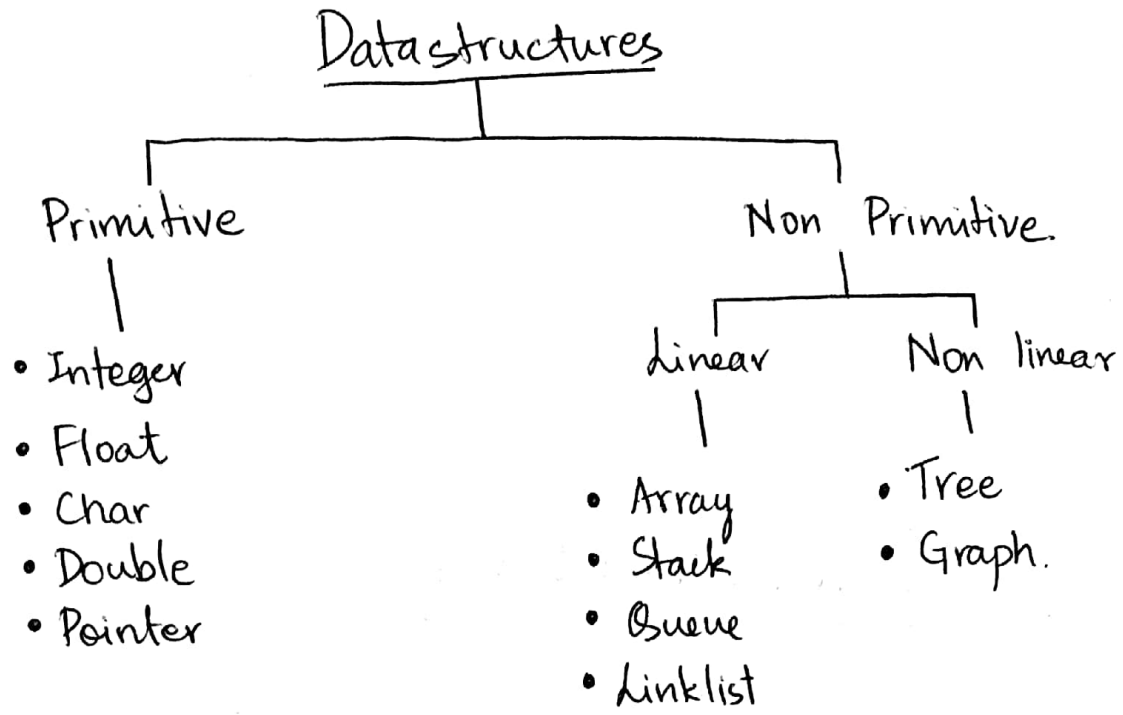


Recursion vs Iteration ★★★ (less code, slower / more code, fast)

Classification of data structures ★★★

Structure is a derived or constructed datatype which holds either similar or dissimilar data items.

The syntax of defining a structure is as follows

```
struct tagname
{
    datatype1  m1;
    datatype2  m2;          /*members*/
        :
    datatype n  mn;
};
```

Write a program to read and print a student info - name, sem, dept. using structure.

```
#include <stdio.h>

struct student
{
    char name[30];
    int  sem;
    char dept[15];
};

main()
{
    struct student s;
    printf("Read name:\n");
    scanf("%s", s.name);
```

```
    printf ("Read sem: \n");
    scanf ("%d", &s.sem);
    printf ("Read dept: \n");
    scanf ("%s", s.dept);
    printf (" %s \n %d \n %s", s.name, s.sem, s.dept);

    return 0;
}
```

WAP to read n student's info then print.

```
#include <stdio.h>

struct student
{
    char name[30];
    int sem
    char dept[15];
};

int main()
{   int n, i;
    struct student s[20];
    printf("Enter no. of students: \n");
    scanf ("%d", &n);

    for (i=0; i<n; i++)
    {
        pf ....
        sf .... s[i]. ...
    }
    for (i=0; i<n; i++)
    {
        pf
    }
```

WAP to read student info and to display it using a pointer

```
struct student
{
    char name[30];
    int sem;
    char dept[15];
};
int main()
{
    struct student s={"Pari", 3, "CSE"};
    struct student *ptr;
    ptr = &s;
    pf("Name %s\n sem %d\n Dept %s\n", ptr->name,
        ptr->sem, ptr->dept);
    return 0;
}
```

WAP n students using pointers.

```
struct student
{ ...
}
int main()
{
    int n, i;
    struct student s[20];
    struct student *ptr;
    read 'n';
    for (i=0; i<n; i++)
    {
        scanf("%s", s[i].name);
    }
```

```
for (ptr = &s ; ptr < &s[n] ; ptr++)
{
    print.....
}
```

WAP to display the content of array in reverse using ptr

```
int   s[5] , * = [1, 2, 3, 4, 5];
int   *ptr ;
   ptr = &s[4]
for ( ptr = &s[4] ; ptr > &s[0] ; ptr-- )
{
    printf(" %d\n", *ptr );
}
```

```
{
    printf("%d\n", *(ptr--));
}
```

STACKS

Stack is a linear non-primitive data structure where in the insertion and the deletion operation are done from the same end referred as top of the stack.

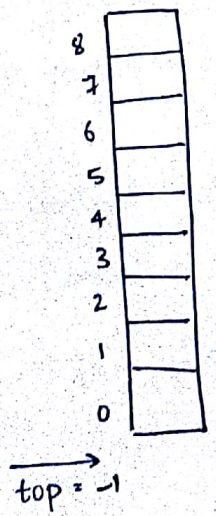
The insertion operation is referred as push and deletion operation is reffered as pop.

In a stack, the last element inserted will be the first one to be retreived out. Hence stack is also termed as LIFO

Implementation of a stack. (using an array)

Let 'SIZE' denotes the size of a stack which is implemented using an array data[SIZE].

Let 'top' denote the end used for 'push' and 'pop' operation with an initial value of -1 which is represented through the following diagram.



push operation

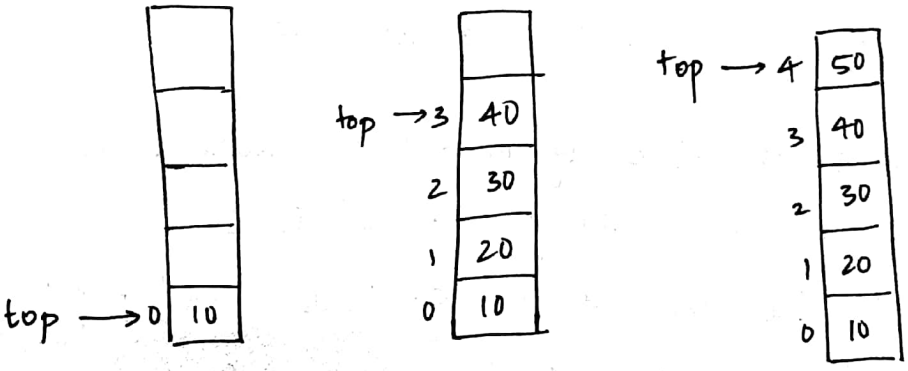
before inserting an element onto a stack, we need to check whether stack is full or not. The condition to denote stack full is

```
if (top == SIZE-1)
```

if stack is not empty, the push operation can be performed by the statement

```
data [++ top] = item;
```

If stack is full and if we try to perform push, it is called as overflow.



A C function to perform push operation.

```
void push (int ele)
{
    if (top == SIZE-1)
        pf ("Stack Overflow");
    else
        data (++top) = ele;
}
```

'pop' operation

The main objective in the 'pop' is to retrieve the top most element from the stack. Before retrieving, check whether stack is empty or not. The condition to check stack empty is

```
if (top == -1)
```

If stack is not empty, the popped element is

```
data[top--];
```

A c function to perform pop operation.

```
int pop()
{
    if (top == -1)
    { pf("Stack underflow\n");
      return -1; }
    else
        return data[top--];
}
```

'display' operation

```
void display()
{   int i;
    if (top == -1)
    { pf("Stack is empty\n");
    else
    { pf("Stack content is\n");
      for (i=top; i>0; i--)
      {
          printf("%d\n", data[i]);
      }
    }
}
```

C program to implement stack of integers

```
#include<stdio.h>
#include<stdlib.h>
#define SIZE 20
int top=-1;
int data[SIZE];
// push function
// pop function
// display function.
int main()
{
    int ch, ele, e;
    for(;;)
    {
        pf("1-push\n2-pop\n3-display\n4-terminate\n");
        pf("Read choice:\n");
        sf("%d", &ch);
        switch(ch)
        {
            case 1: pf("Enter element to be pushed\n");
                    scanf("%d", &ele);
                    push(ele);
                    break;
            case 2: e=pop()
                    if(e!=-1)
                    pf("popped element is %d", &e);
                    break;
            case 3: display.
                    break;
        }
    }return 0;
```

Program to implement stack using structures and pointers

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 5
struct stack
{
    int top;
    int data[SIZE];
};
typedef struct stack STACK;
void push(STACK *p, int ele)
{
    if (p->top == SIZE-1)
        pf("Stack Overflow\n");
    else
        p->data[++(p->top)] = ele;
}
int pop(STACK *s)
{
    if (s->top == -1)
    {   pf("Stack underflow\n");
        return -1;
    }
    else
        return s->data[(s->top)--];
}
void display(STACK s)
{
    int i;
    if (s.top == -1)
        pf("Stack is empty\n");
    else
    {
```

```
        pf("Stack contents are\n");
        for (i=s.top ; i>-1 ; i--)
        {
            pf("%d\n", s.data[i]);
        }
    }

int main()
{
    ... STACK s;
    s.top = -1;
    :
    push(&s, ele)
    :
    pop(&s)
    :
    display(s)
}
```

function to pop n elements.

```
void popn(STACK *s, int n)
{
    if (n > s->top+1)
    {   pf("error")
    else
    {
        pf ...
        for(i=0; i<n ; i++)
            pf("%d\n", pop(s));
    }
```

function to pop the bottom 3rd element.

```
void popb (STACK *s, int n)
{
    STACK s1; int n;
    s1.top = -1; int x;
    n = (s -> top + 1);
    if (n<3)
        pf("Invalid");
    else
    {
        for (i = s -> top; i > 2; i--)
            push (&s1, pop(s));
        x = pop (s);
        pf("Element popped is %d", x);
        push(s, x);
        while (s1.top != -1)
            push (s, pop (&s1));
    }
}
```

function to pop 3rd element from the top

```
void popt (STACK *s)
{
    STACK s1; int n;
    s1.top = -1; int y;
    ...
    else
    {
        push (&s1, pop(s));
        push (&s1, pop(s));
```

```
        y = pop(s);
    push(s,y);
    pf("Element popped is %d", y);
    while (s1.top != -1)
        push(s, pop(&s1));
  }
}
```

WAP to check if a given string is palindrome or not using stack.

```
struct stack
{
    int top;
    char data[SIZE];
};

typedef struct stack STACK;

void push(STACK *s, char ele)
{
    s->data[++(s->top)] = ele;
}

char pop(STACK *s)
{
    return s->data[(s->top)--];
}

int check
```

Expression

The meaningful collection of operands and operators is termed as an expression.

There are 3 types of expressions, mainly

1. Infix expression

   here, the operator is found in between 2 operands

   eg: -a+b

2. Postfix (suffix) expression.

   here, the operator is found at the end of after 2 operands. eg: ab+

3. Prefix expression

   .... before 2 operands.

   eg: +ab.

following rules need to be followed while converting an expression from one form to another form.

1. Always the expression should be evaluated from left to right

2. The operators with the higher preference should be evaluated first

   higher precedence → $ (expomential)

   medium → *, /

   lower → +, -

3, If the part of the expression is in (), should be evaluated first.

① a+b*c

a+ bc*

a bc* + (postfix)

a+ *bc

+a*bc (prefix)

② a+b-c

ab+ -c

ab+c- (postfix)

+ab-c

-+abc (prefix)

③ (a+b)*c

(ab+)*c

ab+c* (postfix)

+ab*c

*+abc (prefix)

④ (((a+(b-c)*d)$e)+f)

(((a+ ~~bc~~ T_1 *d)$e)+f)

T_1 = bc-

((a+ ~~T1~~ T_2) $e)+f)

T_2 = T_1 d*

((T_3 $e)+f)

T_3 = aT_2 +

(T_4+f)

T_4 = T_3 e$

T_4f +

T_3e $f+

aT_2+ e$f+

aT_1d* +e $f+

abc-d*+e$f+ (postfix)

(((a +T_1 *d)$e)+f)

+ $+a*-bcdef (prefix)

Note:-

the exponential operator is right associated.

abc - d/e

abc - d/e

abc$$ - d/e

abc $$ - de/

abc $$ de/-

Algorithm to convert the given infix exp to post fix.

step 1: scan the given infix exp from left to right.

step 2: if the scanned symbol is -

1) an operand - place it on the postfix exp.
2) if left paranthesses - push it onto stack.
3) if right paranthesses - pop the content of stack and place it on postfix exp until we get a corresponding left paranthesses
4) if operator - if stack is empty or top of stack is having left paranthesses, then push the operator onto stack.

else check the precedence of top of stack with the precedence of the scanned symbol,

if sym ≤ stack[top] → if it is greater or equal to, pop the stack content and place on postfix exp. and push the scanned symbol onto the stack

else,
push symbol onto stack

step 3 : until stack become empty, pop the stack content and place on postfix exp.

① a+b*c

ANS IS IMPORTANT

symbol	stack	postfix.
a	empty	a
+	+	a
b	+	ab
*	+,*	ab.
c	+,*	abc
	+	abc*
	empty	abc*+

② (a+b)*c

symbol	stack	postfix
(	(	empty.
a	(	a
+	(,+	a
b	(+	ab
)	~~(~~ empty.	ab+
*	~~(~~*	ab+
c	~~(~~*	ab+c
		ab+c*

(((a+(b-c)*d)$e)+f)

symbol	stack	postfix.
(	(((	empty
a	(((	a
+	(((+	a
(	(((+(	a.
b	((+(	ab
-	(((+(-	ab
c	(((+(-	abc
)	(((	abc-
*	(((*	abc-
d	(((*	abc-d
)	((	abc-d*+
$	(($	abc-d*+
e	(($	abc-d*+e
)	(	abc-d*+e$
+	(+	abc-d*+e$
f	(+	abc-d*+e$f
		abc-d*+e$f+

a*b-c

sym	stack	postfix
a	empty	a
*	*	a
b	*	ab
-	-	ab*
c	-	ab*c
		ab*c-

NOTE.
pop until condition is false. (comparing precedence).

~~a~~	+/*	abc
-	+-	abc*/

Write C func to convert a given infix exp to postfix.

```
void infix to postfix (STACK *s, char infix[15])
{   char symbol; int i,j=0; char postfix[15], temp
    for (i=0 ; infix[i]!='\0' ; i++)
    {
        symbol = infix[i];
        if (~~isalpha~~ isalnum (symbol))
            postfix[j++] = symbol;
        else
        {
            switch (symbol)
            {
                case '(': push(s, symbol);
                          break;
                case ')': temp = pop(s)
                          while (temp!='(')
                          {
```

```
                postfix[j++] = temp;
                temp = pop(s);
            }

        case '+':
        case '-':
        case '*':
        case '/':
        case '$' : if (s->top==-1||s->data[s->top]!=(

                    push(s,symbol);

                else
                {
                  while
                (preced(s->data[s->top] >= preced(symbo
                  && s->top == -1 &&
                   s->data[s->top] != '(')
                     postfix[j++] = pop(s);
                push(s,symbol);
                }

        default: pf("\n Invalid");
                 exit(0);
    }
}
while (s->top != -1)
    postfix[j++] = pop(s);
postfix[j] = '\0';
pf("\n The postfix expr is %s", postfix.
}
```

WAP to convert infix to postfix.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 20

struct stack
{
    int top;
    char data[15];
};
struct stack STACK;

// push func
// pop func

int preced(char a)
{
    switch(a)
    {
        case '$': return 5;

        case '*':
        case '/': return 3;

        case '+':
        case '-': return 1;
    }
}

// infix to postfix

int main()
{
    STACK s
    s.top = -1;
    char infix[20];
```

```
    pf ("Read infix\n");
    sf ("%s", infix);
    infix to postfix (infix);
}.
```

WAP to check for parantheses balancement using stack.

```
int check (char brack[20])
{
    int, i
    STACK *s
    s.top = -1;
    if (strlen(brack)/2 != 0)
    {
        return 0;
    }
    else
    {
        for (i=0 ; brack[i] != '\0'; i++)
        {
            if (brak[i] == '(' || brack[i] == '{' || brack[i] == '[')
                push (s, brack[i]);
            if (brack[i] == ')' || brack[i] == '}' || brack[i] == ']')
            { if (s->top == -1)
                return 0;
              else
              {
                if (brack[i] == s->data[s->top])
                    pop (s);
                else
                    return 0;
              }
        }
    }
```

```
    if (s->top == -1)
        return 1;
    return 0;
}
```

Evaluation of a postfix expression :-

ALGORITHM.

Step 1: Scan the given postfix expression from left to right.

Step 2: If the scanned symbol is :

i) operand : push it onto stack.

ii) operator : pop 2 elements from stack and assign them to operand 2 and operand 1 respectively and find the value operand 1 operator operand 2 and push the value onto the stack.

step 3: pop content of the stack to get final answer.

Trace the algo for the following inputs.

① 345*+

Symbol	Stack	Op1	Op2	value
3	3	-	-	-
4	3,4	-	-	-
5	3,4,5	-	-	-
*	3,20	4	5	20
+	23	3	20	23

final answer = 23.

(2) 632 - 5* + 2$3+

Symbol	Stack	op1	op2	value
6	6	-	-	-
3	6, 3	-	-	-
2	6, 3, 2	-	-	-
-	6, 1	3	2	1
5	6, 1, 5	-	-	-
*	6, 5	1	5	5
+	11	6	5	11
2	11, 2	-	-	-
$	121	11	2	121
3	121, 3	-	-	-
+	124 (pop)	121	3	124

final answer = 124.

A C function to evaluate postfix expression.

```
float evaluate_postfix (STACK *s, char postfix[20])
{
    int i,
    char symbol,
    float val, op1, op2, res
```

```
    for (i=0; postfix[i]!='\0'; i++)
    {
        symbol = postfix[i];
        if (isdigit(symbol))
        {   push(s, symbol - '0');
        else if (isalpha(symbo))
        {
            pf("Enter the value for character %c:" symbol);
            sf("%f", &val);
            push(s, val);
        }
        else
        {
            op2= pop(s);
            op1= pop(s);
        res= operate (symbol, op1, op2);
            push(s, res);
        }
    }
    return  pop(s);
}
float  operate (char  s, float op1, float op2)
{
     switch(s)
     {
         case '+': return op1+op2;
         case '-' : return op1-op2;
         case '*' : return op1* op2;
        case '/' : return  op1/ op2;
         case '$' : return  pow(op1,op2);
     }
}
```

convert the following postfix expression to infix.

