B-Trees, LLRBs, Hashing

Discussion 07

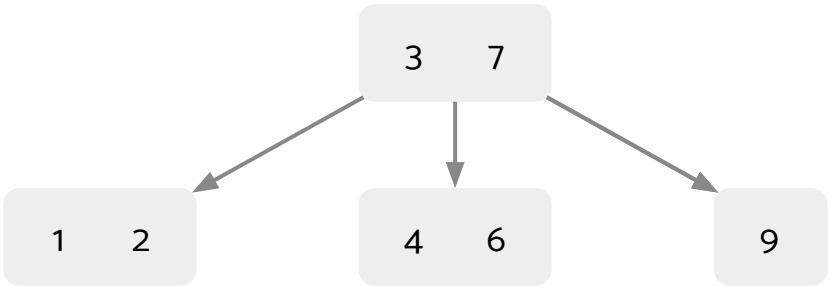
Announcements

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
			3/6 Project 2A Due		3/8 Lab 7 Due	
					3/15 Lab 8 Due	

Content Review

B-Trees

B-Trees are trees that serve a similar function to binary trees while ensuring a bushy structure (check: why don't BSTs/binary trees generally?). In this class, we'll often use B-Tree interchangeably with 2-3 Trees.

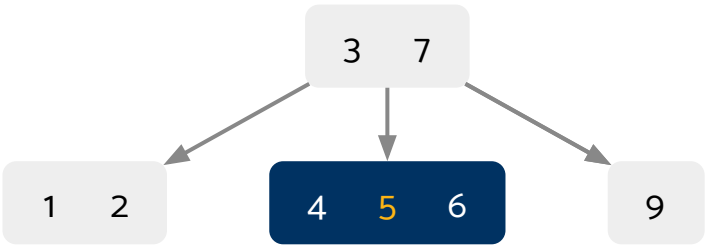


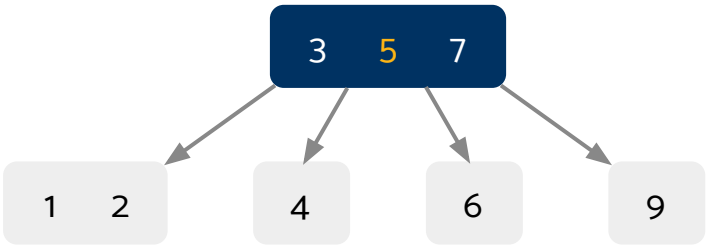
Each node can have up to 2 items and 3 children. There are variations where these values are higher, known as 2-3-4 trees (nodes can have up to 3 items and 4 children).

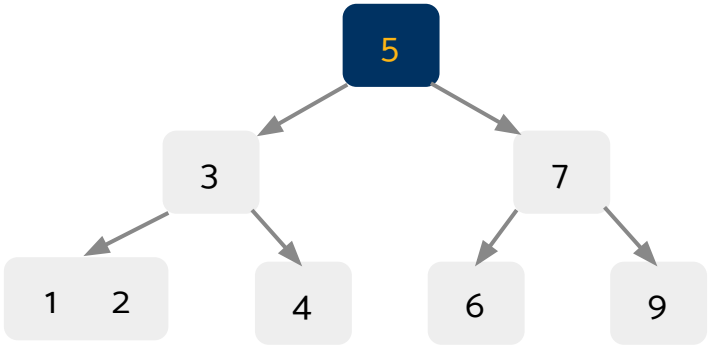
All leaves are the same distance from the root, which makes getting take Θ(log N) time.

B-Trees

When adding to a B-Tree, you first start by adding to a leaf node, and then pushing the excess items (typically the middle element) up the tree until it follows the rules (max 2 elements per node, max 3 children per node).

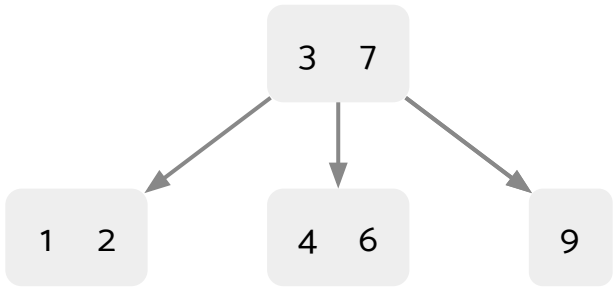


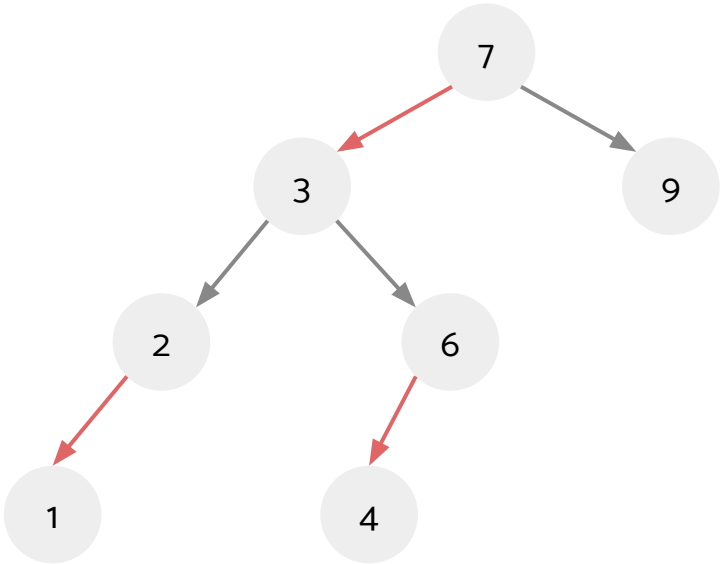




Left Leaning Red Black Trees

LLRBs are a representation of B-trees that we use because it is easier to work with in code. In an LLRB, each multi-node in a 2-3 tree is represented using a red connection on the left side.





LLRB Rules

Each 2-3 tree corresponds to a (unique) LLRB*. This implies that:

1. **The LLRB must have the same number of black links in all paths from root to null (not root to leaf!)**
2. **A node may not have two red children**
3. **All red links should be left-leaning**
4. **Height cannot be more than ~2x the height of its corresponding 2-3 tree**
5. **Additionally, we insert elements as leaves with red links to their parent node**

All these invariants mean that sometimes our LLRB becomes unbalanced (ie. it violates a rule), so we need some way to fix that.

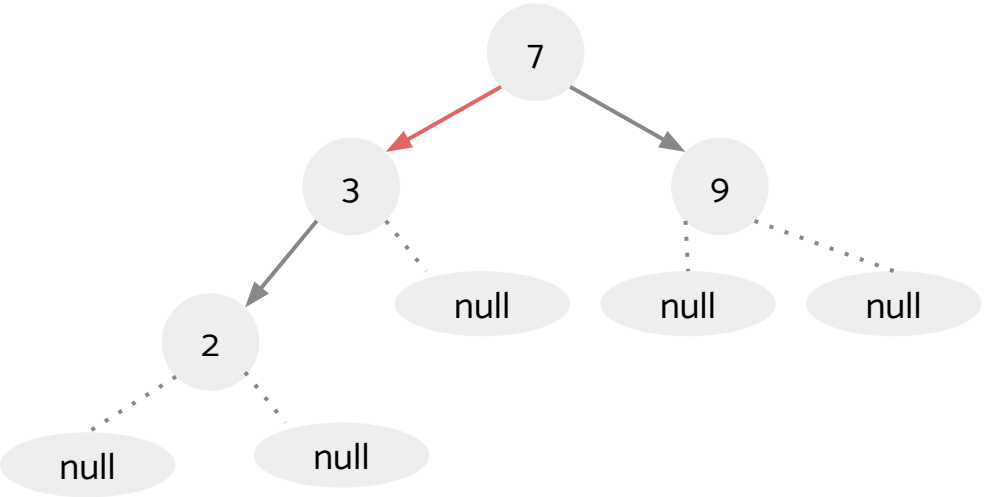
*2-3-4 trees correspond more generally to regular Red Black Trees, but our focus in 61B is on LLRBs.

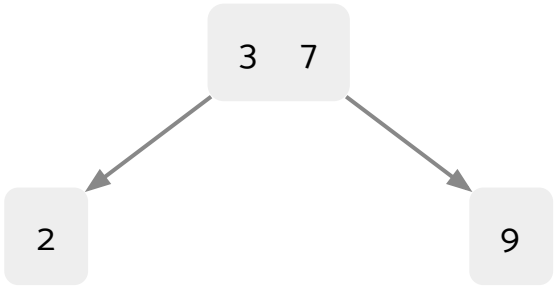
Why root to null?

Consider the left image below: each leaf is 1 black link away from the root, but it's not a valid LLRB!

This is because when we convert it to a B-tree, the [3, 7] node will only have 2 children, not 3!

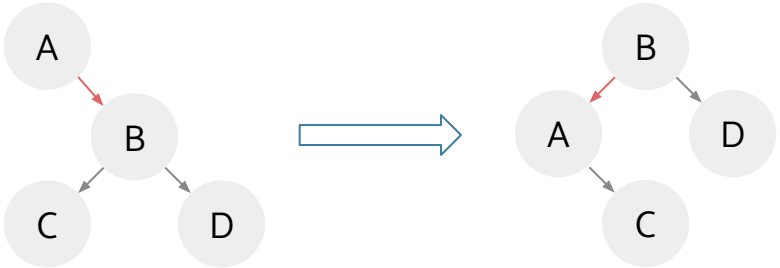
If we check for root-to-null: the right of 3 is 0 black link away, but the other nulls are 1 black link away.





