CS 240 – Data Structures and Data Management

Module 4: Dictionaries

A. Jamshidpey G. Kamath É. Schost

Based on lecture notes by many previous cs240 instructors

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

Spring 2020

References: Goodrich & Tamassia 3.1, 3.2, 3.6

Outline

1. Dictionaries and Balanced Search Trees
   - ADT Dictionary
   - Review: Binary Search Trees
   - AVL Trees
   - Insertion in AVL Trees
   - Restoring the AVL Property: Rotations
   - Deletion in AVL Trees

Outline

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) (also called *findElement(k)*)
- *insert*(k, v) (also called *insertItem(k,v)*)
- *delete*(k) (also called *removeElement(k)*))
- optional: *closestKeyBefore*, *join*, *isEmpty*, *size*, *etc.*

Elementary Implementations

Common assumptions:

- Dictionary has n KVPs
- Each KVP uses constant space
- Keys can be compared in constant time

Unordered array or linked list

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

Ordered array

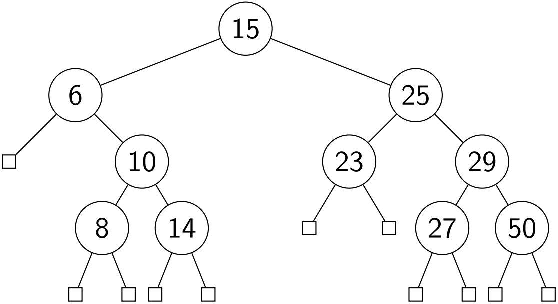
search $\Theta(\log n)$ (via binary search)
insert $\Theta(n)$
delete $\Theta(n)$

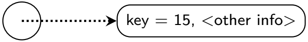
Outline

Binary Search Trees (review)

Structure Binary tree (all internal nodes have two children)
Every internal node stores a KVP
Every external node stores empty tree (usually not shown)

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.



(In our examples we only show the keys, and we show them directly in the node. A more accurate picture would be 

)

BST Search and Insert

BST-search(k) Start at root, compare k to current node.
Stop if found or node is external, else recurse at child.

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

Example:

BST Delete

- First search for the node x that contains the key.
- If x is a leaf, just delete it.
- If x has one child, move child up
- Else, swap key at x with key at *successor* or *predecessor* node and then delete that node

Height of a BST

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

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

- Worst-case: $n - 1 \in \Theta(n)$
- Best-case: $\Theta(\log n)$.
  Any binary tree with n nodes has height $\geq \log(n+1) - 1$
- Average-case: Can show $\Theta(\log n)$

Outline

AVL Trees

Introduced by Adel'son-Vel'skiĭ and Landis in 1962, an *AVL Tree* is a BST with an additional **height-balance** property:

The heights of the left subtree L and right subtree R differ by at most 1.

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

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

-1 means the tree is *left-heavy*

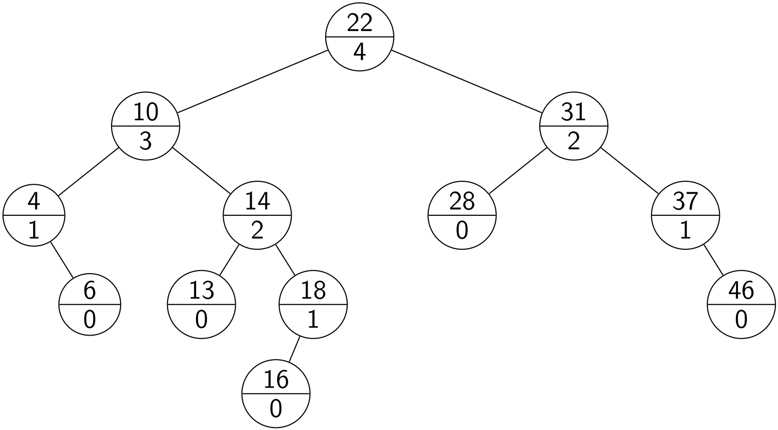
0 means the tree is *balanced*

$+1$ means the tree is *right-heavy*

- Need to store at each node the height of the subtree rooted at it
- Can show: It suffices to store $height(R) - height(L)$ at each node.
  - uses fewer bits
  - code gets more complicated, especially for deleting

AVL tree example

(The lower numbers indicate the height of the subtree.)



Height of an AVL tree

Theorem: An AVL tree on n nodes has $\Theta(\log n)$ height.
$\Rightarrow$ *AVL-search*, *AVL-insert*, *AVL-delete* all cost $\Theta(\log n)$ in the *worst case!*

Proof:

- Define $N(h)$ to be the *least* number of nodes in a height-h AVL tree.
- What is a recurrence relation for $N(h)$?
- What does this recurrence relation resolve to?

