Final Review Solutions

Heaps

- Motivation: What if we always want to find the minimum or maximum element?
- Keep high priority items at the top
  - Min heap: high priority corresponds to low priority value
  - Max heap: high priority corresponds to high priority value
  - Notice the difference between priority and priority value!
- Represented as a binary tree with two more properties:
  - Complete: No empty spaces other than on the right-hand side of the bottommost level. Consequence: height will be $\Theta(\log N)$ where N is the number of nodes
  - (Min/Max)-Heap property: for a particular node n, the children of n must have (greater/-lesser) priority value than n. Consequence: the root will always contain the (lowest/highest) priority value element
- Methods (of a min-heap)
  - `peek()`: returns (but does not remove) the item with the minimum priority value; runtime is $\Theta(1)$
  - `removeMin()`: returns (and does remove) the item with the minimum priority value; runtime is $O(\log N)$
    - Take the item in the bottom-rightmost position and replace the value at the root
    - Bubble down the new root value
  - `insert(T item, int priorityVal)`: Insert the item with priority value of `priorityVal` into the heap; runtime is $O(\log N)$
    - Insert the item in the bottom-rightmost position
    - Bubble up the new inserted value
- Bubbling (of a min-heap)
  - Bubble up: while the priority value of a particular node n is less than the priority value of its parent, swap the two
  - Bubble down: while the priority value of a particular node n is greater than the priority value of its child/children, swap the two (always pick the lesser of the two children if both have priority value less than the current node)

- Representation
  - Number each element in the heap, starting from 1, left to right top to bottom, this will represent the index of the item in the array!
  - For a particular node at index i:
    - Parent is at index $\frac{i}{2}$
    - Left child is at index $2i$
    - Right child is at index $2i + 1$
- `PriorityQueue<T>`
  - Implemented with min heaps
  - Methods: `T poll()`, `T peek()`, `void push(T item)`
  - Can use own `Comparator` object to change how the `PriorityQueue` organizes elements

Practice Problems

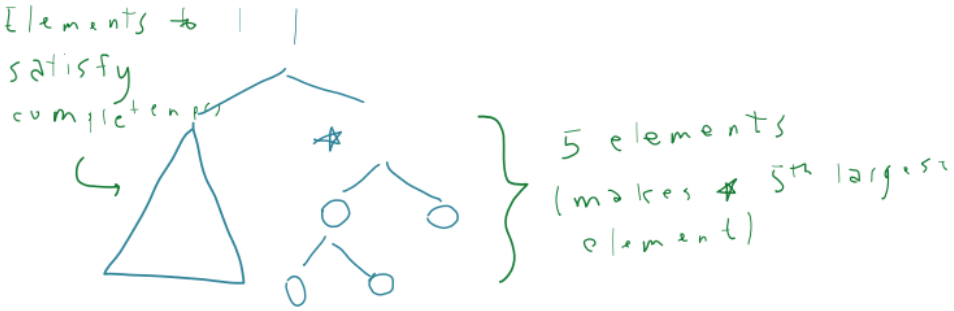
1. What is the size of the largest binary min heap that is also a valid BST? Draw an example assuming all keys are letters from the English alphabet. **Assume the heap has no duplicate elements**
   A heap must satisfy two properties:
   (1) Heap-Order Property: In a min-heap, this means that each node is smaller than each of its children.
   (2) Completeness: there are no holes in the heap as you go through the tree in a level-order traversal.
   Recall that a binary search tree (BST) satisfies the binary search invariant: all elements in the left subtree are less than the key in the root, and all elements in the right subtree are greater than the key in the root.
   Combining these facts, we know that we can only have elements in the right subtree, but this violates the completeness requirement for a heap. **Thus, the maximum size is 1**.

2. What is the size of the largest binary max heap that is also a valid BST? Draw an example assuming all keys are letters from the English alphabet. **Assume the heap has no duplicate elements.**
   See the detailed explanation above, with the following modification: in a max-heap, each node is larger than each of its children for the heap-order property. Because of this, we are able to have one element in the left subtree, and this does not violate the completeness property as before. **Thus, the maximum size is 2 and a valid example is a single node with a single left child.**

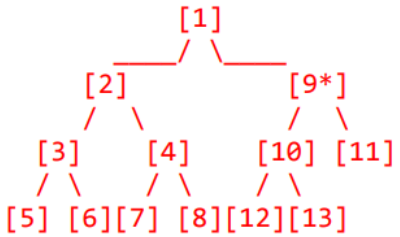
3. What is the size of the largest binary min heap in which the fifth largest element is a child of the root? You do not need to draw an example. **Assume the heap has no duplicate elements.**

During the exam, we posted a clarification such that we were interested in "size" as the number of elements. Based on this, the solution is **13**.

