1. Stack Implementation using array.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAXSIZE 40

int SIZE;

struct stack
{
	int top;
	int STACK[MAXSIZE];
}s;
void initialize_stack()
{
	s.top = -1;
}
int full()
{
	if(s.top == SIZE-1)
		return (1);
	else
		return (0);
}
int empty()
{
	if(s.top == -1)
		return (1);
	else
		return (0);
}
int pop()
{
	int item;
	if(empty())
	{
		printf("Stack Underflow");
		return NULL;
	}
	else
```

```
    {
        item = s.STACK[s.top];
        s.top--;
        return item;
    }
}
void push(int item)
{
    if(full())
    {
        printf("Stack overflow.");
    }
    else
    {
        s.top++;
        s.STACK[s.top] = item;
    }
}
void display()
{
    int i;
    for(i=s.top; i>=0; i--)
    {
        if(!empty() && i==s.top)
            printf("top -> %d\n", s.STACK[i]);
        else
            printf("       %d\n", s.STACK[i]);
    }
}
int main()
{
    int n, item;
    initialize_stack();

    printf("\nEnter Stack Size:");
    scanf("%d", &SIZE);

     while(1)
       {
       printf("MENU- STACK OPERATIONS\n\n\n");
       printf("1. PUSH an ITEM\n");
```

```
        printf("2. POP an ITEM\n");
        printf("3. Exit\n");

        printf("\n______________________________");
        printf("\nStack [Size: %d]:\n", SIZE);
        display();
        printf("______________________________\n");

        printf("\n\nChoose one of the above option. [1-4]: ");
        scanf("%d",&n);

          switch (n)
          {
          case 1:
         printf("\nEnter an ITEM:");
         scanf("%d", &item);
         push(item);
         break;
          case 2:
         item = pop();
         if(item != NULL)
          printf("\nPopped ITEM from stack: %d", item);
         break;
          case 3:
         exit(0);
          default:
         printf("\nEnter correct option!Try again.");
          }

        printf("\n\n--");
        printf("\nContinue? Enter any key... ");
        getch();
        system("cls");
        }
     return 0;
}
```

2. Infix Evaluation

```
#include<iostream>
#include<stack>

using namespace std;
```

```
int pri(char ch)
{
	switch (ch)
	{
		case '(':
			return 1;
		case '+':
			//return 2;
		case '-':
			return 3;

		case '*':
			//return 4;
		case '/':
			return 5;

		case '^':
			return 6;
	}
	return -1;
}


float calculate(char op, float l , float r)
{
	if(op == '+')
	{
		return l + r;
	}
	else if(op == '-')
	{
		return l - r ;
	}
	else if(op == '*')\
	{
		return l * r;
	}
	else if(op == '/')
	{
		if(r > 0)
```

```
        {
            return l/r;
        }
        return 0;
    }
    else if(op == '^')
    {
        int b = l; // l is made int and stored at b
        int p = r; // r is made int and stored at p
        return b ^ p;
    }
    return -1;
}

int main()
{
char str[] = "3+4*5*(4+3)-1/2+1";
//char str[] = "3+4*5*4+3-1/2+1";
float l = sizeof(str)/sizeof(char);
int k = 0;
stack<char> s;
stack<float> op_s;
cout <<"InFix Expression: " << str << endl;
int i = 0;
while(str[i] != '\0')
{
    if(str[i] == '(')
    {
        s.push('(');
    }else if(str[i] == ')')
    {
        while(s.top() != '('){
            float r = op_s.top();
            op_s.pop();
            float l = op_s.top();
            op_s.pop();
            float re = calculate(s.top(),l,r);
            op_s.push(re);
            s.pop();
        }
        s.pop();
```

```
    }else if(str[i] == '+' || str[i] == '-' || str[i] == '*' ||
str[i] == '/' || str[i] == '^'){
        float pC = pri(str[i]);
        while(!s.empty() && pri(s.top()) >= pC){
            float r = op_s.top();
            op_s.pop();
            float l = op_s.top();
            op_s.pop();
            float re = calculate(s.top(),l,r);
            op_s.push(re);
            s.pop();
        }
        s.push(str[i]);
    }else{
        op_s.push(int(str[i])- 48);
    }
    i++;
}
while(!s.empty()){
    float r = op_s.top();
    op_s.pop();
    float l = op_s.top();
    op_s.pop();
    float re = calculate(s.top(),l,r);
    op_s.push(re);
    s.pop();
}
cout <<"Result: " << op_s.top() << endl;
return 0;
}
```

3. Queue implementation using array

```
#include <iostream>
#include <cstdlib>
using namespace std;

const int MAX_SIZE = 100;

class QueueOverFlowException
{
public:
```

```
   QueueOverFlowException()
   {
       cout << "Queue overflow" << endl;
   }
};

class QueueEmptyException
{
public:
   QueueEmptyException()
   {
       cout << "Queue empty" << endl;
   }
};

class ArrayQueue
{
private:
   int data[MAX_SIZE];
   int front;
   int rear;
public:
   ArrayQueue()
   {
       front = -1;
       rear = -1;
   }

   void Enqueue(int element)
   {
       // Don't allow the queue to grow more
       // than MAX_SIZE - 1
       if ( Size() == MAX_SIZE - 1 )
           throw new QueueOverFlowException();

       data[rear] = element;

       // MOD is used so that rear indicator
       // can wrap around
       rear = ++rear % MAX_SIZE;
   }
```

```
    int Dequeue()
    {
        if ( isEmpty() )
            throw new QueueEmptyException();

        int ret = data[front];

        // MOD is used so that front indicator
        // can wrap around
        front = ++front % MAX_SIZE;

        return ret;
    }

    int Front()
    {
        if ( isEmpty() )
            throw new QueueEmptyException();

        return data[front];
    }

    int Size()
    {
        return abs(rear - front);
    }

    bool isEmpty()
    {
        return ( front == rear ) ? true : false;
    }
};

int main()
{
    ArrayQueue q;
    try {
        if ( q.isEmpty() )
        {
            cout << "Queue is empty" << endl;
```

```
        }

        // Enqueue elements
        q.Enqueue(100);
        q.Enqueue(200);
        q.Enqueue(300);

        // Size of queue
        cout << "Size of queue = " << q.Size() << endl;

        // Front element
        cout << q.Front() << endl;

        // Dequeue elements
        cout << q.Dequeue() << endl;
        cout << q.Dequeue() << endl;
        cout << q.Dequeue() << endl;
    }
    catch (...) {
        cout << "Some exception occured" << endl;
    }
}
```

4. Tower of Hanoi using recursive function

```
#include <iostream>
using namespace std;

void towerOfHanoi(int n, char source_rod, char destination_rod,
char auxi_rod)
{
    if (n == 1)
    {
        cout << "Move disk 1 from rod " << source_rod <<" to rod
" << destination_rod<<endl;
        return;
    }
    towerOfHanoi(n - 1, source_rod, auxi_rod, destination_rod);
// step1
    cout << "Move disk " << n << " from rod " << source_rod <<"
to rod " << destination_rod << endl; //step2
```

```
    towerOfHanoi(n - 1, auxi_rod, destination_rod, source_rod);
// step3
}


int main()
{
    int n = 1; // Number of disks
    towerOfHanoi(n, 'S', 'D', 'A'); // S = source rod, D =
Destination rod and A auxiliary rod
    return 0;
}
```

5. Tree traversal implementation

```
// C program for different tree traversals
#include <iostream>
using namespace std;

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node
{
    int data;
    struct Node* left, *right;
    Node(int data)
    {
        this->data = data;
        left = right = NULL;
    }
};

/* Given a binary tree, print its nodes according to the
"bottom-up" postorder traversal. */
void printPostorder(struct Node* node)
{
    if (node == NULL)
        return;

    // first recur on left subtree
    printPostorder(node->left);
```

```
    // then recur on right subtree
    printPostorder(node->right);

    // now deal with the node
    cout << node->data << " ";
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    cout << node->data << " ";

    /* now recur on right child */
    printInorder(node->right);
}

/* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct Node* node)
{
    if (node == NULL)
        return;

    /* first print data of node */
    cout << node->data << " ";

    /* then recur on left sutree */
    printPreorder(node->left);

    /* now recur on right subtree */
    printPreorder(node->right);
}

/* Driver program to test above functions*/
```

```
int main()
{
    struct Node *root = new Node(1);
    root->left             = new Node(2);
    root->right         = new Node(3);
    root->left->left     = new Node(4);
    root->left->right = new Node(5);

    cout << "\nPreorder traversal of binary tree is \n";
    printPreorder(root);

    cout << "\nInorder traversal of binary tree is \n";
    printInorder(root);

    cout << "\nPostorder traversal of binary tree is \n";
    printPostorder(root);
    return 0;
}
```

6. Implementation of insertion sort

```
#include <bits/stdc++.h>
using namespace std;

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
```

```
    }
}
void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

7. Implementation of binary search

```
#include <bits/stdc++.h>
using namespace std;

// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);
```

```
        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    return -1;
}

int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? cout << "Element is not present in array"
                   : cout << "Element is present at index " <<
result;
    return 0;
}
```

8. Transitive Closure using Floyd Warshall Algorithm

```
#include<stdio.h>

// Number of vertices in the graph
#define V 4

// A function to print the solution matrix
void printSolution(int reach[][V]);

// Prints transitive closure of graph[][] using Floyd Warshall
algorithm
void transitiveClosure(int graph[][V])
{
    /* reach[][] will be the output matrix that will finally
have the
       shortest distances between every pair of vertices */
    int reach[V][V], i, j, k;
```

```
    /* Initialize the solution matrix same as input graph
matrix. Or
       we can say the initial values of shortest distances are
based
       on shortest paths considering no intermediate vertex. */
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            reach[i][j] = graph[i][j];

    /* Add all vertices one by one to the set of intermediate
vertices.
      ---> Before start of a iteration, we have reachability
values for
           all pairs of vertices such that the reachability
values
           consider only the vertices in set {0, 1, 2, .. k-1}
as
           intermediate vertices.
      ----> After the end of a iteration, vertex no. k is added
to the
            set of intermediate vertices and the set becomes {0,
1, .. k} */
    for (k = 0; k < V; k++)
    {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++)
        {
            // Pick all vertices as destination for the
            // above picked source
            for (j = 0; j < V; j++)
            {
                // If vertex k is on a path from i to j,
                // then make sure that the value of reach[i][j]
is 1
                reach[i][j] = reach[i][j] || (reach[i][k] &&
reach[k][j]);
            }
        }
    }

    // Print the shortest distance matrix
```

```
    printSolution(reach);
}

/* A utility function to print solution */
void printSolution(int reach[][V])
{
    printf ("Following matrix is transitive closure of the given
graph\n");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
            printf ("%d ", reach[i][j]);
        printf("\n");
    }
}

// driver program to test above function
int main()
{
    /* Let us create the following weighted graph
            10
       (0)------->(3)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (1)------->(2)
            3           */
    int graph[V][V] = { {1, 1, 0, 1},
                        {0, 1, 1, 0},
                        {0, 0, 1, 1},
                        {0, 0, 0, 1}
                      };

    // Print the solution
    transitiveClosure(graph);
    return 0;
}
```

9. Kruskal's algorithm to find Minimum Spanning Tree

```
#include <bits/stdc++.h>
```

```
using namespace std;

// a structure to represent a weighted edge in graph
class Edge
{
    public:
    int src, dest, weight;
};

// a structure to represent a connected, undirected
// and weighted graph
class Graph
{
    public:
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    // Since the graph is undirected, the edge
    // from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    Edge* edge;
};

// Creates a graph with V vertices and E edges
Graph* createGraph(int V, int E)
{
    Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;

    graph->edge = new Edge[E];

    return graph;
}

// A structure to represent a subset for union-find
class subset
{
    public:
    int parent;
```

```
    int rank;
};

// A utility function to find set of an element i
// (uses path compression technique)
int find(subset subsets[], int i)
{
    // find root and make root as parent of i
    // (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high
    // rank tree (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and
    // increment its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
```

```
{
    Edge* a1 = (Edge*)a;
    Edge* b1 = (Edge*)b;
    return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST(Graph* graph)
{
    int V = graph->V;
    Edge result[V]; // Tnis will store the resultant MST
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges

    // Step 1: Sort all the edges in non-decreasing
    // order of their weight. If we are not allowed to
    // change the given graph, we can create a copy of
    // array of edges
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]),
myComp);

    // Allocate memory for creating V ssubsets
    subset *subsets = new subset[( V * sizeof(subset) )];

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1 && i < graph->E)
    {
        // Step 2: Pick the smallest edge. And increment
        // the index for next iteration
        Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);
```

```
        // If including this edge does't cause cycle,
        // include it in result and increment the index
        // of result for next edge
        if (x != y)
        {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
        // Else discard the next_edge
    }

    // print the contents of result[] to display the
    // built MST
    cout<<"Following are the edges in the constructed MST\n";
    for (i = 0; i < e; ++i)
        cout<<result[i].src<<" -- "<<result[i].dest<<" ==
"<<result[i].weight<<endl;
    return;
}

// Driver code
int main()
{
    /* Let us create following weighted graph
            10
        0--------1
        | \ |
    6| 5\ |15
        | \ |
        2--------3
            4 */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    Graph* graph = createGraph(V, E);


    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;
```

```
    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 15;

    // add edge 2-3
    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;
    graph->edge[4].weight = 4;

    KruskalMST(graph);

    return 0;
}
```

10. Dijkstra's single source shortest path algorithm

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

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum distance
value, from
// the set of vertices not yet included in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
```

```
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

// A utility function to print the constructed distance array
int printSolution(int dist[])
{
    printf("Vertex \t\t Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t\t %d\n", i, dist[i]);
}

// Function that implements Dijkstra's single source shortest
path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
    int dist[V]; // The output array.  dist[i] will hold the
shortest
    // distance from src to i

    bool sptSet[V]; // sptSet[i] will be true if vertex i is
included in shortest
    // path tree or shortest distance from src to i is finalized

    // Initialize all distances as INFINITE and stpSet[] as
false
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum distance vertex from the set of
vertices not
        // yet processed. u is always equal to src in the first
iteration.
```

```
        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = true;

        // Update dist value of the adjacent vertices of the
picked vertex.
        for (int v = 0; v < V; v++)

            // Update dist[v] only if is not in sptSet, there is
an edge from
            // u to v, and total weight of path from src to  v
through u is
            // smaller than current value of dist[v]
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist);
}

// driver program to test above function
int main()
{
    /* Let us create the example graph discussed above */
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    dijkstra(graph, 0);

    return 0;
}
```