

# Exam 2 Review

15-110 – Monday 03/23

# Announcements

- **Exam2 on Wednesday!**
  - Bring your paper notes ( $\leq 5$  pages), something to write with, and your andrewID card
  - Arrive **early if possible** – we're checking IDs at the door
- Hw5 includes **Code Review #2!** Same rules as Code Review #1.
  - Sign up for a slot here: <https://www.cs.cmu.edu/~110/hw/hw4-code-review.html>

# Summer TA Hiring

- Message from Daniel Anderson, summer instructor of 15-110:
  - If you're looking for a fun and rewarding way to contribute to 15-110 over the summer, please consider [applying](#) to be a TA for Summer Semester B! The course runs from June 22nd to July 31st and is a great opportunity to practice your teaching skills and solidify your knowledge of 15-110.
- We're planning to hire TAs for F26 too! The fall application will be released in two weeks.

# Review Topics

- Recursion
- Big O Calculation
- Hashed Search (if time)

# Recursion

# Recursion: Big Idea

The core idea of recursion is that we can solve problems by **delegating** most of the work instead of solving it immediately.

This works because we make the input to the problem **slightly smaller** every time the function is called. That means it will eventually hit a **base case**, where the answer is known right away.

Once the base case returns a value, all the recursive calls can start returning their own values up the **chain of function calls** until they reach the initial call, which returns the final result.

# Writing Recursive Code: reverseList

When working with recursive code, it often helps to think **abstractly** about how to solve the problem with delegation before jumping into coding.

For example: what if we wanted to reverse a list using recursion? What is a **base case** that we can solve immediately?

In the recursive case, how do we make the problem smaller? What can we expect the recursive result to be **if the function works correctly**? Use that assumption to create the final result!

```
def reverseList(lst):  
    if len(lst) == 0:  
        return []  
    else:  
        smaller = reverseList(lst[1:])  
        return smaller + [ lst[0] ]
```

# Activity: Recursion Code Reading

It's important to understand **how recursive calls work** behind the scenes.

**You do:** trace by hand what the shown function call will output. Note the print statements!

```
def reverseList(lst):  
    print("Call:", lst)  
    if len(lst) == 0:  
        print("Return:", [])  
        return []  
    else:  
        smaller = reverseList(lst[1:])  
        result = smaller + [ lst[0] ]  
        print("Return:", result)  
        return result  
  
reverseList([3, 6, 9])
```

# Recursion with Multiple Calls

Recursion is most powerful when we make **multiple recursive calls** in the recursive case. This allows us to solve problems we can't solve without recursion.

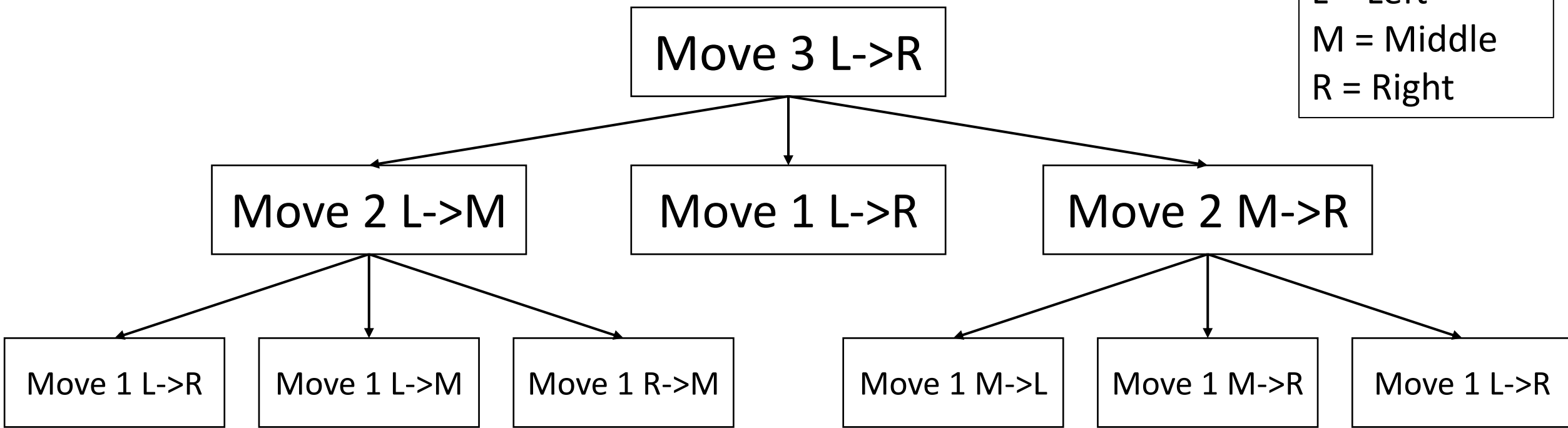
Towers of Hanoi (<https://www.mathsisfun.com/games/towerofhanoi.html>) is an example of a problem you can only solve using recursion.

