CSI2110
Assignment#2
Junhan Liu
7228243
****

I used eclipse to write the code, so all the source code (the *.java file) are in the folder called src. And all the .class file are in folder called bin, but the bin is empty since I delete all the .class file before submission.

The file called output is the output obtained by the main program.

There is also a .pdf file called README, just in case the .txt file won’t open in Windows system computer(this txt file is generated by Mac system). It happened before.

****

Part3
A) Priority queue Q implemented using a Sorted List.

The overall running time in term of big Oh of the algorithm is O(N*N).
-Initializing Q implemented using a Sorted List is O(1).
-Initializing “leafWhereLetterIs” is O(65536), which is also O(1).
-Assigning each position to null of the array “leafWhereLetterIs” is O(65536), which is O(1).
-Inside of the first loop, checking “if (frequencies[i]>0)” for n time for the worst case.
-Therefore, inside the if condition, the worst case, first two assignments take O(N) time.
-Except “heap.insert(node, node)” , it takes O(N*N) time, since each time the if condition is true, “heap.insert(node,node)” will be executed, each time the method executed, it takes O(N) time to insert. Therefore, if the “if” condition is true n times, the method “heap.insert(node,node)” will be called n times. The running time then will be O(N*N).
-creating a “HuffmanNode” is O(1)
-Assigning a position in the “leafWhereLetterIs” is O(1)
- “heap.insert” is O(N)
-Inside the while loop, “heap.removeMin()” is O(N),since the while condition will be checked n times.
-primitive operation is O(N)
-Initializing a Node is O(N)
-e1.getValue().parent is O(N)
-“heap.insert” is O(N*N)

B)Priority queue Q implemented using a Unsorted List.

The overall running time of the algorithm is now O(N*N).
-Since most of operations and assignment remains the same running time except all

“heap.insert” and “heap.removeMin” methods.

–Inside of the first loop, “heap.insert(node,node)” now change to O(N), because the insert methods now takes only O(1) time, and the worst case is that the if condition is true for n times, therefore, the methods “heap.insert(node,node)” will be executed n times, so the running time is O(N).

–Inside of the while loop, “heap.removeMin()” will now be O(N*N) time.

–So ignore all lower O() running time, the overall is O(N*N).

C)Priority queue Q implemented using a Heap.

The overall running time is O(N*logN).

–Most of operations remains the same running time except all “heap.insert” and “heap.removeMin” methods.

–Inside of the first loop, the if condition will be checked, the worst case, n times, therefore “heap.insert(node,node)” will be executed n times, an each time it takes O(logN) time, so the overall is O(N*logN).

–“heap.insert” is O(logN).

–Inside the while loop, the while loop condition will be checked size–1 time, thus n times.

–“heap.removeMin()” will be executed n times, each time takes O(logN), overall O(N*logN).

****
The algorithm code with foot note

```
private HuffmanNode BuildTree(int[] frequencies,char[] letters) {

 HeapPriorityQueue<HuffmanNode, HuffmanNode> heap =
   new HeapPriorityQueue<HuffmanNode,
HuffmanNode>(frequencies.length+1); // O(1)

 leafWhereLetterIs =new HuffmanNode[(int)'\uffff'+2]; // need 2^16+1
spaces // O(65536)
 for (int i=0; i< (int)'\uffff'+2; i++)
  leafWhereLetterIs[i]=null; // loop O(65536)

 for (int i=0; i<frequencies.length; i++) {
  if (frequencies[i]>0) { //check condition for n times


   HuffmanNode node= new HuffmanNode( (int)letters[i],
frequencies[i],null,null,null); //O(n)

   leafWhereLetterIs[(int)letters[i]]=node; // O(n)

   heap.insert(node,node); // 1) O(N*N) 2) O(N)  3) O(N*logN)

  }
 }

 HuffmanNode specialNode= new HuffmanNode( EndOfText,
0,null,null,null);//O(1)
 leafWhereLetterIs[EndOfText]=specialNode; //O(1)
 heap.insert(specialNode,specialNode); // 1) O(N) 2) O(1)  3)
O(logN)


 while(heap.size() > 1){ //check for size-1 times, n times

  Entry<HuffmanNode, HuffmanNode> e1 = heap.removeMin();// 1) O(N)
2) O(N*N) 3) O(logN * N )
  Entry<HuffmanNode, HuffmanNode> e2 = heap.removeMin();// 1) O(N)
2) O(N*N) 3) O(logN * N )

  int newFrequency = e1.getKey().getFrequency() +
e2.getKey().getFrequency(); //O(N)

  HuffmanNode newTree  = new HuffmanNode(0,newFrequency,null,
e1.getValue(), e2.getValue());//O(N)
  e1.getValue().parent = newTree;//O(N)
  e2.getValue().parent = newTree;//O(N)

  heap.insert(newTree, newTree);  //1) O(N*N) 2) O(N) 3) O(logN *
N )
 }
 Entry<HuffmanNode, HuffmanNode> e = heap.removeMin();// 1) O(1) 2)
O(N) 3) O(logN)

 return e.getValue();

 }
```