```
    for(int i=0;i<n;i++)
    {
        printf("%d ", arr[i]);
    }
}
```

```
Enter the length of array: 5
Enter elements: 2 4 5 3 1
Before sorting: 2 4 5 3 1
After sorting: 1 2 3 4 5
```

---

2) Selection Sort

```
#include<stdio.h>
void selectionSort(int a[], int n)
{
    for(int i=0;i<n-1;i++)
    {
        int min = i;
        for(int j=i+1;j<n;j++)
        {
            if(a[min]>a[j])
            min = j;
        }
        int temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}
int main()
{
    printf("Enter the length of array: ");
    int n;
    scanf("%d", &n);
    int arr[n];
    printf("Enter elements: ");
    for(int i=0;i<n;i++)
    {
        scanf("%d", &arr[i]);
```

```
    }
    printf("Before sorting: ");
    for(int i=0;i<n;i++)
    {
        printf("%d ", arr[i]);
    }
    selectionSort(arr, n);
    printf("\nAfter sorting: ");
    for(int i=0;i<n;i++)
    {
        printf("%d ", arr[i]);
    }
}
```

```
Enter the length of array: 5
Enter elements: 2 4 1 5 3
Before sorting: 2 4 1 5 3
After sorting: 1 2 3 4 5
```

---

3) Insertion Sort

```
#include<stdio.h>
void insertionSort(int a[], int n)
{
    for(int i=1;i<n;i++)
    {
        int key = a[i];
        int j = i-1;
        while(a[j]>key&&j>=0)
        {
            a[j+1]=a[j];
            j--;
        }
        a[j+1] = key;
    }
}
int main()
{
    printf("Enter the length of array: ");
```

```
    int n;
    scanf("%d", &n);
    int arr[n];
    printf("Enter elements: ");
    for(int i=0;i<n;i++)
    {
        scanf("%d", &arr[i]);
    }
    printf("Before sorting: ");
    for(int i=0;i<n;i++)
    {
        printf("%d ", arr[i]);
    }
    insertionSort(arr, n);
    printf("\nAfter sorting: ");
    for(int i=0;i<n;i++)
    {
        printf("%d ", arr[i]);
    }
}
```

```
Enter the length of array: 5
Enter elements: 3 1 2 4 5
Before sorting: 3 1 2 4 5
After sorting: 1 2 3 4 5
```

---

4) Merge Sort

```
#include <stdio.h>
void merge(int a[], int l, int u, int p)
{
    int n1 = p-l+1;
    int n2 = u-p;
    int left[n1], right[n2];
    for(int i=0;i<n1;i++)
    {
        left[i] = a[l+i];
    }
    for(int i=0;i<n2;i++)
```

```
    {
        right[i] = a[p+1+i];
    }
    int i = 0, j = 0, k=l;
    while(i<n1||j<n2){
        if(i==n1)
        {
            a[k] = right[j];
            j++;
        }
        else if(j==n2)
        {
            a[k] = left[i];
            i++;
        }
        else
        {
             if(left[i]>right[j])
            {
                a[k] = right[j];
                j++;
            }
            else
            {
                a[k] = left[i];
                i++;
            }
        }
        k++;
    }
}
void mergesort(int a[], int l, int u)
{
    if(l<u)
    {
        int p = (l+u)/2;
        mergesort(a,l,p);
        mergesort(a,p+1,u);
        merge(a,l,u,p);
    }
```

```
}
int main()
{
    printf("Enter the length of array: ");
    int n;
    scanf("%d", &n);
    int arr[n];
    printf("Enter elements: ");
    for(int i=0;i<n;i++)
    {
        scanf("%d", &arr[i]);
    }
    printf("Before sorting: ");
    for(int i=0;i<n;i++)
    {
        printf("%d ", arr[i]);
    }
    mergesort(arr, 0, n-1);
    printf("\nAfter sorting: ");
    for(int i=0;i<n;i++)
    {
        printf("%d ", arr[i]);
    }
}
```

```
Enter the length of array: 6
Enter elements: 5 4 1 2 6 3
Before sorting: 5 4 1 2 6 3
After sorting: 1 2 3 4 5 6
```

---

5) Quick Sort

```
#include <stdio.h>
int partition(int A[], int l, int u)
{
    int pivot = u;
    int i = l, temp;
    for(int j=l;j<u;j++)
    {
```

```
        if(A[j]<A[pivot])
        {
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
            i++;
        }
    }
    temp = A[i];
    A[i] = A[pivot];
    A[pivot] = temp;
    return i;
}
void quicksort(int A[], int l, int u)
{
    if(l<u)
    {
        int pivot = partition(A, l, u);
        quicksort(A, l, pivot-1);
        quicksort(A, pivot+1, u);
    }
}
int main()
{
    printf("Enter the length of array: ");
    int n;
    scanf("%d", &n);
    int arr[n];
    printf("Enter elements: ");
    for(int i=0;i<n;i++)
    {
        scanf("%d", &arr[i]);
    }
    printf("Before sorting: ");
    for(int i=0;i<n;i++)
    {
        printf("%d ", arr[i]);
    }
    quicksort(arr, 0, n-1);
    printf("\nAfter sorting: ");
```

```
    for(int i=0;i<n;i++)
    {
        printf("%d ", arr[i]);
    }
}
```

```
Enter the length of array: 5
Enter elements: 3 45 4 15 6
Before sorting: 3 45 4 15 6
After sorting: 3 4 6 15 45
```

