Some (Hash)Maps

Joonho Kim

Announcements

- Homework 3 is due next week
- Exam next Thursday
- 5 scribes please
  - Write names on white board to remember
- Class Observation Next Tuesday

Instructions

- There will be questions on these slides. Please have a clean piece of paper to write your answers. Write your name on the top right corner for our record. At the end of lecture, we will collect these pieces of paper for your participation grade.
- Scribes should get ready to scribe.

Last Time…

- Take 3 min to write down a summary of the following:
  - BST Remove
    - What are the 3 remove conditions and how do we handle each?
  - Tree Traversal
  - Heaps
    - What properties does a binary heap have?

Last Time…

- Take 3 min to write down a summary of the following:
  - BST Remove
    - What are the 3 remove conditions and how do we handle each?
      - No child: parents child pointer is set to null
      - 1 child: parent's child pointer is set to child's child
      - 2 children: our current node's data copies the data of its successor or predecessor.
        - The successor or predecessor is then removed.
  - Tree Traversal
    - Recursive: pre/in/post order traversals
    - Iterative w/ Queue: Level order
  - Heaps
    - What properties does a binary heap have?
      - Complete tree property: Binary tree must be complete
      - Heap property: a node's data must be better than the children's data.

Let’s Imagine…

- Let’s say we’re writing a student GPA directory for Georgia Tech.

Search: “Alex”

Program: “Too many Alex’s at Tech. Sorry”

Let's Imagine...

- Let's say we're writing a student GPA directory for Georgia Tech.
  1. Functionalities:

Search: "Alex"

Program: "Too many Alex's at Tech. Sorry"

Let's Imagine...

Search: "Alex"

Program: "Too many Alex's at Tech. Sorry"

- Let's say we're writing a student GPA directory for Georgia Tech.
  1. Functionalities:
     - Search for a student and the program gives me that student's GPA.
     - Populate the directory by inserting a student and corresponding GPA.
     - Update a student's GPA.
     - Delete a student's GPA.

Let's Imagine...

Search: "Alex"

Program: "Too many Alex's at Tech. Sorry"

- Let's say we're writing a student GPA directory for Georgia Tech.
  1. Functionalities:
     - Search for a student and the program gives me that student's GPA.
     - Populate the directory by inserting a student and corresponding GPA.
     - Update a student's GPA.
     - Delete a student's GPA.
  2. The program will only allow me to perform a function if I uniquely search for a student.

Let's Imagine...

Search: "Alex"

Program: "Too many Alex's at Tech. Sorry"

- Let's say we're writing a student GPA directory for Georgia Tech.
  1. Functionalities:
     - Search for a student and the program gives me that student's GPA.
     - Populate the directory by inserting a student and corresponding GPA.
     - Update a student's GPA.
     - Delete a student's GPA.
  2. The program will only allow me to perform a function if I uniquely search for a student.
     - If I look up the GPA for "Alex", the program can't determine which "Alex"'s GPA it should return.

Let's Imagine...

Search: "Alex"

Program: "Too many Alex's at Tech. Sorry"

- Let's say we're writing a student GPA directory for Georgia Tech.
  1. Functionalities:
     - Search for a student and the program gives me that student's GPA.
     - Populate the directory by inserting a student and corresponding GPA.
     - Update a student's GPA.
     - Delete a student's GPA.
  2. The program will only allow me to perform a function if I uniquely search for a student.
     - If I look up the GPA for "Alex", the program can't determine which "Alex"'s GPA it should return.
  3. I want to be able to look up a student in roughly O(1).

Let’s Imagine Some Unique Info…

- What student information could I use to uniquely identify a student?

Search: “Alex”

Program: “Too many Alex’s at Tech. Sorry”

Let’s Imagine Some Unique Info…

- What student information could I use to uniquely identify a student?
  - Every Georgia Tech student has a unique GTID number.

Search: “Alex”

Program: “Too many Alex’s at Tech. Sorry”

Let’s Imagine Some Unique Info…

Search: “Alex”

Program: “Too many Alex’s at Tech. Sorry”

- What student information could I use to uniquely identify a student?
  - Every Georgia Tech student has a unique GTID number.
  - My program will allow me to search for GTID’s, and with this information I can successfully return a unique student’s GPA!

Let’s Imagine Some Unique Info…

Search: “Alex”

Program: “Too many Alex’s at Tech. Sorry”

- What student information could I use to uniquely identify a student?
  - Every Georgia Tech student has a unique GTID number.
  - My program will allow me to search for GTID’s, and with this information I can successfully return a unique student’s GPA!
    - Joonho’s GTID = 902905XXX

Let’s Imagine Some Unique Info…

Search: “Alex”

Program: “Too many Alex’s at Tech. Sorry”

- What student information could I use to uniquely identify a student?
  - Every Georgia Tech student has a unique GTID number.
  - My program will allow me to search for GTID’s, and with this information I can successfully return a unique student’s GPA!
    - Joonho’s GTID = 902905XXX
  - You could also use other unique information such as SSN, email address, etc.

Let’s Imagine Some Unique Info…

Search: “Alex”

Program: “Too many Alex’s at Tech. Sorry”

- What student information could I use to uniquely identify a student?
  - Every Georgia Tech student has a unique GTID number.
  - My program will allow me to search for GTID’s, and with this information I can successfully return a unique student’s GPA!
    - Joonho’s GTID = 902905XXX
  - You could also use other unique information such as SSN, email address, etc.
    - Joonho’s SSN = yea right

Let’s Imagine Some Unique Info…

Search: “Alex”

Program: “Too many Alex’s at Tech. Sorry”

- What student information could I use to uniquely identify a student?
  - Every Georgia Tech student has a unique GTID number.
  - My program will allow me to search for GTID’s, and with this information I can successfully return a unique student’s GPA!
    - Joonho’s GTID = 902905XXX
  - You could also use other unique information such as SSN, email address, etc.
    - Joonho’s SSN = yea right
- This unique value is called a **key**

Let’s Imagine Some Functions…

- Our program requires us to use this unique info to perform functions:

Search: “Alex”

Program: “Too many Alex’s at Tech. Sorry”

Let's Imagine Some Functions…

Search: "Alex"

Program: "Too many Alex's at Tech. Sorry"

- Our program requires us to use this unique info to perform functions:
  - search(key) – searches for the data of key
  - insert(key, data) – inserts data for key
  - delete(key) – searches and deletes the data for key
  - update(key, data) – call insert(key, data). Handle updating in the insert() method.

Let's Imagine Some Functions...

Search: "Alex"

Program: "Too many Alex's at Tech. Sorry"

- Our program requires us to use this unique info to perform functions:
  - search(key) – searches for the data of key
  - insert(key, data) – inserts data for key
  - delete(key) – searches and deletes the data for key
  - update(key, data) – call insert(key, data). Handle updating in the insert() method.
- key will be the unique data we use. For now, we have these functions declared.

Let's Imagine Some Fast Lookup...

Search: "901901901"

Program: "4.0. Wow!"

- So I've gotten the unique search info to look up. Now how do I achieve O(1) lookup time?

Let's Imagine Some Fast Lookup...

Search: "901901901"

Program: "4.0. Wow!"

- So I've gotten the unique search info to look up. Now how do I achieve O(1) lookup time?
- What's a data structure that gives us O(1) look up time?

Let's Imagine Some Fast Lookup...

Search: "901901901"

Program: "4.0. Wow!"

- So I've gotten the unique search info to look up. Now how do I achieve O(1) lookup time?
- What's a data structure that gives us O(1) look up time?
  - Arrays (and ArrayLists, but for now we'll use Arrays)
  - Given an index i, we can access the i'th element of an array in O(1) and we can store a value at index i in O(1).

Search: “901901901”

Program: “4.0. Wow!”

Let’s Imagine Some Fast Lookup…

- So I’ve gotten the unique search info to look up. Now how do I achieve O(1) lookup time?
- What’s a data structure that gives us O(1) look up time?
  - Arrays (and ArrayLists, but for now we’ll use Arrays)
  - Given an index i, we can access the i’th element of an array in O(1) and we can store a value at index i in O(1).
- Hey, our GTID key is a number, so why don’t we have GTID’s act as indices to our array. The element at index GTID can hold the GPA of that student.

Let's Imagine Some Fast Lookup…

Search: "901901901"

Program: "4.0. Wow!"

- So I've gotten the unique search info to look up. Now how do I achieve O(1) lookup time?
- What's a data structure that gives us O(1) look up time?
  - Arrays (and ArrayLists, but for now we'll use Arrays)
  - Given an index i, we can access the i'th element of an array in O(1) and we can store a value at index i in O(1).
- Hey, our GTID key is a number, so why don't we have GTID's act as indices to our array. The element at index GTID can hold the GPA of that student.
- So, on your paper, create an array large enough to index GTID's
  - At index 901901901, put 4.0

Let's Imagine Some Fast Lookup...

Search: "901901901"

Program: "4.0. Wow!"

- So I've gotten the unique search info to look up. Now how do I achieve O(1) lookup time?
- What's a data structure that gives us O(1) look up time?
  - Arrays (and ArrayLists, but for now we'll use Arrays)
  - Given an index i, we can access the i'th element of an array in O(1) and we can store a value at index i in O(1).
- Hey, our GTID key is a number, so why don't we have GTID's act as indices to our array. The element at index GTID can hold the GPA of that student.
- So, on your paper, create an array large enough to index GTID's
  - At index 901901901, put 4.0
  - Don't do this, you need an array of size ~900,000,000.

Let's Imagine Some More Issues...

- Turns out having an array of size ~900 million is very inefficient.

Search: "901901901"

Program: "4.0. Wow!"

Let's Imagine Some More Issues...

Search: "901901901"

Program: "4.0. Wow!"

- Turns out having an array of size ~900 million is very inefficient.
  - The amount of space our array allocates is too much.

Let's Imagine Some More Issues...

Search: "901901901"

Program: "4.0. Wow!"

- Turns out having an array of size ~900 million is very inefficient.
  - The amount of space our array allocates is too much.
  - Georgia Tech doesn't even have ~900 million graduates, so we'll have a lot of empty spots.

Search: “901901901”

Program: “4.0. Wow!”

Let’s Imagine Some More Issues…

- Turns out having an array of size ~900 million is very inefficient.
  - The amount of space our array allocates is too much.
  - Georgia Tech doesn’t even have ~900 million graduates, so we’ll have a lot of empty spots.
- What can we do?

Search: “901901901”

Program: “4.0. Wow!”

Let’s Imagine Some More Issues…

- Turns out having an array of size ~900 million is very inefficient.
  - The amount of space our array allocates is too much.
  - Georgia Tech doesn’t even have ~900 million graduates, so we’ll have a lot of empty spots.
- What can we do?
  1. Give all students another GTID that uniquely identifies that student.
  2. Change the size of our array.
  3. Find another key that uniquely identifies students.

Let's Imagine Some More Issues...

Search: "901901901"

Program: "4.0. Wow!"

- Turns out having an array of size ~900 million is very inefficient.
  - The amount of space our array allocates is too much.
  - Georgia Tech doesn't even have ~900 million graduates, so we'll have a lot of empty spots.
- What can we do?
  1. ~~Give all students another GTID that uniquely identifies that student.~~
  2. Change the size of our array.
  3. Find another key that uniquely identifies students.

Search: “901901901”

Program: “4.0. Wow!”

Let’s Imagine Some More Issues…

- Turns out having an array of size ~900 million is very inefficient.
  - The amount of space our array allocates is too much.
  - Georgia Tech doesn’t even have ~900 million graduates, so we’ll have a lot of empty spots.
- What can we do?
  1. ~~Give all students another GTID that uniquely identifies that student.~~
  2. Change the size of our array.
  3. Find another key that uniquely identifies students.

Shortening the Array

- Let's shorten our array to a length of 10.

Search: "901901901"

Program: "4.0. Wow!"

Search: “901901901”

Program: “4.0. Wow!”

Shortening the Array

- Let’s shorten our array to a length of 10.
- We cannot use our GTID to index into the array. How can we still use our GTID as an index?

Search: “901901901”

Program: “4.0. Wow!”

Shortening the Array

- Let’s shorten our array to a length of 10.
- We cannot use our GTID to index into the array. How can we still use our GTID key as an index?
  - What if we **modded** the GTID by our array size?

Search: “901901901”

Program: “4.0. Wow!”

Shortening the Array

- Let’s shorten our array to a length of 10.
- We cannot use our GTID to index into the array. How can we still use our GTID key as an index?
  - What if we **modded** the GTID by our array size?
    - Modding will restrict our GTID to a valid index in our array.

Search: “901901901”

Program: “4.0. Wow!”

Shortening the Array

- Let’s shorten our array to a length of 10.
- We cannot use our GTID to index into the array. How can we still use our GTID key as an index?
  - What if we **modded** the GTID by our array size?
    - Modding will restrict our GTID to a valid index in our array.
    - index = 901901901 % arr.length *// index = 1*

Search: “901901901”

Program: “4.0. Wow!”

Shortening the Array

- Let’s shorten our array to a length of 10.
- We cannot use our GTID to index into the array. How can we still use our GTID key as an index?
  - What if we **modded** the GTID by our array size?
    - Modding will restrict our GTID to a valid index in our array.
    - index = 901901901 % arr.length *// index = 1*
- Now we can do our operations (search/insert/delete/update) given this new index modding idea.

