Binary Trees

Last updated: November 14th 2018, at 3.17pm

Contents

1 Sorted Binary Trees

1.1 Behaviour

The concept of binary trees should be familiar from Computer Science & Mathematics. Here we will look at the process of building up a binary tree, inserting values one by one. We will use `String`s as an example in the discussion below, but the same approach will work for any `Comparable` type.

The values will be insterted in such a way that the tree is *ordered* — i.e. all values in the left subtree will be less than or equal to the value in the root (e.g. for `String`s will appear alphabetically before the root value), and all values in the right subtree will be greater than or equal to the root value[1].

[1]This definition is deliberately ambiguous, and does not specify in which subtree a value equal to the root value should be found, or even whether it is permissible for the same value

The algorithm for insertion will be:

- If the tree is currently empty, insert the data at the root;
- If the data to be inserted is *less* than the data at the root, insert the data into the left subtree;
- If the data to be inserted is *greater* than the data at the root, insert the data into the right subtree;
- There are different possibilities for data equal to the root value.

  If the data to be inserted and the data at the root are identical (consistently) do one of the following:

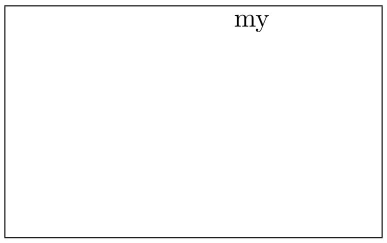
  - do not insert it;
  - insert it into the left subtree;
  - insert it into the right subtree.

1.2 Example

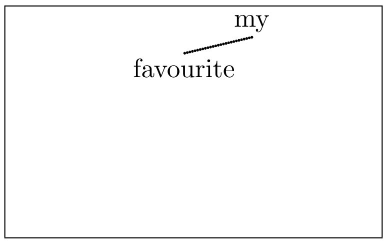
In this example we will look at a binary tree containing `String`s, but a sorted binary tree could contain any `Comparable`s. The `String`s we will insert are:

"my" "favourite" "module" "is" "algorithms" "processes" "and" "data"

Inserting "my". The tree is currently empty, so put "my" at the root.

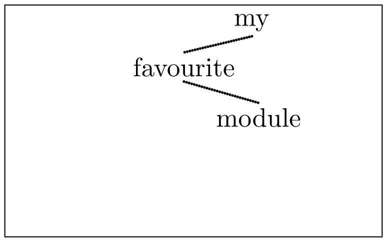


Inserting "favourite". The value at the root is "my", which comes after "favourite", so insert "favourite" into the left subtree. The left subtree is currently empty, so create a new root for it, containing "favourite".

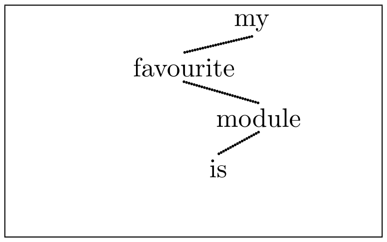


to appear more than once in the tree. These are details that may differ from implementation to implementation.

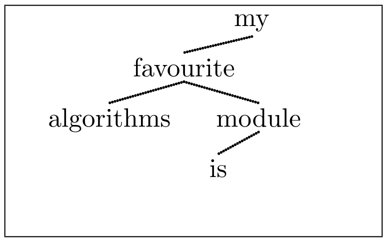
Inserting “module” to the left of “my”, and to the right of “favourite”.



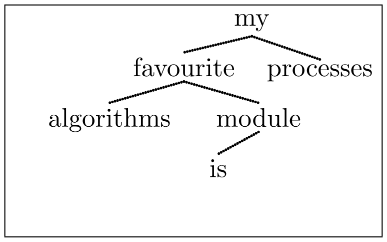
Inserting “is”.



