***Project 2 Memo***

***CIS-3501***

***Mohammad Mukahhal***

***Professor Guo***

***University of Michigan Dearborn***

|  |
| --- |
| public QuadraticProbingHashTable( ) {  this( DEFAULT\_TABLE\_SIZE ); } |

*Time Complexity: O(1)*

* This method has a time complexity of O(1) that is because the method simply contains simple operations.

|  |
| --- |
| /\*\* \* Construct the hash table. \* @param size the approximate initial size. \*/ public QuadraticProbingHashTable( int size ) {  allocateArray( size );  makeEmpty( ); } |

*Time Complexity: O(1)*

* This method has a time complexity of O(1) that is because the method simply contains simple operations.

|  |
| --- |
| /\*\* \* Insert into the hash table. If the item is \* already present, do nothing. \* @param x the item to insert. \*/ public boolean insert( AnyType x ) {  // Insert x as active  int currentPos = findPos( x );  if( isActive( currentPos ) )  return false;   array[ currentPos ] = new HashEntry<AnyType>( x, true );   // Rehash; see Section 5.5  if( ++currentSize > array.length / 2 )  rehash( );  return true; } |

*Time Complexity: O(1)*

* This method has a time complexity of O(1) that is because the method simply contains simple operations.

|  |
| --- |
| /\*\* \* Expand the hash table. \*/ private void rehash( ) {  HashEntry<AnyType> [ ] oldArray = array;   // Create a new double-sized, empty table  allocateArray( nextPrime( 2 \* oldArray.length ) );  currentSize = 0;   // Copy table over  for( int i = 0; i < oldArray.length; i++ )  if( oldArray[ i ] != null && oldArray[ i ].isActive )  insert( oldArray[ i ].element ); } |

*Time Complexity: O(n)*

* This method has a time complexity of O(n) that is because the method undergoes its own loop which has a worst case of O(n).

|  |
| --- |
| private int findPos( AnyType x ) {  int offset = 1;  boolean newWord = true;  chainLength = 1;  int currentPos = myhash( x );   while( array[ currentPos ] != null &&  !array[ currentPos ].element.equals( x ) )  {  currentPos += offset; // Compute ith probe  offset += 2;  if( currentPos >= array.length )  currentPos -= array.length;  if(newWord){  numCollisions++;  newWord = false;  }  chainLength++;  }  totalChainLength += chainLength;  if(chainLength > longestChain){  longestChain = chainLength;  }   return currentPos; } |

*Time Complexity: O(n)*

* This method has a time complexity of O(n) that is because the method undergoes its own loop which has a worst case of O(n).

|  |
| --- |
| /\*\* \* Remove from the hash table. \* @param x the item to remove. \*/ public void remove( AnyType x ) {  int currentPos = findPos( x );  if( isActive( currentPos ) )  array[ currentPos ].isActive = false; } |

*Time Complexity: O(1)*

* This method has a time complexity of O(1) that is because the method simply contains simple operations.

|  |
| --- |
| /\*\* \* Find an item in the hash table. \* @param x the item to search for. \* @return the matching item. \*/ public boolean contains( AnyType x ) {  int currentPos = findPos( x );  return isActive( currentPos ); } |

*Time Complexity: O(1)*

* This method has a time complexity of O(1) that is because the method simply contains simple operations.

|  |
| --- |
| /\*\* \* Return true if currentPos exists and is active. \* @param currentPos the result of a call to findPos. \* @return true if currentPos is active. \*/ private boolean isActive( int currentPos ) {  return array[ currentPos ] != null && array[ currentPos ].isActive; } |

*Time Complexity: O(1)*

* This method has a time complexity of O(1) that is because the method simply contains simple operations.

|  |
| --- |
| /\*\* \* Make the hash table logically empty. \*/ public void makeEmpty( ) {  currentSize = 0;  for( int i = 0; i < array.length; i++ )  array[ i ] = null; } |

*Time Complexity: O(n)*

* This method has a time complexity of O(n) that is because the method undergoes its own loop which has a worst case of O(n).

