

CS 240 – Data Structures and Data Management

Module 4: Dictionaries

A. Storjohann

Based on lecture notes by many previous cs240 instructors

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

Fall 2018

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

- 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

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 $\Theta(n)$

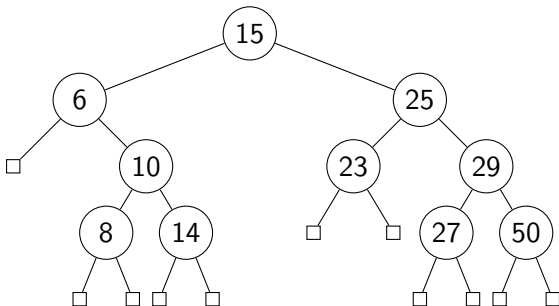
Outline

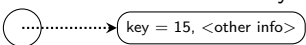
1 Dictionaries and Balanced Search Trees

- ADT Dictionary
- Review: Binary Search Trees
- AVL Trees
- Insertion in AVL Trees
- Restoring the AVL Property: Rotations
- Deletion in AVL Trees

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-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
- Deletion in AVL Trees

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 $\text{height}(R) - \text{height}(L) \in \{-1, 0, 1\}$:

-1 means the tree is *left-heavy*

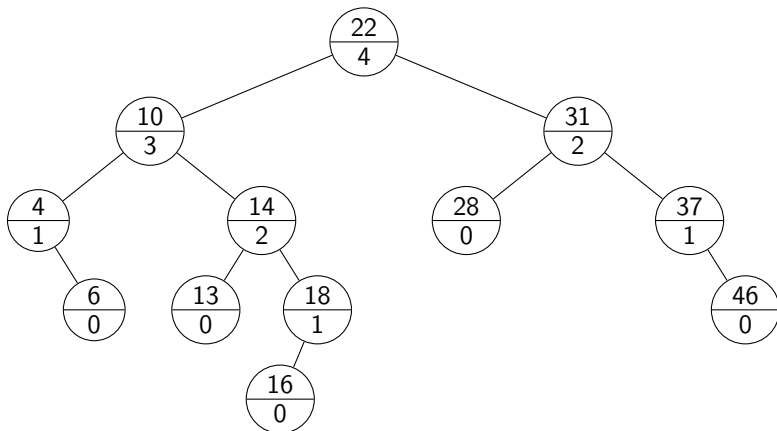
0 means the tree is *balanced*

$+1$ 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 $\text{height}(R) - \text{height}(L)$ 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

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 insertion

To perform $AVL\text{-}insert(T, k, v)$:

- 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)

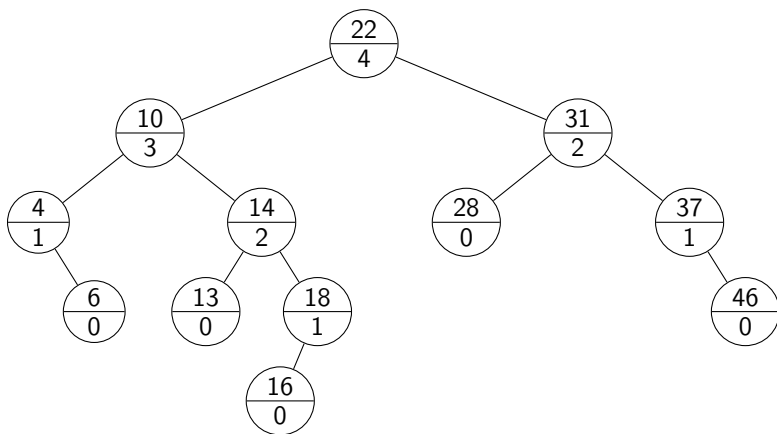
1. $z \leftarrow \text{BST-insert}(r, k, v)$
2. $z.\text{height} \leftarrow 0$
3. **while** (z is not *null*)
4. $\text{setHeightFromChildren}(z)$
5. **if** ($|z.\text{left}.\text{height} - z.\text{right}.\text{height}| = 2$) **then**
6. $\text{AVL-fix}(z)$ // see later
7. **break** // can argue that we are done
8. **else**
9. $z \leftarrow \text{parent of } z$

setHeightFromChildren(u)

1. $u.\text{height} \leftarrow 1 + \max\{u.\text{left}.\text{height}, u.\text{right}.\text{height}\}$

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.

