**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра САПР**

**ЛАБОРАТОРНАЯ РАБОТА №3**

**по дисциплине «Алгоритмы и структуры данных»**

| Студенты гр. 2302 |  | Николаев В.Ю. |
| --- | --- | --- |
| Преподаватель |  | Пестерев Д.О. |

Санкт-Петербург

2023

# 1. Постановка задачи

Реализовать двоичное дерево поиска, красно-чёрное дерево и АВЛ-дерево. Сравнить высоты деревьев на случайном наборе входных данных, распределенных случайно. Сравнить временные затраты на балансировку для красно-черного и АВЛ-дерева при удалении элементов, при вставке элементов. Отдельно реализовать функции обхода по дереву: в глубину, в ширину, прямого (preorder), обратного (postorder), симметричный (inorder).

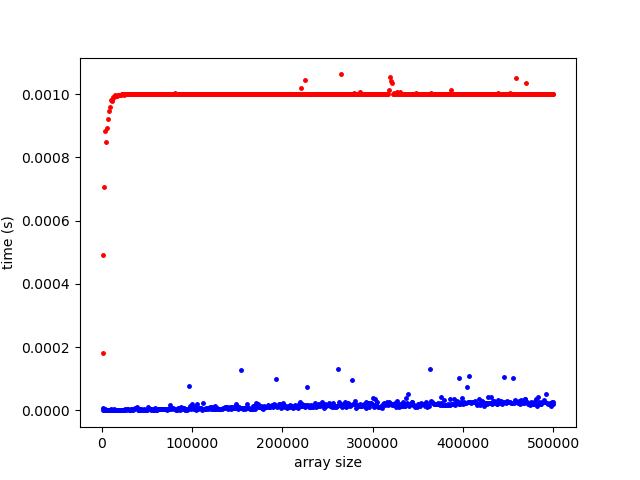
# 2. Высоты деревьев

На графике красными точками обозначена высота дерева двоичного поиска, жёлтыми - высота АВЛ-дерева, синими - высота красно-чёрного дерева.

Как видно из графика дерево двоичного поиска выше остальных деревьев в среднем в 2 раза, а красно-чёрное дерево выше АВЛ-дерева на 1 или 2.

# 3. Время работы балансировок

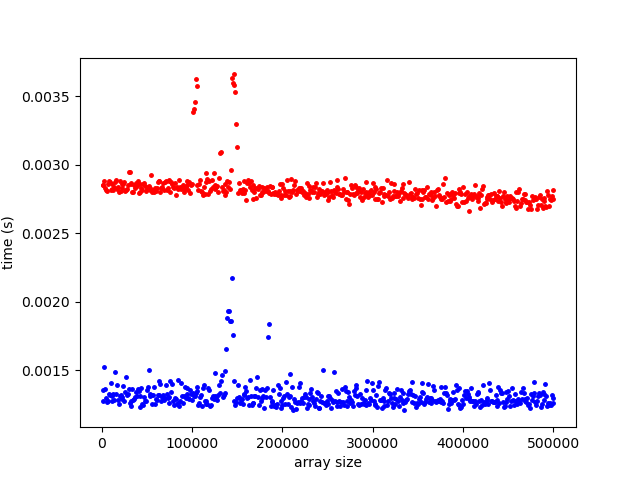
## 3.1. Балансировка после вставки



На графике красными точками обозначено время затраченное на балансировку после добавления 1000 элементов в АВЛ-дерево, синими время затраченное на балансировку после добавления 1000 элементов в красно-чёрное дерево. Так как время затраченное на балансировку дерева после добавления в него одного элемента очень мало каждая точка на графике равна времени затраченному на балансировку после добавления 1000 элементов.

Как видно из графика время, затраченное на балансировку АВЛ-дерева, намного больше времени, затраченного на балансировку красно-чёрного дерева.

## 3.1. Балансировка после удаления



На графике красными точками обозначено время затраченное на балансировку после удаления 1000 элементов в АВЛ-дерево, синими время затраченное на балансировку после удаления 1000 элементов в красно-чёрное дерево. Так как время затраченное на балансировку дерева после удаления в нём одного элемента очень мало каждая точка на графике равна времени затраченному на балансировку после удаления 1000 элементов.

Как видно из графика время, затраченное на балансировку АВЛ-дерева, в среднем в 2 раза больше времени, затраченного на балансировку красно-чёрного дерева.

