18/7/19 Data Structures & Programming Methodology

Token - Smallest unit of C program

- keywords (32 in C)
- Identifiers (name of variables)
- constants
- Strings
- Special symbols
- Operators

* Literal - Whose value cant be changed throughout the program

Datatypes — Primitive datatype
— Abstract datatype/Derived datatype.

Data - Raw, unorganised Eg: Ram, 20

Information - Meaningful data Eg: The age of Ram is 20.

Entities - Real time information to which ~~information~~ meaning can be added

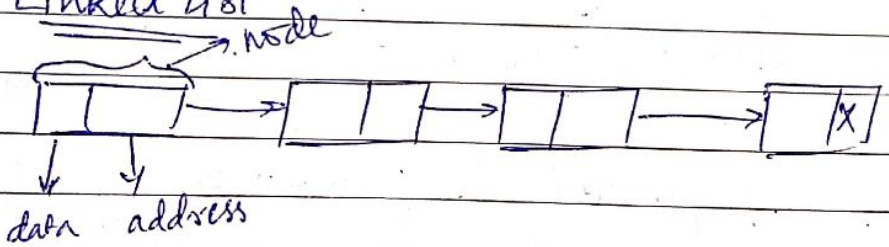
Data Structure - It is a triplet of

Domain (D) : $(0, \pm1, \pm2, \ldots)$
Function (F) : $(+, -, /, *, \%)$
Axioms (A) : (set of rules)

Classic Data Structures

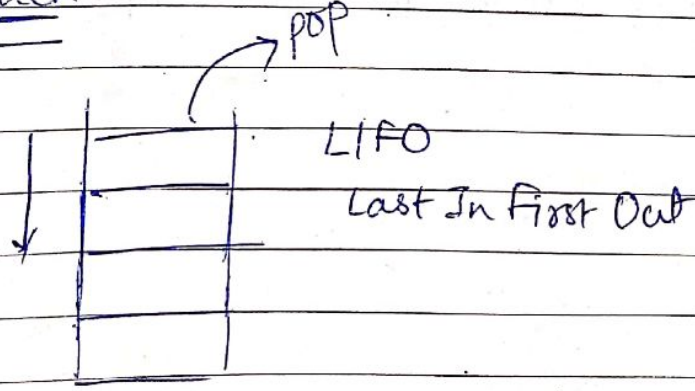
- Linear data structures
  - arrays
  - linked list
  - Stacks
  - Queues
- Non linear data structures
  - Trees
  - Graphs
  - Tables
  - sets

Linked List



Address here is not fixed

Stack



Queue

FIFO
First In First Out

Word: Size of a single element of array (W).

* Array

Size: The no of elements inserted
Type: Int or float or char
Index: A[i] → Index or position
Base address: Address of 1st element
Range: Lower bound (L) Upper bound (U)

Size(A) = U-L+1

If each element requires 1 word, then the location of any element A[i] in the array = M+(i-1)

where the first element can be stored in location M. Likewise an array can be written as A[L.....U] where L and U denote the lower and upper bounds for index. If it is stored starting from memory location M and for each element it requires W no. of words then the address for A[i] will be M+(i-L)W. This formula is called indexing formula.

Q. Suppose an array A[-15.....64] is stored in a memory whose starting address is 459. Assume that the word size for each element is 2. Then obtain the following:

a) How many number of elements are there in array A? 80

b) If one word of the memory = 2 bytes then how much memory is required to store the entire array? 160 bytes

c) What is the location for ~~A=0~~ A_{50}

~~459+(50-1) = 459+49 = 508~~
459+(50-1)2 = 459+98 = 557

Time complexity Pseudo code
Space complexity

d) What is the location of 10th element ? 479-509
e) Which element is located at 589? 30th element
14 15th element 50

Q2. Suppose an array A[-51 . . . 48] is stored in a memory whose starting address is 1000. assume that word size for each element is 4. Then obtain the following.

a) How many no. of elements are there in array A?
b) How much memory is required to store the entire array?
c) What is address location for A_1? → Index 1
d) What is the address location for 53rd element?
e) Which element is located at address 1076?

-52

a) 100 b) 400 bytes c) 1008 d) 1108 e)

1076
4 | 76 19

52
4
108

1000

* Traversing of Array

Algorithm TRAVERSE_ARRAY()

Input : An array A with elements
Output : According to PROCESS() → E is a function
Data structures : Array [U. . . L]

Steps

1. i = L
2. while i ≤ U do
   i) PROCESS(A[i])
   ii) i = i+1
3. End while
4. Stop

* Algorithm SORT_ARRAY()

Steps:

1. i = U
2. While i ≥ L
   1. j = L
   2. while j < i do
      1. If ORDER(A[j], A[j+1]) = FALSE
         1. SWAP(A[j], A[j+1])
      2. End If
      3. j = j+1
   3. End while
   4. i = i-1
3. End while
4. Stop.

25/7/19 Search Array

~~Step 1)~~ i = L, found = 0, location = 0
~~Step 2)~~ while i ≤ U do
}

1) i = L, found = 0, location = 0
2) while (i ≤ U) and (found = 0) do
   1) If COMPARE (A[i], KEY = TRUE) then
      1) found = 1
      2) location = i
   2) Else
      1) i = i + 1
   3) End if
3) End while
4) If found = 0 then
   1) print "Search is unsuccessful, key is not in array."
5) else
   1) print "Search is successful; key is at location", location
6) End if
7) Return (location)
8) Stop.

arr[n]

```
for (i=0; i<=n; i++)
{
    scanf("%d", &arr[i]);
}

for (i=0; i<=n; i++)
{
    if (arr[i] == n)
    {
        printf(&arr[i]);
        & break;
    }
}
```

→ first check whether the array is full. If not, we need to find the empty location

Q. Write an algorithm to insert an element ~~at a~~ given location in array at location.

1) If A[U] ≠ NULL
   1) Print "Array is full, no insertion possible."
   2) Exit
2) Else → No. of elements present in the array.
   1) i = U
   2) while i >= LOCATION do
      1) A[i+1] = A[i]
      2) i = i-1
   3) End while
   4) A[LOCATION] = KEY
   5) U = U+1
3) End if
4) Stop.

Q Homework Write an algorithm to delete an element

Q. Write an algorithm to merge two arrays

A_1 A_2

L_1 U_1 L_2 U_2

~~1) i = L2~~
~~2) A1[i+1] = A2[L2]~~

1) Start
2) $i = U_1$
~~3) A[U1+1]~~
3) $A_1[i+1] = A_2[L_2]$
4) $i = i+1$
5)

Multidimensional Array

Elements can be stored in two ways

- Row major order
- column major order.

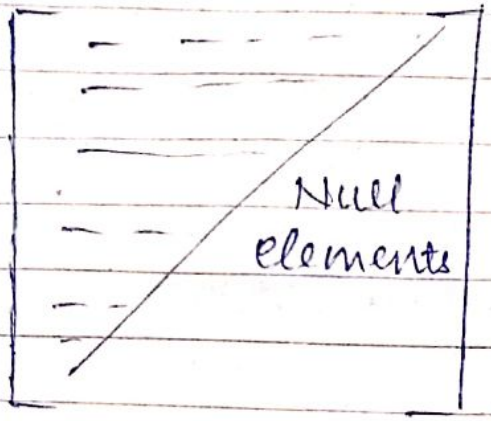
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & \cdots & \cdots \\ \vdots & \vdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{bmatrix} \qquad \begin{bmatrix} a_{11} & \cdots & \cdots \\ a_{21} & \cdots & \cdots \\ a_{31} & \cdots & \cdots \\ \vdots & & \end{bmatrix}$$

Q For the matrix $[A]_{3*4}$ the location A_{32} will be calculated as ____10____ where the base address is 1.

