Java Programming

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

Today's Lecture

- Hash Table Overview
- hashCode

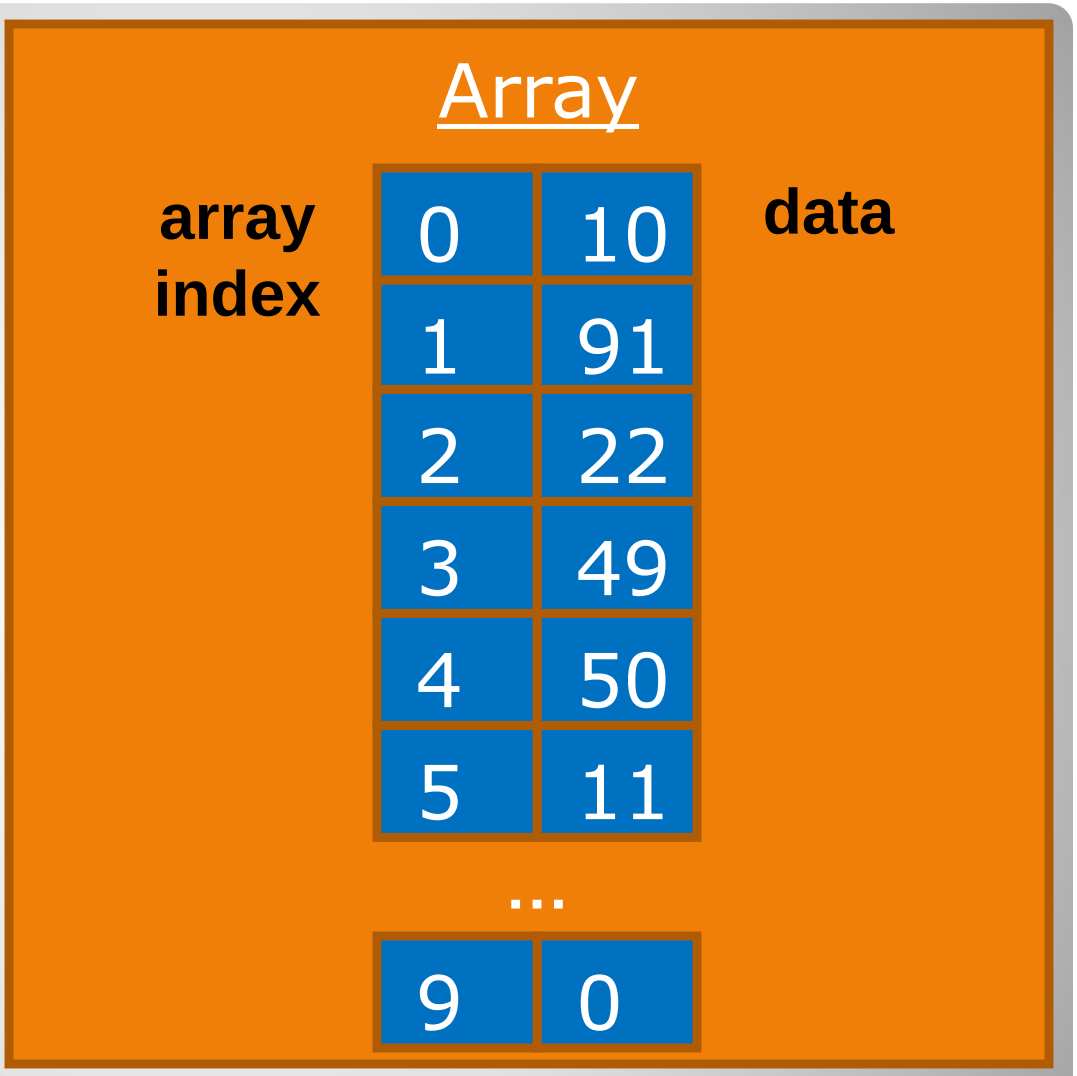
- Now we will cover hash tables in general (prerequisite for describing Java hashCode method)…

Hash Table

Array (Find)

- How would you go about checking the array for the value 49?
- (Advanced) What is the big O runtime of this operation?

Note: If you have not taken a data structures course do not worry about the big O runtime.



