Sets, Maps, and Hash Tables

Sets

- Remember from discrete math: a **set** is a collection of unique, unordered elements
  - Unique: no two items are identical
  - Unordered: {1, 2, 3} is the same set as {3, 2, 1} or {2, 1, 3}
- Basic set operations:
  - **Add** a new element to the set
  - Determine if the set **contains** a particular element
  - **Remove** an existing element from the set

Implementing a set

- Using an array or linked list:
  - **Add**: check all existing elements in the list to ensure the new one doesn't already exist, then insert the new element
  - **Contains**: search the list until the element is found
  - **Remove**: search the list until the element is found, then delete it
- All of these operations are $O(n)$

Implementing a set

- Using a binary search tree:
  - **Add**: insert the new element into the tree (duplicate checking is already incorporated into the tree's add method)
  - **Contains**: search the tree until the element is found
  - **Remove**: search the tree until the element is found, then delete it
- All of these operations are $O(\log n)$, which is significantly better than the $O(n)$ of the list implementation!

Maps

- A map is a set of **key**-**value** pairs
  - Also known as a **dictionary** or **associative array**
  - Every **key** is associated with a particular **value**
  - Every key in a map must be unique (although two or more keys can be associated with the same value)
- Basic operations:
  - **Add** a new key-value pair to the map
  - **Get** the value associated with a particular key
  - **Remove** an existing key-value pair from the map

Maps

- Think of a map as a generalization of an array
  - In an array, each value is associated with an integer index. You use that index to access the value: **names[4]** accesses the value at index 4
  - In a map, each value is associated with a key (of any data type). You use that key to access the value: **names.get("901-867-5309")** accesses the value associated with the string **"901-867-5309"**

Implementing a map

- Using an array/linked list:
  - Each list element is a key-value pair
  - **Add**: check all existing keys to ensure the new key does not already exist, then add the new key-value pair
  - **Get**: search the list for the specified key, then return its associated value
  - **Remove**: search the list for the specified key, then remove that key-value pair
- These are all $O(n)$ operations

Implementing a map

- Using a binary search tree:
  - Each tree element is a key-value pair
  - **Add**: add the new key-value pair to the tree (the add method already accounts for duplicates)
  - **Get**: search the tree for the specified key, then return its associated value
  - **Remove**: search the tree until the key is found, then delete that key-value pair from the tree
- These are all $O(\log n)$ operations

Implementing a map

- Once again, the BST implementation is significantly more efficient – $O(\log n)$ vs. $O(n)$ – than the array/linked list implementation
- But we can do better!!
- Using a **hash table** enables ***O*(1)** performance for map operations
  - This is huge! Constant time means that we can expect the same performance regardless of whether our map contains 10 or 10,000,000 key-value pairs
  - Naturally, the $O(1)$ does come with a few caveats

Hash tables

- Basic idea: translate arbitrary **keys** (of any data type) into **array indices** using a **hash function**

Hash tables: simple example 1

- Let’s say we want to store positive integers into a hash table that contains 7 spots (indices 0-6)
- A straightforward hash function to translate your original number to a valid index is just to mod it by the length of the array:

hash(*n*) = *n* % data.length

Hash tables: simple example 1

- To store the number 74:
  hash(74) = 74 % 7 = 4
  So 74 gets placed at index 4

- To store the number 17:
  hash(17) = 17 % 7 = 3
  So 17 gets placed at index 3

Hash tables: simple example 1

- To retrieve 74 from the hash table:
  hash(74) = 74 % 7 = 4
  So we look at index 4 to see if we can find 74 (yes)

- To retrieve 20 from the hash table:
  hash(20) = 20 % 7 = 6
  So we look at index 6 to see if we can find 20 (no)

Hash tables: simple example 2

- Let's say we want to store strings (containing any characters, and of any length) into a hash table containing 5 spots (indices 0-4)
- Here's a possible hash function that would work:

hash(*n*) = (length of *n*) % data.length

Note: Java's **Object** class defines a **hashCode()** method that returns an **int** value, based on the object's memory address. This method can be overridden by any subclass of **Object** to determine how instances of that class should be "translated" into an integer.

