# RUSTAMJI INSTITUTE OF TECHNOLOGY

**BSF ACADEMY, TEKANPUR**

**Practical File for CS303 (Data Structure)**

**Submitted by**

*Sutiksha yadav*

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, **sutiksha yadav** , 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:20/12/24 [Student's Signature]

Sutiksha yadav

# https://github.com/Sutikshayadav/Data-structure-algorithm-

# 

# ENVIRONMENT USED

**Hardware Configuration : Processor-** AMD Ryzen 5 5600H

with Radeon Graphic

Installed RAM- 8.0 GB

Edition-Windows 11

**Version- 23H2**

**C Compiler :** GCC Compiler

**User Interface : visual studio code (vs code)**

# TABLE OF CONTENTS

**Section-A (Linked List)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.**  **No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Linked List using array. |  | CO-1 |
| 2 | Implementation of Linked List using Pointers. |  | CO-1 |
| 3 | Implementation of Doubly Linked List using Pointers. |  | CO-1 |
| 4 | Implementation of Circular Single Linked List using Pointers. |  | CO-1 |
| 5 | Implementation of Circular Doubly Linked List using Pointers. |  | CO-1 |

**Section-B (Stack)**

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

**Section-C (Queue)**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.**  **No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Queue using Array. |  | CO-2 |
| 2 | Implementation of Queue using Pointers. |  | CO-2 |
| 3 | Implementation of Circular Queue using Array. |  | CO-2 |

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

|  |  |  |  |
| --- | --- | --- | --- |
| **S.**  **No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Binary Search Tree. |  | CO-3 |
| 2 | Conversion of BST PreOrder/PostOrder/InOrder. |  | CO-3 |
| 3 | Implementation of Kruskal Algorithm |  | CO-4 |
| 4 | Implementation of Prim Algorithm |  | CO-4 |
| 5 | Implementation of Dijkstra Algorithm |  | CO-4 |

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

|  |  |  |  |
| --- | --- | --- | --- |
| **S.**  **No.** | **Practical Description** | **Page Nos.** | **COs** |
| 1 | Implementation of Sorting   1. Bubble 2. Selection 3. Insertion 4. Quick 5. Merge |  | CO-5 |
| 2 | Implementation of Binary Search on a list of numbers stored in an Array |  | CO-5 |
| 3 | Implementation of Binary Search on a list of strings stored in an Array |  | 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 |  | CO-5 |

**Experiment No.: 1**

**Program Description:**

implementation of Linked List using array.

**Solution:**

#include <stdio.h>

#define SIZE 100

typedef struct {

int data, next;

} Node;

typedef struct {

Node nodes[SIZE];

int head, free;

} LinkedList;

void initialize(LinkedList\* list) {

list->head = -1;

list->free = 0;

for (int i = 0; i < SIZE - 1; i++) list->nodes[i].next = i + 1;

list->nodes[SIZE - 1].next = -1;

}

void insert(LinkedList\* list, int value) {

if (list->free == -1) return; // Full

int newNode = list->free;

list->free = list->nodes[newNode].next;

list->nodes[newNode] = (Node){value, -1};

if (list->head == -1) list->head = newNode;

else {

int temp = list->head;

while (list->nodes[temp].next != -1) temp = list->nodes[temp].next;

list->nodes[temp].next = newNode;

}

}

void display(LinkedList\* list) {

for (int temp = list->head; temp != -1; temp = list->nodes[temp].next)

printf("%d -> ", list->nodes[temp].data);

printf("NULL\n");

}

int main() {

LinkedList list;

initialize(&list);

insert(&list, 10);

insert(&list, 20);

insert(&list, 30);

display(&list);

return 0;

}

Output: 10 -> 20 -> 30 -> NULL

**Experiment No.: 2**

**Program Description:**

Implementation of Linked List using Pointers.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

// Node structure

typedef struct Node {

int data;

struct Node\* next;

} Node;

// Insert at the end

void insert(Node\*\* head, int value) {

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

newNode->data = value;

newNode->next = NULL;

if (\*head == NULL) {

\*head = newNode;

