Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

К У Р С О В А Я Р А Б О Т А

«АВЛ – дерево»

По дисциплине: Объектно-ориентированное программирование

Выполнил
студент гр. 3331506/70401 М.М. Куликов

Преподаватель М.С. Ананьевский

«___» _____ 2020 г.

Санкт-Петербург
2020

Оглавление

Введение

АВЛ-деревом называется такое дерево поиска, в котором для любого его узла высоты левого и правого поддеревьев отличаются не более, чем на 1. Эта структура данных разработана советскими учеными Адельсон-Вельским Георгием Максимовичем и Ландисом Евгением Михайловичем в 1962 году. Аббревиатура АВЛ соответствует первым буквам фамилий этих ученых. Первоначально АВЛ-деревья были придуманы для организации перебора в шахматных программах. Советская шахматная программа «Каисса» стала первым официальным чемпионом мира в 1974 году.

Данная работа содержит описание алгоритма, его программный код на языке C++, а также анализ алгоритма.

Описание алгоритма

В каждом узле АВЛ-дерева, помимо ключа, данных и указателей на левое и правое поддеревья (левого и правого сыновей), хранится показатель баланса – разность высот правого и левого поддеревьев. В некоторых реализациях этот показатель может вычисляться отдельно в процессе обработки дерева тогда, когда это необходимо.

На рисунке 1(а) приведен пример АВЛ-дерева. Таким образом, получается, что в АВЛ-дереве показатель баланса для каждого узла, включая корень, по модулю не превосходит 1. На рисунке 1(б) приведен пример дерева, которое не является АВЛ-деревом, поскольку в одном из узлов баланс нарушен.

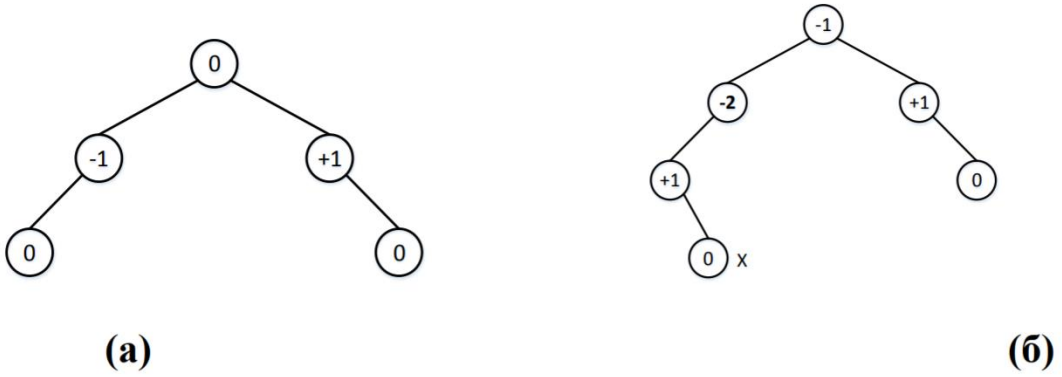


Рисунок 1 — (а) пример АВЛ-дерева; (б) пример дерева, не являющегося АВЛ-деревом: в узле X сбалансированность нарушена

Вставка узла в АВЛ-дерево

Рассмотрим, что происходит с АВЛ-деревом при добавлении нового узла. Сначала узел добавляется в дерево с помощью стандартного алгоритма вставки в двоичное дерево поиска. Показатели баланса в ряде узлов при этом изменятся, и сбалансированность может нарушиться. Сбалансированность считается нарушенной, если показатель баланса по модулю превысил 1 в одном или нескольких узлах. При добавлении нового узла разбалансировка может произойти сразу в нескольких узлах, но все они будут лежать на пути от этого добавленного узла к корню, как показано на рисунке 2. Тем не менее, перестраивать будем поддерево с корнем в том из этих узлов, который

является ближайшим к добавленному. В примере на рисунке 2 этот узел обведен кружком. Общее правило для всех добавляемых узлов, приводящих к разбалансировке: чтобы найти 19 корень поддерева, которое понадобится перестраивать, надо подниматься по дереву от вновь добавленного узла до тех пор, пока не найдется первый узел, в котором нарушена сбалансированность. Назовем его опорным узлом. Таким образом, искать этот узел надо, поднимаясь наверх, а не спускаясь от корня. После того как опорный узел будет найден, будет проведена процедура перестройки поддерева с корнем в этом узле с целью восстановления его сбалансированности. Остальная часть дерева останется в прежнем виде. При этом все дерево также станет сбалансированным — показатель баланса не будет превышать 1 по модулю во всех узлах дерева.

