

A decorative graphic on the left side of the slide consisting of two overlapping parallelograms. The front one is blue and the back one is a light green color. They are positioned diagonally, with the blue one in front of the green one.

SWEN 601

Software Construction

Hashing Data Structures



Activity: Getting Started

1. Begin by accepting the GitHub Classroom invitation for today's homework.
 - a. The project may already contain some code!
2. Create a session package. This is where you will write your solutions to today's activities.
3. Create a homework package. This is where you will implement your solution to the homework.

When you submit your homework, you will include your activities. You may earn up to a 10% bonus on the homework if you have completed all of the activities.

Do not submit code that **does not compile**. Comment it out if necessary.

Next Two Weeks

WEEK 04	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ					Quiz #13		
LECTURE			Practicum 2		hashCode & Hashtables		
HOMEWORK					Hwk 14 Assigned	Hwk 13 Due (11:30pm)	

WEEK 05	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #14		Quiz #15		
LECTURE			Binary Trees & Binary Search Trees		Heaps, N-ary Trees, & B-Trees		
HOMEWORK	Hwk 14 Due (11:30pm)		Hwk 15 Assigned		Hwk 16 Assigned	Hwk 15 Due (11:30pm)	

Abstract Data Structures (ADTs) Recap

- An **array*** is the most basic data structure.
 - A fixed length - its capacity never changes.
 - Stored in a contiguous block of memory.
 - Provides random (non-sequential) access by **index**.
- A **stack** is a FILO (First-In, Last-Out) data structure.
 - Elements are **pushed** onto the top of the stack.
 - Elements are **popped** from the top of the stack.
 - **Peek** returns but does not remove the top element.
- A **queue** is a FIFO (First-In, First-Out) data structure.
 - Elements are **enqueued** at the back of the queue.
 - Elements are **dequeued** from the front of the queue.
 - **Peek** returns but does not remove the front element.
- A **list** is similar to an array.
 - A list is not fixed length; its capacity increases as elements are added.
 - A list provides random (non-sequential) access by **index**.

Each ADT **defines** but does not **implement** its behavior from the perspective of its user.

There are many possible **implementations** of each ADT. We have experimented with array-based and node-based implementations.

Once you are familiar with the implementation, you may begin using the Java equivalent classes in your assignments.

** Arrays are not abstract data types, but were included here for completeness.*



Java ADT Implementations

By this point you have implemented each of the data structures below, and should feel free to use the Java implementations from now on.

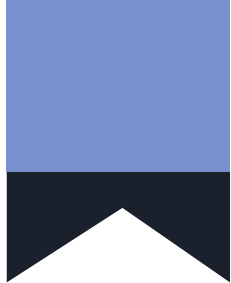
	Interface	Implementation(s)	Comments
Stack	NA	<u>java.util.Stack</u>	There is no stack interface in Java. It is array-based.
Queue	<u>java.util.Queue</u>	<u>java.util.LinkedList</u>	The Queue interface defines add (enqueue) and poll (dequeue) methods. LinkedList is the provided implementation, meaning that random access is possible.
List	<u>java.util.List</u>	<u>java.util.LinkedList</u>	Node-based implementation.
		<u>java.util.ArrayList</u>	Array-based implementation.

Problem: Library Patrons

- Libraries need to keep track of their patrons.
 - A unique ID.
 - A name.
 - Books borrowed.
 - Fines accrued.
- Today we will implement a system to help keep track of library patrons. It will need a data structure to keep track of library patrons.
 - We'll need to be able to use an ID to find students.
 - We'll need to be able to add and remove borrowed books.
 - We'll need to be able to add or pay off fines.

Let's start by making a class for ***library patrons***.



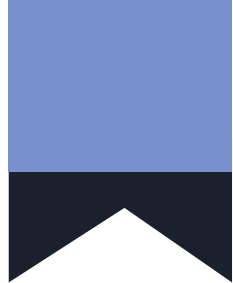


Warm Up: A Patron Class I

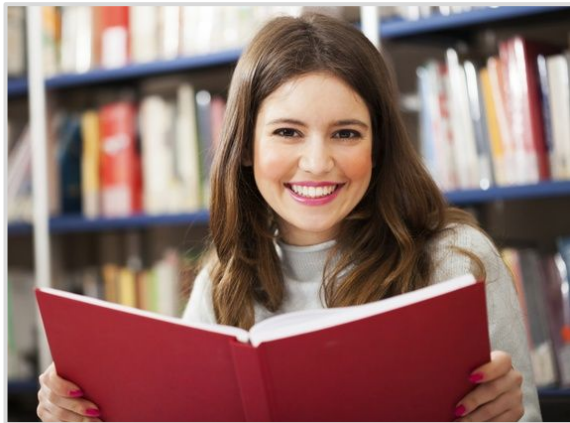


In your session package, write a new Java class to represent a library Patron. All Patron have:

- An ID in the format “ABC1234” (3 letters and 4 digits).
- A name.
- A list of the names of borrowed books (use Java’s ArrayList).
- Accumulated fines.



