Linear	Non-linear
1) Data stored sequentially	Data stored hierarchially
2) All data items are stored in single level	Data stored in multiple level
3) All data can be traversed in a single run	User can't access data in single run
4) Easy to implement	Not easy to implement when compared to linear
5) Memory utilization is not eff	Memory utilization eff
Stack, Queue, list	Trees, graph

Stack

Stack is a linear non-primitive data structure where in insertion and deletion operations are done from the same end. The end used for insertion and deletion is termed as top of the stack

Insertion operation is termed as push and deletion operation is termed as pop

In a stack last element inserted will be the first to be retrieved out or deleted. (LIFO)

Date :
Page No :

Implementation of a stack of integer using an array.

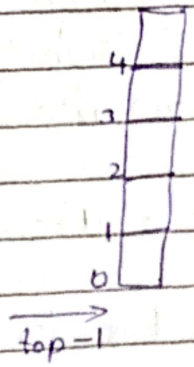
Let stack be an array of integers of size 5 denoted with a macro size.

```
#define SIZE 5
int stack[SIZE]
```

Let top be the index to perform push and pop operation with an initial value of -1.

```
int top=-1;
```

An empty stack looks like



Insertion operation (PUSH)

On a stack which is full we can't perform push operation, if we do so it is called as overflow

If top is having value SIZE-1, it denotes overflow.

If stack is not full we can push an element named item onto the stack using the statements

```
top = top + 1;
stack[top] = item;
```

A C function to perform push operation.

```
void push (int item) {
    if (top == SIZE - 1)
        printf ("\n Stack flo overflow");
    else {
        top = top + 1;
        stack[top] = size;
    }
}
```

Deletion operation [POP]

On a stack which is empty we can't perform pop operation. The condition to check stack is empty.

```
top == -1;
```

If you do so it is termed as underflow.

On a stack which is not empty the pop operation can be performed using statements.

```
delete stack[top];
top = top - 1;
```

A C function to perform POP operation.

```
void pop () {
    if (top == -1) {
        printf ("Stack Underflow");}
```

```
else {
    printf("\n Element popped is %d",
                                stack[top]);
    top=top-1;
  }
```

Display

```
void display () {
   if(top int i;
    if (top==-1) {
    printf ("stack empty");
     }
    else
     { printf (" In stack content are \n");
       for (i=top; i>=0; i++) {
       printf(" %d \n", stack[i]);}
     }
}
```

C program to implement stack of integers.

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

int stack [SIZE];
int top =-1;

// Push function
// Pop function
// display function.
```

```
int main() {
  int ch, item;
  while(1) {
    printf("\n 1.push");
    printf("\n 2. pop");
    printf("\n 3. Display");
    printf("\n 4. Exit");
    printf("\n Read choice:");
    scanf("%d", &ch);
    switch(ch)
    {
    case 1: printf("\n read item to be
                    pushed:")
            scanf("%d", &item);
            push(item);
            break;
    case 2: pop();
            break;
    Case 3: display();
            break;
    default: exit(0);
    }
  }
}
```

C program to implement stack of integers using structures and point

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

struct stack {
    int data[SIZE];
    int top;
};
typedef struct stack STACK;

void push (STACK *S, int item) {
    if (S->top == SIZE-1)
        printf ("\n stack overflow");
    else
    {
        S->top = S->top+1;
        S->data [S->top] = item;
    }
}

void pop (STACK *S) {
    if (S->top == -1)
        printf ("\n stack underflow");
    else
    { printf("\n element popped is %d"
                S->data.-> [ ~~to~~ S->top]);
        S->top = S->top-1;
    }
}
void display (STACK S) {
    if (S.top == -1)
        printf ("\n stack is empty");
    else {
        printf ("\n stack content are\n");
```

```
    for (i=s.top; i>=0; i--) {
        printf("%d\n", s.data[i]);
    }
}

int main() {
    STACK S;
    S.top = -1;
    int ch, item;
    for (;;)
    {
    printf("\n1.push");
        printf("\n2.pop");
        printf("\n3. Display");
        printf("\n4. Exit");
        printf("\n Read choice:");
        scanf("%d", &ch);

switch(ch)
{
    printf("\n
    case 1: printf("\n read item tobe
                    pushed:");
            scanf("%d &item);
            push(&S, item);
            break;
    Case 2: pop(&S)
            break;
    Case 3: display(S);
            break;
    default: exit(0);
}}
    return 0;
}
```

Applications of stack

* Conversion of expression
* Evaluation of postfix expression
* Recursion.

Expr → Prefix
→ Postfix (suffix)
→ Infix

Infix	Postfix	Prefix
a+b	ab+	+ab
a+b*c	abc*+	a+*bc
		+a+a*bc
a+b-c	ab+-c	-a+bc
	ab+c-	-+abc
(a+b)*c	ab+*c	+ab*c
	ab+c*	*+abc

((a+(b-c)*d)^e)+f a+bc- (a+-bc*d)^e+f

(a+*-bcd)^e+f

(+a*-bcd)^e+f

((a+bc-*d)^e+f)

((a+bc-d*)^e+f) ^+a*-bcde+f

(abc-d*+)^e+f +^+a*-bcdef

abc-d*+e^+f

abc-d*+e^f+

Algorithm to convert given infix expression to post fix.

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

Step 2: If scanned symbol is

i) operand - push it on to stack

ii) ~~left parenthesis~~ - place it on post-fix expression

ii) left parenthesis - Push it on to stack.

iii) Right parenthesis - Pop the content of stack and place them on post fix expression until you get corresponding left parenthesis

iv) Operator - if stack is empty or top of stack is left parenthesis push the operator onto stack

else

while top of stack has higher or equal precedance than the scanned symbol

and

stack is not empty

and

top of stack if not left parenthesis

Pop the content of stack and place it on postfix exp

Push the operator onto the stack.

Step 3: Until stack becomes empty pop them and place them on post fix

Convert the following infix to postfix using stack.

a+b*c

Symbol	Stack	Postfix
a	Empty	a
+	+	a
b	+	ab
*	+*	ab
c	+*	abc
	Empty	abc*+

(a+b)* c

Symbol	Stack	Postfix
(	(	Empty
a	(	a
+	(+	a
b	(+	ab
)	Empty	ab+
*	*	ab+
c	*	ab+c
	Empty	ab+c*

Date :
Page No :

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

Symbol	Stack	Postfix
(	(	-
(	((	-
(	(((	-
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+

Q. C function to convert a given infix into postfix.

```
void infix_topostfix (~~char~~ stack *S, char infix[])
{   int i, j=0
    char symbol, postfix[15], temp

    for (i=0; infix[i] != '\0'; i++)
    {   symbol = infix[i];
        if (isalnum(symbol)) {
            postfix[j++] = symbol;
        }
        else {
            switch(symbol) {
            case '(': Push(S, symbol);
                      break;
            case ')': ~~temp = pop~~
                      temp = pop(S);
                      while (temp != '(') {
                          postfix[j++] = temp;
                          temp = pop(S);
                      }
                      break;
            case '+':
            case '-':
            case '*':
            case '^': if (S->top == -1 || S->
                          data[S->top] == '(')
                          push(S, symbol);
                      else {
```

```
while(( (preced(s->data[s->top]) >= preced
                                    (symbol)))
        && s->top != -1  && s->data[s->top]
                                    != '(')

postfix[j++] = pop(s);
 push(s, symbol
}
  break;
}
}
}
while(s->top != -1)
        postfix[j++] = pop(s);

Postfix[j] = '\0';

printf("\n The postfix expression %s\n",
        postfix);

}
```

C program infix to postfix.

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

#define SIZE 15

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

void push (stack STACK, sy

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

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

int preced (char symbol) {
  switch (symbol) {
    case '^' : return 5;
    case '*' :
    case '/' : return 4;
    case '+' :
    case '-' : return 3; }
}
```

// infix to postfix function.

```
int main(){
    STACK S;
    S.top = -1;
    char infix[15];
    printf("Read infix expression: ");
    scanf("%s", infix);
    infix_topostfix(&S, infix);
    return 0;
}
```

Algorithm to convert infix to prefix

Evaluation of postfix expression using stack

Algorithm to evaluate a given postfix exp

1) Scan the expression from left to right
2) If the scanned symbol is
   i) Operand - Push it on to stack
   ii) Operator - Pop 2 operands, assign them to operand 2 and operand 1 respectively and perform the operation
   res = op1 operator op2, push the result on the stack
3) Pop the stack content to get the final result.

Tracing :-

2 3 4 * +

Symbol	Stack	op1	op2	Result
2	2	–	–	–
3	2, 3	–	–	–
4	2, 3, 4	–	–	–
*	2, 12	3	4	3*4=12
+	14	2	12	2+12=14

(3 + 4) */ (6 - 2)

3 4 + * 6 2 -

3 4 + * 6 2 - */

→ 7 6 2 -

7 4 */ 7/4 = 1

3	3	–	–	–
4	3, 4	–	–	–
+	7	3	4	3+4=7
6	7, 6	–	–	–
2	7, 6, 2	–	–	–
-	7, 4	6	2	6-2=4
*/	1.75	7	4	7/4
				1.75

* C function to evaluate a postfix expression

```
float evaluate_postfix(STACK *S, char postfix[])
{   int i
    char symbol,
    float op1, op2, result
    for (i=0; postfix[i] != '\0'; i++)
    {   symbol = postfix[i];
        if (isdigit(symbol))
            push(S, symbol - '0');
```

```
        else {
            op2 = pop(s);
            op1 = pop(s);
            res = operate(op1, op2, symbol);
            push(S, res);
        }
    }
    return pop(s);
}
```

C program to evaluate postfix expression

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#math.include <math.h>
#define SIZE [15]

struct stack {
    float data[SIZE];
    int top;
};
typedef struct stack STACK;

void push (STACK *S, float item) {
    S->data[++(S->top)] = item;
}

float pop (STACK *S) {
    return S->data[(S->top)--];
}
```

```
float operate (float op1, float op2, char symbol)
{   switch (symbol) {
      case '+' : return op1+op2 ;
      case '-' : return op1-op2 ;
      case '*' : return op1 *op2 ;
      case '/' : return op1/op2 ;
      case '^' : return pow (op1, op2);
    }
} //write evaluate_postfix function

int main () {
  char postfix[SIZE]; STACK s ; s.top = -1;
  float result ;
  printf ("\n Read postfix exp\n");
  scanf ("%s", postfix);
  result = evaluate_underscore
  result = evaluate_postfix (&, postfix);
  printf ("\n Result = %f \n", result);
  return 0;
}
```

Algorithm to ~~convert~~ evaluate a ~~postfix~~ prefix expression :

Step 1 : Scan the expression from right to left

Step 2 : If scanned symbol is
  i) operand → push it to stack
  ii) Operator → Pop 2 operands, assign them to op1 and op2 respectively and perform the operation.
  res = op1 operator op2, and push result to the stack

Step 3 : Pop the stack content to get final result

Date :
Page No :

Heap → Dynamic memory
Stack → local static variable
Data → initialized / uninitialized
Text

Recursion :-
A ability of a function to call itself is termed as recursion.

