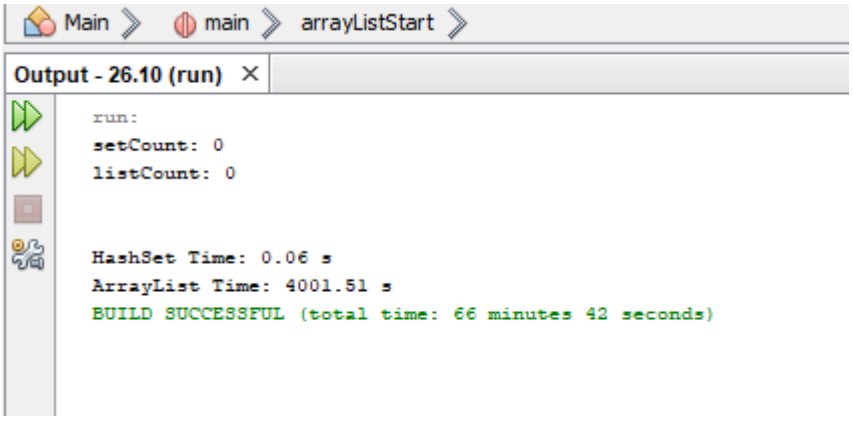
Sean Poston

CS300 Homework 8

27.10:



27.11:

```
234     @Override
        public boolean addAll(Collection<? extends E> arg0) {
236       boolean flag = false;
237       ArrayList<E> list = new ArrayList<>();
238
239       arg0.forEach(e -> list.add(e));
240
241       for (int i = 0; i < list.size(); i++) {
242           if (add(list.get(i))) {}
243           else
244               return false;
245       }
246
247       return true;
248     }
249
250     @Override
        public boolean containsAll(Collection<?> arg0) {
252       Iterator list = arg0.iterator();
253
254       while (list.hasNext()) {
            if (contains(list.next())) {}
256         else
257             return false;
258       }
259
260       return true;
261     }
262
263     @Override
        public boolean removeAll(Collection<?> arg0) {
265       Iterator list = arg0.iterator();
266
          for (int i = 0; i < table.length; i++) {
268         table[i].removeAll(arg0);
269       }
270
271       return true;
272     }
273
274     @Override
        public boolean retainAll(Collection<?> arg0) {
          for (int i = 0; i < table.length; i++) {
277         if (table[i].retainAll(arg0)) {}
278         else
279             return false;
280       }
281
282       return true;
283     }
```

```java
    @Override
    public Object[] toArray() {
      ArrayList<Object> temp = new ArrayList<>();
      for (int i = 0; i < table.length; i++) {
          for (int j = 0; j < table[i].size(); i++) {
              temp.add(table[i].get(j));
          }
      }
      Object[] list = new Object[temp.size()];

      for (int i = 0; i < temp.size(); i++) {
          list[i] = temp.get(i);
      }
      return list;
    }

    @Override
    public <T> T[] toArray(T[] arg0) {

      return null;
    }
}
```

