**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра информационных систем**

отчет

**по практической работе №8**

**по дисциплине «Программирование»**

Тема: Линейные структуры данных. Динамические массивы и двусвязные списки

|  |  |  |
| --- | --- | --- |
| Студент гр. 2372 |  | Соколовский В.Д. |
| Преподаватель |  | Глущенко А.Г. |

Санкт-Петербург

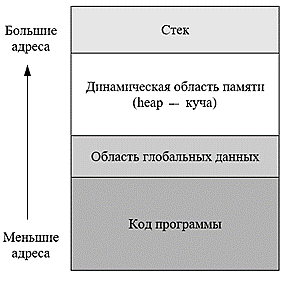
2023

**Цель работы.**

Изучение свойств и организация динамических массивов и двусвязных списков; получение практических навыков в работе с динамическими массивами и двусвязными списками; проведение сравнительной характеристики скорости вставки, получения и удаления элементов из них.

**Основные теоретические положения.**

Схема распределения памяти под программу показана на рис. 1.



*Рис. 1.* Схема распределения памяти под программу

Область кода программы предназначена для хранения инструкций функций программы, обеспечивающих обработку данных.

Данные в программе представляются переменными и константами.

Для хранения глобальных данных предназначена область глобальных данных.

Стек программы используется при вызове функций для передачи параметров и хранения локальных данных.

Распределение памяти для хранения всех обычных переменных осуществляется компилятором, адреса и объемы соответствующих участков памяти (в области глобальных данных) жестко закреплены за этими переменными на все время работы программы и изменены быть не могут.

Однако во многих задачах невозможно заранее предсказать, сколько места (количество переменных, объемы массивов и т. д.) потребуется для решения задачи – это так называемые задачи с неопределенной размерностью.

Решить эту проблему можно лишь в том случае, если иметь механизм, позволяющий создавать новые объекты по мере возникновения необходимости в этих объектах или изменять объемы памяти, выделенные под эти объекты (например, объемы массивов).

Между областью глобальных данных и стеком располагается так называемая динамическая область памяти, которую и можно использовать в процессе работы программы для реализации механизма динамического управления памятью.

Для того чтобы создать в динамической области некоторый объект, необходима одна обычная переменная-указатель (не динамическая переменная). Сколько таких объектов понадобится для одновременной обработки, столько необходимо иметь обычных переменных-указателей. Таким образом, проблема задач неопределенной размерности созданием одиночных динамических объектов решена быть не может.

Решить эту проблему поможет возможность создавать в динамической области памяти массивы объектов с таким количеством элементов, которое необходимо в данный момент работы программы, т. е. создание динамических массивов. Действительно, для представления массива требуется всего одна переменная-указатель, а в самом массиве, на который ссылается этот указатель, может быть столько элементов, сколько требуется в данный момент времени.

Для создания одномерного динамического массива, элементами которого являются, например, действительные числа, используется следующий синтаксис инструкции new (стиль С++):

double \*Arr = new double [100];

Освободить динамическую область от этого массива можно с помощью инструкции delete:

delete [] Arr;

После этого занятый участок памяти будет возвращен в список свободной памяти и может быть повторно использован для размещения других динамических объектов.

Язык C++ поддерживает и «старый», заимствованный от языка C, стиль работы с динамической областью. Довольно часто бывает полезно использовать именно этот механизм управления динамической памятью.

В языке C отсутствуют инструкции new и delete. Вместо них для управления динамической памятью используются библиотечные функции:

// Блок прототипов функций

{

void \*malloc (size);

void \*calloc(num, size);

void free( void \*memblock);

void \*realloc( void \*memblock, size);

}

Функция malloc выделяет в динамической области size байт памяти и возвращает адрес этого участка в виде указателя (void \*).

Поскольку возвращаемый указатель не привязан ни к какому типу данных, при работе с ним потребуется явное приведение типов данных (см. пример далее).

Функция calloc выделяет в динамической области size \* num байт памяти и возвращает адрес этого участка в виде указателя (void \*).

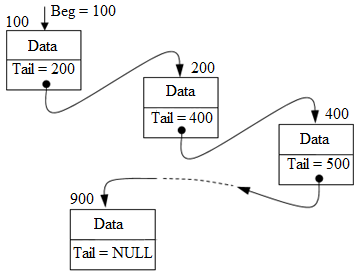
Функция free освобождает участок динамической памяти по адресу memblock и возвращает его в список свободной памяти для повторного использования.

