CSCI 232:
Data Structures and Algorithms

Hashing (Part 2)

Reese Pearsall

Spring 2025

Announcements

Lab 4 due **tomorrow** at 11:59 PM

Program 1 due **one week from today (2/20)** at 11:59 PM

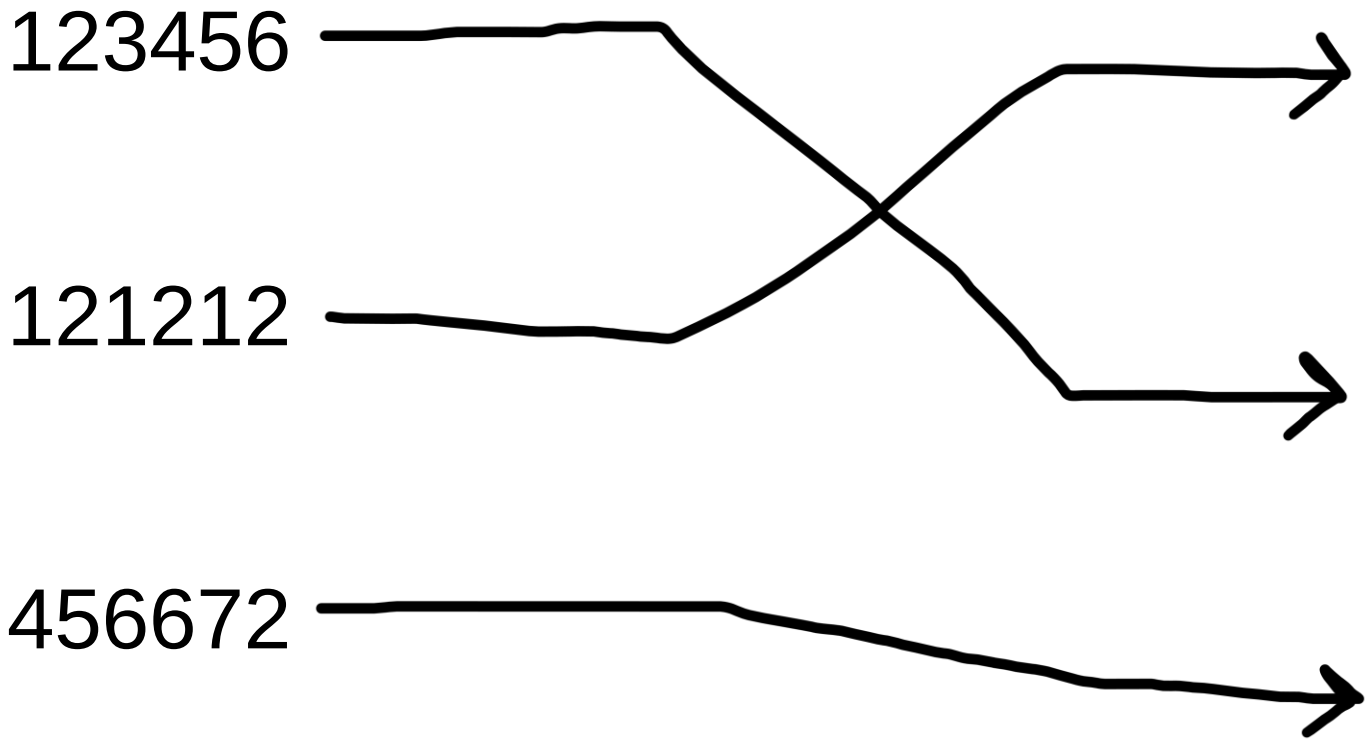
Not feeling well. Next Tuesday's lecture may be a lecture recording

Hash Tables

Student ID

Hash Function

`ID % 100`



Student[] Array

0	null
1	null
…	null
…	null
12	Sam, Political Science, 2.5 — Student Object
…	null
…	null
…	null
56	Sally, Mathematics, 3.0 — Student Object
…	null
…	null
…	null
72	John, Computer Science, 4.0 — Student Object
…	null
99	null

