1 Problem 1 (20 points).

Let $A[1 \dots n]$ be a max-heap. The operation max-heap-delete(A, i) deletes the heap element A[i] from the heap, so that the rest after the deletion is still a max-heap. Please give the pseudocode of max-heap-delete(A, i) and its time complexity in big-oh notation.

```
max-heap-delete(A, i) {
    A[i] = A[A.length - 1] //replace A[i] with last element
    heapsize-- //reduce heap size by 1
    max-heapify(A, i) //heapify from the replaced element
}
```

Since the only recursion happening is the recursion within the heapify function, the time complexity of this algorithm is still $O(\log n)$ just like the heapify function.

2 Problem 2 (20 points).

Give an $O(n \log k)$-time algorithm that uses a min-heap of size no more than k to merge k SORTED lists into one sorted list, where n is the total number of elements in all input lists. Give your algorithmic idea and pseudocode and explain why your algorithm has a time cost of $O(n \log k)$. Explain how you will use your algorithm for mergesort if the mergesort has a setting of k-way splitting.

2.a Algorithm Idea

The biggest hint here is that k is the number of lists and also the maximum number allowed in the heap at any point. It also explains the $\log k$ portion of the time cost, meaning heapify can be used here (having a time cost of $\log k$).

The first step would be to put the first element of each of the k sorted lists into a min heap. This will give the min heap of maximum size k that's required. It also guarantees that the smallest overall element is in the heap since all of the k lists are sorted. Since this is set up as a min-heap, the smallest overall element is going to be on top, so that element is placed in the final sorted list as the very first element, and the root node of the min-heap is replaced by the next element in the list we took that element from. We then call min-heapify to return to the state of the smallest element being the root node.

From here, one of two things are true. Either the next element in the list we took from is the next smallest, or the next smallest element is in the rest of the first elements we took. This is because since all the lists were presorted, everything BEYOND the elements in the heap are

going to be larger than what's there, and even more-so larger than the next smallest element. So we repeat the same steps of putting the root node (smallest, since it's a min-heap) into the final sorted list, replacing it with the next element of the list it came from, and continuing the process.

Once a list is empty after an element is taken from it, we can simply call extract-min to remove it since we have nothing to replace it with. This will cause the heap to be empty at the very end which can save a little memory consumption near the end of the process.

Note: This algorithm idea requires that the k lists be linked lists, or some other data structure that has the ability to get the next element in said lists. This is because of the iteration needed to keep going through the lists we're grabbing elements from.

2.b Pseudocode

```
merge-sorted(lists) {
    for each list {
        minheap.insert(list[0])
    }
    i = 0
    while heapsize > 0 {
        result.next = minheap.extractmin()
        if extracted min.next != null, minheap.insert(extracted min.next)
    }
}
```

2.c Time complexity

The main expensive portions of this algorithm are the minheap.insert calls, the minheap.extractMin calls, and the for/while loops. the insert and extractMin calls all have $\log k$ time complexity, because we've set the heap size to max out at k elements. The instances of these calls are within for and while loops. The for loop is going to go through k elements, one for the first element in each of the k lists, and the while loop is going to end up iterating through every element in the entire dataset, which is n. This means the total time complexity for this algorithm is $((n+k)\log k) = (n\log k + k\log k)$ and the largest term in this $n\log k$, making the time complexity $O(n\log k)$.

3 Problem 3 (20 points).

Planning EWU's thanksgiving party: Everyone (including all the faculty, staff, and students) at EWU has one and only one direct supervisor except the president, so the supervisorship

among the people of EWU can be represented as a rooted tree where a node is the direct supervisor of its child nodes (if the node has child nodes) and the president is the root of this tree. Now the HR office of EWU is planning the Thanksgiving dinner for the EWU community. In order to let everyone relax and have fun, all the people invited for the dinner should not have their direct supervisors invited. Now you, as the director of EWU HR, are given the supervisorship tree and need to decide who should be invited. Your goal is to invite as many people as you can, as long as everyone can relax. Describe your strategy for picking the guests and explain why your strategy works. [Hint: It seems that you all should be invited and I should not, assuming I am your direct supervisor and you all do not have supervisees.]

My strategy would be to use greedy-style recursion to get as many people invited as possible starting from the bottom of the tree. Going from left children to right children in the tree, follow the tree's branches to the very bottom. Once the bottom is reached, return to the parent (supervisor) and send an invitation to all of that node's children (supervisees). Go to the next element on that same level and do the same thing. If the next one has no children (supervisees), invite that person instead.

If, when going up levels in the tree, a child node (supervisee) of a person has been invited, simply don't send and invitation to that person. Repeat this process until the entire tree has been searched.

The reason why this strategy works is because greedy-style recursion is perfect for a problem like this. Since each person has one and ONLY one supervisor, the worst case scenario for inviting one person is losing an invitation for the person above them, when the person above them only has ONE supervisee, meaning no change is made to the maximum possible and nothing is lost. The best case scenarios include a person having several supervisees, in which case as many of those people are invited as possible given the process, and the person in charge of them is the ONLY person not invited. This is why the leaf nodes of the tree are prioritized, because they are the people with zero collision issues with the given constraints, because they have no supervisees.

4 Problem 4 (20 points).

You are given a set of n unit-length tasks, i.e., each task has length 1. However, each task i, for $1 \leq i \leq n$ can start at the earliest possible time r_i, and must finish by its deadline d_i. For example, if $r_i = 3$ and $d_i = 8$ that means you can choose to start the ith task at any of these clock times: $3, 4, 5, 6, 7$, because: 1) it is allowed to be started as early as at clock time 3, and 2) you must get started by no later than clock time 7 because the task's deadline is 8 and its length is 1.

You are asked to schedule a maximum number of tasks within the given constraint, so that all the picked tasks can be scheduled without conflicts. Describe your strategy on picking

what tasks and how to schedule them. Explain why your strategy works.

My strategy for this would start similar to the in-class example by sorting all the tasks by deadline in ascending order. However, simply picking tasks based on deadline is neither necessary nor optimal, since each task has a length of 1 and does not take up the whole space of the start to deadline. Instead, the algorithm would also need to factor in when the start time is, and schedule the tasks with the earliest deadline at the task's earliest start time, prioritizing the earliest start time for ties in deadline.

What this strategy does is, throughout the picking process, those tasks that have the earliest deadlines are scheduled whenever they can, and don't have their available time slot covered up by other tasks that have later deadlines and therefore could be scheduled later. For example, if one task has a deadline of 4, but three other tasks with a deadline of 7 were scheduled first, the entire possible time slot for the first task is completely taken up when all three of the other tasks could be scheduled after the deadline of the first one.

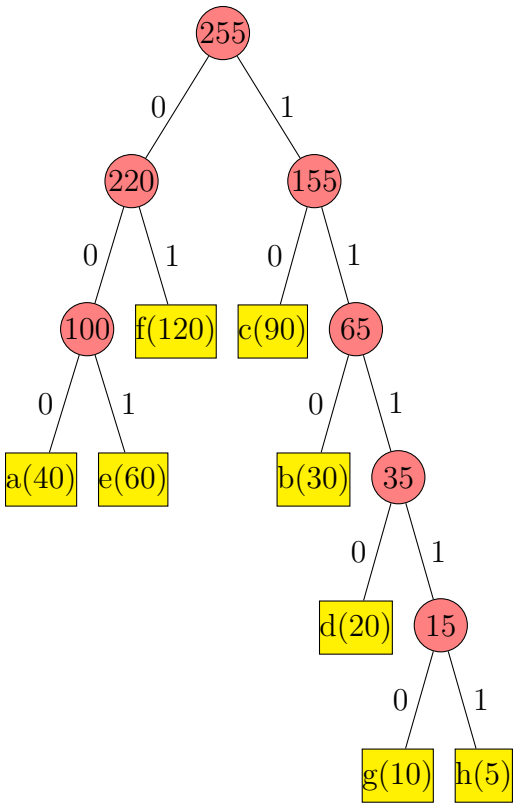
Also, regarding ties, if there are two tasks that share the same deadline, the task with the earliest start time should be chosen first. This can prevent any weird collisions that could happen with much larger data sets or possible instances where tasks could be neglected when done in a random order.

Lastly, the biggest advantage of this strategy is due to how the deadline system works. This system means that for every existing deadline value in the set, at least one task is guaranteed to be scheduled at the time slot minus 1, being that task that has that deadline. This allows for as much tail-chaining as possible, which is the optimal outcome when each task has a set value of 1, meaning the most optimal solution is filling up as much space as possible, therefore completing as many tasks as possible which is the goal.

5 Problem 5 (20 points).

Suppose you are given a text of 375 characters drawn from the alphabet a, b, c, d, e, f, g, h and the frequency of each letter is a:40, b:30, c:90, d:20, e:60, f:120, g:10, h:5

1. Create a Huffman tree for this text (you may have multiple different Huffman trees for this text, but anyone is fine).
2. Show the huffman code of each letter.
3. Compute the size of the Huffman code compressed version of this text in bits.
4. Calculate the compression ratio: compressed text size in bits / raw text size in bits, if you use 8-bit ASCII code for storing the raw text.

1. 

2.
   - a: 000
   - b: 110
   - c: 10
   - d: 1110
   - e: 001
   - f: 01
   - g: 111110
   - h: 111111

3. Full compressed version in bits

   $(40 + 30 + 60) * 3 + (90 + 120) * 2 + 20 * 4 + (10 + 5) * 6$

   $130 * 3 + 210 * 2 + 20 * 4 + 15 * 6$

   $390 + 420 + 80 + 90$

   980 bits

4. Raw text size $= 375 * 8 = 3000$

   Compression ratio $= 980/3000 = 32.67\%$