Warm Up: A Patron Class II



The Patron class will need a few methods.

- The usual getters and setters.
- We'll need to add or remove books from the borrowed list.
- We'll need to be able to add or remove fine amounts.
- While we're at it, how about a toString method as well?



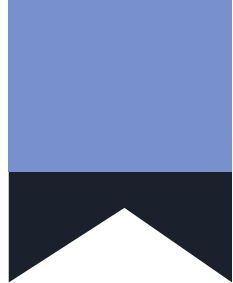
The Library

- The library will need to manage patrons and provide at least the following functions:
 - **void** addPatron(Patron p) - adds a new Patron to the library.
 - Patron getPatron(String id) - given an ID, finds and returns the patron. If no such Patron exists, return null.
 - **void** addBook(String id, String name) - adds a book to the patron's list of checked out books.
 - **void** addFine(String id, **float** fine) - adds the specified amount to the Patron's fines.
- We will need to store and retrieve the Patrons using a data structure. Given the data structures that we have available to us so far (array, stack, queue, list), which data structure makes the most sense?

Things to Consider

1. Order doesn't matter.
2. We need random access.
3. We do not know how many patrons will join the library.

Q: Which ADT most closely fits all of our needs?

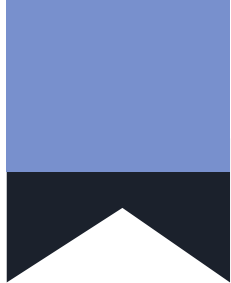


Activity: A Library Class I



The *list* data structure is the only one that meets all of our needs.

- Add a Library class that uses a *list* to store Patron objects.
- Implement the addPatron(Patron p) method.
- Implement the getPatron(String id) method.



Activity: A Library Class II



Now let's add a couple of convenience methods to the Library.

- Implement the method to add a book to the patron's list of borrowed books.
 - `addBook(String id, String title)`
- Implement the method to add some amount to the patron's accrued fines.
 - `addFine(String id, float fine)`
- You should use your `getPatron(String id)` function to find the right patron!



Complexity

- Let's examine the complexity of each of our functions.
 - addPatron - adding to the end of a list is a constant time operation: $O(1)$.
 - getPatron - requires a **linear search** over the list to find the Patron with the correct ID: $O(N)$.
 - addBook - uses getPatron: $O(N)$.
 - addFine - uses getPatron: $O(N)$.
- It seems like using a list to manage library patrons using a unique ID requires a lot of $O(N)$ operations.
 - Most functions require a linear search of the list to find the right student, making the getPatron method a bottleneck that is slowing down the other operations.
- As the library gains more patrons, all of these operations will take more and more time to run.
- Perhaps there is a better alternative?

Right now there is no relationship between the **patron** and the **index** at which that patron is stored in the list.

This means that the only way to find the right patron is to perform a linear search over the list.

Q: Is there some way that we can quickly map an ID to a specific index in the list? What if we used the ID as an index?



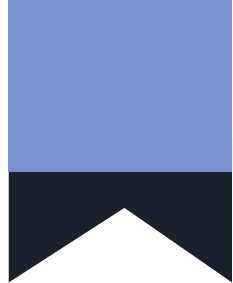
Map

- A Map stores a set of *key/value* pairs.
- A Map provides at least these basic operations:
 - **put** - add a key/value pair to the map. If the key is already in use, the new value replaces the old value.
 - **get** - returns the value associated with the specified key (or null if the key isn't in the map).
 - **size** - returns the number of key/value pairs in the map.
- If implemented properly, a Map can put and get key/value pairs in *constant time*.

K/V

Maps are appropriate to use if the items are identified by some key, such as a name or an ID.

We need a data structure just like this for our library! Given a patron ID, find the correct patron.

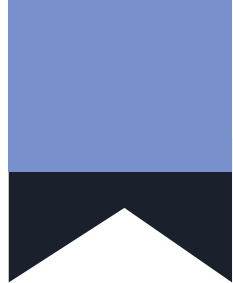


Activity: A Map Interface

K/V

Write an interface to represent a Map data structure.