Hash Tables

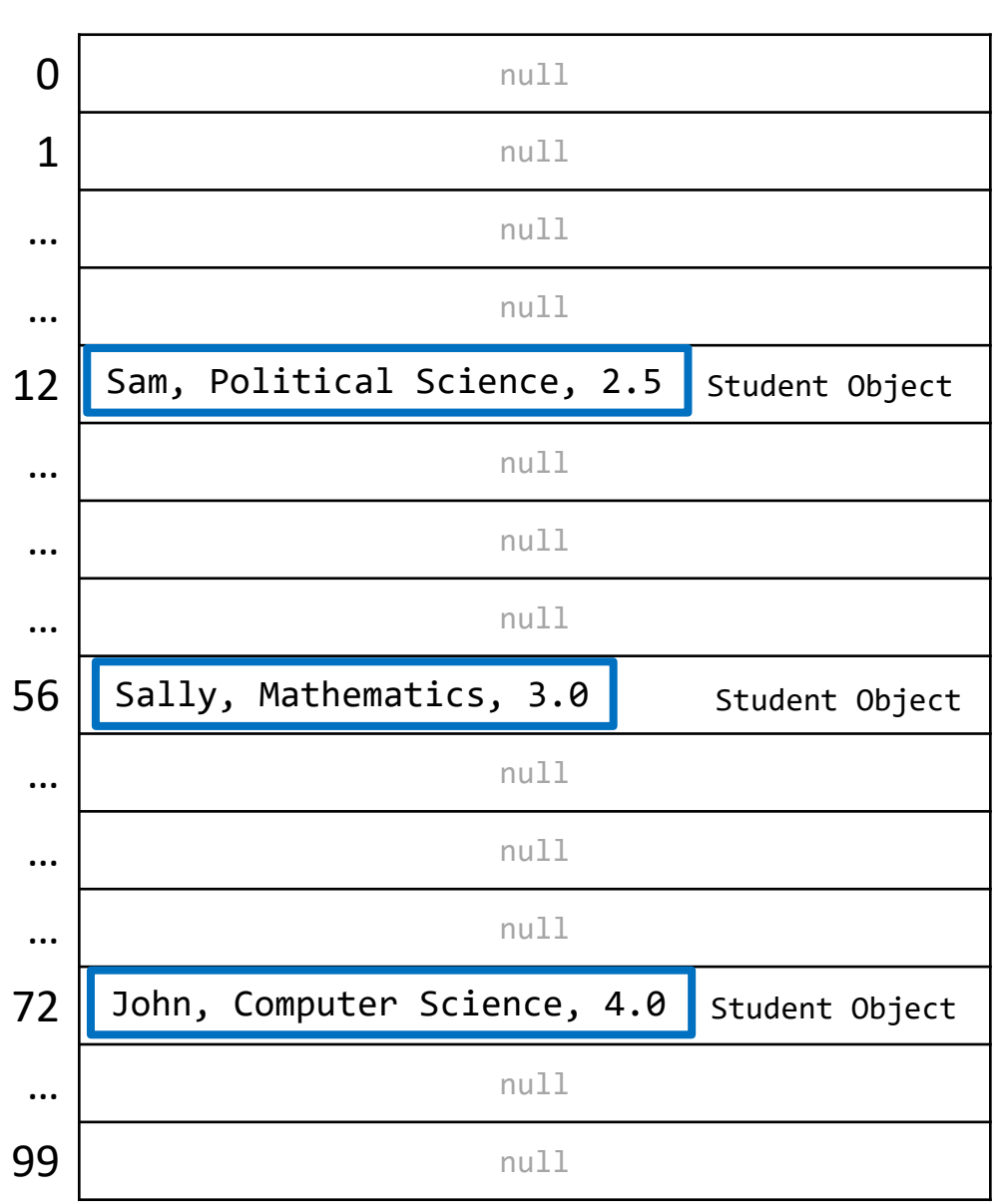
Student ID

Hash Function

```
ID % 100
```

Lookup time?

