|  |  |  |
| --- | --- | --- |
|  |  |  |
| 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»

по теме «сбалансированные деревья поиска (СДП) и их применение для поиска данных в файле»

|  |  |
| --- | --- |
| **Выполнил:**  Студент группыИКБО-13-22 | Козлов К.И. |
| **Проверил:** | ассистент Муравьёва Е.А. |

МОСКВА 2023 г.

**Практическая работа № 5**

**Цель работы**

* получить навыки в разработке 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.

**Задание 1**

Разработать приложение, которое использует бинарное дерево поиска (БДП)

для поиска записи с ключом в файле, структура которого представлена в

задании 2 вашего варианта.

1. Разработать класс (или библиотеку функций) «Бинарное дерево

поиска». Тип информационной части узла дерева: ключ и ссылка на запись в

файле. Методы, которые должны быть реализованы:

* включение элемента в дерево;
* поиск ключа в дереве;
* удаление ключа из дерева;
* отображение дерева.

2. Разработать класс (библиотеку функций) управления файлом (если

не создали в практическом задании 2). Включить методы:

* создание двоичного файла записей фиксированной длины из заранее

подготовленных данных в текстовом файле;

* поиск записи в файле с использованием БДП;
* остальные методы по вашему усмотрению.

3. Разработать и протестировать приложение.

4. Подготовить отчет

**Решение:**

Размер одной записи – **8 байт**

Полный листинг задания №1

|  |
| --- |
| #include <iostream>  #include <fstream>  #include <string>  #include <chrono>  //----Struct of wordRecord----//  struct wordRecord  {  char word[4];  int amountOfEntry;  wordRecord() {};  wordRecord(std::string word, int outAmountOfEntry)  {  for(int i = 0; i < 4; i++)  {  this->word[i] = word[i];  }  this->amountOfEntry = outAmountOfEntry;  }  };  //----Struct of binTree----//  struct binNode  {  unsigned int key;  std::streampos offset;  binNode\* leftNode;  binNode\* rightNode;  binNode(unsigned int outKey, int currOffset)  {  offset = currOffset;  key = outKey;  leftNode = nullptr;  rightNode = nullptr;  }  ~binNode() {};  };  //---Struct of binTree-----//  struct binTree  {  binNode\* root;  binTree() { root = nullptr; }  void insertNode(std::streampos currOffset, unsigned int outKey, binNode\*& start)  {  if (start == nullptr) { start = new binNode(outKey, currOffset); }  else if (start->key > outKey) { insertNode(currOffset, outKey, start->leftNode); }  else { insertNode(currOffset, outKey, start->rightNode); }  return;  }  binNode\*& searchNode(unsigned int searchKey, binNode\*& start)  {  if (start->key == searchKey) { return(start); }  else if (start->key > searchKey) { searchNode(searchKey, start->leftNode); }  else { searchNode(searchKey, start->rightNode); }  }  binNode\* preDeleteNode(unsigned int deleteKey, binNode\*& start)  {  if (start->leftNode == nullptr || start->rightNode == nullptr) return(nullptr);  if (start->leftNode->key == deleteKey) return(start);  if (start->rightNode->key == deleteKey) return(start);  else if (start->key > deleteKey) { preDeleteNode(deleteKey, start->leftNode); }  else { preDeleteNode(deleteKey, start->rightNode); }  }  void removeNode(unsigned int deleteKey)  {  binNode\* preDelete = preDeleteNode(deleteKey, root);  binNode\* targetDelete = nullptr;  if (preDelete == nullptr)  {  return;  }  //if target element is rightChild  if (preDelete->rightNode->key == deleteKey)  {  targetDelete = preDelete->rightNode;  //target element has only right child  if (targetDelete->leftNode == nullptr)  {  preDelete->rightNode = targetDelete->rightNode;  delete targetDelete;  return;  }  //target element has only left child  else if (targetDelete->rightNode == nullptr)  {  preDelete->rightNode = targetDelete->leftNode;  delete targetDelete;  return;  }  //target has both right and left childs  else  {  preDelete->rightNode = targetDelete->rightNode;  binNode\* iterRightLeft = targetDelete->rightNode;  while (iterRightLeft->leftNode != nullptr)  {  iterRightLeft = iterRightLeft->leftNode;  }  iterRightLeft->leftNode = targetDelete->leftNode;  delete targetDelete;  return;  }  }  else  {  targetDelete = preDelete->leftNode;  //target element has only right child  if (targetDelete->leftNode == nullptr)  {  preDelete->leftNode = targetDelete->rightNode;  delete targetDelete;  return;  }  //target element has only left child  else if (targetDelete->rightNode == nullptr)  {  preDelete->leftNode = targetDelete->leftNode;  delete targetDelete;  return;  }  //target has both right and left childs  else  {  preDelete->leftNode = targetDelete->rightNode;  binNode\* iterRightLeft = targetDelete->rightNode;  while (iterRightLeft->leftNode != nullptr)  {  iterRightLeft = iterRightLeft->leftNode;  }  iterRightLeft->leftNode = targetDelete->leftNode;  delete targetDelete;  return;  }  }  }  void showTree(binNode\*& start, unsigned int level)  {  if (start != nullptr)  {  showTree(start->rightNode, level + 1);  for (int i = 0; i < level; i++)  std::cout << " ";  std::cout << "-->" << start->key << '\_' << start->offset << '\n';  showTree(start->leftNode, level + 1);  }  }  ~binTree() {};  };  int main()  {  unsigned short userChoise;  binTree tree;  do  {  std::cin >> userChoise;  switch (userChoise)  {  case 0:  {  unsigned int amount;  unsigned int pos;  std::ofstream fileWriteBin("test.bin", std::ios::binary);  std::cin >> amount;  for (int i = 0; i < amount; i++)  {  wordRecord newWord("word", rand() % 300 + 1);  tree.insertNode(fileWriteBin.tellp(), newWord.amountOfEntry, tree.root);  fileWriteBin.write((char\*)(&newWord), sizeof(wordRecord));  }  fileWriteBin.close();  break;  }  case 1:  {  tree.showTree(tree.root, 0);  break;  }  case 2:  {  unsigned int deleteKey;  std::cin >> deleteKey;  tree.removeNode(deleteKey);  break;  }  case 3:  {  wordRecord check;  std::ifstream fileReadBin("test.bin", std::ios::binary);  unsigned int searchKey;  std::cin >> searchKey;  auto start = std::chrono::high\_resolution\_clock::now();  int pos = tree.searchNode(searchKey, tree.root)->offset;  auto end = std::chrono::high\_resolution\_clock::now();  auto elapsedMs = std::chrono::duration\_cast<std::chrono::milliseconds>(end - start);  fileReadBin.seekg(pos);  fileReadBin.read((char\*)(&check), sizeof(wordRecord));  std::cout << check.amountOfEntry;  }  }  } |

