**Министерство науки и высшего образования Российской Федерации** ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
 **НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

**Факультет безопасности информационных технологий**

**Дисциплина:**

«Алгоритмы и структуры данных»

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 4**

«Реализация алгоритма шейкерной сортировки с использованием дека на основе связного списка.»

**Выполнил:**

Арендаренко М.М., студент группы N3247

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

(подпись)

**Проверил:**

Ерофеев С. А.

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

(подпись)

Санкт-Петербург

2024 г.

Содержание

1.Формулировка задания

2.Описание использованных типов данных

3.Описание функционала программы

4.Блок-схема алгоритма

5.Исходные коды разработанных программ

6.Результаты тестирования

7.Заключение

1. Формулировка индивидуального задания

Задача работы: разработать программу, которая будет выполнять шейкерную сортировку вещественных чисел с использованием дека на основе связного списка. Результат сортировки нужно сохранить в красно-чёрное дерево. Программа должна быть написана на одном из языков программирования.

Для выполнения задачи необходимо:

* Написать структуру «узла» дека;
* Написать структуру дека;
* Реализовать перечисление для красно-черного дерева
* Реализовать структуру «узла» красно-черного дерева
* Реализовать структуру красно-черного дерева
* Реализовать алгоритм шейкерной сортировки
* Реализовать функцию вывода элементов дека
* Реализовать функцию освобождения памяти , занятой деком
* Реализовать функцию вставки элемента в красно-чёрное дерево;
* Реализовать функцию вывода элементов красно-черного дерева
* Реализовать функцию освобождения памяти , занятой красно-черным деревом

Для реализации был выбран язык программирования C.

2. Описание использованных типов данных

Число с плавающей точкой одинарной точности — float, диапазон от -3,4 \* 10^38 до 3,4 \* 10^38 , (спецификатор формата: %f).

3. Описание функционала программы

В этой работе мы будем использовать двусвязный список для реализации дека. Нам нужно создать структуру, которая будет представлять собой «узел» дека. Этот узел будет содержать значение и указатели на следующий и предыдущий элементы. Затем мы используем эту структуру для создания дека. Также нам нужно реализовать структуры для определения красно-черного дерева.

Реализованные структуры для дека:

Структура Node имеет три поля: *data* — хранит считанное число; next — указатель на следующий узел в списке; prev — указатель на предыдущий узел в списке.

Структура Deque содержит два поля: head — указатель на первый узел в очереди; tail — указатель на последний узел в очереди.

Реализованные структуры для красно-черного дерева:

Перечисление Colour имеет два поля: RED –константа, которая используется для определения красного узла дерева, BLACK–константа, которая используется для определения чёрного узла дерева.

Структура RBNode имеет пять полей: *data* – хранит считанное число, *left* – указатель на левое поддерево, *right* – указатель на правое поддерево, *parent* – указатель на родительский узел, *colour* – цвет узла

Структура RBTree содержит одно поле *root –* указатель на корень красно-черного дерева.

В этой программе реализованы различные функции, каждая из которых выполняет свою уникальную задачу:

**Функция CreateDeque.**

Инициализирует пустой дек, устанавливая указатели *head* и *tail* в NULL

**Функция PushBack.**

Эта функция добавляет новый элемент в конец дека. Она вызывает функцию выделения памяти под новый узел и проверяет успешность выделения. Затем она присваивает данные новому узлу и инициализирует его указатели.

**Функция PushFront.**

Эта функция добавляет новый элемент в начало дека. Она вызывает функцию выделения памяти под новый узел и проверяет успешность выделения. Затем она присваивает данные новому узлу и инициализирует его указатели.

**Функция Empty.**

Эта функция проверяет, пуст ли дек. Возвращает 1 (истина), если указатель head равен NULL, что означает, что дек пуст. Иначе возвращает 0 (ложь).

**Функция PopBack**

Эта функция удаляет элемент из конца дека. Если дек содержит только один элемент, она освобождает память для этого элемента и устанавливает указатели на голову и хвост в *NULL*. В противном случае она перемещает указатель хвоста на предыдущий элемент, освобождает память для старого хвоста иинициализирует указатель *next* нового хвоста в *NULL*.

