ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

КУРСОВАЯ РАБОТА

Структура данных «Список»
по дисциплине «Объектно-ориентированное программирование»

Выполнил
Студент
гр. 3331506/90401 _____ Мамиев М.А.
(подпись)

Работу принял _____ Ананьевский М.С.
(подпись)

Санкт-Петербург
2022 г.

1. Введение

Структура данных – это форма хранения и представления информации.

По организации взаимосвязей между элементами сложных структур данных существует следующая классификация:

1. Линейные
    a. Массив
    b. Список
    c. Стек
    d. Очередь
    e. Хэш-таблица
2. Иерархические
    a. Двоичные деревья
    b. N-арные деревья
    c. Иерархический список
3. Сетевые
    a. Простой граф
    b. Ориентированный граф
4. Табличные
    a. Таблица реляционной базы данных
    b. Двумерный массив
5. Другие

Список (связный список от англ. *List*) – это динамическая линейная структура данных, в которой каждый элемент ссылается либо только на предыдущий – однонаправленный линейный список, либо на предыдущий и следующий за ним – двунаправленный линейный список.

Достоинство этой структуры данных, помимо возможности изменять размер, - это простота реализации. Также, благодаря наличию ссылок, каждый элемент в списке, в отличие от массива, может занимать разный объем памяти. Адрес первого элемента в линейном списке однозначно определяется адресом самого списка.

На рисунке 1 представлен однонаправленный линейный список.

Рисунок 1. Однонаправленный линейный список

На рисунке 2 представлен двунаправленный линейный список.

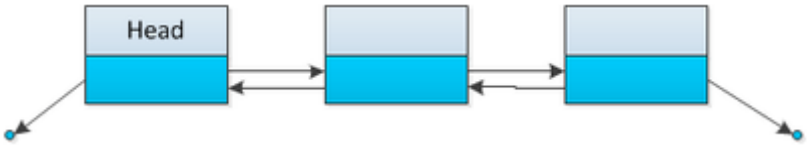

Рисунок 2. Двунаправленный линейный список.

2. Операции на списке

Рассмотрим базовые операции на примере **односвязного** списка.

1) Вставка в начало списка (*push_front*)

Установим в этом новом элементе ссылку на старую голову, и обновим указатель на голову.

```
Node* new_head = new Node(data, head);
head = new_head;
```

Производительность O(1).

2) Вставка в конец списка (*push_back*)

Перебираем все элементы списка, пока не доберемся до последнего. Обновляем ссылку на следующий элемент последнего элемента так, чтобы теперь он указывал на новый элемент.

```
Node* temp_head = head;
while (temp_head->next != nullptr) {
        temp_head = temp_head->next;
}
temp_head->next = new Node(data, nullptr);
```

Производительность O(N).

3) Вставка элемента по индексу (*insert*)

Будем перебирать элементы списка пока не дойдем до элемента с индексом *index – 1*. Установим в новом элементе ссылку на элемент с индексом *index*. Установим в элементе с индексом *index – 1* ссылку на новый элемент.

```
Node* previous = head;
for (int i = 0; i < index - 1; ++i) {
        previous = previous->next;
}
previous->next = new Node(data, previous->next);
```

Производительность O(N).

4) Удаление элемента с начала списка(*pop_front*)

Для того, чтобы удалить голову списка, переназначим указатель на голову на второй элемент списка, а голову удалим.

```
Node* del_node = head;
head = head->next;
delete del_node;
```

Производительность O(1).

5) Удаление элемента по индексу (*removeAt*)

Будем перебирать элементы списка пока не дойдем до элемента с индексом *index – 1*. Переназначим указатель на следующий элемент элемента *index – 1* на элемент с индексом *index +1*. А элемент с индексом *index* удалим.

```
Node* previous = head;
for (int i = 0; i < index - 1; ++i) {
        previous = previous->next;
}
Node* del_node = previous->next;
previous->next = del_node->next;
delete del_node;
```

Производительность O(N).

6) Удаление элемента с конца списка (*pop_back*)

Чтобы удалить элемент с конца списка, применим метод *removeAt* и

передадим туда параметр *index = size() – 1*.

Производительность O(N).

