is originally hashed.

→ If $h(K)$ is occupied then the Positions in the Probing sequence are $norm(h(K)+P(1)), \ldots$ $norm(h(K)+P(i))$ are tried until either an available cell is found or the same position are tried b/w or the table is full.

where P is a Probing function

i is a Probe

~~norm~~ norm is a normalization function.

Linear Probing :-

$P(i) = i \Rightarrow norm(h(k)+i)$

$(h(K)+i)$ % table size

Ex:- 4, 18, 22, 32, 33, 44, 59, 79

9	58
8	18
7	
6	44
5	33
4	4
3	33
2	22
1	
0	79

$32 \Rightarrow (2+1)\%10 = 3\%10 = 3$

$33 \Rightarrow (3+1)\%10 = 4\%10 = 4$

$(3+2)\%10 = 5$

$44 \Rightarrow (4+1)\%10$

$= (4+2)\%10$

$= 6$

$79 \Rightarrow (9+1)\%10$

$= 10\%10$

$= 0$

Disadvantage :-

* whenever you want to search a key it is applying linear search.
* Empty cells following clusters have a much greater chance to be filled than other positions.

Quadratic Probing :-

$P(i) = i^2$

$(h(K) + i^2)$ % table size

A5, A2, A3, B5, A9, B2, B9, C2

table size = 10

B5 → collision

B5 → $(5+1^2)$ % 10 = 6

A9 → 9

B2 → $(2+1^2)$ % 10 = 3

$(2+2^2)$ % 10 = 6

$(2+3^2)$ % 10 = 1

B9 → $(9+1^2)$ % 10

= 0

C2 → $(2+1^2)$ % 10 = 3 $(2+3^2)$ % 10 = 1

$(2+2^2)$ % 10 = 6 $(2+4^2)$ % 10 = 8

9	A9
8	C2
7	
6	B5
5	A5
4	
3	A3
2	A2
1	B2
0	B9

(Here Probing function P(i) is i^2)

Double hashing :-

This method utilizes two hash functions one for accessing the primary collision of a key and a second function for resolving the conflicts.

The Probing function for double hashing is

P(i) = i * hash2(K)
$(h_1(K) + i * hash2(K)) \% tablesize$

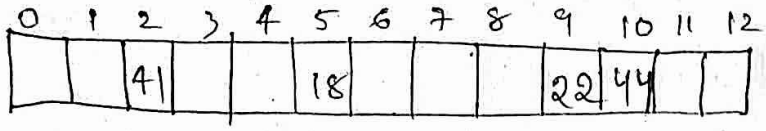
~~hash2(x)~~

If better function for hash2 is R − X Mod R where R is a Prime number and R < table size.

hash2(x) = R − X Mod R

Eg:- 18, 41, 22, 44

Table size = 13



hash1(x) = x mod 13

hash2(x) = 7 − x mod 7 (take R = 7)

18 % 13 = 5

41 % 13 = 2

22 % 13 = 9

44 % 13 = 5 → collision

$(hash1(x) + i * hash2(x)) \%$ table size

$(5 + 1 * [7 - 44 \bmod 7]) \%$ ~~table~~ 13

$(5+5) \% 13 = 10 \% 13$

$= 10$

(i = 1 ∵ it is the first collision for 44)