Hash tables: simple example 2

- To store the string “super sloth”:

  hash(“super sloth”) = 11 % 5 = 1

  So “super sloth” gets placed at index 1

- To store the string “sad sloth”:

  hash(“sad sloth”) = 9 % 5 = 4

  So “sad sloth” gets placed at index 4

Hash tables: simple example 2

- To retrieve “super sloth” from the hash table:
  hash(“super sloth”) = 11 % 5 = 1
  So we look at index 1 to see if we can find “super sloth” (yes)

- To retrieve “uber sloth” from the hash table:
  hash(“uber sloth”) = 10 % 5 = 0
  So we look at index 0 to see if we can find “uber sloth” (no)

Choosing a hash function

- Note that the hash function is very important! It must be run whenever a new element is added to the hash table, and whenever we want to retrieve an element from the hash table
- The hash function must be efficiently computable – ideally it will not depend on the number of elements in the hash table
- Ideally the hash function will also result in minimal **collisions** (discussed next)

Collisions

- We have not considered what happens when two things hash to the same index
  - In Simple Example 1, the numbers 7 and 14 are both hashed to index 0
  - This is called a **collision** and must be handled somehow!
- Two ways to resolve collisions:
  - **Open addressing**
  - **Chaining**

Open addressing

- When adding a new element and a collision occurs, simply find another open spot in the table
- Under the **linear probing** scheme, we just search the indices sequentially:
  - If the item hashes to index 5 and that's not available, check index 6
  - If index 6 is not available, check index 7
  - If index 7 is not available, check index 8
  - And so on… (wrap around back to index 0 if necessary)

Open addr.: adding elements

- Using linear probing with the table from Simple Example 1: Indices in red are the ones that are checked

Initial table (empty)

[0]	
[1]	
[2]	
[3]	
[4]	
[5]	
[6]	

After adding 20 (hashes to 6)

[0]	
[1]	
[2]	
[3]	
[4]	
[5]	
[6]	20

After adding 18 (hashes to 4)

[0]	
[1]	
[2]	
[3]	
[4]	18
[5]	
[6]	20

After adding 27 (hashes to 6, placed at 0)

[0]	27
[1]	
[2]	
[3]	
[4]	18
[5]	
[6]	20

Open addr.: adding elements

- Using linear probing with the table from Simple Example 1: Indices in red are the ones that are checked

[0]	27
[1]	34
[2]	
[3]	
[4]	18
[5]	
[6]	20

After adding 34 (hashes to 6, placed at 1)

[0]	27
[1]	34
[2]	21
[3]	
[4]	18
[5]	
[6]	20

After adding 21 (hashes to 0, placed at 2)

[0]	27
[1]	34
[2]	21
[3]	
[4]	18
[5]	33
[6]	20

After adding 33 (hashes to 5)

[0]	27
[1]	34
[2]	21
[3]	25
[4]	18
[5]	33
[6]	20

After adding 25 (hashes to 4, placed at 3)

Open addr.: retrieving elements

- To retrieve an element from a hash table that uses open addressing, follow the same linear probing procedure as adding a new element
  - If linear probing leads us to an empty spot in the table, that element does not exist in the table

- Example: Find the elements 21 and 103 in this hash table:

[0]	27
[1]	34
[2]	21
[3]	
[4]	18
[5]	33
[6]	20

Open addr.: retrieving elements

- To find 21, start by hashing 21 using our hash function: 21 % 7 = 0
  - Look at index 0. That's not 21, so move to the next index
  - Look at index 1. That's not 21, so move to the next index
  - Look at index 2. 21 found!

