Отчет №9. AVL-деревья

АВЛ-дерево — это прежде всего двоичное дерево поиска, ключи которого удовлетворяют стандартному свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла. Это значит, что для поиска нужного ключа в АВЛ-дереве можно использовать стандартный алгоритм. Для простоты дальнейшего изложения будем считать, что все ключи в дереве целочисленны и не повторяются.

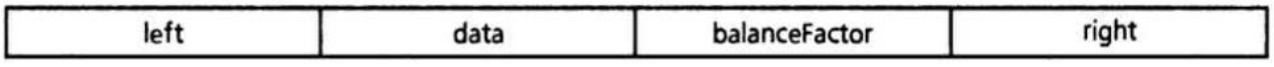
Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле: *для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу*.

Доказано, что этого свойства достаточно для того, чтобы высота дерева логарифмически зависела от числа его узлов: высота h АВЛ-дерева с n ключами лежит в диапазоне от $\log_2(n + 1)$ до $1.44 \log_2(n + 2) - 0.328$. А так как основные операции над двоичными деревьями поиска (поиск, вставка и удаление узлов) линейно зависят от его высоты, то получаем *гарантированную* логарифмическую зависимость времени работы этих алгоритмов от числа ключей, хранимых в дереве.

Напомним, что рандомизированные деревья поиска обеспечивают сбалансированность только в вероятностном смысле: вероятность получения сильно несбалансированного дерева при больших n хотя и является пренебрежимо малой, но остается *не равной нулю*.

Узлы AVL-дерева

AVL-деревья имеют представление, похожее на бинарные деревья поиска. Все операции идентичны, за исключением методов Insert и Delete, которые должны постоянно отслеживать соотношение высот левого и правого поддеревьев узла. Для сохранения этой информации мы расширили определение объекта TreeNode, включив поле balanceFactor (показатель сбалансированности), которое содержит разность высот правого и левого поддеревьев.



AVLTreeNode

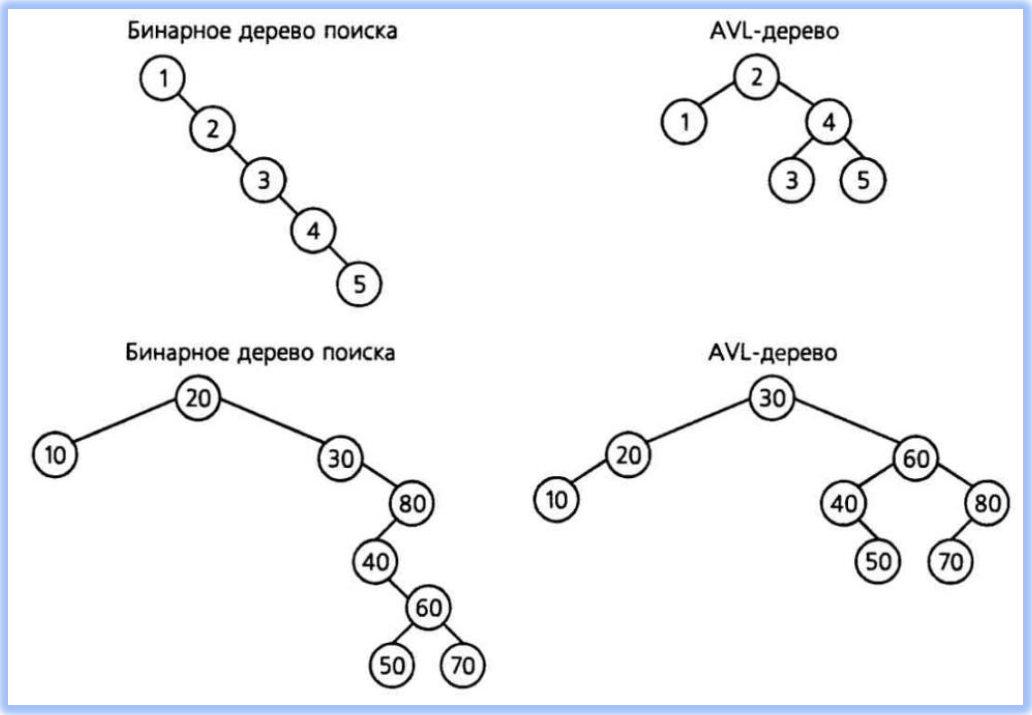
```
balanceFactor=height(right subtree)- height(left subtree)
```

Если balanceFactor отрицателен, то узел "перевешивает влево", так как высота левого поддерева больше, чем высота правого поддерева. При положительном balanceFactor узел "перевешивает вправо". Сбалансированный по высоте узел имеет balanceFactor = 0. В AVL-дереве показатель сбалансированности должен быть в диапазоне [-1, 1].