3. Исследование производительности основных методов для структуры хранения данных «Список»

Терминал с результатами тестов производительности представлен на рисунке 3.

```
Test push_front:
nodes_count: 1000000, milliseconds: 0
nodes_count: 2000000, milliseconds: 0
nodes_count: 3000000, milliseconds: 0
nodes_count: 4000000, milliseconds: 0
nodes_count: 5000000, milliseconds: 0
nodes_count: 6000000, milliseconds: 0
nodes_count: 7000000, milliseconds: 0
nodes_count: 8000000, milliseconds: 0
nodes_count: 9000000, milliseconds: 0
nodes_count: 10000000, milliseconds: 0
Test push_back:
nodes_count: 1000000, milliseconds: 19
nodes_count: 2000000, milliseconds: 36
nodes_count: 3000000, milliseconds: 56
nodes_count: 4000000, milliseconds: 74
nodes_count: 5000000, milliseconds: 96
nodes_count: 6000000, milliseconds: 107
nodes_count: 7000000, milliseconds: 131
nodes_count: 8000000, milliseconds: 157
nodes_count: 9000000, milliseconds: 165
nodes_count: 10000000, milliseconds: 178
Test pop_front:
nodes_count: 1000000, milliseconds: 0
nodes_count: 2000000, milliseconds: 0
nodes_count: 3000000, milliseconds: 0
nodes_count: 4000000, milliseconds: 0
nodes_count: 5000000, milliseconds: 0
nodes_count: 6000000, milliseconds: 0
nodes_count: 7000000, milliseconds: 0
nodes_count: 8000000, milliseconds: 0
nodes_count: 9000000, milliseconds: 0
nodes_count: 10000000, milliseconds: 0
Test pop_back:
nodes_count: 1000000, milliseconds: 16
nodes_count: 2000000, milliseconds: 39
nodes_count: 3000000, milliseconds: 53
nodes_count: 4000000, milliseconds: 69
nodes_count: 5000000, milliseconds: 87
nodes_count: 6000000, milliseconds: 106
nodes_count: 7000000, milliseconds: 128
nodes_count: 8000000, milliseconds: 137
nodes_count: 9000000, milliseconds: 160
nodes_count: 10000000, milliseconds: 183
```

Рисунок 3. Результаты тестов

Из рисунка 3 видно, что методы *push_front*, *pop_front* имеют независимую от количества вершин производительность O(1).

На рисунке 4 представлен совмещенный график производительности методов *push_back* и *pop_back*.

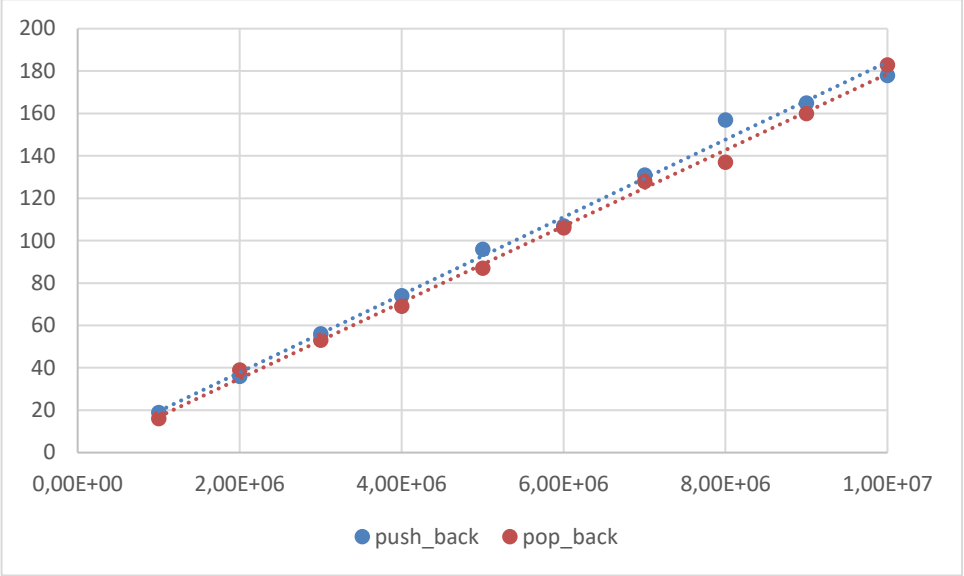

Рисунок 4. Производительность методов *push_back* & *pop_back*

Как видно из рисунка 4 производительность методов *push_back* & *pop_back* линейная - O(N).