- There should be at least one method for each operation:
 - **Put** a key/value pair into the map.
 - Given a key, **get** a value from the map.
 - Get the **size** of the map.
- Make sure that your interface is **generic**.
 - It should declare **two type parameters** - one each for the key (e.g. K) and the value (e.g. V).



Activity: An Entry Class

K/V

A Map must necessarily store **both** the key **and** the value. This is usually done by encapsulating both in a single object called an **entry**.

- Write a class called Entry.
 - It should declare **two type parameters** - one each for the key (e.g. K) and the value (e.g. V).
 - It should declare fields for each as well as implement an initializing constructor.
 - Accessors are needed as well!



Hash Maps

- As with most ADTs, a Map can be implemented in a number of ways.
- A *hashmap* is one such implementation.
 - A hashmap uses an *array* to store *key/value* pairs.
 - A *hash function* is used to translate each key into a *hash code*.
 - The *hash code* is used to find an *index* for the *key/value* pair.
- Provided that the *hash function* runs in constant time, the expected performance of the put and get functions on a hashmap is *constant time*.
- However, the order of the keys is not maintained.
 - This means that keys may be retrieved from the hashmap in a different order than they were added to it.

K/V

Today we will begin implementing a hashmap step-by-step.

By the time that we finish, we should be able to store and retrieve patrons using an ID in constant time.



Activity: Begin Implementing a Hashmap

Let's begin writing our first Map!

- Your class should be called Hashmap and it should be *generic*.
 - It should include **two** type parameters: one each for the **key** (K) and **value** (V).
 - The same parameters should be used when implementing the Map interface.
- It will need an array to store Entries.
 - The array type will be Entry<K, V>.
- Write a constructor that creates an array large enough to hold 100 entries.
- Stub out the put and get methods.

K/V



Hash Functions

- A *hash function* is a *one way* function that, given some *non-numeric input*, produces a number as output.
- That's a lot of information right there. Let's break it down.
- A *one way* function is one that works only in one direction.
 - Given some *input value*, it will produce an *output value*.
 - But given an *output value* the function cannot determine the original *input value*.
- A *hash function* produces a numeric value.
- Given the same input value a hash function will always produce the same output value.
- This process is called *hashing* and the output is called a *hash code*.
- Types that can compute their own hash codes are referred to as *hashable*.
 - More on that later.

This is all a fancy way of saying that a *hash function* can translate some *value* into a *number*.

We need a function like that: given a Patron ID, it will return a number that we can use as an index in a list or an array.

Let's take a look at how such a function might work...

Hash Functions

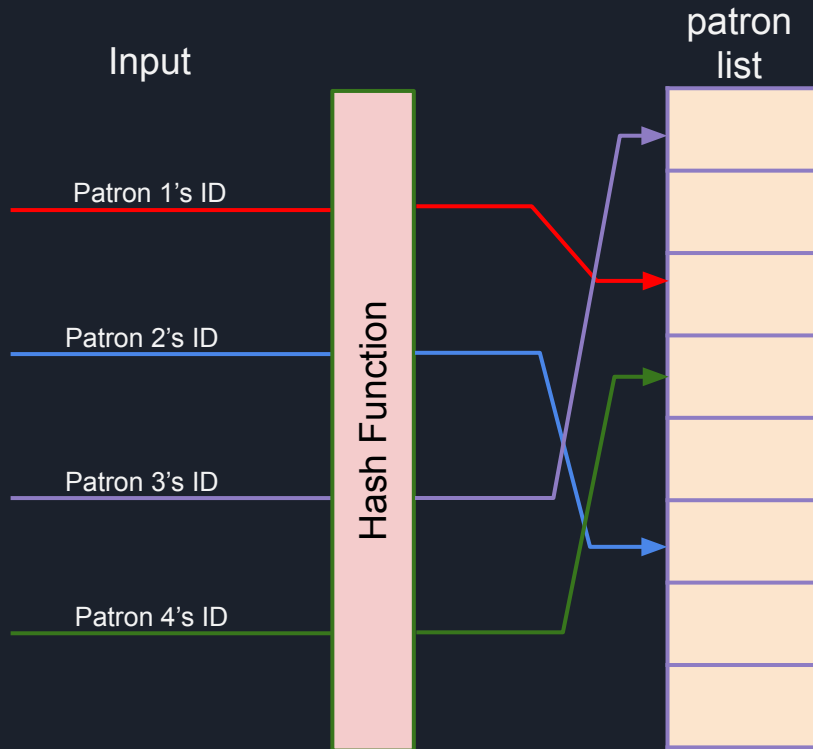
Assume that an empty list is created that is large enough to store **M** patrons.

