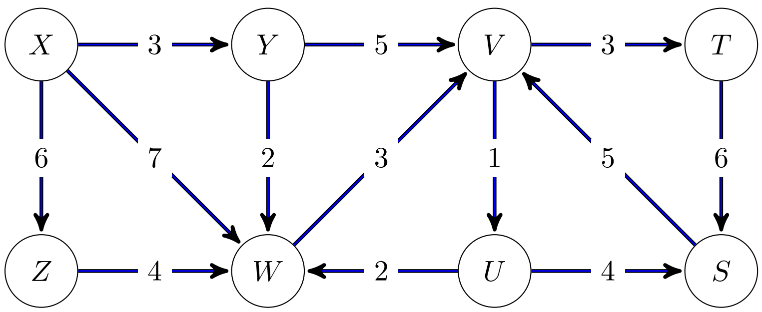
CMPSC 465	Data Structures & Algorithms	
Fall 2024	Mingfu Shao and Debarati Das	HW 5

0. (0 pts.) Acknowledgements. List any resources besides the course material that you consulted in order to solve the assignment problems. If you did not consult anything, write "I did not consult any non-class materials." The assignment will receive a 0 if this question is not answered.

1. (8 pts.) Run Dijkstra's algorithm on the following graph, starting at node X. Whenever there is a choice of vertices with the same *dist* value, always pick the one that is alphabetically first. Please draw a table where each row shows the *dist* array at each iteration of the algorithm.



Solution:

Iteration	X	Y	Z	W	V	U	T	S
0	0	∞	∞	∞	∞	∞	∞	∞
1	0	3	6	7	∞	∞	∞	∞
2	0	3	6	5	8	∞	∞	∞
3	0	3	6	5	8	∞	∞	∞
4	0	3	6	5	8	∞	∞	∞
5	0	3	6	5	8	9	11	∞
6	0	3	6	5	8	9	11	13
7	0	3	6	5	8	9	11	13
8	0	3	6	5	8	9	11	13

2. (12 pts.) Let's see an application of (binary) heap. You are given an array A with n integers, and another integer k, $1 \le k \le n$. The numbers in A are either -1 or a positive integer, and you may assume that all positive integers are distinct (but there could be multiple -1s). You are asked to design data structures and algorithm to produce an output array X. Your algorithm should process the numbers in A one by one: when a positive number is met, you put it to the end of X; if a -1 is met, you need to remove the k-th smallest number in X. You may also assume that, you will never meet a -1 if the size of the current X is smaller than k. For example, if $A = [4, 2, -1, 7, 3, 8, -1, 6]$ and $k = 2$, the output X should be $X = [2, 7, 8, 6]$.

   Design an algorithm to complete this task and analyze its running time. Your algorithm should run in $O(n \log n)$ time. (Hint: consider using two binary-heaps, one max-heap and one min-heap; the max-heap maintains the smallest k numbers in X and the min-heap maintains rest of the numbers.)

   Solution:

   We use a max-heap to maintain the smallest k numbers and a min-heap to maintain the rest of the numbers. This ensures that the root of the max-heap is the k-th smallest number, and the root of the min-heap is the $(k+1)$-th smallest number. To ensure that we are able to reconstruct X, each element added to any of the heap is a (key, value) pair, where key is the number recieved, i.e., $A[i]$, and the value is the index in A, i.e., i.

When we receive $A[i] = -1$, we delete the root of the max-heap. Then the previous $(k+1)$-th smallest number (the root of the min-heap) now becomes the new k-th smallest number, so we delete the root of the min-heap and add it into the max-heap. When we receive $A[i] > 0$, we need to compare it with the key of the root of the max-heap. If it is larger, we insert it into the min-heap. Otherwise, we add it into the max-heap; now, since there are $k+1$ numbers in the max-heap, we move the root of the max-heap (the new $(k+1)$-th smallest number) to the min-heap.

At the end, to reconstruct X, we collect all elements (pairs) in both the min-heap and max-heap in a new array Y. We then sort Y according to their positions in A, i.e., according to the value field stored in the pair. The resulting Y can be transcribed into X by discarding the value field.

```
Algorithm Process(A, k)
    Initialize maxHeap and minHeap
    For i = 1 → |A|
        If A[i] = −1
            // Received -1, delete the root of the max heap
            maxHeap.delete-max()
            // There are only k − 1 numbers in max heap, so move the root of min heap to max heap
            maxHeap.insert(minHeap.find-min())
            minHeap.delete-min()
        Else
            If maxHeap.empty() or A[i] < maxHeap.find-max().key
                // Insert it into max heap if it's one of the k smallest numbers
                maxHeap.insert(A[i], i)
                // Now there are k + 1 numbers in max heap, so move the root of max heap to min heap
                If maxHeap.size() > k
                    minHeap.insert(maxHeap.find-max())
                    maxHeap.delete-max()
                End if
            Else
                // Otherwise, insert it into min heap
                minHeap.insert(A[i], i)
            End if
        End if
    End for
    Collect elements (i.e., pairs) in maxHeap and minHeap in an array Y
    Sort Y according to the value field of the pair (i.e., positions in A)
    Transcribe Y into X sequentially by just keeping the keys (i.e., discarding positions)
    Return X
End algorithm
```

The pseudocode is given above. We use a function heap.size() which returns the number of stored elements in a heap. We did not implement this one in class but it is straightforward and can be done in $\Theta(1)$ time.

