GATE CSE NOTES

by

Joyoshish Saha

Downloaded from https://gatecsebyjs.github.io/

With best wishes from Joyoshish Saha

Introduction.

* **Data** (Datum - fact) - In computers, data is the value assigned to a variable.

* **Data Structure** - Data structure is a data organization & storage format that enables efficient access & modification.

Data structures can implement one or more particular Abstract data types. DS is a concrete implementation of the space provided by an ADT.

* **Abstract Data Type**: An ADT is a mathematical model for data types, where a data is defined by its behaviour from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type & the behaviour of these operations.

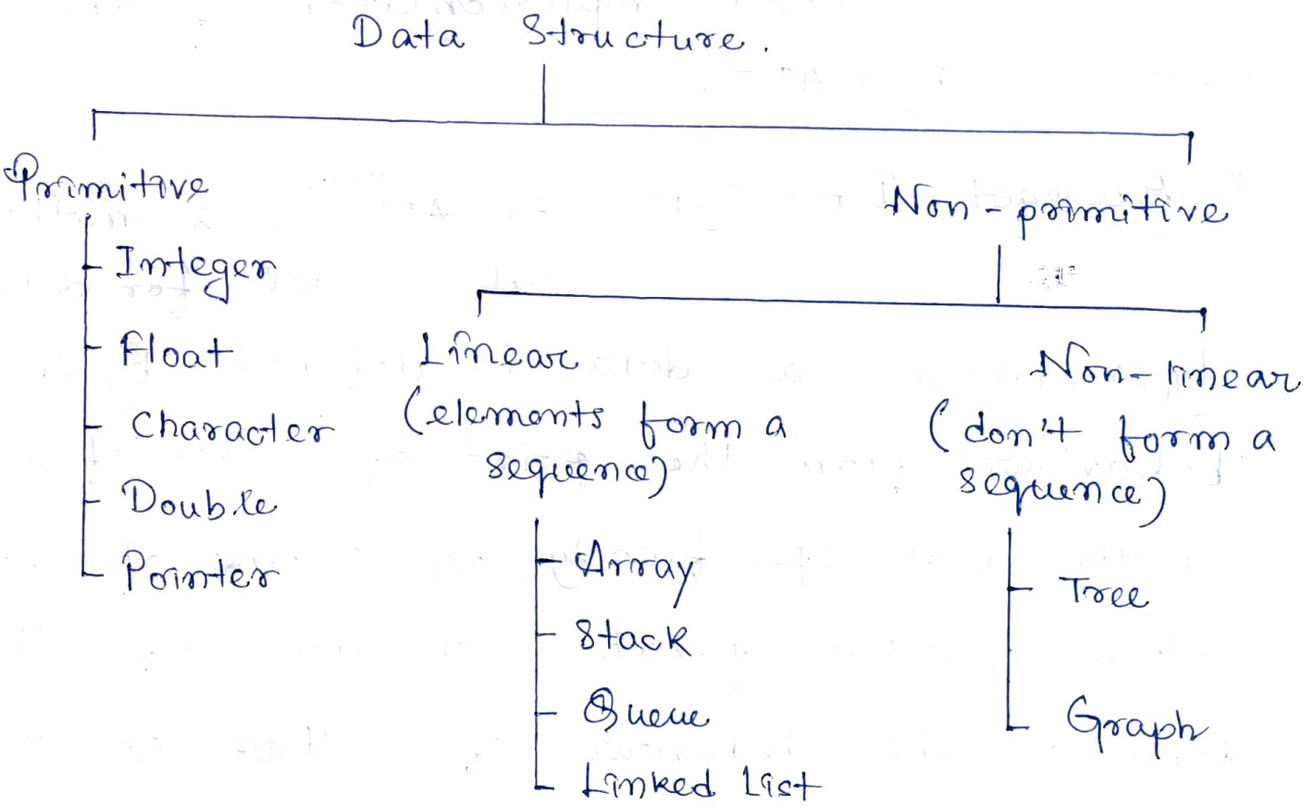
This contrasts with DS, which are concrete representations of data & are the point of view of an implementer, not a user.

Formal def": (Class of objects whose logical behavior is defined by a set of values & a set of operations.)

• Data abstraction is the separation between the specification of a data object from the outside world & its implementation.

Eg. For the set ADT, operations possible are union, intersection, size, complement & also the possible values are $\{\} = \phi$, $\{0,1\}$, $\{-2, 3, 39\}$ etc.

* Classification of Data Structure :



* Operations on Data Structure :

1. Inserting
2. Searching
3. Traversing
4. Sorting
5. Deleting.
6) Selection
7. Merging
8. Splitting.

* Abstraction & implementation :

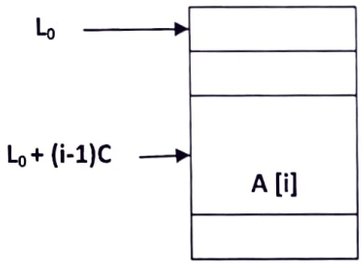
ADT	vs.	DS
1. Vehicle		Golf cart
		Bicycle
		Smart car
2. Map		Tree Map
		Hash Map
		Hash table
3. List		Dynamic array
		Linked list
4. Queue		Linked list based queue
		Array based queue
		Stack based queue

Array

Explain Array in detail

One Dimensional Array

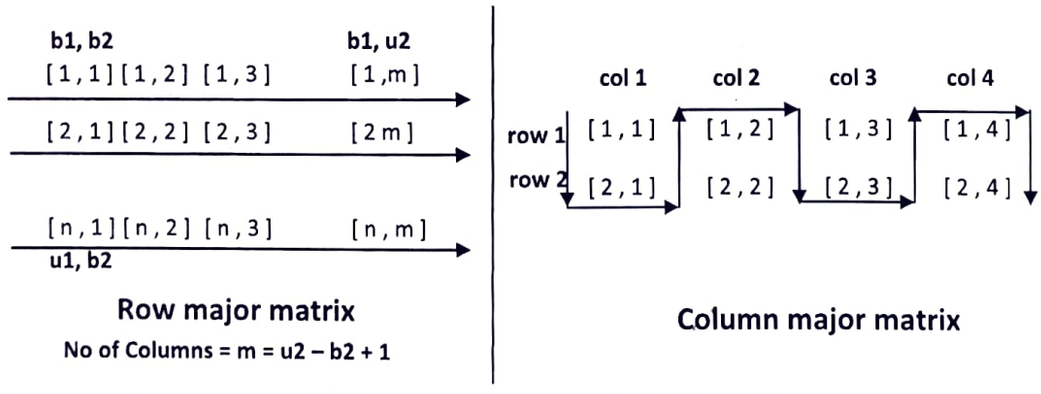
- Simplest data structure that makes use of computed address to locate its elements is the one-dimensional array or vector; number of memory locations is sequentially allocated to the vector.
- A vector size is fixed and therefore requires a fixed number of memory locations.
- Vector A with subscript lower bound of "one" is represented as below....



- L_0 is the address of the first word allocated to the first element of vector A.
- C words are allocated for each element or node
- The address of A_i is given equation **Loc (A_i) = L_0 + C (i-1)**
- Let's consider the more general case of representing a vector A whose lower bound for it's subscript is given by some variable b. The location of Ai is then given by **Loc (A_i) = L_0 + C (i-b)**

Two Dimensional Array

- Two dimensional arrays are also called table or matrix, two dimensional arrays have two subscripts
- Two dimensional array in which elements are stored column by column is called as column major matrix
- Two dimensional array in which elements are stored row by row is called as row major matrix
- First subscript denotes number of rows and second subscript denotes the number of columns
- Two dimensional array consisting of two rows and four columns as above Fig is stored sequentially by columns : A [1, 1], A [2 , 1], A [1 , 2], A [2 , 2], A [1 , 3], A [2 , 3], A [1, 4], A [2, 4]
- The address of element A [i , j] can be obtained by expression **Loc (A [i , j]) = L_0 + (j-1)*2 + i-1**
- In general for two dimensional array consisting of n rows and m columns the address element A [i , j] is given by **Loc (A [i , j]) = L_0 + (j-1)*n + (i – 1)**
- In row major matrix, array can be generalized to arbitrary lower and upper bound in its subscripts, assume that b1 ≤ I ≤ u1 and b2 ≤ j ≤u2



- For row major matrix : **Loc (A [i , j]) = L_0 + (i – b1) *(u2-b2+1) + (j-b2)**

Applications of Array

1. Symbol Manipulation (matrix representation of polynomial equation)
2. Sparse Matrix

Symbol Manipulation using Array

- We can use array for different kind of operations in polynomial equation such as addition, subtraction, division, differentiation etc...
- We are interested in finding suitable representation for polynomial so that different operations like addition, subtraction etc... can be performed in efficient manner
- Array can be used to represent Polynomial equation
- **Matrix Representation of Polynomial equation**

	Y	Y^2	Y^3	Y^4
X	$X\,Y$	$X\,Y^2$	$X\,Y^3$	$X\,Y^4$
X^2	$X^2\,Y$	$X^2\,Y^2$	$X^2\,Y^3$	$X^2\,Y^4$
X^3	$X^3\,Y$	$X^3\,Y^2$	$X^3\,Y^3$	$X^3\,Y^4$
X^4	$X^4\,Y$	$X^4\,Y^2$	$X^4\,Y^3$	$X^4\,Y^4$

e.g. $2x^2+5xy+Y^2$
is represented in matrix form as below

	Y	Y^2	Y^3	Y^4
X	0	0	**1**	0	0
X^2	0	**5**	0	0	0
X^3	**2**	0	0	0	0
X^4	0	0	0	0	0
	0	0	0	0	0

e.g. $x^2+3xy+Y^2+Y-X$
is represented in matrix form as below

	Y	Y^2	Y^3	Y^4
X	0	0	**1**	0	0
X^2	**-1**	**3**	0	0	0
X^3	**1**	0	0	0	0
X^4	0	0	0	0	0
	0	0	0	0	0

- Once we have algorithm for converting the polynomial equation to an array representation and another algorithm for converting array to polynomial equation, then different operations in array (matrix) will be corresponding operations of polynomial equation

---

What is sparse matrix? Explain

- An mXn matrix is said to be sparse if "many" of its elements are zero.
- A matrix that is not sparse is called a dense matrix.
- We can device a simple representation scheme whose space requirement equals the size of the non-zero elements.

- **Example:-**
  - The non-zero entries of a sparse matrix may be mapped into a linear list in row-major order.
  - For example the non-zero entries of 4X8 matrix of below fig.(a) in row major order are 2, 1, 6, 7, 3, 9, 8, 4, 5

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

Fig (a) 4 x 8 matrix

Terms	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Row	1	1	2	2	2	3	3	4	4
Column	4	7	2	5	8	4	6	2	3
Value	2	1	6	7	3	9	8	4	5

Fig (b) Linear Representation of above matrix

- To construct matrix structure we need to record
  (a) Original row and columns of each non zero entries
  (b) No of rows and columns in the matrix
- So each element of the array into which the sparse matrix is mapped need to have three fields: row, column and value
- A corresponding amount of time is saved creating the linear list representation over initialization of two dimension array.

A =

0	0	6	0	9	0	0
2	0	0	7	8	0	4
10	0	0	0	0	0	0
0	0	12	0	0	0	0
0	0	0	0	0	0	0
0	0	0	3	0	0	5

- Here from 6X7=42 elements, only 10 are non zero. A[1,3]=6, A[1,5]=9, A[2,1]=2, A[2,4]=7, A[2,5]=8, A[2,7]=4, A[3,1]=10, A[4,3]=12, A[6,4]=3, A[6,7]=5.
- One basic method for storing such a sparse matrix is to store non-zero elements in one dimensional array and to identify each array elements with row and column indices fig (c).

	ROW	COLUMN	A
1	1	3	6
2	1	5	9

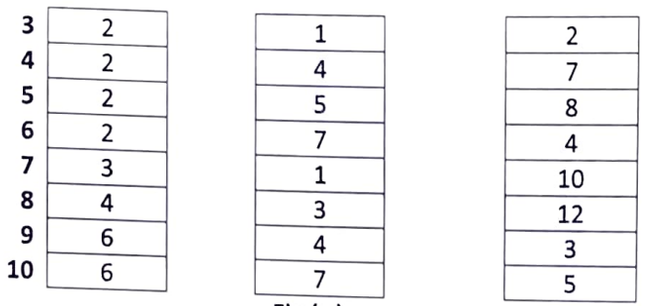
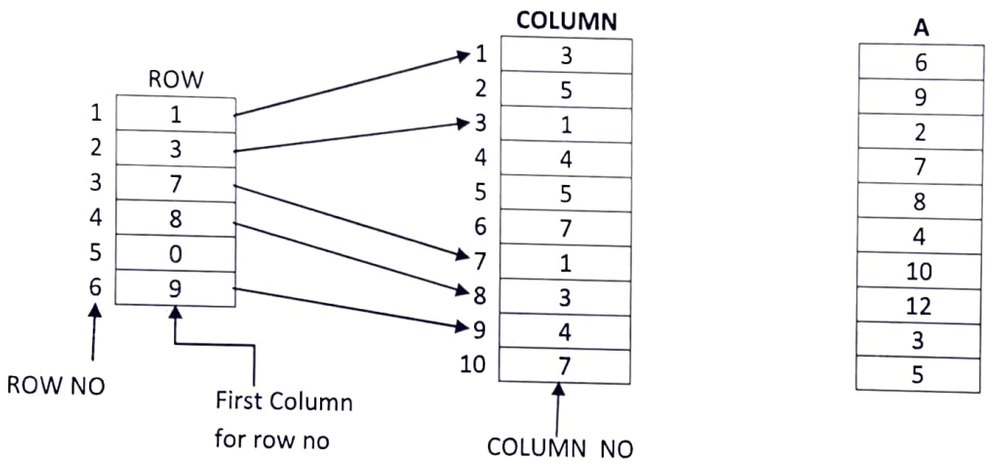


Fig (c)

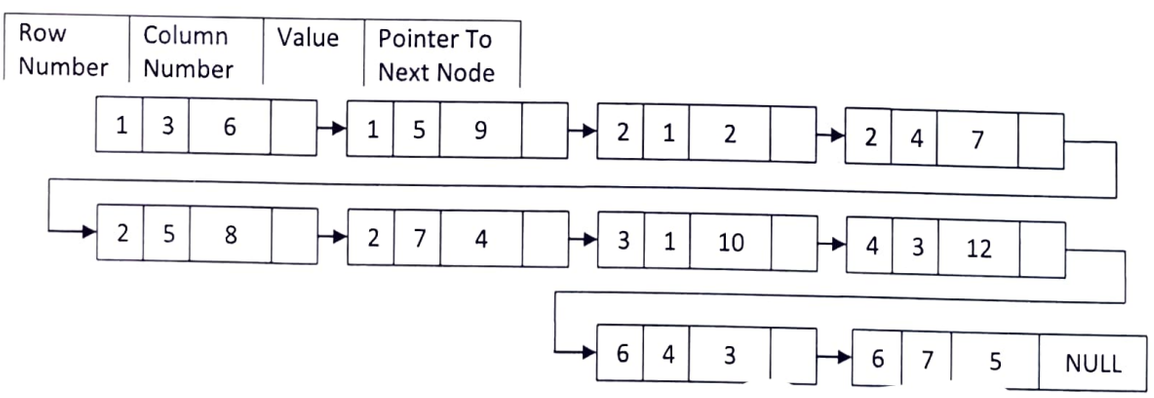


Fig(d)

- A more efficient representation in terms of storage requirement and access time to the row of the matrix is shown in fid (d). The row vector changed so that its i^{th} element is the index to the first of the column indices for the element in row I of the matrix.

Linked Representation of Sparse matrix

Typical node to represent non-zero element is



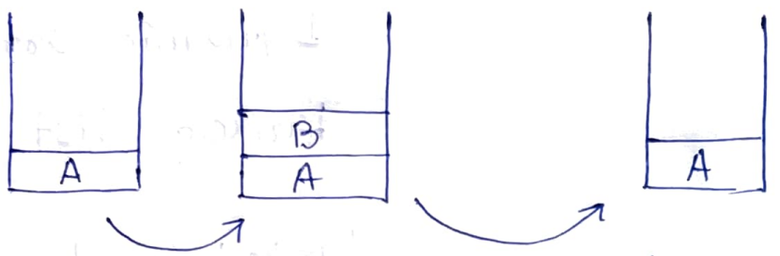
DS1 (Code, Th) DS2 (Adv.)

The Stack.

* **Stack** : A stack is a linear data structure where the elements are added & removed only from one end, that is called the TOP.

LIFO - Last in First out — The element that was inserted last is the first one to be taken out.

* We need stacks in function calls.

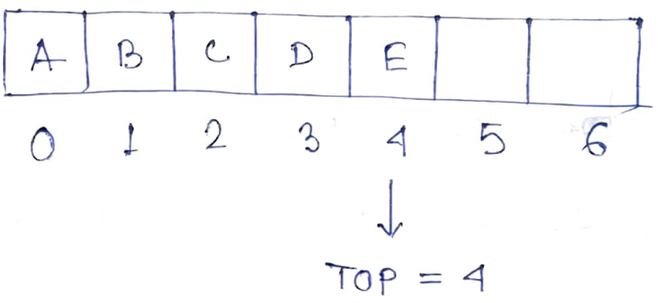


A calls B
B pushed

B executed
B popped.

* **Array Representation of Stack:**

- Every stack has a variable called **top** associated with it, that is used to store the address of the topmost element of the stack.
- **Max** — variable to store the max. no. of elements that the stack can hold.
- TOP = NULL — stack empty
  TOP = MAX - 1 — stack full



Array representation.

* Operations on Stack:

1. PUSH : Used to insert element into the stack. The new element is added to the topmost position of the stack.

Algo

Step 1: If- TOP = MAX-1
Print Overflow
Goto step 4
End if-

Step 2: Set Top = Top + 1

Step 3: Set stack [Top] = Value

Step 4: End.

2. POP : Used to delete element of the topmost position of stack.

Algo

1: If- TOP = NULL
Print Underflow
Goto 4
End if-

2: Set val = Stack [TOP]

3: SET TOP = TOP - 1

4: End.

3. PEEK : Used to return value of the topmost positioned element of stack without deleting it.

Algo

1: If TOP = NULL
   Print Stack empty
   Goto 3
2: Return Stack [TOP]
3: End.

* Program to perform PUSH, POP, PEEK.

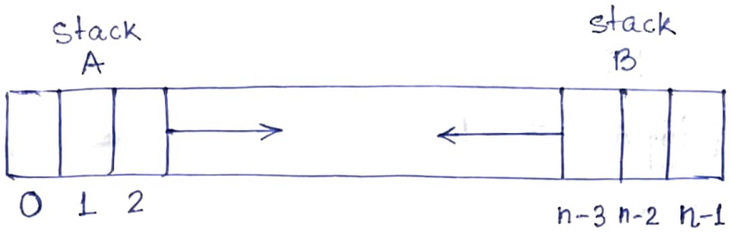
→

```
#include <stdio.h>                    [P]
#include <conio.h>
#include <stdlib.h>
#define MAX 3
int stack [MAX], TOP = -1;
void PUSH (int stack [], int value)
{
    if (TOP == MAX-1)
    {
        printf ("\n Stack overflow! ");
    }
    else
    {
        Top ++;
        stack [Top] = value;
    }
}
```

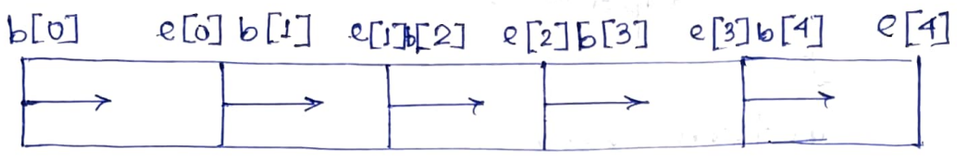
```
int POP (int stack[])
{
        int value;
        if (top == -1)
        {
            printf ("\n Stack underflow!");
            return -1;
        }
        else
        {
            value = stack [TOP];
            TOP --;
            return value;
        }
}
void display (int stack[])
{
    int i;
    if (TOP == -1)
        printf ("\n Stack empty!");
    else
    {
        for (i=TOP; i>=0; i--)
        {
            printf ("\n %d", stack[i]);
        }
    }
}
int peek (int stack[])
{
    if (TOP == -1)
      { printf ("Stack empty!");
        return -1; }
    else {
        return (stack [TOP]); }
}
```

```
int main (
{
    int value, option;
    do
    {
        printf ("1. PUSH");
        printf ("2. POP");
        printf ("3. PEEK");
        printf ("4. DISPLAY");
        printf ("5. EXIT");
        printf ("Enter option - ");
        scanf ("%d", &option);
        switch (option)
        {
            case 1: printf ("Enter element: ");
                    scanf ("%d", &value);
                    PUSH (stack, value);
                    break;
            case 2: value = POP (stack);
                    if (value != -1)
                        printf ("%d deleted", value
                    break;
            case 3: value = peek (stack);
                    if (value != -1)
                        printf ("In %d at top", value)
                    break;
            case 4: display (stack);
                    break;
        }
    } while (option != 5)
    return 0;
}
```

* Multiple Stack :



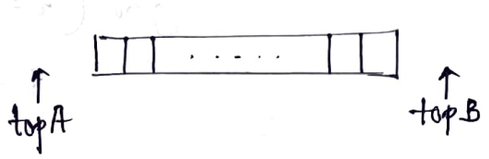
- Double Stack – n will be greater than the size of both the stacks. ~~with~~ Stack A & stack B grow from different directions.

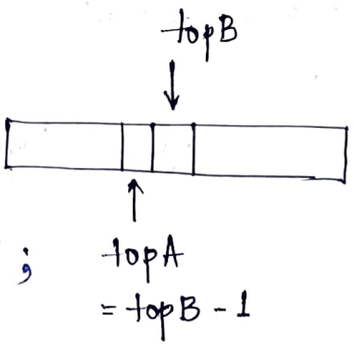


A stack can be used to represent n number of stacks in the same array. Each stack i will be allocated an equal amount of space bounded by indices b[i] & e[i].

* Program to implement multiple stack.

→
```
#include <stdio.h>
#include <conio.h>
#define MAX 10
int stack [MAX] , topA = -1, topB = MAX ;
void pushA (int val)
{
    if (topA == topB - 1)
        printf ("\n Overflow) ;
    else
    {
        topA += 1 ;
        stack [topA] = val ;
    }
}
```





```
int popA ()
{
    int val;
    if (topA == -1)
    {
        printf ("\n Underflow");
        val = -999;
    }
    else
    {
        val = stack [topA];
        topA--;
    }
    return val;
}
void displayA ()
{
    int i;
    if (topA == -1)
        printf ("\n Stack A empty");
    else
    {
        for (i = topA; i >= 0; i--)
            printf ("\t %d", stack[i]);
    }
}
void pushB (int val)
{
    if (topB - 1 == topA)  // topB == topA + 1
        printf ("\n Overflow");
    else
    {
        topB -= 1;
        stack [topB] = val;
    }
}
```

```
int popB()
{
    int val;
    if (topB == MAX)
    {
        printf("\n Underflow");
        val = -999;
    }
    else
    {
        val = stack[topB];
        topB++;
    }
}
void displayB()
{
    int i;
    if (topB == MAX)
        printf("\n stack B empty");
    else
    {
        for(i= topB; i<MAX; i++)
            printf("\t %d", stack[i]);
    }
}
void main()
{
    int option, val;
    clrscr;
    do {
        printf("PUSHA-1, PUSHB-2, POPA-3,
                POPB-4, DISA-5, DISB-6, Exit-7");
        printf("\n Enter choice - ");
        scanf("%d", &option
        switch operation ——— } while(option != 7)
}
```

* Application of Stacks.

1. Reversing a list :

P

Read each # from array & push into stack. Then pop one by one & store @ array.

```
int stk[10];
int top = -1;
int pop(); void push(int);
int main()
{
    int val, n, i;
  int arr[10];
    printf("\n Elements No:");
    scanf("%d", &n);
    printf("\n Elements: ");
    for(i=0; i<n; i++)
        scanf("%d" &arr[i]);
    for(i=0; i<n; i++)
        push(arr[i]);
    for(i=0; i<n; i++)
    {
        val = pop();
        arr[i] = val;
    }
    printf("\n The reversed array: ");
    for(i=0; i<n; i++).
        printf("\n %d", arr[i]);
    return 0;
}
void push(int val)
{
    stk[++top] = val;
}
int pop()
{
    return (stk[top--]);
}
```

2. Implementing Parentheses Checker:

{A + (B + C)} (A + B}

VALID INVALID.

Push all left paranthesis into stack. When a right paranthesis is encountered, try to match with an eqv. left paranthesis. If matched, then valid till that. At end, empty stack if valid.

```
#define MAX 10
int top = -1;
int stk [MAX];
void push (char);
char pop ();
int main ()
{
    char exp [MAX], temp;
    int i, flag = 1; // flag to check if valid or not
    printf ("Enter expression: ");
    gets (exp);
    for (i=0; i<strlen (exp); i++)
    {
        if (exp[i] == '(' || exp[i] == '{' || exp[i] == '[')
            push (exp [i]); // push (, {, [
        if (exp[i] == ')' || exp [i] == '}' || exp[i] == ']')
        {
            if (top == -1)  // if empty stack, invalid
1. A) rejected ------>  flag = 0;
            else  // if non-empty stack
            {
                temp = pop(); // pop & try to match
                if (exp[i] = ')' && (temp == '{' ||
                                     temp == '['))
                    flag = 0;
                if (exp[i] = '}' && (temp = '(' ||
                                     temp == '['))
                    flag = 0;
                if (exp[i] == ']' && (temp == '(' ||
                                      temp == '{'))
                    flag = 0;
            }
        }
```

2. (A rejected)

```
    }
    if (top >=0) //if non empty stack
        flag = 0;
    if (flag == 1)
        printf ("\n Valid");
    else
        printf ("\n Invalid");
    return 0;
}
void push (char c)
{
    if (top == (MAX -1))
        printf ("\n Overflow");
    else
    {
        top+= 1;
        stk [top] = c;
    }
}
Char pop
{
    if (top == -1)
        {printf ("\n Underflow");
        return (-999); }
    else
    {
        return stk [top--];
    }
}
```

3. Evaluation of Arithmetic Expressions :

- **Infix** – operand operator operand.
- **Postfix** – (Developed by Jan Łukasiewicz, Polish logician) . Polish Notation ↓ prefix.

  ↓

  Reverse Polish Notation. (RPN).

  operand operand operator. ⟶ left to right evaluation
- **Prefix** –

  operator operand operand ⟶

Eg. Infix $(A+B)/(C+D)-(D*E)$.

Postfix $[AB+]/[CD+]-[DE*]$

$[AB+CD+/]-[DE*]$

$\boxed{AB+CD+/DE*-}$

Prefix $\boxed{-/+AB+CD*DE}$

Algo [Infix ⟶ Postfix] Operator stack

1. Add ')' to the end of infix.
2. Push '(' to the stack.
3. Repeat until each character in the infix is scanned.
   - 3a) If a '(', push it to stack.
   - 3b) If an operand add to postfix.
   - 3c) If a ')' is encountered,
     - i) Repeatedly pop from stack & add to postfix until a '(' is encountered.

ii) Discard the '(' from stack & do not push to postfix

3d) If an operator ø is encountered, then

i) Repeatedly pop from stack & add each operator to the postfix that has the same precedence than ø.

ii) Push ø to the stack.

4. Repeatedly pop from the stack & add it to the postfix expression until the stack is empty.

5. Exit.

Algo [Evaluation of Postfix]

1. Add a ')' at the end of postfix.
2. Scan every character of postfix & repeat (3) & (4) until ')' is encountered.
3. If an operand is encountered, push to stack.

If an operator O is encountered, then

a) Pop top two elements from the stack as A & B and evaluat B O A, where A is the topmost element & B below A.

b. Push the result on the stack.

4. Set result equal to the topmost element of the stack.
5. Exit.

* Expression evaluation.

Most of the compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

✓ Most programming languages are context-free languages allowing them to be parsed with stack based machines.

✓ Note: natural languages are context sensitive & stacks alone are not enough to interpret their meaning.

- Operator precedence in arithmetic exp^{n}s:

B Bracket
O Order or power
D Division
M Multiplication
A Addition
S Subtraction

- Evaluation of arith. exp^n in computers:
  1. Infix → Postfix conversion
  2. Evaluating postfix exp^n

Evaluation of infix:

1. Read one character.
2. Actions at end of each i/p:
   a) Opening brackets: Push into stack & goto (1)

(b) Number : Push into stack & goto (1).

(c) Operator: Push into stack & goto (1).

(d) Closing brackets : Pop from stack.
(popped char → c)

i. If c is closing bracket discard, goto (1).

ii. Pop four times otherwise.

First pop operand$_2$

Second pop operator

Third pop operand$_1$

Fourth popped elem as assigned the remaining

Opening bracket : Discarded.

Evaluate op_1 op op_2

Push result into stack & goto d.

(e) Newline : pop from stack & print answer.

eg. (((2*5)-(1*2))/(11-9))\n

Symbol	Stack
(	(
(	((
(	(((
2	(((2
*	(((2*
5	(((2*5
)	((10
-	((10-
(	((10-(
1	((10-(1
*	((10-(1*
2	((10-(1*2
)	((10-2
)	(8
/	(8/
(	(8/(
11	(8/(11
-	(8/(11-
9	(8/(11-9
)	(8/2
)	4
\n	pop ↓ 4

- Another algorithm : Using 2 stacks.

Approach : Use 2 stacks, operand & operator stack

Process :

1. Pop out 2 values from operand stack.
2. Pop out operator from operator stack.
3. Evaluate & push result to the operand stack.

Algorithm :

Iterate through expression, one character at a time.

1. If character is an operand, push it to the operand stack.
2. If character is operator,
   i) If operator stack is empty, push it to operator stack.
   ii) Else if operator stack is not empty,
      - if character's precedence is greater than or equal to the precedence of the stack top of operator stack, then push char. to operator stack.
      - otherwise pop 2 value from operand stack, pop out 1 operator from operator stack

& evaluate until character's precedence is less or stack is not empty. Then push result to operand sta

3. If character is '(', push into operat stack.

4. If character is ')', then do process until the corresponding '(' is encountered in operator stack. Now, pop the '('.

Once the expression is iterated fully, if stack is not empty, do process until operator stack is empty. Value left at operand stack is the final result.

TC O(n)

eg. 2 * (5 * (3 + 6)) / 15 - 2

Token	Op^nd	Op^r
2	2	
*	2	*
(	2	*(
5	25	*(
*	25	*(*
(	25	*(*(
3	253	*(*(
+	253	*(*(+
6	2536	*(*(+
)	259	*(*
)	245	*

Token	Op^nd	Op^r
/	245	*/
15	24515	
-	23	*
	6	-
2	62	-

4

◐ Infix to postfix conversion.

Approach: Use an operator stack.

Algorithm:

Initialize result as a blank string. Iterate through given expression, one character at a time:

1. If the character is an operand, add it to the result.

2. If the character is an operator,

- If the operator stack is empty then push it to the stack.
- Else if the operator stack isn't empty,
  - If the operator's precedence is $\geq$ to that of the stack top of the stack, then push to the operator stack.
  - If the operator's precedence is $<$ that of the stack top then pop out an operator from the stack & add it to the result until the stack is empty or operator's precedence is $\geq$ to that of the stack top. At last, push the operator to stack.
    - If top = (, push the operator.

3. If character is '(', push

4. If ')', then pop an operator & add to result until corresponding '(' is encountered. Now, pop out the '('.

Once the expression iteration is completed & the operator stack is not empty, pop out an operator from the stack & add to the result until the operator stack is empty. Result is an

eg. A + B * (C ^ D - E)

Token	Stack	Result
A		A
+	+	
B	+	AB
*	+ *	AB
(	+ * (	AB
C	+ * (	ABC
^	+ * (^	ABC
D	+ * (^	ABCD
-	+ * (	ABCD^
	+ * (-	
E	+ * (-	ABCD^E
)	+ *	ABCD^E-

ABCD^E-*+

◐ Evaluation of postfix expression

Approach: Use operand stack.

Algorithm:

Iterate through given expression, one character at a time.

1. If the character is a digit, initialize number = 0.
   - While the next character is digit, do number = number * 10 + digit
   - Push number to stack.
2. If the character is an operator,
   - Pop operand from stack, say op1
   - Pop operand from stack, say op2
   - Perform op2 operator op1 & push.
3. Once the expression iteration is completed, Stack will have the final result.

eg. 20 50 3 6 + * * 300 / 2 -

Token	Stack
20	20
50	20 50
3	20 50 3
6	20 50 3 6
+	20 50 9
*	20 450
*	9000
300	9000 300
/	30
2	30 2
-	28

Evaluation of prefix expressions

Operand stack

Algorithm: Reverse the expression & iterate.

1. If character is an operand, push it to stack.
2. If character is an operator,
   i) Pop operand, op1
   ii) Pop operand, op2
   iii) Perform op1 operator op2 & push.
3. Once iteration is complete, stack has the result.

Infix to prefix conversion

Operator stack

Algorithm:

1. Reverse expression.
2. Infix to postfix.
3. Reverse to get prefix.

eg. Infix A + B * (C ^ D - E)

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

↓ Reverse brackets

(E - D ^ C) * B + A

Inf → Postf E D C ^ - B * A +

Reverse for prefix + A * B - ^ C D E.

● Prefix to postfix conversion — Operand Stack

Iterate in reverse.

1. If operand, push.
2. If operator,
   pop op1, pop op2, push (op1 op2 op)
3. Once iteration is complete, initiatize the result string & pop out from the stack & add it to the result.

eg. Prefix : * - A / B C - / A K L

Token	Stack
L	L
K	LK
A	LKA
/	L (AK/)
-	(AK/L-)
C	(AK/L-) C
B	(AK/L-) CB
/	(AK/L-) (BC/)
A	(AK/L-) (BC/) A
-	(AK/L-) (ABC/-)
*	ABC/-AK/L-*

Prefix to infix.

Operand stack

Iterate exp^n in reverse order,

1. If character is operand, push.
2. If character is operator,
   i) Pop op1
   ii) Pop op2
   iii) Perform (op1 operator op2) & push.
3. Once iteration is completed, initialize result string & pop out from stack & add to the result.

eg. Prefix * - A / B C - / A K L ←

Token	Stack
L	L
K	L K
A	L K A
/	L (A/K)
-	((A/K) - L)
C	((A/K) - L) C
B	((A/K) - L) C B
/	((A/K) - L) (B/C)
A	((A/K) - L) (B/C) A
-	((A/K) - L) (A - (B/C))
*	((A - (B/C)) * ((A/K) - L))

● Postfix to infix : Operand stack

Iterate, one char at a time,

1. If operand, push.
2. If operator,
   pop op1, pop op2, push (op2 op op1)
3. Once iteration is complete, initialize the result string & pop out & add to result.

eg. Postfix : A B C /- A K / L - *

Token	Stack
A	A
B	AB
C	ABC
/	A (B/C)
-	(A - (B/C))
A	(A - (B/C)) A
K	(A - (B/C)) A K
/	(A - (B/C)) (A/K)
L	(A - (B/C)) (A/K) L
-	(A - (B/C)) ((A/K) - L)
*	((A - (B/C)) * ((A/K) - L))

● Postfix to prefix Operand stack

Iterate

1. If operand, push.
2. If operator,
   pop op1, pop op2, push (op op2 op1)
3. Once iteration is complete, initialize the result string & pop out from the stack & add it to the result.

* Sort stack using temporary stack :

1. While input stack is not empty, do
   i) Pop from input stack, say temp.
   ii) While temporary stack is not empty & top of temporary stack is > temp, pop from temporary stack & push into the input stack.
   iii) Push temp into temporary stack.
2. Sorted numbers in temporary stack.

eg. (34, 3, 31, 98, 92, 23)

Elem. popped from input stack	Input	Temp. Stack
23	34, 3, 31, 98, 92	23
92	34, 3, 31, 98	23, 92
98	34, 3, 31	23, 92, 98
31	34, 3, 98, 92	23, 31
92	34, 3, 98	23, 31, 92

Popped elem.	Input	Temp. stack.
98	34, 3	23, 31, 92, 98
3	34, 98, 92, 31, 23	3
23	34, 98, 92, 31.	3, 23
31	34, 98, 92	3, 23, 31
92	34, 98	3, 23, 31, 92
98	34	3, 23, 31, 92, 98
34	98, 92	3, 23, 31, 34
92	98	3, 23, 31, 34, 92
98	[]	3, 23, 31, 34, 92, 98

* Implement stack using queues.

S, q_1, q_2

Method 1. Newly entered element is always at the front of q_1, so that pop just dequeues from q_1.

Enqueue - to rear
Dequeue - at front

push (s, x)

1. enqueue x to q_2.
2. One by one dequeue everything from q_1 & enqueue to q_2.
3. Swap names of q_1 & q_2. (to avoid one more movement of all elements from q_2 to q_1)

pop(s)

1. Dequeue an item from q_1 & return.

Method 2:

New elem enqueued to q_1. In pop(), if q_2 is empty then all the elems except the last, are moved to q_2. Finally last elem is dequeued from q_1 & returned.

push (s, x)

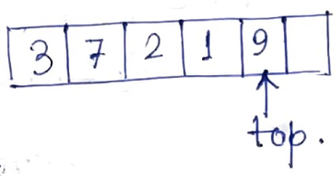
Enqueue x to q_1.

pop

1. One by one dequeue everything except the last elem from q_1 & equeue to q_2.
2. Dequeue last elem of q_1 & it is the result.
3. Swap names $q_1 \leftrightarrow q_2$.
4. Return item of step 2.

NS

- **Application of Stack.** : Balancing of symbols, infix-postfix conversion, evaluation of postfix expression, implementing function calls, finding of spans (spans in stock markets), page visited history in web browser (back buttons), undo sequence in text editor, matching tags in HTML & XML, parsing, backtracking.
- **Simple Array Implementation.** S



Code

```
typedef struct ArrayStack {
        int top;
        int capacity;
        int *array;
} Stack;

Stack *CreateStack() {
        Stack *S = (Stack*) malloc (sizeof(Stack));
        if (!S)
            return NULL;
        S → capacity = 1;
        S → top = -1;
        S → array = (int*) malloc (S → capacity * sizeof(int));
        if (!S → array)
            return NULL;
        return S;
}
int IsEmptyStack (Stack *S) {
        return (S → top == -1);
}
```

```
int IsfullStack ( Stack *S) {
    return ( S->top == S->capacity - 1);
}
void Push ( Stack *S , int data) {
    if (InfullStack (S))
        printf ("Stack overflow!");
    else {
        (S->top)++;
        S->array [S->top] = data;
    }
}
```

```
S->array [++S->top]
    = data ;
```

```
int Pop (Stack *S) {
    if (IsEmpty Stack(S)) {
        printf ("Stack Underflow!");
        return 0;
    }
    else {
        ~~(S->top)++;~~
        return S->array [S->top];
        (S->top)--;
    }
}
```

```
return (S->array [S->top--]);
```

```
void DeleteStack ( Stack *S) {
    if (S) {
        if (S->array)
            free ( S->array);
        free (S);
    }
}
```

```
void Display (Stack *S)
{
    int i;
    if ( S->top != -1) {
    for ( i=S->top; i >= 0; i--)
    {printf ("%d", S->array[i]);
    }
    }
}
```

* Performance.