---

6) Heap Sort

```
#include<stdio.h>

void maxHeapify(int a[], int n, int i)
{
    int left = 2*i+1, right = 2*i+2;
    int largest = i;
    if(left<n&&a[left]>a[largest])
        largest=left;
    if(right<n&&a[right]>a[largest])
        largest=right;

    if(largest!=i)
    {
        int temp = a[largest];
        a[largest] = a[i];
        a[i] = temp;
        maxHeapify(a, n, largest);
    }
}
void heapSort(int a[], int n)
{
    for(int i=n/2-1;i>=0;i--)
    {
        maxHeapify(a, n, i);
    }
    for(int i=n-1;i>=0;i--)
```

```
    {
        int temp = a[0];
        a[0] = a[i];
        a[i] = temp;
        maxHeapify(a, i, 0);
    }
}
int main()
{
    printf("Enter the length of array: ");
    int n;
    scanf("%d", &n);
    int arr[n];
    printf("Enter elements: ");
    for(int i=0;i<n;i++)
    {
        scanf("%d", &arr[i]);
    }
    printf("Before sorting: ");
    for(int i=0;i<n;i++)
    {
        printf("%d ", arr[i]);
    }
    heapSort(arr, n);
    printf("\nAfter sorting: ");
    for(int i=0;i<n;i++)
    {
        printf("%d ", arr[i]);
    }
}
```

```
Enter the length of array: 6
Enter elements: 45
25
87
36
14
20
Before sorting: 45 25 87 36 14 20
After sorting: 14 20 25 36 45 87
```

7) Count Sort

```c
#include<stdio.h>
void countSort(int a[], int n)
{
    int max = 0;
    for(int i=0;i<n;i++)
    {
        if(max<a[i])
        max = a[i];
    }
    int b[n], count[max+1];
    for(int i=0;i<=max;i++)
    {
        count[i]=0;
    }
    for(int i=0;i<n;i++)
    {
        count[a[i]]++;
    }
    for(int i=1;i<=max;i++)
    {
        count[i] += count[i-1];
    }
    for(int i=n-1;i>=0;i--)
    {
        b[count[a[i]]-1] = a[i];
        count[a[i]]--;
    }
    for(int i=0;i<n;i++)
    {
        a[i] = b[i];
    }
}
int main()
{
    printf("Enter the length of array: ");
    int n;
```

```
    scanf("%d", &n);
    int arr[n];
    printf("Enter elements: ");
    for(int i=0;i<n;i++)
    {
        scanf("%d", &arr[i]);
    }
    printf("Before sorting: ");
    for(int i=0;i<n;i++)
    {
        printf("%d ", arr[i]);
    }
    countSort(arr, n);
    printf("\nAfter sorting: ");
    for(int i=0;i<n;i++)
    {
        printf("%d ", arr[i]);
    }
}
```

```
Enter the length of array: 6
Enter elements: 23
8
1
7
2
10
Before sorting: 23 8 1 7 2 10
After sorting: 1 2 7 8 10 23
```

---

8) Radix Sort

```
#include<stdio.h>

void countSort(int a[], int exp, int n)
{
    int b[n], count[10];
    for(int i=0;i<10;i++)
        count[i] = 0;
    for(int i=0;i<n;i++)
```

```
    {
        count[(a[i]/exp)%10]++;
    }
    for(int i=1;i<=9;i++)
    {
        count[i] = count[i]+count[i-1];
    }
    for(int i=n-1;i>=0;i--)
    {
        b[count[(a[i]/exp)%10]-1] = a[i];
        count[(a[i]/exp)%10]--;
    }
    for(int i=0;i<n;i++)
    {
        a[i] = b[i];
    }
}
void radixSort(int a[], int n)
{
    int max = 0;
    for(int i=0;i<n;i++)
    {
        if(max<a[i])
            max = a[i];
    }
    for(int exp=1;max/exp>0;exp*=10)
        countSort(a, exp, n);
}
int main()
{
    printf("Enter the length of array: ");
    int n;
    scanf("%d", &n);
    int arr[n];
    printf("Enter elements: ");
    for(int i=0;i<n;i++)
    {
        scanf("%d", &arr[i]);
    }
    printf("Before sorting: ");
```

```
    for(int i=0;i<n;i++)
    {
        printf("%d ", arr[i]);
    }
    radixSort(arr, n);
    printf("\nAfter sorting: ");
    for(int i=0;i<n;i++)
    {
        printf("%d ", arr[i]);
    }
}
```

```
Enter the length of array: 6
Enter elements: 789
456
123
102
35
68
Before sorting: 789 456 123 102 35 68
After sorting: 35 68 102 123 456 789
```

---

9) Infix To Postfix

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

#define N 100
typedef struct Stack
{
    char array[N];
    int top;
}Stack;
Stack stack;
void push(char c)
{
    stack.array[++stack.top] = c;
}
char peek()
{
```

