07/09/20

Data Structures

Linear Data Structures

1. Arrays

→ 1-D array

→ 2-D array

- upper triangular matrix
- Lower triangular matrix
- Toeplitz matrix
- S-matrix
- C-matrix
- Tri-diagonal matrix
- Square band matrix

→ 3-D array

Ref books
→ DS by Sahani

Programming & DS
15 to 18
Around 7th from DS

2. Stack and Queue

→ Operations on stack
- push, pop, isempty, peep, change

→ Applications of stack
- stack permutation
- ★ Recursion.
- Converting infix to prefix
  infix to postfix
- Constructing expression tree using prefix & using postfix
- Multiple stack implementation

→ Queue (least priority)
- operations on queue
- linear, circular, double ended queue
- Implementation of queue using multiple stacks.
- Implementation of stack using queue

3- Linked List

* → Single linked list
  - Traversing
  - no of nodes.
  - merging two lists
  - Reversing
  - operations
    - Insertion
    - Deletion
→ Doubly Linked list

* → Circular Linked list

Non-Linear Data Structures:

1. Tree

→ Properties of trees.

→ Complete binary tree, full binary tree, k-ary tree.

→ Tree traversals

- Inorder, preorder, postorder
- Converse of inorder, preorder, postorder.

→ Construction of ~~binary~~ a unique binary tree using

(i) Inorder & Preorder

(ii) Inorder & postorder

→ Binary search tree

- Insertion, deletion, searching

→ Balanced binary search tree (AVL Tree)

- Max no of nodes
- Min no of nodes
- Insertion, deletion

★ → Heap Tree ----- There has been a question almost every yr (9 out of 10)

- Insertion
- Deletion
- ~~Max~~ Finding Max & Min elements
- Construction of heap tree
  - Inserting keys one after other in given order.
  - Inserting using heapify method (Build heap)
- Heap Sort

2. Graphs

→ BFS (Breadth First Search Tree)

→ DFS (Depth First Search Tree)

3. Hashing

→ Advantages of hashing

→ Hash function

- good hash function
- collision
  - Collision Resolving techniques
    - ★(i) Linear probing (90%)
    - (ii) Quadratic probing
    - (iii) Double hashing
    - (iv) Separate chaining

Previous Year Question Analysis

1987 - 2020 .... 170 questions (approximately)

V 1. Arrays 5%

II 2. Stack & Queues 15%

VI 3. Linked list 5%

I 4. Tree

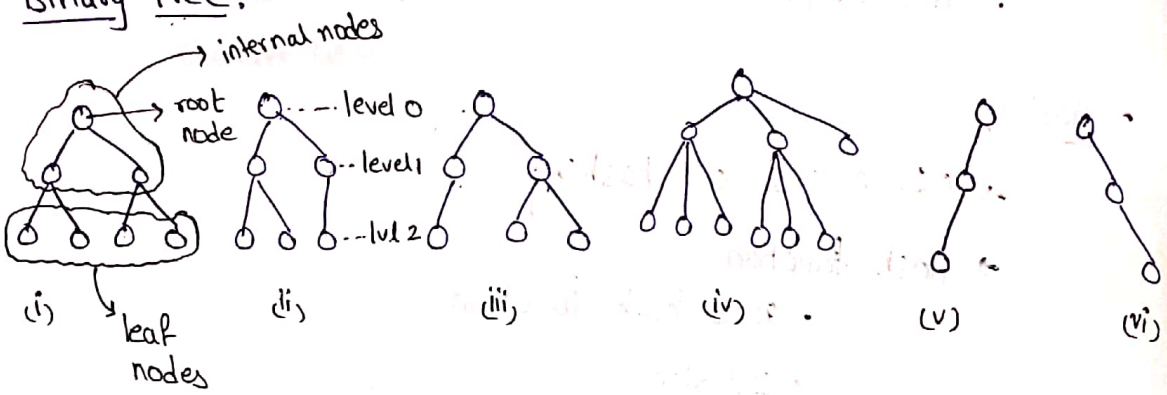
Binary tree
BST
AVL
Heap tree
} 65%

IV 5. Graph 5%

III 6. Hashing 5%

Trees:

Binary tree:



→ In a tree if every node contains atmost 2 children then it is called binary tree.

∴ (iv) is not a binary tree

Full binary tree:

The binary tree in which all levels are full is called full binary tree.

Eg: fig (i)

Every full binary is complete binary tree

Complete binary tree:

In a complete binary tree, all levels are full except possibly last level and that last level is filled from left to right.

Eg: fig (i), fig (ii)

fig (iii) is not complete binary tree cuz last level is not filled from left to right.

Note:

→ (v) is called left skewed binary tree

→ (vi) is called right skewed binary tree.

k-ary tree:

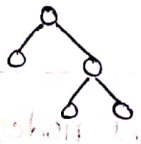
In a k-ary tree every internal node contains exactly k-children.

∴ fig(i) is 2-ary tree

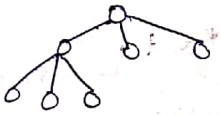
fig (ii) & fig (iii) are not 2-ary trees.

fig(iv) is 3-ary tree.

Eg:



is 2-ary tree



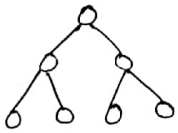
is 3-ary tree

(Q1) In a complete k-ary tree, if there a n-internal nodes, then how many no of leaf nodes are possible.

a) nk b) n(k-1) c) n(k-1)+1 d) k(n-1)+1

Sol:

Consider



This is 2-ary tree

∴ k=2

Here no of internal nodes, n=3

Here we have no of leaf nodes =4

a) n*k = 3(2) = 6

b) 3(2-1) = 3

c) 3(2-1)+1 = 4

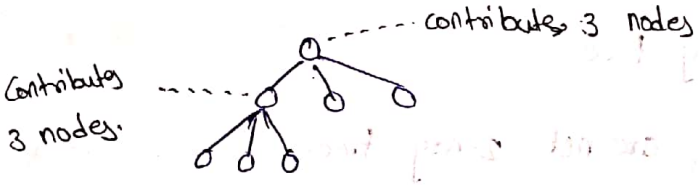
d) 2(3-1)+1 = 5

∴ opt (c)

Method 2: (Derivation)

total no of nodes = total no of internal nodes + total no of leaf nodes

Consider a 3-ary tree



Here total no of nodes

= (no of internal nodes) * 3 + 1
↳ root

Silly in a k-ary tree if no of internal nodes = n

then total no of nodes = n * k + 1
↳ root

↙ Contribution of each internal node.

Now no of leaf nodes = total no of nodes − internal nodes

= nk + 1 − n

no of leaf nodes = n(k−1) + 1

∴ opt (c)

(Q2) In a complete n-ary tree, if there are I no of internal nodes and L-no of leaf nodes. For I = 10 and L = 41, find the value of n

Sol:

total no of nodes = internal + leafs

I * n + 1 = I + L ⇒ 10n + 1 = 10 + 41

⇒ n = 5

(Q3) In a binary tree, there are 10 internal nodes of degree 2 and 5 internal nodes of degree 1. Find no of leaf nodes.

↳ no of child nodes.

Sol:

total nodes = internal + leaf

$(10*2 + 5*1) + 1 = (10+5) + l$

$l = 26 - 15$

$l = 11$

∴ 11 leaf nodes

(Q4) In a binary tree if there are 20 leaf nodes, then how many no of nodes are having degree 2?

Sol:

Note:

Since it is a binary tree, the result will be independent of no of internal nodes with degree 1.

let x be no: of internal with degree 1

i be no of internal nodes with degree 2.

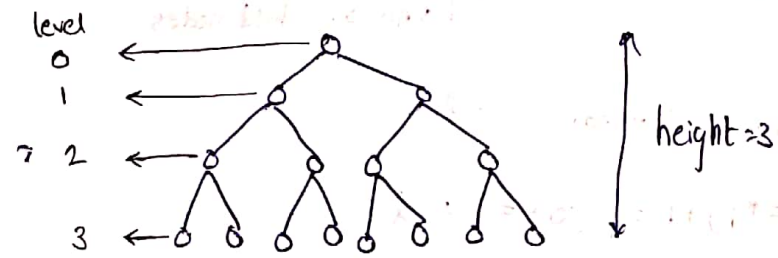
$(i*2 + x*1) + 1 = (x+i) + 20$

$2i + x + 1 = x + i + 20$

$\Rightarrow i = 19$

08/09/20

Properties of full binary tree:



→ Height:

* Height of leaf node is always zero

Height of any internal node = longest path from that node to a leaf node

Height of the tree = height of the root node
(tree → Full binary tree)

Note:

(→ Sometimes height may also be defined as no of levels too.
In exam, it will be defined)

* Height of full binary tree = $\log_2$ (no of leaves)

= $\log_2$ (no of internal nodes + 1)

→ At any level i, we have 2^i nodes.

→ total no of nodes in a full binary tree is $2^{h+1}-1$

h → height of the FBT

→ total no of nodes in a full k-ary tree is

$= k^0 + k^1 + k^2 + \dots + k^h$

$= \frac{1(k^{h+1}-1)}{k-1}$

h → height of the tree.

$= \frac{k^{h+1}-1}{k-1}$

→ Total no of nodes of height 'k' = $\left\lceil \frac{n}{2^{k+1}} \right\rceil$

n :→ total no of nodes

→ Sum of heights of all nodes = $\sum_{k=0}^{h} \left\lceil \frac{n}{2^{k+1}} \right\rceil \times k$

where h ... height of the tree.
n .... total no of nodes in the tree.

$$\sum_{k=0}^{h} \left\lceil \frac{n}{2^{k+1}} \right\rceil \times k = \frac{n}{2} \sum_{k=0}^{h} \frac{k}{2^k}$$

Consider

$$\sum_{k=0}^{\infty} x^k = 1 + x + x^2 + \cdots \infty \quad \text{for, } x < 1$$

$$= \frac{1}{1-x}$$

diff w.r.to x

$$\sum_{k=0}^{\infty} k \cdot x^{k-1} = \frac{1}{(1-x)^2}$$

$$\Rightarrow \sum_{k=0}^{\infty} k \frac{x^k}{x} = \frac{1}{(1-x)^2}$$

$$\Rightarrow \sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2}$$

now $\sum_{k=0}^{h} \frac{k}{2^k} \leq \sum_{k=0}^{\infty} \frac{k}{2^k}$

Here $x = 1/2$

$$\therefore \sum_{k=0}^{h} \frac{k}{2^k} \leq \frac{1/2}{(1-1/2)^2}$$

$$\Rightarrow \sum_{k=0}^{h} \frac{k}{2^k} \leq 2$$

now $\frac{n}{2} \sum_{k=0}^{h} \frac{k}{2^k} \leq \frac{n}{2} \times 2$

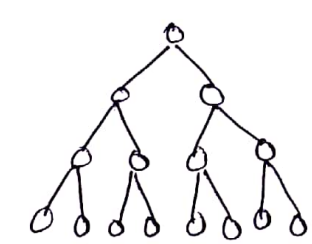
$$\therefore \frac{n}{2} \sum_{k=0}^{h} \frac{k}{2^k} \leq n$$

i.e., $\frac{n}{2} \sum_{k=0}^{h} \frac{k}{2^k} = O(n)$

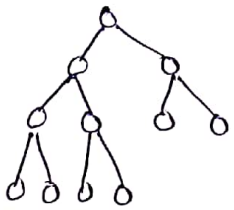
For exact value for sum of heights in an FBT height 'h'
$= 2^{h+1} - h - 2$

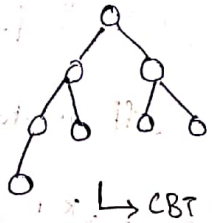
∴ Sum of heights of all nodes = $O(n)$

Properties of Complete Binary tree:



CBT of height 3 with max no of nodes.





↳ CBT of height 3 with min of nodes

→ Max no of nodes of CBT of height (h) = $2^{h+1} - 1$

Min no of nodes of CBT of height (h) = 2^h

∴ If 'n' is no of nodes in CBT of height 'h' then

$$2^h \leq n \leq 2^{h+1} - 1$$

→ Max height of CBT is bounded by $O(\log n)$

Proof:

n .... total no of nodes of CBT

w.k.T $2^h \leq n \leq 2^{h+1} - 1$

$2^h \le n$

$h \le \log_2 n$ — ①

$n \le 2^{h+1} - 1$

$n+1 \le 2^{h+1}$

$\log_2 (n+1) \le h+1$

$\log_2 (n+1) - 1 \le h$ — ②

from ① & ②

$$\log_2 (n+1) - 1 \le h \le \log_2 n$$

$$\therefore h = O(\log_2 n) = O(\log n)$$

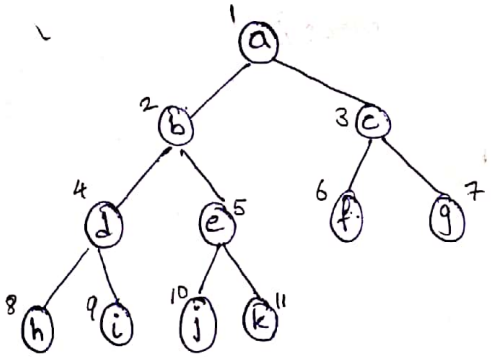
$\because \log_2 n = \frac{\log n}{\log 2}$

$O(\log_2 n)$

$= O\left(\frac{\log n}{\log 2}\right)$

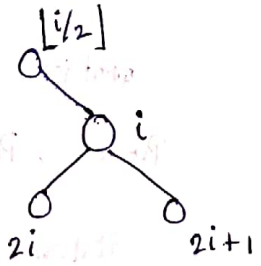
$= O(\log n)$

Array representation of binary tree



This representation can be used for any tree

general indices of nodes



→ Root node index is 1.

→ For any node whose index 'i', its left child at index 2i and its right child at index 2i+1, and its parent is at $\lfloor \frac{i}{2} \rfloor$

→ Advantage:

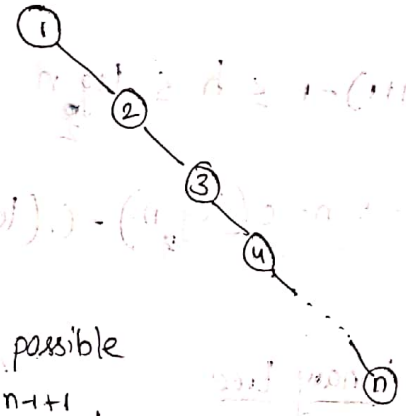
Consumes less spaces, as we don't need to store pointers (left & right pointers) for arrays.

09/09/20

(Q5) What is the minimum size of the array in order to store any type of binary tree, having 'n' nodes?

Sol:

If the binary tree is right skewed then the leaf node has the maximum index value.



Here height = n-1

∴ Max no of nodes possible

$= 2^{n-1+1} - 1$

$= 2^n - 1$

index: $2^n - 1$

∴ Size of array = $2^n - 1$

Tree Traversals

type of questions:

Given	What is
(i) Binary tree diagram	Preorder, Postorder, Inorder
(ii) Preorder + Inorder of binary tree	Postorder
(iii) Postorder + Inorder of a binary tree	Preorder
(iv) Binary Search tree	Inorder, Preorder, Postorder
(v) Preorder of BST	Postorder
(vi) Postorder of BST	Preorder
(vii) Inorder + Preorder	resultant binary tree

(viii) Inorder + Postorder — resultant binary tree

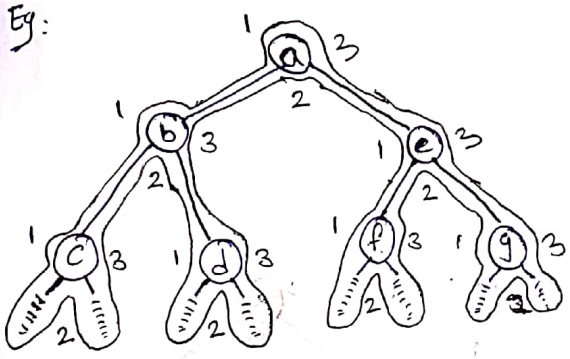
(ix) Inorder, Preorder, Postorder — Time complexity

(x) Converse of Inorder, Preorder, Postorder of a binary tree.

Inorder:

1) Traverse the left subtree of root node in Inorder
2) Print the root node.
3) Traverse the right subtree of rootnode in inorder

Eg:



we touch every three times

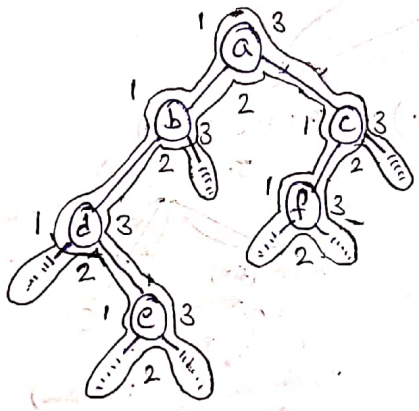
traversal	when to print
inorder	bottom
preorder	left
postorder	right

Inorder: c b d a f e g (all 2's)

Preorder: a b c d e f g (all 1's)

Postorder: c d b f g e a (all 3's)

Eg:



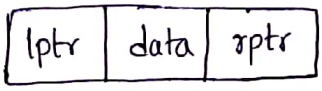
Inorder: d e b a f c

Preorder: a b d e c f

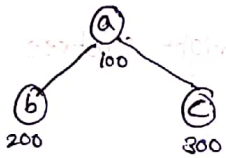
Postorder: e d b f c a

Recursive approach for tree traversals

node structure

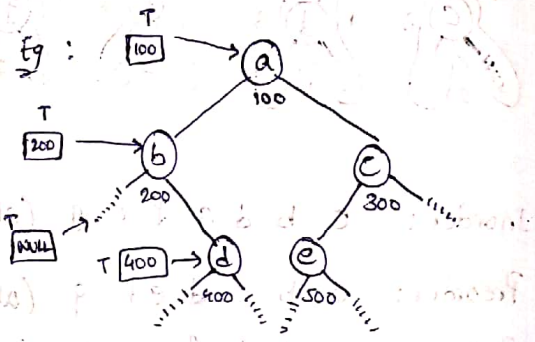


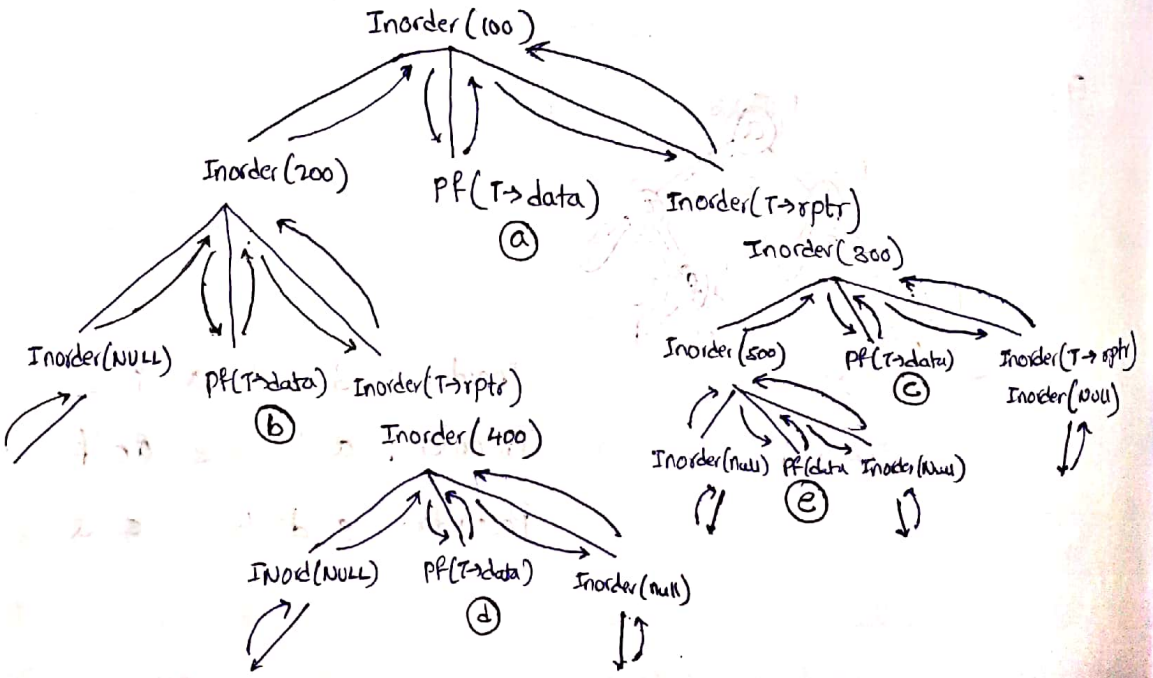

```
Struct node
{
    Struct node* lptr;
    Struct node* rptr;
    int data;
};
Struct node * T;
```



→ Inorder (Struct node *T)

```
{
    if (T==NULL)
        return;
    Inorder (T->lptr);
    printf ("%d", T->data);
    Inorder (T->rptr);
}
```





∴ Inorder traversal: b d a e c

→ Preorder (struct node *T)

```
{
    if(T=NULL)
        return;
    printf("%d", T→data);
    preorder(T→lptr);
    preorder(T→rptr);
}
```

→ Postorder (struct node *T)

```
{
    if(T==NULL)
        return;
    Postorder(T→lptr);
    Postorder(T→rptr);
    printf("%d", T→data);
}
```

Note:

→ For a given binary tree, no of possible inorder traversals = 1

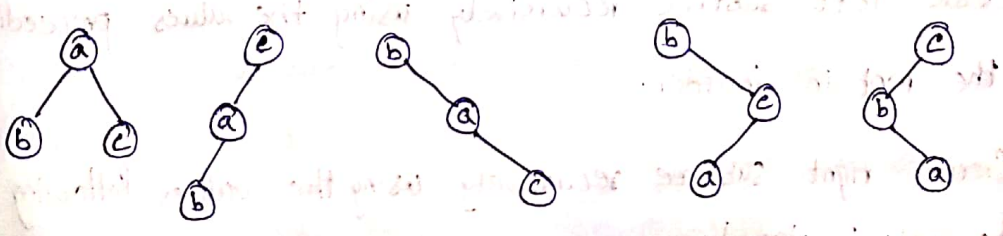
→ For a given ~~binary~~ inorder traversal, with n elements

no of binary trees possible = $\frac{1}{n+1} {}^{2n}C_n$

* ~~Proof~~:

Eg:

assume the inorder traversal = b a c



For all above trees inorder traversal is b a c

for n=3, no of binary trees with given inorder traversal

$$= \frac{1}{3+1} {}^{2(3)}C_3$$

$$= \frac{1}{4} {}^{6}C_3 = 5$$

→ Silly, for the given preorder sequence of n elements,

no of binary trees possible = $\frac{1}{n+1} {}^{2n}C_n$

→ Silly for given postorder sequence of n elements

no of binary trees possible = $\frac{1}{n+1} {}^{2n}C_n$

Note:

→ for given preorder & inorder there exists unique binary tree.

→ " " postorder & inorder " " " "

→ However we cannot build a unique tree (i.e., more than one exists) for given preorder & postorder.

Note:

→ Time complexity of inorder, preorder & ~~poster~~ ~~pre~~ postorder (~~Calculated~~ iteratively ~~recursively~~) is bounded by ~~sim~~ sum of heights of all nodes.

∴ time complexity (iteratively) = O(n)

13/09/20

Construction of unique binary tree using preorder & Inorder

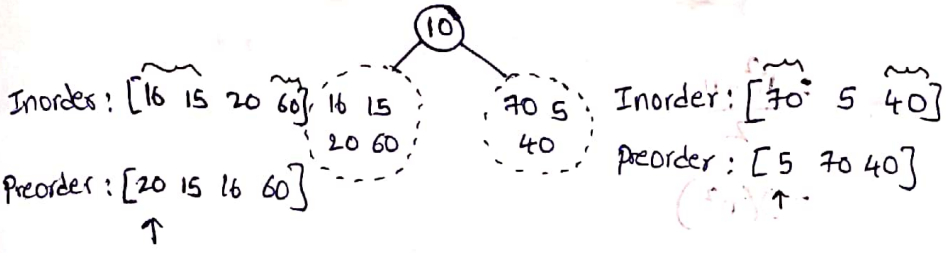
Step1: pickup the first key of preorder and create a node of it.

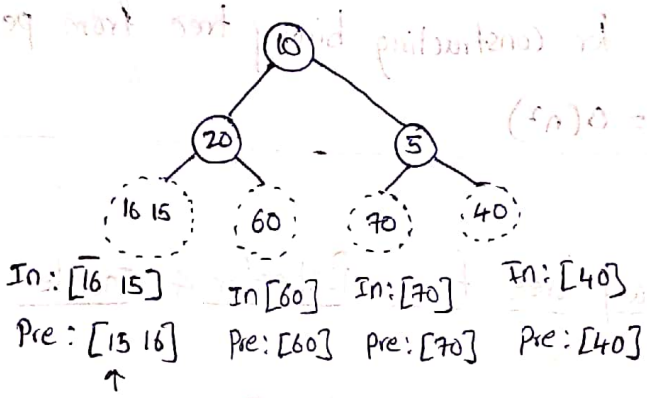
Step2: Create left subtree recursively using the values preceeding the root in inorder.

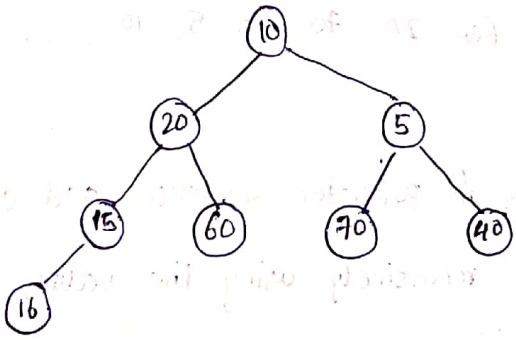
Step3: Create right subtree recursively using the values following the root in inorder.

Eg: Inorder: 16 15 20 60 10 70 5 40
(left subtree | right subtree)

Preorder: 10 20 15 16 60 5 70 40
(left subtree | right subtree)







Analysis:

For analysis we need to consider worst case
→ Here this worst case occurs for left skewed tree.

Assume we have 'n' nodes

→ creating node for 1st element of preorder takes constant time.

→ Now we need to search for that node in the inorder seq.

→ In worst case we need n comparisions.

→ If we continues this, in the next search no of comparision required would be n-1 (worst case)

∴ total no of comparisions

$$= n + (n-1) + (n-2) + \cdots + 1$$

$$= \frac{n(n+1)}{2}$$

$$= \frac{n^2+n}{2}$$

$$= O(n^2)$$

∴ time complexity for constructing binary tree from preorder and inorder = $O(n^2)$

Construction of binary tree from Postorder & Inorder

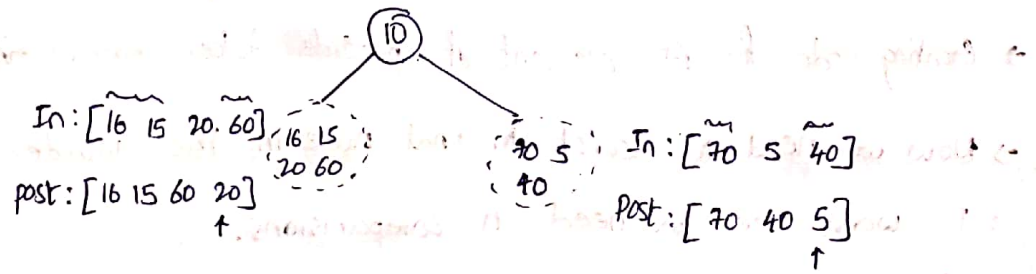
Inorder : 16 15 20 60 10 70 5 40

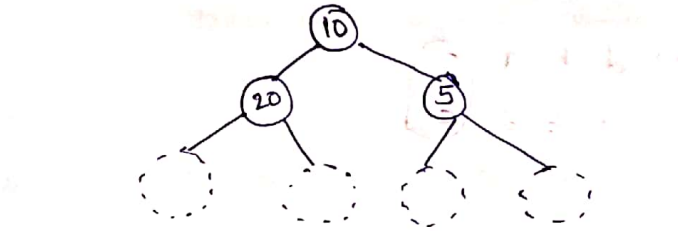
postorder : 16 15 60 20 70 40 5 10

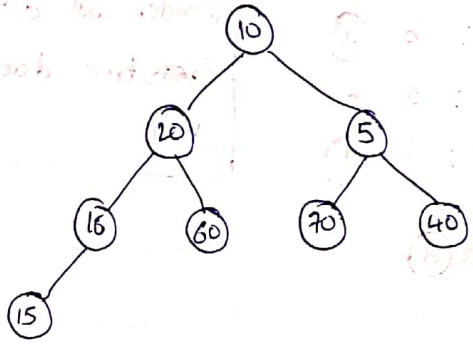
Step 1 : Pickup the last key of postorder sequence and create a node.

Step 2 : Create left subtree recursively using the values preceeding the root in inorder

Step 3 : Create right subtree recursively using the values following the root ~~in~~ in the inorder.







Analysis :

Same as previous analysis

Here also worst case is for left skewed tree.

∴ time complexity : $O(n^2)$

(Q6) Let LASTIN, LASTPRE, LASTPOST represent the last key of inorder, preorder, postorder traversals of a complete binary tree. Then which of the following is true?

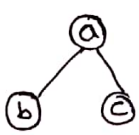
a) LASTIN = LASTPRE

b) LASTPRE = LASTPOST

c) LASTPOST = LASTIN

d) none of these

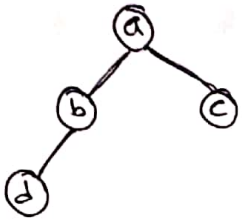
Sol :



In : b a c

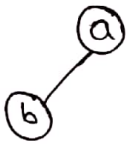
Pre : a b c

Post : b c a



In: d b a c
Pre: a b d c

∴ opt (a)



In: b a
Pre: a b
post: b a

∴ opt (d)

Last elements of preorder and inorder will differ if the ~~trees~~ tree doesn't contain RST

(Q7) The following three are known to be preorder, inorder, postorder traversals of a binary tree. But it is not known which one is which traversal of the binary tree.

I. MBCAFHPYK
II. KAMCBYPFH
III. MABCKYFPH

a) I - inorder III - postorder.
b) I - Inorder III - preorder
c) II - inorder III - preorder
d) III - inorder II - preorder

Sol:

root is 1st key of preorder & last key of postorder.

∴ 1st key of preorder = last key of postorder.

∴ I - post order
II - pre order.
⇒ III - inorder

∴ opt (d)

(Q8) which of the following combination creates a unique binary tree

I a) Inorder & preorder

II b) Inorder & postorder

III c) preorder & postorder

a) I, III b) II, III c) I, II d) I, II, III

∴ opt (c)

(Q9) How many minimum and maximum no of nodes of binary tree are possible if the height of the binary tree is 6.

Sol:

→ (need not to be left or right skewed)

Min nodes ⇒ one node per level = 7

Max nodes ⇒ full binary tree of height 6

$\Rightarrow 2^{6+1}-1 = 127$

(Q10) Let T be a binary tree with 15 nodes. Then the minimum and maximum possible height of binary is

Sol:

If n is no of nodes and h is height the

$$h+1 \le n \le 2^{h+1}-1$$

$n \ge h+1$

$15 \ge h+1 \Rightarrow h \le 14$

$n \le 2^{h+1}-1$

$15 \le 2^{h+1}-1 \Rightarrow 2^{h+1} \ge 16$

$h+1 \ge 4$

$h \ge 3$

$\therefore 3 \le h \le 14$

$\therefore$ min height = 3

max height = 14

→ min height is obtained for complete binary tree

→ Max height is possible if we have exactly one node per level

(Q11) Consider the following nested representation of a binary tree. (x, y, z): indicates that y & z are left subtree and right subtree respectively of node x. y, z may be NULL or further nested. Which of the following is valid representation of binary tree.

a) (1 2 (4 5 6 7))

b) (1 ((2 3) 5 6) 7)

c) (1 (2 3 4)(5 6 7))

d) (1 (2 3 NULL)(4 5))

Sol:

a) (1 2 (4 5 6 7))

left subtree

↳ cannot have 4 elements

∵ It should be of type (x, y, z)

∴ invalid

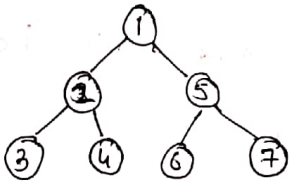
b) (1 ((2,3) 5 6) 7)

invalid

left subtree

right

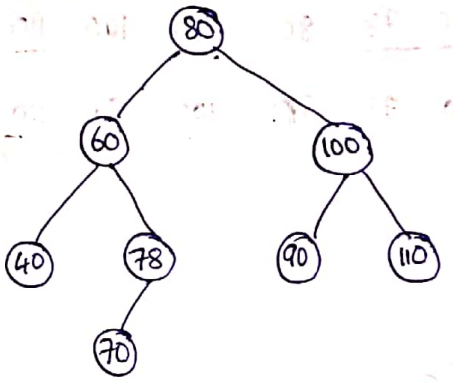
c)



d) right subtree has only 2 elements

∴ invalid.

