CSE 12: Week 6 Discussion

5-7-21

Focus: PA6 HashMaps and File Systems

Reminders

- Start early! Start often!
- PA6 is a **open** assignment
  - Due Wednesday, May 12th 11:59 PM
- PA3 Resubmission due TODAY at 11:59 PM
- PA4 Resubmission due Friday, May 14th 11:59 PM
- PA5 Resubmission due Friday, May 21st 11:59 PM

PA6

Overview of PA6

- Part I: An Implementation of `DefaultMap`
  - Given the interface `DefaultMap`, implement `MyHashMap.java`
  - You may use Linear Probing or Separate Chaining for collision handling
  - 9 methods + 1 constructor in all (use the lecture and discussion slides to help you!)
- Part II: `FileSystem` Implementation
  - Implement the `FileData` class (this stores the information for a specific file)
    - Two methods: constructor and toString()
  - Implement the `FileSystem` class
    - Ten methods: some methods are very similar!
- Tester Files
- Part III: Gradescope Questions
- Style - must follow all indicated guidelines

Part 1: DefaultMap

Tips for Part I: MyHashMap.java

- Each entry (key value pair) is represented by the class HashMapEntry. Use this in your implementation
- For the hash function, you may use hashCode()
- Create your own helper methods, one suggestion would be for rehashing
- If you are implementing separate chaining, use the `buckets` instance variable. If you are using linear probing, use the `entries` instance variable. Do not use both!

Amortized Runtime Analysis

Lecture 17 will cover Amortized Runtime Analysis

Think of it as the **average case** when running the HashMap functions. We do not need to consider the expensive methods that are only called occasionally (i.e. expandCapacity) in our analysis.

In a HashMap that has a good hash function and an appropriate load factor, are we going to be closer to the worst or best case?

Only a few methods actually need the amortized analysis, for others there is no method being called that could be affecting the overall runtime.

Part 2: File System

FileData.java

This class represents the file that contains the information for name, directory, and last modified date.

Two Methods

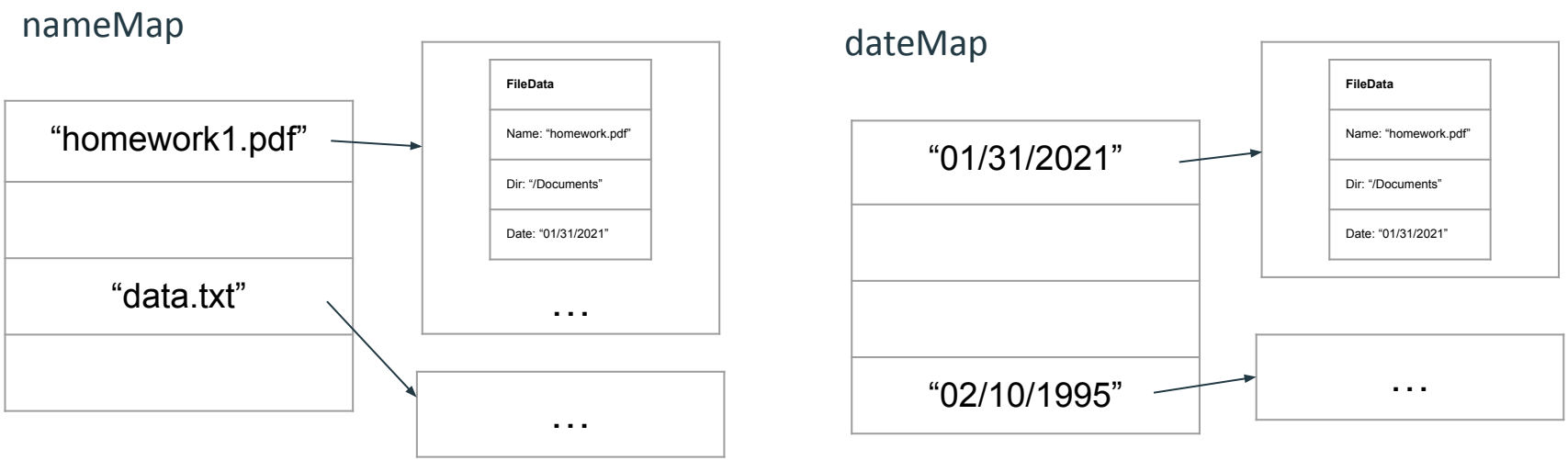
FileData() - constructor, initializes the instance variables

toString() - returns the string representation of the data in the FileData object

FileSystem.java

- FileSystem represents the entire structure of the system where you can add, remove, and search for files.
- One of the constructors populates nameMap and dateMap based on a given file, an example is provided in the starter code “input.txt”

MyHashMap<String, ArrayList<FileData>>



Example of add

```
add(“homework1.pdf”, /Documents”, “01/31/2021”);
```

FileData
Name: “homework.pdf”
Dir: “/Documents”
Date: “01/31/2021”