If we simply store the patrons in the order that they are added, finding a specific patron later will require searching the list, an **$O(N)$** operation.

Instead, some kind of *hashing function* is needed: a constant time operation that, given a patron ID as *input*, produces a number that can be used as the index into which the patron should be stored.

Later, when the patron needs to be retrieved, the hashing function can be used with the ID *again* to find the correct index in the list.

This means that, with the right hashing function, adding and retrieving patrons from the list would be an **$O(1)$** (constant time) operation.



hashCode()

- In Java, objects are responsible for providing their own hashing functions.
- The good news is that the `java.lang.Object` class provides a default `hashCode()` function.
 - Every object has a **public int** `hashCode()` function that, when called, returns an integer called a *hash code*.
- The bad news is that the default implementation of the function is not particularly useful.
 - Does this sound familiar?
- We'd like to use a patron's ID to store and retrieve the patron.
 - This means that we will need to override the default `hashCode()` function to translate the patron's ID into an integer.



Every patron ID is a string in the format “ABC1234”.

If we treat the alphabetic characters like integers, it should be easy to translate any ID into a unique hash code.



Activity: Putting into a Hashmap

K/V

Let's implement the put method!

- Use the hashCode() function on the key to get a hash code.
- Create a new Entry.
- Use the hash code as an index to store the entry in your array.
- Don't forget to increment the size!



Activity: Getting out of a Hashmap

K/V

The hashCode for an integer is the number itself.

So we need to be careful that the integers aren't bigger than our array (100)!

Let's implement the get method!

- Use the hashCode() function on the key to get a hash code.
- Get the Entry from the correct index.
- Return the value!

Now let's test the Hashmap with some integers and strings.

- Create a new JUnit test.
- Write a test that creates an instance of your Hashmap that stores Integer keys and String values.
- Add a few key/value pairs. Make sure not to use numbers over 100! **Why?!**



The Hash Code Contract

- A hash function translates an arbitrary object into a hash code.
- So what would happen if:
 - An key/value pair is **put into** the hashmap.
 - The state of the key changes, and so the **hash code changes** as well.
 - The key is used to try to **get** its value from the map.
- The hash code is used to find an index in the array inside of the map.
- If the hash code changes, the index will almost certainly change as well.
- This means that it is very important that, once an object has been hashed, its hash code never changes.
 - Otherwise we won't be able to find the object in any hashing data structures in which it has been stored!

If the hash code is some value *X* when the key is used to put a key/value pair into the map...

...but it is some other value *Y* when you try to use the key to get the value, you won't find it!

For this reason, the hash code contract states that, **once a hash code is returned, it will never change.**

Activity: A Patron Hash Function



Right now our array is big enough to hold 100 patrons. How big does it really need to be?

Our hashmap works with integers, but we need it to work with patrons! Add a `hashCode()` function to your Patron class.

- Don't forget the `@Override` annotation.
- If you convert it to uppercase, a patron ID is always in the format "ABC1234".
 - Remember that ASCII characters *are* integers, but the values aren't quite right.
 - The first 3 characters are letters of the alphabet with ASCII values between 65 (A) and 90 (Z).
 - Hint: subtract 65
 - The last 4 characters are digits with ASCII values between 48 (0) and 57 (9).
 - Hint: subtract 48

The Array Needs to be How Big?!

- *Good news, everyone!* We now have a constant time hash function that can translate any patron ID into a number that we can use as an index in a list or an array!
 - This means that given an ID, we can fetch that patron from the array in constant time rather than doing a linear search!
- There is just one small very large problem.
 - In order to use the patron ID as an index, we need to allocate an array large enough to store any possible patron ID.
 - The number of possible IDs is $26^3 * 10^4 = 175,760,000$
 - The ID "ZZZ9999" maps to index 175,759,999
- Even if our library has 10,000 patrons, that's ~17,600 empty spaces in the array for every 1 patron.

