* ADVANCED DATA STRUTURES:

→ Binary search Tree:

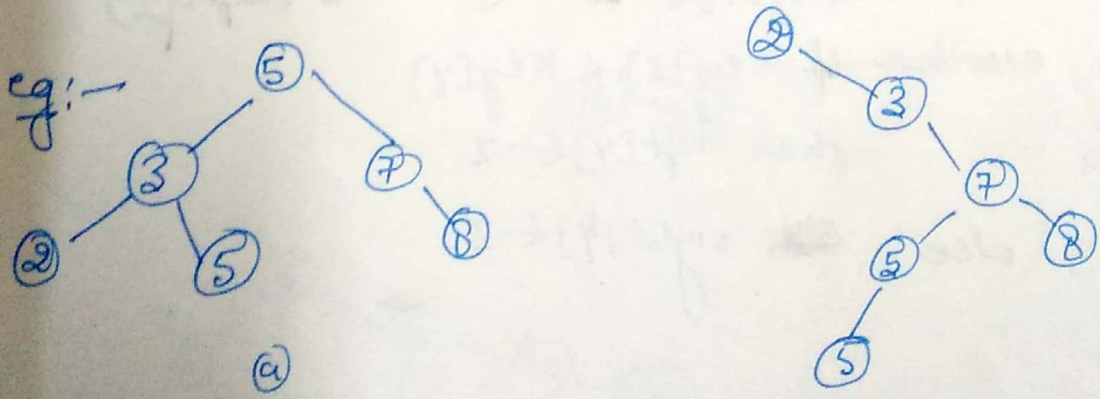
A binary search tree is organized in a binary tree. Each node contains the fields - left, right, key, and p that points to the nodes corresponding to its left, right, and parent of key is the value stored in a node.

→ The keys in a binary search tree are always stored in such a way as to satisfy the binary-search-tree property:

→ let x be a node in a binary search tree. If y is a node in the left subtree of x then key[y] ≤ key[x]

→ If y is a node in the right subtree of x then key[x] ≤ key[y]

eg:-



(a)

Querying a binary search tree :

① Searching ② Minimum & Maximum.

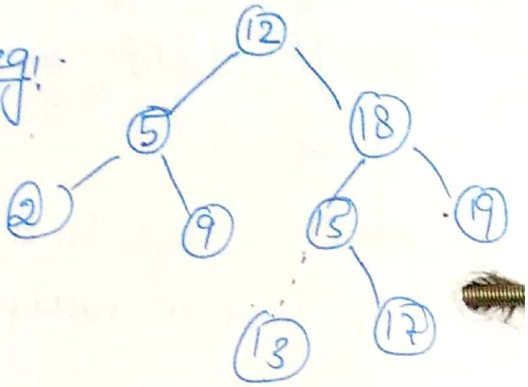
③ Successor & predecessor ④ Insertion ⑤ Deletion

→ Insertion :

TREE-INSERT (T, z)

1. y ← NIL
2. x ← root[T]
3. while x ≠ NIL
4. do y ← x
5. if key[z] < key[x]
6. then x ← left[x]
7. ~~else~~ x ← right[x]
8. p[z] ← y.
9. if y = NIL
10. then root[T] ← z (Tree is empty)
11. else ~~else~~ if key[z] < key[y]
12. then left[y] ← z
13. else ~~else~~ right[y] ← z.

eg.



Deletion:

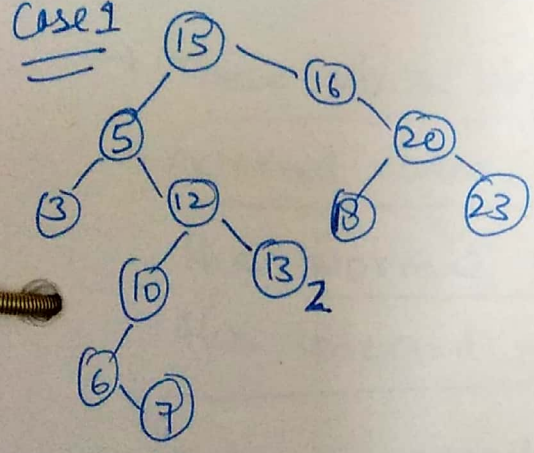
In case of deletion in Binary Search tree, there are three cases: If z is the node to be deleted.

Case 1: If z has no children – we modify its parent P[z] to replace with NIL as its child.

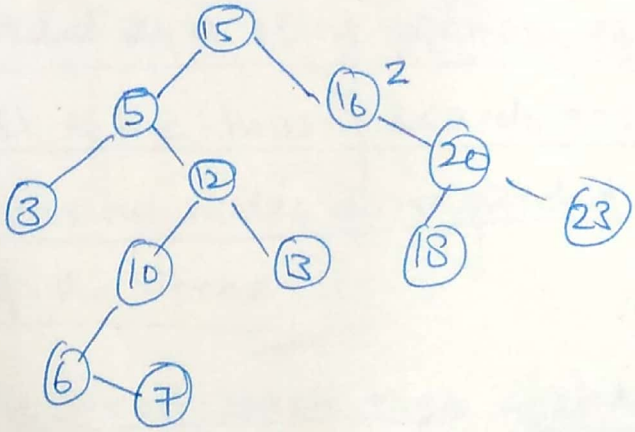
Case 2: If the node has a single child, we "splice out" z by making a new link between its child & its parent.

Case 3: If a node has two child – we "splice out" z's successor y, and replace z's key and satellite data with y's key and satellite data

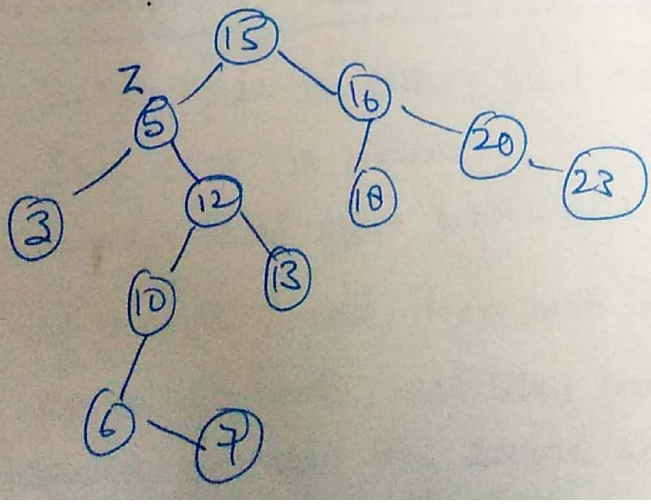
Case 1



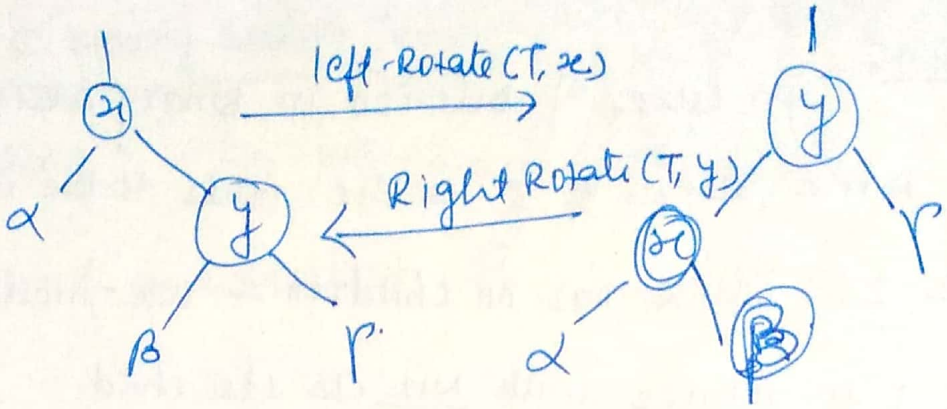
case 2



Case 3



Rotations:



⟨ Red-Black Trees ⟩

→ A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either ~~red~~ RED or BLACK

→ Nodes can be colored on any path from the root to leaf,

→ A red-black tree ensures that no such path is more than twice as long as other, so that the tree is approximately balanced.

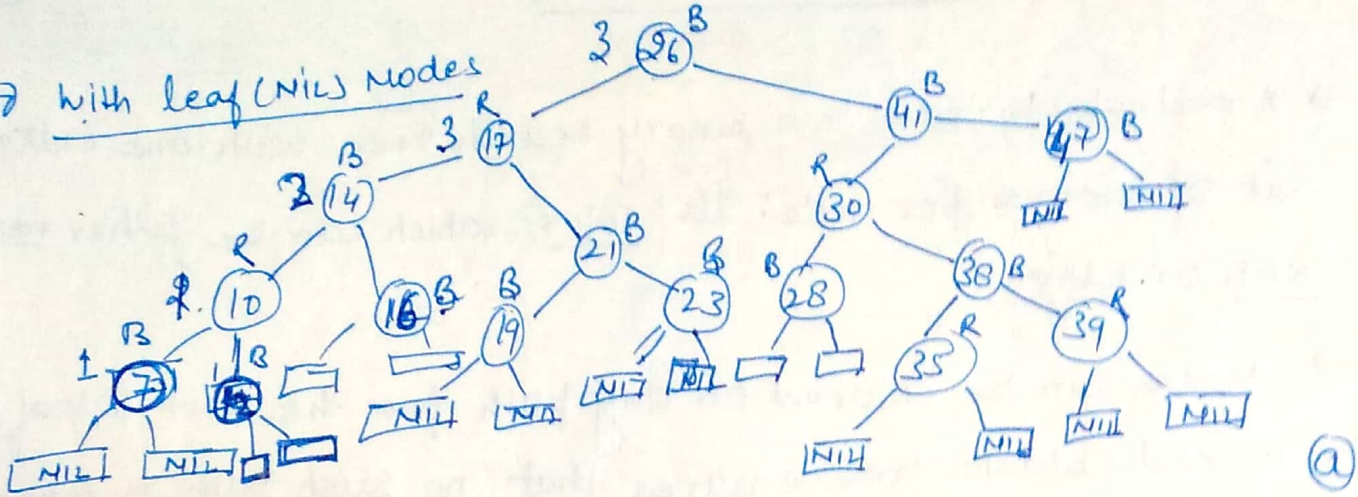
→ Each node of the tree now contains the fields: color, key, left, right, and p. If a child or the parent of a node does not exist the corresponding ptr field ~~cont~~ of the node contains the value NIL

→ These NIL's are regarded as being pointers to the external nodes (leaves) of the binary search tree and the normal, key bearing nodes as regarded as the internal nodes of the tree

⊛ A binary search tree is a red-black tree if it satisfies the following red-black properties:

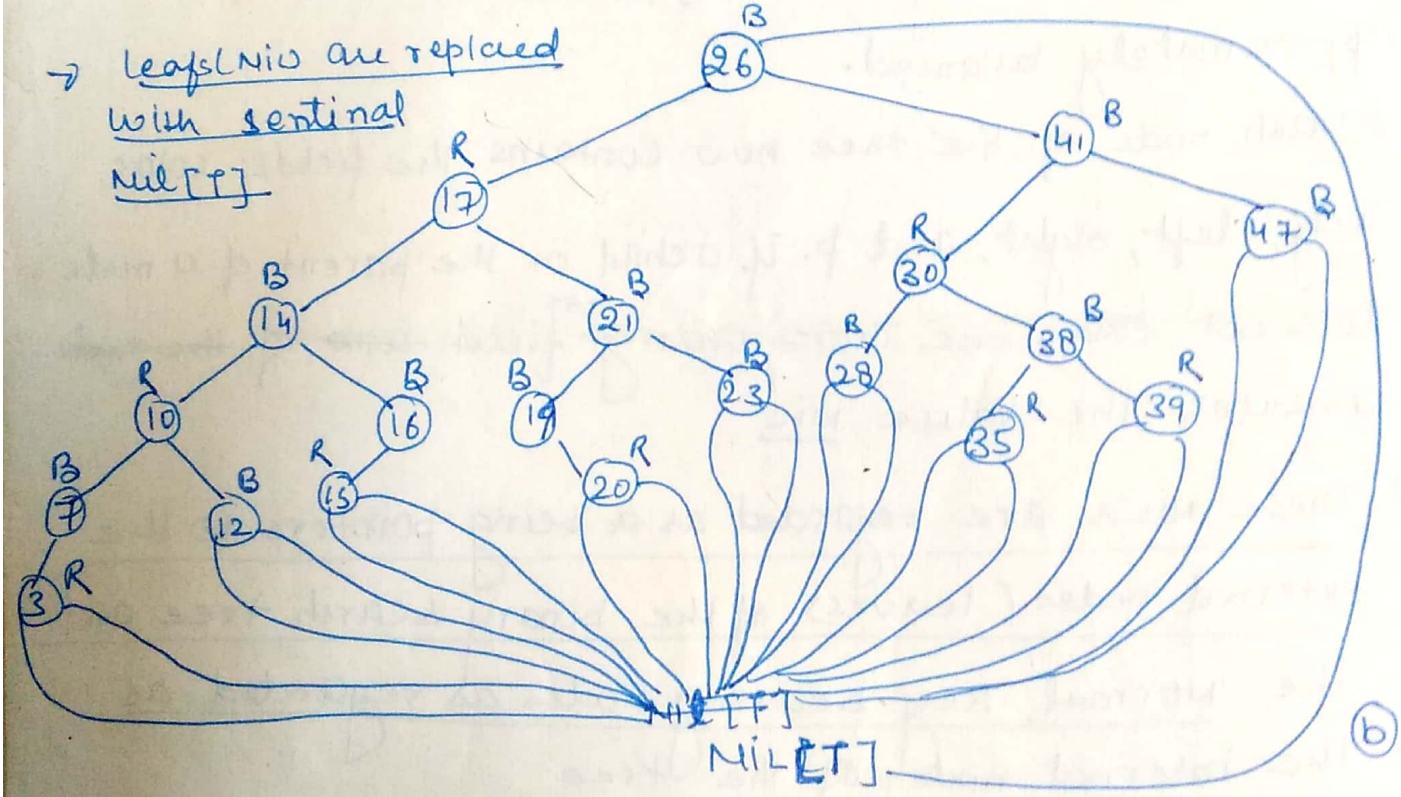
1) Every node is either red or black
2) The root is black
3) Every leaf ~~(NIL)~~ (NIL) is black.
4) If a node is red, then both its children are black
5) For each node, all paths from the node to descendant leaves contain the same number of black nodes.