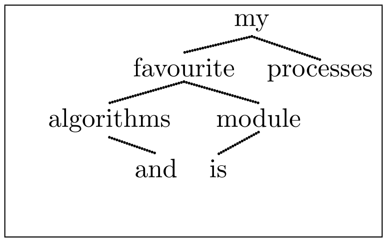
Inserting “algorithms”.



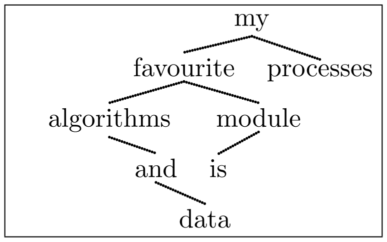
Inserting “processes”.



Inserting “and”.



Inserting “data”.



2 Implementing Binary Trees

2.1 Interface

We want to implement the following interface:

```
public interface BTree <T extends Comparable<?  super T>> {
    // Insert the given value into this tree
    public void insert(T value);

    // Is the tree empty?
    public boolean isEmpty();

    // Does the tree contain the given value?
    public boolean contains(T value);

    ...more methods follow
}


// getter and setter methods

// @return the value held at the root of this tree
public T getValue();
// @return the left (right) subtree of this tree
public BTree<T> getLeft();
public BTree<T> getRight();

// set the value at the root
public void setValue(T value);
// set the left (right) subtree
public void setLeft(BTree<T> tree);
public void setRight(BTree<T> tree);


// Traversal
```

```
public List<T> traverse();
```

2.2 Tree Nodes

We need to define a single node in a binary tree.

```
public class TreeNode<T extends Comparable<? super T>> {
   T value;
   BTree<T> left, right;

   public TreeNode(T value) {
      this.value = value;
      left = new BinaryTree<T>();
      right = new BinaryTree<T>();
   }

   public TreeNode(T value,BTree<T> left,BTree<T> right) {
      this.value - value;
      this.left = left;
      this.right = right;
   }

   ...plus getter and setter methods
}
```

A node contains a single item of data and two pointers to (possibly empty) left and right subtrees. Note that it makes the implementation of the trees easier if these pointers are never null pointers, but may point to an empty tree.

2.3 Implementation

```
public void insert(T value) {
   if (isEmpty()) {
      root = new TreeNode(value);
   } else if (value.compareTo(getValue()) < 0) {
      getLeft().insert(value);
   } else {
      getRight().insert(value);
   }
}
```

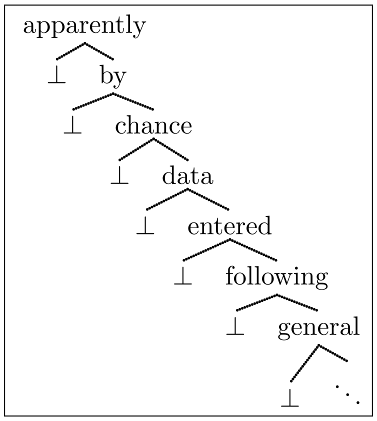
This implementation allows multiple entries of the same value, and places entries equal to the root value in the right subtree.

3 AVL Trees

3.1 Worst Case Binary Trees

Binary trees provide fast access to sorted data. However they can show pathological behaviour. What happens if we enter the data:

> apparently by chance data entered following general happenstance in juxtaposed key listing may not often provide quality resultant structures thus upsetting very worried . . .



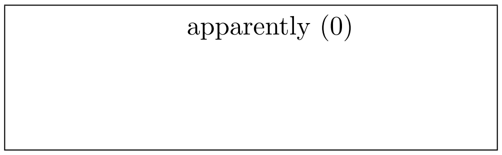
This is essentially a linked list, with the corresponding slow access times, and, in fact, with the `null` nodes (shown here as ⊥), is *less* efficient in space usage than a linked list.