**Функция PopFront**

Эта функция удаляет элемент из начала дека. Если дек содержит только один элемент, она освобождает память для этого элемента и устанавливает указатели на голову и хвост в *NULL*. В противном случае она перемещает указатель головы на следующий элемент, освобождает память для старой головы и инициализирует указатель *prev* новой головы в *NULL*.

**Функция Clear**

Очищает дек, удаляя все элементы. Пока дек не пуст (используется функция Empty для проверки), удаляются элементы с конца (используется функция PopBack), освобождая память один за другим.

**Функция PrintDeque**

Печатает все элементы дека от головы до хвоста. Проходит по всем узлам, начиная с головы, и выводит значение данных каждого узла с точностью до трех знаков после запятой.

**Функция shaker\_sort**

Эта функция реализует шейкерную сортировку дека на базе двусвязного списка.Она проходит по деку сначала слева направо, меняя местами соседние элементы, если они стоят в неправильном порядке. Затем она проходит справа налево, повторяя процесс. Этот цикл продолжается до тех пор, пока не будет произведено ни одного обмена за проход.

**Функция CreateRB**

Эта функция создает красно-чёрное дерево. Она инициализирует корень дерева как NULL, что означает, что дерево изначально пустое.

**Функция GetColour**

Эта функция возвращает цвет заданного узла. Если узел равен *NULL*, он считается черным, что соответствует свойству красно-черного дерева, согласно которому все листья (NULL-узлы) считаются черными.

**Функция Connect**

Эта функция устанавливает связь между родительским и дочерним узлом. Если дочерний узел не равен NULL, она устанавливает указатель на родительский узел.

**Функция LeftTurn**

Эта функция выполняет левый поворот вокруг заданного узла. Левый поворот переставляет указатели таким образом, что правый ребенок становится новым корнем поддерева, а текущий узел становится левым ребенком нового корня. Это необходимо для балансировки дерева после вставки или удаления элементов.

**Функция RightTurn**

Эта функция выполняет правый поворот вокруг заданного узла. Правый поворот переставляет указатели таким образом, что левый ребенок становится новым корнем поддерева, а текущий узел становится правым ребенком нового корня. Это также необходимо для балансировки дерева после вставки или удаления элементов.

**Функция InsertRB**

Эта функция обеспечивает корректную вставку нового элемента в красно-черное дерево с последующей балансировкой для сохранения его свойств.

Свойства красно-черного дерева:

1. Узел может быть либо красным, либо чёрным и имеет двух потомков;
2. Корень всегда чёрный.
3. Все листья, не содержащие данных — чёрные.
4. Оба потомка каждого красного узла — чёрные.
5. Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

**Функция ClearRBRec**

Рекурсивно освобождает память всех узлов дерева. Если текущий узел не NULL, рекурсивно очищаются его левые и правые поддеревья, затем освобождается память самого узла

**Функция ClearRB**

Эта функция является точкой входа для очистки всего дерева.Вызывает ClearRBRec, начиная с корня дерева.

**Функция PrintLevelRec**

Эта функция рекурсивно выводит все узлы красно-черного дерева на консоль. Она принимает указатель на узел и текущий уровень в дереве в качестве параметров. Если узел равен NULL, функция просто возвращается. В противном случае она сначала рекурсивно вызывает себя для правого поддеревья с увеличенным уровнем. Затем она печатает отступы в зависимости от уровня узла и данные узла. После этого она печатает цвет узла (красный или черный). Наконец, она рекурсивно вызывает себя для левого поддеревья с увеличенным уровнем.

**Функция PrintRBTree**

Эта функция выводит все узлы красно-черного дерева на консоль, используя рекурсивную функцию PrintLevelRec.Она является точкой входа для печати всего дерева.

**Главная функция main.**

Эта функция запрашивает у пользователя количество элементов, которые будут добавлены в дек. Если пользователь вводит отрицательное число, ноль или другой некорректный вариант, выводится сообщение об ошибке, и пользователю предлагается ввести число еще раз.

После он вводит заданное ранее количество элементов для заполнения дека. Если пользователь снова введет некорректное значение, появится сообщение об ошибке, и нужно будет ввести число еще раз.