[0]	27
[1]	34
[2]	21
[3]	
[4]	18
[5]	33
[6]	20

Indices in red are the ones that are checked

Open addr.: retrieving elements

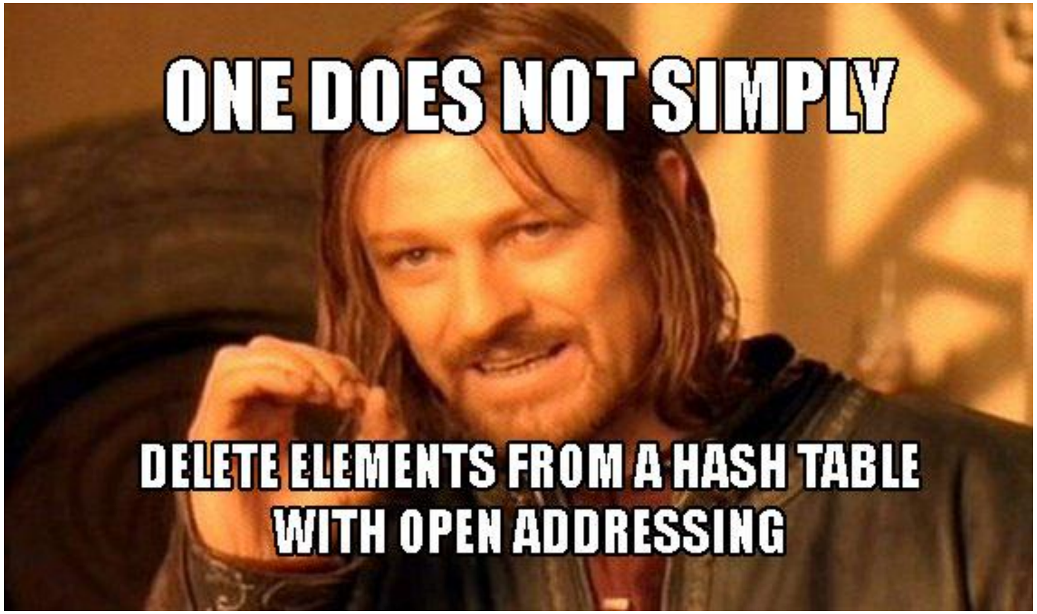
- To find 103, start by hashing 103 using our hash function: 103 % 7 = 5
  - Look at index 5. That's not 103, so move to the next index
  - Look at index 6. That's not 103, so move to the next index
  - Repeat with indices 0, 1, and 2. None of them are 103
  - Once we get to index 3 (which is empty), we know 103 cannot be in the table

[0]	27
[1]	34
[2]	21
[3]	
[4]	18
[5]	33
[6]	20

Open addr.: deleting elements

- We must be careful when deleting elements from a hash table with open addressing!
- Suppose we want to delete the 27 from this hash table:

[0]	27
[1]	34
[2]	21
[3]	
[4]	18
[5]	33
[6]	20



Open addr.: deleting elements

[0]	27
[1]	34
[2]	21
[3]	
[4]	18
[5]	33
[6]	20

Original table

[0]	
[1]	34
[2]	21
[3]	
[4]	18
[5]	33
[6]	20

New table, after naively removing the 27

Now let's say we wanted to retrieve the number 21 from the table. We start by hashing 21 using our hash function: 21 % 7 = 0. We would go to index 0, see that it's empty, and incorrectly conclude that 21 is not in the table!

Open addr.: deleting elements

[0]	27
[1]	34
[2]	21
[3]	
[4]	18
[5]	33
[6]	20

Original table

[0]	D
[1]	34
[2]	21
[3]	
[4]	18
[5]	33
[6]	20

New table, after marking the 27 as deleted

Instead we just replace the 27 with a dummy item to indicate it's been deleted.

Now, to retrieve 21 we would follow the same linear probing procedure as before, which would start at index 0 and eventually lead us to index 2.

This means "deleting" an item from a hash table using open addressing doesn't really free any memory!

