Lab Assignment #5-6 Deadline Code: 29th, July, Fri 11:59pm

Submitted by: Oscar I. Ricaud Report & Demo: ?

ID: 80466791

Email Address: [oiricaud@miners.utep.edu](mailto:xyz@miners.utep.edu)

Class Time: Mon- Fri 10:55 am – 12:00 pm



**Introduction:**

In this lab the main task is to implement *Graphs* which includes the following: Prim’s Algorithm, Dijkstra’s, Euler and much more. To accomplish this, I will be using a nxn adjacency matrix to keep track the edges of each node.

../../../../Desktop/Graph.pdf

**Proposed Solution:**

**Define classes**

1. *Main method Adjacency Matrix:*
2. int NumberOfvertices = 5+1;
3. Object[][] emptyMatrix = adjacencyMatrices.buildMatrix(NumberOfvertices);
4. /\* Print Empty Matrix \*/
5. printMatrix(NumberOfvertices, emptyMatrix);
7. Print(“Label your vertices in the graph, we'll start with the first row");
8. /\* Assign labels to the first row of the matrix \*/
9. for(int row = 1 ; row < NumberOfvertices; row++){
10. emptyMatrix[0][row] = input.next();
11. end for
12. /\* Print Non-Empty Matrix, has references of the vertices \*/
13. Print ("Building our Adjacency Matrix: " );
14. printMatrix(NumberOfvertices, emptyMatrix);
15. Print (" ");
16. Print ("Now do the same for the first column");
17. /\* Assign labels to the first column of the matrix \*/
18. for(int column = 1 ; column < NumberOfvertices; column++){
19. emptyMatrix[column][0] = input.next();
20. end for
21. /\* Print Non-Empty Matrix, has references of the vertices \*/
22. System.out.println("Building our Adjacency Matrix: " );
23. printMatrix(NumberOfvertices, emptyMatrix);
24. Print (" " );
25. /\* Create a matrix to obtain the actual number of edges each vertex connects to \*/
26. Object[][] adjacencyMatrix = connectVertices(NumberOfvertices, emptyMatrix);
27. printMatrix(NumberOfvertices, adjacencyMatrix);
28. /// Array [0] = color
29. /// Array [1] = Predessesor
30. /// Array [3] = distance
31. linked list listofEdges = printEdges(NumberOfvertices, adjacencyMatrix);
32. Print ("Edges = " + listofEdges );
34. LinkedList listVertices = getVertices(NumberOfvertices, adjacencyMatrix);
35. Print ("vertices: " + listVertices);
37. /\* • Breadth Depth Search \*/
38. BreadthDepthSearch.calculator(adjacencyMatrix, listVertices, listofEdges, NumberOfvertices);

41. /\* • Depth First Search \*/
42. DepthSearch.calculator(adjacencyMatrix, listVertices, listofEdges, NumberOfvertices);
43. /\* • Topological Sort \*/
44. //ToplogicalSort.calculator(adjacencyMatrix, listVertices, listofEdges, NumberOfvertices);
45. /\* • Kruskal’s Algorithm \*/
46. Krukasls.calculator(adjacencyMatrix, listVertices, listofEdges, NumberOfvertices);
47. /\* • Dijkstra’s Algorithm \*/
48. /\* • Prim’s Algorithm \*/
49. /\* • Euler Path or Circuit \*/
50. /\* It determines if the graph has an Euler path \*/
    1. eulerPath.calculator(NumberOfvertices, adjacencyMatrix, listofEdges);
51. /\* Obtain the vertices \*/
52. linkedlist getVertices(int vertices, Object[][] adjacencyMatrix) {
53. Create a LinkedList called listVertices
54. for(column = 1; column < adjacencyMatrix.length; column++)
55. vertex = adjacencyMatrix[0][column];
56. listVertices.add(vertex);
57. end for
58. return listVertices;
59. end method
60. linkedList printEdges(int n, Object[][] adjacencyMatrix) {
61. int i = 1;
62. Create a LinkedList call it listofEdges
63. /\* Adds the edges to the list \*/
64. for(int row = 0; row < n; row++)
65. for(int column = 0 ; column < n; column++)
66. if(adjacencyMatrix[row][column].equals(1)){
67. String edges = "{" + adjacencyMatrix[0][row] + ", " + adjacencyMatrix[column][0] + "}";
68. listofEdges.add(edges.toLowerCase());
69. end if
70. end for
71. end for
72. return listofEdges;
73. end method
74. private static Object[][] connectVertices(int n, Object[][] matrix) {
75. /\* Does comparison if edges contain 1 replace at the Adjacency Matrix \*/
76. for(int row = 1; row < n; row++)
77. for(int column = 1 ; column < n; column++)
78. if(edges[row-1][column-1] >= 1)
79. matrix[row][column] = edges[row-1][column-1];
80. end if
81. end for
82. end for
83. return matrix;
84. end method
85. /\* Print method to print any matrix \*/
86. printMatrix(int n, Object[][] emptyMatrix) {
87. for(int row = 0 ; row < n ; row ++){
88. for(int column = 0; column < n; column++){
89. Print (emptyMatrix[row][column] + " ");
90. end for
91. Print ("");
92. end for
93. Print ("");
94. End method
95. *Graph Class* This graph class provides the color of the vertex, the predecessors, and the current weight of the vertex, it also assigns the color of the vertices and adds weights etc.
96. Object [][] vertex;
97. LinkedList<Object> source = new LinkedList<Object>();
98. String predecessor = null;
99. String color = "white";
100. int distance = 0;
101. /\* Column positions \*/
102. int vertexPos = 0;
103. int colorPos = 1;
104. int prePos = 2;
105. int distPos = 3;
106. public graphClass(int rowSize)

