МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра САПР

ОТЧЕТ
по лабораторной работе №1
по дисциплине «Алгоритмы и структуры данных»
Тема: «Связный список»
Вариант №13

Студент гр. 0302 _____ Приезжих Т.А.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург
2021

Цель

Обучится работе со связными списками, составлению функций для них, освоить использование unit тестов.

Описание алгоритма

Сначала задаётся класс односвязного списка, в него вкладывается структура узла. На основе этого построены следующие функции.

1. void push_back(int); добавление элемента в конец списка
2. void push_front(int); добавление элемента в начало списка
3. void pop_back(); удаление последнего элемента списка
4. void pop_front(); удаление первого элемента списка
5. void insert(int, size_t); добавление элемента списка по индексу (вставка перед элементом, который был ранее доступен по этому индексу)
6. int at(size_t); получение элемента списка по индексу
7. void remove(size_t); удаление элемента списка по индексу
8. size_t get_size(); получение размера списка
9. void clear(); удаление всех элементов списка
10. void set(size_t, int); замена элемента по индексу на передаваемый элемент
11. bool isEmpty(); проверка на пустоту списка
12. Перегрузка оператора вывода <<
19. int find_last(List); поиск последнего вхождения другого списка в список

Оценка временной сложности каждого метода

Метод	Временная сложность
push_back(int)	O(n)
push_front(int)	O(1)
void pop_back()	O(1)
void pop_front()	O(1)
void insert(int, size_t)	O(n)
int at(size_t)	O(n)
void remove(size_t)	O(n)
size_t get_size()	O(n)
void clear()	$O(n^2)$
void set(size_t, int)	2*O(n)
bool isEmpty()	O(1)

int find_last(List)	O(m*n)

Описание реализованных unit-тестов

1. void push_back(int); Добавляем в конец списка элемент и сравниваем значение последнего элемента с ожидаемым

2. void push_front(int); Добавляем в начало списка элемент и сравниваем значение первого элемента с ожидаемым

3. void pop_back(); Создаём список из нескольких элементов, вызываем функцию, сравниваем значение элемента с ожидаемым

4. void pop_front(); Создаём список из нескольких элементов, вызываем функцию, сравниваем значение элемента с ожидаемым

5. void insert(int, size_t); Добавляем элемент с индексом 2 (например), затем с помощью функции at(size_t); проверяем значение элемента с индексом 2

6. int at(size_t); Создаем список, добаляем в него элемент и сравниваем его значение с ожидаемым

7. void remove(size_t); Удаляем элемент с индексом 1 (например), сравниваем значение элемента с ожидаемым

8. size_t get_size(); Добавляем в список 10 элементов (например), вызываем функцию и сравниваем её значение с ожидаемым

9. void clear(); Создаём список из нескольких элементов, вызываем функцию, проверяем список на пустоту

10. void set(size_t, int);

11. bool isEmpty(); Создаём список с одним элементом, затем удаляем элемент и проверяем список на пустоту

19. int find_last(List); Создаём первый список с множеством элементов, повторяем вхождение элементов (например 1 2 3) несколько раз, затем создаём второй список с элементами (например 1 2 3), сравниваем полученный индекс с ожидаемым

Пример работы

```
First list:
60 40 36 20 10 90 47 13 10 90 47 2
Second list:
10 90 47

Element with searched index:20
The last occurrence of the second list in the first list (index): 8


First list after editing: 100

Number of elements in the list:1
List is not empty
List is empty
```

