Writing efficient application code is a major factor in producing high performing applications

Big-O Notation

Measure of the performance (time complexity) of an algorithm

Describes the upper bound of the execution time of an algorithm relative to the size of the input

```
int[] numbers = {1,2,3,4};

index = 1;

int number = numbers[index];
```

O(1)

- An algorithm / operation that always completes in constant time regardless of the size of the input

```java
int[] nums = {10, 20, 15, 22, 35};

Arrays.sort(nums);

int key = 22;

int index =
Arrays.binarySearch(nums, key);
```

O(log(n))

- **An algorithm whose completion time is a log function of the size of its input**

```java
int[] nums = {10, 20, 15, 22, 35};
int key = 22; int index = -1;
for (int i=0; i<nums.length; i++){
  if (nums[i] == key) {
    index = i;
    break;
  }
}
```

O(n)

- An algorithm whose completion time grows in direct linear proportion to the size of the input

O(n log(n))

```
List<String> list = new
ArrayList<>();

list.add("foo");

list.add("bar");

list.add("baz");

Collections.sort(list);
```

- An algorithm with a runtime that's directly proportional to the size of the input multiplied by the log of the size of the input

```java
int[] nums = {10, 20, 15, 22, 20};
boolean duplicateFound = false;
for (int i=0; i<nums.length; i++){
  for (int j=0; j<nums.length; j++) {
    if (i == j) continue;
    if (nums[i] == nums[j])
      duplicateFound = true;
  }
}
```

$O(n^2)$

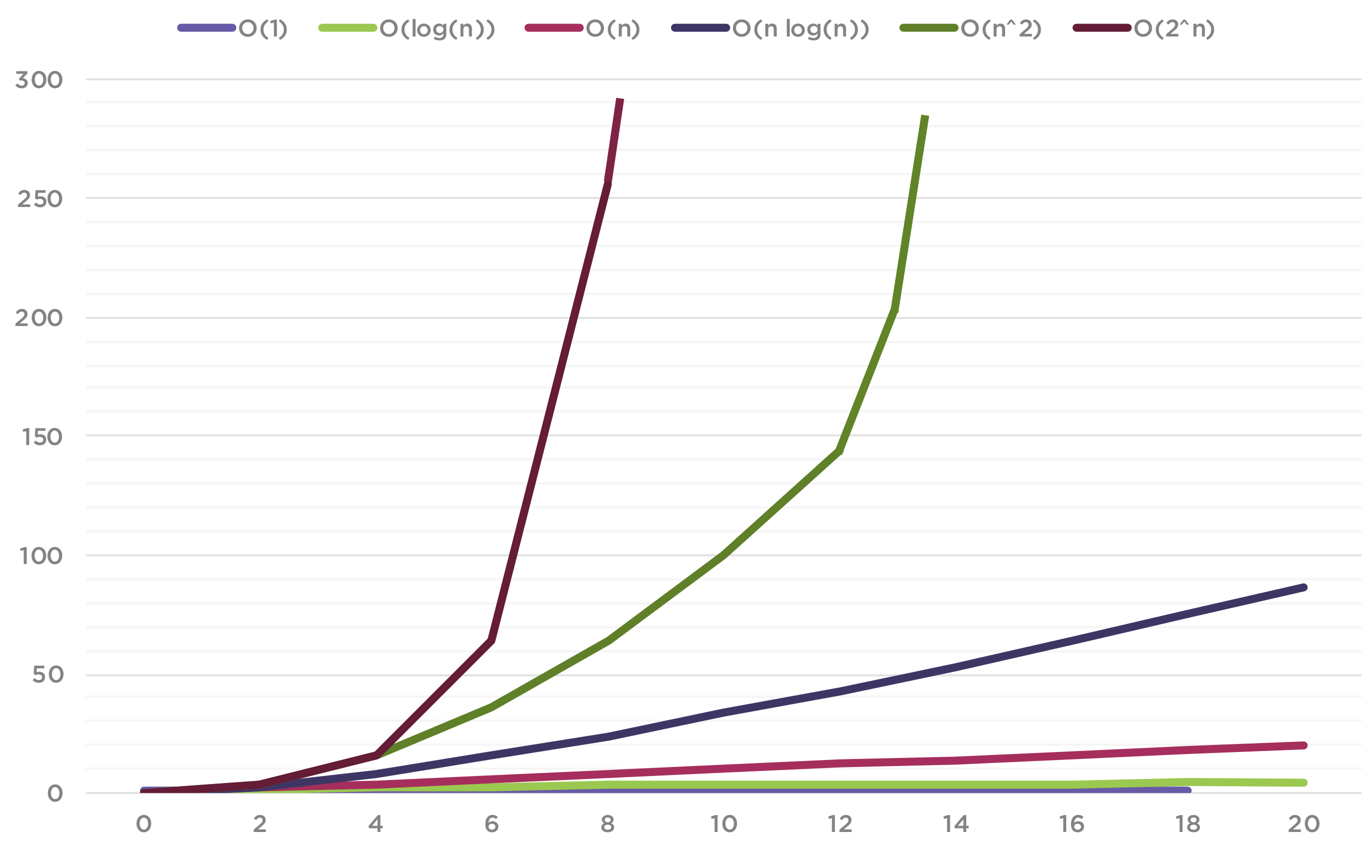
- **An algorithm whose completion time grows in direct quadratic proportion to the size of the input**

```
int Fibonacci(int number) {
  if (number <= 1) return number;

  return Fibonacci(number - 2) +
            Fibonacci(number - 1);
}
```

