Data Struc. and Algo (DSA)

Mam Namra Absar — D1

Sir Mohsin Abbas:

(5:00) — Tuesday: D2 — A202

(5:00) — Thursday: D2 — A006

Tayyab Hassan Tarar
L1F20BSCS0097

Mam Namra Absar — D1

Room#3, block-C

beside backside of Gym faculty

Revision

Lecture:

i) Revision of OOP. ii) stack and queo
FILO

ii)

* Trees
* Time & space complexity

Stack:

* Interface is abstract class, where there is pure V.F in all classes.

TO DOS
* object slicing
* Interface (all func. are virtual)
* pointers arrays
  → double link list
* Linklist * Struct
  → single and cycle/circle
* Stack & heap mem.
*

Lecture

* Time Unit ; → constant time for code compillation.
  Let, time unit = 1
  int loop = 5 x 1 = 5
  Total time unit = 5+1 = 6
* Complexity: O (N+M+....)
  Big O
* Constant function has complexity O(1)

* Diff. b/w array & list.
* Time complexity.
* Linier & quadratic
* Complexityies of sortings (Binary, bubble, selection)
* Log Complexities

* **Big O Notation** / (Upper boundry of a code) / Asymptotic Notation.
  → $O(n) \Rightarrow f(n) = n$; $O(n^2) \Rightarrow f(n) = n^2$
* **Cases of Time Complexity:** for algorithms:-

i) Best case	ii) Average case	iii) Worst case
• TC can be taken in first some iterations.		• have to run for last element.

• Where there is no loop, TC will be 1.

∴ $O(g(n))$ → Asymptotic Not.
n, n^2, n^3, etc.

→ Asymptotic Notations:

upper bound
- $O(g(n))$, c : Big OOH, eg. $n+2 = O(n)$ (c=3) → c>2 ∴ Graph
- $\Omega(g(n))$, c : OMEGA, eg. $n+2 = O(n)$ (c=1) → c<2 always, n's cond.

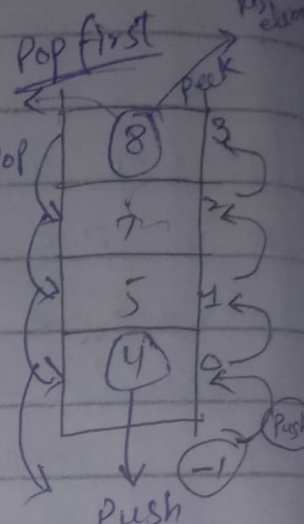
both upper & lower
- $\Theta(g(n))$: Theta, both c's, c_1 & c_2

↳ ∴ Also a exact bound.

---

* Stack DataStructure:

- Push insert element on index
- Pop desert element from top/last index.
- LIFO methodology: last Input → first output.
- Peek: only shows last element from stack.



H-W: string ~~char~~ arr from user and push it in stack, then find whether it is palindrome or not? + complexities:

C-block at 11:00 submission

Infix:	Prefix	Postfix
• 5+4 ← operator eg.	• +54 eg.	• 54+ eg.
• symbol in mid	• before symbol	• After symbol
• It requires parethesis		

- Convert Infix into prefix & postfix.

i) (A+B)*C → (step) ii) +AB*C

iii) *+ABC

Remember about precidence: BDMAS

ii) A*B+(*D → *AB+C*D

+*AB*CD

++*ABDC

∴ a + b (+ : Operator, a, b : Operands)

∴ 1 operator with 2 operands

∴ Binary operator: only +

Tayyab Hassan Tarar

iii) A+B+C+D → =+AB+C+D

= +AB + +CD

= ++AB+CD Ans.

iv) (A+B) *C − (D−E) * (F+G)

+AB *C − −DE * +FG

+−+ABDEFG *C

= *+ABC − *−DE +FG

= −*+ABC *−DE +FG

v) B+D/C+A*C/F

= BB+/DC+A*/CF = B+ +/DCA * /CF

∴ H.W

Convert pre, postfixes. by internet.

• Conversions of operators

~~* Applications of stacks:-~~

*Solve Applications:-

i) +2 *385 ∴ 5×3 = 15

= +2 15 space ∴ Add 2 & 15

= 17

ii) *+235 ∴ +2 3 = 5

= *55 ∴ *55 = 25

= 25 Two operands.

iii) 235*+ ∴ 3 5* = 15

2 15 +

17

∴ No parenthesis needed here either.

∴ 195

divide 9 by 5

∴ Always solve two operands with a operator.

→ Infix to Prefix conversion

i) (((2+(3*(5−3)))*5)

∴ Make pa

=(2+(3*2))*5

* => Expression Making:- ~~prefix~~ Infix to ~~Infix~~ Prefix

(i) (((2 + (3 * (5 - 3))) * 5))

first

∴ Integer Division

- Ignore floating value.

(18) 97
↳ only.

= * + 2 * 3 - 5 3 5

= *

(ii) (((A+B) * C) - ((D+E) / F))

= A B+ C* D E+ F/-

= AB+ C* DE+ F/- Ans.

=> Infix to Postfix

∴ stack : < empty >
output : [AB+CE-*FG+/]

∴ → Parenthesis must be correct.

Push & Pop.

∴ Operands will go in output

- Operators in stack
- Closing bracket ~~etc~~ puts operator in output.