```
    return stack.array[stack.top];
}
void pop()
{
    stack.array[stack.top--];
}
int priority(char ch)
{
    switch(ch)
    {
        case '(': return 0;
        case '+': return 1;
        case '-': return 1;
        case '*': return 2;
        case '/': return 2;
        case '^': return 3;
    }
    return -1;
}
void infixToPostfix(char expression[])
{
    stack.top = -1;
    push('(');
    strcat(expression,")");
    int i = 0;
    while(expression[i]!='\0')
    {
        char ch = expression[i];
        if(ch=='(')
        {
            push('(');
        }
        else if(ch==')')
        {
            while(peek()!='(')
            {
                printf("%c", peek());
                pop();
            }
            pop();
```

```
        }
        else if((ch>=48&&ch<=57)||(ch>=65&&ch<=90)||(ch>=97&&ch<=122))
        {
            printf("%c", ch);
        }
        else
        {
            while(priority(ch)<=priority(peek()))
            {
                printf("%c", peek());
                pop();
            }
            push(ch);
        }
        i++;
    }
}
int main()
{
    printf("Enter infix expression: ");
    char expression[N];
    gets(expression);
    infixToPostfix(expression);
}
```

```
Enter infix expression: A+B*C-(D-E)
ABC*+DE--
```

---

10) Postfix To Infix

```
#include<stdio.h>
#include<math.h>
#define N 100
typedef struct Stack
{
    int array[N];
    int top;
}Stack;
```

```
Stack stack;
void push(int c)
{
    stack.array[++stack.top] = c;
}
int peek()
{
    return stack.array[stack.top];
}
int  pop()
{
    return stack.array[stack.top--];
}

void postfixToInfix(char *expression)
{
    int i=0;
    int b, a;
    while(expression[i]!='\0')
    {
        char ch = expression[i];
        if((ch>=48&&ch<=57))
        {
            push((int)ch-48);
        }
        else
        {
            b = pop();
            a = pop();
            switch(ch)
            {
                case '+': push(a+b);
                break;
                case '-': push(b-a);
                break;
                case '*': push(b*a);
                break;
                case '/': push(b/a);
                break;
                case '^': push(pow(b,a));
```

```
                break;
            }
        }
        i++;
    }
    printf("%d", pop());
}
int main()
{
    stack.top = -1;
    printf("Enter postfix expression: ");
    char expression[N];
    gets(expression);
    postfixToInfix(expression);
}
```

```
Enter postfix expression: 123*+45--
-6
```

---

11) Binary Search Tree

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node
{
    int value;
    struct Node *left, *right;
}Node;
Node* root;
Node* minimum(Node *node)
{
    while(node->left!=NULL)
        node = node->left;
    return node;
}
Node* createNode(int val)
{
    Node *node = (Node*)malloc(sizeof(Node));
```

```
    node->value = val;
    node->left = node->right = NULL;
    return node;
}
Node* successor(Node* node)
{
    if(node->right!=NULL)
        return minimum(node->right);
    Node *success = NULL, *newroot = root;
    while(newroot!=node)
    {
        if(newroot->value>node->value)
        {
            success = newroot;
            newroot = newroot->left;
        }
        else if(newroot->value<node->value)
        {
            newroot = newroot->right;
        }
    }
    return success;
}
void insert(Node *node)
{
    Node *child = root, *parent = NULL;
    while(child!=NULL)
    {
        parent = child;
        if(node->value>child->value)
            child = child->right;
        else
            child = child->left;
    }
    if(parent==NULL)
        root = node;
    else if(parent->value>node->value)
        parent->left = node;
    else
        parent->right = node;
```

```
}
Node* delete(Node *node, int val)
{
    if(node==NULL)
        printf("Error, could not find %d\n", val);
    else if(node->value<val)
        node->right = delete(node->right, val);
    else if(node->value>val)
        node->left = delete(node->left, val);
    else if(node->left!=NULL&&node->right!=NULL)
    {
        Node *temp = successor(node);
        node->value = temp->value;
        node->right = delete(node->right, temp->value);
    }
    else
    {
        Node *temp = node;
        if(node->left!=NULL)
        {
            node = node->left;
        }
        else if(node->right!=NULL)
        {
            node = node->right;
        }
        else
        {
            free(temp);
            return NULL;
        }
    }
}
void inorder(Node *node)
{
    if(node!=NULL)
    {
        inorder(node->left);
        printf("%d ", node->value);
        inorder(node->right);
```

```
    }
}
int main()
{
    printf("---------------Program Starts--------------");
    while(1)
    {
        printf("\n1. Insert\n2. Delete\n3. Print Tree\n4.
Exit\n--------Enter Choice Number---------\n");
        int element;
        int choice;
        scanf("%d", &choice);
        switch(choice)
        {
            case 1: printf("Enter element to insert: ");
                    scanf("%d", &element);
                    Node *newNode = createNode(element);
                    insert(newNode);
                    break;
            case 2: printf("Enter element to delete: ");
                    scanf("%d", &element);
                    delete(root, element);
                    break;
            case 3: printf("Printing Tree: ");
                    inorder(root);
                    printf("\n");
                    break;
            case 4: printf("Exiting application...");
                    exit(1);
        }
    }
}
```

```
---------------Program Starts--------------
1. Insert
2. Delete
3. Print Tree
4. Exit
--------Enter Choice Number---------
1
Enter element to insert: 10

1. Insert
2. Delete
3. Print Tree
4. Exit
--------Enter Choice Number---------
1
Enter element to insert: 12

1. Insert
2. Delete
3. Print Tree
4. Exit
--------Enter Choice Number---------
1
Enter element to insert: 6

--------Enter Choice Number---------
2
Enter element to delete: 10

1. Insert
2. Delete
3. Print Tree
4. Exit
--------Enter Choice Number---------
3
Printing Tree: 6 8 12 45

1. Insert
2. Delete
3. Print Tree
4. Exit
--------Enter Choice Number---------
4
Exiting application...
```

