Programming HW#5 Java Code File (PDF VERSION, for viewing highlights, BUT DON'T COPY AND PASTE FROM THE PDF VERSION, ONLY the .rtf version!)

Include the following interfaces and classes (in which some need to be completed or changed) in your programs (parts to complete and/or change are highlighted), as well as the LList.java file in the Hash Code File folder under Week 6:

```
public interface Hasher<E> {
    public int hash(E elem);
}

// Note that that Comparator<E> interface used below is in the Java library in java.util,
//    so import java.util.*; when you use the Comparator
// ALSO, the Iterator<E> interface you'll use is in java.util
//-----------------------------------------------------------------------------
// HashTable interface definition:

import java.util.*;

public abstract class HashTable<E> {
       protected int numCollisions; // how many collisions since instantiating or rehashing
       protected int longestCollisionRun;// longest run for ONE entry or longest linked list
       protected Hasher<E> hasher;
       protected Comparator<E> comparator;

       public HashTable(Hasher<E> h, Comparator<E> c)
       {
            hasher = h;
            comparator = c;
       }

       public abstract E getEntry(E target);
       public abstract boolean contains( E x);
       public abstract void makeEmpty();
       public abstract boolean insert( E x);
       public abstract boolean remove( E x);
       public abstract int size();
       public abstract boolean setMaxLambda( double lam );
       public abstract Iterator<E> iterator();
       public abstract void displayTable();
       public abstract void displayStatistics();
}

//-----------------------------------------------------------------------------
// HashQP Class Definition
import java.util.*;

// HashQP class ------------------------
public class HashQP<E> // CHANGE TO MAKE THIS A SUBCLASS OF HashTable for HW#5!!!!!!!!!!
{
   protected static final int ACTIVE = 0;
   protected static final int EMPTY = 1;
   protected static final int DELETED = 2;

   static final int INIT_TABLE_SIZE = 97;
   static final double INIT_MAX_LAMBDA = 0.49;

   protected HashEntry<E>[] mArray;
   protected int mSize;
   protected int mLoadSize;
```

```
   protected int mTableSize;
   protected double mMaxLambda;

   // public methods --------------------------------
   public HashQP(int tableSize) // ADD Comparator<E> and Hasher<E> parameters for HW#5!!!!!!!!
   {
       // PASS corresponding parameters to Comparator<E> and Hasher<E> parameters of the
superclass constructor!!!!!


      mLoadSize = mSize = 0;
      if (tableSize < INIT_TABLE_SIZE)
         mTableSize = INIT_TABLE_SIZE;
      else
         mTableSize = nextPrime(tableSize);

      allocateArray();  // uses mTableSize;
      mMaxLambda = INIT_MAX_LAMBDA;
   }

   public HashQP()// ADD Comparator<E> and Hasher<E> parameters for HW#5!!!!!!!!
   {
      this(INIT_TABLE_SIZE); // FIX THIS (also pass Comparator<E> and Hasher<E>)
   }

   public boolean insert( E x)
   {
      int bucket = findPos(x);

      if ( mArray[bucket].state == ACTIVE )
         return false;

      mArray[bucket].data = x;
      mArray[bucket].state = ACTIVE;
      mSize++;

      // check load factor
      if( ++mLoadSize > mMaxLambda * mTableSize )
         rehash();

      return true;
   }

   public boolean remove( E x )
   {
      int bucket = findPos(x);

      if ( mArray[bucket].state != ACTIVE )
         return false;

      mArray[bucket].state = DELETED;
      mSize--; // mLoadSize not dec'd because it counts any non-EMP location
      return true;
   }

   public boolean contains(E x )
   {
      return mArray[findPos(x)].state == ACTIVE;
   }

   public int size()  { return mSize; }
```

```
   public void makeEmpty()
   {
      int k, size = mArray.length;

      for(k = 0; k < size; k++)
         mArray[k].state = EMPTY;
      mSize = mLoadSize = 0;
   }

   public boolean setMaxLambda( double lam )
   {
      if (lam < .1 || lam > INIT_MAX_LAMBDA )
         return false;
      mMaxLambda = lam;
      return true;
   }

   public void displayStatistics() // NEW WITH HW#5 (you'll call this in main)
   {
       System.out.println("\nIn the HashQP class:\n");
       System.out.println( "Table Size = " +  mTableSize );;
       System.out.println( "Number of entries = " + mSize);
       System.out.println( "Load factor = " + (double)mSize/mTableSize);
       System.out.println( "Number of collisions = " + this.numCollisions );
       System.out.println( "Longest Collision Run = " + this.longestCollisionRun );
   }

//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
// DON'T FORGET TO OVERRIDE iterator() (YOU WRITE FOR HW#5)
//    just return a new instance of the HashQPIterator class below
//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
// DON'T FORGET TO OVERRIDE displayTable() (YOU WRITE FOR HW#5)
// FOR EACH ARRAY ELEMENT...