H.W

Infix to prefix cod

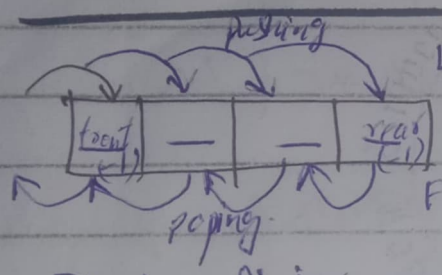
∴ Book at page 40

Solving the Mazes using Stacks:- Rat's Problem

R D U L

- Sequence: Right → Down (if not) → Up (if not) → Left (if not)
- Consider 1 as true and 0 as false.
- If came zero, there is a blockage and we put -1 at that place
- Right has higher precidence, stack always prefer right move to reach at destination.

Queue data Structure:-



LIFO (In-queue)

FIFO (De-queue)

- It is linier queue.
- It works on FIFO pattern (First In, first Out)

⟹

* Enqueue: Add values in queue (push)
  dequeue: remove values from queues (Pop)
* Non-Circular queue implementation. & Circular queue.
* Circular
  - When front & rare are at have same values Stack will be empty.

- Unary oper.
- Circular queue
- Codes of queue

A-

Linked-List:-

- Node is not a datatype.
- Left side pointer, Right side Memory
- datatypes of both sides must be same.
- Head = NULLPTR & Tail = Nullptr → link list is empty.
- while(1){} → infinite loop
- return NULL for integer values & NULLPTR for pointer.

=> Infix to prefix Method by (Inorder) inverse/recursive Method. Preforder

=> A-B / (C*D ^ E) starts.

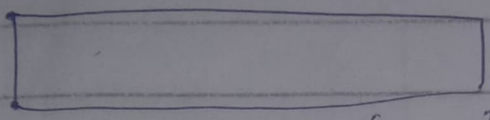
Inverse: starts from End:) E ^ D * C (/ B - A

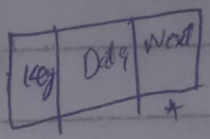
Input	Stack	Output (Only operands)
)	)	
E	)	E
^	)^	E
D	)^	ED
*	)^*	ED
C	)^*	EDC
(	)^*	EDC
/	)^*/	EDC
B	)^*/	EDCB
-	)^*/-	EDCB
A	)^*/-	EDCBA
: One by one from inverse	: Only operators	: Only operands

Example :— A *(B ^ C - E) + F

Reverse : E +) E - C ^ B (* A

~~Imp~~ : Whenever opening bracket came, pop the stack till closing bracket.

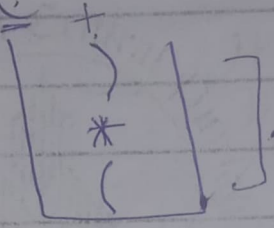




infix to postfix

* / , + - , ^

(A * B) + C

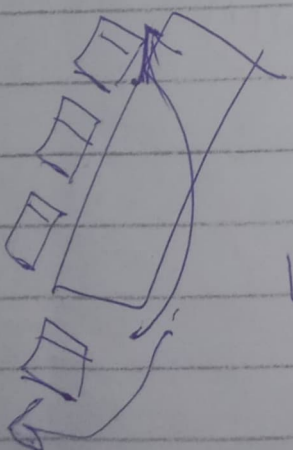


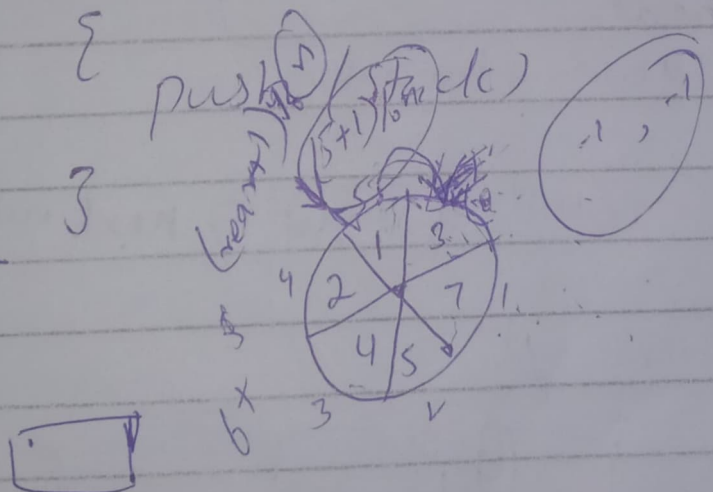
AB* C+

if (alphabet / 88 number)

{

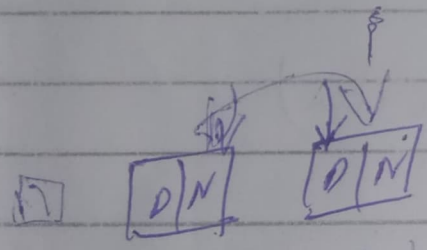
}

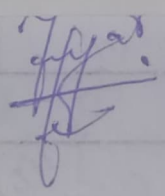


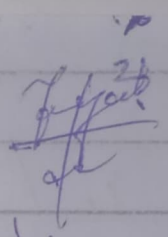


- Prefix to Postfix: (Inverse) → operand1 + operand2 + operator
- Postfix to Prefix: Operator + operand2 + operand1
- Postfix to Infix: operand2 + operator + operand1
- Prefix to Infix: (Inverse) operand1 + operator + operand2
- Infix to Postfix:
  - Use precidences: + − / * ^
  -) bracket will pop all things. → pop
  - precidence order matters. e.g. / −
- Infix to ~~Prefix~~ Prefix: 1- Reverse 2) Postfix 3) reverse
  - No two operators remains of same precidence.

