Exam2 Review Questions

Partition

Given the following specification and description for **`partition()`**

```
int partition(String[] strs)
```

After a pivot index is selected (for example, pick the last element), then the array is partitioned in such a way that the input array has its elements moved around so that element values less than or equal to the pivot index’s value are to the left of the pivot index and element values greater than or equal to the pivot index’s value are to the right. The comparison of string values can be done by **`compareTo()`**.

Partition

Given the input array {“e”, “d”, “c”, “b”, “a”}, assume that after a call to **`partition()`**, we get {“a”, “b”, “e”, “c”, “d”}. List all possible indices that could have been returned from **`partition()`** for this to be a valid partition result.

Answer

0 and 1

Let the pivot be green

{“a”, “b”, “e”, “c”, “d”}   {“a”, “b”, “e”, “c”, “d”}

We can see that all red elements are smaller than pivot, and all blue elements are greater than pivot

Runtime (big-O and big-Ω)

Let f(n) = n^2 + n + log(n), and g(n) = n*log(n) + n^2. Select all statements that are true.

- f is O(g)
- f is Ω(g)

Answer

f is O(g), f is Ω(g)

After we reduce both f(n) and g(n) to their fastest growing term, we can see that all that’s left is n^2 for both of them, and n^2 is both the upper bound and the lower bound for itself.

Runtime (big-O and big-Ω)

Let f(n) = 1/(n^2) + 5, and g(n) = 1/n. Select all statements that are true.

- f is O(g)
- f is Ω(g)

Answer

f is Ω(g)

g(n) is not an upper bound of f(n) because the constant term 5 grows faster than 1/n. g(n) is a lower bound of f(n) because as n grows to infinity, g(n) approaches zero but f(n) approaches 5.

Runtime (tight-bound)

What's the big-θ bound of the following program? Assume that **`doSomeWork()`** does a constant amount of work.

```
for(int i = 1; i < n; i = i * 2) {

  doSomeWork();

}
```

Answer

$\theta(\log(n))$

If n = 8, we will see i = 1, 2, 4, 8. If n = 32, we will see i = 1, 2, 4, 8, 16, 32. Then we can see that the number of times the for loop runs is related to log(n)

Runtime (tight-bound)

What's the big-θ bound of the following program? Assume that **`doSomeWork()`** does a constant amount of work.

```
for(int i = 0; i <= n * n; i = i + 2) {
  for(int j = n; j > 0; j -= 1) {
    doSomeWork();
  }
}
```

Answer

θ(n^3)

The outer for loop runs θ(n^2) times because i is bounded by n^2 and increases by a constant amount in every iteration.

The inner for loop runs θ(n) times because j decrement by 1 in every iteration from n until 0.

Because those two loops are nested, we get θ(n^3)

Sorting

Review: Name this Sorting Algorithm!

```
static int[] combine(int[] p1, int[] p2) {...}

static int[] sort(int[] arr) {
  int len = arr.length
  if(len <= 1) { return arr; }
  else {
    int[] p1 = Arrays.copyOfRange(arr, 0, len / 2);
    int[] p2= Arrays.copyOfRange(arr, len / 2, len);
    int[] sortedPart1 = sort(p1);
    int[] sortedPart2 = sort(p2);
    int[] sorted = combine(sortedPart1, sortedPart2);
    return sorted;
  }
}
```

A: Selection

B: Insertion

C: Merge

D: Quick

Review: Name this Sorting Algorithm!

```
static int[] combine(int[] p1, int[] p2) {...}

static int[] mergeSort(int[] arr) {
  int len = arr.length
  if(len <= 1) { return arr; }
  else {
    int[] p1 = Arrays.copyOfRange(arr, 0, len / 2);
    int[] p2= Arrays.copyOfRange(arr, len / 2, len);
    int[] sortedPart1 = mergeSort(p1);
    int[] sortedPart2 = mergeSort(p2);
    int[] sorted = combine(sortedPart1, sortedPart2);
    return sorted;
  }
}
```

A: Selection

B: Insertion

C: Merge

D: Quick

Review: Name this Sorting Algorithm!

```
static void sort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
      else { break; }
    }
  }
}
```

A: Selection

B: Insertion

C: Merge

D: Quick

Review: Name this Sorting Algorithm!

```
static void insertionSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    for(int j = i; j > 0; j -= 1) {
      if(arr[j] < arr[j-1]) {
        int temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
      else { break; }
    }
  }
}
```