//      if the element isn't ACTIVE, don't display anything
//      if the element IS ACTIVE, display the ARRAY INDEX and the
//          data of the entry (toString)
//      (see the test runs for examples)
//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   public E getEntry(E target)
   {
         // FINISH THIS (should be like remove, but return
         //    the data from the ACTIVE found entry OR null if not found
   }

   // protected methods of class ----------------------

   protected int findPos( E x )
   {
      int kthOddNum = 1;
      int index = myHash(x);

      while ( mArray[index].state != EMPTY
         && !mArray[index].data.equals(x) ) // CHANGE TO USE Comparator's compare for
HW#5!!!!!!!!!!
      {
         index += kthOddNum; // k squared = (k-1) squared + kth odd #
```

```
            kthOddNum += 2;      // compute next odd #
            if ( index >= mTableSize )
               index -= mTableSize;
            ++numCollisions; // ***************** FOR EX. 8.2 **********************
             // ADD HERE: update local counter variable for HW#5!!!
         }

       // ADD HERE: maybe update longestCollisionRun variable for HW#5!!!!!!!


         return index;
      }

      protected void rehash()
      {
          numCollisions = 0; // ***************** FOR EX. 8.2 **********************
       // ADD CODE HERE TO RESET THE HashTable longestCollisionRun TO 0 for HW#5!!!!

         // we save old list and size then we can reallocate freely
         HashEntry<E>[] oldArray = mArray;
         int k, oldTableSize = mTableSize;;

         mTableSize = nextPrime(2*oldTableSize);

         // allocate a larger, empty array
         allocateArray();  // uses mTableSize;

         // use the insert() algorithm to re-enter old data
         mSize = mLoadSize = 0;
         for(k = 0; k < oldTableSize; k++)
            if (oldArray[k].state == ACTIVE)
               insert( oldArray[k].data );
      }

      protected int getNumCollisions(){ return numCollisions; }// ***************** FOR EX. 8.2
**********************

      protected int myHash(E x)
      {
         int hashVal;

         hashVal = x.hashCode() % mTableSize; // CHANGE TO USE Hasher's hash method for
HW#5!!!!!!!!!!!