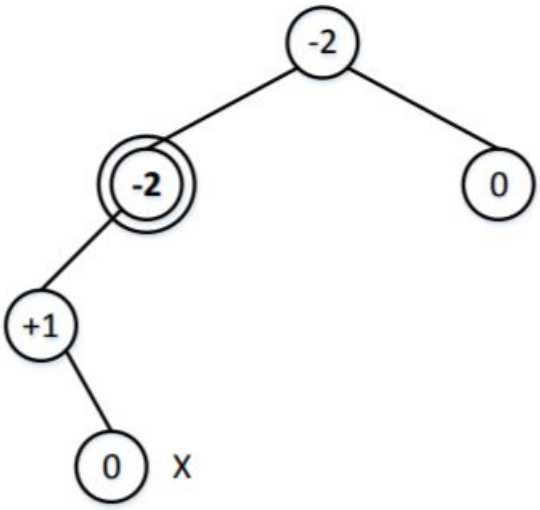


Рисунок 2 — Добавление узла X в АВЛ-дерево привело к разбалансировке в двух узлах. Но перестраивать будем только поддерево с корнем в выделенном узле

В зависимости от того, в какое поддерево опорного узла был добавлен новый узел, рассматривается четыре случая, которые можно разбить на две пары симметричных друг другу случаев. В каждом из них баланс восстанавливается с помощью одного или двух поворотов. На рисунках ниже опорный узел обведен кружком. Вновь добавленный узел обозначен буквой X.

1. Добавление в левое поддерево левого сына опорного узла.

Необходимо произвести правый поворот (R): опорный узел (B) поворачивается направо относительно своего левого сына (A).

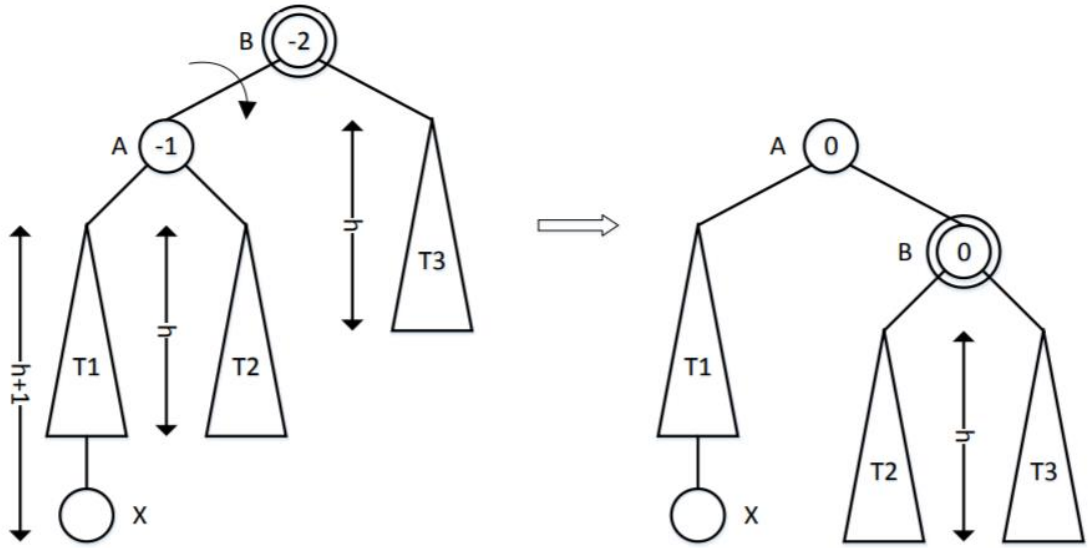


Рисунок 3 — Добавление узла в левое поддерево левого сына опорного узла и балансировка – правый поворот

