Q7. Implement circular queue, priority queue, and double-ended queue (deque).

A.Circular Queue

#Algo:

#include <stdio.h>

#define MAX 5

typedef struct {

int arr[MAX];

int front;

int rear;

} CircularQueue;

// Initialize Circular Queue

void initializeCircularQueue(CircularQueue\* queue) {

queue->front = -1;

queue->rear = -1;

}

// Enqueue operation

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

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

printf("Circular Queue Overflow! Cannot enqueue %d\n", value);

return;

}

if (queue->front == -1) queue->front = 0;

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

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

printf("Enqueued %d into the circular queue\n", value);

}

// Dequeue operation

int dequeue(CircularQueue\* queue) {

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

printf("Circular Queue Underflow! Cannot dequeue\n");

return -1;

}

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

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

queue->front = -1; // Queue becomes empty

queue->rear = -1;

} else {

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

}

return value;

}

// Display circular queue

void displayCircularQueue(CircularQueue\* queue) {

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

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

return;

}

printf("Circular Queue elements: ");

int i = queue->front;

while (1) {

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

if (i == queue->rear) break;

i = (i + 1) % MAX;

}

printf("\n");

}

int main() {

CircularQueue queue;

initializeCircularQueue(&queue);

enqueue(&queue, 10);

enqueue(&queue, 20);

enqueue(&queue, 30);

enqueue(&queue, 40);

displayCircularQueue(&queue);

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

displayCircularQueue(&queue);

enqueue(&queue, 50);

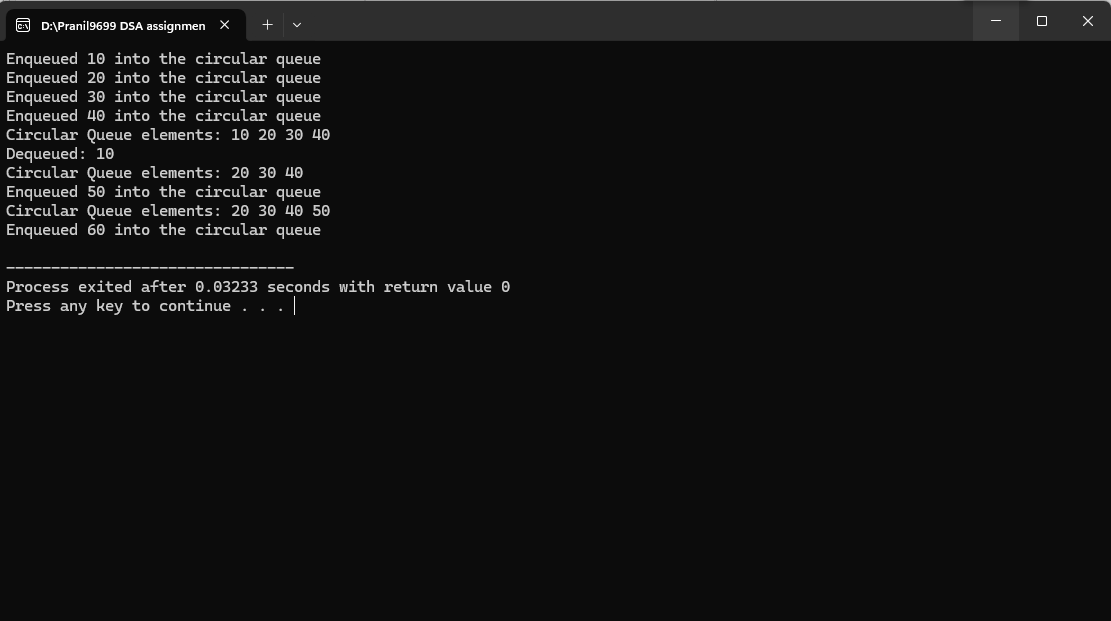
displayCircularQueue(&queue);

enqueue(&queue, 60); // Overflow scenario

return 0;

}

# Output:



B. Priority Queue

# Algorithm:

#Code :

#include <stdio.h>

#define MAX 5

typedef struct {

int arr[MAX];

int size;

} PriorityQueue;

// Initialize Priority Queue

void initializePriorityQueue(PriorityQueue\* queue) {

queue->size = 0;

}

// Enqueue operation

void enqueue(PriorityQueue\* queue, int value) {

if (queue->size == MAX) {

printf("Priority Queue Overflow! Cannot enqueue %d\n", value);

return;

}

int i = queue->size - 1;

while (i >= 0 && queue->arr[i] > value) {

queue->arr[i + 1] = queue->arr[i];

i--;

}

queue->arr[i + 1] = value;

queue->size++;

printf("Enqueued %d into the priority queue\n", value);

}

// Dequeue operation

int dequeue(PriorityQueue\* queue) {

if (queue->size == 0) {

printf("Priority Queue Underflow! Cannot dequeue\n");

return -1;

}

return queue->arr[--queue->size];

}

// Display priority queue

void displayPriorityQueue(PriorityQueue\* queue) {

int i;

if (queue->size == 0) {

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

return;

}

printf("Priority Queue elements: ");

for (i = 0; i < queue->size; i++) {

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

}

printf("\n");

}

int main() {

PriorityQueue queue;

initializePriorityQueue(&queue);

enqueue(&queue, 30);

enqueue(&queue, 10);

enqueue(&queue, 20);

enqueue(&queue, 40);

displayPriorityQueue(&queue);