vertex = new Object[rowSize][4];

1. end method
2. public void addVertex(Object node, int row) {

vertex[row][vertexPos] = node;

// This is our source paint it black

if(row == 0){

vertex[row][colorPos] = "black";

}

else{

vertex[row][colorPos] = color;

}

1. end method
2. public Object getColor(int count ) {

for(int i = 0; i < vertex.length; i++){

if(i == count){

return vertex[i][colorPos];

end if

end for

return null;

1. end method
2. public Object setColor(int count, String tempColor) {

for(int i = 0; i < vertex.length; i++){

if(i == count){

vertex[i][colorPos] = tempColor;

end if

end for

return null;

1. end method
2. public void info(){

for(int row = 0; row < vertex.length; row++){

System.*out*.print("Vertex " + row);

for(int column = 0; column < 4; column++){

System.*out*.print(" | " + vertex[row][column] + " | " );

End for

End for

System.*out*.println(" ");

1. end method
2. public int getSmallestEdge(){
   1. int smallest = (int) vertex[1][3];
3. System.*out*.println("smallest" + smallest);
4. for(int row = 1; row < vertex.length-1; row++){

System.*out*.print("Vertex " + row);

if((Integer) vertex[row][3] < smallest){

smallest = (Integer) vertex[row][3];

* 1. end if
  2. System.*out*.println(" ");

1. End for
2. return smallest;
3. end method
4. public Object getVertex(int count) {
5. for(int i = 0; i < vertex.length; i++){

if(i == count){

return vertex[i][vertexPos];

end if

1. end for
2. end method
3. public void setPredecessor(int count, Object text) {
4. for(int i = 0; i < vertex.length; i++){

if(i == count){

vertex[i][prePos] = text;

end if

1. end for
2. end method
3. public Object getPredecessor(int count) {
4. for(int i = 0; i < vertex.length; i++){
   * 1. if(i == count){
        1. return vertex[i][prePos];
     2. end if
5. end for
6. return null;
7. end method
8. public void setDistance(int count, int distance) {
9. for(int i = 0; i < vertex.length; i++){
   * 1. if(i == count){
     2. vertex[i][distPos] = distance;
     3. end if
10. end for
11. end method
12. public Object getDistance(int count) {
13. for(int i = 0; i < vertex.length; i++){
    * 1. if(i == count){
      2. return vertex[i][vertexPos];
      3. end if
14. end for
15. return null;
16. end method
17. *Breadth First Search:* This algorithm traverses through the adjacency matrix. Once it reads a vertex with a label >= 1, it instantly becomes our initial vertex and the algorithm begins. This initial vertex, S looks at the current row of the adjacency matrix and looks for a label >= 1. We check if they are painted white if they are we reach them out and copy them to our table.

