CS2040 2022/2023 Semester 2 Final Assessment

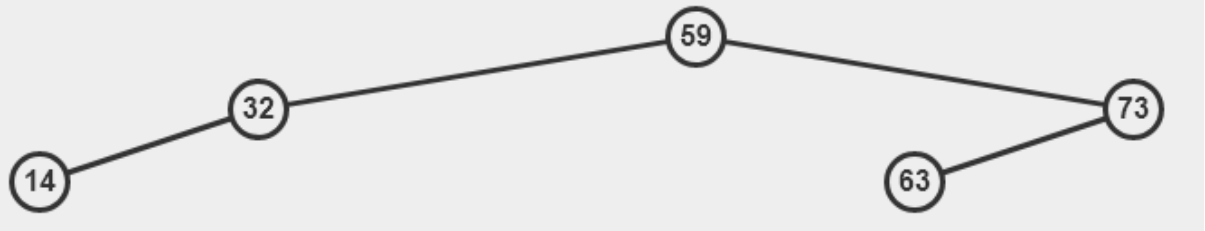
MCQ

This section has 25 questions and is worth 63 marks. 3 marks (Q1 - Q11, Q24 - Q25) or 2 marks (Q12-Q23) per question.

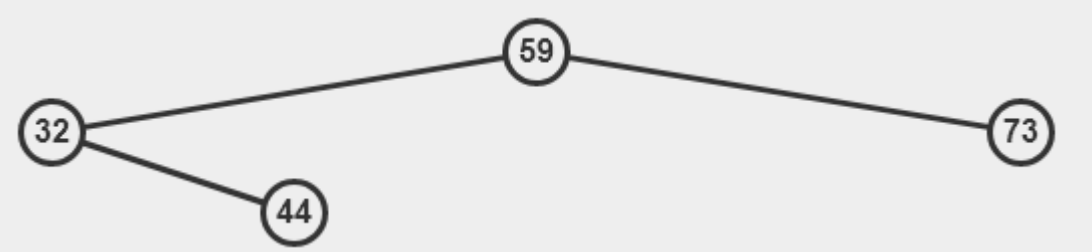
1. Determine the maximum number of nodes with exactly one child that can be found in a valid AVL tree of height 2.

a. 0
b. 1
c. 2
d. 3

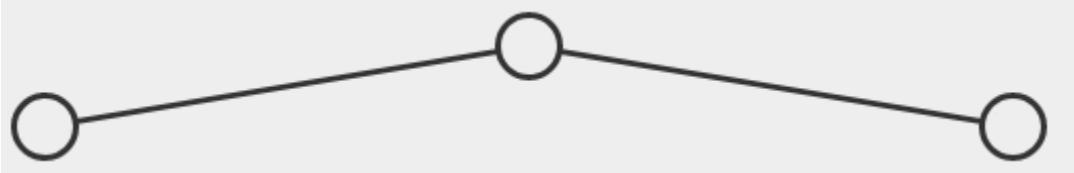
A: Consider the properties of a node with exactly one child in an AVL tree. The side without a child is considered to be height -1. Therefore, to ensure the balance factor does not differ by more than one, the only child must be of height 0 (ie. this child has no additional children of its own). Therefore, only parents of leaves could possibly fulfil this property. To ensure we have the maximum possible number of such nodes, all leaves should be as low as possible in the tree (ie. at the lowest level). As such, the best such tree for height 2 is as follows:



2. You are given the following AVL tree.



One element is removed from the AVL tree, and the tree now looks like this structurally:

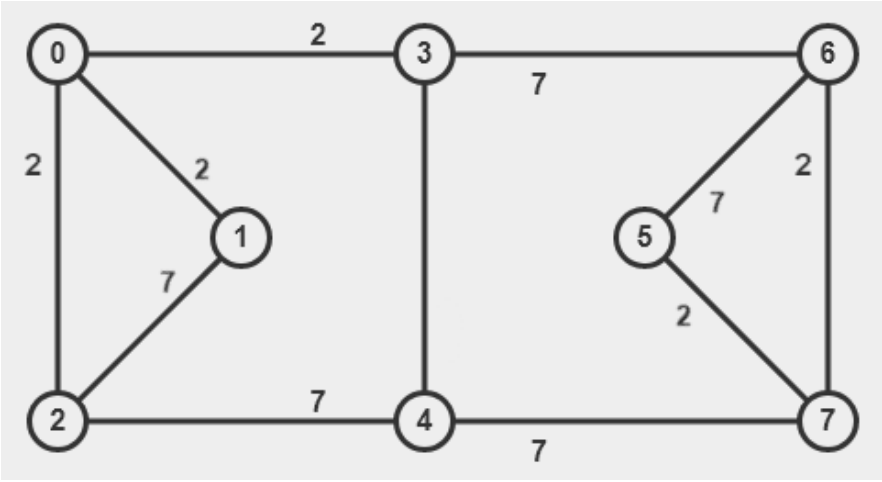


Note that some elements may have shifted as a result of rebalancing. Determine which element could have been removed. Select all that apply.

a. 32
b. 44
c. 59
d. 73

A: The remaining AVL tree only contains 3 elements, which only has one possible valid structure. Therefore, any of the 4 elements could have been removed, and the resulting structure would be the same.

3. You are given the following graph:



Note that the weight of edge (3, 4) is unknown, but can be one of the following integers: 1, 2, 4, 8. Which of these would result in the largest number of unique MSTs in the graph?

a. 1
b. 2
c. 4
d. 8

A: Without taking the edge into consideration, the graph currently contains 3 unique MSTs (either (2, 4), (3, 6) or (4, 7) can be removed).

If the edge has weight 1, the total number of MSTs drops to 2 (edge (2, 4) is definitely discarded, and now either (3, 6) or (4, 7) must remain in the MST).

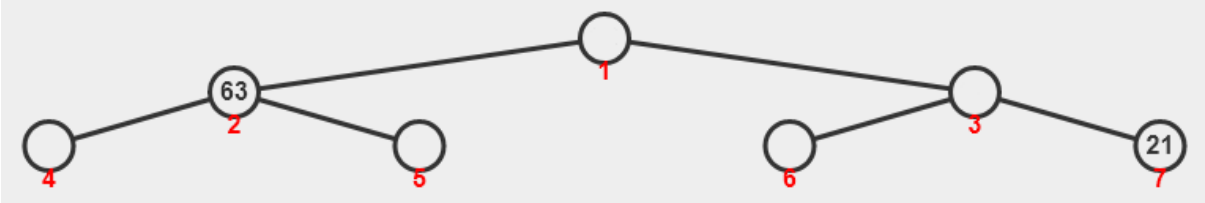
If the edge has weight 2, the total number of MSTs drops to 2 for the same reason.

If the edge has weight 4, the total number of MSTs drops to 2 for the same reason.

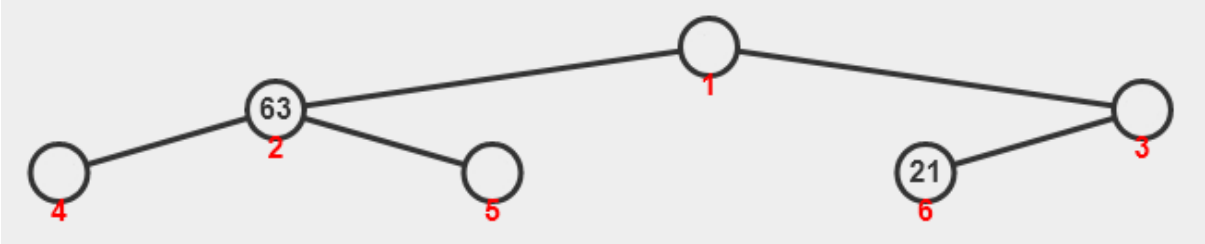
If the edge has weight 8, this is effectively the same as not including the edge in the graph (the weight is too large to be picked in an MST), and the number of MSTs remains at 3.

If weight 7 were provided as an option, this would give a total of 5 MSTs.

4. You are given the following partial information on a heap with distinct elements. It is not known if the heap is a min or a max heap.



