CS2040S Final Assessment

MCQ

Perform inorder traversal on the BST. This gives us a list of elements in ascending order. The inorder traversal will take O(n) time. Once done, we can construct an AVL tree in O(n) time by picking the middle element as the root of the AVL tree. Doing so will split the list into two sublists: one sublist containing all elements to the left of the root, and one sublist containing all elements to the right of the root. Recursively pick the middle element as the root for each of these subtrees. Doing so will take O(n) time.

Q1: Given a BST (which is not balanced) of size **N**, to convert it to a balanced BST (according to AVL property) requires at least:

avl tree: insertion/deletion is O(log n)

a. O(1) time
b. O(log N) time
c. O(N) time
d. O(Nlog N) time

in best case, only one node is imbalanced

max heap -> root element is always max of left and right children

Q2: You are given a binary max heap, which uses an array implementation (as covered in lecture). N is known, and is guaranteed to be odd. If we want to find the median value of the heap, we can do so in:

height of complete binary tree - O(log n)

a. O(1) time
b. O(log N) time
c. O(N) time
d. O(N log N) time

Run quickselect on the underlying array. If preserving the original heap is a concern, we can copy out the contents of the array, and then perform quickselect on the copy instead

Q3: Given an undirected graph, we want to find out if this graph is a tree. Initially, the only information you have is that this graph is an undirected graph. No additional information is provided. Which of the following combinations of information will be sufficient to deduce, **for all possible graphs**, if the graph is a tree?

i. Number of vertices in the graph
ii. Number of edges in the graph
iii. In/out degree of each vertex
iv. All edges, given in the form of an adjacency list

tree: the min no of edges s.t. graph is still connected
E = V - 1 and only one component

a. (i) only
b. (i) and (ii)
c. (i), (ii) and (iii)
d. (i), (ii), (iii), and (iv)

in out degree for DAG check?

(i) alone is not enough. For (ii), if the number of edges is not V-1, then it is definitely not a tree. However, if it is V-1, it could either be, or not be a tree, and we need more information. For (iii), if any vertex has degree 0 (except for the special case where V=1), then it is definitely not a tree. If all vertices have degree > 0, then there is still insufficient information to determine if the graph is a tree, as there could be multiple components. Only by getting all edges are we able to perform graph traversal to determine if it is a tree (E=V-1, and there is only one component)

Q4: You are given an initial list of integers, which you should store in a DS of your choice. Afterwards, your DS should be able to support the **removeMin()** and **removeMax()** operations, which removes and returns the smallest/largest element respectively from the DS. It is guaranteed that **no other operations will be performed on the DS**. In order to **minimise the total time complexity** of the removeMin() and removeMax() operations, which DS would be the most ideal?

a. Hash Table
b. Sorted doubly linked list
c. Minimum binary heap
d. AVL tree

hash -> O(1) but need no order

sorted, double linked -> has pointer to from min to max and max to min : O(1)

A sorted array will be able to do removeMax() in O(1) time, but removeMin() in O(n) time. A sorted doubly linked list can do both in O(1) time. A minimum binary heap can do removeMin() in O(log n) time, but would need O(n) time to removeMax(). An AVL tree can do both in O(log n) time.

why min bin heap
removeMin: O(log n)
removeMax: O(n)

because need to traverse through the array to find max as bin heap is not sorted

Q5: The following 5 strings are inserted into an initially empty Trie:

bear
bold
cold
cord
dear

How many nodes (including the root) will be present in the Trie after all strings are inserted?

a. 9
b. 10
c. 18
d. 21

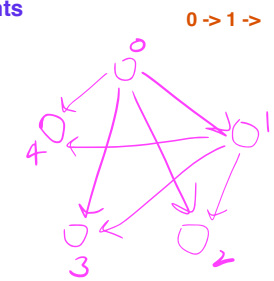
18 including the root

Q6: You are asked to draw a directed graph with 5 vertices and 7 directed edges. Additionally, the graph must contain the largest possible number of valid topological sorts. How many valid topological sorts are present in this graph?