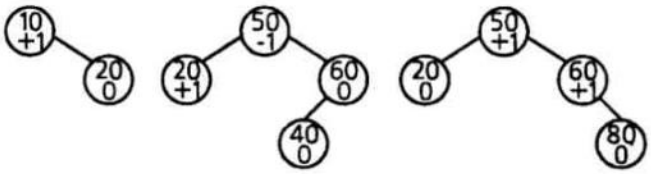
На рис. 13.4 изображены AVL-деревья с пометками -1, 0 и +1 на каждом узле, показывающими относительный размер левого и правого поддеревьев.

-1: Высота левого поддерева на 1 больше высоты правого поддерева.

0: Высоты обоих поддеревьев одинаковы.



Используя свойства наследования, можно образовать класс AVLTreeNode на базе класса TreeNode. Объект типа AVLTreeNode наследует поля из класса TreeNode и добавляет к ним поле balanceFactor. Данные-члены left и right класса TreeNode являются защищенными, поэтому AVLTreeNode или другие производные классы имеют к ним доступ. Класс AVLTreeNode и все сопровождающие его программы находятся в файле avltree.h.



Спецификация класса AVLTreeNode

ОБЪЯВЛЕНИЕ

```
// наследник класса TreeNode
template <class T>
class AVLTreeNode: public TreeNode<T>
{
  private:
    // дополнительный член класса
    int balanceFactor;
    // используются методами класса AVLTree и позволяют
    // избегать "перевешивания" узлов

    AVLTreeNode<T>* & Left(void);
    AVLTreeNode<T>* & Right(void);

  public:
    // конструктор
    AVLTreeNode(const T& item, AVLTreeNode<T> *lptr = NULL,
                AVLTreeNode<T> *rptr = NULL, int balfac = 0);

    // возвратить левый/правый указатель узла типа TreeNode,
    // в качестве указателя узла типа AVLTreeNode; выполнить
    // приведение типов
    AVLTreeNode<T> *Left(void) const;
    AVLTreeNode<T> *Right(void) const;

    // метод для доступа к новому полю данных
    int GetBalanceFactor(void);

    // методы класса AVLTree должны иметь доступ к Left и Right
    friend class AVLTree<T>;
};
```

ОПИСАНИЕ

Элемент данных balanceFactor является закрытым, так как обновлять его должны только сбалансированные операции включения и исключения.

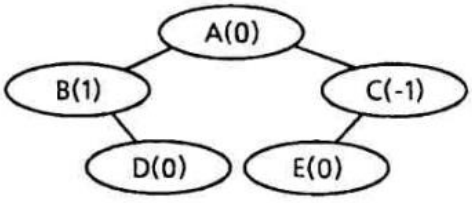
Параметры, передаваемые в конструктор, содержат данные для базовой структуры типа TreeNode. По умолчанию параметр balfac равен 0.

Доступ к полям указателей осуществляется с помощью методов Left и Right. Новые определения для этих методов обязательны, поскольку они возвращают указатель на стуктуру AVLTreeNode.

Основные причины, по которым деструктор объявляется виртуальным, обсуждались в разделе 12.3. Поскольку класс AVLTree образован на базе класса BinSTree, будем использовать деструктор базового класса и ClearList. Эти методы удаляют узлы с помощью оператора delete. В каждом случае указатель ссылается на объект типа AVLTreeNode, а не TreeNode. Если деструктор базового класса TreeNode виртуальный, то при вызове delete используется динамическое связывание и удаляется объект типа AVLTreeNode.

ПРИМЕРЫ

```
AVLTreeNode<char> *root;      // корень AVL-дерева

// эта функция создает AVL-дерево, изображенное ниже.
// каждому узлу присваивается показатель сбалансированности
```



```
void MakeAVLCharTree(AVLTreeNode<char>* &root)
{
  AVLTreeNode<char> *a, *b, *c, *d, *e;

  e = new AVLTreeNode<char>('E', NULL, NULL, 0);
  d = new AVLTreeNode<char>('D', NULL, NULL, 0);
  c = new AVLTreeNode<char>('C', e, NULL, -1);
  b = new AVLTreeNode<char>('B', NULL, d, 1);
  a = new AVLTreeNode<char>('A', b, c, 0);
  root = a;
}
```

Реализация класса AVLTreeNode. Конструктор класса AVLTreeNode вызывает конструктор базового класса и инициализирует balanceFactor.

```
// конструктор. инициализирует balanceFactor и базовый класс.
// нулевые начальные значения полей указателей
// (по умолчанию) инициализируют узел как лист. template <class T>
AVLTreeNode<T>::AVLTreeNode (const T& item,
     AVLTreeNode<T> *lptr, AVLTreeNode<T> *rptr, int balfac):
  TreeNode<T>(item, lptr, rptr), balanceFactor(balfac)
{}
```

Методы Left и Right в классе AVLTreeNode упрощают доступ к полям данных. При попытке обратиться к левому сыну с помощью базового метода Left возвращается указатель на объект типа TreeNode. Чтобы получить указатель на узел сбалансированного дерева, требуется преобразование типов. Например,

```
AVLTreeNode<T> *p, *q;

q = p->Left();                          // недопустимая операция

q = (AVLTreeNode<T> *)p->Left();        // необходимое приведение типа
```

Во избежание постоянного преобразования типа указателей мы определяем методы Left и Right для класса AVLTreeNode, возвращающие указатели на объекты типа AVLTreeNode.

```
template <class T>
AVLTreeNode<T>* AVLTreeNode::Left(void)
{
  return ((AVLTreeNode<T> *)left;
}
```

Класс AVLTree

AVL-дерево представляет списковую структуру, похожую на бинарное дерево поиска, с одним дополнительным условием: дерево должно оставаться сбалансированным по высоте после каждой операции включения или удаления. Поскольку AVL-дерево является расширенным бинарным деревом поиска, класс AVLTree строится на базе класса BinSTree и является его наследником.

