1. Linear search

```
#include <stdio.h>
int search(int arr[], int N, int x) {
    for (int i = 0; i < N; i++) {
        if (arr[i] == x) {
            if (i + 1 < N) {
                printf("%d", arr[i + 1]);
            } else {
                printf("-1");
            }
            return i;
        }
    }
    return -1;
}
int main(void) {
    int arr[] = {2, 3, 4, 10, 40};
    int x = 10;
    int N = sizeof(arr) / sizeof(arr[0]);
    int result = search(arr, N, x);
    if (result == -1) {
        printf("Element is not present in array");
    }
    return 0;
}
```

2. Binary Search 1

```
#include <stdio.h>
int binarySearch(int arr[], int l, int r, int x) {
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] < x)
            l = mid + 1;
        else
            r = mid - 1;
    }
    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);
    if (result == -1) {
        printf("-1");
    } else {
        if (result + 1 < n) {
            printf("%d", arr[result + 1]);
        } else {
            printf("-1");
        }
    }
    return 0;
}
```

3.Selection Sort 1

```
#include <stdio.h>
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
    }
}

int main(void) {
    int arr[] = {2, 3, 4, 10, 40};
    int n = sizeof(arr) / sizeof(arr[0]);
    selectionSort(arr, n);
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

4 Selection Sort II

```
#include <stdio.h>
void swap(int *xp, int *yp) {
   int temp = *xp;
   *xp = *yp;
   *yp = temp;
}
void selectionSort(int arr[], int n) {
   int i, j, min_idx, swap_count = 0;
   for (i = 0; i < n-1; i++) {
      min_idx = i;
      for (j = i+1; j < n; j++) {
         if (arr[j] < arr[min_idx]) {
            min_idx = j;
         }
      }
      swap(&arr[min_idx], &arr[i]);
      swap_count++;
   }
   printf("%d", arr[0]);
   for (i = 1; i < n; i++) {
      printf(" %d", arr[i]);
   }
   printf("\n%d", swap_count);
}
int main() {
   int n, i;
   scanf("%d", &n);
   int arr[n];
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
   selectionSort(arr, n);
   return 0;
}
```

5. Bubble Sort 1

```
#include <stdio.h>
void swap(int *xp, int *yp) {
   int temp = *xp;
   *xp = *yp;
   *yp = temp;
}
```

```
void bubbleSort(int arr[], int n) {
   int i, j, count = 0;
   for (i = 0; i < n-1; i++) {
      count = 0;
      for (j = 0; j < n-i-1; j++) {
         if (arr[j] > arr[j+1]) {
            swap(&arr[j], &arr[j+1]);
            count++;
         }
      }
      if (count == 0) {
         break;
      }
   }
   printf("%d", arr[0]);
   for (i = 1; i < n; i++) {
      printf(" %d", arr[i]);
   }
   printf("\n%d", count);
}

int main() {
   int n, i;
   scanf("%d", &n);
   int arr[n];
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
   bubbleSort(arr, n);
   return 0;
}
```

6. Bubble Sort III

```
#include <stdio.h>
// Swap function
void swap(int *xp, int *yp) {
   int temp = *xp;
   *xp = *yp;
   *yp = temp;
}

// Recursive bubble sort function
void recurbublSort(int arr[], int len) {
   // Base case
   if (len == 1) return;
```

```
    int i, temp;
    for (i = 0; i < len - 1; i++) {
        if (arr[i] > arr[i + 1]) {
            swap(&arr[i], &arr[i + 1]);
        }
    }

    // Recur for all elements except the last of the current subarray
    recurbublSort(arr, len - 1);

    // Print the array after each iteration
    for (i = 0; i < len; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {5, 7, 2, 3, 1, 4};
    int len = sizeof(arr) / sizeof(arr[0]);