Array Index Visualization

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Array Index Visualization

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

insert(901901901, 4.0)

Array Index Visualization

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

insert(901901901, 4.0)
index = 901901901 % arr.length

Array Index Visualization

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

insert(901901901, 4.0)
index = 1

Array Index Visualization

0	
1	4.0
2	
3	
4	
5	
6	
7	
8	
9	

insert(901901901, 4.0)
index = 1
arr[index] = 4.0

Array Index Visualization

0	
1	4.0
2	
3	
4	
5	
6	
7	
8	
9	

insert(901901901, 4.0)
index = 1
arr[index] = 4.0

Perform the following:
insert(902905233, 3.8)
insert(901900008, 3.3)

Array Index Visualization

0	
1	4.0
2	
3	
4	
5	
6	
7	
8	
9	

insert(901901901, 4.0)
index = 1
arr[index] = 4.0
Perform the following:
insert(902905233, 3.8)
index = 902905233 % arr.length
insert(901900008, 3.3)

Array Index Visualization

0	
1	4.0
2	
3	
4	
5	
6	
7	
8	
9	

insert(901901901, 4.0)
index = 1
arr[index] = 4.0
Perform the following:
insert(902905233, 3.8)
index = 3
insert(901900008, 3.3)

Array Index Visualization

0	
1	4.0
2	
3	3.8
4	
5	
6	
7	
8	
9	

insert(901901901, 4.0)
index = 1
arr[index] = 4.0
Perform the following:
insert(902905233, 3.8)
index = 3
arr[index] = 3.8
insert(901900008, 3.3)

Array Index Visualization

0	
1	4.0
2	
3	3.8
4	
5	
6	
7	
8	
9	

insert(901901901, 4.0)
index = 1
arr[index] = 4.0
Perform the following:
insert(902905233, 3.8)
index = 3
arr[index] = 3.8
insert(901900008, 3.3)
index = 901900008 % arr.length

Array Index Visualization

0	
1	4.0
2	
3	3.8
4	
5	
6	
7	
8	
9	

insert(901901901, 4.0)
index = 1
arr[index] = 4.0
Perform the following:
insert(902905233, 3.8)
index = 3
arr[index] = 3.8
insert(901900008, 3.3)
index = 8

Array Index Visualization

0	
1	4.0
2	
3	3.8
4	
5	
6	
7	
8	3.3
9	

insert(901901901, 4.0)
index = 1
arr[index] = 4.0
Perform the following:
insert(902905233, 3.8)
index = 3
arr[index] = 3.8
insert(901900008, 3.3)
index = 8
arr[index] = 3.3

Not using GTID

Search: “901901901”

Program: “4.0. Wow!”

- What if students did not have a GTID? What other unique key could we use?

Search: “901901901”

Program: “4.0. Wow!”

Not using GTID

- What if students did not have a GTID? What other unique key could we use?
  - We could use the GT login. *(jkim866)*

Not using GTID

Search: “901901901”

Program: “4.0. Wow!”

- What if students did not have a GTID? What other unique key could we use?
  - We could use the GT login. *(jkim866)*
    - With our array of size 10, how do we extract an index out of this string?

Search: “901901901”

Program: “4.0. Wow!”

Not using GTID

- What if students did not have a GTID? What other unique key could we use?
  - We could use the GT login. *(jkim866)*
    - With our array of size 10, how do we extract an index out of this string?
      - We can come up with some function that converts strings into an integer.
      - Let’s come up with a **hash function**.

Hash Function

Search: “901901901”

Program: “4.0. Wow!”

- A hash function is a function that converts its input into an integer value. The output value is called a hash value.

Search: “901901901”

Program: “4.0. Wow!”

Hash Function

- A hash function is a function that converts its input into an integer value. The output value is called a hash value.
  - ```
    int hashFunction(String s) {
      int hashValue = 0;
      for (int i = 0; i < s.length(); ++i)
        hashValue += (int) s.charAt(i);
      return hashValue;
    }
    ```


    ```

Search: “901901901”

Program: “4.0. Wow!”

Hash Function

- A hash function is a function that converts its input into an integer value. The output value is called a hash value.
  - ```
    int hashFunction(String s) {
      int hashValue = 0;
      for (int i = 0; i < s.length(); ++i)
        hashValue += (int) s.charAt(i);
      return hashValue;
    }
    ```
- A hash function is **deterministic:** should always return the same hash value for the same input.
  - A hash function should not be dependent on randomness (time, probability, etc.)


    ```

Search: “901901901”

Program: “4.0. Wow!”

Hash Function

- A hash function is a function that converts its input into an integer value. The output value is called a hash value.
  - ```
    int hashFunction(String s) {
      int hashValue = 0;
      for (int i = 0; i < s.length(); ++i)
        hashValue += (int) s.charAt(i);
      return hashValue;
    }
    ```
- A hash function is **deterministic:** should always return the same hash value for the same input.
  - A hash function should not be dependent on randomness (time, probability, etc.)
- In Java, all objects have a hashCode() method that digitizes the object returning an integer representation of the object.
  - You can override this method to create your own hash function.


    ```

Search: “901901901”

Program: “4.0. Wow!”

Not using GTID (again)

- What if students did not have a GTID? What other unique key could we use?
  - We could use the GT login. *(jkim866)*
    - With our array of size 10, how do we extract an index out of this string?
      - We can come up with some function that converts strings into an integer.
      - Let’s come up with a **hash function**.

Search: “901901901”

Program: “4.0. Wow!”

Not using GTID (again)

- What if students did not have a GTID? What other unique key could we use?
  - We could use the GT login. *(jkim866)*
    - With our array of size 10, how do we extract an index out of this string?
      - We can come up with some function that converts strings into an integer.
      - Let’s come up with a **hash function**.
  - Our hash function will take in a string key as input and return the sum of character ascii values.

Search: “901901901”

Program: “4.0. Wow!”

Not using GTID (again)

- What if students did not have a GTID? What other unique key could we use?
  - We could use the GT login. *(jkim866)*
    - With our array of size 10, how do we extract an index out of this string?
      - We can come up with some function that converts strings into an integer.
      - Let’s come up with a **hash function**.
  - Our hash function will take in a string key as input and return the sum of character ascii values.
    - ```
      int hashFunction(String s) {
        int hashValue = 0;
        for (int i = 0; i < s.length(); ++i)
          hashValue += (int) s.charAt(i);
        return hashValue;
      }
      ```


      ```

Search: “901901901”

Program: “4.0. Wow!”

Not using GTID (again)

- What if students did not have a GTID? What other unique key could we use?
  - We could use the GT login. *(jkim866)*
    - With our array of size 10, how do we extract an index out of this string?
      - We can come up with some function that converts strings into an integer.
      - Let’s come up with a **hash function**.
  - Our hash function will take in a string key as input and return the sum of character ascii values.

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

- What is the result of hashFunction(“jkim866”)?

104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o

048	0
049	1
050	2
051	3
052	4
053	5
054	6
055	7
056	8
057	9

Search: “901901901”

Program: “4.0. Wow!”

Not using GTID (again)

- What if students did not have a GTID? What other unique key could we use?
  - We could use the GT login. *(jkim866)*
    - With our array of size 10, how do we extract an index out of this string?
      - We can come up with some function that converts strings into an integer.
      - Let’s come up with a **hash function**.
  - Our hash function will take in a string key as input and return the sum of character ascii values.

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

  - What is the result of hashFunction(“jkim866”)?
    - hashValue = 106 + 107 + 105 + 109 + 56 + 54 + 54 = **591**

104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o

048	0
049	1
050	2
051	3
052	4
053	5
054	6
055	7
056	8
057	9

Search: “901901901”

Program: “4.0. Wow!”

Not using GTID (again)

- What if students did not have a GTID? What other unique key could we use?
  - We could use the GT login. *(jkim866)*
    - With our array of size 10, how do we extract an index out of this string?
      - We can come up with some function that converts strings into an integer.
      - Let’s come up with a **hash function**.
  - Our hash function will take in a string key as input and return the sum of character ascii values.

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

  - What is the result of hashFunction(“jkim866”)?
    - hashValue = 106 + 107 + 105 + 109 + 56 + 54 + 54 = **591**
  - After we get our hashValue, we still need to mod it.
    - index = hashFunction(“jkim866”) % arr.length; // *index = 1*

104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o

048	0
049	1
050	2
051	3
052	4
053	5
054	6
055	7
056	8
057	9

Array Index Visualization with String Key

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Array Index Visualization with String Key

insert(“jkim866”, 4.0)

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Array Index Visualization with String Key

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

insert(“jkim866”, 4.0)
index = hashFunction(“jkim866”) % arr.length

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

insert(“jkim866”, 4.0)
index = 591 % arr.length

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

insert(“jkim866”, 4.0)
index = 1

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	
5	
6	
7	
8	
9	

insert(“jkim866”, 4.0)
index = 1
arr[index] = 4.0

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	
5	
6	
7	
8	
9	

insert(“jkim866”, 4.0)
  index = 1
  arr[index] = 4.0

Perform the following:
insert(“aa22”, 3.8)
insert(“bb22”, 3.3)

‘a’ = 97
‘b’ = 98
‘1’ = 49
‘2’ = 50

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	
5	
6	
7	
8	
9	

insert(“jkim866”, 4.0)
  index = 1
  arr[index] = 4.0
Perform the following:
insert(“aa22”, 3.8)
  index = hashFunction(“aa22”) % arr.length
insert(“bb22”, 3.3)

‘a’ = 97
‘b’ = 98
‘1’ = 49
‘2’ = 50

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	
5	
6	
7	
8	
9	

insert(“jkim866”, 4.0)
  index = 1
  arr[index] = 4.0
Perform the following:
insert(“aa22”, 3.8)
  index = (97 + 97 + 50 + 50) % arr.length
insert(“bb22”, 3.3)

‘a’ = 97
‘b’ = 98
‘1’ = 49
‘2’ = 50

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	
5	
6	
7	
8	
9	

insert(“jkim866”, 4.0)
  index = 1
  arr[index] = 4.0
Perform the following:
insert(“aa22”, 3.8)
  index = 294 % arr.length
insert(“bb22”, 3.3)

‘a’ = 97
‘b’ = 98
‘1’ = 49
‘2’ = 50

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	
5	
6	
7	
8	
9	

insert(“jkim866”, 4.0)
  index = 1
  arr[index] = 4.0
Perform the following:
insert(“aa22”, 3.8)
  index = 4
insert(“bb22”, 3.3)

‘a’ = 97
‘b’ = 98
‘1’ = 49
‘2’ = 50

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	
7	
8	
9	

insert(“jkim866”, 4.0)
  index = 1
  arr[index] = 4.0
Perform the following:
insert(“aa22”, 3.8)
  index = 4
  arr[index] = 3.8
insert(“bb22”, 3.3)

‘a’ = 97
‘b’ = 98
‘1’ = 49
‘2’ = 50

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	
7	
8	
9	

insert(“jkim866”, 4.0)
  index = 1
  arr[index] = 4.0
Perform the following:
insert(“aa22”, 3.8)
  index = 4
  arr[index] = 3.8
insert(“bb22”, 3.3)
  index = (98 + 98 + 50 + 50) % arr.length

‘a’ = 97
‘b’ = 98
‘1’ = 49
‘2’ = 50

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	
7	
8	
9	

insert(“jkim866”, 4.0)
  index = 1
  arr[index] = 4.0
Perform the following:
insert(“aa22”, 3.8)
  index = 4
  arr[index] = 3.8
insert(“bb22”, 3.3)
  index = 296 % arr.length

‘a’ = 97
‘b’ = 98
‘1’ = 49
‘2’ = 50

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	
7	
8	
9	

insert(“jkim866”, 4.0)
  index = 1
  arr[index] = 4.0
Perform the following:
insert(“aa22”, 3.8)
  index = 4
  arr[index] = 3.8
insert(“bb22”, 3.3)
  index = 6

‘a’ = 97
‘b’ = 98
‘1’ = 49
‘2’ = 50

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	3.3
7	
8	
9	

insert(“jkim866”, 4.0)
  index = 1
  arr[index] = 4.0
Perform the following:
insert(“aa22”, 3.8)
  index = 4
  arr[index] = 3.8
insert(“bb22”, 3.3)
  index = 6
  arr[index] = 3.3

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

‘a’ = 97
‘b’ = 98
‘1’ = 49
‘2’ = 50

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	3.3
7	
8	
9	

insert(“jkim866”, 4.0)
  index = 1
  arr[index] = 4.0
Perform the following:
insert(“aa22”, 3.8)
  index = 4
  arr[index] = 3.8
insert(“bb22”, 3.3)
  index = 6
  arr[index] = 3.3

‘a’ = 97
‘b’ = 98
‘1’ = 49
‘2’ = 50

What if we hash something that gives us an occupied index?

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	3.3
7	
8	
9	

insert(“jkim866”, 4.0)
  index = 1
  arr[index] = 4.0
Perform the following:
insert(“aa22”, 3.8)
  index = 4
  arr[index] = 3.8
insert(“bb22”, 3.3)
  index = 6
  arr[index] = 3.3

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

‘a’ = 97
‘b’ = 98
‘1’ = 49
‘2’ = 50

What if we hash something that gives us an occupied index?

We’ll handle this later with “Collision Handling”

Fulfilling our Program Obligations

- Our functionalities (search, insert, update, delete) all can me implemented using:

Fulfilling our Program Obligations

- Our functionalities (search, insert, update, delete) all can me implemented using:
  - index = hashFunction(GT_login) % arr.length

Fulfilling our Program Obligations

- Our functionalities (search, insert, update, delete) all can me implemented using:
  - index = hashFunction(GT_login) % arr.length
  - Search: check if arr[index] has a value
  - Insert: arr[index] = GPA
  - Update: arr[index] = GPA
  - Delete: arr[index] = NULL

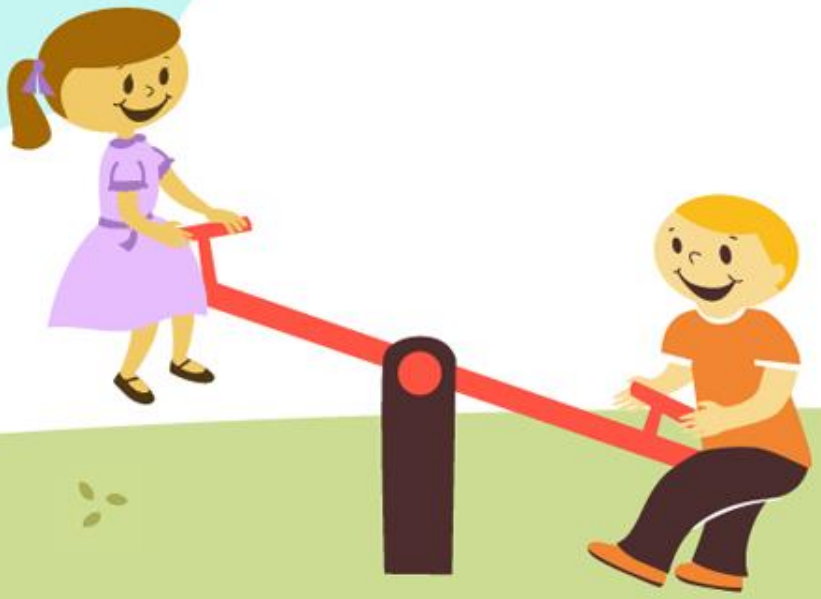
Fulfilling our Program Obligations

- Our functionalities (search, insert, update, delete) all can me implemented using:
  - index = hashFunction(GT_login) % arr.length
  - Search: check if arr[index] has a value
  - Insert: arr[index] = GPA
  - Update: arr[index] = GPA
  - Delete: arr[index] = NULL
- We have a unique association between our GT_Login and GPA

Fulfilling our Program Obligations

- Our functionalities (search, insert, update, delete) all can me implemented using:
  - index = hashFunction(GT_login) % arr.length
  - Search: check if arr[index] has a value
  - Insert: arr[index] = GPA
  - Update: arr[index] = GPA
  - Delete: arr[index] = NULL
- We have a unique association between our GT_Login and GPA
- We achieve O(1) operations

Map ADT



Map ADT

- All of this gives us the functionality of a Map ADT.

Map ADT

- All of this gives us the functionality of a Map ADT.
- A map models a searchable collection of key-value **entries**. (key = GT login, value = GPA).

Map ADT

- All of this gives us the functionality of a Map ADT.
- A map models a searchable collection of key-value **entries**. (key = GT login, value = GPA).
- For each key, there is a single value associated with it. Mathematically speaking, this is a mapping function from key to value.

Map ADT

- All of this gives us the functionality of a Map ADT.
- A map models a searchable collection of key-value **entries**. (key = GT login, value = GPA).
- For each key, there is a single value associated with it. Mathematically speaking, this is a mapping function from key to value.
  - Thus, picking a unique key is important to have a good mapping.

Map ADT

- All of this gives us the functionality of a Map ADT.
- A map models a searchable collection of key-value **entries**. (key = GT login, value = GPA).
- For each key, there is a single value associated with it. Mathematically speaking, this is a mapping function from key to value.
  - Thus, picking a unique key is important to have a good mapping.
  - However, multiple keys can have the same value. (Many students can have the same GPA)

Map ADT Operations

- The Map ADT has the following operations:
  - **get**(key): returns the value of entry (key, value), or null if such entry doesn't exist..
  - **put**(key, value): inserts entry (key, value) into the map. If key doesn't exist, return null. Else, entry (key, value) is not inserted, but key's old value is overridden and key's old value is returned.
  - **remove**(key): removes entry (key, value) and returns value, or null if such entry doesn't exist.
  - **size**() / **isEmpty**()
  - entrySet(): returns a set of all entries in the map
  - keyset(): returns a set of all keys in the map
  - values(): returns a list of all values in the map.

Hash Map and Tables

- A hash map (table) is a data structure that implements map using a backing array to store values and a hash function to hash keys to an index in the array.

Hash Map and Tables

- A hash map (table) is a data structure that implements map using a backing array to store values and a hash function to hash keys to an index in the array.
- Because we're accessing elements in an array, our complexity should ideally give us O(1) search.

Hash Map and Tables

- A hash map (table) is a data structure that implements map using a backing array to store values and a hash function to hash keys to an index in the array.
- Because we're accessing elements in an array, our complexity should ideally give us O(1) search.
- In Java, there is both Hash Map and Hash Table
  - Both give similar behaviors, but there are some small differences I won't go over.

Collision Handling

Collisions

- Using our previous GPA example, what happens if we call put(“red8”, 3.4) on our previous data structure?

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	3.3
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	3.3
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
  index = hashFunction(“red8”) % arr.length

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	3.3
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
  index = 371 % arr.length

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	3.3
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
  index = 1

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	3.4
2	
3	
4	3.8
5	
6	3.3
7	
8	
9	

4.0????

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
  index = 1
  arr[index] = 3.4

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Collisions

- Using our previous GPA example, what happens if we call put(“red8”, 3.4) on our previous data structure?

Collisions

- Using our previous GPA example, what happens if we call put(“red8”, 3.4) on our previous data structure?
  - Calling put(“red8”, 3.4) ends up overriding the old data at that index.

Collisions

- Using our previous GPA example, what happens if we call put(“red8”, 3.4) on our previous data structure?
  - Calling put(“red8”, 3.4) ends up overriding the old data at that index.
- There is 2 major issues here:

Collisions

- Using our previous GPA example, what happens if we call put(“red8”, 3.4) on our previous data structure?
  - Calling put(“red8”, 3.4) ends up overriding the old data at that index.
- There is 2 major issues here:
  - put(“red8”, 3.4) should not have overridden jkim866’s 4.0.

Collisions

- Using our previous GPA example, what happens if we call put(“red8”, 3.4) on our previous data structure?
  - Calling put(“red8”, 3.4) ends up overriding the old data at that index.
- There is 2 major issues here:
  - put(“red8”, 3.4) should not have overridden jkim866’s 4.0.
    - put(“jkim866”, number) should be the only way to override the 4.0.

Collisions

- Using our previous GPA example, what happens if we call put(“red8”, 3.4) on our previous data structure?
  - Calling put(“red8”, 3.4) ends up overriding the old data at that index.
- There is 2 major issues here:
  - put(“red8”, 3.4) should not have overridden jkim866’s 4.0.
    - put(“jkim866”, number) should be the only way to override the 4.0.
  - put(“red8”, 3.4) should have inserted 3.4 into our array somewhere else.

Collisions

- Using our previous GPA example, what happens if we call put(“red8”, 3.4) on our previous data structure?
  - Calling put(“red8”, 3.4) ends up overriding the old data at that index.
- There is 2 major issues here:
  - put(“red8”, 3.4) should not have overridden jkim866’s 4.0.
    - put(“jkim866”, number) should be the only way to override the 4.0.
  - put(“red8”, 3.4) should have inserted 3.4 into our array somewhere else.
- Let’s fix these.

Maintaining Entries (Key and Value)

- When we called put(“red8”, 3.4) and it ends up indexing to an occupied cell from put(“jkim866, 4.0), we must check if our current key is the same key that previously put 4.0 in the array.

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	3.3
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	3.3
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
  index = hashFunction(“red8”) % arr.length

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	3.3
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
  index = 371 % arr.length

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	3.3
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
  index = 1

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	3.3
7	
8	
9	

3.4

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
  index = 1
  arr[index] = 3.4

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	3.3
7	
8	
9	

Did “red8” put 4.0 there before?

3.4