} else {

Node\* temp = \*head;

while (temp->next != NULL)

temp = temp->next;

temp->next = newNode;

}

}

// Delete the first node

void deleteFirst(Node\*\* head) {

if (\*head == NULL) {

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

return;

}

Node\* temp = \*head;

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

free(temp);

printf("First node deleted.\n");

}

// Display the list

void display(Node\* head) {

if (head == NULL) {

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

return;

}

while (head != NULL) {

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

head = head->next;

}

printf("NULL\n");

}

// Main Function

int main() {

Node\* head = NULL;

insert(&head, 10);

insert(&head, 20);

insert(&head, 30);

printf("Initial List: ");

display(head);

deleteFirst(&head);

printf("After Deleting First Node: ");

display(head);

insert(&head, 40);

insert(&head, 50);

printf("After Adding More Elements: ");

display(head);

return 0;

}

Output: Initial List: 10 -> 20 -> 30 -> NULL

First node deleted.

After Deleting First Node: 20 -> 30 -> NULL

After Adding More Elements: 20 -> 30 -> 40 -> 50 -> NULL

**Experiment No.: 3**

**Program Description:**

Implementation of Doubly Linked List using Pointers

**Solution:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

struct Node\* prev;

};

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

newNode->prev = NULL;

return newNode;

}

void insertFront(struct Node\*\* head, int data) {

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

} else {

newNode->next = \*head;

(\*head)->prev = newNode;

\*head = newNode;

}

}

void insertEnd(struct Node\*\* head, int data) {

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

} else {

struct Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

}

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

if (\*head == NULL) {

printf("List is empty\n");

return;

}

struct Node\* temp = \*head;

if (temp != NULL && temp->data == key) {

\*head = temp->next;

if (\*head != NULL) {

(\*head)->prev = NULL;

}

free(temp);

printf("Node with value %d deleted\n", key);

return;

}

while (temp != NULL && temp->data != key) {

temp = temp->next;

}

if (temp == NULL) {

printf("Node with value %d not found\n", key);

return;

}

if (temp->next != NULL) {

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

}

if (temp->prev != NULL) {

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

}

free(temp);

printf("Node with value %d deleted\n", key);

}

void printListForward(struct Node\* head) {

struct Node\* temp = head;

if (temp == NULL) {

printf("List is empty\n");

return;

}

printf("List (Forward): ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("\n");

}

void printListBackward(struct Node\* head) {

struct Node\* temp = head;

if (temp == NULL) {

printf("List is empty\n");

return;

}

while (temp->next != NULL) {

temp = temp->next;

}

printf("List (Backward): ");

while (temp != NULL) {

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

temp = temp->prev;

}

printf("\n");

}

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

struct Node\* temp = \*head;

while (temp != NULL) {

struct Node\* nextNode = temp->next;

free(temp);

temp = nextNode;

}

\*head = NULL;

printf("List freed\n");

}

int main() {

struct Node\* head = NULL;

insertFront(&head, 10);

insertFront(&head, 20);

insertEnd(&head, 30);

insertEnd(&head, 40);

printListForward(head);

printListBackward(head);

deleteNode(&head, 20);

printListForward(head);

freeList(&head);

return 0;

}

Output: List (Forward): 20 10 30 40

List (Backward): 40 30 10 20

Node with value 20 deleted

List (Forward): 10 30 40

List freed

**Experiment No.: 4**

**Program Description:**

Implementation of Circular Single Linked List using Pointers.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* next;

} Node;

Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = newNode;

return newNode;

}

void insertAtEnd(Node\*\* head, int data) {

Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

} else {

Node\* temp = \*head;

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

temp = temp->next;

}

temp->next = newNode;

newNode->next = \*head;

}

}

void insertAtBeginning(Node\*\* head, int data) {

Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

} else {

Node\* temp = \*head;

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

temp = temp->next;

}

temp->next = newNode;

newNode->next = \*head;

\*head = newNode;

}

}

void deleteAtEnd(Node\*\* head) {

if (\*head == NULL) {

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

return;

}

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

free(\*head);

