EXERCISE 1: Hashing

(a) Consider the following Hashtable API for storing words and their frequencies.

```
public class Hashtable
{
      public void put(String word, int freq)    // inserts a word-frequency pair
      public int get(String word)               // returns the frequency of a word
      public boolean contains(String word)      // checks if a word has a frequency
      public Iterable<String> keys()            // returns all words in the table
      public int size()                         // number of words in the table
      public boolean equals(Object other)
}
```

- Complete the following implementation of the `equals` method. Two Hashtables are equal if and only if they both contain exactly the same words and corresponding frequencies.

```
public boolean equals(Object other) {
      if (other == this) return true;
      if (other == null) return false;
      if (other.getClass() != this.getClass()) return false;
      Hashtable that = (Hashtable) other;















}
```

- Provide the order of growth of the running time for both the *best case* and the *worst case* of your implementation of the `equals` method.
    - Assume that the Hashtable is implemented using Separate Chaining.
    - Express your solution using the number of words in the table (n) and the capacity of the hashtable (m).

(b) Consider the following hash method for mapping a String to an index in the Hashtable.
(Assume that the method argument is guaranteed to be a non-null and non-empty String object)

```
public int hash(String word) {
      return word.charAt(0) % capacity;
}
```

This method is bad for *multiple* reasons. What reasons can you think of?

(c) Java's implementation of Hashtables (HashMap) provides *logarithmic* worst-case performance when the elements are *not* uniformly distributed in the hash table. Can you guess how their implementation achieves this?

FUN FACT: The following hashCode method was used in early versions of the Java String class.

```java
public int hashCode() {
      int hash = 0;
      int skip = Math.max(1, length()/8);
      for (int i = 0; i < length(); i += skip)
            hash = (hash * 37) + charAt(i);
      return hash;
}
```

This was done in the hopes of computing the hash function more quickly. Indeed, the hash values were computed more quickly, but it became more likely that many strings hashed to the same values. This resulted in a significant degradation in performance on many real-world inputs (e.g., URLs) which all hashed to the same value.

For example (underlined characters are included in the computation of the hash value):

```
http://www.mywebsite.com/somedir/page1
http://www.mywebsite.com/somedir/page15
http://www.mywebsite.com/somedir/page200
http://www.mywebsite.com/somedir/page.us
```

EXERCISE 2: Operations on Binary Trees

Consider the following Binary Tree class for storing integers.

```java
public class BinaryTree {

      private Node root;

      private class Node {
            private int key;
            private Node left, right;

            private int N;          // # nodes in subtree rooted here

            private int height;     // maximum number of links between the node and
                                    // any of its children

            public Node(int key, int N) {
                  this.key = key;
                  this.N = N;
            }
      }

      private int size(Node x) { /* returns x.N or 0 if x is null. */ }

      // ... other public and private methods
}
```

(a) Implement the `checkMaxHeap` method to check if the tree is *heap-ordered* as a Max-Heap, i.e. every node in the tree is not less than its children.

Note. checking if a binary tree is a valid Max-Heap requires also checking if every level is full, except the last level, which could be partially filled left-to-right. In this exercise check *only* if the tree is *heap-ordered*.

```
private boolean checkMaxHeap(Node x)
{
      if (x == null) return true;







}
```

(b) Implement the method `allLevelsFull`, which returns true if all levels in the tree are full.

```
// Return true if all the levels in the tree rooted at x are full
private boolean allLevelsFull(Node x)
{
      if (x == null) return true;




}
```

(c) Assume that numbers in `BinaryTree` are ordered such that the tree is a *Binary Search Tree* (BST). Implement the method `select(Node x, int k)`, which returns the node in the tree of rank `k`.

```
// Return Node in tree rooted at x containing key of rank k.
private Node select(Node x, int k)
{
      if (x == null) return null;









}
```