# 4. Пример работы

|  |
| --- |

# 5. Листинг

| | lib/BST.h #include <iostream>  #include <algorithm>  #include <vector>  #include <string>  template <typename T>  struct BSNode  {  T data;  BSNode \*left;  BSNode \*right;  BSNode \*parent;    BSNode(const T &data):  data(data),  left(nullptr),  right(nullptr),  parent(nullptr)  {}  };  template <typename T>  class BST  {  private:  BSNode<T> \*root;  void insertNodeHelper(BSNode<T> \*&node, const T &data, BSNode<T> \*parent);  BSNode<T> \*searchHelper(BSNode<T> \*node, const T &data);  void deleteNodeHelper(BSNode<T> \*&node, const T &data);  void deleteTree(BSNode<T> \*node);  void BFSHelper(std::vector<BSNode<T> \*> nodes) const;  void DFSHelper(BSNode<T> \*node) const;  void inorderHelper(BSNode<T> \*node) const;  void preorderHelper(BSNode<T> \*node) const;  void postorderHelper(BSNode<T> \*node) const;  void printHelper(int level, BSNode<T> \*node, int mode, char \*modes) const;  int height(BSNode<T> \*node) const;  public:  BST() : root(nullptr) {}  ~BST() { deleteTree(root); }  void insert(const T &data);  void remove(const T &data);  BSNode<T> \*search(const T &data);  void BFS() const;  void DFS() const;  void inorder() const;  void preorder() const;  void postorder() const;  void print() const;  int getSize() const;  int getHeight() const;  };  template <typename T>  void BST<T>::insertNodeHelper(BSNode<T> \*&node, const T &data, BSNode<T> \*parent)  {  if (node == nullptr)  {  node = new BSNode<T>(data);  node->parent = parent;  return;  }  if (data < node->data)  insertNodeHelper(node->left, data, node);  else if (data > node->data)  insertNodeHelper(node->right, data, node);  }  template <typename T>  void BST<T>::deleteNodeHelper(BSNode<T> \*&node, const T &data)  {  if (node == nullptr)  return;  if (data < node->data)  deleteNodeHelper(node->left, data);  else if (data > node->data)  deleteNodeHelper(node->right, data);  else if (node->left == nullptr && node->right == nullptr)  {  if (node->parent != nullptr)  if (node->parent->left == node)  node->parent->left = nullptr;  else  node->parent->right = nullptr;  else  root = nullptr;  delete node;  }  else if (node->left == nullptr)  {  BSNode<T> \*temp = node;  node = node->right;  node->parent = temp->parent;  if (temp->parent != nullptr)  if (temp->parent->left == temp)  temp->parent->left = node;  else  temp->parent->right = node;  else  root = node;  delete temp;  }  else if (node->right == nullptr)  {  BSNode<T> \*temp = node;  node = node->left;  node->parent = temp->parent;  if (temp->parent != nullptr)  if (temp->parent->left == temp)  temp->parent->left = node;  else  temp->parent->right = node;    else  root = node;  delete temp;  }  else  {  BSNode<T> \*temp = node->right;  while (temp->left != nullptr)  temp = temp->left;  node->data = temp->data;  deleteNodeHelper(node->right, temp->data);  }  }  template <typename T>  BSNode<T> \*BST<T>::searchHelper(BSNode<T> \*node, const T &data)  {  if (node == nullptr || node->data == data)  return node;  if (data < node->data)  {  if (node->left != nullptr)  node->left->parent = node;    return searchHelper(node->left, data);  }  if (node->right != nullptr)  node->right->parent = node;    return searchHelper(node->right, data);  }  template <typename T>  void BST<T>::deleteTree(BSNode<T> \*node)  {  if (node == nullptr)  return;  deleteTree(node->left);  deleteTree(node->right);  delete node;  }  template <typename T>  void BST<T>::BFSHelper(std::vector<BSNode<T> \*> nodes) const  {  if (!nodes.size())  return;  std::vector<BSNode<T> \*> next;  for (BSNode<T> \*node : nodes)  {  std::cout << node->data << " ";  if (node->left != nullptr)  next.push\_back(node->left);  if (node->right != nullptr)  next.push\_back(node->right);  }  BFSHelper(next);  }  template <typename T>  void BST<T>::DFSHelper(BSNode<T> \*node) const  {  if (node == nullptr)  return;  std::cout << node->data << " ";  DFSHelper(node->left);  DFSHelper(node->right);  }  template <typename T>  void BST<T>::inorderHelper(BSNode<T> \*node) const  {  if (node == nullptr)  return;  inorderHelper(node->left);  std::cout << node->data << " ";  inorderHelper(node->right);  }  template <typename T>  void BST<T>::preorderHelper(BSNode<T> \*node) const  {  if (node == nullptr)  return;  std::cout << node->data << " ";  preorderHelper(node->left);  preorderHelper(node->right);  }  template <typename T>  void BST<T>::postorderHelper(BSNode<T> \*node) const  {  if (node == nullptr)  return;  postorderHelper(node->left);  postorderHelper(node->right);  std::cout << node->data << " ";  }  template <typename T>  void BST<T>::printHelper(int level, BSNode<T> \*node, int mode, char \*modes) const  {  if (node == nullptr)  return;  modes[level + 1] = 'r';  printHelper(level + 1, node->right, 1, modes);  modes[level] = (mode == 0) ? 'm' : (mode == 1) ? 