OVERVIEW

- **Brute force approach:** All data is just a sequence of bits. Can treat key as a gigantic number and use it as an array index. Requires exponentially large amounts of memory.
- **Hashing:** Instead of using the entire key, represent entire key by a smaller value. In Java, we hash objects with a hashCode() method that returns an integer (32 bit) representation of the object.
- **hashCode() to index conversion:** To use hashCode() results as an index, we must convert the hashCode() to a valid index. Modulus does not work since hashCode may be negative. Taking the absolute value then the modulus also doesn't work since Math.abs(Integer.MIN_VALUE) is negative. Typical approach: use hashCode & 0x7FFFFFFF instead before taking the modulus.
  - Note: Do not return the modulus of the hash in the hashCode() method; delegate this action to the HashMap so that it accounts for resizes in the HashMap.
- **Designing good hash functions:** Requires a blending of sophisticated mathematics and clever engineering; beyond the scope of this course. Most important guideline is to use all the bits in the key. If hashCode() is known and easy to invert, adversary can design a sequence of inputs that result in everything being placed in one bin. Or if hashCode() is just plain bad, same thing can happen.
- **Uniform hashing assumption:** For our analyses below, we assumed that our hash function distributes all input data evenly across bins. This is a strong assumption and never exactly satisfied in practice.
- **External-chaining hash table:** Understand how resizing may lead to objects moving from one linked list to another. Primary goal is so that M is always proportional to N, i.e. maintaining a load factor bounded above by some constant.
- **Resizing separate chaining hash tables:** Understand how resizing may lead to objects moving from one linked list to another. Primary goal is so that M is always proportional to N, i.e. maintaining a load factor bounded above by some constant.
- **Performance of separate-chaining hash tables:** Cost of a given get, insert, or delete is given by number of entries in the linked list that must be examined.
  - The expected amortized search and insert time (assuming items are distributed evenly) is N / M, which is no larger than some constant (due to resizing).
- **Linear-Probing hash tables:** It's where you use empty array entries to handle collisions, e.g. linear probing.

EXAMPLES

Here are some example implementations of External Chaining Hash Tables and Linear Probing Hash Tables:

External Chaining HT:
http://algs4.cs.princeton.edu/34hash/SeparateChainingHashST.java.html

Linear Probing HT:
http://algs4.cs.princeton.edu/34hash/LinearProbingHashST.java.html

<u>Practice Recognizing Good HashCodes</u>

```
// All instance variables
    private int myNum;

// Simple constructor
    public Nana(int n) {
        this.myNum = n;
    }

   public int hashCodeA() {
        /* Math.random() returns a pseudo-random double between 0.0 and 1.0 */
        return (int) (Math.random() * 100);
    }

    public int hashCodeB() {
        return myNum;
    }

    public int hashCodeC() {
        return myNum * myNum * 101 + 17;
    }

    public int hashCodeD() {
        return 17;
    }

    public final double MY_CONST = Math.random();

    public int hashCodeE() {
        return (int) (MY_CONST * 100);
    }

    public int hashCodeF() {
        return Math.abs(myNum);
    }
```

(a) Which function(s) are perfect hash codes? Explain.

(b) Which function(s) are valid hash codes? Explain.

(c) List hashCode functions A-D (don't consider E or F) in order from worst to best. Consider invalid hashCode functions to be worse than all valid functions. Example answer: "A, B, C, D".
(Hint: Remember that poor hash codes have lots of collisions)

(d) What is the problem with the following implementation of the Wug class:

```
class Wug {
        public static int MAP_SIZE = 100;
        public int a;
        public int b;
        public int c;

        public Wug(int a, int b, int c) {
                this.a = a; this.b = b; this.c = c;
        }

        public hashCode() {
                // assume mod is implemented correctly
                return mod((10 * a + 100 * b + 1000 * c), MAPSIZE);
        }

        public static void main(String[] args) {

                Map<String, Wug> map = new HashMap<String, Wug>(MAP_SIZE);
                map.put(“a”, new Wug(1,2,3));
                map.put(“b”, new Wug(3,9,10));
                …
                // an arbitrarily large number of put operations into the map

        }
}
```

(e) Is the following implementation of hashCode() legal?

```
public int hashCode() {
      return 17;
}
```

If so, describe the effect of using it. If not, explain why.

Midterm Level Question

Gorpy McGorpGorp is the founder of GorpyCorp. GorpyCorp is organized into several independent teams known as "Circles", where every Circle has a leader known as a "lead link". Gorpy uses the following data structure to record the `members` and `teamName` of each Circle:

```
public class Circle {
      HashSet<Member> members;
      String teamName;

      public int hashCode() {
            int hashCode = 0;
            for (Member m : members) {
                  hashCode = hashCode * 31 + m.hashCode();
            }
            hashCode = hashCode + teamName.hashCode();
            return hashCode;
      }

      public int compareTo(Circle other) {
            if (this.members.size() == other.members.size())
                  return this.teamName.compareTo(other.teamName);
            return this.members.size() - other.members.size();
      }

      public void addMember(Member newMember) {
            members.add(newMember);
      }

      ...
}
```

Rather than storing the leader of each Circle inside of the Circle object, Gorpy instead decides to create a separate HashMap defined below, which allows a programmer to look up the lead link of each circle.

```
HashMap<Circle, Member> leadLinks;
```

What is the most significant problem with Gorpy's usage of a HashMap? If he uses a TreeMap instead of a HashMap, will this problem be fixed? Why or why not?