→ With leaf (NIL) Nodes



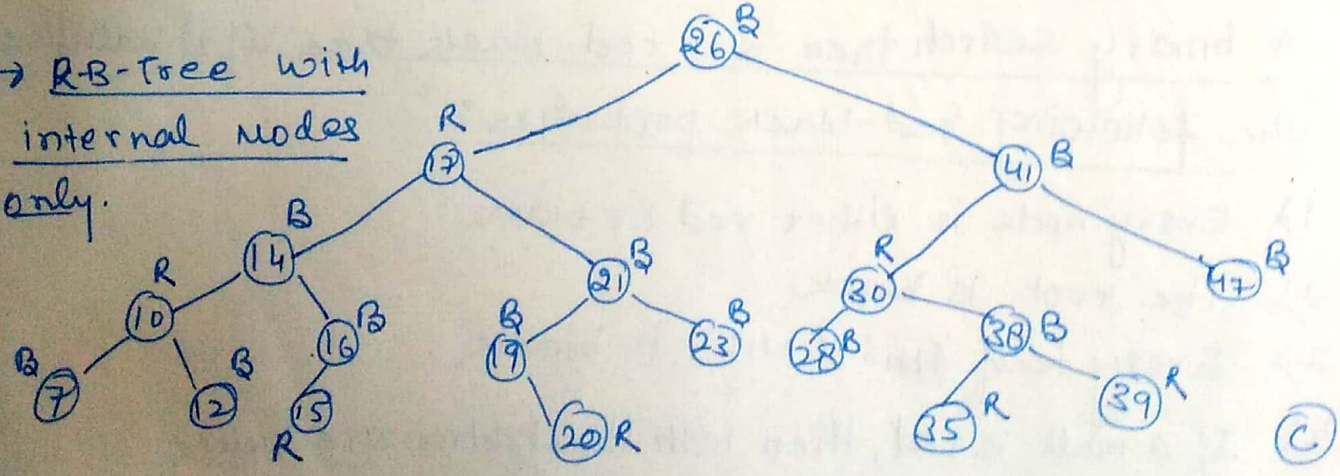
(a)

→ Leafs (NIL) are replaced with sentinal Nil[T]



(b)

→ R-B-Tree with internal nodes only.



(c)

⊕ black-height of the node, bh(x) :

The number of ~~be~~ black nodes from a node x (but not including x), down to a leaf, is called the black-height of the node, denoted as bh(x)

→ The black-height of the R-B tree is the black-height of its root.

V. Imp.

⊕ Lemma :

A red-black tree with n internal nodes has height at most $2 \lg(n+1)$.

Proof :

Its subtree rooted at any node x contains atleast $2^{bh(x)} - 1$ internal nodes.

→ we prove this by induction on the height of x.

↳ if the height of x is zero (0), then x must be a leaf (NIL[T]), and the subtree rooted at x contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 1 - 1 = 0$ internal nodes.

→ For inductive step, consider a node x that has positive height and is an internal node with two children.

↳ each child has black height ~~$2^{bh(x)}$ or $2^{bh(x)}$~~ bh(x) or bh(x) - 1, depending depending on whether its color is red or black, respectively

→ Since the height of a child of x is less than the height of x itself, so we can apply this hypothesis to conclude that each child has atleast $2^{bh(x)-1} - 1$ internal nodes

→ Thus subtree rooted at x. contains atleast —

$$2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1 = \boxed{2^{bh(x)} - 1} \text{ internal nodes.}$$

→ Let h be the height of the tree. According to property 4, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black.

→ Consequently, the black-height of the root must be at least $h/2$; thus.

$$n \geq 2^{h/2} - 1.$$

$$\Rightarrow n+1 \geq 2^{h/2}, \text{ now taking the log}$$

$$\log(n+1) \geq h/2 \Rightarrow \boxed{h \leq 2\log(n+1)}$$

→ As a consequence of this lemma, the dynamic-set of operations, SEARCH, MINIMUM, MAX, PREDECESSOR, SUCCESSOR can be implemented in $O(\log n)$ time.

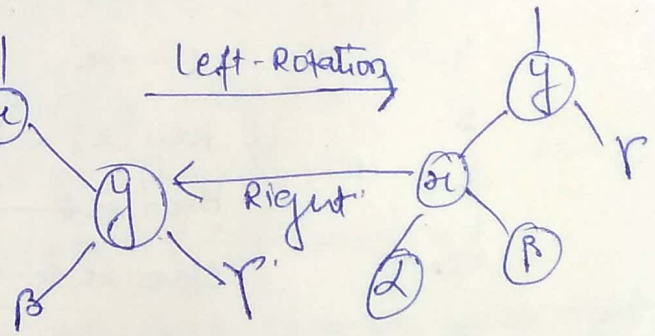
(#) **Rotations:** When we run TREE-INSERT & TREE-DELETE operations on a red-black Tree with n keys, take O(log n) time. These operations modify the tree, the result may violate the red-black properties.

→ To restore these properties, we must change the colors of some of the nodes and also change the structure

→ We change the pointer structure through rotation, which is a local operation in search tree, that preserve the binary-search-tree property.

→ There are two kinds of rotations:

① Left Rotation ② Right Rotation



① **Left-Rotation:**

The Pseudocode for LEFT-ROTATE assumes that right[x] ≠ NIL[T].

```
LEFT-ROTATE(T, x)
1.  y ← RIGHT[x]
2.  right[x] ← left[y]
3.  P[left[y]] ← x
4.  P[y] ← P[x]
5.  if P[x] = NIL[T]
6.      then root[T] ← y
7.      else if x = left[P[x]]
8.          then left[P[x]] ← y
9.      else right[P[x]] ← y
10. left[y] ← x
11. P[x] ← y
```

R-B Tree insertion :

Insertion in R-B tree can be accomplished in O(logn) time. TREE-INSERT procedure to insert a node z into the tree T as if it were an ordinary binary-search tree, and then we color z red. To guarantee that red-black properties are preserved, we call an auxiliary procedure RB-INSERT-FIXUP to recolor nodes and perform rotations.

```
RB-INSERT(T, z)
1.  y ← Nil[T]
2.  x ← root[T]
3.  while x ≠ nil[T]
4.    do y ← x
5.       if key[z] < key[x]
6.         then x ← left[x]
7.         else x ← right[x]
8.  p[z] ← y.
9.  if y = Nil[T]
10.   root[T] ← z
11.   else if key[z] < key[y]
12.          then left[y] ← z
13.          else Right[y] ← z
14. left[z] ← Nil[T]
15. Right[z] ← Nil[T]
16. Color[z] ← red.
17. RB-INSERT-FIXUP(T, z)
```

RB-INSERT-FIXUP (T, z)

```
1.  while color[P[z]] = RED
2.      do if P[z] = left[P[P[z]]]
3.          then y ← right[P[P[z]]]
4.              if color[y] = RED
5.                  then color[P[z]] ← BLACK
6.                       color[y] ← BLACK
7.                       color[P[P[z]]] ← RED
8.                       z ← P[P[z]].
9.              else if z = right[P[z]]
10.                 then z ← P[z]
11.                      LEFT-ROTATE(T, z)
12.             color[P[z]] ← BLACK.
13.             color[P[P[z]]] ← RED
14.             RIGHT-ROTATE(T, P[P[z]])
15.     else (same as then clause with right &
                left exchanged)
16. color[root[T]] ← BLACK.
```

case 1.

case 2

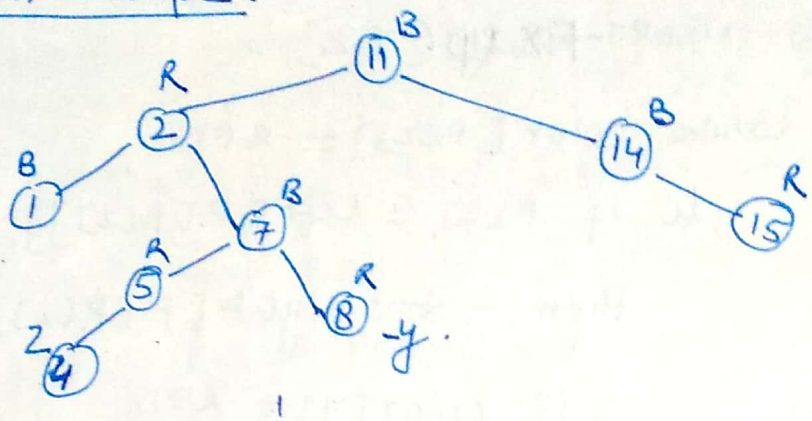
case 3.

→ **case 1**: z's uncle y is red.

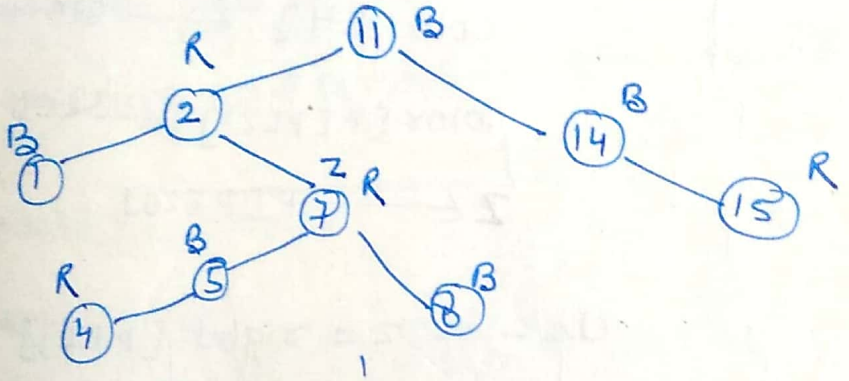
→ **case 2**: z's uncle y is black and z is a right child.

→ **case 3**: z's uncle y is black & z is a left child

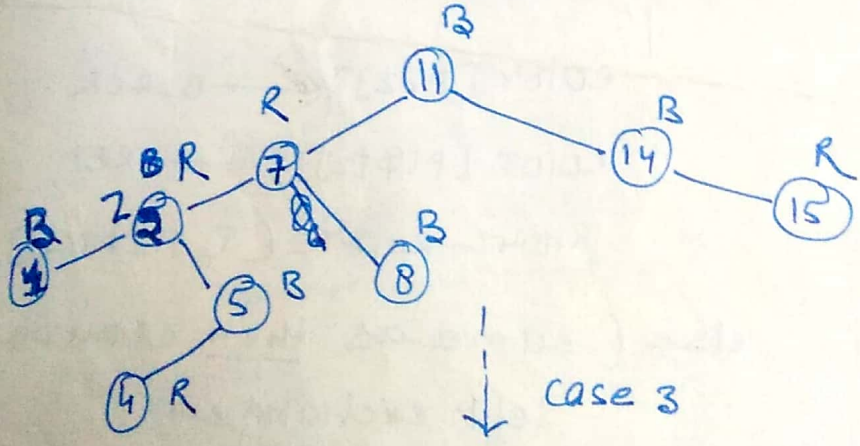
⊛ RB-Tree Insertion-example:



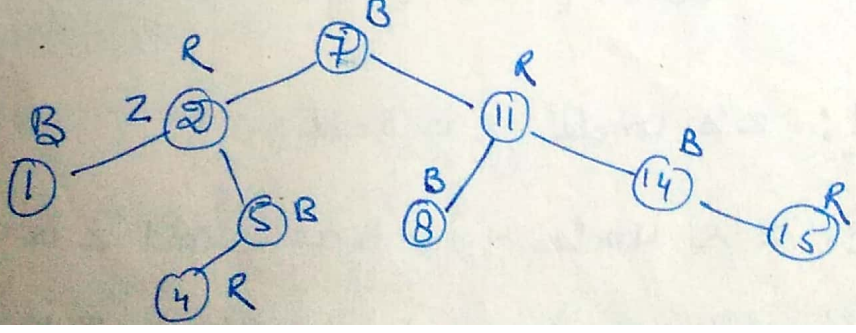
case 1.



case 2



case 3



RB-Tree Deletion:

→ Like other basic operations on n-node red-black Tree, deletion of a node takes O(lg n) time.