→ Space Complexity (for n push) - O(n)

→ TC of push, pop, isempty, isfull, delete - O(1).

* Limitations.

The maximum size of the stack must be defined in prior & cannot be changed. Trying to push a new element into a full stack causes an implementation-specific exception.

• Dynamic Array Implementation.

By array doubling technique; if the array is full, create a new array of twice the size & copy items then push.

For n push operations we double the array size logn times.

T(n) for n push operations =

$$1 + 2 + 4 + 8 + \cdots + \frac{n}{4} + \frac{n}{2} + n$$

$$= n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{4}{n} + \frac{2}{n} + \frac{1}{n}\right).$$

$$= 2n = O(n).$$

Amortised time of push operation is O(1).

[DS2,]
Intro

Code

```
typedef struct DynArrayStack {
    int top;
    int capacity;
    int *array;
} Stack;

Stack *CreateStack() {
    Stack *S = (Stack *) malloc (sizeof (Stack));
    if (!S)
        return NULL;
    S→capacity = 1;
    S→top = -1;
    S→array = (int *) malloc (S→capacity * sizeof (int));
    if (!S→array)
        return NULL;
    return S;
}
int IsFullStack( Stack *S) {
    return ( S→top == S→capacity - 1);
}
int IsEmptyStack ( Stack *S) {
    return ( S→top == -1);
}
Void DoubleStack (Stack *S) {
    S→capacity *= 2;
    S→array = (int *) realloc (S→array, S→capacity);
}
```

```
void Push ( Stack *S , int data) {
    if (IsFullStack(S))
        DoubleStack (S);
    S → array [++S→top] = data;
}
void Peek( Stack *S) {
    if (IsEmptyStack (S))
        printf ("Underflow");
    printf ("Top element - %d", S→array [S→top]);
}
int Pop ( Stack *S) {
    if (IsEmptyStack (S))
        return INT_MIN;
    return S→array [S→top--];
}
void DeleteStack ( Stack *S) {
    if (S) {
        if (S→array)
            free (S→array);
        free (S);
    }
}
void Display ( Stack *S) {
    int i;
    if (S→top != -1) {
        for ( i = S→top ; i >= 0 ; i--)
            printf ("%d\t", S→array[i]);
    }
}
```

* Performance.

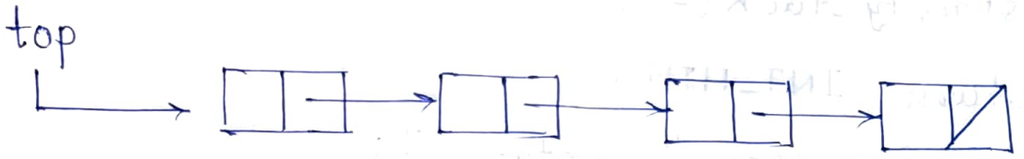
Space complexity (for n push) - O(n).

TC of CreateStack, push, pop, peek, isempty, isfull, deletestack - O(1)

→ Too many doublings may cause memory overflow exception.

- Linked List Implementation.

Push is implemented by inserting element at the front of the list. Pop is implemented by deleting from front.



Code

```
typedef struct ListNode {
    int data;
    struct ListNode *next;
} Stack;

Stack *CreateStack () {
    return NULL;
}
void Push ( Stack **top, int data) {
    Stack *temp;
    temp = (Stack *) malloc ( sizeof (Stack));
    if (!temp){
        return ;
    }
```

```
    temp -> data = data;
    temp -> next = *top ;      // Imp.
    *top = temp;
}
int IsEmptyStack (Stack *top) {
    return top == NULL;
}
int Pop ( Stack **top) {
    int data;
    Stack *temp;
    if (IsEmptyStack (*top))
        return INT_MIN;
    temp = *top;
    *top = *top -> next;
    data = temp -> data;
    free (temp);
    return (data);
}
int Peek ( Stack *top) {
    if (IsEmptyStack ( top))
        return INT_MIN;
    return top -> data;
}
void DeleteStack (Stack **top) {
    Stack *temp, *p;
    p = *top;
    while (p -> next) {
        temp = p -> next;
        p -> next = temp -> next;
        free (temp);
    }
```

```
    free(p);
}
void Display (Stack **top) {
    Stack *temp = *top;
    while (temp != NULL) {
        printf ("%d\n", temp->data);
        temp = temp->next;
    }
}.
```

* Performance.

Space complexity for n push - O(n).

TC for create stack, push, pop, peek, isempty, delete - O(1).

Comparison of Implementations.

→ Incremental Strategy. -

The amortized time (average time per operation) of a push operation is O(n).

→ Doubling strategy -

The amortized time of a push operation is O(n)/n or O(1).

→ Array Implementation. -

- Operations take constant time.
- Expensive doubling operation.
- Amortised bound takes time proportional to n.

→ Linked List Implementation.

- Every operation takes constant time.
- Every operation takes extra space & time to deal with references

Q. Let the min. no. of stacks reqd. to evaluate a prefix expression is A & the value of the prefix expression --+2*34+/8215 evaluated using the same # of stacks is B. A, B = ?

→

--+2*34+/8215

--+2 12 + 4 1 5

-- 14 5 5

- 9 5

4.

1 stack (operator stack) ↓ or reverse ↓ operand stk

{ A = 1
B = 4

DS1 (code, Th). DS2 (Adv)

Stack

* LIFO / FILO data structure.
* Stack ADT –

  Operations – push, pop

  top, size, isempty, isfull.

* Applications.
* Implementation – array,

  Linked list.

* Comparison of Implementations.
* Implementation of Recursion.

Stack is used for recursive calls.

Languages like C, Pascal that have a dynamic memory management mechanism can ~~have~~ directly accept the recursive definition of procedures; compilers of these languages are responsible to produce an object code suitable for execution using a stack. (called run time stack). In other languages like Fortran, COBOL, BASIC which do not have a dynamic memory management mechanism, it is the user's responsibility to define & maintain the stack in order to implement the recursive definition of the procedure.

eg.

Steps

1 5! = 5 x 4!
2 4! = 4 x 3!
3 3! = 3 x 2!
4 2! = 2 x 1!
5 1! = 1 x 0!
6 0! = 1
7 1! = 1
8 2! = 2
9 3! = 6
10 4! = 24
11 5! = 120.

Here, it is required to push the intermediate calculations till the terminal condition is reached. 1 to 6. steps are the push operations. Then subsequent pop operations will evaluate the value of intermediate calculations till the stack is exhausted.

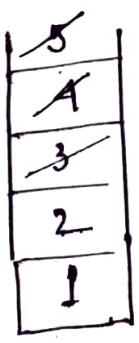
To control the recursion we have to maintain the following stacks:

i) Stacks for parameters
ii) Stack for local variables
iii) Stack to hold return address.

G'94. Which of the following permutations can be obtained. in the o/p (in the same order) using a stack if i/p sequence is 1,2,3,4,5 ?

a) 3 4 5 1 2
b) 3 4 5 2 1
c) 1, 5 2 3 4
d) 5 4 3 1 2

→ a) 3 1 5 ?

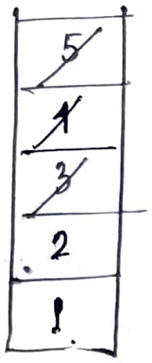


b) ✓

c)

1 5 ?

d)



5 4 3 ?

G'97 A priority queue Q is used to implement a stack S that stores chars. Push(c) is implemented as Insert (Q, c, K) where K is an integer key chosen by the implementation. Pop() is implemented as DeleteMin(Q). For a sequence of Push operations, the keys chosen are in

a) non-decreasing
b) non-increasing order.
c) strictly inc.
d) strictly dec. order.

G'03 Let S be a stack of size $n \geq 1$. Starting with the empty stack, suppose we push the first n natural numbers in sequence & then perform n pop operations. Push & pop take x seconds each & y secs elapse b/w the end of one such stack op^n & the start of next op^n. For $m \geq 1$, define the stack-life of m as the time elapsed from end of push(m) to the start of pop that removes m from S. The avg stack lifetime

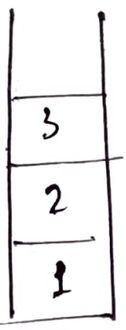
of an elem is:

$y + x + y + x + y$

$= 2x + 2y + y$

$$avg = \frac{2x + 2y + y + y}{2}$$

$$= x + 2y$$



$$\boxed{2(n-1)x + 2(n-1)y + y}$$

~~avg~~ $$\Sigma = 2\left(\sum_{i=1}^{n-1} i\right)x + 2\left(\sum_{i=1}^{n-1} i\right)y + ny$$

$$avg = \frac{\Sigma}{n}$$

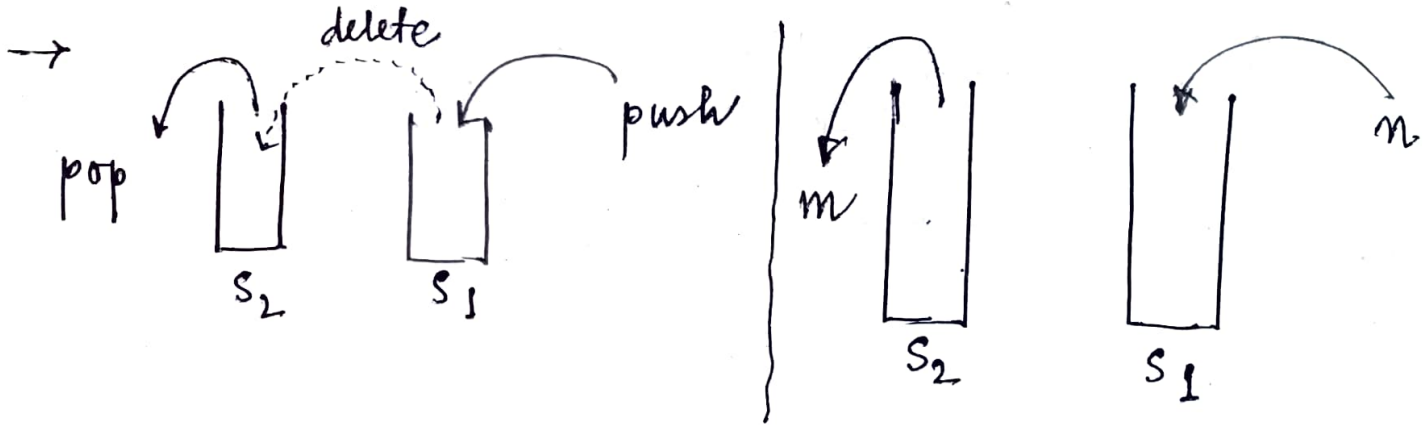
$$\begin{cases} avg = (n-1)x + (n-1)y + *y \\ = (n-1)(x+y) + *y \end{cases}$$

Q'ns Q is implemented using 2 stacks S_1 & S_2 as given below:

```
void insert (Q, x){
    push (S1, x);
}
void delete (Q) {
    if (stack_empty (S2)) then {
        if (stack_empty (S1)) then {
            printf ("Q empty");
            return;
        }
        else { while (! stack_empty (S1)) {
                x = pop (S1); push (S2, x)
            }
        }
        x = pop (S2);
}
```

n inserts, m ($\leq n$) deletes

# of push, pop ? (min, max)
x y



Best case push: Elems go to S_1 then S_2, for m deletes thereby popped from S_2.

1 push to S_1 − 1 push to S_2

$2m$ push + $(n-m)$ pushes = $(m+n)$ pushes

{ m pushes to S_1, m pushes to S_2 } remaining $n-m$ elems

Worst case pushes $2n$

n push to S_1
n push to S_2

Best case pop:

$2m$ pops — each elem is pushed to S_1 then popped to S_2 (pushed to S_2 then popped from S_2

Worst case pops = $(n+m)$

n pops from S_1
m pops from S_2

$m+n \leq x \leq 2n$
$2m \leq y \leq n+m$

G'00 Suppose a stack implementation supports in addition to push, pop, an operation that reverses the order of the elems on the stack. Implement a queue using the above stack impl^{n}. Show how to implement enqueue using a single op^{n} & dequeue using a sequence of 3 op^{n}s.

enqueue : push

dequeue : reverse, pop, reverse

reverse, push, reverse
pop

G'07

```
#define EOF -1.
void push(int);
int pop(void);
void flagerror();
int main() {
    int c, m, n, r;
    while ((c = getchar()) != EOF) {
        if (isdigit(c)) push(c);
        else if ((c == '+') || (c == '*')) {
            m = pop();
            n = pop();
            r = (c == '+') ? n + m : n * m;
            push(r);
        }
        else if (c != ' ')
            flagerror();
    }
```

Ans - 25.

o/p for 52*332+*+.

G'05. A function f defined on stacks of integers satisfies following properties : $f(\phi) = 0$ & $f(\text{push}(S, i)) = \max(f(S), 0) + i$ for all stacks S & integers i. If a stack S contains the integers 2, -3, 2, -1, 2 in order from bottom to top, what is $f(S)$?

i	$f(S)$
2	$f(\phi) = 0$
-3	$\max(0, 0) + (2) = 2$
2	$\max(2, 0) + (-3) = -1$
-1	$\max(-1, 0) + (2) = 2$
2	$\max(2, 0) + (-1) = 1$
	$\max(1, 0) + 2 = 3$

* # of valid stack permutations of n distinct elems $= C_n = \frac{1}{n+1}\binom{2n}{n}$

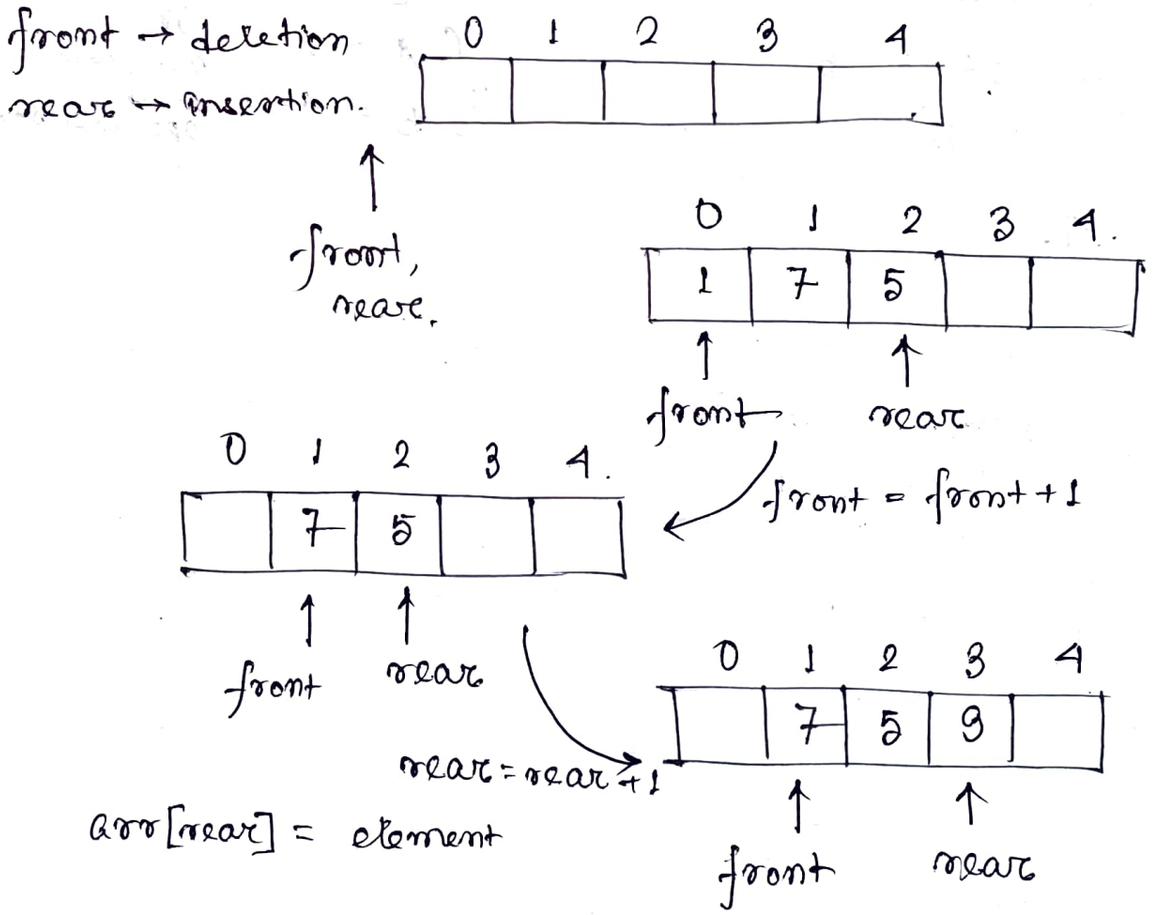
$\Rightarrow$ # invalid permutations $= n! - C_n$

DS1 (Th, Code) DS2 (Adv.)

The Queue.

* A queue is FIFO type (first in first out) linear data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the REAR. & removed from the other end called front.

* Array Implementation of Queue:



- Operations on Queue:

1. enqueue: Insertion of element to the rear,

Algo.

Step 1: If rear = MAX - 1

Write Overflow

Exit

Step 2: If front = -1 and rear = -1

Set front = rear = 0

else

Set rear = rear + 1

Step 3: Set queue [rear] = num

Step 4: Exit.

2. dequeue: Deletion of element from front.

Algo.

Step 1: If front = -1 or rear < front

Write underflow.

Else

Set val = queue [front]

Set front = front + 1

Step 2: Exit.

rear ≥ front

3. Peek:

Algo Step 1: If front == -1 or front > rear

Write Underflow

Else

return queue [front]

Step 2: Exit.

4. display :

Algo.

Step 1 : If front == -1 or front > rear

Print Underflow

else

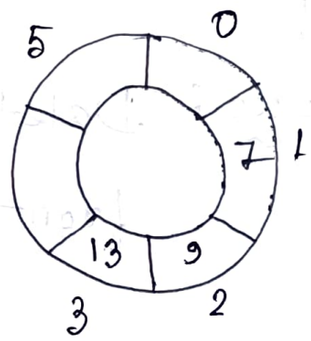
for (i = front to rear, i++)

print queue [i]

Step 2 : Exit.

* Types of Queues :

1. Circular Queue :

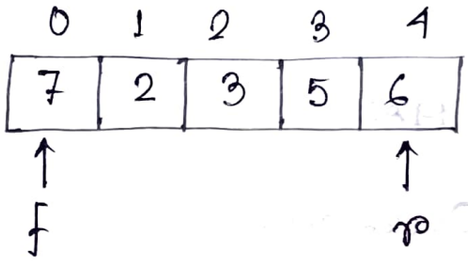


Here the last node is connected to the first node to make a circle. In circular queue if we reach the end for inserting elements to it, it is possible to insert new elements if the nodes at the beginning are empty.

- If front = 0, rear = MAX-1, the queue is full.
- If rear != MAX-1, then rear will be incremented & the value will be inserted.
- If rear = MAX-1 & front != 0 then queue is not full. Set rear = 0 & insert there.

MAX = 5.

full



front == 0

rear == MAX-1.

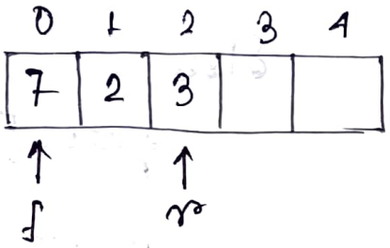
count = 5

insert @ rear

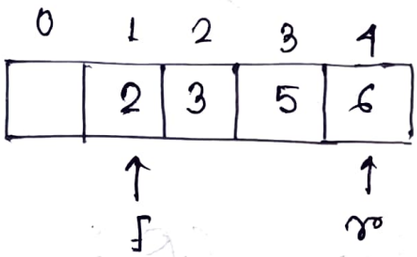
front == 0

rear != MAX-1

count = 3



Wrap around.



front != 0

rear = MAX-1.

count = 4.

- Insertion of element in circular queue:

Algo

Step 1: If front = 0 & rear = MAX-1 // full

Write Overflow

Goto step 4.

Step 2: If front = -1 & rear = -1. // empty

Set front = rear = 0.

Else if rear = MAX-1 & front != 0

Set rear = 0

Else

Set rear = rear + 1.

Step 3: Set queue [rear] = val

Step 4: Exit.

Better algo.

Step 1: If count = MAX

Print Overflow.

Else

queue[rear] = value

rear = (rear + 1) % MAX.

count++

Step 2: Exit.

- Deletion of element from CQ:

Algo Step 1: If count = 0

Print Underflow

Else

~~Step 2:~~ front = (front + 1) % MAX

count--

Step 2: Exit.

- Program to implement circular queue using array.

→

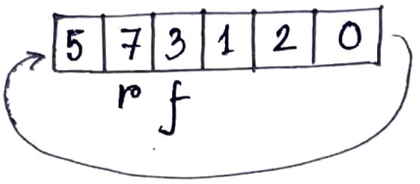
```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5
int queue[MAX];
int front = 0;
int rear = 0;
int count = 0;
```

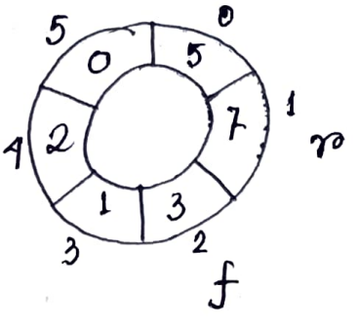
```
void enqueue()
{
    int element;
    if (count == MAX)
    {   printf ("\nOverflow\n");
    }
    else
    {
        printf ("\n Enter element : ");
        scanf ("%d", &element);
        queue [rear] = element;
        rear = (rear + 1) % MAX;
        count ++;
    }   display();
}
void dequeue()
{
    if (count == 0)
    {   printf ("\n Underflow\n");
    }
    else
    {
        front = (front + 1) % MAX;
        count--;
    }
    display();
}
void display()
{
    int i, j;
    if (count == 0)
    {   printf ("\n Underflow ! \n");
    }
    else {
        printf ("\n Queue is : ");
        j = count;
        for (i = front; j != 0; j--)
        {   printf ("%d\t", queue [i]);
            i = (i+1) % MAX;
        }
    } }
```

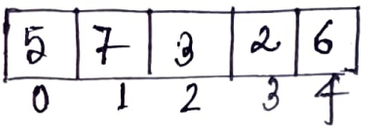
```
void peek()
{
    if (count == 0)
    {
        printf ("\n Underflow!");
    }
    else
    {
        printf ("\n Element @ front: %d \n",
                                    queue[front]);
    }
}
int menu()
{
    int choice;
    printf ("1. enq 2. deq 3. peek 4. disp 5. exit");
    scanf ("%d", &choice);
    return choice;
    printf ("\n");
}
int main(void){
    int choice;
    do{
        choice = menu();
        switch (choice)
        {
            case 1: enqueue();
                    break;
            case 2: dequeue();
                    break;
            case 3: peek();
                    break;
            case 4: display();
                    break;
            case 5: exit(1);
            default: printf ("\n Invalid choice \n");
        }
    } while(1);
    return 0;
}
```

⊙ Circular queue using array

```
→ Void enqueue (int x) {
    if (f == -1 && r == -1) {
        f = r = 0;
        queue[r] = x;
    }
    else if (f == (r+1) % N)
        queue is full
    else {
        r = (r+1) % N;
        queue[r] = x;
    }
}
```





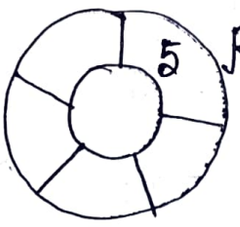


f = 0 r = 4

full when f == (r+1) % N

0 == (4+1) % 5

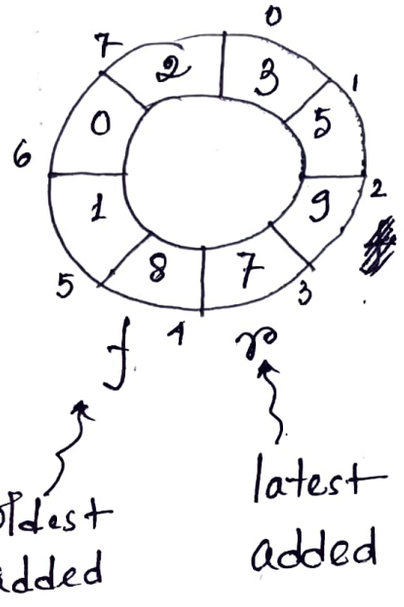
```
→ void dequeue () {
    if (f == -1 && r == -1)
        queue is empty
    else if (f == r) { // 1 elem
        f = r = -1;
    }
    else {
        f = (f+1) % N;
    }
}
```



→ void display() {

```
int i = f;
if (f == -1 && r == -1)
    empty queue
else {
    while (i != rear) {
        printf ("%d", queue[i]);
        i = (i+1) % N;
    } printf ("%d", queue[rear]);
}
}
```

Start from front of queue & go till rear.



Queue using one stack (using call stack)

enqueue(x)
push x to stack

dequeue
- if stack empty, then error
- if stack has only one elem, return it.
- & otherwise, recursively pop everything from stack, store popped item in a variable ret, push ret back to stack & return ret.

(makes sure last popped item is always returned; recursion stops when there's only one item in the stack, so last elem of stack is returned in dequeue)

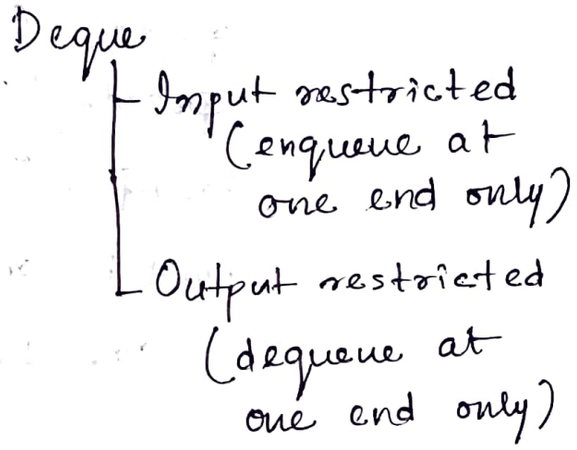
```
Void enqueue(x) {
    stack.push(x);
}
void dequeue() {
    if (stack.empty()) {
        exit(0);
    }
    int x = stack.top();
    stack.pop();
    if (stack.empty())
        return x;

    //recursion
    int ret = dequeue();
    stack.push(x);
    return ret;
}
```

2. Deque (Doubly ended queue)

- A deque is a list in which the elements can be inserted or deleted at either end. It is also known as head-tail linked list, because elements can be added to or removed from either the front (head) or the back (tail) end.

- Program to implement deque:



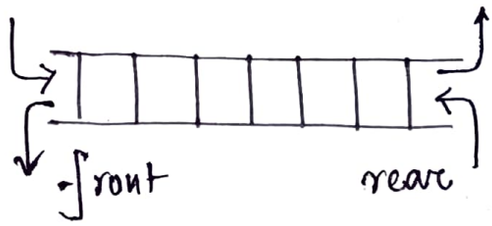

```
#include <stdio.h>
#include <conio.h>
#define MAX 10.
int deque [MAX];
int left = -1, right = -1;
void input_deque (void);
void output_deque (void);
void insert_left (void);
void insert_right (void);
void delete_left (void);
void delete_right (void);
void display (void);
int main ()
{
    int option;
    printf ("\n Main menu\n");
    printf ("\n 1. Input restricted deque \n");
    printf ("\n 2. Output restricted deque \n");
    printf ("\n Enter your option:");
    scanf ("%d", & option);
    switch (option)
    {
        case 1 : input_deque ();
                 break;
        case 2: output_deque ();
                break;
    }
    return 0;
}
```

```
void input_deque()
{
    int option;
    do {
        printf("Input restricted deque\n");
        printf("\n1. Insert at right");
        printf("\n2. Delete from left");
        printf("\n3. Delete from right");
        printf("\n4. Display");
        printf("\n5. Quit");
        printf("\nEnter choice: ");
        scanf("%d", &option);
        switch(option) {
            case 1: insert_right();
                    break;
            case 2: delete_left();
                    break;
            case 3: delete_right();
                    break;
            case 4: display();
                    break;
        }
    } while(option != 5);
}
void output_deque()
{
    int option;
    do {
        printf("Output restricted queue\n");
        printf("1. Insert at right, 2. insert at left,
                3. delete from left 4. display,
                5. Quit\n");
        printf("Enter option: ");
        scanf("%d", &option);
```

```
switch (option) {
    case 1: insert_right();
            break;
    case 2: insert_left();
            break;
    case 3: delete_left();
            break;
    case 4: display();
            break;
}
} while (option != 5);
}
void insert_right()
{
    int val;
    printf("Enter value: ");
    scanf("%d", &val);
    if ((left == 0 && right == MAX-1) || (left == right+1))
    {
        printf("Overflow");
        return;
    }
    if (left == -1) {
        left = 0;
        right = 0;
    }
    else {
        if (right == MAX-1)
            right = 0;
        else
            right = right + 1;
    }
    deque[right] = val;
}
void insert_left() {
    int val;
    printf("Enter value: ");
    scanf("%d", &val);
    if ((left == 0 && right == MAX-1) || (left == right+1))
    {   printf("Overflow"); return;
```

```
    if (left == -1) {
        left = 0; right = 0; }
    else {
        if (left == 0)
            left = MAX-1;
        else
            left = left - 1; }
    deque [left] = val;
}
void delete_left() {
    if (left == -1) {
        printf ("Underflow");
        return; }
    printf ("Deleted: %d", deque [left]);
    if (left == right) {
        left = -1; right = -1; }
    else {
        if (left == MAX-1)
            left = 0;
        else
            left = left + 1; }
}
void delete_right();
void display() {
    int front = left, rear = right;
    if (front == -1) {
        printf ("Queue empty");
        return; }
    printf ("The elements of queue: ");
    if (front <= rear) {
        while (front <= rear) {
            printf ("%d", deque [front]);
            front++; } }
    else {
        while (front <= MAX-1) {
            printf ("%d", deque [front]);
            front++; }
        front = 0;
        while (front <= rear) {
            printf ("%d", deque [front]);
            front++; } }
    printf ("\n");
}
```

* Deque



Supports LIFO, FIFO both. So, deque can be used as stack & queue.

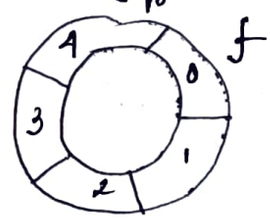
→ Input restricted
→ Output restricted.

→ Applications :
1. Undo-redo
2. Palindrome checker
3. Multiprocessor scheduling

• New operations.

dequeueFront
dequeueRear ✓
enqueueFront ✓
enqueueRear

→ enqueueFront : If f = 0, f ← N-1
dequeueFront : If f = N-1, f ← 0



Deque implementation using Circular queue/array

→ Deque is full when,
f == r + 1 | circular queue
or
f == 0 && r == N-1

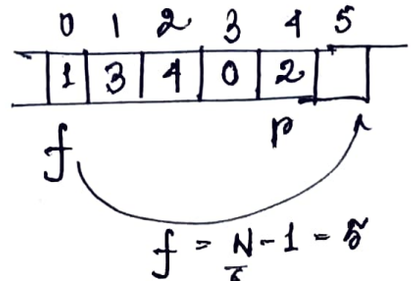
→ dequeueRear : If r = 0, r ← N-1
→ If f = r, only one elem.
If the elem is deleted,
(f = r) ← -1

✓ • void enqueueFront (int x) {

new op^n

if ((f == 0 && r == N-1) || (f == rear && r+1))
— overflow
else if (f == -1 && r == -1) // no elem
f = r = 0;
deque[f] = x;

✓ { else if (f == 0)
f = N-1;
deque[f] = x;



```
else
    f--;
    deque[f] = x;
```

- void enqueueRear (int x) {

```
if ( (f==0 && r = N-1) || (f == r+1))
        - overflow
else if ( f==-1 && r == -1) //no elem
    f = r = 0;
    deque[r] = x;
else if ( r == N-1)
    r = 0;
    deque[r] = a;
else
    r++;
    deque[r] = x;
```

r = (r+1)%N

enqf (a)
enqf (5)
enqr (-1)
enqr (0)
enqf (7)
enqf (4)

} Check with these enqueue op^n's on circular queue.

- void display () {

```
int i = f;
while (i != rear) {
    printf ("%d", deque[i]);
    i = (i+1) % N;
}
printf ("%d", deque[rear]; //to print rear elem.
}
```

- dequeueFront() {

```
if (f==-1 && r==-1)
    - underflow
else if (f == r) // 1 elem
    f = r = -1;
else if (f == N-1)
    f = 0;
else
    f++;
```

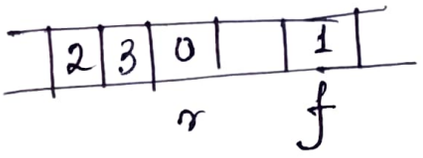
} $f = (f+1) \% N$

}

- dequeueRear() {

```
if (f==-1 && r==-1)
    - underflow
else if (f == r) // 1 elem
    f = r = -1;
else if (r == 0)
    r = N-1;
else
    r--;
```

new opⁿ



Check with

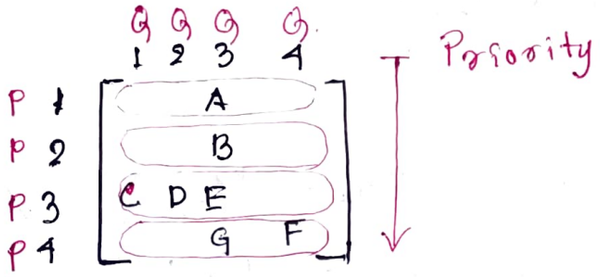
```
deqf();
deqr();
deqf();
```

→ Note: During enqueuefront & dequeueRear (the new op^n's), we do f-- & r-- resp^ly in general case. (contrary to normal enqR & deqF op^ns).

3. Priority Queue

A priority queue is a data structure in which each element is assigned a priority. The priority determines the order in which elements get processed. Two elements with the same priority are processed on a first-come-first-serve (FCFS) basis.

- **Array Implementation:** A separate queue for each priority number is maintained. Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT & REAR pointers. We can use a two-dimensional array for this purpose where each queue will be allocated the same amount of space.



- **Heap Implementation:**

Heaps provide better performance compared to array / LL. Insert can be implemented in O(log n) time. With Fibonacci heap, insert can be implemented in O(1) time (amortized).

- **Application:**
  1. CPU scheduling
  2. Graph algorithms (Dijkstra, Prim)

* Queue using stacks.

Method 1. Costly enqueue.

Makes sure that oldest entered elem. is always at the top of stack, so that dequeue just pops from stack1. To put elem at top of s1, s2 is used.

enqueue (x)

- while s1 is not empty, push everything from s1 to s2
- push x to s1. (latest elem at bottom of stack 1)
- push everything back to s1.

TC O(n)

SC O(n)

dequeue

- if s1 empty, error
- pop from s1 & return it.

TC O(1)

Method 2 Costly dequeue.

In enqueue, simply put new elem at top of s1. During dequeue, if s2 is empty, then all elems of s1 are moved to s2 & top of s2 is returned.

enqueue (x)

Push x to s1

TC O(1)

dequeue()

- If both stacks empty, error.
- If s2 empty, push all from s1 to s2. — O(n)
- Pop from s2 & return it.
- Push back to s1

```
void enqueue (int x) {
    s1.push(x);
    count++;
}
```

O(1)

METHOD 2

```
void dequeue() {
    if (s1top == -1 && s2top == -1)
        → queue empty
    else {
        for (i=0; i < count; i++) {
            temp = s1.pop();
            s2.push(temp);
        }
        x = s2.pop();
        printf("Dequeued : %d", x);
        count--;
        for (i=0; i < count; i++) {
            a = s2.pop();
            s1.push(a);
        }
    }
}
```

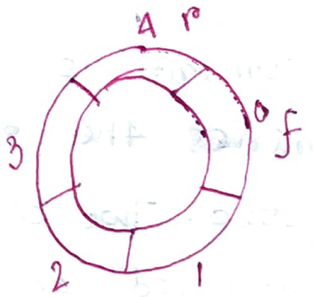
O(n)

```
int dequeue () {                    // queue using 2 stacks
  if (! isEmptyStack (s2))
        return pop (s2);
  else {
     while (! isEmpty Stack (s1))
          push (s2, pop(s1));
     return pop(s2);
  }
}
```

Code

1. Simple Circular Array.

```
typedef struct ArrayQueue {
    int front, rear;
    int capacity;
    int *array;
} Queue;
```



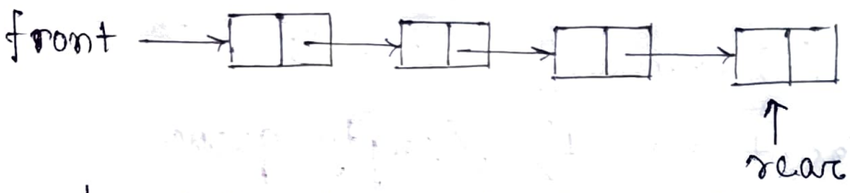

```
Queue *Create (int size) {
    Queue *Q = (Queue *)malloc (sizeof (Queue));
    if (!Q)
        return NULL;
    Q → capacity = size;
    Q → front = Q → rear = -1;
    Q → array = (int *) malloc (Q → capacity *
                                sizeof (int));
    if (!Q → array)
        return NULL;
    return Q;
}
int IsEmptyQueue ( Queue *Q) {      // f == -1
    return (Q → front == -1);
}
int IsFullQueue (Queue *Q) {        // f = (r+1) % N
    return ((Q → rear + 1) % Q → capacity ==
                                Q → front);
}
int QueueSize () {
    return (Q → capacity - Q → front + Q → rear + 1)
                        % Q → capacity;
}
void EnQueue ( Queue *Q, int data) {
    if (IsFullQueue (Q))
        printf ("Queue overflow");
    else {
```

```
// r = (r+1) % N
    Q->rear = (Q->rear + 1) % Q->capacity;
    Q->array[Q->rear] = data;
    if (Q->front == -1) // empty queue
        Q->front = Q->rear;
  }
}
int DeQueue (Queue *Q) {
  int data = 0;
  if (IsEmptyQueue (Q)) {
    printf ("Queue empty!");
    return 0;
  }
  else {
    data = Q->array[Q->front];
    if (Q->front == Q->rear)   // 1 element
      Q->front = Q->rear = -1;
    else
      Q->front = (Q->front + 1) % Q->capacity;
                // f = (f+1) % N
  }
  return data;
}
void DeleteQueue (Queue *Q) {
  if (Q) {
    if (Q->array)
      free (Q->array);
    free (Q);
  }
}
```

SC (for n EnQueue) - O(n)
TC for EnQueue, DeQueue, IsEmptyQueue, IsFullQueue, QueueSize, DeleteQueue - O(1).

