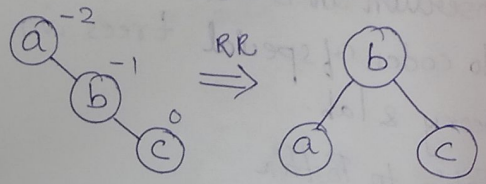
* SPECIAL TREES:

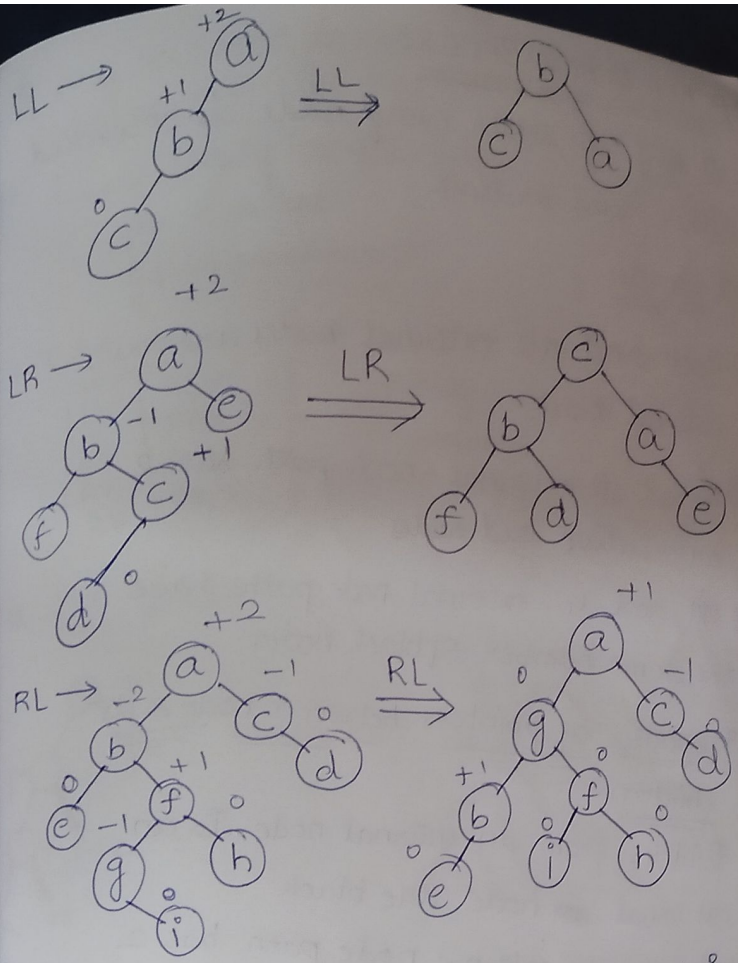
AVL tree, ~~special~~ Red-black tree, splay tree, B-tree

* AVL Trees:

* Balanced binary search tree wrto height of the subtree.
* For every node, the balancing factor is calculated which should be either $-1, 0, 1$ and if the balancing factor increases/changes beyond these values, we must rotate the AVL tree acc. to RR, RL, ~~LR~~, LL rotations.
* balancing factor = height of left subtree − height of right subtree
* Every node should be balanced in an AVL tree
* Rotations:

RR →





→ Transformations done to remedy LL & RR imbalances are often called single rotations while those done for LR & RL imbalances are called double rotations.

- LR → RR + LL
  RL → LL + RR.

} In order

RED BLACK TREES:

* A BST in which every node is coloured either red or black

* Properties:

1) Root and all external nodes are colored black.
2) No root to external-node path has 2 consecutive red nodes.
3) All root-to-external node paths have the same number of black nodes.

* Properties of pointers between node & its children:

1) Pointers from an internal node to an external node are black.
2) No root-to-external node path has 2 consecutive red pointers.
3) All root-to-external node paths have the same number of black pointers.

* Insertion in red-black tree:

1) If the tree was empty; then the newly inserted node is root & is given black colour.
2) The new node inserted to a non-empty tree is red in colour.

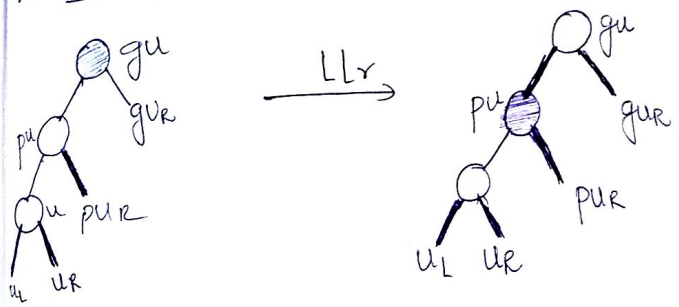
3) any new node inserted may cause imbalance to the tree.
   - u → newly inserted node
   - pu → parent
   - gu → grandparent.