O(1) if you have the key



Hash Tables

Student ID

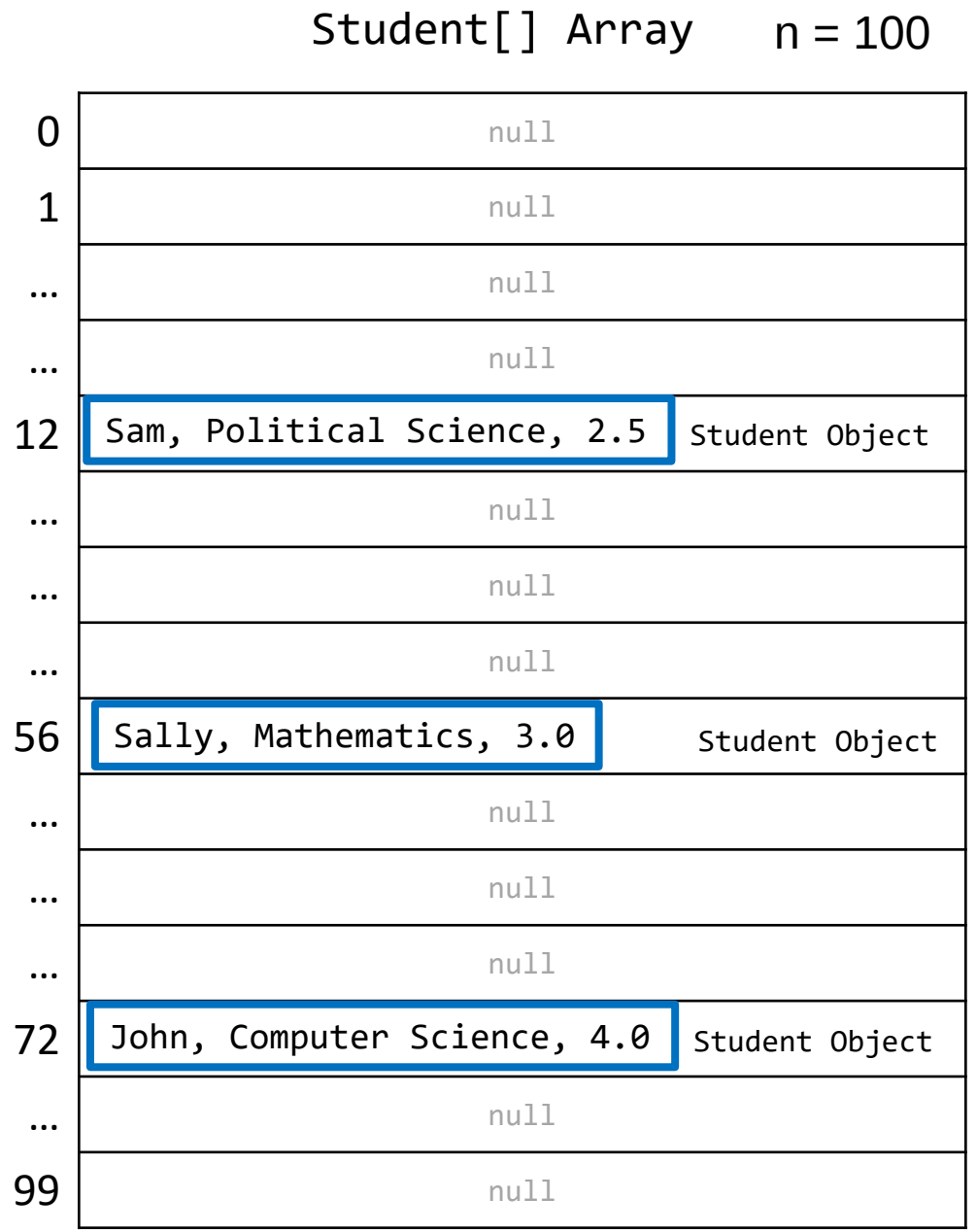
Hash Function

```
ID % 100
```

Lookup time?

O(1) if you have the key

O(n) if you don't have the key



Hash Tables

Student ID

Hash Function

```
ID % 100
```

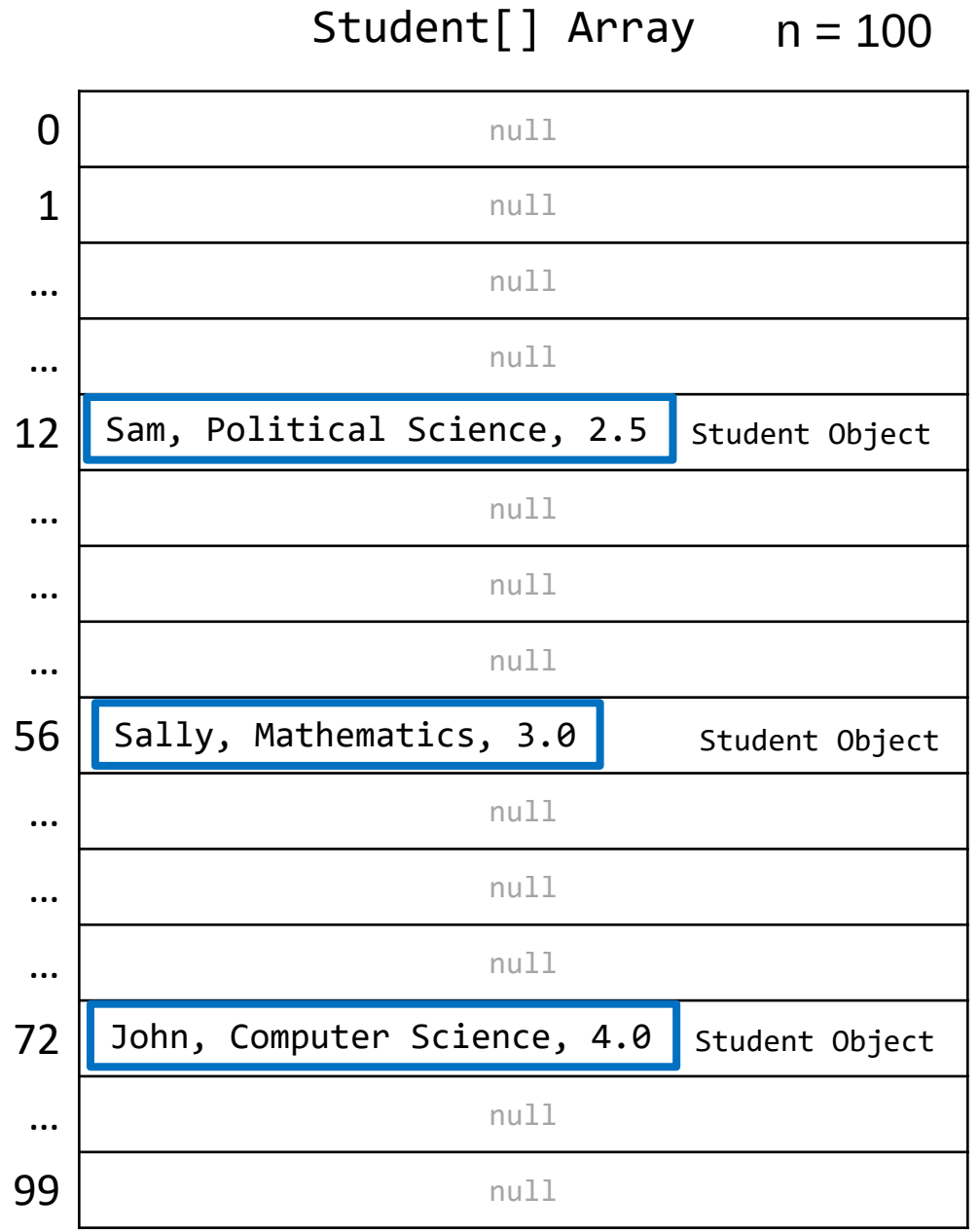
Lookup time?