Source Codes

27.10:

```
import java.util.ArrayList;
public class Main {
   public static void main(String[] args) {
      MyHashSet<Double> hashSet = new MyHashSet<>();
      MyArrayList<Double> arrayList = new MyArrayList<>();
      double randomVar;
      int listCount = 0;
      int setCount = 0;

      for (int i = 0; i < 1000000; i++) {
         randomVar = Math.random() * 1000000;
         hashSet.add(randomVar);
         arrayList.add(randomVar);
      }

      double[] checkList = new double[1000000];

      for (int i = 0; i < 1000000; i++) {
         checkList[i] = Math.random() * 2000000;
      }

      long hashSetStart = System.currentTimeMillis();
      for (int i = 0; i < 1000000; i++) {
         if (hashSet.contains(checkList[i]))
            setCount++;
      }
      long hashSetTime = System.currentTimeMillis() - hashSetStart;
```

```
        long arrayListStart = System.currentTimeMillis();
        for (int i = 0; i < 1000000; i++) {
            if (arrayList.contains(checkList[i]))
                listCount++;
        }
        long arrayListTime = System.currentTimeMillis() - arrayListStart;


        System.out.println("setCount: " + setCount + "\nlistCount: " + listCount);
        System.out.print("\n\nHashSet Time: "); System.out.printf("%.2f s\n", (double)hashSetTime / 1000.0);
        System.out.print("ArrayList Time: "); System.out.printf("%.2f s\n", (double)arrayListTime / 1000.0);
    }//end of main function
}//end of main class
```

```java
import java.util.ArrayList;
public class Main {
    public static void main(String[] args) {
        MyHashSet<Double> hashSet = new MyHashSet<>();
        MyArrayList<Double> arrayList = new MyArrayList<>();
        double randomVar;
        int listCount = 0;
        int setCount = 0;

        for (int i = 0; i < 1000000; i++) {
            randomVar = Math.random() * 1000000;
            hashSet.add(randomVar);
            arrayList.add(randomVar);
        }

        double[] checkList = new double[1000000];

        for (int i = 0; i < 1000000; i++) {
            checkList[i] = Math.random() * 2000000;
        }

        long hashSetStart = System.currentTimeMillis();
        for (int i = 0; i < 1000000; i++) {
            if (hashSet.contains(checkList[i]))
                setCount++;
        }
        long hashSetTime = System.currentTimeMillis() - hashSetStart;

        long arrayListStart = System.currentTimeMillis();
        for (int i = 0; i < 1000000; i++) {
            if (arrayList.contains(checkList[i]))
                listCount++;
        }
        long arrayListTime = System.currentTimeMillis() - arrayListStart;

        System.out.println("setCount: " + setCount + "\nlistCount: " + listCount);
        System.out.print("\n\nHashSet Time: "); System.out.printf("%.2f s\n", (double)hashSetTime / 1000.0);
        System.out.print("ArrayList Time: "); System.out.printf("%.2f s\n", (double)arrayListTime / 1000.0);
    }//end of main function
}//end of main class
```

