Network Routing

Noah Schill CS312

> For each operation (*insert*, *delete-min*, and *decrease-key*) convince us (refer to your included code) that the complexity is what is required here.

Array

My array implementation uses Python's built-in timsort algorithm. This has a time complexity of $O(n \log n)$ as per documentation.

Insert takes $O(n \log n)$ time. While appending an item to the list is constant, the array is sorted using timsort after, which bottlenecks this process.

Delete-min in my implementation is $O(1)$, as my sorting is already done upon insertion and this merely needs to pop the 0th value of my list.

Decrease-key takes $O(n)$ time. It simply searches through my array until it finds the desired value, and then replaces it with a new one. In the worst-case scenario, it would have to go through n items to find the desired node.

Heap

Insert takes $O(\log n)$ in a worst-case scenario as it would need to descend to the lowest nodes of the binary tree, which has height of $\log n$, to append a large value.

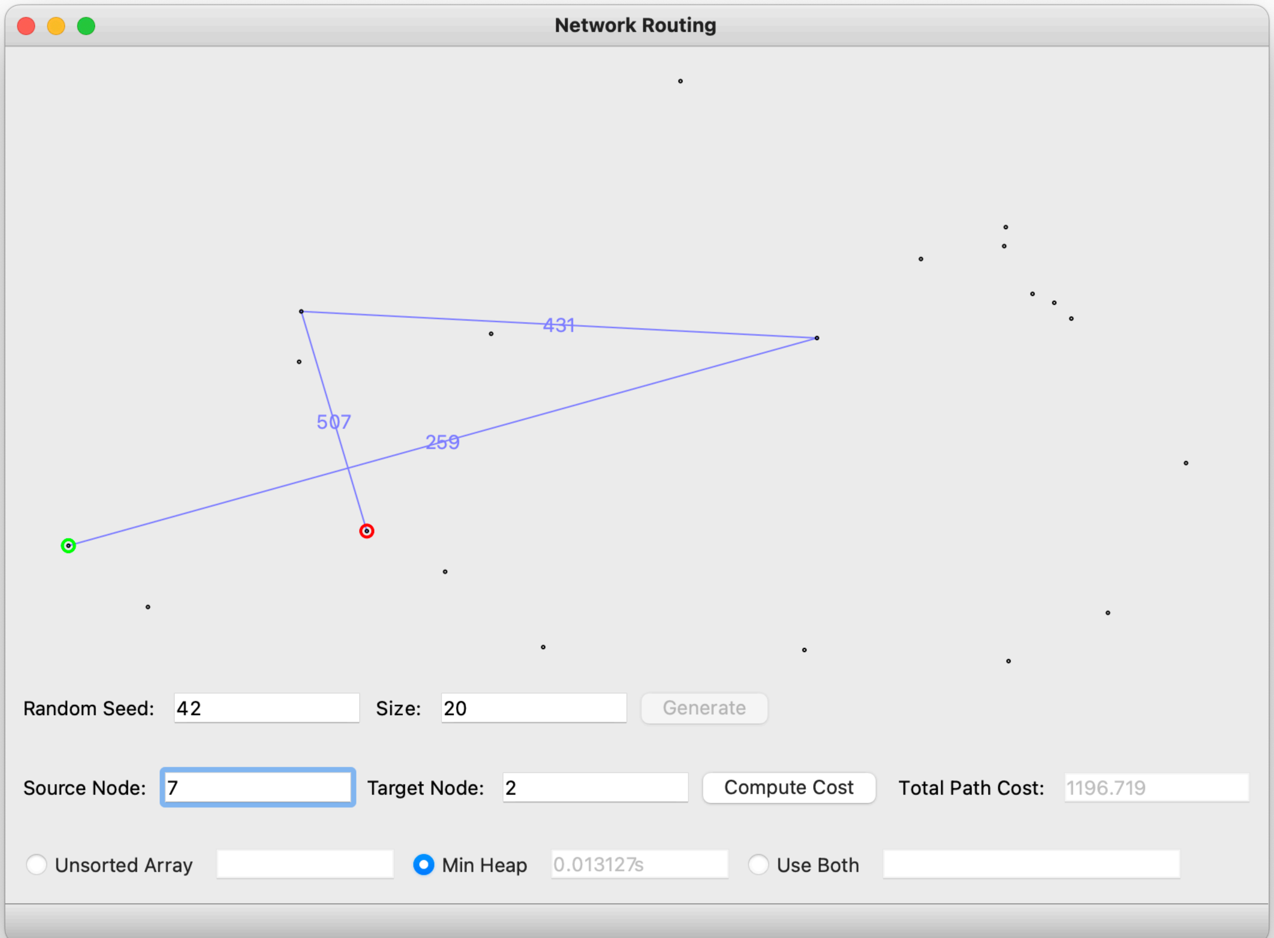
Delete-min in the heap is $O(\log n)$. Returning the minimum itself is only $O(1)$ as the root of any minheap is the minimum value by definition, but then the tree has to be "heapified" to maintain it's nature. This heapify function (as seen in my code) takes $O(\log n)$ if a value has to be sifted down the entire length of the tree because of this.