    // Call the recursive bubble sort function
    recurbublSort(arr, len);

    return 0;
}
```

7. Insertion Sort

```
#include <stdio.h>
void insertionSort(int arr[], int n) {
    int i, j, key;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
        printf("%d", arr[0]);
        for (j = 1; j <= i; j++) {
            printf(" %d", arr[j]);
        }
        printf("\n");
    } }
```

```
int main() {
   int n, i;
   scanf("%d", &n);
   int arr[n];
   for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
   insertionSort(arr, n);
   return 0;
}
```

8. Quick Sort 1

```
#include <stdio.h>
void swap(int* a, int* b) {
   int t = *a;
   *a = *b;
   *b = t;
}

int partition(int arr[], int low, int high) {
   int pivot = arr[low];
   int i = low - 1;
   int j = high + 1;
   while (1) {
      do {
         i++;
      } while (arr[i] < pivot);
      do {
         j--;
      } while (arr[j] > pivot);
      if (i >= j) {
         return j;
      }
      swap(&arr[i], &arr[j]);
   }
}

void quickSort(int arr[], int low, int high) {
   if (low < high) {
      int pi = partition(arr, low, high);
      quickSort(arr, low, pi);
      quickSort(arr, pi + 1, high);
   }
}
```

```
int main() {
    int n;
    scanf("%d", &n);
    int arr[n];
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    quickSort(arr, 0, n - 1);
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

9. Merge Sort 1

```
#include <stdio.h>

void merge(int arr[], int left, int mid, int right) {
    int i = left;
    int j = mid;
    int k = left;

    while (i < mid && j < right) {
        if (arr[i] < arr[j]) {
            arr[k++] = arr[i];
            i++;
        } else {
            arr[k++] = arr[j];
            j++;
        }
    }

    while (i < mid) {
        arr[k++] = arr[i];
        i++;
    }

    while (j < right) {
        arr[k++] = arr[j];
        j++;
    }
}

void mergeSort(int arr[], int low, int high) {
```

```
    if (low < high) {
        int mid = (low + high) / 2;
        mergeSort(arr, low, mid);
        mergeSort(arr, mid + 1, high);
        merge(arr, low, mid, high);
    }
}

int main() {
    int n;
    scanf("%d", &n);
    int arr[n];
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    mergeSort(arr, 0, n - 1);
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

10. Linked list insert at head

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

struct Node {
    int data;
    struct Node* next;
};

struct Node* addFirst(struct Node* head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = head;
    head = newNode;
    return head;
}

void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}
```

```
int main() {
   int n;
   scanf("%d", &n);
   struct Node* head = NULL;
   for (int i = 0; i < n; i++) {
      int data;
      scanf("%d", &data);
      head = addFirst(head, data);
      printList(head);
   }
   return 0;
}
```

11. Linked list insert at tail

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

struct Node {
   int data;
   struct Node* next;
};

struct Node* addFirst(struct Node* head, int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = head;
   head = newNode;
   return head;
}

void printList(struct Node* head) {
   while (head != NULL) {
      printf("%d ", head->data);
      head = head->next;
   }
   printf("\n");
}

int main() {
   int n;
   scanf("%d", &n);
   struct Node* head = NULL;
   for (int i = 0; i < n; i++) {
      int data;
      scanf("%d", &data);
```

```
        head = addFirst(head, data);
        printList(head);
    }
    return 0;
}
```

12. linked list Inserted at given position

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

struct Node {
    int data;
    struct Node* next;
};

struct Node* insertAtPosition(struct Node* head, int data, int position) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;

    if (position == 0) {
        newNode->next = head;
        head = newNode;
    } else {
        struct Node* current = head;
        for (int i = 0; i < position - 1; i++) {
            current = current->next;
        }
        newNode->next = current->next;
        current->next = newNode;
    }

    return head;
}

void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}

int main() {
    int n;
```

```
    scanf("%d", &n);
    struct Node* head = NULL;
    for (int i = 0; i < n; i++) {
        int data;
        scanf("%d", &data);
        head = insertAtPosition(head, data, i);
        printList(head);
    }
    return 0;
}
```