'v' : 'l';  for (int i = 0; i < level; i++)  if (modes[i] == 'v')  std::cout << "│ ";  else  std::cout << " ";  switch (mode)  {  case 1:  std::cout << "┌─";  break;  case 0:  std::cout << "├─";  break;    case -1:  std::cout << "└─";  break;  }  std::cout << node->data << std::endl;  modes[level + 1] = 'v';  printHelper(level + 1, node->left, -1, modes);  }  template <typename T>  int BST<T>::height(BSNode<T> \*node) const  {  if (node == nullptr)  return 0;    return std::max(height(node->left), height(node->right)) + 1;  }  template <typename T>  void BST<T>::insert(const T &data)  {  insertNodeHelper(root, data, nullptr);  }  template <typename T>  void BST<T>::remove(const T &data)  {  deleteNodeHelper(root, data);  }  template <typename T>  BSNode<T> \*BST<T>::search(const T &data)  {  return searchHelper(root, data);  }  template <typename T>  void BST<T>::BFS() const  {  std::vector<BSNode<T> \*> nodes;  nodes.push\_back(root);  BFSHelper(nodes);  std::cout << std::endl;  }  template <typename T>  void BST<T>::DFS() const  {  DFSHelper(root);  std::cout << std::endl;  }  template <typename T>  void BST<T>::inorder() const  {  inorderHelper(root);  std::cout << std::endl;  }  template <typename T>  void BST<T>::preorder() const  {  preorderHelper(root);  std::cout << std::endl;  }  template <typename T>  void BST<T>::postorder() const  {  postorderHelper(root);  std::cout << std::endl;  }  template <typename T>  void BST<T>::print() const  {  int h = height(root);  char \*modes = new char[h + 1];  modes[0] = 'm';  for (int i = 1; i < h + 1; i++)  modes[i] = 'v';  printHelper(0, root, 0, modes);  delete[] modes;  }  template <typename T>  int BST<T>::getSize() const  {  int size = 0;  std::vector<BSNode<T> \*> queue = {root};  while (!queue.empty())  {  std::vector<BSNode<T> \*> next;  for (BSNode<T> \*node : queue)  {  size++;  if (node->left != nullptr)  next.push\_back(node->left);  if (node->right != nullptr)  next.push\_back(node->right);  }  queue = next;  }  return size;  }  template <typename T>  int BST<T>::getHeight() const  {  return height(root) - 1;  } | | --- |  | lib/RBT.h #include <iostream>  #include <algorithm>  #include <vector>  #include <string>  enum Color { RED, BLACK };  template <typename T>  struct RBNode  {  public:  T data;  Color color;  RBNode \*left;  RBNode \*right;  RBNode \*parent;  RBNode(const T &data):  data(data),  color(RED),  left(nullptr),  right(nullptr),  parent(nullptr)  {}  };  template <typename T>  class RBT  {  private:  RBNode<T> \*root;  void rotateLeft(RBNode<T> \*&);  void rotateRight(RBNode<T> \*&);  void fixViolation(RBNode<T> \*&);  RBNode<T> \*minimumNode(RBNode<T> \*) const;  RBNode<T> \*maximumNode(RBNode<T> \*) const;  RBNode<T> \*searchHelper(RBNode<T> \*, const T &) const;  void deleteTree(RBNode<T>\*);  void deleteNodeHelper(RBNode<T>\*, const T &);  void fixDoubleBlack(RBNode<T>\*);  void BFSHelper(std::vector<RBNode<T> \*>) const;  void DFSHelper(RBNode<T> \*) const;  void inorderHelper(RBNode<T> \*) const;  void preorderHelper(RBNode<T> \*) const;  void postorderHelper(RBNode<T> \*) const;  void printHelper(int, RBNode<T> \*, int, char \*) const;  int height(RBNode<T> \*node) const;  public:  RBT() : root(nullptr) {}  ~RBT() { deleteTree(root); }  void insert(const T&);  void remove(const T&);  RBNode<T> \*search(const T&);  void BFS() const;  void DFS() const;  void inorder() const;  void preorder() const;  void postorder() const;  void print() const;    int getSize() const;  int getHeight() const;  };  template <typename T>  void RBT<T>::rotateLeft(RBNode<T> \*&ptr)  {  RBNode<T> \*rightChild = ptr->right;  ptr->right = rightChild->left;  if (ptr->right != nullptr)  ptr->right->parent = ptr;  rightChild->parent = ptr->parent;  if (ptr->parent == nullptr)  root = rightChild;  else if (ptr == ptr->parent->left)  ptr->parent->left = rightChild;  else  ptr->parent->right = rightChild;  rightChild->left = ptr;  ptr->parent = rightChild;  }  template <typename T>  void RBT<T>::rotateRight(RBNode<T> \*&ptr)  {  RBNode<T> \*leftChild = ptr->left;  ptr->left = leftChild->right;  if (ptr->left != nullptr)  ptr->left->parent = ptr;  leftChild->parent = ptr->parent;  if (ptr->parent == nullptr)  root = leftChild;  else if (ptr == ptr->parent->left)  ptr->parent->left = leftChild;  else  ptr->parent->right = leftChild;  leftChild->right = ptr;  ptr->parent = leftChild;  }  template <typename T>  void RBT<T>::fixViolation(RBNode<T> \*&ptr)  {  RBNode<T> \*parentPtr = nullptr;  RBNode<T> \*grandParentPtr = nullptr;  while ((ptr != root) && (ptr->color != BLACK) && (ptr->parent->color == RED))  {  parentPtr = ptr->parent;  grandParentPtr = ptr->parent->parent;  if (parentPtr == grandParentPtr->left)  {  RBNode<T> \*unclePtr = grandParentPtr->right;  if (unclePtr != nullptr && unclePtr->color == RED)  {  grandParentPtr->color = RED;  parentPtr->color = BLACK;  unclePtr->color = BLACK;  ptr = grandParentPtr;  }  else  {  if (ptr == parentPtr->right)  {  rotateLeft(parentPtr);  ptr = parentPtr;  parentPtr = ptr->parent;  }  rotateRight(grandParentPtr);  std::swap(parentPtr->color, grandParentPtr->color);  ptr = parentPtr;  }  }  else  {  RBNode<T> \*unclePtr = grandParentPtr->left;  if ((unclePtr != nullptr) && (unclePtr->color == RED))  {  grandParentPtr->color = RED;  parentPtr->color = BLACK;  unclePtr->color = BLACK;  ptr = grandParentPtr;  }  else  {  if (ptr == parentPtr->left)  {  rotateRight(parentPtr);  ptr = parentPtr;  parentPtr = ptr->parent;  }  rotateLeft(grandParentPtr);  std::swap(parentPtr->color, grandParentPtr->color);  ptr = parentPtr;  }  }  }  root->color = BLACK;  }  template <typename T>  void RBT<T>::insert(const T &data)  {  RBNode<T> \*newNode = new RBNode<T>(data);  if (root == nullptr)  {  newNode->color = BLACK;  root = newNode;  }  else  {  RBNode<T> \*ptr = root;  RBNode<T> \*parentPtr = nullptr;  while (ptr != nullptr)  {  parentPtr = ptr;  if (newNode->data < ptr->data)  ptr = ptr->left;  else  ptr = ptr->right;  }  newNode->parent = parentPtr;  if (newNode->data < parentPtr->data)  parentPtr->left = newNode;  else  parentPtr->right = newNode;  fixViolation(newNode);  }  }  template <typename T>  void RBT<T>::BFSHelper(std::vector<RBNode<T> \*> queue) const  {  if (queue.empty())  return;  std::vector<RBNode<T>\*> nextQueue;  for (auto node : queue)  {  std::cout << node->data << " ";  if (node->left != nullptr)  nextQueue.push\_back(node->left);  if (node->right != nullptr)  nextQueue.push\_back(node->right);  }  BFSHelper(nextQueue);  }  template <typename T>  void RBT<T>::DFSHelper(RBNode<T> \*ptr) const  {  if (ptr == nullptr)  return;  std::cout << ptr->data << " ";  DFSHelper(ptr->left);  DFSHelper(ptr->right);  }  template <typename T>  void RBT<T>::inorderHelper(RBNode<T> \*ptr) const  {  if (ptr == nullptr)  return;  inorderHelper(ptr->left);  std::cout << ptr->data << " ";  inorderHelper(ptr->right);  }  template <typename T>  void RBT<T>::preorderHelper(RBNode<T> \*ptr) const  {  if (ptr == nullptr)  return;  std::cout << ptr->data << " ";  preorderHelper(ptr->left);  preorderHelper(ptr->right);  }  template <typename T>  void RBT<T>::postorderHelper(RBNode<T> \*ptr) const  {  if (ptr == nullptr)  return;  postorderHelper(ptr->left);  postorderHelper(ptr->right);  std::cout << ptr->data << " ";  }  template <typename T>  void RBT<T>::printHelper(int level, RBNode<T> \*node, int mode, char \*modes) const  {  if (node == nullptr)  return;  modes[level + 1] = 'r';  printHelper(level + 1, node->right, 1, modes);  modes[level] = (mode == 0) ? 'm' : (mode == 1) ? 'v' : 'l';  for (int i = 0; i < level; i++)  if (modes[i] == 'v')  std::cout << "│ ";  else  std::cout << " ";  switch (mode)  {  case 1:  std::cout << "┌─";  break;  case 0:  std::cout << "├─";  break;    case -1:  std::cout << "└─";  break;  }  std::cout << (node->color == RED ? 'r' : 'b') << node->data << ' ' << std::endl;  modes[level + 1] = 'v';  printHelper(level + 1, node->left, -1, modes);  }  template <typename T>  RBNode<T> \*RBT<T>::minimumNode(RBNode<T> \*ptr) const  {  while (ptr->left != nullptr)  ptr = ptr->left;  return ptr;  }  template <typename T>  RBNode<T> \*RBT<T>::maximumNode(RBNode<T> \*ptr) const  {  while (ptr->right != nullptr)  ptr = ptr->right;  return ptr;  }  template <typename T>  RBNode<T> \*RBT<T>::searchHelper(RBNode<T> \*ptr, const T &data) const  {  if (ptr == nullptr || ptr->data == data)  return ptr;  if (data < ptr->data)  return searchHelper(ptr->left, data);  return searchHelper(ptr->right, data);  }  template <typename T>  void RBT<T>::deleteTree(RBNode<T> \*ptr)  {  if (ptr == nullptr)  return;  deleteTree(ptr->left);  deleteTree(ptr->right);  delete ptr;  }  template <typename T>  void RBT<T>::deleteNodeHelper(RBNode<T> \*ptr, const T &data)  {  RBNode<T> \*z = nullptr;  RBNode<T> \*x, \*y;  while (ptr != nullptr)  {  if (ptr->data == data)  z = ptr;  if (ptr->data <= data)  ptr = ptr->right;  else  ptr = ptr->left;  }  if (z == nullptr)  return;  if (z->left == nullptr && z->right == nullptr)  {  y = nullptr;  }  else if (z->left == nullptr || z->right == nullptr)  {  y = z->right == nullptr ? z->left : z->right;  y->parent = z->parent;  }  else  {  y = minimumNode(z->right);  z->data = y->data;  deleteNodeHelper(y, y->data);  return;  }  if (z->color == BLACK)  fixDoubleBlack(z);  if (z->parent == nullptr)  root = y;  else if (z == z->parent->left)  z->parent->left = y;  else  z->parent->right = y;  delete z;  }  template <typename T>  void RBT<T>::fixDoubleBlack(RBNode<T> \*ptr)  {  if (ptr == root)  return;  RBNode<T> \*sibling = nullptr;  RBNode<T> \*parent = ptr->parent;  if (ptr == parent->left)  sibling = parent->right;  else  sibling = parent->left;  if (sibling == nullptr)  fixDoubleBlack(parent);  else  {  if (sibling->color == RED)  {  parent->color = RED;  sibling->color = BLACK;  if (sibling == parent->left)  rotateRight(parent);  else  rotateLeft(parent);  fixDoubleBlack(ptr);  }  else  {  if ((sibling->left != nullptr && sibling->left->color == RED) ||  (sibling->right != nullptr && sibling->right->color == RED))  {  if (sibling->left != nullptr && sibling->left->color == RED)  {  if (sibling == parent->left)  {  sibling->left->color = sibling->color;  sibling->color = parent->color;  rotateRight(parent);  }  else  {  sibling->left->color = parent->color;  rotateRight(sibling);  rotateLeft(parent);  }  }  else  {  if (sibling == parent->left)  {  sibling->right->color = parent->color;  rotateLeft(sibling);  rotateRight(parent);  }  else  {  sibling->right->color = sibling->color;  sibling->color = parent->color;  rotateLeft(parent);  }  }  parent->color = BLACK;  }  else  {  sibling->color = RED;  if (parent->color == BLACK)  fixDoubleBlack(parent);  else  parent->color = BLACK;  }  }  }  }  template <typename T>  void RBT<T>::remove(const T &data)  {  deleteNodeHelper(root, data);  }  template <typename T>  RBNode<T> \*RBT<T>::search(const T &data)  {  return searchHelper(root, data);  }  template <typename T>  void RBT<T>::BFS() const  {  BFSHelper({root});  