CSE 373: Data Structures and Algorithms

Hash Tables

Autumn 2018

Shrirang (Shri) Mare
shri@cs.washington.edu

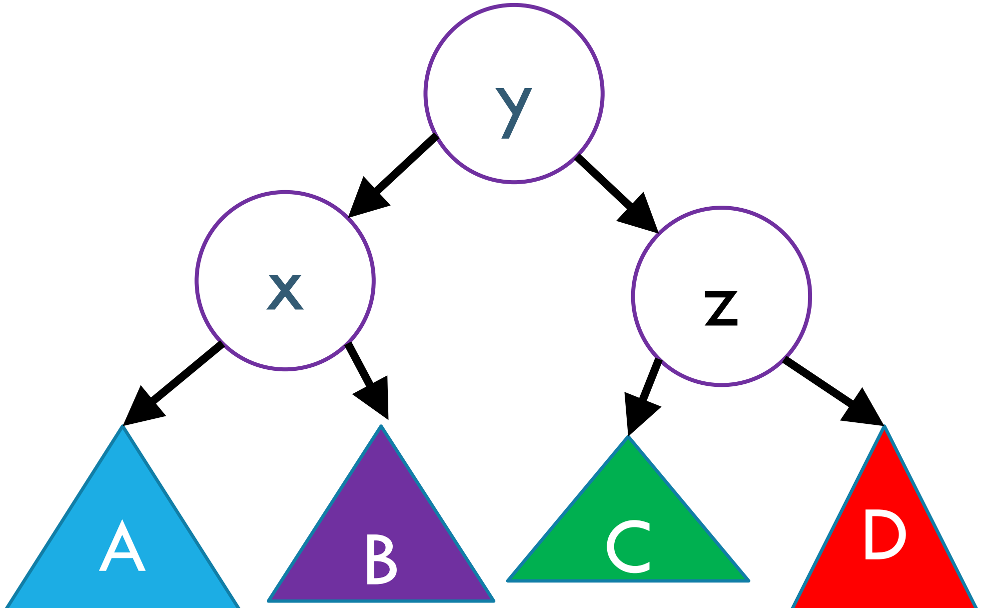
Thanks to Kasey Champion, Ben Jones, Adam Blank, Michael Lee, Evan McCarty, Robbie Weber, Whitaker Brand, Zora Fung, Stuart Reges, Justin Hsia, Ruth Anderson, and many others for sample slides and materials ...

Today

- Wrap up AVL Trees
- Problem: Can we make get(k) operation on dictionaries fast: O(1)
- Motivation
- Hashing
- Separate Chaining

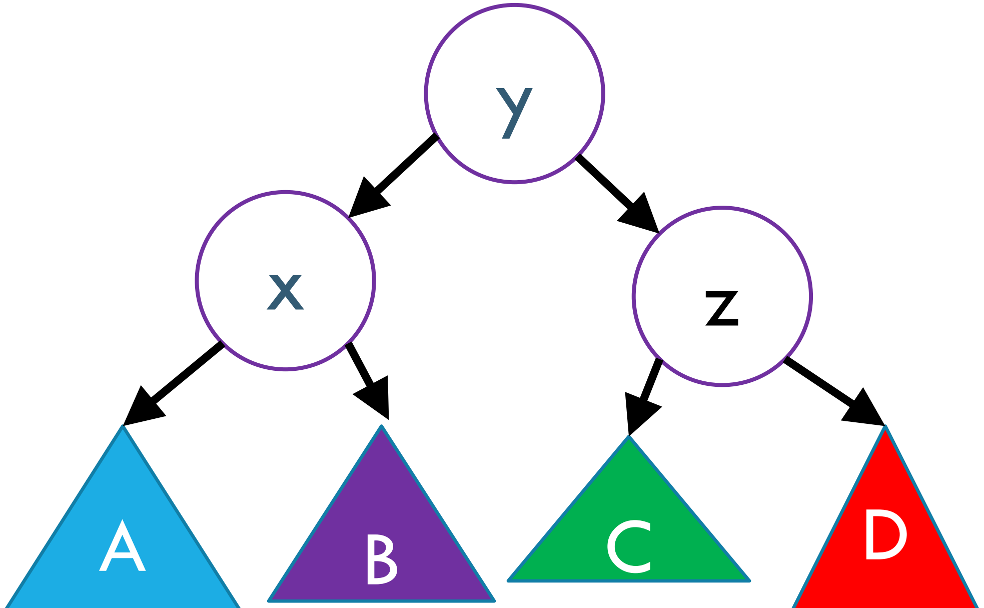
AVL Trees: Four cases to consider

Insert location	Case	Solution
Left subtree of left child of y (A)	Left line case	Single right rotation
Right subtree of left child of y (B)	Left kink case	Double (left-right) rotation
Left subtree of right child of y (C)	Right kink case	Double (right-left) rotation
Right subtree of right child of y (D)	Right line case	Single left rotation