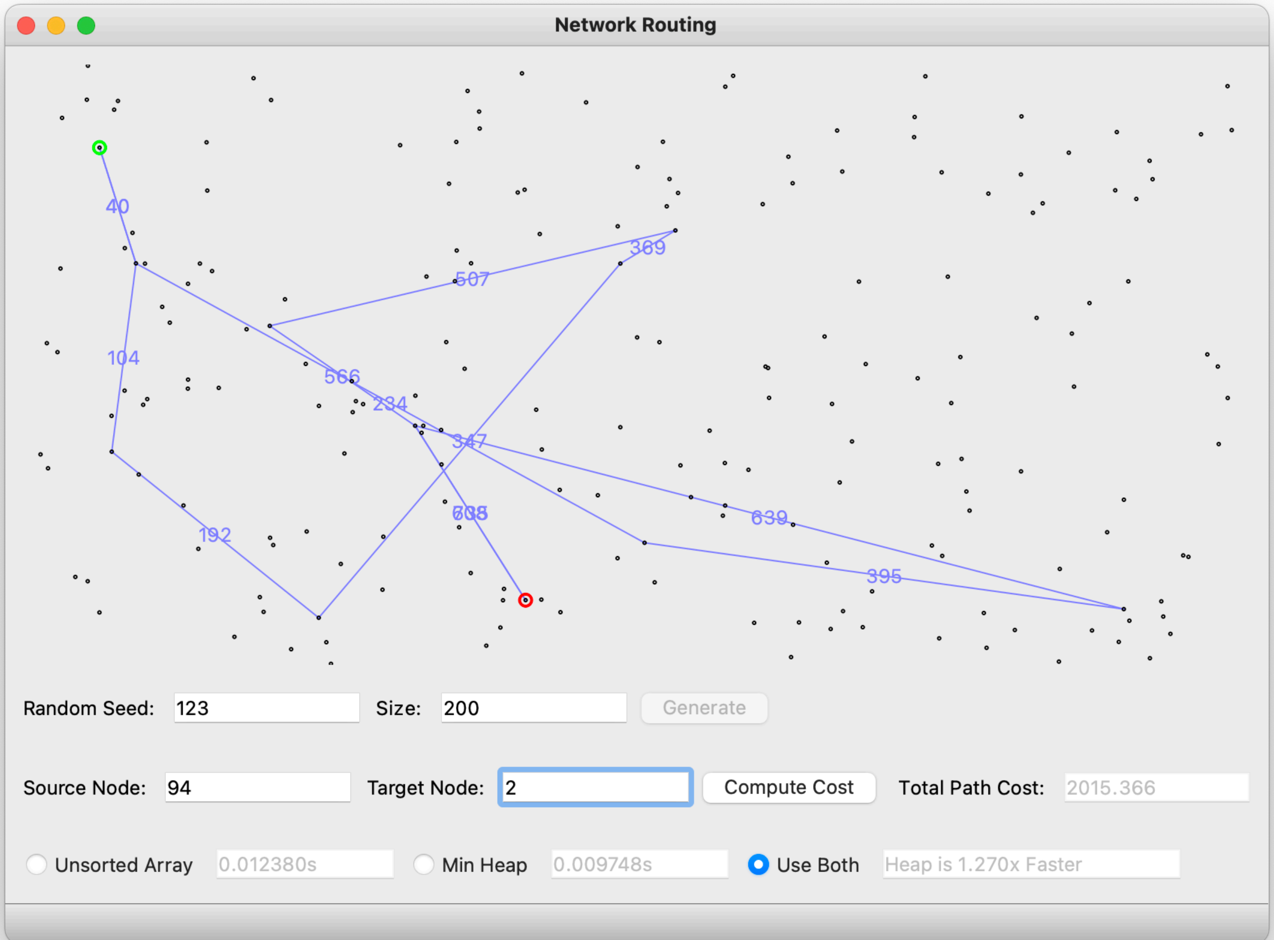
Decrease-key also takes $O(\log n)$ in a worst-case scenario if the value I want to change is at the bottom of the binary tree.

