CSE 12 Week 8 Discussion

2-23-21

Focus: Review

Reminders

- PA7 **(closed)** due Tuesday, March 2 @ 11:59 PM

- PA5 Resubmission due Friday, Feb. 26th @ 11:59 PM
- PA6 Resubmission due Friday, Mar. 5th @ 11:59 PM

- PA_-PRACTICE-ONLY are for your benefit, we will not grade them

Midterm 2 Concepts

- Time Complexities
- Sorting
- Maps and HashTables

Focus on **PAs**, **lecture**, and Quizzes

Runtime Analysis

Counting Steps

```
// a.size = A, b.size = B                          No match        Jth in a, Kth in b   1st in a and b

 1 String findMatch(LinkedList a, LinkedList b) {
 2  Node aNode = a.first.next;                     // 1            // 1                 // 1
 3  for (int i = 0; i < a.size; i++) {             // A + 1        // J                 // 1
 4    Node bNode = b.first.next;                   // a            // J                 // 1
 5    for (int j = 0; j < b.size; j++) {           // A*(B + 1)    // (J-1)*(B+1)+K     // 1
 6      if (aNode.value.equals(bNode.value) {      // A*B          // (J-1)*B+K         // 1
 7        return aNode.value;                      // 0            // 1                 // 1
 8      }
 9      bNode = bNode.next;                        // A*B          // (J-1)*B+(K-1)     // 0
10    }
11    aNode = aNode.next;                          // A            // J-1               // 0
12  }
13  return null;                                   // 1            // 0                 // 0
14 }
```

Worst Case Scenario

Best Case Scenario

Common Time Complexities in Order

ordering of functions from slowest-growing (indeed, the first two *shrink* as n increases) to fastest-growing that you might find helpful:

- f(n) = 1/(n^2)
- f(n) = 1/n
- f(n) = 1
- f(n) = log(n)
- f(n) = sqrt(n)
- f(n) = n
- f(n) = n^2
- f(n) = n^3
- f(n) = n^4
- … and so on for constant polynomials …
- f(n) = 2^n
- f(n) = n!
- f(n) = n^n

Summary

Big-O

- **Upper bound** on a function
- f(n) = O(g(n)) means that we can expect f(n) will always be **under** the bound g(n)
  - But we don't count n up to some starting point n_0
  - And we can "cheat" a little bit by moving g(n) up by multiplying by some constant c

Big-Ω

- **Lower bound** on a function
- f(n) = Ω(g(n)) means that we can expect f(n) will always be **over** the bound g(n)
  - But we don't count n up to some starting point n_0
  - And we can "cheat" a little bit by moving g(n) down by multiplying by some constant c

Big-θ

- **Tight bound** on a function.
- If $f(n) = O(g(n))$ *and* $f(n) = \Omega(g(n))$, then $f(n) = \theta(g(n))$.
- Basically it means that f(n) and g(n) are interchangeable
- Examples:
  - $3n+20 = \theta(10n+7)$
  - $5n^2 + 50n + 3 = \theta(5n^2 + 100)$

Review Question

Which is the upper bound in this statement?

$g(n) = O(f(n))$

a) g(n)
b) f(n)

Review Question

Which is the upper bound in this statement?

$g(n) = O(f(n))$

a) g(n)
b) f(n)

Review Question

Which is the upper bound in this statement?

$g(n) = \Omega(f(n))$

a) g(n)
b) f(n)

Review Question

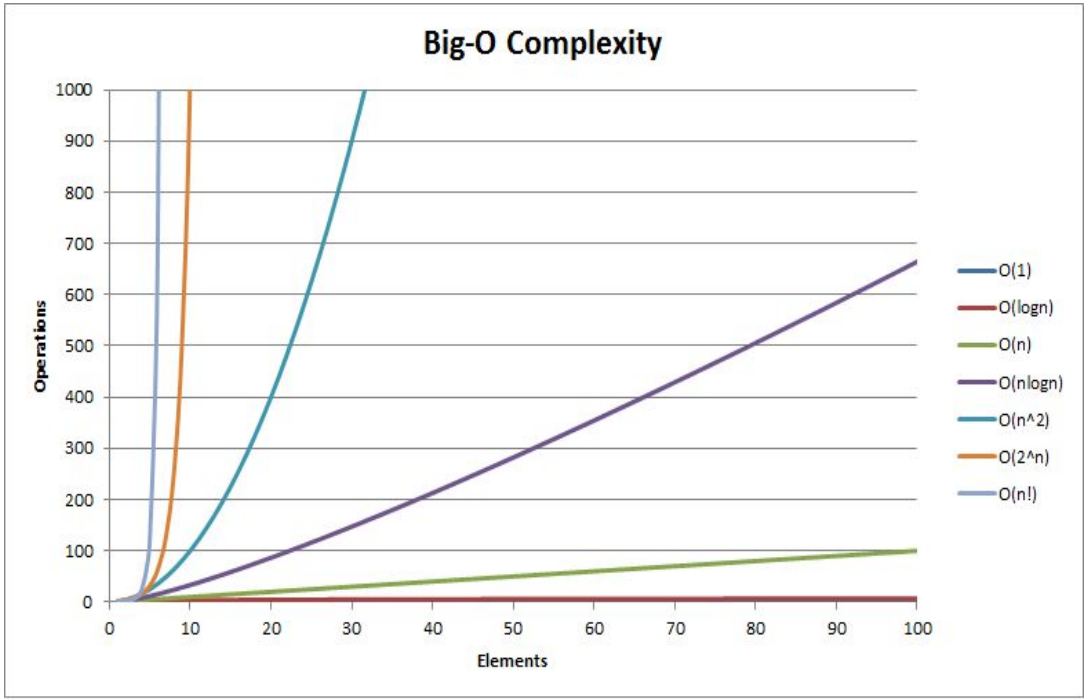
Which is the upper bound in this statement?

$g(n) = \Omega(f(n))$

a) g(n)

b) f(n)

Big-O review



- Relative to input n
- Constants do not matter
  - O(3n)=O(n)
- Higher order values dominate
  - O(n^2 + n) = O(n^2)

True or False?

$n + 5n^3 + 8n^4 = O(n)$

a) True
b) False

True or False?

$n + 5n^3 + 8n^4 = O(n)$

a) True
b) False

True or False?

n! + n^2 = O(nlog(n))

a) True
b) False

True or False?

n! + n^2 = O(nlog(n))

a) True
b) False

True or False?

$2^n + n\log(n) = O(n!)$

a) True
b) False

True or False?

$2^n + n\log(n) = O(n!)$

a) True
b) False

True or False?

$1/(n^2) + 5 = O(1/n)$

a) True
b) False

True or False?

$1/(n^2) + 5 = O(1/n)$

a) True
b) False

Code to Runtime

Mystery Functions of PA4: https://github.com/CSE12-W21-Assignments/cse12-wi21-pa4-Runtime-starter

```
public static void dummyMethod(int n) {
      int a = 0;
      for(int i = 0; i < n * n; i += 1) {
                  a += i;
            }
      }
}
```

Runtime is O(n^2)

Sorting

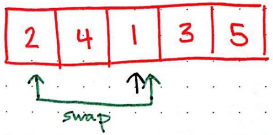
Sorting Algorithms

- Selection Sort: finds the minimum element in a list and moves it to the end of a sorted prefix in the list
- Insertion Sort: repeatedly takes the next element in a list, inserts it into the correct ordered position within a sorted prefix of the list
- Quicksort: picks an element as pivot and partitions the given array around the picked pivot
- Merge sort: recursively sorts half of the array

Simplified Selection Sort:

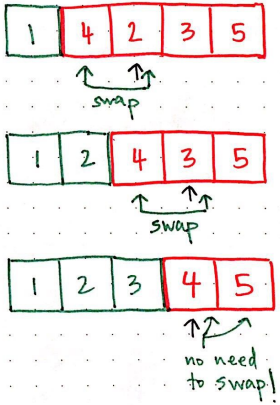
Our smallest number starts off as the first number – whatever it is.