13. Linked list delete at head

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

struct Node {
    int data;
    struct Node* next;
};

struct Node* deleteAtHead(struct Node* head) {
    if (head == NULL) {
        return NULL;
    }

    struct Node* temp = head;
    head = head->next;
    free(temp);

    return head;
}

void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}

int main() {
    int n;
    scanf("%d", &n);
    struct Node* head = NULL;
    for (int i = 0; i < n; i++) {
        int data;
        scanf("%d", &data);
```

```
        head = deleteAtHead(head);
        printList(head);
    }
    return 0;
}
```

14. Linked list delete at tail

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

struct Node {
    int data;
    struct Node* next;
};

struct Node* deleteAtTail(struct Node* head) {
    if (head == NULL) {
        return NULL;
    }

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

    struct Node* current = head;
    while (current->next->next != NULL) {
        current = current->next;
    }

    free(current->next);
    current->next = NULL;

    return head;
}

void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}
```

```
int main() {
    int n;
    scanf("%d", &n);
    struct Node* head = NULL;
    for (int i = 0; i < n; i++) {
        int data;
        scanf("%d", &data);
        head = deleteAtTail(head);
        printList(head);
    }
    return 0;
}
```

15. Linked list delete at index

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

struct Node {
    int data;
    struct Node* next;
};

struct Node* deleteAtIndex(struct Node* head, int index) {
    if (head == NULL) {
        return NULL;
    }

    struct Node* current = head;
    for (int i = 0; i < index; i++) {
        if (current == NULL) {
            return head;
        }
        current = current->next;
    }

    if (current == NULL) {
        return head;
    }

    struct Node* temp = current->next;
    current->next = NULL;
    free(temp);

    return head;
}
```

```
void printList(struct Node* head) {
   while (head != NULL) {
      printf("%d ", head->data);
      head = head->next;
   }
   printf("\n");
}

int main() {
   int n;
   scanf("%d", &n);
   struct Node* head = NULL;
   for (int i = 0; i < n; i++) {
      int data;
      scanf("%d", &data);
      head = deleteAtIndex(head, i);
      printList(head);
   }
   return 0;
}
```

16. Linked list print the position of a node

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

// Definition for singly-linked list.
struct SinglyLinkedListNode {
   int data;
   struct SinglyLinkedListNode* next;
};

void searchNode(struct SinglyLinkedListNode* head, int value) {
   int position = 1;
   struct SinglyLinkedListNode* current = head;

   // Traverse the linked list
   while (current != NULL) {
      // Check if the current node's data matches the given value
      if (current->data == value) {
         // Print the position of the node where the value is found
         printf("%d\n", position);
         return;
      }

      // Move to the next node
      current = current->next;
```

```
        position++;
    }

    // If the value is not found, print -1
    printf("-1\n");
}

// Example usage
int main() {
    int n; // Number of elements in the linked list
    scanf("%d", &n);

    struct SinglyLinkedListNode* head = NULL;
    struct SinglyLinkedListNode* tail = NULL;

    // Create the linked list
    for (int i = 0; i < n; i++) {
        int data;
        scanf("%d", &data);

        // Create a new node
        struct SinglyLinkedListNode* newNode = (struct SinglyLinkedListNode*)malloc(sizeof(struct
SinglyLinkedListNode));
        newNode->data = data;
        newNode->next = NULL;

        // Update the linked list
        if (head == NULL) {
            head = newNode;
            tail = newNode;
        } else {
            tail->next = newNode;
            tail = newNode;
        }
    }

    // Call the searchNode function with a value to search
    int searchValue;
    scanf("%d", &searchValue);
    searchNode(head, searchValue);

    return 0;
}
```

