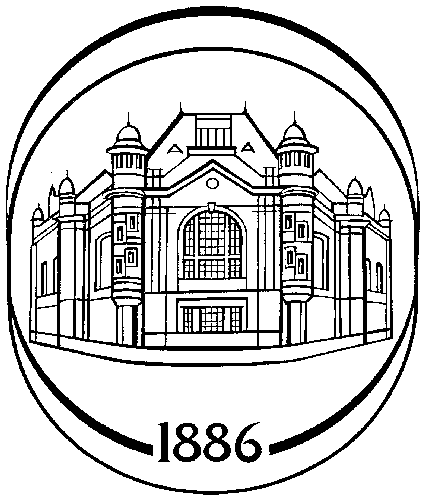
МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное учреждение высшего образования

**Санкт-Петербургский государственный электротехнический университет “ЛЭТИ” им. В. И. Ульянова (Ленина) (СПбГЭТУ «ЛЭТИ»)**



Кафедра САПР

Лабораторная работа № 1

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

Тема: «Списки»

Вариант № 23 (7)

|  |  |  |
| --- | --- | --- |
| Студентка гр. 9301, ФКТИ |  | Синицкая В. А. |
| Преподаватель |  | Тутуева А. В. |

Санкт-Петербург

2020 г.

Постановка задачи

Задача для 7 варианта: односвязный список с реализацией методов:

1. void push\_back(string); // добавление в конец списка
2. void push\_front(string); // добавление в начало списка
3. void pop\_back(); // удаление последнего элемента
4. void pop\_front(); // удаление первого элемента
5. void insert(string, size\_t) // добавление элемента по индексу (вставка перед элементом, который был ранее доступен по этому индексу)
6. element\* at(size\_t); // получение элемента по индексу. Можно сделать типа size\_t
7. void remove(size\_t); // удаление элемента по индексу
8. size\_t get\_size(); // получение размера списка
9. void print\_to\_console();
10. void clear(); // удаление всех элементов списка
11. void set(size\_t, string); // замена элемента по индексу на передаваемый элемент
12. bool isEmpty(); // проверка на пустоту списка
13. void push\_front(L1List); // вставка другого списка в начало

Описание класса и методов

Класс называется L1List и содержит поля-указатели на объекты класса element: head, cur, tail. Класс element содержит поле data типа string и поле next с указателем на следующий элемент класса. В обоих классах реализованы методы set и get для указанных полей, конструкторы и деструкторы, в классе L1List содержатся все перечисленные в постановке задачи методы. Изначально не было tail, и везде в программе признаком конца списка служит nullptr, однако так как push\_back – один из самых часто используемых методов, было решено все же ввести tail для ускорения работы программы. Отдельное внимание стоит уделить вставке списка в начало другого, его реализация может показаться странной, но она такая для уменьшения временной сложности. Будем называть списком слева тот, что мы добавляем в начало, а списком справа тот, к которому добавляем. Тогда алгоритм таков:

В список справа добавляется копия головного элемента списка слева. Копия головного элемента левого списка запоминается как его головной элемент и связывается далее с остальными элементами-копиями. Последующие элементы левого списка (их копии) вставляются в правый на место с индексом 1, притом голова правого списка движется вправо, чтобы процесс вставки был наименее трудоемок, и индекс оставался одним и тем же. 1 – лучший индекс, так как все необходимые связи будут установлены, а лишних итераций будет меньше. Процесс завершается, когда элементы в левом списке заканчиваются. Головной элемент правого списка устанавливается копией головного элемента левого списка. На этом алгоритм завершается.

Оценка временной сложности

1. void push\_back(string) = О(1) – не зависит от количества элементов в списке
2. void push\_front(string) = О(1) – не зависит от количества элементов в списке
3. void pop\_back() = О(n-1) – зависит от количества элементов в списке, так как он не двусвязный, нужно все равно дойти от начала до предпоследнего элемента, чтобы изменить tail
4. void pop\_front() = О(1) – не зависит от количества элементов в списке
5. void insert(string, size\_t) = O(index-1), в худшем случае O(n) - зависит от индекса (от количества элементов в списке в худшем случае)
6. element\* at(size\_t) = O(index), в худшем случае O(n) - зависит от индекса (от количества элементов в списке в худшем случае)
7. void remove(size\_t) = O(index-1), в худшем случае O(n) - зависит от индекса (от количества элементов в списке в худшем случае)
8. size\_t get\_size() = O(n) - зависит от количества элементов в списке
9. void print\_to\_console() = О(n) - зависит от количества элементов в списке
10. void clear() = О(n) - зависит от количества элементов в списке
11. void set(size\_t, string) = O(index), в худшем случае O(n) - зависит от индекса (от количества элементов в списке в худшем случае)
12. bool isEmpty() = О(1) – не зависит от количества элементов в списке
13. void push\_front(L1List) = O(2\*s) – зависит от количества элементов в списке, который становится началом данного