```
put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
   index = 1
   arr[index]  = 3.4
```

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Maintaining Entries (Key and Value)

- When we called put(“red8”, 3.4) and it ends up indexing to an occupied cell from put(“jkim866, 4.0), we must check if our current key is the same key that previously put 4.0 in the array.

Maintaining Entries (Key and Value)

- When we called put(“red8”, 3.4) and it ends up indexing to an occupied cell from put(“jkim866, 4.0), we must check if our current key is the same key that previously put 4.0 in the array.
- We can solve this by storing both the key and value within the array cell. This pair is called an **Entry**.

Maintaining Entries (Key and Value)

- When we called put(“red8”, 3.4) and it ends up indexing to an occupied cell from put(“jkim866, 4.0), we must check if our current key is the same key that previously put 4.0 in the array.
- We can solve this by storing both the key and value within the array cell. This pair is called an **Entry**.
  - This means our array would be an array of Entries.

Maintaining Entries (Key and Value)

- When we called put(“red8”, 3.4) and it ends up indexing to an occupied cell from put(“jkim866, 4.0), we must check if our current key is the same key that previously put 4.0 in the array.
- We can solve this by storing both the key and value within the array cell. This pair is called an **Entry**.
  - This means our array would be an array of Entries.
- When we access an index, we check whether our parameter key equals the entry key. If it does:

Maintaining Entries (Key and Value)

- When we called put(“red8”, 3.4) and it ends up indexing to an occupied cell from put(“jkim866, 4.0), we must check if our current key is the same key that previously put 4.0 in the array.
- We can solve this by storing both the key and value within the array cell. This pair is called an **Entry**.
  - This means our array would be an array of Entries.
- When we access an index, we check whether our parameter key equals the entry key. If it does:
  - search(key) will return arr[index]
  - put(key, value) will replace the entry’s old value
  - remove(key) will remove the entry

Array Index Visualization with String Key

0	
1	4.0
2	
3	
4	3.8
5	
6	3.3
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	(“jkim866”, 4.0)
2	
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	(“jkim866”, 4.0)
2	
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	

(“red8”, 3.4)

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
  index = 1
  check if arr[index].key = “red8”

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Array Index Visualization with String Key

0	
1	(“jkim866”, 4.0)
2	
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	

Okay, this index is taken by jkim866. Now what?

(“red8”, 3.4)

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
  index = 1
  check if arr[index].key = “red8”