a. 1
b. 6
c. 24
d. 120

5 vertices, 5 possible starting points
remaining 4, 4 choices
3 choices

total 5! = 5 x 4 x … x 1 = 120



The following 7 edges are present in the graph:
(0 -> 1), (0 -> 2), (0 -> 3), (0 -> 4), (1 -> 2), (1 -> 3), (1 -> 4)
This forces vertices 0 and 1 to be the first 2 vertices in the toposort, in that order. 2, 3, and 4 can appear in any order afterwards, for a total of 6 possibilities

2 -> 3 -> 4
2 -> 4 -> 3
3 -> 2 -> 4
3 -> 4 -> 2
4 -> 3 -> 2
4 -> 2 -> 3

Q7: 8 elements are added one at a time into an initially empty AVL tree. What is the maximum number of rebalancing operations that occurred across all insertions?

a. 3
b. 4
c. 5
d. 6

1st: no
2nd: no
3rd: yes

max = 8 - 2 = 6

With a perfect binary tree, it is impossible to trigger any rotations via one insertion. The total number of rebalancing operations is 4

Q8: Given a directed graph with V vertices, what is the maximum number of directed edges that can be present in the graph without forming any cycles?

a. V(V-1)
b. V(V-1)/2
c. V(V+1)
d. V(V+1)/2

tree: E = V - 1

each vertex v can connect to v-1 vertices

If a directed graph has no cycles, it is a DAG. A DAG has a valid toposort. In a valid toposort, there are no edges from right to left (ie. a vertex (X) to the right of another vertex (Y) in the toposort would mean that an edge from X to Y definitely does not exist). Owing to this, the maximum number of edges is
(V-1) + (V-2) + (V-3) + ... + 1 = V(V-1)/2

Q9: You want to find out if an **undirected, disconnected graph** is reverse-bipartite (not an actual term). For a graph to be reverse-bipartite, vertices can be assigned to one of two sets, and edges that appear in the graph can only be between two vertices in the same set (so these vertices should not be in two different sets). Your program **only needs to return true/false**. This can be done in:

a. O(1) time
b. O(V) time
c. O(V + E) time
d. $O(V^2)$ time

undirected-> bfs? on each set,

check adjacency list to ensure neighbours are all in same set

can be done meaning best case?

Always true (just put all the vertices into one set, and leave the other set empty)

Q10: You are given an undirected weighted graph G, and the only MST of this graph T. Then, one edge in G (which is also in T) is removed from the graph. You are now required to find an updated MST of G (G is guaranteed to still be connected after the edge is removed). This can be done in :

a. O(V) time
b. O(VlogV) time
c. O(V + E) time
d. O(ElogV) time

run counting CC to label the vertices in the MST (which is now 2 components since one edge is removed) based on which component they belong to. O(V+E) time if using adjacency list.
Now go through all edges in the original graph. find the smallest edge that links 2 vertices that belong to 2 different components and is not the removed edge. O(V+E) time if using adjacency list.
Total time = O(V+E)

Analysis:

False.
If R > N, then it is totally possible for all the N string keys to only differ by their last character. In this case, total space required is O(NR+wR) = O((N+w)R) instead, since the common prefix of length O(w) is stored only once (total space required for this is O(wR)) and only the last character of each of the N strings is stored in their own node (total space required for this is O(NR)). This is better than O(wNR).

no common prefixes

1. Given N distinct string keys (N > 1), each of the same length w (w > 2), that is formed from an alphabet set of size R (R > 1), storing all N keys in a Trie cannot take less then O(wNR) space.

True. Min space requirement of a trie containing n keys each of len 1 is is O(NR) hence with w > 2, the ave time complexity will be O(wNR)

2. There is no non-empty binary max heap containing integer keys which also satisfies BST property.

binary max heap

False.
A max heap containing only 1 integer key or 2 distinct integer keys will also satisfy BST property. There are many other counter-examples.

true, the 2nd(even)largest is always the left subchild of the root while the 3rd(odd)largest is the right subchild

3. In a SCC (strongly connected component), removing any one edge in the SCC will destroy the SCC (meaning not all vertices will be able to visit all other vertices in the SCC after that edge is removed).