Symbol	Operator stack	Postfix







```
Node* temp = head;
head = head->next;
delete temp;
```

240

```
Node* temp = head;

while (1)
{
  if (temp->next == tail)
  {Node* temp2 = temp->next;
   tail = temp;
  }
  delete temp2;
  tail->next = nullptr;
```

temp = temp2

Recursion:

- If a function calls itself within a body ; called recursion

e.g.
```
void solve (int n)
{
    if (      ) // base case
    {
        return ___;   // base condition
    }
    else
    {
        solve (n);   // function calls itself
        return larger problem;
    }
}
```

- Big problem solutions depends on smaller problem, use recursion:

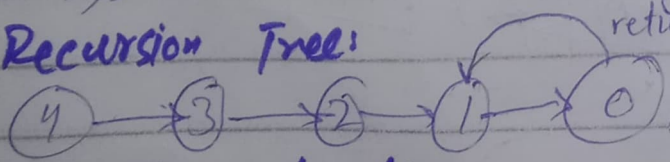
e.g. $2^n = 2 \times 2^{n-1}$ $\Rightarrow$ $f(n) = 2 \times f(n-1)$;

smaller problem — larger problem — larger problem.

recursive relation — larger problem

- The stoping condition in recursion is Base case
  - return typing is mandatory in base case.
- without base class, there would be segmentation fault. (stack overflow)
- so, Base class is mandatory (terminating condition)

Recursion Tree:



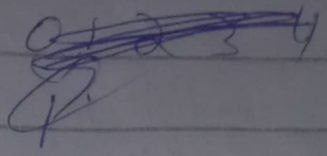
Recursive function:

1. Base case (return mandatory) ; otherwise stack overflow.
2. Recursive relation (return larger problem)

Processing (-, +, other condition) ... : Optional

Order:

```
void function ()
{
    Base case
    Processing
    Recursive relation
}
```



1123

Fibonacci series:

0, 1, 1, 2, 3, 5, 8, ...

5875

i) R.F ● $f(n) = f(n-1) + f(n-2)$

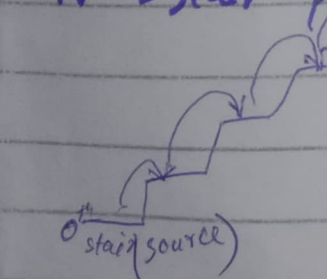
- Sum of previous two elements of series

  $0+1=1$, $1+1=2$, $1+2=3$, $2+3=5$, ...

ii) Base case:

```
if (n==0)          & if (n==1)
return 0;            return 1;
```

N^th - Stair problem:



∴ condition: 1_stair or 2_stair

- R.C = ans = $f(n-1) + f(n-2)$
- B.C = `if (n<0) return 0;` ; `if (n==0) return 1;`

Say digits

e.g. 1 2 3

output: One Two Three

```
string arr[10] = {"Zero", "One", "Two", ... "Nine"};
int digit = number % 10
number = number/10;
```

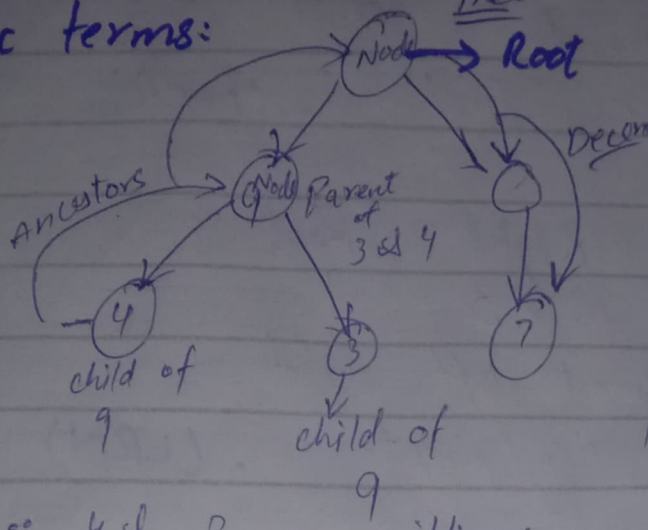
T.C:

```
if (n==0)                    // Base case
    return;
int digit = n % 10           // Processing
    n = n/10;
    cout << arr[digit] << " ";
sayDigit(n, arr);            // recursive call
```

Binary Trees:

- Linier data structure / Non-cycle Tree

- Basic terms:



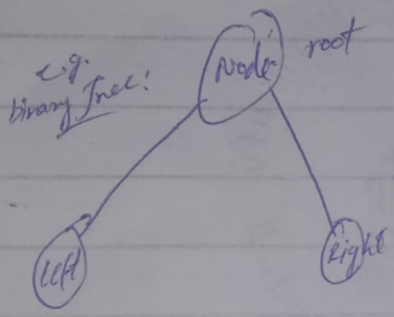
∴ 4 & 3 are siblings

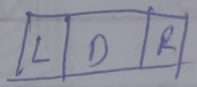
∴ 4, 3 & 7 are leafs.