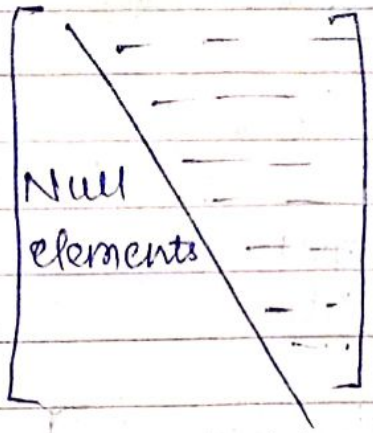
If some base address is given what will happen in case of row major order & column major order.

* Sparse Matrices

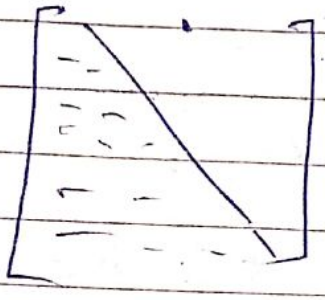
- Triangular
  - Lower
    - lower left
    - lower right
  - Upper
    - Upper left
    - Upper right
- Band Matrices
  - Diagonal
  - AB Band
  - Tri diagonal.



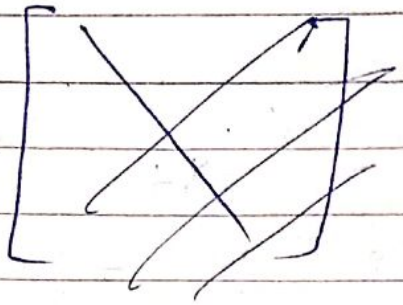
Upper left Triangular

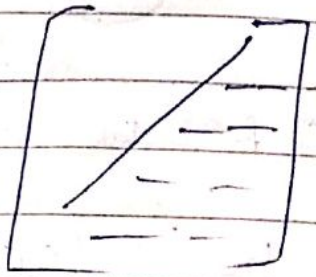


Upper right triangular.

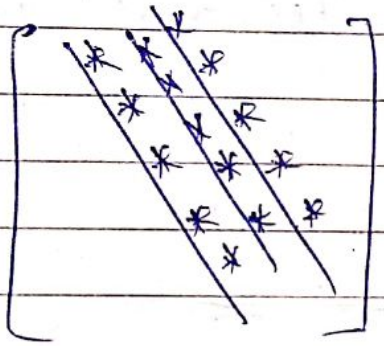


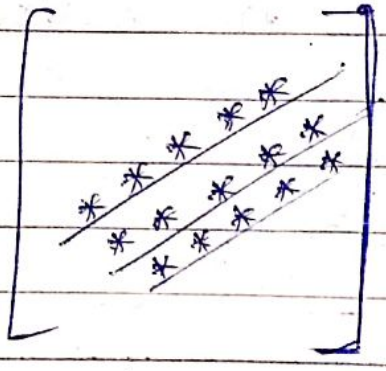
lower - left triangular



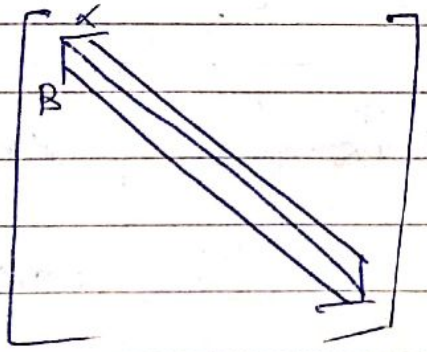


Lower - Right Triangular.

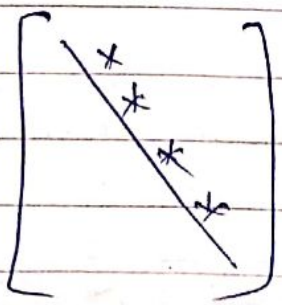




Tridiagonal Matrices



α β band matrices



Diagonal Matrices.

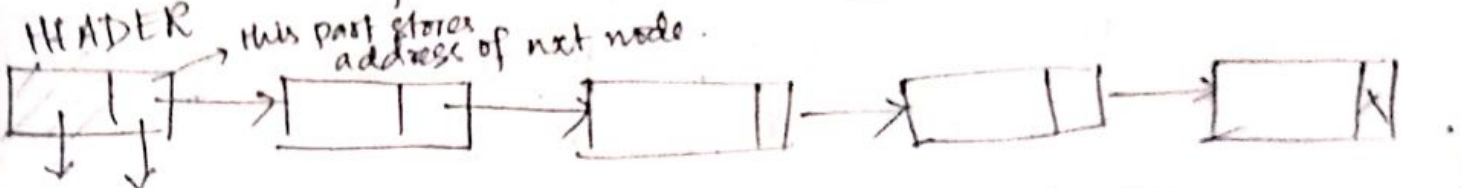
8/8/19

Linked List

→ size of linked list can be changed during runtime.

~~Size of linked~~

Smallest unit of linked list - Node.

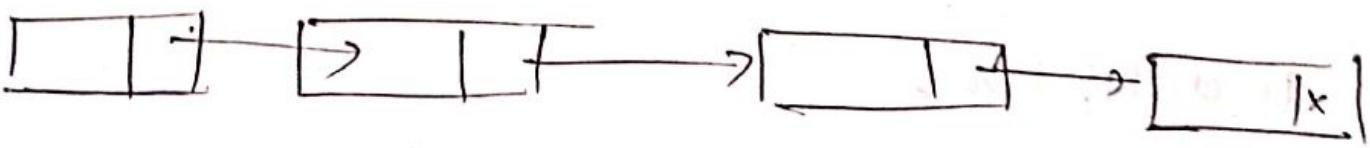


There is no need of consecutive memory locations in a linked list.

```
Struct
  {
    int p;
    struct node * p;
  }
```

TRAVERSE_SL_HEADER

1. ptr = HEADER.LINK
2. while (ptr ≠ NULL) do
   1. PROCESS (ptr)
   2. ptr = ptr.LINK
3. End while
4. Stop

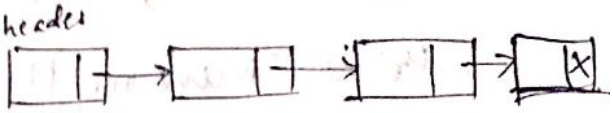


Insertion of a node

1. At front
2. At End
3. At any point

① Insertion at front

1. ptr = HEADER.LINK
2. While (ptr ≠



PROCEDURE GETNODE (NODE)

1. If (AVAIL = NULL)
   1. Return (NULL)
   2. Print "Insufficient Memory. Unable to allocate memory."
2. Else
   1. ptr = AVAIL
   2. while (SIZE OF(ptr) ≠ SIZE OF (NODE))
      and (ptr ≠ NULL)
      1. ptr1 = ptr
      2. ptr = ptr.LINK
   3. Endwhile
   4. If (SIZE OF (ptr) = SIZE OF (NODE))
      1. ptr1.LINK = ptr.LINK
      2. Return (ptr)
   5. Else
      1. Print "The memory is too large to fit"
      2. RETURN (NULL)
   6. End If

3. End If
4. Stop.

INSERT_SL_FRONT

1. new = GETNODE (NODE)
2. If (new = NULL) then
    1. Print "Memory Underflow : NO insertion"
    2. Exit
3. Else
    1. new.LINK = HEADER.LINK
    2. new DATA = x
    3. HEADER.LINK = new
4. End If
5. Stop.

INSERT_SL_BACK

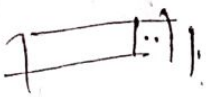
1. new = GETNODE (NODE)
2. If (new = NULL) then
    1. Print " Memory Underflow : No insertion "
    2. Exit
3. ELSE
    1. new.LINK = NULL ~~link~~
    2. new DATA = x
    3. NULL ~~link~~ = new
4. End If
5. Stop.

