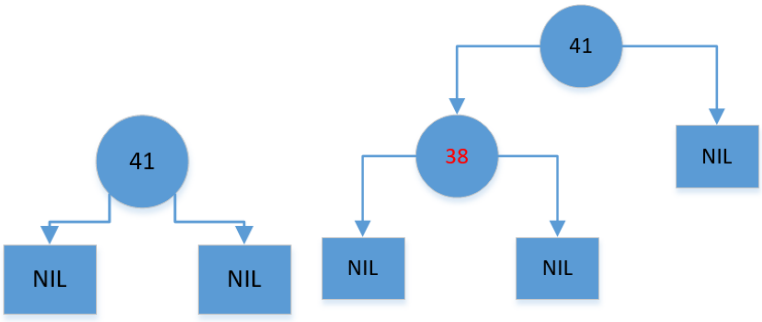
3340 Assignment 2

Justin Grant

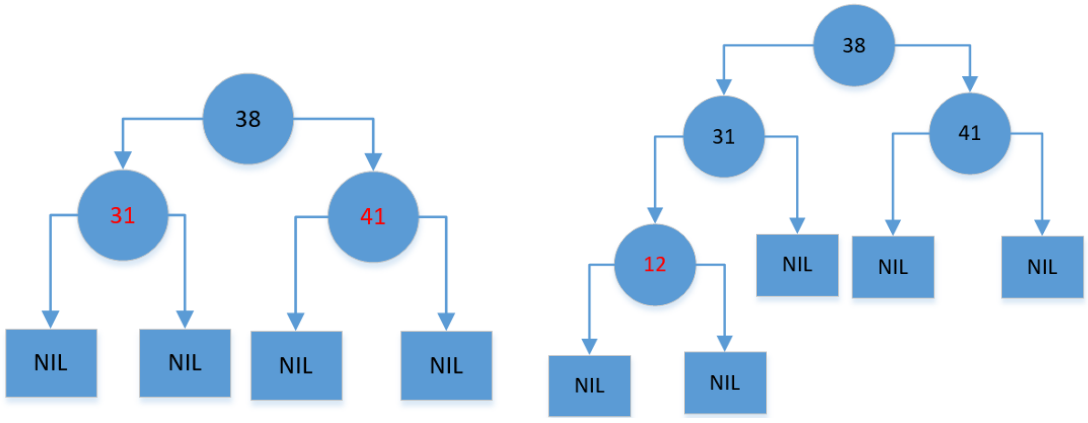
Student#: 250787131

1. Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.

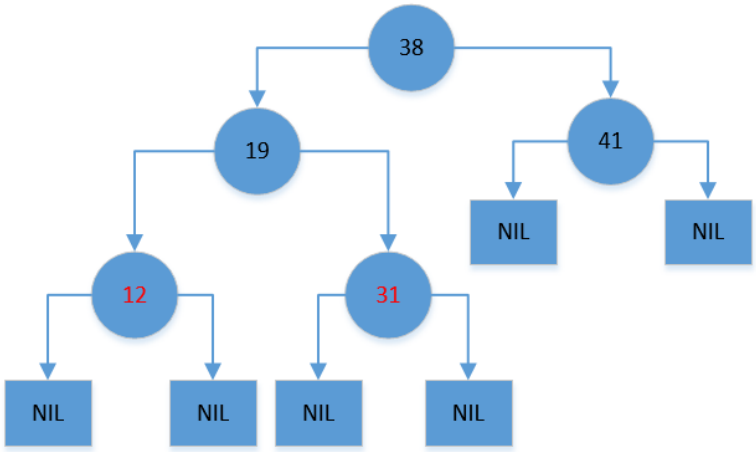
TREE 1: Inserting 41 | TREE 2: Inserting 38