We'll iterate through the whole dataset until we find the actual smallest number. Then, we'll swap it to be in the 1st position.

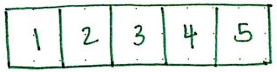


We'll continue this process:

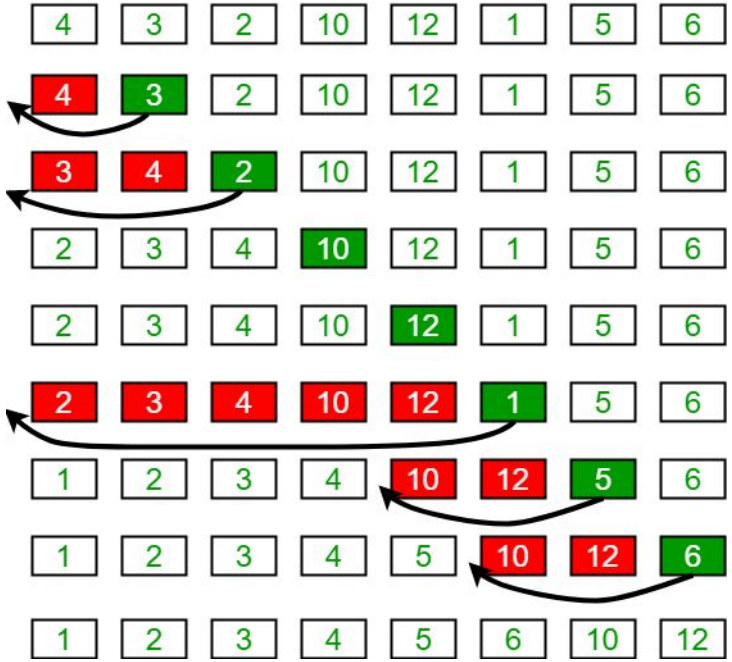
1/ find smallest unsorted number,
2/ swap it to switch places with the unsorted number at the front of the list,
3/ do the same with the next number.

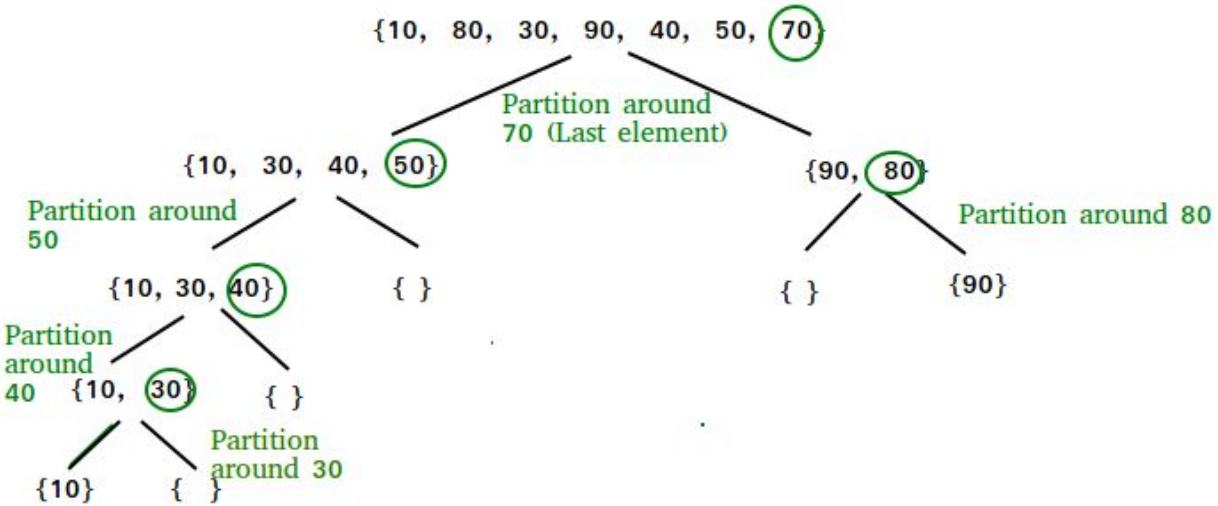


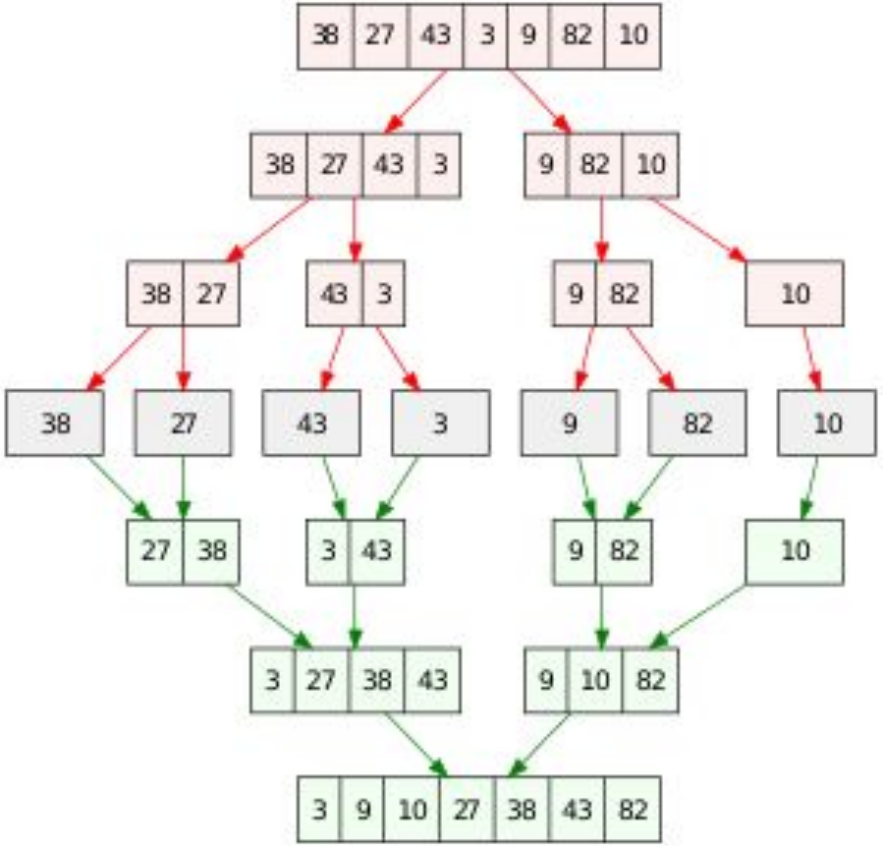
Eventually, we'll end up with a totally sorted dataset!!



