**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: Списки.

|  |  |  |
| --- | --- | --- |
| Студентка гр. |  | Котлова П. В. |
| 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.

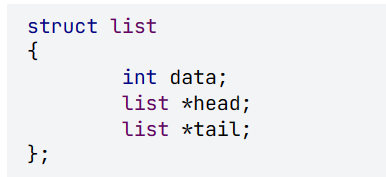
**Основные теоретические положения.**

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

Одним из недостатков односвязных элементов является то, что узел имеет указатель только на следующий элемент. Вернуться из текущего элемента к предыдущему невозможно.

Каждый узел 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:



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.

**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.

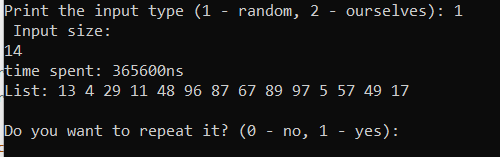
3. Вставка, удаление, обмена и получение элемента двусвязного списка. Удаление и получение элемента необходимо реализовать по индексу и по значению.

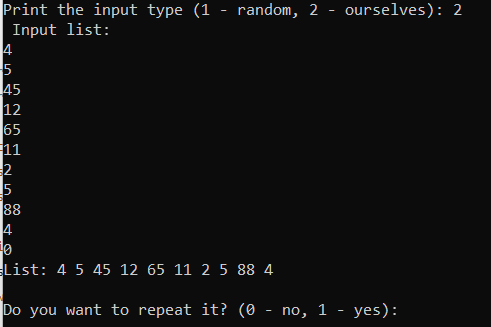
4. Определение скорости вставки, удаление и получения элемента двусвязного списка п. 3.

**Выполнение работы.**

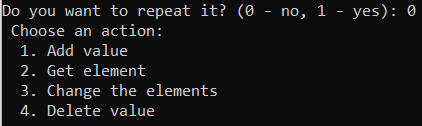
Код программы представлен в приложении А.

1. Формирование двусвязного списка размерности N и определения создания:

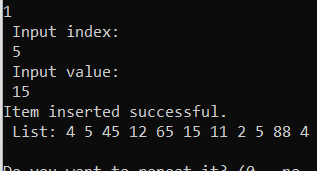


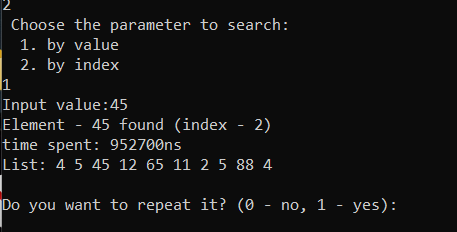


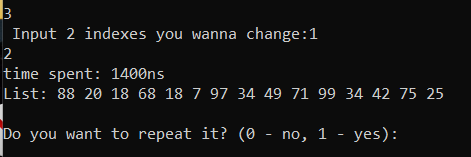
2.Команды:

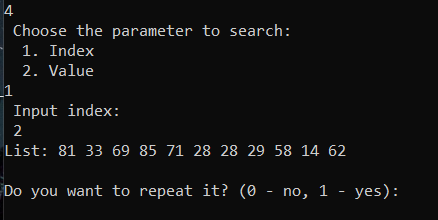


3.Примеры выполнения команд:

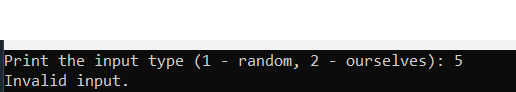








4. Проверка на ошибки



**Выводы.**

В ходе работы были изучены свойства и организация двусвязных линейных списков. Получены практические навыки работы с динамическими массивами и двусвязными списками. Проведена сравнительная характеристика скорости вставки, получения и удаления элементов двухсвязного списка.

