Module 4: Dictionaries and Balanced Search Trees

CS 240 - Data Structures and Data Management

Sajed Haque   Veronika Irvine   Taylor Smith
Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2017

Dictionary ADT

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

- a *key*
- some *data*,

and is called a *key-value pair* (KVP). Keys can be compared and are (typically) unique.

Operations:

- *search*(k)
- *insert*(k, v)
- *delete*(k)
- optional: *join*, *isEmpty*, *size*, *etc.*

Elementary Implementations

Common assumptions:

- Dictionary has n KVPs
- Each KVP uses constant space
- Comparing keys takes constant time

Unordered array or linked list

search $\Theta(n)$
insert $\Theta(1)$
delete $\Theta(n)$ (need to search)

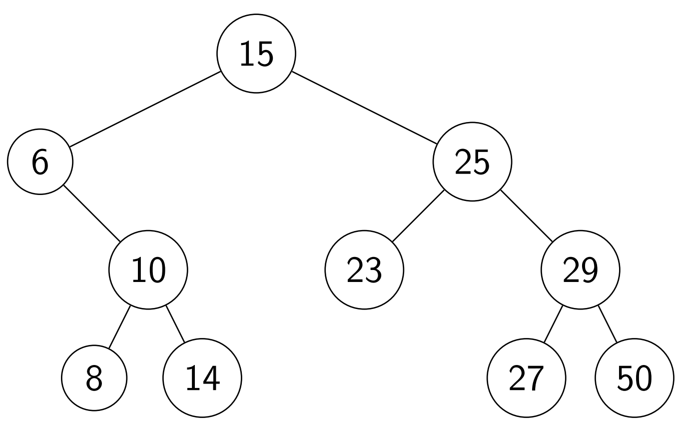
Ordered array

search $\Theta(\log n)$
insert $\Theta(n)$
delete $\Theta(n)$

Binary Search Trees (review)

Structure: A BST is either empty or contains a KVP, left child BST, and right child BST.

Ordering: Every key k in $T.left$ is less than the root key. Every key k in $T.right$ is greater than the root key.



BST Search and Insert

search(k) Compare k to current node, stop if found, else recurse on subtree unless it's empty

insert(k, v) Search for k, then insert (k, v) as new node

Example:

BST Delete

- If node is a leaf, just delete it.
- If node has one child, move child up
- Else, swap with *successor* or *predecessor* node and then delete

Height of a BST

search, *insert*, *delete* all have cost $\Theta(h)$, where
$h =$ height of the tree $=$ max. path length from root to leaf

If n items are *insert*ed one-at-a-time, how big is h?

- Worst-case:
- Best-case:
- Average-case:

AVL Trees

Introduced by Adel'son-Vel'skiĭ and Landis in 1962, an *AVL Tree* is a BST with an additional structural property:
The heights of the left and right subtree differ by at most 1.

(The height of an empty tree is defined to be -1.)

At each non-empty node, we store $height(R) - height(L) \in \{-1, 0, 1\}$:

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

- We could store the actual height, but storing balances is simpler and more convenient.

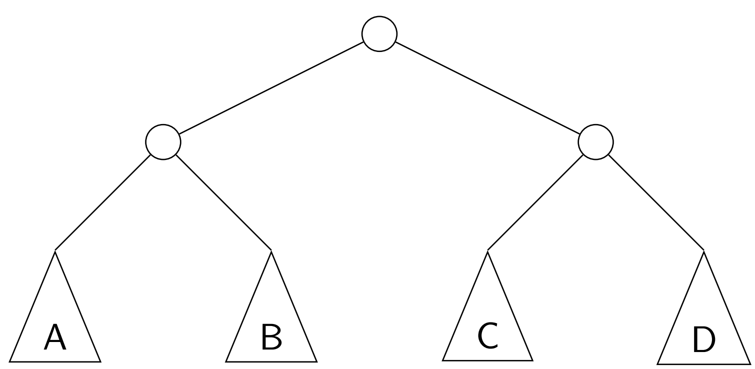
AVL insertion

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

- First, insert (k, v) into T using usual BST insertion
- Then, move up the tree from the new leaf, updating balance factors.
- If the balance factor is -1, 0, or 1, then keep going.
- If the balance factor is ± 2, then call the *fix* algorithm to "rebalance" at that node. We are done.

