CS102 - Lab 13 - 18/04/2024

SHIVAMBU DEV PANDEY
23BCS123

```c
//SHIVAMBU DEV PANDEY
//23BCS123
//CSE B
//LAB 13
//PROGRAM 1
//Date : 18/04/2024

/*
1.Write a C program to implement the various operations
of max-heap data structure such as:
a.Creating a heap
b.Inserting element into the heap.
c.Removing the max element from the heap.
d.Extract max
*/

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

void heapify(int heapArr[], int n, int i) {
   int largest = i;
   int lchild = 2 * i + 1;
   int rchild = 2 * i + 2;

   if (lchild < n && heapArr[lchild] > heapArr[largest])
       largest = lchild;

   if (rchild < n && heapArr[rchild] > heapArr[largest])
       largest = rchild;

   if (largest != i) {
       int temp = heapArr[i];
       heapArr[i] = heapArr[largest];
       heapArr[largest] = temp;
       heapify(heapArr, n, largest);
   }
```

```c
}

void createHeap(int heapArr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(heapArr, n, i);
    }
}

void insert(int **heapArr, int *n, int value) {
    (*n)++;
    *heapArr = (int *)realloc(*heapArr, *n * sizeof(int));
    if (*heapArr == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }

    int i = *n - 1;
    (*heapArr)[i] = value;
    while (i != 0 && (*heapArr)[(i - 1) / 2] < (*heapArr)[i]) {
        int temp = (*heapArr)[i];
        (*heapArr)[i] = (*heapArr)[(i - 1) / 2];
        (*heapArr)[(i - 1) / 2] = temp;
        i = (i - 1) / 2;
    }
}

int removeMax(int heapArr[], int *n) {
    if (*n <= 0)
        return -1;
    if (*n == 1) {
        (*n)--;
        return heapArr[0];
    }

    int max = heapArr[0];
    heapArr[0] = heapArr[*n - 1];
    (*n)--;

    heapify(heapArr, *n, 0);
```

```c
    return max;
}

int extractMax(int heapArr[], int n) {
    if (n <= 0)
        return -1;

    int max = heapArr[0];
    printf("Max element: %d\n", max);

    return max;
}

void print(int heapArr[], int n) {
    printf("Heap: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", heapArr[i]);
    }
    printf("\n");
}

int main() {
    int n, choice,max;
    printf("Enter the initial size of the heap: ");
    scanf("%d", &n);
    int *heapArr = (int *)malloc(n * sizeof(int));
    if (heapArr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    printf("Enter the elements of the heap: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &heapArr[i]);
    }

    createHeap(heapArr, n);

    printf("\nHeap Operations:\n");
    while (1) {
```

```
        printf("1. Insert element 2. Remove max element 3. Extract max 4.
Print heap 5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice){
            case 1:
                    int value;
                    printf("Enter the element to be inserted: ");
                    scanf("%d", &value);
                    insert(&heapArr, &n, value);
                    printf("Element inserted successfully!\n");
                    print(heapArr, n);
                    break;

            case 2:
                     max = removeMax(heapArr, &n);
                    if (max != -1){
                        printf("Max element removed: %d\n", max);
                        print(heapArr, n);}
                    else
                        printf("Heap is empty!\n");
                    break;

            case 3:
                     max = extractMax(heapArr, n);
                    if (max != -1)
                        printf("Max element printed!\n");
                    else
                        printf("Heap is empty!\n");
                    break;

            case 4:
                    print(heapArr, n);
                    break;

            case 5:
                    printf("Exiting...\n");
                    free(heapArr);
                    exit(0);
```

```
            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
    }

    free(heapArr);
    return 0;
}
```

Output:

```
cd "/home/iiit/Desktop/New Folder/23BCS123/23BCS123_LAB13/" && gcc 23BCS123_LAB
3_P1.c -o 23BCS123_LAB13_P1 && "/home/iiit/Desktop/New Folder/23BCS123/23BCS123
iiit@iiit-OptiPlex-3090:~/Desktop/New Folder/23BCS123$ cd "/home/iiit/Desktop/N
1.c -o 23BCS123_LAB13_P1 && "/home/iiit/Desktop/New Folder/23BCS123/23BCS123_LA
Enter the initial size of the heap: 5
Enter the elements of the heap: 1 2 3 4 5

Heap Operations:
1. Insert element 2. Remove max element 3. Extract max 4. Print heap 5. Exit
Enter your choice: 1
Enter the element to be inserted: 7
Element inserted successfully!
Heap: 7 4 5 1 2 3
1. Insert element 2. Remove max element 3. Extract max 4. Print heap 5. Exit
Enter your choice: 1
Enter the element to be inserted: 8
Element inserted successfully!