Ситуация, требующая правого поворота после добавления нового узла, схематично изображена на рисунке 3. Поддеревья T1, T2 и T3 могут быть пустыми, но они обязательно должны иметь одинаковую высоту. Тогда до добавления узла X у опорного узла B высота левого поддерева будет на 1 больше, чем высота правого поддерева, а после добавления X баланс нарушится именно в B, а не в A. После поворота направо (по часовой стрелке) вокруг узла A узел B вместе с поддеревом T3 опустится на два уровня вниз относительно узла A. Самый нижний узел поддерева T3 окажется на одном уровне с узлом X. Так как поддерево с корнем в B станет новым правым сыном A, его бывший правый сын T2 перейдет к B. Высота T2 равна высоте T3, поэтому и в B, и в A показатель баланса станет равен 0.

2. Добавление в правое поддерево правого сына опорного узла.

Случай, симметричный предыдущему.

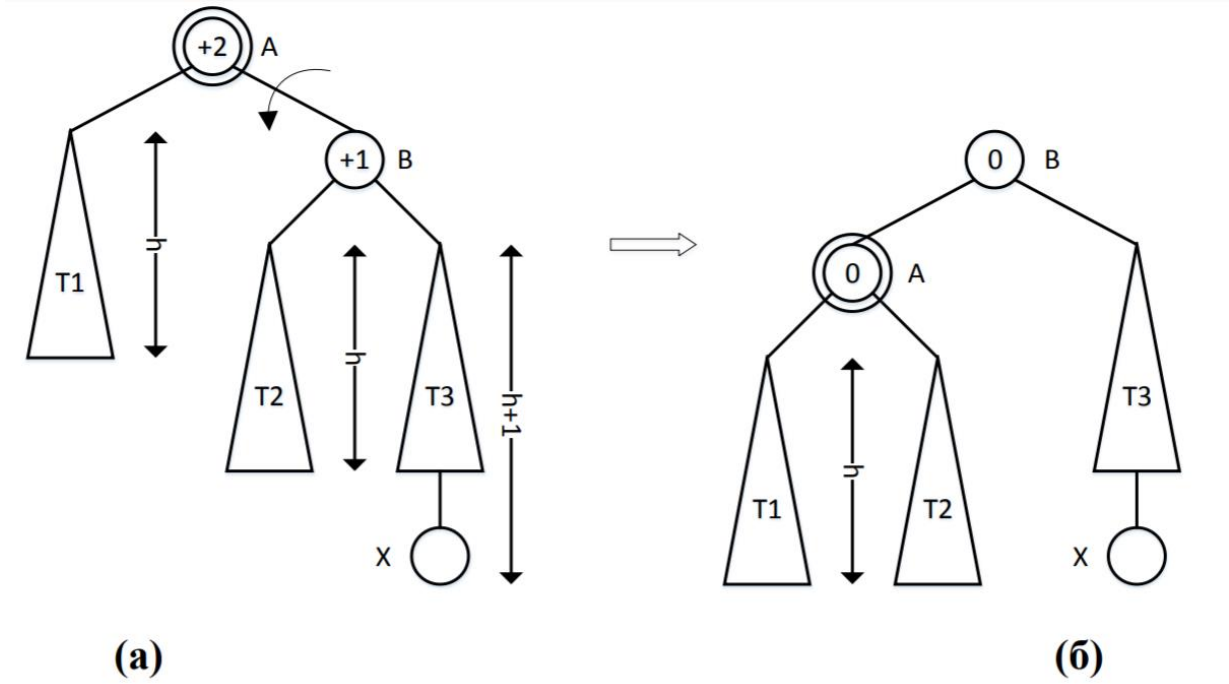


Рисунок 4 — Добавление узла в правое поддерево правого сына опорного узла и балансировка – левый поворот

3. Добавление в правое поддерево левого сына опорного узла.

Необходимо произвести двойной поворот — налево, потом направо (LR): сначала левый сын опорного узла (A) поворачивается налево относительно своего правого сына (B), а затем опорный узел (C) поворачивается направо относительно своего нового левого сына (B).

