Interview Topics: Linear Data Structures

Week 4

What is OOP?

Object Oriented Programming

is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).

Why OOP?

- Modular easily manageable large code bases
  - Code reuse thanks to inheritance and polymorphism
- Fine grained encapsulation
- OOP Friendly Languages: Java, C++, C#, Python, Ruby
  - OOP Frenemies: JavaScript
  - NOT OOP: C

What are Data Structures?

- Basic definition: A way of organizing and storing data + things to do with that data
- **Abstract Data Type (ADT)**
  - *A definition for expected operations and behavior*
  - A mathematical description of a collection with a set of supported operations and how they should behave when called upon
  - Describes what a collection does, not how it does it
  - Can be expressed as an interface
  - Examples: List, Map, Set
- **Data Structure**
  - *A way of organizing and storing related data points*
  - *An object that implements the functionality of a specified ADT*
  - *Describes exactly how the collection will perform the required operations*
  - *Examples: LinkedIntList, ArrayIntList*

Data Structures and ADTs you should know

- ADTs
  - List
  - Set
  - Map/Dictionary
  - Stack
  - Queue
  - Priority Queue
  - Graph
  - Disjoint Set
  -
- Data Structures
  - Array
  - Linear Linked Nodes (Linked List)
  - Hierarchical Linked Nodes (Tree)
    - BST
    - AVL
    - Heap
    - Trie
  - Hash Table
  -

Why Data Structures?

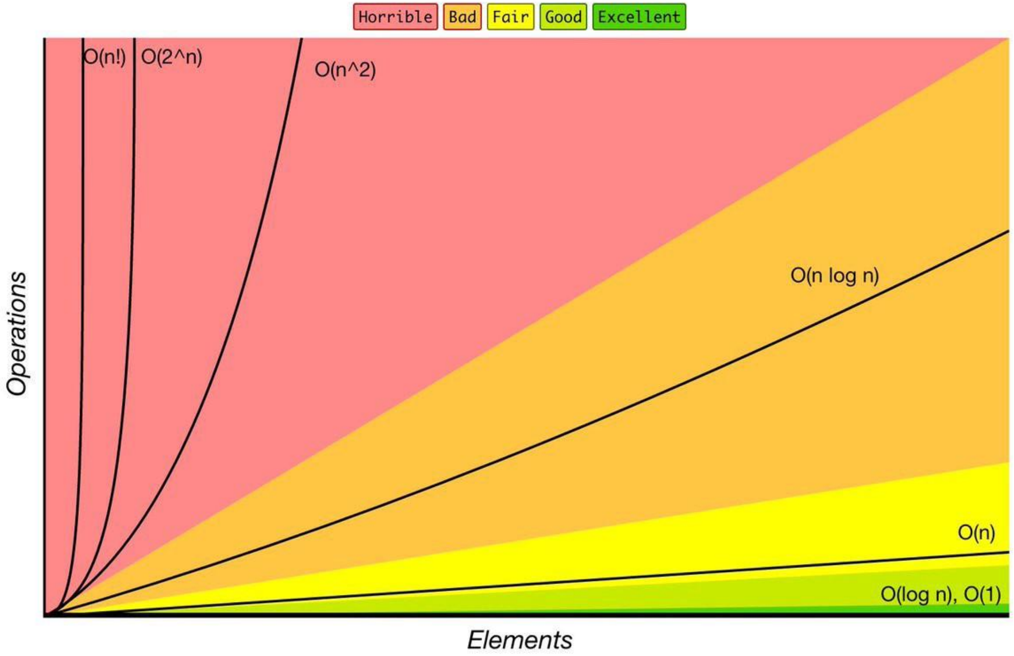
- You use them SOOO much in the real world.
- They are the foundation to more difficult programming concepts.
- You can do fancy things to make your code faster!

Intermission: Big O

With Big O Notation we express the runtime in terms of — how quickly it grows relative to the input, as the input gets larger.

We are mostly talking about complexity, not performance.

Intermission: Big O



Things you should know about Big O

- Time vs. space complexity
- Best case, worst case, expected case
- Dominant vs. non-dominant terms

https://www.bigocheatsheet.com/

Going back to data structures...

Some problems mention explicit data structures, “Given a binary tree”/ “In a Hash table” etc.

But most problems essentially comes down to “what data structure(s) should I use? And how? And why?”

Arrays & Lists

- Both static size and resizable arrays
- Understand runtime for:
  - Finding item at unknown index
  - Insertion at end, insertion at front
  - Adding beyond capacity
- Singly vs. Doubly linked lists
- “Runner Technique”
  - Aka “Two pointers technique”

List ADT + ArrayList & LinkedList Implementation

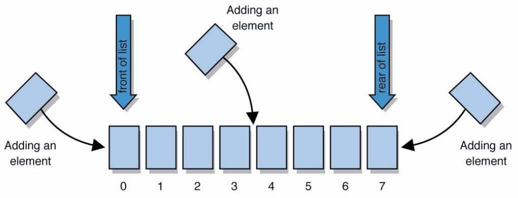
List ADT

state

Set of ordered items
Count of items

behavior

get(index) return item at index
set(item, index) replace item at index
append(item) add item to end of list
insert(item, index) add item at index
delete(index) delete item at index
size() count of items