4) Rotations:

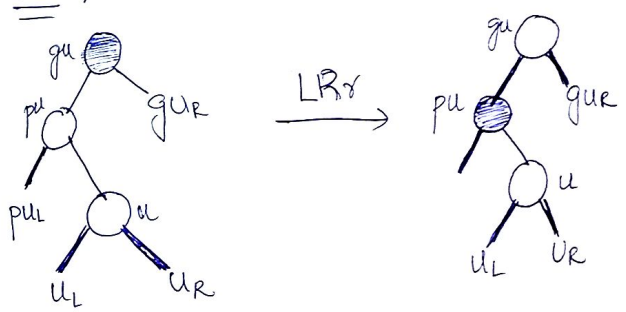
LLr	RLr
LLb	RRb
LRr	RLb
LRb	RRr

- XYr → X & Y may be L or R
  * only colour change is required.
- XYb → require rotation.

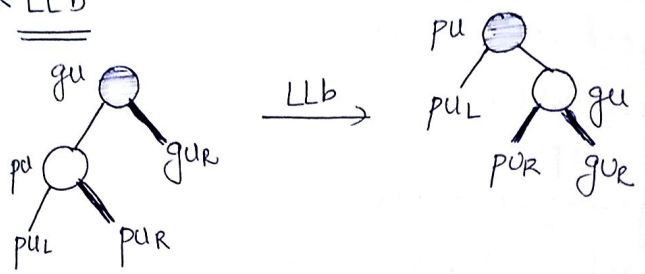
* LLr ~~RLr~~



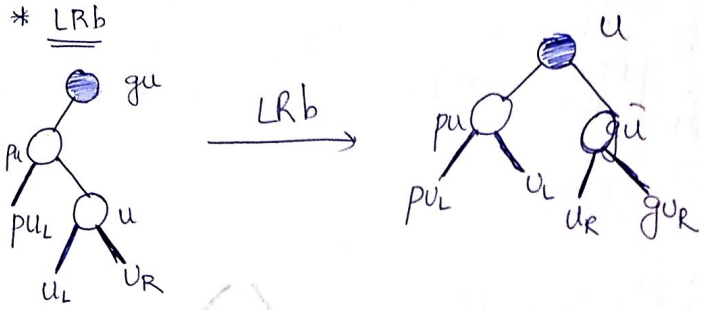
* LRr / RRr



* LLb



* LRb



SPLAY TREES:

* Search time is reduced.
* 2 types ⇒ Bottom-Up, Top-Down splay trees.

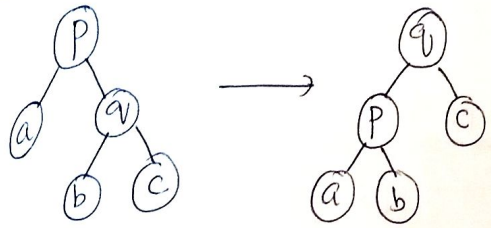
Bottom-Up Splay Tree:

→ The operations such as insert, delete, search are performed after performing splay where the element inserted, deleted or searched must be at the root which is called splay.

→ This tree is generally used for cache applications which makes the retrieval of the data easy.

→ Properties & Rotations: q → splay is performed

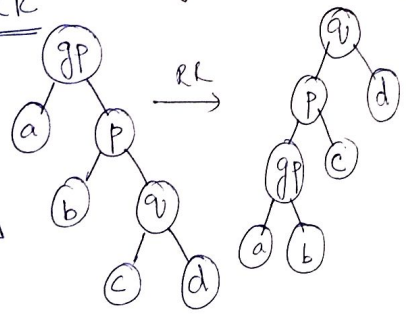
1) q has a parent p & no grandparent



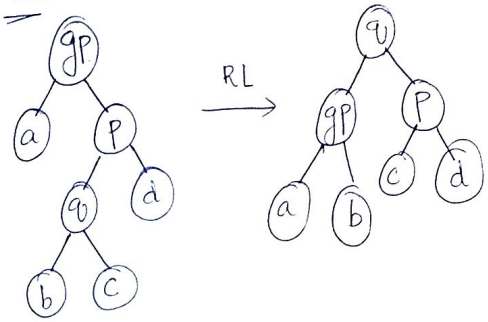
2) If q has a parent p and a grandparent gp then rotation is classified as

- LL
- LR
- RR
- RL

RR



RL:



```
void insert(int e, int a[], int i)
{ a[0]=e;
  while(e<key
```

```
void insertion_sort(int a[], int n, int e)
{ for(i=n-1 ; i>0; i--)
  { if(a[i] <e)
    { a[i+1]=e;
      break;}
  } a[i+1]=a[i]
```

```
void insertion_sort()
{ int v;
  for(int i=0; i<=n-1; i++
  { v=ins[i];
    int j=i-1;
    while(j>=0 && ins[j]>v)
    { ins[j+1]=ins[j];
      j=j-1;
    }
    ins[j+1]=v;
  }
}
```

