A 2-3 Tree Insertion

Programming II - Elixir Version

Johan Montelius

Spring Term 2018

Introduction

You hopefully know about a tree structure called 2-3 trees. This is an ordered tree where nodes either have two or three branches. The beauty is that it is perfectly balanced and this balance is maintained with not to much overhead.

All operations (search, insert, delete) are $O(lg(n))$ and there are no worst case scenarios since the insert and delete operations will preserve the balance. We're not relying on luck as we are if we use a simple binary tree.

In this assignment you're going to learn how to implement a quite tricky algorithm and you will do so using pattern matching. If you don't remember how 2-3 trees work refresh your memory before you start.

1 The 2-3 tree

In this implementation we will create a 2-3 tree that keeps key-value pairs. The key-values will only reside in the leaves of the tree but the keys will of course also be present in the nodes to guide a search.

A tree is either: empty, a leaf, a two-node or a *three-node*. The tree is balanced so all leaves are on the same level in the tree. A simple representation of these different trees could be:

- *the empty tree*: `nil`
- *a leaf*: `{:leaf, key, value}`
- *a two-node*: `{:two, key, left, right}`
- *a three-node*: `{:three, k1, k2, left, middle, right}`

All keys in the left branch of a two-node are smaller or equal to the key of the node. In a three-node the keys of the left branch is smaller or equal to the first key and the keys of the middle branch are less than or equal to the second key.

We will in our implementation also work with a third type of node, a node with four branches. The tree that we work with will never contain a *four-node* but we will allow the insertion function to return this structure.

- *a four-node*: `{:four, k1, k2, k3, left, m1, m2, right}`

Doing search in this tree is of course trivial but doing insert or delete requires some more thinking.

2 The rules of insertion

Remember the rules of the insert operation; we have some simple cases in the bottom of the tree and then some tricky cases where we end up with a four-node. Let's start with a description of the simple cases:

- *empty tree*: To insert a key-value pair in an empty tree return a leaf containing the key and value.
- *a leaf*: To insert a key-value pair in a leaf, return a two-node containing the existing leaf and a new leaf. The smaller of the keys should be the key of the two-node.

So far, so good - now to the case were we have a two-node or a three-node. We should do quite different things depending on if the node holds leaves or if it is an internal node that holds two-nodes or three-nodes. Note that a two-node or three-node either holds only leaves or only two- and three-nodes, it can not hold only one leaf node since all leaves are on the same level in the tree.

So let's see what the rules are if we encounter a two- or three-node that holds only leaves.

- *two-node holding leaves*: Create a new leaf and return a three-node containing all three leaves. Make sure that the leaves are ordered and that the three-node holds the two smallest keys.
- *three-node holding leaves*: Create a new leaf and return a four-node containing all four leaves. Make sure that the leaves are ordered and that the four-node holds the three smallest keys.

Now, this was simple but we have of course possibly returned a four-node. We will see how this will be removed when we do the recursive step.

When we encounter an internal two-node or three-node we will proceed and insert the new key-value pair in the appropriate branch. If we do this we could end up with a four-node and then we have to think twice before we return an new tree.

- *two-node general case*: If the insert operation on a branch returns a four-node, construct a valid three-node, where the middle key of the returned four-node is the new key in the three-node. Otherwise, return an updated two-node.

- *three-node general case*: If the insert operation on a branch returns a four-node, construct a valid four-node, where the middle key of the returned four-node is the new key in the four-node. Otherwise, return an updated three-node.

The only thing that is left is the rule to handle the root of the tree. Everything works fine with the rules that we have but it could of course be that we will return a four-node, and we don't want to have four-nodes in our tree.

- *the root*: If, by updating the root, we return a four-node, then split the four-node into two two-nodes that are branches of a new two-node root.

That is it, now let's see how hard this is to implement.

3 The implementation

Let's implement a function `insertf(key, value, tree)` that takes a key, a value and a 2-3 tree and returns a 2-3 tree or possibly a four-node containing 2-3 trees of equal depth (we will deal with the root case last). Look at the rules and start with the base cases.

```
def insertf(key, value, nil), do: {:leaf, key, value}

def insertf(k, v, {:leaf, k1, _} = l) do
  cond do
    k <= k1 ->
      {:two, k, {:leaf, k, v}, l}
    true ->
      {:two, k1, l, {:leaf, k, v}}
  end
end
```

You might not have seen the construct `= l` in the head but it is a very convenient syntax. We do want the second argument to match the pattern but at the same time we would like to use the data structure. We could have written this without using this syntax but it would be more to write and it would not be as efficient.

