



Алгоритмы и структуры данных

Лекция 12. AVL-деревья

Антон Штанюк (к.т.н, доцент)

27 апреля 2021 г.

Нижегородский государственный технический университет им. Р.Е. Алексеева
Институт радиоэлектроники информационных технологий
Кафедра "Компьютерные технологии в проектировании и производстве"

Сбалансированные бинарные деревья

AVL деревья

Изменение структуры дерева при добавлении элементов

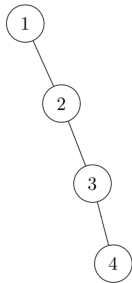
Список литературы

Сбалансированные бинарные деревья

Для бинарного дерева поиска, рассмотренного ранее, возможно состояние вырожденности, при котором высота (глубина) дерева будет максимальной, а количество потомков на каждом уровне не более одного. Такое дерево можно получить, если добавлять в него элементы, отсортированные по возрастанию (или по убыванию)

```
int main()
{
    BST<int> tree;

    for(int i=1; i<=4; i++)
        tree.add(i);
    tree.print();
    return 0;
}
```



Вырожденные деревья теряют главное преимущество - скорость поиска, превращаясь в обычные списки. Поэтому, необходимо исправлять структуру дерева, уменьшая количество уровней. Для этой цели разработаны **сбалансированные** деревья, которые автоматически поддерживают минимальную высоту, меняя автоматически свою структуру.

Среди подобных деревьев можно выделить два, наиболее распространенных вида:

1. AVL-деревья
2. RB-деревья ("красно-черные")

У каждого вида сбалансированных деревьев есть свои достоинства и недостатки. Самое главное, что они обеспечивают скорость поиска порядка $O(\log_2 n)$, затрачивая больше времени при добавлении новых элементов из-за автоматической балансировки.

AVL деревья

Особенностью AVL-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.

Если вставка или удаление элемента приводит к нарушению сбалансированности дерева, то необходимо выполнить его балансировку.

Адельсон-Вельский Г.М., Ландис Е.М.
Один алгоритм организации информации
Доклады АН СССР. – 1962.
Т. 146, No 2. – С. 263–266.

Поле **key** хранит ключ узла, поле **height** — высоту поддерева с корнем в данном узле, поля **left** и **right** — указатели на левое и правое поддерева. Простой конструктор создает новый узел (высоты 1) с заданным ключом **k**.

Хранение высоты в узле дерева оправдано тем, что его не нужно вычислять при каждом обращении.

```
#include <iostream>

template<typename T>
struct NODE // структура для представления узлов дерева
{
    T key;                // ключ узла
    unsigned char height; // высота поддерева
    NODE* left;
    NODE* right;
    NODE(T k):key(k),left(nullptr),right(nullptr),height(1) {} //
    конструктор
};
```

Традиционно, узлы AVL-дерева хранят не высоту, а разницу высот правого и левого поддеревьев (так называемый *balance factor*), которая может принимать только три значения -1, 0 и 1.

Программная реализация дерева весьма нетривиальна из-за наличия процедуры балансировки.

```
template<typename T>
class AVLTree
{
private:
    unsigned char height(NODE<T>* p); // селектор высоты
    int bfactor(NODE<T>* p);          // вычисление баланса узла
    void fixheight(NODE<T>* p);       // установка поля высоты узла

    NODE<T>* rotateright(NODE<T>* p); // правый поворот вокруг p
    NODE<T>* rotateleft(NODE<T>* q);  // левый поворот вокруг q

    NODE<T>* balance(NODE<T>* p);     // балансировка узла p
    NODE<T>* insert(NODE<T>* p, T k); // вставка ключа k в дерево с корнем p
    void print_dfs(NODE<T>* p, int level); // печать структуры дерева
public:
    AVLTree():root(nullptr){};
    void insert(T k);
    void print();
private:
    NODE<T>* root;
};
```

Функция **height** возвращает значение высоты для текущего поддеревя. Для пустого дерева возвращается 0.

```
template<typename T>
unsigned char AVLTree<T>::height(NODE<T>* p)
{
    return p?p->height:0;
}
```

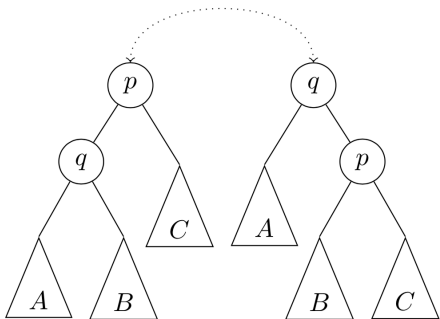
Функция **bfactor** возвращает разницу высот между правым и левым поддеревом.

```
template<typename T>
int AVLTree<T>::bfactor(NODE<T>* p)
{
    return height(p->right)-height(p->left);
}
```