```
int hashFunction(String s) {
  int hashValue = 0;
  for (int i = 0; i < s.length(); ++i)
    hashValue += (int) s.charAt(i);
  return hashValue;
}
```

Last Time…

- Take 3 min to write down a summary of the following:
  - Maps and Hash Map
  - Hashing/Hash Functions
    - How are they incorporated into HashMaps?
  - External Chaining
  - Load Factor

Collision Handling: External Chaining

- When we have two keys that hash to the same index, we have a **collision**

Collision Handling: External Chaining

- When we have two keys that hash to the same index, we have a **collision**
  - One way to handle this is to chain our entries together like a linked list at the index they hashed to.

Collision Handling: External Chaining

- When we have two keys that hash to the same index, we have a **collision**
  - One way to handle this is to chain our entries together like a linked list at the index they hashed to.
  - Our array would not be an array of Entries but an array of Singly Linked Lists of Entries.
    - LinkedList<Entry>[] arr;

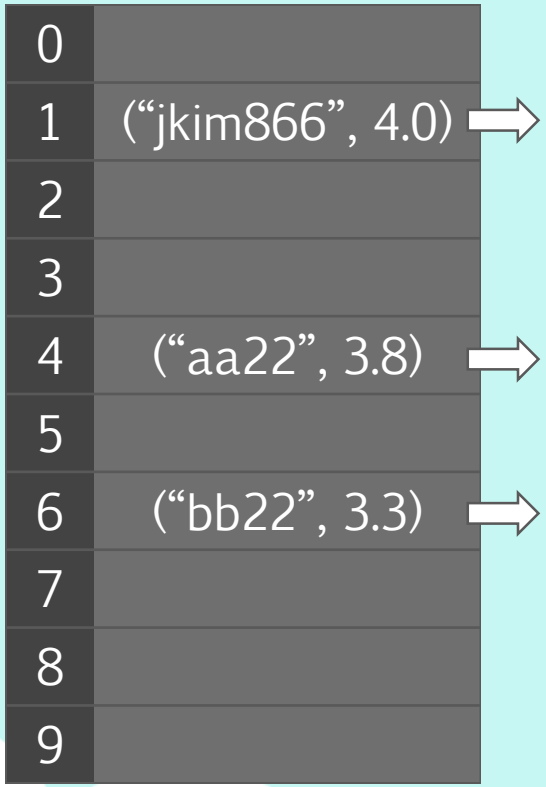
External Chaining Visualization

0	
1	(“jkim866”, 4.0)
2	
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	

(“red8”, 3.4)

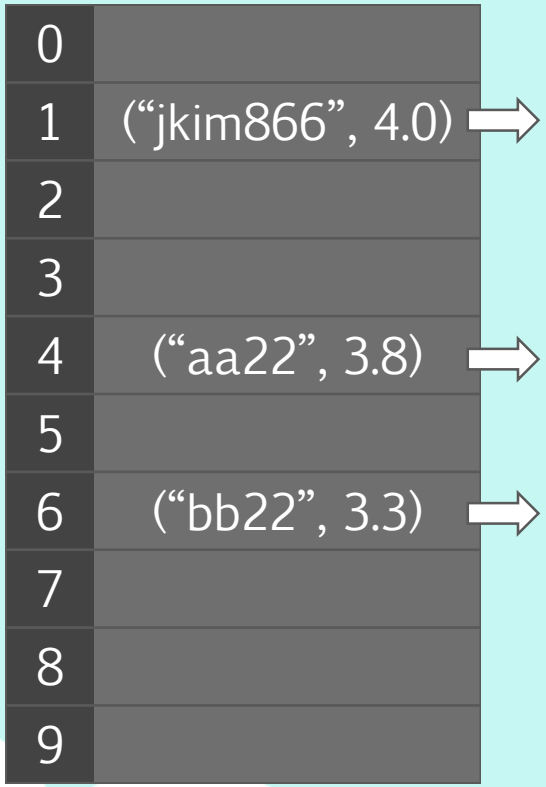
put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
    index = 1
    check if arr[index].key = “red8”

External Chaining Visualization



put("jkim866", 4.0)
put("aa22", 3.8)
put("bb22", 3.3)
put("red8", 3.4)
  index = 1
  check if arr[index].key = "red8"

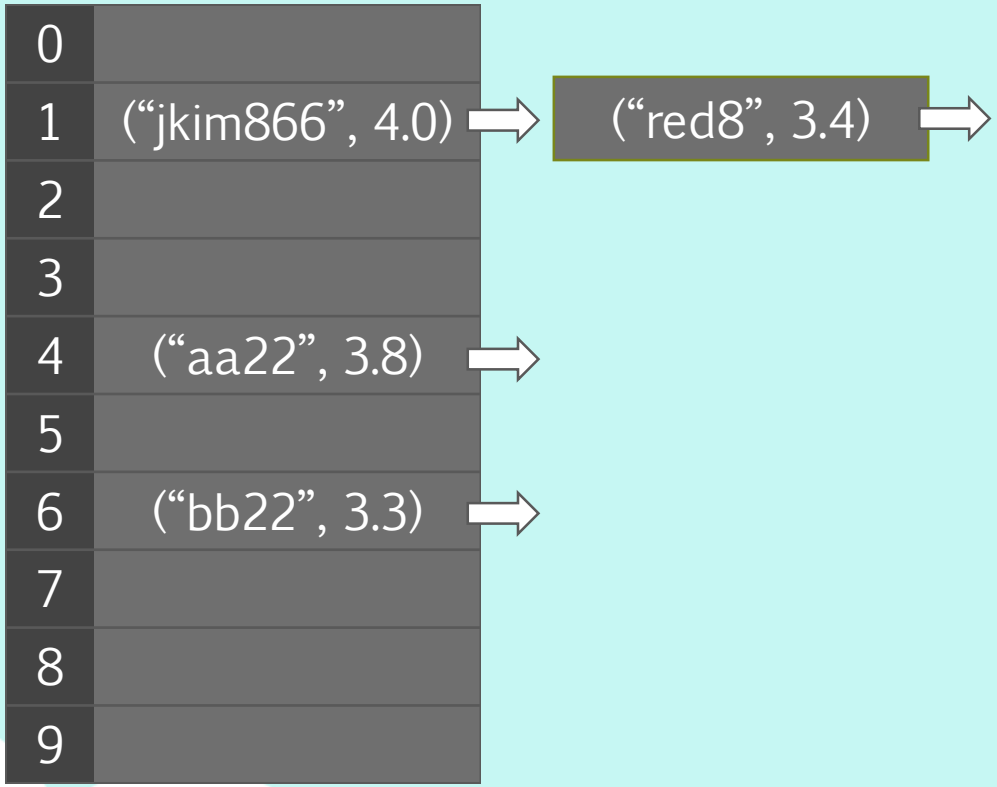
External Chaining Visualization



("red8", 3.4)

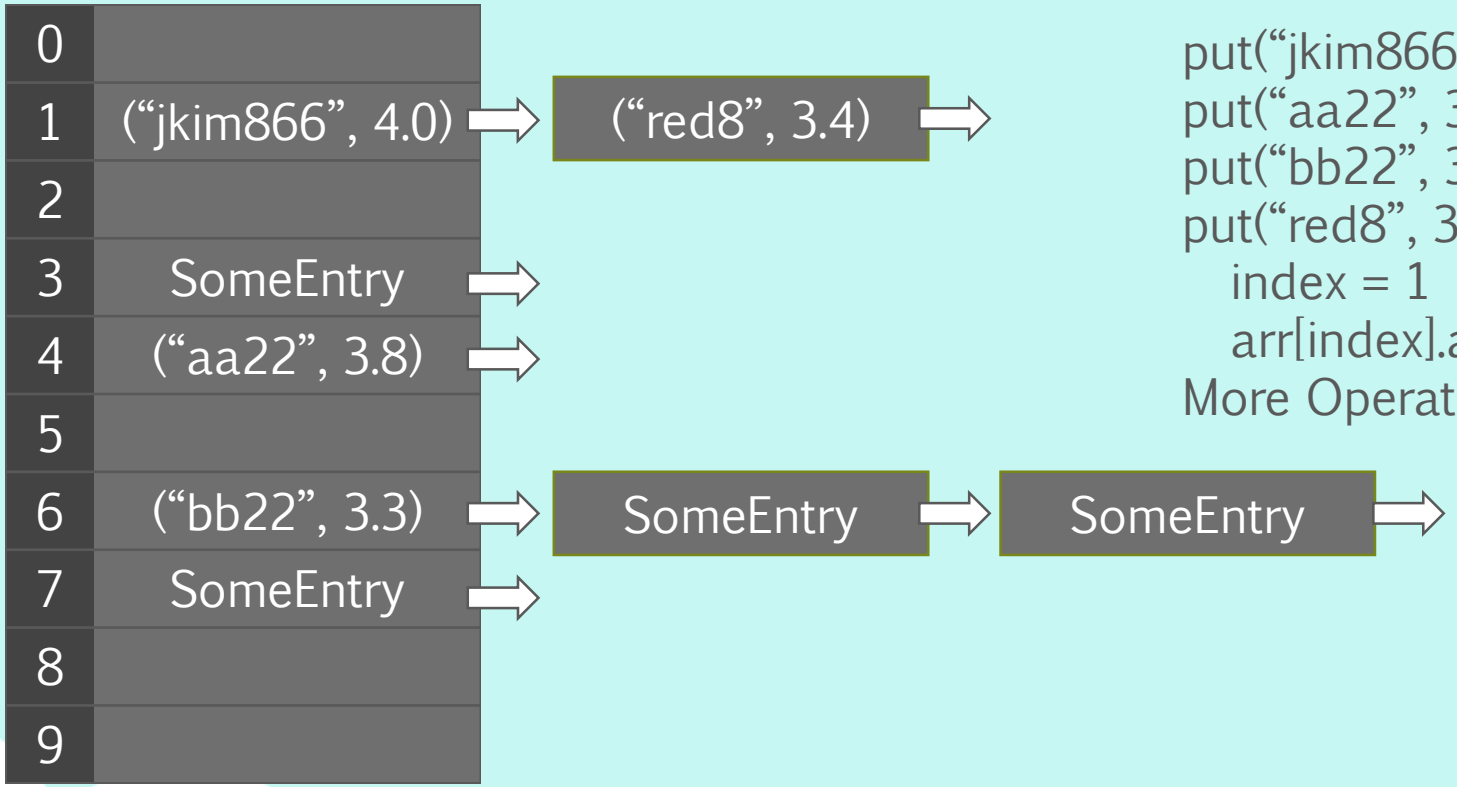
```
put("jkim866", 4.0)
put("aa22", 3.8)
put("bb22", 3.3)
put("red8", 3.4)
    index = 1
    arr[index].addToBack(Entry("red8", 3.4))
```

External Chaining Visualization