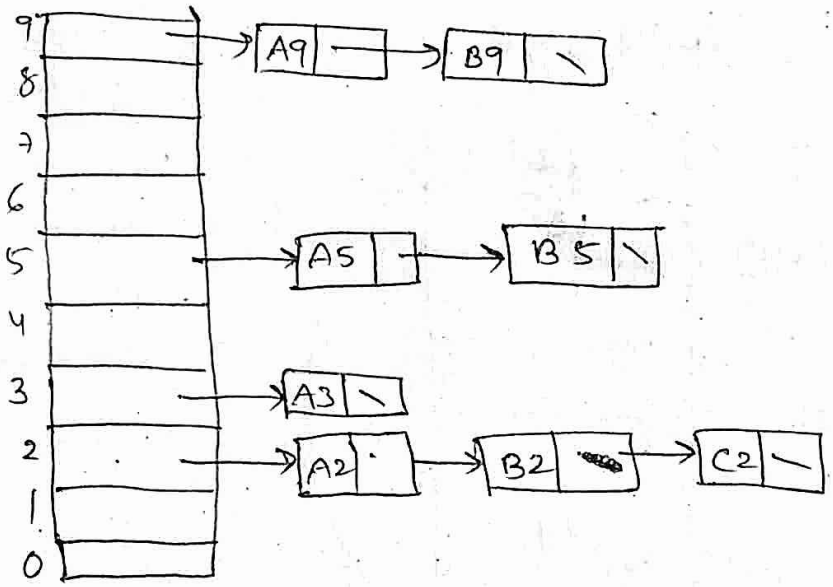
Chaining:-

In this technique, keys do not have to be stored in the table itself

→ In chaining, each position of the table associated with a linked list (or) chain of structures whose information fields store keys (or) reference to the keys.

This method is called seperate chaining and a table of references is called scattered table.

Eg:- A5, A2, A3, B5, A9, B2, B9, C2



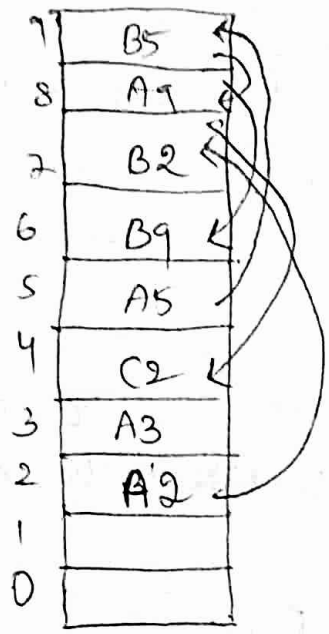
In chaining colliding keys are put on the same linked list

→ It requires additional space for maintaing Pointers. This is a normal chaining technique.

→ There is another version of chaining called Coalesced hashing, Combines linear Probing with hashing.

Coalesced hashing puts a colliding key in the last available position of the table.

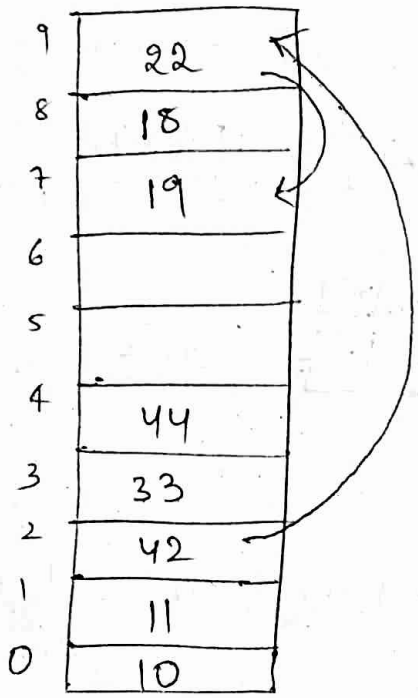
Ex-1:-



A5, A2, A3, B5, A9, B2, B9, C2

Ex-2:-

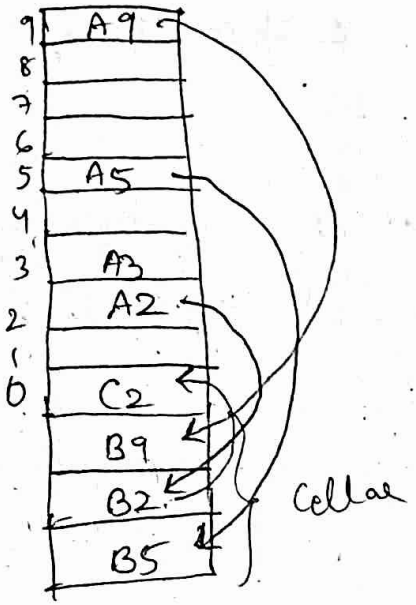
18, 42, 33, 11, 10, 22, 44, 19 Table size = 10.