→ Direct
→ Indirect

Property of a recursive function.
- Recursive function should have a base case & terminating cond^n
- If no base case, it leads to overflow
- Each recursive calls, solves smaller instance of the problem.

Factorial of a number.
5! = 120

Recursive definition for factorial

$$Fact(n) = \begin{cases} 1 & ; \text{if } n == 1 \text{ (or) } n == 0 \\ n * fact(n-1) & ; \text{otherwise} \end{cases}$$

```
int fact(int n)
{  IF (n==1 || n==0)
       return 1;
   else
       return n*fact(n-1);
}
```

Diff between recursion and iteration

Recursion	Iteration
* Recursive function stops on encountering base case	stops on encountering terminating cond^n
* Consumes more memory	less memory
* Slower when compared to...	Faster when compared to

To find GCD of two numbers

~~int GCD(int a, int b) {~~

$$Gcd(x,y) = \begin{cases} x & ; y = 0 \\ Gcd(y, x\%y) & ; \text{otherwise} \end{cases}$$

$$Gcd(x,y) = \begin{cases} x \text{ or } y & ; x = y \\ Gcd(x-y, y) & ; x > y \\ Gcd(x, y-x) & ; x < y \end{cases}$$

```
int gcd (int x, int y) {
  if(y == 0)
    return x;
  return gcd(y, x%y); [Tail-end recursion]
}
```

Tail recursion is a recursive function in which the recursive call is the last statement that gets executed in the function

Recursive function to find whether the elements are sorted or not

```
int check_sort (int a[], int n) {
    if (n == 1)
        return 1;
    return a[n-1] < a[n-2] ? 0 : check_sort(a, n-1);
}

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

Product of 2 numbers

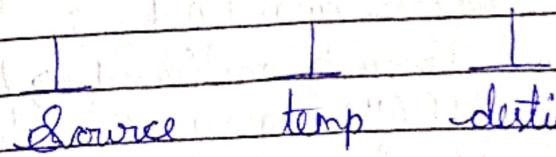
```
int multiply [int a, int b] {
    if (b == 0 || a == 0)
        return 0;
    if (a == 1)
        return b;
    else if (b == 1)
        return a;
```

```
else
    return a + product(a, b-1);
}
```

Binary search

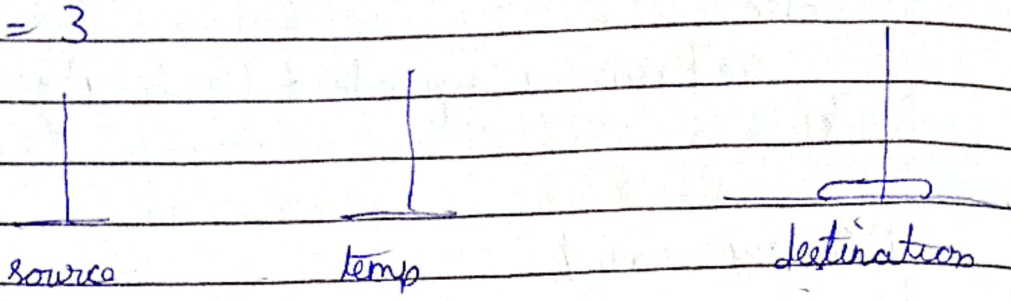
```
int binary (int a[], int n, int key, int low, int high)
{   if (low > high)
        return -1
    else
    { int mid = (low+high)/2
        if (a[mid] == key)
            return mid+1
        else if (a[mid] > key)
            return binary (a, n, key, low, mid)
        return
        else
            return binary (a, n, key, mid+1, high)
```

Tower of hanoi



1) disc is one = no of moves is 1
2) disc is two = no of moves is 3

ii) n = 3



Recursively → move (n-1) disc from source to temp with help of destination → 3, move n th to S→D
Recursively → move (n-1) disc from temp to dest with help of source → 3

$2^n - 1$

C function of tower of hanoi

```
void tower_hanoi (int n, char s, char t, char d)
{
    if (n == 1)
        printf("\n Move %d disc from
            ~~source~~ %c to %c", n, s, d);
    else
    { ~~Tower~~ Hanoi (n-1, s, t, d)
      tower_hanoi (n-1, s, d, t);
      printf ("\n move %d disc from %c to
          %c", n, s, d);
      tower_hanoi (n-1, t, s, d);
    }
}
```

Date :
Page No :

Queue :-

Queue is a linear non-primitive data structure where in the insertion is done from rear end and deletion is done from front end.

The insertion operation is referred as en-que and deletion is refered as deque

In a que, first element inserted will be first to be retrieved out [FIFO]

Types of Queue
* Linear (or) ordinary queue:
* Circular queue:
* Priority queue: (heap - binary tree)
* Dqueue (Double ended queue):

} Syllabus.

Linear Queue:
Implementation of queue of integers:
Let SIZE denotes the size of the queue and the definition of the queue is represented with the following structure

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

The initial value of front and rearend is -1

Insertion: (enque)

```
void enqueue (QUEUE *q, int item)
{   if (q->r == (SIZE-1)
        printf("\n Queue full");
    else
        { q->r = q->r + 1;
          q->data[q->r] = item;
          if (q->f == -1)
              q->f = 0;
        }
}
```

Deque

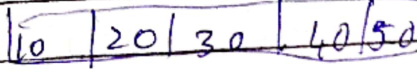
```
int dequeue (QUEUE *q) {
    int del;
    if (q->f == -1) {
        printf("\n Queue empty);
        return -1;
    }
    else {
        del = q->data[q->f];
        if (q->f == q->r) {
            q->f = -1;
            q->r = -1; }
        else {
            q->f = q->f + 1;
        }
        return del
}
```

Display

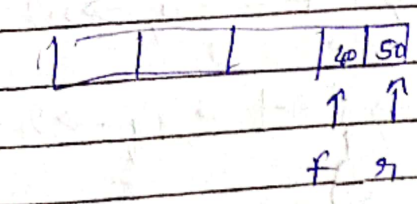
```
void display (QUEUE q) {
  int i
    if(q.f==-1)
      printf("\n queue empty");
    else{
      printf("\n queue data are");
      for(i=0;
      for(i=q.f; i<=q.r; i++){
        printf("%d\t", q.data[i]);
      }
}
```

CIRCULAR QUEUE;

A linear queue with 5 elements looks like



On the above queue, if we perform deletion operation thrice, the queue looks like,



Even though there are some empty slots in the begining of the queue it is not possible to insert new elements as rear end has value of SIZE-1, to overcome this drawback we go for circular queue.

INSERTION:

```
void enqueue (QUEUE *q, int item) {
    if (q->f == (q->r+1) % SIZE)
        printf ("\n Queue full");
    else {
        q->r = (q->r+1) % SIZE;
        q->data[q->r] = item;
        if (q->f == -1)
            q->f = 0;
    }
}
```

DEQUEUE OPERATION:

```
int dequeue (QUEUE *q)
{   int del;
    if (q->f == -1)
    {   printf ("\n queue empty");
        return -1;
    }
    else
    {
        del = q->data[q->f];
        if (q->f == q->r)
        {
            q->f = -1 ; q->r = -1;
        }
        else
            q->f = (q->f +1) % SIZE;
        return del;
    }
}
```

Date :
Page No :

VOID DISPLAY

```
void display (QUEUE q)
{   int i;
    int(q.f == -1)
      printf("\n queue empty");
    else
      {printf("\n queue content are\n");
       for (i=q.f; i != q.r; i=(i+1) %SIZE)
       {  printf("%d\t", q.data[i]);
       printf("%d\t", q.data[i]);
       }
}
```

PROGRAM 3 :

Implementation of msg cubing system

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

#define SIZE 5

struct SIZE {
struct Queue
  {
     int f,r;
     char [SIZE][20];
  };

typedef struct Queue QUEUE;
```

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

char *receiver (QUEUE *q) {
   char *del;
    if (q->f == -1)
    { printf("\n Queue empty");
      return NULL;
    }
   else
    {
      del = q->data[q->f];
      if (q->f == q->r)
        {
          q->f = -1;
          q->r = -1; }
      else
        q->f = (q->f+1) % SIZE;
    } return del;
}
```

```
void display (QUEUE q)
{   int i;
    if (q.f == -1)
        print ("\n Queue empty");
    else
    {
        print ("\n Queue content are \n");
        for (i=q.f; i!=q.r; i=(i+1)%SIZE)
            pF ("%s\n", q.data[i]);
        printf ("%s\n", q.data[i]);
    }
}

int main () {
    QUEUE q;
    q.f = -1;
    q.r = -1;
    char item[20];
    char *d;
    int ch;
    for (;;)
    {
        printf ("\n1.Send\n2.read\n3.display
                4.exit");
        printf ("\n read choice:");
        scanf ("%d", &ch);
        getchar ();

        switch (ch) {
            case 1: printf ("\n read msg to be
                        sent\n");
                    gets (item);
                    sender (&q, item);
                    break;
```

receive

```
case 2 : d = receiver(q);
         if (d != NULL)
             printf("\n msg received is
                 %s\n", d);
         break;
case 3 ; display(q);
         break;

default : exit(0);
  }
}
return 0;
}
```

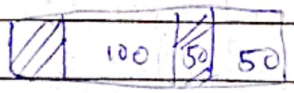
Dynamic memory : (heap)

Allocation of memory during runtime.

UNIX - alloca - allocated dynamic memory from static.

i) malloc()

↳ allocates single block of memory from heap region



malloc(100) success

malloc(150) failure

→ by default will be having some garbage values

On success → return starting address.

On failure → it returns NULL

Syntax

void *malloc(size_t n bytes)

-> malloc function needs one argument

Write syntax for following to allocate memory dynamically

1) 1 integer:

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

2) n integer:

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

3) n students where a student is represented by a structure struct student

```
struct student *ptr;
ptr = (struct student *) malloc (n * sizeof(struct student));
```

Program to add n elements using n elements using dynamic allocation.

