DSA

Hitarth Patel

150096724046

Jensen Huang

Attempt any 6 questions:-

Q1. Write the function to insert a new node after a specific node in a circular linked list?

→

void **insertAtTarget**(Node\* &*head*, int *target*, int *value*) {

if (*head* == nullptr) {

cout << "List is empty, cannot insert after target." << **endl**;

return;

}

Node\* temp = *head*;

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

temp = temp->next;

}

if (temp->data == *target*) {

Node\* newNode = new Node();

newNode->data = *value*;

newNode->next = temp->next;

temp->next = newNode;

return;

} else {

cout << "Value not found in the list" << **endl**;

}

}

Q2. Write the algorithm to delete a node in a linked list?

→

1. Handle the case when the list is empty

○ If head == nullptr, return immediately as there's nothing to delete.

2. Check if the head node is the node to be deleted

○ If head->data == value:

■ Store the current head in temp.

■ Move head to the next node (head = head->next).

■ Delete temp.

■ Return.

3. Search for the node with the given value

○ Initialize two pointers:

■ temp (starting at head) to traverse the list.

■ prev (initially nullptr) to keep track of the previous node.

○ While temp is not nullptr and temp->data is not value:

■ Move prev to temp.

■ Move temp to temp->next.

○ If temp == nullptr, the node was not found. Return.

4. Delete the node if found

○ Adjust prev->next to temp->next (skipping over temp).

○ Delete temp to free memory.

Q3. Implement queue operations? Implement any 3 operations

→

#include <iostream>

using namespace std;

int front, rear;

int \*arr;

int queueSize = 0;

bool **isEmpty**() {

return (queueSize == 0);

}

void **enqueue**(int *data*, int *maxSize*) {

if (queueSize == *maxSize*) {

cout << "Queue is full. Cannot enqueue " << *data* << **endl**;

return;

}

if (rear == -1) {

front = rear = 0;

arr[rear] = *data*;

} else {

arr[++rear] = *data*;

}

queueSize++;

}

void **dequeue**() {

if (**isEmpty**()) {

cout << "Queue is empty. Cannot dequeue" << **endl**;

return;

}

if (front == rear) {

front = rear = -1;

} else {

front++;

}

queueSize--;

}

void **showFront**() {

if (**isEmpty**()) {

cout << "Queue is empty. No front element" << **endl**;

return;

}

cout << "Front element is " << arr[front] << **endl**;

}

void **displayQueue**() {

if (**isEmpty**()) {

cout << "Queue is empty" << **endl**;

return;

}

cout << "Queue elements are: ";

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

cout << arr[i] << " ";

}

cout << **endl**;

}

int **main**() {

front = rear = -1;

queueSize = 0;

int maxSize;

cout << "Enter the maximum size of the queue: ";

cin >> maxSize;

arr = new int[maxSize];

int capacity;

cout << "Enter the capacity of the queue: ";

cin >> capacity;

cout << "Enter the elements of the queue: ";

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

int data;

cin >> data;

**enqueue**(data, maxSize);

}

while (true) {

cout << "1. Enqueue" << **endl**;

cout << "2. Dequeue" << **endl**;

cout << "3. Show Front" << **endl**;

cout << "4. Display Queue" << **endl**;

cout << "5. Exit" << **endl**;

int choice;

cin >> choice;

switch (choice) {

case 1: {

int data;

cout << "Enter the data to enqueue: ";

cin >> data;

**enqueue**(data, maxSize);

break;

}

case 2:

**dequeue**();

break;

case 3:

**showFront**();

break;

case 4:

**displayQueue**();

break;

case 5:

delete[] arr;

return 0;

default:

cout << "Invalid choice. Please choose again." << **endl**;

}

}

delete[] arr;

return 0;

}

Q4. What is a circular queue? Implement any 3 operations?

→

#include <iostream>

using namespace std;

int front, rear;

int \*arr;

int queueSize = 0;

int maxSize;

bool **isEmpty**() {

return (queueSize == 0);

}

void **enqueue**(int *data*) {

if (queueSize == maxSize) {

cout << "Queue is full. Cannot enqueue " << *data* << **endl**;

return;

}

if (rear == -1) {

front = rear = 0;

arr[rear] = *data*;

} else {

rear = (rear + 1) % maxSize;

arr[rear] = *data*;

}

queueSize++;

}

void **dequeue**() {

if (**isEmpty**()) {

cout << "Queue is empty. Cannot dequeue" << **endl**;

return;

}

if (front == rear) {

front = rear = -1;

} else {

front = (front + 1) % maxSize;

}

queueSize--;

}

void **showFront**() {

if (**isEmpty**()) {

cout << "Queue is empty. No front element" << **endl**;

return;

}

cout << "Front element is " << arr[front] << **endl**;

}

void **displayQueue**() {

if (**isEmpty**()) {

cout << "Queue is empty" << **endl**;

return;

}

cout << "Queue elements are: ";

for (int i = front; i != rear; i = (i + 1) % maxSize) {

cout << arr[i] << " ";

}

cout << arr[rear] << **endl**;

}

void **peek**() {

if (**isEmpty**()) {

cout << "Queue is empty. No element to peek" << **endl**;

return;

}

cout << "Peek element is " << arr[front] << **endl**;

}

int **main**() {

front = rear = -1;

queueSize = 0;

cout << "Enter the maximum size of the queue: ";

cin >> maxSize;

arr = new int[maxSize];

int capacity;

cout << "Enter the capacity of the queue: ";

cin >> capacity;

cout << "Enter the elements of the queue: ";

