Erkin Aydın **ID: 22002956** **Section: 2**

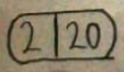
Question1:

Question 1

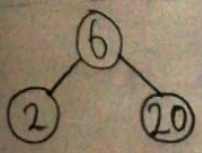
part a)

- Insert 2:

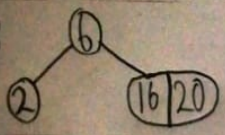
- Insert 20:



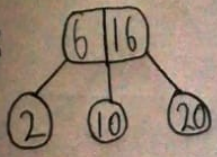
- Insert 6:



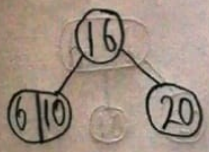
- Insert 16:



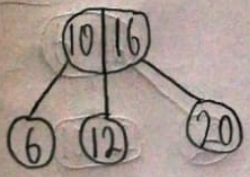
- Insert 10:



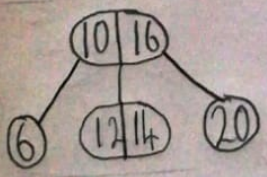
- Delete 2:



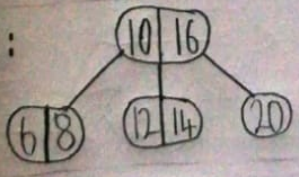
- Insert 12:



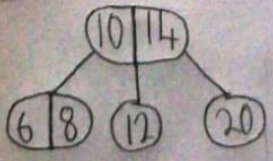
- Insert 14:



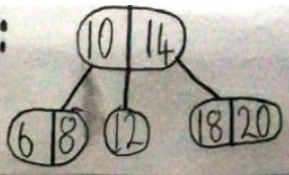
- Insert 8:



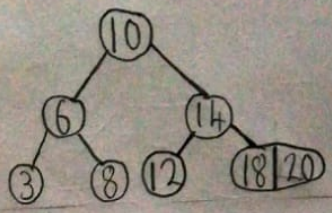
- Delete 16:



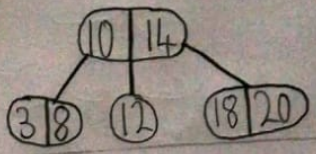
- Insert 18:



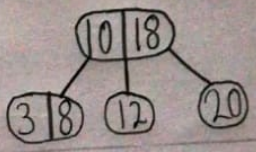
- Insert 3:



- Delete 6:

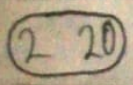


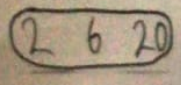
- Delete 14

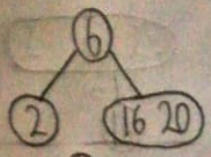


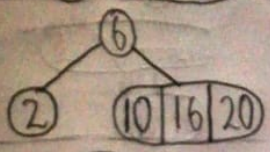
part b)

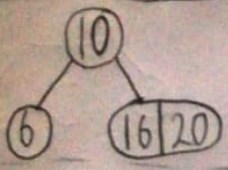
- Insert 2: 

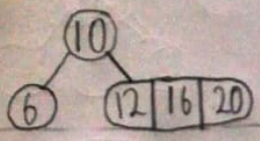
- Insert 20: 

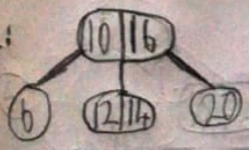
- Insert 6: 

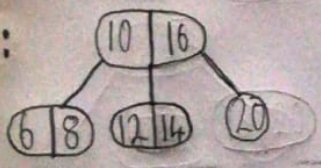
- Insert 16: 

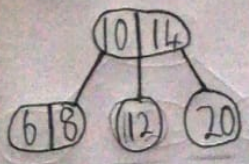
- Insert 10: 

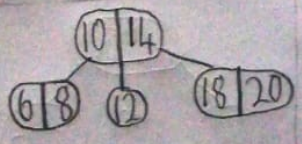
- Delete 2: 

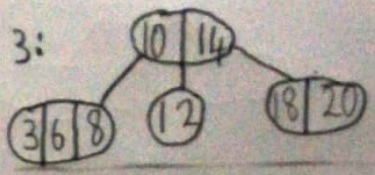
- Insert 12: 

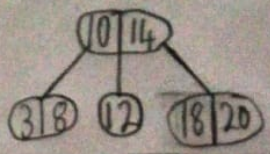
- Insert 14: 

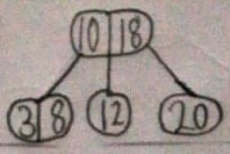
- Insert 8: 

- Delete 16: 

- Insert 18: 

- Insert 3: 

- Ddete 6: 

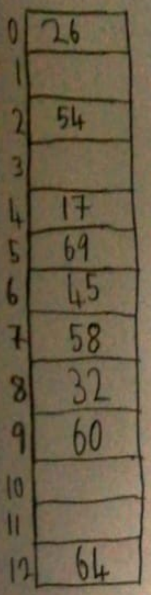
- Delete 14: 

Question2:

Question 2

Part a): Open Addressing with Linear Probing

Hash Table After Insertion



Average Number of Probes for a Successful Search

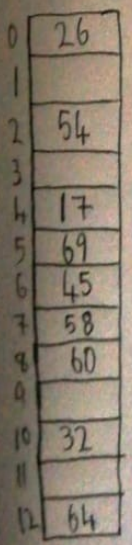
$14/9 = 1.\overline{5}$

Average Number of Probes for an Unsuccessful Search

$38/13 \approx 2.92$

Part b): Open Addressing with Quadratic Probing

Hash Table After Insertion



Average Number of Probes for a Successful Search

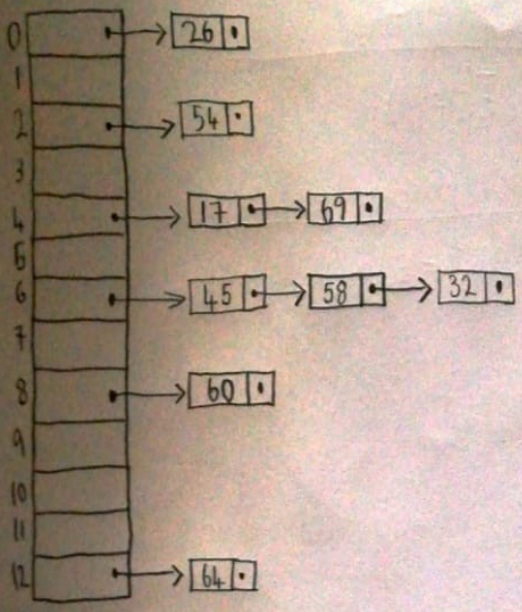
$13/9 = 1.\overline{4}$

Average Number of Probes for an Unsuccessful Search

$32/13 \approx 2.46$

Part C): Separate Chaining

Hash Table After Insertion



Average Number of Probes for a Successful Search

$13/9 \approx 1.4$

Average Number of Probes for an Unsuccessful Search

$22/13 \approx 1.69$

Question3:

Analysis of Insertion(insertEdge):

The worst case time complexity of this function is *O(V)*, where *e* is the amount of vertices in the graph. This case appears where we have two airports that have direct flight to every other airport other than each other. Here, to make the insertion, we need to traverse over two linked lists that have length proportional to number of vertices. Thus, the worst case time complexity of this function is *O(V).*

Analysis of List(listFlights):

This function requires to traverse a linked list to find all the flights, and linked lists in this implementation are proportional to number of vertices in the worst case in this implementation. Thus, the worst case time complexity of this function is *O(V)*, where *V* is the amount of vertices in the graph.

Analysis of Shortest Path(shortestPath):

In this function, I didn't use a priority queue, instead I mimicked it with an array. In a graph with *n* unvisited vertices, in the while loop, for each vertex(included in *n*) there are *n-1* adjacent vertices in the worst case. However, since we will mark visited vertices, the algorithm will will get logarithmic over time for each vertex. In short, we will visit every vertex and then look for the minimum edge to the unvisited vertices. Since there are *V* vertices and E edges in the graph and, the worst time complexity of this function turns into *O(V + Elog(V))* as expected from the Dijkstra's Algorithm.

Analysis of Minimize Costs(minimizeCosts):

In this function, I again mimicked a priority queue by using an array. In this instance, we will be searching for minimum edges to unvisited vertices, but this time we will not stop until all adjacent vertices, that is number of edges, of a visited vertex become visited. We will do this for each newly visited vertex. In other words, each vertex is inserted into the priority queue for once, and insertion to priority queue takes logarithmic time. Thus, the worst time complexity of this algorithm turns into *O((V + E)log(V)).*