Затем программа выводит элементы дека на экран. Далее она сортирует элементы с помощью шейкерной сортировки и выводит отсортированные элементы на экран. Также проверяется, достигнут ли конец файла (EOF) во время чтения стандартного ввода. Если это так, программа выводит сообщение и завершается с ошибкой.Затем создаётся красно-чёрное дерево, в которое заносятся числа из дека. После этого результат выводится на экран.Выделенная память освобождается перед завершением программы.

4.Блок-схема алгоритма

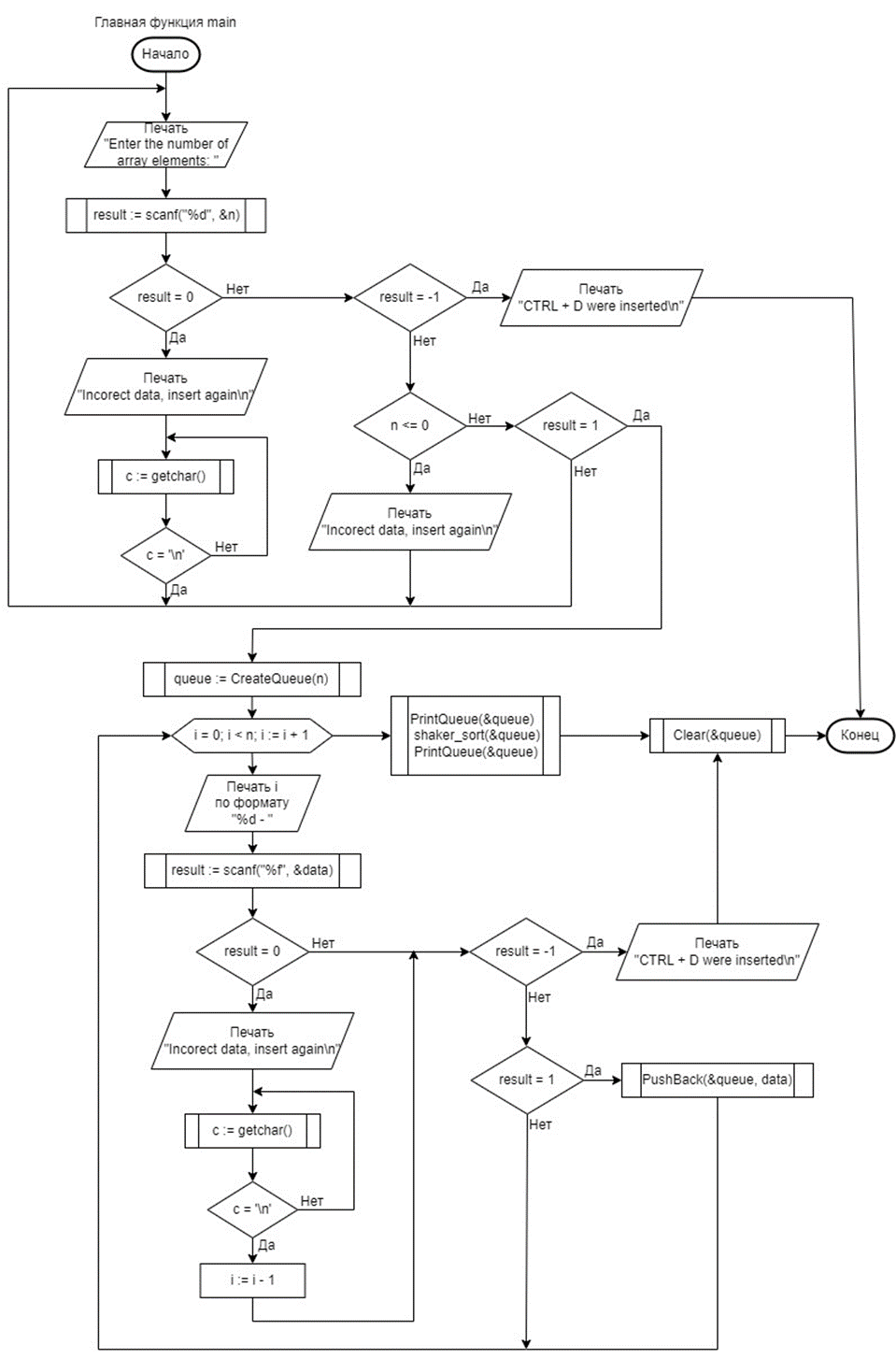


Рисунок 1 – Блок-схема функции main

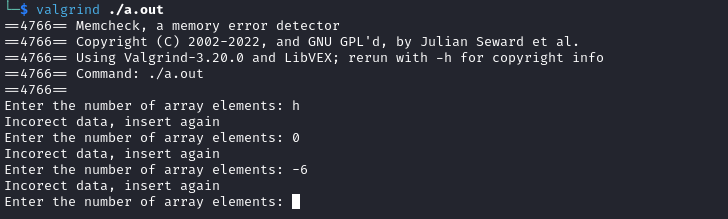
5.Исходные коды разработанных программ

1. #include<stdio.h>
2. #include<stdio.h>
3. #include<stdlib.h>
4. typedef struct node {
5. float data;
6. struct node\* prev;
7. struct node\* next;
8. } Node;
9. typedef struct deque {
10. Node\* head;
11. Node\* tail;
12. } Deque;
13. //Создаём дек
14. Deque CreateDeque() {
15. Deque deque;
16. deque.head = NULL;
17. deque.tail = NULL;
18. return deque;
19. }
20. // Добавление элемента в конец дека
21. void PushBack(Deque\* deque, float data) {
22. Node\* node = malloc(sizeof(Node));
23. if(node == NULL) {
24. return;
25. }
26. node->data = data;
27. node->next = NULL; // Устанавливаем указатель next нового узла в NULL, так как это последний элемент
28. node->prev = deque->tail; // Устанавливаем указатель prev нового узла на текущий хвост дека
29. if(deque->head == NULL) {
30. deque->head = node; // Если дек пуст, новый узел становится головой дека
31. } else {
32. deque->tail->next = node; // Иначе, текущий хвост указывает на новый узел
33. }
34. deque->tail = node; // Новый узел становится хвостом дека
35. }
36. // Добавление элемента в начало дека
37. void PushFront(Deque\* deque, float data) {
38. Node\* node = malloc(sizeof(Node));
39. if(node == NULL) {
40. return;
41. }
42. node->data = data;
43. node->next = deque->head; // Устанавливаем указатель next нового узла на текущую голову дека
44. node->prev = NULL;
45. if(deque->head == NULL) {
46. deque->tail = node; //Если дек пуст, новый узел становится хвостом дека
47. } else {
48. deque->head->prev = node;
49. }
50. deque->tail = node; // Новый узел становится головой дека
51. }
52. //Проверка дека на пустоту
53. int Empty(Deque\* deque) {
54. return deque->head == NULL;
55. }
56. // Удаление элемента из конца дека
57. void PopBack(Deque\* deque) {
58. if(deque->head == deque->tail) { // Проверяем, содержит ли дек только один элемент
59. free(deque->head);
60. deque->head = NULL;
61. deque->tail = NULL;
62. return;
63. }
64. deque->tail = deque->tail->prev; // Перемещаем указатель хвоста на предыдущий элемент
65. free(deque->tail->next); // Освобождаем память для старого хвоста
66. deque->tail->next = NULL;
67. }
68. // Удаление элемента из конца дека
69. void PopFront(Deque\* deque) {
70. if(deque->head == deque->tail) {
71. free(deque->head);
72. deque->head = NULL;
73. deque->tail = NULL;
74. return;
75. }
76. deque->head = deque->head->next; // Перемещаем указатель головы на следующий элемент
