Q1. Sorting with heaps.

1) Build a heap from an array
2) The min-element should be at the root of the heap.
3) Swap the root with the last element (Extract min)
4) Decrease the heap by 1
5) Call heapify on the root.
   Heapify for min: Find the min element of the 2 children node and swap if needed.
6) repeat steps 2-5
7) Since min-heap will be from decending order, we need to swap the array.

This will take $O(n \log n)$ time $O(n)$ from the array and $O(\log n)$ for heapifying + element.

Heapify is not stable:

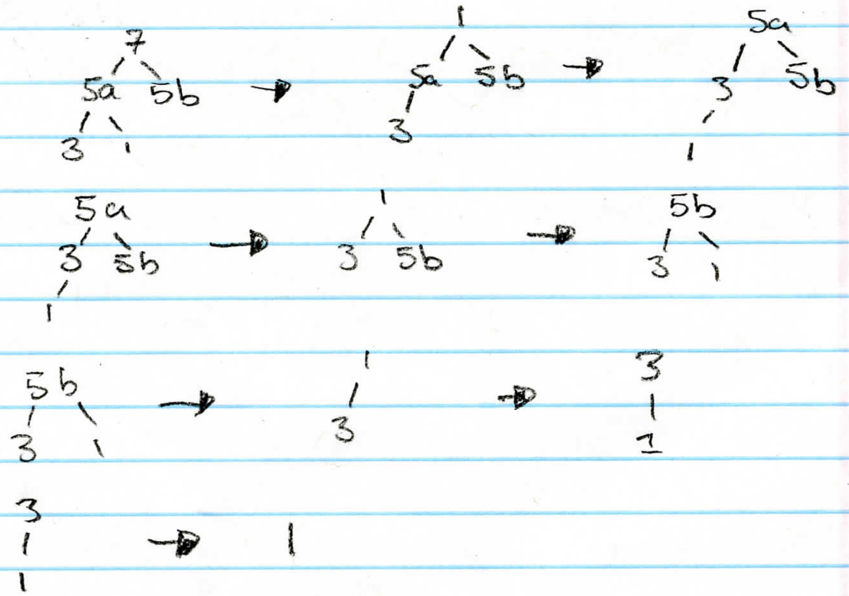
Ex, Max-heapSort

| 7, 5a, 5b, 3, 1 |

| 5a | 3 | 5b | 1 |

| 5b | 3 | 1 |

| 3 | 1 |



Sorted = | 1 | 3 | 5b | 5a | 7 |
Thus unstable, the 5's changed places

HW2

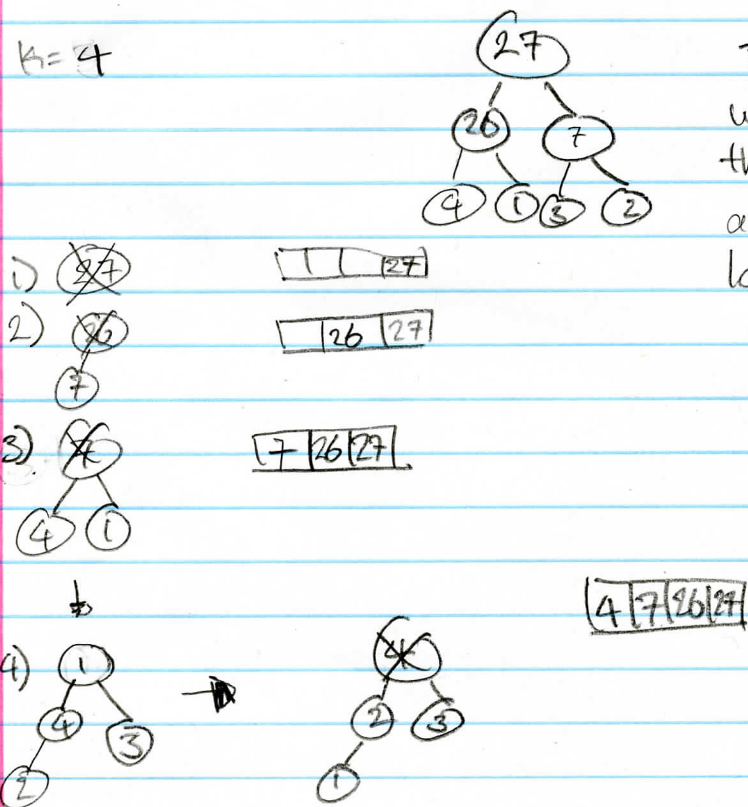
Q2.

1) We build a heap from the array, this heap is read only. $O(n)$
2) We create another heap called p.
3) Add the root of the original heal
4) pop it from p and add its children
5) Call heapify $O(\log k)$
6) Pop the root of p and add the children of the popped element in the original heap.
7) repeat 5-6 (k-1) times $O(k)$

This should take $O(n + k\log k)$ time because n for building a heap and we call heapify k times on p, so it will be $k\log k$.

Since we add 2 and pop 1 we only grow the tree by 1 after a call, so k calls will get us $\log_2 k$ rows

Ex1 k=4



This always works because we have a pointer to the original heap and add its children. We never lose the greatest element

Q3. Proving $\Omega(2^h)$

Assume every node will have 1 element

Base Case (2^h) $h=1$ $2^1=2$

By definition $f(n) \geq c \cdot g(n)$ if $c=\frac{1}{2}$ $1 \geq \frac{1}{2}(2) = 1 \leq 1$ ✓

$\exists c>0 \ \forall n > n_0 \ f(n) \geq c \cdot g(n)$

Inductive hypothesis: Assume 2^k is true number of keys in a 2-3 tree for k height

Inductive Step: prove 2^{k+1} for k+1 height.

Since the total number of nodes in level k is 2^{k-1}, the level after 2^k will have 2^{k-1+1} nodes. $2^k + 2^k$ elements is 2^{k+1}, thus proven.

Lemma: Total number of nodes at level k is 2^{k-1}

Base Case $(k=1)$ $\square \leftarrow$ 1 node at level k $2^{1-1} = 2^0 = 1$ ✓

Inductive hyp.: Assume at level k, there are 2^{k-1} nodes at level k

Prove 2^k Inductive step: At level 2^{k-1} there will be 2 children nodes for every parent node, therefore $2^{k-1} \cdot 2 = 2^{k-1} \cdot 2^1 = 2^{k-1+1}$ $= 2^k$ nodes thus proven

At every level there will be $2^h - 1$ nodes and we have to prove $\Omega(2^h)$, but since $2^h - 1 \geq .5(2^h)$ for every n after n_0, $\Omega(2^h)$ holds

■

Q3. $O(3^h)$ Assuming every Node has 2 keys

Base: $h=1$ $3^1 = 3$ $2<3$
base case holds

Inductive hypothesis: Assume 3^h is the total number of keys for h

Inductive Step: prove 3^{h+1} is the total number of keys for h+1

Since the total amount of nodes on each level is 3^{h-1} if we add a level to the last level it will be 3^{h-1+1} or 3^h and since every node has 2 elements the next level will have $3^h \cdot 2$ nodes. If we add the last level of element with the previous element it will be $3^h \cdot 2 + 3^h = 3^h(2+1)$

$$= 3^h \cdot 3$$
$$= 3^{h+1}$$ as proven.

Lemma: Total amount of nodes on each level is 3^{h-1} assuming every node has 2 elements.

ase Case $h=1$ $3^{1-1} = 1$ 1node = 1node ✓

ductie hypothss Assume 3^{k-1} is true

luctie Step Prove 3^{k-1+1} or 3^k for next level.

Since every parent will have 3 children the next row will have, $3^{k-1} \cdot 3 = 3^{k-1} \cdot 3^1$

$$= 3^{k-1+1}$$
$$= 3^k$$ as proven

Q4.) Assuming $T_1 < T_2$ size Given x

then we find the height of the diffence between $|T_1$ and $T_2|$
$T_2 - T_1 = h1$

Go to the leftmost part of T2 at h1 and Insert x there. Insert T_1 on the left side of that node. If the node has 2 elements. done.
If the node has 3 elements call heapify.
This will take $O(h1)$ time since fixOverfull will push things up.

Case 2: Assume $T_1 > T_2$ size Given $T_1 < x < T_2$
Variation of Case 1

$T_1 - T_2 = h$

Go to the rightmost part of T_1 and Insert x into the node.
Add T_2 tree into the right part of the x node.
Call fixOverfull (taking $O(h)$ time).

Case 3: $T_1 == T_2$ Size

Insert x into the root and place T_1 to the left an T_2 on the right.

This will take $O(1)$ time.

In total this will take $O(|T_1.size - T_2.size|)$ and $\Omega(1)$ time.

Q5. Augment the tree such that nodes hold subtree sizes.

Insert (Node n) {
- Traverse the tree as usual to see when n should be inserted.
- Increment the size of each node that it passes by 1 on its path to a leaf.

}

O(D) time.

Delete (Node n) {
- Traverse the tree to find n
- Decrement the size of each node it passes by 1 to the n node.

}

O(D) time.

Range (node a, node b) {
Int total = root.size();
traverse to find a and if a goes right total - left subtree of that node.
Traverse to find B and if B goes left Total - right subtree of that node
} O(D) time

takes O(D) to find a and O(D) to find b.
O(1) time to find range of a and b.
So O(D).