Описание реализованных unit-тестов

Всего был реализован 41 unit-test:

1. TEST\_METHOD(IsEmptyListWhenEmpty) – проверка, что лист пуст, когда он пуст
2. TEST\_METHOD(IsEmptyListWhenNotEmpty) – проверка, что лист не пуст, когда в него положили элемент
3. TEST\_METHOD(ListSizeZero) – проверка, что у пустого листа размер = 0
4. TEST\_METHOD(ListSizeNotZero) – проверка, что размер = количеству элементов в списке
5. TEST\_METHOD(PushBackOneElement) – проверка, что первый элемент вставляется назад правильно
6. TEST\_METHOD(PushBackElements) – проверка, что не первый элемент вставляется правильно в конец
7. TEST\_METHOD(PopBackFromEmptyList) – проверка, что бросается исключение при попытке удаления с конца из пустого листа
8. TEST\_METHOD(PopBackFromListWithOneElement) – проверка, что лист будет пуст, если из него удалить с конца единственный элемент
9. TEST\_METHOD(PopBackFromList) – проверка, что не первый элемент удаляется с конца верно
10. TEST\_METHOD(PushFrontInEmptyList) – проверка, что первый элемент в списке добавляется в него спереди верно
11. TEST\_METHOD(PushFrontInNotEmptyList) – проверка, что добавление в начало в непустой лист делается верно
12. TEST\_METHOD(PopFrontFromEmptyList) – проверка, что удаление спереди в пустом листе вызывает ошибку
13. TEST\_METHOD(PopFrontFromList) – проверка, что удаление спереди в непустом списке происходит правильно
14. TEST\_METHOD(PopFrontFromOneElementList) – проверка, что после удаления спереди единственного элемента остается пустой лист
15. TEST\_METHOD(AtInEmptyList) – проверка, что будет ошибка при попытке обратиться к элементу с несуществующим индексом
16. TEST\_METHOD(AtInListWhenPushFronted) – проверка получения элемента по индексу, когда элементы были добавлены в начало
17. TEST\_METHOD(AtInListWhenPushBacked) – проверка получения элемента по индексу, когда элементы были добавлены в конец
18. TEST\_METHOD(AtInListWhenPushed) - проверка получения элемента по индексу, когда элементы были добавлены разными способами
19. TEST\_METHOD(InsertToListBegin) – проверка, что вставка на первое место происходит правильно
20. TEST\_METHOD(InsertToListEnd) – проверка, что вставка в конец происходит правильно
21. TEST\_METHOD(InsertToListMiddle) – проверка вставки в середину списка
22. TEST\_METHOD(InsertToListInvalidIndex) – проверка ошибки в случае попытки вставки по несуществующему индексу
23. TEST\_METHOD(RemoveFromListInvalidIndex) – проверка, что нельзя удалить несуществующий элемент
24. TEST\_METHOD(RemoveFromOneElementList) – проверка удаления из списка с одним элементом
25. TEST\_METHOD(RemoveFromList) – проверка удаления из середины списка
26. TEST\_METHOD(RemoveFromListEnd) – проверка удаления элемента с конца списка
27. TEST\_METHOD(RemoveFromListBegin) – проверка удаления элемента с начала списка
28. TEST\_METHOD(SetInListBegin) – проверка замены данных в первом элементе
29. TEST\_METHOD(SetInListEnd) – проверка замены данных в конечном элементе
30. TEST\_METHOD(SetInEmptyList) – проверка запрета смены данных в пустом листе
31. TEST\_METHOD(SetInListInvalidIndex) – проверка запрета смены данных в несуществующем элементе
32. TEST\_METHOD(SetInList) – проверка замены данных в элементе посередине
33. TEST\_METHOD(ClearEmptyList) – проверка запрета на очистку пустого листа
34. TEST\_METHOD(ClearList) – проверка очистки не пустого листа
35. TEST\_METHOD(PushFrontAList) – проверка вставки одного листа в начало другого
36. TEST\_METHOD(PushFrontAListChangeOne) – проверка вставки одного листа в другой, когда добавляемый лист изменяется, чтобы проверить, что другой независим
37. TEST\_METHOD(PushFrontAListClearOne) – проверка вставки одного листа в другой, когда добавляемый лист очищается, чтобы проверить, что нет ошибок с памятью
38. TEST\_METHOD(PushFrontAListClearTwo) – проверка вставки одного листа в другой, когда итоговый лист очищается, чтобы проверить, что нет ошибок с памятью
39. TEST\_METHOD(PushFrontAListChangeTwo) – проверка вставки одного листа в другой, когда итоговый лист изменяется, чтобы проверить, что другой независим
40. TEST\_METHOD(PushFrontAListClearBoth) – проверка вставки одного листа в другой, когда оба очищаются, проверка, что оба пусты и нет ошибок с памятью
41. TEST\_METHOD(PushFrontAListSecondEmpty) – проверка вставки одного листа в другой, когда тот, в который вставляют, пуст.