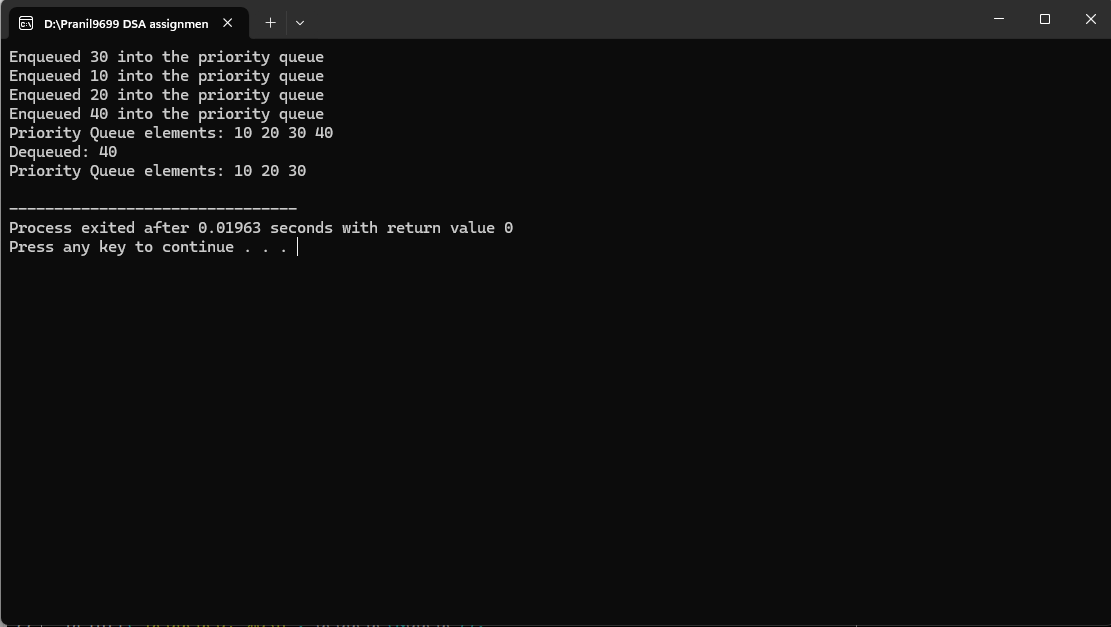
printf("Dequeued: %d\n", dequeue(&queue));

displayPriorityQueue(&queue);

return 0;

}

#Output:



C. Double-Ended Queue (Deque)

#Algorithm

#Code

#include <stdio.h>

#define MAX 5

typedef struct {

int arr[MAX];

int front;

int rear;

} Deque;

// Initialize Deque

void initializeDeque(Deque\* deque) {

deque->front = -1;

deque->rear = -1;

}

// Insert at front

void insertFront(Deque\* deque, int value) {

if ((deque->front == 0 && deque->rear == MAX - 1) || (deque->front ==

deque->rear + 1)) {

printf("Deque Overflow! Cannot insert %d at front\n", value);

return;

}

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

deque->front = deque->rear = 0;

} else if (deque->front == 0) {

deque->front = MAX - 1;

} else {

deque->front--;

}

deque->arr[deque->front] = value;

printf("Inserted %d at the front of the deque\n", value);

}

// Insert at rear

void insertRear(Deque\* deque, int value) {

if ((deque->front == 0 && deque->rear == MAX - 1) || (deque->front ==

deque->rear + 1)) {

printf("Deque Overflow! Cannot insert %d at rear\n", value);

return;

}

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

deque->front = deque->rear = 0;

} else if (deque->rear == MAX - 1) {

deque->rear = 0;

} else {

deque->rear++;

}

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

printf("Inserted %d at the rear of the deque\n", value);

}

// Delete from front

int deleteFront(Deque\* deque) {

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

printf("Deque Underflow! Cannot delete from front\n");

return -1;

}

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

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

deque->front = deque->rear = -1;

} else if (deque->front == MAX - 1) {

deque->front = 0;

} else {

deque->front++;

}

return value;

}

// Delete from rear

int deleteRear(Deque\* deque) {

if (deque->rear == -1) {

printf("Deque Underflow! Cannot delete from rear\n");

return -1;

}

int value = deque->arr[deque->rear];

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

deque->front = deque->rear = -1;

} else if (deque->rear == 0) {

deque->rear = MAX - 1;

} else {

deque->rear--;

}

return value;

}

// Display deque

void displayDeque(Deque\* deque) {

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

printf("Deque is empty\n");

return;

}

printf("Deque elements: ");

int i = deque->front;

while (1) {

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

if (i == deque->rear) break;

i = (i + 1) % MAX;

}

printf("\n");

}

int main() {

Deque deque;

initializeDeque(&deque);

insertRear(&deque, 10);

insertRear(&deque, 20);

insertFront(&deque, 5);

displayDeque(&deque);

printf("Deleted from front: %d\n", deleteFront(&deque));

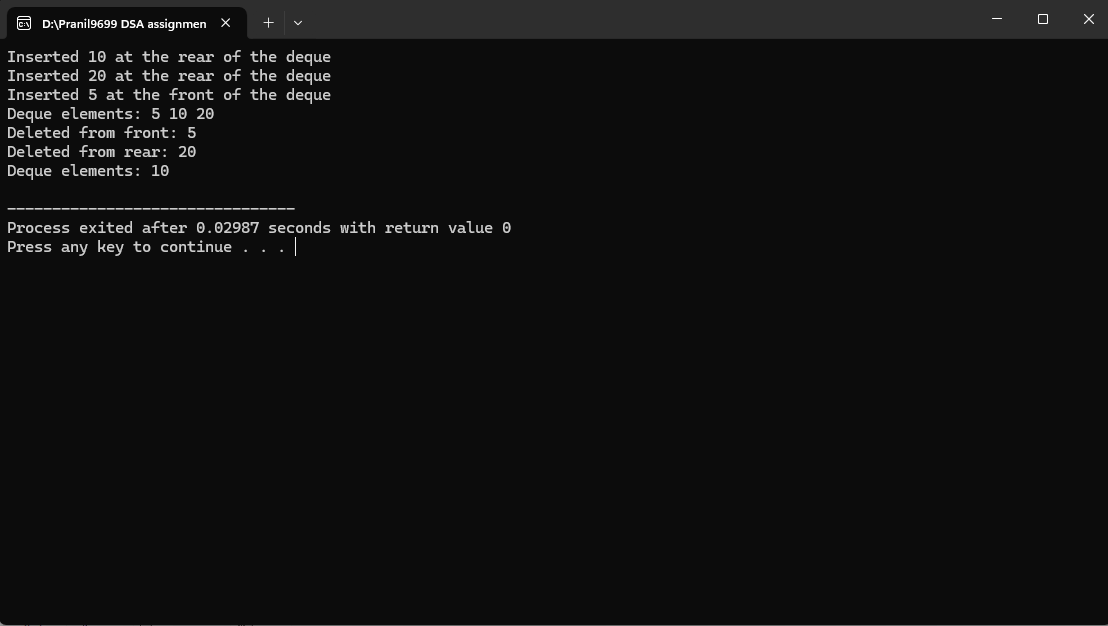
printf("Deleted from rear: %d\n", deleteRear(&deque));