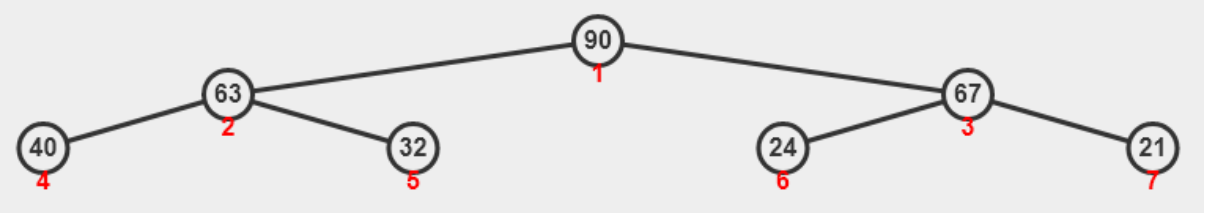
After removing the top element of the heap (via extractMin()/extractMax()), where appropriate for the heap type), the heap now looks like this:



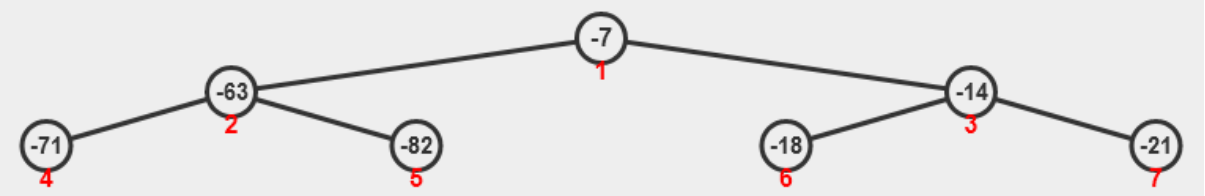
Determine if the heap could be a min heap or a max heap.

a. Could be both a min heap or a max heap
b. Could only be a min heap
c. Could only be a max heap
d. Neither a min heap or a max heap

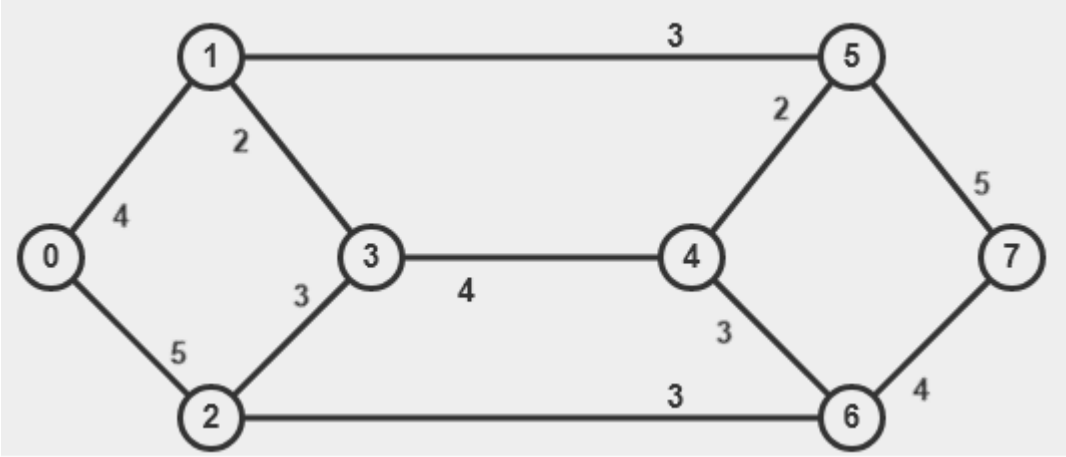
A: The following is an example of a max heap that does this (corresponding to O(N) create with 90,63,67,40,32,24,21 on VisuAlgo):



The following is an example of a max heap that does this (corresponding to O(N) create with -7,-63,-14,-71,-82,-18,-21 on VisuAlgo; the numbers are negative to simulate min heap):



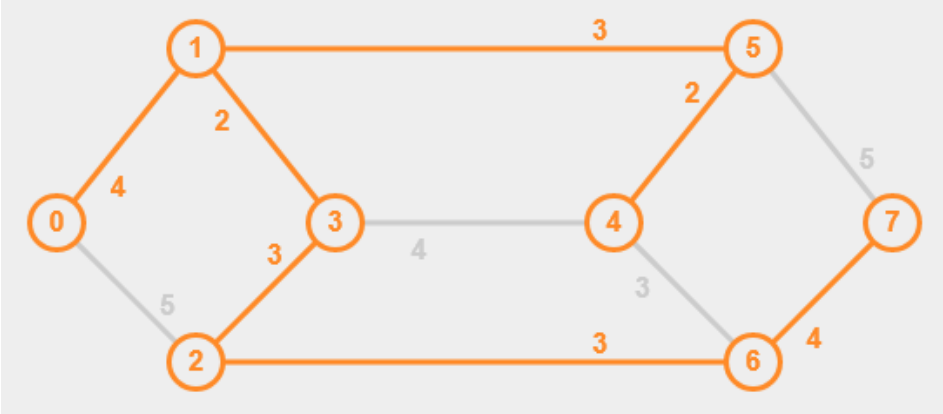
5. You are given the following graph, which may have multiple MSTs:



Which of the following changes (when applied independently) will not increase the total cost of the MST? Select all that apply.

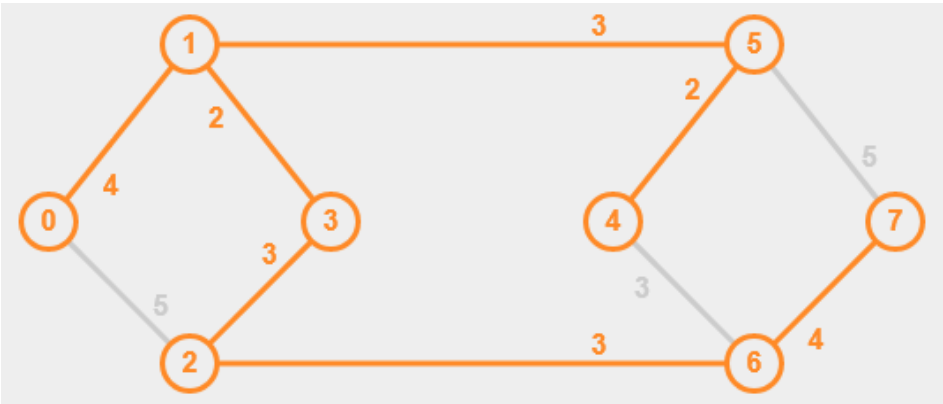
a. **Removing edge (3, 4)**
b. **Increasing edge weight of (1, 5) by 1**
c. **Adding edge (3, 5) with weight 2**
d. **Adding a new vertex (8), with an edge to vertex 7 of weight 3, and two edges to vertices 5 and 6, both of weight 2**

A:



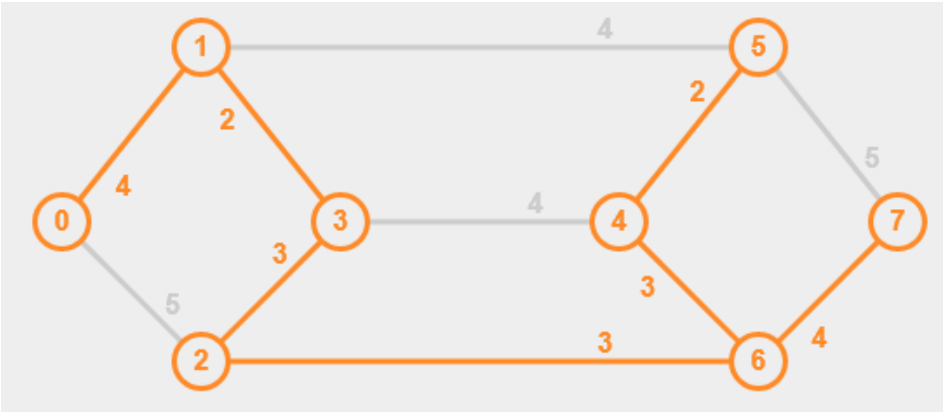
The current graph has an MST cost of 21. In terms of whether the options will increase the MST cost:

Removing edge (3, 4) (a):



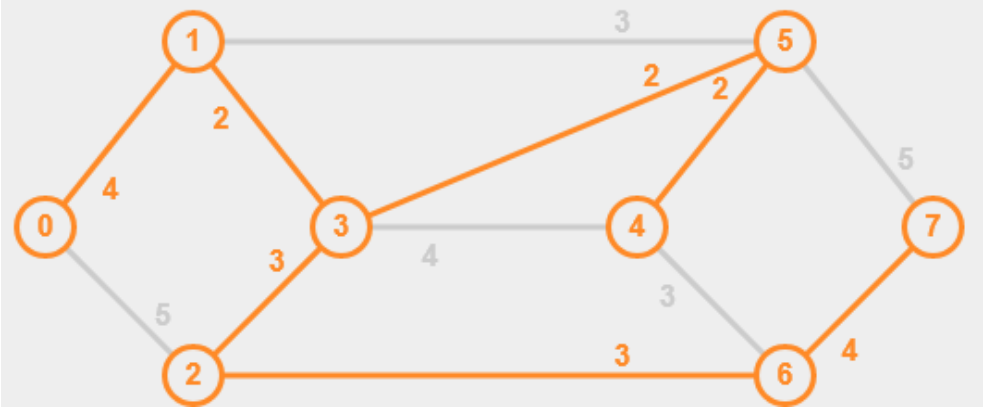
(3, 4) is not an edge in any valid MST. Removing this edge has no effect on the MST.