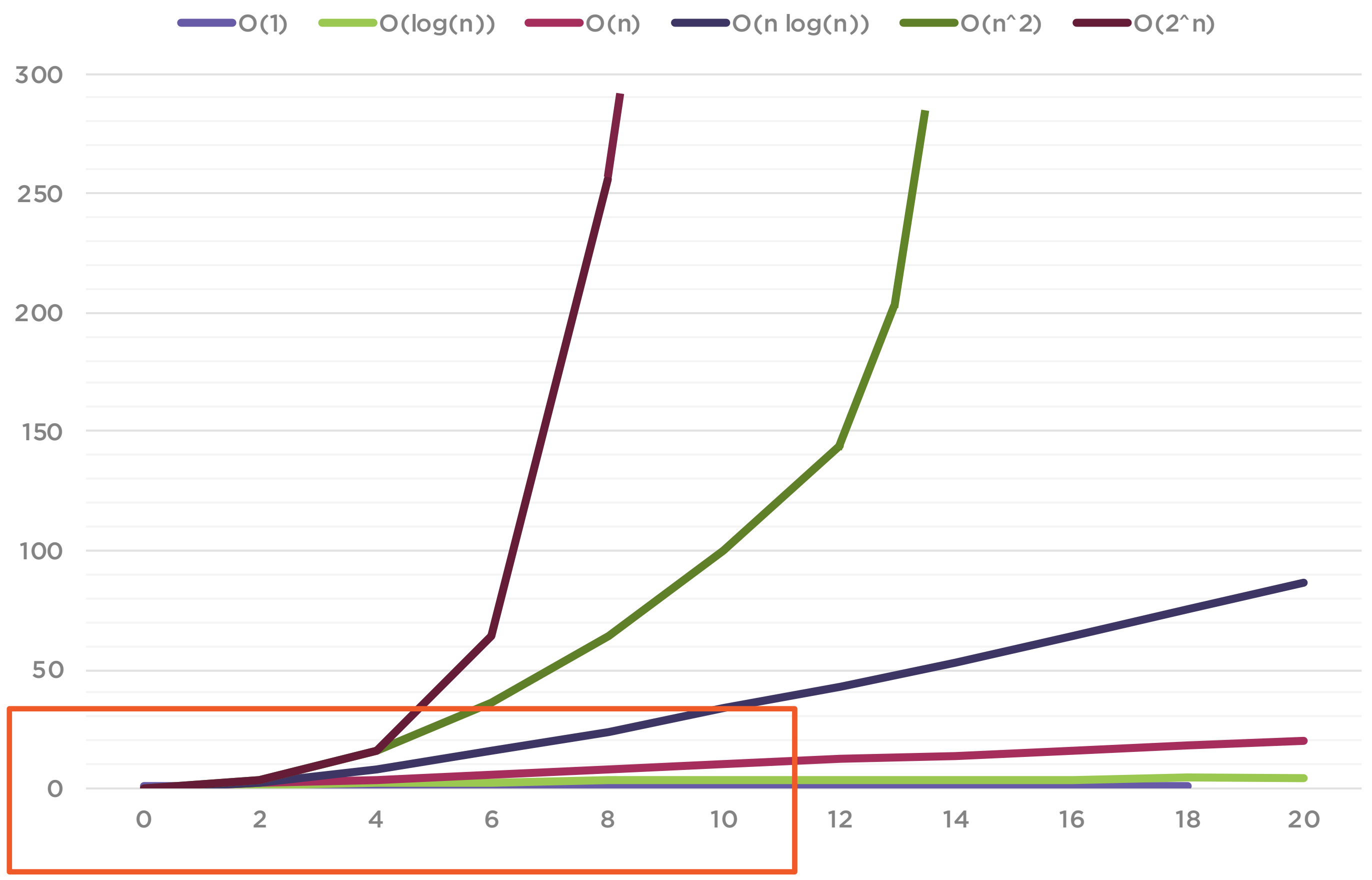
$O(2^n)$

- **An algorithm that doubles in runtime with each addition to the input data set**

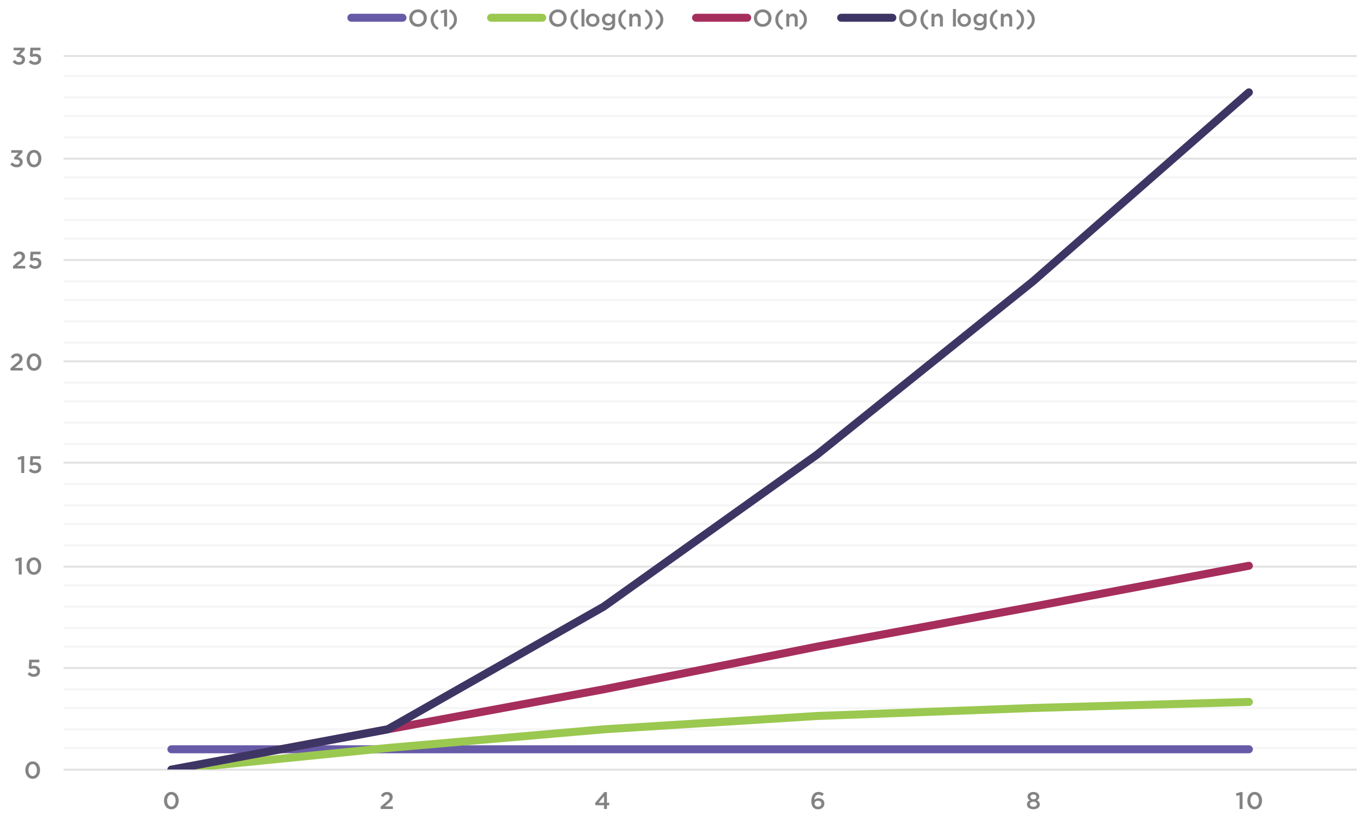
Big-O Complexity Chart



Big-O Complexity Chart



Big-O Complexity Chart



Hybrid Algorithms

O(n)

< 8 items

O(n log(n))

>= 8 items

```
int Fibonacci(int number) {
  if (number <= 1) return number;

  return Fibonacci(number - 2) +
           Fibonacci(number - 1);
}
```

```
int Fibonacci(int number) {
  if (number <= 1) return number;

  return Fibonacci(number - 2) +
            Fibonacci(number - 1);
}
```

```
int Fibonacci(int number) {
  int a = 0; int b = 1;
  for (int i = 0; i < number; i++) {
    int c = a + b;
    a = b;
    b = c;
  }
  return a;
}
```

```
int Fibonacci(int number) {
  if (number <= 1) return number;

  return Fibonacci(number - 2) +
            Fibonacci(number - 1);
}
```

```
int Fibonacci(int number) {
  int a = 0; int b = 1;
  for (int i = 0; i < number; i++) {
    int c = a + b;
    a = b;
    b = c;
  }
  return a;
}
```