ArrayList

uses an Array as underlying storage

ArrayList<E>

state

```
data[]
size
```

behavior

```
get return data[index]
set data[index] = value
append data[size] = value, if out of space grow data
insert shift values to make hole at index, data[index] = value, if out of space grow data
delete shift following values forward
size return size
```

0	1	2	3	4
88.6	26.1	94.4	0	0

list　　free space

LinkedList

uses nodes as underlying storage

LinkedList<E>

state

```
Node front
size
```

behavior

```
get loop until index, return node's value
set loop until index, update node's value
append create new node, update next of last node
insert create new node, loop until index, update next fields
delete loop until index, skip node
size return size
```

Design Decisions

- **Situation #1:** Write a data structure that implements the List ADT that will be used to store a list of songs in a playlist.
  - **ArrayList – I want to be able to shuffle play on the playlist**
- **Situation #2:** Write a data structure that implements the List ADT that will be used to store the history of a bank customer's transactions.
  - **ArrayList – optimize for addition to back and accessing of elements**
- **Situation #3:** Write a data structure that implements the List ADT that will be used to store the order of students waiting to speak to a TA at a tutoring center
  - **LinkedList - optimize for removal from front**
  - **ArrayList – optimize for addition to back**

List ADT tradeoffs

- Last time: we used “slow” and “fast” to describe running times. Let’s be a little more precise.

Recall these basic Big-O ideas from 14X: Suppose our list has N elements

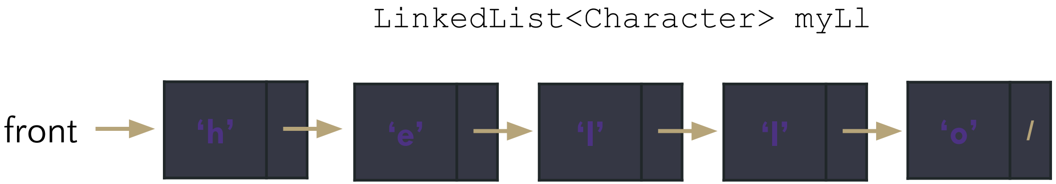
- If a method takes a constant number of steps (like 23 or 5) its running time is O(1)
- If a method takes a linear number of steps (like 4N+3) its running time is O(N)

For ArrayLists and LinkedLists, what is the O() for each of these operations?

- Time needed to access N^{th} element:
- Time needed to insert at end (the array is full!)

What are the memory tradeoffs for our two implementations?

- Amount of space used overall
- Amount of space used per element



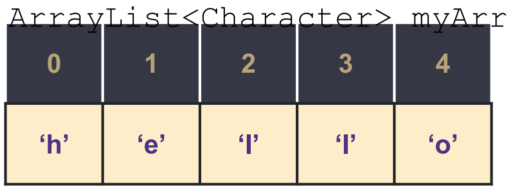
List ADT tradeoffs

- Time needed to access N^{th} element:
  - ArrayList: O(1) constant time
  - LinkedList: O(N) linear time

Time needed to insert at N^{th} element (the array is full!)
  - ArrayList: O(N) linear time
  - LinkedList: O(N) linear time

Amount of space used overall
  - ArrayList: sometimes wasted space
  - LinkedList: compact

Amount of space used per element
  - ArrayList: minimal
  - LinkedList: tiny extra



Dictionaries for Interviews

- THE MOST USEFUL ADT
  - Hash Map = teh MVP

Review: Dictionaries

Dictionary ADT

state
Set of items & keys
Count of items

behavior
put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

- Why are we so obsessed with Dictionaries?

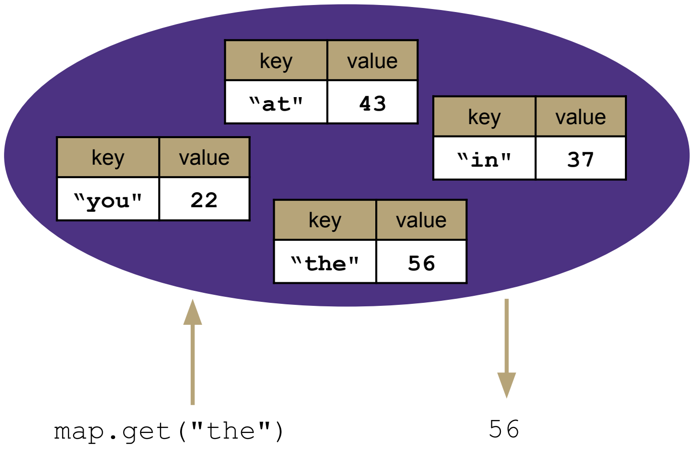
When dealing with data:

- Adding data to your collection
- Getting data out of your collection
- Rearranging data in your collection

Operation		ArrayList	LinkedList	HashTable	BST	AVLTree
put(key,value)	best					
	worst					
get(key)	best					
	worst					
remove(key)	best					
	worst					

Review: Maps

- **map**: Holds a set of distinct *keys* and a collection of *values*, where each key is associated with one value.
  - a.k.a. "dictionary"



