Thomas Muamar

18-2:

a. When taking into consideration this with the insertion implementation needs to understand then when node X here is split into the two other nodes of let's say a and b the median for x when merged would be a new node of y. The height of this new node would technically remain unaffected unless we were to choose the root node as the case. If the X was root this would make the case of height[y] = height[X] +1. The two split nodes which are a and b would be equal to height[a] = max height[a] +1 and height[b] = max.height[b] +1. T. When updating the height for this tree it would take O(t) and the b-tree-insert method would then take as much O(th). The reason is because the insertion method makes h splits which is h is the height of the tree. Basically, the asymptotic run time would remain the same with just a few changes because of the methods of splitting and merging. Which means it would preserve the run time of the insert method. The deletion process would require much because the height wont change much unless the root would contain a single node. If the root node would be deleted then we would need to update the height of the tree by taking the height of the root – 1. The reason for this is because it would need to end up merging with the subtree nodes.

b. 1. Find the heights of both T1 and T2 trees
2. If h1> h2
3.   then would insert the node into right most node = R1
4. If R' is full
5.   take out largest key and fill k into the space recursively inserting the key into R1
6 .attach T2 into the right of k
7. if the h1 = h2
8. create new root with the key k
9. Then merge T1 and T2 to the left and right of the key k.
10. if h1< h2
11. insert key k to the leftmost node at height h1+1 of T2.

The first step if h1>h2 would take the time of h1-h2 +1 and the other steps would at most take constant time. So the total time ends up being |h1-h2| +1).

c. So we have the largest index a will be stored in xi which will be <= to k . When k' = x.largestkey t'i-1 be the tree whose root node consists of the keys in xi which are less than the largest keys contained in xi. In general, the rule is T'i-1 >= T'i. With S'' we have the keys that the nodes passed on p will be greater than k and the tree will be rooted at a node that contains the larger keys. When the node that has K is reached it will designate a tree instead of a key.

d. In order to split the tree we find the root to the keys first. Then we break the set s' into the sets of trees and keys that was described in part c. Then using the join operations we take t'I, ki and T'i-1. When the node for k is encountered then we join the x.key with t1 and x.key+1 qwith T2. So this ends up being 2 join operations being performed with the one insert of k. The height of the tree would just be the difference of t' – t. running time will most likely be O(2(H+H)) which is equal to o(lg n).

22-2:

a. A G_π is an articulation point if removing that vertex disconnects the graph. When a single point would split into 2 or more components. Let say that in this G_π we have an articulation at a certain point of let say x. if this point disconnects from the graph, then it will have two children within G. If it was only one child that was contained with G_π then this would mean that the one child would have a path to each of the other children of x. So when disconnecting X we have vertices that were going from one point to the next point containing X. let say that one point to the next point is a to b. In order to get to a that path will need to go through one of the children of X. If we had X that has two children of w and v then there would be no path between w to v with in g that doesn't go through X. So removing X would disconnect the components that have w and v. So this would make x an articulation point.

b. In order to prove that v which is a non root vertex of G_π is a articulation point we start with a value of let say X be an ancestor of v. If we were to remove V from G_π then we would end up have x become un reachable from the child of v which is s. So this would mean that the graph is disconnected and that would mean we have v as an articulation point. This is assuming we have a undirected graph that s takes us to the descendants of s and no descendants have back edges.

c. Basically using a recursive algorithm this can be computed.
if v is a leaf in G_π
then v.low = min(v.d) and min(w.d) where the back edge is (v, w).
else if v is not a leaf
v = min(v.d, w.d) where w is a back edge and u is a child of v
When computing the values of v.low it would be linear time. So the sum of the vertices gives twice the number of edges which would make the run time O(E).

d. For this portion we just have to take the algorithm in c which calculates v.low and then take that value and make a comparison between the two of v.low and v.d. Basically root would have to have at least two children and non root and there exist an edge (u,v) in that v.low is greater than or equal to u.d.

e. So, a bridge is an edge whose removal disconnects G. Basically if you have a edge that is (a, z) within the cycle and only if it exists at least in one path from a to z and this doesn't contain the edge (a,z). So when removing (a,z) it wont disconnect the graph meaning that it is not a bridge.

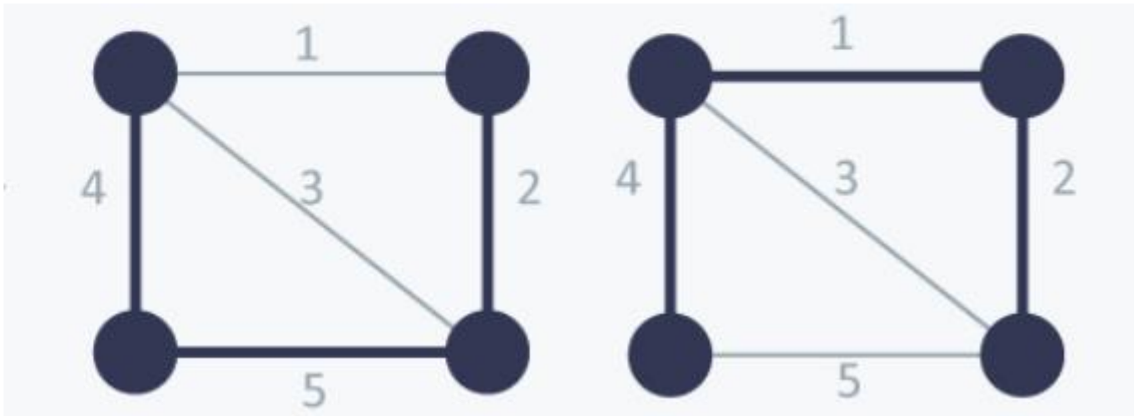
f. The way to compute all bridges in O(E) time is where we take into the consideration that if a we have a edge that has its endpoints are articulation points or having endpoints that have an articulation point and the other is a vertex of degree 1. This would be the cases that we would need to check for a bridge. Since previously I had figured out O(E) time for finding the articulation points then we can use that algorithm and figure out whether that edge would be a bridge.