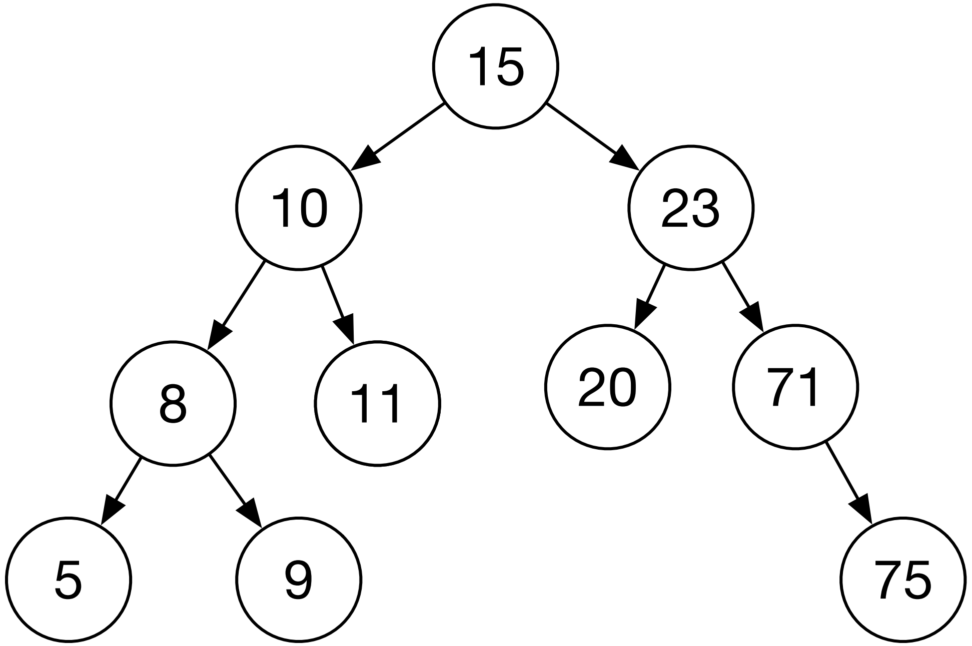


AVL Trees: Four cases to consider

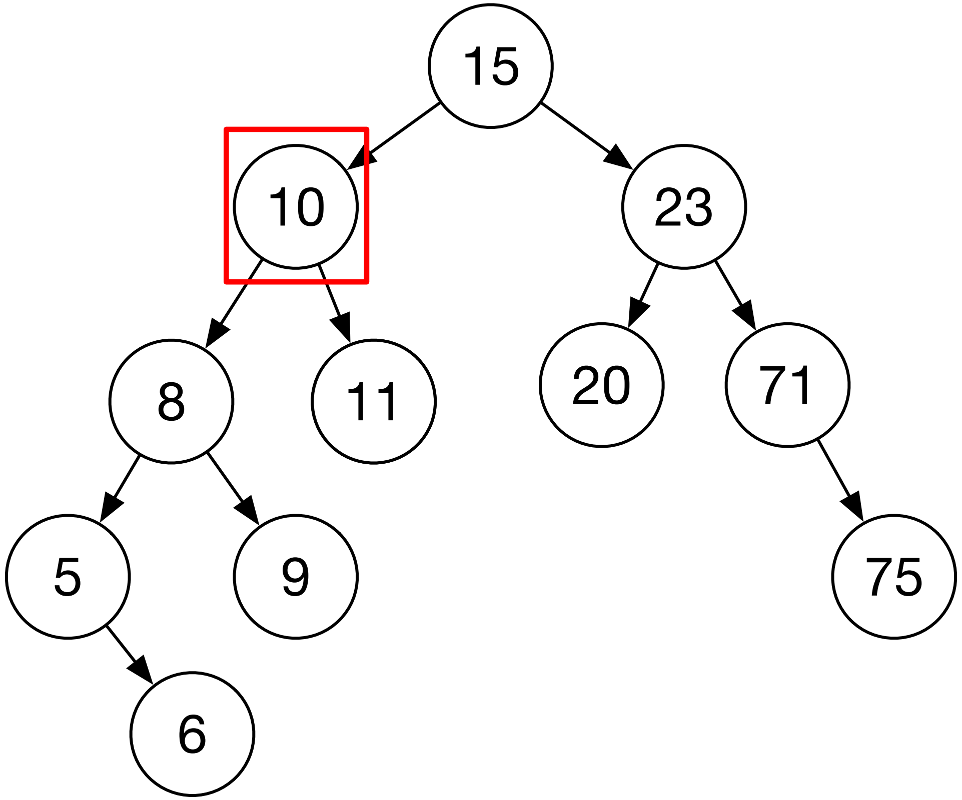
Insert location	Case (also called as)	Solution
Left subtree of left child of y (A)	Left line case (case 1)	Single right rotation
Right subtree of left child of y (B)	Left kink case (case 2)	Double (left-right) rotation
Left subtree of right child of y (C)	Right kink case (case 3)	Double (right-left) rotation
Right subtree of right child of y (D)	Right line case (case 4)	Single left rotation



AVL Tree: Practice. Insert(6)



