AVL Trees (cont); Memory Hierarchy

CSE 332 Spring 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Annie Mao	Howard Xiao	Lucy Jiang
Chris Choi	Jade Watkins	Nathan Lipiarski
Ethan Knutson	Khushi Chaudhari	Richard Jiang

Announcements

- Exercise 5 due tonight! Exercises 6 and 7 out now; due next week

- This week is a quiz week! Released on Wednesday, due on Friday

- If you haven't contacted your p2 partner yet, do so ASAP!

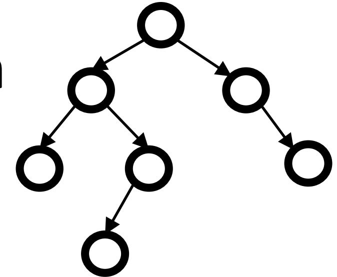
Learning Objectives

- Be able to implement and prove runtimes for AVL operations
- Describe the computer's memory hierarchy at a high level and discuss how it might affect data structures we've studied so far (eg, binary heaps, AVL trees, etc)

Lecture Outline

- AVL Tree
  - **Add (cont): Double rotations**
  - Add exercises
  - Remove
  - Wrapup
- Memory Hierarchy Basics
  - What is the Memory Hierarchy?
  - How does it impact data structure design?

Review: AVL add(): Overall Approach



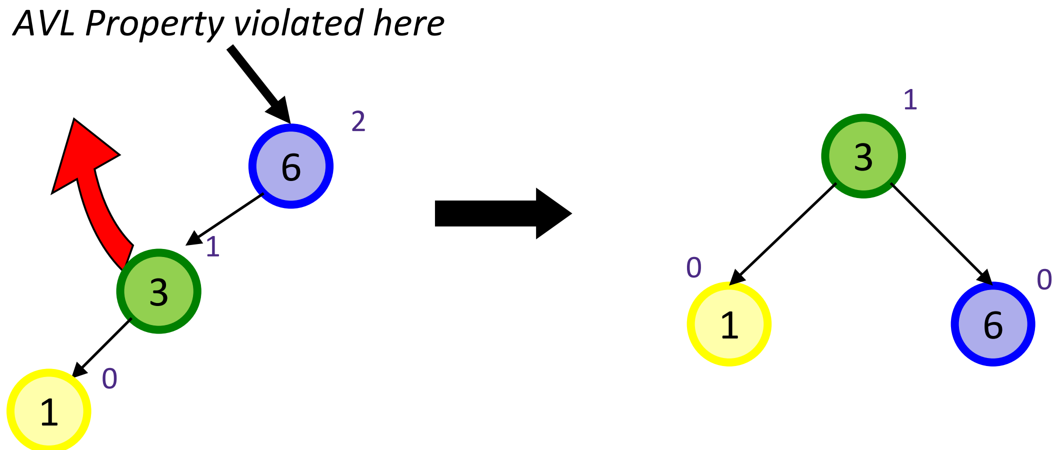
- Our overall algorithm looks like:
  1. Insert the new node as in a BST (a new leaf)
  2. For each node on the path from the root to the new leaf:
     - The insertion may (or may not) have changed the node's height
     - Detect node's height imbalance and perform a *rotation* to restore balance
     - Only one node needs rebalancing, so can exit!

- Four cases to consider. The insertion is in the:
  1. left subtree of the left child of *b*
  2. right subtree of the left child of *b*
  3. left subtree of the right child of *b*
  4. right subtree of the right child of *b*

Reminder: a dictionary maps *keys* to *values*; an *item* or *data* refers to the (key, value) pair

Case #1 and Case #4 Fix: Apply "Single Rotation"

- ***Single rotation:***
  - Move child of unbalanced node into parent position
  - Parent becomes the "other" child



Case #3: Example

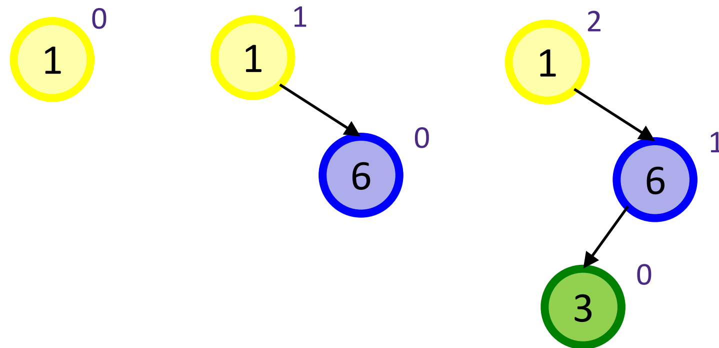
The insertion is in the:

1. left subtree of the left child of *b*
2. right subtree of the left child of *b*
3. left subtree of the right child of *b*
4. right subtree of the right child of *b*

Insert(1)

Insert(6)

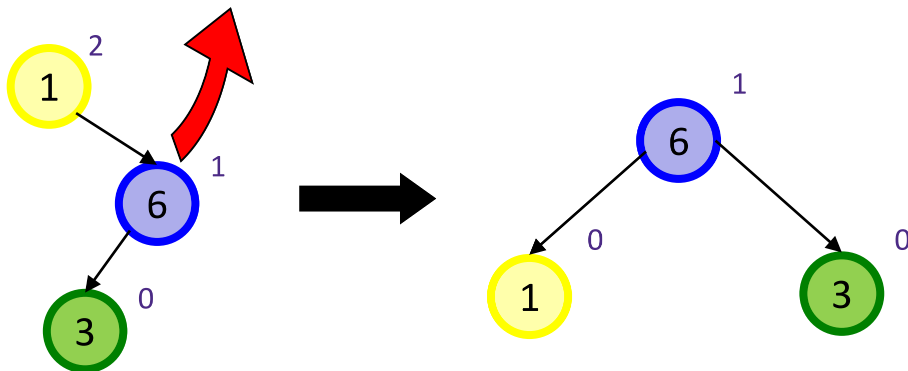
Insert(3)



