**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

ОТЧЁТ

**по лабораторной работе №1**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «Программирование двунаправленных связных списков»**

**Вариант 10**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Кнауб К.В. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

## Постановка задачи. Описание реализуемого класса и методов.

Задание: Реализовать класс двусвязного списка с набором методов.

class Node

|  |  |
| --- | --- |
| **Элемент** | **Описание** |
| Node \*next; | Указатель на следующий элемент списка |
| Node \*prev; | Указатель на предыдущий элемент списка |
| int content; | Целочисленные данные, хранящиеся в списке |

|  |  |  |
| --- | --- | --- |
| **Метод** | **Описание** | **Оценка временной сложности** |
| void push\_back(int) | Добавление в конец списка | O(1) |
| void push\_front(int) | Добавление в начало списка | O(1) |
| void pop\_back() | Удаление последнего элемента | O(1) |
| void pop\_front() | Удаление первого элемента | O(1) |
| void insert(int, size\_t) | Добавление элемента по индексу | O(n) |
| int at(const int) | Получение элемента по индексу | O(n) |
| void remove(size\_t) | Удаление элемента по индексу | O(n) |
| size\_t get\_size() | Получение размера списка | O(1) |
| void print\_to\_console() | Вывод элементов в консоль через разделитель | O(n) |
| void clear() | Удаление всех элементов списка | O(n) |
| void set(size\_t, int) | Замена элемента по индексу на передаваемый элемент | O(n) |
| bool isEmpty() | Проверка на пустоту списка | O(1) |
| bool contains(List) | Проверка на содержание другого списка в списке | O(n) |

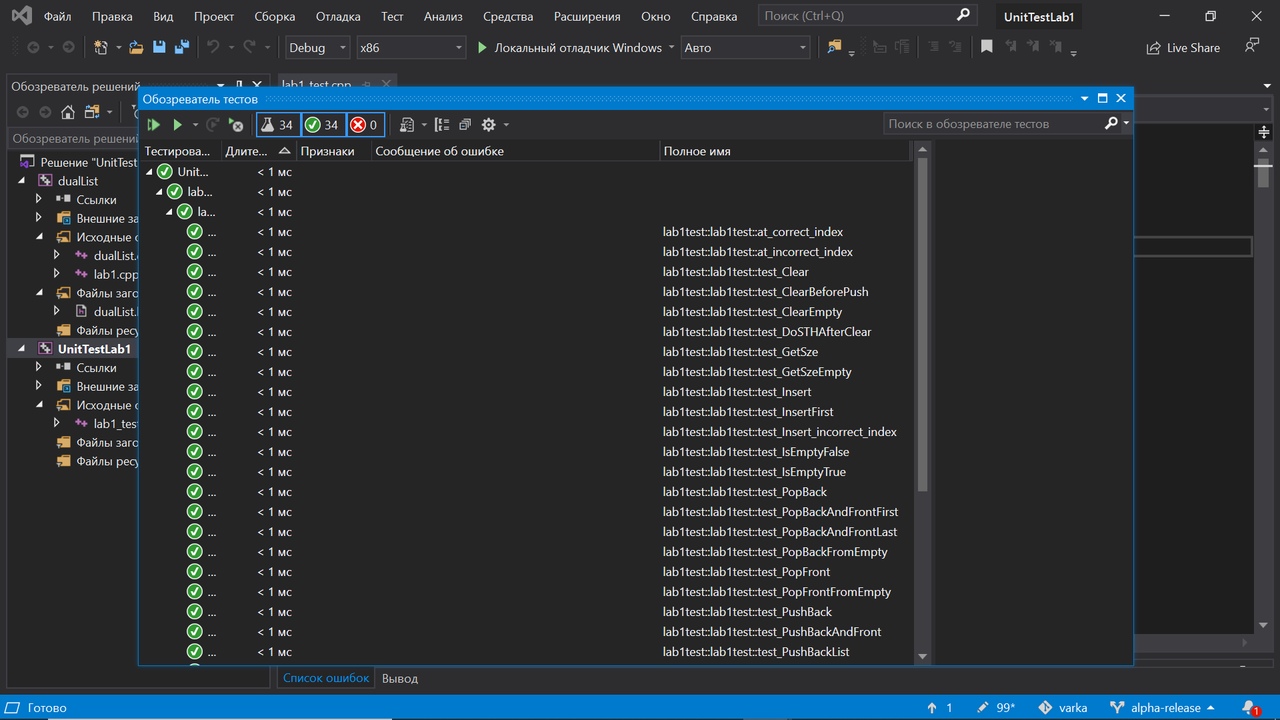
## Описание реализованных unit-тестов.