3.2 Balance Factor

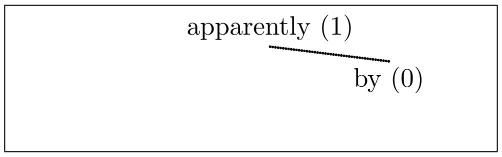
AVL trees[2] are trees that "stay in balance".

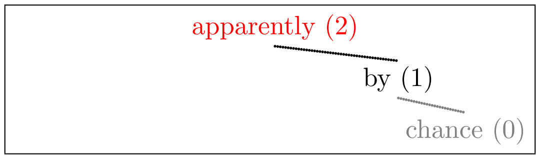
The "balance factor" of a node is the difference in height of its two subtrees. An AVL tree will only allow balance factors of -1, 0 or 1 (left, balanced, or right).

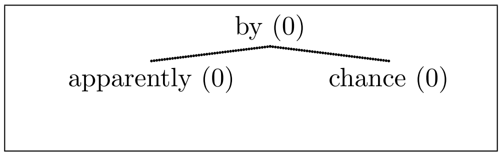
3.3 Example

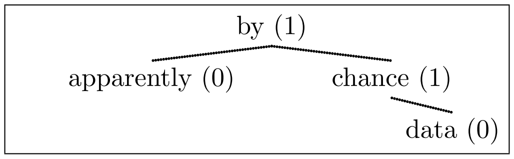


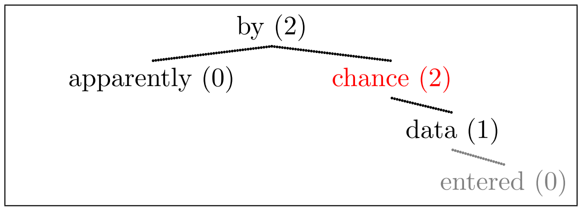
[2](Георгий Адельсон-Вельский & Евгений Ландис, 1962)

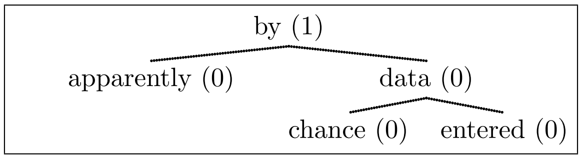

apparently (1)
by (0)

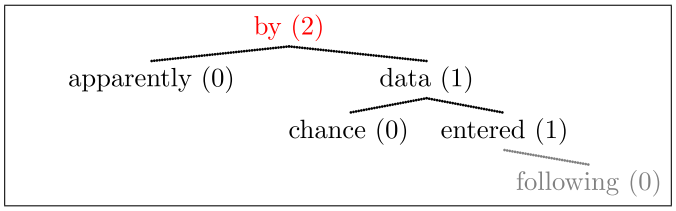

apparently (2)
by (1)
chance (0)

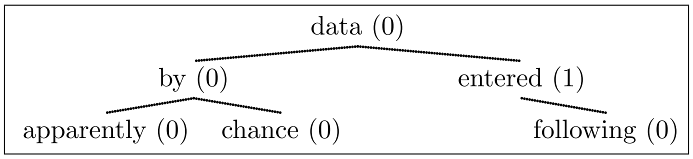

by (0)
apparently (0)
chance (0)

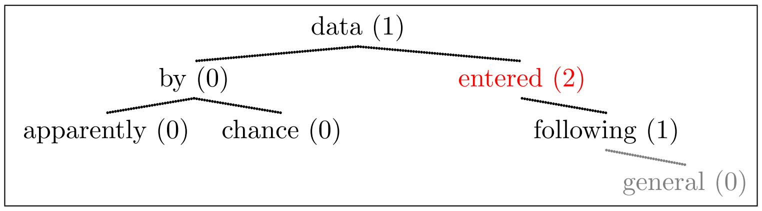

by (1)
apparently (0)
chance (1)
data (0)

