ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

# ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту по дисциплине

«Структуры и алгоритмы обработки данных» на тему

SCAPEGOAT TREE

Выполнил студент Бессонов Алексей Евгеньевич

Ф.И.О.

Группы ИВ-121

Работу принял ст. преп. Кафедры ВС Д. М. Берлизов

подпись

Защищена Оценка

Новосибирск – 2023

**Оглавление**

Оглавление

[ПОЯСНИТЕЛЬНАЯ ЗАПИСКА 1](#_Toc58783442)

[ВВЕДЕНИЕ 3](#_Toc58783443)

[1. Описание структуры 3](#_Toc58783444)

[**1.1 Характеристики и свойства** 3](#_Toc58783445)

[**1.2 Представление scapegoat tree** 4](#_Toc58783446)

[**1.3 Основные операции scapegoat tree** 5](#_Toc58783447)

[**1.3.1 Вставка нового элемента** 5](#_Toc58783448)

[**1.3.2 Удаление элемента** 6](#_Toc58783449)

[**1.3.3 Поиск** 7](#_Toc58783450)

[**1.3.4 Балансировка дерева** 8](#_Toc58783451)

[**1.4 Вывод по дереву** 1](#_Toc58783452)1

2.ОПИСАНИЕ И РЕАЛИЗАЦИЯ ФУНКУЦИЙ………………………………12

2.1 Основные функции……………………………………………………………………………….12

2.2 Дополнительные функции……………………………………………………………………….16

[3.Экспериментальное исследование эффективности структуры данных 21](#_Toc58783453)

[**3.1 Организация моделирования** 21](#_Toc58783454)

[**3.2 Результаты моделирования** 21](#_Toc58783455)

[ЗАКЛЮЧЕНИЕ 24](#_Toc58783456)

# 

# ВВЕДЕНИЕ

Для эффективной работы с данными не обойтись без структур данных, таких как: хеш-таблицы, деревья поиска, кучи и др. Каждая из них по-своему уникальна, и поэтому программисту важно выбрать наиболее подходящую структуру данных для конкретной задачи.

В данной работе рассматривается такой тип данных, как scapegoat tree или же дерево козла отпущения.

# 1. Описание структуры

## **1.1 Характеристики и свойства**

**Scapegoat Tree** — структура данных, представляющая собой частично сбалансированное дерево поиска (степень сбалансированности может быть настроена), такое что операции поиска, вставки и удаления работают за O(logn), при этом скорость одной операции может быть улучшена в ущерб другой.

При работе необходимо поддерживать состояние сбалансированного дерева, иначе время работы операции поиска может превысить O(logn).

#### **Степень сбалансированности**

Будем считать, что дерево является сбалансированным, если выполняются следующее: Введем коэффициент α, который показывает, насколько дерево может быть несбалансированным. Математически это выглядит следующим образом: 1/2⩽α⩽1; size(left[x])⩽α⋅size(x); size(right[x])⩽α⋅size(x),где size(left[x]) и size(right[x])— размер левого и правого поддерева вершины x.

Так же это дерево для балансировки, в отличии красно-черного, АВЛ или Декартового, не использует повороты.

Основные свойства scapegoat tree:

* Выбор коэффициента α позволяет ускорить некоторые операции. Например, выбор большого значения α позволит выполнять очень много операций вставки, но замедлит операции поиска. При этом выбор коэффициента можно выполнять в процессе выполнения, опираясь на входные данные. Однако, неправильный выбор α приводит к сильному увеличению времени работы.
* Не требуется проводить перебалансировку дерева при поиске, что гарантирует максимальное время работы поиска O(logn)
* В некоторых случаях операции модификации занимают O(n), хотя их амортизированная сложность - O(logn)
* За счет отсутствия необходимости хранить дополнительные данные в вершинах данное дерево оптимальнее остальных по памяти.
* Вес узла равен кол-ву узлов в своем поддереве.

## **1.2 Представление scapegoat tree**

Но для начала мы рассмотрим, как вообще выглядит scapegoat tree, для этого нам нужно разобраться какой коэффициент мы выбираем, мы помним, что 1/2⩽α⩽1.

Выберем α равное 0.7 просто для примера и показа самого дерева.

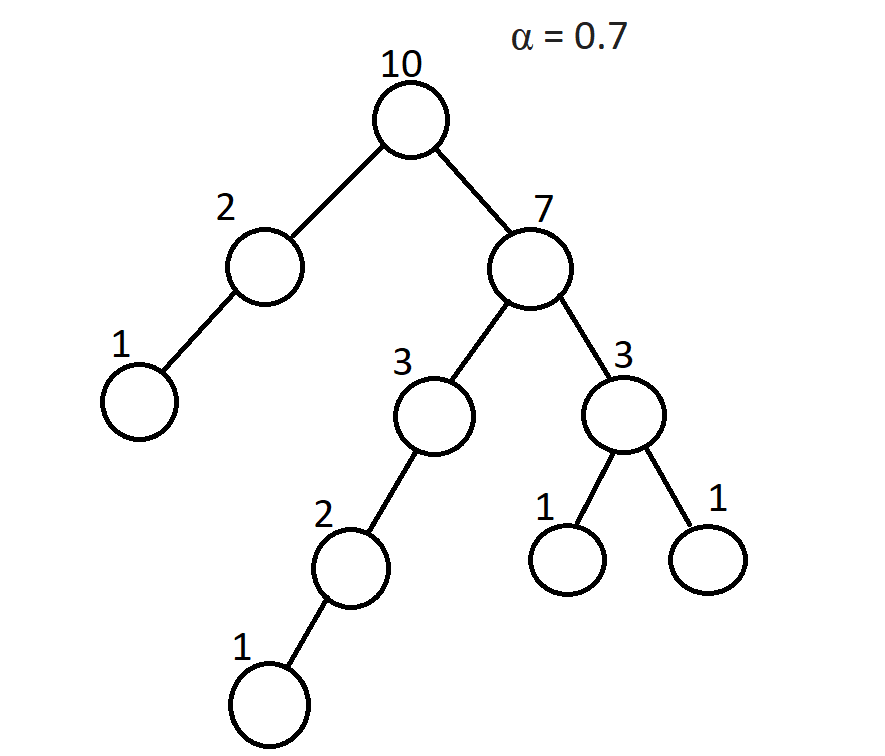


Рис. 1 – Наше дерево

Тут мы видим, что вес корневого узла у нас равен 10, и следующие дочерние элементы не должны превышать 10\*0.7 = 7, и так далее, каждый последующий дочерний элемент, не должен превышать вес своего отца \* α.