SCC: containing 1 or more vertices in which any 2 vertices (if there are >= 2 vertices) are connected to each other by at least one path -> every vertex can visit every other vertex

false, each vertex is a scc with itself hence, even if one edge is removed, the scc is not destroyed

4. Floyd Warshall is the most efficient algorithm for solving APSP for all kinds of graphs.

false

For a tree, simply run DFS or BFS from each vertex as source to compute the APSP. This will take O(V^2) time, while using Floyd Warshall will take O(V^3) time.

False.
For example, a SCC having 4 vertices 0,1,2,3 with directed edges (0,1),(1,3),(3,0),(1,2),(2,3). In this SCC there are 2 directed cycles 0->1->2->3->0 and 0->1->3->0. Removing (1,3) will remove the smaller cycle but it will not destroy the SCC since there is still the larger cycle 0->1->2->3->0.

Structured Questions

avl operations: search, insert, delete, successor, predecessor

1. Given an AVL storing N distinct integer keys, give an algorithm to delete all keys that fall within the range from i to j (i <= j, i and j inclusive) in worst case time <= O(KlogN), where K is the number of keys to be deleted. You cannot use Java library TreeSet or TreeMap in your algorithm. You can only use the AVL operations discussed in lecture. If you need to modify any of the operations, describe your modification.

search for first integer i and get its pos
for(i < k) -> O(k)
  keep searching for each integer and delete it
  int val = avl.search(i) -> O(log N)
  delete(val) -> O(log N)

total complexity = O(K(2logN)) = O(KlogN)

1. Let node = search(i,bBST.root) <- O(logN) time

2. If node exist go to 3 else insert i into the bBST <- O(logN) time

3. Let snode = successor(i,bBST.root) <- O(logN) time
  i. delete(i, bBST.root) <- O(logN) time
  ii. if snode exists go to iii else stop
  iii. if snode.val > j stop
    else i = snode.val, repeat 3

//values are in order so successor of i will be the next node to be deleted

step 1 and 2 will take O(logN) in total. Step 3 will take O(logN) for each node deleted since there are at most K deleted nodes it will take O(KlogN) time in total. Total time taken = O(logN) +O(KlogN) = O(KlogN).

2. Given an unweighted graph (with at least 1 edge) stored in an adjacency list, give an algorithm that will return 1 if it is an undirected graph, 2 if it is a directed graph and 3 if it is a hybrid graph (a mix of directed and undirected edges). Your algorithm must run in average time <= O(V+E) where V is number of vertices and E is the number of edges in the graph.

diff between directed and undirected graph: in directed graph edges in the list will only appear in a particular list if there is there is an incoming edge into it but for undirected graph, every connected vertex is the neighbour of each other

for every vertex
  check if a vertex appears in the list of the vertex it is connected to
  if true: add 1 to the array were the vertex is the index
  if false: add 0 to the array
for i < array.length
  if all 1, it is undirected
  else if all 0, it is directed
  else it is hybrid

1. Let H be a hashset

2. For each vertex u in the adjacency list // average O(V+E) time
    For each neighbour v of u
      insert (u,v) into H //add all vertices to hashset

3. Let undirected1 = true, undirected2 = false //check both directions?

4. For each vertex u in the adjacency list // average O(V+E) time
    For each neighbour v of u
      //if u is neighbour of v and v is neighbour of u
      if H.contains((u,v)) && H.contains((v,u))
        undirected2 = true //the second direction exists
      else
        undirected1 = false //the first direction doesnt even exist

5. if (undirected1 == true)
    return 1
  else if (undirected1 == false && undirected2 == false)
    return 2
  else return 3
Time complexity = time complexity of step 2 + step 4 = average O(V+E) time.