* 1. BDS(adjacancyMatrix, listofVertices, listOfEdges)
  2. Graph BDSgraph = new Graph() // Create new graph
  3. For( i TO listofVertices)
     1. BDSgraph add ( listofVertices at i)
  4. End for
  5. While(all nodes have been marked black)
  6. int rowCount = 0;
  7. int distance = 0;

String bdsPath = “”

* 1. while( int i = 0 < the number of vertices)
  2. if(BDSgraph get color at i ) == black)

BDSpath = BDSpath + (*adjacentVertices*(adjacencyMatrix, BDSgraph.getVertex(rowCount), BDSgraph, i, distance));

End if

* 1. distance++;
  2. i++
  3. end while

*Adjacent Vertices:* This method returns the path for using BDS.

1. adjacentVertices(adjacencyMatrix, vertex, graph, int rowCount, int distance) {
2. int visitEdge = 1;
3. Create new linke list of string = BDSpath
4. BDSpath.add((String) current vertex + Predecessor of current vertex);
5. while(visitEdge < adjacencyMatrix.length){
6. if((Integer) adjacencyMatrix[rowCount+1][visitEdge] >= 1 && graph.getColor(visitEdge-1).equals("white")){

distance = (Integer) adjacencyMatrix[rowCount+1][visitEdge];

graph.setColor(visitEdge-1, "black");

graph.setPredecessor(visitEdge-1, vertex);

graph.setDistance(visitEdge-1, distance);

1. end if
2. end while
3. visitEdge++;
4. return BDSpath;
5. *Depth First Search:* This algorithm is like a missile, it first traverses through the adjacency matrix and once it finds a vertex with a value >= 1 it, this becomes our starting vertex. It then looks at the corresponding row of this specific vertex and looks for the first adjacent vertex it sees and assigns this adjacent vertex to now be the current vertex and it keeps going until all vertices have been colored “black”. This is very similar to depth first search except that instead of checking all adjacent vertices it checks one and it goes with that one and backtracks if necessary.
6. DFS(adjacencyMatrix, listVertices, listofEdges, int numberOfvertices) {
7. Create a new graph from the graphClass and call it DPSgraph
8. for(i = 0 TO listVertices.size)

DPSgraph.addVertex(listVertices.get(i), i);

1. end for
2. int rowCount = 0;
3. int distance =0;
4. *DPSpath*.add(current vertex at rowCount)
5. while(rowCount < listVertices.size()){

if(color from the graph at (rowCount) == black

*adjacentVertices*(adjacencyMatrix, DPSgraph.getVertex(rowCount), DPSgraph, rowCount, distance);

rowCount = listVertices.size +1;

end if

rowCount++;

1. end while
2. *Adjacent Vertices:* This method returns the path for using DPS.
3. adjacentVertices(adjacencyMatrix, vertex, graph, int rowCount, int distance) {
4. int visitEdge = 1;
5. int column = 1;
6. while(visitEdge < adjacencyMatrix.length){
7. if((Integer) adjacencyMatrix[rowCount+1][visitEdge] >= 1 && graph.getColor(visitEdge-1).equals("white")){

distance = (Integer) adjacencyMatrix[rowCount+1][visitEdge];

graph.setColor(visitEdge-1, "black");

graph.setPredecessor(visitEdge-1, vertex);

graph.setDistance(visitEdge-1, distance);

*DPSpath*.add(graph.getVertex(visitEdge-1));

if(column != adjacencyMatrix.length-1){

*adjacentVertices*(adjacencyMatrix, graph.getVertex(visitEdge-1), graph, column, distance);