77. free(deque->head->prev); // Освобождаем память для старой головы
78. deque->head->prev = NULL;
79. }
80. //Очистка памяти из под дека
81. void Clear(Deque\* deque) {
82. while(!Empty(deque)) {
83. PopBack(deque);
84. }
85. }
86. //Вывод элементов дека на экран
87. void PrintDeque(Deque\* deque) {
88. Node\* current = deque->head;
89. while(current != NULL) {
90. printf("%.3f ", current->data);
91. current = current->next;
92. }
93. }
94. //Шейкерная сортировка
95. void shaker\_sort(Deque\* deque) {
96. Node\* begin = deque->head;// Указатель на начало дека
97. Node\* end = deque->tail; // Указатель на конец дека
98. while(begin != end) { // Пока указатели на начало и конец не совпадают
99. Node\* current = begin;
100. int change = 0;
101. // Прямой проход от начала к концу
102. while(current != end) {
103. if(current->data > current->next->data) { // Если текущий элемент больше следующего
104. change = 1;
105. Node\* next = current->next;// Сохраняем указатель на следующий узел
106. // Обновляем указатели дека,
107. if(deque->head == current) {
108. deque->head = next;
109. }
110. if(deque->tail == next) {
111. deque->tail = current;
112. }
113. if(begin == current) {
114. begin = next;
115. }
116. if(end == next) {
117. end = current;
118. }
119. // Обновляем указатели соседних узлов
120. if(current->prev != NULL) {
121. current->prev->next = next;
122. }
123. if(next->next != NULL) {
124. next->next->prev = current;
125. }
126. next->prev = current->prev;
127. current->next = next->next;
128. next->next = current;
129. current->prev = next;
130. } else {
131. current = current->next; // Переходим к следующему узлу
132. }
133. }
134. end = end->prev; // Сдвигаем конец на предыдущий узел
135. if(begin == end || !change) { // Если начало и конец совпали или не было изменений, выходим из цикла
136. break;
137. }
138. current = end;
139. change = 0;
140. // Обратный проход от конца к началу
141. while(current != begin) {
142. if(current->data < current->prev->data) { // Если текущий элемент меньше предыдущего
143. change = 1;
144. Node\* prev = current->prev; // Сохраняем указатель на предыдущий узел
145. if(deque->tail == current) {
146. deque->tail = prev;
147. }
148. if(deque->head == prev) {
149. deque->head = current;
150. }
151. if(begin == prev) {
152. begin = current;
153. }
154. if(end == current) {
155. end = prev;
156. }
157. if(current->next != NULL) {
158. current->next->prev = prev;
159. }
160. if(prev->prev != NULL) {
161. prev->prev->next = current;
162. }
163. prev->next = current->next;
164. current->prev = prev->prev;
165. prev->prev = current;
166. current->next = prev;
167. } else {
168. current = current->prev;
169. }
170. }
171. begin = begin->next;
172. if(!change) {
173. break;
174. }
175. }
176. }
177. typedef enum colour{
178. RED,
179. BLACK,
180. } Colour;
181. typedef struct rb\_node {
182. float data;
183. struct rb\_node\* left;
184. struct rb\_node\* right;
185. struct rb\_node\* parent;
186. Colour colour;