Методы Insert и Delete должны подменяться для выполнения AVL-условия. Кроме того, в производном классе определяются конструктор копирования и перегруженный оператор присваивания, так как мы строим дерево с большей узловой структурой.

Спецификация класса AVLTree

ОБЪЯВЛЕНИЕ

```
// Значения показателя сбалансированности узла
const int leftheavy = -1;
const int balanced = 1;
const int rightheavy = 1;

// производный класс поисковых деревьев
template <class T>
class AVLTree: public BinSTree<T>

{
  private:
    // выделение памяти
    AVLTreeNode<T> *GetAVLTreeNode(const T& item,
          AVLTreeNode<T> *lptr, AVLTreeNode<T> *rptr);

    // используется конструктором копирования и оператором присваивания
    AVLTreeNode<T> *CopyTree(AVLTreeNode<T> *t);

    // используется методами Insert и Delete для восстановления
    // AVL-условий после операций включения/исключения
    void SingleRotateLeft (AVLTreeNode<T>* &p);
    void SingleRotateRight (AVLTreeNode<T>* &p);
    void DoubleRotateLeft (AVLTreeNode<T>* &p);
    void DoubleRotateRight (AVLTreeNode<T>* &p);
    void UpdateLeftTree (AVLTreeNode<T>* &tree,
                         int &reviseBalanceFactor);
    void UpdateRightTree (AVLTreeNode<T>* &tree,
                          int &reviseBalanceFactor);

    // специальные версии методов Insert и Delete
    void AVLInsert(AVLTreeNode<T>* &tree,
           AVLTreeNode<T>* newNode, int &reviseBalanceFactor);
    void AVLDelete(AVLTreeNode<T>* &tree,
           AVLTreeNode<T>* newNode, int &reviseBalanceFactor);

  public:
    // конструкторы
    AVLTree(void);
    AVLTree(const AVLTree<T>& tree);

    // оператор присваивания
    AVLTree<T>& operator= (const AVLTree<T>& tree);

    // стандартные методы обработки списков
    virtual void Insert(const T& item);
    virtual void Delete(const T& item);
};
```

ОПИСАНИЕ

Константы leftheavy, balanced и rightheavy используются в операциях вставки/удаления для описания показателя сбалансированности узла.

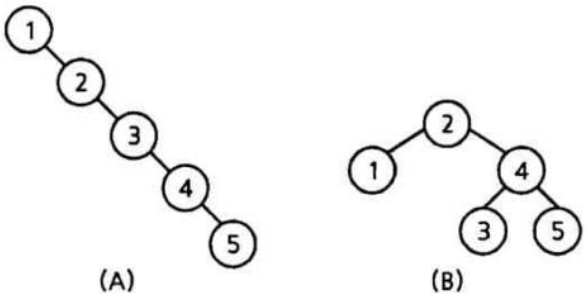
Метод GetAVLTreeNode управляет выделением памяти для класса. По умолчанию balanceFactor нового узла равен нулю.

В этом классе заново определяется функция CopyTree для использования с конструктором копирования и перегруженным оператором присваивания. Несмотря на то, что алгоритм идентичен алгоритму для функции CopyTree класса BinSTree, новая версия корректно создает расширенные объекты типа AVLTreeNode при построении нового дерева.

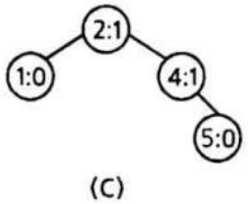
Функции AVLInsert и AVLDelete реализуют методы Insert и Delete, соответственно. Они используют закрытые методы наподобие SingleRotateLeft. Открытые методы Insert и Delete объявлены как виртуальные и подменяют соответствующие функции базового класса. Остальные операции наследуются от класса BinSTree.

