Lecture 3: Maps and Iterators
Warm Up – Design Decisions

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

Situation #1: Syntax checker to determined correct alignment of Java code curly braces

LinkedStack – optimize for “sandwich” pattern of closing most recent sets first and possible reordering during development

Situation #2: Scheduling print jobs sent to a single printer by multiple users

ArrayQueue – optimize for maintaining order of requests received, possible cancellations and adhering to maximum queue size

Situation #3: The collection of comments left by users on a single Instagram post

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

Socrative:
www.socrative.com
Room Name: CSE373
Please enter your name as: Last, First
Course Announcements

Website is live
Discussion Board posted
HW 1 posted, due Friday Jan 18
Office Hours start next week
Sorry, still no add code
**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
# Implementing a Dictionary with an Array

## Dictionary ADT

<table>
<thead>
<tr>
<th>state</th>
<th>behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of items &amp; keys</td>
<td>put(key, item) add item to collection indexed with key</td>
</tr>
<tr>
<td>Count of items</td>
<td>get(key) return item associated with key</td>
</tr>
</tbody>
</table>

## ArrayDictionary\(<K, V>\>

<table>
<thead>
<tr>
<th>state</th>
<th>behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair(&lt;K, V&gt;&gt;[] data</td>
<td>put(key, item) create new pair, add to next available spot, grow array if necessary</td>
</tr>
<tr>
<td>size</td>
<td>get(key) scan all pairs looking for given key, return associated item if found</td>
</tr>
</tbody>
</table>

## Big O Analysis

<table>
<thead>
<tr>
<th>method</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>put()</td>
<td>O(n) linear</td>
</tr>
<tr>
<td>get()</td>
<td>O(n) linear</td>
</tr>
<tr>
<td>containsKey()</td>
<td>O(n) linear</td>
</tr>
<tr>
<td>remove()</td>
<td>O(n) linear</td>
</tr>
<tr>
<td>size()</td>
<td>O(1) constant</td>
</tr>
</tbody>
</table>

## Example

```java
ArrayDictionary<String, Integer> dictionary = new ArrayDictionary<>();

// Insert items
dictionary.put("a", 1);
dictionary.put("b", 2);
dictionary.put("c", 3);
dictionary.put("d", 4);

// Remove item
dictionary.remove("b");

// Update item
dictionary.put("a", 97);
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>('a', 97)</td>
<td>('b', 2)</td>
<td>('c', 3)</td>
<td>('d', 4)</td>
</tr>
</tbody>
</table>
```
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

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

**Big O Analysis**

- `put()` \(O(n)\) linear
- `get()` \(O(n)\) linear
- `containsKey()` \(O(n)\) linear
- `remove()` \(O(n)\) linear
- `size()` \(O(1)\) constant
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**

**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

---

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

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

---

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

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

hasNext()

front → 4 → 3 → 2 → 1
itr

true

next()

front → 4 → 3 → 2 → 1
itr

itr

false

front → 4 → 3 → 2 → 1
itr

4

front → 4 → 3 → 2 → 1
itr

itr

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
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

```java
@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(booleand expression)
- assertNotNull(item)

More: [https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html](https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html)