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: Поиск и вставка в БДП

Студент гр. 7381 _____ Павлов А.П.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2018

Цель работы.

Изучить структуру рандомизированной пирамиды поиска (treap). Научиться работать с ней, а именно добавлять элементы и выполнять поиск.

Задание.

Вариант 13:

БДП: Рандомизированная пирамида поиска (treap), действие 1+2а

Постановка задачи.

1) По заданному файлу F (типа file of Elem), все элементы которого различны, построить БДП типа рандомизированная пирамида (treap);
2) Для построенного БДП проверить, входит ли в него элемент типа Elem, и если не входит, то добавить элемент в дерево поиска.

Описание алгоритма вставки элемента.

Метод `insert` получает на вход указатель на корень дерамиды и значение добавляемого элемента. Сначала проверяется, существует ли данный элемент в дереве, если да, то метод завершает работу и возвращает указатель на корень. В ином случае происходит разрезание дерева на два дерева по значение переданного элемента таким образом, что в первой дерамиде находятся элементы меньшие или равные значению, а во второй – большие. Затем выделяется память под переданный элемент, то есть создается дерево, состоящее из одного узла, которое потом объединяется с первым деревом. Созданное дерево с учетом приоритетов объединяется с вторым деревом, таким образом переданный элемент вставляется в дерево.

Методы класса Treap и структуры данных:

1.
```
struct node{
        elem key;
        double priority;
        node * left;
```

```
        node * right;
    };
```

`key` – ключ, хранящийся в узле рандомизированной пирамиды;

`priority` – приоритет, хранящийся в узле рандомизированной пирамиды;

`left` – указатель на левого сына узла;

`right` – указатель на правого сына узла.

2. `bool exists(node * root, Type val)`

`root` – указатель на узел пирамиды.

`val` – значение элемента, который ищется в пирамиде.

Метод возвращает true, если переданный элемент существует в пирамиде, иначе false.

4. `std::pair <node *, node *> split(node * root, Type val)`

`root` – указатель на узел пирамиды;

`val` – значение элемента.

Метод разрезает исходное дерево `root` по ключу `val`. Возвращает такую пару деревьев, что в первом дереве ключи меньше `val`, а во втором дереве ключи больше `val`.

5. `node * merge(node * root1, node * root2)`

`root1` – указатель на узел первой пирамиды;

`root2` – указатель на узел второй пирамиды;

Метод объединяет два декартовых дерева в одно с учетом их приоритетов и возвращает указатель на новое дерево.

6. `node * insert(node * root, Type val)`

`root` – указатель на узел пирамиды.

`val` – значение элемента.

Метод вставки элемента в дерамиду. Возвращает указатель на декартово дерево.

7. `void remove(node * root)`

`node` – указатель на узел пирамиды;

Метод удаляет пирамиду.

8. `void display(node * root, Trunk * prev, bool isRight)`

`root` – указатель на узел пирамиды;

`prev` – указатель на структуру `Trunk`;

`isRight` – булева переменная, отвечающая за то, является ли элемент правым сыном.

Тестирование.

Было сделано 9 тестов для демонстрации и проверки работы программы.

Файл с данными для 1 и 2, 3 и 4, 5 и 6, 7 и 8 один, чтобы продемонстрировать, что пирамида действительно рандомизированная.

Тест №	Данные	Результат
1	4 8 6 14 74 2 3 5	The built treap: .---(74; 19169) | `---(14; 26500) .---(8; 18467) .---(6; 6334) ---(4; 41) `---(3; 11478) `---(2; 15724) Enter element that you want to insert: Adding new element: 5 .---(74; 19169) | `---(14; 26500) .---(8; 18467) .---(6; 6334) | `---(5; 29358) ---(4; 41) `---(3; 11478) `---(2; 15724)
2	74 354 596 123 695 596	The built treap: .---(695; 19169) .---(596; 6334) | `---(354; 18467) | `---(123; 26500) ---(74; 41) Enter element that you want to insert: Entered element exists in treap!
3	1 2 3 4 5 6 7 8 9 10	The built treap:

	11	.---(10; 24464) | `---(9; 26962) | `---(8; 29358) .---(7; 11478) | `---(6; 15724) | `---(5; 19169) | `---(4; 26500) .---(3; 6334) | `---(2; 18467) ---(1; 41) Enter element that you want to insert: Adding new element: 11 .---(11; 5705) | | .---(10; 24464) | | | `---(9; 26962) | | | `---(8; 29358) | | .---(7; 11478) | | | `---(6; 15724) | | | `---(5; 19169) | | | `---(4; 26500) | `---(3; 6334) | `---(2; 18467) ---(1; 41)
4	84 473 13 58 457 234 96 98	The built treap: .---(473; 18467) | `---(457; 19169) .---(234; 15724) .---(96; 11478) ---(84; 41) | .---(58; 26500) `---(13; 6334) Enter element that you want to insert: Adding new element: 98 .---(473; 18467) | `---(457; 19169) .---(234; 15724) | `---(98; 29358) .---(96; 11478) ---(84; 41) | .---(58; 26500) `---(13; 6334)

Для более наглядной демонстрации работы программы был создан bat-скрипт, последовательно выводящий содержимое очередного теста и результат работы программы для этого теста.

Выберем 1 тест для детального обзора. Программа считывает элемент, который надо найти в пирамиде. Это число 5. С помощью метода split, дерево разрезается на два таким образом, что в первом дереве элементы меньше или равные 5, в нашем случае этим деревом является корень исходного дерева и его левое поддерево. Вторым деревом становится правое поддерево исходного корня. Вызывается функция merge для первого поддерева и добавляемого элемента. Слияние происходит таким образом, что добавляемый становится правым сыном 1 дерева, так как его случайный приоритет больше приоритета корня. Дальше происходит слияние первого и второго дерева аналогично и возвращается указатель на корень дереваю.