Caution, Goodrich & Tamassia uses a different height-definition, therefore their base cases are different from ours

Outline

AVL insertion

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

- First, insert (k, v) into T with the usual BST insertion.
- We assume that this returns the new leaf z where the key was stored.
- Then, move up the tree from z, updating heights.
  - We assume for this that we have parent-links. This can be avoided if BST-Insert returns the full path to z.
- If the height difference becomes ± 2 at node z, then z is *unbalanced*. Must re-structure the tree to rebalance.

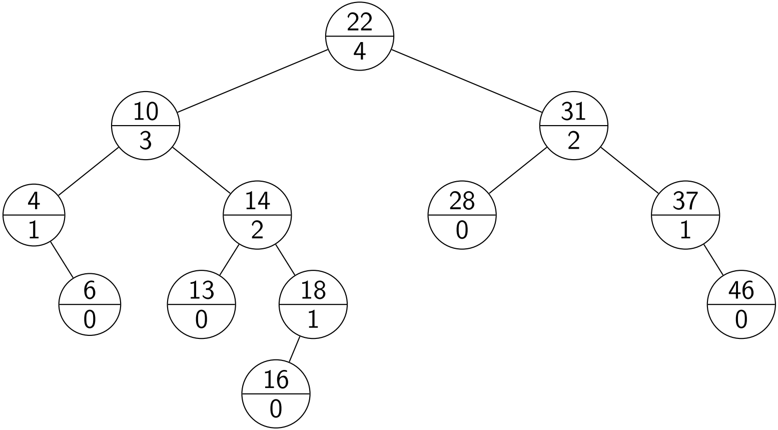
AVL insertion

```
AVL-insert(r, k, v)
    z ← BST-insert(r, k, v)
    z.height ← 0
    while (z is not null)
        setHeightFromChildren(z)
        if (|z.left.height − z.right.height| = 2) then
            AVL-fix(z) // see later
            break      // can argue that we are done
        else
            z ← parent of z
```

```
setHeightFromChildren(u)
    u.height ← 1 + max{u.left.height, u.right.height}
```

AVL Insertion Example

Example:



Fixing a slightly-unbalanced AVL tree

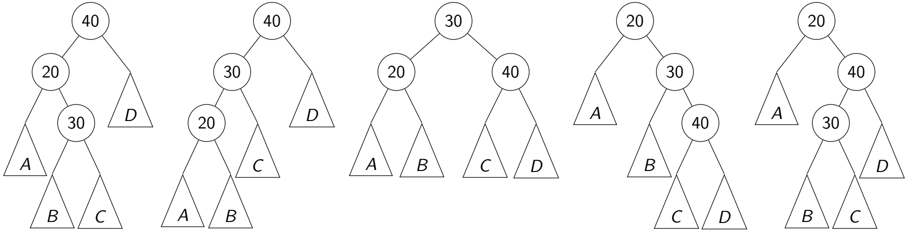
This is applied at a node z that has balance ± 2, but the subtrees at z are AVL-trees. It makes the subtree rooted at z balanced.

```
AVL-fix(z)
// Find child and grand-child that go deepest.
    if (z.right.height > z.left.height) then
        y ← z.right
        if (y.left.height > y.right.height) then
            x ← y.left
        else x ← y.right
    else
        y ← z.left
        if (y.right.height > y.left.height) then
            x ← y.right
        else x ← y.left
    Apply appropriate rotation to restructure at x, y, z
```

Outline

How to "fix" an unbalanced AVL tree

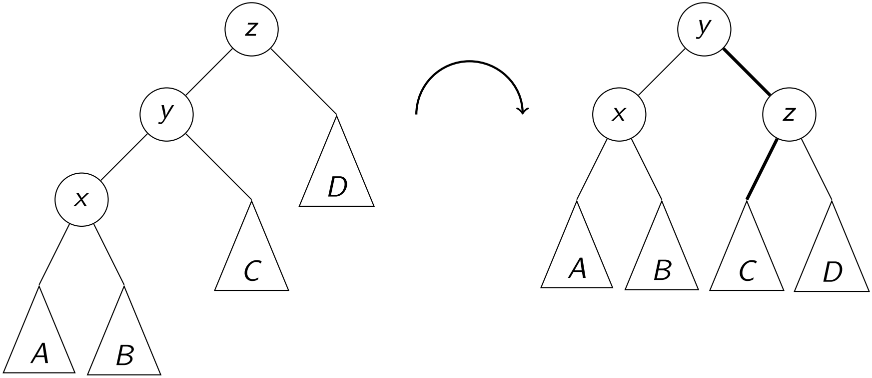
Note: there are many different BSTs with the same keys.