```
int main() {
  int *ptr, n, sum=0, i=0;
```

```
printf("Read 'n' value");
scanf("%d", &n);
ptr = (int *) malloc (n * sizeof(int));
printf("\n Read elements \n");
for(i=0; i<n; i++)
    { scanf("%d", ptr+i);
      sum = sum + *(ptr+i)
    }
    print sum;
}
```

Calloc

allocates memory dynamically with multiple blocks on success returns static address and on failure returns NULL

Syntax

~~void *malloc~~

void *calloc (int nblocks, size_t nbytes)

nblocks → number of blocks

nbytes → size of each block

Calloc function has 2 arguements, by default a value of 0 will be stored in the calloc

Realloc ();

Can be used to either increase or decrease the memory allocated previously either by malloc or calloc.

Syntax
void *realloc (void *ptr, size_t newsize)

free();
It deallocates the memory which was allocated dynamically
void free (void *ptr);

realloc (ptr, 0)

realloc (ptr, 0);

Linked list :
→ collection of sequential data items.
↓
referred as "nodes"

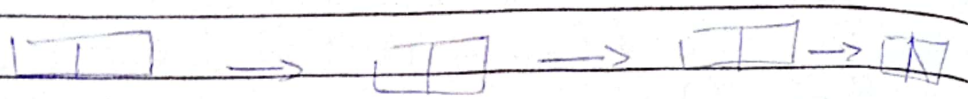
A node of a linked list has 2 fields (parts)
- data → stores the information
- address → address of next node

Representation of a node.

```
struct node {
    int data;
    struct node *addr;
};
typedef struct node *Node;
```

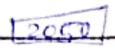
Types of linked list.

* Singly linked list.



reference → start → holds address of the first node.

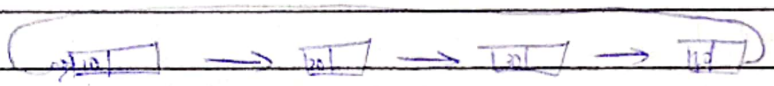
start



start is NULL - list is empty

start → addr is NULL - only one node.

* Circular single linked list.



reference node → last → ~~holds the address of last node~~ refers last node.

last is NULL - list is empty

last → address is pointing to last → one node.

iii) Double linked list.

Reference - start → holds the address of first node

start is NULL - list is empty

start of next is null → only one node

Single linked list:

To insert a new node at beginning of the list.

```
NODE *insertbegin (NODE start, int item) {
    NODE temp;
    temp = (NODE) malloc (sizeof (struct node));
    temp -> data = item;
    temp -> address = NULL;
    if (start == NULL)
        return temp;
    temp -> address = start;
        return temp
}
```

To insert a new node at end of the list.

```
NODE insertend (NODE start, int item) {
    NODE temp, cur
    temp = (NODE) malloc (sizeof (struct node));
    temp -> data = item;
    temp -> address = NULL;
    if (start == NULL)
        return temp;
    cur = start
    while (cur -> addr != NULL)
        cur = cur -> address;
        cur -> address = temp;
        return start;
}
```

To delete a node at the beginning of the list.

```
NODE deletebegin (NODE start)
{
    if (start = NULL) {
    printf ("\n list is empty");
    return NULL;
    }
    temp = start;
    start = start -> addr;
    printf ("\n Node del is %d", temp -> data);
    free (temp);
    return start;
}
```

To delete node at the end of the list

```
NODE deleteend (NODE start) {
  NODE
    if (start = = NULL) {
    printf ("\n listis MT");
    return NULL;
    }
    prev = NULL
    cur = start;
    while (cur -> addr != NULL)
    { prev = cur;
      cur = (cur -> addr;
    }
    prev -> address = NULL
    printF (" \n Node deleted is %d",
                cur -> data);
```

Free(cur);
return start;
}

Display content of the list.

```
void display (NODE start)
{  NODE=temp;
   if (start==-NULL){
    printf(" list is empty"); }
   else {
    printf("\n list data are \n");
    temp=start;
    while (temp!=NULL)
    {
      printf("%d \t", temp->data);
      temp=temp->address;
    }
  }
}
```

Stack → insert begin, delete begin, display
→ insertend, delete end, display

Queue → insertend, delete begin display

Multiplication of polynomial using linked list

(4)

Date :
Page No :

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

struct node {
    int co,po;
    struct node *addr;
};

typedef struct node *NODE;

//Write insert end function.
//Write display function.
//Write add term function.
//Write multiply function.

int main() {
  int m,n,i,j,co,po
  NODE poly1=NULL, poly2=NULL, poly=NULL;
  printf("Read number of term for 1st poly\n");
  scanf("%d",&m);
  printf("Read number of term for 2nd poly\n");
  scanf("%d",&n);
  for(i=0;i<=m;i++)
  { printf("\n\t Read CO and PO for %d term:");
    scanf("%d %d",&co,&po);
    poly1 = insertend(poly1,co,po);
  }
printf("\n First polynomial is\n");
 display(poly1);

printf("Read number of terms for second p
 scanf("%d",&n);
```

```
for (i=1 ; i<=n ;i++) {
    printf ("\n\t Read CO and PO for %d term:");
    scanf ("%d %d", &CO, &PO);
    poly2 = insertend (poly2, CO, PO);
}
printf ("\n First second polynomial is \n");
display (poly2);
poly = multiply (poly1, poly2);
printf ("\n The resultant polynomial is \n");
display (poly);
return 0;
}

NODE insertend (NODE start, int CO, int PO) {
    NODE temp, current;
    temp = (NODE) malloc (sizeof (struct node));
    temp -> CO = CO;
    temp -> PO = PO;
    temp -> addr = NULL;
    if (start == NULL)
        return temp;
    else {
        current = start;
        while (current -> addr != NULL)
            current = current -> address
        current -> addr = temp
        return start;
    }
}
```

```
void display(NODE start){
    NODE temp;
    if(start==NULL)
        printf("\nList empty");
    else
    { temp=start;
      while(temp->addr!=NULL)
      { printf("%d *x^%d + ", temp->co,
        temp->po);
        temp=temp->addr;
      }
      printf("%d *x^%d", temp->co, temp->po);
    }
}

NODE multiply(NODE poly1, NODE poly2){
    NODE first, sec, res=NULL;
    for(first=poly1; first!=NULL; first=fir
        for(sec=poly2; sec!=NULL; sec=sec->
        { res=addterm(res, first->co*second->
                      first->co
          res=addterm(res, first->co*sec->co,
                      first->po+sec->po);
        }
    return result;
}

NODE addterm(NODE start, int co, int po){
  NODE temp,current;
  int flag=0
  temp=(NODE) malloc(sizeof(struct node));
  temp->co=co;
  temp->po=po;
```

(insertion)

```
if (start == NULL)
        return temp ;
else { current = start;
    while (current != NULL)
        {
            if (current -> PO == PO) {
            current -> CO = Cur -> CO + CO
            flag = 1;
            return start;
            }
            current = current -> addr.
        }
if ( flag == 0)
        start = insertend (start, CO, PO);
    return start;
}
```

C function to create a single linked list in ascending ordered list.

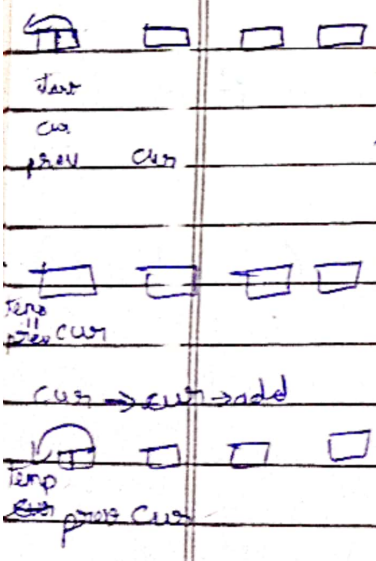
```
NODE orderlist(NODE start, int item) {
    NODE temp,
    {
    temp = (NODE) malloc (sizeof(struct node));
    temp -> data = item;
    temp -> addr = NULL;
      if (start == NULL)
            return temp;
      if ( item <= start -> data)
        {
          temp -> addr = start
          return temp;
        }
```

A B Diff borrow

```
prev = NULL;
cur = start;
while (cur != NULL && item > cur->data)
{
    prev = cur;
    cur = cur -> address
}
prev -> address = temp;
temp -> address = current;
return start;
}
```

**** C function to reverse single list (Non recursive)

```
NODE reverse (NODE start) {
    NODE cur, prev, temp;
    if (start -> address == NULL)
        return start;
    temp = NULL;
    prev = NULL;
    cur = start
    while (cur != NULL)
        temp = prev;
        prev = cur;
        cur = cur -> addr
        prev -> addr = temp;
    }
    return prev;
}
```



curr = start1

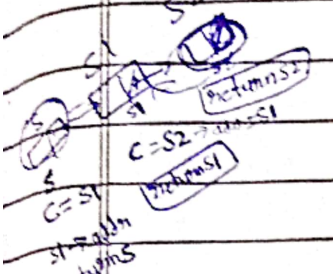
start prev

curr

curr -> address

Reverse linked list using recursion

```
NODE reverse_rec (NODE start)
    NODE cur, prev, temp

    if (start -> address == NULL)
        return start
    curr=temp = NULL   (reverse_rec (start -> address))
    prev=NULL          prev = start
    cur = start        curr -> address = prevstart
```