         if(hashVal < 0)
            hashVal += mTableSize;

         return hashVal;
      }

      protected static int nextPrime(int n)
      {
         int k, candidate, loopLim;

         // loop doesn't work for 2 or 3
         if (n <= 2 )
            return 2;
         else if (n == 3)
            return 3;

         for (candidate = (n%2 == 0)? n+1 : n ; true ; candidate += 2)
         {
```

```
         // all primes > 3 are of the form 6k +/- 1
         loopLim = (int)( (Math.sqrt((double)candidate) + 1)/6 );

         // we know it is odd.  check for divisibility by 3
         if (candidate%3 == 0)
            continue;

         // now we can check for divisibility of 6k +/- 1 up to sqrt
         for (k = 1; k <= loopLim; k++)
         {
            if (candidate % (6*k - 1) == 0)
               break;
            if (candidate % (6*k + 1) == 0)
               break;
         }
         if (k > loopLim)
            return candidate;
      }
   }

   void allocateArray()
   {
      int k;

      mArray = new HashEntry[mTableSize];
      for (k = 0; k < mTableSize; k++)
         mArray[k] = new HashEntry<E>();
   }

   // INNER CLASS: HashEntry, used ONLY internally in HashQP----------------------
   class HashEntry<E>
   {
      public E data;
      public int state;

      public HashEntry( E x, int st )
      {
         data = x;
         state = st;
      }

      public HashEntry()
      {
         this(null, HashQP.EMPTY);
      }

    }
//-------------------------------------------------------------------------------
// WRITE THE HashQPIterator<E> class (inner class in HashQP, so it WON'T be public)
//     This class MUST implement Iterator<E>, AND include the following:
//          private instance variable for HashEntry<E>[] , and an int (for current index)
//          constructor with a HashEntry<E>[] parameter in which you assign the parameter
//              the corresponding instance variable
// Override methods:
//       boolean hasNext()
//          find the next current index (inside the array) where the HashEntry's state
//              is ACTIVE (loop)
//          if the current index is < array's length return true
//          otherwise return false
//       E next()
//          if hasNext() returns true, return the data at the current index's HashEntry
```

```
//              (and increment the current index before returning)
//          else return false

} // end HashQP class

//----------------------------------------------------------------------------------------

import java.util.*;

public class HashSC<E> // CHANGE TO MAKE THIS A SUBCLASS OF HashTable for HW#5!!!!!!!!!!
{
   static final int INIT_TABLE_SIZE = 97;
   static final double INIT_MAX_LAMBDA = 1.5;

   protected LList<E>[] mLists;
   protected int mSize;
   protected int mTableSize;
   protected double mMaxLambda;


   public HashSC(int tableSize) // ADD Comparator<E> and Hasher<E> parameters for HW#5!!!!!!!!
   {
       // Pass Comparator<E> and Hasher<E> parameters to the SUPERCLASS constructor for
HW#5!!!!!!!!!!


      if (tableSize < INIT_TABLE_SIZE)
         mTableSize = INIT_TABLE_SIZE;
      else
         mTableSize = nextPrime(tableSize);

      allocateMListArray();  // uses mTableSize;
      mMaxLambda = INIT_MAX_LAMBDA;
   }

   public HashSC()// ADD Comparator<E> and Hasher<E> parameters for HW#5!!!!!!!!
   {
      this(INIT_TABLE_SIZE); // FIX THIS (also pass Comparator<E> and Hasher<E>)
   }

//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
// DON'T FORGET TO OVERRIDE iterator() (YOU WRITE FOR HW#5)
//    just return a new instance of the HashQPIterator class below
//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
// DON'T FORGET TO OVERRIDE displayTable() (YOU WRITE FOR HW#5)
// FOR EACH ARRAY ELEMENT...
//   if the linked list at that element is empty, don't display anything
//   otherwise, display the ARRAY INDEX AND the
//       data in each linked list node all on ONE line, BUT
//       YOU MUST USE THE ITERATOR RETURNED FROM EACH LINKED LIST
//       to retrieve each Node's data (YOU ARE NOT ALLOWED TO CALL getEntry)
//       (see the test runs for examples)
//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   public E getEntry(E target)
   {
         // FINISH THIS: should be like remove, (SO YOU USE A LinkedList's iterator
         //   AND Comparator), but return what the iterator returned if the comparator's compare
```

```
        //   method returns 0  OR null if not found
    }

    public boolean contains( E x)
    {
       LList<E> theList = mLists[myHash(x)];

       return theList.contains(x); // CHANGE AS INDICATED ON HW#5!!!!!!!!!!!!!!!
                          // replace with SEVERAL LINES (USE ITERATOR AND COMPARATOR)
    }

    public void makeEmpty()
    {
       int k, size = mLists.length;

       for(k = 0; k < size; k++)
          mLists[k].clear();
       mSize = 0;
    }

    public boolean insert( E x)
    {
       LList<E> theList = mLists[myHash(x)];

       if ( theList.contains(x) ) // CHANGE AS INDICATED ON HW#5!!!!!!!!!