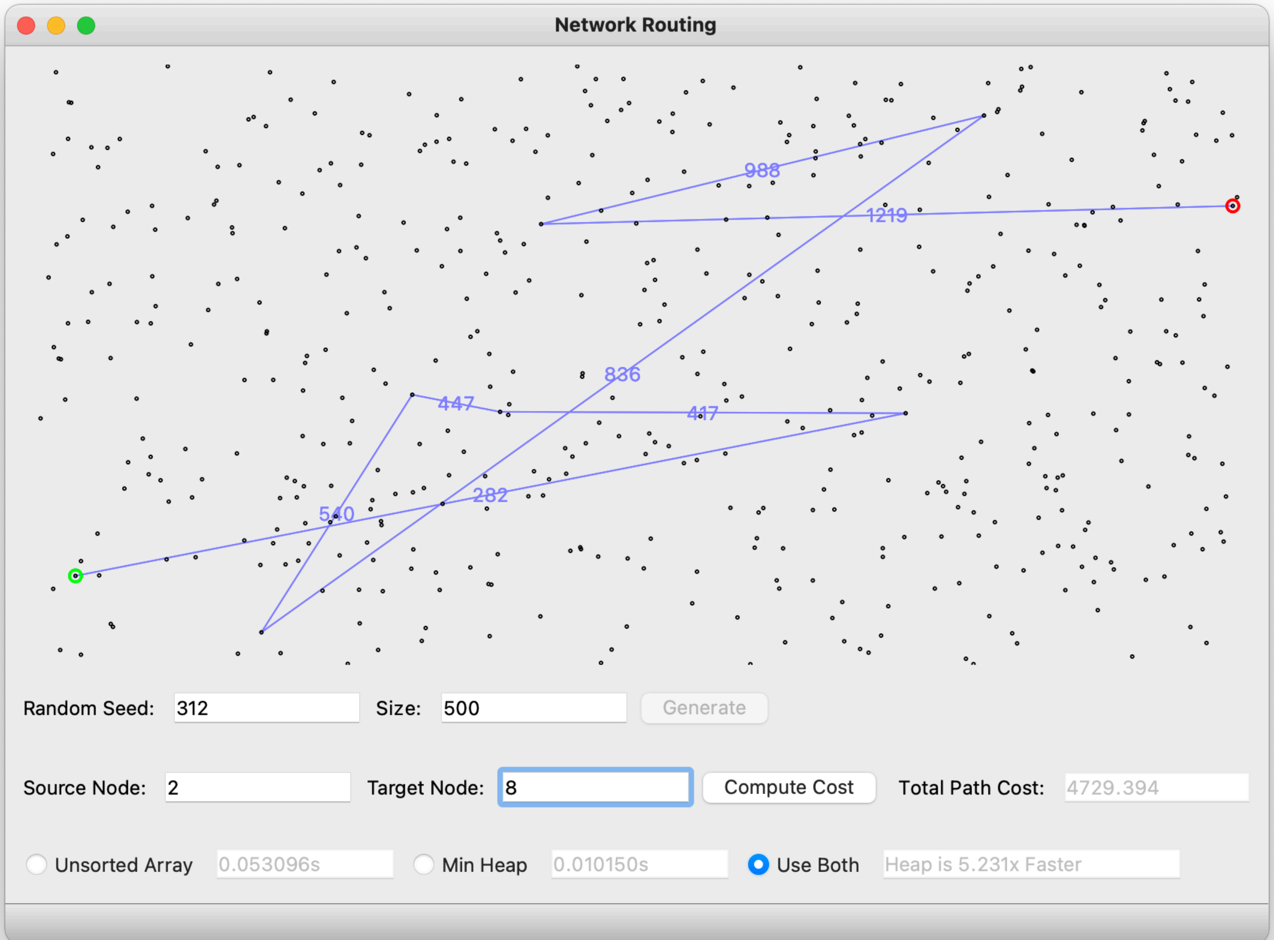
Space complexity analysis

Dijsktra itself consists of two main data structures aside from the graph it analyses. One is a table for every node with the distance and previous node. For this table, the space complexity is $O(3n) = O(n)$ for n nodes. The second structure is a priority queue, which is implemented as both an unsorted list, and a minimum heap. The unsorted list is fairly straightforward, as there are only as many elements in the list as there are nodes, so the space comlexity for the list is $O(n)$ for n nodes. For the heap, I have to assign empty values depending on the maximum size of my expected heap, fixing the actual storage at a constant value. If the storage complexity is only analysing based on how many of those elements in my implementation array are populated, this would also be $O(n)$ for n nodes as those values are only populated as I insert nodes.