① 345 * +

symbol	stack	intermediate exp.
3	3	-
4	3,4	-
5	3,4,5	-
*	3, (4*5)	(4*5)
+	3+4*5	3+4*5

② 632 -5 * + 2 $3 +

symbol	stack	iex.
6,3,2.	6,3,2	-
-	6, 3-2	3-2
5	6, 3-2, 5	-
*	6, (3-2)*5	(3-2)*5
+	6+(3-2)*5	6+(3-2)*5
2	6+(3-2)*5, 2	-
$	(6+(3-2)*5)$2	(6+(3-2)*5)$2
3	(6+(3-2)*5)$2, 3	
+	((6+(3-2)*5)$2)+3	

③ abc-+de-fg-h+/*

symbol	stack	i exp.
	a, b, c	-
-	a, (b-c)	b-c
+	(a+(b-c))	a+(b-c)
d, e	(a+(b-c)), d, e	-
-	(a+(b-c)), (d-e)	(d-e)
f, g	(a+(b-c)), (d-e), f, g	-
-	(a+(b-c)), (d-e), (f-g)	(f-g)
h	(a+(b-c)), (d-e), (f-g), h	-
+	(a+(b-c)), (d-e), [(f-g)+h]	~~
/	(a+(b-c)), (d-e)/[(f-g)+h]	~~
*	[a+(b-c)] * [(d-e)/[(f-g)+h]]	~~

Algo to convert infix to prefix

step 1. Scan the expr from right to left

step 2. If scanned symbol is

i) operand - place it of prefix expr.

ii) right parantheses - push it onto stack

iii) If it is left parantheses, pop the content of stack and place them on prefix until you get a corresponding right parantheses

iv) If operator, if stack is empty or top of stack is having right parantheses, push operator onto stack.

else, while stack isn't empty and top of stack is not right paranthesis and precedence of top of stack is greater than or equal to precedence of ~~stacked~~ scanned symbol, pop the stack content and place them on ~~post~~ pre fix and push the operator onto stack.

step3. until stack is empty, pop the stack content and place of prefix.

step4. reverse the prefix to get the actual result

(a+b)*c

symbol	stack	prefix.
c	-	c
*	*	c
)	*)	c
b	*)	cb
+	*)+	cb
a	*)+	cba
(	*	cba+
		cba+*

⇓

*+abc

convert the following prefix to infix.

⑤ - + abc

symbol	stack	intermediate exp.
c	c	-
b	c, b	-
a	c, b, a	-
+	c, (a+b)	(a+b)
-	(a+b)-c	(a+b)-c

⑥ + $ + a * - bcd ef

symbol	stack	intermediate
	f e d c b	-
-	f e d, (b-c)	b-c
*	f e, [(b-c)*d]	(b-c)*d.
a	f e , [(b-c)*d], a	-
+	f e, a+[(b-c)*d]	a+[(b-c)*d]
$	f, (a+((b-c)*d))$e	(a+((b-c)*d))$e
+	[(a+((b-c)*d))$e]+f	

Recursion

The process in which a function calls itself directly or indirecty is called recursion. and the corresponding function is called a recursive function.

direct recursion.

```
f()
{
   f()
}
```

indirect recursion

```
f()
{
   f1()
}
```

```
f1()
{
   f()
}
```

In case of a recursive function, all the recursive functions will be pushed onto stack., if there is no base case or base condition, the stack becomes overflow, so, to avoid this, every recursive function should have a base case.

A recursive function to find the factorial of a given number

$$fact(n)\begin{cases} 1 & : \text{if } n=1 \\ n * fact(n-1) & : \text{otherwise.} \end{cases}$$

```
int fact (int n)
{
    if (n == 1)
        return 1;
    else
        return n* fact(n-1);
}
```

Write recursive function to find sum of n elements.

```
int sum (int n, int a[10])
{
    if (n==1)
        return a[0];
    else
        return a[n-1] + sum (n-1, a);
}
```

To get biggest element in the array.

```
int max (int max, int n, int a [10]
{
    if (n==1)
        if (max < a[0])
            return a[0];
        return max;
    return max (
```

```
int big (int n, int a[10])
{
    if (n==1)
        return a[0];
    return max (big (n-1, a), a [n-1])
}
```

```
int max (int a, int b)
{
    if (a>b)
        return a;
    return b;
}
```

GCD of 2 numbers

```
int gcd (int x, int y)
{
    if (x == y)
        return x;
    if (x > y)
        return gcd(x-y, y);
    if (x < y)
        return gcd(y, x);
}
```

Write recursive function to multiply 2 numbers

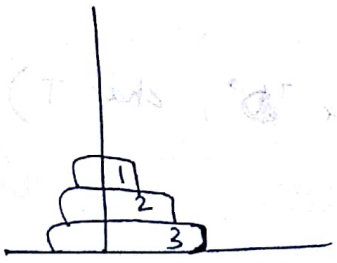
```
int mul (int x, int y)
{
    if (y == 1)
        return x;
    else {
        return [mul(x, y-1) + m];
}
```

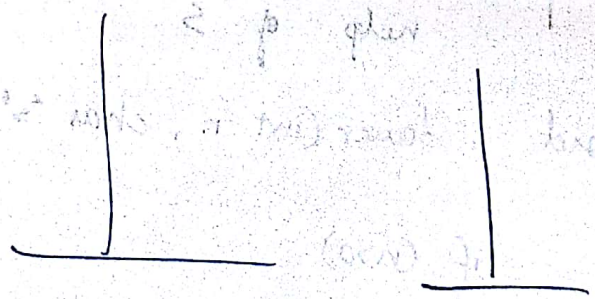
Write a recursive func to perform binary search

```
int bs(int a[30], int low, int high, int key)
{
    if (low > high)
        return -1;
    int mid = (low+high)/2;
    if (key = a[mid])
        return mid+1;
    if (key < a[mid])
        bs(a, low, mid-1, key);
    if (key > a[mid])
        bs(a, mid+1, high, key);
}
```

Tower of Hanoi

In the tower of Hanoi prob, we have 3 poles denoted with source (S), destination(D) and temp(T). On the source (S), n discs are stacked one above the other such that the disc with the larger diameter will be at the bottom.





The objective here is to move all the discs from the source to destination such that the destination should have the same property of source. To acheive this, the limitations are

i) only one disc can be moved across the poles.
ii) at any time, the disc with the larger dia should always be at the bottom.

Let n denotes the number of discs

Case 1: if n=1,

- we need only one move ie move disc from S to D.

if n=2, move are as follows.

- 1st (S to T)
- 2nd (S to D)
- 1 (T to D).

In general, the TOH problem can be solved recursively in the following manner

step 1; recursively move (n-1) discs from S to T. with the help of D.

step 2: Move nth disc from ~~source~~ S to D.

step 3: recursively (n-1) discs from T to D with the help of S.

```
void tower(int n, char 'S', char 'D', char T)
{
    if (n>0)
    {
        Tower(n-1, S, T, D);
        pf("\n move %d disc from %c to %c", n, S, D);
        Tower (n-1, T, D, S);
    }
```

Queues

rear insertion (enqueue)

front deletion (dequeue)

First in first out (FIFO)

Applications

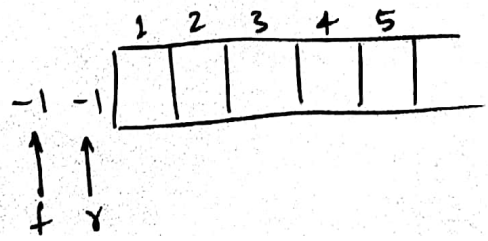
1. process and job scheduling.
2. resource allocation.

Implementation of queue

Let SIZE denote the size of the queue and the following structure definition is used to represent the queue.

```
Struct queue
{
    int f, r;
    int data[SIZE];
};  typedef struct queue QUEUE;
```

where f denotes front end and r denotes rear end. and array data is used to hold the elements. Initial value of f and r are -1. An empty queue can be represented as



A C function to insert an element onto a queue

```
void insertq (QUEUE *q, int ele)
{
    if (q->r == SIZE-1)
        pf("Queue is full");
    else
    {
        q->data[++(q->r)] = ele;
        if (q->f == -1)
            (q->f)++;
    }
```

A C func to delete an element from the queue.

```
int deleteq (QUEUE *q)
{
    int e

    if (q->f == -1)
    {   printf("Queue empty");
        return -1;
    }
    else
    {
        e = q->data[q->f];
        if (q->f == q->r)
        {
            q->f = -1;
            q->r = -1;
        }
        else
            (q->f)++;
        return e;
    }
}
```

A function to display the content of queue.

```
void display q (QUEUE *q)
{
    int i;
    if (q.f == -1)
        pf("Queue empty");
    else
    {
        pf("\n Queue content is");
        for (i= q.f ; i<= q.r ; i++)
            pf("%d \t", q.data[i]);
    }
}
```

There are 4 types of queues.

1. Ordinary queue (Linear queue)
2. Circular queue
3. Priority queue
4. Double ended queue (D-queue)

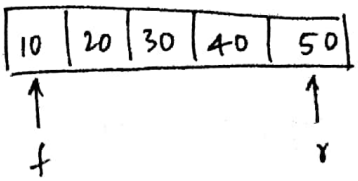
   Two types of D-queue

   i) input restricted queue. (ins only r , del using both)

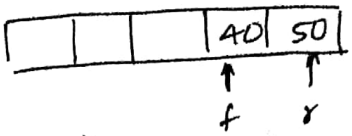
   ii) output restricted queue (del " f , ins " ")

of queue.

Consider a linear or ordinary queue with 5 elements as.



After deleting 3 elements, the queue will look like.



even though there are some empty slots in the begining of the queue, it cant perform an insertion operation as the rear end has reached a value of (size-1). To overcome this disadvantage, we make use of a queue type named circular queue.

Implementation of a circular queue

Let SIZE denotes the size of the queue and the following struct definition is used to represent a circular queue.

```
struct queue
{
    int f,r;
    int data[SIZE];
};
typedef struct queue CQUEUE;
```

insertion operation

SIZE 5

A c function to perform insertion op.

```
void    cqueue insert (CQUEUE *q, int item)
{
    if (q->f == (q->r + 1) % SIZE)
        pf ("\n queue full");
    else
    {
        q->r = (q->r+1) % SIZE;
        q->data [q->r] = item;
        if (q->f == -1)
            q->f = 0;
    }
}
```

Deletion operation

```
int    cqueue delete (CQUEUE *q)
{
    if (q->f == -1)
    {   pf ("\n q empty");
        return -1;
    }
    e = q->data [q->f];
    if (q->f == q->r)
```

→ `q->f = (q->f+1) % SIZE`
`if (q->f == q->r)`

```
    q->f = -1;
    q->r = -1;
  }
  else
    q->f = (q->f+1)%SIZE;
  return e;
}
```

Display operation

```
void displayq(QUEUE q)
{
  if (q.f == -1)
    printf("\n empty");
  else if (q.f <= q.r)
  {
    printf("\n Queue data");
    for(i=q.f ; i<=q.r ; i++)
      printf("%d\t", q.data[i]);
  }
  else
  {
    printf("\n Queue data");
    for(i=q.f ; i<SIZE ; i++)
      printf("%d\t", q.data[i]);
    for(i=0 ; i<=q.r ; i++)
      printf("%d\t", q.data[i]);
  }
}
```

i=q.f
i!=q.r
i=(i+1)%SIZE

Lab Prog 3

Implement a message queuing system using circular queue.

```
#include <stdio.h>
        <stdlib.h>
        <string.h>
#define   SIZE   5

struct   queue
{
     int  f,r;
     char  data[SIZE][20];
};
typedef   struct   queue   CQUEUE;

void   cqinsert (CQUEUE *q, char item[20])
{
     if (q->f==(q->r+1) % SIZE)
          pf("\n queue full");
     else
     {
          q->r = (q->r +1) % SIZE;
          strcpy (q->data[q->r], item);
          if (q->f==-1)
               q->f=0;
     }
}
```

```
char *  cqdelete (CQUEUE *q)
{
    char *e;
    if (q->f == -1)
    {   pf ("\n empty");
        return NULL;
    }
    e=q->data[q->f];
    if (q->f == q->r)
    {
        q->f=q->r=-1;
    }
    else
        q->f = (q->f+1) % SIZE;
    return e;
}
void  cqdisplay (CQUEUE q)
{
    if (q.f==-1)
        pf("\n empty");
    else if (f<=r)
    {
        pf("\n data is");
        for (i=q.f ; i<=q.r ; i++)
            pf ("%s\n", q.data[i]);
    }
```

```
        else
        {
            pf ("\n Queue data\n")
            for (i=q.f ; i<SIZE ; i++)
                pf ("%s\n", q.data [i]);
            for (i=0 ; i<=q.r; i++)
                pf ("%s\n", q.data[i]);
        }
}

int main()
{
    char item[20], *e

        int ch
    CQUEUE q
    q.f=-1; q.r=-1;

    for(;;)
    {
        pf(1. ins 2. del 3. disp 4. exit);
        scanf("%d", &ch); getchar();
        switch (ch)
        {
            case 1: pf("\n read message to be ins");
                    gets(item);
                    cqinsert (&q, item);
                    break
```

```
case 2: e= cq_delete(&q);
        if (*e != NULL)
            pf("The message deleted is %s", *e);
        break.
case 3: cq_display(q);
default: exit(0);
    }
}
```

1st test portions

Priority Queue

is a queue type where in the deletion (dequeue) op is based on the priority.

There are 2 types of priority queues

1. Ascending priority queues
2. Descending priority queues

There are 2 ways of implemendation

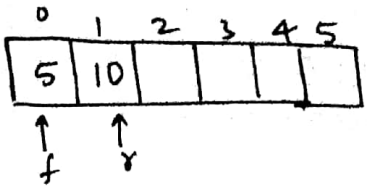
1. The insertion op is the normal one as compared to queue and the deletion operation needs to be modified as compared to the normal queue
2. The insertion op is a modified one where in the elements are inserted in an order onto the queue and the deletion is the normal deletion op

A C func to insert an element onto a priority queue

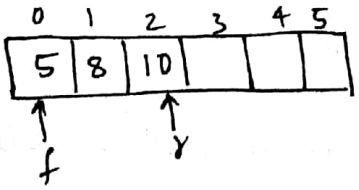
```
void pqinsert (QUEUE *q, int item)
{
    if (q->r == SIZE-1)
        pf("Full");
    else
    {
        int pos = q->r;
        q->r = q->r + 1;
        while (p>=0 && q->data[pos] >= item)
        {
            q->data[pos+1] = q->data[pos];
            pos--;
        }
        q->data[pos+1] = item;
        if (q->f == -1)
            q->f = 0;
    }
}
```

Trace the insertion function on a priority queue for the following elements

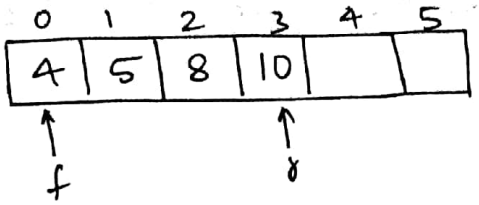
5 10 8 4 6.

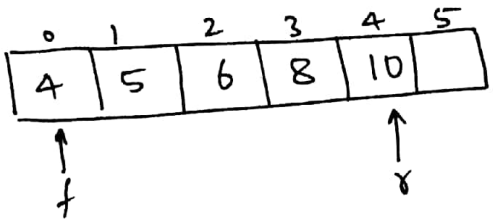


pos = 0
r = 1



pos = 1
r = 2

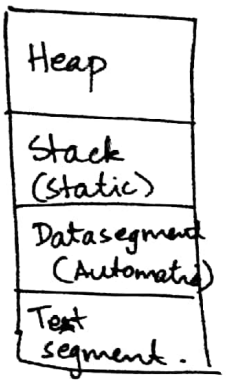




Dynamic Memory Allocation

Allocation of memory during runtime or execution time is called dynamic memory allocation.

In dynamic memory allocation, the memory will be allocated (for the calling function) (from the region called heap or store)



Q. Compare static with dynamic memory allocation.

Functions used to perform dynamic memory allocation are as follows:-

1. malloc
2. calloc
3. realloc

} allocation.

4. free } deallocation.

Dagling pointer → one that points to garbage address

malloc

- It allocates single block of memory
- On failure, it returns NULL
- On success, it returns the starting address (pointer) from where memory is allocated.
- The return type of the malloc function is a void pointer. which can be type casted based on the purpose
- By default, garbage values will be stored under the allocated memory

Syntax

```
#include <stdlib.h>

void * malloc (int size)   /* one argument for mallac func*/
```

-

```
datatype *ptr;
    ptr= (datatype*) malloc (sizeof (datatype));
  if (ptr==NULL)
  {
  pf("insufficient memory");
     exit(0);
   }
```

Write the syntax to allocate memory dynamically for the following:-

i) for 1 integer.

```
int *ptr;
ptr = (int *) malloc (sizeof(int));
```

ii) for n integers.

```
int *ptr , n;
ptr = (int *) malloc (n * sizeof (int));
```

iii) for n characters

```
char *ptr;
int n;
ptr = (char *) malloc(n*size of (char));
```

WAP to find Σ^{n} elements ~~of array~~ using dynamic memory allocation.

```
int main()
{
    int *ptr, i, sum=0, n;
    pf("\n Read value for n: ");
    sf("%d ", &n);
    ptr = (int *) malloc(n * sizeof (int));
    pf("Reand n elements");
    for (i=0; i<n; i++)
    {
        sf("%d", ptr+i);
        sum = sum + *(ptr+i);
    }
```

WAP to read and print one student info using DMA.

```
Struct student
{
    char name[10];
    int sem;
}
int main()
{
    struct student *ptr;
    ptr = (struct student *) malloc (sizeof (struct student));
    pf("Read name");
    sf("%s", ptr->name);
    pf("Read sem");
    sf("%d", &ptr->sem);
}  pf...
```

WAP to read n students...

```
struct student
{
    char name[10];
    int sem;
}
int main()
{
    struct student *ptr;
    int n, i;
    pf(Read n); sf(n);
    ptr = (struct student *) malloc(n * sizeof(struct student));
```

calloc

Allocates memory dynamically using multiple blocks return type is void pointer which can be typecasted.

Syntax:

```
void * calloc (int n, int size)
```

first argument n is no of blocks
2nd arg size is size of each block.

By default, a value of arithmetic zero will be stored.

Realloc used to resize (increase or decrease) the memory allocated either by malloc or calloc

SYNTAX:

```
void * realloc (void *ptr, int newsize)
```

Free deallocates the memory allocated previously either by malloc, calloc, or realloc.

SYNTAX:

```
void free (void *ptr)
```

LINKED LIST.

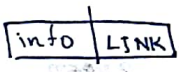
is defined as collection of sequential data items. These sequential data items are termed as "nodes"

(or) LL is a collection of nodes.

linked list uses dynamic memory allocation.

A node in a LL has 2 fields

1. info field - where info is stored.
2. link field - holds the address of next.



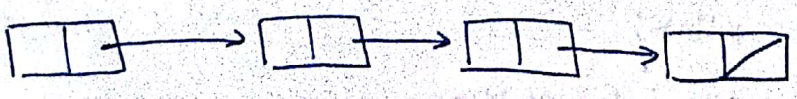
A node in a LL can be represented with the following struct def".

```
struct node
{
   int info;
   struct node * link;   → self referential structure.
};
```

There are 4 types of LL, namely.

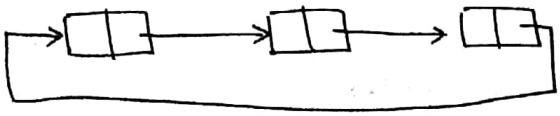
1) Singly LL

here, each node apart from info field, the link field contains only one address ie, address of the next node, and the last node link field contains null.



