ADTs, Asymptotics II, BSTs

Exam-Level 06

Announcements

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
	2/26 Lab 5 Due Homework 2 Due				3/1 Lab 6 Due	
			3/6 Project 2A Due		3/8 Lab 7 Due	

Note: I don't display these during discussion; I draw out concepts on the chalkboard. This is just for future review/my reference.

Content Review

Asymptotics Advice

- Asymptotic analysis is only valid on very large inputs, and comparisons between runtimes is only useful when comparing inputs of different orders of magnitude.
- Use Θ where you can, but won't always have tight bound (usually default to O)
- **Reminder: total work done = sum of all work per iteration or recursive call**
- While common themes are helpful, rules like "nested for loops are always N^2" can easily lead you astray (pay close attention to stopping conditions and how variables update)
- Drop lower-order terms (ie. $n^3 + 10000n^2 - 5000000 \rightarrow \Theta(n^3)$)

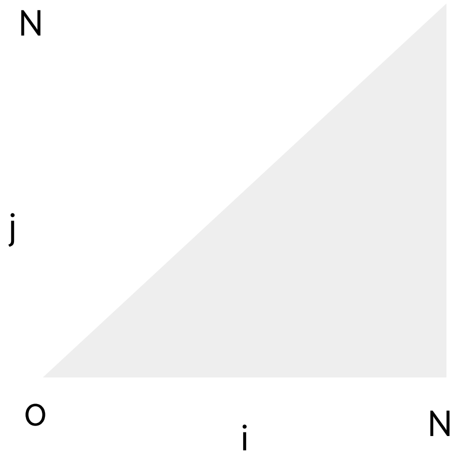
Asymptotics Advice

- For recursive problems, it's helpful to draw out the tree/structure of method calls
- Things to consider in your drawing and calculations of total work:
  - Height of tree: how many levels will it take for you to reach the base case?
  - Branching factor: how many times does the function call itself in the body of the function?
  - Work per node: how much actual work is done per function call?
- Life hack pattern matching when calculating total work where f(N) is some function of N
  - $1 + 2 + 3 + 4 + 5 + \ldots + f(N) = [f(N)]^2$
  - $1 + 2 + 4 + 8 + 16 + \ldots + f(N) = f(N)$
    - Rule applies with any geometric factor between terms, like $1 + 3 + 9 + \ldots + f(N)$

Asymptotics Advice

- Doing problems graphically can be helpful if you're a visual learner (plot variable values and calculate area formula):

```
for (int i = 0; i < N; i++) {
     for (int j = 0; j < i; j++) {
          /* Something constant */
     }
}
```