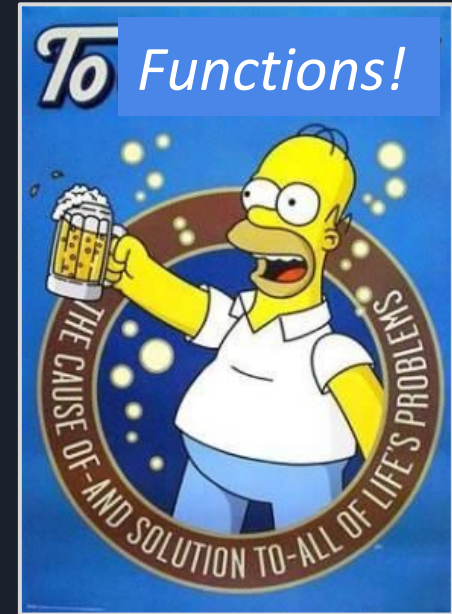


In **general** when time complexity and space complexity are in competition, we prefer to sacrifice space complexity for better time complexity.

But using 17,600 times the space needed to actually store our patrons is **absurd**.

There Has to be a Better Way

- I think we can all agree that using an array with 176 million available indices to store 10 thousand patrons is silliness.
- It's OK to allocate *some* additional space, but within reason.
 - And we'd definitely like to get that constant time performance when accessing the array.
- What if we could somehow translate the very large numbers that are returned by the hashCode function into an index that would fit into a much smaller array?
 - And what if the size of the array changes?



It sounds like we need yet another function!

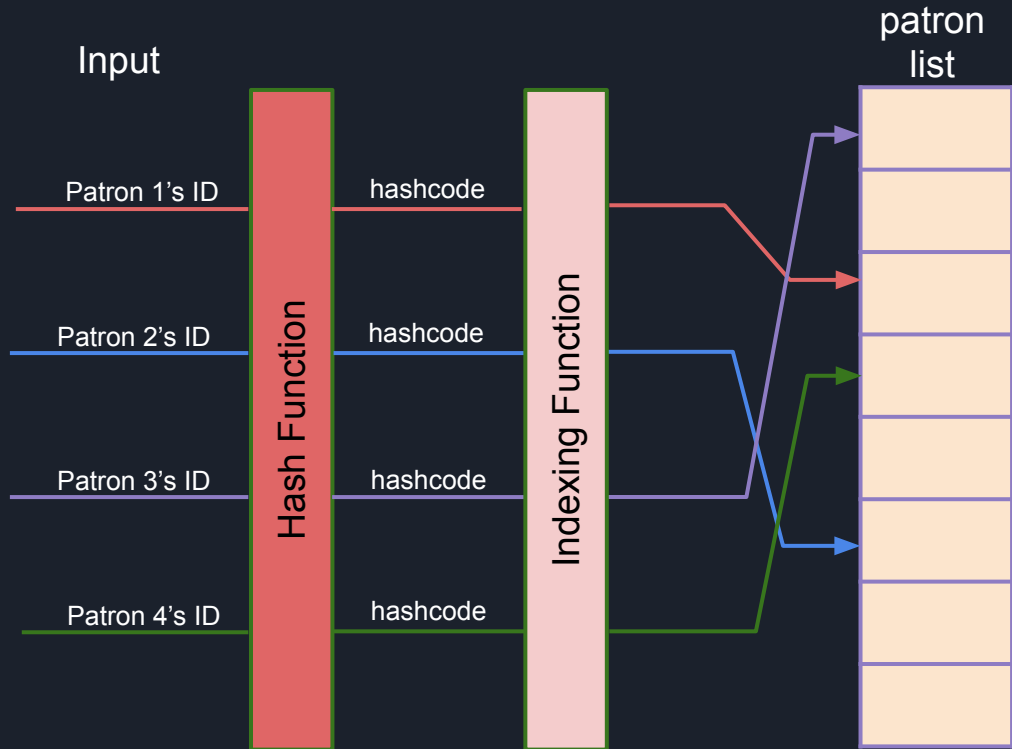
An Indexing Function

The indexing function is inserted *between* the output from the hash function and the list.

Each patron ID is first *hashed* to translate it into a numeric *hashcode*, which may be an arbitrarily large number.

The hashcode is then used as input to the indexing function which maps it onto an index that will actually fit within the array.

Q: What would an indexing function like this actually look like?





An Indexing Function

- So we'd like our hashmap's array to be small to start, e.g. big enough to hold ~100 patrons.
 - We'll worry about making it bigger if needed later.
- So how do we convert a hashCode with a value between 0 and 175,759,999 into an index that will fit into a table of size 100?
 - Is there a simple mathematical operation that we can use that, given a size S , will guarantee to return an integer that is between 0 and $S-1$?
 - Yes! Modulo!
- There are many possible indexing functions, but we will simply use modulo (%) to convert an arbitrarily large integer into an index using the size of the table.

The hashCode function in our Patron class will always produce a positive integer.

