**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра информационных систем**

отчет

**по практической работе №2**

**по дисциплине «Программирование»**

Тема: Динамические массивы, списки.

|  |  |  |
| --- | --- | --- |
| Студент гр. 2372 |  | Гечис В.Р. |
| Преподаватель |  | Глущенко А. Г. |

Санкт-Петербург

2023

**Цель работы.**

изучение свойств и организация 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.**

## Динамические объекты

В C++ можно использовать различные типы объектов, которые различаются по использованию памяти. Так, глобальные объекты создаются при запуске программы и освобождаются при ее завершении. Локальные автоматические объекты создаются в блоке кода и удаляются, когда этот блок кода завершает работу. Локальные статические объекты создаются перед их первым использованием и освобождаются при завершении программы.

Глобальные, а также статические локальные объекты помещаются в статической памяти, а локальные автоматические объекты размещаются в стеке. Объекты в статической памяти и стеке создаются и удаляются компилятором. Статическая память очищается при завершении программы, а объекты из стека существуют, пока выполняется блок, в котором они определены.

В дополнение к этим типам в C++ можно создавать динамические объекты. Продолжительность их жизни не зависит от того, где они созданы. Динамические объекты существуют, пока не будут удалены явным образом. Динамические объекты размещаются в динамической памяти.

Для управления динамическими объектами в С++ применяются операторы **new** и **delete**.

Оператор **new** выделяет место в динамической памяти для объекта и возвращает указатель на этот объект.

Оператор **delete** получает указатель на динамический объект и удаляет его из памяти.

Создание динамического объекта:

int \*ptr = new int;

Оператор **new** создает новый объект типа int в динамической памяти и возвращает указатель на него. Значение такого объекта неопределенно.

Также можно инициализировать объект при создании:

int \*p1 = new int(); // значение по умолчанию - 0

std::cout << "p1: " << \*p1 << "\n"; // 0

int \*p2 = new int(12);

std::cout << "p2: " << \*p2 << "\n"; // 12

## Освобождение памяти

Динамические объекты будут существовать пока не будут явным образом удалены. И после завершения использования динамических объектов следует освободить их память с помощью оператора **delete**:

int \*p1 = new int(12);

std::cout << "p1: " << \*p1 << "\n"; // 0

delete p1;

Особенно это надо учитывать, если динамический объект создается в одной части кода, а используется в другой. Например:

#include <iostream>

int\* createPtr(int value)

{

int \*ptr = new int(value);

return ptr;

}

void usePtr()

{

int \*p1 = createPtr(10);

std::cout << \*p1 << "\n"; // 10

delete p1; // объект надо освободить

}

int main()

{

usePtr();

return 0;

}

 В функции **usePtr** получаем из функции **createPtr** указатель на динамический объект. Однако после выполнения функции **usePtr** этот объект автоматически не удаляется из памяти (как это происходит в случае с локальными автоматическими объектами). Поэтому его надо явным образом удалить, использовав оператор **delete**.

Использование объекта по указателю после его удаления или повторное применение оператора **delete** к указателю могут привести к непредсказуемым результатам:

int \*p1 = new int(12);

std::cout << \*p1 << std::"\n"; // 0

delete p1;

// ошибочные сценарии

std::cout << \*p1 << std::"\n"; // объект по указателю p1 уже удален!

delete p1; // объект по указателю p1 уже удален!

Поэтому следует удалять объект только один раз.

Также нередко имеет место ситуация, когда на один и тот же динамический объект указывают сразу несколько указателей. Если оператор **delete** применен к одному из указателей, то память объекта освобождается, и по второму указателю этот объект  использовать уже невозможно. Если же после этого ко второму указателю применить оператор **delete**, то динамическая память может быть нарушена.

В то же время недопустимость указателей после применения к ним оператора **delete** не означает, что эти указатели невозможно использовать. Их можно использовать, если присвоить им адрес другого объекта:

#include <iostream>

int main()

{

int \*p1 = new int(12);

int \*p2 = p1;

delete p1; // адреса в p1 и p2 недопустимы

p1 = new int(11); // p1 указывает на новый объект

std::cout << \*p1 << "\n"; // 11

delete p1;

return 0;

}

Здесь после удаления объекта, на который указывает **p1**, этому указателю передается адрес другого объекта в динамической памяти. Соответственно мы также можем использовать указатель **p1**. В то же время адрес в указателе **p2** по прежнему будет недействительным.