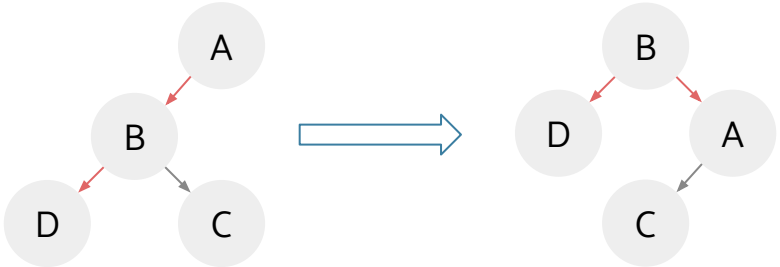
LLRB Balancing Operations

`rotateLeft(A);`



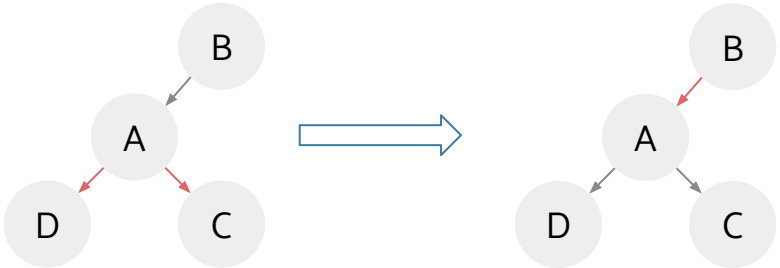
we can't have a right red link

`rotateRight(A);`



we can't have 2+ consecutive (left) red links

`colorFlip(A);`



we can't have both child links of a node be red

Hashing

Hash functions are functions that represent objects as integers so we can efficiently use data structures like HashSet and HashMap for fast operations (ie. get, put/add).

Once we have a hash for our object, we use modulo to find out which "bucket" it goes into. For example, we can create a hash function for the Dog class by overriding Object's hashCode():

```
@Override
public int hashCode() {
    return 37 * this.size + 42;
}
```

Then, when we try to put the dog into a HashSet, the HashSet code might look something like this:

```
int targetBucket = dog.hashCode() % numBuckets;
addToTargetBucket(dog, targetBucket);
```

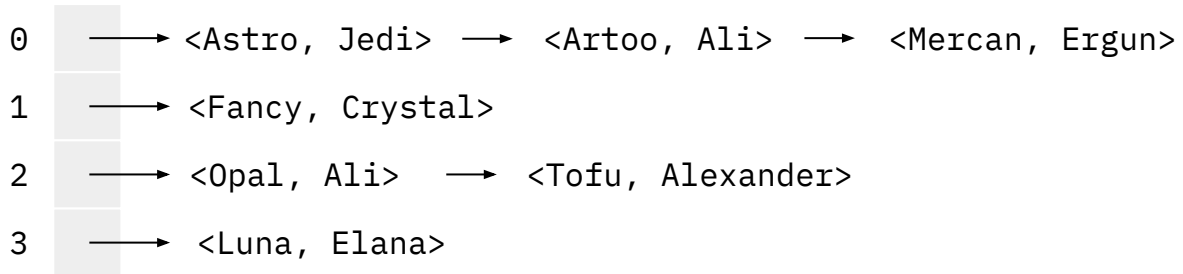


5 6 7 8 9
0 1 2 3 4


尾号0
尾号1
尾号2
尾号3
尾号4
尾号5
尾号6
尾号7
尾号8
尾号9
八大权益 全新升级
新品尝鲜
免配送费
35090-1
35082-1
35062-1
35073-1
35053-1
35094-1
35006-1
35067-1
35987-1
35087-1
35098-1
35099-1

Hashing

In each bucket, we deal with having lots of items by chaining the items and using .equals to find what we are looking for. In a HashMap, we're specifically concerned with equality of keys in key-value pairs (in HashSet, we only have a value to compare to).



Therefore, it is important that your .equals() function matches the result of comparing hashcodes - if two items are equal, they must also have the same hashcode

Hashing

The **load factor** tells us when we should resize. We calculate it by dividing the total number of elements added by the number of buckets we currently have. When resizing up, if the load factor exceeds some threshold, we increase the number of buckets we use in the data structure.

Because all elements were initially placed into buckets based on how many buckets were previously available, we also need to rehash all elements into a potentially new destination bucket when resizing, or else subsequent calls to get() may fail.*

* Resizing sounds like a linear-time operation…how does that affect the runtimes of our operations?

Valid vs. Good Hashcodes

Properties of a valid hashcode:

1) Must be an integer
2) The hashcode for the same object should always be the same
3) If two objects are “equal”, they have the same hashcode
    - Check! What about the reverse?

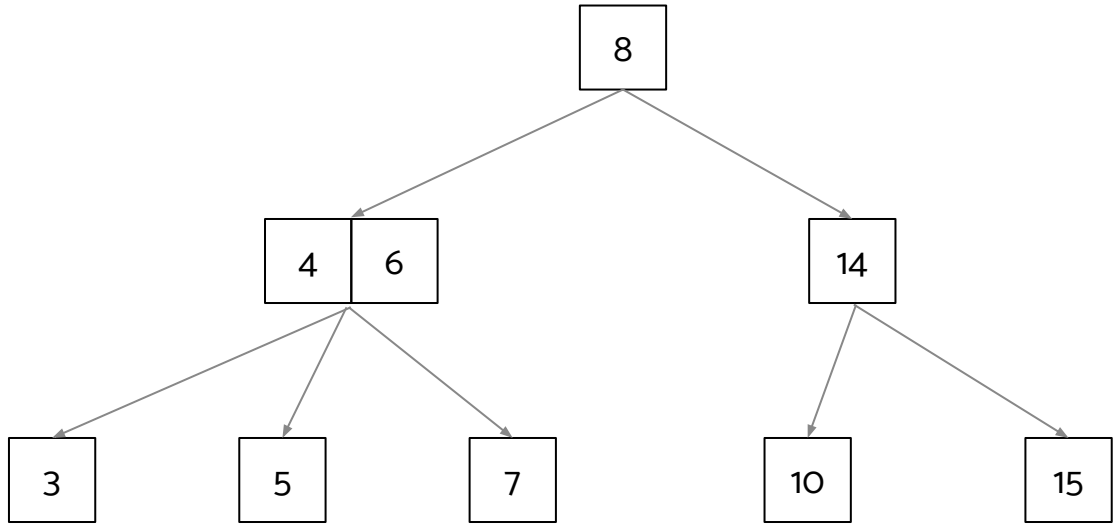
Properties of a good hashcode:

1) Distributes elements evenly
    - What does this even mean?

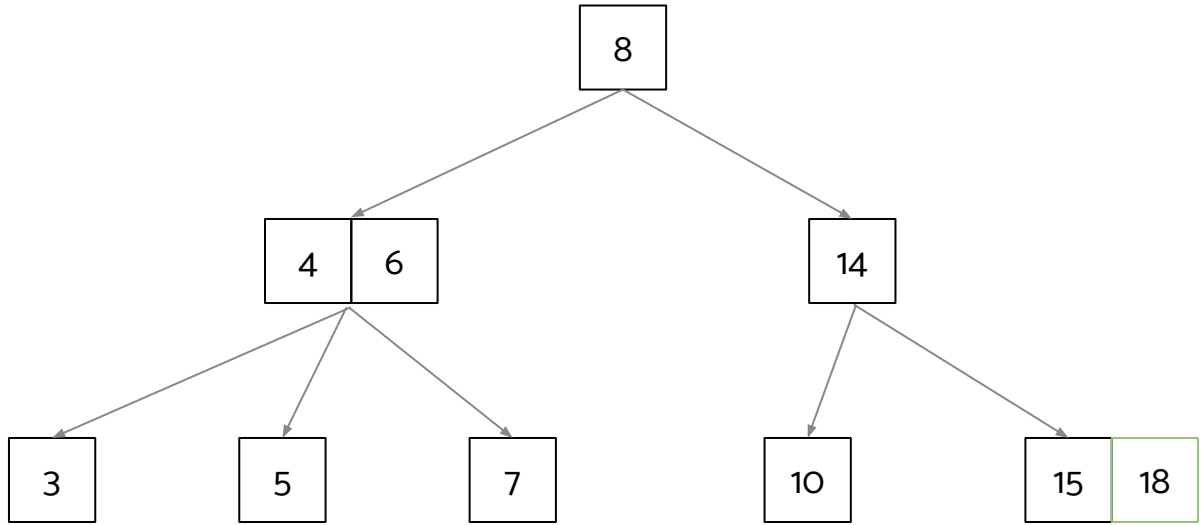
Worksheet

1A 2-3 Trees and LLRBs

Draw what the following 2-3 tree would look like after inserting 18, 38, 12, 13, and 20.



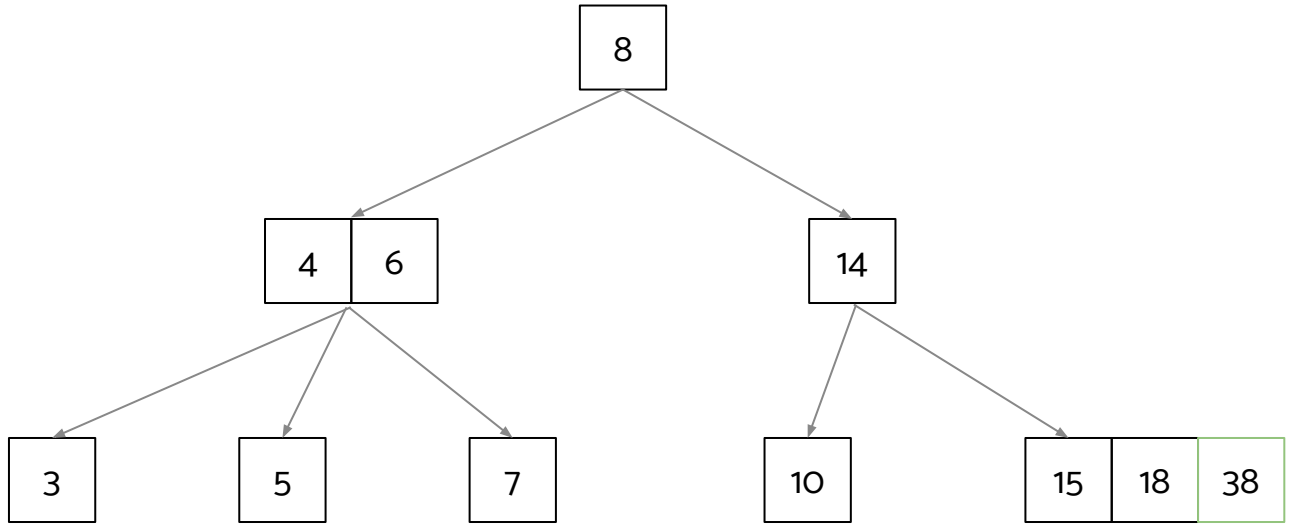
1A 2-3 Trees and LLRBs Draw what the following 2-3 tree would look like after inserting 18, 38, 12, 13, and 20.



