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. 9303 _____ Муратов Р.А.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург
2020

Цель работы.

Изучить и реализовать АВЛ-дерево.

Задание.

Вариант 17.

БДП: АВЛ-дерево; действие: 1+2в.

В вариантах заданий 2-ой группы (БДП и хеш-таблицы) требуется:

1. По заданной последовательности элементов *Elem* построить структуру данных определённого типа – БДП или хеш-таблицу;
2. Выполнить одно из следующих действий:

а. Для построенной структуры данных проверить, входит ли в неё элемент *e* типа *Elem*, и если входит, то в скольких экземплярах. Добавить элемент *e* в структуру данных. Предусмотреть возможность повторного выполнения с другим элементом.

б. 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.

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

г. Другое действие.

Основные теоретические положения.

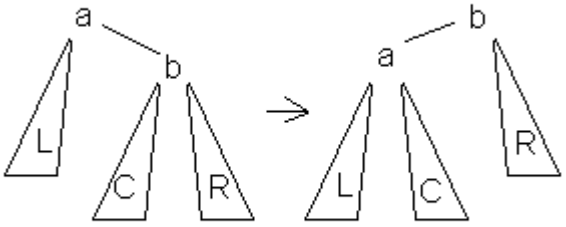
АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу. Доказано, что этого свойства достаточно для того, чтобы высота

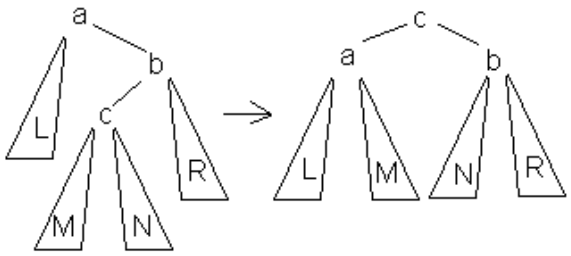
дерева логарифмически зависела от числа его узлов: высота h АВЛ-дерева с n ключами лежит в диапазоне от $\log_2(n+1)$ до $1.44 \cdot \log_2(n+2) - 0.328$. А так как основные операции над двоичными деревьями поиска (поиск, вставка и удаление узлов) линейно зависят от его высоты, то получаем гарантированную логарифмическую зависимость времени работы этих алгоритмов от числа ключей, хранимых в дереве. Напомним, что рандомизированные деревья поиска обеспечивают сбалансированность только в вероятностном смысле: вероятность получения сильно несбалансированного дерева при больших n хотя и является пренебрежимо малой, но остается не равной нулю.

Относительно АВЛ-дерева балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев = 2, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится <= 1, иначе ничего не меняет. Указанный результат получается вращениями поддерева данной вершины.

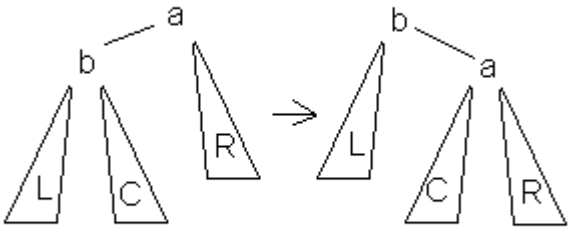
Используются 4 типа вращений:



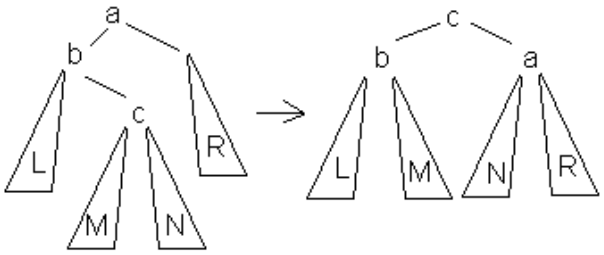
Малое левое вращение. Данное вращение используется тогда, когда (высота b-поддерева - высота L) = 2 и высота c-поддерева <= высота R.



Большое левое вращение. Данное вращение используется тогда, когда (высота b-поддерева - высота L) = 2 и высота c-поддерева > высота R.



Малое правое вращение. Данное вращение используется тогда, когда (высота b-поддерева - высота R) = 2 и высота C <= высота L.



Большое правое вращение. Данное вращение используется тогда, когда (высота b-поддерева - высота R) = 2 и высота c-поддерева > высота L.

В каждом случае достаточно просто доказать то, что операция приводит к нужному результату и что полная высота уменьшается не более чем на 1 и не может увеличиться. Также можно заметить, что большое левое вращение — это композиция правого малого вращения и левого малого вращения.