How to "fix" an unbalanced AVL tree

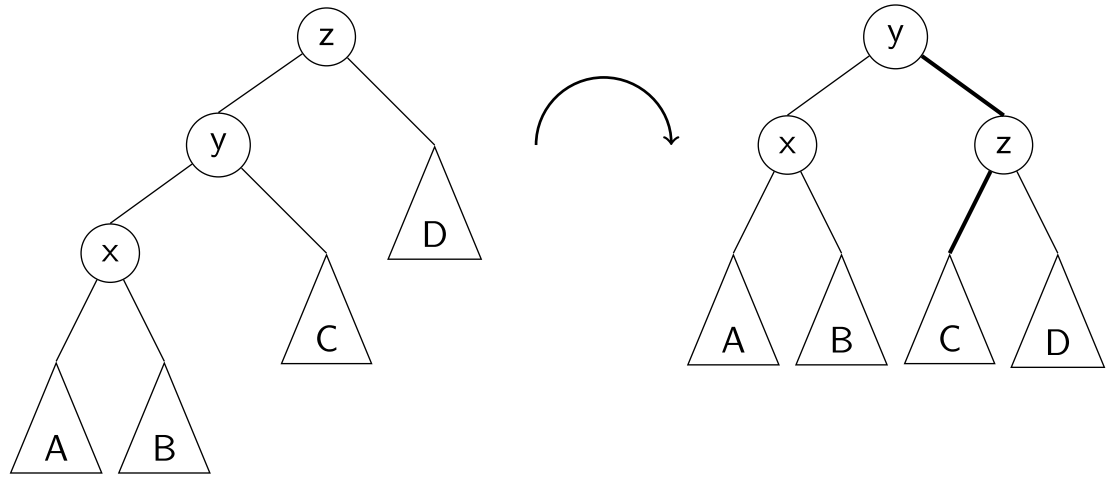
Goal: change the *structure* without changing the *order*



Notice that if heights of A, B, C, D differ by at most 1, then the tree is a proper AVL tree.

Right Rotation

This is a *right rotation* on node z:

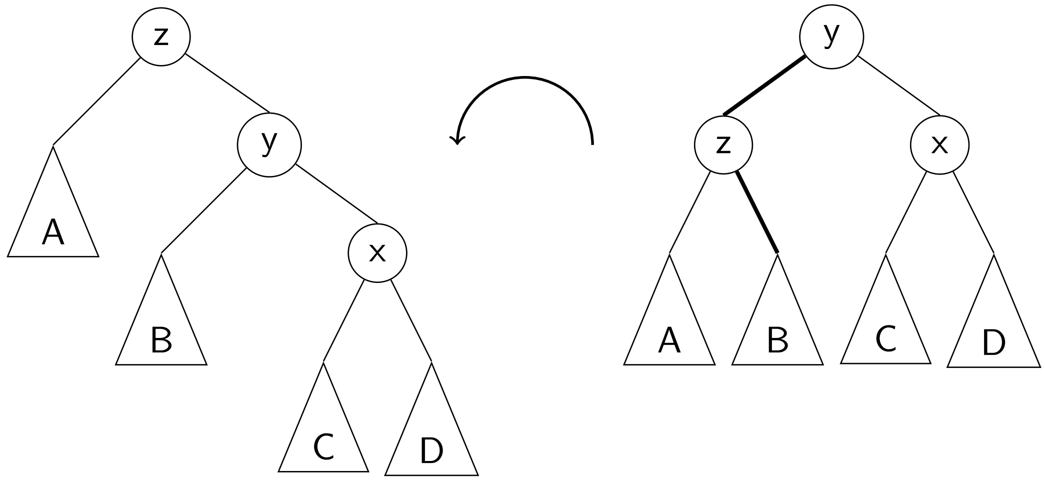


Note: Only two edges need to be moved, and two balances updated. Useful to fix left-left imbalance.

Again ...

Left Rotation

This is a *left rotation* on node *z*:



Again, only two edges need to be moved and two balances updated.
Useful to fix right-right imbalance.

Again ...