Функция realloc позволяет изменить размер (уменьшить или увеличить) ранее выделенной по адресу memblock памяти, установив новый размер выделенного участка равным size байт. При увеличении размера выделенного участка данные, которые хранились в старом участке, копируются в новый участок памяти. При уменьшении объема выделенного участка данные, которые хранились в нем, усекаются до нового размера. Функция возвращает указатель на область памяти нового размера.

Работа с одномерным динамическим массивом осуществляется так же, как и с обычным. При этом стиль использования динамических массивов С имеет весомое преимущество над С++, которое заключается в изменении размерности массива. Дело в том, что в C++ нет функций увеличения размерности. Увеличить размер массива можно, создав новый динамический массив нужной размерности, скопировав данные из старого массива в новый и освободив память от старого массива.

Одномерный однонаправленный список представляет собой совокупность отдельных элементов, каждый из которых содержит две части – информационную () и адресную ().

Информационная часть предназначена для хранения полезных данных и может иметь практически любой тип. Адресная часть каждого элемента содержит адрес следующего элемента списка. Схематическое изображение такого списка представлено на рис. 2.



*Рис. 2.* Схематическое изображение односвязного списка

Для работы со списком достаточно знать только адрес его первого элемента (). Зная адрес первого элемента списка можно последовательно получить доступ к любому другому его элементу.

Поскольку каждый элемент списка должен иметь две части, логичнее всего представить его в виде следующей структуры:

struct list

{

int data;

list \*tail;

};

Типовыми операциями при работе со списками являются:

1. создание списка;
2. освобождение памяти от списка (удаление списка);
3. доступ к заданному элементу списка для манипуляций с его информационной частью;
4. добавление нового элемента к списку;
5. удаление элемента из списка;
6. перестановка элемента списка на новую позицию внутри списка.

Достоинством подобных структур является простота добавления, удаления и перестановки элементов списка, которые осуществляются путем манипуляций с адресными частями без перезаписи всего списка.

Одним из недостатков односвязных списков является то, что узел (элемент списка) имеет указатель только на следующий элемент. Вернуться из текущего элемента к предыдущему явным способом невозможно.

Каждый узел двусвязного (двунаправленного) линейного списка содержит два поля указателей – на следующий и на предыдущий узлы. Указатель на предыдущий узел корня списка содержит нулевое значение. Указатель последнего узла также содержит нулевое значение.

Поскольку каждый элемент списка должен иметь три части, логичнее всего представить его в виде следующей структуры:

struct list

{

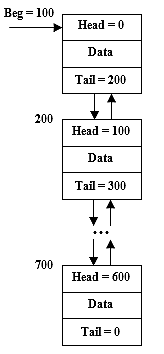
int data;

list \*head;

list \*tail;

};

На рис. 3 показано схематическое представление двусвязного списка. Поле содержит адрес предыдущего элемента, поле содержит адрес следующего элемента списка. Такая организация списка позволяет перемещаться по его элементам в двух направлениях.



*Рис. 3.* Схематическое изображение двусвязного списка

Основные действия, производимые над узлами двусвязного линейного списка (ДЛС):

1. инициализация списка;
2. добавление узла в список;
3. удаление узла из списка;
4. удаление корня списка;
5. вывод элементов списка;
6. вывод элементов списка в обратном порядке;
7. взаимообмен двух узлов списка.

Порядок действия очень похож на односвязный линейный список, но необходимо учитывать, что в двусвязном списке имеется два указателя: на следующий и предыдущий элементы.

**Постановка задачи.**

Необходимо реализовать программу, которая выполняет следующие действия.

1. Формирование целочисленного одномерного массива размерности N, где:

a) пользователь вводит количество элементов в массиве, который будет автоматически заполняться случайными числами (0 до 99);

б) пользователь вводит в консоль элементы массива, N определяется автоматически по количеству введенных элементов;

в) массив считывается с файла, N определяется как количество элементов массива в файле.

2. Определение скорости создания динамического массива п. 1.

3. Вставка, удаление и получение элемента массива. Удаление и получение элемента необходимо реализовать по индексу и по значению.

4. Определение скорости вставки, удаления и получения элемента массива п. 3.

5. Формирование двусвязного списка размерности N, где:

a) пользователь вводит количество элементов в списке, который будет автоматически заполняться случайными числами (0 до 99);

б) пользователь вводит в консоль элементы списка, N определяется автоматически по количеству введенных элементов;

в) список считывается с файла, N определяется как количество элементов списка в файле.

6. Определение скорости создания двусвязного списка п. 5.