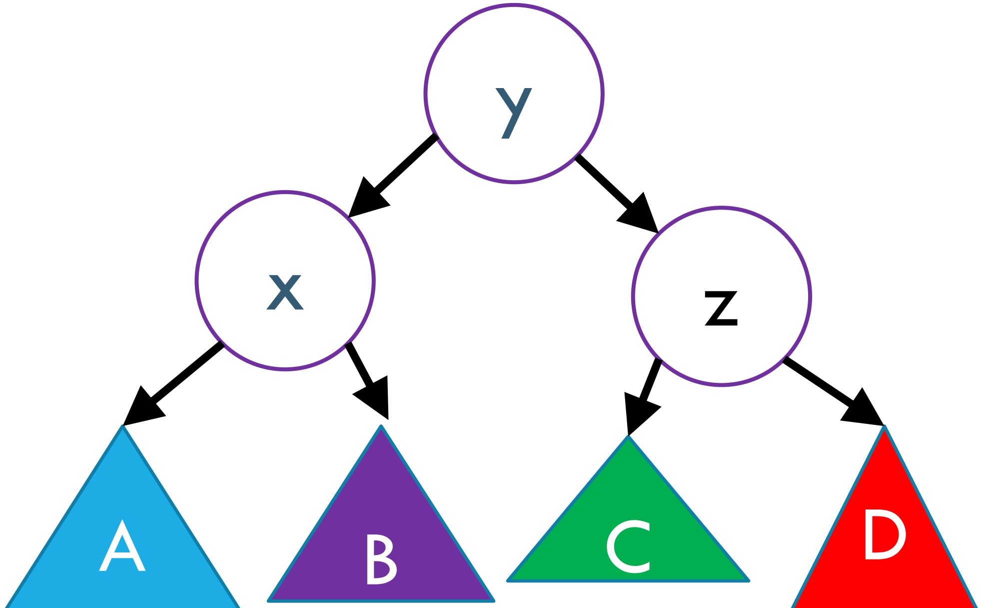
AVL Tree: Practice



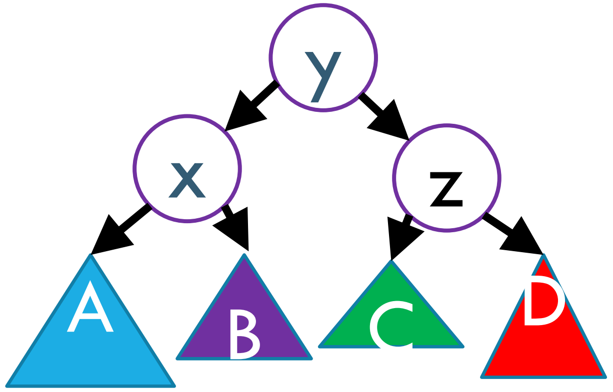
Unbalanced

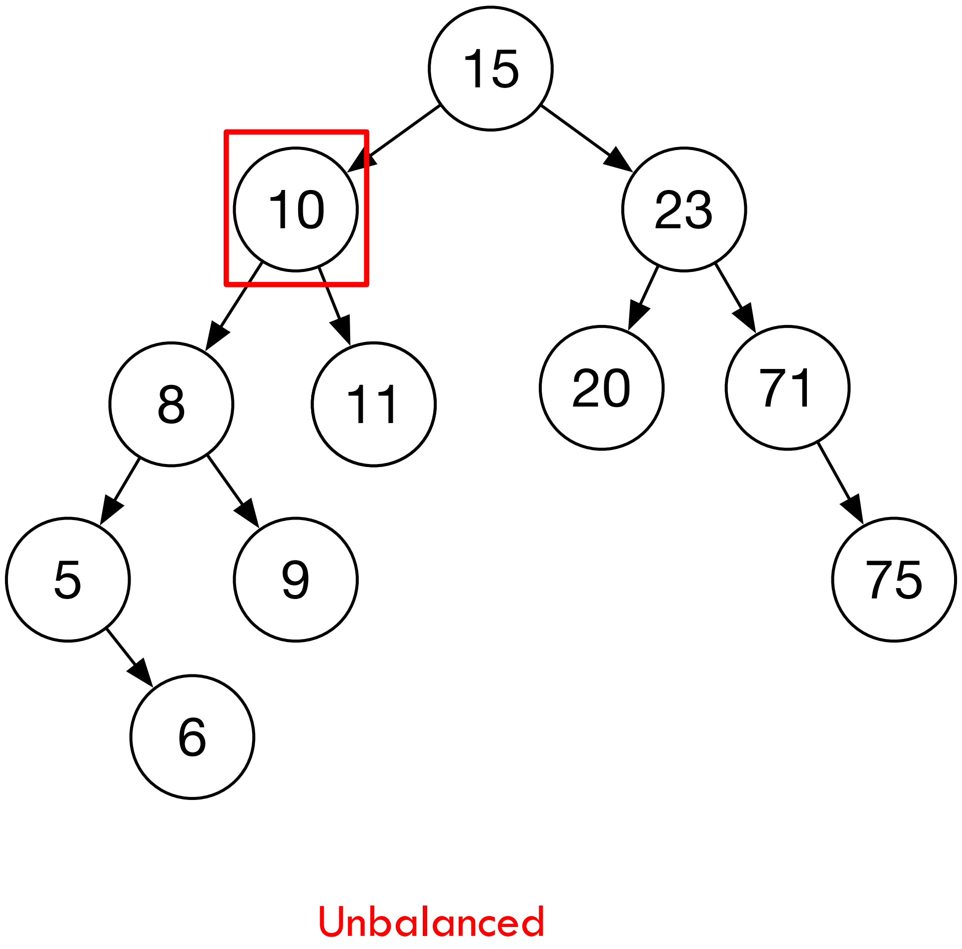
AVL Trees: Four cases to consider

Insert location	Case (also called as)	Solution
Left subtree of left child of y (A)	Left line case (case 1)	Single right rotation
Right subtree of left child of y (B)	Left kink case (case 2)	Double (left-right) rotation
Left subtree of right child of y (C)	Right kink case (case 3)	Double (right-left) rotation
Right subtree of right child of y (D)	Right line case (case 4)	Single left rotation

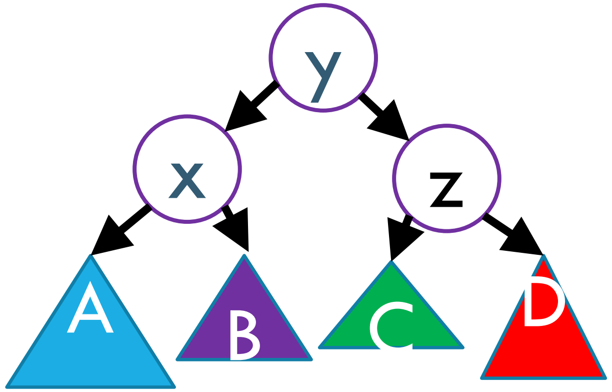


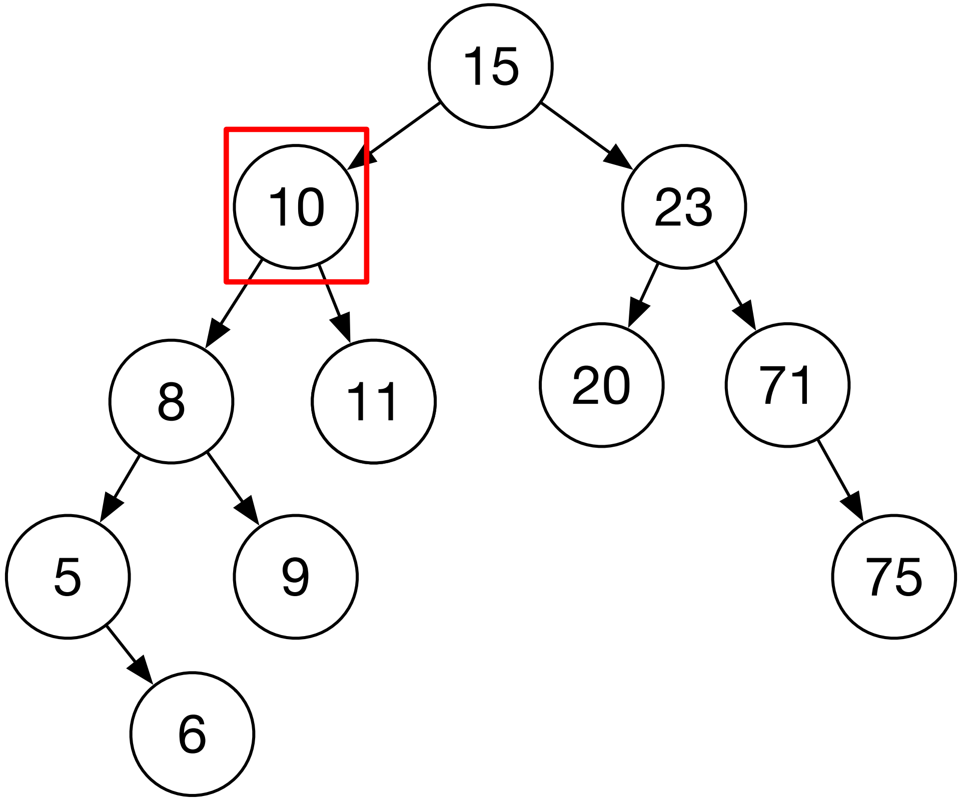
AVL Tree: Practice



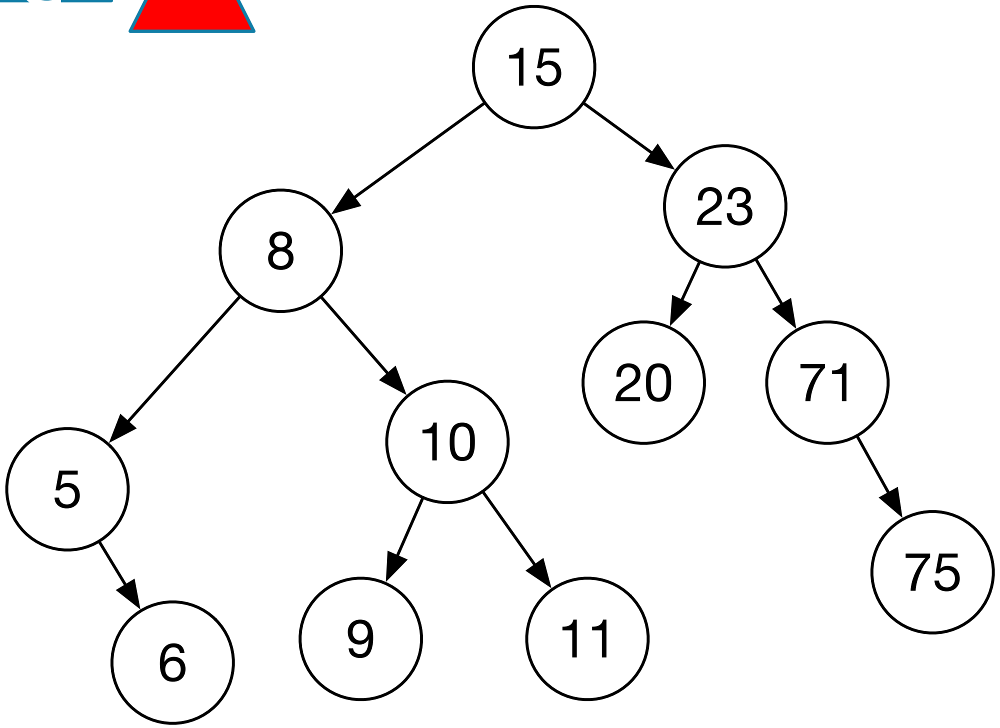