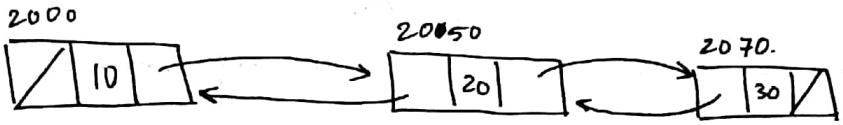
2) Circular singly LL

singly LL wherein the last node contains the address of the first node

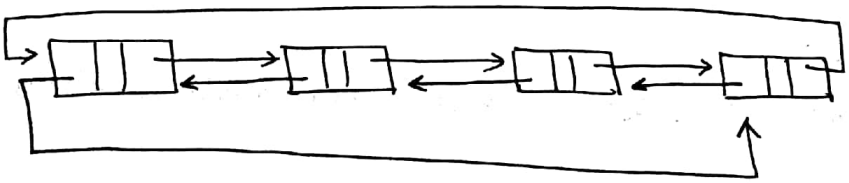


3) Doubly LL

here, each node contains the info field and the the link field contains 2 address. ie address of next node and previous node.



4) Circular double linked list



Single Linked List

To implement a single LL, we'll consider following struct defn to represent a node.

```
struct node
{
    int info;
    struct node *link;
}; typedef struct node * NODE;
```

To insert a new node at the front end.

```
NODE insertfront (NODE first , int item)
{
   NODE temp ;
   temp = (NODE) malloc (sizeof (struct node));
   temp → info = item;
   temp → link = first;
   return temp;
}
```

first is a variable of the ~~at~~ data type • NODE = (struct node *)

To insert a node at rear end.

```
NODE insertrear (NODE first ~~rear~~, int item)
{
   NODE temp; cur ,
   temp = (NODE) malloc (sizeof (struct node));
   temp → info = item;
   temp → link = NULL;
   if (first = = NULL)
        return temp;
   ~~while (cur → link ! = NULL)~~
   cur = first;
   while ( cur → link ! = NULL)
            cur = cur → link;
   cur → link = temp;
   return first;
}
```

To delete a node at front end.

```
NODE deletefront (NODE first)
{
    NODE temp;
    temp = first;
    if ( first == NULL)
    {
        pf("\n empty");
        return NULL;
    }
    first = first -> link;
    pf ("\n element del is %d ", temp -> info);
    free (temp);
    return first;
}
```

To delete a node at rear end.

```
NODE deleterear (NODE first)
{
    NODE ~~temp~~ cur, prev
    if (first == NULL)
    {
        pf("\n empty");
        return NULL;
    }
    cur = first
    prev = NULL
    while (cur -> link != NULL)
    {   prev = cur;
        cur = cur -> link;
    }
    prev -> link = NULL;
```

```
        pf(" element deleted is %d " cur->info);
        free (cur)
        return first;
}
```

To display content of single LL

```
void display (NODE first)
{
    if (first == NULL)
        pf("list empty");
    else
    {
        pf(" list content is");
        temp = first;
        while (temp != NULL)
        {
            pf("%d", temp->info);
            temp = temp->link;
        }
    }
}
```

A stack can be implemented by following funcs

1. insert front - push.
2. delete front - pop.
3. display

A queue can be implemented by the following funcs

1. insert rear - enqueue
2. delete front - dequeue.
3. display.

Lab prog 4.

to multiply 2 polynomials

ex:- $4x^3 + 2x^2 + 8x + 1$ (4 term).

4 * (x)^3 + 2 * (x)^2 + 8 * (x)^1 + 1 * (x)^0.

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
   int co, po;
   struct node *link;
};
typedef struct node *NODE;

NODE insert (NODE first, int co, int po)
{
   NODE temp, cur;
   temp = (NODE) malloc (sizeof (struct node));
   temp -> co = co;
   temp -> po = po;
   temp -> link = NULL;
   if (first == NULL)
         return temp;
   cur = first;
   while (cur -> link != NULL)
         cur = cur -> link;
```

```
    cur->link = temp;
    return first;
}

void display (NODE first)
{
    NODE temp;
    if (first == NULL)
        pf("\n polynomial is empty");
    else
    {
        temp = first
        while (temp->link != NULL)
        {
            pf(" %d * (x)^ %d +", temp->co, temp->po);
            temp = temp->link;
        }
        pf ("%d * (x)^ %d \n", temp->co, temp->po);
    }
}

NODE polymultiply (NODE poly1, NODE poly2)
{
    NODE first, second;
    NODE res = NULL;
    for (first = poly1; first != NULL ; first = first->link)
        for (second = poly2; second != NULL; second = second->link).
            res = addterm (res, first->co * second->co, first->po * second->po)
    return res;
}
```

```
NODE addterm (NODE res ; int co, int po)
{
    NODE temp, cur ;    int flag = 0;
    temp = (NODE) malloc (sizeof (struct node));
    temp -> co = co ;
    temp -> po = po ;
    temp -> link = NULL;
    if (res == NULL)
            return temp;
    cur = res
    while (cur != NULL)
    {
        if (cur -> po == po)
        {
            cur -> co = cur -> co + co;
            flag = 1;
            return res;
        }
        cur = cur -> link;
    }
    if (flag == 0)
    {
        cur = res;
        while (cur -> link != NULL)
                cur = cur -> link;
        cur -> link = temp;
        return res;
    }
```

```
int main ()
{
    NODE poly1 = NULL, poly2 = NULL, poly3 = NULL;
    int m, n, co, po, i;
    printf ("\n Read no of terms for first poly:");
    scanf ("%d", &m);
    for (i=1 ; i<=m ; i++)
    {
        pf("\n Read co and po for %d term", i);
        sf ("%d %d", &co, &po);
        poly1 = insert (poly1, co, po);
    }
    printf ("\n Read no of terms for second poly:");
    scanf("%d", &n);
    for (i=1 ; i<=n ; i++)
    {
        pf ("\n Read co and po for %d term", i);
        sf ("%d %d", &co, &po);
        poly2 = insert (poly2, co, po);
    }
    printf ("\n first poly is\n");
    display (poly1);
    printf ("\n second poly is\n");
    display (poly2);
    poly = poly multiply (poly1, poly2);
    pf ("\n result poly is\n");
    display (poly);
    return 0;
}
```

WAP func to delete a node based on info field.

```
NODE deleteinfo (NODE first , int key)
{
    NODE temp; cur, prev
    if (key == first ->info)
    {
        temp = first
        first = first-> link;
        pf ("\n ele del is %d", temp -> info);
        free (temp);
       return first;
    }

  prev = NULL;
  cur = first;
  while (cur != NULL && key != cur -> info)
  {
     prev = cur ;
     cur = cur -> link;
  }
  if (cur == NULL)
  {
    pf ("\n %d has not been there ", key);
    return first ;
  }
  prev -> link = cur -> link ;
  pf ("\n %d is deleted ", cur -> info.
  free (cur);
  return first;
```

C function to delete a node based on the pos.

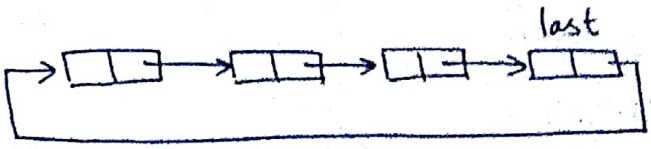
```
NODE deletepos(NODE first, int pos)
{
    NODE temp, cur, prev;
    int ct = 1;
    if (first == NULL)
    {
        pf("Nope");
        return first;
    }
    if (pos == 1)
    {
        temp = first;
        first = first -> link;
        pf("\n %d is deleted", temp -> info);
        free(temp);
        return first;
    }
    cur = first;
    while (cur != NULL && ct != pos)
    {
        prev = cur;
        cur = cur -> link
        ct = ct + 1;
    }
    if (ct != pos)
    {
        pf("\n ___ ");
        return first;
    }
    prev -> link = cur -> link;
    pf("%d is deleted", cur -> info);
    free(cur);
    return first
}
```

WAF to create an ordered list.

```
NODE insertorder(NODE first, int item)
{
   NODE temp, cur, prev;
   temp= (NODE) malloc (sizeof (struct node));
   temp -> info= item
   temp -> link = NULL
   if (first == NULL )
        return temp;
   if (item <= first ->info)
   {
       temp-> link= first ;
        return temp;
    }
    cur = first
   while (cur != NULL  &&  item > cur -> info)
   {   prev = cur;
       cur = cur -> link;
   }
   prev -> link = temp;
   temp -> link= cur;
    return first;
}
```

Circular Single linked list

In case of a circular single linked list, The link field of the last node contains address of the first node.



Let last be the reference of a circular list which holds the address of the last node in the list

If last = null, implies list is empty

If last → link = last, implies only one node

WAF to add a node at front end.

```
NODE insert front (NODE last, int item)
{
  NODE temp;
  temp = (NODE)malloc (sizeof (struct node));
  temp → info = item;
  if (last == NULL)
  {
    temp → link = temp;
    return temp;
  }
  else
  {
    temp → link = last → link
    last → link = temp;
    return last;
  }
}
```

WAF to insert at rear end.

```
NODE insert_rear (NODE last, int item)
{
    NOD temp;
    temp = (NODE) malloc (sizeof(struct node));
    temp -> info = item;
      if (last == NULL)
      {
          temp -> link = temp;
          return temp;
      }
      else
      {
          temp -> link = last -> link;
          last -> link = temp;
          return temp;
      }
}
```

WAP to delete a node from front end.

```
NODE deletefront(NODE last)
{
    NODE * temp;
    if (last == NULL)
    {
        pf(empty)
        return NULL;
    }
    if (last->link == last)
    {
        printf("%d deleted", last->info);
        free(last);
        return NULL;
    }
```

→ allocate memory for temp

```
    temp = last->link;
    last->link = temp->link;
    pf("\n %d is deleted", temp->info);
    free(temp);
    return last;
}
```

WAF to delete rear.

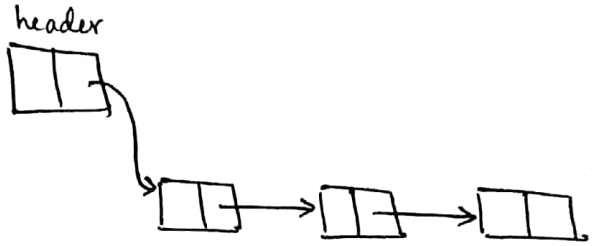
```
NODE deldelrear(NODE last)
{
    NODE cur;
    if (last == NULL)
    {
        pf(empty);
        return NULL;
    }
    if (last -> link == last)
    {
        pf("%d is deleted", last -> info);
        free(last);
        return NULL;
    }
    cur ...
    cur = last -> link;
    while (cur -> link != last)
            cur = cur -> link;
    cur -> link = last -> link;
    pf("%d ...", laust -> info);
    free(last);
    return cur;
}
```

WAF to display

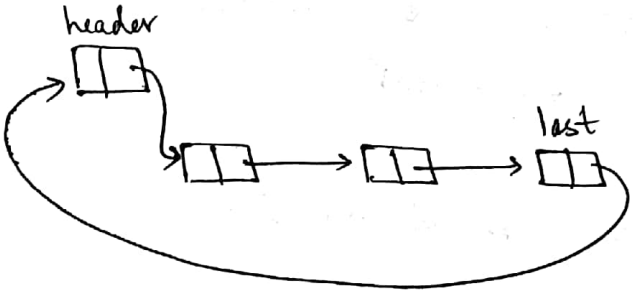
```
void display ( NODE last).
{
    NODE temp
    if (last == NULL)
    {
      pf (empty);
    }
    else
    {
      allocate .. temp.
      temp = last->link;
      while (temp != last)
      {
          print("%d\t", temp->info);
          temp = temp->link;
      }
      pf("%d\t", last->info);
    }
}
```

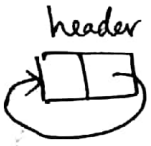
Header node

Tis a special node in case of a LL which generally contains the info about the linked list (no of nodes, objective of LL) and the link field of the header node contains address of first node in the list.



single LL





list is empty.

Circular single LL.

In case of CSLL, the list is empty if header of link points to header

In any list which has an header node before creating the list, memory should be allocated to the header node and the link field should be assigned to itself

A func to insert node at front end. in a circular list with header node.

```
NODE insertfront (NODE head, int item)
{
    NODE temp;
    temp = (NODE) malloc (sizeof (struct node));
    temp → info = item;
    if (head → link == head)
    {
        temp → link = head;
        head → link = temp;
        return head;
    }
    else
    {
        temp → link = head → link;
        head → link = temp;
        return head;
    }
}
```

Add a node at rear end.

```
NODE insertrear (NODE head, int item)
{
    NODE temp;
    temp = (NODE) malloc (sizeof (struct node));
    temp → info = item;
    if (head → link == head)
    {
        temp → link = head;
        head → link = temp;
        return head;
    }
```

```
    cur = head -> link;
    while (cur -> link != head)
        cur = cur -> link;
    ~~cur -> link = temp;~~
    temp -> link = cur -> link;
    ~~temp~~ cur -> link = temp;
    return head;
}
```

Deletion from front end

```
NODE delete front (NODE head) .
{
    NODE temp;
    if (head -> link == head)
    {
        printf ("In list empty");
        return head;
    }
    ~~temp~~ = ~~head->link~~ temp = head -> link;
    head -> link = temp -> link ;   pf (temp -> info);
    free (temp);
    return head;
}
```

Delete at rear end.

```
NODE delete rear (NODE head)
{
    NODE ~~temp~~ cur, prev.
    if (head -> link == head)
    {
        pf (empty)
    return head;
```

```
    cur = head->link.
    while (cur->link != head)
            prev = cur;
            cur = cur->link;
    prev->link = head;
    pf (cur->info);
    free(cur);
    return head;
}

// display function
void display (NODE head)
{
    if (head->link == head)
    {
        ~~head~~ print (empty)
        return;
    }
    temp = head->link.
    while (temp != head)
    {
        pf (temp->info)
        temp = temp->link;
    }
}
```

Lab program 5.

To add 2 long integers using circular list with header node.

---

① Read 2 nos & create list to hold no. (insert rear).

② check size of both, if equal. ie if not, add zeros from front

③ reverse both lists.

```
struct node
{
   int info;
   struct node * link;
};
typedef struct node * NODE;

int main()
{
   NODE h1, h2,
   int i,
   char a[100], b[100];

   h1 = (NODE) malloc (sizeof(struct node));
   h2 = (NODE) malloc (sizeof (struct node));
```

```
    h1->link = h1;
    h2->link = h2;
    pf("Read the first no");
    scanf("%s", a);
    for (i=0; a[i]!='\0'; i++)
    {
        h1 = insert rear (h1, a[i]-'0');
    }
    pf("first no is");
    display (h1);
    pf("read second no");
    scanf("%s", b);
    for (i=0; b[i]!='\0'; i++)
        h2 = insertrear (h2, b[i]-'0');
    pf("second no is");
    display (h2);
    append (h1,h2);
    pf("First no"); display (h1);
    pf("Second no"); display (h2);
    add (h1, h2);
    return 0;
}

NODE insert rear (NODE head, int item)
{
    NODE temp;
    temp = (NODE*) malloc (sizeof (struct node));
    temp -> info = item;
    if (head -> link == head)
        ;
}
```

```
void display (NODE head);
{
    :
}
void append (NODE h1 , NODE h2)
{
    int ct1=1 , ct2=1 , diff
    s1= size (h1);
    s2= size (h2);
    if (s1>s2)
    NODE temp;
    temp = h1→link;
    while(temp → link != h1)
    {
        ct1 = ct1+ 1;
        temp = temp →link;
    }
    temp = h2→ link;
    while (temp → link != h2)
    {
        ct2 = ct2 +1)
        temp = temp →link;
    }
    if (ct1 >ct2)
    {
        diff = ct1 - ct2;
        for (i = 1; i <= diff ; i++)
            h2= insert zero(h2,0);
    }
    else if (ct2 > ct1)
    {
        diff = ct2 - ct1;
        for (i=1; i<= diff ;i++)
            h1= insert zero (h1, 0);
    }
```

```
NODE insterzero (NODE head, int item)
{ ;
                                    // insert front func.
}

void add (NODE h1, NODE h2)
{
   NODE h, temp1, temp2;
   int sum = 0, carry = 0, x;
   h = (NODE) malloc (sizeof (struct node));
   h -> link = h;
   h1 = reverse (h1);
   h2 = reverse (h2);
   temp1 = h2 -> link;
   temp2 = h2 -> link;
   while (temp1 != h1 && temp2 != h2)
   {
      x = temp1 -> info + temp2 -> info + carry;
      sum = x % 10;
      carry = x / 10;
      h = insert (h, sum);
      temp1 = temp1 -> link;
      temp2 = temp2 -> link;
   }
   if (carry > 0)
      h = insert (h, carry);
   h = reverse (h);
   display (h);
}
```

```
NODE reverse(NODE head)
{
    NODE prev, cur, next;
    cur = head → link;
    prev = head.
    while (cur != head)
    {
        next = cur → link;
        cur → link = prev;
        prev = cur;
        cur = next;
    }
    head → link = prev;
    return head;
}
```

Write a C func to allocate memory to a node in LL.

```
NODE getnode()
{
    NODE temp;
    temp = (NODE) malloc (sizeof (struct node));
    return temp;
    if (temp == NULL)
    {
        pf ("\n insufficient memory");
        exit(0);
    }
    return temp;
}
```

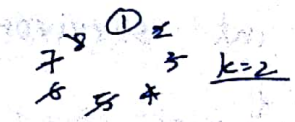
Write a func to deallocate memory for a node.

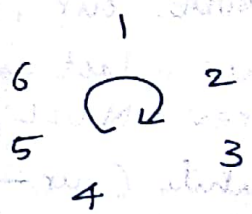
```
void freenode (NODE temp)
{
    free(temp);
}
```

JOSEPHUS PROBLEM

Let us assume that there are n number of people standing in a circle where we need to eliminate every third person until we are left out with one person.

Let the value of n be 6.





order of elimination is 3, 6, 4, 2, 5 ∴ 1 is survive.

In general, if we eliminate every k^{th} person from the given set, then J(n, k) should give us the survivor

If k=2, then J(n,k) can be obtained from the binary expansion of n by a circular left shift of the binary digits of n.

If

mory to a node

node));

Implemat ation of Josephus problem

```
struct node
{
    int info;
    struct node *link;
};
typedef struct node *NODE;

// write insertrear function on a circular list.
// write display function.

int survivor (NODE last, int k)
{
    NODE cur, prev;
    cur = last -> link;
    prev = NULL;
    while (cur -> link != cur)
    {
        for (i=0; i <k-1 ; i++)
        {
            prev = cur;
            cur = cur -> link;
        }
        prev -> link = cur -> link;
        pf (cur -> info);

        free (cur);
        cur = prev -> link.i

    }
    retur current -> info;
}
```

```
int main()
{
    int i, n, k, sur
    NODE last = NULL;
    pf("Read no of people");
    scanf("%d", &n);
    ~~insert~~ ~~last = insertrear(last,~~
    for (i=1; i<=n; i++)
        last = ~~insertrear(last~~, insertrear(last, i);
    display(last);
    pf("Read k");
    scanf("%d", &k);
    ~~last=>~~
    sur = survivor(last, k);
    printf("%d is survivor", sur);
}
```

cular list.

from DMA.

Quiz two

Doubly LL

A node ← info, llink, rlink

```
struct node
{
    int info;
    struct node *llink;
    struct node *rlink;
};  typedef struct node *NODE.
```

```
NODE insertfront (NODE first, int item)
{
  NODE temp;
  temp=(NODE) malloc (sizeof (struct node));
  temp->info = item;
  temp->llink = temp->rlink = NULL;
  if (first == NULL)
    return temp;
  temp->rlink = first;
  first->llink = temp;
  return temp;
}
```

```
NODE insertrear (NODE first, int item)
{
  NODE temp;
  temp = (NODE) malloc (sizeof (struct node));
  temp->info = item;
  temp->llink = temp->rlink = NULL;
  if (first == NULL)
    return temp;
  cur = first;
  while (cur->rlink != NULL)
    cur = cur->rlink;
  temp->llink = cur;
  cur->rlink = temp;
  return first;
}
```

```
NODE delete front(NODE first)
{
    NODE temp;
    if (first == NULL)
    {
        printf(" empty");
        return NULL;
    }
    temp = first;
    temp = temp -> rlink.
    temp -> llink = NULL;
    printf ("first -> info");
    free(first);
    return temp;
}

NODE delete rear (NODE first)
{
    NODE cur, prev;
    if (first == NULL)
    {
        printf("(" empty");
        return NULL;
    }
    if(first -> rlink == NULL)
    {
        pf(" first -> info ");
        free (first);
        return NULL;
    }
    cur = first.
    prev = NULL
    while (cur -> rlink != NULL)
    {
        prev = cur;
        cur = cur -> rlink;
    }
```

```
        pf("deleted is %d", cur->info);
        prev->rlink = NULL;
        free(cur);
        return first;
    }

    void display (NODE first)
    {
        NODE temp;
        if (first == NULL)
        {
            printf("empty");
            return;
        }
        temp = first;
        while (temp != NULL)
        {
            printf("%d\t", temp->info);
            temp = temp->rlink;
        }
    }
```

i) Function to delete a node based on info field:

ii) ~~based~~ delete based on position

iii) insert based on position.

iv) ordered doubly LL.

v) to reverse doubly LL.

vi) to find max and min.

Lab program 6

To represent a sparse matrix using doubly linked list.

Spare matrix is a matrix where 90-95% of the matrix contents are zeroes.