Increasing edge weight of (1, 5) by 1 (b):



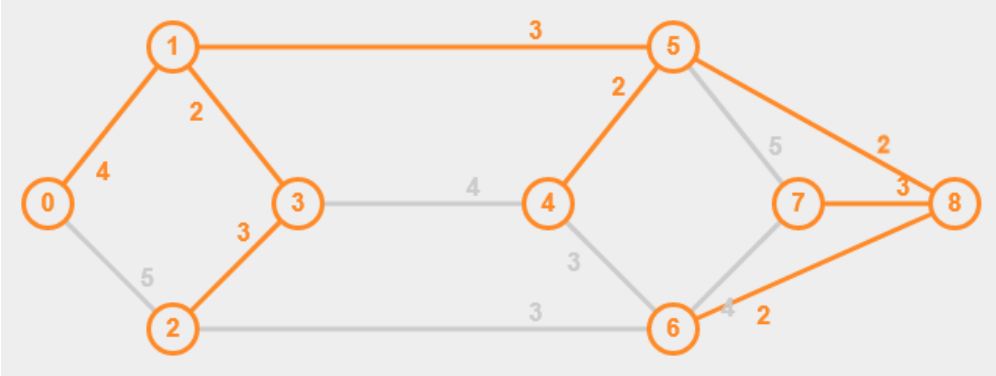
Edge (2, 6) or (4, 6) can be taken if an MST used to take edge (1, 5).

Adding edge (3, 5) with weight 2 (c):



Adding a new edge (on its own) can never increase MST cost.

Adding a new vertex (8), with an edge to vertex 7 of weight 3, and two edges to vertices 5 and 6, both of weight 2 (d):



Adding this vertex allows us to replace the edges (6, 7) and whichever edge was used to connect to vertex 6 (either (2, 6) or (4, 6)) with all 3 new edges from the vertex. This results in no overall change to the total cost.

6. Given the following adjacency matrix representing a directed graph, determine the properties of the graph. Blank entries should be regarded as containing a 0, for ease of reading.

	0	**1**	**2**	**3**	**4**	**5**	**6**
0							
1				3			
2		4				4	
3					5		
4			7				
5							
6							

a. Cyclic and weighted
b. Cyclic and unweighted
c. Acyclic and weighted
d. Acyclic and unweighted

A: The presence of edges with different weights show the graph is weighted. A cycle exists from 1 -> 3 -> 4 -> 2 -> 1.

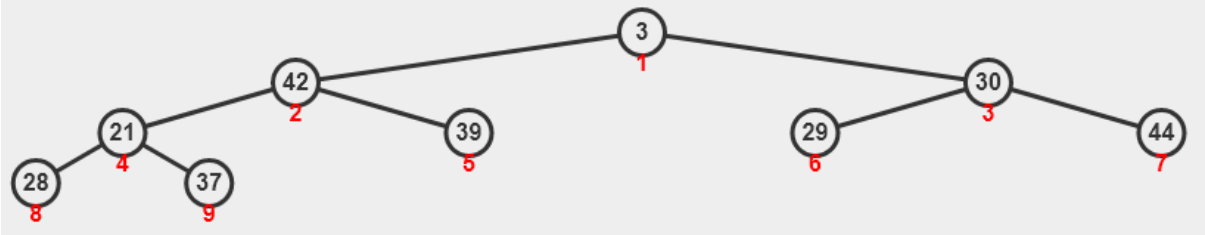
7. You are given an AVL tree, initially consisting of one node with the value 20. You are then asked to create the tallest tree by inserting integers one at a time to the tree, under the following constraints:
   1. Only integers in the range [0..1B] (ie. from 0 to 1 billion, both ends inclusive) can be inserted.
   2. An integer cannot appear more than once in the tree.

   Determine the maximum possible height of the tree.

a. 5
b. 6
c. 7
d. 8 or more

A: We can simply add all 1B values to the tree in any order. While there is no way to know exactly how the tree will look, we know that a complete binary tree of height 8 contains 511 nodes, and therefore the height of a tree with 1B nodes must be (much) larger than 8.

8. The following heap is the result of copying an array into a min heap during O(N) buildHeap(), but before any swaps are made:



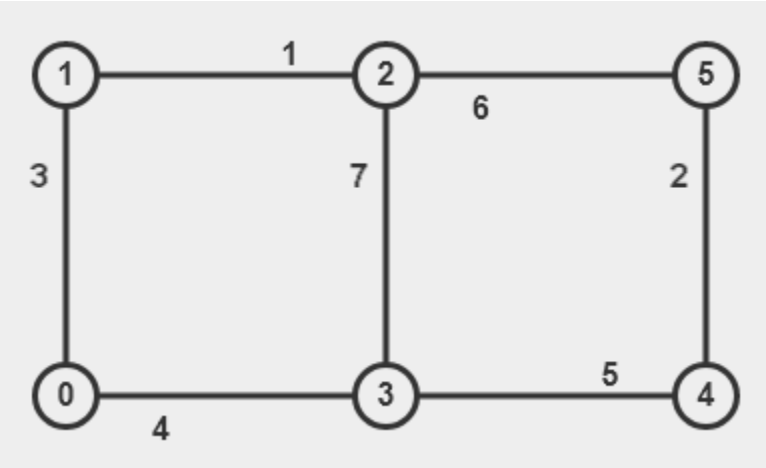
Determine the number of swaps that are made as a result of the subsequent steps in O(N) buildHeap().

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

A: The following swaps occur during the remainder of the steps (the first number shown in the swaps is always that of the parent):

30 -> 29
42 -> 21
42 -> 28

9. You are given the graph below.



Prim's was run on the graph from an unknown source vertex, and the first 3 edges picked to be part of the MST are (0, 3), (3, 4), (4, 5), though not necessarily in that order. Determine how many vertices among the set {0, 3, 4, 5} are possible source vertices for Prim's.

a. 1
b. 2
c. 3
d. 4

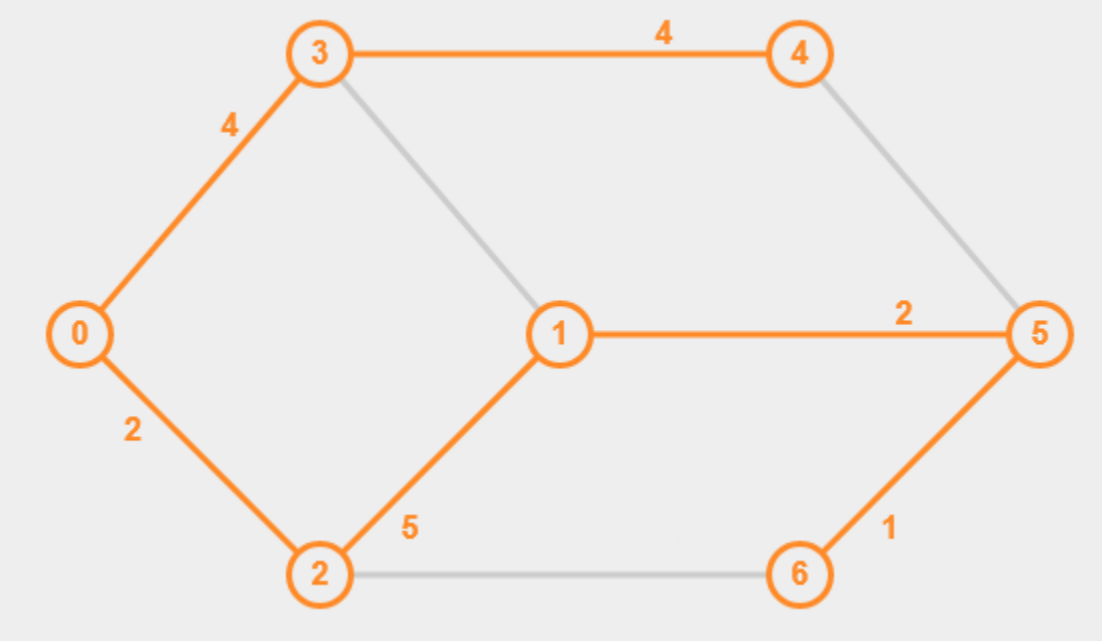
A:The first edge picked when starting from 0 would be (0, 1). This is not part of the 3 edges given.

The first two edges picked when starting from vertex 3 would be (0, 3) and (0, 1). (0, 1) is not part of the 3 edges given.

When starting from 4, the first 3 edges picked are (4, 5), (3, 4) and (0, 3).

When starting from 5, the first 3 edges picked are (4, 5), (3, 4) and (0, 3).

