Assignment 2 - COSC 3320

Alex Bennett (ID: 1901408)

Theory Problem 1

(a)

To check if any element in L_1 is in L_2 we must iterate over each element of L_1 (size n), and then compare each element of L_1 with every element of L_2 (size n). This would be comprised of two nested loops and thus would have a lower bound of $O(n^2)$.

(b)

The algorithm to implement the above is quite simple: we iterate over one list, starting with the head node, and compare that node with the nodes (not elements) of the second until we reach the end of the second list. If at any point there's a successful comparison, we return true, otherwise return false. Since we use two nested while-loops that iterate over the two lists the time complexity is $O(n^2)$ in the worst case.

The algorithm in pseudocode:

```
1  Linked_List_Compare(list L_1, list L_2) {
2
3          node temp1 = L_1.head;
4
5          while (temp1 != NULL) {
6                  node temp2 = L_2.head;
7
8                  while(temp2 != NULL) {
9                          if (temp1 == temp2) {
10                                 return true
11                         }
```

```
                        temp2 = temp2->next;
                }
                temp1 = temp1->next;
        }
        return false;
}
```

Theory Problem 2

The insert and delete functions will be identical to the standard implementation of AVL trees except we will modify the insert and delete functions to include a *shift*() function that will shift the nodes based on $\lceil \frac{n}{3} \rceil$.

The algorithm for *shift*(node) is: if the node's value is greater than $\lceil \frac{n}{3} \rceil$, then *shift*(node.right), move the node to the left subtree, move node.right to the root, and then rebalance. If the new node's value is less than $\lceil \frac{n}{3} \rceil$, then *shift*(node.left), move the node to the right subtree, move node.left to the root, and then rebalance.

Consequently the *find*($\lceil \frac{n}{3} \rceil$) function can be executed in constant time ($O(1)$) since it only has to access the root node. The tradeoff is that the addition of the *shift*() function to insert and delete makes those function calls slower than $O(\log_2 n)$. The space complexity will be the same for insert and delete as in the original algorithms (that is, $O(\log_2 n)$) while the *find*() function will have a space complexity of $O(1)$.

Theory Problem 3

To determine the average number of scalar multiplications for a sequence of n matrices we will use the following rule we learned in class:

$S[i,i] = 0$
$S[i,i+1] = p_i + p_{i+1} + p_{i+2}$
$S[i,j] = \text{avg}(S[i,k] + S[k+1,j] + p_i + p_{k+1} + p_{k+1}),\ \text{for } i \leq k \leq j-1$

Where $S[i,j]$ is the matrix representing the average work for a parentheses configuration grouping every element from i to j and p_i represents the ith dimension of the original matrix sequence. The "average work" in this case

is equal to the sum of scalar multiplications for possible k-values divided by the total number of k-values.

The algorithm to generate those values for S takes an integer array as its argument (the dimensions of the different arrays being multiplied), calculates $S[i,k]+S[k+1,j]+p_i+p_{k+1}+p_{k+1}$ for each k-value using three for-loops while keeping track of the number of k-values iterated over each loop, then stores the sum of different values of $S[i,k]+S[k+1,j]+p_i+p_{k+1}+p_{k+1}$ for each k-value divided by the total number of k-values into the corresponding entry of S. Written in pseudo-code we get:

```
1  average_operations(int[] p) {
2
3          int n = p.length - 1;
4          avg =  new int[n][n];
5
6          for (int len = 1;  len < n; len++) {
7                  for (int i = 0; i < n - len; i++) {
8                          int j = i + len;
9
10                         int k_count = 0;
11                         for (int k = i; k < j; k++) {
12                                 avg[i][j] += avg[i][k] + avg[k+1][j] +
                                   ↪  p[i]*p[k+1]*p[j+1];
13                                 k_count++;
14                         }
15                         avg[i][j] /= k_count;
16                 }
17         }
18
19         return avg;
20 }
```

Since this algorithm will be iterated over 3 nested for-loops the time complexity is $O(n^3)$. Since S is two dimensional our space complexity is $O(n^2)$

Programming Problem 1

Procedure

In this program we use three two-dimensional arrays X, Y, and Z, each of size n, with X and Y initialized to 1 for each entry. We then add corresponding entries of X and Y together via matrix addition to generate an entry in Z. We use two different algorithms to perform the addition, the first traversing the arrays in row-major order and the second traversing them in column-major order, and compare their timings. The program is written in C++ and run on a MacOS supporting VMM.