We know that the size of the subtree rooted at the fifth largest element is 5, including that element. This gives us the following subtree:



Given this information, the tree structure we get looks something like this:



Graph Traversals

BFS/DFS

Idea: We must look through all the values of our graph. So, given some starting point, we do a BFS/DFS traversal with the caveat that we now track what vertices we've visited before (to avoid cycles). Below is the code for DFS. For BFS, we need only replace that `Stack` fringe with a `Queue` fringe.

```
public void dfs() {
    Stack fringe = new Stack();
    Set visited = new Set();
    fringe.push(startVertex);
    while (!fringe.isEmpty()) {
        Vertex v = fringe.pop();
        if (!visited.contains(v)) {
            process(v); //Do something with v
            for (Vertex neighbor: v.neighbors) {
                fringe.push(neighbor);
            }
            visited.add(v);
        }
    }
}
```

Topological Sort

Idea: Given a directed, acyclic graph, how do we 'sort' the vertices based on their dependencies on one another?

```
public void topologicalSort() {
    Stack fringe = new Stack();
    Map currentInDegree = new Map<Vertex, Integer>();

    while (!fringe.isEmpty()) {
      Vertex v = fringe.pop();
        process(v); //Do something with v
        for (Vertex neighbor: v.neighbors) {
            currentInDegree(neighbor) -= 1; //Not actual Java code
            if (currentInDegree(neighbor) == 0) {
                fringe.push(neighbor);
            }
        }
    }

}
```

Dijkstra's Algorithm

Runtime: $O((|V|+|E|)\log(|V|))$ Important Assumption: When a vertex is removed from the fringe, we assume we have then found the shortest path to that vertex. Thus, we will no longer try to update the shortest path to that vertex; the path is finalized.

Pseudocode

- **Initializing data structures**
  1. Initialize the following structures:
     - Fringe
     - DistanceMap
     - PredecessorMap
  2. Add the start vertex to the fringe and the DistanceMap with distance 0
  3. For all other vertices, add them to the fringe and the DistanceMap with distance infinity
- **While-Loop (processing the shortest paths)**
  1. Pop off a vertex v from the fringe
  2. Loop over each neighbor n of v:
     (a) Let `newDistance` = `DistanceMap`[v] + edge(v, n)
     (b) If `newDistance` < `DistanceMap`[n], then
     (1) Update the priority value of n to be `newDistance`
     (2) Update the predecessor map so that the parent of n is v

Practice Problems

1. Consider a weighted, directed graph G with distinct and positive edge weights, a start vertex s and a target vertex t. Asssume that G contains at least 3 vertices and that every vertex is reachable form s.
   - (T/F) Any shortest $s \to t$ path must include the lightest edge.
     False. Consider an edge-weighted directed graph with these edges and weights: $s \to v$ (1), $v \to w$ (2), $w \to t$ (3), $s \to t$ (4). Then the shortest path is $s \to t$, which excludes the lightest edge.
   - (T/F) Any shortest $s \to t$ path must include the second lightest edge.
     False. In the example above, the shortest path excludes the second lightest edge.
   - (T/F) Any shortest $s \to t$ path must exclude the heaviest edge.
     False. In the example above, the shortest path includes the heaviest edge.
   - (T/F) The shortest $s \to t$ path is unique.
     False. Consider an edge-weighted directed graph with these edges and weights: $s \to v$ (1), $v \to t$ (2), $s \to t$

2. The runtime for Dijkstra's algorithm is $O((V + E) \log V)$. However, this is specific only to binary heaps. Let's provide a more general runtime bound. Suppose we have a priority queue backed by some arbitrary heap implementation. Given that this unknown heap contains N elements, suppose the runtime of remove-min is $f(N)$ and the runtime of change-priority is $g(N)$.

   (a) What is the runtime of Dijkstra's in terms of $f(V)$ and $g(V)$?
   In the worst case, we check every edge and decrease or increase the key of a node in the heap. In addition, we also always need to remove all vertices from the heap. The overall runtime is in $O(E * g(V) + V * f(V))$

   (b) Turns out the optimal version of Dijkstra's algorithm uses something called a fibonacci heap. The fibonacci heap has amortized $\Theta(1)$ change-priority, and $\Theta(\log(N))$ remove-min. What is the runtime of Dijkstra's algorithm?
   $O(E + V \log V)$