Высота дерева не превышает

## **1.3 Основные операции scapegoat tree**

Мы рассмотрим в этой работе все основные операции для работы со scapegoat tree:

* 1. Вставка нового элемента
  2. Удаление элемента
  3. Поиск элемента
  4. Балансировка дерева

### **1.3.1 Вставка нового элемента**

Пока дерево остается α-сбалансированным, выполняем модифицированную вставку элемента в дерево, которая аналогична обычной вставке в двоичное дерево. В тот момент, когда дерево стало несбалансированным, надо начать поиск вершины, которая нарушает условие сбалансированности. Для этого надо пройти по дереву вверх. Только что вставленная вершина ей быть не может. После нахождения этой вершины надо запустить операцию балансировки.

Наглядно покажем, как происходит вставка нового элемента, вернемся к нашему рисунку 1, где мы построили дерево, опираясь на α равное 0.7.

Теперь добавим новый элемент в наше дерево рисунок 2.

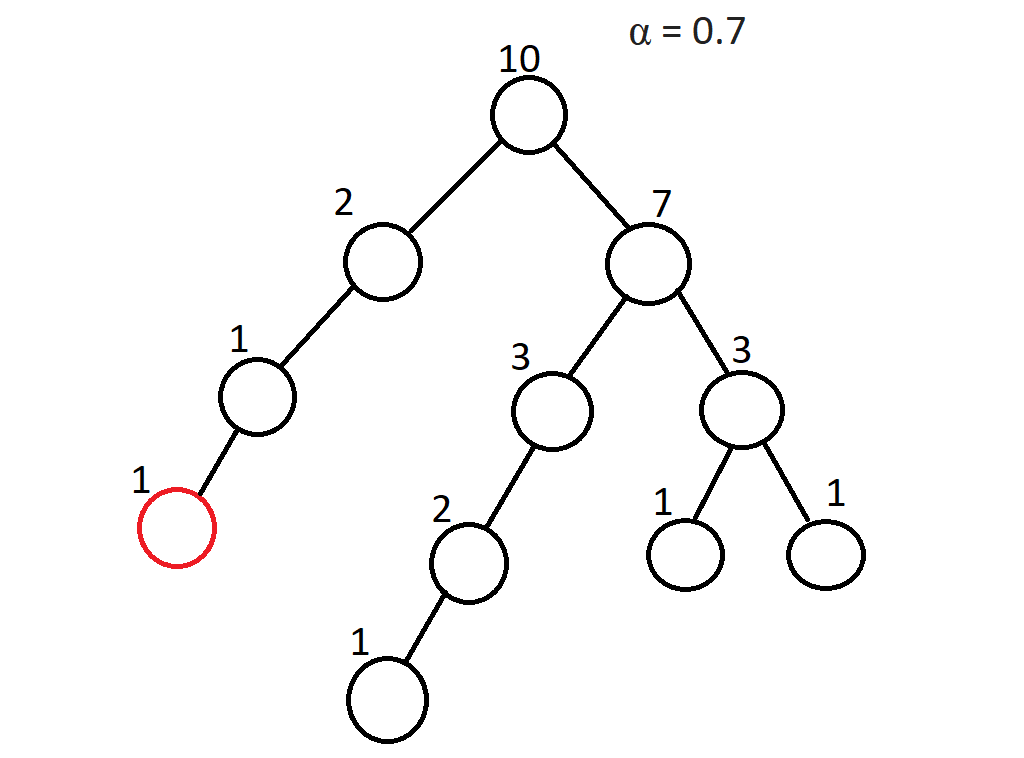


Рис. 2 – добавления узла в дереве

Но, мы сталкиваемся с проблемой, что наше дерево перестало отвечать своему свойству, что вес узла, равен кол-ву узлов его поддерева, поэтому прибавляем по единице к каждому родительскому узлу связанного с этой частью дерева рисунок 3.

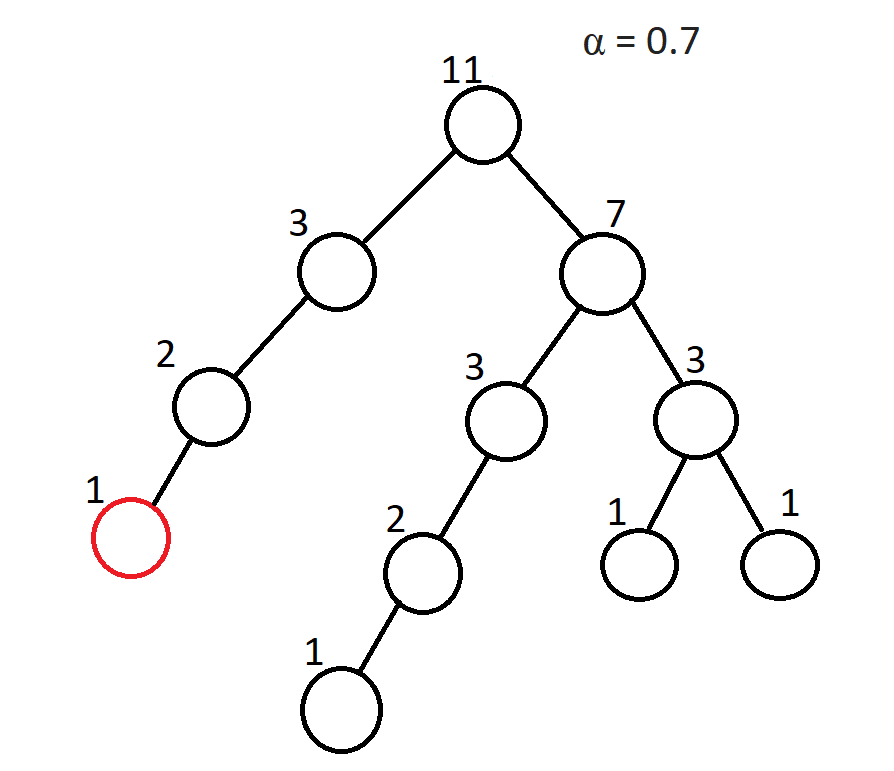


Рис. 3 – изменения веса узлов

Далее для каждого узла нужно проверить не нарушил ли он баланс дерева size(left[x])⩽α⋅size(x); size(right[x])⩽α⋅size(x),где size(left[x]) и size(right[x])— размер левого и правого поддерева вершины x.

