CS 240 – Data Structures and Data Management

Module 4: Dictionaries

T. Biedl   E. Kondratovsky   M. Petrick   O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2022

version 2022-01-25 09:52

Outline

Outline

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

Dictionary ADT

Dictionary: An ADT consisting of a collection of items, each of which contains

- a *key*
- some *data* (the "value")

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 $\Theta(1)$ (except array occasionally needs to resize)

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

Ordered array

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

insert $\Theta(n)$

delete $\Theta(n)$

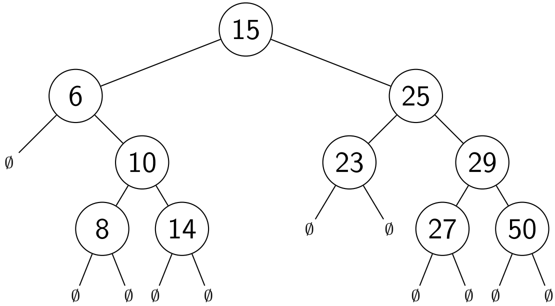
Outline

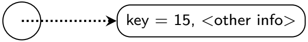
1. Dictionaries and Balanced Search Trees
   - ADT Dictionary
   - Review: Binary Search Trees
   - AVL Trees
   - Insertion in AVL Trees
   - Restoring the AVL Property: Rotations

Binary Search Trees (review)

Structure Binary tree: all nodes have two (possibly empty) subtrees
Every node stores a KVP
Empty subtrees 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 as realization of ADT Dictionary

BST::search(k) Start at root, compare k to current node's key.
Stop if found or subtree is empty, else recurse at subtree.

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

Example:

Deletion in a BST

- First search for the node x that contains the key.
- If x is a **leaf** (both subtrees are empty), delete it.
- If x has one non-empty subtree, 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 inserted one-at-a-time, how big is h?

- Worst-case: $n - 1 = \Theta(n)$
- 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

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

AVL Trees

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

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

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

Rephrase: If node v has left subtree L and right subtree R, then

$$\textbf{balance}(v) := \mathit{height}(R) - \mathit{height}(L) \text{ must be in } \{-1, 0, 1\}$$

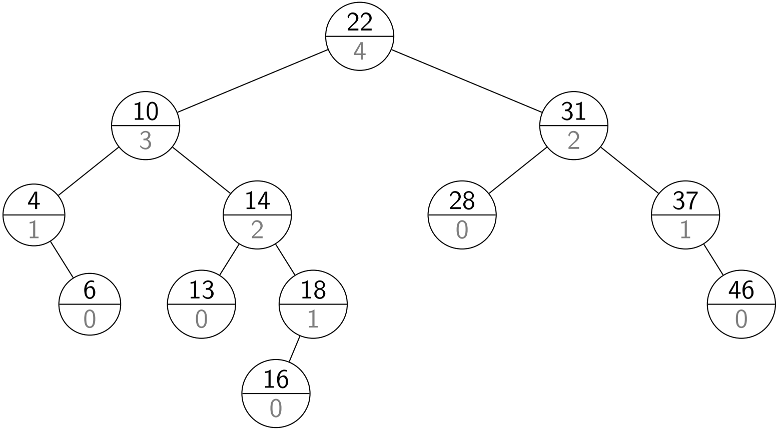
$\mathit{balance}(v) = -1$ means v is *left-heavy*

$\mathit{balance}(v) = +1$ means v is *right-heavy*

- Need to store at each node v the height of the subtree rooted at it
- Can show: It suffices to store $\mathit{balance}(v)$ instead
  - uses fewer bits, but code gets more complicated

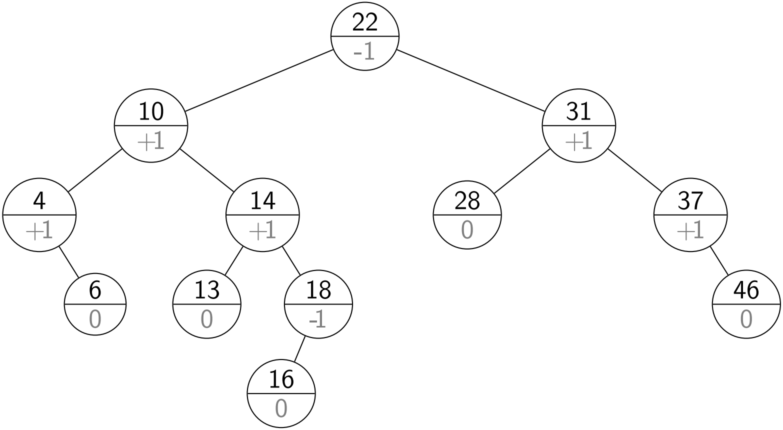
AVL tree example

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



AVL tree example

Alternative: store balance (instead of height) at each node.



Height of an AVL tree

Theorem: An AVL tree on n nodes has $\Theta(\log n)$ height.
$\Rightarrow$ *search*, *insert*, *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?

Outline

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

AVL insertion

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

- First, insert (k, v) 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.

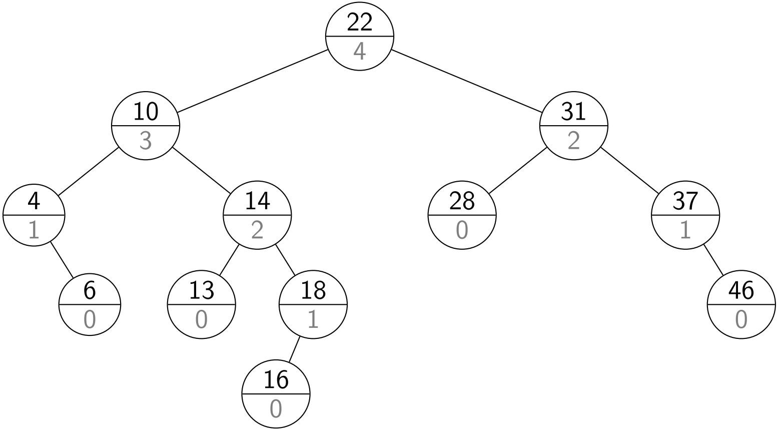
AVL insertion

```
AVL::insert(k, v)
    z ← BST::insert(k, v)   // leaf where k is now stored
    while (z is not NIL)
        if (|z.left.height − z.right.height| > 1) then
            Let y be taller child of z
            Let x be taller child of y
            z ← restructure(x, y, z) // see later
            break           // can argue that we are done
        setHeightFromSubtrees(z)
        z ← z.parent
```

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

AVL Insertion Example

Example:

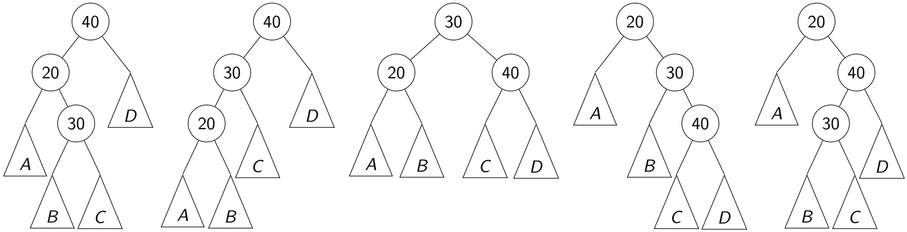


Outline

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

How to "fix" an unbalanced AVL tree

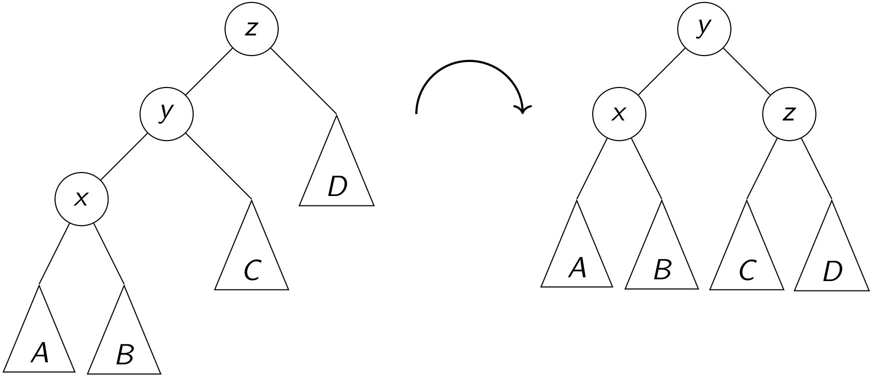
Note: there are many different BSTs with the same keys.