To count the number of nodes in a simple linked list

C function to delete a node based on info[key]

```
NODE deletenode (NODE start, int key) {
    NODE temp, cur, prev
    if (key == start -> data)
        temp = start;
        start = start -> addr;
    printf ("\n Node %d is deleted", key);
    free (temp); }
```

```
prev = NULL;
cur = start;
while (cur != NULL && key != curr->data)
{
    previous =
    prev = curr;
    cur = cur->addr;
}
if (cur != NULL)
{
    print ("\n key not found");
    return start;
}
prev->addr = cur->addr;
printf ("\n NODE %d is deleted", cur->data);
free (cur);
return start;
}
```

C function to delete a node based on position

C function based on position insertion

```
NODE insertposition (NODE start, int key, int pos)
{
    NODE temp, prev, curr
    int count=1;

    temp = (NODE) malloc (sizeof(struct node));
    temp->data=item;
    temp->addr = NULL;
    if (start = NULL)
        return temp;
```

```
if(pos == -1)
{
    temp -> addr = start;
    return temp;
}
prev = NULL
cur = start;
while ( cur != NULL && count != pos)
{
    prev = cur;
    cur = cur -> addr;
    count ++ ;
}
if ( cur == NULL && pos != ct)
{ printf ("\n Invalid pos");
    return start;
}
prev -> addr = temp;
temp -> addr = curr;
return start;
}
```

The representation of a node in doubly linked list is as follows.

```
struct node {
  int data;
  struct node *prev;
  struct node *next;
};
typedef struct node *NODE;
```

start == NULL // list is empty
start -> next == NULL // list is having only one node

Function to insert node at beginning.

```
NODE insertbegin(NODE struct, int item){
        NODE temp;
        temp = (NODE) malloc(sizeof(struct node));
        temp->data=item;
        temp->prev=NULL;
        temp->next=NULL;
        if (start==NULL)
lab);       return temp;
        temp->next=start;
        start->prev=temp;
        return temp; }
```

Function to insertend

```
NODE    insertend(NODE struct, int item) {
        NODE temp, cur
          temp->data=item;
          temp->prev = NULL;
          temp->next = NULL;
         if (start==NULL)
            return temp;
          cur=start;        next
          while (cur->address != NULL)
              cur = cur->next

          cur->next = temp;
          temp->prev = cur;
          return start; }
```

Function to display content of double linked list.

```
void display (NODE start) {
  NODE temp;
    if (start == NULL)
        printf("\n list empty");
    else {
        printf("\n list content are \n");
        temp = start
        while (temp != NULL) {
          printf("\t %d", temp->data);
          temp = temp->next; } }
  }
```

Function to delete node at end

```
NODE deleteend (NODE start) {
  NODE temp, cur, prev;
    temp = start;
    if (temp == NULL)
       printf("\n list empty"); return NULL;
    else {
                          ----------> if (start->next = NULL) {
    cur = start;                        printf("Element deleted is %d",
                                        free(start);
                                        return NULL; }
    prev->next = cur
     while (cur->start != NULL) {
         cur = cur->start;
        prev->next = cur; }
    free (cur);
    prev->next = NULL;
    return start;
    }
```

To deletenode in a singly linked list whose address is stored in ptr

```
void delete (NODE ptr)
    {
        temp = ptr->addr;
        ptr->ddata = temp->data;
        ptr->addr = temp->addr;
         Free (temp);
    }
```

Sparse matrix.
It is a ~~matrix~~ dense matrix when majority of the element is 0.

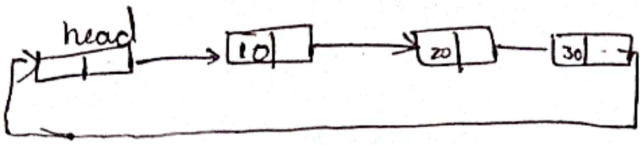
Problem to rep a sparse matrix with DLL

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

struct node {
    int col, row, data;
    struct node *prev;
    struct node *start;
};
typedef struct node *NODE;

~~struct in~~ NODE insertend (NODE start, int row, int col, int item){
        NODE temp;
        temp = (NODE)malloc(sizeof(struct node));
        temp -> data = item
        temp -> row = row
        temp -> col = col
        /* remaining same code */;
}

void display (NODE start) {
    NODE temp;
      temp=start;
    if (start ==NULL) {
      printf("\nlist kaali");
    else {
        printf("\n Row\t Col\t Value\n");
        while ( temp!=NULL) {
          printf(" %d\t %d \t %d\n", temp->row, temp->col,
                                                temp->data);

          temp = temp->next;
    }
```

```
d displaymatrix (NODE start, int m, int n)
{
NODE temp;
  int i,j=0;
 temp=start;
  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->data);
             temp=temp->next;
          }
           else
             printf("0\t");
      }
     printf("\n");
   }
}
```

Header node is a special node in a list where in the address part of the header node which holds the address of first node of list and data part of header node will hold information about the list.

no of nodes.



Write a C function to insert a node at the beginning on a S.L.L with a header node.

```
NODE insertbegin (NODE head, int item) {
    NODE temp;
    temp = (NODE) malloc (sizeof (struct node));
    temp->data = item;
    temp->addr = NULL;
    if (head->addr == NULL) {
      head->addr = temp;
      return head; }
    temp -> addr = head->addr;
    head->addr = temp;
    return head;
    }
```

Insert a node at end in C.LL with head node.

```
NODE insertend (NODE head, int item)
{   NODE temp; cur;
    temp = (NODE) malloc (sizeof (struct node));
    temp->data = item
    if (head -> addr == head) {
     temp -> addr = head;
     head->addr = temp;
```

```
    return head; }
    cur = head->addr;
    while (cur->addr != head)
     cur = cur->addr;

    cur->addr = temp;
    temp->addr = head;
    return head;
}
```

Addition of 2 long integers using S.L.L with header node:

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

struct node {
    int data;
    struct node *addr;
};
typedef struct node *NODE;

NODE insertbegin (NODE head, int item) {
    NODE temp;
    temp = (NODE) malloc (sizeof(struct NODE));
    temp->data = item;
    temp->addr = NULL;
    if (head->addr == NULL) {
        head->addr = temp;
        return head; }
    temp->addr = head->addr
    head->addr = temp;
    return head;
}
```

```
NODE insertend(NODE head, int item){
    NODE temp;cur;
    temp=(NODE)malloc(sizeof(struct NODE));
    temp->data=item;
    temp->addr=NULL;
    if(head->addr==NULL){
        head->addr=temp;
        return head;
    }
    cur=head->addr;
    while(cur->addr!=NULL)
        cur=cur->addr;
    cur->addr=temp;
    return head;}

NODE reverse(NODE head){
    NODE cur,prev,~~temp;~~ next
    cur=head->addr;
    prev=NULL
    while(cur!=NULL){
        next=cur->addr;
        cur->addr=prev;
        prev=cur;
        cur=next;
    }
    head->addr=prev;
    return head;
}

void display(NODE head){
    NODE temp;
    ~~if(head==NULL)~~
    if(head->addr==NULL)
        printf("List empty");
    else{ temp=head->addr;
```

```
        while (temp!=NULL){
        printf("%d", temp->data);
          temp = temp->addr; }
      }
    }

void appendzero ( NODE head1, NODE head2) {
    NODE cur
     int ~~count=0~~; c1=0, c2=0;
     cur = head1->addr;
       while (cur!=NULL) {
         ~~cou~~ c1 = c1+1 ;
             cur = cur->addr;
         }
      cur = head2->addr;
        while (cur!=NULL) {
            c2++;
           cur = cur->addr;
         }
      if (c1>c2) {
       for(int i=0 ; i<c1-c2; i++) {
           head2 = insertbegin (head2,0) ;}}
      else {
         for(int i=0 ; i< c2-c1; i++) {
           head1 = insertbegin (head1,0); }
         }
}

void add (NODE head1, NODE head2) {
        NODE head, t1, t2;
        int sum = 0, carry = 0;
  head = (NODE) malloc (sizeof(struct node));
  head->addr = NULL;
```

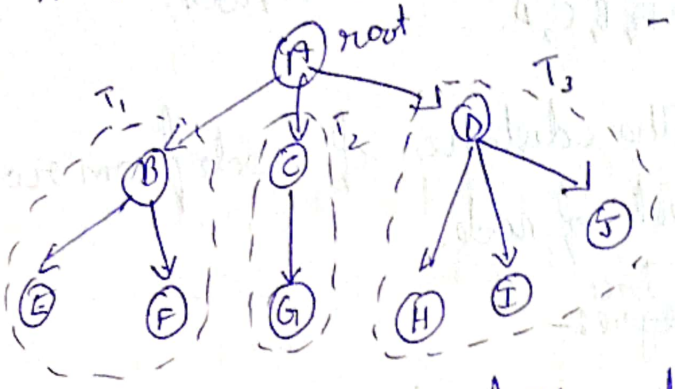
```
t1 = head1 -> address;
t2 = head2 -> address;
while (t1 != NULL)
head1 = reverse (head1)
head2 = reverse (head2)
t1 = head1 -> addr ;
t2 = head2 -> addr;
while (t1 != NULL) {
  sum = t1 -> data + t2 -> data + carry
  sum= carry = sum/10;
    Sum = sum % 10 ;
    head = insertbegin (head, sum);
      t1 = t1 -> addr;
      t2 = t2 -> addr ;
  }
  if (carry > 0)
      head = insertbegin (head, carry) **
    printf ("Resultant is\n") ;
    display (head);
}

intmain () {
  NODE head1, head2;
  char first[20], second[20];
  int i;
  printf (" Read first num\n");
  scanf ("%s", first);
  printf ("Read 2nd\n");
  scanf ("%s", second);
  head1 = (NODE) malloc (sizeof(struct node));
  head -> addr = NULL;
  for ( i=0 ; first[i] != '\0' ; i++)
```

```
    head1 = insertend (head1, first[i] - '0');
  printf("First numis : ");
  display (head1);
allocate head2 →
  for (i=0; second[i] != '\0'; i++)
      head2 = insertend (head2, first[i] - '0');
  display (head2);
  appendzero (head1, head2);
  add (head1, head2);
}
```

TREES :-

Trees is a finite set of one or more nodes that exhibits the parent and child relation such that

1) There is a special node termed root node
2) The remaining nodes are partitioned into disjoint subsets of nodes like $T_1, T_2, \ldots T_n (n \geq 0)$ which are termed as sub-trees



Root node : It is the reference or first node which is present at top of the tree. Root node doesn't have a parent.