- Move all but one of the discs to the temporary position – **recursive call**
- Move the remaining disc to the goal – **base case**
- Move the all-but-one discs to the goal – **recursive call**

# Towers of Hanoi – Visualize Moving 3 Discs

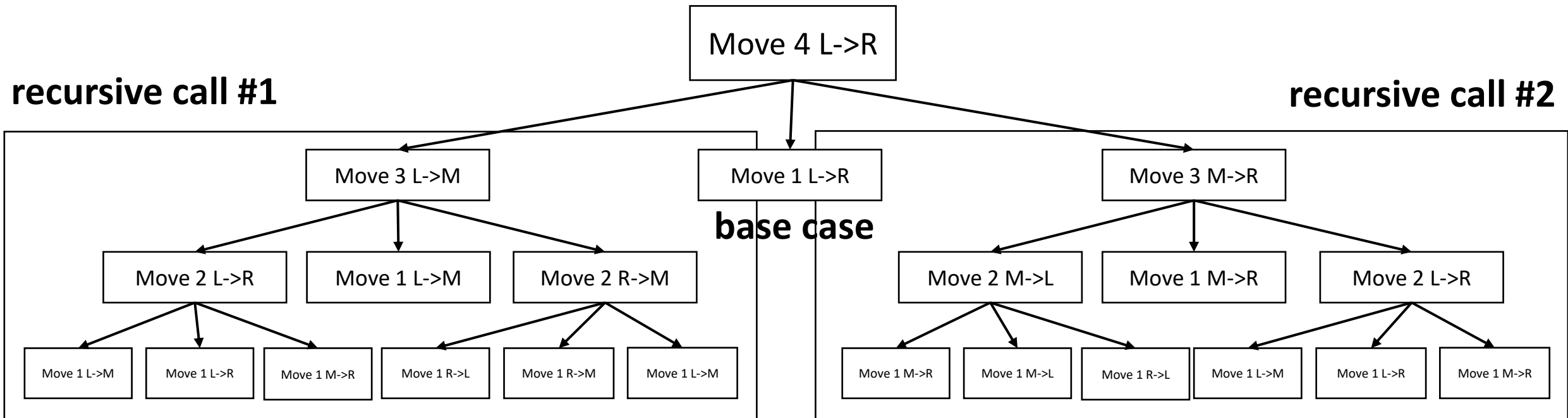
To move 3 discs, move two, then the one remaining, then move the two again.

L = Left  
M = Middle  
R = Right



# Towers of Hanoi – Visualize Moving 4 Discs

To move 4 discs, move 3 discs by **following the same procedure we outlined before**, just with a different destination. Use **abstraction** to solve the problem!



# Writing Recursive Code: getLeaves

We also use recursion with multiple calls when we write programs to traverse **trees**.

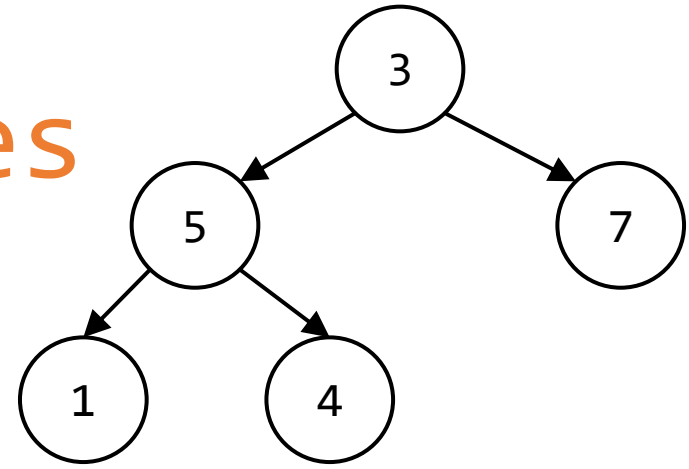
Specifically, we do two calls – one for each child of the root node.

