Выполнил

студент группы КТбо1-2 А. С. Мумладзе

Принял

доцент кафедры САиТ В. С. Лапшин

Таганрог 2024

Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное

учреждение высшего образования

«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

(ФГАОУ ВО «ЮФУ»)

Институт компьютерных технологий и информационной безопасности

Кафедра системного анализа и телекоммуникаций

ЛАБОРАТОРНАЯ РАБОТА №2

по дисциплине «Основы алгоритмизации и программирования»

**«ДИНАМИЧЕСКОЕ РАСПРЕДЕЛЕНИЕ ПАМЯТИ, СПИСКИ»**

Вариант 9

# Содержание

Содержание 2

Техническое задание 3

Цель задания 3

Задача 3

Ход работы 4

Алгоритм 4

Написание программы 4

Пример работы программы 6

Вывод 7

Листинг 8

main.cpp 8

common.h 15

# Техническое задание

## Цель задания

Целью данной лабораторной работы является освоение студентами способов представления и обработки данных в виде списков. Для успешного выполнения лабораторной работы необходимо владеть теоретическими сведениями по таким понятиям, как списки, указатели, способы выделения памяти, функции, отвечающие за выделение памяти, функции malloc() calloc() free(), структуры данных, информационная и адресная составляющие списка.

## Задача

Задание. Написать программы для работы с односвязным и двусвязным списками в соответствии с выданным вариантом задания. Предусмотреть в программах следующие функции:

1. Включить новый элемент в конец списка.
2. Включить новый элемент на заданное пользователем место списка.
3. Включить новый элемент после элемента с заданной информационной частью.
4. Включить новый элемент перед элементом с заданной информационной частью.
5. Включить новый элемент в середину списка.
6. Исключить элемент из середины списка.
7. Исключить элемент с заданной информационной частью.
8. Исключить элемент из конца списка.
9. Исключить элемент из заданного пользователем места списка.
10. Исключить элемент из головы списка.

Задание моего варианта: обработать структуру данных, содержащую в себе адресную книгу.

# Ход работы

## Алгоритм

Алгоритм моей программы следующий:

1. Открывается и читается файл list.txt, содержащий в себе список адресов. Если файл не будет найден, пользователю предложат создать новый.
2. Каждый прочитанный адрес заносится в структуру address, и связывается с предыдущим. Структура list хранит указатели на начало списка, конец, его длину и статус (если программа завершается, статус is\_work меняется на false)
3. После полного прочтения файла в двусвязный список программа ожидает действия пользователя. Пользователю печатается полный список возможных команд, от прочтения списка и завершения программы с записью текущего состояния, до добавления и удаления нужных записей в нужном месте списка.
4. После проведения необходимых манипуляций и выбора «завершение программы», она предложит сохранить измененный список в файл. Если пользователь сохранится, то файл будет перезаписан обновленным списком.

## Написание программы

Каждая запись списка – это выделенное calloc() место ровно под одну структуру address. Благодаря этому новые узлы легко связать со старыми при помощи указателей, а при удалении узлов не происходит утечка памяти, поскольку память под удаленный узел освобождается при помощи free().

Функции создания узла и удаления не зависят напрямую от предназначения функции, в которой эти операции используются. Они лишь принимают указатель на нужный узел, и уже после этого работают с ним. Благодаря этому функционал программы можно расширить.

Название каждой функции лежит в enum{} списке, благодаря чему пользователю предлагается выбрать номер функции, а не писать ее название, и для разработчика сразу очевидно, какой «номер функции» вызывает какую функцию.

Сравнение записей в узлах достаточно гибкое благодаря посимвольному сравнению. Программа учитывает, что пользователь мог пропустить одну букву, или перепутать их местами, благодаря чему итогом поиска может стать как точное совпадение, так и наиболее близкая запись (и пользователю дается выбор, подтвердить найденную запись, или потом снова попытаться ввести).

## Пример работы программы

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

# Вывод

Итогом работы стала программа, работающая с двусвязным списком адресов. Она динамически может создавать новые узлы (при чтении файла или по запросу пользователя), а также удалять их без утечки памяти, сравнивать данных в узлах, и записывать список в файл.

Я не реализовал работу с односвязным списком, поскольку логика их работы практически идентична, но с ней мне пришлось бы заново копировать все функции, но удалять «связь с прошлым узлом», единственное отличие двусвязного списка от односвязного.

В ходе работы я ознакомился с функциями calloc() и free(), воспользовался структурами и нумерованным списком, а также ознакомился с возможностями использования указателей в подобных задачах.