187. } RBNode;
188. typedef struct rb\_tree {
189. RBNode\* root;
190. } RBTree;
191. //создание пустого красно-чёрного дерева
192. RBTree CreateRB() {
193. RBTree tree;
194. tree.root = NULL;
195. return tree;
196. }
197. //получение цвета вершины
198. Colour GetColour(RBNode\* node) {
199. if(node == NULL) {
200. return BLACK;
201. }
202. return node->colour;
203. }
204. // Соединение узла с его родителем
205. void Connect(RBNode\* parent, RBNode\* child) {
206. if(child != NULL) {
207. child->parent = parent;
208. }
209. }
210. // Левый поворот вокруг узла
211. RBNode\* LeftTurn(RBNode\* node) {
212. RBNode\* right = node->right; // Сохраняем правого потомка текущего узла
213. node->right = right->left;// Перемещаем левого потомка правого узла на место правого потомка текущего узла
214. right->left = node; // Устанавливаем текущий узел левым потомком правого узла
215. Connect(node, node->right); // Соединяем текущий узел с его новым правым потомком
216. Connect(right, right->left); // Соединяем правый узел с его новым левым потомком (текущим узлом)
217. return right;
218. }
219. // Правый поворот вокруг узла
220. RBNode\* RightTurn(RBNode\* node) {
221. RBNode\* left = node->left;// Сохраняем правого потомка текущего узла
222. node->left = left->right; // Перемещаем левого потомка правого узла на место правого потомка текущего узла
223. left->right = node; // Устанавливаем текущий узел левым потомком правого узла
224. Connect(node, node->left); // Соединяем текущий узел с его новым правым потомком
225. Connect(left, left->right); // Соединяем правый узел с его новым левым потомком (текущим узлом)
226. return left;
227. }
228. //вставка элемента в дерево
229. void InsertRB(RBTree\* tree, float data) {
230. if(tree->root == NULL) { // Если дерево пустое , создается новый корневой узел черного цвета
231. tree->root = malloc(sizeof(RBNode));
232. if(tree->root == NULL) {
233. return;
234. }
235. tree->root->data = data;
236. tree->root->left = NULL;// Левый потомок отсутствует
237. tree->root->right = NULL;// Правый потомок отсутствует
238. tree->root->parent = NULL;// Родитель отсутствует (это корень)
239. tree->root->colour = BLACK; // Корень всегда черный
240. return;
241. }
242. RBNode\* current = tree->root;
243. // Поиск подходящего места для нового узла
244. while(1) {
245. if(current->data < data) { // Если вставляемая вершина больше текущей - идём вправо
246. if(current->right == NULL) { // Если правый потомок отсутствует
247. current->right = malloc(sizeof(RBNode));
248. if(current->right == NULL) {
249. return;
250. }
251. current->right->data = data;
252. current->right->colour = RED;// // Новый узел всегда красный
253. current->right->left = NULL;
254. current->right->right = NULL;
255. current->right->parent = current; // Устанавливаем родителя для нового узла
256. current = current->right; // Перемещаемся на новый узел
257. break;
258. }
259. current = current->right; // Идем направо и повторяем поиск
260. } else if (data < current->data) { // Если вставляемая вершина меньше текущего узла - идем влево
