Middlebury

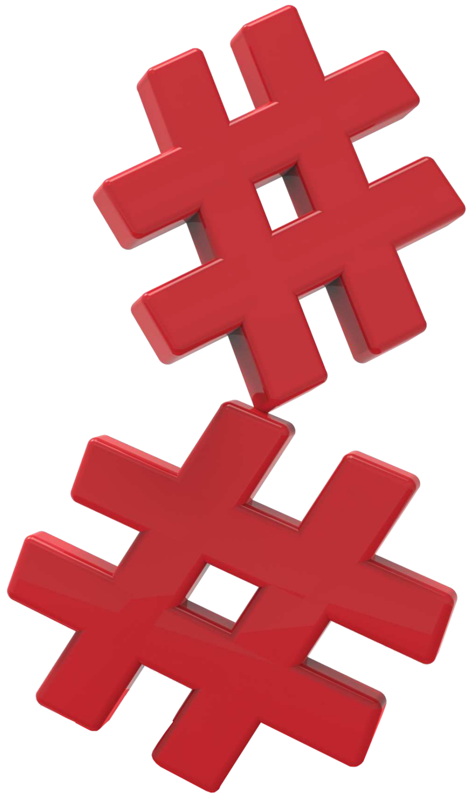
CSCI 201: Data Structures

Fall 2024

Lecture 10T: Hash Tables (Part 1)

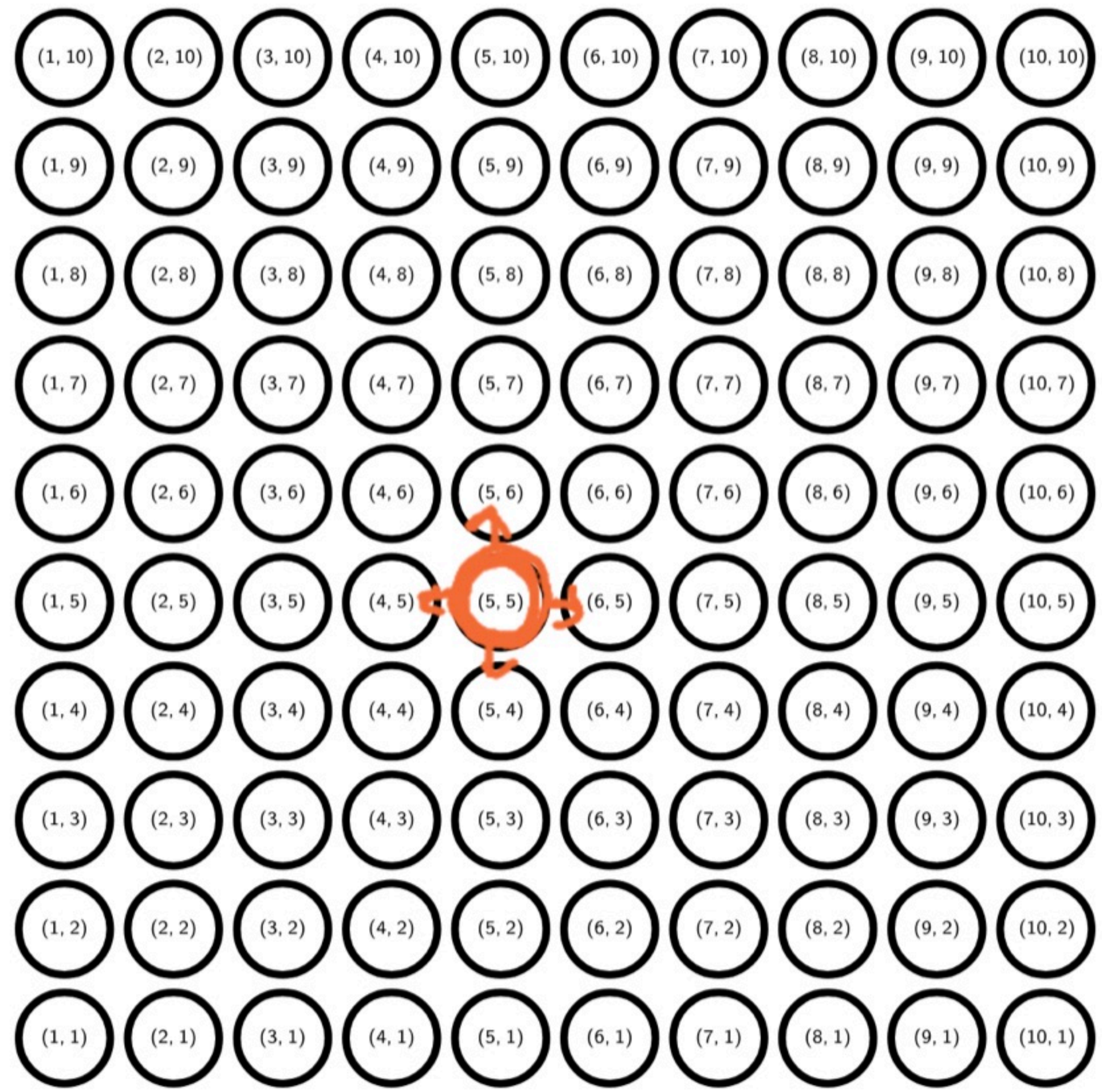
Goals for today:

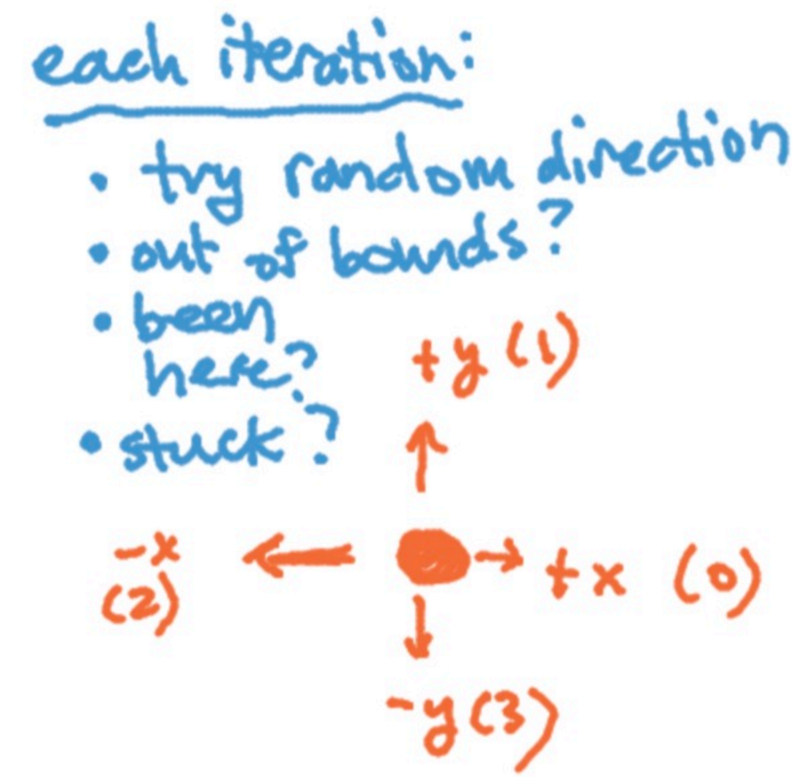
- Solve problems using `Java`'s built-in `HashSet` and `HashMap` (which are built using a **hash table**).
- Write our own `hashCode` and `equals` methods for our custom types in order to use them with `Java`'s built-in `HashSet` and `HashMap`.
- Use a hash function to determine the index of a key in a hash table.
- Practice with bitwise operators.

Consider this problem:

- Suppose we start at some point in a grid and can move left, right, up or down (no diagonal movement).
- We can only step into a grid point that we have not yet visited and cannot step out of bounds of the grid.
- Open `LocationTracker.java` and brainstorm how you would design a solution for the `boolean haveBeenHere(Point point)` method (without a `Set` or `Map`).

