|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | МИНОБРНАУКИ РОССИИ | | | | Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |   Институт Информационных технологий | |
|  | |
| Кафедра Математического обеспечения и стандартизации информационных технологий | |
|  | |
|  | |

|  |  |
| --- | --- |
| **ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 5** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  **Тема: «Сбалансированные деревья поиска (СДП) и их применение для поиска данных в файле»** | |
|  | |
| Выполнил студент группы ИКБО-10-21 | Смольников А.Б. |
| Принял преподаватель | Филатов А.С. |

|  |  |  |
| --- | --- | --- |
| Лабораторная работа выполнена | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись студента)* |
|  |  |  |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись руководителя)* |

Москва 2022

# **Цель работы**

* получить навыки в разработки и реализации алгоритмов управления бинарным деревом поиска и сбалансированными бинарными деревьями поиска (АВЛ – деревьями);
* получить навыки в применении файловых потоков прямого доступа к данным файла;
* получить навыки в применении сбалансированного дерева поиска для прямого доступа к записям файла.

# **Постановка задачи**

* 1. **Задача 1**

Вариант 22. Вид дерева: АВЛ

1. Разработать класс управления файлом. Включить методы: создание двоичного файла записей фиксированной длины из заранее подготовленных данных в текстовом файле; поиск и вывод записи в файле с помощью линейного поиска; вывод записи по известному адресу в файле, добавление записи в файл. Структура записи файла определена индивидуальным вариантом задания.
2. Разработать класс «Бинарное дерево поиска». Структура информационной части узла дерева включает ключ и ссылку на запись в файле. Разработать основные методы: построение дерева по существующему файлу; добавление элемента в дерево; поиск по ключу в дереве; удаление элемента из дерева; вывод дерева в форме дерева.
3. Разработать класс «Сбалансированное дерево поиска». Структура информационной части узла дерева включает ключ и ссылку на запись в файле. Разработать основные методы: построение дерева по существующему файлу; добавление элемента в дерево; поиск по ключу в дереве; удаление элемента из дерева; вывод дерева в форме дерева.
4. Разработать приложение, демонстрирующее выполнение всех методов и поиск записей в файле с помощью БДП, СДП и линейного поиска.
5. Провести анализ времени выполнения поиска записей, находящихся в начале, середине и конце большого файла тремя способами.
6. Определить среднее число выполненных поворотов (число поворотов на общее число вставленных ключей) при включении ключей в СДП при его формировании из двоичного файла.
7. Выполнить тестирование.
8. Составить отчет, отобразив в нем описание выполнения всех этапов разработки, тестирования и код всей программы со скриншотами результатов тестирования.

Таблица 1. Задания варианта №22

|  |  |
| --- | --- |
| Задание 1 | Вид дерева: AVL  Страны: название, количество граждан, факт членства в ООН (bool) |

# **Решение**

* 1. **Теоретическое введение**

Нелинейные структуры данных – такие структуры данных, отображающие сложные отношения между элементами структуры. Пример нелинейной структуры данных граф, и дерево, в частности.

Граф - топологическая модель, состоящая из множества вершин и множества ребер, соединяющих эти вершины. Дерево – связный граф, в котором нет циклов. Это означает, что из любой вершины можно попасть в любую, притом единственным путем. Деревья часто используются для организации иерархической структуры данных, например, при создании двоичных деревьев поиска или кучи, в этом случае одну вершину дерева называют корнем. Лист – вершина, не имеющая потомков. Обход дерева – систематический просмотр всех вершин, при котором каждая вершина встречается один раз.

Бинарное дерево – то дерево, в котором каждая вершина имеет два потомка (правые и левые поддеревья – left и right).

Идеально сбалансированное дерево – то дерево, в котором высоты поддеревьев отличаются не более, чем на 1.

Дерево выражений - такое дерево, которое в своем корне хранит число (если операторов нет), или оператор, связные вершины на расстоянии 1 хранят операторы (оператором может быть выражение). При этом каждое поддерево данного дерева является деревом выражений.

Бинарное дерево поиска – такое бинарное дерево, которое в левом узле хранит значение, меньшее значения потомка, а в правом – значение, большее значения потомка. Данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

Обход дерева – алгоритм, в котором рассматриваются все вершины дерева.

Обход в глубину – рекурсивный метод обхода, в котором посещается корень дерева, левое поддерево и правое поддерево (в нужном порядке).

Обход в ширину – метод обхода, в котором используется очередь. Начиная с корня добавляем текущий узел в конец стека, далее пока очередь не кончится, итеративно посещаем каждую вершину из начала очереди, добавляя в ее конец дочерние вершины.

АВЛ дерево – такое бинарное дерево поиска, в котором для каждой вершины высота двух его поддеревьев отличается не более чем на 1, причем левый дочерний узел меньше родительского, а правый – больше. Узел дерева хранит ключ, значение, указатели на поддеревья, а также разницу высот левого и правого поддерева. Когда происходит разбалансировка дерева, выполняется специальный алгоритм – левый или правый поворот, чтобы сбалансировать дерево, не потеряв свойств соединенных элементов.

* 1. **Функции задания №1**

Для организации записей в файле используется структура Note, состоящая из поля названия страны длиной 45 символов, целочисленного поля населения, а также поля типа bool, хранящего факт членства в ООН:

|  |
| --- |
| struct Note {  char name[45];  int people;  bool Uno;  }; |

Для взаимодействия с бинарным файлом используется класс binfile, состоящий из методов конвертации, поиска, прямого доступа, а также добавления в конец файла.

|  |
| --- |
| struct binfile{  static void textToBin( std::string text, std::string bin);  static int searchBin( std::string bin, std::string name);  static Note directAccessBin( std::string bin, int index);  static void addBin( std::string bin, Note note);  }; |

Метод textToBin() получает на вход имя текстового файла, имя бинарного фалйа и конвертирует информации из текстового файла в бинарный при помощи разделения строки с пробелами, дальнейшем заполнении полей структуры и записи ее в файл.

|  |
| --- |
| void binfile::textToBin(std::string text, std::string bin){  std::ifstream fin(text, std::ios::in);  std::ofstream fout(bin, std::ios::binary | std::ios::out);  std::string line;  while(std::getline(fin, line)){  Note note;  std::stringstream ss(std::ios::in | std::ios::out);  ss << line;  ss >> note.name >> note.people >> note.Uno;  fout.write((char\*)&note, sizeNote);  }  fin.close();  fout.close();  } |

Метод searchBin() выполняет линейный поиск необходимой записи в файле при помощи полного перебора с самого начала

|  |
| --- |
| int binfile::searchBin( std::string bin, std::string name){  std::ifstream fin(bin, std::ios::binary | std::ios::in);  Note note;  int i=0;  while(!fin.eof()){  fin.read((char\*)&note, sizeNote);  if(strcmp(note.name, name.c\_str())!=0){  printNoteNative(note);  return i;  }  i++;  }  fin.close();  return -1;  } |

Метод directAccessBin() получает запись из бинарного файла при помощи прямого доступа по номеру строки, используются функции seekg и read.

|  |
| --- |
| Note binfile::directAccessBin( std::string bin, int index){  std::ifstream fin(bin, std::ios::binary | std::ios::in);  Note note;  fin.seekg(index\*sizeNote);  fin.read((char\*)&note, sizeNote);  printNoteNative(note);  fin.close();  return note;  } |

Метод addBin() добавляет запись в конец файла при помощи флага ios::app и функции write

|  |
| --- |
| void binfile::addBin(std::string bin, Note note){  std::ofstream fout(bin, std::ios::binary | std::ios::app);  fout.write((char\*)&note, sizeNote);  fout.close();  } |

Для управления бинарным деревом реализуем класс binTree, содержащий структуру записи, состоящую из ключа, значения, указателей на дочерние элементы и высоту дерева, указатель на корень и размер. В методах реализованы все необходимые алгоритмы в соответствии с постановкой задачи, они представлены ниже.

|  |
| --- |
| class binTree{  public:  struct Node{  char key[45];  int index;  Node\* left;  Node\* right;  int height;  };  Node\* root=nullptr;  int size;  int height(Node\* p);  int fixheight(binTree::Node \*p);  void BuildByFile( std::string bin);  void addNode(Node\* &node, char\* key, int index);  void deleteByKey(Node\* &node, char\* key);  int searchByKey(Node\* node, char\* key);  void directAccess(Node\* node, int index);  void print(Node\* node, std::string offset);  void printTree(Node \*element= nullptr, std::string prefix = "", bool root = true);  }; |

Метод height() является служебным, выполняющий необходимые действия по определению высоты деревьев (в том числе и пустых)

|  |
| --- |
| int binTree::height(binTree::Node \*p) {  return p ? p->height : 0;  } |