             // replace with SEVERAL LINES (USE ITERATOR AND COMPARATOR)
          return false;
          // ADD HERE: check and maybe UPDATE member counter variable


       // not found so we insert
       theList.add(x);
          // ADD HERE: possibly update longestCollisionRun variable
          //    which should be counting the longest linked list

       // check load factor
       if( ++mSize > mMaxLambda * mTableSize )
          rehash();

       return true;
    }

    public boolean remove( E x)
    {
        LList<E> theList = mLists[myHash(x)];
        Iterator<E> iter = theList.iterator();
        E currElem;

        for(int i=0; iter.hasNext(); ++i )
        {
             currElem = iter.next();
             if(comparator.compare(currElem, x)==0)
             {
                theList.remove(i+1);
                --mSize;
                return true;
             }
        }

       // not found
       return false;
```

```
   }

   public int size()  { return mSize; }

   public boolean setMaxLambda( double lam )
   {
      if (lam < .1 || lam > 100.)
         return false;
      mMaxLambda = lam;
      return true;
   }

   public void displayStatistics()
   {
       System.out.println("\nIn the HashSC class:\n");
       System.out.println( "Table Size = " +  mTableSize );;
       System.out.println( "Number of entries = " + mSize);
       System.out.println( "Load factor = " + (double)mSize/mTableSize);
       System.out.println( "Number of collisions = " + this.numCollisions );
       System.out.println( "Longest Linked List = " + this.longestCollisionRun );
   }

   // protected methods of class ----------------------
   protected void rehash()
   {
         // ADD CODE HERE TO RESET THE HashTable COUNTERS TO 0 for HW#5!!!!!!!!!!!!!!!


      // we save old list and size then we can reallocate freely
      LList<E>[] oldLists = mLists;
      int k, oldTableSize = mTableSize;
      Iterator<E> iter;

      mTableSize = nextPrime(2*oldTableSize);

      // allocate a larger, empty array
      allocateMListArray();  // uses mTableSize;

      // use the insert() algorithm to re-enter old data
      mSize = 0;
      for(k = 0; k < oldTableSize; k++)
         for(iter = oldLists[k].iterator(); iter.hasNext() ; )
            insert( iter.next());
   }

   protected int myHash( E x)
   {
      int hashVal;

      hashVal = x.hashCode() % mTableSize; // CHANGE TO USE Hasher's hash method for
HW#5!!!!!!!!!!
      if(hashVal < 0)
         hashVal += mTableSize;

      return hashVal;
   }

   protected static int nextPrime(int n)
   {
      int k, candidate, loopLim;
```

```
      // loop doesn't work for 2 or 3
      if (n <= 2 )
         return 2;
      else if (n == 3)
         return 3;

      for (candidate = (n%2 == 0)? n+1 : n ; true ; candidate += 2)
      {
         // all primes > 3 are of the form 6k +/- 1
         loopLim = (int)( (Math.sqrt((double)candidate) + 1)/6 );

         // we know it is odd.  check for divisibility by 3
         if (candidate%3 == 0)
            continue;

         // now we can check for divisibility of 6k +/- 1 up to sqrt
         for (k = 1; k <= loopLim; k++)
         {
            if (candidate % (6*k - 1) == 0)
               break;
            if (candidate % (6*k + 1) == 0)
               break;
         }
         if (k > loopLim)
            return candidate;
      }
   }

   private  void allocateMListArray()
   {
      int k;

      mLists = new LList[mTableSize];
      for (k = 0; k < mTableSize; k++)
         mLists[k] = new LList<E>();
   }

//-----------------------------------------------------------------------------------
// WRITE THE HashSCIterator<E> class (inner class in HashSC, so it WON'T be public)
//     This class MUST implement Iterator<E>, AND include the following:
//          private instance variables for LList<E>[] , an int (for current index),
//              AND an Iterator<E> (for the current linked list's iterator)
//          constructor with a LList<E>[] parameter in which you assign the parameter
//              the corresponding instance variable,
//              and the element[0]'s iterator to the Iterator instance variable
// Override methods:
//       boolean hasNext()
//           This method MUST determine if the array has an LList that hasn't been
//              completely iterated AT or AFTER the current index
//               (MUST advance the current index and assign Iterator
//              instance variable to the current index's Iterator if the current iterator
//              doesn't have any more (HINT for this will be given in class)
//       E next()