Inserting 18

1A 2-3 Trees and LLRBs

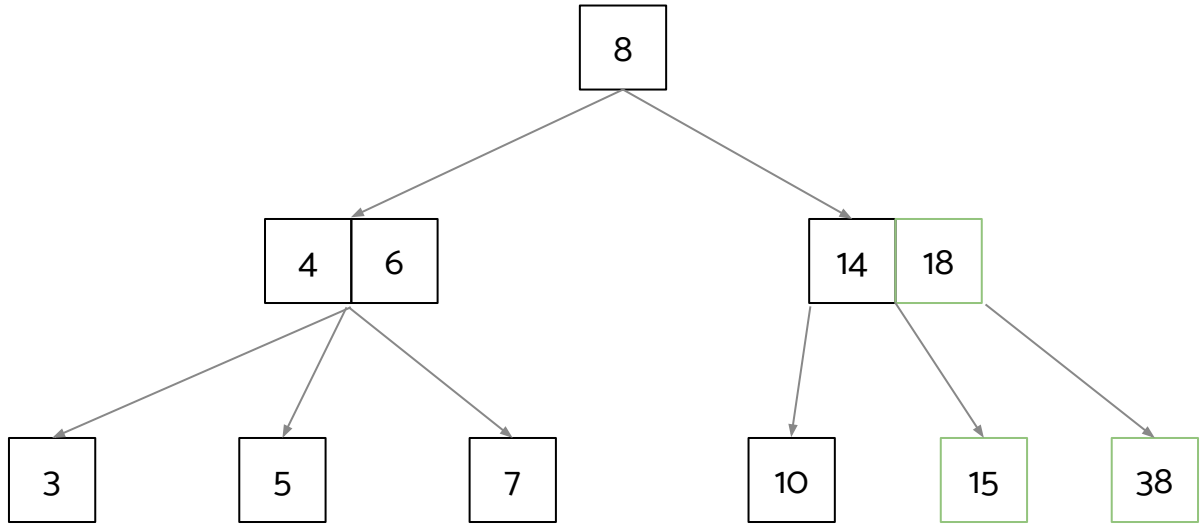
Draw what the following 2-3 tree would look like after inserting 18, 38, 12, 13, and 20.



Inserting 38 (part 1)

1A 2-3 Trees and LLRBs

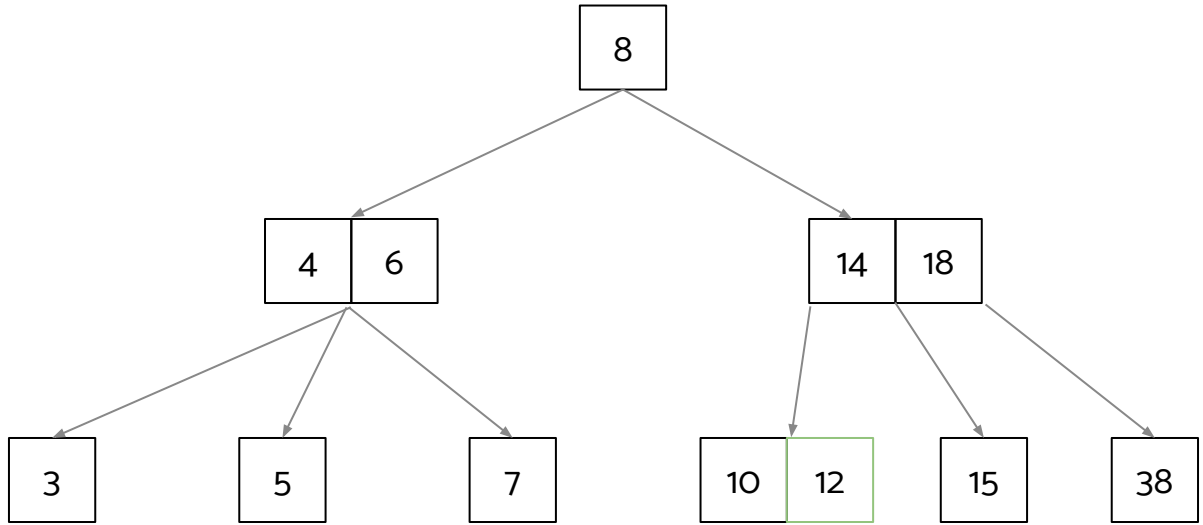
Draw what the following 2-3 tree would look like after inserting 18, 38, 12, 13, and 20.



Inserting 38 (part 2)

1A 2-3 Trees and LLRBs

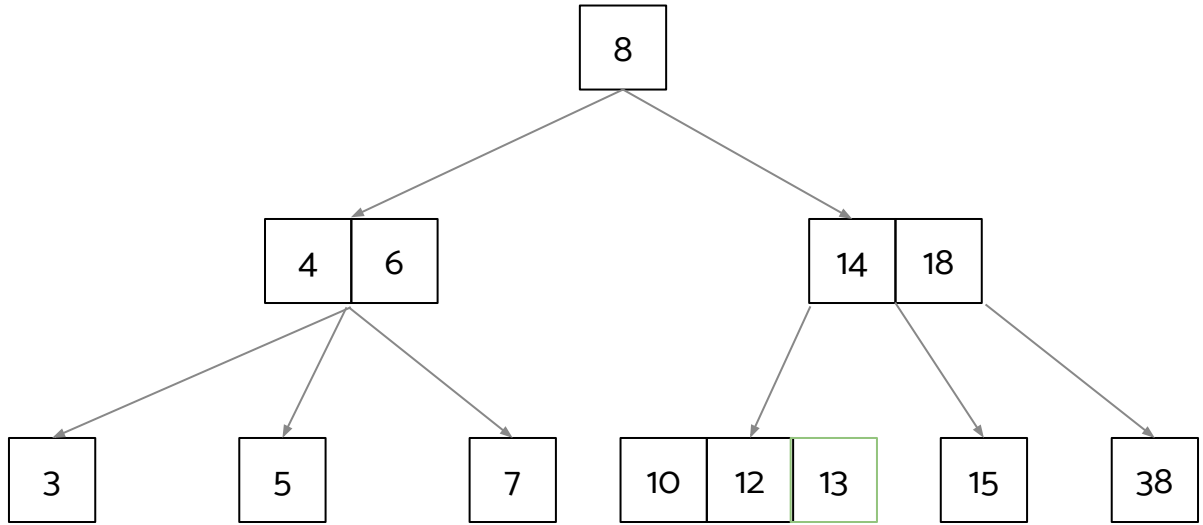
Draw what the following 2-3 tree would look like after inserting 18, 38, 12, 13, and 20.



Inserting 12

1A 2-3 Trees and LLRBs

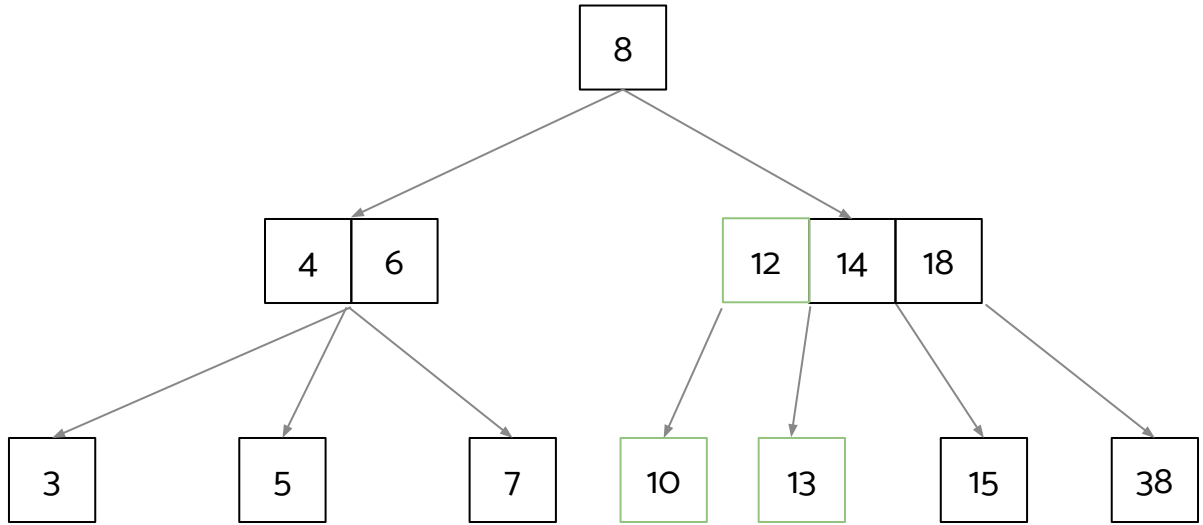
Draw what the following 2-3 tree would look like after inserting 18, 38, 12, 13, and 20.