ПРИМЕР

```
AVLTree<int>  avltree;          // AVLTree-список целых чисел
BinSTree<int> bintree;          // BinSTree-список целых чисел

for (int i=1; i<=5; i++)
```




```
{
  bintree.Insert(i);    // создать дерево A
  avltree.Insert(i);    // создать дерево B
}
avltree.Delete(3);      // удалить 3 из AVL-дерева

// функция AVLPrintTree эквивалентна функции вертикальной распечатки
// дерева из гл. 11. кроме собственно данных для каждого узла
// распечатываются показатели сбалансированности. дерево (C) есть дерево (B)
// без удаленного узла 3. AVLPrintTree находится в файле avltree.h

AVLPrintTree((AVLTreeNode<int> *)avltree.GetRoot(), 0);
```



Распределение памяти для AVLTree

Класс AVLTree образован от класса BinSTree и наследует большинство его операций. Для создания расширенных объектов типа AVLTreeNode мы разработали отдельные методы выделения памяти и копирования.

```
// разместить в памяти объект типа AVLTreeNode. прервать программу,
// если во время выделения памяти произошла ошибка
template <class T>
AVLTreeNode<T> *AVLTree<T>::GetAVLTreeNode(const T& item,
                    AVLTreeNode<T> *lptr, AVLTreeNode<T> *rptr)
{
  AVLTreeNode<T> *p;

  p = new AVLTreeNode<T> (item, lptr, rptr);
  if (p == NULL)
  {
    cerr << "Ошибка выделения памяти!" << endl;
    exit(1);
  }
  return p;
}
```

Для удаления узлов AVL-дерева достаточно методов базового класса. Метод DeleteTree из класса BinSTree задействует виртуальный деструктор класса TreeNode.

Метод Insert класса AVLTree. Преимущество AVL-деревьев состоит в их сбалансированности, которая поддерживается соответствующими алгоритмами вставки/удаления. Опишем метод Insert для класса AVLTree, который перекрывает одноименную операцию базового класса BinSTree. При реализации метода Insert для запоминания элемента используется рекурсивная функция AVLInsert. Сначала приведем код метода Insert на C++, а затем сосредоточим внимание на рекурсивном методе AVLInsert, реализующем алгоритм Адельсона-Вельского и Ландиса.

```
template <class T>
void AVLTree<T>::Insert(const T& item)
{
  // объявить указатель AVL-дерева, используя метод
  // базового класса GetRoot.
  // произвести приведение типов для указателей
  AVLTreeNode<T>  *treeRoot = (AVLTreeNode<T> *)GetRoot(),
 *newNode;

 // флажок,  используемый функцией AVLInsert для перебалансировки узлов
 int  reviseBalanceFactor = 0;

 // создать новый узел AVL-дерева с нулевыми  полями указателей
 newNode = GetAVLTreeNode(item, NULL, NULL);

 //  вызвать рекурсивную процедуру для фактической вставки элемента
  AVLInsert(treeRoot, newNode, reviseBalancefactor);

  // присвоить новые значения элементам данных базового класса
  root = treeRoot;
  current = newNode;
  size++;
}
```

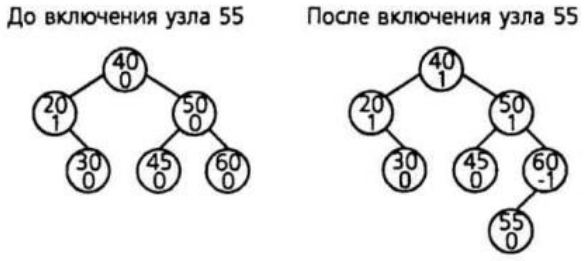
Ядром алгоритма включения является рекурсивный метод AVLInsert. Как и его аналог в классе BinSTree, этот метод осуществляет прохождение левого поддерева, если item меньше данного узла, и правого поддерева, если item больше или равен данному узлу. Эта закрытая функция имеет параметр с именем tree, в котором находится запись текущего узла при сканировании, новый узел для вставки в дерево, флажок revisebalanceFactor. При сканировании левого или правого поддерева некоторого узла, этот флажок является признаком изменения любого параметра balanceFactor в поддереве. Если да, то нужно проверить, сохранилась ли AVL-сбалансированность всего дерева. Если в результате включения нового узла она оказалась нарушенной, то мы обязаны восстановить равновесие. Данный алгоритм рассматривается на ряде примеров.

Алгоритм AVL-вставки. Процесс включения является почти таким же, что и для бинарного дерева поиска. Осуществляется рекурсивный спуск по левым и правым сыновьям, пока не встретится пустое поддерево, а затем производится пробное включение нового узла в этом месте. В течение этого процесса мы посещаем каждый узел на пути поиска от корневого к новому элементу.

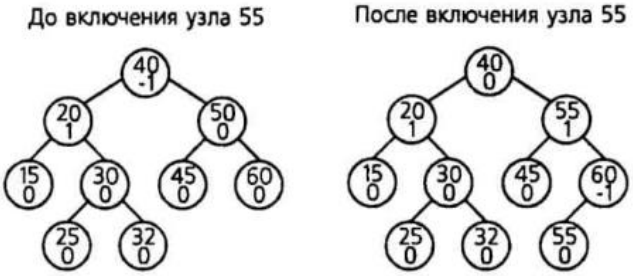
Поскольку процесс рекурсивный, обработка узлов ведется в обратном порядке. При этом показатель сбалансированности родительского узла можно скорректировать после изучения эффекта от добавления нового элемента в одно из поддеревьев. Необходимость корректировки определяется для каждого узла, входящего в поисковый маршрут. Есть три возможных ситуации.