**Тестирование программы**

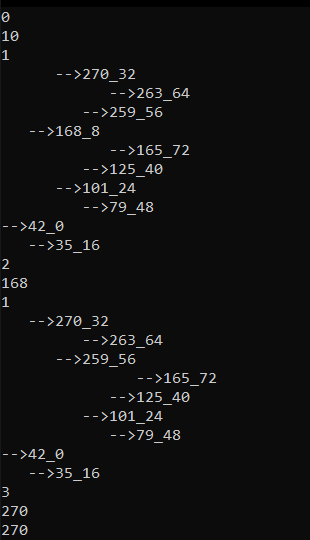


Рисунок 1 – Тестирование программы для 10 записей

**Задание 2**

Разработать приложение, которое использует сбалансированное дерево

поиска (СДП), предложенное в варианте, для доступа к записям файла.

1. Разработать класс СДП с учетом дерева варианта. Структура

информационной части узла дерева включает ключ и ссылку на запись в

файле (адрес места размещения). Основные методы:

* включение элемента в дерево;
* удаление ключа из дерева;
* поиск ключа в дереве с возвратом записи из файла;
* вывод дерева в форме дерева (с отображением структуры дерева).

2. Разработать приложение, которое создает и управляет СДП в

соответствии с заданием.

3. Выполнить тестирование.

4. Определить среднее число выполненных поворотов (число поворотов

на общее число вставленных ключей) при включении ключей в дерево при

формировании дерева из двоичного файла.

5. Оформить отчет

**Решение:**

**Вариант 17**

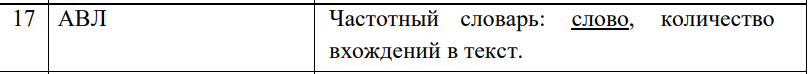


Рисунок 2 – Условие варианта

По реализации, AVL-дерево очень похоже на БДП, однако есть ряд отличий. Самое главное отличие: сбалансированность. AVL-дерево это сбалансированное бинарное дерево поиска, что значит, что разница глубин поддеревьев любого дерева не больше чем 1 по модулю. Из этого условия выкает надобность специальных функций – поворотов. Это функции балансируют дерево. Также нужны вспомогательные функции, которые будут искать фактор для сбалансированности, установка высоты корректной для каждого узла. Все эти функции добавлены в реализации AVL-дерева