3. There is a game where you have N dots numbered from 1 to N, and for some pairs of dots A,B there is an arrow going from A to B (if there is an arrow going from A to B then there will not be an arrow going from B to A). one directional

The objective of the game is to place the pencil at some starting dot and trace the arrows in such a way as to go through as many dots as possible without lifting the pencil (you may traverse some arrows multiple times to achieve this). can go back and forth

Given there is at least 1 starting dot A' which allows to you to trace the arrows in the above stated way and go through all the other dots, model the game as a graph and give the most efficient algorithm in terms of worst case time complexity you can think of to return one such A'.

1. add all pairs of dots to a hashset
  visited[n] = false
2. for every dot n in N
  if(!visited[n])
    visited[n] = true
    if n.neighbour is a dot that is in the hashset with it,
    set value of pair in hashset to 1

  else if visited
3.

Model the game as a directed graph where the dots are vertices and there is a directed edge from vertex i to vertex j if there is an arrow going from dot i to dot j.
Now simply run DFS topological sort algo. Without reversing the toposort array, just return the last vertex in the array as an A'.
Time complexity is O(V+E) time where V is number of vertices and E is the number of edges in the graph

4. Routers in a network are often susceptible to failure. Dr Ferdaus of Ferulock fame/infamy has developed his own protocol to test if a router in a network is still functioning.

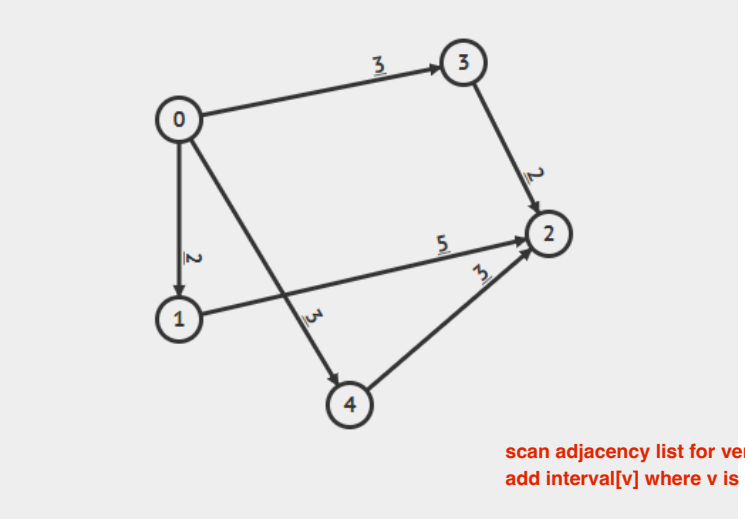
Given a graph representing the network, vertices are the routers (there are V number of them numbered from 0 to V-1) and the routers are connected by E directed edges ($\frac{V(V-1)}{10} \leq E \leq \frac{V(V-1)}{2}$). For any pair of router A and B connected by a directed edge from A to B with integer weight W(A,B) (W(A,B) >= 1), A will send a ping request to B at intervals of W(A,B) seconds (you can assume the time to send the request to B is 0 seconds). If B is alive it will have to send a default message back to A. All the routers in the network is synchronized to a global clock, so they will start counting of their intervals to send pings at the same time.

Now given that the graph as described above is stored in an adjacency list, give the best algorithm in terms of worst case time complexity you can think of to answer the following query (there can be many of such queries):

RequestList(i,K) - Return the 1st K vertices that sends a ping request to vertex i ordered by increasing time of request. Number of vertices sending pings to i <= K <= V. If two different vertices sends a request at the same time, they should be ordered by increasing vertex number.

If required, you can perform pre-processing that takes no more than O(V+E) time. Describe the algorithm for your pre-processing too.

In the example graph given below that represents a network of routers, RequestList(2,4) will return {3,4,3,1} as the first 4 vertices sending pings to vertex 2 in order of increasing time of request. The time of their requests are {2 secs, 3 secs, 4 secs, 5 secs} respectively.