7. Вставка, удаление и получение элемента двусвязного списка. Удаление и получение элемента необходимо реализовать по индексу и по значению.

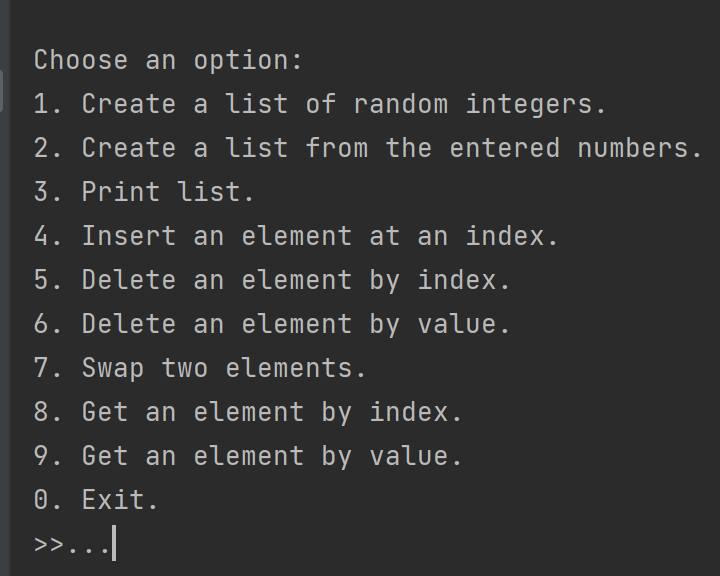
8. Определение скорости вставки, удаление и получения элемента двусвязного списка п. 7.

Должна быть возможность запуска каждого пункта многократно, если есть возможность (если в списке/массиве нет элементов, то нельзя ничего удалить и об этом нужно сообщить пользователю). Необходимо сравнить результаты. Для этого пункты 1–4 и 5–8 должны принимать одинаковые значения.

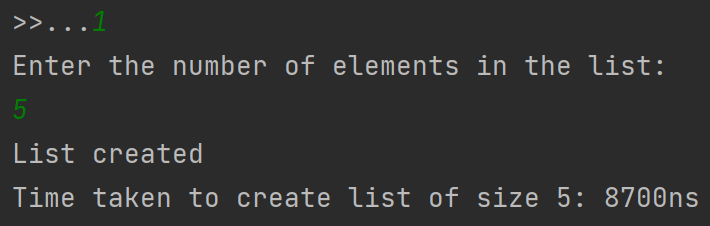
**Выполнение работы.**

Код программы представлен в приложении А.

1. Пользователю предоставляется интерактивное меню:



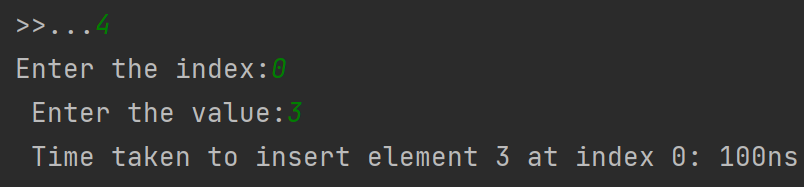
1. Формирование целочисленного одномерного массива размерности N, где пользователь вводит количество элементов в массиве, который будет автоматически заполняться случайными числами (0 до 99); Определение скорости его создания;



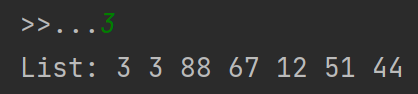
Вывод списка:



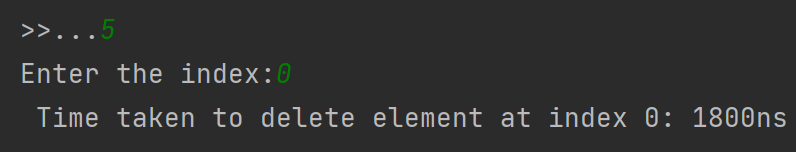
1. Вставка элементов по индексу и определение его скорости:



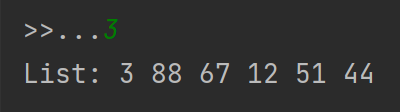
Вывод списка:



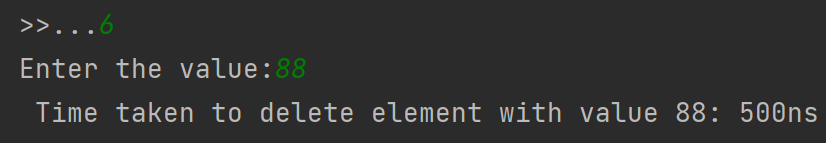
1. Удаление элемента по индексу и определение его скорости:



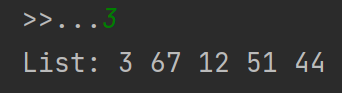
Вывод списка:



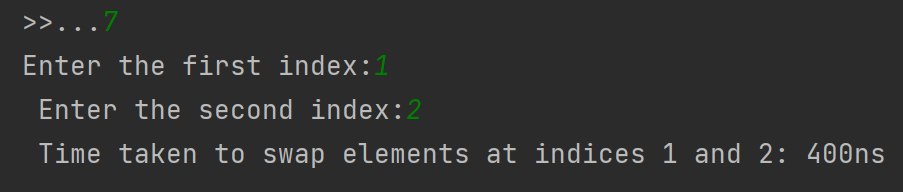
1. Удаление элемента и определение его скорости:



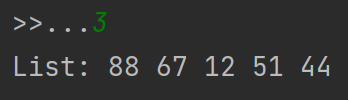
Вывод списка:



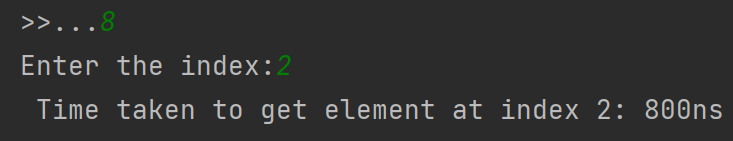
1. Обмен элементов и определение его скорости:



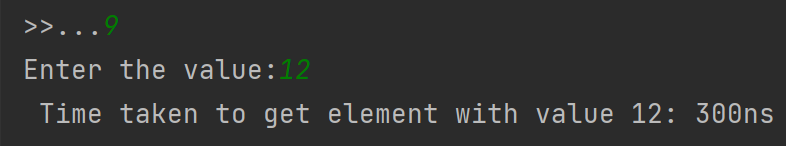
Вывод списка:



1. Получение элемента по индексу и определение его скорости:



1. Получение элемента и определение его скорости:



**Выводы**.

В ходе данной практической работы я изучил свойства и организацию динамических массивов и двусвязных списков; получил практические навыки в работе с динамическими массивами и двусвязными списками; провел сравнительную характеристику скорости вставки, получения и удаления элементов из них