Binary Search Tree:



values of LST(80) $\leq$ 80 $\leq$ values of RST(80)

values of LST(60) $\leq$ 60 $\leq$ values of RST(60)

values of LST(100) $\leq$ 100 $\leq$ values of RST(100)

Defn: For every node 'k' of binary tree if it satisfies the property where values of LST(k) $\leq$ k $\leq$ values of RST(k) then such binary tree is called binary search tree.

Consider

Inorder:	40	60	70	78	80	90	100	110
Preorder:	80	60	40	78	70	100	90	110
Postorder:	40	70	78	60	90	110	100	80

****

Note:

The inorder sequence of BST is always in ascending order. This condition is bidirectional.

Model 1: If the preorder sequence of BST is

80 60 40 78 70 100 90 110

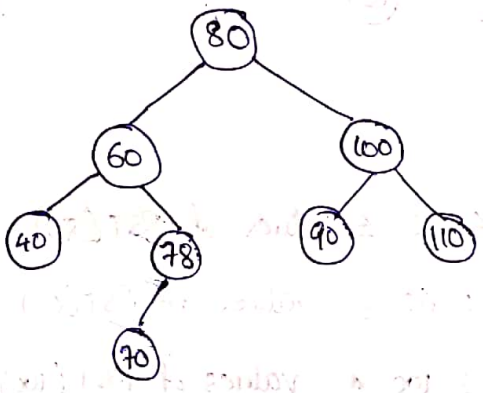
then what is post order sequence.

Sol:

Inorder is in increasing order

∴ Inorder: 40 60 70 78 80 90 100 110

Preorder : 80 60 40 78 78 100 90 110



Now postorder sequence is

40 70 78 60 90 110 100 80

Analysis (Construction of BST from inorder & preorder)

→ Here we choose the first element of preorder and create a node.

→ Now we search for this node ~~using~~ in the inorder sequence using log n.

~~→ Also~~

∴ total no of comparisions

$= \log n + \log(n-1) + \log(n-2) + \cdots + \log 1$

$= \log(n(n-1)(n-2)\cdots(1))$

$= \log(n!) \leq \log(n^n) \leq n \log n$ ~~∵ A ≤~~

$= O(n \log n)$ $(\because n! \leq n^n)$

Q12) You are given the post order traversal 'P' of a BST on n elements : [1, 2, 3 ... n]. You have to determine the unique binary search tree that has P as post order traversal. Then what is the time complexity of most efficient algorithm for doing this.

a) $O(n^2)$ b) $O(n \log n)$ c) $O(\log n)$ d) $O(n)$.

Sol:

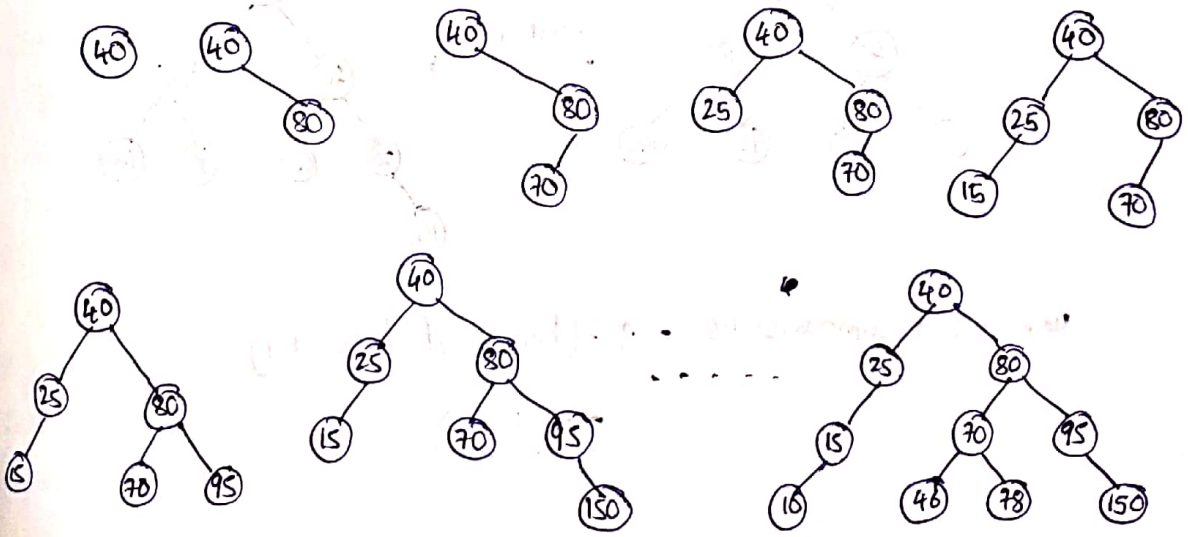
time to sort = $O(n \log n)$ ... Heap sort

time to build tree = $O(n \log n)$

$\therefore O(n \log n)$

Model 2 : Construct BST by inserting the following keys in the given order and find height of the binary search tree. Find no of nodes in the left subtree of root node. List out all leaf nodes of the BST.

keys : 40 80 70 25 15 95 150 78 46 10



height of the BST : 3

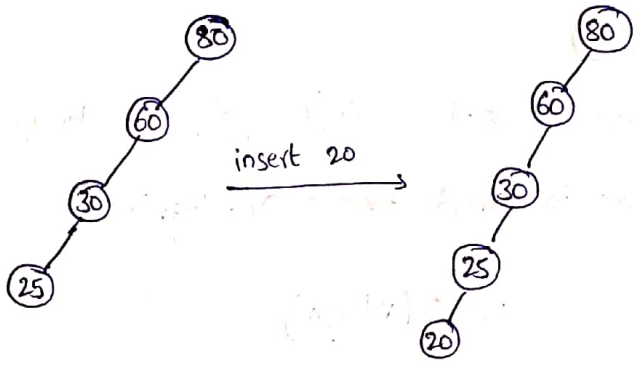
no of nodes in left subtree of the root : 3

leaf nodes : 10 46 78 150

Analysis (Inserting node into BST)

Case (i):

Insert 20 into following BST



Here no of comparisions = 4

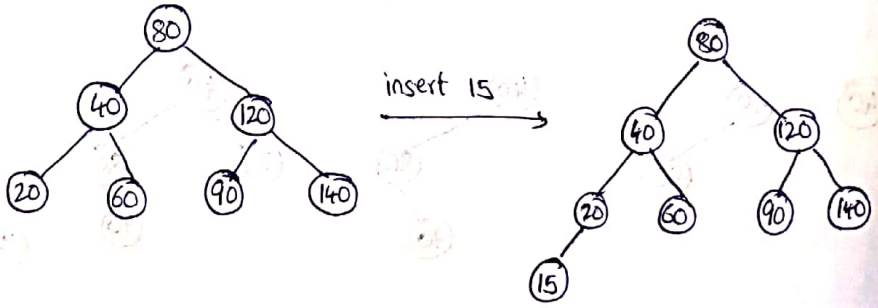
= no of nodes = height + 1

∴ time complexity = O(n)

↳ no of nodes

Case (ii):

Insert 15 into the following BST



Here no of comparisions = 3 = (height of tree + 1)

$= O(\log_2 n)$

Note :

time complexity for ~~inde~~ ~~inser~~ insertion into BST $= \begin{cases} O(n) & \text{---- left or right skew (or 1 node per level)} \\ O(\log_2 n) & \text{---- full binary search tree} \end{cases}$

(Q13) Suppose the numbers 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 are inserted into an initially empty BST. what is inorder traversal of resultant tree?

a) 7 5 1 0 3 2 4 6 8 9

b) 0 2 4 3 1 6 5 9 8 7

c) 0 1 2 3 4 5 6 7 8 9

d) 9 8 6 4 2 3 0 1 5 7

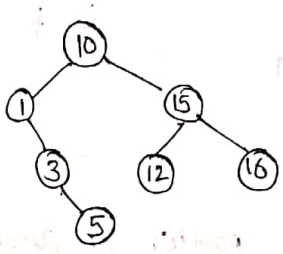
Sol :

Inorder traversal is increasing order

∴opt ©

(Q14) The following numbers are ~~into em~~ inserted into an empty BST in the given order : 10 1 3 5 15 12 16. what is height of the BST (the height is max distance of leaf node from root)

Sol :

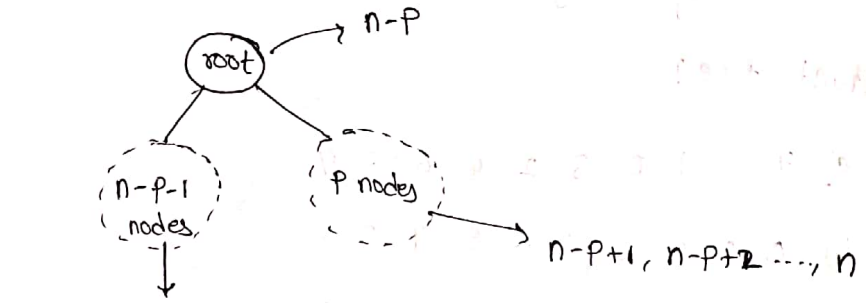


∴ height = 3

(Q15) The numbers 1, 2, ... n are inserted in a BST in some order. In the resulting tree, the right subtree of the root contains p nodes. The first number to be ~~inserted~~ inserted in the tree must be _____

a) p b) p+1 c) n-p d) n-p+1

Sol:

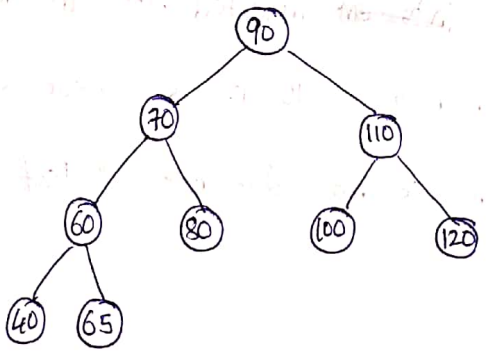


∴ 1, 2, 3, ... n-p-1

∴ root = n-p is 1st node to be inserted.

(Q16) Deletion operation on BST:

Case (i): Deleting leaf node



Step 1: Search if the given key is present or not

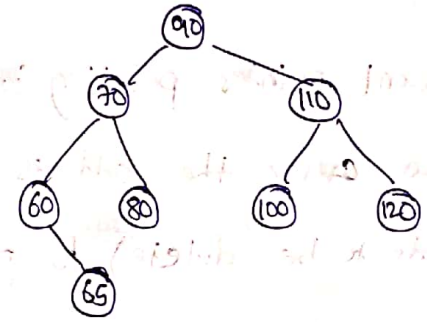
Step 2: Identify the parent pointer pointing to the node to be deleted and assign it to NULL.

Delete 40 from above BST

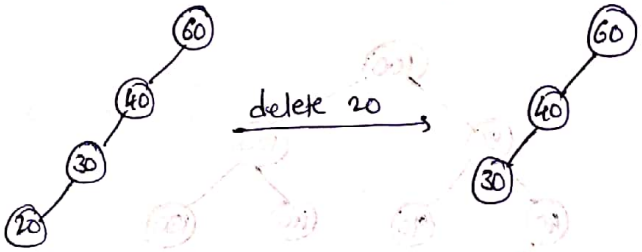
→ Now search for (40)

→ Now ~~make poi~~ assign ~~initialize~~ the pointer of parent of (40) pointing to (40), to NULL

resulting tree is



Eg 2: Delete 20 from below BST



∴ no of comparisions = $h+1$ ⟵ height

$= n$ (no of nodes)

↳ In the worst case.

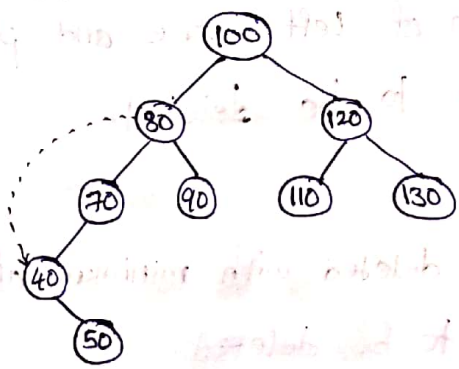
$= O(n)$

∴ time complexity for deleting node from BST $= \begin{cases} O(\log n) & \text{--- complete BST} \\ O(n) & \text{--- one noder per level} \end{cases}$

∴ ~~west~~ worst case time complexity $= O(n)$

Case (ii): Deleting a node ~~with~~ which has only one subtree,

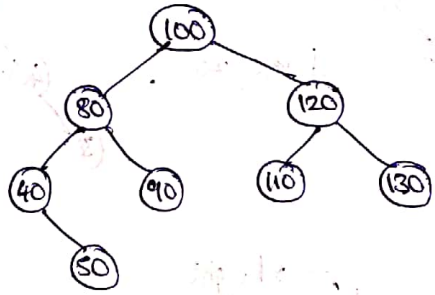
Eg: Delete node 70



Step (i) : Search the key

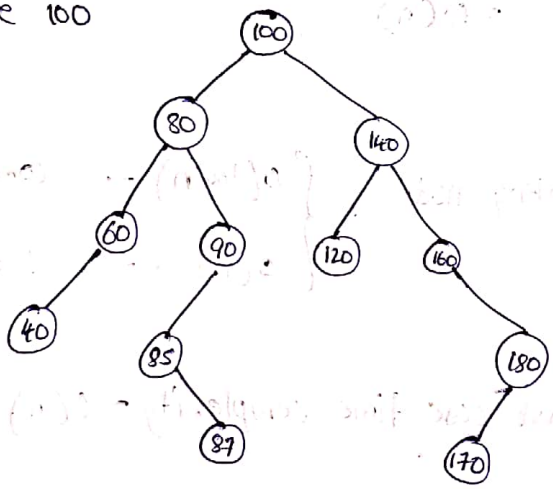
Step (ii) : Identify the parent pointer pointing to the node to be deleted. Now assign the address of the child node (of the node to be delete) to parent node's pointer.

∴ Resultant tree is



Case (iii) : Deleting a node with both left & right subtrees.

Delete node 100



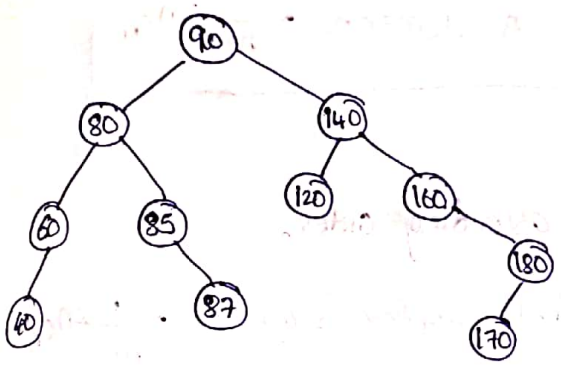
Method 1:

Find the maximum of Left subtree and place in the place of the node to be deleted.

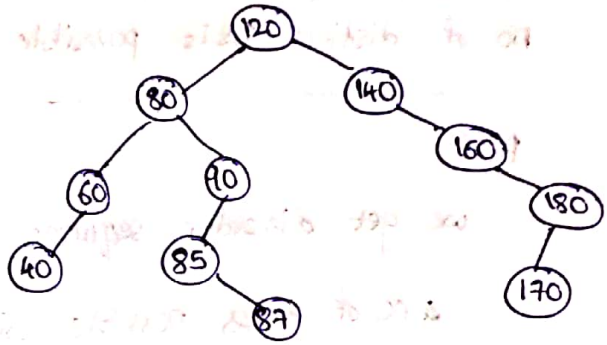
Method 2:

Replace the node to be deleted with minimum of right subtree of the node to be deleted.

Method 1 :



Method 2 :



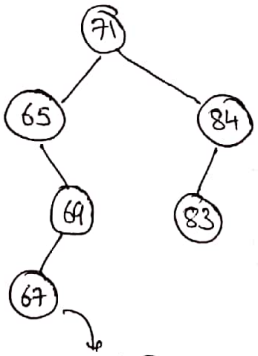
→ In both case (ii) & case (iii) also, worst case time complexity is $O(n)$

∴ time complexity for deletion from BST $= O(n)$

(Q.16) while inserting the elements 71 65 84 69 67 83 in an empty BST in the sequence show, the element in lowest level is

a) 65 b) 67 c) 69 d) 83

Sol:



∴ opt (b)

(Q.17) ~~How many no of binary search trees are possible by storing the following keys in the given order.~~

Note:

No of distinct BSTs possible for n elements = $\frac{1}{n+1} 2nC_n$

Proof:

we get a inorder sequence in ascending order.

∴ no of trees possible with that inorder sequence = $\frac{1}{n+1} 2nC_n$

No of binary trees possible with 'n' unlabled nodes = $\frac{1}{n+1} 2nC_n$

No of binary trees possible with 'n' distinct nodes = $\frac{n!}{n+1} 2nC_n$

(Q17) Suppose that we have numbers b/w 1 and 100 in a BST, and want to search for the number 55. which of the following sequences cannot be the sequence of nodes examined?

a) {10, 75, 64, 43, 60, 57, 55}

b) {90, 12, 68, 34, 45, 55}

c) {9, 85, 47, 68, 43, 57, 55}

d) {79, 14, 72, 56, 16, 53, 55}

Sol:

Method:

A sequence is valid iff at every element of the sequence, either all the elements right to it are smaller than the element or all are larger (from property of BST)

a) 10 75 64 43 60 57 55

all are larger than 10

75 64 43 60 57 55

all are smaller than 75

64 43 60 57 55

all are smaller than 64

43 60 57 55

all are larger

60 57 55

all are smaller

57 55

all are smaller

∴ valid sequence

c) 9 85 47 68 43 57 55

68 is larger than 47

43 is smaller than 47

∴ invalid sequence.

(Q18) Number of ways in which the numbers 1, 2, 3, 4, 5, 6, 7 can be inserted in an empty binary search tree such that the height of resulting tree is 6 is _____ (height of tree with single node is 0)

Sol:

no of nodes = 7

height = 6

∴ nodes per level = 1

At each level we can choose one of the two pointers and point it to the node in next level.

∴ no of possible BSTs $= 2^6 = 64$

15/09/20

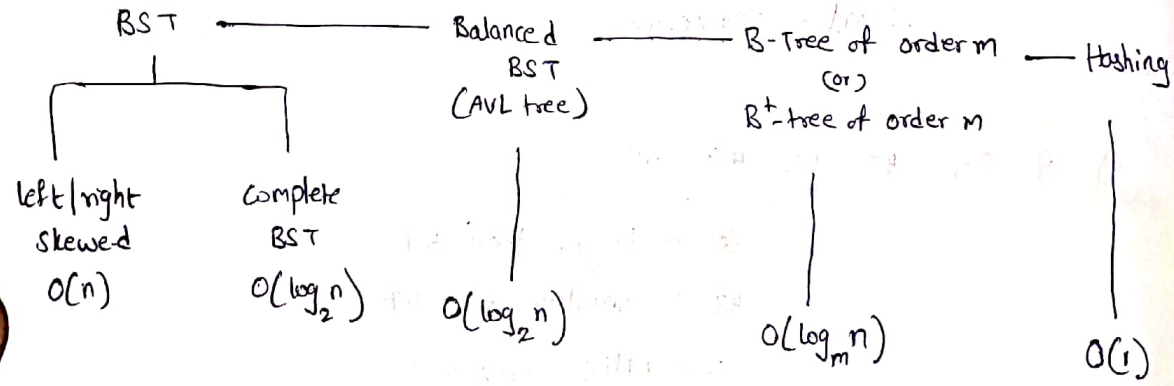
AVL - Tree : (Balanced BST)

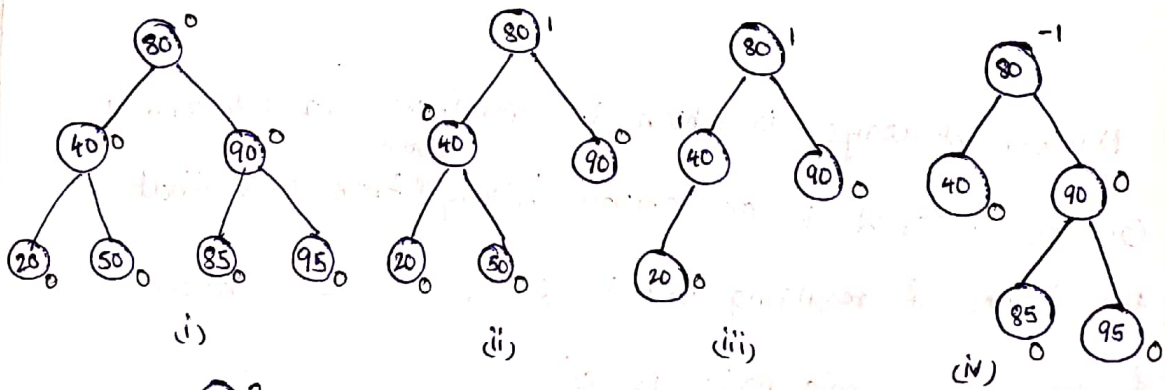
[Adelson - Velski - Landis]

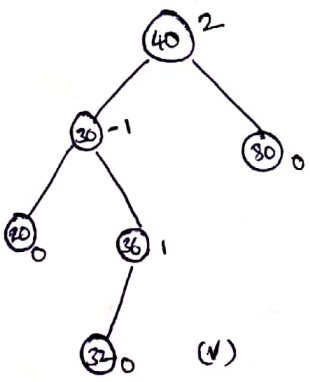
→ If BST is left or right skewed then search time is $O(n)$

→ If BST is complete then search time is $O(\log_2 n)$

Search time







Balancing factor: Height of left subtree − Height of right subtree

Defn of AVL tree: A BST is called AVL tree if balancing factor of every node is −1 (or) 0 (or) 1

Eg: (i) (ii) (iii) (iv) are AVL trees

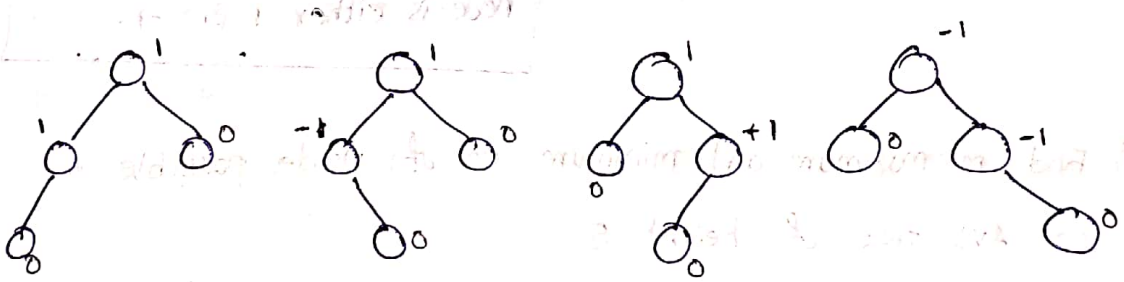
(v) is not an AVL tree

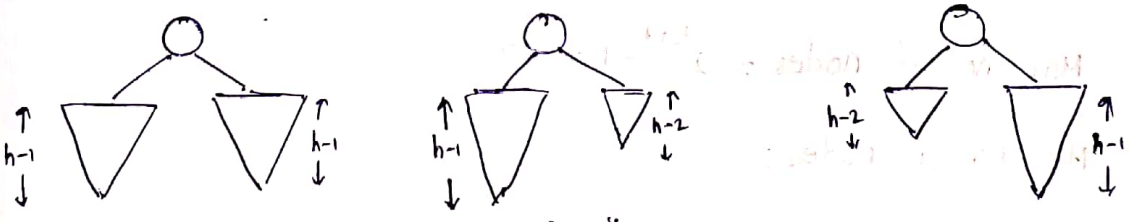
Note:

Max no of nodes possible in an AVL tree of height 'h' = $2^{h+1} - 1$ (i.e., FBT)

Min no of nodes possible in an AVL tree of height 'h'.

Eg: for h = 2, AVL tree with min no of nodes





case (i) case (ii) case (iii)

Min no of nodes are possible in case (ii) & case (iii)

Let n(h) - denote minimum no of nodes possible in an AVL tree of height 'h'.

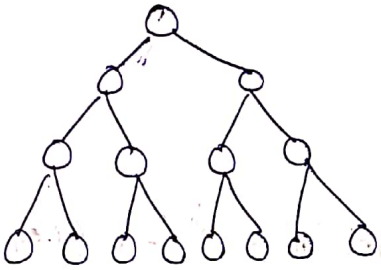
from case (ii) & case (iii),

$$n(h) = \underset{\text{root}}{1} + n(h-1) + n(h-2)$$

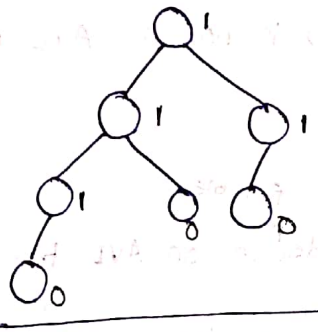
$$\therefore n(h) = \begin{cases} 1 + n(h-1) + n(h-2) & \text{if } h \geq 2 \\ 1 & \text{if } h = 0 \\ 2 & \text{if } h = 1 \end{cases}$$

For height $h=3$

Max no of nodes = 15



Min no of nodes = 7



In the case of min no of nodes balancing factor of every internal node is either 1 (or) -1.

(Q19) Find maximum and minimum no of nodes possible for an AVL tree of height 5.

Sol:

Max no of nodes = $2^{5+1} - 1 = 63$

Min no of nodes:

height	minimum
0	1
1	2
2	4
3	7
4	12
5	20

$n(h) = 1 + n(h-1) + n(h-2)$

$n(2) = 1 + n(1) + n(0)$

$= 1 + 2 + 1$

$= 4$

(Q20) what is max height of any AVL tree with 7 nodes (height of leaf nodes is 0)

Sol:

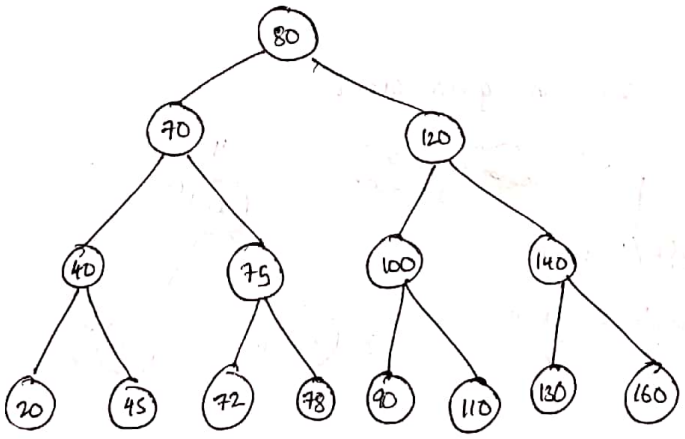
Min no of nodes possible for an AVL tree

height	min no of nodes
0	1
1	2
2	4
3	7 ← ∴ Max possible height = 3
4	12

(Q21) In a balanced BST with n elements, what is the worst case time complexity of reporting all elements in range [a,b]? Assume that the no of elements is k?

a) $\theta(\log n)$ b) $\theta(\log n + k)$ c) $\theta(k \log n)$ d) $\theta(n \log k)$

Sol:



Max height of BST with n elements $= O(\log_2 n)$

Let a = 20 b = 78

So the interval is [20,78]

Step 1:

to report the elements, we first need to search 20

This searching takes $O(\log_2 n)$

Silly we search b=78 in $O(\log_2 n)$

So to construct [20, 78] @ time = $O(\log_2 n)$

Now we need to report all the elements in [20, 78]

no of elements to be reported, k = 7

now we apply inorder traversal on elements [20, 78]

[a, b] has k elements

∴ Inorder traversal on k elements takes $O(k)$

∴ total time complexity

$= O(\log_2 n + k)$

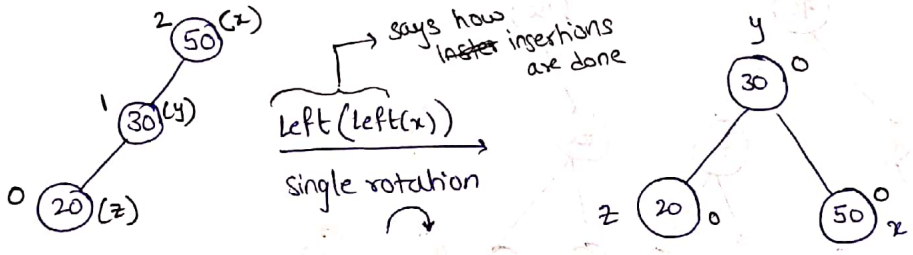
∴ opt (b)

16/09/20

Construction of AVL tree by inserting keys in the given order:

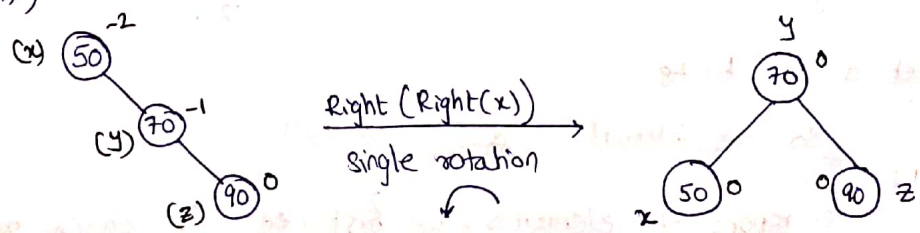
Case (i) (LL(x))

Insert 50, 30, 20 in given order.