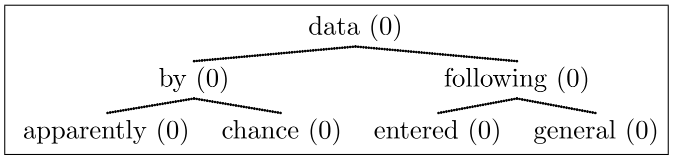

by (2)
apparently (0)
chance (2)
data (1)
entered (0)


by (1)
apparently (0)
data (0)
chance (0)
entered (0)


by (2)
apparently (0)
data (1)
chance (0)
entered (1)
following (0)


data (0)
by (0)
entered (1)
apparently (0)
chance (0)
following (0)

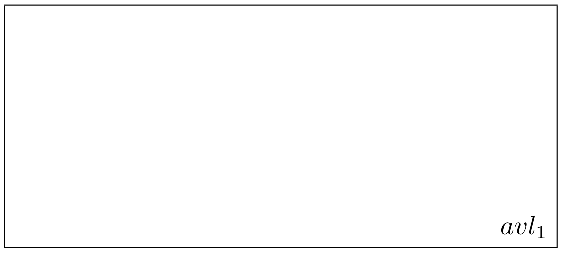




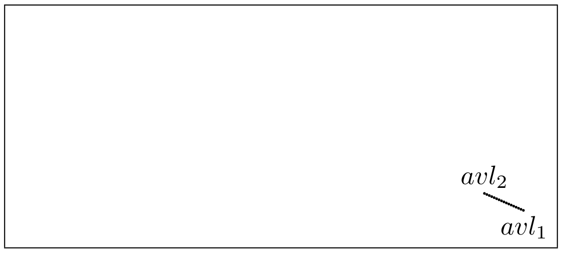
3.4 Worst Case AVL Tree

What is the worst case for AVL trees — i.e. what is the greatest depth of tree we can construct for a given number of nodes? Equivalently, what is the smallest number of nodes that we can fit into an AVL tree of a given depth?

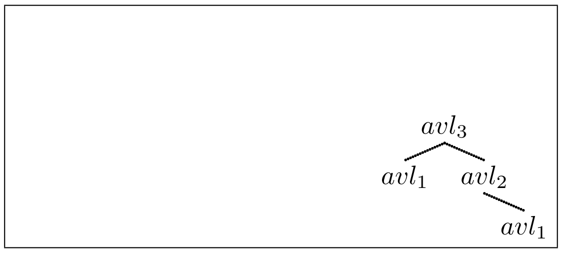
Obviously, the smallest number of nodes in an AVL tree of depth one is one. We will call this tree avl_1



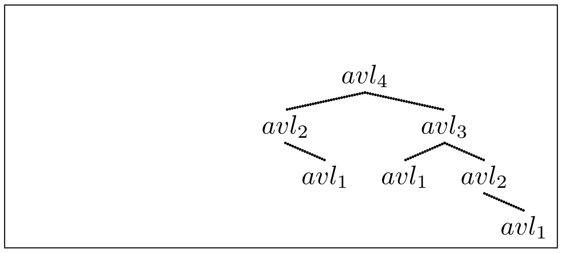
For an AVL of depth two, we can take a right subtree of depth one, and a left subtree of depth zero (i.e. an empty left subtree, and join them with a new root node. We will call this tree avl_2



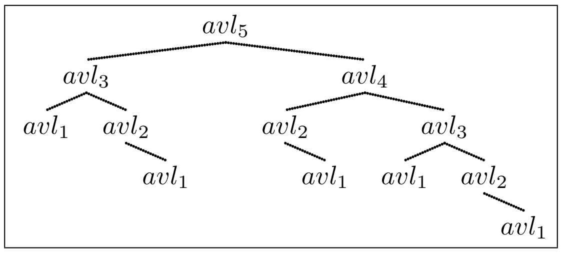
For depth three we need a subtree (e.g. on the right) of depth two. To have an AVL tree the other subtree must have depth one. To minimise we use avl_1 and avl_2 for these subtrees, and again join them with a new root node.