Hypothesis

Given that both algorithms have the same number of operations, one might expect that the timings for the two algorithms will be similar. However, as we've studied in class, if a multidimensional array is stored in memory using row-major order then row-major traversal is significantly faster than column-major traversal. Given that C++ must store these arrays in either row-major or column-major order, my hypothesis is that there will be significant difference in timing between the two algorithms.

C++ Implementation

```
1  #include <stdio.h>
2  #include <time.h>
3  #include <stdlib.h>
4
5  int main(int argc, char *argv[]) {
6
7          char *arg = argv[1];
8          int n = atoi(arg);
9
10         int** X = new int*[n];
11         int** Y = new int*[n];
12         int** Z = new int*[n];
13
14         //Initialize X, Y, Z
15         for (int i = 0; i < n; i++) {
16                 X[i] = new int[n];
```

```
                Y[i] = new int[n];
                Z[i] = new int[n];
        }

        for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                        X[i][j] = 1;
                        Y[i][j] = 1;
                }
        }

        //Version 1
        clock_t begin1 = clock();
        for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                        Z[i][j] = X[i][j] + Y[i][j];
                }
        }
        clock_t end1 = clock();

        //Version 2
        clock_t begin2 = clock();
        for (int j = 0; j < n; j++) {
                for (int i = 0; i < n; i++) {
                        Z[i][j] = X[i][j] + Y[i][j];
                }
        }
        clock_t end2 = clock();

        int time1 = (double)(end1 - begin1)/(CLOCKS_PER_SEC/1000);
        int time2 = (double)(end2 - begin2)/(CLOCKS_PER_SEC/1000);

        printf("Version 1: %d milliseconds\n", time1);
        printf("Version 2: %d milliseconds\n", time2);

}
```

Results

Dimension of Array	Time of Vers. 1 (ms)	Time of Vers. 2 (ms)
128	0	0
256	0	0
512	0	3
1024	5	22
2048	20	108
4096	81	545
8192	325	2609
16384	1421	15618
32768	11147	109160
65536	46973	824990

Explanation

My hypothesis was correct: there was a substantial difference in timing between Version 1 and Version 2, with Version 2 significantly slower. The difference is marginal for $n = 512$ but becomes more noticeable at $n = 1024$. For Version 2 the timing for $n = 2048$ is over five times as slow as $n = 1024$, which is greater than the expected quadrupling from the algorithm. By $n = 65536$, the increase is seven and a half times greater than the previous value of n. The rate of increase for Version 2 is linear. Compare this to Version 1, the rate of increase of which is a constant quadrupling.

Row-major and column-major traversals of an array can vary quite significantly in their timings depending on how a programming language stores multidimensional arrays in linear storage. Since Version 1 (row-major) was much faster than Version 2 (column-major), we can conclude that C++ stores multidimensional arrays in row-major order. In fact this is true for most programming languages of the C family.

Programming Problem 2

Procedure

In this program we take an n by n matrix, initializing each entry to zero, then perform m simple operations (adding a random number between 0-100 to a random entry in the matrix) for $m = 1677721600$ and $m = 13421772800$. The program is written in Java and run on a MacOS supporting VMM.

Hypothesis

The algorithm is fairly simple, performing m computations on the same matrix, so we might expect a time complexity of $O(m)$. However since we will be dealing with fairly large matrices, I expect that for certain values of n the program will exceed physical memory and thus require the VMM to transfer data to disk storage. My hypothesis is that the timing of the program will jump up dramatically for a particular n value, and all larger n will take significantly more time.

Java Implementation

```
1  import java.util.Random;
2
3  public class Prog2 {
4
5          public static void main (String[] args) {
6
7                  int n = Integer.parseInt(args[0]);
8
9                  int[][] M = new int[n][n];
10
11                 for (int k = 0; k < n; k++) {
12                         for (int q = 0; q < n; q++) {
13                                 M[k][q] = 0;
14                         }
15                 }
16
17                 long start1 = System.currentTimeMillis();
18                 long m_1 = 1677721600;
19                 Random r = new Random();
20
21                 for (long k = 0; k < m_1; k++) {
22                         int x = r.nextInt(100) + 1;
23                         int j = r.nextInt(n);
24                         int i = r.nextInt(n);
25                         M[i][j] = M[i][j] + x;
26                 }
27
28                 long end1 = System.currentTimeMillis();
```

```
29
30                  long start2 = System.currentTimeMillis();
31                  long m_2 = 13421772800L;
32
33                  for (long k = 0; k < m_2; k++) {
34                          int x = r.nextInt(100) + 1;
35                          int j = r.nextInt(n);
36                          int i = r.nextInt(n);
37                          M[i][j] = M[i][j] + x;
38                  }
39
40                  long end2 = System.currentTimeMillis();
41
42                  System.out.println("Time elalpsed for n=" + n + " for m_1:
                    ↪  " + (end1 - start1) + " milliseconds");
43                  System.out.println("Time elalpsed for n=" + n + " for m_2:
                    ↪  " + (end2 - start2) + " milliseconds");
44
45
46          }
47  }
```