Inserting 13 (part 1)

1A 2-3 Trees and LLRBs

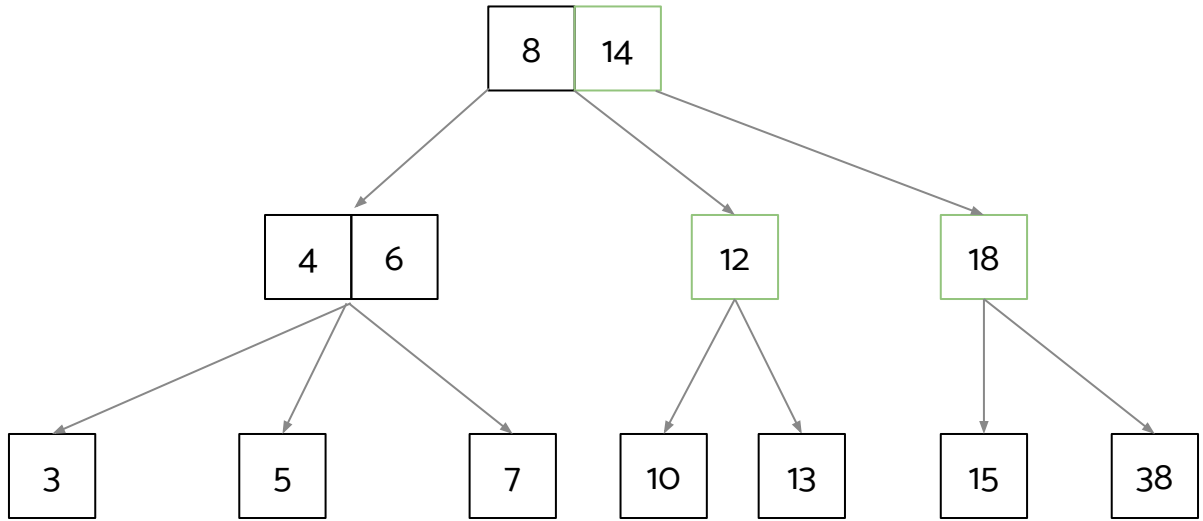
Draw what the following 2-3 tree would look like after inserting 18, 38, 12, 13, and 20.



Inserting 13 (part 2)

1A 2-3 Trees and LLRBs

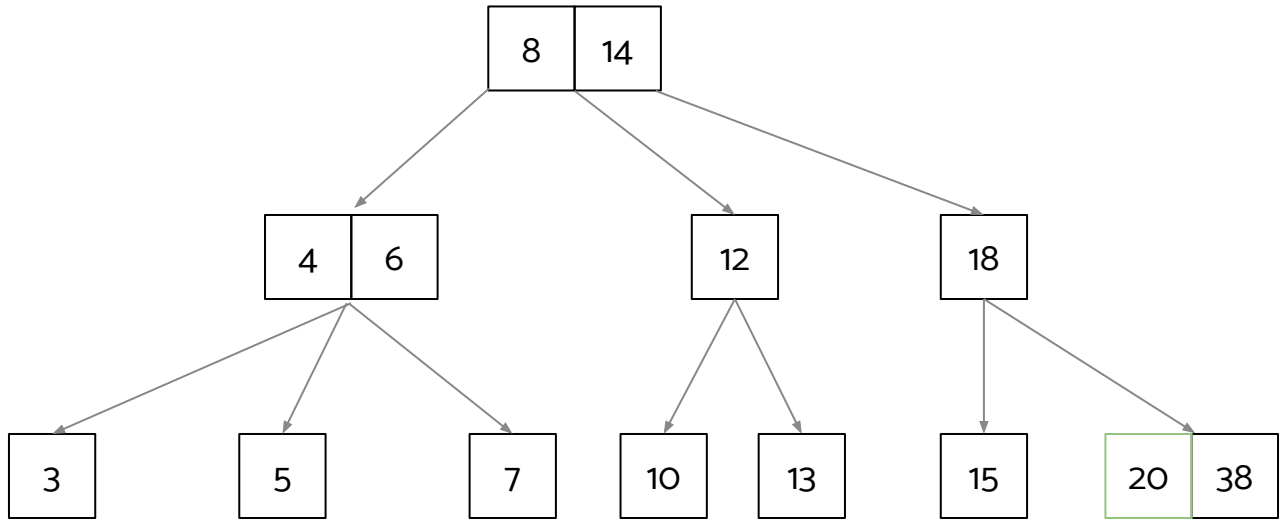
Draw what the following 2-3 tree would look like after inserting 18, 38, 12, 13, and 20.



Inserting 13 (part 3)

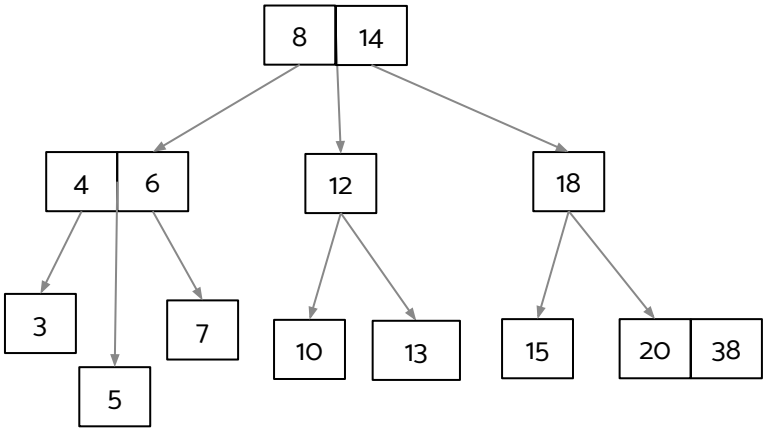
1A 2-3 Trees and LLRBs

Draw what the following 2-3 tree would look like after inserting 18, 38, 12, 13, and 20.



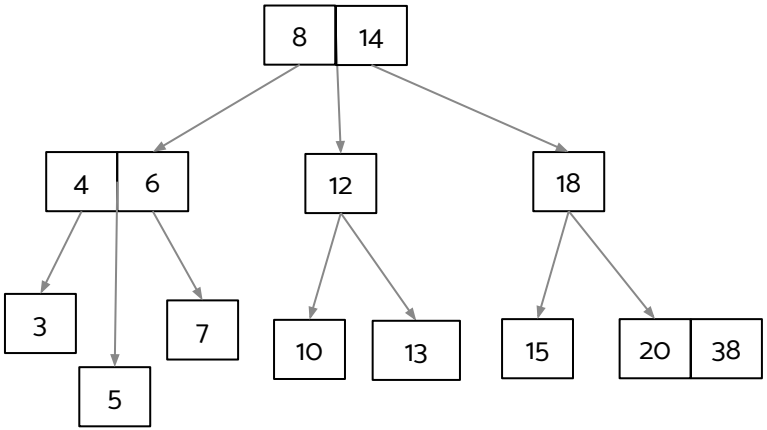
Inserting 20.

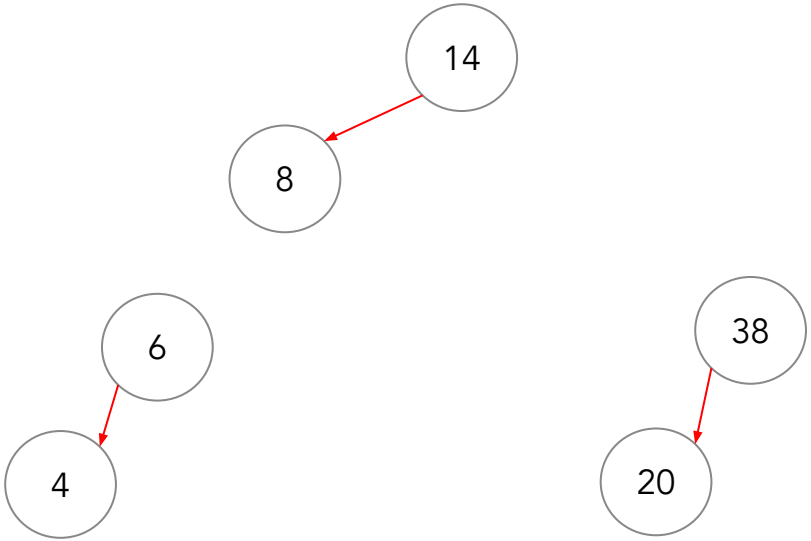
1B 2-3 Trees and LLRBs Convert the resulting 2-3 tree to a red-black tree.



1B 2-3 Trees and LLRBs

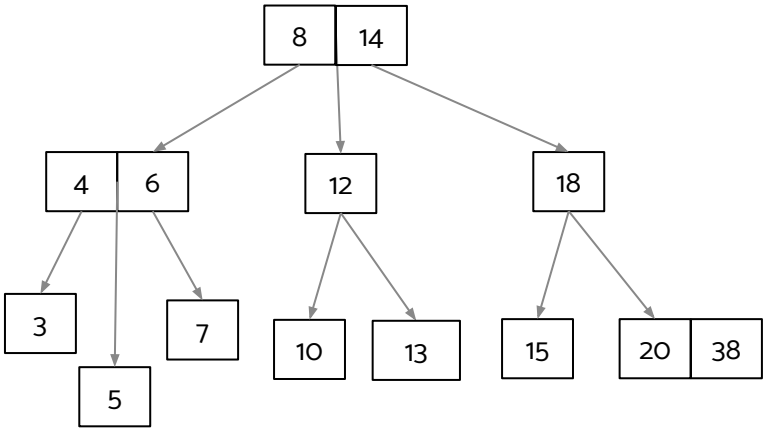
Convert the resulting 2-3 tree to a red-black tree.