AVL Tree: Practice



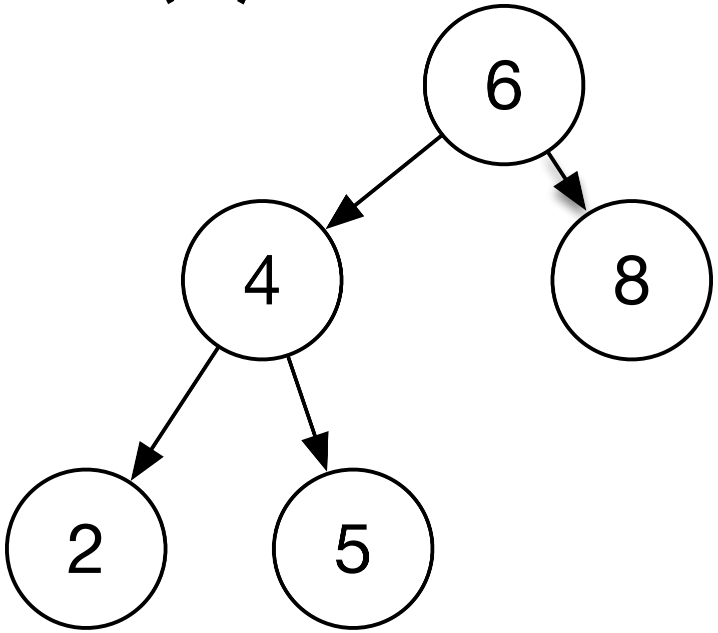


Unbalanced



Worksheet Q1

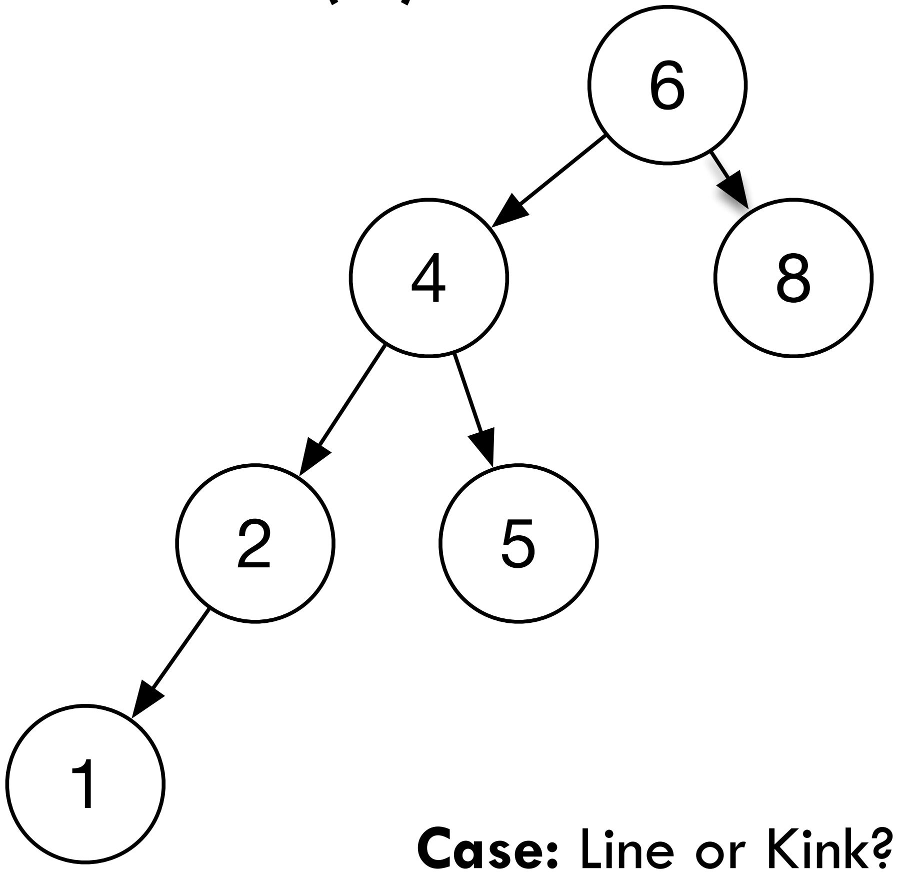
insert(1)



Case: Line or Kink?

Worksheet Q1

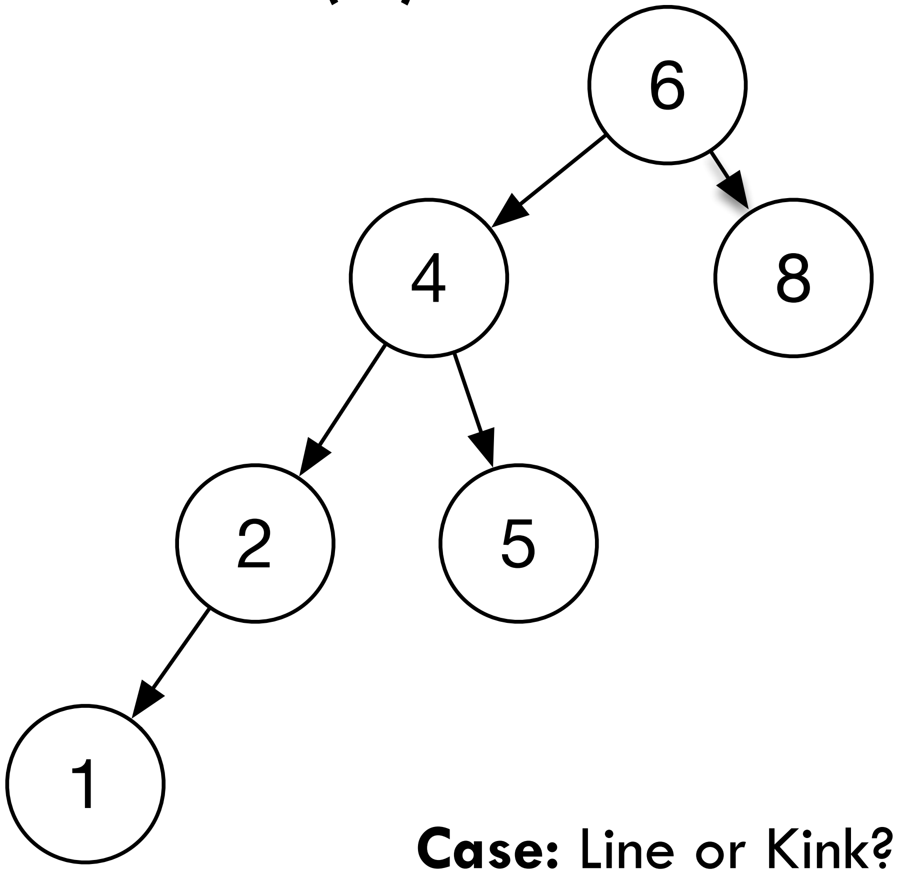
insert(1)



Case: Line or Kink?

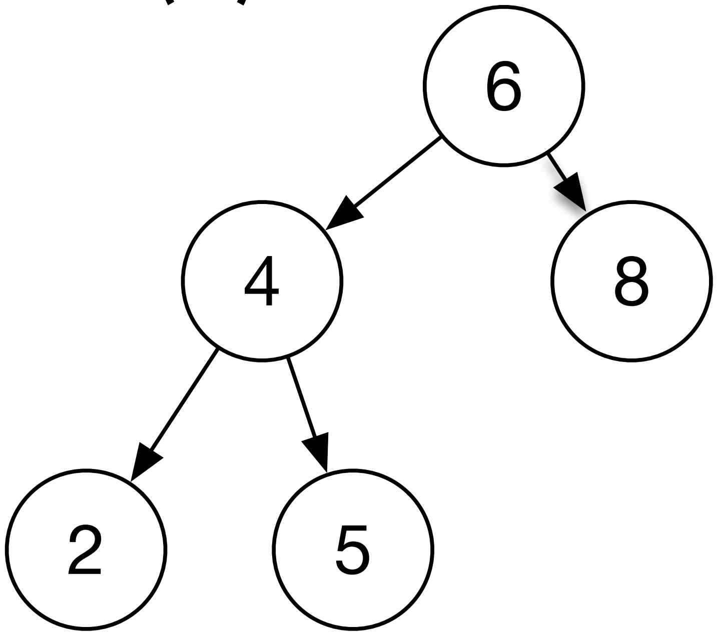
Worksheet Q1

insert(1)



Case: Line or Kink?