//          if hasNext() returns true, return the next() of the instance var. Iterator
//          else return false

}

//-----------------------------------------------------------------------------------
// Use the following data class:
```

```java
public class Color
{
    public static final int MAX_COLOR_CODE = 0xFFFFFF;

    private int code=0;
    private String name="";

    public Color(){}

    public Color(int c, String n)
    {
         setCode(c);
         setName(n);
    }

    public boolean setCode(int c)
    {
         if( c < 0 || c > MAX_COLOR_CODE )
            return false;
         code = c;
         return true;
    }

    public boolean setName( String s )
    {
         if( s==null || s.length() == 0 )
            return false;
         name = s;
         return true;
    }

    public int getCode(){ return code; }

    public String getName(){ return name; }

    public String toString()
    {
         String decFormat = String.format("%06X", code);
         return "Color: " + name + " = " + decFormat;
    }

} // end class Color
 //--------------------------------------------------------------------------------
// Include the following methods in the main class:
//
// Remember to import the correct package for Scanner and Files

    public static Scanner userScanner = new Scanner(System.in);

    // opens a text file for input, returns a Scanner:
    public static Scanner openInputFile()
    {
         String filename;
         Scanner scanner=null;

         System.out.print("Enter the input filename: ");
         filename = userScanner.nextLine();
         File file= new File(filename);
```

```
        try{
           scanner = new Scanner(file);
        }// end try
        catch(FileNotFoundException fe){
           System.out.println("Can't open input file\n");
               return null; // array of 0 elements
        } // end catch
        return scanner;
    }

    public static Color testContains(HashTable<Color> tableQP,
            HashTable<Color> tableSC,
            Color tempColor)
    {
        // YOU FINISH SO IT CREATES A COPY OF tempColor (new Color,
        //    NOT assigning the same reference, but another Color with
        //    the same code and name), then call contains for tableQP
        //    passing the copy Color and display the return value as
        //    shown in the test run.
        // Now call contains for tableSC (pass the same copy Color) and
        //    display the return value as shown in the test run.
        // Return the copy Color
    }

// Call the following in main for both HashTables (in one call):

    public static void testHashTables(HashTable<Color> tableQP,
            HashTable<Color> tableSC)
    {
        Color tempColorQP, tempColorSC;
        Color targetColor1=null, targetColor2=null;


        Iterator<Color> iter = tableQP.iterator(); // get Color from HashQP
        if( iter.hasNext()  ){
           tempColorQP = iter.next();
           targetColor1 = testContains(tableQP, tableSC, tempColorQP);// YOU WRITE
        }

        iter = tableSC.iterator(); // get Color from HashSC
        if( iter.hasNext()  ){
           tempColorSC = iter.next();
           targetColor2 = testContains(tableQP, tableSC, tempColorSC);// YOU WRITE
        }
        // found a Color in table1 in table2, now test getEntry and remove methods

        tempColorQP = tableQP.getEntry(targetColor1);
        if( tempColorQP != null )
        {
           System.out.println("Retrieved in HashQP, Color: " + tempColorQP.getName() + ",
now trying to delete it");
           // now delete it
           if( tableQP.remove(targetColor1))
               System.out.println("Successfully removed from HashQP: " +
targetColor1.getName());
           else
```

```
                System.out.println("Unsuccessful attempt to remove from HashQP: " +
targetColor1.getName());
        }else
            System.out.println("Error in HashQP: can't retrieve "+
targetColor1.getName());

        tempColorSC = tableSC.getEntry(targetColor2);
        if( tempColorSC != null )
        {
            System.out.println("Retrieved in HashSC, Color: " + tempColorSC.getName() + ",
now trying to delete it");
            // now delete it
            if( tableSC.remove(targetColor2))
                System.out.println("Successfully removed from HashSC: " +
targetColor2.getName());
            else
                System.out.println("Unsuccessful attempt to remove from HashSC: " +
targetColor2.getName());
        }
        else
            System.out.println("Error in HashSC: can't retrieve "+
targetColor2.getName());

    } // testHashTables
}
```