**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №5**

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

Тема: Рандомизированная дерамида

|  |  |  |
| --- | --- | --- |
| Студент гр. 9304 |  | Борисовский В.Ю. |
| Преподаватель |  | Филатов Ар.Ю. |

Санкт-Петербург

2020

## Цель работы.

Изучить случайные бинарные деревья поиска. Реализовать случайное бинарное дерево поиска на языке программирования С++.

## Задание.

Вариант 13

БДП: Рандомизированная дерамида поиска (treap); действие: 1) По заданной последовательности элементов Elem построить структуру данных определённого типа – БДП или хеш-таблицу; 2) Для построенной структуры данных проверить, входит ли в неё элемент е типа Elem, и если входит, то удалить элемент е из структуры данных (первое обнаруженное вхождение). Предусмотреть возможность повторного выполнения с другим элементом.

## Выполнение работы.

## 1) Сперва я создал класс bin\_tree\_node, данный класс предназначен для хранения узла бинарного дерева, он имеет поля:

## std::shared\_ptr<bin\_tree\_node> left - левое поддерево.

## std::shared\_ptr<bin\_tree\_node> right - правое поддерево.

## int key - ключ элемента.

## int prior - приоритет элемента.

## 2) Затем был написан чекер строки bool string\_checker(std::string &str, int &index, std::vector<int> &vec), который проверяет ее на валидность и добавляет элементы строки в вектор, чтобы потом их было легче извлекать.

## 3) Затем была сделана структура struct Elem\_Pair предназначенная для удобства хранения данных одного элемента, дабы в дальнейшем создать дерево.

## 4) Дальше я написал функции split, merge и insert. Необходимые для работы с дерамидой.

## Функция split разделяет дерево T на два дерева L и R (которые являются возвращаемым значением) таким образом, что L содержит все элементы, меньшие по ключу X, а R содержит все элементы, большие X. Эта операция выполняется за O (log N). Реализация её довольно проста - очевидная рекурсия.

## Функция merge объединяет два поддерева T1 и T2, и возвращает это новое дерево. Эта операция также реализуется за O (log N). Она работает в предположении, что T1 и T2 обладают соответствующим порядком (все значения X в первом меньше значений X во втором). Таким образом, нам нужно объединить их так, чтобы не нарушить порядок по приоритетам Y. Для этого просто выбираем в качестве корня то дерево, у которого Y в корне больше, и рекурсивно вызываем себя от другого дерева и соответствующего сына выбранного дерева.

## Теперь очевидна реализация Insert (X, Y). Сначала спускаемся по дереву (как в обычном бинарном дереве поиска по X), но останавливаемся на первом элементе, в котором значение приоритета оказалось меньше Y. Мы нашли позицию, куда будем вставлять наш элемент. Теперь вызываем Split (X) от найденного элемента (от элемента вместе со всем его поддеревом), и возвращаемые ею L и R записываем в качестве левого и правого сына добавляемого элемента.

## 5) После этого мы реализовали функцию генерации дерамиды std::shared\_ptr<bin\_tree\_node> Treaps\_Building(Elem\_Pair \*seq, std::vector<int> vec), которая работает крайне просто. Сначала для каждого элемента из seq создается свой узел, а затем все узлы последовательно вставляются в нулевой узел. Тем самым получаем дерамиду.

## 6) Так же были реализованы функции Elem\_Pair \*Elem\_Generator(int count) и std::shared\_ptr<bin\_tree\_node> Treaps\_Generator(int count) предназначенные для генирации рандомных дерамид ключи и приоритеты которых находятся в диапазное от 0 до 100.

## 7) После этого была написана функция данная в условии варианта Erase. Эта функция рекурсивно проходится по дереву и при нахождении элемента удаляет его, а сыновей удаленного элемента объединяется с помощью функции merge, что гарантирует нам возможность повторного выполнения с другим элементом.

## 8) В заключение была реализована функция main(), в ней из аргументов командой строки принимается строка, если строка проходит проверку, создается дерамида и выводится на экран, иначе будет выведено «wrong string\n».