261. if(current->left == NULL) { // Если левый потомок отсутствует
262. current->left = malloc(sizeof(RBNode));
263. if(current->left == NULL) {
264. return;
265. }
266. current->left->data = data;
267. current->left->colour = RED; // новый узел всегда красный
268. current->left->left = NULL;
269. current->left->right = NULL;
270. current->left->parent = current; // Устанавливаем родителя для нового узла
271. current = current->left; // Перемещаемся на новый узел
272. break;
273. }
274. current = current->left; // Идем налево и повторяем поиск
275. } else { // Если вершина с таким значением уже существует - ничего не делаем
276. return;
277. }
278. }
279. while(1) { // Балансировка дерева после вставки
280. if(current->parent == NULL) {// Если текущий узел - корень дерева
281. current->colour = BLACK;
282. return;
283. }
284. RBNode\* father = current->parent;
285. if(GetColour(father) == BLACK) { // Если родительский узел черный, балансировка не требуется
286. return;
287. }
288. RBNode\* grandfather = father->parent; //Если отец красный - значит точно есть дедушка и он чёрный
289. RBNode\* uncle;
290. if(grandfather->left == father) { // Если текущий узел находится слева от дедушки
291. uncle = grandfather->right; // Дядя - правый потомок дедушки
292. if(GetColour(uncle) == RED) { // Случай 1: красный дядя
293. uncle->colour = BLACK; // Перекрашиваем дядю в черный цвет
294. father->colour = BLACK; // Перекрашиваем родителя в черный цвет
295. grandfather->colour = RED; // Продолжаем балансировку на уровне дедушки
296. current = grandfather;
297. continue;
298. }
299. RBNode\* grandgrandfather = grandfather->parent; // Прадедушка текущего узла
300. if(current == father->right) { // Случай 2: Треугольник (правый ребенок левого родителя)
301. grandfather->left = LeftTurn(father); // Левый поворот вокруг отца
302. Connect(grandfather, grandfather->left);
303. }
304. if(grandgrandfather != NULL) {// Случай 3: Линия (левый ребенок левого родителя)
305. if(grandgrandfather->left == grandfather) {
306. grandgrandfather->left = RightTurn(grandfather); // Правый поворот вокруг дедушки
307. Connect(grandgrandfather, grandgrandfather->left);
308. } else {
309. grandgrandfather->right = RightTurn(grandfather); // Правый поворот вокруг дедушки
310. Connect(grandgrandfather, grandgrandfather->right);
311. }
312. } else {
313. tree->root = RightTurn(grandfather);
314. tree->root->parent = NULL;
315. }
316. father->colour = BLACK;
317. grandfather->colour = RED;
318. return;
319. } else { // Аналогичные действия для случая, когда текущий узел находится справа от дедушки
320. uncle = grandfather->left;
321. if(GetColour(uncle) == RED) {
322. uncle->colour = BLACK;
323. father->colour = BLACK;
324. grandfather->colour = RED;
325. current = grandfather;
326. continue;
327. }
328. RBNode\* grandgrandfather = grandfather->parent;
329. if(current == father->left) {
330. grandfather->right = RightTurn(father);
331. Connect(grandfather, grandfather->right);
