CSCI 665
Assignment 4

4.T(1)=2
T(n)=(n+1)+1/n (n-1)Σ(q=1) T(q)

Multiply by n,
nT(n)=n(n+1)+ (n-1)Σ(q=1) T(q)

nT(n)-(n-1)T(n-1)=

=n(n+1)+ (n-1)Σ(q=1) T(q) -n(n-1)- (n-2)Σ(q=1) T(q)

=n(n+1)-n(n-1)+T(n-1)

nT(n)-(n-1)T(n-1)=2n+T(n-1)

nT(n)=2n+T(n-1)+nT(n-1)-T(n-1)

nT(n)=2n+nT(n-1)

Divide by n on both sides,

T(n)=2+T(n-1)

Expanding the recurrence,
=2+2+T(n-2)
=2+2+2+T(n-3)
=2+…..+ 2+T(n-k)
=2k+T(n-k)
Let k=n-1
=2(n-1)+T(1)
=2n-2+2
T(n)=2n

5.a
Consider, we have a adjacency-list representation of a directed graph in the form of an array, ADJ. of size |V| where V is the number of vertices in the graph.
ADJ[v] corresponds to the list of adjacent vertices of v.
Out-degree:- it is total number of vertices in the linked list of vertex v. To compute the out-degree for each vertex,
we must count the number of out-degrees for each, which will take $O(V + E)$

because each edge and each vertex is referenced once.

in-degree: The in-degree of a vertex u is the number of nodes pointing to it, which is the total number of times u appears in an adjacency list. To compute the in-degree of each vertex, we have to look at the Adj[u] for each vertex u and increment the in-degree of a vertex every time it appears in Adj[u]. Since each edge and each vertex is referenced once, this also takes O(V + E).

b. Consider there are n elements in our Hash table. Since the contents of the hash table are of the form of a set and not a key value pair, the average number of element each list is n/m where m is the size of the table
The time complexity to determine whether an edge is in the graph is $O(n/m)$.
In this scheme disadvantages are more space than linked list and more query time than adjacency matrix. If our hash table supports dynamic resizing then we would need $O(n)$ space for each hash table in order to have $O(1)$ query time which results in $O(n^2)$ space.
The alternate data structure we can use is priority queue.
It is time and space efficient and hence solves the problem.
Yes, the data structure has disadvantage of complexity.

6.a.

Node	Distance(d)	Parent
u	0	NIL
x	1	u
y	1	u
t	1	u
w	2	t
s	3	w
r	4	w
v	5	r

6.b
In the BFS procedure with line 18 where the colour of the visited node is set BLACK:-
-The colour of all the nodes are initialized to WHITE
-The colour is changed to GRAY once it is enqueued.
-Eventually, the colour is set BLACK after the node is dequeued and its neighbours are visited.

The colour is finalized to BLACK once it is dequeued after all its neighbours have been visited. Hence, the colour does not have any real meaning as such.Hence, in order to accommodate single bit usage for vertex's colour, the line 18 can be removed without any consequences.

The only condition which the algorithm checks whether the colour of the node is WHITE or not so that it can deem it to be visited or not. Hence, if a colour is BLACK or GREY, it is quite meaningless.Hence we can safely remove the Line 18.

6.c.The BFS algorithm is a not a complete procedure .As a matter of fact, we can say that the procedure only traverses the vertex and its adjacent vertices. All the vertices' d value is initialized to infinity at the start. Therefore the ordering of the vertex in the adjacency list does not matter. If a vertex is adjacent to the current node then irrespective of its order, it will be traversed. Also, each vertex has a cost d attached to it from the current vertex. Hence, even if the order is ignored the value of d would be the same in all cases.
In Figure 22.3, if vertex t precedes vertex x in Adj[w], we can get the resulting breadth-first tree computed by BFS as shown in the figure.
But if vertex x precedes vertex t in Adj[w] and vertex u precedes vertex y in Adj[x], we would get edge (x, u) in the breadth-first tree computed by BFS.
Hence, Breadth-first tree computed by BFS can depend on the ordering within adjacency lists.

7.DFS-STACK(G)

```
   For each vertex u ∈ G.V
      u.color = WHITE
      u.π = NIL
   time = 0
   for each vertex u ∈ G.V
      if u.color == WHITE
         DFS-VISIT-STACK(G, u)

DFS-VISIT-STACK(G, u)
   S = Ø
   PUSH(S, u)
   Time = time + 1            // white vertex u has just been discovered
   u.d = time
   u.color = GRAY
   while !STACK-EMPTY(S)
      u = TOP(S)
      v = FIRST-WHITE-NEIGHBOR(G, u)
      if v == NIL
         // u's adjacency list has been fully explored
         POP(S)
         Time = time + 1
         u.f = time
         u.color = BLACK     // blackend u; it is finished
      else
```

```
        // u's adjacency list hasn't been fully explored
        v.π = u
        time = time + 1
        v.d = time
        v.color = GRAY
        PUSH(S, v)

FIRST-WHITE-NEIGHBOR(G, u)
   For each vertex v ∈ G.Adj[u]
      If v.color == WHITE
         Return v
   Return NIL
```