10. The following is a graph with its only MST shown. The weights of edges not in the MST are unknown.



Determine the minimum total sum of the weights of the 3 edges not included in the MST. For those with difficulty seeing grey lines on a grey background, the edges are:

(1, 3), (2, 6), (4, 5)

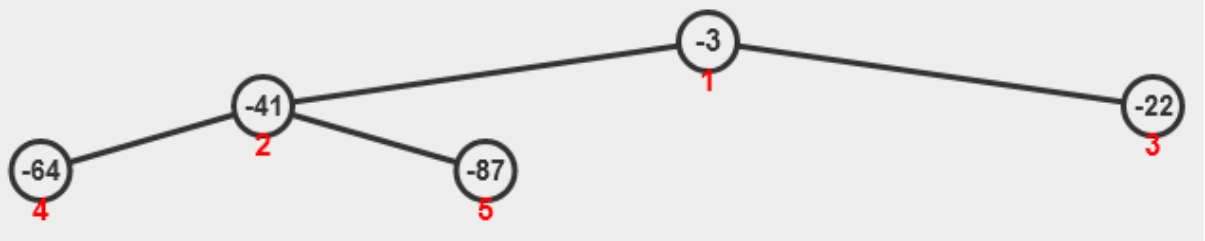
a. 10
b. 17
c. 18
d. 21

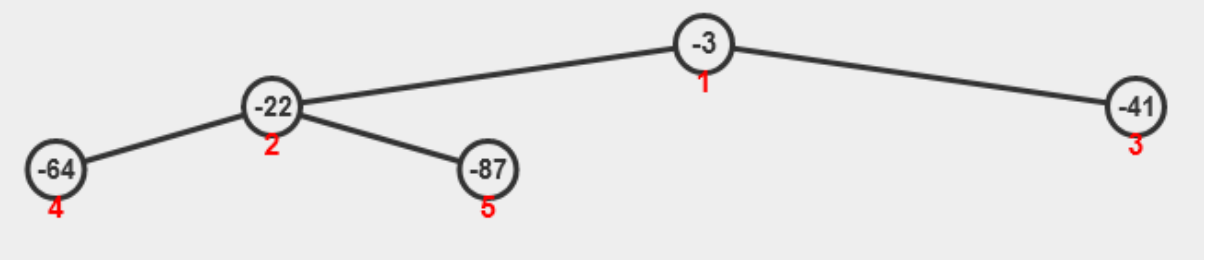
A: The minimum weight of edge all edges is 6. If any of these edges had weights 5 or below, they could be picked over edge (1, 2) in the MST. Therefore, the total weight is 18.

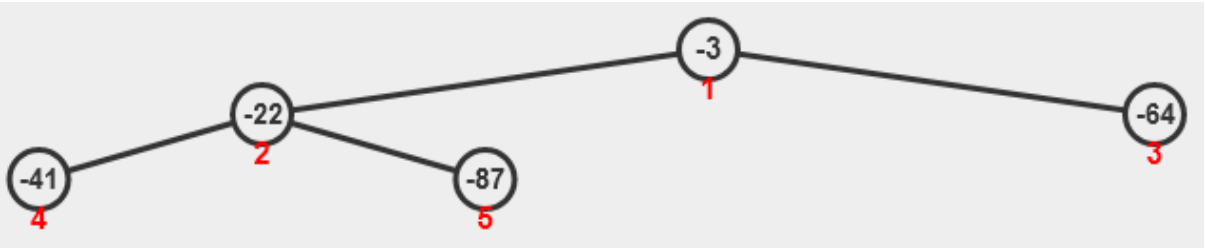
11. The numbers {3, 22, 41, 64, 87} are stored in a binary min heap. Determine how many possible positions (from index 1 to index 5 inclusive) that the element 41 could be in the min heap.

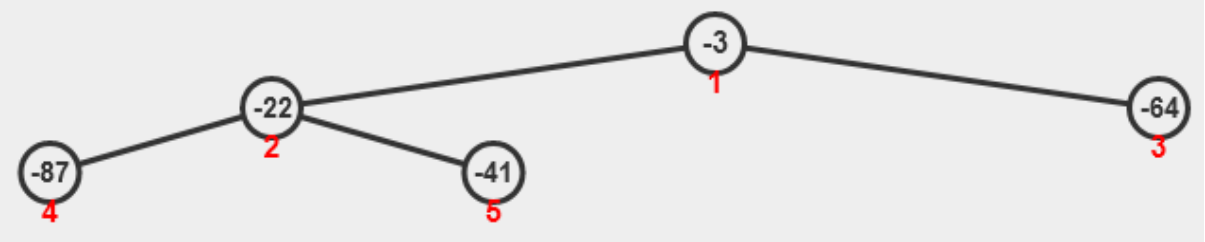
a. 1
b. 2
c. 3
d. 4

A: 41 could be in any of the indices from 2-5 inclusive. Some examples below (negative numbers are used to simulate a min heap:









Below is a modified BFS snippet for questions 12 to 15. The bolded lines indicate the modification from the original BFS. Assume that all graphs are undirected.

```
for all v in V
   visited[v] ← false
Q ← {s} // Q is a queue
visited[s] ← true

while Q is not empty
   u ← Q.dequeue()
   visited[u] ← true // this line is added
   for all v adjacent to u
      if visited[v] = false
         // visited[v] ← true (this line is removed)
         Q.enqueue(v)
```

12. For a **tree** with V vertices, the worst-case time complexity of the modified BFS is
    - **a. O(V)**
    - b. $O(V^2)$
    - c. $O(V^3)$
    - d. $O(2^V)$
    - e. $O(V * 2^V)$

13. For a **complete graph** with V vertices, the worst-case time complexity of the modified BFS is
    - a. O(V)
    - b. $O(V^2)$
    - **c. $O(V^3)$**
    - d. $O(2^V)$
    - e. $O(V * 2^V)$

14. Suppose we modify the BFS further by changing Q into a min-heap. For a **tree** with V vertices, the worst-case time complexity of the modified BFS is
    - **a. O(V log V)**
    - b. $O(V^2 \log V)$
    - c. $O(V^3 \log V)$
    - d. $O(V * 2^V)$
    - e. $O(V^2 * 2^V)$

15. Suppose we modify the BFS further by changing Q into a min-heap. For a **complete graph** with V vertices, the worst-case time complexity of the modified BFS is

a. O(V log V)
b. $O(V^2 \log V)$
c. $O(V^3 \log V)$
d. $O(V * 2^V)$
e. $O(V^2 * 2^V)$

Added during exam: The min-heap is keyed using the vertex number.

Ans:
One thing about this modified BFS: a vertex may be enqueued and processed multiple times. This is a common bug for people who tried to write BFS for the first time :)
An important observation: Once a vertex has been dequeued from Q, it will never be enqueued again in the future.

Q12: In a tree graph, the modified BFS will behave like the original BFS.

Q13: Without loss of generality, assume the neighbor list is iterated in increasing order and s = 1. In a complete graph, the modified BFS will enqueue vertex i (i = 2, …, V) i-1 times. To be precise:

- 1 will enqueue 2, 3, …, V in this order
- When we dequeue vertex i for the first time, we will enqueue all vertex j > i once
- After the first V dequeue, we have dequeued 1, 2, 3, …, V. Thus, we will not enqueue anymore

In total, there will be $O(V^2)$ enqueue. As each vertex of a complete graph has O(V) edges, in total, the time complexity will be $O(V^2 * V) = O(V^3)$

Q14: due to min-heap, there is an additional factor of O(log V) from Qn 1 answer. Example graph: a star graph centered at s.

Q15: Without loss of generality, assume s = 1. We can show that vertex i (i = 2, …, V) will be pushed to the min-heap 2^{i-2} times. To be precise:

- There will be 1 copies of 1 in the min-heap, so 1 will enqueue 2, …, V once
- There will be 1 copies of 2 in the min-heap (pushed by 1), so 2 will enqueue 3, ..., V once
- There will be 1+1=2 copies of 3 in the min-heap (pushed by 1 and 2), so 3 will enqueue 4, ..., V twice
- And so on, with the pattern that vertex i (i = 2, …, V) will be pushed 2^{i-2} times.

Also, we can observe that the size of the min-heap will be bounded by $O(2^{V-1}) = O(2^V)$. As each vertex has O(V) edges, the time complexity will be $O(2^V * \log(2^V) + 2^V * V) = O(V * 2^V)$. (the first term is from min-heap usage, while the second term is from iterating the neighbors)

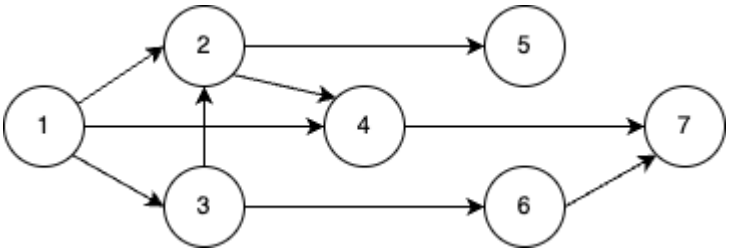
16. On a UFDS which only uses **union-by-rank**, the worst-case time complexity for any UFDS operation is
    a. O(α(N))
    b. O(N)
    c. O(log N)
    d. O(sqrt N)

17. On a UFDS which only uses **path compression**, the worst-case time complexity for any UFDS operation is
    a. O(α(N))
    b. O(N)
    c. O(log N)
    d. O(sqrt N)

Ans:
Q16: is similar to what was discussed in the UFDS tutorial.