Heap: 8 4 7 1 2 3 5
1. Insert element 2. Remove max element 3. Extract max 4. Print heap 5. Exit
Enter your choice: 2
Max element removed: 8
Heap: 7 4 5 1 2 3
1. Insert element 2. Remove max element 3. Extract max 4. Print heap 5. Exit
Enter your choice: 3
Max element: 7
Max element printed!
1. Insert element 2. Remove max element 3. Extract max 4. Print heap 5. Exit
Enter your choice:
```

```c
//SHIVAMBU DEV PANDEY
//23BCS123
//CSE B
//LAB 13
//PROGRAM 2
//Date : 18/04/2024

/*
2.Write a C program to perform the DFS traversal for a given graph
(the program should be able to consider directed and undirected
graph)
using the below representations:
a.adjacency list
b.adjacency matrix
*/

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

#define MAX 100

// Adjacency List Node
struct Node {
    int vertex;
    struct Node* next;
};

// Adjacency List representation
struct Graph {
    int numVertices;
    struct Node** adjLists;
    int* visited;
};

// Adjacency Matrix representation
int adjMatrix[MAX][MAX];
int visitedMatrix[MAX];

// Function to create a node
struct Node* createNode(int v) {
```

```c
    struct Node* newNode = malloc(sizeof(struct Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

// Function to create a graph
struct Graph* createGraph(int vertices) {
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

    graph->adjLists = malloc(vertices * sizeof(struct 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;
}

// Function to add an edge to an adjacency list
void addEdgeList(struct Graph* graph, int src, int dest, int
directed) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // For undirected graph add edge from dest to src
    if (!directed) {
        newNode = createNode(src);
        newNode->next = graph->adjLists[dest];
        graph->adjLists[dest] = newNode;
    }
}

// Function to add an edge to an adjacency matrix
void addEdgeMatrix(int src, int dest, int directed) {
    adjMatrix[src][dest] = 1;
```

```
    // For undirected graph add edge from dest to src
    if (!directed) {
        adjMatrix[dest][src] = 1;
    }
}

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

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

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

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

// DFS using adjacency matrix
void DFSMatrix(int vertices, int startVertex) {
    visitedMatrix[startVertex] = 1;
    printf("Visited %d\n", startVertex + 1);

    for (int i = 0; i < vertices; i++) {
        if (adjMatrix[startVertex][i] == 1 && visitedMatrix[i] == 0)
{
            DFSMatrix(vertices, i);
        }
    }
}

int main() {
    int vertices, edges, src, dest, start, choice, directed;
```

```
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);

    printf("Enter the number of edges: ");
    scanf("%d", &edges);

    printf("Is the graph directed? (1 for yes, 0 for no): ");
    scanf("%d", &directed);

    struct Graph* graph = createGraph(vertices);

    printf("Enter edges (Source Destination): \n");
    for (int i = 0; i < edges; i++) {
        scanf("%d %d", &src, &dest);
        addEdgeList(graph, src - 1, dest - 1, directed);
        addEdgeMatrix(src - 1, dest - 1, directed);
    }

    printf("Enter the starting vertex: ");
    scanf("%d", &start);

    printf("Choose the graph representation:\n1. Adjacency List\n2.
Adjacency Matrix\n");
    scanf("%d", &choice);

    switch (choice) {
        case 1: DFSList(graph, start - 1);
                break;
        case 2: DFSMatrix(vertices, start - 1);
                break;
        default: printf("Invalid choice!\n");
                 break;
    }

    return 0;
}
```

OUTPUT:

```
PS C:\Users\shiva\Desktop\CODES> cd "c:\Users\shiva\Desktop\CODES\C
Enter the number of vertices: 4
Enter the number of edges: 3
Is the graph directed? (1 for yes, 0 for no): 0
Enter edges (Source Destination):
1 2
2 3
3 4
Enter the starting vertex: 1
Choose the graph representation:
1. Adjacency List
2. Adjacency Matrix
1
Visited 1
Visited 2
Visited 3
Visited 4
PS C:\Users\shiva\Desktop\CODES\CS102_LABS> .\lab13
Enter the number of vertices: 4
Enter the number of edges: 6
Is the graph directed? (1 for yes, 0 for no): 1
Enter edges (Source Destination):
1 2
2 3
3 4
4 1
1 3
2 4
Enter the starting vertex: 1
Choose the graph representation:
1. Adjacency List
2. Adjacency Matrix
2
Visited 1
Visited 2
Visited 3
Visited 4
PS C:\Users\shiva\Desktop\CODES\CS102_LABS>
```

```
//SHIVAMBU DEV PANDEY
```

```c
//23BCS123
//CSE B
//LAB 13
//PROGRAM 3
//Date : 18/04/2024

/*
3.Write a C program to perform the BFS traversal for a given graph
(the program should be able to consider directed and undirected
graph)
using the below representations:
a.adjacency list
b.adjacency matrix
*/

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

#define MAX 100

// Queue structure for BFS