**Тестирование.**

Запуск программы начинается с ввода команды “make”, что приведёт к компиляции программы и созданию исполняемого файла lab5. Запуск программы производится командой ./lab5 и последующим вводом строки, содержащей логическое выражение. Тестирование производится с помощью

скрипта test\_skript.py. Запуск скрипта производится командой «python3 test\_skript.py» в директории lab5. Результаты тестирования представлены в приложении Б.

## Выводы.

Изучили случайные бинарные деревья поиска. Реализовали дерамиду на языке программирования С++.

Была написана программа, которая создает дерамиду и применяет к ней операцию удаления элемента. В процессе написания программы, использовались знания программирования рекурсивных алгоритмов на языке С++.

# Приложение А Исходный код программы

Название файла: main.cpp

1. #include <iostream>
2. #include <memory>
3. #include <vector>
4. #include <queue>
5. #include <iomanip>
6. struct Elem\_Pair{
7. int key\_elem, prior\_elem;
8. };
9. class bin\_tree\_node{
10. public:
11. std::shared\_ptr<bin\_tree\_node> left;
12. std::shared\_ptr<bin\_tree\_node> right;
13. int key, prior;
14. bin\_tree\_node(){};
15. bin\_tree\_node(int key, int prior) : key(key), prior(prior), left(NULL), right(NULL){};
16. };
17. Elem\_Pair \*Elem\_Generator(int count){
18. srand(time(0));
19. Elem\_Pair \*Elem\_Array = new Elem\_Pair[count];
20. for (int i = 0; i < count; i++){
21. Elem\_Array[i].key\_elem = rand() % 101;
22. Elem\_Array[i].prior\_elem = rand() % 101;
23. }
24. return Elem\_Array;
25. }
26. void split(std::shared\_ptr<bin\_tree\_node> t, int key, std::shared\_ptr<bin\_tree\_node> &left, std::shared\_ptr<bin\_tree\_node> &right){
27. if (!t){
28. left = right = NULL;
29. } else if (key < t -> key){
30. split(t -> left, key, left, t -> left);
31. right = t;
32. } else {
33. split(t -> right, key, t -> right, right);
34. left = t;
35. }
36. }
37. void insert (std::shared\_ptr<bin\_tree\_node> &t, std::shared\_ptr<bin\_tree\_node> it) {
38. if (!t)
39. t = it;
40. else if (it->prior > t->prior)
41. split (t, it -> key, it -> left, it -> right), t = it;
42. else if (it -> key < t -> key){
43. insert(t -> left, it);
44. } else {
45. insert(t->right, it);
46. }
47. }
48. std::shared\_ptr<bin\_tree\_node> Treaps\_Generator(int count){
49. if (count < 0){
50. std::cout << "there must be at least 1 element\n";
51. return nullptr;
52. }
53. Elem\_Pair \*seq = Elem\_Generator(count);
54. for(int i = 0; i < count; i++){
55. std::cout << seq[i].key\_elem << " " << seq[i].prior\_elem << "\n";
56. }
57. std::shared\_ptr<bin\_tree\_node> Array\_Items[count];
58. for (int i = 0; i < count; i++){
59. Array\_Items[i] = std::make\_shared<bin\_tree\_node>();
60. Array\_Items[i] -> key = seq[i].key\_elem;
61. Array\_Items[i] -> prior = seq[i].prior\_elem;
62. }
63. for (int i = 1; i < count; i++){
64. insert(Array\_Items[0], Array\_Items[i]);
65. }
66. delete []seq;
67. return Array\_Items[0];
68. }
69. std::shared\_ptr<bin\_tree\_node> Treaps\_Building(Elem\_Pair \*seq, std::vector<int> vec){
70. std::shared\_ptr<bin\_tree\_node> Array\_Items[vec.size() / 2];
71. for (int i = 0; i < vec.size() / 2; i++){
72. Array\_Items[i] = std::make\_shared<bin\_tree\_node>();
73. Array\_Items[i] -> key = seq[i].key\_elem;
74. Array\_Items[i] -> prior = seq[i].prior\_elem;
75. }
76. for (int i = 1; i < vec.size() / 2; i++){
77. insert(Array\_Items[0], Array\_Items[i]);
78. }
79. return Array\_Items[0];
80. }