Асимптотическая сложность данной операции в среднем случае O(log(N)), но в худшем случае составляет O(N), почему так? Чтобы добавить новый элемент, нам понадобится O(logn) времени, но так как нам еще придется прибегнуть к балансировке дерева, если элемент окажется козлом отпущения, а балансировка занимает O(N), то и в худшем случае мы получим O(N).

### **1.3.2 Удаление элемента**

Удаляем вершину обычным удалением вершины бинарного дерева поиска (поиск, удаление, возможное переподвешивание детей).  
Далее проверяем выполнение условия  
size(left[x])⩽α⋅size(x); size(right[x])⩽α⋅size(x),где size(left[x]) и size(right[x])— размер левого и правого поддерева вершины x.  
если оно выполняется — дерево не нужно ребалансировать

Так же покажем на примере уже знакомого нам дерева с рисунка 1 α равное 0.7

Давайте удалим один из элементов данного дерева рисунок 4.

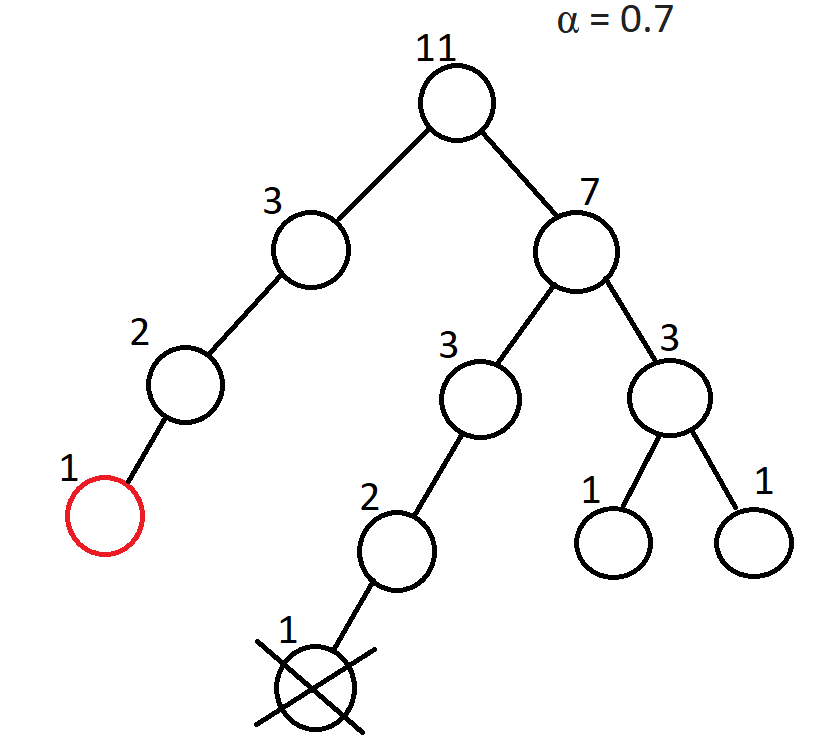


Рис. 4 – удаление узла

Мы снова столкнулись с нарушением свойства, что вес узла, не соответствует кол-ву элементов его поддерева, только теперь мы отнимем единицу у всех родительских узлов, связанного с этой частью дерева рисунок 5.

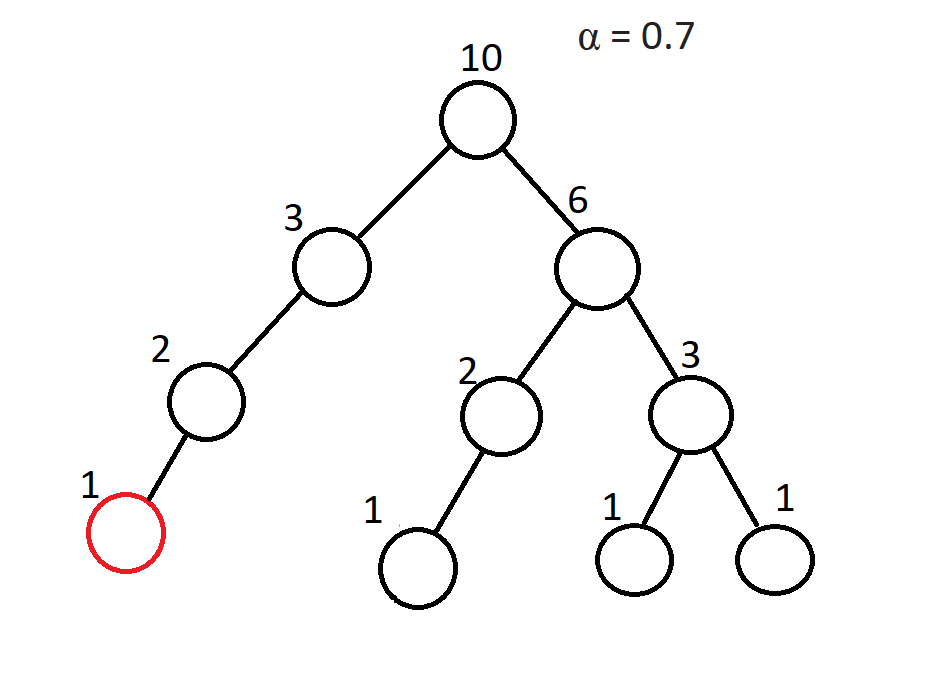


Рис. 5 – перерасчёт веса узлов

Асимптотическая сложность будет такая же, как у вставки нового элемента. В среднем случае O(log(N)), а в худшем O(N).