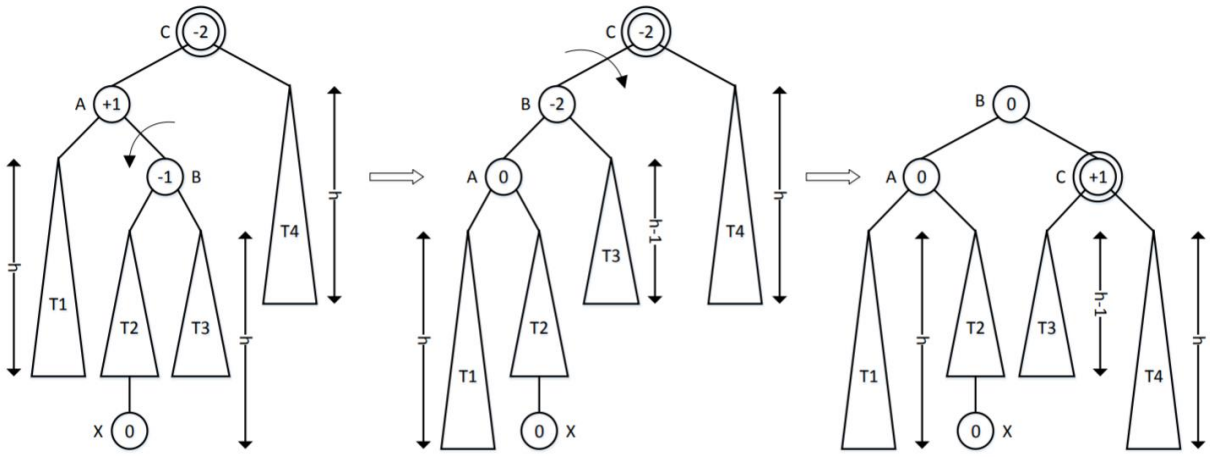


Рисунок 5 — Добавление узла в правое поддерево левого сына опорного узла и балансировка – левый-правый поворот

На рисунке 5 правым поддеревом левого сына опорного узла является поддерево с корнем в B. При этом узел X может быть добавлен как в поддерево T2, так и в поддерево T3 – тип поворота при балансировке от этого не изменится. Если до добавления узла X в правое поддерево узла A это правое поддерево было пусто, то в роли B выступает сам узел X. В результате двойного поворота узел B окажется наверху комбинации узлов ABC. За один поворот это сделать нельзя, потому как в начальный момент узел B является самым нижним в комбинации ABC. Поэтому первый поворот (A относительно B налево) поднимает B на один уровень вверх относительно C, а второй поворот (C относительно B направо) поднимает B еще на один уровень вверх относительно C. В итоге, слева у B окажется A с поддеревом T1, а справа у B окажется C с поддеревом T4. Высоты поддеревьев T1 и T4 совпадают. Поддерево T2 с узлом X и поддерево T3 в соответствии со свойствами дерева поиска прикрепятся к узлам A и C соответственно. Поскольку их высоты отличаются от высот T1 и T4 не более, чем на 1, баланс во всех узлах – A, B, C – по модулю не превысит 1. Баланс в B будет равен 0.

4. Добавление в левое поддерево правого сына опорного узла.

Случай, симметричный предыдущему.