27.11:

```
import java.util.*;

public class MyHashSet<E> implements Collection<E> {
  // Define the default hash table size. Must be a power of 2
  private final static int DEFAULT_INITIAL_CAPACITY = 4;

  // Define the maximum hash table size. 1 << 30 is same as 2^30
  private final static int MAXIMUM_CAPACITY = 1 << 30;

  // Current hash table capacity. Capacity is a power of 2
  private int capacity;

  // Define default load factor
  private final static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;

  // Specify a load factor threshold used in the hash table
  private float loadFactorThreshold;

  // The number of elements in the set
  private int size = 0;

  // Hash table is an array with each cell that is a linked list
  private LinkedList<E>[] table;

  /** Construct a set with the default capacity and load factor */
  public MyHashSet() {
    this(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
```

```
}

/** Construct a set with the specified initial capacity and
 * default load factor */
public MyHashSet(int initialCapacity) {
  this(initialCapacity, DEFAULT_MAX_LOAD_FACTOR);
}

/** Construct a set with the specified initial capacity
 * and load factor */
public MyHashSet(int initialCapacity, float loadFactorThreshold) {
  if (initialCapacity > MAXIMUM_CAPACITY)
    this.capacity = MAXIMUM_CAPACITY;
  else
    this.capacity = trimToPowerOf2(initialCapacity);

  this.loadFactorThreshold = loadFactorThreshold;
  table = new LinkedList[capacity];
}

public MyHashSet(E[] list) {
  this();
  for (int i = 0; i < list.length; i++) {
    add(list[i]);
  }
}

@Override /** Remove all elements from this set */
public void clear() {
```

```
  size = 0;
  removeElements();
}

@Override /** Return true if the element is in the set */
public boolean contains(Object e) {
  int bucketIndex = hash(e.hashCode());
  if (table[bucketIndex] != null) {
    LinkedList<E> bucket = table[bucketIndex];
    return bucket.contains(e);
  }

  return false;
}

@Override /** Add an element to the set */
public boolean add(E e) {
  if (contains(e)) // Duplicate element not stored
    return false;

  if (size + 1 > capacity * loadFactorThreshold) {
    if (capacity == MAXIMUM_CAPACITY)
      throw new RuntimeException("Exceeding maximum capacity");

    rehash();
  }

  int bucketIndex = hash(e.hashCode());
```

```
  // Create a linked list for the bucket if it is not created
  if (table[bucketIndex] == null) {
    table[bucketIndex] = new LinkedList<E>();
  }

  // Add e to hashTable[index]
  table[bucketIndex].add(e);

  size++; // Increase size

  return true;
}

@Override /** Remove the element from the set */
public boolean remove(Object e) {
  if (!contains(e))
    return false;

  int bucketIndex = hash(e.hashCode());

  // Create a linked list for the bucket if it is not created
  if (table[bucketIndex] != null) {
    LinkedList<E> bucket = table[bucketIndex];
    bucket.remove(e);
  }

  size--; // Decrease size

  return true;
```

```
}

@Override /** Return true if the set contains no elements */
public boolean isEmpty() {
  return size == 0;
}

@Override /** Return the number of elements in the set */
public int size() {
  return size;
}

@Override /** Return an iterator for the elements in this set */
public java.util.Iterator<E> iterator() {
  return new MyHashSetIterator(this);
}

/** Inner class for iterator */
private class MyHashSetIterator implements java.util.Iterator<E> {
  // Store the elements in a list
  private java.util.ArrayList<E> list;
  private int current = 0; // Point to the current element in list
  private MyHashSet<E> set;

  /** Create a list from the set */
  public MyHashSetIterator(MyHashSet<E> set) {
    this.set = set;
    list = setToList();
  }
```

```
  @Override /** Next element for traversing? */
  public boolean hasNext() {
    return current < list.size();
  }

  @Override /** Get current element and move cursor to the next */
  public E next() {
    return list.get(current++);
  }

  @Override /** Remove the element returned by the last next() */
  public void remove() {
    // Left as an exercise
    // You need to remove the element from the set
    // You also need to remove it from the list
  }
}

/** Hash function */
private int hash(int hashCode) {
  return hashCode & (capacity - 1);
}

/** Return a power of 2 for initialCapacity */
private int trimToPowerOf2(int initialCapacity) {
  int capacity = 1;
  while (capacity < initialCapacity) {
    capacity <<= 1;
```

```
  }

  return capacity;
}

/** Remove all e from each bucket */
private void removeElements() {
  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      table[i].clear();
    }
  }
}

/** Rehash the set */
private void rehash() {
  java.util.ArrayList<E> list = setToList(); // Copy to a list
  capacity <<= 1; // Double capacity
  table = new LinkedList[capacity]; // Create a new hash table
  size = 0; // Reset size

  for (E element: list) {
    add(element); // Add from the old table to the new table
  }
}

/** Copy elements in the hash set to an array list */
private java.util.ArrayList<E> setToList() {
  java.util.ArrayList<E> list = new java.util.ArrayList<>();
```

```
  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      for (E e: table[i]) {
        list.add(e);
      }
    }
  }

  return list;
}

@Override
public String toString() {
  java.util.ArrayList<E> list = setToList();
  StringBuilder builder = new StringBuilder("[");

  // Add the elements except the last one to the string builder
  for (int i = 0; i < list.size() - 1; i++) {
    builder.append(list.get(i) + ", ");
  }

  // Add the last element in the list to the string builder
  if (list.size() == 0)
    builder.append("]");
  else
    builder.append(list.get(list.size() - 1) + "]");

  return builder.toString();
```

```
}

@Override
public boolean addAll(Collection<? extends E> arg0) {
  boolean flag = false;
  ArrayList<E> list = new ArrayList<>();

  arg0.forEach(e -> list.add(e));

  for (int i = 0; i < list.size(); i++) {
     if (add(list.get(i))) {}
     else
        return false;
  }

  return true;
}

@Override
public boolean containsAll(Collection<?> arg0) {
  Iterator list = arg0.iterator();

  while (list.hasNext()) {
     if (contains(list.next())) {}
     else
        return false;
  }

  return true;
```

```
}

@Override
public boolean removeAll(Collection<?> arg0) {
  Iterator list = arg0.iterator();

  for (int i = 0; i < table.length; i++) {
     table[i].removeAll(arg0);
  }

  return true;
}

@Override
public boolean retainAll(Collection<?> arg0) {
  for (int i = 0; i < table.length; i++) {
     if (table[i].retainAll(arg0)) {}
     else
        return false;
  }

  return true;
}

@Override
public Object[] toArray() {
  ArrayList<Object> temp = new ArrayList<>();
  for (int i = 0; i < table.length; i++) {
     for (int j = 0; j < table[i].size(); i++) {
```

```
            temp.add(table[i].get(j));
        }
    }
    Object[] list = new Object[temp.size()];

    for (int i = 0; i < temp.size(); i++) {
        list[i] = temp.get(i);
    }
    return list;
  }

  @Override
  public <T> T[] toArray(T[] arg0) {

    return null;
  }
}
```