Q17: An example of the worst case: Do N-2 union operation without path compression being triggered (we only do union on the root). This will make a chain of N-1 nodes. Finally, do a union with the N-th node, but this time we invoke the union at the leaf. This will traverse the whole chain, incurring O(N) time.

Refer to the following directed graph for questions 18 to 19.



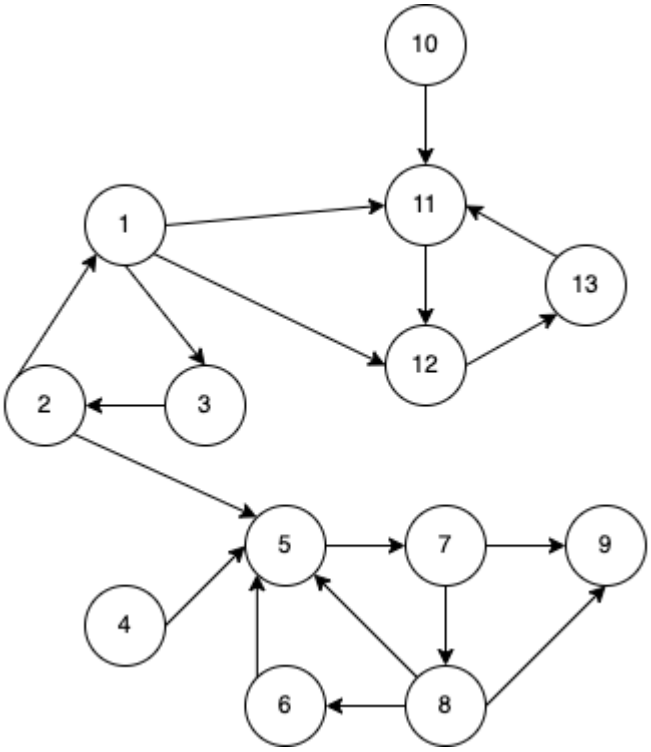
18. A possible topological ordering of this graph is
    a. 7, 6, 4, 5, 2, 3, 1
    b. 1, 3, 4, 2, 6, 7, 5
    c. 1, 3, 2, 6, 5, 4, 7
    d. 1, 2, 5, 3, 4, 6, 7
    e. 1, 3, 2, 5, 6, 7, 4

19. We want to add a new directed edge in the graph while preserving the number of different topological orderings. That is, if in the old graph there are 11 topological orderings, the new graph must also have 11 topological orderings. Among the following edges, the one that satisfies this requirement is
    a. From 3 to 1
    b. From 2 to 7
    c. From 4 to 6
    d. From 7 to 5
    e. From 5 to 6

Ans:
Q18: just check one by one.

Q19: A new edge u → v will not change the number of the topological orderings if and only if in the original graph, u can reach v and v cannot reach u. Among the choices, the only one that satisfies this is 2 → 7.

Refer to the following directed graph for questions 20 to 22.



20. The number of strongly connected components in the graph is
    - **a. 6**
    - b. 7
    - c. 8
    - d. 9
    - e. 10

21. The minimum number of edges that has to be removed so that the graph does not have any cycle is
    - a. 0
    - b. 1
    - c. 2
    - **d. 3**
    - e. 4

22. The minimum number of edges that has to be added so that each vertex is reachable from any other vertices is
    - a. 0
    - b. 1
    - c. 2
    - **d. 3**
    - e. 4

Ans:
Q20: There are 6. The strongly connected components of the graph are {1, 2, 3}, {4}, {5, 6, 7, 8}, {9}, {10}, {11, 12, 13}.

Q21: We only need to focus on edges that go from and end at the same SCC. Manually counting gives us 3. For example, by removing 1 → 3, 5 → 7, and 11 → 12.

Q22: If the graph is already strongly connected, then no edges are needed. Otherwise, let A be the number of SCC with no incoming edges from other SCC. Similarly, let B be the number of SCC with no outgoing edges to other SCC. We need at least
max(A, B) edges. From the graph, A = 3 (SCCs: {1, 2, 3}, {4}, {10}) and B = 2 (SCCs: {9}, {11, 12, 13}). Thus, we need at least max(3, 2) = 3 edges. An example is by adding 9 → 4, 13 → 1, and 9 → 10.

Question 23 to 25 refers to the problem described below:

You are given an array A of N **distinct** integers with values between [1, K] (1 and K inclusive).

For an integer X > 1 (which can be very large), determine whether there is a sequence of **non-negative** (can be 0) integers $u_0, u_1, \ldots, u_{N-1}$ such that $u_0A_0 + u_1A_1 + \ldots + u_{N-1}A_{N-1} = X$.

If there is such a sequence, output 1. If there is no such sequence, output -1.

23. Let N = 4, K = 100, and A = [51, 10, 32, 37]. Among the following X, the one where we must output -1 is

    a. 102
    b. 103
    c. 104
    d. 105
    e. 106

24. Suppose K = N+1. The best algorithm to solve the problem (output 1 or -1 given A, K and X) will have a worst-case time complexity of

    a. O(1)
    b. O(N)
    c. $O(N^2)$
    d. $O(N^2 \log N)$

25. This problem can be modelled as a graph problem.
    The problem we are trying to solve is #blank 1# and it can be best solved in #blank 2#

    Choose the correct option for #blank 1# and #blank 2# respectively from the following:

a. Graph traversal, O(N + K) worst case time complexity
b. Graph traversal, O(NK) worst case time complexity
c. SSSP, O((N + K) log (N + K)) worst case time complexity
d. SSSP, O(NK) worst case time complexity
e. SSSP, O(NK log NK) worst case time complexity

Ans:

Q23: A heuristic we can use is to first put away 10 from the list. Then, notice that if we can construct an integer Y from the remaining integers, then Y + 10*u (u >= 0) can always be constructed. Hence, we just need to focus on the last digit. Observe that

- 51 * 2 = 102
- 52 * 1 + 32 * 1 = 83
- 32 * 2 = 64
- 32 * 3 = 96

Using the previous observation, it means we can construct 102, 103, 104, and 106. Meanwhile, the smallest integer ending with the digit 5 that can be constructed is 115, leaving 105 as the answer.

Q24: There are few cases to consider:

- If 1 is on the list, then an answer always exists.
- Otherwise, in sorted order, the list must be 2, 3, …, K-1, K. If K = 2, an answer exists if and only if X is an even integer. Otherwise, K >= 3, and an answer always exists (since every integer value > 1 can be formed from 2a+3b where a,b >= 0)
- Thus for K >= 3 return 1 in O(1) time, otherwise scan A (which is of size <= 2) for 1 and return the answer again in O(1) time.

Q25: We can extend the idea from Q23. Construct a graph with vertices numbered $0…A_0-1$. For each vertex i, make a directed edge to $(i + A_j) \% A_0$ with weight A_j for $1 <= j <= N-1$ (ignore all edges that point back to vertex i itself). Observe that a path starting from vertex 0 will correspond to a sequence $u_1, u_2, …, u_{N-1}$. Initially, at vertex 0, we have $u_i = 0$ for all $1 <= i <= N-1$. When we take an edge corresponding to A_j, it means we increase u_j by 1. Then, a path ending at vertex i will correspond to $u_1A_1 + … + u_{N-1}A_{N-1} = Y$ where $Y \% A_0 = i$. Note that at this point, u_0 will still be 0.

Finally, we run SSSP on this graph with vertex 0 as the source. Denote the distance to vertex i as dist[i]. Intuitively, dist[i] denotes the smallest integer Y which we can construct using $A_1, A_2, …, A_{N-1}$ where $Y \% A_0 = i$. Observe that for any integer X, we can construct it if and only if $dist[X \% A_0] <= X$.

- If $dist[X \% A_0] <= X$, we can set $u_0 = (X - dist[X \% A_0]) / A_0$, and obtain $u_1, u_2, …, u_{N-1}$ from the shortest path to vertex $X \% A_0$.
- If $dist[X \% A_0] > X$, by the definition of $dist[X \% A_0]$, we can never construct X.

We can use modified Dijkstra's algorithm as there is no negative weight edge. As the graph has V = O(K) vertices and E = O(NK) edges, the time complexity of running modified Dijkstra's will be O(NK log NK).

Note: When constructing the graph, we do not have to use A_0; In fact, the best choice is to use the smallest element from A to make our graph smaller. However, this will not change the worst-case time complexity of the solution.

