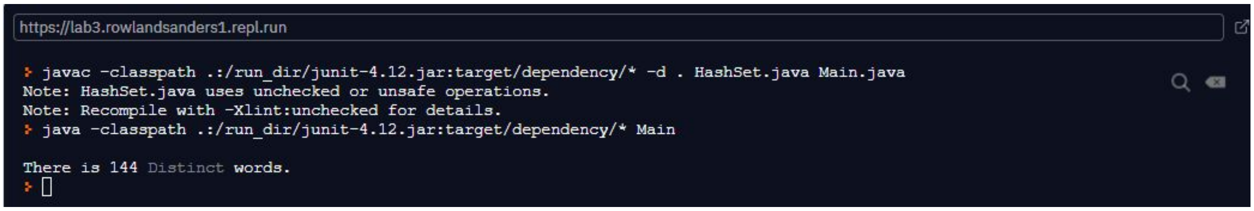
//OUTPUT



https://lab3.rowlandsanders1.repl.run

```
› javac -classpath .:/run_dir/junit-4.12.jar:target/dependency/* -d . HashSet.java Main.java
Note: HashSet.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
› java -classpath .:/run_dir/junit-4.12.jar:target/dependency/* Main

There is 144 Distinct words.
› ▯
```

//Main.java

```
import java.io.FileNotFoundException;
import java.io.File;
import java.util.Scanner;

public class Main {
   public static void main(String[] args) {
       try {
           HashSet<String> HS = new HashSet<String>(50);

           File myObj = new File("trump_speech.txt");
           Scanner read = new Scanner(myObj);
           while (read.hasNext()) {

               String word =
read.next().replaceAll("[^a-zA-Z0-9]", "");

               if (!word.equals("")){
                   HS.add(word);
               }
           }

           System.out.println("\nThere is "+ HS.count() + "
\033[2mDistinct\033[0m words. ");
           read.close();

       } catch (FileNotFoundException e) {
           System.out.println("txt file is missing");
       }
     }
}
```

//HashSet.java

```
import java.lang.reflect.Array;
import java.lang.Math;

public class HashSet<T> {
   private class Entry {
       public T mKey;
       public boolean mIsNil;

       public Entry(T key, boolean nilCheck)
       {
           mKey = key;
           mIsNil = nilCheck;
       }

   }
   private Entry[] mTable;
   private int mCount;


   public int probe(int i){
     int temp = (((i * i) + i) / 2);
     return temp;
   }

   public double loadFactor(){
     double temp = (mCount / mTable.length);
     return temp;
   }

   public int count(){
     return mCount;
   }

  public HashSet(int tableSize)
  {
     int powerChange = -1;
     do
     {
         tableSize = (tableSize / 2);
         powerChange++;
     }
     while (tableSize > 0);
     tableSize = 2 << powerChange;

     mTable = (Entry[])Array.newInstance(Entry.class,
tableSize);

     this.mCount = 0;
  }


   public void add(T key) {

       double fValue = 0.8;

       if (loadFactor() > fValue){
           Entry[] newTable;

           newTable = (Entry[])Array.newInstance(Entry.class,
mTable.length*2);
           for (int i = 0; i < mTable.length; i++){
               if (mTable[i].mIsNil == false && mTable[i] !=
null )
               {
                   Entry newEntry = new Entry(mTable[i].mKey,
mTable[i].mIsNil);

                   int currentIndex = 0;
                   int hashCode =
Math.abs(mTable[i].mKey.hashCode());


                   int currentPos = hashCode +
probe(currentIndex);

                   currentPos = (currentPos % newTable.length);
                   do {
                       if (newTable[currentPos] == null)
                       {
                           newTable[currentPos] = newEntry;
                           break;
                       }
                       currentIndex++;

                       currentPos = (hashCode +
probe(currentIndex)) % newTable.length;
                   } while (true);
               }
           }
           mTable = newTable;
       }

       Entry newEntry = new Entry(key, false);

       int hashCode = Math.abs(key.hashCode());
       int currentIndex = 0;
       int currentPos = hashCode + probe(currentIndex);

       currentPos = currentPos % mTable.length;
       while (!(mTable[currentPos] != null &&
mTable[currentPos].mKey.equals(key))) {
           if (mTable[currentPos] == null)
           {
               mTable[currentPos] = newEntry;
               mCount++;
               break;

           }else if (mTable[currentPos] != null &&
mTable[currentPos].mIsNil == true)
           {
               mTable[currentPos] = newEntry;
               mCount++;
               break;
           }

           currentIndex++;
           currentPos = (hashCode + probe(currentIndex)) %
mTable.length;
       }
   }
```

```
   public boolean find(T key) {

       int hashCode = Math.abs(key.hashCode());
       int currentIndex = 0;
       int currentPos = hashCode + probe(currentIndex) %
mTable.length;

       while (mTable[currentPos] != null){
           if (mTable[currentPos] != null &&
mTable[currentPos].mIsNil == true)
             continue;

           if (mTable[currentPos] != null &&
mTable[currentPos].mKey.equals(key))
             return true;

           if (currentIndex >= mTable.length)
             return false;

           currentIndex++;
           currentPos = (hashCode + probe(currentIndex)) %
mTable.length;
       }
       return false;
   }


   public void remove(T key) {

       int hashCode = Math.abs(key.hashCode());
       int currentIndex = 0;
```

```
        int currentPos = (hashCode + probe(currentIndex)) %
mTable.length;

        while (mTable[currentPos] != null) {
            if (mTable[currentPos].mIsNil == true){
                continue;
            }
            if (mTable[currentPos].mKey.equals(key)){
                mTable[currentPos].mKey = null;
                mTable[currentPos].mIsNil = true;
                mCount = mCount - 1;
            }
            if (currentIndex >= mTable.length){
                break;
            }
            currentIndex++;
            currentPos = (hashCode + probe(currentIndex)) %
mTable.length;
        }
    }
}
```