\*head = NULL;

} else {

Node\* temp = \*head;

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

temp = temp->next;

}

Node\* secondLast = \*head;

while (secondLast->next != temp) {

secondLast = secondLast->next;

}

secondLast->next = \*head;

free(temp);

}

}

void deleteAtBeginning(Node\*\* head) {

if (\*head == NULL) {

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

return;

}

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

free(\*head);

\*head = NULL;

} else {

Node\* temp = \*head;

Node\* last = \*head;

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

last = last->next;

}

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

last->next = \*head;

free(temp);

}

}

void displayList(Node\* head) {

if (head == NULL) {

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

return;

}

Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

printf("\n");

}

int main() {

Node\* head = NULL;

insertAtEnd(&head, 10);

insertAtEnd(&head, 20);

insertAtEnd(&head, 30);

insertAtBeginning(&head, 5);

displayList(head);

deleteAtBeginning(&head);

displayList(head);

deleteAtEnd(&head);

displayList(head);

return 0;

}

Output: 5 10 20 30

10 20 30

10 20

**Experiment No.: 5**

**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;

};

struct CircularDoublyLinkedList {

struct Node\* head;

};

void initList(struct CircularDoublyLinkedList\* list) {

list->head = NULL;

}

void insertAtEnd(struct CircularDoublyLinkedList\* list, int value) {

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

newNode->data = value;

if (list->head == NULL) {

list->head = newNode;

newNode->next = newNode;

newNode->prev = newNode;

} else {

struct Node\* temp = list->head;

while (temp->next != list->head) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

newNode->next = list->head;

list->head->prev = newNode;

}

}

void insertAtBegin(struct CircularDoublyLinkedList\* list, int value) {

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

newNode->data = value;

if (list->head == NULL) {

list->head = newNode;

newNode->next = newNode;

newNode->prev = newNode;

} else {

struct Node\* last = list->head->prev;

newNode->next = list->head;

newNode->prev = last;

last->next = newNode;

list->head->prev = newNode;

list->head = newNode;

}

}

void deleteNode(struct CircularDoublyLinkedList\* list, int value) {

if (list->head == NULL) {

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

return;

}

struct Node\* temp = list->head;

do {

if (temp->data == value) {

if (temp == list->head && temp->next == list->head) {

free(temp);

list->head = NULL;

return;

} else if (temp == list->head) {

struct Node\* last = list->head->prev;

list->head = list->head->next;

list->head->prev = last;

last->next = list->head;

free(temp);

return;

} else {

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

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

free(temp);

return;

}

}

temp = temp->next;

} while (temp != list->head);

printf("Value not found in the list!\n");

}

void display(struct CircularDoublyLinkedList\* list) {

if (list->head == NULL) {

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

return;

}

struct Node\* temp = list->head;

do {

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

temp = temp->next;

} while (temp != list->head);

printf("\n");

}

int main() {

struct CircularDoublyLinkedList list;

initList(&list);

insertAtEnd(&list, 10);

insertAtEnd(&list, 20);

insertAtEnd(&list, 30);

insertAtEnd(&list, 40);

insertAtEnd(&list, 50);

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

display(&list);

insertAtBegin(&list, 5);

printf("After inserting 5 at the beginning: ");

display(&list);

deleteNode(&list, 30);

printf("After deleting 30: ");

display(&list);

deleteNode(&list, 10);

printf("After deleting 10 (head node): ");

display(&list);

deleteNode(&list, 100);

printf("After trying to delete 100 (non-existent value): ");

display(&list);

return 0;

}

Output: Circular Doubly Linked List after insertions: 10 20 30 40 50

After inserting 5 at the beginning: 5 10 20 30 40 50

After deleting 30: 5 10 20 40 50

After deleting 10 (head node): 5 20 40 50

After trying to delete 100 (non-existent value): 5 20 40 50

**Section-B (Stack)**

**Experiment No.: 1**

**Program Description:**

Implementation of Stack using Array.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 5

struct Stack {

int arr[MAX];

int top;

};

void initStack(struct Stack\* stack) {

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! Cannot push %d\n", value);