Coalesced hashing that uses a cellar:-

An overflow area known as cellar can be allocated to share keys for which there is no room in the table. This area should be located dynamically.

Eg:- A5, A2, A3, B5, A9, B2, B9, C2



Bucket Addressing:-

Another solution to Collision Problem is to store colliding elements in the same position in the table.

→ A bucket is a block of space large enough to store multiple items.

→ If a bucket is already full, then an item hashed it it has to be stored somewhere else.

→ By incorporating the open addressing approach, the Colliding item can be stored in the next bucket if it has an available slot when using linear Probing or it can be stored in some other bucket by using quadratic Probing

Eg:- A5, A2, A3, B5, A9, B2, B9, C2

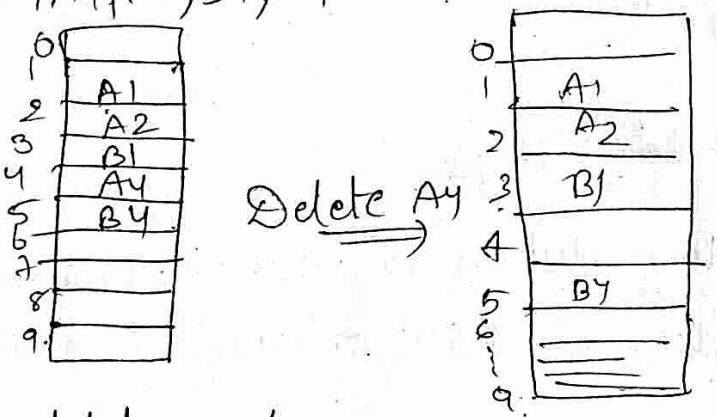
9	A9	B9
8		
7		
6		
5	A5	B5
4		
3	A3	C2
2	A2	B2
1		
0		

} Linear Probing.

Deleting an element leads to the deletion of a node from the linked list holding the element by using Chaining method.

For other methods, a deletion operation may require a more careful treatment of collision resolution.

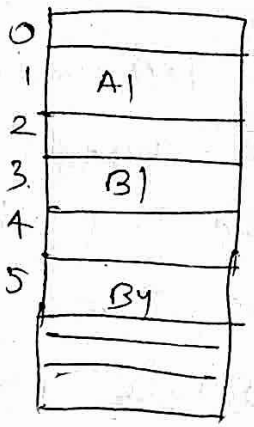
Eg:- A1, A4, A2, B4, B1



After the deletion of A4 we try to find B4 by first checking at position 4

But this position is empty, so we may conclude that B4 is not in the table

Deletion of A2



After the deletion of A2, when we are trying to search for B1, it is unsuccesfull because we are using linear Probing, search terminates at Position 2

So the solution for the above Problem is Purge

the keys to the previous available slots.

PERFECT HASH FUNCTION :-

In many situations the keys are known before hand, and a hash function can be devised after the data are known, such a hash function may be really be a Perfect hash function, if it hashes items on the first attempt itself.

Such function requires only ~~as~~ many cells in the table as the no: of data, so that no empty cell remains after the completion of hashing is called Perfect hash function.

Chichelli's Algorithm :-

It is a type of brute force algorithm It is used to hash ~~function~~ a relatively small number of reserved ~~codes~~ words.

Step1 :- Get the count of starting & ending characters.