Array (Find)

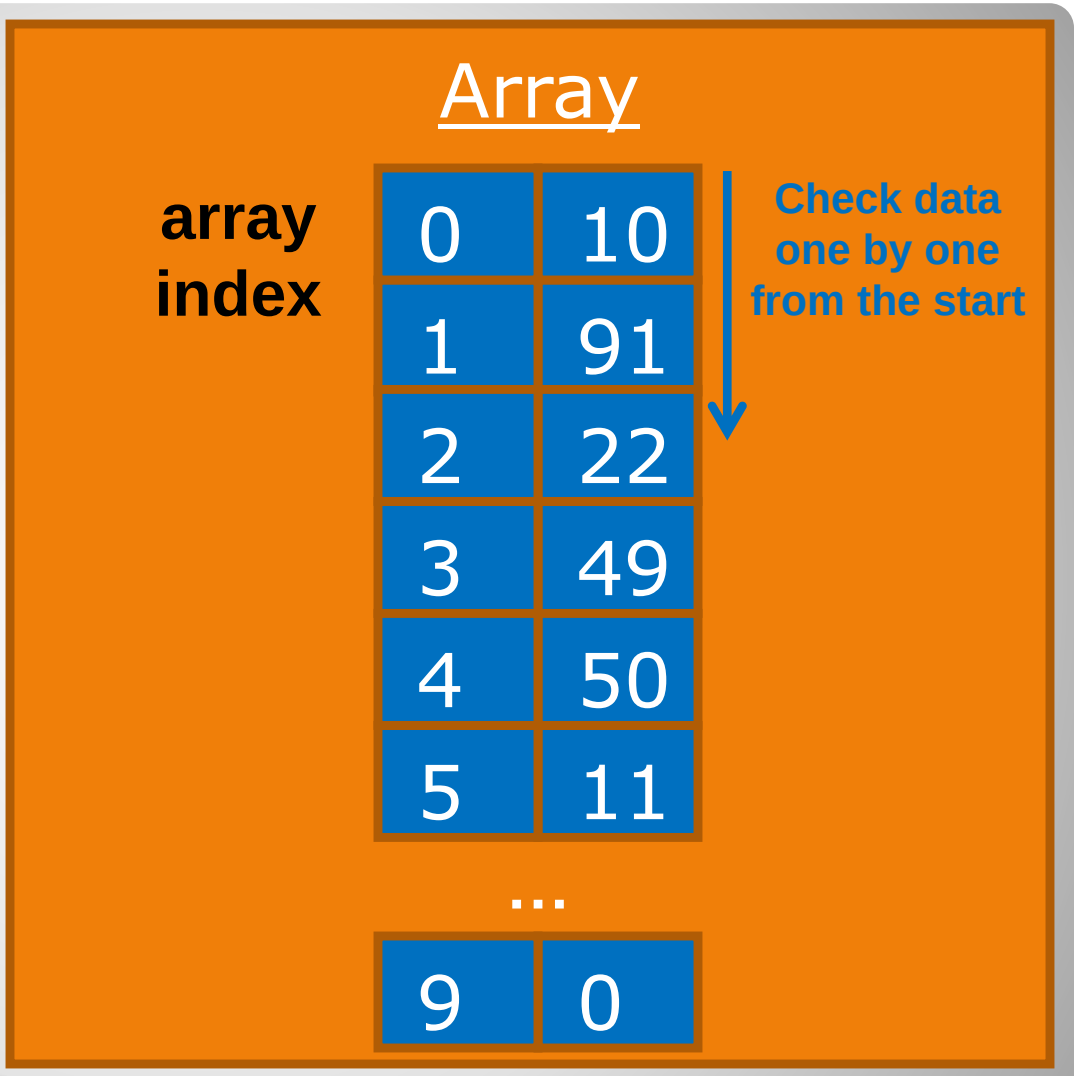
- How would you go about checking the array for the value 49?

Answer: Start from the beginning of the array and check each element to see if it has 49 in it.

- (Advanced) What is the big O runtime of this operation?

Answer: O(n)