- Max size of queue must be defined in prior & can't be changed.

2. LL Implementation.




```
typedef struct ListNode {
    int data;
    ListNode *next;
} LLNode;
typedef struct QueueLL {
    LLNode *front;
    LLNode *rear;
} Queue;
Queue *Create() {
    Queue *Q;
    LLNode *temp;
    Q = (Queue*)malloc(sizeof(LLNode));
    if (!Q)
        return NULL;
    temp = (LLNode*)malloc(sizeof(LLNode));
    Q->front = Q->rear = NULL;
    return Q;
}
int IsEmptyQueue(Queue *Q) {
    return (Q->front == NULL);
}
void Enqueue(Queue *Q, int data) {
    LLNode *newnode;
    newnode = (LLNode*)malloc(sizeof(LLNode));
    if (!newnode)
        return NULL;
    newnode->data = data;
    newnode->next = NULL;
    if (Q->rear)
        Q->rear->next = newnode;
```

```
    Q->rear = newnode;
    if (Q->front == NULL)
        Q->front = Q->rear;
}
int Dequeue (Queue *Q) {
    int data = 0;
    LLNode *temp;
    if (IsEmptyQueue(Q)) {
        printf("Empty queue!");
        return 0;
    }
    else {
        temp = Q->front;
        data = Q->front->data;
        Q->front = Q->front->next;
        free(temp);
    }
    return data;
}
void DeleteQueue (Queue *Q) {
    LLNode *temp;
    while (Q) {
        temp = Q;
        Q = Q->next;
        free(temp);
    }
    free(Q);
}
```

SC for n EnQueue - O(n)

TC of EnQueue, DeQueue,
IsEmptyQueue, DeleteQueue - O(1).

DS1 (Th, Code) DS2 (Adv.)

Queue.

* Ordered list of FIFO form.
* Enqueue, dequeue operation.
* ADT — Enqueue, Dequeue, Front (returns front), QueueSize, IsEmpty.

* **Applications** —

Direct

Operating systems schedule jobs in the order of arrival, Simulation of real world queues, multiprogramming, asynchronous data transfer (file IO, pipes, sockets), auxiliary data structure algorithms.

* **Implementation.**
  - → Simple Circular Array
  - → Dynamic Circular Array
  - → Linked List.

G'12 Suppose a circular queue of capacity n-1 elems is implemented with an array of n elems. Assume that the insertion and deletion operations are carried out using rear & front. Initially, rear = front = 0. Cond^{n}s to detect overflow & underflow :

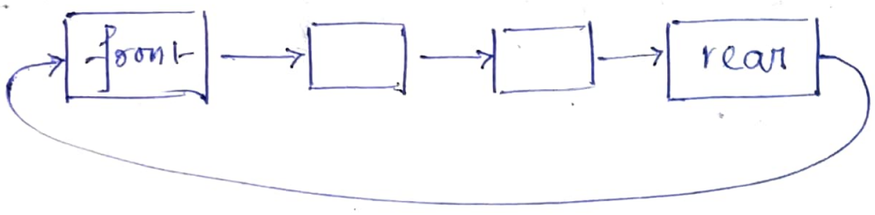
a

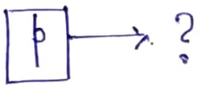
b

queue full a. (rear + 1) % n == front

queue empty b. rear == front

G'04. A circular LL is used to represent a queue. Variable p is used to access the queue. To which node should p point to such that both enqueue & dequeue can be performed in O(1)





Ans. rear node.

G'16 Q Q denotes a queue containing 16 numbers & S be an empty stack. Head(Q) returns the elem at head of queue; top(S) returns top of S.

**Max. possible no. of iterations of while loop in the algo:

Ans. 256

```
While (Q is not empty) do
    if S is empty or top(S) ≤ head(Q)
        x := dequeue (Q)
        push (S, x)
    else
        x := pop (S)
        enqueue (Q, x)
```

→ Go

Q. MultiDequeue (Q) {

```
m = k
while ( (Q is not empty) & (m > 0) ) {
    Dequeue (Q);
    m = m-1;
}
}
```

k being global parameter

Worst case TC of a sequence of n queue op^n's on an initially empty queue?

→ O(n)

Q. Implementation of queue:

i) isEmpty(Q)

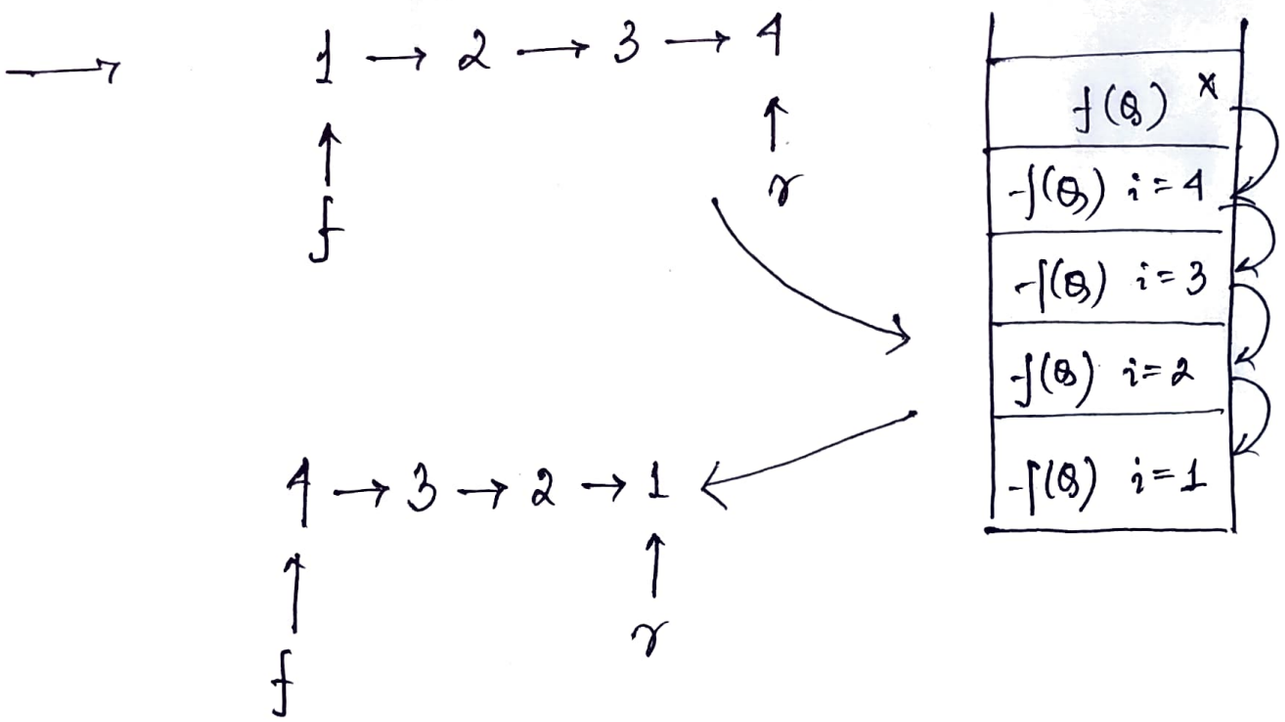
ii) delete(Q) - deletes front & returns value

iii) insert(Q, i) - insert i @ rear

Consider function -

```
void f(queue Q){
    int i;
    if (!isEmpty(Q)){
1.      i = delete(Q);
2.      f(Q);
3.      insert(Q, i);
    }
}
```

What f does?



Reverses order of elems in the queue.

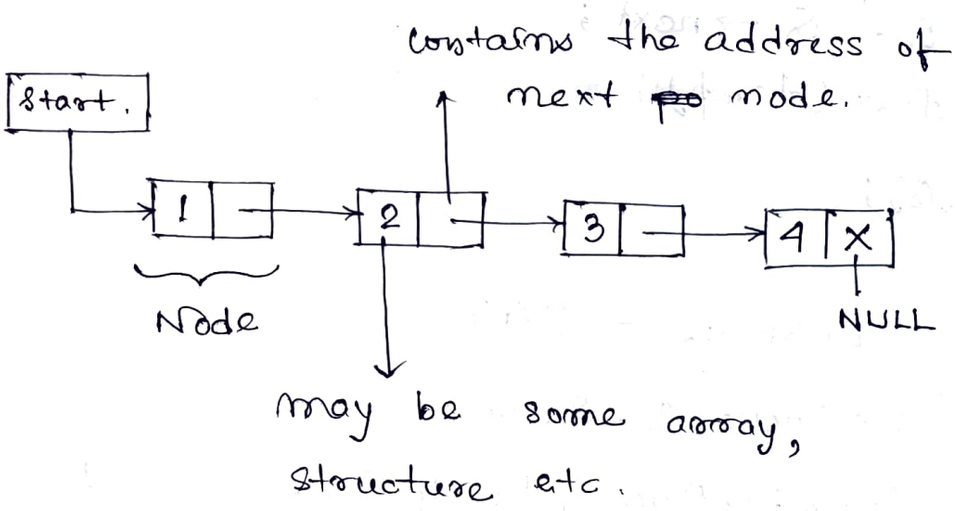
DS2. (Th) DS1 (code, Adv.) git

Linked Lists.

* A linked list is a collection of nodes (data elements) in which the linear representation is given by links from one node to the next node.

* A linked list does not allow random access of data. Elements in a linked list can be accessed in a sequential manner. Insertion & deletion of elements can be done at any point in the list in a constant time.

* A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.



* Since in a ll, every node contains a pointer to another node which is of the same type, it is also called a self-referencial data type.

* LL contains a pointer variable START that stores the address of the first node.

We can traverse the entire list using START that contains the address of first node; the next part of the first node in turn stores address of its succeeding node.

If START = NULL, the ll is empty, containing no nodes.

* Implementation of LL. in C.

Self-referencial structure.

```
struct node {
    int data;
    struct node *next;
};
```

* MAR.

Start
1000

	Address	Data	NEXT
1 →	1000	H	1006
2	1002		
3	1004		
4	1006	E	1012
5	1008		
6	1010		
7	1012	L	1014
8	1014	L	1018
9	1016		
10	1018	O	-1

* LL vs Arrays:

(i) Unlike an array, a linked list does not store its data elements (Nodes) in consecutive memory locations.

(ii) Unlike an array, a LL does not allow random access of data. Nodes in a LL can be accessed only in a sequential manner.

(iii) Unlike an array, we can add any number of elements in LL.

* Memory allocation & de-allocation for LL:

Insertion: Computer maintains a LL of all free memory cells. (the free pool). The free pool has a pointer AVAIL that stores the address of the next first free space. If we need to insert an element to a LL, the address of AVAIL will be taken & the element will be stored there. The next available free space's address will be stored in AVAIL.

Deletion:

by deleting 2 from LL,

Start → 1

	Data	Next
1	1	3
2		
3	2	5
4		
5	3	6
6	4	-1

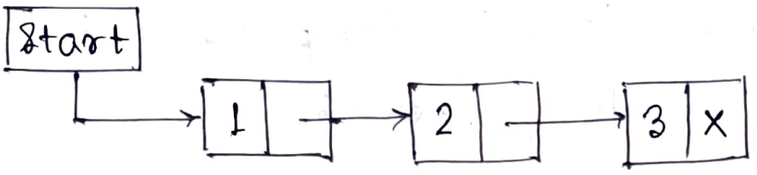
← 4

	Data	Next
1	1	5
2		
3		
4		
5	3	6
6	4	-1

The next ptr of the previous node gets changed to the deleted node's ptr.

By garbage collection, the OS collects the free memory to make it reusable.

* Singly LL: A singly LL is the simplest type of LL in which every node contains some data and a pointer to the next node of the same data type. A singly LL allows traversal of data only in one way.



- Traversing:

1. Set Ptr = Start.
2. Repeat 3 and 4 while ptr != NULL
3. Apply process to ptr -> data
4. Set ptr = ptr -> Next.
5. Exit.

- Searching for a value:

1. Set ptr = Start
2. Repeat step 3 while Ptr != NULL
3. If Val = Ptr → data
   Set Pos = Ptr
   Go to step 5
   Else
   Set Ptr = Ptr → next
4. Set Pos = NULL
5. Exit.

Code

Singly Linked List.

```
typedef struct ListNode {
    int data;
    struct ListNode *next;
} LLNode;
void Traverse (LLNode *head) {
    LLNode *temp = head;
    while (temp != NULL)
    {
        printf ("%d\t", temp -> data);
        temp = temp -> next;
    }
}
void Insert ( LLNode **head, int data, int position)
{
    int k = 1;
    LLNode *p, *q, *newnode;
    if (!newnode) {
        printf ("Memory error!");
        return;
    }
    newnode -> data = data;
    p = *head;
    if (position == 1) {
        newnode -> next = p;
        *head = newnode;
    }
    else {
        while (p != NULL && k < position) {
            k++;
            q = p;
            p = p -> next;
    }
```

newnode = (LLNode*) malloc (sizeof(LLNode));

```
        q->next = newnode;
        newnode->next = p;
    }
}
void Delete (LLNode **head, int position) {
    int k = 1;
    LLNode *p, *q;
    if (*head == NULL) {
        printf ("Empty list");
        return;
    }
    p = *head;
    if (position == 1) {
        *head = (*head)->next;
        free(p);
        return;
    }
    else {
        while ((p != NULL) && (k < position)) {
            k++;
            q = p;
            p = p->next;
        }
        if (p == NULL)
            printf ("Wrong position");
        else {
            q->next = p->next;
            free(p);
        }
    }
}
```

```
void DeleteLL (LLNode **head) {
    LLNode * auxiliary , * iterator;
    iterator = *head;
    while (iterator) {
        auxiliary = iterator → next;
        free(iterator);
        iterator = auxiliary;
    }
    *head = NULL;
}.
```

Code

Doubly Linked List.

```
typedef struct Doubly {
    int data;
    struct Doubly *next;
    struct Doubly *prev;
} DLLNode;
void DLLInsert ( DDLNode **head, int data, int position)
{
    int k=1;
    DLLNode *~~temp~~ current, *newNode, *previous;
    newNode = (DLLNode *) malloc (sizeof (DLLNode));
    newNode → data = data;
    current = * head;
    if (position == 1)
    {
        newNode → next = *head;
        newNode → prev = NULL;
        if (*head)
            (*head) → prev = newNode;
        *head = newNode;
    }
```

```
else
{
    while (current && k < position)
    {
        k++;
        previous = current;
        current = current->next;
    }
    if (!current)          // if (k != position)
    {
        printf ("Position does not exist");
    }
    newNode->next = current;
    newNode->prev = previous;
    if (current)
        current->prev = newNode;
    previous->next = newNode;
}
}
```

- Rest of the codes — Github.

G'08

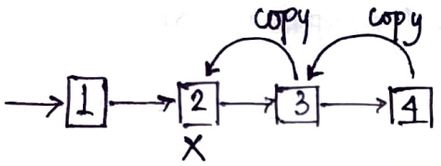
```
struct node {
    int val;
    struct node * next;
};
void rearrange (struct node *list) {
    struct node *p, *q;
    int temp;
    if (!list || !list->next) return;
    p = list; q = list->next;
    while (q) {
        temp = p->val; p->val = q->val;
        q->val = temp; p = q->next;
        q = p ? p->next : 0;
    }
}
```

content after the code runs on 1, 2, 3, 4, 5, 6, 7.

→ 2, 1, 4, 3, 6, 5, 7

Q'02. In the worst case, # comparisons needed to search a SLL of length n → n

Q'04 Let f be a SLL, Q be a ptr to an intermediate node x in the list. What is the worst case time complexity of best known algo to delete the node x ?



→ O(1).

Copy next elem's data to x & delete Q

```
X.data = (X → link).data
X.link = (X → link).data = (x → link → link).data
(X → link) → link = NULL
```

```
node* t = Q;
Q→data = Q→next→data;
Q→next = Q→next→next;
free(t);        O(1)
```

When f returns 1

Q'03

```
struct item {
    int data;
    struct item *next;
};
int f (struct item* p){
    return ((p == NULL) || (p→next == NULL) ||
        ((p→data <= p→next→data) &&
            f(p→next)));
}
```

→ a || b || (c && d)

1. If list is empty p == NULL, returns true.
2. If list has one element p→next == NULL returns true.
3. Goes to the (c && d) part only when have more than 1 elems.

4. c : p→data <= p→next→data

If 1st elem data is <= next elem data.

Goto d : f (p→next) only when c returns true.

So, f is checking if the list is in non-decreasing order.

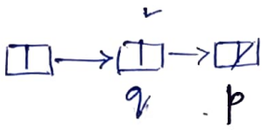
OR, Copy next elem's data to x & delete next elem.

→

```
node * temp = q → next;
q → data = temp → data;
q → next = temp → next;
free (temp);
```

---

* The f^n modifies the SLL by moving the last elem Q.6(10 to the front of the list & returns the list.




```
node * move-to-front (node *head) {
    node* p, *q;
    if ((head == NULL || (head→next == NULL))
        return head;
    q = NULL; p = head;
    while (p→next != NULL) {
        q = p;
        p = p→next;
    }
?? — STATEMENT — ??
    return head;
}
```

Ans.
```
q→next = NULL
p→next = head
head = p
```

G'16. N items are stored in a sorted DLL. For a delete operation, a ptr is provided to the record to be deleted. For a decrease-key op^n, a ptr is provided to the record on which the op^n is to be performed. An algo performs the following op^{n}s on the list in this order:

$\Theta(n)$ delete	delete $O(1)$
$O(\lg n)$ insert	insert $O(n)$
$O(\lg n)$ find	find $O(n)$
$\Theta(n)$ decrease-key	decrease-key $O(n)$

What is the TC of all these op^{n}s put together?

a) $O(\lg^2 n)$ c) $O(n^2)$

b) $O(n)$ d) $\Theta(n^2 \lg n)$

$\rightarrow$ $n + n\lg n + n\lg n + n^2$.

$O(n^2)$

Q. n^{th} node from end of SLL.

$\rightarrow$ Approach 1. Find length of LL. $O(n)$

Compute len $- n + 1^{th}$ elem. $O(n)$

$O(n)$.

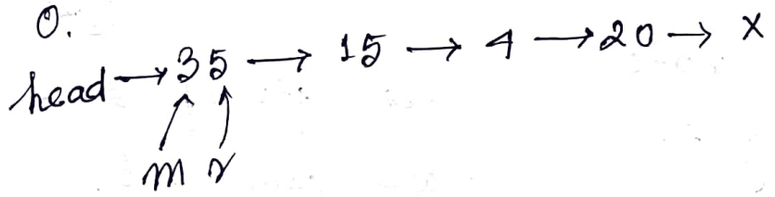
Approach 2.

Maintain 2 ptrs - reference & main ptr.

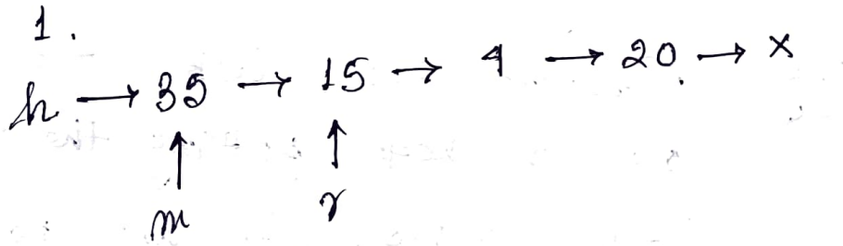
Initialize both to head. First, move

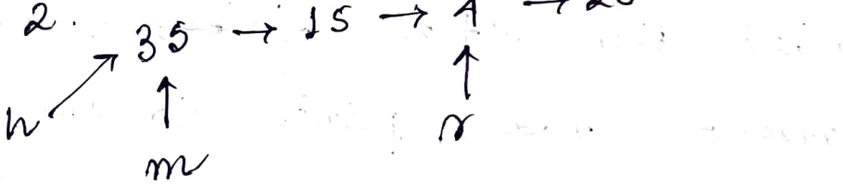
ref-ptr to n^{th} node from head. Now, move both ptrs one by one until the ref-ptr reaches the end. Now, the main ptr will point to n^{th} node from end. O(n) TC | O(1) SC.

eg. $n = 3$

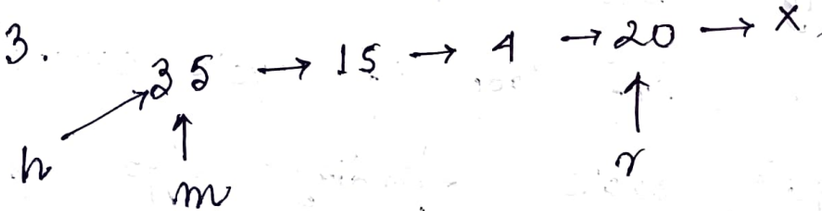
0.


main-ptr m
ref-ptr r

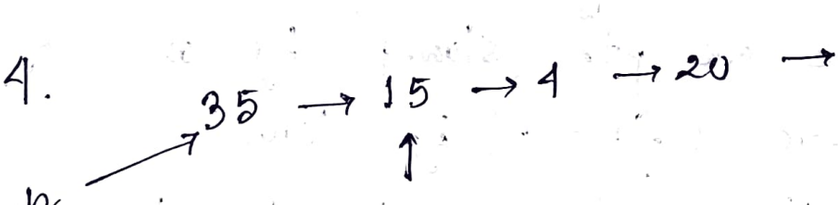
1.


2.


move r to 3rd node from head.

3.


Now, m & r are 3 positions away.

4.


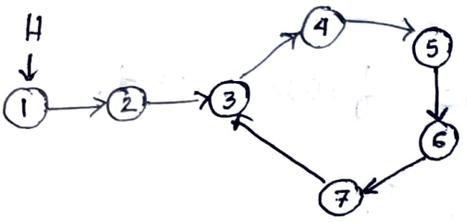
move m & r together until r → NULL

So, 3rd node from end is 15.

Q. Detect loop in a linked list.

1. Brute force.

Start with first node & see whether there's any node whose next ptr is current node's address. (May go to infinite loop)



next ptrs of 2 & 7 are same : 3.

2. Hashing Traverse the list one by one & keep putting the node's address in a hash table. At any point, if NULL is reached then return false, & if next of current node points to any of the previously stored nodes in hash then return true. (Inf. loop if cycle exists)

3. Floyd's cycle finding algo O(n) TC O(1) SC

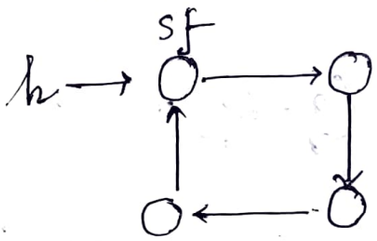
- Traverse using 2 ptrs.
- Move one ptr (slow_p) by one & another (fast_p) by two.
- If these ptrs meet at the same node then there's a loop. If don't then no loop.

(A tortoise & a hare running on a track. The faster running hare will catch up with the tortoise if they are running in a loop.)

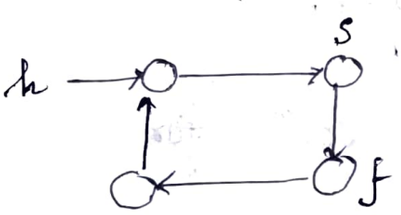
Slow- s fast - f

eg

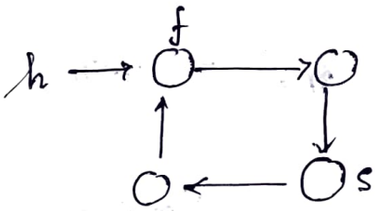
0.



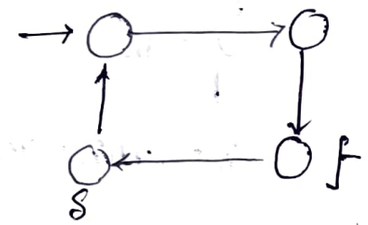
1.



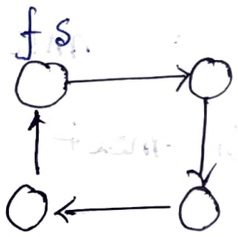
2.



3.

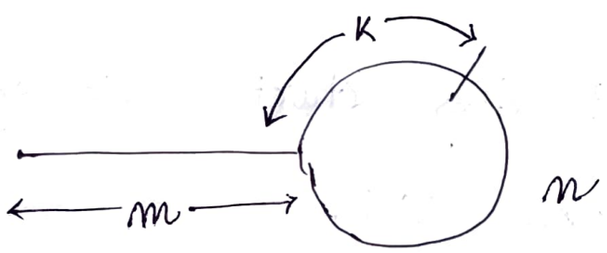


4.



have a cycle

? How it works



Meeting pt is k steps away from the beginning of cycle.

Say, the 2 ptrs meet when slow_ptr has taken d steps (fast-ptr would have taken 2d steps).

cycle length n

So, d = m + p times cycle through the loop of length n + k
(for slow)

$= m + pn + k$

$2d = m +$ q times cycle through the loop of length n $+ k$
(for fast)

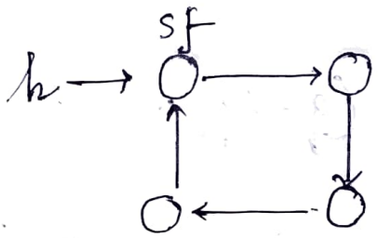
$= m + qn + k$

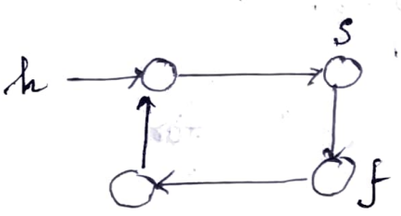
$p, q \rightarrow$ +ve integers

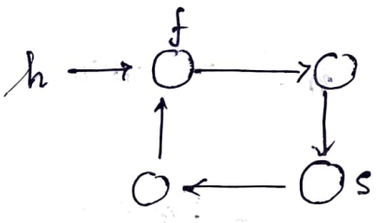
$\Rightarrow 2(m + pn + k) = m + qn + k \Rightarrow m + k = (q - 2p)n$

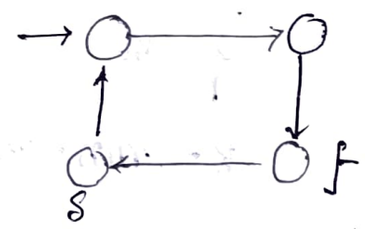
Slow - s fast - f

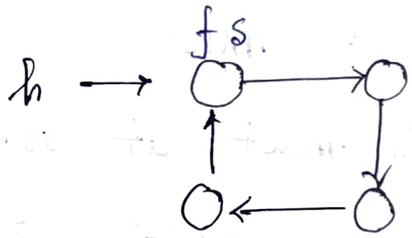
eg -

0. 

1. 

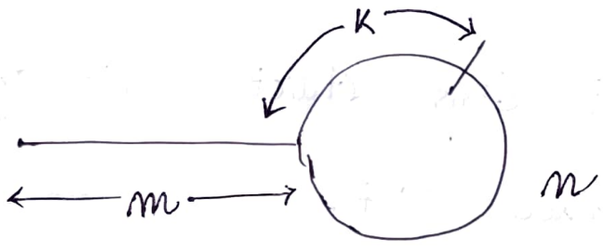
2. 

3. 

4. 

have a cycle

? How it works



Meeting pt. is k steps away from the beginning of cycle.

Say, the 2 ptrs meet when slow_ptr has taken d steps (fast-ptr would have taken 2d steps).

| cycle length n

So, d = m + p times cycle through the loop of length n + k

(for slow)

= m + pn + k

2d = m + q times cycle through the loop of length n + k

(for fast)

= m + qn + k

| p, q → +ve integers

$\Rightarrow 2(m + pn + k) = m + qn + k \Rightarrow m + k = (q - 2p)n$

$(q-2p)n$ is multiple of n.

One soln for $(m+k) = (q-2p)n$ is as follows

$p = 0$
$q = m$
$k = mn - m$

$\Rightarrow m + (mn - m) = (m - 2 \cdot 0)n$

$\Rightarrow mn = mn$

$d = m + pn + k$

$= mn$

So, hypothesis (that both meet at some point) is correct as there exists a soln.

Q. Finding start node of the loop in a linked list.

→ After finding the loop by Floyd's algo, we initialize the slow-ptr to point to the head. From that point onwards both slow & fast ptr move by only one node at a time. Now, the point at which they meet is the start of the loop.

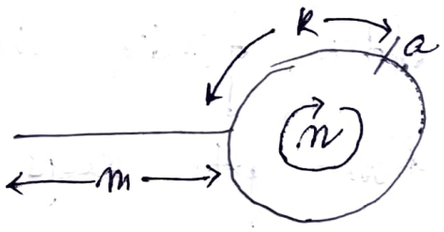
TC O(n)
SC O(1)

? How it works (contd. from above discussion)

Assume, we know m.

If we let both ptrs move $m+k$ steps, the slow ptr would have to arrive at the

point they met originally (k steps away from the cycle beginning).



Previously, shown

$$m + k = (q - 2p)n$$

Since, m+k steps is a multiple of cycle length, the fast ptr, in the mean time, would go through the cycle (q-2p) times & would come back to the same point (starts from point a, ends at a - k steps away from cycle beginning).

Now, instead of letting them move m+k steps, if we let them move only m steps, the slow ptr would arrive at the cycle beginning; after the fast ptr would be k steps short of completing (q-2p) rotations. Since, it started k steps in front of the cycle beginning, the fast ptr would have to arrive at the cycle beginning.

As a result, this explains that they would have to meet at the cycle beginning after some number of steps for the very first time (because slow ptr just arrived at the cycle after m steps & it could never see fast ptr which was already in cycle).

↝ Find # steps to move to go to the beginning of cycle : m (until they meet)

↝ Length of cycle : list length − m

Q: Length of loop in a SLL.

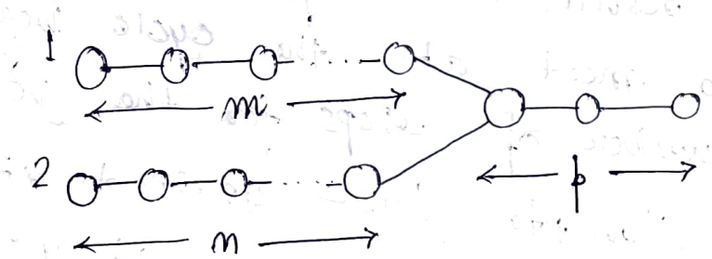
→ After finding the loop, keep slowptr as it is. fast ptr keeps on moving, until it again comes back to slowptr. While moving fastptr, use a counter that increments at a rate of 1. (at a pace-one node once)

```
While (slow != fast) {
    fast = fast → next;
    counter ++;
}
return counter;
```

Here, we are just counting # nodes in the loop using the fast ptr by moving one node at a time.

Q. Find the merging pt. of 2 linked lists.

→

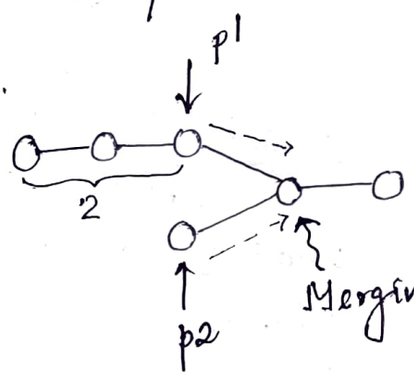


- Get count of list 1 (say m+p)
- Get count of list 2 (say n+p)
- Difference of them (m-n)

- Now traverse the bigger list from 1st node till abs(m-n) nodes, so that from here onwards both lists have equal no. of nodes.

- Then we can traverse both the lists in parallel till we come across a common node (by comparing addr. of nodes).

eg.



len 1 = 5
len 2 = 3
diff = 2

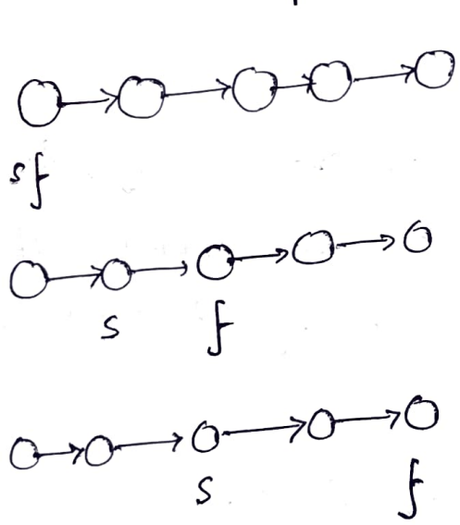
TC	O(m+n)
SC	O(1)
m →	len (list 1)
n →	len (list 2)

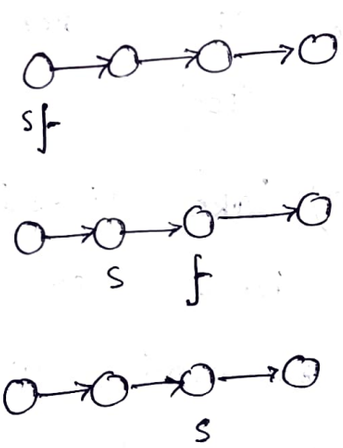
8. Middle of LL.

Use 2 ptrs - slow & fast. Move fast ptr twice the speed of slow, while slow moves one node at a time. By the time, fast reaches end of list, slow will be pointing to the middle node (if list has even # nodes, middle node will be ~~n/2~~ n/2 + 1.

TC O(n) SC O(1).

eg.





f → next == NULL

f == NULL.

```
while(f != NULL && f->next != NULL) {
    f = f->next->next;
    s = s->next
}
return s;
```

Adv. DS1 (code) → github DS2 (Th)

Linked Lists.

* In linked list,

→ Successive elements are connected by pointers.

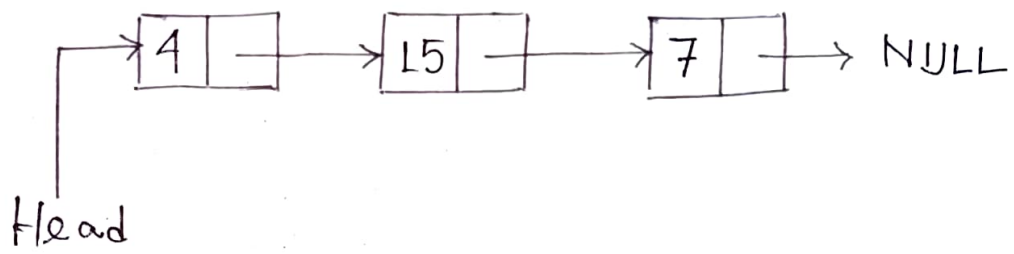
→ Last element points to NULL.

→ Can grow or shrink in size during execution of program.

→ Can be made just as long as required.

→ Doesn't waste memory space.

(But, takes some extra memory for pointers).



* Linked list operations —

Insert
Delete
} main.

Delete list
Count
Finding n^{th} node
} auxiliary

---

* In an array, memory is assigned during compile time while in a Linked list it is allocated during execution or runtime.
* Memory utilisation is inefficient in array, but memory utilisation is efficient in LL.

* Array vs Linked List.

Array	LL.
1. Collection of elements of similar data type.	1. Ordered collection of elements of same type, connected to each other using pointers.
2. Array supports random-access. Hence, accessing element is fast with TC of O(1).	2. LL supports sequential access. Hence, accessing element is of O(n) TC for woost case.
3. Elements are stored in contiguous memory location.	3. New elements can be stored anywhere in memory.
4. Insertion & deletion takes more time.	4. These operations are fast.
5. Memory is allocated as soon as the array is declared (Static Memory allocation).	5. Memory is allocated at runtime. (Dynamic memory allocation)
6. Each element is independent & can be accessed with index.	6. Each node is linked with other.
7. Array can be 1d, 2d, 3d or multi-d.	7. LL can be singly, doubly, circular.
8. Size must be specified at declaration.	8. Size of LL is variable.
9. Array gets memory allocated on stack.	9. LL gets memory allocated in heap.

* Comparison of LL with array & dynamic array :

Parameter	LL	Array	D-array.
Indexing	O(n)	O(1)	O(1)
Insertion/ Deletion at beginning	O(1)	O(n) array not full	O(n).
Insertion @ ending	O(n)	O(1) array not full	O(1), array no full O(n), array fu
Deletion @ ending	O(n)	O(1)	O(n).
Insertion @ middle	O(n)	O(n), array not full	O(n).
Deletion @ middle.	O(n)	O(n) array not full	O(n).
Wasted Space	O(n)	0 zero.	O(n)

* Singly Linked List.

→ Node Declaration.

```
struct ListNode {
    int data;
    struct ListNode *next;
};
```

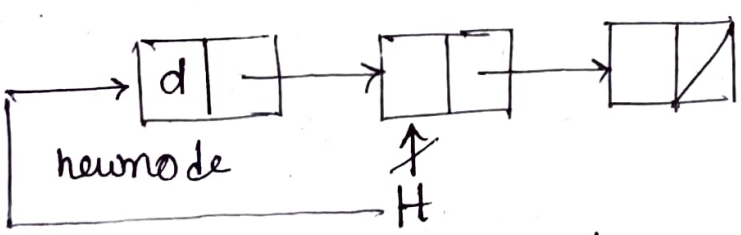
→ Traversing LL. ($T(n) = O(n)$; $S(n) = O(1)$)

1. Follow the next pointers.
2. Display the data part while traversing.
3. Stop when next pointer is NULL.

→ Inserting Node

a) At the beginning.

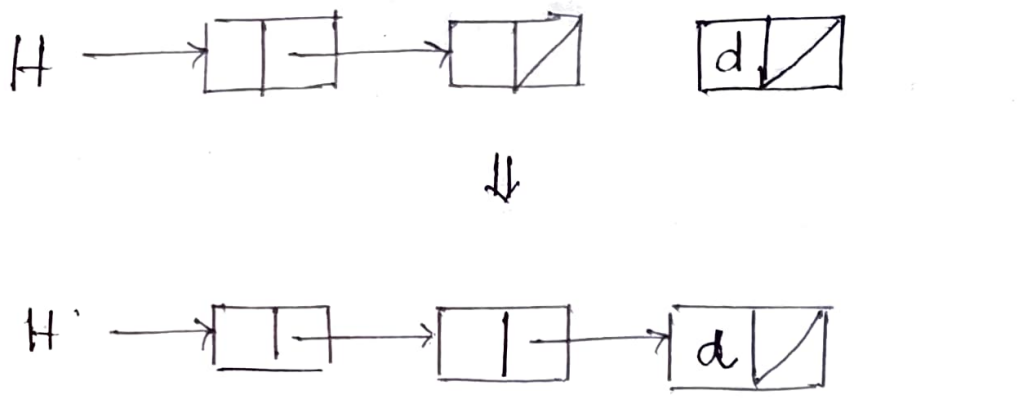
▶ Update next pointer of new node to point to the current head. Then update head pointer to point to new-node.



b) At the ending

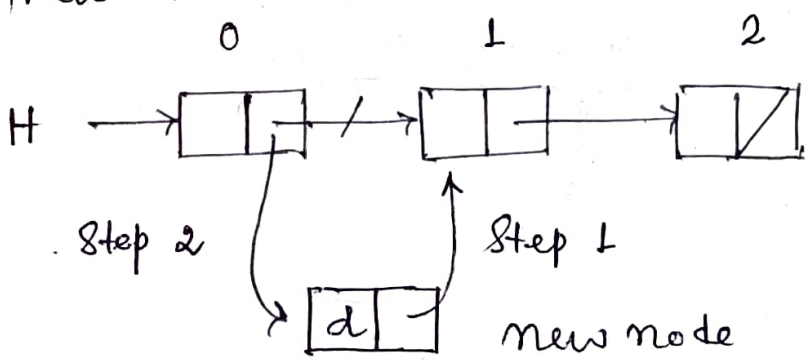
▶ Update new node's next pointer to NULL.