Вывод

В ходе лабораторной работы были изучены принципы работы с БДП типа рандомизированная пирамида, научились добавлять элементы и выполнять поиск.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ФАЙЛА MAIN.CPP

```
#include <iostream>
#include <cstring>
#include <sstream>

#include "treap.hpp"

int main(){
    std::string list;
    std::cout << "Enter leafs for tree:" << std::endl;
    getline(std::cin, list);
    std::cout << std::endl;
    std::stringstream ss;
    for(size_t i = 0; i < list.size(); i++)
        if(isalpha(list[i]))
            std::cout << "Error: " << list[i] << " - was not digit! Stop
building the tree..." << std::endl;
    ss.str(list);
    Treap<int> tree;
    int value;
    std::cout << "\t\tWork with tree..." << std::endl;
    while(ss >> value) {
       tree.top = tree.insert(tree.top, value);
    }

    std::cout << "The built treap:" << std::endl;
    tree.display(tree.top, nullptr, false);

    int element;
    std::cout << "Enter element that you want to insert: ";
    std::cin >> element;
    if(std::cin.fail()){
        std::cout << "Error: entered value is not digit!" << std::endl;
        return 0;
    }

    if(tree.exists(tree.top, element)){
        std::cout << "Entered element exists in treap!" << std:: endl;
        return 0;
    }
    else{
        std::cout << "Adding new element:\t" << element << std::endl;
        tree.insert(tree.top, element);
        tree.display(tree.top, nullptr, false);
    }
}
```

ПРИЛОЖЕНИЕ Б
ИСХОДНЫЙ КОД ФАЙЛА TREAP.HPP

```
#include <iostream>
#include <cstdlib>
#include <utility>//for pair

template <typename Type>
class Treap{
    struct node{
        Type key;
        int priority;
        node * left;
        node * right;
    };
public:
    node * top;//pointer to the root of treap

    Treap(){
        top = nullptr;
    }

    bool exists(node * root, Type val){
        if(root == nullptr)
            return false;
        if(val == root->key)
            return true;
        if(val > root->key)
            return exists(root->right, val);
        return exists(root->left, val);
    }
```

```
        std::pair <node *, node *> split(node * root, Type
val){//cutting two trees by value
            if(root == nullptr)
                return {nullptr, nullptr};
            if(root->key <= val){
                auto res = split(root->right, val);
                root->right = res.first;
                return {root, res.second};
            }
            else{
                auto res = split(root->left, val);
                root->left = res.second;
                return {res.first, root};
            }
        }

        node * merge(node * root1, node * root2){//merging two trees
with priority
            if(root1 == nullptr)
                return root2;
            if(root2 == nullptr)
                return root1;
            if(root1->priority <= root2->priority){
                root1->right = merge(root1->right, root2);
                return root1;
            }
            else{
                root2->left = merge(root1, root2->left);
                return root2;
            }
        }

        node * insert(node * root, Type val){
            if(exists(root, val)){
```

```
                std::cout << "Entered element has already inserted in
treap" << std::endl;
                return root;
            }
            std::cout << "Adding:\t" << val << std::endl;
            auto res = split(root, val);
            std::cout << "Two trees after split:" << std::endl;
            std::cout << "First tree:" << std::endl;
            display(res.first, nullptr, false);
            std::cout << "Second tree:" << std::endl;
            display(res.second, nullptr, false);
            node * newnode = new node;
            newnode->key = val;
            newnode->priority = rand();
            newnode->left = nullptr;
            newnode->right = nullptr;

            node * tmp1 = merge(res.first, newnode);
            std::cout << "Treap after first merge: " << std::endl;
            display(tmp1, nullptr, false);
            node * tmp2 = merge(tmp1, res.second);
            std::cout << "Treap after second merge: " << std::endl;
            display(tmp2, nullptr, false);
            return tmp2;
          //  return merge(merge(res.first, newnode), res.second);
        }

        ~Treap(){//destructor
            if(top)
                remove(top);
        }

        void remove(node * root){//remove tree
            if(root == nullptr)
```

```
            return;
        remove(root->left);
        remove(root->right);
        delete root;
    }

    struct Trunk {//structure for printing treap
        Trunk *prev;
        std::string branch;
        Trunk(Trunk *prev, std::string branch) {
            this->prev = prev;
            this->branch = branch;
        }
    };
     void showTrunks(Trunk *p) {//An auxiliary method for printing
branches of a treap
        if(p == nullptr)
            return;
        showTrunks(p->prev);
        std::cout << p->branch;
    }
     void inorder(node* t, Trunk* prev, bool isRight)
{      //method for print treap
        if(t == nullptr)
            return;
        std::string prev_str = "    ";
        Trunk *trunk = new Trunk(prev, prev_str);
        inorder(t->right, trunk, true);
        if(!prev)
            trunk->branch = "---";
        else if(isRight) {
            trunk->branch = ".---";
            prev_str = "   |";
        }
```

```
            else {
                trunk->branch = "`---";
                prev->branch = prev_str;
            }
            showTrunks(trunk);
            std::cout << "(" <<t->key << "; " << t->priority << ")" <<
std::endl;
             if(prev)
                prev->branch = prev_str;
            trunk->branch = "   |";
             inorder(t->left, trunk, false);
            delete trunk;
        }

         void display(node * root, Trunk * prev, bool isRight)
{       //Метод отображения дерева
                if(!root) {
                    std::cout << "Treap is empty!" << std::endl <<
std::endl;
                    return;
                }
                inorder(root, nullptr, false);
                std::cout << std::endl;
            }
     };
```