Step 2 :- According to the frequencies the words Can be put in the sorted order.

step3 : you need to calculate the hash function of the Particular word.

$$h(word) = (length(word) + g(firstletter(word)) + g(lastletter(word))) \% tablesize$$

c=2 Calliope → fre(c)+fre(e) = 2+6 = 8

e=6 Clio → " + fre(o) = 2+2 = 4

o=2 Erato → 6+2 = 8

M=1 Euterpe → 6+6 = 12

P=1 Melpamene → 1+6 = 7

a=3 Polyhymnia → 1+3 = 4

T=2 Terpsichore → 2+6 = 8

U=1 Thalia → 2+2 = 4

Urania → 1+3 = 4

Arrange according to frequency sum

	length	g(first letter)	g(last letter)	
Euterpe	7	0	0	(7+0+0)%9 = 7
Calliope	8	0	0	(8+0+0)%9 = 8
Erato	5	0	0	(5+0+0)%9 = 5
Terpsichore	11	0	0	(11+0+0)%9 = 2
Melpomene	9	0	0	(9+0+0)%9 = 0
Thalia	6	0	0	(6+0+0)%9 = 6
Clio	4	0	0	(4+0+0)%9 = 4
Polyhymnia	10	0	0	(10+0+0)%9 = 1
Urania	6	0	0	(6+0+0)%9 = 6 ↓ collision
Urania	6	1	0	(6+1+0)%9 = 7 ✗
	6	2	0	(6+2+0)%9 = 8 ✗
	6	3	0	(6+3+0)%9 = 0 ✗
(we can increment g(first letter) only upto table size/2) 9/2 ≈ 4	6	4	0	(6+4+0)%9 = 1 ✗
	~~6~~	~~5~~	~~0~~	~~(6+5+0)%9 = 8 ✗~~
Polynomia	10	1	0	(10+1+0)%9 = 2 ✗
	10	2	0	(10+2+0)%9 = 3 ✓

Goback to urania

Urania	6	0	0	(0+0+0)%9 = 6 ✗
	6	1	0	(6+1+0)%9 = 7 ✗
	6	2	0	(6+2+0)%9 = 8 ✗
	6	3	0	(6+3+0)%9 = 0 ✗
	6	A	0	(6+4+0)%9 = 1 ✓

The function g assigns values to letters so that the resulting function h returns unique hash values for all words in a predefined set of words.

Initially take g(first character) of the corresponding word equals to 0 and g(last character) of the Corresponding word equals to 0.

g(first char) can increase to k where k less than or equal to tablesize/2 when collision occurs.

g(first char) must be increased till k whenever collision occurs, still if the word does not fit then backtrack by changing g values of previous words until k, still if the word doesn't fit then increment the g(last character) from where collision had started.

Hash function for Extendible files:-

New techniques have been developed specifically taken into account the variable size of the table or file.

We Distinguishing two classes of such techniques:-

1, Directory schemes.

2, Directoryless schemes.

Extendible hashing:-

Accessing the data stored in the buckets indirectly through an index.

Global depth:- It is associated with the directories which denote the, no: of bits used by the hash function to categorize the keys.

Each bucket has a local depth associated with it, that indicates the no: of rightmost bits in h(k).

The rightmost bits are same for all the keys in the bucket

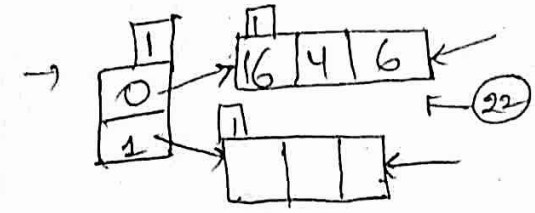
When local depth is equal to depth of the directory, it is necessary to change the size of the directory, after the bucket is split in case of overflow.

When the local depth is smaller than the depth of the directory, splitting the bucket only ~~differs~~ requires changing half of the pointers pointing to this bucket.

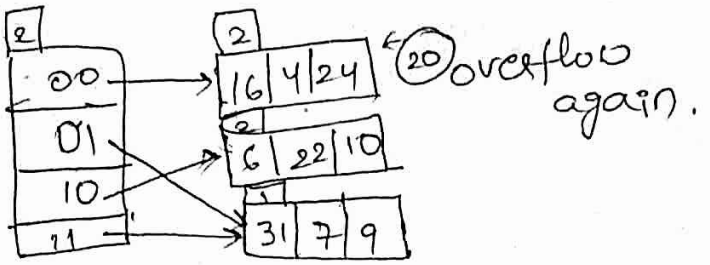
Ex:- 16, 4, 6, 22, 24, 10, 31, 7, 9, 20, 26.