### **1.3.3 Поиск**

Итак, у нас есть некоторое Scapegoat-дерево и мы хотим найти в нём элемент. Поскольку это двоичное дерево поиска, то и поиск будет стандартным: идём от корня, сравниваем вершину с искомым значением, если нашли — возвращаем, если значение в вершине меньше — рекурсивно ищем в левом поддереве, если больше — в правом. Дерево по ходу поиска не модифицируется.  
  
 Сложность операции поиска, зависит от α и выражается формулой так же формулой, что H ⩽   
  
 Т.е. сложность, конечно, логарифмическая, вот только основание логарифма интересное. При α близком к 0.5 мы получаем двоичный (или почти двоичный) логарифм, что означает идеальную (или почти идеальную) скорость поиска. При α близком к единице основание логарифма стремится к единице, а значит общая сложность стремится к O(N).

### **1.3.4 Балансировка дерева**

Давайте добавим еще один элемент в наше дерево и сразу пересчитаем вес рисунок 6.

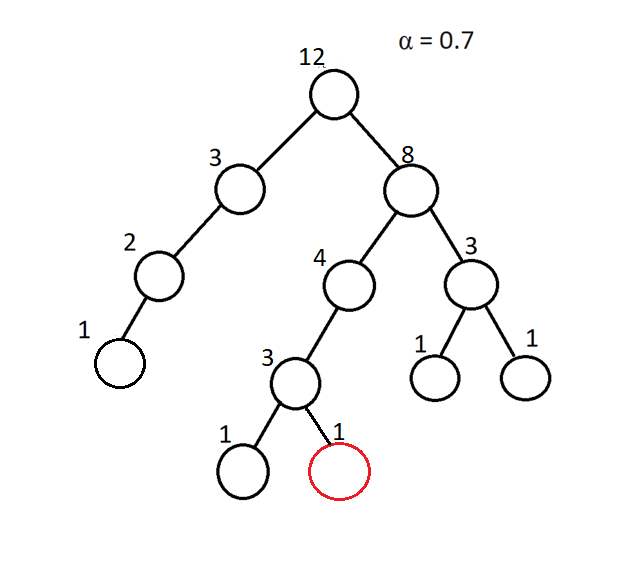


Рис. 6 – Добавления узла в дерево

И мы сталкиваемся с новой проблемой, у узла с весом 4, дочерний узел, превышает нашу формулу, где каждый дочерний узел, не должен превышать вес отца умноженный на константу.

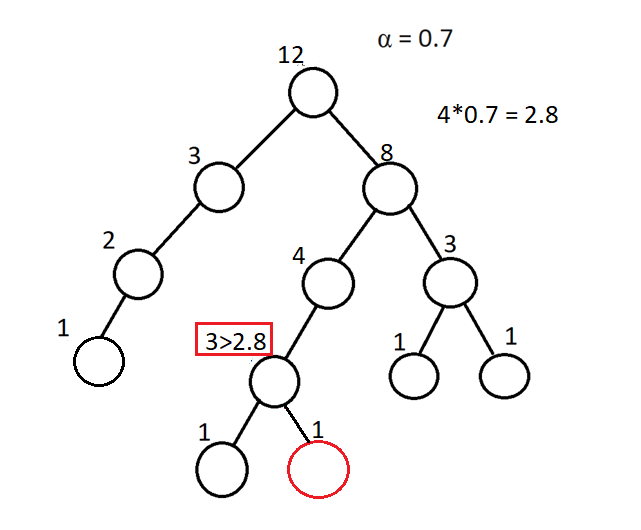


Рис. 7 – Нарушение условий баланса дерева

В этом случае нам нужно взять все поддерево и перестроить с нуля новое поддерево, которое будет соответствовать нашей формуле.

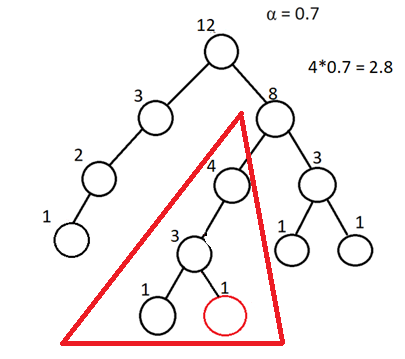


Рис. 8 – Поддерево в котором нарушилось условие баланса

Обойдем все наше поддерево с помощью обхода бинарного дерева и получаем на выходе отсортированный список с вершинами.

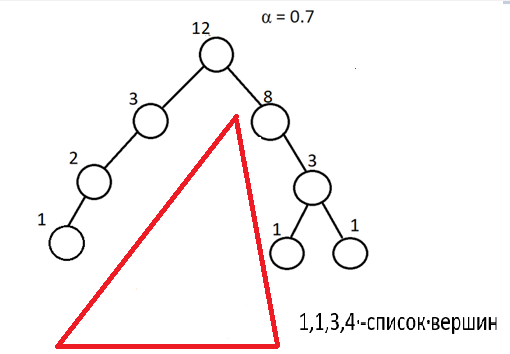


Рис. 9 – Поддерево и его список вершин

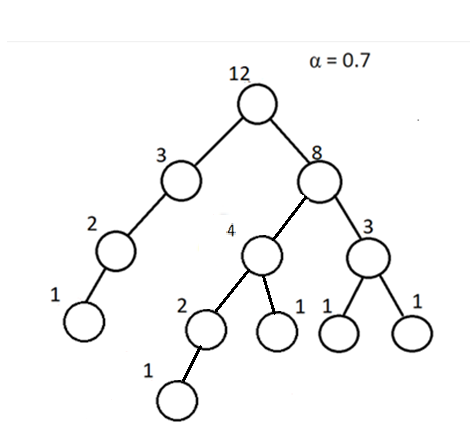


Рис. 10 – Перебалансированое дерева

Теперь наше дерево сбалансировано и снова отвечает условию формулы.

Асимптотическая сложность операции балансировки имеет O(n), так как приходится пройти узлы поддерева, чтобы найти их значения в отсортированном порядке.