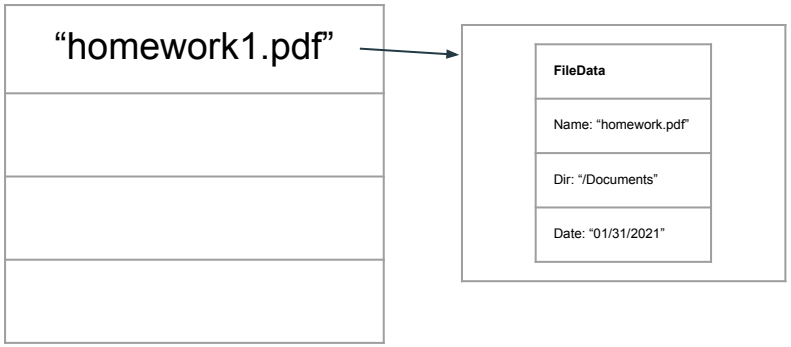
nameMap

dateMap

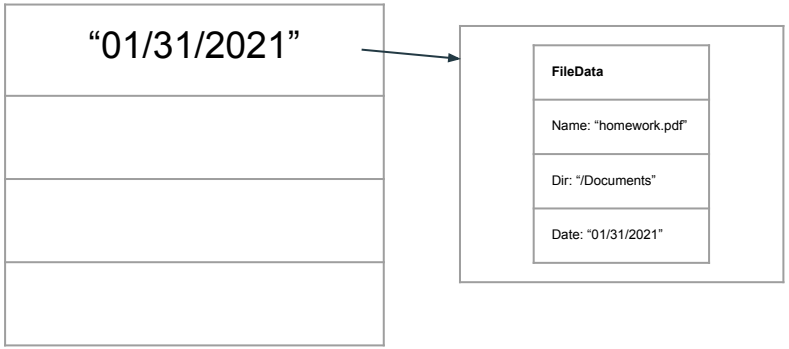
After adding

MyHashMap<String, ArrayList<FileData>>

nameMap



dateMap



Adding and Removing

- Adding
  - First check if the key exists
  - If the key exists, add FileData to the end of the ArrayList, which is the value to the key
  - If the key doesn’t exist, create an ArrayList first, and then add the FileData to the new ArrayList
- Removing
  - Remove fail if the key doesn’t exist
  - Remove the FileData by looking for it in the key’s corresponding ArrayList
  - If the ArrayList is empty after removing the FileData, remove the key from the map

Watch out!

Our File System allows duplicates, ONLY if they are in different directories.

If the file already exists with the same name, then return “false” when adding

Use this chart as a reference to help figure out when to deal with duplicates.

name	date	directory	ok?
same	same	diff	ok
same	diff	diff	ok
diff	same	diff	ok
diff	diff	diff	ok
same	same	same	no!
same	diff	same	no!
diff	same	same	ok
diff	diff	same	ok

Notice Methods are Similar

```
public ArrayList<FileData> findFilesByName(String name)

public ArrayList<FileData> findFilesByDate(String modifiedDate)
```

Same method, just different variable to work with!

Testing

How can we test the FileSystem?

- Create helper methods for comparing FileData objects and ArrayLists of FileData
- Use the constructor that takes an input file to quickly populate your FileSystem
- Utilize the toString() method in the FileData class to get a string representation
- The most crucial method is the add() method, focus on this method and then from there you can use it in your tests for other methods

Overall Tips

- Test your MyHashMap thoroughly before moving on to part 2
    - The autograder will use our implementation of MyHashMap to ensure you do not get penalized for both parts but you may have a hard time working on part 2 if your hash map has bugs
- Look at the method headers, specifically, what does the method return. This can help you keep track of what each method is supposed to do.
- Use Separate Chaining! This way you do not have to deal with tombstones and finding a good key value for place holding.

Maps

Maps

Maps are an Abstract Data Type (ADT)
Assign a **key** to each **value** we are trying to keep track of.

Key 1 ---> Some value a

Key 2 ---> Some value b

Key 3 ---> Some value c

etc...

Map<K,V> Interface

- Implemented in Java by AbstractMap, HashMap, TreeMap etc.
- Index for an entry is determined by a hash function that calculates an index using the key value (useful for quick lookup and insert)
- Contains methods such as get(Object key), put(K key, V value), size(), replace(K key, V value) etc.
- Keys need to be unique
- Existing data structures can be used to implement this - ArrayList!

Inserting Into Map

Key	Value
“a”	“apple”
“o”	“orange”

Note - this is a simplified view of map entries.
May not be in this exact order

Can we insert the following as a new entry?

```
put(“c”, “orange”)
```

A. Yes
B. No
C. I don’t know

Answer - A

Key	Value
“a”	“apple”
“o”	“orange”
“c”	“orange”

Note - this is a simplified view of map entries.
May not be in this exact order

Values across key-value pairs in maps do not need to be unique.

HashMaps - Separate Chaining

Example Hash Functions

```
int hash1(String s) {
  return s.length();
}
```

```
int hash2(String s) {
  int hash = 0;
  for(int i = 0; i < s.length(); i += 1) {
    hash += Character.codePointAt(s, i);
  }
  return hash;
}
```

```
public int hash3(String s) {
  int h = 0;
  for (int i = 0; i < s.length(); i++) {
    h = 31 * h + Character.codePointAt(s, i);
  }
  return h;
}
```

Given the example below, what does the HashMap look like after line: `set("red", 70)`?

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```

0	
1	
2	
3	

A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

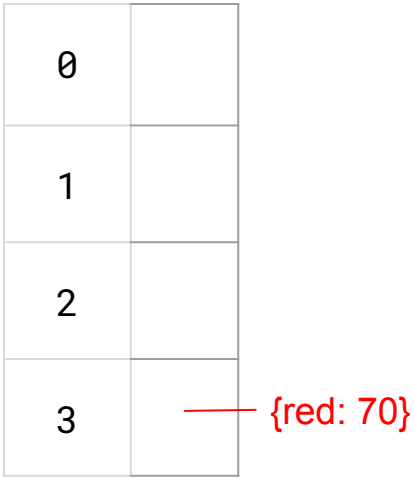
An Entry is a single {key: value} pair.

```
void set(key, value):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    update that Entry to contain value
  else:
    increment size
    bucket = buckets[index]
    add {key: value} to end of bucket


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    return the value of that entry
  else:
    return null/report an error
```

Given the example below, what does the HashMap look like after line: `set("red", 70)`?

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```



A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

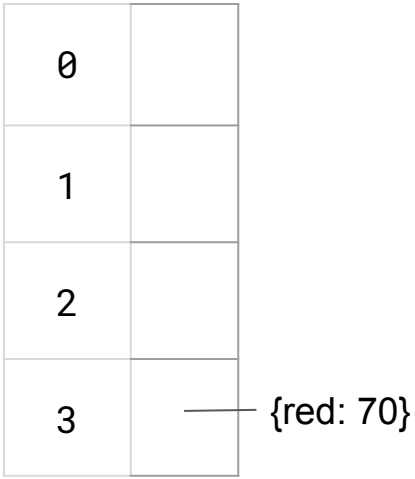
An Entry is a single {key: value} pair.

```
void set(key, value):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    update that Entry to contain value
  else:
    increment size
    bucket = buckets[index]
    add {key: value} to end of bucket


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    return the value of that entry
  else:
    return null/report an error
```

Given the example below, what does the HashMap look like after line: `set("blue", 90)`?