|  |
| --- |
| private int myhash( AnyType x ) {  int hashVal = x.hashCode( );   hashVal %= array.length;  if( hashVal < 0 )  hashVal += array.length;   return hashVal; } |

*Time Complexity: O(1)*

* This method has a time complexity of O(1) that is because the method simply contains simple operations.

|  |
| --- |
| public HashEntry( AnyType e )  {  this( e, true );  } |

*Time Complexity: O(1)*

* This method has a time complexity of O(1) that is because the method simply contains simple operations.

|  |
| --- |
| public HashEntry( AnyType e, boolean i )  {  element = e;  isActive = i;  } |

*Time Complexity: O(1)*

* This method has a time complexity of O(1) that is because the method simply contains simple operations.

|  |
| --- |
| /\*\* \* Internal method to allocate array. \* @param arraySize the size of the array. \*/ @SuppressWarnings("unchecked") private void allocateArray( int arraySize ) {  array = new HashEntry[ nextPrime( arraySize ) ]; } |

*Time Complexity: O(1)*

* This method has a time complexity of O(1) that is because the method simply contains simple operations.

|  |
| --- |
| /\*\* \* Internal method to find a prime number at least as large as n. \* @param n the starting number (must be positive). \* @return a prime number larger than or equal to n. \*/ private static int nextPrime( int n ) {  if( n <= 0 )  n = 3;   if( n % 2 == 0 )  n++;   for( ; !isPrime( n ); n += 2 )  ;   return n; } |

*Time Complexity: O(n)*

* This method has a time complexity of O(n) that is because the method undergoes its own loop which has a worst case of O(n).

|  |
| --- |
| /\*\* \* Internal method to test if a number is prime. \* Not an efficient algorithm. \* @param n the number to test. \* @return the result of the test. \*/ private static boolean isPrime( int n ) {  if( n == 2 || n == 3 )  return true;   if( n == 1 || n % 2 == 0 )  return false;   for( int i = 3; i \* i <= n; i += 2 )  if( n % i == 0 )  return false;   return true; } |

*Time Complexity: O(n)*

* This method has a time complexity of O(n) that is because the method undergoes its own loop which has a worst case of O(n).

|  |
| --- |
| // Simple main public static void main( String [ ] args ) throws IOException {  QuadraticProbingHashTable<String> H = new QuadraticProbingHashTable<String>( );  final int NUMS = 400000;  final int GAP = 37;    ArrayList<String> testWords = new ArrayList<String>(); //Test file ArrayList  ArrayList<String> incorrectWords = new ArrayList<String>();  ArrayList<Integer> lineNumberList = new ArrayList<Integer>();  ArrayList<Integer> updatedLineNumberList = new ArrayList<Integer>();  System.out.println( "Checking... (no more output means success)" );   //Reading from a file  BufferedReader words = new BufferedReader(new FileReader(args[1]));  //Scanner scan = new Scanner(words);   //insert file words into hashtable  String currentLine;  while((currentLine = words.readLine()) != null){  //System.out.println(scan.next());  H.insert(currentLine);  }   //Reading the testFile  // File test = new File("test1.txt");  LineNumberReader line = new LineNumberReader(new FileReader(args[2]));  //Placing test file words into an ArrayList  String l;  while ((l = line.readLine()) != null) {  Scanner scanTest = new Scanner(l);  while (scanTest.hasNext()) {  testWords.add(scanTest.next());  lineNumberList.add(line.getLineNumber());    }  }    for(int i = 0; i < testWords.size(); i++){  //Filtering words from punctuation  testWords.set(i, testWords.get(i).replaceAll("[^a-zA-Z]", ""));   //Check for misspelled words  if(!H.contains(testWords.get(i))){  //checks if word is not contained with uppercase  if(!H.contains(testWords.get(i).toLowerCase())){  //checks if code is not contained with lowercase  incorrectWords.add(testWords.get(i));  updatedLineNumberList.add(lineNumberList.get(i));  }  }  }   //Print stats BufferedWriter out = **new** BufferedWriter(**new** FileWriter("outfile.txt"));  out.write("Number of words: " + H.currentSize + " ");  out.write("Table size: " + H.array.length + " ");  out.write("Load factor: " + (H.currentSize\*1.0)/(H.array.length\*1.0) + " \n");  out.write("Number of collisions: " + H.numCollisions + " ");  out.write("Average Chain Length: " + H.totalChainLength/H.currentSize + " ");  out.write("Longest Chain Length: " + H.longestChain + " \n");  *generateCorrections*(H,incorrectWords,updatedLineNumberList,out);  out.close();  //Given Code  for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )  H.insert( ""+i );  for( int i = 1; i < NUMS; i+= 2 )  H.remove( ""+i );   for( int i = 2; i < NUMS; i+=2 )  if( !H.contains( ""+i ) )  System.out.println( "Find fails " + i );   for( int i = 1; i < NUMS; i+=2 )  {  if( H.contains( ""+i ) )  System.out.println( "OOPS!!! " + i );  }  } |