- Single rotations are not enough for insertions into the left-right subtree (or the right-left subtree; ie, case #2)

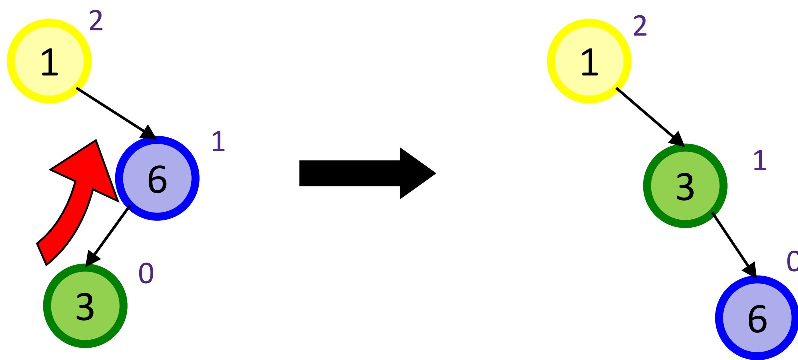
Case #3: Wrong Fix #1

- **First wrong idea:** single rotation like we did for left-left
  - Violates BST ordering property!



Case #3: Wrong Fix #2

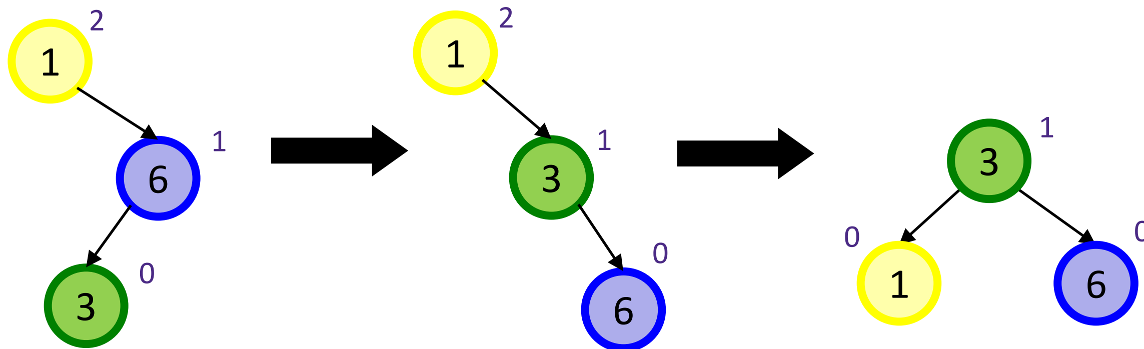
- **Second wrong idea**: single rotation on the child of the unbalanced node
  - Doesn't actually fix anything!



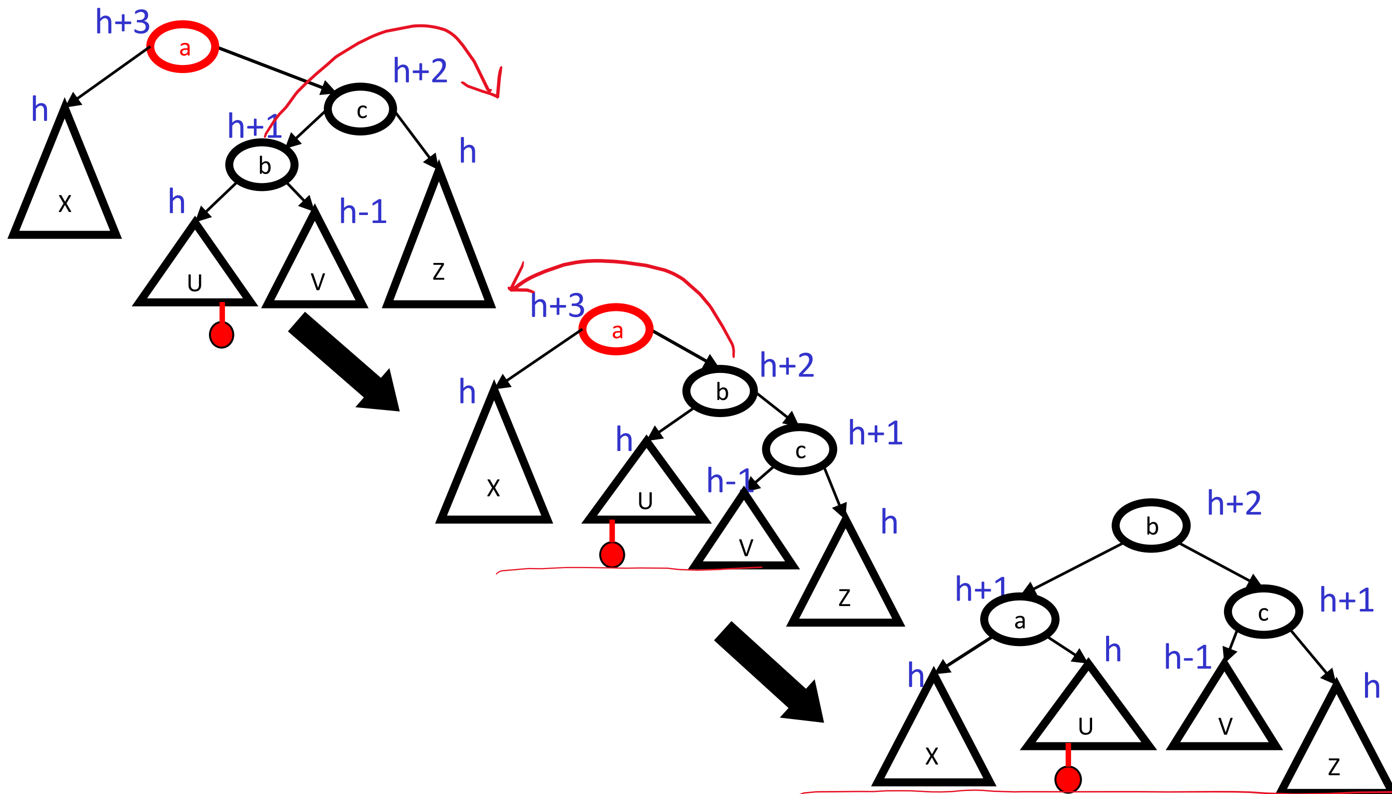
Case #3: Sometimes Two Wrongs Make a Right ☺