O(1) if you have the key

~~**O(n)** if you don't have the key~~

O(k) if you don't have the key

k = | keyspace |



Hash Tables

Student ID

Hash Function

```
ID % 100
```

Lookup time?

O(1) if you have the key*

n = # of elements in data structure

Array – **O(logn)****

BST – **O(logn)*****

Linked List – **O(n)**

* If we can avoid collisions

**if the array is sorted

***if the tree is balanced

Student[] Array n = 100

0	null
1	null
…	null
…	null
12	Sam, Political Science, 2.5 Student Object
…	null
…	null
…	null
56	Sally, Mathematics, 3.0 Student Object
…	null
…	null
…	null
72	John, Computer Science, 4.0 Student Object
…	null
99	null

Hash Tables

Student ID

Hash Function

```
ID % 100
```

Insertion time?

O(1) *

n = # of elements in data structure

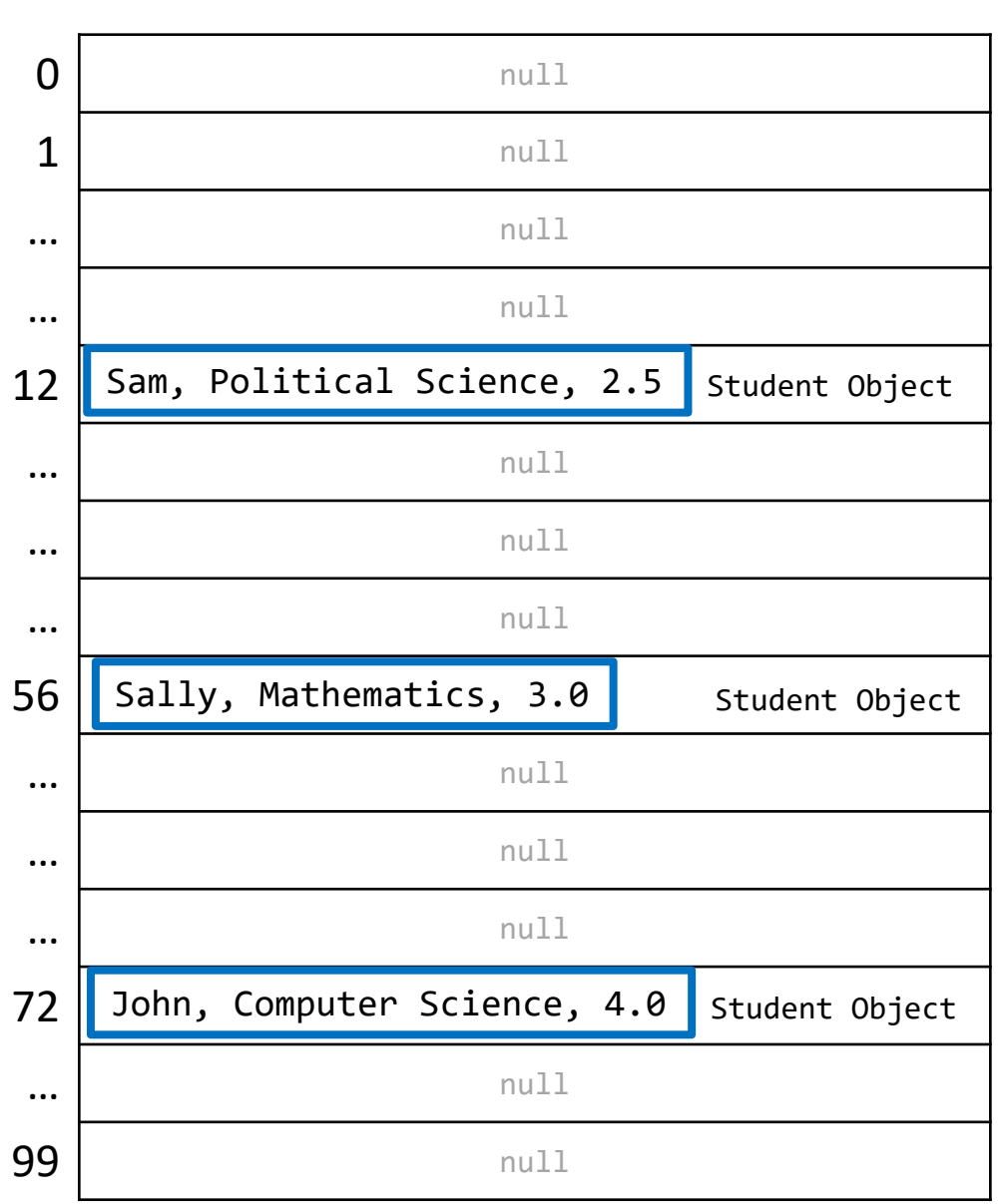
Array – **O(n)**

BST – **O(logn)****