Overview

Using Java Data Structures

Setting the ArrayList Initial Size

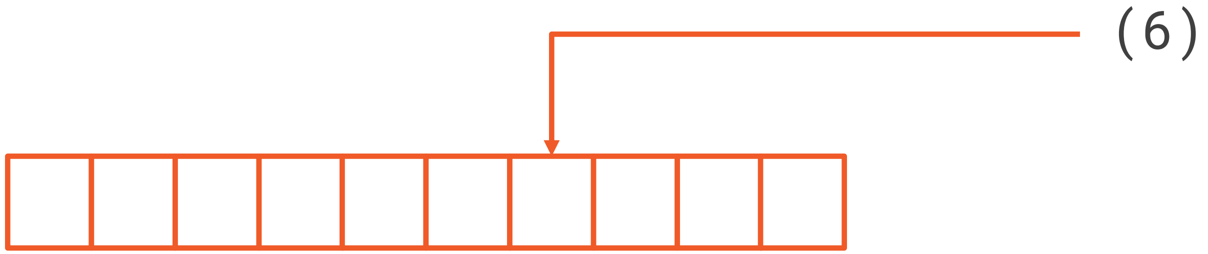
Optimizing HashMap Performance

Using Java Data Structures

Common Java Data Structures

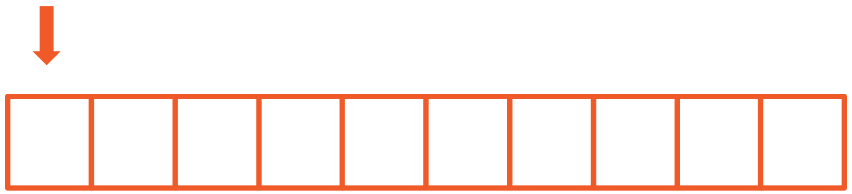
- Array
- ArrayList
- HashMap
- HashSet

Arrays / ArrayLists



O(1)

Arrays / ArrayLists



O(n)

ArrayList Use Case

Adding items to a collection + processing all items or accessing a specific item

Adding to the end of an ArrayList is O(1)

LinkedList is more performant if you frequently need to add an item to the head or middle of a list

LinkedList Use Cases

Implementing a stack or queue

Implementing an in-place filter

Disadvantage of LinkedLists:

- Random access is slow

- Iterating a LinkedList is not as fast as iterating an ArrayList

ArrayList vs. LinkedList

ArrayList

- ✓ Fast random access
- ✓ Fast iteration
- ✗ Fast in-place removal
- ✗ Fast mid-list addition

LinkedList

- ✗ Fast random access
- ✗ Fast iteration
- ✓ Fast in-place removal
- ✓ Fast mid-list addition

```java
ArrayList<String> filterItems(ArrayList<String> items, String subStr) {
  ArrayList<String> newItems = new ArrayList<>();

  for (String item: items)
    if (item.contains(subStr))
      newItems.add(item);

  return newItems;
}
```

```
ArrayList<String> filterItems(ArrayList<String> items, String subStr) {
  ArrayList<String> newItems = new ArrayList<>();

  for (String item: items)
    if (item.contains(subStr))
      newItems.add(item);

  return newItems;
}
```

Trade-offs: Filtering a List

Use a LinkedList	Modify an ArrayList	Returning a new ArrayList

Trade-offs: Filtering a List

Use a LinkedList

Modify an ArrayList

Returning a new ArrayList

Trade-offs: Filtering a List

Use a LinkedList	Modify an ArrayList	Returning a new ArrayList

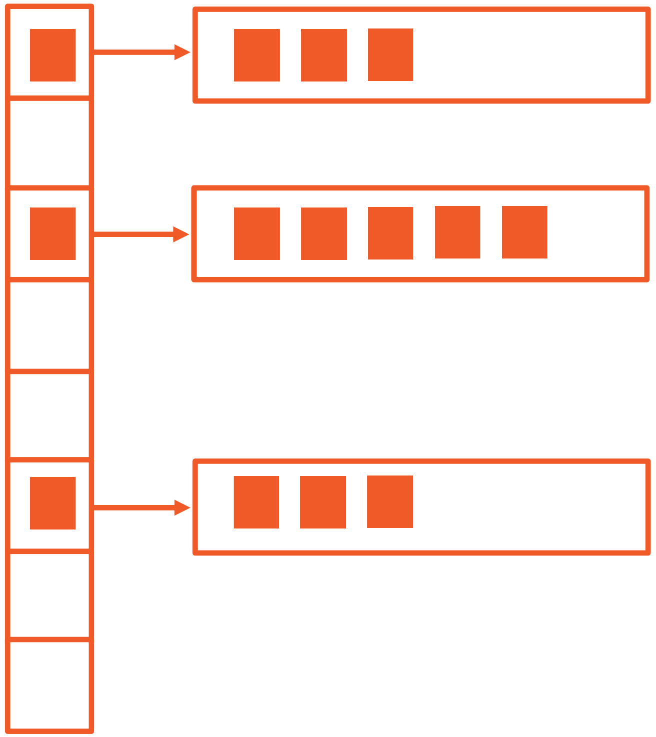
HashMaps

Offer an average time complexity of O(1) on searches, insertions, and deletions

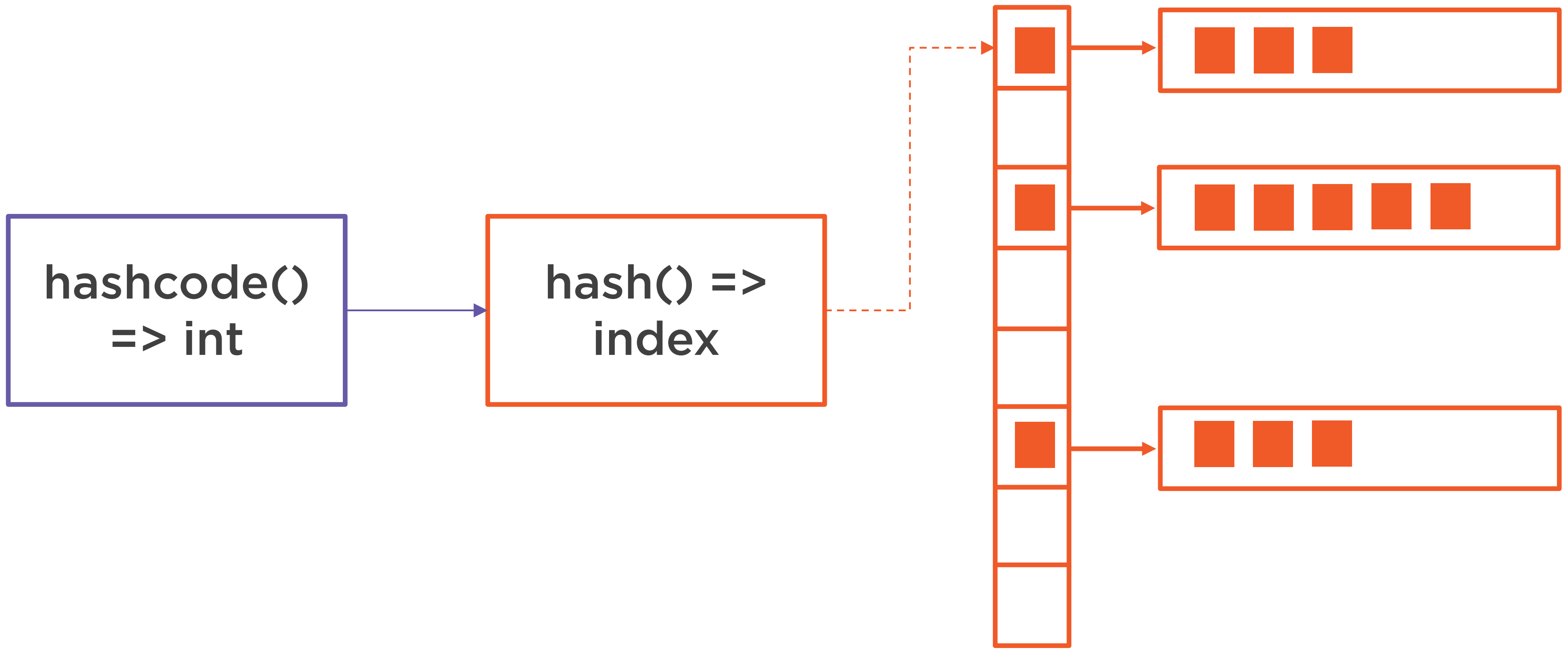
Key-value structure

The hashcode of the key is used to determine where to store and retrieve an entry