## Одномерные динамические массивы

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 используется следующий синтаксис инструкции new (стиль С++):

int \*Arr = new int [100];

Причем в этом случае оператор new также возвращает указатель на объект типа int - первый элемент в созданном массиве.

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

delete [] Arr;

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

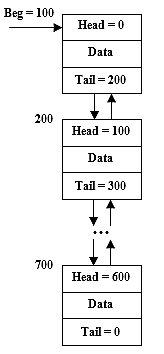
{

int data;

list \*head;

list \*tail;

};



 Поле  **Head** содержит адрес предыдущего элемента, поле **Tail** содержит адрес следующего элемента списка. Такая организация списка позволяет перемещаться по его элементам в двух направлениях.

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.**

1.   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;

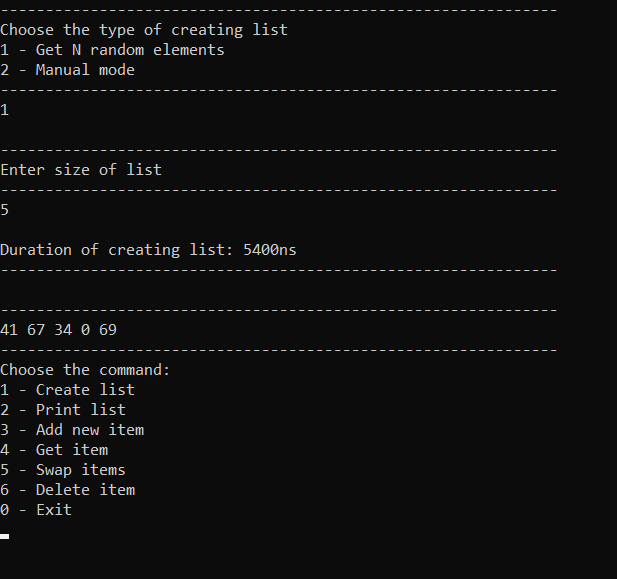
2.   Определение скорости создания двусвязного списка п. 2.

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

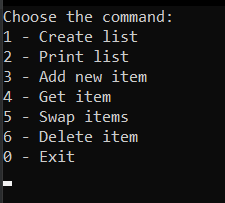
4.   Определение скорости вставки, удаление и получения элемента двусвязного списка п. 3.

**Выполнение работы.**

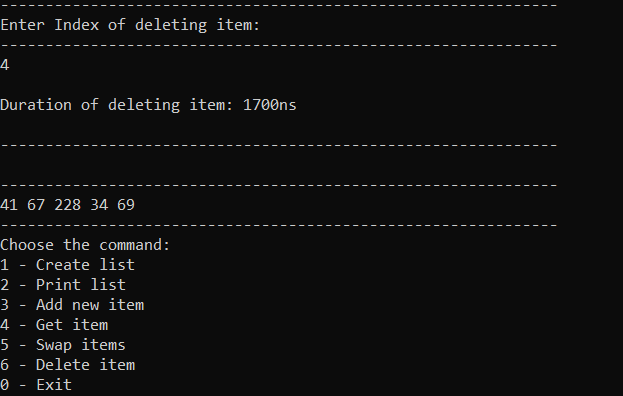
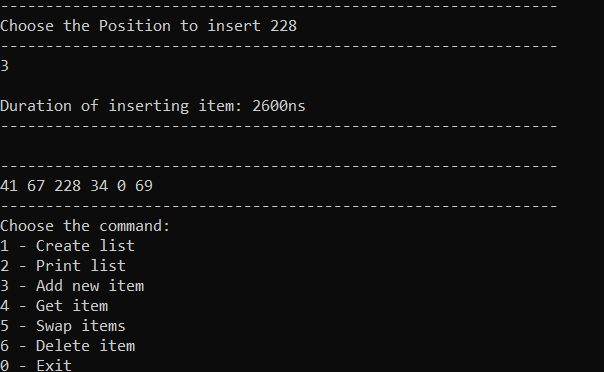
1. Формирование двусвязного списка размерности *N* и определения создания:

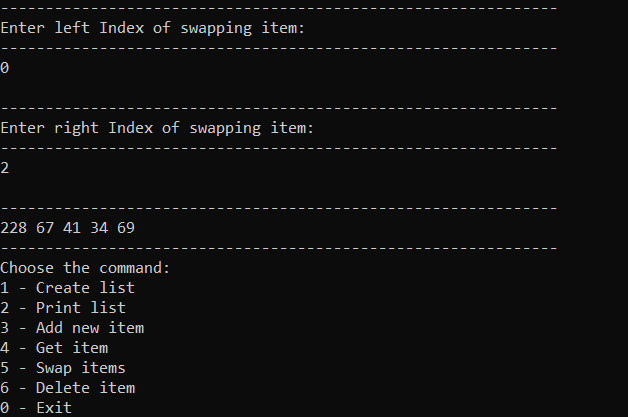


2.Команды, которые можно выполнять со списком и динамическим массивом:

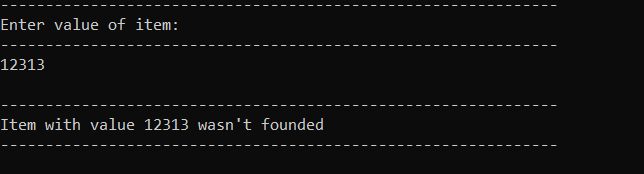


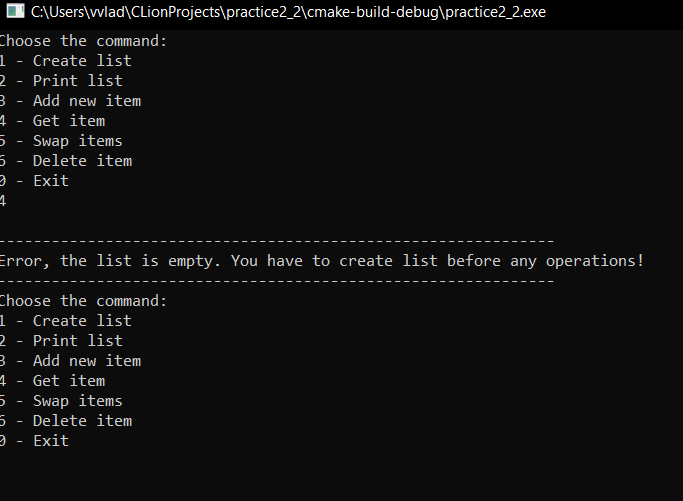
3. Примеры выполнения команд:





4. Проверка на ошибки





**Полный код программы:**

#include <iostream>  
#include <windows.h>  
#include <chrono>  
  
  
  
  
using namespace std;  
struct list  
{  
 int data;  
 list \*head;  
 list \*tail;  
};  
list \*CreateList()  
{  
 int userChoice;  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Choose the type of creating list\n1 - Get N random elements\n2 - Manual mode" ;  
 cout << "\n--------------------------------------------------------------\n";  
 while (!(cin >> userChoice)) {  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Invalid input";  
 cout << "\n--------------------------------------------------------------\n";  
 }  
 if(userChoice == 1) {  
 int Length;  
// srand(time(NULL));  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Enter size of list" ;  
 cout << "\n--------------------------------------------------------------\n";  
 while (!(cin >> Length)) {  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Invalid input";  
 cout << "\n--------------------------------------------------------------\n";  
 }  
 auto start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 list \*Curr = 0,  
 \*Next = 0;  
 for (unsigned i = 1; i <= Length; ++i) {  
 Curr = new list;  
 Curr->head = Next;  
 if (Next) {  
 Next->tail = Curr;  
 }  
 Next = Curr;  
 }  
 Curr->tail = 0;  
 list \* beg = Curr;  
 while ( beg )  
 {  
 beg->data =rand()%100;  
 beg = beg->head;  
 }  
 auto end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 cout << "\nDuration of creating list: " << end - start << "ns";  
 cout << "\n--------------------------------------------------------------\n";  
 return Curr;  
  