A: Selection

B: Insertion

C: Merge

D: Quick

Review: Name this Sorting Algorithm!

```
static int partition(String[] array, int l, int h) {...}

static void sort2(String[] array, int low, int high) {
  if(high - low <= 1) { return; }
  int splitAt = partition(array, low, high);
  sort2(array, low, splitAt);
  sort2(array, splitAt + 1, high);
}

public static void sort1(String[] array) {
  sort2(array, 0, array.length);
}
```

A: Selection

B: Insertion

C: Merge

D: Quick

Review: Name this Sorting Algorithm!

```
static int partition(String[] array, int l, int h) {...}

static void qsort(String[] array, int low, int high) {
  if(high - low <= 1) { return; }
  int splitAt = partition(array, low, high);
  qsort(array, low, splitAt);
  qsort(array, splitAt + 1, high);
}

public static void sort(String[] array) {
  qsort(array, 0, array.length);
}
```

A: Selection

B: Insertion

C: Merge

D: Quick

Review: Name this Sorting Algorithm!

```
static void sort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    int minIndex = i;
    for(int j = i; j < arr.length; j += 1) {
      if(arr[minIndex] > arr[j]) { minIndex = j; }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
}
```

A: Selection

B: Insertion

C: Merge

D: Quick

Review: Name this Sorting Algorithm!

```
static void selectionSort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {
    int minIndex = i;
    for(int j = i; j < arr.length; j += 1) {
      if(arr[minIndex] > arr[j]) { minIndex = j; }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
}
```

A: Selection

B: Insertion

C: Merge

D: Quick

Fill out the table

	Insertion	**Selection**	**Merge**	**Quick**
Best case time				
Worst case time				

Key operations	`swap(a, j, j-1)`
	(until in the right place)
		`swap(a, i, indexOfMin)`
		(after finding minimum value)
			`l = copy(a, 0, len/2)`
			`r = copy(a, len/2, len)`
			`ls = sort(l)`
			`rs = sort(r)`
			`merge(ls, rs)`
				`p = partition(a, l, h)`
				`sort(a, l, p)`
				`sort(a, p + 1, h)`

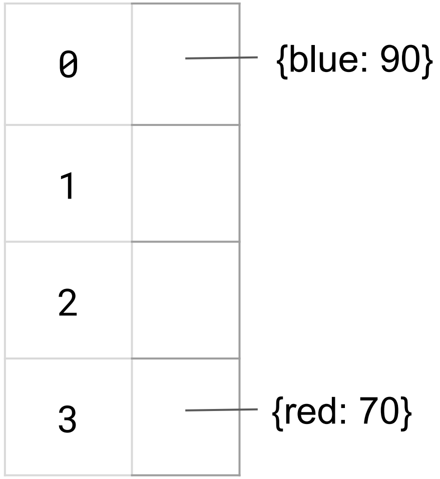
	Insertion	**Selection**	**Merge**	**Quick**
Best case time	O(n)	O(n^2)	O(n*logn)	O(n*logn)
Worst case time	O(n^2)	O(n^2)	O(n*logn)	O(n^2)

Key operations	`swap(a, j, j-1)`
	(until in the right place)
		`swap(a, i, indexOfMin)`
		(after finding minimum value)
			`l = copy(a, 0, len/2)`
			`r = copy(a, len/2, len)`
			`ls = sort(l)`
			`rs = sort(r)`
			`merge(ls, rs)`
				`p = partition(a, l, h)`
				`sort(a, l, p)`
				`sort(a, p + 1, h)`

HashMap (Separate Chaining)

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

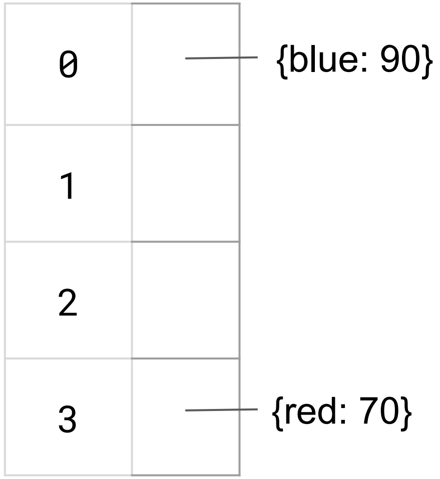
```
Example:
Start buckets array with size 4
Use string length as the hash function (In general
this is a BAD hash function)

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



```
void set(key, value):
  if LoadFactor > 0.5: expandCapacity()
  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