displayDeque(&deque);

return 0;

}

# Output



Q8. Implement singly linked list and perform operations such as insertion, deletion, and traversal.

#Algorithm:

#Code:

#include <stdio.h>

#include <stdlib.h>

// Node structure

typedef struct Node {

int data;

struct Node\* next;

} Node;

// Function to create a new node

Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Function to insert a node at the beginning

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

Node\* newNode = createNode(data);

newNode->next = \*head;

\*head = newNode;

printf("Inserted %d at the beginning\n", data);

}

// Function to insert a node at the end

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

Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

printf("Inserted %d at the end\n", data);

return;

}

Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

printf("Inserted %d at the end\n", data);

}

// Function to delete a node by value

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

if (\*head == NULL) {

printf("List is empty. Cannot delete %d\n", value);

return;

}

// If the value is in the head node

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

Node\* temp = \*head;

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

free(temp);

printf("Deleted %d from the list\n", value);

return;

}

// Traverse the list to find the node to delete

Node\* temp = \*head;

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

temp = temp->next;

}

if (temp->next == NULL) {

printf("Value %d not found in the list\n", value);

} else {

Node\* nodeToDelete = temp->next;

temp->next = nodeToDelete->next;

free(nodeToDelete);

printf("Deleted %d from the list\n", value);

}

}

// Function to traverse and display the list

void traverse(Node\* head) {

if (head == NULL) {

printf("List is empty\n");

return;

}

printf("Linked list elements: ");

Node\* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

int main() {

Node\* head = NULL;

// Perform operations

insertAtBeginning(&head, 10);

insertAtBeginning(&head, 20);

insertAtEnd(&head, 30);

insertAtEnd(&head, 40);

traverse(head);

deleteByValue(&head, 20);

traverse(head);

deleteByValue(&head, 50); // Attempt to delete non-existent value

traverse(head);

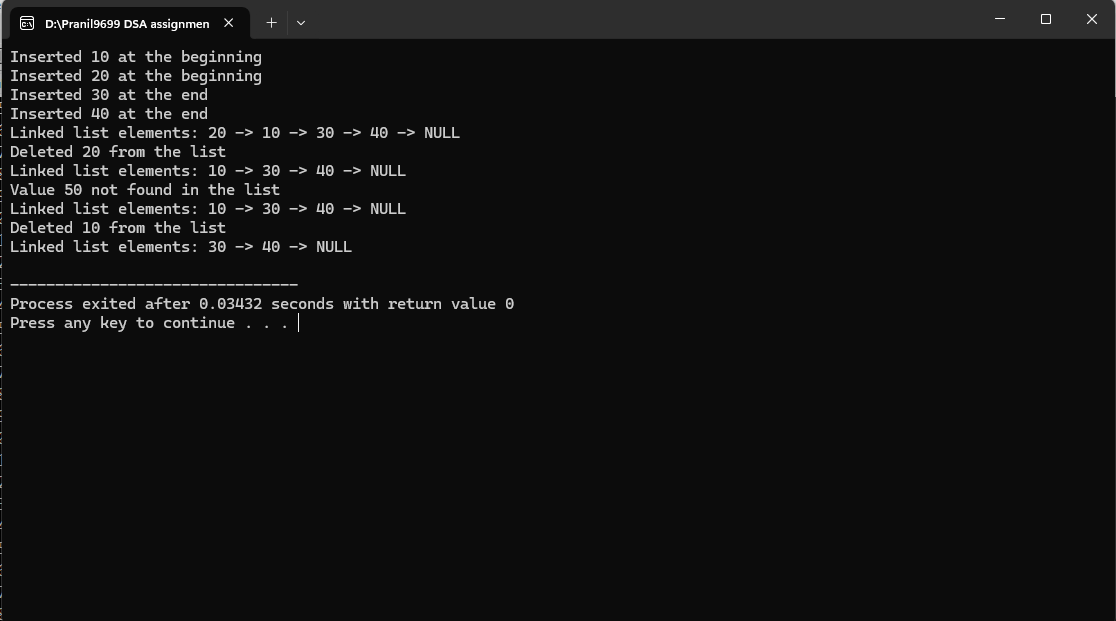
deleteByValue(&head, 10);

traverse(head);

return 0;

}

#Output:



Q9. Extend to doubly and circularly linked lists with all operations.