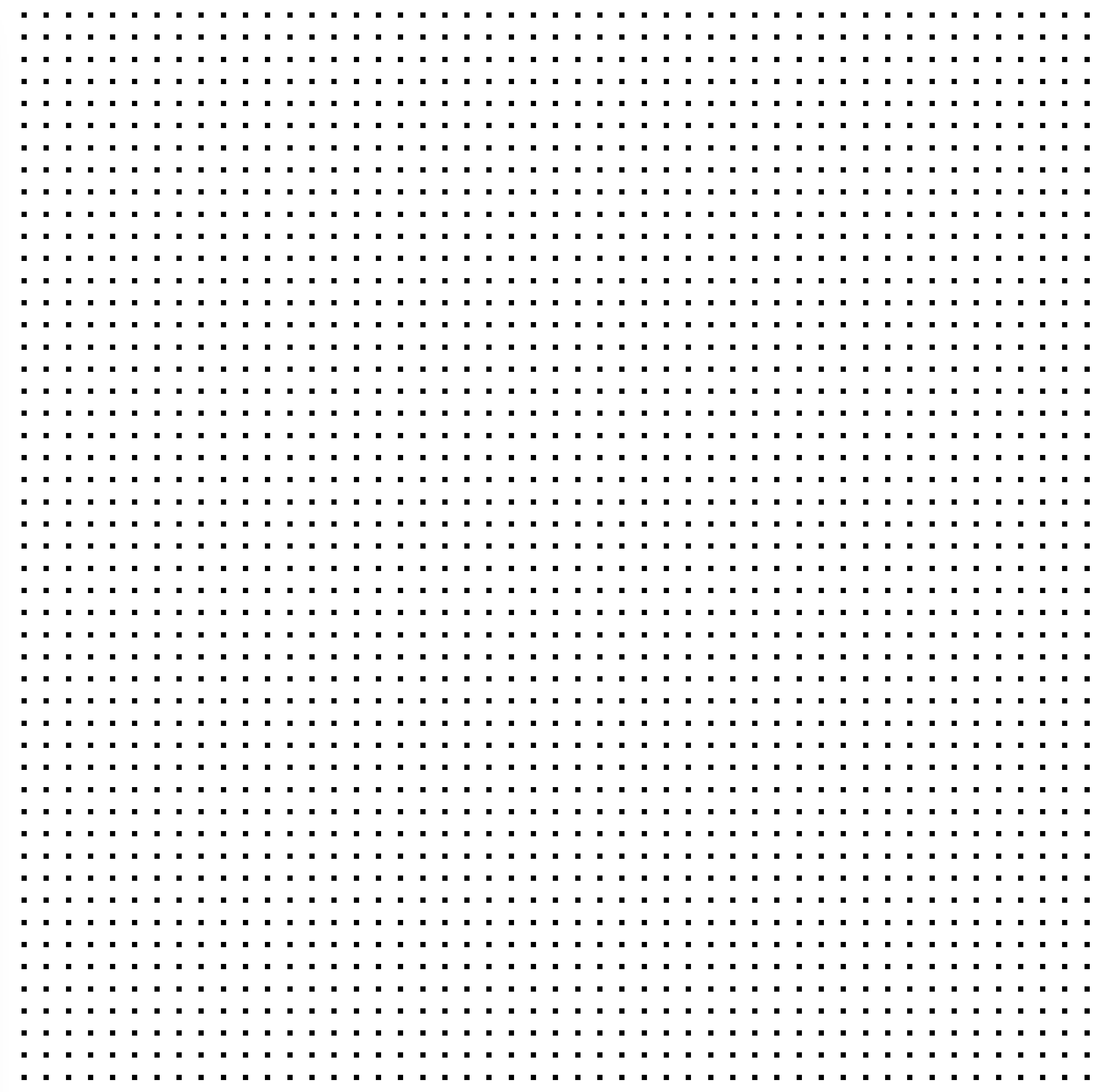




One possible solution:

but...