Приложение А

**КОД ПРОГРАММЫ**

**#include <iostream>**

**#include <chrono>**

**using namespace std;**

**using namespace chrono;**

**struct List{**

**int data;**

**List \*head;**

**List \*tail;**

**};**

**List\* createListRandom(int n){**

**List\* head = new List();**

**head->data = rand() % 100;**

**head->head = nullptr;**

**List \*current = head;**

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

**current->tail = new List();**

**current->tail->data = rand() % 100;**

**current->tail->head = current;**

**current = current->tail;**

**}**

**current->head = nullptr;**

**return head;**

**}**

**List\* createList() {**

**List\* head = nullptr, \*tail = nullptr;**

**int n = 0, value;**

**cout << "Enter the elements of the list (end with a non-integer character): ";**

**while(cin >> value){**

**List\* newNode = new List;**

**newNode->data = value;**

**newNode->head = tail;**

**newNode->tail = nullptr;**

**if(tail != nullptr){**

**tail->tail = newNode;**

**}**

**tail = newNode;**

**if(n == 0){**

**head = newNode;**

**}**

**n++;**

**}**

**cin.clear();**

**cin.ignore();**

**return head;**

**}**

**void printList(List\* head){**

**List \*current = head;**

**cout << "List: ";**

**while (current != nullptr) {**

**cout << current->data << " ";**

**current = current->tail;**

**}**

**cout << endl;**

**}**

**void deleteList(List\* head){**

**List \*current;**

**while(head != nullptr){**

**current = head->tail;**

**delete head;**

**head = current;**

**}**

**}**

**void insertElement(List\* &head, int index, int value){**

**List \*temp = new List();**

**temp->data = value;**

**if(head == nullptr || index <= 0){**

**temp->head = nullptr;**

**temp->tail = head;**

**if(head != nullptr){**

**head->head = temp;**

**}**

**head = temp;**

**}**

**else{**

**List \*current = head;**

**for(int i = 0; i < index - 1 && current->tail != nullptr; i++){**

**current = current->tail;**

**}**

**temp->head = current;**

**temp->tail = current->tail;**

**if(current->tail != nullptr){**

**current->tail->head = temp;**

**}**

**current->tail = temp;**

**}**

**}**

**void deleteElementByIndex(List\* &head, int index){**

**if(head == nullptr){**

**return;**

**}**

**List \*current = head;**

**if(index == 0){**

**head = head->tail;**

**if(head != nullptr){**

**head->head = nullptr;**

**}**

**delete current;**

**}**

**else{**

**for(int i = 0; i < index && current != nullptr; i++){**

**current = current->tail;**

**}**

**if(current == nullptr){**

**return;**

**}**

**current->head->tail = current->tail;**

**if(current->tail != nullptr){**

**current->tail->head = current->head;**

**}**

**delete current;**

**}**

**}**

**void deleteElementByValue(List\* &head, int value){**

**if(head == nullptr){**

**return;**

**}**

**List \*current = head;**

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

**head = head->tail;**

**if(head != nullptr){**

**head->head = nullptr;**

**}**

**delete current;**

**return;**

**}**

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

**current = current->tail;**

**}**

**if(current->tail == nullptr){**

**return;**

**}**

**List \*temp = current->tail;**

**current->tail = temp->tail;**

**if(temp->tail != nullptr){**

**temp->tail->head = current;**

**}**

**delete temp;**

**}**

**void swapElements(List\* &head, int index1, int index2){**

**if(head == nullptr || index1 == index2){**

**return;**

**}**

**List \*temp, \*current1 = head, \*current2 = head;**

**for(int i = 0; i < index1 && current1 != nullptr; i++){**

**current1 = current1->tail;**

**}**

**for (int i = 0; i < index2 && current2 != nullptr; i++){**

**current2 = current2->tail;**

**}**

**if(current1 == nullptr || current2 == nullptr){**

**return;**

**}**

**if(current2->tail == current1){**

**swap(current1, current2);**

**}**

**if(current1->tail == current2){**

**temp = current1;**

**current1 = current2;**

**current2 = temp;**

**}**

**if(current1->head != nullptr){**

**current1->head->tail = current2;**

**}**

**if(current1->tail != nullptr){**

**current1->tail->head = current2;**

**}**

**if(current2->head != nullptr){**

**current2->head->tail = current1;**

**}**

**if(current2->tail != nullptr){**

**current2->tail->head = current1;**

**}**

**temp = current1->head;**

**current1->head = current2->head;**

**current2->head = temp;**

**temp = current1->tail;**

**current1->tail = current2->tail;**

**current2->tail = temp;**

**if(head == current1){**

**head = current2;**

**}**

**else if(head == current2){**

**head = current1;**

**}**

**}**

**List\* getElementByIndex(List\* &head, int index){**

**if(head == nullptr){**

**return nullptr;**

**}**

**List \*current = head;**

**for(int i = 0; i < index && current != nullptr; i++){**

**current = current->tail;**

**}**

**return current;**

**}**

**List\* getElementByValue(List\* &head, int value){**

**if(head == nullptr){**

**return nullptr;**

**}**

**List \*current = head;**

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

**current = current->tail;**

**}**

**return current;**

**}**

**void testCreateListRandom(int n){**

**auto start = high\_resolution\_clock::now();**

**List \*head = createListRandom(n);**

**auto end = high\_resolution\_clock::now();**

**auto duration = duration\_cast<nanoseconds>(end - start);**

**cout << "Time taken to create list of size " << n << ": " << duration.count() << "ns\n";**

**deleteList(head);**

**}**

**void testInsertElement(List\* &head, int index, int value){**

**auto start = high\_resolution\_clock::now();**

**insertElement(head, index, value);**

**auto end = high\_resolution\_clock::now();**

**auto duration = duration\_cast<nanoseconds>(end - start);**

**cout << "Time taken to insert element " << value << " at index " << index << ": " << duration.count() << "ns\n";**

**}**

**void testDeleteElementByIndex(List\* &head, int index){**

**auto start = high\_resolution\_clock::now();**

**deleteElementByIndex(head, index);**

**auto end = high\_resolution\_clock::now();**

**auto duration = duration\_cast<nanoseconds>(end - start);**

**cout << "Time taken to delete element at index " << index << ": " << duration.count() << "ns\n";**

**}**

**void testDeleteElementByValue(List\* &head, int value){**

**auto start = high\_resolution\_clock::now();**

**deleteElementByValue(head, value);**

**auto end = high\_resolution\_clock::now();**

**auto duration = duration\_cast<nanoseconds>(end - start);**

**cout << "Time taken to delete element with value " << value << ": " << duration.count() << "ns\n";**

**}**

**void testSwapElements(List\* &head, int index1, int index2){**

**auto start = high\_resolution\_clock::now();**

**swapElements(head, index1, index2);**

**auto end = high\_resolution\_clock::now();**

**auto duration = duration\_cast<nanoseconds>(end - start);**

**cout << "Time taken to swap elements at indices " << index1 << " and " << index2 << ": " << duration.count() << "ns\n";**