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```



A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

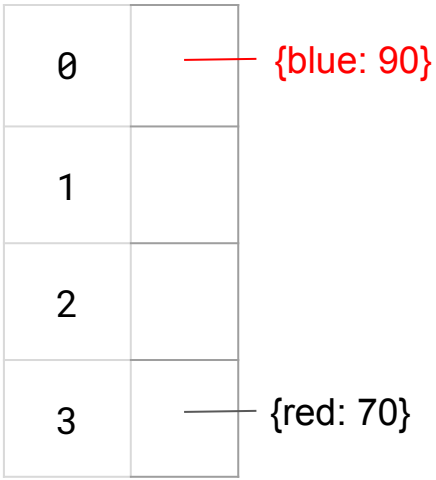
An Entry is a single {key: value} pair.

```
void set(key, value):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    update that Entry to contain value
  else:
    increment size
    bucket = buckets[index]
    add {key: value} to end of bucket


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    return the value of that entry
  else:
    return null/report an error
```

Given the example below, what does the HashMap look like after line: `set("blue", 90)`?

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```



A HashMap<Key, Value> using Separate Chaining has:
- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

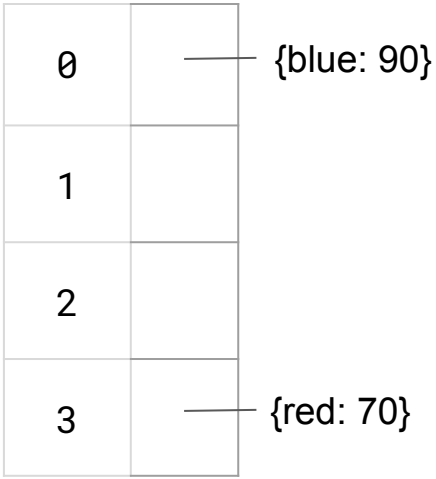
An Entry is a single {key: value} pair.

```
void set(key, value):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    update that Entry to contain value
  else:
    increment size
    bucket = buckets[index]
    add {key: value} to end of bucket


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    return the value of that entry
  else:
    return null/report an error
```

Given the example below, what does the HashMap look like after line: `set("pink", 100)`?

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```



A HashMap<Key, Value> using Separate Chaining has:
- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

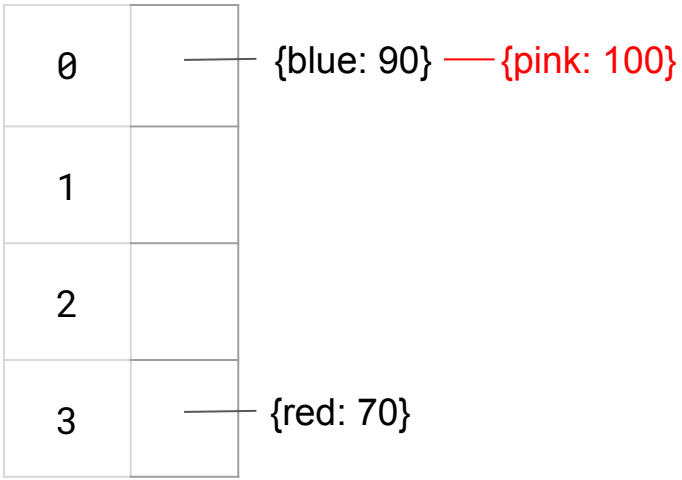
An Entry is a single {key: value} pair.

```
void set(key, value):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    update that Entry to contain value
  else:
    increment size
    bucket = buckets[index]
    add {key: value} to end of bucket


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    return the value of that entry
  else:
    return null/report an error
```

Given the example below, what does the HashMap look like after line: `set("pink", 100)`?

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```



A HashMap<Key, Value> using Separate Chaining has:
- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    update that Entry to contain value
  else:
    increment size
    bucket = buckets[index]
    add {key: value} to end of bucket


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    return the value of that entry
  else:
    return null/report an error
```

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```

Assuming the above example has been executed, how many elements are in bucket 0?

A: 0

B: 1

C: 2

D: 3

E: more than 3

A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    update that Entry to contain value
  else:
    increment size
    bucket = buckets[index]
    add {key: value} to end of bucket


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    return the value of that entry
  else:
    return null/report an error
```

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```

Assuming the above example has been executed, how many elements are in bucket 0?

A: 0

B: 1

C: 2

D: 3

E: more than 3

A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    update that Entry to contain value
  else:
    increment size
    bucket = buckets[index]
    add {key: value} to end of bucket


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    return the value of that entry
  else:
    return null/report an error
```

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```

Assuming the above example has been executed, how many elements are in bucket 2?

A: 0

B: 1

C: 2

D: 3

E: more than 3

A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    update that Entry to contain value
  else:
    increment size
    bucket = buckets[index]
    add {key: value} to end of bucket


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    return the value of that entry
  else:
    return null/report an error
```

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```

Assuming the above example has been executed, how many elements are in bucket 2?

A: 0

B: 1

C: 2

D: 3

E: more than 3

A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    update that Entry to contain value
  else:
    increment size
    bucket = buckets[index]
    add {key: value} to end of bucket


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    return the value of that entry
  else:
    return null/report an error
```

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```

Assuming the above example has been executed, how many entries are checked for `get("purplish")`?

A: 0

B: 1

C: 2

D: 3

E: more than 3

A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    update that Entry to contain value
  else:
    increment size
    bucket = buckets[index]
    add {key: value} to end of bucket


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    return the value of that entry
  else:
    return null/report an error
```

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```

Assuming the above example has been executed, how many entries are checked for `get("purplish")`?

A: 0

B: 1

C: 2

D: 3

E: more than 3

A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

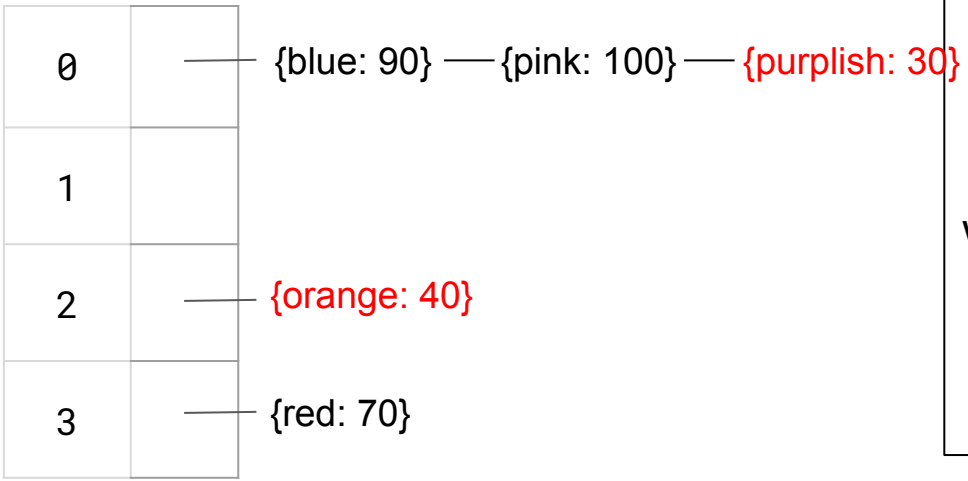
An Entry is a single {key: value} pair.

```
void set(key, value):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    update that Entry to contain value
  else:
    increment size
    bucket = buckets[index]
    add {key: value} to end of bucket


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    return the value of that entry
  else:
    return null/report an error
```

Please complete the below HashMap, assuming the entire example code has executed.