Метод fixheight() определяет правильное значение поля height, используя вспомогательный метод для избежания обращения к нулевым указателям

|  |
| --- |
| int binTree::fixheight(binTree::Node \*p) {  int hl = height(p->left);  int hr = height(p->right);  return (hl > hr ? hl : hr) + 1;  } |

Метод buildByFile() строит бинарное дерево по бинарному файлу при помощи итеративного добавления элементов в дерево

|  |
| --- |
| void binTree::BuildByFile(std::string bin){  std::ifstream file(bin, std::ios::binary | std::ios::in);  if(!file.is\_open()){  std::cout << "File not found" << std::endl;  return;  }  Note note;  int i=0;  while(file.read((char\*)&note, sizeNote)){  addNode(root, note.name, i);  i++;  }  file.close();  } |

Метод addNode() принимает на вход корень, ключ и значение и добавляет элемент в дерево используя лексикографическое сравнение строк, предварительно проверяя указатель на корень дерева на значение nullptr

|  |
| --- |
| void binTree::addNode(Node\* &node, char\* key, int index){  if(node==nullptr){  node = new Node;  strcpy(node->key, key);  node->index = index;  node->left = nullptr;  node->right = nullptr;  node->height = fixheight(node);  size++;  }else if(strcmp(key, node->key)<0){  addNode(node->left, key, index);  }else if(strcmp(key, node->key)>0){  addNode(node->right, key, index);  }  } |

Метод deleteByKey() принимет на вход корень и ключ, постепенно обходит дерево в глубину, пока не не найдется совпадение ключа и удаляет элемент, сохраняя дочерние при их наличии.

|  |
| --- |
| void binTree::deleteByKey(Node\* &node, char\* key){  if(node==nullptr){  return;  }else if(strcmp(key, node->key)<0){  deleteByKey(node->left, key);  }else if(strcmp(key, node->key)>0){  deleteByKey(node->right, key);  }else{  if(node->left==nullptr && node->right==nullptr){  delete node;  node = nullptr;  }else if(node->left==nullptr){  Node\* temp = node;  node = node->right;  delete temp;  }else if(node->right==nullptr){  Node\* temp = node;  node = node->left;  delete temp;  }else{  Node\* temp = node->right;  while(temp->left!=nullptr){  temp = temp->left;  }  strcpy(node->key, temp->key);  node->index = temp->index;  deleteByKey(node->right, temp->key);  }  }  } |

Метод searchByKey() выполняет поиск элемента, обходя дерево в глубину пока не найдется совпадение, выводит на экран запись и возвращает ее индекс.

|  |
| --- |
| int binTree::searchByKey(Node\* node, char\* key){  if(node==nullptr){  std::cout << "Not found" << std::endl;  return -1;  }else if(strcmp(key, node->key)<0){  searchByKey(node->left, key);  }else if(strcmp(key, node->key)>0){  searchByKey(node->right, key);  }else{  std::cout << "Found" << node->key << " at "<< node->index << std::endl;  return node->index;  }  } |

Метод printTree() принимет на вход указатель на элемент, префикс и признак корня изначального дерева, выводит дерево, повернув его против часовой стрелки на 90 градусов. Используется система префиксирования для понятного вывода дерева.

|  |
| --- |
| void binTree::printTree(Node \*element , std::string prefix, bool root) {  if (root) {  element = this->root;  }  if(!element) return;  if(element->right){  printTree(element->right, prefix + (root ? " " : "│ "), false);  }  std::cout << prefix << (root ? "└── " : "├── ") << element->key<<" "<<element->index<<std::endl;  if(element->left){  printTree(element->left, prefix + (root ? " " : "│ "), false);  }  } |

Для реализации AVL дерева используется специальный класс AVLTree, содержащий в себе аналогичную структуру, поля и методы. Отличием являются только методы балансировки и поворотов.

|  |
| --- |
| class AVLTree{  public:  struct Node{  char key[45];  int index;  Node\* left;  Node\* right;  int height;  };  Node\* root=nullptr;  int size;  int rotations=0;  void BuildByFile(std::string bin);  int height(Node\* p);  int bfactor(Node\* p);  void fixheight(Node\* p);  Node\* rotateright(Node\* p);  Node\* rotateleft(Node\* q);  Node\* balance(Node\* p);  Node\* insert(Node\* p, char\* k, int index);  Node\* findMin(Node\* p);  Node\* removeMin(Node\* p);  Node\* deleteNode(Node\* p, char\* k);  int search(char\* k);  void print(Node\* p, std::string prefix);  void printTree(Node \*element= nullptr, std::string prefix = "", bool root = true);  }; |

Метод BuildByFile() строит бинарное дерево по бинарному файлу при помощи итеративного добавления элементов в дерево.

|  |
| --- |
| void AVLTree::BuildByFile(std::string bin) {  std::ifstream fin(bin, std::ios::binary | std::ios::in);  Note note;  int i=0;  while(!fin.eof()){  fin.read((char\*)&note, sizeNote);  root = insert(root, note.name, i);  i++;  }  fin.close();  } |

Метод height() является служебным, выполняющий необходимые действия по определению высоты деревьев (в том числе и пустых)

|  |
| --- |
| int AVLTree::height(Node \*p) {  return p ? p->height : 0;  } |

Метод bfactor() является служебным и возвращает разницу высот дочерних поддеревьев узла, переданного в метод

|  |
| --- |
| int AVLTree::bfactor(Node \*p) {  return height(p->right) - height(p->left);  } |

Метод fixheight() определяет правильное значение поля height, используя вспомогательный метод для избежания обращения к нулевым указателям.

|  |
| --- |
| void AVLTree::fixheight(AVLTree::Node \*p) {  int hl = height(p->left);  int hr = height(p->right);  p->height = (hl > hr ? hl : hr) + 1;  } |

Метод insert() принмает на вход указатель на корень, ключ и значение. Если указатель не существует, то инициализируем его, иначе обойдем дерево в глубну, лексикографически сравнивая ключи, добавим элемент и вернем сбалансированную версию дерева.

|  |
| --- |
| AVLTree::Node\* AVLTree::insert(AVLTree::Node \*p, char \*k, int index) {  if(!p){  p = new Node;  strcpy(p->key, k);  p->index = index;  strcpy(p->key, k);  p->left = p->right = nullptr;  p->height = 1;  size++;  return p;  }  if(strcmp(k, p->key) < 0){  p->left = insert(p->left, k, index);  } else{  p->right = insert(p->right, k, index);  }  return balance(p);  } |

Метод balance() принимает на вход корень дерева и при помощи определения разницы высот выполняет один из больших поворотов дерева: левый или правый. В результате дерево балансируется и возвращается указатель.

|  |
| --- |
| AVLTree::Node\* AVLTree::balance(AVLTree::Node\* p){  fixheight(p);  //выполение одного из больших поворотов  if(bfactor(p) == 2){  if(bfactor(p->right) < 0)  p->right = rotateright(p->right);  return rotateleft(p);  }  if(bfactor(p) == -2){  if(bfactor(p->left) > 0)  p->left = rotateleft(p->left);  return rotateright(p);  }  return p;  } |

Метод rotateright() принимает на вход корень дерева и выполняет правый поворот, обновляя высоты узлов.

|  |
| --- |
| AVLTree::Node\* AVLTree::rotateright(AVLTree::Node \*p) {  Node\* q = p->left;  p->left = q->right;  q->right = p;  fixheight(p);  fixheight(q);  rotations++;  return q;  } |

Метод rotateleft() принимает на вход корень дерева и выполняет левый поворот, обновляя высоты узлов.

|  |
| --- |
| AVLTree::Node\* AVLTree::rotateleft(AVLTree::Node \*q) {  Node\* p = q->right;  q->right = p->left;  p->left = q;  fixheight(q);  fixheight(p);  rotations++;  return p;  } |

Метод printTree() принимет на вход указатель на элемент, префикс и признак корня изначального дерева, выводит дерево, повернув его против часовой стрелки на 90 градусов. Используется система префиксирования для понятного вывода дерева.

|  |
| --- |
| void AVLTree::printTree(Node \*element , std::string prefix, bool root) {  if (root) {  element = this->root;  }  if(!element) return;  if(element->right){  printTree(element->right, prefix + (root ? " " : "│ "), false);  }  std::cout << prefix << (root ? "└── " : "├── ") << element->key<</\* " ("<< height(element) << ")"<<\*/std::endl;  if(element->left){  printTree(element->left, prefix + (root ? " " : "│ "), false);  }  } |