- Last node's next pointer points to the new node.



c) At the middle

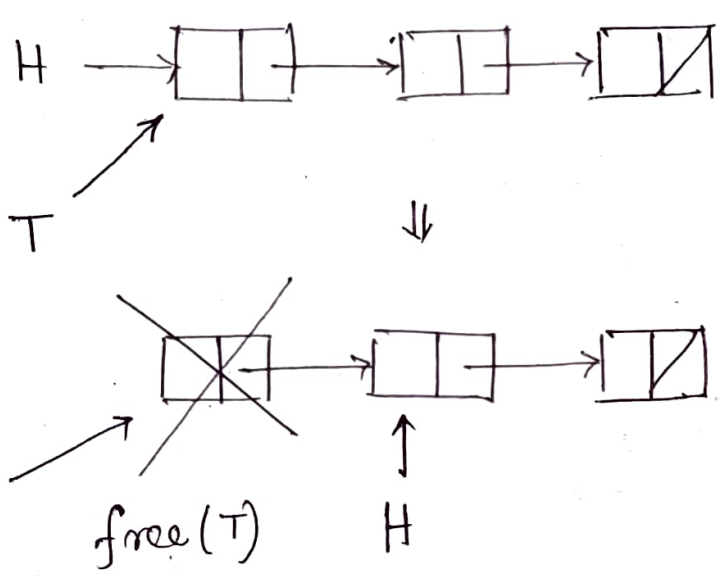
- Find the middle position.
- Traverse till the position before middle.
- Update newnode's next pointer to point to the node where ~~position~~, the previous node's next is pointing.
- Update previous node's next pointer to point to the new-node.



→ Deleting Node.

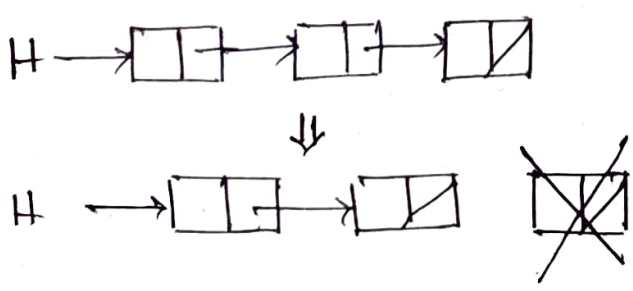
a) The first node

- Create temp node that will point to same node as of head.
- Move the head node's pointer to the next node & dispose the temp node.



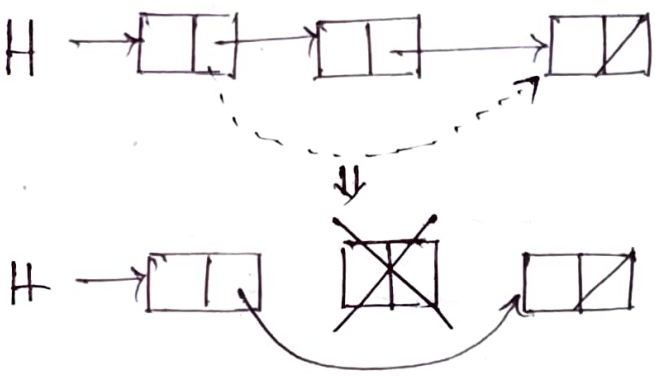
b) The last node.

- Traverse list & while traversing maintain the previous node address too.
- Update previous node's ~~address~~ next pointer to NULL.
- Dispose the last node.



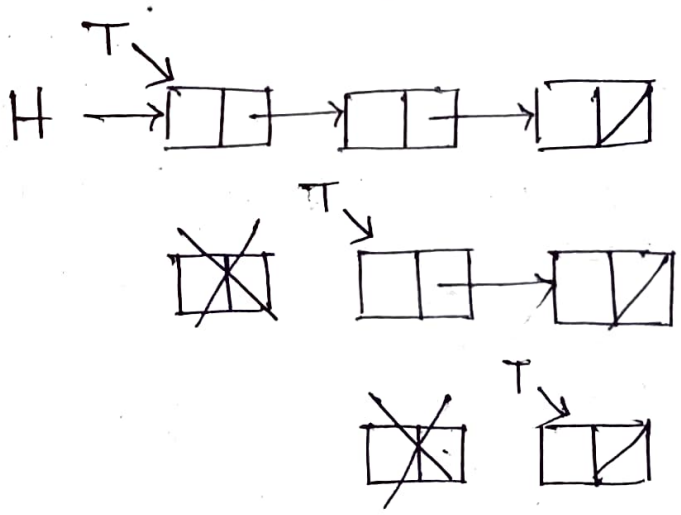
c) An Intermediate Node.

- Maintain previous node while traversing.
- Once found the node to be deleted, change previous node's next pointer to next pointer of the node to be deleted.
- Dispose the current node to be deleted.



→ Deleting Singly LL.

- Store current node in some temp & free the current node.
- Traverse to the next node with help of temp & repeat.



T → NULL

H = NULL.

* Doubly Linked List

→ The advantage of a doubly linked list is that given a node in the list we can navigate in both directions.

→ Disadvantages are each node requires an extra pointer, requiring more space & the insertion/deletion of node takes a little bit longer, due to more pointer operations.

→ Node Declaration.

```
struct DLLNode {
    int data;
    struct DLLNode *next;
    struct DLLNode *prev;
};
```

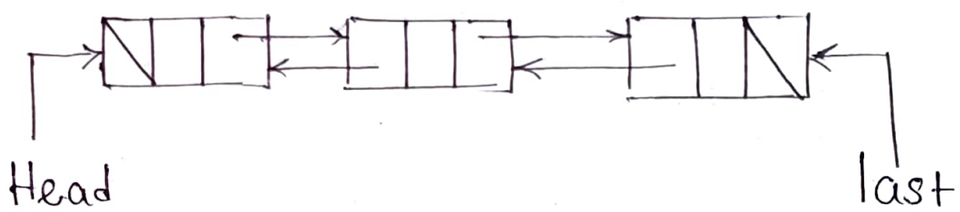
→ Traversal.

a) from beginning.

- Follow the next pointer using some temp variable.
- Print the data part while traversing.
- Stop when temp encounters NULL.

b) from end.

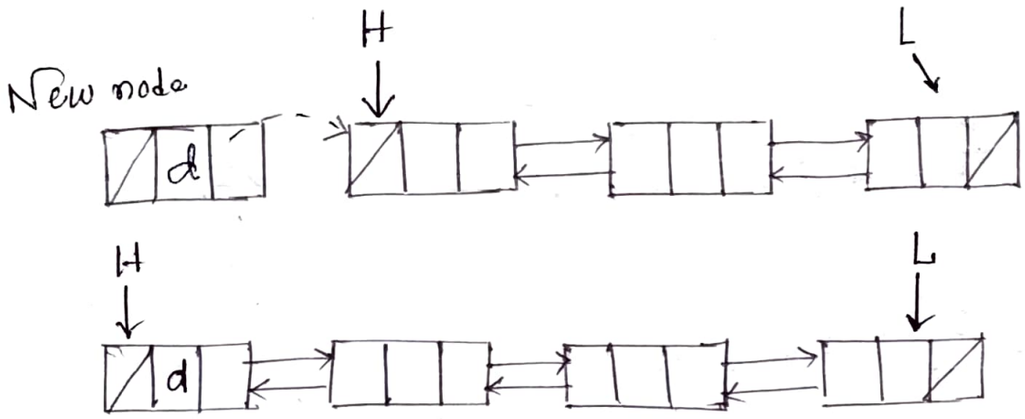
- Assign a temp variable to the pointer to the last node.
- Follow the previous pointer.
- Print the data part while traversing.
- Stop until temp becomes NULL.



→ Insertion.

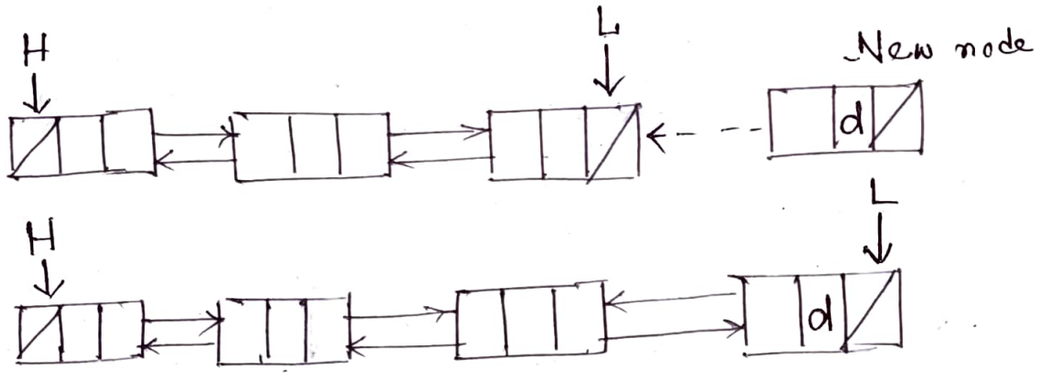
a) At the beginning.

- Update the right pointer of new node to point to the current head node & also make left pointer of new node as NULL.
- Update head node's left pointer to point to the new node & make new node as head.



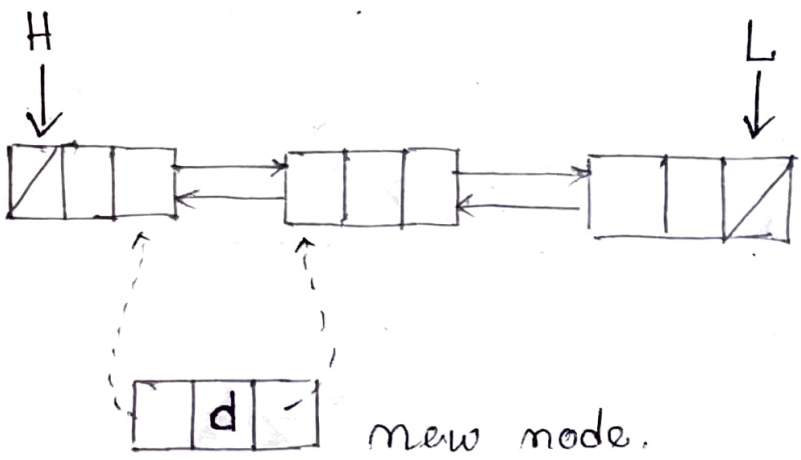
b) At the ending.

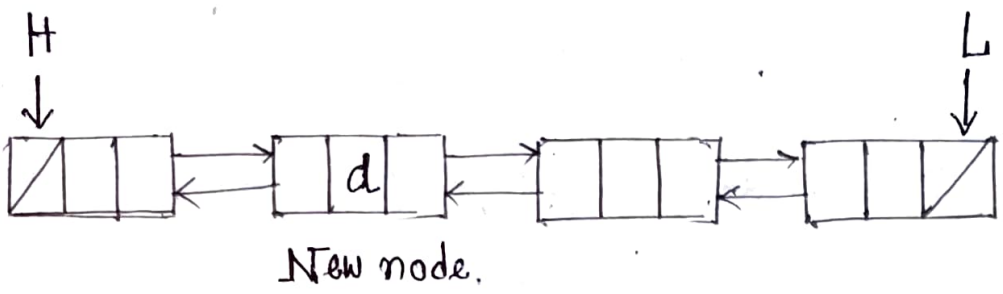
- New node's right pointer points to NULL & left pointer to the last node.
- Update right of pointer of last node to point to new node.



c) At the middle.

- New node's right pointer points to the next node of the position node where we want to insert the new node. New node's left pointer points to the position node.
- Position node right pointer points to the new node & the next node of position & node's left pointer points to new node.

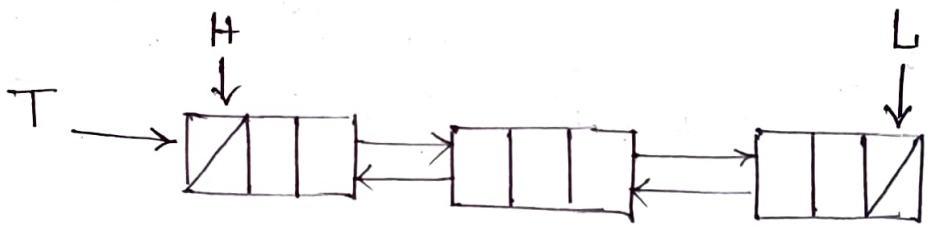




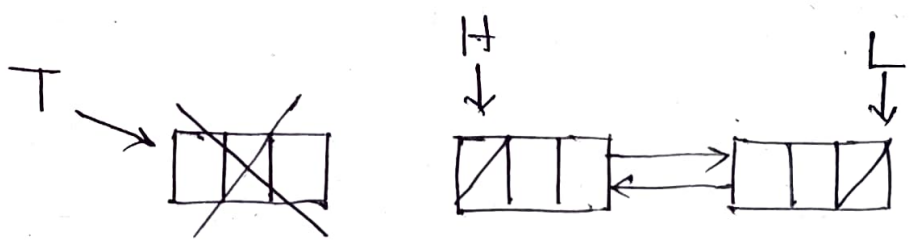
→ Deletion.

a) Deleting first node.

- Create a temporary node that will point to same node as that of head.



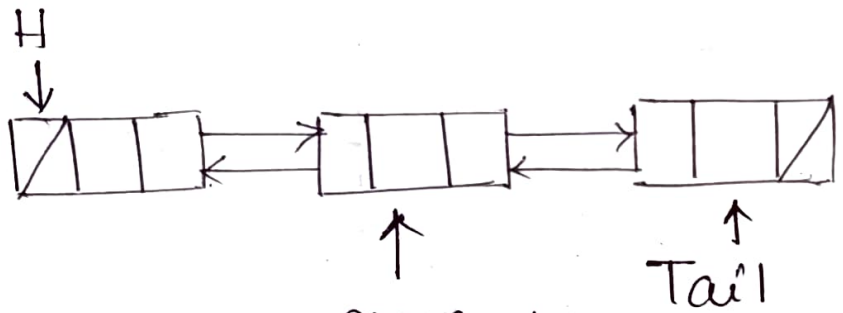
- Move the head node's pointer to the next node & change the head's left pointer to NULL. Then, dispose the temporary node.



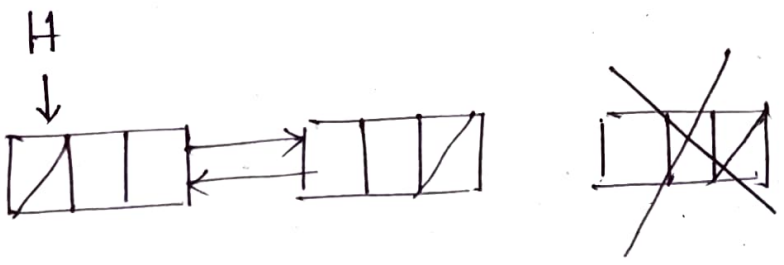
(If only last node's pointer - last is available, we need to traverse till NULL is encountered).

b) Deleting last node.

▶ Traverse the list & while traversing maintain the previous node's address too. By the time we reach the end of list, we will have two pointers one pointing to the tail & other pointing to the node before tail node.



▶ Update tail node's previous node's next pointer to NULL.



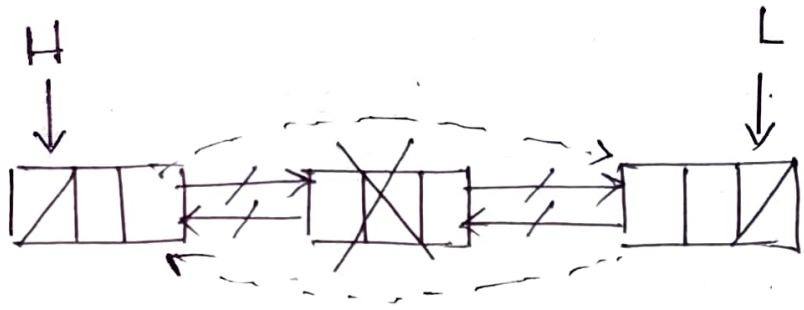
▶ Dispose tail node.

(If last node's pointer - last is available we delete it by moving last's pointer to it's previous.).

c) An Intermediate Node.

▶ Maintain previous node while traversing the list. Change the previous node's next pointer to the next node of the node to be deleted.

▶ Dispose the current node to be deleted.



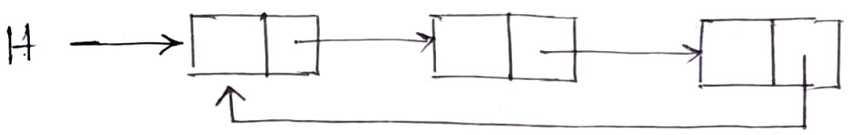
* Circular Linked List.

→ In circular linked list every node has a successor & no node has a NULL pointer to indicate ending of list.

→ Node Declaration.

```
struct CLLNode {
    int data;
    struct CLLNode *next;
};
```

→ Counting Nodes



To count the nodes, the list has to be traversed from node marked head, with the help of a dummy node current & stop the counting when current reaches the starting node head.

→ Traversal

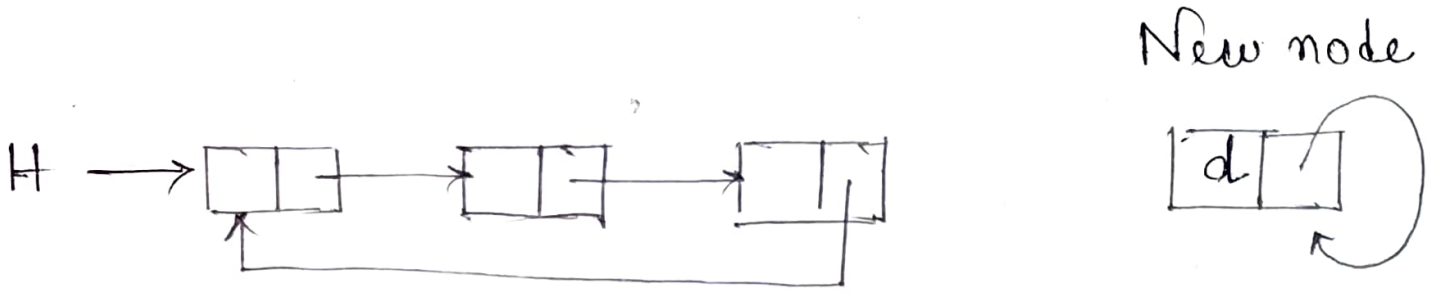
- Follow the next pointer starting from the head node.
- Print the data part till we reach the head again.

→ Insertion.

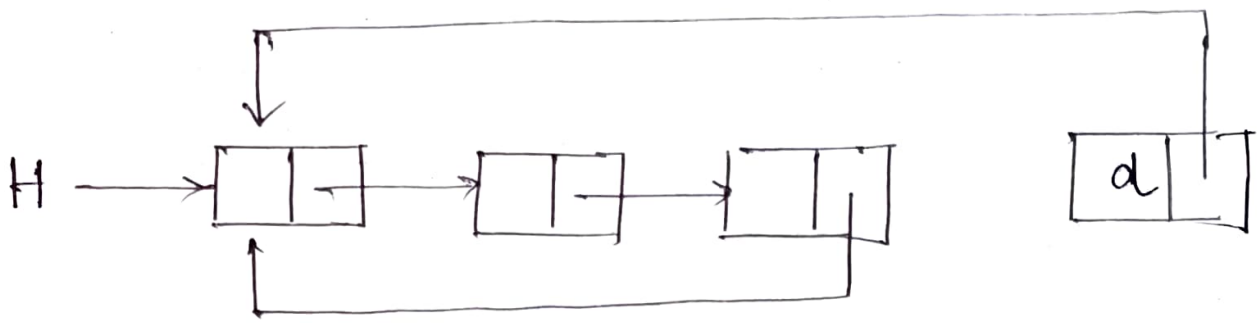
a) At the end.

(Insertion in between tail & head node).

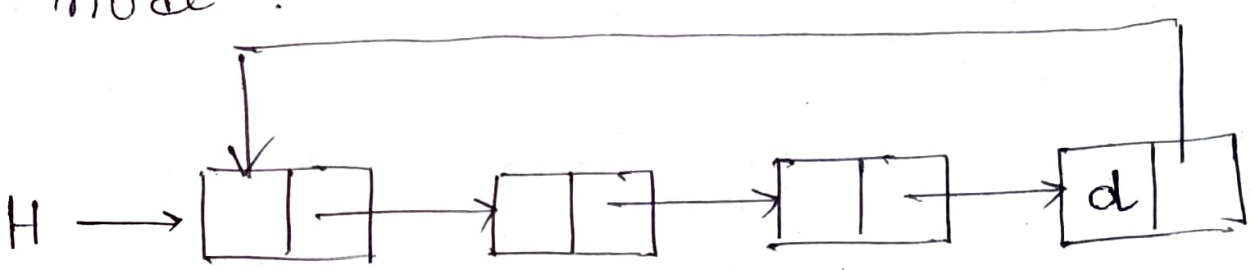
- Create a new node & initially keep its next pointer pointing to itself.



▶ Update the next pointer of new node with head node. & also traverse the list until the tail.

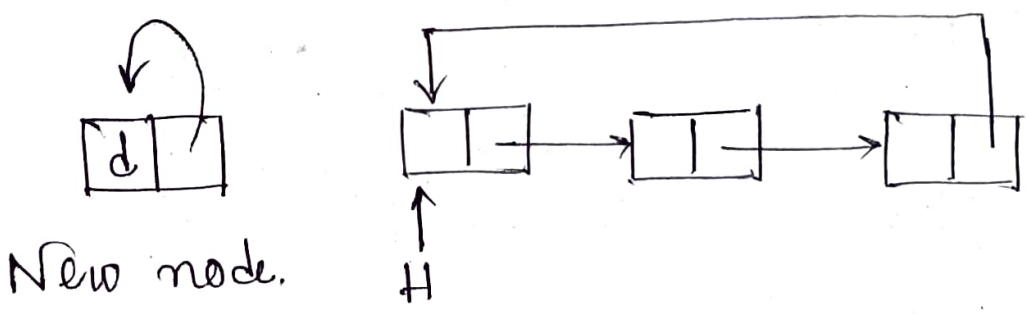


▶ Update the next pointer of tail node to point to new node.

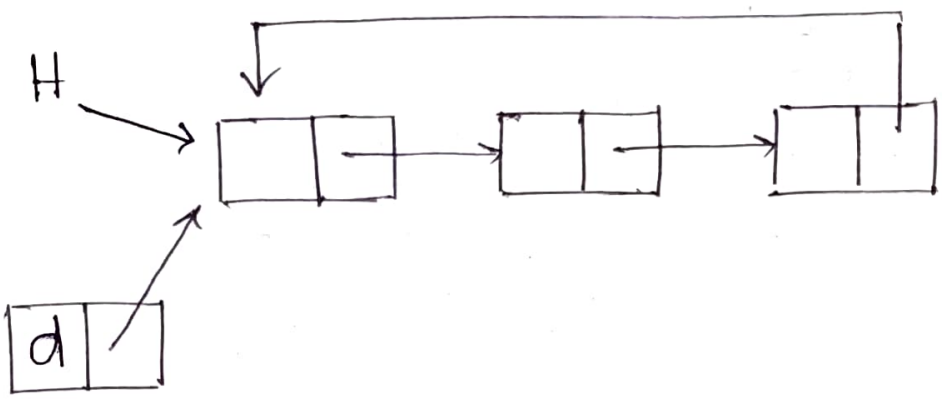


b) At the front.

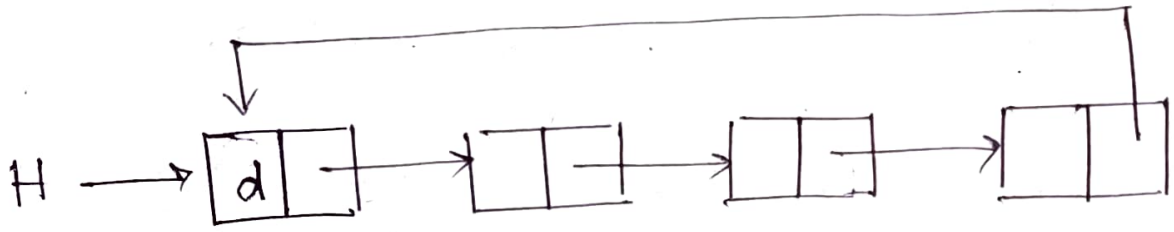
▶ Create new node & initially keep its next pointer pointing itself



- Update the next pointer of new node with head node & also traverse the list until the tail.



- Update the previous node of head on the list to point to new node.

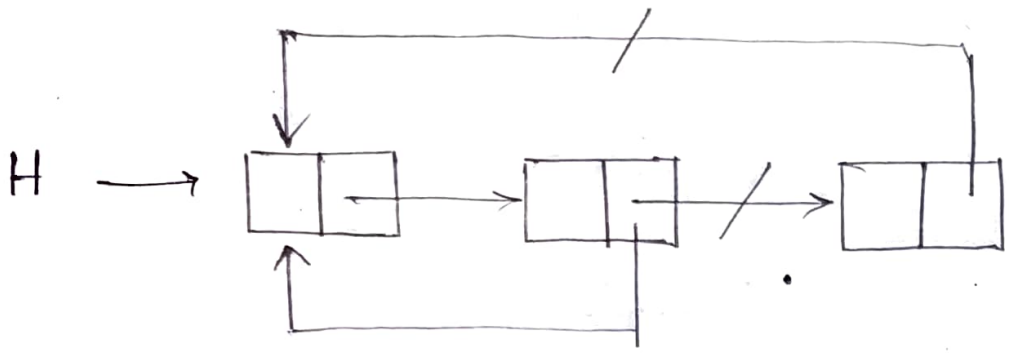


- Make newnode as head.

→ Deletion.

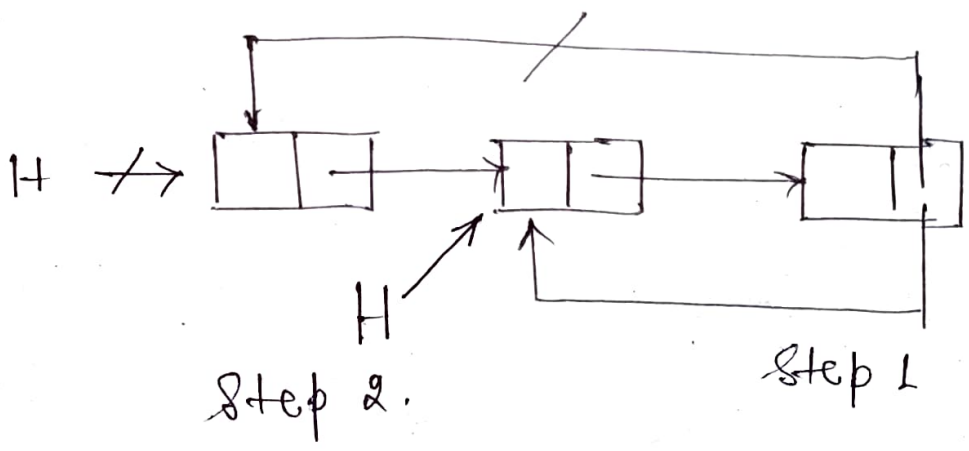
a) Last Node.

- Traverse the list & find the tail node & its previous node.
- Update the previous node's next pointer to point to head
- Dispose the tail node.



b) First Node.

- Find the tail node of the lost by traversing.
- Create a temp node that will point to head. Also, update the tail node's next pointer to point to next node of head.
- Move the head pointer to next node. ~~Create~~



- Dispose the temp.

→ Application.

Used in managing the computing resources of a computer.

* Memory-Efficient DLL (XOR LL)

Prokash Sinha

Nov 30, 2004.

Based on pointer difference.

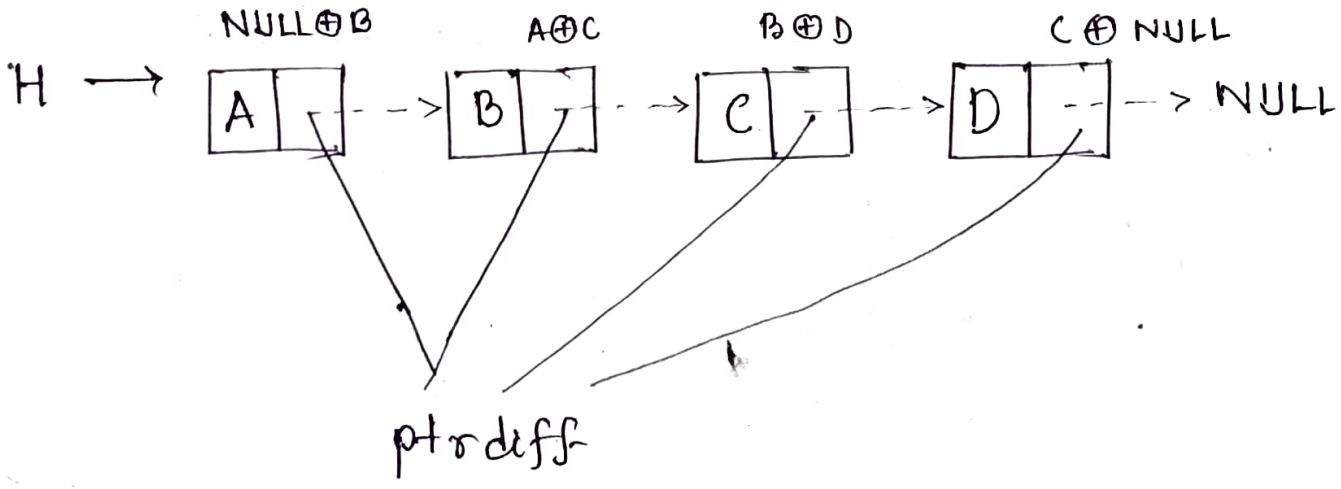
Each node uses only one pointer field to traverse the list back & forth.

→ Node declaration.

```
struct DLLNode {
        int data;
        struct DLLNode *ptrdiff;
};
```

The ptrdiff pointer field contains the difference between the pointer to the next node & the pointer to the previous node. It's calculated using XOR operation.

ptrdiff = pointer to previous node ⊕ pointer to next node.



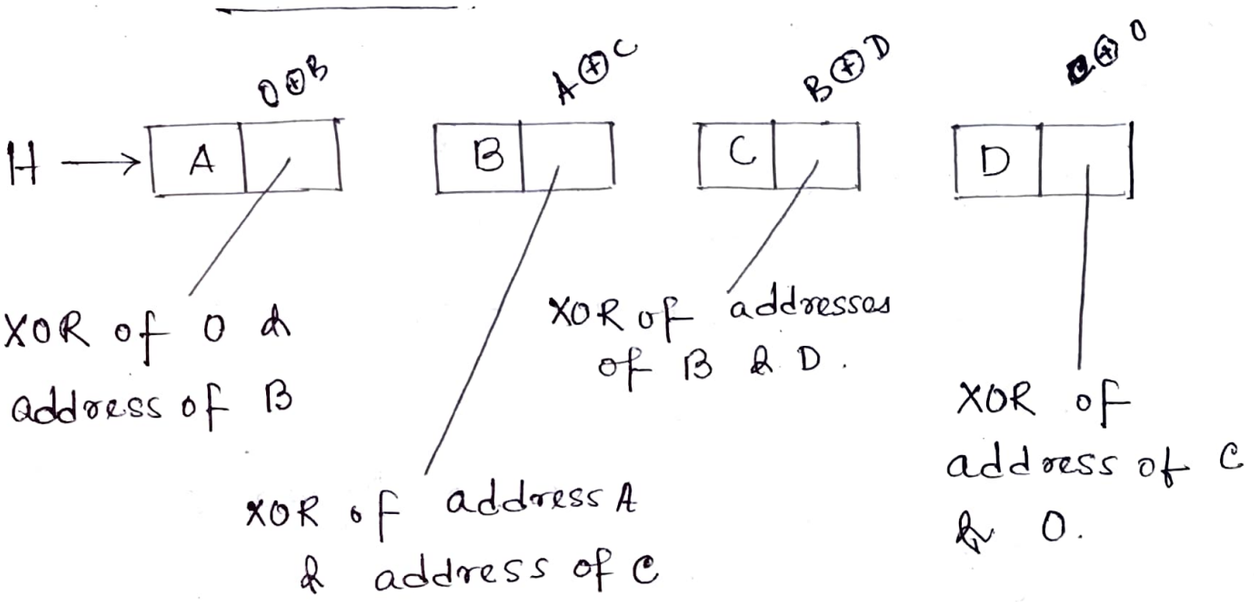
→ $X \oplus X = 0$

$X \oplus 0 = X$

$X \oplus Y = Y \oplus X$

$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z)$.

→ Traversal.



We can traverse the XOR list on both forward & backward direction. While traversing the list we need to remember the address of previously accessed node in order to calculate the next node's address.

✓ eg. We are at C & want to move to B.

$$(B \oplus D) \oplus D = B \oplus (D \oplus D)$$
$$= B \oplus 0$$
$$= B.$$

eg. We are at C & want to move to D.

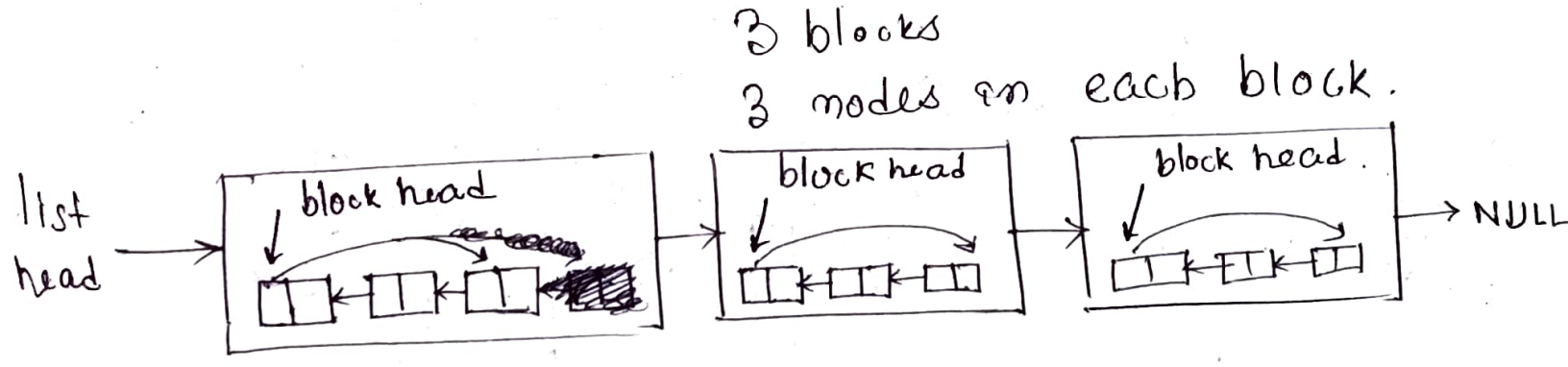
$$(B \oplus D) \oplus B = (D \oplus B) \oplus B$$
$$= D \oplus (B \oplus B)$$
$$= D.$$

→ Ans.

Insertion function
Traversal function
} GfG.

* Unrolled Linked List.

An unrolled Linked list stores multiple elements in each node or block. In each block, a circular linked list is used to connect all nodes.

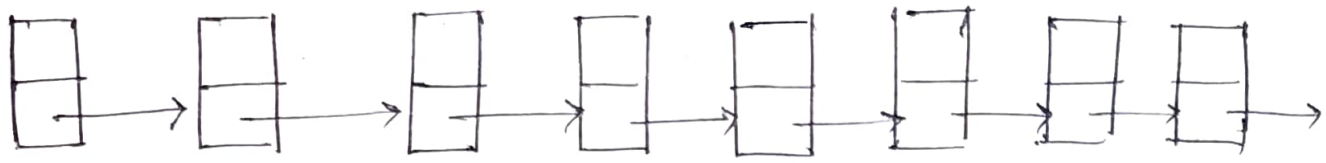


Number of nodes n (9).

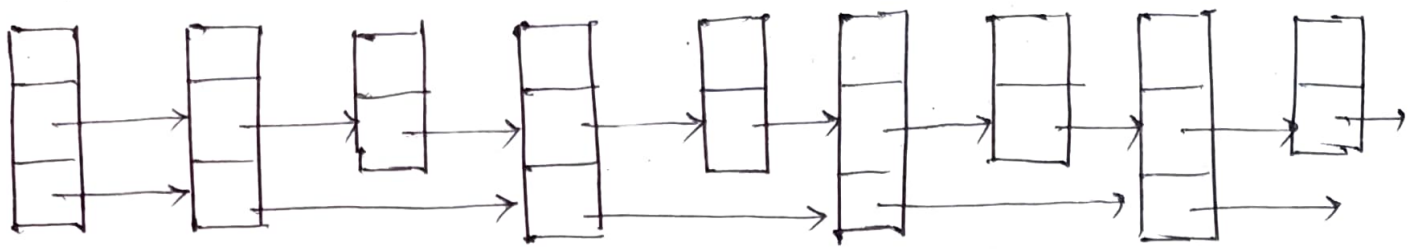
All blocks should contain exactly $\lceil \sqrt{n} \rceil$ elements. No more than $\lceil \sqrt{n} \rceil$ blocks at any time.

* Skip List

→ With one level.



→ With two levels.



* Reversing a singly linked list.

