Lecture 2: Stacks and Queues

CSE 373: Data Structures and Algorithms

Administrivia

Course Stuff

- Office hours are on class webpage: cs.washington.edu/373
- Piazza: https://piazza.com/class/jwcann1clfq7bn
  - Add code is on Canvas (or ask a staff member)

Project 0 Live!

- Individual assignment
- 14x content review
- GitLab/IntelliJ setup
  - You should have already gotten an automatic email with a link to your gitlab repo.
  - Check your spam folder

Project 1 out next week, partner project

- find your own partner
- Lecture, section, piazza, office hours

Last 5-10 minutes of section will be help with gitlab/intelliJ setup (if you're stuck bring your laptop and get some help.)

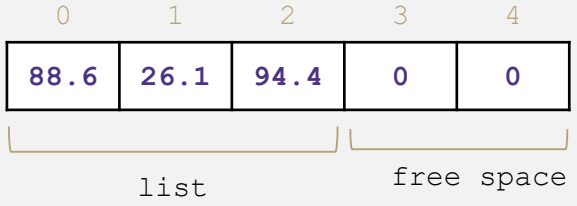
Warm Up

Q: Would you use a LinkedList or ArrayList implementation for each of these scenarios?

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
```



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
```

Situation #1: Write a data structure that implements the List ADT that will be used to store a list of songs in a 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.

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

Instructions

Take 3 Minutes

1. Introduce yourself to your neighbors ☺
2. Discuss your answers
3. Log onto Poll Everywhere
   1. Go to PollEv.com/cse373su19
   2. OR Text CSE373Su19 to 22333 to join session, text "1" "2" or "3" to select your answer
4. Get extra credit!

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

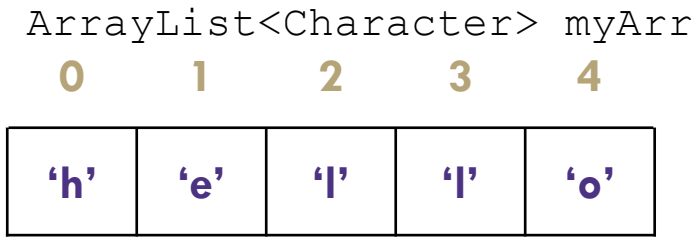
- If a method takes a constant number of steps (like 23 or 5) its running time is O(1)
- If a method takes a linear number of steps (like 4N+3) its running time is O(N)

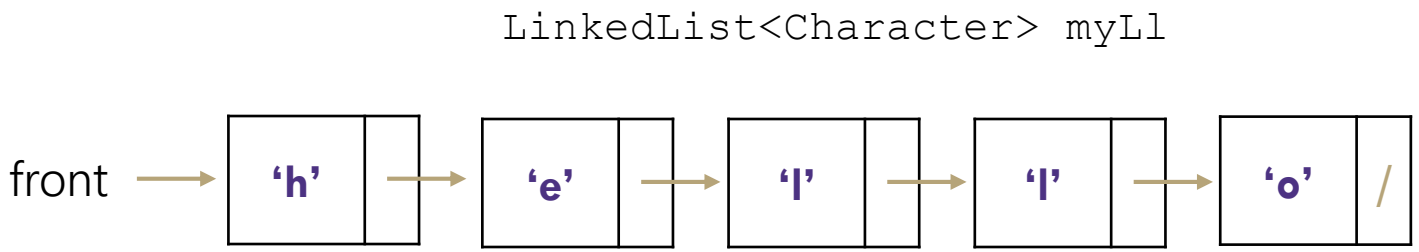
For ArrayLists and LinkedLists, what is the O() for each of these operations?

- 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!)