Пример работы

При функции main():

1. int main()
2. {
3. L1List a;
4. L1List b;
5. b.push\_back("hey");
6. b.push\_front("hmmm");
7. b.push\_back("have you ever thought that...");
8. a.push\_back("maybe");
9. a.push\_back("is");
10. a.insert("today", 1);
11. a.push\_back("bla");
12. a.set(3, "a very");
13. a.push\_back("good");
14. a.insert("day?", 5);
15. a.push\_front(&b);
16. cout << a.get\_size() << endl;
17. a.print\_to\_console();
18. a.clear();
19. cout << a.isEmpty() << endl;
20. return 0;
21. }

Результат, выведенный в консоль:

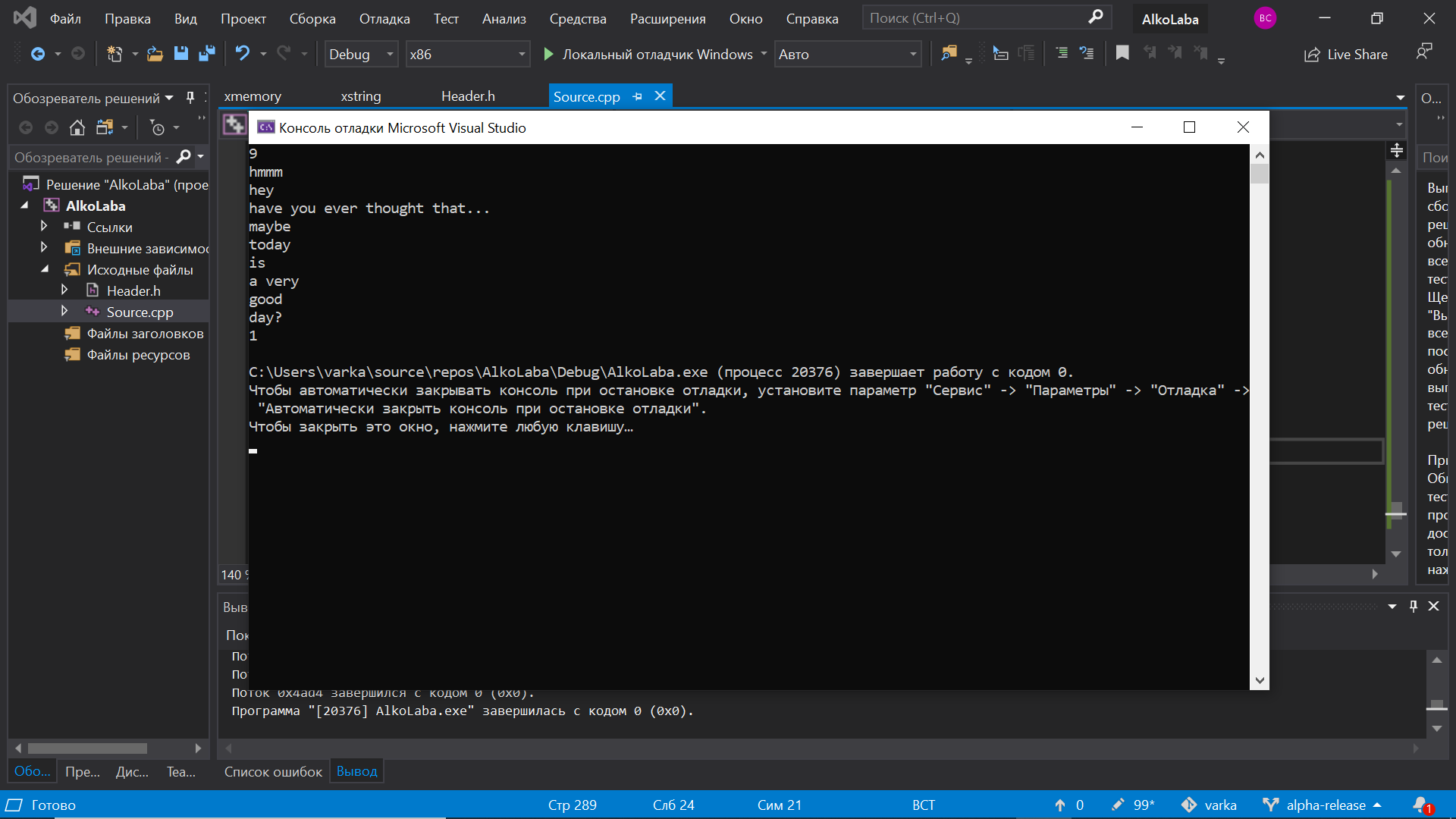


Рис. 1.1 – Пример работы программы

Листинг

L1List.h:

1. #pragma once
2. #include <iostream>
3. #include <stdexcept>
4. using namespace std;
5. class element
6. {
7. element\* next;
8. string data;
9. public:
10. element();
11. element\* get\_next();
12. string get\_data();
13. void set\_next(element\*);
14. void set\_data(string);
15. ~element();
16. };
17. class L1List
18. {
19. element\* head = nullptr;
20. element\* cur = nullptr;
21. element\* tail = nullptr;
22. public:
23. L1List();
24. void set\_head(element\*);
25. void set\_cur(element\*);
26. void set\_tail(element\*);
27. element\* get\_head();
28. element\* get\_cur();
29. element\* get\_tail();
30. void set\_next(element\*); // change next regarding to cur
31. void set\_data(string); // chage cur data
32. string get\_data(); // get cur data
33. element\* get\_next(); // get next regarding to cur
34. bool isEmpty(); // check for empty list
35. void push\_back(string); // adding to the end of the list
36. void push\_front(string); // adding to the begining of the list
37. void pop\_back(); // delete last element
38. void pop\_front(); // delete first element
39. size\_t get\_size(); // get size of the list
40. void insert(string, size\_t); // adding element on index (before the element, that had this index lately)
41. element\* at(size\_t); // return element on index
42. void remove(size\_t); // delete element on index
43. void print\_to\_console(); // print elements to console without using at()
44. void clear(); // delete all elements of the list
45. void set(size\_t, string); // change data of element on index
46. void push\_front(L1List\*); // insertion another list into begining of the data-list
47. ~L1List();
48. };

L1List.cpp:

1. #include "L1List.h"
2. using namespace std;
3. element::element()
4. {
5. data = "";
6. next = nullptr;
7. }
8. element\* element::get\_next() { return next; }
9. string element::get\_data() { return data; }
10. void element::set\_next(element\* e) { next = e; }
11. void element::set\_data(string s) { data = s; }
12. element::~element() { }
13. L1List::L1List()
14. {
15. head = nullptr;
16. cur = nullptr;
17. tail = nullptr;
18. }
19. element\* L1List::get\_head() { return head; }
20. element\* L1List::get\_cur() { return cur; }
21. element\* L1List::get\_tail() { return tail; }
22. void L1List::set\_head(element\* e) { head = e; }
23. void L1List::set\_cur(element\* e) { cur = e; }
24. void L1List::set\_tail(element\* e) { tail = e; }
25. void L1List::set\_next(element\* e) { cur->set\_next(e); }
26. void L1List::set\_data(string data) { cur->set\_data(data); }
27. element\* L1List::get\_next() { return (cur->get\_next()); }
28. string L1List::get\_data() { return (cur->get\_data()); }
29. bool L1List::isEmpty()
30. {
31. if (head != nullptr)
32. return false; // list is not empty
33. return true;
34. }
35. void L1List::push\_back(string data)
36. {
37. element\* e = new element;
38. e->set\_data(data);
39. if (isEmpty())
40. head = cur = tail = e;
41. else
42. {
43. tail->set\_next(e);
44. tail = tail->get\_next();
45. cur = tail;
46. }
47. }
48. void L1List::push\_front(string data)
49. {
50. element\* e = new element;
51. e->set\_data(data);
52. if (isEmpty())
53. head = cur = tail = e;
54. else
55. {
56. cur = e; // cur = new elem
57. cur->set\_next(head); // cur->next = head
58. head = cur; // head = cur
59. }
60. }
61. void L1List::pop\_back()
62. {
63. if (!isEmpty())
64. {
65. set\_cur(get\_head()); // cur = head
66. if (get\_next() == nullptr) // delete head
67. {
68. element\* del = head;
69. cur = head = tail = nullptr;
70. delete del;
71. }
72. else
73. {
74. while (get\_next()->get\_next() != nullptr) // while cur->next->next exists
75. {
76. set\_cur(get\_next()); // cur = cur->next
77. } // cur = the element before the last existing element
78. element\* del = get\_next();
79. cur->set\_next(nullptr);
80. tail = cur;
81. delete del;
82. }
83. }
84. else
85. throw out\_of\_range("The List is empty");
86. }
87. void L1List::pop\_front()
88. {
89. if (!isEmpty())
90. {
91. cur = head;
92. if (get\_next() == nullptr) // delete head
93. {
94. element\* del = head;
95. head = cur = tail = nullptr;
96. delete del;
97. }
98. else
99. {
100. element\* del = head;
101. set\_cur(head->get\_next());// cur = head->next
102. set\_head(cur); // head = cur
103. delete del;
104. }
105. }
106. else
107. throw out\_of\_range("The List is empty");
108. }
109. size\_t L1List::get\_size()
110. {
111. size\_t Lsize = 0;
112. if (!isEmpty())
113. {
114. Lsize = 1;
115. set\_cur(get\_head()); // now = head
116. while (get\_next() != nullptr) // while cur->next exists
117. {
118. set\_cur(get\_next()); // cur = cur->next
119. Lsize++;
120. } // cur = last existing element
121. }
122. return Lsize;
123. }
124. void L1List::insert(string data, size\_t index) // first index = 0
125. {
126. if (index == get\_size())
127. push\_back(data);
128. else if (index == 0)
129. push\_front(data);
130. else if (index > get\_size())
131. throw out\_of\_range("Invalid index");
132. else
133. {
134. set\_cur(get\_head()); // now = head
135. while (index > 1)
136. {
137. index--;
138. set\_cur(get\_next()); // cur = cur->next
139. } // cur is the element before the future new element
140. element\* e = new element;
141. e->set\_data(data);
142. e->set\_next(get\_next()); //e->next = cur->next
143. set\_next(e); // cur->next = e
144. set\_cur(e);
145. }
146. }
147. element\* L1List::at(size\_t index) // first index = 0
148. {
149. if (index >= get\_size())
150. throw out\_of\_range("Invalid index");
151. else if (index == 0)
152. return head;
153. else
154. {
155. cur = head;
156. while (index > 0)
157. {
158. index--;
159. set\_cur(get\_next()); // cur = cur->next
160. } // cur is the element with data index
161. return cur;
162. }
163. }
164. void L1List::remove(size\_t index)
165. {
166. if (index >= get\_size())
167. throw out\_of\_range("Invalid index");
168. else if (index == 0) // delete head
169. pop\_front();
170. else if (index == get\_size() - 1)
171. pop\_back();
172. else
173. {
174. cur = head;
175. while (index > 1)
176. {
177. index--;
178. set\_cur(get\_next()); // cur = cur->next
179. } // cur is the element before the deleting element
180. element\* e = new element;
181. e = get\_next();
182. set\_next(e->get\_next()); //cur->next = cur->next->next
183. if (e == nullptr)
184. tail = cur;
185. delete e;
186. }
187. }
188. void L1List::print\_to\_console()
189. {
190. set\_cur(get\_head()); // now = head
191. while (get\_next() != nullptr)
192. {
193. cout << get\_data() << endl;
194. set\_cur(get\_next()); // cur = cur->next
195. }
196. cout << get\_data() << endl;
197. }
198. void L1List::clear()
199. {
200. if (!isEmpty())
201. {
202. set\_cur(get\_head()); // cur = head
203. while (get\_next() != nullptr) // while next exists
204. {
205. set\_cur(get\_next()); // cur = cur->next
206. delete get\_head();
207. set\_head(get\_cur()); // head = cur
208. } //cur - the last in the list
209. element\* e = get\_head();
210. head = cur = tail = nullptr;
211. delete e;
212. }
213. else
214. throw exception("The List is empty");
215. }
216. void L1List::set(size\_t index, string data) // change data on index element
217. {
218. if (index >= get\_size())
219. throw out\_of\_range("Invalid index");
220. else if (index == 0)
221. head->set\_data(data);
222. else if (index == get\_size() - 1)
223. tail->set\_data(data);
224. else
225. {
226. set\_cur(get\_head()); // now = head
227. while (index > 0)
228. {
229. index--;
230. set\_cur(get\_next()); // cur = cur->next
231. } // cur is the element with data index
232. cur->set\_data(data);
233. }
234. }
235. void L1List::push\_front(L1List\* L) // insertion another list into the begining of data
236. {
237. if (!L->isEmpty())
238. {
239. push\_front(L->get\_head()->get\_data()); // first in this is the same as the first in L now
240. element\* h = head; // now head is a new element because that's how my push\_front works
241. L->set\_cur(L->get\_head());
242. while (L->get\_next() != nullptr) // while cur->next exists
243. {
244. L->set\_cur(L->get\_next());
245. insert(L->get\_cur()->get\_data(), 1);
246. head = cur; // head is an inserted element - this is made for next iteration
247. }
248. set\_head(h);
249. }
250. }
251. L1List::~L1List()
252. {
253. if (!isEmpty())
254. clear();
255. }
256. int main()
257. {
258. L1List a;
259. L1List b;
260. b.push\_back("hey");
261. b.push\_front("hmmm");
262. b.push\_back("have you ever thought that...");
263. a.push\_back("maybe");
264. a.push\_back("today");
265. a.push\_back("is");
266. a.push\_back("a very");
267. a.push\_front(&b);
268. a.push\_back("good");
269. a.push\_back("day?");
270. a.print\_to\_console();
271. a.clear();
272. cout << a.isEmpty() << endl;
273. return 0;
274. }

UnitTestForOneWayLinkedList.cpp:

1. #include "CppUnitTest.h"
2. #include "/Users/varka/source/repos/AlgoLaba1\_OneWayLinkedList/L1List.h"
3. using namespace Microsoft::VisualStudio::CppUnitTestFramework;
4. namespace UnitTestForOneWayLinkedList
5. {
6. TEST\_CLASS(UnitTestForOneWayLinkedList)
7. {
8. public:
9. TEST\_METHOD(IsEmptyListWhenEmpty)
10. {
11. L1List a;
12. Assert::AreEqual(a.isEmpty(), true);
13. }
14. TEST\_METHOD(IsEmptyListWhenNotEmpty)
15. {
16. L1List a;
17. a.push\_back((string)"bla");
18. Assert::AreEqual(a.isEmpty(), false);
19. }
20. TEST\_METHOD(ListSizeZero)
21. {
22. L1List a;
23. Assert::AreEqual(a.get\_size(), (size\_t)0);
24. }
25. TEST\_METHOD(ListSizeNotZero)
26. {
27. L1List a;
28. a.push\_back("bla");
29. a.push\_back("ha");
30. a.push\_front("hehe");
31. Assert::AreEqual(a.get\_size(), (size\_t)3);
32. }
33. TEST\_METHOD(PushBackOneElement)
34. {
35. L1List a;
36. a.push\_back("bla");
37. Assert::AreEqual(a.get\_head()->get\_data(), (string)"bla");
38. }
39. TEST\_METHOD(PushBackElements)
40. {
41. L1List a;
42. a.push\_back("bla");
43. a.push\_back("car");
44. Assert::AreEqual(a.get\_data(), (string)"car");
45. }
46. TEST\_METHOD(PopBackFromEmptyList)
47. {
48. L1List a;
49. try
50. {
51. a.pop\_back();
52. }
53. catch (const exception& message)
54. {
55. Assert::AreEqual(message.what(), "The List is empty");
56. }
57. }
58. TEST\_METHOD(PopBackFromListWithOneElement)
59. {
60. L1List a;
61. a.push\_back("hello");
62. a.pop\_back();
63. Assert::AreEqual(a.isEmpty(), true);
64. }
65. TEST\_METHOD(PopBackFromList)
66. {
67. L1List a;
68. a.push\_back("!!!");
69. a.push\_back("hello");
70. a.push\_back("bro");
71. a.pop\_back();
72. Assert::AreEqual(a.get\_data(), (string)"hello");
73. }
74. TEST\_METHOD(PushFrontInEmptyList)
75. {
76. L1List a;
77. a.push\_front("hey");
78. Assert::AreEqual(a.get\_data(), (string)"hey");
79. Assert::AreEqual(a.get\_next() == nullptr, true);
80. Assert::AreEqual(a.get\_head() == a.get\_tail(), true);
81. }
82. TEST\_METHOD(PushFrontInNotEmptyList)
83. {
84. L1List a;
85. a.push\_front("a good day");
86. a.push\_front("i wish you");
87. a.push\_front("hey");
88. Assert::AreEqual(a.get\_data(), (string)"hey");
89. }
90. TEST\_METHOD(PopFrontFromEmptyList)
91. {
92. L1List a;
93. try
94. {
95. a.pop\_front();
96. }
97. catch (const exception & message)
98. {
99. Assert::AreEqual(message.what(), "The List is empty");
100. }
101. }
102. TEST\_METHOD(PopFrontFromList)
103. {
104. L1List a;
105. a.push\_front("a good day");
106. a.push\_front("i wish you");
107. a.push\_front("hey");
108. a.pop\_front();
109. Assert::AreEqual(a.get\_data(), (string)"i wish you");
110. }
111. TEST\_METHOD(PopFrontFromOneElementList)
112. {
113. L1List a;
114. a.push\_front("hey");
115. a.pop\_front();
116. Assert::AreEqual(a.isEmpty(), true);
117. }
118. TEST\_METHOD(AtInEmptyList)
119. {
120. L1List a;
121. try
122. {
123. a.at(0);
124. }
125. catch (const exception & message)
126. {
127. Assert::AreEqual(message.what(), "Invalid index");
128. }
129. }
130. TEST\_METHOD(AtInListWhenPushFronted)
131. {
132. L1List a;
133. a.push\_front("a good day");
134. a.push\_front("i wish you");
135. a.push\_front("hey");
136. Assert::AreEqual(a.at(2)->get\_data(), (string)"a good day");
137. Assert::AreEqual(a.at(1)->get\_data(), (string)"i wish you");
138. Assert::AreEqual(a.at(0)->get\_data(), (string)"hey");
139. }
140. TEST\_METHOD(AtInListWhenPushBacked)
141. {
142. L1List a;
143. a.push\_back("hey");
144. a.push\_back("i wish you");
145. a.push\_back("a good day");
146. Assert::AreEqual(a.at(2)->get\_data(), (string)"a good day");
147. Assert::AreEqual(a.at(1)->get\_data(), (string)"i wish you");
148. Assert::AreEqual(a.at(0)->get\_data(), (string)"hey");
149. }
150. TEST\_METHOD(AtInListWhenPushed)
151. {
152. L1List a;
153. a.push\_back("i wish you");
154. a.push\_front("hey");
155. a.push\_back("a good day");
156. Assert::AreEqual(a.at(2)->get\_data(), (string)"a good day");
157. Assert::AreEqual(a.at(1)->get\_data(), (string)"i wish you");
