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: АВЛ-деревья

Студент гр. 9382 _____ Докукин В. М.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2020

Цель работы.

Изучение структуры данных — бинарное дерево поиска, её разновидностей, методов и особенностей работы с ней.

Основные теоретические положения.

Двоичное(бинарное) дерево поиска(БДП) — это бинарное дерево, на которое наложены дополнительные условия:

1. Поддеревья БДП — также БДП;
2. Ключи узлов левого поддерева БДП меньше или равны ключу корня БДП;
3. Ключи узлов правого поддерева БДП больше или равны ключу корня БДП.

Ключи узлов при этом должны быть сравнимы между собой.

АВЛ-дерево — это БДП, для каждого узла которого выполняется следующее условие: высота поддеревьев каждого узла различается не более чем на 1.

Задание.

Вариант 17. БДП: АВЛ-дерево; действие: 1+2в.

1. По заданной последовательности элементов Elem построить структуру данных определённого типа – БДП или хеш-таблицу;

2. в) Записать в файл элементы построенного БДП в порядке их возрастания; вывести построенное БДП на экран в наглядном виде.

Ход работы.

1. Классы и их методы.

В программе реализован один класс — AVLTree, реализующий АВЛ-дерево и методы работы с ним. Объекты класса хранят 4 значения: int height – высота дерева; int key – значение ключа; AVLTree* left и AVLTree* right – указатели на левое и правое поддерево.

Методы класса:

1) AVLTree(int key) – конструктор класса. Создаёт узел дерева с пустыми поддеревьями и устанавливает его ключ равным аргументу конструктора.

2) bool findKey(int k) – рекурсивный метод, проверяющая , есть ли в данном дереве элемент с ключом, равным аргументу. Если ключ дерева равен искомому или дерево является пустым, рекурсия прекращается, иначе — вызывается findKey() для левого и правого поддерева. Метод возвращает true, если элемент с таким ключом найден, иначе — false.

3) int getHeight() – метод-геттер, предоставляет доступ к значению приватного поля height дерева.

4) int bFactor() – считает баланс-фактор дерева, т. е. разницу высот правого и левого поддеревьев. Возвращает баланс-фактор данного дерева путём вызова getHeight() для правого и левого поддеревьев.

5) void fixHeight() – метод, обновляющий высоту дерева. Для обновления выбирается максимальная из высот поддеревьев дерева, после чего поле height устанавливается в полученное значение + 1. Используется при поворотах.

6) AVLTree* rotateLeftSmall(std::ostream* stream) – метод малого поворота дерева влево. Метод меняет местами указатели на поддеревья дерева и его правого поддерева в соответствии с принципом малого левого поворота, изображённым на иллюстрации 1, а также записывает сообщение о повороте в поток stream. Возвращаемое значение метода — указатель на дерево.

7) AVLTree* rotateLeftSmall(std::ostream* stream) – метод малого поворота дерева вправо. Метод меняет местами указатели на поддеревья дерева и его левого поддерева в соответствии с принципом малого правого поворота, изображённым на иллюстрации 1, а также записывает сообщение о повороте в поток stream. Возвращаемое значение метода — указатель на дерево.

8) AVLTree* balance(std::ostream* stream) – метод балансировки дерева. Метод обновляет высоту дерева и проверяет, нарушен ли баланс. Если баланс нарушен — производятся повороты в соответствии с отклонением баланса. В

методе используется тот факт, что большой левый поворот является композицией малого правого и малого левого поворотов, а большой правый — малого левого и малого правого поворотов. Сообщения о поворотах записываются в поток stream. Метод возвращает указатель на дерево после балансировки.

9) AVLTree* insert(int key) – вставляет в дерево элемент в соответствии с алгоритмом, описанным в 2.1. После вставки производится баланс дерева, если необходимо. Метод возвращает указатель на вставленный элемент после балансировки.

10) AVLTree* findMin() - рекурсивный метод поиска минимального элемента. Согласно теоретической информации, минимальный элемент в АВЛ-дереве находится левее всех остальных. Таким образом, метод рекурсивно идёт влево по дереву, пока не найдёт элемент, у которого нет левого сына. Возвращаемое значение — указатель на минимальный элемент.