Rehashing

- What happens once the table becomes full?
  - Make a new larger table, copy the old elements over (just like an array list)
- But once more, we have to be careful! We can't just copy the existing elements over directly
- In the following table, suppose we decided to make a new table of length 9 after the original table was filled

[0]	27
[1]	34
[2]	21
[3]	25
[4]	18
[5]	33
[6]	20

Rehashing

[0]	27
[1]	34
[2]	21
[3]	25
[4]	18
[5]	33
[6]	20

Original table (full)

[0]	27
[1]	34
[2]	21
[3]	25
[4]	18
[5]	33
[6]	20
[7]	
[8]	

New table, after copying old elements directly

Now let's say we wanted to retrieve the number 34 from the new table. With the new table, 34 would be hashed to 34 % 9 = 7. We would go to index 7, see that it's empty, and incorrectly conclude that 34 is not in the table!

Rehashing

[0]	27
[1]	34
[2]	21
[3]	25
[4]	18
[5]	33
[6]	20

Original table (full)

[0]	27
[1]	18
[2]	20
[3]	21
[4]	
[5]	
[6]	33
[7]	34
[8]	25

New table, after rehashing old elements

Instead, all the old elements must be **rehashed** (i.e., passed through the hash function again) to correctly place them into the new table:

27 % 9 = 0
34 % 9 = 7
21 % 9 = 3
25 % 9 = 7 (collision, placed at 8)
18 % 9 = 0 (collision, placed at 1)
33 % 9 = 6
20 % 9 = 2

Note: Rehashing should also exclude any deleted dummy items from being in the new table

Analysis of open addressing

- With a $O(1)$ hash function and no collisions, open addressing allows $O(1)$ insertion and retrieval from the hash table
- But every time a collision occurs, we need to search for an empty spot. Worst case: the item hashes to index n, and index $n - 1$ is the only remaining spot in the table. This requires searching through the entire table, which is a $O(n)$ operation!

Analysis of open addressing

- To reduce collisions and improve performance, we should rehash the table well before it gets completely full
- Your textbook (as well as the Java API) suggests rehashing the table as soon as the **load factor** reaches 0.75
  - Load factor = (number of occupied spots in the table) / (total number of indices in the table)
  - "Occupied spots" includes both existing and deleted elements
- Why not rehash at a load factor of 0.25 or 0.50 instead?

Chaining

- **Chaining** is an alternative to open addressing
- Each spot in the hash table can store not just one element, but rather a <u>list</u> of elements

[0]	list 0
[1]	list 1
[2]	list 2
[3]	list 3
[4]	list 4
[5]	list 5
[6]	list 6

Chaining

- To **add** a new element:
  - Hash the element to determine its table index
  - Add the element to the list at that index
- To **retrieve** an element:
  - Hash the element to determine its table index
  - Search the list at that index for the element
- To **delete** an element:
  - Hash the element to determine its table index
  - Delete the element from the list at that index (note that no dummy item is needed here!)

Analysis of chaining

- Collisions in chaining affect only the index where the collision occurred (unlike in open addressing, where collisions affect the area of the table around the collision index)
- In general:
  - For low load factors, open addressing and chaining have similar performance
  - As the load factor increases, chaining offers superior performance

Maps and hash tables

- To implement a map using a hash table:
  - Each hash table element is a key-value pair, but the hashing is done only on the keys
  - **Add**: add the new key-value pair to the table, being sure to check for duplicate keys
  - **Get**: hash the key to find it in the table, then return the associated value
  - **Remove**: hash the key to find it in the table, then delete the key-value pair from the table
- Assuming a $O(1)$ hash function and minimal collisions, these are all $O(1)$ operations

Java API classes

- Sets:
  - **java.util.TreeSet**: based on a red-black tree (a red-black tree is a type of BST that ensures it always remains balanced – more on this in Ch. 9!)
  - **java.util.HashSet**: based on a hash table
  - Both implement the **java.util.Set** interface
- Similar story for maps:
  - **java.util.TreeMap**, **java.util.HashMap** (both implement the **java.util.Map** interface)