struct Queue {
    int items[MAX];
    int front;
    int rear;
};

// Functions for Queue operations
struct Queue* createQueue() {
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct
Queue));
    queue->front = -1;
    queue->rear = -1;
    return queue;
}

int isEmpty(struct Queue* queue) {
    if (queue->rear == -1)
        return 1;
    else
```

```
        return 0;
}

void enqueue(struct Queue* queue, int value) {
    if (queue->rear == MAX - 1)
        printf("Queue is full\n");
    else {
        if (queue->front == -1)
            queue->front = 0;
        queue->rear++;
        queue->items[queue->rear] = value;
    }
}

int dequeue(struct Queue* queue) {
    int item;
    if (isEmpty(queue)) {
        printf("Queue is empty\n");
        item = -1;
    } else {
        item = queue->items[queue->front];
        queue->front++;
        if (queue->front > queue->rear) {
            queue->front = queue->rear = -1;
        }
    }
    return item;
}

// Adjacency List Node
struct Node {
    int vertex;
    struct Node* next;
};

// Adjacency List representation
struct Graph {
    int numVertices;
    struct Node** adjLists;
    int* visited;
```

```
};

// Adjacency Matrix representation
int adjMatrix[MAX][MAX];
int visitedMatrix[MAX];

// Function to create a node
struct Node* createNode(int v) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

// Function to create a graph
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));

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

// Function to add an edge to an adjacency list
void addEdgeList(struct Graph* graph, int src, int dest, int
directed) {
    // Add edge from src to dest
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
```

```
    // Add edge from dest to src (for undirected graph)
    if (!directed) {
        newNode = createNode(src);
        newNode->next = graph->adjLists[dest];
        graph->adjLists[dest] = newNode;
    }
}

// Function to add an edge to an adjacency matrix
void addEdgeMatrix(int src, int dest, int directed) {
    adjMatrix[src][dest] = 1;

    // Add edge from dest to src (for undirected graph)
    if (!directed) {
        adjMatrix[dest][src] = 1;
    }
}

// BFS traversal using adjacency list
void BFSList(struct Graph* graph, int startVertex) {
    struct Queue* queue = createQueue();

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

    while (!isEmpty(queue)) {
        int currentVertex = dequeue(queue);
        printf("Visited %d\n", currentVertex + 1);

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

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

            if (graph->visited[adjVertex] == 0) {
                graph->visited[adjVertex] = 1;
                enqueue(queue, adjVertex);
            }
            temp = temp->next;
        }
```

```
    }
}

// BFS traversal using adjacency matrix
void BFSMatrix(int vertices, int startVertex) {
    struct Queue* queue = createQueue();

    visitedMatrix[startVertex] = 1;
    enqueue(queue, startVertex);

    while (!isEmpty(queue)) {
        int currentVertex = dequeue(queue);
        printf("Visited %d\n", currentVertex + 1);

        for (int i = 0; i < vertices; i++) {
            if (adjMatrix[currentVertex][i] == 1 && visitedMatrix[i]
== 0) {
                visitedMatrix[i] = 1;
                enqueue(queue, i);
            }
        }
    }
}

int main() {
    int vertices, edges, src, dest, start, choice, directed;

    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);

    struct Graph* graph = createGraph(vertices);

    printf("Enter the number of edges: ");
    scanf("%d", &edges);

    printf("Is the graph directed? (1 for yes, 0 for no): ");
    scanf("%d", &directed);

    printf("Enter edges (Source Destination): \n");
    for (int i = 0; i < edges; i++) {
```

```
        scanf("%d %d", &src, &dest);
        addEdgeList(graph, src - 1, dest - 1, directed);
        addEdgeMatrix(src - 1, dest - 1, directed);
    }

    printf("Enter the starting vertex: ");
    scanf("%d", &start);

    printf("Choose the graph representation:\n1. Adjacency List\n2.
Adjacency Matrix\n");
    scanf("%d", &choice);

    switch (choice) {
        case 1: BFSList(graph, start - 1);
                break;
        case 2: BFSMatrix(vertices, start - 1);
                break;
        default: printf("Invalid choice!\n");
                 break;
    }
    return 0;
}
```

OUTPUT:

```
PS C:\Users\shiva\Desktop\CODES> cd "c:\Users\shiva\Desktop\CODES
Enter the number of vertices: 6
Enter the number of edges: 5
Is the graph directed? (1 for yes, 0 for no): 1
Enter edges (Source Destination):
1 2
1 3
2 4
2 5
3 6
Enter the starting vertex: 1
Choose the graph representation:
1. Adjacency List
2. Adjacency Matrix
1
Visited 1
Visited 3
Visited 2
Visited 6
Visited 5
Visited 4
PS C:\Users\shiva\Desktop\CODES\CS102_LABS>
```