→ The procedure, RB-DELETE after splicing out a node, it calls an auxiliary procedure RB-DELETE-FIXUP to restore the red-black properties.

```
RB-DELETE (T, z)
1. if left[z] = Nil[T] or right[z] = Nil[T]
2.    then y ← z
3.    else y ← Tree-Successor(z)
4. if left[y] ≠ Nil[T]
5.    then x ← left[y]
6.    else x ← Right[y]
7  P[x] ← P[y]
8. if P[y] = Nil[T]
9.    then root[T] ← x
10.   else if y = left[P[y]]
11.        then left[P[y]] ← x
12.        else right[P[y]] ← x
13. if y ≠ z
14.    then key[z] ← key[y]
15. if color[y] = BLACK
16.    then RB-DELETE-FIXUP(T, x)
17. return y.
```

P.T.O

RB-DELETE-FIXUP(T, x)

```
1. while x ≠ root[T] and color[x] = BLACK
2.     do if x = left[p[x]]
3.         then w ← right[p[x]]
4.             if color[w] = RED
5.                 then color[w] ← BLACK
6.                     color[p[x]] ← RED
7.                     LEFT-ROTATE(T, p[x])
8.                     w ← right[p[x]]
9.             if color[left[w]] = BLACK & color[right[w]] = BLACK
10                then color[w] ← RED
11                    x ← p[x]
12.            else if color[right[w]] = BLACK
13                then color[left[w]] ← BLACK
14.                   color[w] ← RED
15.                   RIGHT-ROTATE(T, w)
16.                   w ← right[p[x]].
17.            color[w] ← color[p[x]]
18.            color[p[x]] ← BLACK
19.            color[right[w]] ← BLACK
20.            left-Rotate(T, p[x])
21.            x ← root[T]
22.    else (same as 'then' clause with right & left exchanged)
23 color[x] ← BLACK.
```

case 1.

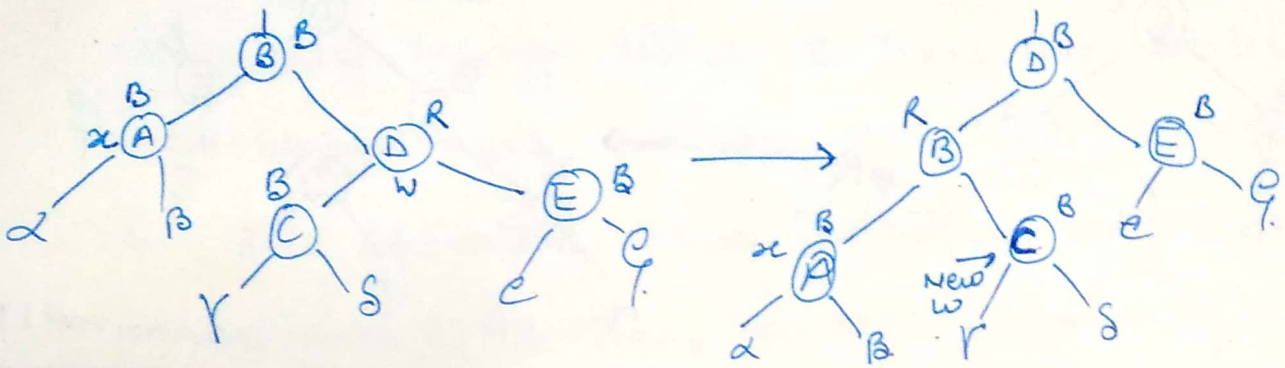
case 2

case 3.

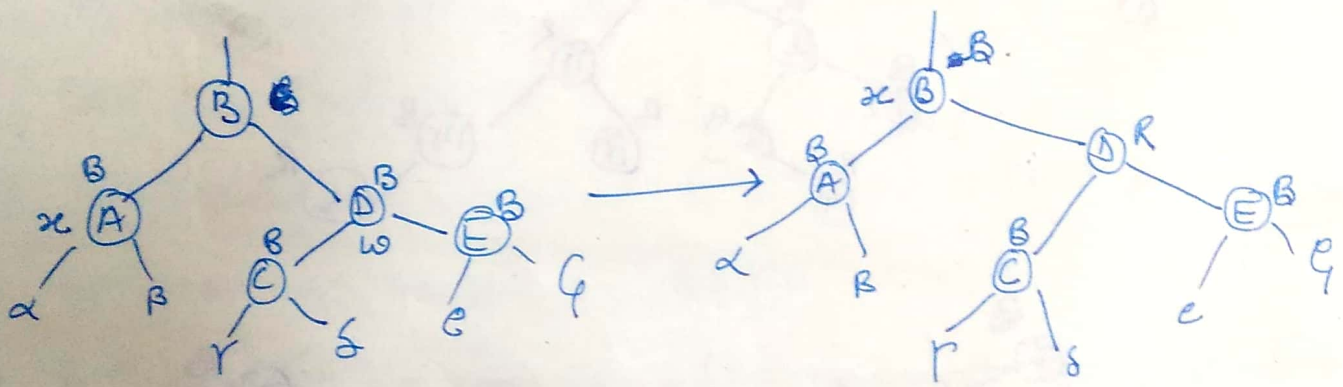
case 4.

Example (RB-Deletion)

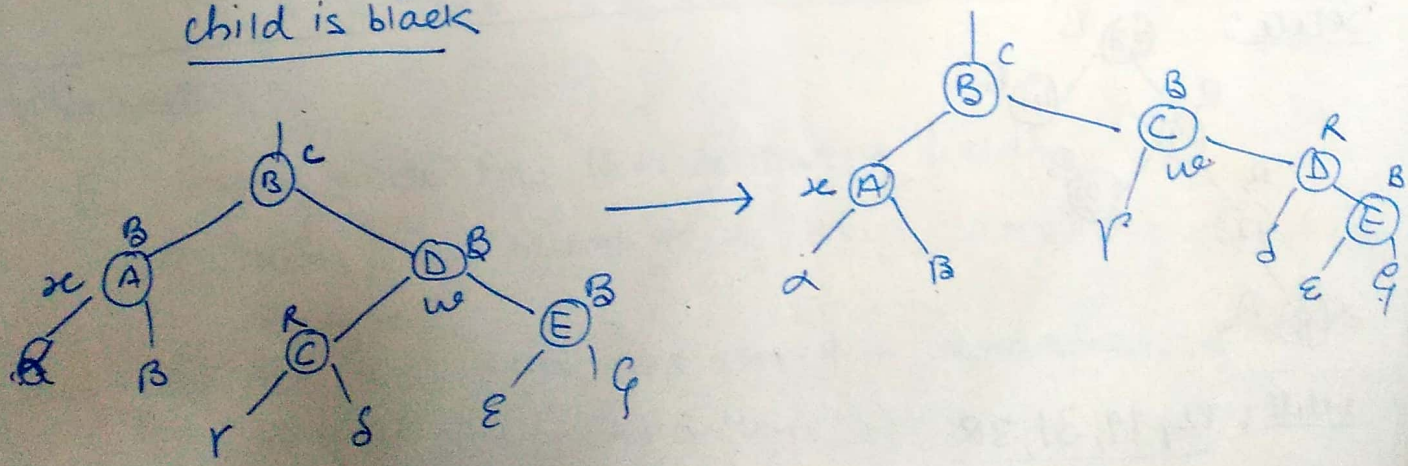
Case 1: x's sibling w is red –



Case: 2: – x's sibling w is black, and both of w's children are black

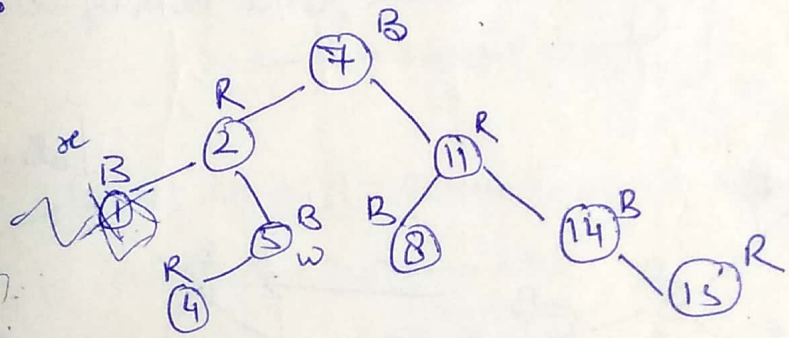


Case: 3 - x's sibling w is black, w's left child is red & right child is black

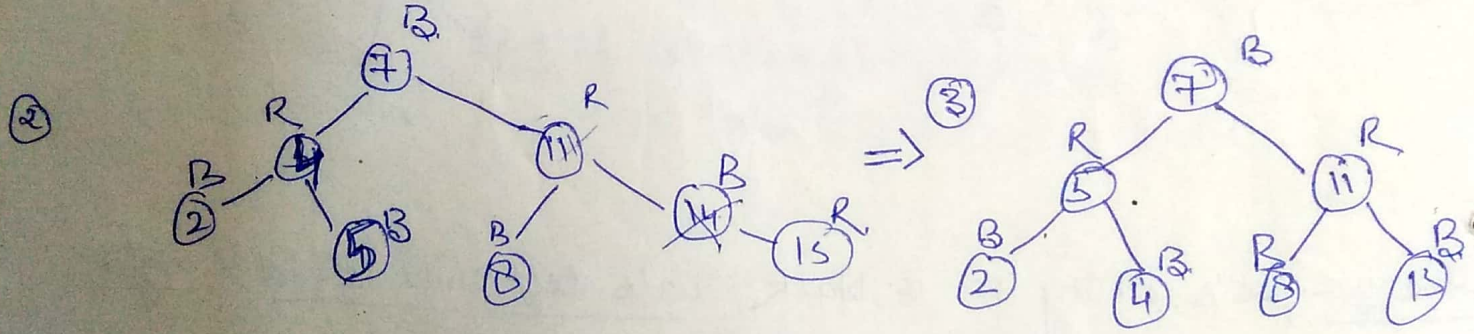


→ R-BT Deletion :

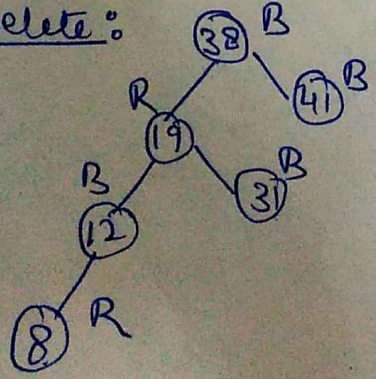
①



②



Delete :



Delete : 12, 19, 31, 38

⊛ **B-Trees:** B-Trees are balanced search trees. to work well on magnetic disks or other direct access secondary storage device. B-Trees are similar to red-black trees, but they are better in minimizing disk I/O operations. Many database systems use B-Tree or variants of B-Trees to store information

→ B-Trees differ from the R-B Trees in that B-Trees node may have many children from a handful to thousands. That is, the branching factor of B-Tree can be quite large.

→ B-Trees are similar to R-B Trees in that every n-node B-Tree has height $O(\lg n)$

→ B-Trees can also be used to implement many dynamic set of operations in time $O(\lg n)$

⊛ **Definition :-**

A B-Tree is a rooted tree having the following properties:

1. Every node has the following fields:
   - (a) $n[x]$, the number of keys currently stored in node x
   - (b) the $n[x]$ keys are stored in nondecreasing order, so that $key_1[x] \leq key_2[x] \leq \cdots \leq key_{n[x]}[x]$
   - (c) $leaf[x]$, a boolean value that is True if x is a leaf and false if x is an internal node

(2) Each internal node x also contains $n[x]+1$ pointers $c_1[x], c_2[x]$ ... $c_{n+1}[x]$ to its children. Leaf node have no children, so their c_i fields are ~~###~~ undefined.