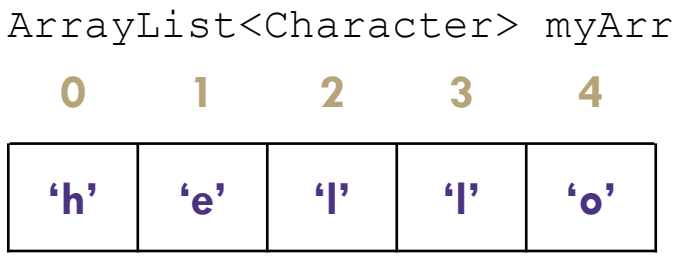
- ArrayList: O(N) linear time
- LinkedList: O(N) linear time

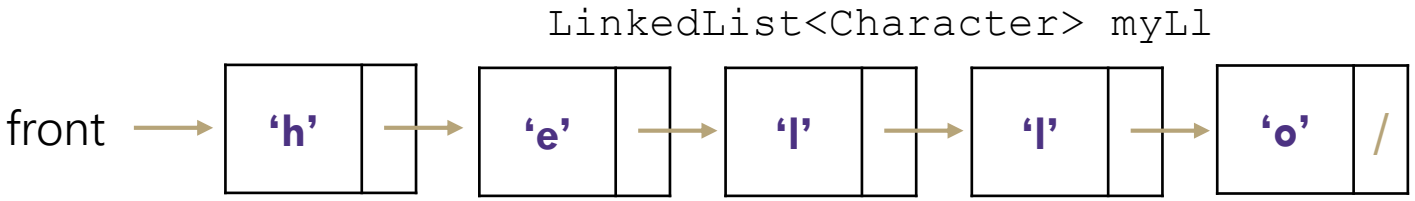
Amount of space used overall

- ArrayList: sometimes wasted space
- LinkedList: compact

Amount of space used per element

- ArrayList: minimal
- LinkedList: tiny extra





Goals for Today

Review Stacks, Queues

- What are the ADTs
- How can you implement both of them with arrays and with nodes?

Basics of Testing your code.

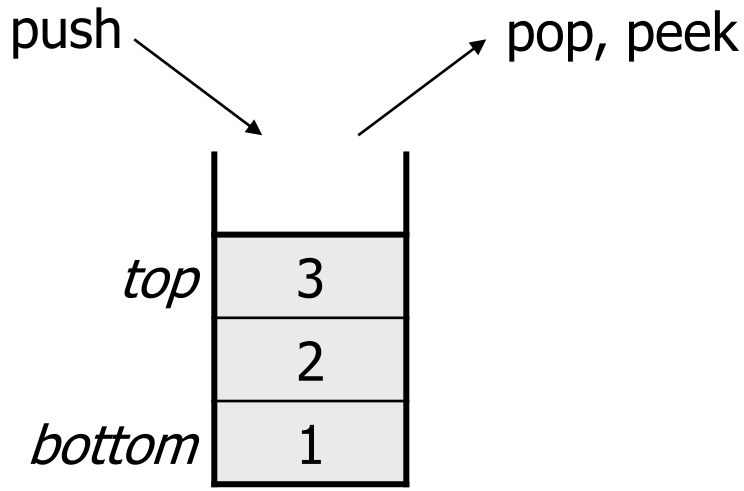
(maybe) Review Dictionaries.

- What is the ADT
- Can we implement well with arrays and nodes?

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()**: false if there are 1 or more items in stack, true 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[numItems] = value, if
out of room grow data, numItems++
pop return data[numItems - 1],
numItems-=1
peek return data[numItems - 1]
size return numItems
isEmpty return numItems == 0
```

Big O Analysis

`pop()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`push()`	Don't resize: O(1) Constant Do resize: O(N) linear

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

numItems = 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
```

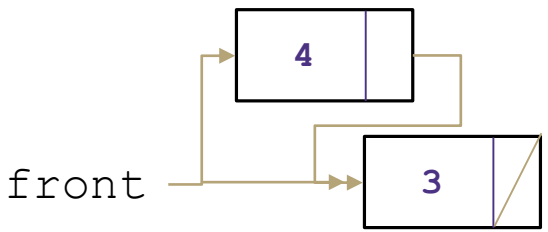
behavior

```
push add new node at top
numItems++
pop return and remove node at top, numItems-=1
peek return node at top
size return numItems
isEmpty return numItems == 0
```

