Assignment 2 - COSC 3320

Alex Bennett (ID: 1901408)

Theory Problem 1

(a)

To check if an element in L_1 is in L_2 we must iterate over each element of L_1 (n elements), and then compare each element of L_1 with every element of L_2. Thus the lower bound is $O(n^2)$.

(b)

```
1   Linked_List_Compare(list L_1, list L_2) {

2

3         node temp1 = L_1;

4

5         while (temp1 != NULL) {
6               node temp2 = L_2;

7

8               while(temp2 != NULL) {
9                     if (temp1 == temp2) {
10                          return true
11                    }
12                    temp1 = temp1->next;
13              }

14

15              temp1 = temp1->next;
16        }

17

18        return false;
19  }
```

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

Then 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 $O(\log n)$.

Theory Problem 3

To determine the average number of scalar multiplications for a sequence of n matrices we will use the following informal algorithm:

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

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

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;
```

2

```
 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  }
```

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

```cpp
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

        char *arg = argv[1];
        int n = atoi(arg);

        int** X = new int*[n];
        int** Y = new int*[n];
        int** Z = new int*[n];

        //Initialize X, Y, Z
        for (int i = 0; i < n; i++) {
                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();
```

```
39      for (int j = 0; j < n; j++) {
40              for (int i = 0; i < n; i++) {
41                      Z[i][j] = X[i][j] + Y[i][j];
42              }
43      }
44      clock_t end2 = clock();
45
46      int time1 = (double)(end1 - begin1)/(CLOCKS_PER_SEC/1000);
47      int time2 = (double)(end2 - begin2)/(CLOCKS_PER_SEC/1000);
48
49      printf("Version 1: %d milliseconds\n", time1);
50      printf("Version 2: %d milliseconds\n", time2);
51
52  }
```

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. As explained in the hypothesis depending on how programming languages store arrays in memory, row-major and column-major traversals of an array can vary quite significantly in their timings. 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 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. 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 particular n, and all larger n will take significantly more time.

Java Implementation

```java
import java.util.Random;

public class Prog2 {

    public static void main (String[] args) {

        int n = Integer.parseInt(args[0]);

        int[][] M = new int[n][n];

        for (int k = 0; k < n; k++) {
            for (int q = 0; q < n; q++) {
                M[k][q] = 0;
            }
        }

        long start1 = System.currentTimeMillis();
        long m_1 = 1677721600;
        Random r = new Random();
```

```
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 the first three values of n the program took roughly the same amount of time to execute. But with $n = 1024$ the timing increased dramatically. Larger values of n took significantly longer as well. Interestingly 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 increase in execution time for larger values of n.

Programming Problem 3

Procedure

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

Hypothesis

My hypothesis is that insertion of new nodes will generally take longer than removing nodes. This is because the allocation of memory for new nodes will cause memory fragmentation and require garbage collection, whereas the removal of nodes won't be as memory-intensive.

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   }
```

```java
class AVLtree {
        int M_0 = (int) Math.pow(2, 20);
        int M_1 = (int) (Math.pow(2, 19) + Math.pow(2, 18));
        int M_2 = (int) (Math.pow(2, 18) + Math.pow(2, 17));
        int inserts = 0;
        int deletes = 0;
        int nodes = 0;
        Node root;

        public void AVLtree() {
                this.root = null;
        }

        public void AVL_insert(int a) {
                this.root = insert(a, this.root);
        }

        public void AVL_remove(int a) {
                this.root = remove(a, this.root);
        }

        public int getNodeCount() {
                return this.nodes;
        }

        public int getInserts() {
                return this.inserts;
        }

        public int getDeletes() {
                return this.deletes;
        }

        public void resetInsertCount() {
                this.inserts = 0;
        }

        private Node insert(int a, Node b) {
                int mod = a % 3;
```

```java
            if (b == null) {
                    this.inserts++;
                    b = new Node();
                    this.nodes++;

                    b.height = 0;
                    b.left = null;
                    b.right = null;

                    if (mod == 0) {
                            b.matrix = new int[M_0];
                    } else if (mod == 1) {
                            b.matrix = new int[M_1];
                    } else {
                            b.matrix = new int[M_2];
                    }
            } else if (a < b.val) {
                    b.left = insert(a, b.left);

                    if (height(b.left) - height(b.right) == 2) {
                            if (a < b.left.val) b =
                            ↪  singRightRotate(b);
                            else b = doubRightRotate(b);
                    }
            } else if (a > b.val) {
                    b.right = insert(a, b.right);
                    if (height(b.right) - height(b.left) == 2) {
                            if (a > b.right.val) b =
                            ↪  singLeftRotate(b);
                            else b = doubLeftRotate(b);
                    }
            }

            b.height = Math.max(height(b.left), height(b.right)) + 1;
            return b;
    }

    private Node singRightRotate(Node b) {

```

```java
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
```

```java
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
```

```java
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                          }
```

13

```
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 mostly correct. Insertion took more time than removal
both overall and on average. However the average time for the initial inser-
tion was greater than the average time for subsequent insertions. Apparently
this is due to the Java Virtual Machine using an algorithm that increases the
number of sweeps in garbage collection as the amount of the JVM's memory
being utilized increases. Thus the process of removing and inserting became
faster as the tree became larger. This is a testament to how effective Java's
Garbage Collection can be.

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 Java Virtual Machine, but for $C > M$ the JVM will be constantly paging and thus thrashing will certainly occur. As such my hypothesis is that the program's timing will increase considerably when $C > M$.

Java Implementation

```java
import com.sun.management.OperatingSystemMXBean;
import java.lang.management.ManagementFactory;

public class Prog4 {

    public static void main (String[] args) {

        OperatingSystemMXBean opsys =
          ManagementFactory.getPlatformMXBean(OperatingSystemMXBean.class);
        System.out.println("Phys Mem: " +
          opsys.getFreePhysicalMemorySize());
        System.out.println("Virtual Mem: " +
          opsys.getCommittedVirtualMemorySize());
        System.out.println("Free swap space: " +
          opsys.getFreeSwapSpaceSize());

        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};
        long freeBytes = opsys.getFreePhysicalMemorySize();

        for (int i = 0; i < 15; i++) {
```

```java
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 $1.5 * M$. However, the increase is not exponential. This suggests that the MacOSX's VMM replacement policy is effective at dealing with large numbers of swaps.

Programming Problem 5

Description of algorithm

Our 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. This resultant tree supplies our Huffman encoding.

Java Implementation

```java
import java.util.Scanner;
import java.util.Comparator;
import java.util.ArrayList;
import java.util.PriorityQueue;

class Node {
        char ch;
        int freq;
        Node left;
        Node right;
}

class HuffComparator implements Comparator<Node> {
        public int compare(Node a, Node b) {
                return a.freq - b.freq;
        }
}

public class Prog5 {

        public static void main(String[] args) {

                Scanner reader = new Scanner(System.in);
                ArrayList<Character> chars = new ArrayList<Character>();
                ArrayList<Integer> frequency = new ArrayList<Integer>();
                boolean check = true;

                while(check) {
                        System.out.println("Enter character (to exit enter
                        ↪  '0')");
                        char elem = reader.nextLine().charAt(0);

                        if (elem == '0') {
                                check = false;
                        } else {
                                System.out.println("What is the frequency
                                ↪  for the character?");
                                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