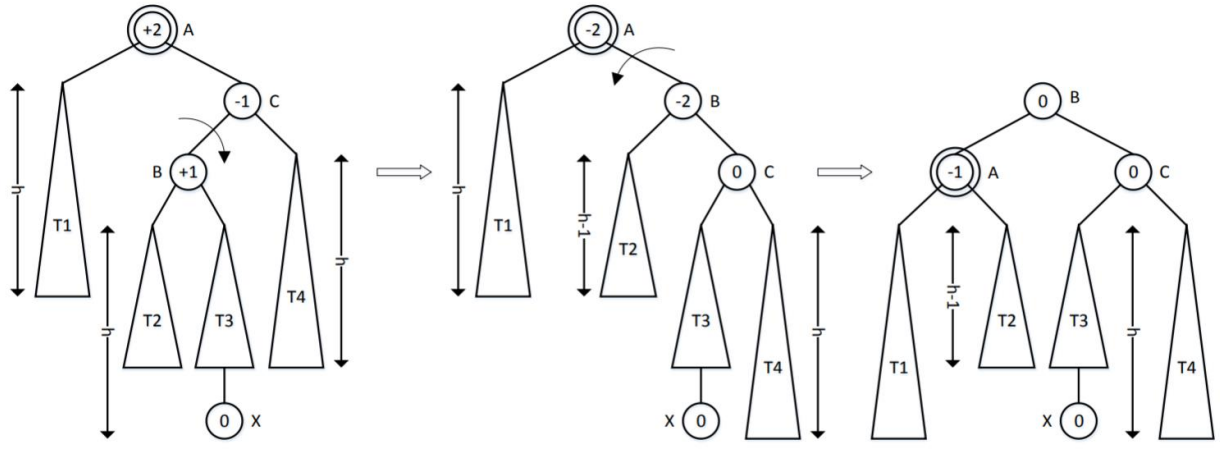


Рисунок 6 — Добавление узла в левое поддерево правого сына опорного узла и балансировка – правый-левый поворот

Резюмируя правила поворотов при выполнении операции балансировки АВЛ-деревьев, отметим общее правило: если добавление нового узла, приводящее к разбалансировке, происходит в левое поддерево левого сына опорного узла или в правое поддерево правого сына опорного узла, т.е. если стороны сына и внука опорного узла одноименны, то необходимо произвести одинарный поворот. Если добавление происходит в правое поддерево левого сына опорного узла или в левое поддерево правого сына опорного узла, т.е. стороны разноименны, то необходимо произвести двойной поворот. Мнемоническое правило для запоминания того, в какую сторону производится поворот:

- добавление в левое поддерево левого сына опорного узла – правый (R);
- добавление в правое поддерево левого сына опорного узла – левый-правый (LR);
- добавление в левое поддерево правого сына опорного узла – правый-левый (RL);
- добавление в правое поддерево правого поддерева сына опорного узла – левый (L).

Удаление узла из АВЛ-дерева

Непосредственно удаление узла из АВЛ-дерева происходит так же, как и удаление узла из обычного двоичного дерева поиска. Таким образом, если у узла менее двух сыновей, то удаляется сам узел, а если два сына, то удаляемым узлом становится его последователь, информация (ключ) из которого предварительно переписывается в удаляемый узел. Чтобы после удаления сохранились свойства АВЛ-дерева, возможно, понадобится выполнить балансировку. Для этого надо подниматься вверх по пути от удаленного узла к корню и проверять в этих узлах баланс. Если в узле баланс нарушен, то надо выполнить соответствующий поворот – одинарный или двойной. Остановить просмотр можно на том узле, в котором показатель баланса не поменялся. Это означает, что высота его поддерева, левого или правого, в котором

производилось удаление, не изменилась. При балансировке после удаления используются те же виды поворотов, что и после вставки узла в дерево.

Анализ алгоритма

Ниже представлена таблица, описывающая сложность различных операций в АВЛ-дереве.

Таблица 1 — Сложность операций в АВЛ-дереве

Операция	**Средний случай (average case)**	**Худший случай (worst case)**
Add(key, value)	$O(\log n)$	$O(\log n)$
Lookup(key)	$O(\log n)$	$O(\log n)$
Remove(key)	$O(\log n)$	$O(\log n)$
Min	$O(\log n)$	$O(\log n)$
Max	$O(\log n)$	$O(\log n)$

Сложность по памяти: $O(n)$

Оценка сложности поиска в АВЛ-дереве

Два крайних случая АВЛ-деревьев это: (а) совершенное дерево – все узлы имеют показатель баланса 0; (б) дерево Фибоначчи – все узлы, кроме листовых, имеют показатель баланса +1, либо все узлы, кроме листовых, имеют показатель баланса –1.