Big O Analysis

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

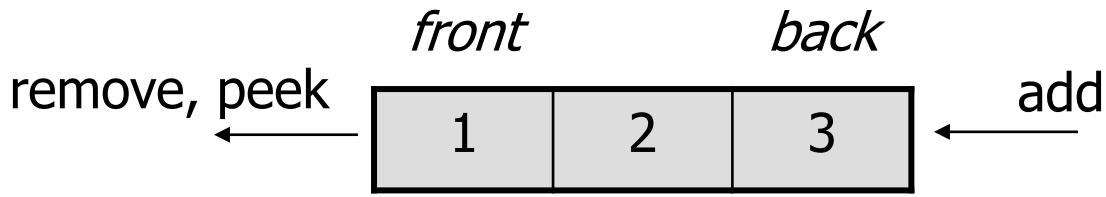
```
push(3)
push(4)
pop()
```



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[]
numItems
front index
back index
```

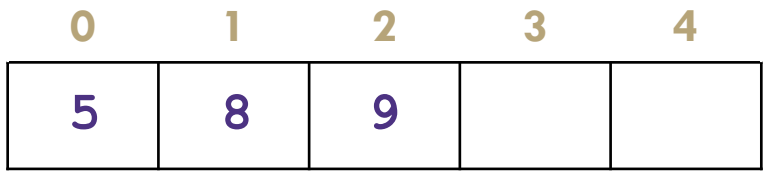
behavior

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

Big O Analysis

`remove()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	Don't resize: O(1) Constant Do resize: O(N) linear

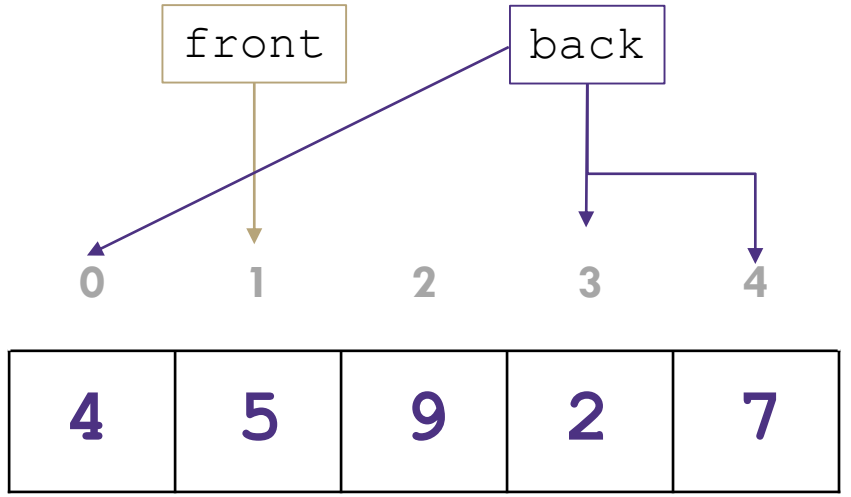
```
add(5)
add(8)
add(9)
remove()
```

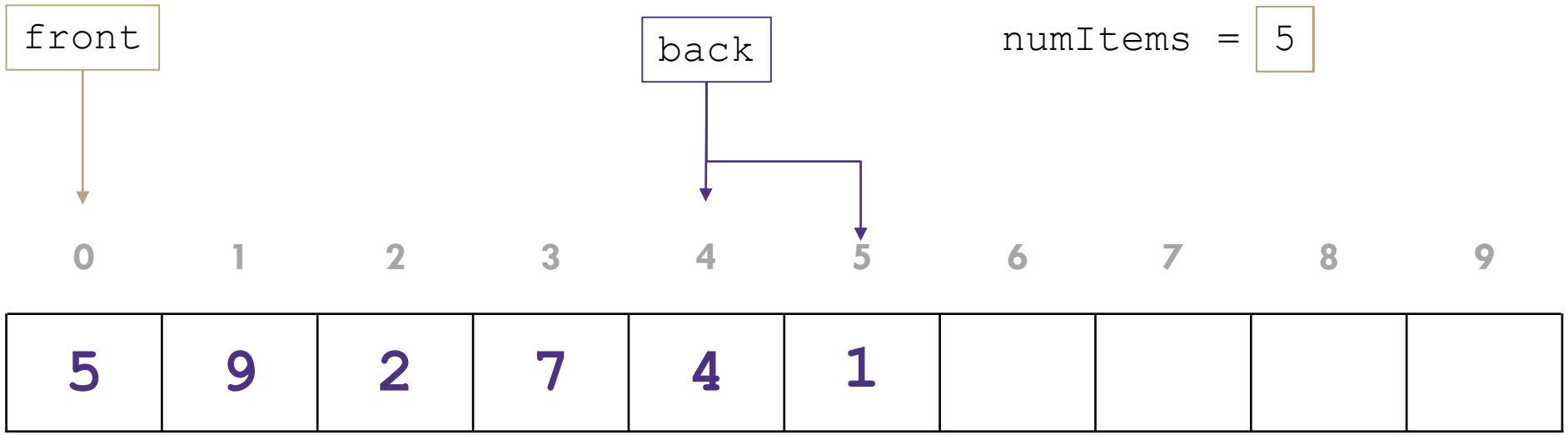


numItems = 3
front = 1
back = 2

Implementing a Queue with an Array
> Wrapping Around

add(7)
add(4)
add(1)





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
numItems
```