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 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)
```

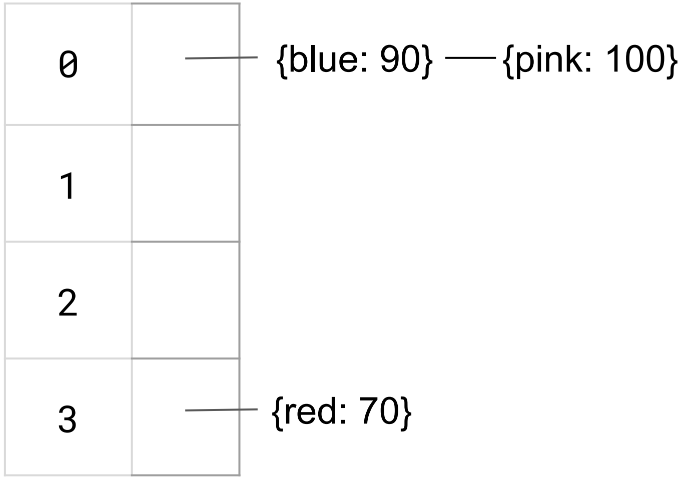


Load Factor: 3/4

```
void set(key, value):
  if LoadFactor > 0.5: expandCapacity()
  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

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?

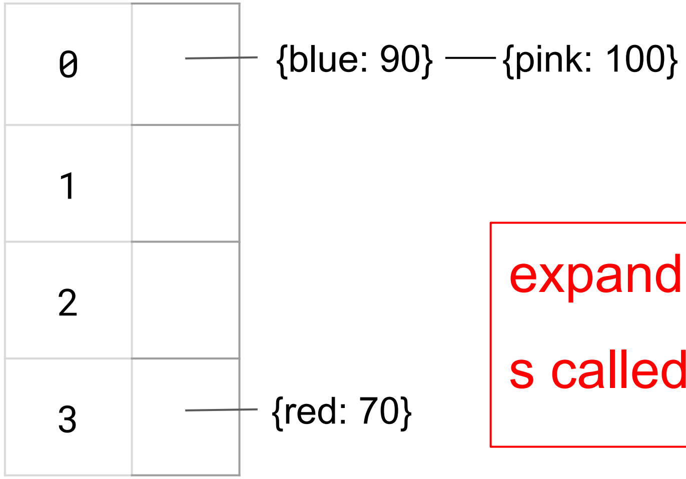
```
Example:
Start buckets array with size 4
Use string length as the hash function (In general
this is a BAD hash function)

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



```
void set(key, value):

  if LoadFactor > 0.5: expandCapacity()

  ... as before ...

Value get(key):

  ... 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?

```
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)
```



expandCapacityis called!

```
void set(key, value):

  if LoadFactor > 0.5: expandCapacity()

  ... as before ...

Value get(key):

  ... 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 HashTable 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)
```