Листинг 2 – полный код задания №2

|  |
| --- |
| #include <iostream>  #include <fstream>  #include <chrono>  static double inserts = 0;  static double rotates = 0;  static double coef = 0;  struct wordRecord  {  char word[4];  int amountOfEntry;  wordRecord() {};  wordRecord(std::string word, int outAmountOfEntry)  {  for (int i = 0; i < 4; i++)  {  this->word[i] = word[i];  }  this->amountOfEntry = outAmountOfEntry;  }  };  struct AVLNode  {  unsigned int key;  unsigned char height;  std::streampos offset;  AVLNode\* leftNode;  AVLNode\* rightNode;  AVLNode(unsigned int userKey, int currOffset)  {  offset = currOffset;  key = userKey;  leftNode = nullptr;  rightNode = nullptr;  height = 1;  }  };  unsigned char getHeight(AVLNode\* treeNode)  {  if (treeNode == nullptr) return(0);  return(treeNode->height);  }  int bfactor(AVLNode\* treeNode)  {  return(getHeight(treeNode->rightNode) - getHeight(treeNode->leftNode));  }  void fixHeight(AVLNode\* treeNode)  {  unsigned char leftHeight = getHeight(treeNode->leftNode);  unsigned char rightHeight = getHeight(treeNode->rightNode);  if (leftHeight > rightHeight) treeNode->height = leftHeight + 1;  else treeNode->height = rightHeight + 1;  }  void showTree(AVLNode\* start, unsigned int level)  {  if (start != nullptr)  {  showTree(start->rightNode, level + 1);  for (int i = 0; i < level; i++)  std::cout << " ";  std::cout << "-->" << start->key << ' ' << start->offset << '\n';  showTree(start->leftNode, level + 1);  }  }  AVLNode\* searchNode(int searchKey, AVLNode\* start)  {  if(start->key == searchKey) { return(start); }  else if (start->key > searchKey) { searchNode(searchKey, start->leftNode); }  else { searchNode(searchKey, start->rightNode); }  }  AVLNode\* rotateRight(AVLNode\* treeNode)  {  rotates++;  AVLNode\* leftChild = treeNode->leftNode;  treeNode->leftNode = leftChild->rightNode;  leftChild->rightNode = treeNode;  fixHeight(treeNode);  fixHeight(leftChild);  return (leftChild);  }  AVLNode\* rotateLeft(AVLNode\* treeNode)  {  rotates++;  AVLNode\* rightChild = treeNode->rightNode;  treeNode->rightNode = rightChild->leftNode;  rightChild->leftNode = treeNode;  fixHeight(treeNode);  fixHeight(rightChild);  return (rightChild);  }  AVLNode\* balanceFunc(AVLNode\* treeNode)  {  fixHeight(treeNode);  if (bfactor(treeNode) == 2)  {  if (bfactor(treeNode->rightNode) < 0)  treeNode->rightNode = rotateRight(treeNode->rightNode);  return rotateLeft(treeNode);  }  if (bfactor(treeNode) == -2)  {  if (bfactor(treeNode->leftNode) > 0)  treeNode->leftNode = rotateLeft(treeNode->leftNode);  return rotateRight(treeNode);  }  return (treeNode);  }  AVLNode\* insertNode(AVLNode\* treeNode, int key, int currOffset)  {  if (!treeNode) return (new AVLNode(key, currOffset));  if (key < treeNode->key) treeNode->leftNode = insertNode(treeNode->leftNode, key, currOffset);  else treeNode->rightNode = insertNode(treeNode->rightNode, key, currOffset);  return balanceFunc(treeNode);  }  AVLNode\* findMin(AVLNode\* treeNode)  {  if (treeNode->leftNode == nullptr) return(treeNode);  else return(findMin(treeNode->leftNode));  }  AVLNode\* removeMin(AVLNode\* treeNode)  {  if (treeNode->leftNode == 0) return (treeNode->rightNode);  treeNode->leftNode = removeMin(treeNode->leftNode);  return (balanceFunc(treeNode));  }  AVLNode\* removeNode(AVLNode\* treeNode, int deleteKey)  {  if (!treeNode) return (0);  if (deleteKey < treeNode->key) treeNode->leftNode = removeNode(treeNode->leftNode, deleteKey);  else if (deleteKey > treeNode->key) treeNode->rightNode = removeNode(treeNode->rightNode, deleteKey);  else  {  AVLNode\* l = treeNode->leftNode;  AVLNode\* r = treeNode->rightNode;  delete treeNode;  if (!r) return (l);  AVLNode\* min = findMin(r);  min->rightNode = removeMin(r);  min->leftNode = l;  return (balanceFunc(min));  }  return (balanceFunc(treeNode));  }  int main()  {  AVLNode\* root = new AVLNode(0, 0);  unsigned int amount;  unsigned int pos;  unsigned int userChoice;  std::ofstream fileWriteBin("testTwo.bin", std::ios::binary);  std::cin >> amount;  for (int i = 0; i < amount; i++)  {  wordRecord newWord("word", rand() % 300 + 1);  if (i == 0)  {  root = new AVLNode(newWord.amountOfEntry, fileWriteBin.tellp());  inserts++;  }  else  {  root = insertNode(root, newWord.amountOfEntry, fileWriteBin.tellp());  inserts++;  }  fileWriteBin.write((char\*)(&newWord), sizeof(wordRecord));  }  fileWriteBin.close();  showTree(root, 0);  do  {  std::cin >> userChoice;  switch (userChoice)  {  case 1:  {  int deleteKey;  std::cin >> deleteKey;  root = removeNode(root, deleteKey);  showTree(root, 0);  break;  }  case 2:  {  wordRecord check;  int pos;  int key;  std::cin >> key;  auto start = std::chrono::high\_resolution\_clock::now();  pos = searchNode(key, root)->offset;  auto end = std::chrono::high\_resolution\_clock::now();  auto elapsedMs = std::chrono::duration\_cast<std::chrono::milliseconds>(end - start);  std::ifstream fileReadBin("testTwo.bin", std::ios::binary);  fileReadBin.seekg(pos);  fileReadBin.read((char\*)(&check), sizeof(wordRecord));  std::cout << check.amountOfEntry << ']\n';  break;  }  case 3:  {  double coeff = rotates / inserts;  std::cout << coeff << '\n';  break;  }  }  } while (userChoice == 1 || userChoice == 2 || userChoice == 3);  } |