Insertion at back

1. new = GETNODE (NODE)
2. If (new = NULL) then
    1. Print "Memory underflow : No insertion".
    2. Exit
3. Else
    1. ptr = HEADER
    2. while (ptr.LINK ≠ NULL) do
        1. ptr = ptr.LINK
    3. Endwhile
    4. ptr.LINK = new
    5. new.DATA = X
    6. new.LINK = NULL
4. End If
5. Stop.

Insertion at any point

1. new = GETNODE (NODE)
2. If (new = NULL) then
    1. Print "Memory Underflow :
       No insertion"
    2. Exit
3. Else
    ~~1. ptr = HEADER.LINK~~
    ~~2. new.DATA = X~~



    1. ptr = HEADER
    2. while (ptr.DATA ≠ KEY) and (ptr.LINK ≠ NULL) do
        1. ptr = ptr.LINK
    3. Endwhile
    4. If (ptr.LINK = NULL) then
        1. Print "key is not there
           in the list
           Insertion is not possible"
        2. Exit
    5. Else
        1. new.LINK = ptr.LINK
        2. new.DATA = X
        3. Ptr.LINK = new
    6. End If
    7. Stop.

RETURN NODE (PTR)

1. ptr1 = AVAIL
2. while (ptr1. LINK ≠ NULL) do
    1. ptr1 = ptr1. LINK
3. Endwhile
4. ptr1. LINK = PTR
5. PTR. LINK = NULL
6. Stop.

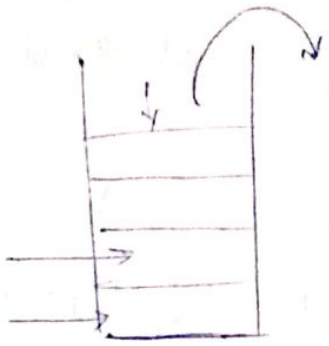
DELETE_SL_FRONT

1. ptr = HEADER. LINK
2. If (ptr = NULL)
    1. print "The list is empty!
       No deletion"
    2. Exit
3. Else
    1. ptr1 = ptr. LINK
    2. HEADER. LINK = ptr1
    3. RETRN NODE (ptr)
4. End If
5. Stop.

1. ptr1 = HEADER
2. ptr = ptr1 · LINK
3. while (ptr ≠ NULL) do
    1. If (ptr · DATA ≠ KEY) ~~do~~ then
        1. ptr1 = ptr
        2. ptr = ptr · LINK
    2. Else
        1. ptr1 · LINK = ptr · LINK
        2. RETURNNODE (ptr)
        3. Exit
    3. End If
4. Endwhile
5. If (ptr = NULL)
    1. Print "Key does not exist!"
6. End If
7. Stop.

29/8/19

Stacks



LIFO - Last In first out

Insert → push
Remove → pop.

* Push_A(ITEM)

Steps

1. If TOP ≥ SIZE then
    1. Print "Stack is full"
2. Else
    1. TOP = TOP+1
    2. A[TOP] = ITEM
3. End if
4. Stop

* POP_A ( )

1. If TOP < 1
    1. Print "Stack is empty"
2. Else
    1. ITEM = A[TOP]
    2. TOP = TOP-1
3. End If
4. Stop

* Status of Stack (How much space is left after insertion and deletion)

HW

-

Infix : operator b/w the operands. Eg : (A+B)

Prefix +AB also called Polish Notation

Postfix : AB+

Operator - +, -, /, %

Operands - variables

Stack Application : Infix to Postfix

Algo infix_postfix_conversion (q, p)

Push '(' on stack

q = q || ')'

Until stack is empty read q from left to right symbol one at a time

{
- if operand then put it at the end of P
- If '(' then push it to stack
- If operator (⊗) then pop all the operators from the top of the stack and add to P whose precedence is greater than or equal to the precedence of operator (⊗) push (⊗) at the top of the stack.
- If ')' then pop all the operators from the top of the stack and add them to p in order until '(' is found
- POP '(' '(' and ')' will be discarded

Infix to Postfix

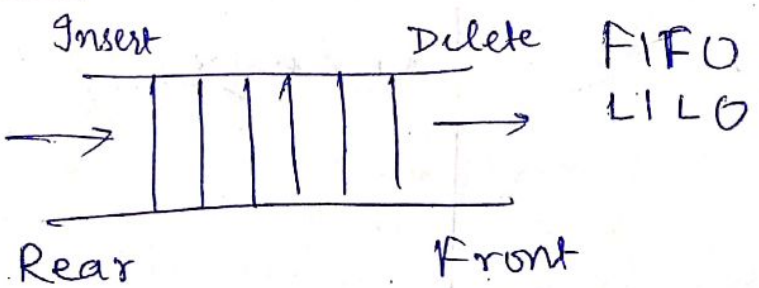
Q. (A+(B*C-(D/E↑F)*G)*H)

Stack Symbol	Stack Content	Expression as output
A	(	A
+	(+	A
(	(+(	AB
B	(+(	AB
*	(+(*	ABC
C	(+(*	ABC*
-	(+(-	ABC*
(	(+(-(	ABC*D
D	(+(-(	ABC*D
/	(+(-(/	ABC*DE
E	(+(-(/	
↑	(+(-(/↑ →	ABC*DE
F	→	ABC*DEF
)	(+(-	ABC*DEF↑/

$(A+B) \wedge C - (D * E) / f$

Stack Symbol	Stack Content	Expression P as output
A	(	A
+	(+	A
B	(+	AB.
(	(+)	AB+
^		
C		
-		
(		
D		
*		
E		
/		
f		

* Queue



if Front = Rear then Queue is empty

SIZE = Rear - Front + 1

Front = Rear ≠ 0

Enqueue (Rear end)

Steps

1. If (REAR = N) then
    1. Print "Queue is full"
    2. Exit
2. Else
    1. If (REAR = 0) and (FRONT = 0)
        1. FRONT = 1
    2. End If
    3. REAR = REAR + 1
    4. Q[REAR] = ITEM
3. End If
4. Stop.

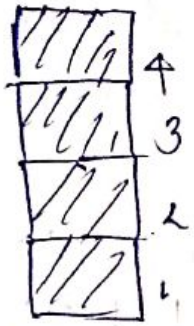
Dequeue (Front end).

1. If (FRONT = 0) then
    1. Print "Queue is empty"
    2. Exit
2. Else
    1. ITEM = Q[FRONT]
    2. If (FRONT = REAR)
        1. FRONT = 0
        2. REAR = 0
    3. Else
        1. FRONT = FRONT + 1
    4. End If
3. End If
4. Stop

30/8/19

STACK → ~~Data structure stores data~~

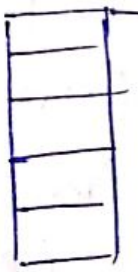
$$Z = (A+B)/F + \overline{C * (D+E)} * (A-B)$$



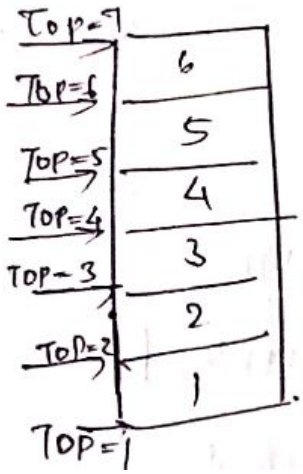
Stack is a linear data structure.
It in which all insertion and deletions are made only at one end which is called top.
The stack was 1st proposed in 1955 and first patented in 1957 by a German ~~Ete~~ Friedrich L. Bauer.