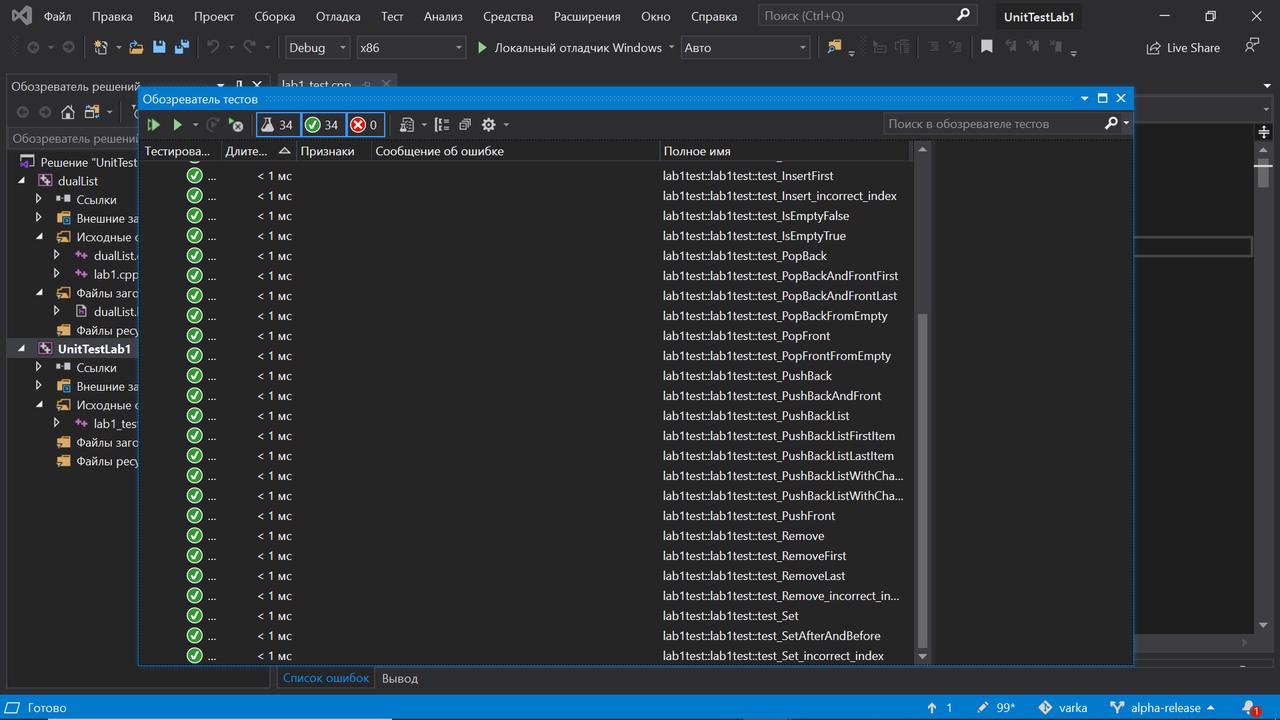
|  |  |
| --- | --- |
| Название теста | Что проверяет |
| TEST\_METHOD(at\_correct\_index) | Проверка на что по индексу получим элемент лежащий по этому индексу |
| TEST\_METHOD(at\_incorrect\_index) | Проверка на обращение к несуществующему элементу |
| TEST\_METHOD(test\_IsEmptyTrue) | Проверка на пустоту |
| TEST\_METHOD(test\_PushFront) | Проверка что последний элемент в начале |
| TEST\_METHOD(test\_PushBack) | Проверка что последний элемент в конце |
| TEST\_METHOD(test\_PushBackAndFront) | Проверка элемента на определенном месте |
| TEST\_METHOD(test\_GetSize) | Проверка на количество элементов |
| TEST\_METHOD(test\_GetSzeEmpty) | Проверка на пустоту |
| TEST\_METHOD(test\_PopBack) | Проверка правильно ли работает удаление с конца |
| TEST\_METHOD(test\_PopBackFromEmpty) | Проверка на удаление с конца при пустом списке |
| TEST\_METHOD(test\_PopFront) | Проверка правильно ли работает удаление с начала |
| TEST\_METHOD(test\_PopFrontFromEmpty) | Проверка на удаление с начала при пустом списке |
| TEST\_METHOD(test\_PopBackAndFrontFirst) | Проверка сколько элементов осталось после удаления с конца и начала |
| TEST\_METHOD(test\_PopBackAndFrontLast) | Проверка на размер осталось после удаления с конца и начала |
| TEST\_METHOD(test\_Remove) | Проверка на удаление элемента по индексу |
| TEST\_METHOD(test\_RemoveFirst) | Проверка с начала на удаление элемента по индексу |