Sorting Algorithms

Comparison-Based Sorts

- These sorts compare elements to each other to determine their final ordering.
- Selection Sort
  - Keep a sorted and unsorted portion of the list, 'select' the minimum value from the unsorted portion and append it to the end of the sorted portion
  - Initialize the sorted portion to nothing and the unsorted portion to the entire list
  - Tips to identify: In intermediate steps, the front of the list will always be the ending sorted order of the elements
- Insertion Sort
  - Inversion: any two elements that are in the wrong place relative to each other
  - Insertion sort sorts elements by removing all inversions until the list is sorted in the end
  - Keep a sorted and unsorted portion of the list, for the first element of the unsorted portion, 'insert' it into the correct position in the sorted list by continuously comparing with the element before it and swapping if necessary
  - Initialize the sorted portion to nothing and the unsorted portion to the entire list
  - Tip to identify: In intermediate steps, front of the list will contain the items in sorted order, though it may not be the end sorted order (include example HERE)
- Merge Sort
  - Split the list in half, recursively call `mergeSort` on each half, then merge the halves together
- Quicksort
  - Choose a pivot element and partition the list into elements less than the pivot and elements greater than the pivot, recursively call `quickSort` on each partition
  - Pivot choice is key here!
- Heapsort
  - Throw all elements into a heap, remove the root to put everything in sorted order
  - Can use a min or max heap, though max heap will allow in-place sorting to occur
  - Can use bottom-up heapification to heapify the elements to begin with in linear time

Counting-Based Sorts

Radix Sort

- Sort the elements digit by digit based on the radix by running counting sort on each digit
- Variations
  - LSD (least significant digit):
    - run counting sort on the rightmost digit and move towards the left
    - Used more in practice, stable sort
  - MSD (most significant digit):
    - run counting sort on the leftmost digit, and repeat towards the right
    - Harder to implement, usually done recursively which can increase how many things are on the stack, unstable if done in place

Sorting Runtimes

Algorithm	Best-case Scenario	Best-case Runtime	Worst-case Scenario	Worst-case Runtime
Heap Sort	A list where all numbers are the same, and no bubbling is necessary.	$\Theta(n)$	Any list where we need to bubble down on all removals.	$\Theta(n \log n)$
Quicksort	Good pivot that is near the median of the list.	$\Theta(n \log n)$	Bad pivot that is an extreme value of the list.	$\Theta(n^2)$
Merge Sort	Always the same order of growth for a list of length n.	$\Theta(n \log n)$	Always the same order of growth for a list of length n.	$\Theta(n \log n)$
Selection Sort	Always the same order of growth for a list of length n.	$\Theta(n^2)$	Always the same order of growth for a list of length n.	$\Theta(n^2)$
Insertion Sort	Sorted list, where the number of inversions K is less than n	$\Theta(n)$	Reverse-sorted list, where $k = n^2$	$\Theta(n^2)$
LSD Radix Sort	LSD radix sort does the same amount of work in all cases.	$\Theta(D * (N + K))$	LSD radix sort does the same amount of work in all cases.	$\Theta(D * (N + K))$
MSD Radix Sort	If all elements start with different characters, one pass of counting sort will sort the elements.	$\Theta(N + K)$	Will have to run counting sort on all digits, like LSD radix sort.	$\Theta(D * (N + K))$

Practice Problems

1. Give an example of when insertion sort is more efficient than merge sort.
   When the input is small, almost sorted, or has few inversions.

2. When would you use merge sort over quicksort?
   When you require stability. Merge sort is a stable sort, but quicksort is not guaranteed to be stable.

3. When would you use radix sort over a comparison sort?
   When the alphabet is well-defined and each object can be individually compared on each radix.

4. You have an array of integers with a length of N, where the maximum number of digits is given by K. Suppose that $K >> N$ (that is, K is much larger than N). What sort could you use?
   Quicksort since the input is an array of integers where stability is not a concern.

5. Assume for the previous question that you used LSD radix sort. What is the runtime?
   Because $K >> N$, the runtime will be in $O(R(N+K))$ which is in $O(K)$ for this problem.

6. Suppose you are given an integer array with length N such that there are $N - \sqrt{N}$ copies of the minimum element located in the front of the array. What is the worst case runtime of insertion sort on this list?
   The total number of inversions can be given by the arithmetic sum to $\sqrt{N}$ which is in $\Theta(N)$. Insertion sort on the remaining elements takes $\Theta(N)$ time to resolve each inversion. The total runtime is in $\Theta(N+N)$ or $\Theta(N)$.

7. For the same construction, what is the worst case runtime of selection sort?
   Still $\Theta(N^2)$ because we need to find the minimum element in each iteration.