(3) The keys $k_i[x]$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $c_i[x]$, then

$$k_1 \le key_1[x] \le k_2 \le key_2[x] \ldots \le key_n[x] \le k_{n[x]+1}$$

(4) All leaves have the same depth, which is the tree's height h.

(5) There are lower & upper bounds on the number of keys a node can contain. This bound can be expressed interm of fixed integer $t \ge 2$ called the minimum degree of the ~~tree~~. B-Tree.

→ (a) Every node other than root must have atleast $(t-1)$ keys. Every internal nodes other than root has atleast t children. If the tree is nonempty, the root must have one key

→ (b) Every node can contain at most $(2t-1)$ keys. Thus an internal node can have at most $2t$ children.

↳ A Node is said to be full if it contains exactly $(2t-1)$ keys

→ The simplest B-Tree occurs when $[t=2]$. Every internal node then has either 2, 3, or 4 children & we have 2-3-4 tree

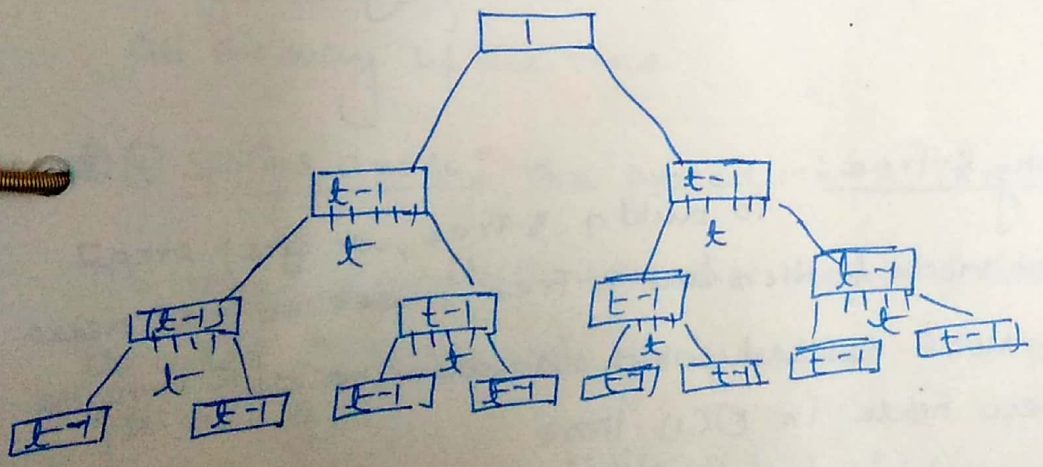
The height of a B-Tree :

The number of disk accesses required for most operations on a B-tree is proportional to the height of the B-Tree.

→ Worst-case height of a B-Tree :

Theorem : If $n \geq 1$, then for any n-key B-Tree T of height h and minimum degree $t \geq 2$, $\left[h \leq \log_t \frac{n+1}{2}\right]$

Proof : If a B-Tree has height h, the root contains at least one key & all other node contain atleast $(t-1)$ keys. Thus, there are at least 2 nodes at depth 1, atleast $2t$ nodes at depth 2, at least $2t^2$ nodes at depth 3, and so on, until at depth h there are at least $2t^{h-1}$ nodes.



Depth	No. of nodes
0	1
1	2
2	$2t$
3	$2t^2$

Thus the number of keys n satisfy.

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

$$n \geq 1 + 2(t-1)\sum_{i=1}^{h} t^{i-1}$$

$$n \geq 1 + 2(t-1)\left[\frac{t^h - 1}{(t-1)}\right] \quad \text{(GP)}$$

$$\rightarrow$$

$$n \geq 2(t^h - 1) + 1$$

$$n \geq 2t^h - 2 + 1$$

$$n \geq 2t^h - 1 \Rightarrow (n+1) \geq 2t^h$$

$$\Rightarrow \frac{(n+1)}{2} \geq t^h.$$

take log ~~$\log_t h = \log_t \frac{(n+1)}{2}$~~

Basic Operations on B-Tree:

(1) Searching:

```
B-TREE-SEARCH(x, k)
1. i ← 1
2 while i ≤ n[x] and k > keyi[x]
3     do i ← i+1
4 if i ≤ n[x] and k = keyi[x]
5     then Return (x, i)
6 elseif leaf[x]
7.    then Return NIL
8.    else Disk-READ(ci[x])
9.    return B-Tree-SEARCH(ci[x], k)
```

(9) → The No. disk accesses in B-Tree search is $O(h) = O(\log_t n)$

↳ Since $n[x] \geq 2t$, the time taken by each node is $O(t)$,

So total time is $\boxed{O(th) = O(t \log_t n)}$

Creating an empty B-Tree:

To build a B-Tree, we first create an empty root node then call B-Tree-INSERT to add new keys. ALLOCATE-NODE is used which allocate one disk page to be used as a new node in $O(1)$ time.

```
B-TREE-CREATE(T)
1.  x ← ALLOCATE-NODE()
2.  leaf[x] ← True
3.  n[x] ← 0
4.  Disk-write(x)
5.  root[T] ← x.
```

LECTURE-18

③ Inserting a key into a B-Tree:

Inserting a key into a B-tree is ~~signifi~~ different than inserting a key into a BST. ~~In~~ In BST, we search for leaf position at which to insert the new key. But, with B-Tree we can not simply create a new leaf node & insert it. Instead, we insert the new key into an existing leaf node.

→ Since we can not insert key into a leaf node that is full, so we use an operation that splits a full node (having $2t-1$ keys) around its median key into two nodes having $(t-1)$ keys each. The median key is moved up to the parent node to identify the dividing point b/w two new trees.

→ If parent node is also full, then it must be split before new key insertion, & thus this need to split full nodes can propagate all the way up the tree.

④ Spliting a node:

The procedure B-TREE-SPLIT-CHILD takes as input a nonfull internal ~~x~~ node x (assumed in main memory), an index i, and a node y such that $y = c_i[x]$ is full is a full child of x.

→ This procedure splits this child (y) into two & adjusts x so that it has an ~~addition~~ additional child.

→ B-Tree-INSERT(T, k)

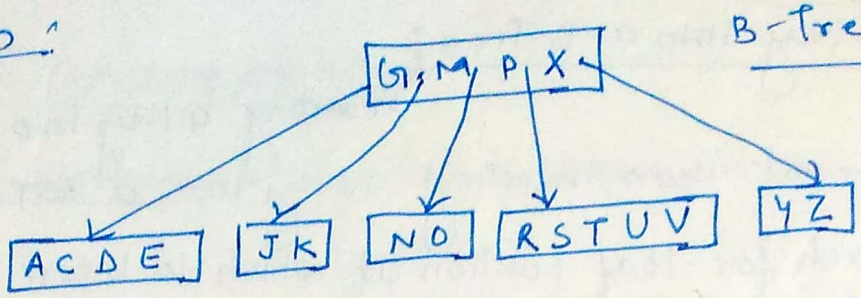
→ B-TREE-INSERT-NONFULL(r, k)

→

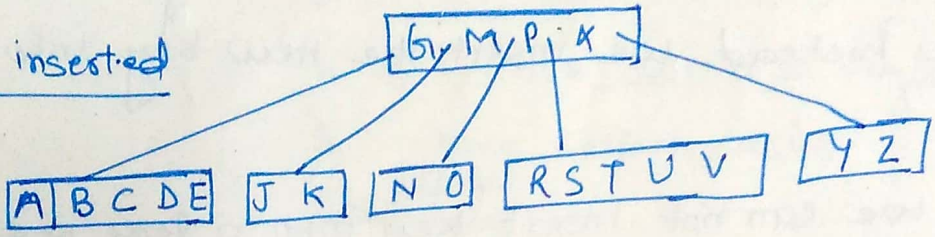
INSERTIONS :

B-Tree with t=8

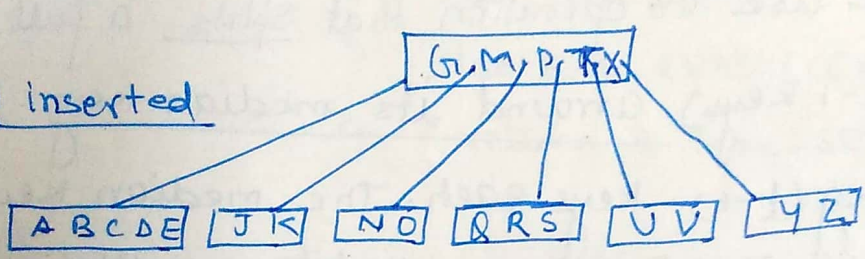
(a)



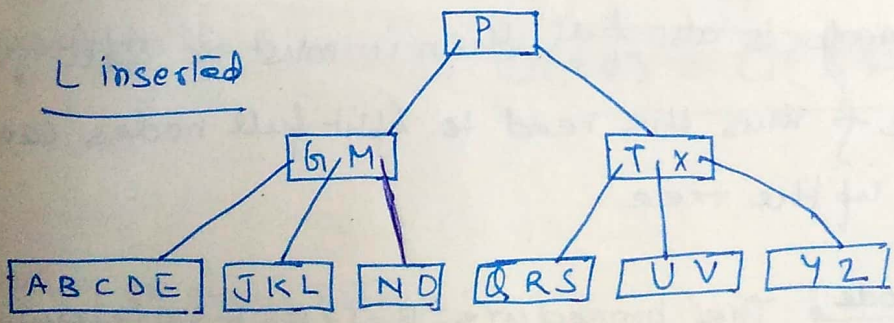
(b) B inserted



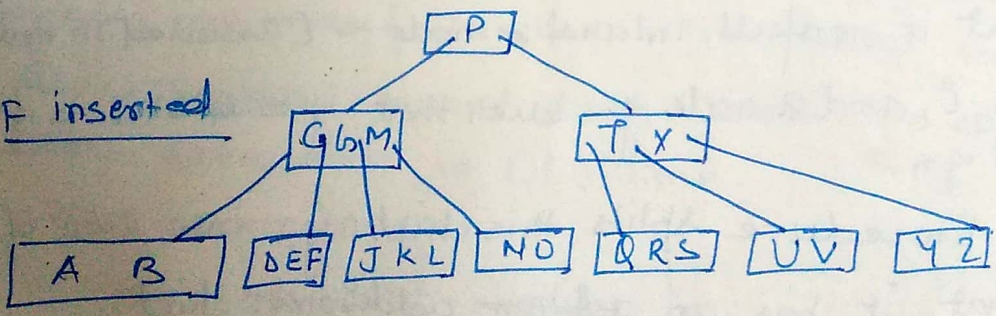
(c) Q inserted



(d) L inserted



(e) F inserted



Deletion :

(1) If the key k is in node x & x is a leaf, delete the the key from x.

(2) If the key k is in node x & x is an internal node, then do the following:-

→ (a) If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k' & replace k by k' in node x

→ (b) Symmetrically, if the child z that follows k in node x has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k' & replace k with k' in x.

→ (c) Otherwise, if both y & z have only (t-1) keys, merge k and all of z into y, so that x loses both k & pointer to z, & y now contains (2t-1) keys. Then, free z & recursively delete k from y.

(3) If the key k is not present in internal node x, determine the root $c_i[x]$ of the appropriate subtree that must contain k_i. If $c_i[x]$ has only (t-1) keys, execute steps 3a or 3b as necessary.

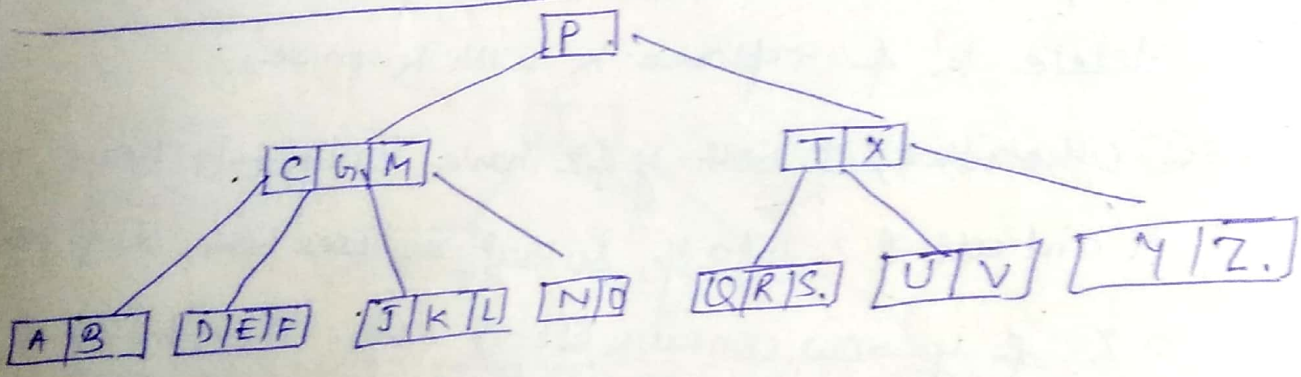
→ (a) If $c_i[x]$ has only (t-1) keys but has an immediate sibling with at least t

~~(A) F, 7, 30 (B)~~

* F, M, G, D, B

P, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X
Y, D, Z, E

→ (3b) If Ci[x] and both of Ci[x] immediate siblings have (t-1) keys, merge Ci[x] with one sibling, which involve moving a key from x down into the new merged node to become the median key for that node.