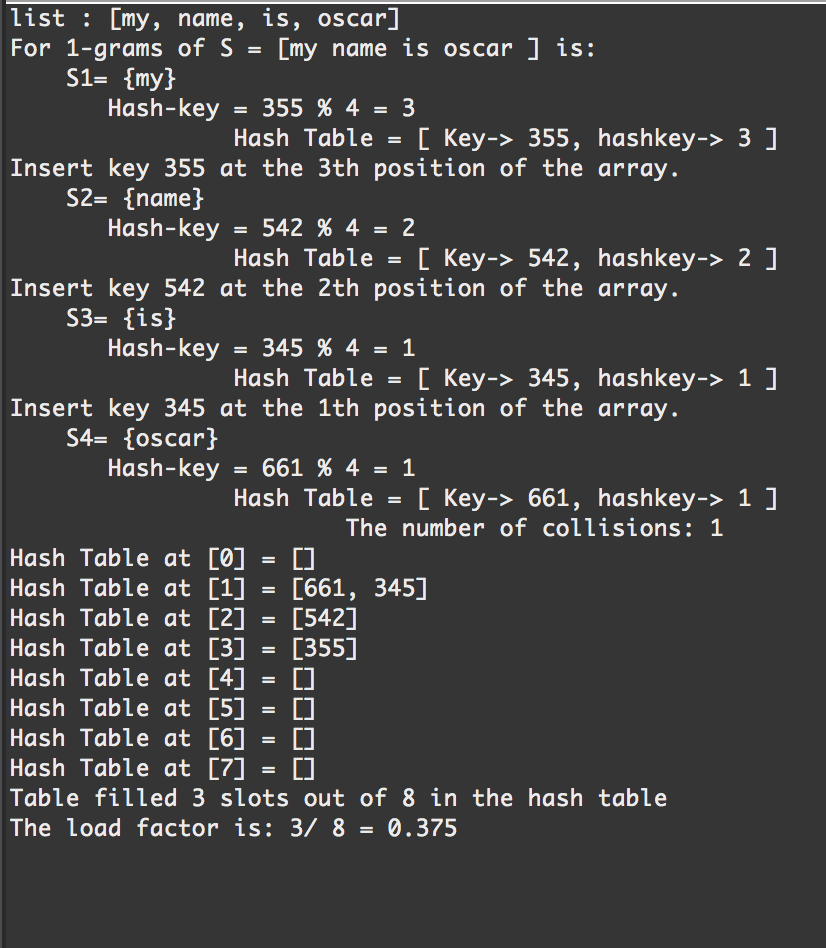
end if

else

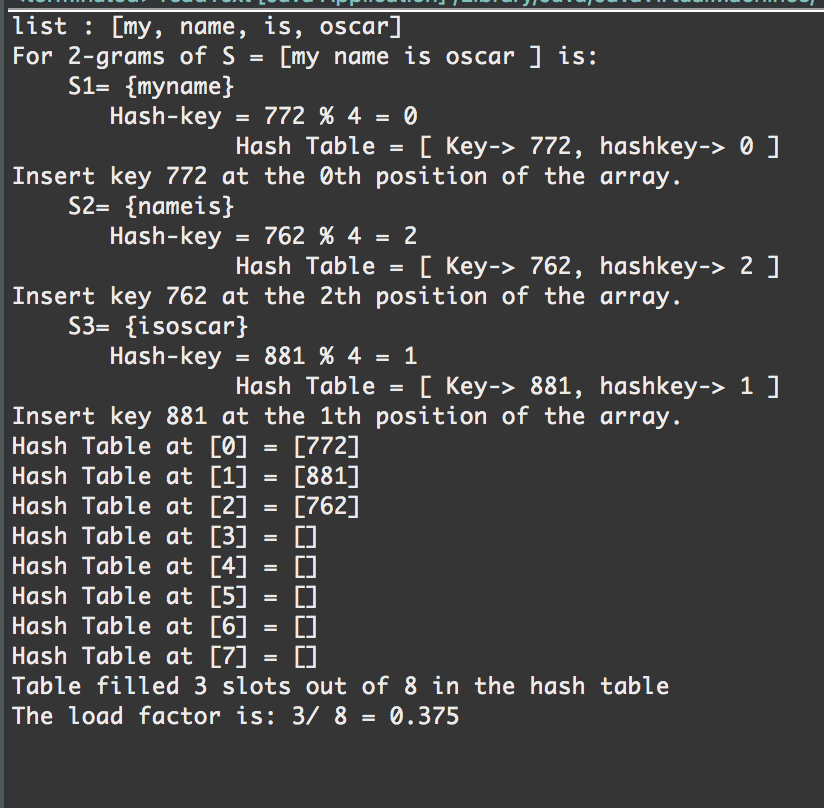
column=0;