В первых двух случаях узел сохраняет сбалансированность и реорганизация поддеревьев не требуется, а нужно лишь скорректировать показатель сбалансированности данного узла. В третьем случае расбалансировка дерева требует одинарного или двойного поворотов узлов.

Случай 1. Узел на поисковом маршруте изначально является сбалансированным (balanceFactor = 0). После включения в поддерево нового элемента узел стал перевешивать влево или вправо в зависимости от того, в какое поддерево было произведено включение. Если элемент был включен в левое поддерево, показателю сбалансированности присваивается -1, а если в правое, то 1. Например, на пути 40–50–60 каждый узел сбалансирован. После включения узла 55 показатели сбалансированности изменяются.



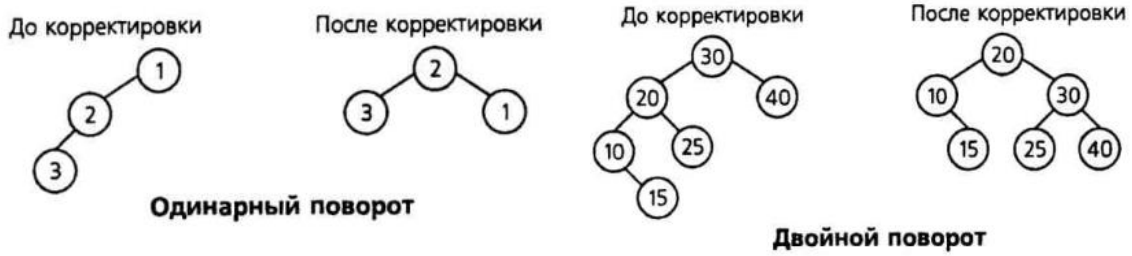
Случай 2. Одно из поддеревьев узла перевешивает, и новый узел включается в более легкое поддерево. Узел становится сбалансированным. Сравните, например, состояния дерева до и после включения узла 55.



Рассмотрим пример. Предположим, дерево разбалансировалось слева и мы восстанавливаем равновесие, вызывая одну из функций поворота вправо. Разбалансировка справа влечет за собой симметричные действия.

Сказанное иллюстрируется следующими рисунками. При разработке алгоритма поворота мы включили дополнительные детали.

Метод AVLInsert. Продвигаясь вдоль некоторого пути для вставки нового узла, этот рекурсивный метод распознает все три указанных выше случая корректировки. При нарушении условия сбалансированности восстановление равновесия осуществляется с помощью функций UpdateLeftTree и UpdateRightTree.



Одинарный поворот

Двойной поворот

```
template <class T>
void AVLTree<T>::AVLInsert(AVLTreeNode<T>* &tree,
                AVLTreeNode<T>* newNode, int &reviseBalanceFactor)
{
  // флажок "Показатель сбалансированности был изменен"
  int rebalanceCurrNode;

  // встретилось пустое поддерево. пора включать новый узел
  if (tree == NULL)
  {
    // вставить новый узел
    tree = newNode;

    // объявить новый узел сбалансированным
    tree->balanceFactor = balanced;

    // сообщить об изменении показателя сбалансированности
    reviseBalanceFactor = 1;
  }
  // рекурсивно спускаться по левому поддереву,
  // если новый узел меньше текущего
  else if (newNode->data < tree->data)
  {
    AVLInsert(tree->Left(), newNode, rebalanceCurrNode);
    // проверить, нужно ли корректировать balanceFactor
    if (rebalanceCurrNode)
    {
      // включение слева от узла, перевешивающего влево. будет нарушено
      // условие сбалансированности; выполнить поворот (случай 3)
      if (tree->balanceFactor == leftheavy)
        UpdateLeftTree(tree, reviseBalanceFactor);

      // вставка слева от сбалансированного узла.
      // узел станет перевешивать влево (случай 1)
      else if (tree->balanceFactor == balanced)
      {
        tree->balanceFactor = leftheavy;
        reviseBalanceFactor = 1;
      }
      // вставка слева от узла, перевешивающего вправо.
      // узел станет сбалансированным (случай 2)
```

