CSE 373: BSTs, AVL trees

Michael Lee
Wednesday, Jan 17, 2018

Warmup

```java
public static void mystery(int n) {
    if (n <= 4) {
        System.out.println("Hello");
    } else {
        mystery(n - 1);
        for (int i = 0; i < n; i++)
            System.out.println("World");
        mystery(n - 2);
    }
}
```

With your neighbor, answer the following questions:

1. How much work is done JUST within the base case?
2. Within the recursive case, how much work do we do IGNORING the recursive calls?
3. How much work does each recursive call make, in terms of $T(\ldots)$ and n?

Warmup


```java
public static void mystery(int n) {
    if (n <= 4) {
        System.out.println("Hello");
    } else {
        mystery(n - 1);
        for (int i = 0; i < n; i++)
            System.out.println("World");
        mystery(n - 2);
    }
}
```


Now, fill in the gaps to construct your recurrence:

$$T(n) = \begin{cases} \text{workDoneInBaseCase} & \text{When } n \text{ is...} \\ \text{nonrecursiveWork} + \text{recursiveWork} & \text{Otherwise} \end{cases}$$

Answer:

$$T(n) = \begin{cases} 1 & \text{When } n \leq 4 \\ n + T(n-1) + T(n-2) & \text{Otherwise} \end{cases}$$

Announcements

- CSE 373 section AD has been changed to THO 125
- Project 2 due tonight
  PSA: After uploading to Canvas, double-check and make sure you've submitted the correct files.
- Written homework 1 will be released tonight; due in a week. (Reminder: work on this solo)
- Everybody gets an extra late day.

Last time...

Observation: sometimes, keys are *comparable* and *sortable*.

Idea: Can we exploit the "sortability" of these keys?

Suppose we add the following **invariant** to ArrayDictionary:

SortedArrayDictionary invariant
The internal array, at all times, **must** remain sorted.

How do you implement get? What's the big-Θ bound?

The binary search algorithm

Core algorithm (in *pseudocode*):

```
public V get(K key):
    return search(key, 0, this.size)

private K search(K key, int lowIndex, int highIndex):
    if lowIndex > highIndex:
        key not found, throw an exception
    else:
        middleIndex = average of lowIndex and highIndex
        pair = this.array[middleIndex]

        if pair.key == key:
            return pair.value
        else if pair.key < key:
            return search(key, lowIndex, middleIndex)
        else if pair.key > key:
            return search(key, middleIndex + 1, highIndex)
```

Answer: $T(n) \approx \begin{cases} 1 & \text{When } n \leq 0 \\ c + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{Otherwise} \end{cases}$

Finding a closed form

Our answer: $T(n) \approx \begin{cases} 1 & \text{When } n \leq 0 \\ c + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{Otherwise} \end{cases}$

Question: how do we find a closed form? Try unfolding?

$$T(n) = c + (c + (c + \ldots + (c + 1))) = \underbrace{c + c + \ldots + c}_{t = \text{Num times}} + 1$$

n	0	1	2	4	6	8	10	12	16	...	32	...	64
t	0	1	2	3	3	4	4	4	5	...	6	...	7

What's the relationship? $n \approx 2^{t+1}$

Solve for t: $t \approx \log_2(n) - 1$

Final model: $T(n) \approx c(\log_2(n) - 1) + 1$

So, we conclude: $T(n) \in \Theta(\log_2(n))$

The punchline

The punchline:

Binary search takes about $\Theta(\log(n))$ time, where n is the initial size of the array.

Note: in computer science, we assume $\log(n) = \log_2(n)$.

SortedArrayDictionary

Fill in the remainder of this table for SortedArrayDictionary:

Operation	Description of algorithm	Big-Θ bound
get	Use binary search.	$\Theta(\log(n))$
put	Use binary search to find key. If it doesn't exist, insert into array.	$\Theta(n)$
remove	Use binary search to find key. Once found, remove it and shift over remaining elements.	$\Theta(n)$
containsKey	Use binary search.	$\Theta(\log(n))$

Idea: Moving away from lists

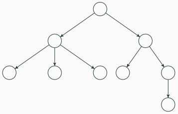
Observation: *Changing* our array is still difficult

Idea: Use a different data structure optimized for both searching and insertion?