Analysis

This section has 8 questions and is worth 17 marks. 2 marks per question except for Q31 which is worth 3 marks.

26. Question 26 and 27 refers to the following problem:

    You are given an unordered set of *n* integers. From this unordered set, you would like to print out a "max-zig-min-zag" ordered output of length *k* (where *k* << n, that is, k is much smaller than *n*) with the following property: the first element is the largest number, followed by the smallest number (which is the 2nd element), followed by the 2nd largest number, followed by the 2nd smallest number, etc., up to the *k*-th element.

    For example, given the following set of unordered *n* = 10 numbers: 31, 5, 9, 27, 13, 45, 39, 3, 19, 22, and *k* = 5, the output should be: 45, 3, 39, 5, 31.

    Since your friend John has taken CS2040, he suggested that the best data structure(s) (in term of worst case time complexity) to solve this problem is an additional array where you copy over the original set of n integers, sort the array in descending order and then just alternate between printing and deleting the elements from the front and from the back of the array until you have printed k elements.

    *Note: you may assume that printing out an integer takes O(1) time.

    Select the appropriate option for the statement "John is correct based on what is taught in CS2040":

    a. True
    b. False

27. Select the most appropriate reason for your answer in Q26:
    a. It is true because the total time complexity of his suggested data structure and algorithm is O(nlogn + k), where O(nlogn) is required for sorting, while deleting from front and back of an array is only O(1) time, so doing this k times is only O(k). There is no other faster data structure and algorithm from what is taught in CS2040.
    b. It is false because using a bBST is the best. Simply insert all the n integers into the bBST, then alternate between printing and removing the largest and smallest, taking a total of O(nlogn + klogn). John's method involves deleting an element from the front of the array thus the total time complexity of his algorithm will be O(nlogn + kn).
    c. It is false because simply accessing and printing the required element rather than deleting it will result in a faster time complexity of O(nlogn + k) instead of O(nlogn + kn), and there is no faster data structure and algo from what is taught in CS2040.
    d. It is false, because there are other data structure(s) taught in CS2040 which can be used to solve the problem that can result in a worst case time complexity of O(n + klogn) which is better than O(nlogn + kn), O(nlogn + klogn) or O(nlogn + k) since k << n as stated in the problem description.

Ans:

Q26: False.

Q27:

Option a. is not correct since deleting from front of array is not O(1) but O(N). Thus John's algo is O(nlogn + kn) time not O(nlogn + k) time.

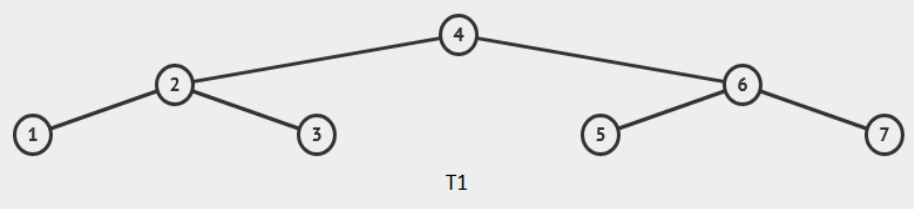
Option b. while the time complexities are correct is not correct because bBST is not the best option here.

Option c. just like Option b. while the analysis of the time complexities is correct is it not the best option here.

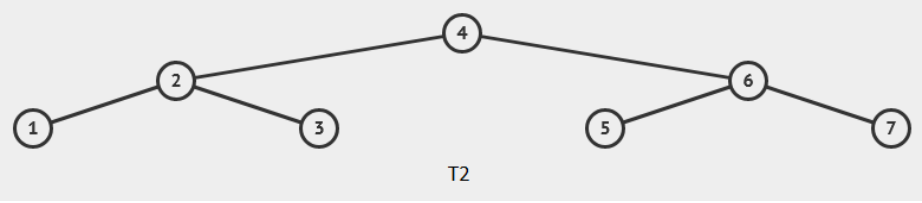
Option d. is correct, simply use a max heap and a min heap to store the n integers. You can create the min heap and max heap is O(n) time using fast heap create, then to print out the "max-zig-min-zag", you just alternately extract from the max heap and the min heap for a total of O(klogn) time. Thus the time complexity for this method using a max heap and min heap is O(n+klogn) which is better than O(nlogn+kn) (John's method), O(nlogn+klogn) (bBST method) or O(nlogn + k) (John's method without deletion) since k << n.

28. Question 28 and 29 refers to the following problem:

Two binary search trees are *identical* if they are both structurally identical (i.e., both of them have nodes in the same places of the tree) and the corresponding nodes have the same node values. For example, these two binary search trees T1 and T2 are identical:



T1



T2

Becky has designed the following algorithm to check if two binary search trees are identical:

1. Perform an inorder traversal for both trees and output the node values during the traversal.
2. If both of the output lists are identical (e.g., T1: 1,2,3,4,5,6,7 and T2: 1,2,3,4,5,6,7) then the BSTs are identical.

Becky's algorithm is correct.

Select the appropriate option for the statement "Becky's algorithm is correct."

a. True
b. False

29. Select the most appropriate reason for your answer in Q28:
    a. It is true, because if they contain the same node values then in-order traversal will have them all in the same sorted order.
    b. It is true because since both are binary search trees, and if the node values are the same (based on the output lists given by in-order traversal) then the structure must also be the same.
    c. It is false because in-order traversal can result in random ordering of the node values in the output list and thus even if the node values are the same in both binary search trees, you cannot compare the 2 output lists generated.
    d. It is false because the structure of the 2 binary search trees can be different even if the node values are the same, depending on the order of the insertion of the values into the binary search tree. Thus even though in-order traversal will give an sorted output list, all you can conclude is that both binary search trees contain the same node values (if both output lists are the same) but we cannot conclude they have the same structure.

Ans:

Q28: False

Q29:

Option a. is correct only up to determining that the node values are the same but not the structure.

Option b. is not correct because BSTs that contain the same values may have different structure.

Option c. is obviously wrong.

Option d is right, and an example is given below:

T1 can be created by inserting 4,2,6,1,3,5,7 in that order, while T2 can be created by inserting 5,4,6,2,7,1,3 in that order among other possible insertion orders.

30. Question 30 and 31 refers to the following problem:

    For every directed weighted graph G with unique edge weights, for every source vertex s in G, the shortest path tree rooted at s always includes the lightest edge (edge with the smallest weight) in G. (Recall, the shortest path tree is the tree consisting of the shortest paths from s to every other node in the graph).

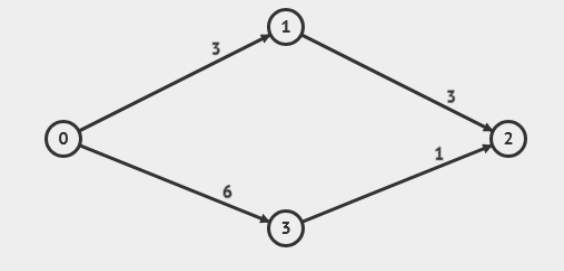
    a. True
    b. False

31. Select the most appropriate reason for your answer in Q30:
    a. This is TRUE. Assume the lightest edge is v'->u'.
       Now there is some source vertex v and destination vertex u where the SP from v to u (v~>u) does not include v'->u'.
       We now find the SP from v to v' (v~>v') and the SP from u' to u (u'~>u) and form an even shorter path v~>v'->u'~>u. This is a contradiction.
    b. It is TRUE because in the case of such a graph the MST will be the same as the shortest path tree and since the lightest edge must be included in the MST when all edge weights are unique, it must therefore also be included in the shortest path tree.
    c. It is FALSE because it is possible that for some source vertex s, some subgraph of G might not be reachable from s and the lightest edge can be part of that subgraph, and this is the only reason why it is FALSE.
    d. It is FALSE because it is possible that for some source vertex s, some subgraph of G might not be reachable from s and the lightest edge can be part of that subgraph, and this is one of multiple reasons why it is FALSE.

Ans:

Q30: False

Q31: option d is correct. The obvious reason is that the lightest edge lies in some subgraph that is not reachable from the source vertex, but there are also other reasons, one of which is that the lightest edge unfortunately lie along the most expensive path from source to some vertex an example of which is given below:



Clearly for vertex 0 as source, the edge 3->2 will not be in the SSSP tree rooted at 0 since it lies along the costlier path from 0 to 2 thus will not be used.

32. Question 32 and 33 refers to the following problem:

    Another way of counting the number of strongly connected components (SCCs) in a directed graph with *V* vertices and *E* edges is to use union-find disjoint sets (UFDS).
    1. Start with a UDFS containing *V* disjoint sets, each representing a single vertex in the graph.
    2. For each edge (u, v) in the graph, we call the UDFS operation unionSet(u, v).
    3. Count the number of unique returned values of calling the UFDS operation find(*v*) for each vertex *v* in the graph. This is the number of SCCs in the graph.

