Lecture 3: Maps and Iterators

CSE 373: Data Structures and Algorithms

Warm Up

Take 3 Minutes

Q: Which ADT and data structure implementation would you choose if asked to implement a collection of comments for an Instagram post?

List ADT	Stack ADT	Queue 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	**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?	**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?

ArrayList – optimize for addition in order, the ability to remove regardless of position and update number of likes

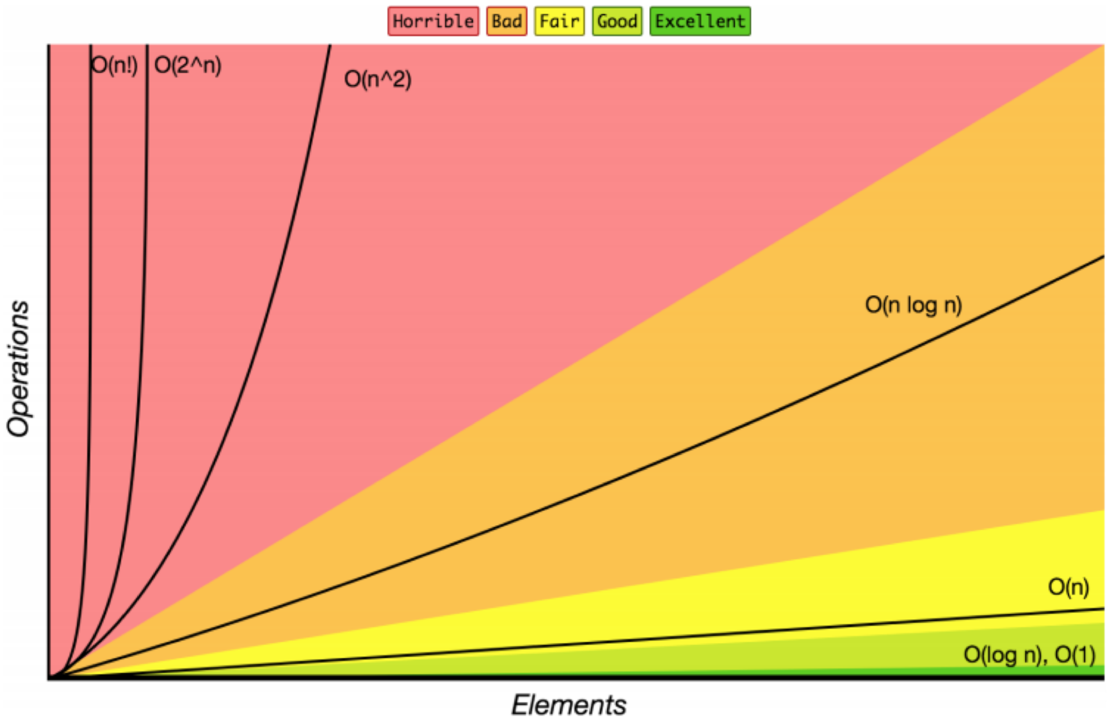
Extra Credit:
Go to PollEv.com/champk
Text CHAMPK to 22333 to join session, text “1” or “2” to select your answer

Administrivia

Review: Complexity Class

complexity class: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

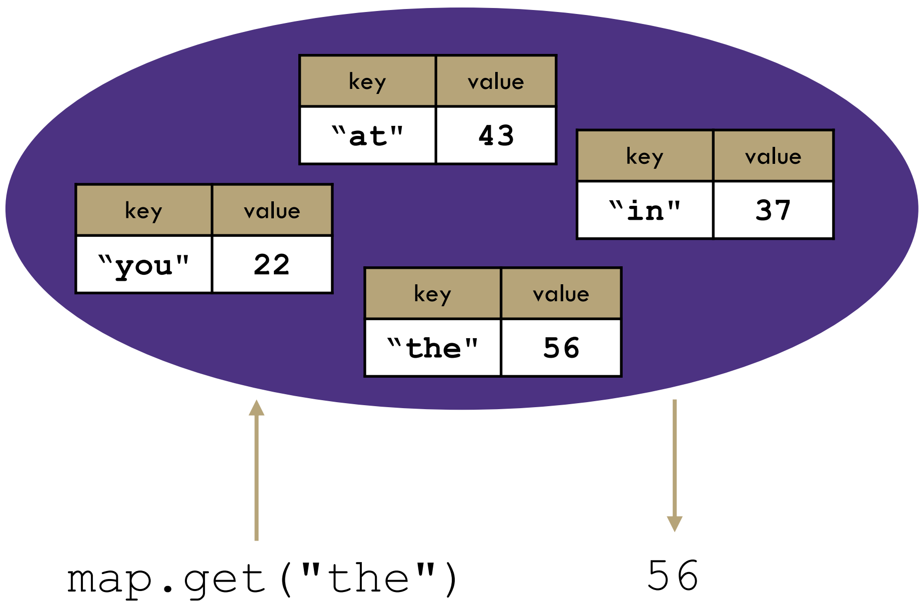
Complexity Class	Big-O	Runtime if you double N	Example Algorithm
constant	O(1)	unchanged	Accessing an index of an array
logarithmic	$O(\log_2 N)$	increases slightly	Binary search
linear	O(N)	doubles	Looping over an array
log-linear	$O(N \log_2 N)$	slightly more than doubles	Merge sort algorithm
quadratic	$O(N^2)$	quadruples	Nested loops!
...	...	...	...
exponential	$O(2^N)$	multiplies drastically	Fibonacci with recursion