 }  
 if (userChoice==2){  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Enter value:(To stop manual mode enter 0)" ;  
 cout << "\n--------------------------------------------------------------\n";  
 int userValue = 1, Length;  
 list \*Curr = 0,  
 \*Tail = 0;  
 auto start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 while (userValue!=0) {  
 cin >> userValue;  
 if (userValue) {  
 Length += 1;  
 Curr = new list;  
 Curr->data = userValue;  
 Curr->tail = Tail;  
 if (Tail) {  
 Tail->head = Curr;  
 }  
 Tail = Curr;  
 }  
 }  
 auto end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 cout << "\nDuration of creating list: " << end - start << "ns\n";  
 cout << "\n--------------------------------------------------------------\n";  
 if(Length){  
 Curr->head = 0;  
 while(Curr->tail){  
 Curr = Curr->tail;  
 }  
  
 return Curr;  
 }  
 else{  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "List is empty!\n";  
 cout << "\n--------------------------------------------------------------\n";  
 return 0;  
 }  
 }  
  
}  
void PrintList(list \*&Beg){  
 list \* Curr = Beg;  
 cout << "\n--------------------------------------------------------------\n";  
 while(Curr){  
 cout << Curr->data << " ";  
 Curr = Curr->head;  
 }  
 cout << "\n--------------------------------------------------------------\n";  
}  
int GetListLength(list \*&Beg){  
 list \*Curr = Beg;  
 int i = 0;  
 while(Curr){  
 ++i;  
 Curr= Curr->head;  
 }  
 return i;  
}  
void DeleteList ( list \* &Beg ){  
 list \*Next;  
 while ( Beg )  
 {  
  
 Next = Beg->head;  
 delete Beg;  
 Beg = Next;  
 }  
}  
list \*getAdr(list \* &beg,unsigned index){  
 while (beg && (index--)) {  
 beg = beg->head;  
 }  
 return beg;  
}  
  
void DelItem( list \* &Beg )  
{  
 int Index;  
 list \*Curr = Beg;  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Enter Index of deleting item:";  
 cout << "\n--------------------------------------------------------------\n";  
 while(!(cin >> Index)){  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Invalid input";  
 cout << "\n--------------------------------------------------------------\n";  
 }  
  
 auto start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
  
 list \* Item;  
 if (!Index)  
 {  
 Item = Beg->head;  
 delete Beg;  
 Beg = Item;  
 Beg->tail = 0;  
 auto end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 cout << "\nDuration of deleting item: " << end - start << "ns";  
 cout << "\n--------------------------------------------------------------\n";  
 return;  
 }  
 Item = getAdr( Curr, Index-1);  
 Curr = Beg;  
 list \* DItem = Item->head;  
 if(DItem->head) {  
 Item->head = DItem->head;  
 Item->head->tail = Item;  
 delete DItem;  
 }  
 else{  
 Item->head = 0;  
 delete DItem;  
 }  
 auto end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 cout << "\nDuration of deleting item: " << end - start << "ns\n";  
 cout << "\n--------------------------------------------------------------\n";  
}  
  
void InsertItem(list\* &Beg){  
 list\*Curr = Beg;  
 int userValue,userIndex;  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Enter value to insert:";  
 cout << "\n--------------------------------------------------------------\n";  
 while(!(cin>>userValue)){  
 cout << "Invalid value\n";  
 }  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Choose the Position to insert " << userValue;  
 cout << "\n--------------------------------------------------------------\n";  
 while(!(cin>>userIndex)){  
 cout << "Invalid value\n";  
 }  
 auto start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 list\*Item = new list;  
 Item->data = userValue;  
 Curr = Beg;  
 if(userIndex==1){  
 Beg->tail = Item;  
 Item->tail = 0;  
 Item->head = Beg;  
 Beg = Item;  
 auto end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 cout << "\nDuration of inserting item: " << end - start << "ns\n";  
 cout << "--------------------------------------------------------------\n";  
 return;  
 }  
 if(userIndex == GetListLength(Curr)){  
 Item->head = 0;  
 list\* lItem = getAdr(Curr,userIndex-1);  
 Item->tail = lItem;  
 lItem->head = Item;  
 auto end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 cout << "\nDuration of inserting item: " << end - start << "ns\n";  
 cout << "--------------------------------------------------------------\n";  
 return;  
 }  
 list\* lItem = getAdr(Curr,userIndex-2);  
 Item->tail = lItem;  
 Item->head = lItem->head;  
 Item->head->tail = Item;  
 lItem->head = Item;  
 auto end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 cout << "\nDuration of inserting item: " << end - start << "ns\n";  
 cout << "--------------------------------------------------------------\n";  
 return;  
  