- First idea violated the BST ordering
- Second idea didn't fix balance
- … but if we do both single rotations, starting with the second, it works!

DoubleRotation:
1. Rotate problematic child and grandchild
2. Then rotate between self and new child

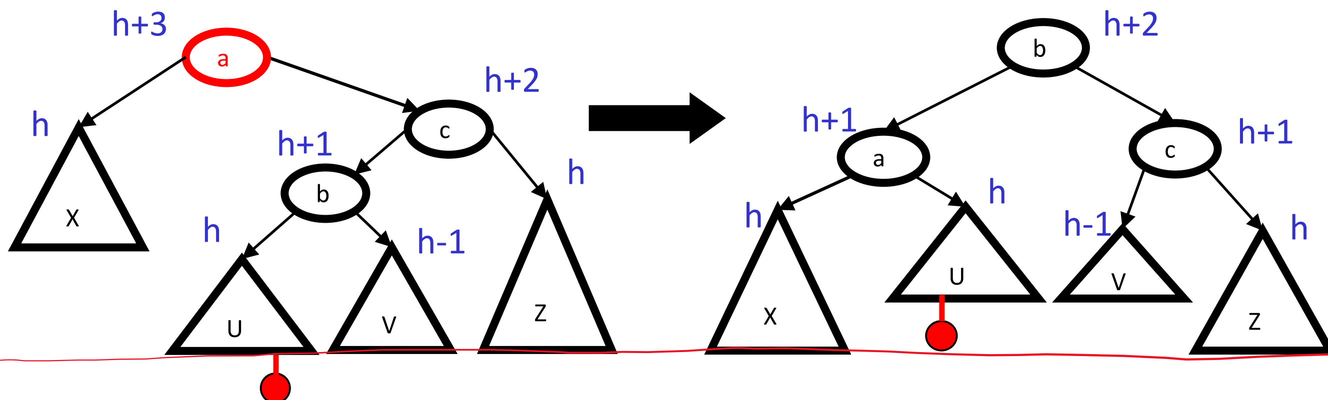


Case #3: Why It Works



Case #3: Comments

- Height of subtree after rebalancing is the same as before insert
  - So, no ancestor in the tree will need rebalancing
- Doesn't have to be two rotations; can just move b to grandparent's position and put a, c, X, U, V, and Z in the only legal positions for a BST



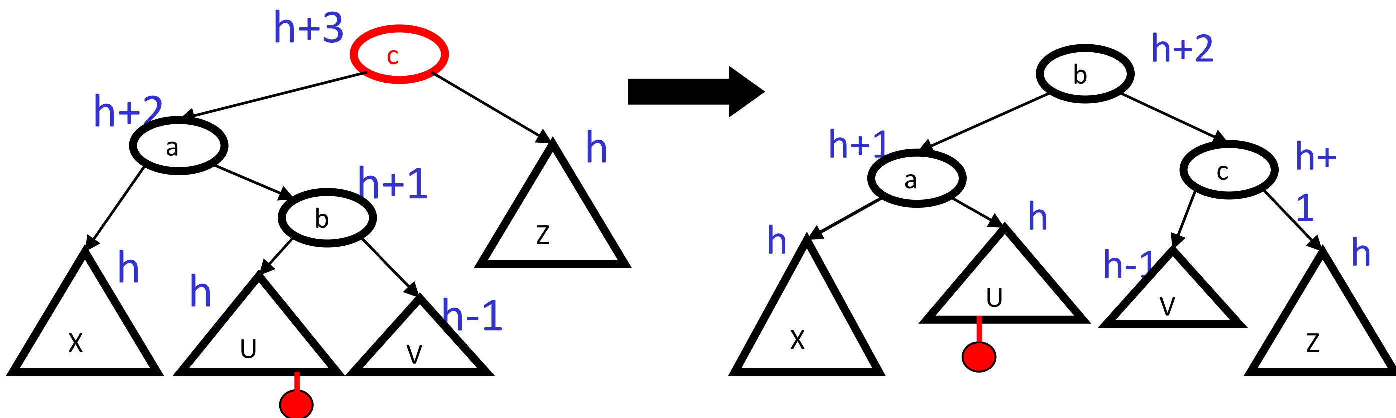
Case #3: Pseudocode

```
void DoubleRotateWithRightChild(Node root) {
  RotateWithLeftChild(root.right)
  RotateWithRightChild(root)
}
```

