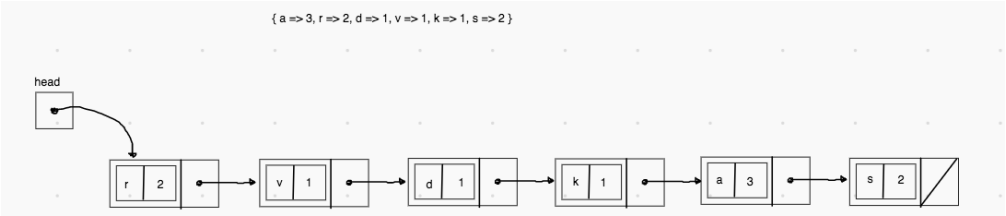
Hashing

Implement the Map API using link chains.

Attempt 1: NaiveMap

Store entries (a key and value pair) in a link chain:




```
Link<Entry<K,V>> head = ... ;

head.element.getKey()
head.element.setValue(element)
```

How do we `put(s,3)`?

1. Find 's' and update the value: that means stepping through the chain and using `.equals()` to determine the link containing that key.
2. Keys that don't appear in the map: we need to check the entire chain!

Upside: it works!

Downside: we might have to step through the entire chain, which will cost us because each time we need to compare keys using `.equals()`.

Attempt 2: Hashing

To make searching for keys more efficient we will group keys into "buckets". We then "hash" the keys into individual buckets based on their "code", which is an integer value that is computed from the key. For example, single characters we will use their ASCII code value.

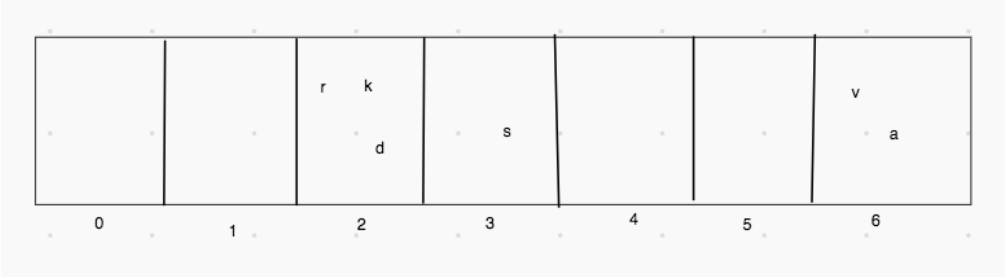
The buckets are numbered from 0 to `buckets.length-1`, so we will use this *hash* function to make sure that code values will be placed inside this range:

```
hash(x) = x % buckets.length;
```

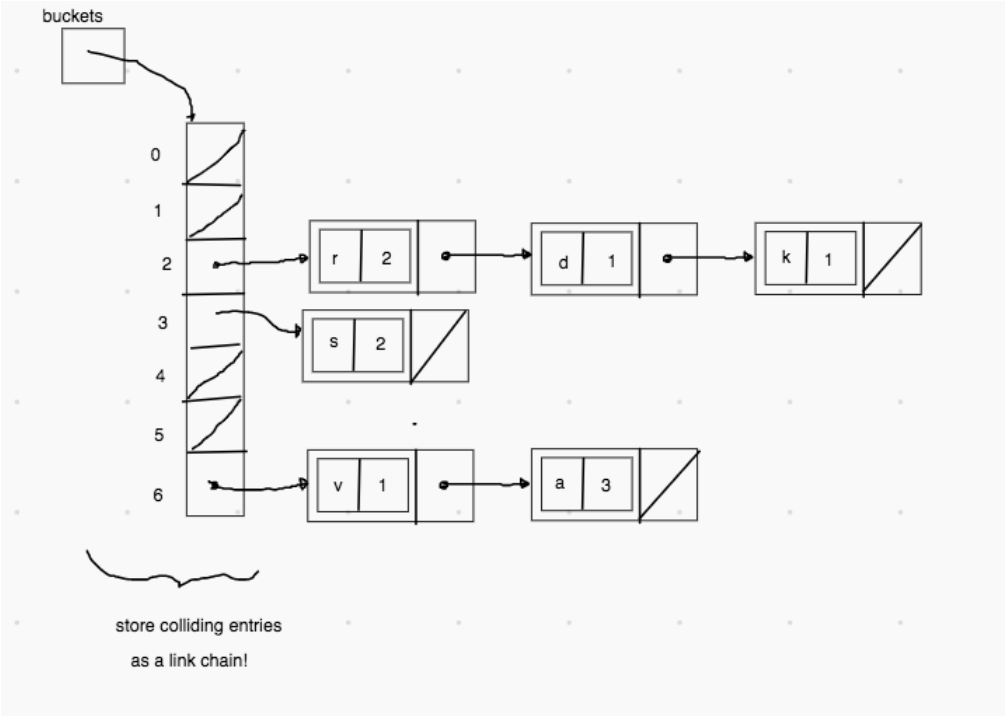
which will produce a value between [0, buckets.length - 1]. Here are the hashed characters from the previous example:

Key	Code	hash(key)
r	114	2
v	118	6
d	100	2*
k	107	2
a	97	6
s	115	3
t	116	4

- keys can hash to the same bucket: called a ”collision”.



Now, instead of a single chain, we have one chain per bucket. The array `buckets` is an array of chain ”heads”:



Ex: `containsKey('t')`

Firstly, hash('t') is 4. Then look in bucket 4 for 't'. Since it's not there, this is false.

Ex: `containsKey('y')`

hash('y') is 2. Check in bucket 2, at most 3 key compares.

Exercise: hashing Strings

In Java, the "code" is actually a call to an object's `.hashCode()` method. It should be implemented for each object used as a key. This means that i in practice hashing in Java is really:

```
hash(key) = key.hashCode() % buckets.length;
```

Attempt 1: constant

```
@Override
```

```
   public int hashCode() {
       return 0;
   }
```

This is terrible because each string will hash to the same bucket, so this is basically the same as `NaiveMap`.

Attempt 2: first character

```
@Override
    public int hashCode() {
        return value[0];
    }
```

Still pretty bad since it will hash to only a small range of values with more collisions on characters that start words and names with higher frequency.

Attempt 3: String length

```
@Override
   public int hashCode() {
       return value.length;
   }
```

Still pretty bad since it will hash to only a small range of values.

Attempt 4: sum the character codes

```
@Override
   public int hashCode() {
       int sum = 0;
       for(char c : value)
           sum += c;
       return sum;
   }
```

Much better since the entire contents of the string were used.

Attempt 5: weighted sum

```
@Override
  public int hashCode() {
      int sum = 0;
      int pos = 1;
      for(char c : value) {
          sum += c * pos;
          pos++;
      }
      return sum;
  }
```

Does a better job that the above, since simple summing give the same values for *permutations* of a strings characters: ”abc”, ”cab”, ”bca”. This is similar to how many hash-code method use *bit-shifting* (« and » operators).

Observations

- Use all the information in the key, since it will generate more variety in the codes generated.
- Incorporate different significance to the positions of the information in the key. This might avoid having the same code for keys whose data are just permutations of each other.

When to implement `hashCode()`?

You should implement `hashCode` when either:

1. If you plan on using your class as a key in a `HashMap` or an element in a `HashSet`.
2. If you are overriding `equals(..)` for your class, since this means it might eventually be used as a key.