COMPSCI 2CO3

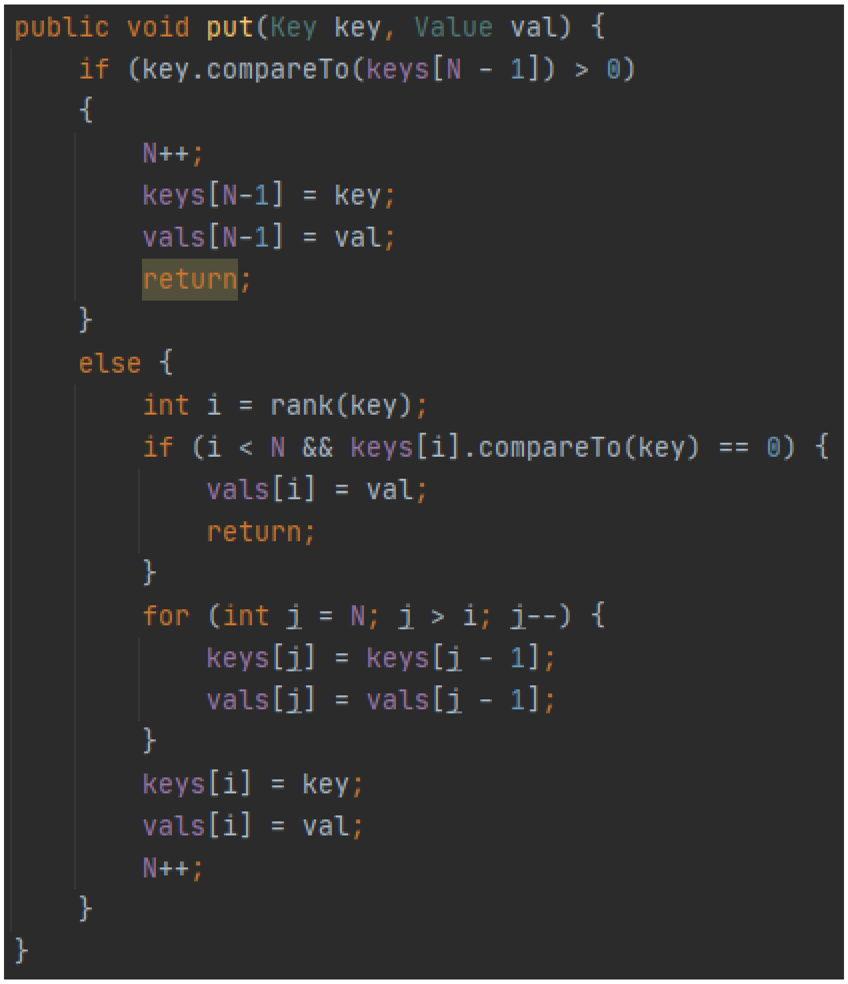
Mohammad Omar Zahir: zahirm1
Talha Amjad: amjadt1

March 15, 2021

SOLUTIONS

Question 1:

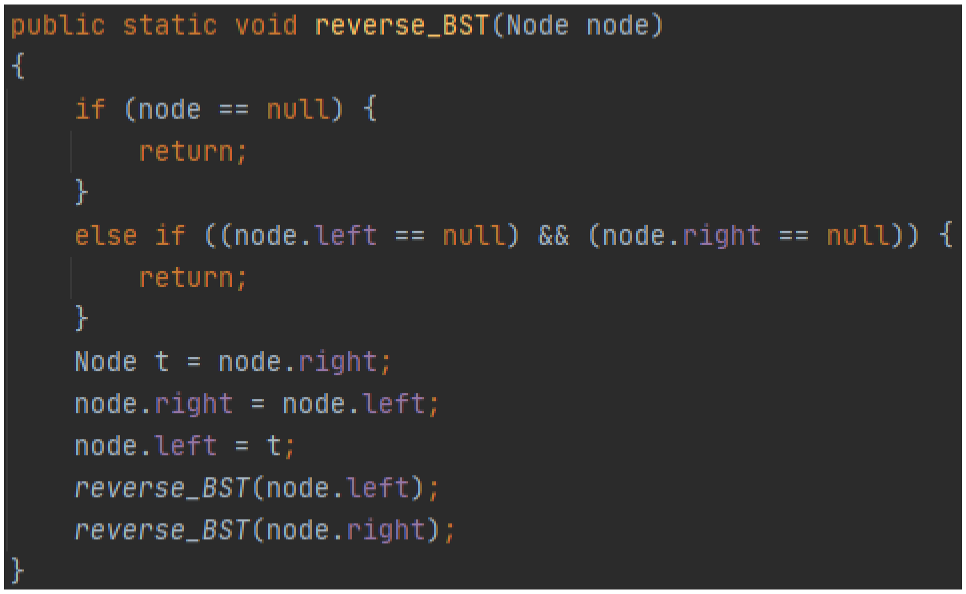
In this implementation, we have added the functionality for dealing with multiple inputs that are larger than all the previous key values. We check to make sure that the value is larger than the rightmost value, which would be at the end of the keys list and would be the largest value (since the list is ordered), and once it is, we simply add a new key with the given value at the end. Otherwise, the code behaves the same way as before for any other case. We know that this code will run in constant time in the case where the inputs are all in increasing order because that would mean that only the code within the first if statement would get executed, and as we can see, all the operations within the first if statement are all constant time operations, which would mean that inserting a new largest value each time will also always be done in constant time. In the case that the list is ordered, we will be doing this for n values, which would take linear complexity.



```java
public void put(Key key, Value val) {
    if (key.compareTo(keys[N - 1]) > 0)
    {
        N++;
        keys[N-1] = key;
        vals[N-1] = val;
        return;
    }
    else {
        int i = rank(key);
        if (i < N && keys[i].compareTo(key) == 0) {
            vals[i] = val;
            return;
        }
        for (int j = N; j > i; j--) {
            keys[j] = keys[j - 1];
            vals[j] = vals[j - 1];
        }
        keys[i] = key;
        vals[i] = val;
        N++;
    }
}
```

Question 2:

This algorithm assumes that the root node of the tree is a type of generic node class implementation with three distinct state variables, one for the data, the left node, and the right node. The function assumes that the node that is being given is the root node of the tree. Through this basic implementation, we can apply the following reverse algorithm to find the "reverse" of the regular binary search tree. The algorithm simply switches the children of each node of the tree, meaning each right node will become the left node and each left node will become the right node. This is then done recursively for all the nodes in the tree, by calling the function on both children. This ensures that all the nodes in the tree are looked over and are swapped to be in the order opposite to themselves, which would satisfy a symmetric property. We know that this algorithm will be done in linear time because the function itself only performs constant time operations for each recursive call, or each node in this case. As we have shown above that each node will be called on the function recursively once, that would mean that performing constant time calculations on n nodes would give us a complexity of O(n), or linear time.

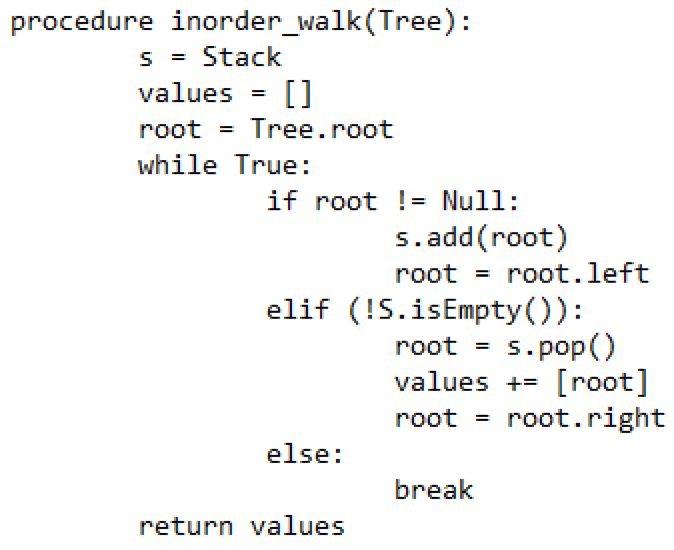


```
public static void reverse_BST(Node node)
{
    if (node == null) {
        return;
    }
    else if ((node.left == null) && (node.right == null)) {
        return;
    }
    Node t = node.right;
    node.right = node.left;
    node.left = t;
    reverse_BST(node.left);
    reverse_BST(node.right);
}
```