Linked List – **O(1)**

* If we can avoid collisions

**if the tree is balanced



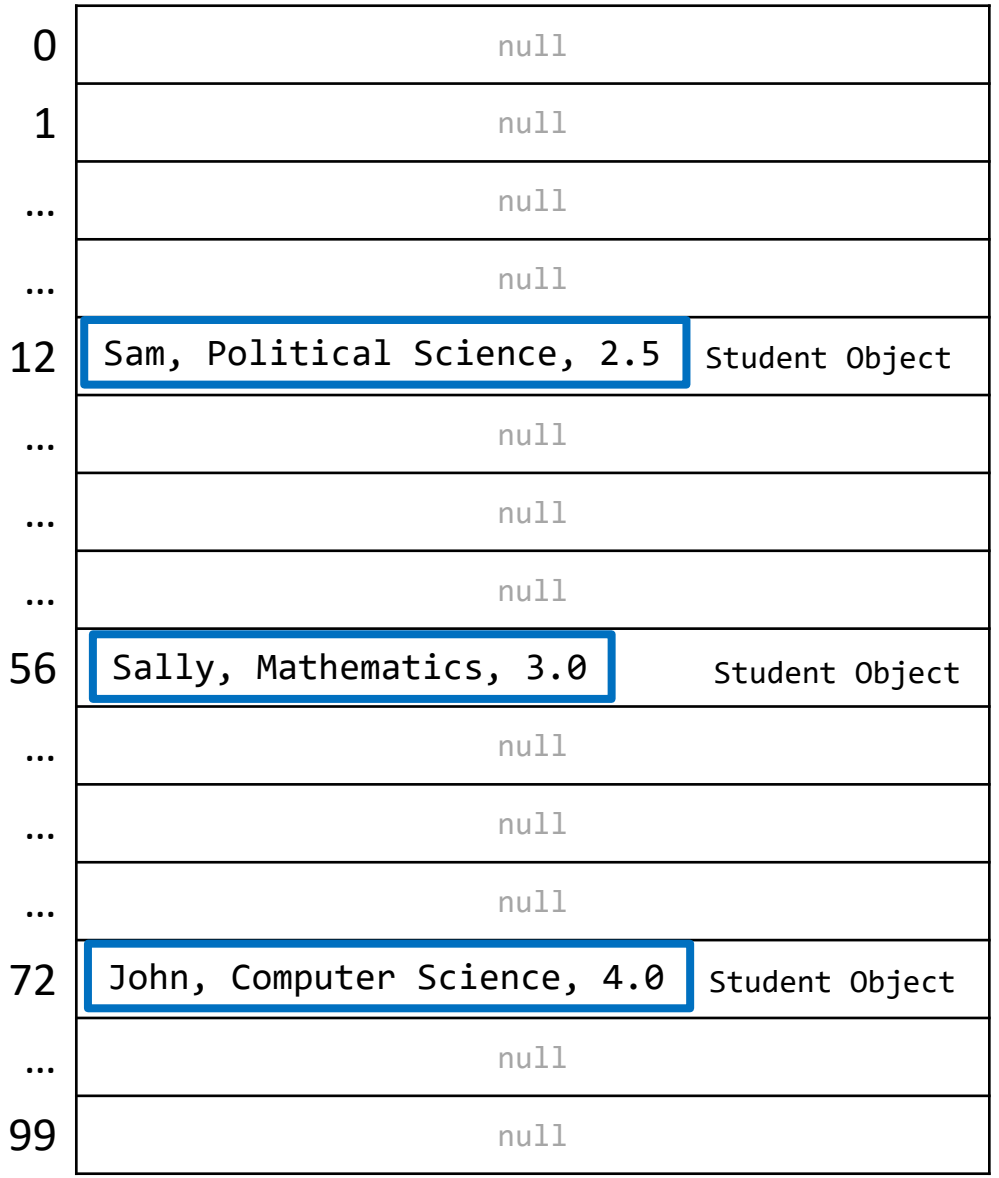
Hash Tables

Student ID

Hash Function

```
ID % 100
```

Student[] Array n = 100



Removal time?

O(1) *

n = # of elements in data structure

Array – **O(n)**

BST – **O(logn)****

Linked List – **O(1) / O(n)**

* If we can avoid collisions

**if the tree is balanced

Hash Tables

Student ID

Hash Function

`ID % 100`

Insertion

```
Student newStudent = new Student(name, major, id);

int arrayIndex = hash(id);
database[arrayIndex] = newStudent;

keySpace.add(id);
```

Student[] Array n = 100

0	null
1	null
…	null
…	null
12	Sam, Political Science, 2.5 — Student Object
…	null
…	null
…	null
56	Sally, Mathematics, 3.0 — Student Object
…	null
…	null
…	null
72	John, Computer Science, 4.0 — Student Object
…	null
99	null

