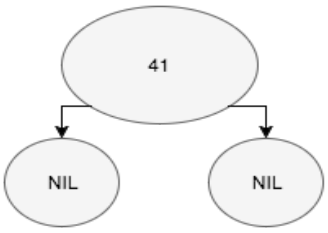
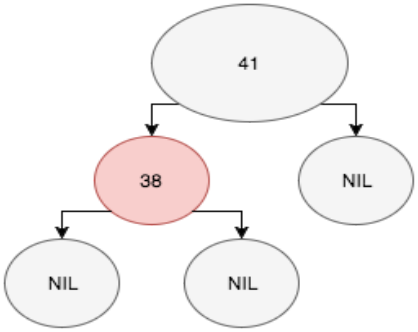
CS 3340 Assignment 2
Linsheng Ding
250757782

1)

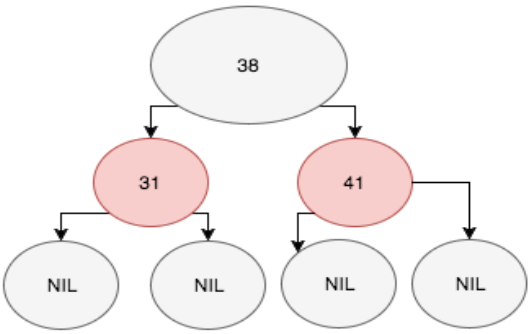
1) Insert 41



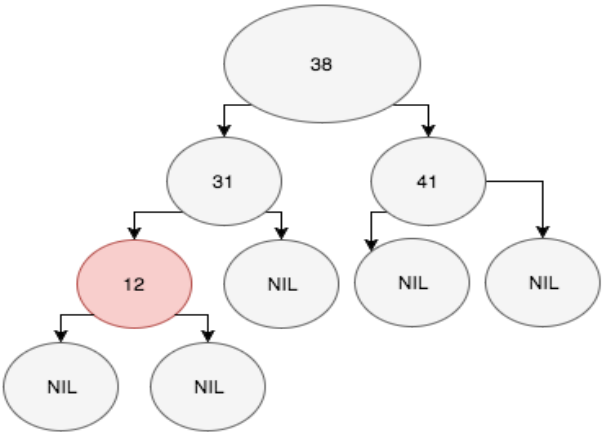
2) Insert 38



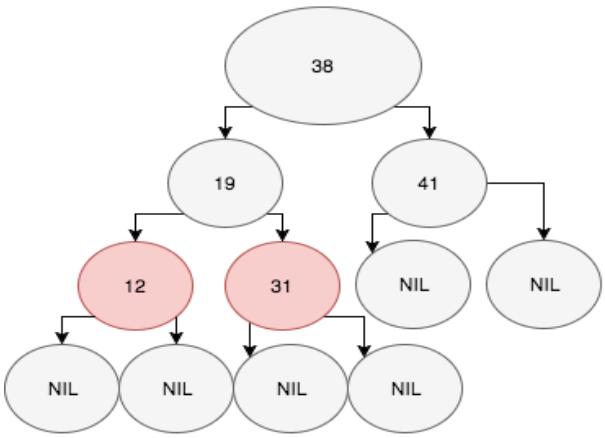
3) Insert 31



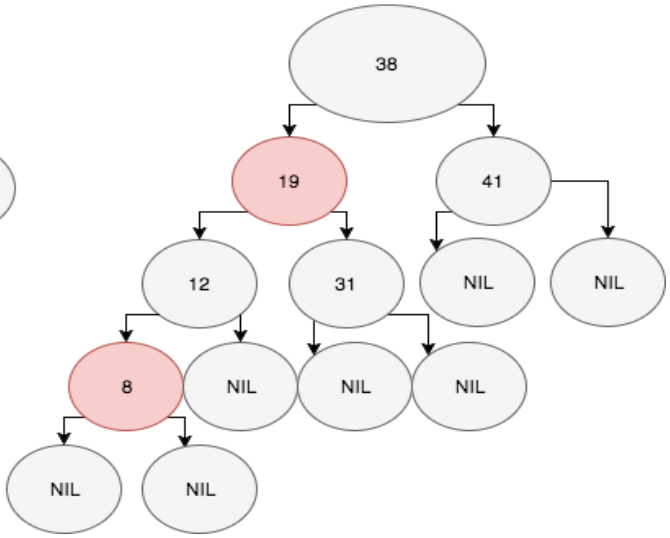
4) Insert 12



5) Insert 19

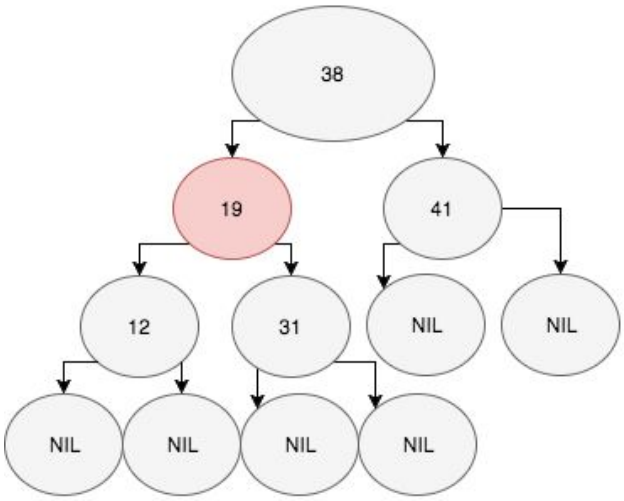