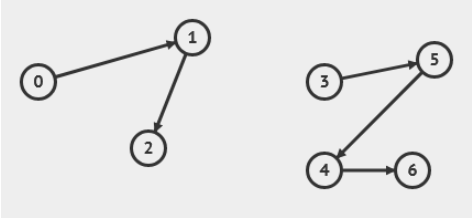
a. True
b. False

33. Select the most appropriate reason for your answer in Q32:
    a. This is TRUE because the number of unique returned values in step 3 is the number of disjoint set in the UFDS after finishing step 1 and 2 and since each disjoint set now represents a SCC, the number of disjoint set is the number of SCCs.
    b. This is TRUE because the algorithm can be used to correctly count the number of components in an undirected graph and thus can also be used to count the number of SCCs in a directed graph.
    c. This is FALSE because it is not necessarily true that each disjoint set after step 1 and 2 represents an SCC. This is because the algorithm does not take into consideration the direction of the edges.
    d. This is FALSE because the algorithm given will only be true for counting the number of components in an undirected graph and will never be able to give the correct number of strongly connected components for any directed graph.

Ans:

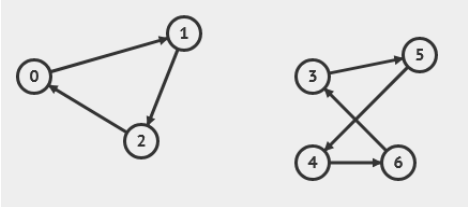
Q32: False

Q33: option c is correct. An example will be as follows:



Using the algorithm described, {0,1,2} will be in a disjoint set and {3,4,5,6} will be in another disjoint set, but none of the disjoint set represent a SCC. In fact there are not 2 SCCs but 6 SCCs in the graph (each vertex is its own SCC).

However this does not mean the algorithm will never be able to give the correct number of SCCs (option d). Another example is given:



For this graph, again {0,1,2} and {3,4,5,6} will be in the same disjoint set and each disjoint set actually represents a SCC.

Structured Questions

This section has 5 questions and is worth 20 marks.

Write in **pseudo-code** for questions that require you to write the solution.

Any algorithm/data structure/data structure operation not taught in CS2040 must be described, there must be no black boxes.

Partial marks will be awarded for correct answers not meeting the time complexity required.

34. Question 34 and 35 refers to the problem below:

    Envision that you are working at Tesla, which announces next year that they will produce a new car model. Your supervisor asks you to create a website to manage pre-orders. On the website prospective customers can enter their name and pay a small reservation fee to secure their spot in the future list of deliveries. You are tasked with creating an efficient data structure that manages all the names and the time of each registration. The registration time&date is used to maintain the pre-orders in a first-come-first-serve order.

    Your data structure needs to support the following three operations efficiently:

- **insert(name, time_date)**: Lets a person insert a pre-order. The time_date information is used to order the pre-orders in the data structure. You may assume time_date is an integer value describing the current number of second from epoch (00:00:00 UTC on 1 January 1970) at the point the pre-order was made, and you may assume that time_date is unique for all pre-orders made.
- **delete(name)**: Lets a person delete a pre-order, i.e., some people change their mind and withdraw their pre-order.
- **getPosition(name)**: Lets a person check in which position in the list of orders they are right now. If the person has the smallest time_date for his pre-order, he will be in the first position and if he has the largest time_date then he will be in the last position. Remember, a person's position may move up if somebody in front of them cancels their pre-order. So, assigning a person a fixed position does not work.

[2 marks] Describe which data structure(s) you would use to support the above three operations efficiently, all with worst case time complexity O(log n) or better, where *n* is the number of pre-orders. Describe any augmentations that you add to your data structure.

**We will look at your answer for Q34 and Q35 before giving marks for either of them.*

Added during exam:
1. name is unique
2. There can only be 1 pre_order per person at any one time.

Ans:
An AVL tree A where each node stores time_date as key value, augmented with a *size* attribute at each node, to support the rank() operation for getPosition()
+
Another AVL tree A' where each node stores the pair info <name, time_date> but uses name as the key value.

35. **[9 marks]** Now using the data structure you have stated in Q34 along with any augmentation given, describe the algorithm for each of the 3 operations so that each results in a worst case time complexity of O(logn) or better.

Ans:

Insert(name, time_date):
    A.insert(time_date)
    A'.insert(<name, time_date>)

delete(name):
    pair_info = A'.find(name) // since A' is ordered using name this can be done in O(logn) time
    A.delete(pair_info.time_date) // now delete the corresponding entry from A in O(logn) time

getPosition(name)
    pair_info = A'.find(name) // since A' is ordered using name this can be done in O(logn) time
    A.rank(pair_info.time_date) // get the rank of the corresponding node in A in O(logn) time

Grading Scheme:

Q34:

The marks allocation is as follows:

Possible to implement all 3 methods correctly (ignoring time complexity; that part is penalized later in q35) with no further data structures (temporary or otherwise) needed: 1m
Whether your methods in q35 follow the constraints provided by your answer in q34: 1m

Q35:

The marks allocated to each method is as follows:

insert(name, time_date): 2m
delete(name): 3m
getPosition(name): 4m

For each method, 1m is deducted for running in O(n) worst case time, and 2m for worst case time complexity slower than O(n). Take note that this does not mean you are automatically awarded 2m for a fast but incorrect method; you may still be awarded 0 marks for a method owing to lack of correctness.

The following are several common incorrect responses, and their associated marks. However, note that owing to differences between individual students' answers, you may still be awarded more or less marks than the ones shown here:

Using AVL (time_date as key), combined with HashMap (name as key, time_date as value)

All relevant HashMap operations (put,get,remove) run in worst case O(n) time. Therefore, most such answers get 6m. The comments on such answers generally follow the following example:

insert:
O(n) time

delete:
O(n) time

getPosition:
O(n) time

Using AVL (time_date as key) only

It becomes impossible to find a person by name in O(log n) time, as the AVL tree is not ordered by name. Therefore, comparing against the name in the current node offers no clue as to whether the person you are looking for should be in the left or right subtree. The correct way would require going through the entire tree, which takes O(n) time. An additional 1 mark is deducted for attempting to search for the person in an incorrect manner, therefore most such answers get 5m. The comments on such answers generally follow the following example:

insert:

delete:
O(n) time (not ordered by name)

getPosition:
O(n) time (not ordered by name)

Using Binary Min Heap (time_date or name as key) only

It is not possible to search for a person (regardless of whether it is keyed by name or time_date) in O(log n) time. Any references to "search" therefore assume the use of the usual operation which runs in O(n) time.

However, most such answers also claim that the position can be found by simply returning the index at which the person is found. This is incorrect; there is no guarantee that the index represents their sorted position in the DS, since the underlying array is not necessarily in sorted order. This is a major issue (theoretically solvable at a cost to speed), and therefore has a higher penalty of -2 to correctness. As such, most such answers get a total of 5m. The comments on such answers generally follow the following example:

insert:

delete:
O(n) time

getPosition:
O(n) time
incorrect result

Other mistakes (in no particular order):

Specifying a BST (instead of a bBST or AVL tree) as your data structure. This will affect your marks as the worst case time complexity of most operations in a BST are O(n).

Assuming that orders are given in chronological order. There is no such constraint; orders could have been placed offline and then entered into the system at a different time (which is why the time_date of order placed is used as a parameter for insertion). Thus you are required to order them based on the provided time_date value.

Concluding that the rank of an element in a BST is simply node.left.size + 1. This is incorrect; the other parts of the rank() method are important too.

Implementing getPosition() by simply calling search(). This only returns the node containing that key in the tree, and does not give its position.

Using array-based BST, and concluding that its index in the array is its rank. There is no such constraint on the array.

Assuming that time_date is provided as a parameter for delete() and getPosition(). The method signatures show that only name is provided.

Storing the rank of a BST node directly. Insertions and deletions may cause the entire tree to have their ranks updated (by adding to or removing from the very left of the BST), requiring O(n) time to update correctly.

Claiming that eg. operations will run in O(log n) time because bBST operations run in O(log n) time, with no further explanation. Without specifying what methods you intend to call in your DS for each of the 3 operations, your implementation becomes unclear, and marks will be deducted accordingly.

Using black box operations. Any non-standard operation you use must be clearly implemented. Hiding your implementation will also lead to us hiding your marks.

Confusion between binary heaps and BSTs. BSTs do not use shiftUp/shiftDown methods, and heaps cannot be searched through by going left if searchedKey < node.key, and going right if searchedKey > node.key, where searchedKey is the key being searched for, and node is the current node in the heap.

Using select() instead of rank(). Rank() gives the position of a node in a BST given its key, while select() gives the node (or key) in a BST given its position.

