1. Understanding S&W ST classes.

   a. See page 363 for the ST API, shown in homework 2 to be very close to the Map API of the JDK. Write a Java interface ST.java for this API.

      ```java
      public interface ST<Key, Value> {
        int size();
        boolean contains(Key key);
        void put(Key key, Value value);
        Value remove(Key key)
        Value get(Object key);
        Iterable<Key> keys();
      }
      ```

   b. What classes in S&W implement your ST interface? See page 386 on 5 classes in S&W with class names ending in ST, but not in fact simply ST.java, so we can use that name for our interface without causing compatibility problems. They are called "symbol table implementations" in the text on that page. Thus at least in theory they should all implement ST. The first class listed is SequentialSearchST, which we have seen in use for separate chaining hash tables. For the homework answer, put "The 5 classes listed on page 386", unless you disagree with this answer. Do part d. before finalizing this answer.

      Answer: The 5 classes listed on page 386, (except for small differences.)

   c. Give the modified first code line of SequentialSearchST.java (on page 375) that would claim that it implements ST.

      Answer: public class SequentialSearchST<Key, Value> implements ST<Key, Value> {

   Look at the code for SequentialSearchST.java. Does this code (on page 375) satisfy the claim we just made?

      Answer: No What is missing?

      Answer: methods size, contains, remove, keys.

   d. The code for [SequentialSearchST.java at princeton.edu](princeton.edu) has more methods: does it properly implement ST?

      Answer: Almost. It has "remove" instead of "delete", and is missing isEmpty(), but size()
can handle that. Clearly this is the "real" SequentialSearchST.java referred to on page 386.

2. The simplest ST class: SequentialSearchST.java

a. Write size() for the code on page 375. Compute the size by looping down the list (unlike the code at princeton.edu).
```java
public int size() {
    int count = 0;
    Node p = first;
    while (p != null) {
        count++;
        p = p.next;
    }
    return count;
}
```
b. Why doesn't the code at princeton.edu use the method of part a. for size()? Think about its performance.
   Answer: This takes \( T = O(n) \), vs. \( T = O(1) \) for the code at princeton.edu.

c. Draw an object diagram for a SequentialSearchST for the mapping "A" --&gt; "yeah", "B" --&gt; "hohum", "C" --&gt; "yuck". For a model of this diagram, look at one of the boxes containing "first" on pg. 464. See how "first" is in the outer box (the ST object) but outside of the inner boxes, which represent the Nodes. You may show the String refs or not (the diagrams on pg 464 do not show them, but they are really there.

3. Practice on Hash Tables. Given a hash table of size 11, the data is the following for some inventory: part A29 has 21 units, C42 has 4, E12 has 12, D31 has 10, F08 has 9, G34 has 22, and B10 has 92. The hash function is \( h(LN) = ((L - 'A') + N) \mod 11 \), where \( L \) is the letter, \( N \) is the number and 'A' is the ASCII value of A. For example: \( h("C29") = (2 + 29) \mod 11 = 9 \) since 'C' - 'A' = 2.

a. Draw the final configuration of the table after all the elements are inserted into a SeparateChainingHashST. Use the diagram on page 464 as a model. Do not resize.
b. Do the same for linear probing, using the "E 10" part of the diagram on page 469 as a model. Again, do not resize.

4. Dealing with frozen code. Suppose you want to use a HashSet for element objects of a type BigElement implemented by some code you’re not supposed to change, yet doesn’t have a hashCode() implementation. Suppose it has an equals method based on its name field, and getters for all fields. What can you do? Hint: you can put the BigElement into a BiggerElement and put that in the HashSet. Show the code for your BiggerElement class.

