Section 4

Agenda

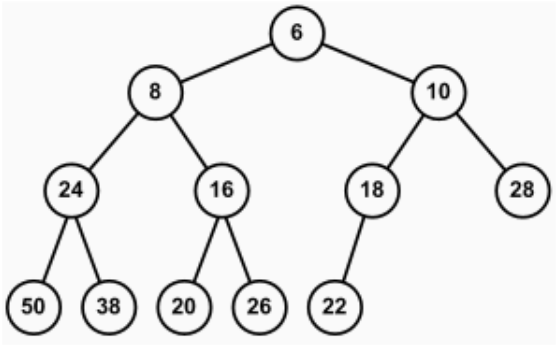
- Mini Assignment
- Heaps and Deques
- Sorting and Master Theorem Review
- Pseudocode Review
- Optional Problems

Mini Assignment

1. What is the difference between a position and an entry? Which holds the key/value pair?
    a. Position is the tree node, and it contains an entry. The entry is the data (key-value pair stored in that position.
2. When you upheap, does the position or the entry move?
    a. Entry moves
3. Give a real world application of a priority queue. What would the key/value pair types be?
    a. Emergency room triage -- key would be the severity of the issue, and the value would be the patient.
4. What's the runtime of heap add and remove?
    a. Heap add and remove are O(logn). This is because when an element is added or removed, in the worst case, you'll have to upheap or downheap the entire height of the tree, and we know the height of a binary tree is worst-case log(n).

Heap Clarifications

- Remember you will be implementing an **adaptable priority queue**. This means that we must be able to remove any internal nodes, not just the min or the max
- What makes that different? **You must reorganize the heap.** This means upheaping and downheaping when necessary



Upheping and Downheaping

- Downheaping example: remove 8 from the above heap.
  - Switch 8 and 22
  - Delete 8 (which is now the furthest right leaf)
  - Downheap 22 (first swap 16 and 22, and then swap 20 and 22)
- Upheap example: remove 50 from the above heap. (before the modifications above)
  - Switch 50 and 22.
  - Switch 22 and 24.
  - Remove 50.

Deques

Deques (stands for double-ended queues) are a data structure that will become very useful to you (especially while implementing heap!). They are essentially stacks, but you can push and pop from either side (like you could a deck of cards). You'll use a deque in heap in your MyLinkedHeapTree class to maintain the tree's left-completeness when you add and remove nodes. Take a look at the Deque Help Slides that we went over.

Add and Remove Pseudocode

```
def add(N node):
    if deque.isEmpty():
          root = node
          deque.addToBack(root)
    else:
         front = deque.getFront()
         if front.hasLeft(): // has a left child
            deque.popFront()
            front.addRight(node)
            deque.addToBack(node)
         Else: // does not have a left child:
            front.addLeft(node)
            deque.addToBack(node)
```

```
def remove():
        if not deque.isEmpty():
          removed = deque.popBack()
        if deque.isEmpty():
          root = null
        else:
          parent = removed.getParent()
          if removed.isLeftChild():
              parent.removeLeft()
          else:
              parent.removeRight()
              deque.pushFront(parent)
```

Sorting and Master Theorem

Take a look at lecture slides for the pseudocode on insertion and selection sort as well as for the rules for Master Theorem! No new information here, just review :)

Editing Pseudocode

We went over how to write "good" pseudocode using the following example:

```
function FUNction():
  aIndex = bIndex = 0
  while aIndex < A.length:
        while bIndex < B.length:
            if A[aIndex] <= B[bIndex]:
              result add A[aIndex]
                        aIndex = aIndex+1
            else:
                        result add B[bIndex]
                        bIndex = bIndex +1
  if aIndex < A.length:
    result = result + A[aIndex...end]
  if bIndex < B.length:
    result = result + B[bIndex...end]
  return result
```

What's wrong with this pseudocode?

- No input parameters (what if i create array C and array D and want to manipulate those?)
- Bad name (the name should tell us what the function is doing; this one is far too vague)
- Result is never defined
- Double while loops (runtime runtime runtime)
- Not specific (where will A[aIndex] be added to result?)
- Not concise (should use aIndex++ instead of aIndex = aIndex+1)

Now here's some improved pseudocode:

```
function merge(A, B):
  // Input: two sorted lists
  // Output: 1 sorted list

  result = []
  aIndex = bIndex = 0
  while aIndex < A.length and
          bIndex < B.length:
    if A[aIndex] <= B[bIndex]:
      result.append(A[aIndex++])
    else:
      result.append(B[bIndex++])
  if aIndex < A.length:
    result = result + A[aIndex...end]
  if bIndex < B.length:
    result = result + B[bIndex...end]
  return result
```

Optional Problems

1. Given a list of N comparable items, return a list of the M largest items from that list.

Hint: Use a minimum priority queue/heap

Solution (SPOILER ALERT! No peeking! Give the problem your best shot before scrolling): Build min-heap of capacity M that you populate as you go, removing the minimum and replacing it with the next element if that element is greater than the smallest item of the min-heap

```
function getMlargest(array, M):
    pq = priority queue
    for element in array:
        if (pq.size < M):
            pq.insert(element, element)

        else:
            smallest = pq.peek()
            if smallest < element
                pq.removeMin()
                pq.insert(element, element)
    largest = build array from pq
    return largest
```

Unrelated Note: You can use built-in data structures along with their functions! You don't have to make and use data structures from scratch! Ex: "h = Hashset(), pq = Priority Queue" in pseudocode

2. Space vs. Runtime efficiency: Find the duplicates in a list.

Multiple solutions (no scrolling until you've tried it!):

- Add to hashtable which is linear for setup and space usage but constant for each duplicate searched
- Sort the list which is constant in space usage and O(nlogn) setup, but linear for each duplicate searched

- In what cases is it better to use each solution? If you have 100 items in the set vs. 100,000,000? How about if you're making searching for the duplicate of 1 term vs. 500? If you're constrained by space but not by processing speed?