Метод findMin() принимает на вход корень дерева, находит наименьший узел дерева и возвращает указатель на него

|  |
| --- |
| AVLTree::Node\* AVLTree::findMin(Node\* p){  return p->left ? findMin(p->left) : p;  } |

Метод removeMin() принимает на вход корень дерева, находит наименьший узел дерева, удаляет его и балансирует дерево

|  |
| --- |
| AVLTree::Node\* AVLTree::removeMin(Node\* p){  if(p->left == 0)  return p->right;  p->left = removeMin(p->left);  return balance(p);  } |

Метод deleteNode() принимает на вход корень дерева, ключ. Находит узел p с заданным ключом k (если не находим, то делать ничего не надо), в поддереве находит узел min с наименьшим ключом и заменяет удаляемый узел p на найденный узел min. Возвращает сбалансированное дерево.

|  |
| --- |
| AVLTree::Node\* AVLTree::deleteNode(AVLTree::Node \*p, char \*k) {  if(!p) return nullptr;  if(strcmp(k, p->key) < 0){  p->left = deleteNode(p->left, k);  } else if(strcmp(k, p->key) > 0){  p->right = deleteNode(p->right, k);  } else{  Node\* q = p->left;  Node\* r = p->right;  delete p;  if(!r) return q;  Node\* min = findMin(r);  min->right = removeMin(r);  min->left = q;  return balance(min);  }  return balance(p);  } |

Метод search() выполняет поиск узла с заданным ключом, обходя дерево в глубину. Если элемент не найдет, возвращается -1.

|  |
| --- |
| int AVLTree::search(char \*k) {  Node\* p = root;  while(p){  if(strcmp(k, p->key) < 0){  p = p->left;  } else if(strcmp(k, p->key) > 0){  p = p->right;  } else{  return p->index;  }  }  return -1;  } |

* 1. **Интерфейс**

При запуске программы пользователю предоставляется выбор одного из предложенных заданий, указанных в постановке задачи. Любой другой введенный символ спровоцирует завершение программы.

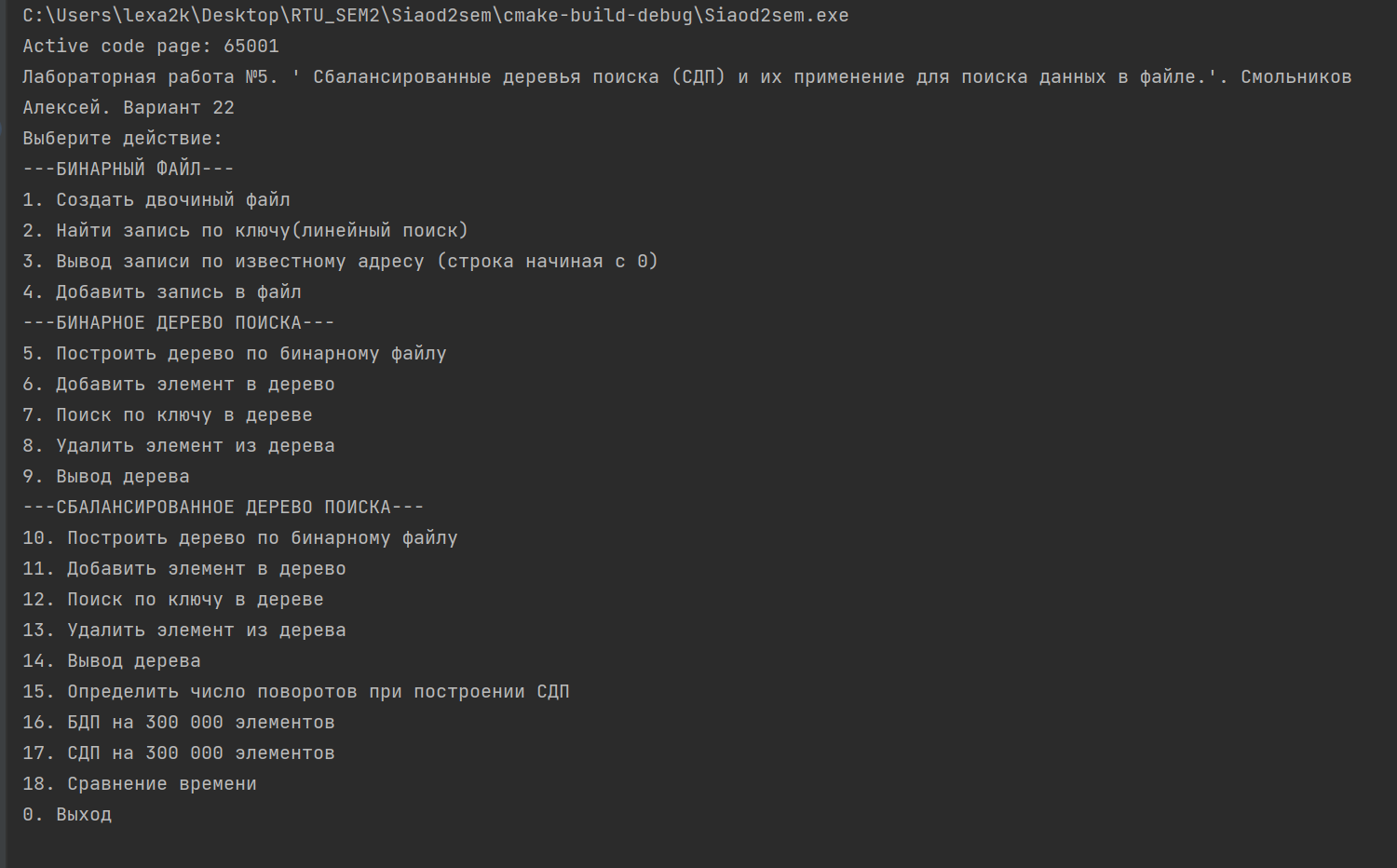


Рисунок 1. Интерфейс программы

# **Тестирование**

Пункт 1 отвечает за конвертацию файла из текста в бинарный файл, используя заранее разработанную структуру:

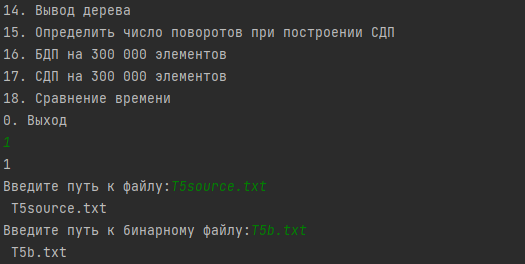


Рисунок 2. Интерфейс конвертации файла

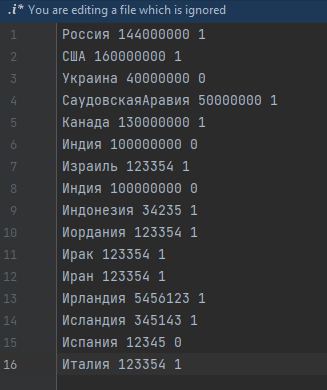


Рисунок 3. Исходный файл

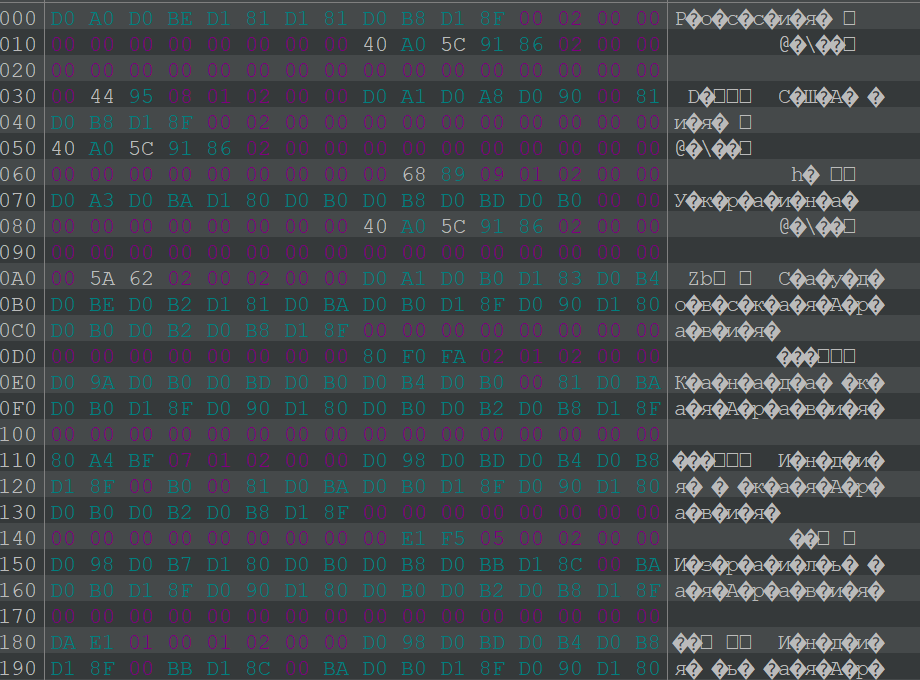


Рисунок 4. Сгенерированный бинарный файл

Пункт 2 выполняет линейный поиск в бинарном файле. Из рисунка 3 видно, что программа работает корректно:

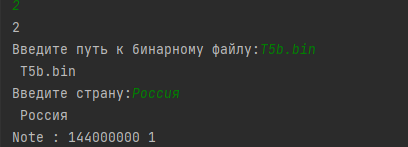


Рисунок 5. Линейный поиск

Пункт меню 3 выполняет прямой доступ к записи в файле по ее порядковому номеру. Из рисунка 3 видно, что программа работает корректно:

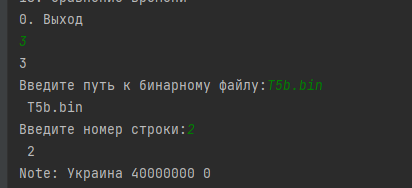


Рисунок 6. Прямой доступ

Пункт меню 4 добавляет запись в бинарный файл. Из рисунков 7 и 8 видно, что добавление работает корректно:

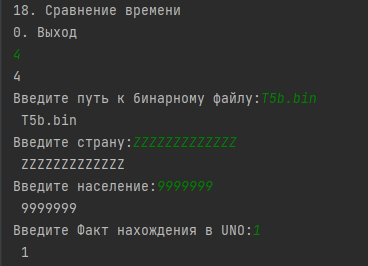


Рисунок 7. Добавление записи

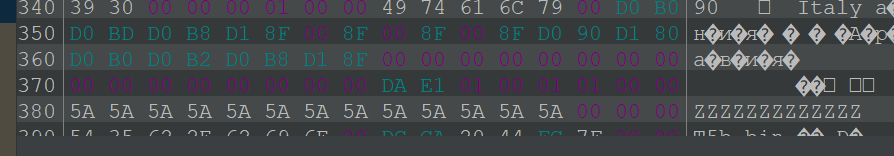


Рисунок 8. Добавленная запись

Пункт под номером 5 открывает часть взаимодействия с бинарным деревом и отвечает за построение дерева по бинарному файлу. Из рисунка 9 видно, что дерево создается корректно. Также данный рисунок показывает правильность выполнения вывода дерева в консоль (9 пункт меню):

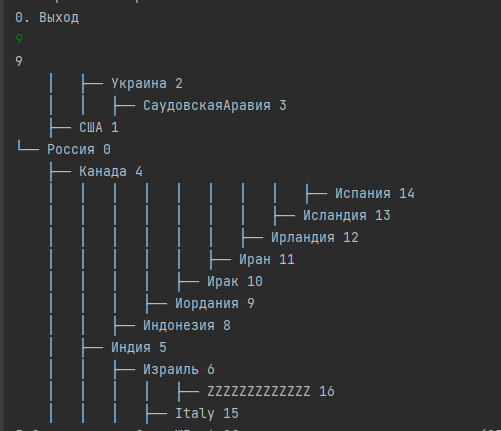


Рисунок 9. Выведенное дерево

Пункт меню 6 добавляет в дерево новую страну, в соответствии с рисунком 10 операция выполняется корректно (дерево не пересоздавалось с момента запечатления рисунка 9)

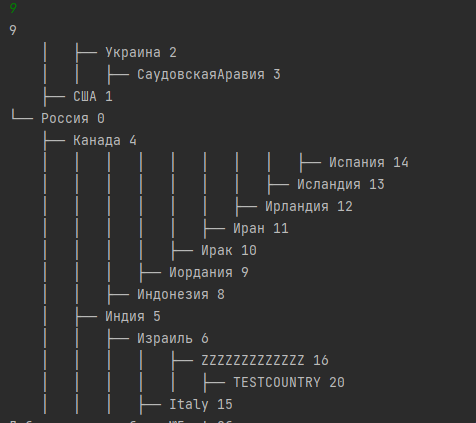


Рисунок 10. Добавление элемента в дерево

Пункт меню под номером 7 выполняет поиск в дереве по ключу, обходя его в глубину. Из рисунков 11, 10 видна правильность выполнения:

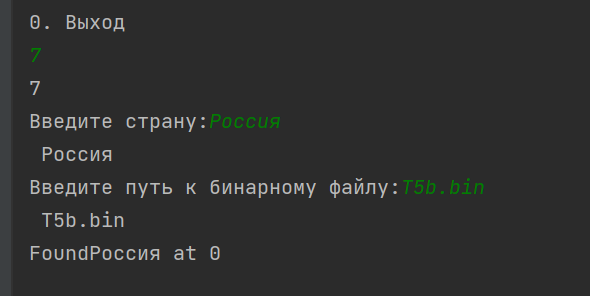


Рисунок 11. Поиск в бинарном дереве

Пункт 8 отвечает за удаление записи из дерева, в соответствии с рисунками 10, 12 удаление происходит верно (удаляется “Italy”, узел “TESTCOUNTRY” пропал из за перезапуска программы)

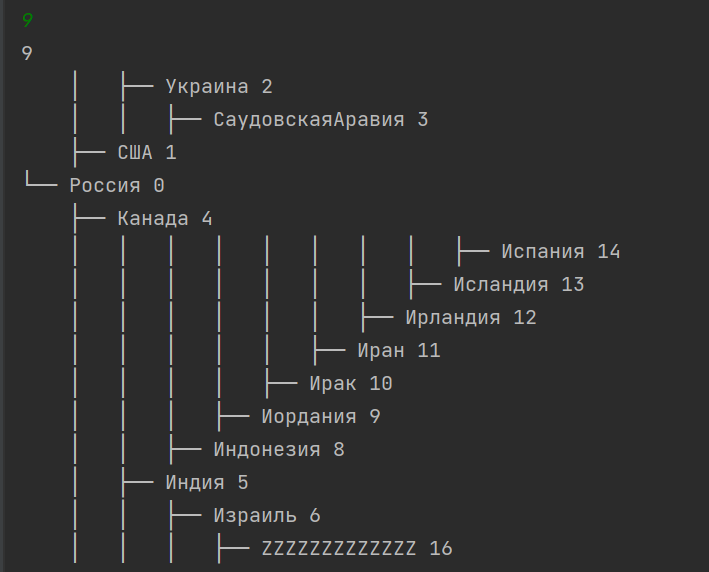


Рисунок 12. Удаление из бинарного дерева

Пункт 10 отвечает за создание AVL дерева из бинарного файла. Исходя из определения AVL дерева, последнее стоится правильно. В соответствии с рисунком 13 создание происходит корректно. Сразу же тестируем вывод дерева в консоль (пункт 14):

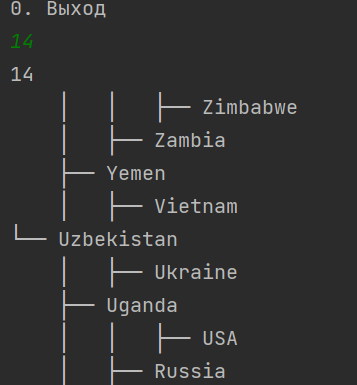


Рисунок 13. Построенное AVL дерево

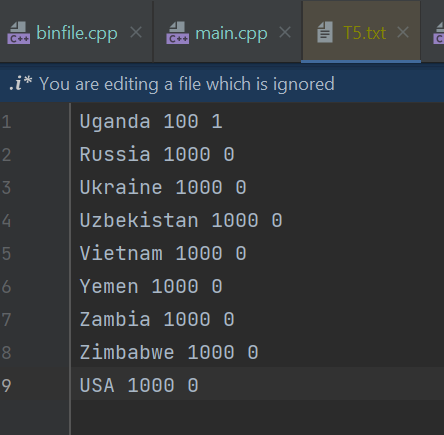


Рисунок 14. Исходный файл

Пункт меню 11 добавляет запись файл, из рисунков 15 и 16 отчетливо видно, что операция проходит корректно:

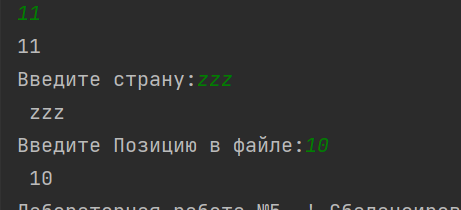


Рисунок 15. Интерфейс добавления узла

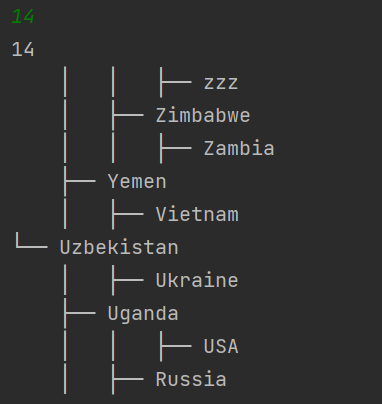


Рисунок 16. Добавленный узел

Пункт 12 отвечает за поиск элемента в дереве. Из рисунков 14, 16, 17 видно, что поиск совершается правильно и возвращается истинный индекс записи в файле:

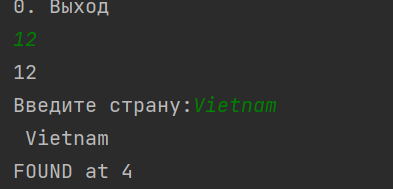


Рисунок 17. Поиск в AVL дереве по ключу

Пункт 13 отвечает за удаление узла из дерева, в соответствии с рисунками 16, 18, 19 удаление происходит корректно

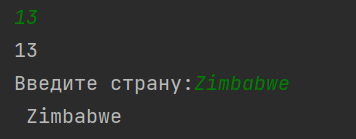


Рисунок 18. Удаление страны по ключу

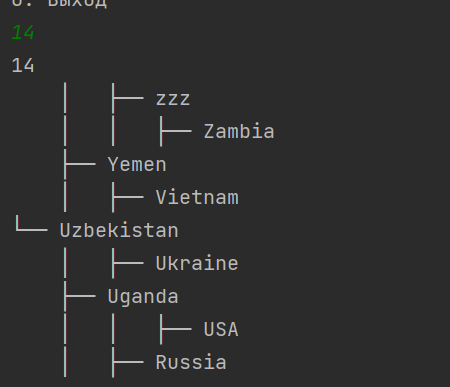


Рисунок 19. Результат удаления узла из дерева

В пункте 15 определим число поворотов при построении СДП из 300 000 элементов. На рисунке 20 видно, что число поворотов примерно на 30% меньше исходного числа записей, однако повороты выполняются за O(1), так что эта мера вполне оправдана в перспективе многократного поиска элементов. Среднее число поворотов – 0,6996 на вставленный элемент:

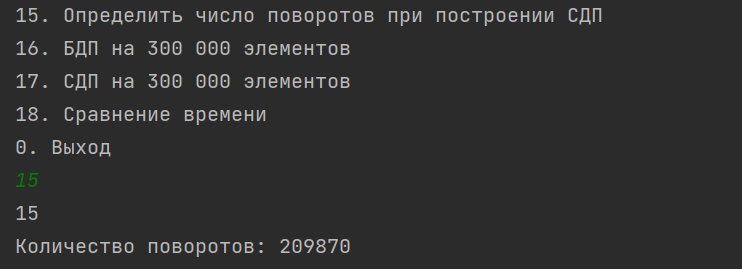


Рисунок 20. Количество поворотов дерева из 300 000 элементов

Пункт 16 отвечает за автоматизированное создание БДП на 300 000 элементов и поиск некоторого ключа в нем. Из рисунка 21 видно, что операция выполняется корректно:

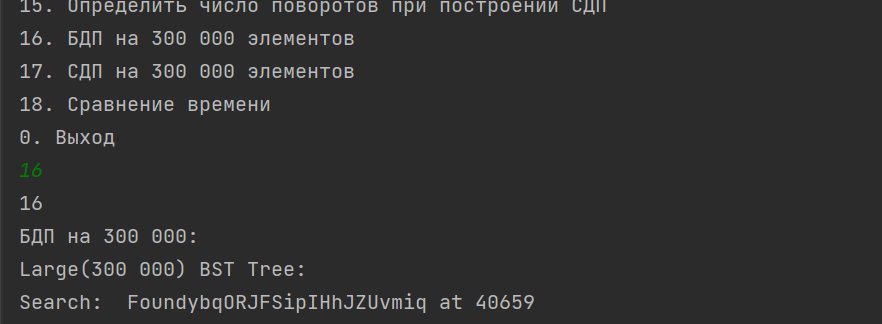


Рисунок 21. БДП на 300 000 элементов

Пункт 16 отвечает за автоматизированное создание СДП на 300 000 элементов и поиск некоторого ключа в нем. Из рисунка 22 видно, что операция выполняется корректно:

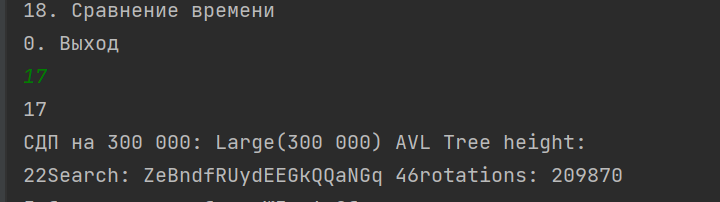


Рисунок 22. СДП на 300 000 элементов

Пункт 18 выполняет поиск записи в каждом случае: линейный поиск в файле, бинарное дерево поиска и AVL дерево в начале, середине и конце базы данных. Рассмотрим результаты:

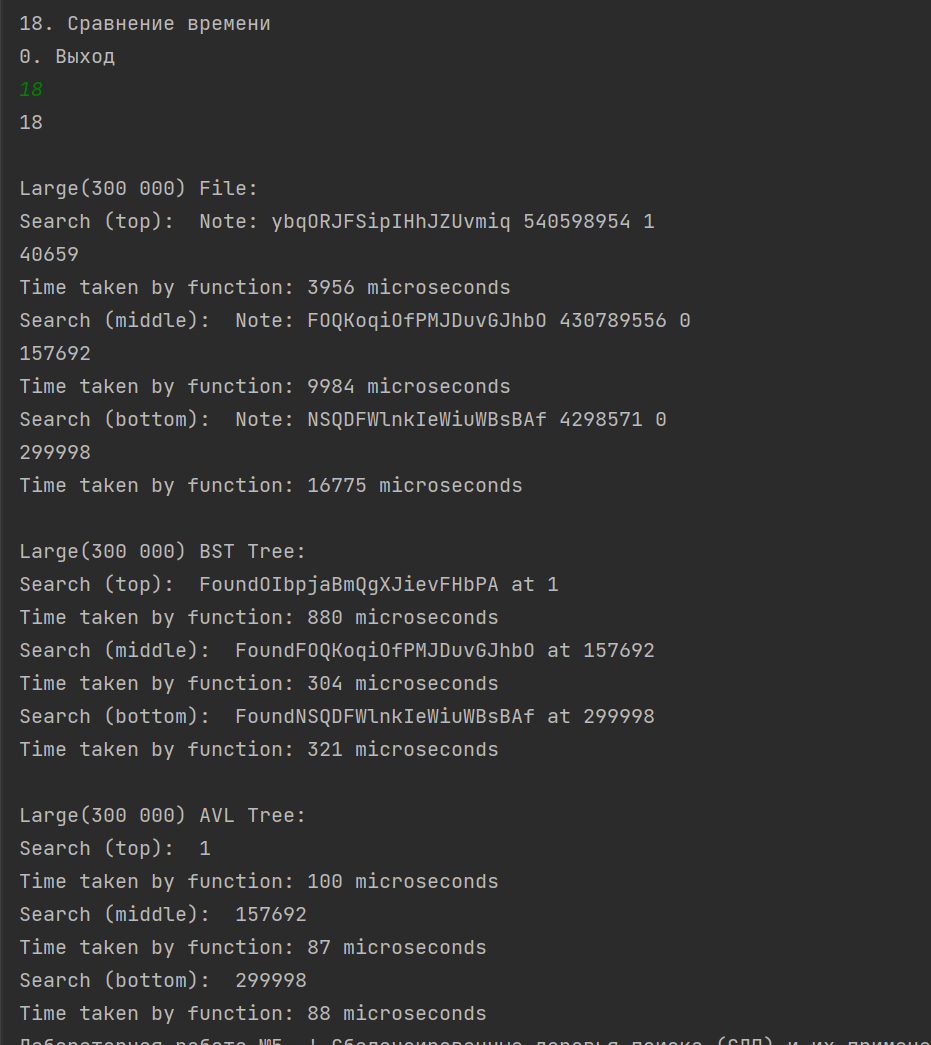


Рисунок 23. Сравнение времени поиска всех методов

Запишем их в таблицу для более удобного сравнения:

Таблица 2. Результаты замеров времени поиска тремя способами.

|  |  |  |  |
| --- | --- | --- | --- |
| Расположение элемента | Линейный поиск, мкс | Поиск в бинарном дереве, мкс | Поиск в сбалансированном AVL-дереве, мкс |
| Начало файла | 3956 | 880 | 100 |
| Середина файла | 9984 | 304 | 87 |
| Конец файла | 16775 | 321 | 88 |

Среднее число поворотов было определено ранее, и равняется 0.6996 поворота на вставку. Замеры времени проводились с учетом вывода в консоль, так что итоговое время больше истинного, но на сравнение трех методов это никак не влияет. Из таблицы 2 очевидно, что линейный поиск оказался самым медленным, чем дальше от начала файла находится нужная запись, тем дольше выполняется поиск. Поиск в бинарном дереве все еще медленнее из-за проблемы неоптимизированной высоты дерева, которая в AVL дереве технически верно решена, как следствие – самое быстрое время поиска среди всех представленных методов.