Dictionary ADT

state

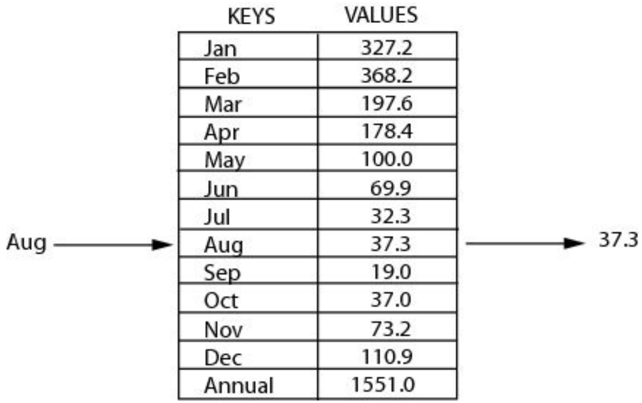
Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

supported operations:

- **put**(*key*, *value*): Adds a given item into collection with associated key,
  - **if the map previously had a mapping for the given key, old value is replaced.**
- **get**(*key*): Retrieves the value mapped to the key
- **containsKey**(key): returns true if key is already associated with value in map, false otherwise
- **remove**(*key*): Removes the given key and its mapped value



Implementing a Map with an Array

Map ADT

state
Set of items & keys
Count of items

behavior
put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

ArrayMap<K, V>

state
`Pair<K, V>[] data`

behavior
put find key, overwrite value if there. Otherwise create new pair, add to next available spot, grow array if necessary
get scan all pairs looking for given key, return associated item if found
containsKey scan all pairs, return if key is found
remove scan all pairs, replace pair to be removed with last pair in collection
size return count of items in dictionary

```
containsKey('c')
get('d')
put('b', 97)
put('e', 20)
```

0	1	2	3	4
('a', 1)	('b', 97)	('c', 3)	('d', 4)	('e', 20)

Big O Analysis – (if key is the last one looked at / not in the dictionary)

put()	O(N) linear
get()	O(N) linear
containsKey()	O(N) linear
remove()	O(N) linear
size()	O(1) constant

Big O Analysis – (if the key is the first one looked at)

put()	O(1) constant
get()	O(1) constant
containsKey()	O(1) constant
remove()	O(1) constant
size()	O(1) constant

Implementing a Map with Nodes

Map ADT

state
Set of items & keys
Count of items

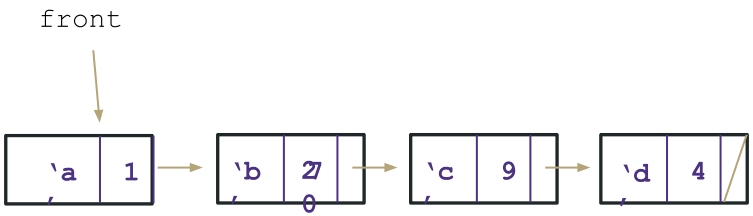
behavior
put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

LinkedMap<K, V>

state
`front`
`size`

behavior
`put` if key is unused, create new with pair, add to front of list, else replace with new value
`get` scan all pairs looking for given key, return associated item if found
`containsKey` scan all pairs, return if key is found
`remove` scan all pairs, skip pair to be removed
`size` return count of items in dictionary

```
containsKey('c')
get('d')
put('b', 20)
```



Big O Analysis – (if key is the last one looked at / not in the dictionary)

`put()`	O(N) linear
`get()`	O(N) linear
`containsKey()`	O(N) linear
`remove()`	O(N) linear
`size()`	O(1) constant

Big O Analysis – (if the key is the first one looked at)

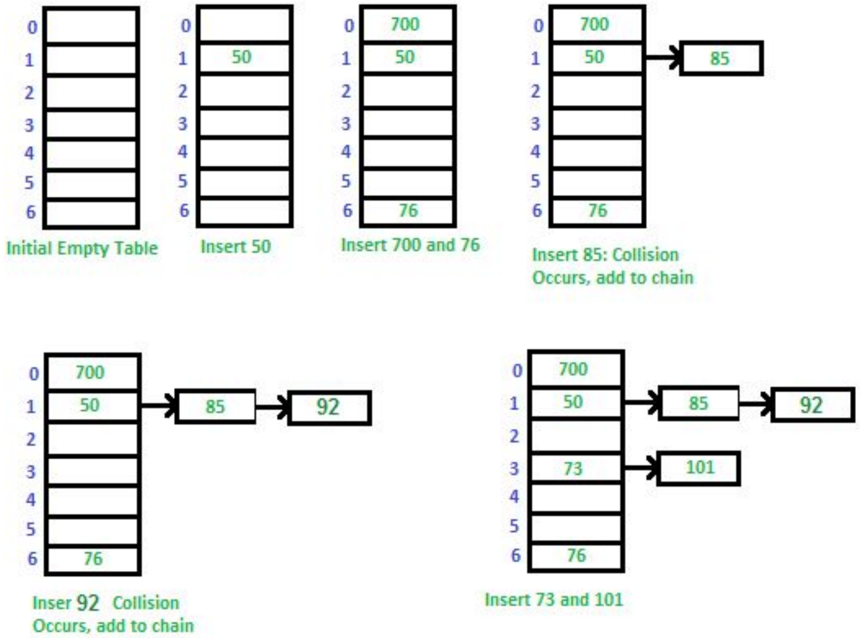
`put()`	O(1) constant
`get()`	O(1) constant
`containsKey()`	O(1) constant
`remove()`	O(1) constant
`size()`	O(1) constant