```
#include <stdlib.h>
#include <stdio.h>

struct node
{
    int row, col, info;
    struct node *rlink, *llink;
}; typedef struct node *NODE;

#
NODE insert (NODE first, int r, int c, int item)
{
    NODE temp, cur;
    temp = (NODE) malloc (sizeof(struct node));
    temp->info = item;
    temp->row = r;
    temp->col = c;
    temp->llink = temp->rlink = NULL;
    if (first == NULL)
        return temp;
    cur = first;
    while (cur->rlink != NULL)
        cur = cur->rlink;
```

```
        cur->rlink = temp;
        temp->llink = cur;
        return first;
}

void display (NODE first)
{
    NODE temp;
    if (first == NULL)
    {
        printf ("empty");
        return;
    }
    temp= first
    pf("\n row\t col \t value\n");
    while(temp != NULL)
    {
        pf("%d \t %d\t %d \n", temp-> row, temp->col , temp->info
        temp = temp -> rlink;
    }
}
void display matrix (NODE first , int m , int n)
{
    int i,j
    NODE temp = first
    for (i= 1 ; i<=m ; i++)
    {
        for (j=1; j<=n ; j++)
        {
            if (temp != NULL && temp->row==i &&
                    temp -> col==j)
            {
                printf ("%d\t", temp->info);
```

```
            else
                printf("0\t");
        }
        printf("\n");
    }
```

```
int main()
{
    int m,n,i,j, item;
    NODE first=NULL;
    pf("\n read order of matrix");
    sf("%d %d",&m, &n);
    pf("\n read the elements\n");
    for(i=1; i<=m; i++)
    {
        for (j=1; j<=n; j++)
        {
```

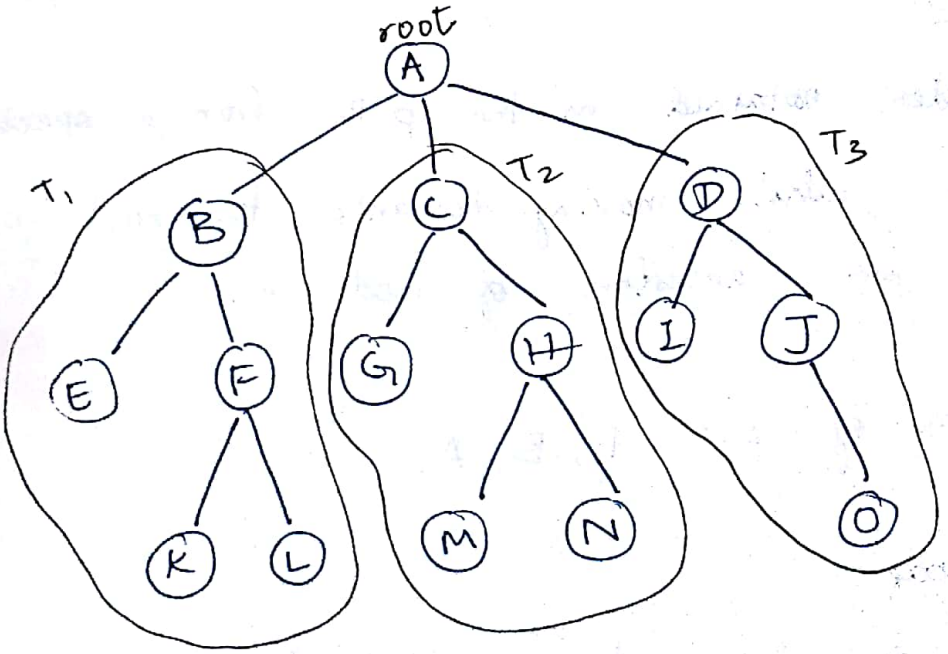
```
                sf("%d", &item);
                if (item != 0)
                    first = insert (first, i, j, item);
            }
        }
    display (first); pf("matrix is");
    display matrix (first, m, n);
    return 0;
}
```

UNIT 4

Trees

A tree is a finite set of one or more nodes that exhibits the parent and the child relationship such that

- There is a special node called as root node
- The remaining nodes are partitioned into disjoint subsets denoted by $T_1 T_2 \ldots T_n$ which are termed as subtrees



Tree is non-linear non-primitive DS where data is stored in a heirarchial fashion.

Degree of a node

The number of subtrees of a given node is termed as the degree of a node.

degree of node A = 3

E = 0

Sibling

2 or more nodes having a common parent are called siblings.

Ancestors

The nodes obtained in the path from a specific node x while moving towards the root node are termed as ancestors of node x.

Ancestors of L = F, B, A.

Leaf node:

The nodes in a tree which have degree of 0.

E, K, L, G, M, N, I, O.

Level of a node

The distance of a node from the root node

the no of edges passed through root node to reach a node.

A = L0

BCD = L1

EFGHJI = L2

KLMNO = L3

Height of a tree

The no of edges on the longest downward path b/w the root and the leaf.

Last level + 1 height = 4 ✓

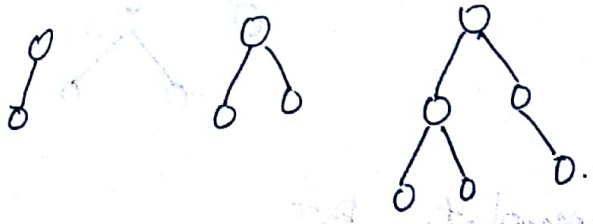
if ~~nodes~~ ~~are~~ no of nodes processed, he = 4 ✓

if no edges, he = 3 ✗

Binary tree

A tree in which each node has either 0, 1 or 2 subtrees.

Eg:-



(OR) (for marks):

A binary tree T is defined as a finite set of elements called nodes such that, ~~T is empty~~

i) T is empty (called empty tree or null tree)

ii) T contains a distinguished node 'R' called root of the node and the remaining nodes of T

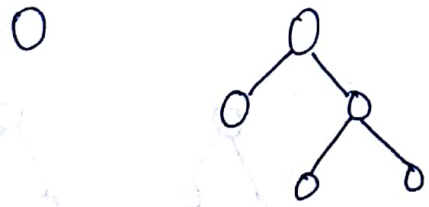
form an ordered pair of disjoint binary trees T_1 and T_2 respectively where T_1 is left subtree and T_2 is ~~left~~ right subtree.

Types of binary trees

1. Strictly binary tree
2. Complete BT
3. Almost complete BT
4. B search T
5. expression T
6. Skewed BT

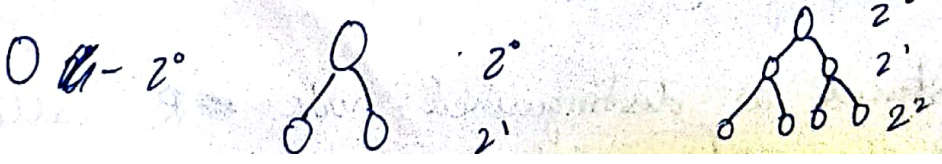
Strictly BT

If outdegree of every node in the tree is either 0 or 2 then it is Strictly BT.



Complete BT

A strictly BT having 2^i nodes at any given level i.

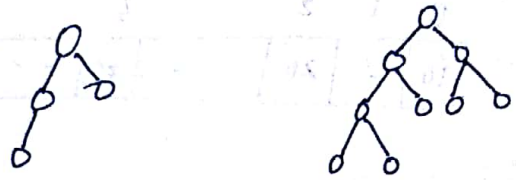


Almost complete BT (Heap) ☆☆☆.

It is CBT with the following properties

i) Apart from the last level, the other remaing levels should have 2^i nodes

ii) The last level the nodes should be left filled.



Skewed BT

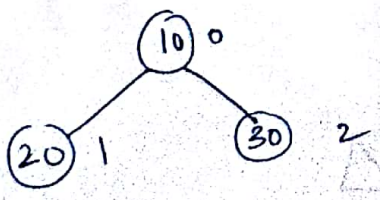
If ~~the~~ tree is built either on only one side, it's called skewed tree (left or right skBT)

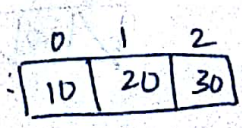
Repr of a BT

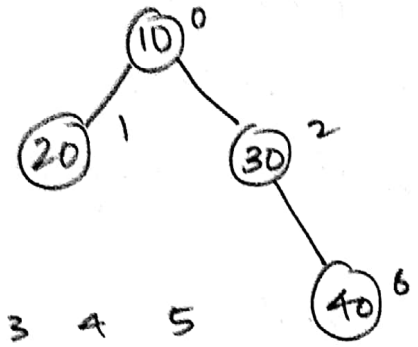
A binary tree can be represented in 2 ways

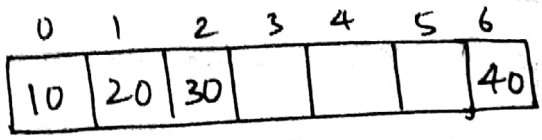
i) Array repr
ii) List repr

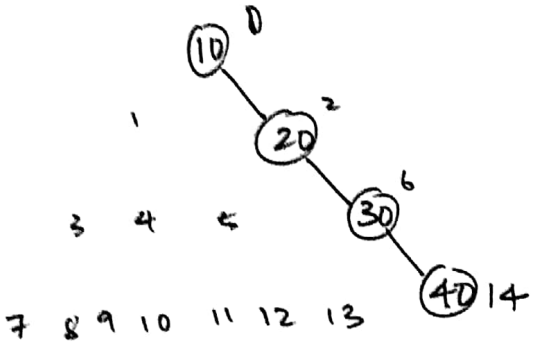
Array repr

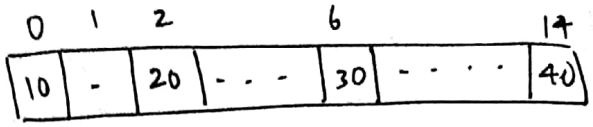










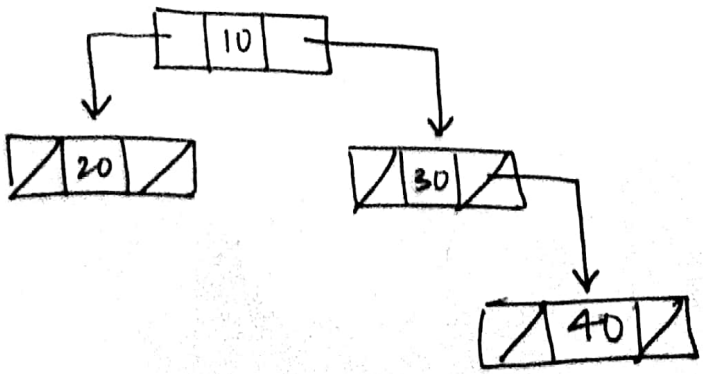


If tree is not complete, then wastage of memory.

List repr

In the list repr of a BT, we create a DLL where in each node has 3 fields.

1. info field - stores the info or key
2. llink - holds address of left subtree
3. rlink - holds address of right subtree.



The following struct def used to repr a node of BT

```
struct node
{
    int info;
    struct node *llink;
    struct node *rlink;
};
type def struct node * NODE;
```

root is a variable of type NODE which holds the address of root node

Operations on BTs

traverser opn ★☆★

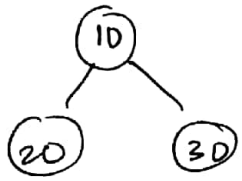
Traversal is the most common operation wherein we visit each node of a binary tree exactly once. There are 3 ways of traversal.

i) Preorder
ii) In order
iii) Postorder.

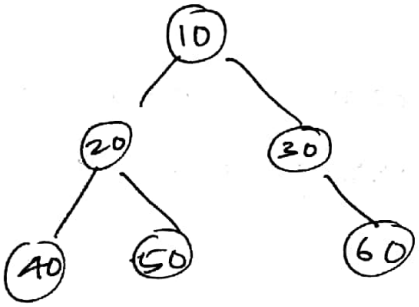
i) Preorder traverser.

The recursive definition to perform traversal on the binary tree is as follows.

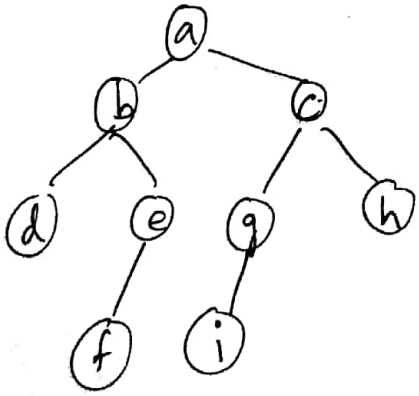
1. Visit the root node
2. recursively traverse the left subtree in the preorder.
3. recursively traverse the right subtree in the preorder.



i) 10 20 30
ii) 20 10 30
iii) 20 30 10



i) 10 20 40 50 30 60
ii) 40 20 50 10 30 60
iii) 40 50 20 | 60 30 10



i) a b d e f c g i h

~~ii) d b f e a i g c h~~

ii) d b f e a i g c h

iii) d f e b i g h c a.

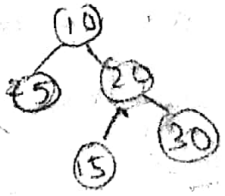
Recursive function to perform preorder traverser

```
void preorder (NODE root)
{
    if (root != NULL)
    {
      printf ("%d\t", root -> info);
      preorder (root -> llink);
      preorder (root -> rlink);
    }
}
```

(ii) Inorder traverser

The recursive definition to perform the inorder traversal on a binary tree is as follows

1. recursively visit the left subtree in inorder
2. visit the root node
3. recursively visit the right subtree in inorder




```
void inorder (NODE root)
{
    if (root != NULL)
    {
      inorder (root -> llink);
      pf("%d \t", root -> info);
      inorder (root -> rlink);
    }
}
```

iii) post order traverser

The recursive def^n to perform post order traverser is as follows ⇒

1. Recursively visit the left subtree in post order
2. Recursively visit the right subtree in post order
3. Visit the root node.

```
void postorder (NODE root)
{
    if (root != NULL)
    {
        postorder (root → llink);
        postorder (root → rlink);
        pf("%d\t", root → info);
    }
}
```

Assignment Q

Q. Write an iterative function to perform traversal on binary tree.

Construct a binary tree given the following traverser

i) preorder : a b d g h c e i f

inorder : g d h b a e i c f.

nder traverser

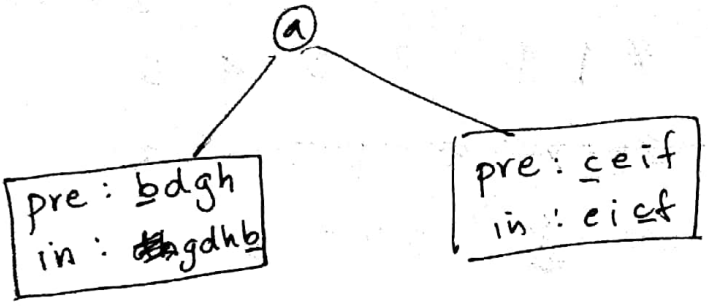
post order

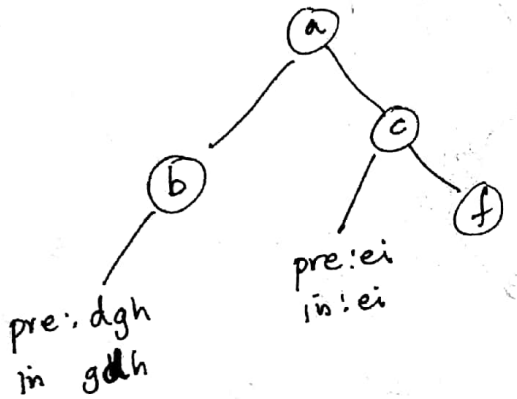
post order

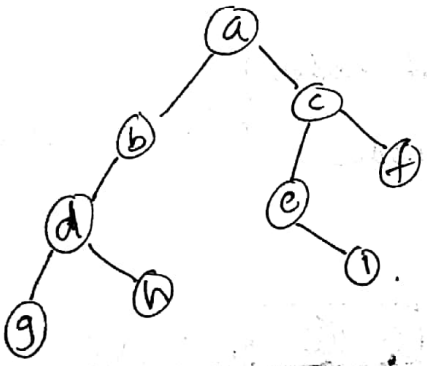
form traversal

lowing traverser.

i)



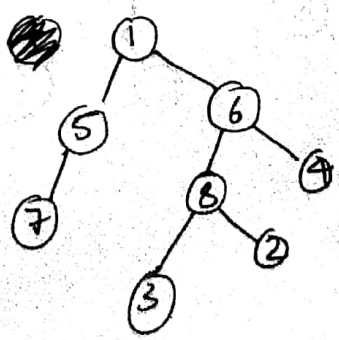




ii)

post : 7 5 3 2 8 4 6 1

in : 7 5 1 3 8 2 6 4



iii)

~~* $34~~ *$234

2 * 3 $4

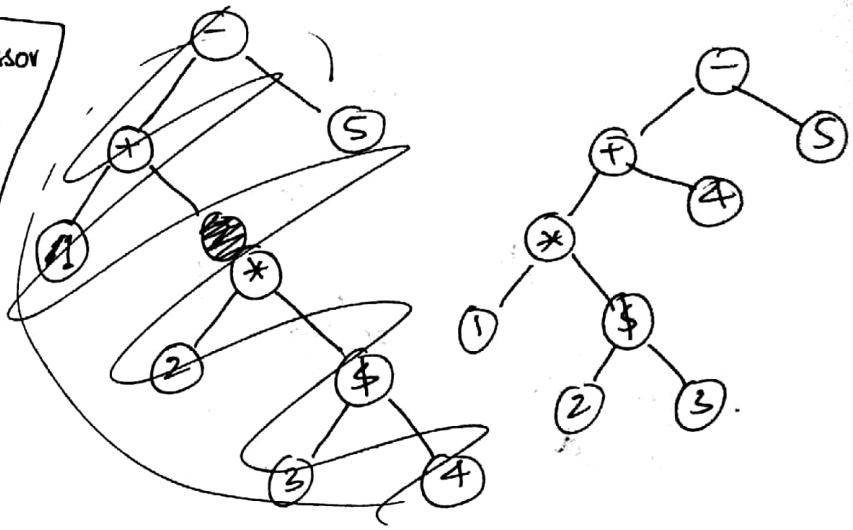
pre: − + * 1 $ 2 3 4 5

in : 1 + 2 * 3 $ 4 − 5

1 * 2 $ 3 + 4 − 5

preorder predecessor node of 2 is $

successor is 3

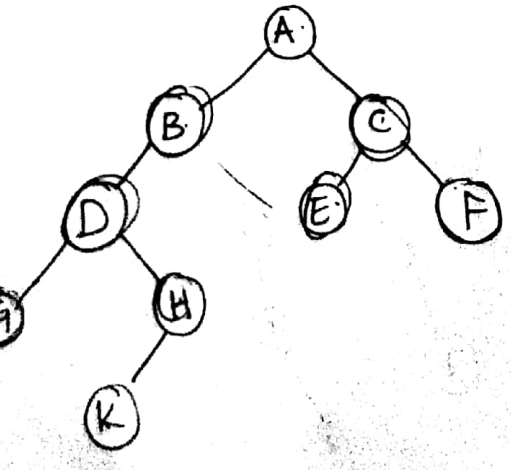


operators are non-leaf nodes
operands are leaf nodes
} AKA expression tree.

iv)

~~post~~ pre : A B D G H R C E F

post : G K H D B E F C A



A

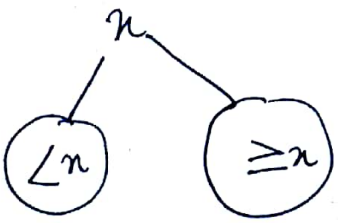
x = right child = predecessor of root node in post

y = left child = $ successor of root in pre.

if at any point x == y, tree is not unique.

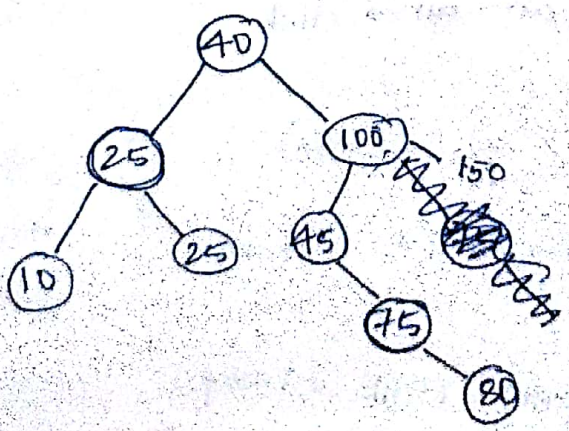
Binary search tree

is a binary tree in which for every node x, the left subtree of x will always be less than x and right subtree of x will always be greater than or equal to x.



Create a binary search tree for the given set of elements.

40 100 25 45 25 75 10 80



in: 10 25 25 40 45 75 80 100

A C func to create BST