81. void Merge(std::shared\_ptr<bin\_tree\_node> &t, std::shared\_ptr<bin\_tree\_node> left, std::shared\_ptr<bin\_tree\_node> right){
82. if (!left || !right){
83. t = left ? left : right;
84. } else if (left -> prior > right -> prior){
85. Merge(left -> right, left -> right, right);
86. t = left;
87. } else {
88. Merge(right -> left, left, right -> left);
89. t = right;
90. }
91. }
92. void Erase(std::shared\_ptr<bin\_tree\_node> &t, int key, int prior){
93. if (t -> key == key && t -> prior == prior){
94. Merge(t, t -> left, t -> right);
95. } else {
96. if (key < t -> key){
97. Erase(t -> left, key, prior);
98. } else {
99. Erase(t -> right, key, prior);
100. }
101. }
102. }
103. bool string\_checker(std::string &str, int &index, std::vector<int> &vec){
104. if (str[index] != '('){
105. return false;
106. }
107. index++;
108. std::string check\_num = "";
109. while (str[index] != ' ' && str[index]){
110. check\_num += str[index];
111. index++;
112. }
113. if (!str[index]){
114. return false;
115. }
116. if (!check\_num.empty()) {
117. char \*endptr;
118. const char \*c\_string = check\_num.c\_str();
119. vec.push\_back(strtol(c\_string, &endptr, 10));
120. if (\*endptr) {
121. return false;
122. }
123. } else {
124. return false;
125. }
126. index++;
127. check\_num = "";
128. while (str[index] != ')' && str[index]){
129. check\_num += str[index];
130. index++;
131. }
132. if (!str[index]){
133. return false;
134. }
135. if (!check\_num.empty()) {
136. char \*endptr;
137. const char \*c\_string = check\_num.c\_str();
138. vec.push\_back(strtol(c\_string, &endptr, 10));
139. if (\*endptr) {
140. return false;
141. }
142. } else {
143. return false;
144. }
145. if (str[index + 1]){
146. index++;
147. return string\_checker(str, index, vec);
148. } else {
149. return true;
150. }
151. }
152. void display\_Treap(std::shared\_ptr<bin\_tree\_node> root, int space = 0, int height = 10) { //display treap
153. if (root == nullptr)
154. return;
155. space += height;
156. display\_Treap(root->right, space);
157. std::cout << '\n';
158. for (int i = height; i < space; i++)
159. std::cout << ' ';
160. std::cout << root -> key << "(" << root -> prior << ")\n";
161. std::cout << '\n';
162. display\_Treap(root->left, space);
163. }
164. int main(int argc, char\* argv[]) {
165. std::shared\_ptr<bin\_tree\_node> head = std::make\_shared<bin\_tree\_node>();
166. if(argc == 1){
167. std::cout << "Wrong expression\n";
168. return 0;
169. }
170. std::string str(argv[1]);
171. int index = 0;
172. std::vector<int> vec;
173. bool k = string\_checker(str, index, vec);
174. Elem\_Pair seq[vec.size() / 2];
175. if (k){
176. for (int i = 0; vec[i]; i += 2){
177. seq[i / 2].key\_elem = vec[i];
178. seq[i / 2].prior\_elem = vec[i + 1];
179. }
180. std::cout << "success\n";
181. head = Treaps\_Building(seq, vec);
182. display\_Treap(head);
183. } else {
184. std::cout << "wrong string\n";
185. return 0;
186. }
187. return 0;
188. }

# Приложение Б Тестирование

Результаты тестирования представлены в таблице Б.1

Таблица Б.1 — Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Результат проверки |
| 1. | (10 60)(20 80)(30 10)(40 30)(50 90)(60 40)(70 50)(80 20) | success  80(20)  70(50)  60(40)  50(90)  40(30)  30(10)  20(80)  10(60) | success |
| 2. | (20 30)(40 50)(10 20)(50 70) | success  50(70)  40(50)  20(30)  10(20) | success |
| 3. | (15 20)(20 22)(30 34) | success  30(34)  20(22)  15(20) | success |
| 4. | (10 20 | wrong string | wrong string |
| 5. | (cd10 32) | wrong string | wrong string |
| 6. | 50 60)(60 70) | wrong string | wrong string |