Results

n	Time for m=1677721600 (ms)	Time for m=13421772800 (ms)
16	55336	430628
64	55951	424845
256	59988	445868
1024	121313	795774
4096	232810	1858610
16384	286520	2290697

Explanation

My hypothesis was correct: for both values of m, the timing increases significantly for $n = 1024$. For all values of n the timing of $m = 13421772800$ is roughly six to eight times $m = 1677721600$. For smaller values of the n, the timing is dependent on m (the expected $O(m)$ complexity) with little variation variation between m-timings for different n values. However at

$n = 1024$ we see both m-timings increase linearly. Interestingly for both m values the jump between $n = 1024$ and $n = 4096$ was much greater than the jump between $n = 4096$ and $n = 16384$.

As I predicted in the hypothesis, the program ran out of physical memory and had to utilize the VMM to transfer data stored in RAM into disk storage. This swapping time accounts for the dramatic increase in execution time for $n \geq 1024$.

Programming Problem 3

Procedure

In this program we implement the AVL tree data structure with a large matrix as data for each node. The size of the matrix is determined by a randomly generated number modulo 3. The algorithm itself initializes 50 nodes to produce a tree of height 6, and then performs 100,000 random insertions and removals into the tree and compares their timing to the initial insertion of the 50 nodes into the tree.

Hypothesis

My hypothesis is that insertion and removal will take roughly the same amount of time. The time complexity for the standard insertion and removal functions in an AVL tree is $O(\log n)$ and so averaged out over 100,000 operations their timing should end up being fairly similar.