We must search the whole array in the worst case

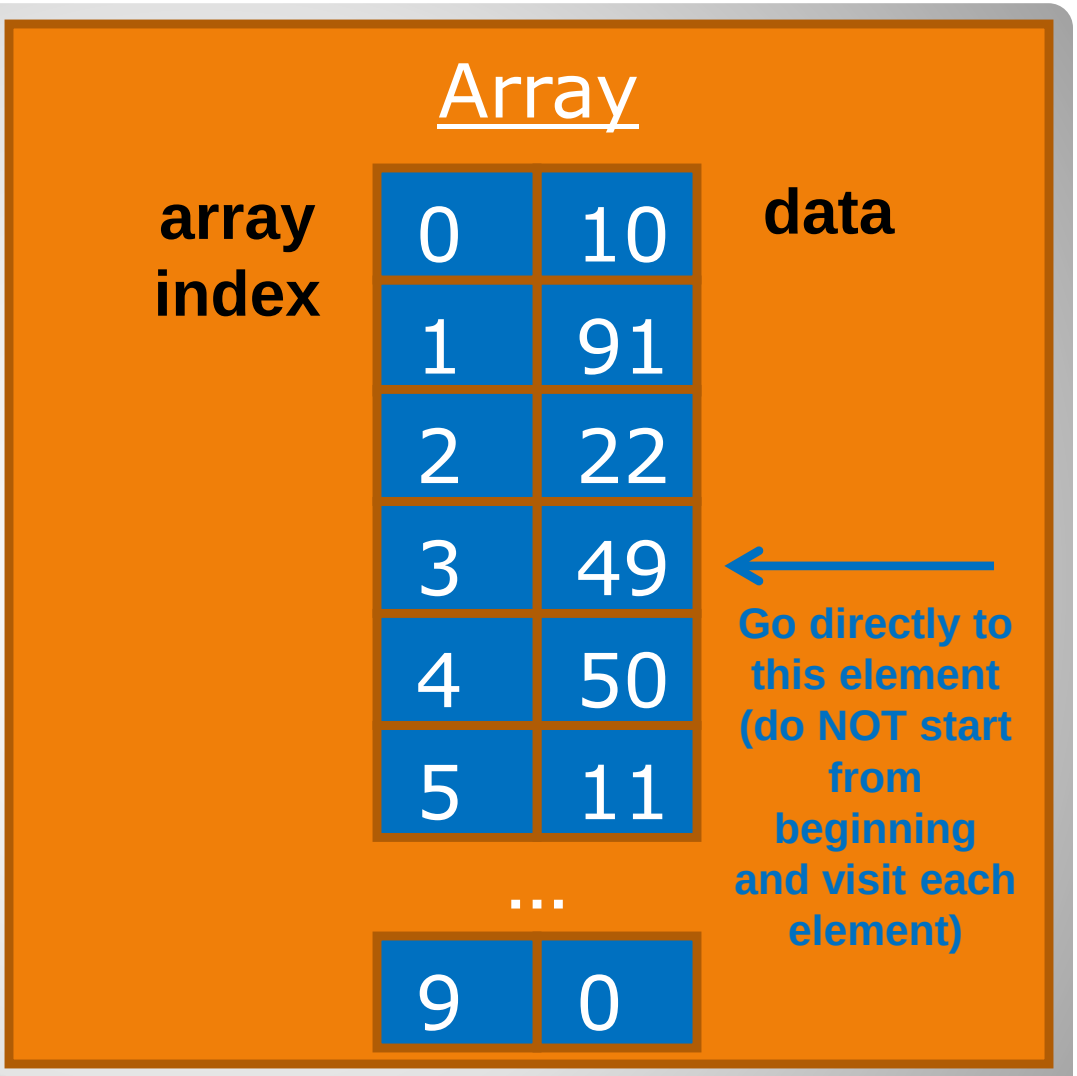


- Now assume we know the index of where the element we are searching for is located.
- If we did know the index, then we could just go directly to that element.
- For example…

What if we knew the target element's index?

Array (Find)

- Assume that we know 49 is at index 3.
- We can access it directly
- We do not need to "visit" indexes 0, 1, or 2 to get there (arrays have random access).
- (Advanced) The runtime of this type of access is O(1).



Hash Table

- Hash tables provide a mechanism to allow direct access to a piece data (no searching from the beginning of an array).
- A hash table uses a function to convert the data we are searching for into an index. This function is called a **hash function**. This is the "magic" we use to go directly to an index.

- Unfortunately, multiple pieces of data could end up having the same index be returned by the hash function. This is called a **collision**.
- To remedy the collision problem each index is treated as a bucket that can contain multiple values.
- Once a bucket is found, the bucket must then be searched for the target value (this is not a big deal though).

