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: Случайное бинарное дерево поиска

Студент гр. 9304 _____ Ковалёв П. Д.

Преподаватель _____ Филатов А. Ю.

Санкт-Петербург
2020

Цель работы.

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

Задание.

Вариант 8

БДП: случайное БДП; 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.

Сначала были написаны функции проверки строки на корректность *digitChecker()* и *charChecker()*, которые проверяют входную строку на корректность, а именно: входные данные могут быть или последовательностью целых чисел, или последовательностью символов. Функции проходятся по всей строке и проверяют, состоит ли она только из цифр/символов и пробелов. Если встречается неопознанный элемент, значит строка некорректна и работа программы прекращается.

После создается объект шаблонного класса BSTree с определенным типом, в зависимости от входных данных. Класс BSTree содержит поле *std::shared_ptr<Node<T>>* head, которое является указателем на корень дерева и методы работы со случайным БДП, а также вспомогательные методы. Сами узлы случайного БДП представлены шаблонным классом *Node*, имеющим поля *counter* — счетчик элементов, *data* — поле, хранящее элемент узла, а также два указателя на потомков узла. В конструкторе данного класса с помощью метода *searchAndInsert()* происходит построение случайного БДП.

Алгоритм работы метода *searchAndInsert()*:

Метод принимает элемент и указатель. Если указатель пуст, то создается узел дерева, который инициализируется элементом. Если же передан непустой указатель, то введенный элемент сравнивается с элементом, лежащим в узле по данному указателю: если введенный элемент меньше элемента узла, то вызывается метод *searchAndInsert()* для левого поддерева, в противном случае для правого. Иначе поле *counter* узла дерева увеличивается на 1.

После создания объекта класса BSTree, проводится КЛП — обход данного дерева, с целью вывести его узлы. После этого, в цикле происходит считывание данных из строки, содержащей элементы для удаления и для каждого элемента строки вызывается метод *searchAndDelete()*.

Алгоритм работы метода *searchAndDelete()*:

Сначала происходит проверка на непустой указатель. Далее сравнивается элемент, который надо удалить, с элементом, лежащим в узле по данному указателю. Если элемент, который надо удалить, меньше элемента в узле, метод *searchAndDelete()* запускается для левого поддерева, если же больше элемента в узле, то рекурсивно запускается для правого поддерева. Если же удаляемый элемент равен элементу в узле, то сначала проверяется счетчик элементов: если он больше 1, то он декрементируется. Если нет, то происходит проверка на потомков узла: если потомков нет, то узел является листом и его можно спокойно удалить. Для этого вызывается метод *findParent()*, алгоритм работы которого состоит в том, чтобы найти родителя определенного узла и сделать указатель на определенный узел пустым.

Если же у узла есть правый потомок, то происходит поиск минимального элемента в правом поддереве и замена данного узла на минимальный элемент, который удаляется. В случае, если узел с минимальным элементом не является листом, а имеет потомков, для него вызывается метод *searchAndDelete()*.

Если же у узла есть левый потомок, то происходит поиск максимального элемента в левом поддереве и замена данного узла на максимальный элемент, который удаляется. В случае, если узел с максимальным элементом не является листом, а имеет потомков, для него вызывается метод *searchAndDelete()*.

Если же метод не может найти элемент, который надо удалить, он сообщает об этом и прекращает работу.

После удаления элементов происходит завершение работы программмы.

Тестирование.

Запуск программы начинается с запуска команды *make* в терминале, что приведет к созданию исполняемого файла *lab5*. Запуск программы начинается с ввода команды *./lab5* в терминале в директории *lab5*. Тестирование же проводится с помощью скрипта *tester.py*, который запускается командой *python3 tester.py* в командной строке в директории *lab5*. В текстовых файлах лежат входные данные. Подавать на вход программе нужно две строки, элементы в которых разделены пробелами, а сами строки взята в кавычки. Первая строка содержит элементы, по которым будет построено случайное БДП, вторая строка содержит элементы, которые необходимо удалить. Таким образом, поддерживается возможность удалить несколько элементов.

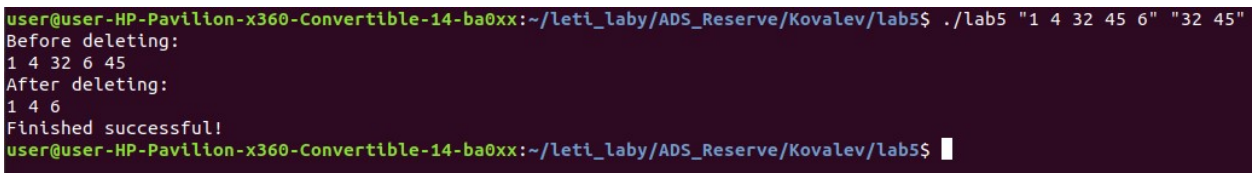

Рисунок 1 — Пример запуска программы

Результаты тестирования представлены в приложении Б.