std::cout << std::endl;  }  template <typename T>  void RBT<T>::DFS() const  {  DFSHelper(root);  std::cout << std::endl;  }  template <typename T>  void RBT<T>::inorder() const  {  inorderHelper(root);  std::cout << std::endl;  }  template <typename T>  void RBT<T>::preorder() const  {  preorderHelper(root);  std::cout << std::endl;  }  template <typename T>  void RBT<T>::postorder() const  {  postorderHelper(root);  std::cout << std::endl;  }  template <typename T>  void RBT<T>::print() const  {  int h = height(root);  char \*modes = new char[h + 1];  modes[0] = 'm';  for (int i = 1; i < h + 1; i++)  modes[i] = 'v';  printHelper(0, root, 0, modes);  delete[] modes;  }  template <typename T>  int RBT<T>::height(RBNode<T> \*node) const  {  if (node == nullptr)  return 1;    return std::max(height(node->left), height(node->right)) + 1;  }  template <typename T>  int RBT<T>::getSize() const  {  int size = 0;  std::vector<RBNode<T> \*> queue = {root};  while (!queue.empty())  {  std::vector<RBNode<T> \*> next;  for (RBNode<T> \*node : queue)  {  size++;  if (node->left != nullptr)  next.push\_back(node->left);  if (node->right != nullptr)  next.push\_back(node->right);  }  queue = next;  }  return size;  }  template <typename T>  int RBT<T>::getHeight() const  {  return height(root);  } | | --- |  | src/CheckingRemove.cpp #include <stdlib.h>  #include <iomanip>  #include <fstream>  #include <time.h>  #include <math.h>  #include "../lib/Tree.h"  int main()  {  std::string fileNames[] =  {  "results/checkingRemove/AVLT.txt",  "results/checkingRemove/RBT.txt"  };  int size = 500000;  int points = 500;  int repeats = 30;  std::vector<int> data(size);  for (int i = 0; i < size; i++)  data[i] = i - size / 2;  for (int i = 0; i < size; i++)  {  int j = i + rand() % (size - i);  std::swap(data[i], data[j]);  }  std::vector<Type> types = { Type::\_AVLT, Type::\_RBT };  for (int typeNum = 0; typeNum < types.size(); typeNum++)  {  Type type = types[typeNum];  std::fstream fout(fileNames[typeNum], std::ios::out);  std::vector<Tree<int>> trees(repeats);  for (auto &tree : trees)  tree.setType(type);  for (auto &tree : trees)  for (auto &i : data)  tree.insert(i);    for (int i = 0; i < points; i++)  fout << size - i \* size / points << " ";  fout << std::endl;  for (int i = 0; i < points; i++)  {  long double time = 0;  for (int j = (i + 1) \* size / points - 1; j >= i \* size / points; j--)  {  long double min = 1e9;  for (auto &tree : trees)  {  clock\_t start = clock();  tree.insert(data[j]);  clock\_t end = clock();    min = std::min(min, (long double)(end - start) / CLOCKS\_PER\_SEC);  }  time += min;  }  fout << std::setprecision(200) << time << " ";  }  fout << std::endl;  fout.close();  }  return 0;  } | | --- | | | lib/AVLT.h #include <iostream>  #include <algorithm>  #include <vector>  #include <string>  template <typename T>  struct AVLNode  {  T data;  AVLNode \*left;  AVLNode \*right;  AVLNode \*parent;  int height;  AVLNode(const T &data):  data(data),  height(1),  left(nullptr),  right(nullptr),  parent(nullptr)  {}  };  template <typename T>  class AVLT  {  private:  AVLNode<T> \*root;  int height(AVLNode<T> \*) const;  int balanceFactor(AVLNode<T> \*) const;  void fixHeight(AVLNode<T> \*);  AVLNode<T> \*rotateLeft(AVLNode<T> \*);  AVLNode<T> \*rotateRight(AVLNode<T> \*);    AVLNode<T> \*insertNodeHelper(AVLNode<T> \*, const T &, AVLNode<T> \*);  AVLNode<T> \*balance(AVLNode<T> \*);  void deleteTree(AVLNode<T> \*);  AVLNode<T> \*deleteNodeHelper(AVLNode<T> \*, const T &);  AVLNode<T> \*deleteMin(AVLNode<T> \*);  AVLNode<T> \*minimumNode(AVLNode<T> \*) const;  AVLNode<T> \*maximumNode(AVLNode<T> \*) const;  AVLNode<T> \*searchHelper(AVLNode<T> \*, const T &) const;  void BFSHelper(std::vector<AVLNode<T> \*>) const;  void DFSHelper(AVLNode<T> \*) const;  void inorderHelper(AVLNode<T> \*) const;  void preorderHelper(AVLNode<T> \*) const;  void postorderHelper(AVLNode<T> \*) const;  void printHelper(int, AVLNode<T> \*, int, char \*) const;  public:  AVLT() : root(nullptr) {}  ~AVLT() { deleteTree(root); }  void insert(const T &data);  void remove(const T &data);  AVLNode<T> \*search(const T &data);  void BFS() const;  void DFS() const;  void inorder() const;  void preorder() const;  void postorder() const;  void print() const;  int getSize() const;  int getHeight() const;  };  template <typename T>  int AVLT<T>::height(AVLNode<T> \*node) const  {  if (node == nullptr)  return 0;    return node->height;  }  template <typename T>  int AVLT<T>::balanceFactor(AVLNode<T> \*node) const  {  if (node == nullptr)  return 0;    return height(node->right) - height(node->left);  }  template <typename T>  void AVLT<T>::fixHeight(AVLNode<T> \*node)  {  int leftHeight = height(node->left);  int rightHeight = height(node->right);  node->height = (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;  }  template <typename T>  AVLNode<T> \*AVLT<T>::rotateRight(AVLNode<T> \*node)  {  AVLNode<T> \*left = node->left;  node->left = left->right;  if (left->right != nullptr)  left->right->parent = node;  left->right = node;  left->parent = node->parent;  node->parent = left;  fixHeight(node);  fixHeight(left);  return left;  }  template <typename T>  AVLNode<T> \*AVLT<T>::rotateLeft(AVLNode<T> \*node)  {  AVLNode<T> \*right = node->right;  