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: Бинарные деревья поиска

Студент гр. 9303 _____ Максимов Е.А.

Преподаватель _____ Филатов Ар. Ю.

Санкт-Петербург

2020

Цель работы.

Реализовать программу для создания декартова дерева (*treap*) с соответствующим функционалом.

Задание.

Вариант №12 лабораторной работы.

Реализовать рандомизированную дерамиду поиска. Для построенной структуры данных проверить, входит ли в неё элемент, и если входит, то в скольких экземплярах. Реализовать возможность добавить элемент в структуру данных. Предусмотреть возможность повторного выполнения с другим элементом.

Основные теоретические положения.

Декартово дерево (или дерамида) – структура данных, объединяющая в себе бинарное дерево поиска и бинарную кучу.

Декартово дерево состоит из узлов, которые содержат два основных типа данных о них: сами данные и приоритет узла. Если рассматривать данные каждого узла, то полученная структура данных представляет собой бинарное дерево поиска. Если рассматривать приоритет каждого узла, то структура представляет собой структуру данных типа кучи.

Свойства декартового дерева:

1) структура данных представляет собой бинарное дерево поиска по значениям;

2) структура представляет собой структуру данных типа кучи по приоритетам;

3) для заданных значений данных и приоритетов декартово дерево определяется однозначно.

Выполнение работы.

В программе реализованы два класса для работы с дерамидой.

1. Класс *TreapNode* представляет собой узел дерамиды.

Свойства класса:

- *int key* – данные узла;
- *int priority* – приоритет узла;
- *int count* – количество экземпляров с одинаковыми данными;
- *TreapNode* left, right* – указатели на соседние узлы.

Методы класса:

1.1. Конструктор класса принимает на вход целочисленную переменную и создаёт узел со значением количество *count = 1,* указателями на соседние узлы *nullptr* и случайным приоритетом с помощью функции *rand().*

2. Класс *Treap* содержит всю структуру дерамиды и методы для работы с ней.

Свойства класса:

- *TreapNode* root* – указатель на корневой узел дерамиды.

Методы класса:

2.1. Класс имеет два конструктора.

2.1.1. Конструктор принимает на вход указатель на корень дерамиды *TreapNode* root*. Конструктор устанавливает соответствующее поле на это значение.

2.1.2. Конструктор принимает на вход вектор целочисленных переменных *vector<int> keys*. Для каждого элемента вызывается метод

add (см. ниже).

2.2. Деструктор класса вызывает рекурсивную функцию *destruct(root)* для освобождения динамической памяти (см. ниже).

2.3. *void print()* вызывает рекурсивную функцию *printNode()* для печати данных дерамиды (см. ниже).

2.4. *TreapNode* find(int key, TreapNode* root)* принимает на вход данные, которые необходимо найти в дерамиде и указатель на корень структуры дерамиды. Метод возвращает *nullptr,* если узла с такими данными нет, или указатель на узел с требуемыми данными.

2.5. *void add(int key)* добавляет данные в структуру дерамиды. Приоритет узла выбирается случайным образом. Если узел с такими данными уже есть, то его поле *count* увеличивается на 1. Метод не имеет возвращаемого значения.

2.6. *void remove (int key)* удаляет узел с данными, которые указаны в методе. Если узла с такими данными нет, то структура не изменяется. Метод не имеет возвращаемого значения.

2.7. Приватный метод *void destruct (TreapNode* node)* рекурсивно удаляет выделенную память на узлы дерамиды. Метод не имеет возвращаемого значения.

2.8. Приватный метод *void merge(Treap* another)* объединяет вместе две дерамиды при условии, что границы множества узлов дерамид не пересекаются. Метод вызывает рекурсивный метод *mergeTreapNodes* (см. ниже). После объединения другая структура дерамиды удаляется.

2.9. Приватный метод *Treap* split (int delimiter)* принимает на вход значение, по которому нужно разделить дерамиду на две независимые дерамиды. Метод вызывает рекурсивный метод *splitTreapNode* (см. ниже). Метод возвращает указатель на новую дерамиду.