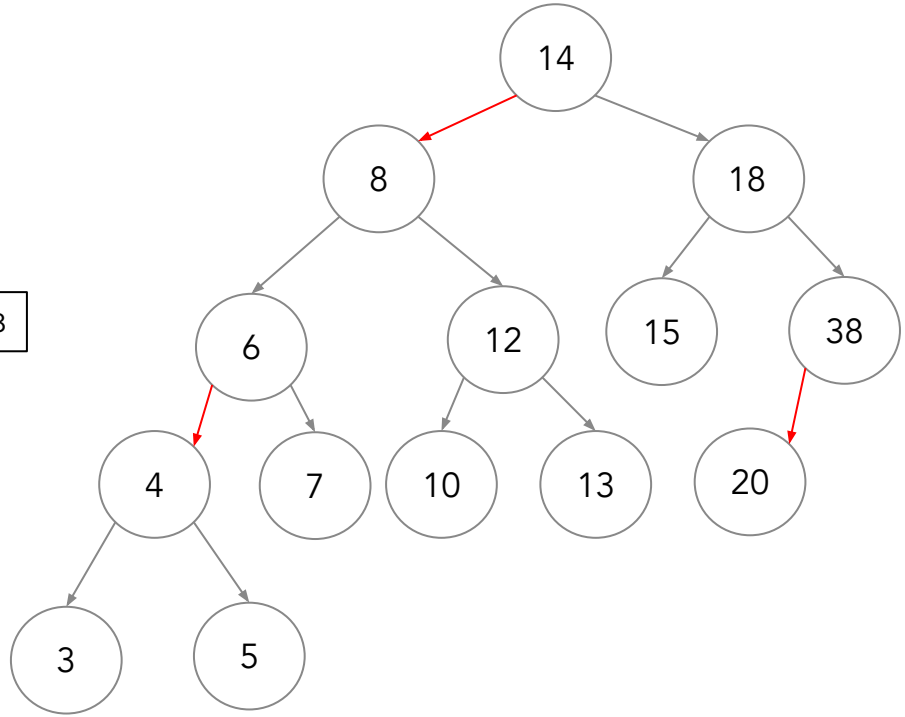




1B 2-3 Trees and LLRBs

Convert the resulting 2-3 tree to a red-black tree.





1C 2-3 Trees and LLRBs

If a 2-3 tree has depth H (that is, the leaves are at distance H from the root), what is the maximum number of comparisons done in the corresponding red-black tree to find whether a certain key is present in the tree?

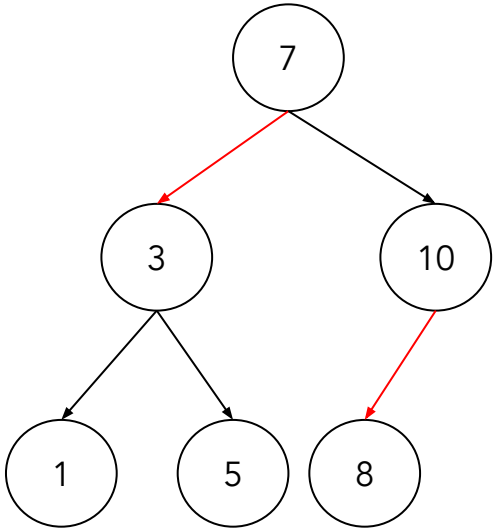
1C 2-3 Trees and LLRBs

If a 2-3 tree has depth H (that is, the leaves are at distance H from the root), what is the maximum number of comparisons done in the corresponding red-black tree to find whether a certain key is present in the tree?

2H + 2 comparisons (longest path from leaf to root * 2 items per node)

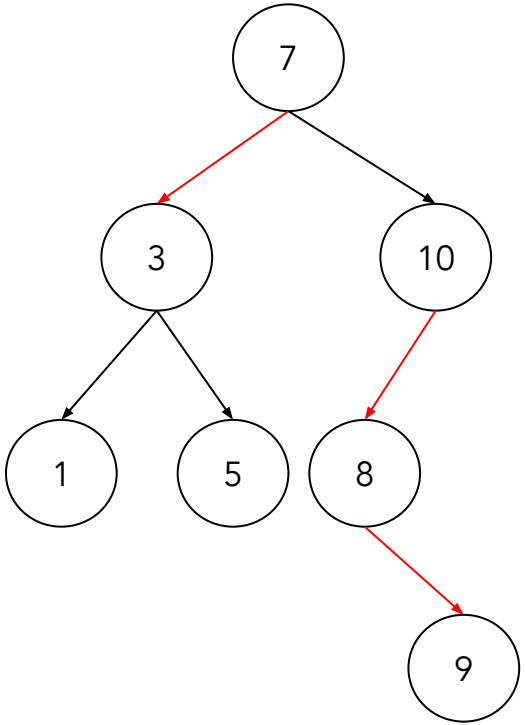
1D 2-3 Trees and LLRBs

Describe all the balancing operations needed to convert the final tree after inserting 9 to an LLRB.



1D 2-3 Trees and LLRBs

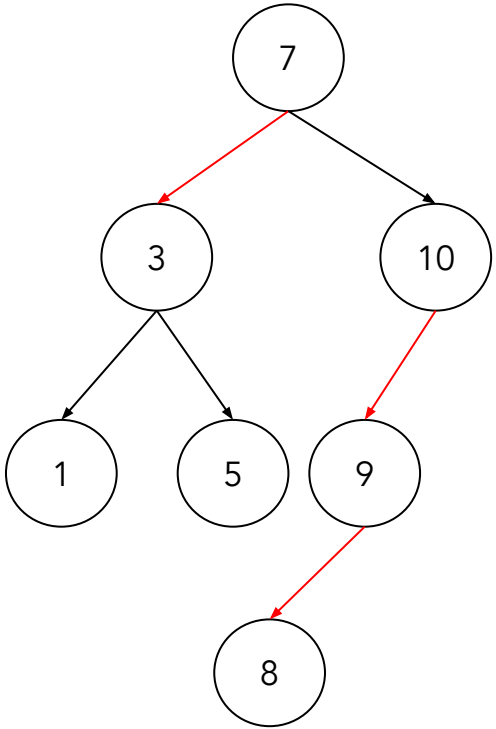
Describe all the balancing operations needed to convert the final tree after inserting 9 to an LLRB.



Initial insertion of 9

1D 2-3 Trees and LLRBs

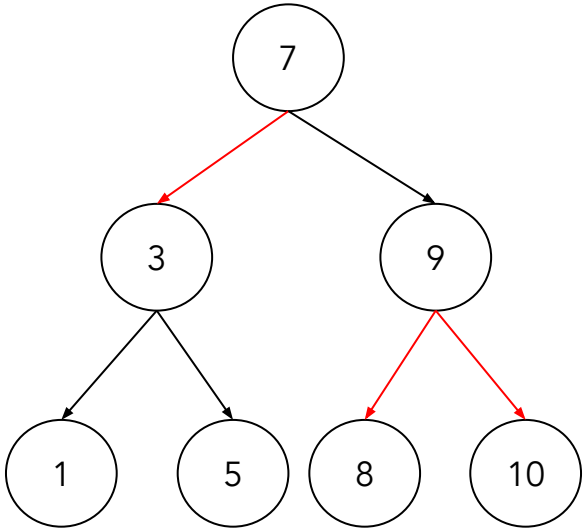
Describe all the balancing operations needed to convert the final tree after inserting 9 to an LLRB.



After `rotateLeft(8)`

1D 2-3 Trees and LLRBs

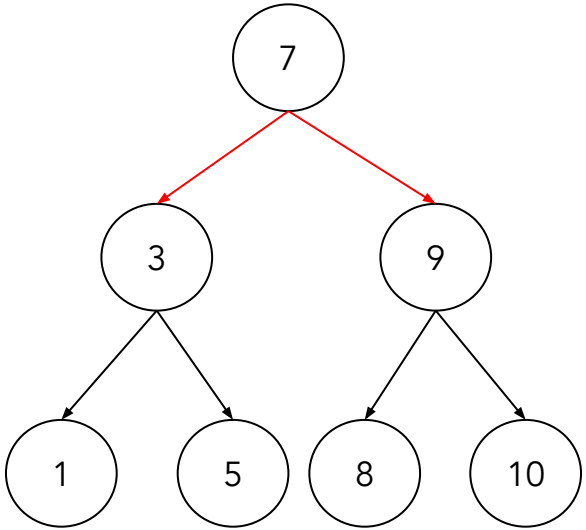
Describe all the balancing operations needed to convert the final tree after inserting 9 to an LLRB.



After `rotateRight(10)`

1D 2-3 Trees and LLRBs

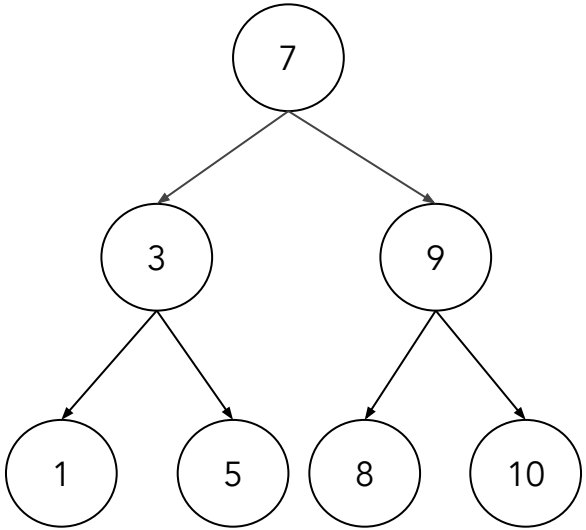
Describe all the balancing operations needed to convert the final tree after inserting 9 to an LLRB.



After `colorFlip(9)`

1D 2-3 Trees and LLRBs

Describe all the balancing operations needed to convert the final tree after inserting 9 to an LLRB.