| TEST\_METHOD(test\_RemoveLast) | Проверка с конца на удаление элемента по индексу |
| TEST\_METHOD(test\_Remove\_incorrect\_index) | Проверка на вставку при некорректном индексе |
| TEST\_METHOD(test\_Insert) | Проверка на добавление элемента по индексу |
| TEST\_METHOD(test\_InsertFirst) | Проверка на добавление элемента по индексу в начало |
| TEST\_METHOD(test\_Insert\_incorrect\_index) | Проверка на добавление элемента при некорректном индексе |
| TEST\_METHOD(test\_Clear) | Проверка на удаление всех элементов списка |
| TEST\_METHOD(test\_ClearEmpty) | Проверка на удаление всех элементов списка при пустом |
| TEST\_METHOD(test\_ClearBeforePush) | Проверка на пустоту и не пустоту при добавлении и удалении |
| TEST\_METHOD(test\_DoSTHAfterClear) | Проверка на пустоту после того как добавили элементы а потом их удалили |
| TEST\_METHOD(test\_Set) | Проверка на замену элемента по индексу на передаваемый элемент |
| TEST\_METHOD(test\_SetAfterAndBefore) | Проверка на замену элемента по индексу на заданное значение |
| TEST\_METHOD(test\_Set\_incorrect\_index) | Проверка на замену элемента по индексу на передаваемый элемент при некорректном индексе |
| TEST\_METHOD(test\_list\_in\_list) | Проверка на содержание другого списка в списке (лист 1 2 3 4 содержит лист 2 4 - не верно) |
| TEST\_METHOD(test\_list\_in\_list2) | Проверка на содержание другого списка в списке (лист 1 2 3 4 содержит лист 1 2 - верно) |