This is not necessarily true for every Java class; negative hash codes are not only possible, but likely in many cases (including Strings).

Our indexing function should use the `Math.abs()` function to make sure the index is positive.

Activity: An Indexing Function



Yes, this is backwards, but we're just practicing writing hashCode methods for now.

Let's add a new method to our HashMap class:

- `private int index(K key)` - this method, given a key, should get its hash code and apply our indexing function.
 - Use modulo (%) and `Math.abs(int)` to convert it into an index that will fit in our array.
- Update the put and get functions to use this new method to find an index for each new Entry.
- Update your JUnit test to try hashing some much larger integers to make sure that it works.
- Now write another test that creates a map of Patron keys and String (ID) values. Add a few!

Collisions!

- Our hashCode function has ~176 million possible outputs.
 - One for every possible patron ID.
- In Java, there are over 4 *billion* possible hash codes.
 - Any object may return a hash code between -2.14B and +2.14B.
- However, for an array of size 100, the indexing function only has 100 possible outputs.
 - One for every index in the table between 0 and 99.
- It is therefore not only *possible*, but *likely* that two different patron IDs, once hashed, will be indexed to the same location in the array.
- In fact, there are about 1.8 *million* possible patron IDs for each index in an array of size 100.
- When we try to store two *different* patrons in the same location in the array, this is called a **collision**.



The good news (everybody!) is that there are mechanisms for handling collisions when they occur.

Let's take a look at one such mechanism called **open addressing**...

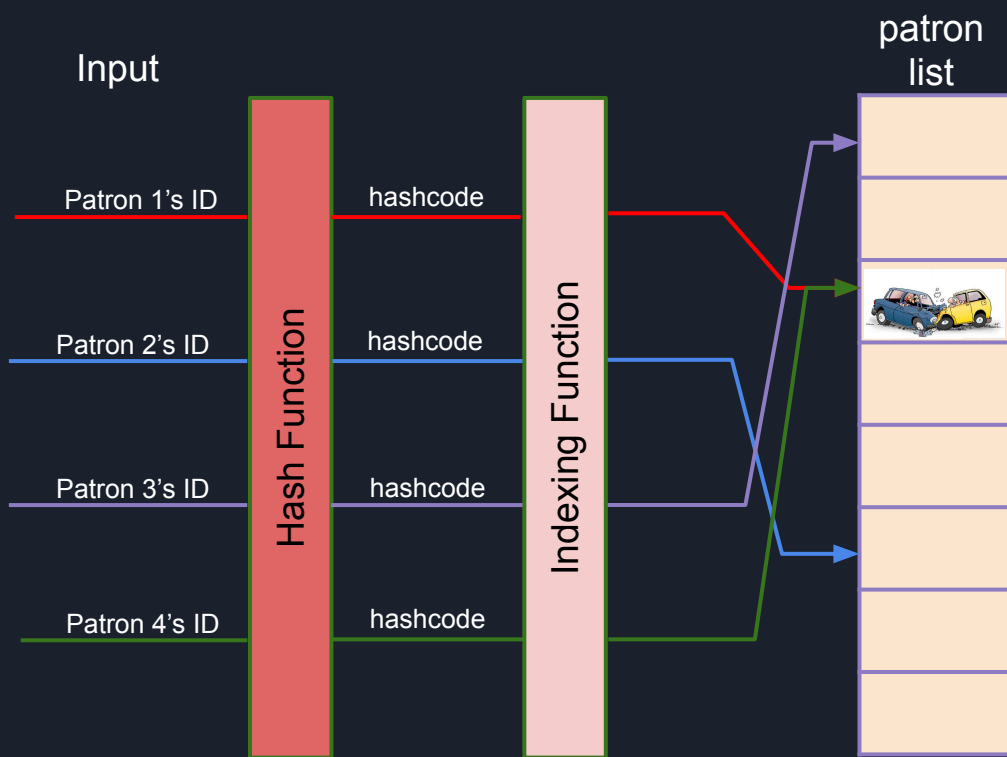
Collisions

In the best case scenario, our hashing function and indexing function work together to put each key/value pair in its own index in the array.

However, there are **significantly** more possible hashcodes than there are indexes in the array...

And so, as we store more and more keys, it becomes increasingly more likely that two keys will be mapped to the same index in the array.

We could blindly overwrite the old value, but this probably isn't the right thing to do...