> Iterative. (rbr)

```
struct node {
    int i;
    struct node * link;
};
```

```
struct node *reverse (struct node *cur)
{
    struct node *prev = NULL;
    struct node *next = NULL;
    while (cur) {
        next = cur → link ; // save next
        cur → link = prev; // reverse
        prev = cur;
        cur = next;
    }
    return prev ; //new head @ end
}
```

advance prev & current

TC: O(n)

SC: O(1)

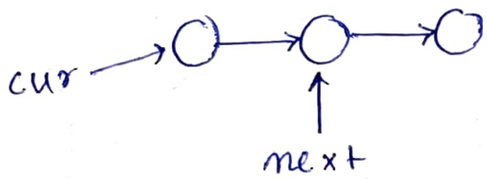
> Recursive (rbr)

```
struct node *head;
void reverse (struct node *prev,
              struct node *cur)
{ if (cur) {
      reverse (cur, cur→link);
      cur→link = prev;
  }
  else {
      head = prev;
  }
}
void main ( ) {
      :
      reverse (NULL, head);
      :
}
```

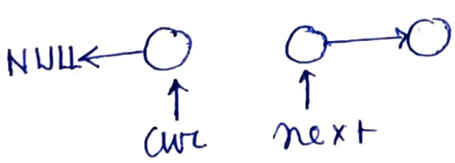
TC O(n)

SC O(n)

1. Save the next node's address to the next-ptr.

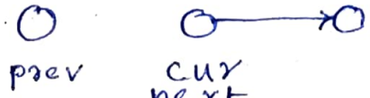


2. Reverse the connection, by making the current node's link/next point to the previous node.



Do these 3 steps for all nodes, to reverse the connections, hence the LL.

3. Go forward to next nodes by making prev point to current, & current to next.



Q. Display LL from the end or display in reverse.

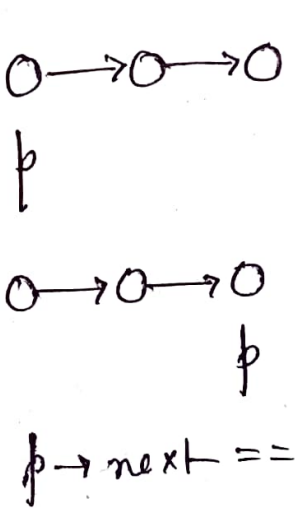
→ Use recursion. Traverse recursively till the end. While unrolling the recursion, print elements.

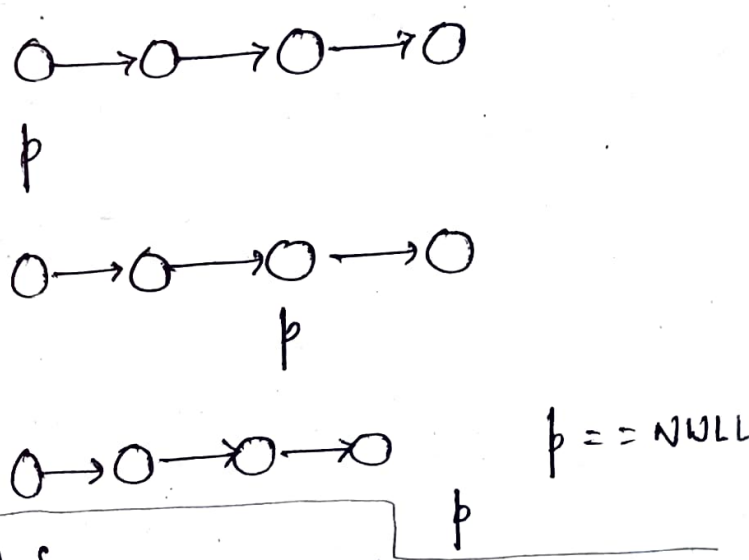
```
print ( node* head) {
    if (! head)
        return;
    print ( head → next) ;
    printf ("%d", head → data) ;
}
```

TC	O(n)
SC	O(n)

Q. Check whether LL length is even or odd.

→ Use a 2x ptr (2 nodes at a time). At the end, if ptr is NULL then even, otherwise odd.






```
int f ( node* head) {  node* h = head;
    while ( h && h → next) {
        h = h → next → next;
    }
    if (!h)  return 0; // odd
    return 1; // even
}
```

TC	O(n)
SC	O(1).

8. Merge 2 sorted LLs to a new sorted LL.

→ Maintain a head & a tail ptr on the merged list. Then choose the head of the merged LL by comparing the 1st node of both lists. For all subsequent nodes in both lists, & choose the smaller current node & link it to the tail of merged list, & moving the current ptr of that list one step forward. (Initially merged list is NULL).

```
node* merge(node* h1, node* h2){
    if (h1 == NULL)
        return h2;
    else if (h2 == NULL)
        return h1;
    node* mergehead = NULL;
    if (h1->data <= h2->data) {          // h1 data smaller than h2 data
        mergehead = h1;
        h1 = h1->next;
    } else {
        mergehead = h2;
        h2 = h2->next;
    }
    node* mergetail = mergehead;
    while (h1 && h2) {                   // next nodes
        node* temp = NULL;
        if (h1->data <= h2->data) {
            temp = h1;
            h1 = h1->next;
        } else {
            temp = h2; h2 = h2->next;
        }
        mergetail->next = temp;          // add next smaller elem to merged list.
        mergetail = temp;
    }
    if (h1) mergetail->next = h1;
    else if (h2) mergetail->next = h2;
    return mergehead;
}
```

TC $O(m+n)$

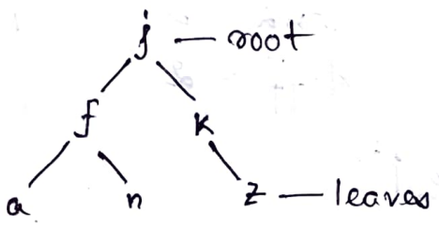
SC $O(1)$

DS1 - Th, Code DS2' - Adv., Th

TREES.

* Tree is a non-linear, hierarchical data structure.

* Tree Vocab. The topmost node is called root of the tree. The elements that are directly linked under an element are called its children. The element directly above some node is its parent.



Tree: Non-linear, hierarchical, undirected, acyclic, connected graph.

* Why trees?

i) One reason to use trees might be because we want to store information that naturally forms a hierarchy.

ii) Trees provide moderate access/search (quicker than list, slower than arrays).

iii) Trees provide moderate insertion/deletion. (quicker than arrays & slower than unordered linked list).

iv) Like linked list & unlike array, tree doesn't have an upper limit on number of nodes as nodes are linked using pointers.

* Application of Tree:

1. Manipulate hierarchical data.
2. Make information easy to search.
3. Manipulate sorted list of data.

Bi-tree: Binary tree (used here).

4. As a workflow for compositing digital images for visual effects.

5. Router algorithms.

6. Form of a multi-stage decision making.

* Binary Tree: A tree whose elements have at most 2 children is called a binary tree.

Representation. A tree is represented by a pointer to the topmost node in tree. If tree is empty, value of root is NULL.

A tree node contains following atoms -

i) data
ii) pointer to left child
iii) pointer to right child.

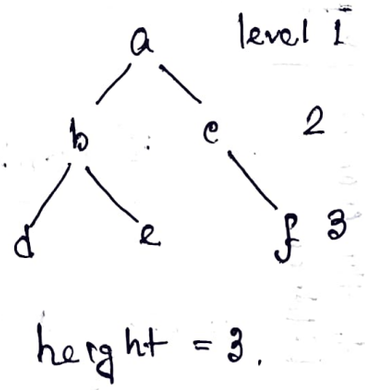
```
struct bitreenode {
    int data;
    struct bitreenode *left;
    struct bitree node *right;
};
```

Properties.

1. The maximum number of nodes at level l of a binary tree is 2^{l-1}.

(Proof by Induction)

2. Maximum number of nodes in a binary tree of height h is $2^h - 1$.



height = 3.

• [There is only one path from the root node to any node in a tree.

3. In a binary tree with N nodes, minimum possible height or minimum number of levels is $\log_2(N+1)$. [Corollary]

4. A binary tree with L leaves has at least $\log_2 L + 1$ levels.

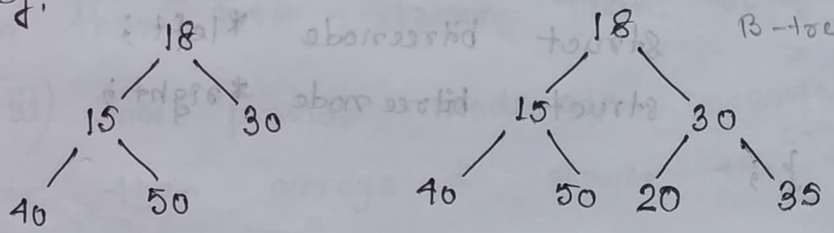
5. In binary tree where every node has 0 or 2 children, number of leaf nodes is always one more than nodes with two children. (Handshaking lemma)

Types.

i) Full binary tree (Strict binary tree): A binary tree is full if every node has 0 or 2 children. All nodes except leaves have two children.

• According to Karumanchi, if every node has two children & all leaf nodes are at same level. And the above defn is called as Strict B-tree.

eg.



• In a full binary tree no. of leaf nodes is no. of internal nodes plus 1. (Handshaking lemma).

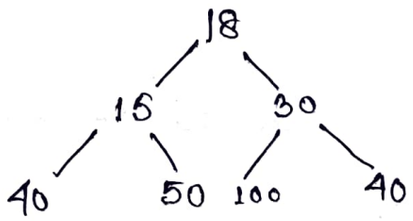
ii) Complete Binary Tree: If all leaves are completely filled except possibly the last level & the last level has all keys as left as possible.

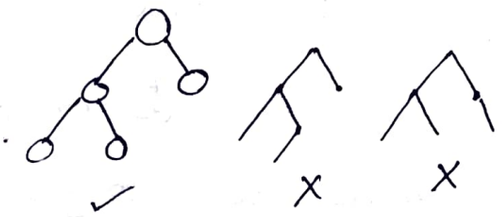
Here, if we give numbering for the nodes by starting at the root (1) then we get a complete sequence from 1 to the no. of nodes in a tree. While traversing we should give numbering to the NULL pointers also. All leaf nodes are at height h or h-1 & also without any missing number in sequence.

Full - every node has 2 children except leaves; leaves are at same level.

Strict - Strictly 0 or 2 children.

eg.

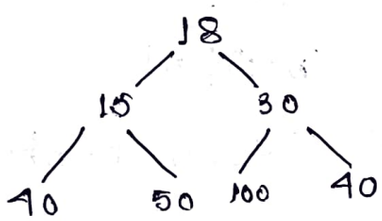




- Practical example of complete bi tree is binary heap.

iii) Perfect B-tree. In which all internal node have two children & all leaves are at same level. (or full)

eg.



- A perfect B-tree of height h has $2^h - 1$ node.

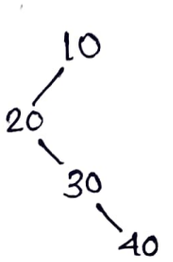
✓ iv) Balanced B-tree. If height of the tree is $O(\log n)$ where n is no. of nodes. eg. AVL tree maintains $O(\log n)$ height by making sure that the difference between heights of left & right subtree is 1.

Balanced binary search trees are performance wise good as they provide $O(\log n)$ time for search, insert, delete. ✓

v) Degenerate/Pathological Tree. (Skew tree)

Where every internal node has one child. These are performance wise same as linked list.

eg.



Handshaking Lemma (Undirected Graph)

In every finite undirected graph number of vertices with odd degree is always even.

It's a consequence of degree sum formula

$$\sum deg(v) = 2|E|$$

Summation of all degrees of vertices — Number of edges multiplied by two.

- In a K-ary tree where every node has either 0 or K children following holds.

$$L = (K-1)I + 1$$

L → No. of leaf nodes
I → No. of internal nodes.

- In Binary tree, no. of leaf nodes is always one more than nodes with two children. $L = T + 1$

- Let T be a nonempty, full binary tree, then

i) If T has I internal nodes, total no. of nodes is $N = 2I + 1$

ii) If T has a total of N nodes, no. of internal nodes is $I = (N-1)/2$; no. of leaves is $L = (N+1)/2$

iii) If T has L leaves, total no. of nodes is $N = 2L - 1$; no. of internal nodes $I = L - 1$.

→ Number of nodes (N), no. of leaves (L), no. of internal nodes (I) — any one known, we can determine other two.

$$N = 2I + 1 = 2L - 1$$

Q. G'15 Consider a binary tree that has 200 leaf nodes. Then number of nodes in T that have exactly two children are — $200 - 1 = 199$. $[L = N_2 + 1]$

Q. G'15 A binary tree T has 20 leaves. Number of nodes in T having two children is 19.

For a k-ary tree, $\boxed{\text{Sum of all degrees} = 2|E|}$ (*)

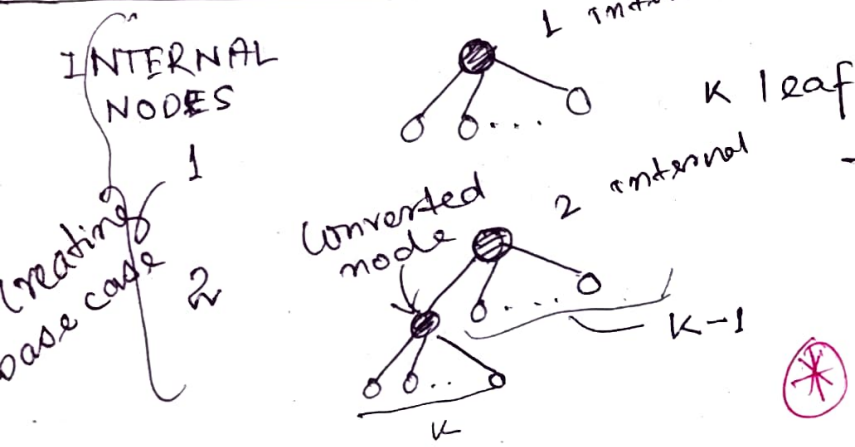
Sum of degrees of leaves + Sum of degrees for internal nodes except root + root's degree $= 2 \times (\text{No. of nodes} - 1)$

$\Rightarrow L + (I-1)(K+1) + K = 2 \times (L + I - 1)$.

$\Rightarrow L = (K-1)I + 1$.

$K = 2 \Rightarrow L = I + 1 \Rightarrow I = L - 1$

Q. G'05 In a complete k-ary tree, every internal node has exactly k children. No. of leaves in such a tree with n internal nodes is $(K-1)n + 1$.



(*)

$T(n) \rightarrow$ No. of leaf nodes when n internal nodes

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

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

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

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

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

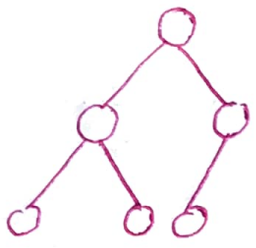
$= T(1) + (n-1)(K-1)$

$= K + (n-1)(K-1)$ $\quad T(1) = K$

$= n(K-1) + 1$

• In a complete binary tree, all leaves are at height h or h-1.

1. Max #nodes at level $l = 2^{l-1}$



2. Max #nodes in a binary tree with height $h = 2^h - 1$

$\Rightarrow$ Min. height / Min. no. of levels in a binary tree with n nodes =

$n = 2^h - 1 \Rightarrow h = \log_2(n+1)$

3. In a binary tree, $n_0 = n_2 + 1$.

#leaves = (#nodes with degree 2) + 1

4. A binary tree with l leaves, has at least $(\log_2 l + 1)$ levels.

5. In a binary tree where every node has 0 or 2 children, #leaves = (# non-leaves) + 1.

(Handshaking lemma)

6. In a k-ary tree where every every node has 0 or k children, $L = (k-1)I + 1$

L #leaves
I # internal nodes.

*7. In a binary tree where every node has 0 or 2 children (strict binary tree),

i) Total #nodes = 2 (# internal nodes) + 1 | $N = 2I + 1$

ii) Total #nodes = 2 (# leaves) − 1 | $N = 2L - 1$

iii) #leaves = # internal nodes + 1. | $L = I + 1$

$N = 2I + 1 = 2L - 1$

$L = I + 1$.

10. #NULL links = #nodes + 1.

IP	$2I + 1$
LM	$2L - 1$

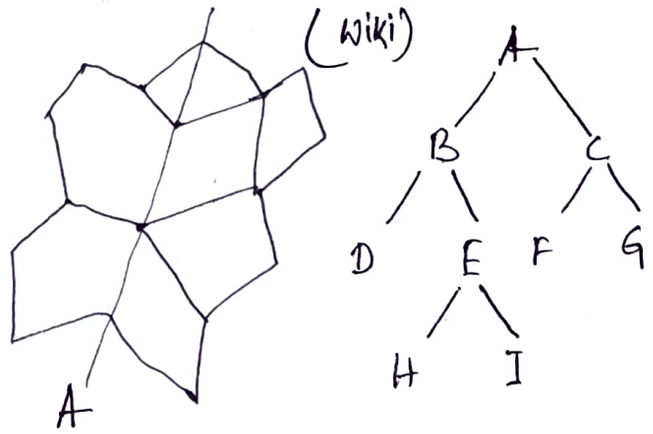
8. $\sum deg_v = 2E$

9. $\# v = \# E + 1$

• Application of Binary Trees.

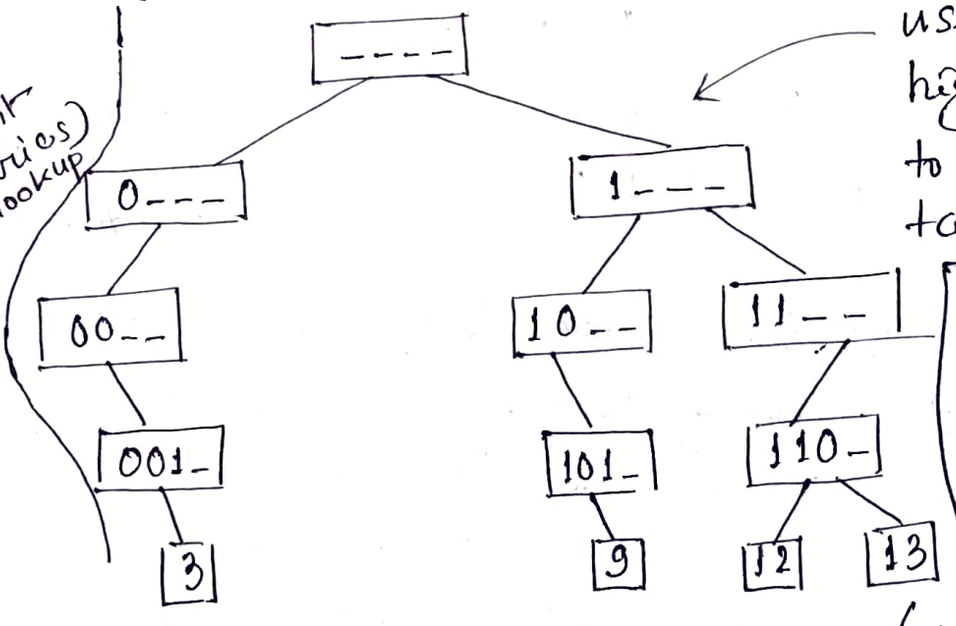
1. BST : Used in search applications where data is constantly entering/leaving. (map, set objects)

2. BSP tree (Binary space partition) : Recursively subdividing a space into 2 convex sets by using hyperplanes as partitions.



Used in 3d video games to determine what objects need to be rendered.

3. Binary tries : Encodes a set of w bit integers in binary tree. All leaves have depth w & each integer is encoded as root to leaf path.



used in high-bw router to store routing tables.

• Trie (Implement dictionaries) prefix lookup

• Suffix tree Pattern searching

• Spanning tree (used in routers & bridges)

4. Heaps : Implementing priority queues (used in scheduling processes). Heap sort. A* path finding algo.

5. Huffman coding trees : Used in compression algos.

6. Syntax tree (parsing)

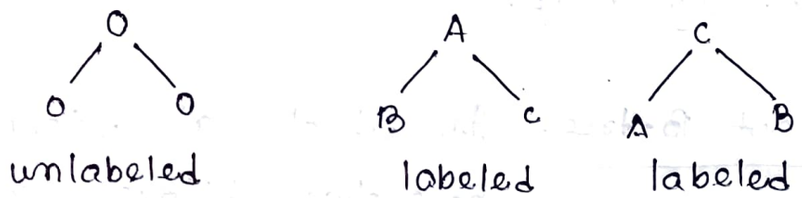
7. Treap (Wireless networking, memory allocations)

8. Store hierarchical data (XML/HTML data)

9. Indexing in DB (B tree, B+ tree)

Enumeration.

A binary tree is labeled if every node is assigned a label & binary tree is unlabeled if nodes are not assigned any label.



unlabeled labeled labeled

* • Number of unlabeled binary trees with n nodes $= \frac{(2n)!}{(n+1)!\,n!}$

$\frac{1}{n+1}\binom{2n}{n}$

[n^{th} Catalan number]

$$C_n = \binom{2n}{n} - \binom{2n}{n+1}.$$

* • Number of labeled binary trees with n nodes $= C_n \times n!$

Each unlabeled tree can be arranged in $n!$ ways

[$C_n \rightarrow n^{th}$ catalan number]

* Insertion.

• Given a tree & key, inserting the key into the Bi tree at first position available in level order.

Iterative level order traversal of the tree using queue.

* Deletion.

• Given a binary tree, deleting a node from it by making sure that tree shrinks from the bottom.

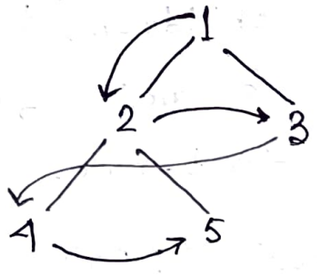
1. Starting at root, find the deepest & rightmost node in Bi-tree & node which we want to delete.

2. Replace the deepest rightmost node's data with node to be deleted.

3. Delete the node.

Traversal.

1. Breadth First Traversal (BFT, level order traversal).



1 2 3 4 5

* Algo 1.

Using two functions. One is to print all nodes at a given level (print given level) & other is to print level order traversal of the tree (print level order). printlevelorder makes use of printgivenlevel to print nodes at all levels one by one starting from root.

```
printlevelorder (tree)
  for d=1 to height (tree)
      printgivenlevel (tree, d);
printgivenlevel (tree, level)
  of tree is NULL then return
  if level is 1 then
          print (tree → data)
  else if level greater than 1, then
          printgivenlevel (tree→left, level-1);
          printgivenlevel (tree → right, level-1);
```

Algo 2. For each node first the node is visited & then its child nodes are put in a FIFO queue.

printlevelorder (tree)

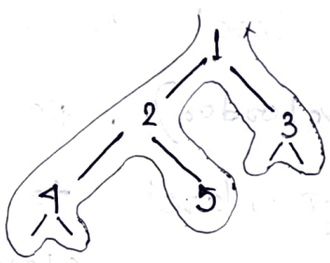
1. Create an empty queue q
2. temp_node = root
3. loop while temp_node is not NULL
   a) print temp_node → data
   b) Enqueue temp_node's children (first left then right children) to q
   c) Dequeue a node from q & assign it's value to temp_node.

2. Depth First Traversal (DFT)

i) Preorder Traversal (Root - left - Right)

ii) Inorder Traversal (Left - root - right)

iii) Postorder Traversal (left - right - root)



Preorder - 1 2 4 5 3 | 1st visit

Inorder - 4 2 5 1 3 | 2nd visit

Postorder - 4 5 2 3 1 | 3rd visit

i) Preorder Traversal -

a) visit the root

b) traverse the left subtree, i.e. call preorder (left-subtree)

c) traverse the right subtree, i.e. call preorder (right-subtree)

ii) Inorder Traversal –

a) Traverse the left subtree i.e. call inorder (left-subtree)

b) visit the root

c) traverse the right subtree i.e. call inorder (right-subtree)

iii) Postorder Traversal –

a) Traverse the left subtree; call postorder (left-subtree)

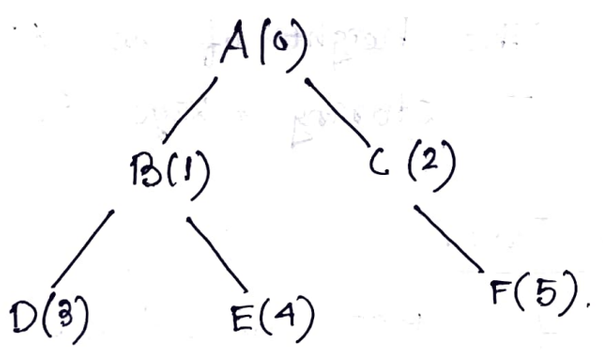
b) Traverse the right subtree ; call postorder (right-subtree)

c) Visit the root.

- All four traversals require O(n) time as they visit every node exactly once.

Array Implementation.

Sequential representation.

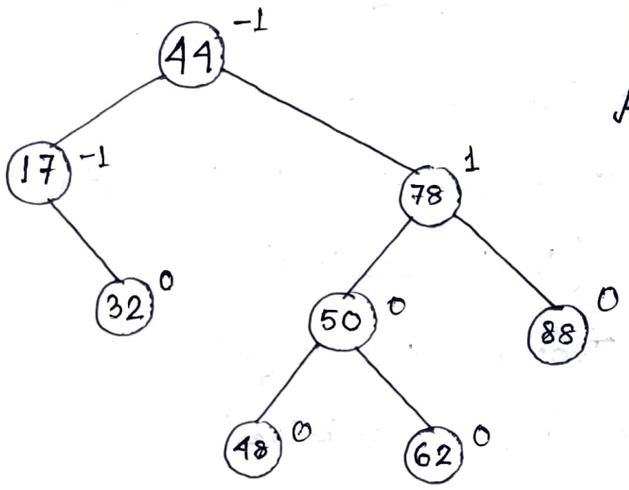


0 to n-1.

* AVL Tree. (More later)

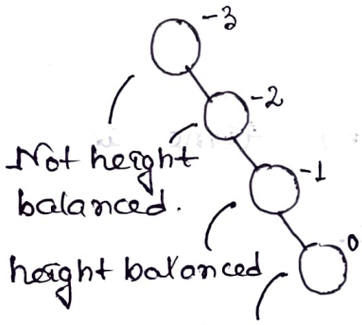
→ AVL Trees are balanced.

→ An AVL Tree is a BST such that for every internal node v of tree, the heights of the children of v can differ by at most 1.



Load factor {-1, 0, 1}

An AVL Tree



Not an AVL Tree.

→ Height of an AVL Tree.

Proposition. The height of an AVL tree T storing n keys is $O(\log n)$

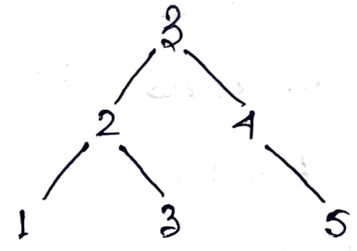
→ Why AVL Tree.

Most of the BST operations take O(h) time. Cost of this can become O(n) for a skewed Bi-tree. Height of AVL tree is always $O(\log n)$. So, we can guarantee an upper bound of $O(\log n)$ for all the operations.

* Continuous Tree.

A tree is continuous if on each root to leaf path, absolute difference between keys of two adjacent nodes is 1.

eg.

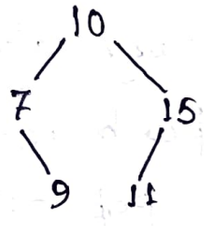


* Foldable Bi Tree:

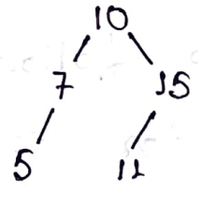
A tree can be folded if left & right subtrees of the tree are structure wise mirror image of each other.

An empty tree is considered as foldable.

eg.



Foldable

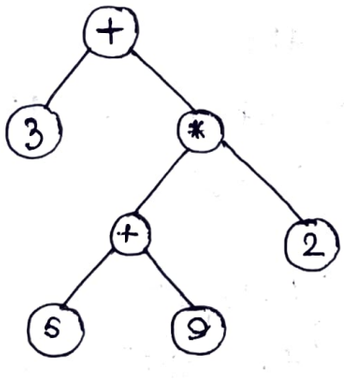


Not foldable.

* Expression Tree.

A binary tree in which each internal node corresponds to operator & each leaf node corresponds to operand.

eg. $3 + ((5+9) * 2)$.



Inorder

$3 + ((5+9) * 2)$

Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression).

- Evaluating the expression represented by expression tree -

Let t be the expression tree
If t is not null then
    If t.value is operand than
        Return t.value
    A = Solve (t.left)
    B = Solve (t.Right)

    return calculate (A, B, t.value).

- Construction of Expression Tree. -

Using a stack. Loop through input expression & do following for every character -

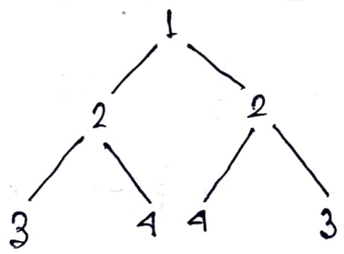
1. If character is operand push into stack
2. If character is operator pop two values from stack. Make them its child & push current node again.

At the end only element of stack will be root of expression tree.

- Evaluation of Expression Tree. -

Let t be the expression tree.
If t is not null then
    If t.value is operand then
        return t.value
    else
        A = Solve (t.left)
        B = solve (t.right)
        return A operator B

* Symmetric Tree.



Left subtree & right subtree are mirror images.

* Glossary.

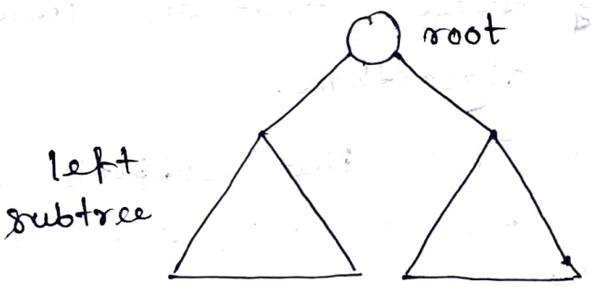
root, edge, leaf node, siblings, ancestor, descendant, level, depth (length of the path from the root to the node), height of tree (length of the path from the root to the deepest node), size of a node (number of descendants it has including itself), skew tree (every node except leaf nodes has only one child) - left & right skew trees.

* Binary Tree.

[Bi-Tree → Binary Tree]

→ Empty tree is a valid Bi-tree.

→ We can visualise a Bi-tree as consisting of a root & two disjoint Bi-trees called the left & right subtrees of the root.



→ Properties. Height of tree is h. Root at height zero (just a convention, sometimes 1).

a) Number of nodes n in a full binary tree is $2^{h+1} - 1$ $[2^0 + 2^1 + 2^2 + \dots + 2^h]$

b) Number of nodes n in a complete binary tree is between 2^h (min) & $2^{h+1} - 1$ (max).

✓ c) Number of leaf node in a full Bi-tree is 2

✓ d) Number of NULL links in a complete Bi-tree of n nodes is $n+1$.

Operations on Bi-trees.

Basic Operations.

i) Inserting an element into a tree
ii) Deleting an element from a tree
iii) Searching for an element
iv) Traversing the tree.

Auxiliary Operations.

i) Finding the size of the tree
ii) Finding the height of the tree
iii) Finding the level which has maximum sum.
iv) Finding the least common ancestor (LCA) for a given pair of nodes

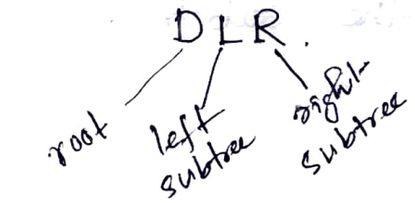
etc.

Application of Bi-tree.

i) Expression trees are used in compilers.
ii) Huffman coding trees that are used in data compression algorithms.
iii) Binary Search Tree that supports search, insertion & deletion on a collection of items in $O(\log n)$ (Average)
iv) Priority queue that supports search & deletion of minimum (or maximum) on a collection of items in logarithmic time (in worst case).

Binary Tree Traversals.

i) Preorder Traversal.



Each node is processed before either of its subtrees.

[c]

Recursion –

```
void PreOrder ( struct BinaryTreeNode *root)
{
    if (root) {
        printf ("%d\t", root->data);
        PreOrder ( root -> left);
        PreOrder ( root -> right);
    }
}
```

```
struct BinaryTreeNode {
    int data;
    struct BinaryTreeNode
        *left, *right;
};
```

Time Complexity – O(n)

Space Complexity – O(n).

Non-recursive.

```
void PreOrder ( struct BinaryTreeNode * root) {
    struct Stack *S = Create_Stack ( );
    while (1) {
        while (root) {
            printf ("%d\t", root->data);
            Push (S, root);
            root = root -> left;
        }
        if (IsEmpty Stack (S))
            break;
        root = Pop (S);
        root = root -> right;
    }
    Delete Stack (S);
}
```

Time Complexity : O(n)

Space complexity : O(n)

ii) Inorder Traversal.

LDR — The root is visited between the subtrees.

C

Using Recursion.

```
void InOrder (struct BinaryTreeNode *root) {
    if (root) {
        InOrder (root -> left);
        printf ("%d\t", root -> data);
        InOrder (root -> right);
    }
}
```

Time Complexity - O(n)
Space Complexity - O(n)

Non-recursive.

```
void InOrder (struct BinaryTreeNode *root) {
    struct Stack *S = CreateStack ();
    while (1) {
        while (root) {
            Push (S, root);
            root = root -> left;
        }
        if (IsEmptyStack (S))
            break;
        root = Pop (S);
        printf ("%d\t", root -> data);
        root = root -> right;
    }
    DeleteStack (S);
}
```

Time Complexity - O(n)
Space complexity - O(n)

iii) Postorder Traversal.

LRD. Root is visited after both subtrees.

C

By recursion:-

```
void PostOrder (struct BinaryTreeNode *root) {
    if (root) {
        PostOrder (root -> left);
        PostOrder (root -> right);
        printf ("%d\t", root -> data);
    }
}
```

Time Complexity - O(n)

Space Complexity - O(n).

Non-recursive.

```
void PostOrder (struct BinaryTreeNode *root) {
    if (!root)
        return;
    struct Stack *S = CreateStack();
    Push (S, root);
    struct BinaryTreeNode *previous = NULL;
    while (!IsEmptyStack(S)) {
        struct BinaryTreeNode *current = Pop(S);
        if (!previous || previous -> left == current ||
                         previous -> right == current) {
            if (current -> left)
                Push (S, current -> left);
            else if (current -> right)
                Push (S, current -> right);
        }
```

```
        else if (current -> left == previous) {
            if (current -> right)
                Push (S, current -> right);
        }
        else {
            printf ("%d\t", current -> data);
            Pop(S);
        }
        previous = current;
    }
}
```

Time Complexity - O(n)

Space Complexity - O(n)

iv) Level Order Traversal.

→ Visit the root.

→ While traversing level l, keep all the elements at level l+1 in queue.

→ Go to the next level & visit all the nodes at that level.

→ Repeat.

(c)

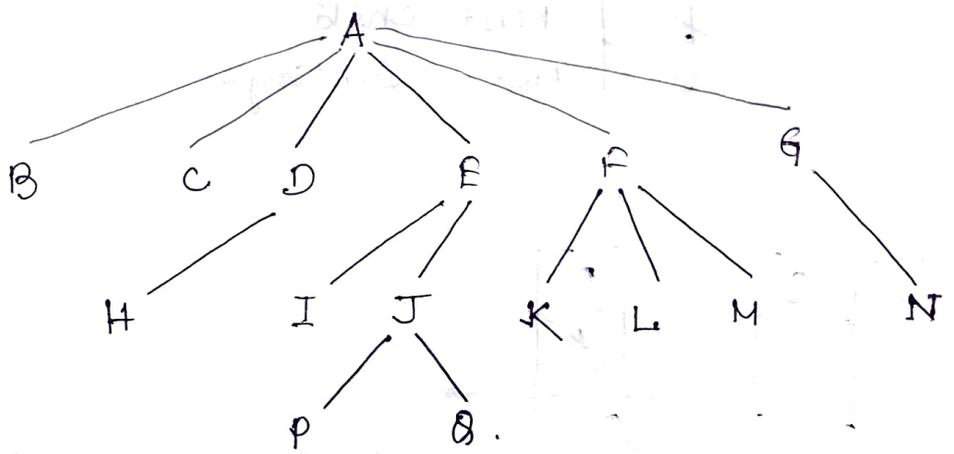
```
void LevelOrder (struct BinaryTreeNode *root) {
    struct BinaryTreeNode *temp;
    struct Queue *Q = CreateQueue ();
    if (!root)
        return;
    EnQueue (Q, root);
```

```
while (!IsEmptyQueue(Q)) {
    temp = DeQueue(Q);
    printf("%d ", temp->data);
    if (temp->left)
        EnQueue(Q, temp->left);
    if (temp->right)
        EnQueue(Q, temp->right);
}
DeleteQueue(Q);
}
```

TC – O(n)

SC – O(n)

* Generic Trees (N-ary Trees).

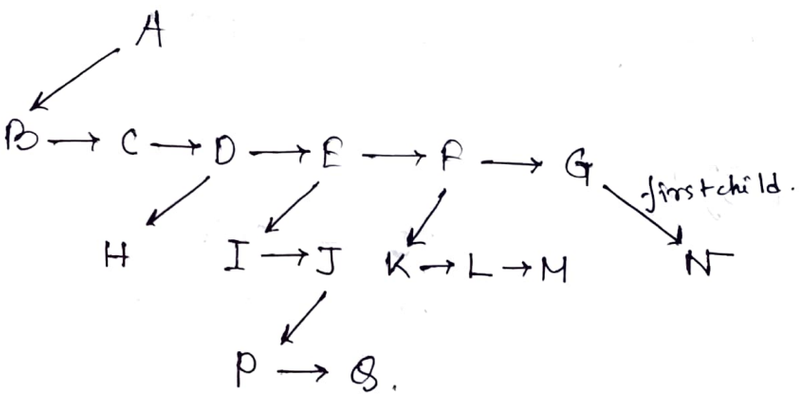


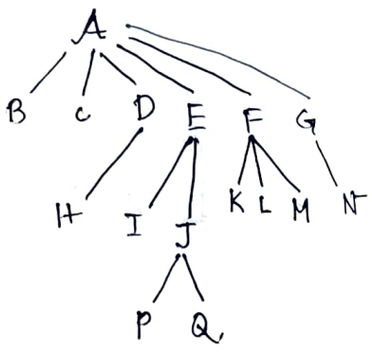
→ Node representation with worst case.