Pseudocode for rotations

rotate-right(T)
T: AVL tree
returns rotated AVL tree

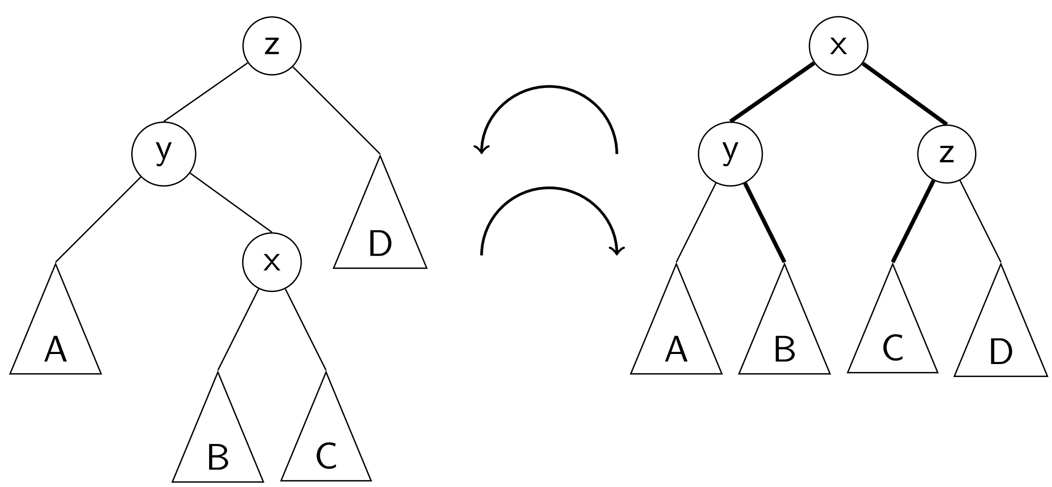
```
newroot ← T.left
T.left ← newroot.right
newroot.right ← T
return newroot
```

rotate-left(T)
T: AVL tree
returns rotated AVL tree

```
newroot ← T.right
T.right ← newroot.left
newroot.left ← T
return newroot
```


Double Right Rotation

This is a *double right rotation* on node z:

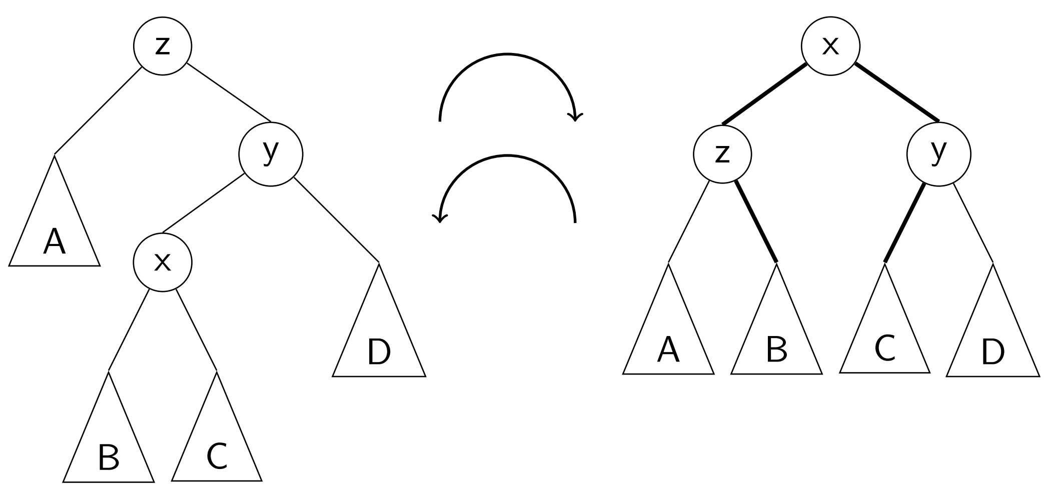


First, a left rotation on the left subtree (y). Second, a right rotation on the whole tree (z).
Useful for left-right imbalance.

Again . . .

Double Left Rotation

This is a *double left rotation* on node z:



Right rotation on right subtree (y), followed by left rotation on the whole tree (z).
Useful for right-left imbalance.

Fixing a slightly-unbalanced AVL tree

Idea: Identify one of the previous 4 situations, apply rotations

fix(T)
T: AVL tree with $T.balance = \pm 2$
returns a balanced AVL tree

```
if T.balance = −2 then
      if T.left.balance = 1 then
            T.left ← rotate-left(T.left)
      return rotate-right(T)
else if T.balance = 2 then
      if T.right.balance = −1 then
            T.right ← rotate-right(T.right)
      return rotate-left(T)
```


AVL Tree Operations

search: Just like in BSTs, costs $\Theta(height)$

insert: Shown already, total cost $\Theta(height)$

- *fix* restores the height of the tree it fixes to what it was,
- so *fix* will be called *at most once*.