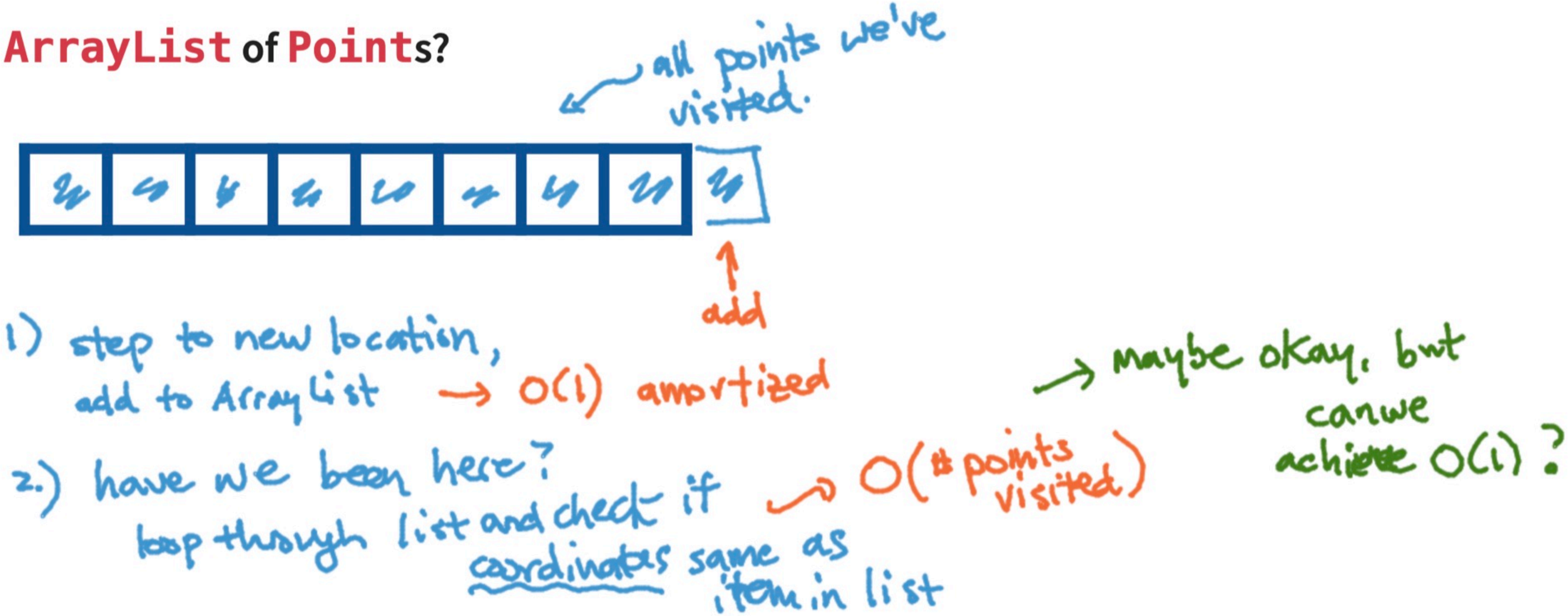
```java
public class PointTracker {
  boolean[][] visited;

  public PointTracker() {
    visited = new boolean[GRID_SIZE][GRID_SIZE];
    for (int i = 0; i < GRID_SIZE; i++) {
      for (int j = 0; j < GRID_SIZE; j++) {
        visited[i][j] = false;
      }
    }
    location = new Point(0, 0); // or in middle
    move(location);
  }

  public boolean haveBeenHere(Point point) {
    return visited[point.x][point.y];
  }

  public void move(Point newLocation) {
    visited[newLocation.x][newLocation.y] = true;
    location = newLocation;
  }
}
```



Is there a way to use less memory?

We only want to keep track of points we've visited.

ArrayList of Points?



But we also want to be able to (quickly) look up if we've visited a point.