```
NODE createBST(NODE root, int key)
{
    NODE temp, prev, cur;
    temp = (NODE) malloc(sizeof(struct node));
    temp->info = key;
    temp->llink = temp->rlink = NULL;
    if (root == NULL)
        return temp;
    prev = NULL;
    cur = root;
    while (cur != NULL)
    {
        prev = cur;
        if (key < cur->info)
            cur = cur->llink;
        else
            cur = cur->rlink;
    }
    if (key < prev->info)
        prev->llink = temp;
    else
        prev->rlink = temp;
    return root;
```

Program 7

To create BST and to perform the following operation
insert, delete, pre, in, post.

```
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int info;
    struct node *llink;
    struct node *rlink;
}; typedef struct node *NODE;

//create BST func

// pre, in, post

NODE delete (NODE root, int key)
{
  NODE temp

  if (root == NULL)
      return root;
  if (key < root->info)
      root->llink = delete (root->llink, key);
  else if (key > root->info)
      root->rlink = delete (root->rlink, key);
```

```
else
{
    if(root->llink == NULL)
    {
        temp = root->rlink;
        free(root);
        return temp;
    }
    else if(root->rlink == NULL)
    {
        temp = root->llink;
        free(root);
        return temp;
    }
    temp = inordersuccessor(root->rlink);
    root->info = temp->info;
    root->rlink = delete(root->rlink, temp->info)
}
return root;
}
```

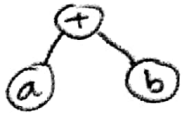
```
NODE inordersuccessor(NODE root)
{
    NODE cur = root;
    while(cur->llink != NULL)
        cur = cur->llink;
    return cur;
}
```

Expression tree

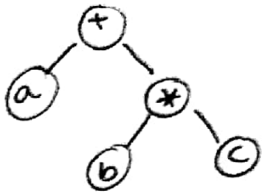
An expression tree is a binary tree wherein all the operands of the expression will be the leaf nodes and the operators in the expn will be non leaf nodes (internal nodes).

Ex:-

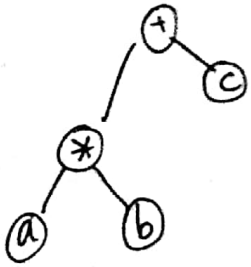
a+b



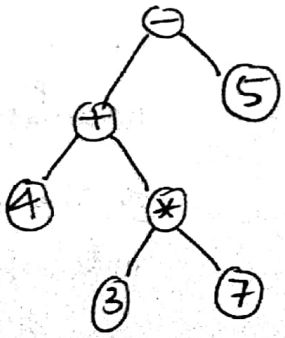
a+b*c



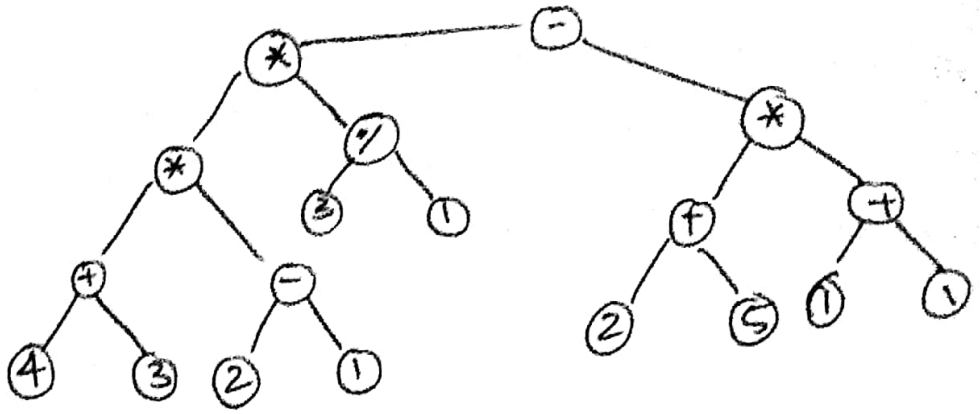
a*b+c



4+3*7-5



$((((4+3)*(2-1))*(3/1))-((2+5)*(1+1)))$



Algorithm to convert given infix expn to an ~~BST~~. Expression tree.

Step 1 : Scan the given infix expression from left to right

Step 2 : Create a node for each scanned symbol
Initialise, 2 stacks namely i) tree stack ii) op stack.

Step 3 : If the scanned symbol is

i) An operand - push it onto the tree stack

ii) An operator - <u>if</u> op stack is empty, push the op node onto op stack,

<u>else</u> - until top of op stack is having (precedence either <u>greater than or equal to</u> the scanned op, pop an op node from op stack and pop 2 nodes from tree stack and assign them as right and left child to the popped node of the op stack and push op node onto ~~tree~~ stack.

push the scanned node onto op stack.

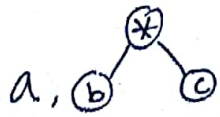
Step 4: Until the op stack becomes empty, pop a node from the op stack and pop 2 nodes from the tree stack., assign as right and left child and push onto tree stack.

Step 5: The root node of the tree stack gives the tree.

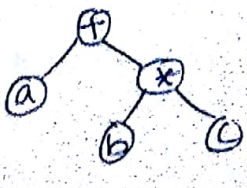
TRACE

Ex :- a+b.*c

symbol	tree stack	op stack
a	a	
+	a	+
b	a b	+
*	a b	+ *
c	a b c	+ *



+



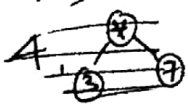
ex 4+3*7-5

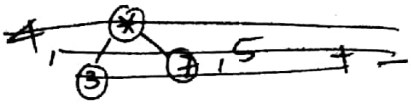
4 +

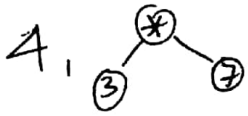
4 3 +

4 3 +*

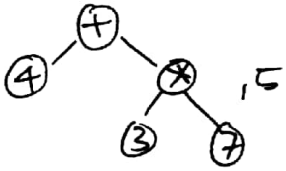
4 3 7 +*



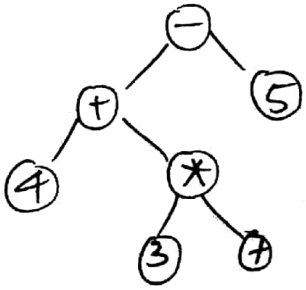




+



–



Program 8

To create an expn tree for given infix expn and traverse the tree in pre, in, post

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
struct node
{
   char info;
   struct node *llink;
   struct node *rlink;
}; typedef struct node *NODE;

// write preorder func - change %d to %c

// write in, post - change.

NODE create node (char item)
{
    NODE temp;
    temp = (NODE) malloc(sizeof(struct node));
    temp -> info = item
    temp -> llink = temp -> rlink = NULL;
    return temp;
}

int preced (char x)
{
    switch (x)
    {
        case ($) : return 3;
        case (*)
        case (/) : return 2;
        case (+) : case(-) : return 1;
```

```
NODE create exp tree (char infix [20])
{
    int i , t1=-1 , t2=-1 ;
    char sym ;
    NODE temp , treestack [15] , opstack [15] , temp1 , r, l;

    for (i=0; infix[i] != NULL; i++)
    {
        sym = infx [i] ;
        temp = create node (sym) ;
        if (isalphanum (sym))
            tree stack [++t1] = temp ;
        else
        {
            if (t2 == -1)
                opstack [++ t2] = temp;
            else
            {
                while ( t2 != -1 &&
                        ( preced ( opstack [t2] → info) ≥
                          preced (sym) ) )
                {
                    temp1 = opstack [t2--] ;
                    r = tree stack [t1--];
                    l = tree stack [t1--];
                    temp1 → rlink = r;
                    temp1 → llink = l;
                    tree stack [++t1] = temp1;
                }
                opstack [++t2] = temp;
            }
        }
    }
}
```

```
while (t2 != -1)
{
    temp1 = opstack [t2--];
    temp1 -> rlink = treestack [t1--];
    temp1 -> llink = treestack [t1--];
    treestack [++t1] = temp1;
}
return treestack [t1];


nt main ()
{
NODE root = NULL;
char infix [15];
printf ("\n Read infix exp\n");
scanf ("%s", infix);
root = createexptree (infix);
printf ("\n The preorder traversele is \n");
preorder (root);
pf (" In");
inorddder (root);
pf (" post");
postorder (root);
}
```

AVL Tree (Adelson Velskoi/ Lendis)

The worst case efficiency of any operation on a BST is O(n) where n is the no of nodes in the binary tree

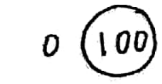
An AVL tree is a BST wherein, any operation in the worst case will take time of O (log n)

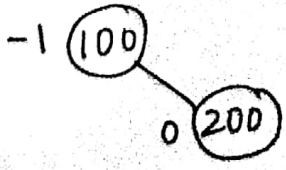
* An AVL tree is a BST wherein, the balance factor of each node in a tree should either be 0, +1 or −1 .

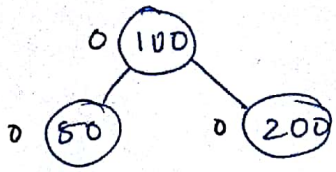
The balance factor of a node is calculated as

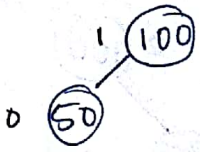
BF = ht of left subtree − ht of right subtree

Eg for AVLT :-







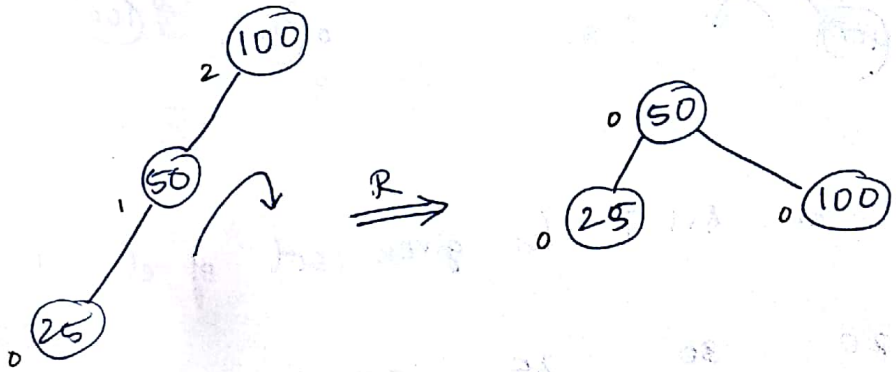


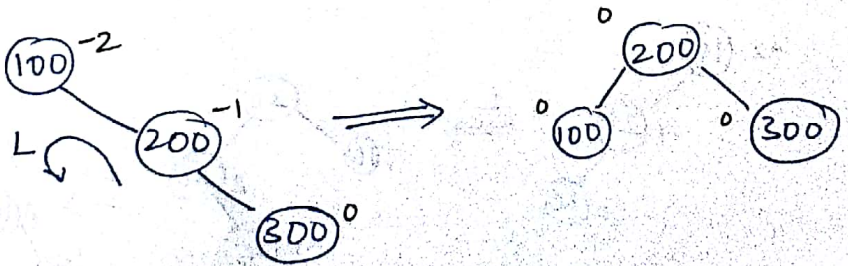
While constructing an AVL tree, to balance the tree, we come across 2 types of rotation opⁿ namely,

i) Single rotation

ii) Double rotation.