332. }
333. if(grandgrandfather != NULL) {
334. if(grandgrandfather->left == grandfather) {
335. grandgrandfather->left = LeftTurn(grandfather);
336. Connect(grandgrandfather, grandgrandfather->left);
337. } else {
338. grandgrandfather->right = LeftTurn(grandfather);
339. Connect(grandgrandfather, grandgrandfather->right);
340. }
341. } else {
342. tree->root = LeftTurn(grandfather);
343. tree->root->parent = NULL;
344. }
345. father->colour = BLACK;
346. grandfather->colour = RED;
347. return;
348. }
349. }
350. }
351. //Рекурсивная очистка дерева
352. void ClearRBRec(RBNode\* node) {
353. if(node == NULL) {
354. return;
355. }
356. ClearRBRec(node->left);// Рекурсивно очищаем левое поддерево
357. ClearRBRec(node->right);// Рекурсивно очищаем правое поддерево
358. free(node);
359. }
360. //Функция очистки дерева(использующая рекурсивную)
361. void ClearRB(RBTree\* tree){
362. ClearRBRec(tree->root);
363. tree->root = NULL;
364. }
365. //Рекурсивный вывод дерева
366. void PrintLevelRec(RBNode\* node, int level) {
367. if(node == NULL) {
368. return;
369. }
370. PrintLevelRec(node->right, level + 1);// Рекурсивно выводим правое поддерево, увеличивая уровень на 1
371. for(int i = 0; i < level; ++i) { // Печатаем отступы в зависимости от уровня
372. printf("$-");
373. }
374. printf(" %.3f ", node->data);
375. if(node->colour == RED) { // Проверяем цвет узла
376. printf("RED\n");
377. } else {
378. printf("BLACK\n");
379. }
380. PrintLevelRec(node->left, level + 1); // Рекурсивно выводим левое поддерево, увеличивая уровень на 1
381. }
382. //функция вывода дерева(использует рекурсивную функцию)
383. void PrintRBTree(RBTree\* tree) {
384. PrintLevelRec(tree->root, 0);
385. }
386. int main() {
387. int n;
388. while(1) {
389. printf("Enter the number of array elements: ");
391. int result = scanf("%d", &n);
392. if(result == 0) {
393. printf("Incorect data, insert again\n");
394. while(getchar() != '\n') {}
395. continue;
396. }
397. if(result == -1) {
398. printf("CTRL + D were inserted\n");
399. return 0;
400. }
401. if(n <= 0) {
402. printf("Incorect data, insert again\n");
403. }
404. else if(result == 1) {
405. break;
406. }
407. }
408. Deque deque = CreateDeque();
409. for(int i = 0; i < n; ++i) {
410. float data;
411. printf("%d - ",i);
412. int result = scanf("%f", &data);
413. if(result == 0) {
414. printf("Incorect data, insert again\n");
415. while(getchar() != '\n') {}
416. --i;
417. }
418. if(result == -1) {
419. printf("CTRL + D were inserted\n");
420. Clear(&deque);
421. return 0;
422. }
423. else if(result == 1) {
424. PushBack(&deque, data);
425. }
426. }
427. printf("Неотсортированный дек: ");
428. PrintDeque(&deque);
429. printf("\n");
430. shaker\_sort(&deque);
431. printf("Отсортированный дек: ");
432. PrintDeque(&deque);
433. printf("\n");
434. RBTree tree = CreateRB();
435. while(!Empty(&deque)) {
436. InsertRB(&tree, data.head->data);
437. PopFront(&deque);
438. }
439. PrintRBTree(&tree);
440. printf("\n");
441. Clear(&deque);
442. ClearRB(&tree);
443. }