Традиционно, узлы АВЛ-дерева хранят не высоту, а разницу высот правого и левого поддеревьев (так называемый balance factor), которая может

принимать только три значения -1, 0 и 1. Однако, заметим, что эта разница все равно хранится в переменной, размер которой равен минимум одному байту (если не придумывать каких-то хитрых схем «эффективной» упаковки таких величин). Вспомним, что высота $h<1.44\cdot\log_2(n+2)$, это значит, например, что при $n=10^9$ (один миллиард ключей, больше 10 гигабайт памяти под хранение узлов) высота дерева не превысит величины $h=44$, которая с успехом помещается в тот же один байт памяти, что и balance factor. Таким образом, хранение высот с одной стороны не увеличивает объем памяти, отводимой под узлы дерева, а с другой стороны существенно упрощает реализацию некоторых операций.

Выполнение работы.

Реализован шаблонный класс узла `Node`, который имеет следующие поля: шаблонный поле `key`, по которому проводиться сравнение; поле `height`, в котором хранится высота узла (максимальная высота из двух поддеревьев + 1); `left` и `right` — указатели на левого и правого сына соответственно.

Также реализован шаблонный класс `AvlTree`, который представляет собой AVL-дерево. Поле `head_` хранит указатель на «голову» дерева, поле `nodeCounter_` является словарем, в котором для каждого элемента хранится количество его вхождений в дерево (в DOT-файле количество вхождении указано в скобках). Методы `insert` (вставка в дерево ключа), `remove` (удаление ключа из дерева, если он там есть), `printInfix` (вывод элементов дерева в файл в порядке возрастания), `printDot` (конвертация исходного дерева в валидный DOT-файл, чтобы его можно было передать в утилиту `graphviz`) — пользовательский интерфейс данного класса. Остальные методы являются вспомогательными:

`uint8_t height(NodePtr node)` — высота узла `node`;

`int8_t balanceFactor(NodePtr node)` — расчет balance factor для узла `node` (входной узел должен быть ненулевым);

`void fixHeight(NodePtr node)` — поддержание высоты узла `node` в актуальном состоянии (причем левый и правый сын узла `node` должны иметь актуальную высоту);

`NodePtr rotateLeft(NodePtr node)` — левое вращение дерева с корнем `node`, в зависимости от конфигурации дерева производится малое или большое левое вращение.

`NodePtr rotateRight(NodePtr node)` — правое вращение дерева с корнем `node`, в зависимости от конфигурации дерева производится малое или большое правое вращение.

`NodePtr balance(NodePtr node)` — балансировка дерева с корнем `node`.

`NodePtr insertAux(NodePtr node, const T& key)` — вспомогательная функция для вставки элемента `key` в дерево с корнем `node`.

`NodePtr findMin(NodePtr node)` — поиск узла с наименьшим ключом в дереве с корнем `node`.

`NodePtr removeMin(NodePtr node)` — удаление узла с минимальным ключом из дерева с корнем `node`.

`NodePtr removeAux(NodePtr node, const T& key)` — удаление ключа `key` из дерева с корнем `node` (вспомогательный метод для метода `remove`).

`void printInfixAux(NodePtr node, std::ofstream& outFile)` — вспомогательный метод для вывода элементов дерева с корнем `node` в файл `outFile` в порядке возрастания.

`printDotAux(NodePtr node, std::ofstream& dotFile)` — вспомогательный метод для конвертации дерева в валидный DOT-файл.

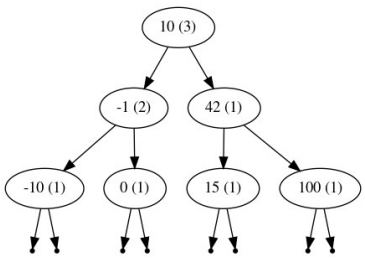
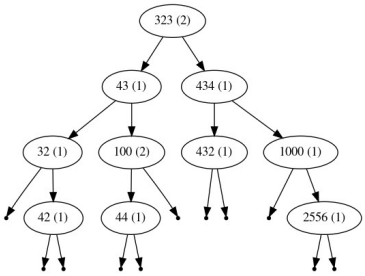
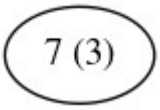
При запуске программы пользователь может выбрать, откуда читать последовательность данных: из консоли или из файла (при выборе консоли пользователь также должен указать количество элементов, которое он собирается ввести). Последовательность элементов в порядке возрастания сохраняется в файле infix_print.txt, а DOT-файл имеет имя output.gv.

Разработанный программный код см. в приложениях А-Г.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	10 -1 10 42 15 -1 0 -10 10 100		Корректная работа программы.
2.	432 43 32 42 100 323 434 100 323 1000 44 2556		Корректная работа программы.
3.	7 7 7		Один (повторяющийся) элемент в последовательности.

Выводы.

Были изучены АВЛ-деревья, их реализация была произведена с помощью языка программирования C++. Также для «красивого» вывода полученного дерева была использована утилита graphviz.