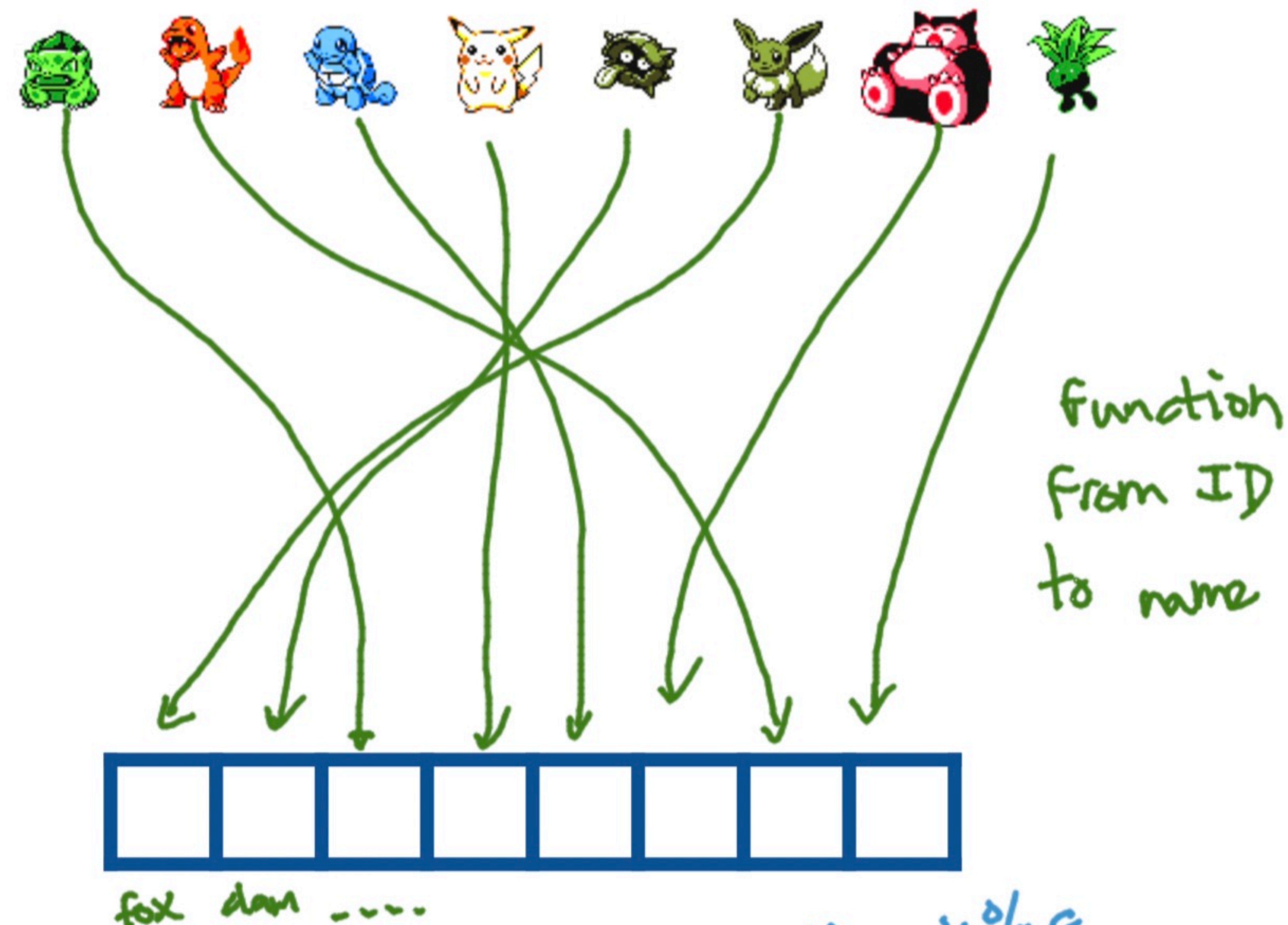
Idea: use an array, but find a way to map items to array indices.

Introducing *hash functions*: map items (keys) to an array index.

Imagine our keys are **integers** (we'll come back to custom types like `Point` later).

Example: storing student ID to name.

- **Green onion:** 00837194 2
- **Fire lizard:** 00833462 6
- **Blue turtle:** 00749132 4
- **Electric mouse:** 00678395 3
- **Clam:** 00889321 1
- **Fox:** 00765432 0
- **Sleepy panda:** 00812493 5
- **Radish:** 00754639 7



one option: store huge array from 0 → max ID
works but uses a lot of memory

another method: map ID → array index

propose: id % 8

to look up name from ID: ① evaluate function id % 8
② retrive name at the index

Hash functions: map the universe of keys to a restricted range (e.g. the size of an array m).

What makes a good hash function?

- **Deterministic:** $h(k)$ should always return the same value.
- **Fast to evaluate:** if it's expensive (e.g. $\log n$) then we don't gain anything by using a table.

The challenge is that we generally don't know the distribution of the keys.

Example: division method ($h(k) = k \bmod m$).