Quick Sort: elements<pivot (pivot) elements > pivot

* Used when the data is large and random.
* worst case behaviour = $O(n^2)$

```
* void quicksort(int x[20], int first, int last)
* { int pivot, j, temp, i;
  if(first<last)
  { pivot=first;
    i=first;
    j=last;
  while(i<j)
  { while(x[i]<=x[pivot] && i<last)
      i++;
    while(x[j]>x[pivot])
      j--;
    if(i<j)
    { temp=x[i];
      x[i]=x[j];
      x[j]=temp;}}
    temp=x[pivot];
    x[pivot]=x[j];
    x[j]=temp;
    quicksort(x, first, j-1);
    quicksort(x, j+1, last);}}
```

Merge Sort:

* divide and conquer rule =

```
* void merge (int arr[], int l, int m, int r)
{ int i,j,k;
  int n1 = m-l+1;
  int n2 = r-m;
  int L[n1], R[n2];
for(i=0;i<n1;i++)
{ L[i] = arr[l+i];}
for(j=0; j<n2;j++)
{R[j] = arr[m+1+j];}
 i=0; j=0; k=l;
while (i<n1 && j<n2)
{ if(L[i] <=R[j])
  { arr[k] = L[i]; i++;}
  else
   { arr[k] = R[j]; j++;}
  k++; }
while (i<n1)
{ arr[k] = L[i];
   i++; k++;}
while(j<n2)
{ arr[k] =R[j]; k++; j++;}}
```

```
void mergesort (int arr[], int l, int r)
{ if(l<r)
  {int m = l+(r-1)/2;
   mergesort(arr, l, m);
   mergesort (arr, m+1, r);
   merge (arr, l, m, r);}}
```

Heap Sort:

* worst case and average computing time = O(n log n)

* slower than merge sort.

```
* void swap(int *a, int *b)
  { int temp= *a;
      * a = * b;
      * b = temp;}
void heapify (int arr[], int N, int i)
 { int large = i;
   int left = 2*i+1;
   int right = 2*i+2;
   if (left <N && arr[left] > arr[large])
      { large = left;}
   if(right <N && arr[right] >arr[large])
      { large = right;}
```

```
if (large != i)
{ swap(&arr[i], &arr[large]);
  heapify (arr, N, large); } }
void heapsort (int arr[], int N)
{ for (int i = N/2 - 1; i >= 0; i --).
  { heapify (arr, N, i); }
  for (int i = N-1; i >= 0; i --)
  { swap(&arr[0], &arr[i]);
    heapify (arr, i, 0); }
}
```

~~* Hashing~~

~~→ 2 types: static & dynamic~~

* GRAPHS:

→ A graph G, consists of 2 sets; V and E.
V is a finite, non-empty set of vertices.
E is a set of pair of vertices called edges.
G = (V, E).

- → Directed graph ⟹ one-way edges / directions are specified
- → Undirected graph ⟹ no arrows / directions

→ Properties:

* A graph may not have an edge from a vertex v back to itself.
* A graph may not have multiple occurrences of the same edge.
* Tree is connected acyclic graph.

→ A directed graph is said to be strongly connected iff for every pair of distinct vertices u & v in V(G) there is a directed path from u to v and also from v to u.

→ Graph Representations:

* Adjacency Matrix
* Adjacency List

* ADJACENCY MATRIX:

→ If there is an edge between 2 vertices the matrix intersection of row & column shall have 1 as the matrix shall be the size of n × n where n = no. of vertices

→ It is not symmetric for a directed graph and this takes n^2 bits of space.

→ It is symmetric for a directed graph and we can store either upper or lower triangular matrix.

→ If the no. of zeroes are more in a graph; we shall use sparse matrix to store the data where time shall be O(e+n) e → no. of edges

* Adjacency List:

→ n rows of adjacency matrix are represented as n chains. There is one chain for each vertex in G.

→ The nodes in chain i represent the vertices that are adjacent from vertex i.

→ For an undirected graph with n vertices & e edges; the linked adjacency lists representation requires an array of size n and 2e chain nodes.

→ The degree of any vertex in an undirected graph is determined by just counting the no. of nodes in its adjacency list.

* Operations on Graphs:

DFS: depth first search:

→ We start visiting a vertex V and push the adjacent vertices to the stack.

→ Then we select the next node which is not visited until we reach a vertex which has no unvisited vertices.

→ At this point; we remove a vertex from the stack.

→ Previously visited vertices are discarded; unvisited vertices are visited and placed on the stack.