158. Assert::AreEqual(a.at(0)->get\_data(), (string)"hey");
159. }
160. TEST\_METHOD(InsertToListBegin)
161. {
162. L1List a;
163. a.push\_front("khe");
164. a.push\_back("is this a coronavirus???");
165. a.push\_back("we are going to die");
166. a.insert("Oh no!", 0);
167. Assert::AreEqual(a.at(0)->get\_data(), (string)"Oh no!");
168. }
169. TEST\_METHOD(InsertToListEnd)
170. {
171. L1List a;
172. a.push\_front("khe");
173. a.push\_back("is this a coronavirus???");
174. a.push\_back("we are going to die");
175. a.insert("Oh no!", 3);
176. Assert::AreEqual(a.at(3)->get\_data(), (string)"Oh no!");
177. }
178. TEST\_METHOD(InsertToListMiddle)
179. {
180. L1List a;
181. a.push\_front("khe");
182. a.push\_back("is this a coronavirus???");
183. a.push\_back("we are going to die");
184. a.insert("Oh no!", 2);
185. Assert::AreEqual(a.at(2)->get\_data(), (string)"Oh no!");
186. }
187. TEST\_METHOD(InsertToListInvalidIndex)
188. {
189. L1List a;
190. a.push\_front("khe");
191. try
192. {
193. a.insert("bla", 2);
194. }
195. catch (const exception & message)
196. {
197. Assert::AreEqual(message.what(), "Invalid index");
198. }
199. }
200. TEST\_METHOD(RemoveFromListInvalidIndex)
201. {
202. L1List a;
203. a.push\_front("khe");
204. try
205. {
206. a.remove(2);
207. }
208. catch (const exception & message)
209. {
210. Assert::AreEqual(message.what(), "Invalid index");
211. }
212. }
213. TEST\_METHOD(RemoveFromOneElementList)
214. {
215. L1List a;
216. a.push\_front("khe");
217. a.remove(0);
218. Assert::AreEqual(a.isEmpty(), true);
219. }
220. TEST\_METHOD(RemoveFromList)
221. {
222. L1List a;
223. a.push\_front("khe");
224. a.push\_back("is this a coronavirus???");
225. a.push\_back("we are going to die");
226. a.insert("Oh no!", 2);
227. a.remove(2);
228. Assert::AreEqual(a.at(2)->get\_data(), (string)"we are going to die");
229. }
230. TEST\_METHOD(RemoveFromListEnd)
231. {
232. L1List a;
233. a.push\_front("khe");
234. a.push\_back("is this a coronavirus???");
235. a.push\_back("we are going to die");
236. a.insert("Oh no!", 2);
237. a.remove(3);
238. Assert::AreEqual(a.at(2) == a.get\_tail(), true);
239. }
240. TEST\_METHOD(RemoveFromListBegin)
241. {
242. L1List a;
243. a.push\_front("khe");
244. a.push\_back("is this a coronavirus???");
245. a.push\_back("we are going to die");
246. a.insert("Oh no!", 2);
247. a.remove(0);
248. Assert::AreEqual(a.at(0)->get\_data(), (string)"is this a coronavirus???");
249. }
250. TEST\_METHOD(SetInListBegin)
251. {
252. L1List a;
253. a.push\_front("khe");
254. a.push\_back("is this a coronavirus???");
255. a.push\_back("we are going to die");
256. a.set(0, "apchi!");
257. Assert::AreEqual(a.at(0)->get\_data(), (string)"apchi!");
258. }
259. TEST\_METHOD(SetInListEnd)
260. {
261. L1List a;
262. a.push\_front("khe");
263. a.push\_back("is this a coronavirus???");
264. a.push\_back("we are going to die");
265. a.set(2, "apchi!");
266. Assert::AreEqual(a.at(2)->get\_data(), (string)"apchi!");
267. }
268. TEST\_METHOD(SetInEmptyList)
269. {
270. L1List a;
271. try
272. {
273. a.set(0, "apchi!");
274. }
275. catch (const exception & message)
276. {
277. Assert::AreEqual(message.what(), "Invalid index");
278. }
279. }
280. TEST\_METHOD(SetInListInvalidIndex)
281. {
282. L1List a;
283. a.push\_back("khe-khe");
284. try
285. {
286. a.set(1, "apchi!");
287. }
288. catch (const exception & message)
289. {
290. Assert::AreEqual(message.what(), "Invalid index");
291. }
292. }
293. TEST\_METHOD(SetInList)
294. {
295. L1List a;
296. a.push\_front("khe");
297. a.push\_back("is this a coronavirus???");
298. a.push\_back("we are going to die");
299. a.set(1, "apchi!");
300. Assert::AreEqual(a.at(1)->get\_data(), (string)"apchi!");
301. }
302. TEST\_METHOD(ClearEmptyList)
303. {
304. L1List a;
305. try
306. {
307. a.clear();
308. }
309. catch (const exception & message)
310. {
311. Assert::AreEqual(message.what(), "The List is empty");
312. }
313. }
314. TEST\_METHOD(ClearList)
315. {
316. L1List a;
317. a.push\_front("khe");
318. a.push\_back("is this a coronavirus???");