12) Breadth First Search

```c
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40

typedef struct queue {
  int items[SIZE];
  int front;
  int rear;
}queue;

queue* createQueue();
void enqueue(queue* q, int);
int dequeue(queue* q);
void display(queue* q);
int isEmpty(queue* q);
void printQueue(struct queue* q);

typedef struct node {
  int vertex;
  struct node* next;
}node;

node* createNode(int);

typedef struct Graph {
  int numVertices;
  node** adjLists;
  int* visited;
}Graph;

void bfs(Graph* graph, int startVertex) {
  queue* q = createQueue();

  graph->visited[startVertex] = 1;
  enqueue(q, startVertex);

  while (!isEmpty(q)) {
    printQueue(q);
    int currentVertex = dequeue(q);
```

```
    printf("Visited %d\n", currentVertex);

    node* temp = graph->adjLists[currentVertex];

    while (temp) {
      int adjVertex = temp->vertex;

      if (graph->visited[adjVertex] == 0) {
        graph->visited[adjVertex] = 1;
        enqueue(q, adjVertex);
      }
      temp = temp->next;
    }
  }
}
node* createNode(int v) {
  node* newNode = malloc(sizeof(node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}
Graph* createGraph(int vertices) {
  Graph* graph = malloc(sizeof(Graph));
  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices * sizeof(node*));
  graph->visited = malloc(vertices * sizeof(int));

  int i;
  for (i = 0; i < vertices; i++) {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }

  return graph;
}

void addEdge(Graph* graph, int src, int dest) {
  node* newNode = createNode(dest);
  newNode->next = graph->adjLists[src];
```

```
  graph->adjLists[src] = newNode;

  newNode = createNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}
queue* createQueue() {
  queue* q = malloc(sizeof(queue));
  q->front = -1;
  q->rear = -1;
  return q;
}

int isEmpty(queue* q) {
  if (q->rear == -1)
    return 1;
  else
    return 0;
}

void enqueue(queue* q, int value) {
  if (q->rear == SIZE - 1)
    printf("\nQueue is Full!!");
  else {
    if (q->front == -1)
      q->front = 0;
    q->rear++;
    q->items[q->rear] = value;
  }
}
int dequeue( queue* q) {
  int item;
  if (isEmpty(q)) {
    printf("Queue is empty");
    item = -1;
  } else {
    item = q->items[q->front];
    q->front++;
    if (q->front > q->rear) {
      printf("Resetting queue ");
```

```
      q->front = q->rear = -1;
    }
  }
  return item;
}

void printQueue(queue* q) {
  int i = q->front;

  if (isEmpty(q)) {
    printf("Queue is empty");
  } else {
    printf("\nQueue contains \n");
    for (i = q->front; i < q->rear + 1; i++) {
      printf("%d ", q->items[i]);
    }
  }
}

int main() {
    int vertices, key, edges, u, v;
    printf("Enter number of vertices for the graph: ");
    scanf("%d", &vertices);
    Graph* graph = createGraph(vertices);

    printf("Enter the number of edges: ");
    scanf("%d", &edges);
    printf("Enter the vertices on the edges: ");
    for(int i=0;i<edges;i++)
    {
        scanf("%d %d", &u, &v);
        addEdge(graph, u, v);
    }
    bfs(graph, 0);

    return 0;
}
```

```
Enter number of vertices for the graph: 5
Enter the number of edges: 5
Enter the vertices on the edges: 1 2
0 1
2 0
1 3
3 4

Queue contains
0 Resetting queue Visited 0

Queue contains
2 1 Visited 2

Queue contains
1 Resetting queue Visited 1

Queue contains
3 Resetting queue Visited 3

Queue contains
4 Resetting queue Visited 4
```