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```



A HashMap<Key, Value> using Separate Chaining has:
- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    update that Entry to contain value
  else:
    increment size
    bucket = buckets[index]
    add {key: value} to end of bucket


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  if this.buckets[index] contains an Entry with key:
    return the value of that entry
  else:
    return null/report an error
```

What is the load factor?

A: # elements * # buckets / 2

B: # buckets * # elements

C: # buckets / # elements

D: # elements / # buckets

What is the load factor?

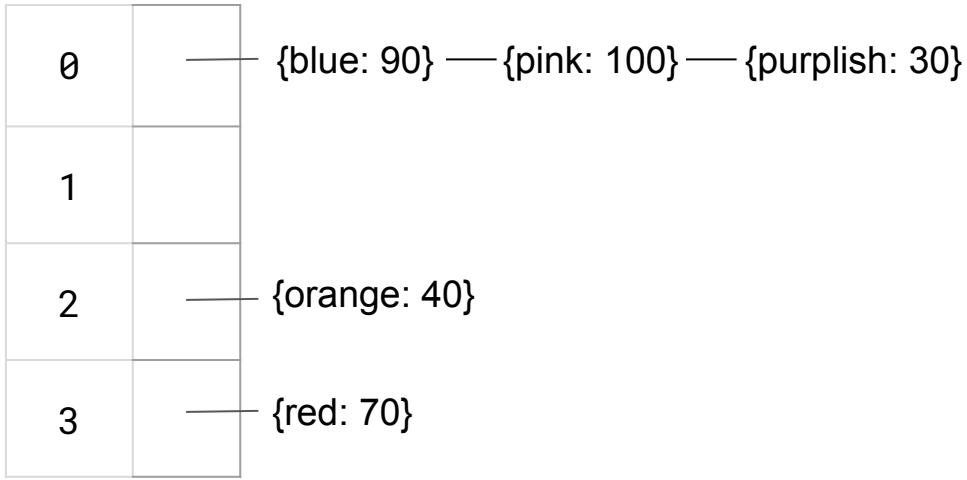
A: # elements * # buckets / 2

B: # buckets * # elements

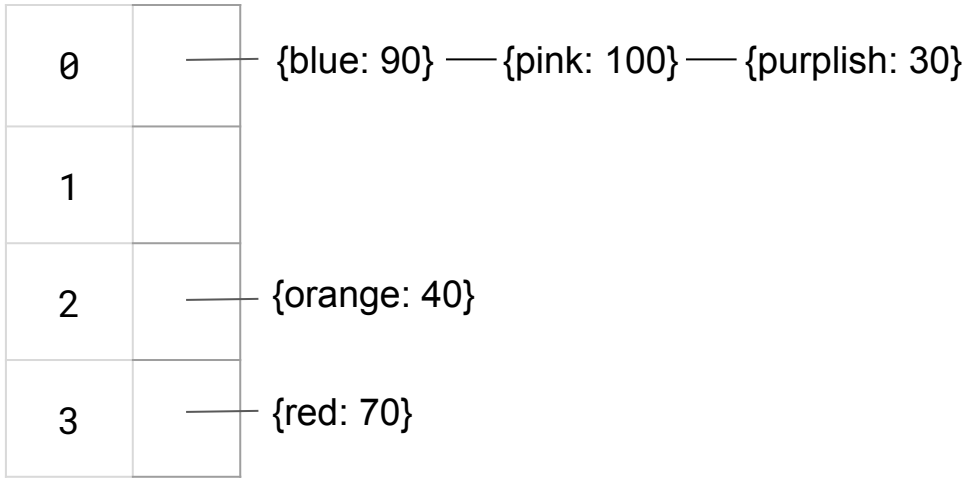
C: # buckets / # elements

D: # elements / # buckets

What is the load factor of the HashMap below?



What is the load factor of the HashMap below?

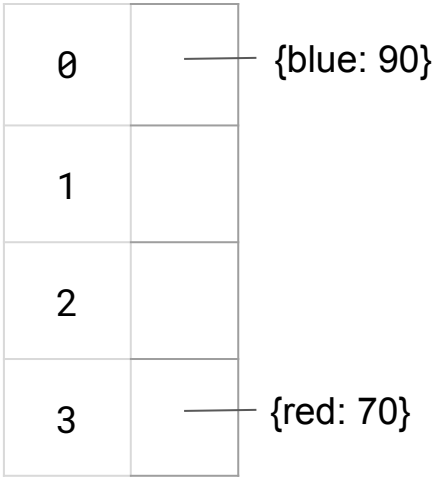


Load Factor: 5/4

What is the load factor after the line `set("pink", 100)` is executed?

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```



Load Factor: 3/4

A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

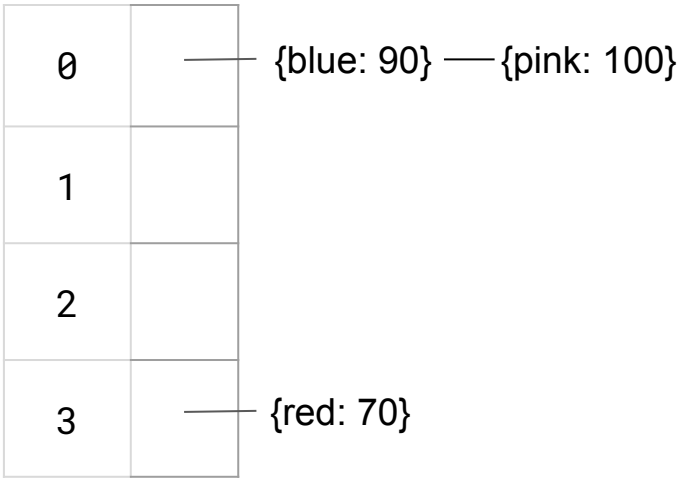
An Entry is a single {key: value} pair.