```
put("jkim866", 4.0)
put("aa22", 3.8)
put("bb22", 3.3)
put("red8", 3.4)
    index = 1
    arr[index].addToBack(Entry("red8", 3.4))
```

External Chaining Visualization



put("jkim866", 4.0)
put("aa22", 3.8)
put("bb22", 3.3)
put("red8", 3.4)
    index = 1
    arr[index].addToBack(Entry("red8", 3.4))
More Operations.

Collision Handling: External Chaining

- When we have two keys that hash to the same index, we have a **collision**
  - One way to handle this is to chain our entries together like a linked list at the index they hashed to.
  - Our array would not be an array of Entries but an array of Singly Linked Lists of Entries.
    - LinkedList<Entry>[] arr;

Collision Handling: External Chaining

- When we have two keys that hash to the same index, we have a **collision**
  - One way to handle this is to chain our entries together like a linked list at the index they hashed to.
  - Our array would not be an array of Entries but an array of Singly Linked Lists of Entries.
    - LinkedList<Entry>[] arr;
  - If we want to add more to the hash table, then we would hash our key and add to the linked list at the index.

Collision Handling: External Chaining

- When we have two keys that hash to the same index, we have a **collision**
  - One way to handle this is to chain our entries together like a linked list at the index they hashed to.
  - Our array would not be an array of Entries but an array of Singly Linked Lists of Entries.
    - LinkedList<Entry>[] arr;
  - If we want to add more to the hash table, then we would hash our key and add to the linked list at the index.
  - Searching and removing are the same, but we act on the linked list at the correct index.

Resizing

- What if our hash table has a bad hash function where everything hashes to index 1?

Resizing

- What if our hash table has a bad hash function where everything hashes to index 1?
  - We'd end up with a long linked list at index 1.

Resizing

- What if our hash table has a bad hash function where everything hashes to index 1?
  - We'd end up with a long linked list at index 1.
  - We can fix this by changing our hash function.
    - A good hash function will attempt to distribute our entries at different indices.

Resizing

- What if our hash table has a bad hash function where everything hashes to index 1?
  - We'd end up with a long linked list at index 1.
  - We can fix this by changing our hash function.
    - A good hash function will attempt to distribute our entries at different indices.
- What if our hash table has a good hash function but our array gets full?

Resizing

- What if our hash table has a bad hash function where everything hashes to index 1?
  - We'd end up with a long linked list at index 1.
  - We can fix this by changing our hash function.
    - A good hash function will attempt to distribute our entries at different indices.
- What if our hash table has a good hash function but our array gets full?
  - The more our hash table fills up, the more chance we have for a collosion

Resizing

- What if our hash table has a bad hash function where everything hashes to index 1?
  - We'd end up with a long linked list at index 1.
  - We can fix this by changing our hash function.
    - A good hash function will attempt to distribute our entries at different indices.
- What if our hash table has a good hash function but our array gets full?
  - The more our hash table fills up, the more chance we have for a collosion
  - Like any fixed size array, we can resize our backing array.

Resizing

- What if our hash table has a bad hash function where everything hashes to index 1?
  - We'd end up with a long linked list at index 1.
  - We can fix this by changing our hash function.
    - A good hash function will attempt to distribute our entries at different indices.
- What if our hash table has a good hash function but our array gets full?
  - The more our hash table fills up, the more chance we have for a collosion
  - Like any fixed size array, we can resize our backing array.
    - But when do we resize?

Resizing

- What if our hash table has a bad hash function where everything hashes to index 1?
  - We'd end up with a long linked list at index 1.
  - We can fix this by changing our hash function.
    - A good hash function will attempt to distribute our entries at different indices.
- What if our hash table has a good hash function but our array gets full?
  - The more our hash table fills up, the more chance we have for a collosion
  - Like any fixed size array, we can resize our backing array.
    - But when do we resize?
      - We use a **Load Factor**

Load Factor

- Our hash table will maintain a ratio called a **Load Factor**

Load Factor

- Our hash table will maintain a ratio called a **Load Factor**
  - Load factor is calculated = **size/arr.length**

Load Factor

- Our hash table will maintain a ratio called a **Load Factor**
  - Load factor is calculated = **size/arr.length**
  - This presents a ratio of how filled up our backing array is.

Load Factor

- Our hash table will maintain a ratio called a **Load Factor**
  - Load factor is calculated = **size/arr.length**
  - This presents a ratio of how filled up our backing array is.
    - If our load factor is higher, we have a higher possibility of having a collision

Load Factor

- Our hash table will maintain a ratio called a **Load Factor**
  - Load factor is calculated = **size/arr.length**
  - This presents a ratio of how filled up our backing array is.
    - If our load factor is higher, we have a higher possibility of having a collision
- We have a **Max Load Factor** which determines whether we resize our backing array or not.

Load Factor

- Our hash table will maintain a ratio called a **Load Factor**
  - Load factor is calculated = **size/arr.length**
  - This presents a ratio of how filled up our backing array is.
    - If our load factor is higher, we have a higher possibility of having a collision
- We have a **Max Load Factor** which determines whether we resize our backing array or not.
  - Example, MaxLoadFactor = 0.7. If our load factor goes above 0.7, then we would resize our backing array.

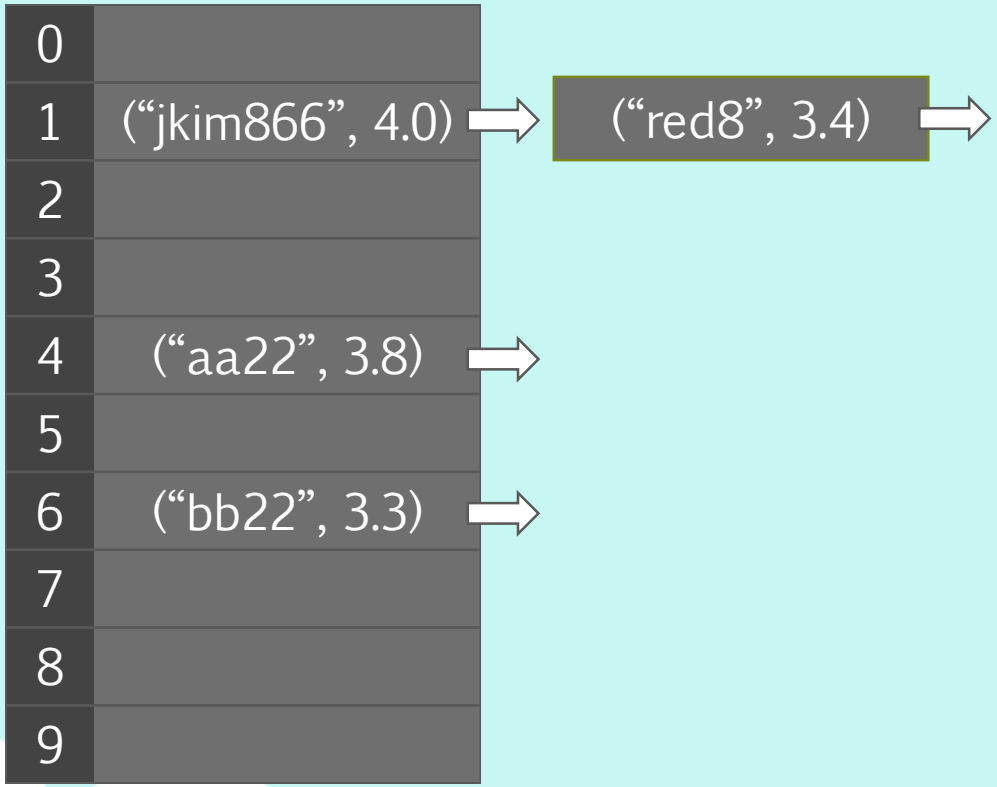
Load Factor

- Our hash table will maintain a ratio called a **Load Factor**
  - Load factor is calculated = **size/arr.length**
  - This presents a ratio of how filled up our backing array is.
    - If our load factor is higher, we have a higher possibility of having a collision
- We have a **Max Load Factor** which determines whether we resize our backing array or not.
  - Example, MaxLoadFactor = 0.7. If our load factor goes above 0.7, then we would resize our backing array.
  - When we move things from our old array to our new array, we call put() on every entry.

Load Factor

- Our hash table will maintain a ratio called a **Load Factor**
  - Load factor is calculated = **size/arr.length**
  - This presents a ratio of how filled up our backing array is.
    - If our load factor is higher, we have a higher possibility of having a collision
- We have a **Max Load Factor** which determines whether we resize our backing array or not.
  - Example, MaxLoadFactor = 0.7. If our load factor goes above 0.7, then we would resize our backing array.
  - When we move things from our old array to our new array, we call put() on every entry.
    - The backing array length has changed, so our hashValue % arr.length could possibly change.

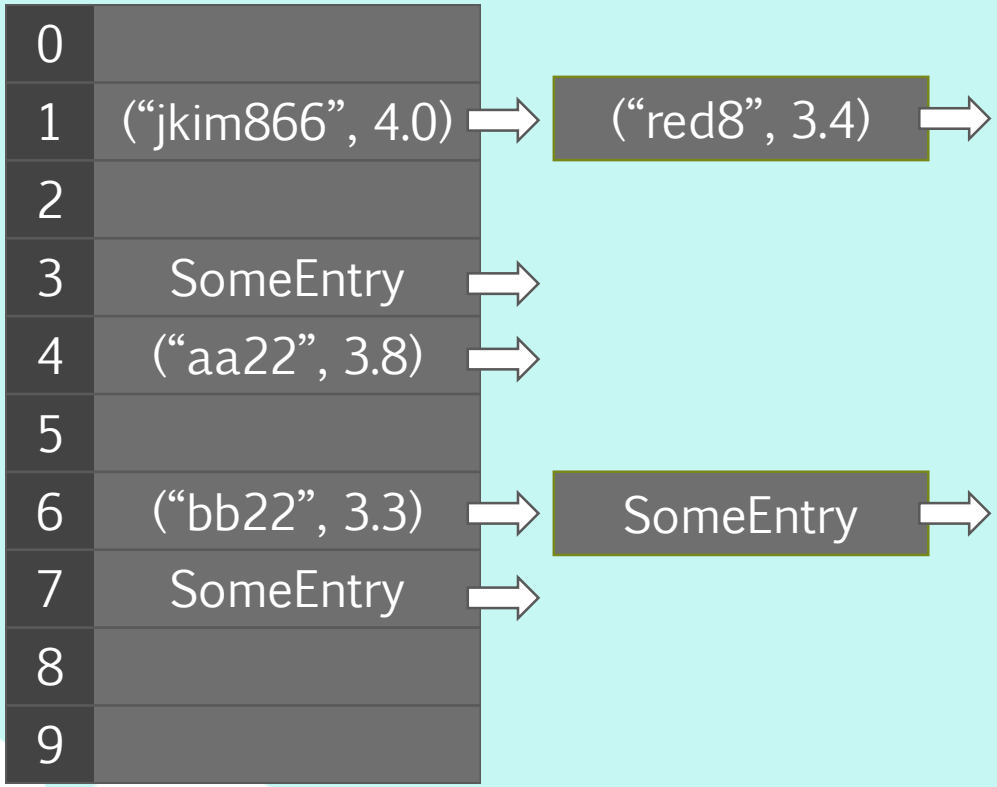
Resize Visualization



put("jkim866", 4.0)
put("aa22", 3.8)
put("bb22", 3.3)
put("red8", 3.4)

loadFactor = 0.4
MaxLoadFactor = 0.7

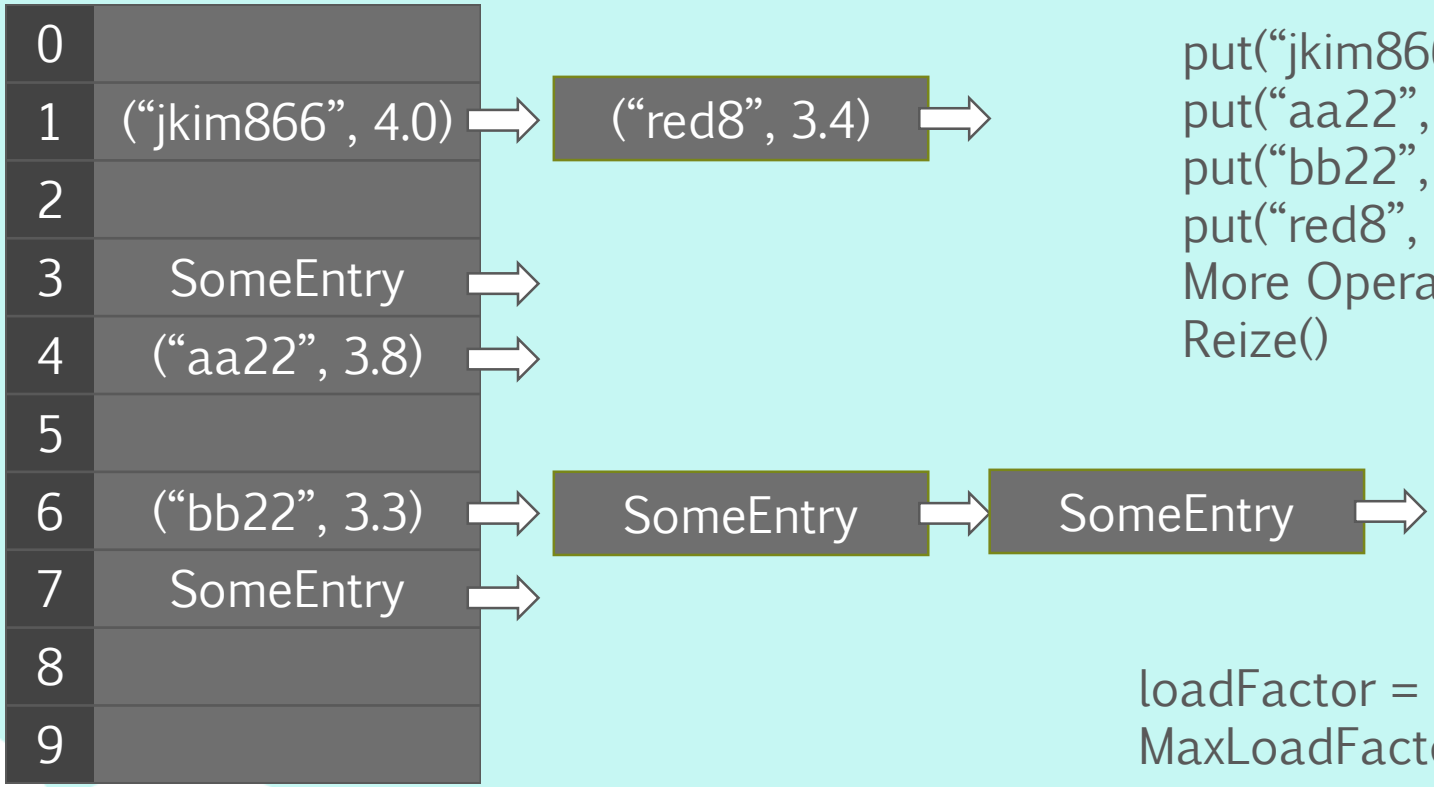
Resize Visualization



put("jkim866", 4.0)
put("aa22", 3.8)
put("bb22", 3.3)
put("red8", 3.4)
More Operations.

loadFactor = 0.7
MaxLoadFactor = 0.7

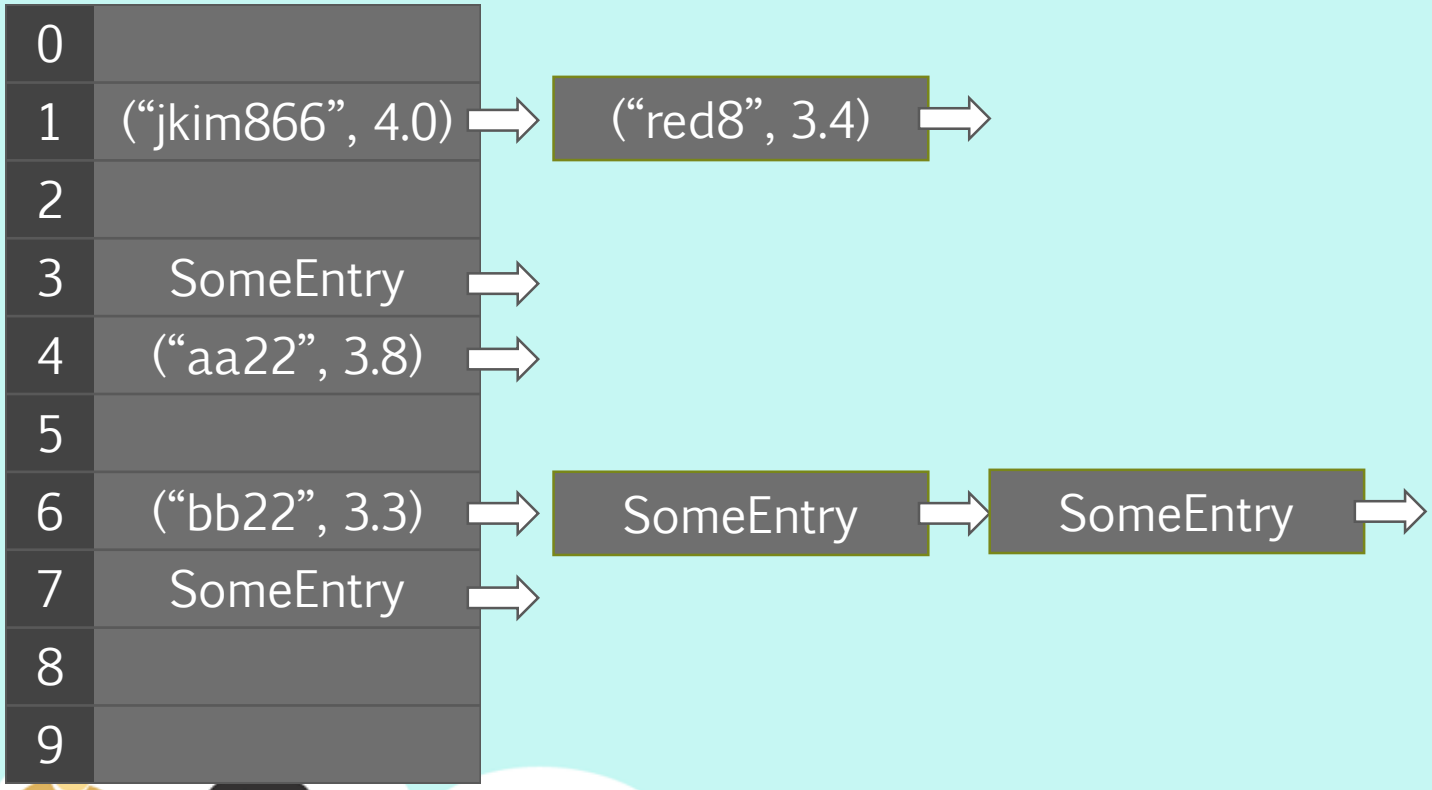
Resize Visualization



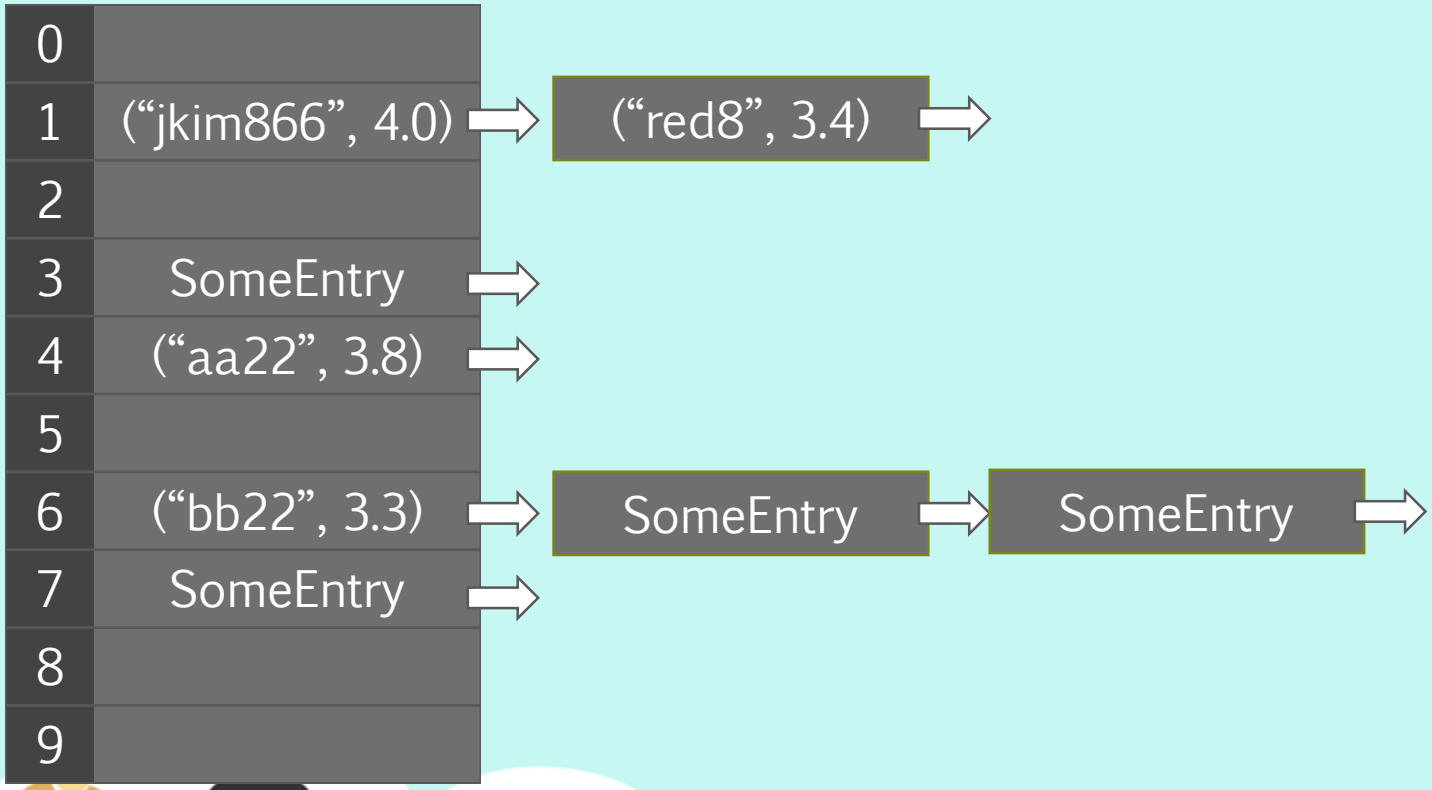
put("jkim866", 4.0)
put("aa22", 3.8)
put("bb22", 3.3)
put("red8", 3.4)
More Operations.
Reize()

loadFactor = **0.8**
MaxLoadFactor = 0.7

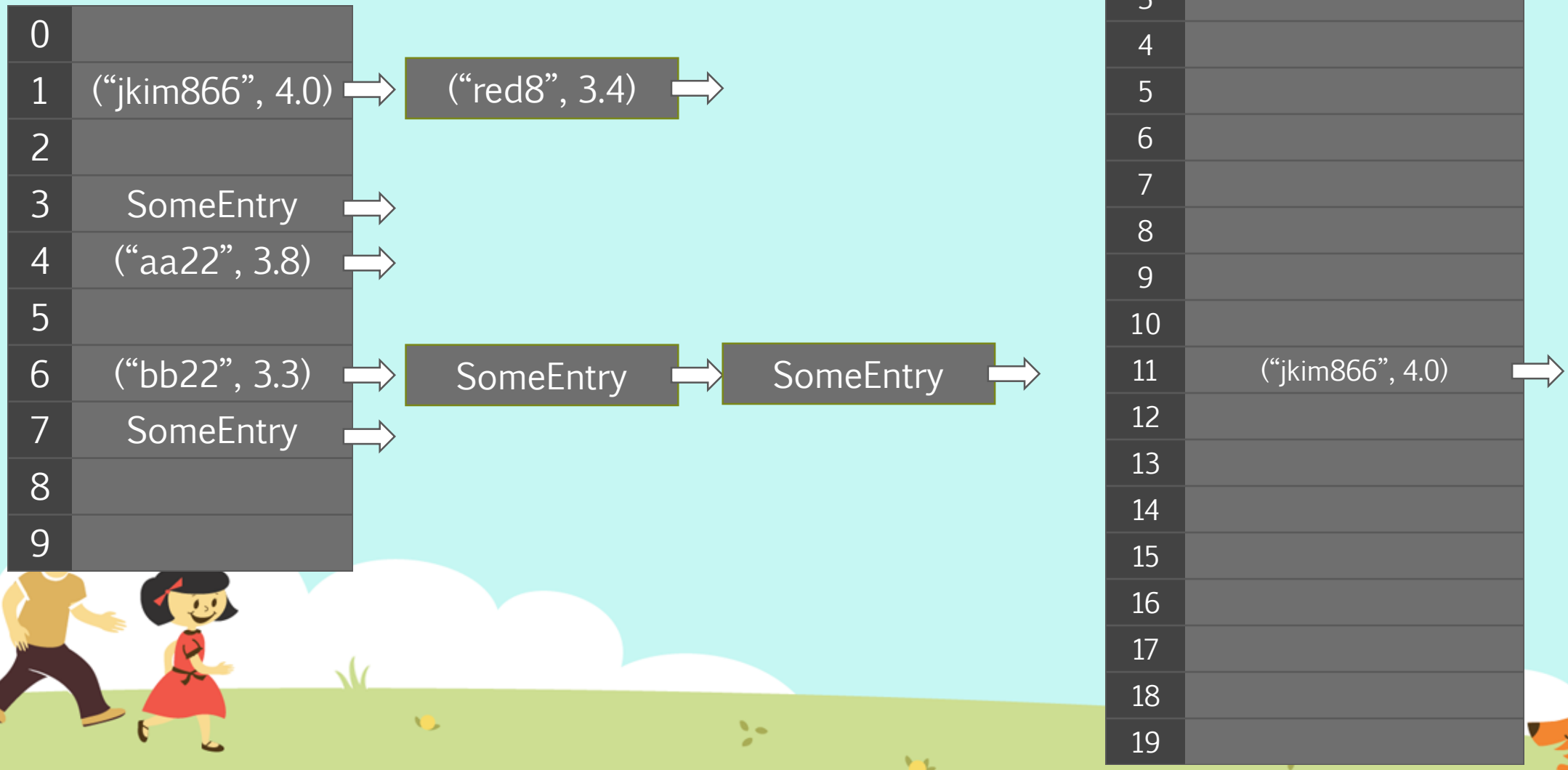
Resize Visualization



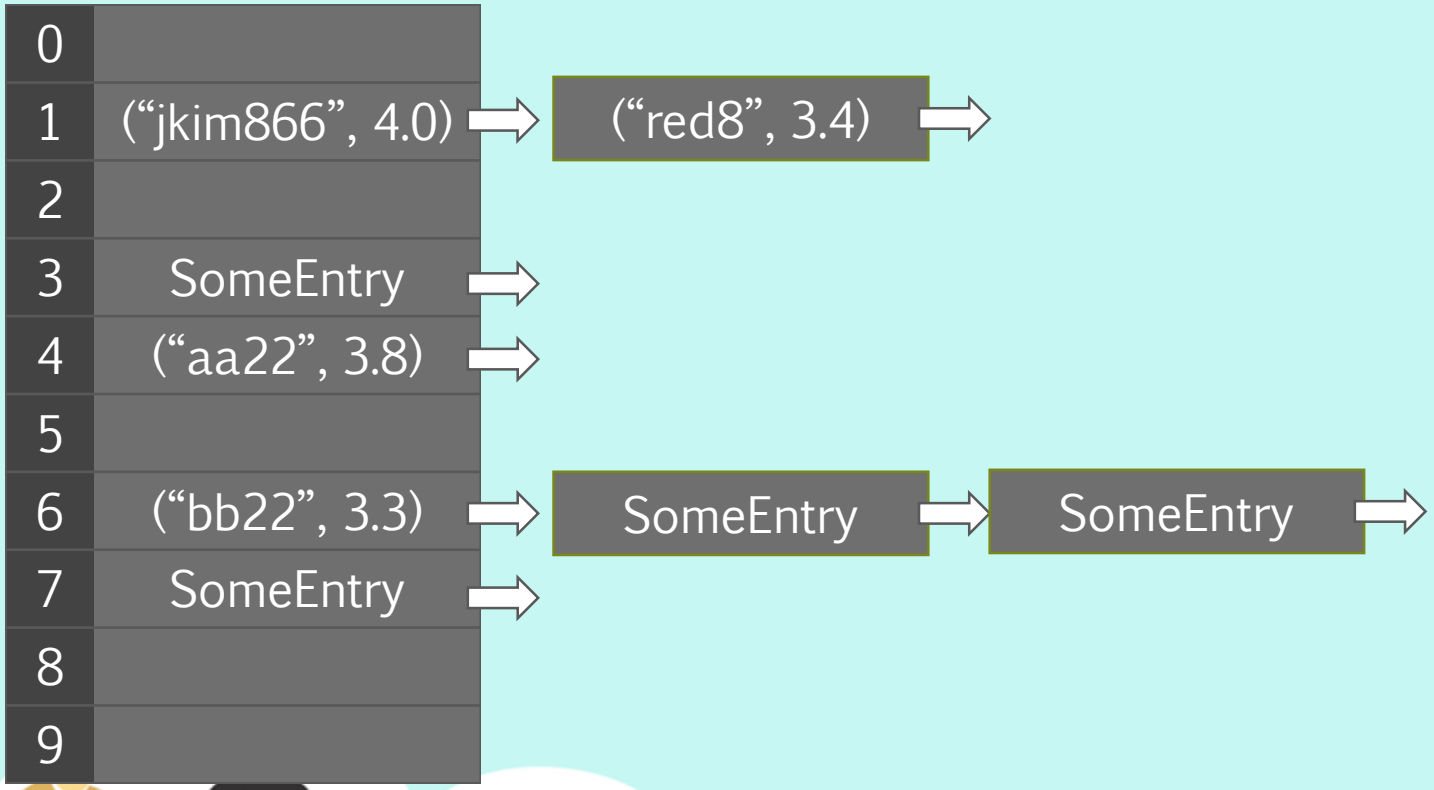
Resize Visualization

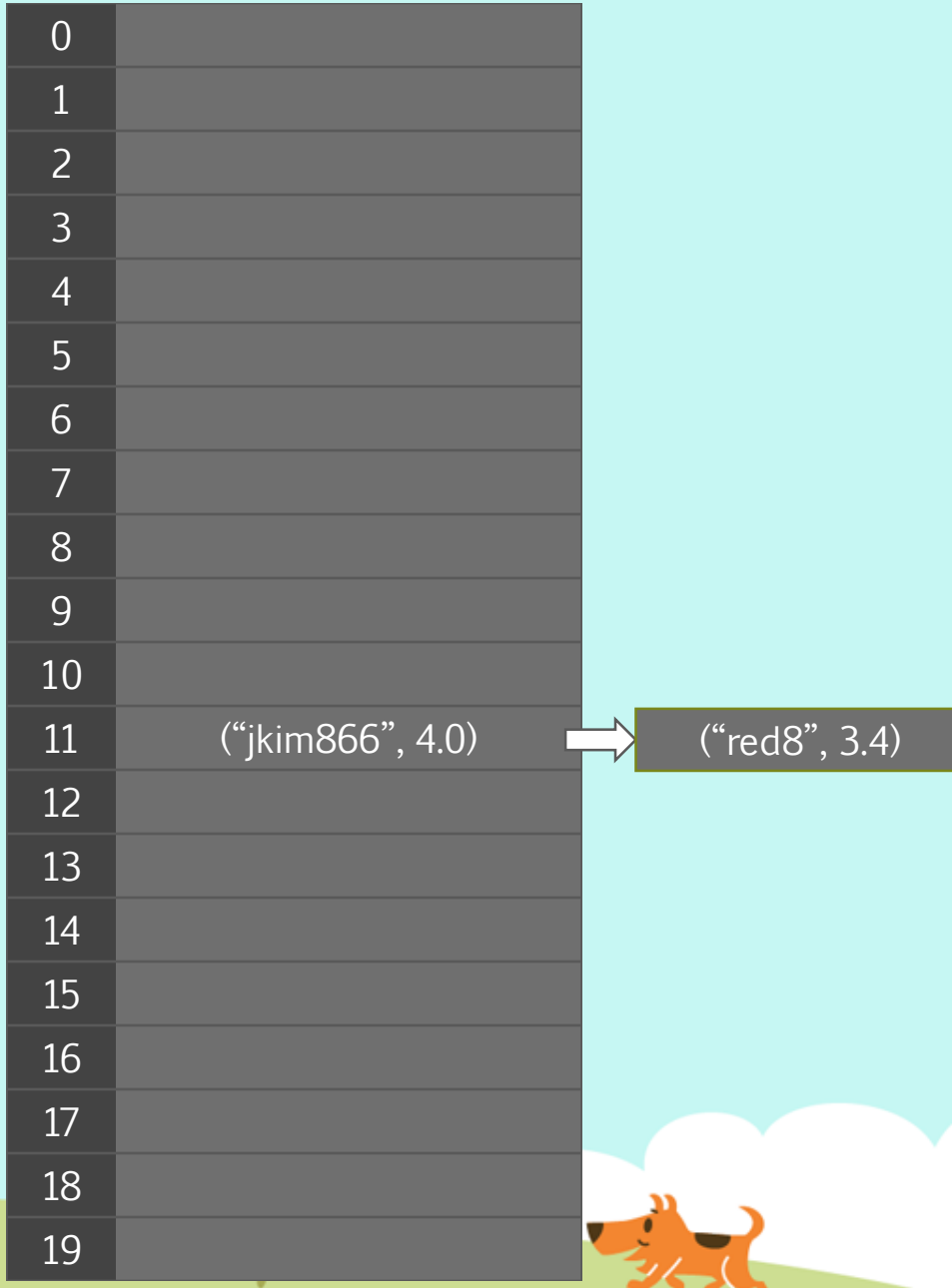


Resize Visualization

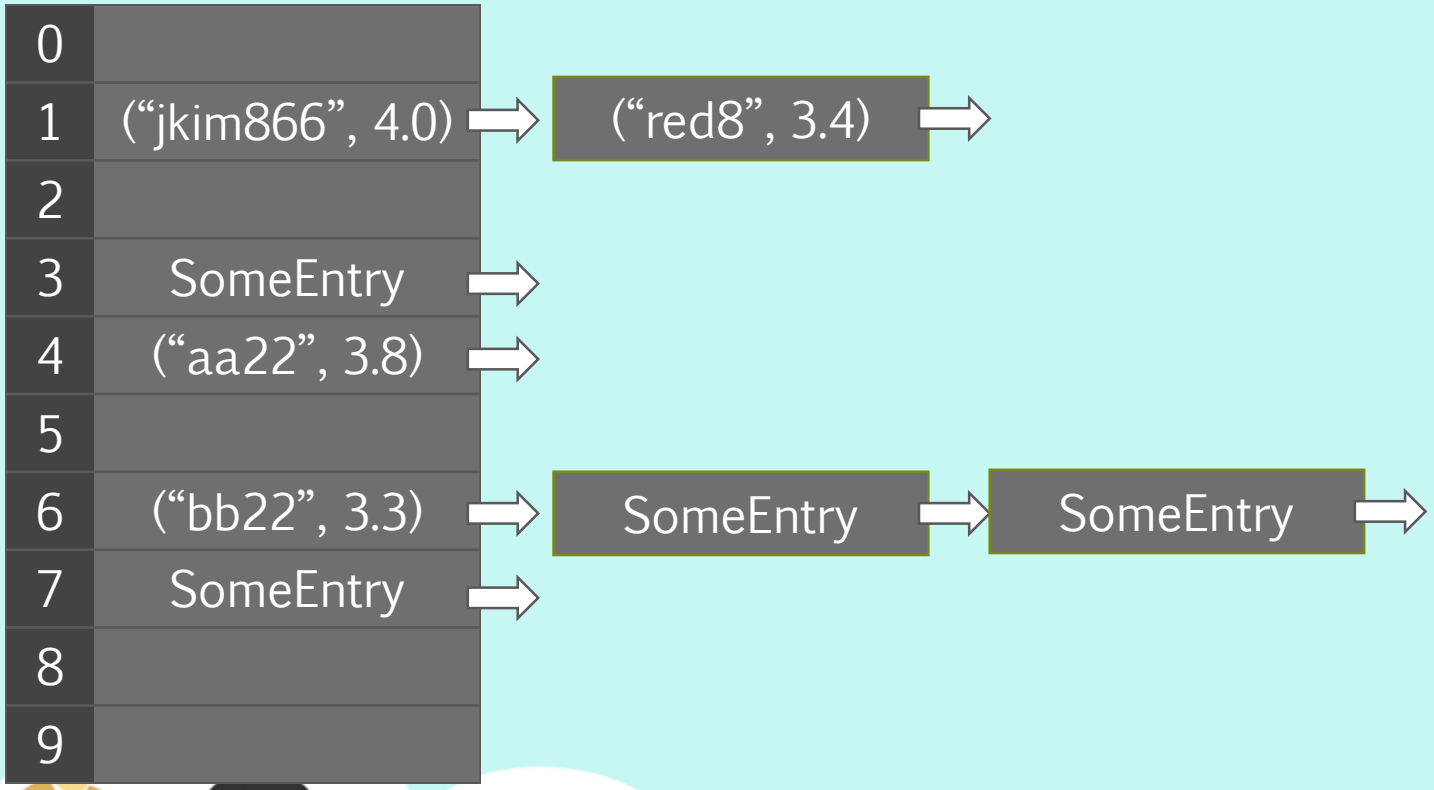


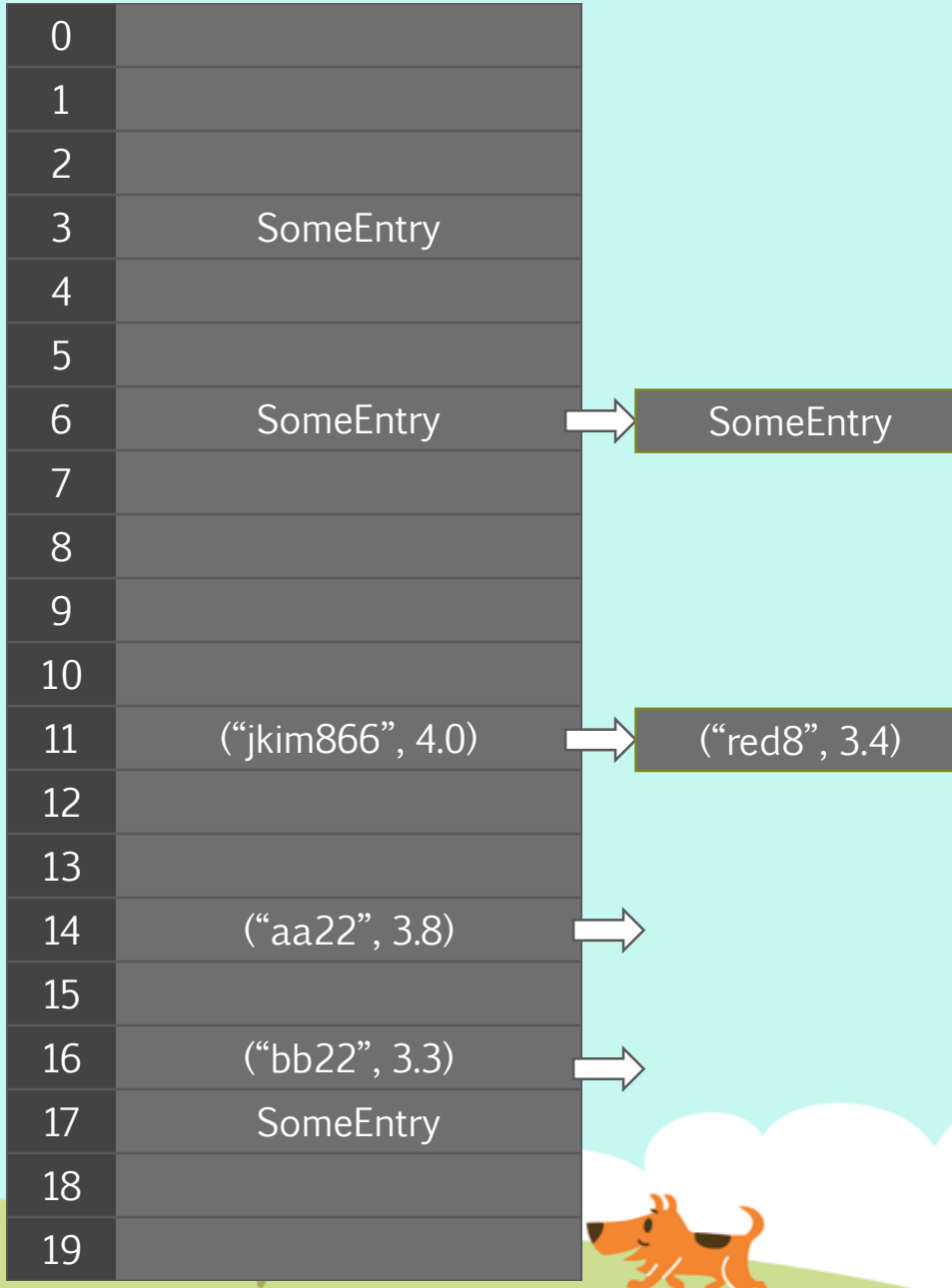
Resize Visualization





Resize Visualization





ization

0	
1	
2	
3	SomeEntry
4	
5	
6	SomeEntry → SomeEntry
7	
8	
9	
10	
11	(“jkim866”, 4.0) → (“red8”, 3.4)
12	
13	
14	(“aa22”, 3.8)
15	
16	(“bb22”, 3.3)
17	SomeEntry
18	
19	

loadFactor = 0.4
MaxLoadFactor = 0.7

Last Time…

- Take 3 min to write down a summary of the following:
  - What is a HashMap
  - What is the idea of Hash Functions?
    - How are they integrated into Hash Maps
  - What is external chaining?

Collision Handling: Open Addressing

- Another we can handle collisions is a type of method called Open Addressing

Collision Handling: Open Addressing

- Another we can handle collisions is a type of method called Open Addressing
  - When we have a collision at an index, we find alternative locations inside the array until we find our entry or an empty cell.

Collision Handling: Open Addressing

- Another we can handle collisions is a type of method called Open Addressing
  - When we have a collision at an index, we find alternative locations inside the array until we find our entry or an empty cell.
  - This idea of continually looking at alternative locations is called **probing**.

Collision Handling: Open Addressing

- Another we can handle collisions is a type of method called Open Addressing
  - When we have a collision at an index, we find alternative locations inside the array until we find our entry or an empty cell.
  - This idea of continually looking at alternative locations is called **probing**.
  - We will go over 3 different types of probing:

Collision Handling: Open Addressing

- Another we can handle collisions is a type of method called Open Addressing
  - When we have a collision at an index, we find alternative locations inside the array until we find our entry or an empty cell.
  - This idea of continually looking at alternative locations is called **probing**.
  - We will go over 3 different types of probing:
    - Linear Probing
    - Quadratic Probing
    - Double Hashing

Collision Handling: Linear Probing

- Linear probing is a probing method where given an index i = hashFunction(key) % arr.length, if we have a collision at index i, we will check sequential cells at i + 1, i + 2, i + 3, and so on until:
  - We find our entry with our key
  - We find an empty array index.

Collision Handling: Linear Probing

- Linear probing is a probing method where given an index i = hashFunction(key) % arr.length, if we have a collision at index i, we will check sequential cells at i + 1, i + 2, i + 3, and so on until:
  - We find our entry with our key
  - We find an empty array index.
- For adding and searching, we start out with index i, then continually increment i until we find our index.

Linear Probing Visualization

0	
1	(“jkim866”, 4.0)
2	
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	

(“red8”, 3.4)

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
index = 1

Linear Probing Visualization

0	
1	(“jkim866”, 4.0)
2	
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	

(“red8”, 3.4)

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
    index = 1
    if (arr[index] is occupied),
        index = (index + 1)% arr.length

Linear Probing Visualization

0	
1	(“jkim866”, 4.0)
2	(“red8”, 3.4)
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	