In general, to create the next worst case AVL tree we combine the two previous worst case trees, using a new root node, so that avl_4 has, of course, a root node with avl_3 and avl_2 as its two subtrees...



...and avl_5 uses avl_4 and avl_3.



Let n_k be the number of nodes in avl_k. Then

$$\begin{aligned} n_1 &= 1 \\ n_2 &= 2 \\ n_i &= n_{i-1} + n_{i-2} + 1, \text{ for } i > 2 \end{aligned}$$

These numbers are clearly closely related to the Fibonacci numbers, and analysis shows that:

$$\text{for large } i\text{: } n_i \geq \frac{1.62^{i-2}}{\sqrt{5}}$$

or, equivalently

$$d \leq 1.44 \log n$$

where d is the height of an AVL tree, and n is the number of nodes.

3.5 Building AVL Trees

- Each node is aware of its own balance factor
- An insert notifies if the tree inserted into has become deeper
- Node uses this information to adjust balance factor
- If node is now too out of balance it must be rebalanced

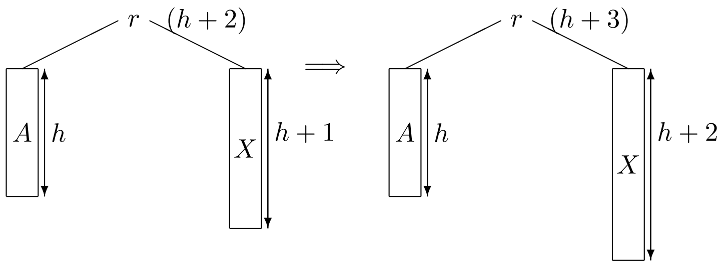
3.5.1 Adjusting the balance factor

If the inserted tree has increased in depth:

- if it *was* balanced it now "leans toward" the side of the inserted tree;
- if it leant in the opposite direction it is now balanced;
- otherwise it is now out of balance, and needs rebalancing.

3.5.2 Rebalancing AVL trees

We will only look at rebalancing trees "heavy" on the right. Rebalancing "left heavy" trees is analogous.



Assume that this is the lowest node in the tree that is excessively out of balance.

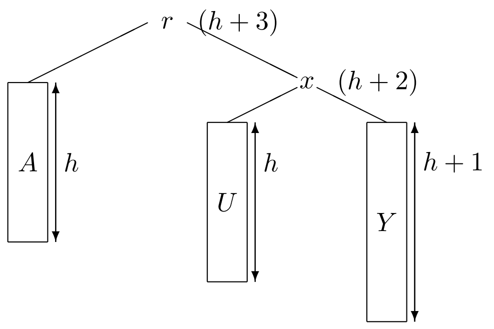
- depth$(A) = h$
- depth$(X) = h+2$ ($h+1$ before item was added)
- depth(oldtree) $= h+2$

There are three possibilities:

- balance$(X) = 1$
- balance$(X) = 0$
- balance$(X) = -1$

Actually, one of these is impossible, so there are only two possibilities.

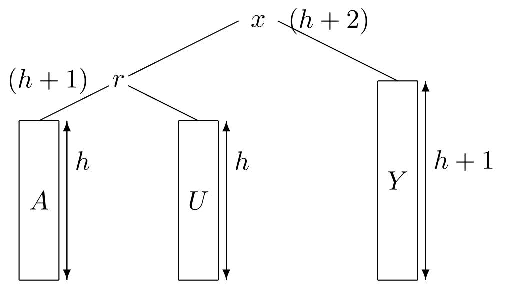
balance(X) = 1



$$A < r < U < x < Y$$

$$\text{depth}(A) = h,\ \text{depth}(U) = h,\ \text{depth}(Y) = h+1$$

Move **x** up to the root, and **r** down to the root of its left subtree.



