**RUSTAMJI INSTITUTE OF TECHNOLOGY**

**BSF ACADEMY, TEKANPUR**

**Practical File for**

**CS303 (Data Structure)**



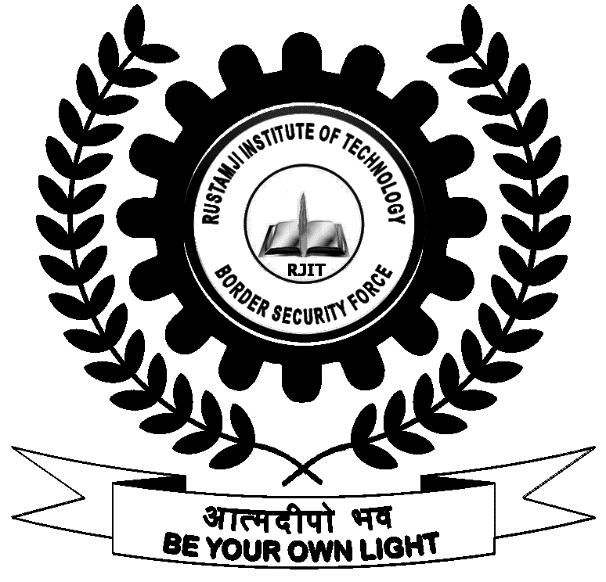
**Submitted by**

*Sumesh Kumar Singh(0902CS231120)*

B.Tech. Computer Science & Engineering 3rd Semester

(2023-2027 batch)

|  |  |
| --- | --- |
| **Subject Teacher**  Dr. Jagdish Makhijani | **File Checked by**  Mr. Yashwant Pathak |



**Self-Declaration Certificate**

I, **Sumesh Kumar Singh**, hereby declare that I have completed the lab work of CS303 (Data Structure) at my own effort and understanding.

I affirm that the work submitted is my own, and I take full responsibility for its authenticity and originality.

Date:

Sumesh Kumar Singh

0902CS231120

**ENVORIONMENT USED**

**Hardware Configuration :** Hardware configuration of machine

**C Compiler :** GCC Compiler

**User Interface :** VS Code

**GROUP MEMBERS**

**Member-1 :** Ayush Kumar & 0902CS231033

GitHub Repository URL

**Member-2 :** Aditya Raj & 0902CS231006

GitHub Repository URL

**Member-3 :** Sumesh Kumar Singh & 0902CS231120

GitHub Repository URL

**Member-4 :** Sanjeev Kumar & 0902CS231097

GitHub Repository URL

**TABLE OF CONTENTS**

**Section-A (Linked List)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Linked List using array. | 1-3 | CO-1 |
| 2 | Implementation of Linked List using Pointers. | 4-8 | CO-1 |
| 3 | Implementation of Doubly Linked List using Pointers. | 9-12 | CO-1 |
| 4 | Implementation of Circular Single Linked List using Pointers. | 13-15 | CO-1 |
| 5 | Implementation of Circular Doubly Linked List using Pointers. | 16-18 | CO-1 |

**Section-B (Stack)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Stack using Array. | 19-21 | CO-2 |
| 2 | Implementation of Stack using Pointers. | 22-24 | CO-2 |
| 3 | Program for Tower of Hanoi using recursion. | 25 | CO-2 |
| 4 | Program to find out factorial of given number using recursion. Also show the various states of stack using in this program. | 26 | CO-2 |

**Section-C (Queue)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Queue using Array. | 27-30 | CO-2 |
| 2 | Implementation of Queue using Pointers. | 31-34 | CO-2 |
| 3 | Implementation of Circular Queue using Array. | 35-38 | CO-2 |

**Section-D (Trees & Graphs)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Binary Search Tree. | 39-41 | CO-1 |
| 2 | Conversion of BST PreOrder/PostOrder/InOrder. | 42-47 | CO-3 |
| 3 | Implementation of Kruskal Algorithm | 48-51 | CO-4 |
| 4 | Implementation of Prim Algorithm | 52-55 | CO-4 |
| 5 | Implementation of Dijkstra Algorithm | 56-59 | CO-4 |

**Section-E (Sorting & Searching)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Sorting  a. Bubble  b. Selection  c. Insertion  d. Quick  e. Merge | 60-68 | CO-5 |
| 2 | Implementation of Binary Search on a list of numbers stored in an Array | 69-70 | CO-5 |
| 3 | Implementation of Binary Search on a list of strings stored in an Array | 71-72 | CO-5 |
| 4 | Implementation of Linear Search on a list of strings stored in an Array  OR  Implementation of Binary Search on a list of strings stored in a Single Linked List | 73 | CO-5 |

**Section-A (Linked List)**

**Experiment No.: 1**

**Program Description:**

Implementation of Linked List using array.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_NODES 100

struct Node {

int data;

int next;

};

struct Node linkedList[MAX\_NODES];

int head = -1;

void initList() {

for (int i = 0; i < MAX\_NODES; i++) {

linkedList[i].next = -1; // Initialize the 'next' pointer as -1 (null)

}

}

void insertNode(int value) {

int newNodeIndex = -1;

for (int i = 0; i < MAX\_NODES; i++) {

if (linkedList[i].next == -1) {

newNodeIndex = i;

break;

}

}

if (newNodeIndex == -1) {

printf("Error: Linked list is full.\n");

return;