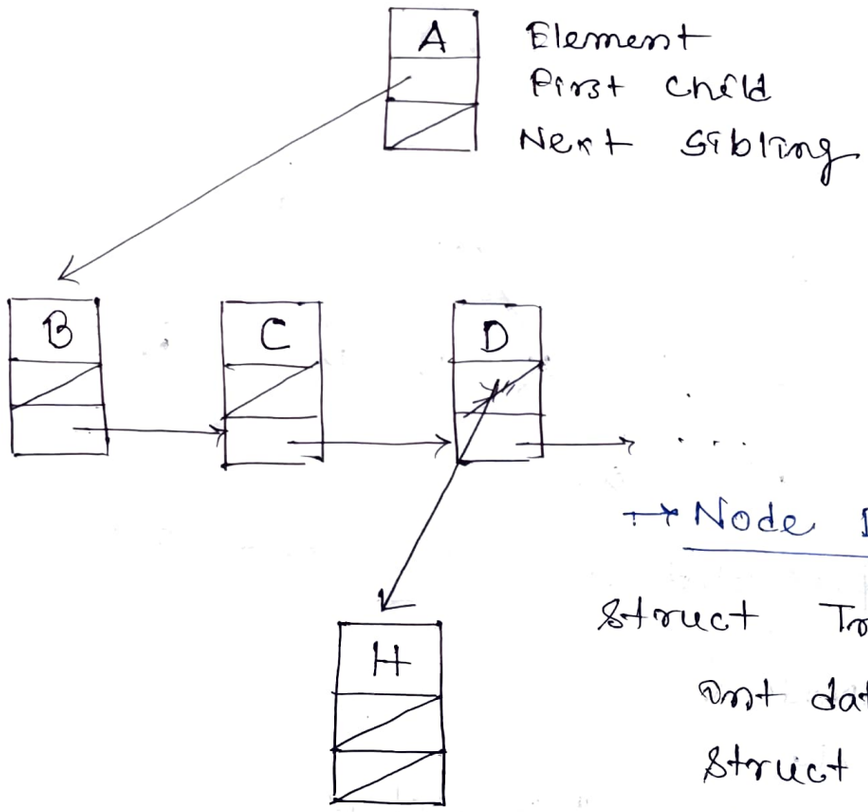
```
struct TreeNode {
    int data;
    struct TreeNode * firstChild;
    .
    .
    .
    struct TreeNode * fifthchild;
    struct TreeNode * sixthchild;
};
```

→ First child / Next Sibling Representation

- At each node link children of same parent (siblings) from left to right.
- Remove the links from parent to all children except the first child.







→ Node Declaration.

```
struct TreeNode {
    int data;
    struct TreeNode * firstchild;
    struct TreeNode * nextsibling;
};
```

* Threaded Binary Tree Traversals

(Stack / Queue less Traversals).

→ Issues with Regular Binary Tree Traversals –

The storage space required for the stack & queue is large. The majority of pointers in any binary tree are NULL. For example, a binary tree with n nodes has n+1 NULL pointers & these were wasted. It is difficult to find successor node for a given node.

→ Motivation for Threaded Binary Tree. –

A number of nodes contain a NULL pointer, either in their left or right fields or in both. This space that is wasted can be efficiently used to store some other useful information. For example, they can be replaced to store a pointer to the inorder predecessor or the inorder successor of the node.

These special pointers are called Threads; binary trees containing threads are called threaded trees.

→ Classification: Based on whether we are storing info in both NULL pointers or only in one.

i) If we store predecessor information in NULL left pointers only then we call them left threaded binary trees. (One way threading)

ii) If we store successor information in NULL right pointers only then we can call such binary trees right threaded binary trees. (One way threading)

iii) If we store predecessor & successor informations in NULL left & right pointers, then we call them fully threaded binary trees or simply threaded binary trees. (Two way threading)

→ Types of Threaded Binary Trees.

i) Preorder threaded binary trees.

NULL left pointer will contain Pre Order predecessor information & NULL right pointer will contain preorder successor information.

ii) Inorder threaded binary trees

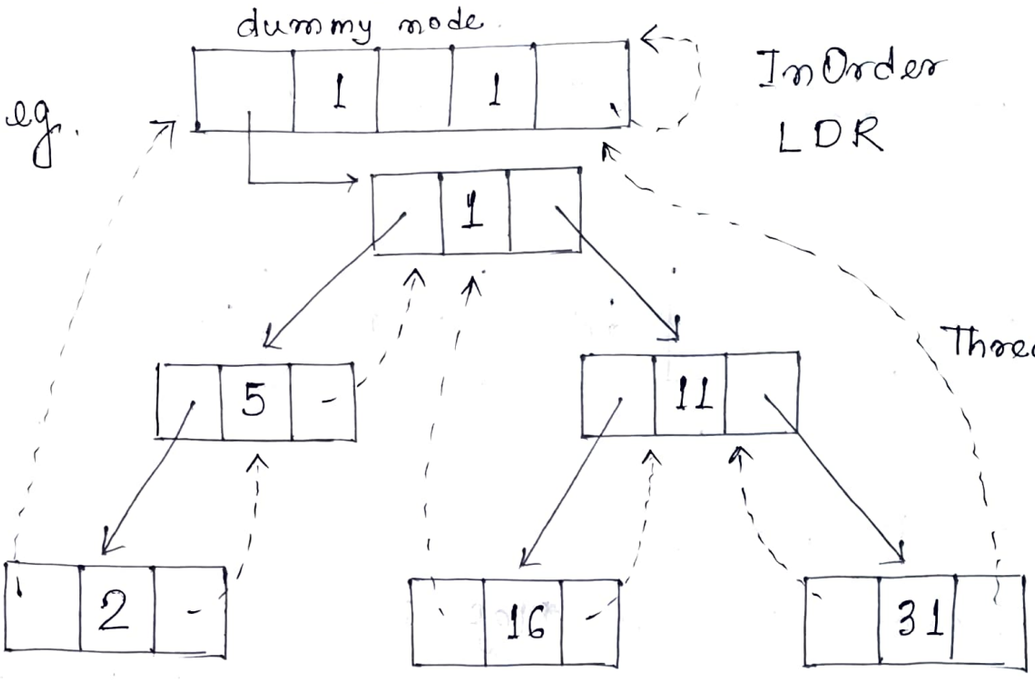
iii) Postorder threaded binary trees.

→ Threaded Binary Tree Structure.


```
Struct TBTNode {
    Struct TBTNode *left;
    int LTag;   // indicating link to the predecessor
    int data;
    int RTag;   // indicating link to the successor
    Struct TBTNode *right;
};
```

→ Difference between Binary Tree & Threaded Binary Tree Structures. :

	Regular	TBT.
LTag == 0	NULL	left points to the inorder predecessor.
LTag == 1	left points to the left child	left points to left child.
RTag == 0	NULL	right points to the inorder successor
RTag == 1	right points to the right child	right points to the right child.



[Convenient to use to special dummy node that is always present .]

→ Finding InOrder successor in InOrder Threaded Binary Tree.

If node has no right subtree, then return the right child of node. If the node has right subtree, return the left of the nearest node whose left subtree contains the node of which we need to find the successor.

Code

```
struct TBTNode *InSuccessor (TBTNode *P) {
    TBTNode * position ;
    if ( P → RTag == 0)
        return P → right ;
    else {
        Position = P → right ;
        while ( Position → LTag == 1)
            Position = Position → left ;
        return Position ;
    }
}
```

TC - O(n)

SC - O(1)

→ InOrder Traversal in Inorder TBT

Code

```
void InOrderTraversal ( TBTNode *root) {
    TBTNode *P = InSuccessor (root);
    While ( P != root) {
        P = InSuccessor (P);
        printf ( "%d", P → data);
    }
}
```

```
void InOrderTraversal ( TBTNode *root) {
    TBTNode *P = root;
    while (1) {
        P = InSuccessor (P);
        if (P == root)
            return;
        printf ( "%d ", P → data );
    }
}
```

TC - O(n)
SC - O(1)

→ Finding PreOrder Successor in InOTBT.

If P has a left subtree, then return the left child of P. If P has no left subtree, return the right child of the nearest node whose right subtree contains P.

Code

```
void TBTNode * PreSuccessor (TBTNode *P) {
    TBTNode *Position;
    of (P → LTag == 1)
        return P → left;
    else {
        Position = P;
        while (Position → RTag == 0)
            Position = Position → right;
        return Position → right;
    }
}
```

SC - O(1)
TC - O(n)

→ PreOrder Traversal of InOTBT.

Code

```
void PreOrder_Traversal ( TBTNode *root) {
    TBTNode *P;
    P = PreSuccessor (root);
    while ( P != root) {
        P = PreSuccessor (P);
        printf ("%d ", P→data);
    }
}
```

TC - O(n)
SC - O(1)

```
void PreOrder Traversal ( TBTNode *root) {
    TBTNode *P = root;
    while (1) {
        P = PreSuccessor (P);
        if ( P == root)
            return;
        printf ("%d ", P→data);
    }
}
```

→ Insertion of Nodes in InOTBT:

Code

```
void InsertRightInTBT (TBTNode *P, TBTNode *Q) {
    TBTNode * temp;
    Q → right = P → right;
    Q → RTag = P → RTag;
    Q → left = P;
    Q → LTag = 0;
    P → right = Q;
    P → RTag = 1;
    if (Q → RTag == 1) {
        temp = Q → right;
        while (temp → LTag) {
            temp = temp → left; }
        temp → left = Q; }
}
```

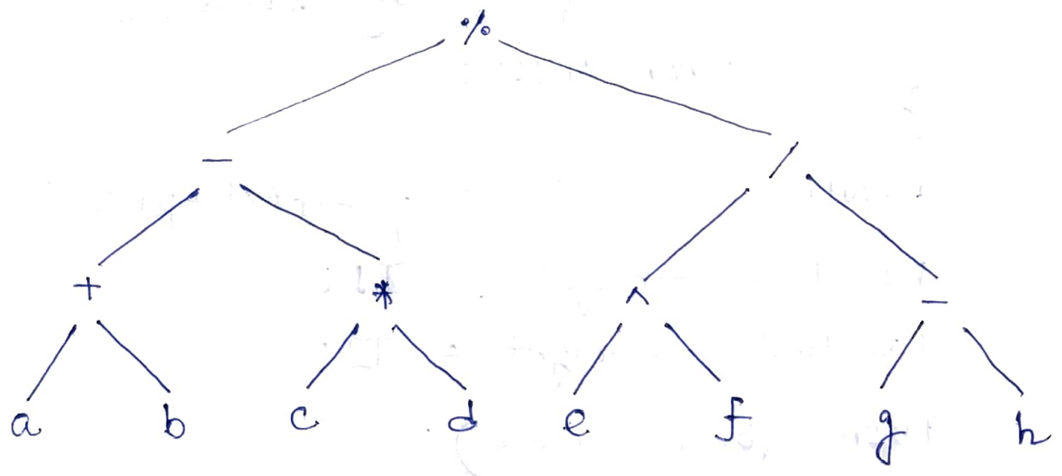
TC - O(n)
SC - O(1)

* Expression Trees.

→ In expression trees, leaf nodes are operands & non-leaf nodes are operators. Internal nodes are operators & leaves are operands.

→ eg.

((a+b)-(c*d)) % ((e^f)/(g-h)).



→ Building expression tree from Postfix Expression.

```
BTNode *BuildExpT (char postfixExp[], int size) {
    Stack *S = CreateStack (size);
    for (int i = 0; i < size; i++) {
        if (postfixExp[i] is an operand) {
            BTNode *newNode = (BTNode *) malloc
                              (sizeof (BTNode));
            if (!newNode) {
                printf ("Memory error!");
                return NULL;
            }
            newNode → data = postfixExp[i];
            newNode → left = newNode → right = NULL;
            Push (S, newNode); }
```

```
        else {
            BTNode *T2 = Pop(S);
            BTNode *T1 = Pop(S);
            BTNode *newNode = (BTNode*)
                malloc(sizeof(BTNode));
            if (!newNode) {
                printf("Memory error!");
                return NULL;
            }
            newNode -> data = postfixExp[i];
            newNode -> left = T1;
            newNode -> right = T2;
            Push(S, newNode);
        }
    }
    return S;
}
```

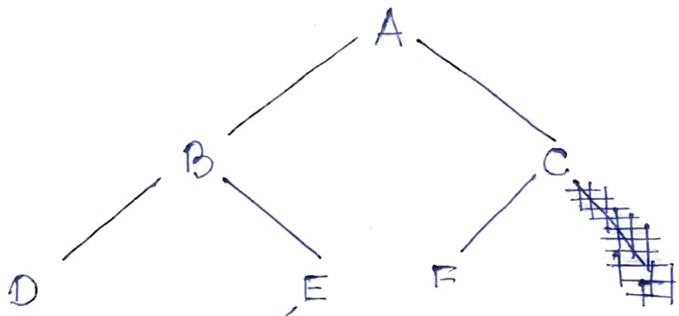
* XOR Trees.

→ For each node, rules to represent an XOR Trees -

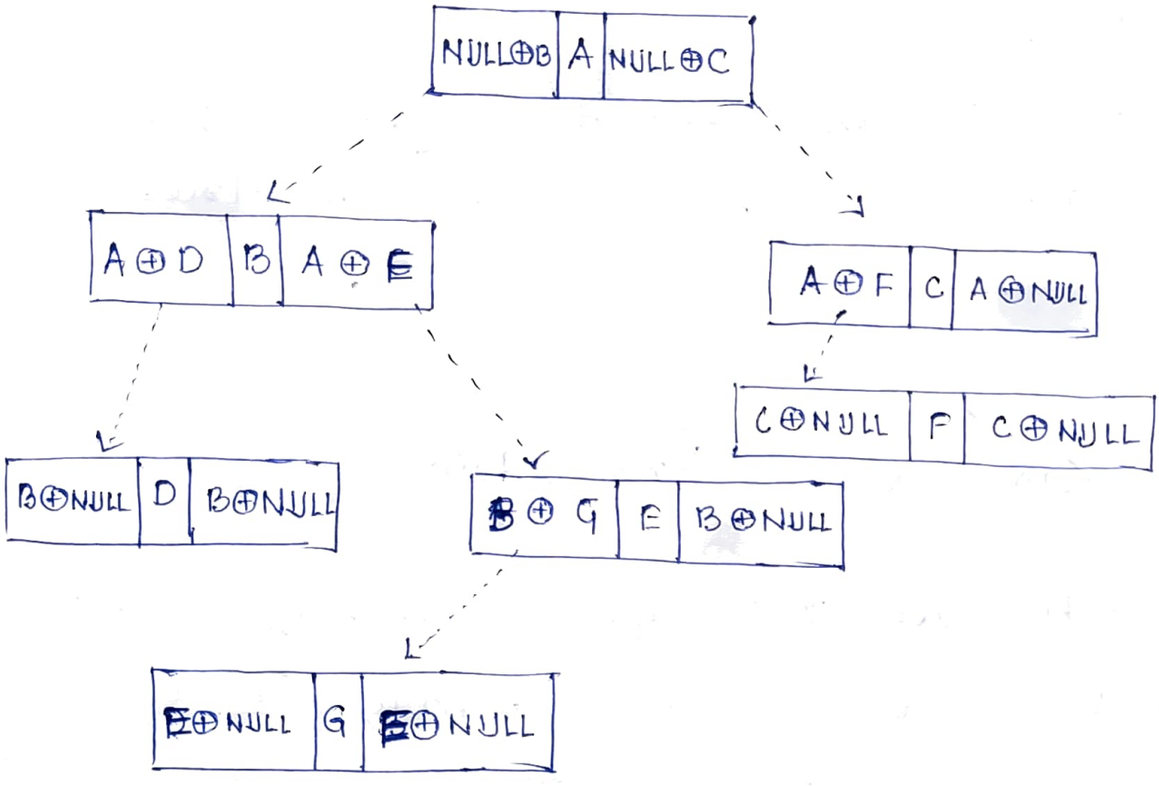
i) Each node's left will have the ⊕ of its parent & its left child.

ii) Each node's right will have the ⊕ of its parent & its right child.

iii) The root node's parent is NULL & also leaf nodes children are NULL nodes.



XOR Tree Representation



Assume, we are at B. If we have to go to A (parent) then $(A \oplus D) \oplus D =$

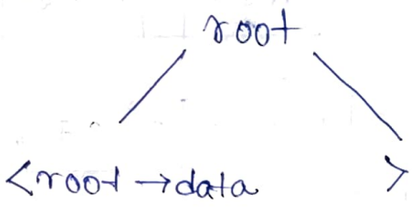
$A \oplus (D \oplus D) = A \oplus 0 = A$.

Similarly, if we need to go to D (left child) then $(A \oplus D) \oplus A = D$.

* Binary Search Trees (BSTs).

→ BST Property.

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left & right subtrees must also be binary search trees.



→ We use BST for searching. Unlike normal unrestricted binary trees, BST reduces search operation TC to O(log

→ Node Declaration.

```
typedef struct BST {
    int data;
    struct BST * left;
    struct BST * right;
} BSTNode;
```

→ Operations on BST

Main - Finding min/max element, inserting, deleting element.

Auxiliary - Finding k^{th} smallest element, sorting elements.

→ Notes

i) Since root data is always between left subtree data & right subtree data, performing inorder traversal on binary search tree produces a sorted list.

ii) The binary search trees consider either left or right subtrees for searching an element but not both.

→ Finding an Element in Binary Search Trees.

[c] Using recursion,

```
BSTNode *Find (BSTNode *root, int data) {
    if (root == NULL)
        return NULL;
    if (data < root → data)
        return Find (root → left, data);
    else if (data > root → data)
        return Find (root → right, data);
    return root;
}
```

TC - O(n), in worst case when BST is a skew tree

SC - O(n), for recursive stack.

Non-recursive version -

```
BSTNode *Find ( BSTNode *root, int data) {
    if (root == NULL)
        return NULL;
    if (data < root->data)
        ret
    while (root) {
        if (data == root->data)
            return root;
        else if (data > root->data)
            root = root->right;
        else
            root = root->left;
    }
    return NULL;
}.
```

TC - O(n)
SC - O(1)

→ Finding minimum element on BST

Minimum element is the left-most node, that does not have left child.

(c)

```
BSTNode *FindMin ( BSTNode *root) {
    if (root == NULL)
        return NULL;
    else if ( root->left == NULL)
        return root;
    else return FindMin (root->left);
}
```

SC - O(n) for recursive stack

TC - O(n), [Worst case when BST is left skew tree]

Non-recursive version.

```
BSTNode *FindMin ( BSTNode *root) {
    of (root == NULL)
        return NULL;
    while (root → left != NULL)
        root = root → left;
    return root;
}
```

→ Finding maximum element in BST.

The maximum element is the right-most node, that does not have right child.

C

```
BSTNode *FindMax ( BSTNode *root) {
    if (root == NULL)
        return NULL;
    else if ((root → right) == NULL)
        return root;
    else
        return FindMax (root → right);
}
```

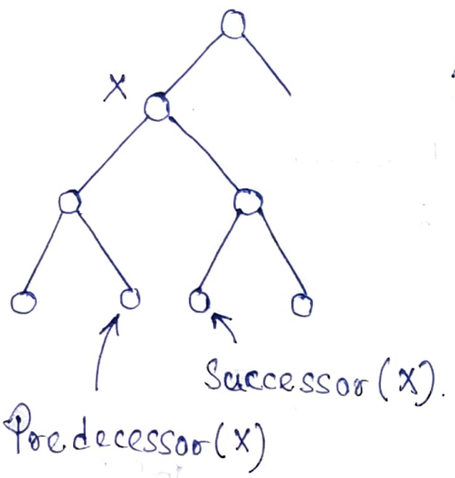
TC - O(n)
SC - O(n)

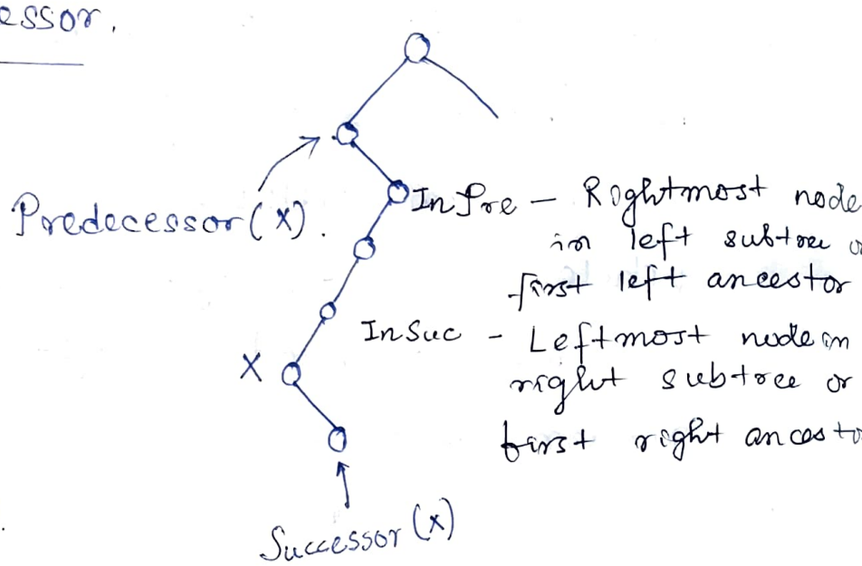
Non-recursive

```
BSTNode *FindMax ( BSTNode *root) {
    if (root == NULL)
        return NULL;
    while (root → right != NULL)
        root = root → right;
    return root;
}
```

TC - O(n)
SC - O(1)

→ Position of InOrder Predecessor & Successor.





→ Inserting an element

[C]

```
BSTNode *Insert (BSTNode *root, int data) {
    if (root == NULL) {
        root = (BSTNode *) malloc (sizeof (BSTNode)
        if (root == NULL) {
            printf ("Memory error!.");
            return;
        }
        else {
            root → data = data;
            root → left = root → right = NULL;
        }
    }
    else {
        of (data < root → data)
            root → left = Insert (root → left, data)
        else if (data > root → data)
            root → right = Insert (root → right, data)
    }
    return root;
}
```

TC - O(n)
SC - O(n).

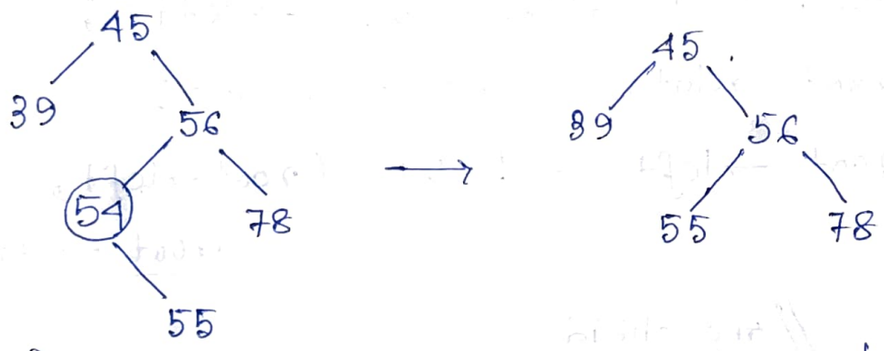
→ Deleting Node.

Different cases –

i) Deleting node that has no children –

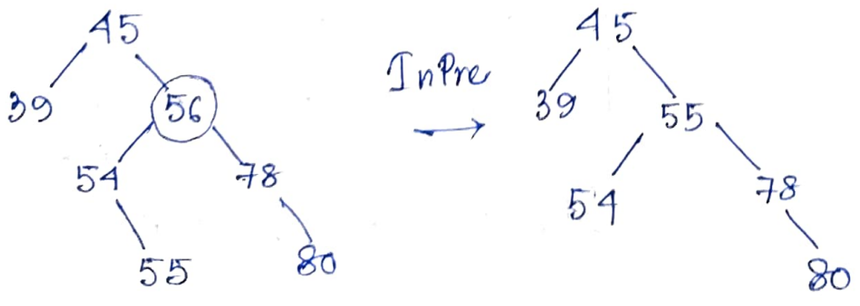
ii) Deleting node with one child –

The node's child is set as the child of the node's parent, i.e. replace the node with its child. Now, if the node is left child of its parent, the node's child becomes the left child of the node's parent. Correspondingly, for node with rc.



iii) Deleting node with two children –

Replace the node's value with its inorder predecessor (largest value in the left subtree) or inorder successor (smallest value in the right subtree). The inorder predecessor or successor can then be deleted using any of the above cases.



55 56 78

C

```
BSTNode *Delete (BSTNode *root, int data) {
    BSTNode *temp;
    if (root == NULL)
        printf ("Empty Tree");
    else if (data < root->data)
        root->left = Delete (root->left, data);
    else if (data > root->data)
        root->right = Delete (root->right, data);
    else {      //Found element
        if (root->left && root->right) {
            temp = findMax (root->left);
            root->data = temp->data;
            root->left = Delete (root->left,
                                 root->data);
        }
        else {  //one child
            temp = root;
            if (root->left == NULL)
                root = root->right;
            if (root->right == NULL)
                root = root->left;
            free (temp);
        }
    }
    return root;
}
```

TC - O(h)
SC - O(h), recursive stack

Iterative → SC - O(1)

Searching in BST.

```
node *bst_search (node *p, int k) {
    if (p != NULL) {
        if ( k < p→x)
            p = bst-search (p→left , k);
        else if (k > p→x)
            p = bst-search (p→right, k);
    }
    return p;
}
```

→ Time Complexity.

- Best Case : $T(n) = O(1)$, since the search key k may be the very root.
- Worst Case : $T(n) = T(n-1) + O(1) = O(n)$, which arises when the BST is fully skewed & the search terminates at the bottommost leaf node.
- Average case. :

Internal path length, $I(n)$ = Sum of path lengths of all internal nodes from the root of BST.

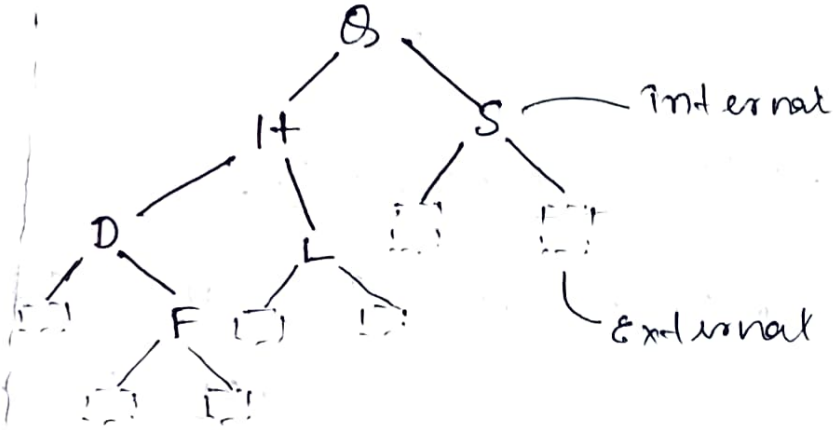
External Path Length, $E(n)$ = Sum of path lengths of all external (dummy) nodes from the root.

Average no. of comparison for successful search: $S(n)$

Avg. no. of comp. for unsuc. search : $U(n)$

Observe,

$$\begin{cases} S(n) = \dfrac{I(n) + n}{n} \\ U(n) = \dfrac{E(n)}{n+1} \end{cases}$$



Can be proved by Induction that,

$$\begin{cases} E(n) = I(n) + 2n \\ U(n) = \dfrac{I(n) + 2n}{n+1} \end{cases}$$

$I(n) = 1\times2 + 2\times2 + 3\times1 = 9$

$E(n) = 2\times2 + 3\times3 + 4\times2 = 21$

$S(n) = \dfrac{9+6}{6} = 2.5$

$U(n) = \dfrac{21}{7} = 3$

by eliminating $I(n)$,

$$nS(n) = (n+1)U(n) - n$$

$$\Rightarrow \boxed{S(n) = \left(1 + \frac{1}{n}\right)U(n) - 1}$$

To find the avg. no. of comparisons for successfully searching each of the n keys, we consider its insertion order. If a key x was inserted as i^{th} node, namely u_i, then the avg. no of comparisons for its successful search in that instance of the tree containing the preceding $(i-1)$ nodes is given by U_{i-1}. Once, it's inserted, we can successfully search for it & the search terminates at the node u_i, which was a dummy node where its unsuccessful search terminated. Hence, $S_i = U_{i-1} + 1$, as one extra comparison is required for the successful search terminating at u_i compared to unsuccessful search terminating at the dummy node located at the same position. We estimate the avg. no. of comparisons for all these n nodes based on their insertion orders. Thus, we get,

$$S(n) = \frac{1}{n} \sum_{i=1}^{n} (U_{i-1} + 1)$$

Now, $(n+1)U(n) = 2n + U(0) + U(1) + \cdots + U(n-1)$

To solve this recurrence rel^n, we substitute $n-1$ for n to get

$$nU(n-1) = 2(n-1) + U(0) + U(1) + \cdots + U(n-2)$$

Subtracting,

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

Since, $U(0) = 0$, we get $U(1) = 2/2$, $U(2) = \frac{2}{2} + \frac{2}{3}$, $U(3) = \frac{2}{2} + \frac{2}{3} + \frac{2}{4}$ & so on, rescaling to

$$U(n) = 2\left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n+1}\right) = 2H_{n+1} - 2$$

$$= 2 \ln n$$

$$= (2 \ln 2) \log_2 n$$

$H_n' = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ is the n^{th} harmonic number & approx equals $\ln n$.

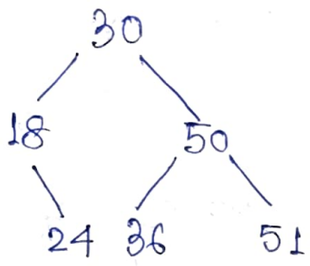
$$\boxed{S(n) \approx U(n) \approx (2 \ln 2) \log_2 n}$$

* Balanced Binary Search Tree

- Due to $\theta(n)$ complexity for worst case of Binary Search Tree, we look forward to height balanced tree.
- A node in a tree is height-balanced if the heights of its subtrees differ by no more than 1. If the subtrees have heights h_1 & h_2, then $|h_1 - h_2| \leq 1$.

A tree is height balanced if all of its nodes are height balanced. (An empty tree is height-balanced).

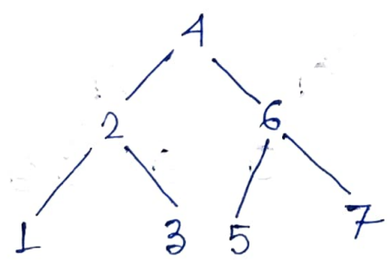
eg.



- Height-balanced trees are represented with HB(k), where k is the difference between left & right subtree height. k is called balance factor.

$$k \leq 1$$

- If for HB(k), $k = 0$, then it is full balanced binary search trees & it ensures it is a full binary tree.



HB(k), $k = 0$

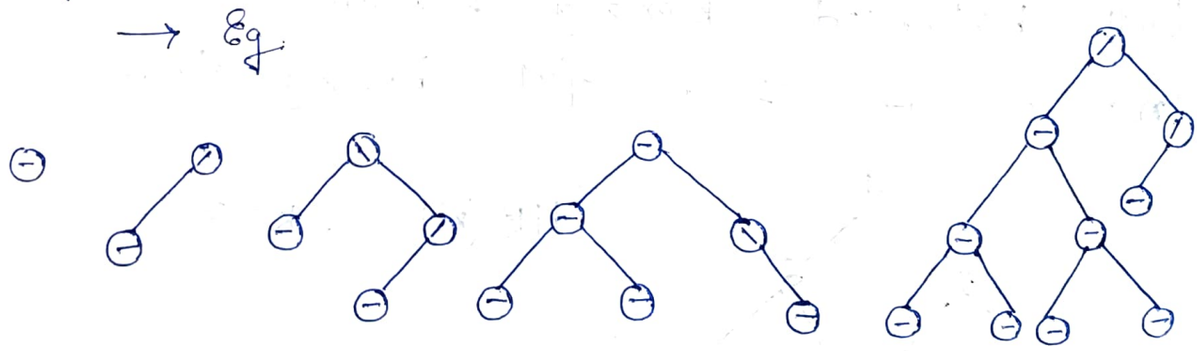
- It can be shown that a height-balanced tree with n nodes has height $\Theta(\log_2(n))$. Since the cost of our algorithms is proportional to the height of the tree, each operation (lookup, insertion, deletion) will take time $\Theta(\log_2(n))$ in the worst case.

* AVL (Adel'son-Vel'skii and Landis) Trees.

→ Definition. – An AVL tree is a binary search tree in which the heights of the left & right subtrees of the root differ by at most 1 & in which the left & right subtrees are again AVL trees. In HB(k), $k \leq 1$.

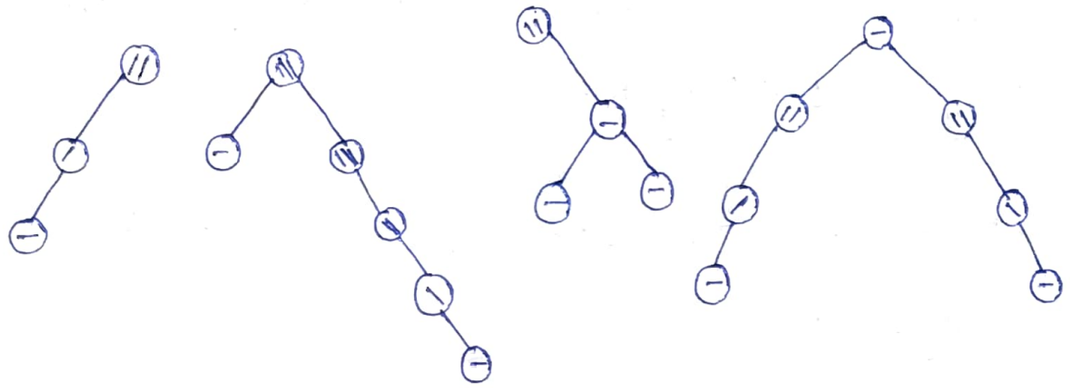
With each node of an AVL tree is associated a balance factor that is left high (/), equal (–) & right high (\) according, respectively, as the left subtree has height greater than, equal to or less than that of the right subtree.

→ Eg.



AVL Trees.

Non-AVL Trees.



→ Properties of AVL Trees.

A binary tree is an AVL Tree, if it is a binary search tree & it is height balanced ($K \leq 1$).

→ Minimum / Maximum Number of Nodes in AVL Tree

$h \rightarrow$ height of AVL tree

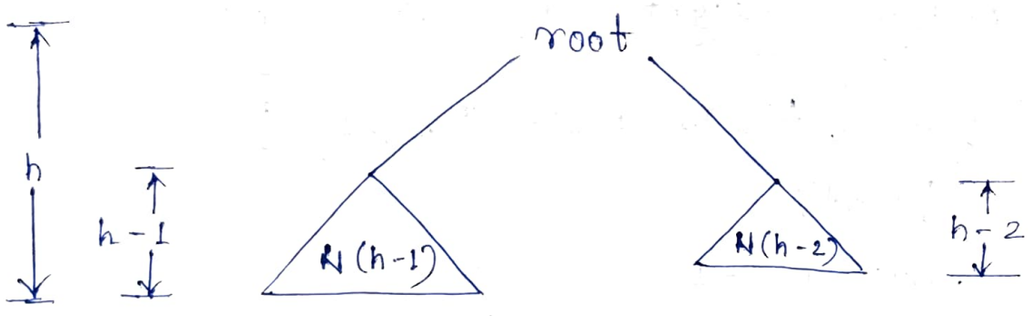
$N(h) \rightarrow$ number of nodes in AVL tree with height h.

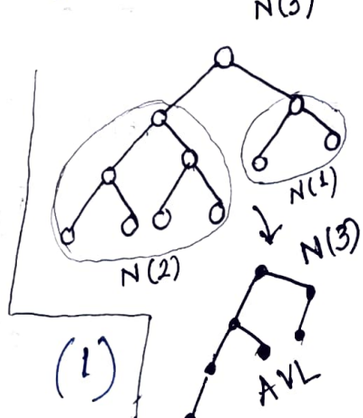
- Minimum number of nodes with height h is

$$N(h)_{min} = N(h-1) + N(h-2) + 1 \quad \text{---(1)}$$

$N(h) = 1, h = 0$

$= 2, h = 1$





By adding 1 to both sides of (1)

$$1 + N(h)_{min} = \frac{1}{\sqrt{5}} \left[\frac{1+\sqrt{5}}{2} \right]^{h+2}$$

Fibonacci number evaluation

$$\Rightarrow h \approx 1.44 \log N(h)_{min}$$

This means that the sparsest possible AVL tree with n nodes has height approximately $1.44 \log n$.

- Maximum number of nodes with height h –

$$N(h)_{max} = N(h-1) + N(h-1) + 1$$

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

Solving the recurrence relation, we get

$$N(h) = O(2^h) \Rightarrow h = \log n = O(\log n).$$

→ Node declaration.

```
Struct AVLTreeNode {
    Struct AVLTreeNode *left;
    int data;
    Struct AVLTreeNode *right;
    int height;
}
```

→ finding height of AVL Tree.

```
int Height (AVLTreeNode * root) {
    if (!root)
        return -1;
    else
        return root → height;
}
```

TC - O(1)

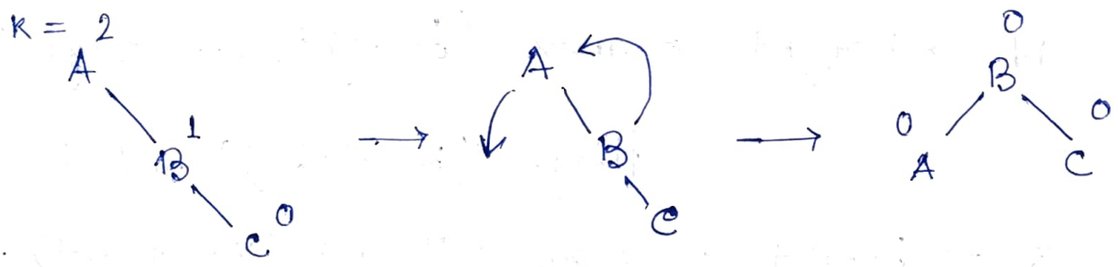
→ AVL Tree Rotations

We use rotation operations to make the tree balanced whenever the tree is becoming imbalanced due to insertion/deletion.

- Rotation is the process of moving the nodes to either left or right to make tree balanced.

- Rotations
  - Single rotations
    - left rotation (LL)
    - right rotation (RR)
  - Double rotations
    - left-right rotation (LR)
    - right-left rotation (RL)

- Single Left Rotation.



In LL rotation every node moves one position to left from the current position

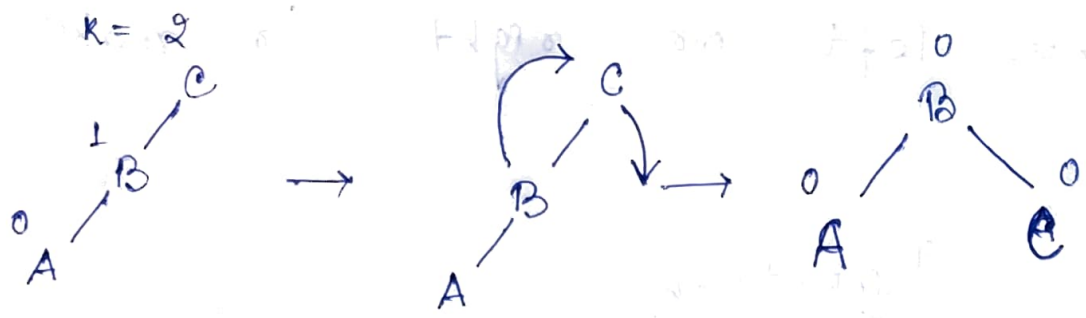
Code

```
Struct AVLTreeNode * LLRotate ( Struct AVLTreeNode *X)
{
    Struct AVLTreeNode *W = X → left ;
    X → left = W → right ;
    W → right = X ;
    X → height = max (Height (X → left), Height (X → right))
                 + 1 ;
```

```
W -> height = max (Height (W -> left), X -> height)
                                                + 1 ;
return W ;
}
```

TC - O(1)
SC - O(1).