Java Implementation

```
1  import java.util.Random;
2
3  class Node {
4          int val;
5          int[] matrix;
6          Node left;
7          Node right;
8          int height;
9  }
10
11 class AVLtree {
```

```
12          int M_0 = (int) Math.pow(2, 20);
13          int M_1 = (int) (Math.pow(2, 19) + Math.pow(2, 18));
14          int M_2 = (int) (Math.pow(2, 18) + Math.pow(2, 17));
15          int inserts = 0;
16          int deletes = 0;
17          int nodes = 0;
18          Node root;
19
20          public void AVLtree() {
21                  this.root = null;
22          }
23
24          public void AVL_insert(int a) {
25                  this.root = insert(a, this.root);
26          }
27
28          public void AVL_remove(int a) {
29                  this.root = remove(a, this.root);
30          }
31
32          public int getNodeCount() {
33                  return this.nodes;
34          }
35
36          public int getInserts() {
37                  return this.inserts;
38          }
39
40          public int getDeletes() {
41                  return this.deletes;
42          }
43
44          public void resetInsertCount() {
45                  this.inserts = 0;
46          }
47
48          private Node insert(int a, Node b) {
49                  int mod = a % 3;
50
51                  if (b == null) {
```

```
52                      this.inserts++;
53                      b = new Node();
54                      this.nodes++;
55
56                      b.height = 0;
57                      b.left = null;
58                      b.right = null;
59
60                      if (mod == 0) {
61                              b.matrix = new int[M_0];
62                      } else if (mod == 1) {
63                              b.matrix = new int[M_1];
64                      } else {
65                              b.matrix = new int[M_2];
66                      }
67              } else if (a < b.val) {
68                      b.left = insert(a, b.left);
69
70                      if (height(b.left) - height(b.right) == 2) {
71                              if (a < b.left.val) b =
                                ↪  singRightRotate(b);
72                              else b = doubRightRotate(b);
73                      }
74              } else if (a > b.val) {
75                      b.right = insert(a, b.right);
76                      if (height(b.right) - height(b.left) == 2) {
77                              if (a > b.right.val) b =
                                ↪  singLeftRotate(b);
78                              else b = doubLeftRotate(b);
79                      }
80              }
81
82              b.height = Math.max(height(b.left), height(b.right)) + 1;
83              return b;
84      }
85
86      private Node singRightRotate(Node b) {
87
88              if (b == null || b.left == null) {
89                      return b;
```

```
90                  } else {
91                          Node temp = b.left;
92                          b.left = temp.right;
93                          temp.right = b;
94                          b.height = Math.max(height(b.left),
                            ↪  height(b.right)) + 1;
95                          temp.height = Math.max(height(temp.left),
                            ↪  b.height) + 1;
96                          return temp;
97                  }
98          }
99
100         private Node singLeftRotate(Node b) {
101
102                 if (b == null || b.right == null) {
103                         return b;
104                 } else {
105                         Node temp = b.right;
106                         b.right = temp.left;
107                         temp.left = b;
108                         b.height = Math.max(height(b.left),
                            ↪  height(b.right)) + 1;
109                         temp.height = Math.max(height(b.right), b.height)
                            ↪  + 1;
110                         return temp;
111                 }
112         }
113
114         private Node doubLeftRotate(Node b) {
115                 b.right = singRightRotate(b.right);
116                 return singLeftRotate(b);
117         }
118
119         private Node doubRightRotate(Node b) {
120                 b.left = singLeftRotate(b.left);
121                 return singRightRotate(b);
122         }
123
124         private Node remove(int a, Node b) {
125                 Node temp;
```

```
126
127                 if (b == null) return null;
128                 else if (a < b.val) b.left = remove(a, b.left);
129                 else if (a > b.val) b.right = remove(a, b.right);
130                 else if (b.left != null && b.right != null) {
131                         temp = findMin(b.right);
132                         b.val = temp.val;
133                         b.right = remove(b.val, b.right);
134                 } else {
135                         temp = b;
136                         if (b.left == null) b = b.right;
137                         else if (b.right == null) b = b.left;
138
139                         this.nodes--;
140                         this.deletes++;
141                 }
142                 if (b == null) return b;
143
144                 b.height = Math.max(height(b.left), height(b.right)) + 1;
145
146                 if ((height(b.left) - height(b.right)) == 2) {
147                         if ((height(b.left.left) - height(b.left.right))
                        ↪  == 1) return singLeftRotate(b);
148                         else return doubLeftRotate(b);
149                 } else if ((height(b.right) - height(b.left)) == 2) {
150                         if ((height(b.right.right) - height(b.right.left))
                        ↪  == 1) return singRightRotate(b);
151                         else return doubRightRotate(b);
152                 }
153                 return b;
154         }
155
156         private Node findMin(Node b) {
157                 if (b == null) return null;
158                 else if (b.left == null) return b;
159                 else return findMin(b.left);
160         }
161
162         private int height(Node b) {
163                 return (b == null ? -1 : b.height);
```

```
164         }
165 }
166
167 public class Prog3 {
168
169         public static void main(String[] args) {
170
171                 AVLtree tree = new AVLtree();
172                 Random r = new Random();
173
174                 double begin = System.currentTimeMillis();
175
176                 while(tree.getNodeCount() < 50) {
177                         tree.AVL_insert(r.nextInt(299));
178                 }
179
180                 double initialTime = System.currentTimeMillis() - begin;
181
182                 double insertTime, removeTime;
183                 insertTime = removeTime = 0;
184
185                 tree.resetInsertCount();
186
187                 for (int i = 0; i < 100000; i++) {
188
189                         while (tree.getNodeCount() < 50) {
190                                 begin = System.currentTimeMillis();
191                                 tree.AVL_insert(r.nextInt(299));
192                                 insertTime += (System.currentTimeMillis()
                                ↪  - begin);
193                         }
194
195                         while (tree.getNodeCount() >= 50) {
196                                 begin = System.currentTimeMillis();
197                                 tree.AVL_remove(r.nextInt(299));
198                                 removeTime += (System.currentTimeMillis()
                                ↪  - begin);
199                         }
200                 }
201
```

```
202                 System.out.println("Total time for initial insertion: " +
                    ↪  initialTime + " ms");
203                 System.out.println("Average time for initial insertion: "
                    ↪  + (initialTime/50) + " ms");
204                 System.out.println("Total time for subsequent insertions:
                    ↪  " + insertTime + " ms");
205                 System.out.println("Average time for insertions: " +
                    ↪  (insertTime/tree.getInserts()) + " ms");
206                 System.out.println("Total time for removals: " +
                    ↪  removeTime + " ms");
207                 System.out.println("Average time for removals: " +
                    ↪  (removeTime/tree.getDeletes()) + " ms");
208         }
209 }
```