Case #3 ≈ Case #2

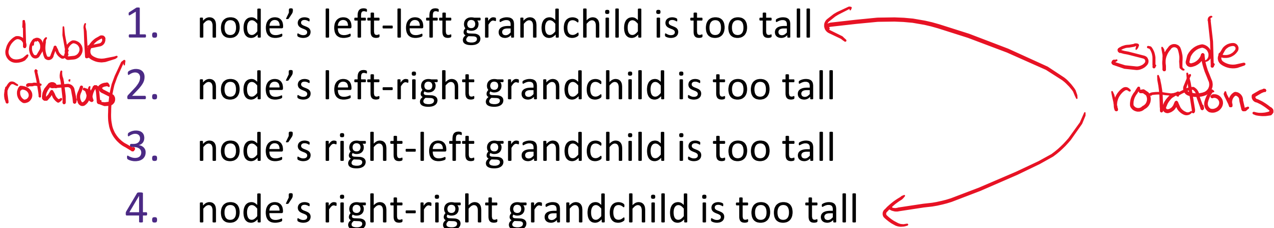
The insertion is in the:
1. left subtree of the left child of *b*
2. right subtree of the left child of *b*
3. left subtree of the right child of *b*
4. right subtree of the right child of *b*

- Mirror image of right-left
  - Again, no new concepts, only new code to write



AVL add(): Summary

- Insert as if a BST
- Check back up path for imbalance, which will be 1 of 4 cases:



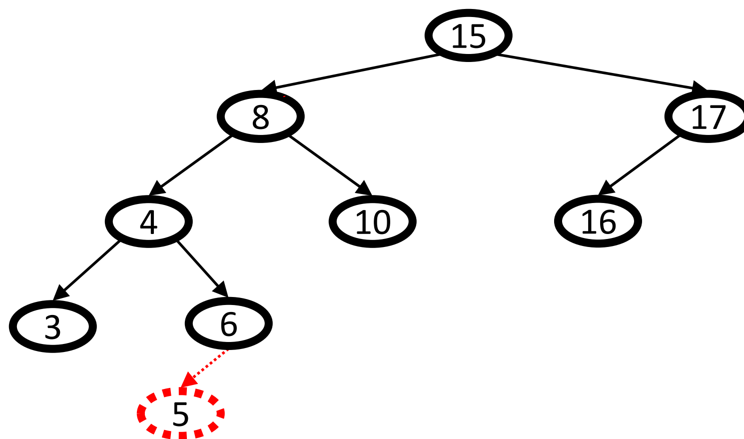
1. node's left-left grandchild is too tall
2. node's left-right grandchild is too tall
3. node's right-left grandchild is too tall
4. node's right-right grandchild is too tall

- *Only one case occurs* because tree was balanced before insert
- After the appropriate single or double rotation, the smallest-unbalanced subtree has the same height as before insertion
  - So all ancestors are now balanced

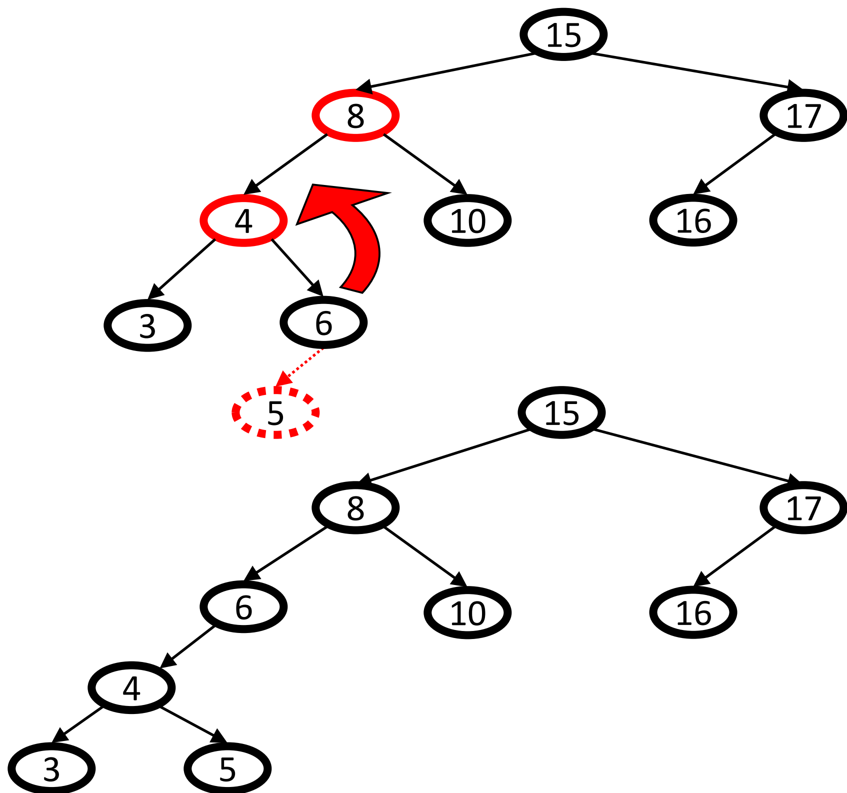
Lecture Outline

- AVL Tree
  - Add (cont): Double rotations
  - **Add exercises**
  - Remove
  - Wrapup
- Memory Hierarchy Basics
  - What is the Memory Hierarchy?
  - How does it impact data structure design?