Child node : Nodes obtained from the parents

A parent can have 0 or more child nodes. (E & F are child nodes of B)

Siblings : Two or more nodes having common parent are termed as siblings.

I and J siblings of H

The nodes obtained in the path from a specific node 'X' by moving upwards towards root node are termed as ancestors

A and C are ancestors of G.

Parent node:

Leaf node : Leaf nodes are those nodes which have degree of 0 (External node

Ex :- E, F, G, H, I, J

degree of a node: The number of sub trees for a given node is the degree of a node.

Internal node: All nodes of a tree apart from leaf nodes is termed as internal node.
Ex:- A, B, C, D.

level of a tree: The distance of node from root node is termed as the level of node.

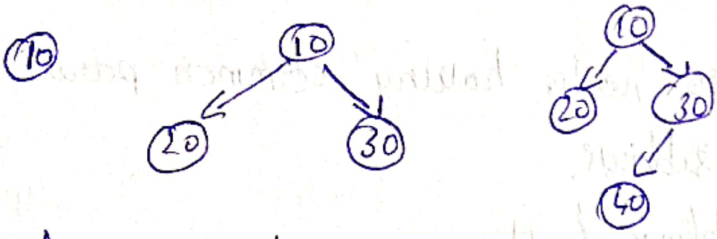
Height of the ~~degree~~ tree:-

Height = max level + 1

The maximum number of nodes processed to reach a leaf node at last level starting from root node is the height of the tree.

Binary Tree

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



Types of binary trees:-

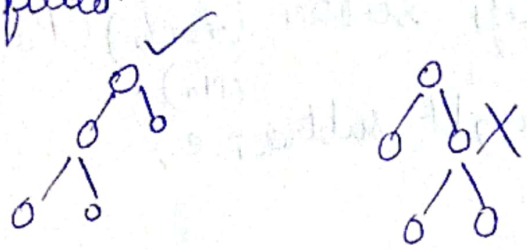
1) Strictly binary tree (FULL):
A binary tree having 2^i nodes at any given level i is termed as strictly binary tree

~~It is a binary tree in w~~

2) Complete binary tree
It is a binary tree in which every level apart from last level should have 2^i and last level should be

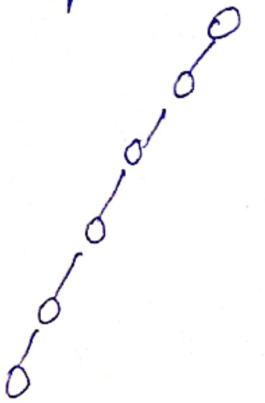
left filled.

→

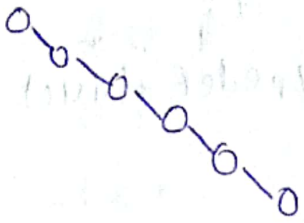


3) Skewed binary tree.

left skewed



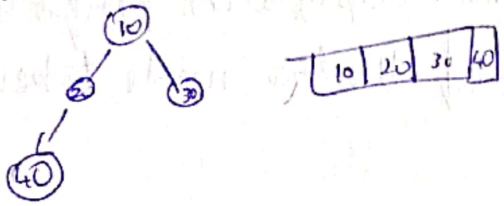
Right skewed

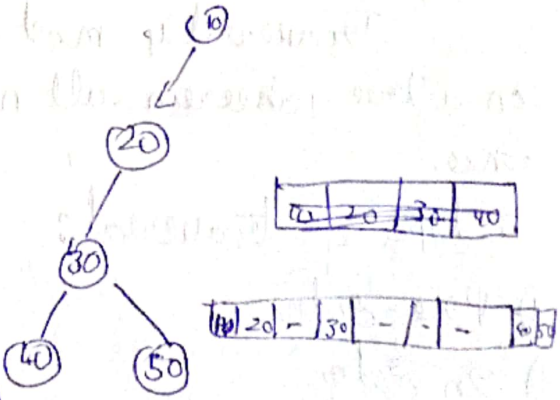


Representation of binary tree.

There are 2 ways of representing a binary tree. Array representation, list representation

1) Array representation.





For any node stored at index (i)

left child → $2i+1$

right child → $2i+2$

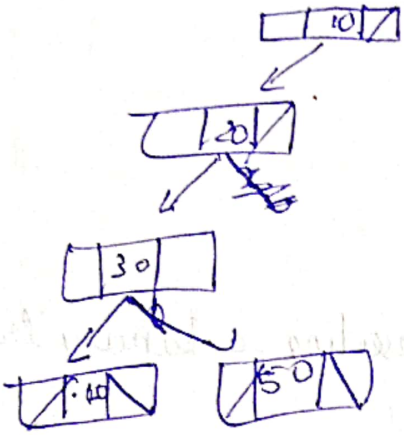
2) List representation:

In this representation we create a node which contains the following info

1) Data field → stores data

2) Left field → address of left subtree (Pointer) (Ptr)
3) Right right → address of right subtree. i.e.,

```
struct node {
    int data;
    struct node *left;
    struct node *right;
};
typedef struct node * NODE;
```



binary tree traversal:

Traversal is most common operation performed on a tree wherein all nodes of the tree needs to be visited once.

3 types of traversal:

1) Pre-order
2) In order
3) Post order.

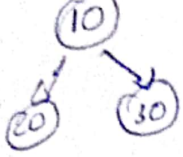
Pre-order.

The recursive definition is visit the root node

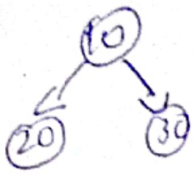
ii) Traverse left subtree in preorder

iii) Traverse right subtree in preorder.

Ex:

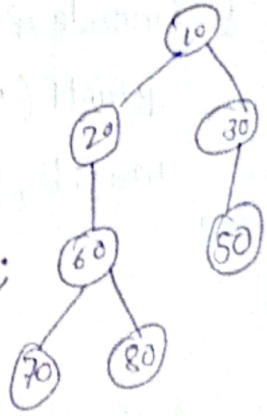


10, 20, 30



10 20 60 70 80 30 50

A C function to perform preorder recursively.



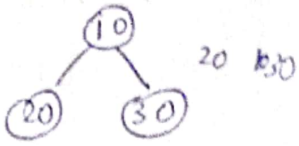

```
Void preorder (NODE root) {
    if (root != NULL) {
        printf("%d\t", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
```

Recursive definition for in-order traversal is

i) Visit the left subtree recursively in inorder

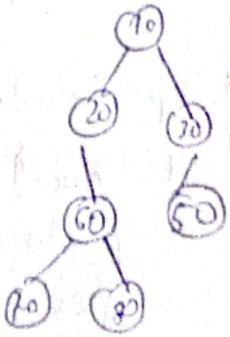
ii) Visit the root node.

iii) Visit right subtree in inorder.



20 10 30

20 70 80 10 50 30



2) Left
3) Right

C function to perform inorder

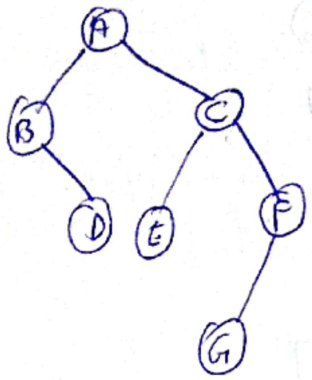
```
void inorder(NODE root) {
    if (root != NULL)
    { inorder(root->left);
      printf("%d\t", root->data);
      inorder(root->right);
    }
}
```

Recursive definition for post-order traversal

i) Visit left subtree recursively in post order
ii) Visit right subtree recursively in post order
iii) Visit root node

70 80 60 20 50 30 10

```
void postorder(NODE root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d", root->data);
    }
}
```