Results

Total time for initial insertions	88 ms
Average time for initial insertion	1.76 ms
Total time for subsequent insertions	26348 ms
Average time for subsequent insertions	0.26348 ms
Total time for removals	3338 ms
Average time for removals	0.03338 ms

Explanation

My hypothesis was incorrect. Insertion took more time than removal both overall and on average. This might be due to memory fragmentation, as the insertion function requires memory to be allocated for additional nodes and thus potentially extensive paging if the program exceeds RAM. The removal function would not have this additional memory cost.

The average time for the initial insertion was greater than the average time for subsequent insertions as well. Apparently this is because the Java Virtual Machine uses an algorithm that increases the number of sweeps in garbage collection as the amount of the JVM's heap memory increases. Thus up to a point the process of removing and inserting became faster as the tree became larger. This is a testament to the JVM's successful handling of memory fragmentation and compaction.

Programming Problem 4

Procedure

In this program we generate an array of size $C = val * M$, where M is the size of the active memory set and val is one of a list of values between 0.5 and 50. We perform many small operations on the array extensively, measuring the timing and amount of memory utilized in the program. The program is written in Java and run on a MacOSX supporting VMM.

Hypothesis

The program is designed such that any value of $C < M$ will be easily handled by the VMM, but for $C > M$ the VMM will be constantly paging and thus thrashing will likely occur. As such my hypothesis is that the program's timing will increase considerably when $C > M$.

Java Implementation

```
1  import com.sun.management.OperatingSystemMXBean;
2  import java.lang.management.ManagementFactory;
3
4  public class Prog4 {
5
6          public static void main (String[] args) {
7
8                  OperatingSystemMXBean opsys =
                   ↪  ManagementFactory.getPlatformMXBean(OperatingSystemMXBean.class);
9                  System.out.println("Phys Mem: " +
                   ↪  opsys.getFreePhysicalMemorySize());
10                 System.out.println("Virtual Mem: " +
                   ↪  opsys.getCommittedVirtualMemorySize());
11                 System.out.println("Free swap space: " +
                   ↪  opsys.getFreeSwapSpaceSize());
12
13                 double[] C = {0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 0.99, 1.0,
                   ↪  1.01, 1.1, 1.5, 2, 5, 10, 50};
14                 long freeBytes = opsys.getFreePhysicalMemorySize();
15
16                 for (int i = 0; i < 15; i++) {
```

```
17                      System.out.println("----------------------\n" +
                        ↪  "Cache Size: " + C[i] + "*M");
18                      long start = System.currentTimeMillis();
19
20                      int numBytes = Math.abs((int)(C[i] *
                        ↪  (freeBytes)));
21                      int size = numBytes/4;
22                      int[] testArr = new int[size];
23
24                      System.out.println("Phys Mem: " +
                        ↪  opsys.getFreePhysicalMemorySize());
25                      System.out.println("Virtual Mem: " +
                        ↪  opsys.getCommittedVirtualMemorySize());
26                      System.out.println("Free swap space: " +
                        ↪  opsys.getFreeSwapSpaceSize());
27
28                      for (int j = 0; j < size; j++) {
29                              testArr[j] = i + 1;
30                      }
31                      for (int j = 0; j < size; j++) {
32                              testArr[j] -= 2;
33                      }
34
35                      System.out.println("Time elapsed: " + ((double)
                        ↪  System.currentTimeMillis() - start) + "
                        ↪  milliseconds");
36
37              }
38
39      }
40  }
```

Results

Cache Size	Free Physical Mem (bytes)	Free Virtual Mem (bytes)	Free Swap Space (bytes)	Time (ms)
Start	292106240	10460585984	766246912	N/A
$0.5 * M$	142024704	10476437504	766246912	138
$0.6 * M$	105308160	10485899264	766246912	98
$0.7 * M$	92807168	10486038528	766246912	78
$0.8 * M$	78573568	10487099392	766246912	89
$0.9 * M$	78573568	10488160256	766246912	102
$0.95 * M$	78573568	10488160256	766246912	98
$0.99 * M$	78573568	10488160256	766246912	100
$1.0 * M$	78573568	10496548864	766246912	99
$1.01 * M$	78573568	10496548864	766246912	100
$1.1 * M$	78573568	10496548864	766246912	116
$1.5 * M$	21618688	10496548864	766246912	187
$2.0 * M$	22835200	10496548864	766246912	246
$5.0 * M$	23138304	10498646016	766246912	1072
$10.0 * M$	23121920	10499694592	766246912	824
$50.0 * M$	21336064	10499796992	766246912	663