→ Terms:

1) Node
2) Root (Top node)
3) Children
4) Parent
5) Siblings (same parent)
6) Ancestor (neeche se upar)
7) Decendent (upar se neeche)
8) Leaf (No child)

Node: e.g. Binary Tree:

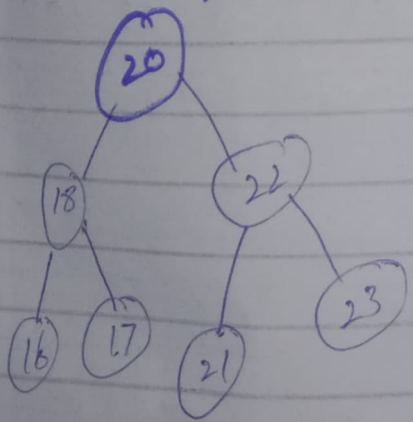





```
Node {
int data;
Node * left;
Node * right;
};
```

- **Binary Tree:** . only two childs. . any one child may be NULL.

→ **Binary Search Tree: (BST)**



In BST,

∴ . In left childs, there must be number/data less than upper/parent node

. In Right childs, number greater than leaf/parent node.

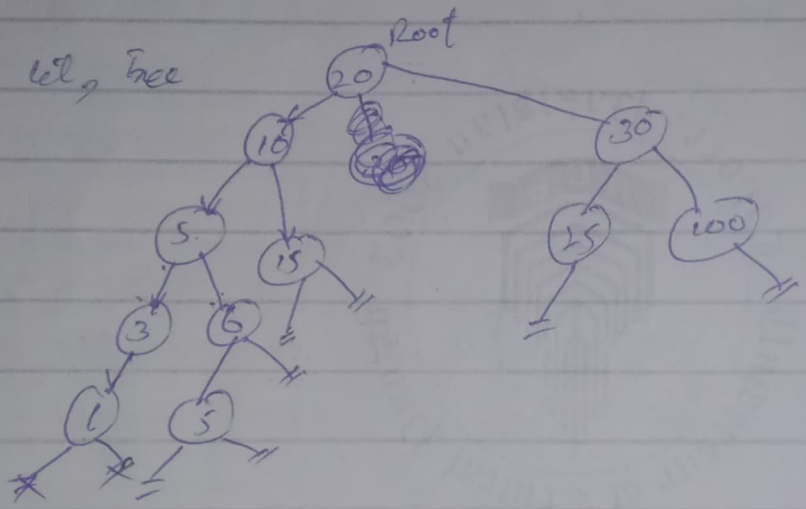
- **Inorder:** start from last left child node. (LNR) (Left Root Right)

16 18 17 20 21 22 23

Left last most — leaf — left most from right.

- **Pre-order:** Root Left Right
  1 2 4 5 3 6 7
  - starts from top to bottom
  - left side traverse first . (NLR)
    ↑ root
- **Post-order:** Left Right Root
  4 5 2 6 7 3 1
  - start from last nodes last most left child
  - first write childs then parent. . (LRN)

e.g. tree



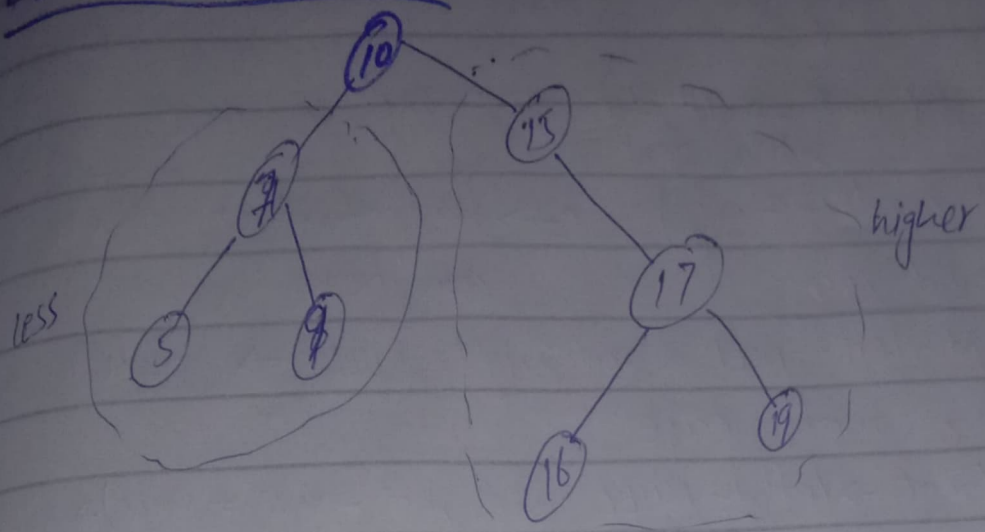
Pre-Order. (NLR) ∴ first Node then Left then Right.
display node.

→ 20 10 5 3 1 6 5 15 30 25 100

∴ dry run of recursive

RXJS, Node JS, react → w3schools

BST in depth:



∴ In BST, Left nodes have node→data < root
& Right nodes have node→data > root

- Consider each node in BST

→ Conditions while inserting data in BST:

i) if (Root == NULL)
// create new node

ii) if (data > root→data)
// insert at right
e.g. 7 > 5
(5 ← root data)
{
root→right = createBST(root→right, data);
}

iii) if (data < root→data) {
root→left = createBST(root→left, data);

- Time complexity of insertion O(log n)

- **Search a data/node**

Condition: i) → (X) NULL → return false
ii) if (root→data > d) → return left
iii) if (root→data < d) → return Right
by bool datatype

→ Basic BST operations
i) Preorder
ii) Inorder • Postorder
- search a node
- Deletion of nodes
- min (from left); most, sca
- max (from right)
- deletion of Node

deletions of key/data from Tree.