# Листинг

## main.cpp

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <unistd.h>
5. #include "common.h"
6. int main() {
7. FILE\* list\_file = fopen("list.txt", "r");
8. list list\_address = create\_list(list\_file);
9. fclose(list\_file);
10. ListFunc next\_act;
11. print\_funcs\_list();
12. while (true) {
13. printf("\nChoose action: ");
14. scanf("%d", (int\*) &next\_act);
15. switch (next\_act) {
16. case STOP\_PROGRAM:
17. stop\_program(&list\_address);
18. exit(0);
19. case PRINT\_LIST:
20. print\_list(&list\_address);
21. break;
22. case ADD\_TO\_HEAD:
23. add\_to\_head(&list\_address);
24. break;
25. case ADD\_TO\_TAIL:
26. add\_to\_tail(&list\_address);
27. break;
28. case ADD\_NEXT\_MATCH:
29. add\_next\_match(&list\_address);
30. break;
31. case ADD\_PRIOR\_MATCH:
32. add\_prior\_match(&list\_address);
33. break;
34. case ADD\_TO\_MIDDLE:
35. add\_to\_middle(&list\_address);
36. break;
37. case REMOVE\_FROM\_HEAD:
38. remove\_from\_head(&list\_address);
39. break;
40. case REMOVE\_FROM\_TAIL:
41. remove\_from\_tail(&list\_address);
42. break;
43. case REMOVE\_FROM\_MIDDLE:
44. remove\_from\_middle(&list\_address);
45. break;
46. case REMOVE\_MATCH:
47. remove\_match(&list\_address);
48. break;
49. case REMOVE\_BY\_INDEX:
50. remove\_by\_index(&list\_address);
51. break;
52. default:
53. printf("An unexpected entry has been made. To terminate the program, enter '0'.");
54. break;
55. }
56. }
57. }
58. list create\_list(FILE\* list\_file) {
59. if (!list\_file) {
60. printf("Warning! File 'list.txt' is not exist. An empty list will be created\n");
61. return {nullptr, nullptr, 0, true};
62. }
63. else {
64. size\_t list\_len = 0;
65. fscanf(list\_file, "%lld\n", &list\_len);
66. address\* head = nullptr;
67. address\* tail = nullptr;
68. for (size\_t i = 0; i < list\_len; i++) {
69. address\* new\_node = (address\*) calloc(1, sizeof(address));
70. fscanf(list\_file, "%[^;]; %[^;]; %[^;]; %s\n", new\_node->country,
71. new\_node->city, new\_node->street, new\_node->house\_num);
72. if (head == nullptr) {
73. head = new\_node;
74. tail = new\_node;
75. }
76. else {
77. tail->next = new\_node;
78. new\_node->prior = tail;
79. tail = new\_node;
80. }
81. }
82. return {head, tail, list\_len, true};
83. }
84. }
85. void print\_funcs\_list() {
86. printf("List of actions:\nSTOP\_PROGRAM - 0\nPRINT\_LIST - 1\nADD\_TO\_HEAD - 2\nADD\_TO\_TAIL - 3\nADD\_NEXT\_MATCH - 4\nADD\_PRIOR\_MATCH - 5\nADD\_TO\_MIDDLE - 6\nREMOVE\_FROM\_HEAD - 7\nREMOVE\_FROM\_TAIL - 8\nREMOVE\_FROM\_MIDDLE - 9\nREMOVE\_MATCH - 10\nREMOVE\_BY\_INDEX - 11\n");
87. }
88. void print\_list(list\* list\_address) {
89. address\* ptr = list\_address->head;
90. int index = 0;
91. if (ptr == nullptr) {
92. printf("List is empty\n");
93. }
94. else {
95. printf("Len of list: %llu\n", list\_address->len);
96. while (ptr) {
97. printf("%d: %s; %s; %s; %s\n", index, ptr->country, ptr->city, ptr->street, ptr->house\_num);
98. ptr = ptr->next;
99. index++;
100. }
101. }
102. }
103. address\* get\_user\_node() {
104. address\* node = (address\*) calloc(1, sizeof(address));
105. scanf("\n%[^;]; %[^;]; %[^;]; %[^\n]", node->country, node->city, node->street, node->house\_num);
106. while (getchar() != '\n');
107. return node;
108. }
109. void insert\_node(list\* list\_address, address\* prior, address\* next) {
110. printf("Enter new node:\n");
111. address\* new\_node = get\_user\_node();
112. if (prior == nullptr && next == nullptr) {
113. list\_address->head = new\_node;
114. list\_address->tail = new\_node;
115. }
116. else if (prior == nullptr) {
117. list\_address->head->prior = new\_node;
118. new\_node->next = list\_address->head;
119. list\_address->head = new\_node;
120. }
121. else if (next == nullptr) {
122. list\_address->tail->next = new\_node;
123. new\_node->prior = list\_address->tail;
124. list\_address->tail = new\_node;
125. }
126. else {
127. new\_node->prior = prior;
128. prior->next = new\_node;
129. new\_node->next = next;
130. next->prior = new\_node;
131. }
132. list\_address->len = list\_address->len + 1;
133. }
134. void remove\_node(list\* list\_address, address\* curr) {
135. if (curr == nullptr) {
136. printf("List is empty. Operation cannot be performed\n");
137. return;
138. }
139. else if (curr->prior == nullptr && curr->next == nullptr) {
140. if (list\_address->is\_work) {
141. printf("Last item in the list is deleted\n");
142. }
143. list\_address->head = nullptr;
144. list\_address->tail = nullptr;
145. }
146. else if (curr->prior == nullptr) {
147. curr->next->prior = nullptr;
148. list\_address->head = curr->next;
149. }
150. else if (curr->next == nullptr) {
151. curr->prior->next = nullptr;
152. list\_address->tail = curr->prior;
153. }
154. else {
155. curr->next->prior = curr->prior;
156. curr->prior->next = curr->next;
157. }
158. free(curr);
159. list\_address->len = list\_address->len - 1;
160. }
161. address\* search\_node(list\* list\_address) {
162. printf("Enter in the string you're looking for:\n");
163. address\* searched\_node = get\_user\_node();
165. size\_t min\_diff = 140;
166. address\* matched;
167. size\_t curr\_diff = 0;
168. address\* ptr = list\_address->head;
170. while (ptr) {
171. curr\_diff += compare\_strings(ptr->country, searched\_node->country);
