AL Trees

Johan Montelius

August 25, 2017

Introduction

In this assignment you're going to implement a operations on what is called an AVL tree. An AVL tree is a fairly well balanced sorted binary tree where the depth of two branches will differ by at most one. Searching in the tree is straight forward but when we add an element to the tree we might have to rearrange the nodes to preserve the AVL property. The general idea how nodes are rearranged is easy to explain if you leave out the details; the devil is of course in the details.

1 AVL trees

AVL trees are named after two Russian mathematicians, Georgy Adelson-Velsky and Evgenii Landis, who firs described how we could rearrange a binary search tree in constant time in each insert operation to keep it balanced. The idea is of course to keep the look-up operations to $O(lg(n))$ and avoid the worst case $O(n)$ scenarios that we could have if the tree becomes to imbalanced.

The trick is to keep track of the difference in depth of branches so each node will be augmented with a flag to signal that the left branch is deeper (-1), the two branches are of equal depth (0) or if the right branch is deeper (+1). When we insert a new node in the tree we could temporarily have a situation where a node is labeled with -2 or +2. This would violate the AVL property and is therefore immediately mitigated by a *rotation*. The rotation must of course preserve the ordering but still remove the violating imbalance. The different rotation operations are not tricky per see but there are many cases to keep track of so it's very easy to do mistakes.

We will first look at the two basic rotations that we will do and then dig deeper into the details of how to implement them.

1.1 single rotate

The single rotate operation is performed when the left-left branch or right-right branched has increased with one step and causes an imbalance. The left-left situation is shown in figure 1. The depth of the sub-tree A is one greater than the sub-tree of B and also two greater than the sub-tree of C. In the rearranged tree, the branches are balanced and the total depth of the tree is one less than the imbalanced tree.

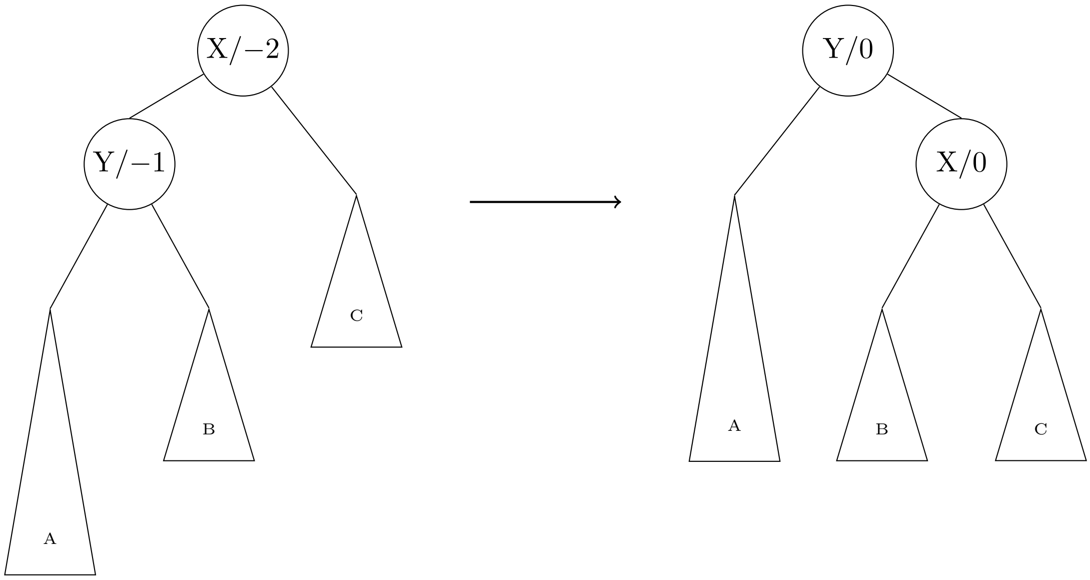

Figure 1: Single rotation: the left-left branch has grown and caused an imbalance.

The left-left single rotation has its mirror in the right-right single rotation used when the root has a difference of +2 and the right sub-tree a difference of +1.

1.2 double rotate

In the double rotate we have added an element in the left-right branch (or the right-left branch). There are three different flavors of this rule depending on what the left-right branch looks like. The first rule is shown in figure 2 and shows the situation where the left-right branch is marked with -1. This is the case where the left-right-left branch (B) has been extended. Note the difference in depth of the sub-trees, A has to be of the same depth as B and D, and C must be one level shorter.

The second version of this rule covers the situation where the left-right-left branch is marked with +1 i.e. the sub-tree C is deeper than the sub-tree B. The same transformation is done but the resulting depth differences are of course different. You're encouraged to write this rule down since you will use it in the implementation.

The third version of this rule is a special case where the left-right-left sub-tree is marked with 0. In this case, the sub-trees A, B, C and D are all of the same depth. The transformation will then generate a tree where all nodes: X, Y and Z, all have the difference 0.

Could it not be that B and C are shorter than A? How would then A compare to D? Could this situation occur? Remember, this is the rule

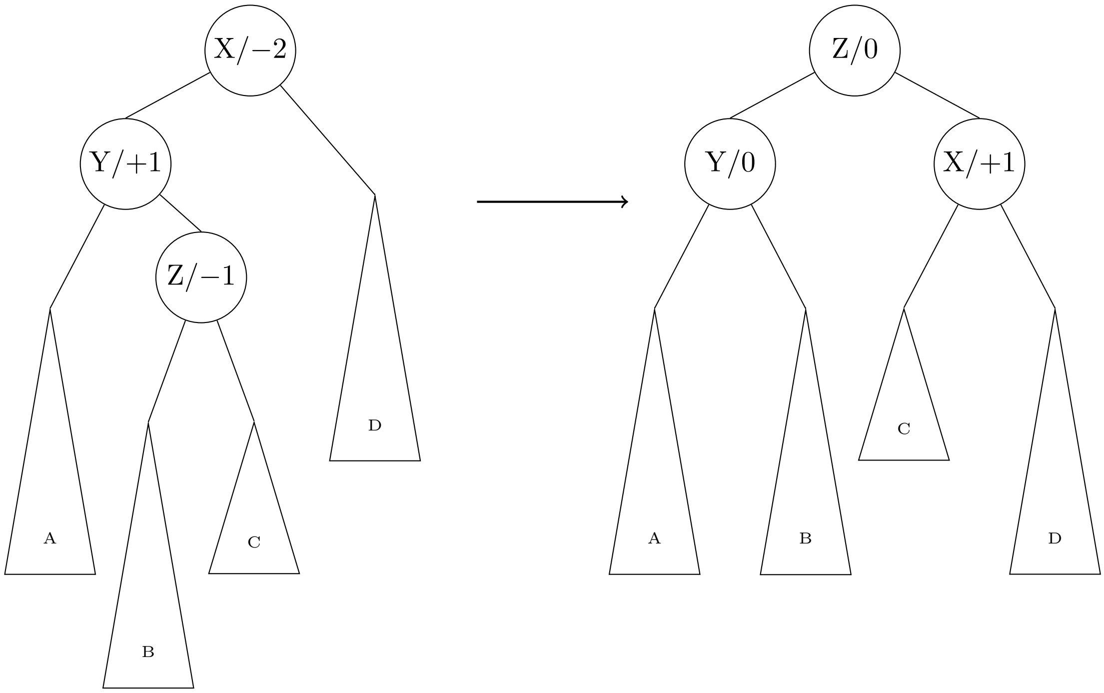

Figure 2: Double rotation: the left-right branch has grown and caused an imbalance.

applied when the left-right (or right-left) brnach is extended.

1.3 all the rotations

One thing that is important to note is that the maximum depth of the tree after the rotation is one less than before the rotation. We will use this fact in the implementation when we keep track of depth differences in the recursive operations.

All in all there are eight rotations that we will have to implement. There are two single rotations, left ad right, and these are quite straight forward. The three left-right and three right-left are trickier since we have to get the depth differences right. If you draw the transformations graphs of these rules you're more likely to get it right, if you think you can hold these rules in your head, I can only wish you good luck - you'll need it :-)

2 the implementation

Ok, let's go how hard can this be. Let's start by defining the data structure to represent a tree. We're going to use a tuple with: a key and value, a depth difference and a left and right branch. An empty tree or empty branch is represented by `nil` and there is no special representation of leaf nodes.

- {`node, Key, Value, Diff, Left, Right`}
- `nil`

2.1 insert

We will implement an insertion function that will take an AVL tree, a key and value and return an AVL tree where the key-value pair has been added or, if it existed, updated. In the recursive implementation of the insertion function we will need to keep track of when the depth of a sub-tree changes and update the depth differences accordingly. We therefor us an internal function `insrt/3` that returns either {`ok, T`} if the resulting tree is of equal depth or, {`inc, T`} if the depth has been increased by one (the depth can only increase by at most one).

```
insert(Tree, Key, Value) ->
    case insrt(Tree, Key, Value) of
        {inc, Q} ->
            Q;
        {ok, Q} ->
            Q
    end.
```

The implementation of the `insrt/3` function has two special cases and six general rules, three for each branch. The two special cases if of course if the tree is empty or if we find the key in the root of the tree. Note that we in the first case return a tuple that indicates that the tree has grown by one level.

```
insrt(nil, Key, Value) ->
    {inc, {node, Key, Value, 0, nil, nil}};

insrt({node, Key, _, F, A, B}, Key, Value) ->
    {ok, {node, Key, Value, F, A, B}};
```

In the general case we have to go down either the left branch or the right branch but now we have several alternatives depending on the depth difference of the root node and weather or not the insertion of the key-value pair increases the depth of the branch. We will first look at the simple cases and leave the two problematic cases until the end.

If the tree is balanced and we are going down the left branch either of two things can happened: the depth of the left branch is incremented by one or its depth remains the same. If it does increase we must of course return a structure that indicates that the depth of the resulting tree has increased. We also provide the correct depth difference that now is -1.

```
insrt({node, Rk, Rv, 0, A, B}, Kk, Kv)  when Kk < Rk ->
    case insrt(A, Kk, Kv) of
```

```
        {inc, Q} ->
            {inc, {node, Rk, Rv, -1, Q, B}};
        {ok, Q} ->
            {ok, {node, Rk, Rv, 0, Q, B}}
    end;
```

This rule of course has its right counter part; which branch should we go down, what should we do if the depth of that branch is increased, what is the resulting depth difference?

The second alternative is if we're going down the left branch but in a tree that has a deeper right branch. This case is almost the same but now we will not increase the total depth of the tree even if we increase the depth of the left branch. This rule also has its right counterpart and the difference is minute (but oh how important).

```
insrt({node, Rk, Rv, +1, A, B}, Kk, Kv)  when Kk < Rk ->
    case insrt(A, Kk, Kv) of
        {inc, Q} ->
            {ok, {node, Rk, Rv, 0, Q, B}};
        {ok, Q} ->
            {ok, {node, Rk, Rv, +1, Q, B}}
    end;
```

The tricky case comes when we're going down the left or right branch and this is already a branch that is longer than its sibling. In this case we could end up in a situation were have a depth difference of 2. This is were we rely on our rotator to fix things for us. Note that the rotation will result in a tree that does not change the maximum depth of the original tree. After a rotation we can safely return the result as it is without any warning that the depth has increased.

```
insrt({node, Rk, Rv, -1, A, B}, Kk, Kv)  when Kk < Rk ->
    case insrt(A, Kk, Kv) of
        {inc, Q} ->
            {ok, rotate({node, Rk, Rv, -2, Q, B})};
        {ok, Q} ->
            {ok, {node, Rk, Rv, -1, Q, B}}
    end;
```

If you also implement the final right branch version of this last rule you have completed all the rules needed. We of course have the problem of describing the rotations but this is more of a pattern matching exercise if you have drawn all the graphs.

2.2 rotation

Remember the why we do a rotation and the two different versions; we do the rotation to correct the imbalance and we do this using either a single or double rotation. The single rotation can be used if we the tree is too deep in its left-left or right-right branch. The double rotation is more complicated and deals with the cases were the left-right or right-left branch is too deep.

2.2.1 single rotation

Let's start with the simple cases were we can use a simple rotation. If the root is annotated with -2 and the left branch is annotated with -1 we can do a rotation of the left branch. If its the opposite we do a rotation of the right branch.

```
rotate({node, Xk, Xv, -2, {node, Yk, Yv, -1, A, B}, C}) ->
    {node, Yk, Yv, 0, A, {node, Xk, Xv, 0, B, C}};

rotate({node, Xk, Xv, +2, A, {node, Yk, Yv, +1, B, C}}) ->
    {node, Yk, Yv, 0, {node, Xk, Xv, 0, A, B}, C};
```

Note how the first rule is a direct mapping if the graph in figure 1. If we know that the drawn rule is correct there is little room for doing any mistakes i.e. if the logic is right the coding is trivial.

2.2.2 double rotations

The double rotations are of course more complicated but the complication is more in getting the logic right and not in the implementations. We can first look at the rule that is described in figure 2. We know what the tree should look like and we know what the result should be.

```
rotate({node, Xk, Xv, -2, {node, Yk, Yv, +1, A, {node, Zk, Zv, -1, B, C}}, D}) ->
    {node, Zk, Zv, 0, {node, Yk, Yv, 0, A, B}, {node, Xk, Xv, +1, C, D}};
```

The second rule covers the case where the Z node has a right branch that is deeper than its left branch. If you have drawn the trees the implementation of the rule should be straight forward.

```
rotate({node, Xk, Xv, -2, {node, Yk, Yv, +1, A, {node, Zk, Zv, +1, B, C}}, D}) ->
    {node, Zk, Zv, 0, {node, Yk, Yv, -1, A, B}, {node, Xk, Xv, 0, C, D}};
```

The last of the left-right rules is the one that handles the special case where the two branches of the Z node are of equal length. Remember that this was a very special case and if we can detect this earlier we might do without this rule all together.

```
rotate({node, Xk, Xv, -2, {node, Yk, Yv, +1, A, {node, Zk, Zv, 0, B, C}}, D}) ->
    {node, Zk, Zv, 0, {node, Yk, Yv, 0, A, B}, {node, Xk, Xv, 0, C, D}};
```

The three left-right rules of course have their right-left counterparts but these are all left for the reader as an exercise. One advice is to draw the rules first and only then implement them. It's very easy to mix up a -1 for a +1 or swap a left branch for a right branch. If you draw the rules first it should not be a problem.

3 benchmarks

The whole idea with AVL trees is that we should gain some performance so let's do some benchmarks to see if all the trouble was worth the effort.

After all, if the performance gain is not enough it might be better to have less code to make it easier to verify that we have done the right thing.

In order to evaluate our implementation we could generate a tree from a sequence of keys and then see what the tree looks like and compare it to a tree generated by the regular binary sorted tree algorithms. What is interesting is if the AVL tree is more balanced i.e. that the average depth at which we will find our keys is small. A perfectly balanced tree will have a depth of $lg(n)$ and it's interesting to see how close to this that the AVL tree is.

Let's create two modules, `avl` and `bst`, that implements the following functions:

- `tree()` : returns an empty tree.
- `insert(Tree, Key, Value)` : returns a tree where the key value has been inserted (or updated if key existed).
- `depth(Tree, Key)` : returns {`ok, Depth`}, with the depth at which the key is found, or `fail` if the key is not found.
- `max_depth(Tree)` : returns the maximum depth of the tree.

We will use the `depth/2` function to both gather statistics on the depth at which keys are found and also measure the time it tales to find a key. The depth function will of course do exactly the same steps as a `lookup/2` function would do.

Let's now in a module called `test` implement some benchmarks that compares the `avl` and `bst` module. The first thing wee need is a function that generates a sequence of keys in random order. We will use this sequence to first build a tree and then examine how long time it takes to access the keys. Define a function `sequence(N,M)` that generates a list of N integers from 1 to M (we're using the library module `random`).

```
sequence(0, M) ->
    [];
sequence(I) ->
    [rand:uniform(M)|sequence(I-1, M)].
```

Now define a function

4 Summary

Pattern matching is a powerful technique to describe the rules of a transformation. Together with recursion even a complex algorithm becomes quite easy to implement.