Insertion Sort Execution Example




{10, 80, 30, 90, 40, 50, 70}
Partition around 70 (Last element)
{10, 30, 40, 50}
{90, 80}
Partition around 50
Partition around 80
{10, 30, 40}
{ }
{ }
{90}
Partition around 40
{10, 30}
{ }
Partition around 30
{10}
{ }


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

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

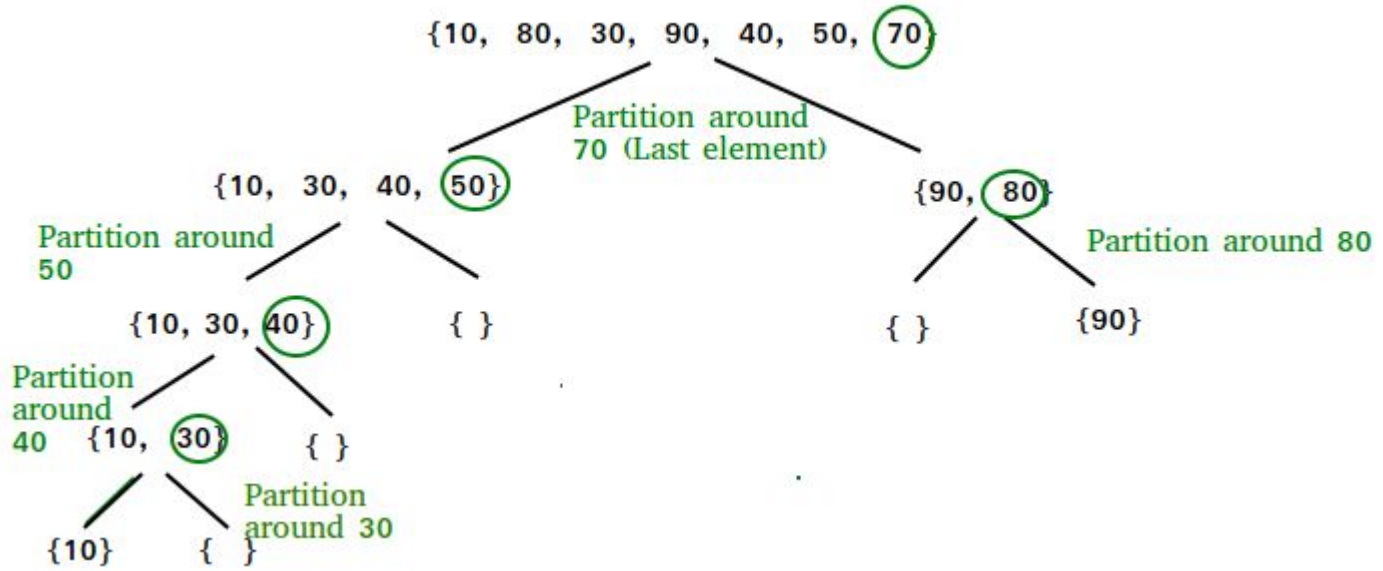
	Insertion	**Selection**	**Merge**	**Quick**
Best case time	O(n)	O(n^2)	O(n*logn)	O(n*logn)
Worst case time	O(n^2)	O(n^2)	O(n*logn)	O(n^2)

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)`

What is a partition in quicksort?

- Partitioning is a component in quicksort and an algorithm that is called again and again until the original array elements are sorted as single-element arrays
- 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
    - **Note:** the array is not necessarily completely sorted here (and probably isn't)!
- Now the array is divided into two at the point where the pivot index ended up
- Partition is called on each of these two subarrays using the "same" pivot point location (we picked the last element and do so again for each of these subarrays)
- We partition again, and this process is repeated until we have single-element arrays that are sorted between themselves; **see visualization in next slide**

Graphic visualization of quicksort



Resultant single-element arrays to combine: `{{10}, {30}, {40}, {50}, {70}, {80}, {90}}`

How can we test **any** partition?

Direct unit testing is not always possible. For example, if we step through our *own* partition algorithm, we know what exact array it should output, and we can use `assertArrayEquals` to compare the returned array and what we expect. But, different partitioners could return different arrays that are still valid partitions! In this case, we want to test for a valid partition by testing whether or not the array produced matches with our general constraints: all elements from before are still present, the array length hasn't changed, all elements to the left of the pivot index are less than or equal to the pivot value, and all elements to the right of the pivot index are greater than or equal to the pivot value

Review Quiz Questions

Which of the following will result in the most number of element comparisons using selection sort? Select all that apply:

A. {1, 2, 3, 4, 5}
B. {5, 4, 3, 2, 1}
C. {1, 3, 5, 2, 4}
D. {1, 4, 5, 3, 2}
E. {1, 1, 1, 1, 1}

Review Quiz Questions

Which of the following will result in the most number of element comparisons using selection sort? Select all that apply:

A. {1, 2, 3, 4, 5}
B. {5, 4, 3, 2, 1}
C. {1, 3, 5, 2, 4}
D. {1, 4, 5, 3, 2}
E. {1, 1, 1, 1, 1}

All of them!

Which of the following descriptions of pivot selection will result in the best case quicksort runtime?

A. Randomly choosing the pivot
B. Choosing the first value as the pivot
C. Choosing the median index as the pivot
D. Choosing the median value as the pivot
E. There is no definite pivot selection method that will always result in best case runtime

Which of the following descriptions of pivot selection will result in the best case quicksort runtime?

A. Randomly choosing the pivot
B. Choosing the first value as the pivot
C. Choosing the median index as the pivot
D. Choosing the median value as the pivot
E. There is no definite pivot selection method that will always result in best case runtime

Maps and HashTables

Maps

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 entry is determined by a hash function that calculates index using key value (useful for quick lookup and insert)
- Contains methods get(Object key), put(K key, V value), size(), replace(K key, V value) etc.
- ***Keys need to be unique***
- Existing data structures we can use to implement this - ArrayList!

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;
}
```

What is a bucket?

A: The collection of key values at a specific index

B: The sum of the keys in a given hash table

C: The collection of elements at a specific index

D: The collection of keys at a specific index

What is a bucket?

A: The collection of key values at a specific index

B: The sum of the keys in a given hash table

C: The collection of elements at a specific index

D: The collection of keys at a specific index

Given the example below, what does the HashTable 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 HashTable<Key, Value> using Separate Chaining has:

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

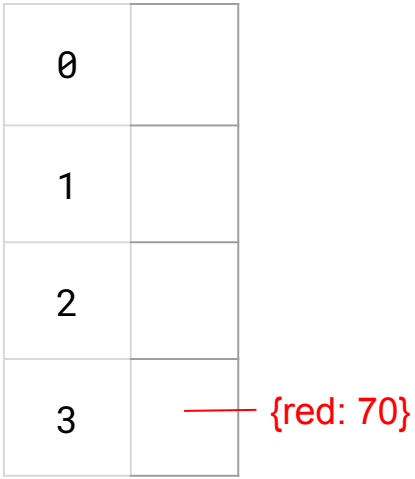
An Entry is a single {key: value} pair.

```
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 HashTable 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 HashTable<Key, Value> using Separate Chaining has:

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

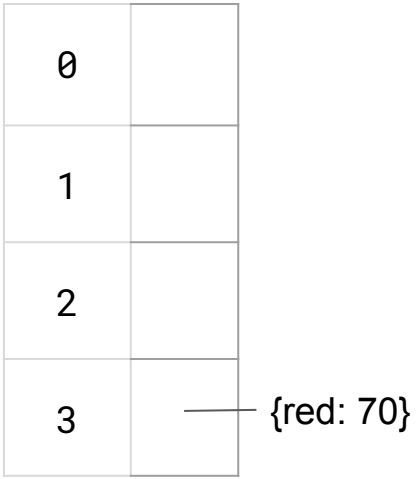
An Entry is a single {key: value} pair.

```
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 HashTable 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 HashTable<Key, Value> using Separate Chaining has:

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

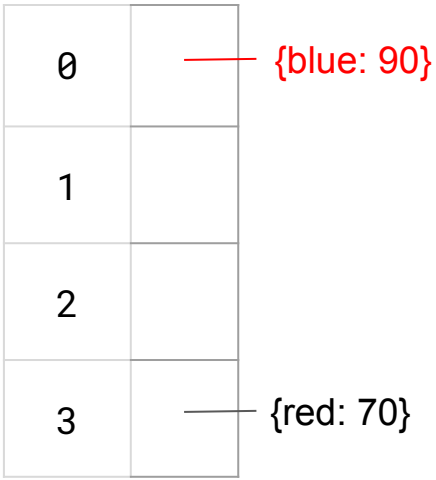
An Entry is a single {key: value} pair.

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

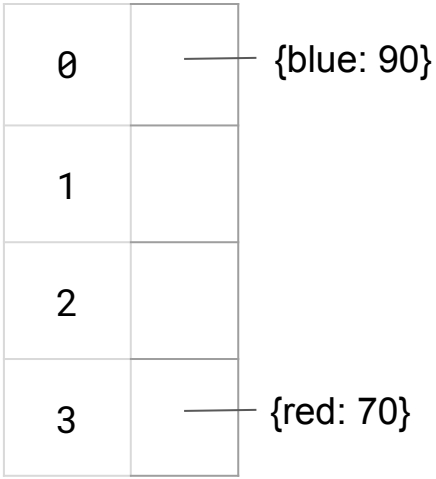
An Entry is a single {key: value} pair.

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

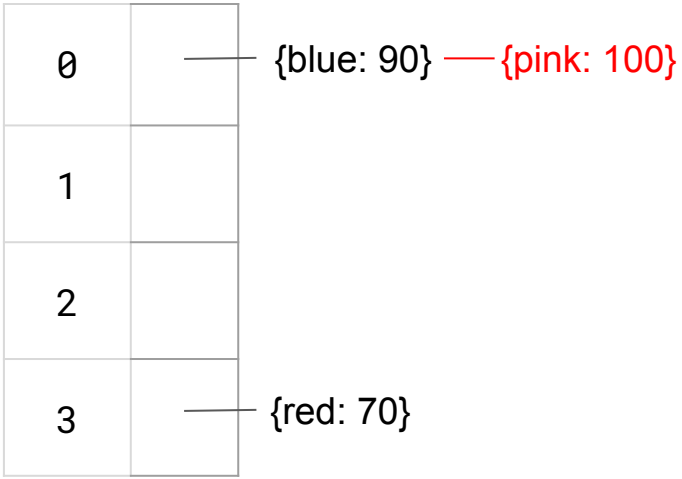
An Entry is a single {key: value} pair.

```
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 HashTable 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 HashTable<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 HashTable<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 HashTable<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 1?

A: 0

B: 1

C: 2

D: 3

E: more than 3

A HashTable<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 1?

A: 0

B: 1

C: 2

D: 3

E: more than 3

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

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

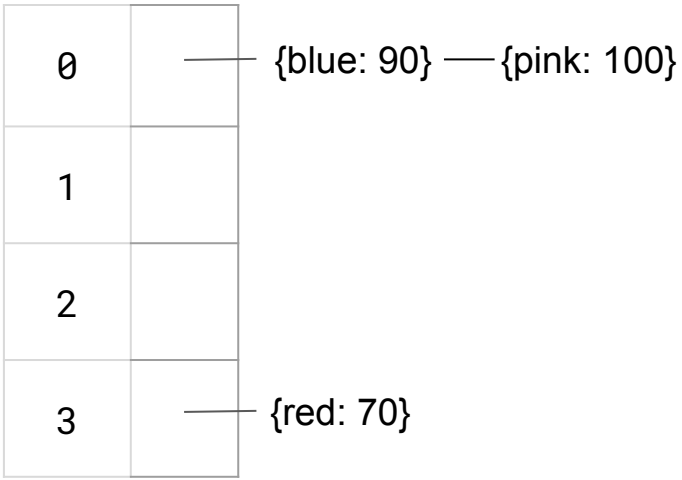
An Entry is a single {key: value} pair.

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

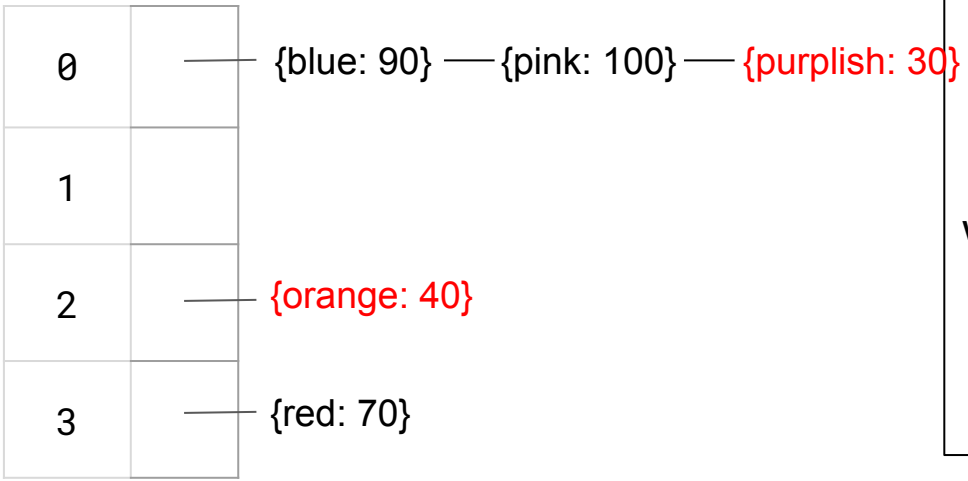
An Entry is a single {key: value} pair.

```
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 HashTable, 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 HashTable<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?

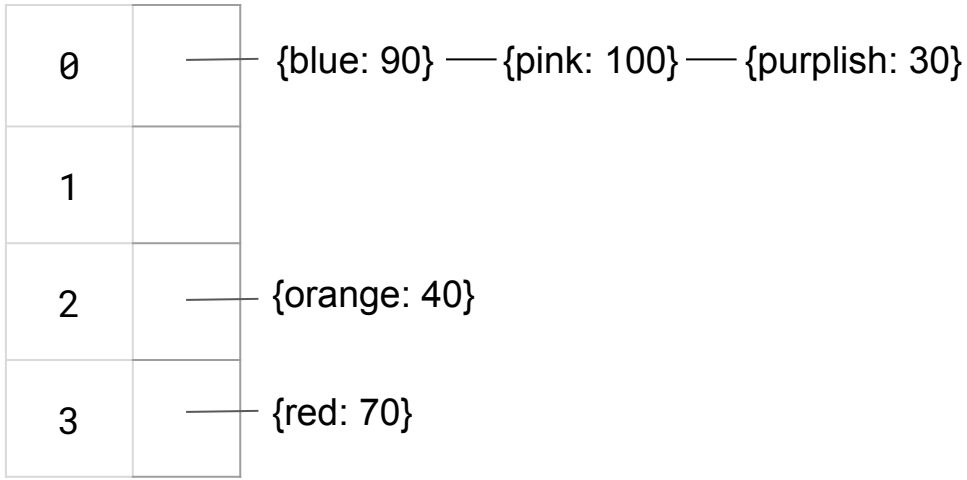
A: # elements * # buckets / 2

B: # buckets * # elements

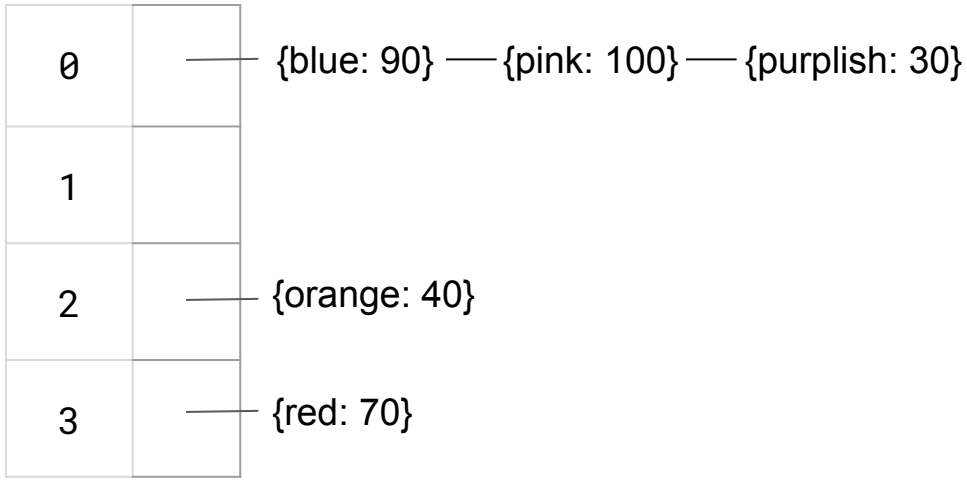
C: # buckets / # elements

D: # elements / # buckets

What is the load factor of the HashTable below?



What is the load factor of the HashTable below?



Load Factor: 5/4

What is the load factor after the line `set("red", 70)` 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)
```