```
put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
    index = 1
    if (arr[index]  is occupied),
        index = (index + 1)% arr.length
    arr[index]  = (“red8,” 3.4)
```

Collision Handling: Linear Probing

- Linear probing is a probing method where given an index i = hashFunction(key) % arr.length, if we have a collision at index i, we will check sequential cells at i + 1, i + 2, i + 3, and so on until:
  - We find our entry with our key
  - We find an empty array index.
- For adding and searching, we start out with index i, then continually increment i until we find our index.

Collision Handling: Linear Probing

- Linear probing is a probing method where given an index i = hashFunction(key) % arr.length, if we have a collision at index i, we will check sequential cells at i + 1, i + 2, i + 3, and so on until:
  - We find our entry with our key
  - We find an empty array index.
- For adding and searching, we start out with index i, then continually increment i until we find our index.
- What about removing?

Collision Handling: Linear Probing Removing

- For removing, we cannot simply find our entry and delete the cell
  - What if all our entries hashed at index 1 and we then delete index 1?

Collision Handling: Linear Probing Removing

- For removing, we cannot simply find our entry and delete the cell
  - What if all our entries hashed at index 1 and we then delete index 1?
    - If we were to search for another key that hashed to index 1 but probed to index 3, we would be unable to locate that key.

Linear Probing Remove Visualization

0	
1	(“jkim866”, 4.0)
2	(“red8”, 3.4)
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)

Linear Probing Remove Visualization

0	
1	~~(“jkim866”, 4.0)~~
2	(“red8”, 3.4)
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
remove(“jkim866”)

Linear Probing Remove Visualization

0	
1	
2	(“red8”, 3.4)
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
remove(“jkim866”)

Linear Probing Remove Visualization

0	
1	
2	(“red8”, 3.4)
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
remove(“jkim866”)
search(“red8”)

Linear Probing Remove Visualization

0	
1	
2	(“red8”, 3.4)
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	