4. Заключение

В ходе выполнения работы была разобрана структура данных «Список». Был написан код реализации односвязного списка. Была оценена временная сложность методов для работы с односвязным списком.

Следует сделать вывод, что работа с односвязными списками удобна в случаях, когда мы часто

1) Добавляем элементы
2) Удаляем элементы

Стоит отметить, что если нам нужен постоянный доступ к элементам, то следует использовать другие структуры данных (к примеру, массив), поскольку в случае списка понадобится перебор всех элементов, пока не достигнем нужного.

Список литературы

1. Хайнеман, Д. Алгоритмы. Справочник. С примерами на C, C++, Java иPython /Д. Хайнеман, Г. Поллис, С. Селков. – Вильямс, 2017.
2. Седжвик Роберт. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск: Пер. с англ./Роберт Седжвик.- Издательство «ДиаСофт», 2001.

Приложение 1. Файл *List.h*

```
#ifndef LIST
#define LIST
template <typename T>
class List {
public:
    List();
    ~List();
public:
    bool empty() const;
    void clear();
    void print();
    void pop_front();
    void pop_back();
    void removeAt(const int index);
    void push_front(const T& data);
    void push_back(const T& data);
    void insert(const T& data, int index);
    int size() { return list_size; }
    T& operator [](const int index);
private:
    class Node {
    public:
        T data;
        Node* next;
        Node(T data, Node* next) {
            this->data = data;
            this->next = next;
        }
    };
    Node* head;
    int list_size;
};
```

Приложение 2. Файл *List.cpp*

```cpp
#include <iostream>
#include "List.h"
#include <stdexcept>

#define EXPLICIT_INSTANTIATION(CLASSNAME)    \
    template class CLASSNAME<int8_t>;        \
    template class CLASSNAME<int16_t>;       \
    template class CLASSNAME<int32_t>;       \
    template class CLASSNAME<int64_t>;       \
                                             \
    template class CLASSNAME<float>;         \
    template class CLASSNAME<double>;

EXPLICIT_INSTANTIATION(List);

template <typename T>
List<T>::List()
{
    list_size = 0;
    head = nullptr;
}

template <typename T>
bool List<T>::empty() const {
    return head == nullptr;
}

template <typename T>
void List<T>::clear() {
    while (size()) {
        pop_front();
    }
}

template <typename T>
void List<T>::print() {
    Node* temp_head = head;
    while (temp_head != nullptr) {
        std::cout << temp_head->data << " ";
        temp_head = temp_head->next;
    }
}
```

```cpp
template <typename T>
void List<T>::removeAt(const int index) {
    if (index >= size()) {
        throw std::length_error("going overboard the list");
    }
    if (index == 0) {
        pop_front();
    }
    else {
        Node* previous = head;
        for (int i = 0; i < index - 1; ++i) {
            previous = previous->next;
        }
        Node* del_node = previous->next;
        previous->next = del_node->next;
        delete del_node;
        list_size--;
    }
}

template <typename T>
void List<T>::insert(const T& data, int index) {
    if (index >= size()) {
        throw std::length_error("going overboard the list");
    }
    if (index == 0) {
        push_front(data);
    }
    else {
        Node* previous = head;
        for (int i = 0; i < index - 1; ++i) {
            previous = previous->next;
        }
        previous->next = new Node(data, previous->next);
        list_size++;
    }
}

template <typename T>
void List<T>::push_front(const T& data) {
    Node* new_head = new Node(data, head);
    head = new_head;
    list_size++;
}
```

```cpp
template <typename T>
void List<T>::push_back(const T& data) {
    if (empty()) {
        head = new Node(data, nullptr);
        list_size++;
    }
    else {
        Node* temp_head = head;
        while (temp_head->next != nullptr) {
            temp_head = temp_head->next;
        }
        temp_head->next = new Node(data, nullptr);
        list_size++;
    }
}

template <typename T>
void List<T>::pop_back() {
    removeAt(size() - 1);
}

template <typename T>
void List<T>::pop_front() {
    Node* del_node = head;
    head = head->next;
    delete del_node;
    list_size--;
}

template<typename T>
T& List<T>::operator[](const int index) {
    if (index >= size()) {
        throw std::length_error("going overboard the list");
    }
    int count = 0;
    Node* temp_head = head;
    while (temp_head != nullptr) {
        if (count == index) {
            return temp_head->data;
        }
        count++;
        temp_head = temp_head->next;
    }
}

template <typename T>
List<T>::~List() {
    clear();
}
```