behavior

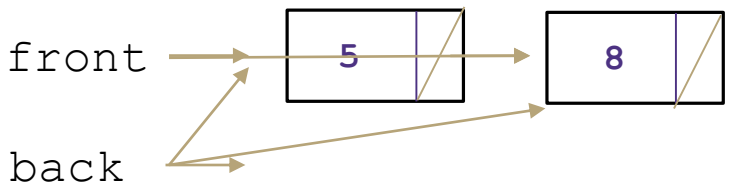
```
add - add node to back, numItems++
remove - return and remove node at
front, numItems--
peek - return node at front
size - return numItems
isEmpty - return numItems == 0
```

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()
```

`numItems =` 2



Multiple Levels of Design Decisions

Implementation Details

- Do we overwrite old values with `null` or do we leave the garbage value?
- Do we validate input and throw exceptions or just wait for the code to fail?

Data structure choice

- Do we use a LinkedList or an ArrayList?
- Do we use a Node-based queue implementation or an array-based implementation?

Choice of ADT

- Which of the ADTs that we've seen is the best fit?

(We'll see other kinds of design decisions later in the quarter).

Design Decisions

Take 3 Minutes

Discuss with your neighbors: For each scenario select the appropriate ADT and implementation to best optimize for the given scenario.

Situation #1: You are writing a program to manage a todo list with a specific approach to tasks. This program will order tasks for so that the **most recent** task is addressed first. You don't want to risk a long delay between submission of an item and its appearance.

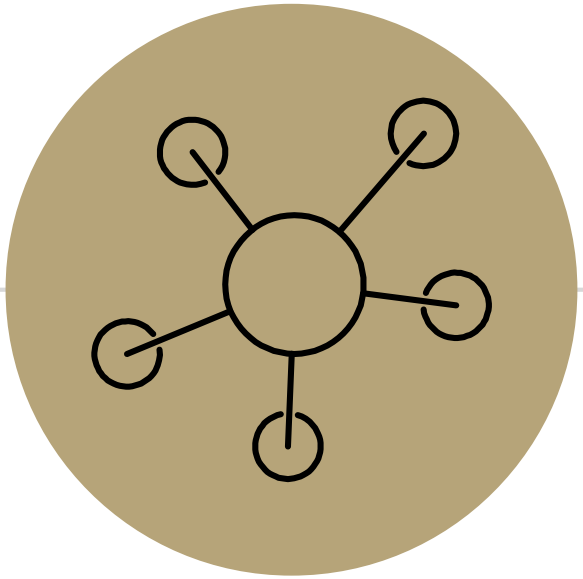
Stack – First in Last out

Nodes – make addition and removal of tasks very easy

Situation #2: You are writing a program to schedule jobs sent to a laser printer. The laser printer should process these jobs in the order in which the requests were received. The printer has very limited memory.

Queue – First in First out

Array – want to save the extra pointers to fit in our limited space



Testing Your Code

Testing: Why are we doing this?

The ability to test your own code is integral to an understanding of data structures.

- Differentiating between ADT requirements and design decisions you made.
- Coming up with test cases is one of the best ways to understand data structures more deeply
  - What cases will cause certain implementations to slow down?
  - How long do I expect certain operations to take?
  - What edge cases are there in the definition?
  - Where else might I find bugs?

In the real world, coding projects don't come with their own tests.

- You have to write your own.

You might be frustrated with us at some point for not giving you test cases.

- I understand. I was frustrated with my data structures professor when she didn't give us tests.
- Learning to test your own code is integral to maturing as a computer scientist.
- We're always tweaking things to make this as painless as we can.

Testing

Today: Strategies for generating tests. Ways to think about testing.

Thursday: Activity to practice our particular testing framework

Testing

Computers don't make mistakes- people do!

"I'm almost done, I just need to make sure it works"
– Naive 14Xers

Software Test: a separate piece of code that exercises the code you are assessing by providing input to your code and finishes with an assertion of what the result should be.

1. Isolate - break your code into small modules
2. Build in increments - Make a plan from simplest to most complex cases
3. Test as you go - As your code grows, so should your tests

Types of Tests

Black Box

- Behavior only – ADT requirements
- From an outside point of view
- Does your code uphold its contracts with its users?
- Performance/efficiency

White Box

- Includes an understanding of the implementation
- Written by the author as they develop their code
- Break apart requirements into smaller steps
- "unit tests" break implementation into single assertions

What to test?

Expected behavior
- The main use case scenario
- Does your code do what it should given friendly conditions?

Forbidden Input
- What are all the ways the user can mess up?

Empty/Null
- Protect yourself!
- How do things get started?
- 0, -1, null, empty collections

Boundary/Edge Cases
- First items
- Last item
- Full collections (resizing)

Scale
- Is there a difference between 10, 100, 1000, 10000 items?

Testing Strategies

You can't test everything
- Break inputs into categories
- What are the most important pieces of code?

Test behavior in combination
- Call multiple methods one after the other
- Call the same method multiple times

Trust no one!
- How can the user mess up?

If you messed up, someone else might
- Test the complex logic

5 Minutes

Thought Experiment

Discuss with your neighbors: Imagine you are writing an implementation of the List interface that stores integers in an Array. What are some ways you can assess your program's correctness in the following cases:

Expected Behavior
- Create a new list
- Add some amount of items to it
- Remove a couple of them

Forbidden Input
- Add a negative number
- Add duplicates
- Add extra large numbers
- Add something to index 10 of a size 3 list

Empty/Null
- Call remove on an empty list
- Add to a null list
- Call size on an null list

Boundary/Edge Cases
- Add 1 item to an empty list
- Set an item at the front of the list
- Set an item at the back of the list

Scale
- Add 1000 items to the list
- Remove 100 items in a row
- Set the value of the same item 50 times

JUnit

JUnit: a testing framework that works with IDEs to give you a special GUI when testing your code