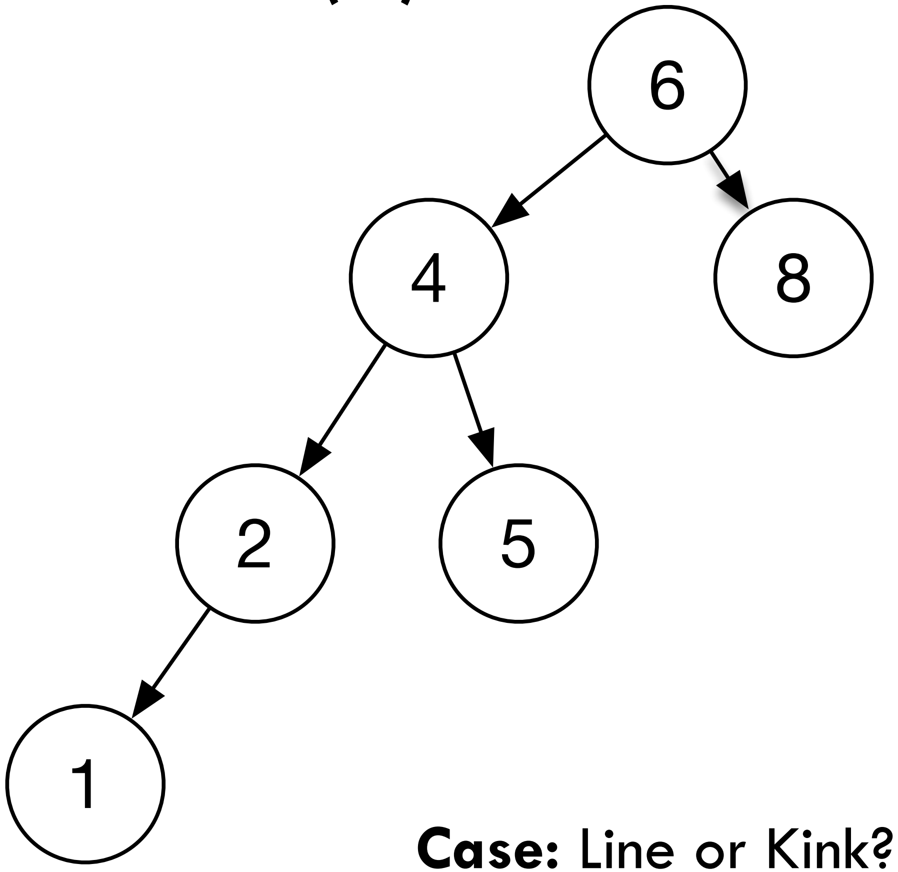
insert(3)



Case: Line or Kink?

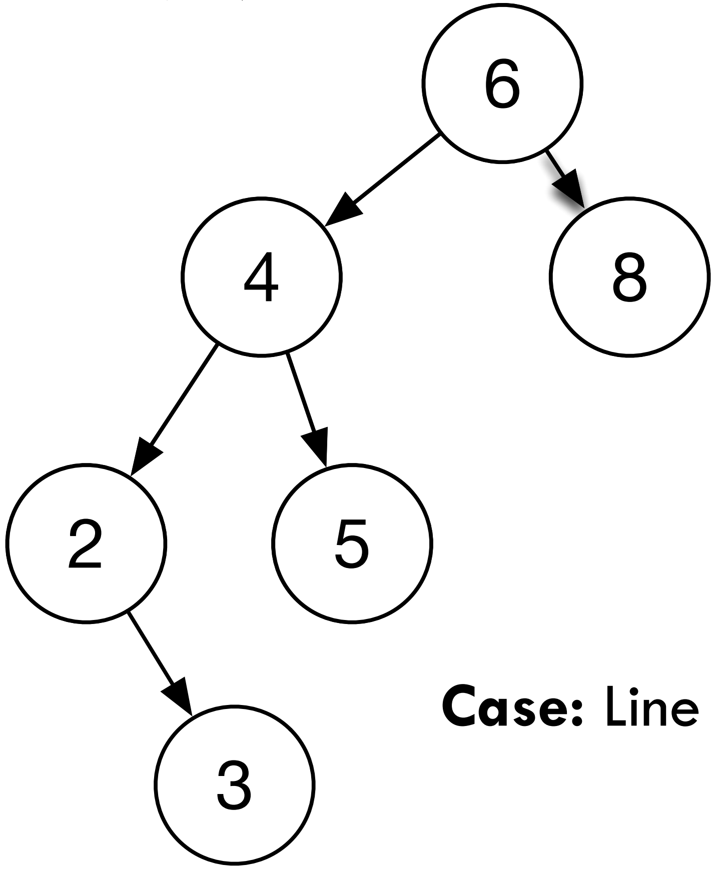
Worksheet Q1

insert(1)



Case: Line or Kink?

insert(3)



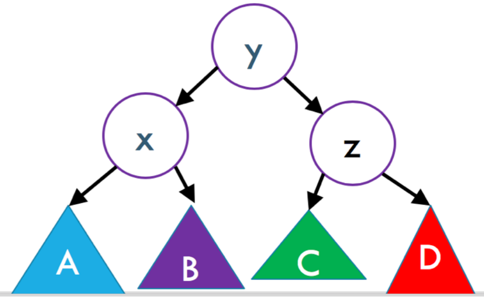
Case: Line or Kink?

AVL Tree insertions

1. Do a BST insert – insert a node as you would in a BST.
2. Check balance condition at each node in the path from the inserted node to the root.
3. If balance condition is not true at a node, identify the case
4. Do the corresponding rotation for the case

AVL Trees: Four cases to consider

Insert location	Case (also called as)	Solution
Left subtree of left child of y (A)	Left line case (case 1)	Single right rotation
Right subtree of left child of y (B)	Left kink case (case 2)	Double (left-right) rotation
Left subtree of right child of y (C)	Right kink case (case 3)	Double (right-left) rotation
Right subtree of right child of y (D)	Right line case (case 4)	Single left rotation



Worksheet Q2

Draw the AVL tree that results from inserting the keys 1, 3, 7, 5, 6, 9 in that order into an initially empty AVL tree. (*Hint:* Drawing intermediate trees as you insert each key can help.)