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", "associative array", "hash"



Dictionary ADT

state

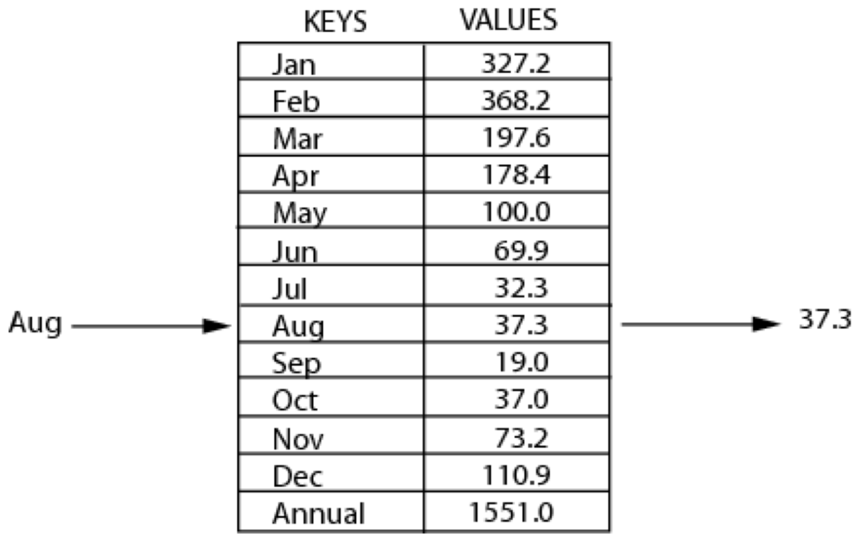
Set of items & keys
Count of items

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)
```

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

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`

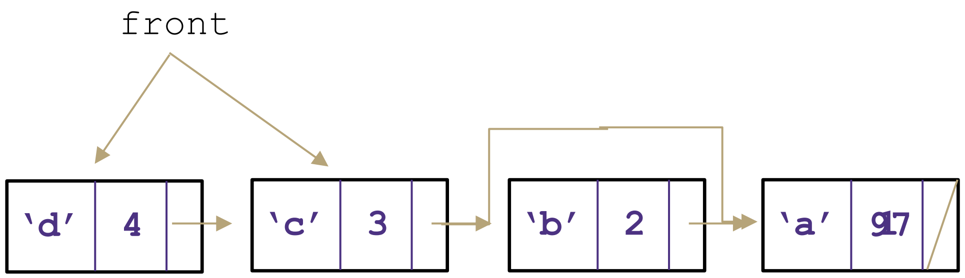
behavior

`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)
```



Design Decisions

Take 5 Minutes

Discuss with your neighbors: Which implementation of the Dictionary ADT would you choose if asked to implement each of the following situations? For each consider the most important functions to optimize.

Situation #1: You are writing a program to store incidents in a software service you maintain. The keys will be time stamps, so you know they will be unique. You will be adding incidents as they occur and removing them as they are resolved. You are more likely to need to access and remove incidents that have recently been added to the collection.

LinkedDictionary – optimize for addition and removal of incidents without need of examining entire data set reguarly

Situation #2: You are writing a program to store a rather small dictionary that maps exam questions to the average score for that question across all students. The questions are numbered sequentially starting at 0. Often you will want to read the entire set of scores in the order of the test.

ArrayDictionary – optimize for accessing all entries in set in specific order or individually

Traversing Data

Array

```
for (int i = 0; i < arr.length; i++) {
   System.out.println(arr[i]);
}
```

List

```
for (int i = 0; i < myList.size(); i++) {
   System.out.println(myList.get(i));
}
```

```
for (T item : list) {
   System.out.println(item);
}
```

Iterator!

Review: Iterators

iterator: a Java interface that dictates how a collection of data should be traversed. Can only move in the forward direction and in a single pass.

