|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | МИНОБРНАУКИ РОССИИ | | | | Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |   Институт Информационных технологий | |
|  | |
| Кафедра Математического обеспечения и стандартизации информационных технологий | |
|  | |
|  | |

|  |  |
| --- | --- |
| **ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 5** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  **Тема: «Однонаправленный динамический список»** | |
|  | |
| Выполнил студент группы ИКБО-72-23 | Шатохин Б.А. |
| Принял преподаватель | Филатов А.С. |

|  |  |  |
| --- | --- | --- |
| Лабораторная работа выполнена | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись студента)* |
|  |  |  |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись руководителя)* |

Москва 2024

1. Цель работы

Получить знания и практические навыки управления динамическим однонаправленным списком.

1. Постановка задачи

Реализуйте программу решения задачи варианта по использованию линейного однонаправленного списка.

1. Разработать функцию для создания исходного списка, его вывода и добавления узла.
2. Информационная часть узла списка определена вариантом.
3. Разработать функции дополнительного задания варианта.
4. В основной программе выполнить тестирование каждой функции, описанной в задании. Программа должна позволять пользователю непрерывно выполнять операции над списком в произвольном порядке.
5. Составить отчет по выполненному заданию. В отчет включить ответы на вопросы к практической работе.

Индивидуальное задание:

Тип узла – int;

Задача - Даны два однонаправленных списка L1 и L2.

Создать новый списка из тех значений, которые входят и в список L1 и L2.

Модифицировать список L2: вставить список L1 в список L2 после узла с заданным значением.

Удалить из списка L1 (полученного в результате модификации), все узлы с четными значениями.

1. Решение

Если хотите понять, как построен односвязный список, представьте цепь. У неё есть и начало, и конец, плюс звенья последовательно соединены друг с другом. Можно легко пройти от начала до конца цепи, перебирая звенья.

Схожим образом работают односвязные списки. В них начало списка — это головной элемент, звенья — это узлы, а конец списка определяется посредством NULL — специального узла. Причём, чтобы структура списка была полезной, на каждый узел «вешают» определённое значение.

У односвязного списка есть преимущество — вставка и удаление узлов осуществляется довольно легко в любом месте списка. Однако список нельзя индексировать в качестве массива, а структура списка ограничивает доступ к узлам по индексу. Если нужно попасть на какой-нибудь узел односвязного списка, нужно пройти весь путь последовательно, начиная от головного элемента, заканчивая нужным узлом.

Для создания списка была создана структура node.

struct node { // Структура для создания списка

int value;

node\* p\_next;

node(int x) {

value = x;

p\_next = nullptr;

}

};

Для добавления элемента в список была реализованная функция ListAppend, которая принимает в качестве аргумента головной элемент списка, в который мы хотим добавить элемент, а так же сам элемент в виде целочисленного значения.

void ListAppend(node\*\* head, int x) { // Функция добавления в наш список

node\* new\_elem = new node(x); // Инициализация нового элемента, которого мы хотим добавить

// Проверку на пустотту списка

if (\*head == nullptr) { // Если список пуст

\*head = new\_elem;

}

else { // Если что-то есть

node\* p\_last = \*head;

while (p\_last->p\_next != nullptr) { // Перебор значений нашего списка

p\_last = p\_last->p\_next;

} // Дошли до конца

p\_last->p\_next = new\_elem; // Добавили элемент в конец

}

}

Для вывода элементов списка была реализована функция ListPrint, которая принимает в качестве аргумента головной элемент списка, которого мы хотим вывести.

void ListPrint(node\* head) {

node\* p\_last = head;

while (p\_last != nullptr) { // Перебор значений нашего списка

std::cout << p\_last->value << " -> ";

p\_last = p\_last->p\_next;

} // Дошли до конца

std::cout << "nullptr\n";

}

Для нахождения пересечения двух списков была разработана функция ListIntersection, которая принимает в качестве аргументов два списка.

node\* ListIntersection(node\* L1, node\* L2) { // Поиск пересечения L1 и L2

node\* L3 = nullptr; // Корень списка

node\*\* next = &L3; // Последующий эл

for (node\* l1 = L1; l1 != nullptr; l1 = l1->p\_next) { // Пробегаемся по всем значениям L1

for (node\* l2 = L2; l2 != nullptr; l2 = l2->p\_next) { // Пробегаемся по всем значениям L2

if (l1->value == l2->value) {

\*next = new node(l1->value);

next = &(\*next)->p\_next;

break;

}

}

}

return L3; // Возврат адреса на начало списка с пересечением двух списков L1 и L2

}

Для вставки одного списка в другой была разработан функция ListInsert, которая принимает в качестве аргументов два списка и элемент после которого необходимо вставить список.

void ListInsert(node\*\* L2, node\* L1, int anchor) { // Вставка одного списка в дргугой

node\* inserted\_list = \*L2;

while (inserted\_list != nullptr && inserted\_list->value != anchor) {

inserted\_list = inserted\_list->p\_next;

