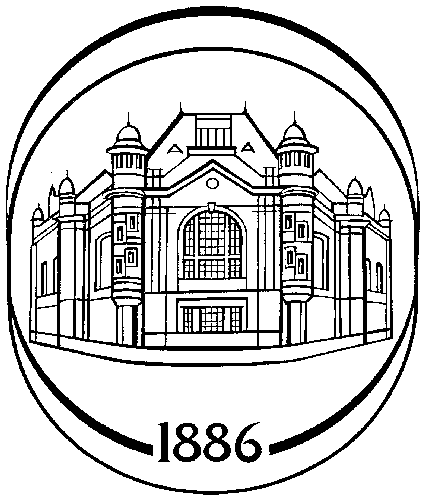
МИНОБРНАУКИ РОССИИ

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 main\_List;
4. L1List to\_begining\_List;
5. to\_begining\_List.push\_back("hey");
6. to\_begining\_List.push\_front("hmmm");
7. to\_begining\_List.push\_back("have you ever thought that...");
8. main\_List.push\_back("maybe");
9. main\_List.push\_back("today");
10. main\_List.push\_back("is");
11. main\_List.push\_back("a very");
12. main\_List.push\_front(&to\_begining\_List);
13. main\_List.push\_back("good");
14. main\_List.push\_back("day?");
15. main\_List.print\_to\_console();
16. main\_List.clear();
17. cout << main\_List.isEmpty() << endl;
18. return 0;
19. }

Результат, выведенный в консоль:

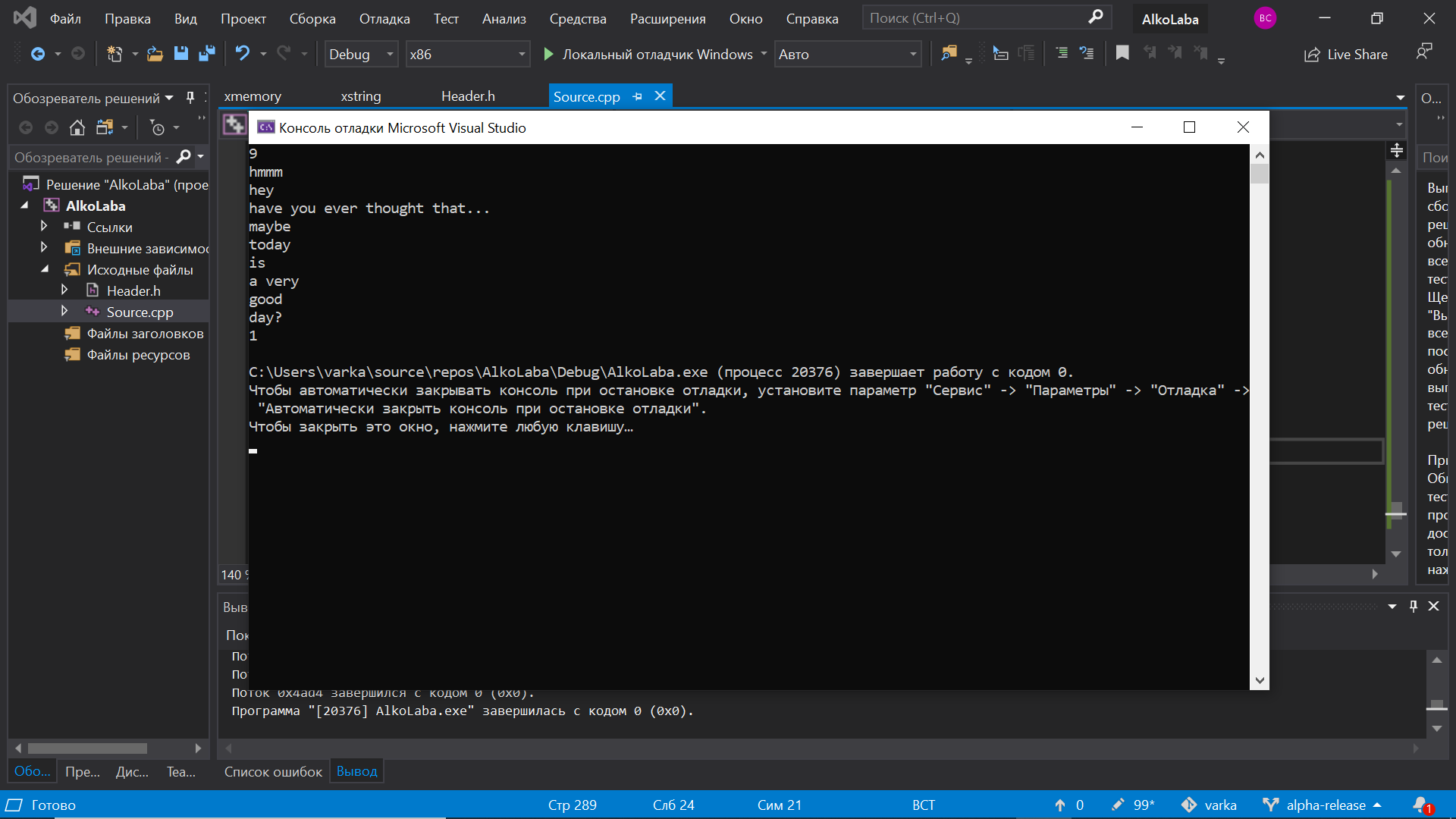


Рис. 1.1 – Пример работы программы

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

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

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