Hash *

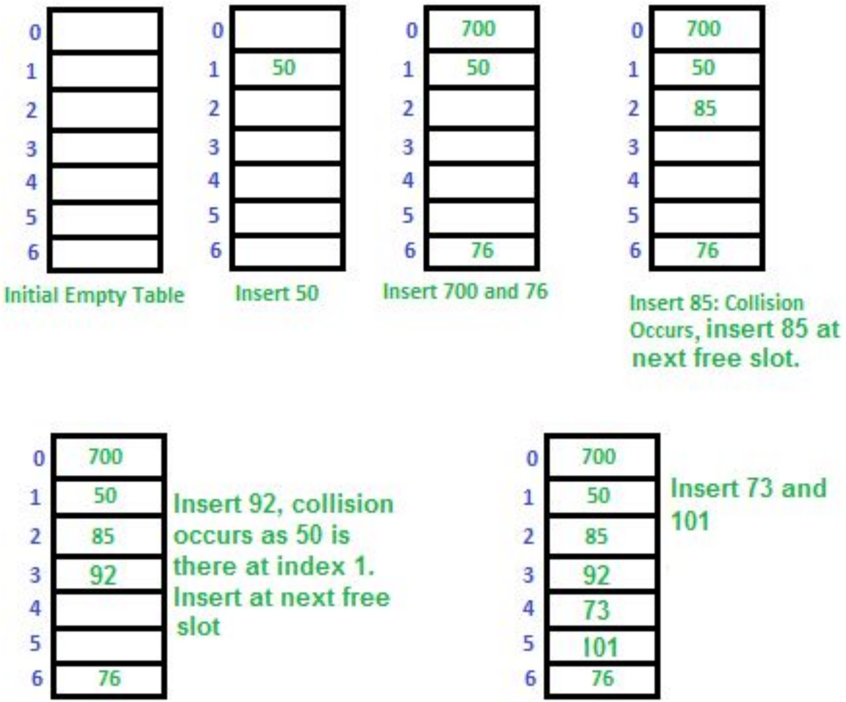
Hash *

- Efficient lookup
  - Constant time!
- Hash functions
  - "What's a good hash function?"
    - Efficiently computable.
    - Should uniformly distribute the keys
  - Rehashing
    - When to rehash (lambda)
    - How to rehash?
- Collision Resolution
  - Separate chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing

Separate Chaining



Open Addressing



First Hash Function: % table size

indices	0	1	2	3	4	5	6	7	8	9
elements	"foo"	"biz"				"bar"			"bop"	

```
put(0, "foo");
put(5, "bar");
put(11, "biz")
put(18, "bop");
```

```
0 % 10 = 0
5 % 10 = 5
11 % 10 = 1
18 % 10 = 8
```

Implement First Hash Function

```
public void put(int key, int value) {
    data[hashToValidIndex(key)] = value;
}

public V get(int key) {
    return data[hashToValidIndex(key)];
}

public int hashToValidIndex(int k) {
   return k % this.data.length;
}
```

Operation		Array w/ indices as keys
put(key,value)	best	
	worst	
get(key)	best	
	worst	
containsKey(key)	best	
	worst	