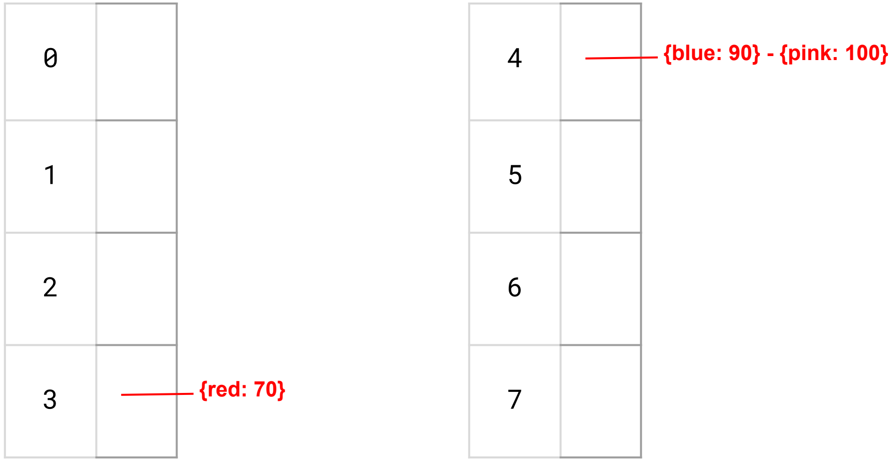
0	
1	
2	
3	

4	
5	
6	
7	

```
void set(key, value):

  if LoadFactor > 0.5: expandCapacity()

  ... as before ...

Value get(key):

  ... 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 HashTable look like after `expandCapacity` is called in `set("orange", 40)`?

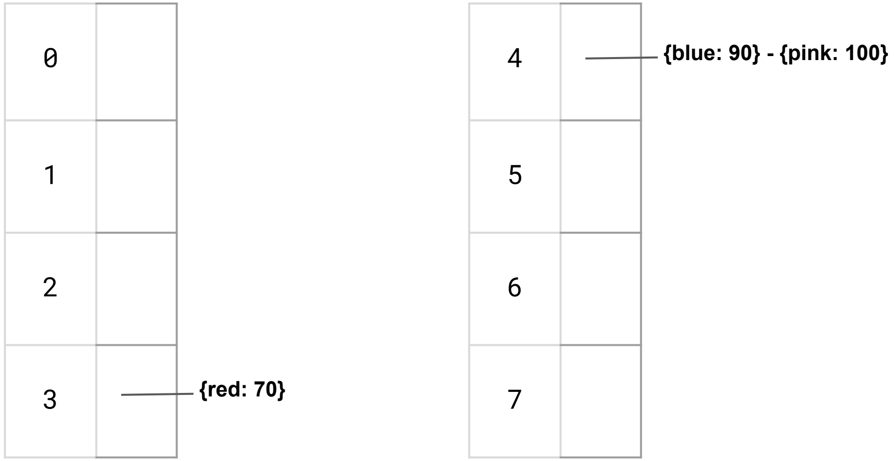
```
Example:
Start buckets array with size 4
Use string length as the hash function (In general
this is a BAD hash function)

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



```
void set(key, value):

  if LoadFactor > 0.5: expandCapacity()

  ... as before ...

Value get(key):

  ... 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 HashTable look like after set(“orange”, 40) is called?

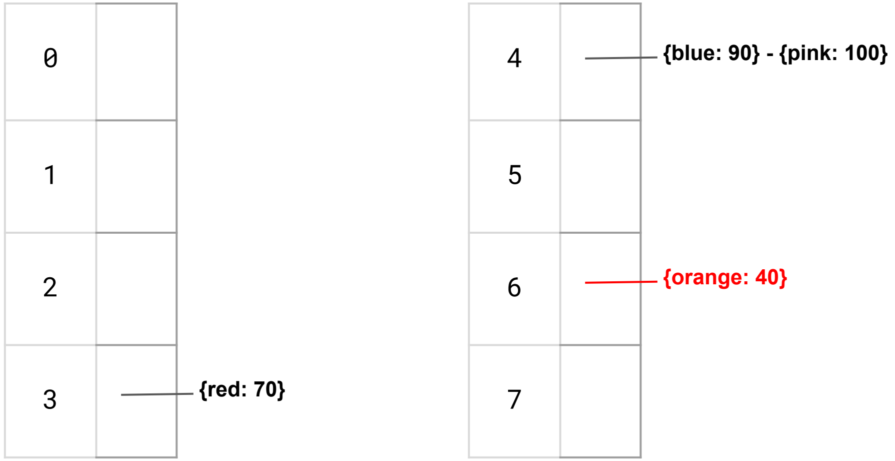
```
Example:
Start buckets array with size 4
Use string length as the hash function (In general
this is a BAD hash function)

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



```
void set(key, value):

  if LoadFactor > 0.5: expandCapacity()

  ... as before ...

Value get(key):

  ... 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 HashTable look like after set("orange", 40) is called?

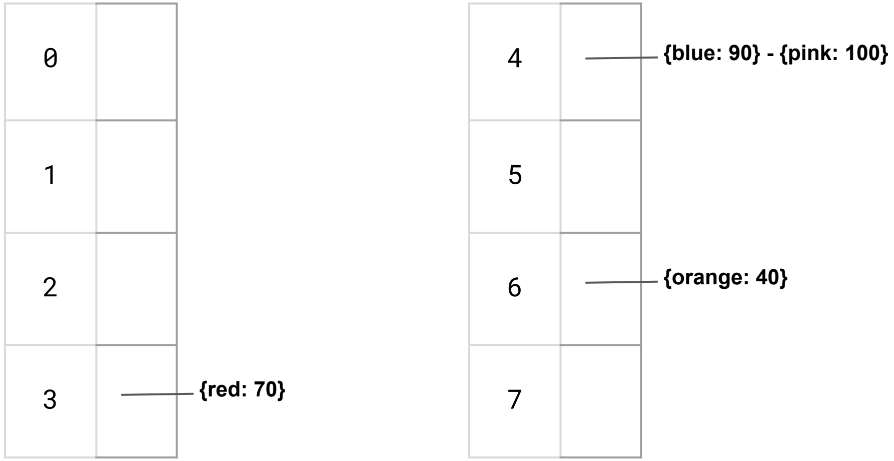
```
Example:
Start buckets array with size 4
Use string length as the hash function (In general
this is a BAD hash function)