} else {

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

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

}

}

int pop(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack Underflow! Cannot pop\n");

return -1;

} else {

int poppedValue = stack->arr[stack->top--];

return poppedValue;

}

}

int peek(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack is empty! No element to peek\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);

push(&stack, 10);

push(&stack, 20);

push(&stack, 30);

push(&stack, 40);

push(&stack, 50);

display(&stack);

push(&stack, 60);

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

printf("%d popped from stack\n", pop(&stack));

printf("%d popped from stack\n", pop(&stack));

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

Stack Overflow! Cannot push 60

Top element is 50

50 popped from stack

40 popped from stack

Stack elements: 10 20 30

**Experiment No.: 2**

**Program Description:**

Implementation of Stack using Pointers

**Solution:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Stack {

struct Node\* top;

};

void initStack(struct Stack\* stack) {

stack->top = NULL;

}

int isEmpty(struct Stack\* stack) {

return stack->top == NULL;

}

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

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

if (!newNode) {

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

return;

}

newNode->data = value;

newNode->next = stack->top;

stack->top = newNode;

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

}

int pop(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack Underflow! Cannot pop\n");

return -1;

}

struct Node\* temp = stack->top;

int poppedValue = temp->data;

stack->top = stack->top->next;

free(temp);

return poppedValue;

}

int peek(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack is empty! No element to peek\n");

return -1;

}

return stack->top->data;

}

void display(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack is empty!\n");

return;

}

struct Node\* temp = stack->top;

printf("Stack elements: ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("\n");

}

int main() {

struct Stack stack;

initStack(&stack);

push(&stack, 10);

push(&stack, 20);

push(&stack, 30);

push(&stack, 40);

push(&stack, 50);

display(&stack);

push(&stack, 60);

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

printf("%d popped from stack\n", pop(&stack));

printf("%d popped from stack\n", pop(&stack));

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: 50 40 30 20 10

Stack is empty! No element to peek

Top element is 50

50 popped from stack

40 popped from stack

Stack elements: 30 20 10

**Experiment No.: 3**

**Program Description:**

Program for Tower of Hanoi using recursion

**Solution:**

#include <stdio.h>

void towerOfHanoi(int n, char source, char auxiliary, char destination) {

if (n == 1) {

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

return;

}

towerOfHanoi(n - 1, source, destination, auxiliary);

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

towerOfHanoi(n - 1, auxiliary, source, destination);

}

int main() {

int n;

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

scanf("%d", &n);

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

return 0;

}

Output: Enter the number of disks: 3

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 No.: 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) {

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

return 1;

} else {

return n \* factorial(n - 1);

}

}

int main() {

int number;

printf("Enter a number to find its factorial: ");

scanf("%d", &number);

int result = factorial(number);

printf("The factorial of %d is: %d\n", number, result);

return 0;

}

Output: Enter a number to find its factorial: 3

The factorial of 3 is: 6

**Section-C (Queue)**

**Experiment No.: 1**

**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 initializeQueue(struct Queue\* q) {

q->front = -1;

q->rear = -1;

}

int isFull(struct Queue\* q) {

if (q->rear == MAX - 1) {

return 1;

}

return 0;

}

int isEmpty(struct Queue\* q) {

if (q->front == -1 || q->front > q->rear) {

return 1;

}

return 0;

}

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

if (isFull(q)) {

printf("Queue is full. Cannot enqueue %d.\n", value);

} else {

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

q->front = 0;

}

q->rear++;

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

printf("Enqueued: %d\n", value);

}

}

int dequeue(struct Queue\* q) {

if (isEmpty(q)) {

printf("Queue is empty. Cannot dequeue.\n");

return -1;

} else {

int dequeuedValue = q->arr[q->front];

q->front++;

if (q->front > q->rear) {

q->front = q->rear = -1;

}

return dequeuedValue;

}

}

void displayQueue(struct Queue\* q) {

if (isEmpty(q)) {

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

} else {

printf("Queue elements: ");

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

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

}

printf("\n");