**}**

**void testGetElementByIndex(List\* &head, int index){**

**auto start = high\_resolution\_clock::now();**

**List \*element = getElementByIndex(head, index);**

**auto end = high\_resolution\_clock::now();**

**auto duration = duration\_cast<nanoseconds>(end - start);**

**if(element != nullptr){**

**cout << "Time taken to get element at index " << index << ": " << duration.count() << "ns\n";**

**}**

**else{**

**cout << "Element at index " << index << " does not exist.\n";**

**}**

**}**

**void testGetElementByValue(List\* &head, int value){**

**auto start = high\_resolution\_clock::now();**

**List \*element = getElementByValue(head, value);**

**auto end = high\_resolution\_clock::now();**

**auto duration = duration\_cast<nanoseconds>(end - start);**

**if(element != nullptr){**

**cout << "Time taken to get element with value " << value << ": " << duration.count() << "ns\n";**

**}**

**else{**

**cout << "Element with value " << value << " does not exist.\n";**

**}**

**}**

**int main(){**

**srand(time(0));**

**List \*head = nullptr;**

**int choice, n = 0, index, value;**

**while(true){**

**cout << "\n\nChoose an option:\n";**

**cout << "1. Create a list of random integers.\n";**

**cout << "2. Create a list from the entered numbers.\n";**

**cout << "3. Print list.\n";**

**cout << "4. Insert an element at an index.\n";**

**cout << "5. Delete an element by index.\n";**

**cout << "6. Delete an element by value.\n";**

**cout << "7. Swap two elements.\n";**

**cout << "8. Get an element by index.\n";**

**cout << "9. Get an element by value.\n";**

**cout << "0. Exit.\n";**

**cout << ">>...\n";**

**cin >> choice;**

**switch(choice){**

**case 1: {**

**cout << "Enter the number of elements in the list: " << endl;**

**cin >> n;**

**head = createListRandom(n);**

**cout << "List created" << endl;**

**testCreateListRandom(n);**

**break;**

**}**

**case 2: {**

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

**head = createList();**

**cout << "List created" << endl;**

**break;**

**}**

**case 3: {**

**if(head == nullptr){**

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

**}**

**else{**

**printList(head);**

**}**

**break;**

**}**

**case 4: {**

**if(head == nullptr){**

**cout << "The list is empty.\n";**

**break;**

**}**

**cout << "Enter the index: ";**

**cin >> index;**

**cout << "Enter the value: ";**

**cin >> value;**

**insertElement(head, index, value);**

**testInsertElement(head, index, value);**

**break;**

**}**

**case 5: {**

**if(head == nullptr){**

**cout << "The list is empty.\n";**

**break;**

**}**

**cout << "Enter the index: ";**

**cin >> index;**

**testDeleteElementByIndex(head, index);**

**break;**

**}**

**case 6: {**

**if(head == nullptr){**

**cout << "The list is empty.\n";**

**break;**

**}**

**cout << "Enter the value: ";**

**cin >> value;**

**testDeleteElementByValue(head, value);**

**break;**

**}**

**case 7: {**

**if(head == nullptr){**

**cout << "The list is empty.\n";**

**break;**

**}**

**cout << "Enter the first index: ";**

**cin >> index;**

**cout << "Enter the second index: ";**

**cin >> value;**

**testSwapElements(head, index, value);**

**break;**

**}**

**case 8: {**

**if(head == nullptr){**

**cout << "The list is empty.\n";**

**break;**

**}**

**cout << "Enter the index: ";**

**cin >> index;**

**testGetElementByIndex(head, index);**

**break;**

**}**

**case 9: {**

**if(head == nullptr){**

**cout << "The list is empty.\n";**

**break;**

**}**

**cout << "Enter the value: ";**

**cin >> value;**

**testGetElementByValue(head, value);**

**break;**

**}**

**case 0: {**

**exit(0);**

**}**

**default: {**

**cout << "Invalid choice.\n";**

**}**

**}**

**}**

**}**