A. Doubly Linked List

#Algorithm:

#Code:

#include <stdio.h>

#include <stdlib.h>

// Node structure for doubly linked list

typedef struct Node {

int data;

struct Node\* next;

struct Node\* prev;

} Node;

// Function to create a new node

Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

newNode->prev = NULL;

return newNode;

}

// Insert at the beginning

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

Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

} else {

newNode->next = \*head;

(\*head)->prev = newNode;

\*head = newNode;

}

printf("Inserted %d at the beginning\n", data);

}

// Insert at the end

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

Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

} else {

Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

printf("Inserted %d at the end\n", data);

}

// Delete a node by value

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

if (\*head == NULL) {

printf("List is empty. Cannot delete %d\n", value);

return;

}

Node\* temp = \*head;

// If the value is at the head

if (temp->data == value) {

\*head = temp->next;

if (\*head != NULL) (\*head)->prev = NULL;

free(temp);

printf("Deleted %d from the list\n", value);

return;

}

// Traverse the list

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

temp = temp->next;

}

if (temp == NULL) {

printf("Value %d not found in the list\n", value);

return;

}

// Delete the node

if (temp->next != NULL) {

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

}

if (temp->prev != NULL) {

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

}

free(temp);

printf("Deleted %d from the list\n", value);

}

// Traverse the list

void traverse(Node\* head) {

if (head == NULL) {

printf("List is empty\n");

return;

}

Node\* temp = head;

printf("Doubly Linked List elements: ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

int main() {

Node\* head = NULL;

// Perform operations

insertAtBeginning(&head, 10);

insertAtBeginning(&head, 20);

insertAtEnd(&head, 30);

insertAtEnd(&head, 40);

traverse(head);

deleteByValue(&head, 20);

traverse(head);

deleteByValue(&head, 50); // Non-existent value

traverse(head);

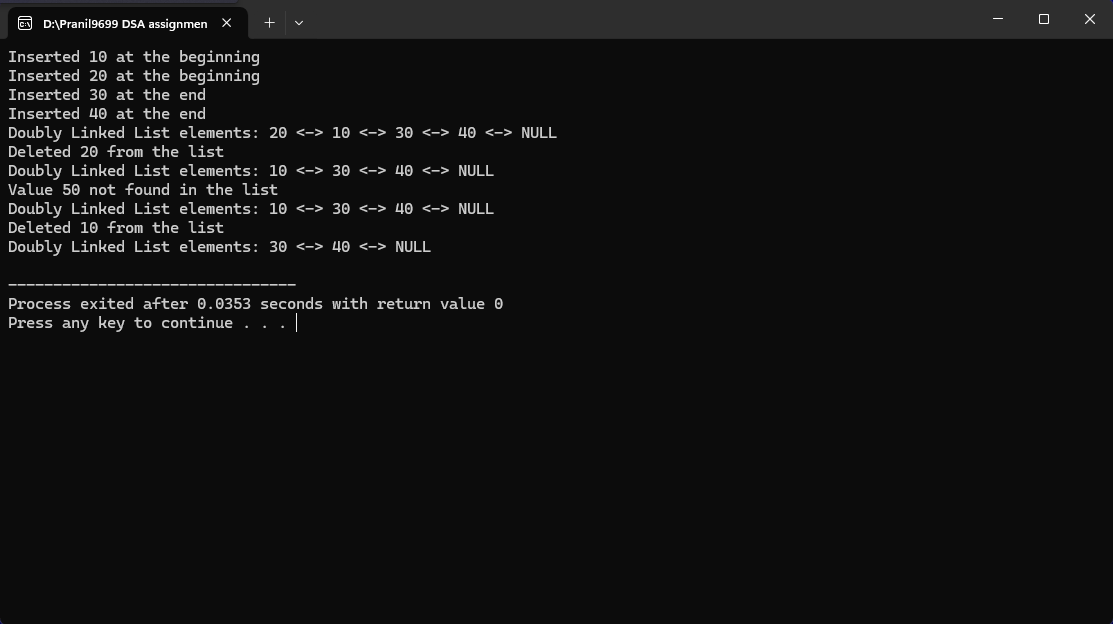
deleteByValue(&head, 10);

traverse(head);

return 0;

}

#Output:



B. Circular Linked List

#Algorithm:

#Code:

#include <stdio.h>

#include <stdlib.h>

// Node structure for circular linked list

typedef struct Node {

int data;

struct Node\* next;

} Node;

// Function to create a new node

Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Insert at the beginning

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

Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

newNode->next = \*head; // Points to itself

} else {

Node\* temp = \*head;

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

temp = temp->next;

}

temp->next = newNode;

newNode->next = \*head;

\*head = newNode;

}

printf("Inserted %d at the beginning\n", data);

}

// Insert at the end

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

Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

newNode->next = \*head; // Points to itself

} else {

Node\* temp = \*head;

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

temp = temp->next;

}

temp->next = newNode;

newNode->next = \*head;

}

printf("Inserted %d at the end\n", data);

}

// Delete a node by value

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

if (\*head == NULL) {

printf("List is empty. Cannot delete %d\n", value);

return;

}

Node\* temp = \*head;

Node\* prev = NULL;

// If the head node is to be deleted

if (temp->data == value) {

prev = \*head;

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

prev = prev->next;

}

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

free(\*head);

\*head = NULL;

} else {

prev->next = temp->next;

\*head = temp->next;

free(temp);

}

printf("Deleted %d from the list\n", value);

return;

}

// Traverse the list to find the node to delete

prev = temp;

temp = temp->next;

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

prev = temp;

temp = temp->next;

}

if (temp == \*head) {

printf("Value %d not found in the list\n", value);

return;

}

prev->next = temp->next;

free(temp);

printf("Deleted %d from the list\n", value);

}

// Traverse the list

void traverse(Node\* head) {

if (head == NULL) {

printf("List is empty\n");

return;

}

Node\* temp = head;

printf("Circular Linked List elements: ");

do {

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

temp = temp->next;

} while (temp != head);

printf("(head)\n");

}

int main() {

Node\* head = NULL;

// Perform operations

insertAtBeginning(&head, 10);

insertAtBeginning(&head, 20);

insertAtEnd(&head, 30);

insertAtEnd(&head, 40);

traverse(head);

deleteByValue(&head, 20);

traverse(head);

deleteByValue(&head, 50); // Non-existent value

traverse(head);