```
put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
remove(“jkim866”)
search(“red8”)
    index = 1
    if (arr[index]  == null) stop
```

Collision Handling: Linear Probing Removing

- For removing, we cannot simply find our entry and delete the cell
  - What if all our entries hashed at index 1 and we then delete index 1?
    - If we were to search for another key that hashed to index 1 but probed to index 3, we would be unable to locate that key.

Collision Handling: Linear Probing Removing

- For removing, we cannot simply find our entry and delete the cell
  - What if all our entries hashed at index 1 and we then delete index 1?
    - If we were to search for another key that hashed to index 1 but probed to index 3, we would be unable to locate that key.
- We will do a **lazy removal** by not removing our entry, but marking it with a flag value and pretending it's removed

Collision Handling: Linear Probing Removing

- For removing, we cannot simply find our entry and delete the cell
  - What if all our entries hashed at index 1 and we then delete index 1?
    - If we were to search for another key that hashed to index 1 but probed to index 3, we would be unable to locate that key.
- We will do a **lazy removal** by not removing our entry, but marking it with a flag value and pretending it's removed
  - Our Entry class will include a boolean flag that indicates whether it has been marked as removed or not.

Linear Probing Remove Visualization

0	
1	(“jkim866”, 4.0)
2	(“red8”, 3.4)
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)

Linear Probing Remove Visualization

0	
1	(“jkim866”, 4.0)
2	(“red8”, 3.4)
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
remove(“jkim866”)

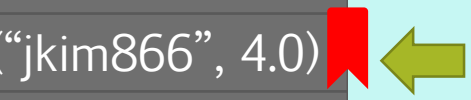
Linear Probing Remove Visualization

0	
1	(“jkim866”, 4.0)
2	(“red8”, 3.4)
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
remove(“jkim866”)
search(“red8”)

Linear Probing Remove Visualization

0	
1	(“jkim866”, 4.0)
2	(“red8”, 3.4)
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	



put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
remove(“jkim866”)
search(“red8”)
    index = 1
    if (arr[index] == null) stop
    else probe

Linear Probing Remove Visualization

0	
1	(“jkim866”, 4.0)
2	(“red8”, 3.4)
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
remove(“jkim866”)
search(“red8”)
    index = 1
    if (arr[index] == null) stop
    else probe

Linear Probing Remove Visualization

0	
1	(“jkim866”, 4.0)
2	(“red8”, 3.4)
3	
4	(“aa22”, 3.8)
5	
6	(“bb22”, 3.3)
7	
8	
9	

put(“jkim866”, 4.0)
put(“aa22”, 3.8)
put(“bb22”, 3.3)
put(“red8”, 3.4)
remove(“jkim866”)
search(“red8”)
    index = 1
    if (arr[index] == null) stop
    else probe
    return 3.4

Collision Handling: Linear Probing Removing

- For removing, we cannot simply find our entry and delete the cell
  - What if all our entries hashed at index 1 and we then delete index 1?
    - If we were to search for another key that hashed to index 1 but probed to index 3, we would be unable to locate that key.
- We will do a **lazy removal** by not removing our entry, but marking it with a flag value and pretending it's removed
  - Our Entry class will include a boolean flag that indicates whether it has been marked as removed or not.

Collision Handling: Linear Probing Removing

- For removing, we cannot simply find our entry and delete the cell
  - What if all our entries hashed at index 1 and we then delete index 1?
    - If we were to search for another key that hashed to index 1 but probed to index 3, we would be unable to locate that key.
- We will do a **lazy removal** by not removing our entry, but marking it with a flag value and pretending it's removed
  - Our Entry class will include a boolean flag that indicates whether it has been marked as removed or not.
  - Flagged entries will be removed later

Collision Handling: Linear Probing Removing

- For removing, we cannot simply find our entry and delete the cell
  - What if all our entries hashed at index 1 and we then delete index 1?
    - If we were to search for another key that hashed to index 1 but probed to index 3, we would be unable to locate that key.
- We will do a **lazy removal** by not removing our entry, but marking it with a flag value and pretending it's removed
  - Our Entry class will include a boolean flag that indicates whether it has been marked as removed or not.
  - Flagged entries will be removed later
- With this flag, before we act upon an entry, we must check if it has been flagged or not.
  - For add, flagged entries can be overwritten by a new entry

Linear Probing Exercise

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Apply these operations:
put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)
put(5, “Sarah”)
remove(3)
remove(23)
put(33, “Poppy”)

