CSE332: Data Abstractions

Lecture 8: AVL Delete; Memory Hierarchy

Dan Grossman

Spring 2012

The AVL Tree Data Structure

Structural properties

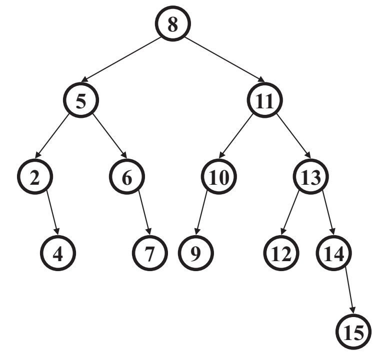
1. Binary tree property
2. Balance property: balance of every node is between -1 and 1

Result:

Worst-case depth is $O(\log n)$

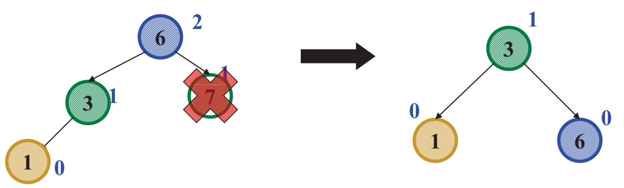
Ordering property

– Same as for BST



AVL Tree Deletion

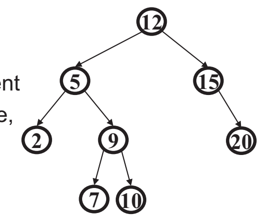
- Similar to insertion: do the delete and then rebalance
  - Rotations and double rotations
  - Imbalance may propagate upward so rotations at multiple nodes along path to root may be needed (unlike with insert)
- Simple example: a deletion on the right causes the left-left grandchild to be too tall
  - Call this the *left-left case*, despite deletion on the *right*
  - `insert(6) insert(3) insert(7) insert(1) delete(7)`



Properties of BST delete

We first do the normal BST deletion:

- 0 children: just delete it
- 1 child: delete it, connect child to parent
- 2 children: put successor in your place, delete successor leaf



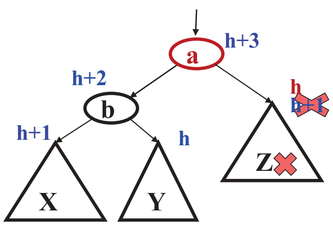
Which nodes' heights may have changed:

- 0 children: path from deleted node to root
- 1 child: path from deleted node to root
- 2 children: path from *deleted successor leaf* to root

Will rebalance as we return along the "path in question" to the root

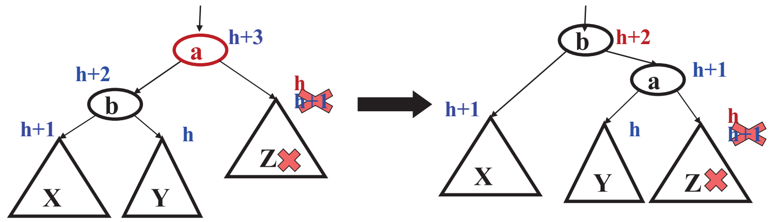
Case #1 Left-left due to right deletion

- Start with some subtree where if right child becomes shorter we are unbalanced due to height of left-left grandchild



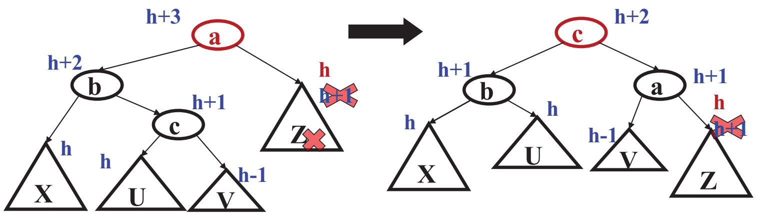
- A delete in the right child could cause this right-side shortening

Case #1: Left-left due to right deletion



- Same single rotation as when an insert in the left-left grandchild caused imbalance due to X becoming taller
- But here the "height" at the top decreases, so more rebalancing farther up the tree might still be necessary

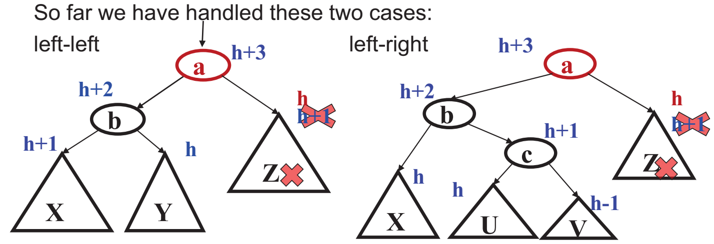
Case #2: Left-right due to right deletion



- Same double rotation when an insert in the left-right grandchild caused imbalance due to c becoming taller
- But here the "height" at the top decreases, so more rebalancing farther up the tree might still be necessary

No third right-deletion case needed

So far we have handled these two cases:



But what if the two left grandchildren are now *both* too tall (h+1)?