11) AVLTree* removeMin(std::ostream* stream) – рекурсивный метод, который удаляет элемент для замещения из поддерева в случае удаления узла, не являющегося листом. Согласно теоретической информации, при удалении элемента, если он не является листом, следует заменить его наиболее близким по значению элементом правого поддерева. Таким образом, метод ищет в правом поддереве минимальный элемент и удаляет его. Метод возвращает указатель на правое поддерево, если у дерева нет левых поддеревьев, либо указатель на дерево после балансировки.

12) AVLTree* remove(int k, std::ostream* stream) – метод, удаляющий элемент из дерева согласно алгоритму, описанному в 2.1. После удаления элемента производится балансировка дерева. Возвращает указатель на

13) void destroy() - рекурсивно удаляет дерево. Вызывает самого себя для левого и правого поддеревьев, после чего удаляет себя.

14) void printTree(std::ostream* stream, int treeHeight, int depth = 0, int level = 0) – рекурсивный метод печати дерева на экран. Выводит дерево на экран по уровням в удобном для восприятия виде.

15) void printLKP(std::ostream* stream) – рекурсивный метод ЛПК-обхода дерева. Используется для вывода отсортированных элементов на экран и в файл.

Кроме того, была реализована функция void treeOutput(std::ostream* stream, AVLTree* tree), принимающая указатель на дерево и печатающая его на экран в наглядной форме.

Тестирование.

Результаты тестирования представлены в таблице ниже.

№ теста	Входные данные	Выходные данные	Комментарий
1	7 6 43 23 9 8 4 5	Inserting element 43 Inserting element 23 Rotating element 43 to the right. Rotating element 6 to the left. Inserting element 9 Inserting element 8 Rotating element 9 to the right. Rotating element 6 to the left. Inserting element 4 Rotating element 23 to the right. Inserting element 5 Rotating element 4 to the left. Rotating element 6 to the right. Sorted elements: 4 5 6 8 9 23 43	Проверка различных поворотов дерева
2	9 -5 5 -17 17 -3 3 -23 23 4	Inserting element 5 Inserting element -17 Inserting element 17 Inserting element -3 Inserting element 3 Rotating element 5 to the right. Rotating element -5 to the left. Inserting element -23 Rotating element -5 to the right. Inserting element 23 Inserting element 4 Sorted elements: -23 -17 -5 -3 3 4 5 17 23	Работа с отрицательными числами
3	-5	Sorted elements: Empty tree Your tree: [empty tree] ----------	Ввод некорректного количества элементов
4	0	Sorted elements: Empty tree Your tree:	Ввод нуля в качестве количества агрументов

		[empty tree] ----------	

Выводы.

В ходе выполнения лабораторной работы:

1. Были изучены бинарные деревья поиска, их разновидности(в частности — АВЛ-деревья), методы работы с ними. Был реализован класс, осуществляющий создание АВЛ-дерева и работу с ним.

2. Была написана программа, использующая реализованный класс и выполняющая поставленную задачу.

3. Была написана серия тестов, позволяющих качественно оценить работу программы.

Исходный код программы размещён в приложении 1.

ПРИЛОЖЕНИЕ 1
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: AVLTree.h

```
#ifndef COURSEWORK_AVLTREE_H
#define COURSEWORK_AVLTREE_H

#include <iostream>
#include <fstream>

class AVLTree {
private:
     int height;
     int key;
     AVLTree* left;
     AVLTree* right;
public:
     AVLTree(int);
     int getHeight();
     bool findKey(int);
     void printTree(std::ostream*, int, int, int);
     int bFactor();
     void fixHeight();
     AVLTree* rotateRightSmall(std::ostream*);
     AVLTree* rotateLeftSmall(std::ostream*);
     AVLTree* balance(std::ostream*);
     AVLTree* insert(int, std::ostream*);
     AVLTree* findMin();
     AVLTree* removeMin(std::ostream*);
     AVLTree* remove(int, std::ostream*);
     void destroy();
     void printLKP(std::ostream*);
};

#endif
```

