Prac 10 Documentation

by Pavin Wu (26916592)

Task 1

Min Heap implementations: use an array implementation but instead of storing a value, we store a tuple of vertex: (*ID, total distance, path*). We then use *total distance* to sort the vertex based on Min Heap structure. We also update position of each vertex in the heap every time a vertex is moved i.e. deleted or inserted.

Find Camera function: similar to original Dijkstra's algorithm with certain modifications:

1. Being able to check if a vertex is a vertex with camera. This means we can terminate function beforehand if user's vertex has the camera. Otherwise, we can also use this when finding path with Dijkstra algorithm to check if current vertex (having its breadth searched) has camera. If so, we stop searching through current vertex and append current vertex into list of cameras found.
2. Use Min Heap instead of an array to store discovered vertex. This is so access time to find vertex with minimum distance is O(1). Since we also need to update distance and path to a breadth vertex if we found a shorter path to it through current vertex, we store index of each vertex in the heap in an array. The index in the array that we store index of each vertex in the heap is the ID of the intersection. Also:
    a. Since path length is strictly non-negative, we also use the same array to store -1 for a vertex if it is finalised or -2 if it is not discovered.
    b. When updating path of the breadth vertex, we simply need to take the path to the current vertex, then append it with the ID of breadth vertex. Similarly, with distance, we simply need to take the distance of current vertex, then add it with the distance from current vertex to the breadth vertex. Note that we also need to up heap this updated vertex.
3. Instead of continue looking for next breadth vertex right away, we check first if the path we are checking is a TOLL, or if the breadth vertex is the terminal (i.e. no other vertices connected to it), then we skip current path.
4. When finished, return the list of camera found. We use this to print the cameras and their info later in the main process.

Worst-case time: Reading files takes O(V+E). Heap operations are O(logV) each. For loop in Dijkstra's algorithm runs a total of O(ElogV). The parent while loop, when consider separately, runs in O(VlogV). So total complexity of Dijkstra's algorithm is O(ElogV + VlogV). However, there's an additional O(V) worst case time for printing the path for each camera (path may not be as large as V, but V is always upper bound). Since there are k cameras, the cost is O(kV). Therefore total worst-case time complexity is O(kV + ElogV + VlogV).

Worst-case space: the heap is as big as O(V) in worst case (for a dense network) where V is the total number of vertices. Array of index for each vertex in heap is also O(V), as well as array containing indices with cameras. The list containing camera found from the algorithm is O(k) where k is total number of camera, but k <= V in any cases, so it's also O(V). Adjacency list is O(V+E) where E is the total number of edges. Other objects have O(1) complexity. So overall worst case space is O(V+E).

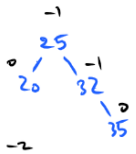
Task 2a

Start by inserting like a normal binary search tree.

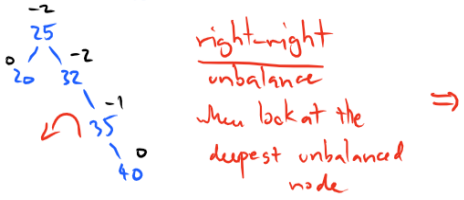
Balance factors are in black. The node with balance factor >1 or <-1 is AVL unbalanced and need adjustments.

Insert 20, 25, 32

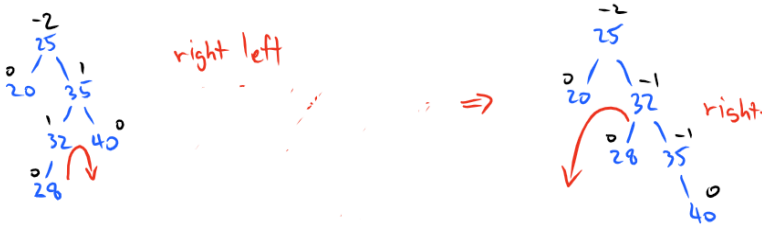
Insert 35

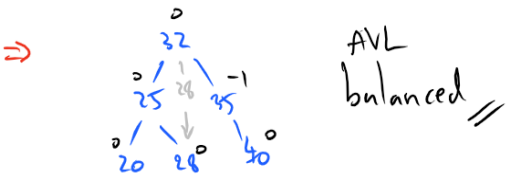


Insert 40



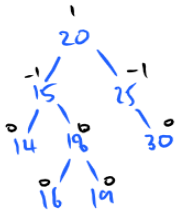
Insert 28



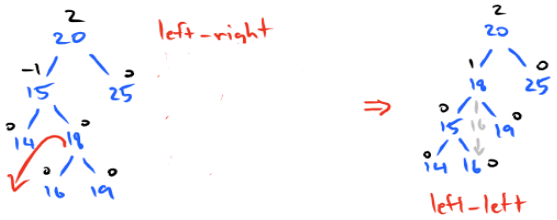


b

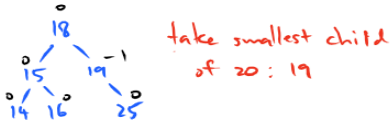
We have



Delete 30 :



Delete 20 :



Delete 25 :