Разработана программа, выполняющая считывание с клавиатуры источник последовательности (консоль или файл) и создающая на основе полученных данных АВЛ-дерево. На выходе программы получаются два файла, в первом хранится последовательность элементов дерева в порядка возрастания, второй представляет собой DOT-файл, соответствующий полученному дереву

ПРИЛОЖЕНИЕ А

КЛАСС УЗЛА

Название файла: node.h

```
#include <cinttypes>
#include <memory>

template<class T>
struct Node {
  using NodePtr = std::shared_ptr<Node<T>>;

  Node(T key);
  ~Node() = default;

  T key;
  uint8_t height = 1;
  NodePtr left = nullptr;
  NodePtr right = nullptr;
};

template<class T>
Node<T>::Node(T key)
    : key(key) {}
```

ПРИЛОЖЕНИЕ Б

КЛАСС АВЛ-ДЕРЕВА (ОБЪЯВЛЕНИЕ)

Название файла: avltree.h

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include <map>

#include "node.h"

template<class T>
class AvlTree {
  using NodePtr = std::shared_ptr<Node<T>>;
public:
  AvlTree() = default;
  ~AvlTree() = default;

  void insert(const T& key);
  void remove(const T& key);

  void printInfix(std::ofstream& outFile) const;
  void printDot(std::ofstream& dotFile) const;

protected:
  uint8_t height(NodePtr node);
  int8_t balanceFactor(NodePtr node);
  void fixHeight(NodePtr node);
  NodePtr rotateLeft(NodePtr node);
  NodePtr rotateRight(NodePtr node);
  NodePtr balance(NodePtr node);

     NodePtr insertAux(NodePtr node, const T& key); // Вспомогательная функция для метода insert

     NodePtr findMin(NodePtr node); // Поиск узла с наименьшим ключом в дереве с корнем node
     NodePtr removeMin(NodePtr node); // Удаление узла с минимальным ключом из дерева с корнем node
     NodePtr removeAux(NodePtr node, const T& key); // Удаление ключа k из дерева с корнем node (вспомогательная функция для метода remove)

  void printInfixAux(NodePtr node, std::ofstream& outFile) const;
  void stepPrint(NodePtr node, uint8_t step) const;
  void printDotAux(NodePtr node, std::ofstream& dotFile) const;

private:
  NodePtr head_ = nullptr;
  std::map<T, size_t> nodeCounter;
};

#include "avltree.cpp"
```

ПРИЛОЖЕНИЕ В

КЛАСС АВЛ-ДЕРЕВА (РЕАЛИЗАЦИЯ)

Название файла: avltree.cpp

```
#include "avltree.h"

template<class T>
void AvlTree<T>::insert(const T& key) {
  head_ = insertAux(head_, key);
}

template<class T>
void AvlTree<T>::remove(const T& key) {
  head_ = removeAux(head_, key);
}

template<class T>
void AvlTree<T>::printInfix(std::ofstream& outFile) const {
  printInfixAux(head_, outFile);
}

template<class T>
void AvlTree<T>::printDot(std::ofstream& dotFile) const {
  dotFile << "digraph AVL_tree {\n";

  // Здесь позже можно настроить узлы и ребра

  if (!head_)
    dotFile << "\n";
  else if (!head_->left && !head_->right)
      dotFile << "   n [label = \"" << head_->key << " (" <<
nodeCounter.at(head_->key) << ")\"]" << ";\n";
  else
    printDotAux(head_, dotFile);

  dotFile << "}\n";
}

template<class T>
uint8_t AvlTree<T>::height(NodePtr node) {
  return node ? node->height : 0;
}

template<class T>
int8_t AvlTree<T>::balanceFactor(NodePtr node) {
  return height(node->right) - height(node->left);
}

template<class T>
void AvlTree<T>::fixHeight(NodePtr node) {
     node->height   =   std::max(height(node->left),   height(node-
>right)) + 1;
}
```

```
template<class T>
typename AvlTree<T>::NodePtr AvlTree<T>::rotateLeft(NodePtr node)
{
  NodePtr newHead(node->right);
  node->right = newHead->left;
  newHead->left = node;
  fixHeight(node);
  fixHeight(newHead);
  return newHead;
}

template<class T>
typename    AvlTree<T>::NodePtr    AvlTree<T>::rotateRight(NodePtr
node) {
  NodePtr newHead(node->left);
  node->left = newHead->right;
  newHead->right = node;
  fixHeight(node);
  fixHeight(newHead);
  return newHead;
}