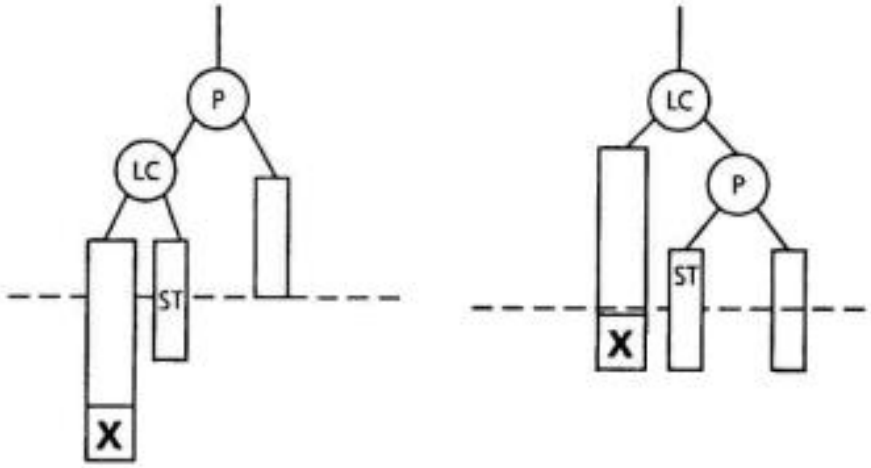
```
      else
      {
        tree->balanceFactor = balanced;
        reviseBalanceFactor = 0;
      }
    }
    else
      // перебалансировка не требуется. не опрашивать предыдущие узлы
      reviseBalanceFactor = 0;
  }
  // иначе рекурсивно спускаться по правому поддереву
  else if (newNode->data < tree->data)
  {
    AVLInsert(tree->Right(), newNode, rebalanceCurrNode);
    // проверить, нужно ли корректировать balanceFactor
    if (rebalanceCurrNode)
    {
      // вставка справа от узла, перевешивающего вправо. будет нарушено
      // условие сбалансированности; выполнить поворот (случай 3)
      if (tree->balanceFactor == rightheavy)
        UpdateRightTree(tree, reviseBalanceFactor);

      // вставка справа от сбалансированного узла.
      // узел станет перевешивать вправо (случай 1)
      else if (tree->balanceFactor == balanced)
      {
        tree->balanceFactor = rightheavy;
        reviseBalanceFactor = 1;
      }
      // вставка справа от узла, перевешивающего влево.
      // узел станет сбалансированным (случай 2)
      else
      {
        tree->balanceFactor = balanced;
        reviseBalanceFactor = 0;
      }
    }
    else
      // перебалансировка не требуется. не опрашивать предыдущие узлы
      reviseBalanceFactor = 0;
  }
}
```

Метод AVLInsert распознает случай 3, когда нарушается AVL-условие. Для выполнения перебалансировки используются методы UpdateLeftTree и UpdateRightTree. Они выполняют одинарный или двойной поворот для уравновешивания узла, а затем сбрасывают флажок reviseBalanceFactor. Перед тем как обсудить специфические детали поворотов, приведем программный код функции UpdateLeftTree.

```
template <class T>
void AVLTree<T>::UpdateLeftTree(AVLTreeNode<T>* &p,
                                int reviseBalanceFactor)
{
  AVLTreeNode<T> *lc;

  lc = p->Left();
  // перевешивает левое поддерево?
  if (lc->balanceFactor == leftheavy)
  {
    SingleRotateRight(p);       // однократный поворот
```

```
    reviseBalanceFactor = 0;
  }
  // перевешивает правое поддерево?
  else if (lc->balanceFactor == rightheavy)
  {
    // выполнить двойной поворот
    DoubleRotateRight(p);
    // теперь корень уравновешен
    reviseBalanceFactor = 0;
  }
}
```

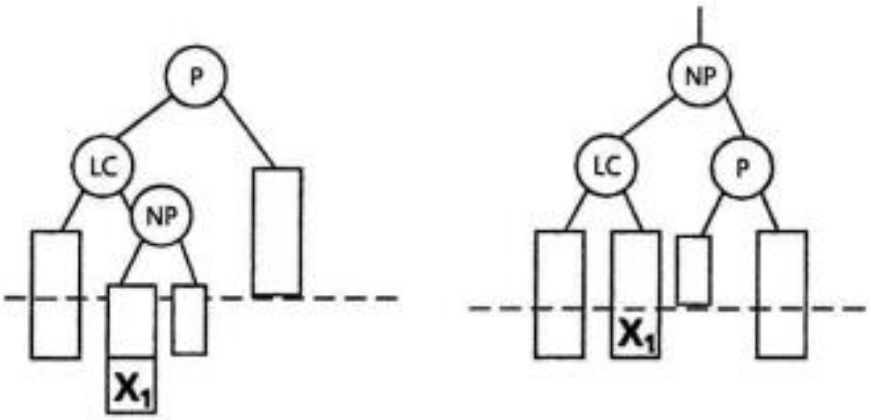
Повороты. Повороты необходимы, когда родительский узел P становится расбалансированным. **Одинарный поворот вправо (single right rotation)** происходит тогда, когда родительский узел P и его левый сын LC начинают перевешивать влево после включения узла в позицию X. В результате такого поворота LC замещает своего родителя, который становится правым сыном. Бывшее правое поддерево узла LC (ST) присоединяется к P в качестве левого поддерева. Это сохраняет упорядоченность, так как узлы в ST больше или равны узлу LC, но меньше узла P. Поворот уравновешивает как родителя, так и его левого сына.