1. end if
2. column++
3. visitEdge++
4. end while
5. return DPS path
6. *HashTable(int key, int hashKey) The magic happens*
   1. Print Key and Hash Key
   2. If myTable[hashKey] is empty
      1. Print Insert key at the haskKey position
      2. myTable[hashKey].add(key)
   3. else
      1. numCollisions++;
      2. int counter = 0;
      3. myTable[hashKey].add(counter, key)
      4. counter++
      5. Print the number of collisions.
   4. end method
7. *Print HashTable() Print table*
   1. Int filled = 0
   2. For( i = 0 TO column.myTable.length; i++)
      1. If(column.myTable[i] is not empty)
         1. Filled++;
      2. End if
      3. Print Hash table at [i]
      4. Print “ Table filled” + filled + “ slots out of “ + column.myTable.length + “ in the hash table”;
      5. Print load factor
   3. end method

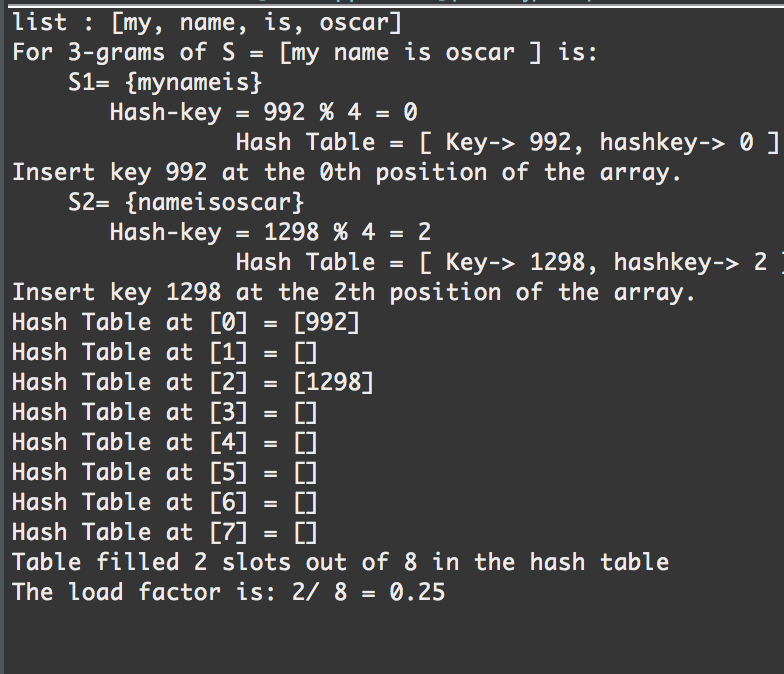
Input1 : “my name is oscar” n =1



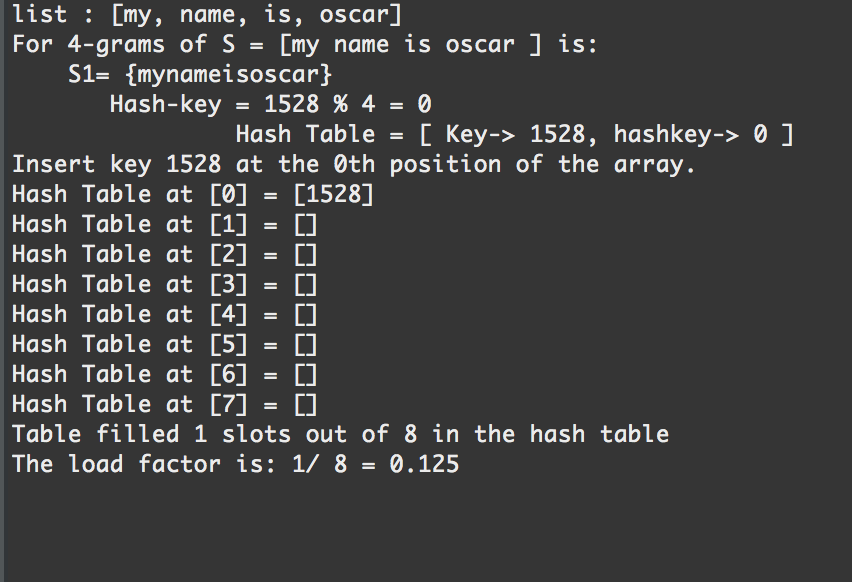
Input2: “my name is oscar” n = 2



Input3 : “my name is oscar” n = 3



Input4 : “my name is oscar” n = 4



Expirement 2:

List: id like to share a revelation that ive had during my time here It came to me when I tried to classify your species and I realized that youre not actually mammal every mammal on this planet instinctively develops a natural equilibrium with the surrounding environment but you humans do not you move to an area and you multiply and multiply until every natural resource is consumed and the only way you can survive is to spread to another area there is another organism on this planet that follows the same pattern. do you know what it is question mark a virus human beings are a disease a cancer of this planet youre a plague and we are the cure

Appendix: ­

package lab4;

import java.io.BufferedReader;

import java.io.File;

import java.io.FileReader;

import java.io.IOException;

import java.util.LinkedList;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* //

//Author: Oscar Ivan Ricaud.

/\*Assignment: Lab 4

Instructor: Professor Julio Urenda

TA: Saiful Abu

Course 2302

Data of Last modification July 17th, 2016

\*/

//Program purpose:

/\* The purpose of this program is to implement a hash table to get a better understanding how this algorithm is fast.

\* It first reads a text file and converts each word into a node and then using the ASCII it converts the word

\* into an integer. I then created a hash table size on powers of 2 based on the size of the text file. So if the text

\* file is size 100 it will round up to the nearest power, 2^7 = 128 table size.

\*/

//How to operate lab4 readText.java:

/\* Click play button and read console for output

\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* //

/\*

\* for n = 1 and file= "my name is oscar", n-grams would be "my", "name", "is", "oscar",

\* for n = 2 n-grams would be "my name", "name is", "is oscar".

\* for n = 3, n-grams would be "my name is", "name is oscar".

\* for n = 4 n-grams would be "my name is oscar".

\*/

public class readText {

static int counter = 0;

static String emptySpace = "";

static int n = 5;

public static void main(String[] args) throws IOException {

/\* Begin to read a text file, change path if necessery \*/

File file = new File("/Users/oscarricaud/Desktop/name.txt");

BufferedReader br = new BufferedReader(new FileReader(file));

StringBuffer fileContents = new StringBuffer();

String line = br.readLine();

while (line != null) {

fileContents.append(line);

line = br.readLine();

}

br.close();

LinkedList <String> list = new LinkedList<String>();

list = convertoTxtLinkedList(fileContents.toString());

System.out.println("list : " + list);

/\* End to read a text file \*/

/\* Begin to convert the words into nodes \*/

iNode x = null;

x = convertToNode(x, list);

System.out.print("For " + n + "-grams of S = [" );

iNode.printList(x);

System.out.println("] is:");

/\* End to convert the words into nodes\*/

/\* Create the hash table respect to the table length by rounding up the nearest powers \*/

int log = (int) (Math.log(list.size()) / Math.log(2));

int tableSize = (int) Math.pow(2, log+1 );

Hashtable column = new Hashtable(tableSize);

int min = 0;

int max = n;

int ListSize = list.size();

iNode x2 = null;

/\* Build the n-Gram sublists \*/

column.buildSublists(x, min, max , ListSize, x2);

/\* Print hash table \*/

int filled = 0;

for(int i = 0 ; i < column.myTable.length; i++){

if(!(column.myTable[i].isEmpty())){

filled++;

}

System.out.println("Hash Table at [" + i + "] = " + column.myTable[i].toString());

}

System.out.println("Table filled " + filled + " slots out of " + (column.myTable.length) + " in the hash table");

double loadFactor = (double)(filled)/(column.myTable.length);

System.out.println("The load factor is: " + filled + "/ " + (column.myTable.length) + " = " + loadFactor);

}

// Converts the list into nodes //

private static iNode convertToNode(iNode x, LinkedList<String> list) {

for(int i = list.size()-1; i >= 0; i--){

x = new iNode(list.get(i), x);

}

return x;

}

/\* Converts the text file into a signly linked list \*/

private static LinkedList<String> convertoTxtLinkedList(String string2) {

LinkedList <String> list = new LinkedList<String>();

int end = 0;

int begin = 0;

for(int i = 0 ; i < string2.length(); i++){

char temp = string2.charAt(i);

if(temp !=(' ')){

end++;

}

else{

String temp2 = string2.substring(begin, end);

list.add(temp2);

begin = end+1;

end = begin;

}

}

return list;

}

}

**package** lab4;

**import** java.util.LinkedList;

**public** **class** **Hashtable** {

**int** count = 1;

**int** numCollisions = 0;

**public** **LinkedList** [] myTable;

**public** **Hashtable**(**int** size){

myTable = **new** **LinkedList** [size];

**for**(**int** **i** = 0; i < myTable.length; i++){

myTable[i] = **new** LinkedList();

}

}

**public** **String** **buildSublists**(**iNode** sentence, **int** turtle, **int** rabbit, **int** listSize, **iNode** x){

/\* Debug Here

System.out.println("In the insert method table length = " + myTable.length);

System.out.println("n = " + n);

System.out.println("min = " + min);

System.out.println("max = " + max);

System.out.println("listSize = " + listSize);

\*/

**String** **temp** = "{";

**iNode** **copy** = sentence;

/\* Rabbit is at the end therefore terminate \*/

**if**(rabbit == listSize+1){

// System.out.println("Max Cap hit: ");

**return** **null**;

}

**else**{

**for**(**int** **i** = turtle; i < rabbit; i++){

temp = temp + copy.element;

copy=copy.next;

}

temp = temp + "}";

x = **new** iNode(temp, x);

**while**(x!= **null**){

**System**.***out***.println(" S" + count + "= " + x.element);

count++;

**int** **key** = hashCode(x.element);

**int** **hashKey** = key % (listSize);

**System**.***out***.println(" Hash-key = " + key + " % " + listSize + " = " + hashKey);

hashTable(key, hashKey);

x = x.next;

}

}

sentence= sentence.next;

buildSublists(sentence, turtle+1, rabbit+1, listSize, x);

**return** temp;

}

**private** **void** **hashTable**(**int** key, **int** hashKey) {

**System**.***out***.println(" Hash Table = [ Key-> " + key + ", hashkey-> " + hashKey + " ]");

**if**(myTable[hashKey].isEmpty()){

**System**.***out***.println( "Insert key " + key + " at the " + hashKey + "th position of the array. ");

myTable[hashKey].add(key);

}

**else**{

numCollisions++;

**int** **counter** = 0;

myTable[hashKey].add(counter, key);

counter++;

**System**.***out***.println(" The number of collisions: " + numCollisions);

}

}

**private** **int** **hashCode**(**Object** cargo) {

**int** **temp** = 0;

**int** **finalInt** = 0;

**String** **s** = (**String**) cargo;

**for**(**int** **i** = 1 ; i < s.length (); i ++){

**char** **character** = s.charAt(i);

**int** **ascii** = (**int**) character; //convert the first character

finalInt = finalInt + ascii;

}

**return** finalInt;

}

**public** **void** **print**(){

**for**(**int** **i** = 0; i < myTable.length; i++){

**System**.***out***.println("hashtable[" + i + "] = " + myTable[i] );

}

**package** lab4;

**public** **class** **iNode** {

**Object** element;

**iNode** next;

**public** **iNode** () {

element = 0;

next = **null**;

}

**public** **iNode**(**String** givenval){

element = givenval;

}

**public** **iNode** (**Object** element, **iNode** next) {

**this**.element = element;

**this**.next = next;

}

**public** **int** **size**(**iNode** x){

**int** **size** = 0;

**while**(x!=**null**){

size++;

x = x.next;

}

**return** size++;

}

**public** **String** **toString** () {

**return** element + "";

}

**public** **Object** **getData**(){

**return** element;

}

**public** **iNode** **getNext**(){

**return** next;

}

**public** **iNode** **prev**(**iNode** head, **iNode** tail){

**iNode** **previous** = **null**;

**while**(head != **null**){

**if**(head.next == tail){

previous = head;

**return** previous;

}

head = head.next;

}

**return** previous;

}

**public** **static** **void** **printList**(**iNode** x){

**if** (x!= **null**){

**System**.***out***.print(x.element+" ");

*printList*(x.next);

}

}

}

Conclusion:

Attempting to build a hash table has given me the opportunity to understand more how n-grams are very much different than finding the permutations or combinations of a given list. Surprisingly I thought I had finished the lab the first day but soon enough I realized I had done everything wrong but I worked on it again and fixed everything. I also realized how fast these hash tables are but one of the biggest disadvantage is building the perfect hash function to equally distributed the data among the array. So a hash table can be very fast but if its implemented poorly the worst case would be having one array position with the whole list.

Academic Honesty Certification

“I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.”

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_