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

Right Rotation

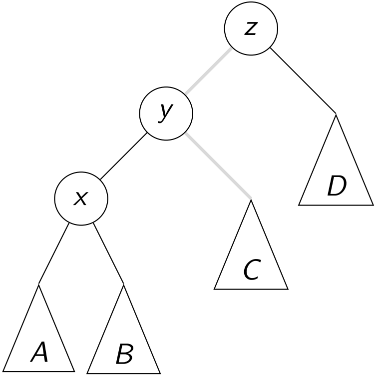
This is a **right rotation** on node z:



rotate-right(z)

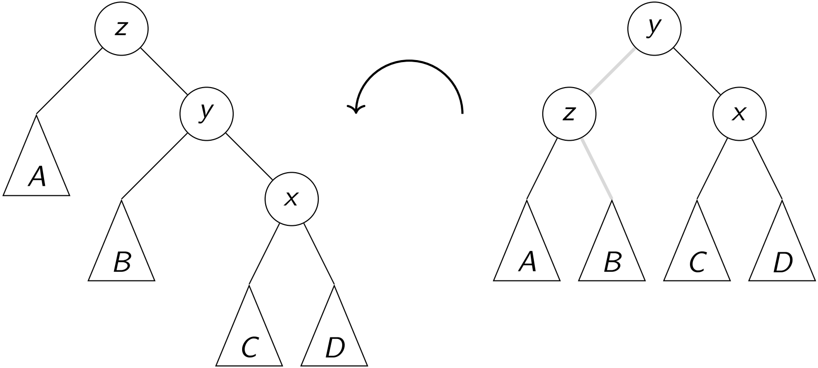
$y \leftarrow z.left$, $z.left \leftarrow y.right$, $y.right \leftarrow z$
setHeightFromSubtrees(z), *setHeightFromSubtrees*(y)
return y // returns new root of subtree

Why do we call this a rotation?



Left Rotation

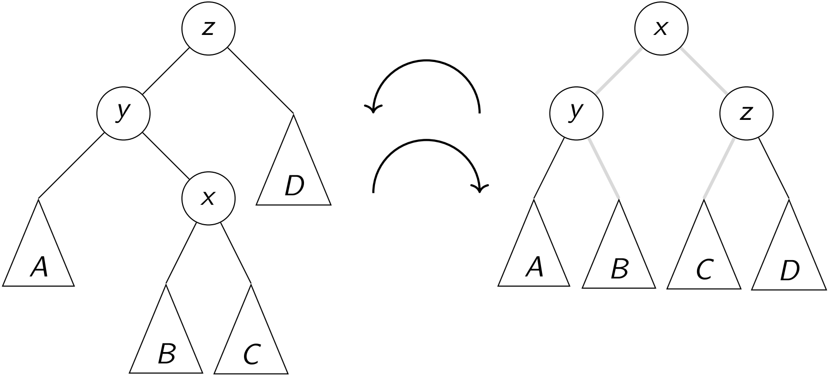
Symmetrically, this is a **left rotation** on node z:



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

Double Right Rotation

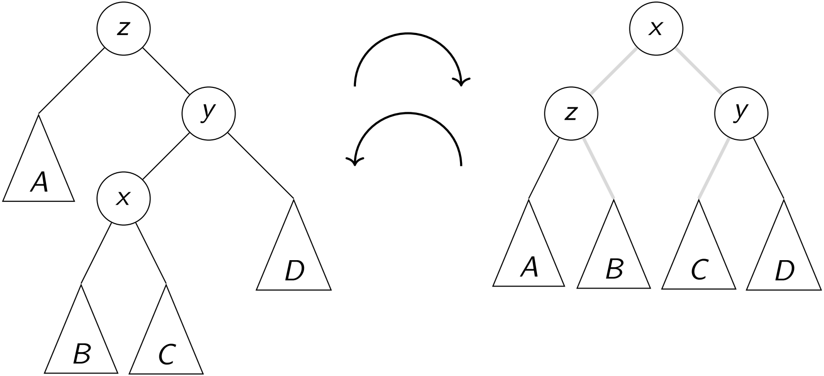
This is a **double right rotation** on node z:



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

Double Left Rotation

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

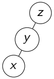


First, a right rotation at y.
Second, a left rotation at z.