→ Search terminates when stack is empty.

```
#define FALSE 0
#define TRUE 1
int visited[MAXSIZE];
void dfs(int v)
{ nodePointer w;
  visited[v] = TRUE;
  printf("%d", v);
  for (w = graph[v]; w; w=w→link)
  { if(!visited[w→vertex])
      { dfs(w→vertex);}}
}
```

BFS: breadth first search:

→ Starts at a vertex V and marks it as visited and then visits the adjacent vertices.

→ We use queue to implement BFS.

→ When we have exhausted the adjacency list; we remove a vertex from the queue & proceeds by examining each of the vertices on its adjacency lists

→ Unvisited vertices are visited & then placed on the queue; visited vertices are ignored.

```
void bfs(int v)
{ nodePointer w;
  front = rear = NULL;
  printf("%d", v);
  visited[v] = TRUE;
  addq(v);
  while (front)
  { v = deleteq();
    for(w = graph[v]; w; w = w->link)
      { if (!visited[w->vertex])
        { printf("%d", w->vertex);
          addq(w->vertex);
          visited[w->vertex] = TRUE;
        }}
  }
```

```
* void connected(void)
{ int i;
  for(i=0; i<n; i++)
  { if (!visited[i])
    { dfs(i);
      printf("\n");}}
```

Spanning Trees:

→ A spanning tree is any tree that consists solely of edges in G and that includes all the vertices in G

→ Minimum cost spanning trees.

* The cost of a spanning tree of an undirected graph is the sum of the costs of edges in the spanning tree.
* A minimum cost spanning tree is a tree with least cost.
* Algorithms used for obtaining minimum cost spanning tree are: Kruskal's, Prim's and Sollin's algorithm.
* The minimum cost spanning trees must satisfy the following criteria:
  → We must use only edges within the graph.
  → We must use exactly n-1 edges.
  → We may not use edges that would produce a cycle.

* Kruskal's Algorithm:

→ It builds a minimum cost spanning tree T by adding edges to T one at a time.

1) The edges of a graph are arranged in increasing order of their costs.

2) Then one after the other edges are selected ; such that there is no cycle formed with already inserted one.

→ Algorithm:

```
T = { };
While (T contains less than n-1 edges && E is not empty)
{ choose least cost edge (V,W) from E;
  delete (V,W) from E;
  if ((V,W) does not create a cycle in T)
    { add (V,W) to T; }
  else
    { discard (V,W); }
}
if (T contains fewer than n-1 edges.)
  { printf("No spanning tree\n"); }
```

*Prim's Algorithm:

→ constructs minimum cost spanning tree one edge at a time

→ This begin with a tree 'T' ; that contains a single vertex. This may be any vertex in the original graph

1) We select a vertex from the original graph and build a tree T.

2) Then we add a least cost edge (u,v) to T such that ; becomes a tree itself

3) We repeat this until 'T' contains n-1 edges.

4) To make sure that the added edge does not form a cycle ; at each step we choose the edge (u,v) such that exactly either of u or v is in T.

→ Algorithm:

```
T = { };   → set of tree edges
TV = {0};  → set of tree vertices.
while (T contains fewer than n-1 edges)
{ let (u,v) be a least cost edge such
  that u ∈ TV and v ∉ TV;
  if (there is no such edge)
    { break; }
  add v to TV;
  add (u,v) to T;
}
if (T contains fewer than n-1 edges)
  { printf("No spanning tree\n"); }
```

* TREES:

→ A tree is a non-linear data structure that starts with a root node and continues hierarchically.

- → Binary Tree
- → Binary Search Tree
- → Heap Tree.

* Node: Information is held and the branches to other nodes.

*→ Degree: no. of subtrees of a node.

Degree of a node:

Degree of a tree: Max. of the degree of the nodes in a tree.

→ Binary Tree:

* A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called left subtree and right subtree.

Properties:

→ maximum no. of nodes on a level 'i' $= 2^{i-1}$

→ maximum no. of nodes in a tree of height 'h' $= 2^k - 1$

→ A binary tree with n nodes and depth 'k' is complete iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k.

* Representations:

1) Array representation:

→ If a binary tree with n nodes is represented sequentially as 1D array then for any node with index i:

→ parent (i) is at $\lfloor i/2 \rfloor$
→ left child (i) is at $2i$ $2i > n \rightarrow$ no lchild
→ right child (i) is at $2i+1$, $2i+1 > n \rightarrow$ no rchild

2) Linked list:

```
typedef struct node *treePointer;
typedef struct
{int data;
 treePointer lchild, rchild;
}node;
```

lchild	Node	rchild

→ Level order Traversal of a binary tree:

```
void levelOrder (treePointer ptr)
{ int front=rear=0;
  treePointer queue[MAX_QUEUE_SIZE];
  if(!ptr) return;
  addq(ptr);
  for(;;)
  { ptr= deleteq();
    if(ptr)
    { printf("%d", ptr→data);
      if(ptr→leftchild)
       {addq(ptr→leftchild);}
      if(ptr→rightchild)
       {addq(ptr→rightchild);}
    }
    else break;}
}
```

* copying of a binary tree:

```
treePointer copy(treePointer original)
{ treePointer temp;
  if(original)
  { Malloc(temp, sizeof(*temp));
    temp→leftchild = copy(original→leftchild);
    temp→rightchild = copy(original→rightchild);
    temp→data = original data;
    return temp;}
  return NULL;}.
```

* Heaps:

→ Heaps are frequently used to implement priority queues.

