COMP 252 Assignment 6
Zenghao (Mike) Gao

Exercise 1

1.1

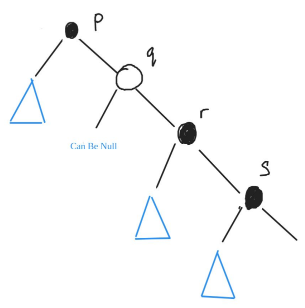
Insert can be done by simply creating a new iceberg *n* that consists of only the element *e* and then performing merge (e,I).
deletemin(I) can be done by simply removing the minimum, leaving 2 icebergs left and right, then you simply merge(left,right).

1.2

Suppose there are b black nodes and w white nodes, after the merge operation, w white nodes has to become black nodes. In the case of black nodes, some can remain black nodes (in the case when two subtrees are of equal size) or it can be changed into the white nodes. So the amortized cost is
b+w+potential <= b+w+(b-w) <= 2b

1.3

Suppose there is an iceberg with size = n. P, Q, R, S are consecutive nodes on the right roof



At P: Size of iceberg = n
At Q: Since P is black, size P's left child > size P's right child.
n $\underline{=\text{size(P.left)}}$ + size(P.right) + 1
n $\underline{<\text{size(P.right)}}$ + size(P.right) + 1
n < 2*size(P.right) + 1
size(P.right) <= n/2 - 0.5
size(P.right) < n/2 which means size(Q) < n/2
At R: Since Q is white, we have to consider the case where R size is maximal, which occurs when Q.left == null.

So Upper Bound on Q = Upper Bound on R < n/2
At S: Similar to at node Q, size(S) < n/(2^2)

<u>Observation</u>: The size of a node on the right roof is upper bounded by n/(2^b) where b is the number of black nodes above that node along the right roof.
We approach this problem from a bottom up approach. Consider the last node on the right roof. From the observation, the size of that node < n/(2^b), however, the size of such node is at least 1, so
1 <= size < n/(2^b)
1 < n/(2^b)
2^b < n
b < log2(n)

Finally, we recognizes the fact that the last node has to be black (since its right subtree is empty), so the number of black nodes on the right roof < 1+log2(n)

1.4
We represent the two Icebergs as I1, I2, each has the size of n1, n2 respectively. Suppose number of black nodes in the right roof of I1 = b1, and number of black nodes in the right roof of I2 = b2. From the conclusion of 1.3 we can conclude that b1 <= 1 + log2(n1), b2 <= 1 + log2(n2), and from 1.2 we can deduce that the amortized time of Merge(I1,I2) <= 2(b1+b2) = 4 + 2 log2(n1) + 2 log2(n2) <= 4 + 4 log2(n) = O(log2(n))

1.5
Since *insert* and *deletemin* are implemented through merge, they each cost O(log2(m)), where m is the size of the Iceberg after such an operation. Since m < n, t operations of *insert* and *deletemin* cost no more than O(t*log2(n))

1.6
Recursive divide and conquer.

```
makeIceberg(Array):
        If array.size == 1
                MakeNull(l)
                return l.insert(refToArray[0])
        Else
                L1 = makeIceberg(refToArray[0...n/2])
                L2 = makeIceberg(refToArray[n/2+1...n])
                return merge(l1,l2)
```

Note: refToArray takes pointer reference to the array elements.
By master theorem, Tn = *cost for merge* O(logn) + 2Tn/2 = O(n)

Exercise 2
2.1
E=1/3log2(3) + 2/3log2(3/2) = 0.918

2.2
nE = 0.918*n

2.3
You simply *do not compress*. Such implementation is trivial and fits the requirements.

2.4
Using huffman codes to encode all possible 2 bit sequences.
(0,0) -> 0 -> 4/9 (gets compressed by 1 bit)
(0,1) -> 10 -> 2/9 (unchanged length)
(1,0) -> 110 -> 2/9 (increase 1 bit)
(1,1) -> 111 -> 1/9 (increase 1 bit)

$$1-\left(\frac{4/9*1+(-2/9*1)+(-1/9*1)}{2}\right)=17/18<1.03*0.918$$

The result gives us an expected length of 0.9444 per bit, which makes the expected length 94.4% of n, which is within 3% of 0.918*n.

Exercise 3
Requirements: Exactness ✓ Elegance ✓ Readability ✓

<u>Unweighted-Longest-Path(G):</u>
Input: Unweighted DAG $G = (V, E)$
Output: Longest path distance in G

```
We first Topologically sort G
For each vertex $v \in V$ in linearlized order:
        do $dist(v) = max_{(u,v) \in E} \{dist(u) + 1\}$
Return $max_{v \in V} \{dist(v)\}$
```

<u>Topological-Sort(graph):</u>

```
n = graph.numberOfNodes()
v = [false,....false] # length n
ordering = [0,...,0] # length n
i = n - 1
For (int pos =0; pos < n; pos++)
        If v[pos] == false:
                i = dfs(i, pos, v, ordering, graph)
return ordering
```

<u>dfs(i, pos, v, ordering, graph):</u>

```
V[pos] = true
edges = graph.getEdgesFromNode(pos)
for e in edges
        if v[e.to] == false
                i = dfs(i, edge.to, v, ordering, graph)
ordering[i] = pos
Return i - 1
```

After finding the topological order (which takes O(|V|+|E|)), the algorithm processes all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is O(E). So the inner loop runs O(V+E) times. Therefore, the overall time complexity of this algorithm is linear.