0	
1	
2	
3	

A HashTable<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 the load factor after the line `set("red", 70)` 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)
```

0	
1	
2	
3	

Load Factor: 1/4

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

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

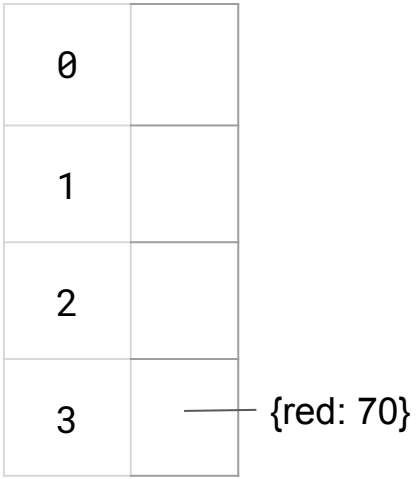
An Entry is a single {key: value} pair.

```
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 the load factor after the line `set("blue", 90)` 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)
```



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

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

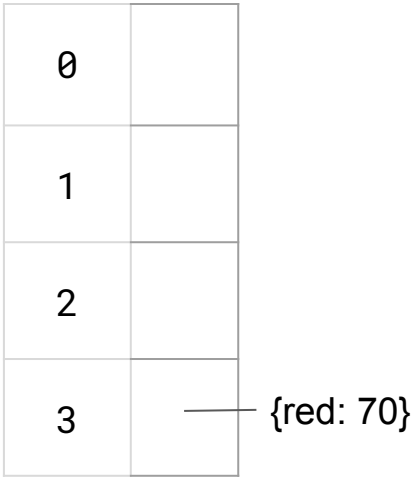
An Entry is a single {key: value} pair.

```
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 the load factor after the line `set("blue", 90)` 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: 1/2

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

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

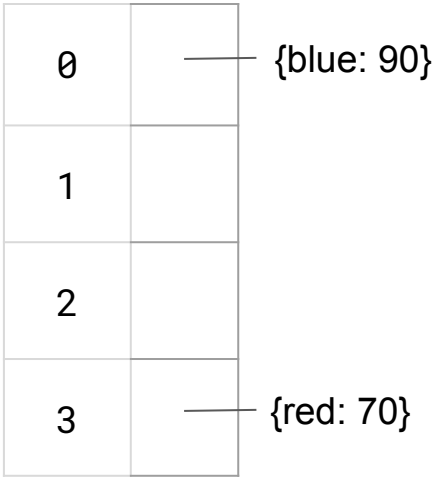
An Entry is a single {key: value} pair.

```
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 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)
```



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

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

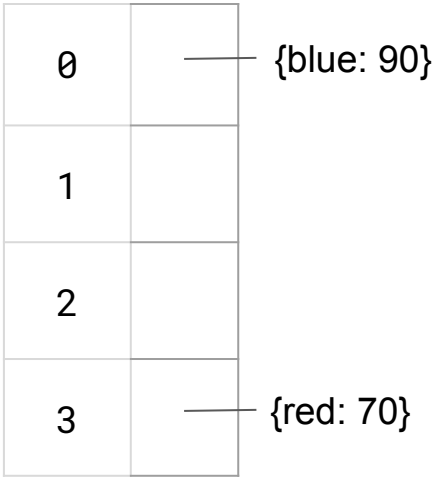
An Entry is a single {key: value} pair.

```
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 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 HashTable<Key, Value> using Separate Chaining has:

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

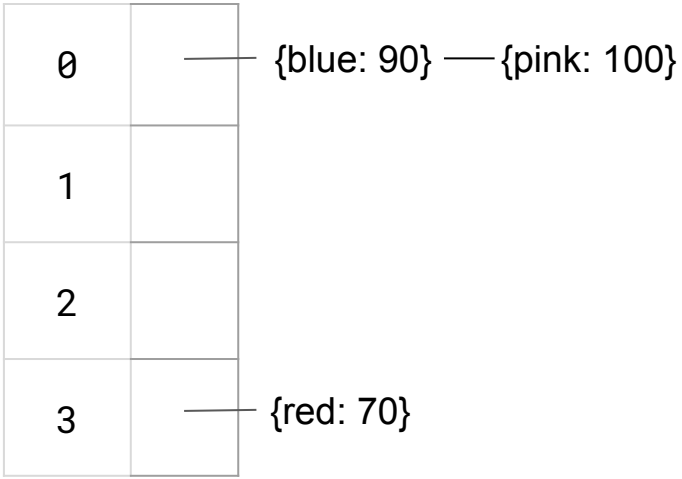
An Entry is a single {key: value} pair.

```
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 HashTable<Key, Value> using Separate Chaining has:

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

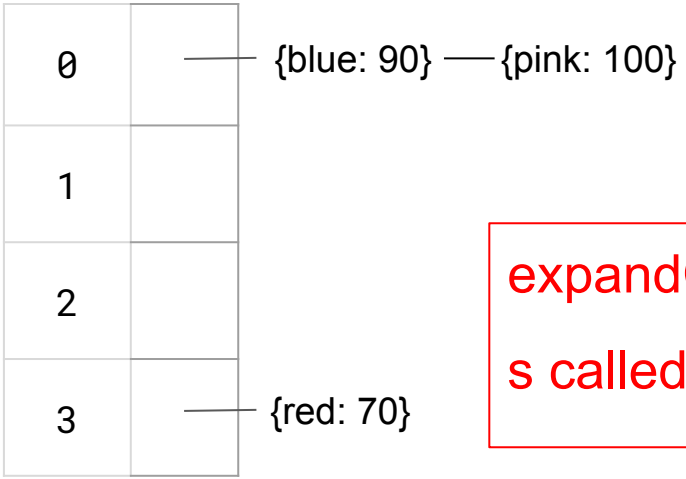
An Entry is a single {key: value} pair.

```
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 HashTable<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 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	

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

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

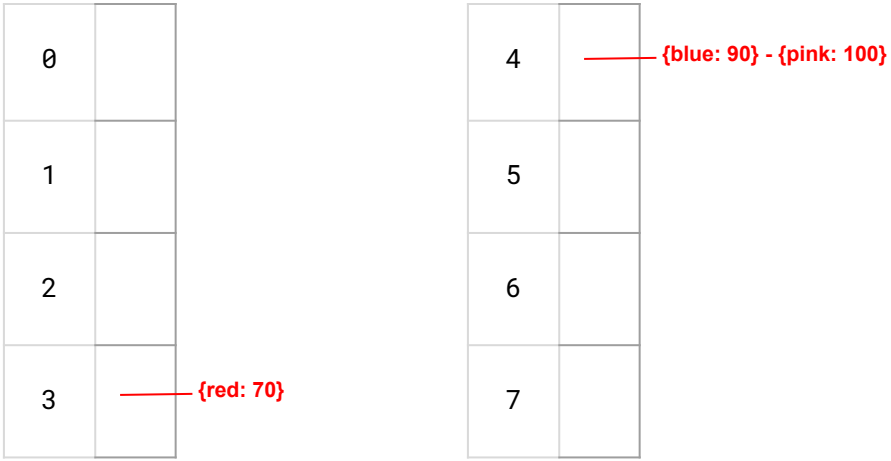
An Entry is a single {key: value} pair.