After `colorFlip(7)`

2A Hashing Are the following hashCodes valid? If they are, what are the advantages or disadvantages?

```
public int hashCode() {
      return -1;
}

public int hashCode() {
      return intValue() * intValue();
}

// Object's hashCode() based on memory location
public int hashCode() {
      return super.hashCode();
}

// return current time as an int
public int hashCode() {
      return (int) new Date().getTime();
}

public int hashCode() {
      return intValue() + 3;
}
```


2A Hashing

Are the following hashCodes valid? If they are, what are the advantages or disadvantages?

```
public int hashCode() {                                    // Valid, but collisions for everything
      return -1;
}

public int hashCode() {
      return intValue() * intValue();
}

// Object's hashCode() based on memory location
public int hashCode() {
      return super.hashCode();
}

// return current time as an int
public int hashCode() {
      return (int) new Date().getTime();
}

public int hashCode() {
      return intValue() + 3;
}
```


2A Hashing

Are the following hashCodes valid? If they are, what are the advantages or disadvantages?

```
public int hashCode() {                                    // Valid, but collisions for everything
      return -1;
}

public int hashCode() {                                    // Valid, but collisions: consider -3 and 3
      return intValue() * intValue();
}

// Object's hashCode() based on memory location
public int hashCode() {
      return super.hashCode();
}

// return current time as an int
public int hashCode() {
      return (int) new Date().getTime();
}

public int hashCode() {
      return intValue() + 3;
}
```


2A Hashing Are the following hashCodes valid? If they are, what are the advantages or disadvantages?

```
public int hashCode() {                                    // Valid, but collisions for everything
      return -1;
}

public int hashCode() {                                    // Valid, but collisions: consider -3 and 3
      return intValue() * intValue();
}

// Object's hashCode() based on memory location
public int hashCode() {                                    // Invalid: memory address unique per Integer
      return super.hashCode();
}

// return current time as an int
public int hashCode() {
      return (int) new Date().getTime();
}

public int hashCode() {
      return intValue() + 3;
}
```


2A Hashing

Are the following hashCodes valid? If they are, what are the advantages or disadvantages?

```
public int hashCode() {                                  // Valid, but collisions for everything
     return -1;
}

public int hashCode() {                                  // Valid, but collisions: consider -3 and 3
     return intValue() * intValue();
}

// Object's hashCode() based on memory location
public int hashCode() {                                  // Invalid: memory address unique per Integer
     return super.hashCode();
}

// return current time as an int
public int hashCode() {                                  // Invalid: non consistent for same object
     return (int) new Date().getTime();
}

public int hashCode() {
     return intValue() + 3;
}
```


2A Hashing

Are the following hashCodes valid? If they are, what are the advantages or disadvantages?

```
public int hashCode() {                                  // Valid, but collisions for everything
      return -1;
}

public int hashCode() {                                  // Valid, but collisions: consider -3 and 3
      return intValue() * intValue();
}

// Object's hashCode() based on memory location
public int hashCode() {                                  // Invalid: memory address unique per Integer
      return super.hashCode();
}

// return current time as an int
public int hashCode() {                                  // Invalid: non consistent for same object
      return (int) new Date().getTime();
}

public int hashCode() {                                  // Valid and good
      return intValue() + 3;
}
```


2B Hashing

1. When you modify a key that has been inserted into a HashMap will you be able to retrieve that entry again? Explain.

2. When you modify a value that has been inserted into a HashMap will you be able to retrieve that entry again? Explain.

2B Hashing

1. When you modify a key that has been inserted into a HashMap will you be able to retrieve that entry again? Explain.

   Sometimes: If the `hashCode()` for the key happens to change as a result of the modification (which is very likely but not guaranteed), then we won't be able to retrieve the entry in our hashtable (unless we were to recompute which bucket the new key would belong to)

2. When you modify a value that has been inserted into a HashMap will you be able to retrieve that entry again? Explain.

2B Hashing

1. When you modify a key that has been inserted into a HashMap will you be able to retrieve that entry again? Explain.

   Sometimes: If the `hashCode()` for the key happens to change as a result of the modification (which is very likely but not guaranteed), then we won't be able to retrieve the entry in our hashtable (unless we were to recompute which bucket the new key would belong to)

2. When you modify a value that has been inserted into a HashMap will you be able to retrieve that entry again? Explain.

   Always: The bucket index for an entry in a `HashMap` is decided by the key, not the value.

3A A Side of Hash Browns Draw the HashMap after the following operations.

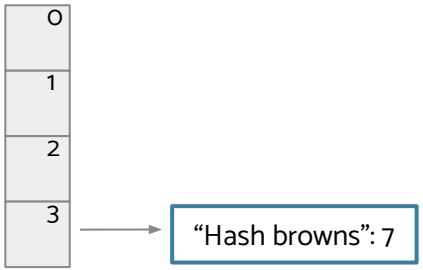
```
HashMap<String, Integer> hm = new HashMap<>();
hm.put("Hashbrowns", 7);
hm.put("Dim sum", 10);
hm.put("Escargot", 5);
hm.put("Brown bananas", 1);
hm.put("Burritos", 2);
hm.put("Buffalo wings", 8);
hm.put("Banh mi", 9);
hm.put("Burritos", 10);
```

Helpful: A = 0, B = 1, D = 3, E = 4, H = 7

3A A Side of Hash Browns

Draw the HashMap after the following operations.

```
HashMap<String, Integer> hm = new HashMap<>();
hm.put("Hashbrowns", 7);
hm.put("Dim sum", 10);
hm.put("Escargot", 5);
hm.put("Brown bananas", 1);
hm.put("Burritos", 2);
hm.put("Buffalo wings", 8);
hm.put("Banh mi", 9);
hm.put("Burritos", 10);
```



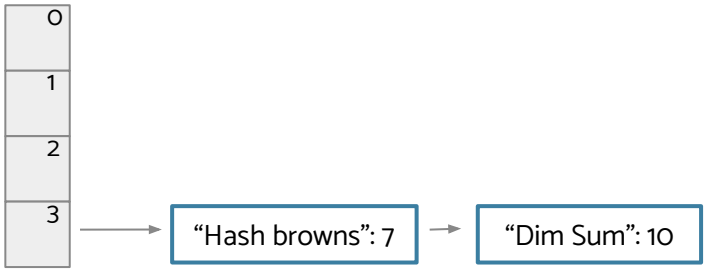

```
N = 1
M = 4
N/M = 0.25
```

Helpful: A = 0, B = 1, D = 3, E = 4, H = 7

3A A Side of Hash Browns

Draw the HashMap after the following operations.

```
HashMap<String, Integer> hm = new HashMap<>();
hm.put("Hashbrowns", 7);
hm.put("Dim sum", 10);
hm.put("Escargot", 5);
hm.put("Brown bananas", 1);
hm.put("Burritos", 2);
hm.put("Buffalo wings", 8);
hm.put("Banh mi", 9);
hm.put("Burritos", 10);
```



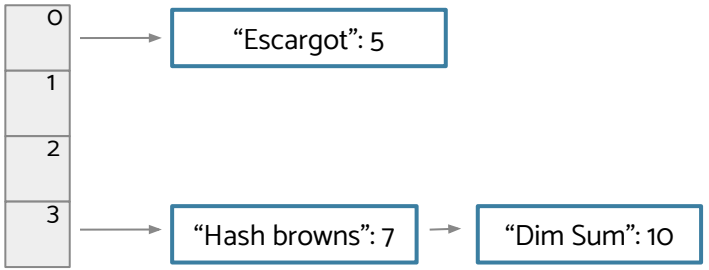

```
N = 2
M = 4
N/M = 0.5
```

Helpful: A = 0, B = 1, D = 3, E = 4, H = 7

3A A Side of Hash Browns

Draw the HashMap after the following operations.

```
HashMap<String, Integer> hm = new HashMap<>();
hm.put("Hashbrowns", 7);
hm.put("Dim sum", 10);
hm.put("Escargot", 5);
hm.put("Brown bananas", 1);
hm.put("Burritos", 2);
hm.put("Buffalo wings", 8);
hm.put("Banh mi", 9);
hm.put("Burritos", 10);
```



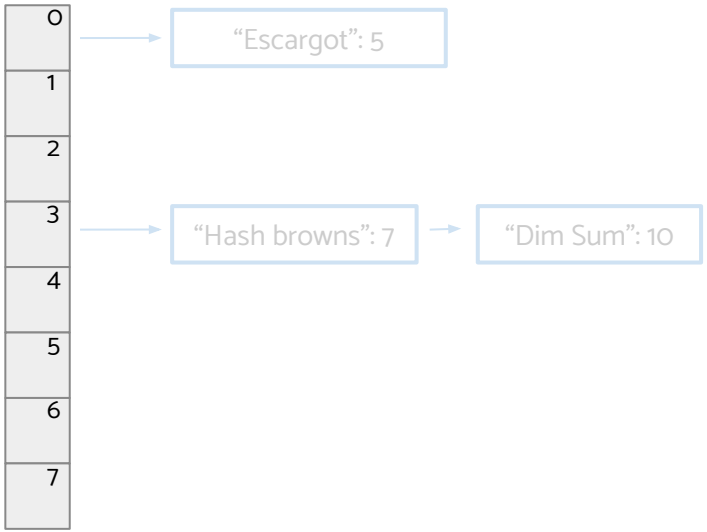

```
N = 3
M = 4
N/M = 0.75
```

Helpful: A = 0, B = 1, D = 3, E = 4, H = 7

3A A Side of Hash Browns

Draw the HashMap after the following operations.

```
HashMap<String, Integer> hm = new HashMap<>();
hm.put("Hashbrowns", 7);
hm.put("Dim sum", 10);
hm.put("Escargot", 5);
hm.put("Brown bananas", 1);
hm.put("Burritos", 2);
hm.put("Buffalo wings", 8);
hm.put("Banh mi", 9);
hm.put("Burritos", 10);
```