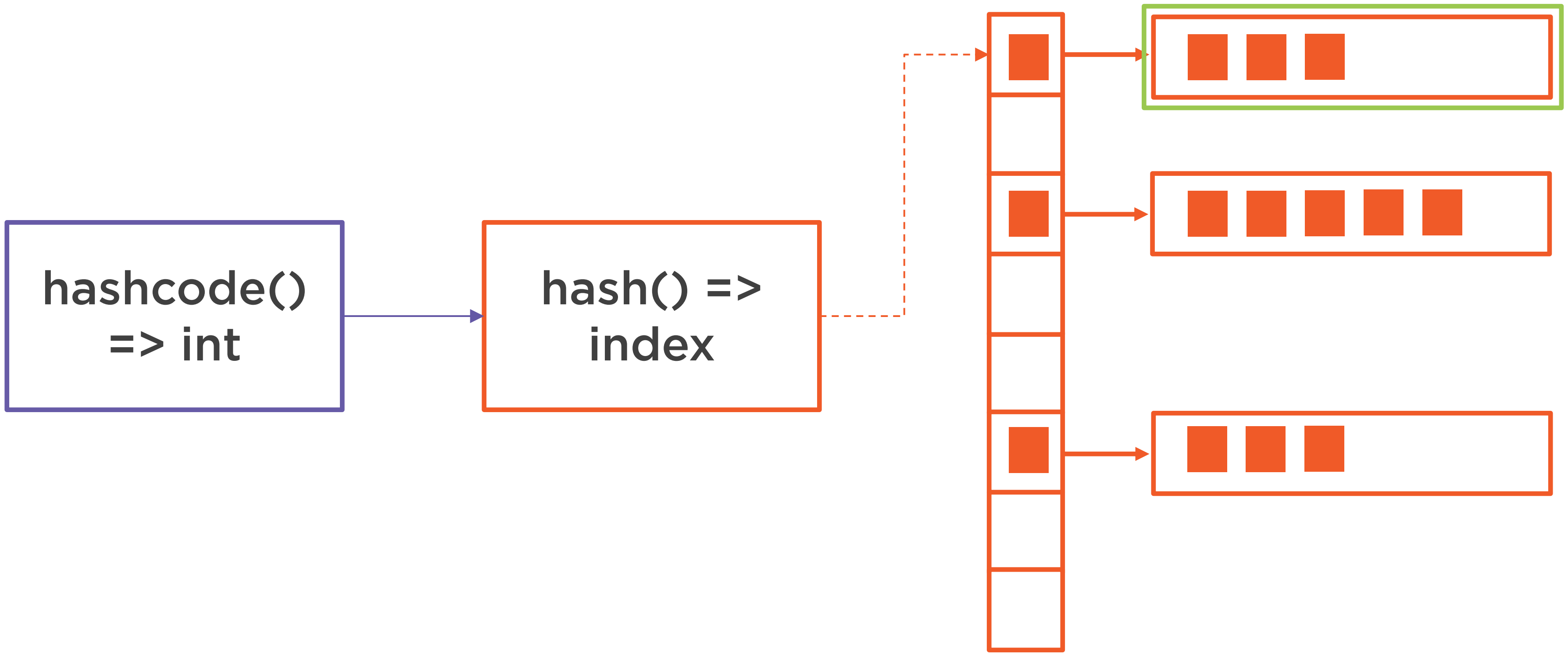
HashMaps



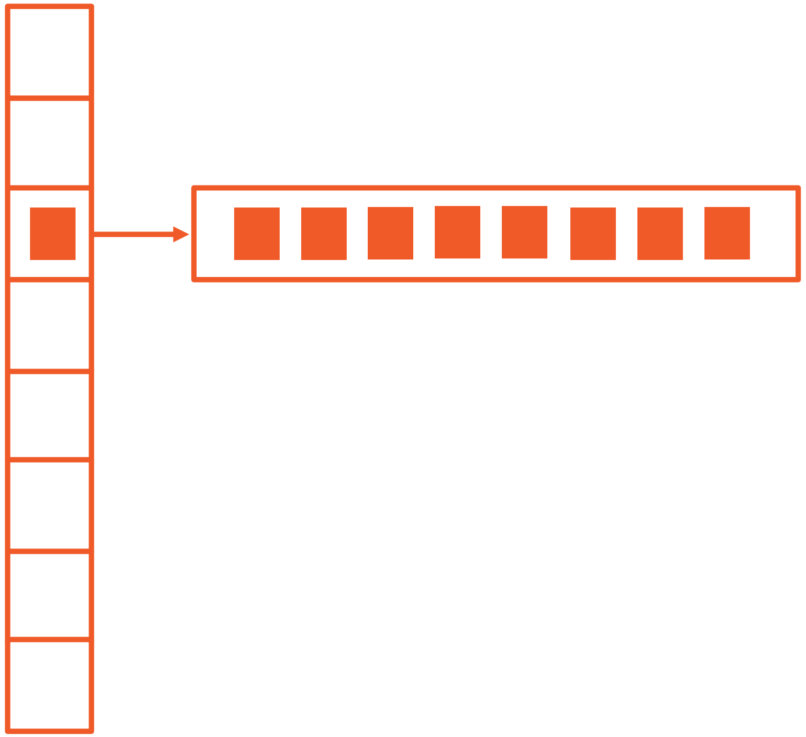
HashMaps



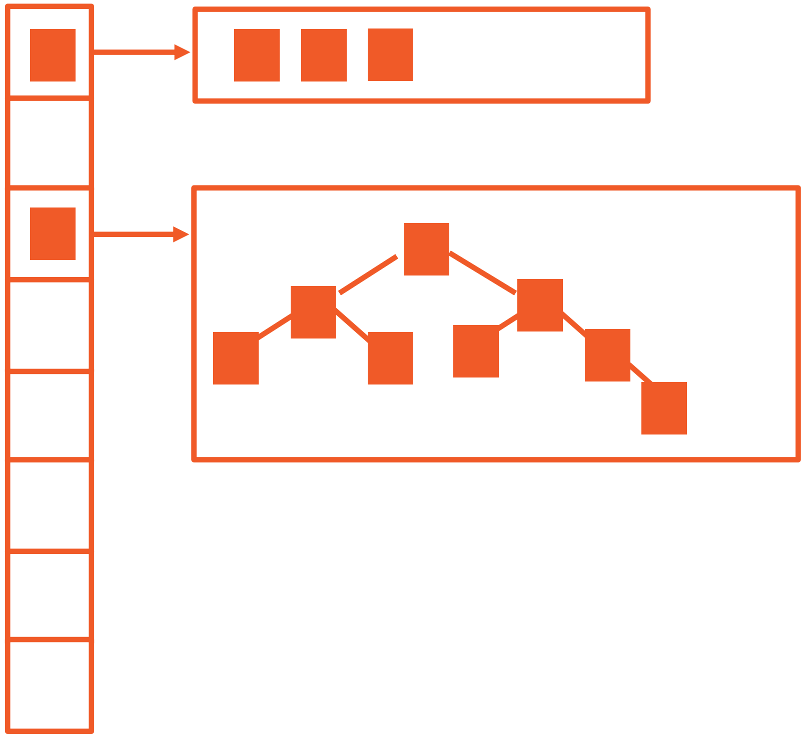
HashMaps



HashMaps



HashMaps



Other Map Implementations

LinkedHashMap: maintains insertion order

TreeMap: sorted map

Sorted Collections

Getting the lowest item, highest item and searching the collection can be done in $O(\log(n))$