2.10. Приватный метод *Pair splitTreapNode (TreapNode* node, int delimiter)* принимает на вход указатель на узел дерамиды и целочисленную переменную, относительно которой необходимо разделить дерамиду. Метод возвращает пару узлов, которые являются корнями дерамиды, представляющую собой тип данных *Pair*.

2.11. Приватный метод *TreapNode* mergeTreapNodes (TreapNode* L, TreapNode* R)* принимает на вход указатели на узлы, которые необходимо объединить. Функция рекурсивно объединяет узлы в одну дерамиду согласно их данным и приоритетам. Функция возвращает указатель на корень новой дерамиды.

2.12. *void printNode(TreapNode* node, int flag = 1)* принимает на вход указатель на узел. Функция рекурсивно печатает данные узлов в виде таблицы, содержащей значения данных в корневом узле, направление (слева или справа от текущего узла), значение данных в текущем узле, приоритет и количество экземпляров с таким типом данных.

В программе реализованы функции, приведённые ниже.

1. Функция *vector<int> readVector()* считывает данные из файла «*input.txt*» и создает на основе этих данных вектор целых чисел, который будет обработан конструктором класса *Treap*.

2. Функция *bool isNumber(char c)* возвращает *true,* если символ *c* является цифрой, и *false* в противном случае.

3. Функция *int main()* вызывает необходимые функции для обработки данных из файла и обрабатывает данные пользователя, которые предлагается ввести в консоль для дополнения структуры дерамиды, а после печатает данные на экран в виде таблицы.

Исходный код программы представлен в приложении А.

Результаты тестирования программы представлены в приложении Б.

Выводы.

Были реализованы классы и их методы, функции для создания и работы с данными дерамиды.

