Steven Kravitsky
CS 260

Homework 5

1)

$$\begin{tikzcd} 1 \arrow[dr] & & & \\ & 2 \arrow[dr] & & \\ & & 3 \arrow[dr] & \\ & & & 4 \end{tikzcd}$$

$$\begin{tikzcd} 1 \arrow[dr] & & \\ & 3 \arrow[dl] \arrow[dr] & \\ 2 & & 4 \end{tikzcd}$$

$$\begin{tikzcd} 1 \arrow[dr] & & \\ & 2 \arrow[dr] & \\ & & 4 \arrow[dl] \\ & 3 & \end{tikzcd}$$

$$\begin{tikzcd} & 1 \arrow[dr] & \\ & & 4 \arrow[dl] \\ & 3 \arrow[dl] & \\ 2 & & \end{tikzcd}$$

$$\begin{tikzcd} & 2 \arrow[dl] \arrow[dr] & & \\ 1 & & 3 \arrow[dr] & \\ & & & 4 \end{tikzcd}$$

$$\begin{tikzcd} & 2 \arrow[dl] \arrow[dr] & \\ 1 & & 4 \arrow[dl] \\ & 3 & \end{tikzcd}$$

$$\begin{tikzcd} & & 3 \arrow[dl] \arrow[dr] & \\ & 2 \arrow[dl] & & 4 \\ 1 & & & \end{tikzcd}$$

$$\begin{tikzcd} & & 3 \arrow[dl] \arrow[dr] & \\ & 1 \arrow[dl] & & 4 \\ 2 & & & \end{tikzcd}$$

$$\begin{tikzcd} & & & 4 \arrow[dl] \\ & & 3 \arrow[dl] & \\ & 2 \arrow[dl] & & \\ 1 & & & \end{tikzcd}$$

$$\begin{tikzcd} & & 4 \arrow[dl] \\ & 2 \arrow[dl] \arrow[dr] & \\ 1 & & 3 \end{tikzcd}$$

$$\begin{tikzcd} & 4 \arrow[dl] & & \\ 1 \arrow[dr] & & & \\ & 2 \arrow[dr] & & \\ & & 3 & \end{tikzcd}$$

$$\begin{tikzcd} & 4 \arrow[dl] \\ 1 \arrow[dr] & \\ & 3 \arrow[dl] \\ 2 & \end{tikzcd}$$

2)

$$\begin{tikzcd}
 & & \boxed{7} \arrow[dl] \arrow[drr] & & \\
 & \boxed{2} \arrow[dl] \arrow[dr] & & & \boxed{8} \arrow[dl] \\
\boxed{0} & & \boxed{5} \arrow[dl] \arrow[dr] & \boxed{9} & \\
 & \boxed{1} & & \boxed{6} &
\end{tikzcd}$$

3)

$$\begin{tikzcd}
 & & \boxed{8} \arrow[dl] \arrow[dr] & \\
 & \boxed{2} \arrow[dl] \arrow[dr] & & \boxed{9} \\
\boxed{0} & & \boxed{5} \arrow[dl] \arrow[dr] & \\
 & \boxed{1} & & \boxed{6}
\end{tikzcd}$$

$$\begin{tikzcd}
 & & \boxed{8} \arrow[dl] \arrow[dr] & \\
 & \boxed{5} \arrow[dl] \arrow[dr] & & \boxed{9} \\
\boxed{0} & & \boxed{1} \arrow[dr] & \\
 & & & \boxed{6}
\end{tikzcd}$$

4) The order of deleting the nodes will not matter as long as the tree is being restructured. This will not change the final structure of the tree in any way.

5) The make_heap() algorithm is creating a heap using random integers between 1 and 500. At most the algorithm would move n items up the tree, with the height on log(n) – 1. The maximum tree height and n items in the tree means that the worst case for the tree is O(n * log(n))

6) A binary heap is a tree that is O(n*log(n)) to create from scratch if there are n elements used. This means that to pop off each element it would take n time to complete the sort. This means that the complexity and average time of this sorting method is about the same as merge sort and quicksort.

7) As stated in the problem we will need a structure that will keep the data stored to remember the correspondence between the first iteration and array indices. This structure will be called temp_data and use a height, position, and computed_value variable. After this structure is created we can look through the values of my_data using the given first() and next() iterator functions given to us which will give us a float value and height that we can store in an array, we will call impressive_A() using this array later. The temp_data struct should be used here to keep track of the position, height, and computed_value so that we can compare them later. We can now call impressive_A() on the array we created now that the data has been all looped through. A second pass of the data needs to be done, iterating through the my_data struct again in order to compare the data from temp_data to my_data. If the height matches then the position variable can be used to put the value from the array we made into my_data.

```
Struct temp_data {

        Float height;

        Int position;

        Double computed_value;

}
```

8)

```
def print():
    while(Stack.first() is not Null):
        Stack2.push(Stack.pop())
    while(Stack2.first() is not Null):
        print Stack2.pop()

def insert(d):
    while(Stack.first() is not Null):
        Stack2.push(Stack.pop())
    Stack.push(d)
    while(Stack2.first is not Null)
        Stack.push(Stack2.pop())

def remove(d):
    while(Stack.first() is not Null):
        x = Stack.pop()
        if x != d:
            Stack2.push(x)
    while(Stack2.first is not Null)
        Stack.push(Stack2.pop())
```

The time complexity for insert() would be O(1) because first(), push() and pop() can all be done in constant time. The time complexity for remove() would also be O(1) because first(), pop(), and push() can all be done in constant time.

9) In this example we are given two lists, List1 with the server checksums and List2 with the client checksums. We can create a third list, List3, that will store all the values that did not match between the two original lists. We can iterate over the checksums in List1 and compare them to the values in List2. If the value does not match anything in List2 we can put that value in List3 to show that it did not have a match.