Выводы.

Изучили случайные бинарные деревья поиска. Реализовали случайное бинарное дерево поиска на языке программирования C++.

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

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

main.cpp:

```
#include <iostream>
#include <string>
#include <sstream>
#include <memory>

template <typename T>
class Node{
public:
    int counter;
    T data;
    std::shared_ptr<Node<T>> left {nullptr};
    std::shared_ptr<Node<T>> right {nullptr};
    Node(){
        counter = 0;
    }
};

template <typename T>
class BSTree {
public:

    std::shared_ptr<Node<T>> head {nullptr};

    BSTree(std::string &input) {
        std::istringstream s(input);
        T elem;
        while (s >> elem) {
            searchAndInsert(elem, head);
        }
    }
```

```
        ~BSTree() = default;

         void searchAndInsert(T elem, std::shared_ptr<Node<T>>&
ptr) {
            if (!ptr) {
                ptr = std::make_shared<Node<T>>();
                ptr->data = elem;
                ptr->counter = 1;
            } else if (elem < ptr->data) {
                searchAndInsert(elem, ptr->left);
            } else if (elem > ptr->data) {
                searchAndInsert(elem, ptr->right);
            } else {
                ptr->counter++;
            }
        }

         void searchAndDelete(T elem, std::shared_ptr<Node<T>>&
ptr){
           if(ptr) {
                if(ptr == head && !ptr->left && !ptr->right){
                    head = nullptr;
                    return;
                }
                if (elem < ptr->data) {
                    searchAndDelete(elem, ptr->left);
                } else if (elem > ptr->data) {
                    searchAndDelete(elem, ptr->right);
                } else if (elem == ptr->data) {
                    if (ptr->counter > 1) {
                        ptr->counter -= 1;
                        return;
                    } else {
```

```
                        if (!ptr->left && !ptr->right) {//если
нет потомков
                            findParent(elem, head);
                            return;
                        } else if (ptr->right) {
                             std::shared_ptr<Node<T>> leftSon =
std::make_shared<Node<T>>();
                            leftSon = findMin(ptr->right);
                            T value = leftSon->data;
                            int ct = leftSon->counter;
                                  searchAndDelete(leftSon->data,
leftSon);
                            ptr->counter = ct;
                            ptr->data = value;
                            return;
                        } else if (ptr->left) {
                             std::shared_ptr<Node<T>> rightSon =
std::make_shared<Node<T>>();
                            rightSon = findMax(ptr->left);
                            T value = rightSon->data;
                            int ct = rightSon->counter;
                                 searchAndDelete(rightSon->data,
rightSon);
                            ptr->counter = ct;
                            ptr->data = value;
                            return;
                        }
                    }
                }
            }else{
                std::cout << "There is no element " << elem << "
in tree.\n";
            }
        }
```

```
        void findParent(T elem, std::shared_ptr<Node<T>> p){
            std::shared_ptr<Node<T>> ptr = p;
            if(ptr) {
                if (ptr->data == elem) {
                    return;
                }
                if (ptr->left) {
                    if (ptr->left->data == elem) {
                        ptr->left = nullptr;
                    }
                }
                if (ptr->right) {
                    if (ptr->right->data == elem) {
                        ptr->right = nullptr;
                    }
                }
                if (ptr->data < elem) {
                    ptr = ptr->right;
                    findParent(elem, ptr);
                } else if (ptr->data > elem) {
                    ptr = ptr->left;
                    findParent(elem, ptr);
                }
            }else{
                return;
            }
        }

                                            std::shared_ptr<Node<T>>
findMin(std::shared_ptr<Node<T>> p){
                                 std::shared_ptr<Node<T>>   temp   =
std::make_shared<Node<T>>();
            temp = p;
```

```
            while(temp->left){
                temp = temp->left;
            }
            return temp;
        }

                                                std::shared_ptr<Node<T>>
findMax(std::shared_ptr<Node<T>> p){
                                std::shared_ptr<Node<T>>   temp   =
std::make_shared<Node<T>>();
            temp = p;
            while(temp->right){
                temp = temp->right;