```
@Test
public void myTest() {
    MyArrayList<String> basicAl = new MyArrayList<String>();
    basicAl.append("373 Rocks");
    assertThat(basicAl.get(0), is("373 Rocks"));
}
```

Assertions:

- `assertThat(thingYoureTesting, is(ExpectedResult))` is most common. Calls .equals() method
- May write your own helper methods here to check that internal state is identical.
- Other assertions exist; see official documentation, or our documentation on the webpage.

More: https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html

Review: Generics

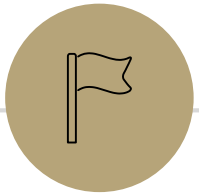
```
// a parameterized (generic) class
public class name<TypeParameter> {
    ...
}
```

- Forces any client that constructs your object to supply a type
  - Don't write an actual type such as String; the client does that
  - Instead, write a type variable name such as `E` (for "element") or `T` (for "type")
  - You can require multiple type parameters separated by commas
- The rest of your class's code can refer to that type by name

```
public class Box {
   private Object object;
   public void set(Object object) {
      this.object = object;
   }
   public Object get() {
      return object;
   }
}
```

```
public class Box<T> {
   private T t;
   public void set(T t) {
      this.t = t;
   }
   public T get() {
      return t;
   }
}
```

More details: https://docs.oracle.com/javase/tutorial/java/generics/types.html



Dictionaries

Dictionaries (aka Maps)

Every Programmer's Best Friend

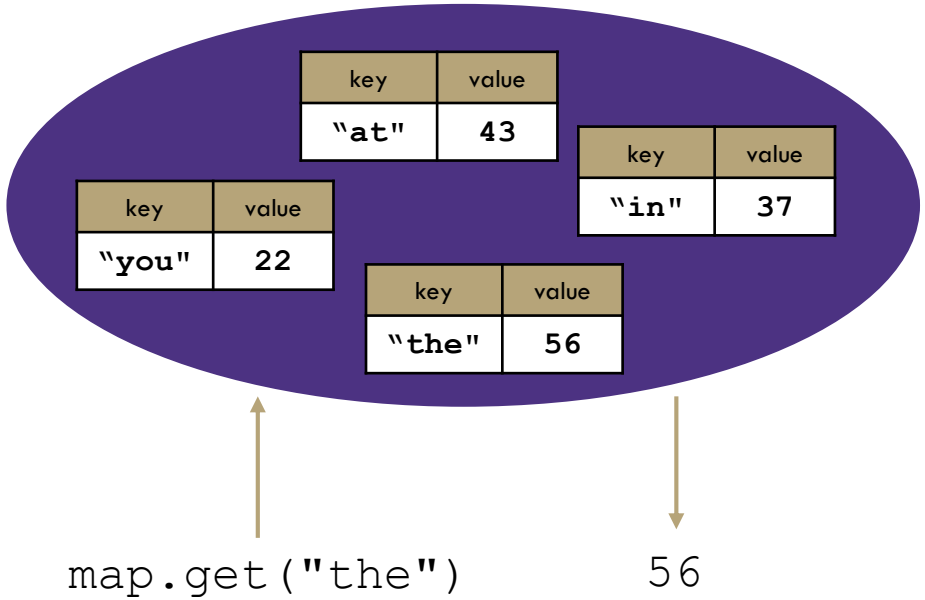
You'll use one in every single programming project.

- Because I don't think we could really design an interesting project that doesn't use one.

Review: Maps

map: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.

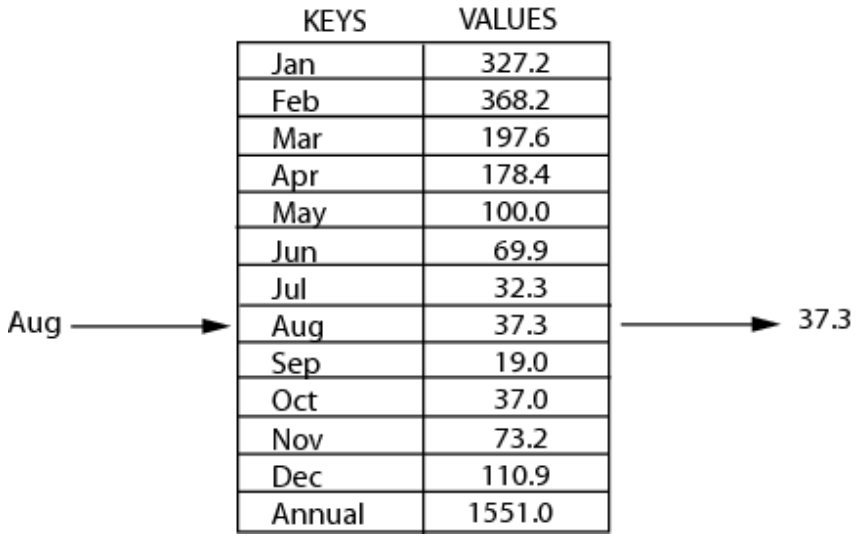
- a.k.a. "dictionary"



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



2 Minutes

Implementing a Dictionary with an Array

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

ArrayDictionary<K, V>

state

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

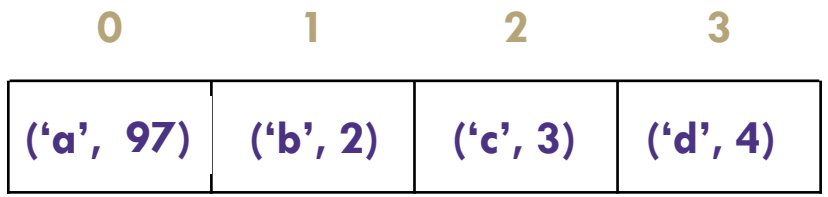
behavior

put 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

Big O Analysis

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

```
put('a', 1)
put('b', 2)
put('c', 3)
put('d', 4)
remove('b')
put('a', 97)
```



2 Minutes

Implementing a Dictionary with Nodes

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

LinkedDictionary<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

Big O Analysis

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

```
put('a', 1)
put('b', 2)
put('c', 3)
put('d', 4)
remove('b')
put('a', 97)
```