---

13) Depth First Search

```c
#include<stdio.h>
#include<stdlib.h>
typedef struct Node
{
    int key;
    struct Node* next;
}Node;

typedef struct Graph
{
    int vertices;
    Node** adjList;
    int* visited;
}Graph;

int search(Graph* graph, int val)
```

```
{
    for(int i=0;i<graph->vertices;i++)
    {
        if(graph->adjList[i]->key==val)
        return i;
    }
    return -1;
}

Node* createNode(int src)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->key = src;
    node->next = NULL;
    return node;
}

Graph* addEdge(Graph *graph, int src, int trg)
{
    int index = search(graph, src);
    Node* newNode = createNode(trg);
    newNode->next = graph->adjList[index]->next;
    graph->adjList[index]->next = newNode;

    index = search(graph, trg);
    newNode = createNode(src);
    newNode->next = graph->adjList[index]->next;
    graph->adjList[index]->next = newNode;

    return graph;
}

void DFSVisit(Graph* graph, int u)
{
    int index = search(graph, u);
    graph->visited[index] = 1;
    printf("%d ", u);
    Node *temp = graph->adjList[index]->next;
    while(temp!=NULL)
    {
```

```
        if(graph->visited[search(graph, temp->key)]==0)
        {
            DFSVisit(graph, temp->key);
        }
        temp = temp->next;
    }
}

void DFS(Graph *graph)
{
    for(int i=0;i<graph->vertices;i++)
    {
        if(graph->visited[i]==0)
        DFSVisit(graph, graph->adjList[i]->key);
    }
}

void printGraph(Graph *graph)
{
    for(int i=0;i<graph->vertices;i++)
    {
        Node* temp = graph->adjList[i]->next;
        printf("%d->", graph->adjList[i]->key);
        while(temp!=NULL)
        {
            printf("%d ", temp->key);
            temp = temp->next;
        }
        printf("\n");
    }
}

int main()
{
    int v1, v2;
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    printf("Enter number of vertices: ");
    scanf("%d", &graph->vertices);
    graph->adjList = (Node**)malloc(graph->vertices*sizeof(Node*));
    graph->visited = (int*)malloc(graph->vertices*sizeof(int));
```

```
    printf("Enter vertices: ");
    for(int i=0;i<graph->vertices;i++)
    {
        graph->adjList[i] = (Node*)malloc(sizeof(Node));
        scanf("%d", &graph->adjList[i]->key);
        graph->adjList[i]->next = NULL;
        graph->visited[i] = 0;
    }
    int edges;
    printf("Enter number of edges: ");
    scanf("%d", &edges);
    printf("Enter vertex on the edges: ");
    for(int i=0;i<edges;i++)
    {
        scanf("%d %d", &v1, &v2);
        graph = addEdge(graph, v1, v2);
    }
    printGraph(graph);
    DFS(graph);
}
```

```
Enter number of vertices: 5
Enter vertices: 1 2 3 4 5
Enter number of edges: 5
Enter vertex on the edges: 1 2
2 3
3 4
4 5
2 4
1->2
2->4 3 1
3->4 2
4->2 5 3
5->4
1 2 4 5 3
```

---

14) Prim's Minimum Spanning Tree

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

```c
#define MAX 100

typedef struct Node
{
    int key;
    int weight;
    struct Node* next;
}Node;
typedef struct PriorityQueue
{
    Node* array[MAX];
    int size;
}PriorityQueue;

typedef struct Graph
{
    int vertices;
    Node** adjList;
}Graph;

PriorityQueue *queue;

void enqueue(Node* n)
{
    if(queue->size==MAX)
        return;

    queue->array[queue->size++] = n;
}
int isEmpty()
{
    if(queue->size==0)
        return 1;
    return 0;
}
void minHeapify(int i)
{
    int smallest = i, left = 2*i+1, right = 2*i+2;
```

```
if(left<queue->size&&queue->array[left]->weight<queue->array[smallest]->we
ight)
        smallest = left;
    else
if(right<queue->size&&queue->array[right]->weight<queue->array[smallest]->
weight)
        smallest = right;

    if(smallest!=i)
    {
        Node* temp = queue->array[smallest];
        queue->array[smallest] = queue->array[i];
        queue->array[i] = temp;
        minHeapify(smallest);
    }
}
void decreaseKey(Node *temp, int val)
{
    int index = 0;
    for(int i=0;i<queue->size;i++)
    {
        if(queue->array[i]->key==temp->key)
        {
            index = i;
            break;
        }
    }
    if(queue->array[index]->key<val)
        return;

while(index>0&&queue->array[index]->key<queue->array[(index-1)/2]->key)
    {
        Node* swap = queue->array[index];
        queue->array[index] = queue->array[(index-1)/2];
        queue->array[(index-1)/2] = swap;
        index = (index-1)/2;
    }
}
int getMin()
```

```
{
    if(queue->size==0)
        return -1;
    Node* temp = queue->array[0];
    queue->array[0] = queue->array[queue->size-1];
    minHeapify(0);
    queue->size--;
    return temp->key;
}
void buildHeap()
{
    for(int i=queue->size/2-1;i>=0;i--)
    {
        minHeapify(i);
    }
}

int search(Graph* graph, int val)
{
    for(int i=0;i<graph->vertices;i++)
    {
        if(graph->adjList[i]->key==val)
        return i;
    }
    return -1;
}

Node* createNode(int trg, int weight)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->key = trg;
    node->next = NULL;
    node->weight = weight;
    return node;
}
Graph* addEdge(Graph *graph, int src, int trg, int weight)
{
    int index = search(graph, src);
    Node* newNode = createNode(trg, weight);
    newNode->next = graph->adjList[index]->next;
```

```
    graph->adjList[index]->next = newNode;

    return graph;
}
void printGraph(Graph *graph)
{
    for(int i=0;i<graph->vertices;i++)
    {
        Node* temp = graph->adjList[i]->next;
        printf("%d->", graph->adjList[i]->key);
        while(temp!=NULL)
        {
            int index = search(graph, temp->key);
            printf("%d (wt: %d), ", temp->key, temp->weight);
            temp = temp->next;
        }
        printf("\n");
    }
}
void Prims(Graph *graph, int source)
{
    int parent[graph->vertices], mst[graph->vertices];
    for(int i=0;i<graph->vertices;i++)
    {
        parent[i] = -1;
        mst[i] = 0;
        graph->adjList[i]->weight = INT_MAX;
        enqueue(graph->adjList[i]);
    }
    int index = search(graph, source);
    graph->adjList[index]->weight = 0;
    mst[index] = 1;
    buildHeap();
    while(!isEmpty())
    {
        index = search(graph, getMin());
        Node* temp = graph->adjList[index]->next;
        mst[index] = 1;
        while(temp!=NULL)
        {
```

```
            int index1 = search(graph, temp->key);
            if(!mst[index1]&&graph->adjList[index1]->weight>temp->weight)
            {
                parent[index1] = graph->adjList[index]->key;
                graph->adjList[index1]->weight = temp->weight;
                decreaseKey(graph->adjList[index1], temp->weight);
            }
            temp = temp->next;
        }
    }
    int sum = 0;
    for(int i=0;i<graph->vertices;i++)
    {
        sum += graph->adjList[i]->weight;
    }
    printf("Total cost of tree: %d", sum);
}

int main()
{
    int v1, v2, w;
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    printf("Enter number of vertices: ");
    scanf("%d", &graph->vertices);
    queue = (PriorityQueue*)malloc(sizeof(PriorityQueue));
    queue->size = 0;
    graph->adjList = (Node**)malloc(graph->vertices*sizeof(Node*));
    printf("Enter vertices: ");
    for(int i=0;i<graph->vertices;i++)
    {
        graph->adjList[i] = (Node*)malloc(sizeof(Node));
        scanf("%d", &graph->adjList[i]->key);
        graph->adjList[i]->next = NULL;
    }
    int edges;
    printf("Enter number of edges: ");
    scanf("%d", &edges);
    printf("Enter vertex on the edges and the weight: ");
    for(int i=0;i<edges;i++)
    {
```

```
        scanf("%d %d %d", &v1, &v2, &w);
        graph = addEdge(graph, v1, v2, w);
        graph = addEdge(graph, v2, v1, w);
    }
    printGraph(graph);
    printf("\nEnter the vertex to start with: ");
    scanf("%d", &w);
    Prims(graph, w);
}
```

```
Enter number of vertices: 5
Enter vertices: 1 2 3 4 5
Enter number of edges: 5
Enter vertex on the edges and the weight: 1 2 3
2 3 4
3 4 5
2 5 1
1 3 4
1->3 (wt: 4), 2 (wt: 3),
2->5 (wt: 1), 3 (wt: 4), 1 (wt: 3),
3->1 (wt: 4), 4 (wt: 5), 2 (wt: 4),
4->3 (wt: 5),
5->2 (wt: 1),

Enter the vertex to start with: 2
Total cost of tree: 13
```