→ Same concept developed at the same time by an Australian Charles Leonard Hamblin.

→ Size of stack is denoted by MAX.



MAX :→ Maximum capacity.



MAX = 6

If TOP <= MAX, then ~~stack~~ ~~can~~ elements can be added to stack

A stack is an ordered collection of homogenous data elements where the insertion and deletion happens at only one end.

The stack is also called Last In First Out (LIFO)

* Stack Implementation

Stacks can be implemented in 2 ways

1. Static Implementation
2. Dynamic Implementation

1. — Static implementation uses array to implement stack — Not — flexible — the size has to be ~~defined~~ declared while defining during the program designed - size can't be changed after that — also - not efficient with memory utilisation

2. Dynamic Implementation - Dynamic implementation is done through linked list representation and uses pointer to implement the stack.

PUSH
↙
overflow / Stack full

POP
↙
Underflow / Stack empty.

24/10/19

Time Complexity → Depends on number of loops.

Best Case, Average Case, Worst case.

```
for(i=0; i<n; i++)
  { stmt; }
```

→ This statement will be executed 'n' times

∴ Time complexity $O(n)$ → order of n

Same applies for, 2) `for(i=n; i>0; i--)`

$O(n)$

3. `for(i=1; i<n; i=i+2)`

```
{ stmt; }
```

$O(n)$.

4) `for(i=0; i<n; i++)` → n

```
{
  for(j=0; j<n; j++)  → n×n
  {
    { stmt }
  }
```

5) `for(i=0; i<n; i++)`

```
{
  for(j=0; j<i; j++)
  {
    stmt;
  }
```

0/1

i	j	no. of times
0	0	0
1	0 1x (j<i)	1
2	0 1 2x	2
3	0 1 2 3x	3
.		
.		
.		
n		n. (from ~~0 to~~ i=0 to n-1)

$\therefore 1+2+3+\ldots+n = O\left(\frac{n(n+1)}{2}\right) = O\left(\frac{n^2+n}{2}\right)$

6)
```
P=0
for (i=1; P<=n; i++)
  {
    P=P+1;
  }
```

i	P	no. of times
1	0+1	
2	1+2	
3	1+2+3	
.		
.		
k	$1+2+3+\ldots+k$	

$P > n$

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

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

$$\boxed{k = \sqrt{n}}$$

7) for (i = 1; i < n; i = i * 2)

{ stmt }

i	no. of times
1	
1×2 = 2	
1×2×2 = 4	
1×2×2×2 = 8	
16	

2^k

when $i >= n$.

$2^k >= n$

$2^k = n$

$k = \log_2 n$

$O(\log_2 n)$

for n = 8

1
2
4

$\log_2 8 = 3$

for n = 10

1
2
4
8
.
.
$\log_2 10$

$= 3.32$

$\lceil \log_2 10 \rceil =$

$\lceil 3.32 \rceil = 4$

8) for (i = n; i >= 1 ; i = i/2)

i

n

$\frac{n}{2}$

.

.

.

$\frac{n}{2^k}$

$\frac{n}{2^k} < 1$

$n < 2^k$

$k = \log_2 n$

9) for (i = 0; ~~i <~~ i*i < n ; i++)

i

0

1

2

.

.

.

1+2+...+n

~~$i^2 >= \frac{n(n+1)}{2}$~~

~~$i^2 >= \frac{n^2+1}{2}$~~

i^2

$i^2 >= n$

$i^2 = n$

$i = \sqrt{n}$

$O(\sqrt{n})$

$k^2 > n$

$k^2 = n$

$k = \sqrt{n}$

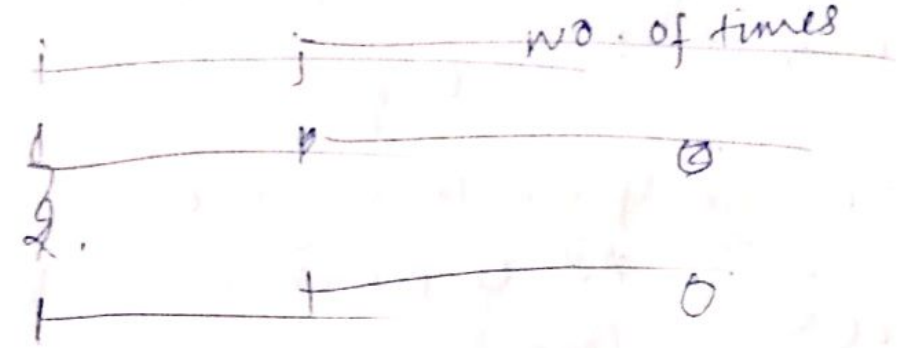
10) for(i=0; i<n; i++)
{ stmt; }

for (j=0; j<n; j++)
{ stmt; }

11) p=0

```
for (i=1, i<n, i=i*2)
{ p++; }
for ( j=1; j<p; j=j*2)
{ stmt; }
```

$\log\log n$



~~for (i=0~~ p=0;

```
for (i=1; i<n; i=i*2)
{ p++; } ⟶ log2n
for (j=1; j<p; j=j*2)
{ stmt; } ⟶ log2p
```

$2^k = n$

$k = \log_2 n$

$O(\log\log n)$

~~$2^{\log_2 n} = n$~~

$k = \log_2 \log_2 n$

$2^k = \log_2 n$

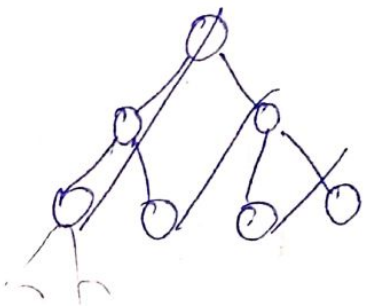
Q. Suppose following are the ~~inde~~ inorder and postorder traversals of a binary tree

L Root R

In order $n_1, n_2, n_3, \ldots, n_9$

L R Root

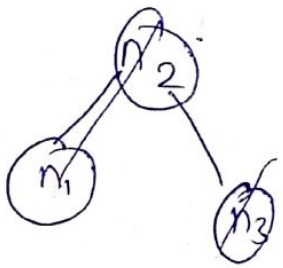
Post order $n_1, n_3, n_5, n_4, n_2, n_8, n_7, n_9, n_6$

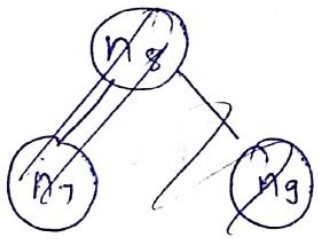


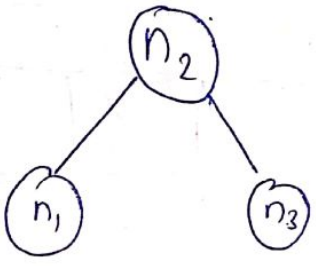
L Root R L Root R

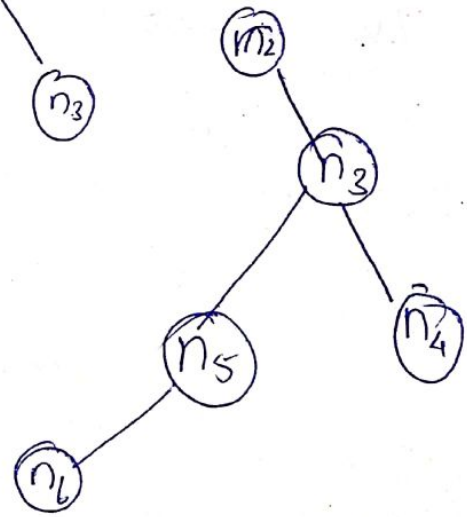
In order $n_1\ n_2\ n_3\ n_4\ n_5\ n_6\ n_7\ n_8\ n_9$