Conditions: if(root == NULL) return root // Base case

1) • if (root->data == key) → has further there condition.

→ • One child cases:

i) if(root->left == NULL && root->right == NULL) → delete root

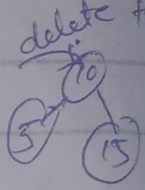
• root case

ii) if(root->left != NULL && root->right == NULL) → left

• by temp, delete left leaf

iii) if(root->left == NULL && root->right != NULL) → right le

• by temp, delete right leaf.



→ ii) Two Child cases: ?

∴ it would become single child or leaf case.

2) else if(root->data > key) => root->left = deleteF(root->left, key

3) else if(root->data < key) => root->right = deleteF(root->right, key)

Tree: left side → points, right side → memory

→ Deletion in BST:

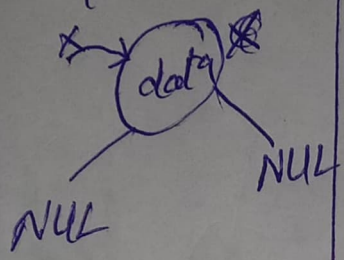
Basic Cases ①

i) if (root → data == x) → . . . . .
ii) if (root → data > x) → left part del.
iii) if (root → data < x) → right part dele.

→ deletion has four cases:

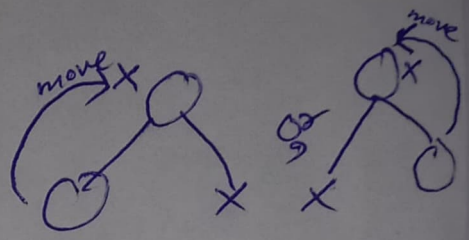
i) 0 child

- delete Node
  return NULL;

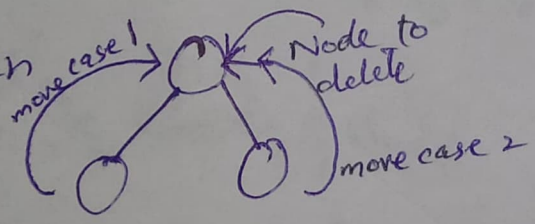


ii) 1-child

- Left child
  → temp; root → left
  delete root;
  return temp
- Right child
  → temp; root → right.
  delete root
  return temp



iii) 2-children



or

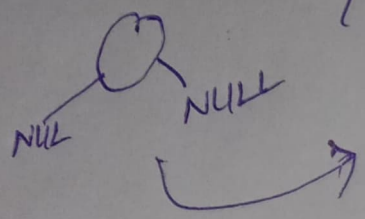
- Now, there are two case to move the value at deleted Node.

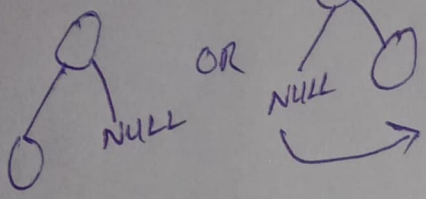
i) Maximum value from left side
- Use the Node → delete

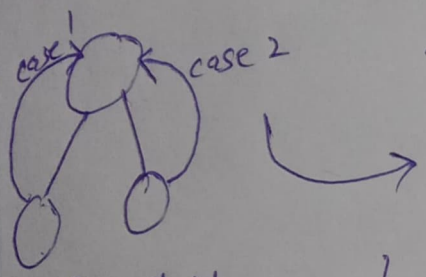
ii) minimum value from right side
- use the Node → delete

Code:

```
Node * deleteTree(Node * root, int key)
{
    if (root == NULL)   // Base case
        return root;

    if (root->data == key)   // Data of key exists
    {
        // 0 child case
        if (root->left == NULL, root->right == NULL)
            delete root;
            return NULL;

        // 1 child case
        if (root->left != NULL && root->right == NULL)
            Node * temp = root->left;
            delete root;
            return temp;

        // 2 child case
    }
    if (root->data > key)   // Left child case
    {
        root->left = deleteTree(root->left, key)
        return root;
    }
    if (root->data < key)   // Right child case
    {
        root->right = deleteTree(root->right, key)
        return root;
    }
}
```



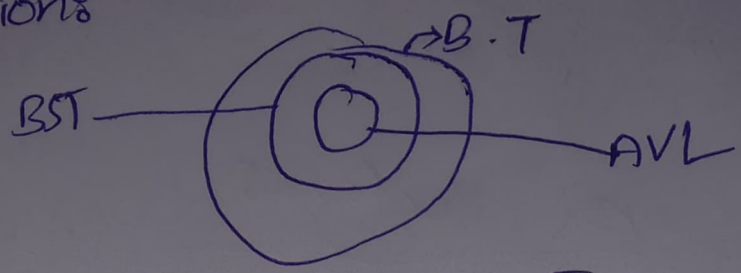




∴ In left side, maximum value.
∴ In right side, minimum value.

