Programming Abstractions

CS106X

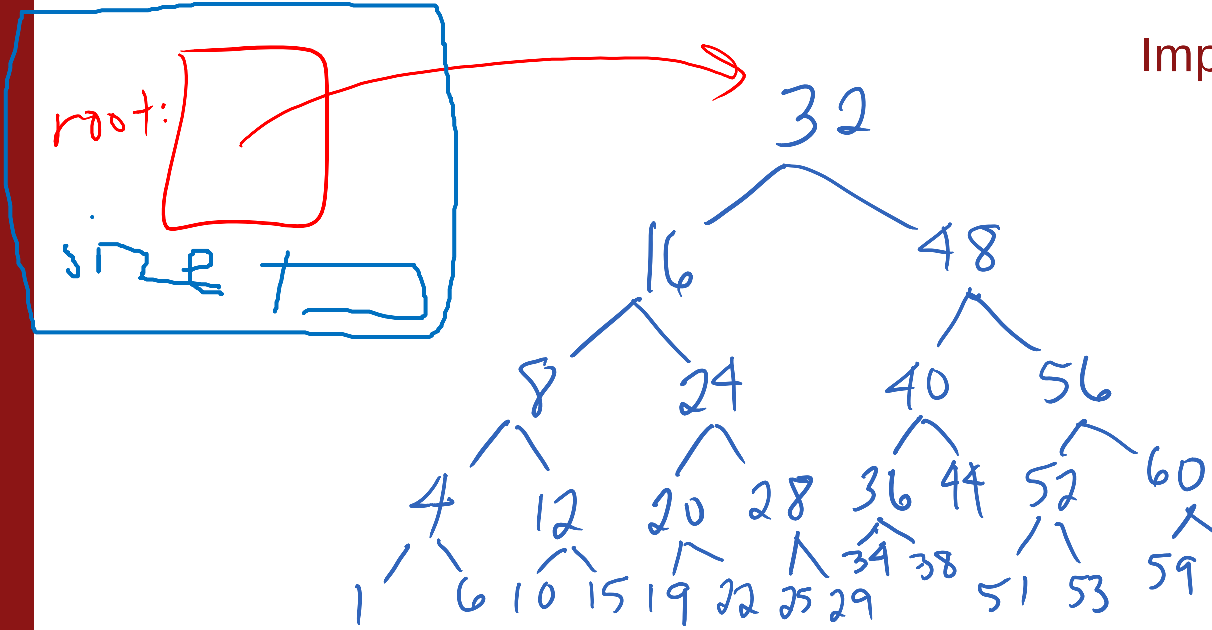
Cynthia Lee

Stanford University

Topics:

- Binary Search Tree (BST)
  - Starting with a dream: binary search in a linked list?
  - How our dream provided the inspiration for the BST
    - Note: we do NOT actually construct BSTs using this method
  - BST insert
  - Big-O analysis of BST
  - BST balance issues
- Traversals
  - Pre-order
  - In-order
  - Post-order
  - Breadth-first
- Applications of Traversals

An Idealized Binary Search Tree



Important note of clarification:

When I was talking about setting up the tree as a binary search (pic at left), that was an explanation of the *inspiration* for BST. Lining up the values and then arranging the pointers all at once is not how we use them (insert one at a time using algorithm we talked about).

What is the worst case cost for doing containsKey() in BST ***if the BST is balanced?***

O(logN)—awesome!

BSTs is that they are great when balanced

BST is bad when unbalanced

Balance depends on order of insert of elements

Ok, so, long-chain BSTs are bad, should we worry about it? [math puzzle time]

One way to create a bad BST is to insert the elements in *decreasing* order: 34, 22, 9, 3

That's not the only way…

How many **distinctly structured** BSTs are there that exhibit the worst case height (height equals number of nodes) for a tree with the 4 nodes listed above?

A. 2-3
B. 4-5
C. 6-7
D. 8-9
E. More than 9

Bonus question: general formula for any BST of size n?

Extra bonus question (CS109): what is this as a fraction of all trees (i.e., probability of worst-case tree).

BST Balance Strategies

So we definitely need to balance, how can we do that if the tree location is fixed when we insert?

Red-Black trees

- One of the most famous (and most tricky) strategies for keeping a BST balanced
- Not guaranteed to be perfectly balanced, but “close enough” to keep O(log n) *guarantee* on operations