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

int data;

cin >> data;

**enqueue**(data);

}

while (true) {

cout << "1. Enqueue" << **endl**;

cout << "2. Dequeue" << **endl**;

cout << "3. Show Front" << **endl**;

cout << "4. Display Queue" << **endl**;

cout << "5. Peek" << **endl**;

cout << "6. Exit" << **endl**;

int choice;

cin >> choice;

switch (choice) {

case 1: {

int data;

cout << "Enter the data to enqueue: ";

cin >> data;

**enqueue**(data);

break;

}

case 2:

**dequeue**();

break;

case 3:

**showFront**();

break;

case 4:

**displayQueue**();

break;

case 5:

**peek**();

break;

case 6:

delete[] arr;

return 0;

default:

cout << "Invalid choice. Please choose again." << **endl**;

}

}

delete[] arr;

return 0;

}

Q5. Write an algorithm to delete nodes in a circular linked list?

→

1. Handle Empty List:

○ If head == nullptr, return (No nodes exist in the list).

2. Initialize Pointers:

○ Set temp = head (to traverse the list).

○ Set prev = nullptr (to track the previous node).

3. Case 1: Deleting the Head Node (head->data == value)

○ Find the last node (last) in the list by iterating until last->next == head.

○ If the list has only one node (head->next == head):

■ Delete head.

■ Set head = nullptr.

○ Otherwise:

■ Update last->next to point to head->next.

■ Delete head.

■ Update head = last->next.

○ Return (since the deletion is complete).

4. Case 2: Deleting a Non-Head Node

○ Traverse the list while keeping track of the previous node (prev).

○ Continue until:

■ temp reaches head again (ensuring a full traversal).

■ Or temp->data == value (node found).

5. If Node is Found (temp->data == value):

○ Update prev->next = temp->next to remove temp from the list.

○ Delete temp.

6. If Node is Not Found:

○ Print "Node with value X not found!".

Q6. Search an element in a linked list? Write the implementation?

→

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

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

Node\* newNode = new Node();

newNode->data = *value*;

newNode->next = *head*;

*head* = newNode;

}

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

Node\* newNode = new Node();

newNode->data = *value*;

newNode->next = nullptr;

if (*head* == nullptr) {

*head* = newNode;

return;

}

Node\* temp = *head*;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

}

void **insertAfterName**(Node\* &*head*, int *value*, int *name*) {

Node\* newNode = new Node();

newNode->data = *value*;

if (*head* == nullptr) {

*head* = newNode;

return;

}

Node\* temp = *head*;

while (temp->next != nullptr && temp->data != *name*) {

temp = temp->next;

}

if (temp->data == *name*) {

newNode->next = temp->next;

temp->next = newNode;

}

}

void **display**(Node\* *head*) {

Node\* temp = *head*;

while (temp != nullptr) {

cout << temp->data << " -> ";

temp = temp->next;

}

cout << "NULL" << **endl**;

}

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

if (*head* == nullptr) return;

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

Node\* temp = *head*;

*head* = *head*->next;

delete temp;

return;

}

Node\* current = *head*;

Node\* previous = nullptr;

while (current != nullptr && current->data != *value*) {

previous = current;

current = current->next;

}

if (current != nullptr) {

previous->next = current->next;

delete current;

}

}

int **searchNode**(Node\* *head*, int *key*) {

Node\* temp = *head*;

int index = 0;

while (temp != nullptr) {

if (temp->data == *key*) {

return index;

}

temp = temp->next;

index++;

}

return -1; *// Return -1 if key is not found*

}

int **main**() {

Node\* head = nullptr;

**insertAtBeginning**(head, 10);

**insertAtBeginning**(head, 20);

**insertAtEnd**(head, 30);

**insertAtEnd**(head, 40);

**insertAfterName**(head, 45, 30);

cout << "Updated Linked List: ";

**display**(head);

**deleteNode**(head, 30);

cout << "Linked List after deleting 30: ";

**display**(head);

int searchKey = 40;

int index = **searchNode**(head, searchKey);

if (index != -1) {

cout << searchKey << " is found in the Linked List at index " << index << "." << **endl**;

} else {

cout << searchKey << " is not found in the Linked List." << **endl**;

}

return 0;

}

Q7. Write the function to insert a new node at the end position in a linked list?

→

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

Node\* newNode = new Node();

newNode->data = *value*;

newNode->next = nullptr;

if (*head* == nullptr) {

*head* = newNode;

return;

}

Node\* temp = *head*;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

}

Q8. Explain a linked list along with its types. Explain the structure of nodes for singly linked lists.

→

A Linked List is a linear data structure where each element is a separate object, known as a node. Each node contains two items: the data and a reference (or link) to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence.

Types of Linked Lists

1. Singly Linked List: In a singly linked list, each node only contains a reference to the next node in the sequence. This means that traversal can only occur in one direction, from the beginning of the list to the end.
2. Doubly Linked List: In a doubly linked list, each node contains references to both the next and previous nodes in the sequence. This allows for efficient traversal in both directions.
3. Circularly Linked List: In a circularly linked list, the last node in the sequence contains a reference to the first node, forming a circle. This allows for efficient traversal without having to maintain a reference to the beginning of the list.

Structure of Nodes for Singly Linked Lists

In a singly linked list, each node typically contains the following structure:

* Data: This is the actual data stored in the node. The type and size of the data can vary depending on the application.
* Next: This is a reference (or pointer) to the next node in the sequence. In languages that support null or None values, the next reference in the last node is typically set to null to indicate the end of the list.