Листинг

Lab1.h

```
#pragma once
#include <iostream>
using namespace std;

class List {
private:
	struct Node {
		int data;
		Node* next;

		Node(int input) {
			data = input;
			next = nullptr;
		}

		int getData() {
			return data;
		}

		Node* getNext() {
			return next;
		}

		void setNext(Node* newnext) {
			next = newnext;
		}
	};

	Node* head;

	void push_front(Node* temp) {
		temp->next = head;
		head = temp;
	}

	void push_back(Node* temp) {
		Node* p = head;
		for (size_t i = 0; i < get_size() - 1; i++) {
			p = p->next;
		}
		p->next = temp;
	}
```

```
void push(Node* temp, Node* prev, Node* prev_plus_one) {
	temp->next = prev_plus_one;
	prev->next = temp;
	temp = prev;

}

void pop_front_() {
	Node* p = head;
	head = head->next;
	delete p;
}

void pop_back_() {
	Node* p = head;
	head = head->next;
	if (head == nullptr)
		delete p;
}

size_t get_size_() {
	Node* p = head;
	size_t count = 0;
	while (p != nullptr) {
		count++;
		p = p->next;
	}
	return count;
}

void remove_(size_t number) {
	Node* p = head;
	Node* cur = head;
	Node* prev = NULL;
	bool found = false;
	for (size_t i = 0; i < get_size() - 1; i++) {
		if (i == number - 1) {
			prev = cur;
			cur = cur->getNext();
		}
		else {
			if (i < number)
				cur = cur->getNext();
		}
	}
	if (prev == NULL) {
		head = cur->getNext();
	}
	else {
		prev->setNext(cur->getNext());
	}
}

int at_(size_t number) {
	Node* p = head;
	for (size_t i = 0; i < get_size(); i++) {
		if (i == number) {
			return p->getData();
		}
		else {
			p = p->next;
		}
	}
	return false;
}
```

```
        void insert_(int data, size_t number) {
                Node* cur = head;
                Node* prev = NULL;
                bool found = false;
                for (size_t i = 0; i < get_size() - 1; i++) {
                        if (i == number - 1) {
                                prev = cur;
                                cur = cur->getNext();
                        }
                        else {
                                if (i < number)
                                        cur = cur->getNext();
                        }
                }
                push(new Node(data), prev, cur);
        }

        int find_last_(List* second_list) {
                Node* p1 = head;
                Node* p2 = second_list->head;
                size_t size1 = get_size();
                size_t size2 = second_list->get_size();
                int index = -1;
                for (size_t i = 0; i < size1; i++) {
                        int n1 = this->at(i);
                        int n2 = second_list->at(0);
                        if (n1 == n2) {
                                if (!(i + (size2 - 1) < size1)) break;
                                bool is_sublist = true;
                                for (size_t j = 0; j < size2 && i + j < size1; j++) {
                                        n1 = this->at(i + j);
                                        n2 = second_list->at(j);
                                        if (n1 != n2) is_sublist = false;
                                }
                                if (is_sublist) index = i;
                        }
                }
                return index;
        }
public:

        List() {
                head = nullptr;
        }

        List(int data) {
                head = new Node(data);
        }

        ~List() {
                while (head != nullptr) {
                        Node* temp = head;
                        head = head->next;
                        delete temp;
                }
        }

        void push_front(int data) { //pushes the element in the front
                if (head == nullptr) {
                        head = new Node(data);
                }
                else
                        push_front(new Node(data));
        }
```

```
	void push_back(int data) { //pushes the element in the back
		if (head == nullptr) {
			head = new Node(data);
		}
		else
			push_back(new Node(data));
	}

	void pop_back() {		//deletes the last element
		if (head == nullptr) return;
		else pop_back_();
	}

	void pop_front() {		//deletes the first element
		if (head == nullptr) return;
		else pop_front_();
	}

	void insert(int data, size_t number) { //inserts an element (data) on index
(number)
		if (head == nullptr) return;
		else insert_(data, number);
	}

	int at(size_t number) {		//Finds an element with index (number)
		if (head == nullptr) return 0;
		else return at_(number);
	}

	void remove(size_t number) {	//Removes an element with index (number)
		if ((head == nullptr) || (number > get_size()) || (number < 0)) return;
		else remove_(number);
	}

	size_t get_size() {		//Gets the size of the list
		if (head == nullptr) return 0;
		else return get_size_();
	}


	void clear() {		//Removes all the elements in the list
		if (head == nullptr) return;
		else while (head != nullptr) pop_front();
	}

	void set(size_t number, int data) {	//Removes an element with index (number)
and puts another element (data) on index
		if (head == nullptr) return;
		else {
			remove(number);
			insert(data, number);
		}
	}

	bool isEmpty() {		//Finds out if the list is empty
		if (head == nullptr)  return true;
		else return false;
	}

	int find_last(List* second_list) {	//Searches for the last occurrence of
another list (second_list) in the list
		if (head == nullptr) return -1;
		else {
			return find_last_(second_list);
```

```
		}
	}

	friend ostream& operator<< (ostream& out, List& l);

};

ostream& operator<< (ostream& out, List& l) {

	List::Node* p = l.head;
	while (p != nullptr) {
		cout << p->data << " ";
		p = p->next;
	}

	return out;
}
```