            }
            return temp;
        }

        void klp(std::shared_ptr<Node<T>> tmp) {
            if (tmp) {
                std::cout << tmp->data << ' ';
                if (tmp->left) klp(tmp->left);
                if (tmp->right) klp(tmp->right);
            }
        }
    };

    bool digitChecker(std::string& s){
        int len = s.length();
        int counter = 0;
        for(int i = 0; i < len; i++){
            if(isdigit(s[i]) || s[i] == ' '){
                counter++;
            }else{
                break;
```

```
            }
        }
        return counter==len;
    }

    bool charChecker(std::string& s){
        int len = s.length();
        int counter = 0;
        for(int i = 0; i < len; i++){
            if(isalpha(s[i]) || s[i] == ' '){
                counter++;
            }else{
                break;
            }
        }
        return counter==len;
    }

    int main(int argc, char* argv[]) {
        if(argc < 3){
         std::cout << "Incorrect input!\n";
            return 0;
        }
        std::string input(argv[1]);
        std::string elemsToDelete(argv[2]);
        if(isdigit(input[0])){
                                    if(digitChecker(input)    &&
digitChecker(elemsToDelete)){
                BSTree<int> BTree(input);
                std::cout << "Before deleting:\n";
                BTree.klp(BTree.head);
                std::cout << '\n';
                std::istringstream s(elemsToDelete);
                int elem;
```

```
                while (s >> elem) {
                    BTree.searchAndDelete(elem, BTree.head);
                }
                std::cout << "After deleting:\n";
                BTree.klp(BTree.head);
                std::cout << '\n';
            }else{
                std::cout << "Incorrect input!\n";
                return 0;
            }
        }else if(isalpha(input[0])){
            if(charChecker(input) && charChecker(elemsToDelete))
{
                BSTree<char> BTree(input);
                std::cout << "Before deleting:\n";
                BTree.klp(BTree.head);
                std::cout << '\n';
                std::istringstream s(elemsToDelete);
                char elem;
                while (s >> elem) {
                    BTree.searchAndDelete(elem, BTree.head);
                }
                std::cout << "After deleting:\n";
                BTree.klp(BTree.head);
                std::cout << '\n';
            }else{
                std::cout << "Incorrect input!\n";
                return 0;
            }
        }else{
            std::cout << "Incorrect input!\n";
            return 0;
        }
        std::cout << "Finished successful!\n";
```

```
    return 0;
}
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Таблица Б.1 - Примеры тестовых случаев

№ п/п	Входные данные	Выходные данные	Комментарии
1.	a b c d d c	Before deleting: a b c d After deleting: a b Finished successful!	
2.	1 2 4 5 5	Before deleting: 1 2 4 5 After deleting: 1 2 4 Finished successful!	
3.	a h i o t w o g h j s e e h l a u	Before deleting: a h g e i o j t s w There is no element l in tree. There is no element u in tree. After deleting: g h i o j t s w Finished successful!	
4.	6 7 3 923 58 1 0 8 54 6 7 8 9 0	Before deleting: 6 3 1 0 7 923 58 8 54 There is no element 9 in tree. After deleting: 54 3 1 58 923 Finished successful!	
5.	1 2 3 4 5 6 7 8 9 10 11 3 45 76 1 3 4	Before deleting: 1 2 3 4 5 6 7 8 9 10 11	

		There is no element 45 in tree. There is no element 76 in tree. There is no element 3 in tree. After deleting: 2 5 6 7 8 9 10 11 Finished successful!	
6.	8 5 4 14 4 10 13 0 14 5 0 12 6 7 9 12 3 0	Before deleting: 8 5 4 0 14 10 13 12 There is no element 6 in tree. There is no element 7 in tree. There is no element 9 in tree. There is no element 3 in tree. After deleting: 8 5 4 0 14 10 13 Finished successful!	
7.	h g e q p t u o n g s l f e t r y i p o	Before deleting: h g e q p o n t s u There is no element l in tree. There is no element f in tree. There is no element r in tree. There is no element y in tree. There is no element i	

		in tree. After deleting: h g q n u s Finished successful!	
8.	p o i u y t r e w q h u k o l e r d	Before deleting: p o i e u t r q y w There is no element h in tree. There is no element k in tree. There is no element l in tree. There is no element d in tree. After deleting: p i w t q y Finished successful!	
9.	b n m d e q s x h j k k h r t y u	Before deleting: b n m d e h j k q s x There is no element r in tree. There is no element t in tree. There is no element y in tree. There is no element u in tree. After deleting: b n m d e j q s x Finished successful!	
10.	d f e r t y u a f d a d f g h j	Before deleting: d a f e r t y u There is no element g	

		in tree. There is no element h in tree. There is no element j in tree. After deleting: d f e r t y u Finished successful!	