Функция **fixheight** обновляет поле **height** для текущего узла. Для этого запрашиваются высоты поддеревьев, выбирается наибольшая и к ней прибавляется 1.

```
template<typename T>
void AVLTree<T>::fixheight(NODE<T>* p)
{
    unsigned char hl = height(p->left);
    unsigned char hr = height(p->right);
    p->height = (hl>hr?hl:hr)+1;
}
```

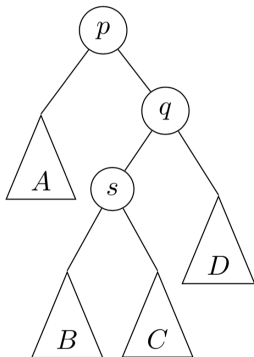
Наиболее важные операции: простые повороты вправо и влево, выполняемые функциями **rotateright** и **rotateleft**. В эти функции передаются адреса текущих корневых элементов, а возвращаются адреса новых корневых элементов. В результате высоты левого (или правого) поддерева уменьшается на 1.




```
template<typename T>
NODE<T>* AVLTree<T>::rotateright(NODE<T>* p)
{
    NODE<T>* q = p->left;
    p->left = q->right;
    q->right = p;
    fixheight(p);
    fixheight(q);
    return q;
}
```

```
template<typename T>
NODE<T>* AVLTree<T>::rotateleft(NODE<T>* q)
{
    NODE<T>* p = q->right;
    q->right = p->left;
    p->left = q;
    fixheight(q);
    fixheight(p);
    return p;
}
```

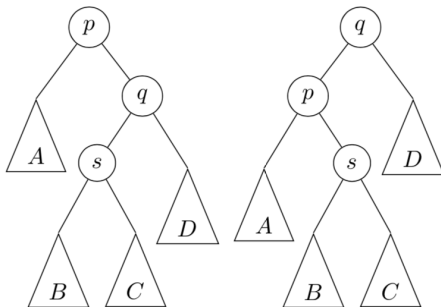
Функция **balance** вычисляет расбалансировку и выполняет необходимые повороты. В общем случае необходимо рассмотреть деревья следующего вида.



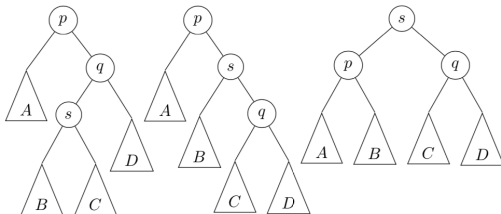
Производится вычисление баланса между левым и правым поддеревом для p . Если дисбаланс присутствует, то нужно найти баланс для правого или левого поддерева, в зависимости от числа (+2 или -2). Если получаем +2, то высота правого поддерева больше высоты левого. Если получаем -2, то высота правого поддерева оказывается меньше высоты левого.

Теперь измеряем баланс для поддерева: если в первом случае для q окажется, что высота D окажется больше высоты s (или равной ей), то достаточно будет сделать простой поворот **влево**.

Аналогично для второго случая: если окажется, что высота A окажется больше высоты s , то можно ограничиться простым поворотом вправо.



Если перечисленные выше условия не выполняются, то нужно совершить двойной поворот: в первом случае сперва **вправо**, потом **влево**, а во втором случае: сначала **влево**, а потом **вправо**.



```
template<typename T>
NODE<T>* AVLTree<T>::balance(NODE<T>* p)
{
    fixheight(p);
    if( bfactor(p)==2 )
    {
        if( bfactor(p->right) < 0 )
            p->right = rotateright(p->right);
        return rotateleft(p);
    }
    if( bfactor(p)==-2 )
    {
        if( bfactor(p->left) > 0 )
            p->left = rotateleft(p->left);
        return rotateright(p);
    }
    return p; // балансировка не нужна
}
```

Функция **insert** выполняет вставку узла в дерево. После вставки узла автоматически выполняется балансировка (в случае необходимости).

```
template<typename T>
NODE<T>* AVLTree<T>::insert(NODE<T>* p, T k)
{
    if( !p ) return new NODE<T>(k);
    if( k<p->key )
        p->left = insert(p->left,k);
    else
        p->right = insert(p->right,k);
    return balance(p);
}
```