> For Random seed 42 - Size 20, Random Seed 123 - Size 200 and Random Seed 312 - Size 500, submit a screenshot showing the shortest path (if one exists) for each of the three source-destination pairs, as shown in the images below.




Network Routing
40
369
507
104
566
234
347
639
192
395
Random Seed: 123
Size: 200
Generate
Source Node: 94
Target Node: 2
Compute Cost
Total Path Cost: 2015.366
Unsorted Array 0.012380s
Min Heap 0.009748s
Use Both
Heap is 1.270x Faster



Empirical Analysis

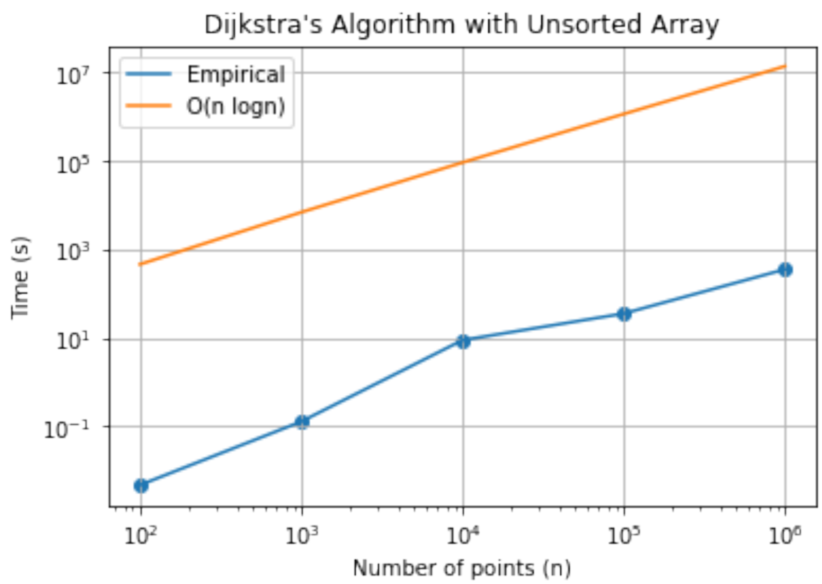
Predicting the array implementation for large values:

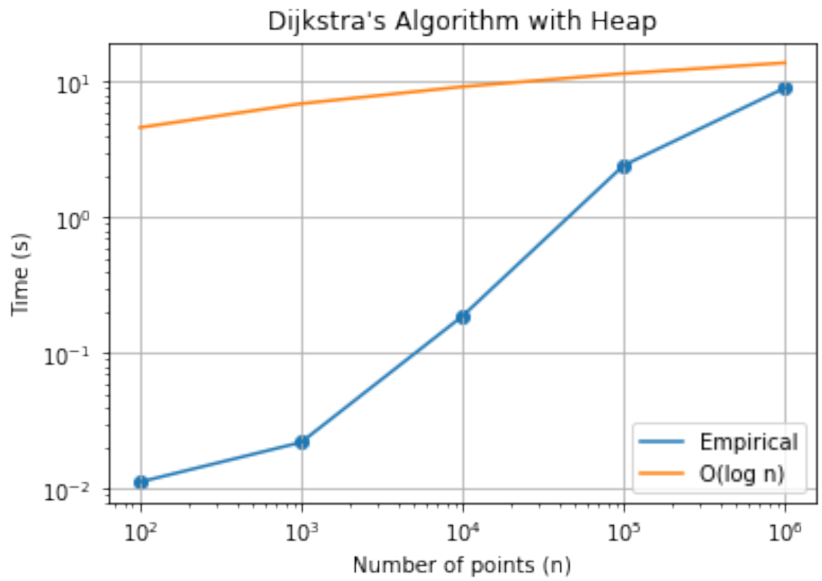
For the array implementation emperical data, I calculated a constant of proportionality to be $k = 0.000355232$. If $n = 1000000, n \times k = 355.232$

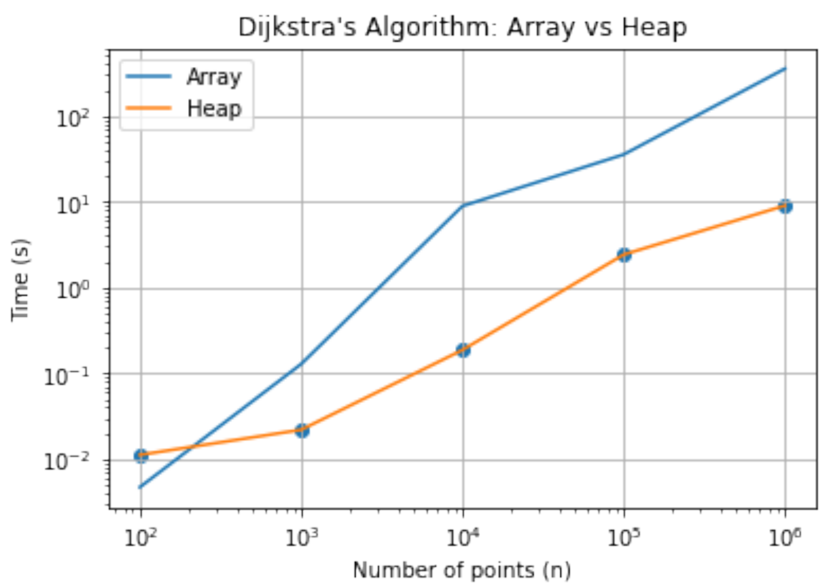
Appending this to my table of results:

Results:

	n	array_time	heap_time
0	100	0.004721	0.011219
1	1000	0.128458	0.021994
2	10000	8.900305	0.187346
3	100000	35.523200	2.421670
4	1000000	355.232000	9.019000







These results make sense. The fact that the graphs don't match perfectly illustrate the nature of Big O notation, that it estimates the worst-case scenario. It is interesting how with smaller n-values, the array is actually faster than the heap. This is probably due to the insertion time taking much longer than appending lists to arrays for small values.

Code:

```
#!/usr/bin/python3
import math

from CS312Graph import *
```

```
import time


class NetworkRoutingSolver:
    def __init__(self):
        self.dijsktra_result = {}

    def initializeNetwork(self, network):
        assert (type(network) == CS312Graph)
        self.network = network

    def getShortestPath(self, destIndex):
        self.dest = destIndex
        dest_node = self.network.nodes[destIndex]
        # TODO: RETURN THE SHORTEST PATH FOR destIndex
        #       INSTEAD OF THE DUMMY SET OF EDGES BELOW
        #       IT'S JUST AN EXAMPLE OF THE FORMAT YOU'LL
        #       NEED TO USE
        total_length = 0
        path_edges = []
        # node = self.network.nodes[self.source]
        # edges_left = 3
        # while edges_left > 0:
        #     edge = node.neighbors[2]
        #     path_edges.append((edge.src.loc, edge.dest.loc,
'{:.0f}'.format(edge.length)))
        #     total_length += edge.length
        #     node = edge.dest
        #     edges_left -= 1
        # Assemble edges
        p = dest_node
        while p != None:
            prev = self.dijsktra_result[p][1]
            if p.node_id != self.source and p is not None:
                l = self.dijsktra_result[p][0]
                total_length += l
                path_edges.append((p.loc, prev.loc, '{:.0f}'.format(l)))
            p = prev
        return {'cost': total_length, 'path': path_edges}

    def computeShortestPaths(self, srcIndex, use_heap=False):
        self.source = srcIndex
        src = self.network.nodes[srcIndex]
        t1 = time.time()
        # TODO: RUN DIJKSTRA'S TO DETERMINE SHORTEST PATHS.
        #       ALSO, STORE THE RESULTS FOR THE SUBSEQUENT
        #       CALL TO getShortestPath(dest_index)
```

```
        #       CALL TO getShortestPath(dest_index)
        self.dijkstra(src, use_heap)
        t2 = time.time()
        return (t2 - t1)

    def dijkstra(self, src, use_heap):
        # Create queue
        if use_heap:
            H = PriorityHeap()
        else:
            H = PriorityQueue()

        nodes = self.network.nodes

        the_table = {src: (0, None)}  # Format is Node:(Distance, Previous)
        dists = []  # To be passed to the heap
        dists.append((src, 0))

        for node in nodes:  # O(n)
            if node is not src:
                the_table[node] = (math.inf, None)
                dists.append((node, math.inf))

        H.make_queue(dists)  # O(n log n) for array, O(log n) for heap if it needs to be
swapped

        while not H.is_empty():
            u, u_l = H.delete_min()  # O(1) for array as it's already sorted,
            for neighbor_edge in u.neighbors:  # O(neighbors) for any node
                if the_table[neighbor_edge.dest][0] > the_table[u][0] +
neighbor_edge.length:
                    the_table[neighbor_edge.dest] = (the_table[u][0] +
neighbor_edge.length, u)
                    H.decrease_key(neighbor_edge.dest, the_table[u][0] +
neighbor_edge.length)

        self.dijsktra_result = the_table


class PriorityQueue:
    def __init__(self):
        self.array = []

    def __str__(self):
        print(__name__ + " : " + self.array)

    def make_queue(self, array_of_tuples):  # O(n log n)
```

```
    def make_queue(self, array_of_tuples):  # O(n log n)
        self.array = array_of_tuples.copy()
        self.array.sort(key=lambda x: x[1])  # Timsort is O(n log n)

    def insert(self, node, distance):  # O(n log n)
        self.array.append((node, distance))
        self.array.sort(key=lambda x: x[1])  # Timsort is O(n log n)

    def delete_min(self):
        return self.array.pop(0)  # O(1)

    def decrease_key(self, node_name, new_value):  # O(n)
        for tup in self.array:
            if tup[0] == node_name:
                new_tuple = (tup[0], new_value)
                self.array.remove(tup)
                self.insert(new_tuple[0], new_tuple[1])

    def length(self):
        return len(self.array)

    def is_empty(self):
        return len(self.array) == 0


class PriorityHeap:
    def __init__(self):
        # makequeue happens here
        self.maxsize = 1000001
        self.size = 0
        self.map = {}
        self.array = [(None, -math.inf)] * self.maxsize
        self.array[0] = (None, -math.inf)  # A zero will screw up indexing

    def is_empty(self):
        return self.size == 0

    def parent(self, i):
        return i // 2

    def left_child(self, i):
        return 2 * i

    def right_child(self, i):
        return (2 * i) + 1


    def swap(self, i, j):  # O(1)
```

```
    def swap(self, i, j):  # O(1)
        self.array[i], self.array[j] = self.array[j], self.array[i]

    def is_leaf(self, i):
        if (self.size // 2) <= i <= len(self.array):
            return True
        return False

    def pop(self): # O(log n)
        if len(self.array) <= 1:
            raise ValueError("No items in heap")

        popped = self.array[1]
        self.array[1] = self.array[self.size]
        self.heapify(1) #O(log n)
        self.size -= 1
        return popped

    def peek(self):
        return self.array[1]

    def heapify(self,
                i):  # O(log n), as the tree height is O(log n) and at worst will need to
be bubbled up the entire height
        if not self.is_leaf(i):
            if self.array[i][1] > self.array[self.left_child(i)][1] or self.array[i][1] >
\
                    self.array[self.right_child(i)][1]:

                # left side recursive call
                if self.array[self.left_child(i)][1] < self.array[self.right_child(i)][1]:
 # left child swap
                    self.swap(i, self.left_child(i))
                    self.heapify(self.left_child(i))

                # right side recursive call
                else:
                    self.swap(i, self.right_child(i))
                    self.heapify(self.right_child(i))

    def insert(self, node, distance):  # O(log n)
        if self.size >= self.maxsize:
            return
        self.size += 1
        self.array[self.size] = (node, distance)
        p = self.size
```

```
        while self.array[p][1] < self.array[self.parent(p)][1]:
            self.swap(p, self.parent(p))
            p = self.parent(p)
        self.map[node] = p

    def make_queue(self, elements):  # O(n)
        # build a priority queue out of given elements
        for tup in elements:
            self.insert(tup[0], tup[1])

    def delete_min(self):  # O(1)
        # Return element with the smallest key and remove it from the
        return self.pop()

    def decrease_key(self, node_name, new_value):  # O(log n), at worst will need to
descend the entire tree
        i = self.map[node_name]
        self.array[i] = (node_name, new_value)
        while i > 1:
            if self.array[i][1] < self.array[i // 2][1]:
                self.swap(i, i // 2)
                i = i // 2
            else:
                break
        self.map[node_name] = i
```