L Root R L Root R









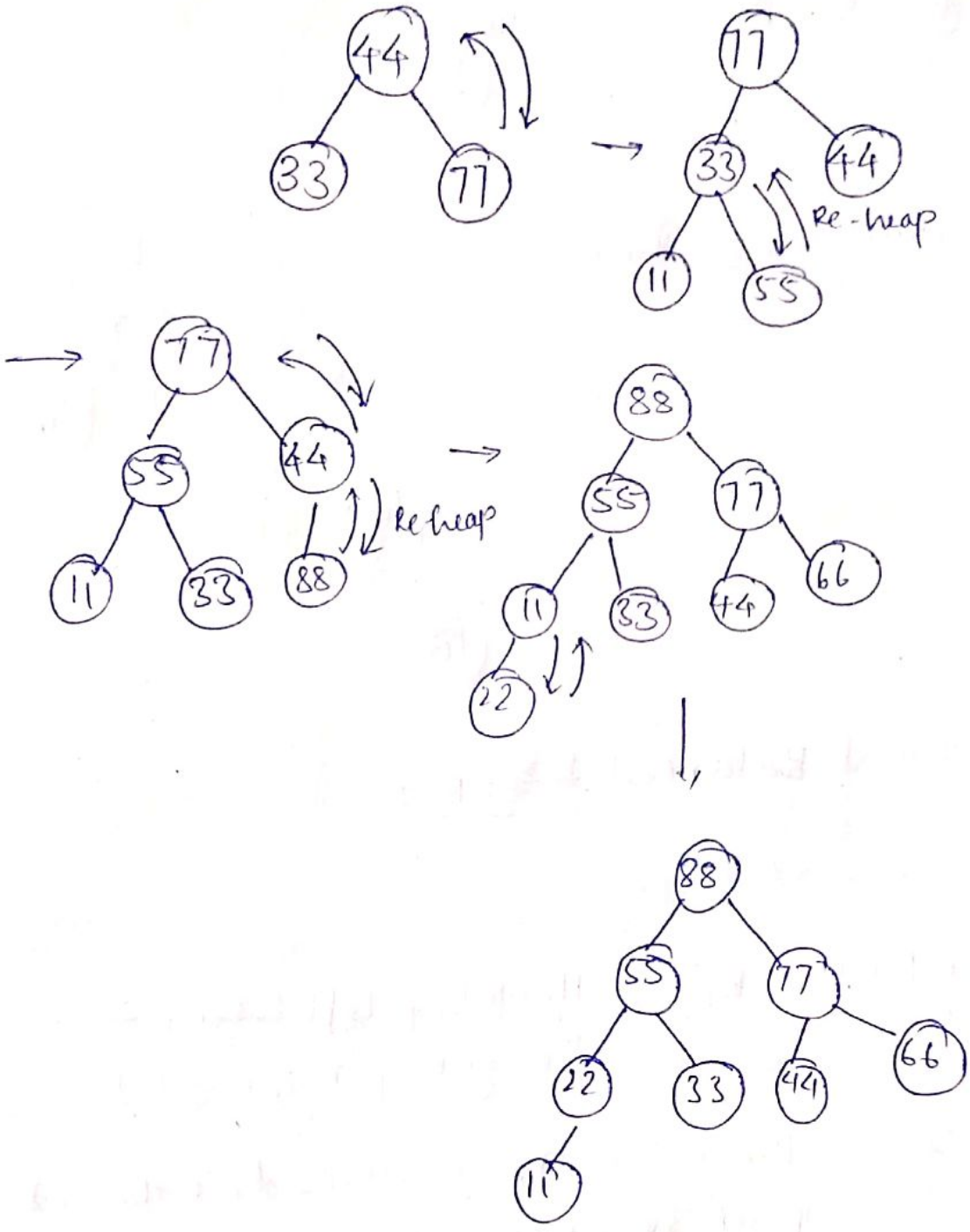
7/11/19

Data Structure

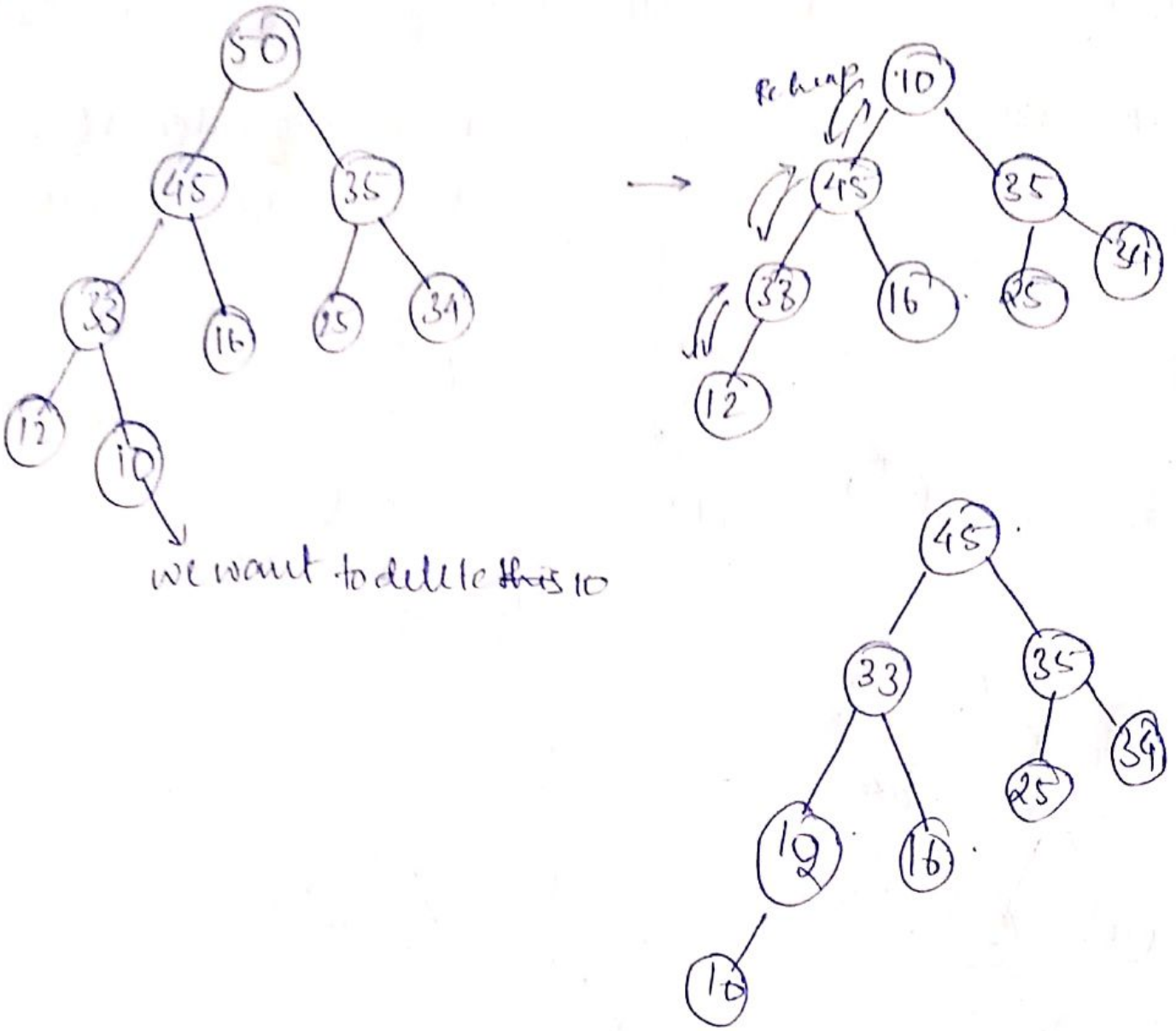
Heap tree , Re-heap

Right - greater value
left - smaller value

44, 33, 77, 11, 55, 88, 66, 22.