Answer:
We can create a new "wrapper" class BigElement1, which has an instance of BigElement as a field. In this class BigElement1, we implement hashCode using the getter for name of BigElement and equals using BigElements equals. We can then use HashSet<BigElement1> safely. When we get a BigElement1 out of the Set, we can extract the original BigElement object from it.

```java
public class BigElement1 {

    private BigElement element;

    BigElement1(BigElement e) {
        element = e;
    }

    public BigElement getElement() { return element; }

    public hashCode() {
        return getName().hashCode();
    }

    public equals(Object o) {
        if (this == o) return true; // as on Pg. 103
    
        if (o == null) return false;
        if (this.getClass() != o.getClass())
```
return false;
else BigElement1 other = (BigElement1)o;

return element.equals(other.element); // use equals of BigElement

// or
// return getName().equals(other.getName()); // or mimic it

5. Practice on inner classes, needed for clean HashSet/HashMap implementations and other container classes. Consider the Student class provided in the PriorityQueue.zip code example. Enhance it to have methods addBook(String title, String ISBN) and boolean hasBook(String ISBN) by designing a private inner class Book (full name Student.Book) to hold a book owned by the student, and adding a Collections class field to Student to hold all the books owned by the student. In this case Book never gets used outside Student. Consider if you need equals(), etc. for Book. Show the complete Student.java in your homework paper.

a. /* Like Orders, Students are ordered on basis of id only. */
   import java.lang.Integer;
   import java.util.HashSet;
   import java.util.Set;
   class Student implements Comparable<Student> {
      private int id;
      private String name;
      private double gpa;
      private Set<Book> books = new HashSet<Book>(); // or use HashMap<String, Book>
      private class Book {
         String title;
         String isbn;
         Book(String title, String isbn) {
            this.title = title;
            this.isbn = isbn;
         }
         public String toString() { return isbn + ": " + title; } // not required
         // The following is needed to ensure that the set doesn't have duplicates
         // It assumes that an ISBN uniquely id's a book, the intent of an ISBN
         // Alternatively, use both ISBN and title to id a book, since ISBN uniqueness
         // was not specified.
         @Override // not required, but a good idea
         public boolean equals(Object other) {
            // code on pg. 103, adapted
            if (this == other) return true; // this line not required
            if (other == null) return false;
            if (this.getClass() != other.getClass())
               return false;
            Book o = (Book)other;
            return o.isbn.equals(other.isbn); // Note == here would be wrong
public Student(String n, int i, double gpa) { name = n; id = i;
this.gpa = gpa;}

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public double getGpa() {
    return gpa;
}

public void setGpa(double gpa) {
    this.gpa = gpa;
}

public void addBook(String title, String ISBN) {
    Book book = new Book(title, ISBN);
    books.add(book);
}

public boolean hasBook(String ISBN) {
    for (Book b: books) {
        if (b.isbn.equals(ISBN)) {
            return true;
        }
    }
    return false;
}

public String getBookTitle(String ISBN) {
    for (Book b: books) {
        if (b.isbn.equalsIgnoreCase(ISBN)) {
            return b.title;
        }
    }
    return "";
}

public boolean equals(Object rhs) { // more compact code, but
    // still correct
    if (rhs == null || getClass() != rhs.getClass())
        return false;
    Student other = (Student) rhs;
    return id == other.id;
}
public int compareTo(Student other) {
    return Integer.valueOf(id).compareTo(Integer.valueOf(other.id));
}

public int hashCode() {
    return Integer.valueOf(id).hashCode();
}

public String toString() {
    return "(" + name + " " + id + " " + gpa + " " + books + ")";
}

public static void main (String args[]) {
    // main not required
    Student student = new Student("Joe", 100, 3.0);
    student.addBook("Algorithms", "1234");
    student.addBook("Java", "3456");
    System.out.println("has 1234? " + student.hasBook("1234"));
    System.out.println("has 3456? " + student.hasBook("3456"));
    System.out.println(student.getBookTitle("1234"));
    System.out.println(student.getBookTitle("3456"));
    System.out.println(student);
}

6. Designing a method. Propose a new method for your enhanced Student class of
problem 5 so that given a Student object “student”, a program can find out the title of a
book for which student.hasBook(isbn) returns true. Show the method code and the calls
to it from a test program.