6) Insert 8

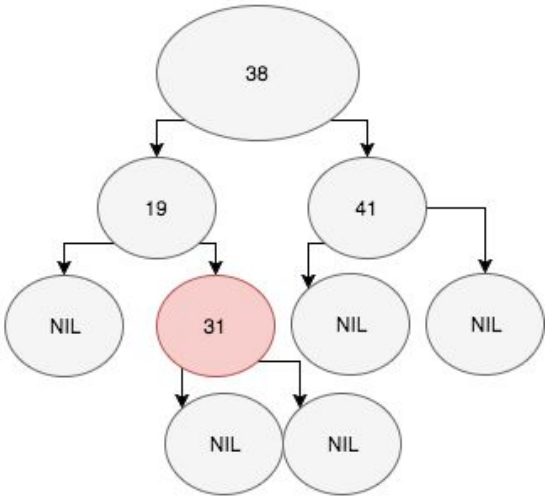


2)

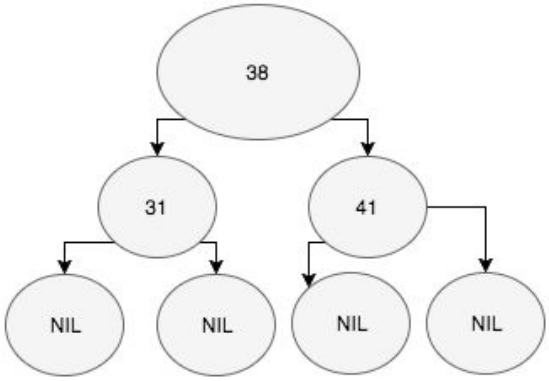
1) Delete 8



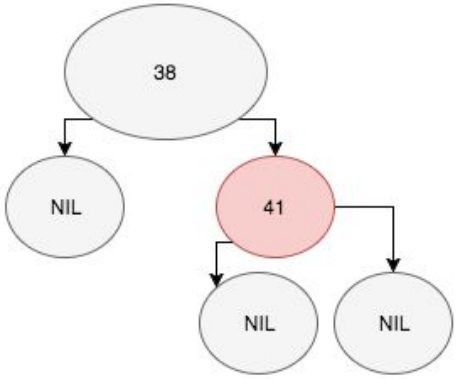
2) Delete 12



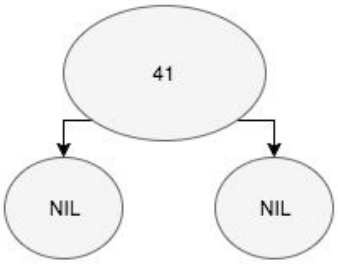
3) Delete 19



4) Delete 31



5) Delete 38



6) Delete 41

3)

Question: Develop an algorithm that computes the kth smallest element of a set of n distinct integers in O(n + k log n) time.