Case 1 : Delete F

Case: 2a : Delete

Case 2 b : G (deleted

Case 3b : D (deleted

Case 3a : B (deleted)

Q.1 Show the result of inserting keys into empty B-Tree of order 5

(a) S, A, B, H, I, E, K, J, N, V, M, Y, Z, W

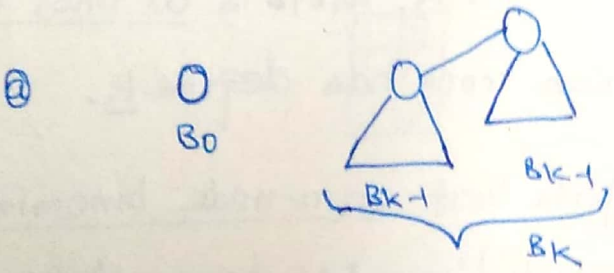
(b) 25, 31, 38, 76, 5, 60, 8, 30, 15, 35, 17, 23, 53, 27, 43, 65, 48

LECTURE-19 Binomial Heap

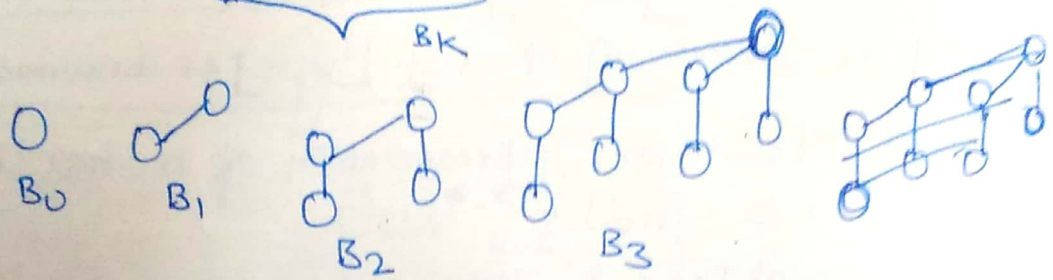
Binomial Trees:

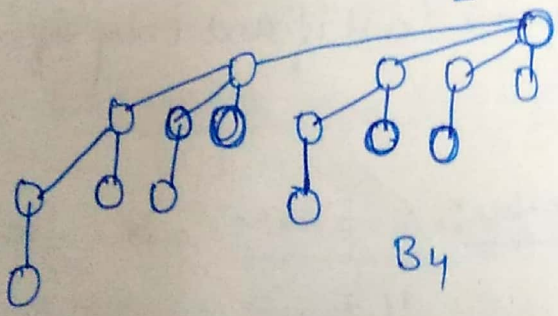
The binomial Tree B_k is an ordered tree defined recursively. The Binomial tree B_0 consist of a single node. The binomial tree B_k consists of two binomial trees B_{k-1} that are linked together: the root of one is the leftmost child of the root of the other.

(a)



(b)





Properties:

1. There are 2^k nodes
2. The height of the tree is k
3. There are exactly $\binom{k}{i}$ i.e. ${}^kC_i$ nodes at each depth i for $i = 0, 1, 2 \cdots k$

$${}^kC_i = \frac{k!}{i! * k-i!}$$

4. The root has degree k, which is greater than that of any other node

* The maximum degree of any node in an n-node tree is

$$\lfloor \log n \rfloor$$

⊕ Binomial Heaps :

A binomial Heap H is a set of binomial trees that satisfies the following binomial-heap properties:

→ ① Each binomial tree in H obeys the min-heap property i.e each tree is min-heap ordered

→ ② For any nonnegative integer k, there is at most one binomial tree in H whose root has degree k.

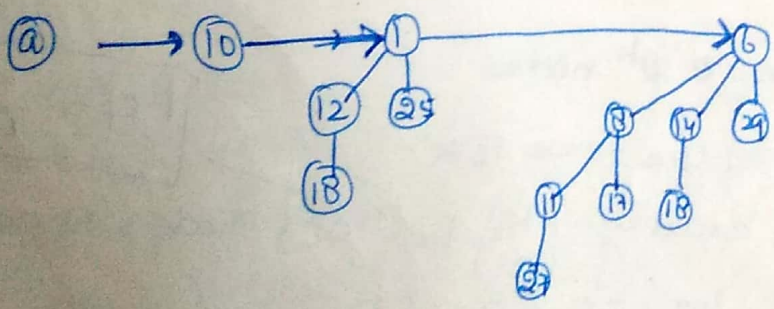
↳ The property ② implies that an n-node binomial heap H consists of at most $\lfloor \lg n \rfloor + 1$ binomial trees.

Since the binary representation of n has $\lfloor \lg n \rfloor + 1$ bits i.e $\left[n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i \right]$

↳ Binomial tree B_i appears in H if and only if bit $b_i = 1$

⊕ Representation of Binomial Heaps :

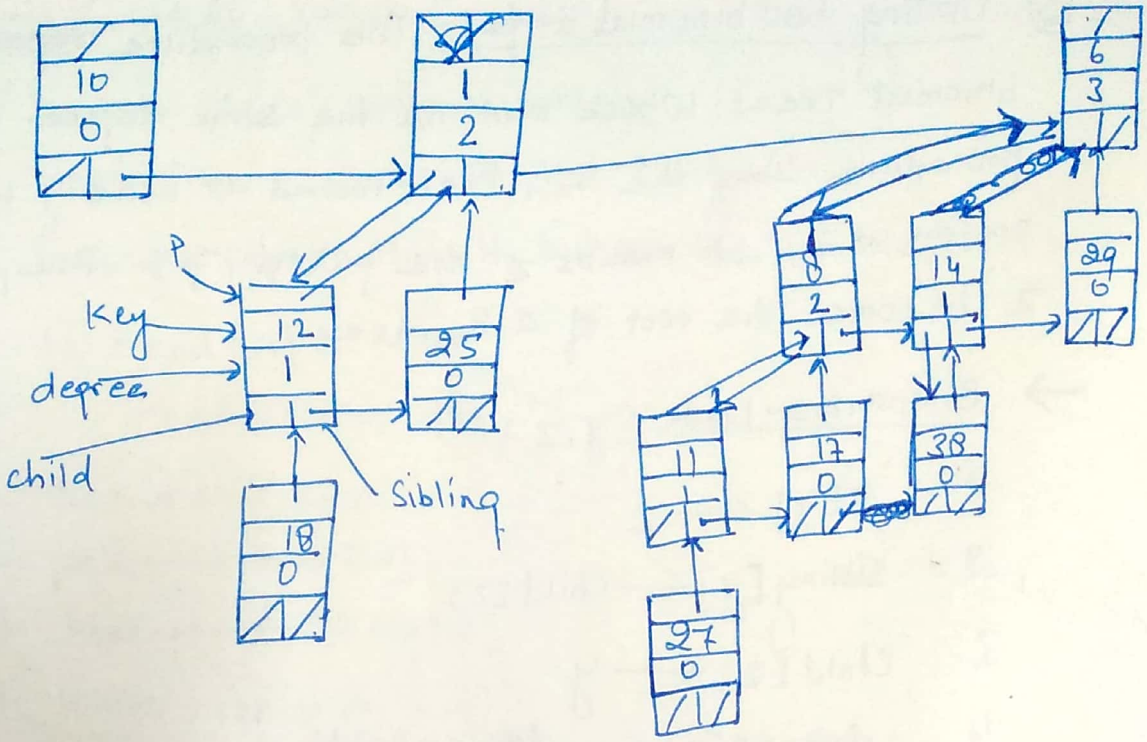
$n = 13$



roots are sorted according to degree & stored as link list.

→ It consist of B_0, B_2, & B_3 binomial trees which have 1, 4, 8 Nodes respectively

P.T.O



⇒ Operations on binomial heaps:

(1) Creating a new binomial heap:

To make an empty binomial heap, the MAKE-BINOMIAL-HEAP procedure simply allocates & returns an object H, where head[H] = NIL.

(2) Finding the minimum key:

This procedure returns a pointer to the node with the minimum key in an n-node binomial heap H. This assumes that there are no keys with ∞.

BINOMIAL-HEAP-MINIMUM(H)

1. y ← NIL
2. x ← Head[H]
3. min ← ∞
4. while x ≠ NIL
5. do if key[x] < min
6. then min ← key[x]
7. y ← x
8. x ← sibling[x]
9. Return y.

P.T.O

(Binomial link)

⇒ ③ Uniting two binomial Heaps: This procedure ~~repeatedly~~ links binomial trees whose root has the same degree. The following procedure links the B_{k-1} tree rooted at node y to the B_{k-1} rooted at z; it makes z the parent of y. Thus, the node z becomes the root of a B_k tree.

→ BINOMIAL-LINK (y, z)

1. P[y] ← z
2. Sibling[y] ← child[z]
3. child[z] ← y
4. degree[z] ← degree[z] + 1

→ Binomial-Union: The binomial-union ~~operati~~ procedure repeatedly links binomial trees whose root has the same degree.

↳ The following union procedure unites binomial heaps H_1 & H_2, & and returns the resulting Heap H

↳ Besides the Binomial-link, procedure uses an auxiliary procedure Binomial-Heap-merge — that merges the root lists of H_1 & H_2 into a single linked list that is sorted by degree into monotonically increasing order.

P.T.O

LECTURE-20

→ BINOMIAL-HEAP-UNION(H_1, H_2)

1. H ⟵ MAKE-BINOMIAL-HEAP()
2. Head[H] ⟵ Binomial-Heap-Merge(H_1, H_2)
3. Free the objects H_1 & H_2 but not the list they point to
4. if head[H] = NIL
5. then return H.
6. Prev-x ⟵ NIL
7. x ⟵ head[H]
8. next-x ⟵ Sibling[x]
9. while next-x ≠ NIL
10. if (degree[x] ≠ degree[Next-x]) OR (Sibling[next-x] ≠ NIL & degree[Sibling[Next-x]] = degree[x]) — case 1 & case 2
11. then prev-x ⟵ x
12. x ⟵ next-x
13. else if key[x] ≤ key[next-x] — case 3
14. Sibling[x] ⟵ Sibling[next-x]
15. Binomial-Link(next-x, x)
16. else if prev-x = NIL — case 4.
17. then head[H] ⟵ next-x
18. else Sibling[prev-x] ⟵ next-x
19. Binomial-Link(x, next-x)
20. x ⟵ next-x
21. next-x ⟵ Sibling[x]
22. Return H.

Case 1 : Case 1 occurs when $degree[x] \neq degree[next\text{-}x]$

→ so we do not link x & next-x, we simply move the pointers one position.

Case 2 : Case 2 occurs when x is the first of three roots of equal degree, that is, when

$$degree[x] = degree[next\text{-}x] = degree[sibling[next\text{-}x]].$$

Thus, in this case we also simply move the pointers one position like case 1.

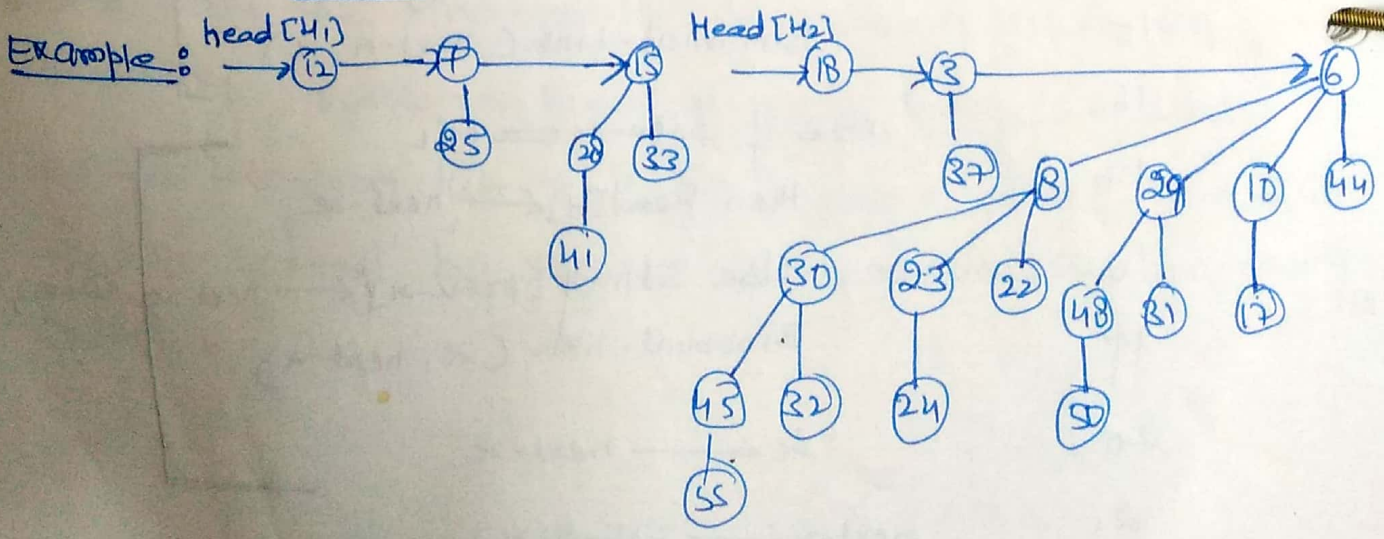
Case 3 & 4 :- Case 3 & 4 occurs when x is the first of two roots of equal degree i.e.

$$degree[x] = degree[next\text{-}x] \neq degree[sibling[next\text{-}x]]$$

↳ In case 3, $key[x] \leq key[next\text{-}x]$, so next-$x$ is linked to x.