*Time Complexity: O(n^3)*

* This method has a time complexity of O(n^3) that is because the method calls the generateCorrections method which has a time complexity of n^3. If we were to ignore the generateCorrections method then main would have a time complexity of n^2.

|  |
| --- |
| private static void generateCorrections(QuadraticProbingHashTable<String> H,ArrayList<String> words,  ArrayList<Integer> lineNumber, BufferedWriter out){  for(int i = 0; i < words.size(); i++){  ArrayList<String> possibleWords = new ArrayList<String>();  for (int j = 0; j < words.get(i).length(); j++){  String temp1 = words.get(i);  //Construct every string that can be made by deleting one letter from the word.  temp1 = temp1.substring(0, j) + temp1.substring(j+1);  if(H.contains(temp1)){  possibleWords.add(temp1);  }  //Construct every string that can be made by inserting any letter of  //the alphabet at any position in the string. (26\*(n+1) possibilities)  char[] alphabet = "abcdefghijklmnopqrstuvwxyz".toCharArray();  for(int x = 0; x < 26; x++){  //temp.set(0,testWords.get(i));  //temp.add(j,Character.toString(alphabet[x]));  temp1 = words.get(i);  temp1 = temp1.substring(0,j) + alphabet[x] + temp1.substring(j);  if(H.contains(temp1)){  possibleWords.add(temp1);  }  //Construct every string that can be made by replacing each letter in the  //word with some letter of the alphabet. (26\*n possibilities  //including the original word n times, which is probably easier than avoiding it)  StringBuilder temp2 = new StringBuilder(words.get(i));  temp2.setCharAt(j,alphabet[x]);  if(H.contains(temp2.toString())){  possibleWords.add(temp2.toString());  }  }  //Construct every string that can be made by swapping two neighboring  //characters in the string. (n-1 possibilities)  temp1 = words.get(i);  if(j < words.get(i).length() - 1){  temp1 = swap(temp1,j, j+1);  if(H.contains(temp1)){  possibleWords.add(temp1);  }  }  }  print(words.get(i),possibleWords,lineNumber.get(i));  }  } |

*Time Complexity: O(n^3)*

* This method has a time complexity of O(n^3) that is because the method in the worst case undergoes 3 for loops making it have a time complexity of n^3.

|  |
| --- |
| private static String swap(String str, int i, int j) {  if (j == str.length() - 1)  return str.substring(0, i) + str.charAt(j)  + str.substring(i + 1, j) + str.charAt(i);   return str.substring(0, i) + str.charAt(j)  + str.substring(i + 1, j) + str.charAt(i)  + str.substring(j + 1, str.length()); } |

*Time Complexity: O(1)*

* This method has a time complexity of O(1) that is because the method simply contains simple operations.

|  |
| --- |
| private static void print(String incorrectWord, ArrayList<String> possibleWords, int lineNumber, BufferedWriter out){ out.write(incorrectWord + "(" + lineNumber + "): ");  **for** (**int** i = 0; i < possibleWords.size(); i++){  out.write(possibleWords.get(i)+ " ");  }  out.write("\n");  } |

*Time Complexity: O(n)*

* This method has a time complexity of O(n) that is because the method undergoes its own loop which has a worst case of O(n).