# **Выводы**

В результате выполнения работы были:

* получены навыки в разработки и реализации алгоритмов управления бинарным деревом поиска и сбалансированными бинарными деревьями поиска (АВЛ – деревьями);
* получены навыки в применении файловых потоков прямого доступа к данным файла;
* получены навыки в применении сбалансированного дерева поиска для прямого доступа к записям файла.

# **Исходный код программы**

main.cpp

|  |
| --- |
|  |
| //  // Created by lexa2k on 29.10.2022.  //  #include <iostream>  #include <string>  #include <chrono>  #include "AVLTree.h"  #include "Note.h"  #include "binfile.h"  #include "binTree.h"  using namespace std;  void avlLarge(){  AVLTree treeLarge;  binfile binLarge;  binLarge.textToBin("C:\\Users\\lexa2k\\Desktop\\RTU\_SEM2\\Siaod2sem\\Task5\\T5Large.txt", "T5large.bin");  treeLarge.BuildByFile("T5large.bin");  cout<<"Large(300 000) AVL Tree height: "<<endl;  cout<<treeLarge.root->height;  cout<<"Search: ZeBndfRUydEEGkQQaNGq "<<treeLarge.search("ZeBndfRUydEEGkQQaNGq");  cout<<"rotations: "<<treeLarge.rotations<<endl;  }  void BSTLarge(){  binTree treeLarge;  binfile binLarge;  binLarge.textToBin("C:\\Users\\lexa2k\\Desktop\\RTU\_SEM2\\Siaod2sem\\Task5\\T5Large.txt", "T5large.bin");  treeLarge.BuildByFile("T5Large.bin");  cout<<"\nLarge(300 000) BST Tree: "<<endl;  cout<<"Search: ";  treeLarge.searchByKey(treeLarge.root, "ybqORJFSipIHhJZUvmiq");  // treeLarge.printTree();  }  void analyzeLinearTime(){  binfile binLarge;  binLarge.textToBin("C:\\Users\\lexa2k\\Desktop\\RTU\_SEM2\\Siaod2sem\\Task5\\T5Large.txt", "T5large.bin");  cout<<"\nLarge(300 000) File: "<<endl;  cout<<"Search (top): ";  auto start = chrono::high\_resolution\_clock::now();  cout<<binLarge.searchBin("T5large.bin", "ybqORJFSipIHhJZUvmiq")<<endl;  auto end = chrono::high\_resolution\_clock::now();  auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);  cout << "Time taken by function: "<< duration.count() << " microseconds" << endl;  cout<<"Search (middle): ";  start = chrono::high\_resolution\_clock::now();  cout<<binLarge.searchBin("T5large.bin", "FOQKoqiOfPMJDuvGJhbO")<<endl;  end = chrono::high\_resolution\_clock::now();  duration = chrono::duration\_cast<chrono::microseconds>(end - start);  cout << "Time taken by function: "<< duration.count() << " microseconds" << endl;  cout<<"Search (bottom): ";  start = chrono::high\_resolution\_clock::now();  cout<<binLarge.searchBin("T5large.bin", "NSQDFWlnkIeWiuWBsBAf")<<endl;  end = chrono::high\_resolution\_clock::now();  duration = chrono::duration\_cast<chrono::microseconds>(end - start);  cout << "Time taken by function: "<< duration.count() << " microseconds" << endl;  return;  }  void avalyzeBSTTime(){  binTree treeLarge;  binfile binLarge;  binLarge.textToBin("C:\\Users\\lexa2k\\Desktop\\RTU\_SEM2\\Siaod2sem\\Task5\\T5Large.txt", "T5large.bin");  treeLarge.BuildByFile("T5Large.bin");  cout<<"\nLarge(300 000) BST Tree: "<<endl;  cout<<"Search (top): ";  auto start = chrono::high\_resolution\_clock::now();  treeLarge.searchByKey(treeLarge.root, "OIbpjaBmQgXJievFHbPA");  auto end = chrono::high\_resolution\_clock::now();  auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);  cout << "Time taken by function: "<< duration.count() << " microseconds" << endl;  cout<<"Search (middle): ";  start = chrono::high\_resolution\_clock::now();  treeLarge.searchByKey(treeLarge.root, "FOQKoqiOfPMJDuvGJhbO");  end = chrono::high\_resolution\_clock::now();  duration = chrono::duration\_cast<chrono::microseconds>(end - start);  cout << "Time taken by function: "<< duration.count() << " microseconds" << endl;  cout<<"Search (bottom): ";  start = chrono::high\_resolution\_clock::now();  treeLarge.searchByKey(treeLarge.root, "NSQDFWlnkIeWiuWBsBAf");  end = chrono::high\_resolution\_clock::now();  duration = chrono::duration\_cast<chrono::microseconds>(end - start);  cout << "Time taken by function: "<< duration.count() << " microseconds" << endl;  return;  }  void avalyzeAVLTime(){  AVLTree treeLarge;  binfile binLarge;  binLarge.textToBin("C:\\Users\\lexa2k\\Desktop\\RTU\_SEM2\\Siaod2sem\\Task5\\T5Large.txt", "T5large.bin");  treeLarge.BuildByFile("T5Large.bin");  cout<<"\nLarge(300 000) AVL Tree: "<<endl;  cout<<"Search (top): ";  auto start = chrono::high\_resolution\_clock::now();  cout<<treeLarge.search( "OIbpjaBmQgXJievFHbPA")<<endl;  auto end = chrono::high\_resolution\_clock::now();  auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);  cout << "Time taken by function: "<< duration.count() << " microseconds" << endl;  cout<<"Search (middle): ";  start = chrono::high\_resolution\_clock::now();  cout<<treeLarge.search("FOQKoqiOfPMJDuvGJhbO")<<endl;  end = chrono::high\_resolution\_clock::now();  duration = chrono::duration\_cast<chrono::microseconds>(end - start);  cout << "Time taken by function: "<< duration.count() << " microseconds" << endl;  cout<<"Search (bottom): ";  start = chrono::high\_resolution\_clock::now();  cout<<treeLarge.search("NSQDFWlnkIeWiuWBsBAf")<<endl;  end = chrono::high\_resolution\_clock::now();  duration = chrono::duration\_cast<chrono::microseconds>(end - start);  cout << "Time taken by function: "<< duration.count() << " microseconds" << endl;  return;  }  // Я знаю — моё дерево не проживёт и недели  // Я знаю — моё дерево в этом городе обречено  // Но я всё своё время провожу рядом с ним  // Мне все другие дела надоели  // Мне кажется, что это мой дом  // Мне кажется, что это мой друг  //  // Я посадил дерево  // Я посадил дерево  // Я посадил дерево  // Я, посадил дерево  // Я знаю — моё дерево, завтра, может, сломает школьник  // Я знаю — моё дерево скоро оставит меня  // Но, пока оно есть, я всегда рядом с ним  // Мне с ним радостно, мне с ним больно  // Мне кажется, что это мой мир  // Мне кажется, что это мой сын  int main(){  system("chcp 65001");  int choice = -1;  binfile bin;  AVLTree avlTree;  binTree binTree;  while(choice!=0){  cout<<"Лабораторная работа №5. ' Сбалансированные деревья поиска (СДП) и их применение для поиска данных в файле.'. Смольников Алексей. Вариант 22"<<endl;  cout<<"Выберите действие:"<<endl;  cout<<"---БИНАРНЫЙ ФАЙЛ---"<<endl;  cout<<"1. Создать двочиный файл"<<endl;  cout<<"2. Найти запись по ключу(линейный поиск)"<<endl;  cout<<"3. Вывод записи по известному адресу (строка начиная с 0)"<<endl;  cout<<"4. Добавить запись в файл"<<endl;  cout<<"---БИНАРНОЕ ДЕРЕВО ПОИСКА---"<<endl;  cout<<"5. Построить дерево по бинарному файлу"<<endl;  cout<<"6. Добавить элемент в дерево"<<endl;  cout<<"7. Поиск по ключу в дереве"<<endl;  cout<<"8. Удалить элемент из дерева"<<endl;  cout<<"9. Вывод дерева"<<endl;  cout<<"---СБАЛАНСИРОВАННОЕ ДЕРЕВО ПОИСКА---"<<endl;  cout<<"10. Построить дерево по бинарному файлу"<<endl;  cout<<"11. Добавить элемент в дерево"<<endl;  cout<<"12. Поиск по ключу в дереве"<<endl;  cout<<"13. Удалить элемент из дерева"<<endl;  cout<<"14. Вывод дерева"<<endl;  cout<<"15. Определить число поворотов при построении СДП"<<endl;  cout<<"16. БДП на 300 000 элементов"<<endl;  cout<<"17. СДП на 300 000 элементов"<<endl;  cout<<"18. Сравнение времени"<<endl;  cout<<"0. Выход"<<endl;  cin>>choice;  switch (choice) {  case 1: {  cout << "Введите путь к файлу: ";  string path;  cin >> path;  cout << "Введите путь к бинарному файлу: ";  string binPath;  cin >> binPath;  bin.textToBin(path, binPath);  break;  }  case 2:{  cout<<"Введите путь к бинарному файлу: ";  string binPath;  cin>>binPath;  cout<<"Введите страну: ";  string key;  cin>>key;  bin.searchBin(key, binPath);  break;  }  case 3:{  cout<<"Введите путь к бинарному файлу: ";  string binPath;  cin>>binPath;  cout<<"Введите номер строки: ";  int line;  cin>>line;  bin.directAccessBin(binPath, line);  break;  }  case 4:{  cout<<"Введите путь к бинарному файлу: ";  string binPath;  cin>>binPath;  cout<<"Введите страну: ";  string country;  cin>>country;  cout<<"Введите население: ";  int population;  cin>>population;  cout<<"Введите Факт нахождения в UNO: ";  int uno;  cin>>uno;  Note note;  strcpy(note.name, country.c\_str());  note.people = population;  note.Uno = uno;  bin.addBin(binPath, note);  break;  }  case 5:{  cout<<"Введите путь к бинарному файлу: ";  string binPath;  cin>>binPath;  binTree.BuildByFile(binPath);  break;  }  case 6:{  cout<<"Введите страну: ";  string country;  cin>>country;  cout<<"Введите Позицию в файле: ";  int pos;  cin>>pos;  char \*cstr = new char[country.length() + 1];  strcpy(cstr, country.c\_str());  binTree.addNode(binTree.root, cstr , pos);  delete [] cstr;  break;  }  case 7:{  cout<<"Введите страну: ";  string country;  cin>>country;  cout<<"Введите путь к бинарному файлу: ";  string binPath;  cin>>binPath;  char \*cstr = new char[country.length() + 1];  strcpy(cstr, country.c\_str());  int res = binTree.searchByKey(binTree.root, cstr);  cout<<endl;  Note note = bin.directAccessBin(binPath, res);  cout<<"Страна: "<<note.name<<endl;  cout<<"Население: "<<note.people<<endl;  cout<<"Факт нахождения в UNO: "<<note.Uno<<endl;  delete [] cstr;  break;  }  case 8:{  cout<<"Введите страну: ";  string country;  cin>>country;  char \*cstr = new char[country.length() + 1];  strcpy(cstr, country.c\_str());  binTree.deleteByKey(binTree.root, cstr);  delete [] cstr;  break;  }  case 9:{  binTree.printTree(binTree.root);  break;  }  case 10:{  cout<<"Введите путь к бинарному файлу: ";  string binPath;  cin>>binPath;  avlTree.BuildByFile(binPath);  break;  }  case 11:{  cout<<"Введите страну: ";  string country;  cin>>country;  cout<<"Введите Позицию в файле: ";  int pos;  cin>>pos;  char \*cstr = new char[country.length() + 1];  strcpy(cstr, country.c\_str());  avlTree.root = avlTree.insert(avlTree.root, cstr, pos);  delete [] cstr;  break;  }  case 12:{  cout<<"Введите страну: ";  string country;  cin>>country;  char \*cstr = new char[country.length() + 1];  strcpy(cstr, country.c\_str());  cout<<"FOUND at "<<avlTree.search(cstr)<<endl;  delete [] cstr;  break;  }  case 13:{  cout<<"Введите страну: ";  string country;  cin>>country;  char \*cstr = new char[country.length() + 1];  strcpy(cstr, country.c\_str());  avlTree.root = avlTree.deleteNode(avlTree.root, cstr);  delete [] cstr;  break;  }  case 14:{  avlTree.printTree(avlTree.root);  break;  }  case 15:{  //Повороты  cout<<"Количество поворотов: ";  cout<<avlTree.rotations<<endl;  break;  }  case 16:{  cout<<"БДП на 300 000: ";  BSTLarge();  break;  }  case 17:{  cout<<"CДП на 300 000: ";  avlLarge();  break;  }  case 18:{  analyzeLinearTime();  avalyzeBSTTime();  avalyzeAVLTime();  break;  }  }  }  return 0;  } |

