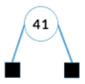
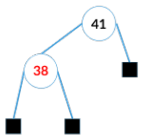
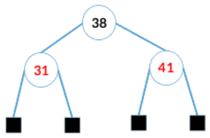
Valli Subasri
250697158

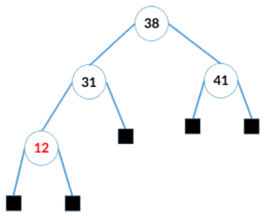
ASSIGNMENT 2

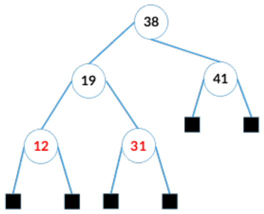
QUESTION 1

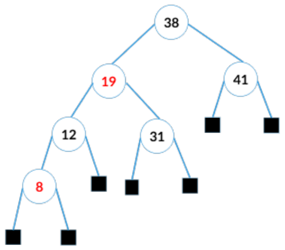




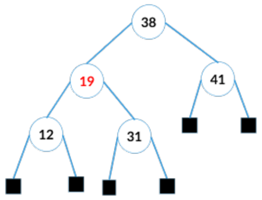


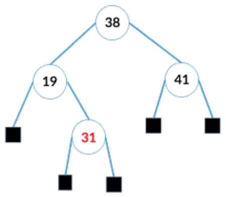


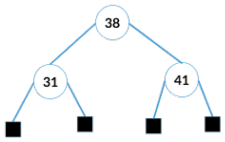


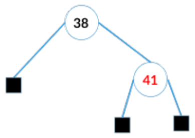


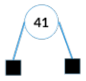
QUESTION 2

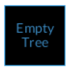












QUESTION 3

To compute the kth smallest element of a set of n distinct integers there are two steps:

(1) Construct a min-heap of the set of n distinct integers. This can be done via bottom-up construction.

    Time complexity: Constructing a min heap of n elements takes O(n) time.

(2) Call removeMin() k times. The remaining root of the heap is the kth smallest element.

    Time complexity: removeMin() takes logn time and it is being called k times thus the time complexity of this step is O(k log n)

Total time complexity: O(n+klogn)

QUESTION 4

To implement a min-max stack that supports the stack operations push(), pop(), min() and max() in O(1) time use two extra stacks, a minStack to keep track of the minimum values and a maxStack to keep track of the maximum values. To retrieve the current minimum, you return the top element from minStack. To retrieve the current maximum you return the top element from the maxStack. Every time you push(), you have to check if the pushed element is a new minimum or maximum. If it is a new minimum then it is pushed to the min stack too, if it is a new maximum then it is pushed to the maxStack too. When you perform a pop operation, you check if the popped element is the same as the current minimum or maximum. If the popped element is the same as the current minimum then pop it off the minStack, and if it is the same as the current maximum then pop it off the maxStack.

```
public class MinMaxStack {
   s ← stack
   minStack ← stack
   maxStack ← stack

public void push(int k){

      if minStack.isEmpty() or k <= minStack.top() then
         minStack.push(k);

      if maxStack.isEmpty() or k >= maxStack.peek() then
         maxStack.push(k);

       s.push(k);
   }
public void pop(){
      if s.top() = min() then
          minStack.pop();
      }

      if s.top() = max() then
         maxStack.pop();

      s.pop();
   }

   public int min(){
      return minStack.top()
}

public int max() {
      return maxStack.top()
}
```

QUESTION 5

Floating-Point Summation Algorithm: Given a set S of n floating-point numbers, S = {x1, x2,...,xn}, let x_i and x_j be the two numbers in S with smallest magnitudes. Sum x_i and x_j and return the result to S. Repeat this process until only one number remains in S, which is the sum.

Time complexity: O(n log n)

This algorithm can be implemented by implementing a min-heap structure containing the set S of n floating point-numbers. From this heap the minimum floating point number can be removed via removeMin() and stored in a variable xi; this takes logn time. removeMin() extracts the minimum element from the heap (ie. the root node) and restores the heap-order property via down heap bubbling. The new root of the heap is the next smallest element in the set S. By calling removeMin() again the next smallest floating point number can be removed from the heap and stored in a variable xj. Xi and xj can then be summed and added back to the heap via insert(); this takes logn time. insert() adds this new summed value to the heap and restores heap-order property by performing up-heap bubbling. Now this continues until all elements in the set have been summed.