Insertions also take $O(\log(n))$

Sorted Collections

ArrayList + Collections.sort()	TreeSet / TreeMap
Many insert operations	Relatively few insert operations
Few sorted traversals	Many sorted traversals

TreeSet Alternatives

Use an ArrayList and maintain sort order by hand: O(log(n)) + O(n)

Use a TreeMultiSet from Google's Guava library

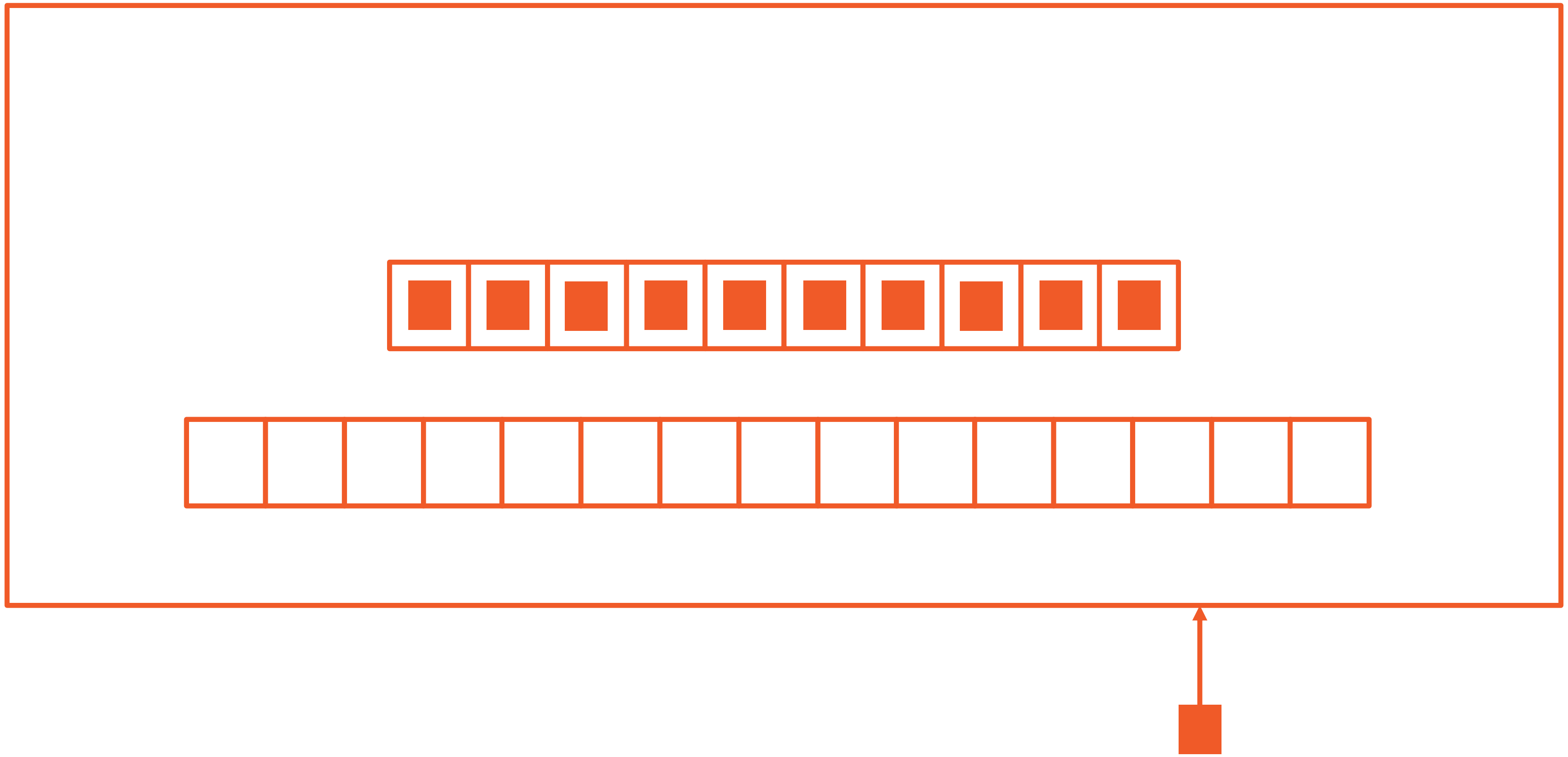
Other Common Collection Implementations

Guava

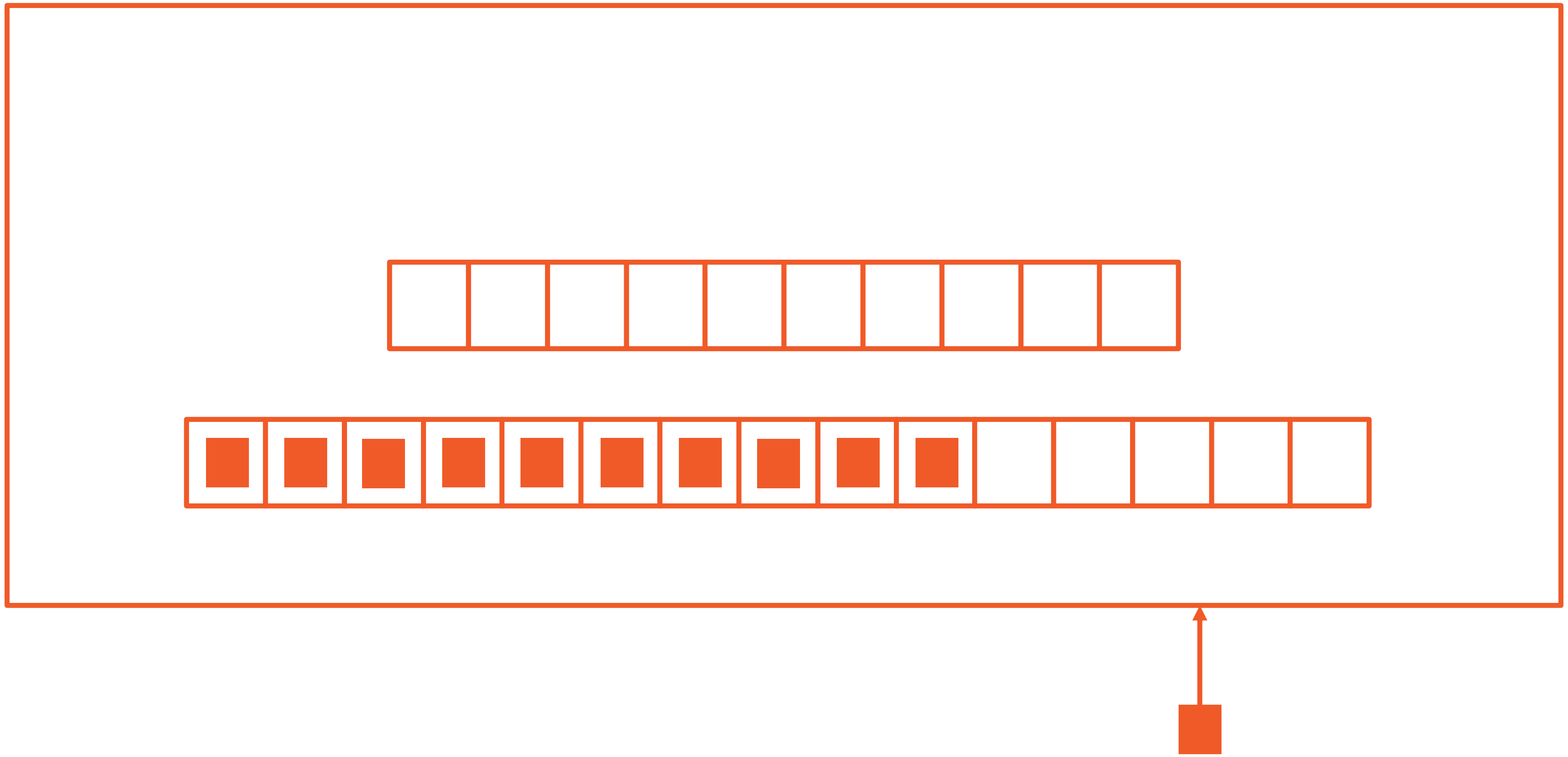
Apache Commons

Setting the ArrayList Initial Size

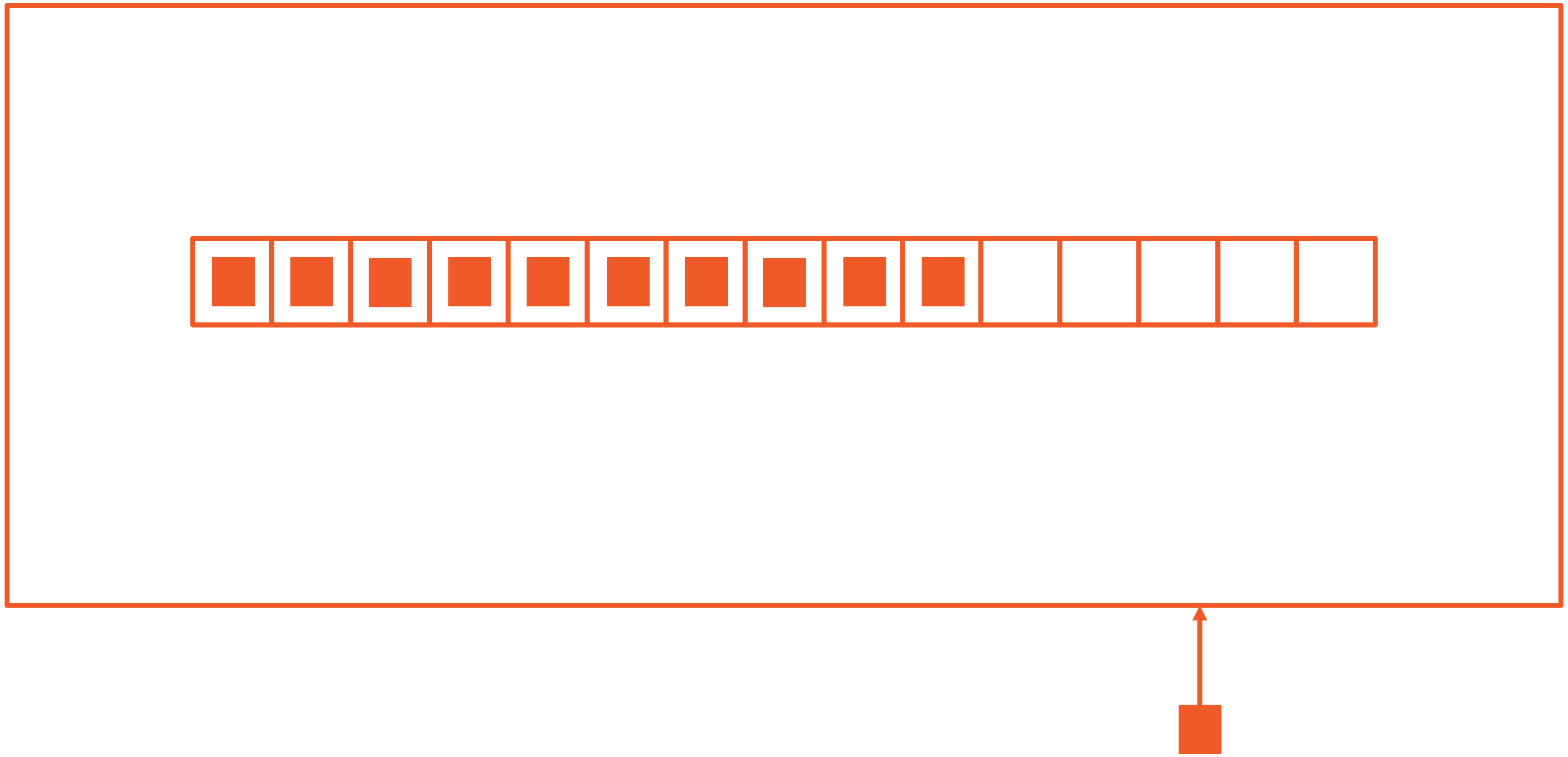
ArrayLists