}

linkedList[newNodeIndex].data = value;

linkedList[newNodeIndex].next = -1

if (head == -1) {

head = newNodeIndex;

} else {

int current = head;

while (linkedList[current].next != -1) {

current = linkedList[current].next;

}

linkedList[current].next = newNodeIndex;

}

}

void printList() {

int current = head;

if (current == -1) {

printf("List is empty.\n");

return;

}

while (current != -1) {

printf("%d -> ", linkedList[current].data);

current = linkedList[current].next;

}

printf("NULL\n");

}

int main() {

initList();

insertNode(10);

insertNode(20);

insertNode(30);

insertNode(40);

printf("Linked List: ");

printList();

return 0;

};

**Output**

Linked List: 10 -> 20 -> 30 -> 40 -> NULL

**Experiment-02**

**Program Description:**

Implementation of Linked List using Pointers.

**Solution**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Node\* createNode(int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

if (newNode == NULL) {

printf("Memory allocation failed!\n");

return NULL;

}

newNode->data = value;

newNode->next = NULL;

return newNode;

}

void insertAtBeginning(struct Node\*\* head, int value) {

struct Node\* newNode = createNode(value);

if (newNode == NULL) return;

newNode->next = \*head;

\*head = newNode;

}

void insertAtEnd(struct Node\*\* head, int value) {

struct Node\* newNode = createNode(value);

if (newNode == NULL) return;

if (\*head == NULL) {

\*head = newNode;

} else {

struct Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

}

void printList(struct Node\* head) {

if (head == NULL) {

printf("List is empty.\n");

return;

}

struct Node\* temp = head;

while (temp != NULL) {

printf("%d -> ", temp->data);

temp = temp->next;

}

printf("NULL\n");

}

void deleteAtBeginning(struct Node\*\* head) {

if (\*head == NULL) {

printf("List is empty, nothing to delete.\n");

return;

}

struct Node\* temp = \*head;

\*head = (\*head)->next;

free(temp);

}

void deleteAtEnd(struct Node\*\* head) {

if (\*head == NULL) {

printf("List is empty, nothing to delete.\n");

return;

}

if ((\*head)->next == NULL) {

free(\*head);

\*head = NULL;

return;

}

struct Node\* temp = \*head;

while (temp->next != NULL && temp->next->next != NULL) {

temp = temp->next;

}

free(temp->next);

temp->next = NULL;

}

void deleteAtPosition(struct Node\*\* head, int position) {

if (\*head == NULL) {

printf("List is empty, nothing to delete.\n");

return;

}

struct Node\* temp = \*head;

if (position == 0) {

\*head = temp->next;

free(temp);

return;

}

for (int i = 0; temp != NULL && i < position - 1; i++) {

temp = temp->next;

}

if (temp == NULL || temp->next == NULL) {

printf("Position out of range.\n");

return;

}

struct Node\* nodeToDelete = temp->next;

temp->next = temp->next->next;

free(nodeToDelete);

}

int main() {

struct Node\* head = NULL;

insertAtBeginning(&head, 10);

insertAtBeginning(&head, 20);

insertAtBeginning(&head, 30);

insertAtBeginning(&head, 40);

printf("Linked List after inserting at beginning: ");

printList(head);

insertAtEnd(&head, 50);

insertAtEnd(&head, 60);

printf("Linked List after inserting at end: ");

printList(head);

deleteAtBeginning(&head);

printf("Linked List after deleting from beginning: ");

printList(head);

deleteAtEnd(&head);

printf("Linked List after deleting from end: ");

printList(head);

deleteAtPosition(&head, 1);

printf("Linked List after deleting node at position 1: ");

printList(head);

return 0;

}

**Output**

Linked List after inserting at beginning: 40 -> 30 -> 20 -> 10 -> NULL

Linked List after inserting at end: 40 -> 30 -> 20 -> 10 -> 50 -> 60 -> NULL

Linked List after deleting from beginning: 30 -> 20 -> 10 -> 50 -> 60 -> NULL

Linked List after deleting from end: 30 -> 20 -> 10 -> 50 -> NULL

Linked List after deleting node at position 1: 30 -> 10 -> 50 -> NULL

**Experiment-03**

**Program Description**

Implementation of Doubly Linked List using Pointers.

**Solution**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_VERTICES 100

#define MAX\_EDGES 1000

struct Edge {

int u, v, weight;

};

struct Subset {

int parent;

int rank;

};

int compareEdges(const void\* a, const void\* b) {

return ((struct Edge\*)a)->weight - ((struct Edge\*)b)->weight;

}

int find(struct Subset subsets[], int i) {

if (subsets[i].parent != i) {

subsets[i].parent = find(subsets, subsets[i].parent);

}

return subsets[i].parent;

}

void unionSets(struct Subset subsets[], int x, int y) {

int rootX = find(subsets, x);

int rootY = find(subsets, y);

if (rootX != rootY) {

if (subsets[rootX].rank < subsets[rootY].rank) {

subsets[rootX].parent = rootY;

} else if (subsets[rootX].rank > subsets[rootY].rank) {

subsets[rootY].parent = rootX;

} else {

subsets[rootY].parent = rootX;

subsets[rootX].rank++;

}

}

}

void kruskal(int V, int E, struct Edge edges[]) {

qsort(edges, E, sizeof(edges[0]), compareEdges);

struct Subset subsets[V];

for (int i = 0; i < V; i++) {

subsets[i].parent = i;

subsets[i].rank = 0;