How long does AVL insert take?

AVL insert time = BST insert time + time it takes to rebalance the tree

= O(log n) + time it takes to rebalance the tree

How long does rebalancing take?

- Assume we store in each node the height of its subtree.
- How long to find an unbalanced node:
  - Just go back up the tree from where we inserted.
- How many rotations might we have to do?
  - Just a single or double rotation on the lowest unbalanced node.

AVL insert time = O(log n)+ O(log n) + O(1) = O(log n)

How long does AVL insert take?

AVL insert time = BST insert time + time it takes to rebalance the tree

= O(log n) + time it takes to rebalance the tree

How long does rebalancing take?

- Assume we store in each node the height of its subtree.
- How long to find an unbalanced node:
  - Just go back up the tree from where we inserted. ← O(log n)
- How many rotations might we have to do?
  - Just a single or double rotation on the lowest unbalanced node. ← O(1)

AVL insert time = O(log n)+ O(log n) + O(1) = O(log n)

AVL wrap up

Pros:

- $O(\log n)$ worst case for `find`, `insert`, and `delete` operations.
- Reliable running times than regular BSTs (because trees are balanced)

Cons:

- Difficult to program & debug [but done once in a library!]
- (Slightly) more space than BSTs to store node heights.

Lots of cool Self-Balancing BSTs out there!

Popular self-balancing BSTs include:

AVL tree

Splay tree

2-3 tree

AA tree

Red-black tree

Scapegoat tree

Treap

(Not covered in this class, but several are in the textbook and all of them are online!)

(From https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree#Implementations)

Announcements

- HW3 out. Due this Friday (10/26) at **Noon** (not at the usual time 11:59pm)
- HW4 out later today (or latest by tomorrow morning). Due next Tuesday (10/30)

- Midterm coming up – Nov 2, 2:30-3:20pm, here in the class
- If you can't take the midterm on Nov 2, let me know ASAP.
- Midterm practice material will be posted on the website tomorrow
- Midterm review next Wednesday

Hash tables

Revisiting Dictionaries

- data = (key, value)
- operations: `put(key, value);` `get(key);` `remove(key)`

- O(n) with Arrays and Linked List
- O(log n) with BST and AVL trees.

- Can we do better? Can we do this in O(1) ?

Motivation

Why we are so obsessed with making dictionaries fast?

Dictionaries are extremely most common data structures.

- Databases
- Network router tables
- Compilers and Interpreters
- Faster than O(log n) search in certain cases
- Data type in most high level programming languages

Question

How would you implement at dictionary such that dictionary operations are O(1)?

(Assume all keys are non-zero integers)

Question

How would you implement at dictionary such that dictionary operations are O(1)?

(Assume all keys are non-zero integers)

Idea: Create a giant array and use keys as indices

Question

How would you implement at dictionary such that dictionary operations are O(1)?

(Assume all keys are non-zero integers)

Idea: Create a giant array and use keys as indices

Problems?

1. ?

2. ?

Question

How would you implement at dictionary such that dictionary operations are O(1)?

(Assume all keys are non-zero integers)

Idea: Create a giant array and use keys as indices

Problems?

1. Can only work with integer keys?
2. Too much wasted space

Idea 2: Can we convert the key space into a smaller set that would take much less memory

Solve problem: Too much wasted space

Review: Integer remainder with %

The `%` operator computes the remainder from integer division.

`14 % 4` is `2` `218 % 5` **is** `3`

```
      3            43
4 ) 14       5 ) 218
    12           20
     2            18
                  15
                   3
```

Applications of `%` operator:

- Obtain last digit of a number: `230857 % 10` is `7`
- See whether a number is odd: `7 % 2` is `1`, `42 % 2` is `0`
- Limit integers to specific range: `8 % 12` is `8`, `18 % 12` is `6`

Implement Direct Access Map

```
public V get(int key) {
   // input validation
   return this.array[key].value;
}

public void put(int key, V value) {
    this.array[key] = value;
}

public void remove(int key) {
   // input validation
   this.array[key] = null;
}
```

Implement First Hash Function

```
public V get(int key) {
   // input validation
   int newKey = key % this.array.length;
   return this.array[newkey].value;
}

public void put(int key, V value) {
    this.array[key % this.array.length] = value;
}
public void remove(int key) {
   // input validation
   int newKey = key % this.array.length;
   this.array[newKey] = null;
}
```

First Hash Function: % table size

indices	0	1	2	3	4	5	6	7	8	9
elements	"foo"	"biz"				"bar"			"bop"	

"poo"

```
put(0, "foo");  0 % 10 = 0
put(5, "bar");  5 % 10 = 5
put(11, "biz")  11 % 10 = 1
put(18, "bop"); 18 % 10 = 8
put(20, "poo"); 20 % 10 = 0
```

→ Collision!

Today

√ Wrap up AVL Trees

√ Problem: Can we make get(k) operation on dictionaries fast: O(1)

√ Motivation

√ Hashing

- Separate Chaining

Hashing: Separate Chaining