```
void set(key, value):

  if LoadFactor > 0.5: expandCapacity()

  ... as before ...


void expandCapacity():
  newBuckets = new List[this.buckets.length * 2];
  oldBuckets = this.buckets
  this.buckets = newBuckets
  this.size = 0
  for each list of entries in oldBuckets:
    for each {k: v} in the list:
      this.set(k, v)
```

What is <u>different</u> when the line `set("orange", 40)` is executed?

<u>Example:</u>
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```



A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

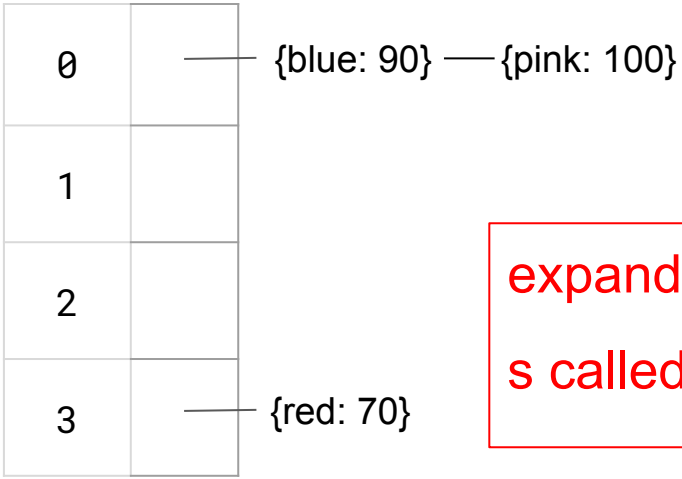
An Entry is a single {key: value} pair.

```
void set(key, value):

  if LoadFactor > 0.5: expandCapacity()

  ... as before ...


void expandCapacity():
  newBuckets = new List[this.buckets.length * 2];
  oldBuckets = this.buckets
  this.buckets = newBuckets
  this.size = 0
  for each list of entries in oldBuckets:
    for each {k: v} in the list:
      this.set(k, v)
```

What is <u>different</u> when the line `set("orange", 40)` is executed?

<u>Example:</u>
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```



expandCapacityis called!

A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):

  if LoadFactor > 0.5: expandCapacity()

  ... as before ...


void expandCapacity():
  newBuckets = new List[this.buckets.length * 2];
  oldBuckets = this.buckets
  this.buckets = newBuckets
  this.size = 0
  for each list of entries in oldBuckets:
    for each {k: v} in the list:
      this.set(k, v)
```

What does the HashMap look like after expandCapacity is called in set("orange", 40)?

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```

0	
1	
2	
3	

4	
5	
6	
7	

A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

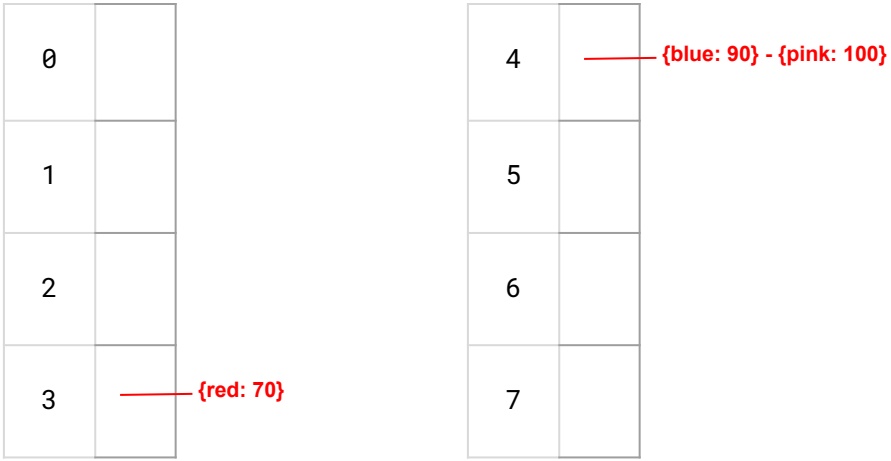
An Entry is a single {key: value} pair.

```
void set(key, value):

  if LoadFactor > 0.5: expandCapacity()

  ... as before ...


void expandCapacity():
  newBuckets = new List[this.buckets.length * 2];
  oldBuckets = this.buckets
  this.buckets = newBuckets
  this.size = 0
  for each list of entries in oldBuckets:
    for each {k: v} in the list:
      this.set(k, v)
```

What does the HashMap look like after expandCapacity is called in set("orange", 40)?

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```



A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

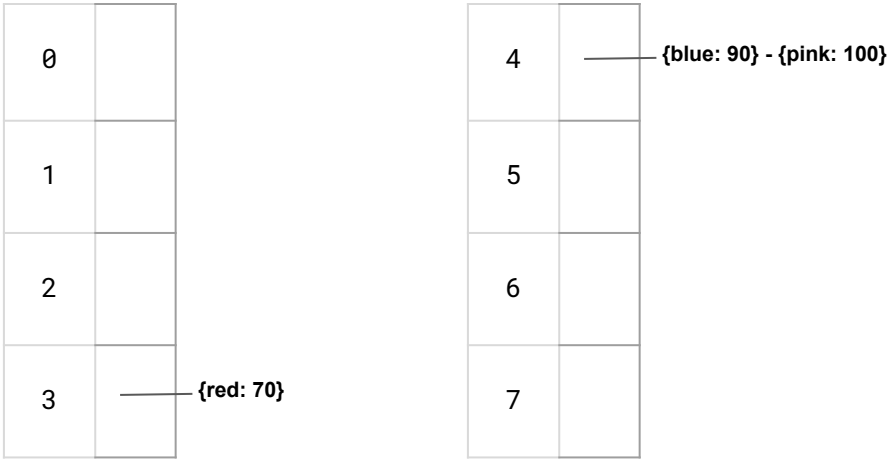
An Entry is a single {key: value} pair.

```
void set(key, value):

  if LoadFactor > 0.5: expandCapacity()

  ... as before ...


void expandCapacity():
  newBuckets = new List[this.buckets.length * 2];
  oldBuckets = this.buckets
  this.buckets = newBuckets
  this.size = 0
  for each list of entries in oldBuckets:
    for each {k: v} in the list:
      this.set(k, v)
```

What does the HashMap look like after set("orange", 40) is called?