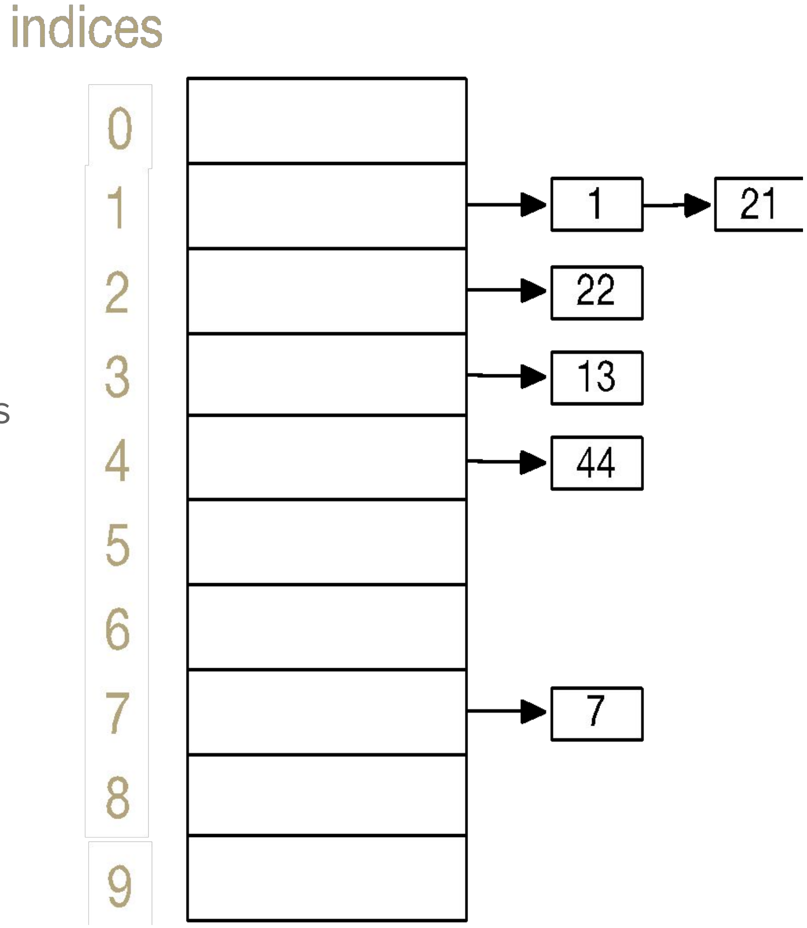
SimpleHashMap<Integer>

state
Data[]
size

behavior
put mod key by table size, put item at result
get mod key by table size, get item at result
containsKey mod key by table size, return data[result] == null
remove mod key by table size, nullify element at result
size return count of items in dictionary

Note: % is just a math operator like +, -, /, *, so it's constant runtime

Separate chaining



Solution 1: Separate Chaining

Each index in our array represents a “bucket”. When an item x hashes to index h:

- If the bucket at index h is empty: create a new list containing x
- If the bucket at index h is already a list: add x if it is not already present

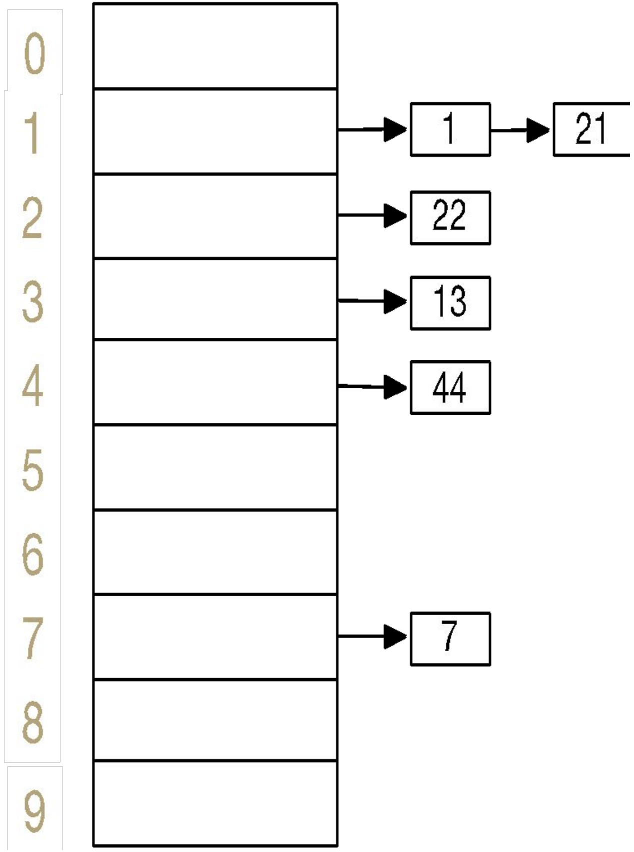
in other words:

If multiple things hash to the same index, then we’ll just put all of those in that same index bucket. Often, you’ll see the data structure chosen is a linked-list like structure.

Separate chaining

Reminder: the implementations of put/get/containsKey are all very similar, and almost always will have the same complexity class runtime

```
// some pseudocode
 public boolean containsKey(int key) {
     int bucketIndex = key % data.length;
     loop through data[bucketIndex]
         return true if we find the key in
         data[bucketIndex]
     return false if we get to here (didn't find it)
 }
```



<u>runtime analysis</u>
Are there different possible states for our Hash Map that make this code run slower/faster, assuming there are already n key-value pairs being stored?

Yes! If we had to do a lot of loop iterations to find the key in the bucket, our code will run slower.

In-practice situations for separate chaining

Generally we can achieve something close to the best case situation from the previous slide and maintain our Hash Map so that every bucket only has a small constant number of items. There may be some outliers that have slightly more buckets, but generally if we follow all the best practices, the runtime will still be $\Theta(1)$ for most cases!

(The worst case is still $\Theta(n)$ but again, we'll try really hard to prevent that)

Operation		Array w/ indices as keys
put(key,value)	best	
	In-practice	
	worst	
get(key)	best	
	In-practice	
	worst	
remove(key)	best	
	In-practice	
	worst	O(n)

Reminder: the in-practice runtimes are assuming an even distribution of the keys inside the array and following of best-practices to ensure the average chain length is low.

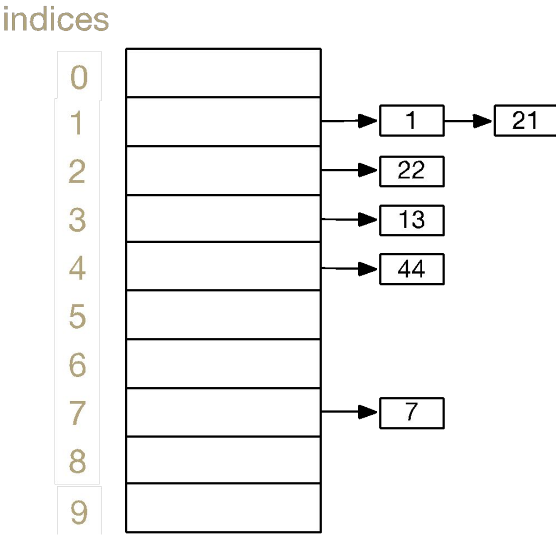
Lambda + resizing rephrased

To be more precise, the in-practice runtime depends on λ, the current average chain length.

