**C++ Code for Circular Linked List with the Requested Operations**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

// Class to represent Circular Linked List

class CircularLinkedList {

public:

Node\* head;

CircularLinkedList() {

head = nullptr;

}

// Insert at the end

void insertEnd(int value) {

Node\* newNode = new Node();

newNode->data = value;

if (!head) {

head = newNode;

newNode->next = head;

} else {

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head;

}

}

// Insert at the beginning

void insertBeginning(int value) {

Node\* newNode = new Node();

newNode->data = value;

if (!head) {

head = newNode;

newNode->next = head;

} else {

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head;

head = newNode;

}

}

// Insert at a specific position

void insertAtPosition(int value, int position) {

Node\* newNode = new Node();

newNode->data = value;

if (position == 1) {

insertBeginning(value);

return;

}

Node\* temp = head;

int count = 1;

while (temp->next != head && count < position - 1) {

temp = temp->next;

count++;

}

if (count == position - 1) {

newNode->next = temp->next;

temp->next = newNode;

} else {

cout << "Position out of bounds!" << endl;

}

}

// Delete at the end

void deleteEnd() {

if (!head) {

cout << "List is empty!" << endl;

return;

}

Node\* temp = head;

if (temp->next == head) {

delete head;

head = nullptr;

} else {

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

temp = temp->next;

}

delete temp->next;

temp->next = head;

}

}

// Delete at the beginning

void deleteBeginning() {

if (!head) {

cout << "List is empty!" << endl;

return;

}

Node\* temp = head;

if (temp->next == head) {

delete head;

head = nullptr;

} else {

while (temp->next != head) {

temp = temp->next;

}

temp->next = head->next;

delete head;

head = temp->next;

}

}

// Delete a node at a specific position

void deleteAtPosition(int position) {

if (!head) {

cout << "List is empty!" << endl;

return;

}

Node\* temp = head;

if (position == 1) {

deleteBeginning();

return;

}

int count = 1;

while (temp->next != head && count < position - 1) {

temp = temp->next;

count++;

}

if (count == position - 1) {

Node\* nodeToDelete = temp->next;

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

delete nodeToDelete;

} else {

cout << "Position out of bounds!" << endl;

}

}

// Search for a value

void search(int value) {

if (!head) {

cout << "List is empty!" << endl;

return;

}

Node\* temp = head;

int position = 1;

do {

if (temp->data == value) {

cout << "Value " << value << " found at position " << position << endl;

return;

}

temp = temp->next;

position++;

} while (temp != head);

cout << "Value " << value << " not found in the list!" << endl;

}

// Display the list

void display() {

if (!head) {

cout << "List is empty!" << endl;

return;

}

Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

cout << "HEAD" << endl;

}

// Update the value at a specific position

void update(int position, int newValue) {

if (!head) {

cout << "List is empty!" << endl;

return;

}

Node\* temp = head;

int count = 1;

while (temp->next != head && count < position) {

temp = temp->next;

count++;

}

if (count == position) {

temp->data = newValue;

} else {

cout << "Position out of bounds!" << endl;

}

}

// Reverse the list

void reverse() {

if (!head || head->next == head) {

return; // No need to reverse if list is empty or has only one element

}

Node \*prev = nullptr, \*curr = head, \*next = nullptr;

do {

next = curr->next;

curr->next = prev;

prev = curr;

curr = next;

} while (curr != head);

head->next = prev;

head = prev;

}

};

int main() {

CircularLinkedList list;

list.insertEnd(10);

list.insertEnd(20);

list.insertEnd(30);

list.insertBeginning(5);

list.insertAtPosition(15, 3);

cout << "Initial List: ";

list.display();

list.deleteEnd();

cout << "After deleting end: ";

list.display();

list.deleteBeginning();

cout << "After deleting beginning: ";

list.display();

list.deleteAtPosition(2);

cout << "After deleting position 2: ";

list.display();

list.search(20);

list.update(2, 25);

cout << "After updating position 2 to 25: ";

list.display();

list.reverse();

cout << "After reversing: ";

list.display();

return 0;

}

**C++ Code for Array Operations**

#include <iostream>

using namespace std;

class ArrayOperations {

private:

int\* arr;

int capacity;

int size;

public:

// Constructor

ArrayOperations(int capacity) {

this->capacity = capacity;

size = 0;

arr = new int[capacity];

}

// Destructor

~ArrayOperations() {

delete[] arr;

}

// Insert at the end

void insertEnd(int value) {

if (size == capacity) {

cout << "Array is full!" << endl;

return;

}

arr[size] = value;

size++;

}

// Insert at the beginning

void insertBeginning(int value) {

if (size == capacity) {

cout << "Array is full!" << endl;

return;

}

for (int i = size; i > 0; i--) {

arr[i] = arr[i - 1];

}

arr[0] = value;

size++;

}

// Insert at a specific position

void insertAtPosition(int value, int position) {

if (size == capacity) {

cout << "Array is full!" << endl;

return;

}

if (position < 0 || position > size) {

cout << "Invalid position!" << endl;

return;

}

for (int i = size; i > position; i--) {

arr[i] = arr[i - 1];

}

arr[position] = value;

size++;

}

// Delete at the end

void deleteEnd() {

if (size == 0) {

cout << "Array is empty!" << endl;

return;

}

size--;

}

// Delete at the beginning

void deleteBeginning() {

if (size == 0) {

cout << "Array is empty!" << endl;

return;

}

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

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

}

size--;

}

// Delete at a specific position

void deleteAtPosition(int position) {

if (size == 0) {

cout << "Array is empty!" << endl;

return;

}

if (position < 0 || position >= size) {

cout << "Invalid position!" << endl;

return;

}

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

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

}

size--;

}

// Search for a value

void search(int value) {

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

if (arr[i] == value) {

cout << "Value " << value << " found at position " << i << endl;

return;

}

}

cout << "Value " << value << " not found!" << endl;

}

// Display the array

void display() {

if (size == 0) {

cout << "Array is empty!" << endl;

return;

}

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

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

}

cout << endl;

}

// Update the value at a specific position

void update(int position, int newValue) {

if (position < 0 || position >= size) {

cout << "Invalid position!" << endl;

return;

}

arr[position] = newValue;

}

// Reverse the array

void reverse() {

int start = 0, end = size - 1;

while (start < end) {

swap(arr[start], arr[end]);

start++;

end--;

}

}

};

int main() {

ArrayOperations arrOps(10); // Create an array with capacity 10

arrOps.insertEnd(10);

arrOps.insertEnd(20);

arrOps.insertEnd(30);

arrOps.insertBeginning(5);

arrOps.insertAtPosition(15, 2);

cout << "Initial Array: ";

arrOps.display();

arrOps.deleteEnd();

cout << "After deleting end: ";

arrOps.display();

arrOps.deleteBeginning();

cout << "After deleting beginning: ";

arrOps.display();

arrOps.deleteAtPosition(2);

cout << "After deleting position 2: ";

arrOps.display();

arrOps.search(20);

arrOps.update(2, 25);

cout << "After updating position 2 to 25: ";

arrOps.display();

arrOps.reverse();

cout << "After reversing: ";

arrOps.display();

return 0;

}