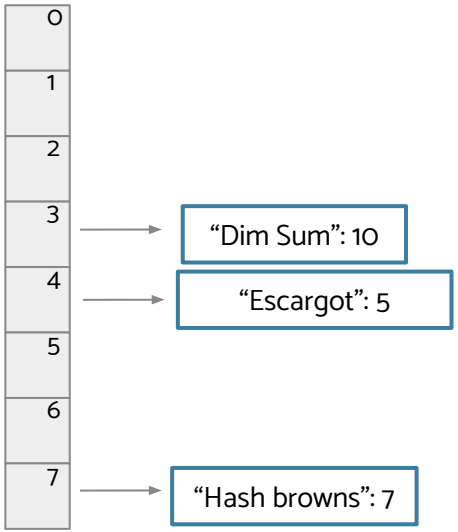


Helpful: A = 0, B = 1, D = 3, E = 4, H = 7

3A A Side of Hash Browns

Draw the HashMap after the following operations.

```
HashMap<String, Integer> hm = new HashMap<>();
hm.put("Hashbrowns", 7);
hm.put("Dim sum", 10);
hm.put("Escargot", 5);
hm.put("Brown bananas", 1);
hm.put("Burritos", 2);
hm.put("Buffalo wings", 8);
hm.put("Banh mi", 9);
hm.put("Burritos", 10);
```



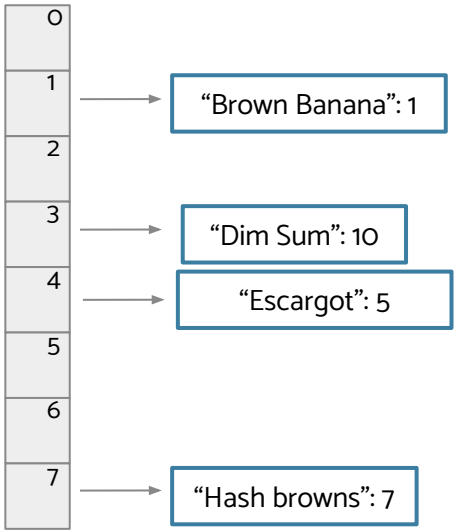
N = 3
M = 8
N/M = 0.375

Helpful: A = 0, B = 1, D = 3, E = 4, H = 7

3A A Side of Hash Browns

Draw the HashMap after the following operations.

```
HashMap<String, Integer> hm = new HashMap<>();
hm.put("Hashbrowns", 7);
hm.put("Dim sum", 10);
hm.put("Escargot", 5);
hm.put("Brown bananas", 1);
hm.put("Burritos", 2);
hm.put("Buffalo wings", 8);
hm.put("Banh mi", 9);
hm.put("Burritos", 10);
```



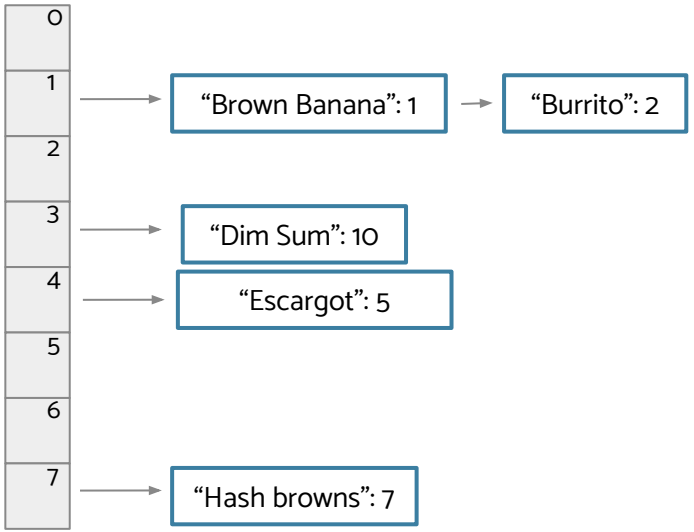

```
N = 4
M = 8
N/M = 0.5
```

Helpful: A = 0, B = 1, D = 3, E = 4, H = 7

3A A Side of Hash Browns

Draw the HashMap after the following operations.

```
HashMap<String, Integer> hm = new HashMap<>();
hm.put("Hashbrowns", 7);
hm.put("Dim sum", 10);
hm.put("Escargot", 5);
hm.put("Brown bananas", 1);
hm.put("Burritos", 2);
hm.put("Buffalo wings", 8);
hm.put("Banh mi", 9);
hm.put("Burritos", 10);
```



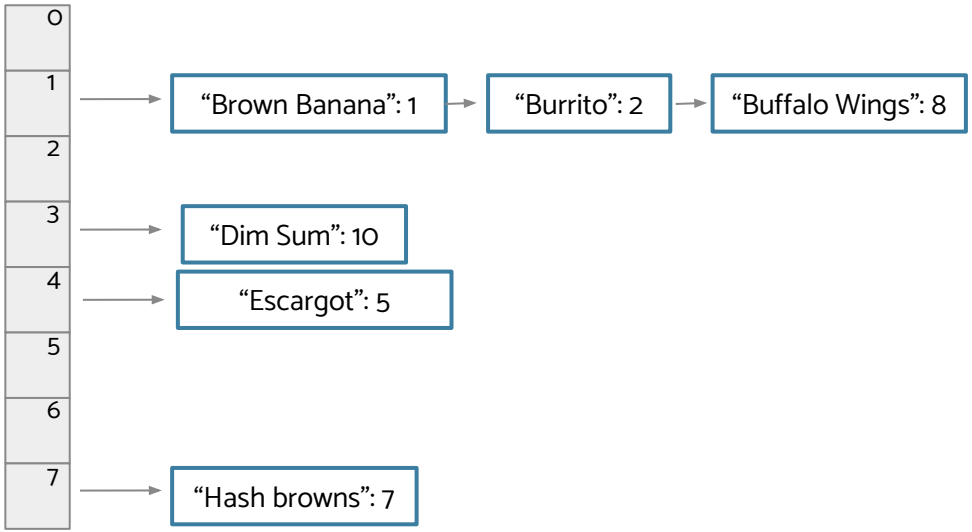
N = 5
M = 8
N/M = 0.625

Helpful: A = 0, B = 1, D = 3, E = 4, H = 7

3A A Side of Hash Browns

Draw the HashMap after the following operations.

```
HashMap<String, Integer> hm = new HashMap<>();
hm.put("Hashbrowns", 7);
hm.put("Dim sum", 10);
hm.put("Escargot", 5);
hm.put("Brown bananas", 1);
hm.put("Burritos", 2);
hm.put("Buffalo wings", 8);
hm.put("Banh mi", 9);
hm.put("Burritos", 10);
```



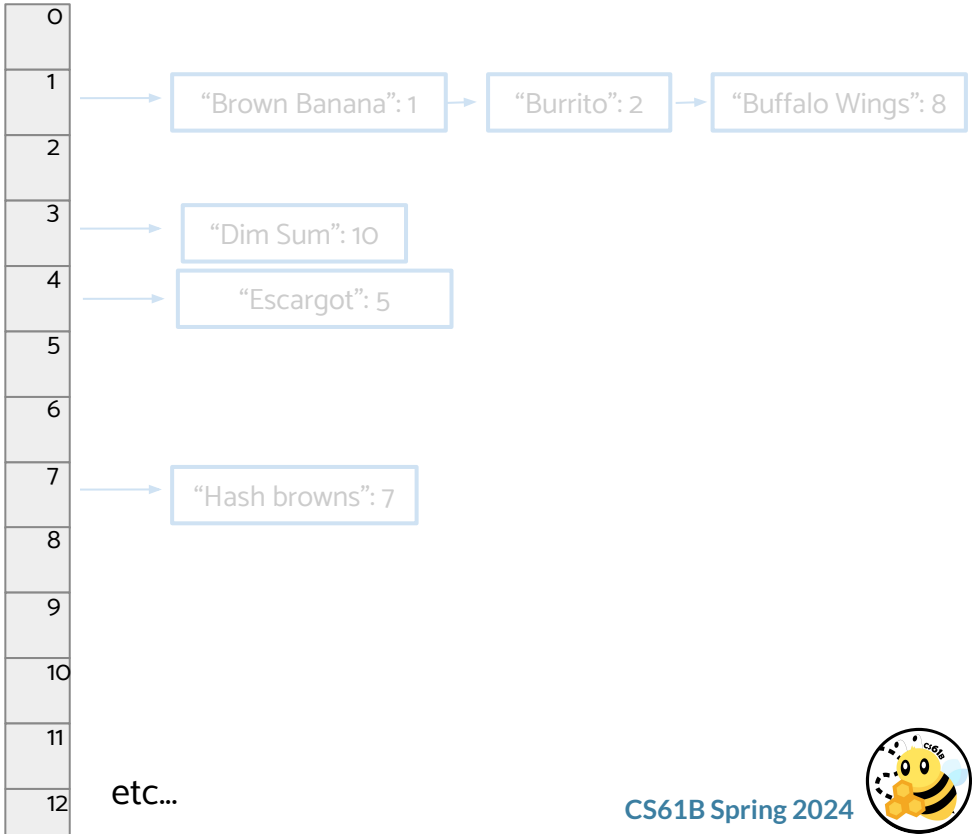
N = 6
M = 8
N/M = 0.75

Helpful: A = 0, B = 1, D = 3, E = 4, H = 7

3A A Side of Hash Browns

Draw the HashMap after the following operations.

```
HashMap<String, Integer> hm = new HashMap<>();
hm.put("Hashbrowns", 7);
hm.put("Dim sum", 10);
hm.put("Escargot", 5);
hm.put("Brown bananas", 1);
hm.put("Burritos", 2);
hm.put("Buffalo wings", 8);
hm.put("Banh mi", 9);
hm.put("Burritos", 10);
```



etc...

