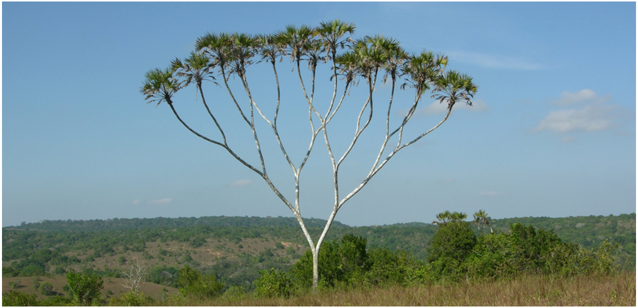
CSCB63 – Design and Analysis of Data Structures

Anya Tafliovich[1]

[1] based on notes by Anna Bretscher and Albert Lai

our dear friend the BST (Binary Search Tree)



Q. What did we use it for?
A.
Q. Can things go wrong?
A.

BST — when things go wrong

Example: show a sequence of values that, when inserted into an initially empty BST, creates a "bad" BST.

Q. What is the complexity of `search` in this tree?
A.

solution — balanced trees

Idea: Maintain a Binary Search Tree that always stays balanced.

Several ways to accomplish more-or-less the same result:

- Red-Black trees
- AVL trees — G. Abelson-Velsky and E. Landis
- B-trees
- Splay trees

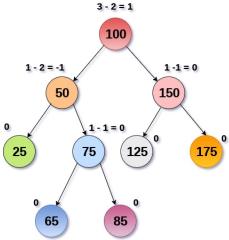
AVL tree

- stores key/value pairs in all nodes (both leaf and internal)

- has a property relating the keys stored in a subtree to the key stored in the parent node (ordering)

- maintains the height (number of nodes/edges on a root-to-leaf path) of $\mathcal{O}(\log n)$
  - balance factor $=$ height(left subtree) $-$ height(right subtree)
  - maintain balance factor of ± 1 or 0 for all nodes

AVL tree



AVL Tree

- maintains the height (number of nodes/edges on a root-to-leaf path) of $\mathcal{O}(\log n)$
  - balance factor = height(left subtree) − height(right subtree)
  - maintain balance factor of ± 1 or 0 for all nodes

AVL tree operations

Operations of an ordered dictionary:

- `search(k, T)`: return the value corresponding to key k in the tree T
    - special scenario if $k \notin T$
- `insert(k, v, T)`: insert the new key/value pair k/v into the tree T
    - special scenario if $k \in T$
- `delete(k, T)`: delete the key/value pair with key k from the tree T
    - special scenario if $k \notin T$

AVL search

Q. How should we implement `search(k, T)`?

A.

AVL insert

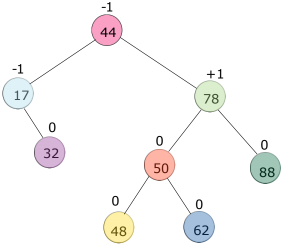
Q. How should we implement `insert(k, v, T)`?

Q. Can we take the same approach as with `search`?
A.

For example:

AVL insert

Insert scenario:



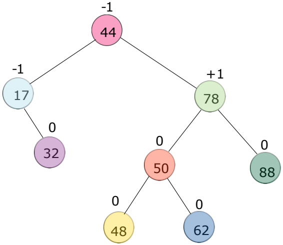
Insert key 6.

Q. What does the new tree look like?
Q. What are the new balance factors?

A.

AVL insert

Another insert scenario:



Insert key 35.

Q. What does the new tree look like? What are the new balance factors?

A.

Q. What is the problem?

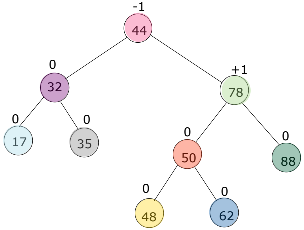
A.

AVL insert

Solve the problem:

AVL insert

Another insert scenario:



Insert key 45.

Q. What does the new tree look like? What are the new balance factors?

A.

Q. What is the problem?

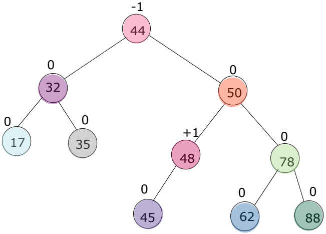
A.

AVL insert

Solve the problem:

AVL insert

Another insert scenario:



Insert key 46.

Q. What is the problem? Can we fix it by one of the methods above?

A.

AVL insert

Solve the problem:

Q. How do we know that we need a double rotation?

A.

AVL insert

Important observations before we develop the complete algorithm:

- do we need to perform only 1 rotation per insert?

- how can we be sure this won't end up being $\mathcal{O}(n)$?
  - Intuitively:

  - Formally:

AVL rebalancing

For each node v on the new-node-to-root path:

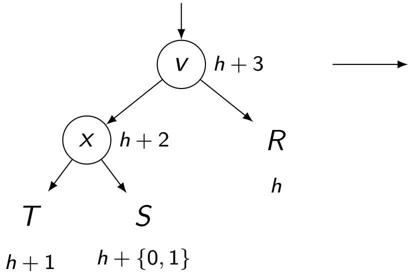
```
if height(v.left) - height(v.right) > 1:
    let x = v.left
    if height(x.left) >= height(x.right):
        single rotation: clockwise
    else:
        double rotation: counter-clockwise then clockwise

else if height(v.right) - height(v.left) > 1:
    let x = v.right
    if height(x.left) <= height(x.right):
        single rotation: counter-clockwise
    else:
        double rotation: clockwise then counter-clockwise

else:
    no rotation
```