## Листинг

|  |
| --- |
| dualList.h |
| |  | | --- | | #pragma once | |  | #include <iostream> | |  |  | |  | class dualList | |  | { | |  | private: | |  | class Node | |  | { | |  | public: | |  | Node(int content = 0, Node\* before = NULL, Node\* next = NULL) | |  | { | |  | this->content = content; | |  | this->before = before; | |  | this->next = next; | |  | }; | |  | ~Node() | |  | { } | |  | int content; | |  | Node\* next; | |  | Node\* before; | |  | private: | |  |  | |  | }; | |  |  | |  | Node\* head; //head list | |  | Node\* tail; //tail list | |  | size\_t size; //list size | |  |  | |  | public: | |  | dualList(Node\* head = NULL, Node\* tail = NULL) { | |  | this->head = head; | |  | this->tail = tail; | |  | size = 0; | |  | }; | |  | void push\_back(int); // add item to end | |  | void push\_front(int); //add to begin | |  | size\_t get\_size(); // get list size | |  | void pop\_back(); // delete last item | |  | void pop\_front(); // delete first item | |  | int at(size\_t); // get item by index | |  | void remove(size\_t); // delete item by index | |  | void insert(int, size\_t); // add item before "size\_t" item | |  | void print\_to\_console(); // print item to console | |  | void clear(); // clear list | |  | void set(size\_t, int); // change "size\_t" item to "int" | |  | bool isEmpty(); // test for emptiness | |  | bool list\_in\_list(dualList\*); // chek if first list contains second list | |  | ~dualList() | |  | { | |  | while (head->next != NULL) //while we can go next | |  | { | |  | head = head->next; | |  | delete head->before; | |  | } | |  | size = 0; | |  | delete head; | |  | }; | |  | }; | |
| dualList.cpp |
| |  | | --- | |  | | #include "dualList.h" | |  | using namespace std; | |  |  | |  | bool dualList::list\_in\_list(dualList) | |  | { | |  | Node\* what\_current = list\_what->head; | |  | for (Node\* where\_current = list\_where->head; where\_current != list\_where->tail; where\_current = where\_current->next) | |  | { | |  | if (where\_current->info == list\_what->head->info) | |  | { | |  | Node\* start = where\_current; | |  | while (where\_current->info == what\_current->info) | |  | { | |  | if (what\_current == list\_what->tail) | |  | return true; | |  | if (where\_current == list\_where->tail) | |  | return false; | |  | where\_current = where\_current->next; | |  | what\_current = what\_current->next; | |  | } | |  | where\_current = start->next; | |  | what\_current = list\_what->head; | |  | } | |  | } | |  | return false; | |  | } | |  |  | |  | void dualList::push\_back(int add) | |  | { | |  | if (size == 0) //if list is empty | |  | { | |  | head = new Node(add); //create head | |  | tail = head; | |  | } | |  | else | |  | { | |  | Node\* curr = new Node(add); | |  | curr->before = tail; | |  | tail->next = curr; | |  | tail = curr; //set new tail | |  | } | |  | size++; | |  | } | |  |  | |  | void dualList::push\_front(int add) | |  | { | |  | if (size == 0) | |  | { | |  | head = new Node(add); | |  | tail = head; | |  | } | |  | else | |  | { | |  | Node\* curr = new Node(add); | |  | curr->next = head; | |  | head->before = curr; | |  | head = curr; //set new head | |  | } | |  | size++; | |  | } | |  |  | |  | size\_t dualList::get\_size() | |  | { | |  | return size; | |  | } | |  |  | |  | void dualList::pop\_back() | |  | { | |  | if (size == 1) | |  | { | |  | size = 0; //setup zero position | |  | head = NULL; | |  | tail = NULL; | |  | } else if (size>1) //if list is not empty | |  | { | |  | tail = tail->before; //set new tail | |  | delete tail->next; | |  | tail->next = NULL; | |  | size--; | |  | } else //else return error | |  | throw out\_of\_range("Segmentation fault"); | |  | } | |  |  | |  | void dualList::pop\_front() | |  | { | |  | if (size == 1) | |  | { | |  | size = 0; //setup zero position | |  | head = NULL; | |  | tail = NULL; | |  | } else if (size>1) //if list is not empty | |  | { | |  | head = head->next; //set new head | |  | delete head->before; | |  | head->before = NULL; | |  | size--; | |  | } else //else return error | |  | throw out\_of\_range("Segmentation fault"); | |  | } | |  |  | |  | void dualList::remove(size\_t i) | |  | { | |  | if (i >= size) //if index is larger than size return error | |  | throw out\_of\_range("Index is greater than list size"); | |  | else if (i == 0) // if index points to first | |  | pop\_front(); | |  | else if (i == size - 1) // if index points to first | |  | pop\_back(); | |  | else | |  | { | |  | Node\* cur = head; | |  | while (i) //go to "i" item | |  | { | |  | cur = cur->next; | |  | i--; | |  | } | |  | cur->before->next = cur->next; //cut item from list | |  | size--; | |  | delete cur; | |  | } | |  | } | |  |  | |  | int dualList::at(size\_t i) | |  | { | |  | if (i >= size) //if index is larger than size return error | |  | throw out\_of\_range("Index is greater than list size"); | |  | else | |  | { | |  | Node\* cur=head; | |  | while (i) //go to "i" item | |  | { | |  | cur = cur->next; | |  | i--; | |  | } | |  | return cur->info; | |  | } | |  | } | |  |  | |  | void dualList::insert(int add, size\_t i) | |  | { | |  | if (i>=size) //if index is larger than size return error | |  | throw out\_of\_range("Index is greater than list size"); | |  | else if (i == 0) //if insert before first item | |  | push\_front(add); | |  | else | |  | { | |  | Node\* cur = head; | |  | while (i) | |  | { | |  | cur = cur->next; | |  | i--; | |  | } | |  | Node\* nadd = new Node(add, cur->before, cur); //create new Node with new connection | |  | cur->before->next = nadd; //add poin to new Node | |  | cur->before = nadd; | |  | size++; | |  | } | |  | } | |  |  | |  | void dualList::print\_to\_console() | |  | { | |  | Node\* cur=head; | |  | while (cur != NULL) | |  | { | |  | cout << cur->info << "| "; | |  | cur = cur->next; | |  | } | |  | } | |  |  | |  | void dualList::clear() | |  | { | |  | if (head != NULL) | |  | { | |  | while (head->next != NULL) //delete all item | |  | { | |  | head = head->next; | |  | delete head->before; | |  | } | |  | size = 0; //setup zero position | |  | head = NULL; | |  | tail = NULL; | |  | } | |  | } | |  |  | |  | void dualList::set(size\_t i, int ins) | |  | { | |  | if (i>=size) //if index is larger than size return error | |  | throw out\_of\_range("Index is greater than list size"); | |  | else { | |  | Node\* cur = head; | |  | while (i) //go to "i" item | |  | { | |  | cur = cur->next; | |  | i--; | |  | } | |  | cur->info = ins; //change content | |  | } | |  | } | |  |  | |  | bool dualList::isEmpty() | |  | { | |  | if (head == NULL) | |  | return true; | |  | return false; | |  | } | |  |  | |
| dualList\_test.cpp |
| |  | | --- | | #include "CppUnitTest.h" | |  | #include "dualList.h" /\*/Users/Documents/Алгоритмы/кхм/\*/ | |  |  | |  | using namespace Microsoft::VisualStudio::CppUnitTestFramework; | |  |  | |  | namespace lab1test | |  | { | |  | TEST\_CLASS(lab1test) | |  | { | |  | public: | |  | dualList\* current\_dual\_list; | |  | dualList\* another\_dual\_list; | |  | TEST\_METHOD\_INITIALIZE(setup) | |  | { | |  | current\_dual\_list = new dualList(); | |  | another\_dual\_list = new dualList(); | |  | } | |  |  | |  | TEST\_METHOD(at\_correct\_index) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | Assert::AreEqual(current\_dual\_list->at(1),2); | |  | } | |  | TEST\_METHOD(at\_incorrect\_index) | |  | { | |  | try { | |  | current\_dual\_list->at(-1); | |  | } | |  | catch (std::out\_of\_range e) { | |  | Assert::AreEqual("Index is greater than list size", e.what()); | |  | } | |  |  | |  | } | |  |  | |  | TEST\_METHOD(test\_IsEmptyTrue) | |  | { | |  | Assert::AreEqual(current\_dual\_list->isEmpty(), true); | |  | } | |  |  | |  | TEST\_METHOD(test\_IsEmptyFalse) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | Assert::AreEqual(current\_dual\_list->isEmpty(), false); | |  | } | |  |  | |  | TEST\_METHOD(test\_PushFront) | |  | { | |  | current\_dual\_list->push\_front(3); | |  | current\_dual\_list->push\_front(2); | |  | current\_dual\_list->push\_front(1); | |  | Assert::AreEqual(current\_dual\_list->at(0), 1); | |  | } | |  | TEST\_METHOD(test\_PushBack) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(3); | |  | Assert::AreEqual(current\_dual\_list->at(2), 