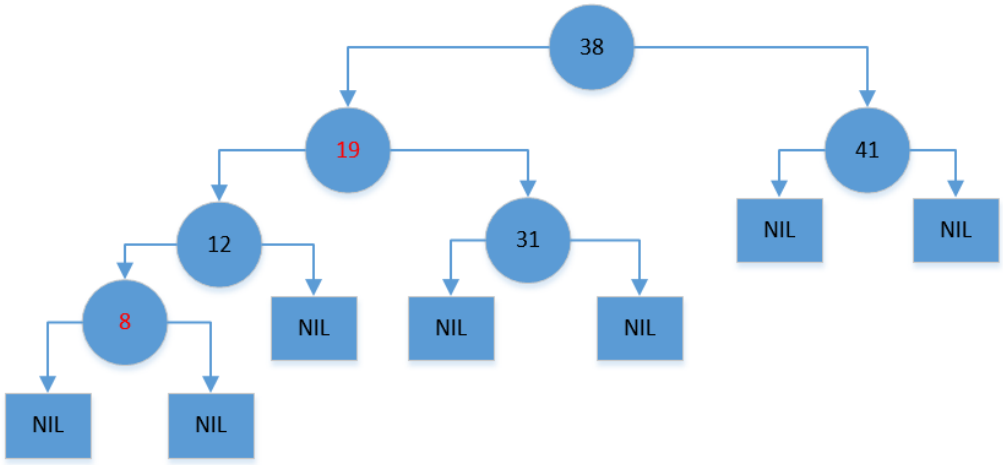
TREE 3: Inserting 31 | TREE 4: Inserting 12



TREE 5: Inserting 19

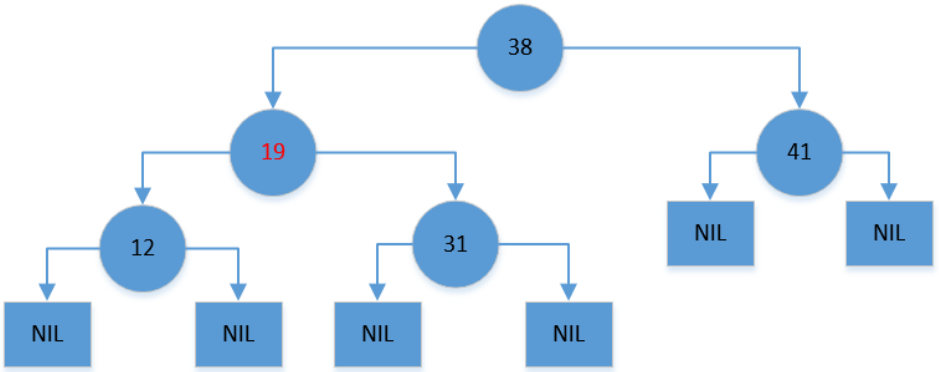


TREE 6: Inserting 8

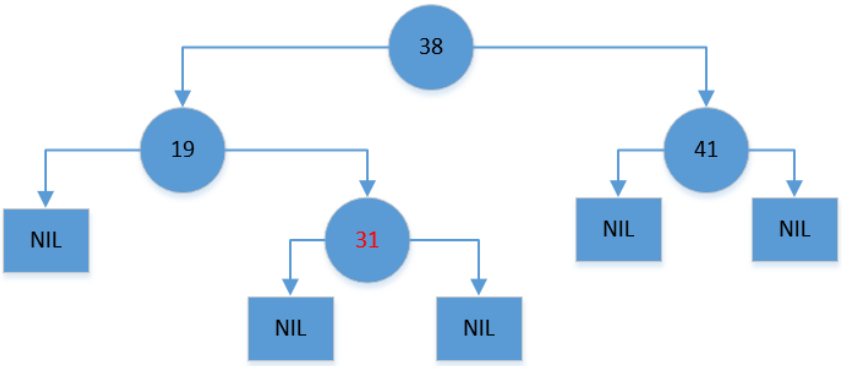


2. Show the red-black trees that result after successively deleting the keys 8, 12, 19, 31, 38, 41 from the final red-black tree of question 1.