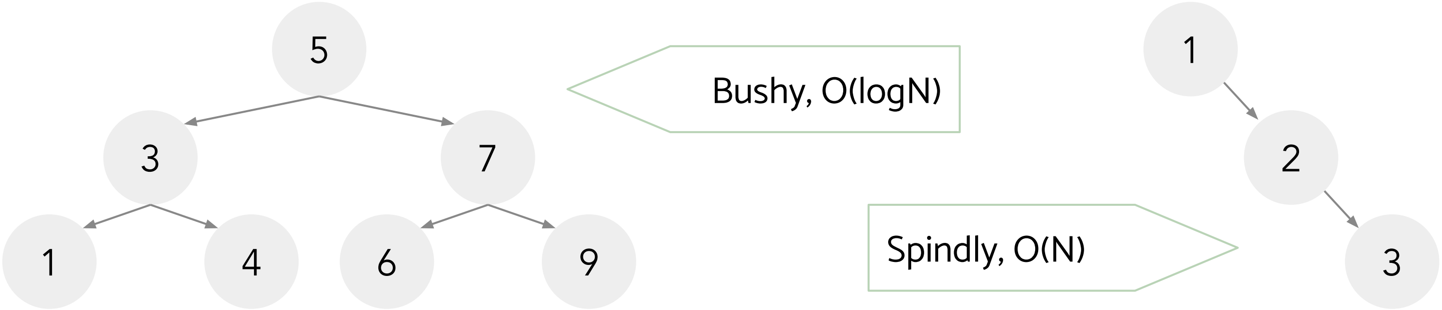
$\frac{1}{2} N^2 = N^2$

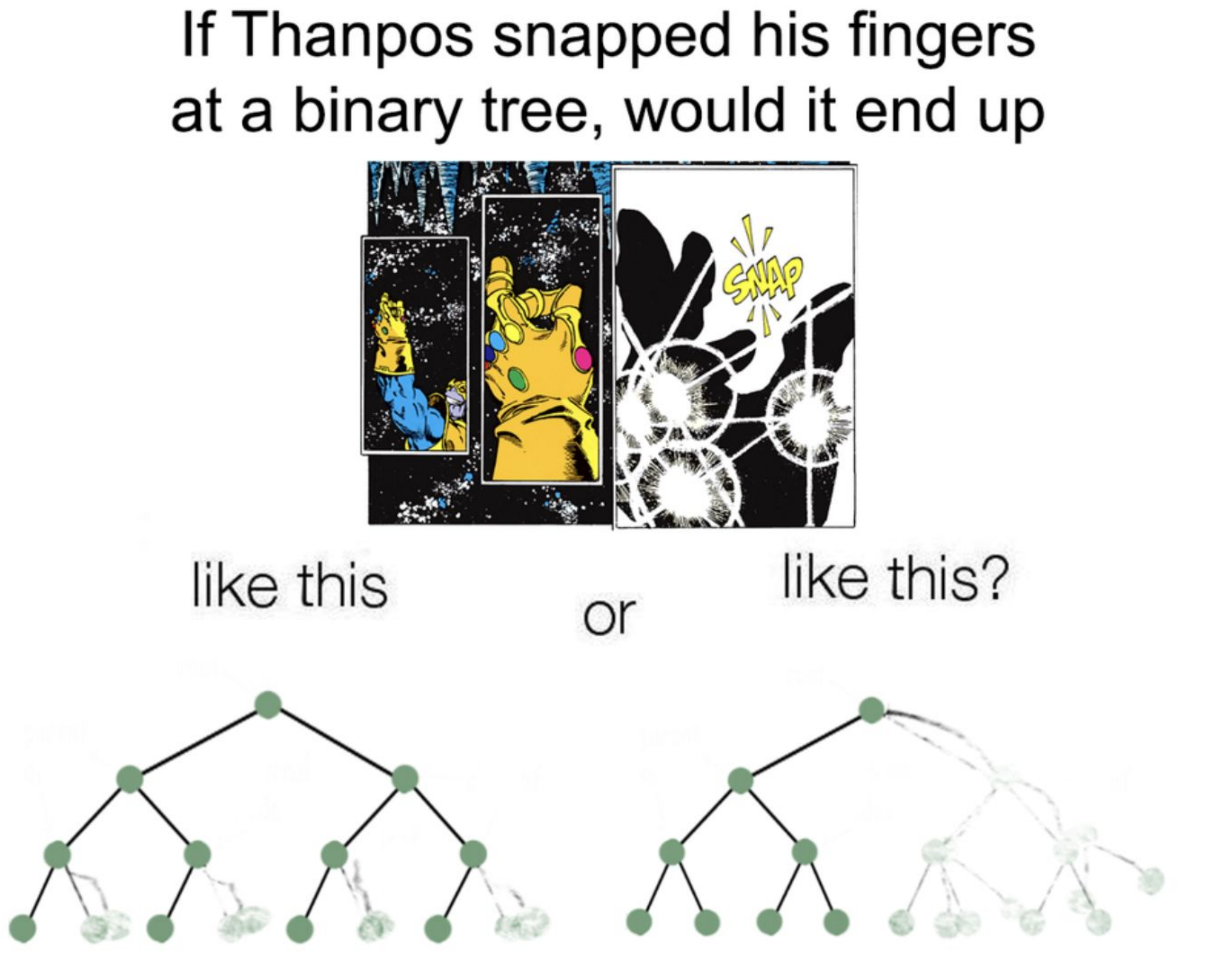
Binary Search Trees

Binary Search Trees are data structures that allow us to quickly access elements in sorted order. They have several important properties:

1. Each node in a BST is a root of a smaller BST
2. Every node to the left of a root has a value “lesser than” that of the root
3. Every node to the right of a root has a value “greater than” that of the root

BSTs can be bushy or spindly:

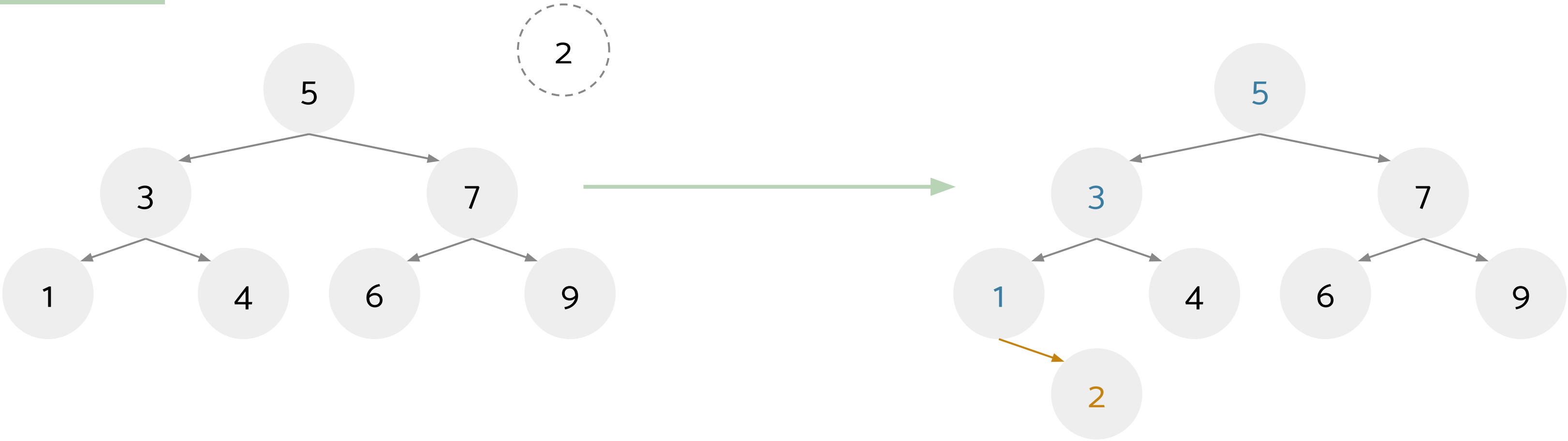



If Thanpos snapped his fingers
at a binary tree, would it end up
SNAP
like this
or
like this?

BST Insertion

Items in a BST are always inserted as leaves.