→ Max heap tree: A heap tree where the node has the maximum element compared to its child nodes.

→ Min heap tree: A heap tree where the node has the minimum element compared to its child nodes.

* Heap Sort algorithm deals with heap trees where the elements are arranged as a tree and converted either to max heap (or) min heap tree and then sorted accordingly.

*Binary Search Tree:

```
struct node
{ int data
  struct node *lc, *rc;
```

→ A binary search tree is a binary tree which may be empty.
If it is not empty then it should satisfy

1) Each node has exactly one key and the keys in the tree are distinct.

2) The keys in left subtree are smaller than the key in the root.

3) The keys in right subtree are larger than the key in the root.

4) The left and right subtrees are also binary search trees.

* CODES:

```
→ void preorder (node *t)
  { if (t)
    { printf("%d", t→data);
      preorder (t→lchild);
      preorder (t→rchild); }}

→ void inorder (node *t)
  { if (t)
    { inorder (t→lchild);
      printf("%d", t→data);
      inorder (t→rchild); }}

→ void postorder (node *t)
  { if (t)
    { postorder (t→lchild);
      postorder (t→rchild);
      printf("%d", t→data); }}
```

```
→ struct node * createNode (int value)
  { struct node *newNode = malloc(sizeof(struct node));
    newNode→item = value;
    newNode→left = NULL;
    newNode→right = NULL;
    return newNode; }

→ struct node *insert (struct node *root, int x)
  { if (root == NULL)
      return createNode (x);
    else if (x > root→item)
      root→right = insert(root→right, x);
    else
      root→left = insert(root→left, x);
    return root;
  }

→ struct node *search (struct node *root, int x)
  { if (root == NULL || root→item == x)
      return root;
    else if (x > root→item)
      return search(root→right, x);
    else
      return search(root→left, x); }
```

```
struct node *find_minimum(struct node *root)
{ if (root == NULL)
      return NULL;
  else if (root->left != NULL)
      return find_minimum(root->left);
  return root;}

struct node *find_maximum(struct node *root)
{ if (root == NULL)
      return NULL;
  else if (root->right != NULL)
      return find_maximum(root->right);
  return root;
}

struct node *delete(struct node *root, int x)
{ if (root==NULL) return NULL;
  if (x > root->item)
      root->right = delete(root->right, x);
  else if (x < root->item)
      root->left = delete(root->left, x);
  else
  { if (root->left==NULL && root->right==NULL)
    { free(root); return NULL; }
    else if (root->left==NULL || root->right==NULL)
    { struct node *temp;
      if (root->left == NULL)
          temp = root->right;
      else
          temp = root->left;
      free(root);
      return temp; }
    else
    { struct node *temp = find_minimum(root->right);
      root->item = temp->item;
      root->right = delete(root->right, temp->item);
    } return root; }
}
```

* Stacks and Queues:

→ **Stack:** A stack is an ordered list in which insert/push and delete/pop is done at one end called top.

LAST IN FIRST OUT – LIFO

→ **Stack using dynamic arrays:**

```
void push(char e, char stack[], int *top,
          int size)
{ if (*top == -1)
  { stack[size-1] = e;
    *top = size-1; }
  else if(*top==0)
    {pf("Stack overflow"); }
```

```
else
{stack[*top - 1] = e;
  *top --; }
}
void pop(char stack[], int *top, int size)
{ if (*top == 1)
  {printf("Stack is empty");}
  else
  {printf("Element popped : %c\n", stack[*top]);
   if (*top == size-1) { *top = -1; }
   else { *top ++; }
  }}
```

→ Queue: A queue is an ordered list in which insert/add are done at the rear end and delete/remove are done at the front end.