}  
void GetItem(list\* &Beg){  
 int userChoice;  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "1 - Get item by index\n2 - Get item by value";  
 cout << "\n--------------------------------------------------------------\n";  
 while(!(cin>>userChoice)){  
 cout << "Invalid input\n";  
 }  
 if(userChoice==1) {  
 auto start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 int userIndex;  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Enter Index of item:";  
 cout << "\n--------------------------------------------------------------\n";  
 while (!(cin >> userIndex)) {  
 cout << "Invalid input\n";  
 }  
 list \*Curr = Beg;  
 cout << getAdr(Curr, userIndex)->data << endl;  
 auto end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 cout << "\nDuration of searching item: " << end - start << "ns\n";  
 cout << "--------------------------------------------------------------\n";  
  
 }  
 if(userChoice==2){  
 auto start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 cout << "\n--------------------------------------------------------------\n";  
 cout <<"Enter value of item:";  
 cout << "\n--------------------------------------------------------------\n";  
 int userValue;  
 while(!(cin>>userValue)){  
 cout << "Invalid input\n";  
 }  
 list \*Curr = Beg;  
 bool flag = false;  
 while(Curr){  
 if(Curr->data==userValue){  
 cout << Curr->data << " ";  
 flag = true;  
 }  
 Curr = Curr->head;  
 }  
 if(!flag){  
 cout << "\n--------------------------------------------------------------\n";  
 cout <<"Item with value " << userValue <<" wasn't founded";  
 cout << "\n--------------------------------------------------------------\n";  
 }  
 cout << endl;  
 auto end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 cout << "\nDuration of searching item: " << end - start << "ns\n";  
 cout << "--------------------------------------------------------------\n";  
 }  
  
}  
void swapItem(list \* &beg){  
 int IndexL, IndexR;  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Enter left Index of swapping item:";  
 cout << "\n--------------------------------------------------------------\n";  
 while(!(cin >> IndexL)){  
 cout << "\nInvalid input\n";  
 }  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Enter right Index of swapping item:";  
 cout << "\n--------------------------------------------------------------\n";  
 while(!(cin >> IndexR)){  
 cout << "\nInvalid input\n";  
 }  
 list \* st = beg;  
  
 list \* Litem = getAdr(st,IndexL);  
 st = beg;  
 list \* Ritem = getAdr(st,IndexR);  
 st = beg;  
 if(Litem->tail){  
 Litem->tail->head = Ritem;  
 }else{  
 beg = Ritem;  
 }  
 if((IndexR-IndexL!=1)){  
 Litem->head->tail = Ritem;  
 Ritem->tail->head = Litem;  
 }  
 if(Ritem->head){  
 Ritem->head->tail = Litem;  
 }  
  
  
 if((IndexR-IndexL==1)){  
 Ritem->tail = Litem->tail;  
 Litem->tail = Ritem;  
 Litem->head = Ritem->head;  
 Ritem->head = Litem;  
 return;  
 }  
  
 list \* buf = Litem->tail;  
 Litem->tail = Ritem->tail;  
 Ritem->tail = buf;  
 buf = Litem->head;  
 Litem->head = Ritem->head;  
 Ritem->head = buf;  
  