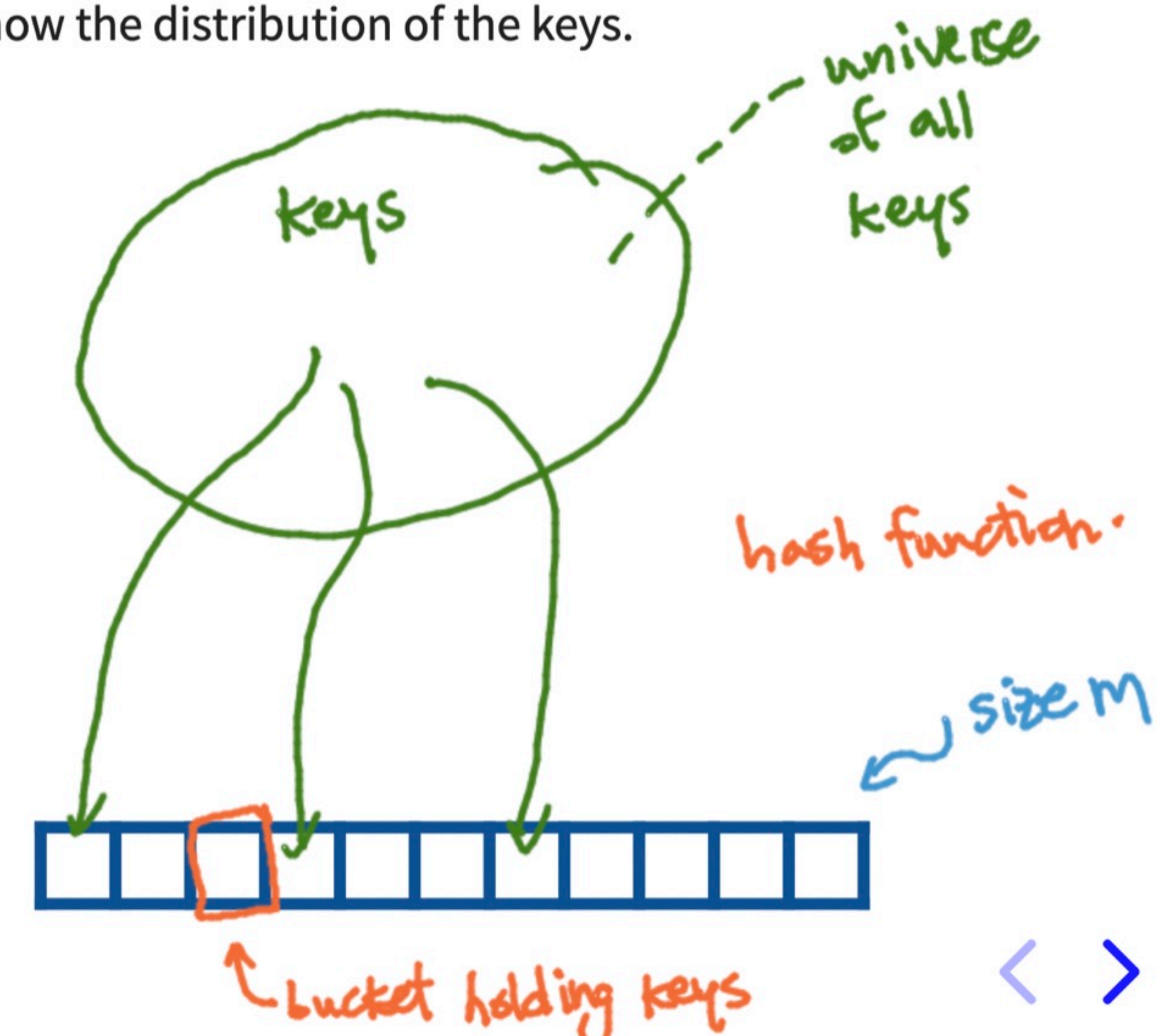
k % m

m	k	$h(k)$
11	25	3
11	1	1
11	17	6
13	133	3
13	7	7
13	25	12

u = # possible keys

some keys might map to same array index

if u > m at least one array index will have 2 keys



Good rule of thumb (for division method) is that m is a prime number not too close to a power of 2.

Concern: what if multiple keys map to the same array index?