319. a.push\_back("we are going to die");
320. a.set(1, "apchi!");
321. a.clear();
322. Assert::AreEqual(a.isEmpty(), true);
323. Assert::AreEqual(a.get\_head() == nullptr, true);
324. Assert::AreEqual(a.get\_tail() == nullptr, true);
325. Assert::AreEqual(a.get\_cur() == nullptr, true);
326. Assert::AreEqual(a.get\_size(), (size\_t)0);
327. }
328. TEST\_METHOD(PushFrontAList)
329. {
330. L1List one, two;
331. one.push\_back("Hello");
332. one.push\_back("my name is Barbara");
333. two.push\_back("I'm to become a programmer");
334. two.push\_back("And this is good");
335. two.push\_back("What do you think?");
336. two.push\_front(&one);
337. Assert::AreEqual(two.at(1)->get\_data(), (string)"my name is Barbara");
338. Assert::AreEqual(one.get\_size(), (size\_t)2);
339. Assert::AreEqual(two.get\_size(), (size\_t)5);
340. }
341. TEST\_METHOD(PushFrontAListChangeOne)
342. {
343. L1List one, two;
344. one.push\_back("Hello");
345. one.push\_back("my name is Barbara");
346. two.push\_back("I'm to become a programmer");
347. two.push\_back("And this is good");
348. two.push\_back("What do you think?");
349. two.push\_front(&one);
350. one.pop\_front();
351. Assert::AreEqual(two.at(0)->get\_data(), (string)"Hello");
352. Assert::AreEqual(one.get\_size(), (size\_t)1);
353. Assert::AreEqual(two.get\_size(), (size\_t)5);
354. }
355. TEST\_METHOD(PushFrontAListClearOne)
356. {
357. L1List one, two;
358. one.push\_back("Hello");
359. one.push\_back("my name is Barbara");
360. two.push\_back("I'm to become a programmer");
361. two.push\_back("And this is good");
362. two.push\_back("What do you think?");
363. two.push\_front(&one);
364. one.clear();
365. Assert::AreEqual(two.at(0)->get\_data(), (string)"Hello");
366. Assert::AreEqual(one.get\_size(), (size\_t)0);
367. Assert::AreEqual(two.get\_size(), (size\_t)5);
368. }
369. TEST\_METHOD(PushFrontAListClearTwo)
370. {
371. L1List one, two;
372. one.push\_back("Hello");
373. one.push\_back("my name is Barbara");
374. two.push\_back("I'm to become a programmer");
375. two.push\_back("And this is good");
376. two.push\_back("What do you think?");
377. two.push\_front(&one);
378. two.clear();
379. Assert::AreEqual(two.isEmpty(), true);
380. Assert::AreEqual(one.get\_size(), (size\_t)2);
381. Assert::AreEqual(one.at(0)->get\_data(), (string)"Hello");
382. }
383. TEST\_METHOD(PushFrontAListChangeTwo)
384. {
385. L1List one, two;
386. one.push\_back("Hello");
387. one.push\_back("my name is Barbara");
388. two.push\_back("I'm to become a programmer");
389. two.push\_back("And this is good");
390. two.push\_back("What do you think?");
391. two.push\_front(&one);
392. two.set(1, "I'm Bond. James Bond.");
393. Assert::AreEqual(one.at(1)->get\_data(), (string)"my name is Barbara");
394. Assert::AreEqual(two.at(1)->get\_data(), (string)"I'm Bond. James Bond.");
395. }
396. TEST\_METHOD(PushFrontAListClearBoth)
397. {
398. L1List one, two;
399. one.push\_back("Hello");
400. one.push\_back("my name is Barbara");
401. two.push\_back("I'm to become a programmer");
402. two.push\_back("And this is good");
403. two.push\_back("What do you think?");
404. two.push\_front(&one);
405. one.clear();
406. two.clear();
407. Assert::AreEqual(two.isEmpty(), true);
408. Assert::AreEqual(one.isEmpty(), true);
409. }
410. TEST\_METHOD(PushFrontAListSecondEmpty)
411. {
412. L1List a, b;
413. a.push\_back("maybe"); // 0
414. a.push\_back("today"); // 1
415. a.push\_back("is"); // 2
416. a.push\_back("a very"); // 3
417. a.push\_back("good"); // 4
418. a.push\_back("day?"); // 5
419. b.push\_front(&a);
420. Assert::AreEqual(b.at(0)->get\_data(), (string)"maybe");
421. Assert::AreEqual(b.at(1)->get\_data(), (string)"today");
422. Assert::AreEqual(b.at(2)->get\_data(), (string)"is");
423. Assert::AreEqual(b.at(3)->get\_data(), (string)"a very");
424. Assert::AreEqual(b.at(4)->get\_data(), (string)"good");
425. Assert::AreEqual(b.at(5)->get\_data(), (string)"day?");
426. Assert::AreEqual(b.get\_size(), (size\_t)6);
427. }
428. };
429. }