Note.h

|  |
| --- |
| //  // Created by lexa2k on 29.10.2022.  //  #ifndef SIAOD2SEM\_NOTE\_H  #define SIAOD2SEM\_NOTE\_H  #include <iostream>  struct Note {  char name[45];  int people;  bool Uno;  };  const int sizeNote = sizeof(Note);  #endif //SIAOD2SEM\_NOTE\_H |

binfile.h

|  |
| --- |
| //  // Created by lexa2k on 29.10.2022.  //  #ifndef SIAOD2SEM\_BINFILE\_H  #define SIAOD2SEM\_BINFILE\_H  #include "Note.h"  struct binfile{  static void textToBin( std::string text, std::string bin);  static int searchBin( std::string bin, std::string name);  static Note directAccessBin( std::string bin, int index);  static void addBin( std::string bin, Note note);  };  #endif //SIAOD2SEM\_BINFILE\_H |

binfile.cpp

|  |
| --- |
| //  // Created by lexa2k on 29.10.2022.  //  #include <fstream>  #include <iostream>  #include <cstring>  #include <sstream>  #include "Note.h"  #include "binfile.h"  void printNoteNative(Note note){  std::cout << "Note: " << note.name << " " << note.people << " " << note.Uno << std::endl;  }  void binfile::textToBin(std::string text, std::string bin){  std::ifstream fin(text, std::ios::in);  std::ofstream fout(bin, std::ios::binary | std::ios::out);  std::string line;  while(std::getline(fin, line)){  Note note;  std::stringstream ss(std::ios::in | std::ios::out);  ss << line;  ss >> note.name >> note.people >> note.Uno;  fout.write((char\*)&note, sizeNote);  }  fin.close();  fout.close();  }  int binfile::searchBin( std::string bin, std::string name){  std::ifstream fin(bin, std::ios::binary | std::ios::in);  Note note;  int i=0;  while(!fin.eof()){  fin.read((char\*)&note, sizeNote);  if(strcmp(note.name, name.c\_str())!=0){  printNoteNative(note);  return i;  }  i++;  }  fin.close();  return -1;  }  Note binfile::directAccessBin( std::string bin, int index){  std::ifstream fin(bin, std::ios::binary | std::ios::in);  Note note;  fin.seekg(index\*sizeNote);  fin.read((char\*)&note, sizeNote);  printNoteNative(note);  fin.close();  return note;  }  void binfile::addBin(std::string bin, Note note){  std::ofstream fout(bin, std::ios::binary | std::ios::app);  fout.write((char\*)&note, sizeNote);  fout.close();  } |

bintree.h

|  |
| --- |
| //  // Created by lexa2k on 29.10.2022.  //  #ifndef SIAOD2SEM\_BINTREE\_H  #define SIAOD2SEM\_BINTREE\_H  #include <string>  class binTree{  public:  struct Node{  char key[45];  int index;  Node\* left;  Node\* right;  int height;  };  Node\* root=nullptr;  int size;  int height(Node\* p);  int fixheight(binTree::Node \*p);  void BuildByFile( std::string bin);  void addNode(Node\* &node, char\* key, int index);  void deleteByKey(Node\* &node, char\* key);  int searchByKey(Node\* node, char\* key);  void directAccess(Node\* node, int index);  void print(Node\* node, std::string offset);  void printTree(Node \*element= nullptr, std::string prefix = "", bool root = true);  };  #endif //SIAOD2SEM\_BINTREE\_H |