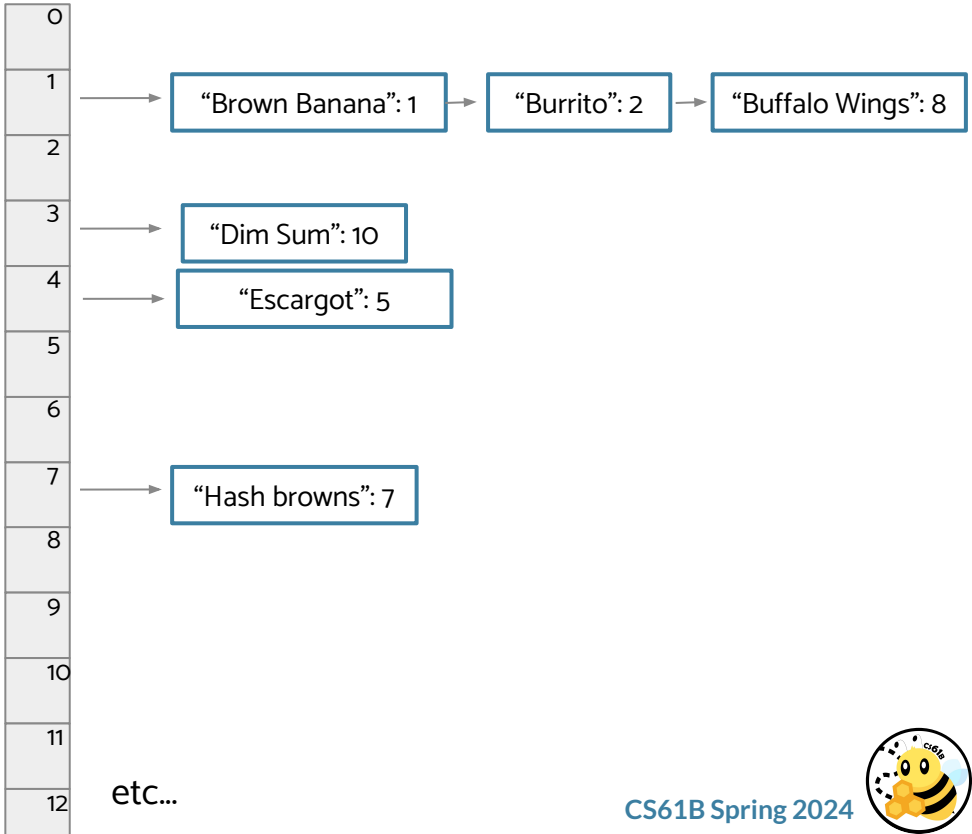
Helpful: A = 0, B = 1, D = 3, E = 4, H = 7

3A A Side of Hash Browns

Draw the HashMap after the following operations.

```
HashMap<String, Integer> hm = new HashMap<>();
hm.put("Hashbrowns", 7);
hm.put("Dim sum", 10);
hm.put("Escargot", 5);
hm.put("Brown bananas", 1);
hm.put("Burritos", 2);
hm.put("Buffalo wings", 8);
hm.put("Banh mi", 9);
hm.put("Burritos", 10);
```

N = 6
M = 16
N/M = 0.375



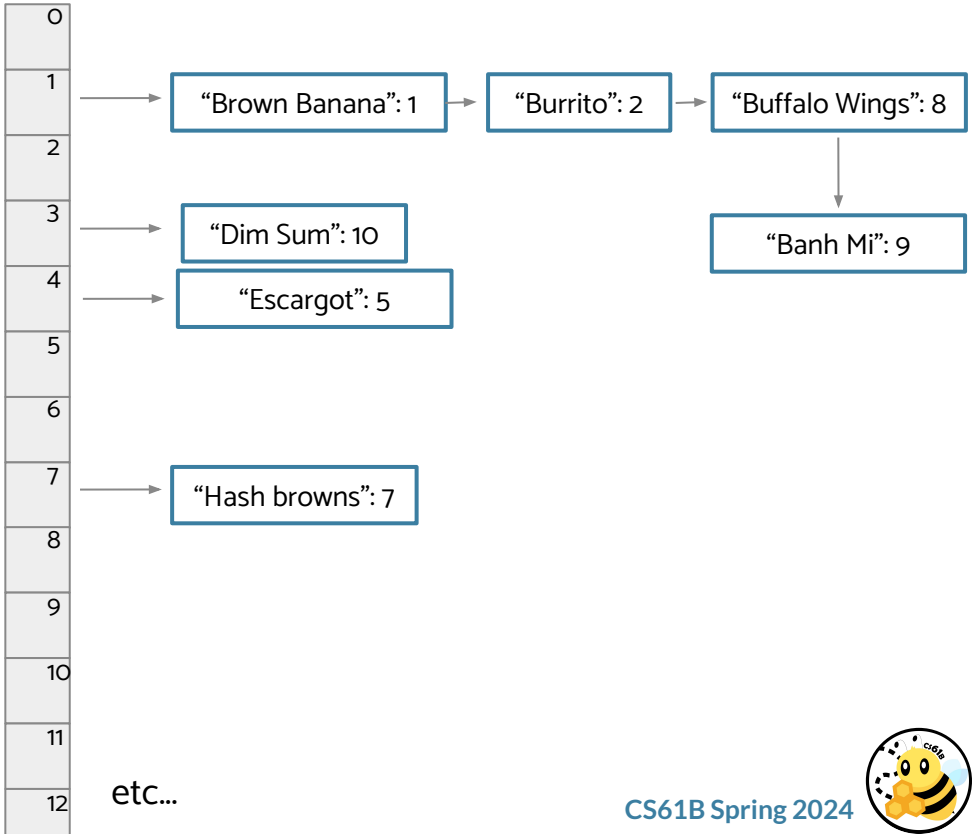
Helpful: A = 0, B = 1, D = 3, E = 4, H = 7

3A A Side of Hash Browns

Draw the HashMap after the following operations.

```
HashMap<String, Integer> hm = new HashMap<>();
hm.put("Hashbrowns", 7);
hm.put("Dim sum", 10);
hm.put("Escargot", 5);
hm.put("Brown bananas", 1);
hm.put("Burritos", 2);
hm.put("Buffalo wings", 8);
hm.put("Banh mi", 9);
hm.put("Burritos", 10);
```

N = 7
M = 16
N/M = 0.4375



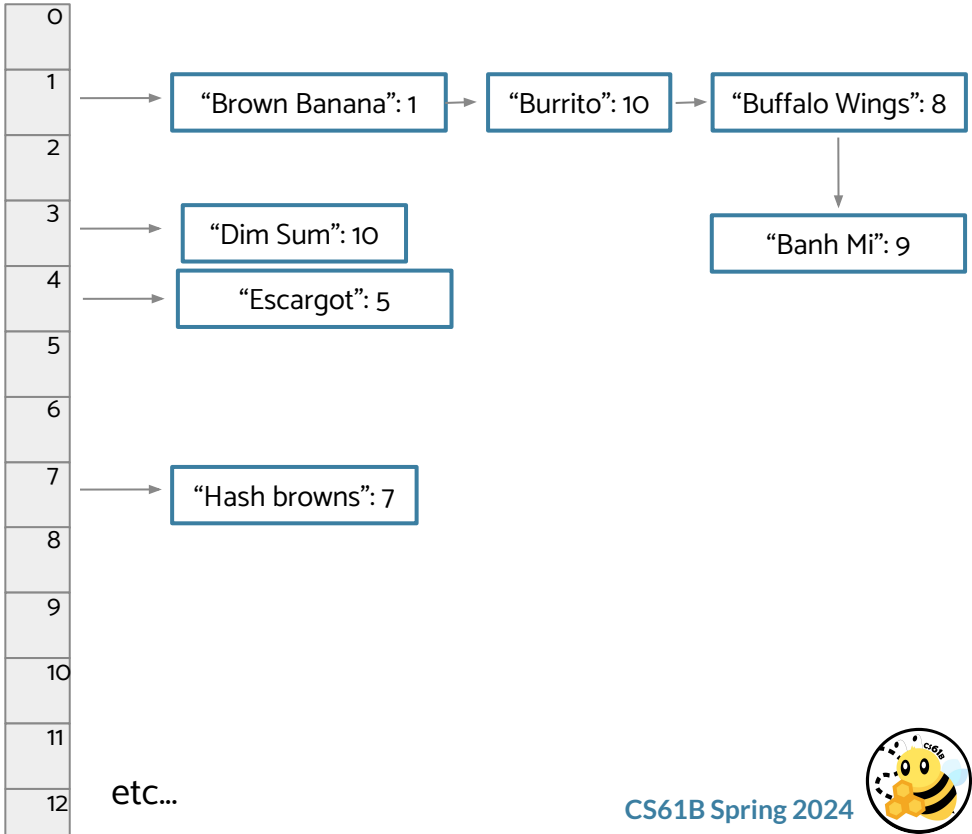
Helpful: A = 0, B = 1, D = 3, E = 4, H = 7

3A A Side of Hash Browns

Draw the HashMap after the following operations.

```
HashMap<String, Integer> hm = new HashMap<>();
hm.put("Hashbrowns", 7);
hm.put("Dim sum", 10);
hm.put("Escargot", 5);
hm.put("Brown bananas", 1);
hm.put("Burritos", 2);
hm.put("Buffalo wings", 8);
hm.put("Banh mi", 9);
hm.put("Burritos", 10);
```

N = 7
M = 16
N/M = 0.4375



Helpful: A = 0, B = 1, D = 3, E = 4, H = 7

3B A Side of Hash Browns

Do you see a potential problem here with the behavior of our `HashMap`? How could we solve this?

3B A Side of Hash Browns

Do you see a potential problem here with the behavior of our `HashMap`? How could we solve this?

Inserting many words with the same first letter results in that letter's bucket growing very large, and our current hashing scheme means that resizing doesn't help us re-distribute the elements. We could solve this by having a better hash function!