Except the root node, no other node can be deleted from heap tree.



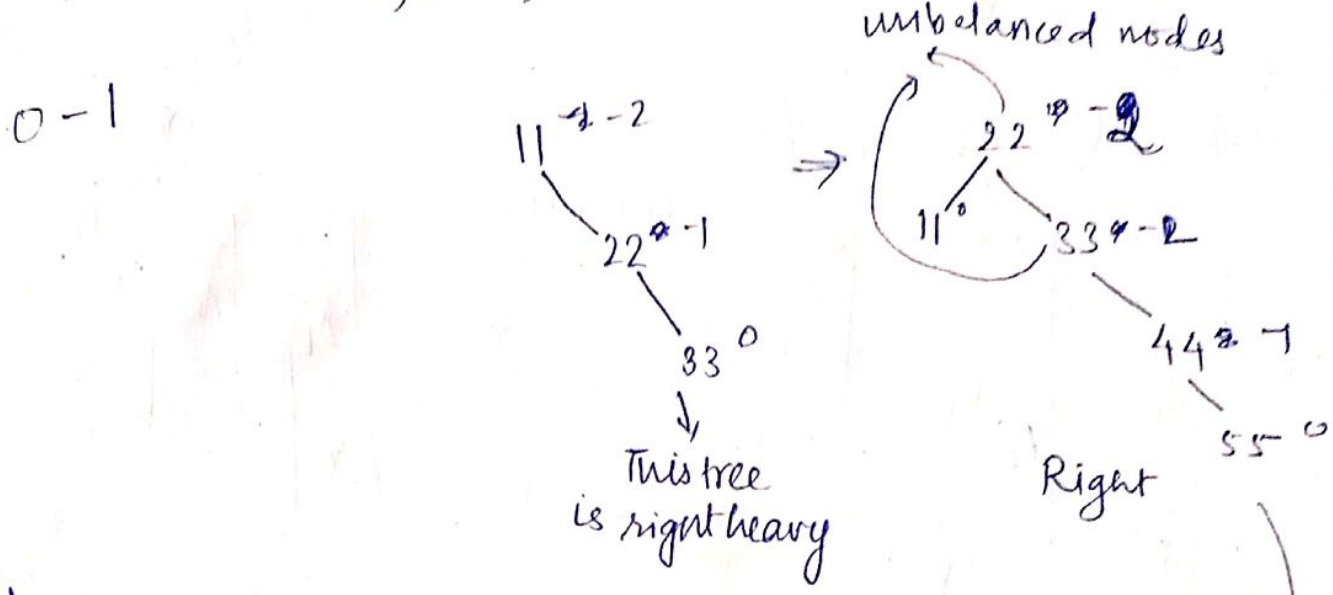
* AVL (Height Balanced ~~Tree~~ BST)

11, 22, 33, 44, 55, 66

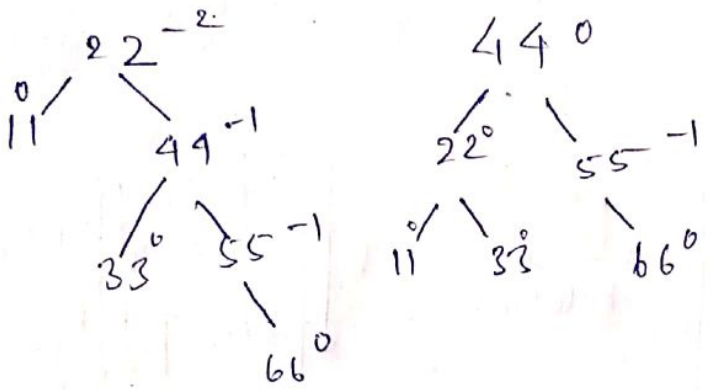
Balancing Factor (B_F) = Height of left subtree − Height of right subtree

B_f (-1, 0, 1) → Then the tree is already balanced if all the nodes have B_f either -1 or 0 or 1.

11, 22, 33, 44, 55, 66

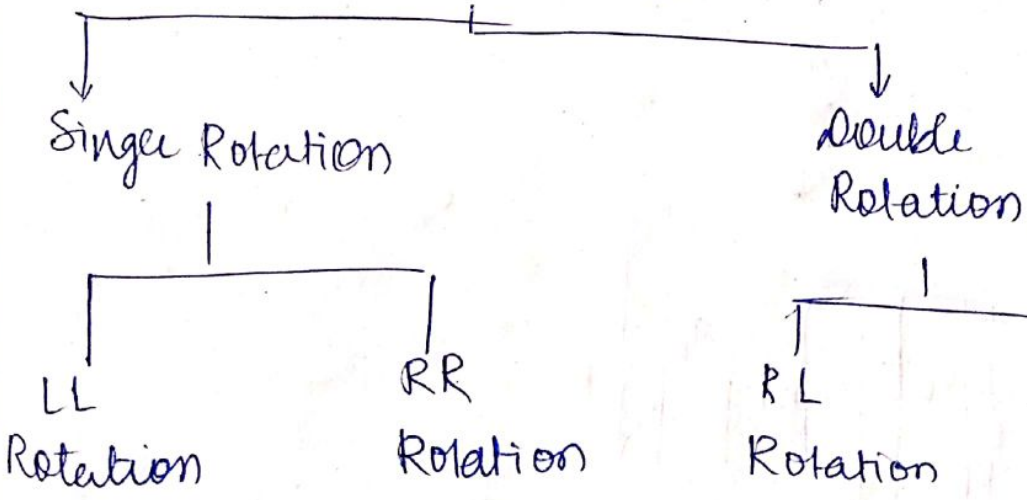


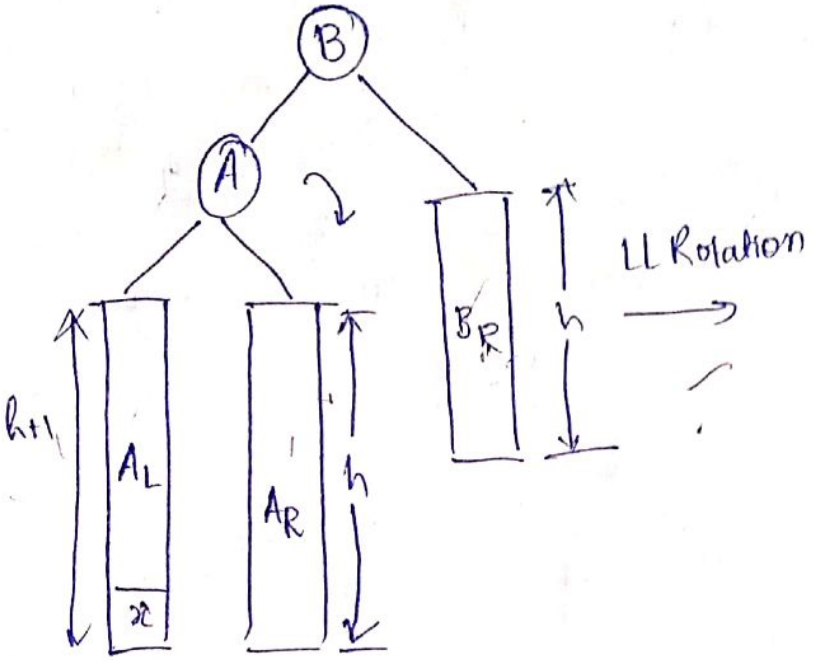
Always rotate that node which is closer to the newly inserted node.