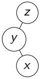
Fixing a slightly-unbalanced AVL tree

restructure(x, y, z)
node x has parent y and grandparent z

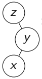
1. **case**



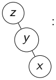
: // Right rotation
  return *rotate-right*(z)



: // Double-right rotation
  z.left $\leftarrow$ *rotate-left*(y)
  return *rotate-right*(z)



: // Double-left rotation
  z.right $\leftarrow$ *rotate-right*(y)
  return *rotate-left*(z)

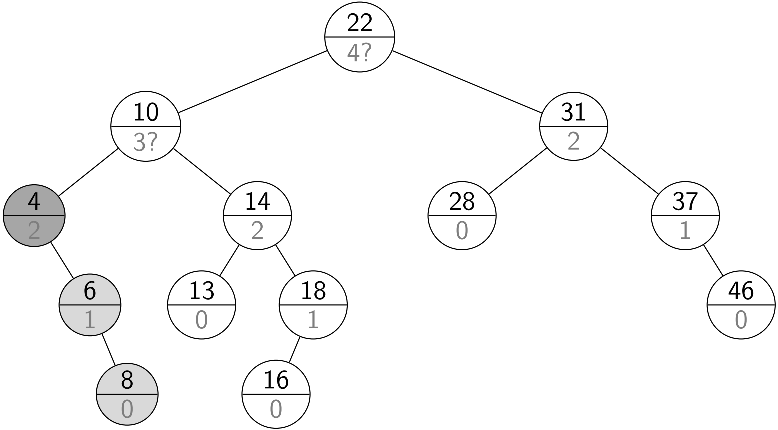


: // Left rotation
  return *rotate-left*(z)

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

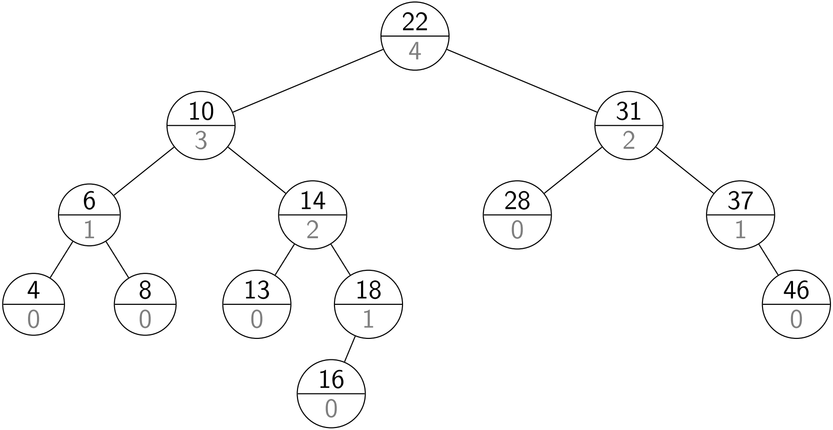
AVL Insertion Example revisited

Example:



AVL Insertion: Second example

Example: *AVL::insert*(45)



AVL Deletion

Remove the key k with *BST::delete*.

Find node where *structural* change happened.

(This is not necessarily near the node that had k.)

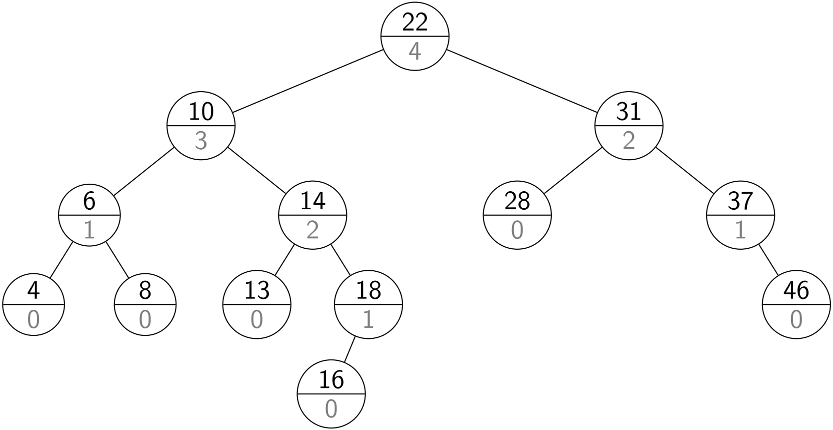
Go back up to root, update heights, and rotate if needed.

AVL::delete(k)

$z \leftarrow$ *BST::delete*(k)
// Assume z is the parent of the BST node that was removed
while (z is not `NIL`)
    if $(|z.left.height - z.right.height| > 1)$ **then**
        Let y be taller child of z
        Let x be taller child of y (break ties to prefer single rotation)
        $z \leftarrow$ *restructure*(x, y, z)
    // *Always* continue up the path and fix if needed.
    setHeightFromSubtrees(z)
    $z \leftarrow z.parent$

AVL Deletion Example

Example:



AVL Tree Operations Runtime

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

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

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

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

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

Worst-case cost for all operations is $\Theta(height) = \Theta(\log n)$.

But in practice, the constant is quite large.