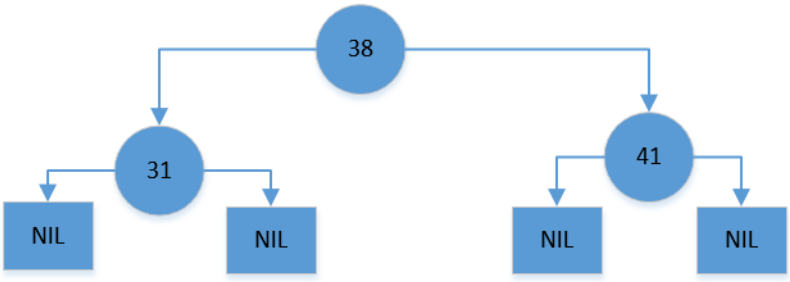
TREE 1: Deleting 8



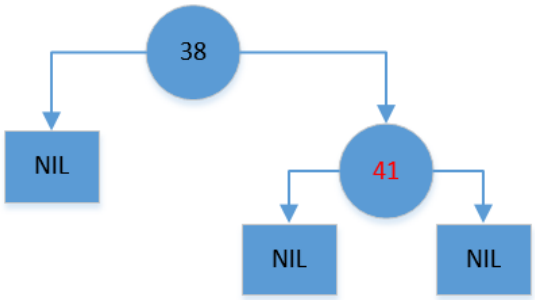
TREE 2: Deleting 12

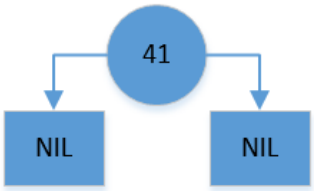


TREE 3: DELETING 19



TREE 4: Deleting 31 | TREE 5: Deleting 38





TREE 6: Deleting 41

3. In the text book C-5.8, pp. 184.

Develop an algorithm the computes the kth smallest element of a set of n distinct integers in O(n+klogn) time.

We can iterate through the set of integers, and put them into a heap using a bottom up heap construction. Each element representing a node, we can connect them the same way as a linked list or tree. We would iterate through the set and sort in this fashion. We take I the index of the number in the set, and compare it to 2i and 2i+1, which will be its children. We compare the node at "I" to it's children, and the children to each other. If the node at "I" is larger we swap it's position with its smaller children. This one comparison takes constant time as we make one check between the children, one check between the parent and smaller child, and one swap. We do this n times as we iterate through the array and the array should be heap sorted. We could go through the array and connect the array elements physically as a tree using linked list nodes, and it would take n time as well. Thus so far we have done O(n) time. now to find the kth's smallest element we could find the smallest element in the heap, which simply involves traversing the tree which is approximately log2n nodes in height, and thus takes log2n comparisons, in the standard way we traverse binary trees. Next we remove the smallest element and start again. We do this k times until we reach the kth element in the tree. Thus this portion of the algorithm takes klog2n time as we traverse the tree k times and each traversal takes log2n time. In total the algorithm now takes O(n + klog2n) time.

4. In the text book C-5.10, pp. 184.

Define a min-max stack to be a data structure that supports the stack operations of push() and pop() for objects that come from a total order, as well as operations min() and max(), which return, but do not delete the minimum or maximum element in the min-max stack, respectively. Describe an implementation for a min-max stack that can perform each of these operations in O(1) time.

We can implement the data structure as a set of three stacks. One stack, the main stack, manages push and pop commands. One stack manages the max, and one stack manages the min. We implement these stacks using linked lists. When we push a number to the min-max stack data structure we push it to the main stack first. This is done by having a reference to the first element in the linked list reference the number we are pushing, and having that number reference the previous first number. It is done in O(1) time as we only work with the first number in the list.

We then compare the number we added to the minimum stack and maximum stack, which are more priority queues rather than stack. These stacks are implemented as linked lists with a reference to the head and tail of the list. We compare the number to push to the head of each of the max stack and the min stack. If the value is greater than the value in the max stack we push it to the head of the linked list, and if not we push it to the tail in the same way we added the number to the main stack. We do the same to the min stack if the value is instead lower than the value at the head. We are doing a constant number of operations here as we are only working with 6 references at most for one operation.

For pop it is very simple to do in O(1) time as well. We simply get the value of the head of the main stack, and set the reference to the top of the stack to the next element. Then we compare value we got to the max values and min values. If it is equal to the max value we remove the top element on the max stack, but if it isn't we remove the tail value from the max stack. If it is equal to the min value we remove the top element on the min stack, but if it isn't we remove the tail value from the min stack. This ensure that the stacks all have the same elements and maintains the max and min properties of those stacks. As we only do 2 comparison and 6 pointer switches we operate in O(1) time.

For the final operations getMax() and getMin() we simply look at the "head" references to our max and min stack and return the value pointed to by these references. This is clearly O(1) time.

5. In the text book A-5.5, pp. 185.

…Describe how to implement this floating-point summation algorithm in O (nlogn) time.

For this algorithm we would use a sorting algorithm such as merge sort the sort the set of n elements. As mergesort completes in O(nlogn) time the algorithm is at most using O(nlogn) time so far. Next we do a binary search for 0 in the sorted set which takes O(logn) time. It will likely not be a match, but we can compare up to 5 elements nearest that position to find the two elements with the smallest magnitude. With that set we find the two elements of smallest magnitude, so we have at most approximately 10 comparisons there. Once we've found the two smallest we sum and remove the elements from the set, which we can do in O(1) time if we're using a linked list. We must repeat this

process n/2 times because we remove 2 elements from the set each time we search. Thus in total the time it takes is O(nlogn + ~10*n/2*(logn)) time which is of O(nlogn) time.

6. Design an efficient data structure using (modified) red-black trees that supports the

following operations:

Insert(x): insert the key x into the data structure if it is not already there.

Delete(x): delete the key x from the data structure if it is there.

Find Smallest(k): find the kth smallest key in the data structure.

What are the time complexities of these operations?

The overall structure of the red-black tree would be typical for red black trees. It would be a series of nodes which contain pointers to their children, the data they store, and a flag for whether that specific node is red or black. We could potentially add a flag to represent an “extra black” to the following operations a little simpler. The very bottom leaves of the tree would be represented by a special node representing Null, however these nodes are still “black”. We will also have each child node point to their parent as well.

The main difference with the red-black tree is that we would keep track of the number of nodes in each subtree. With insertion we would insert a node, which has a count of 0, then continuously follow up the parents to the root and insertion the number of nodes in the subtree by one.

This becomes tricky when we adjust the tree. When we swap nodes, we swap their node count as well. When we do a rotation we recalculate the number of nodes in a subtree but looking at the child’s node counts. I won’t list it in the description of the each method but that is what it essentially entails. Any node who has its children changed must recalculate the subtree count it has, then update its ancestors count.

Insertion

When it comes to insertion we must follow a specific method. To start the insertion we will start at the root of the tree. If the number we wish to insert is smaller we go to the left child, and if it is greater we go to the right. We continue this until the node we plan to move to is a null, or find the node we are trying to insert as we only insert if it is not in the tree already. We replace that null with the node we wish to insert, color this node red, and link it to the parent of the null node. We must check to see if this an appropriate action and does not violate red-black tree rules, so we check if the parent is red. If the parent is red we must reference the parent’s sibling node, by using the backwards references to go up the tree and reach the “uncle” node of the node we inserted. There are 3 cases that may arise.

If the parent and uncle of the node we are inserting are both red, we change them both to black, and switch the grandparent to red. We then do the same comparison for the grandparent and its sibling and continue until we reach the root.

Case two is when the node we are inserting is a “right” child, the “uncle” node is black and a right child or the node we are inserting is a “left” child, and the “uncle” node is black and a left child. In this case we do a left or right rotation at the parent respectively. The parent will become the left child of the node we are inserting and it becomes the left child of the grandparent node. Or the other way around if the node was a left child. After this we consider that case three may be affecting the graph.

Case three is similar to case two except the node we are inserting is the opposite direction of the black “uncle” node, ie the node we insert is a left child but the uncle node is a right child. Similar to case two we do a rotation, but at the grandparent of the node we are inserting instead. The parent of the node we are inserting takes the grandparents position, the node we insert takes the parents position, and the uncle moves down one rank.

Deletion

With deletion we look for the element to delete in the exact same way that we do with insertion. Once we delete we must consider how it affects the graph.

If the node we want to delete is red we can simply delete it in a very normal fashion, and it does not affect the graph. If the node we want to delete is black, but it has a child node that is not null and is instead red, we can color the child black and then delete the node.

If the node we want to delete is black and has no children we delete it, but then create an “extra black” node at that position instead. We consider 4 cases after this, and loop through these 4 cases until we’ve fixed the tree.

Case 1: the sibling of the extra black node is red. This means we can color the node to black, and do a left rotation if the extra node is a left child or a right rotation if it is a right child. The rotation is done at the parent. The parent also switches color to red as well. We keep the extra black to deal with in another case.

Case 2: The sibling of the extra black is black, and has two black children. We can color the sibling red instead, and then make the parent of the node we deleted have an “extra black” flag, and start considering these four cases from its perspective.

Case 3: the sibling to the extra black is black and has a red left child and a black right child. In this case you make the left child the new sibling of the extra black node, making the previous one the child of the previous left child. The new sibling becomes black and its child becomes red.

Case 4: The sibling of the extra black is black and has a red child. We do a left or right rotations depends on if the sibling if right or left child respectively. The sibling becomes the parent and swaps colors with the old parent. The child of the sibling is colored black, and the issue is resolved.

Find Smallest (k)

For this algorithm we know how many nodes there are in each subtree. So we can easily find the kth node. We set L equal to k at the start. Starting from the root if the number of nodes in the left subtree is

greater or equal to L than we check the left subtree and subtract 1 from L. If we find that the number of nodes in the left subtree is greater than kLwe move to the right, and subtract the number of nodes in the left subtree from L and then subtract 1 from L. If L is 0 the node we are looking at is the node we found. Also if the number of nodes in the left path is equal to the value of k-1 then we have also found the node we are looking for.

These three operations can be done in $O(log_2 n)$ time.

For insertion. We traverse the tree to the position we wish to search. As it is a red-black tree we know that the length of the tree is less than 2log2n in the worst case. So in the worst case we do 2log2n comparisons, which takes O(log2n) time so far. Next we have to make adjustments if there are conflicts with the red-black tree. If this is the case we must do a combination of changing colors of nodes, and rotations. Both rotations and color changes can be done in constant time. With a rotation involves changing the references of a constant number of nodes and this number does not increase with scale, and thus takes a constant time to rotate. We only must do at most 3 color changes as well. The number of rotations we must do is not exactly constant, but it is within log2n time. The 3 cases described before must be done to ensure the properties of the tree are kept, however after using solution 2 described above once, we then do solution 3. If we do solution 3 we have balanced the tree. Solution 1 can lead back into doing solution 1 once again, however we will do this at most Log2n times, as each time we do it the double red node issue that arised is pushed two ranks upward in the tree until it reaches the root. So it takes at most log2n solution 1's before we've balanced the tree or must move to solution 2 or 3. So in total the time it takes is (2log2n + clog2n) at most which his O(log2n) time. The only other thing to account for is the updating of number of nodes in subtree count, which is trivial and takes O(2logn) time maximum as it involves traversing the tree upwards and increase the count by 1 each time.

With Deletion it is very similar to Insertion. Getting to the node to delete takes O(log2n) time. Left and right rotations take constant time. Similarly we only need at most 1 rotation for each "case" (the issue and solution we take to fix the red and black inconsistency in the tree), and the number of color switches are at max three. A case 1 problem will either lead to a case 2 problem that finishes a case 3 problem that leads to a case 4 problem that finishes, or just a case 4 problem which if done is always the last solution. A case 2 problem can lead to a case1 problem, case 3 problem or itself. For all of the cases except a case 2 solution they will eventually lead to finishing the algorithm. Thus taking a constant number of executions until reaching a case 4 problem. With Case 2, we can only loop this problem so many times, because it only persists while we have an "extra black" node, and the result of using the case 2 solution is the "extra black" node raises a level. Eventually the extra black node reaches the route, which means the tree is now fixed, thus we have to do case 2 solution 2log2n times at max, as the tree is in worst case 2log2n nodes in size. log2n to move the node to delete, and c*log2n to adjust the tree in the worst case scenarios. Thus in total it takes approximately O(log2n) time to delete an element.

The Find Smallest is also O(log2n) time. With this algorithm we iterate to potentially the bottom of the tree. Before we descend we make 3 comparisons to check the number of nodes in the left and right subtree. Then we go down one level and repeat the process until we find the node we were looking for which could be at the bottom of the tree. With the high being at most 2log2n, then the number of times we iterate this algorithm is only 2log2n times. Thus c * 2log2n is how long it will take in the worst case. Which means the algorithm in total is O(log2n) time.

7. Prove that every node has rank at most ⌊log2 n⌋

Induction Base: A node which is part of a tree containing 1 node (itself) has rank 0

Induction Hypothesis: A node which is part of a tree containing n nodes has rank <= ⌊log2 n⌋

So we assume that 2^r <= n

So we have to show that

rank <= ⌊log2 n+1⌋

2^r <= n+1

2^r – 1 <= n

and using the induction hypothesis we know that

2^r -1 <= 2^r <= n

So the inequality holds for n+1

Therefore, we've proven that every node has rank at ⌊log2 n⌋

8. In light of question 8, how many bits are necessary to store rank[x] for each node x? When implementing an Union-Find data structure with any programming language, is one byte enough to store rank value for each node?

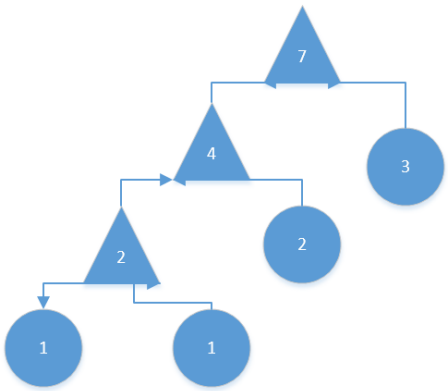
In Question 7 we proved that every node has at most rank $\lfloor log_2 n \rfloor$. This means that we only need to store numbers from Zero to $\lfloor log_2 n \rfloor$. We are able to store numbers from zero to 2^d, with d binary digits, or we an store numbers from zero to d with $log_2 d$ number of binary digits. Thus we only need $log_2 log_2 d$ number of binary digits to store numbers from Zero to $log_2 n$, which will let us store the rank of a node. One byte is enough to store the rank of a tree containing 2^{2^8} number of nodes, which in general will be enough to store the rank of each node, unless the tree was ridiculously and impossibly large. The rank of a node would most likely not exceed the number 256, unless the tree has more than 2^{2^8} nodes.

9. In the text book C-10.7, pp. 299

C-10.7

Suppose T is a Huffman tree for a set of characters having frequencies equal to the first n nonzero Fibonacci numbers {1,1,2,3,5,8,13,21,34…}, where $f_0 = 1$, $f_1 = 1$, and $f_i = f_{i-1} + f_{i-2}$. Prove that every internal node in T has an external-node child.

As the frequencies are derived from the Fibonacci sequence the first two frequencies are both 1. They would be at the very bottom of the Huffman tree and have the longest Huffman code. The subsequent members in the tree would continue to grow, but only in one direction.



In short the Huffman Tree will continue growing in that direction. As the tree will continue in this pattern it is clear to see that the each of the internal nodes do have at least one external leaf child. This is because the sum of all the previous terms will always be equal to or smaller than the new term that the Fibonacci sequence generates.

Essentially our inductive hypothesis is.

$$\sum_{i=0}^{n} F(i) \geq F(n+1)$$

Which we can expand to

$$\sum_{i=0}^{n} F(i) \geq F(i) + F(i-1)$$

And using the recursive definition of the Fibonacci sequence we get

$$F(0) + \cdots F(i-1) + F(i) \geq F(i) + F(i-1)$$

Therefore it is clear that the sum of previous terms will be greater than the next term.

10. View the attached .java files.