Название файла: AVLTree.cpp

```
#include "pch.h"
#include <cmath>

#include "AVLTree.h"

AVLTree::AVLTree(int key) { // Конструктор дерева
     this->height = 1;
     this->key = key;
     this->left = nullptr;
     this->right = nullptr;
}
```

```
bool AVLTree::findKey(int k) { // Проверяем, есть ли в дереве
элемент с ключом == k
     if (this == nullptr) { // Если пустой элемент - прекращаем
рекурсию
          return false;
     }
     if (this->key == k) { // Если нашли - также прекращаем
рекурсию
          return true;
     }
     return this->left->findKey(k) | this->right->findKey(k); //
Запускаем поиск в левом и правом поддеревьях
}

int AVLTree::getHeight() { // Доступ к высоте элемента
     if (this != nullptr) {
          return this->height;
     }
     return 0;
}

int AVLTree::bFactor() { // Высчитывает баланс-фактор элемента
     return this->right->getHeight() - this->left->getHeight();
}

void AVLTree::fixHeight() { // Высчитываем высоту после поворота
     int lheight = left->getHeight();
     int rheight = right->getHeight();
     if (lheight > rheight) {
          this->height = lheight + 1;
     }
     else {
          this->height = rheight + 1;
     }
}

AVLTree* AVLTree::rotateRightSmall(std::ostream* stream) { //
Правый поворот
     if (stream != nullptr) *stream << "Rotating element " <<
this->key << " to the right.\n";
     //std::cout << "Производится малый поворот элемента " <<
this->key << " вправо.\n";
     AVLTree* tmp = this->left;
     this->left = tmp->right;
     tmp->right = this;
     fixHeight();
     tmp->fixHeight();
     return tmp;
}

AVLTree* AVLTree::rotateLeftSmall(std::ostream* stream) { // Левый
поворот
```

```
     if (stream != nullptr) *stream << "Rotating element " <<
this->key << " to the left.\n";
     //std::cout << "Производится малый поворот элемента " <<
this->key << " влево.\n";
     AVLTree* tmp = right;
     this->right = tmp->left;
     tmp->left = this;
     fixHeight();
     tmp->fixHeight();
     return tmp;
}

AVLTree* AVLTree::balance(std::ostream* stream) { //Балансируем
дерево
     fixHeight();
     if (this->bFactor() == 2)
     {
          if (this->right->bFactor() < 0) this->right = this-
>right->rotateRightSmall(stream);
          return this->rotateLeftSmall(stream);
     }
     if (this->bFactor() == -2)
     {
          if (this->left->bFactor() > 0) this->left = this->left-
>rotateLeftSmall(stream);
          return this->rotateRightSmall(stream);
     }
     return this;
}

AVLTree* AVLTree::insert(int k, std::ostream* stream) {
Вставляем элемент с ключом == k
     if (this == nullptr) { // Если дошли до пустого элемента,
создаём новый элемент
          //std::cout << "Производится вставка элемента " << k <<
'\n';
          if (stream != nullptr) *stream << "Inserting element "
<< k << '\n';
          return new AVLTree(k);
     }
     // Иначе...
     if (k < this->key) {
          this->left = this->left->insert(k, stream); // ...если
ключ элемента меньше ключа текущего, смотрим левое поддерево
     }
     if (k > this->key) {
          this->right = this->right->insert(k, stream); // ...если
ключ элемента больше ключа текущего, смотрим правое поддерево
     }
     return balance(stream);
}

AVLTree* AVLTree::findMin() { // Возвращает минимальный элемент
```

```
     if (this->left != nullptr) {
          return this->left->findMin();
     }
     else if (this->right != nullptr) {
          return this->right->findMin();
     }
     return this;
}

AVLTree* AVLTree::removeMin(std::ostream* stream) {  // Удаляет
минимальный элемент
     if (left == nullptr) {
          return right;
     }
     left = left->removeMin(stream);
     return balance(stream);
}

AVLTree* AVLTree::remove(int k, std::ostream* stream) { // Удаляет
элемент с ключом == k
     if (this == nullptr) return this; // Если дошли до пустого
элемента - возвращаем 0
     if (k < this->key) this->left = this->left->remove(k,
stream); // Если ключ меньше ключа текущего элемента, ищем в левом
поддереве
     else if (k > this->key) this->right = this->right->remove(k,
stream); // Если ключ больше ключа текущего элемента, ищем в
правом поддереве
     else {
          AVLTree* ltree = this->left;
          AVLTree* rtree = this->right;
          //std::cout << "Производится удаление элемента " << k <<
'\n';
          if (stream != nullptr) *stream << "Deleting element " <<
k << '\n';
          delete this;
          if (!rtree) return ltree;
          AVLTree* min = rtree->findMin();
          min->right = rtree->removeMin(stream);
          min->left = ltree;
          return min->balance(stream);
     }
     return balance(stream);
}

void AVLTree::destroy() { // Освобождаем память, выделенную под
дерево
     if (this != nullptr) {
          this->left->destroy();
          this->right->destroy();
          delete this;
     }
}
```

```
void AVLTree::printTree(std::ostream* stream, int treeHeight, int
depth = 0, int level = 0) { // Печатаем дерево в std::cout и в
файл по уровням
     if (depth == level) {
          if (this != nullptr) *stream << key; // Печатаем корень
          else *stream << "  ";
          for (int i = 0; i < pow(2, treeHeight - (level + 1));
i++) *stream << "\t"; //отступ
          if (level < treeHeight - 1) *stream << "    ";
          return;
     }
     else {
          if (this == nullptr) {
               this->printTree(stream, treeHeight, depth + 1,
level);
               this->printTree(stream, treeHeight, depth + 1,
level);
          }
          else {
               this->left->printTree(stream, treeHeight, depth +
1, level); // Печатаем левое поддерево
               this->right->printTree(stream, treeHeight, depth +
1, level); // Печатаем левое поддерево
          }
     }
}

void AVLTree::printLKP(std::ostream* stream) {
     if (this->left != nullptr) this->left->printLKP(stream);
     *stream << this->key << " ";
     if (this->right != nullptr) this->right->printLKP(stream);
}
```