}

printf("Edges in the Minimum Spanning Tree (MST):\n");

int mstWeight = 0;

for (int i = 0; i < E; i++) {

int u = edges[i].u;

int v = edges[i].v;

int weight = edges[i].weight;

if (find(subsets, u) != find(subsets, v)) {

printf("%d - %d: %d\n", u, v, weight);

mstWeight += weight;

unionSets(subsets, u, v);

}

}

printf("Total weight of MST: %d\n", mstWeight);

}

int main() {

int V, E;

printf("Enter the number of vertices: ");

scanf("%d", &V);

printf("Enter the number of edges: ");

scanf("%d", &E);

struct Edge edges[E];

printf("Enter the edges (u, v, weight):\n");

for (int i = 0; i < E; i++) {

scanf("%d %d %d", &edges[i].u, &edges[i].v, &edges[i].weight);

}

kruskal(V, E, edges);

return 0;

}

**Output**

Enter the number of vertices: 4

Enter the number of edges: 5

Enter the edges (u, v, weight):

0 1 10

0 2 6

0 3 5

1 3 15

2 3 4

Edges in the Minimum Spanning Tree (MST):

2 - 3: 4

0 - 3: 5

0 - 1: 10

Total weight of MST: 19

**Experiment-04**

**Program Description:**

Implementation of Circular Single Linked List using Pointers.

**Solution**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

void insertAtEnd(struct Node\*\* head, int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

struct Node\* temp = \*head;

newNode->data = value;

newNode->next = \*head;

if (\*head == NULL) {

\*head = newNode;

} else {

while (temp->next != \*head) {

temp = temp->next;

}

temp->next = newNode;

}

}

void printList(struct Node\* head) {

struct Node\* temp = head;

if (head != NULL) {

do {

printf("%d -> ", temp->data);

temp = temp->next;

} while (temp != head);

}

printf("NULL\n");

}

void deleteNode(struct Node\*\* head, int value) {

if (\*head == NULL) return;

struct Node \*temp = \*head, \*prev = NULL;

if (temp->data == value) {

if (temp->next == \*head) {

free(temp);

\*head = NULL;

} else {

while (temp->next != \*head) {

prev = temp;

temp = temp->next;

}

prev->next = temp->next;

free(temp);

}

return;

}

while (temp->next != \*head && temp->data != value) {

prev = temp;

temp = temp->next;

}

if (temp->data == value) {

prev->next = temp->next;

free(temp);

}

}

int main() {

struct Node\* head = NULL;

insertAtEnd(&head, 10);

insertAtEnd(&head, 20);

insertAtEnd(&head, 30);

insertAtEnd(&head, 40);

printf("Circular Linked List: ");

printList(head);

deleteNode(&head, 20);

printf("Circular Linked List after deletion: ");

printList(head);

return 0;

}

**Output**

Circular Linked List: 10 -> 20 -> 30 -> 40 -> NULL

Circular Linked List after deletion: 10 -> 30 -> 40 -> NULL

**Experiment-05**

**Program Description**

Implementation of Circular Doubly Linked List using Pointers.

**Solution**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

struct Node\* prev;

};

void insertAtEnd(struct Node\*\* head, int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

struct Node\* temp = \*head;

newNode->data = value;

newNode->next = \*head;

newNode->prev = NULL;

if (\*head == NULL) {

\*head = newNode;

newNode->next = newNode; // Point to itself to form the circle

newNode->prev = newNode; // Point to itself to form the circle

} else {

while (temp->next != \*head) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

(\*head)->prev = newNode;

}

}

void printList(struct Node\* head) {

struct Node\* temp = head;

if (head != NULL) {

do {

printf("%d <-> ", temp->data);

temp = temp->next;

} while (temp != head);

}

printf("NULL\n");

}

void deleteNode(struct Node\*\* head, int value) {

if (\*head == NULL) return;

struct Node \*temp = \*head;

if ((\*head)->data == value && (\*head)->next == \*head) {

free(\*head);

\*head = NULL;

return;

}

do {

if (temp->data == value) {

if (temp == \*head) {

\*head = temp->next;

}

temp->prev->next = temp->next;

temp->next->prev = temp->prev;

free(temp);

return;

}

temp = temp->next;

} while (temp != \*head);

}

int main() {

struct Node\* head = NULL;

insertAtEnd(&head, 10);

insertAtEnd(&head, 20);

insertAtEnd(&head, 30);

insertAtEnd(&head, 40);

printf("Circular Doubly Linked List: ");

printList(head);

deleteNode(&head, 20);

printf("Circular Doubly Linked List after deletion: ");

printList(head);

return 0;

}

**Output**

Circular Doubly Linked List: 10 <-> 20 <-> 30 <-> 40 <-> NULL

Circular Doubly Linked List after deletion: 10 <-> 30 <-> 40 <-> NULL

**Section-B (Stack)**

**Experiment no -01**

**Program Description**

Implementation of Stack using Array.