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```



A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

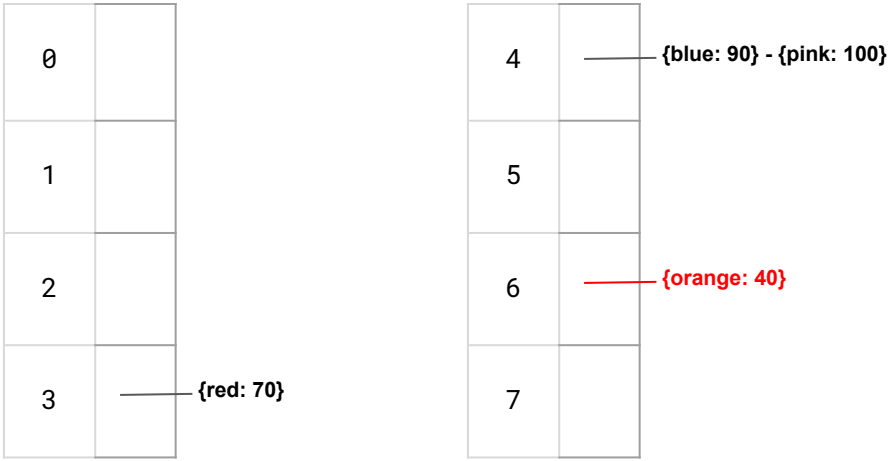
An Entry is a single {key: value} pair.

```
void set(key, value):

  if LoadFactor > 0.5: expandCapacity()

  ... as before ...


void expandCapacity():
  newBuckets = new List[this.buckets.length * 2];
  oldBuckets = this.buckets
  this.buckets = newBuckets
  this.size = 0
  for each list of entries in oldBuckets:
    for each {k: v} in the list:
      this.set(k, v)
```

What does the HashMap look like after set("orange", 40) is called?

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```



A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

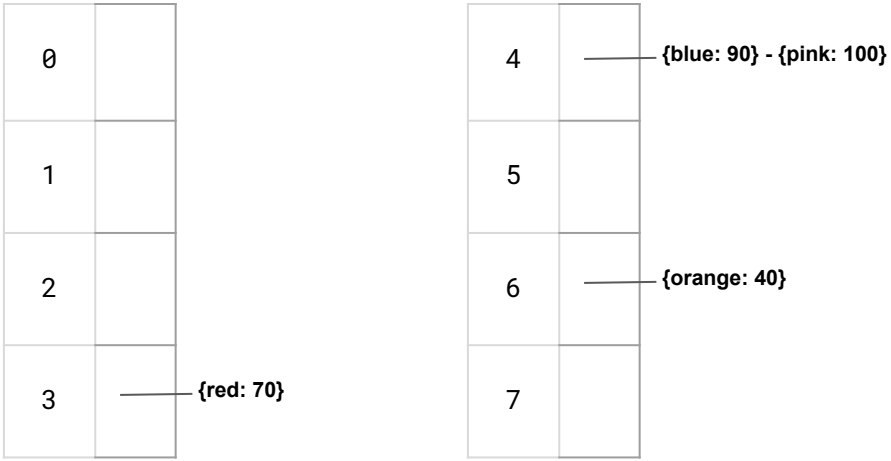
An Entry is a single {key: value} pair.

```
void set(key, value):

  if LoadFactor > 0.5: expandCapacity()

  ... as before ...


void expandCapacity():
  newBuckets = new List[this.buckets.length * 2];
  oldBuckets = this.buckets
  this.buckets = newBuckets
  this.size = 0
  for each list of entries in oldBuckets:
    for each {k: v} in the list:
      this.set(k, v)
```

What does the HashMap look like after set("purplish", 40) is called?

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```



A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

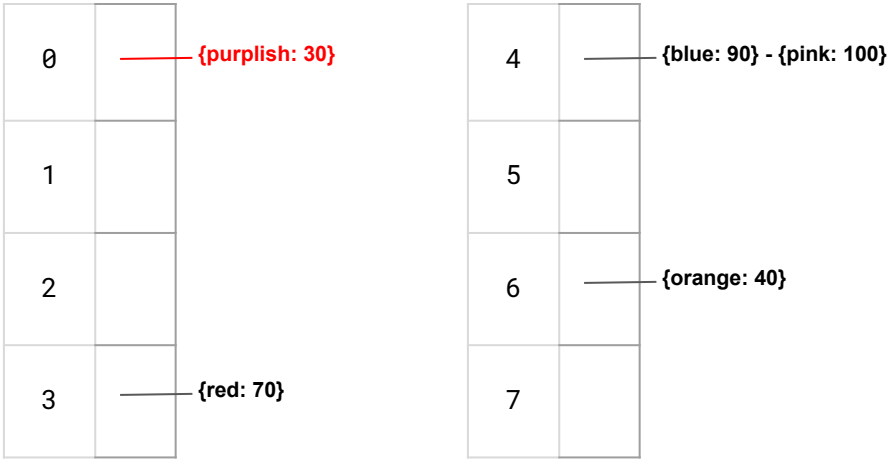
An Entry is a single {key: value} pair.

```
void set(key, value):

  if LoadFactor > 0.5: expandCapacity()

  ... as before ...


void expandCapacity():
  newBuckets = new List[this.buckets.length * 2];
  oldBuckets = this.buckets
  this.buckets = newBuckets
  this.size = 0
  for each list of entries in oldBuckets:
    for each {k: v} in the list:
      this.set(k, v)
```

What does the HashMap look like after set("purplish", 30) is called?

Example:
Start buckets array with size 4
Use string length as the hash function **(In general this is a BAD hash function)**

```
set("red", 70)
set("blue", 90)
set("pink", 100)
set("orange", 40)
set("purplish", 30)
```



A HashMap<Key, Value> using Separate Chaining has:

- size: an int
- buckets: an array of lists of Entries
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):

  if LoadFactor > 0.5: expandCapacity()

  ... as before ...


void expandCapacity():
  newBuckets = new List[this.buckets.length * 2];
  oldBuckets = this.buckets
  this.buckets = newBuckets
  this.size = 0
  for each list of entries in oldBuckets:
    for each {k: v} in the list:
      this.set(k, v)
```

HashMaps - Linear Probing

What does the HashMap below look like after the example code has executed?