Hash Tables

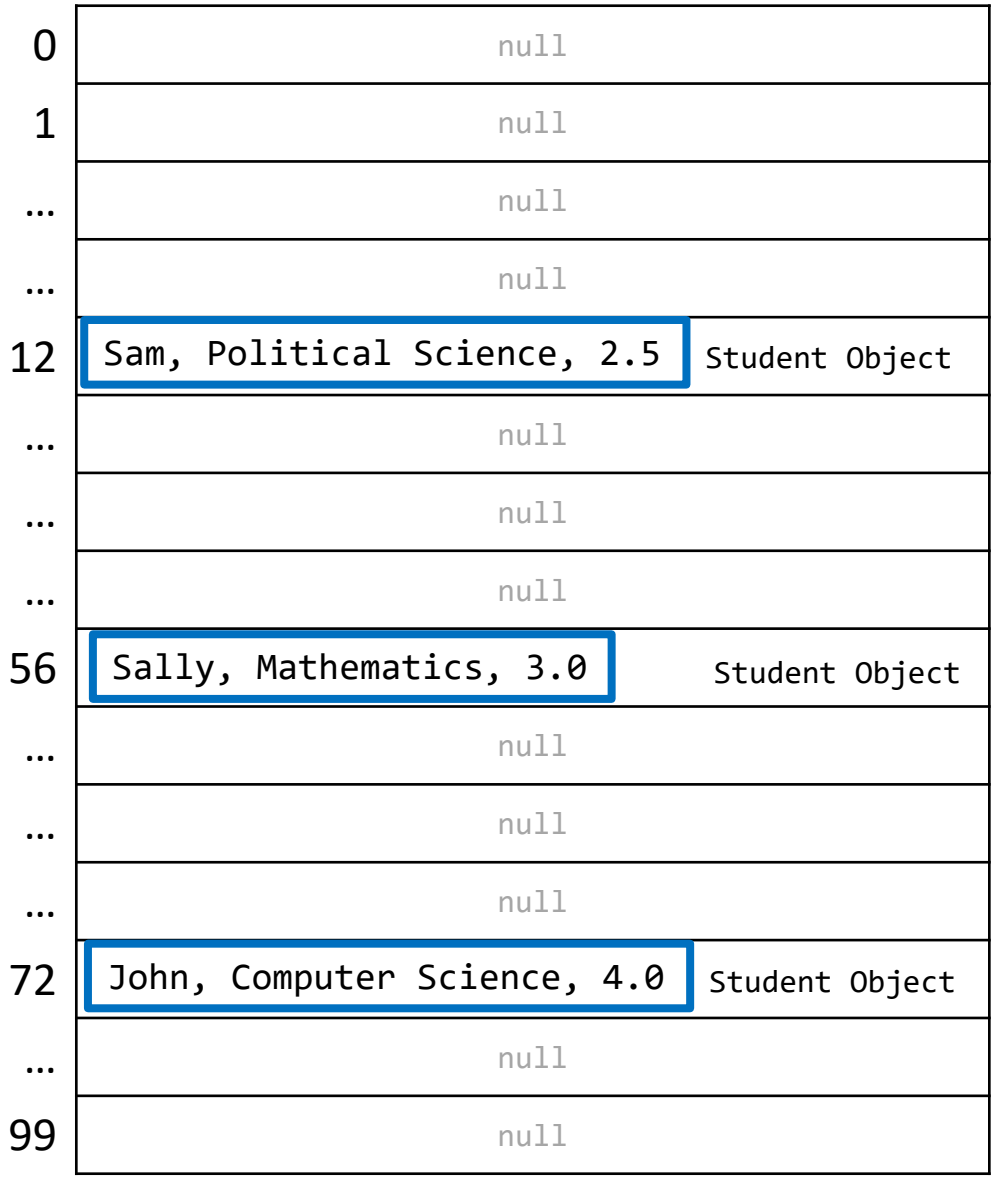
Student ID

Hash Function

`ID % 100`

Lookup (**get**)

```
int arrayIndex = hash(id);
return database[arrayIndex];
```
O(1)



Hash Tables

Student ID

Hash Function

`ID % 100`

Remove Method

Student[] Array n = 100

0	null
1	null
…	null
…	null
12	Sam, Political Science, 2.5 — Student Object
…	null
…	null
…	null
56	Sally, Mathematics, 3.0 — Student Object
…	null
…	null
…	null
72	John, Computer Science, 4.0 — Student Object
…	null
99	null

Hash Tables in Java

Typically, we will never have to create our own **`HashTable`** class, instead we will **import** the one that Java provides

```
import java.util.HashMap;
import java.util.HashSet;
```

Hash Maps

Hash Maps are a collection of key-values pairs (**`Map`**) that uses hashing when inserting, removing, lookup, etc

```
HashMap<String, String> capitalCities = new HashMap<String, String>();
```

This is a HashMap that maps Strings (keys) to Strings (values)

Adding a new Key-Value pair

```
capitalCities.put("England", "London");
capitalCities.put("Germany", "Berlin");
capitalCities.put("Norway", "Oslo");
capitalCities.put("USA", "Washington DC");
```

Retrieving a Value

```
capitalCities.get("England");
```

Removing a Value

```
capitalCities.remove("England");
```

Other Helpful Methods

- `keySet()` → returns set of keys
- `values()` → returns set of values
- `containsKey()`
- `containsValue()`
- `replace()`
- `size()`