bintree.cpp

|  |
| --- |
| //  // Created by lexa2k on 29.10.2022.  //  #include "binTree.h"  #include "Note.h"  #include <fstream>  #include <iostream>  #include <cstring>  void binTree::BuildByFile(std::string bin){  std::ifstream file(bin, std::ios::binary | std::ios::in);  if(!file.is\_open()){  std::cout << "File not found" << std::endl;  return;  }  Note note;  int i=0;  while(file.read((char\*)&note, sizeNote)){  addNode(root, note.name, i);  i++;  }  file.close();  }  int binTree::height(binTree::Node \*p) {  return p ? p->height : 0;  }  int binTree::fixheight(binTree::Node \*p) {  int hl = height(p->left);  int hr = height(p->right);  return (hl > hr ? hl : hr) + 1;  }  void binTree::addNode(Node\* &node, char\* key, int index){  if(node==nullptr){  node = new Node;  strcpy(node->key, key);  node->index = index;  node->left = nullptr;  node->right = nullptr;  node->height = fixheight(node);  size++;  }else if(strcmp(key, node->key)<0){  addNode(node->left, key, index);  }else if(strcmp(key, node->key)>0){  addNode(node->right, key, index);  }  }  void binTree::deleteByKey(Node\* &node, char\* key){  if(node==nullptr){  return;  }else if(strcmp(key, node->key)<0){  deleteByKey(node->left, key);  }else if(strcmp(key, node->key)>0){  deleteByKey(node->right, key);  }else{  if(node->left==nullptr && node->right==nullptr){  delete node;  node = nullptr;  }else if(node->left==nullptr){  Node\* temp = node;  node = node->right;  delete temp;  }else if(node->right==nullptr){  Node\* temp = node;  node = node->left;  delete temp;  }else{  Node\* temp = node->right;  while(temp->left!=nullptr){  temp = temp->left;  }  strcpy(node->key, temp->key);  node->index = temp->index;  deleteByKey(node->right, temp->key);  }  }  }  int binTree::searchByKey(Node\* node, char\* key){  if(node==nullptr){  std::cout << "Not found" << std::endl;  return -1;  }else if(strcmp(key, node->key)<0){  searchByKey(node->left, key);  }else if(strcmp(key, node->key)>0){  searchByKey(node->right, key);  }else{  std::cout << "Found" << node->key << " at "<< node->index << std::endl;  return node->index;  }  }  void binTree::print(Node\* node, std::string offset){  if(node==nullptr){  return;  }  printTree(node->right, offset+" ");  std::cout << offset << node->key<<" Pos: "<<node->index<< std::endl;  printTree(node->left, offset+" ");  }  void binTree::printTree(Node \*element , std::string prefix, bool root) {  if (root) {  element = this->root;  }  if(!element) return;  if(element->right){  printTree(element->right, prefix + (root ? " " : "│ "), false);  }  std::cout << prefix << (root ? "└── " : "├── ") << element->key<<" "<<element->index<<std::endl;  if(element->left){  printTree(element->left, prefix + (root ? " " : "│ "), false);  }  } |

bintree.h

|  |
| --- |
| //  // Created by lexa2k on 29.10.2022.  //  #ifndef SIAOD2SEM\_BINTREE\_H  #define SIAOD2SEM\_BINTREE\_H  #include <string>  class binTree{  public:  struct Node{  char key[45];  int index;  Node\* left;  Node\* right;  int height;  };  Node\* root=nullptr;  int size;  int height(Node\* p);  int fixheight(binTree::Node \*p);  void BuildByFile( std::string bin);  void addNode(Node\* &node, char\* key, int index);  void deleteByKey(Node\* &node, char\* key);  int searchByKey(Node\* node, char\* key);  void directAccess(Node\* node, int index);  void print(Node\* node, std::string offset);  void printTree(Node \*element= nullptr, std::string prefix = "", bool root = true);  };  #endif //SIAOD2SEM\_BINTREE\_H |

AVLTree.h

|  |
| --- |
| //  // Created by lexa2k on 29.10.2022.  //  #ifndef SIAOD2SEM\_AVLTREE\_H  #define SIAOD2SEM\_AVLTREE\_H  #include <iostream>  #include <fstream>  #include <cstring>  #include "Note.h"  class AVLTree{  public:  struct Node{  char key[45];  int index;  Node\* left;  Node\* right;  int height;  };  Node\* root=nullptr;  int size;  int rotations=0;  void BuildByFile(std::string bin);  int height(Node\* p);  int bfactor(Node\* p);  void fixheight(Node\* p);  Node\* rotateright(Node\* p);  Node\* rotateleft(Node\* q);  Node\* balance(Node\* p);  Node\* insert(Node\* p, char\* k, int index);  Node\* findMin(Node\* p);  Node\* removeMin(Node\* p);  Node\* deleteNode(Node\* p, char\* k);  int search(char\* k);  void print(Node\* p, std::string prefix);  void printTree(Node \*element= nullptr, std::string prefix = "", bool root = true);  };  #endif //SIAOD2SEM\_AVLTREE\_H |

AVLTree.cpp

|  |
| --- |
| //  // Created by lexa2k on 29.10.2022.  //  #include "AVLTree.h"  void AVLTree::BuildByFile(std::string bin) {  std::ifstream fin(bin, std::ios::binary | std::ios::in);  Note note;  int i=0;  while(!fin.eof()){  fin.read((char\*)&note, sizeNote);  root = insert(root, note.name, i);  i++;  }  fin.close();  }  int AVLTree::height(Node \*p) {  return p ? p->height : 0;  }  int AVLTree::bfactor(Node \*p) {  return height(p->right) - height(p->left);  }  void AVLTree::fixheight(AVLTree::Node \*p) {  int hl = height(p->left);  int hr = height(p->right);  p->height = (hl > hr ? hl : hr) + 1;  }  AVLTree::Node\* AVLTree::insert(AVLTree::Node \*p, char \*k, int index) {  if(!p){  p = new Node;  strcpy(p->key, k);  p->index = index;  strcpy(p->key, k);  p->left = p->right = nullptr;  p->height = 1;  size++;  return p;  }  if(strcmp(k, p->key) < 0){  p->left = insert(p->left, k, index);  } else{  p->right = insert(p->right, k, index);  }  return balance(p);  }  AVLTree::Node\* AVLTree::balance(AVLTree::Node\* p){  fixheight(p);  //выполение одного из больших поворотов  if(bfactor(p) == 2){  if(bfactor(p->right) < 0)  p->right = rotateright(p->right);  return rotateleft(p);  }  if(bfactor(p) == -2){  if(bfactor(p->left) > 0)  p->left = rotateleft(p->left);  return rotateright(p);  }  return p;  }  AVLTree::Node\* AVLTree::rotateright(AVLTree::Node \*p) {  Node\* q = p->left;  p->left = q->right;  q->right = p;  fixheight(p);  fixheight(q);  rotations++;  return q;  }  AVLTree::Node\* AVLTree::rotateleft(AVLTree::Node \*q) {  Node\* p = q->right;  q->right = p->left;  p->left = q;  fixheight(q);  fixheight(p);  rotations++;  return p;  }  void AVLTree::print(AVLTree::Node \*p, std::string prefix) {  if(p){  print(p->right, prefix + " ");  std::cout << prefix << p->key << std::endl;  print(p->left, prefix + " ");  }  }  void AVLTree::printTree(Node \*element , std::string prefix, bool root) {  if (root) {  element = this->root;  }  if(!element) return;  if(element->right){  printTree(element->right, prefix + (root ? " " : "│ "), false);  }  std::cout << prefix << (root ? "└── " : "├── ") << element->key<</\* " ("<< height(element) << ")"<<\*/std::endl;  if(element->left){  printTree(element->left, prefix + (root ? " " : "│ "), false);  }  }  AVLTree::Node\* AVLTree::findMin(Node\* p){  return p->left ? findMin(p->left) : p;  }  AVLTree::Node\* AVLTree::removeMin(Node\* p){  if(p->left == 0)  return p->right;  p->left = removeMin(p->left);  return balance(p);  }  AVLTree::Node\* AVLTree::deleteNode(AVLTree::Node \*p, char \*k) {  if(!p) return nullptr;  if(strcmp(k, p->key) < 0){  p->left = deleteNode(p->left, k);  } else if(strcmp(k, p->key) > 0){  p->right = deleteNode(p->right, k);  } else{  Node\* q = p->left;  Node\* r = p->right;  delete p;  if(!r) return q;  Node\* min = findMin(r);  min->right = removeMin(r);  min->left = q;  return balance(min);  }  return balance(p);  }  int AVLTree::search(char \*k) {  Node\* p = root;  while(p){  if(strcmp(k, p->key) < 0){  p = p->left;  } else if(strcmp(k, p->key) > 0){  p = p->right;  } else{  return p->index;  }  }  return -1;  } |