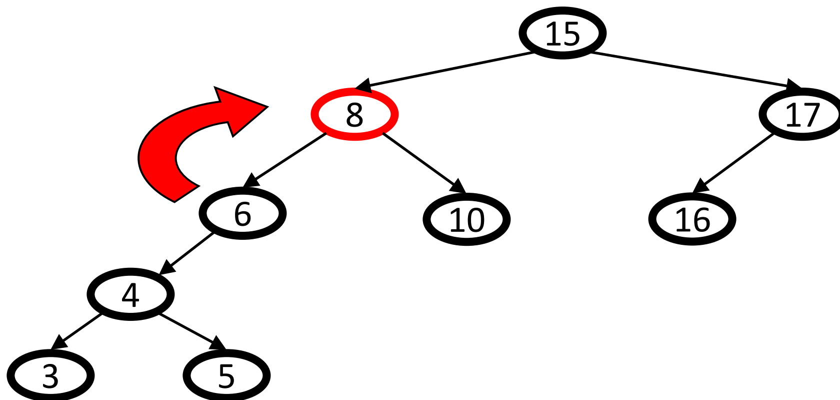
Double Rotation: Example (1 of 3)

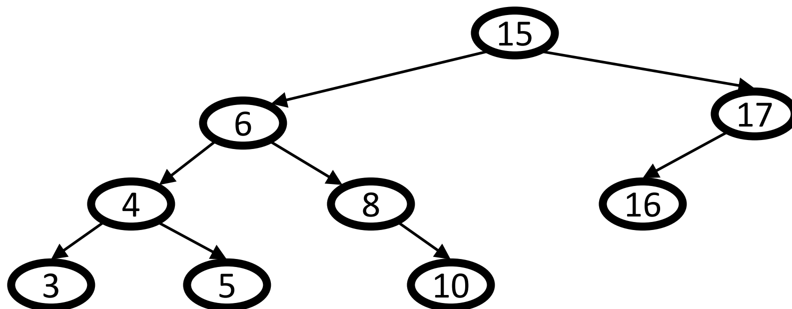


Double Rotation: Example (2 of 3)



Double Rotation: Example (3 of 3)



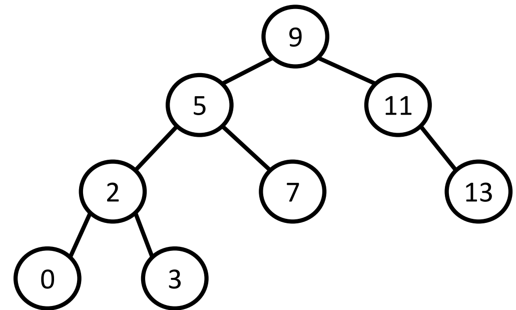


Student Activity #1: add() into an AVL tree

- add(a)
- add(b)
- add(e)
- add(c)
- add(d)

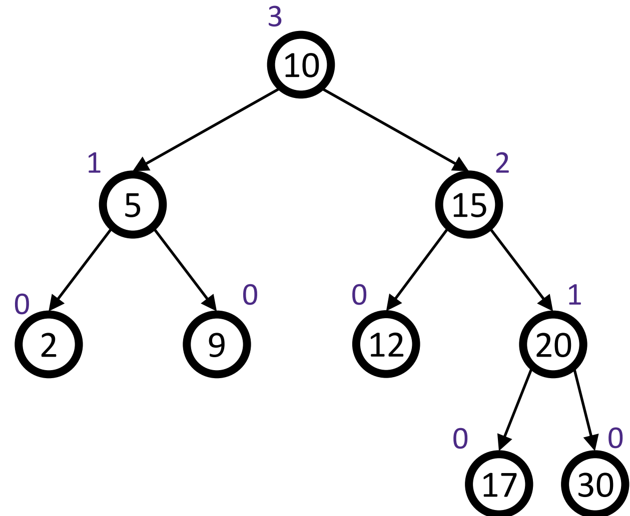
Student Activity #2: Single and Double Rotations

- Inserting which integer values would cause this tree to need a:
  - Single Rotation?
  - Double Rotation?
  - No Rotation?



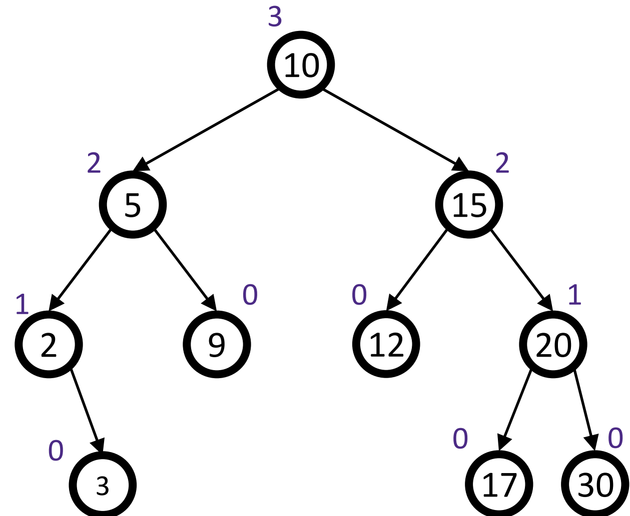
Student Activity #3: Add Sequence (1 of 2)

- Insert(3)
  - Is the resultant tree balanced?
  - If not, how would you fix it?



