Programming Abstractions

CS106B

Cynthia Lee

Topics:

Wednesday:

- 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

Today:

- Binary Search Tree (BST)
  - BST balance issues
- Traversals
  - Pre-order
  - In-order
  - Post-order
  - Breadth-first
- Applications of Traversals

BST Balance Strategies

We need to balance the tree (keep O(logN) instead of O(N)), how can we do that if the tree structure is decided by key insert order?

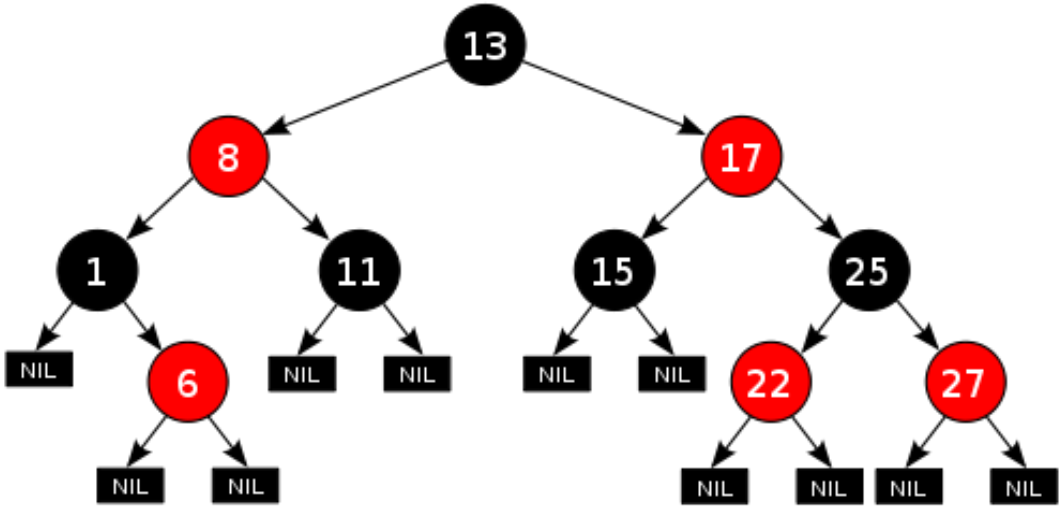
Red-Black trees

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

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

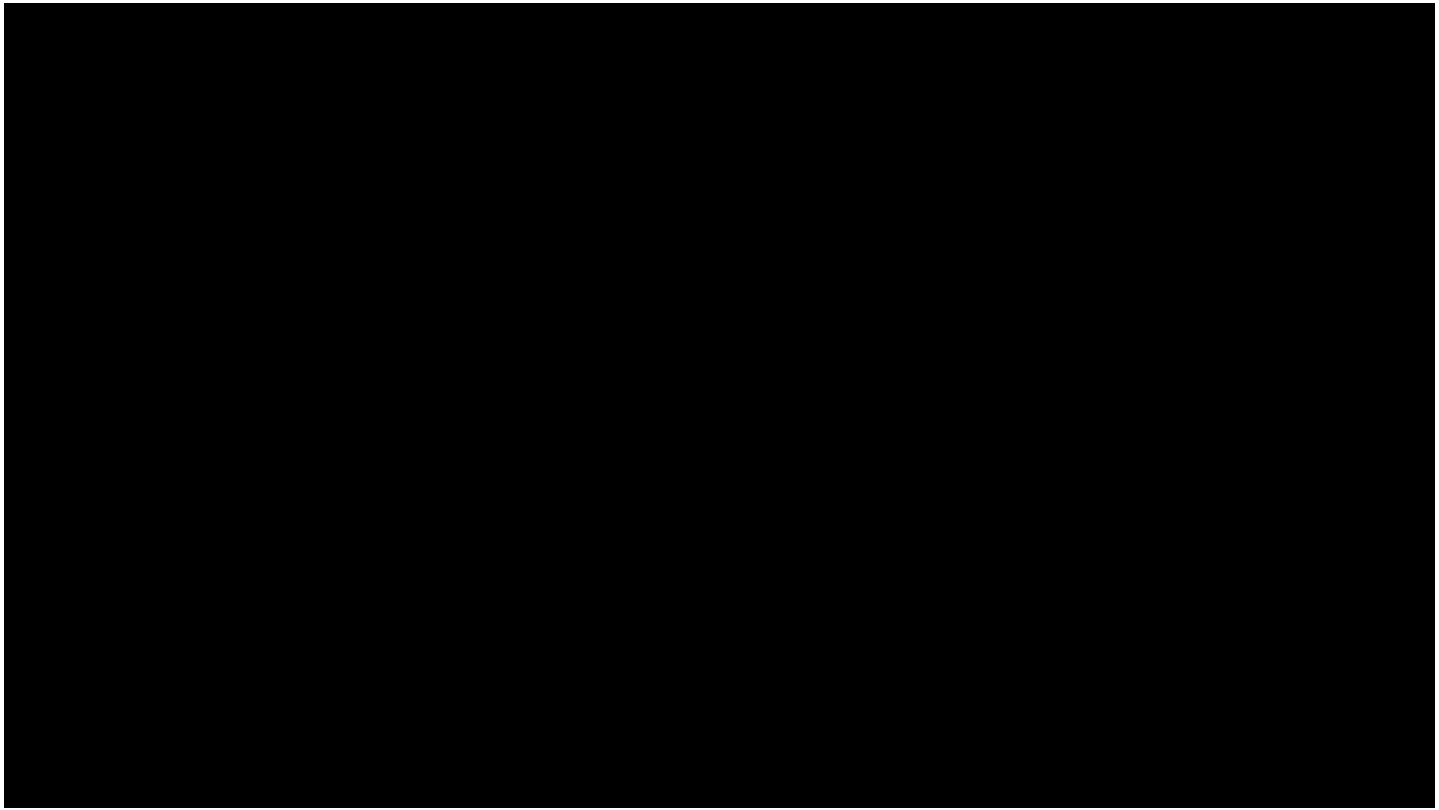
Red-Black trees



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)