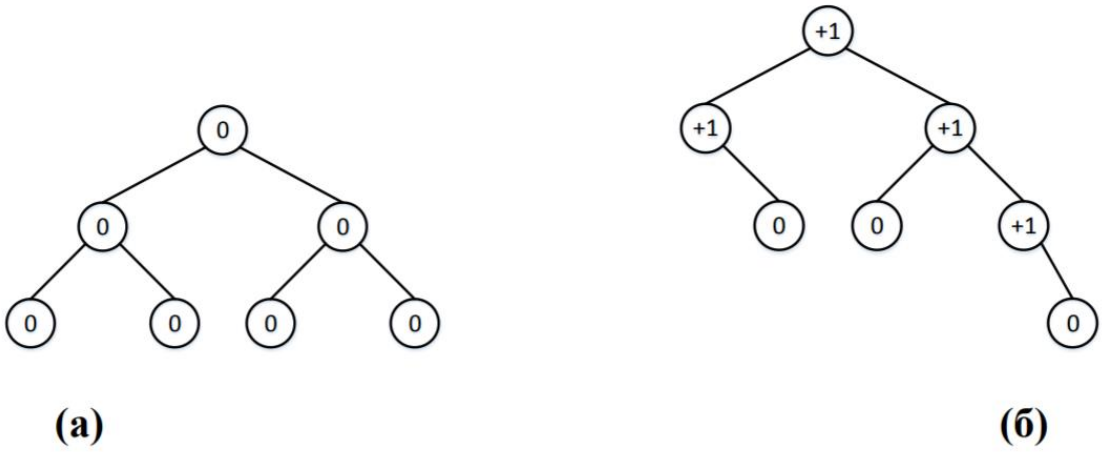


Рисунок 7 — Крайние случаи АВЛ-деревьев: (а) совершенное дерево; (б) дерево Фибоначчи

Не для каждого набора ключей можно построить совершенное дерево, равно как и не для каждого набора ключей можно построить дерево Фибоначчи. Но эти деревья позволяют оценить диапазон возможных высот АВЛ-деревьев. Совершенное дерево является частным случаем идеально сбалансированного дерева, поэтому оно имеет минимально возможную высоту для данного количества узлов. Дерево Фибоначчи, напротив, имеет максимально возможную высоту для данного количества узлов, при условии, что сохраняются свойства АВЛ-дерева.