}

if (inserted\_list != nullptr) {

node\* l1 = nullptr;

node\*\* next = &l1;

for (node\* n = L1; n != nullptr; n = n->p\_next) {

\*next = new node(n->value);

next = &(\*next)->p\_next;

}

\*next = inserted\_list->p\_next;

inserted\_list->p\_next = l1;

}

}

Для удаления из списка всех чётных элементов была разработана функция ListRemoveEvenNumbers, которая принимает в качестве аргумента список, из которого нужно удалить эти элементы.

void ListRemoveEvenNumbers(node\*\* L) { // Удаление чётных чисел из списка

node\* current = \*L;

node\* p\_prev = nullptr;

while (current != nullptr) {

if (current->value % 2 == 0) {

if (p\_prev == nullptr) {

\*L = current->p\_next;

}

else {

p\_prev->p\_next = current->p\_next;

}

node\* del = current;

current = current->p\_next;

delete del;

}

else {

p\_prev = current;

current = current->p\_next;

}

}

}

1. Тестирование

В исходном коде программы мы создаём два списка L1 и L2. L1 заполнен 6 числами от 1 до 6, а L2 заполнен числами 1, 2, 2, 3, 3, 4, 4, 5. Так же выполнение индивидуального варианта осуществляется автоматически, L3 – список пересечений L1 и L2.

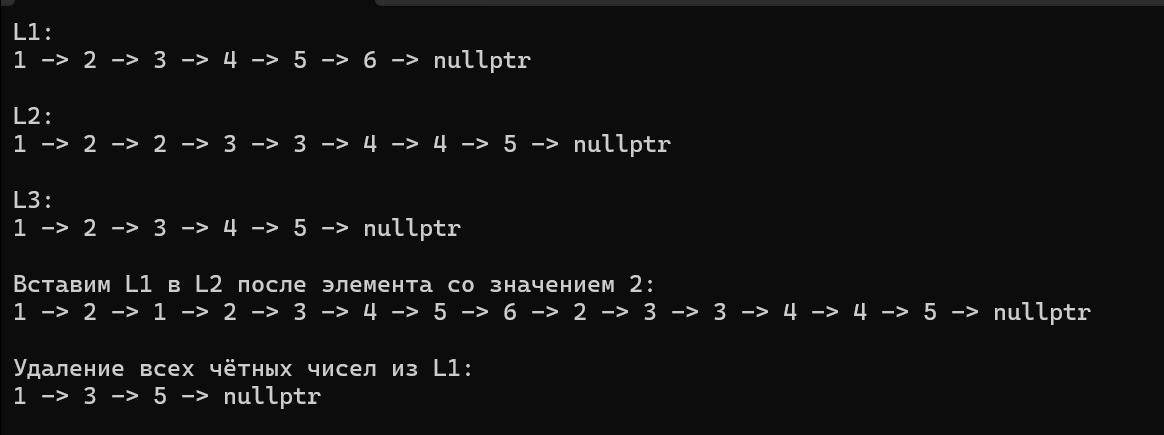


Рисунок 1. Результат тестирования программы

1. Вывод

Получены знания и практические навыки управления динамическим однонаправленным списком.

1. Исходный код

int main() {

setlocale(LC\_ALL, "ru");

node\* L1 = nullptr;

node\* L2 = nullptr;

// Инициализация L1

ListAppend(&L1, 1);

ListAppend(&L1, 2);

ListAppend(&L1, 3);

ListAppend(&L1, 4);

ListAppend(&L1, 5);

ListAppend(&L1, 6);

// инициализация L2

ListAppend(&L2, 1);

ListAppend(&L2, 2);

ListAppend(&L2, 2);

ListAppend(&L2, 3);

ListAppend(&L2, 3);

ListAppend(&L2, 4);

ListAppend(&L2, 4);

ListAppend(&L2, 5);

std::cout << "L1:\n";

ListPrint(L1);

std::cout << "\nL2:\n";

ListPrint(L2);

node\* L3 = ListIntersection(L1, L2);

std::cout << "\nL3:\n";

ListPrint(L3);

ListInsert(&L2, L1, 2);

std::cout << "\nВставим L1 в L2 после элемента со значением 2:\n";

ListPrint(L2);

ListRemoveEvenNumbers(&L1);

std::cout << "\nУдаление всех чётных чисел из L1:\n";

ListPrint(L1);

return 0;

}

std::cout << "\nL2:\n";

ListPrint(L2);

node\* L3 = ListIntersection(L1, L2);

std::cout << "\nL3:\n";

ListPrint(L3);

ListInsert(&L2, L1, 2);

std::cout << "\nВставим L1 в L2 после элемента со значением 2:\n";

ListPrint(L2);

ListRemoveEvenNumbers(&L1);

std::cout << "\nУдаление всех чётных чисел из L1:\n";

ListPrint(L1);

return 0;

}