```
Example:
Start buckets array with size 4, containing null
ASCII code as hash function ("a" = 97)

set("b", 70) # note 98 % 4 is 2
set("f", 90)
set("f", 100)
```

0	
1	
2	
3	

A HashMap<Key, Value> using Linear Probing has:

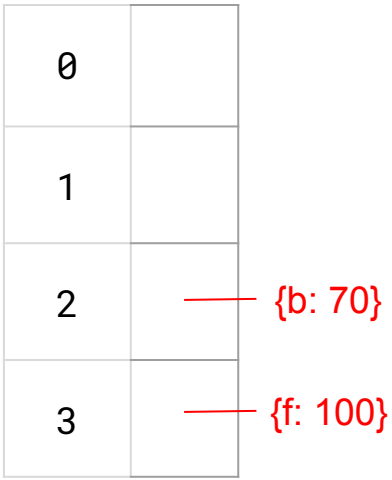
- size: an int
- buckets: an array of Entries (not of lists of Entries!)
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):
  if loadFactor > 0.67: expandCapacity()
  hashed = hash(key)
  index = hashed % array length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key):
      b.value = value
      return
    index += 1

  // key not in table, add it at first index containing null
  this.buckets[index] = {key: value}


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key): return b.value
    index += 1

  // haven't found the key
  return null/throw exception


void expandCapacity():
  newEntries = new Entry[this.buckets.length * 2];
  oldEntries = this.buckets
  this.buckets = newEntries
  this.size = 0
  for each entry {k:v} in oldEntries:
    this.set(k, v)
```

What does the HashMap below look like after the example code has executed?

```
Example:
Start buckets array with size 4, containing null
ASCII code as hash function ("a" = 97)

set("b", 70) # note 98 % 4 is 2
set("f", 90)
set("f", 100)
```



A HashMap<Key, Value> using Linear Probing has:

- size: an int
- buckets: an array of Entries (not of lists of Entries!)
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):
  if loadFactor > 0.67: expandCapacity()
  hashed = hash(key)
  index = hashed % array length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key):
      b.value = value
      return
    index += 1

  // key not in table, add it at first index containing null
  this.buckets[index] = {key: value}


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key): return b.value
    index += 1

  // haven't found the key
  return null/throw exception


void expandCapacity():
  newEntries = new Entry[this.buckets.length * 2];
  oldEntries = this.buckets
  this.buckets = newEntries
  this.size = 0
  for each entry {k:v} in oldEntries:
    this.set(k, v)
```

```
Example:
Start buckets array with size 4, containing null
ASCII code as hash function ("a" = 97)

set("b", 70) # note 98 % 4 is 2
set("f", 90)
set("f", 100)
```

Assuming the above example has been executed,

how many entries are checked when doing

`set("f", 100)`?

A: 0

B: 1

C: 2

D: 3

E: more than 3

A HashMap<Key, Value> using Linear Probing has:

- size: an int
- buckets: an array of Entries (not of lists of Entries!)
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):
  if loadFactor > 0.67: expandCapacity()
  hashed = hash(key)
  index = hashed % array length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key):
      b.value = value
      return
    index += 1

  // key not in table, add it at first index containing null
  this.buckets[index] = {key: value}


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key): return b.value
    index += 1

  // haven't found the key
  return null/throw exception


void expandCapacity():
  newEntries = new Entry[this.buckets.length * 2];
  oldEntries = this.buckets
  this.buckets = newEntries
  this.size = 0
  for each entry {k:v} in oldEntries:
    this.set(k, v)
```

```
Example:
Start buckets array with size 4, containing null
ASCII code as hash function ("a" = 97)

set("b", 70) # note 98 % 4 is 2
set("f", 90)
set("f", 100)
```

Assuming the above example has been executed,

how many entries are checked when doing

`set("f", 100)`?

A: 0

B: 1

C: 2

D: 3

E: more than 3

A HashMap<Key, Value> using Linear Probing has:

- size: an int
- buckets: an array of Entries (not of lists of Entries!)
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):
  if loadFactor > 0.67: expandCapacity()
  hashed = hash(key)
  index = hashed % array length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key):
      b.value = value
      return
    index += 1

  // key not in table, add it at first index containing null
  this.buckets[index] = {key: value}


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key): return b.value
    index += 1

  // haven't found the key
  return null/throw exception


void expandCapacity():
  newEntries = new Entry[this.buckets.length * 2];
  oldEntries = this.buckets
  this.buckets = newEntries
  this.size = 0
  for each entry {k:v} in oldEntries:
    this.set(k, v)
```

```
Example:
Start buckets array with size 4, containing null
ASCII code as hash function ("a" = 97)

set("b", 70) # note 98 % 4 is 2
set("f", 90)
set("f", 100)
```

Assuming the above example has been executed,

and an additional line is added below: set(“c”, 40),

Which bucket is “c” stored in?

A: 0

B: 1

C: 2

D: 3

E: it causes an error

A HashMap<Key, Value> using Linear Probing has:

- size: an int
- buckets: an array of Entries (not of lists of Entries!)
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):
  if loadFactor > 0.67: expandCapacity()
  hashed = hash(key)
  index = hashed % array length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key):
      b.value = value
      return
    index += 1

  // key not in table, add it at first index containing null
  this.buckets[index] = {key: value}


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key): return b.value
    index += 1

  // haven't found the key
  return null/throw exception


void expandCapacity():
  newEntries = new Entry[this.buckets.length * 2];
  oldEntries = this.buckets
  this.buckets = newEntries
  this.size = 0
  for each entry {k:v} in oldEntries:
    this.set(k, v)
```

```
Example:
Start buckets array with size 4, containing null
ASCII code as hash function ("a" = 97)

set("b", 70) # note 98 % 4 is 2
set("f", 90)
set("f", 100)
```

Assuming the above example has been executed,

and an additional line is added below: set(“c”, 40),

Which bucket is “c” stored in?