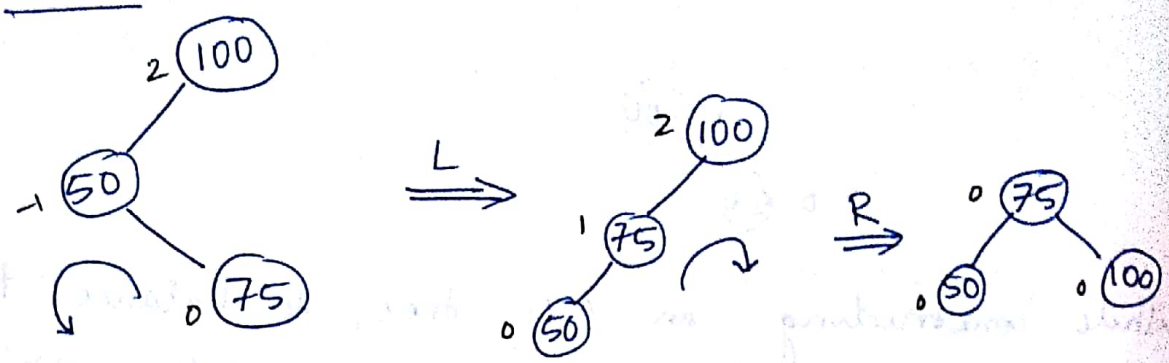
Single Rotation



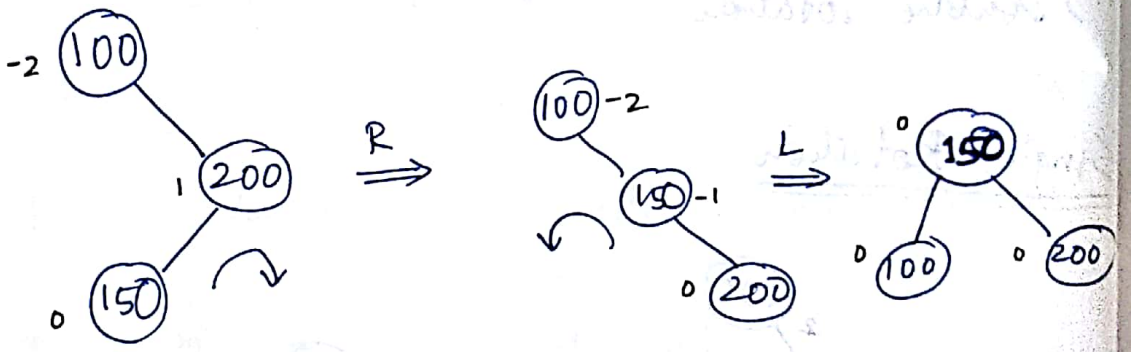


Double Rotation

i) LR rotn

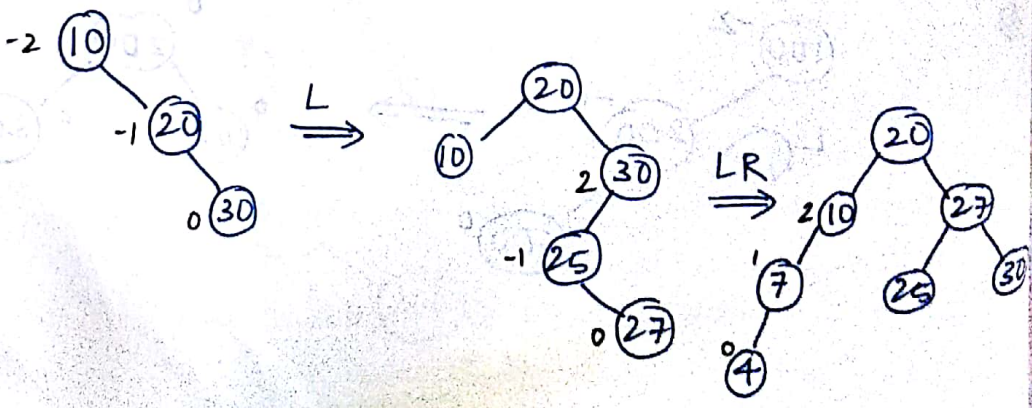


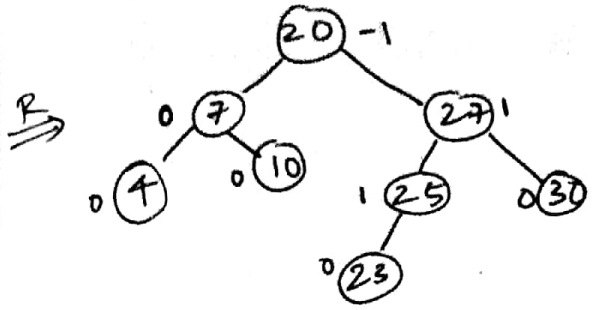
ii) RL rotn



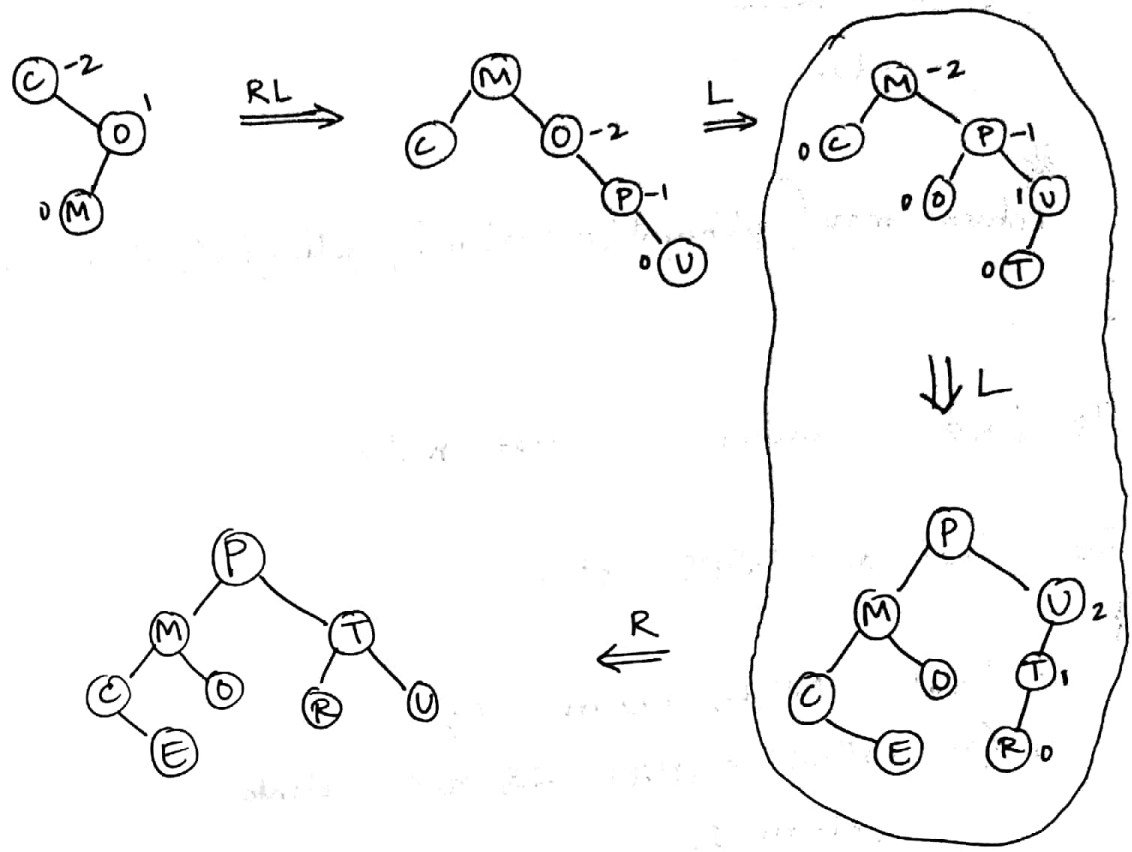
⑧ Construct an AVL T for given set of elements.

10 20 30 25 27 7 4 23





(B) The characters of the word "COMPUTER".



(S) "COMPUTING"

Write a C func to ~~write~~ find height of a tree.

```
int getheight(NODE root)
{
    if (root == NULL)
        return 0;
    return max(getheight(root->llink), getheight(root->rlink)) + 1;
}
```

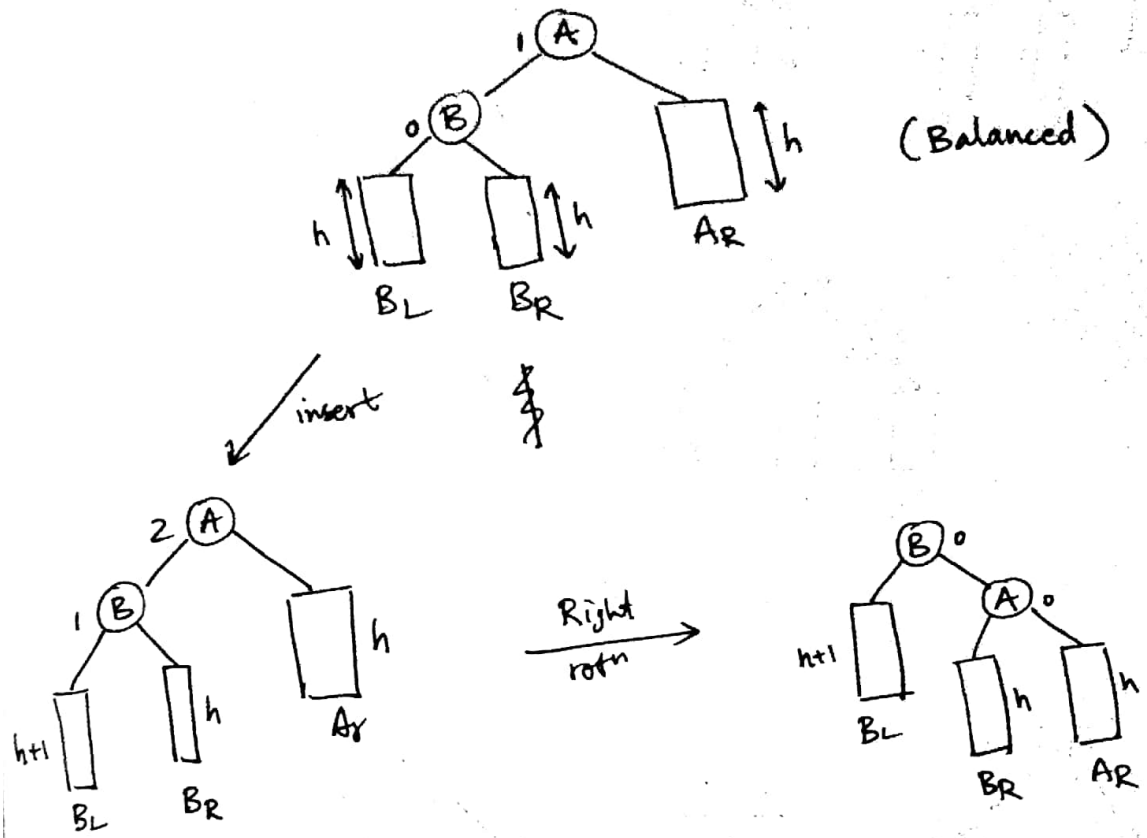
To find number of leaf nodes

```
int leafnode(NODE root)
{
    if (root == NULL) return 0;
    if (root->llink == NULL && root->rlink == NULL)
        return 1;
    return leafnode(root->llink) + leafnode(root->rlink);
```

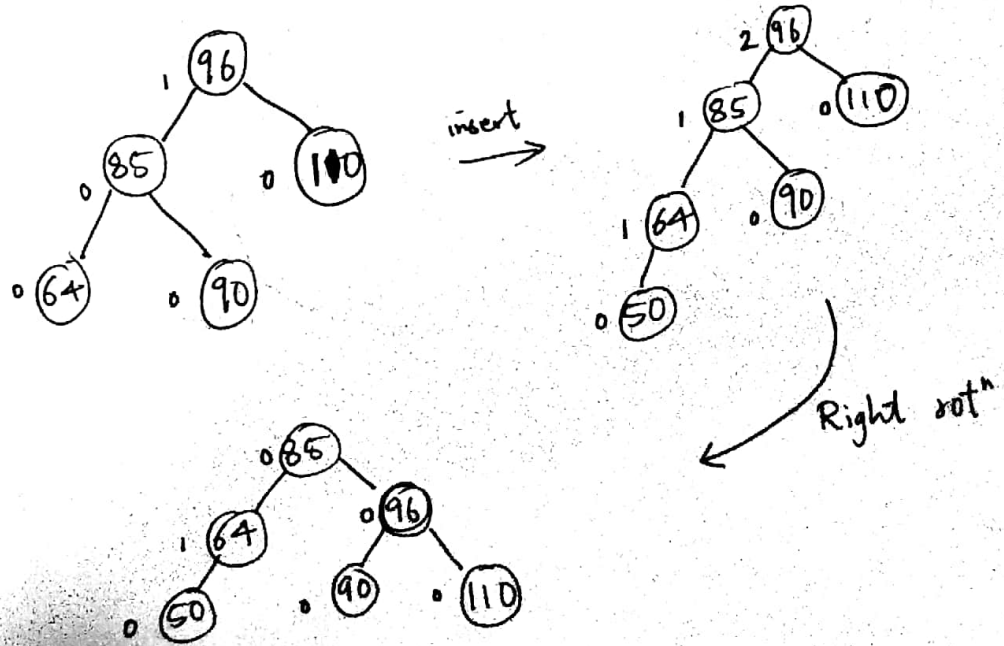
Single Rotation

Right Rotation (LL)

If a new node is inserted in the left subtree of the left subtree of the reference node

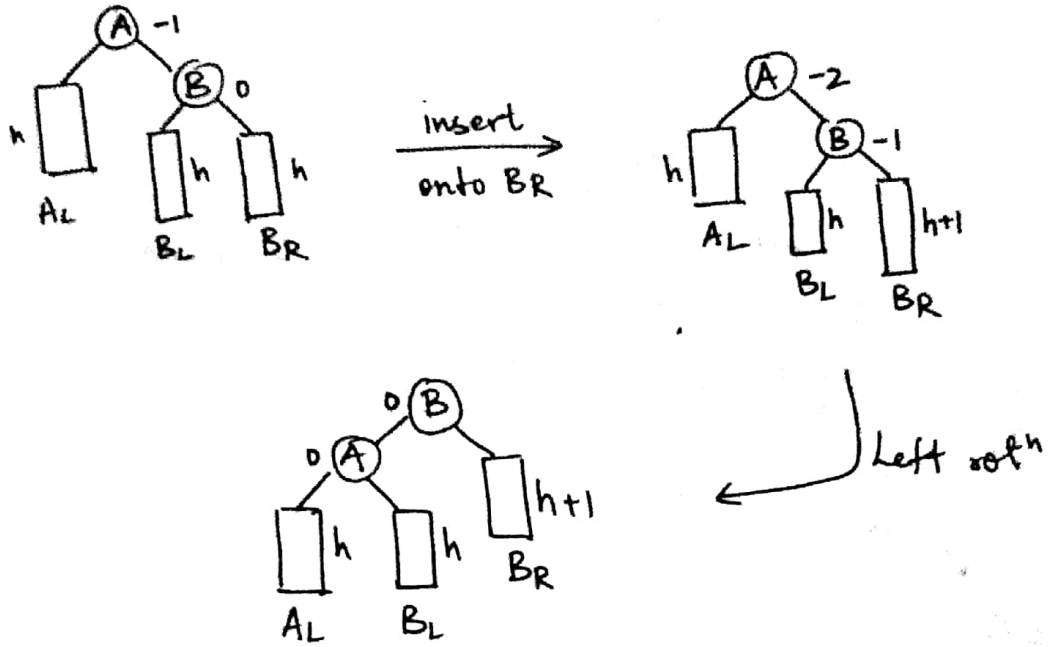


ex:-

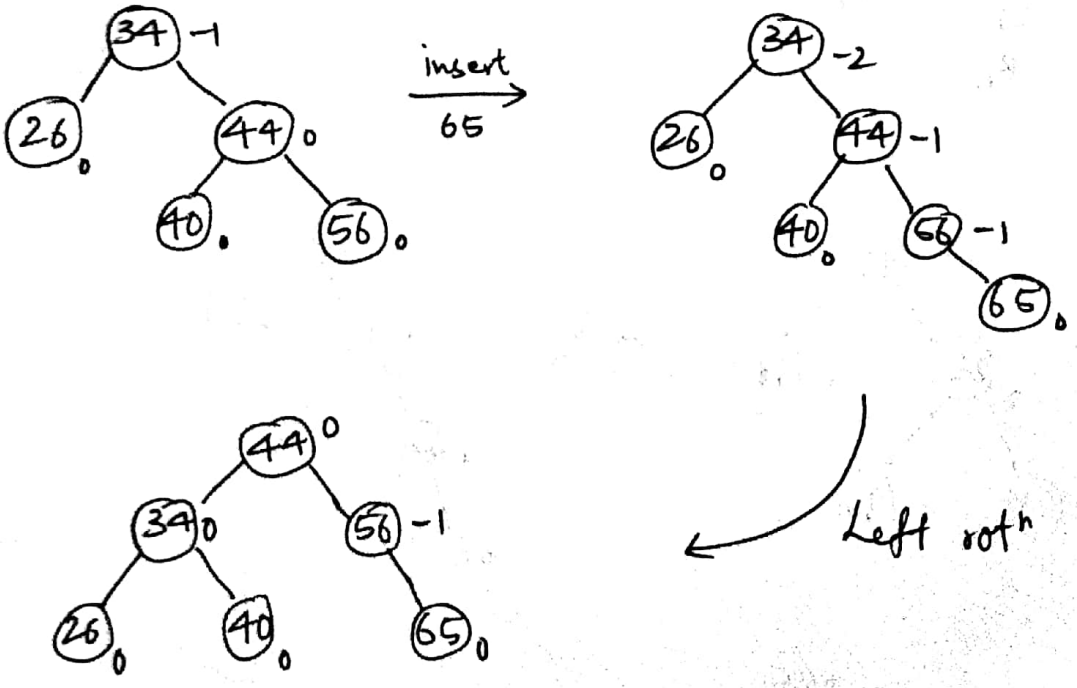


Left Rotation (RR)

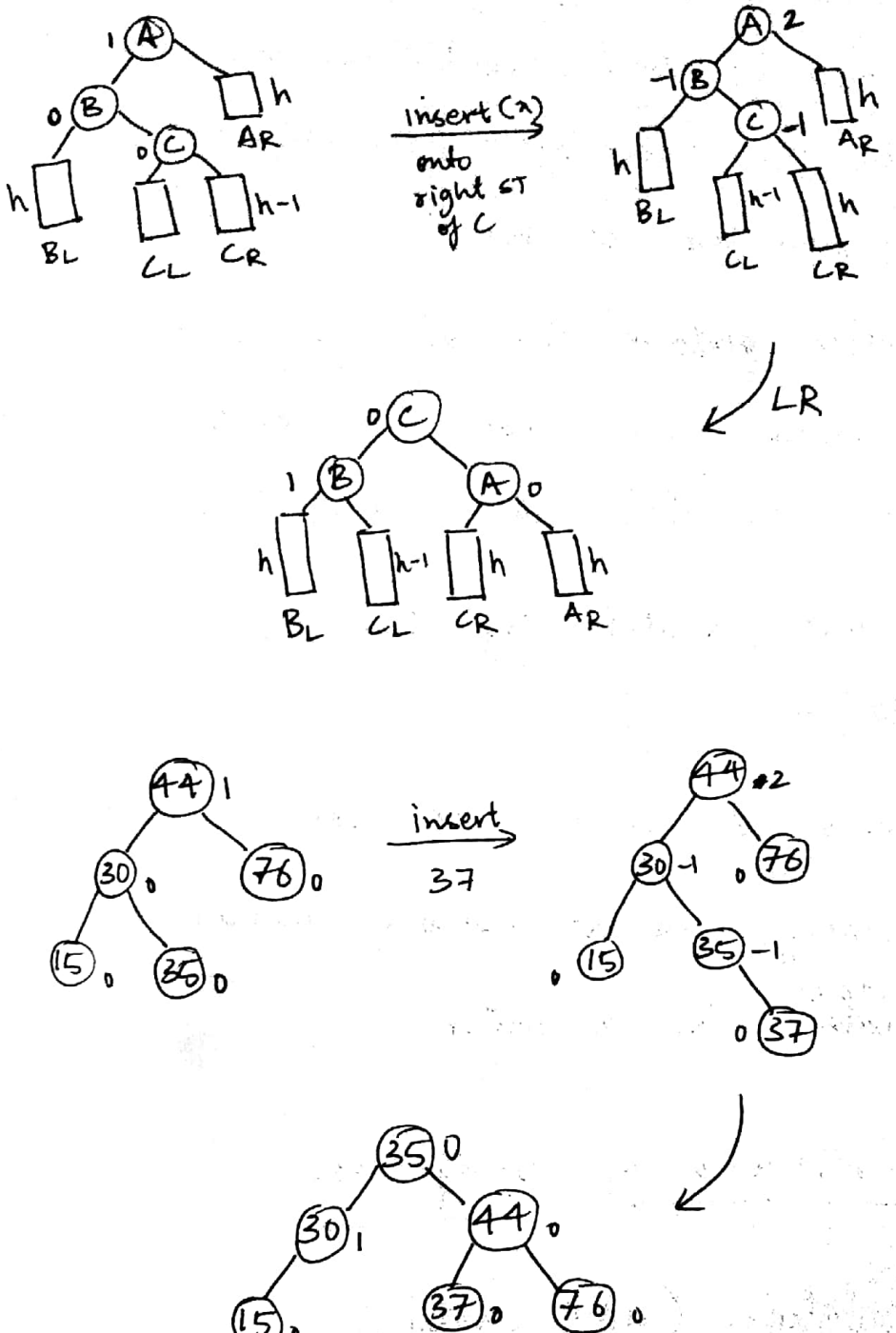
A new node is inserted in the right subtree of the right subtree of the reference node.



eg:-



LR rotation



ex:-

Splay Tree

A splay tree is a self adjusting BST with the additional property that recently accessed elements are quick to access again (recently accessed elements should be promoted as root)

The splay tree performs the basic operation like insertion deletion and search in O(log n) time as compared to O(n) in a BST

Splay tree was invented by Daniel.D.Sleator, and Robert Tarjan

Applications of Splay tree can be found in Cache memory, virtual memory management, the routing tables in a router.

Types of rotations in Splay Tree

1. Zig rotation (right rotation)
2. Zag rotn (left rotn)
3. Zig Zig
4. Zag Zag
5. Zig Zag
6. Zag Zig.

The decision on what rotn to be performed depends on the following.

① Does the node we are trying to rotate have a grand parent?

② Is the node a left or a right child of the parent.

③ Is the parent the left or the right child of the GP.

④ If the node does not have a GP, we carry out the Left rotn if it is the right child of parent, otherwise we carry out right rotation.