AVL rebalancing: rotation clockwise

```
if height(v.left) - height(v.right) > 1:
    let x = v.left
    if height(x.left) >= height(x.right):
        single rotation: clockwise
```

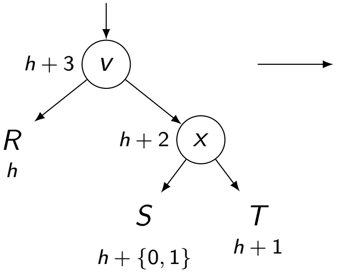


Q. How do we know `x` exists?

Q. How do we know the result is a BST?

AVL rebalancing: rotation counter-clockwise

```
if height(v.right) - height(v.left) > 1:
    let x = v.right
    if height(x.left) <= height(x.right):
        single rotation: counter-clockwise
```

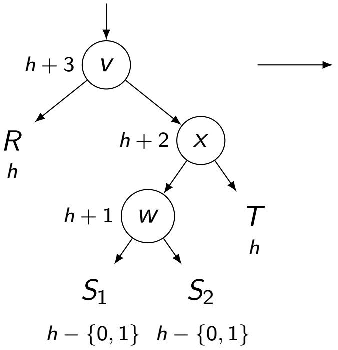


Q. How do we know `x` exists?

Q. How do we know the result is a BST?

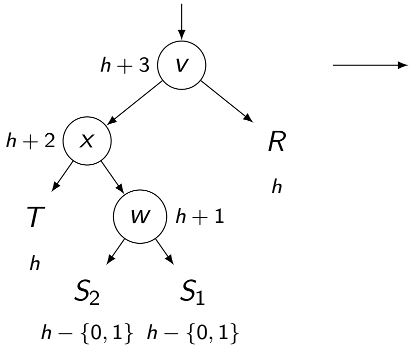
AVL rebalancing: double rotation

```
if height(v.right) - height(v.left) > 1:
    let x = v.right
    ...
    // height(x.left) > height(x.right):
        let w = x.left
        double rotation: clockwise then counter-clockwise
```



AVL rebalancing: double rotation

```
if height(v.left) - height(v.right) > 1:
    let x = v.left
    ...
    // height(x.left) < height(x.right):
        let w = x.right
        double rotation: counter-clockwise then clockwise
```



AVL insert

- find the node to become parent of new node
  - complexity:
- put new node there
  - complexity:
- rebalance and update heights if needed
  - complexity:

This means we need to store, for each node, either the height of the subtree rooted at it or its balancing factor.

We will prove that the height of the AVL tree is $\mathcal{O}(\log n)$ shortly.

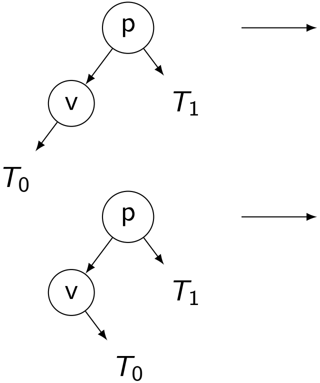
AVL delete

1. delete the node using the algorithm for BST delete

2. rebalance and update as needed

AVL delete (easier case)

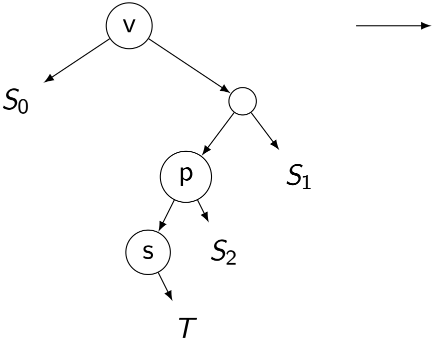
If node v has one child:



- v's parent adopts v's child
- go from p up to root, rebalancing on the way

AVL delete (harder case)

If node v has two children, successor node s:



- s's parent adopts s's child (if it exists)
- s's key/value moves to v
- go from p up to root, rebalancing on the way

AVL delete

1. find the node to delete; call it v
   - complexity:
2. if v has no children, delete v, update height of v's parent
   - complexity:
3. if v has one child, v's parent adopts v's child, delete v, update height of v's parent
   - complexity:
4. if v has two children
   4.1 find the successor s of v (complexity:
   4.2 move the key/value pair of s into v (complexity:
   4.3 delete s, s's parent adopts s's (right) child if it exists, update height of s's parent (complexity:
5. starting from the parent of deleted node, go up to root, updating heights and rebalancing as necessary
   - complexity:

AVL tree height

These two questions are equivalent:

- in a tree with n nodes, what is the maximum possible height h?
- if the tree height is h, what is the minimum possible number of nodes n?

Let $minsize(h)$ denote the minimum size (number of nodes) of a tree of height (number of nodes of the longest root-to-leaf path) h. Then:

$$
\begin{aligned}
minsize(0) &= 0 \\
minsize(1) &= 1 \\
minsize(h+2) &=
\end{aligned}
$$

Does this look familiar?

AVL tree height

Exercise: prove by induction that

$$minsize(h) = fib(h + 2) - 1$$

Now recall the "golden ratio" and how it relates to Fibonacci numbers:

$$\begin{aligned} \phi &= (1 + \sqrt{5})/2 \\ \psi &= (1 - \sqrt{5})/2 \\ fib(n) &= (\phi^n - \psi^n)/\sqrt{5} \end{aligned}$$

We therefore have

$$minsize(h) = \frac{\phi^{h+2} - \psi^{h+2}}{\sqrt{5}} - 1$$

AVL tree height

$n = minsize(h) = \frac{\phi^{h+2} - \psi^{h+2}}{\sqrt{5}} - 1 =$

Thus we have height of an AVL tree with n nodes is $\in \mathcal{O}(\log n)$.