Были выполнены следующие требования: возможность ввода данных из консоли, вывод информации о работе программы в консоль.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp.

```
#include <iostream>
#include <fstream>
#include <ctime>
#include <vector>

using namespace std;

class Treap;
class TreapNode;
typedef pair<TreapNode*, TreapNode*> Pair;
bool isNumber(char c);
vector<int> readVector();

class TreapNode
{
public:
    int key, priority, count;
    TreapNode* left;
    TreapNode* right;

    TreapNode(int key)
    {
        this->key = key;
        this->priority = rand();
        left = nullptr;
        right = nullptr;
        count = 1;
    }

    ~TreapNode(){}
};

class Treap
{
public:
    TreapNode* root = nullptr;

    Treap(TreapNode* root = nullptr) { this->root = root; }
    Treap(vector<int> keys)
    {
        for (int i=0; i < keys.size(); i++)
            add(keys[i]);
    }

    ~Treap() { destruct(root); }
    void print() { printNode(this->root); }

    TreapNode* find(int key, TreapNode* root)
    {
```

```
        if (root == nullptr)
            return nullptr;
        else if (key < root->key)
            return find(key, root->left);
        else if (key > root->key)
            return find(key, root->right);
        else
            return root;
    }

    void add(int key)
    {
        TreapNode* findNode = this->find(key, this->root);
        if (findNode != nullptr)
        {
            findNode->count++;
            return;
        }
        Treap* another = this->split(key);
        Treap* aloneNode = new Treap((new TreapNode(key)));
        this->merge(aloneNode);
        this->merge(another);
        return;
    }

    void remove(int key)
    {
        TreapNode* findNode = this->find(key, this->root);
        if (findNode == nullptr)
            return;
        Treap* left = this->split(key-1);
        Treap* middle = this->split(key+1);
        delete middle;
        merge(left);
        return;
    }

private:
    void destruct(TreapNode* node)
    {
        if (node == nullptr)
            return;
        if (node->left != nullptr)
            destruct(node->left);
        if (node->right != nullptr)
            destruct(node->right);
        delete node;
        return;
    }

    void merge(Treap* another)
    {
        this->root = mergeTreapNodes(this->root, another->root);
        another->root = nullptr;
        delete another;
```

```
        return;
    }

    Treap* split(int delimiter)
    {
        Pair splitResult = splitTreapNode((this->root), delimiter);
        this->root = splitResult.first;
        Treap* out = new Treap(splitResult.second);
        return out;
    }

    Pair splitTreapNode(TreapNode* node, int delimiter)
    {
        if (node == nullptr)
            return {nullptr, nullptr};

        if (node->key <= delimiter)
        {
            Pair temp = splitTreapNode(node->right, delimiter);
            node-
>right = temp.first;                // Первый, слева
            return {node, temp.second};
        }
        else // Тогда node->key > delimiter
        {
            Pair temp = splitTreapNode(node->left, delimiter);
            node-
>left = temp.second;                // Второй, справа
            return {temp.first, node};
        }
    }

    TreapNode* mergeTreapNodes(TreapNode* L, TreapNode* R)
    {
        if (L == nullptr)
            return R;
        if (R == nullptr)
            return L;
        if (L->priority > R->priority)
        {
            L->right = mergeTreapNodes(L->right, R);
            return L;
        }
        else    // Тогда L->priority <= R->priority
        {
            R->left = mergeTreapNodes(L, R->left);
            return R;
        }
    }

    void printNode(TreapNode* node, int flag = 1)
    {
        if (node == nullptr)
            return;
        if(flag)
```

```
            cout << "\t\t";
        cout << node->key << "\t" << node-
>priority << "\t\t" << node->count << "\n";
        if (node->left != nullptr)
        {
            cout << node->key << ":\tleft\t";
            printNode(node->left, 0);
        }
        if (node->right != nullptr)
        {
            cout << node->key << ":\tright\t";
            printNode(node->right, 0);
        }
    }
};

vector<int> readVector()
{
    ifstream infile("input.txt");
    if (!infile.is_open())
    {
        cout << "Can't load \"input.txt\" data file. Stopping...\n
";
        exit(1);
    }
    vector<int> inputVector;
    int input;
    while(infile >> input)
        inputVector.push_back(input);
    infile.close();
    return inputVector;
}

bool isNumber(char c) { return ((c>=48)&&(c<=57)); }

int main()
{
    srand(time(NULL));
    Treap treap(readVector());
    string tempValueString;
    string inputString;
    while(true)
    {
        cout << "\nInput data loaded successfully.\n";
        cout << "Please, write new treap element or empty string f
or print:\t";
        getline(cin, inputString);
        if(inputString.length() == 0)
            break;
        if(!isNumber(inputString[0]))
            break;
        treap.add(stoi(inputString));
    }
    cout << "\nroot\tdir\tkey\tpriority\tcount\n";
    cout << "=============================================\n";
```

```
    treap.print();
    cout << endl;
    system("pause");
    return 0;
}
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Таблица 1 - Тестирование программы.

№	Входные данные	Выходные данные	Комментарий
1.		root dir key priority count ==========	Тест обработки пустого файла и пустой строки.
2.	1	root dir key priority count ========== 1 30182 1	Тест обработки одного элемента.
3.	1 7 1 3 7 3 5 7 5	root dir key priority count ========== 1 30368 2 1: right 5 21118 2 5: left 3 16060 2 5: right 7 16653 3	Тест обработки одинаковых элементов.
4.	0 1 3 5 7 9 11 13 15 0 2 4 6 8 10 12 14 16 0	root dir key priority count ========== 0 30655 3 0: right 16 30616 1 16: left 12 29503 1 12: left 2 28609 1 2: left 1 12240 1 2: right 9 26511 1 9: left 8 24604 1 8: left 4 22586 1 4: left 3 15243 1 4: right 5 8769 1 5: right 7 8026 1 7: left 6 1328 1 9: right 10 11036 1 10: right 11 5860 1 12: right 13 23994 1 13: right 14 10783 1 14: right 15 1192 1	Тест композиции двух тестов выше.