A: 0

B: 1

C: 2

D: 3

E: it causes an error (ArrayIndexOutOfBounds)

A HashMap<Key, Value> using Linear Probing has:

- size: an int
- buckets: an array of Entries (not of lists of Entries!)
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):
  if loadFactor > 0.67: expandCapacity()
  hashed = hash(key)
  index = hashed % array length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key):
      b.value = value
      return
    index += 1

  // key not in table, add it at first index containing null
  this.buckets[index] = {key: value}


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key): return b.value
    index += 1

  // haven't found the key
  return null/throw exception


void expandCapacity():
  newEntries = new Entry[this.buckets.length * 2];
  oldEntries = this.buckets
  this.buckets = newEntries
  this.size = 0
  for each entry {k:v} in oldEntries:
    this.set(k, v)
```

How can we fix the ArrayOutOfBounds issue?

A HashMap<Key, Value> using Linear Probing has:

- size: an int
- entries: an array of Entries (not of lists of Entries!)
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):
  if loadFactor > 0.67: expandCapacity()
  hashed = hash(key)
  index = hashed % array length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key):
      b.value = value
      return
    index += 1

  // key not in table, add it at first index containing null
  this.buckets[index] = {key: value}


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key): return b.value
    index += 1

  // haven't found the key
  return null/throw exception


void expandCapacity():
  newEntries = new Entry[this.buckets.length * 2];
  oldEntries = this.buckets
  this.buckets = newEntries
  this.size = 0
  for each entry {k:v} in oldEntries:
    this.set(k, v)
```

How can we fix the ArrayOutOfBounds issue?

A HashMap<Key, Value> using Linear Probing has:

- size: an int
- entries: an array of Entries (not of lists of Entries!)
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

When you get to the end of the array just fall off the end, wrap around to the beginning, and starting searching again at 0.

We would no longer have ArrayIndexOutOfBounds issue!

Loadfactor - never update size!!! Where should we increment size?

Assume load factor is size/currentlength (helper method)

```
void set(key, value):
  if loadFactor > 0.67: expandCapacity()
  hashed = hash(key)
  index = hashed % array length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key):
      b.value = value
      return
    index += 1
    index = index % buckets.length
  // key not in table, add it at first index containing null
  this.buckets[index] = {key: value}


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key): return b.value
    index += 1
    index = index % buckets.length
  // haven't found the key
  return null/throw exception


void expandCapacity():
  newEntries = new Entry[this.buckets.length * 2];
  oldEntries = this.buckets
  this.buckets = newEntries
  this.size = 0
  for each entry {k:v} in oldEntries:
    this.set(k, v)
```

Are there any other issues that need to be fixed?

A HashMap<Key, Value> using Linear Probing has:

- size: an int
- entries: an array of Entries (not of lists of Entries!)
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):
  if loadFactor > 0.67: expandCapacity()
  hashed = hash(key)
  index = hashed % array length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key):
      b.value = value
      return
    index += 1
    index = index % buckets.length
  // key not in table, add it at first index containing null
  this.buckets[index] = {key: value}


Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key): return b.value
    index += 1
    index = index % buckets.length
  // haven't found the key
  return null/throw exception


void expandCapacity():
  newEntries = new Entry[this.buckets.length * 2];
  oldEntries = this.buckets
  this.buckets = newEntries
  this.size = 0
  for each entry {k:v} in oldEntries:
    this.set(k, v)
```

Are there any other issues that need to be fixed?

A HashMap<Key, Value> using Linear Probing has:

- `size`: an `int`
- `entries`: an array of Entries (not of lists of Entries!)
- `hash`: a hash function for the Key type

An Entry is a single {key: value} pair.

YES! `size` is never being updated!

(Let's assume `loadFactor` is actually a helper method that returns the current size divided by the current length.)

```
void set(key, value):
  if loadFactor() > 0.67: expandCapacity()
  hashed = hash(key)
  index = hashed % array length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key):
      b.value = value
      return
    index += 1
    index = index % buckets.length
  // key not in table, add it at first index containing null
  this.buckets[index] = {key: value}
  size += 1

Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key): return b.value
    index += 1
    index = index % buckets.length
  // haven't found the key
  return null/throw exception


void expandCapacity():
  newEntries = new Entry[this.buckets.length * 2];
  oldEntries = this.buckets
  this.buckets = newEntries
  this.size = 0
  for each entry {k:v} in oldEntries:
    this.set(k, v)
```

What happens if we set `loadFactor` to be 1 instead of 0.67?

A HashMap<Key, Value> using Linear Probing has:

- size: an int
- buckets: an array of Entries (not of lists of Entries!)
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

```
void set(key, value):
  if loadFactor() > 0.67: expandCapacity()
  hashed = hash(key)
  index = hashed % array length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key):
      b.value = value
      return
    index += 1
    index = index % buckets.length
  // key not in table, add it at first index containing null
  this.buckets[index] = {key: value}
  size += 1

Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key): return b.value
    index += 1
    index = index % buckets.length
  // haven't found the key
  return null/throw exception


void expandCapacity():
  newEntries = new Entry[this.buckets.length * 2];
  oldEntries = this.buckets
  this.buckets = newEntries
  this.size = 0
  for each entry {k:v} in oldEntries:
    this.set(k, v)
```

What happens if we set `loadFactor` to be 1 instead of 0.67?

A HashMap<Key, Value> using Linear Probing has:

- size: an int
- buckets: an array of Entries (not of lists of Entries!)
- hash: a hash function for the Key type

An Entry is a single {key: value} pair.

INFINITE LOOP!

There would be an infinite loop once the array is full. If the array is full of entries the method will search until it finds a bucket equal to null and there is no null to find.

```
void set(key, value):
  if loadFactor() > 0.67: expandCapacity()
  hashed = hash(key)
  index = hashed % array length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key):
      b.value = value
      return
    index += 1
    index = index % buckets.length
  // key not in table, add it at first index containing null
  this.buckets[index] = {key: value}
  size += 1

Value get(key):
  hashed = hash(key)
  index = hashed % this.buckets.length
  while this.buckets[index] != null:
    b = this.buckets[index]
    if b.key.equals(key): return b.value
    index += 1
    index = index % buckets.length
  // haven't found the key
  return null/throw exception


void expandCapacity():
  newEntries = new Entry[this.buckets.length * 2];
  oldEntries = this.buckets
  this.buckets = newEntries
  this.size = 0
  for each entry {k:v} in oldEntries:
    this.set(k, v)
```