Open Addressing

- It is not acceptable to simply overwrite (and lose) one patron that is stored in the array when another patron is indexed to the same location.
- But what is the alternative?
- Hopefully the array is **sparse**, which means that there are some empty spaces in the array.
- Why not start at the index and search for the next empty space in the array?
 - We can tell which spaces are empty, because they will have a value of **null**.
- This technique is called **open addressing**. In the event of a collision, it refers to finding the next **open address** in the array.

Open addressing adds two wrinkles to our hashmap.

The first is that we can't blindly insert a new value into the array.

Instead, we must search for the next empty address, wrapping around to the start of the array if necessary.

The second is that we can't blindly return the value at a specific index.

instead, we must make sure that it is the *right* value before returning it.

Open Addressing

In this diagram the **red** indices are occupied and the **green** indices are free.

If a new item is indexed to an occupied space...

A search is performed to find the next **open address** in the array...



And the new value is added at that index.

Open Addressing

If the starting index is near the end of the array, it is possible that there won't be any free addresses before the end.



If this happens, the search will wrap to the start of the array and continue from there.

If an open address is found, the new element is stored there.


Comparing Keys

- Before we worry about handling a collision, we need to determine whether or not a collision has occurred at all.
 - It is possible that the same key was used to add a new value to the map.
 - In this case, we simply want to replace the old value with the new one.
- So how do we tell if a collision has occurred, or if the same key is being used again?
 - That is to say: when a key is used to put a value in the map and that index is already occupied, can we check to see if the keys are the same?
 - How about the `equals(Object)` method?
- This would imply that we have implemented an `equals(Object)` method that, when two keys are compared, returns the correct result.
 - How do we know if two instances of the `Patron` class represent the same patron?



Right now, two different patrons with the same ID are not considered equal!

Let's fix that! But first...



equals(Object) & hashCode()

- You are already familiar with how the equals(Object) method on an object is supposed to work.
 - It performs a **deep comparison** of the state of two objects.
- In Java there is a general contract between hashCode() and equals(Object).
 - Two objects that are equal according to the equals(Object) method should **return the same hash code**.
 - As much as possible, two distinct (i.e. unequal) objects should **not** return the same hash code. This is impossible to guarantee, but an effort should be made.
- It is therefore strongly recommended that whenever the hashCode() method is overridden, the equals(Object) method should be as well.

Keeping equals(Object) and hashCode() consistent can be tricky.

The hash code should not change even if the state of the object changes.

But two objects that are equal should have the same hash code.

This implies that the equals(Object) method should not use state that is ever going to change.

Activity: An equals(Object) Method



The **final** keyword can be used to make sure that the value of a field never changes.

The `equals(Object)` and `hashCode()` methods should ideally use **final** fields.

Before we can tell if a collision has occurred, we need to be able to compare keys to make sure that the same key isn't being used again.

- Add an `equals(Object)` method to the Patron class.
- Don't forget the `@Override` annotation.
- How do we know if two patrons are the same person?
- How do we insure that the `equals(Object)` and `hashCode()` contract is fulfilled?

Activity: Open Addressing I

Now that we can compare keys to make sure that we found the right one, let's add open addressing to the `put(K,V)` method.

- Use the key's hash code and our indexing function to find the index for the key.
- Set `searchIndex` to `index`.
- **while** the `searchIndex` is not **null** and the key at the `searchIndex` is not equal:
 - Add 1 to `searchIndex` and mod by the length of the array. The mod will wrap the index around to the beginning if it gets to the end.
 - If `searchIndex == index`, the array is full!
- Assuming we found an address, store the Entry there.



What should we do if there are no available addresses in the array?

Activity: Open Addressing II



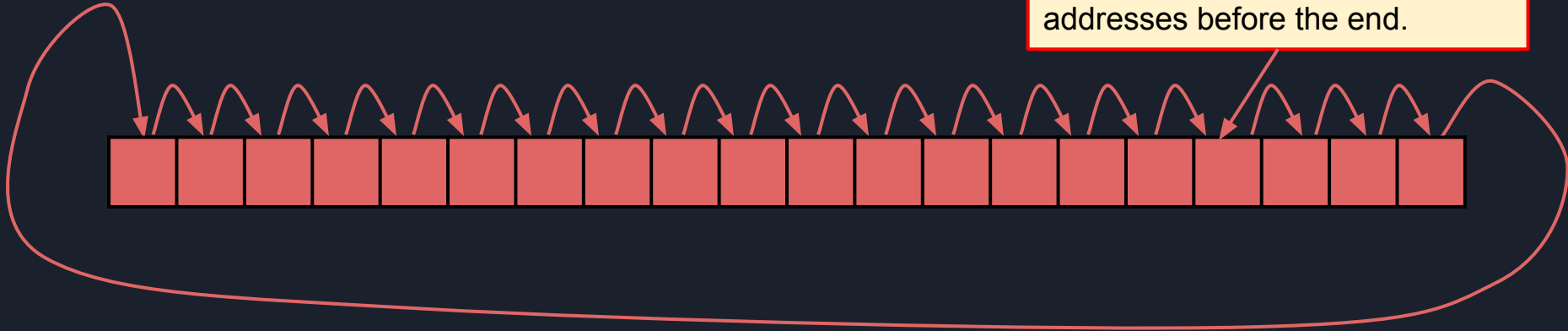
Now let's talk about what to do if the array is completely full...