- Single Right Rotation.



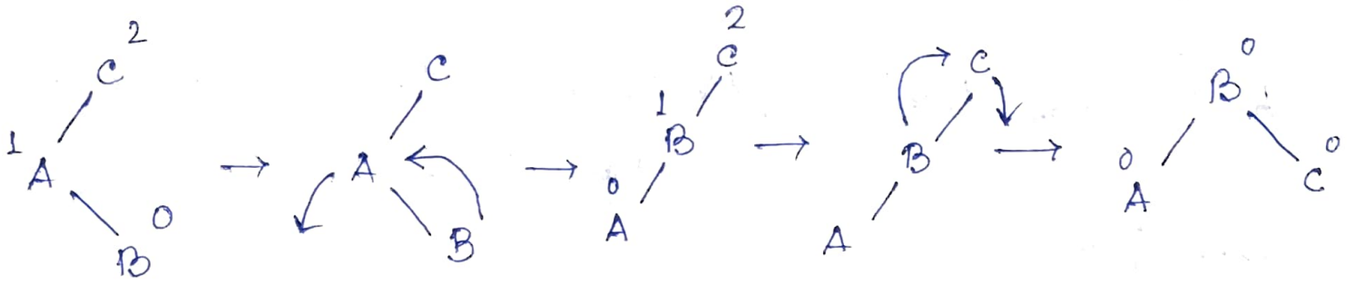
In RR rotation every node moves one position to right from current position.

Code

```
AVLTreeNode * RRRotate ( AVLTreeNode *W) {
    AVLTreeNode * X = W -> right ;
    W -> right = X -> left ;
    X -> left = W ;
    W -> height = max (Height (W -> right),
                       Height (W -> left)) + 1 ;
    X -> height = max (Height (X -> right),
                       W -> height ) + 1 ;
    return X ;
}
```

TC - O(1)
SC - O(1).

- Left - Right Rotation.

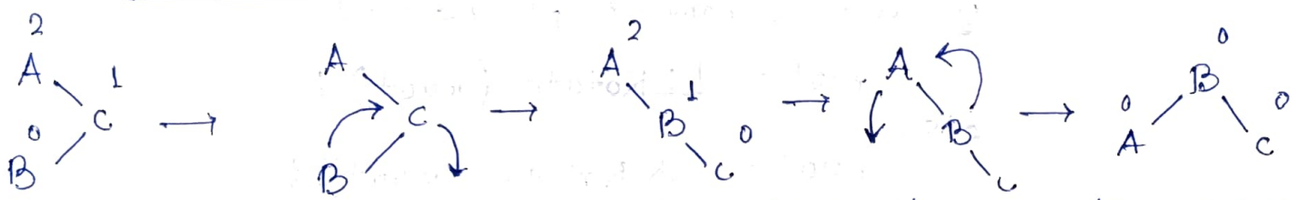


The LR rotation is a combination of single left rotation followed by single right rotation. First, every node moves one position to left then one position to right from the current position.

Code

```
AVLTreeNode * LRRotate ( AVLTreeNode *Z) {
        Z → left = RRRotate ( Z → left);
        return LLRotate ( Z );
}
```

- Right - Left Rotation.



The RL rotation is a combination of single right rotation followed by single left rotation. First, every node moves one position to right then one position to left from the current position.

Code

```
AVLTreeNode * RLRotate ( AVLTreeNode *Z) {
        Z → right = LLRotate ( Z → right);
        return RRRotate (Z);
}
```

• Insertion.

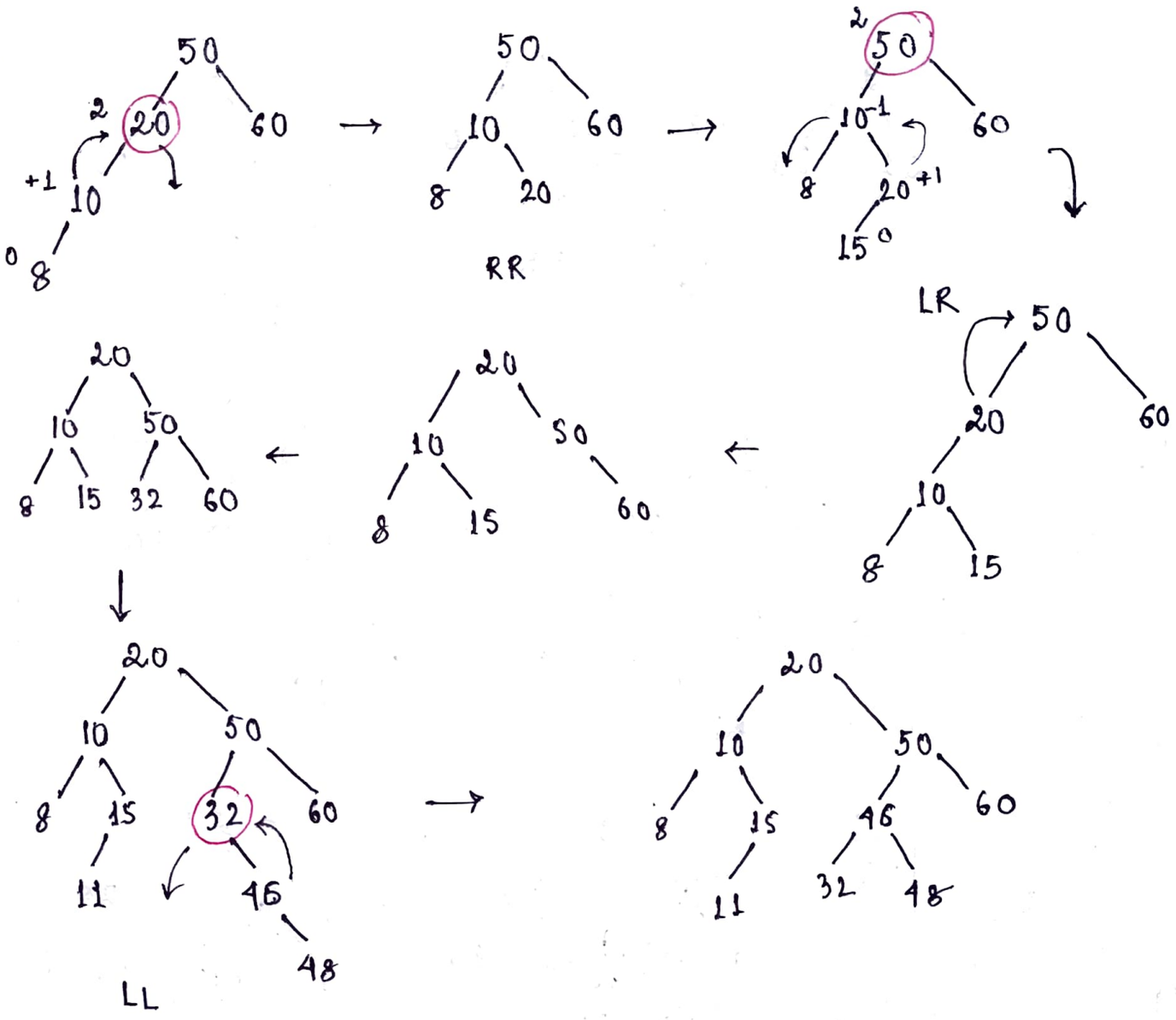
Code

```
AVLTreeNode *Insert ( AVLTreeNode *root , AVLTreeNode
                                *parent , int data) {
  if ( !root) {
      root = (AVLTreeNode *) malloc (sizeof (AVLTreeNode)
      if (!root) {
          printf ("Memory error");
          return NULL;
      }
      else {
          root → data = data ;
          root → height = 0 ;
          root → left = root → right = NULL ;
      }
  }
  else if ( data < root → data) {
      root → left = Insert ( root → left , root, data) ;
      if ((Height (root → left) - Height (root → right)) == 2 ) {
          if (data < root → left → data)
              root = LLRotate ( root) ;
          else
              root = LR Rotate (root) ;
      }
  }
  else if (data > root → data) {
      root → right = Insert (root → right , root , data) ;
      if ((Height (root → right) - Height (root → left)) == 2) {
          if ( data < root → right → data)
              root = RR Rotate (root) ;
          else
              root = RL Rotate (root) ;
      }
  }
  root → height = max (Height (root → left) , Height (root →
                                  right) + 1 ;
  return root ;
}
```

SC - O(logn)
TC - O(n)

• AVL Tree construction

1. 50, 20, 60, 10, 8, 15, 32, 46, 11, 48



• AVL tree time complexities : Height of tree, $h = O(\lg n$

Search = $O(\log h)$

Insertion = $O(\log n) + O(\log n) + c$

$O(\log n)$ → Searching to insert

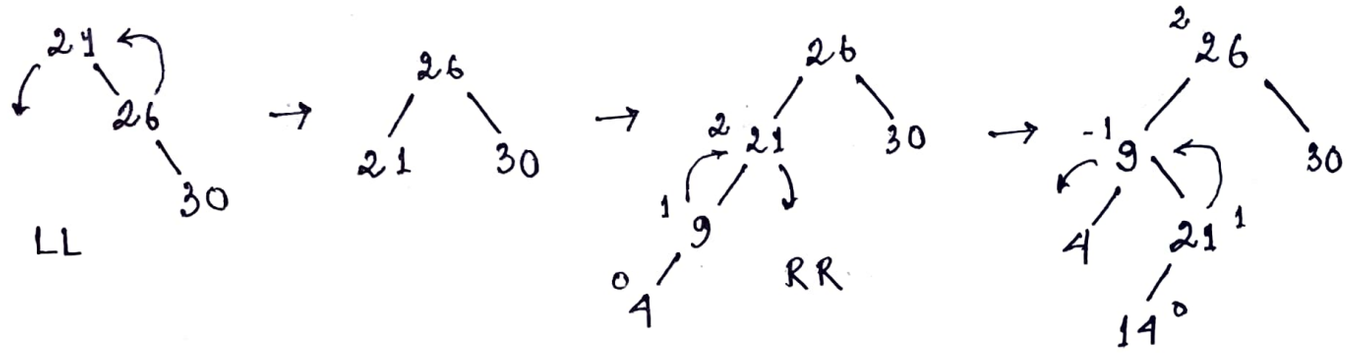
$O(\log n)$ → searching for imbalance

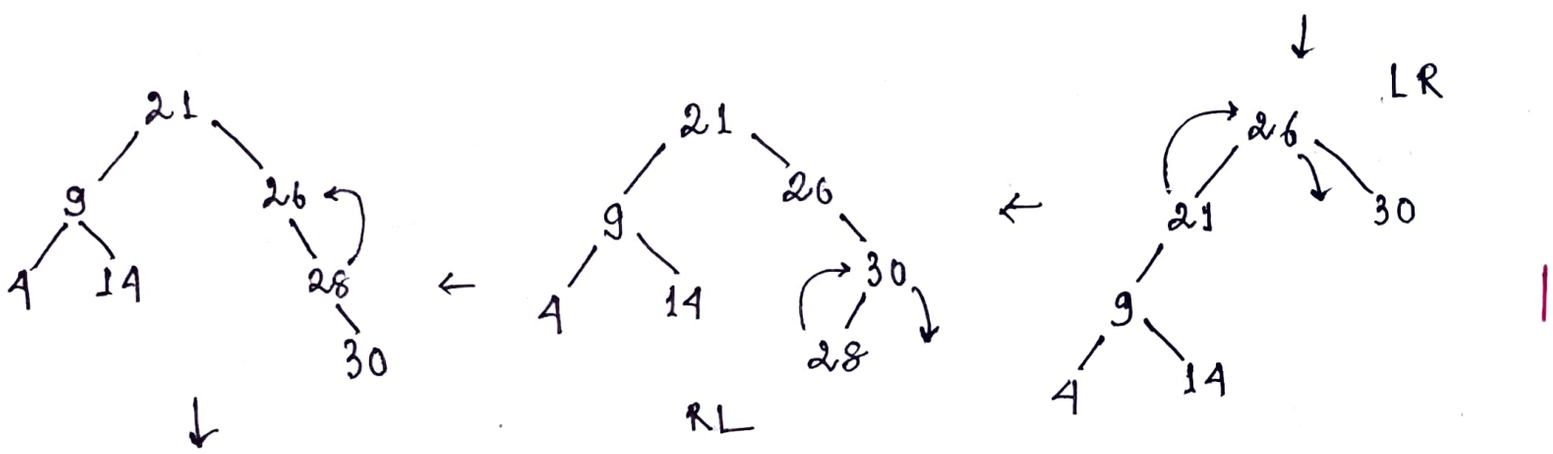
c → rotation

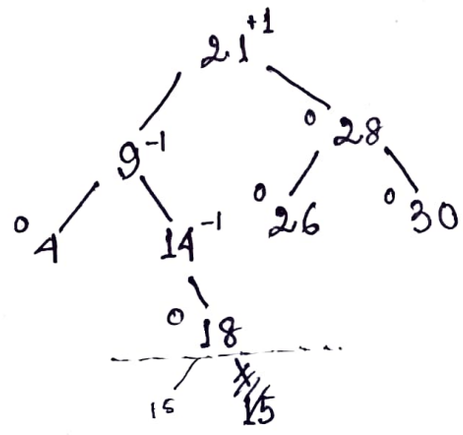
$= O(\log n)$

- AVL tree construction :

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

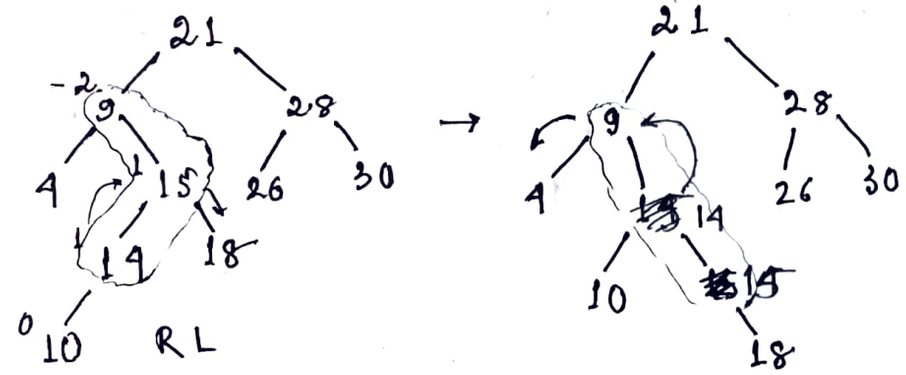


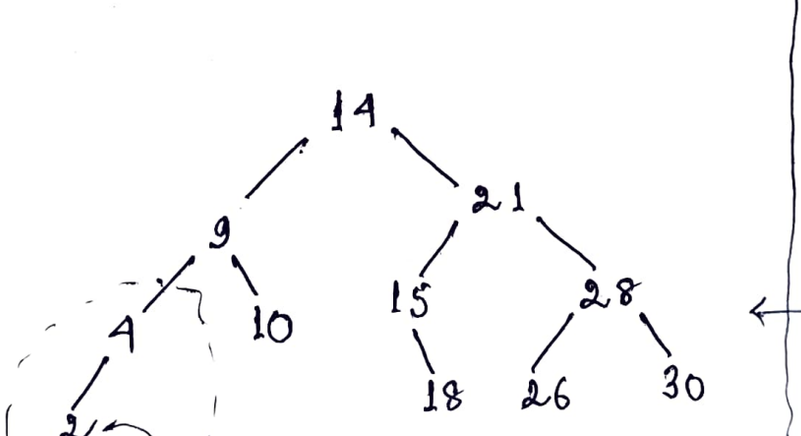




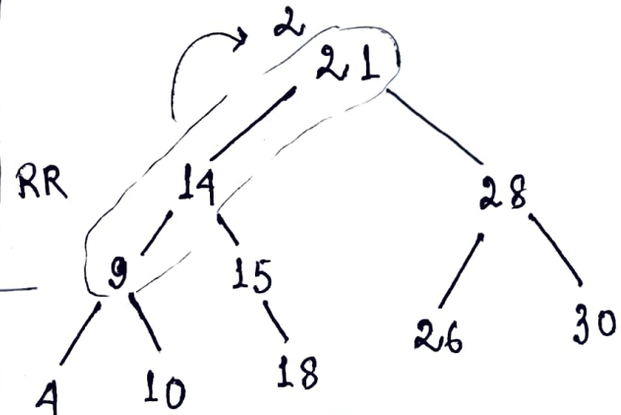
15
RL

Imp





RR



- AVL Tree deletion :

```
node* delete (node * root, int key) {
    //standard BST delete
    if (!root)  return root;
    if (key < root→key)
        root→left = delete (root→left, key);
    else if (key > root→key)
        root→right = delete (root→right, key);
    else {   // key == root→key , node to be deleted
        if (!root→left || !root→right) { // one or no child
            node* temp = root→left ? root→left : root→right
            if (!temp)  // no child {
                temp = root;
                root = NULL;
            }
            else   // one child
                *root = *temp   // copy content of child
            free (temp);
        }
        else {   // two children
            node * temp = minValueNode (root→right);
                                    // inorder successor
            root→key = temp→key;
            root→right = delete (root→right, temp→key);
                                    //delete inorder successor
        }
    }
    if (!root→left && !root→right) return root;
                        // if only one node in tree.
```

```
root → height = 1 + max (height(root → left),
                         height(root → right));
                    //update height of current node
int balance = getBalance(root); // get balance factor
                                   to check imbalance

//if imbalance rotate
if (balance > 1 && getBalance(root→left) >= 0)  //LL
        return rightRotate(root);
if (balance > 1 && getBalance(root→left) < 0)  //LR {
    root → left = leftRotate(root→ left);
    return rightRotate(root);
}
if (balance < -1 && getBalance(root → right) <= 0) //R
        return leftRotate(root);
if (balance < -1 && getBalance(root → right) > 0) //RL {
    root → right = rightRotate(root → right);
    return leftRotate(root);
}
return root;
}
```

Balance factor

```
int getBalance (node* N) {
    if (!N) return 0;
    return height(N→left) - height(N→right);
}
```

*Comparison Tree. / Decision Tree. / Search Tree.

→ Obtained by tracing through the action of the algorithm.

→ A decision tree is a tree in which

i) Internal nodes represent actions.

ii) Arcs represent outcomes of an action.

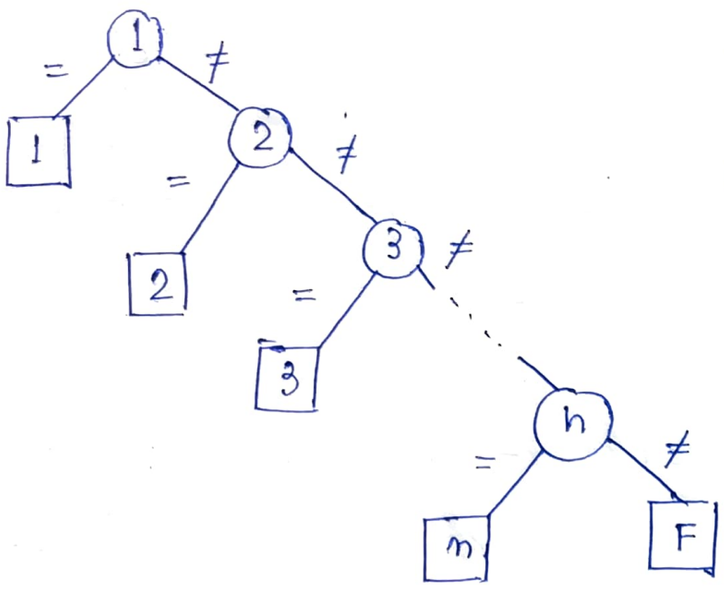
iii) Leaves represent final outcomes.

→ Internal nodes (vertex) represent comparison of keys & we draw them as a circle & inside that we put the index of the key against which we are comparing the target key.

→ When the algorithm terminates, we put either F (for failure) or the location where the target is found at the end of the appropriate branch, which we call a leaf (end/external vertices) & draw as a square.

→ The number of comparisons done by an algorithm on a particular search is the number of internal vertices traversed in going from the top of the tree (root) down the appropriate path to leaf.

→ Comparison tree for Sequential Search :

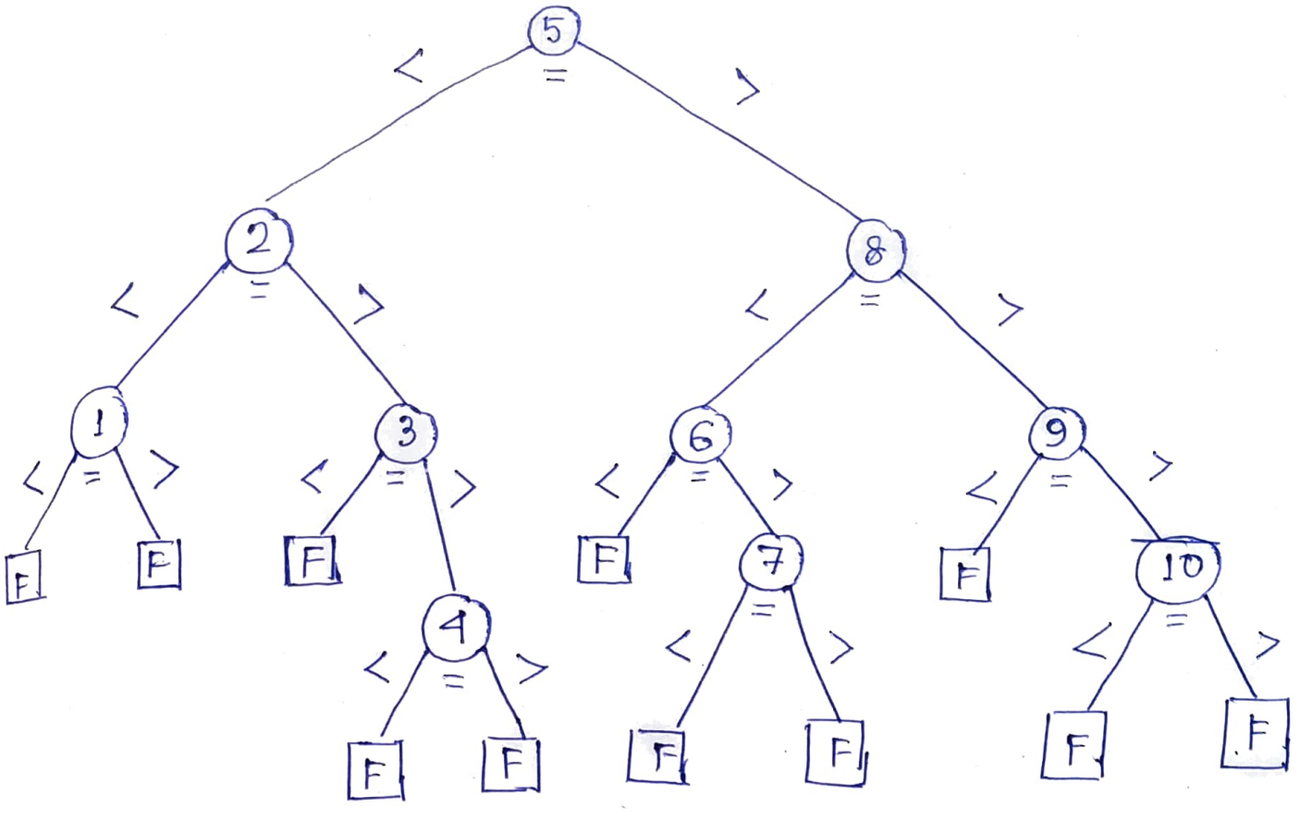


→ Comparison tree for Binary Search

```
int BinS ( List_type list, key_type target) {
    int top, bottom, middle;
    top = list. count - 1;
    bottom = 0;
    while (top ≥ bottom) {
        middle=(top + bottom) /2 ;
        if (EQ (list. entry [middle] . key , target)
            ~~bottom = middle + 1;~~
            return middle;
        else if ( LT( list . entry [middle]. key, targe
            bottom = middle + 1;
        else top = middle - 1;
    }
    return -1 ;
}
```

}

For $n = 10$.



---

* Huffman Coding Tree. (Huffman Encoding)
* Radix tree
* Tries.
* Treaps (Tree + Heaps).

Red – black Tree.

→ **Defination** – Red-black tree is a self-balancing BST where every node follows the following rules —

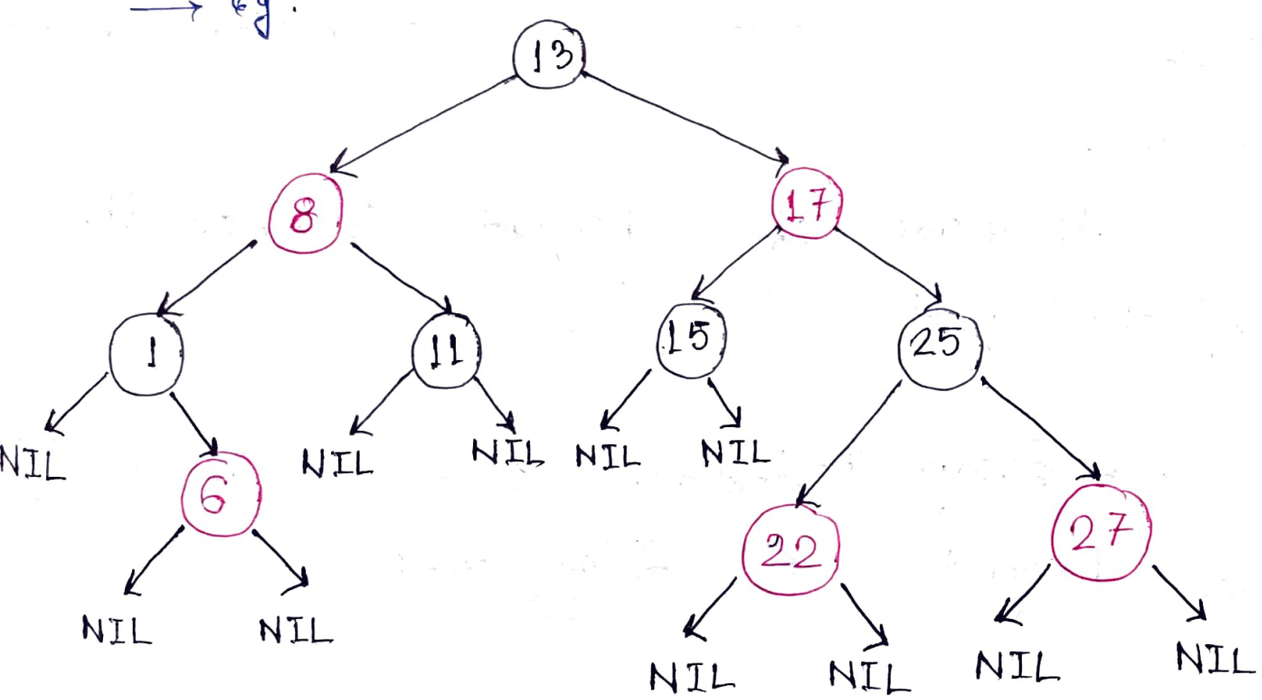
i) Each node is either red or black (In Red-black tree, nodes are associated with extra attribute : the colour).

ii) Root of tree is always black.

iii) There are no two adjacent red nodes (A red node cannot have a red parent or red child.)

iv) Every path from a node (excluding root) to any of its descendant NULL node has the same number of black nodes.

→ Eg.



→ Height of a red-black tree is always $O(\log n)$ which ensures the search, max, min, insertion, deletion operations take $O(\log n)$ time.

→ Comparison with AVL Tree.

The AVL Trees are more balanced compared to red-black trees, but they may cause more rotations during insertion and deletion. Red-black trees are preferred when there is frequent insertions & deletions. Otherwise, where search is a more frequent operation, AVL Tree should be preferred.

→ Black Height of a Red-black Tree.

Black height is number of black nodes ~~one~~ on a path from a node to leaf. Leaf nodes are also counted black nodes.

A node of height h has the black-height $\geq h/2$.

→ Every Red-black tree with n nodes has height $\leq 2 \log_2 (n+1)$

proof

i) For a general binary tree, let k be the minimum number of nodes on all root to NULL paths, then $n \geq 2^k - 1$ or

$k \leq 2\log_2(n+1)$.

ii) In a Red-black tree with n nodes, there is a root to leaf path with at most $\log_2(n+1)$ black nodes. (P.4)

iii) Number of black nodes in a Red-black tree is at least $\lfloor h/2 \rfloor$.

Hence, proved.

→ A tree with all nodes black can be a Red-black tree. The tree has to be a perfect binary tree & so it is the only tree whose Black-height equals to its tree height.

* Splay Trees.

→ A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again.

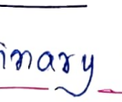
→ It performs basic operations such as insertion, lookup, removal in $O(\log n)$ amortized time.

→ Starting with empty tree, any sequence of K operations with maximum of n nodes takes $O(K \log n)$ time complexity in worst case.

→ Splaying. - Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree.

---

* General Points.

- Complete binary tree () $h = d+1$

  2^l nodes at level l ($l = 0$ at root)

  # nodes $2^{d+1} - 1$ (leaves 2^d, non-leaves $2^d - 1$

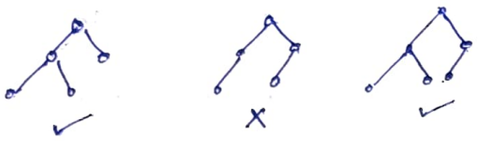
  $d = 0$ @ root

  If n leaves ⇒ $2n - 1$ nodes in total.

  $d = \log(N+1) - 1$

  $\boxed{T = 2I + 1 = 2L - 1}$

- Strictly binary tree: (non-leaf node has non-empty subtrees)
  0, or 2 children
  Every non-leaf has degree 2
  n leaves ⇒ 2n-1 nodes
  odd # of nodes
- Almost complete binary tree (Left-filled)
  All levels except the last one are completely filled & the last level is filled from left to right.



numbering

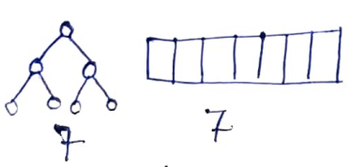
n nodes ⇒ leaf nodes have numbers $\lceil n/2+1 \rceil, \lceil n/2+2 \rceil, \ldots, n$

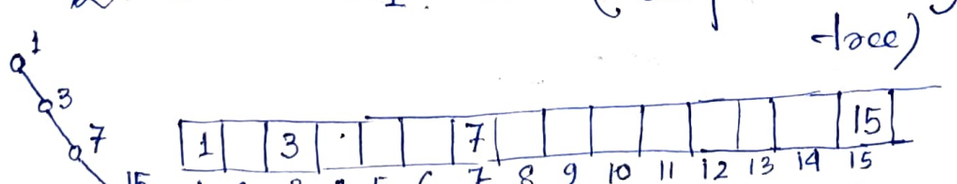
- In a tree with n nodes, there are n-1 edges.
- $n_0 = n_2 + 1$ # nodes with degree 0, 2 (n_0, n_2)
- In heap array (tree representation),

array rep" of heap			index starts @ 1
	parent @ location i		
⇒	children @	2i, 2i+1 (L, R)	
	parent @ location i		
⇒	children @	2i+1, 2i+2	@ ≠ 0

- In linked list rep", if there are n nodes then # of NULL links = n+1.

- Full binary tree every node (non-leaf) has 2 children, all leaves @ same level.
- Max size of array to store a binary tree with n nodes = $2^n - 1$. (Skewed binary tree).
  Min = n (Complete binary tree)





max. $2^4 - 1 = 15$

- Worst case time complexity :

		Search	Insert	Delete
Binary tree	skewed	$O(n)$	$O(n)$	$O(n)$
BST	skewed	$O(n)$	$O(n)$ in general $O(h)$	$O(n)$ $O(h)$
AVL tree		$O(\log n)$	$O(\log n)$	$O(\log n)$

- There's one & only path b/w every pair of vertices in a tree.
- Any connected graph with n vertices, $n-1$ edges is a tree.

→ Level : Each step from top to bottom. (root level = 0)

Height : # of edges that lies on the longest path from any leaf node to a particular node. Height of tree = height of root. Height of leaf = 0.

Depth : # of edges from root to a particalare node. Depth of tree is # of edges from root to a leaf node in the longest path. Depth of root = 0. Level ≈ Depth.

→ Forest is a set of disjoint trees.

- Min # nodes in a binary tree with height h is $h+1$. (Skewed)

Max # nodes in a binary tree of height h is $2^{h+1}-1$.

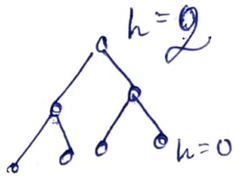
- # leaves = (# nodes with degree 2) + 1
- Max # nodes at level $l = 2^l$
- In a full/complete binary tree (0 or 2 children),

min # nodes = $2h + 1$ (by induction).

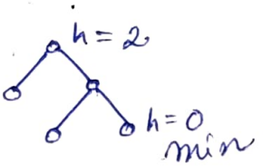
height h

max # nodes = $2^0 + 2^1 + \dots + 2^h$

$= \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1.$



max



min

- In a complete k-ary tree (or k children),

max # nodes = $\frac{k^{h+1} - 1}{k - 1}$.

min # nodes = $k.h + 1$.

- For a k-ary tree, $L = (k-1)I + 1$

$T = L + I$

$= (k-1)I + 1 + I$

$\begin{cases} T = kI + 1. \\ T = \frac{kL - 1}{k - 1}. \end{cases}$

* # trees possible with n nodes (not only binary trees) = $2^n - n$
* Complete n-ary tree (either 0 or n children), having k internal nodes,

leaves = $k(n-1) + 1$

Huffman Coding (Encoding technique)

Message: BCCABB DD AE CC BB AE DD CC

Character	count / frequency	code.
A	3	000
B	5	001
C	6	010
D	4	011
E	2	100
	20	

To represent 5 chars we need $\lceil \log_2 5 \rceil = 3$ bits

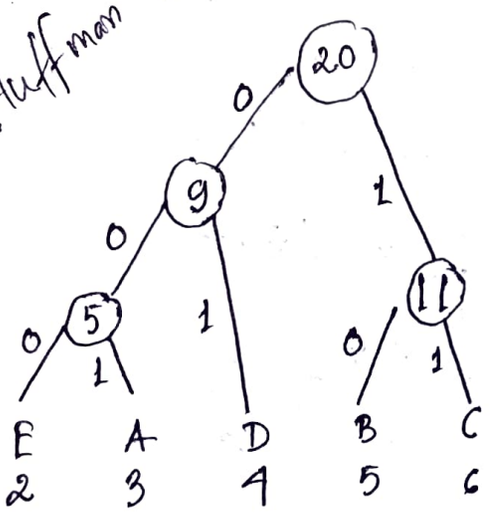
Send the table along with code

- Arrange acc. increasing order of count

E	A	D	B	C
2	3	4	5	6

A	001
B	10
C	11
D	01
E	000

Huffman tree



- Merge 2 smallest nodes.
- LHS 0 RHS 1
- Now encode by following path to char.

We can see, more frequent chars are given less no. of bits to represent themselves.

Previously total #bits needed for message =
20×8 bits (for each ASCII char 8 bits)
$= 160$ bits.

After huffman encoding,

message size + table size

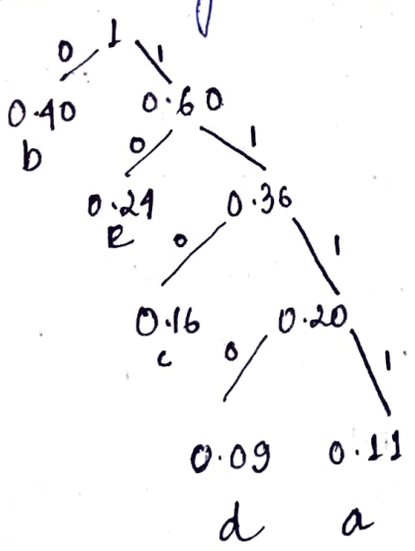
$= 3\times3 + 5\times2 + 6\times2$ (A, B, C)
$+ 4\times2 + 2\times3$ (D, E)
$= 15$ bits

$= 5\times8$ bits $+ (3+2+2+2+3)$ bits
$= 82$ bits

$\Rightarrow$ total $15 + 82 = 97$ bits needed instead of 160 bits.

Decoding is done using the table or the huffman coding tree.

Q. Chars a, b, c, d, e, each occurring with probability 0.11, 0.40, 0.16, 0.09, 0.24. Optimal huffman coding has avg. length of character ——.

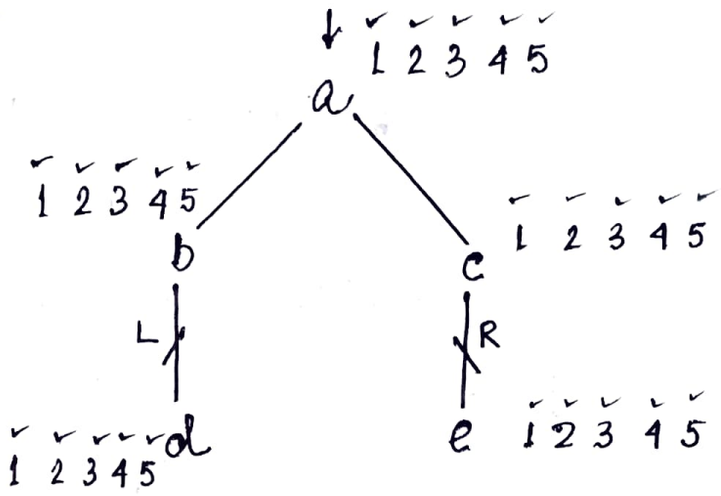


$\sum p_i \times (\#\text{bits})$
$= 0.11 \times 4 + \ldots + 0.24 \times 2$
$= 2.16$ (Ans)

We need 2.16 bits/symbol on average.