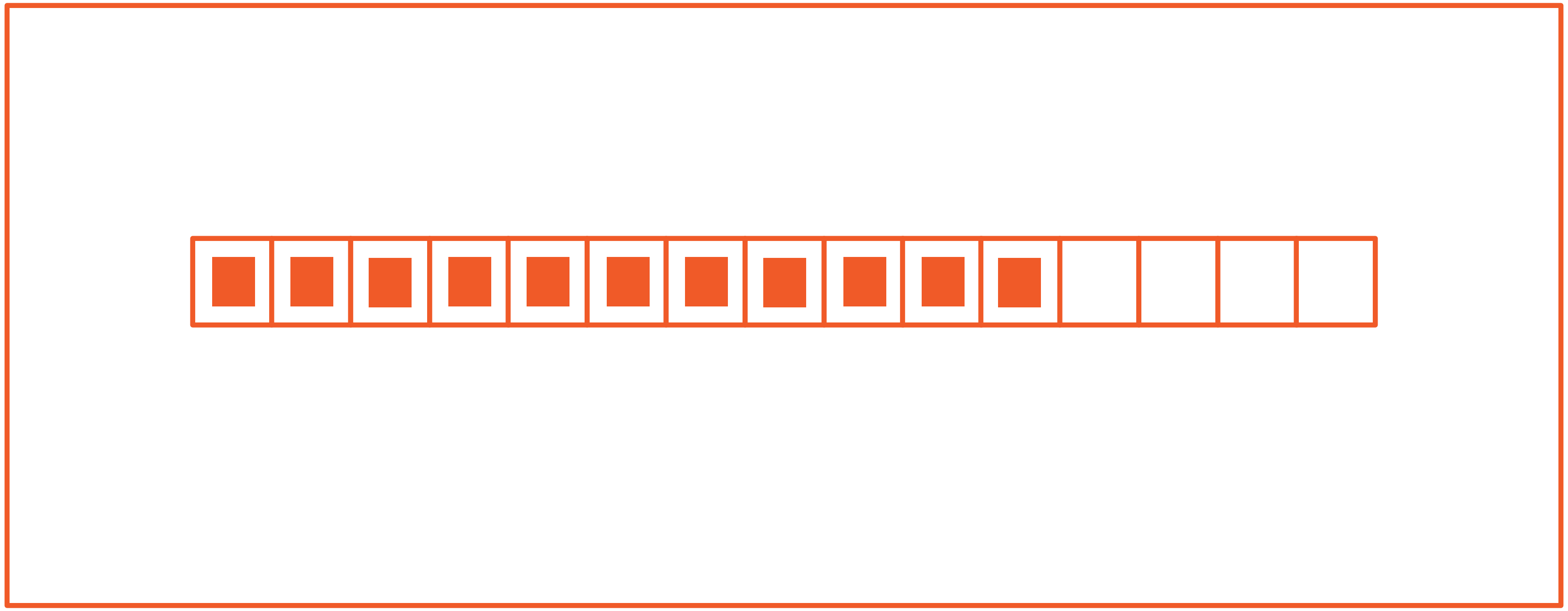
ArrayLists



ArrayLists



ArrayLists



Setting the ArrayList Initial Size

```java
/**
 * Constructs an empty list with the specified initial capacity.
 *
 * @param  initialCapacity  the initial capacity of the list
 * @throws IllegalArgumentException if the specified initial capacity
 *         is negative
 */
public ArrayList(int initialCapacity)
```

Setting the ArrayList Initial Size

```
ArrayList<Object> list = new ArrayList<>(100);
```

```java
List<Integer> doStuff(int[] arr) {
  List<Integer> list = new
ArrayList<>(arr.length);
  ...