Приложение 3. Файл *main.cpp*

```cpp
#include <iostream>
#include "List.h"
#include <vector>
#include <ctime>

//Тест push_front
void test_push_front(int nodes_count) {
    List<int> my_list;
    for (int i = 0; i < nodes_count; i++) {
        my_list.push_front(rand() % nodes_count);
    }
    int start = clock();
    my_list.push_front(1);
    int end = clock();
    std::cout << "nodes_count: " << nodes_count << ", milliseconds: ";
    std::cout << (end - start) * 1000 / CLOCKS_PER_SEC << std::endl;
}

//Тест push_back
void test_push_back(int nodes_count) {
    List<int> my_list;
    for (int i = 0; i < nodes_count; i++) {
        my_list.push_front(rand() % nodes_count);
    }
    int start = clock();
    my_list.push_back(1);
    int end = clock();
    std::cout << "nodes_count: " << nodes_count << ", milliseconds: ";
    std::cout << (end - start) * 1000 / CLOCKS_PER_SEC << std::endl;
}

//Тест pop_front
void test_pop_front(int nodes_count) {
    List<int> my_list;
    for (int i = 0; i < nodes_count; i++) {
        my_list.push_front(rand() % nodes_count);
    }
    int start = clock();
    my_list.pop_front();
    int end = clock();
    std::cout << "nodes_count: " << nodes_count << ", milliseconds: ";
    std::cout << (end - start) * 1000 / CLOCKS_PER_SEC << std::endl;
}

//Тест pop_back
void test_pop_back(int nodes_count) {
    List<int> my_list;
    for (int i = 0; i < nodes_count; i++) {
        my_list.push_front(rand() % nodes_count);
    }
    int start = clock();
    my_list.pop_back();
    int end = clock();
    std::cout << "nodes_count: " << nodes_count << ", milliseconds: ";
    std::cout << (end - start) * 1000 / CLOCKS_PER_SEC << std::endl;
}
```

```cpp
int main() {

    //Тест push_front
    std::cout << "Test push_front:" << std::endl;
    test_push_front(1000000);
    test_push_front(2000000);
    test_push_front(3000000);
    test_push_front(4000000);
    test_push_front(5000000);
    test_push_front(6000000);
    test_push_front(7000000);
    test_push_front(8000000);
    test_push_front(9000000);
    test_push_front(10000000);

    //Тест push_back
    std::cout << "Test push_back:" << std::endl;
    test_push_back(1000000);
    test_push_back(2000000);
    test_push_back(3000000);
    test_push_back(4000000);
    test_push_back(5000000);
    test_push_back(6000000);
    test_push_back(7000000);
    test_push_back(8000000);
    test_push_back(9000000);
    test_push_back(10000000);

    //Тест pop_front
    std::cout << "Test pop_front:" << std::endl;
    test_pop_front(1000000);
    test_pop_front(2000000);
    test_pop_front(3000000);
    test_pop_front(4000000);
    test_pop_front(5000000);
    test_pop_front(6000000);
    test_pop_front(7000000);
    test_pop_front(8000000);
    test_pop_front(9000000);
    test_pop_front(10000000);

    //Тест pop_back
    std::cout << "Test pop_back:" << std::endl;
    test_pop_back(1000000);
    test_pop_back(2000000);
    test_pop_back(3000000);
    test_pop_back(4000000);
    test_pop_back(5000000);
    test_pop_back(6000000);
    test_pop_back(7000000);
    test_pop_back(8000000);
    test_pop_back(9000000);
    test_pop_back(10000000);

    return 0;
}
```