**Solution**

#include <stdio.h>

#define MAX 10

int stack[MAX];

int top = -1;

void push(int value) {

if(top < MAX - 1) {

stack[++top] = value;

} else {

printf("Stack Overflow\n");

}

}

int pop() {

if(top >= 0) {

return stack[top--];

} else {

printf("Stack Underflow\n");

return -1;

}

}

int peek() {

if(top >= 0) {

return stack[top];

} else {

printf("Stack is Empty\n");

return -1;

}

}

int isEmpty() {

return top == -1;

}

int isFull() {

return top == MAX - 1;

}

int main() {

push(10);

push(20);

push(30);

printf("Top element is %d\n", peek());

printf("Popped element is %d\n", pop());

printf("Top element is %d\n", peek());

return 0;

}

**Output**

Top element is 30

Popped element is 30

Top element is 20

**Experiment 2**

**Program Description:**

Implementation of Stack using Pointers.

**Solution**

#include <stdio.h>

#include <stdlib.h>

#define MAX 5

struct Stack {

int \*arr;

int top;

};

// Function to initialize the stack

void initStack(struct Stack\* stack) {

stack->arr = (int\*)malloc(MAX \* sizeof(int));

stack->top = -1)

}

int isFull(struct Stack\* stack) {

return stack->top == MAX - 1;

}

int isEmpty(struct Stack\* stack) {

return stack->top == -1;

}

void push(struct Stack\* stack, int value) {

if (isFull(stack)) {

printf("Stack Overflow\n");

} else {

stack->arr[++(stack->top)] = value;

printf("%d pushed to stack\n", value);

}

}

int pop(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack Underflow\n");

return -1; // Return -1 if the stack is empty

} else {

int poppedValue = stack->arr[(stack->top)--]; // Pop the top element

return poppedValue;

}

}

int peek(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack is empty\n");

return -1;

} else {

return stack->arr[stack->top];

}

}

void display(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack is empty\n");

} else {

printf("Stack elements: ");

for (int i = 0; i <= stack->top; i++) {

printf("%d ", stack->arr[i]);

}

printf("\n");

}

}

int main() {

struct Stack stack;

initStack(&stack); // Initialize the stack

push(&stack, 10);

push(&stack, 20);

push(&stack, 30);

push(&stack, 40);

push(&stack, 50);

display(&stack);

printf("Popped element: %d\n", pop(&stack));

printf("Top element is: %d\n", peek(&stack));

display(&stack);

push(&stack, 60);

display(&stack);

return 0;

}

**Output**

10 pushed to stack

20 pushed to stack

30 pushed to stack

40 pushed to stack

50 pushed to stack

Stack elements: 10 20 30 40 50

Popped element: 50

Top element is: 40

Stack elements: 10 20 30 40

Stack Overflow

Stack elements: 10 20 30 40 50

**Experiment 3**

**Program Description:**

Program for Tower of Hanoi using recursion.

**Solution**

#include <stdio.h>

void towerOfHanoi(int n, char from, char to, char aux) {

if (n == 1) {

printf("Move disk 1 from %c to %c\n", from, to);

return;

}

towerOfHanoi(n - 1, from, aux, to);

printf("Move disk %d from %c to %c\n", n, from, to);

towerOfHanoi(n - 1, aux, to, from);

}

int main() {

int n = 3;

towerOfHanoi(n, 'A', 'C', 'B');

return 0;

}

**Output**

Move disk 1 from A to C

Move disk 2 from A to B

Move disk 1 from C to B

Move disk 3 from A to C

Move disk 1 from B to A

Move disk 2 from B to C

Move disk 1 from A to C

**Experiment 4**

**Program Description:**

Program to find out factorial of given number using recursion. Also show the various states of stack using in this program

**Solution**

#include <stdio.h>

int factorial(int n) {

printf("Calling factorial(%d)\n", n);

if (n == 0 || n == 1) {

return 1;

} else {

return n \* factorial(n - 1);

}

}

int main() {

int num = 5;

printf("Factorial of %d is %d\n", num, factorial(num));

return 0;

}

**Output**

Calling factorial(5)

Calling factorial(4)

Calling factorial(3)

Calling factorial(2)

Calling factorial(1)

Factorial of 5 is 120

**Section-C (Queue)**

**Experiment-01**

**Program Description:**

Implementation of Queue using Array

**Solution**

#include <stdio.h>

#include <stdlib.h>

#define MAX 5

struct Queue {

int arr[MAX];

int front;

int rear;

};

void initQueue(struct Queue\* queue) {

queue->front = -1;

queue->rear = -1;

}

int isFull(struct Queue\* queue) {

return queue->rear == MAX - 1;

}

int isEmpty(struct Queue\* queue) {

return queue->front == -1;

}

void enqueue(struct Queue\* queue, int value) {

if (isFull(queue)) {

printf("Queue is Full\n");

} else {

if (queue->front == -1) {

queue->front = 0;

}

queue->arr[++(queue->rear)] = value;

printf("%d enqueued to queue\n", value);

}

}

int dequeue(struct Queue\* queue) {

if (isEmpty(queue)) {

printf("Queue is Empty\n");

return -1;

} else {

int value = queue->arr[(queue->front)];

if (queue->front == queue->rear) {

queue->front = queue->rear = -1;

} else {

queue->front++;

}

return value;

}

}

void display(struct Queue\* queue) {

if (isEmpty(queue)) {

printf("Queue is Empty\n");

} else {

printf("Queue elements: ");

for (int i = queue->front; i <= queue->rear; i++) {

printf("%d ", queue->arr[i]);

}

printf("\n");