However, if you resize once you hit that 1:1 threshold, the current λ is expected to be less than 1 (which is a constant / constant runtime, so

v

Operation		Array w/ indices as keys
put(key,value)	best	O(1)
	In-practice	O(λ)
	worst	O(n)
get(key)	best	O(1)
	In-practice	O(λ)
	worst	O(n)
remove(key)	best	O(1)
	In-practice	O(λ)
	worst	O(n)



"In-Practice" Case:
Depends on average number of elements per chain

Load Factor λ
If n is the total number of key-value pairs
Let c be the capacity of array
Load Factor λ = $\frac{n}{c}$

Good Hashing

The hash function of a HashDictionary gets called a LOT:

- When first inserting something into the map
- When checking if a key is already in the map
- When resizing and redistributing all values into new structure

This is why it is so important to have a “good” hash function. A good hash function is:

1. Deterministic – same input should generate the same output
2. Efficiency - it should take a reasonable amount o time
3. Uniformity – inputs should be spread “evenly” over output range

```
public int hashFn(String s) {
      return random.nextInt()
}
```

NOT deterministic

```
public int hashFn(String s) {
   if (s.length() % 2 == 0) {
      if (s.length(). % 2 == 0) {
         return 17;
      } else {
         return 43;
      }
   }
}
```

NOT efficient

```
public int hashFn(String s) {
   int retVal = 0;
   for (int I = 0; I < s.length(); i++) {
      for (int j = 0; j < s.length(); j++) {
         retVal += helperFun(s, I, j);
      }
   }
   return retVal;
}
```

NOT uniform

Linear Probing

Insert the following values into the Hash Table using a hashFunction of % table size and linear probing to resolve collisions
38, 19, 8, 109, 10

8

0	1	2	3	4	5	6	7	8	9
10								38	109

Problem:

- Linear probing causes clustering
- Clustering causes more looping when probing

Primary Clustering
When probing causes long chains of occupied slots within a hash table

Linear Probing

Insert the following values into the Hash Table using a hashFunction of % table size and linear probing to resolve collisions
38, 19, 8, 109, 10

8

0	1	2	3	4	5	6	7	8	9
10								38	109

Problem:

- Linear probing causes clustering
- Clustering causes more looping when probing

Primary Clustering
When probing causes long chains of occupied slots within a hash table

Linear Probing

Insert the following values into the Hash Table using a hashFunction of % table size and linear probing to resolve collisions
38, 19, 8, 109, 10

8

0	1	2	3	4	5	6	7	8	9
10								38	109

Problem:

- Linear probing causes clustering
- Clustering causes more looping when probing

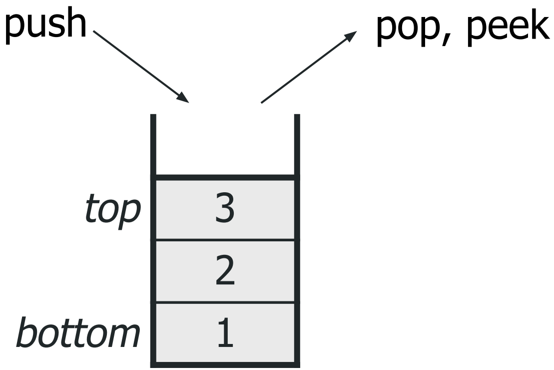
Primary Clustering
When probing causes long chains of occupied slots within a hash table

Stacks & Queues

- LIFO vs FIFO
  - Stack is LIFO
  - Queue is FIFO
- Stacks
  - Scenarios in which to use Stacks (LIFO)
    - Matching curly braces / tags
    - Undo function
  - Implementing with an array
- Queues
  - Scenarios in which to use Queues (FIFO)
    - Printer queue / task management
  - Priority Queue ADT
    - Disjoint Set implementation (graph)

Review: What is a Stack?

- **stack**: A collection based on the principle of adding elements and retrieving them in the opposite order.
  - Last-In, First-Out ("LIFO")
  - Elements are stored in order of insertion.
    - We do not think of them as having indexes.
  - Client can only add/remove/examine the last element added (the "top").



Stack ADT

state
Set of ordered items
Number of items

behavior
push(item) add item to top
pop() return and remove item at top
peek() look at item at top
size() count of items
isEmpty() count of items is 0?

supported operations:
- **push(item)**: Add an element to the top of stack
- **pop()**: Remove the top element and returns it
- **peek()**: Examine the top element without removing it
- **size():** how many items are in the stack?
- **isEmpty():** true if there are 1 or more items in stack, false otherwise

Implementing a Stack with an Array

Stack ADT

state

Set of ordered items
Number of items

behavior

push(item) add item to top
pop() return and remove item at top
peek() look at item at top
size() count of items
isEmpty() count of items is 0?