```
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 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)
```



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

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

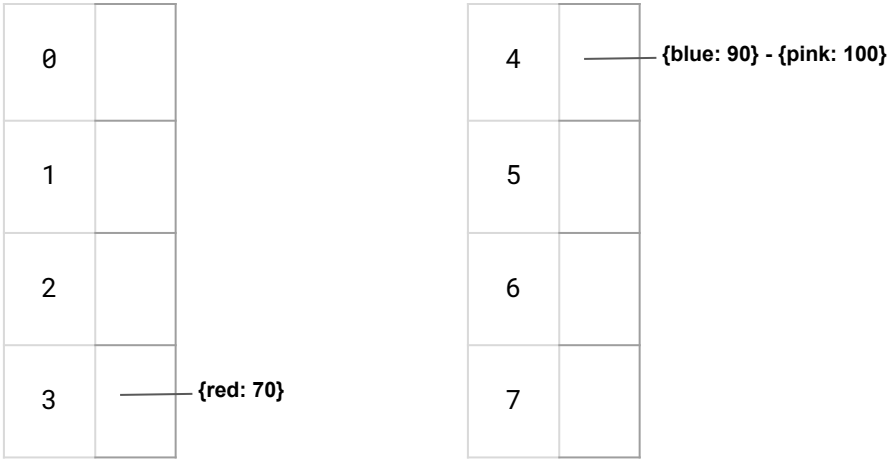
An Entry is a single {key: value} pair.

```
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 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)
```



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

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

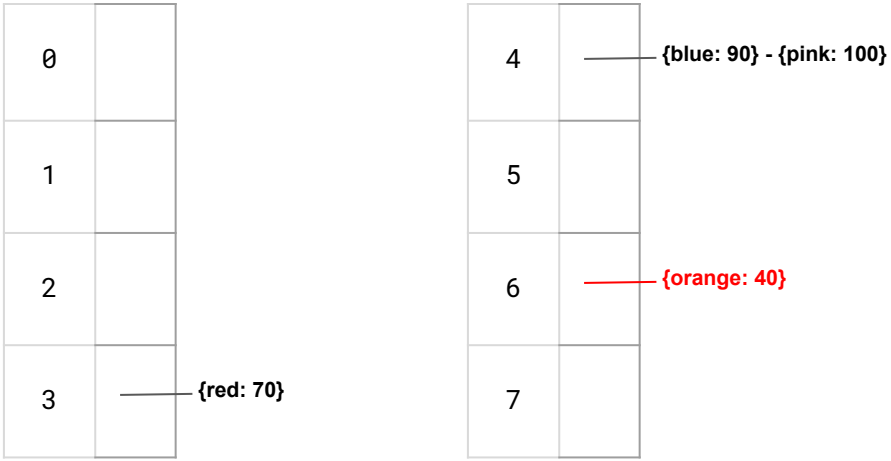
An Entry is a single {key: value} pair.

```
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 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)
```



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

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

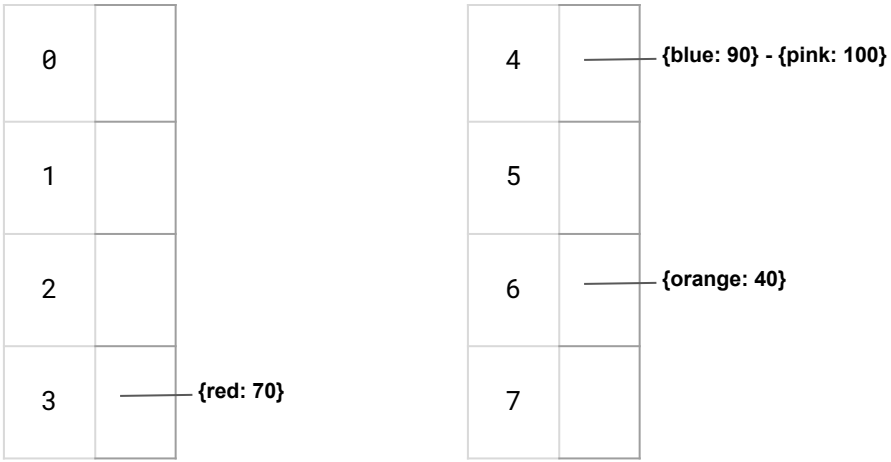
An Entry is a single {key: value} pair.

```
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 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)
```



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

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

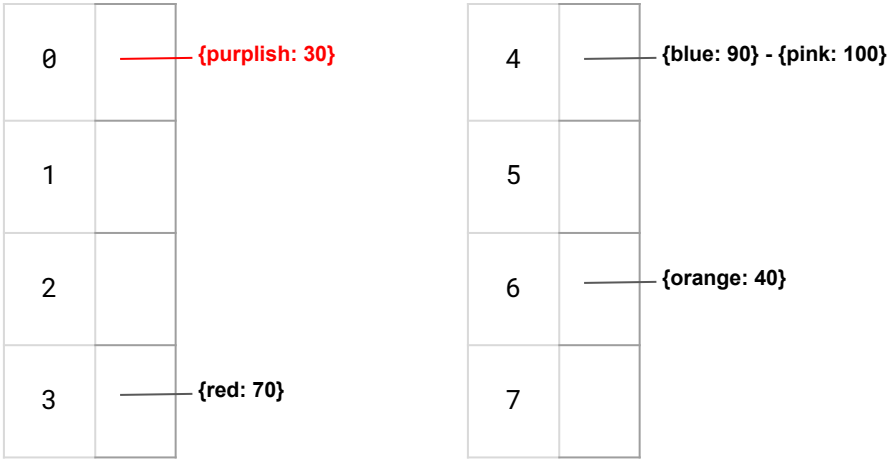
An Entry is a single {key: value} pair.

```
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 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)
```



A HashTable<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)
```

During which line of the example will expandCapacity be called inside of set?

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: set("red", 70)

B: set("blue", 90)

C: set("pink", 100)

D: set("orange", 40)

E: set("purplish", 30)

A HashTable<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)
```

During which line of the example will expandCapacity be called inside of set?

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: set("red", 70)

B: set("blue", 90)

C: set("pink", 100)

D: set("orange", 40)

E: set("purplish", 30)

A HashTable<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)
```

Amortized Analysis and Linear Probing

```
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 elements are in bucket 1?

A: 0

B: 1

C: 2

D: 3

E: more than 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)
```

```
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 elements are in bucket 1?

A: 0

B: 1

C: 2

D: 3

E: more than 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)
```

```
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 elements are in bucket 2?

A: 0

B: 1

C: 2

D: 3

E: more than 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)
```

```
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 elements are in bucket 2?

A: 0

B: 1

C: 2

D: 3

E: more than 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)
```

```
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 elements are in bucket 3?

A: 0

B: 1

C: 2

D: 3

E: more than 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)
```

```
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 elements are in bucket 3?

A: 0

B: 1

C: 2

D: 3

E: more than 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)
```

```
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 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,

how many entries are checked when doing

`set("f", 100)`?

A: 0

B: 1

C: 2

D: 3

E: more than 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)
```

```
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,

what will the result of `get("f")` be after this sequence?

A: 70

B: 90

C: 100

D: null

E: 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,

what will the result of `get("f", 100)` be after this sequence?

A: 70

B: 90

C: 100

D: null

E: 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)
```