}

}

int main() {

struct Queue queue;

initQueue(&queue);

enqueue(&queue, 10);

enqueue(&queue, 20);

enqueue(&queue, 30);

enqueue(&queue, 40);

enqueue(&queue, 50);

display(&queue);

printf("%d dequeued from queue\n", dequeue(&queue));

display(&queue);

enqueue(&queue, 60);

display(&queue);

printf("%d dequeued from queue\n", dequeue(&queue));

display(&queue);

return 0;

}

**Output**

10 enqueued to queue

20 enqueued to queue

30 enqueued to queue

40 enqueued to queue

50 enqueued to queue

Queue elements: 10 20 30 40 50

10 dequeued from queue

Queue elements: 20 30 40 50

60 enqueued to queue

Queue elements: 20 30 40 50 60

20 dequeued from queue

Queue elements: 30 40 50 60

**Experiment-02**

**Program Description:**

Implementation of Queue using Pointers

**Solution**

#include <stdio.h>

#include <stdlib.h>

struct Queue {

int \*arr;

int front;

int rear;

int capacity;

};

void initQueue(struct Queue \*queue, int capacity) {

queue->capacity = capacity;

queue->front = -1;

queue->rear = -1;

queue->arr = (int \*)malloc(capacity \* sizeof(int));

}

int isFull(struct Queue \*queue) {

return queue->rear == queue->capacity - 1;

}

int isEmpty(struct Queue \*queue) {

return queue->front == -1;

}

void enqueue(struct Queue \*queue, int value) {

if (isFull(queue)) {

printf("Queue is Full\n");

} else {

if (queue->front == -1) {

queue->front = 0;

}

queue->arr[++(queue->rear)] = value;

printf("%d enqueued to queue\n", value);

}

}

int dequeue(struct Queue \*queue) {

if (isEmpty(queue)) {

printf("Queue is Empty\n");

return -1;

} else {

int value = queue->arr[queue->front];

if (queue->front == queue->rear) {

queue->front = queue->rear = -1;

} else {

queue->front++;

}

return value;

}

}

void display(struct Queue \*queue) {

if (isEmpty(queue)) {

printf("Queue is Empty\n");

} else {

printf("Queue elements: ");

for (int i = queue->front; i <= queue->rear; i++) {

printf("%d ", queue->arr[i]);

}

printf("\n");

}

}

int main() {

struct Queue queue;

initQueue(&queue, 5);

enqueue(&queue, 10);

enqueue(&queue, 20);

enqueue(&queue, 30);

enqueue(&queue, 40);

enqueue(&queue, 50);

display(&queue);

printf("%d dequeued from queue\n", dequeue(&queue));

display(&queue);

enqueue(&queue, 60);

display(&queue);

printf("%d dequeued from queue\n", dequeue(&queue));

display(&queue);

free(queue.arr);

return 0;

}

**Output**

10 enqueued to queue

20 enqueued to queue

30 enqueued to queue

40 enqueued to queue

50 enqueued to queue

Queue elements: 10 20 30 40 50

10 dequeued from queue

Queue elements: 20 30 40 50

60 enqueued to queue

Queue elements: 20 30 40 50 60

20 dequeued from queue

Queue elements: 30 40 50 60

**Experiment-03**

**Program Description:**

Implementation of Circular Queue using Array

**Solution**

#include <stdio.h>

#include <stdlib.h>

#define MAX 5

struct CircularQueue {

int arr[MAX];

int front;

int rear;

};

void initQueue(struct CircularQueue \*queue) {

queue->front = -1;

queue->rear = -1;

}

int isFull(struct CircularQueue \*queue) {

return (queue->rear + 1) % MAX == queue->front;

}

int isEmpty(struct CircularQueue \*queue) {

return queue->front == -1;

}

void enqueue(struct CircularQueue \*queue, int value) {

if (isFull(queue)) {

printf("Queue is Full\n");

} else {

if (queue->front == -1) {

queue->front = 0;

}

queue->rear = (queue->rear + 1) % MAX;

queue->arr[queue->rear] = value;

printf("%d enqueued to queue\n", value);

}

}

int dequeue(struct CircularQueue \*queue) {

if (isEmpty(queue)) {

printf("Queue is Empty\n");

return -1;

} else {

int value = queue->arr[queue->front];

if (queue->front == queue->rear) {

queue->front = queue->rear = -1;

} else {

queue->front = (queue->front + 1) % MAX;

}

return value;

}

}

void display(struct CircularQueue \*queue) {

if (isEmpty(queue)) {

printf("Queue is Empty\n");

} else {

int i = queue->front;

printf("Queue elements: ");

while (i != queue->rear) {

printf("%d ", queue->arr[i]);

i = (i + 1) % MAX;

}

printf("%d\n", queue->arr[queue->rear]);

}

}

int main() {

struct CircularQueue queue;

initQueue(&queue);

enqueue(&queue, 10);

enqueue(&queue, 20);

enqueue(&queue, 30);

enqueue(&queue, 40);

enqueue(&queue, 50);

display(&queue);

printf("%d dequeued from queue\n", dequeue(&queue));

display(&queue);

enqueue(&queue, 60);

display(&queue);

printf("%d dequeued from queue\n", dequeue(&queue));

display(&queue);

return 0;

}

**Output**

10 enqueued to queue

20 enqueued to queue

30 enqueued to queue

40 enqueued to queue

50 enqueued to queue

Queue elements: 10 20 30 40 50

10 dequeued from queue

Queue elements: 20 30 40 50

60 enqueued to queue

Queue elements: 20 30 40 50 60

20 dequeued from queue

Queue elements: 30 40 50 60

**Section-D (Trees & Graphs)**

**Experiment-01**

**Program Description:**

Implementation of Binary Search Tree.