Questions 36 to 38 refers to the problem described below:

There are *V* radio stations (numbered from 0 to V-1), including a special station *s* (one of the *V* radio stations) that is the source of all the radio signals. There are *E* (*E* > *V*) pairs of stations which are close enough to be connected. For each such pair of station, there is a maintenance fee to connect the pair. This maintenance fee is stored in an array A of size E where each entry is a triple <x,y,c>, x and y being the radio stations which can be connected and c is the fee of maintaining the connection.
Assume that the fees are unique (i.e., no two costs are the same), and assume the graph induced by the stations is connected. Assume that you have already computed a minimum spanning tree T_0 of the radio stations stored in an adjacency list *AL*, thus minimizing the cost to distribute the signal to all nodes.

36. **[1 mark]** A pair of radio stations x and y which are connected in T_0 is experiencing some problems with their connection, thus breaking the link between *x* and *y*. This will break T_0 into

    a. 2 components

    b. 3 components

    c. 4 components

    d. Not enough information to determine how many components it will break T_0 into

Ans: Since T_0 is a tree, each edge is a bridge. Removing any edge will break T_0 into 2 components.

37. **[3 marks]** Assuming there are other connections (which were not used in T_0) that can be used to re-form a spanning tree of the radio stations, the best algorithm to do so while minimizing the maintenance cost of the resultant spanning tree will run in worst case time complexity:

    (You are given *s* the source radio station, A the array of maintenance fees and also AL the adjacency list containing T_0 the original MST)

    a. O(ElogV)

b. O(1)

c. O(V)

d. O(E)

Ans: Removing edge (A, B) divides the graph into two components. The MST of the new graph must include the lightest edge connecting these two components. To see that the resulting tree is an MST, consider adding some edge e = (u, v) that is not in the MST. There are two cases: either e connects the two components created by removing (A, B) or it does not.
If e does connect those two components, then we know that it used to be the heaviest edge on the cycle containing (A, B); the only question is whether or not the new edge is heavier. But we know that the new edge must be lighter than e, because we chose the lightest edge to connect the two components. Hence e is the heaviest edge on the cycle and does not belong in the MST.
If e does not connect the two components (i.e., it forms a cycle within one of the two components), then we know immediately that e is the heaviest edge on the cycle – since it was the heaviest edge on the cycle before the new edge was added.
In either case, we know that every edge not in the tree is the heaviest edge on some cycle, and hence should not be in the MST.
In order to find the lightest edge that crosses the cut:

- Starting at A, do a DFS labelling each node as part of component 1.
- Starting at B, do a DFS labelling each node as part of component 2.
- Iterate through all the edges in the graph, finding the lightest that has one node in component 1 and one node in component 2.

The running time of this algorithm is O(E).

38. **[5 marks]** A startup company wants to test a new breakthrough technology for connecting stations, and offers, for free, to connect your source station *s* to another station. The new technology is strong enough to reach *any* station from *s*. You can pick one of the stations other than the *s* and connect this station to *s* with NO maintenance cost! Free!
    Given the above, it is quite certain there is a better MST T_1 that can be formed. Describe the most efficient algorithm you can think to create T_1.
    (You are given *s* the source radio station, A the array of maintenance fees and also AL the adjacency list containing T_0 the original MST)

Ans:

When a new edge is added (in this case the new connection which costs 0), it creates a cycle in the MST, and we can remove the heaviest edge on the cycle to create a new MST. Thus, on adding the new edge, the MST will remain the same except for the new edge that is added (costing 0) and one existing edge that is removed.
That is, the cost of the new MST will be equal to the cost of the old MST minus the weight of one edge. Given that, it is clear that we want to remove the heaviest edge on the MST.

Solution 1: Worst case O(V) time (remember that in a tree E = V-1)
1. Identify the largest edge by go through AL in O(V) time.
2. Assuming largest edge is (u,v) remove u from neighbor list of v and vice versa in O(V) time.
3. The MST into now broken into 2 components and now the answer is simply the same as that for Q37 which takes O(V) time. After getting the components that each vertex is in you want to have a 0 weight edge from s to any vertex s' that is not in the same component as s. Simply add s to neighbor list of s' and vice versa in O(1) time. Total time for this step is O(V).
4. Return AL as the new MST T1.

Solution 2: Worst case O(V) time (remember that in a tree E = V-1)
1. Identify the largest edge by go through AL in O(V) time.
2. Assuming largest edge is (u,v) remove u from neighbor list of v and vice versa in O(V) time.
3. Now, perform a DFS from the s until we reach either u or v. (We cannot reach both, as by disabling (u, v) we have disconnected the MST.) Takes O(V) time.
4. If the DFS from s reaches u, then we create a new connection (s, v) in O(1) time (adding s to neighbor list of v and vice versa); if the DFS from s reaches v, then we create a new connection (s, u) similarly.
5. Once the new connection is created, we have a path from u to v again (via s), and hence the resulting graph is connected, and it must be an MST T1 required.

Grading Scheme:

Correction solutions:
5 marks - worst case O(V) solution

4 marks - Other solutions taking between worst case O(V^2logV) and O(V) time. Usually a solution using the same algorithm as Solution 1 or Solution 2 but using more expensive algorithms to find the largest edge or to determine how to link the 2 components created.

3 marks - Worst case $O(V^2 \log V)$ solution (call this **Solution 3**): for each station v (excluding s) add a free connection from s to v in T_0 (independently of the other stations) and rerun kruskals/prims. Return the best MST that results.

2 marks -
Worst case O(VElogV) same as Solution 3 but run on the original graph instead of the MST.
Worst case $O(V^3)$ solution: Solution 2 variant that uses Floyd Warshall to find largest edge (u,v) then remove it and replace it with s to the either u or v depending on which is not in the same component as s when (u,v) is removed.

1 mark - Worst case $O(V^4)$ same as Solution 3 but use MiniMax variant of Floyd Warshall to solve it instead of kruskal's/prims.

Deductions of correct solution due to mistakes:
1.Wrong algorithm to identify the largest edge in T_0. **-3 marks**

2. Find the path from s that contains the largest edge but no details given on how to do this. **-2 marks**

3. using Solution 2 or variant: remove the largest edge (u,v) from T_0 but connect s to the wrong vertex between u and v (s should connect to vertex it no longer has a path to in the MST once (u,v) is broken). **-2 marks**

4. Using Solution 2 or variant: remove the largest edge (u,v) from T_0 but not made clear which vertex s is connected to with 0 weight in order to connect the 2 components created, or simply replace the largest edge (u,v) with a 0 weight edge, or the algorithm to link s to another station that is not u or v is not correct. **-3 marks**

5. Using Solution 2 or variant: remove the largest edge (u,v) from T_0 but does not replace it to connect the 2 components created. **-4 marks**

6. Using Solution 1 or variant and just mention connecting the 2 components but no details given for the algorithm to do so. **-3 marks**

7. using Solution 1 or variant with details given for the algorithm to connect the 2 components but does not consider the free connections from source s correctly. **-2 marks**

8. other minor mistakes or minor missing details in the solution. **-1 mark**

9. Those who give multiple solutions and some of the solution is not correct. **-1 mark** for giving wrong solution.

If you get at least 1 mark for your solution, you will not get less than 1 mark even with the deductions.

Wrong solutions: **0 marks**

1. No answer
2. Simply just use Prim's/Kruskal's without describing how the graph is re-modeled (thus I will take it as running them on the original graph which will simply give T_0 or an MST equivalent to T_0).
    - Some students who give this answer assume the new 0 weight edges are already added to the array A so re-running Prim's/Kruskal's will give the correct answer, which is not correct. A only contain the edges of the original graph. Even assuming it does contain the 0 weight edges, the solution is still not correct. If s has 0 weight edges to every other station, running Prim's/Kruskal's will include all these edges, and exclude all other edges (since now you get a MST of cost 0 with edges from s to the other stations). However as per the problem description, s can only have 1 free connection to one other station.
3. Incomplete answer without enough information to determine the solution.
4. Solve this using SSSP/APSP algorithm.
    - A lot of such answers look for the most expensive SP in T_0 or the original graph and identify the largest edge in such a path to be removed. However the largest edge in T_0 need not necessarily be part of such a path.
    - Other students simply mistaken the problem to be a SSSP/APSP problem.
    - Can be correct when used to identify the largest edge in T_0 (although not efficient, so grade given according to time complexity. Graph traversal or simply going through AL will suffice).
5. Assuming the station that s should be linked to is given.
6. Remove s first from the original graph then re-run Prim's/Kruskal's on the remaining graph before adding s back in with a free connection.
    - Removing s can disconnect the graph into multiple components, thus Prim's and Kruskal's will not give a ST much less a MST. Adding back s with 1 edge is not guaranteed to link back the multiple components into a ST too (if there are more than 2 components).
7. incomplete solutions/very vague solutions with no details at all.
8. Other obviously wrong solutions (too many to be listed here).