A B D C E F G
B D A E C G F
D B ~~A E~~ F C A
E G

Pre-order - A B D C E F G

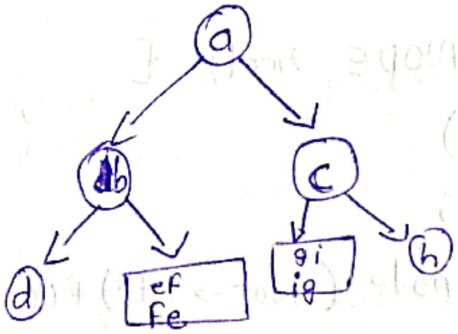
In-order - B D A E C G F

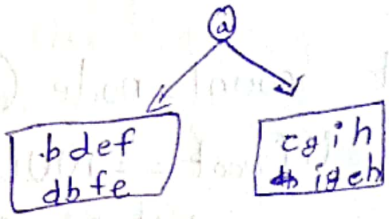
Post order - D B E G F C A

= From given traversal construct the binary tree

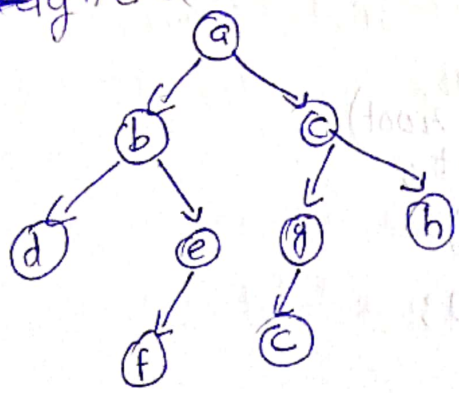
Pre-order -> a b d e f c g i h

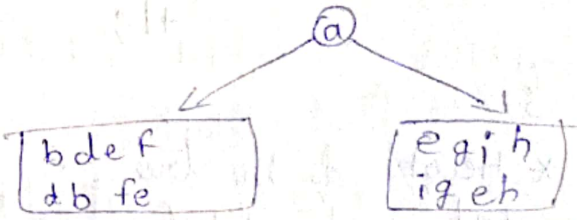
Inorder -> d b f e a i g c h

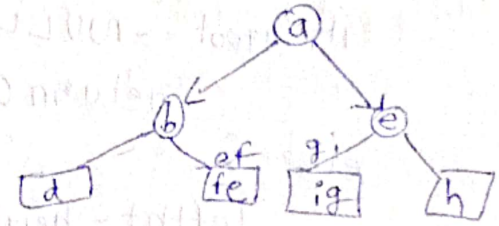




d f e b ~~e~~ i g h c a

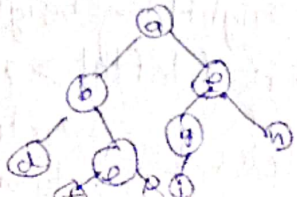






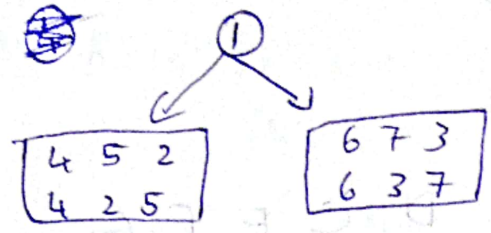
~~c b d f e~~

a b d e f e g i h
d b f e a i g c h



Postorder : 4 5 2 6 7 3 ①

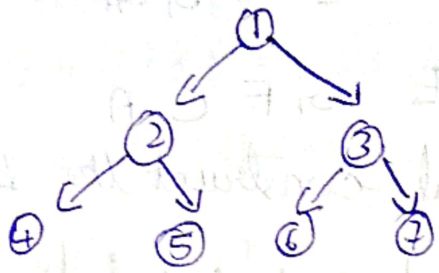
Inorder : 4 2 5 1 6 3 7



P

I

④ ⑤ ② 6 7 3 ①



---

*C function to find number of nodes in a binary tree

~~count node~~

```
int count_node (NODE root) {
    if(root == NULL)
        return = 0;
    return count_node (root->left) + count_node(root->
        +1;
}
```

* Height of the tree :-

```
int height(NODE root)
{   int leftht, rightht;
    if (root == NULL)
        return 0;
    else {
        leftht = height (root->left);
        rightht = height (root->right);
        if (leftht > rightht)
            return leftht+1;
        else {
```

C function to count number of leaf nodes.

```
int count_leaf(NODE root){
    if(root==NULL)
        return 0;
    else if(root->left==NULL && root->right==NULL)
        return 1;
    else {
        return count_leaf(root->left)+count_leaf(
                root->right);
```

C function to count non-leaf node

```
int count_nonleaf(NODE root){
    if(root==NULL)
        return 0;
    else if(root->left==NULL && root->right==NULL)
        return 0;
    else
        return count_nonleaf(root->left) + count_nonleaf
               (root->right)+1;
```

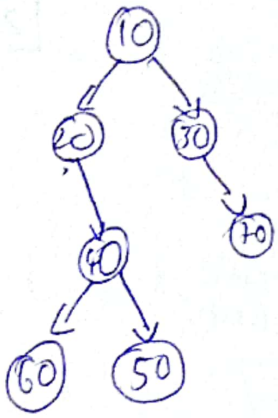
C function to create a node for a binary tree / bst

```
NODE createnode(~~NODE root~~, int item){
    NODE temp;
    temp=(NODE)malloc(sizeof(struct node));
    temp->data=item;
    temp->left=NULL;
    temp->right=NULL;
    return temp;
}
```

Create binary tree and perform operation on binary tree.

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

struct node {
   int data;
   struct node *left;
   struct node *right; };

typedef struct node *NODE;

void display (NODE root) {
   if (root != NULL) {
        display (root->left);
        printf("%d\t ", root->data);
        display(root->right);
    }
}

// write create_node function
NODE insertleft (NODE root, int item) {
      root -> left = create_node (item);
      return root->left;
 }

NODE insertright( NODE root, int item) {
      root->right = create_node(item);
      return root->right;
}

// write count_nodes function
// write height function
```

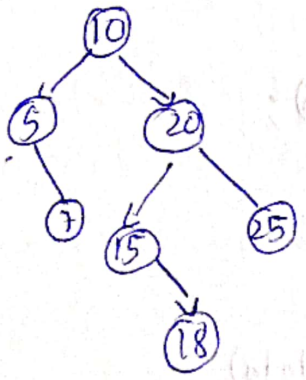
```
int main() {
    NODE root=NULL;
    root=createnode(10);
    insertleft(root, 20);
    insertright(root, 30);
    insertright(root->left, 40);
    insertright(root->right, 70);
    insertleft(root->left->right, 60);
    insertright(root->left->right, 50);
    insertright(
    printf("\n Tree data are\n");
    display(root);
    printf("\n Number of node S=%d\n", count_nodes(root));
    printf("\n Height of tree = %d\n", height(root));
    return 0;
}
```



Binary search tree.

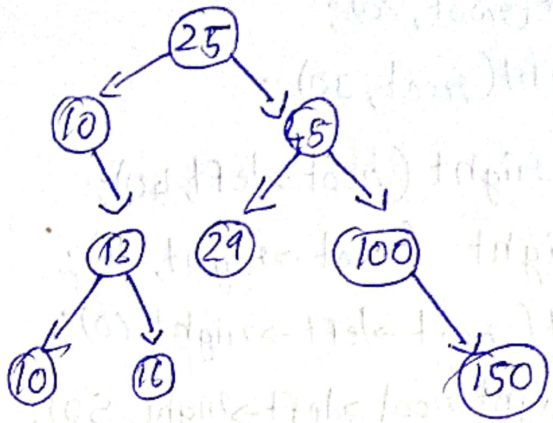
It is a binary tree wherein each node x has a property such that left child of x should have less value and right child x can have either greater or equal value.

Traverse → sorted
↓
B.S.T



Construct B.S.T for elements.

[25, 45, 10, 29, 100, 12, 16, 10, 150]



---

10) B.S.T and operations

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

typedef struct node {
        int data
        struct node *left;
        struct node *right;
        };

typedef struct node *NODE;

//write create node function
//preorder, inorder, postorder.

NODE InsertBST(NODE root, int item) {
    NODE temp;
    temp = create_node(item);
    if (root == NULL)
        return temp
    else {
        if(item < root->data)
            root->left = insertBST(root->left, item)
```

```
else
    root->right = insertBST(root->right, item)
}

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

NODE deletenode (NODE root, int key)
        NODE temp;
    if (root == NULL)
        return NULL;
    if (key < root->data)
        root->left = deletenode (root->left, key);
    else if (key > root->data
        root->right = deletenode (root->right, key);
    else {
        if (root->left == NULL) {
            temp = root->right
            free (root);
            return temp;
        }
        if (root->right == NULL) {
            temp = root->left
            free (root);
            return temp;
        }
        temp = inordersuccessor (root->right);
        root->data = temp->data;
```

```
        root->right = deletenode(root->right, temp->data);
    }
    return root;
}

int main(){
NODE root=NULL;
int ch,item,key;
for(;;) {
printf("1. Insert \t 2. Inorder\t 3. Preorder\t 4 Poster
        \t 5 Traversal \t 6. Delete\n");
printf("Read choice\n");
scanf("%d, &ch);
switch (ch) {
Case 1 : printf("Read item");
         scanf("%d", &item);
         root=insertbst(root,item);
         break;
case 2 : printf("\n Preorder\n");
         preorder(root);
         break;
/* Similarly for inorder and post-order */
Case 6: Printf("Node to be deleted");
        scanf("%d", &key);
        root=deletenode(root, item);
        break;
Default : exit(0);
}
}
return 0
}
```

write a function to create BST (Non-recursive)

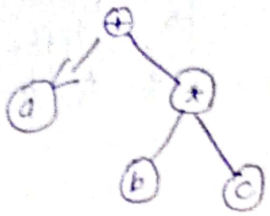
```
NODE insertBST (NODE root, int item) {
    NODE temp, prev, cur;
    temp = (create_node (item);
    if (root == NULL)
        return temp;
    cur = root;
    prev = NULL;
    while (cur != NULL) {
        prev = cur;
        if (item < cur->data)
            cur = cur->left;
        else
            cur = cur->right;
    }
    if (prev->data > item)
        prev->left = temp;
    else
        prev->right = temp;
    return root;
}
```

temp->data = item
left = NULL
right = NULL

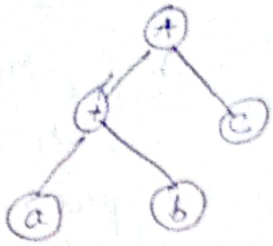
Expression tree: (Pakka)

↳ is a binary tree wherein the given expression (infix) is represented in a form of tree where operators are internal nodes and operands are leaf nodes

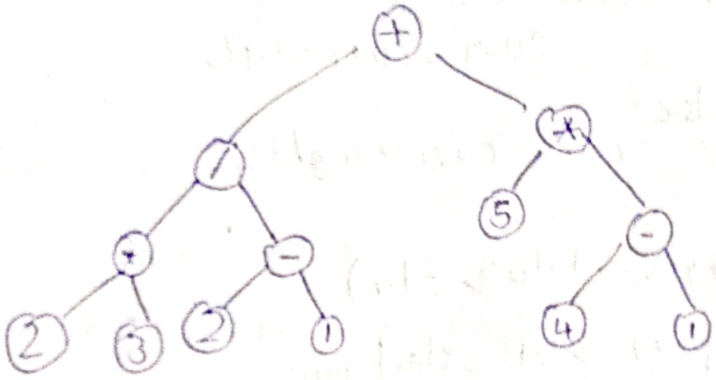
a+b*c



(a+b)*c



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



Algorithm to construct an expression tree for a given infix expression. (parenthesis free)

1) Initialize 2 stacks by name tree stack and operator stack
2) Scan the given infix expression from left to right
3) Create a node for the scanned symbol
4) If scanned symbol is

operand : Push corresponding node onto tree stack

operator : Push ~~corresponding node into~~

ii) If operator stack is empty push to corresponding node to operator stack.

else
until operator stack is not empty, check the precedence of topmost operator on operator stack ~~with~~ with precedence of scanned symbol.

if it is greater than or equal.

pop one node for operator stack and pop (2) nodes from 2 stacks and have them as right and left child of the operator node.

Push the ~~scanned~~ resultant tree node to tree stack

5) Until operator stack becomes repeated the procedure

6) Pop the tree stack.