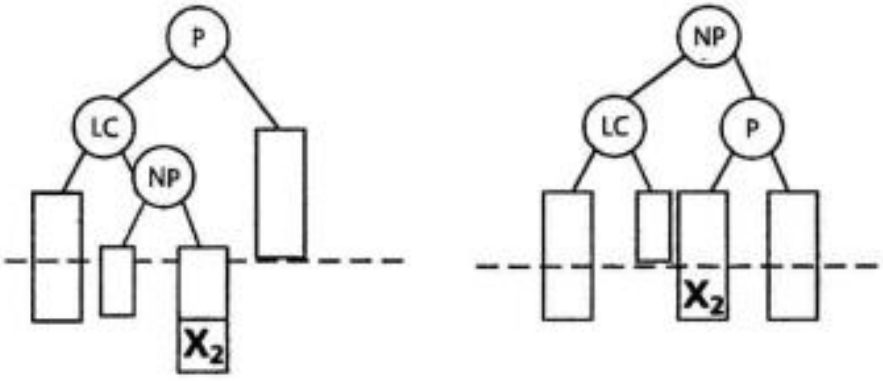

```
// выполнить поворот по часовой стрелке вокруг узла p.
// сделать lc новой точкой вращения
template <class T>
void AVLTree<T>::SingleRotateRight (AVLTreeNode<T>* &p)
{
  // левое, перевешивающее поддерево узла p
  AVLTreeNode<T> *lc;

  // назначить lc левым поддеревом
  lc = p->Left();

  // скорректировать показатель сбалансированности для
  // родительского узла и его левого сына
  p->balanceFactor = balanced;
  lc->balanceFactor = balanced;

  // правое поддерево узла lc в любом случае должно оставаться справа
  // от lc. выполнить это условие, сделав st левым поддеревом узла p
  p->Left() = lc->Right();

  // переместить p в правое поддерево узла lc.
  // сделать lc новой точкой вращения.
  lc->Right() = p;
  p = lc;
}
```

Попытка включить узел 5 в изображенное ниже AVL-дерево нарушает AVL-условие для узла 30 из равновесия. Одновременно левое поддерево узла 15 (LC) становится перегруженным. Для переупорядочения узлов вызывается процедура SingleRotateRight. В результате родительский узел (30) становится сбалансированным, а узел 10 — перевешивающим влево.

Двойной поворот вправо (double rigyn rotation) происходит тогда, когда родительский узел (P) становится перевешивающим влево, а его левый сын (LC) — перевешивающим вправо. NP — корень правого перевешивающего поддерева узла LC. Тогда в результате поворота узел NP замещает родительский узел. На следующих далее рисунках показаны два случая включения нового узла в качестве сына узла NP. В обоих случаях NP становится родительским узлом, а бывший родитель P становится правым сыном NP.

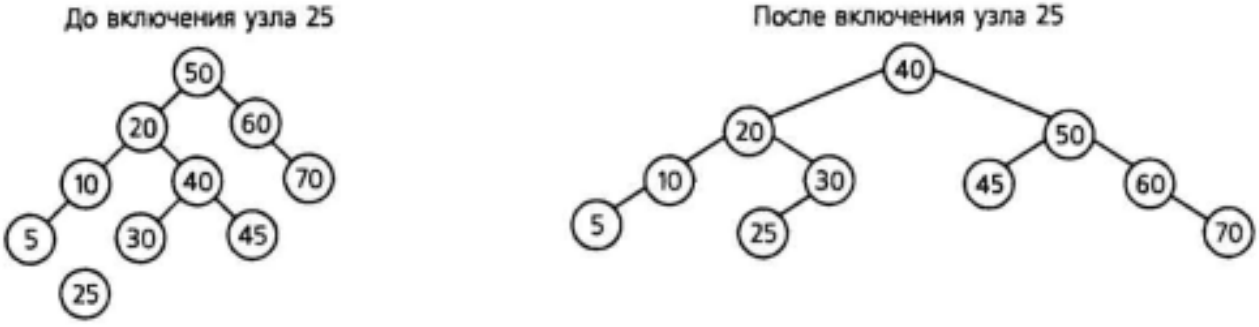


На верхней схеме мы видим сдвиг узла X_1, после того как он был вставлен в левое поддерево узла NP. На нижней схеме изображено перемещение узла X_2 после его включения в правое поддерево NP.



```
// двойной поворот вправо вокруг узла p
template <class T>
void AVLTree<T>::DoubleRotateRight (AVLTreeNode<T>* &p)
{
  // два поддерева, подлежащих повороту
  AVLTreeNode<T> *lc, *np;

  // узел lc <= узел np < узел p
  lc = p->Left();      // левый сын узла p
  np = lc->Right();    // правый сын узла lc

  // обновить показатели сбалансированности в узлах p, lc и np
  if (np->balanceFactor == rightheavy)
  {
    p->balanceFactor = balanced;
    lc->balanceFactor = rightheavy;
  }
  else
  {
    p->balanceFactor = rightheavy;
    lc->balanceFactor = balanced;
  }
  np->balanceFactor = balanced;

  // перед тем как заменить родительский узел p,
  // следует отсоединить его старых детей и присоединить новых
  lc->Right() = np->Left();
  np->Left() = lc;
  p->Left() = np->Right();
  np->Right() = p;
  p = np;
}
```

Двойной поворот иллюстрируется на изображенном ниже дереве. Попытка включить узел 25 разбалансирует корневой узел 50. В этом случае узел 20 (LC) приобретает слишком высокое правое поддерево и требуется двойной поворот. Новым родительским узлом (NP) становится узел 40. Старый родительский узел становится его правым сыном и присоединяет к себе узел 45, который также переходит с левой стороны дерева.