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:

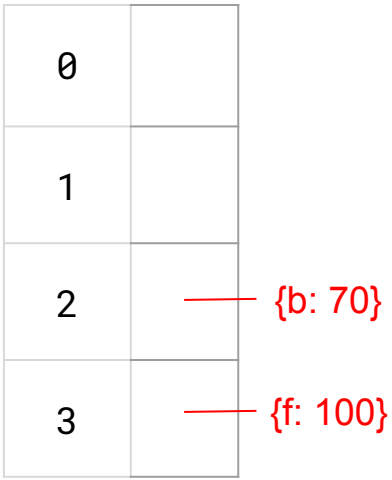
- size: an int
- buckets: an array of Entries (not of lists of Entries!)
- hash: a hash function for the Key type

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)
```

Are there any other issues that need to be fixed?

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
    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 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.

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 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
    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 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.

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)
```

What is the best case for add()

```
public class AList<E> implements List<E> {

  E[] elements;
  int size;

  @SuppressWarnings("unchecked")
  public AList() {
    this.elements = (E[])(new Object[2]);
    this.size = 0;
  }

  public void add(E s) {
    expandCapacity();
    this.elements[this.size] = s;
    this.size += 1;
  }

  @SuppressWarnings("unchecked")
  private void expandCapacity() {
    int currentCapacity = this.elements.length;
    if(this.size < currentCapacity) { return; }

    E[] expanded = (E[])(new Object[currentCapacity * 2]);

    for(int i = 0; i < this.size; i += 1) {
      expanded[i] = this.elements[i];
    }
    this.elements = expanded;
  }
}
```

What is the best case for add()

```
public class AList<E> implements List<E> {

  E[] elements;
  int size;

  @SuppressWarnings("unchecked")
  public AList() {
    this.elements = (E[])(new Object[2]);
    this.size = 0;
  }

  public void add(E s) {
    expandCapacity();
    this.elements[this.size] = s;
    this.size += 1;
  }

  @SuppressWarnings("unchecked")
  private void expandCapacity() {
    int currentCapacity = this.elements.length;
    if(this.size < currentCapacity) { return; }

    E[] expanded = (E[])(new Object[currentCapacity * 2]);

    for(int i = 0; i < this.size; i += 1) {
      expanded[i] = this.elements[i];
    }
    this.elements = expanded;
  }
}
```

Best Case: O(1)

expandCapacity returns immediately

What is the worst case for add()

```
public class AList<E> implements List<E> {

  E[] elements;
  int size;

  @SuppressWarnings("unchecked")
  public AList() {
    this.elements = (E[])(new Object[2]);
    this.size = 0;
  }

  public void add(E s) {
    expandCapacity();
    this.elements[this.size] = s;
    this.size += 1;
  }

  @SuppressWarnings("unchecked")
  private void expandCapacity() {
    int currentCapacity = this.elements.length;
    if(this.size < currentCapacity) { return; }

    E[] expanded = (E[])(new Object[currentCapacity * 2]);

    for(int i = 0; i < this.size; i += 1) {
      expanded[i] = this.elements[i];
    }
    this.elements = expanded;
  }
}
```

What is the worst case for add()

```
public class AList<E> implements List<E> {

  E[] elements;
  int size;

  @SuppressWarnings("unchecked")
  public AList() {
    this.elements = (E[])(new Object[2]);
    this.size = 0;
  }

  public void add(E s) {
    expandCapacity();
    this.elements[this.size] = s;
    this.size += 1;
  }

  @SuppressWarnings("unchecked")
  private void expandCapacity() {
    int currentCapacity = this.elements.length;
    if(this.size < currentCapacity) { return; }

    E[] expanded = (E[])(new Object[currentCapacity * 2]);

    for(int i = 0; i < this.size; i += 1) {
      expanded[i] = this.elements[i];
    }
    this.elements = expanded;
  }
}
```

Worst Case: O(n)

When `expandCapacity` is actually triggered

If we add 6 elements to an empty AList, what is the **sum of all the lengths of arrays created in (including constructor and expandCapacity)?**

A: 8

B: 10

C: 12

D: 14

E: 16

```
public class AList<E> implements List<E> {

  E[] elements;
  int size;

  @SuppressWarnings("unchecked")
  public AList() {
    this.elements = (E[])(new Object[2]);
    this.size = 0;
  }

  public void add(E s) {
    expandCapacity();
    this.elements[this.size] = s;
    this.size += 1;
  }

  @SuppressWarnings("unchecked")
  private void expandCapacity() {
    int currentCapacity = this.elements.length;
    if(this.size < currentCapacity) { return; }

    E[] expanded = (E[])(new Object[currentCapacity * 2]);

    for(int i = 0; i < this.size; i += 1) {
      expanded[i] = this.elements[i];
    }
    this.elements = expanded;
  }
}
```

If we add 6 elements to an empty AList, what is the **sum of all the lengths of arrays created in (including constructor and expandCapacity)?**

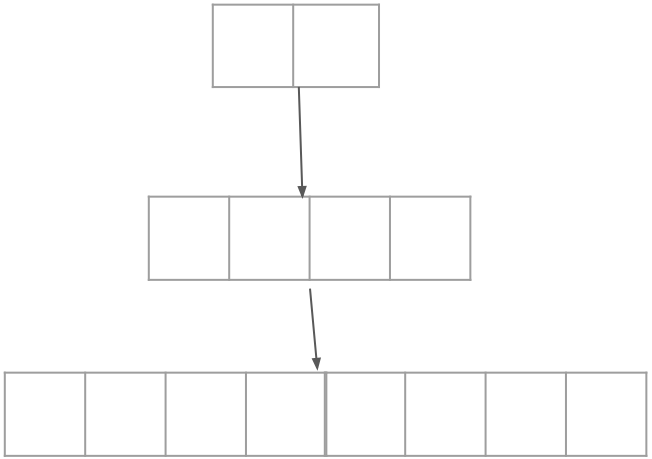
A: 8

B: 10

C: 12

D: 14

E: 16



```
public class AList<E> implements List<E> {

  E[] elements;
  int size;

  @SuppressWarnings("unchecked")
  public AList() {
    this.elements = (E[])(new Object[2]);
    this.size = 0;
  }

  public void add(E s) {
    expandCapacity();
    this.elements[this.size] = s;
    this.size += 1;
  }

  @SuppressWarnings("unchecked")
  private void expandCapacity() {
    int currentCapacity = this.elements.length;
    if(this.size < currentCapacity) { return; }

    E[] expanded = (E[])(new Object[currentCapacity * 2]);

    for(int i = 0; i < this.size; i += 1) {
      expanded[i] = this.elements[i];
    }
    this.elements = expanded;
  }
}
```

If we add 6 elements to an empty AList, what is the **total number of times an element is copied in expandCapacity?**

A: 6

B: 8

C: 10

D: 12

E: 16

```
public class AList<E> implements List<E> {

  E[] elements;
  int size;

  @SuppressWarnings("unchecked")
  public AList() {
    this.elements = (E[])(new Object[2]);
    this.size = 0;
  }

  public void add(E s) {
    expandCapacity();
    this.elements[this.size] = s;
    this.size += 1;
  }

  @SuppressWarnings("unchecked")
  private void expandCapacity() {
    int currentCapacity = this.elements.length;
    if(this.size < currentCapacity) { return; }

    E[] expanded = (E[])(new Object[currentCapacity * 2]);

    for(int i = 0; i < this.size; i += 1) {
      expanded[i] = this.elements[i];
    }
    this.elements = expanded;
  }
}
```

If we add 6 elements to an empty AList, what is the **total number of times an element is copied in expandCapacity?**