3); | |  | } | |  | TEST\_METHOD(test\_PushBackAndFront) | |  | { | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(3); | |  | current\_dual\_list->push\_front(1); | |  | current\_dual\_list->push\_back(4); | |  | Assert::AreEqual(current\_dual\_list->at(2), 3); | |  | Assert::AreEqual(current\_dual\_list->at(0), 1); | |  | Assert::AreEqual(current\_dual\_list->at(3), 4); | |  | } | |  | TEST\_METHOD(test\_GetSze) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(3); | |  | Assert::AreEqual(current\_dual\_list->get\_size(), (size\_t)3); | |  | } | |  | TEST\_METHOD(test\_GetSzeEmpty) | |  | { | |  | Assert::AreEqual(current\_dual\_list->get\_size(), (size\_t)0); | |  | } | |  |  | |  | TEST\_METHOD(test\_PopBack) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(3); | |  | current\_dual\_list->pop\_back(); | |  | current\_dual\_list->pop\_back(); | |  | Assert::AreEqual(current\_dual\_list->get\_size(), (size\_t)1); | |  | Assert::AreEqual(current\_dual\_list->at(0), 1); | |  | } | |  |  | |  | TEST\_METHOD(test\_PopBackFromEmpty) | |  | { | |  | try { | |  | current\_dual\_list->pop\_back(); | |  | } | |  | catch (std::out\_of\_range e) { | |  | Assert::AreEqual("Segmentation fault", e.what()); | |  | } | |  | } | |  | TEST\_METHOD(test\_PopFront) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(3); | |  | current\_dual\_list->pop\_front(); | |  | current\_dual\_list->pop\_front(); | |  | Assert::AreEqual(current\_dual\_list->get\_size(), (size\_t)1); | |  | Assert::AreEqual(current\_dual\_list->at(0), 3); | |  | } | |  |  | |  | TEST\_METHOD(test\_PopFrontFromEmpty) | |  | { | |  | try { | |  | current\_dual\_list->pop\_front(); | |  | } | |  | catch (std::out\_of\_range e) { | |  | Assert::AreEqual("Segmentation fault", e.what()); | |  | } | |  | } | |  |  | |  | TEST\_METHOD(test\_PopBackAndFrontFirst) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(3); | |  | current\_dual\_list->push\_front(0); | |  | current\_dual\_list->pop\_front(); | |  | current\_dual\_list->pop\_back(); | |  | Assert::AreEqual(current\_dual\_list->get\_size(), (size\_t)2); | |  | Assert::AreEqual(current\_dual\_list->at(0), 1); | |  | } | |  | TEST\_METHOD(test\_PopBackAndFrontLast) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(3); | |  | current\_dual\_list->push\_front(0); | |  | current\_dual\_list->pop\_front(); | |  | current\_dual\_list->pop\_back(); | |  | Assert::AreEqual(current\_dual\_list->get\_size(), (size\_t)2); | |  | Assert::AreEqual(current\_dual\_list->at(1), 2); | |  | } | |  |  | |  | TEST\_METHOD(test\_Remove) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(3); | |  | current\_dual\_list->remove(1); | |  | Assert::AreEqual(current\_dual\_list->get\_size(), (size\_t)2); | |  | Assert::AreEqual(current\_dual\_list->at(1), 3); | |  | } | |  | TEST\_METHOD(test\_RemoveFirst) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(3); | |  | current\_dual\_list->remove(0); | |  | Assert::AreEqual(current\_dual\_list->get\_size(), (size\_t)2); | |  | Assert::AreEqual(current\_dual\_list->at(0), 2); | |  | } | |  | TEST\_METHOD(test\_RemoveLast) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(3); | |  | current\_dual\_list->remove(2); | |  | Assert::AreEqual(current\_dual\_list->get\_size(), (size\_t)2); | |  | Assert::AreEqual(current\_dual\_list->at(1), 2); | |  | } | |  | TEST\_METHOD(test\_Remove\_incorrect\_index) | |  | { | |  | try { | |  | current\_dual\_list->remove(-1); | |  | } | |  | catch (std::out\_of\_range e) { | |  | Assert::AreEqual("Index is greater than list size", e.what()); | |  | } | |  | } | |  |  | |  | TEST\_METHOD(test\_Insert) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(4); | |  | current\_dual\_list->insert(3, 2); | |  | Assert::AreEqual(current\_dual\_list->get\_size(), (size\_t)4); | |  | Assert::AreEqual(current\_dual\_list->at(2), 3); | |  | } | |  | TEST\_METHOD(test\_InsertFirst) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(4); | |  | current\_dual\_list->insert(0, 0); | |  | Assert::AreEqual(current\_dual\_list->get\_size(), (size\_t)4); | |  | Assert::AreEqual(current\_dual\_list->at(0), 0); | |  | Assert::AreEqual(current\_dual\_list->at(1), 1); | |  | } | |  | TEST\_METHOD(test\_Insert\_incorrect\_index) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(3); | |  | try { | |  | current\_dual\_list->insert(4, 3); | |  | } | |  | catch (std::out\_of\_range e) { | |  | Assert::AreEqual("Index is greater than list size", e.what()); | |  | } | |  | } | |  |  | |  | TEST\_METHOD(test\_Clear) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(3); | |  | current\_dual\_list->clear(); | |  | Assert::AreEqual(current\_dual\_list->isEmpty(), true); | |  | } | |  | TEST\_METHOD(test\_ClearEmpty) | |  | { | |  | current\_dual\_list->clear(); | |  | Assert::AreEqual(current\_dual\_list->isEmpty(), true); | |  | } | |  | TEST\_METHOD(test\_ClearBeforePush) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->clear(); | |  | current\_dual\_list->push\_back(1); | |  | Assert::AreEqual(current\_dual\_list->isEmpty(), false); | |  | Assert::AreEqual(current\_dual\_list->at(0), 1); | |  | } | |  | TEST\_METHOD(test\_DoSTHAfterClear) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(3); | |  | current\_dual\_list->clear(); | |  | try { | |  | current\_dual\_list->at(0); | |  | } | |  | catch (std::out\_of\_range e) { | |  | Assert::AreEqual("Index is greater than list size", e.what()); | |  | } | |  | } | |  |  | |  | TEST\_METHOD(test\_Set) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(3); | |  | current\_dual\_list->set(1, 0); | |  | Assert::AreEqual(current\_dual\_list->at(1), 0); | |  | Assert::AreEqual(current\_dual\_list->at(2), 3); | |  | } | |  |  | |  | TEST\_METHOD(test\_SetAfterAndBefore) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(3); | |  | current\_dual\_list->set(1, 0); | |  | Assert::AreEqual(current\_dual\_list->at(0), 1); | |  | Assert::AreEqual(current\_dual\_list->at(2), 3); | |  | } | |  | TEST\_METHOD(test\_Set\_incorrect\_index) | |  | { | |  | current\_dual\_list->push\_back(1); | |  | try { | |  | current\_dual\_list->set(8, 0); | |  | } | |  | catch (std::out\_of\_range e) { | |  | Assert::AreEqual("Index is greater than list size", e.what()); | |  | } | |  | } | |  | TEST\_METHOD(test\_list\_in\_list) | |  | { | |  | current\_dual\_list = new dualList(); | |  | another\_dual\_list = new dualList(); | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(3); | |  | current\_dual\_list->push\_back(4); | |  | another\_dual\_list->push\_back(2); | |  | another\_dual\_list->push\_back(4); | |  | Assert::AreEqual(current\_dual\_list->list\_in\_list(another\_dual\_list), false); | |  | } | |  | TEST\_METHOD(test\_list\_in\_list2) | |  | { | |  | current\_dual\_list = new dualList(); | |  | another\_dual\_list = new dualList(); | |  | current\_dual\_list->push\_back(1); | |  | current\_dual\_list->push\_back(2); | |  | current\_dual\_list->push\_back(3); | |  | current\_dual\_list->push\_back(4); | |  | another\_dual\_list->push\_back(1); | |  | another\_dual\_list->push\_back(2); | |  | Assert::AreEqual(current\_dual\_list->list\_in\_list(another\_dual\_list), true); | |  | } | |  |  | |  | }; | |  | } | |
| main.cpp |
| |  | | --- | | #include <iostream> | |  | #include "dualList.h" | |  |  | |  | using namespace std; | |  |  | |  | int main() | |  | { | |  | dualList\* DualList=new dualList; | |  | size\_t i, count; | |  | int information; | |  | cout << "How many elements you want to add? "; | |  | cin >> count; | |  | for (i = 0; i < count; i++) | |  | { | |  | cout << i << " item is: "; | |  | cin >> information; | |  | DualList->push\_back(information); | |  | } | |  | cout << "ListNow: " << endl; | |  | DualList->print\_to\_console(); | |  | cout << endl; | |  | cout <<"ListSize: " << DualList->get\_size() << endl; | |  | cout << "Index item to insert and contend " << endl; | |  | cin >> i >> information; | |  | DualList->insert(information, i); | |  | cout << "ListNow: "<< endl; | |  | DualList->print\_to\_console(); | |  | } | |

## Результат выполнения всех unit-тестов





## Вывод

В данной лабораторной работе мы научились реализовывать связные списки и работать с ними. Также мы научились проверять работу написанной программы с помощью unit-тестов.