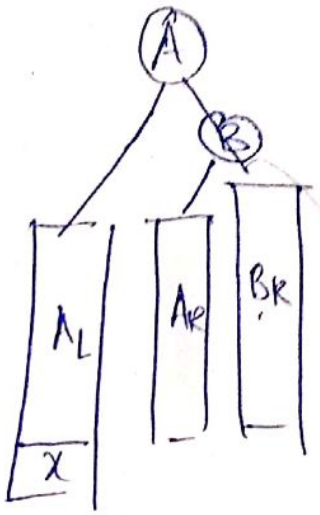


~~* Different types of rotations~~

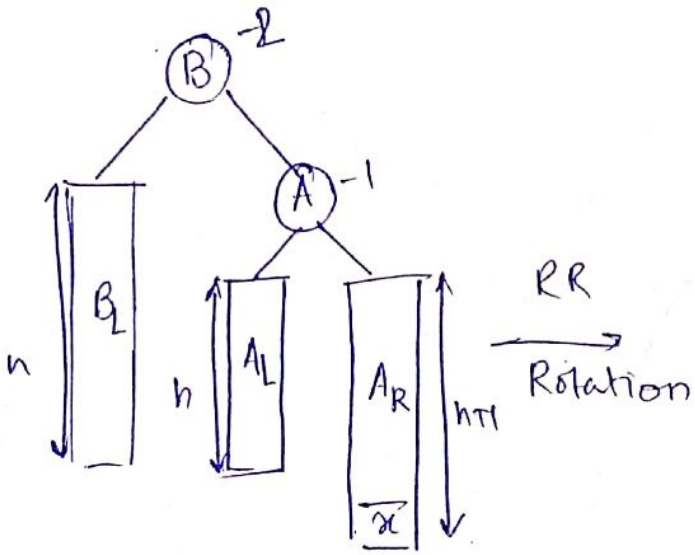
* Rotation In AVL Tree

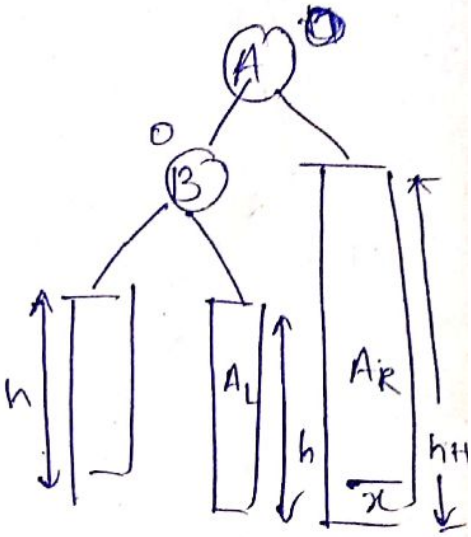




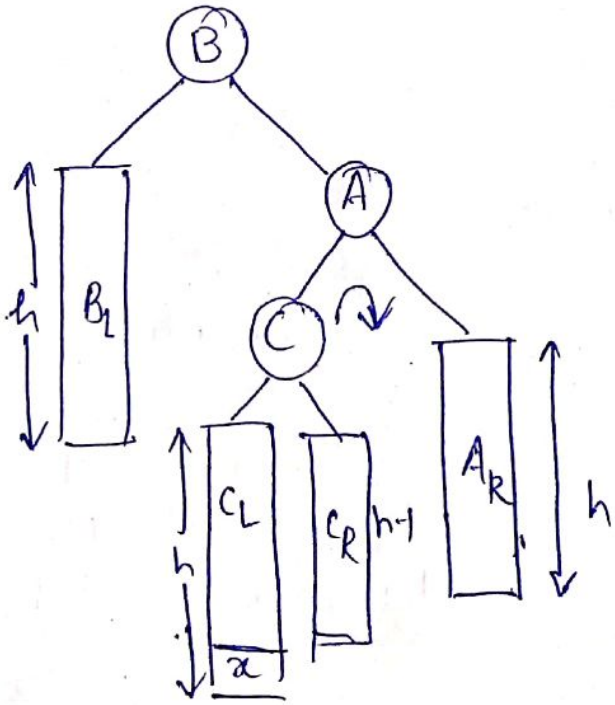


Q.

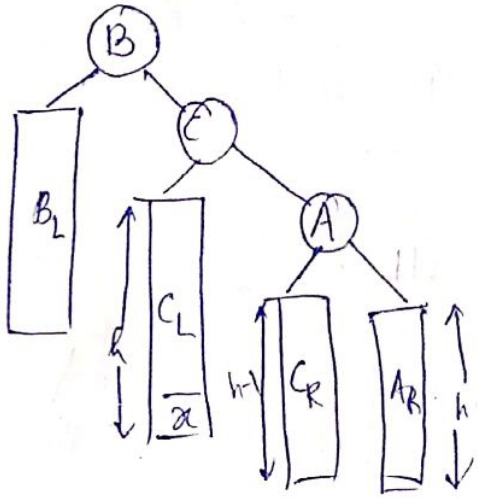




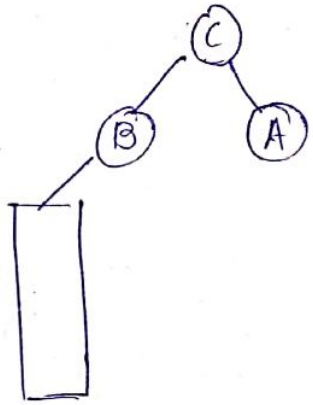
Q.