Iterator Interface
behavior hasNext() – true if elements remain next() – returns next element

supported operations:

hasNext() – returns true if the iteration has more elements yet to be examined

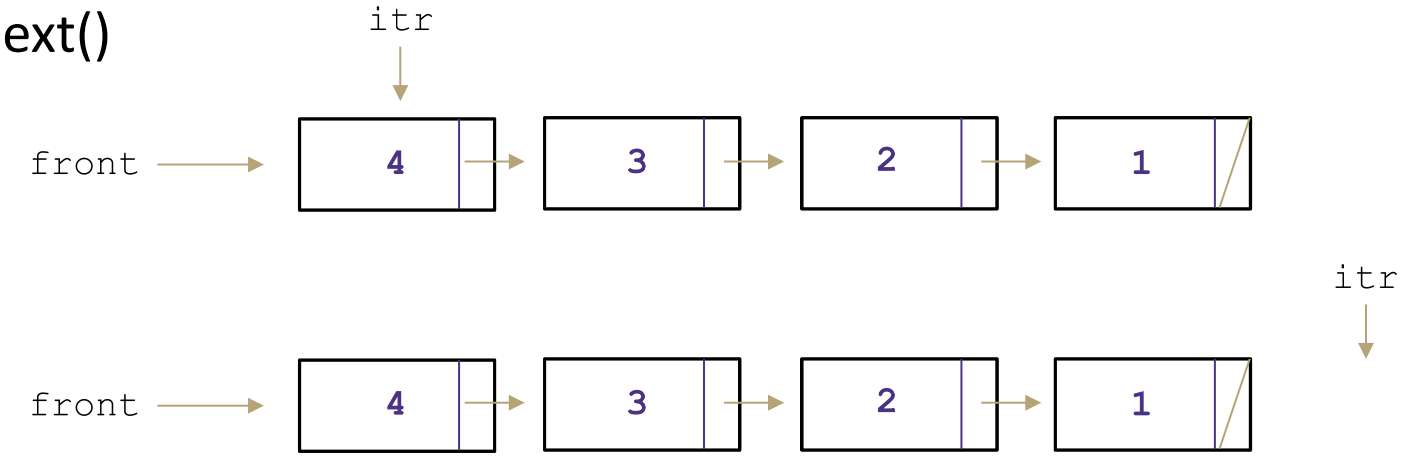
next() – returns the next element in the iteration and moves the iterator forward to next item

```
ArrayList<Integer> list = new ArrayList<Integer>();
//fill up list

Iterator itr = list.iterator();
while (itr.hasNext()) {
   int item = itr.next();
}
```

```
ArrayList<Integer> list = new ArrayList<Integer>();
//fill up list

for (int i : list) {
   int item = i;
}
```

Implementing an Iterator

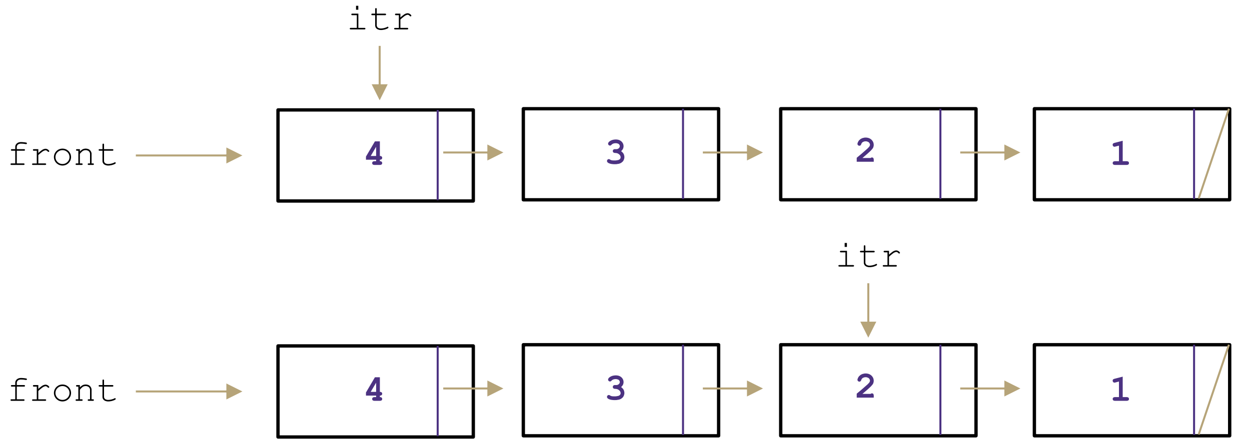
hasNext()



true

false

next()



4

2

Testing Your Code

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

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

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 experience when testing your code

```
@Test
public void myTest() {
   Map<String, Integer> basicMap = new LinkedListDict<String, Integer>();
   basicMap.put("Kasey", 42);
   assertEquals(42, basicMap.get("Kasey"));
}
```

Assertions:

- `assertEquals(item1, item2)`
- `assertTrue(Boolean expression)`
- `assertFalse(bollean expression)`
- `assertNotNull(item)`

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