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 C++.

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 e типа Elem, и если входит, то удалить элемент e 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 C++.

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

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <memory>
#include <vector>
#include <queue>
#include <iomanip>


struct Elem_Pair{
    int key_elem, prior_elem;
};


class bin_tree_node{
public:
    std::shared_ptr<bin_tree_node> left;
    std::shared_ptr<bin_tree_node> right;
    int key, prior;
    bin_tree_node(){};
        bin_tree_node(int key, int prior) : key(key), prior(prior), left(NULL), right(NULL){};
};


Elem_Pair *Elem_Generator(int count){
    srand(time(0));
```

```
    Elem_Pair *Elem_Array = new Elem_Pair[count];
    for (int i = 0; i < count; i++){
        Elem_Array[i].key_elem = rand() % 101;
        Elem_Array[i].prior_elem = rand() % 101;
    }
    return Elem_Array;
}



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){
    if (!t){
        left = right = NULL;
    } else if (key < t -> key){
        split(t -> left, key, left, t -> left);
        right = t;
    } else {
        split(t -> right, key, t -> right, right);
        left = t;
    }
}


void insert (std::shared_ptr<bin_tree_node> &t,
  std::shared_ptr<bin_tree_node> it) {
    if (!t)
```

```
53.         t = it;
54.     else if (it->prior > t->prior)
55.         split (t, it -> key, it -> left, it -> right),  t =
  it;
56.     else if (it -> key < t -> key){
57.         insert(t -> left, it);
58.     } else {
59.         insert(t->right, it);
60.     }
61.}
62.
63.
64.std::shared_ptr<bin_tree_node> Treaps_Generator(int count){
65.     if (count < 0){
66.         std::cout << "there must be at least 1 element\n";
67.         return nullptr;
68.     }
69.
70.     Elem_Pair *seq = Elem_Generator(count);
71.     for(int i = 0; i < count; i++){
72.                 std::cout  <<  seq[i].key_elem  <<  "  "  <<
  seq[i].prior_elem << "\n";
73.     }
74.     std::shared_ptr<bin_tree_node> Array_Items[count];
75.     for (int i = 0; i < count; i++){
76.         Array_Items[i] = std::make_shared<bin_tree_node>();
77.         Array_Items[i] -> key = seq[i].key_elem;
78.         Array_Items[i] -> prior = seq[i].prior_elem;
79.     }
```

```
    for (int i = 1; i < count; i++){
        insert(Array_Items[0], Array_Items[i]);
    }
    delete []seq;
    return Array_Items[0];
}


std::shared_ptr<bin_tree_node>    Treaps_Building(Elem_Pair *seq, std::vector<int> vec){
     std::shared_ptr<bin_tree_node> Array_Items[vec.size() / 2];
    for (int i = 0; i < vec.size() / 2; i++){
        Array_Items[i] = std::make_shared<bin_tree_node>();
        Array_Items[i] -> key = seq[i].key_elem;
        Array_Items[i] -> prior = seq[i].prior_elem;
    }
    for (int i = 1; i < vec.size() / 2; i++){
        insert(Array_Items[0], Array_Items[i]);
    }
    return Array_Items[0];
     }


    void      Merge(std::shared_ptr<bin_tree_node>      &t, std::shared_ptr<bin_tree_node>                          left, std::shared_ptr<bin_tree_node> right){
        if (!left || !right){
            t = left ? left : right;
```

```
106.        } else if (left -> prior > right -> prior){
107.            Merge(left -> right, left -> right, right);
108.            t = left;
109.        } else {
110.            Merge(right -> left, left, right -> left);
111.            t = right;
112.        }
113.    }
114.
115.
116.    void Erase(std::shared_ptr<bin_tree_node> &t, int key,
  int prior){
117.        if (t -> key == key && t -> prior == prior){
118.            Merge(t, t -> left, t -> right);
119.        } else {
120.            if (key < t -> key){
121.                Erase(t -> left, key, prior);
122.            } else {
123.                Erase(t -> right, key, prior);
124.            }
125.        }
126.    }
127.
128.    bool  string_checker(std::string  &str,  int  &index,
  std::vector<int> &vec){
129.        if (str[index] != '('){
130.            return false;
131.        }
132.        index++;
```

```
        std::string check_num = "";
        while (str[index] != ' ' && str[index]){
            check_num += str[index];
            index++;
        }
        if (!str[index]){
            return false;
        }
        if (!check_num.empty()) {
            char *endptr;
            const char *c_string = check_num.c_str();
            vec.push_back(strtol(c_string, &endptr, 10));
            if (*endptr) {
                return false;
            }
        } else {
            return false;
        }
        index++;
        check_num = "";
        while (str[index] != ')' && str[index]){
            check_num += str[index];
            index++;
        }
        if (!str[index]){
            return false;
        }
        if (!check_num.empty()) {
            char *endptr;
```

```
            const char *c_string = check_num.c_str();
            vec.push_back(strtol(c_string, &endptr, 10));
            if (*endptr) {
                return false;
            }
        } else {
            return false;
        }
        if (str[index + 1]){
            index++;
            return string_checker(str, index, vec);
        } else {
            return true;
        }
    }


    void       display_Treap(std::shared_ptr<bin_tree_node>
  root, int space = 0, int height = 10) { //display treap
        if (root == nullptr)
            return;
        space += height;
        display_Treap(root->right, space);
        std::cout << '\n';
        for (int i = height; i < space; i++)
            std::cout << ' ';
```

```
190.         std::cout << root -> key << "(" << root -> prior
  << ")\n";
191.         std::cout << '\n';
192.         display_Treap(root->left, space);
193.     }
194.
195.
196.
197.
198.
199.
200.
201.
202.     int main(int argc, char* argv[]) {
203.                     std::shared_ptr<bin_tree_node>    head    =
  std::make_shared<bin_tree_node>();
204.         if(argc == 1){
205.             std::cout << "Wrong expression\n";
206.             return 0;
207.         }
208.         std::string str(argv[1]);
209.         int index = 0;
210.         std::vector<int> vec;
211.         bool k = string_checker(str, index, vec);
212.         Elem_Pair seq[vec.size() / 2];
213.         if (k){
214.             for (int i = 0; vec[i]; i += 2){
215.                 seq[i / 2].key_elem = vec[i];
216.                 seq[i / 2].prior_elem = vec[i + 1];
217.             }
```

```
            std::cout << "success\n";
            head = Treaps_Building(seq, vec);
            display_Treap(head);

        } else {
            std::cout << "wrong string\n";
            return 0;
        }
        return 0;
    }
```

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