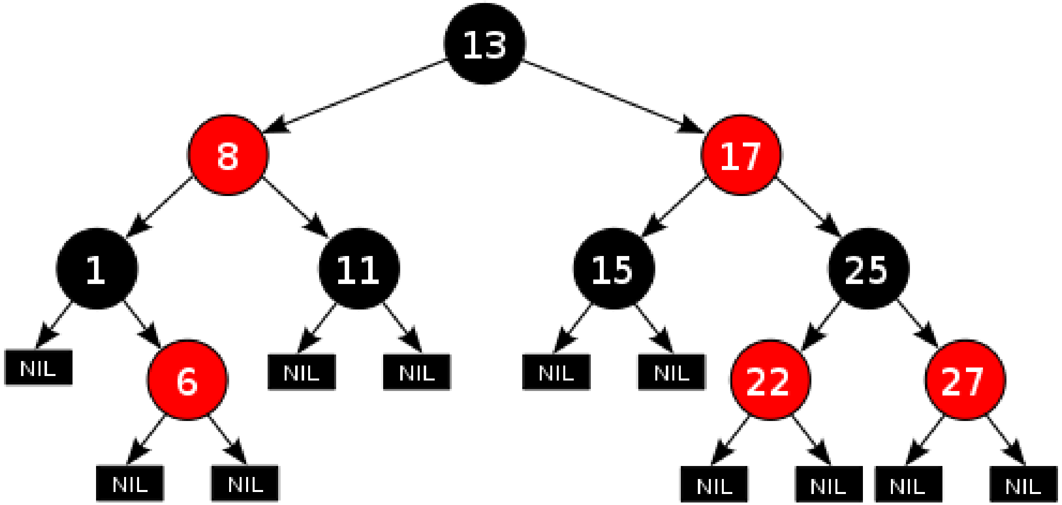
Red-Black trees

In addition to the requirements imposed on a binary search trees, red–black trees must meet these:

- A node is either red or black.
- The root is black.
- All leaves (null children) are black.
- Both children of every red node are black.
- **Every simple path from a given node to any of its descendant leaves contains the same number of black nodes.**
  - (This is what guarantees “close” to balance)

Video: http://www.youtube.com/watch?v=vDHFF4wjWYU

Red-Black trees



Every simple path from a given node to any of its descendant leaves contains the same number of black nodes.

- (This is what guarantees "close" to balance)

Other BST balance strategies

Red-Black tree

AVL tree

Treap (BST + heap in one tree! What could be cooler than that, amirite? ♥ ♥ ♥)

Other fun types of **BST**:

Splay tree

B-Tree

Other fun types of **BST**

Splay tree

- Rather than only worrying about balance, Splay Tree dynamically readjusts based on **how often users search for an item**. Most commonly-searched items move to the top, saving time
  - For search terms, imagine "Bieber" would be near the root, and "polymorphism" would be further down by the leaves, because humanity is disappointing sometimes…

B-Tree

- Like BST, but a node can have many children, not just 2
- Used for huge databases

BST and Heap quick recap/cheat sheet

BST and Heap quick recap/cheat sheet

Heap (Priority Queue)

- Structure: must be "complete"
- Order: parent priority must be <= both children
  - This is for min-heap, opposite is true for max-heap
  - No rule about whether left child is > or < the right child
- Big-O: guaranteed log(n) enqueue and dequeue
- Operations: always add to end of array and then "bubble up"; for dequeue do "trickle down"

BST (Map)

- Structure: any valid binary tree
- Order: leftchild.key < self.key < rightchild.key
  - No duplicate keys
  - Because it's a Map, values go along for the ride w/keys
- Big-O: log(n) if balanced, but might not be balanced, then linear
- Operations: recursively repeat: start at root and go left if key < root, go right if key > root

Tree Traversals!

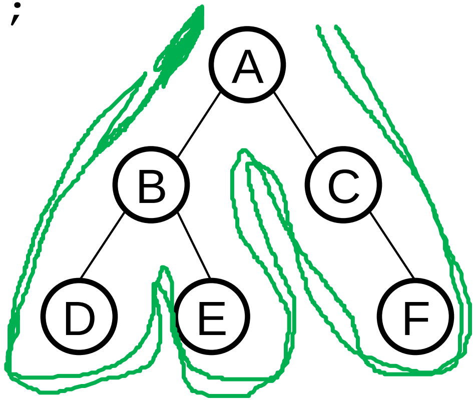
These are not only for Binary Search Trees, but we often do them on BSTs

What does this print?

(assume we call traverse on the root node to start)

pre-order traversal

```
void traverse(Node *node) {
  if (node != NULL) {
     cout << node->key << " ";
     traverse(node->left);
     traverse(node->right);
  }
}
```



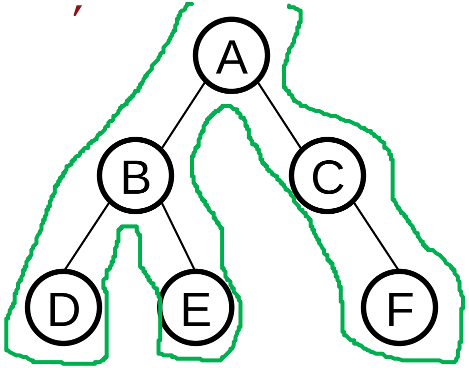
A. A B C D E F
B. A B D E C F
C. D B E F C A
D. D E B F C A
E. Other/none/more

What does this print?