## **1.4 Вывод по дереву**

Отсутствие дополнительных затрат по памяти и перебалансировки при поиске — это хорошо, это работает на нас. С другой стороны перебалансировки при модификациях не так чтобы уж очень дешевы. Ну и выбор α существенно влияет на производительность тех или иных операций. Сами изобретатели сравнивали производительность Scapegoat-дерева с красно-черными и Splay-деревьями. У них получилось подобрать α так, что на случайных данных Scapegoat-дерево превзошло по скорости все другие типы деревьев, на любых операциях (что вообще-то весьма неплохо). К сожалению, на монотонно-возрастающем наборе данных Scapegoat-дерево работает хуже в части операций вставки и выиграть у Scapegoat не вышло ни при каком α.

**2 Описание и реализация функций**

**2.1 Основные функции**

Node \*insert(Node \*\*node, int key) – функция добавления нового узла

Эта функция вставляет новый узел с заданным ключом в бинарное дерево поиска, представленное указателем на корневой узел. Если дерево пустое, то функция создает новый узел с заданным ключом и делает его корневым узлом дерева.

Если дерево не пустое, то функция ищет место для вставки нового узла, сравнивая ключ нового узла с ключом текущего узла. Если новый ключ меньше ключа текущего узла, то функция рекурсивно вызывает саму себя для левого поддерева текущего узла, передавая указатель на указатель на левый дочерний узел и заданный ключ. Если новый ключ больше ключа текущего узла, то функция рекурсивно вызывает саму себя для правого поддерева текущего узла, передавая указатель на указатель на правый дочерний узел и заданный ключ. Если новый ключ равен ключу текущего узла, функция возвращает нулевой указатель.

После вставки нового узла функция проверяет, не превышает ли отношение размера левого или правого поддерева к общему размеру дерева заданную константу ALPHA. Если это отношение превышает заданную константу, то функция вызывает функцию rebuild для перестройки дерева.

Функция возвращает указатель на корневой узел дерева.

# 

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30** | Node \*insert(Node \*\*node, int key) {      if (\*node == NULL) {          \*node = new\_node(key);          return \*node;      }      Node \*new\_node;      if (key < (\*node)->key) {          (\*node)->left = insert(&(\*node)->left, key);      } else if(key > (\*node)->key) {          (\*node)->right = insert(&(\*node)->right, key);      }else{          return NULL;      }      int left\_size = size((\*node)->left);      int right\_size = size((\*node)->right);      if (left\_size > ALPHA \* (left\_size + right\_size + 1) ||          right\_size > ALPHA \* (left\_size + right\_size + 1)) {          rebuild(node);      }      return \*node;  } |

Node \*deleteNode(Node \*node, int key) – функция удаления узла в дереве

Эта функция удаляет узел с заданным ключом из бинарного дерева поиска, представленного корневым узлом node. Если узел с заданным ключом не найден в дереве, функция возвращает нулевой указатель.

Если дерево не пустое, функция ищет узел с заданным ключом, сравнивая ключ заданного узла с ключом текущего узла. Если заданный ключ меньше ключа текущего узла, то функция рекурсивно вызывает саму себя для левого поддерева текущего узла, передавая указатель на левый дочерний узел и заданный ключ. Если заданный ключ больше ключа текущего узла, то функция рекурсивно вызывает саму себя для правого поддерева текущего узла, передавая указатель на правый дочерний узел и заданный ключ.

Если заданный ключ равен ключу текущего узла, то функция выполняет удаление узла из дерева, следуя одному из двух случаев:

Случай 1: удаляемый узел не имеет дочерних узлов или имеет только один дочерний узел. В этом случае функция сохраняет указатель на дочерний узел, если он есть, и освобождает память, выделенную под удаляемый узел, затем возвращает сохраненный указатель на дочерний узел.

Случай 2: удаляемый узел имеет два дочерних узла. В этом случае функция находит узел с наименьшим ключом в правом поддереве удаляемого узла, копирует его ключ в удаляемый узел и рекурсивно вызывает саму себя для удаления узла с наименьшим ключом из правого поддерева.

После удаления узла функция проверяет, не превышает ли отношение размера левого или правого поддерева к общему размеру дерева заданную константу ALPHA. Если это отношение превышает заданную константу, то функция вызывает функцию rebuild для перестройки дерева.

Функция возвращает указатель на корневой узел дерева после удаления узла с заданным ключом. Если узел с заданным ключом не найден в дереве, функция возвращает нулевой указатель.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42** | Node \*deleteNode(Node \*node, int key) {      if (node == NULL) {          return NULL;      }      if (key < node->key) {          node->left = deleteNode(node->left, key);      } else if (key > node->key) {          node->right = deleteNode(node->right, key);      } else {          // Случай 1: Нет ребенка или только один ребенок          if (node->left == NULL) {              struct Node \*temp = node->right;              free(node);              return temp;          } else if (node->right == NULL) {              struct Node \*temp = node->left;              free(node);              return temp;          }          // Случай 2: Двое детей          struct Node \*temp = minKeyNode(node->right);          node->key = temp->key;          node->right = deleteNode(node->right, temp->key);      }      int left\_size = size(node->left);      int right\_size = size(node->right);      // Проверка, не разбалансировано ли дерево и не нуждается ли оно в перестройке      if (left\_size > ALPHA \* (left\_size + right\_size + 1) ||          right\_size > ALPHA \* (left\_size + right\_size + 1)) {          printf("\n\n\nrebild\n\n\n");          return rebuild(&node);      }      return node;  } |

Node \*search(Node \*node, int key) – функция поиска элемента по ключу

Данная функция также выполняет поиск узла в дереве по заданному ключу.

Входные параметры:

node: указатель на корневой узел дерева, в котором выполняется поиск.

key: ключ, по которому ищется узел.

Алгоритм работы функции следующий:

Если указатель на текущий узел равен NULL, то возвращается NULL, так как дерево пусто или узел не найден.

Если заданный ключ меньше ключа текущего узла, то поиск продолжается в левом поддереве текущего узла.

Если заданный ключ больше ключа текущего узла, то поиск продолжается в правом поддереве текущего узла.

Если ключ текущего узла совпадает с заданным ключом, то возвращается указатель на текущий узел.