Symbol	Tree stack	operator stack
a	a	empty
+	a	+
b	a b	+
*	a b	+ *
c	a, + (b, c)	+
~~d~~	+ (a, * (b, c))	Empty

Program (8) : ★★★★★

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

struct node {
   char item;
   struct node *left;
   struct node *right;
  };

typedef struct node *NODE;

struct stack {
    int top;
   NODE data[10];
  };
typedef struct stack STACK;

NODE createnode (char item) {
      NODE temp;
       temp = (NODE) malloc (sizeof (struct node));
       temp -> data = item;
       temp -> left = NULL;
       temp -> right = NULL;
       return temp; }

 void push (STACK *S, NODE item) {
       S -> data [++ (S -> top)] = item; }

NODE pop (STACK *S) {
return  S -> data [(S -> top) --] ; }
```

```
int preced(char symbol) {
    switch(symbol) {
    case '$' : return 5;
    case '*':
    case '/' : return 4;
    case '+':
    case '-' : return 3;
    }
}

Void preorder (NODE root) {
        if (root != NULL) {
            printf("%c", root->data);
            preorder (root->left);
            preorder (root->right);
        }}

void inorder (NODE root) {
            if (root != NULL) {
                printf("%c",r
                inorder(root->left);
                printf("%c", root->data);
                inorder(root->right);
            }
}

void postorder (NODE root) {
        if (root != NULL) {
            postorder(root->left);
            postorder(root->right);
            printf("%c", root->data);
            }
}
```

```
NODE create_exp_tree (NODE root, char infix[10]) {
  STACK TS, OS;   char symbol.
   TS.top = -1;
   OS.top = -1;
   NODE temp, t
   int i;
  for (i=0; infix[i] != '\0'; i++) {
      symbol = infix[i];
      temp = createnode (symbol);
      if (isalphanum (symbol))
          push (&TS, temp);
      else {
          if (OS.top == -1)
              push (&OS, temp);
          else
            {
              while (OS.top != -1 &&
                  preced (OS.data[OS.top] -> data) >=
                  preced (symbol))
              {   t = pop (&OS);
                  t->right = pop (&TS);
                  t->left = pop (&TS);
                  push (&TS, t);
              }
              push (&OS, temp);
          }
      }
  }
```

```
    while(OS.top != -1) {
            t = pop(&OS)
        t->right = pop(&TS);
        t->left = pop(&TS);
        push(&TS,t);
        }
    return pop(&TS);
}