Lab1.cpp

```
#include <iostream>
#include <locale>
#include "Lab1.h"

using namespace std;


int main()
{

	//Testing if everything is working fine
	List* p = new List(10);
	List* r = new List(10);

	r->push_back(90);
	r->push_back(47);
	p->push_front(20);
	p->push_back(47);
	p->push_front(30);
	p->push_back(13);
	p->push_front(40);
	p->push_front(50);
	p->push_front(60);
	p->push_back(10);
	p->push_back(90);
	p->push_back(47);
	p->push_back(2);
	p->remove(1);
	p->insert(90, 5);
	p->set(2, 36);

	cout << "First list:\n" << *p << "\n";
	cout << "Second list:\n" << *r << "\n";
	cout << "\nElement with searched index:" << p->at(3) << "\n";
	if (p->find_last(r) == -1) cout << "First list do not contain second list\n\n";
	else cout << "The last occurrence of the second list in the first list (index): "
<< p->find_last(r) << "\n\n";
	p->clear();
	p->push_back(100);
	cout << "\nFirst list after editing: " << *p << "\n";
	cout << "\nNumber of elements in the list:" << p->get_size() << "\n";
	if (p->isEmpty()) cout << "List is empty\n"; else cout << "List is not empty\n";
	p->pop_front();
	p->pop_front();
```

```
	p->pop_back();
	p->pop_back();
	if (p->isEmpty()) cout << "List is empty\n"; else cout << "List is not empty\n";
}
```

UnitTest1.cpp

```
#include "pch.h"
#include "CppUnitTest.h"
#include "../Lab1/Lab1.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace Tests
{
	TEST_CLASS(Tests)
	{
	public:

		TEST_METHOD(AddingElements)
		{
			List* p = new List(6);		//List: 6
			Assert::AreEqual(6, p->at(0));

			p->push_back(11);		//List: 6  11
			Assert::AreEqual(11, p->at(1));

			p->push_front(17);		//List: 6  11  17
			Assert::AreEqual(17, p->at(0));

			p->insert(13, 2);		//List: 6  11  13  17
			Assert::AreEqual(13, p->at(2));
		}

		TEST_METHOD(DeletingElements) {
			List* p = new List(6);		//List: 6

			p->pop_back();			//List: nothing
			Assert::IsTrue(p->isEmpty());

			p->push_back(1);
			p->push_back(2);
			p->push_back(3);		//List: 1  2  3

			Assert::IsFalse(p->isEmpty());
			p->pop_front();			//List: 2  3

			p->push_back(4);
			p->push_back(5);		//List: 2  3  4  5
			p->remove(1);			//List: 2  4  5
			Assert::AreEqual(4, p->at(1));

			p->clear();			//List: nothing
			Assert::IsTrue(p->isEmpty());
		}

		TEST_METHOD(SetElement) {
			List* p = new List(6);		 //List: 6

			p->push_back(1);
			p->push_back(2);		 //List: 6  1  2

			p->set(1, 4);			 //List: 6  4  2
			Assert::AreEqual(4, p->at(1));

		}
```

```
		TEST_METHOD(GetSize) {
			List* p = new List(6);		//List: 6

			for (size_t i = 0; i < 9; i++)
				p->push_back(1);		//List: 6  1  1  1  1  1  1  1  1
1	(10 numbers)

			Assert::AreEqual((size_t)10, p->get_size());
		}

		TEST_METHOD(FindLastElement) {
			List* p = new List();
			p->push_back(1);
			p->push_back(2);
			p->push_back(3);			//List: 1  2  3

			for (int i = 0; i < 5; i++) p->push_back(i + 10); //List: 1  2  3  10
11  12  13  14
			p->push_back(1);
			p->push_back(2);
			p->push_back(3);

			for (int i = 0; i < 5; i++) p->push_back(i + 7);  //List: 1  2  3  10
11  12  13  14  1  2  3  7  8  9  10  11
			p->push_back(1);
			p->push_back(2);
			p->push_back(3);

			for (int i = 0; i < 5; i++) p->push_back(i + 8);  //List: 1  2  3  10
11  12  13  14  1  2  3  7  8  9  10  11  1 <--(16th element by index)  2  3  8  9  10
11  12


			List* sub_p = new List();
			sub_p->push_back(1);
			sub_p->push_back(2);
			sub_p->push_back(3);				//List2: 1  2  3

			Assert::AreEqual(16, p->find_last(sub_p));
		}
	};
}
```