Итоговый результат выполнения функции - указатель на узел с заданным ключом, если такой узел существует, или NULL, если узел с заданным ключом не найден в дереве.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13** | Node \*search(Node \*node, int key) {      if (node == NULL) return NULL;      if (key < node->key) {          return search(node->left, key);      } else if (key > node->key) {          return search(node->right, key);      } else {          return node;      }  } |

Node \* rebuild(Node \*\*node) – функция ребалансировка дерева

Данная функция выполняет перестройку дерева, если оно становится разбалансированным (т.е. отношение размеров левого и правого поддеревьев превышает заданную константу ALPHA). Перестройка заключается в построении нового сбалансированного дерева на основе ключей узлов старого дерева.

Входной параметр:

node: указатель на указатель на корневой узел дерева, которое нужно перестроить.

Алгоритм работы функции следующий:

Вычисляется количество узлов в текущем дереве с помощью функции size().

Выделяется память под массив arr, в котором будут храниться ключи узлов текущего дерева.

С помощью функции addArr() ключи узлов текущего дерева добавляются в массив arr.

С помощью функции build\_bst() строится новое сбалансированное дерево на основе ключей из массива arr.

Указатель на корневой узел старого дерева заменяется указателем на корневой узел нового дерева.

Освобождается память, выделенная под массив arr.

Возвращается указатель на корневой узел нового дерева.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13** | Node \* rebuild(Node \*\*node) {      int n = size(\*node);      int \*arr = (int\*) malloc(n \* sizeof(int));      int i = 0;        addArr(\*node, arr, &i);      \*node = build\_bst(arr, 0, n - 1);      free(arr);      return \*node;  } |

**2.2 Дополнительные функции**

void addArr(Node \*node, int \*arr, int \*i) – создаёт упорядоченный по ключам массив

Данная функция добавляет все ключи из дерева node в массив arr в порядке возрастания ключей. Она рекурсивно обходит дерево в порядке левый поддерево - текущий узел - правое поддерево и добавляет ключ текущего узла в массив arr. Индекс массива i передается по указателю, чтобы его значение было сохранено между рекурсивными вызовами и каждый раз увеличивалось на 1 для корректной записи ключа в массив.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8** | void addArr(Node \*node, int \*arr, int \*i){          if (node != NULL) {              addArr(node->left, arr, i);              arr[(\*i)++] = node->key;              addArr(node->right, arr, i);          }  } |

Node \*new\_node(int key) – создаёт новый узел

Данная функция создает новый узел scapegoat дерева с заданным ключом и возвращает указатель на него. Функция выделяет память для структуры узла с помощью функции malloc() и инициализирует поля key, left и right созданного узла переданными значениями. Функция не изменяет значения полей родительских узлов и не связывает созданный узел с родительскими узлами.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9** | Node \*new\_node(int key) {      Node \*node = malloc(sizeof(Node));      node->key = key;      node->left = NULL;      node->right = NULL;      return node;  } |

int size(Node \*node) – функция для подсчёта веса узла

Данная функция вычисляет количество узлов в scapegoat дереве. Если дерево пустое, то возвращается 0, иначе возвращается сумма 1 (за узел, в котором функция вызвана) и размеров левого и правого поддеревьев, которые рекурсивно вычисляются с помощью вызова функции size для соответствующих дочерних узлов.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8** | int size(Node \*node) {      if (node == NULL) {          return 0;      } else {          return 1 + size(node->left) + size(node->right);      }  } |

Node \*build\_bst(int \*arr, int start, int end) – данная функция перестраивает дерева.

Функция build\_bst принимает отсортированный массив целых чисел arr и границы start и end, которые определяют текущий подмассив. Функция строит бинарное дерево поиска из подмассива, заданного границами start и end.

Алгоритм работы функции:

Если start больше end, то возвращается значение NULL, так как подмассив пустой.

Иначе, выбирается элемент в середине подмассива (индекс mid) и создаётся новый узел с этим элементом.

Затем рекурсивно вызывается build\_bst для левой половины подмассива (границы start и mid-1) и результат сохраняется в качестве левого поддерева узла.

Рекурсивно вызывается build\_bst для правой половины подмассива (границы mid+1 и end) и результат сохраняется в качестве правого поддерева узла.

Наконец, созданный узел возвращается в качестве корня построенного дерева.

Функция возвращает указатель на корень построенного дерева.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14** | Node \*build\_bst(int \*arr, int start, int end) {      if (start > end) {          return NULL;      }      int mid = (start + end) / 2;      Node \*node = new\_node(arr[mid]);      node->left = build\_bst(arr, start, mid - 1);      node->right = build\_bst(arr, mid + 1, end);      return node;  } |

void inorder(Node \*node) – данная функция выводит узлы дерева в порядке возрастания

Эта функция реализует обход дерева в порядке "in-order", то есть сначала посещает левое поддерево, затем текущий узел и потом правое поддерево. В данном случае функция выводит значения ключей узлов дерева в порядке возрастания.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8** | void inorder(Node \*node) {      if (node != NULL) {          inorder(node->left);          printf("%d ", node->key);          inorder(node->right);      }  } |

Tree \*new\_tree() – создаёт новое дерево

Данная функция создает новое дерево Tree и возвращает указатель на него. При создании, корневой узел устанавливается как NULL, а размер дерева устанавливается равным 0.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8** | Tree \*new\_tree() {      Tree \*tree = malloc(sizeof(Tree));      tree->root = NULL;      tree->size = 0;      return tree;  } |

void printTree(struct Node \*root, int space) – функция вывода в консоль дерева в виде дерева

Данная функция используется для вывода scapegoat дерева на экран в виде пирамиды. Она принимает указатель на корень дерева и количество пробелов, которые необходимо распечатать перед каждым узлом для отображения его уровня в дереве.

Функция сначала рекурсивно вызывает себя для правого поддерева, увеличивая количество пробелов на 5. Затем она распечатывает текущий узел, отображая его ключ и отступы для выравнивания. Наконец, она рекурсивно вызывает себя для левого поддерева, также увеличивая количество пробелов на 5.

В итоге, при вызове этой функции для корня дерева и количества пробелов, равного 0, на экран будет выведена пирамида, представляющая структуру бинарного дерева.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13** | void printTree(struct Node \*root, int space) {      if (root == NULL)          return;      space += 5;      printTree(root->right, space);      printf("\n");      for (int i = 5; i < space; i++)      printf(" ");      printf("%d\n", root->key);      printTree(root->left, space);  } |

struct Node\* minKeyNode(struct Node\* node) – функция для определения минимального ключа в поддереве.