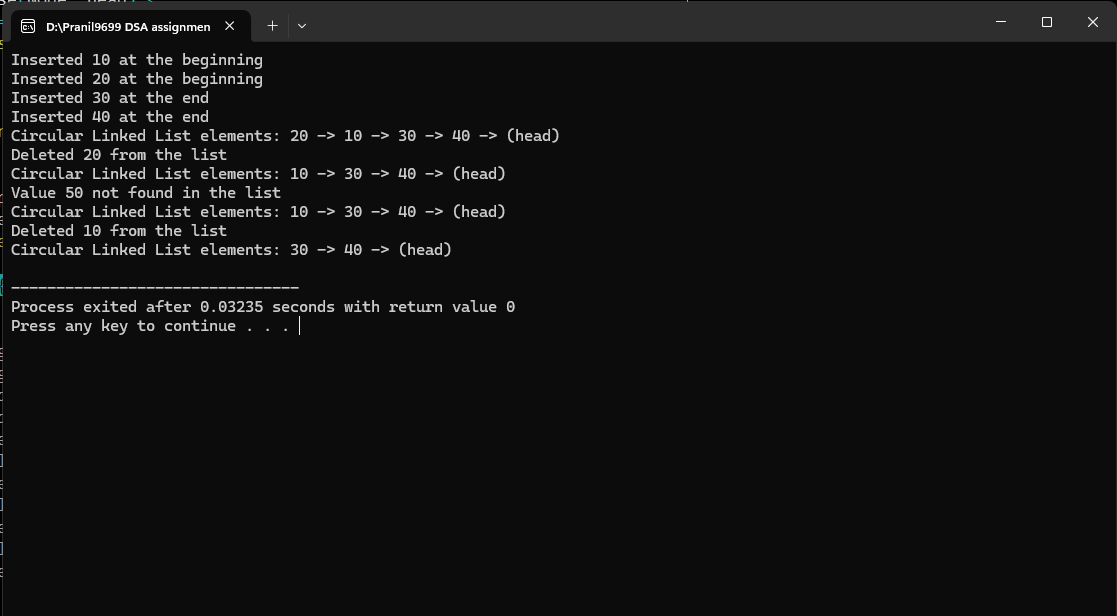
deleteByValue(&head, 10);

traverse(head);

return 0;

}

#Output:



Q10. Implement Linear Search and Binary Search algorithms.

#Algorithm:

#Code:

#include <stdio.h>

// Function for Linear Search

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

int i;

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

if (arr[i] == target) {

return i; // Target found, return index

}

}

return -1; // Target not found

}

// Function for Binary Search (assuming array is sorted)

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

int left = 0, right = size - 1;

while (left <= right) {

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

// Check if target is at mid

if (arr[mid] == target) {

return mid; // Target found at mid

}

// If target is smaller, search in the left half

if (arr[mid] > target) {

right = mid - 1;

}

// If target is larger, search in the right half

else {

left = mid + 1;

}

}

return -1; // Target not found

}

int main() {

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

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

int target = 13;

// Perform Linear Search

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

if (linearResult != -1) {

printf("Linear Search: Element %d found at index %d\n", target,

linearResult);

} else {

printf("Linear Search: Element %d not found\n", target);

}

// Perform Binary Search

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

if (binaryResult != -1) {

printf("Binary Search: Element %d found at index %d\n", target,

binaryResult);

} else {

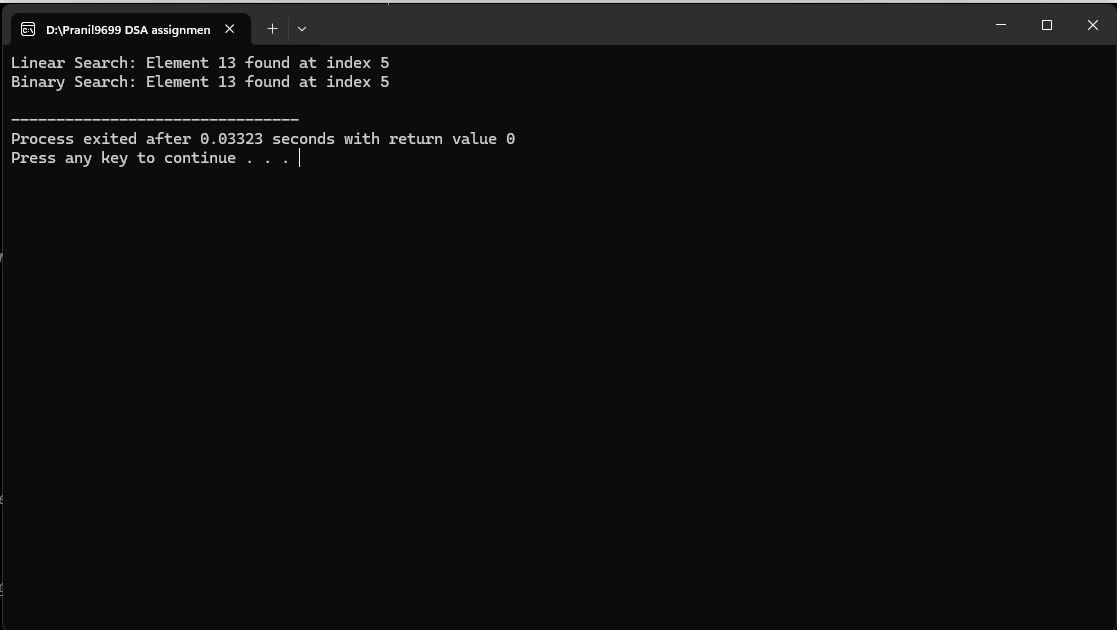
printf("Binary Search: Element %d not found\n", target);

}

return 0;

}

#Output:



Q11 . Implement and compare Selection Sort, Bubble Sort, Insertion Sort, Quick Sort, and Merge Sort.