With each summation there is one fewer floating-point number in the set; there are n-1 summations that occur and each of those summations requires to removeMin() calls:

$$2(n-1)\log n$$

There are also n-1 insertions back into the heap that occur at logn time:

$$(n-1)\log n$$

In total:

$$T(n) = 2(n-1)\log n + (n-1)\log n$$
$$T(n) = 2n\log n - 2\log n + n\log n - \log n$$
$$T(n) = \log n(2n - 2 + n - 1)$$
$$T(n) = (n-3)\log n$$
$$T(n) = O(n\log n)$$

<u>**QUESTION 6**</u>

Insert(x) – We insert the key into the tree, like any other red black tree. However, we also add 2 other attributes for this node: the number of nodes in the right subtree and the number of nodes in the left subtree. Whenever we add a node it gets updated on how many nodes are in its right and left subtrees.

Time complexity:

= Binary tree insertion + Colour changes + Rotations + Updating right/left subtree attribute for each node
= $O(\log_2 n) + O(\log_2 n) + O(1) + O(\log_2 n)$
= $O(\log_2 n)$

Therefore, the time complexity is $O(\log_2 n)$.

Delete(x) – This occurs like any other red black tree deletion, except we also recursively update the parent of the deleted node's parent until the root. We update the amount of nodes in the subtrees, decrementing the value accordingly.

Time complexity:

= Binary tree insertion + Colour changes + Rotations + Updating right/left subtree attribute for each parent node
= $O(\log_2 n) + O(\log_2 n) + O(1) + O(\log_2 n)$
= $O(\log_2 n)$

Therefore, the time complexity is $O(\log_2 n)$.

Find_Smallest(k) – This method uses a breadth first search. Starting from the root, compare k to the attribute representing the number of elements in the left subtree. If k is *greater than* the number of nodes in left subtree ($k > n(T_L)$), then we can ignore the left subtree, and because it's smaller than the number we are looking for, we proceed to the right subtree. If k is *less than* the number of nodes in left subtree ($k < n(T_L)$), we proceed to the left subtree, and repeat the procedure. When k is *equal* to the number of nodes in a given subtree, then we've found the kth smallest element.

Time complexity: $O(\log_2 n)$

QUESTION 7

Let r represent rank of a node, n represent number of Make-Set operations.

Inductive Hypothesis:

$r \leq \log n$, which can be written as $2^r \leq n$

Base Case:

When n = 1, one node exists and the node has a rank of 0

$0 \leq \log(1) \rightarrow 2^0 = 1$

Inductive Step:

A node x has rank r+1 when it is previously has rank r and is joined (via union) to another tree rooted by a node with at least rank r. Then x becomes the root of the union of the two trees. Each tree by inductive hypothesis is at least of size 2^r, and so now x is the root of a tree that is of size of at least $2^r + 2^r = 2^{r+1}$. Now the number of nodes in the forest is n and we have at least 2^r nodes in every tree with rank r.

$2^r + 2^r \leq n + n$

$2^r(1 + 1) \leq 2n$

$2(2^r) \leq 2n$

$2^r \leq n$

$r \leq \log n$

Therefore rank of a node is bounded by log(n).

QUESTION 8

Because rank[x] needs $\log n$ bits to be stored and the node also needs $\log n$ bits to be stored. Therefore, the amount of bits necessary to store rank[x] is $\log(\log n)$. Since one byte has 8 bits, a byte can store the rank value for any node as long as the rank is less than or equal to 256.

$\log(\log n) \le 8$
$\log r \le 2^8$
$r \le 256$

QUESTION 9

The Huffman algorithm considers the two least frequent elements recursively as the sibling leaves of maximum depth in code tree. The Fibonacci sequence (as frequencies list) is defined to satisfy $F(n) + F(n+1) = F(n+2)$. As a consequence, the resulting tree will be the most unbalanced one, being a full binary tree, in which every internal node has two children. Since we can now restrict our attention to full binary trees, we can say that if F is the amount of Fibonacci numbers from which the numbers are drawn, then the tree T has exactly $|F|$ leaves and exactly $|F| - 1$ internal nodes. In other words, given a tree T of n internal nodes, there are n + 1 external nodes.

Proof:

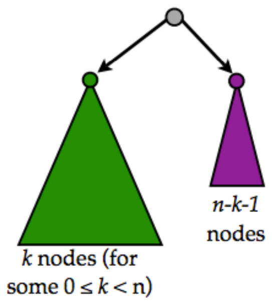
$X(n)$:= number of external nodes in Huffman tree of n Fibonacci numbers as internal nodes

Inductive Hypothesis: A Huffman tree of n internal nodes has n + 1 external nodes.

Base Case:

$X(0) = 1$ which is equal to the $n + 1 = 0 + 1 = 1$; tree consists of only the root.

Inductive Step:



$X(n) = X(k) + X(n-k-1))$
$X(n) = k + 1 + n-k-1 + 1$
$X(n) = n + 1$

Thus a Huffman tree T for a set of characters having frequencies equal to the first n nonzero Fibonacci numbers has n leaves and n-1 internal nodes; thus we have proven that every internal node in T has an external-node child.