Disjoint Sets

Disjoint sets are used to determine what sets, or groups, elements belong in to. Our main goal with implementing a disjoint set is to be able to check whether or not two elements are in the same set, and to be able to merge two sets together. You can also extend this feature to checking the connectivity of two elements (think of Kruskal's algorithm, for instance). Thus, there are three operations we want our disjoint set to support: `MakeSet`, `Find`, and `Union`.

Weighted Quick-Union trees are a way to implement disjoint sets. In WQU trees, we will represent sets as a tree-like structure. each element is itself a node, that may point to another node. The root of any tree acts as the 'representative' of the set. Let us examine how the disjoint set operations are implemented in WQU trees:

- `Find(n)`
  1. Find and return the root node of n
  2. **Path Compression** optimization:
     As you follow the parent pointers from n to the root, reassign the parent pointers of n and all other nodes you see along the way to be the root.
     Purpose: Prevent our tree from getting too tall.
- `Union(a, b)`
  Goal: merge the set of a with the set of b
  1. Let `root1 = find(a)`
  2. Let `root2 = find(b)`
  3. **Union by size** optimization:
     If `root2` has a smaller set size than `root1`, then set the parent pointer of `root2` to point to `root1`.
     Else, set the parent pointer of `root1` to point to `root2`.

Notice that both of the optimizations introduced have the goal of preventing the tree from getting too tall.

Runtime (including both optimizations):

- One call to `find`:
  Worst case $\Theta(\log(n))$
  Best case $\Theta(1)$
- One call to `Union`:
  Worst case $\Theta(\log(n))$
  Best case $\Theta(1)$
- For f calls to `find` and u calls to `union`:
  Essentially $\Theta(u + f * \alpha(f + u, u))$

Where α is the inverse Ackerman function, a function that grows veeeeeeeeeeeeerrrrrrrrrryyyyyyyyyy slowly.

Minimum Spanning Trees

A *spanning tree* of a graph is a tree that connects all vertices of the graph. We define the weight of a spanning tree as the sum of the weights of all edges in the tree. A **minimum spanning tree** is a spanning tree of a graph that has the least weight. Note that any tree that connects V vertices will have $V - 1$ edges.

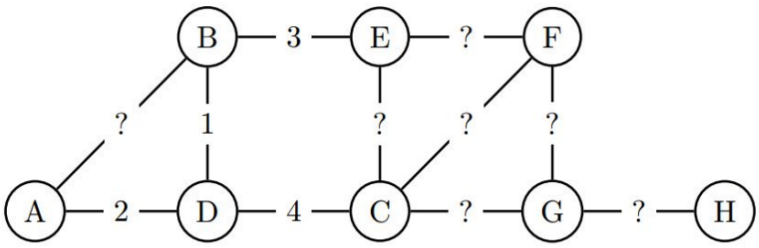
The main idea of Kruskal's algorithm is to successively select edges with the smallest weight to be in our tree, until a MST is formed. Thus, given a graph G, Kruskal's algorithm works as follows:

1. Let T be an empty list of edges. This is where you will store the edges for your final MST.
2. Make a sorted list `SortedEdges` of all the edges in G, from smallest to greatest.
3. For each edge (`u->w`) in `SortedEdges`
   (a) If `u` and `w` are not already connected by the edges in T, then add the edge (`u->w`) to T.
   (b) If `u` and `v` are already connected by the edges in T, then continue.

We can see that the runtime of Kruskal's algorithm is dependent on the sorting step in (2), and the check in (3a/b) to see if two vertices are already connected by T. The sorting step in (2) takes $O(E \log E)$. By using WQU trees with path compression to accomplish steps (3a/b), a single check to see if two vertices are already connected by T will take time proportional to $\alpha(V)$. Since this factor is extremely small, the total runtime of Kruskal's ends up being $O(E \log E)$ due to the sorting step.

Practice Problems

1. In the graph below, some of the edge weights are known, while the rest are unknown. List all edges that must belong to a minimum spanning tree, regardless of what the unknown edge weights turn out to be. Justify your answers briefly.



$$cost(A, D) = 2, cost(B, D) = 1, cost(C, D) = 4, cost(B, E) = 3$$

(G, H): This is the only edge that can connect H to any other vertex, so it must be included in any MST. Remember that any MST must span all of the vertices.

(B, D): The cut property. This is the minimum weight edge between D and the rest of the vertices in the graph.

(B, E): Also by the cut property. This is the minimum weight edge between the ABD tree and the rest of the vertices in the graph.

2. Given a minimum spanning tree T of a weighted graph G, describe an $O(V)$ algorithm for determining whether or not T remains a MST after an edge (x, y) of weight w is added.
By the tree property, consider the (one and only) path between x and y. For each edge on that path, if it's the case that w is less than the weight of any one edge on that path, then the MST will change.