}

}

int main() {

struct Queue q;

initializeQueue(&q);

enqueue(&q, 10);

enqueue(&q, 20);

enqueue(&q, 30);

enqueue(&q, 40);

enqueue(&q, 50);

enqueue(&q, 60);

displayQueue(&q);

printf("Dequeued: %d\n", dequeue(&q));

printf("Dequeued: %d\n", dequeue(&q));

displayQueue(&q);

enqueue(&q, 60);

displayQueue(&q);

return 0;

}

Output: Enqueued: 10

Enqueued: 20

Enqueued: 30

Enqueued: 40

Enqueued: 50

Queue is full. Cannot enqueue 60.

Queue elements: 10 20 30 40 50

Dequeued: 10

Dequeued: 20

Queue elements: 30 40 50

Queue is full. Cannot enqueue 60.

Queue elements: 30 40 50

**Experiment No.: 2**

**Program Description:**

Implementation of Queue using Pointers.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

struct Queue {

int \*arr;

int front;

int rear;

int size;

};

void initializeQueue(struct Queue\* q, int size) {

q->size = size;

q->arr = (int\*)malloc(q->size \* sizeof(int));

q->front = -1;

q->rear = -1;

}

int isFull(struct Queue\* q) {

return (q->rear == q->size - 1);

}

int isEmpty(struct Queue\* q) {

return (q->front == -1 || q->front > q->rear);

}

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

if (isFull(q)) {

printf("Queue is full. Cannot enqueue %d.\n", value);

} else {

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

q->front = 0;

}

q->rear++;

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

printf("Enqueued: %d\n", value);

}

}

int dequeue(struct Queue\* q) {

if (isEmpty(q)) {

printf("Queue is empty. Cannot dequeue.\n");

return -1;

} else {

int dequeuedValue = q->arr[q->front];

q->front++;

if (q->front > q->rear) {

q->front = q->rear = -1;

}

return dequeuedValue;

}

}

void displayQueue(struct Queue\* q) {

if (isEmpty(q)) {

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

} else {

printf("Queue elements: ");

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

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

}

printf("\n");

}

}

void freeQueue(struct Queue\* q) {

free(q->arr);

q->arr = NULL;

}

int main() {

struct Queue q;

int size;

printf("Enter the size of the queue: ");

scanf("%d", &size);

initializeQueue(&q, size);

enqueue(&q, 10);

enqueue(&q, 20);

enqueue(&q, 30);

enqueue(&q, 40);

enqueue(&q, 50);

enqueue(&q, 60);

displayQueue(&q);

printf("Dequeued: %d\n", dequeue(&q));

printf("Dequeued: %d\n", dequeue(&q));

displayQueue(&q);

enqueue(&q, 60);

displayQueue(&q);

freeQueue(&q);

return 0;

}

Output: Enter the size of the queue: 10

Enqueued: 10

Enqueued: 20

Enqueued: 30

Enqueued: 40

Enqueued: 50

Enqueued: 60

Queue elements: 10 20 30 40 50 60

Dequeued: 10

Dequeued: 20

Queue elements: 30 40 50 60

Enqueued: 60

Queue elements: 30 40 50 60 60

**Experiment No.: 3**

**Program Description:**

Implementation of Circular Queue using Array.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 5 // Maximum size of the queue

struct CircularQueue {

int arr[MAX];

int front;

int rear;

};

void initializeQueue(struct CircularQueue\* q) {

q->front = -1;

q->rear = -1;

}

int isFull(struct CircularQueue\* q) {

if ((q->rear + 1) % MAX == q->front) {

return 1;

}

return 0;

}

int isEmpty(struct CircularQueue\* q) {

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

return 1;

}

return 0;

}

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

if (isFull(q)) {

printf("Queue is full. Cannot enqueue %d.\n", value);

} else {

if (q->front == -1) { // Queue is empty

q->front = 0;

}

q->rear = (q->rear + 1) % MAX; // Circular increment

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

printf("Enqueued: %d\n", value);

}

}

int dequeue(struct CircularQueue\* q) {

if (isEmpty(q)) {

printf("Queue is empty. Cannot dequeue.\n");

return -1;