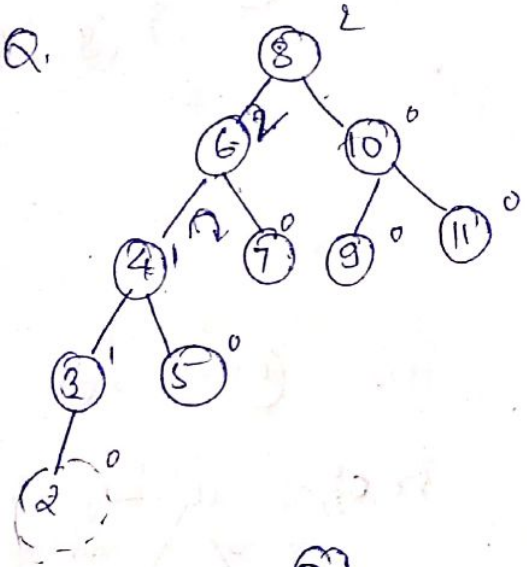
→

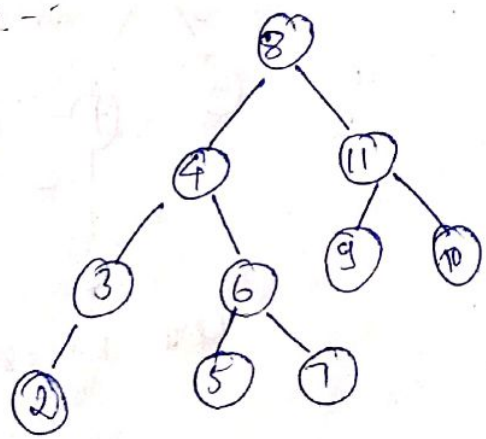


↓

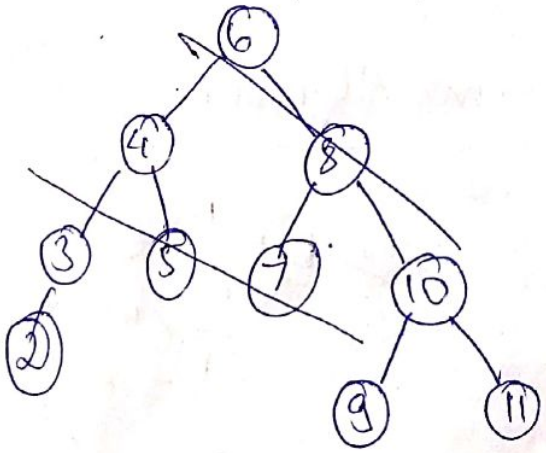


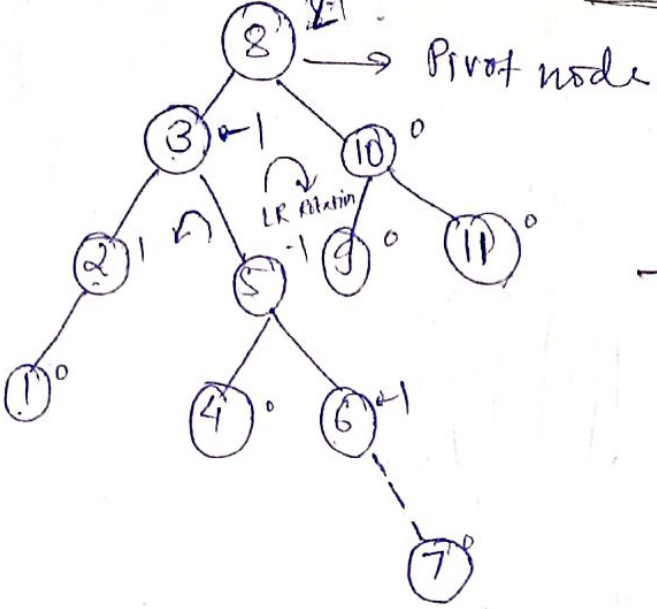
Q.

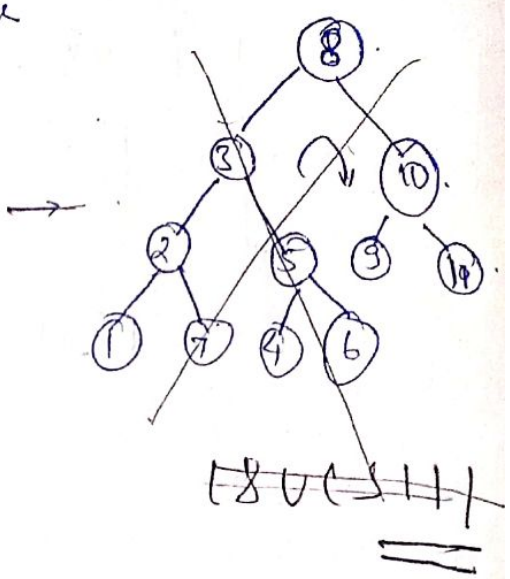


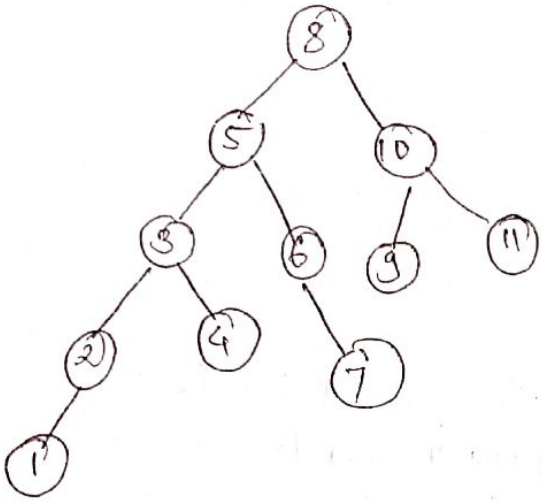


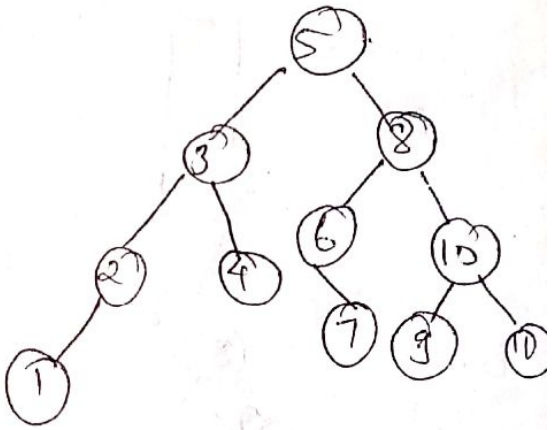
It is a height unbalanced Binary search tree.



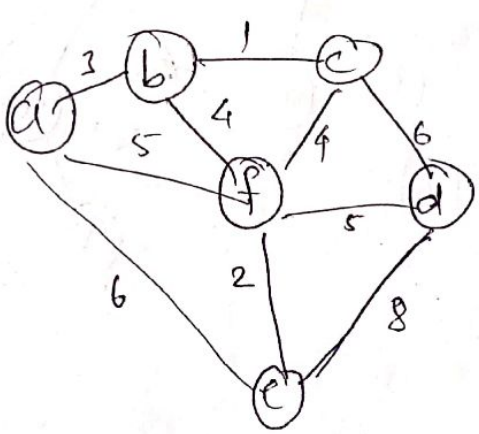




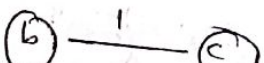




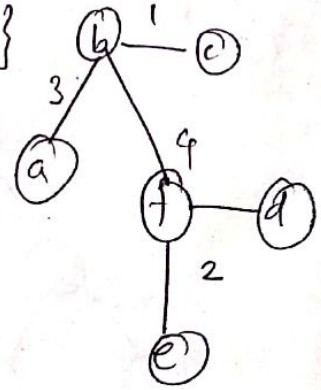
* Prim's Algorithm



{b,c}

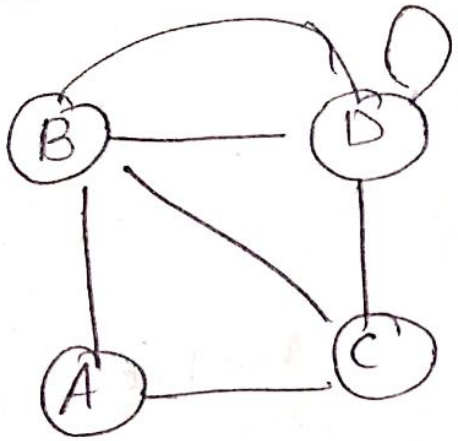


{b, c, a}

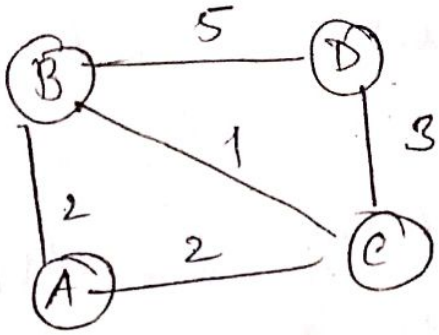


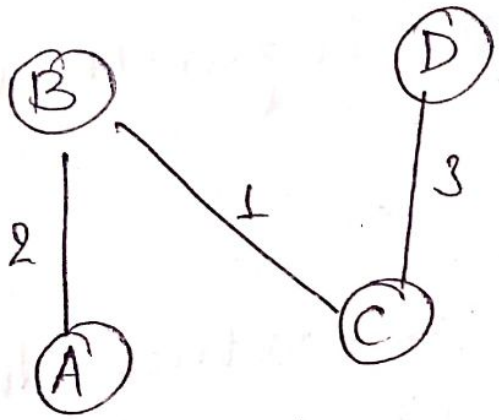
Cost = 3+1+2+4+5
= 15

Kruskal's Algorithm



Omit self loop & parallel edges





Wt = 6.