So the two first rules was easy, now for the rules were we are looking at a two-node that holds only leaves. I leave this with some holes for you to fill in; you should do more than just copy and paste. Look at the rules, you should return a three-node that holds two keys and three leaves.

```
def insertf(k, v, {:two, k1, {:leaf, k1, _} = l1,
                             {:leaf, k2, _} = l2}) do
  cond do
    k <= k1 ->
      ...
    k <= k2 ->
      ...
    true ->
      ...
  end
end
```

The keys should of course be `k1`, `k2` and `k`, but in the right order and the corresponding leaves should follow. When you think you have it, take it for a spin, terminate the clause with a dot, compile and do some experiments. You should be able to handle an empty tree, a tree of only a leaf and a two-node with two leaves. If you got it, proceed to the three-node.

```
def insertf(k, v, {:three, k1, k2, {:leaf, k1, _} = l1,
                                   {:leaf, k2, _} = l2,
                                   {:leaf, k3, _} = l3}) do
  cond do
    k <= k1 ->
      ...
    k <= k2 ->
      ...
    k <= k3 ->
      ...
    true ->
      ...
  end
end
```

What is the result of inserting a key-value pair in a three-node holding three leaves? Look at the rule, complete the code, compile and test. Ok - then the recursive cases.

```
def insertf(k, v, {:two, k1, left, right}) do
  cond do
    k <= k1 ->
```

```
      case insertf(k, v, left) do
        {:four, q1, q2, q3, t1, t2, t3, t4} ->
          ...
        updated ->
          {:two, k1, updated, right}
      end

    true ->
      case insertf(k, v, right) do
        {:four, q1, q2, q3, t1, t2, t3, t4} ->
          ...
        updated ->
          {:two, k1, left, updated}
      end
  end
end
```

Look at the first case, we have tried to insert the new key-value in the left branch but ended up with a four-node. The four-node should be divided into two two-nodes, one holding `q1` and one holding `q3`. The key `q2` should be used to create a new three-node, holding the two newly created two-nodes and the original right node. The three-node should thus have the keys: `q2` and `k1` but you need to put this together in the right order.

If you manage to solve the first puzzle then the second case is easy. Again, compile and test - can you handle the test below?

```
def test do
  insertf(14, :grk, {:two, 7, {:three, 2, 5, {:leaf, 2, :foo},
      {:leaf, 5, :bar}, {:leaf, 7, :zot}}, {:three, 13, 16,
      {:leaf, 13, :foo}, {:leaf, 16, :bar}, {:leaf, 18, :zot}}})
end
```

The case where we have a three-node is of course very similar, it's a lot of code but it should not be to difficult. You will probably make mistakes but these are more mistakes of typos. It could be easier to write the tree on a paper that clearly shows what `q1`, `q2` etc are referring to.

```
def insertf(k, v, {:three, k1, k2, left, middle, right}) do
  cond do
    k <= k1 ->
      case insertf(k, v, left) do
        {:four, q1, q2, q3, t1, t2, t3, t4} ->
          {:four, q2, k1, k2, ..., ..., ..., ...}
        updated ->
          {:three, k1, k2, ..., ..., ...}
      end
    k <= k2 ->
      case insertf(k, v, middle) do
        {:four, q1, q2, q3, t1, t2, t3, t4} ->
          ...
        updated ->
          {:three, k1, k2, ..., ..., ...}
      end
    true ->
      case insertf(k, v, right) do
        {:four, q1, q2, q3, t1, t2, t3, t4} ->
          ...
        updated ->
          {:three, k1, k2, ..., ..., ...}
      end
  end
end
```

Last but not least, we must handle the root case. Take another look at the final part of the rules for insertion. When updating the root results in a four-node, we must split it up into two-nodes, increasing the height of the tree by one in the process. We define a new function for this, that will also act as the interface to this module.

```
def insert(key, value, root) do
  case insertf(key, value, root) do
    {:four, q1, q2, q3, t1, t2, t3, t4} ->
      ...
    updated ->
      ...
  end
end
```

You should now be able to create a 2-3 tree of arbitrary size by repeatedly calling the new function.