} else {

int dequeuedValue = q->arr[q->front];

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

q->front = q->rear = -1; // Queue becomes empty

} else {

q->front = (q->front + 1) % MAX; // Circular increment

}

return dequeuedValue;

}

}

void displayQueue(struct CircularQueue\* q) {

if (isEmpty(q)) {

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

} else {

printf("Queue elements: ");

int i = q->front;

while (i != q->rear) {

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

i = (i + 1) % MAX;

}

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

}

}

int main() {

struct CircularQueue q;

initializeQueue(&q);

enqueue(&q, 10);

enqueue(&q, 20);

enqueue(&q, 30);

enqueue(&q, 40);

enqueue(&q, 50);

enqueue(&q, 60); // This should print "Queue is full"

displayQueue(&q);

printf("Dequeued: %d\n", dequeue(&q));

printf("Dequeued: %d\n", dequeue(&q));

displayQueue(&q);

enqueue(&q, 60);

enqueue(&q, 70);

displayQueue(&q);

return 0;

}

Output: Enqueued: 10

Enqueued: 20

Enqueued: 30

Enqueued: 40

Enqueued: 50

Queue is full. Cannot enqueue 60.

Queue elements: 10 20 30 40 50

Dequeued: 10

Dequeued: 20

Queue elements: 30 40 50

Enqueued: 60

Enqueued: 70

Queue elements: 30 40 50 60 70

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

**Experiment No.: 1**

**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 if (data > root->data) {

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);

}

struct Node\* findMin(struct Node\* root) {

while (root->left != NULL) {

root = root->left;

}

return root;

}

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

if (root == NULL) {

return root;

}

if (key < root->data) {

root->left = deleteNode(root->left, key);

} else if (key > root->data) {

root->right = deleteNode(root->right, key);

} else {

if (root->left == NULL && root->right == NULL) {

free(root);

return NULL;

} else if (root->left == NULL) {

struct Node\* temp = root->right;

free(root);

return temp;

} else if (root->right == NULL) {

struct Node\* temp = root->left;

free(root);

return temp;

} else {

struct Node\* temp = findMin(root->right);

root->data = temp->data;

root->right = deleteNode(root->right, temp->data);

}

}

return root;

}

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("Node with value %d found in the BST.\n", key);

} else {

printf("Node with value %d not found in the BST.\n", key);

}

printf("Deleting node with value 20\n");

root = deleteNode(root, 20);

printf("Inorder Traversal after deletion: ");

inorder(root);

printf("\n");

printf("Deleting node with value 30\n");

root = deleteNode(root, 30);

printf("Inorder Traversal after deletion: ");

inorder(root);

printf("\n");

return 0;

}

Output: Inorder Traversal: 20 30 40 50 60 70 80

Node with value 40 found in the BST.

Deleting node with value 20

Inorder Traversal after deletion: 30 40 50 60 70 80

Deleting node with value 30

Inorder Traversal after deletion: 40 50 60 70 80

**Experiment No.: 2**

**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\* 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;

}

void preorder(struct Node\* root) {

if (root != NULL) {

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

preorder(root->left);

preorder(root->right);

}

}

void postorder(struct Node\* root) {

if (root != NULL) {

postorder(root->left);

postorder(root->right);

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

}

}

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("Preorder Traversal: ");

preorder(root);

printf("\n");

printf("Postorder Traversal: ");

postorder(root);

printf("\n");

printf("Inorder Traversal: ");

inorder(root);

printf("\n");

return 0;

}

output: Preorder Traversal: 50 30 20 40 70 60 80

Postorder Traversal: 20 40 30 60 80 70 50

Inorder Traversal: 20 30 40 50 60 70 80

**Experiment No.: 3**

**Program Description:**

Implementation of Kruskal Algorithm

**Solution:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

struct Edge {

int u, v, weight;

};

struct Subset {

int parent;

int rank;

};

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 xroot = find(subsets, x);

int yroot = find(subsets, y);

if (subsets[xroot].rank < subsets[yroot].rank) {

subsets[xroot].parent = yroot;

