HW #2

Q1) Heapsort (A)

```
    // build empty array
    N = buildMinHeap (A)
    while N is not empty {
            min = Extract-Min (N)
            append min to the empty array
    }
    return empty array
```

The time complexity for Heapsort is $O(n \log n)$. When a sort is stable, it does not change the position of same or equal elements. When you call buildMinHeap(A), Heapify (A, i) is called, which runs instable swaps while swapping a parent and child.

Suppose the left and right child share the same key:

ex; <u>Left child</u>: Suppose there is an array $[7\ 8\ 2_a\ 5\ 2_b]$. After one run, it becomes $[7\ 2_b\ 2_a\ 5\ 8]$. After the second run, it becomes $[2_b\ 5\ 2_a\ 7\ 8]$, thus the 2_b would be unstable because it is at the first index.

<u>Right child</u>: Suppose there is an array $[7\ 2a\ 2_b]$. First run results in $[2_b\ 2a\ 7]$, with 2_b being unstable at the first index as well

Q2) Largest K Elements (A, K)
H= buildMaxHeap (A)
for k
Extract-Max (empty array)
Append value for empty array
Empty array = (H [i_1], i_1) and (H[i_2], i_2)
Reverse sorted order
Return array

The time complexity for building a max heap is O(n), Extract- Max is O(log k), and reversing is O(k). The overall runtime is $\boxed{O(n + k \log k)}$ since making the heap was O(n) and extract-max was O(log k) with a maximum of 2k elements.

$\Omega(2^h)$ $O(3^h)$

Q3) BC: $h=1$

$2^1-1=1$ $3^1-1=2$

$2-1=1$ $3-1=2$

$1=1$ ✓ $2=2$ ✓

IH: Assume the number of keys stored is true up to h.

IS: Prove h+1 is true that the number of keys is between 2^h-1 and 3^h-1.

A tree with height h+1 has a root with either 1 or 2 keys and 2 or 3 subtrees with a height of h.

According to the IH, these subtrees keys are at least 2^h-1 but less than 3^h-1. Thus the total keys is at least $2(2^h-1)+1$ but less than $3(3^h-1)+2$.

when simplified:

$$2(2^h-1)+1 = 2^1(2^h-1)+1 = 2^{h+1}-1$$
$$3(3^h-1)+1 = 3^1(3^h-1)+1 = 3^{h+1}-1$$

Hence, the number of keys is

$\Omega(2^h)$ → at least

and

$O(3^h)$ → at most.

Q4)

Algorithm:

1.) Find height of T_1 and T_2.
2.) If the heights are the same, return a new 2-3 tree with a root x, left child T_1, and right child T_2.
3.) Find the leftmost node in T_2.
4.) once you find it, insert X into this to make it the new leftmost node and make T_1 the left child.
5.) Fix_overfull (N)
6.) Return T_2

when you insert X into N the time complexity is $O(1)$. The final running time however is $O(\log h_1 + \log h_2)$ where h= heights of T_1 and T_2. when using Fix_overfull. The tree is originally having the leaves at equal heights, so by using Fix_overfull you can recursively fix the problem of having three keys.

Q5) InsertWithSize (x)
// Call Insert (x)
// Increment N.size for all nodes by 1

DeleteWithSize (x)
// Call Delete (x)
// Decrement N.size for all nodes by 1

Range (a, b)
// Search for a and move left. Search for b and move right.
// Form left array and right array of nodes.

```
k = 0;
while (left array equals right array) {
    root = leftArray [k]
    k++;
}
s = 0;
for (nodes after left array)
    if left node is next, then s = s + N.left.size
for (nodes after right array)
    if right node is next, then s = s + N.right.size
return s
```

continued →

The running time analysis in terms of n and D is O(D) because it is the maximum depth. It is O(D) for Insert, Delete, and Range. Range uses O(D) for Find and s so the sum simply equals O(D) as well.

Examples of correctness:

1) If the left node is in the left array, then the node is greater than a, thus the keys after it are after a and are in [a,b].
2) If the right node is in the left array, then the node is less than a, thus the keys before it are before a and are not in [a,b].
3) If the left node is in the right array, then the node is greater than b, thus the keys after it are after b and are not in [a,b].
4) If the right node is in the left array, then the node is less than b, thus the keys before it are before b and are in [a,b].