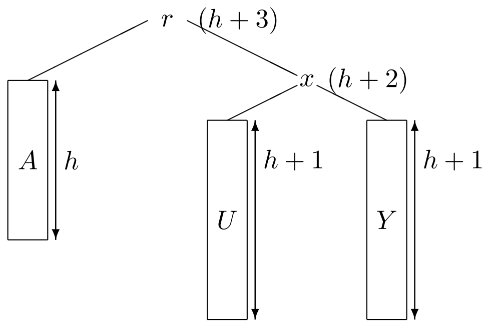
$$A < r < U < x < Y$$

$$\text{balance}(r) = \text{balance}(x) = 0$$

$$\text{depth}(\text{newtree}) = \text{depth}(\text{oldtree}) = h+2$$

Since the depth of the tree has not changed since before the insertion, no node further up the tree can be unbalanced.

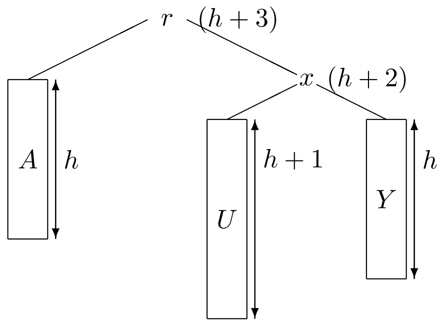
balance(X) = 0



$$\text{depth}(A) = h,\ \text{depth}(U) = \text{depth}(Y) = h+1$$

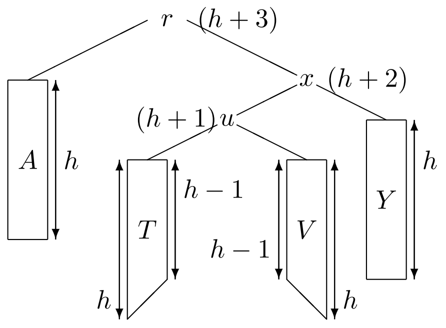
This can't happen since one of U and Y must have had depth $h+1$ before the insertion, so q would have had depth $h+2$ and the tree would already have been out of balance.

balance(X) = −1



$$\text{depth}(A) = h,\ \text{depth}(U) = h+1,\ \text{depth}(Y) = h$$

Can't simply rearrange A, U and Y to rebalance the tree — need to expand U.

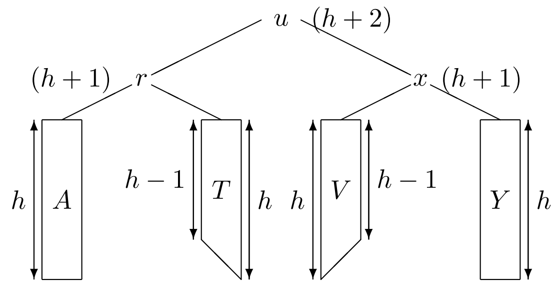


$$A < r < T < u < V < x < Y$$

- $\text{depth}(A) = \text{depth}(Y) = h$
- $\text{balance}(u) = 1 \Rightarrow \text{depth}(T, V) = (h-1, h)$
- $\text{balance}(u) = -1 \Rightarrow \text{depth}(T, V) = (h, h-1)$

Depths for T and V depend on the balance factor of u. If the balance factor of u is -1 then T has depth $h+1$ and V has depth h, while if the balance factor of u is 1 then T has depth h and V depth $h+1$. It is impossible for u to have balance factor 0.

Lift **u** to the root, with **r** as its left subtree and **x** as its right subtree.



- $\text{balance}(u) = 0$
- $\text{balance}(u)$ was $1 \Rightarrow \text{balance}(r, x) = (-1, 0)$
- $\text{balance}(u)$ was $-1 \Rightarrow \text{balance}(r, x) = (0, 1)$
- $\text{depth}(\text{newtree}) = \text{depth}(\text{oldtree})$