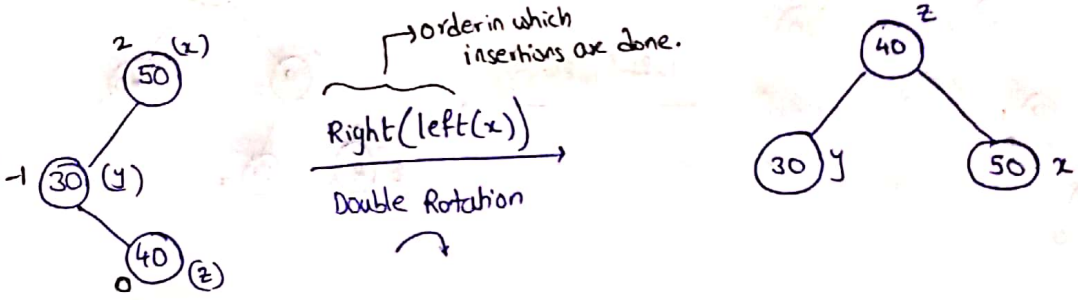
Case (ii): Insert 50, 70, 90 in given order

(RR(x))



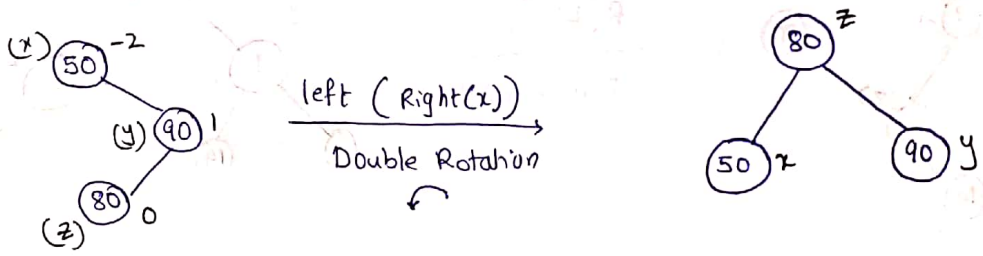
Case (iii) : (R-L(x))

Insert 50, 30, 40



Case(iv) (L-R(x))

Insert 50, 90, 80



P/29

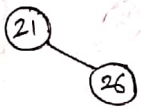
Insertion :

21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7

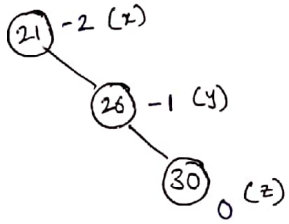
Insertion :

21 :

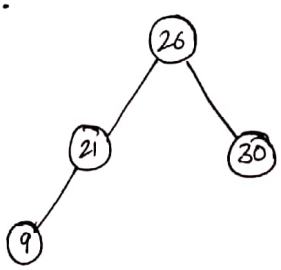
26 :



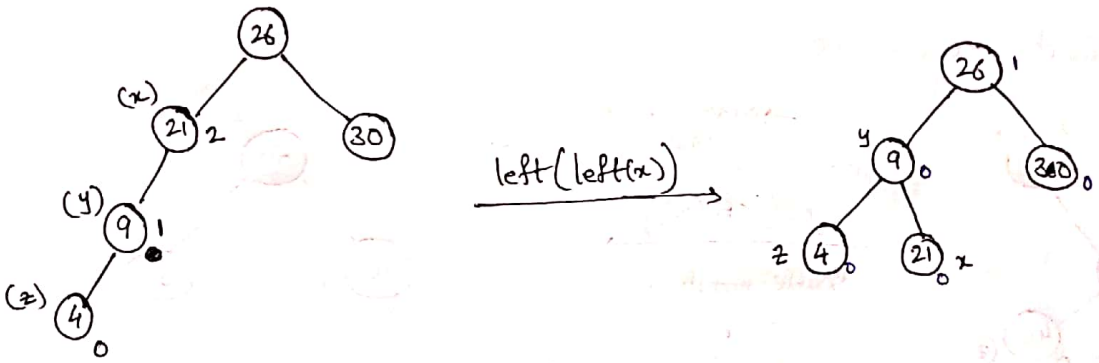
30 :



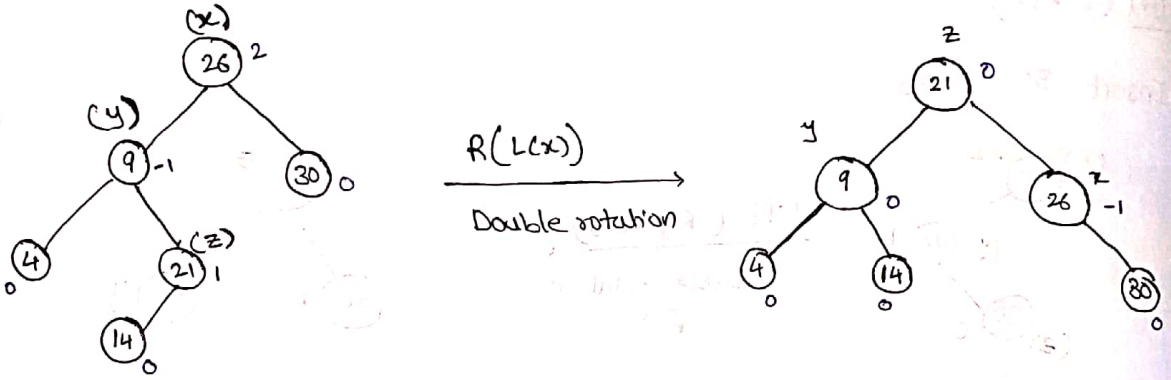
9 :



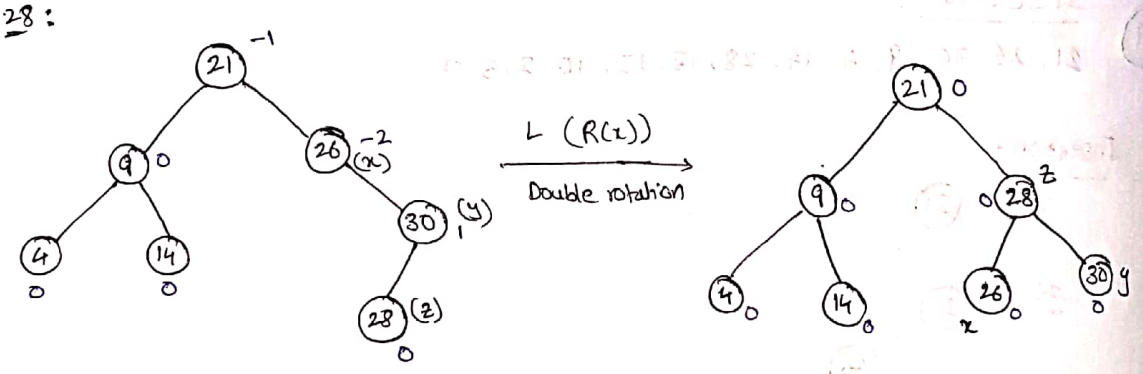
4:



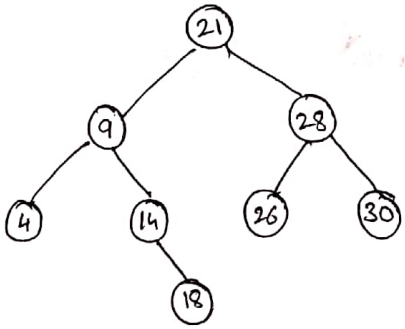
14:



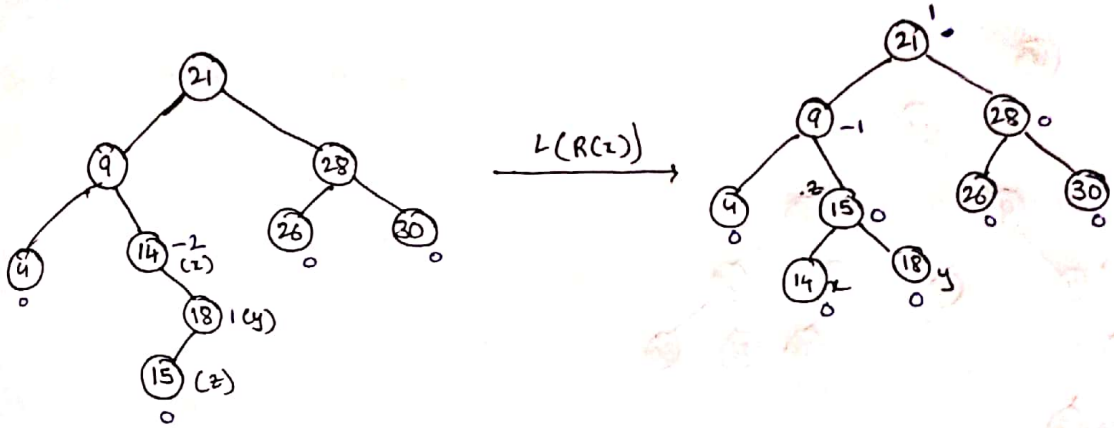
28:



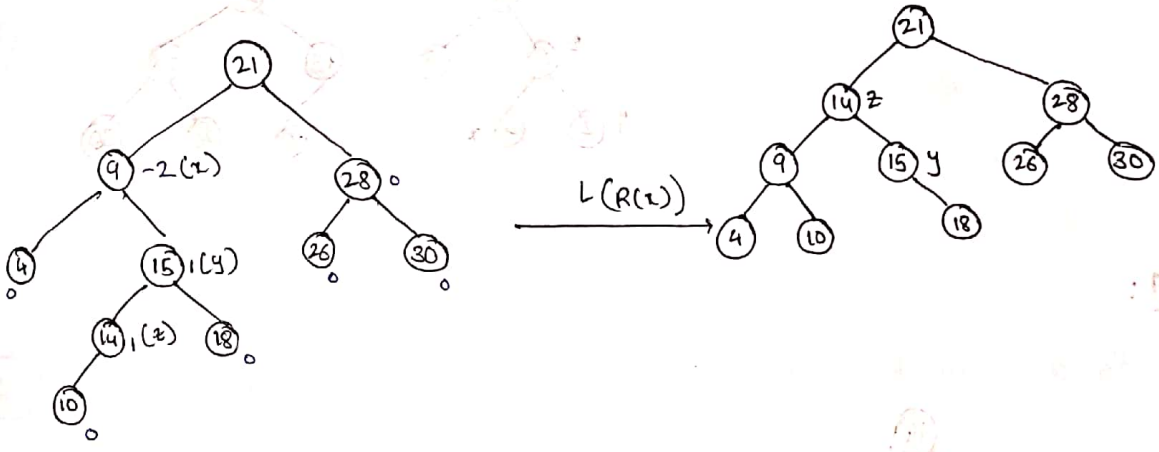
18:



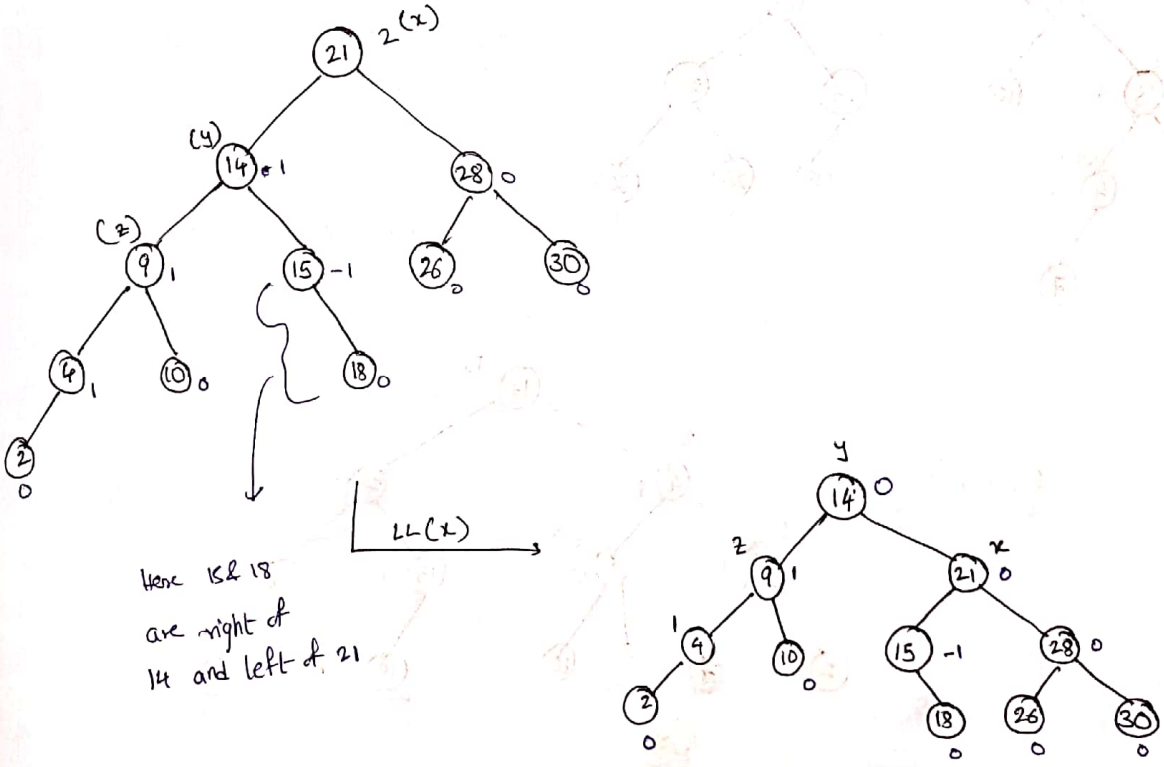
15 :



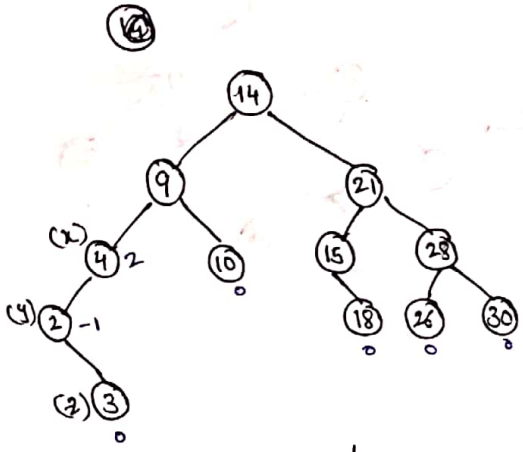
10 :



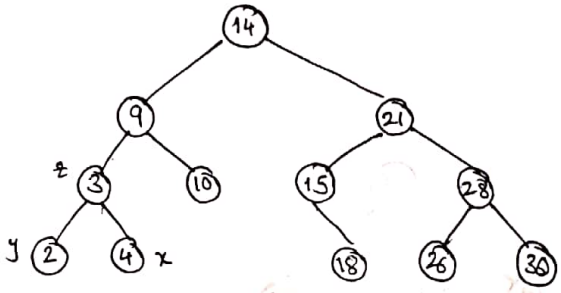
2 :



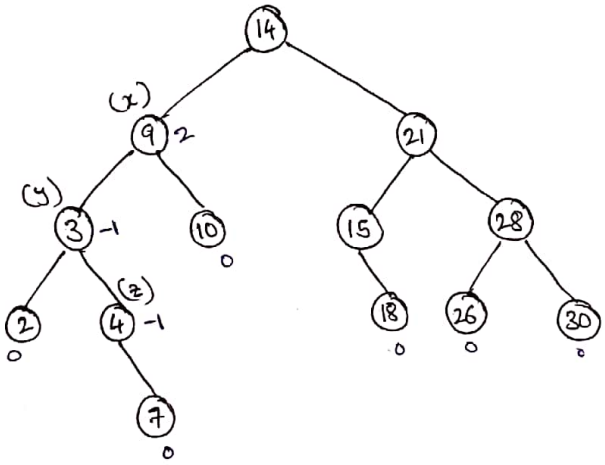
3 :



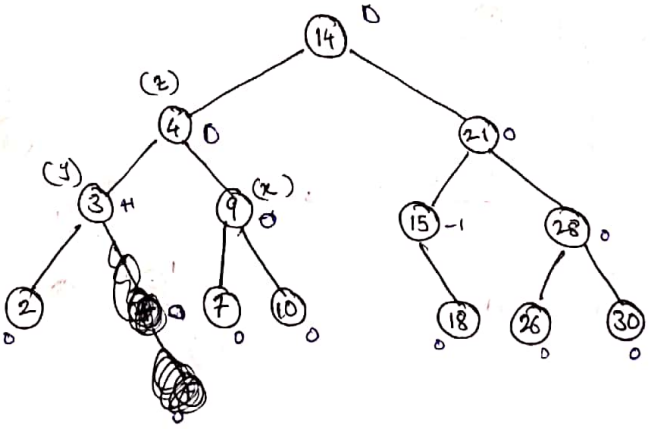
R(L(x))



7 :



R(L(x))



Analysis:

→ No of comparisions performed in worst case for insertion

= height of AVL tree + 1

$= \log_2 n + 1$

$= O(\log n)$

→ total time complexity for constructing AVL with n elements is

$\log 1 + \log 2 + \log 3 + \cdots + \log n$

$= \log (1 \cdot 2 \cdot 3 \cdots n)$

$= \log n!$ $\qquad n! \leq n^n$

$= O(\log n^n)$

$= O(n \log n)$

(Q22) what is the ~~worst~~ worst case time complexity of inserting n^2 elements into an AVL-tree with n elements initially?

a) $\theta(n^2)$ b) $\theta(n^2 \log n)$ c) $\theta(n^4)$ d) $\theta(n^3)$

Sol:

inserting one element .... $O(\log n)$

inserting n^2 elements ... $O(n^2 \log n)$

since we are talking about worst case we consider

Method 2:

$\log n + \log (n+1) + \log (n+2) + \cdots \log (n+n^2)$

$= \log (n \cdot (n+1)(n+2) \cdots (n+n^2)) = \log \left(\frac{1 \cdot 2 \cdot 3 \cdots (n+n^2)}{1 \cdot 2 \cdot 3 \cdots (n-1)} \right)$

$= \log (n+n^2)! - \log (n-1)!$

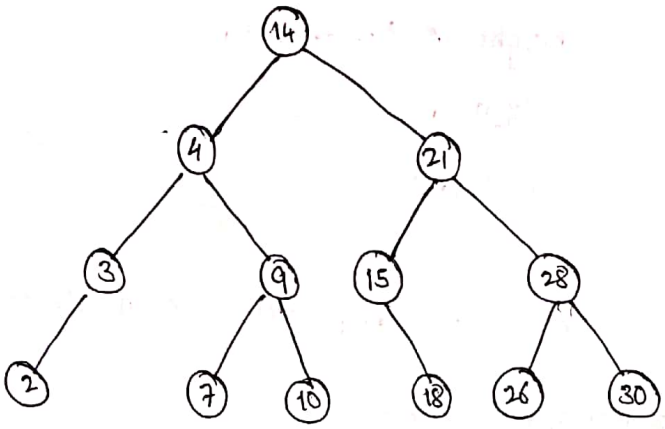
$= O(\log (n+n^2)!) = O(\log (n+n^2)^{(n+n^2)}) = O(n^2 \log n)$

17/04/20

P130

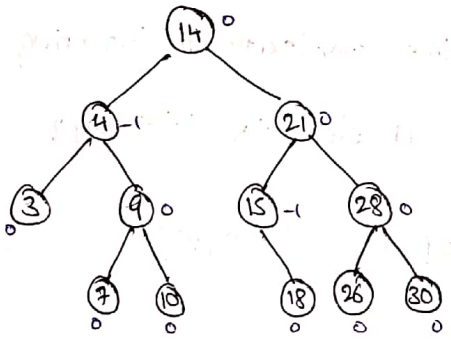
Deletion on AVL Tree

Delete 2, 3, 10, 18, 4, 9, 14, 7, 15



Here we perform deletion in the same way as we did for BST. But after every deletion we check whether AVL tree property is satisfied or not.

2: Deleting leaf node:



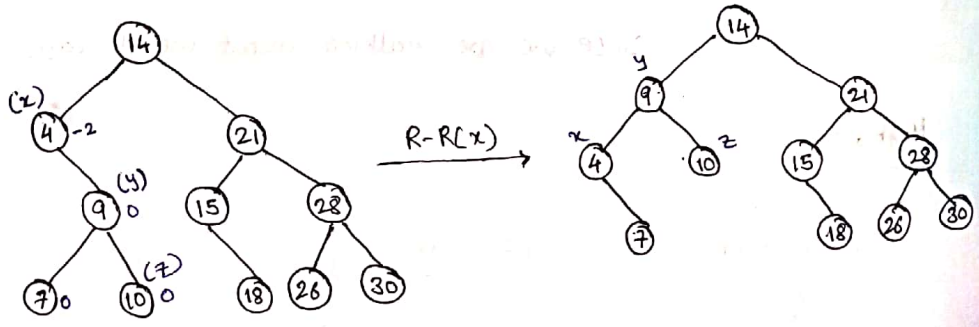
Step 1:

→ Search the node ---- O(log n)

→ Identify the deleted nodes parent pointer and assign it to NULL.

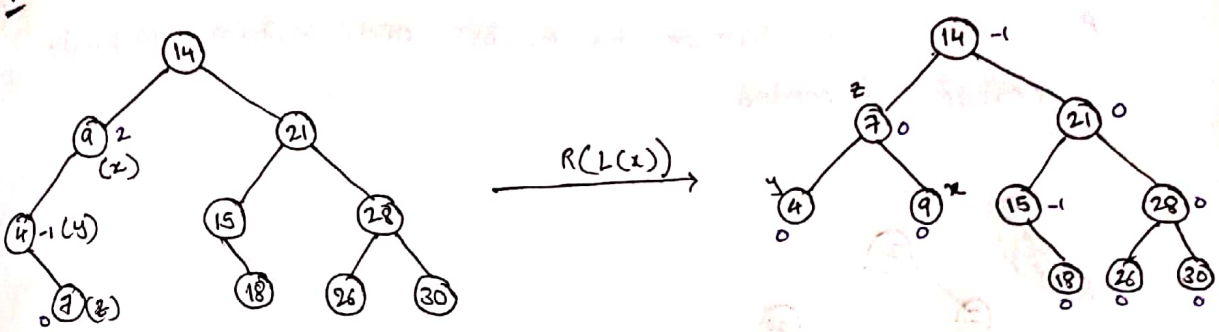
→ Now if the AVL property is satisfied then stop the process.
Otherwise perform necessary rotations to convert it into AVL tree.

3:

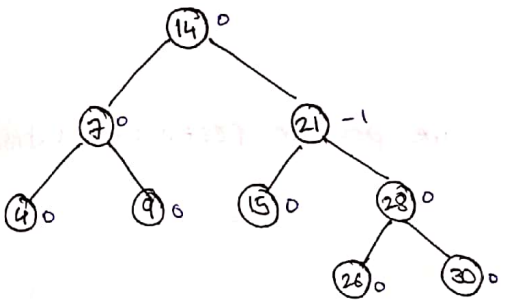


Here we can choose either 7 or 10 as z.

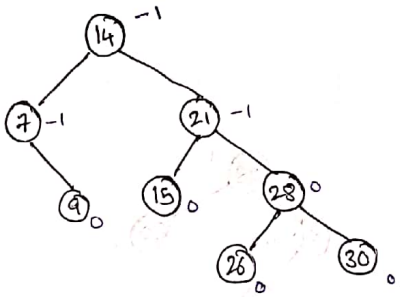
10:



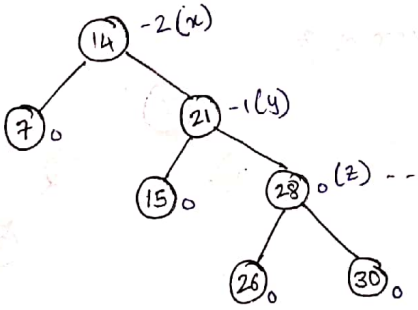
18:



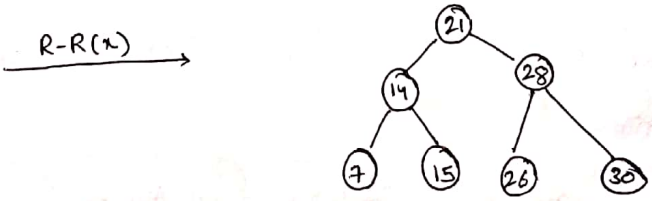
~~4~~ 4:



9:

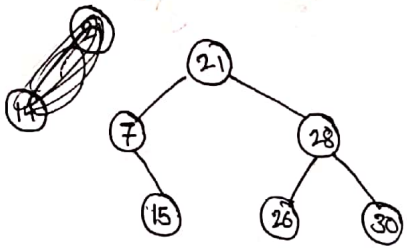


This is choosen as z cuz we have longest path in this direction



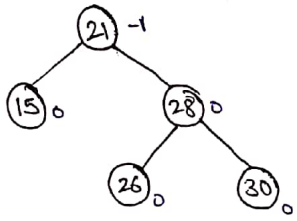
14: Deleting node with 2 children

A we delete same like we did for BST and perform appropriate rotations if needed.

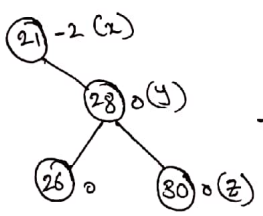


7: (Deleting node with 1 child)

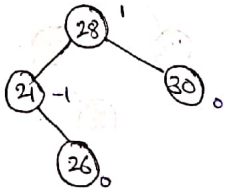
Same as we did for BST and we perform necessary rotations.



15:



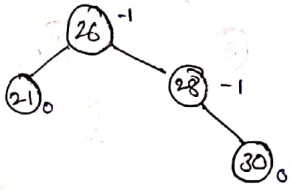
R-R(x)



Here we can choose either 26 (or) 30 as z.

(or)

if 26 is choosen as z



Analysis:

Deleting a node is done by search it first and delete and performing necessary rotations.

∴ time complexity for deletion from AVL tree = $O(\log n)$

Deleting n nodes = $\log n + \log(n-1) + \cdots + \log 2 + \log 1 = n \cdot O(n \log n)$

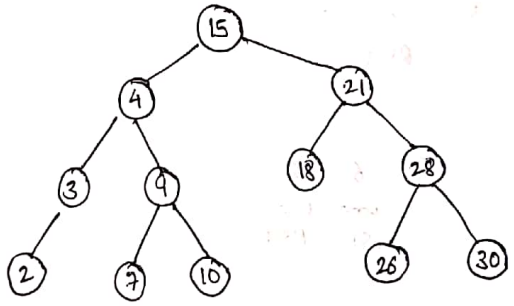
P/28

height	min no of nodes
0	1
1	2
2	4
3	7
4	12
5	20
6	33
7	54
8	88

P/27 Deleting 14

we need replace with successor 15

and 15 is replaced with its successor i.e., 18



∴ root node : 15

P/26 In a BST for every node, nodes in LST are lesser and nodes is RST are greater

a) 5 3 1 2 4 7 8 6

↑

5 is root

∴ next 4 elements must be among 1 to 4

and next 3 elements should be 6 to 7

3 (1 2 4) (7) 8 6 ✓

↑ ✓ ↓

left subtree have 8

∴ ✗

b) 5 3 1 2 6 4 8 7

LST of node 5 — 6 cannot be in the LST of 5

$\therefore \alpha$

c) 5 3 2 4 1 6 7 8

LST RST

3 2 4 1

LST of 3 RST of 3

$\therefore \alpha$

d) 5 3 1 2 4 7 6 8

LST RST

3 1 2 4 — LST RST

7 6 8 — LST RST

1 2 — RST

$\therefore$ opt (d)

P/25 RC : Right child

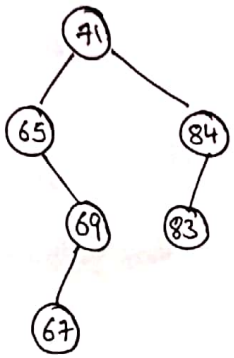
```
if(!t->RC)
```

↳ if we have right child then condition fails else we return data

So this program keeps traversing towards right and finally prints when there is no right child.

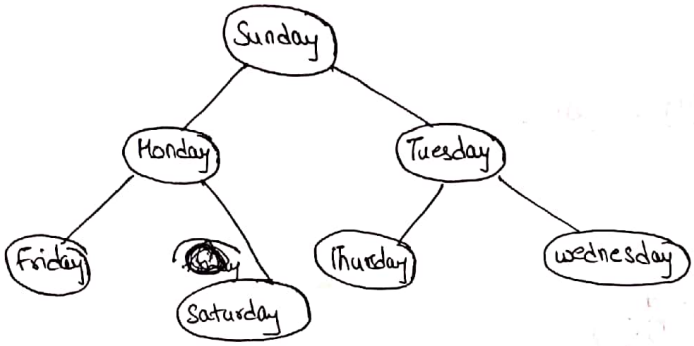
$\therefore$ 30

P/24



∴ 67

P/23

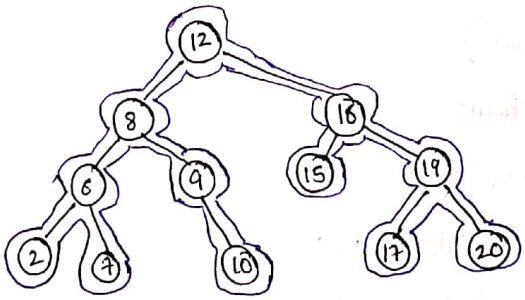


∴ 4 leaf nodes

P/22

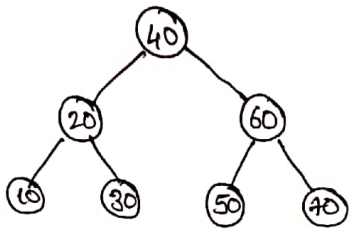
2 6 7 8 9 10 12 15 16 17 19 20

12 8 6 2 7 9 10 16 15 19 17 20



Postorder: 2 7 6 10 9 8 15 17 20 19 16 12

P/21



∴ 4

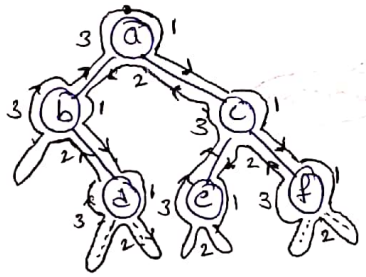
19/09/20

~~P/20~~

P/01 left, right, root

∴ post order

Converse of ~~postorder~~ traversels :



Converse of post order : f e c d b a (3's)

converse of preorder : a c f e b d (1's)

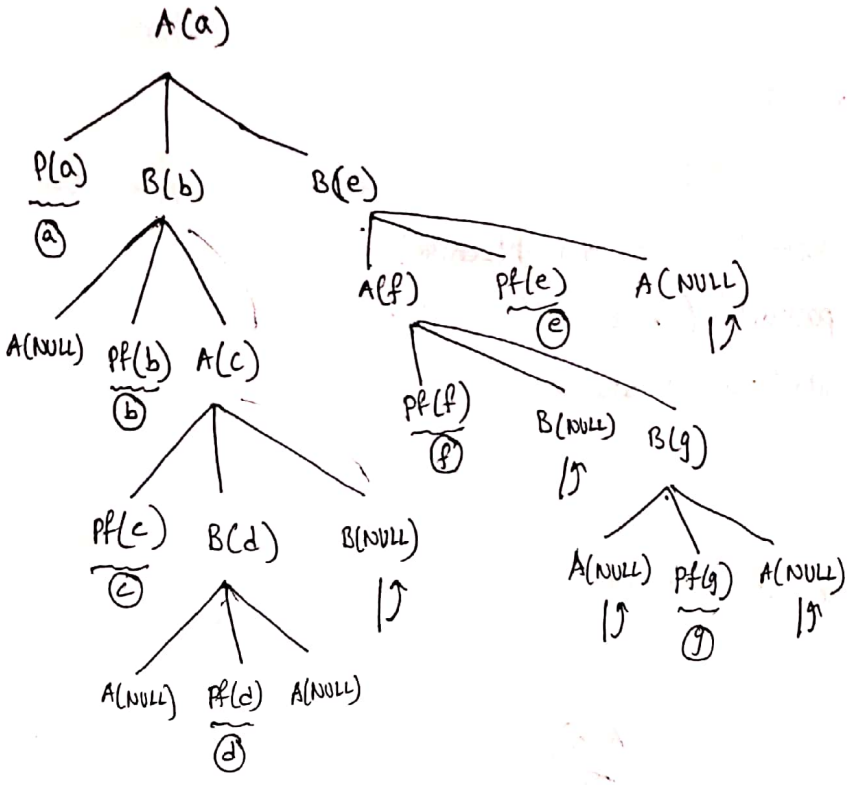
converse of inorder : f c e a d b (2's)

```
→ C_preorder(struct node* root)
  {     if(!root) return;
        printf(root→data);
        C_preorder(root→right_child);
        C_preorder(root→left_child);
  }
→ C_inorder(struct node* root)
  {     if(!root) return;
        C_inorder(root→right_child);
        printf(root→data);
        C_inorder(root→left_child);
  }
```

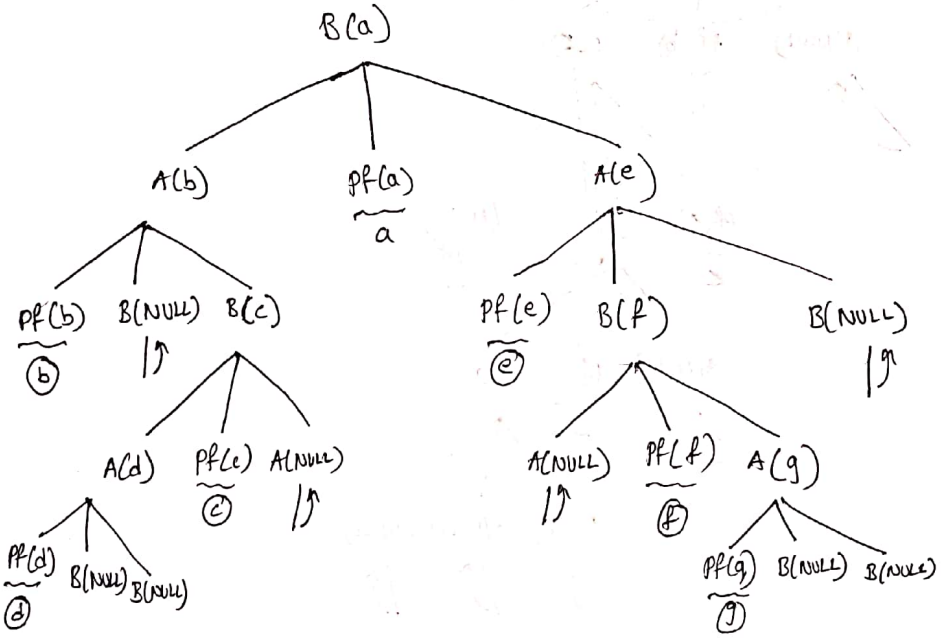
→ c_postorder (struct node* root)

```
{
    if (!root)
        return;
    c_postorder( root → right_child);
    c_postorder ( root → left_child);
    printf( root → data);
}
```

P|3



∴ A(a) : a b c d f g e



∴ B(a) : b d c a e f g

(P/4) B(t) : A(e) a A(b)

A(e) : e B(f)
e A(g) f
e g f

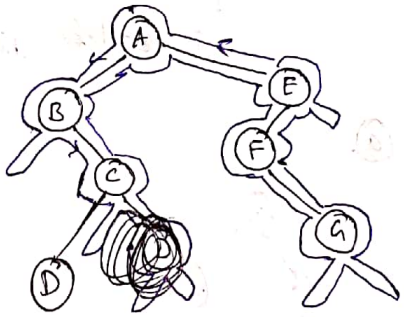
A(b) : b B(c)
b c A(d)
b c d

∴ B(t) : e g f a b c d

(P/5) $\frac{1}{n+1} {}^{2n}C_n = \frac{1}{4} {}^{6}C_3 = \frac{1}{4} \frac{6 \times 5 \times 4}{3 \times 2 \times 1} = 5$

(P/6) Inorder : B D C A F G E

preorder : A B C D E F G



postorder : D C B G F E A

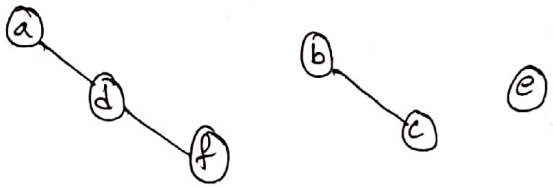
Note :

→ For a given preorder + right child pointer & terminal nodes we can construct a unique binary tree.