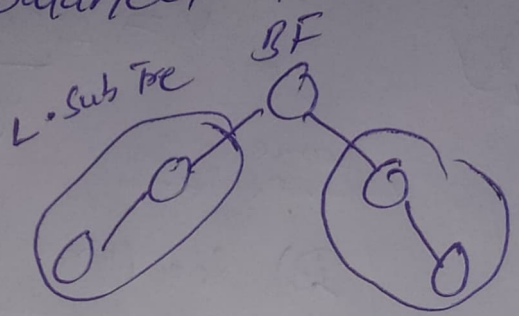
→ AVL Tree

Introduction:



- BST converts to AVL Tree.

→ To find balanced Tree:



B.F
∴ = 2 − 2 = 0 ✓

- Find balanced factor:

$$B.F = (\text{Height of } L.ST - \text{Height of } R.ST)$$

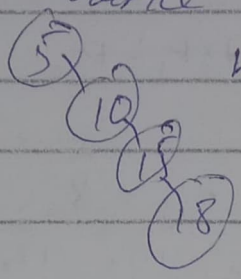
→ Balanced factor (B.F) only arrives if answer is $\boxed{-1, 0, +1}$

- Balanced Tree is actually called as AVT Tree.

AVL :-

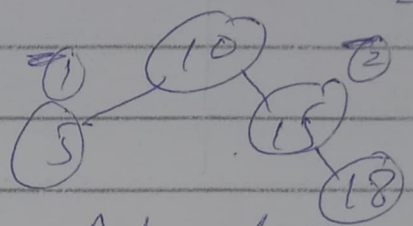
O(n)

- How to balance AVL Tree :-

e.g. 

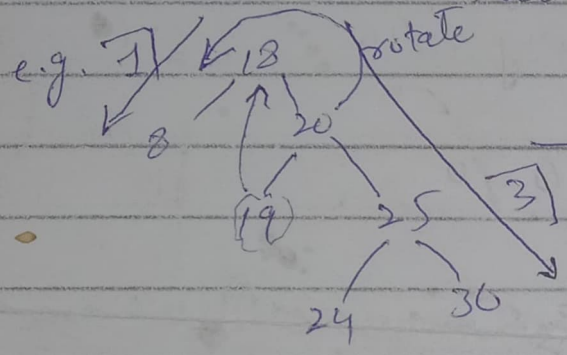
h=3 → Not balanced.

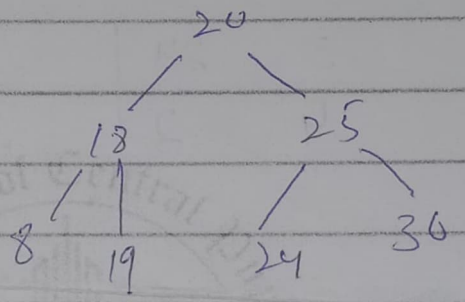
2-1=1



Balanced Tree -

- These are balanced by rotations:

e.g. 

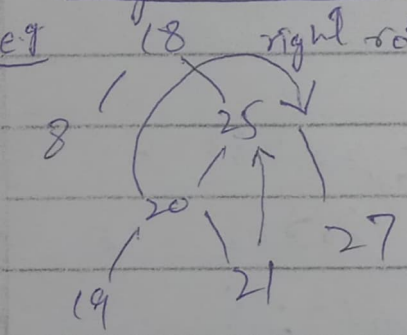


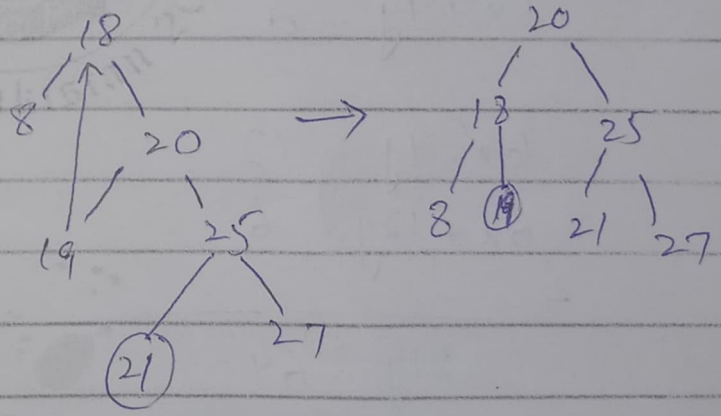
Balanced

B.F = 3-1 = 2 N.B ✗

B.F = 2-2 = 0 ✓

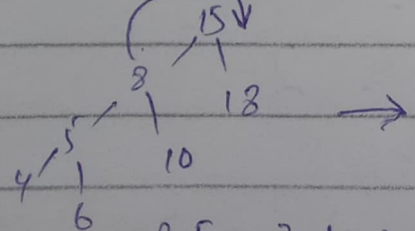
- Right - Left case:

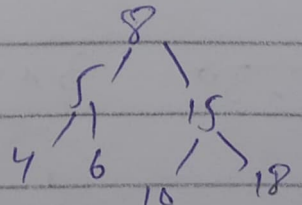
e.g 



N. Balalqceel

- Left - Left case:-





B.F = 3-1 = 2 X

B.F = 2-2 = 0 ✓

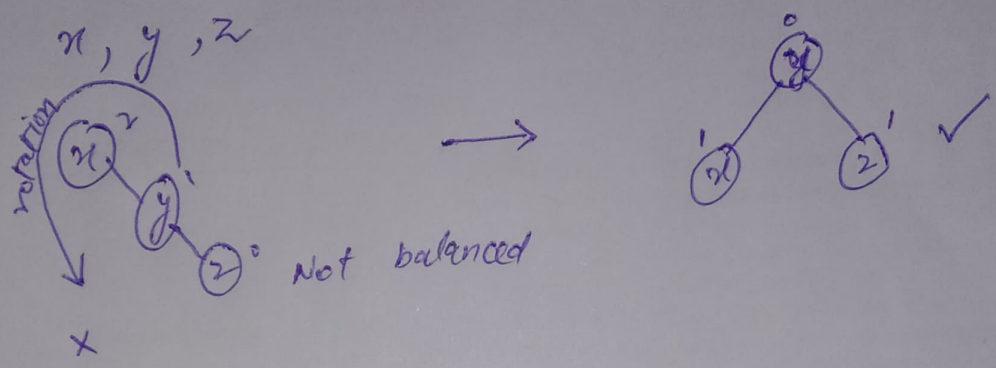
AVL Tree

- It is a BST
- Height of left subtree — height of right subtree
  $= \{-1, 0, 1\}$
  ↳ called Balance factor.

→ three conditions of insertion:

let x, y, z

⇒1)



- ~~first insert right node.~~
- right Node > left node

⇒2)

OR

TO DO:
- List Vs Array
- Unordered Map

→ Red - Black Tree :—

- It is also a balancing Tree

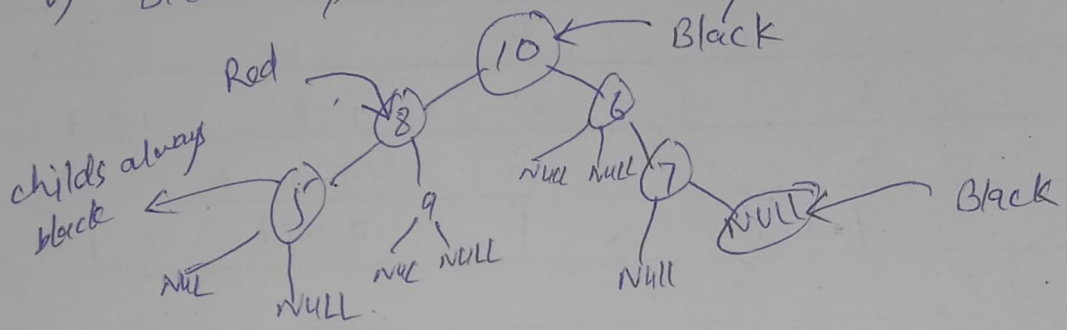
→ Conditions-

i) should be BST ii) Root always Black

iii) NULL nodes should be black.

iv) Red parent have ~~black~~ only black child.

v) Black parent have black/red child.



vi) Blacks should be of same number of all black nodes.

→ Red - Black Tree is almost Balanced Tree -

→ Hash Table :—

Red - Black Tree:

- Average case $O(\log_2 n)$ for searching.
  - for left & right BST sub tree
- Worst case & also $O(n)$ — for series in Right S.T.

- AVL needs many rotations but RBT don't.
  - Maximum two rotations.
  - Use Red — Black colors.
  - Searching is faster in AVL but deletion is faster in RBT

Properties

i) Self balancing
ii) Every Node is either Black or Red.
iii) Root is always Black
iv) Every Leaf which is NULL is Black.
v) If Node is Red, then its children are Black.
vi) Any path from a Node to any of its descendant NULL node has same no. of Black nodes.

→ Hash Tables: . Provide O(1) on:

- Insertion takes O(1)
- ~~Insertion~~ Search takes O(1)
- Deletion takes O(1)

→ Hash function:

. maps big number or integer to small integer that can be used as "index" in hash table.

→ Requirement from hash function. $h(x)$

- Efficiently computeable.
- Should uniformly distribute the keys.
- minimize collisions.

→ Collision . Two keys resulting in same value/index

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

let, $x = 9864567654 \bmod 7 = (4)$

and other x

$x = 9854854542 \bmod 7 = (4)$. same index (collision problem)

→ Solutions for Collision Handling

1) Chaining: (separate)

2) Open Addressing
   - i) Linier Probing
   - ii) Quadratic Probing
   - iii) Double hashing

---

① ★ Separing chaining.

Let Hash function $h(x) = x \cdot \text{mod size}$ (x → key)

keys: 50, 700, 76, 85, 92, 73, 101

indexes

* 50 mod 7 = 1 → place at 1

700 mod 7 = 0

76 mod 7 = 6

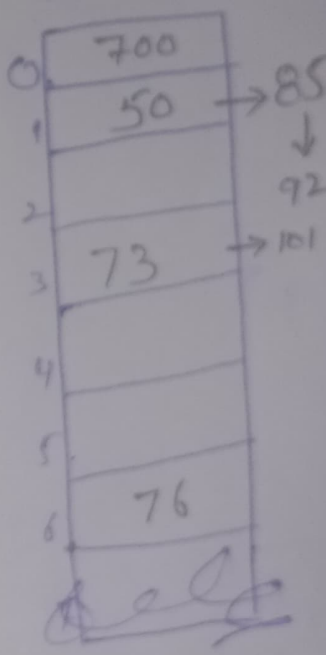
85 mod 7 = 1 → collision occurs

92 mod 7 = 1

73 mod 7 = 3

101 mod 7 = 3

Total size 7



Hash Table

∴ • At index, where collision occurs, it makes the Linked List to add other values at that index.