Данная функция находит узел с минимальным ключом в поддереве с корнем в заданном узле. Она проходит по левым дочерним узлам текущего узла, пока не дойдет до листа, и возвращает этот лист как узел с минимальным ключом.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12** | struct Node\* minKeyNode(struct Node\* node) {      struct Node\* current = node;      /\* Проходимся влево от текущего узла, пока не дойдем до листа \*/      while (current && current->left != NULL)          current = current->left;      return current;  } |

# 3.Экспериментальное исследование эффективности структуры данных

## **3.1 Организация моделирования**

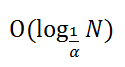
Прежде всего нас интересует функция поиска, а именно - скорость поиска узла по ключу в зависимости от количества ключей. Для этого проведем анализ в среднем и в худшем случае, а после - изобразим результаты графически. График будем строить при помощи Excel, замерять время внутри программы – при помощи функции clock() из библиотеки time: чтобы получить время выполнения алгоритма, будем сохранять время перед началом поиска, а затем вычитать из времени после завершения поиска предыдущее найденное значение времени.

## **3.2 Результаты моделирования**

Рис. 11 – график зависимости времени от количества элементов функции вставки

Посмотрев на график, можно сделать вывод, что время вставки элемента, зависит напрямую от нашего коэффициента, так как, чем больше коэффициент, тем меньше шансов, что каждый последующий элемент нарушит баланс и нам придется запускать балансировку дерева.

Рис. 12 – график зависимости времени от количества элементов функции поиска

Посмотрев на график, можно понять, что поиск тоже зависит, от выбранного нами коэффициента, но теперь, чем меньше наш коэффициент, тем быстрее поиск, мы уже говорили, что это из-за того, что поиск напрямую зависит от формулы  и чем, больше коэффициент, тем сильнее логарифм стремится к единице, а значит общая сложность будет стремиться к линейной, что мы и видим на графике.

Удаление

Рис. 13 – график зависимости времени от количества элементов функции удаления

Посмотрев на график, можно сделать вывод, что время вставки элемента, зависит напрямую от нашего коэффициента, так как, чем больше коэффициент, тем меньше шансов, что каждый последующий удалённый элемент нарушит баланс и нам придется запускать балансировку дерева.

# 

# ЗАКЛЮЧЕНИЕ

В результате выполнения работы разработана и исследована структура данных «scapegoat»: были описаны ее основные свойства, характеристики, а также подробно описаны алгоритмы работы с этой структурой данных во всех случаях.

Осуществлено моделирование разработанной структуры данных. На графиках наглядно показано время работы на кол-ве элементов в зависимости от выбранного коэффициента на всех основных операциях scapegoat tree.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Scapegoat-деревья/Блог компании Инфопульс*: // Хабр, 2014. URL: https://habr.com/ru/company/infopulse/blog/246759/ (дата обращения: 20.11.2020).
2. CMSC 420: Lecture 12 Extended and Scapegoat Trees//, 2019. – С. 8.
3. Dr. Anil Maheshwari*.* A STUDY AND AN IMPLEMENTATION OF SCAPEGOAT TREES, 2006. – С. 37
4. *Scapegoat tree* // Википедия: свободная энциклопедия. URL: https://neerc.ifmo.ru/wiki/index.php?title=Scapegoat\_Tree (дата обращения: 16.11.2020).