- Then it turns out left-left solution still works
- The children of the "new top node" will have heights differing by 1 instead of 0, but that's fine

And the other half

- Naturally two more mirror-image cases (not shown here)
  - Deletion in left causes right-right grandchild to be too tall
  - Deletion in left causes right-left grandchild to be too tall
  - (Deletion in left causes both right grandchildren to be too tall, in which case the right-right solution still works)
- And, remember, "lazy deletion" is a lot simpler and might suffice for your needs

Pros and Cons of AVL Trees

Arguments for AVL trees:

1. All operations logarithmic worst-case because trees are *always* balanced
2. Height balancing adds no more than a constant factor to the speed of `insert` and `delete`

Arguments against AVL trees:

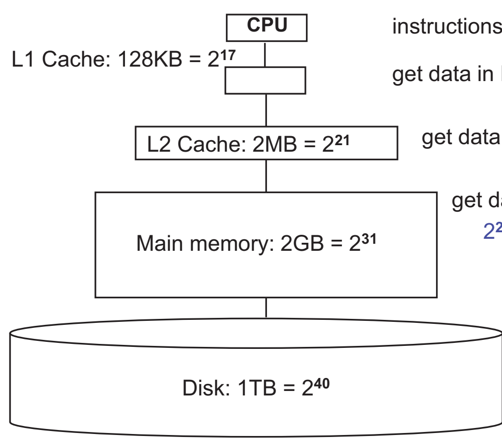
1. Difficult to program & debug
2. More space for height field
3. Asymptotically faster but rebalancing takes a little time
4. Most large searches are done in database-like systems on disk and use other structures (e.g., B-trees, our next data structure)
5. If *amortized* (later, I promise) logarithmic time is enough, use splay trees (skipping, see text)

Now what?

- Have a data structure for the dictionary ADT that has worst-case *O*(`log` *n*) behavior
  - One of several interesting/fantastic balanced-tree approaches
- About to learn another balanced-tree approach: B Trees
- First, to motivate why B trees are better for really large dictionaries (say, over 1GB = 2^{30} bytes), need to understand some ***memory-hierarchy basics***
  - Don't always assume "every memory access has an unimportant *O(1)* cost"
  - Learn more in CSE351/333/471, focus here on relevance to data structures and efficiency

A typical hierarchy

Every desktop/laptop/server is different but here is a plausible configuration these days



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

get data in L1: 2^{29}/sec = 2 insns

get data in L2: 2^{25}/sec = 30 insns

get data in main memory: 2^{22}/sec = 250 insns

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

"streamed": 2^{18}/sec

Morals

It is much faster to do: Than:

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

Why are computers built this way?
- Physical realities (speed of light, closeness to CPU)
- Cost (price per byte of different technologies)
- 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 makes lower levels *relatively slower*

"Fuggedaboutit", usually

The hardware automatically moves data into the caches from main memory for you
- Replacing items already there
- So algorithms much faster if "data fits in cache" (often does)

Disk accesses are done by software (e.g., ask operating system to open a file or database to access some data)

So most code "just runs" but sometimes it's worth designing algorithms / data structures with knowledge of memory hierarchy
- And when you do, you often need to know one more thing…

Block/line size

- Moving data up the memory hierarchy is slow because of *latency* (think distance-to-travel)
  - May as well send more than just the one int/reference asked for (think "giving friends a car ride doesn't slow you down")
  - Sends nearby memory because:
    - It is easy
    - Likely to be used soon (think fields/arrays)

Principle of *Locality*

- Amount of data moved from disk into memory called the "block" size or the "page" size
  - Not under program control
- Amount of data moved from memory into cache called the "line" size
  - Not under program control

Connection to data structures

- An array benefits more than a linked list from block moves
  - Language (e.g., Java) implementation can put the list nodes anywhere, whereas array is typically contiguous memory
- Suppose you have a queue to process with 2^{23} items of 2^7 bytes each on disk and the block size is 2^{10} bytes
  - An array implementation needs 2^{20} disk accesses
  - If "perfectly streamed", > 4 seconds
  - If "random places on disk", 8000 seconds (> 2 hours)
  - A list implementation in the worst case needs 2^{23} "random" disk accesses (> 16 hours) – probably not that bad
- Note: "array" doesn't mean "good"
  - Binary heaps "make big jumps" to percolate (different block)

BSTs?

- Looking things up in balanced binary search trees is $O($`log` $n)$, so even for $n = 2^{39}$ (512GB) we need not worry about minutes or hours
- Still, number of disk accesses matters
  - AVL tree could have height of 55 (see lecture7.xlsx)
  - So each **`find`** could take about 0.5 seconds or about 100 finds a minute
  - Most of the nodes will be on disk: the tree is shallow, but it is still many gigabytes big so the *tree* cannot fit in memory
    - Even if memory holds the first 25 nodes on our path, we still need 30 disk accesses

Note about numbers; moral

- All the numbers in this lecture are "ballpark" "back of the envelope" figures
- Even if they are off by, say, a factor of 5, the moral is the same: If your data structure is mostly on disk, you want to minimize disk accesses
- A better data structure in this setting would exploit the block size and relatively fast memory access to avoid disk accesses…