FIRST IN FIRST OUT - FIFO

~~Stack using~~

→ Queue using arrays:

```
void insert()
{ int add;
  if (rear == MAX-1)
  {printf("Queue Overflow\n");}
else
{ if (front == -1)
  {front = 0;
   printf("Insert the element in queue");
   scanf("%d", &add);
   rear += 1;
   queue[rear] = add;
  }}
}
void delete()
{ if (front == -1 || front > rear)
  {printf("Queue Underflow\n");
   return;}
  else
  {printf("Element deleted from queue: %d\n",
                                  queue[front]);
   front += 1;}
}
```

→ Circular Queue:

```
void addq(int item)
{rear = (rear+1) % QUEUE_SIZE;
  if (front == rear)
   { queueFull(); }
  queue[rear] = item; }
```

→ Postfix expression evaluation using stack:

```
#include<stdio.h>
#include<ctype.h>
#define MAX 100
#define POSTSIZE 100
int stack[MAX];
int top=-1;
void push(int i)
{ if(top>=MAX-1)
   {printf("Stack overflow"); return;}
   else { top+=1; stack[top]=i; }
}
int pop()
{int i;
 if(top<0){printf("Stack Underflow");}
 else
 { i=stack[top];
   top=top-1;
   return i; } }
void EvalPostfix(char postfix[])
 {int i;
  char ch;
  int val, A, B;
```

```
for(i=0; postfix[i]!=')'; i++)
{ ch=postfix[i];
  if (isdigit(ch))
   { push(ch-'0'); }
  else if (ch=='+'||ch=='-'||ch=='*'||
                                ch=='/')
   { A=pop(); B=pop();
     switch(ch)
     { case '*': val=B*A;
                 break;
       case '/': val=B/A;
                 break;
       case '+': val=B+A;
                 break;
       case '-': val=B-A;
                 break; }
     push(val); } }
  printf("Result: %d\n", pop()); }
int main()
{ int i;
  char postfix[POSTSIZE];
  printf("Four operators (*, /, +, -) & operand
          is single digit:\n");
  printf("\n Enter postfix expression\n,
          press ')' for end expression:");
  for(i=0; i<=POSTSIZE-1; i++)
  { scanf("%c", &postfix[i]);
    if(postfix[i]==')')
     { break; } }
  EvalPostfix(postfix); return 0; }
```

→ Infix to Postfix:

```
#include <stdio.h>
#include <ctype.h>
char stack[100];
int top = -1
void push(char x)
{stack[++top] = x; }
char pop()
{ if(top==-1) return -1;
  return stack[top--]; }
int priority(char x)
{ if(x=='(') return 0;
  else if(x=='+' || x=='-') return 1;
  else if(x=='*' || x=='/') return 2;
  return 0; }
```

```
int main()
{ char exp[100], *e, x;
  scanf("%s", exp);
  e=exp;
  while(*e != '\0')
  { if ( isalnum(*c))
        printf("%c", *e);
    else if (*e=='(')
        push (*e);
    else if (*e== ')')
     { while ((x=pop()) != '(')
          { printf("%c", x); }
    else
    {while (priority (stack[top] >=priority (*e))
       { printf("%c", pop());
     push (*e); }
    e++; }
while (top != -1)  printf("%c", pop());
return 0; }
```

* Linked List:

→ Linked list: It is a linear data structure where each node of a linked list consists of the address of the next element alongwith data element.

→ Code:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct
{struct node * link;
 int data;} node;

node * add (int n)
 { node * head = NULL;
   node * temp = NULL;
   node  * p = NULL;
   for(int i=0; i<n; i++)
    { temp = (node *) malloc(sizeof(node *));
      printf("Enter the data of the
                element %d", i+1);
      scanf("%d", &(temp->data));
      temp->link=NULL;
      if (head == NULL)
       { head = temp; }
```

```
else
{ p = head;
  while (p->link != NULL)
    {p = p->link;}
  p->link = temp;}}
  return head;}
void display (node *head)
 {node *p=head;
  while (p != NULL)
  { printf("d -> %d", p->data);
    p=p->link; }
 }
node * insertion (node *head, inte, int k)
 { node *p=head;
   int i=0;
   node * temp = (node *) malloc(sizeof (node *));
   temp->data =e;
   if(k != 0)
   { while i<k-1)
     { i++;
       p= p->link;}
   temp-> link = p->link;
   p-> link = temp; }
```

```
else
{temp->link = head;
    head = temp;}
 return head;
}
node * deletion (node *head, int k)
{node  * p = head ;
 node *temp = (node *) malloc(sizeof(node *));
 int i = 0;
  if(k > 0)
 { while(i<k-1)
   { i++;
     p = p->link;}
  temp = p->link;
   p->link = temp->link;
   free (temp);}
 else if (k==0)
   { temp ->link = p;
     head = p->link;
      free(temp);}
  return head;}
```

* Stacks using Linked list:

```
#include <stdio.h>
#include <stdlib.h>
Struct Node
{ int data;
  struct Node *next; };
Node *top=NULL;
void push(int value)
{ struct Node *newNode;
  newNode = (struct Node) malloc(sizeof
                                 (struct Node));
  newNode -> data =value;
  if (top == NULL)
   {newNode -> next = NULL; }
  else
   {newNode -> next = top;}
  top = newNode;
  printf("Node is inserted\n");}
int pop ()
{ if (top==NULL)
  {printf(" Stack Underflow\n");}
else
{ struct Node *temp =top;
int temp -> data = top -> data;
  top = top -> next;
  free(temp);
  return temp -> data;}}
void display()
{ if (top==NULL)
 {printf("\n Stack Underflow\n");}
  else
 {printf("Stack is \n");
  Struct Node *temp=top;
  while(temp -> next != NULL)
  { printf("%d -> ", temp -> data);
    temp = temp -> next;}
    printf("%d -> NULL\n\n", temp -> data);}}
```

* Queues using Linked list:

```
Struct Node
{ int data;
  Struct node *next;};
  Struct node *front =NULL;
  Struct node *rear =NULL;
 void insert(int value)
  { Struct node *ptr;
    ptr = (struct Node*)
          malloc(sizeof(struct no
```

```
ptr→ data = value;
ptr→ next = NULL;
if((front ==NULL)&& (rear==NULL))
   { front = rear = ptr;}
else
{ rear→next=ptr;
   rear=ptr;}
   printf(" Node is inserted\n");}

int delete().
{ if (front==NULL)
   { printf("Underflow\n");
     return -1;}
  else
   { struct node *temp= front;
     int temp→data =front→data;
     front= front →next;
     free (temp);
     return temp→data;}}
void display()
{ struct node *temp;
 { if((front ==NULL)&& (rear==NULL))
   { printf("\n Queue is empty\n");}
else
{ printf(" Queue is\n");
  temp= front;
  while (temp)
  { printf("%d→ ", temp→data);
    temp = temp→next;}
    printf("NULL\n");}}
```

* Adding polynomials using linked list:

```
typedef struct link
 { int coeff; int pow;
   struct link *next;} poly;
void create_poly(poly **node)
{ int flag;
  int coeff, pow;
  temp = (poly*) malloc(sizeof(poly));
  *node =temp;
  do{
   printf(" Enter coefficient:");
   scanf("%d", &coeff);
    node→coeff = coeff;
   printf(" Enter power:");
   scanf("%d", &pow);
     temp → pow = pow;
     temp→ next = NULL;
```

```
    printf(" Continue adding more terms?
            (Y=1/N=0):");
    scanf("%d", &flag);
    if (flag)
   { temp->next = (poly *) malloc (sizeof(poly));
     temp = node->next;
     temp->next = NULL; }
   } while (flag); }

void show_poly (poly *node)
{ printf(" Polynomial :\n");
  while (node!=NULL)
  { printf ("%dx%d ", node->coeff,
                      node->pow);
    node = node->next;
    if (node!=NULL)
        printf("+ "); } }

void add_poly(poly **result,
              poly *poly1, poly *poly2)
{ poly *temp;
  temp = (poly *) malloc (sizeof (poly));
  temp->next = NULL;
  *result = temp;
  while (poly1 && poly2)
  { if (poly1->pow > poly2->pow)
    { temp->pow = poly1->pow;
      temp->coeff = poly1->coeff;
      poly1 = poly1->next; }
    else if(poly1->pow < poly2->pow)
    { temp->pow = poly2->pow;
      temp->coeff = poly2->coeff;
      poly2 = poly2->next; }
    else
    { temp->pow = poly1->pow;
      temp->coeff = poly1->coeff +
                    poly2->coeff;
      poly1 = poly1->next;
      poly2 = poly2->next; }
  if (poly1 && poly2)
  { temp->next = (poly *) malloc(sizeof(poly));
    temp = temp->next;
    temp->next = NULL; } }
```

```
while (poly1 || poly2)
{ temp->next = (poly*) malloc (sizeof(poly
  temp=temp->next;
  temp->next =NULL;
  if(poly1)
  { temp->pow = poly1->pow;
    temp->coeff = poly1->coeff;
    poly1 =poly1->next; }
  if(poly2)
  { temp->pow =poly2->pow;
    temp->coeff = poly2->coeff;
    poly2 = poly2->next; }
} }
```

*Circular Linked list:

```
Struct Node
{ int data;
  struct Node *next;
} *head=NULL;
Void insert(int value, int location)
{ struct Node *newNode;
  newNode =(struct Node *)malloc(sizeof(struct
Node
```

```
newNode->data=value;
if(head==NULL)
{ head = newNode;
  newNode->next=head; }
else
{ struct Node *temp = head;
  while (temp->data != location)
  { if(temp->next == head)
    { printf("Not found");
      goto EndFunction; }
  else
  {temp=temp->next;}
}
newNode->next =temp->next;
  temp->next=newNode;
  printf("Insertion success!."); }
EndFunction; }
void delete(int value)
{ if(head == NULL)
  { printf("List is Empty !."); }
  else
  { struct Node *temp1 =head, *temp2;
    while (temp1->data != value)
    { if (temp1->next ==head)
```

```
{printf("Not found");
  goto FunctionEnd;}
else
{temp2 = temp1;
 temp1 = temp1->next;}}

if(temp1->next == head)
 { head = NULL;
   free(temp1);}

else
{ if (temp1 == head)
   {temp2 = head;
    while(temp2->next != head)
      {temp2 = temp2->next;}
    head = head->next;
    temp2->next = head;
    free(temp1);}
else
{ if (temp1->next == head)
   {temp2->next = head;}

else
 {temp2->next = temp1->next;}
  free(temp1);}
}
FunctionEnd;}
```

* Doubly Linked list:

```
typedef struct node *nodePointer;
typedef struct
   {nodePointer llink;
    int data;
    nodePointer rlink;
   }node;

void dinsert(nodePointer node, nodePointer newnode)
{ newnode->llink = node;
  newnode->rlink = node->rlink;
  node->rlink->llink = newnode;
  node->rlink = newnode;}

void ddelete(nodePointer node, nodePointer d)
{ if(node == d)
    printf("Deletion of header node not permitted.\n");
  else
  {d->llink->rlink = d->rlink;
   d->rlink->llink = d->llink;
   free(d);}}
```

* Algorithm Specification:

→ Algorithm is finite set of instructions which are followed to accomplish a particular task.

* Properties:
  - → Input
  - → Output
  - → Definiteness
  - → Finiteness
  - → Effectiveness

* Binary Search Algorithm

```
While (there are more integers to check)
{ middle = (l+r)/2
  if (searchn < list[middle])
      r = middle - 1;
  else if (searchn == list[middle])
      return middle;
  else l = middle+1.
```

* Recursive binarysearch

```
int binsearch(int list[], int searchn,
              int l, int r)
{ int mid;
  if (l<=r)
  { mid = (l+r)/2;
    switch (COMPARE(list[middle], searchn))
    { case-1: return binsearch
                (list, searchn, mid+1);
      case 0: return mid;
      case 1: return binsearch(list, searchn,
                l, mid-1);
  return -1; }
```

* Data Abstraction:

→ An abstract data type is a data type, where only behaviour is defined but not implementation.

* Performance Analysis:

1) Space Complexity: amount of memory that it needs to run to completion.

$$S(P) = c + S_p(I)$$

c: const. $S_p(I)$: space for instance variables.

2) Time complexity: amount of computer time that it needs to run to completion.

$$T(P) = c + T_p(I)$$

c: Compile Time. $T_p(I)$: Run Time

→ Asymptotic notation:

1) O (Big "oh") ⇒ $f(n) = O(g(n))$ iff there exist +ve const. c & n_0 such that $f(n) \leq cg(n)$ $\forall$ $n, n \geq n_0$.

2) Omega ⇒ $f(n) = \Omega(g(n))$ iff there exist positive constants c & n_0 such that $f(n) \geq cg(n)$ $\forall n, n \geq n_0$

3) Theta ⇒ $f(n) = \theta(g(n))$ iff there exist +ve constants c_1, c_2, & n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ $\forall n, n \geq n_0$

*ARRAYS:

→ Collection of data of single type.

→ Structures are collection of data of same or different data types.

→ Union is similar to structure but the variables share their memory space.

→ Self-referential structures are one in which one or more of its components is a pointer to itself.

* Sparse Matrix:

→ Abstract data type used to store the matrix data in which no. of zeroes are more.

representation:

```
struct term
{ int row;
  int col;
  int value; };
```

* Transpose of a sparse matrix:

```
void transpose(term a[], term b[])
{int n,i,j, currentb;
  n = a[0].value;
  b[0].row = a[0].col;
  b[0].col = a[0].row;
  b[0].value = n;
  if(n>0)
 {currentb = 1;
   for(i=0; i<a[0].col; i++)
    { for(j=1; j<=n; j++)
      { if(a[j].col == i)
        { b[currentb].row = a[j].col;
          b[currentb].col = a[j].row
          b[currentb].value = a[j].value;
          currentb++; } } } }
}
```

2) *KMP algorithm:

```
int pmatch();
void fail();
int failure[100];
char string[100];
char pat[100];
int pmatch(char *string, char *pat)
{ int i=0, j=0;
  int lens = strlen(string);
  int lenp = strlen(pat);
  while(i<lens && j<lenp)
  { if(string[i]==pat[j])
    { i++; j++; }
    else if (j==0) i++;
    else j=failure[j-1]+1; }
  return ((j==lenp)? (i-lenp):-1);
}
void fail(char *pat)
{ int n=strlen(pat);
  failure[0]=-1;
  for(j=1; j<n; j++)
  { i=failure[j-1];
    while((pat[j]!=pat[i+1])&&(i>=0))
    { i=failure[i]; }
    if(pat[j]==pat[i+1])
    { failure[j]=i+1; }
    else failure[j]=-1;
  }
}
```