  return list;
}
```

◀ ArrayList size is known up front

```
List<Object> process(Request rqst) {

  List<Object> list = new
ArrayList<>();

  while(rqst.hasNext()) {

    list.add(rqst.next());

  }

  ...

  return list;

}
```

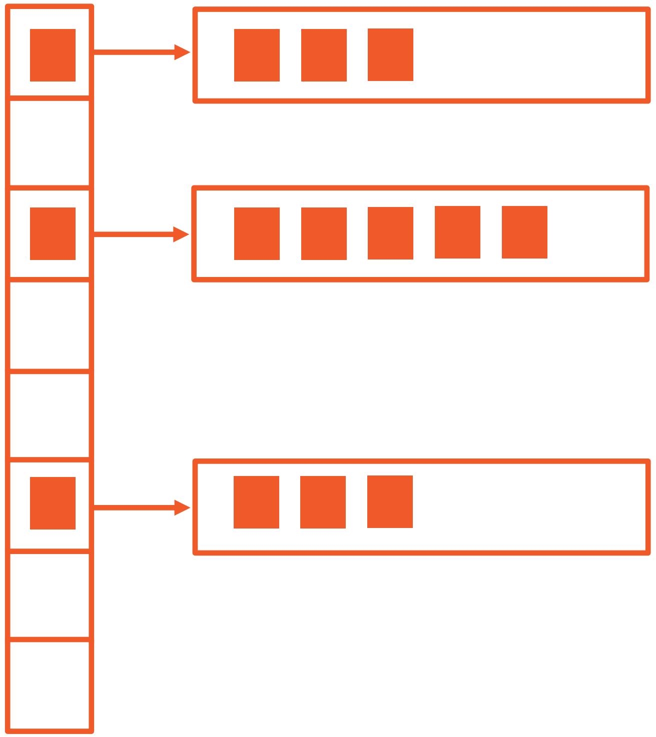
◀ ArrayList size is unknown

```java
List<Object> process(Request rqst) {

  List<Object> list = new
ArrayList<>(AVG_INITIAL_RQST_SIZE);

  while(rqst.hasNext()) {

    list.add(rqst.next());

  }

  ...

  return list;

}
```

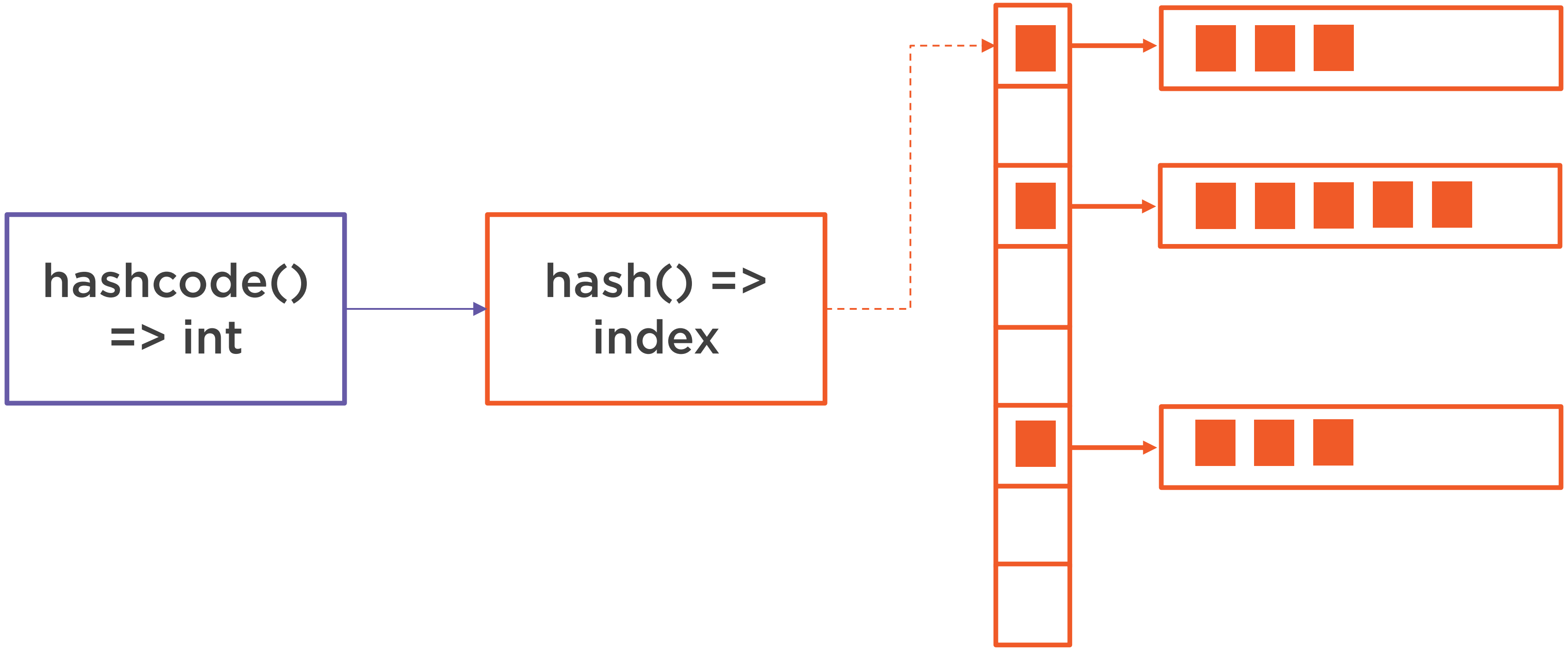
- Using an estimated average for the initial size