• So, it actually makes chains

Advantages of chaining:

1) Simple to implement
2) Hash table never fills-up, we can always add more elements to chain.
3) Mostly used, when data entries are unknown, can add frequently keys at a single index.

Disadvantage of chaining:

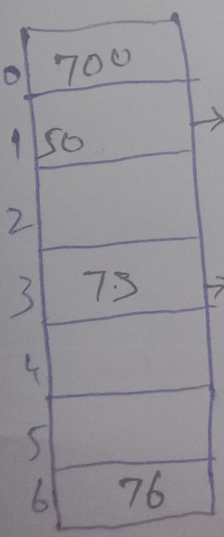
1) Cache performance is not good, as in chaing keys are stored used Linked List.
2) Wastages of space.
3) In long chain, seach time can become O(n) in worst case.
4) Uses extra ~~links~~ spaces for links.

Complexity of chaing:

→ n = number of keys stored in table

m = number of slots in table

α = Average keys / Load factor = $\frac{n}{m}$



- Expected time to insert / search / delete:

$O(1 + \text{Load factor}(\alpha))$

∴ So; L.F = $\frac{7}{7} = 1$ ⇒ $O(1+1) = O(2)$

2★ Open Addressing :-

- 2nd method of resolving collision in hashing.
- All items (keys) are stored in table itself.
- because, size of table ≥ No. of keys

  ∴ No need to insert values outside the hash table (No LL needed)

- Hash function specifies order of slots to probe (try) for a key (for insert / search / delete), not just one slot.

→ Now, hash collision is resolved by Probing :-

(Types)

1- Linear Probing 2) Quadratic Probing 3) Double Hashing

→ (1) Linear Probing :-

→ Hash (index) function:

$h_i(x) = (\text{Hash}(X) + \text{index}) \% \text{ Hash Table Size}$

→ • if $h_0 = (\text{Hash}(\overrightarrow{x}) + 0) \%$ HashTableSize is full, we try h_1 (Next index)

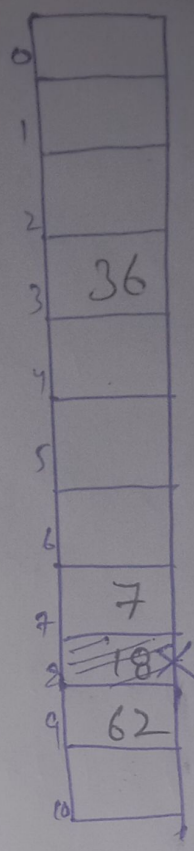
(key)

If $h_1 = (\text{Hash}(X) + 1) \%$ Table full, try for h_2

and so on....

Examples of Linear Probing:-

Let, keys = 7, 36, 18, 62 (total 4 keys)



∴ Hash table > keys
11 > 4

* Insertions:

$h_0(7) = 7 \bmod 11 = 7$

$h_0(36) = 36 \bmod 11 = 3$

$h_0(18) = 18 \bmod 11 = 7$ → collision occurs

↳ $h_1(18) = (18+1) \bmod 11 = 08$ → Next index

$h_0(62) = 62 \bmod 11 = 7$ → Once again collision

↳ $h_1(62) = (62+1) \bmod 11 = 8$ → Already occupied

↳ $h_2(62) = (62+2) \bmod 11 = 9$ ✓

* Delete (18)

2- Quadratic Probing:-

Same, but the index is square of index.

- $h_i = (\text{Hash}(\text{key}) + i^2) \% \text{ hashtablesize}$

if

$h_0 = (\text{Hash}(\text{key}) + i^2) \%$ hashtable full, try for h_1

i and so on.

∴ formula only applies in case of collision

↳

Example:

let, keys = 7, 36, 18, 62

insert (36):

$h_0(36) = (36 \bmod 11) = 3$

$h_0(7) = 7 \bmod 11 = 7$

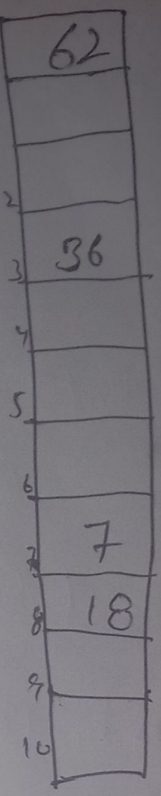
$h_0(18) = 18 \bmod 11 = 7 \rightarrow$ collision

$h_1(18) = (18 + 1*1) \bmod 11 = 8$

$h_0(62) = 62 \bmod 11 = 7 \rightarrow$ collission

$h_1(62) = (62 + 1*1) \bmod 11 = 8 \rightarrow$ again collision

$h_2(62) = (62 + 2*2) \bmod 11 = 0$ ✓



3- Double Hashing:

- Use another hash function hash 2(x), and look for $i * \text{hash 2}(x)$ slot in the i'th iteration.

→ $hi(X) = (\text{Hash}(X) + i * \text{Hash 2}(X)) \% \text{Hash table size}$

! So on.

⇒ Comparison

1- Linear Probing	2- Quadratic Probing	3- Double Hashing
• Easy to implement	• Average cache perform	• Poor cache performa.
• Best cache performance		• No clustering
• Suffers from clustering	• suffers less clustering	• Requires more computation time.