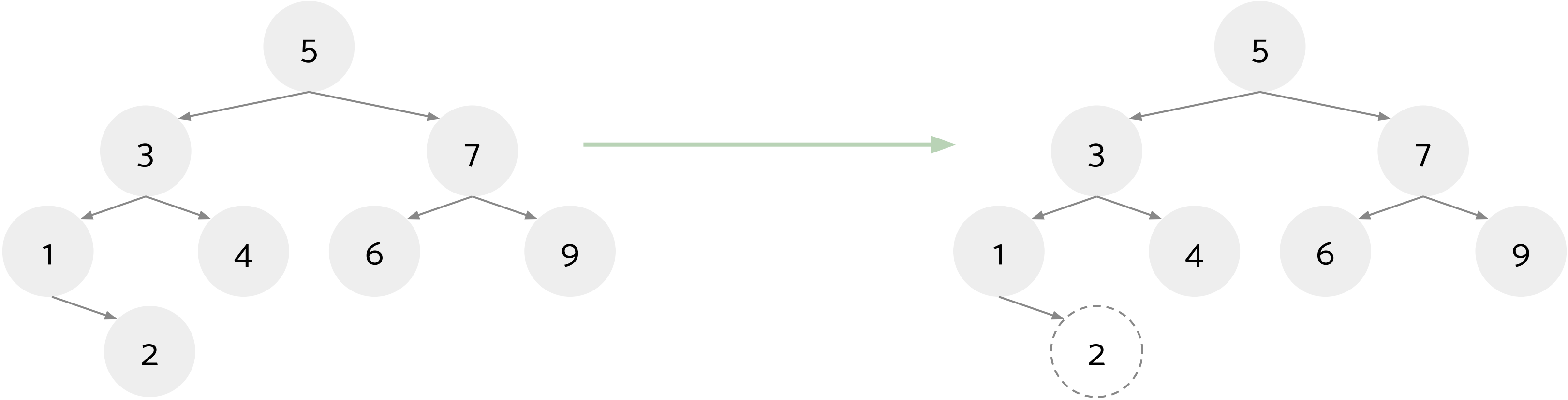
```
insert(2)
```



BST Deletion

Items in a BST are always deleted via a method called **Hibbard Deletion**. There are several cases to consider:

`delete(2)`

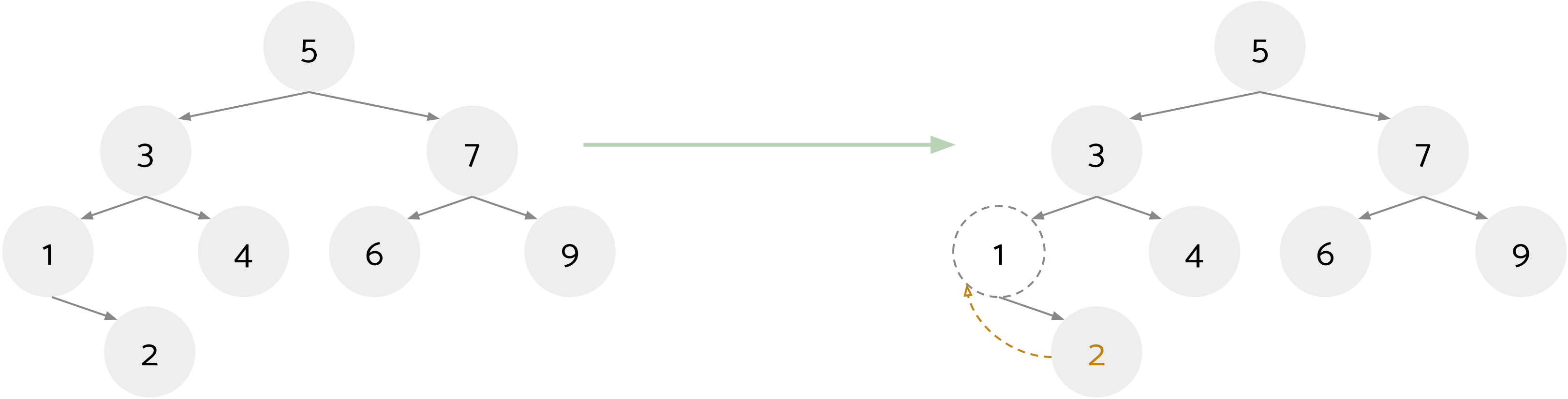


In this case, the node has no children so deletion is an easy process.