Оценка сбалансированных деревьев

Ценность AVL-деревьев зависит от приложения, поскольку они требуют дополнительных затрат на поддержание сбалансированности при включении или исключении узлов. Если в дереве постоянно происходят вставки и удаления элементов, эти операции могут значительно снизить быстродействие. С другой стороны, если ваши данные превращают бинарное дерево поиска в вырожденное, вы теряете поисковую эффективность и вынуждены использовать AVL-дерево.

Для сбалансированного дерева не существует наихудшего случая, так как оно является почти полным бинарным деревом. Сложность операции поиска составляет $O(\log_2 n)$. Опыт показывает, что повороты требуются примерно в половине случаев включений и удалений. Сложность балансировки обусловливает применение AVL-деревьев только там, где поиск является доминирующей операцией.

Программа 13.4. Оценка AVL-деревьев

Эта программа сравнивает сбалансированное и обычное бинарное дерево поиска, каждое из которых содержит N случайных чисел. Они хранятся в едином массиве и включаются в оба дерева. Для каждого элемента массива осуществляется его поиск в обоих деревьях. Длины поисковых путей суммируются, а затем подсчитывается средняя длина поиска по каждому дереву.

Программа прогоняется на 1000- и на 10000-элементном массивах. Обратите внимание, что на случайных данных поисковые характеристики AVL-дерева несколько лучше. В самом худшем случае вырожденное дерево поиска, содержащее 1000 элементов, имеет среднюю глубину 500, в то время как средняя глубина AVL-дерева всегда равна 9.

```
#include <iostream.h>
#include "bstree.h"
#include "avltree.h"
#include "random.h"

// загрузить массив, бинарное поисковое дерево и AVL-дерево
// одинаковыми множествами, состоящими из n случайных чисел от 0 до 999
void SetupLists(BinSTree<int> &Tree1, AVLTree<int> &Tree2,
                int A[], int n)
{
  int i;
  RandomNumber rnd;

  // запомнить случайное число в массиве A, а также включить его
  // в бинарное дерево поиска и в AVL-дерево
  for (i=0; i<n; i++)
  {
    A[i] = rnd.Random(1000);
    Tree1.Insert(A[i]);
    Tree2.Insert(A[i]);
  }
}

// поиск элемента item на дереве t. накапливается суммарная длина поиска
template <class T>
void PathLength(TreeNode<T> *t, long &totallength, int item)
{
  // возврат, если элемент найден или отсутствует в списке
  if (t == NULL || t->data == item)
    return;
  else
  {
    // перейти на следующий уровень.
    // увеличить суммарную длину пути поиска
    totallength++;
    if (item < t->data)
```

```
      PathLength(t->Left(), totallength, item);
    else
      PathLength(t->Right(), totallength, item);
  }
}
void main(void);
{
  // переменные для деревьев и массива
  BinSTree<int> binTree;
  AVLTree<int>  avlTree;
  int *A;

  // суммарные длины поисковых путей элементов массива
  // в бинарном дереве поиска и в AVL-дереве
  long totalLengthBintree = 0, totalLengthAVLTree = 0;
  int n, i;

  cout << "Сколько узлов на дереве? ";
  cin >> n;

  // загрузить случайными числами массив и оба дерева
  SetupLists(binTree, avlTree, A, n);

  for (i=0; i<n; i++)
  {
    PathLength(binTree.GetRoot(), totalLengthBintree, A[i]);
    PathLength((TreeNode<int> *)avlTree.getRoot(),
               totalLengthAVLTree, A[i]);
  }
  cout << "Средняя длина поиска для бинарного дерева = "
       << float(totalLengthBintree)/n << endl;
  cout << "Средняя длина поиска для сбалансированного дерева = "
       << float(totalLengthAVLTree)/n << endl;
}
/*

<Прогон #1 программы 13.4>

Сколько узлов на дереве? 1000
Средняя длина поиска для бинарного дерева = 10.256
Средняя длина поиска для сбалансированного дерева = 7.901

<Прогон #2 программы 13.4>

Сколько узлов на дереве? 10000
Средняя длина поиска для бинарного дерева = 12.2822
Средняя длина поиска для сбалансированного дерева = 8.5632

*/
```

Источники:

1. У. Топп, У. Форд – Структуры данных в C++

2. https://habr.com/ru/post/150732/ - наглядная информация по балансировке, удалению и тд, много полезного кода

3. https://blog.skillfactory.ru/glossary/avl-derevo/ - про все что уже было написано и про особенности.