Pre-processing step:
1. Let AL be the given adjacency list. Create another adjaceny list AL' of same size as AL 2. For each vertex u in AL // O(V+E) time
For each neighbour v of u
add u to neighbour list of v in AL' with weight W(v,u)
Time taken is O(V+E) to convert AL to AL'
RequestList(i,K):
1. Create a PQ P and array T
2. For each neighbour v of i in AL' // O(K'logK') time where K' is number of neighbors of i
// or can do fast heap create in O(K') time
insert triplet (W(i,v),j,W(i,v)) into P where they are ordered by the first field then the second
field
3. For j = 1 to K // O(KlogK) time
t = P.dequeue()
add t.second to back of T
insert triplet (t.first+t.third,t.second,t.third) into P
4. return T
Total time complexity = O(K'logK') + O(KlogK) = O(KlogK) since K >= K' as given in the problem description.

scan adjacency list for vertices pointing to i,
add interval[v] where v is index and value is the interval

for(i < interval.size)
  for(m <= K)
    calculate no of times each vertex sends a ping request -> interval^1 to interval ^k
    and add the vertex to a priority queue
    if two vertices have the same time, break tie by vertex number using comparator

for(j < K)
    print pq.get(j)

minimax path: minimum number of edges with max weight

5. Given a graph G that is a DAG with V vertices and E edges (where E = O(V)) stored in an adjacency list, you want to find the weight of the largest edge along the minimax path from a given source vertex A to a given destination vertex B. Give the best algorithm in terms of worst case time complexity you can think of to do it. You may assume there is at least 1 path from A to B.

1. use DFS to find the minimax path and store it in a hashset with vertex and key and weight as value - O(V + E)
2. largest_weight = -9999;
   for every vertex in hashset -> O(V)
      if curr_val > largest_weight
         largest_weight = curr_val
Total time complexity: O(V) + O(V + E) = O(V + E)

O(VlogV) time algo: Run Prim's starting from A and stop when it hits B. return the largest edge found so far

6. The salesman from tutorial 11 has begun another round of travelling around different cities and peddling his wares. This time he has set aside funds to pay the toll fee for every city he passes when getting from some source city A to some destination city B. Of course he still wants the shortest route to get from A to B as time is of the essence. However he has calculated that he has only enough money to pay the toll fee of at most K cities where 1<= K <= 10, thus he cannot pass through more than K cities when getting from A to B (including A and B themselves).

Given the value K and a graph G with V vertices and E edges, where the vertices are cities and bi-directional edges are roads connecting pairs of cities, and edge weight is the travel time (same in both direction), give the best algorithm you can think of in terms of worst case time complexity to find the cost of the shortest path the salesman should use to get from some city A to some other city B that does not involve more than K cities. If no such path exists output "no valid path from A to B".

do DFS from A to B, keeping a count of each
city visited, if count exceed and not reached ->
no value path, else if B reached
output the path -> O(K + E)

7. The global merchant guild has set her sights on being the most represented guild on the planet. In order to do so, the best way is of course to build an office in every country on the planet. However, the cost is too prohibitive and also some countries are at war with neighbouring countries and it would not be profitable for the guild to do business in such countries. After analyzing all the N countries (the countries are numbered from 0 to N-1) and their relationship to each other in terms of distance, politics etc.., the guild has come up the following possible relationships between a pair of country A,B.

A B 1 - Exactly one office will be built in either A or B because they are close to each other and you do not need one office in both A and B.

A B 0 - No office should be built on both A and B because they are at war and having a presence in either country would not be profitable at this time

A B 2 - An office should be built on both A and B because they are far apart but have very strong economic ties and to maximize profit it would be expedient to build an office in both countries.

Given M such unique pairs of countries and their relationships (there is only 1 relationship per country pair, e.g if there is 0 2 1, there will not be 0 2 0 or 0 2 2), determine the minimum number of offices to build to satisfy the M relationships so as to achieve maximum profitability and also which countries they should be built in. If it is impossible to satisfy the M pairs of relationships, simply return -1.

Give the best algorithm in terms of worst case time complexity you can think of to solve the problem.

run bfs + djikstra to
make