Network Routing Design Experience

Class Unsorted Array

def \_\_init\_\_():

list = []

def insert(nodeIndex, value): # O(1)

list[nodeIndex] = value

def decrease key(nodeIndex, new value): # O(1) basically same as insert here

list[nodeIndex] = new value

def delete min: # O(V)

min = null

minIndex = null

for I in range (len(list)):

if list[i] == -1: continue

if list[i] == 0 | list[i] < min:

min = list[i]

minIndex = i

list[i] = -1

return minIndex

def create queue(node list): # O(V)

for i in range number of nodes:

list[i] = -1

return list

Class Heap

def \_\_init\_\_():

list = []

def insert(u) # O(logV)

if u.

def decrease key # O(logV)

def delete min # O(logV)

def create queue # O(VlogV) ?

Class Node

Def \_\_init\_\_(self, key):

Self.parent = null

Self.Left = null

Self.Right = null

Self.Key = key

# E is this the set of edges and V is the set of vertices or nodes in the graph

# Whole thing is O(V x Insert + V x DeleteMin + E x DecreaseKey)

# Array implementation: O(V x 1 + V x V + E x 1) so O(V^2)

# Heap implementation: O(V x

def Dijkstra’s algorithm(Graph(E, V), start node s)

for all vertices/nodes u in V: #O(V)

u.dist = math.inf

u.prev = null

s.dist = 0

H = makeQueue(V) # O(V x Insert complexity)

While H.size > 0: # -1 index means it is removed

u = deleteMin(H) # O(V x delete min complexity)

for all edges (u, v) in E: # This loop exactly E times so O(E)

if v.dist > dist.u + l(u,v):

v.dist = dist.u + l(u,v) # These all constant

v.prev = u

decreaseKey(H, v) # or O(E x decrease key complexity)