**Тестирование программы**

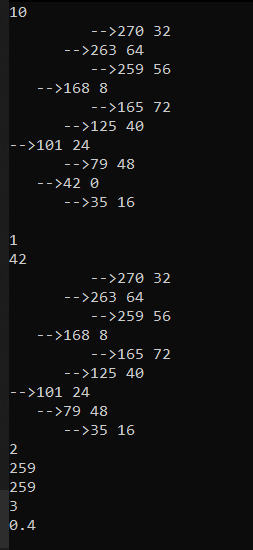


Рисунок 3 – Тестирование программы задания 2

**Задание 3**

Выполнить анализ алгоритма поиска записи с заданным ключом при

применении структур данных:

* хеш – таблица;
* бинарное дерево поиска;
* СДП

Требования по выполнению задания

1. Построить хеш-таблицу из чисел файла.

2. Осуществить поиск введенного целого числа в двоичном дереве

поиска, в сбалансированном дереве и в хеш-таблице.

3. Протестировать на данных:

а) небольшого объема (100, 1000 записей);

б) большого объема (1 000 000 записей).

4. Оформить таблицу результатов

5. Провести анализ алгоритма поиска ключа на исследованных

поисковых структурах на основе данных, представленных в таблице.

6. Оформить отчет

**Решение:**

Таблица №1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| Вид  поисковой  структуры | Количество  элементов,  загруженных  в структуру в  момент  выполнения  поиска | Емкостная  сложность: объем  памяти для  структуры в байтах | Количество  выполненных  сравнений,  время на поиск  ключа в  структуре |
| Бинарное древо | 100 | 800 | 5 | 0.0014901 |
| 1000 | 8000 | 8 | 0.0013877 |
| 1000000 | 8000000 | 3 | 0.0010273 |
| Косое дерево | 100 | 800 | 5 | 0.001346 |
| 1000 | 8000 | 8 | 0.001352 |
| 1000000 | 8000000 | 1 | 0.0011362 |
| Хеш табл. | 100 | 2200 | 3 | 0.0009 |
| 1000 | 22000 | 4 | 0.0013 |
| 1000000 | 22000000 | 1 | 0.0014 |

**Вывод**

При выполнении этих задач были освоены бинарные деревья поиска как структуры хранения данных и работа с ними, а также применение их для записи данных. Подводя итог, деревья являются важной структурой данных и имеют широкое применение в компьютерной науке и информационных технологиях. Главным преимуществом бинарных деревьев поиска является удобство поиска и вставки:

* Бинарные деревья поиска позволяют эффективно выполнять операции поиска и вставки. Благодаря упорядоченной структуре данных, поиск в дереве может быть выполнен за логарифмическое время, что делает его быстрым и эффективным.
* AVL-дерево ускоряет и оптимизирует процесс поиска, а следовательно и удаления элемента в дерево за счет своей сбалансированности.

Обобщая результаты, СДП имеет лучшее время поиска востребованных элементов, а в худшем случае получаем тоже время что и для обычного дерева, но в деревьях занимается больше памяти под каждый элемент, чем в хеш-таблице.