---

15) Dijkstra Shortest Path

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 100

typedef struct Node
{
    int key;
    int distance;
    struct Node* next;
}Node;
```

```
typedef struct PriorityQueue
{
    Node* array[MAX];
    int size;
}PriorityQueue;

typedef struct Graph
{
    int vertices;
    Node** adjList;
}Graph;

PriorityQueue *queue;

void enqueue(Node* n)
{
    if(queue->size==MAX)
        return;

    queue->array[queue->size++] = n;
}
int isEmpty()
{
    if(queue->size==0)
        return 1;
    return 0;
}
void minHeapify(int i)
{
    int smallest = i, left = 2*i+1, right = 2*i+2;

if(left<queue->size&&queue->array[left]->distance<queue->array[smallest]->
distance)
        smallest = left;
    else
if(right<queue->size&&queue->array[right]->distance<queue->array[smallest]
->distance)
        smallest = right;

    if(smallest!=i)
```

```
    {
        Node* temp = queue->array[smallest];
        queue->array[smallest] = queue->array[i];
        queue->array[i] = temp;
        minHeapify(smallest);
    }
}
void decreaseKey(Node *temp, int val)
{
    int index = 0;
    for(int i=0;i<queue->size;i++)
    {
        if(queue->array[i]->key==temp->key)
        {
            index = i;
            break;
        }
    }
    if(queue->array[index]->key<val)
        return;

while(index>0&&queue->array[index]->key<queue->array[(index-1)/2]->key)
    {
        Node* swap = queue->array[index];
        queue->array[index] = queue->array[(index-1)/2];
        queue->array[(index-1)/2] = swap;
        index = (index-1)/2;
    }
}
int getMinDistance()
{
    if(queue->size==0)
        return -1;
    Node* temp = queue->array[0];
    queue->array[0] = queue->array[queue->size-1];
    minHeapify(0);
    queue->size--;
    return temp->key;
}
void buildHeap()
```

```
{
    for(int i=queue->size/2-1;i>=0;i--)
    {
        minHeapify(i);
    }
}

int search(Graph* graph, int val)
{
    for(int i=0;i<graph->vertices;i++)
    {
        if(graph->adjList[i]->key==val)
        return i;
    }
    return -1;
}

Node* createNode(int trg, int weight)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->key = trg;
    node->next = NULL;
    node->distance = weight;
    return node;
}
Graph* addEdge(Graph *graph, int src, int trg, int weight)
{
    int index = search(graph, src);
    Node* newNode = createNode(trg, weight);
    newNode->next = graph->adjList[index]->next;
    graph->adjList[index]->next = newNode;

    return graph;
}
void printGraph(Graph *graph)
{
    for(int i=0;i<graph->vertices;i++)
    {
        Node* temp = graph->adjList[i]->next;
        printf("%d->", graph->adjList[i]->key);
```