Hash Sets

Hash Sets is an implementation of the **`Set`** interface that uses a Hash Map under the hood

A **set** is a collection of elements with no duplicate elements

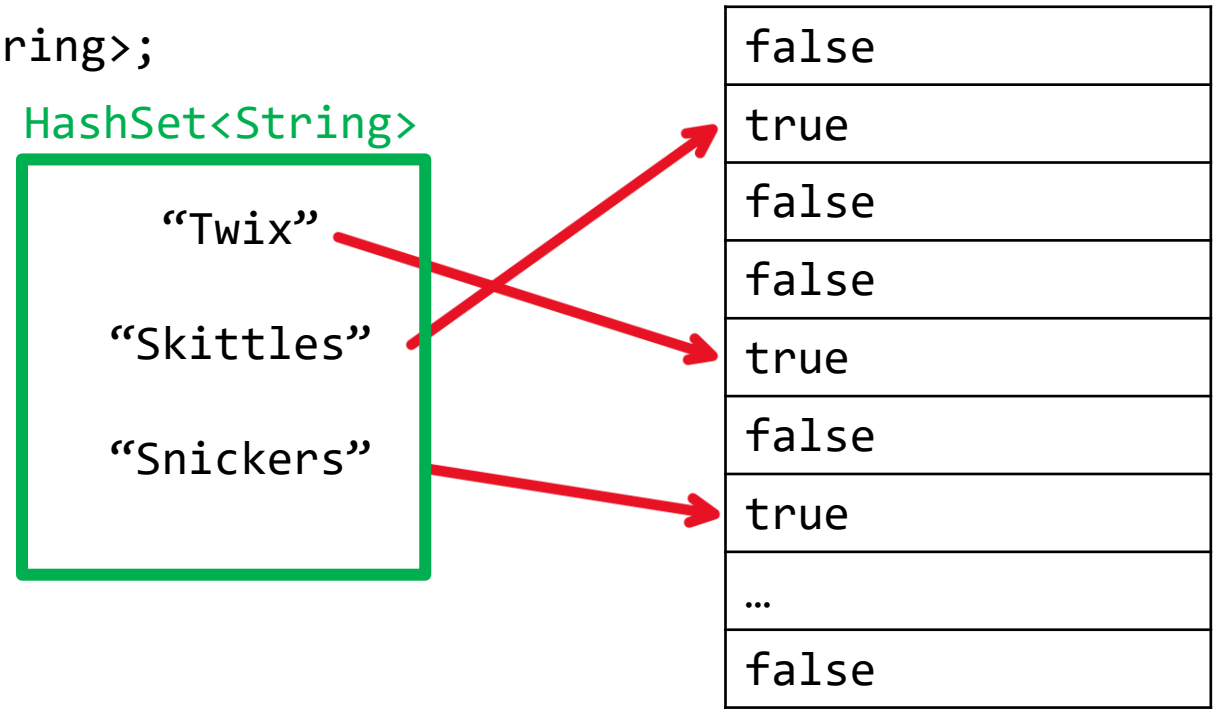
- You can think of this as a List, but without the ability to use indices

```
HashSet<String> candy = new HashSet<String>;
```

Hash Set that stores Strings

```
candy.add("Twix");
candy.add("Skittles");
candy.add("Snickers");

candy.contains("Skittles");
candy.remove("Twix");
```



Hash Sets

Hash Sets is an implementation of the **`Set`** interface that uses a Hash Map under the hood

A **set** is a collection of elements with no duplicate elements

- You can think of this as a List, but without the ability to use indices

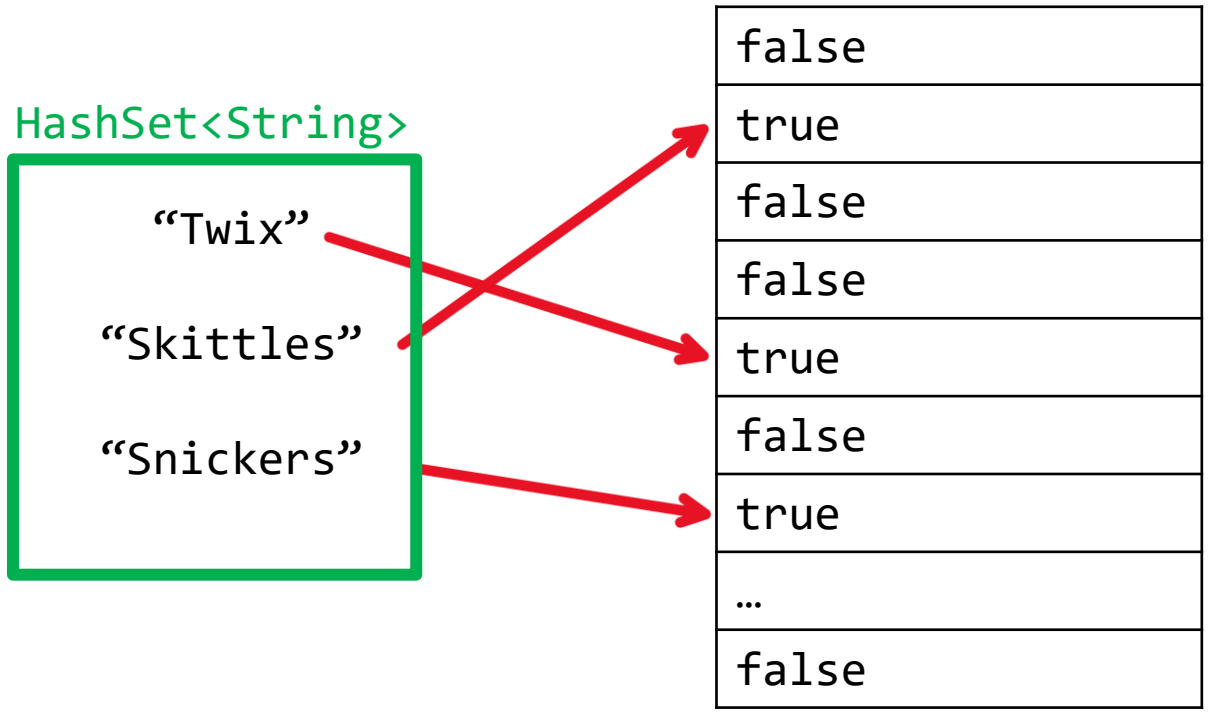
The order in the HashSet may not be the same order you added with