ArrayStack<E>

state

```
data[]
size
```

behavior

```
push data[size] = value, if
out of room grow data
pop return data[size - 1],
size-1
peek return data[size - 1]
size return size
isEmpty return size == 0
```

Big O Analysis

`pop()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`push()`	O(N) linear if you have to resize O(1) otherwise

```
push(3)
push(4)
pop()
push(5)
```

0	1	2	3
3	5		

`numberOfItems =` 2

Implementing a Stack with Nodes

Stack ADT

state

Set of ordered items
Number of items

behavior

push(item) add item to top
pop() return and remove item at top
peek() look at item at top
size() count of items
isEmpty() count of items is 0?

LinkedStack<E>

state

Node top
size

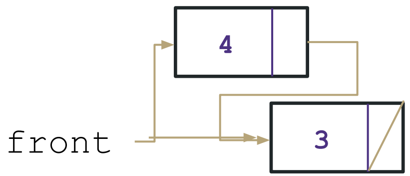
behavior

push add new node at top
pop return and remove node at top
peek return node at top
size return size
isEmpty return size == 0

Big O Analysis

pop()	O(1) Constant
peek()	O(1) Constant
size()	O(1) Constant
isEmpty()	O(1) Constant
push()	O(1) Constant

```
push(3)
push(4)
pop()
```



numberOfItems = 2

Review: What is a Queue?

- **queue**: Retrieves elements in the order they were added.
  - First-In, First-Out ("FIFO")
  - Elements are stored in order of insertion but don't have indexes.
  - Client can only add to the end of the queue, and can only examine/remove the front of the queue.

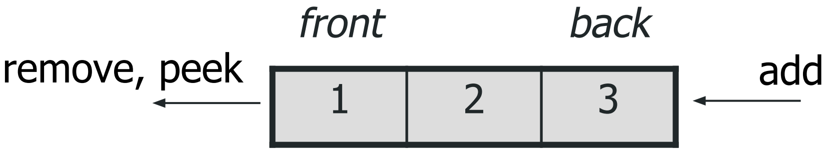
Queue ADT

state

Set of ordered items
Number of items

behavior

add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count of items is 0?



supported operations:

- **add(item):** aka "enqueue" add an element to the back.
- **remove():** aka "dequeue" Remove the front element and return.
- **peek()**: Examine the front element without removing it.
- **size():** how many items are stored in the queue?
- **isEmpty():** if 1 or more items in the queue returns true, false otherwise

Implementing a Queue with an Array

Queue ADT

state

Set of ordered items
Number of items

behavior

add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count of items is 0?

ArrayQueue<E>

state

```
data[]
Size
front index
back index
```

behavior

```
add - data[size] = value, if
out of room grow data
remove - return data[size -
1], size-1
peek - return data[size - 1]
size - return size
isEmpty - return size == 0
```

Big O Analysis

`remove()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	O(N) linear if you have to resize O(1) otherwise

```
add(5)
add(8)
add(9)
remove()
```

0	1	2	3	4
5	8	9		

```
numberOfItems = 3
front = 1
back = 2
```

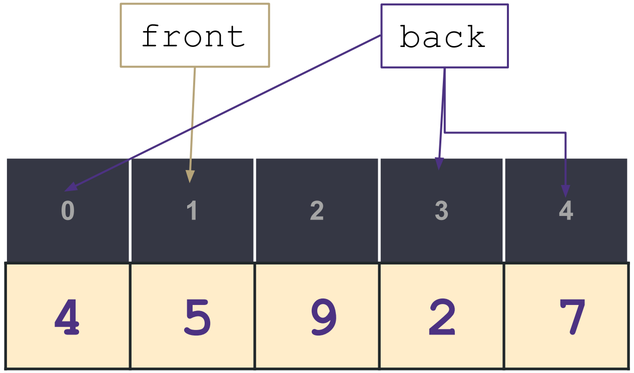
Implementing a Queue with an Array
> Wrapping Around

add(7)

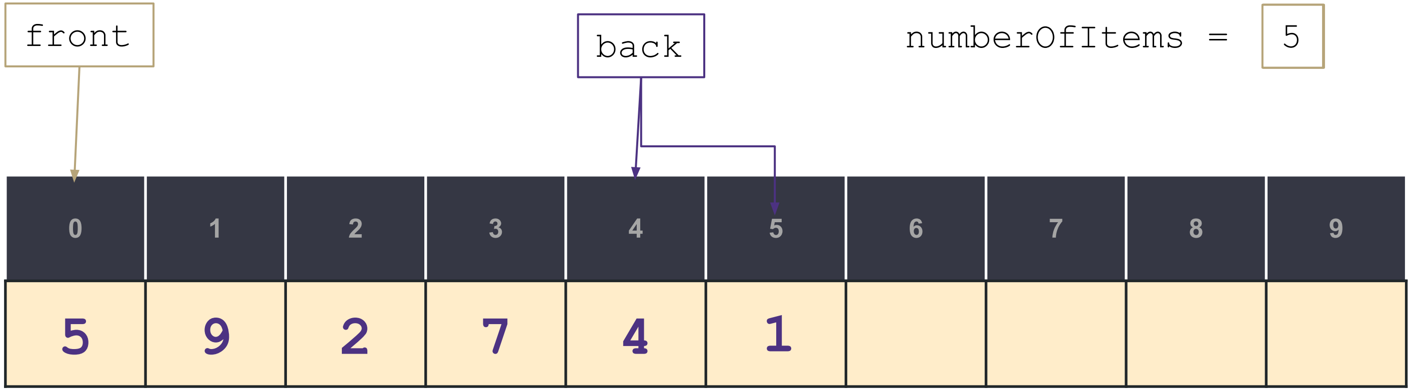
CSE 373 SP 18 - KASEY CHAMPION

add(4)

add(1)



numberOfItems = 5



Implementing a Queue with Nodes

Queue ADT

state

Set of ordered items
Number of items

behavior

add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count of items is 0?