Высота совершенного дерева вычисляется как

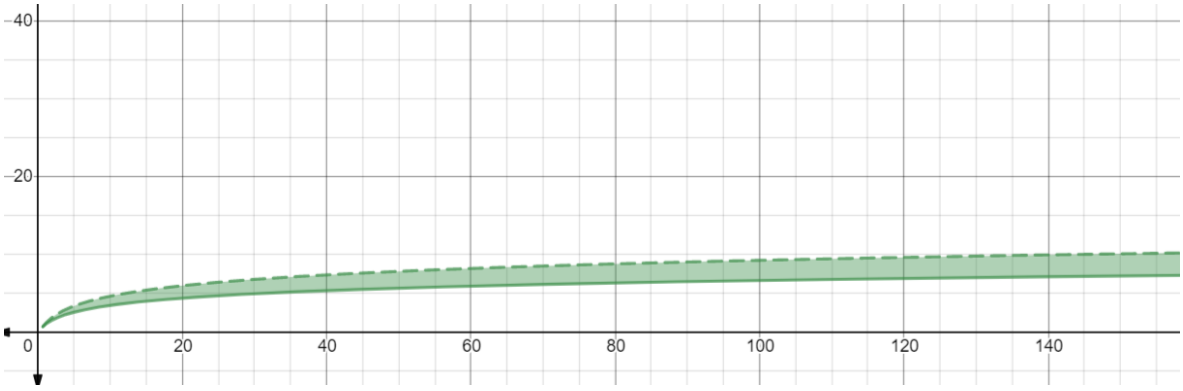
$$h = \log_2(m+1) + 1$$

Высота дерева Фибоначчи вычисляется как

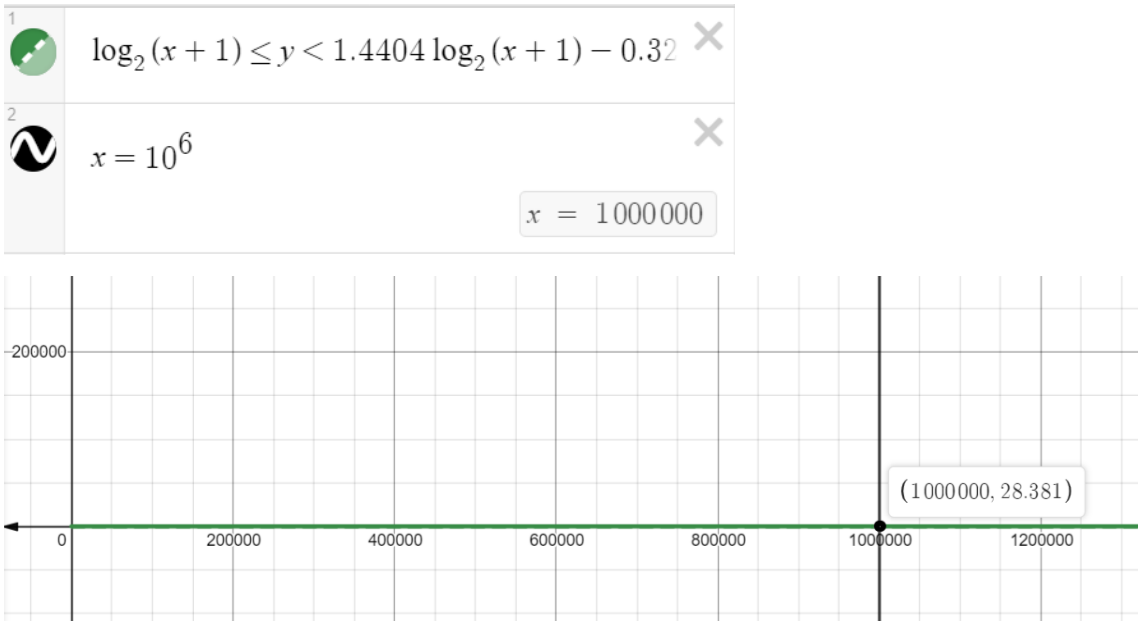
$$h < 1.44 \log_2(m+1) - 0.32$$

Таким образом, учитывая оценку высоты совершенного дерева и оценку высоты дерева Фибоначчи получаем, что высота h АВЛ-дерева из m узлов находится в диапазоне

$$\log_2(m+1) \leq h < 1.44 \log_2(m+1) - 0.32$$



Это соотношение задает оценку количества сравнений при поиске узла в АВЛ-дереве на пути от корня к этому узлу. Если, например, в АВЛ-дереве 10^6 вершин, то его высота, а, следовательно, и сложность поиска узла в нем, не превысит 29.



Программная реализация

В приложении 1 приведена реализация АВЛ-дерева на языке программирования C++, реализованы функции вставки и удаления узла, а также балансировки дерева.

Заключение

В ходе данной работы было рассмотрено такое дерево поиска как АВЛ-дерево, рассмотрены способы его балансировки при вставке и удалении узлов. Также был проведен анализ алгоритма и оценка сложности поиска.

Список литературы

1. Сбалансированные деревья поиска: Учебно-методическое пособие. – М.: Издательский отдел факультета ВМиК МГУ имени М.В. Ломоносова (лицензия ИД N 05899 от 24.09.2001 г.); МАКС Пресс, 2014. – 68 с.
2. AVL Tree // Programiz URL: https://www.programiz.com/dsa/avl-tree (дата обращения: 30.05.2020).