node->right = right->left;  if (right->left != nullptr)  right->left->parent = node;    right->left = node;  right->parent = node->parent;  node->parent = right;    fixHeight(node);  fixHeight(right);    return right;  }  template <typename T>  AVLNode<T> \*AVLT<T>::balance(AVLNode<T> \*node)  {  fixHeight(node);  if (balanceFactor(node) == 2)  {  if (balanceFactor(node->right) < 0)  node->right = rotateRight(node->right);  return rotateLeft(node);  }  if (balanceFactor(node) == -2)  {  if (balanceFactor(node->left) > 0)  node->left = rotateLeft(node->left);  return rotateRight(node);  }  return node;  }  template <typename T>  AVLNode<T> \*AVLT<T>::insertNodeHelper(AVLNode<T> \*node, const T &data, AVLNode<T> \*parent)  {  if (node == nullptr)  return new AVLNode<T>(data);  if (data < node->data)  node->left = insertNodeHelper(node->left, data, node);  else  node->right = insertNodeHelper(node->right, data, node);  return balance(node);  }  template <typename T>  AVLNode<T> \*AVLT<T>::minimumNode(AVLNode<T> \*node) const  {  if (node->left == nullptr)  return node;  return minimumNode(node->left);  }  template <typename T>  AVLNode<T> \*AVLT<T>::maximumNode(AVLNode<T> \*node) const  {  if (node->right == nullptr)  return node;  return maximumNode(node->right);  }  template <typename T>  AVLNode<T> \*AVLT<T>::deleteMin(AVLNode<T> \*node)  {  if (node->left == nullptr)  return node->right;  node->left = deleteMin(node->left);  return balance(node);  }  template <typename T>  void AVLT<T>::deleteTree(AVLNode<T> \*node)  {  if (node == nullptr)  return;  deleteTree(node->left);  deleteTree(node->right);  delete node;  }  template <typename T>  AVLNode<T> \*AVLT<T>::deleteNodeHelper(AVLNode<T> \*node, const T &data)  {  if (node == nullptr)  return nullptr;    if (data < node->data)  node->left = deleteNodeHelper(node->left, data);  else if (data > node->data)  node->right = deleteNodeHelper(node->right, data);  else  {  AVLNode<T> \*left = node->left;  AVLNode<T> \*right = node->right;  AVLNode<T> \*parent = node->parent;  delete node;  if (right == nullptr)  {  if (left != nullptr)  left->parent = parent;  return left;  }  AVLNode<T> \*min = minimumNode(right);  min->right = deleteMin(right);  min->left = left;  min->parent = parent;  if (left != nullptr)  left->parent = min;    return balance(min);  }  return balance(node);  }  template <typename T>  AVLNode<T> \*AVLT<T>::searchHelper(AVLNode<T> \*node, const T &data) const  {  if (node == nullptr || node->data == data)  return node;  if (data < node->data)  return searchHelper(node->left, data);    return searchHelper(node->right, data);  }  template <typename T>  void AVLT<T>::BFSHelper(std::vector<AVLNode<T> \*> queue) const  {  if (queue.empty())  return;  std::vector<AVLNode<T>\*> next;  for (AVLNode<T> \*node : queue)  {  std::cout << node->data << " ";  if (node->left != nullptr)  next.push\_back(node->left);  if (node->right != nullptr)  next.push\_back(node->right);  }  BFSHelper(next);  }  template <typename T>  void AVLT<T>::DFSHelper(AVLNode<T> \*node) const  {  if (node == nullptr)  return;  std::cout << node->data << " ";  DFSHelper(node->left);  DFSHelper(node->right);  }  template <typename T>  void AVLT<T>::inorderHelper(AVLNode<T> \*node) const  {  if (node == nullptr)  return;  inorderHelper(node->left);  std::cout << node->data << " ";  inorderHelper(node->right);  }  template <typename T>  void AVLT<T>::preorderHelper(AVLNode<T> \*node) const  {  if (node == nullptr)  return;  std::cout << node->data << " ";  preorderHelper(node->left);  preorderHelper(node->right);  }  template <typename T>  void AVLT<T>::postorderHelper(AVLNode<T> \*node) const  {  if (node == nullptr)  return;  postorderHelper(node->left);  postorderHelper(node->right);  std::cout << node->data << " ";  }  template <typename T>  void AVLT<T>::printHelper(int level, AVLNode<T> \*node, int mode, char \*modes) const  {  if (node == nullptr)  return;  modes[level + 1] = 'r';  printHelper(level + 1, node->right, 1, modes);  modes[level] = (mode == 0) ? 'm' : (mode == 1) ? 'v' : 'l';  for (int i = 0; i < level; i++)  if (modes[i] == 'v')  std::cout << "│ ";  else  std::cout << " ";  switch (mode)  {  case 1:  std::cout << "┌─";  break;  case 0:  std::cout << "├─";  break;    case -1:  std::cout << "└─";  break;  }  std::cout << node->data << std::endl;  modes[level + 1] = 'v';  printHelper(level + 1, node->left, -1, modes);  }  template <typename T>  void AVLT<T>::insert(const T &data)  {  root = insertNodeHelper(root, data, nullptr);  }  template <typename T>  void AVLT<T>::remove(const T &data)  {  root = deleteNodeHelper(root, data);  }  template <typename T>  AVLNode<T> \*AVLT<T>::search(const T &data)  {  return searchHelper(root, data);  }  template <typename T>  void AVLT<T>::BFS() const  {  BFSHelper({root});  std::cout << std::endl;  }  template <typename T>  void AVLT<T>::DFS() const  {  DFSHelper(root);  std::cout << std::endl;  }  template <typename T>  void AVLT<T>::inorder() const  {  inorderHelper(root);  std::cout << std::endl;  }  template <typename T>  void AVLT<T>::preorder() const  {  preorderHelper(root);  std::cout << std::endl;  }  template <typename T>  void AVLT<T>::postorder() const  {  postorderHelper(root);  std::cout << std::endl;  }  template <typename T>  void AVLT<T>::print() const  {  int h = height(root);  char \*modes = new char[h + 1];  modes[0] = 'm';  for (int i = 1; i < h + 1; i++)  modes[i] = 'v';  printHelper(0, root, 0, modes);  delete[] modes;  }  template <typename T>  int AVLT<T>::getSize() const  {  int size = 0;  std::vector<AVLNode<T> \*> queue = {root};  while (!queue.empty())  {  std::vector<AVLNode<T> \*> next;  for (AVLNode<T> \*node : queue)  {  size++;  if (node->left != nullptr)  next.push\_back(node->left);  if (node->right != nullptr)  next.push\_back(node->right);  }  queue = next;  }  return size;  }  template <typename T>  int AVLT<T>::getHeight() const  {  return height(root);  } | | --- |  | lib/Tree.h #include "BST.h"  #include "AVLT.h"  #include "RBT.h"  enum Type { \_BST, \_AVLT, \_RBT };  template <typename T>  struct TNode  {  Type type;    BSNode<T> \*bst;  AVLNode<T> \*avlt;  RBNode<T> \*rbt;  TNode(BSNode<T> \*node)  {  type = Type::\_BST;  bst = node;  }  TNode(AVLNode<T> \*node)  {  type = Type::\_AVLT;  avlt = node;  }  TNode(RBNode<T> \*node)  {  type = Type::\_RBT;  rbt = node;  }  };  template <typename T>  class Tree  {  private:  Type type;  BST<T> \*bst;  AVLT<T> \*avlt;  RBT<T> \*rbt;  public:  Tree() : bst(nullptr), avlt(nullptr), rbt(nullptr) {};  Tree(Type type);  ~Tree();  void insert(const T &data);  void remove(const T &data);  TNode<T> \*search(const T &data) const;  void BFS() const;  void DFS() const;  void inorder() const;  void preorder() const;  void postorder() const;  void print() const;  int getSize() const;  int getHeight() const;  void setType(Type type);  };  template <typename T>  Tree<T>::Tree(Type type):  type(type),  bst(nullptr),  avlt(nullptr),  rbt(nullptr)  {  switch (type)  {  case Type::\_BST:  bst = new BST<T>();  break;  case Type::\_AVLT:  avlt = new AVLT<T>();  break;  case Type::\_RBT:  rbt = new RBT<T>();  break;  }  }  template <typename T>  Tree<T>::~Tree()  {  switch (type)  {  case Type::\_BST:  delete bst;  break;  case Type::\_AVLT:  delete avlt;  break;  case Type::\_RBT:  delete rbt;  break;  }  }  template <typename T>  void Tree<T>::insert(const T &data)  {  switch (type)  {  case Type::\_BST:  bst->insert(data);  break;  case Type::\_AVLT:  avlt->insert(data);  break;  case Type::\_RBT:  rbt->insert(data);  break;  }  }  template <typename T>  void Tree<T>::remove(const T &data)  {  switch (type)  {  case Type::\_BST:  bst->remove(data);  break;  case Type::\_AVLT:  avlt->remove(data);  break;  case Type::\_RBT:  rbt->remove(data);  break;  }  }  template <typename T>  TNode<T> \*Tree<T>::search(const T &data) const  {  switch (type)  {  case Type::\_BST:  return new TNode(bst->search(data));  case Type::\_AVLT:  return new TNode(avlt->search(data));  case Type::\_RBT:  return new TNode(rbt->search(data));  }  return nullptr;  }  template <typename T>  void Tree<T>::BFS() const  {  switch (type)  {  case Type::\_BST:  bst->BFS();  break;  case Type::\_AVLT:  avlt->BFS();  break;  case Type::\_RBT:  rbt->BFS();  break;  }  }  template <typename T>  void Tree<T>::DFS() const  {  switch (type)  {  case Type::\_BST:  bst->DFS();  break;  case Type::\_AVLT:  avlt->DFS();  break;  case Type::\_RBT:  rbt->DFS();  break;  }  }  template <typename T>  void Tree<T>::inorder() const  {  switch (type)  {  case Type::\_BST:  bst->inorder();  break;  case Type::\_AVLT:  avlt->inorder();  break;  case Type::\_RBT:  rbt->inorder();  break;  }  }  template <typename T>  void Tree<T>::preorder() const  {  switch (type)  {  case Type::\_BST:  bst->preorder();  break;  case Type::\_AVLT:  avlt->preorder();  break;  case Type::\_RBT:  rbt->preorder();  break;  }  }  template <typename T>  void Tree<T>::postorder() const  {  switch (type)  {  case Type::\_BST:  bst->postorder();  break;  case Type::\_AVLT:  avlt->postorder();  break;  case Type::\_RBT:  rbt->postorder();  break;  }  }  template <typename T>  void Tree<T>::print() const  {  switch (type)  {  case Type::\_BST:  bst->print();  break;  case Type::\_AVLT:  avlt->print();  break;  case Type::\_RBT:  rbt->print();  break;  }  }  template <typename T>  int Tree<T>::getSize() const  {  switch (type)  {  case Type::\_BST:  return bst->getSize();  case Type::\_AVLT:  return avlt->getSize();  case Type::\_RBT:  return rbt->getSize();  }  return 0;  }  template <typename T>  int Tree<T>::getHeight() const  {  switch (type)  {  case Type::\_BST:  return bst->getHeight();  case Type::\_AVLT:  return avlt->getHeight();  case Type::\_RBT:  return rbt->getHeight();  }  return 0;  }  template <typename T>  void Tree<T>::setType(Type type)  {  this->type = type;  switch (type)  {  case Type::\_BST:  bst = new BST<T>();  break;  case Type::\_AVLT:  avlt = new AVLT<T>();  break;  case Type::\_RBT:  rbt = new RBT<T>();  break;  }  } | | --- |  | src/Example.cpp #include "../lib/Tree.h"  int main()  {  std::cout << "Enter a tree type (BST, AVLT, RBT): ";  std::string treeType;  std::cin >> treeType;  Type type;  if (treeType == "BST")  type = Type::\_BST;  else if (treeType == "AVLT")  type = Type::\_AVLT;  else if (treeType == "RBT")  type = Type::\_RBT;  else  {  std::cout << "Invalid tree type.