Red-Black trees insert



A few BST balance strategies

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

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
  - Example: if Google did this, "Bieber" would be near the root, and "splay tree" would be further down by the leaves

B-Tree

- Like BST, but a node can have many children, not just two
- More branching means an even "flatter" (smaller height) tree
- Used for huge databases

BST and Heap quick recap/cheat sheet

BST and Heap Facts (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 O(n)
- Operations: recursively repeat: start at root and go left if key < root, go right if key > root

Tree Traversals!

These are for any binary trees, but we often do them on BSTs

What does this print?

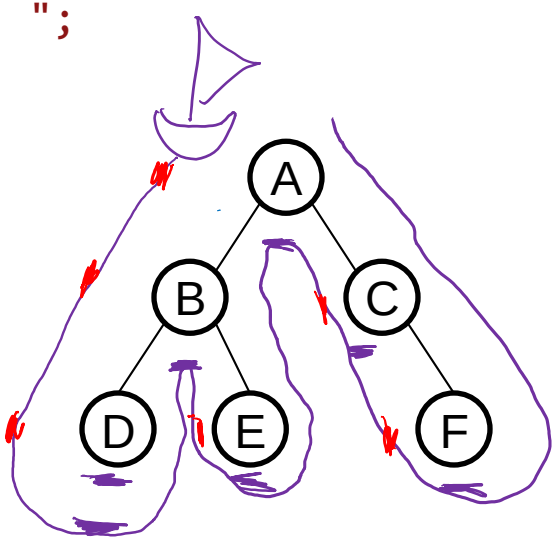
(assume we call traverse on the root node to start)

pre-order traversal

A B D E C F

```
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



D B E A C F

What does this print?

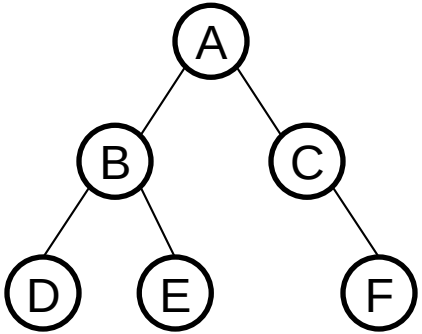
(assume we call traverse on the root node to start)

post-order traversal

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

DEBFCA

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?

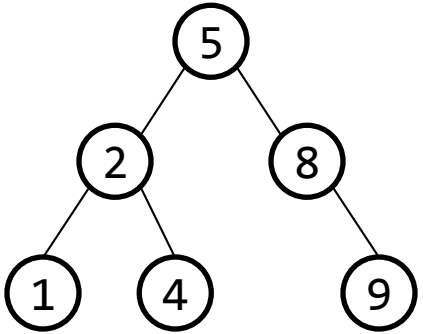
(assume we call traverse on the root node to start)

in-order traversal

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

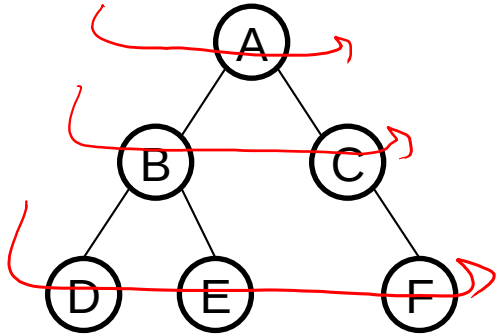
1 2 4 5 8 9

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 of BFS vs DFS!*

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);
  }
}
```

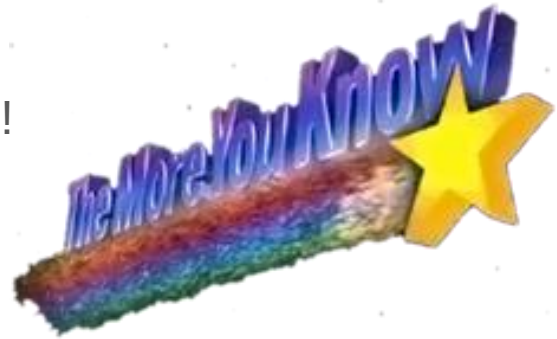
- Customize and move the “do something,” and that’s the basis for dozens of algorithms and applications

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 << endl; // printed in sorted order
    }
}
```

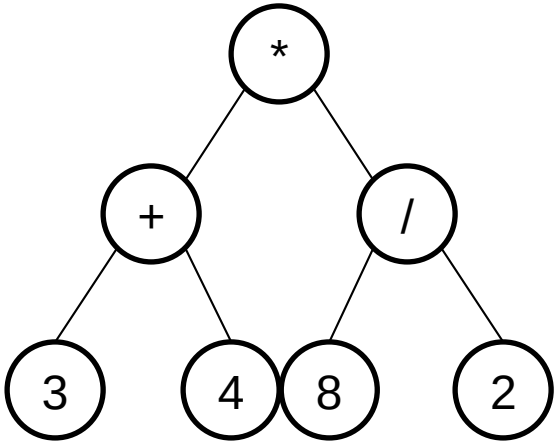
- Now you know why it can do that in O(N) time!
  - "In-order" traversal



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~~



(3 + 4) * (8/2)

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