Explanation

My hypothesis was correct. For $C = 0.5 * M$ to $C = 1.1 * M$ the execution time hovers around 100 ms, but the execution time increases considerably once the array size exceeds $1.5 * M$. Changes in virtual memory are minor and no changes occur for swap space memory.

Bizarrely, while the increase in execution time is not exponential, it is not strictly linear either as $5.0 * M$ takes 400 ms longer than $50.0 * M$. This suggests that the MacOSX's VMM replacement policy is effective at dealing with large numbers of swaps and that thrashing is not occurring. Perhaps testing for higher values of C would result in a more strictly linear or exponential increase in execution time.

Programming Problem 5

Description of algorithm

The algorithm for generating an optimal Huffman code works as follows. We first take characters and their frequencies as input from the user. We create tree nodes for each character and its frequency, and store them in a priority

queue sorted by frequency. We then take the two least frequent nodes in the queue and construct a new node connecting the two, whose frequency is derived from their two frequencies combined. This new node is added back into the queue. We continue this process until only one node remains. The resulting tree supplies our Huffman encoding.

Java Implementation

```
1  import java.util.Scanner;
2  import java.util.Comparator;
3  import java.util.ArrayList;
4  import java.util.PriorityQueue;
5
6  class Node {
7          char ch;
8          int freq;
9          Node left;
10         Node right;
11 }
12
13 class HuffComparator implements Comparator<Node> {
14         public int compare(Node a, Node b) {
15                 return a.freq - b.freq;
16         }
17 }
18
19 public class Prog5 {
20
21         public static void main(String[] args) {
22
23                 Scanner reader = new Scanner(System.in);
24                 ArrayList<Character> chars = new ArrayList<Character>();
25                 ArrayList<Integer> frequency = new ArrayList<Integer>();
26                 boolean check = true;
27
28                 while(check) {
29                         System.out.println("Enter character (to exit enter
                           ↪  '0')");
30                         char elem = reader.nextLine().charAt(0);
31
```

```
32                      if (elem == '0') {
33                              check = false;
34                      } else {
35                              System.out.println("What is the frequency
                                ↪  for the character?");
36                              int freq =
                                ↪  Integer.parseInt(reader.nextLine());
37                              chars.add(elem);
38                              frequency.add(freq);
39                      }
40              }
41
42              PriorityQueue<Node> queue = new
                ↪  PriorityQueue<Node>(chars.size(), new
                ↪  HuffComparator());
43
44              for (int i = 0; i < chars.size(); i++) {
45                      Node newNode = new Node();
46
47                      newNode.ch = chars.get(i);
48                      newNode.freq = frequency.get(i);
49                      newNode.right = null;
50                      newNode.left = null;
51
52                      queue.add(newNode);
53              }
54
55              Node root = null;
56
57              while (queue.size() > 1) {
58
59                      Node a = queue.peek();
60                      queue.poll();
61                      Node b = queue.peek();
62                      queue.poll();
63
64                      Node c = new Node();
65
66                      c.freq = a.freq + b.freq;
67                      c.ch = '-';
```

```
68                          c.left = a;
69                          c.right = b;
70                          root = c;
71
72                          queue.add(c);
73
74                  }
75
76                  printCode(root, "");
77
78          }
79
80          public static void printCode(Node root, String code) {
81                  if (root.left == null && root.right == null &&
                    ↪  Character.isLetter(root.ch)) {
82                          System.out.println(root.ch + ": " + code);
83                          return;
84                  }
85
86                  printCode(root.left, code + "0");
87                  printCode(root.right, code + "1");
88          }
89
90  }
```

Sample Huffman Code

For character frequency input:

a—8
v—1
z—1
b—5
c—6
p—4
e—10

Our algorithm generates the Huffman code:

c: 00

a: 01
e: 10
b: 110
v: 11100
z: 11101
p: 1111