Answer:
Store each of the n integers into a min heap where if the heap is empty, the element put in is the root, else insert it at the bottom level of the tree and compare with parent and swap it around if child is less than parent and keep repeating this until every child is greater than parent which is O(n) time.

Next extract the root or smallest integer in the heap k times where we also move the last element at the deepest level in the heap to the root and keep swapping around until the root is the smallest integer for k times, generating a time complexity of O(klogn).

Overall the two above operations yield a time complexity of O(n + klogn).

4)
Question:
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.

Answer:
INITIALIZATION:
Create a stack data structure constructor with attributes min and max and set the first element pushed onto the stack as both the min and max element. The stack will also include a hash table for keeping count in case we already pushed an element that was a minimum and then push another similar element, a stack for storing minimums and one for storing maximums

FUNCTIONS:
push() : Whatever gets pushed into the stack is compared to both min and max, if it is less than min, the new element will be the new min and pushed onto the min stack as well or if it is greater it will the new max and pushed onto the max stack and then set the new min or max attribute to our newly pushed element. If it is min or max, check if it is already in the hashtable, if it is, increment the count associated with the element or create a new one in the hashtable if it not there.

pop(): If an element is being popped, check if its either the min or the max and removes if its the last element of that kind from the hashtable, pop the min or max stack depending on if its min or max and then set the new min or max attributes from whatever returns from a peek() of the min or max stacks.

Min(): will return the min attribute set in the constructor.

Max(): will return the max attribute set in the constructor.

TIME COMPLEXITY:
All of these above operations are O(1), max() and min() just return an already stored value and push() and pop() all use other operations that are O(1)

5)
Question:
The problem of accurately summing a set S of n floating-point numbers, S = {x1, x2, . . . , xn}, on a real-world computer is more challenging than might first appear. For example, using the standard accumulating-sum algorithm to compute the harmonic numbers,
Hn = Sigma(from i =1 to n) 1/n
in floating-point produces a sequence that converges to a constant instead of growing to infinity as a function close to ln n. This phenomenon is due to the fact that floating-point addition can suffer from round-off errors, especially if the two numbers being added differ greatly in magnitude. Fortunately, there are several useful heuristic summation algorithms for dealing with this issue, with one of them being the following.

Let xi and xj be two numbers in S with smallest magnitudes. Sum xi and xj and return the result to S. Repeat this process until only one number remains in S, which is the sum. This algorithm will often produce a more accurate sum than the straightforward summation algorithm, because it tends to minimize the round-off errors of each partial sum.

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

Answer:

- Initialize a min heap and store all the values in S as floating-pointing numbers into the heap which is O(n).

- Initialize a floating-point variable called **sum**, and another two floating-point variables called **x_1** and **x_2** which should take O(1) time.

- Use extractMin() twice from the min heap which is O(logn) which extracts the root from the min heap store them into x_1 and x_2 respectively. Then add x_1 and x_2 together and add them together and store them as sum.

- Insert sum back into the min heap which should be O(log n).

- Repeat the above enough times so that you only have one node in the heap which should be n times as there are n nodes in the tree.

- Overall the time complexity would be O(nlogn) as each extractMin() and insert() operation is O(logn) and you would do this a total of n times to go through all of S.

6)

For this question, we will create an extra private function called **checkIfExists()**. The **checkIfExists(key)** will be called by both insert() and delete() and will using a Binary Search Tree traversal method (compares if key is less or greater than current node and moves left if less or right if greater) to find a certain key and returns the key if found else it returns null if none is found which O(log n) time complexity. The overall space complexity for this data structure is O(n) because it uses a tree and stores n elements.

Insert(): First calls **checkIfExists(key)**, if a key is returned from **checkIfExists**, then show an exception or an error message stating that key is already there. Else traverse the tree in Binary Search Tree fashion and then store key in new position. Also performs Red-Black Tree rotations to meet Red Black Tree conditions.

Delete(): First calls **checkIfExists(key)**, if a key is returned from **checkIfExists**, we then remove the key from the tree and perform rotations if needed to return to Red-Black Tree conditions. Else if the key isn't there, just call an exception or print an error message.

find_Smallest(k): Use inorder traversal like below:

```
inorder(node root) {
        inorder(root.left)
        print(root)
        inorder(root.right)
}
```

Where you would move in order from smallest to largest due to it being a Binary Search Tree from smallest to largest.
initialize a counter variable that starts from 0, everytime you move one node in the inorder operation, increment count, if count is equal to key, return the node you're at.

TIME COMPLEXITY:
Insert(): O(logn) as both checkIfExists and a Red-Black Tree's insert are O(logn) operations as you will be traversing in a Binary Search Tree and handling rotations in a Red-Black Tree manner.
Delete(): O(logn) as both checkIfExists and a Red-Black Tree's delete are O(logn) operations as you will be traversing in a Binary Search Tree and handling rotations in a Red-Black Tree manner.
find_Smallest(k): Inorder traversal is O(n) as you could end up traversing the whole tree

7)

We can prove every node has rank at most ⌊log2 n⌋ using a proof by induction.

Proof by Induction:

We can assume that based on the above claim and lecture notes that if a node has rank k, then its height is bounded by its rank k and that it is the root of a subtree of size at least 2^k.

Base case:

If a node has rank 0 and is the root of a subtree then it includes at least itself which is at least 1 and 2^0 is 1 which thus satisfies the base case.

Inductive hypothesis: Node k+1 will be a root that also have at least 2^(k+1) size in its subtree

Inductive case:

We use union between two nodes of rank k where two subtrees have been joined together and a node from the two original roots become the root of the union of the two trees which is of rank k+1 as stated in our lecture notes. This new root has two trees in it of size 2^k which is 2^k + 2^k or 2^(k+1) which satisfied our inductive hypothesis.

As well then, we can see that using n nodes in all the trees and at least 2^k nodes in each tree with rank k, therefore we can get:

$n >= 2^k$

Which when we turn into a logarithm becomes $\log_2(n) >= k$ or:
Rank $k <= \log_2(n)$
Which proves our claim.

8)

Each node would need to have a rank of at most log2(n) and thus means that if we have a node that is a root of size 1000, would have a rank of at most 3.

I think 5 bits is necessary to store rank[x] for each node x as given 5 bits we can count up to 2^63 in size using below steps:

- Sum up all the possibilities from 1 to 5 bits: 2^5 = 64
- Turn logarithm into powers: log2(n) = 64 ⇒ n = 2^64
- 2^64 = 1.8446744e+19 which can accommodate for a lot of nodes and size to store for rank value of a tree.

I also believe one byte is enough as one byte is 8 bits and when we turn that into a size we get using the below steps:

- Sum up possibilities from 1 to 8 bits: 2^8 = 256
- Turn logarithm into powers: log2(n) = 256 ⇒ n = 2^256

- 2^256 = 1.1579209e+77 which is enough to accommodate for size and to store for rank value of a tree

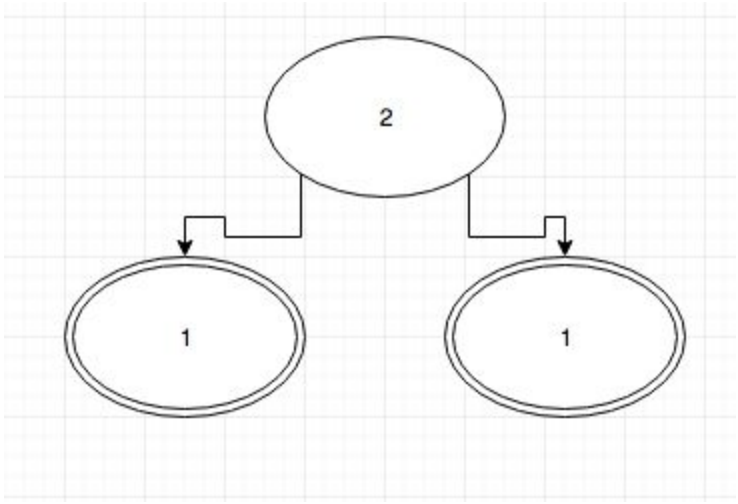
9)
Question: Suppose T is a Huffman tree for a set of characters having frequencies equal to the first n nonzero Fibonacci numbers, {1, 1, 2, 3, 5, 8, 13, 21, 34, . . .}, where f0 = 1, f1 = 1, and f(i) = f(i−1) + f(i−2). Prove that every internal node in T has an external-node child.

ANSWER:
We can prove this using a proof by induction:

We will use Huffman's Algorithm for heapifying the numbers and start with our base case where we add the first numbers to form a small subtree and then show the inductive case where we group together f(k) and f(k+1) where f(k) is the fibonacci numbers.

Base case:

When we use Huffman's Algorithm, we put the first two numbers under a new root which is the addition of the two children which is 1 + 1 = 2

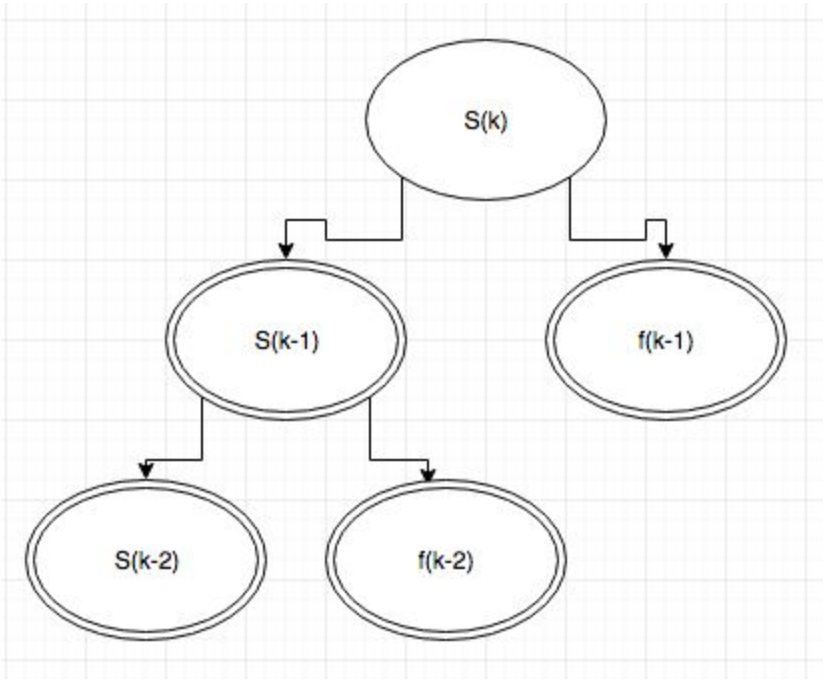


As we can see in the base case where the new root is an internal node has at least 1 external node and thus satisfies the condition.

Inductive Hypothesis: There is an external node for each internal node in T

Inductive case:

Using Huffman's algorithm, where you pick two minimums and then put them under a new sum such as seen above where 1 and 1 is taken and added to equal to 2 and then re-inserted into the list.
We know that with a fibonacci's number aside from the base case which we already proved is that f(k) = f(k-1) + f(k-2) and lets call S(k) what happens when you put f(k-1) in a union with S(k-1) which is a union of f(k-2) and S(k-2) like below:



S(k) will be greater than or equal to f(k) as f(k) is equal to f(k-1) + f(k-2) while S(k) is equal to S(k-1) + f(k-1) and we can cancel out it as seen below and we can also used S(k-1) or the 2 from the union of 1 and 1 from our base case for an example:

f(k) = f(k-1) + f(k-2)
S(k) = S(k-1) + f(k-1)
S(k-1) = S(k-2) + f(k-2)

We can substitute S(k-1) to get:

S(k) = S(k-2) + f(k-2) + f(k-1)

And if we cancel out both f(k-2) and f(k-1) from both equations, we are still left with S(k-2) which can not be 0 as we don't have any zeros in the sequence.

f(k) = ~~f(k-1) + f(k-2)~~
S(k) = S(k-2) + ~~f(k-2) + f(k-1)~~

Which means that S(k) > f(k)

As well S(k) < f(k+1) as seen below:

f(k+1) = f(k) + f(k-1)
S(k) = S(k-1) + f(k-2) + f(k-1)

We can rewrite and cancel out:
f(k+1) = f(k-1) + ~~f(k-2) + f(k-1)~~
S(k) = S(k-2) + ~~f(k-2) + f(k-1)~~
And f(k-1) will be greater than S(k-2) as we can use our base case example where S(k-2) stores 1 and f(k-1) stores 2.

So by huffman's algorithm, S(k) and f(k) will be the minimum and they will be put into a minimum and they will be put together creating a new root but because f(k) is only one node, it will leave behind an external node and this will be repeated for f(k+1) and thus proving that each step there will be one external node left out and thus proving the inductive case.

10)

```
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
Euns-MacBook-Air:src mding$ ls
ConnectedComponentsImage$1.class        asn2.sh
ConnectedComponentsImage.class          girl.img
ConnectedComponentsImage.java           testUnionFind.class
FindConnectedComp.class                 testUnionFind.java
FindConnectedComp.java                  uandf.class
UFNode.class                            uandf.java
UFNode.java
Euns-MacBook-Air:src mding$ whoami
mding
Euns-MacBook-Air:src mding$
```

My english name is Michael Ding and my whoami leads to mding

Running my program using bash:

```
Euns-MacBook-Air:src mding$ ./asn2.sh < girl.img
Will read image from ./girl.img file unless you specify not to
```

Q10b)

```
Euns-MacBook-Air:src mding$ whoami
mding
Euns-MacBook-Air:src mding$ ./asn2 < girl.img
-bash: ./asn2: No such file or directory
Euns-MacBook-Air:src mding$ ./asn2.sh < girl.img
Will read image from ./girl.img file unless you specify not to

Q10b1) Printing input binary image:
------------------
```


Q10b2) Printing unique characters:
Final_sets is called, number of sets : 21

```
Q10b3) Ranked component list:
-----------------

Component Label: e, size: 1
Component Label: f, size: 1
Component Label: k, size: 1
Component Label: n, size: 1
Component Label: p, size: 1
Component Label: r, size: 1
Component Label: s, size: 1
Component Label: t, size: 1
Component Label: c, size: 2
Component Label: i, size: 4
Component Label: g, size: 5
Component Label: a, size: 11
Component Label: q, size: 18
Component Label: b, size: 20
Component Label: d, size: 23
Component Label: h, size: 33
Component Label: l, size: 110
Component Label: u, size: 354
Component Label: m, size: 510
Component Label: j, size: 578
Component Label: o, size: 779

Q10b4) Ranked component list w/o sizes 1 or 2:
-----------------

Component Label: i, size: 4
Component Label: g, size: 5
Component Label: a, size: 11
Component Label: q, size: 18
```