3. True or False: Suppose G is a graph and T is a MST of G. If we decrease the weight of any edge in T, then T is still a MST.
Yes. We know that if T is an MST of G with total weight W, then it must be the case that all other valid MSTs of the graph now have total weight $W -$ `weight(edge)` just like T.

4. Suppose instead of decreasing the weight of an edge in the MST, we decrease the weight of any random edge not in the MST. Give an efficient algorithm for finding the MST in the modified graph.
Remember that MSTs are still trees! There is only one unique path between any two vertices.

Suppose an edge (u, v) chosen at random is decreased. First, find a path from u to v in the MST. Then, add the edge (u, v), creating a cycle. Traverse the cycle and remove the edge with maximum weight. This will break the cycle and result in the new MST.

5. Consider a weighted, undirected graph G with distinct and positive edge weights. Assume that G contains at least 3 vertices and is connected.

   - (T/F) Any MST must include the lightest edge.
     True. Apply the cut property to either endpoint of the lightest edge.
   - (T/F) Any MST must include the second lightest edge.
     True. Let $e = v - w$ be the lightest edge and $f = x - y$ be the second lightest edge. Apply the cut property to either the vertex x or y (pick x or y so that it is different from both v and w).
   - (T/F) Any MST must exclude the heaviest edge.
     False. Consider an edge-weighted graph with these edges and weights: $v - w$ (1), $w - x$ (2), $x - v$ (3), $x - y$ (4). Then, the unique MST contains the heaviest edge $x - y$.
   - (T/F) The MST is unique.
     True. As asserted in lecture/textbook, if the edge weights are distinct, then the MST is unique.

Regular Expressions

- Special Characters
  - + matches one or more of the preceding character (greedy operator)
  - * matches zero or more of the preceding character (greedy operator)
  - ? matches zero or one of the preceding character (greedy operator), ? after a greedy operator will make the operator reluctant/lazy
  - {n} matches exactly n instances of the preceding character
  - {n, m} matches at least n but no more than m instances of the preceding character
  - . matches any character
  - [...] denotes a character class, matches one character that is described within the brackets
  - (...) denotes a capturing group, can be used to group characters together
  - To match special characters (strings including periods or question marks), you must escape them with a backslash
- `Pattern`
  - `Pattern Pattern.compile(String s)`: returns a `Pattern` that is represented by `s`
  - `Matcher matcher(String phraseToMatch)`: returns a `Matcher` that matches the `Pattern`
- `Matcher`
  - `boolean matches()`: returns true if `Matcher`'s `Pattern` matches the given phrase to match entirely
  - `boolean find()`: returns true if `Matcher`'s `Pattern` matches a substring of the given phrase (will only match up to the substring, subsequent calls to `find()` may result in more matches of the same string)
  - `String group(int index)`: returns the `String` corresponding to the group indicated by `index`, `index` 0 will correspond to the entire matched phrase
- Greedy vs. Reluctant
  - Some operators (+, *, ?) will match as much as possible
  - Want to match the 'lo000l' in 'lo000l helo haha', if our pattern is 'l.*l', will actually match 'lo000l hel' because the * operator will match as much as possible
  - Use the ? operator after a greedy operator to make it reluctant (match as little as possible while still following the pattern described)
- Capturing
  - Can use (...) and `group` method to return certain parts of a matched substring
  - `group(1)` will refer to the first grouped elements in the String, same idea with 2, 3, etc.

Practice Problems

1. Write the regex that matches any string that has an odd number of a's and ends with b.

   ```
   ^[^a]*a([^a]*a[^a]*a)*[^a]*b$
   ```

2. Write a regex that matches any binary strin gof length exactly 8, 16, 32, or 64.

   ```
   ^(0|1){64}|(0|1){32}|(0|1){16}|(0|1){8}$
   ```

3. Write the regex that matches any string with the word "banana" in it.

   ```
   ^.*banana.*$
   ```

4. A certain type of bracketed list consists of words composed of one or more lower-case letters alternating with unsigned decimal numerals and separated with commas:

   ```
   []
   [wolf, 12, cat, 41]
   [dog, 9001, cat]
   ```

   As illustrated, commas may be followed (but not preceded) by blanks, and the list may have odd length (ending in a word with no following numeral.) There are no spaces around the `[ ]` braces. Write the **Java pattern** that matches an alternating list.

   ```
   \[(([a-z]+, *\d+, *)*[a-z]+(, *[0-9]+)?)?\]
   ```