- For example…

Hash Table

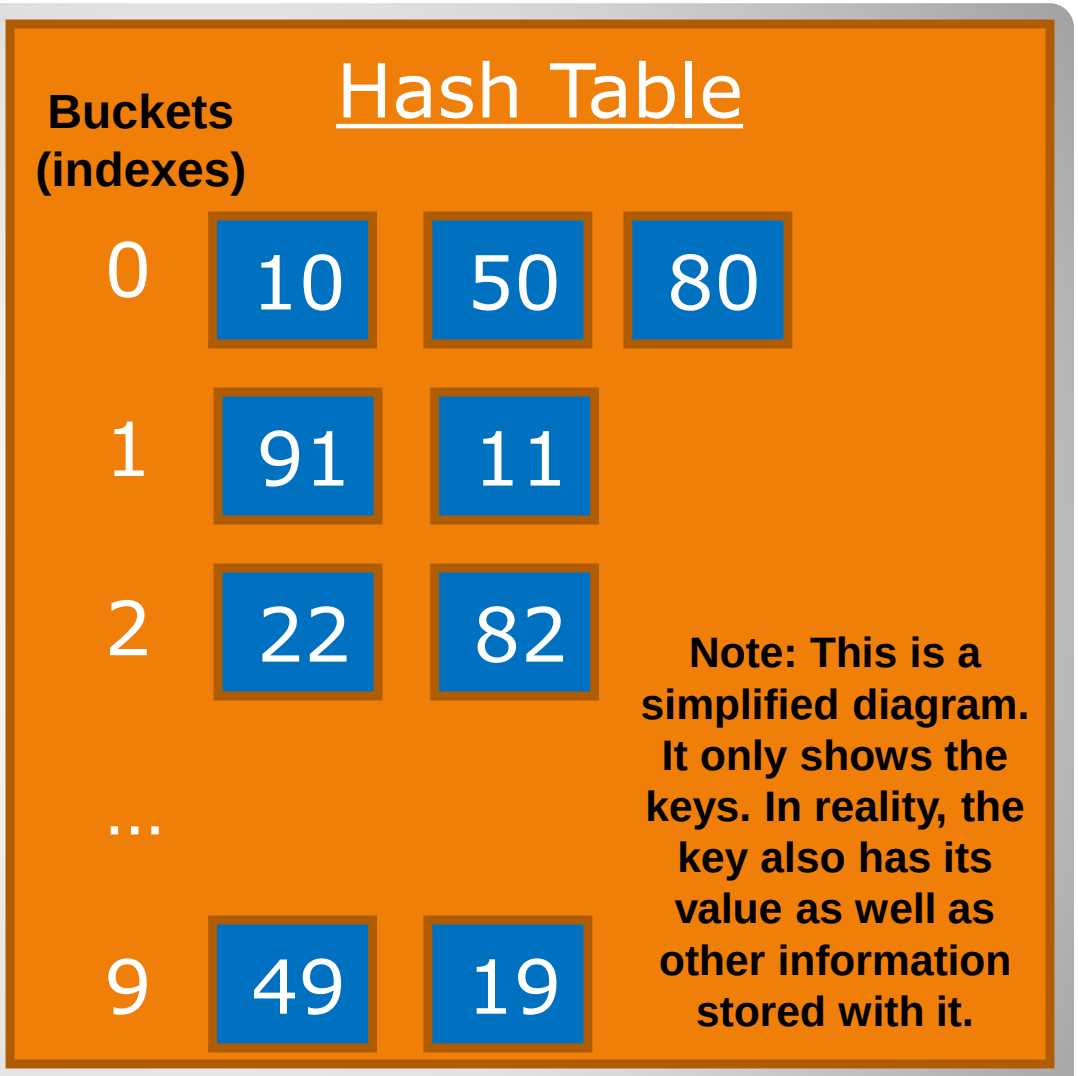
Hash Table

- A hash function is used to determine which bucket to put data into (the data is a key).
- Keys added to this table are:

10, 91, 22, 49, 50, 11, 82, 19, 80

- The hash function in this example is the mod function.

Bucket = Key % NumBuckets



Hash Table – Add

- Hash function:

Bucket = Key % NumBuckets

- Now add the key 52 to the hash table:

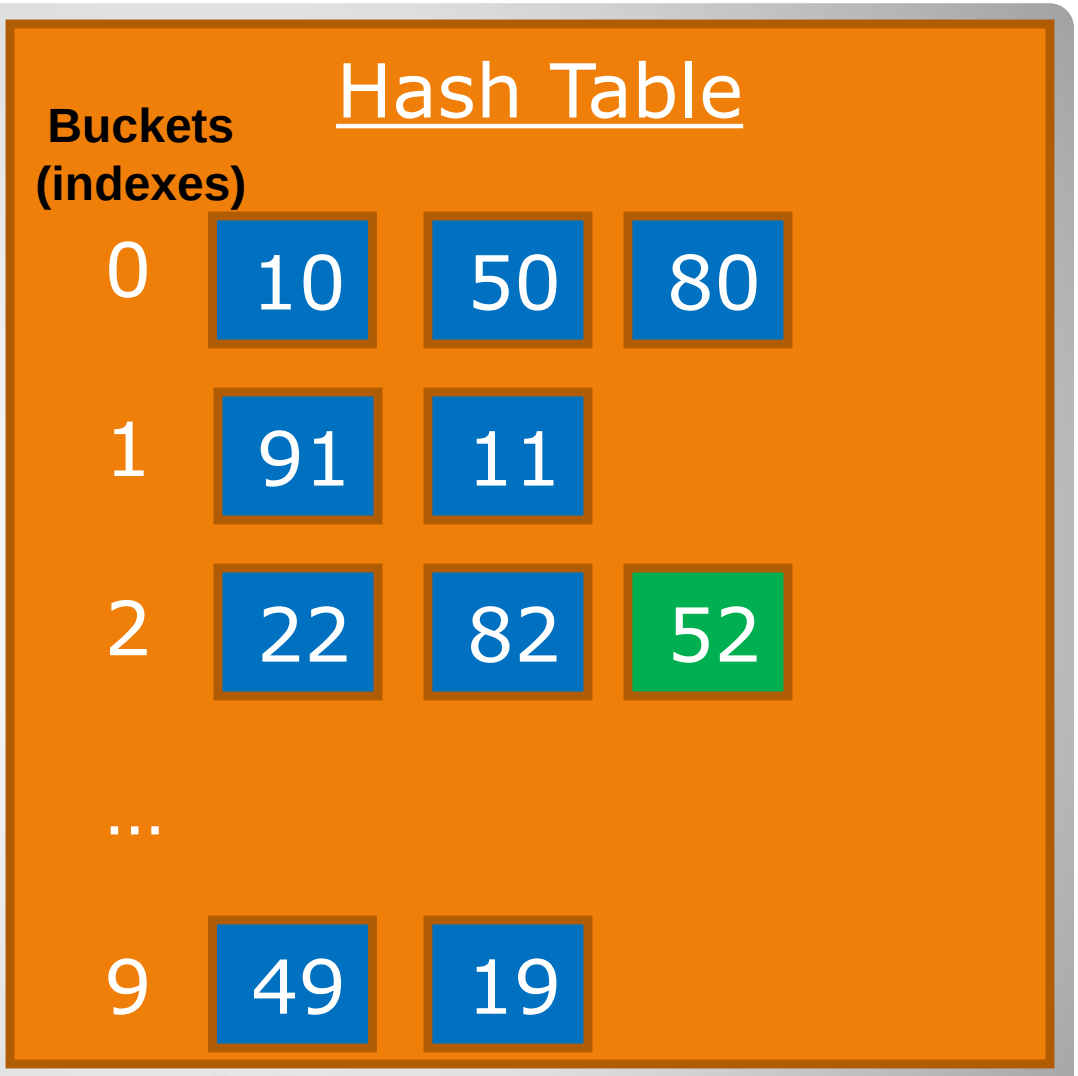
Key = 52

NumBuckets = 10

Bucket = Value % NumBuckets

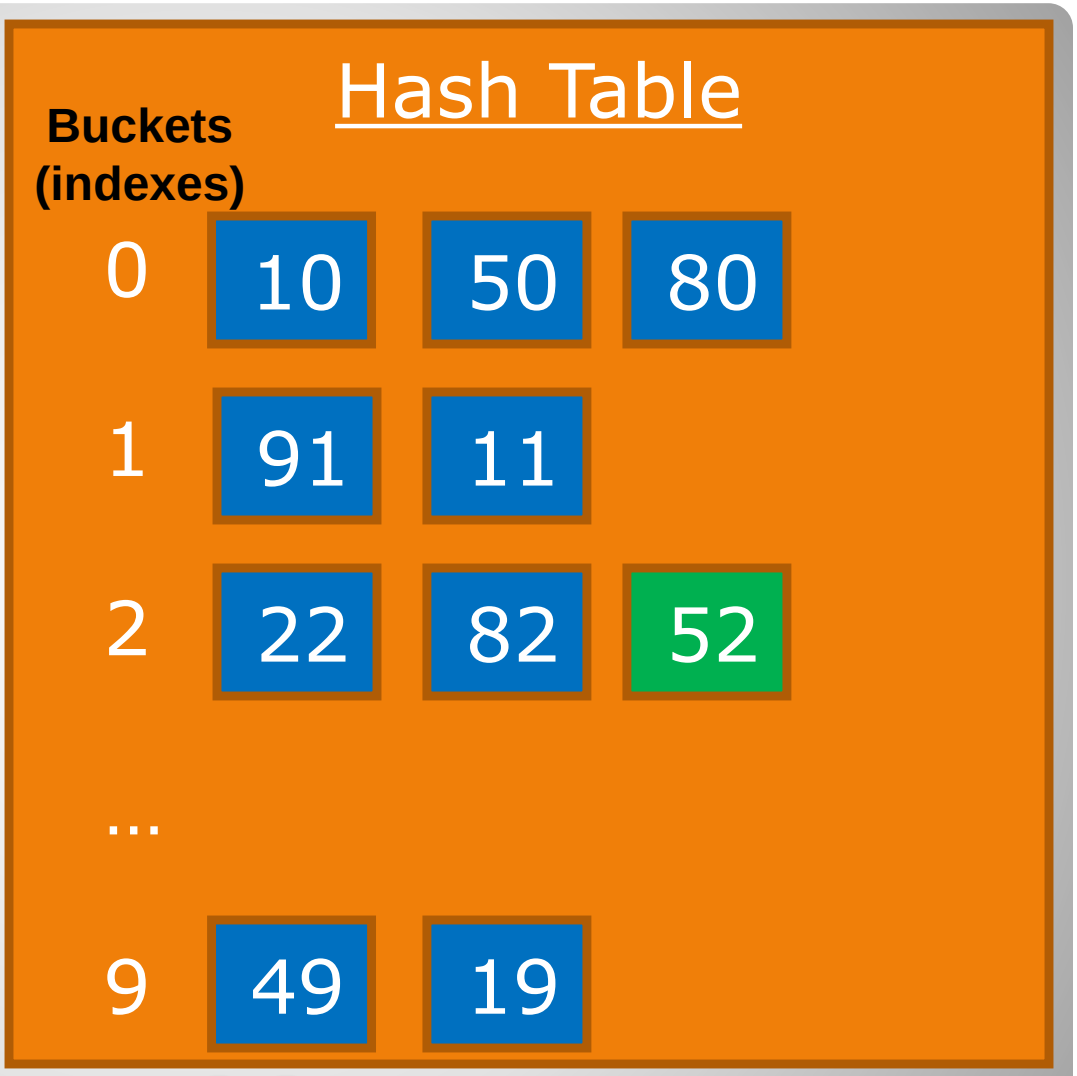
2 = 52 % 10

- Key 52 → Bucket 2



Hash Table – Add

- Add 71???



Hash Table – Add

- Hash function:

Bucket = Key % NumBuckets

- Now add the key 71 to the hash table:

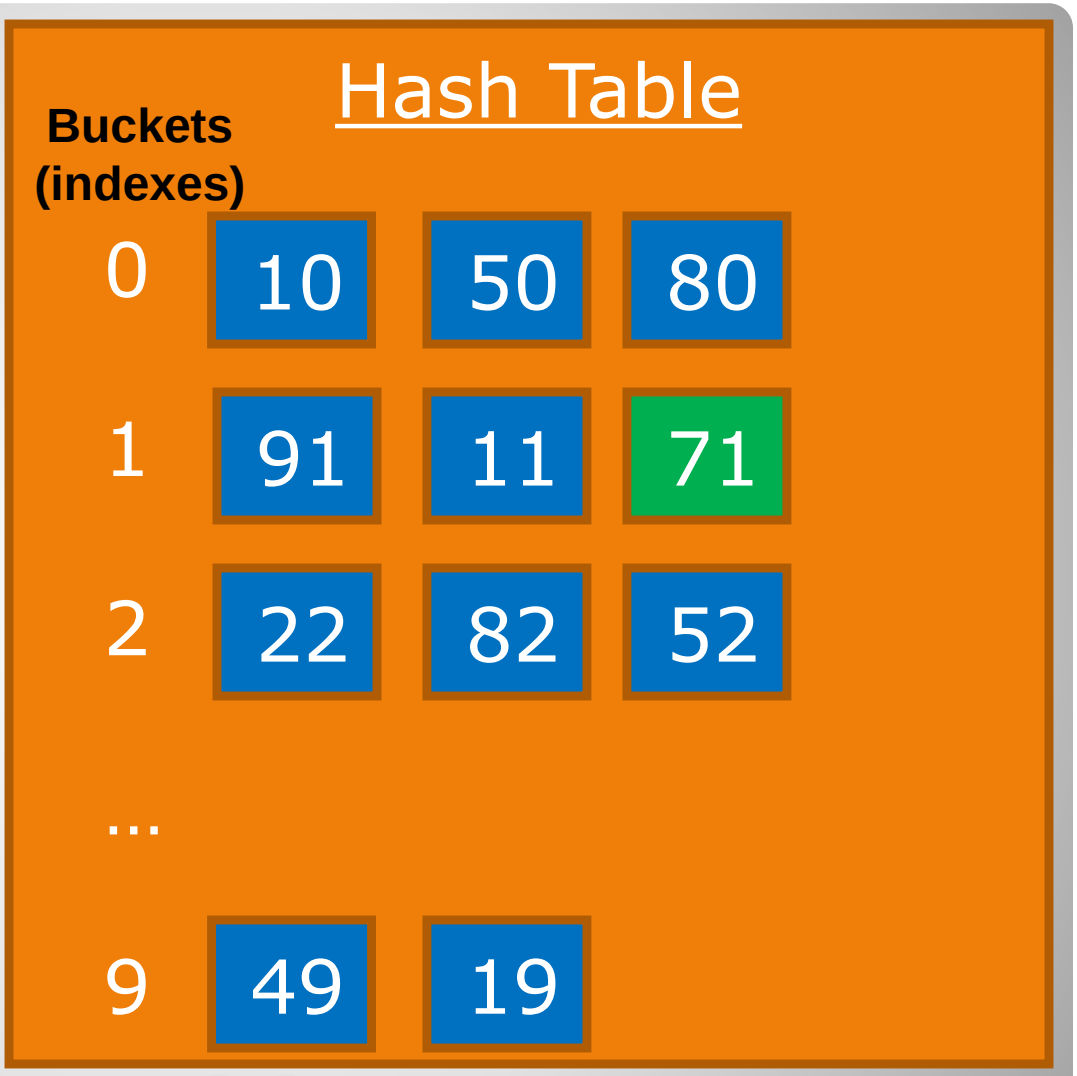
Key = 71

NumBuckets = 10

Bucket = Value % NumBuckets

1 = 71 % 10

- Key 71 → Bucket 1



Hash Table – Search

- Hash function:

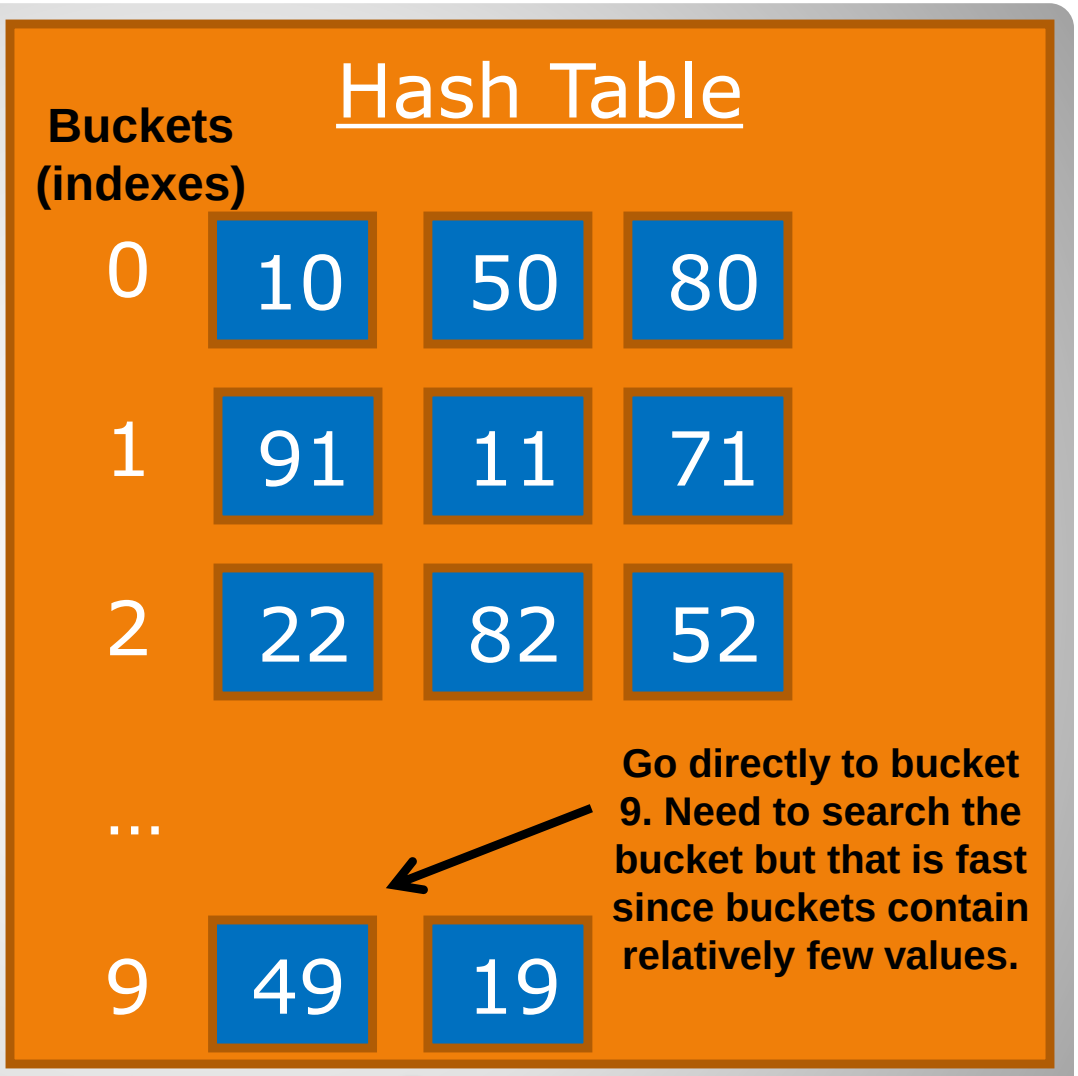
Bucket = Key % NumBuckets

- Now do a search for 49.

Key = 49
NumBuckets = 10

Bucket = Value % NumBuckets
9 = 49 % 10

- Key 49 → Bucket 9



Hash Table Time Complexity

- Hash tables are primarily used for their fast search speed.
- The number of buckets in a hash table needs to increase at certain times to keep individual bucket lengths small (resizing the bucket array needs to occur).
- The analysis below assumes the following:
  - The hash function uniformly distributes elements across the buckets.
  - Double the number of buckets when the load factor reaches .75.
  - Note: The load factor is the #items/#buckets.
- **Find - O(1).** Resizing by load factor will keep individual bucket lengths very small, so the search time is constant.
- **Add – O(1)+.** The + means amortized cost. Costs O(1) when not resizing and O(n) if it resizes. The amortized cost turns out to be constant because over the course of many adds the extra cost of occasionally resizing can be offset by assuming a small amount of extra constant time for each of the non-resizing add operations.

Hash Table Time Complexity

- Now we will cover the Java hashCode method…

Java hashCode

Java hashCode

- Java provides a hashCode method that serves as a hash function.
- The hashCode method is used by Java's hash-based collections.
- **IMPORTANT! - If you provide an override for equals in a class you should also provide an override for hashCode.**

Object.hashCode() Method

- The Object.hashCode method returns a hash value for an object.

int hashCode() – Returns a hash code for the current instance.

- **Object's default implementation of hashCode generally returns the address of the object instance.**

Object hashCode Implementation

hashCode Properties

- If two object's return the same value from equals, they MUST also return the same value from hashCode.

Same equals value → Same hashCode value

- In contrast, if two objects return the same value from hashCode they do not necessarily return the same value from equals. This is possible because objects with different values can have the same hashCode value (they would go in the same bucket).

~~Same hashCode value → Same equals value~~

hashCode Properties

```
public class Employee {
    public String firstName;
    public String lastName;

    @Override
    public int hashCode() {
        return firstName.hashCode() * lastName.hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        Employee other = (Employee) obj; // Copy to Employee var

        if (firstName.equals(other.firstName) == false)
                return false;

        if (lastName.equals(other.lastName) == false)
                return false;

        return true;
    }
}
```

Hash Code

Includes all members (firstName and lastName) in the hash code calculation. This will allow unequal objects to have different values for their hash codes.

Many IDEs will have an option that allows you to automatically generate an override of the hashCode method

hashCode Implementation

hashCode Using a Constant

- What effect does the following hashCode implementation have on a hash table?

```
@Override
public int hashCode() {
   return 1;
}
```

hashCode Using a Constant

hashCode Using a Constant

- We could write a hashCode method that just returns a constant. For example:

```
@Override
public int hashCode() {
   return 1;
}
```

Always returns 1. This is bad!!! Will NOT evenly distribute elements in buckets.

- This would cause all elements to be placed in the same bucket in the hash table (hash table deteriorates into a list).
- Since it is effectively a list the search time is now O(n).
- This defeats the purpose of using the hash table in the first place (hash tables are used for their fast searches).
- **Moral of the Story - The hashCode function must evenly distribute elements in buckets to ensure a fast search.**

hashCode Using a Constant

Java Collections and hashCode

- The hashCode method is used to generate a hash.
- Java HashMap and HashSet classes generate hash values of items they are storing using the hashCode method.
- It can then mod those hash values as necessary to put them in the appropriate bucket.

Java Collections and hashCode

Hash Table vs Hash Sets

- **Hash tables are designed around calculating an index so we can go directly to the data.**

Hash Table (general description)

- Maps keys to values (associative array). Will store key/value pairs.

Hash Set (general description)

- An unordered collection of values (not key/value pairs).

- Both are good for searching for values
- Both use "buckets" to store data (fixes collisions)
- How it works:

Given a key the hash function transforms the key into a bucket number then it stores/searches in ONLY that bucket.

- End of Slides

End of Slides