Goal: change the *structure* among three nodes without changing the *order* and such that the subtree becomes balanced.

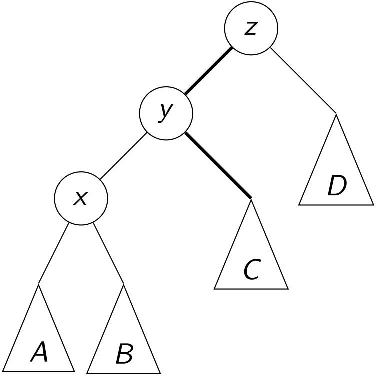
Right Rotation

This is a *right rotation* on node z:



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

Right Rotation in detail



Pseudocode for right rotation

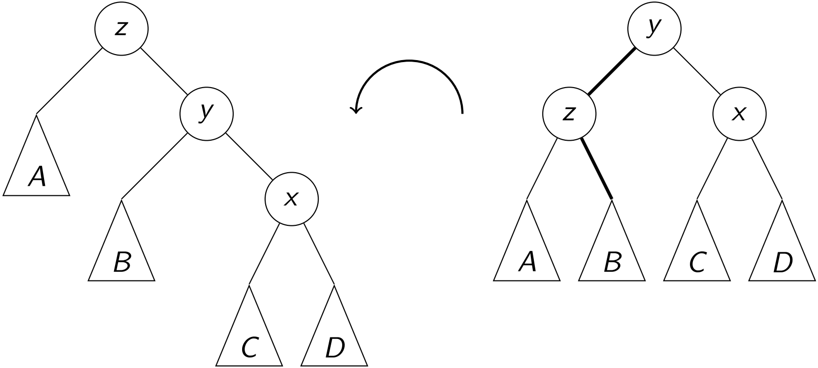
```
rotate-right(z)
z: node of BST tree
    y ← z.left
    make y.right the new left child of z
    make y the new root of the subtree
    make z the new right child of y
    setHeightFromChildren(z)
    setHeightFromChildren(y)
```

Recall: update to links also need to update the parent! For example, to make y the new root of the subtree:

```
    p ← parent of z
    if p is not null
        if z = p.left
            p.left ← y
        else p.right ← y
    else make y the overall root of the tree
```

Left Rotation

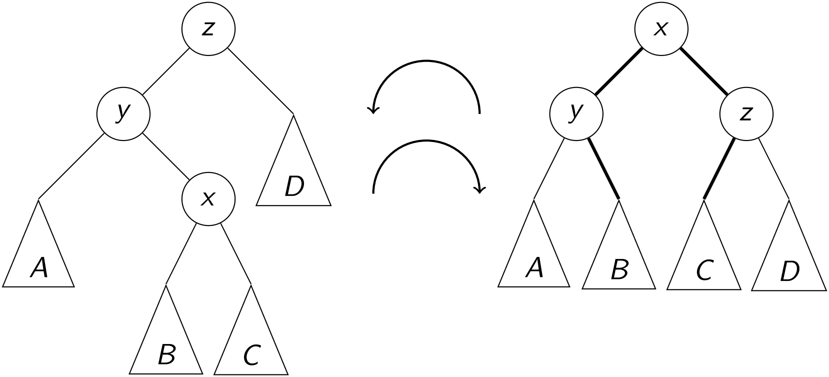
Symmetrically, this is a *left rotation* on node z:



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

Double Right Rotation

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

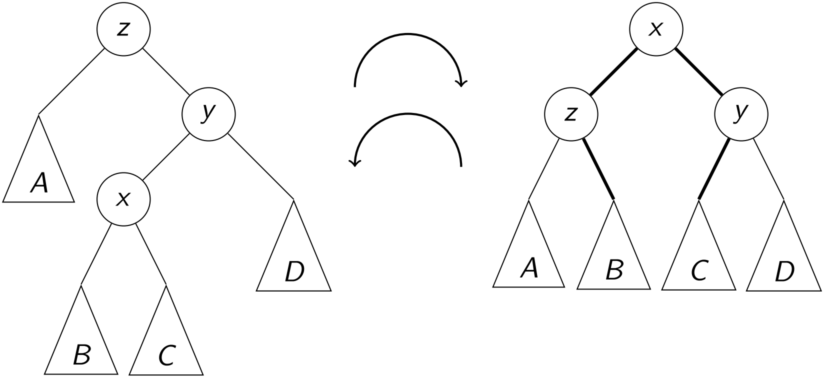


First, a left rotation at y.
Second, a right rotation at z.

Useful for left-right imbalance.

Double Left Rotation

Symmetrically, there is a *double left rotation* on node z:



First, a right rotation at y.
Second, a left rotation at z.

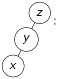
Useful for right-left imbalance.

Fixing a slightly-unbalanced AVL tree revisited

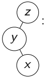
AVL-fix(z)

. . . // identify y and x as before

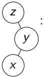
case



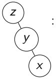
: // Right rotation
rotate-right(z)



: // Double-right rotation
rotate-left(y)
rotate-right(z)



: // Double-left rotation
rotate-right(y)
rotate-left(z)

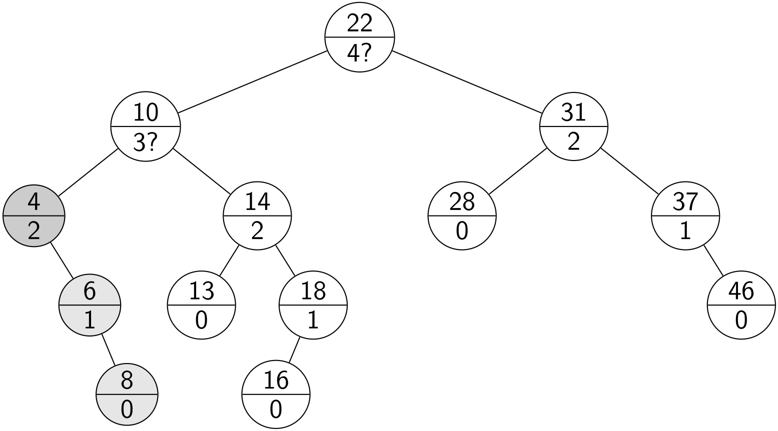


: // Left rotation
rotate-left(z)

Rule: The middle key of x, y, z becomes the new root.

AVL Insertion Example revisited

Example:



Outline

1. Dictionaries and Balanced Search Trees
   - ADT Dictionary
   - Review: Binary Search Trees
   - AVL Trees
   - Insertion in AVL Trees
   - Restoring the AVL Property: Rotations
   - Deletion in AVL Trees

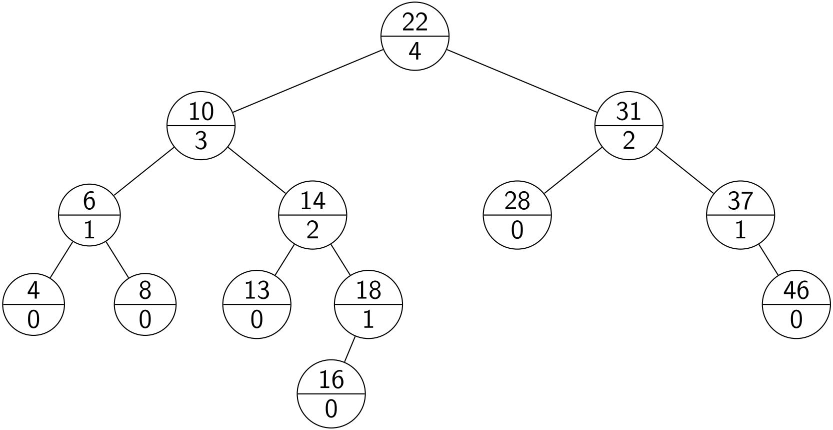
AVL Deletion

Remove the key k with *BST-delete*.
We assume that *BST-delete* returns the place where *structural* change happened, i.e., the parent z of the node that got deleted. (This is not necessarily near the one that had k.)
Now go back up to root, update heights, and rotate if needed.

AVL-delete(r, k)
    $z \leftarrow$ *BST-delete*(r, k)
    while (z is not *null*)
        setHeightFromChildren(z)
        if $(|z.left.height - z.right.height| = 2)$ **then**
            AVL-fix(z)
        // **Always** continue up the path and fix if needed.
        $z \leftarrow$ parent of z

AVL Deletion Example

Example:



AVL Tree Operations Runtime

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

AVL-insert: *BST-insert*, then check & update along path to new leaf

- total cost $\Theta(height)$
- *AVL-fix* restores the height of the tree it fixes to what it was,
- so *AVL-fix* will be called *at most once*.

AVL-delete: *BST-delete*, then check & update along path to deleted node

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

Total cost for all operations is $\Theta(height) = \Theta(\log n)$.