BST Deletion

Items in a BST are always deleted via a method called **Hibbard Deletion**. There are several cases to consider:

`delete(1)`

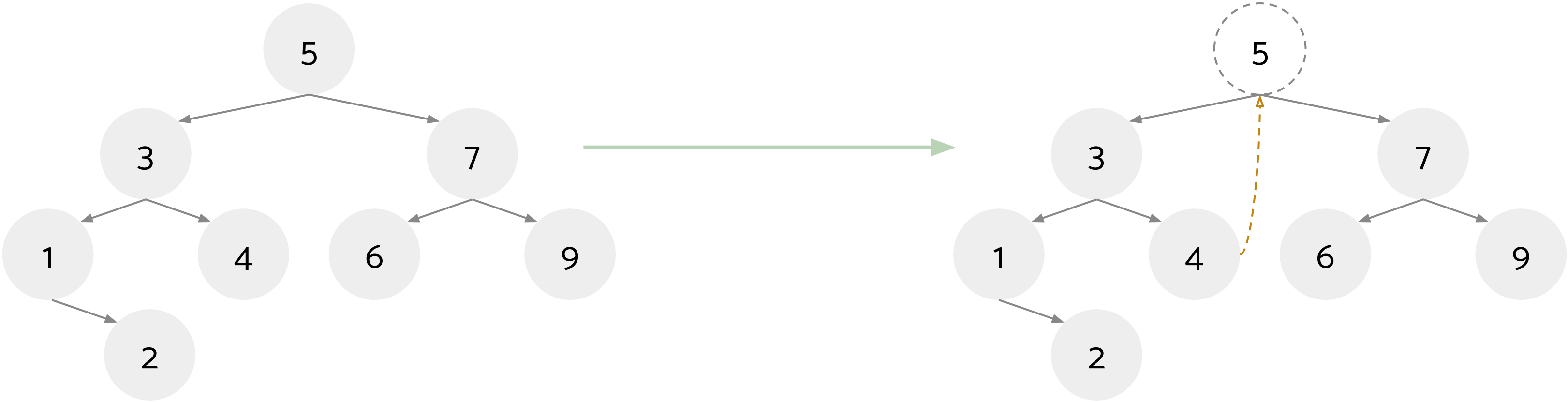


In this case, the node has one child, so it simply replaces the deleted node, and then we act as if the child was deleted in a recursive pattern until we hit a leaf.

BST Deletion

Items in a BST are always deleted via a method called **Hibbard Deletion**. There are several cases to consider:

`delete(5)`



In this case, the node has two children, so we pick either the leftmost node on in the right subtree or the rightmost node in the left subtree.

Worksheet

1 Finish the Runtimes

Below we see the standard nested for loop, but with missing pieces!

```
for (int i = 1; i < ______; i = ______) {
    for (int j = 1; j < ______; j = ______) {
        System.out.println("Circle is the best TA");
    }
}
```

For each part below, **some** of the blanks will be filled in, and a desired runtime will be given. Fill in the remaining blanks to achieve the desired runtime! There may be more than one correct answer.

Hint: You may find `Math.pow` helpful.

(a) Desired runtime: $\Theta(N^2)$

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < i; j = _j+k___) {
        System.out.println("This is one is low key hard");
    }
}
```

$k \in \mathbb{N}$, such as j+1

(b) Desired runtime: $\Theta(\log(N))$

```
for (int i = 1; i < N; i = i * 2) {
    for (int j = 1; j < __k___; j = j * 2) {
        System.out.println("This is one is mid key hard");
    }
}
```

already logarithmic

$k \in \mathbb{N}$

(c) Desired runtime: $\Theta(2^N)$

```
for (int i = 1; i < N; i = i + 1) {
    for (int j = 1; j < Math.pow(2,i); j = j + 1) {
        System.out.println("This is one is high key hard");
    }
}
```

Create dominating sum

Could also do $\frac{2^N}{N}$; does that N times which sums to 2^N

(d) Desired runtime: $\Theta(N^3)$

```
for (int i = 1; i < Math.pow(2,i); i = i * 2) {
    for (int j = 1; j < N * N; j = j+1___) {
        System.out.println("yikes");
    }
}
```

makes loops independent of each other

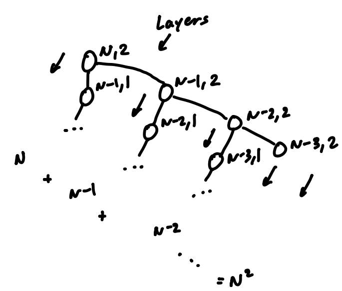
2 Asymptotics is Fun!

(a) Using the function `g` defined below, what is the runtime of the following function calls? Write each answer in terms of `N`. Feel free to draw out the recursion tree if it helps.