LinkedQueue<E>

state

```
Node front
Node back
size
```

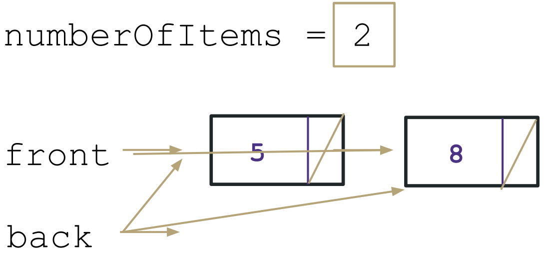
behavior

```
add - add node to back
remove - return and remove node at front
peek - return node at front
size - return size
isEmpty - return size == 0
```

Big O Analysis

`remove()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	O(1) Constant

```
add(5)
add(8)
remove()
```



Question 1

Given a string, determine if it has all unique Characters.

What if you can't use an additional data structure?

Solution 1

1. Brute Force: 2 for loops O(n^2)
2. Optimized: Use a Hashset O(n)
3. Optimized no additional data structure: sorting and go through O(n * logn)

Question 2

Given an array of integers, return indices of the two numbers such that they add up to a specific target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

Solution 2

- Brute Force: 2 for loops O(n^2)
- Optimized: 2 pass with Hash table O(n)
- Optimized: 1 pass with Hash table O(n)

```java
public int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
        int complement = target - nums[i];
        if (map.containsKey(complement)) {
            return new int[] { map.get(complement), i };
        }
        map.put(nums[i], i);
    }
    throw new IllegalArgumentException("No two sum solution");
}
```

Question 3

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

The left subtree of a node contains only nodes with keys less than the node's key.

The right subtree of a node contains only nodes with keys greater than the node's key.

Both the left and right subtrees must also be binary search trees.

Solution 3

1. Recursion

    One compares the node value with its upper and lower limits if they are available. Then one repeats the same step recursively for left and right subtrees.

    O(n)

```java
public boolean helper(TreeNode node, Integer lower, Integer upper) {
  if (node == null) return true;

  int val = node.val;
  if (lower != null && val <= lower) return false;
  if (upper != null && val >= upper) return false;

  if (! helper(node.right, val, upper)) return false;
  if (! helper(node.left, lower, val)) return false;
  return true;
}

public boolean isValidBST(TreeNode root) {
  return helper(root, null, null);
}
```

Solution 3

2. Iteration

    Use 3 stacks to convert the recursive solution to an iterative one.

    O(n)

```java
public void update(TreeNode root, Integer lower, Integer upper) {
  stack.add(root);
  lowers.add(lower);
  uppers.add(upper);
}

public boolean isValidBST(TreeNode root) {
  Integer lower = null, upper = null, val;
  update(root, lower, upper);

  while (!stack.isEmpty()) {
    root = stack.poll();
    lower = lowers.poll();
    upper = uppers.poll();

    if (root == null) continue;
    val = root.val;
    if (lower != null && val <= lower) return false;
    if (upper != null && val >= upper) return false;
    update(root.right, val, upper);
    update(root.left, lower, val);
  }
  return true;
}
```

Solution 3

2. Inorder traversal

Left -> Node -> Right order of inorder traversal means for BST that each element should be smaller than the next one.

O(n)

```java
public boolean isValidBST(TreeNode root) {
  Stack<TreeNode> stack = new Stack();
  double inorder = - Double.MAX_VALUE;

  while (!stack.isEmpty() || root != null) {
    while (root != null) {
      stack.push(root);
      root = root.left;
    }
    root = stack.pop();
    // If next element in inorder traversal
    // is smaller than the previous one
    // that's not BST.
    if (root.val <= inorder) return false;
    inorder = root.val;
    root = root.right;
  }
  return true;
}
```

Question 4

Given a linked list, rotate the list to the right by k places, where k is non-negative.

Question 4 Example

Input: 1->2->3->4->5->NULL, k = 2

Output: 4->5->1->2->3->NULL

Explanation:

rotate 1 steps to the right: 5->1->2->3->4->NULL

rotate 2 steps to the right: 4->5->1->2->3->NULL

Solution 4

Runner technique

Use two pointer p and q, q go k steps, p and q go together until q is at the end of the list. This way we find the node that is the new head.

O(n)

```java
public ListNode rotateRight(ListNode head, int k) {
    if (k==0 || head==null) return head;
    ListNode p=head,q=head;
    int step=0;
    while (step<k && q!=null) {
        q = q.next;
        step++;
    }
    if (q == null) {
        return rotateRight(head, k%step);
    } else {
        while (q.next != null) {
            p=p.next;
            q=q.next;
        }
        q.next=head;
        head=p.next;
        p.next=null;
    }
    return head;
}
```