#Algorithm:

#Code:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Function to print the array

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

int i;

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

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

}

printf("\n");

}

// \*\*Selection Sort\*\* Implementation

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

int i,j;

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

int minIndex = i;

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

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

minIndex = j;

}

}

// Swap the found minimum element with the first element

int temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

}

}

// \*\*Bubble Sort\*\* Implementation

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

int i,j;

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

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

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

// Swap the elements

int temp = arr[j];

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

arr[j + 1] = temp;

}

}

}

}

// \*\*Insertion Sort\*\* Implementation

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

int i;

for ( i = 1; i < size; i++) {

int key = arr[i];

int j = i - 1;

// Move elements of arr[0..i-1] that are greater than key to one position ahead

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

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

j--;

}

arr[j + 1] = key;

}

}

// \*\*Quick Sort\*\* Implementation

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

if (low < high) {

// Partitioning index

int pivot = arr[high];

int i = (low - 1);

int j;

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

if (arr[j] < pivot) {

i++;

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

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

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

int temp = arr[i + 1];

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

arr[high] = temp;

int pi = i + 1;

// Recursively sort elements before and after partition

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

// \*\*Merge Sort\*\* Implementation

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

int n1 = mid - left + 1; // Size of the left subarray

int n2 = right - mid; // Size of the right subarray

// Temporary arrays

int L[n1], R[n2];

// Copy data to temporary arrays L[] and R[]

int i,j;

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

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

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

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

// Merge the temporary arrays back into arr[]

i = 0; // Initial index of first subarray

j = 0; // Initial index of second subarray

int k = left; // Initial index of the merged subarray

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

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

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

// Copy any remaining elements of L[]

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

// Copy any remaining elements of R[]

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

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);

}

}

int main() {

int arr1[] = {64, 34, 25, 12, 22, 11, 90};

int size = sizeof(arr1) / sizeof(arr1[0]);

int arr2[size];

int arr3[size];

int arr4[size];

int arr5[size];

int arr6[size];

int i;

// Make copies of the original array for different sorting algorithms

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

arr2[i] = arr1[i];

arr3[i] = arr1[i];

arr4[i] = arr1[i];

arr5[i] = arr1[i];

arr6[i] = arr1[i];

}

// Measure time for Selection Sort

clock\_t start = clock();

selectionSort(arr2, size);

clock\_t end = clock();

printf("Selection Sort: ");

printArray(arr2, size);

printf("Time: %.6f seconds\n", (double)(end - start) / CLOCKS\_PER\_SEC);

// Measure time for Bubble Sort

start = clock();

bubbleSort(arr3, size);

end = clock();

printf("Bubble Sort: ");

printArray(arr3, size);

printf("Time: %.6f seconds\n", (double)(end - start) / CLOCKS\_PER\_SEC);

// Measure time for Insertion Sort

start = clock();

insertionSort(arr4, size);

end = clock();

printf("Insertion Sort: ");

printArray(arr4, size);

printf("Time: %.6f seconds\n", (double)(end - start) / CLOCKS\_PER\_SEC);

// Measure time for Quick Sort

start = clock();

quickSort(arr5, 0, size - 1);

end = clock();

printf("Quick Sort: ");

printArray(arr5, size);

printf("Time: %.6f seconds\n", (double)(end - start) / CLOCKS\_PER\_SEC);

// Measure time for Merge Sort

start = clock();

mergeSort(arr6, 0, size - 1);

end = clock();

printf("Merge Sort: ");

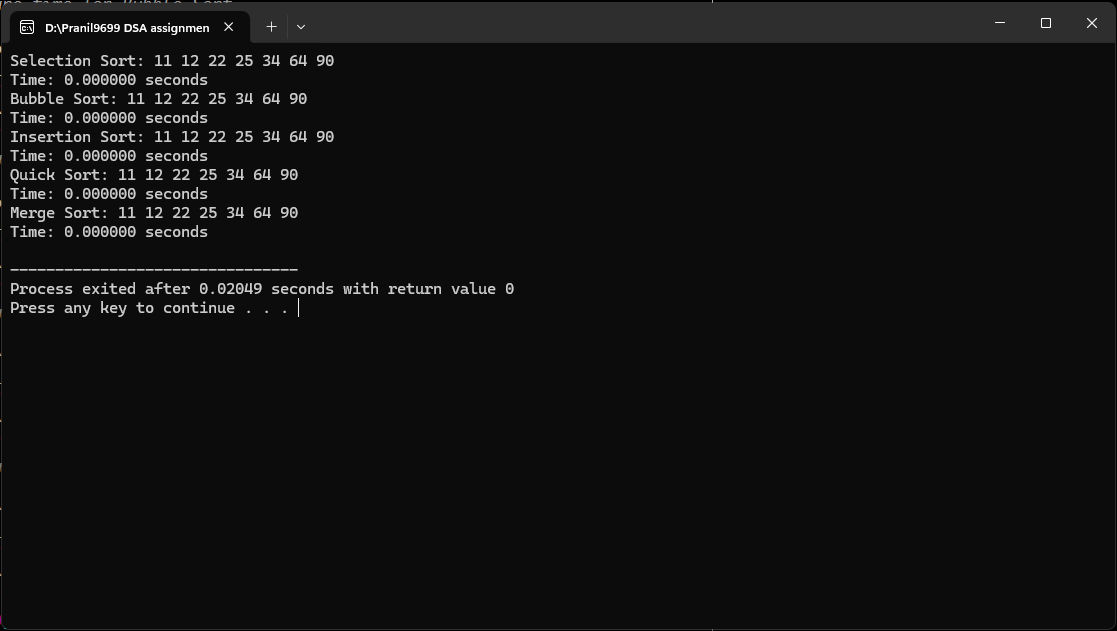
printArray(arr6, size);

printf("Time: %.6f seconds\n", (double)(end - start) / CLOCKS\_PER\_SEC);

return 0;

}

#Output:



Q12. Implement the creation of a binary tree and perform traversals (Inorder,Preorder,Postorder).