```
hashFunction(key):
    return key
```

Linear Probing Exercise

0	
1	
2	
3	
4	(4, “Amy”)
5	
6	
7	
8	
9	

Apply these operations:
put(4, “Amy”)

hashFunction(key):
    return key

Linear Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	
7	
8	
9	

Apply these operations:
put(4, “Amy”)
put(3, “John”)

hashFunction(key):
    return key

Linear Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	

Apply these operations:
put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)

hashFunction(key):
    return key

Linear Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	(23, “Joonho”)
6	(16, “Luke”)
7	
8	
9	

Apply these operations:
put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)

hashFunction(key):
    return key

Linear Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	(23, “Joonho”)
6	(16, “Luke”)
7	(5, “Sarah”)
8	
9	

Apply these operations:
put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)
put(5, “Sarah”)

hashFunction(key):
    return key

Linear Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	(23, “Joonho”)
6	(16, “Luke”)
7	(5, “Sarah”)
8	
9	

Apply these operations:
put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)
put(5, “Sarah”)
remove(3)

hashFunction(key):
    return key

Linear Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	(23, “Joonho”)
6	(16, “Luke”)
7	(5, “Sarah”)
8	
9	

Apply these operations:
put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)
put(5, “Sarah”)
remove(3)
remove(23)

```
hashFunction(key):
    return key
```

Linear Probing Exercise

0	
1	
2	
3	(33, “Poppy”)
4	(4, “Amy”)
5	(23, “Joonho”)
6	(16, “Luke”)
7	(5, “Sarah”)
8	
9	

Apply these operations:
put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)
put(5, “Sarah”)
remove(3)
remove(23)
put(33, “Poppy”)

```
hashFunction(key):
    return key
```

Collision Handling: Quadratic Probing

- Quadratic probing is a probing method where given an index i = hashFunction(key) % arr.length, if we have a collision at index i, we will check cells following this equation:

Collision Handling: Quadratic Probing

- Quadratic probing is a probing method where given an index i = hashFunction(key) % arr.length, if we have a collision at index i, we will check cells following this equation:
  - Index i = (hashFunction(key) + c1* p + c2*p^2) % arr.length
    - c1, c2 are constants
    - p is the current probe number attempt

Collision Handling: Quadratic Probing

- Quadratic probing is a probing method where given an index i = hashFunction(key) % arr.length, if we have a collision at index i, we will check cells following this equation:
  - Index i = (hashFunction(key) + c1* p + c2*p^2) % arr.length
    - c1, c2 are constants
    - p is the current probe number attempt
  - Example: for c1 = c2 = 2, hashFunction(key) = 3, and we're attempting our 2nd probe. i = (3 + 2*2 + 2*4) % arr.length = 15 % arr.length.

Collision Handling: Quadratic Probing

- Quadratic probing is a probing method where given an index i = hashFunction(key) % arr.length, if we have a collision at index i, we will check cells following this equation:
  - Index i = (hashFunction(key) + c1* p + c2*p^2) % arr.length
    - c1, c2 are constants
    - p is the current probe number attempt
  - Example: for c1 = c2 = 2, hashFunction(key) = 3, and we're attempting our 2nd probe. i = (3 + 2*2 + 2*4) % arr.length = 15 % arr.length.
- Attempts to avoid clustering from linear probing

Quadratic Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)

Index i = (hashFunction(key) + 1* p + 2*p^2) % arr.length

Quadratic Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)

Index i = (hashFunction(key) + 1* p + 2*p^2) % arr.length

Quadratic Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)
i = 23 + 1*0 + 2*0 % arr.length

Index i = (hashFunction(key) + 1* p + 2*p^2) % arr.length

Quadratic Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)
i = 23 + 1*1 + 2*1 % arr.length

Index i = (hashFunction(key) + 1* p + 2*p^2) % arr.length

Quadratic Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)
i = 23 + 1*2 + 2*4 % arr.length

Index i = (hashFunction(key) + 1* p + 2*p^2) % arr.length

Quadratic Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)
i = 23 + 1*3 + 2*9 % arr.length

Index i = (hashFunction(key) + 1* p + 2*p^2) % arr.length

Quadratic Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	(23, “Joonho”)

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)
i = 23 + 1*4 + 2*16 % arr.length

Index i = (hashFunction(key) + 1* p + 2*p^2) % arr.length

Quadratic Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	(23, “Joonho”)

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)

Apply these operations:
put(33, “Poppy”)
put(44, “Jojo”)

Index i = (hashFunction(key) + 1* p + 2*p^2) % arr.length

Quadratic Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	(23, “Joonho”)

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)

Apply these operations:
put(33, “Poppy”)
skipping probe iterations…
i = 3 + 1*4 + 2*16 % arr.length

Index i = (hashFunction(key) + 1* p + 2*p^2) % arr.length

Quadratic Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	(33, “Poppy”)
9	(23, “Joonho”)

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)

Apply these operations:
put(33, “Poppy”)
skipping probe iterations…
i = 3 + 1*5 + 2*25 % arr.length

Index i = (hashFunction(key) + 1* p + 2*p^2) % arr.length

Quadratic Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	(33, “Poppy”)
9	(23, “Joonho”)

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)

Apply these operations:
put(33, “Poppy”)
put(44, “Jojo”)
i = 4 + 1*0 + 2*0 % arr.length

Index i = (hashFunction(key) + 1* p + 2*p^2) % arr.length

Quadratic Probing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	(44, “Jojo”)
8	(33, “Poppy”)
9	(23, “Joonho”)

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)

Apply these operations:
put(33, “Poppy”)
put(44, “Jojo”)
i = 4 + 1*1 + 2*1 % arr.length

Index i = (hashFunction(key) + 1* p + 2*p^2) % arr.length

Collision Handling: Quadratic Probing

- Quadratic probing is a probing method where given an index i = hashFunction(key) % arr.length, if we have a collision at index i, we will check cells following this equation:
  - Index i = (hashFunction(key) + c1* p + c2*p^2) % arr.length
    - c1, c2 are constants
    - p is the current probe number attempt
  - Example: for c1 = c2 = 2, hashFunction(key) = 3, and we're attempting our 2nd probe. i = (3 + 2*2 + 2*4) % arr.length = 15 % arr.length.
- Attempts to avoid clustering from linear probing

Collision Handling: Quadratic Probing

- Quadratic probing is a probing method where given an index i = hashFunction(key) % arr.length, if we have a collision at index i, we will check cells following this equation:
  - Index i = (hashFunction(key) + c1* p + c2*p^2) % arr.length
    - c1, c2 are constants
    - p is the current probe number attempt
  - Example: for c1 = c2 = 2, hashFunction(key) = 3, and we're attempting our 2nd probe. i = (3 + 2*2 + 2*4) % arr.length = 15 % arr.length.
- Attempts to avoid clustering from linear probing
- For this class, you can assume c1 = 0 and c2 = 1.
  - Index i = (hashFunction(key) + p^2)

Collision Handling: Double Hashing

- Double Hashing is a probing method where instead if having 1 hash function, we use 2 hash functions.

Collision Handling: Double Hashing

- Double Hashing is a probing method where instead if having 1 hash function, we use 2 hash functions.
  - Index i = (h1(key) + p*h2(key)) % arr.length
    - h1() and h2() are different hash functions
    - p is the current probe number attempt

Collision Handling: Double Hashing

- Double Hashing is a probing method where instead if having 1 hash function, we use 2 hash functions.
  - Index i = (h1(key) + p*h2(key)) % arr.length
    - h1() and h2() are different hash functions
    - p is the current probe number attempt
  - Example: h1(key) = key, h2(key) = key % 5. if key = 7 and we're attempting our 2^{nd} probe, then i = (h1(7) + 2 * h2(7)) % arr.length = 11 % arr.length

Double Hashing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)

Index i = (h1(key) + p*h2(key)) % arr.length
h1(key) = key
h2(key) = key % 5

Double Hashing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)

Index i = (h1(key) + p*h2(key)) % arr.length
h1(key) = key
h2(key) = key % 5

Double Hashing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)
i = (23 + 0 * 3) % arr.length

Index i = (h1(key) + p*h2(key)) % arr.length
h1(key) = key
h2(key) = key % 5

Double Hashing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)
i = (23 + 1 * 3) % arr.length

Index i = (h1(key) + p*h2(key)) % arr.length
h1(key) = key
h2(key) = key % 5

Double Hashing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)
i = (23 + 2 * 3) % arr.length

Index i = (h1(key) + p*h2(key)) % arr.length
h1(key) = key
h2(key) = key % 5

Double Hashing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	(23, “Joonho”)

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)
i = (23 + 2 * 3) % arr.length

Index i = (h1(key) + p*h2(key)) % arr.length
h1(key) = key
h2(key) = key % 5

Double Hashing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	(23, “Joonho”)

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)

Apply these operations:
put(33, “Poppy”)
put(44, “Jojo”)

Index i = (h1(key) + p*h2(key)) % arr.length
h1(key) = key
h2(key) = key % 5

Double Hashing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	(23, “Joonho”)

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)

Apply these operations:
put(33, “Poppy”)
i = (33 + 0*3) % arr.length

Index i = (h1(key) + p*h2(key)) % arr.length
h1(key) = key
h2(key) = key % 5

Double Hashing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	(23, “Joonho”)

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)

Apply these operations:
put(33, “Poppy”)
i = (33 + 1*3) % arr.length

Index i = (h1(key) + p*h2(key)) % arr.length
h1(key) = key
h2(key) = key % 5

Double Hashing Exercise

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	(23, “Joonho”)

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)

Apply these operations:
put(33, “Poppy”)
i = (33 + 2*3) % arr.length

Index i = (h1(key) + p*h2(key)) % arr.length
h1(key) = key
h2(key) = key % 5

Double Hashing Exercise

0	
1	
2	(33, “Poppy”)
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	(23, “Joonho”)

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)

Apply these operations:
put(33, “Poppy”)
i = (33 + 3*3) % arr.length

Index i = (h1(key) + p*h2(key)) % arr.length
h1(key) = key
h2(key) = key % 5

Double Hashing Exercise

0	
1	
2	(33, “Poppy”)
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	
9	(23, “Joonho”)

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)

Apply these operations:
put(33, “Poppy”)
put(44, “Jojo”)
i = (44 + 4*0) % arr.length

Index i = (h1(key) + p*h2(key)) % arr.length
h1(key) = key
h2(key) = key % 5

Double Hashing Exercise

0	
1	
2	(33, “Poppy”)
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	
8	(44, “Jojo”)
9	(23, “Joonho”)

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)

Apply these operations:
put(33, “Poppy”)
put(44, “Jojo”)
i = (44 + 4*1) % arr.length

Index i = (h1(key) + p*h2(key)) % arr.length
h1(key) = key
h2(key) = key % 5

Hash Map Analysis

- Add/search/remove are all O(1) on average.
- Worst case, our operations are all O(n) which could result from
  - Bad hash function
  - Bad load factor
- It’s good to have array lengths of prime numbers.

Index i = (hashFunction(key) + 1* p + 2*p^2) % arr.length

Array of Length 10 vs 11 (Quadratic Probing)

0	
1	
2	
3	(3, “John”)
4	(4, “Amy”)
5	
6	(16, “Luke”)
7	(44, “Jojo”)
8	(33, “Poppy”)
9	(23, “Joonho”)

0	(33, “Poppy”)
1	(23, “Joonho”)
2	
3	(3, “John”)
4	(4, “Amy”)
5	(16, “Luke”)
6	
7	
8	
9	
10	(44, “Jojo”)

put(4, “Amy”)
put(3, “John”)
put(16, “Luke”)
put(23, “Joonho”)
put(33, “Poppy”)
put(44, “Jojo”)