Вставка для пользователя выполняется функцией-оберткой.

```
template<typename T>
void AVLTree<T>::insert(T k)
{
    root=insert(root,k);
}
```

Для того, чтобы визуально следить за структурой AVL-дерева, напомним функцию, которая позволяет распечатать в консоли данные, расположенные по уровням. Функция эта будет рекурсивной, поэтому напомним к ней обертку.

```
template<typename T>
void AVLTree<T>::print_dfs(NODE<T> *p, int level)
{
    int i;
    if (p == nullptr)
        return;
    for (i = 0; i < level; i++)
        std::cout<<"    ";
    std::cout<<p->key<<std::endl;
    print_dfs(p->left, level + 1);
    print_dfs(p->right, level + 1);
}
```

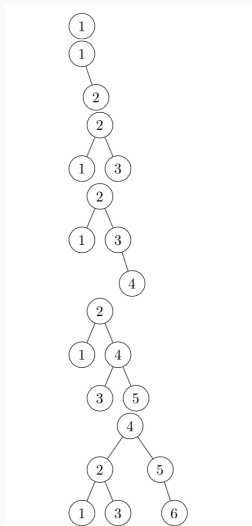
```
template<typename T>
void AVLTree<T>::print()
{
    print_dfs(root, 0);
}
```

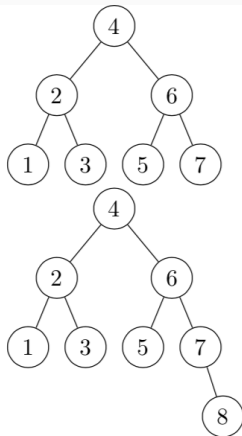
Изменение структуры дерева при добавлении элементов

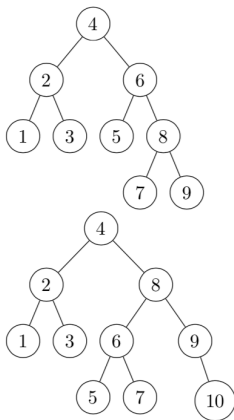
Теперь рассмотрим задачу, когда в дерево добавляют элементы, расположенные по возрастанию. После каждого добавления будем печатать содержимое дерева.

```
int main()
{
    AVLTree<int> tree;
    for(int i=1;i<=10;i++)
    {
        tree.insert(i);
        tree.print();
    }
    return 0;
}
```






В результате получены изображения структуры дерева после добавления каждого элемента с 1 до 10.












Список литературы

-  Кормен Т., Лейзерсон Ч., Ривест Р.
Алгоритмы: построение и анализ
МЦНМО, Москва, 2000
-  Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы:
построение и анализ.
2-е изд. — М.: «Вильямс», 2006
-  Википедия
Алгоритм
<http://ru.wikipedia.org/wiki/Алгоритм>
-  Википедия
Список алгоритмов
http://ru.wikipedia.org/wiki/Список_алгоритмов
-  Традиция
Задача коммивояжёра
<http://traditio.ru/wiki/Задача>

-  Википедия
NP-полная задача
<http://ru.wikipedia.org/wiki/NP-полная>
-  Серджвик Р.
Фундаментальные алгоритмы на C++. Части 1-4
Diasoft, 2001
-  Седжвик Р.
Фундаментальные алгоритмы на C. Анализ/Структуры данных/Сортировка/Поиск
СПб.: ДиаСофтЮП, 2003
-  Седжвик Р.
Фундаментальные алгоритмы на C. Алгоритмы на графах
СПб.: ДиаСофтЮП, 2003
-  Ахо А., Хопкрофт Д., Ульман Д. Структуры данных и алгоритмы.
Издательский дом «Вильямс», 2000



Кнут Д.

Искусство программирования, том 1. Основные алгоритмы
3-е изд. — М.: «Вильямс», 2006



Кнут Д.

Искусство программирования, том 2. Получисленные методы
3-е изд. — М.: «Вильямс», 2007



Кнут Д.

Искусство программирования, том 3. Сортировка и поиск
2-е изд. — М.: «Вильямс», 2007



Кнут Д.

Искусство программирования, том 4, выпуск 3. Генерация всех сочетаний и разбиений
М.: «Вильямс», 2007



Кнут Д.

Искусство программирования, том 4, выпуск 4. Генерация всех деревьев. История комбинаторной генерации

М.: «Вильямс», 2007