```
        while(temp!=NULL)
        {
            int index = search(graph, temp->key);
            printf("%d (wt: %d), ", temp->key, temp->distance);
            temp = temp->next;
        }
        printf("\n");
    }
}
void Dijkstra(Graph *graph, int source)
{
    int parent[graph->vertices], set[graph->vertices];
    for(int i=0;i<graph->vertices;i++)
    {
        parent[i] = -1;
        set[i] = 0;
        graph->adjList[i]->distance = INT_MAX;
        enqueue(graph->adjList[i]);
    }
    int index = search(graph, source);
    graph->adjList[index]->distance = 0;
    set[index] = 1;
    buildHeap();
    while(!isEmpty())
    {
        index = search(graph, getMinDistance());
        Node* temp = graph->adjList[index]->next;
        set[index] = 1;
        while(temp!=NULL)
        {
            int index1 = search(graph, temp->key);

if(!set[index1]&&graph->adjList[index1]->distance>temp->distance+graph->ad
jList[index]->distance)
            {
                parent[index1] = graph->adjList[index]->key;
                graph->adjList[index1]->distance =
temp->distance+graph->adjList[index]->distance;
                decreaseKey(graph->adjList[index1],
temp->distance+graph->adjList[index]->distance);
```

```
            }
            temp = temp->next;
        }
    }
    for(int i=0;i<graph->vertices;i++)
    {
        printf("Distance of %d from source: %d\n", graph->adjList[i]->key,
graph->adjList[i]->distance);
    }
}

int main()
{
    int v1, v2, w;
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    printf("Enter number of vertices: ");
    scanf("%d", &graph->vertices);
    queue = (PriorityQueue*)malloc(sizeof(PriorityQueue));
    queue->size = 0;
    graph->adjList = (Node**)malloc(graph->vertices*sizeof(Node*));
    printf("Enter vertices: ");
    for(int i=0;i<graph->vertices;i++)
    {
        graph->adjList[i] = (Node*)malloc(sizeof(Node));
        scanf("%d", &graph->adjList[i]->key);
        graph->adjList[i]->next = NULL;
    }
    int edges;
    printf("Enter number of edges: ");
    scanf("%d", &edges);
    printf("Enter vertex on the edges and the weight: ");
    for(int i=0;i<edges;i++)
    {
        scanf("%d %d %d", &v1, &v2, &w);
        graph = addEdge(graph, v1, v2, w);
        graph = addEdge(graph, v2, v1, w);
    }
    printGraph(graph);
    printf("\nEnter the source vertex: ");
    scanf("%d", &w);
```

```
    Dijkstra(graph, w);
}
```

```
Enter number of vertices: 6
Enter vertices: 0 1 2 3 4 5
Enter number of edges: 9
Enter vertex on the edges and the weight: 0 1 1
0 2 5
1 2 2
1 3 2
1 4 1
2 4 2
3 5 1
3 4 3
4 5 2
0->2 (wt: 5), 1 (wt: 1),
1->4 (wt: 1), 3 (wt: 2), 2 (wt: 2), 0 (wt: 1),
2->4 (wt: 2), 1 (wt: 2), 0 (wt: 5),
3->4 (wt: 3), 5 (wt: 1), 1 (wt: 2),
4->5 (wt: 2), 3 (wt: 3), 2 (wt: 2), 1 (wt: 1),
5->4 (wt: 2), 3 (wt: 1),