↳ In case 4 $key[next\text{-}x] \leq key[x]$, so x is linked to the next-x.

Example :



→ **Running time of BINOMIAL-HEAP-UNION:** The running time of binomial heap union is $O(\lg n)$, where n is the total Number of nodes in binomial heaps H_1 & H_2. Let H_1 contains n_1 nodes & H_2 contain n_2 nodes, so that $n = n_1 + n_2$. Thus H_1 contains at most $\lfloor \lg n_1 \rfloor + 1$ roots of H_2 contains $\lfloor \lg n_2 \rfloor + 1$ roots so H contains at most $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2 \leq 2\lfloor \lg n \rfloor + 2 = O(\lg n)$ immediately after the call of Binomial-Heap-Merge.

→ The time to perform binomial Heap merge is $O(\lg n)$.

→ Each iteration of while loop takes $O(1)$ time, and there are at most $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$ iterations b'coz each iteration either advances the pointer one position down in the root-list or removes a root from the root list. Thus, the total time is $O(\lg n)$.

(ii) **Inserting a node:** The following procedure inserts a node x into Binomial Heap H, assuming that x has already been allocated and key[x] has already been filled in.

Binomial-Heap-insert(H, x)

1. H' ← MAKE-BINOMIAL-HEAP()
2. p[x] ← NIL
3. child[x] ← NIL
4. sibling[x] ← NIL
5. degree[x] ← 0
6. head[H'] ← x
7. H ← Binomial-HEAP-UNION(H, H')

⇒ This procedure simply make one node in $O(1)$ time Binomial Heap H' in $O(1)$ time & unites it with n-node binomial Heap H with $O(\lg n)$

P.T.O

(#) Extracting node with minimum key: This procedure extracts ~~Binomial~~ the node with minimum key from binomial Heap H & returns a pointer to the extracted node.

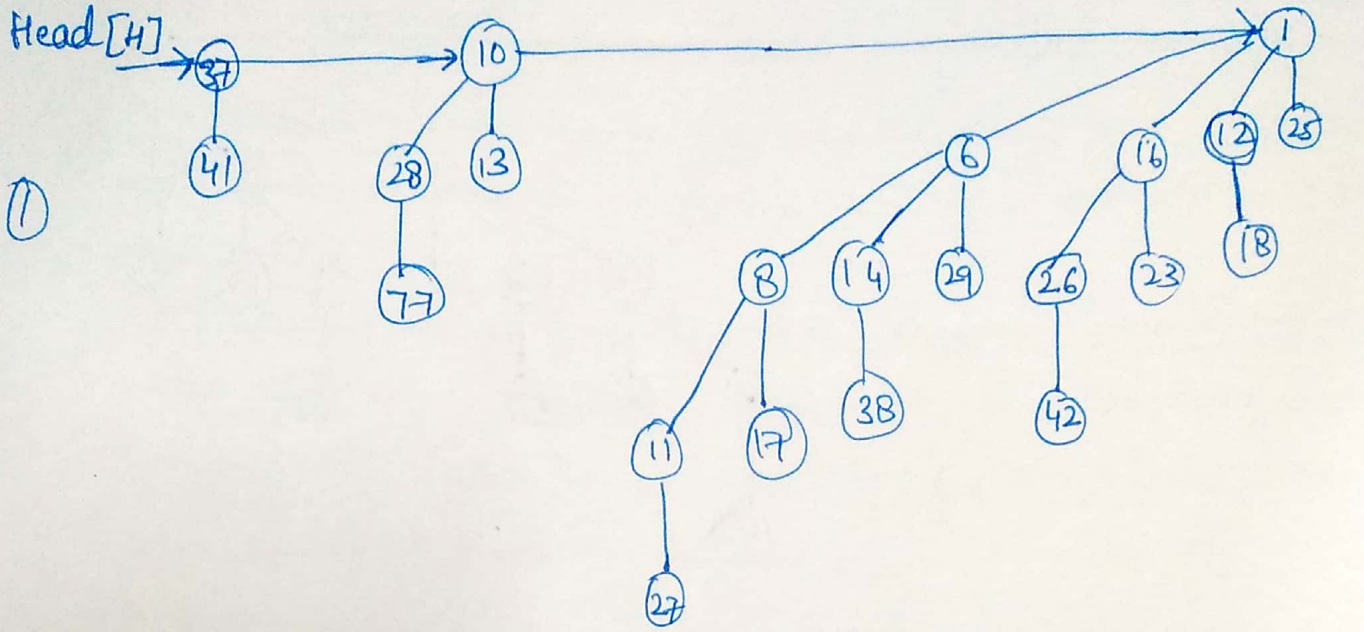
Binomial-HEAP-Extract-min(H)

1. Find the root x with min. key in the root list of H & remove x from the roots list of H.
2. H' ← MAKE-BINOMIAL-HEAP()
3. Reverse the order of the linked list of x's children & set head[H'] to point the head of the resulting list.
4. H ← BINOMIAL-HEAP-UNION(H, H')
5. Return x

(#) Decreasing a key: (Same as in case of Heap.)

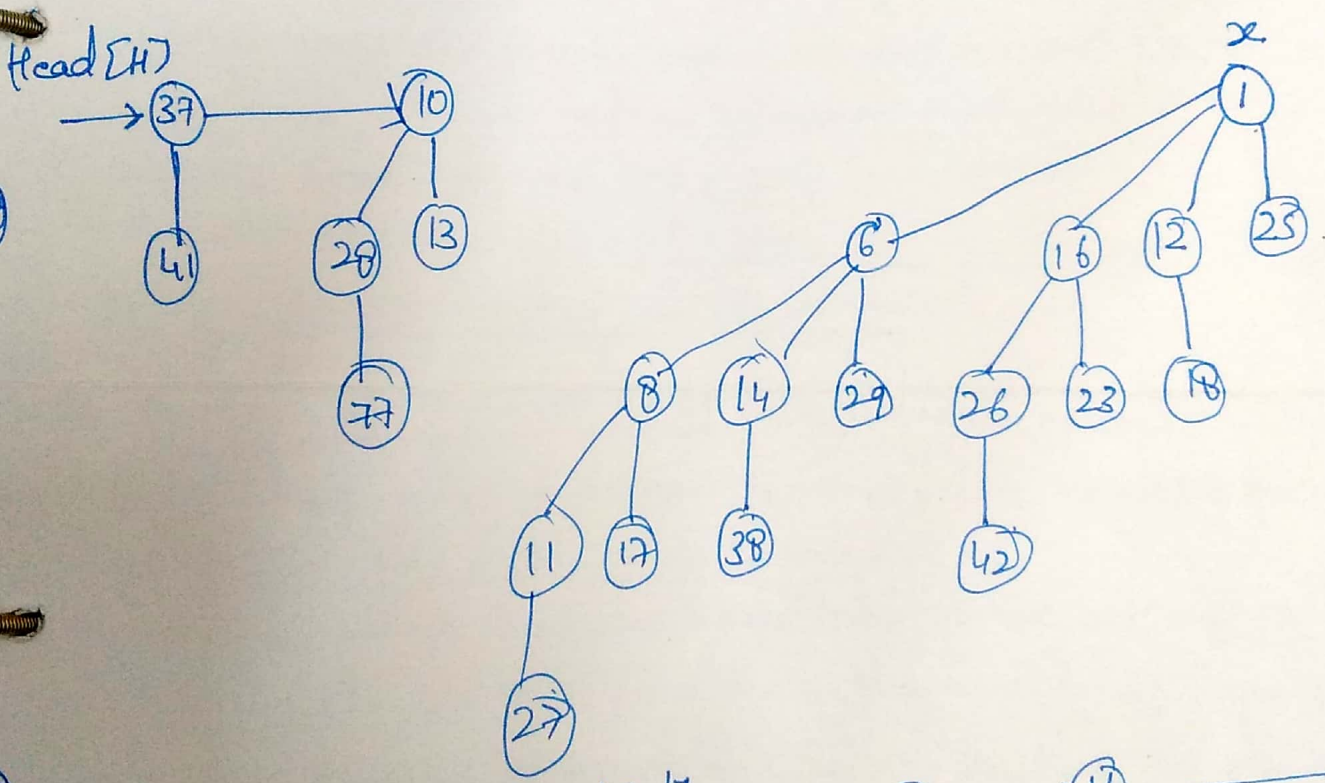
(#) ~~Error~~ Deleting a key:

Extract-Min - Binomial Heap :

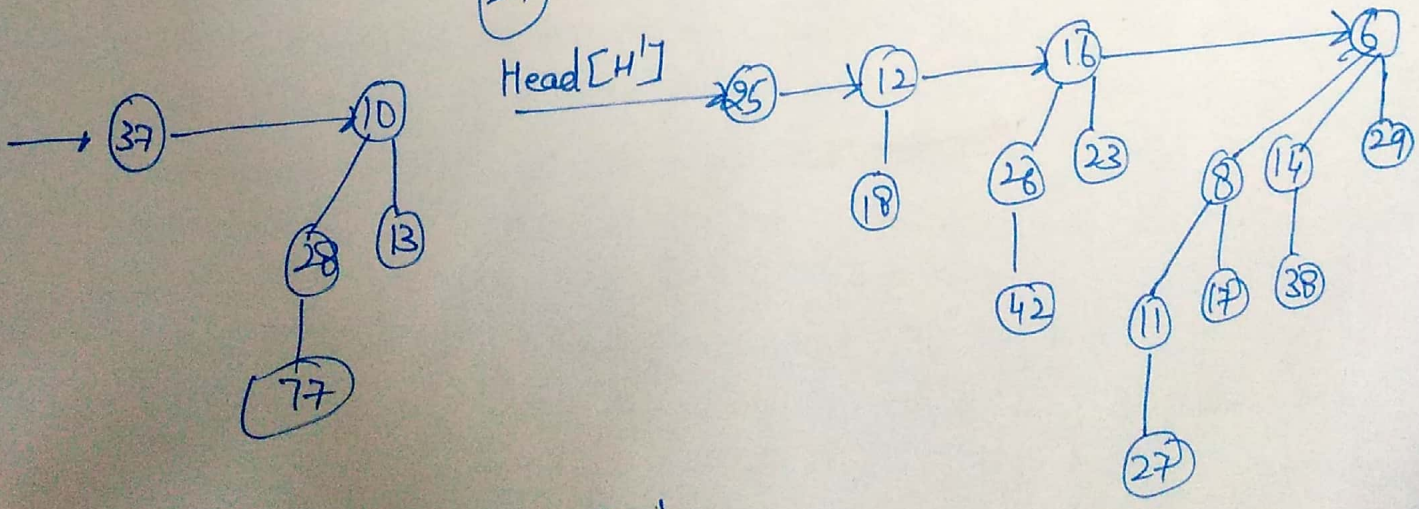


(1)

(2)

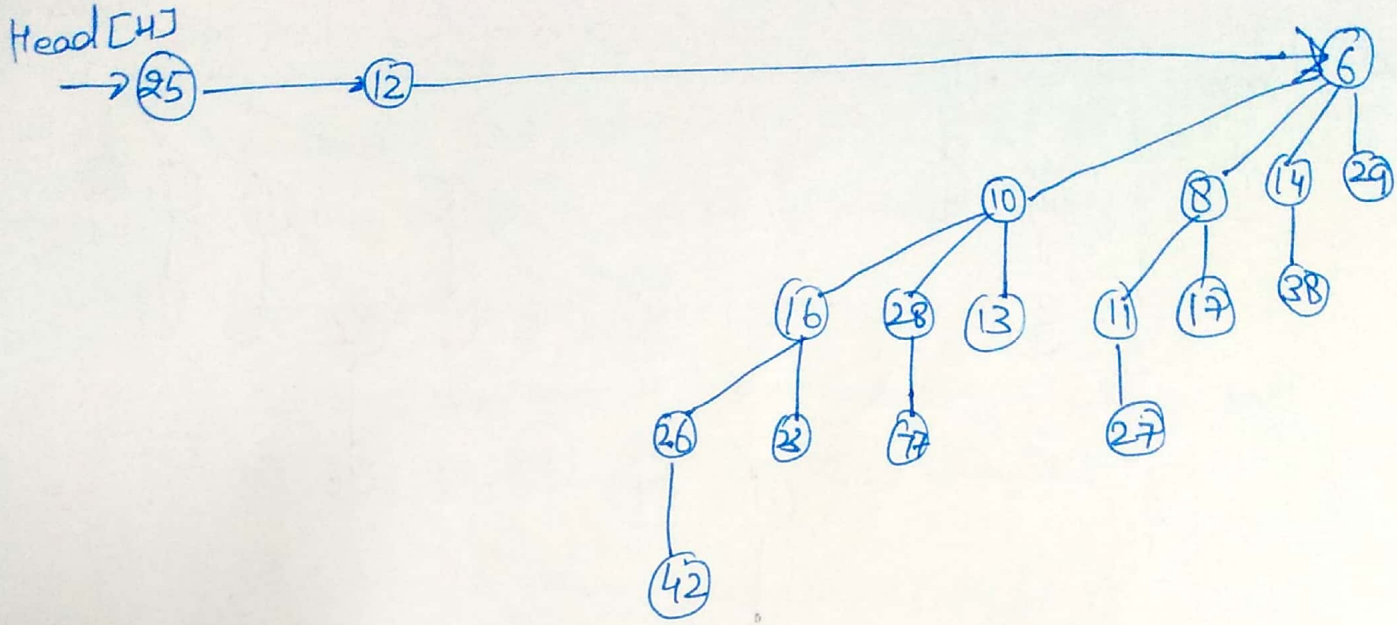


(3)



Now union H & H' :

P.T.O



LECTURE-21

Fibonacci Heaps:

Fibonacci heaps support the same set of operations as binomial heaps, but have advantage that operations ~~the that~~ operations that do not involve deleting an element run in O(1) amortized time

→ Fibonacci heaps are desirable when the Extract-min & Delete operations is relatively small. This situation may arises in many applications. for example algos for graph problem may call Decrease-key once per edge.

→ like binomial heap, a fibonacci heap is a collection of trees. Fibonacci heaps are loosely based on ~~fibon~~ binomial heaps. Fibonacci heaps have a more relaxed structure than binomial heaps, allowing more improved asymptotic time bounds.

→ like binomial heaps, Fibonacci heaps are not designed to give efficient support to the search operations.

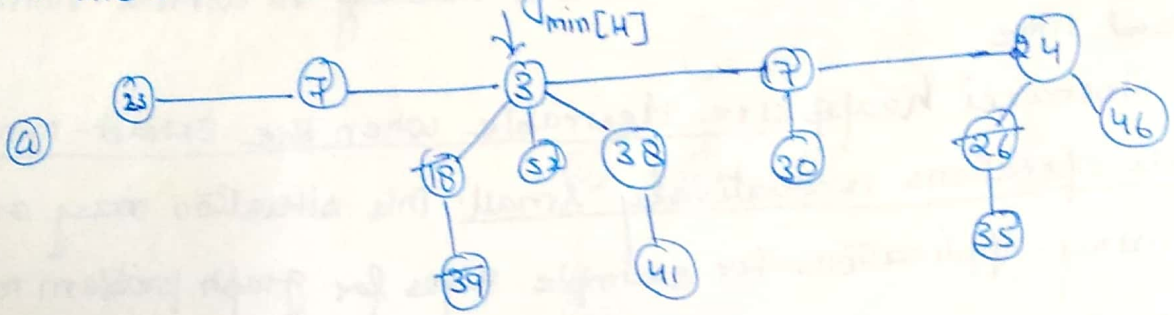
Structure of Fibonacci heaps:

A fibonacci heap is a collection of min-heap-ordered trees. The trees in a fibonacci heap are not constrained to be binomial trees.

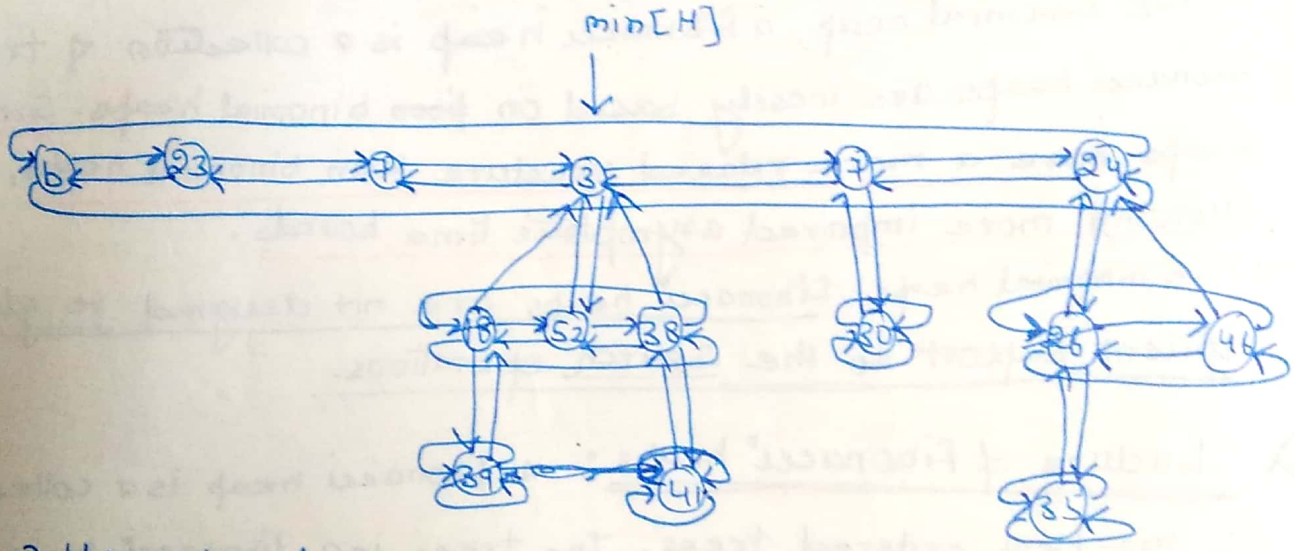
→ Trees within fibonacci heaps are rooted but not ordered i.e unordered. Each node x in fibonacci heap contains a pointer p[x] to its parent & a pointer child[x] to any of its child. Childrens of x are linked together in a circular, doubly linked list, which is called child list of x.

→ Each child y in the child list has pointers left[y] & Right[y] that point to y's left & right siblings, respectively. If y is the only child, then left[y] = Right[y] = y. the order in which child appears in child list is arbitrary.

→ Circular, doubly linked lists have two advantages for use in Fibonacci heaps:- ① we can remove a node from doubly linked list in O(1) time, ② given two such lists, we can concatenate them into one circular, doubly linked list in O(1) time.



(a)



(b)

→ Here two fields are important- the No. of children in the child list of x is stored in degree[x]. The boolean-valued field mark[x] indicates whether the node x has lost a child since the last time x was made the child ~~a child~~ of another node. Newly created nodes are unmarked, and a node x has

→ The roots of all the trees in fibonacci heap are linked together using their left & right pointers into a circular, doubly linked list called the root list of the fibonacci heap. The order of trees within root list is arbitrary.

→ The number of nodes in fibonacci heap H currently is in n[H].

(#) Maximum degree: There is an upper bound ~~on~~ D(n) on the max. degree of any node in an n-node fibonacci heap. For mergable heap operations [D(n) ≤ ⌊lg n⌋]. When we support Decrease key & Delete [D(n) = O(lg n)].

(#) Mergable-heap operations:

→ For the unordered binomial tree U_k, the root has degree k which is greater than that of any other node. The children of the root are the roots of subtrees $U_0, U_1, U_2 \cdots U_{k-1}$ in some order.

→ If an n-node fibonacci heap is a collection of unordered binomial trees, then [D(n) = lg n].

(1) Creating a new fibonacci Heap: To make an empty fibonacci heap, the procedure allocates & return the fibonacci heap object H, where n[H] = 0 & min[H] = NIL; there are no trees in H.

→ The following procedure insert a node x into fibonacci Heap H, assuming that x has already been allocated a key.

PT.O

FIB-HEAP-INSERT (H, x)

1. degree[x] ← 0
2. P[x] ← NIL
3. child[x] ← NIL
4. left[x] ← x
5. Right[H] ← x
6. mark[x] ← false
7. concatenate the root list containing x with root list H.
8. if min[H] = NIL or key[x] < key[min[H]]
9. then min[H] ← x
10. n[H] ← n[H] + 1

⑨ Finding the min. node:

③ Uniting the two Fibonacci Heaps:

→ It simply concatenates the root list of H_1 & H_2 and then determine the new min node.

FIB-HEAP-UNION (H_1, H_2)

1. H ← MAKE-FIB-HEAP()
2. min[H] ← min[H_1]
3. Concatenate the root list of H_2 with the root list of H
4. if ((min[H_1] = NIL) or (min[H_2] ≠ NIL & min[H_2] < min[H_1]))
5. then min[H] ← min[H_2]
6. n[H] ← n[H_1] + n[H_2]
7. Free the objects H_1 & H_2
8. Return H.

LECTURE-22

Extracting the minimum node :

FIB-HEAP-EXTRACT-MIN(H)

1. z ← min[H]
2. if z ≠ NIL
3. then for each child x of z
4. do add x to the root list of H
5. p[x] ← NIL
6. remove z from the root list of H
7. if z = right[z]
8. then min[H] ← NIL
9. else min[H] ← right[z]
10. CONSOLIDATE (H)
11. n[H] ← n[H] - 1
12. return z.

→ Consolidating :— This process reduce the number of trees in the fibonacci heap. This involves consolidating the root list consisting of repeatedly executing following steps. Untill every root in the root list has a distinct degree.

(1) Find the two roots x & y in the root list with same degree, where key[x] ≤ key[y]

(2) Link y to x :

P.T.O

```
CONSOLIDATE(H)
1.  For i ← 0 to D(n[H])
2.    do A[i] ← NIL
3.  For each node w in the root list of H
4.      do x ← w
5.          d ← degree[x]
6.          While A[d] ≠ NIL
7.          do y ← A[d]        △ Another node of same degree
8.            if key[x] > key[y]
9.                then exchange x ↔ y
10.               FIB-HEAP-LINK(H, y, x)
11.           A[d] ← NIL
12            d ← d+1
13.         A[d] ← x
14. min[H] ← NIL
15. For i ← 0 to D(n[H])
16      do if A[i] ≠ NIL
17          then add A[i] to the root list of H
18              if min[H] = NIL or key[A[i]] < key[min[H]]
19                  then min[H] ← A[i]
```

```
FIB-LINK(H, y, x)
1. remove y from the root list of H
2. make y a child of x, increment degree of x
3. mark[y] ← false
```

Decreasing a key & deleting a node: Fibonacci heap decreases the key of a node in O(1) amortized time and delete any node in O(D(n)) amortized time.

① Decreasing a key:

```
FIB-HEAP-DECREASE-KEY(H,x,k)
1. if k > key(x)
2      then error
3. key[x] ← k.
4. y ← P[x]
5 if y ≠ NIL & key[x] < key[y]
6.   then CUT(H,x,y)
7       CASCADING-CUT(H,y)
8. if key[x] < key[min[H]]
9    then min[H] ← x
```

CUT (H,x,y)

① remove x from the child list of y, decrementing the degree [
2. add x to the root list of H.
3. P[x] ← NIL
4. mark[x] ← False.

CASCADING-CUT(H,y)

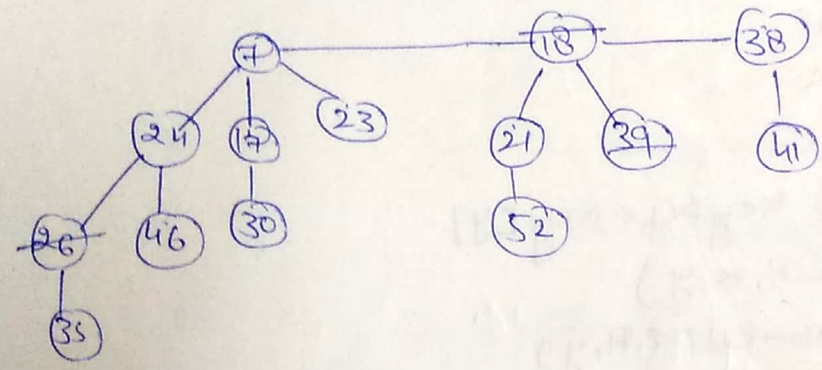
```
1. z ← P[y]
2. if z ≠ NIL
3     then if mark[y] = False
4.            then mark[y] ← TRUE
5             else CUT(H,y,z)
6                  CASCADING-CUT(H,z)
```

Deleting a node :

FIB-HEAP-DELETE(H, x)

1. FIB-HEAP-DECREASE-KEY(H, x, −∞)
2. FIB-HEAP-EXTRACT-MIN(H)

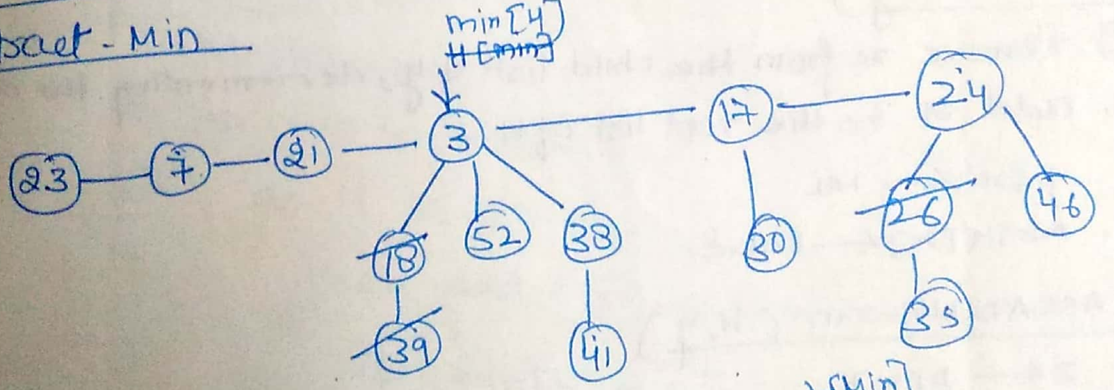
Example



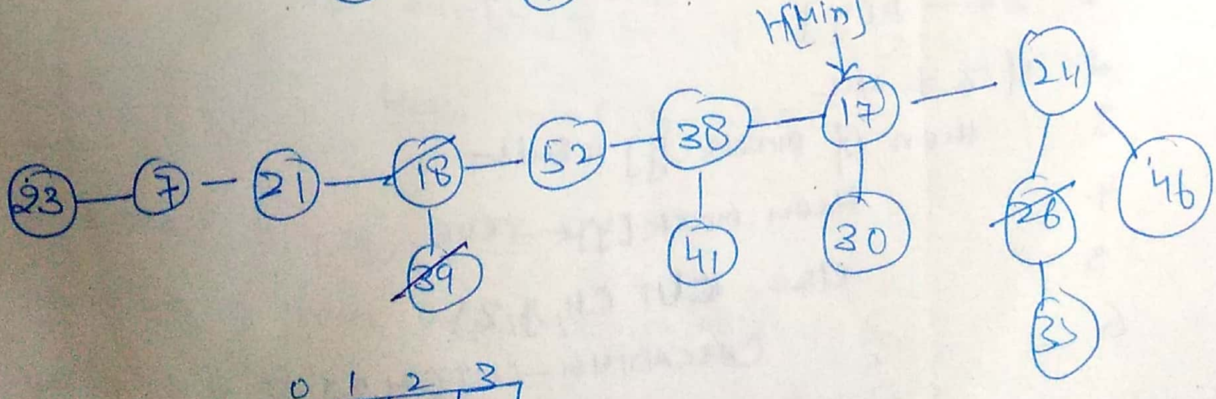
1st Decrease 46 — 15

2nd Decrease 35 — 05

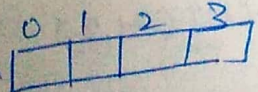
Extract-Min



⇒



Recall: A



↳ Fibonacci Heap: Analysis:

— Amortized Analysis
- ① Aggregate Analysis
- ② Accounting Method
- ③ Potential Method.

→ For the analysis of Fibonacci Heap, Potential Method of amortized analysis is used. Potential method works as follows: We start with an initial data structure D_0 on which n operations are performed. For each $i = 1, 2, \cdots n$ let c_i be the actual cost of the ith operation and D_i be the data structure that results after applying ith operation to data structure D_{i-1}. A potential function ϕ maps each data structure D_i to a real number $\phi(D_i)$, which is the potential associated with data structure D_i. The amortized cost $\hat{c}_i$ of ith operation with respect to potential function ϕ is defined by:

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$$

→ The amortized cost of each operation is its actual cost plus increase in potential due to the operation. The total ~~to~~ amortized cost of n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \phi(D_i) - \phi(D_{i-1}))$$

→ **Potential function for Fibonacci Heap**: For a Given Fibonacci Heap H, we indicate t(H) the Number of trees in the root list of H and m(H) the Number of marked nodes in H. The potential of Fibonacci Heap H is defined by

$$\Phi(H) = t(H) + 2\,m(H)$$

→ The potential of a set of Fibonacci Heaps is the sum of the potentials of its constituent Fibonacci heaps.

→ We assume that a fibonacci heap application begins with no heaps. Therefore, the initial potential is (zero) 0.

LECTURE-23

– Tries Data Structure : [IIT-K By Naveen Garg.]

A trie, also called digital tree, radix tree, or prefix tree, is a kind of search tree - an ordered tree data structure used to store strings. All the descendants of a node have a common prefix of the strings associated with that node, and the root is associated with the empty string. Keys tend to be associated with leaves, though some inner nodes may correspond to keys.

→ Tries data structures are used in pattern matching, to speeds up the pattern matching queries.

→ If the text is large, immutable and searched for often, we want to preprocess the text instead of the pattern in order to pattern matching queries in time proportional to the pattern length

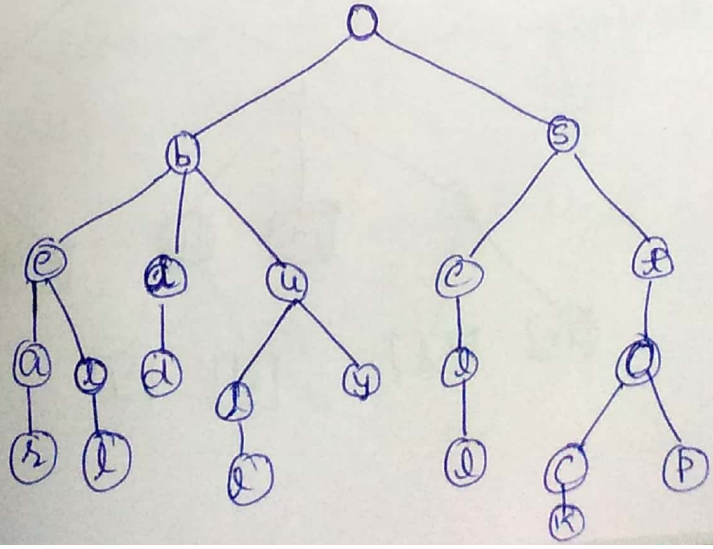
– Standard Tries :

The standard trie for a set of strings S is an ordered tree such that:

- Each node node but the root is labelled with a character
- the children of a node are alphabetically ordered from left to right.
- the paths from the external node to the root yield the strings of S.

Example:

S= bear, bell, bid, bull, buy, sell, stock, stop



P7o

↳ Running time for operations:

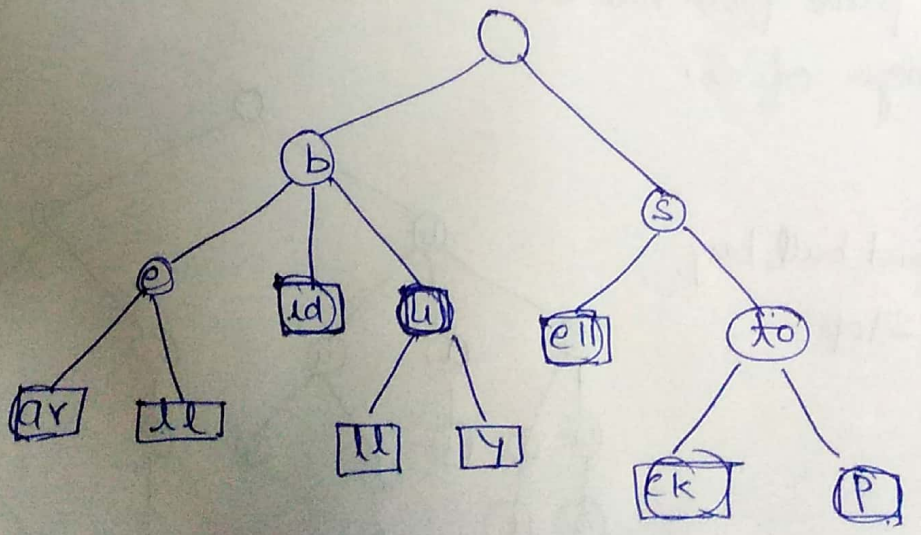
- A standard tries uses O(W) space.
- Operations find, insert, delete take O(dm) time, where
  - W = total size of strings in S.
  - m = size of string involved in operation.
  - d = alphabet size.

- Application of Tries:

- (i) Word matching: find the first occurance of word X in the text.
- (ii) prefix matching: find the first occurance of the longest prefix of word X in the text.

Note - Each operation is performed by tracing a path in the trie starting at the root.

- Compressed Tries:
  - Trie with nodes of degree at least 2.
  - Obtained from standard trie by compressing chains of redundant nodes.



LECTURE-24

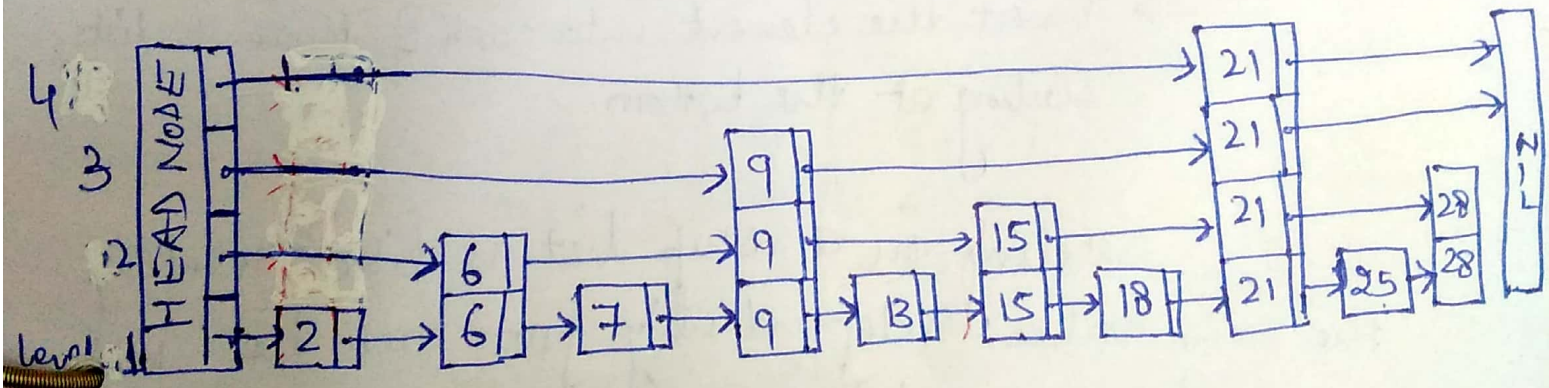
Skip List

→ Founded by William Pugh in 1989.

→ Randomized data structure

→ Easy to implement (as compared to search trees)

→ A skip list is a data structure that allows fast search (logn) with an ordered sequence of elements fast search is made possible by maintaining a linked hierarchy of subsequences, with each successive seq: subsequence skipping over fewer elements than the previous one.



→ Each link of the sparser lists skips over many items of the full list in one step. These forward links may be added in randomized way (with geometric or negative binomial distribution).

→ A level k element is a list element that has k forward pointers.

→ Operations on skip list:

① Search: For searching, we scan through the top list until we find the search key or a node with smaller key that has a link to a node with a larger key. Then, we move second-from-top list and iterate the procedure, continuing until the search key is found, or a search miss happens on the bottom level.

② Insert: → Find the place for the new element
→ Assign to it its level k computed by flipping the coin.
→ Insert the element into each of those k lists, starting at the bottom.

③ Delete: Deleting on a skip list is like deleting the same value independently from each list in which forward pointers of the deleted elements are involved.

↦ Analysis:

	Average case	Worst case
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)