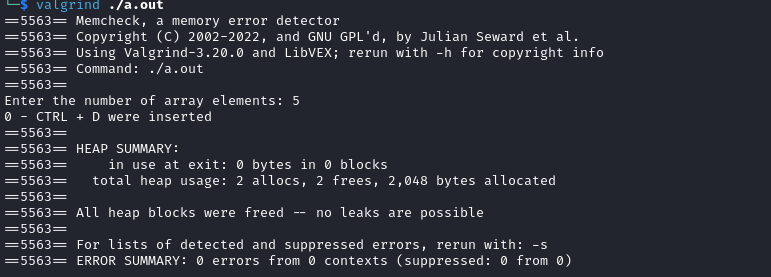
6 . Результаты тестирования

Были проведены тесты программы с различными входными данными. Все реализованные случаи вывода корректно обрабатываются и отображаются. Тестирование производилось на виртуальной машине с установленной на ней операционной системой Кали Линукс и версией компилятора gcc (Debian 13.2.0-13) 13.2.0

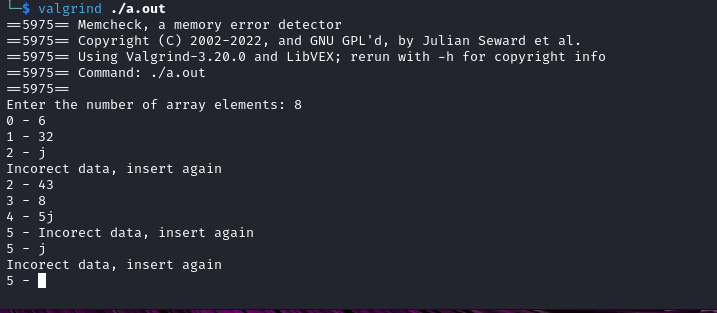
Попытка задать размер очереди, который невозможно реализовать.



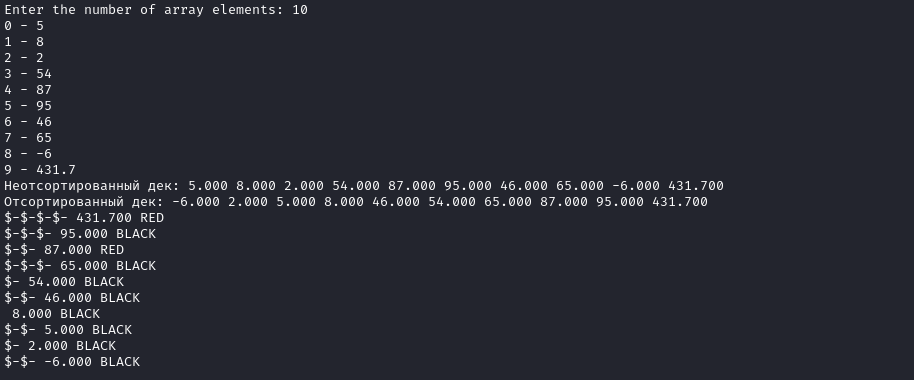
Нажатие CTL+D:



Попытка ввести некорректные данные:



Ввод десяти корректных значений



7. Заключение

В результате выполнения лабораторной работы была достигнута поставленная цель: разработана программа, которая выполняет шейкерную сортировку вещественных чисел с использованием дека на основе связного списка, результат которой записывается в красно-чёрное дерево.

Было проведено тестирование программы на языке C с использованием виртуальной машины с установленной на ней операционной системой Кали Линукс и версией компилятора gcc (Debian 13.2.0-13) 13.2.0.

в результате которого все разработанные выводы успешно работали.