⑤ If the node has a GP, then, the following 4 possibilities can happen

i) If the node is the left of parent and parent is left of GP. We need to do zig zig

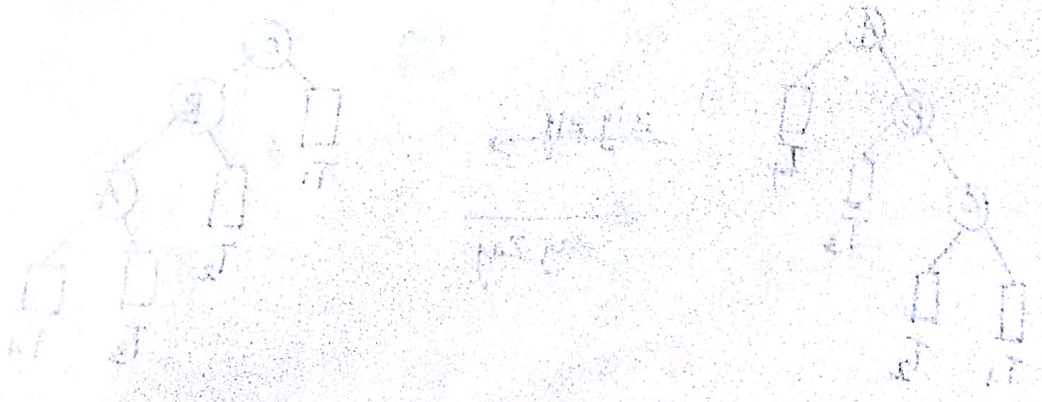
ii) If node is right of P and P is right of GP.

zag zag.

iii) If node is right of P but parent is left of GP. Zag Zig

iv) If node is left of P but P is right of GP

ZigZag



Procedure to perform insertion operation in splay tree

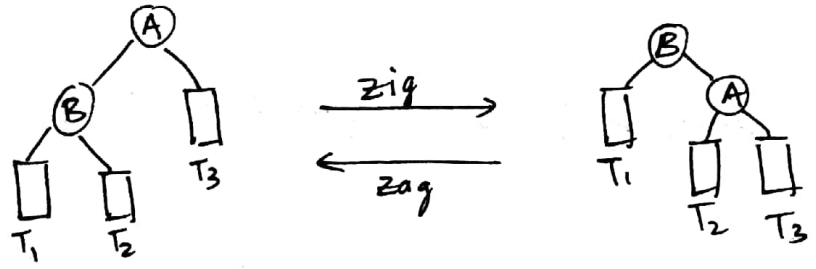
Step 1: Check whether tree is empty

2: If tree is empty, then insert the new node as root and exit.

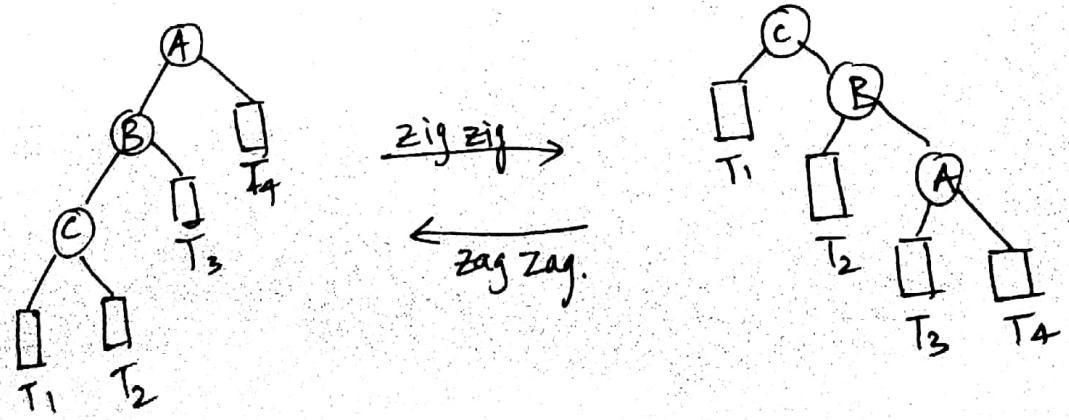
3: If tree is not empty then insert the new node as a leaf node based on the logic of BST

4: After insertion, splay the new node.

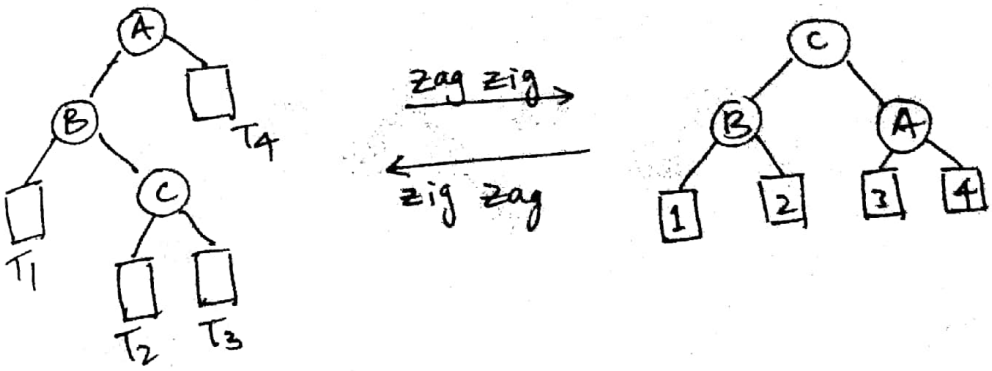
i) zig and zag.



ii) zig zig.

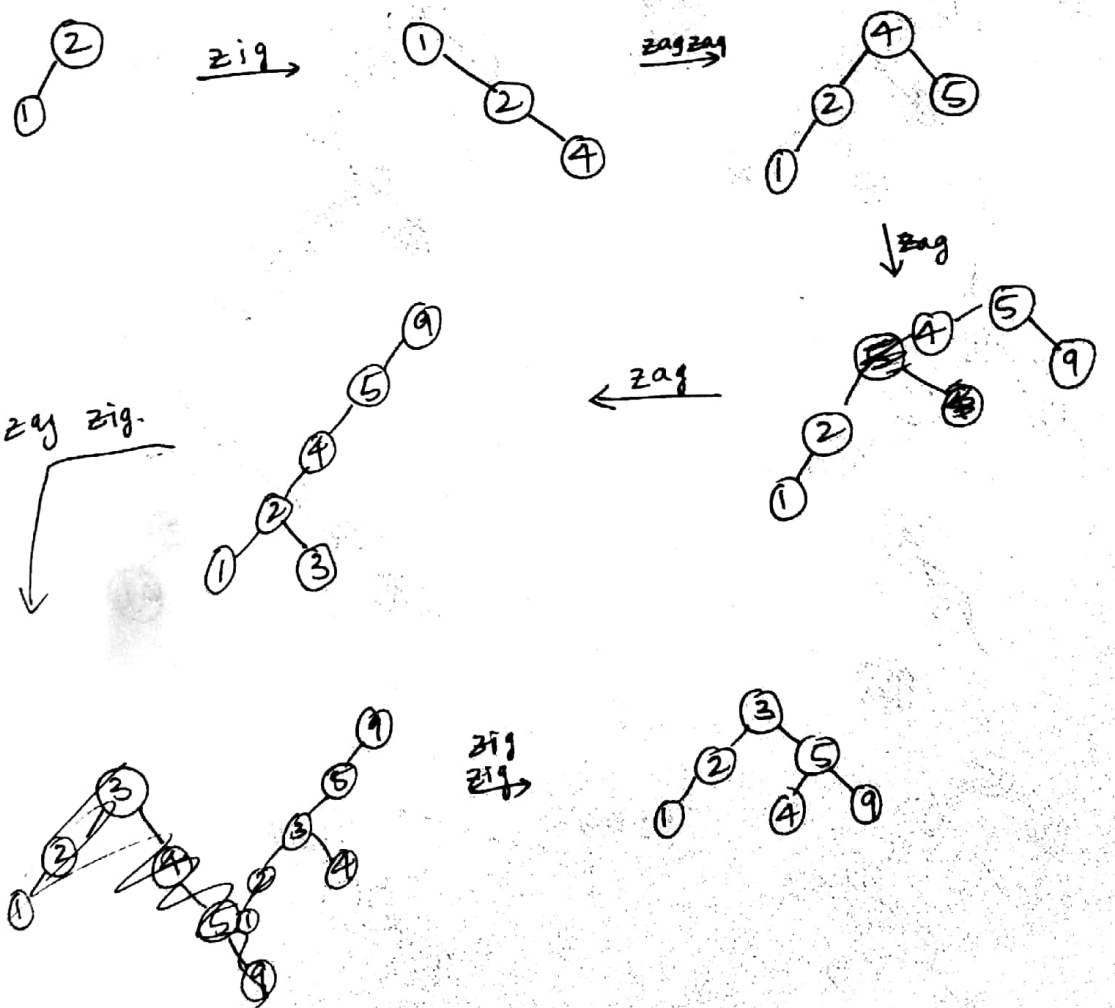


iii) zig zag.

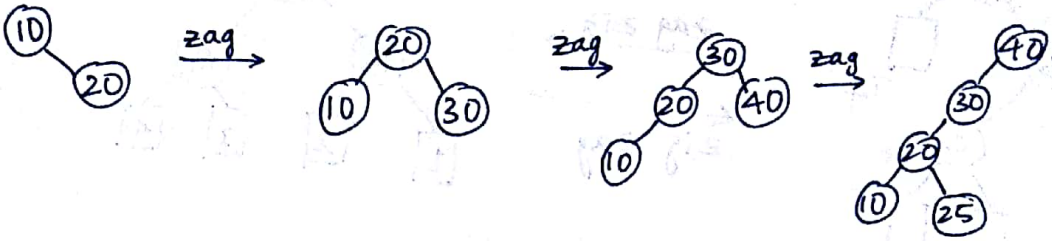


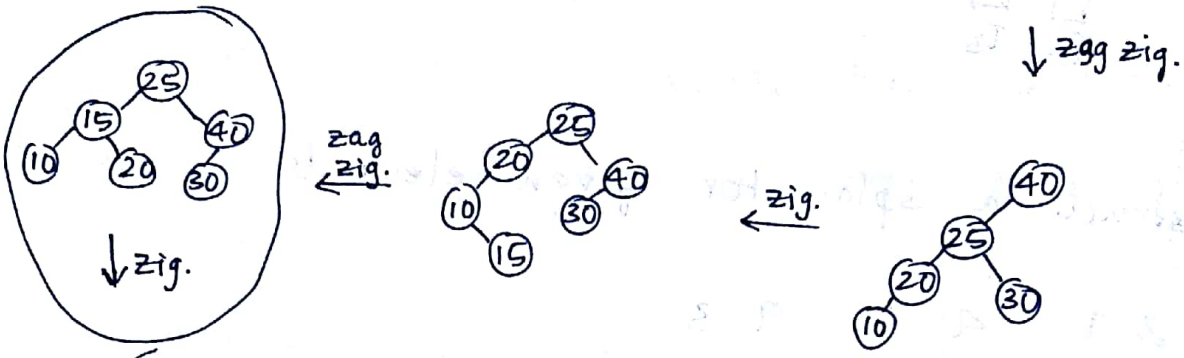
b) Construct a splay for given elements.

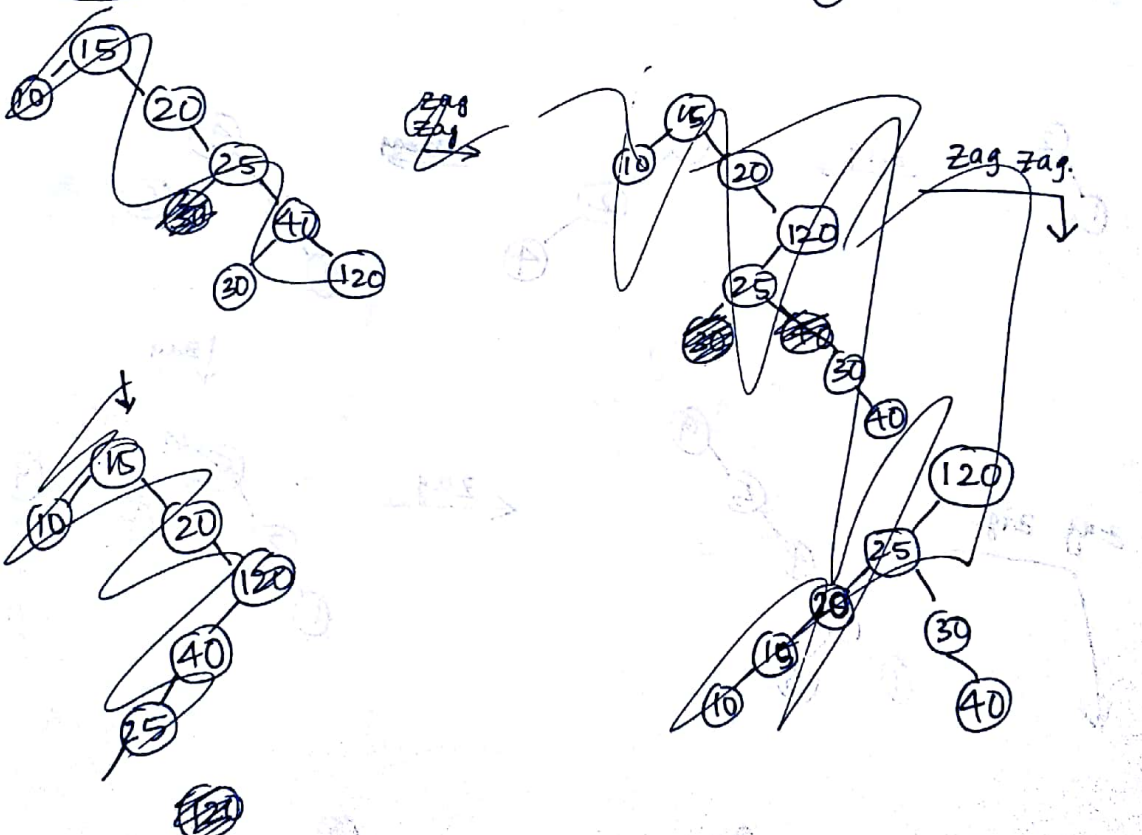
2 1 4 5 9 3

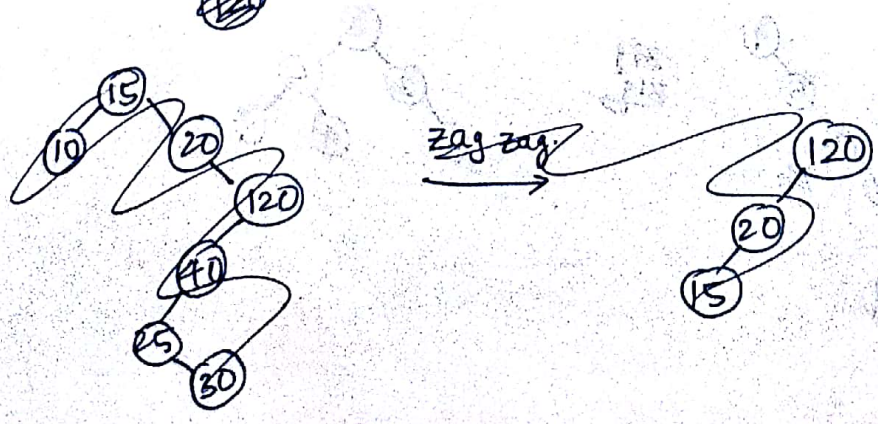


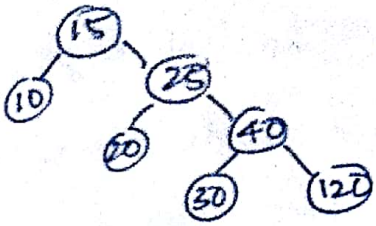
(8) 10 20 30 40 25 15 120



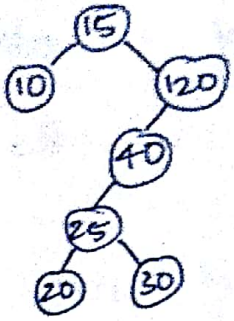




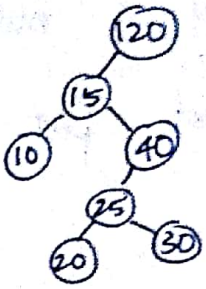


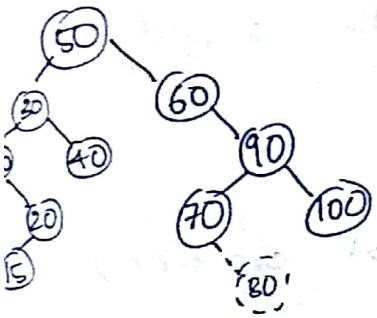


zag zag.

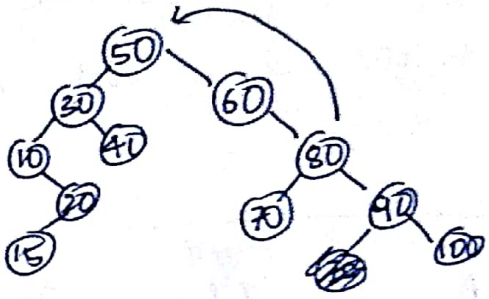


↓ zag.

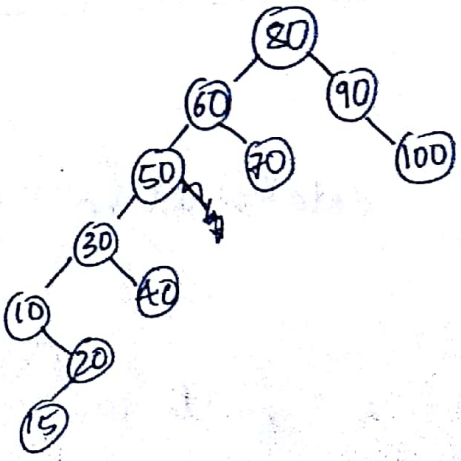




80
zag
zig



↓ zig zag.



The depth of a node is the no of edges from the node to the tree's root node.

The depth of root node is zero.

The height of a node is the number of edges on the longest path from the node to a leaf.

The height of leaf node is zero.
The height of a root node gives the height of the tree.

DMA , LLs , Trees (except exp tree)
for test 2

Trie

Its a data structure to maintain a set of strings.

Trie is a multi way search ~~search~~ tree.

Trie is derived from the word Information Retrieval.

edges from

.

of edges

le to a

height of the

exp tree)

a set of

rch tree.

Information

There are two types of tries

i) Standard trie
ii) Compressed trie.

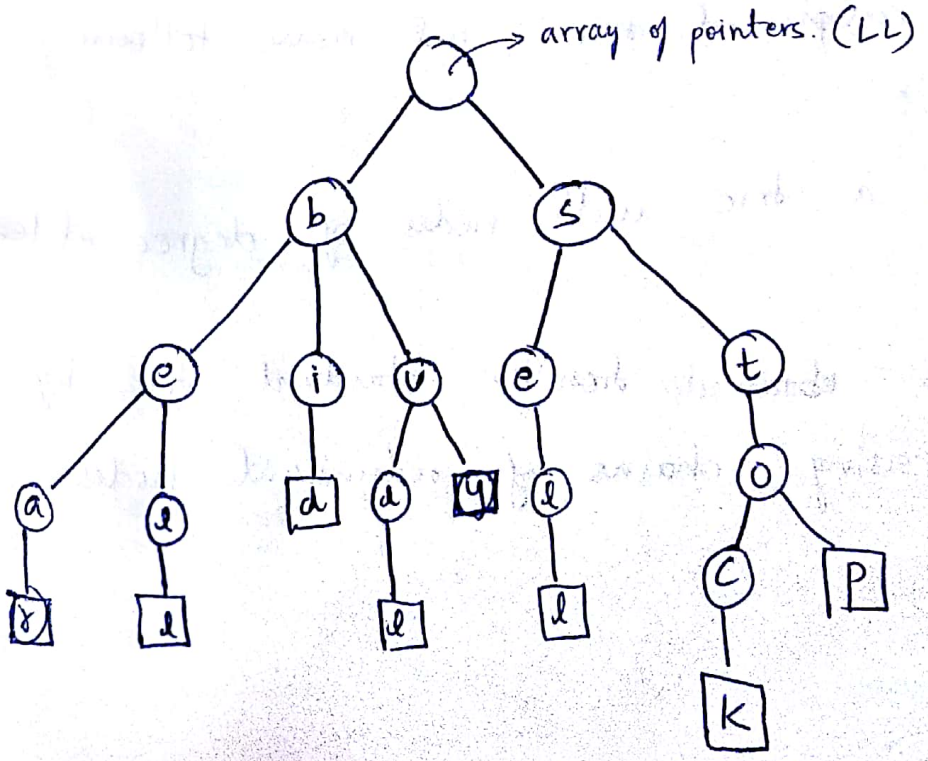
STANDARD

1) A std trie for a set of strings 's' is an ordered tree such that each node apart from root is labelled with a character.