Optimizing HashMaps

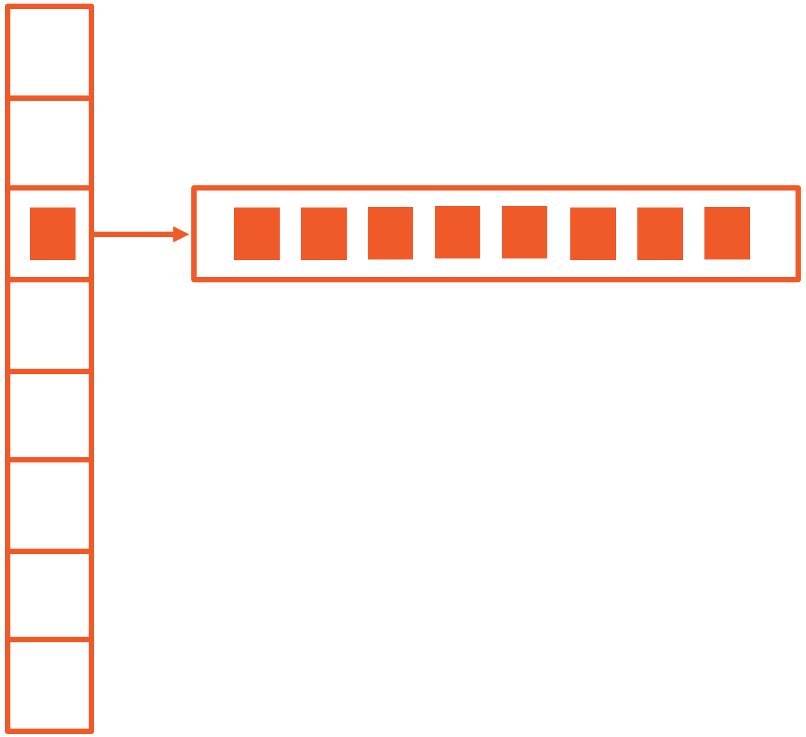
HashMaps



HashMaps



HashMaps



Generating Good Hashcodes

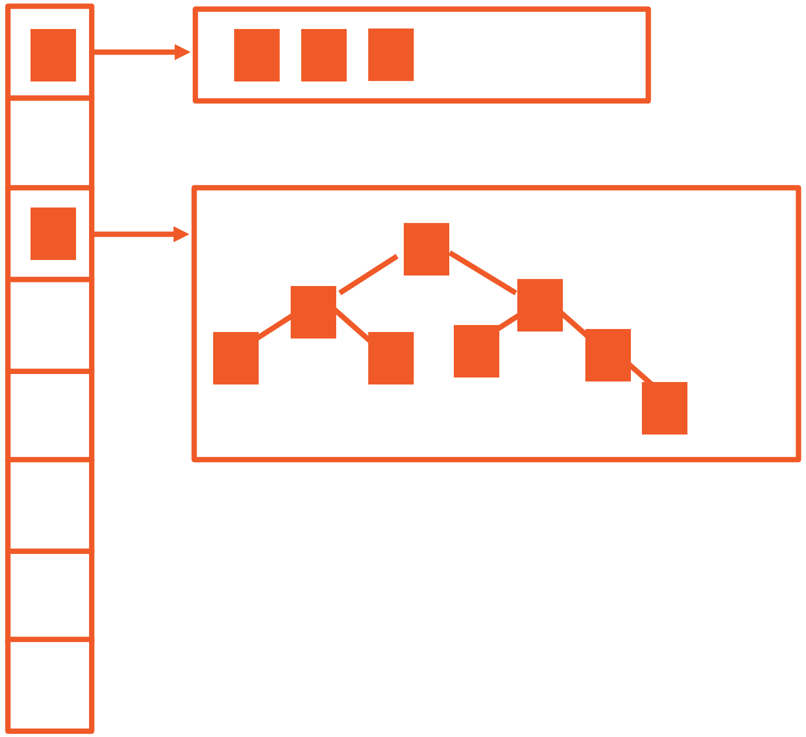
Use the hashcode generator built into the IDE

Use the Objects.hash() method

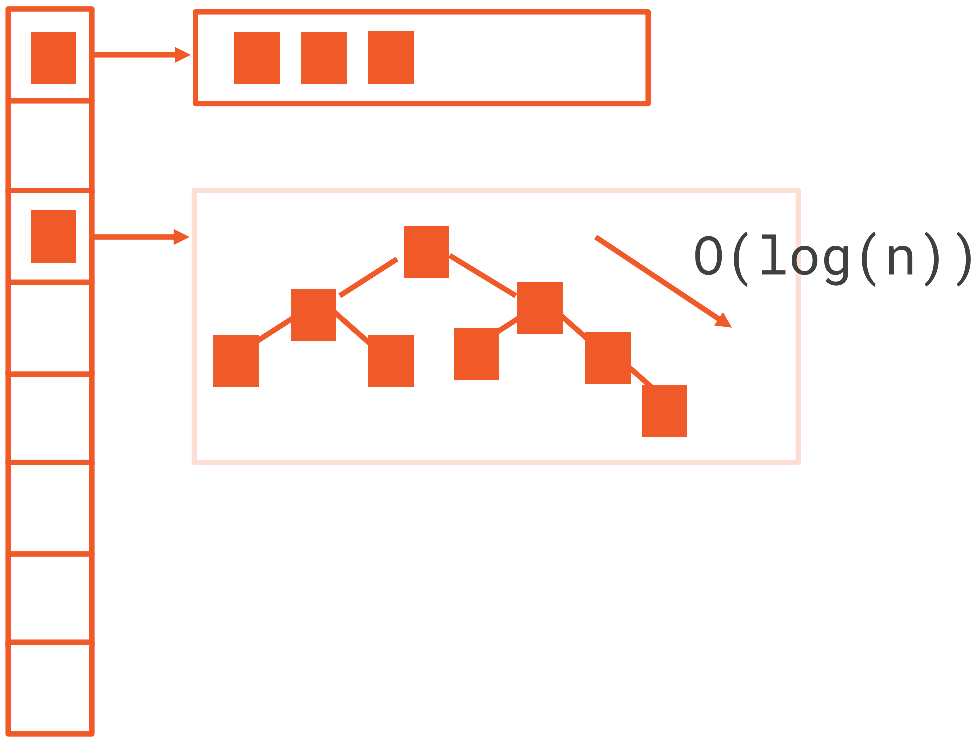
```
class Key {
  final int id; final int categoryId;
  Key(id, categoryId) {
    id = id;
    categoryId = categoryId
  }
  ...
  int hash = 0;
  public int hashCode() {
    if (hash == 0)
      hash = Objects.hash(id, categoryId);
    return hash;
  }
}
```

◂ Pre-computed hashcode

HashMaps



HashMaps



```
class Key implements Comparable {
  final int id; final int categoryId;
  Key (id, categoryId) {
    id = id;
    categoryId = categoryId
  }
  ...
  public in compareTo(Key other) {
    int order = this.getCategoryId() – other.getCategoryId();
    if (order != 0) return order;
    return this.getId() – other.getId();
  }
}
```

Implementing Comparable

User when:

- HashMap object is central to your application, frequently accessed, and contains 10s or 100s of thousands of records