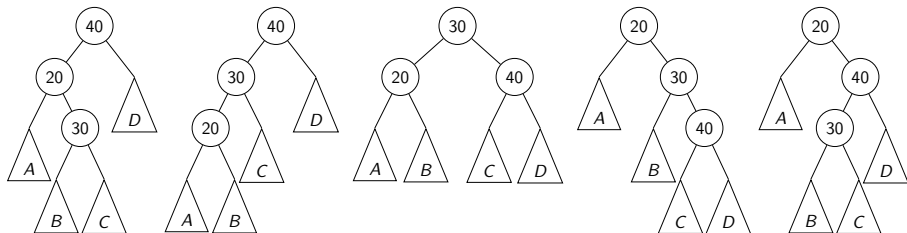
```
AVL-fix(z)
// Find child and grand-child that go deepest.
1.   if (z.right.height > z.left.height) then
2.       y ← z.right
3.       if (y.left.height > y.right.height) then
4.           x ← y.left
5.       else x ← y.right
6.   else
7.       y ← z.left
8.       if (y.right.height > y.left.height) then
9.           x ← y.right
10.      else x ← y.left
11.   Apply appropriate rotation to restructure at x, y, z
```

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

How to “fix” an unbalanced AVL tree

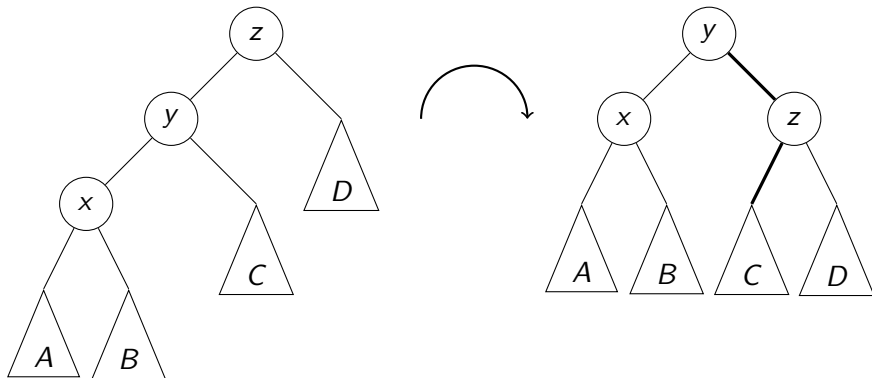
Note: there are many different BSTs with the same keys.



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

Right Rotation

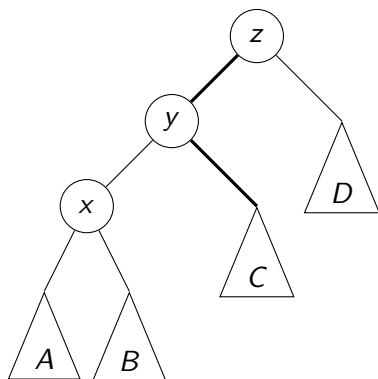
This is a *right rotation* on node z :



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

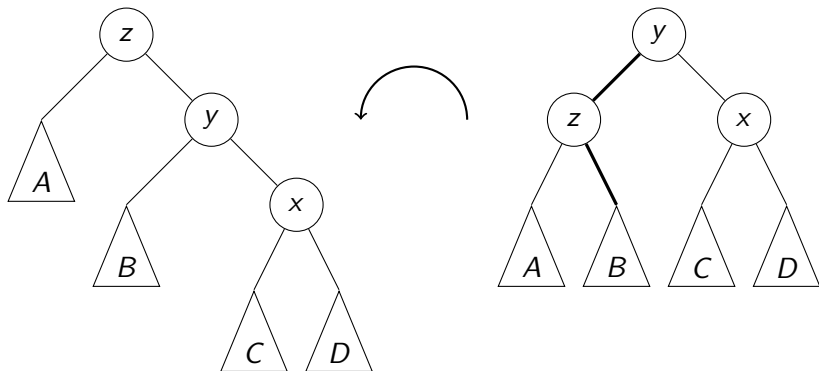
1. $y \leftarrow z.\text{left}$
2. make *y.right* the new left child of *z*
3. make *y* the new root of the subtree
4. make *z* the new right child of *y*
5. *setHeightFromChildren*(*z*)
6. *setHeightFromChildren*(*y*)

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

1. $p \leftarrow \text{parent of } z$
2. **if** *p* is not null
3. **if** $z = p.\text{left}$
4. $p.\text{left} \leftarrow y$
5. **else** $p.\text{right} \leftarrow y$
6. **else** make *y* the overall root of the tree

Left Rotation

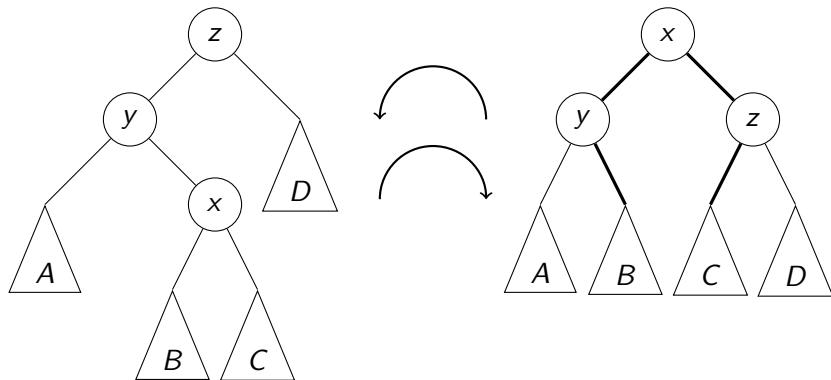
Symmetrically, this is a *left rotation* on node z :



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 :



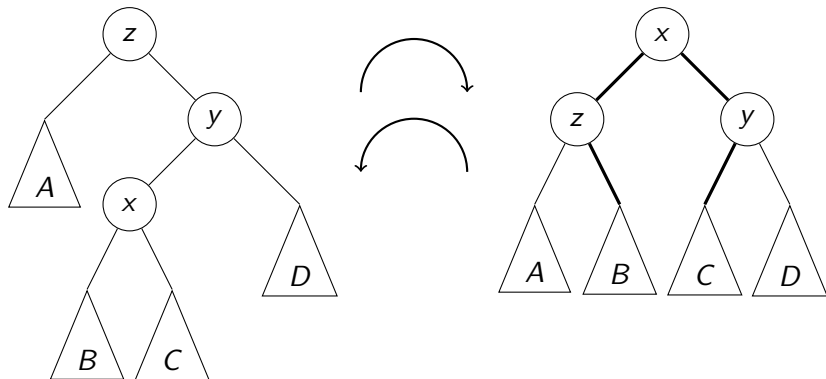
First, a left rotation at y .

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)


1. ... // identify y and x as before

2. **case**

3.  : // Right rotation
rotate-right(z)

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

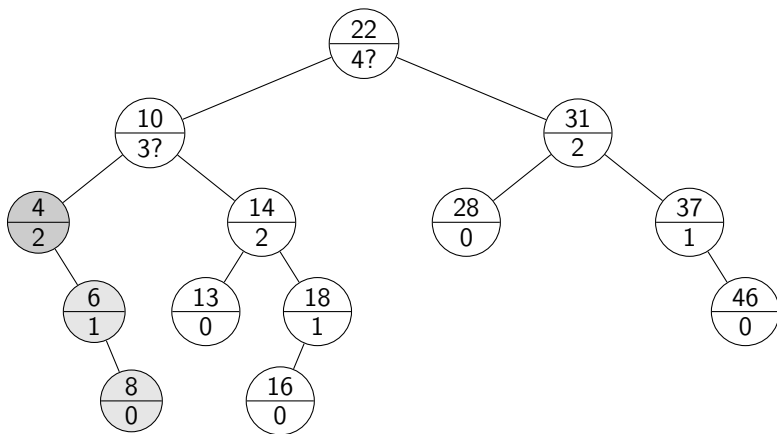
5.  : // Double-left rotation
rotate-right(y)
rotate-left(z)

6.  : // 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 .)

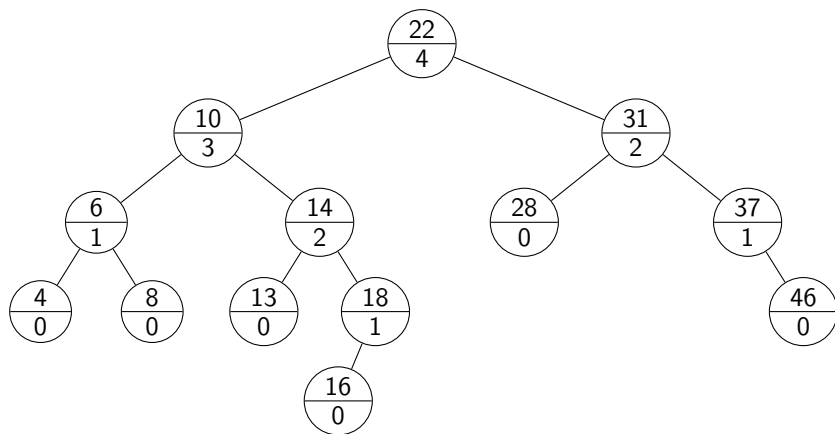
Now go back up to root, update heights, and rotate if needed.

```
AVL-delete( $r, k$ )
```

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

AVL Deletion Example

Example:



AVL Tree Operations Runtime

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

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

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

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