2) The children of a node are alphabetically ordered.

3) The path from the external nodes to the root yeild the string of 's'

ex:- s = { bear, bell, bid, bull, buy, sell, stock, stop }



each node

↓

i) character

ii) array of pointers

iii) pointer pointing to parent.

Time taken to search a string in a trie is ~~order of~~ $O(ns)$

n = size of string

s = no of pointers

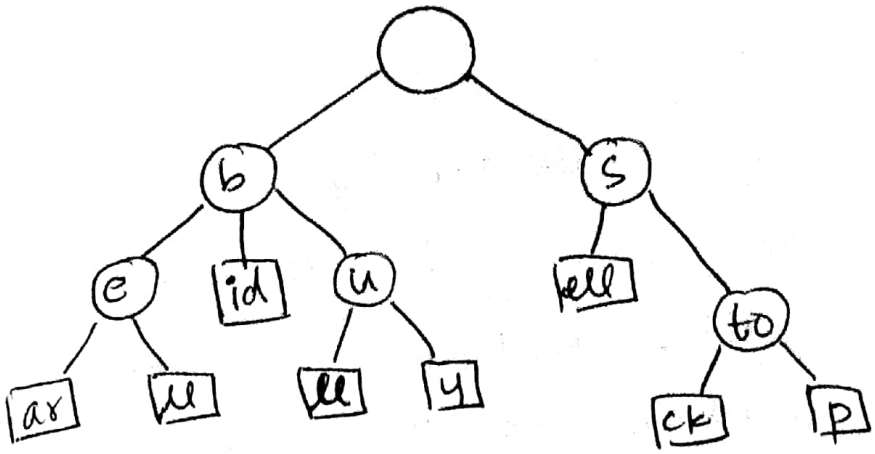
COMPRESSED TRIE

The compressed trie reduces the size of the standard trie.

In a compressed trie, we have following properties :-

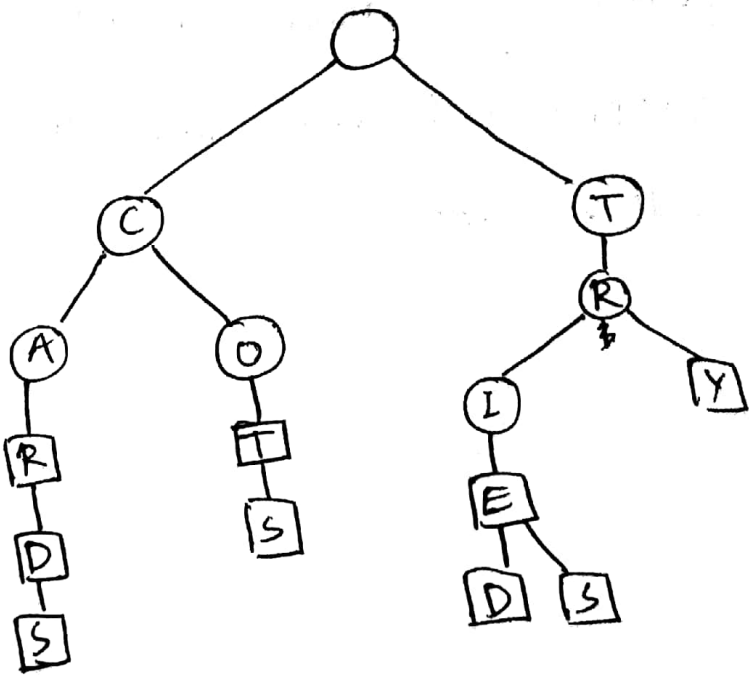
i) it is a trie with nodes of degree at least 2

ii) It is obtained from the standard trie by compressing chains of redundant nodes.

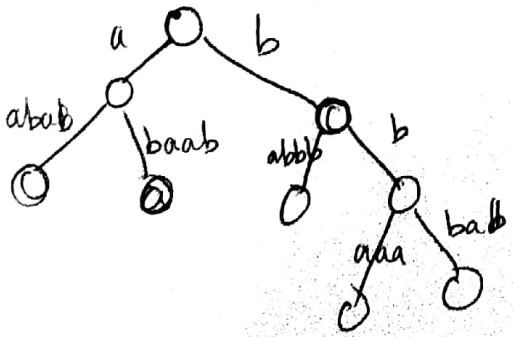


3) Contruct a trie for the given set of words

CAR
CARD
CARDS
COT
COTS
TRIE
TRIED
TRIES
TRY



Insertion operation on a trie



i) To implement a dictionary

ii) Used in the implementation of search engine.

iii) Used in pattern matching - generally in pattern matching algorithm, we do pre processing for the pattern. Using tries, we can preprocess the text itself while performing pattern matching and the preprocessed tree using trie is termed as suffix tree.

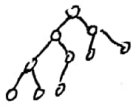
search engine.
enerally in
e do pre
lsing tries,
itself while
d the ~~tree~~
is termed as

UNIT 5

Heap

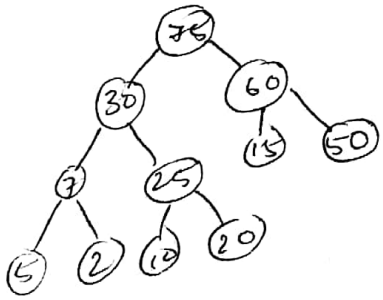
Heap is a Binary tree such that the keys stored in each node should have the following properties.

i) <u>structural property</u> -
The tree should be complete binary tree.
ie., all levels apart from last level should be completely filled and the last level should be filled from left to right.



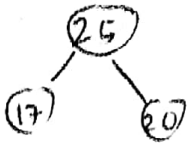
ii) <u>Parent dominant property</u>:

Parent node should always have higher priority when compared to its children

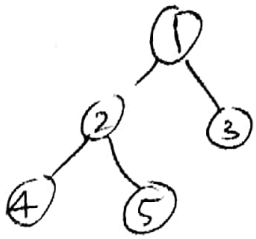


There are 2 types of heaps, namely,

i) Max heap - where the root node should have max element



ii) min heap - the root node should have the min element.



There are to approaches to construct a heap.

i) Bottom up approach

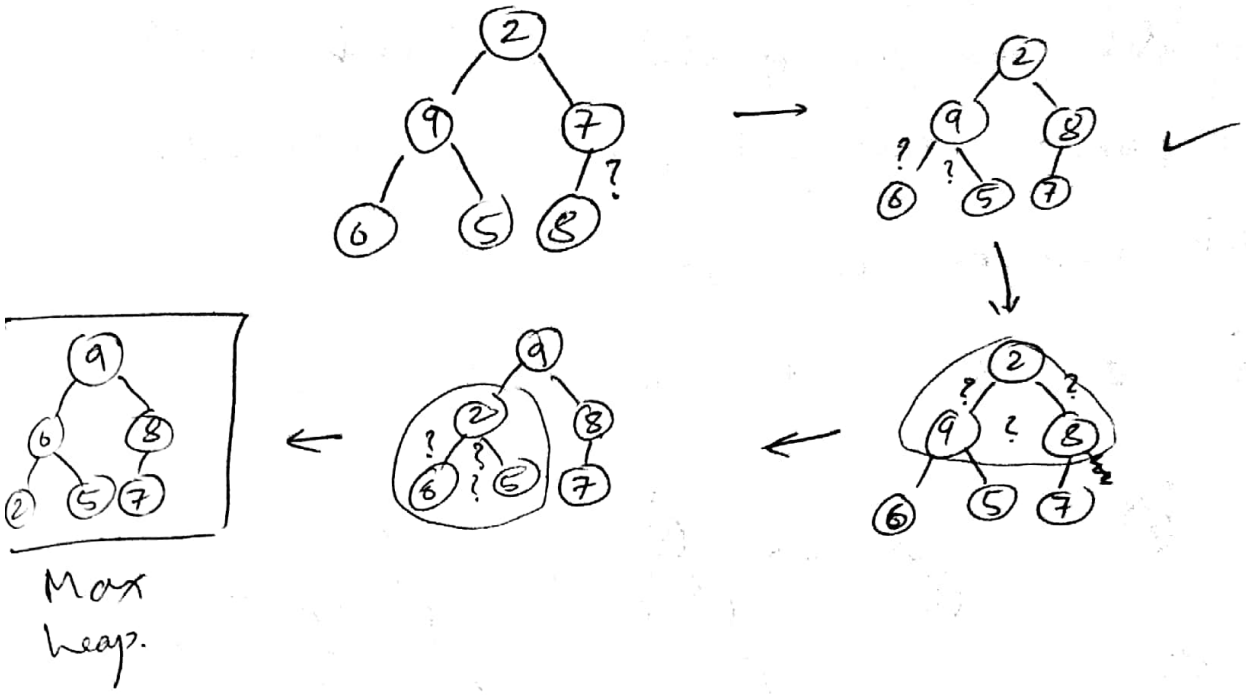
ii) Top down approach.

Bottom Up

Here, for the given set of elements we construct a BT first and then we transform the binary tree to a heap.

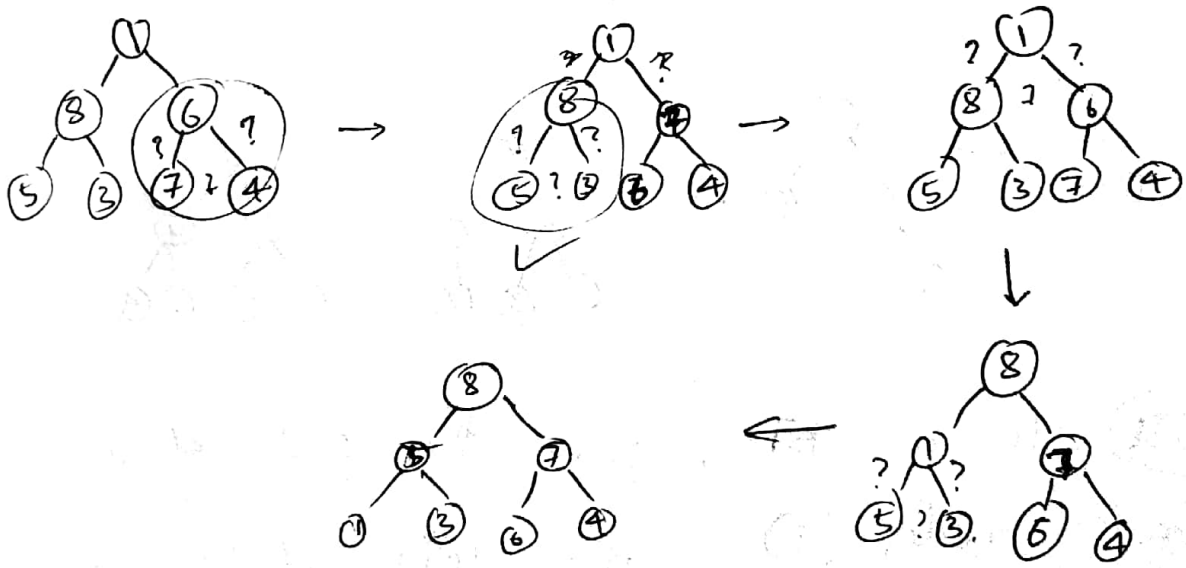
2 9 7 6 5 8

Build max heap.



Max heap.

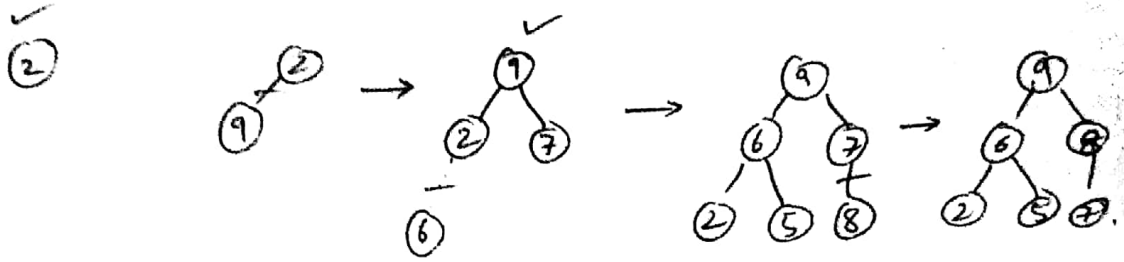
(3) 1 8 6 5 3 7 4.



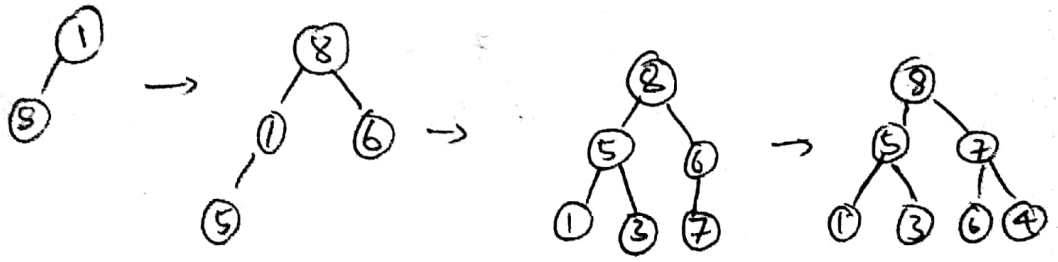
Top Down Approach

In this approach, while creating the BT using repeated insertion operation, we check whether the parent dominant propert is achieved after each insertion or not.

(a) 2 9 7 6 5 8



(b) 1 8 6 5 3 7 4.



NOTE: In a BT, given the posn of a parent node, (i), it's left child posn is denoted with (2i) and right ~~child~~ child is denoted with (2i+1).

Given the position of a child node, (c) it's parent node posn is denoted by

$$P = \lceil (C-1)/2 \rceil$$

Procedure or Algo

to construct a max heap.

O(n) to build heap with n elements.

```
Buildmaxheap (A[ ], n)
// builds max heap for the given set of n elements.
// Input : A set of n elements stored in A.
// Output : An ordered set of elements which
// represents max heap.

for i ← ⌊n/2⌋ to 1
    heapify (A, i, n)


heapify (A, i, n)
//
//
//
left ← 2i
right ← 2i + 1
if (left <= n and A[left] > A[i])
    largest ← left
else
    largest ← i
```

```
if ( right <= n  &&  A[right] > A[largest])
        largest = right.
if (largest != i)
{
        swap (a[i] , a[largest])
        heapify [A, largest , n]
```

~~Procedure to construct delete from max heap.~~

Algorithm to extract max element. (deletion)

```
                        array, rep heap
Algorithm Extract max ( A , n )
// Retrieves max element from the heap.

if n is 0
        heap is empty
else                                    index is from 1 to n
        max = A[1];
        A[1] = A[n];
        n = n-1;
        heapify (A, 1, n);
```

O(1) to extract max element

Program 10

Implement priority Queue using heap.

```
#include<stdio.h>
#include<stdlib.h>

int n;

void swap (int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

void heapify (int a[10], int i)
{
    int left, right, largest;
    left = 2*i;
    right = 2*i+1;
    if (left <= n && a[left] > a[i])
        largest=left;
    else
        largest = i;
    if (right <= n && a[right] > a[largest])
        largest=right;
    if(largest!=i)
    {
        swap (&a[i], &a[largest]);
        heapify(a, largest);
    }
```

```
void buildheap (int a[10])
{
   int i;
   for(i= n/2 ; i >= 1 ; i--)
        heapify(a , i);
}

int extract max (int a [10])
{
    int max;
    if (n==0)
    {
       pf(" Heap is empty ");
       return -1 ;
    }
    else
    {
        max= a[1];
        a[1]= a[n];
        n=n-1;
        heapify (a, 1);
    } return max;
    }
}

int main ()
{
  int a[10]; i , ch
  while(1)
  {
    pf(" Enter choice\n1. create heap\n2. delete\n3. exit\n:");
    sf("%d", &ch);
    switch(ch);
    {
```

```
case 1: pf(" Enter value for n");
        sf("%d", &n);
        pf("Reade elements\n");
        for(i=1 ; i<=n ; i++)
            scanf("%d ", &a[i]);
        buildheap(a);
        printf("Elements after heap are \n");
        for (i=1; i<=n; i++)
            pf("%d\t", a[i]);
        break;
case 2: pf("\n The element retrieved is %d",
                extract max(a));
        pf("Element afte deletion\n");
        for(i=1; i<=n;i++)
            pf("%d\t", a[i]);
        break;

default: exit(0);

}
}
return 0;
```

Hashing

It is the process of mapping the large amount of data item to a smaller table with the help of hash function. ie., hashing lets us create a map where each string (value) is mapped to a unique key by a function known as hashing function

If H denotes the hash function and x is the data, then hash key 'y' can be generated as $y = H(x)$;

If x_1 and x_2 are two datas, we should never have a scenario where in $H(x_1) = H(x_2)$

Almost all hash functions in use are imperfect that means, we might end up in a situation wherein $H(x_1) = H(x_2)$, such a condition is termed as 'collision'

ex:- $H(x) = x \bmod 10$

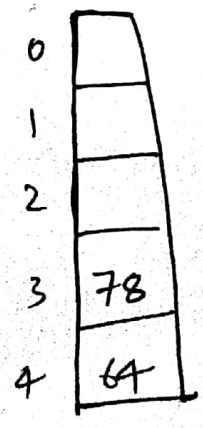
Eg:-

$H(x) = x \bmod 5$

64 78 98 100 12

H(78) = 3

H(98) = 3



Collision in hashing can be avoided using the following methods,

i) Open Adressing (hash table)

ii) Closed Adressing. (seperate chaining)

open <
- linear probing
- quadratic "
- double hashing.

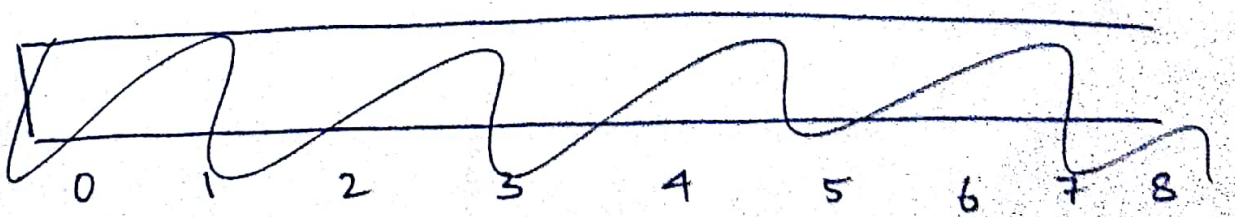
Open Addressing

In open addressing, when a new key collides, find the next empty slot and store the data there.
while searching for the next empty slot, if in case we reach end of table., start searching from the beginning of the table.

∴

0	98
1	100
2	12
3	78
4	64

(b) For the input, 0 1 4 9 16 25 36 49 64 81 100 and $H(x) = x \bmod 7$, construct open hash table.



Closed Addressing (seperate chaining)

all the collided items are added at the end of the list whose header is plugged at the hashed location in the hash table.

62 64 24 77 42 97 22

$H(x) = x \bmod 10.$