Название файла: main.cpp

```
#include "pch.h"
#include <iostream>
#include <fstream>
#include <string>

#include "AVLTree.h"

void treeOutput(std::ostream* stream, AVLTree* tree) { //
Наглядный вывод дерева на экран
     if (tree == nullptr) {
          *stream << "[empty tree]\n";
          return;
     }
     for (int j = 0; j < tree->getHeight(); j++) {
          if (j < tree->getHeight() - 1) {
```

```
                for (int k = 0; k < pow(2, tree->getHeight() - j -
2) - 1; k++) {
                     *stream << "\t";
                }
                *stream << "    ";
           }
           tree->printTree(stream, tree->getHeight(), 0, j);
           *stream << "\n\n";
      }
}

int main()
{
      int a;
      std::cout << "Choose input option(0 - file input, 1 - console
input):\n";
      std::cin >> a;

      int count;
      int key;
      AVLTree* tree = nullptr;

      std::filebuf outputf;
      outputf.open("output.txt", std::ios::out);
      std::ostream output(&outputf);

      if (a) {
           std::cout << "Insert elements' count: ";
           std::cin >> count;
           for (int i = 0; i < count; i++) {
                std::cout << "Insert element key: ";
                std::cin >> key;
                if (tree == nullptr) tree = new AVLTree(key);
                else tree = tree->insert(key, &std::cout);
                treeOutput(&std::cout, tree);
           }
           output << "Sorted elements: ";
           if (tree != nullptr) tree->printLKP(&output);
           else output << "Empty tree\n";
           output << "\nYour tree:\n";
           treeOutput(&output, tree);
           tree->destroy();
           tree = nullptr;
           return 0;
      }

      std::ifstream inputf("tests.txt");
      if (!inputf) {
           std::cout << "Couldn't open file.\n";
           return 1;
      }

      while (!inputf.eof()) {
```

```
            inputf >> count;
            for (int i = 0; i < count; i++) {
                 inputf >> key;
                 if (tree == nullptr) tree = new AVLTree(key);
                 else tree = tree->insert(key, &std::cout);
            }
            output << "Sorted elements: ";
            if (tree != nullptr) tree->printLKP(&output);
            else output << "Empty tree\n";
            output << "\nYour tree:\n";
            treeOutput(&output, tree);
            tree->destroy();
            tree = nullptr;
            output << "-------------------------------------------
\n";
      }
      return 0;
}
```