Example: we want to collect all the **leaves** of the tree.

What will each recursive call return **if it works the way it's supposed to?**

Which **base cases** do we need?

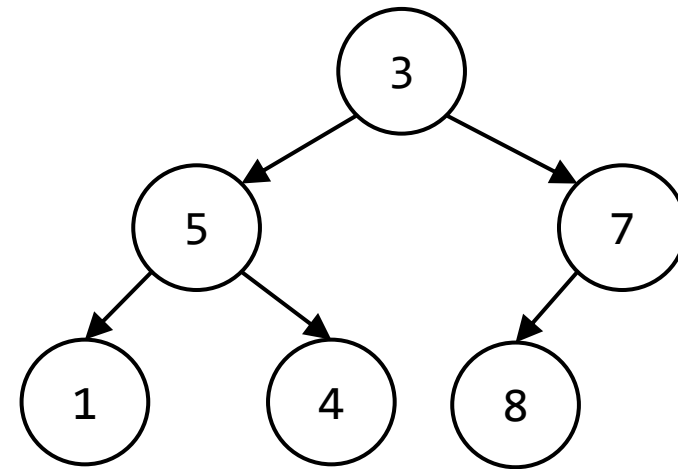
```
def getLeaves(t):  
    if t == None:  
        return []  
    elif t["left"] == None and t["right"] == None:  
        return [ t["contents"] ]  
    else:  
        # don't add this node, it's not a leaf!  
        result = []  
        result = result + getLeaves(t["left"])  
        result = result + getLeaves(t["right"])  
        return result
```



# Activity: addNodes

**You do:** write a recursive function `addNodes(t)` which takes a tree as input and returns the sum of all the nodes in the tree as output. For example, if the function is called on the tree shown below, it should return **28**.

```
t = { "contents" : 3,  
      "left" : { "contents" : 5,  
                  "left" : { "contents" : 1,  
                              "left" : None,  
                              "right" : None },  
                  "right" : { "contents" : 4,  
                              "left" : None,  
                              "right" : None } },  
      "right" : { "contents" : 7,  
                  "left" : { "contents" : 8,  
                              "left" : None,  
                              "right" : None },  
                  "right" : None } }
```



# Big O Calculation

# Purpose of Big O

We perform Big O calculations when we want to know how the number of actions an algorithm takes will change as the size of the input grows.

We don't care about **constant** actions; we only want to examine ones that are **dependent on the input**.

# Consider Each Line

Look at each line of the function. Is it connected to the input? How many actions does it do **in terms of the input size n?**

Don't worry about constant factors; we only care about n.

```
def getPalindrome(lst): # n = len(lst)
    result = [ ]
    for i in range(0, len(lst) // 2):
        if lst[i] == lst[len(lst) - 1 - i]:
            result.append(lst[i])
    return result
```

# Put Them All Together

Once you've considered each line, combine them properly to get the full function's runtime.

Lines that are **sequential** have runtimes added together. Lines that are **repeated** multiply the number of repetitions \* the work being repeated.

```
def getPalindrome(lst): # n = len(lst)
    result = [ ] # 0(1)
    for i in range(0, len(lst) // 2): # n reps
        if lst[i] == lst[len(lst) - 1 - i]: # 0(1)
            result.append(lst[i]) # 0(1)
    return result # 0(1)
```

$1 + n*(1 + 1) + 1 \rightarrow O(n)$

# Example: Quizlet6

What is the runtime of each line? What is the overall runtime?

```
# lst is a list, n = len(lst)
1. def example(lst, target):
2.     print(binarySearch(lst, target))
3.     sum = 0
4.     x = len(lst)
5.     for i in range(5, x, 10):
6.         if target in lst:
7.             sum = sum + i
8.     return sum
```

How many times will the loop iterate?

Are there any operations that rely on `lst` and don't run in constant time?