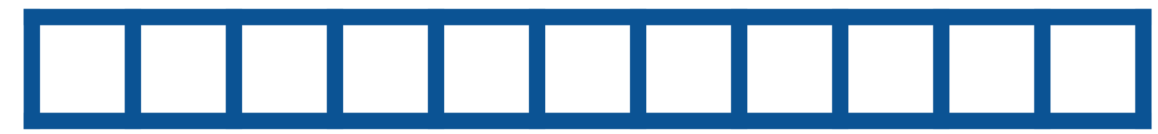
COLLISION

m	k	$h(k)$
11	25	3
11	1	1
11	17	6
11	34	1

map to same array index

Implementing a hash table for custom types.

- Override `public int hashCode()`.
- To `add(K key)` or `put(K key, V value)`:
  1. Evaluate `key.hashCode()` to get an integer.
  2. Use the result to determine the array index.
  3. Place item in bucket at array index.
- To `get(K key)`:
  1. Evaluate `key.hashCode()` to get an integer.
  2. Use the result to determine the array index.
  3. Retrieve item in bucket at array index.
- But as we said, there may be collisions (or multiple keys that map to the same bucket). We need to iterate through items in the bucket to find the item that is associated with the `key`:
  - So we need to find the item with a `key` (i.e. check `key` equivalence).
  - Requires overriding `public boolean equals(Object otherObj)`.
- **Load factor:** $\alpha = n/m$.
  - n: number of items (`size`).
  - m: number of buckets (length of table array, i.e. capacity).
  - Higher α: potentially go through many items in one bucket to search.
  - Low α means wasted memory.
- Resize the table if load factor is outside acceptable range.
  - And be sure to rehash items (recompute table indices).



```java
class Point {
  public int x;
  public int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  @Override
  public int hashCode() {
    // TODO how should we hash two integer values?
  }

  @Override
  public boolean equals(Object otherObject) {
    Point otherPoint = (Point) otherObject;
    return (otherPoint.x == x) && (otherPoint.y == y);
  }
}
```

A detailed look into how Java does this (OpenJDK).

https://github.com/openjdk/jdk/blob/7b0f273e37625461baa333a3ef20fbbd93647243/src/java.base/share/classes/java/util/HashMap.java#L320

```java
/**
 * Computes key.hashCode() and spreads (XORs) higher bits of hash
 * to lower.  Because the table uses power-of-two masking, sets of
 * hashes that vary only in bits above the current mask will
 * always collide. (Among known examples are sets of Float keys
 * holding consecutive whole numbers in small tables.)  So we
 * apply a transform that spreads the impact of higher bits
 * downward. There is a tradeoff between speed, utility, and
 * quality of bit-spreading. Because many common sets of hashes
 * are already reasonably distributed (so don't benefit from
 * spreading), and because we use trees to handle large sets of
 * collisions in bins, we just XOR some shifted bits in the
 * cheapest possible way to reduce systematic lossage, as well as
 * to incorporate impact of the highest bits that would otherwise
 * never be used in index calculations because of table bounds.
 */
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

- Table (array) size is always a power of 2.
- Table index computed from `hash(key) & (m - 1)` where m is the capacity (table length).
- `^` means bitwise XOR (exclusive OR): e.g. `01101 ^ 11001` is
- `>>>` means unsigned right shift (here, by 16 bits): e.g. `01101010 >>> 4` is
- `&` means bitwise AND: e.g. `00110 & 111` is
- In your **Terminal**, type `jshell` and use `Integer.toBinaryString` to try these out! `Ctrl-D` to exit.

Exercise: compute the hash table indices for the following `Points` (using `Java`'s technique).

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

- Starting with a table size `m = 16`.
- Table index = `hash(key) & (m - 1)`.
- Point 1: (1, 1)
- Point 2: (12345, 678)

```java
class Point {
  public int x;
  public int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  @Override
  public int hashCode() {
    return 31 * x + y;
  }

  @Override
  public boolean equals(Object otherObject) {
    Point otherPoint = (Point) otherObject;
    return (otherPoint.x == x) && (otherPoint.y == y);
  }
}
```

Additional notes:

- For more hash functions, see: https://en.wikipedia.org/wiki/List_of_hash_functions
- **Lab 7** due tonight.
- **Homework 8** due on Friday 11/15: use a `TreeMap` to solve a problem and then implement a DIY-version based on what your algorithm needs.
- **Next class:** how can we handle collisions?