#Algorithm:

#Code:

#include <stdio.h>

#include <stdlib.h>

// Structure for a binary tree node

struct Node {

int data;

struct Node \*left, \*right;

};

// Function to create a new node

struct Node\* newNode(int data) {

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

node->data = data;

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

return node;

}

// \*\*Inorder Traversal\*\* (Left -> Root -> Right)

void inorderTraversal(struct Node\* root) {

if (root != NULL) {

inorderTraversal(root->left); // Traverse left subtree

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

inorderTraversal(root->right); // Traverse right subtree

}

}

// \*\*Preorder Traversal\*\* (Root -> Left -> Right)

void preorderTraversal(struct Node\* root) {

if (root != NULL) {

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

preorderTraversal(root->left); // Traverse left subtree

preorderTraversal(root->right); // Traverse right subtree

}

}

// \*\*Postorder Traversal\*\* (Left -> Right -> Root)

void postorderTraversal(struct Node\* root) {

if (root != NULL) {

postorderTraversal(root->left); // Traverse left subtree

postorderTraversal(root->right); // Traverse right subtree

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

}

}

int main() {

// Create the binary tree

struct Node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

root->right->left = newNode(6);

root->right->right = newNode(7);

// Print the traversals

printf("Inorder Traversal: ");

inorderTraversal(root);

printf("\n");

printf("Preorder Traversal: ");

preorderTraversal(root);

printf("\n");

printf("Postorder Traversal: ");

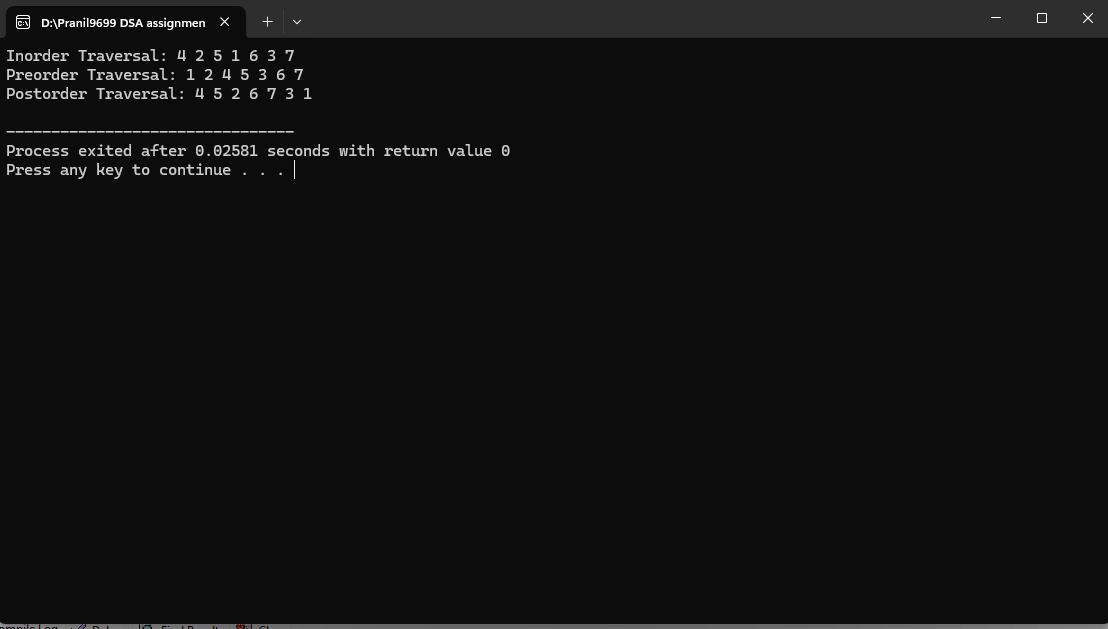
postorderTraversal(root);

printf("\n");

return 0;

}

#Output



Q13. Implement a binary search tree (BST) with insertion, deletion, and searching operations.

#Algorithm:

#Code:

#include <stdio.h>

#include <stdlib.h>

// Structure for a binary tree node

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// Function to create a new node

struct Node\* newNode(int data) {

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

node->data = data;

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

return node;

}

// \*\*Insertion\*\* into the BST

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

if (root == NULL) {

return newNode(data); // If tree is empty, create a new node

}

if (data < root->data) {

root->left = insert(root->left, data); // Insert in the left subtree

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

root->right = insert(root->right, data); // Insert in the right subtree

}

return root; // Return the unchanged node pointer

}

// \*\*Search\*\* for a node in the BST

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

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

return root; // Return NULL if not found, or the node if found

}

if (data < root->data) {

return search(root->left, data); // Search in the left subtree

}

return search(root->right, data); // Search in the right subtree

}

// \*\*Find the minimum node\*\* in the BST (used for deletion)

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

while (root->left != NULL) {

root = root->left; // Keep going left until the minimum node is found

}

return root;

}

// \*\*Delete a node from the BST\*\*

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

if (root == NULL) {

return root; // If tree is empty, return NULL

}

if (data < root->data) {

root->left = delete(root->left, data); // Search for the node to delete in the left subtree

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

root->right = delete(root->right, data); // Search for the node to delete in the right subtree

} else {

// Node with only one child or no child

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;

}

// Node with two children: Get the inorder successor (smallest in the right subtree)

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

// Copy the inorder successor's content to this node

root->data = temp->data;

// Delete the inorder successor

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

}

return root;

}

// \*\*Inorder Traversal\*\* (Left -> Root -> Right)

void inorderTraversal(struct Node\* root) {

if (root != NULL) {

inorderTraversal(root->left); // Traverse left subtree

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

inorderTraversal(root->right); // Traverse right subtree

}

}

int main() {

struct Node\* root = NULL;

// Inserting nodes into the BST

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);