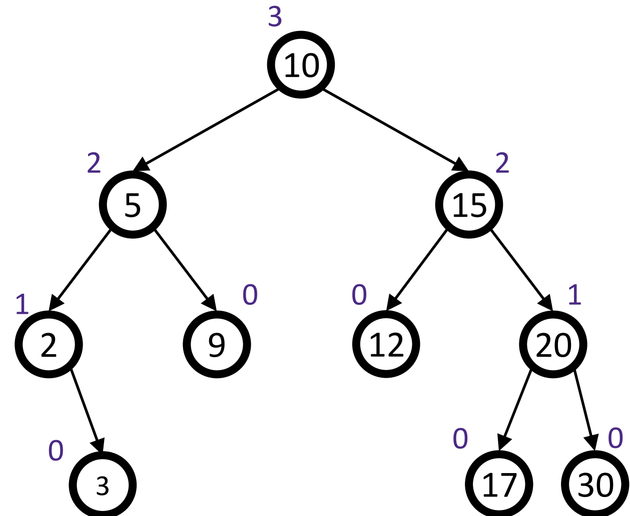
Student Activity #3: Add Sequence (2 of 2)

- Insert(33)
  - Is the resultant tree balanced?
  - If not, how would you fix it?



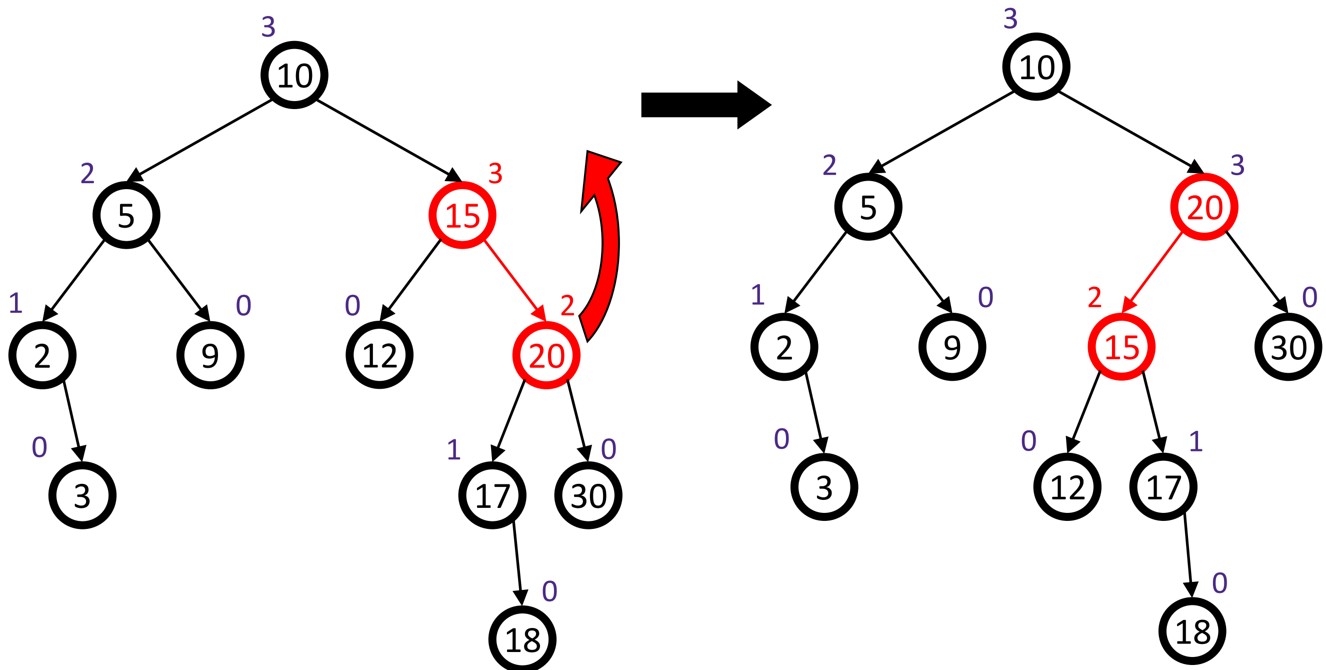
Student Activity #4: Harder Add Sequence (1 of 2)

- Insert(18)
  - Is the resultant tree balanced?
  - If not, how would you fix it?



Student Activity #4: Harder Add Sequence (2 of 2)

- Single Rotation doesn't work



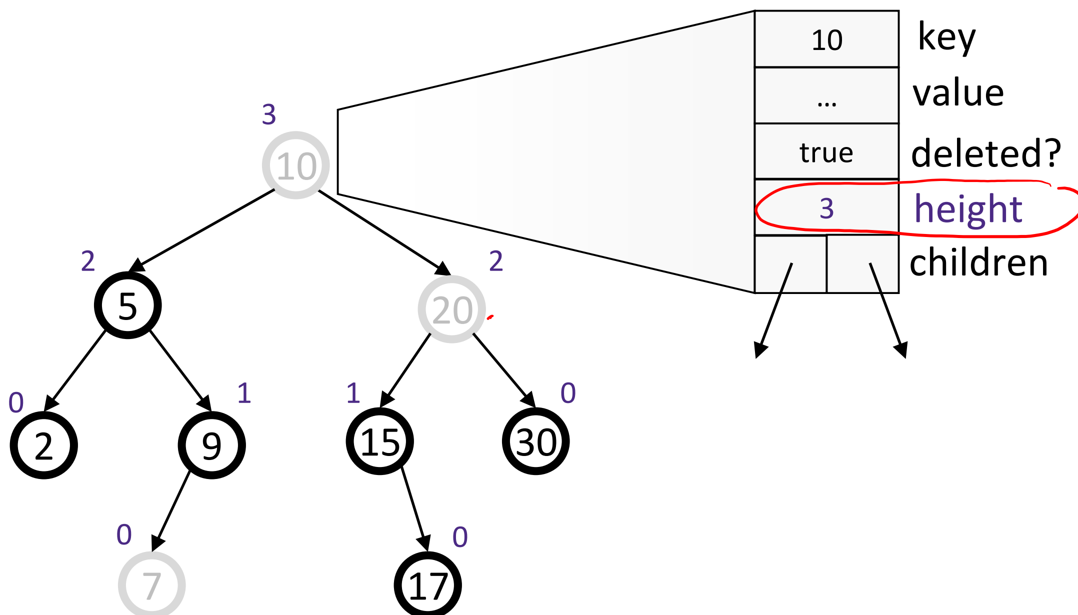
Lecture Outline

- AVL Tree
  - Add (cont): Double rotations
  - Add exercises
  - **Remove**
  - Wrapup

- Memory Hierarchy Basics
  - What is the Memory Hierarchy?
  - How does it impact data structure design?