**Solution**

#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 = newNode->right = NULL;

return newNode;

}

struct Node\* insert(struct Node\* root, int data) {

if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

root->left = insert(root->left, data);

} else {

root->right = insert(root->right, data);

}

return root;

}

struct Node\* search(struct Node\* root, int key) {

if (root == NULL || root->data == key) {

return root;

}

if (key < root->data) {

return search(root->left, key);

}

return search(root->right, key);

}

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

printf("%d ", root->data);

inorder(root->right);

}

}

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 70);

root = insert(root, 60);

root = insert(root, 80);

printf("Inorder traversal: ");

inorder(root);

printf("\n");

int key = 40;

struct Node\* result = search(root, key);

if (result != NULL) {

printf("Found %d in the tree\n", result->data);

} else {

printf("%d not found in the tree\n", key);

}

return 0;

}

**Output**

Inorder traversal: 20 30 40 50 60 70 80

Found 40 in the tree

**Experiment - 02**

**Program Description**

Conversion of BST PreOrder/PostOrder/InOrder.

Solution

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = node->right = NULL;

return node;

}

struct Node\* insert(struct Node\* root, int data) {

if (root == NULL)

return newNode(data);

if (data < root->data)

root->left = insert(root->left, data);

else if (data > root->data)

root->right = insert(root->right, data);

return root;

}

void preOrder(struct Node\* root) {

if (root != NULL) {

printf("%d ", root->data);

preOrder(root->left);

preOrder(root->right);

}

}

int main() {

int preorder[] = {10, 5, 1, 7, 40, 50};

int n = sizeof(preorder) / sizeof(preorder[0]);

struct Node\* root = NULL;

for (int i = 0; i < n; i++) {

root = insert(root, preorder[i]);

}

printf("PreOrder Traversal: ");

preOrder(root);

printf("\n");

return 0;

}

Output

PreOrder Traversal: 10 5 1 7 40 50

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = node->right = NULL;

return node;

}

struct Node\* insert(struct Node\* root, int data) {

if (root == NULL)

return newNode(data);

if (data < root->data)

root->left = insert(root->left, data);

else if (data > root->data)

root->right = insert(root->right, data);

return root;

}

void postOrder(struct Node\* root) {

if (root != NULL) {

postOrder(root->left);

postOrder(root->right);

printf("%d ", root->data);

}

}

int main() {

int postorder[] = {1, 7, 5, 50, 40, 10};

int n = sizeof(postorder) / sizeof(postorder[0]);

struct Node\* root = NULL;

for (int i = 0; i < n; i++) {

root = insert(root, postorder[i]);

}

printf("PostOrder Traversal: ");

postOrder(root);

printf("\n");

return 0;

}

**Output**

PostOrder Traversal: 1 7 5 50 40 10

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = node->right = NULL;

return node;

}

struct Node\* insert(struct Node\* root, int data) {

if (root == NULL)

return newNode(data);

if (data < root->data)

root->left = insert(root->left, data);

else if (data > root->data)

root->right = insert(root->right, data);

return root;

}

void inOrder(struct Node\* root) {

if (root != NULL) {

inOrder(root->left);

printf("%d ", root->data);

inOrder(root->right);

}

}

int main() {

int inorder[] = {1, 5, 7, 10, 40, 50};

int n = sizeof(inorder) / sizeof(inorder[0]);

struct Node\* root = NULL;

for (int i = 0; i < n; i++) {

root = insert(root, inorder[i]);

}

printf("InOrder Traversal: ");

inOrder(root);

printf("\n");

return 0;

}

**Output**

InOrder Traversal: 1 5 7 10 40 50

**Experiment no-3**

**Program Discription**

Implementation of Kruskal Algorithm

**Solution-**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

struct Edge {

int u, v, weight;

};

struct Subset {

int parent, rank;

};

struct Edge edges[MAX];

struct Subset subsets[MAX];

int compare(const void\* a, const void\* b) {

return ((struct Edge\*)a)->weight - ((struct Edge\*)b)->weight;

}

int find(int i) {

if (subsets[i].parent != i)

subsets[i].parent = find(subsets[i].parent);

return subsets[i].parent;

}

void unionSets(int x, int y) {

int rootX = find(x);

int rootY = find(y);

if (subsets[rootX].rank < subsets[rootY].rank)

subsets[rootX].parent = rootY;

else if (subsets[rootX].rank > subsets[rootY].rank)

subsets[rootY].parent = rootX;

else {

subsets[rootY].parent = rootX;

subsets[rootX].rank++;

}

}

void kruskal(int n, int m) {

int i, j = 0;

qsort(edges, m, sizeof(edges[0]), compare);

for (i = 0; i < n; i++) {

subsets[i].parent = i;

subsets[i].rank = 0;

}

printf("Edges in MST:\n");

while (j < m) {

int u = edges[j].u;

int v = edges[j].v;

int weight = edges[j].weight;

j++;

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV) {

printf("%d - %d: %d\n", u, v, weight);

unionSets(rootU, rootV);

}

}

}

int main() {

int n, m, i;

printf("Enter number of vertices and edges: ");

scanf("%d %d", &n, &m);

printf("Enter edges (u, v, weight):\n");

for (i = 0; i < m; i++) {

scanf("%d %d %d", &edges[i].u, &edges[i].v, &edges[i].weight);