A: 6

B: 8

C: 10

D: 12

E: 16

```
public class AList<E> implements List<E> {

  E[] elements;
  int size;

  @SuppressWarnings("unchecked")
  public AList() {
    this.elements = (E[])(new Object[2]);
    this.size = 0;
  }

  public void add(E s) {
    expandCapacity();
    this.elements[this.size] = s;
    this.size += 1;
  }

  @SuppressWarnings("unchecked")
  private void expandCapacity() {
    int currentCapacity = this.elements.length;
    if(this.size < currentCapacity) { return; }

    E[] expanded = (E[])(new Object[currentCapacity * 2]);

    for(int i = 0; i < this.size; i += 1) {
      expanded[i] = this.elements[i];
    }
    this.elements = expanded;
  }
}
```

If we add 20 elements to an empty AList, **how many times is expandCapacity expanding?**

A: 2

B: 3

C: 4

D: 5

E: 6

```
public class AList<E> implements List<E> {

  E[] elements;
  int size;

  @SuppressWarnings("unchecked")
  public AList() {
    this.elements = (E[])(new Object[2]);
    this.size = 0;
  }

  public void add(E s) {
    expandCapacity();
    this.elements[this.size] = s;
    this.size += 1;
  }

  @SuppressWarnings("unchecked")
  private void expandCapacity() {
    int currentCapacity = this.elements.length;
    if(this.size < currentCapacity) { return; }

    E[] expanded = (E[])(new Object[currentCapacity * 2]);

    for(int i = 0; i < this.size; i += 1) {
      expanded[i] = this.elements[i];
    }
    this.elements = expanded;
  }
}
```

If we add 20 elements to an empty AList, **how many times is expandCapacity expanding?**

A: 2

B: 3

C: 4

D: 5

E: 6

2 -> 4 4 -> 8 8 -> 16 16 -> 32

```
public class AList<E> implements List<E> {

  E[] elements;
  int size;

  @SuppressWarnings("unchecked")
  public AList() {
    this.elements = (E[])(new Object[2]);
    this.size = 0;
  }

  public void add(E s) {
    expandCapacity();
    this.elements[this.size] = s;
    this.size += 1;
  }

  @SuppressWarnings("unchecked")
  private void expandCapacity() {
    int currentCapacity = this.elements.length;
    if(this.size < currentCapacity) { return; }

    E[] expanded = (E[])(new Object[currentCapacity * 2]);

    for(int i = 0; i < this.size; i += 1) {
      expanded[i] = this.elements[i];
    }
    this.elements = expanded;
  }
}
```

If we add 20 elements to an empty AList, **what is the length of the array created in each of those calls to expandCapacity?**

(open-ended, no multiple-choice)

```
public class AList<E> implements List<E> {

  E[] elements;
  int size;

  @SuppressWarnings("unchecked")
  public AList() {
    this.elements = (E[])(new Object[2]);
    this.size = 0;
  }

  public void add(E s) {
    expandCapacity();
    this.elements[this.size] = s;
    this.size += 1;
  }

  @SuppressWarnings("unchecked")
  private void expandCapacity() {
    int currentCapacity = this.elements.length;
    if(this.size < currentCapacity) { return; }

    E[] expanded = (E[])(new Object[currentCapacity * 2]);

    for(int i = 0; i < this.size; i += 1) {
      expanded[i] = this.elements[i];
    }
    this.elements = expanded;
  }
}
```

If we add 20 elements to an empty AList, **what is the length of the array created in each of those calls to expandCapacity?**

Length of the array created each time:

2 4 8 16 32

This answer also represents the total count of the number of times we have spent allocating array slots and the amount of time we have spent copying.

The sum of the lengths of the previous arrays allocated is 2 less than the length of the next array we will allocate. So, consider performing *m* adds, where *m* is some power of 2 (to keep things simple at first). We must have an array whose size is exactly *m*, and we overall performed

2 + 4 + 8 + ... + m

array slot allocations (and the number of times we copied an element is less).

Altogether, that means we've done work *proportional to m* in *all* the calls to expandCapacity to make enough space for *m* elements. The specific formula for this summation results in a conclusion for *m + (m-2)* slots allocated.

So **on average**, per add we do constant work *(m + (m - 2))/m*

Add is <u>amortized</u> constant time - another kind of average case!

(If *m* is not a power of two, we must have an array that is the next power of two that is greater than *m*, which is no greater than *2 * m*. This means the total work is no more than *(2m + (2m - 2))*, still proportional to *m*.

```
public class AList<E> implements List<E> {

  E[] elements;
  int size;

  @SuppressWarnings("unchecked")
  public AList() {
    this.elements = (E[])(new Object[2]);
    this.size = 0;
  }

  public void add(E s) {
    expandCapacity();
    this.elements[this.size] = s;
    this.size += 1;
  }

  @SuppressWarnings("unchecked")
  private void expandCapacity() {
    int currentCapacity = this.elements.length;
    if(this.size < currentCapacity) { return; }

    E[] expanded = (E[])(new Object[currentCapacity * 2]);

    for(int i = 0; i < this.size; i += 1) {
      expanded[i] = this.elements[i];
    }
    this.elements = expanded;
  }
}
```

Quiz Questions

Which of the following statements about hash tables is/are true?

1. Items are stored as key, value pairs
2. All keys must be unique
3. All values must be unique
4. The best case for get and set is O(n)
5. Hash tables are ordered by keys
6. A hash table can have keys of only one type
7. A hash table can have values of multiple types

A. 1, 3, 5
B. 1, 4, 6
C. 1, 2, 6
D. 2
E. 3, 4, 7

Quiz Questions

Which of the following statements about hash tables is/are true?

1. Items are stored as key, value pairs
2. All keys must be unique
3. All values must be unique
4. The best case for get and set is O(n)
5. Hash tables are ordered by keys
6. A hash table can have keys of only one type
7. A hash table can have values of multiple types

A. 1, 3, 5
B. 1, 4, 6
C. 1, 2, 6
D. 2
E. 3, 4, 7

```
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

```
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

```
int hash(char key) {
    return (int) key;
}
```

Which of the following sequences of insertions would cause the most collisions for a hash table with **four** buckets and assuming expandCapacity is not called during the adds?

A. add('A', 56); add('B', 5); add('C', 65); add('D', 2);
B. add('E', 43); add('F', 7); add('K', 6); add('L', 160);
C. add('M', 58); add('Q', 14); add('U', 20); add('W', 37);
D. add('N', 7); add('R', 24); add('V', 92); add('Z', 100);
E. add('Z', 91); add('R', 604); add('P', 9); add('L', 5);

```
int hash(char key) {
    return (int) key;
}
```

Which of the following sequences of insertions would cause the most collisions for a hash table with **four** buckets and assuming expandCapacity is not called during the adds?

A. add('A', 56); add('B', 5); add('C', 65); add('D', 2);
B. add('E', 43); add('F', 7); add('K', 6); add('L', 160);
C. add('M', 58); add('Q', 14); add('U', 20); add('W', 37);
D. add('N', 7); add('R', 24); add('V', 92); add('Z', 100);
E. add('Z', 91); add('R', 604); add('P', 9); add('L', 5);

```
int hash(String key) {
    return key.length;
}

/*
 ----------
|          |
|          -
|          |
 ----------
|          |
|          - {"greetings" : 6}
|          |
 ----------
|          |
|          - {"hi" : 5}
|          |
 ----------
|          |
|          - {"bye" : 9}
|          |
 ----------
|          |
|          - {"happy week 7" : 3}
|          |
```

After expandCapacity is called, which of the following elements will have a different index in the new array after rehashing?

- {"greetings" : 6}
- {"hi" : 5}
- {"bye" : 9}
- {"happy week 7" : 3}
- {"hello" : 2}