}  
int\* CreateArray(){  
 int arrSize;  
 cout << "\n----------------------------------";  
 cout << "\nEnter size of array for comparsion:";  
 cout << "\n----------------------------------\n";  
 while(!(cin>>arrSize)){  
 cout << "\nInvalid Input\n";  
 }  
 int \*arr = (int \*)malloc(arrSize \* sizeof(int));  
 for (int \*i = arr; i < arr + arrSize;i++){  
 \*i = rand()%100;  
 }  
 return arr;  
}  
void PrintArray(int\* arr,int arrSize){  
 for (int \*i = arr; i < arr + arrSize;i++){  
 cout << \*i << " ";  
 }  
}  
  
void ComparsionIdz(list \* &Beg, int\*arr,int size){  
 int userValue;  
 cout << "\n----------------------------------";  
 cout << "\nEnter value for any even-numbered item:";  
 cout << "\n----------------------------------\n";  
 while(!(cin >> userValue)){  
 cout << "\nInvalid input\n";  
 }  
 auto start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 for (int \*i = arr,g=0; i < arr + size;i++,g++){  
 if (g%2==0){  
 \*i -= userValue;  
 }  
 else{  
 \*i -= rand()%6;  
 }  
 }  
 auto end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Duration of operation in dynamic array: " << end - start << "ns\n";  
 cout << "--------------------------------------------------------------\n";  
 PrintArray(arr,size);  
 list\*Curr = Beg;  
 int i = 0;  
 start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 while(Curr){  
 if(i%2==0){  
 Curr->data -= userValue;  
 }  
 else{  
 Curr->data -= rand()%6;  
 }  
 Curr = Curr->head;  
 i++;  
 }  
  
 Curr = Beg;  
 end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Duration of operation in list: " << end - start << "ns\n";  
 cout << "--------------------------------------------------------------\n";  
 PrintList(Curr);  
 delete [] arr;  
 DeleteList(Beg);  
  
}  
  
int main() {  
 int cycleMain;  
// bool isListExist = false;  
// List = CreateList();  
  
 int userChoiceIdz;  
 while(true) {  
 cout << "\n-------------------------------------\n";  
 cout << "0 - Start Individual home task #9\n1 - Start main program";  
 cout << "\n-------------------------------------\n";  
 while (!(cin >> userChoiceIdz)) {  
 cout << "\nInvalid Input\n";  
 }  
  
 if (!userChoiceIdz) {  
 int \*arr = CreateArray();  
 list \*List = 0;  
 List = CreateList();  
 list \*Curr = List;  
 list \*Beg = List;  
 PrintArray(arr, GetListLength(Beg));  
 Beg = List;  
 PrintList(Beg);  
 ComparsionIdz(Curr, arr, GetListLength(Beg));  
 Beg = List;  
  
// Sleep(10000);  
 } else {  
 list \*List = 0;  
 list \*Curr = List;  
 getchar();  
 system("cls");  
 while (true) {  
 cout << "Choose the command:\n1 - Create list\n2 - Print list\n3 - Add new item\n4 - Get item\n5 - Swap items\n6 - Delete item\n0 - Exit\n";  
 cin >> cycleMain;  
 if (!cycleMain) {  
 DeleteList(List);  
 return 0;  
 }  
 if (cycleMain == 1) {  
 List = CreateList();  
// isListExist = true;  
 Curr = List;  
 PrintList(Curr);  
 } else if (List) {  
 switch (cycleMain) {  
 case 2:  
 PrintList(Curr);  
 break;  
  
 case 3:  
 InsertItem(Curr);  
 PrintList(Curr);  
 break;  
 case 4:  
 GetItem(Curr);  
 break;  
 case 5:  
 swapItem(Curr);  
 PrintList(Curr);  
 break;  
  
 case 6:  
 DelItem(Curr);  
 PrintList(Curr);  
 break;  
 default:  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Error, unknown command";  
 cout << "\n--------------------------------------------------------------\n";  
 break;  
 }  
 } else {  
 cout << "\n--------------------------------------------------------------\n";  
 cout << "Error, the list is empty. You have to create list before any operations!";  
 cout << "\n--------------------------------------------------------------\n";  
 }  
  
  
 }  
 }  
 }  
  
}

**Вывод:**

Проведя тесты на большом количестве элементов, я выяснил, что списки работают быстрее при удалении и вставки элемента, а динамический массив лучше справляется с нахождением элемента по индексу и обмену элементов. Это обуславливается разной работай с памятью каждого из типов хранения данных. Поэтому при работе с данными, где постоянно необходимо удалять и добавлять элементы лучше подойдут списки, например, очередь или стек. Динамические массивы лучше подойдут статическим данным, где необходимо в основном брать элементы из массива, например, статическая база данных.