(assume we call traverse on the root node to start)

post-order traversal

```
void traverse(Node *node) {
  if (node != NULL) {
     traverse(node->left);
     traverse(node->right);
     cout << node->key << " ";
  }
}
```



A. A B C D E F
B. A B D E C F
C. D B E F C A
D. D E B F C A
E. Other/none/more

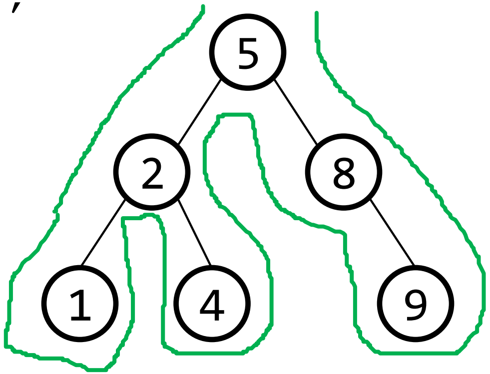
What does this print?

(assume we call traverse on the root node to start)

in-order traversal

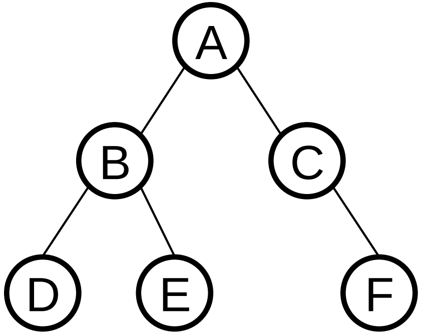
```
void traverse(Node *node) {
   if (node != NULL) {
      traverse(node->left);
      cout << node->key << " ";
      traverse(node->right);
   }
}
```

A. 1 2 4 5 8 9
B. 1 4 2 9 8 5
C. 5 2 1 4 8 9
D. 5 2 8 1 4 9
E. Other/none/more



How can we get code to print our ABCs in order as shown? (note: not BST order)

```
void traverse(Node *node) {
  if (node != NULL) {
 ??   cout << node->key << " ";
      traverse(node->left);
      traverse(node->right);
  }
}
```



You can't do it by using this code and moving around the cout—we already tried moving the cout to all 3 possible places and it didn't print in order

- You can but you use a **queue** instead of recursion
- **"Breadth-first"** search
- Again we see this key theme

Applications of Tree Traversals

Beautiful little things from an algorithms/theory standpoint, but they have a practical side too!

Traversals a very commonly-used tool in your CS toolkit

```
void traverse(Node *node) {
  if (node != NULL) {
     traverse(node->left);
     // “do something”
     traverse(node->right);
  }
}
```

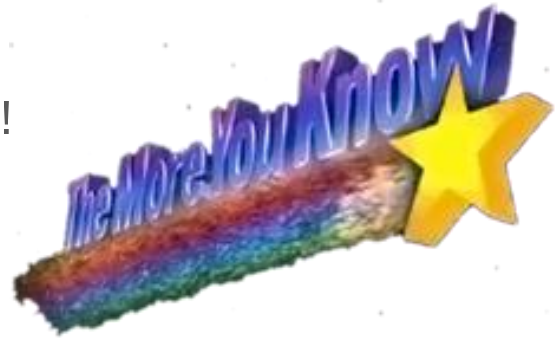
- Customize and move the “do something,” and that’s the basis for dozens of algorithms and applications

Map interface implemented with BST

- Remember how when you iterate over the Stanford library Map you get the keys in sorted order?
  - (we used this for the word occurrence counting code example in class)

```
void printMap(const Map<string, int>& themap) {
    for (string s : themap) {
        cout << s; // printed in sorted order
    }
}
```

- Now you know why it can do that in O(N) time!
  - “In-order” traversal



Applications of the traversals

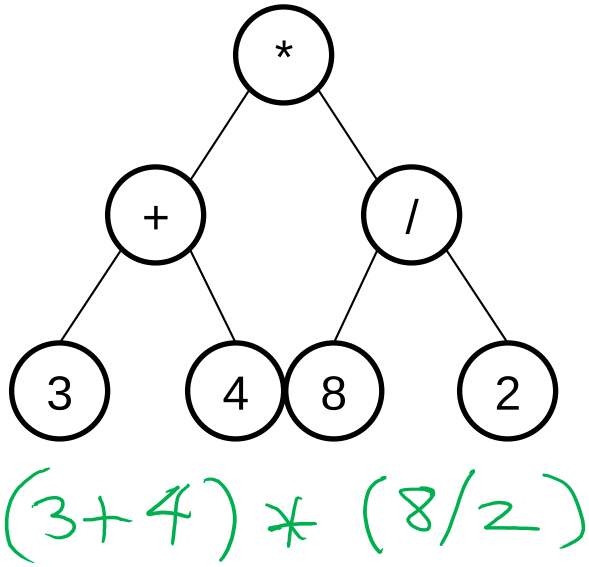
- You have a tree that represents **evaluation of an arithmetic expression**. Which traversal would form the foundation of your evaluation algorithm?

A. Pre-order

B. In-order

C. Post-order

D. Breadth-first



Applications of the traversals

- You are writing the **destructor** for a BST class. Given a pointer to the root, it needs to free each node. Which traversal would form the foundation of your destructor algorithm?

  A. Pre-order

  B. In-order

  C. Post-order

  D. Breadth-first