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



```
void set(key, value):

  if LoadFactor > 0.5: expandCapacity()

  ... as before ...

Value get(key):

  ... 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 HashTable look like after set(“purplish”, 40) is called?

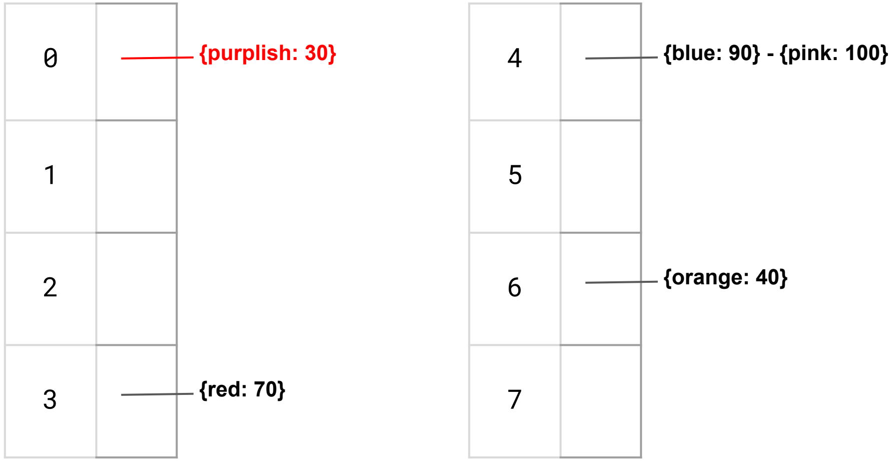
```
Example:
Start buckets array with size 4
Use string length as the hash function (In general
this is a BAD hash function)

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



```
void set(key, value):

  if LoadFactor > 0.5: expandCapacity()

  ... as before ...

Value get(key):

  ... 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 HashTable 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)
```



```
void set(key, value):

  if LoadFactor > 0.5: expandCapacity()

  ... as before ...

Value get(key):

  ... 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)
```

HashMap (Linear Probing)

What does the HashTable 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 HashTable<Key, Value> using Linear Probing has:

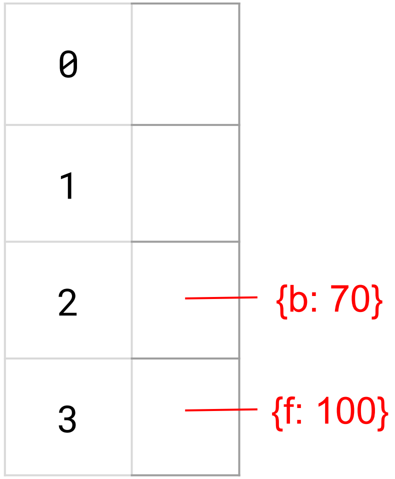
- size: an int
- buckets: an array of Entries (not of lists of Entries!)
- hash: a hash function for the Key type

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 HashTable 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 HashTable<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 HashTable<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 HashTable<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 HashTable<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 HashTable<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.

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?

Asume 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)
```

HashMap Question

```
int getIndex(String k) {
    return k.length() % 10;
}
```

Using the above hash function, which of the following pairs of subsequent set calls will result in a collision, assuming that the current number of buckets is large enough to store all the elements? (the hash table uses String keys and double values)

1. set("shampoo", 2); set("conditioner", 3);
2. set("boba", 4.5); set("crispy chicken", 8.5);
3. set("sandwich", 4); set("salad", 5);
4. set("coca cola", 2); set("coke zero", 2);
5. set("water bottle", 1.5); set("flamin' hot cheetohs", 2);

A. 1, 3, 5
B. 2, 4
C. 3, 4
D. 1, 2, 3

Answer

set("boba", 4.5); set("crispy chicken", 8.5);
set("coca cola", 2); set("coke zero", 2);

“boba” and “crispy chicken” gives the same hash. “coca cola” and “coke zero” also gives the same hash, so they both result in one collision. Other choices don’t cause collision.