Answer: Use a *Binary Search Tree* (BST)

Formal definition of trees

Example of a tree:



A tree consists of *nodes* where each node has at most one *parent* and zero or more *children*. Every single node (except one) must have a parent.

Some definitions

Some definitions:

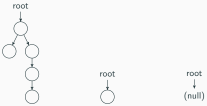
- **Root node:** The (single) node with no parent – the "top" of the tree
- **Branch node:** A node with one or more children
- **Leaf node:** A node with no children
- **Edge:** A pointer from one node to another
- **A subtree:** A node and all of its descendants
- **Height:** The number of edges contained in the longest path from the root node to some leaf node

Height of a tree

Height of a tree

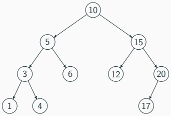
The **height** of a tree is the number of edges contained in the **longest** path from the root node to some leaf node.

What are the heights of these trees?



Binary Search Trees

Example of a binary **SEARCH** tree (BST):



A binary SEARCH tree contains comparable items such that for every node, all children to the left have smaller keys and all children to the right have larger keys.

Binary Search Tree vs Binary Tree

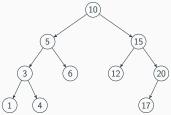
Important:

Binary Search Tree (BST) $\neq$ Binary Tree

Implementing the dictionary interface

Question: how do we implement the dictionary operations?

What are their runtimes with respect to n (number of nodes in the tree) and/or h (height of the tree)?



Binary Search Trees

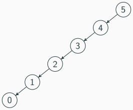
What is h, in terms of n?

For "balanced" trees, $h \approx \log_c(n)$, where c is the maximum number of children a node can have.

So for "balanced" trees, our dictionary operations are all in $\Theta(\log(n))$.

Binary Search Trees

Is this a valid binary tree? A valid binary search tree?



Yes. We call this a **degenerate tree**. What is h now?

For "degenerate" trees, $h \approx n$.

BinarySearchTreeDictionary

Fill in the remainder of this table for BinarySearchTreeDictionary:

Operation	Description of algorithm	Big-Θ bound
get	Recursively traverse down left or right child until we find the correct node.	$\Theta(h)$
put	Recursively search for node. If it doesn't exist, keep recursing until we hit an empty spot and add a new node.	$\Theta(h)$
remove	Recursively find node to remove. Once found, replace it with the largest node in the left subtree (or the smallest node in the right subtree).	$\Theta(h)$
containsKey	Do a recursive search.	$\Theta(h)$

A question

Core issue:

All BST operations take $\mathcal{O}(h)$ time, where h can be anywhere from $\log(n)$ to n, depending on the shape of the tree!

Question:

Is there some way we can make h always equal about $\log(n)$?

Can we somehow modify a BST so it always stays "balanced"?

AVL Trees: Invariants

Core idea: add extra **invariant** to BSTs that enforce balance.

AVL Tree Invariants

An AVL tree has the following invariants:

- **The "structure" invariant:**
  All nodes have 0, 1, or 2 children.
- **The "BST" invariant:**
  For all nodes, all keys in the *left* subtree are smaller; all keys in the *right* subtree are larger
- **The "balance" invariant:**
  For all nodes, $\text{abs}(\text{height}(\text{left})) - \text{height}(\text{height}(\text{right})) \leq 1$.

AVL = **A**delson-**V**elsky and **L**andis

Interlude: Exploring the balance invariant

Question: why $\text{abs}(\text{height}(\text{left})) - \text{height}(\text{height}(\text{right})) \leq 1$?

Why not $\text{height}(\text{left}) = \text{height}(\text{right})$?

What happens if we insert two elements. What happens?

AVL tree invariants review

Question: is this a valid AVL tree?