AVL Remove

- The "easy way" is lazy deletion
  - Otherwise, we have several imbalance cases
  - See Weiss, 3rd ed. for more details



Lecture Outline

- AVL Tree
  - Add (cont): Double rotations
  - Add exercises
  - Remove
  - **Wrapup**
- Memory Hierarchy Basics
  - What is the Memory Hierarchy?
  - How does it impact data structure design?

AVL Tree Operations (1 of 2)

- AVL find:
  - Same as BST find
  - Worst-case complexity: O(log n)
    - Tree is balanced!
- AVL add:
  - First BST add, then check balance and potentially "fix" the AVL tree
  - Four different imbalance cases
  - Worst-case complexity: O(log n)
    - Tree starts and ends balanced
    - A rotation is O(1) and there's an O(log n) path to root
- AVL buildTree:
  - Worst-case complexity: O(n log n)

AVL Tree Operations (2 of 2)

- AVL remove
  - We suggest lazy deletion
    - Worst-case complexity: O(log n)
  - Deletion requires more rotations than insert; but worst-case complexity still O(log n)

Pros and Cons of AVL Trees

- Arguments for AVL trees:
  - All operations are logarithmic worst-case because trees are always balanced
  - Height rebalancing adds no more than a constant factor to the speed of add and remove

- Arguments against AVL trees:
  - Difficult to program and debug
  - Additional space for the height and deleted? fields
  - Asymptotically faster, but rebalancing takes time
  - Most large data sets require database-like systems on disk, and thus use other structures (e.g., B-trees, our next data structure)

Lecture Outline

- AVL Tree
  - Add (cont): Double rotations
  - Add exercises
  - Remove
  - Wrapup

- Memory Hierarchy Basics
  - **What is the Memory Hierarchy?**
  - How does it impact data structure design?

And Now for Something Completely Different…

- We have a simple and elegant data structure for the Dictionary ADT: the Binary Search Tree
  - But its worst-case behavior isn't great
- We can guarantee worst-case $O(\log n)$ with an AVL tree
  - … but at the cost of increased implementation complexity and space
  - One of several interesting/fantastic balanced-tree approaches!
- We will learn another balanced-tree approach: B-trees
  - It performs really well on large dictionaries (eg >1GB = 2^{30} bytes)
  - But to understand why, we need some ***memory-hierarchy basics***

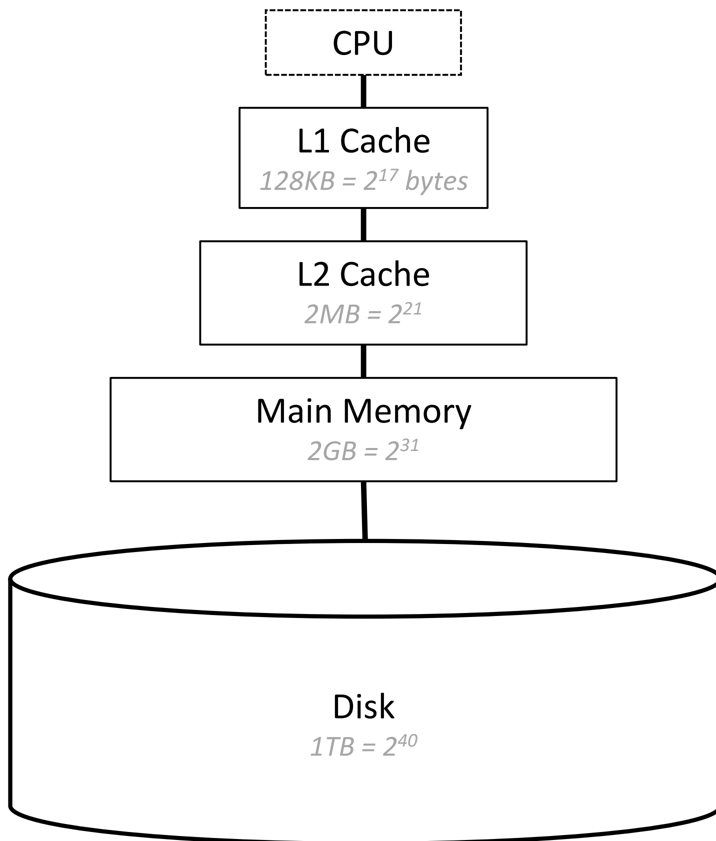
Why Does the Memory Hierarchy Matter?

- Asymptotic analysis supposedly helps us reason about large inputs. Why doesn't it work for B-trees?
  - We assumed "every memory access has an unimportant *O(1)* cost"
  - Learn more in CSE351/333/471; focus here on relevance to data structures and efficiency

```
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k) {
  for (int i=0; i < arr.length; ++i)
    if(arr[i] == k)
      return true;
  return false;
}
```

We claimed these two operations were approximately equal!