17.Linked list transverse

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

```
struct Node {
   int data;
   struct Node* next;
};

void printLinkedList(struct Node* head) {
   struct Node* current = head;
   while (current != NULL) {
      printf("%d\n", current->data);
      current = current->next;
   }
}

int main() {
   int n;
   scanf("%d", &n);
   struct Node* head = NULL;
   struct Node* tail = NULL;
   for (int i = 0; i < n; i++) {
      int data;
      scanf("%d", &data);
      struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
      newNode->data = data;
      newNode->next = NULL;
      if (head == NULL) {
         head = newNode;
         tail = newNode;
      } else {
         tail->next = newNode;
         tail = newNode;
      }
   }
   printLinkedList(head);
   return 0;
}
```

18.Stack Operation 1

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 1000

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

void push(int x) {
   if (top == SIZE - 1) {
```

```
        printf("Overflow!!\n");
    } else {
        stack[++top] = x;
    }
}

void pop() {
    if (top == -1) {
        printf("Underflow!!\n");
    } else {
        top--;
    }
}

void printStack() {
    if (top == -1) {
        printf("Stack is Empty\n");
    } else {
        printf("Elements present in the stack:\n");
        for (int i = top; i >= 0; --i) {
            printf("%d\n", stack[i]);
        }
    }
}

int main() {
    int n, query, x;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        char operation[10];
        scanf("%s", operation);
        if (operation[0] == 'P') {
            printStack();
        } else if (operation[0] == 'P' && operation[1] == 'U') {
            scanf("%d", &x);
            push(x);
        } else if (operation[0] == 'P' && operation[1] == 'O') {
            pop();
        }
    }
    return 0;
}
```

19. Queue operation 1

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

```
#define SIZE 1000

int front = -1, rear = -1;
int queue[SIZE];

void enqueue(int x) {
   if (rear == SIZE - 1) {
      printf("Queue is Full!!\n");
   } else {
      if (front == -1) {
         front = 0;
      }
      queue[++rear] = x;
   }
}

void dequeue() {
   if (front == -1 || front > rear) {
      printf("Queue is Empty!!\n");
   } else {
      printf("Deleted : %d\n", queue[front++]);
   }
}

void printQueue() {
   if (front == -1 || front > rear) {
      printf("Queue is Empty!!\n");
   } else {
      printf("Queue elements are:\n");
      for (int i = front; i <= rear; i++) {
         printf("%d ", queue[i]);
      }
      printf("\n");
   }
}

int main() {
   int n, query, x;
   scanf("%d", &n);
   for (int i = 0; i < n; i++) {
      char operation[10];
      scanf("%s", operation);
      if (operation[0] == 'P') {
         printQueue();
      } else if (operation[0] == 'E') {
         scanf("%d", &x);
         enqueue(x);
      } else if (operation[0] == 'D') {
```

```
            dequeue();
        }
    }
    return 0;
}
```

20. Queue Operation 2

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 1000

int front = -1, rear = -1;
int queue[SIZE];

void enqueue(int x) {
    if (rear == SIZE - 1) {
        printf("Queue is Full!!\n");
    } else {
        if (front == -1) {
            front = 0;
        }
        queue[++rear] = x;
    }
}

void dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue is Empty!!\n");
    } else {
        printf("Deleted : %d\n", queue[front]);
        front++;
    }
}

void printQueue() {
    if (front == -1 || front > rear) {
        printf("Queue is Empty!!\n");
    } else {
        printf("Queue elements are:\n");
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}
```

```
int main() {
    int n, query, x;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        char operation[10];
        scanf("%s", operation);
        if (operation[0] == 'P') {
            printQueue();
        } else if (operation[0] == 'E') {
            scanf("%d", &x);
            enqueue(x);
        } else if (operation[0] == 'D') {
            dequeue();
        }
    }
    return 0;
}
```

21. Binary tree Creation

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

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

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        root = createNode(data);
    } else if (data <= root->data) {
        root->left = insert(root->left, data);
    } else {
        root->right = insert(root->right, data);
    }
    return root;
```