Local depth = 1, global depth = 1 bucket size = 3

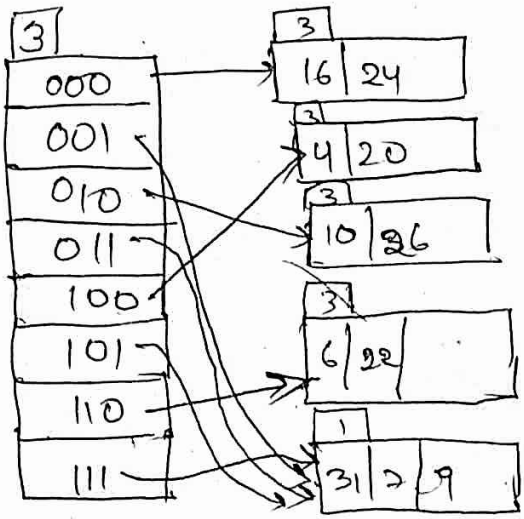
16 - 010000
4 - 000100
6 - 000110
22 - 010110
24 - ~~011~~ 011000
10 - 001010
31 - 011111
7 - 000111
9 - 001001
20 - 010100
26 - 011010



take localdepth = 2



take local depth = 3



Advantages :-

→ It avoids reorganization of entire file if directory overflows.

→ Data retreival is very fast

Disadvantages :-

→ Bucket size is fixed.

→ Memory is wasted in Pointers.

→ The directory size may increase significantly if several records are hashed on to the same directory.

UNIT - III

Red Black trees :-

In the binary search tree, normally searching time is O(logn), but in skewed trees the time complexity for searching an element is O(n) in worst case.

Inorder to reduce the time complexity for searching a node we use self balancing tree called as AVL tree, which takes o(logn) for search operation.

→ But inorder to balance the tree we need to perform rotations and we might need to perform rotations for all nodes between leaf node and the root node.

So there is a need of data structure which performs better than AVL tree for performing inserting and deleting operations of a node in binary search tree. That tree is called as "Red Black tree"

A tree which follows the following properties are known as Red Black trees :-

1) The tree should be a binary search tree
2) Every node in the tree should be either black or red.
3) The root node and all the external nodes of the tree are coloured with Black.
4) If a node is red, then its children should be Black that means no two consecutive nodes

should be in red colour.

5) Every path from a node to any of its descendent null nodes have same no. of black nodes

Insertion in a Red black tree:-

step-1:- If the tree is empty, then create a new node and colour it as black.

step-2:- If tree is not empty, create a new node as leaf node and colour it as red.

step-3:- a) If parent of new node is black then insertion of new node is completed.

b) If the parent of new node is red, then check the colour of parent sibling of new node.

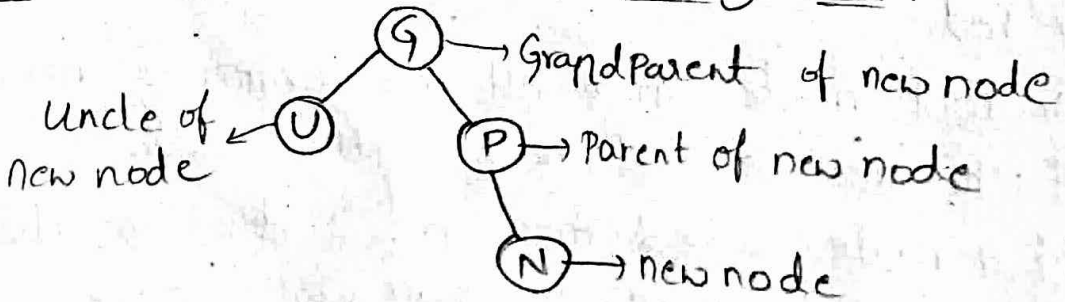
i) If the colour is red, we recolour the new node, Parent, sibling & grandparent.
(If grandparent is root node, we cannot recolour it)

ii) If the colour is Black or Null, Perform suitable rotation and recolour

1→ If the Parent of new node, new node and grandparent of new node form a triangle, then rotate parent in opp direction of new node