A Typical Memory Hierarchy



instructions (e.g., addition): 2^{30}/sec

fetch data in L1: 2^{29}/sec = 2 instructions

fetch data in L2: 2^{25}/sec = 30 instructions

fetch data in main memory: 2^{22}/sec = 250 instructions

fetch data from "new place" on disk: 2^{7}/sec = 8,000,000 instructions

Said In Another Way …

- Jeff Dean's "Numbers Everyone Should Know" (LADIS '09)

Numbers Everyone Should Know

Analogy	Operation	Time
Sticky note on monitor	L1 cache reference	0.5 ns
	Branch mispredict	5 ns
Yelling for your roommate	L2 cache reference	7 ns
	Mutex lock/unlock	100 ns
Flipping through textbook	Main memory reference	100 ns
	Compress 1K bytes with Zippy	10,000 ns
	Send 2K bytes over 1 Gbps network	20,000 ns
	Read 1 MB sequentially from memory	250,000 ns
	Round trip within same datacenter	500,000 ns
Retaking 311 and then retaking 332	Disk seek	10,000,000 ns
	Read 1 MB sequentially from network	10,000,000 ns
	Read 1 MB sequentially from disk	30,000,000 ns
	Send packet CA->Netherlands->CA	150,000,000 ns

Google

Memory Hierarchy: Result

It is much faster to do …	*Than …*
5 million arithmetic ops	1 hard disk access
2500 L2 cache accesses	1 hard disk access
400 main memory accesses	1 hard disk access

- Why are computers built this way?
  - Physical realities (speed of light, closeness to CPU)
  - Cost (price per byte of different technologies)
  - Hard disks get much bigger not much faster
    - Spinning at 7200 RPM accounts for much of the slowness and unlikely to spin faster in the future
  - Speedup at higher levels (e.g. a faster processor) makes lower levels relatively slower
  - Later in the course: more than 1 CPU!

Hardware and OS Support

- The hardware and OS work together to automatically move data into and out of successive levels for you!
  - Replace items currently in memory/L2/L1
  - Data structures and algorithms faster if "fits in cache" (it often does)

- Most code "just works" most of the time
  - … but sometimes designing data structures and algorithms with knowledge of memory hierarchy is worth it
  - And when you do, you often need to know one more thing …

How Data Moves Around the Hierarchy

Spatial Locality

- Hardware/OS often fetches a chunk of data instead of a byte
  - Moving data up the hierarchy is slow because of the *lower level's latency* (think: distance-to-travel)
  - However, the latency is the same regardless if your program requests one byte or one chunk (think: carpool)
  - So a single fetch often causes the hardware/OS to send *nearby memory* because it's easy and likely to be asked for soon (think: object fields or arrays)

Temporal Locality

- Once data has moved up the hierarchy, keep it around
  - A *particular piece of data* is more likely to be accessed again in the near future than some random other piece of data

Locality Principles, in Detail

- **Spatial Locality** (locality in space)
  - If an address is referenced, ***addresses that are close by*** tend to be referenced soon

- **Temporal Locality** (locality in time)
  - If an address is referenced, ***that same address*** tends to be referenced again soon

Lecture Outline

- AVL Tree
  - Add (cont): Double rotations
  - Add exercises
  - Remove
  - Wrapup
- Memory Hierarchy Basics
  - What is the Memory Hierarchy?
  - **How does it impact data structure design?**

Spatial Locality: Arrays vs. Linked Lists (1 of 3)

- Which has the potential to take advantage of **spatial locality**?

- Terminology:
  - The amount of data moved from **disk** into **memory** is called the "block" size or the "page" size
  - The amount of data moved from **memory** into **cache** is called the cache "line" size

- Reminder:
  - Neither the movement nor the sizes are under programmer control!

Spatial Locality: Arrays vs. Linked Lists (2 of 3)

- An array benefits more than a linked list from spatial locality
  - Language (e.g., Java) implementation can put LL nodes anywhere, whereas an array is typically implemented as contiguous memory
  - Contiguous memory benefits from spatial locality

- Suppose 2^{23} items of 2^{7} bytes each. They are stored on disk and the block size is 2^{10} bytes
  - An **array** needs 2^{20} disk accesses
    - If "perfectly streamed", > 4 seconds
    - If "random places on disk", 8000 seconds (> 2 hours)
  - A **linked list** *in the worst case* needs 2^{23} disk accesses
    - Assuming "random" placement around disk, >16 hours

Spatial Locality: Arrays vs. Linked Lists (3 of 3)

- However! "Array" doesn't necessarily mean "good"
  - Binary heaps "make big jumps" to percolate
  - Constantly loading/unloading different blocks from disk

What About BSTs? (1 of 2)

- Operations on balanced BSTs are $O(\log n)$
  - Even for $n = 2^{39}$ (512 GB), we "should be ok"
- Still, number of disk accesses matters:
  - Pretend for a minute we had an AVL tree of height 55
  - The total number of nodes could be: __________
  - Most of the nodes will be on disk: the tree is shallow, but it is still many gigabytes big so the entire *tree* cannot fit in memory
    - Even if memory holds the first 25 nodes on our path, we still potentially need 30 disk accesses if we are traversing the entire height of the tree.

What about BSTs? (2 of 2)

If your data structure is mostly on disk, minimize disk accesses!

- In this scenario, a better data structure would exploit the block size and (relatively) fast memory access to ***avoid disk accesses***

Our B-Tree Goal

- **Problem**: A dictionary with so many items data *most of it is on disk*

- **Desire**: A balanced tree (logarithmic height) that minimizes disk accesses and exploits disk-block size

- **A key idea**: Increase the branching factor of our tree