172. curr\_diff += compare\_strings(ptr->city, searched\_node->city);
173. curr\_diff += compare\_strings(ptr->street, searched\_node->street);
174. curr\_diff += compare\_strings(ptr->house\_num, searched\_node->house\_num);
175. if (curr\_diff == 0) {
176. min\_diff = 0;
177. matched = ptr;
178. break;
179. }
180. else if (curr\_diff < min\_diff) {
181. matched = ptr;
182. min\_diff = curr\_diff;
183. }
184. curr\_diff = 0;
185. ptr = ptr->next;
186. }
187. free(searched\_node);
188. if (min\_diff == 0) {
189. return matched;
190. }
191. else if (min\_diff <= 18) {
192. printf("\nNo exact match was found. The closest match was found:\n%s; %s; %s; %s\n"
193. "Enter 1 if it is a match and 0 if it is not: ", matched->country, matched->city,
194. matched->street, matched->house\_num);
195. int conitnue = 0;
196. scanf("%d", &conitnue);
197. if (conitnue) {
198. return matched;
199. }
200. else {
201. return nullptr;
202. }
203. }
204. else {
205. printf("\nNo exact match was found. The operation was canceled\n");
206. return nullptr;
207. }
208. }
209. size\_t compare\_strings(char\* searched, char\* curr) {
210. size\_t diff = 0;
211. while (\*searched != '\0') {
212. if (\*searched == \*curr) {
213. searched++;
214. curr++;
215. }
216. else if (\*searched != \*curr && \*searched == \*(curr + 1) ) {
217. diff++;
218. curr++;
219. }
220. else if (\*searched != \*curr && \*(searched + 1) == \*curr) {
221. diff++;
222. searched++;
223. }
224. else {
225. diff++;
226. searched++;
227. curr++;
228. }
229. }
230. return diff;
231. }
232. void stop\_program(list\* list\_address) {
233. list\_address->is\_work = false;
234. printf("Do you want to update 'list.txt' with changed list? Input 1 if yes, and 0 if false - ");
235. int is\_save\_data = 0;
236. scanf("%d", &is\_save\_data);
237. if (is\_save\_data == 1) {
238. char\* output\_string = (char\*) calloc(list\_address->len \* len\_of\_node, sizeof(char));
239. address\* ptr = list\_address->head;
240. char\* temp = (char\*) calloc(10, sizeof(char));
241. sprintf(temp, "%llu\n", list\_address->len);
242. strcat(output\_string, temp);
243. free(temp);
244. while (ptr) {
245. char\* node = (char\*) calloc(len\_of\_node, sizeof(char));
246. sprintf(node, "%s; %s; %s; %s\n", ptr->country, ptr->city, ptr->street, ptr->house\_num);
247. strcat(output\_string, node);
248. free(node);
249. ptr = ptr->next;
250. remove\_from\_head(list\_address);
251. }
252. FILE\* list\_file = fopen("list.txt", "w");
253. fputs(output\_string, list\_file);
254. fclose(list\_file);
255. free(output\_string);
256. printf("The program is corrently terminated. All new data is written to the file\n");
257. }
258. else {
259. while (list\_address->head != nullptr) {
260. remove\_from\_head(list\_address);
261. }
262. printf("The program is corrently terminated\n");
263. }
264. sleep(10);
265. }
266. void add\_to\_head(list\* list\_address) {
267. insert\_node(list\_address, nullptr, list\_address->head);
268. }
269. void add\_to\_tail(list\* list\_address) {
270. insert\_node(list\_address, list\_address->tail, nullptr);
271. }
272. void add\_next\_match(list\* list\_address) {
273. address\* searched\_node = search\_node(list\_address);
274. if (searched\_node != nullptr) {
275. insert\_node(list\_address, searched\_node, searched\_node->next);
276. }
277. }
278. void add\_prior\_match(list\* list\_address) {
279. address\* searched\_node = search\_node(list\_address);
280. if (searched\_node != nullptr) {
281. insert\_node(list\_address, searched\_node->prior, searched\_node);
282. }
283. }
284. void add\_to\_middle(list\* list\_address) {
285. if (list\_address->len == 0) {
286. printf("List is empty. Add node into head\n");
287. add\_to\_head(list\_address);
288. }
289. else if (list\_address->len == 1) {
290. add\_to\_tail(list\_address);
291. }
292. else {
293. size\_t index = list\_address->len / 2;
294. address\* ptr = list\_address->head;
295. for (size\_t i = 0; i < index; i++) {
296. ptr = ptr->next;
297. }
298. insert\_node(list\_address, ptr->prior, ptr);
299. }
300. }
301. void remove\_from\_head(list\* list\_address) {
302. remove\_node(list\_address, list\_address->head);
303. }
304. void remove\_from\_tail(list\* list\_address) {
305. remove\_node(list\_address, list\_address->tail);
306. }
307. void remove\_from\_middle(list\* list\_address) {
308. if (list\_address->len == 0) {
309. remove\_node(list\_address, nullptr);
310. }
311. else if (list\_address->len == 1) {
312. remove\_node(list\_address, list\_address->head);
313. }
314. else {
315. size\_t index = list\_address->len / 2;
316. address\* ptr = list\_address->head;
317. for (size\_t i = 0; i < index; i++) {
318. ptr = ptr->next;
319. }
320. remove\_node(list\_address, ptr);
321. }
323. }
324. void remove\_match(list\* list\_address) {
325. address\* searched\_node = search\_node(list\_address);
326. if (searched\_node != nullptr) {
327. remove\_node(list\_address, searched\_node);
328. }
329. }
330. void remove\_by\_index(list\* list\_address) {
331. printf("Enter index of node: ");
332. size\_t index = 0;
333. scanf("%llu", &index);
334. address\* ptr = list\_address->head;
335. for (size\_t i = 0; i < index; i++) {
336. ptr = ptr->next;
337. }
338. remove\_node(list\_address, ptr);
339. }