See above, for method student.getBookTitle(isbn) and test code in main

7. Instant Resume Scorer. Many companies use computer scoring systems for resumes,
hopefully smarter than this. Take the job description in one text file, the resume in
another (both without HTML or other markup), and compute the score as defined here:

JD1 = set of words of job description each longer than 6 characters

JD2 = set of words of job description in all upper case, at least 2 characters (acronyms,
like HTTP)

R1 = set of words of resume each longer than 6 characters

R2 = set of words of resume in all upper case, at least 2 characters

Then score = size of (JD1 intersect R1) + size of (JD2 intersect R2)

a. Write pseudocode for this program, mentioning JDK Collections classes in use. Read
each file only once. You do not need to "materialize" all these sets (represent them in
memory) as long as you compute the right scores. Show the pseudocode in your
homework submission.

Create JD1 and JD2 in 2 HashSet<String>s: (or TreeSet<String>):
    Create empty HashSet<String> JD1
    Create empty HashSet<String> JD2
    Use a Scanner to parse words from the job description
For each word:
    if its length > 6, add to JD1 (JD1.add(word))
    if word.toUpperCase().equals(word), add to JD2

Process a resume:
    score = 0
    Use a Scanner to parse words out of a resume:
        for each word: 0(N) passes
            if word's length > 6 and word in JD1 (JD1.contains(word)),
                score++  O(1)/O(logN)
            if word.toUpperCase().equals(word) and word in JD2,
                score++  O(1)/O(logN)
T(N) = O(N) for HashMap, O(NlogN) for TreeSet

Alternatively: follow the set intersection and union formula
Create 2 HashSet<String> for R1 and R2
Use a Scanner to parse words out of a resume:
    for each word: 0(N) passes
        if word's length > 6, add to R1  O(1)
        if word.toUpperCase().equals(word), add to R2  O(1)
Compute JD1 intersect R1 (by R1.retainAll(JD1) to keep JD1 intact)  O(N)
Compute JD2 intersect R2 (... ) O(N)
T(N) = O(N) + O(N) + O(N) = O(N), but expect slower than earlier approach.

b. Once you have processed the job description of N words, you should be able to process a resume of O(N) words in O(N) time or O(NlogN) time: show an analysis of this for your algorithm.

8. A Programming Tool. (optional, 0 points) From Weiss, Problem 20.20. A Map app. A BASIC program consists of a series of statements, each of which is numbered in ascending order. Control is passed by use of a GOTO or GOSUB and a statement number to go to. We want a program that reads a legal BASIC program and renumbers the statements so that the first starts at a line number F and each statement has a number D higher than the previous statement. The statement numbers in the input are 32-bit integers (like Java), and you may assume that the renumbered statements will fit into a 32-bit number. Your program must run in linear time. Do NOT write the program, just give pseudocode and describe how it works. Here is a little BASIC program to think about. A GOSUB would be of the same form as the corresponding GOTO, so you can ignore them for simplicity.

Let N = number of lines in program, the size of this problem
oldn = old line no.
newn = new line no.

create a map M, a HashMap<Integer, Integer>, for mapping old to new line numbers
newn = F
for each line in the file N passes
    oldn = get_number_from_line()  O(1)
    M.put(key = oldn, value = newn)  O(1)
    newn += D
    /* read the file again, or save the lines in memory above */
for each line in the file   N passes
newn = M.get(key = oldn)   O(1)
if line contains 'GOTO x'   O(1)
   newx = M.get(x)   O(1)
   change 'GOTO x' -> 'GOTO newx'   O(1)
write line with lineno
newn to new file   O(1)

We see two loops with N passes and O(1) bodies, so O(N) + O(N) = O(N)
overall, as required.