template<class T>
typename AvlTree<T>::NodePtr AvlTree<T>::balance(NodePtr node) {
  fixHeight(node);
  if (balanceFactor(node) == 2) { // h(right) - h(left) == 2
    if (balanceFactor(node->right) < 0) {
      node->right = rotateRight(node->right);
    }
    return rotateLeft(node);
   } else if (balanceFactor(node) == -2) { // h(right) - h(left)
== -2
    if (balanceFactor(node->left) > 0) {
      node->left = rotateLeft(node->left);
    }
    return rotateRight(node);
  }
  return node; // Балансировка не нужна
}

template<class T>
typename  AvlTree<T>::NodePtr  AvlTree<T>::insertAux(NodePtr  node,
const T& key) {
  if (!node) {
    nodeCounter[key] = 1;
    return std::make_shared<Node<T>>(key);
  }
  if (key < node->key)
    node->left = insertAux(node->left, key);
  else if (key > node->key)
    node->right = insertAux(node->right, key);
  else // Если узел уже имеется в дереве, то учитываем это
    ++nodeCounter.at(key);
  return balance(node);
}
```

```
template<class T>
typename AvlTree<T>::NodePtr AvlTree<T>::findMin(NodePtr node) {
  while (node->left)
    node = node->left;
  return node;
}

template<class T>
typename AvlTree<T>::NodePtr AvlTree<T>::removeMin(NodePtr node)
{
  if (!node->left) {
    return node->right;
  }
  node->left = removeMin(node->left);
  return balance(node);
}

template<class T>
typename AvlTree<T>::NodePtr AvlTree<T>::removeAux(NodePtr node,
const T& key) {
  if (!node)
    return nullptr;
  if (key < node->key) {
    node->left = removeAux(node->left, key);
  } else if (key > node->key) {
    node->right = removeAux(node->right, key);
  } else {
     if (--nodeCounter.at(key) == 0) { // Удаляем узел из дерева
только тогда, когда его счетчик обнулился
      if (!node->right)
        return node->left;
      NodePtr minElem = findMin(node->right);
      minElem->right = removeMin(node->right);
      minElem->left = node->left;
      return balance(minElem);
    }
  }
  return balance(node);
}

template<class T>
void   AvlTree<T>::printInfixAux(NodePtr   node,   std::ofstream&
outFile) const {
  if (node) {
    printInfixAux(node->left, outFile);
    outFile << node->key << " ";
    printInfixAux(node->right, outFile);
  }
}


template<class T>
void    AvlTree<T>::printDotAux(NodePtr    node,    std::ofstream&
dotFile) const {
  static int counter = 0;
  if (!node) {
```

```
            dotFile << "  n" << ++counter << " [ shape = point ];\n";
            return;
        }

        std::tuple<int, int, int> indexes = {-1, -1, -1};
        dotFile << "  n" << ++counter << " [ label = " << "\"" << node-
>key << " (" << nodeCounter.at(node->key) << ")\" ];\n";

        std::get<0>(indexes) = counter;

        std::get<1>(indexes) = counter + 1;
        printDotAux(node->left, dotFile);

        std::get<2>(indexes) = counter + 1;
        printDotAux(node->right, dotFile);

        dotFile << "  n" << std::get<0>(indexes) << " -> {" <<
                                ((std::get<1>(indexes)  >  0)  ?  "n"  +
std::to_string(std::get<1>(indexes)) : "") << " " <<
                                ((std::get<2>(indexes)  >  0)  ?  "n"  +
std::to_string(std::get<2>(indexes)) : "") << "};\n";
    }
```

ПРИЛОЖЕНИЕ Г

ФУНКЦИЯ MAIN

Название файла: main.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include "avltree.h"

enum InputSource {
  Console, File
};

int main() {
  int is;
  AvlTree<int> avlt;
  int elem;

  while (true) {
    std::cout << "Choose source of data (0 - console, 1 - file):"
<< std::endl;
    std::cin >> is;
    if (is == Console) {
      int num;
      std::cout << "Enter number of elements: ";
      std::cin >> num;
      std::cout << "Enter elements:\n";
      for (int i = 0; i < num; ++i) {
        std::cin >> elem;
        avlt.insert(elem);
      }
    } else if (is == File) {
      std::cout << "Enter file name: " << std::endl;
      std::string fileName;
      std::cin >> fileName;
      std::ifstream inFile(fileName);
      if (!inFile.is_open()) {
        std::cerr << "Cannot open a file!" << std::endl;
        return 1;
      }
      inFile >> elem;
      while (!inFile.eof()) {
        avlt.insert(elem);
        inFile >> elem;
      }
    } else {
      std::cerr << "Invalid input!" << std::endl;
      continue;
    }
    break;
  }

  std::ofstream outDotFile("output.gv");
  std::ofstream outTextFile("infix_print.txt");
```

```
  avlt.printDot(outDotFile);
  avlt.printInfix(outTextFile);

  return 0;
}
```