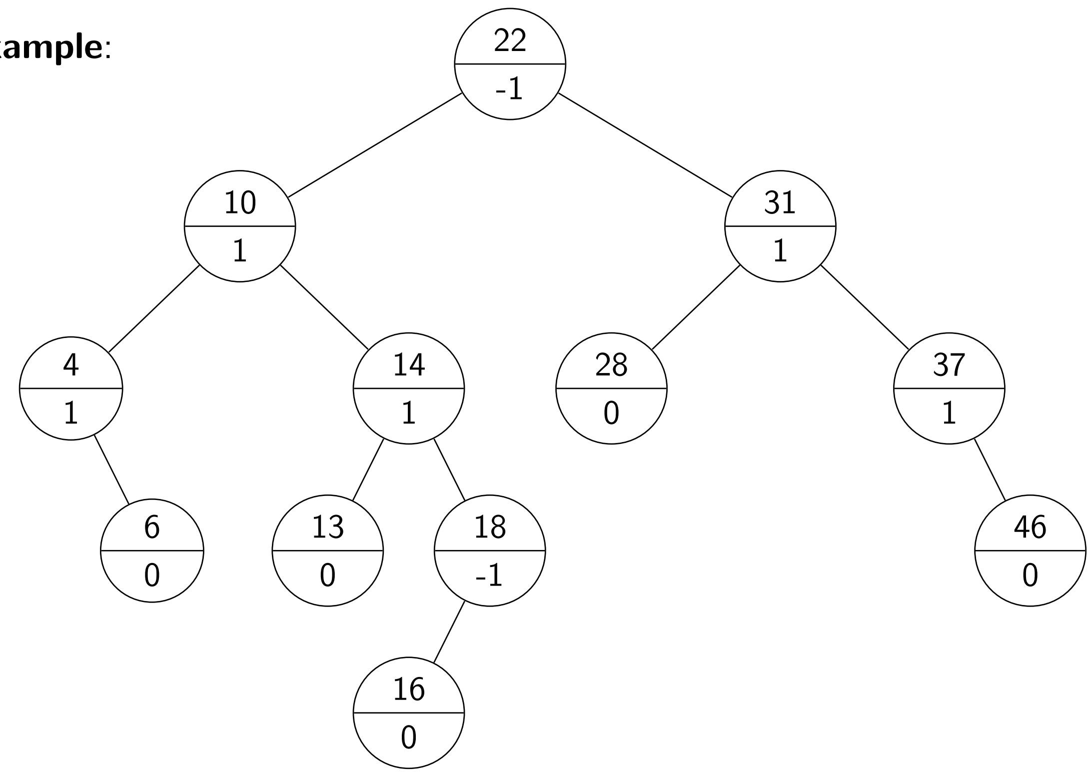
delete: First search, then swap with successor (as with BSTs), then move up the tree and apply *fix* (as with *insert*).

- *fix* may be called $\Theta(height)$ times.

Total cost is $\Theta(height)$.

AVL tree examples

Example:



Height of an AVL tree

Define $N(h)$ to be the *least* number of nodes in a height-h AVL tree.

One subtree must have height at least $h-1$, the other at least $h-2$:

$$N(h) = \begin{cases} 1 + N(h-1) + N(h-2), & h \geq 1 \\ 1, & h = 0 \\ 0, & h = -1 \end{cases}$$

What sequence does this look like?

AVL Tree Analysis

Easier lower bound on $N(h)$:

$$N(h) > 2N(h-2) > 4N(h-4) > 8N(h-6) > \cdots > 2^i N(h-2i) \geq 2^{\lfloor h/2 \rfloor}$$

Since $n > 2^{\lfloor h/2 \rfloor}$, $h \leq 2 \lg n$,
and thus an AVL tree with n nodes has height $O(\log n)$.
Also, $n \leq 2^{h+1} - 1$, so the height is $\Theta(\log n)$.

$\Rightarrow$ *search*, *insert*, *delete* all cost $\Theta(\log n)$.