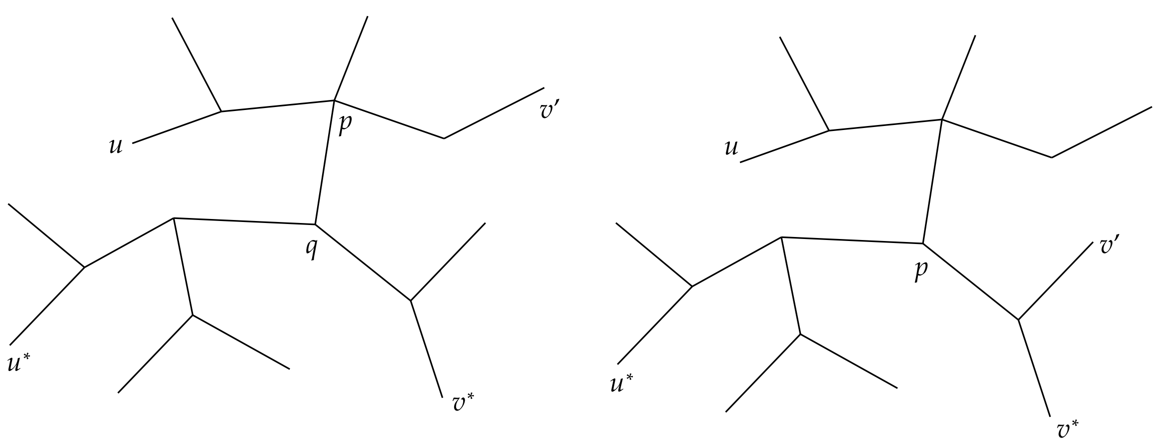
Time complexity: It is obvious that, to process each $A[i]$, the algorithm calls a constant number of heap operations. Each operation either takes $\Theta(1)$ or $O(\log n)$, assuming binary-heap is used. Hence, this part takes $O(n \log n)$ time. The sorting Y certainly can be done in $O(n \log n)$ time. Thus, the overall algorithm runs in $O(n \log n)$ time.

3. (10 + 5 bonus pts.) A tree is defined as a connected, undirected graph without any cycle. It is obvious that there is a single, unique path between any two vertices in a tree. Let $T = (V, E)$ be a tree and let $\delta(u, v)$ be the length of the path between u and v. Here we assume unit edge length, i.e., $\delta(u, v)$ is the number of edges in the path between u and v. We aim to design an algorithm to find the length of the longest path in T, i.e., $\max_{u,v \in V} \delta(u, v)$. If (u', v') reaches this maximum, i.e., $\delta(u', v') = \max_{u,v \in V} \delta(u, v)$, we say (u', v') forms a furthest pair. Note that T may contain multiple furthest pairs.

   a. (5 bonus pts) For any vertex $u \in V$, we say v' is the furthest vertex to u, if $\delta(u, v') = \max_{v \in V} \delta(u, v)$. Prove that, for any vertex $u \in V$ with v' being (any one of) its furthest vertex, there must exist vertex u' such that (u', v') is one furthest pair of T.

   b. (10 pts) Given T, design an algorithm to find a furthest pair and analyze its running time. Your algorithm should run in $O(|V| + |E|)$ time. (Hint: consider using the results of part (a) and BFS.)

Solution: First of all, a useful and simple fact is that there is a unique path from any two vertices in a tree. We use $L(u, v)$ to denote the length (number of edges) in the unique path from u to v.



   a. We prove the statement by contradiction. Let v' be one furthest vertex to u. Suppose conversely that v' is not one end of any furthest pair of T. Let (u^*, v^*) be one furthest pair of T. Consider two cases (see the figure). The first case is that u-v' path does not overlap with the u^*-v^* path. These two paths must be connected; let p-q path be the "bridging" path, where p is on the u-v' path and q is on the u^*-v^* path. If $L(p, v') \le L(q, v^*)$, then the path u-p-q-v^* is longer than u-v', contradicting to that v' is the furthest vertex to u. If $L(p, v') > L(q, v^*)$, then the path u^*-p-q-v' is longer than u^*-v^*, contradicting to that u^*-v^* is one furthest pair in T. Now consider the second case that the u-v' path overlaps with the u^*-v^* path. Let p be the first vertex on the u^*-v^* path following the path from u to v'. We must have that $L(p, v') \geq L(p, u^*)$ and $L(p, v') \geq L(p, v^*)$ since v' is the furthest vertex to u. This implies that either u^*-v' path or v^*-v' path must be also one longest path in T, contradicting to the assumption that v' is not one end of any furthest pair of T.

b. Part (a) immediately implies a simple algorithm. Run BFS from an arbitrary vertex $u \in V$. Let u' be vertex with the largest distance from u. Then run BFS again starting from u' to find a vertex, denoted as v', with the largest distance from u'. The $(u', v'$ is one furthest pair. This algorithm runs in $O(|V| + |E|$ time. (In fact, $|V| = |E| + 1$ in a tree.)