\n";  return 1;  }  Tree<int> tree(type);  while (true)  {  std::cout << "Enter a command (help, insert, remove, search, getsize, getheight, print, traversal, exit): ";  std::string command;  std::cin >> command;  if (command == "help")  {  std::cout << "insert: insert a value into the tree\n";  std::cout << "remove: remove a value from the tree\n";  std::cout << "search: search for a value in the tree\n";  std::cout << "getsize: get the number of nodes in the tree\n";  std::cout << "getheight: get the height of the tree\n";  std::cout << "print: print the tree in pre-order\n";  std::cout << "traversal: traverse the tree (BFS, DFS, pre, in, post)\n";  std::cout << "exit: exit the program\n";  }  else if (command == "insert")  {  int value;  std::cout << "Enter a value to insert: ";  std::cin >> value;  tree.insert(value);  }  else if (command == "remove")  {  int value;  std::cout << "Enter a value to remove: ";  std::cin >> value;  tree.remove(value);  }  else if (command == "search")  {  int value;  std::cout << "Enter a value to search for: ";  std::cin >> value;  if (tree.search(value) != nullptr)  std::cout << "Value found in tree.\n";  else  std::cout << "Value not found in tree.\n";  }  else if (command == "getsize")  std::cout << "Size: " << tree.getSize() << std::endl;  else if (command == "getheight")  std::cout << "Height: " << tree.getHeight() << std::endl;  else if (command == "print")  {  std::cout << "Tree:\n";  tree.print();  }  else if (command == "traversal")  {  std::cout << "Enter a traversal (BFS, DFS, pre, in, post): ";  std::string traversal;  std::cin >> traversal;  if (traversal == "BFS")  tree.BFS();  else if (traversal == "DFS")  tree.DFS();  else if (traversal == "pre")  tree.preorder();  else if (traversal == "in")  tree.inorder();  else if (traversal == "post")  tree.postorder();  else  std::cout << "Invalid traversal.\n";  }  else if (command == "exit")  break;  else  std::cout << "Invalid command.\n";    }  return 0;  } | | --- |  | src/Plot.py import os  import matplotlib.pyplot as plt  ss = [  "checkingHeight",  "checkingInsert",  "checkingRemove",  ]  for s in ss:  path = "results/" + s  files = os.listdir(path)  x, y, t = [], [], []  for file in files:  f = open(path + "/" + file, "r")  l = [[float(i) for i in line.split()] for line in f.read().splitlines()]  x.append(l[0])  y.append(l[1])  t.append(file)  c = ['b', 'r', 'y']  for i in range(len(x)):  plt.plot(x[i], y[i], c[i] + ".", markersize=5)  plt.xlabel("array size")  if s == "checkingHeight":  plt.ylabel("height of tree")  else:  plt.ylabel("time (s)")    plt.savefig("png/" + s + ".png")  plt.clf()    print(s)  if s == "checkingHeight":  m0, m1, m2 = 0, 0, 0  for i in range(len(x[0])):  m0 += y[0][i] / y[1][i]  m1 += y[1][i] / y[2][i]  m2 += y[2][i] / y[0][i]    m0 /= len(x[0])  m1 /= len(x[0])  m2 /= len(x[0])  print(t[0] + ' / ' + t[1] + ' = ', m0)  print(t[1] + ' / ' + t[2] + ' = ', m1)  print(t[2] + ' / ' + t[0] + ' = ', m2) | | --- |  | src/CheckingHeight.cpp #include <stdlib.h>  #include <fstream>  #include <math.h>  #include "../lib/Tree.h"  int main()  {  std::string fileNames[] =  {  "results/checkingHeight/BST.txt",  "results/checkingHeight/AVLT.txt",  "results/checkingHeight/RBT.txt"  };  int size = 500000;  int points = 500;  std::vector<int> data(size);  for (int i = 0; i < size; i++)  data[i] = i - size / 2;  for (int i = 0; i < size; i++)  {  int j = i + rand() % (size - i);  std::swap(data[i], data[j]);  }  std::vector<Type> types = { Type::\_BST, Type::\_AVLT, Type::\_RBT };  for (int typeNum = 0; typeNum < types.size(); typeNum++)  {  Type type = types[typeNum];  std::fstream fout(fileNames[typeNum], std::ios::out);  Tree<int> tree(type);  for (int i = 0; i < points; i++)  fout << (i + 1) \* size / points << " ";  fout << std::endl;  for (int i = 0; i < points; i++)  {  for (int j = i \* size / points; j < (i + 1) \* size / points; j++)  tree.insert(data[j]);  fout << tree.getHeight() << " ";  }  fout << std::endl;  fout.close();  }  return 0;  } | | --- |  | src/CheckingInsert.cpp #include <stdlib.h>  #include <iomanip>  #include <fstream>  #include <time.h>  #include <math.h>  #include "../lib/Tree.h"  int main()  {  std::string fileNames[] =  {  "results/checkingInsert/AVLT.txt",  "results/checkingInsert/RBT.txt"  };  int size = 500000;  int points = 500;  int repeats = 30;  std::vector<int> data(size);  for (int i = 0; i < size; i++)  data[i] = i - size / 2;  for (int i = 0; i < size; i++)  {  int j = i + rand() % (size - i);  std::swap(data[i], data[j]);  }  std::vector<Type> types = { Type::\_AVLT, Type::\_RBT };  for (int typeNum = 0; typeNum < types.size(); typeNum++)  {  Type type = types[typeNum];  std::fstream fout(fileNames[typeNum], std::ios::out);  std::vector<Tree<int>> trees(repeats);  for (auto &tree : trees)  tree.setType(type);  for (int i = 0; i < points; i++)  fout << (i + 1) \* size / points << " ";  fout << std::endl;  for (int i = 0; i < points; i++)  {  long double time = 0;  for (int j = i \* size / points; j < (i + 1) \* size / points; j++)  {  long double min = 1e9;  for (auto &tree : trees)  {  clock\_t start = clock();  tree.insert(data[j]);  clock\_t end = clock();    min = std::min(min, (long double)(end - start) / CLOCKS\_PER\_SEC);  }  time += min;  }  fout << std::setprecision(200) << time << " ";  }  fout << std::endl;  fout.close();  }  return 0;  } | | --- | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |