CS 240 — Lecture 9

Bartosz Antczak *Instructor: Eric Schost* *January 31, 2017*

9.1 Dictionaries

A **dictionary** is a collection of items, each which contains

- a *key*
- some *data* to which the key is mapped to

This grouping of keys and data is called a *key-value pair* (KVP). Keys can be compared and are usually unique. Some basic operations we'll use on dictionaries are:

- `search(k)`
- `insert(k, v)`
- `delete(k)`

9.2 Binary Search Trees (Review)

A **binary search tree** (BST) is either empty or contains

- a KVP
- a left child BST
- a right child BST

The tree is ordered such that every key in the left child is less than the root key and every key in the right child is greater than the root key.

BST Search and Insert

This method takes in one argument `k` and compares `k` to the current node (starting at the root), and stops if it's found; otherwise it recurses on both subtrees until it's empty. `insert(k, v)` uses `search(k)`, then inserts the key as a new node.

BST Delete

This algorithm is slightly harder to implement:

- If the node we want to delete is a leaf, just delete it
- If node has one child, move the child up
- Otherwise, swap with the *successor* or *predecessor* (i.e., the node with the next largest or next smallest key value) node then delete

9.2.1 Height of a BST

All of the aforementioned operations on a tree have cost $\Theta(h)$, where h = height of a tree = maximum path length from root to leaf. If n items are inserted one at a time, how big is h?

- Worst-case: $n - 1 = \Theta(n)$
- Best-case: $\lfloor \log(n) \rfloor = \Theta(\log n)$
- Average-case: $\Theta(\log n)$

9.3 AVL Trees

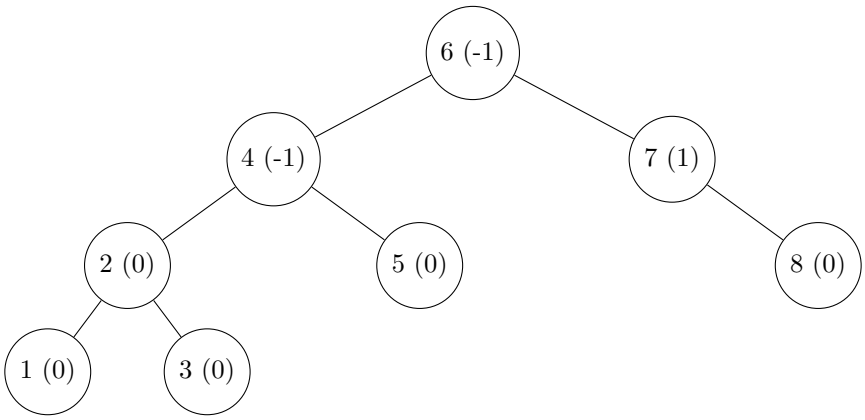
An **AVL tree** is a BST with an additional constraint:

the heights of the left and right subtree differ by at most 1

At each non-empty node, we store `height(Right subtree) - height(Left subtree)` $\in \{-1, 0, 1\}$, where

- -1 means the tree is *left-heavy*
- 0 means the tree is *balanced*
- 1 means the tree is *right-heavy*

Example 9.3.1. *A typical AVL (the balance factor is stored in brackets)*



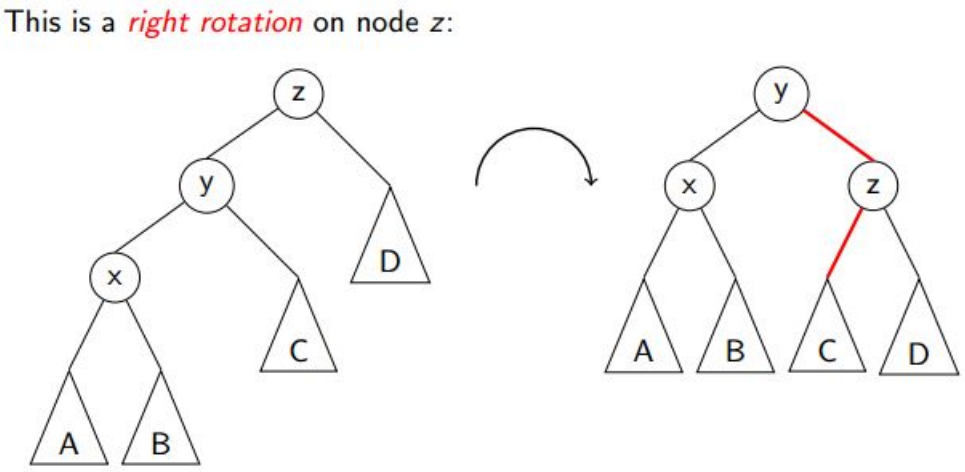
9.3.1 AVL insertion

To perform `insert(T, k, v)`:

- Insert (`k, v`) in `T` using usual BST insertion
- Then from the newly inserted leaf, move up the tree, updating balance factors
- If the balance factor is -1, 0, or 1, then keep going
- If the balance factor is ± 2, then we call the `fix` algorithm to rebalance that node (will be outlined later)

Right-rotation

To fix an unbalanced AVL tree, we want to change the structure of the tree without changing the order (i.e., ensuring that all left children have a smaller key value and all right children have a larger key value). We do this through a special *rotation*:



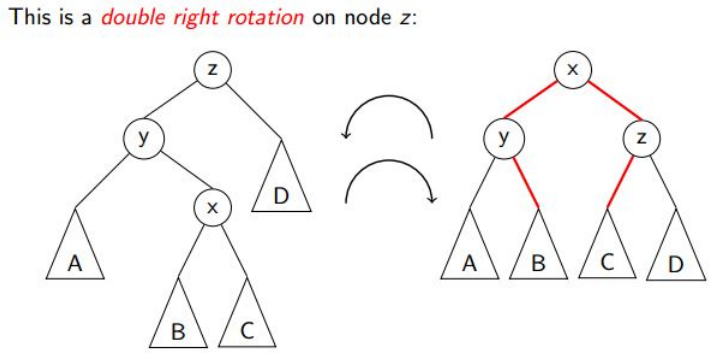
The left rotation is the mirror-image of the right rotation. In either case, only two edges need to be moved and two balances updated. The pseudocode for the right rotation is:

```
rotate-right(T)
// T: AVL tree
// This method returns rotated AVL tree
  newroot = T.left
  T.left = newroot.right
  newroot.right = T
  return newroot
```

Again, the left-rotation pseudocode is nearly identical. Observe that the runtime of this algorithm is $O(1)$.

Double Right Rotation

This function performs two rotations: first, a left rotation on the left subtree. Second, a right rotation on the whole tree:



Similarly, there is also a **double left rotation**, which is defined as a *right rotation* on the right subtree, followed by a left rotation on the whole tree. The algorithm that will include all of these rotations will be called `fix`:

```
// T: AVL tree with a balance +- 2
fix(T)
 if (T.balance = -2)
  if (T.left.balance = 1)
    T.left = rotate-left(T.left)
  return rotate-right(T)
 else if (T.balance = 2)
  if (T.right.balance = -1)
    T.right = rotate-right(T.right)
  return rotate-left(T)
```

9.3.2 AVL Tree Operations

Just like the BST, we'll use `search`, `insert`, and `delete`:

- `search`: same as the BST, costs $\Theta(h)$
- `insert`: already dicussed in "9.3.1: AVL Insertion". Since `fix` is called at most once, the total cost is $\Theta(h)$
- `delete`: first search, then swap with successor (as with BSTs), then move up the tree and apply `fix`. Here, `fix` may be called $\Theta(h)$ times