Now that there is the possibility that a key is not in the exact index that our indexing function suggests, we need to update the `get(K)` method to search for the correct key.

- Use the key's hash code and our indexing function to find the index for the key.
- Set `searchIndex` to `index`.
- **while** the `searchIndex` is not **null** and the key at the `searchIndex` is not equal:
 - Add 1 to `searchIndex` and mod by the length of the array. The mod will wrap the index around to the beginning if it gets to the end.
 - If `searchIndex == index`, the array is full!
 - Add 1 to search index
- If the key at the `searchIndex` is the right key, return the value! Otherwise, return **null**.

Open Addressing

If the starting index is near the end of the array, it is possible that there won't be any free addresses before the end.



If this happens, the search will wrap to the start of the array and continue from there.

If the search wraps around and reaches the starting index, this means that the array is full!

At this point it's appropriate to raise an error (or is it...?).

Complexity!

- The *expected* complexity of the put and get methods on our hashmap are constant time: $O(C)$.
 - Except when it's not.
- But each time a collision occurs, we need to search the array for the right location for the key/value pair.
- As the array becomes more and more full, the average search gets longer and longer.
- The worst case scenario is that the array is completely full. This means that we need to do a linear search over the *entire array* when trying to add a new key.
 - $O(N)$!
- Trying to get a key/value pair is also $O(N)$ when the array is full.
- Even worse, if our array is full we can't add new key/value pairs!



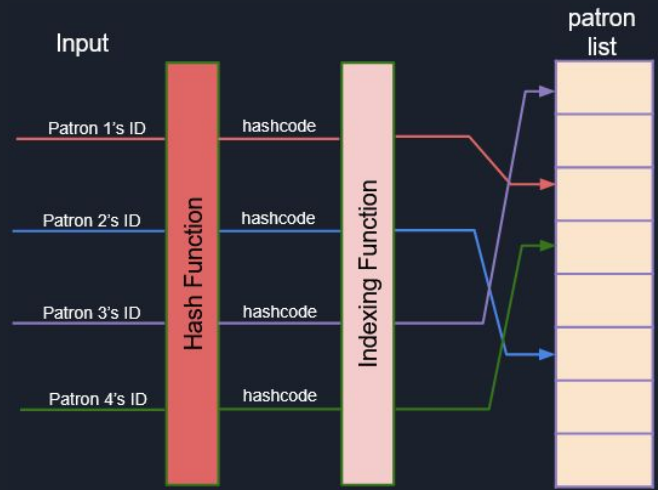
The more often that collisions occur, the worse that our super array performs.

That is to say that operations we hope will be $O(1)$ are $O(N)$.

The only way to preserve the performance is to reduce the probability of collisions. How?

Rehashing

- As the array gets more and more full the put and get operations get closer and closer to linear time.
 - This is no better than searching a list or an array!
- To insure that this doesn't happen, we need to keep track of the *sparsity* of the array.
 - This is a measure of how empty the array is.
- If this falls below a certain threshold, we need to resize the array.
 - Just like an ArrayList!
- However, we can't simply copy the old key/value pairs into the new array.
 - Why?



The index of each key in the hashmap is a product of the length of the array.

Therefore, if the length of the array changes, the index will, too.

When resizing the array, we must **rehash** all of the key/value pairs into the correct, new indexes in the larger array.

Activity: Rehashing



We now have a completely functional hashmap!

Modify the `put(K,V)` method to implement rehashing!

- The sparsity of the array can be measured using the ratio of its size (the number of entries) and its capacity (the length of the array):
 - i.e. `size / array.length`
- If the table is over 75% full, it should be rehashed.
 - Save the old array in a temporary variable.
 - Create a new, empty array with twice the capacity, and set the hashmap to use it from now on.
 - Iterate over the old array, and for each non-**null** entry, call the `put(K,V)` method to hash it into the new array.