2→ If the new node, Parent of new node and grandParent of new node form a straight line, then rotate grandparent in opp direction of new node, recolour original Parent & grand Parents

Relationship of new node with existing nodes:-

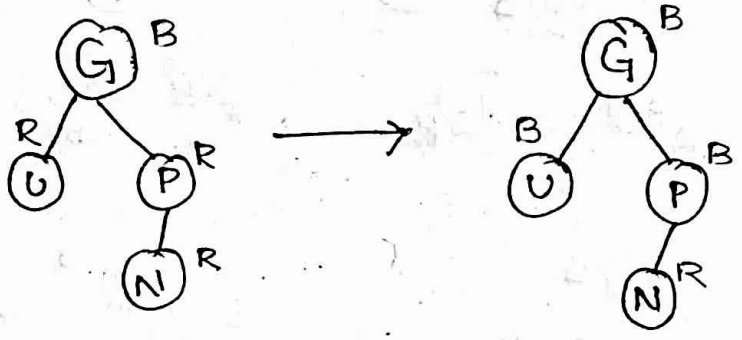


Four Cases :-

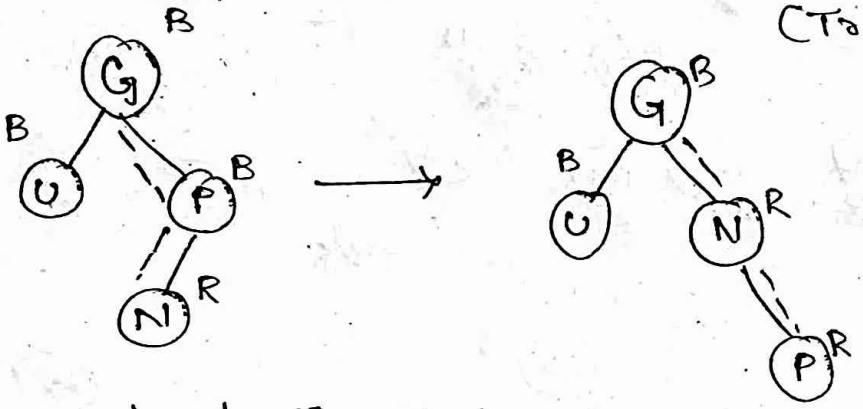
Case 0 :- Newly inserted node is root node.

~~Case 1~~

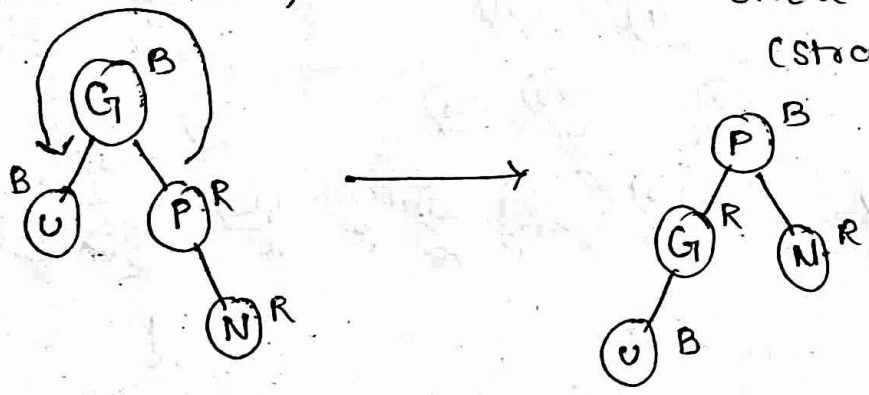
Case 1 :- Newly inserted node's uncle is red



Case 2 :— Newly inserted node's uncle is black (Triangle)



Case 3 :— Newly inserted node's uncle is black (straight line)



Ex:- 25, 14, 45, 51, 55, 40, 32, 60, 57, 12, 13