}

kruskal(n, m);

return 0;

}

**Output**

Enter number of vertices and edges: 4 5

Enter edges (u, v, weight):

0 1 10

0 2 6

0 3 5

1 3 15

2 3 4

Edges in MST:

2 - 3: 4

0 - 3: 5

0 - 1: 10

**Experiment no-4**

**Program description**

Implementation of Prim Algorithm

#include <stdio.h>

#include <limits.h>

#define MAX 100

int graph[MAX][MAX];

int parent[MAX];

int key[MAX];

int visited[MAX];

int n;

void prim() {

for (int i = 0; i < n; i++) {

key[i] = INT\_MAX;

visited[i] = 0;

}

key[0] = 0; // Start from the first vertex

parent[0] = -1; // No parent for the root vertex

for (int count = 0; count < n - 1; count++) {

int u = -1;

// Find the vertex with the minimum key value

for (int v = 0; v < n; v++) {

if (!visited[v] && (u == -1 || key[v] < key[u])) {

u = v;

}

}

visited[u] = 1; // Include the vertex in MST

// Update the key values of the adjacent vertices

for (int v = 0; v < n; v++) {

if (graph[u][v] && !visited[v] && graph[u][v] < key[v]) {

key[v] = graph[u][v];

parent[v] = u;

}

}

}

printf("Edges in MST:\n");

for (int i = 1; i < n; i++) {

printf("%d - %d: %d\n", parent[i], i, graph[i][parent[i]]);

}

}

int main() {

int m, u, v, weight;

printf("Enter number of vertices and edges: ");

scanf("%d %d", &n, &m);

printf("Enter edges (u, v, weight):\n");

for (int i = 0; i < m; i++) {

scanf("%d %d %d", &u, &v, &weight);

graph[u][v] = weight;

graph[v][u] = weight; // Since it's an undirected graph

}

prim();

return 0;

}

**Output**

Enter number of vertices and edges: 5 7

Enter edges (u, v, weight):

0 1 2

0 3 6

1 3 8

1 2 3

2 3 7

2 4 5

3 4 9

Edges in MST:

0 - 1: 2

1 - 2: 3

0 - 3: 6

2 - 4: 5

**Experiment no -5**

**Program description**

Implementation of Dijkstra Algorithm

**Solution**

#include <stdio.h>

#include <limits.h>

#define MAX 100

#define INF INT\_MAX

int graph[MAX][MAX];

int dist[MAX];

int visited[MAX];

int parent[MAX];

int n;

void dijkstra(int source) {

for (int i = 0; i < n; i++) {

dist[i] = INF;

visited[i] = 0;

parent[i] = -1;

}

dist[source] = 0;

for (int count = 0; count < n - 1; count++) {

int u = -1;

for (int v = 0; v < n; v++) {

if (!visited[v] && (u == -1 || dist[v] < dist[u])) {

u = v;

}

}

visited[u] = 1;

for (int v = 0; v < n; v++) {

if (graph[u][v] != 0 && !visited[v] && dist[u] + graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph[u][v];

parent[v] = u;

}

}

}

printf("Vertex\tDistance from Source\tPath\n");

for (int i = 0; i < n; i++) {

printf("%d\t%d\t\t\t", i, dist[i]);

int p = i;

while (parent[p] != -1) {

printf("%d <- ", p);

p = parent[p];

}

printf("%d\n", source);

}

}

int main() {

int m, u, v, weight;

printf("Enter number of vertices and edges: ");

scanf("%d %d", &n, &m);

printf("Enter edges (u, v, weight):\n");

for (int i = 0; i < m; i++) {

scanf("%d %d %d", &u, &v, &weight);

graph[u][v] = weight;

graph[v][u] = weight; // Since it's an undirected graph

}

int source;

printf("Enter the source vertex: ");

scanf("%d", &source);

dijkstra(source);

return 0;

}

**Output**

Enter number of vertices and edges: 5 7

Enter edges (u, v, weight):

0 1 10

0 2 5

1 2 2

1 3 1

2 3 9

2 4 2

3 4 4

Enter the source vertex: 0

Vertex Distance from Source Path

0 0 0

1 7 1 <- 0

2 5 2 <- 0

3 8 3 <- 1 <- 0

4 7 4 <- 2 <- 0

**Section-E (Sorting & Searching)**

**Experiment no-01**

**Program Description**

**Solution**

Implementation of Sorting

a. Bubble

b. Selection

c. Insertion

d. Quick

e. Merge

**Solution**

1. **Bubble**

#include <stdio.h>

void bubbleSort(int arr[], int n) {

int temp;

for (int i = 0; i < n-1; i++) {

for (int j = 0; j < n-i-1; j++) {

if (arr[j] > arr[j+1]) {

// Swap

temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

}

}

void printArray(int arr[], int size) {

for (int i = 0; i < size; i++) {

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int arr[] = {64, 34, 25, 12, 22, 11, 90};

int n = sizeof(arr)/sizeof(arr[0]);

printf("Original Array: ");

printArray(arr, n);

bubbleSort(arr, n);

printf("Sorted Array: ");

printArray(arr, n);

return 0;