g. So with biconnected points when we find the articulation point of let say v going from that point on from that articulation point will form a biconnected component. A biconnected component of G would be a set of edges that have any two edges in the set lie on the simple cycle. No vertex can be in two or more biconnected components of a graph. Saying that means that the biconnected components of G partition the edges of G.

h. Start by using the algorithm in part F which allows us to find the bridge edges in O(E) complexity. Then by removing each bridge from E the biconnected components become edges.

23-1:

a. Image reference: https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial/



Looking at the Right we have a minimum spanning tree that gives a value of 7. When you take this graph on the right the second best minimum spanning tree would be of a value of 9. There is two possibilities which is why it isn't unique . We have 2,3,4 and 1,3,5 for the second best minimum spanning tree.

b. For this problem we are basically trying to show that the one edge swap would cause the minimum spanning tree to become a second best minimum spanning tree. In order to get the second best minimum spanning tree we basically have a cut of a vertex from the rest of the edges. This edge that we cut will be called (x,y). The next edge that was selected when obtaining T is (u,v). Since T- {(u,v)} U {(x,y)} is the second best minimum spanning tree because it will be selecting a edge that is not shaded in which would end up making it a more expensive spanning tree. So for every edge that is cut we would just need to select that edge that was not shaded in.

c. The way we can fill the array max[u,v] is by first doing a search from each of the vertex u and not being able to visit the other edges that were previously visited in the spanning tree. The search would most likely be breadth-first step and this would end up making the time complexity here o(v^2).
I used the Breadth first search algorithm from the book in chapter 22 with modifications to it because we didn't need to compute a the values.

```
BFS(G,T, a)
arr[u, v] is the new table
For each vertex u E g.v
arr[u,v] = nil
Q = 0
Enqueue(q,u)
While q cannot equal 0
        X= dequeue(Q)
        For each v E g.adj[x]
                If arr[u, v] == nil and v cannot equal u
                        If x == u || a(x,v) > arr[u,x]
                                arr[u,v] = (x,v)
Else arr[u,v ] = arr[u,x]
Enqueue(q, v)
Return arr
```

With this algorithm the arr is the table that will record whether a vertex has been visited in the search.

d. The best way to compute this is by using what I basically said in b in order to find the second best minimum spanning tree by taking one edge and replacing it with an edge that is not selected. So, the algorithm would start by finding out the minimum spanning tree T which will take O(E+vLgv) by using prims algorithm then using a structure like priority queue which will take o(v^2) together. Then in part c we can use the table which takes o(v^2). Then finding the edge(u,v) which minimizes a(max[u, v]) – a(u,v) taking another O(v^2). Once edge is found then we set the value of T to be the second best minimum spanning tree which would give a total time of o(v^2)

Extra credit:

For this BW tree they are taking a new ARS which is atomic record stores that provided higher performance. The bw-tree design provides a higher latch free failure rate than a b-tree. This is depicted in the article when they compare the traditional b-tree architecture of the berkelyDB showing that the bw-tree is speed up compared to the traditional. This speed up is due to having amore latch free environment. The bw-tree has no thread blocks when it comes to updates and reads. Another aspect of a BW-tree that makes it great is the cpu cache efficiency shows to be superior to the B-tree. The reason for that is because it updates the pages with rarely invalidating the cpu caches of other threads. With a b-tree a page "involves including a new element in vector of key ordered records, on average moving half the elements and invalidating multiple cache lines." (Levandoski). The node split for a bw-tree employs a b-link atomic split which works in two phases. One phase is where the child is split and then the parent node is atomically updated with a new index term containing a new separator key and a pointer to the newly created split page. Another thing that the bw-tree is good at is the delta updating the preserves the prior page state and it improves processor cache performance. Another thing that sets BW-trees apart from normal b-trees is their pages. The BW-tree pages are logical which means that they do not occupy fixed physical locations or have a fixed size. The BW-tree pages have a PID index which is in the mapping table. Using the mapping table, it is able to use the PID to get the physical address of the current item. The reason there is no size limit is because the pages are elastic they basically just keep growing. The page search for a BW-tree involves going through a delta chain and stops at the first occurrence of the key. If the search fails and it doesn't have the key, then it will do a normal search like the b-tree which is the binary search. Also, when updating a page, it doesn't do it in place it just updates the delta record by prepending the existing page.

Reference:

1. J. J. Levandoski, D. B. Lomet and S. Sengupta, "The Bw-Tree: A B-tree for new hardware platforms," 2013 IEEE 29th International Conference on Data Engineering (ICDE), 2013, pp. 302-313, doi: 10.1109/ICDE.2013.6544834.