} else if (subsets[xroot].rank > subsets[yroot].rank) {

subsets[yroot].parent = xroot;

} else {

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

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

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

}

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

struct Subset subsets[V];

struct Edge result[V-1];

int e = 0, i = 0;

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

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

subsets[i].parent = i;

subsets[i].rank = 0;

}

i = 0;

while (e < V - 1) {

struct Edge nextEdge = edges[i++];

int x = find(subsets, nextEdge.u);

int y = find(subsets, nextEdge.v);

if (x != y) {

result[e++] = nextEdge;

unionSets(subsets, x, y);

}

}

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

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

printf("%d - %d: %d\n", result[i].u, result[i].v, result[i].weight);

}

}

int main() {

int V, E, i;

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

scanf("%d %d", &V, &E);

struct Edge edges[E];

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

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

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

}

kruskal(edges, V, E);

return 0;

}

Output: Edges in the Minimum Spanning Tree (MST):

Edge: 2 - 3, Weight: 4

Edge: 0 - 3, Weight: 5

Edge: 0 - 1, Weight: 10

**Experiment No.: 4**

**Program Description:**

Implementation of Prim Algorithm

**Solution:**

#include <stdio.h>

#include <limits.h>

#define MAX\_VERTICES 100

int minKey(int key[], int included[], int V) {

int min = INT\_MAX, minIndex;

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

if (included[v] == 0 && key[v] < min) {

min = key[v];

minIndex = v;

}

}

return minIndex;

}

void primMST(int graph[MAX\_VERTICES][MAX\_VERTICES], int V) {

int parent[V];

int key[V];

int included[V];

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

key[i] = INT\_MAX;

included[i] = 0;

}

key[0] = 0;

parent[0] = -1;

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

int u = minKey(key, included, V);

included[u] = 1;

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

if (graph[u][v] != 0 && included[v] == 0 && graph[u][v] < key[v]) {

parent[v] = u;

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

}

}

}

printf("Edge \tWeight\n");

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

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

}

}

int main() {

int V, E;

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

scanf("%d", &V);

int graph[MAX\_VERTICES][MAX\_VERTICES] = {0};

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

scanf("%d", &E);

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

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

int u, v, weight;

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

graph[u][v] = weight;

graph[v][u] = weight;

}

primMST(graph, V);

return 0;

}

Output: Edges in the Minimum Spanning Tree (MST):

0 -- 1 == 2

1 -- 2 == 3

0 -- 3 == 6

1 -- 4 == 5

**Experiment No.: 5**

**Program Description:**

Implementation of Dijkstra Algorithm

**Solution:**

#include <stdio.h>

#include <limits.h>

#define MAX\_VERTICES 100

int minDistance(int dist[], int sptSet[], int V) {

int min = INT\_MAX, minIndex;

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

if (sptSet[v] == 0 && dist[v] <= min) {

min = dist[v];

minIndex = v;

}

}

return minIndex;

}

void dijkstra(int graph[MAX\_VERTICES][MAX\_VERTICES], int V, int src) {

int dist[V];

int sptSet[V];

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

dist[i] = INT\_MAX;

sptSet[i] = 0;

}

dist[src] = 0;

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

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

sptSet[u] = 1;

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

if (!sptSet[v] && graph[u][v] != 0 && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v]) {

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

}

}

}

printf("Vertex \t Distance from Source\n");

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

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

}

}

int main() {

int V, E;

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

scanf("%d", &V);

int graph[MAX\_VERTICES][MAX\_VERTICES] = {0};

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

scanf("%d", &E);

printf("Enter each edge (u v weight):\n");

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

int u, v, weight;

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

graph[u][v] = weight;

graph[v][u] = weight;

}

int src;

printf("Enter the source vertex: ");

scanf("%d", &src);

dijkstra(graph, V, src);

return 0;

}

Output:

Vertex Distance from Start

0 0

1 9

2 16

3 24

4 19

5 14

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

**Experiment No.: 1**

**Program Description:**

implementation of Sorting

1. Bubble

**Solution:**

#include <stdio.h>

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

int i, j, temp;