Question 3:

This algorithm assumes that the Tree object is a generic tree type which contains a root node that has the property of a generic Node type which has been described above where each node stores three distinct variables, the left, right, and value of the tree which acts as a node implementation where the values can be accessed. This way the Tree type contains a root node that has left and right nodes linked to it, effectively showing a tree, which the algorithm leverages.

This algorithm uses a stack, to perform its main function, and the algorithm does so as follows. The algorithm adds the root to the stack, and continues traversing to the left of all the nodes until it reaches the left-most node. Once the left-most node is reached, we will then add the node to the value array and then add the root node as well. We then begin iterating to the right of the node and perform the same recursive structure. As such, we can see that this code will perform the inorder walk that will start from the left most node and iterate all the way to the right most node.



```
procedure inorder_walk(Tree):
        s = Stack
        values = []
        root = Tree.root
        while True:
                if root != Null:
                        s.add(root)
                        root = root.left
                elif (!S.isEmpty()):
                        root = s.pop()
                        values += [root]
                        root = root.right
                else:
                        break
        return values
```

We know this algorithm will perform in O(n) time complexity as we can see that the while loop only iterates over all the nodes in the tree once, meaning that the iteration will be over the length of the tree, which would be n. We also know all the other operations, including the conditional statements and assignment statements that are within the loop are all done in constant time, which should not affect the linear complexity. Similarly, any of those operations that are done outside of the while loop would simply just add a constant value *c* to the final complexity, which would also not change the linear runtime.

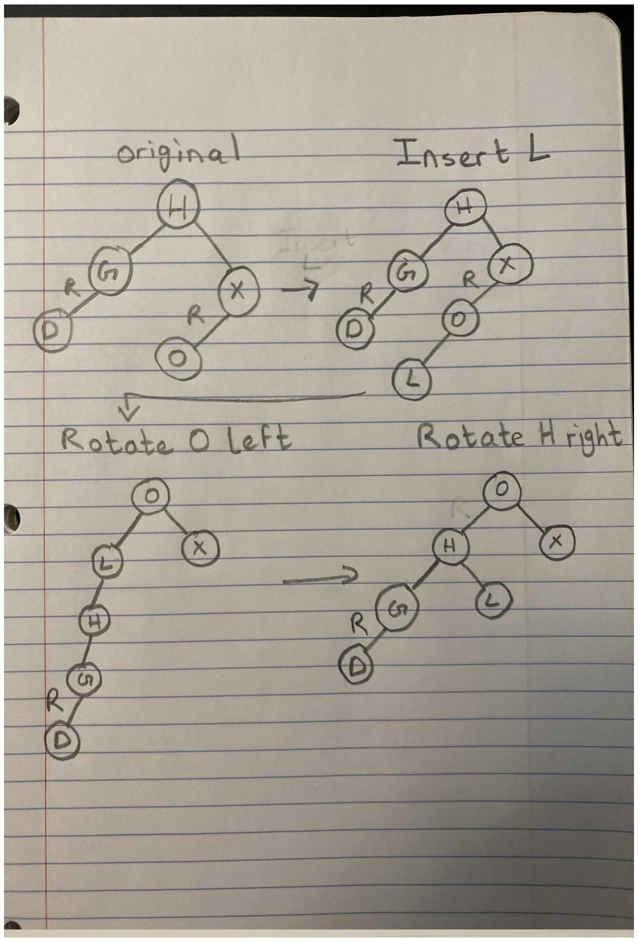
Question 4:

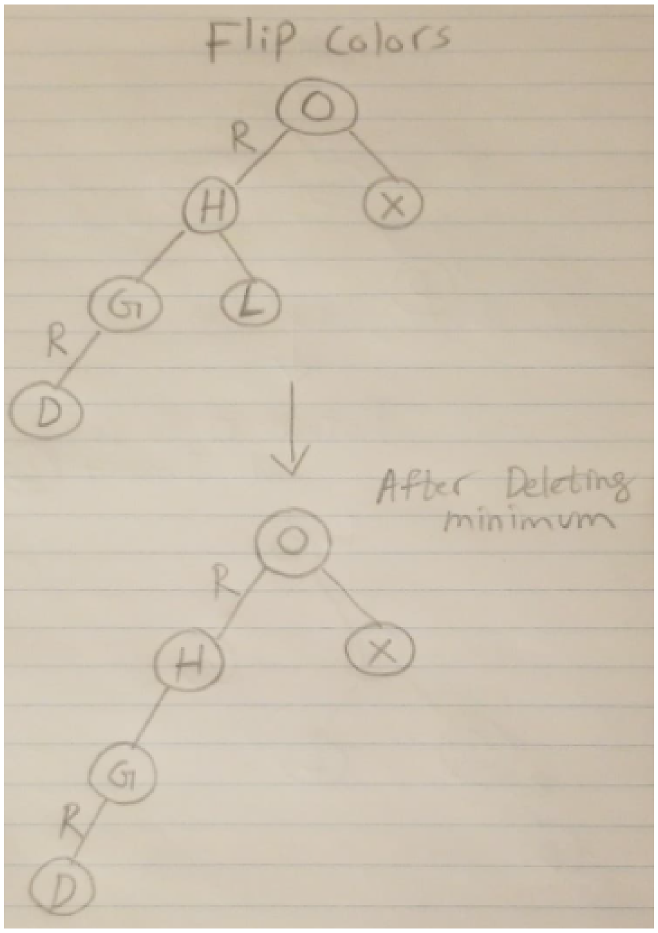
We know that a regular binary search tree has internal nodes equivalent to 2^(h) -1, where h is the height of the tree. Because the Red-Black tree is a variation of the binary search tree, we can apply the same formula for the number of internal nodes for the RBT. We also know that the maximum number of internal nodes will be created when the RBT is full. The height of an RBT with a maximum number of nodes is the sum of the red height and the black height because we know from the rules of an RBT that there cannot be two consecutive red edges, meaning that in the best case situation where we would maximise the amount of red edges as that would increase the amount of internal nodes, there will be an alternating number of red and black edges since a black edge must always separate two consecutive red edges. That would mean that the red height would be equivalent to the black height in the best, maximising case. This would mean that the height of the tree will be 2k, as it is the sum of the red and black heights, which are both k from our deduction. Therefore, we can deduce that the maximum number of internal nodes that are possible will be 2^(2k) - 1.