Приложение 1

```
#include <iostream>
using namespace std;

class Node {
   public:
  int key;
  Node *left;
  Node *right;
  int height;
};

int max(int a, int b);

// Calculate height
int height(Node *N) {
  if (N == NULL)
    return 0;
  return N->height;
}

int max(int a, int b) {
  return (a > b) ? a : b;
}

// New node creation
Node *newNode(int key) {
  Node *node = new Node();
  node->key = key;
  node->left = NULL;
  node->right = NULL;
  node->height = 1;
  return (node);
}

// Rotate right
Node *rightRotate(Node *y) {
  Node *x = y->left;
  Node *T2 = x->right;
  x->right = y;
  y->left = T2;
  y->height = max(height(y->left),
          height(y->right)) +
        1;
  x->height = max(height(x->left),
          height(x->right)) +
        1;
```

```
  return x;
}

// Rotate left
Node *leftRotate(Node *x) {
  Node *y = x->right;
  Node *T2 = y->left;
  y->left = x;
  x->right = T2;
  x->height = max(height(x->left),
          height(x->right)) +
        1;
  y->height = max(height(y->left),
          height(y->right)) +
        1;
  return y;
}

// Get the balance factor of each node
int getBalanceFactor(Node *N) {
  if (N == NULL)
    return 0;
  return height(N->left) -
       height(N->right);
}

// Insert a node
Node *insertNode(Node *node, int key) {
  // Find the correct postion and insert the node
  if (node == NULL)
    return (newNode(key));
  if (key < node->key)
    node->left = insertNode(node->left, key);
  else if (key > node->key)
    node->right = insertNode(node->right, key);
  else
    return node;

  // Update the balance factor of each node and
  // balance the tree
  node->height = 1 + max(height(node->left),
                height(node->right));
  int balanceFactor = getBalanceFactor(node);
  if (balanceFactor > 1) {
    if (key < node->left->key) {
      return rightRotate(node);
    } else if (key > node->left->key) {
      node->left = leftRotate(node->left);
```

```
      return rightRotate(node);
    }
  }
  if (balanceFactor < -1) {
    if (key > node->right->key) {
      return leftRotate(node);
    } else if (key < node->right->key) {
      node->left = rightRotate(node->left);
      return leftRotate(node);
    }
  }
  return node;
}

// Node with minimum value
Node *nodeWithMimumValue(Node *node) {
  Node *current = node;
  while (current->left != NULL)
    current = current->left;
  return current;
}

// Delete a node
Node *deleteNode(Node *root, int key) {
  // Find the node and delete it
  if (root == NULL)
    return root;
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
  else {
    if ((root->left == NULL) ||
      (root->right == NULL)) {
      Node *temp = root->left ? root->left : root->right;
      if (temp == NULL) {
        temp = root;
        root = NULL;
      } else
        *root = *temp;
      free(temp);
    } else {
      Node *temp = nodeWithMimumValue(root->right);
      root->key = temp->key;
      root->right = deleteNode(root->right,
                    temp->key);
    }
  }
```

```
  if (root == NULL)
    return root;

  // Update the balance factor of each node and
  // balance the tree
  root->height = 1 + max(height(root->left),
                         height(root->right));
  int balanceFactor = getBalanceFactor(root);
  if (balanceFactor > 1) {
    if (getBalanceFactor(root->left) >= 0) {
      return rightRotate(root);
    } else {
      root->left = leftRotate(root->left);
      return rightRotate(root);
    }
  }
  if (balanceFactor < -1) {
    if (getBalanceFactor(root->right) <= 0) {
      return leftRotate(root);
    } else {
      root->right = rightRotate(root->right);
      return leftRotate(root);
    }
  }
  return root;
}

// Print the tree
void printTree(Node *root, string indent, bool last) {
  if (root != nullptr) {
    cout << indent;
    if (last) {
      cout << "R----";
      indent += "   ";
    } else {
      cout << "L----";
      indent += "|  ";
    }
    cout << root->key << endl;
    printTree(root->left, indent, false);
    printTree(root->right, indent, true);
  }
}

int main() {
  Node *root = NULL;
  root = insertNode(root, 33);
```

```
  root = insertNode(root, 13);
  root = insertNode(root, 53);
  root = insertNode(root, 9);
  root = insertNode(root, 21);
  root = insertNode(root, 61);
  root = insertNode(root, 8);
  root = insertNode(root, 11);
  printTree(root, "", true);
  root = deleteNode(root, 13);
  cout << "After deleting " << endl;
  printTree(root, "", true);
}
```