ПРИЛОЖЕНИЕ

Исходный код программы

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44**  **45**  **46**  **47**  **48**  **49**  **50**  **51**  **52**  **53**  **54**  **55**  **56**  **57**  **58**  **59**  **60**  **61**  **62**  **63**  **64**  **65**  **66**  **67**  **68**  **69**  **70**  **71**  **72**  **73**  **74**  **75**  **76**  **77**  **78**  **79**  **80**  **81**  **82**  **83**  **84**  **85**  **86**  **87**  **88**  **89**  **90**  **91**  **92**  **93**  **94**  **95**  **96**  **97**  **98**  **99**  **100**  **101**  **102**  **103**  **104**  **105**  **106**  **107**  **108**  **109**  **110**  **111**  **112**  **113**  **114**  **115**  **116**  **117**  **118**  **119**  **120**  **121**  **122**  **123**  **124**  **125**  **126**  **127**  **128**  **129**  **130**  **131**  **132**  **133**  **134**  **135**  **136**  **137**  **138**  **139**  **140**  **141**  **142**  **143**  **144**  **145**  **146**  **147**  **148**  **149**  **150**  **151**  **152**  **153**  **154**  **155**  **156**  **157**  **158**  **159**  **160**  **161**  **162**  **163**  **164**  **165**  **166**  **167**  **168**  **169**  **170**  **171**  **172**  **173**  **174**  **175**  **176**  **177**  **178**  **179**  **180**  **181**  **182**  **183**  **184**  **185**  **186**  **187**  **188**  **189**  **190**  **191**  **192**  **193**  **194**  **195**  **196**  **197**  **198**  **199**  **200**  **201**  **202**  **203**  **204**  **205**  **206**  **207**  **208**  **209**  **210**  **211**  **212**  **213**  **214**  **215**  **216**  **217**  **218**  **219**  **220**  **221**  **222**  **223**  **224**  **225**  **226**  **227**  **228**  **229**  **230**  **231**  **232**  **234**  **235**  **236**  **237**  **238**  **239**  **240**  **241**  **242**  **243**  **244**  **245**  **246**  **247**  **248**  **249**  **250**  **251**  **252**  **253**  **254**  **255**  **256**  **257**  **258**  **259**  **260**  **261**  **262**  **263**  **264**  **265**  **266**  **267**  **268**  **269**  **270**  **271**  **272**  **273**  **274**  **275**  **276**  **277**  **278**  **279**  **280**  **281**  **282**  **283**  **284**  **285**  **286**  **287**  **288**  **289**  **290**  **291**  **292**  **293**  **294**  **295**  **296**  **297**  **298**  **299**  **300**  **301**  **302**  **303**  **304**  **305**  **306** | #include <stdio.h>  #include <stdlib.h>  #define ALPHA 0.75 // balancing factor  // Node struct  typedef struct Node {      int key;      struct Node \*left;      struct Node \*right;  } Node;  // Tree struct  typedef struct {      Node \*root;      int size;  } Tree;  // Функция для создания нового узла  Node \*new\_node(int key) {      Node \*node = malloc(sizeof(Node));      node->key = key;      node->left = NULL;      node->right = NULL;      return node;  }  // Функция для вычисления размера поддерева  int size(Node \*node) {      if (node == NULL) {          return 0;      } else {          return 1 + size(node->left) + size(node->right);      }  }  // Функция для формирования сбалансированного дерева из отсортированного массива  Node \*build\_bst(int \*arr, int start, int end) {      if (start > end) {          return NULL;      }      int mid = (start + end) / 2;      Node \*node = new\_node(arr[mid]);      node->left = build\_bst(arr, start, mid - 1);      node->right = build\_bst(arr, mid + 1, end);      return node;  }  // Сохраняет ключи узлов дерева при обходе по порядку в массиве  void addArr(Node \*node, int \*arr, int \*i){          if (node != NULL) {              addArr(node->left, arr, i);              arr[(\*i)++] = node->key;              addArr(node->right, arr, i);          }  }  // Функция для перестройки дерева с корнем в данном узле  Node \* rebuild(Node \*\*node) {      int n = size(\*node);      int \*arr = (int\*) malloc(n \* sizeof(int));      int i = 0;        addArr(\*node, arr, &i);      \*node = build\_bst(arr, 0, n - 1);      free(arr);      return \*node;  }  // Функция для вставки ключа в дерево с корнем в данном узле  Node \*insert(Node \*\*node, int key) {      if (\*node == NULL) {          \*node = new\_node(key);          return \*node;      }      if (key < (\*node)->key) {          (\*node)->left = insert(&(\*node)->left, key);      } else if(key > (\*node)->key) {          (\*node)->right = insert(&(\*node)->right, key);      }else{          return NULL;      }      int left\_size = size((\*node)->left);      int right\_size = size((\*node)->right);      if (left\_size > ALPHA \* (left\_size + right\_size + 1) ||          right\_size > ALPHA \* (left\_size + right\_size + 1)) {          rebuild(node);      }      return \*node;  }  // Функция для поиска ключа в дереве с корнем в данном узле  Node \*search(Node \*node, int key) {      if (node == NULL) return NULL;      if (key < node->key) {          return search(node->left, key);      } else if (key > node->key) {          return search(node->right, key);      } else {          return node;      }  }  // Функция для печати ключей в дереве с корнями в данном узле  void inorder(Node \*node) {      if (node != NULL) {          inorder(node->left);          printf("%d ", node->key);          inorder(node->right);      }  }  // Функция для создания нового дерева  Tree \*new\_tree() {      Tree \*tree = malloc(sizeof(Tree));      tree->root = NULL;      tree->size = 0;      return tree;  }  void printTree(struct Node \*root, int space) {      if (root == NULL)          return;      space += 5;      printTree(root->right, space);      printf("\n");      for (int i = 5; i < space; i++)      printf(" ");      printf("%d\n", root->key);      printTree(root->left, space);  }    struct Node\* minKeyNode(struct Node\* node) {      struct Node\* current = node;      /\* Проходимся влево от текущего узла, пока не дойдем до листа \*/      while (current && current->left != NULL)          current = current->left;      return current;  }  Node \*deleteNode(Node \*node, int key) {      if (node == NULL) {          return NULL;      }      if (key < node->key) {          node->left = deleteNode(node->left, key);      } else if (key > node->key) {          node->right = deleteNode(node->right, key);      } else {          // Случай 1: Нет ребенка или только один ребенок          if (node->left == NULL) {              struct Node \*temp = node->right;              free(node);              return temp;          } else if (node->right == NULL) {              struct Node \*temp = node->left;              free(node);              return temp;          }          // Случай 2: Двое детей          struct Node \*temp = minKeyNode(node->right);          node->key = temp->key;          node->right = deleteNode(node->right, temp->key);      }      int left\_size = size(node->left);      int right\_size = size(node->right);      // Проверка, не разбалансировано ли дерево и не нуждается ли оно в перестройке      if (left\_size > ALPHA \* (left\_size + right\_size + 1) ||          right\_size > ALPHA \* (left\_size + right\_size + 1)) {          return rebuild(&node);      }      return node;  }  int main() {      Tree \*tree = new\_tree();      insert(&tree->root, 5);      printTree(tree->root, 5);      printf("\n==========\n");      insert(&tree->root, 4);      printTree(tree->root, 5);      printf("\n==========\n");      insert(&tree->root, 3);      printTree(tree->root, 5);      printf("\n==========\n");      insert(&tree->root, 2);      printTree(tree->root, 5);      printf("\n==========\n");      insert(&tree->root, 7);      printTree(tree->root, 5);      printf("\n==========\n");      insert(&tree->root, 6);      printTree(tree->root, 5);      printf("\n==========\n");      insert(&tree->root, 8);      printTree(tree->root, 5);      printf("\n==========\n");      insert(&tree->root, 1);      printTree(tree->root, 5);      printf("\n==========\n");      insert(&tree->root, 9);      printTree(tree->root, 5);      printf("\n==========\n");      inorder(tree->root);      tree->root = deleteNode(tree->root, 1);      printTree(tree->root, 5);      printf("\n==========\n");      tree->root = deleteNode(tree->root, 2);      printTree(tree->root, 5);      printf("\n==========\n");      tree->root = deleteNode(tree->root, 3);      printTree(tree->root, 5);      printf("\n==========\n");      tree->root = deleteNode(tree->root, 4);      printTree(tree->root, 5);      printf("\n==========\n");      tree->root = deleteNode(tree->root, 5);      printTree(tree->root, 5);      printf("\n==========\n");      tree->root = deleteNode(tree->root, 6);      printTree(tree->root, 10);      printf("\n==========\n");      tree->root = deleteNode(tree->root, 7);      inorder(tree->root);      printTree(tree->root, 5);      printf("\n");        return 0;  } |