→ Silly postorder + left child pointer + terminal nodes ⇒ unique binary tree

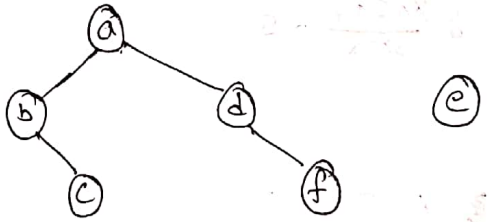
P17 TAG = 1 => leaf nodes

we have following parts of tree



If b c is present in right subtree then preorder sequence cannot start with a, b.

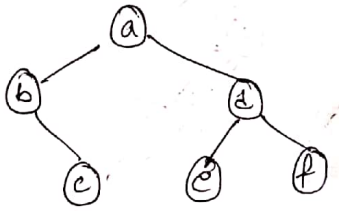
∴



∵ preorder : a b c d e f

e is left child of d

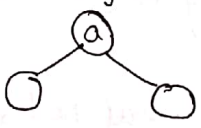
∴



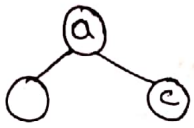
∴ no of nodes in RST = 3 (d, e, f)

P18 post order : d e b f g c a
degree : 0 0 2 0 0 2 2

a is root & deg of a is 2



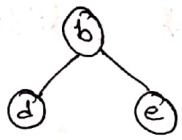
a has right child & preceding node of a in postorder is c. ∴ Right child of a is c.



the starting 3 nodes of post order are d, e, b

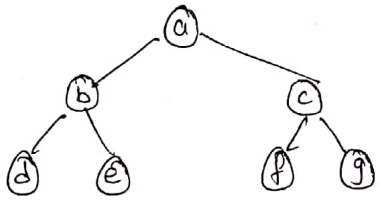
d must be left most node of tree.

∵ e is leaf & b has degree 2 the possible structure is



silly for f, g, c we have

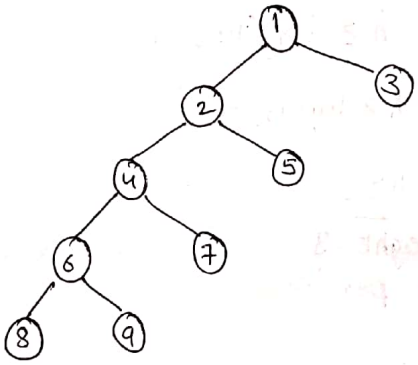
∴ the binary tree is



∴ no of nodes in the LST = 3

(P/9) In. : 8 6 9 4 7 2 5 1 3

post : 8 9 6 7 4 5 2 3 1 ↑root



∴ height = 4

P/10 a) total nodes = internal + leaves

~~2*i+1 = i+20~~ (i → internal nodes)

~~2i = i+19~~

Let i_1 be no of nodes with 1 child

i_2 be no of nodes with 2 children

$i_1 + 2*i_2 + 1 = i_1 + i_2 + 20$

$i_2 = 19$

∴ no of nodes having 2 children = 19

b) $i_1 + 2i_2 + 1 = i_1 + 200$

$i_2 = 199$

P/11 Min height ⇒ complete binary tree

If n is no of nodes in CBT

$2^h \le n \le 2^{h+1} - 1$

$2^h \le n$ $\quad$ $n+1 \le 2^{h+1}$

$h \le \log_2 n$ ⇒ $h+1 \ge \log_2(n+1)$

$h \ge \log_2(n+1) - 1$

$h \ge \log_2(16) - 1$

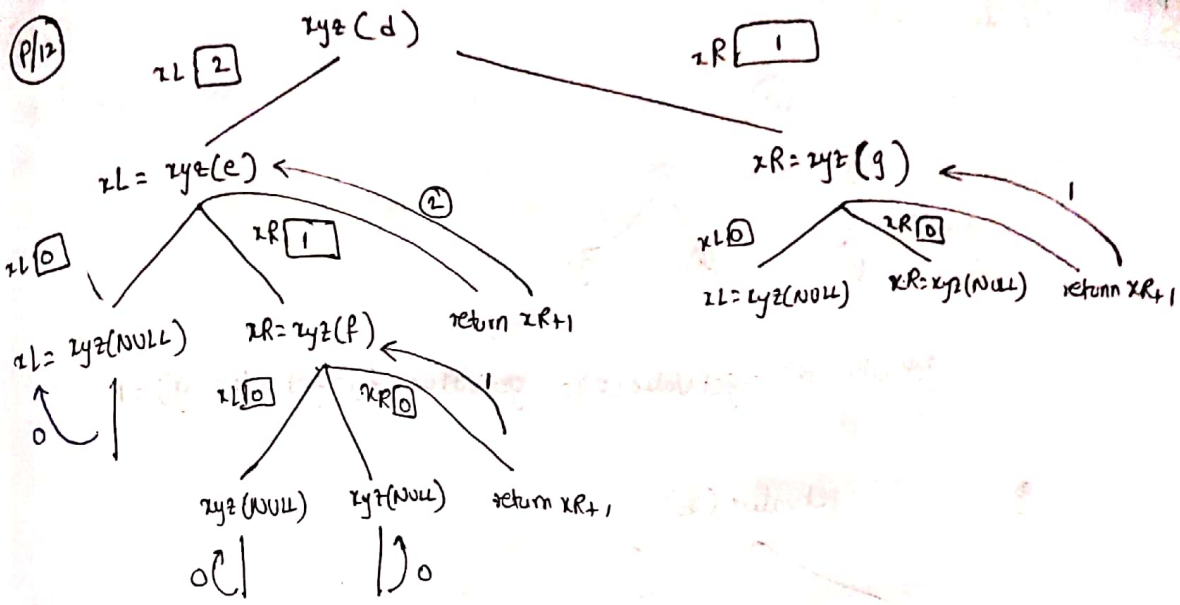
$h \ge 3$

min height = 3

Max height ---- one node per level

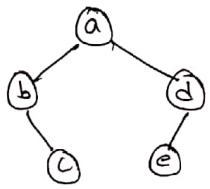
⇒ max height = 14

(P/12)



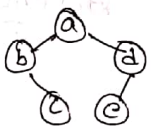
$\therefore xL = 2 \quad xR = 1$

(P/13) From the code, swapping will be done on nodes in the order of postorder.

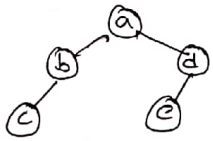


Postorder: c b e d a

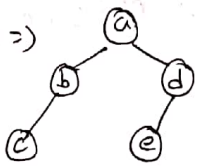
Swap on c =>



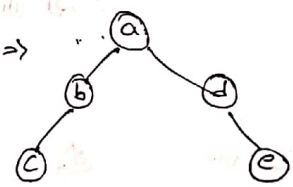
Swap on b =>



Swap on e =>

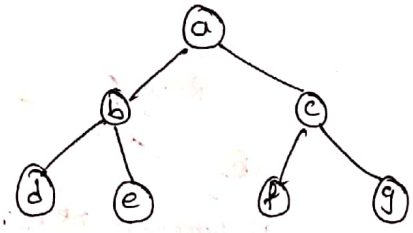


Swap on d =>



This is the tree before last swap.

P/14 Consider below tree



getvalue(d) = getvalue(e) = getvalue(f) = getvalue(g) = 1

getvalue(b)

value = value + getval(d) + getval(e)

return value;
2

1 1

∴ value = 2

∴ get value (b) = 2

Silly get value (c) = 2

Now getval(a)

value = value + getval(b) + getval(c)

return value;
4

2 1 1

⇒ value = 4

∴ getvalue = 4 = no of leaf nodes = no of nodes without right sibling

~~∴ opt (d)~~

Consider

getval(b) = 1

getval(a) = val + getval(NULL) + getval(b) = 0 + 0 + 1 = 1

= ~~no~~ no of leaf node

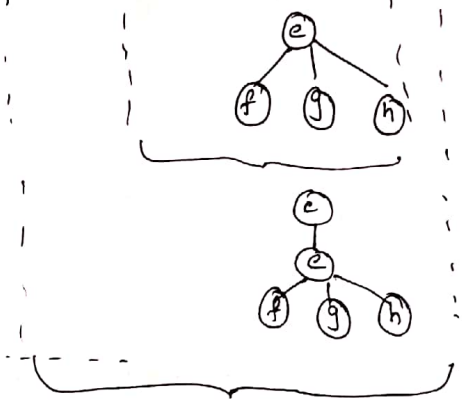
≠ no of nodes without right sibling

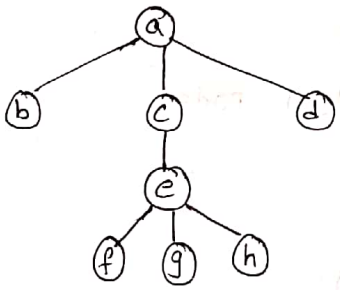
∴ opt (d)

P/15

nodes outside the paranthesis is considered parent for nodes in the the paranthesis

a(b, c(e(f, g, h))d)



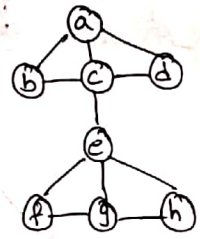


∴ Height = 3

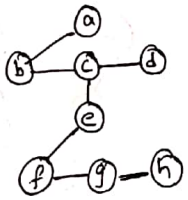
Leftmost child right sibling representation

Converting into binary tree

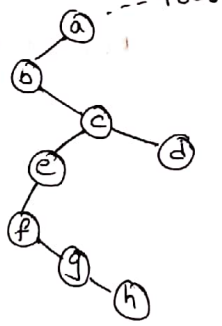
Connect every child to its right sibling



keep only leftmost link



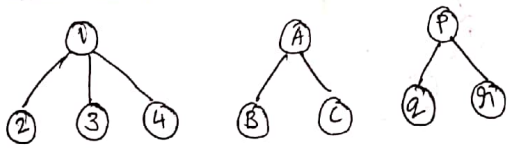
This is redrawn as



∴ Height = 6

P/16

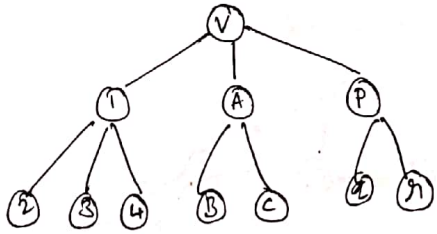
Converting forest into binary tree:

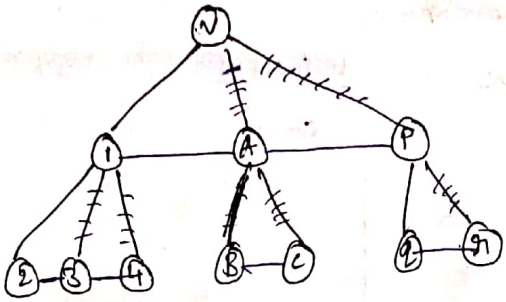


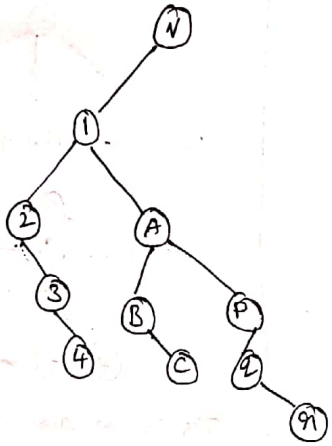
Rule for converting general tree into binary tree

→ Using leftmost child right sibling representation we convert into binary tree.

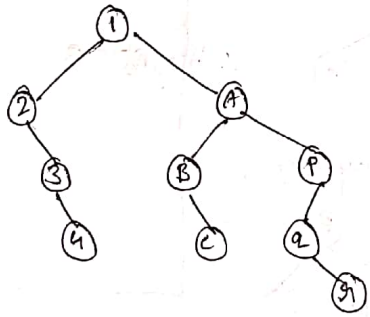
Let 'v' be a virtual node





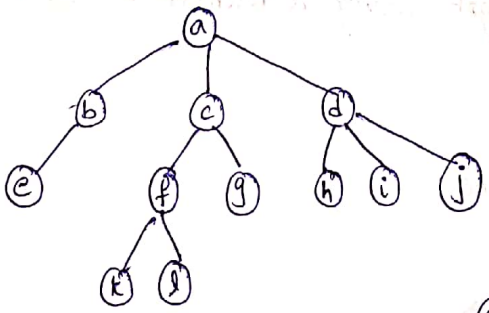


Now we remove the virtual node

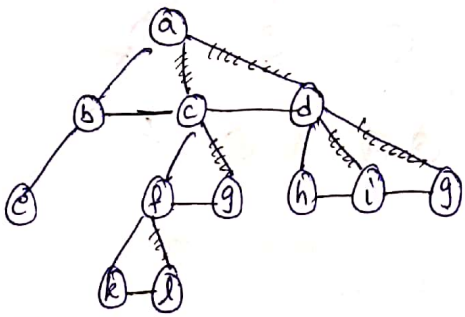


Height = 4

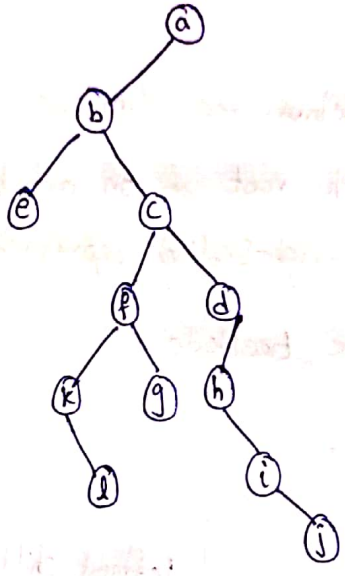
Eg: Convert below tree into binary tree



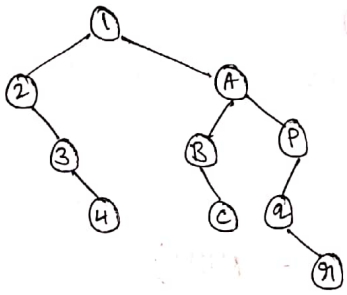
=)

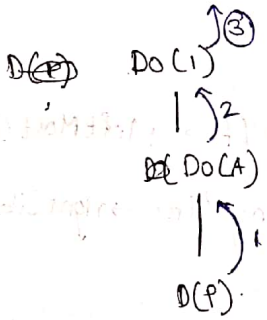


->



P|17 Consider converted binary tree in P|16





Here '3' is returned and 3 is no of ~~trees~~ trees in the forest.

Also consider the tree in P|15

the root of converted binary tree has no right child

∴ the value returned = 1

i.e, no of trees in the forest.

∴ opt (d)

20/09/20

Q23 Consider the pseudocode given below. The function DoSomething() takes as argument a pointer to root of an arbitrary tree represented by the leftMostChild-rightSibling representation. Each node of the tree is of type treeNode.

```
typedef struct treeNode* treeptr;
struct treenode
{
    treeptr leftMostChild, rightSibling;
};
int DoSomething (treeptr tree)
{
    int value=0;
    if (tree != NULL)
    {
        if (tree -> leftMostChild == NULL)
            value=1;
        else
            value = DoSomething (tree -> leftMostChild);
        value= value + DoSomething (tree -> rightSibling);
    }
    return(value);
}
```

leftmost child - right sibling representation.

when the pointer to the root of tree is passed as the argument to DoSomething the value returned by the function corresponds to

a) no of internal nodes in the tree.
b) height of the tree
c) no of nodes without right sibling in the tree.
d) no of leaf nodes in the tree.

Sol :

Consider tree (a)---to (right sibling is NULL)

DoS(a) returns 1

Here no of internal nodes = 0

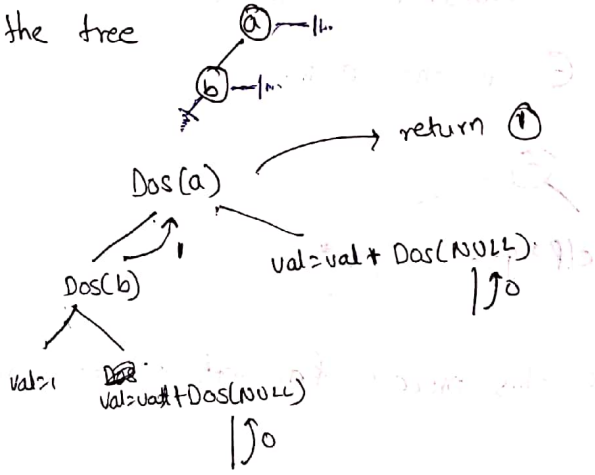
height = 0

nodes without right sibling = 1

no of leaf nodes = 1

∴ eliminate opt (c) (a) & (b)

Consider the tree



∴ no of nodes without right sibling = 2

no of leaf nodes = 1

∴ opt (d)

P/18 a = -b + c * d/e + f ↑ g ↑ h + i * j

Precedence : ↑ > (-) > (*=/) > + > =
(unary minus)

↑, (-) unary — associativity is right to left

+, * - - - - - left to right

Expression tree :

(i) In expression tree, operators are parents
operands are children

(ii) Least precedence operator is the root node.

Eg: Consider expression

a * b/c + e/f * g + k - x * y

Step 1: Scan the infix expression from left to right

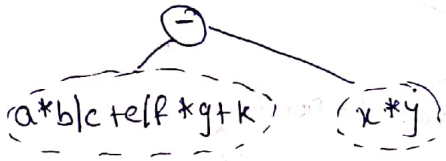
+ & - have least precedence

the root node will be evaluate in the end

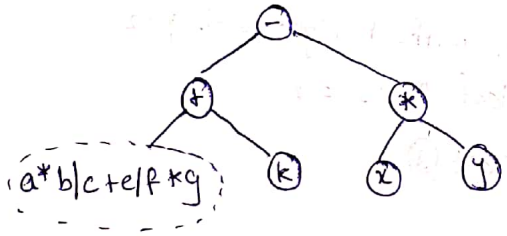
associativity of + & - is left to right.

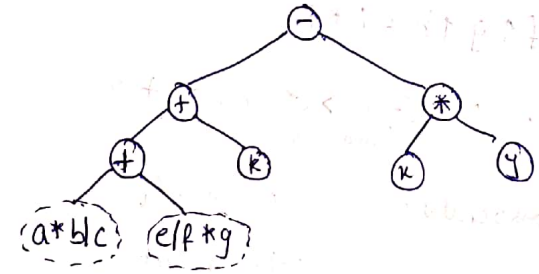
It means rightmost of + or - must be the root. since it will be evaluated in the end.

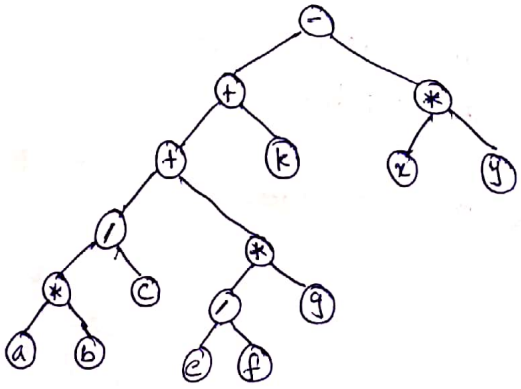
∴ ⊖ is the root.



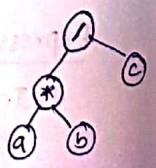
Now we continue this process for left subtree & rightsubtree recursively.



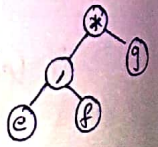




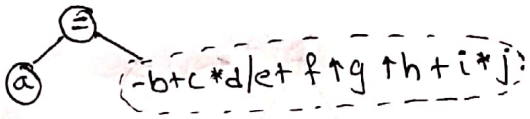
a*b/c =

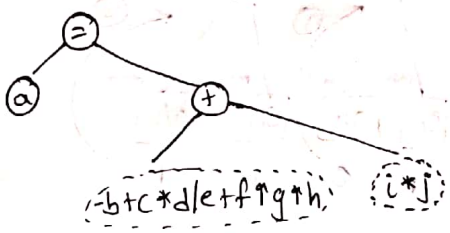


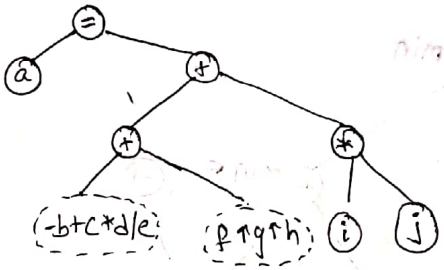
e/f*g =

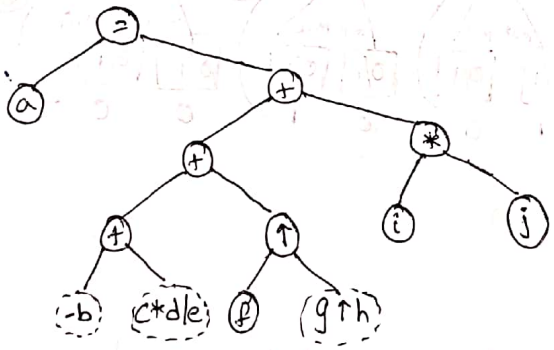


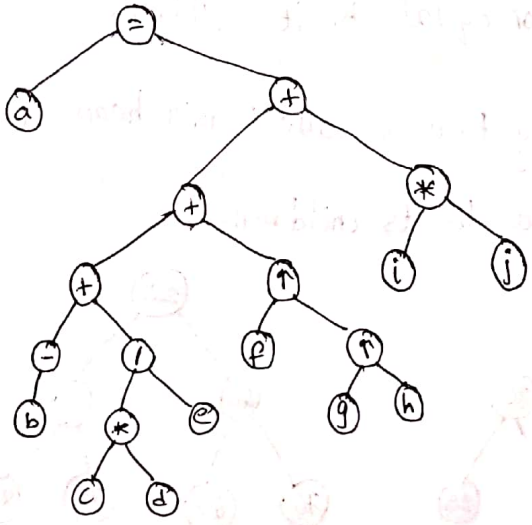
a = -b + c * d / e + f ↑ g ↑ h + i * j



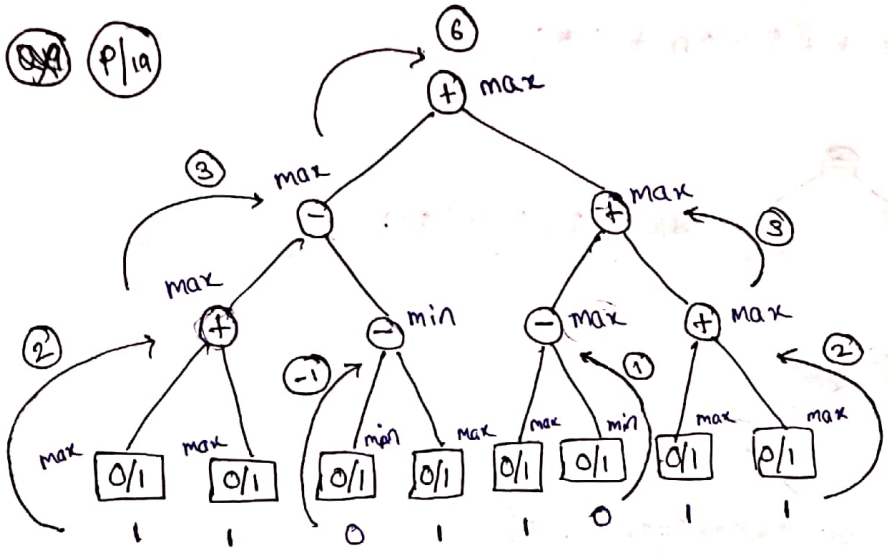


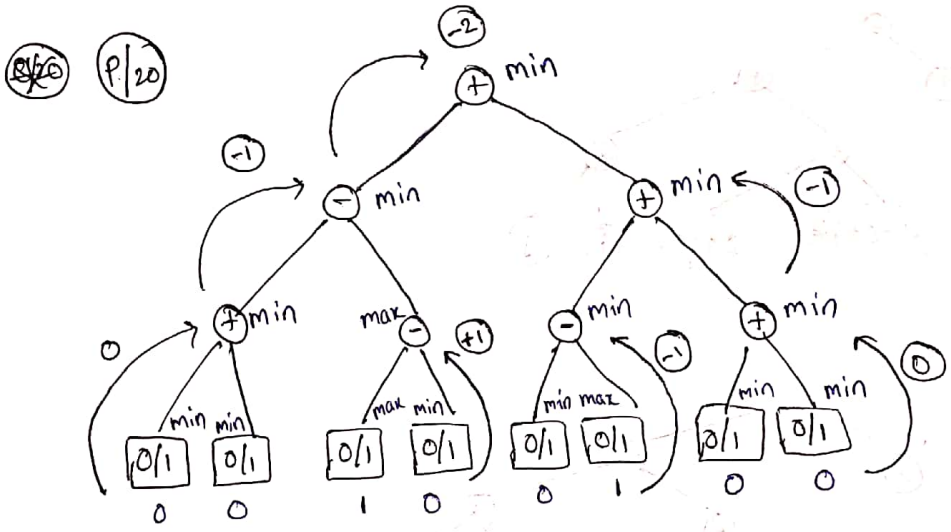






∴ root node is =



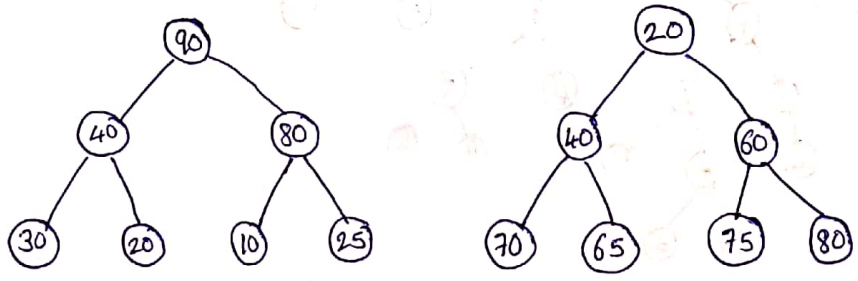


** V.Imp

* Heap Tree:

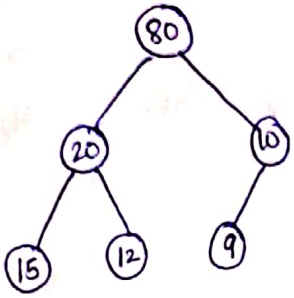
→ A complete binary tree is called max heap, if parent value is greater than or equal to it children.

→ A complete binary tree is called min heap if parent value is less than or equal to its children.



Max Heap

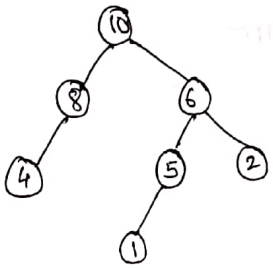
Min Heap



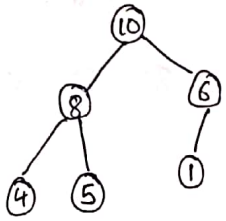
Max Heap

(Q24) A max heap is a heap where the value of each parent is greater than or equal to the value of its children. Which of the following is a max heap?

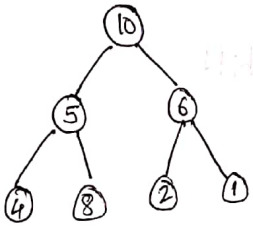
a)



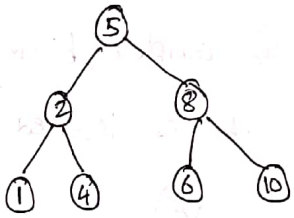
b)



c)



d)



Sol:

opt (a) is not complete binary tree.

∴ not max heap

opt (b) is true.

Construction of heap tree:

Method 1:

(i) Construct heap tree by inserting keys one after another in the given order ..... $O(n \log n)$

Method 2:

Convert the given array into heap (Build heap or heapify algorithm)

↳ $O(n)$

Eg 1: Construct heap tree (max heap) by inserting following keys in the given order. (insertion must be done one after other)

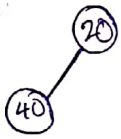
20, 40, 80, 60, 30, 75, 95, 110

20:

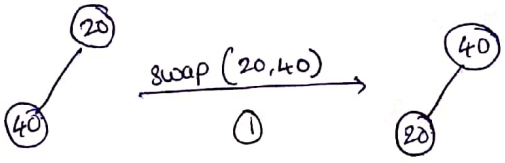
40:

Heap tree is complete binary tree

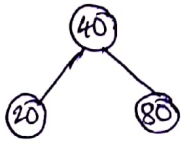
∴ 40 is inserted as left child



Here 40 > 20 and hence this doesn't satisfy heap tree property. So we need to perform an interchange.

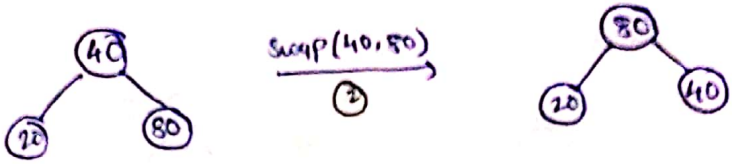


80:

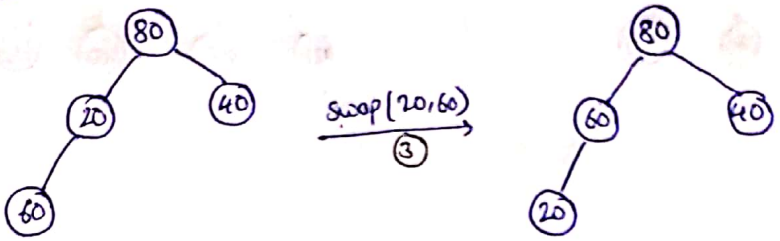


80 > 40

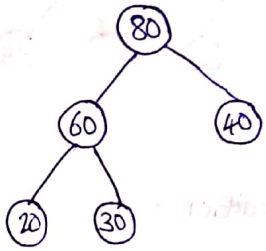
∴ Max heap property is not satisfied



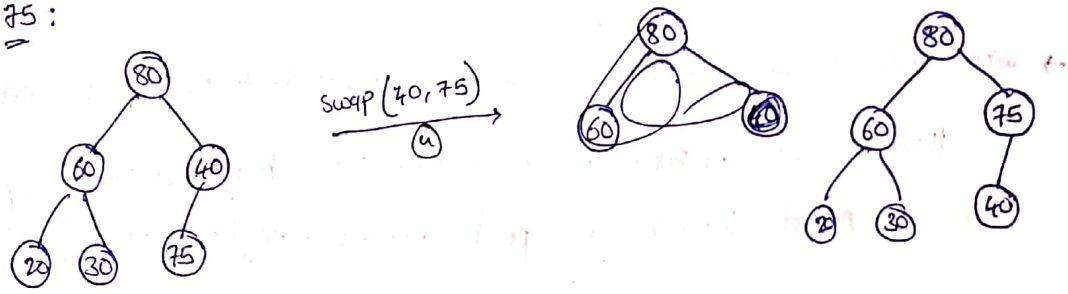
60:



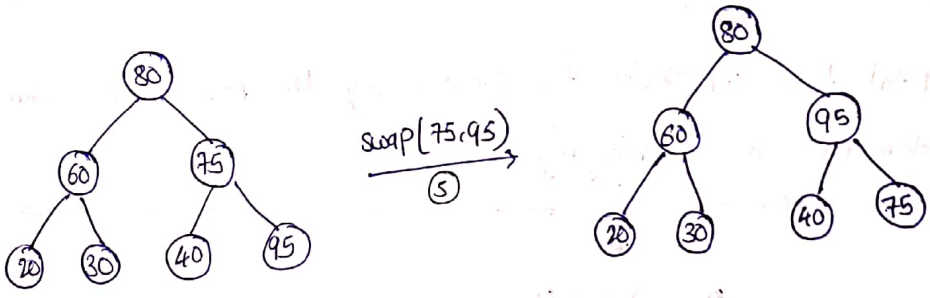
30:

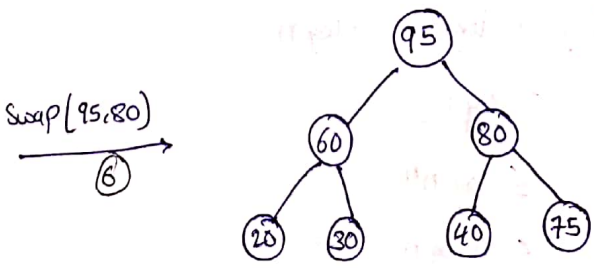


75:

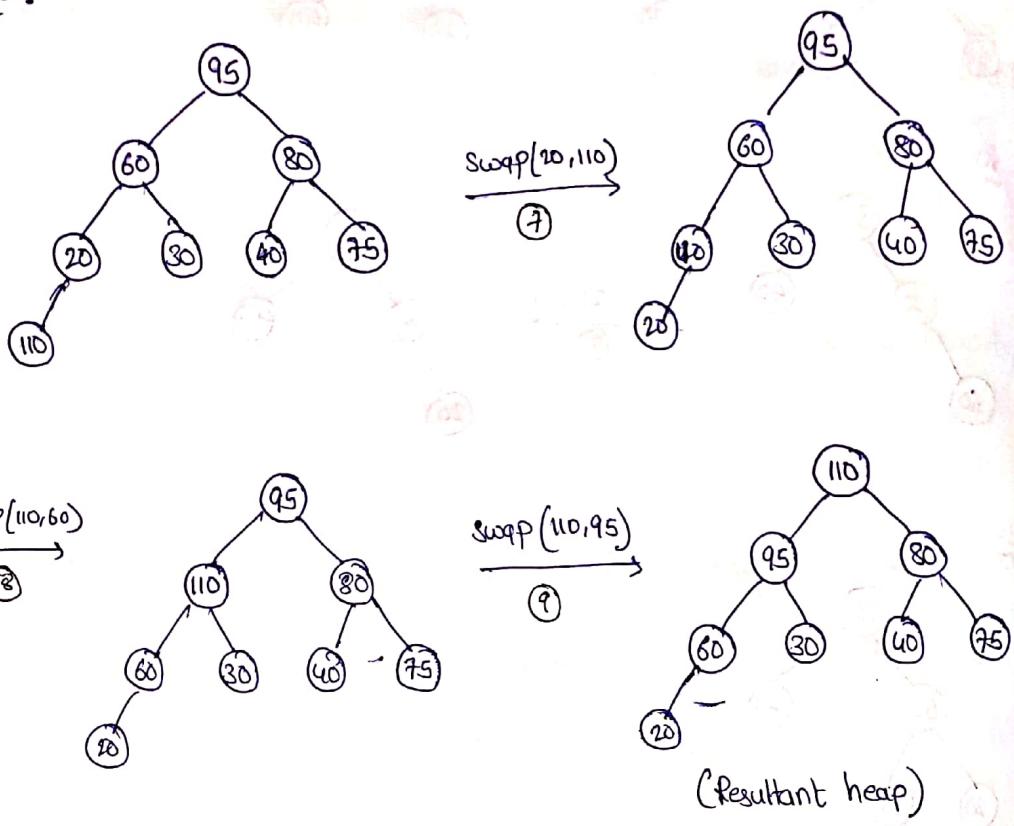


95:





110 :



(Resultant heap)

Here no of interchanges performed = 9

→ when 110 is inserted, it is swapped from leaf to root (worst case)

Here no of comparisions performed : 3 -- height of tree
no of interchanges performed : 3 -- height of tree
} $O(\log_2 n)$

∴ The time complexity for inserting a node into a heap with n node is $O(\log_2 n)$

∴ The total time complexity for constructing the heap tree with n elements is $O(n \log n)$

$\cancel{O(\log 1) + O(\log 2)} + \cdots$

$\log 1 + \log 2 + \cdots + \log n$

$= \log n!$

$\leq \log n^n$

$\leq n \log n$

$= O(n \log n)$

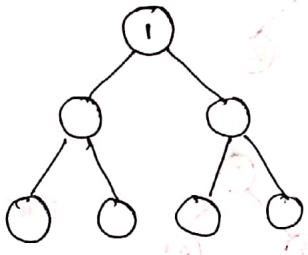
21/09/20

(025) How many no of min heaps are possible by using keys 1, 2, 3, 4, 5, 6, 7.

Sol:

Heap tree is complete binary tree

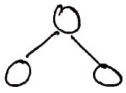
and since it is minheap the tree must be



LST has 3 locations

So we can choose 3 elements from 2, 3, 4, 5, 6, 7 to construct LST

i.e., $6C_3 = 20$ ways

After selecting 3 elements for LST 

the minimum of 3 has to be in the root

the other two elements into the leaf can be arranged in 2 ways.

i.e., 2 way for each combination of 3 elements.

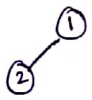
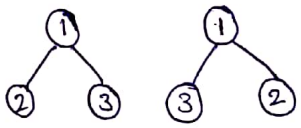
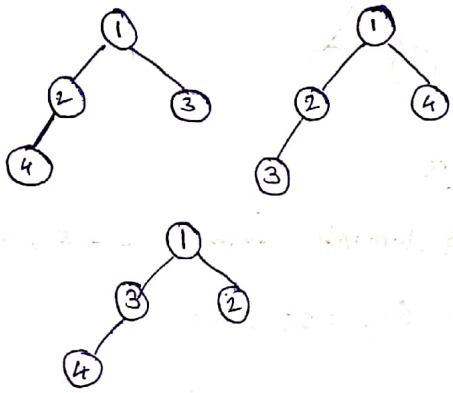
Sily the RST can also be constructed in 2ways.

∴ total no of min heaps possibles = $6C_3 \times 2 \times 2$

$= 20 \times 2 \times 2 = 80$ ways.

~~Finding no of min heaps possible with key 1, 2, 3, 4, 5, ... n~~

→ Finding no of min heaps possible with keys 1, 2, 3 ... n be T(n)

n	Possible minheaps	T(n)
1		T(1) = 1
2		T(2) = 1
3		T(3) = 2 = $^1C_1$ $^2C_1$ $^1C_1$ (root)
4		T(4) = $^1C_1 \cdot {}^3C_2 \cdot {}^1C_1 \cdot 1! \cdot {}^1C_1 = 3$ (root, LST, LST parent, arranging leaf nodes of LST, RST)

Now for n elements, '1' is placed at the root.

Now assume LST has k elements, then no of elements in RST will be n-k-1

$$T(n) = {}^1C_1 \cdot {}^{(n-1)}C_k \cdot T(k) \cdot T(n-k-1)$$

Array representation of heap tree:

i) Array index of root node is 1. (general convention but not compulsory)

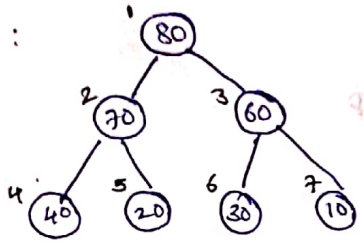
ii) For any node whose index is at 'i' then

index of leftchild ... 2i

index of rightchild ... 2i+1

index of parent .... $\lfloor i/2 \rfloor$

Ex:



1	2	3	4	5	6	7
80	70	60	40	20	30	10

(Q26) Consider a binary max heap implemented using an array. which of the following represents a binary max-heap?

a) 25 12 16 13 10 8 14

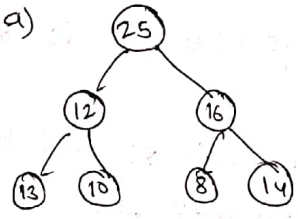
b) 25 12 16 13 10 8 14

c) 25 14 16 13 10 8 12

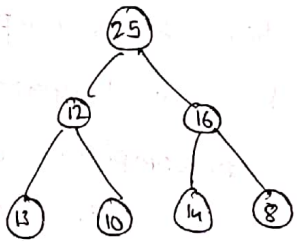
d) 25 14 12 13 10 8 16

Sol:

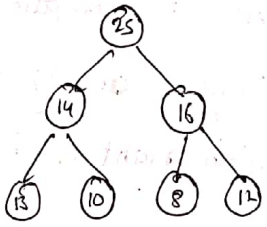
a)



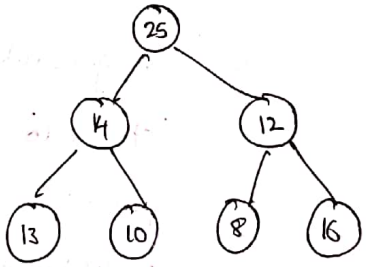
b)



c)



d)



∴ opt (c)

Method 2:

a) 25 12 16 13 10 8 14

b) 25 12 16 13 10 14 8

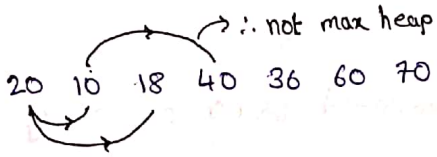
c) 25 14 16 13 10 8 12

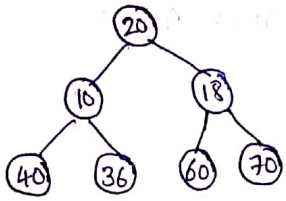
∴ opt (c)

Build Heap (Heapify) Algorithm:

Convert the following array into max heap

1	2	3	4	5	6	7
20	10	18	40	36	60	70





1. By default all leaf nodes satisfy the heap tree property.

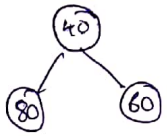
Eg: Consider 

Here 40 is compared with 60 to find largest(40,60).

Now the ~~largest of~~ largest(40,60) is 80 ~~compared~~ compared with 80.

i.e., 60 is compared with 80 and then 80 is swapped with 40.

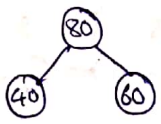
i.e., In general every element is compared with its left child and the largest among left child & parent is compared with right child and swap is performed.

Eg: Consider 

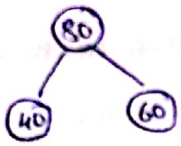
40 & 80 are compared ~~and~~ and 80 is largest

Now 80 is compared wit 60 and 80 is largest.

~~So 80 moves to the root~~ So 80 is swapped with root.

⇒ 

Eg:

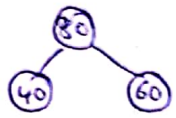


80 & 40 are compared ⟹ 80 is largest

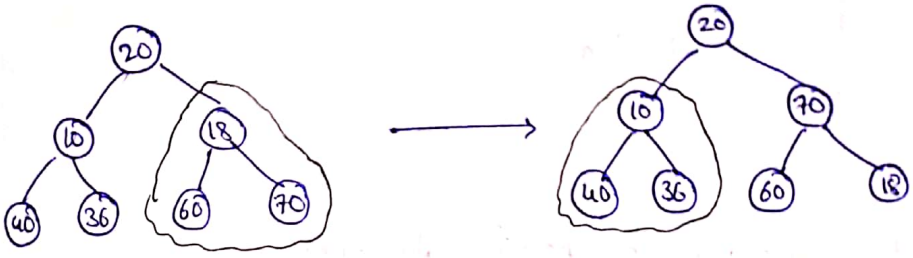
80 & 60 are compared ⇒ 80 is largest.

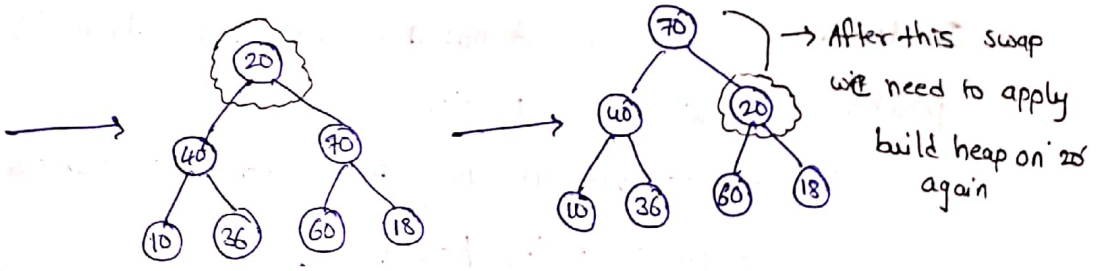
∴ no swapping is performed.

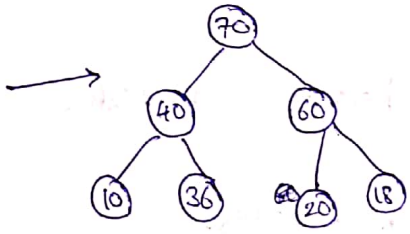
⇒



Now we apply build heap on the given problem







At every node we perform two comparisons

(i) parent & left child

(ii) largest (parent, left child) & right child

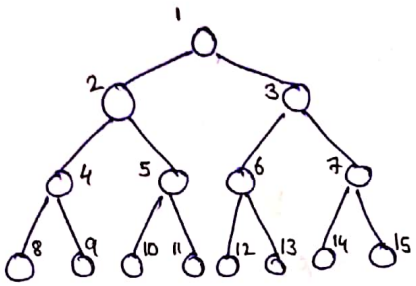
Analysis of Build heap:

(i) By default all leaf nodes satisfy heap tree property. So we don't need to apply heapify on leaf nodes.

(ii) If there a 'n' elements in array representation of complete binary tree then,

leaf node index starts from $\lfloor \frac{n}{2} \rfloor + 1$ and ends at n.

i.e., the highest index of non-leaf node is $\lfloor \frac{n}{2} \rfloor$



Here we call build heap on nodes

7, 6, 5, 4, 3, 2, 1 in this order.

$\lfloor \frac{n}{2} \rfloor = \lfloor \frac{15}{2} \rfloor = 7$

(iii) we have to apply heapify algorithm starting from highest index of non leaf node and continue the process till we reach the root node.

i.e.,
```
for (i=⌊n/2⌋; i≥1; i--)
{
    heapify(a,i); // a is array
}
```

(iv) The total time complexity for heapify algorithm.

In the worst case, an internal node may move from its position to leaf node position.

i.e., inwost case it may move (no of comparision) upto distance of its height.

∴ ~~time~~

∴ time complexity = sum of heights of all nodes of building heap using build heap $= O(n)$

↳ no of nodes.

(Q27) we have a binary heap on n elements and wish to insert n more elements (not neccessarily one after another) into this heap. The total time required for this is ⇓ Build heap

a) $\Theta(\log n)$
b) $\Theta(n)$
c) $\Theta(n \log n)$
d) $\Theta(n * 2)$

Sol:

we first insert 'n' node into array normally.

So we have 2n nodes.

we apply build heap on this.

$\Rightarrow O(n)$

i.e., $\Theta(n)$

(Q28) what is the time complexity for converting complete binary tree into max-heap if only root node does not satisfy the heap tree ~~prec~~ property.

a) $\Theta(\log n)$ b) $\Theta(1)$ c) $\Theta(n)$ d) $\Theta(n \log n)$

Sol:

In the worst case root node may be moved to leaf node.

This takes $\log_2 n$ comparision

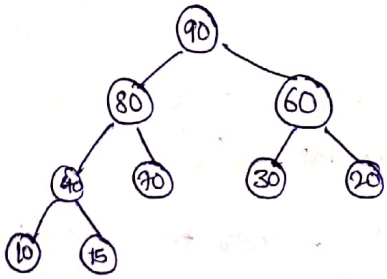
$\therefore O(\log n)$

Silly from above question,

if there is only one node in tree which doesn't satisfy the heap tree property, then converting the tree into heap tree take $O(\log n)$

Deletion Operation on heaptree:

In heap tree, which is a priority queue, we perform deletion only on root node.

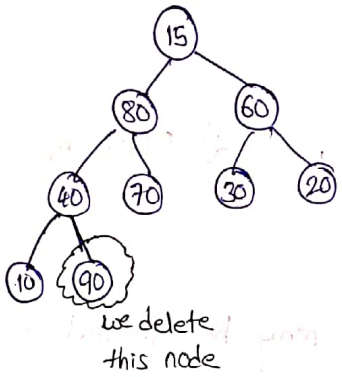


Step 1: Swap (A[1], A[n])

Step 2: Apply heapify algorithm on root node and convert complete binary of (n-1) nodes into heap tree.

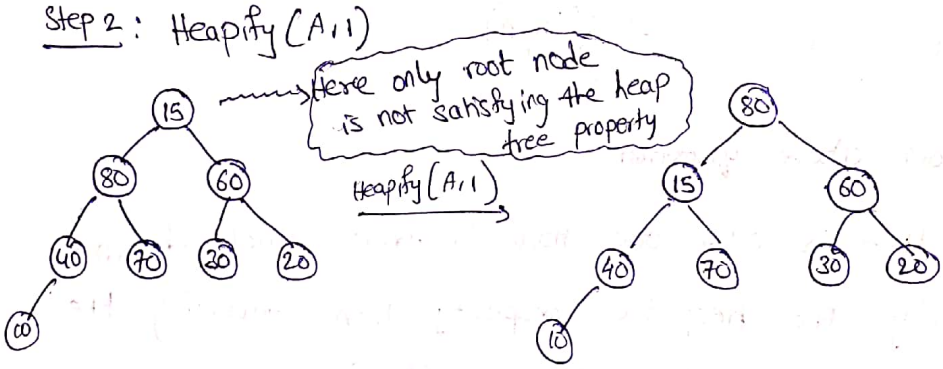
Delete 90:

Step 1: Swap (90, 15)

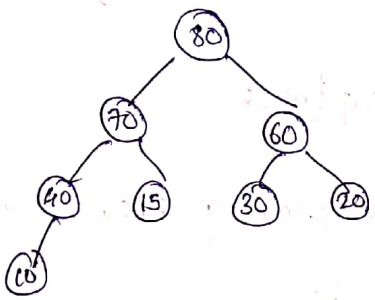


Now no of nodes = 9-1 = 8

Step 2: Heapify (A, 1)



Heapify (A, 2) is ↑



Analysis:

Swapping can be done in constant time.

Applying heapify on root node takes $O(\log n)$

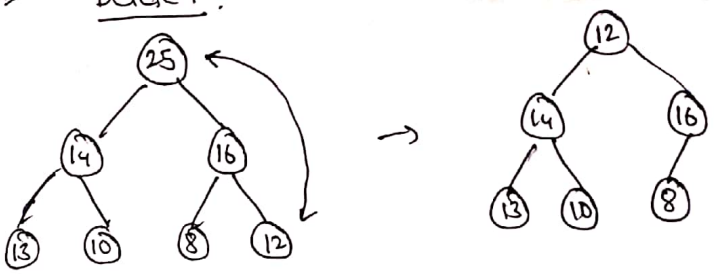
∴ time complexity for deletion on heap tree = $O(\log n)$

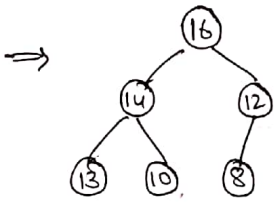
(Q29) Consider max heap implemented with array. as given below

25 14 16 13 10 8 12.

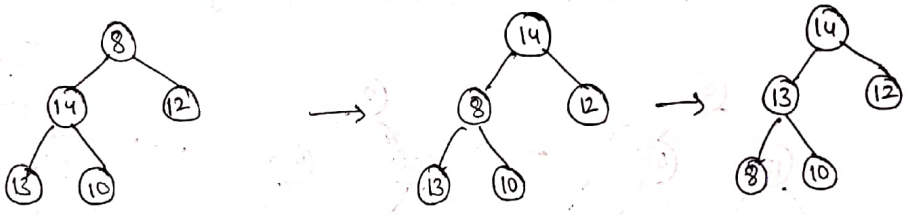
what is the content of the array after two delete operations?

Sol: Delete 1:





Delete:



∴ Ans: 14 13 12 8 10

(Q30) An operator delete(i) for a binary heap data structure is to be designed to delete the item in i-th node. Assume that the heap is implemented in an array and i refers to ith index of the array. If the heap has depth d (no of edges on the path from root to the furthest leaf) then what is the time complexity to re-fix the heap efficiently after the removal of the element?

a) $O(1)$ b) $O(d)$ but not $O(1)$

c) $O(2^d)$ but not $O(d)$ d) $O(d2^d)$ but not $O(2^d)$

Sol:

Here d is height

we perform this operation ~~similar to th~~ in a similar way we deleted the root node.

So the ~~delete~~ swapped node may move 'd' distance (in worst case)

$\therefore O(d)$

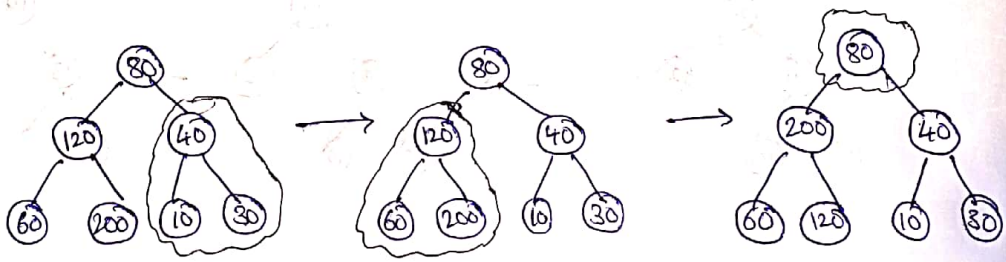
Heapsort:

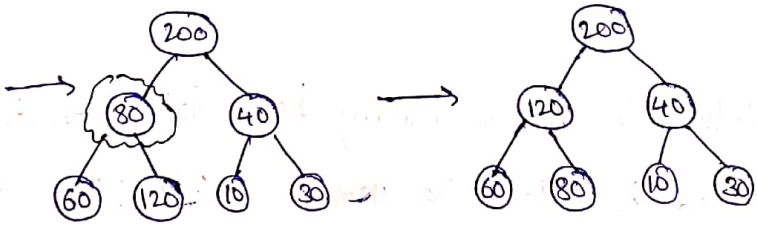
Sort the following array using heapsort.

1	2	3	4	5	6	7
80	120	40	60	200	10	30

Step 1: Convert the given array into max heap.

we do this using build heap algorithm .... $O(n)$

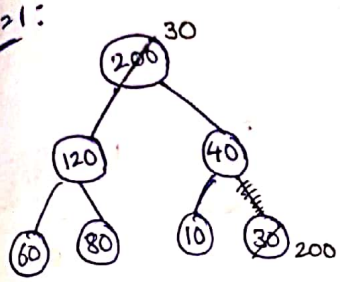




Step 2: In each iteration 'i', we have to delete root node by replacing it with leaf node at last index and by applying heapify (A, 1) by taking size of A as n-i

value of 'i' starts from 1. and runs till n-1

i=1:

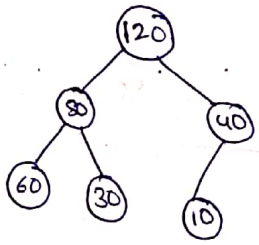


30	120	40	60	80	10	200

Heapify (A, 1)

size of A = 7 - i = 7 - 1 = 6

Heapify (A, 1)

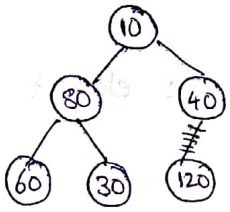


120	80	40	60	30	10	200

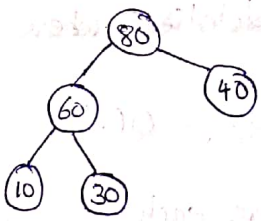
↳ Deleted from heap tree

i=2:

- Swap (A[1], A[n-i+1])
  - n = 6



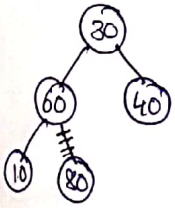
Heapify (A, 1) .. size of array = n - i = 7 - 2 = 5



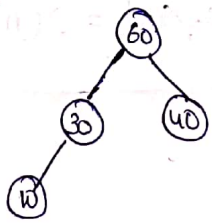
80	60	40	10	30	120	200

Delete from heap tree

i=3:



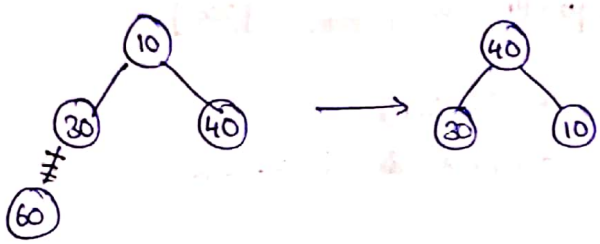
Heapify (A, 1)



60	30	40	10	80	120	200

↳ Deleted part

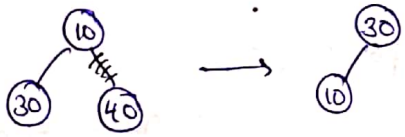
i=4



40	30	10	60	80	120	200

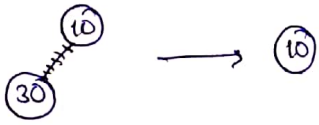
Deleted part

i=5



30	10	40	60	80	120	200

i=6:



10	30	40	60	80	120	200

Now this deletion of last but one element forms the sorted array.

Analysis:

~~Here deleting we perform 'n' deletions where each de~~

- Here we first construct heap ... O(n)

later we perform n deletions where each deletion takes $\log n$ time ... $O(n \log n)$

∴ Time complexity of heapsort = $O(n \log n)$

Finding maximum & minimum elements in heap tree:

	max heap	min heap
max element	O(1)	O(n)
Min element	O(n)	O(1)

Note: In max heap, the min element is always at leaf node

The leaf node index starts from $\lfloor \frac{n}{2} \rfloor + 1$ and ends at n

$\therefore$ no of comparisions = $n - \lfloor \frac{n}{2} \rfloor$

$= \lceil \frac{n}{2} \rceil - 1$

Sly for min heap also $\lceil \frac{n}{2} \rceil - 1$

$\therefore$ TC for finding min element in max heap & finding max element in min heap = O(n)

(Q31) Consider array representation of min heap containing 1023 elements. The minimum no of comparisions required to find maximum in the heap is ——

Sol:

leaf node index starts at $\lfloor \frac{n}{2} \rfloor + 1 = 511 + 1 = 512$

leaf node index ends at n = 1023

$\therefore$ no of leaf nodes = 1023 - 512 + 1

= 512

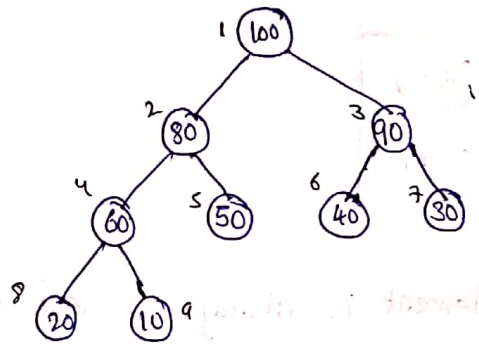
~~512 comparisons~~.

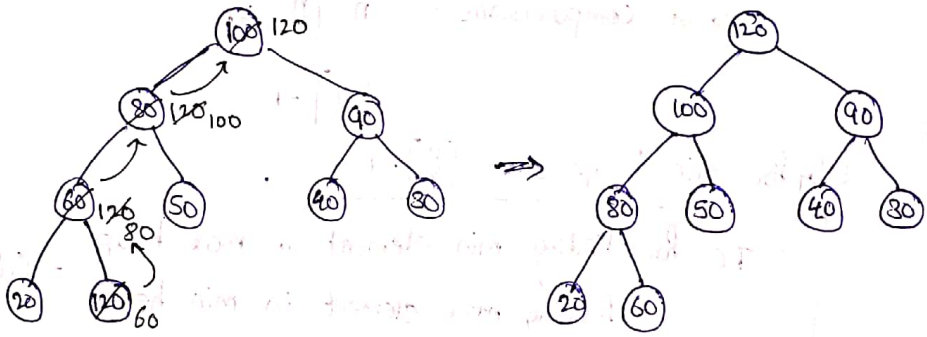
For k nodes we need k-1 comparisions.

$\therefore$ 512 - 1 = 511 comparisions

Increasing key (or) Decreasing key Operations

Consider Increase value of ~~key~~ A[9] to 120





Time Complexity : $O(\log_2 n)$
for increase key & decrease key

Note :

→ Insertion into heap tree : $O(\log n)$

→ Deletion on heap tree (root node) : $O(\log n)$

→ Increase or Decrease key : $O(\log n)$

→ Construction of heap tree
- insertion elements one after other : $O(n \log n)$
- Build heap method : $O(n)$

→ Heapsort : $O(n \log n)$

→ ~~Heapify~~

Find	max heap	min heap
max element	O(1)	O(n)
min element	O(n)	O(1)

(Q32) Consider the process of inserting an element into a max heap represented by an array. Suppose that we perform binary search on the path from the new leaf to root to find the position for the newly inserted element the no of comparisions performed is

a) $O(\log_2 n)$ b) $O(\log_2(\log_2 n))$ c) $O(n)$ d) $O(n \log n)$

Sol:

without using binary search we need compare approximately $\log_2 n$. elements.

∴ no of comparsion for binary search is $O(\log_2(\log_2 n))$

However Note that here the comparisions required is $O(\log_2(\log_2 n))$ is for find position of newly inserted element but not for entire insertion.

Note:

→ no of swap operations for insertion into heap tree using binary search are $O(\log n)$ i.e., same as using sequential search.

- It is because after finding comparing with 'mid' all the elements under the node 'mid' must be swapped.

Insertion	Comparision	swapping	TC
seq. search	$O(\log n)$	$O(\log n)$	$O(\log n)$
binary search	$O(\log_2 \log_2 n)$	$O(\log n)$	$O(\log n)$

(Q33) Suppose we have a balanced BST holding n numbers. we are given two number L and H and wish to sum up all the numbers in T that lie b/w L and H. Suppose there are m such numbers in T. If the tightest upper bound on the time to compute the sum is $O(n^a \log^b m + m^c \log^d n)$ then value of $a+10b+100c+1000d$

a) 60 b) 110 c) 210 d) 50

Sol:

given $O(n^a \log^b n + m^c \log^d n)$

we first search L in $O(\log n)$ time and then we search H in $O(\log n)$ time.

Now we perform inorder on L till H i.e, for m elements this can be done in $O(m)$ time

∴ time complexity is $O(\log n + m)$

a) $a=0 \quad b=1 \quad c=1 \quad d=0$

$\Rightarrow a+10b+100c+1000d = 110$

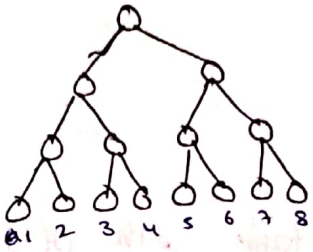
(Q34) Let T be a full binary tree with 8 leaves. Suppose two leaves a and b of T are choosen uniformly and independently at random. The expected value of the distance b/w a and b in T (i.e, the no of edges in the unique path b/w a and b) is ——

Sol:

The nodes are choosen independently

i.e, repetition is allowed.

∴ total ways for choosing $= 8 \times 8 = 64$



~~For this If~~

two nodes choosen can be same ---- path length = 0

this can be done in 8 ways

two nodes choose can be sibling --- path length = 2

this can be done in ~~2*4 = 8~~ 8 ways

two nodes choose may have path length = 4

~~this is possible if one node is from (1,2)~~ and ~~other~~

this is possible if one node is from (1,2) & other is from (3,4)

Sly we have for (5,6) & (7,8)

(1,2) & (3,4) ---- 2*2*2 = 8

(5,6) & (7,8) ---- 2*2*2 = 8

16

two nodes choosen may have path length = 6

this is possible if one nodes is from (1,2,3,4) and other is from (5,6,7,8)

no of ways = 2*4*4 = 32

∴ expected value = $0*\frac{8}{64} + 2*\frac{8}{64} + 4*\frac{16}{64} + 6*\frac{32}{64}$

$= \frac{2}{8} + \frac{8}{8} + \frac{8}{8} + \frac{24}{8} = \frac{2}{8} + 1 + 3$

$= \frac{42}{8} = 5.25$ $= 0.25 + 4$

$= 4.25$

22/09/20

Hashing:

1. Accessing data items from hash table with O(1) time, but the goal is not yet achieved.

2. Hash function:

$H(x) = x \bmod m$

H: Hash function

x: key

m: hash table size, where index of hash table ranges from 0 to m-1.

Eg: Let

x: 10, 20, 30, 40, 50

h(x): x mod 7

h(10) = 10 mod 7 = 3rd location

h(20) = 20 mod 7 = 6th location

h(30) = 30 mod 7 = 2

h(40) = 40 mod 7 = 5

h(50) = 50 mod 7 = 1

Index	key
0	
1	50
2	30
3	10
4	
5	40
6	20

Accessing data:

Assume we need to access key 40

H(40) = 40 mod 7 = 5

so we obtain 40 in the index 5.

∴ Time complexity for accessing data = frequency count of fundamental instructions. = O(1)

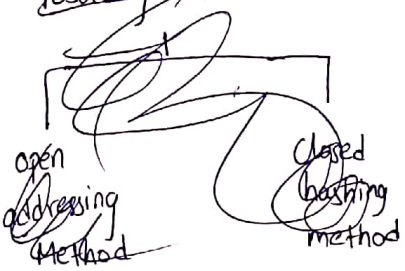
Eg: Let $x: 10, 20, 30, 40, 50$

$h(x): x \bmod 10$

$h(10) = 10 \bmod 10 = 0$
$h(20) = 20 \bmod 10 = 0$ } → collision

index	key
0	10
1	
2	
3	
4	
5	
6	
7	
8	
9	

resolving collision



Seperate chaining:

→ Here we resolve collisions by creating a separate linked list for each hash index

index	key
0	→ 10 → 20 → 30 → 40 → 50
1	
2	
3	
4	
5	
6	
7	
8	
9	

Accessing data:

Assume we need to access 50.

$H(50) = 50 \bmod 10 = 0$

So at 0th index we perform linear search.

Here

∴ Time complexity = $O(n)$

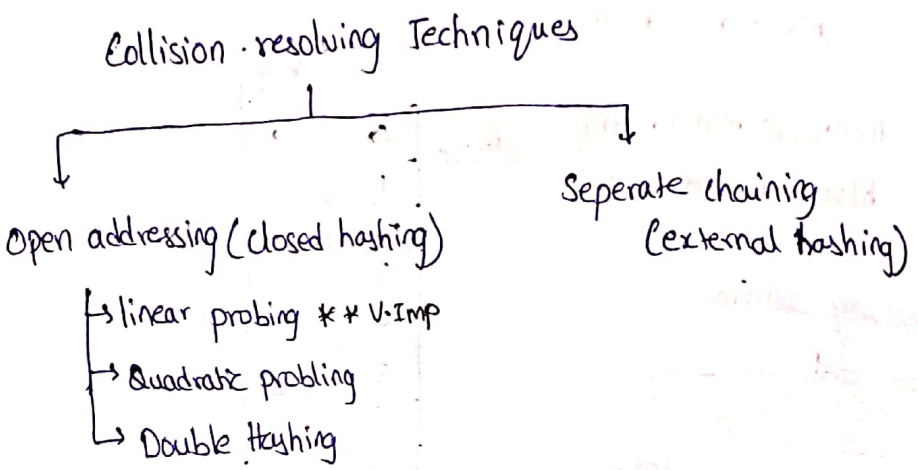
→ The hash function which takes $O(1)$ time for accessing data is called good hash function.

→ The hash function which takes $O(n)$ time for accessing data is called bad hash function.

Collision:

The situation in which more than one key hashes to same location is called collion.

Collisions can be resolved using following techniques.



Open Addressing:

i) Linear probing:

Let $h(x) = x \bmod m$

suppose $h(x)$ leads to collision and let collision number is 'i'.

$$H(x,i) = (h(x) + i) \bmod m$$

↑ key ↑ collision number.

If there are no collisions then value of 'i' is 0.

Eg:

keys: 14, 21, 28, 33, 54, 61

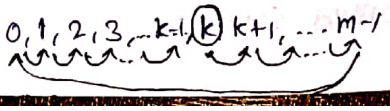
$h(x)$: $x \bmod 7$

Here load factor : $\frac{6}{7} \leq 1$

∴ we can use linear probing.

If m = hash table size and
n = no of keys then
load factor of hash table,
$\lambda = \frac{n}{m}$.
To apply linear probing,
$\lambda \leq 1$
sly for any closed hashing, $\lambda \leq 1$

Rule: If k^{th} location is already occupied by some key then we have to search in $(k+1)^{th}$ location, if $(k+1)$ is also occupied then we search in $(k+2)^{th}$ location and so on till $(m-1)^{th}$ and then we search from '0' to $(k-1)^{th}$ location



$h(x) = x \bmod 7$

14: $h(14) = 14 \bmod 7 = 0^{th}$ location
21: $h(21) = 21 \bmod 7 = 0^{th}$ location
} collision
↳ 1st collision

Now we resolve this using linear probing

now i=1 (collision number)

$H(21, i) : (h(21)+1) \bmod 7$

$= (0+1) \bmod 7 = 1^{st}$ location

$h(28) = 28 \bmod 7 = 0^{th}$ location

↓

1st collision

0	14
1	21
2	28
3	
4	
5	33
6	54

28:

$i=1, H(28,1) = (h(28)+1) \bmod 7$

$= (0+1) \bmod 7 = 1^{st}$ location

↓

2nd collision

$i=2, H(28,2) = (h(28)+2) \bmod 7$

$= (0+2) \bmod 7$

$= 2 \bmod 7 = 2^{nd}$ location

33:

$h(33) = 33 \bmod 7 = 5^{th}$ location

54:

$h(54) = 54 \bmod 7 = 5^{th}$ location

∴ we store in 6^{th} location

Q35 Given the following input (1322, 1331, 1171, 9679, 1989, 6171, 6173, 1199) and the hash function x mod 10. which of the following statements are true.

(i) 9679, 1989, 4199 hash to same value.

(ii) 1471, 6171 hash to the same value.

(iii) All elements has to the same value

(iv) Each element hashes to a different value.

a) (i) only b) (ii) only c) (i) & (ii) only d) iii or) iv

Sol:

(Q36) A hash table contains 10 buckets and uses linear probing to resolve collisions. The key values are integers and hash function is key % 10. If the values 43, 165, 62, 123, 142 are inserted in the table, in what location 142 will be inserted?

a) 2 b) 3 c) 4 d) 6

Sol:

43 --- 3rd loc

165 --- 5th loc

62 --- 2nd loc

123 --- 3 is filled
∴ 4th loc

142 --- 2nd is filled
3rd is filled
4th is filled
5th is filled
∴ 6th loc

(P/4) $h(x) = (3x+4) \bmod 7$

0	1
1	8
2	10
3	
4	
5	
6	3

$h(1) = (3+4) \bmod 7 = 0$th loc

$h(3) = 13 \bmod 7 = 6$

$h(8) = 28 \bmod 7 = 0$ --- collision
∴ insert at 1

$h(10) = 34 \bmod 7 = 6$ --- collision
∴ insert at 2

∴ 1, 8, 10, –, –, –, 3

P/6

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

P/3 load factor = $\frac{\text{no of elements}}{\text{no of slots}} = \frac{2000}{25} = 80$

P/7

	0	1	2	3	4	5	6	7	8	9
a			42	52	34	23	46	33		
b			42	23	34	52	33	46		
c			42	23	34	52	46	33		
d			42	33	23	34	46	52		

∴ opt (c)

P/8

0	
1	~~42~~
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

~~from hash~~ +

33 hashes to index '3'

However in the given table it present into the index 33.

This means 33 must be the last key of the sequence.

keys: | | | | | | 33 |

42 & 52 ~~map~~ hashes to 2nd index.

From figure 42 must have come first.

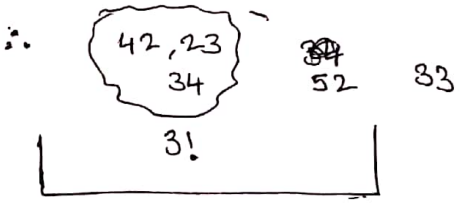
In fig., 23 is present in 3rd loc.

this means 23 must have come earlier than 52

Also 34 is in 4th index & 52 is in 5th index.

So 34 must have come earlier.

The only restriction for key 46 is that it must come before 33.

∴ 

46 can be inserted in any of these 5 loc in 5 ways

∴ no of ways = 3! × 5 = 30 ways

23/09/20

Disadvantage of Linear probing

Consider $h(x) = x \bmod 11$

keys:	29	46	18	36	43		21	24	54
h(x):	7	2	8	3	10		0	4	1

0	21
1	54
2	46
3	36
4	24
5	-
6	-
7	29
8	18
9	-
10	43

→ Now if a key to be inserted maps to 10th index, then it has to be stored in 5th index.

→ In this example 10, 0, 1, 2, 3, 4 forms a cluster of indices.

→ 76 mod 11 = 10

∴ it is stored at 5th index.

→ The no of probes for search 76 (or) inserting 76 := 7

→ The main disadvantage of linear probing is primary clustering i.e., group of records stored next to each other

Quadratic Probing:

→ In linear probing, the colliding keys are stored near the initial collision point, it results formation of cluster.

→ In quadratic probing, this problem is solved by storing the colliding key away from the initial collision point.

→ The formula for quardratic probing is

$$H(x,i) = (h(x)+i^2) \bmod m$$

In linear probing searching process is $h(x), h(x)+1, h(x)+2, \ldots \bmod m$

In quadratic probing searching process is $h(x), h(x)+1, h(x)+4, \ldots \bmod m$

Eg: let $h(x) = x \bmod 11$ use quadratic probing.

keys:	46	28	21	35	57	39	19	50
h(x):	2	6	10	3	2	7	8	4

0	57
1	
2	46
3	35
4	50
5	
6	28
7	39
8	19
9	
10	21

$h(35) = 35 \bmod 11 = 2$

$\Rightarrow H(35,1) = (2+1^2) \bmod 11 = 3$

→ $h(57) = 57 \bmod 11 = 2$

$H(57,1) = (2+1^2) \bmod 11 = 3$

$H(57,2) = (2+2^2) \bmod 11 = 6$

$H(57,3) = (2+9) \bmod 11 = 0$

→ $h(50) = 50 \bmod 11 = 6$

$H(50,1) = (6+1^2) \bmod 11 = 7$

$H(50,2) = (6+2^2) \bmod 11 = 10$

$H(50,3) = (6+3^2) \bmod 11 = 4$

Consider for some key, k;

$$h(k) = 2$$

Now $H(k,1) = (2+1) \bmod 11 = 3$

$H(k,2) = (2+4) \bmod 11 = 6$

$H(k,3) = (2+9) \bmod 11 = 0$

$H(k,4) = (2+16) \bmod 11 = 7$

$H(k,5) = 5$

$H(k,6) = 5$

$H(k,7) = 7$

$H(k,8) = 0$

$H(k,9) = 6$

⋮

Here any key that follows collision path 0,2,3,5,6,7 the problem of secondary clustering occurs.

(2,3,6,0,7,5,5,7,0,6,3,2,3,6,0,7,5,5---)
(observe the sequence)

So any key k with $h(k) = 2$, we search slot for it only in 6 distinct locations i.e., 0,2,3,5,6,7.

If the slots 0,2,3,5,6,7 are full, we cannot insert key 'k' despite having other free slots.

This problem is known as secondary clustering.

Here only half of the hash table is used.

Note:

→ Quadratic probing overcomes the problem of primary clustering, but it has another problem i.e., secondary clustering.

Note:

In quadratic probing, if the hash table size is m, then the no of location searched for finding empty location using quadratic probing is atmost $\frac{m+1}{2}$

Eg: In previous example for m=11

we can search only $\frac{m+1}{2} = 6$ locations /
(i.e., 0,2,3,5,6,7)

→ Hence only half of the hash table is searched.

Note: To get more no of locations for search, it is better to have hash table of prime size.

Double Hashing:

→ In double hashing, the increament factor is not constant as in linear and quadratic but depends on the key.

The increament factor is hash function.

So it is called double hashing.

→ Let $h(x) = x \bmod m$

The formula for double hashing is

$$H(x,i) = [h(x) + i(h'(x))] \bmod m$$

where i : probe number (collision number, i.e., $i=0 \Rightarrow$ no collision)

$h(x)$: primary hash function

$h'(x)$: secondary hash function.

Search for empty locations is $h(x)$

$[h(x) + 1 * h'(x)] \bmod m$

$[h(x) + 2 * h'(x)] \bmod m$

$[h(x) + 3 * h'(x)] \bmod m$

⋮

Eg: Let $h(x) = x \bmod 11$

$h'(x) = 7 - x \bmod 7$ $\quad H(x,i) = [h(x) + i * h'(x)] \bmod m$

Keys x	46	28	21	35	57	39	19	50
h(x)	2	6	10	9	8	1	3	2

→ $h(46) = 46 \bmod 11 = 2$

→ $h(28) = 28 \bmod 11 = 6$

→ $h(21) = 21 \bmod 11 = 10$

→ $h(35) = 35 \bmod 11 = 2$ (collision), $h'(35) = 7 - 35 \bmod 7 = 7$

$H(35,1) = (2 + 1*7) \bmod 11 = 9$

→ $h(57) = 57 \bmod 11 = 2$

$h'(57) = 7 - 57 \bmod 7 = 7 - 1 = 6$

$H(57,1) = (2 + 1*6) \bmod 11 = 8$

→ $h(39) = 39 \bmod 11 = 6$

$h'(39) = 7 - 39 \bmod 7 = 7 - 4 = 3$

$H(39,1) = (6 + 1*3) \bmod 11 = 9$

$H(39,2) = (6 + 2*3) \bmod 11 = 1$

→ $h(19) = 19 \bmod 11 = 8$

$h'(19) = 7 - 19 \bmod 7 = 2$

$H(19,1) = (8 + 1*2) \bmod 11 = 10$

$H(19,2) = (8 + 4) \bmod 11 = 1$

$H(19,3) = (8 + 6) \bmod 11 = 3$

∴ 3 probes for 19

↳ collisions

→ $h(50) = 50 \bmod 11 = 6$

$h'(50) = 7 - 50 \bmod 7 = 7 - 1 = 6$

$H(50,1) = (6 + 6) \bmod 11 = 1$

$H(50,2) = (6 + 12) \bmod 11 = 2$

∴ 2 probes for 50

Note:

→ The problem of secondary clustering is resolved in double hashing because the keys that have the same hash address probe different sequence of locations.

→ The disadvantage of double hashing is that we need to compute two hash functions and hence it takes more time compared to linear and quadratic probing.

Note:

while computing secondary hash function $h'(x)$, we must keep the following points in our mind.

→ $h'(x)$ does not give value '0'.

→ The value of $h'(x)$ has to be relatively prime to the size of hash table. (∵ to make distribution uniform)

Q37. Consider a double hashing scheme in which the primary hash function is $h_1(k) = k \bmod 23$ and the ~~necessary~~ secondary hash function is is $h_2(k) = 1 + (k \bmod 19)$. Assume that the size of hash table is 23. Then the address returned by probe 1 in probe sequence (assume that the probe sequence begins at probe 0) for key value $k = 90$ is ——

Sol:

$h_1(90) = 90 \bmod 23 = 21$

$h_2(90) = 1 + (90 \bmod 19) = 1 + 14 = 15$

$h(x) = (21 + 1 * 15) \bmod 23$

(probe 1; ↳ size of hash table)

$= 36 \bmod 23 = 13$

Load factor: If hash table size is 'm' and the no of keys which are hasing to hash table = n.

$\Rightarrow$ Load factor, $\lambda = \frac{n}{m}$

In general if $\lambda \leq 1$, then only we can apply linear probing or quadratic probing or double hashing

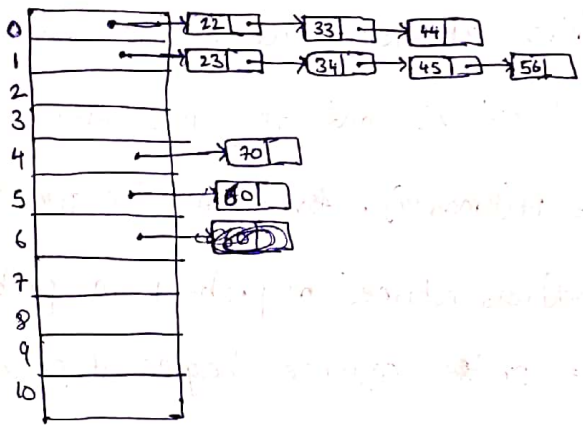
Separate Chaining

→ Separate Chaining can be applied for any value of load factor

$h(x) = x \bmod 11$

x: 22 33 44 23 34 45 56 60 70

h(x):



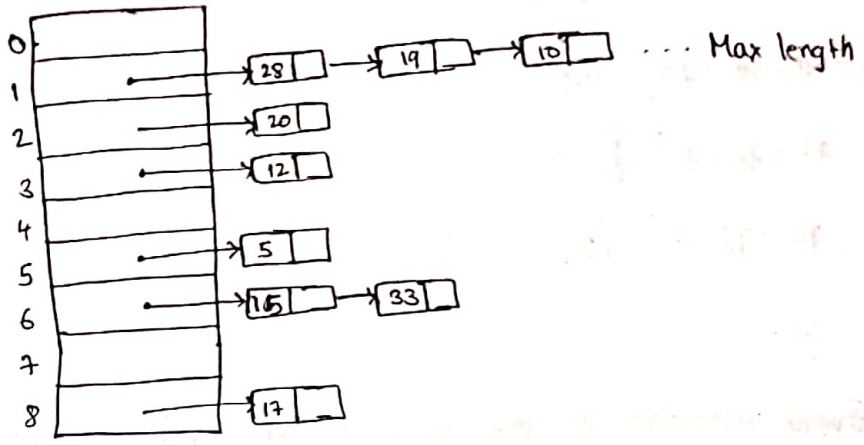
Here each index hold starting address of the list

Disadvantage:

→ More space is required for pointers.

Linear Probing	Quadratic Probing	Double Hashing	Separate Chaining
→ $(h(k)+i) \bmod m$	$(h(k)+i^2) \bmod m$	$(h(k)+i*h'(k)) \bmod m$	$h(x) = x \bmod m$
→ Primary clustering	Secondary clustering	works slow	More space is required
→ O(n) for insertion & searching in worst case	O(n) (worst case)	O(n)	O(n)
→ Deletion is difficult (Rehashing is required)	Deletion is difficult (Rehashing is req)	Deletion is difficult (Rehashing is req)	Deletion is easy

(P/2)



Max length = 3

Min length = 0

avg length = $\frac{\text{total keys}}{\text{no of slots}} = \frac{9}{9} = 1$

Note :

→ If hash indices are $0, 1, 2, 3, \cdots m-1$

then we use hash function

$h(x) = x \bmod m$.

→ If hash indices are $1, 2, 3, \cdots m$

then we use hash function

$h(x) = (x \bmod m) + 1$

because $x \bmod m$ never return has index m.

(P/1) hash index address range ... 1 to 1000

⇒ Hash function is $(x \bmod m) + 1$

(S38) Consider a hash table with 100 slots. Collisions are resolved using chaining. Assuming simple uniform hashing, what is the probability that the first 3 slots are unfilled after the first 3 insertions?

a) $(97 \times 97 \times 97) / 100^3$

b) $(99 \times 98 \times 97) / 100^3$

c) $(97 \times 96 \times 96) / 100^3$

d) $(97 \times 96 \times 95) / (3! \times 100^3)$

Sol:

Every insertion is independent of each other.

∴ probability that the inserted key doesn't fall into first 3 slots $= \frac{100-3}{100} = \frac{97}{100}$

∴ prequired probability $= \left(\frac{97}{100}\right)^3$

∴ opt (a)

(Q39) which one of the following hash function on integers will distribute keys most uniformly over 10 buckets numbered 0 to 9 for i ranging from 0 to 2020?

a) $h(i) = i^2 \bmod 10$

b) $h(i) = i^3 \bmod 10$

c) $h(i) = (11 * i^2) \bmod 10$

d) $h(i) = (12 * i) \bmod 10$

Sol:

i	$i^2 \bmod 10$	$i^3 \bmod 10$	$(11 * i^2) \bmod 10$	$(12 * i) \bmod 10$
0	0	0	0	0
1	1	1	1	2
2	4	8	4	4
3	9	7	9	6
4	6	4	6	8
5	5	5	5	0
6	6	6	6	2
7	9	8	9	4
8	4	2	4	6
9	1	9	1	8
10	0	0	0	0
11	1	1	1	2
⋮	⋮	⋮	⋮	⋮

6 distinct hash indices

10 distinct hash indices

6 distinct hash indices

5 distinct hash indices

$i^3 \bmod 10$ hashes to every hash index for keys 0 to 9
" " 10 to 19
" " 20 to 29
⋮

$\therefore i^3 \bmod 10$ distributes keys uniformly.

$\therefore$ opt (b)

24/09/20

(Q40) Consider a hash table with n buckets where external chaining is used to resolve collisions. The hash function is such that the probability that a key value is hashed to a particular bucket is $\frac{1}{n}$. The hash table is initially empty and k distinct values are inserted in the table.

a) what is the probability that bucket number 1 is empty after the k^{th} insertion

Sol:

Probability that a key is not inserted into bucket 1 is $\frac{n-1}{n}$

$\therefore$ req probability $= \left(\frac{n-1}{n}\right)^k$

b) what is the probability that no collision has occured in any of the k insertions?

Sol:

total no of ways to places k keys into n buckets is n^k

no of ways to ~~pet~~ place k keys into n buckets such that there is no collision $= {}^nP_k$

$\therefore$ req probability $= \frac{{}^nP_k}{n^k}$

c) what is the probability that the first collision occurs at the k^{th} insertion?

Sol:

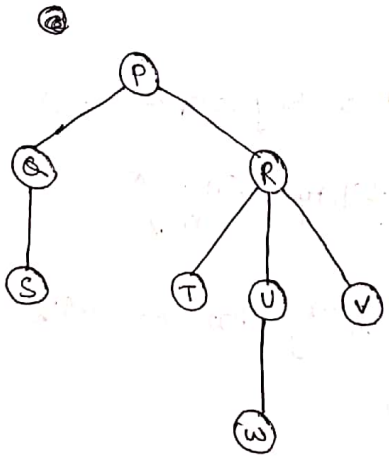
Probability that there is no collision for first k-1 insertions

$$= \frac{{}^{n}P_{k-1}}{n^{k-1}}$$

Probability that k^{th} insertion is a collision is $\frac{k-1}{n}$

$$\therefore \text{req probability} = \frac{{}^{n}P_{k-1}}{n^{k-1}} \cdot \frac{k-1}{n} = \frac{{}^{n}P_{k-1}(k-1)}{n^{k}}$$

(Q41) Consider the following rooted tree with the vertex P labeled as root



The order in which the nodes are visited during inorder traversal is

a) SQPTRWUV

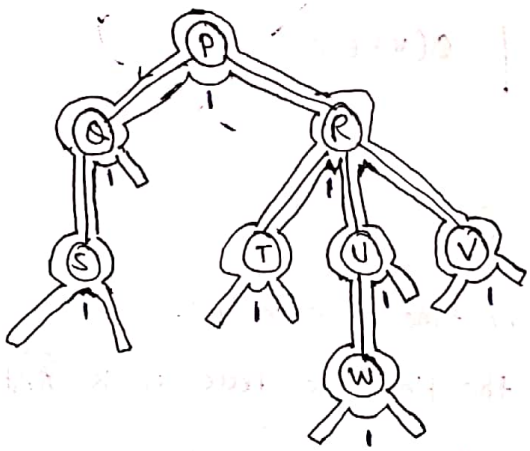
b) SQPTURWV

c) SQPTWUVR

d) SQPTRUWV

Sol:

Sol:
Here S & W must be treated as leftmost children of Q & V respectively



S Q P T R W U V

:opt a

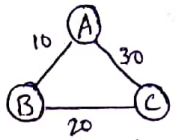
Graph Traversals:

(i) Depth First Search (DFS)

(ii) Breadth First Search (BFS)

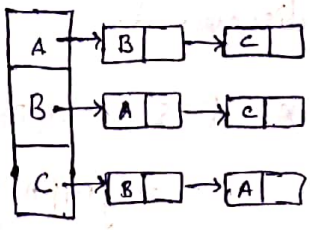
Representation of Graph:

G(V, E)



Adjacency Matrix [O(V²)]

Adjacency List [O(V+E)]



weighted graph

	A	B	C
A	0	10	30
B	10	0	20
C	30	20	0

unweighted graph

	A	B	C
A	0	1	1
B	1	0	1
C	1	1	0

Note :

Representation	TC
Adjacency Matrix	$O(V^2)$
Adjacency List	$O(V+E)$

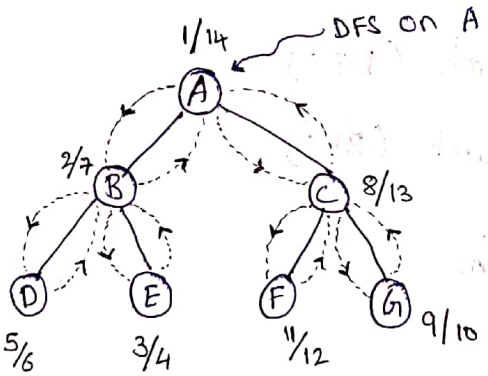
Depth First Search (DFS):

d[u] : Discovery time of node 'u'.
i.e., the time the node 'u' is first visited.

f[u] : finishing time of a node 'u'
i.e., the time the node 'u' is last visited.

→ At every node, let us follow the general convention $\frac{d[u]}{f[u]}$

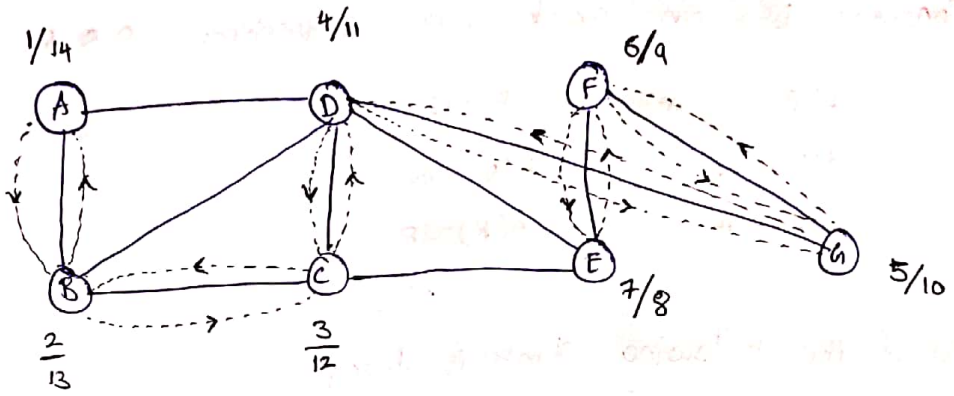
Also finishing time of a node is greater than discovery time of that node.



DFS : A B E D C G F

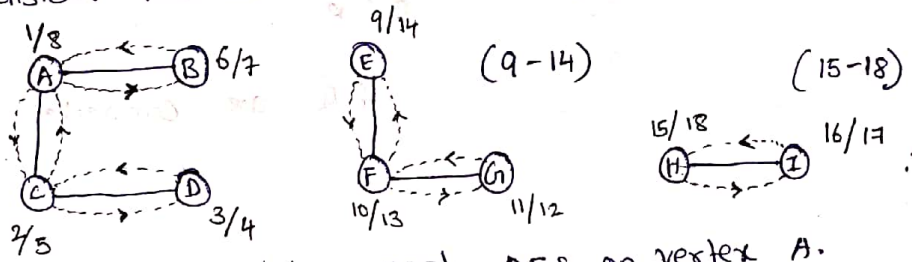
→ we visit only those nodes which are yet to be discovered.

→ Here DFS is applied only once. i.e., on A
⇒ graph is connected.



DFS: A B C D G F E

Consider the below disconneted graph



(1-8) Now let us apply DFS on vertex A.

DFS: Here we applied DFS 3 times (on A, on E, on H)

⇒ the graph is disconnected with 3 connected components.

Note:

(i) In DFS nodes get visited in deeper direction until we reach either a leaf node or a node whose adjacent nodes have already been visited.

(ii) It uses stack data structure.

(iii) The time complexity for DFS using

adjacency list is $O(V+E)$

adjacency matrix is $O(V^2)$

(iv) If the finishing time of starting vertex is greater than finishing time of remaining all vertices, then the graph is said to be connected.

(Q42) Consider DFS on graph with 3 vertices P, Q, R.

$d(P) = 5$ $f(P) = 12$

$d(Q) = 6$ $f(Q) = 10$

$d(R) = 14$ $f(R) = 18$

which of the following stmts is true?

a) graph is connected.

b) two connected components, P & R are connected.

c) " " " , Q & R " "

d) " " " , P & Q are connected

Sol:

P is discovered first and P's finishing time is greater than finishing time of Q only.

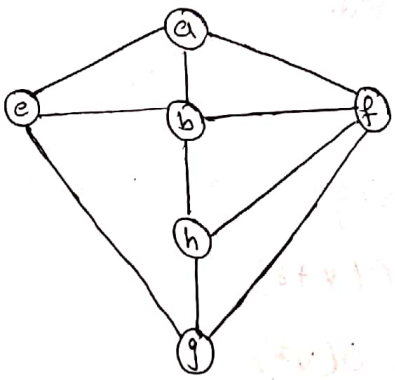
$\Rightarrow$ P & Q are connected.

R is discovered at 14 and finished at 18.

$\therefore$ R forms a connected component.

$\therefore$ opt (d)

(Q43)



Among the sequences

I) a b e g h f

II) a b f e h g

III) a b f e h g e

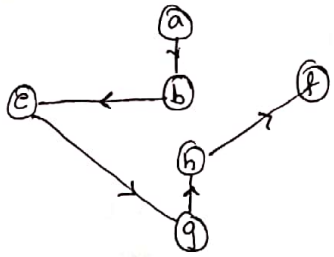
IV) a f g h b e

which of the above are valid DFS traversals

a) I, II, IV b) I & IV c) II, III, IV d) I, III, IV

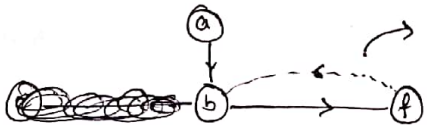
Sol :

a) a b e g h f



∴ valid

b)



→ Here f has adj unvisited vertex

∴ e cannot follow f

∴ Invalid

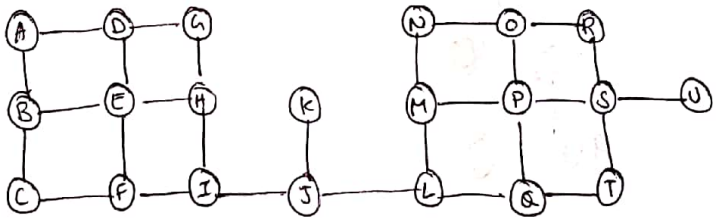
c) a → b → f → h → g → e

∴ valid

d) Silly (d) is valid

∴ opt (d)

P/6



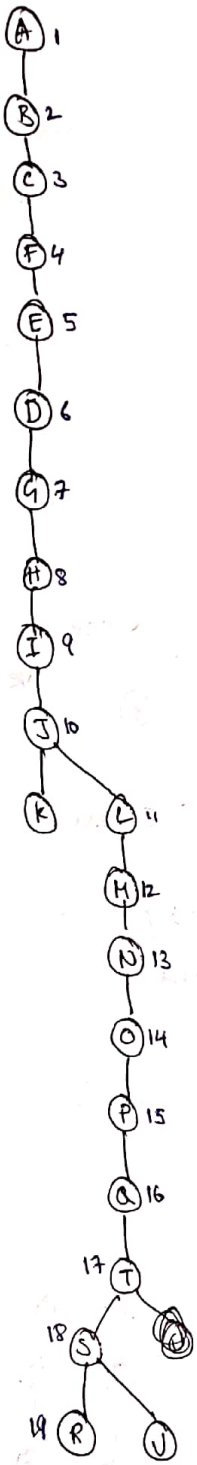
Consider starting ~~recur~~ DFS on A

~~A-B-C-F-E-D-G-H-I-J-K-L-M-N-O-P-Q-T-S-U~~

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

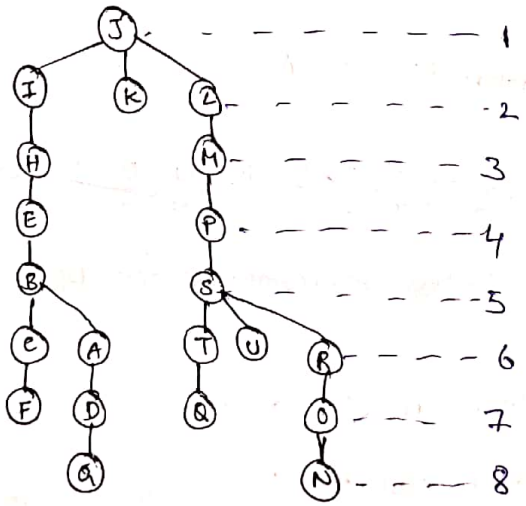
~~∴ Max recursion depth = 19~~

Recursion depth is no of recursive calls from the topmost recursive call to the bottom most one.



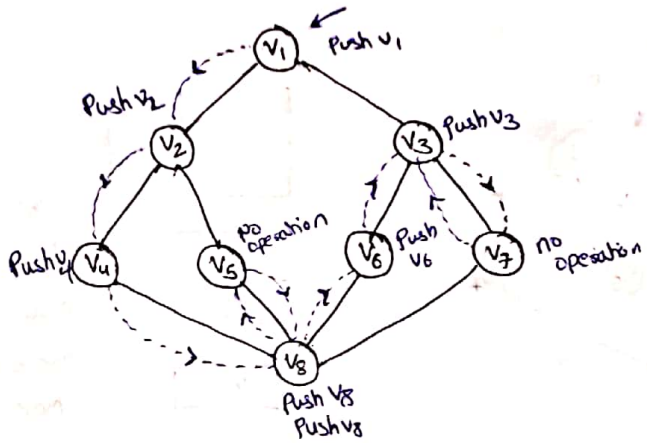
∴ Max recursion depth = 19

P/7



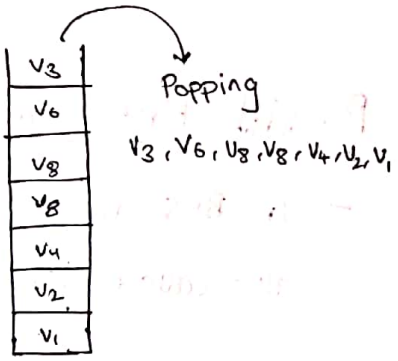
∴ Mini recursion depth = 8

P/1



whenever we visit a node, for DFS traversal, we push a node into stack iff it has atleast one adjacent node which is unvisited.

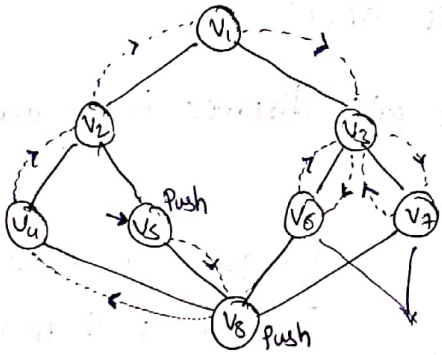
DFS: $V_1, V_2, V_4, V_8, V_5, V_6, V_3, V_7$

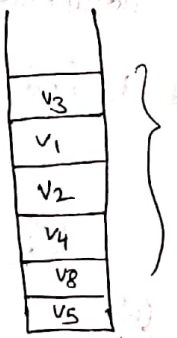


we pop a node, if there is no any unvisited adjacent node left.

∴ V_8 is pushed more than once.

P/2





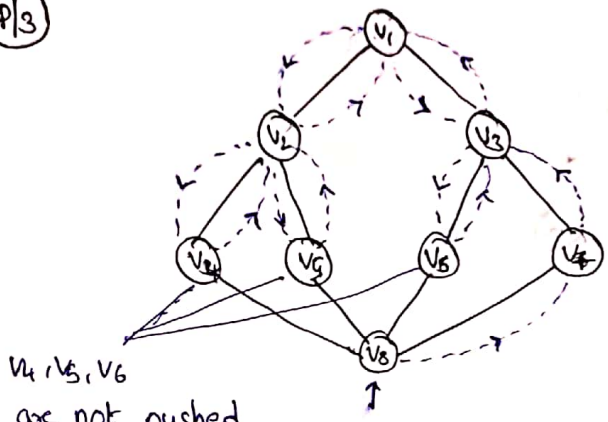
This stack doesn't show nodes that are pushed more than once

V_6 & V_7 are not pushed

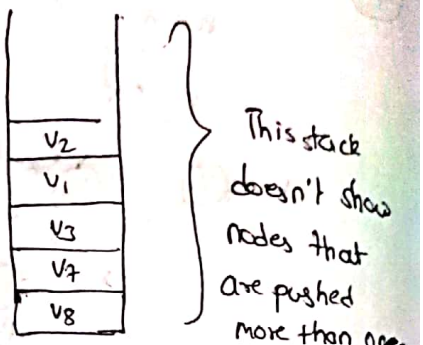
DFS: $V_5, V_8, V_4, V_2, V_1, V_3, V_6, V_7$

∴ 2 nodes

Q/3



v_4, v_5, v_6 are not pushed



DFS: v_8 v_7 v_3 v_1 v_2 v_4 v_5 v_6

∴ 3 nodes

Breadth First Search (BFS):

→ In BFS, nodes get visited level by level. so BFS is also called level order traversal.

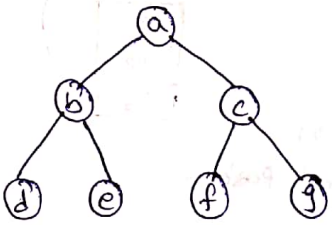
→ To implement BFS, we can use queue data structure.

→ Time complexity for finding BFS using

Adjacency list is $O(V+E)$

adjacency matrix is $O(V^2)$

→ BFS is used to find shortest path distance in an unweighted graph.



(i) Enqueue the node

(ii) Explore its children

(iii) dequeue the node.

Applying BFS on vertex 'a'.

BFS(a)

a	b	c

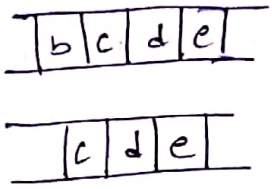
BFS: a b c

now enqueue b & c

now we have no more unvisited children of a.

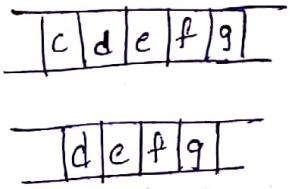
∴ dequeue(a): | b | c |
|---|---|

Now explore unvisited adjacent nodes of 'b'.



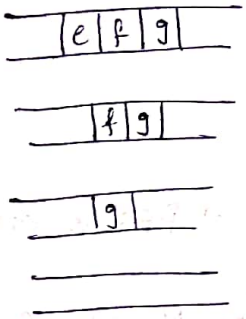
BFS: a b c d e

Now explore unvisited adjacent nodes of 'c'



BFS: a b c d e f g

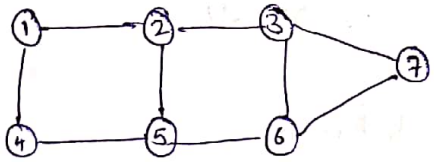
Now explore childeren of d, e, f, g



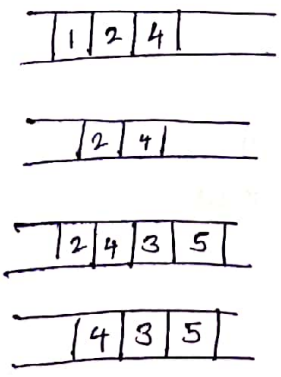
... empty queue

∴ BFS: a b c d e f g

Eg:

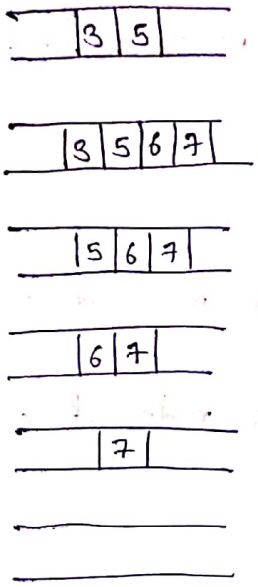


Applying BFS on '1'



BFS: 1 2 4

BFS: 2 4 3 5

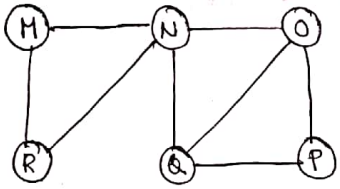


BFS : 1 2 4 3 5 6 7

----- empty queue

∴ BFS traversal : 1 2 4 3 5 6 7

P|10



a) M N O P Q R ⇒ BFS is called on M

O is not adjcent to M and O is visited before R which is adjacent to M.

∴ not valid

b) N Q M P O R ⇒ BFS on N

P is visited before R

but R is adjacent to P

∴ not valid

c) Q M N R O P ⇒ BFS on Q

M is not adjacent to Q

∴ invalid

d) P O Q N M R

∴ valid

Data Structure

Data structure deals with physical representation of data

It gives information about storage representation of data, accessing data from the memory in efficient manner.

types

Linear DS

(Data is represented in sequential way)

- → Array
- → stack
- → Queue
- → Linked list

Nonlinear DS

(Data is represented on a plane)

- → Tree
- → Graphs
- → Hashing

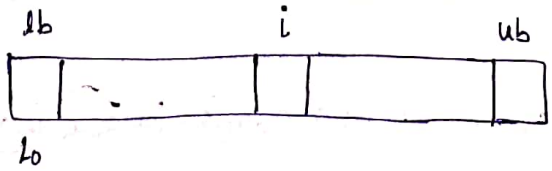
Arrays

Formula for 1-D array:

Consider A[lb....ub] of elements lb → lower boundary (or) lower index
ub → upper boundary (or) upper index

let c = size data at an index

l_0 = Base address of array



⇒ Loc(A[i]) = l_0 + (no of elements crossed before reaching i) x c

$$Loc(A[i]) = l_0 + (i - lb) * c$$

(P/1) $lb = -5 \quad ub = 5$

$L_0 = 1000, \quad c = 2$

$$Loc(A[0]) = L_0 + (i - lb) * c$$

$$= 1000 + (0 - (-5)) * 2$$

$$= 1010$$

Two Dimensional array:

Consider 2D array A

LB of col UB of col

A : array $[b_1, \ldots u_1, b_2 \ldots u_2]$ of elements

(b_1: lower boundary of rows; u_1: UB of rows)

Let L_0 : Base address

c : size of an element

r_1 : no of rows $= u_1 - b_1 + 1$

r_2 : no of columns $= u_2 - b_2 + 1$

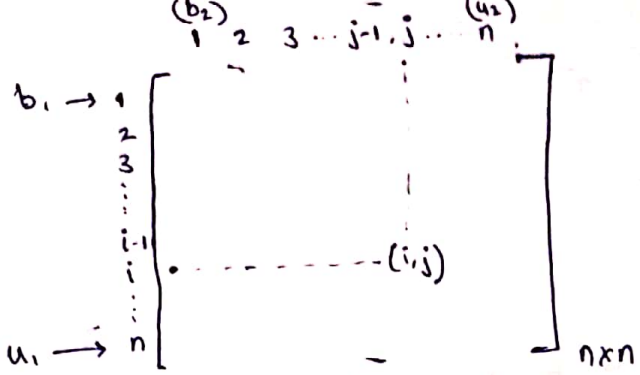
	b_2 = 1	2	3	4	u_2 = 5
b_1 = 1	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
2	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
3	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
4	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}
u_1 = 5	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}

Row major ordering (RMO): (By default we consider RMO)

a_{11} a_{12} a_{13} a_{14} a_{15}	a_{21} a_{22} a_{23} a_{24} a_{25}	a_{31} a_{32} a_{33} a_{34} a_{35}	a_{41} a_{42} a_{43} a_{44} a_{45}	a_{51} a_{52} a_{53} a_{54} a_{55}

Column Major Ordering (CMO):

a_{11} a_{21} a_{31} a_{41} a_{51}	a_{12} a_{22} a_{32} a_{42} a_{52}	a_{13} a_{23} a_{33} a_{43} a_{53}	a_{14} a_{24} a_{34} a_{44} a_{54}	a_{15} a_{25} a_{35} a_{45} a_{55}



Finding Loc(A[i][j]):

RMO:

$$Loc(A[i][j]) = L_0 + \left(\begin{array}{c}\text{no of rows crossed}\\ \text{before reaching ith row}\end{array}\right) * \left(\begin{array}{c}\text{elements per}\\ \text{row}\end{array}\right) * c$$

(no of columns)

$$+ \left(\begin{array}{c}\text{no of elements crossed}\\ \text{in ith row before}\\ \text{reaching jth element}\end{array}\right) * c$$

$$Loc(A[i][j]) = L_0 + [(i-1)*n + (j-1)] * c$$

$$\boxed{Loc(A[i][j]) = L_0 + [(i-b_1)*(u_2-b_2+1) + (j-b_2)] * c}$$

(u_2-b_2+1) = no of elements per row = no of columns

CMO:

$$Loc(A[i][j]) = L_0 + \left(\begin{array}{c}\text{no of columns crossed}\\ \text{before reaching } j^{th} \text{ column}\end{array}\right) * \left(\begin{array}{c}\text{elements per}\\ \text{column}\end{array}\right) * c$$

$$+ \left(\begin{array}{c}\text{no of elements to be}\\ \text{crossed in jth column}\\ \text{to reach element in}\\ \text{ith row}\end{array}\right) * c$$

$$= L_0 + [(j-1)*n + (i-1)] * c$$

$$\boxed{Loc[A[i][j]) = L_0 + [(j-b_2)*(u_1-b_1+1) + (i-b_1)] * c}$$

(u_1-b_1+1) = no of elements per column = no of rows

P/2 A: array [-2...2, 3...7]

$b_1 = -2 \quad u_1 = 2$

$b_2 = 3 \quad u_2 = 7$

$L_0 = 1000$

no of rows = $u_1 - b_1 + 1 = 2 + 2 + 1 = 5$

no of columns = $u_2 - b_2 + 1 = 7 - 3 + 1 = 5$

$c = 2$

RMO: (no of elements per row = no of cols)

$$Loc(A[0,5]) = L_0 + ([0-(-2)] * 5 * 2) + (5-3) * 2$$

$$= 1000 + 20 + 4$$

$$= 1024$$

CMO:

$$Loc(A[0,5]) = L_0 + [(5-3) * (\text{no of rows}) * 2] + (0-(-2)) * 2$$

$$= 1000 + 2 * 5 * 2 + 2 * 2$$

$$= 1024$$

P/3 A: array [1....10][1...15]

$b_1 = 1 \quad u_1 = 10$ no of rows = $u_1 - b_1 + 1 = 10$

$b_2 = 1 \quad u_2 = 15$ no of cols = $u_2 - b_2 + 1 = 15$

$c = 1$

$L_0 = 100$

given RMO

$$A[i][j] = 100 + (i-1) * (15) * (1) + (j-1) * 1$$

(15) ↓ no of cols

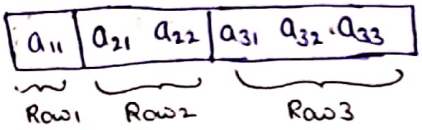
$$= 100 + 15i - 15 + j - 1$$

$$= 15i + j + 84$$

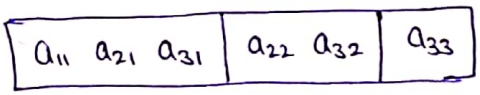
Lower Triangular matrix:

$$\begin{bmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

RMO:



CMO:



no of non-zero elements in 1st row = 1

" " " " " 2nd row = 2

" " " " " 3rd row = 3

" " " " " nth row = n

∴ no of non-zero elements in a lower triangular matrix of order $n \times n = \frac{n(n+1)}{2}$

RMO for lower triangular matrix:

$$Loc(A[i][j]) = L_0 + \begin{pmatrix} \text{no of non-zero} \\ \text{elements in } (i-1) \text{ row} \end{pmatrix} * c + \begin{pmatrix} \text{no of non-zero} \\ \text{elements crossed} \\ \text{in ith row} \end{pmatrix} * c$$

$$Loc(A[i][j]) = L_0 + \left[\left(\sum(i-1) + (j-1)\right)\right] * c \quad , \text{if } i \geq j$$

$$= 0 \quad , \text{if } i < j$$

CMO formula for lower triangular matrix:

$$Loc(A[i][j]) = L_0 + \begin{pmatrix}\text{no of non-zero}\\ \text{elements in } (j-1)\\ \text{columns}\end{pmatrix} * c + \begin{pmatrix}\text{no of non-zero}\\ \text{elements crossed}\\ \text{in } j^{th} \text{ column}\\ \text{to reach } i^{th} \text{ row}\end{pmatrix} * c$$

$$= L_0 + [n + (n-1) + \cdots + (n-(j-1)+1)] * c + \underbrace{[i-(j-1)-1]}_{\text{no of zeroes in } j^{th} \text{ col}} * c$$

no of non-zero elements crossed in j^{th} column

$$= L_0 + [n + (n-1) + \cdots + (n-(j-2))] * c + [i-(j-1)-1] * c$$

$$= L_0 + [(j-1)n - \Sigma(j-2) + i - (j-1)] * c$$

$$= L_0 + [n(j-1) + i - \Sigma(j-1)] * c$$

$$= L_0 +$$

$$= L_0 + [n(j-1) - \Sigma(j-2) + i - j] * c$$

$$\text{Loc}(A[i][j]) = L_0 + \left[n(j-1) - \frac{(j-2)(j-1)}{2} + i - j\right] * c$$

- $n(j-1)$: elements in $j-1$ columns
- $\frac{(j-2)(j-1)}{2}$: zero elements in $j-1$ cols
- $i-j$: non-zero elements crossed in j^{th} column

Upper Triangular matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22} & a_{23} & a_{24} \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{44} \end{bmatrix}$$

RMO:

$$a_{11}\ a_{12}\ a_{13}\ a_{14} \mid a_{22}\ a_{23}\ a_{24} \mid a_{33}\ a_{34} \mid a_{44}$$

CMO:

a_{11}	a_{12} a_{22}	a_{13} a_{23} a_{33}	a_{14} a_{24} a_{34} a_{44}

- From the formulae of Loc(A[i][j]) in lower triangular matrix replace i by j and j by i.

RMO: $$Loc(A[i][j]) = L_0 + \left[n(i-1) - \frac{(i-1)(i-2)}{2} + (j-i) \right] * c$$

CMO: $$Loc(A[i][j]) = L_0 + \left[\Sigma(j-1) + (i-1) \right] * c$$

Tridiagonal Matrix:

The matrix in which the elements except principle diagonal elements, elements above & below principle diagonal, rest of the elements a zeroes.

i.e.,
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix}$$

i.e.,
$$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix}$$

RMO:

a_{11} a_{12}	a_{21} a_{22} a_{23}	a_{32} a_{33} a_{34}	a_{43} a_{44} a_{45}	a_{54} a_{55}

→ In tridiagonal matrix, except first row & last row, every other row contains 3 non-zero elements.

$$Loc(A[i][j]) = L_0 + \begin{pmatrix}\text{no of non-zero}\\ \text{elements crossed}\\ \text{in } (i-1) \text{ rows}\end{pmatrix} * c + \begin{pmatrix}\text{no of non-zero}\\ \text{elements crossed}\\ \text{in ith row}\end{pmatrix} * c$$

$$= L_0 + [3(i-1) - 1] * c + [(j-1) - (i-2)] * c$$

- $3(i-1)$: every row has 3 non-zero elements except 1st row. 1st row has only 2 non-zero elements.
- $(j-1)$: total elements crossed
- $(i-2)$: no of zeroes in ith row

$$= L_0 + [3i - 4 + j - i + 1] * c$$

RMO:

$$\Rightarrow \boxed{\begin{aligned} Loc(A[i][j]) &= L_0 + (2i + j - 3), && 0 \le |i-j| \le 1 \\ &= 0, && |i-j| > 1 \end{aligned}}$$

CMO for tridiagonal matrix:

$$Loc(A[i][j]) = L_0 + \begin{pmatrix}\text{no of non-zero}\\ \text{elements crossed}\\ \text{in } (j-1) \text{ columns}\end{pmatrix} * c + \begin{pmatrix}\text{no of non-zero}\\ \text{elements crossed}\\ \text{in jth row}\end{pmatrix} * c$$

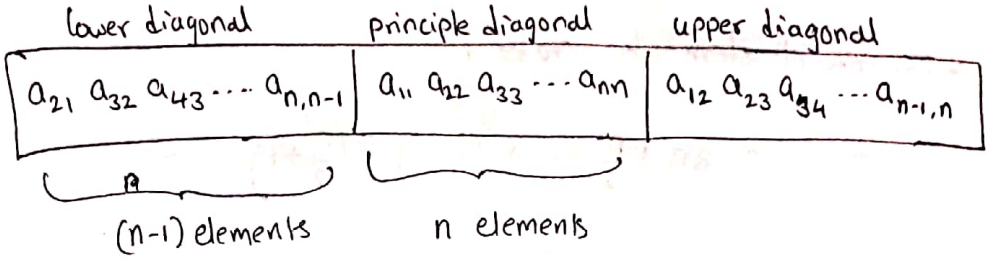
$$= L_0 + [3(j-1) - 1] * c + [(i-1) - (j-2)] * c$$

$$= L_0 + [3j - 4 + i - j + 1] * c$$

$$= L_0 + [i + 2j - 3] * c$$

$$\boxed{\begin{aligned} Loc(A[i][j]) &= L_0 + (i + 2j - 3) * c, && \text{for } 0 \le |i-j| \le 1 \\ &= 0, && \text{for } |i-j| > 1 \end{aligned}}$$

P/8



it is clear that we need to cross $(n+n-1)$ no of elements to reach upper diagonal element.

To reach an upper diagonal element $a_{i,i+1}$ we need to cross $(i-1)$ upper diagonal elements.

$\therefore$ total no of elements crossed $= (2n-1) + (i-1)$

$\therefore Loc(A[i][j]) = L_0 + (2n-1) + (i-1)$

P/9

Before accessing the elements of 1st column we must cross the elements of all 3 rows.

now for element in ith of 1st column we need to cross $(i-1) - 1$

this element is already included in row 1

$\therefore$ no of elements crossed $= 3*n + i - 2$

$\therefore$ index of A[i][j] in 1-D array is $3*n + i - 2$

Before accessing the elements of nth column we must cross $3n$ elements (3 rows) and $\frac{n}{2} - 1$ elements (1st column)

$\therefore$ In the last column to access ~~1st elements~~ element in ith row we need to cross $(i-1) - \left(\frac{n}{2} + 1\right)$

elements that '0' or already allocated in 1-D array

∴ total no of elements crossed

$$= 3n + \frac{n}{2} - 1 + (i-1) - \left(\frac{n}{2}+1\right)$$

$$= 3n + \frac{n}{2} - 1 + i - 1 - \frac{n}{2} - 1$$

$$= 3n + i - 3$$

∴ index of A[i][j] which is in n^{th} column is $3n - i - 3$

Toeplitz Matrix:

A nxn matrix T is a Toeplitz matrix iff

$T(i,j) = T(i-1, j-1)$ for all $i>1, j>1$

For a toeplitz matrix of order n,

no of independent (distinct) elements $= 2n-1$

(P/10) For $i=1$ or $j=1$

A[i][j] value is independent

Let A[1][1] = a

A[1][2] = b

A[1][3] = c

A[1][4] = d

A[1][5] = e

A[2][1] = f

A[3][1] = g

A[4][1] = h

A[5][1] = i

$$\begin{bmatrix} a & b & c & d & e \\ f & a & b & c & d \\ g & f & a & b & c \\ h & g & f & a & b \\ i & h & g & f & a \end{bmatrix}$$

∴ optimum no of elements stored = 9

RMO:

$$Loc(A[i][j]) = L_0 + (j-i) * c \quad , \text{for } i \le j$$
$$= L_0 + (n+i-j-1) * c \quad , \text{for } i > j$$

Square Band Matrix:

A Square Band Matrix $D_{n,a}$ is an $n \times n$ matrix in which all non-zero elements lie in a band centered around the main diagonal. The band includes the main diagonal and a-1 diagonal below and above the main diagonal.

P/11

In a square matrix of order n,

no of elements in each diagonal as we move away to the main diagonal are $n-1, n-2, \ldots$

In $D_{1000,101}$ we have $101-1 = 100$ diagonals other than main diagonal.

no of elements in band $= [(n-1)+(n-2)+\ldots+(n-100)] * 2 + 1000$

$= 100n - \frac{100(100+1)}{2} * 2 + 1000$

$= 100000 - 10100 + 1000$

$= 190900$

P/12

$n-(a-1), n-(a-2), n-(a-3), \ldots n-1, n$ are the ~~elem~~ no. of elements in each diagonal from lowest diagonal.

No of elements before k^{th} diagonal

$$= [n-(a-1)] + [n-(a-2)] + \cdots + [n-(a-(k-1))]$$

$$= [(n-a)+1] + [(n-a)+2] + \cdots + [(n-a)+(k-1)]$$

$$= (k-1)(n-a) + \frac{k(k-1)}{2}$$

elements:

$a_{21}, a_{32}, a_{43}, a_{54}$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix}$$

In any lower diagonal the column index starts from '1'

i.e., 'j' starts from 1.

∴ within k^{th} diagonal to access of A[i][j] we need to cross (j-1) elements

⇒ total no of elements crossed

⇒

$$= (k-1)(n-a) + \frac{k(k-1)}{2} + (j-1)$$

Method 2:

Consider $\begin{bmatrix} a & b \\ b & c \end{bmatrix}$

Here ~~k=2~~ n=2

$a-1=1 \Rightarrow a=2$

let i=2 j=1 i.e., we are accessing 'b'.

$\Rightarrow k=1$

The elements are stored in 1-D array as

b	a	c

a) $L_0 + [0 + 0 + 0] = L_0$ ✓

b) $L_0 + [0 + 0 + 0] = L_0$ ✓

c) $L_0 + [0 + 1 + 0] = L_0 + 1$ ✗

d) $L_0 + [0 + 0 + 0] = L_0$ ✓

Now let $i = 1$ $j = 1$

$\Rightarrow k = 2$ (main diagonal)

a) $L_0 + [0 + 1 + 0] = L_0 + 1$ ✓

b) $L_0 + [0 + 1 + 0] = L_0 + 1$ ✓

c) $L_0 + [0 + 3 + 0] = L_0 + 3$ ✗

d) $L_0 + [4 + 1 + 0] = L_0 + 5$ ✗

Silly taking some other example we can eliminate opt-(b)

Note:

In general for a toeplitz matrix, optimum no of elements to be stored in 1-D array = $n + (a-1)(2n-a)$

Stack & Queue

Stack:

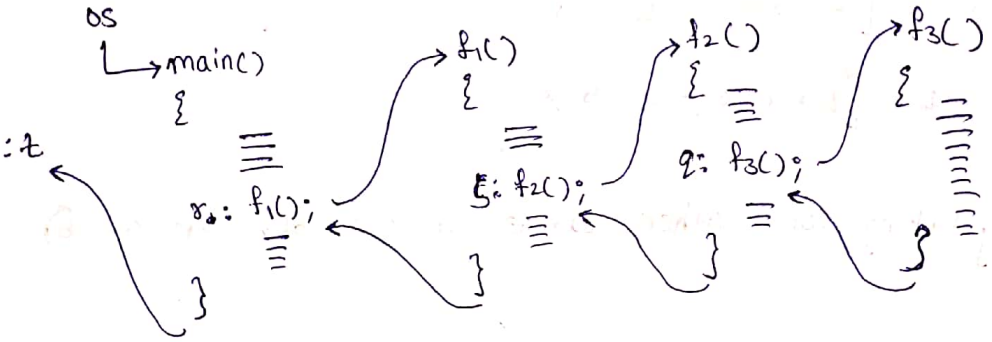
→ Stack is an ordered list in which insertion & deletion can be performed only at one end i.e., at top end.

→ Stack uses Last In First Out.

→ Operations on stack

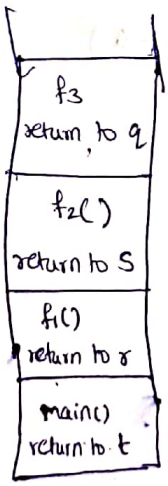
* push : Insertion
* pop : Deletion

Importance of stack:

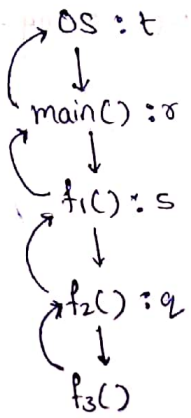


Let 't' return address given by OS to main

(t, r, s, q)



Stack



→ Stack is used to maintain return addresses of procedure calls.

Stack Permutation:

- we can generate stack permutation by inserting 6 keys in the given order but pop off in any order.

Eg: Consider following keys & the stack permutations possible.

1, 2, 3

we print an element after we pop it

1 2 3 : PUSH 1, POP, PUSH 2, POP, PUSH 3, POP

1 3 2 : PUSH 1, POP, PUSH 2, PUSH 3, POP, POP

2 3 1 : PUSH, PUSH, POP, PUSH, POP, POP

2 1 3 : PUSH, PUSH, POP, POP, PUSH, POP

3 1 2 : PUSH, PUSH, PUSH, POP(3), ------- not possible

3 2 1 : PUSH, PUSH, PUSH, POP, POP, POP

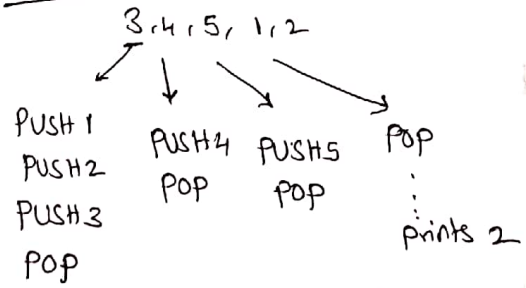
Note:

→ In genral no of permutations possible with n distinct keys = n!

→ no of stack permutation possible with n distinct key where we can push elements in given order and pop off in any order $= \frac{1}{n+1} {}^{2n}C_n$

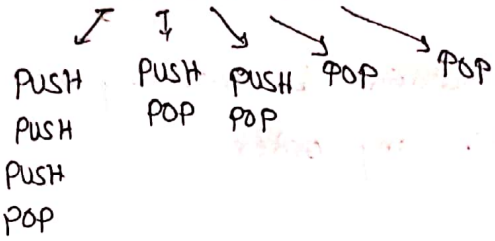
(P/5) keys: 1, 2, 3, 4, 5

opt q:

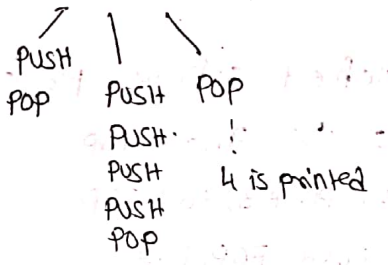


∴ invalid

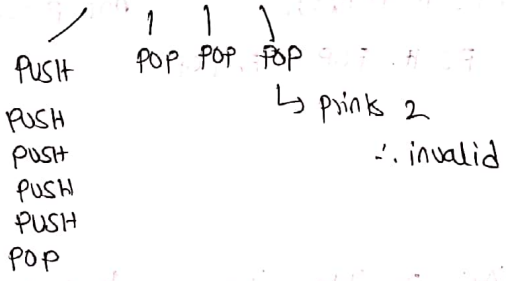
opt b: 3, 4, 5, 2, 1



opt c: 1, 5, 2, 3, 4



opt d: 5 4 3 1 2

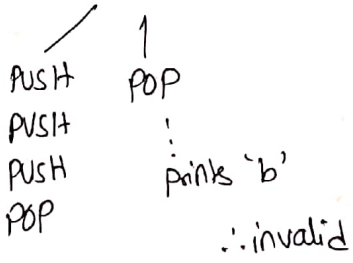


(Q44) A program attempt to generate as many permutations as possible for the string abcd by pushing characters in same order but pop off top character at any time. which of the following cannot be a stack permutation.

a) a b c d b) c b a d c) d e b a d) c a b d

Sol:

opt d: c a b d



∴ invalid

∴ opt (d)

Recursion:

→ Head recursion
→ Tail "
→ Head & tail recursion
→ Excessive recursion
→ Nested "

Head Recursion:

Consider

```
head(int x)
{
    if(x>0)
    {
        head(x-1);
        printf("%d",x);
    }
}
```

(i) Find o/p of head(3)

(ii) find no of function calls for head(n)

(iii) find depth of function calls for head(n)

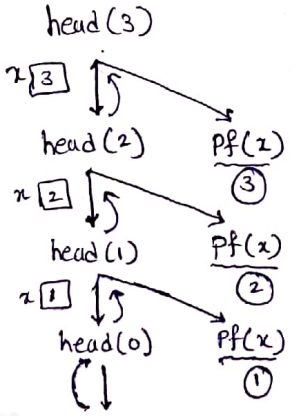
(iv) If T(n) = no of function calls for head(n)
then what is the reccurence equation?

(v) what is time complexity?

(vi) solution to the reccurence eqn of T(n) in (iv)

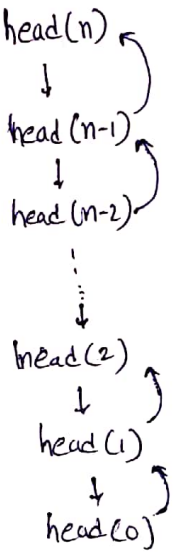
Sol:

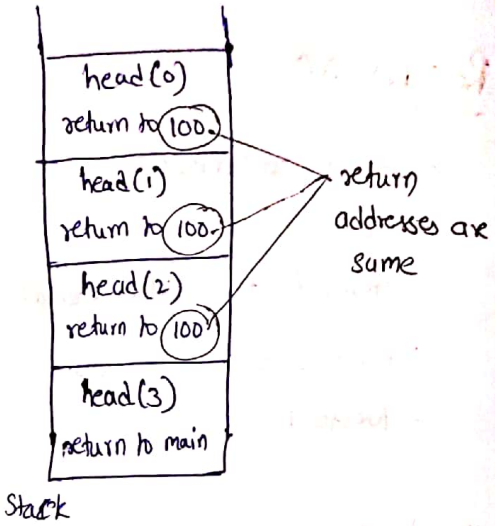
(i)



∴ o/p: 1, 2, 3

(ii)





$\therefore$ no of function calls = $n+1$

(iii) depth of function calls = $n+1$

(iv) $T(n) = 1 + T(n-1)$ if $n > 0$

$= 1$ if $n \leq 0$

(v) time complexity of a recursive function is bounded by no of recursive calls

$\Rightarrow O(n)$

(vi) $T(n) = \begin{cases} 1 + T(n-1) & , \text{if } n > 0 \\ 1 & , \text{if } n \leq 0 \end{cases}$

$T(n) = 1 + T(n-1)$

$= 1 + 1 + T(n-2)$

$= 2 + T(n-2)$

$= 2 + 1 + T(n-3)$

$= 3 + T(n-3)$

$\vdots$

$= n + T(n-n)$

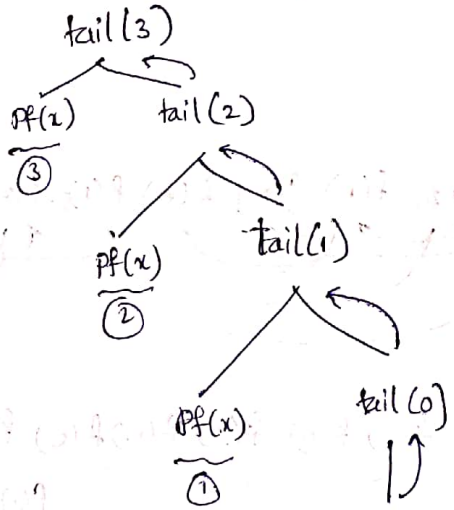
$= n + T(0)$

$\Rightarrow T(n) = n+1$

Tail Recursion:

```
Tail(int x)
{
    if(x>0)
    {
        printf("%d", x);
        Tail(x-1);
    }
}
```

(i) Find o/p of tail(3).



(ii) no of function calls: n+1

(iii) depth of function calls = n+1

(iv) $T(n) = 1 + T(n-1)$ if $n>0$

$= 1$ if $n \leq 0$

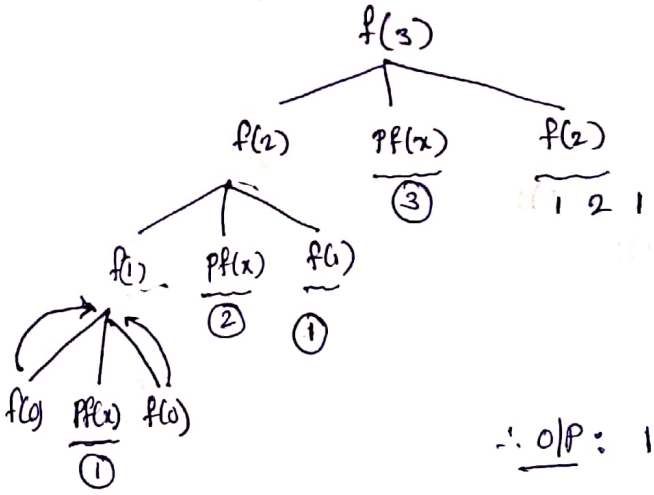
(v) time complexity = $O(n)$

25/09/20

Head - Tail Recursion

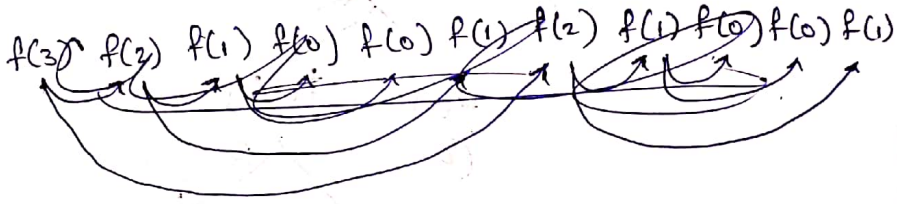
```
f(int x)
{
    if(x>0)
    {
        f(x-1);
        printf("%d", x);
    }
    f(x-1);
}
```

(i) Find o/p of f(3).



∴ o/p: 1 2 1 3 1 2 1

Seq of function calls



f(3) f(2) f(1) f(0) f(0) f(1) f(0) f(0) f(2) f(1) f(0) f(0)
f(1) f(0) f(0)

For n=3, no of function calls = 15

(ii) Find no of function calls for f(n)?

$2^{n+1} - 1$

(iii) Depth of function calls for f(n).

n+1

(iv) Find ~~time~~ time complexity.

time complexity is bounded by no of function calls
i.e., $O(2^n)$

(v) Find space complexity.

we we need to find maximum no of activation records stored in the stack.

∴ when f(0) is executing,

stack may contain n+1 activation records

∴ O(n) is space complexity

(vi) write and solve recurrence relation for no of function calls for f(n).

Sol:

let T(n) be no of function calls in f(n).

$$T(n) = 1 + T(n-1) + T(n-1) \quad \text{if } n>0$$

$$\text{and } T(n) = 1 \quad \text{if } n \le 0$$

$$\Rightarrow T(n) = \begin{cases} 1+2T(n-1) & \text{if } n>0 \\ 1 & \text{if } n \le 0 \end{cases}$$

$$T(n) = 1 + 2T(n-1) \rightsquigarrow 2^{nd} \text{ call}$$

$$= 1 + 2(1 + 2T(n-2))$$

$$= 1 + 2 + 2^2 T(n-2) \rightsquigarrow 3^{rd} \text{ call}$$

$$= 1 + 2 + 2^2(1 + 2T(n-3))$$

$$= 1 + 2 + 2^2 + 2^3 T(n-3) \rightsquigarrow 4^{th} \text{ call}$$

$$\vdots$$

$$= 1 + 2 + 2^2 + \cdots + 2^{k-2} + 2^{k-1} T(n-(k-1)) \rightsquigarrow k^{th} \text{ call}$$

$$= 2^{k-1} - 1 + 2^{k-1} T(n-(k-1))$$

Assuming k^{th} call is base call

$$\Rightarrow n-(k-1) = 0$$

$$\Rightarrow n-k+1 = 0$$

$$\Rightarrow k = n+1$$

$$\Rightarrow T(n) = 2^{n+1-1} - 1 + 2^{n+1-1} T(0)$$

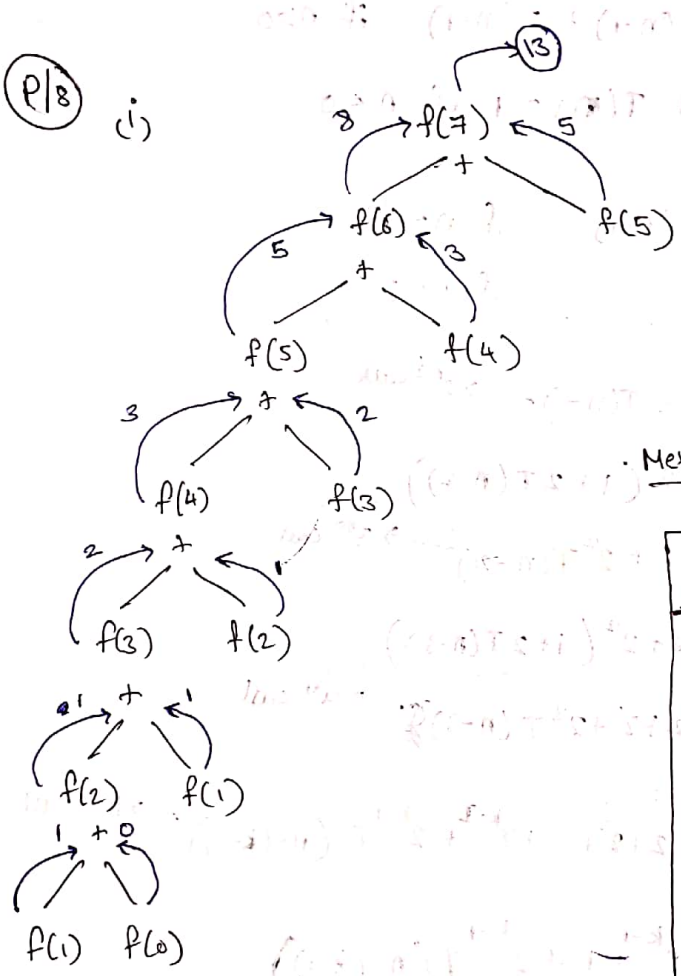
$$= 2^n + 2^n - 1 = 2^{n+1} - 1$$

Excessive Recursion

```
int f(n)
{
    if (n==0 || n==1)
        return 0;
    else
        return f(n-1) + f(n-2);
}
```

P|8

(i)



Method 2:

n	fib(n)
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34

If T(n) is no of function calls in fib(n) then

$$T(n) = 2\,fib(n+1) - 1$$

(ii) $\therefore T(8) = 2\,fib(9) - 1$

$= 2(34) - 1$

$= 67$

Total no of '+' operation for fib(n) is
fib(n+1) - 1

(ii) no of '+' operations in fib(9) = fib(10) - 1
= 55 - 1 = 54

Ackermann's number:

Ackermann's number A(x,y) is defined as

$$A(x,y) = \begin{cases} y+1 & \text{if } x=0 \\ A(x-1,1) & \text{if } y=0 \\ A(x-1, A(x,y-1)) & \text{otherwise} \end{cases}$$

↳ nested recursion

Note:

→ A(0,y) = y+1
→ A(1,y) = y+2
→ A(2,y) = 2y+3
→ A(3,y) = $2^{y+3} - 3$

Eg:

A(2,3) = ?

~~A(2,y) = 2y+3~~

⇒ A(2,3) = A(2-1, A(2,2))
= A(1, A(2,2))
= A(1,7)
= 7+2
= 9
∴ A(2,3) = 9

A(2,y) = 2y+3
⇒ A(2,2) = 4+3 = 7
A(1,y) = y+2

Eg: $A(2,5) = ?$

Sol:

WKT $A(2,y) = 2y+3$

$A(2,5) = 2(5)+3$

$= 13$

Eg: $A(0,3) = ?$

Sol:

$A(0,y) = y+1$

$\Rightarrow A(0,3) = 3+1$

$= 4$

Eg: $A(3,0) = ?$

Sol:

$A(3,y) = 2^{y+3} - 3$

$A(3,0) = 2^{0+3} - 3$

$= 2^3 - 3$

$= 5$

Towers of Hanoi

→ Only one disk could be moved at a time.

→ A larger disk must never be stacked above smaller one

→ One and only one auxillary needle could be used for the intermediate storage of disk

Assume that source has n-1 disks.

step1: Move (n-1) disks from source to source to auxilary.

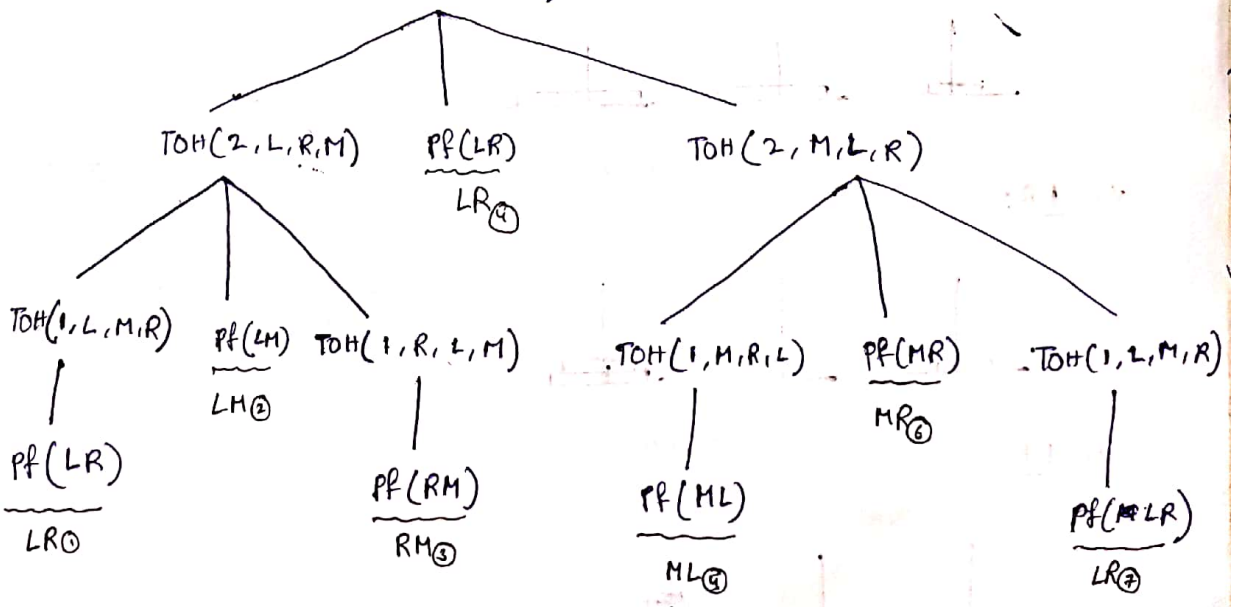
step2: Move one disk from source to destination.

step3: Move (n-1) disks from auxilary to destination

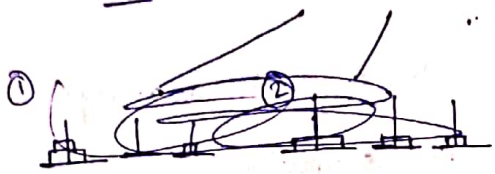
source auxilary Destination
↑ ↑ ↑

```
void TOH (n, L, M, R)
{
    if(n==1)
    {
        printf("LR");
    }
    else
    {
        TOH(n-1, L, R, M);
        printf("LR");
        TOH(n-1, M, L, R);
    }
}
```

Consider TOH(3, L, M, R)



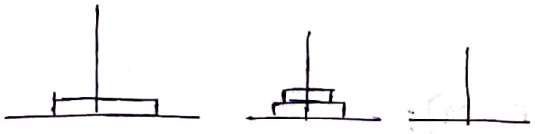
∴ O/P : ~~LLRR~~ LR, LM, RM, LR, ML, MR, LR



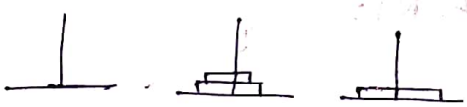
(i) LR :

(ii) LM :

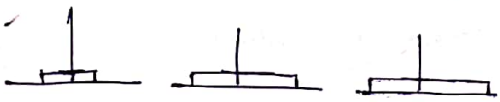
(iii) RM :



(iv) LR :



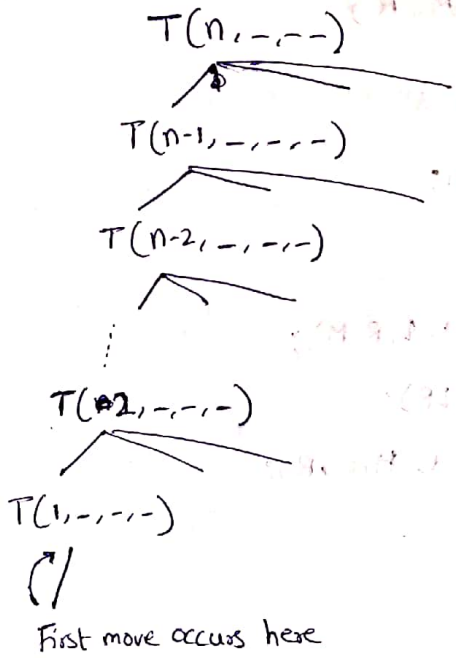
(v) ML :



(vi) MR :

(vii) LR :

Eg: Find no of invocations require for towers of hanoi to make first move.



∴ n invocations. for first move

Eg: Find no of invocations required for last move.

Sol:

we obtain a binary tree of height n-1, for n disks.

last move is done at last invocations

⇒ we need to find no of invocations.

= no of nodes in FBT of height n-1

$= 2^{(n-1)+1} - 1 = 2^n - 1$

Eg: Total no of invocations for 'n' disks $= 2^n - 1$

Eg: Find total no of moves for 'n' disks.

Sol:

Each function invocation has 1 move

⇒ no of moves = no of function calls

$= 2^n - 1$

→ The below program is also possible for towers of hanoi.

```
void TOH(n, L, M, R)
{
    if(n==0)
        return;
    else
    {
        TOH(n-1, L, R, M);
        printf(LR);
        TOH(n-1, M, L, R);
    }
}
```

However, this has more function invocations

i.e., $2^{n+1}-1$

Here total move = $2^n - 1$ (∵ moves must remain same)

- no of invocations of last move = $2^{n+1}-1-1$

  $= 2^{n+1}-2$ (∵ last move occurs in last but one invocation)

no of invocations for first move = n+1

(Q45) which of the following is recurrence eqn of tower of hanoi for computing its optimal execution time.

a) $T(n) = T(n-2)+2$

b) $T(n) = 2T(n-1)+n$

c) $T(n) = 2T(n/2)+1$

d) $T(n) = 2T(n-1)+1$

Sol:

In func TOH(n,L,M,R) we call ~~2 times~~ TOH() twice

$T(n) = 2T(n-1)+1$ → calling func

called func

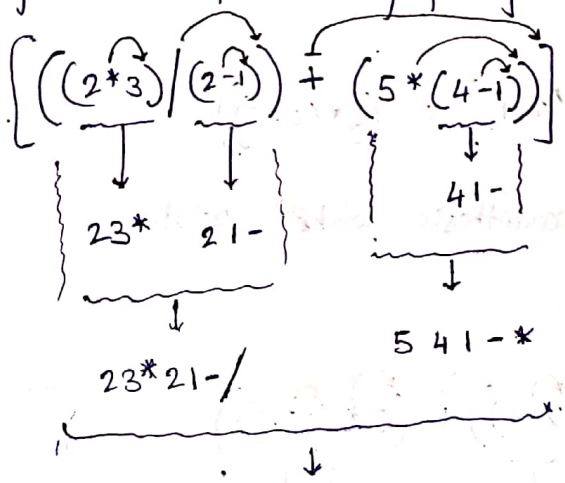
∴ opt (d)

Converting Infix to Postfix

Consider below infix notation & convert it into post fix

$$2*3/(2-1)+5*(4-1)$$

write the given infix expression using paralysis using precedence & associativity



Postfix: 23*21-/541-*+

(Q46) The postfix expression for the infix expression

A+B*(C+D)/F+D*E is

a) AB+CD+*F/D+E*

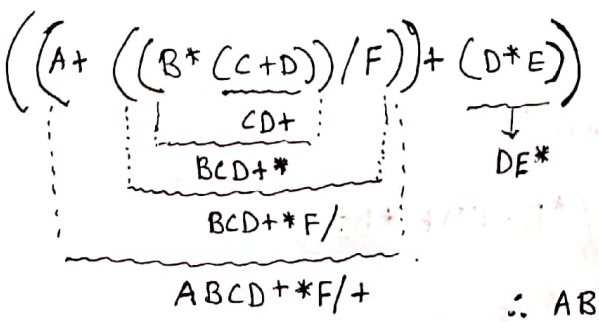
b) ABCD+*F/+DE*+

c) A*B+CD/F*DE++

d) A+*BCD/F*DE++

Sol:

writing given expression using paralysis



∴ ABCD+*F/+DE*+

∴ opt (b)

Converting infix to Prefix

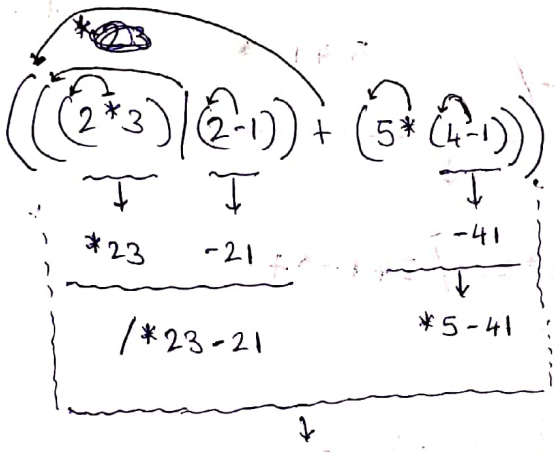
Consider

$2*3/(2-1)+5*(4-1)$

write the exp using paranthesis

$((((2*3)/(2-1))+(5*(4-1))))$

replace left paranthesis with the operator



Eg: $A+B*(C+D)/F+D*E$

$((((A+(((B*(C+D))/F)))+(D*E))))$

$+CD$

$*B+CD$

$/*B+CDF$

$+A/*B+CDF$

∴ final prefix expression is

$++A/*B+CDF*DE$

P|12 $a=b+c*d/e+f\uparrow g\uparrow h-i*j$

↑ has highest precedence and it is right associative

$(a=(((-b+((c*d)/e))+(f\uparrow(g\uparrow h)))-(i*j)))$

$*cd$

$/*cde$

$\uparrow gh$

$\uparrow f\uparrow gh$

$+b/*cde$

$+-b/*cde\uparrow f\uparrow gh$

$=a-+-b/*cde\uparrow f\uparrow gh*ij$

∴ opt (a)

Infix to postfix using stack:

Consider

$2*3/(2-1)+5*(4-1)$

(i) Examine the input sequence. (one character)

(ii) If it is operand then print it and goto step (i)

(iii) If it is opening paranthesis then push it into stack & goto step (i)

(iv) If it is an operator

a) If stack is empty then push operator into stack and goto step (i)

b) If top of the stack is opening paranthesis then push operator into the stack and goto step (i)

c) If ~~top of the stack~~ it has higher priority than the operator on top of the stack, then push operator on to the stack, otherwise pop off operator from the stack and goto step (iv). (If the operator is right associative ~~then~~ and has equal precendence to the operator on top of stack a then push goto step (i))

(v) If it is closed parenthesis then pop off operators from stack until we encounter opening paranthesis and pop the opening paranthesis and discard it.

(vi) After scanning entire string pop out remaining contents of stack

Consider below infix expression

→ 2*3/(2-1)+5*(4-1)

↑

o/p: 2

→ 2 * 3/(2-1)+5*(4-1)

↑

stack is empty.

so push '*'

→ 2 * 3/(2-1)+5*(4-1)

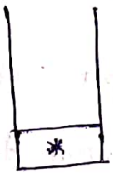
↑

o/p: 2 3

→ 2 * 3/(2-1)+5*(4-1)

↑

/ doesn't have high priority than *



So pop *

o/p: 2 3 *

now stack is empty.

So push /

→ 2*3/(2-1)+5*(4-1)

↑

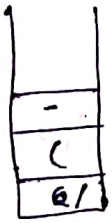
push '(' on to stack

→ 2*3/(2-1)+5*(4-1)
  ↑

o/p: 23*2

→ 2*3/(2-1)+5*(4-1)
    ↑

push '-'



→ 2*3/(2-1)+5*(4-1)
    ↑

o/p: 23*21

→ 2*3/(2-1)+5*(4-1)
      ↑

now pop the stack until we encounter '('

o/p: 23*21-

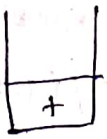
→ 2*3/(2-1)+5*(4-1)
        ↑

+ doesn't have higher precedence than '/'

∴ pop

o/p: 23*21-/

now stack is empty.
So push '+'



→ o/p: 23*21-/5

→ 2+3/(2-1)+5*(4-1)
          ↑

* has higher precedence than +

→ 2*3/(2-1)+5*(4-1)

O/P: 23*21-/54

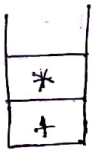
→ 2*3/(2-1)+5*(4-1)

O/P: 23*21-/541

→ 2*3/(2-1)+5*(4-1)

Pop until (is encountered

⇒ O/P: 23*21-/541-



→ Now entire string is scanned.

Now pop out the remaining stack

∴ O/P: 23*21-/541-*+

Eg: Convert below infix expression into postfix

A^B*C/(D*E-F)

Symbol	Stack	postfix
A		A
^	^	A
B	^	A B
*	*	AB^
C	*	AB^C
/	/	AB^C*

Symbol	Stack	postfix
C	/C	AB^C *
D	/C	AB^C*D
*	/C*	AB^C*D
E	/C*	AB^C*DE
-	/C-	AB^C*DE*
F	/C-	AB^C*DE*F
)	/	AB^C*DE*F-

Pop remaining symbols AB^C*DE*F-/

Eg: Convert below infix expression into postfix

8 * (5^4 +2) - 6^2 /9*3

Symbol	stack	postfix
8		8
*	*	8
(	*(	8
5	*(	85
^	*(^	85
4	*(^	854
+	*(+	854^-
2	*(+	854^2
)	*	854^2+
-	-	854^2+*
6	-	854^2+*6
^	-^	854^2+*6
2	-^	854^2+*62
/	-/	854^2+*62^
9	-/	854^2+*62^9
*	-*	854^2+*62^9/
3	-*	854^2+*6^29/3

Pop remaining symbols 854^2+*6^29/3*-

∴ postfix : 854^2+*62^9/3*-

Infix to prefix using stack:

(i) Reverse the input string.

(ii) Examine the input sequence (one character).

(iii) If it is an operand then o/p it and goto step (ii).

(iv) If it is ~~opening para~~ closing parenthesis then push it onto stack and goto step (ii)

(v) If it is an operator

a) If stack is empty push operator into stack & goto step (ii)

b) If top of stack is having closing paranthesis then push operator into stack

c) If the operator has higher precedence than the operator on top of the stack, then push it.

If it has lower precedence then pop the operator and goto step (v)

If the precedence is same and operator is ~~left~~ right associative then pop the operator & goto step (v). If ~~the~~ it is left ~~right~~ associative then push it onto stack and goto step (ii)

(vi) If it is opening paranthesis then pop off operator until we encounter a closing paranthesis and pop it & discard it.

(vii) If there is no more i/p then pop out the remaining symbols from stack.

(viii) Reverse the o/p

Eg: Convert below infix into ~~postfix~~ prefix

$$2*3/(2-1)+5*(4-1)$$

Sol:

Reverse the i/p string

$$)1-4(*5+)1-2(/3*2$$

Symbol	stack	Prefix
)	)	
1	)	1
-	)-	1
4	)-	14
(		14-
*	*	14-
5	*	14-5
+	+	14-5*
)	+)	14-5*
1	+)	14-5*1
-	+)-	14-5*1
2	+)-	14-5*12
(	+	14-5*12-
/	+/	14-5*12-
3	+/	14-5*12-3

Symbol	stack	Prefix
*	+/*	14-5*12-3
2	+/*	14-5*12-32
pop out remaining elements		14-5*12-32*/+

Reverse the o/p

Prefix: +/*23-21*5-41

Postfix evaluation using stack

(i) Scan the i/p sequence. (one character)

(ii) If it is an operand then push it.

(iii) If it is an operator, then apply two pop operation, where the first pop operation element is assigned to the the operator as a right child and the second pop operand is assigened to operator as a left child

~~Now push the result onto stack.~~

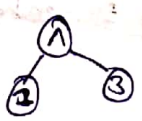
(iv) ~~Without creating tree~~ push the result from step(iii) into stack.

(v) goto step(i)

Eg: Evaluate below postfix expression using stack

823^/23*+51*-

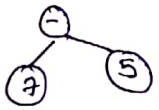
Symbol	stack
8	8
2	82
3	823
^	88
/	1
2	12



= 2^3 = 8

= 8/8 = 1

Symbol	Stack
3	123
*	16
+	7
5	75
1	751
*	75
-	2



≅ 7-5

∴ Final result is obtained in the stack

i.e. 2

Prefix evaluation using stack

(i) Reverse the prefix expression

(ii) Scan the expression (one character)

(iii) If it is an operand push it into stack

(vi) If it is an operator then perform two pops operation

(i) pop operand from stack and assign it as left child of the operator

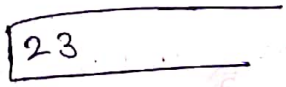
(ii) pop operand from stack and assign it as right child of the operator.

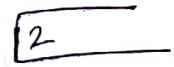
(iii) Evaluate the result & push it onto stack

(v) goto step (ii)

Expression tree from postfix

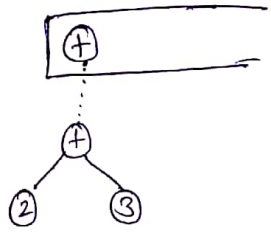
Ex: 23+6*4*



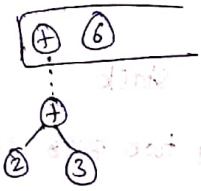
+: pop (3) make it left child of ⊕ 

pop (2) make it right child of ⊕ 

Now push the address of node ⊕ into the stack

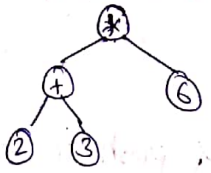


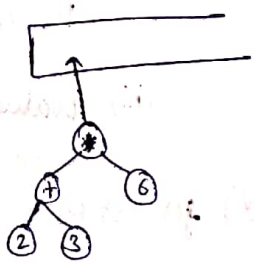
6: Push onto stack

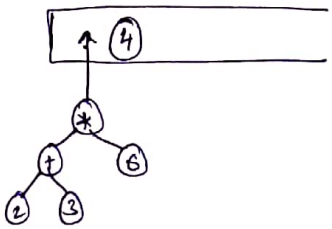


*: pop (6) ... make it right child

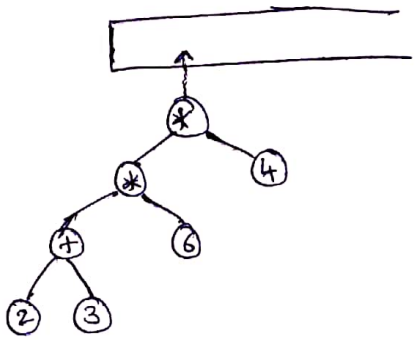
pop (⊕) ... make it left child



Now push address of ⊛ onto stack 

4: 

(8) *:



Note:

→ Inorder traversal on expression tree gives infix expression.

→ Preorder traversal on expression tree gives prefix expression

→ Postorder traversal on expression tree gives postfix expression.

Perform preorder traversal on above expression tree.

⇒ prefix expression: * * + 2 3 6 4

Eg: Evaluate below prefix expression using stack.

* * + 2 3 6 4

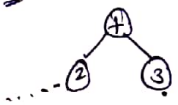
Sol:

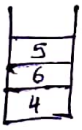
Reverse the expression

4 6 3 2 + * *

After reading 4 6 3 2:

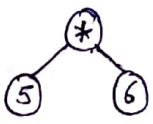
+:

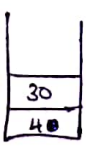




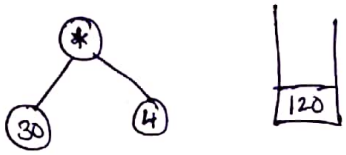
1st popped operand is left child

*:





*:



∴ Ans: 120

Expression tree using prefix notation:

Eg: Build expression tree for below prefix expression

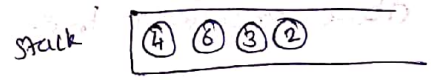
~~4 6 3 2 + * *~~ * * + 2 3 6 4

Sol:

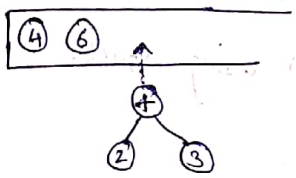
Reverse, the expression

4 6 3 2 + * *

After reading 4, 6, 3, 2:



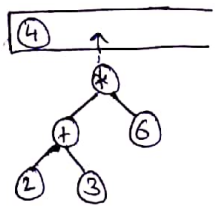
+:



1st popped node is added as left child

2nd popped node is added as right child

*:



*: