Announcements

• Performance bug alert: in `merge()`, don’t make a copy of the whole array if you’re only merging part of it!

• Quiz grades will be released via GradeScope

• How’s A1 going?
Goals (Today and Wednesday):

• Know the definition and uses of a binary search tree.

• Be prepared to implement, and know the runtime of, the following BST operations:
  • searching
  • inserting
  • deleting

• Know what a balanced BST is and why we want it.
Tree Terminology: Lighting Round!

ABCD:
What’s the root of G’s right subtree?

What’s an ancestor of F?

What’s C’s parent?

What’s a node at depth 1?

What’s a node at the root of a subtree of height 0?
public class Tree {
    int value;
    Tree parent;
    Tree left;
    Tree right;
}
/** BST: a binary tree, in which:
 * - all values in left are < value
 * - all values in right are > value
 * - left and right are BSTs */

public class BST {
    int value;
    BST parent;
    BST left;
    BST right;
}
Binary Search Tree

- Node 2 with values 0 and 3
- Node 5
- Node 8 with values 7 and 9

Key: < 5, > 5
ABCD: Which of these is **not** a binary search tree?
Write the values printed by an **in-order** traversal of each of the following BSTs:
Searching a Binary Tree

- A **binary tree** is
  - Empty, or
  - Three things:
    - value
    - a left **binary tree**
    - a right **binary tree**

Find v in a binary tree:

```java
boolean findVal(Tree t, int v):
  if t == null:
    return false
  if t.value == v:
    return true
  return findVal(t.left) || findVal(t.right)
```

(not BST!)
Searching a BST

search(t, 11)
11 > 10
search(right, 11)
Searching a BST

\[ t: \begin{array}{c}
10 \\
8 \\
4 \\
9 \\
11 \\
16 \\
17
\end{array} \]

\begin{align*}
\text{search}(t, 11) &\quad 11 > 10 \\
\text{search(right, 11)} &\quad 11 < 16 \\
\text{search(left, 11)}
\end{align*}
Searching a BST

$t: \quad 10$

search(t, 11)

11 > 10

search(right, 11)

11 < 16

search(left, 11)

11 < 16

found it! return.
Searching a BST - the nonexistent case

t: 10

8
4 9

16
11 17

search(t, 5)
5 < 10
search(left, 5)
Searching a BST - the nonexistent case

```
search(t, 5)
5 < 10
search(left, 5)
5 < 8
search(left, 5)
```
Searching a BST - the nonexistent case

```
t:          search(t, 5)
   10
  /  \                     5 < 10
 8    16                   search(left, 5)
 /    /                    5 < 8
4    9                    search(left, 5)
     /                     5 > 4
    11                    search(right, 5)
      \                     null - not found!
     17
```

null - not found!
Searching: BT vs BST

Compare binary tree to binary search tree:

```java
boolean searchBT(n, v):
    if n==null, return false
    if n.v == v, return true
    return searchBST(n.left, v)
    || searchBST(n.right, v)

2 recursive calls
```

```java
boolean searchBST(n, v):
    if n==null, return false
    if n.v == v, return true
    if v < n.v
        return searchBST(n.left, v)
    else
        return searchBST(n.right, v)

1 recursive call
```
Searching a BST: What’s the runtime?

```java
boolean search(BST t, int v):
    if t == null:
        return false
    if t.value == v:
        return true
    if v < t.value:
        return search(t.left)
    else:
        return search(t.right)
```

If h is the tree’s **height**, search can visit at most h+1 nodes!

Runtime of search is O(h).

*That’s great, but how does h relate to n, the number of nodes?*
How many nodes does a tree with height $h$ have?

Consider $h = 2$:

**Fewest possible:**
- $n = h + 1$
- $n$ is $O(h)$
- $h$ is $O(n)$

**Most possible:**
- At depth $d$: $2^d$ nodes possible.
- At all depths: $2^0 + 2^1 + \ldots + 2^h$
- $n = 2^{h+1} - 1$
- $n$ is $O(2^h)$
- $h$ is $O(\log n)$
Searching a BST: What’s the runtime?

```java
boolean search(BST t, int v):
    if t == null:
        return false
    if t.value == v:
        return true
    if t.value < v:
        return search(t.left)
    else:
        return search(t.right)
```

Runtime of search is $O(h)$.  
Worst: $O(n)$  
Best: $O(\log n)$  

We want our trees to look more like this than this.
Inserting into a BST
Inserting into a BST

\[ t: \begin{array}{c}
10 \\
8 \\
4 \\
9 \\
11 \\
17 \\
16 \\
\end{array} \]

\[ \text{insert}(t, 11) \]
\[ 11 > 10 \]
\[ \text{insert}(\text{right}, 11) \]
Inserting into a BST

$t$: 10

- $11 > 10$
  - insert(right, 11)

- $11 < 16$
  - insert(left, 11)
Inserting into a BST

```
t: 10
   8   16
  4  9  11  17
insert(t, 11)
11 > 10
insert(right, 11)
11 < 16
insert(left, 11)
11 < 16
found it! no duplicates, allowed; nothing to do.
return.
```
Inserting into a BST - the nonexistent case

```
insert(t, 5)
5 < 10
insert(left, 5)
```
Inserting into a BST - the nonexistent case

\[ t: 10 \]

\[ \text{insert}(t, 5) \]
\[ 5 < 10 \]
\[ \text{insert}(\text{left}, 5) \]
\[ 5 < 8 \]
\[ \text{insert}(\text{left}, 5) \]
Inserting into a BST - the nonexistent case

\[
t: \quad 10
\]

- \(5 < 10\)
- \(5 < 8\)
- \(5 > 4\)

null - not found. Insert it here!
Let’s Build Some Trees

```java
// First tree:
BST t = new BST();
t.insert(10);
t.insert(15);
t.insert(16);
t.insert(8);
t.insert(16);
t.insert(9);
t.insert(11);
t.insert(-1);
```

```java
// Second tree:
BST t = new BST();
t.insert(-1);
t.insert(8);
t.insert(9);
t.insert(11);
t.insert(10);
t.insert(15);
t.insert(16);
t.insert(16);
```
Let’s Build Some Trees

Insertion order matters. We can’t always control it.
Deleting a node from a BST

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children