- Triple order traversal.

```
void TO ( struct node *t) {
  if (t) {
1.   pf ("%d", t→data);
2.   TO (t→LC);
3.   pf ("%d", t→data);
4.   TO (t→Rc);
5.   pf ("%d", t→data);
  }
}
```



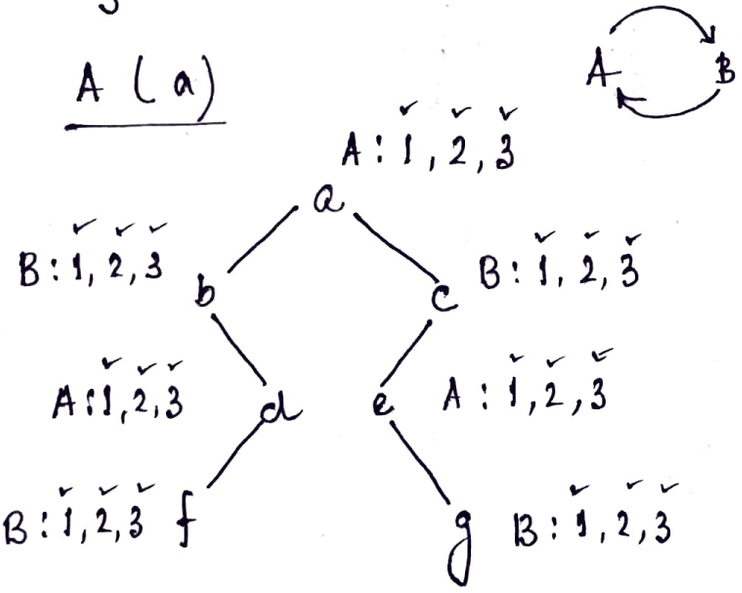
O/P: a b d d d b b a c c c e e e c a

- Indirect recursion on trees

```
void A (struct node *t) {
  if (t) {
1.   pf (t→data);
2.   B (t→left);
3.   B (t→right);
  }
}
```

```
void B ( struct node *t) {
  if (t) {
1.   pf (t→left);
2.   pf (t→data);
3.   A (t→right);
  }
}
```

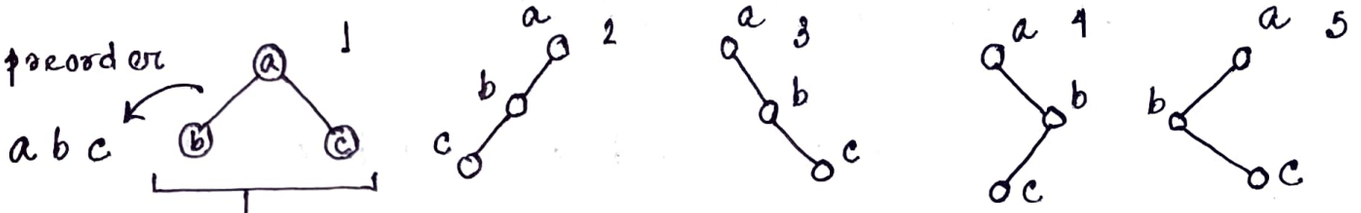
A (a)



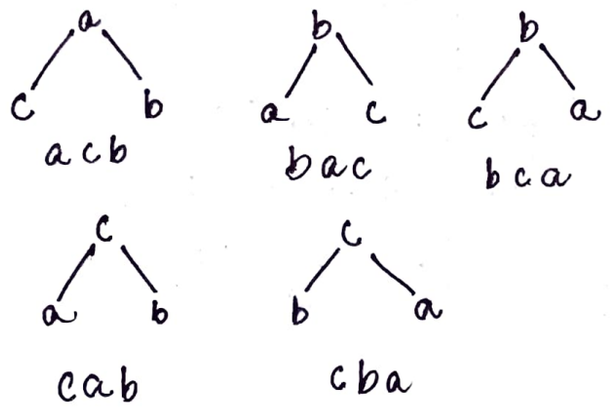
O/p: a b d f e g c

• Number of binary trees possible with 3 nodes a, b, c, that has preorder sequence as abc.

→ For each unlabeled version of the tree with 3 nodes we will have 3! combinations possible for labels. (Out of this 3! arrangements only 1 arrangement can have the preorder as abc.)



Ans : 5.



Same q^n with extra constraint: postorder is cba

	postorder
1.	bca
2.	cba
3.	cba
4.	cba
5.	cba

Ans : 4.

✓ # For a given preorder & #nodes = n, we can build $\frac{1}{n+1}\binom{2n}{n}$ trees (binary).

✓ # For a given preorder along with postorder we get more than/equal to one binary tree(s).

✓ # Given (inorder, preorder, postorder) or (inorder, preorder) or (inorder, postorder), we can build unique binary tree.

• Recursive program to count #nodes in binary tree.

→ #nodes (T) = 1 + #nodes (LST) + #nodes (RST)

= 0 , if T is NULL.

```
int NN ( struct node* T) {
  if (T) {
      return 1 + NN (T→left) + NN (T→right);
  }
  else
      return 0;
}
```

TC O(n)
SC O(n) o(h)

• Recursive program to count #leaves.

→ #leaves(T) = 1 , if T is a leaf

= #leaves (LST) + #leaves (RST)

```
int NL ( struct node* T) {
  if(T) {
    if (T→left && T→right)
        return 1;
    else
        return NL (T→left) + NL (T→right)
  }
}
```

```
int NL ( struct node* T) {
    if (T == NULL) return 0;
else if (T→left == NULL && T→right == NULL)
        return 1;
    else return NL (T→left) + NL (T→right);
}
```

TC O(n)
SC O(n) O(h)

✓ • Count # non-leaves.

Nonleaves(T) = 0 , if T is a leaf

= 1 + # nonleaves (LST) + # nonleaves (RST)
, otherwise

```
int nnl ( struct node* T) {
    if (T == NULL || (T → left == NULL && T → right == N
        return 0;
    else
        return 1 + nnl (T → left) + nnl (T → right
}
```

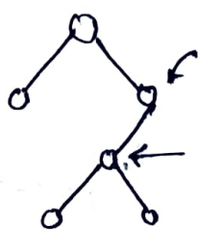
TC O(n)

SC O(n) O(h)

✓ • Count # full-nodes. (nodes that have both children)

FN (T) = 0 if T = NULL

= 0 if T = leaf



= FN (T → left) + FN (T → right)
if only one child

= 1 + FN (T → left) + FN (T → right)
if T is full node.

```
int FN ( node * T) {
    if (!T) return 0;
    else if ( !T→left && !T→right) return 0;
    else {
        return FN(T→left) + FN (T→right) +
            (T→left && T→right) ? 1 : 0 ;
    }
}
```

TC O(n)

SC O(n) O(h)

• Recursive program for height of binary tree.

→ $H(T) = 0$ T is empty or T is a leaf

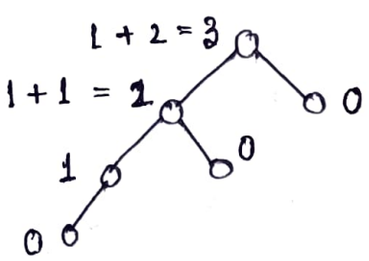
* $= 1 + \max(H(LST), H(RST))$; otherwise

```
int H(node * T) {
    if (!T) return 0;
    else if (!T->left && !T->right) return 0;
    else
        return (1 + (H(T->left) > H(T->right))?
                H(T->left) : H(T->right));
}
```

bad programming

use 2 variables l, r

(l > r) ? l : r



TC = O(n)

SC = O(n) O(h)

• Postorder for BST : 10, 9, 23, 22, 27, 25, 15, 50, 95, 60, 40, 29

G'05 What's the inorder ? → Sort & o/p. as in BST the inorder traversal gives the sorted sequence.

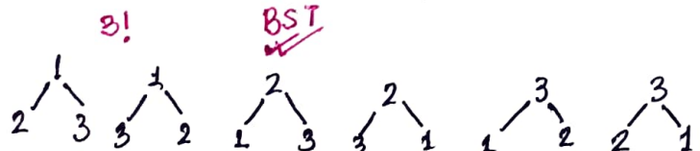
(Also, can write the preorder by building tree)

G'05 # of BSTs possible with 4 distinct keys.

For each structure (having n! arrangements), we can have only one BST.

$$\# = \frac{{}^{8}C_{4}}{5} = 14 \qquad \frac{1}{n+1}\binom{2n}{n}$$

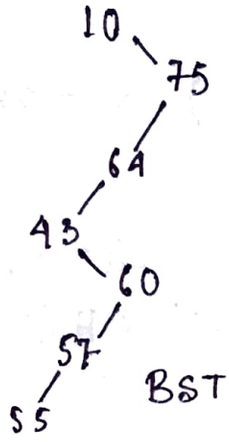
3! BST



G'06 IT (1-100) in BST. Search for 55. Which of the following sequences can't be the sequence of nodes examined?

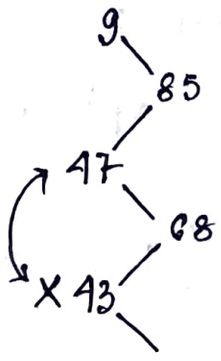
a) 10, 75, 64, 43, 60, 57, 55
b) 90, 12, 68, 34, 62, 45, 55
c) 9, 85, 47, 68, 43, 57, 55 ✓
d) 79, 14, 72, 56, 16, 53, 55

a.



Valid. BST

c.



× BST

G'08 IT Which is what (pre, in, post order)?

HBCAFHPYK | KAMCBYPFH | MABCKYFPH

post pre in

pre - root @ first
post - root @ last

G'98 In a complete n-ary tree if x is the #internal nodes of the tree, # leaves =

(0 or n)

1	internal node	:	n	leaves
2	n	nodes :	$(n-1)+n$	leaves
			$= 2n-1$	
3	n	n :	$(2n-2)+n$	leaves
			$= 3n-2$	
⋮				
x	n	n :	$xn-(x-1)$	
			$= x(n-1)+1$	

equating # edges
$xn = (x+L)-1$
$\Rightarrow L = x(n-1)+1$

G'02 #leaves in a rooted tree of n nodes, with each node having 0 or 3 children is:

#leaves = #internal (3-1) + 1

#leaves + #internal = n

$\Rightarrow$ #leaves = $\frac{2n+1}{3}$

$L = (K-1)I + 1$

$T = 2L - 1 \mid L = I + 1$

$= 2I + 1$

binary

- Recursive program to check whether a tree is complete binary tree.

```
int isComplete (node *t) {
    if (!t) return 1;          // null t
    if (!t->left && !t->right)) return 1;
                               // leaf
    else if (t->left && t->right) // both children
        return isComplete(t->left) && isComplete(t->right);
    else return 0;
}
```

G'95 A binary tree T has n leaf nodes. The # of nodes of degree 2 in T is: $n-1$.

$n_2 = ?$

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

$= N(n-1) + 1$

$= N(n-K) + K$

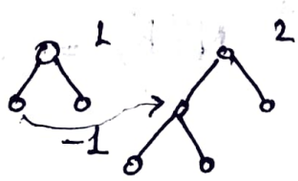
$N(1) = 2$

$N(1) = (2-1) + 2$

$= N(1) - 1 + 2$

N - #leaves

n - #internal nodes of deg 2



$n - K = 1 \Rightarrow K = n - 1$

$N(n) = n + 1$

$n_0 \quad n_2$

G'06 In a binary tree, # of internal nodes of degree 1 is 5 & # internal nodes of degree 2 is 10. # of leaves = 10 + 1 = 11.

G'98 Which is false?

a) A tree with n nodes has n-1 edges.

b) A labeled rooted binary tree can be uniquely constructed given its postorder & preorder.

c) A complete binary tree with n internal nodes has n+1 leaves.

d) Max. no. of nodes in a binary tree of height h is $2^{h+1} - 1$.

G'05 IT In a binary tree, for every node the difference b/w the #nodes in the left & right subtrees is at most 2. If height of tree $h > 0$, then the min #nodes in the tree is: (by elimin faster

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

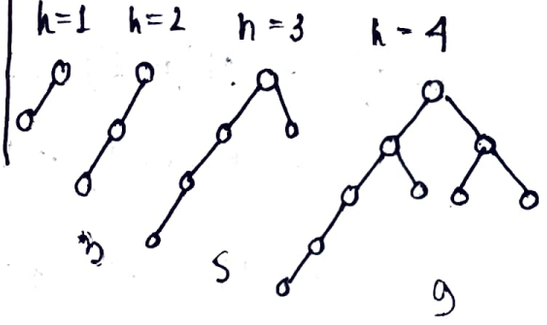
$$N(h) = 2N(h-1) - 1, \quad h > 2$$

$$= 3, \quad h = 2$$

By substitution,

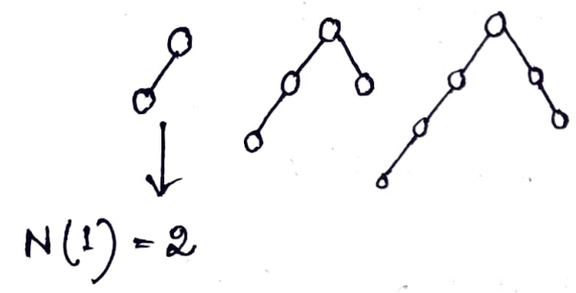
$$N(h) = 2^{h-2} N(2) - (2^{h-2} - 1)$$

$$= 2^{h-1} + 1.$$



Q'97 A size balanced binary tree is a binary tree in which for each node the diff b/w the # of nodes in the left & right subtree is at most 1. The distance of a node from the root is the length of the path from the root to the node. The height of a binary tree is the max dist. of a leaf node from the root.

Prove by induction on h, that a size-balanced binary tree of height h contains at least 2^h nodes.



$$N(h) = 1 + N(h-1) + (N(h-1)-1)$$
$$= 2N(h-1)$$
$$N(h) = 2^K N(h-K) \qquad h-K = 1$$
$$= 2^h$$

By induction, $N(h) = 2^h$

To prove, $N(h+1) = 2^{h+1}$

nodes for h+1 height =

$$2^h + 1 + 2^h - 1$$
$$= 2^{h+1}.$$

G'14 Suppose we have a balanced BST 'T' holding n numbers. We are given 2 numbers L & H & wish to sum up all the numbers in T that lie b/4 L & H. Suppose there're m such numbers. If the tightest upper bound on the time to compute the sum is $O(n^a \log^b n + m^c \log^d n)$, value of $a + 10b + 100c + 1000d = ?$ (GFG, GO).

```
int getSum (node* root) {
    if (!root) return 0;
    if (root → key < L)
        return getSum (root → right);
    if (root → key > H)
        return getSum (root → left);
    if (root → key >= L && root → key <= H)
        return getSum (root → left) + root → key +
                getSum (root → right);
}
```

Finding L, H takes $O(\log n)$
Traversing m elems b/w L & H $O(m)$

$\Rightarrow O(m + \log n)$

a = 0 | b = 1 | c = 1 | d = 0

$\Rightarrow$ expression = 110.

Traversal algorithm:

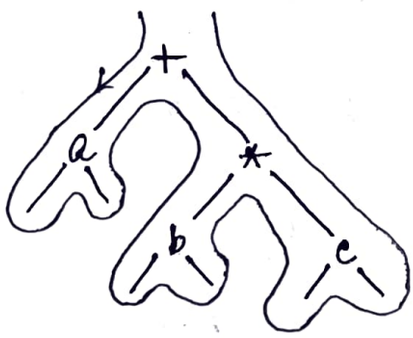
1. Find L using binary search & keep nodes (that have value b/w L & H) encountered in the search in a stack.

2. After finding L add it to a stack as well & initialise sum = 0.

3. Now, for all nodes in the stack, do an Inorder traversal starting from their right node & adding the node value to sum. Stop when H is found.

Inorder	2nd time seen
Preorder	1st time seen
Postorder	3rd time seen

eg.



Pre + a * b c

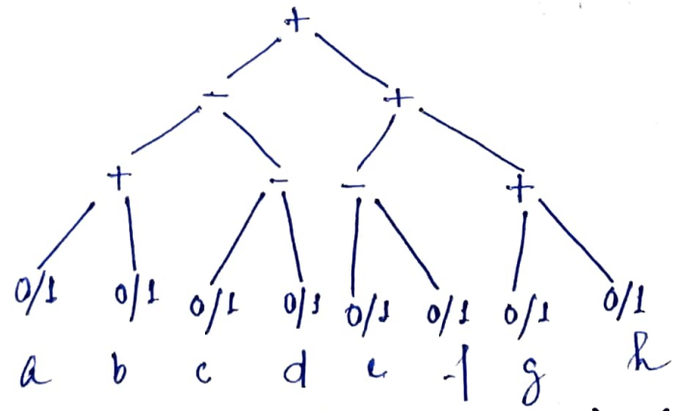
In a + b * c

Post a b c * +

G'14 Max value possible

$$\underset{1\ 1}{((a+b)} \overset{3}{-} \underset{0\ 1}{(c-d))} + \underset{1\ 0}{((e-f)} \overset{3}{+} \underset{1\ 1}{(g+h))}$$

= 6



max (a+b) − (c+d)

↓

max(a+b), min(c−d)

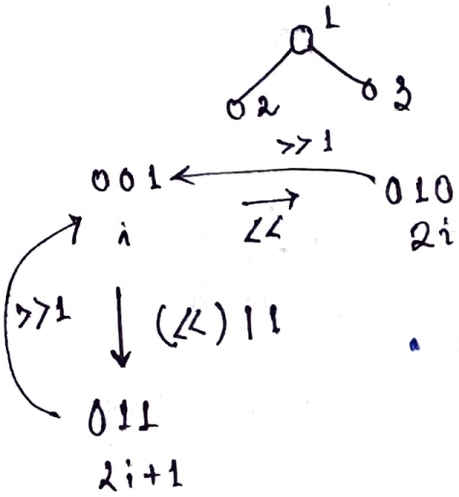
max (e−f) + (g+h)

max(e−f) max(g+h)

- Left child right sibling rep^n:

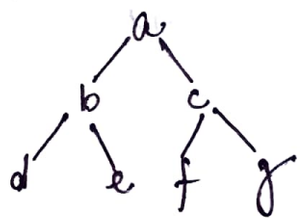
useful for $n>2$ an trees

- Array rep^n of tree.



parent i	root @ array pos^n 1
left child	2i (<< i)
right child	2i+1 ((<< i) | 1)
finding parent : >>	right shift

- Nested child rep^n.



a(bde)(cfg).

Q'00 Which represents valid binary tree of (x y z) is (root L R) & y, z can be null ?

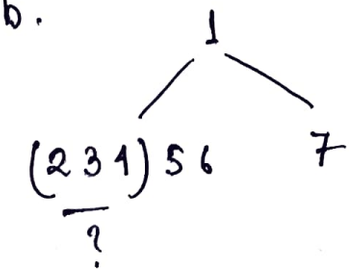
a) (1 2 (4 5 6 7))

b) (1 ((234)56) 7)

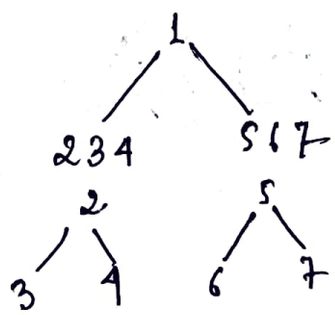
c) (1 (234) (5 6 7))

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

b.



c.

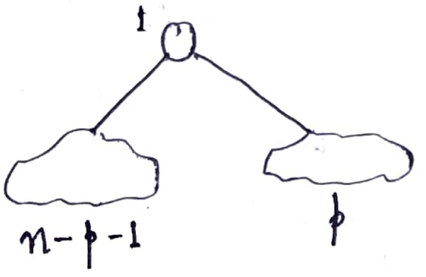


Q'05 IT

1, 2, ..., n are inserted into BST in some order. In the resulting tree, the right subtree of the root contains p nodes. First no. to be inserted in the tree must be:

a) p b) p+1 ✓c) n-p d) n-p+1.

→



1, 2, 3, ..., n-p-1, n-p, n-p+1, ..., n (Inorder)

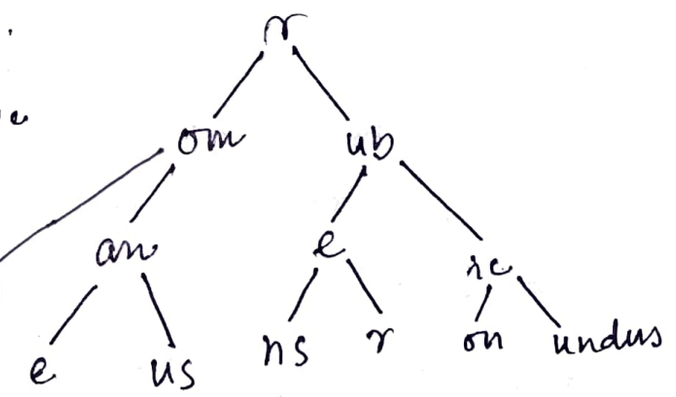
root ⇒ 1st to be inserted

(For balanced BST also, n-p will be root)

→ refer 39 Tree.*

• Radix tree.

(trie, where we combine nodes together if they ~~have~~ have only one child)



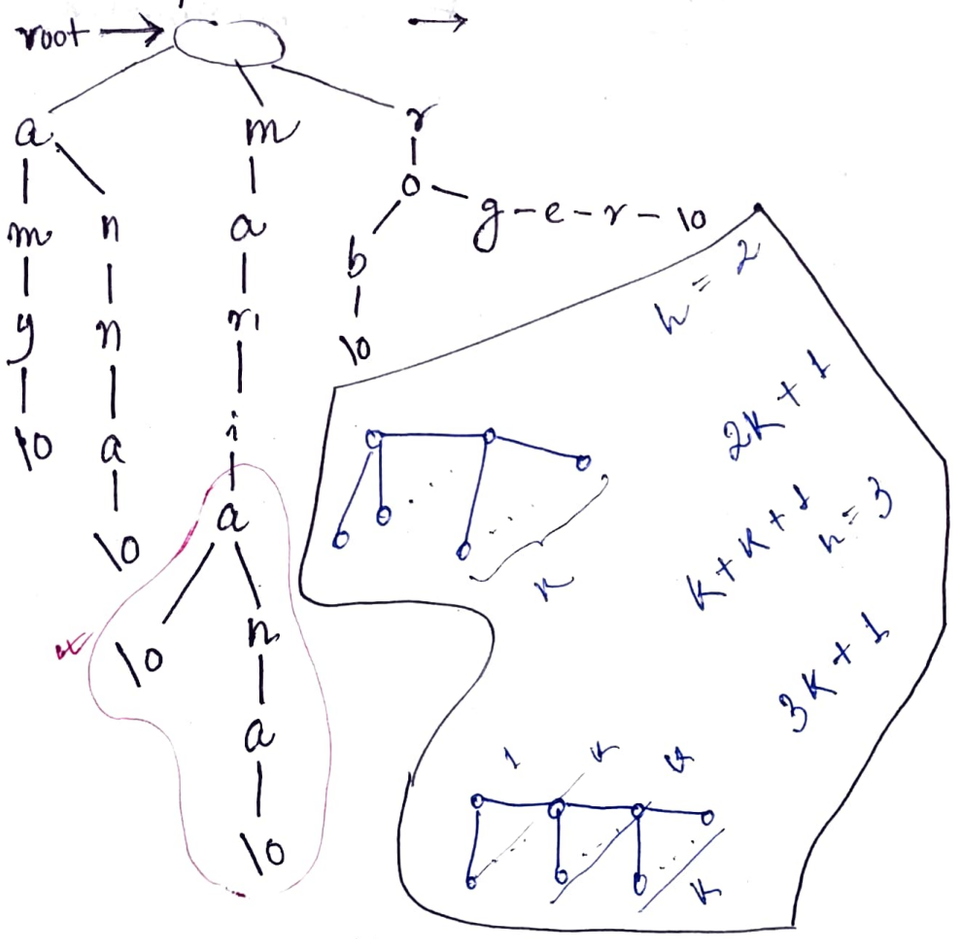
1. romane
2. romanus
3. romulus
4. rubens
5. ruber
6. rubicon
7. rubicundus

• Trie (from retrieval)

To compactly ~~store~~ strings.

(Space efficient, efficient prefix queries)

eg. amy, anna, maria, mariana, rob, roger



Radix trees are more cache-friendly than tries, since the chars within a single node are usually stored in an array of characters, adjacent to each other in a memory.

(refer NK)

Priority Queues and Heaps.

(More on heaps @ DAA)

* Priority Queue has following properties:

i) Every item has a priority associated.

ii) An element with higher priority is dequeued before an element with lower priority.

iii) If two elements have same priority, they are served according to their order in the queue.

* Main Priority Queue Operations.

i) Insert (item, priority)

ii) getHighestPriority ( )

iii) deleteHighestPriority ( )

* Priority Queue Applications.

i) data compression : Huffman Coding algo

ii) shortest path algo : Dijkstra's Algo

iii) minimum spanning tree algo : Prim's Algo

iv) Event-driven simulation : customers in a line

v) Selection problem : finding kth smallest element.

* Priority Queue Possible Implementations

i) Unordered array
ii) Unordered list
iii) Ordered array
iv) Ordered list
v) Binary Search Trees
vi) Balanced BST
vii) Binary Heap (Preferred)

Implementation	Insertion	Deletion (DeleteMin)	FindMax
Unordered array	1	n	n
Unordered list	1	n	n
Ordered array	n	1	1
Ordered list	n	1	1
BST	$\log n$ (avg)	$\log n$ (avg)	$\log n$ (av
Balanced BST	$\log n$	$\log n$	$\log n$
Binary heaps	$\log n$	$\log n$	1

* Heap:

Heap is a tree with properties:

i) Value of a node must be $\geq$ (or $\leq$) to the values of its children.

ii) Heap should form a complete binary tree (for binary heaps).

- Max No. of Nodes in heap = $2^{h+1}-1$ | $2^0+2^1+\cdots+2^h$
  Min No. of Nodes in heap = 2^h ✓ | $\frac{2^{h-1+1}}{\text{full } h-1} - 1 + 1$
- Height of a heap with n elements is $\log n$.
  $h \le \log n \le h+1$.
- Last element of heap is at $(\text{No. of nodes}) - 1$ position index in the array representing the heap. (arr. index starts @ 0)
- Position of parent of last node
  $$\frac{\text{No. of nodes} - 1}{2}$$
- For a complete binary tree of height h, sum of the heights of all nodes is $O(n-h)$, $n \rightarrow$ no. of nodes.

→ 2^{i-1} nodes on level i. (root at index 1)

Node on level i has height $(h+1-i)$

Sum of heights = $\sum_{i=1}^{h} 2^{i-1}(h+1-i) = S$

$S = h + 2(h-1) + 4(h-2) + \cdots + 2^{h-1}$

$2S = 2h + 4(h-1) + 8(h-2) + \cdots + 2^h$.

$2S - S = -h + 2 + 4 + \cdots + 2^h \Rightarrow S = (2^{h+1}-1) - (h-1)$

Now, $n = 2^{h+1} - 1$ Max.

$\Rightarrow h = \log(n+1) - 1$

$S = (n-h+1) = O(n-\log n) = O(n-h)$

↳ (root @ 0)

Sum = $\sum_{0}^{h} 2^i (h-i)$

$S = h + 2(h-1) + 4(h-2) + \cdots + 2^{h-1}(1)$

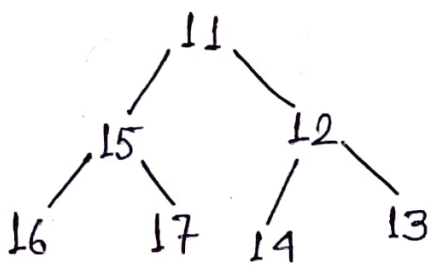
Same as before

level i ⇒ depth i ⇒ height $h-i$

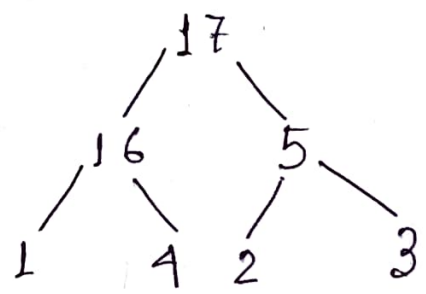
↳ #nodes = 2^i

* Types of Heaps

i) **Min heap** : Value of a node must be less than or equal to the values of its children.

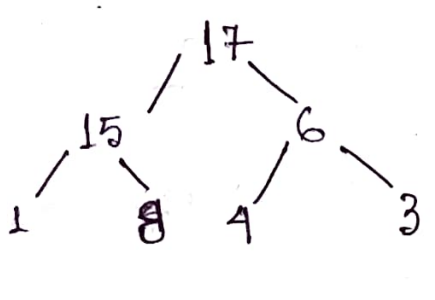


ii) **Max heap** :



* Binary Heaps.

→ **Representing Heaps.** & Using arrays



17	15	6	1	8	4	3

→ **Declaration of heap.**

```
struct Heap {
    int *array;
    int count;
    int capacity;
    int heap_type; };// Min or Max heap
```

→ **Heap Creation function.**

```
struct Heap *CreateHeap (int capacity, int heap_type)
{
    struct Heap *h = (struct Heap*)malloc(sizeof(struct Heap));
    if (h == NULL) {
        printf("memory error");
        return;
    }
```

```
    h→heap_type = heap_type;
    h→count = 0;
    h→capacity = capacity;
    h→array = (int*)malloc(sizeof(int)*
                           h→capacity);
    if (h→array == NULL) {
        printf("Memory error");
        return;
    }
    return h;
}
```

TC - O(1)

→ Parent of a Node. (root at 0 index)

For a node at ith location, its parent is at $\lfloor \frac{i-1}{2} \rfloor$ location.

C

```
int Parent(struct Heap *h, int i) {
    if (i <= 0 || i >= h→count)
        return -1;
    return (i-1)/2;
}
```

TC - O(1)

→ Children of a Node. (root at 0 index)

For a node at ith location, its children are at 2i+1 & 2i+2 locations.

C

```
int leftChild(Heap *h, int i)
{
    int left = 2*i + 1;
    if (left >= h→count)
        return -1;
    return left;
}
```

TC - O(1)

C

```
int rightChild(Heap *h, int i
{
    int right = 2*i + 2;
    if (right >= h→count)
        return -1;
    return right;
}
```

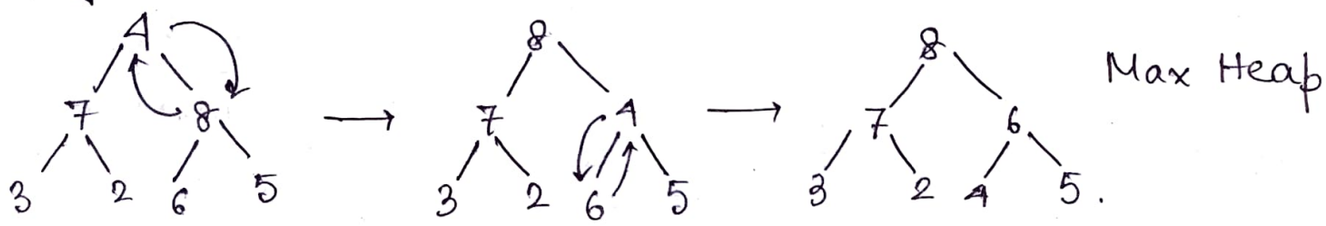
TC - O(1).

→ Getting the Maximum Element.

C
```
int GetMaximum (Heap *h) {
    if (h→ count == 0)
        return -1;                    TC - O(1)
    return h→ array[0];
}
```

→ Heapifying an element

eg.

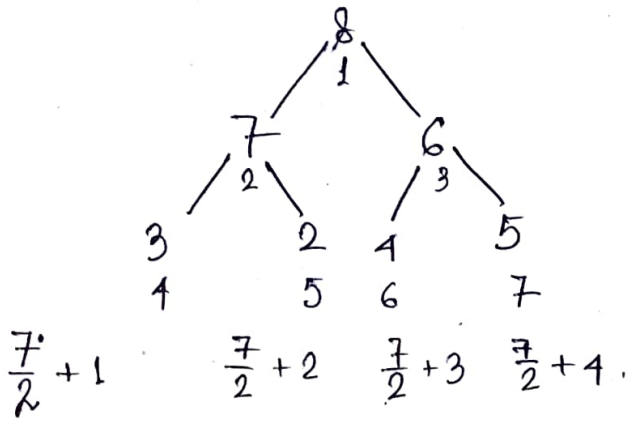



```
void max_heapify (Heap *h, int i) {
    int l, r, max, temp;
    l = leftChild (h, i);
    r = rightChild (h, i);
    if (l != -1 && h→ array[l] > h→ array[i])
        max = l;
    else
        max = i;
    if (r != -1 && h→ array[r] > h→ array[max])
        max = r;
    if (max != i) {
        // Swap h→ array[i] & h→ array[max]
        temp = h→ array[i];
        h→ array[i] = h→ array[max];
        h→ array[max] = temp;
    }
    max_heapify (h, max);
}
```

$T(n) \leq T(2n/3) + \theta(1)$

SC - O(1) | TC - O(log n)

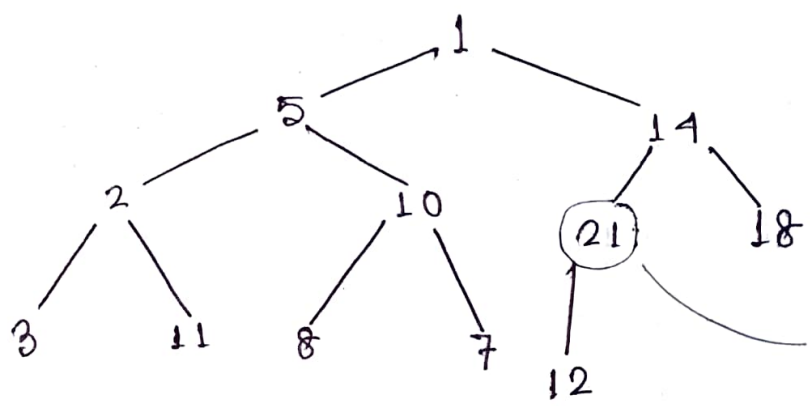
→ Property – A N element heap stored in an array has leaves indexed by $\lfloor N/2 \rfloor + 1, \lfloor N/2 \rfloor + 2, \ldots$ upto N.



N = 7

→ Building Max heap | Heapifying array

CLRS



last element of heap
(h → count - 1)

All leaf nodes always satisfy the heap property.

```
C void buildHeap (Heap *h, int A[], int n) {
    if (h == NULL)
        return;
    while (n > h→capacity)
        resizeHeap (h); // code in insert f^n
    for (int i = 0; i < n; i++)
        h→array[i] = A[i];
    h→count = n;
    for (int i = (n-1)/2; i >= 0; i--)
        maxHeapify (h, i);
}
```

TC O(n) CLRS
Applied (DSA/19/1)

→ Deleting element

After deleting root, copy last element of the heap & delete last element. Then heapify.

[C]
```
int deleteMax (Heap *h) {
    int data;
    if (h→ count == 0)
        return -1;
    data = h→ array [0] ;
    h→ array [0] = h→ array [h→ count - 1] ;
    h→ count -- ;
    maxHeapify (h, 0) ;
    return data;
}
```

TC → O(log n)

→ Inserting element

→ increase heap size
→ keep new element at the end of heap
→ heapify the element from bottom to top.

[C]
```
int insert (Heap *h, int data) {
    int i ;
    if (h → count == h → capacity)
        ResizeHeap (h) ;
    h → count ++ ;
    i = h → count - 1 ;
    while (i >= 0 && data > h→ array [ (i-1) / 2] ) {
        h → array [i] = h → array [ (i-1)/2 ] ;
        i = (i - 1) / 2 ;
    }
    h → array [i] = data ;
}
```

```
void ResizeHeap (Heap *h) {
    int * array_old = h → array ;
    h → array = (int *) malloc (sizeof (int) * h → capacity
                                * 2)
    ~~if (h.~~
    for (int i = 0 ; i < h → capacity ; i++)
        h → array [i] = array_old [i] ;
    h → capacity *= 2 ;
    free (array_old) ;
}
```

TC - O(log n)

→ Destroying Heap.

```
void destroy (Heap *h) {
    if (h == NULL)
        return ;
    free (h → array) ;
    free (h) ;
    h = NULL ;
}
```

- Examples / Questions / Problems.
  → (DAA 2)
- Refer NK for example problems.

* **Stack using heap**

We would use a minheap & a variable to be the priority for the elems pushed into the stack. Whenever we push an elem, we reduce the variable value by 1 (therefore the elem comes up to the root as minheap is used). While popping, we just then delete the root & heapifying the remaining will put the least priority element at the root. (Take example)

```
Void Push (int elem){
    insert_minheap (c, elem);
    c--;
}
int Pop (){
    return delete_min ();
}
int Top (){
    return minHeap ();
}
int size (){
    return sizeHeap ();
}
int isEmpty (){
    return emptyHeap ();
}
```

c = known_value
= 0 (say)

// We can increase c in Pop instread of decrement in Push.

* Queue using heap.

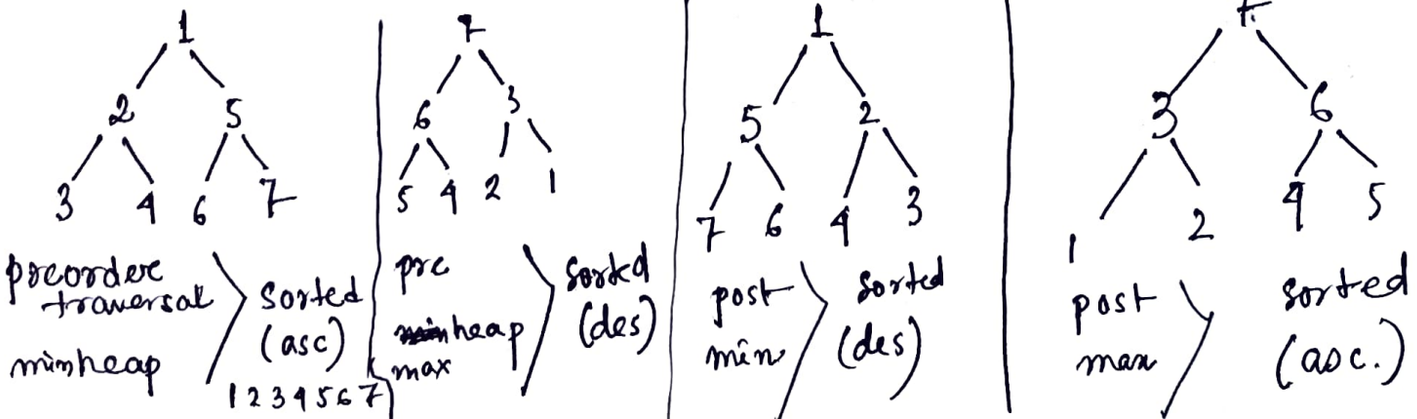
(We use min-heap, increase c by 1 while inserting in the queue).

```
void Push (int elem){
    insert (c, elem);
    c ++
}
int Pop (){
    return deleteMin();
}
int Top (){
    return minHeap();
}
```

.... Same as stack.

// We could also decrement c in Pop, instead of incrementing at push.

- There's no min/max heap with given elems, for which the inorder traversal gives the elems in sorted order (as the root is visited at second step, which is not the appropriate place as root contains the min or max.).
- There exists min/max heap with given elems for which the post/preorder traversal gives the elems in sorted order.



* Merge K sorted lists containing n elems in total.

→ Using minheap:

1. Create a min heap & insert the first element of all K arrays.

2. Run a loop until the size of minheap is greater than zero.

3. Remove the top elem. of the minheap & print the elem.

4. Now insert the next element from the same array in which the removed elem belonged.

5. If the array doesn't have any more elems, the replace root with infinite. After replacing the root, heapify the tree.

TC	$O(\frac{nK}{K} \log K)$	$O(n \log K)$	n - total elems across all lists
SC	$O(K)$	$O(n \log K)$	

K no. of lists each having n/K elems. So, each of $K \cdot n/K = n$ elems needs to be examined once. $\log K$ is the maximum # of bubble down operations for element (for inserting, deleting in the min-heap -