```
}

void inorder(struct Node* root) {
    if (root == NULL) {
        return;
    }
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

int main() {
    int n, data;
    scanf("%d", &n);
    struct Node* root = NULL;
    for (int i = 0; i < n; i++) {
        scanf("%d", &data);
        root = insert(root, data);
    }
    printf("Inorder traversal of the binary tree is: ");
    inorder(root);
    printf("\n");
    return 0;
}
```

22. Tree Transverse Inorder

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

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

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        root = createNode(data);
```

```
    } else if (data <= root->data) {
        root->left = insert(root->left, data);
    } else {
        root->right = insert(root->right, data);
    }
    return root;
}

void inorder(struct Node* root) {
    if (root == NULL) {
        return;
    }
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

int main() {
    int n, data;
    scanf("%d", &n);
    struct Node* root = NULL;
    for (int i = 0; i < n; i++) {
        scanf("%d", &data);
        root = insert(root, data);
    }
    printf("Inorder traversal of the binary tree is: ");
    inorder(root);
    printf("\n");
    return 0;
}
```

23. DFS Iterative

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

struct Node {
    int vertex;
    struct Node* next;
};

struct Graph {
    int numVertices;
    struct Node** adjLists;
```

```
    int* visited;
};

struct Node* createNode(int v) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

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

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

void addEdge(struct Graph* graph, int src, int dest) {
    struct 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;
}

void DFS(struct Graph* graph, int vertex) {
    struct Node* adjList = graph->adjLists[vertex];
    struct Node* temp = adjList;

    graph->visited[vertex] = 1;
    printf("%d ", vertex);

    while (temp != NULL) {
        int connectedVertex = temp->vertex;

        if (graph->visited[connectedVertex] == 0) {
            DFS(graph, connectedVertex);
        }
        temp = temp->next;
```

```
    }
}

int main() {
    int n, vertices, src, dest;
    scanf("%d", &vertices);

    struct Graph* graph = createGraph(vertices);

    for (int i = 0; i < vertices - 1; i++) {
        scanf("%d %d", &src, &dest);
        addEdge(graph, src, dest);
    }

    printf("Depth First Search Traversal: ");
    DFS(graph, 0);

    return 0;
}
```

24. BFS Recursive

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

struct Node {
    int vertex;
    struct Node* next;
};

struct Graph {
    int numVertices;
    struct Node** adjLists;
    int* visited;
};

struct Node* createNode(int v) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
```

```
    graph->adjLists = (struct Node*)malloc(vertices * sizeof(struct Node));
    graph->visited = (int*)malloc(vertices * sizeof(int));

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

void addEdge(struct Graph* graph, int src, int dest) {
    struct 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;
}

void BFS(struct Graph* graph, int vertex) {
    if (graph->visited[vertex] == 0) {
        printf("%d ", vertex);
        graph->visited[vertex] = 1;
    }

    struct Node* adjList = graph->adjLists[vertex];
    struct Node* temp = adjList;

    while (temp != NULL) {
        int connectedVertex = temp->vertex;

        if (graph->visited[connectedVertex] == 0) {
            printf("%d ", connectedVertex);
            graph->visited[connectedVertex] = 1;
        }
        temp = temp->next;
    }

    temp = adjList;
    while (temp != NULL) {
        int connectedVertex = temp->vertex;

        if (graph->visited[connectedVertex] == 0) {
            BFS(graph, connectedVertex);
        }
        temp = temp->next;
    }
```

```
}

int main() {
   int n, vertices, src, dest;
   scanf("%d", &vertices);

   struct Graph* graph = createGraph(vertices);

   for (int i = 0; i < vertices - 1; i++) {
      scanf("%d %d", &src, &dest);
      addEdge(graph, src, dest);
   }

   printf("Breadth First Search Traversal: ");
   BFS(graph, 0);

   return 0;
}
```