## common.h

1. #pragma once
2. #include <stdio.h>
3. #include <stdlib.h>
4. int len\_of\_node = 150;
5. struct address {
6. char country[40] = "";
7. char city[40] = "";
8. char street[50] = "";
9. char house\_num[10] = "";
10. address\* next = nullptr;
11. address\* prior = nullptr;
12. };
13. struct list {
14. address\* head = nullptr;
15. address\* tail = nullptr;
16. size\_t len = 0;
17. bool is\_work = true;
18. };
19. enum ListFunc {
20. STOP\_PROGRAM = 0,
21. PRINT\_LIST = 1,
22. ADD\_TO\_HEAD = 2,
23. ADD\_TO\_TAIL = 3,
24. ADD\_NEXT\_MATCH = 4,
25. ADD\_PRIOR\_MATCH = 5,
26. ADD\_TO\_MIDDLE = 6,
27. REMOVE\_FROM\_HEAD = 7,
28. REMOVE\_FROM\_TAIL = 8,
29. REMOVE\_FROM\_MIDDLE = 9,
30. REMOVE\_MATCH = 10,
31. REMOVE\_BY\_INDEX = 11,
32. };
33. list create\_list(FILE\* list\_file);
34. void print\_funcs\_list();
35. void print\_list(list\* list\_address);
36. address\* get\_user\_node();
37. void insert\_node(list\* list\_address, address\* prior, address\* next);
38. void remove\_node(list\* list\_address, address\* curr);
39. address\* search\_node(list\* list\_address);
40. size\_t compare\_strings(char\* searched, char\* curr);
41. void stop\_program(list\* list\_address);
42. void add\_to\_head(list\* list\_address);
43. void add\_to\_tail(list\* list\_address);
44. void add\_next\_match(list\* list\_address);
45. void add\_prior\_match(list\* list\_address);
46. void add\_to\_middle(list\* list\_address);
47. void remove\_from\_head(list\* list\_address);
48. void remove\_from\_tail(list\* list\_address);
49. void remove\_from\_middle(list\* list\_address);
50. void remove\_match(list\* list\_address);
51. void remove\_by\_index(list\* list\_address);