// Print the inorder traversal of the BST

printf("Inorder Traversal (BST): ");

inorderTraversal(root);

printf("\n");

// Search for a node

int searchValue = 40;

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

if (result != NULL) {

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

} else {

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

}

// Delete a node

int deleteValue = 20;

root = delete(root, deleteValue);

printf("Inorder Traversal after deleting %d: ", deleteValue);

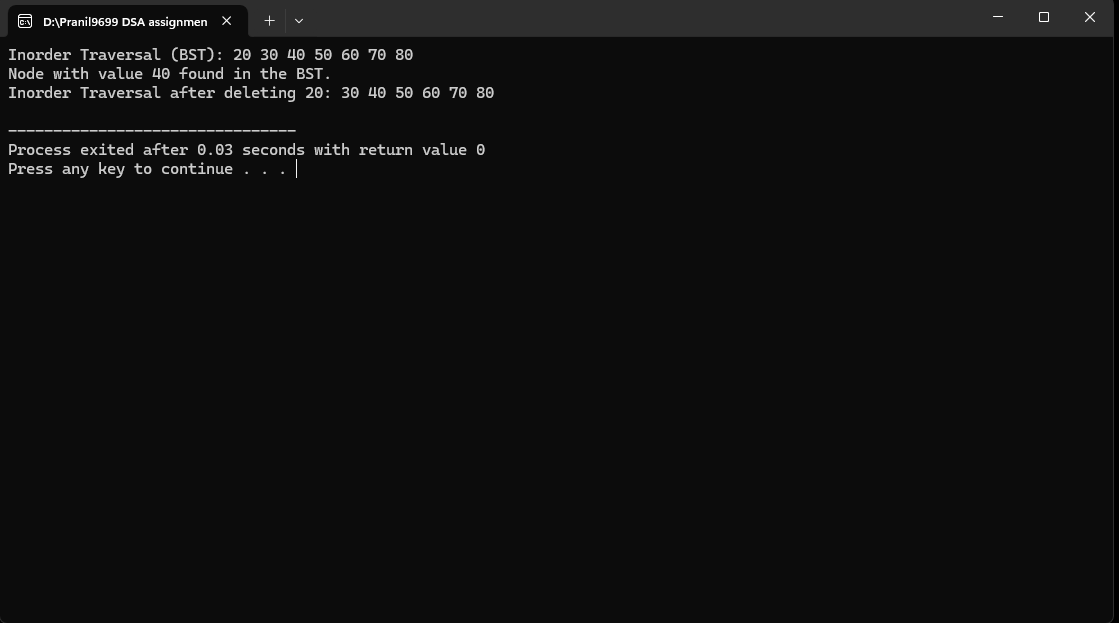
inorderTraversal(root);

printf("\n");

return 0;

}

#Output:



Q14. Implement a heap data structure (min-heap or max-heap) and demonstrate heap insertion and deletion.

#Algorithm:

#Code:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

// Structure for a Min-Heap

struct MinHeap {

int arr[MAX\_SIZE];

int size;

};

// Function to initialize the Min-Heap

void initHeap(struct MinHeap\* heap) {

heap->size = 0;

}

// Function to swap two elements in the heap

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to get the index of the left child of a node

int leftChild(int i) {

return 2 \* i + 1;

}

// Function to get the index of the right child of a node

int rightChild(int i) {

return 2 \* i + 2;

}

// Function to get the index of the parent of a node

int parent(int i) {

return (i - 1) / 2;

}

// Function to heapify the Min-Heap (bubble down)

void minHeapify(struct MinHeap\* heap, int i) {

int left = leftChild(i);

int right = rightChild(i);

int smallest = i;

// Check if left child is smaller than the current smallest

if (left < heap->size && heap->arr[left] < heap->arr[smallest]) {

smallest = left;

}

// Check if right child is smaller than the current smallest

if (right < heap->size && heap->arr[right] < heap->arr[smallest]) {

smallest = right;

}

// If the smallest is not the current node, swap and continue heapifying

if (smallest != i) {

swap(&heap->arr[i], &heap->arr[smallest]);

minHeapify(heap, smallest);

}

}

// Function to insert an element into the Min-Heap

void insert(struct MinHeap\* heap, int value) {

if (heap->size == MAX\_SIZE) {

printf("Heap is full\n");

return;

}

// Insert the new value at the end of the heap

heap->arr[heap->size] = value;

int i = heap->size;

heap->size++;

// Bubble up to maintain the heap property

while (i != 0 && heap->arr[parent(i)] > heap->arr[i]) {

swap(&heap->arr[i], &heap->arr[parent(i)]);

i = parent(i);

}

}

// Function to delete the root (minimum element) from the Min-Heap

int deleteMin(struct MinHeap\* heap) {

if (heap->size <= 0) {

printf("Heap is empty\n");

return -1;

}

if (heap->size == 1) {

heap->size--;

return heap->arr[0];

}

// Get the minimum element (root)

int root = heap->arr[0];

// Replace the root with the last element

heap->arr[0] = heap->arr[heap->size - 1];

heap->size--;

// Heapify the root element to maintain the heap property

minHeapify(heap, 0);

return root;

}

// Function to print the elements of the heap

void printHeap(struct MinHeap\* heap) {

int i;

for ( i = 0; i < heap->size; i++) {

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

}

printf("\n");

}

int main() {

struct MinHeap heap;

initHeap(&heap);

// Inserting elements into the Min-Heap

insert(&heap, 10);

insert(&heap, 15);

insert(&heap, 20);

insert(&heap, 17);

insert(&heap, 8);

insert(&heap, 5);

printf("Min-Heap after insertion: ");

printHeap(&heap);

// Deleting the root (minimum) element

printf("Deleted min element: %d\n", deleteMin(&heap));

printf("Min-Heap after deletion: ");

printHeap(&heap);

return 0;

}

#Output:

