**Lab 5**

**Introduction** – In this lab I are introduced to coding with heaps and hash tables. I are first asked to draw a heap using an StdDraw.java file correctly basing the code off the draw\_tree method that I have been using before for binary search tree. I am asked to do some to create a simplified version of this system that can understand human languages in written and spoken forms. Use the word embeddings to see in such a way that if two words are used in similar contexts and how. With the file provided to us I are to see how the words are used in a similar context and provide extra information regarding the use of these words.

**drawHeapTree (main)** – This method accepts the heap, accept a variable to keep track of the index(which should start at 1 since if I start at zero that would be the length and not the actual heap), two more variables to hold the farthest left and right x-coordinate to know the horizontal boundary of where I are to draw the heap tree, a variable to know how long our vertical boundary , and lastly one more variable to keep track of how many levels the heap tree has. Our base case which is if the index is bigger than the length of the tree (provided by **H[0]**) I are to stop drawing the tree as it has exceeded the amount of nodes I are to draw. Then I are to get the use a variable to keep track of the middle of the canvas I are drawing in and another variable to keep track of which y-coordinate I are to draw in. I would have two conditions which is if it goes to the left or it goes to the right. For the left child I used the **index \* 2** as this would draw the child in the left and **index \* 2 + 1** to draw the right child. I would then draw a that was left alone from the original **drawHeapTree** method and for the recursion call I would make two changes, one instead of passing a binary search tree node I would pass the heap and two was to pass the index depending on which condition is met. If it Ire to be draw on the left, I am to pass **index \* 2** in the recursion call and **index \* 2 + 1** for the right side. I am to also draw the circle and the correct value whenever a recursion call is done, in which it should be the name of the heap, then the array and whatever the value of index is, in my case it would be **heap.H[index]**.

**getHeight (heap.java) –** This method accepts nothing and would get the height of the heap tree. I would create a local variable to keep track of the height. I would then traverse the heap tree array by using **index \* 2**. So, I would make a loop that would start at one (ignoring the length) and end when it reaches the endpoint of the heap and increment by multiplying the index by two as this would always go left. The reason for this as the left most “node” or index when physically drawing a tree will always be the at the maximum height for the heap tree.

**Pause (main) –** In this method I wouldn’t accept any values and primary use it to show every step of the heap being drawn. I would accept an input and enter would be needed to be pressed to continue, printout what input to accept and lastly just clear the canvas I was drawing.

**Experiment:** Time done in Nano Time

|  |  |  |  |
| --- | --- | --- | --- |
| Heap size | 10 | 100 | 1000 |
| Drawing |  |  |  |
| Time | 52470566 | 196622402 | 236474564 |

**createHTableFromFile (Main) –** This method accepts nothing and would return a hash table that was created from reading a file. First, I create the hash table (with a prime number as the size) so I can populate it and return it, a float array to hold the numbers that are within the file, a variable that will hold everything in one line as a string, and lastly a string array to split everything from the previous variable to modify it and manipulate it. I would first try to read the file once I read the file I would get the first line and split it into the string array, so I can convert it into a float array which is what **stringArrToFloatArr** method is for. Once I converted the string array to a float array I would insert it into the hash table which also including the first piece of the line which contains text and not numbers. I would insert this using the **insert** method inside our **hashTableString** file. I would continue this until the variable that holds a whole line is completely empty telling me that the file has been fully read. Lastly, I would return the hash table that I had created.

**Insert (hashTableStrings)** – This method takes in a string and a float array as parameters and would insert a **sNode** into hashTable. This method would first check if the amount of total elements divided by the height is bigger than two if so it would run the method **doubleSize()**. Afterwards, it would get the position of where the string parameters would be inserted using the **h** method and increase the size of the total elements by one everything its inserted.

**Insert (hashTableStrings) –** An auxiliary method that would accept a **sNode** and when it is called when I just want to insert the **sNode** into the hash table.

**doubleSize (hashTableStrings) –** Double size accepts nothing and returns nothing and would double the size of the hash table and assign the created hash table to the current one. First, I create a new hash table that is double the size of the original one and add one to make sure it’s a prime number. I would then traverse the whole original hash table and re-insert all the nodes inside the hash table into the new hash table being created. After this I would assign the new hash table to the old one and assign the number of elements in the new hash table to the old hash table.

**loadFactor (hashTableStrings)** **–** The load factor accepts nothing and returns nothing. This calculates the load factor of the hash table. I would first create a local variable to hold the number of current items inside the hash table whenever its called. I would traverse the whole hash table and increase the count per item in the hash table. I would then return value of count being divided by length of the hash table.

**Search (hashTableStrings)** **–** This method accepts a string parameter and searches the given string in the hash table and return either the node that is has the same string as the parameters or **null** if it doesn’t find it. I would create a variable that would hold the key or position of where I are to search inside the hash table. I would create a loop that would traverse the hash table at the given position and if any of the **sNode** at that certain position that of the given parameter I would return that **sNode** or return null if I don’t find it.

**h –** This method accepts a string as a parameter and would return an integer or the position its suppose to be in the hash table. I would create a local variable to hold the position of where the accept string would be in. I would then loop for as many letters as the string parameter has and I would convert it into an ascii value and mod it by the length of the hash table and assign it to H and return H.

**standardDeviationFunc (Main) –** This method accepts a hash table and prints out what the standard deviation. I would create a double array that would hold how many nodes are inside of each index of the hash table. I would traverse the whole hash table to see how many nodes are inside each index of the hash table and assign the count to the double array respective fully. Then I would create a local variable **f1** to keep track of the first pass of calculating the standard deviation function. I would sum up all the values inside the double array to the local variable created and divide by the length of the double array. I would then create a loop that would subtract the all values inside the double array by the local variable and raise it to the poIr of 2 and sum it all up to another local variable called **f2** and then find the square root of **f2** and this would be the standard deviation.

**compareHTableWords (main) –** This method accepts a hash table and reads another file containing pairs of words (two words per line) and for every pair of words find and display the “similarity” of the words. I would create variables that would hold the line that is read from the file, an array to separate the line into different parts to use and a string that holds the place the while is at. We would read the file then and while we the line we read wasn’t empty we would split the line into an array to easily use them. We would then have to variables that would hold the two words we are to compare. I would then search for the sNode that would contain the same word as the one found inside the while and I would do this for both the first word and the second word contained in one line. Then I call the method **cosineSimilarityFunc** with the parameters of the two node that contain the first and second wordand that would find the similarity between both words and print the first and second word plus the similarity between both words. I would then assign then proceed to the next line and continue until we can’t the line we read isn’t empty.

**cosineSimilarityFunc (main) –** This method accepts two parameters which are both **sNodes** and prints the returns the similarity between them. We would create 3 variables, **aX, bX**, and **top** which are used to save the values when the cosine similarity function is being called. We first have a loop that would run for the entire length of the float array contained inside the **sNode**. We would multiply both the embeddings (float arrays) of the words together to the value of **top**. For **aX** I would raise each of the values of the embedding it to the power of 2 and add the resulting values to **aX** and I would repeat the same step with **bX**. I would then find the square root of both **aX** and **bX** and assign its value back itself respectively. Lastly, I would return the **top** divided by **aX** times **bX**.

**percentageOfEmptyLists (main) –** This would accept a hash table and traverse through the hash table to find if any of indexes of the hash table are empty. I would create a local variable that would keep track of how many of the lists are empty. I would then traverse the hash table but not the nodes inside the hash table and check if any of them are null. If they are I would increment the local variable and return the variable divided by the length of the hash table and make sure to type cast it to float.

**Conclusion –** This lab was confusing as it has aspects of thing I didn’t have an idea from, from cosine similarities to the standard deviation of certain numbers. This was a fun and challenging lab as it included some aspect of how certain natural language programs work. I was able to understand heaps and hash tables a lot more now that I was able to work with them.

1. import java.io.BufferedReader;
2. import java.io.FileReader;
3. import java.io.IOException;
4. import java.util.Scanner;
5. public class lab5 {
6. /\*\*  \* Obtains a string array and converts its contents into floats  \* @param S  \* @return   \*/
7. public static float[] stringArrToFloatArr(String[] S) {
8. float[] convertedArray = new float[S.length]; //create array
9. for (int i = 1; i < S.length; i++) {
10. convertedArray[i] = Float.parseFloat(S[i]); //convert
11. }
12. return convertedArray; //return
13. }
14. /\*\*  \* Creates a hash table given the text file.     \* @return returns a hash table  \*/
15. public static hashTableStrings createHTableFromFile() {
16. hashTableStrings hashTable = new hashTableStrings(7); //create hashtable
17. float[] floatArray; //create float array for numbers
18. String[] holdingArray; //create array for holding items from text file
19. String word; //first text item within each line
20. String line; //a whole line
21. String filename = "D:\\OneDrive\\Documents\\UTEP\\Computer Science\\CS2302\\Lab 5\\src\\glove.6B.50d.txt";
22. try {
23. BufferedReader br = new BufferedReader(new FileReader(filename)); //read file
24. line = br.readLine();
25. while (line != null) {
26. holdingArray = line.split(" ");
27. floatArray = stringArrToFloatArr(holdingArray); //convert to float
28. hashTable.insert(holdingArray[0], floatArray); //insert the text and the embedding
29. line = br.readLine();
30. }
31. br.close();
32. } catch (Exception e) //catch errors
33. {
34. System.out.println("Error: " + e);
35. }
36. return hashTable;
37. }
38. /\*\*  \* Auxiliary Method for {@link#cosineSimilarityFunc(sNode, sNode)}   \* @param hashTable  \*/
39. public static void compareHTableWords(hashTableStrings hashTable) {
40. String[] holdingArray = null;
41. String line = "";
42. String filename = "D:\\OneDrive\\Documents\\UTEP\\Computer Science\\CS2302\\Lab 5\\src\\appendix.txt"; //file location
43. try {
44. BufferedReader br = new BufferedReader(new FileReader(filename)); //reat the file
45. line = br.readLine();
46. while (line != null) {
47. holdingArray = line.split(" "); //split the array
48. String firstComp = holdingArray[0]; //get the first word
49. String secondComp = holdingArray[1]; //get the second word
50. sNode firstWord = hashTable.search(firstComp);
51. sNode secondWord = hashTable.search(secondComp);
52. float similarity = cosineSimilarityFunc(firstWord, secondWord); //obtain the similiarity value
53. System.out.println("Similarity [" + firstComp + ", " + secondComp + "] = " + similarity); //display
54. line = br.readLine();
55. }
56. br.close();
57. } catch (Exception e) //catch any errors
58. {
59. System.out.println("Error: " + e);
60. }
61. }
62. /\*\*  \* Reads a file containing pairs of words (two words per line) and for every pair of words find and  \* display the â€similarityâ€ of the words.    \* @param w0 left word   \* @param w1 right word  \* @return   \*/
63. public static float cosineSimilarityFunc(sNode w0, sNode w1) {
64. float aX = 0;
65. float bX = 0;
66. float top = 0;
67. for (int i = 0; i < w0.embedding.length; i++) {
68. top += w0.embedding[i] \* w1.embedding[i];
69. }
70. for (int i = 0; i < w0.embedding.length; i++) {
71. aX += Math.pow(w0.embedding[i], 2);
72. }
73. for (int i = 0; i < w1.embedding.length; i++) {
74. bX += Math.pow(w1.embedding[i], 2);
75. }
76. aX = (float) Math.sqrt(aX);
77. bX = (float) Math.sqrt(bX);
78. return (top) / (aX \* bX);
79. }
80. /\*\*  \* Traverse through the hash table to find if any of indexes of the hash table are empty.    \* @param hashTable  \*/
81. public static float percentageOfEmptyLists(hashTableStrings hashTable) {
82. int countEmpty = 0;
83. for (int i = 0; i < hashTable.H.length; i++) {
84. if (hashTable.H[i] == null) //checks if the hash table is empty
85. {
86. countEmpty++;
87. }
88. }
89. return (float) countEmpty / hashTable.H.length;
90. }
91. /\*\*  \* Obtain the standard Deviation from the following array given.     \* @param hashTable for length of the linked list inside the hash table  \*/
92. public static void standardDeviationFunc(hashTableStrings hashTable) {
93. double[] stndDev = new double[hashTable.H.length];
94. for (int i = 0; i < hashTable.H.length; i++) {
95. double countList = 0.0; //keep track of how many nodes their are in the index i of the hash table
96. for (sNode T = hashTable.H[i]; T != null; T = T.next) //traverses through the nodes inside the hash table
97. {
98. countList++;
99. }
100. stndDev[i] = countList;
101. }
102. float f1 = 0;
103. for (int i = 0; i < stndDev.length; i++) {
104. f1 += stndDev[i];
105. }
106. f1 /= stndDev.length;
107. float f2 = 0;
108. for (int i = 0; i < stndDev.length; i++) {
109. f2 += (float) Math.pow(stndDev[i] - f1, 2);
110. }
111. f2 /= stndDev.length;
112. System.out.println("Standard Deviation: " + Math.sqrt(f2));
113. }
114. public static void drawHeapTree(heap heap, int index, double x0, double x1, double y, double y\_inc) {
115. if (index > heap.H[0]) //Base case
116. {
117. return;
118. }
119. double xm = (x0 + x1) / 2;
120. double yn = y - y\_inc;
121. if (index \* 2 <= heap.H[0]) //Checks the left child
122. {
123. StdDraw.line(xm, y, (x0 + xm) / 2, yn);
124. drawHeapTree(heap, index \* 2, x0, xm, yn, y\_inc);
125. }
126. if (index \* 2 + 1 <= heap.H[0]) //Checks the right child
127. {
128. StdDraw.line(xm, y, (x1 + xm) / 2, yn);
129. drawHeapTree(heap, index \* 2 + 1, xm, x1, yn, y\_inc);
130. }
131. StdDraw.setPenColor(StdDraw.WHITE);
132. StdDraw.filledCircle(xm, y, 3);
133. StdDraw.setPenColor(StdDraw.BLACK);
134. StdDraw.circle(xm, y, 3);
135. StdDraw.text(xm, y, Integer.toString(heap.H[index])); //Prints the value
136. }
137. public static void pauseAndClear() {
138. Scanner s = new Scanner(System.in);
139. System.out.println("Press enter to continue.....");
140. s.nextLine();
141. StdDraw.clear();
142. }
143. public static void main(String[] Args) throws IOException {
144. int x\_max = 100;
145. int y\_max = 100;
146. StdDraw.setXscale(0, x\_max);
147. StdDraw.setYscale(0, y\_max);
148. StdDraw.setPenColor(StdDraw.BLACK);
149. int[] heapArray = {
150. 1, 2, 3, 4, 5, 6, 7, 8
151. };
152. heap heapObject = new heap(50);
153. for (int i = 0; i < heapArray.length; i++) {
154. heapObject.insert(heapArray[i]);
155. pauseAndClear();
156. drawHeapTree(heapObject, 1, 0, x\_max, y\_max - 5, (y\_max - 10.0) / heapObject.getHeight());
157. }
158. hashTableStrings hashTable = new hashTableStrings(7);
159. hashTable = createHTableFromFile();
160. System.out.println("Word Similarities: ");
161. compareHTableWords(hashTable);
162. System.out.println("Load factor: " + hashTable.loadFactor());
163. System.out.println("Percentage of empty lists:" + percentageOfEmptyLists(hashTable) \* 100);
164. standardDeviationFunc(hashTable); //Obtains the standard deviation
165. }
166. }
167. public class hashTableStrings {
168. public sNode[] H;
169. public int nElements = 0;
170. /\*\*  \* Creates a hash table with the given parameters.   \* @param n length of the hash table     \*/
171. public hashTableStrings(int n) {
172. H = new sNode[n];
173. for (int i = 0; i < n; i++) {
174. H[i] = null;
175. }
176. }
177. /\*\*  \* Obtains the key from a string     \* @param S any string   \* @return   \*/
178. private int h(String S) {
179. int h = 0;
180. for (int i = 0; i < S.length(); i++) {
181. h = (h \* 27 + S.charAt(i)) % H.length;
182. }
183. return h;
184. }
185. /\*\*  \* Searches the linked list for the given string     \* @param S any string   \* @return   \*/
186. public sNode search(String S) {
187. int k = h(S);
188. for (sNode T = H[k]; T != null; T = T.next) {
189. if (S.equals(T.word)) {
190. return T;
191. }
192. }
193. return null;
194. }
195. /\*\*  \*   \* @param text   \* @param x  \*/
196. public void insert(String text, float[] x) {
197. if (nElements / H.length > 2) {
198. doubleSize();
199. }
200. int pos = h(text);
201. H[pos] = new sNode(text, x, H[pos]);
202. nElements++;
203. }
204. /\*\*  \* Insert a node into the hash table     \* @param t  \*/
205. private void insert(sNode t) {
206. insert(t.word, t.embedding);
207. }
208. /\*\*  \* Obtains the load factor   \* @return   \*/
209. public float loadFactor() {
210. int count = 0;
211. for (int i = 0; i < H.length; i++) {
212. for (sNode t = H[i]; t != null; t = t.next) {
213. count++;
214. }
215. }
216. return (float) count / H.length;
217. }
218. /\*\*  \*   \*/
219. private void doubleSize() {
220. hashTableStrings hashTableTemp = new hashTableStrings((H.length \* 2) + 1);
221. for (int i = 0; i < H.length; i++) {
222. for (sNode t = H[i]; t != null; t = t.next) {
223. hashTableTemp.insert(t);
224. }
225. }
226. H = hashTableTemp.H;
227. nElements = hashTableTemp.nElements;
228. }
229. }
230. public class heap {
231. public int[] H;
232. public heap(int n) {
233. H = new int[n];
234. H[0] = 0;
235. }
236. public int insert(int element) { //Check if heap is full
237. if (H[0] >= H.length - 1) {
238. return -1;
239. } //Increase the size of the heap
240. H[0] += 1;
241. int i = H[0]; //Bubble up
242. while ((i > 1) && (element < H[i / 2])) {
243. H[i] = H[i / 2];
244. i = i / 2;
245. } //Insert the elemnt
246. H[i] = element;
247. return i;
248. }
249. public int getHeight() {
250. int count = -1;
251. for (int i = 1; i <= H[0]; i \*= 2) { //traverse the left
252. count++;
253. }
254. return count;
255. }
256. }

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.

Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_