Приложение А

рабочий код

#include <iostream>

#include <sstream>

#include <chrono>

#include <limits>

using namespace std;

struct Node {

int data;

Node\* next;

Node\* pred;

};

int checking() {

int a;

while (true)

{

cin >> a;

if (std::cin.fail())

{

cout << "Invalid input\n";

cin.clear();

cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

continue;

}

break;

}

return a;

}

void printList(Node\* head) {

cout << "List: ";

while (head) {

cout << head->data << " ";

head = head->next;

}

}

void deleteList(Node\*& list) {

Node\* Next;

while (list)

{

Next = list->next;

delete list;

list = Next;

}

}

Node\* createList(unsigned size) {

auto start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();

Node\* head = nullptr,

\* tail = nullptr;

srand(time(nullptr));

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

head = new Node;

head->data = rand() % 100;

head->next = tail;

if (tail) {

tail->pred = head;

}

tail = head;

}

if (size != 0) head->pred = nullptr;

auto end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();

cout << "time spent: " << end - start << "ns\n";

return head;

}

Node\* createListFromInput() {

Node\* head = nullptr;

Node\* tail = nullptr;

int input;

while (true) {

input = checking();

if (input == 0) return head;

auto\* curr= new struct Node;

curr->data = input;

curr->next = nullptr;

if (tail) {

tail->next = curr;

curr->pred = tail;

}

else {

curr->pred = nullptr;

head = curr;

}

tail = curr;

}

}

Node\* getItemByIndex( Node\* head, int index) {

if (!head) {

cout << "List is empty\n";

return nullptr;

}

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

if (!head->next) {

cout << "Index out of range\n";

return nullptr;

}

head = head->next;

}

return head;

}

Node\* listofValue(Node\* Head, unsigned int index) {

int id = 0;

while (id != index) {

Head = Head->next;

id++;

}

return Head;

}

Node\* getItemByValue(struct Node\* head, int value) {

if (!head) {

cout << "List is empty\n";

return nullptr;

}

while (head) {

if (head->data == value) break;

if (!head->next) {

cout << "Item not found\n";

return nullptr;

}

head = head->next;

}

return head;

}

bool insertItem(Node\*& head) {

int index, value;

cout << " Input index:\n ";

index = checking();

cout << " Input value:\n ";

value = checking();

struct Node\* point = getItemByIndex(head, index);

if (!point) return false;

auto\* newNode = new struct Node;

newNode->data = value;

newNode->pred = point->pred;

newNode->next = point;

if (newNode->pred) newNode->pred->next = newNode;

if (newNode->next) newNode->next->pred = newNode;

if (index == 0) head = newNode;

return true;

}

void inputList1(Node\*& list, int& length) {

cout << "Print the input type (1 - random, 2 - ourselves): ";

int input = checking();

int size;

string stringList;

if (input == 1) {

cout << " Input size: \n";

size = checking();

if (size <= 0) {

cout << "Invalid input.\n";

inputList1(list, length);

}

length = size;

list = createList(length);

}

else if (input == 2) {

cout << " Input list: \n";

list = createListFromInput();

}

else {

cout << "Invalid input.\n";

inputList1(list,length);

}

}

void deleteItem(struct Node\*& head) {

int index;

cout << " Input index:\n ";

index = checking();

if (!head) return;

struct Node\* found = getItemByIndex(head, index - 1);

if (!found) return;

found->pred ? found->pred->next = found->next : head = found->next;

found->next ? found->next->pred = found->pred : nullptr;

delete found;

}

void deleteItemByValue(struct Node\*& head) {

int value;

cout << " Input value:\n ";

value = checking();

if (!head) return;

struct Node\* point = getItemByValue(head, value);

if (!point) return;

point->pred ? point->pred->next = point->next : head = point->next;

point->next ? point->next->pred = point->pred : nullptr;

delete point;

}

pair<Node\*, Node\*> find(Node\*& head, int index1, int index2)

{

Node\* curr1 = nullptr;

Node\* curr2 = nullptr;

Node\* temp = head;

while (temp != nullptr) {

if (temp->data == index1)

curr1 = temp;

else if (temp->data == index2)

curr2 = temp;

temp = temp->next;

}

return make\_pair(curr1, curr2);

}

void swapByIndex(Node\*& list, int length) {

int index1, index2;

cout << " Input 2 indexes you wanna change:";

index1 = checking();

index2 = checking();

if (!((index1 >= 1 && index1 <= length) && (index2 >= 1 && index2 <= length))) {

cout << "Wrong input!\n";

return;