}

**b) selection**

#include <stdio.h>

void selectionSort(int arr[], int n) {

int minIndex, temp;

for (int i = 0; i < n-1; i++) {

minIndex = i;

for (int j = i+1; j < n; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

// Swap

temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

}

}

void printArray(int arr[], int size) {

for (int i = 0; i < size; i++) {

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int arr[] = {64, 34, 25, 12, 22, 11, 90};

int n = sizeof(arr)/sizeof(arr[0]);

printf("Original Array: ");

printArray(arr, n);

selectionSort(arr, n);

printf("Sorted Array: ");

printArray(arr, n);

return 0;

}

**c) Insertion**

solution

#include <stdio.h>

void insertionSort(int arr[], int n) {

int key, j;

for (int i = 1; i < n; i++) {

key = arr[i];

j = i - 1;

// Shift elements of arr[0..i-1] that are greater than key

while (j >= 0 && arr[j] > key) {

arr[j+1] = arr[j];

j = j - 1;

}

arr[j+1] = key;

}

}

void printArray(int arr[], int size) {

for (int i = 0; i < size; i++) {

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int arr[] = {64, 34, 25, 12, 22, 11, 90};

int n = sizeof(arr)/sizeof(arr[0]);

printf("Original Array: ");

printArray(arr, n);

insertionSort(arr, n);

printf("Sorted Array: ");

printArray(arr, n);

return 0;

}

**d) Quick**

#include <stdio.h>

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pivot = arr[high];

int i = low - 1;

int temp;

for (int j = low; j < high; j++) {

if (arr[j] < pivot) {

i++;

// Swap arr[i] and arr[j]

temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// Swap arr[i+1] and arr[high] (pivot)

temp = arr[i+1];

arr[i+1] = arr[high];

arr[high] = temp;

int pi = i + 1;

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

void printArray(int arr[], int size) {

for (int i = 0; i < size; i++) {

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int arr[] = {64, 34, 25, 12, 22, 11, 90};

int n = sizeof(arr)/sizeof(arr[0]);

printf("Original Array: ");

printArray(arr, n);

quickSort(arr, 0, n-1);

printf("Sorted Array: ");

printArray(arr, n);

return 0;

}

**D) Solution**

#include <stdio.h>

void merge(int arr[], int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

int L[n1], R[n2];

for (int i = 0; i < n1; i++) L[i] = arr[left + i];

for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k++] = L[i++];

} else {

arr[k++] = R[j++];

}

}

while (i < n1) {

arr[k++] = L[i++];

}

while (j < n2) {

arr[k++] = R[j++];

}

}

void mergeSort(int arr[], int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

void printArray(int arr[], int size) {

for (int i = 0; i < size; i++) {

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int arr[] = {64, 34, 25, 12, 22, 11, 90};

int n = sizeof(arr)/sizeof(arr[0]);

printf("Original Array: ");

printArray(arr, n);

mergeSort(arr, 0, n-1);

printf("Sorted Array: ");

printArray(arr, n);

return 0;

};

**Output**

Sorted Array: 11 12 22 25 34 64 90

**Experiment no-02**

**Program Description**

Implementation of Binary Search on a list of numbers stored in an Array

**Solution**

#include <stdio.h>

int binarySearch(int arr[], int n, int target) {

int low = 0, high = n - 1, mid;

while (low <= high) {

mid = low + (high - low) / 2;

if (arr[mid] == target)

return mid;

if (arr[mid] < target)

low = mid + 1;

else

high = mid - 1;

}

return -1;

}

int main() {

int arr[] = {11, 12, 22, 25, 34, 64, 90};

int n = sizeof(arr) / sizeof(arr[0]);

int target = 25;

int result = binarySearch(arr, n, target);

if (result != -1)

printf("Element found at index %d\n", result);

else

printf("Element not found\n");

return 0;

}

**Output**

Element found at index 3.

**Experiment no-03**

**Program Description**

Implementation of Binary Search on a list of strings stored in an Array

**Solution**

#include <stdio.h>

#include <string.h>

int binarySearch(char \*arr[], int n, char \*target) {

int low = 0, high = n - 1, mid;

while (low <= high) {

mid = low + (high - low) / 2;

int res = strcmp(arr[mid], target);

if (res == 0)

return mid;

if (res < 0)

low = mid + 1;

else

high = mid - 1;

}

return -1;

}

int main() {

char \*arr[] = {"apple", "banana", "grape", "kiwi", "orange", "pear", "watermelon"};

int n = sizeof(arr) / sizeof(arr[0]);

char \*target = "kiwi";

int result = binarySearch(arr, n, target);

if (result != -1)

printf("Element found at index %d\n", result);

else

printf("Element not found\n");

return 0;

}

**Output**

Element found at index 3

**Experiment no-04**

**Program Description**

Implementation of Linear Search on a list of strings stored in an Array OR Implementation of Binary Search on a list of strings stored in a Single Linked List

**solution**

#include <stdio.h>

#include <string.h>

int linearSearch(char \*arr[], int n, char \*target) {

for (int i = 0; i < n; i++) {

if (strcmp(arr[i], target) == 0)

return i;

}

return -1;

}

int main() {

char \*arr[] = {"apple", "banana", "grape", "kiwi", "orange", "pear", "watermelon"};

int n = sizeof(arr) / sizeof(arr[0]);

char \*target = "kiwi";

int result = linearSearch(arr, n, target);

if (result != -1)

printf("Element found at index %d\n", result);

else

printf("Element not found\n");

return 0;

};

**Output**

Element found at index 3