int main() {
  NODE root=NULL;
  char infix[10];
  printf("\n Read the infix expression");
   scanf("%s", infix);
   root= create_exp_tree(root, infix);
   printf("\n Preorder:");
   preorder(root);
   /*Inorder and Postorder*/
    return 0;
  }
```

HEAP;

Heap is a binary tree with the following property

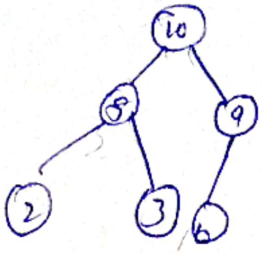
1) Structural property
2) Parent dominant property → It is should be an complete binary tree.

Parent dominant property means parent node should be

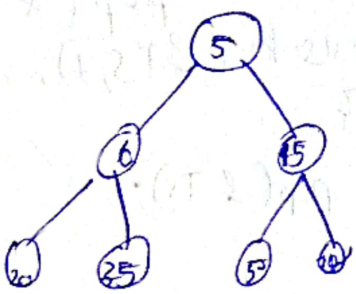
more dominant than its children.

Two types of heap:

1) MAX HEAP:



2) MIN HEAP:

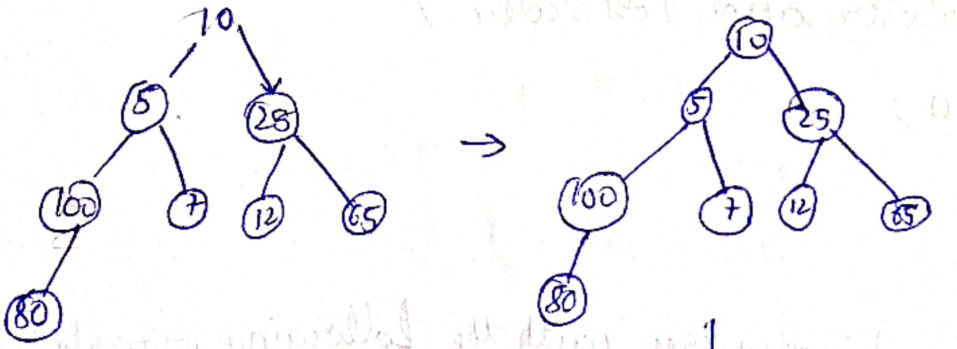


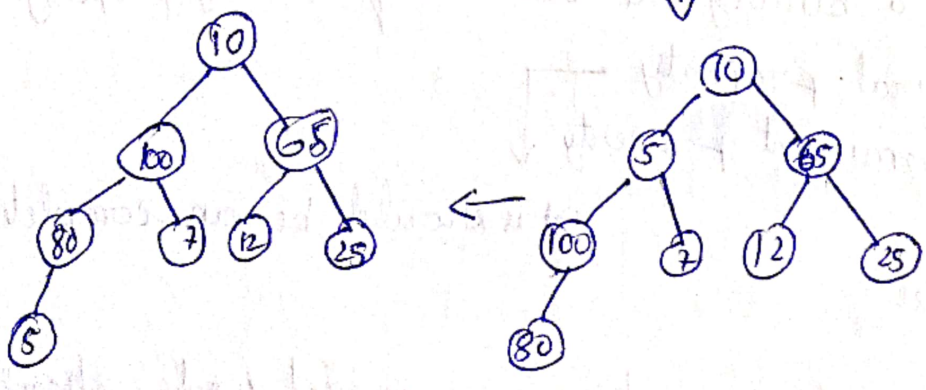
Two methods of constructing a heap

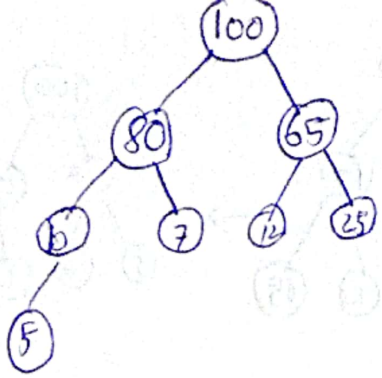
1) Bottom up approach
2) Top down approach

Construct max and a min heap using bottomup appro

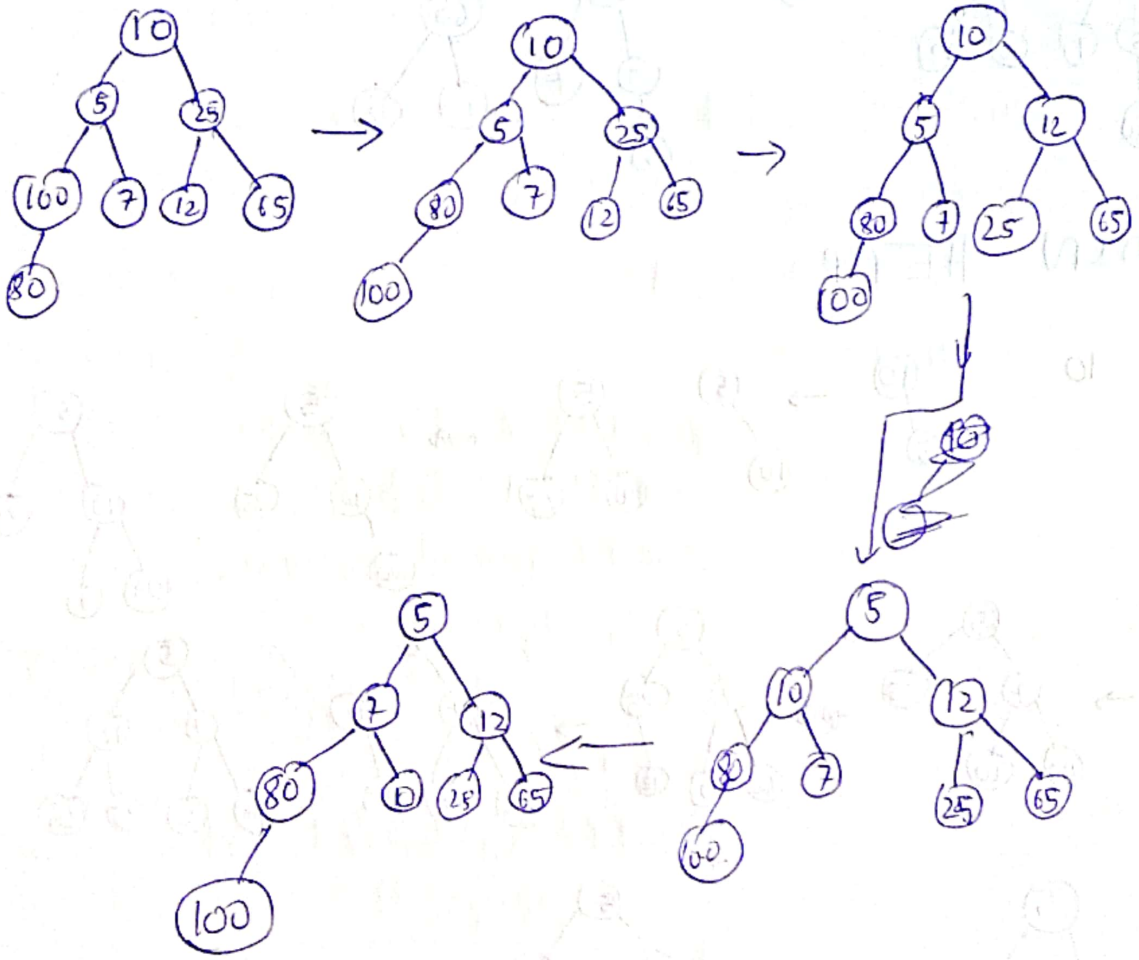
10 5 25 100 7 12 65 80







Min heap

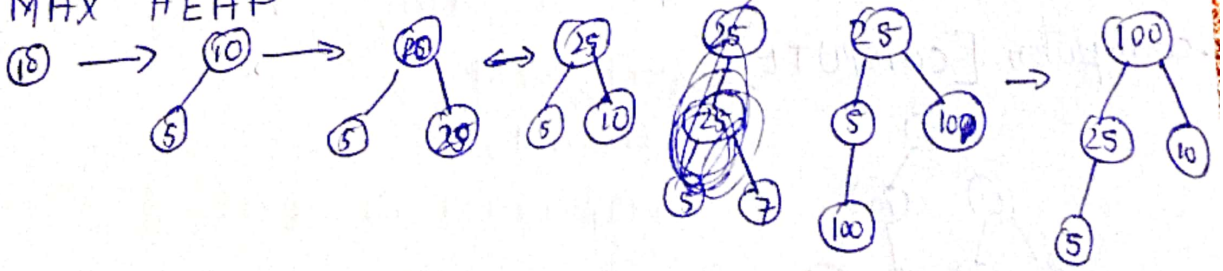


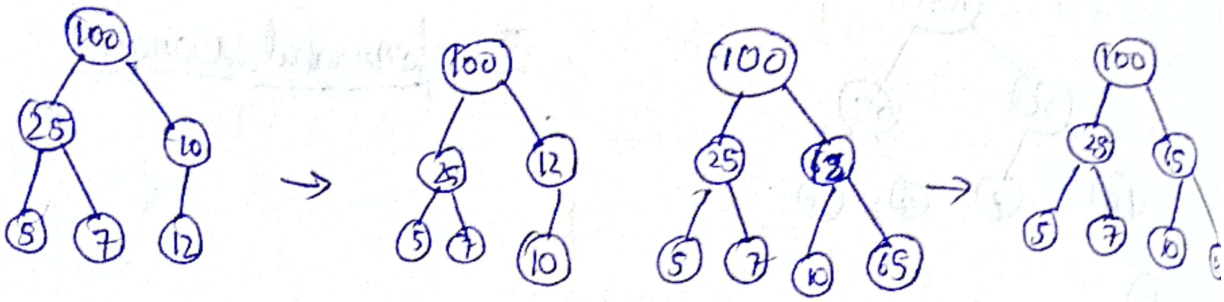
Top down approach

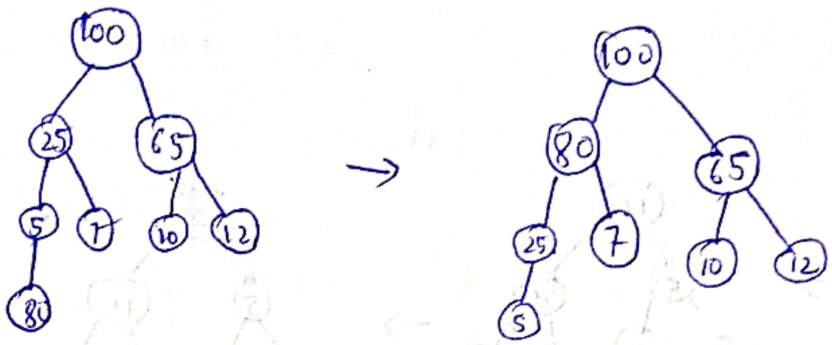
Construct max heap and min heap using

10 5 25 100 7 12 65 80

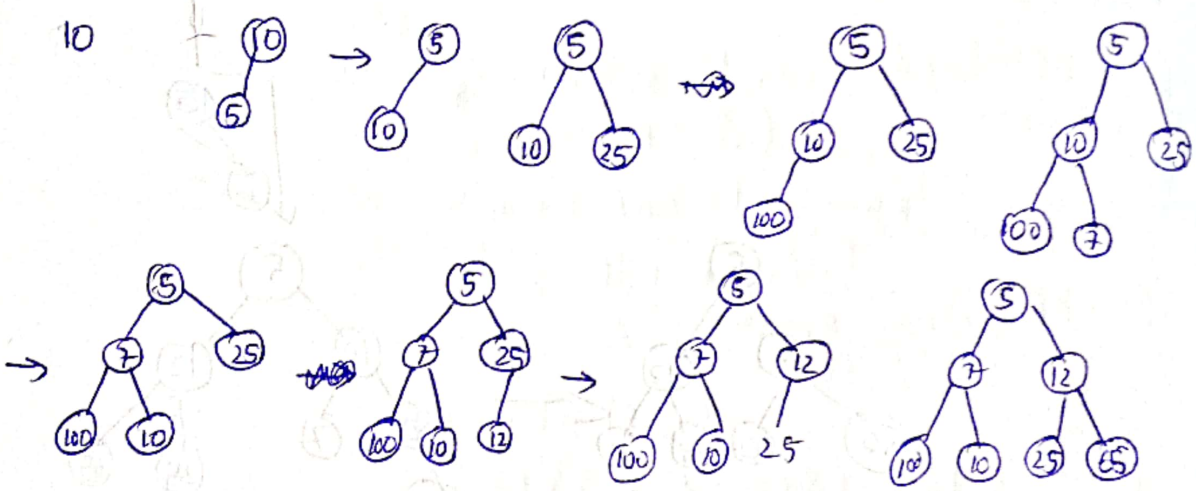
MAX HEAP

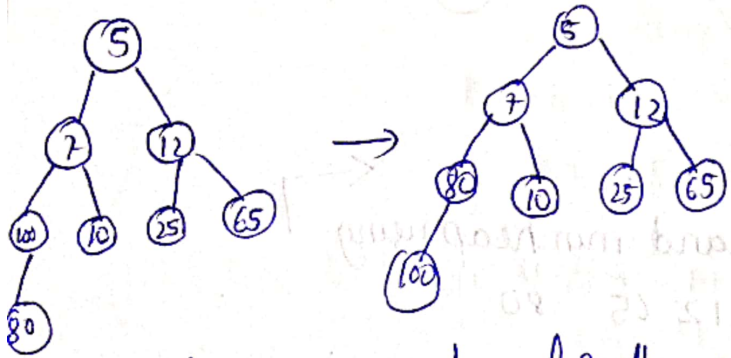




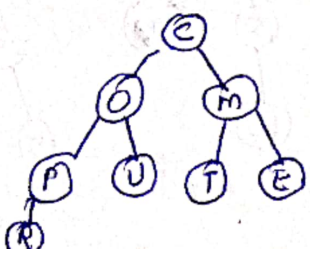


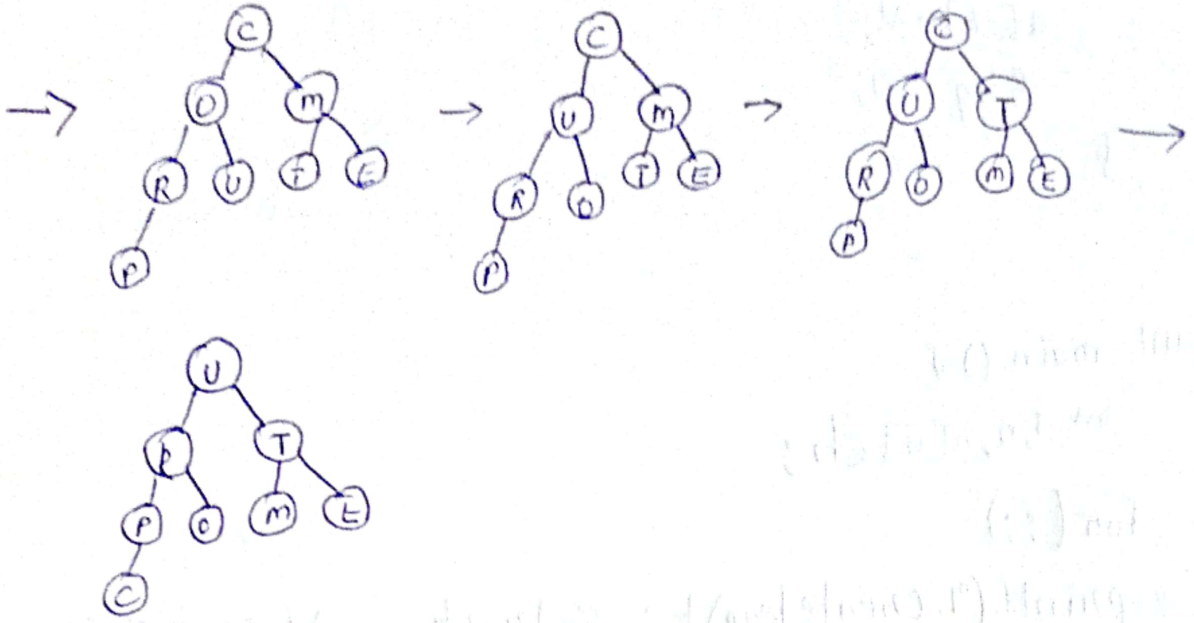
MIN HEAP





Construct a max heap for the characters of the word computer. [COMPUTER]





Program - 7

Ao implement priority queue using heap

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

void heapify(int a[10], int n) {
    int i, j, k, v, flag = 0;
    for (i = n/2; i >= 1; i--) {
        k = i
        v = a[k]
        while (! flag && 2 * k <= n) {
            j = 2*k
            if (j < n)
                { if (a[j] < a[j+1])
                    j = j + 1;
                }
            if (v >= a[j])
                flag = 1;
            else {
                a[k] = a[j];
                k = j;
            } }
```

```
        a[k] = v;
        Flag = 0;
    }
}

int main () {
    int i, n, a[12], ch;
    for (;;)
    printf("1. Create heap \t 2. Extract max \t 3. Exit");
    printf("Read choice : \n");
    scanf ("%d", &ch);
    switch (ch) {
        case 1 : printf("Read no of elements");
                 scanf("%d", &n);
                 printf("\n Read elements");
                 for (i=0; i<=n; i++)
                     scanf("%d", a[i]);
                 heapify (a, n);
                 printf("\n Elements after construct
                     heap\n");
                 for(i=1; i<=n; i++)
                     printf("%d\t", a[i]);
        case 2 : if (n >= 1)
                     printf("\n Element deleted is
                         %d", a[1]);
                 a[1] = a[n]
                 n = n-1;
                 heapify (a, n) if (n != 0) {
                 printf("Elements after reconstruction");
                 for(i=0; i<n; i++)
```

```
printf("%d\t",a[i]);}
        break;
    default : exit(0)
    }
  }
return 0;
}
```