`( “Twix” , “Snickers” , “Skittles” )`

Insertion: O(1)
Lookup: O(1)

When to use HashSet?

- For fast lookups and insertions
- When order doesn’t matter
- Only need unique elements



Today's Mandatory Fun

Updating our Student Database Class

- Replace Array with HashMap
- Replace ArrayList with HashSet

- Write a method that will compute the number of CS majors, Math Majors, History majors, etc
- Add method that will compute which student(s) have a 4.0, 3.0, 3.1, etc

Write a program that will convert an English sentence to sentence in Pirate

"Hello" → "Ahoy"
"Friends" → "Mateys"

HashMap Inner-workings

Every object has a HashCode in Java

Every object has one HashCode and two objects usually don't have the same hash code

```
String dog = "dog";
System.out.println(dog.hashCode());   // 99644

String dog = "dogs";
System.out.println(dog.hashCode());   // 3089079
```

HashMap Inner-workings

Every object has a `HashCode` in Java

Every object has one HashCode and two objects usually don't have the same hash code

```
String dog = "dog";
System.out.println(dog.hashCode());  // 99644

String dog = "dogs";
System.out.println(dog.hashCode());  // 3089079
```

Value is run through the HashMap **`hash()`** method

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

Bucket is determined by:

Bit-level XOR operator

Shift bits right by 16 places

`index = hash & (n - 1)`

(& = bit-level AND operator, n = ***table size****)*

HashMap Inner-workings

Every object has a HashCode in Java

```
String dog = "dog";
System.out.println(dog.hashCode());   // 99644

String dog = "dogs";
System.out.println(dog.hashCode());   // 3089079
```

Every object has one HashCode and two objects usually don't ha

Value is run through the HashMap **hash()** method

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

putVal() is called to place it in array

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        Node<K,V> e; K k;
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                p = e;
            }
        }
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
}
```

HashMap Inner-workings

Every object has a `HashCode` in Java

Every object has one HashCode and two objects usually don't have the same hash code

```
String dog = "dog";
System.out.println(dog.hashCode());  // 99644

String dog = "dogs";
System.out.println(dog.hashCode());  // 3089079
```

Value is run through the HashMap **`hash()`** method

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

Bucket is determined by:

```
index = hash & (n - 1)
```

`put()` calls a `putVal()` method that inserts into the array and deals with collisions

`HashMap` Inner-workings

A new HashMap can fit 16 values

`HashMap` Inner-workings

A new HashMap can fit 16 values

If it ever reaches 75% capacity, it will **double** the array size (16 → 32 → 64 → 128)

HashMap Inner-workings

A new HashMap can fit 16 values

If it ever reaches 75% capacity, it will **double** the array size (16 → 32 → 64 → 128)

When the arrays doubles, we have to re-hash all our Key-Value pairs

HashMap Inner-workings

A new HashMap can fit 16 values

If it ever reaches 75% capacity, it will **double** the array size (16 → 32 → 64 → 128)

When the arrays doubles, we have to re-hash all our Key-Value pairs
→ This most likely will require O(k) moves

HashMap Inner-workings

A new HashMap can fit 16 values

If it ever reaches 75% capacity, it will **double**
the array size (16 → 32 → 64 → 128)

When the arrays doubles, we have to re-hash
all our Key-Value pairs
→ This most likely will require O(k) moves

Why is this not taken into consideration for running time?

HashMap Inner-workings

A new HashMap can fit 16 values

If it ever reaches 75% capacity, it will **double** the array size (16 → 32 → 64 → 128)

When the arrays doubles, we have to re-hash all our Key-Value pairs
→ This most likely will require O(k) moves

→ Why is this not taken into consideration for running time?

(Amortized analysis)

On average, expansion happens very rarely compared to put() method calls when N is really big.

"Since doubling happens exponentially, the total cost of resizing is spread out across many operations, making amortized time per operation O(1)."