}

if (index1 == index2) return;

index1--; index2--;

if (index1 > index2) {

swap(index1, index2);

}

auto start = chrono::duration\_cast<chrono::nanoseconds>(chrono::system\_clock::now().time\_since\_epoch()).count();

Node\* curr1 = listofValue(list, index1);

Node\* curr2 = listofValue(list, index2);

if (curr1->pred != nullptr) {

curr1->pred->next = curr2;

}

else {

list = curr2;

}

if (curr1->next != curr2) {

curr1->next->pred = curr2;

}

if (curr2->pred != curr1) {

curr2->pred->next = curr1;

}

if (curr2->next != nullptr) {

curr2->next->pred = curr1;

}

if (curr1->next == curr2) {

curr2->pred = curr1->pred;

curr1->pred = curr2;

curr1->next = curr2->next;

curr2->next = curr1;

auto end = chrono::duration\_cast<chrono::nanoseconds>(chrono::system\_clock::now().time\_since\_epoch()).count();

cout << "time spent: " << end - start << "ns\n";

return;

}

Node\* nope = curr1->pred;

curr1->pred = curr2->pred;

curr2->pred = nope;

nope = curr1->next;

curr1->next = curr2->next;

curr2->next = nope;

auto end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();

cout << "time spent: " << end - start << "ns\n";

}

void getValue(Node\*& list, int length) {

cout << " Choose the parameter to search:\n" <<

" 1. by value\n"

" 2. by index\n";

int type, value, index, cnt = 0;

bool flag = false;

Node\* head = list;

type = checking();

auto start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();

auto end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();

switch (type) {

case 1:

cout << "Input value:";

value = checking();

start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();

while (head) {

if (head->data == value) {

flag = true;

index = cnt;

break;

}

head = head->next;

cnt++;

}

if (flag) {

cout << "Element - " << value << " found (index - " << index << ")" << "\n";

}

else {

cout << "Element - " << value << "does not exist in the list."<< "\n";

}

end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();

cout << "time spent: " << end - start << "ns\n";

break;

case 2:

cout << " Input index:";

index = checking();

while (!(index >= 1 && index <= length)) {

cout << "Wrong input!\n";

index = checking();

}

start = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();

head = listofValue(list, index - 1);

cout << "Value with index " << index << " - " << head->data << "\n";

end = std::chrono::duration\_cast<std::chrono::nanoseconds>(std::chrono::system\_clock::now().time\_since\_epoch()).count();

cout << "time spent: " << end - start << "ns\n";

break;

default:

cout << "Wrong input!\n";

break;

}

}

void inputList2(Node\*& list, int& length) {

cout << " Choose an action:\n";

int inputType = 0;

if (length > 0) {

cout << " 1. Add value\n" <<

" 2. Get element\n" <<

" 3. Change the elements\n" <<

" 4. Delete value\n";

inputType = checking();

}

else {

cout <<

" 1. Add value\n" <<

" 2. Get element\n" <<

" 3. Change the elements\n";

inputType = checking();

}

switch (inputType) {

case 1: {

if (!insertItem(list))

break;

cout << "Item inserted successful. \n ";

break;

}

case 2: {

getValue(list, length);

break;

}

case 3:{

swapByIndex(list, length);

break;

}

case 4: {

cout << " Choose the parameter to search:\n";

cout << " 1. Index\n";

cout << " 2. Value\n";

int input;

input = checking();

if (input != 2 && input != 1) {

cout << "Wrong input!\n";

return;

}

if (input == 1) {

deleteItem(list);

}

else {

deleteItemByValue(list);

}

break;

}

}

}

int main() {

Node\* list = nullptr;

int enter = 1, length = 0;

while (enter) {

inputList1(list, length);

while (list == nullptr) {

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

inputList1(list, length);

}

printList(list);

cout << "\n";

cout << "\nDo you want to repeat it? (0 - no, 1 - yes): ";

enter = checking();

}

enter = 1;

while (enter) {

while (list == nullptr) {

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

inputList1(list, length);

}

inputList2(list, length);

printList(list);

cout << "\n";

cout << "\nDo you want to repeat it? (0 - no, 1 - yes): ";

enter = checking();

}

deleteList(list);

return 0;

}