# Activity: Compute the Big-O

Consider the following function. What is its Big-O runtime in the worst case?

```
def example(s): # s is a string, n = len(s)
    result = ""
    for i in range(len(s)//2, len(s)):
        result = s[i] + result

    for j in range(len(s)//2):
        if s[j].isupper():
            result = result + s[j].lower()
        else:
            result = result + s[j]
    return result
```

**Hint:** `s.isupper` and `s.lower` both run in  $O(n)$  time, but what is the length of the strings we call them on?

# Hashed Search

# Big Idea

Why do we care about hash functions?

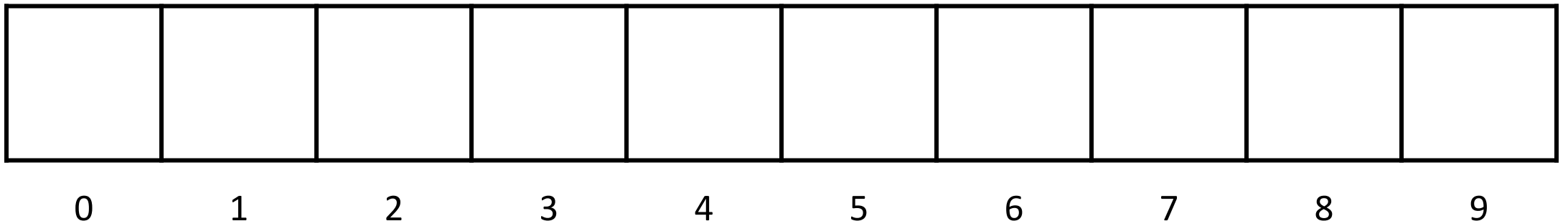
We search all the time, so we want the fastest possible search. Storing items in a hashtable lets us look up whether an item is in the table in  **$O(1)$**  time. You can't get faster than that!

How can we search in constant time? The algorithm needs to know **where** the value it's looking for will be stored **if** that value is actually in the table.

# Hashtables

A **hashtable** is like a big, empty list of a designated size. Like in a list, each slot ('bucket') in the table is associated with an **integer index**, from 0 to  $\text{len}(\text{table})-1$ .

When we want to put a value in the hashtable, we insert it at a specific index based on the result of a **hash function**.



# Hash Functions

A **hash function** is a function that maps Python values to integers. Those integers can then be used to find an index in the hashtable to store the value.

We can use the built-in Python `hash` function or write our own. Either way, the hash function must follow two rules:

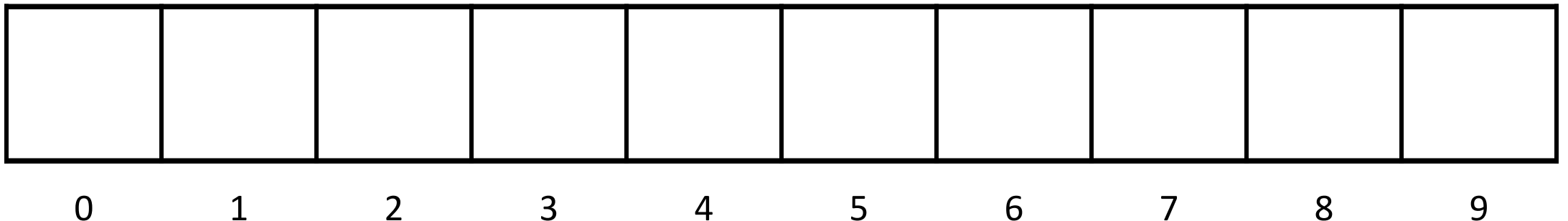
- The result returned when the function is called on some value must not change across calls
- The function should usually return different numbers when called on different values

# Storing/Finding Values in Hashtables

Both storing a value in a hashtable and checking whether a value is in a hashtable follow the same procedure, which produces which index to check.

1. Run the hash function on the value to get the hashed value.
2. Mod the hashed value by the hashtable size to get the final index

**Demo:** Let's practice with some strings and the built-in hash function.



# Why $O(1)$ ?

Why is looking up a value in a hashtable  $O(1)$  time?

We don't need to check every bucket in the hashtable. Only look in one bucket, the one with the index associated with the hashed value.

**Important:** this only works if the value we're searching for can't change (it's **immutable**), the hash function is **well-designed** (follows the two rules), and if the hashtable is **large enough** for the stored values to spread out. (10 buckets isn't nearly enough!)