Enter the source vertex: 0
Distance of 0 from source: 0
Distance of 1 from source: 1
Distance of 2 from source: 3
Distance of 3 from source: 3
Distance of 4 from source: 2
Distance of 5 from source: 4
```

---

16) Strongly Connected Components

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
typedef struct Node
{
    int key;
    struct Node* next;
}Node;
```

```
typedef struct Graph
{
    int vertices;
    Node** adjList;
    int* visited;
}Graph;
typedef struct Stack
{
    int array[MAX];
    int top;
}Stack;

Stack *stack;

void push(int val)
{
    if(stack->top==MAX-1)
        return;
    stack->array[++stack->top] = val;
}

int pop()
{
    if(stack->top==-1)
        return -1;
    return stack->array[stack->top--];
}
int isEmpty()
{
    return stack->top==-1;
}

int search(Graph* graph, int v1)
{
    for(int i=0;i<graph->vertices;i++)
    {
        if(graph->adjList[i]->key == v1)
        {
            return i;
        }
```

```
    }
    return -1;
}
void DFSVisit(Graph *graph, int value)
{
    int index = search(graph, value);
    Node *temp = graph->adjList[index]->next;
    graph->visited[index] = 1;
    while(temp!=NULL)
    {
        printf("%d ", temp->key);
        int i = search(graph, temp->key);
        if(graph->visited[i]==0)
        {
            DFSVisit(graph, temp->key);
        }
        temp = temp->next;
    }
    push(value);
}
void DFS(Graph *graph, int v)
{
    for(int i=0;i<v;i++)
    {
        graph->visited[i] = 0;
    }
    for(int i=0;i<v;i++)
    {
        if(graph->visited[i]==0)
            DFSVisit(graph, graph->adjList[i]->key);
    }
}
void DFSVisitPrint(Graph *graph, int value)
{
    int index = search(graph, value);
    Node *temp = graph->adjList[index]->next;
    graph->visited[index] = 1;
    printf("%d", value);
    while(temp!=NULL)
    {
```

```
        int i = search(graph, temp->key);
        if(graph->visited[i]==0)
        {
            DFSVisitPrint(graph, temp->key);
        }
        temp = temp->next;
    }
}
 void SCC(Graph *graph, Graph *transGraph)
{
    DFS(graph, graph->vertices);
    for(int i=0;i<graph->vertices;i++)
    {
        transGraph->visited[i] = 0;
    }
    while(!isEmpty())
    {
        int u = pop();
        if(!transGraph->visited[search(transGraph, u)])
        {
            printf("Component: ");
            DFSVisitPrint(transGraph, u);
            printf("\n");
        }
    }
}
Node* createNode(int val)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->key = val;
    node->next = NULL;
    return node;
}
Graph* addEdge(Graph *graph, int src, int trg)
{
    int index = search(graph, src);
    Node* newNode = createNode(trg);
    newNode->next = graph->adjList[index]->next;
    graph->adjList[index]->next = newNode;
```

```
    return graph;
}
void printGraph(Graph *graph)
{
    for(int i=0;i<graph->vertices;i++)
    {
        Node* temp = graph->adjList[i]->next;
        printf("%d->", graph->adjList[i]->key);
        while(temp!=NULL)
        {
            printf("%d ", temp->key);
            temp = temp->next;
        }
        printf("\n");
    }
}
int main() {
    printf("Enter number of vertices for the graph: ");
    int v;
    scanf("%d", &v);
    Graph *graph = (Graph*)malloc(sizeof(Graph));
    graph->adjList = (Node**)malloc(v*sizeof(Node*));
    Graph *transGraph = (Graph*)malloc(sizeof(Graph));
    transGraph->adjList = (Node**)malloc(v*sizeof(Node*));
    printf("Enter number of edges: ");
    int e;
    scanf("%d", &e);
    stack = (Stack*)malloc(sizeof(Stack));
    stack->top = -1;
    graph->visited = (int*)malloc(v*sizeof(int));
    printf("Enter vertices: ");
    for(int i=0;i<v;i++)
    {
        int vertex;
        scanf("%d", &vertex);
        graph->adjList[i] = createNode(vertex);
        transGraph->adjList[i] = createNode(vertex);
    }
    printf("Enter edges: ");
    for(int i=0;i<e;i++)
```

```
    {
        int v1, v2;
        scanf("%d %d", &v1, &v2);
        graph = addEdge(graph, v1, v2);
        transGraph = addEdge(transGraph, v2, v1);
    }
    printGraph(graph);
    SCC(graph, transGraph);
}
```

```
Enter number of vertices for the graph: 5
Enter number of edges: 5
Enter vertices: 1 2 3 4 5
Enter edges: 1 2
2 3
3 1
2 4
4 5
1->2
2->3 4
3->1
4->5
5->
[3, 5, 4, 2, 1]
Component: 1 3 2
Component: 4
Component: 5
```

---

17) Bellman_Ford

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

struct Edge {
  int u;
  int v;
  int w;
};
struct Graph {
  int V;
  int E;
```

```
  struct Edge *edge;
};

void bellmanford(struct Graph *g, int source);
void display(int arr[], int size);

int main(void) {
    struct Graph *g = (struct Graph *)malloc(sizeof(struct Graph));
    printf("Enter number of vertices for the graph: ");
    int v;
    scanf("%d", &g->V);
    printf("Enter number of edges: ");
    int e;
    scanf("%d", &g->E);
    g->edge = (struct Edge *)malloc(g->E * sizeof(struct Edge));
    printf("Enter vertex on the edges and the weight: ");
    int v1, v2, w;
    for(int i=0;i<g->E;i++)
    {
        scanf("%d %d %d", &g->edge[i].u, &g->edge[i].v, &g->edge[i].w);
    }
    bellmanford(g, 0);

    return 0;
}

void bellmanford(struct Graph *g, int source) {

  int i, j, u, v, w;
  int tV = g->V;
  int tE = g->E;

  int d[tV];
  int p[tV];

  for (i = 0; i < tV; i++) {
    d[i] = INT_MAX;
    p[i] = 0;
  }
```

```
  d[source] = 0;

  for (i = 1; i <= tV - 1; i++) {
    for (j = 0; j < tE; j++) {
      u = g->edge[j].u;
      v = g->edge[j].v;
      w = g->edge[j].w;

      if (d[u] != INT_MAX && d[v] > d[u] + w) {
        d[v] = d[u] + w;
        p[v] = u;
      }
    }
  }
  for (i = 0; i < tE; i++) {
    u = g->edge[i].u;
    v = g->edge[i].v;
    w = g->edge[i].w;
    if (d[u] != INT_MAX && d[v] > d[u] + w) {
      printf("Negative weight cycle detected!\n");
      return;
    }
  }
  printf("Distance array: ");
  display(d, tV);
  printf("Parent array: ");
  display(p, tV);
}

void display(int arr[], int size) {
  int i;
  for (i = 0; i < size; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");
}
```

```
Enter number of vertices for the graph: 5
Enter number of edges: 9
Enter vertex on the edges and the weight: 0 4 4
0 2 2
4 2 3
2 4 1
4 1 2
2 1 4
2 3 5
3 1 -5
4 3 3
Distance array: 0 1 2 6 3
Parent array: 0 3 0 4 2
```