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

int swapped = 0;

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

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

temp = arr[j];

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

arr[j + 1] = temp;

swapped = 1;

}

}

if (!swapped) {

break;

}

}

}

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: \n");

printArray(arr, n);

bubbleSort(arr, n);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

Output: Original array: 64 34 25 12 22 11 90

Sorted array: 11 12 22 25 34 64 90

1. Selection

#include <stdio.h>

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

int i, j, minIndex, temp;

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

minIndex = i;

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

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

minIndex = j;

}

}

temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = 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: \n");

printArray(arr, n);

selectionSort(arr, n);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

Output: Original array: 64 34 25 12 22 11 90

Sorted array: 11 12 22 25 34 64 90

c .Insertion

#include <stdio.h>

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

int i, j, minIndex, temp;

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

minIndex = i;

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

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

minIndex = j;

}

}

temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = 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: \n");

printArray(arr, n);

selectionSort(arr, n);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

Output: Original array: 64 34 25 12 22 11 90

Sorted array: 11 12 22 25 34 64 90

1. Quick

#include <stdio.h>

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high - 1; j++) {

if (arr[j] < pivot) {

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

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

if (low < high) {

int pi = partition(arr, low, high);

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: \n");

printArray(arr, n);

quickSort(arr, 0, n - 1);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

Output: Original array: 64 34 25 12 22 11 90

Sorted array: 11 12 22 25 34 64 90

1. Merge

#include <stdio.h>

void merge(int arr[], int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

for (int i = 0; i < n1; i++)

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

for (int j = 0; j < n2; j++)

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

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

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

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

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

void mergeSort(int arr[], int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

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: \n");

printArray(arr, n);

mergeSort(arr, 0, n - 1);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

Output: Original array: 64 34 25 12 22 11 90

Sorted array: 11 12 22 25 34 64 90

**Experiment No.: 2**

**Program Description:**

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

**Solution:**

#include <stdio.h>

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

int low = 0;

int high = size - 1;

while (low <= high) {

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

if (arr[mid] == target) {

return mid;

}

else if (arr[mid] > target) {

high = mid - 1;

}

else {

low = mid + 1;

}

}

return -1;

}

int main() {

int arr[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};

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

int target;

printf("Enter the number you want to search: ");

scanf("%d", &target);

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

if (result != -1) {

printf("Element %d is present at index %d.\n", target, result);

} else {

printf("Element %d is not present in the array.\n", target);

}

return 0;

}

Output: Enter the number you want to search: 2

Element 2 is not present in the array.

**Experiment No.: 3**

**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 size, char \*target) {

int low = 0;

int high = size - 1;

while (low <= high) {

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

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

if (result == 0) {

return mid;

} else if (result > 0) {

high = mid - 1;

} else {

low = mid + 1;

}

}

return -1;

}

int main() {

char \*arr[] = {"apple", "banana", "cherry", "date", "fig", "grape", "kiwi", "lemon", "mango", "pear"};

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

char target[50];

printf("Enter the string you want to search: ");

scanf("%s", target);

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

if (result != -1) {

printf("String \"%s\" is present at index %d.\n", target, result);

} else {

printf("String \"%s\" is not present in the array.\n", target);

}

return 0;

}

Output: Enter the string you want to search: orange

String "orange" is not present in the array.

**Experiment No.: 4**

**Program Description:**

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

**Solution:**

#include <stdio.h>

#include <string.h>

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

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

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

return i; // Target found at index i

}

}

return -1; // Target not found

}

int main() {

char \*arr[] = {"apple", "banana", "cherry", "date", "fig", "grape", "kiwi", "lemon", "mango", "pear"};

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

char target[50];

printf("Enter the string you want to search: ");

scanf("%s", target);

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

if (result != -1) {

printf("String \"%s\" is present at index %d.\n", target, result);

} else {

printf("String \"%s\" is not present in the array.\n", target);

}

return 0;

}

Output: Enter the string you want to search: apple

String "apple" is present at index 0.