```
1 void g(int N, int x) {
2     if (N == 0) {
3         return;
4     }
5     for (int i = 1; i <= x; i++) {
6         g(N - 1, i);
7     }
8 }
```

`g(N, 1)`: $\Theta($ N $)$

`g(N, 2)`: $\Theta($ N^2 $)$



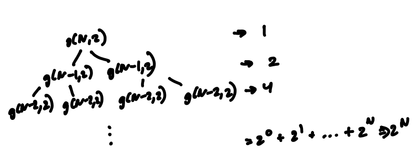
(b) Suppose we change line 6 to `g(N - 1, x)` and change the stopping condition in the for loop to `i <= f(x)` where `f` returns a random number between 1 and `x`, inclusive. For the following function calls, find the tightest Ω and big O bounds. Feel free to draw out the recursion tree if it helps.

```
1 void g(int N, int x) {
2     if (N == 0) {
3         return;
4     }
5     for (int i = 1; i <= f(x); i++) {
6         g(N - 1, x);
                   ↰ x, not i
7     }
8 }
```

`g(N, 2)`: $\Omega($ N $)$, $O($ 2^N $)$

↑ f(x)=1 — value of x doesn't matter

↑ f(x) = x = 2 — 2 is the branching factor



`g(N, N)`: $\Omega($ N $)$, $O($ N^N $)$

Same but base is N: $N^0 + N^1 + N^2 + \dots + N^N \Rightarrow N^N$

↑ f(x)=1 — value of x doesn't matter

↑ f(x) = x = N — N is the branching factor

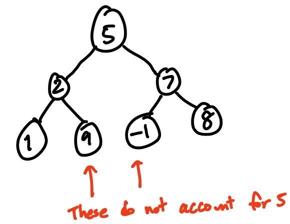
3 Is This a BST?

In this setup, assume a BST (Binary Search Tree) has a key (the value of the tree root represented as an int) and pointers to two other child BSTs, left and right.

(a) The following code should check if a given binary tree is a BST. However, for some trees, it returns the wrong answer. Give an example of a binary tree for which brokenIsBST fails.

```
public static boolean brokenIsBST(BST tree) {
    if (tree == null) {
        return true;
    } else if (tree.left != null && tree.left.key > tree.key) {
        return false;
    } else if (tree.right != null && tree.right.key < tree.key) {
        return false;
    } else {
        return brokenIsBST(tree.left) && brokenIsBST(tree.right);
    }
}
```

checks only one layer



(b) Now, write isBST that fixes the error encountered in part (a).

Hint: You will find Integer.MIN_VALUE and Integer.MAX_VALUE helpful.

Hint 2: You want to somehow store information about the keys from previous layers, not just the direct parent and children. How do you use the parameters given to do this?

```
public static boolean isBST(BST T) {
    return isBSTHelper(T, Integer.MAX_VALUE, Integer.MIN_VALUE);
}

public static boolean isBSTHelper(BST T, int min, int max) {

    if (T == null) {

        return true

    } else if (T.key < min || T.key > max) {

        return false

    } else {

        return isBSTHelper(T.left, min, T.key) && isBSTHelper(T.right, T.key, max)

    }
}
```

chosen to be replaced immediately

had to split into left and right cases

assumes no duplicates

should be min < key < max

Elements on left must be smaller than current node's value

Elements on the right must be greater than current node's value