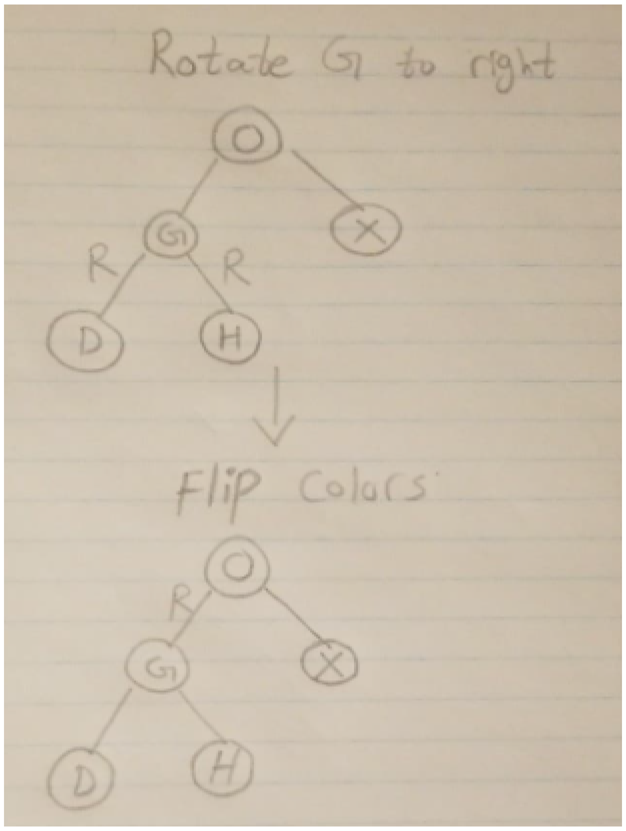
In the worst case where we would like to minimize the amount of internal nodes, we must also try to minimize the number of red edges, as the black edges are already given as k. This would be when there are no red edges and only black edges, which can still yield a valid red-black tree as there are no conditions on the minimum number of red edges in a red-black tree. For that reason, the black height *k* of a red-black tree is the total height of this red-black tree. Using the same formula for the internal nodes of a binary tree, we can replace h with k, which will give us 2^k - 1. This would be the minimum number of internal nodes that are possible in a red-black tree.

Question 5:

The definition of memory-less would imply that the data structure maintains the same shape before and after the data value has been inserted and deleted. For the Red-Black Trees to be memoryless, the data structure should be the same as the original data structure after the deletion of the data value. We can see in our example below that the binary tree before and after the insertion of the letter L is quite different from each other. We perform the adding and removal operations, as well as the color flipping and rotating operations. As we can see, the first tree before the insertion and the final tree after the removal are different, hence proving that the red-black tree is memory-less. It is also worth noting that L was not the minimum element in this case.

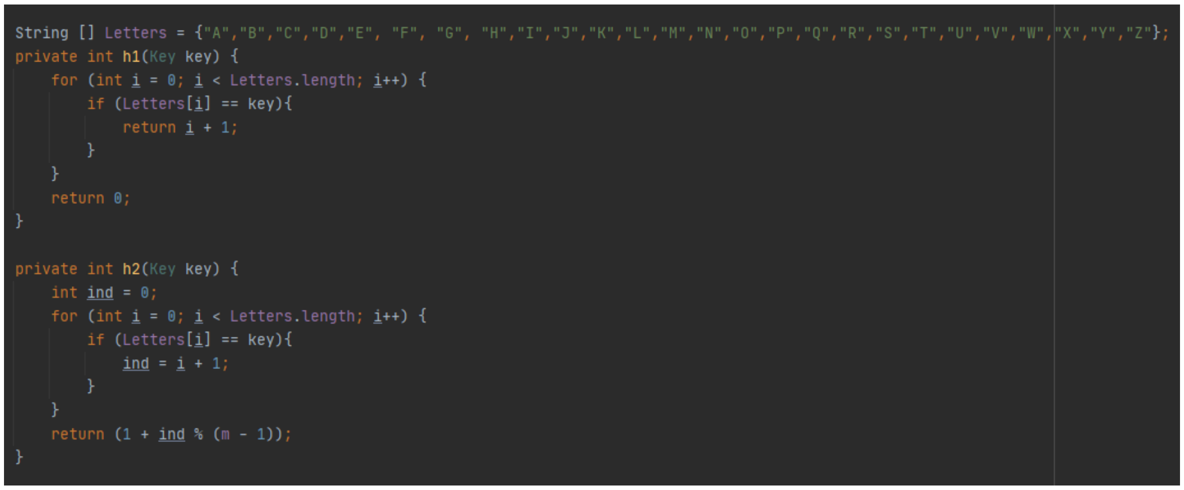

original
Insert L
Rotate O left
Rotate H right


Flip Colors
After Deleting minimum

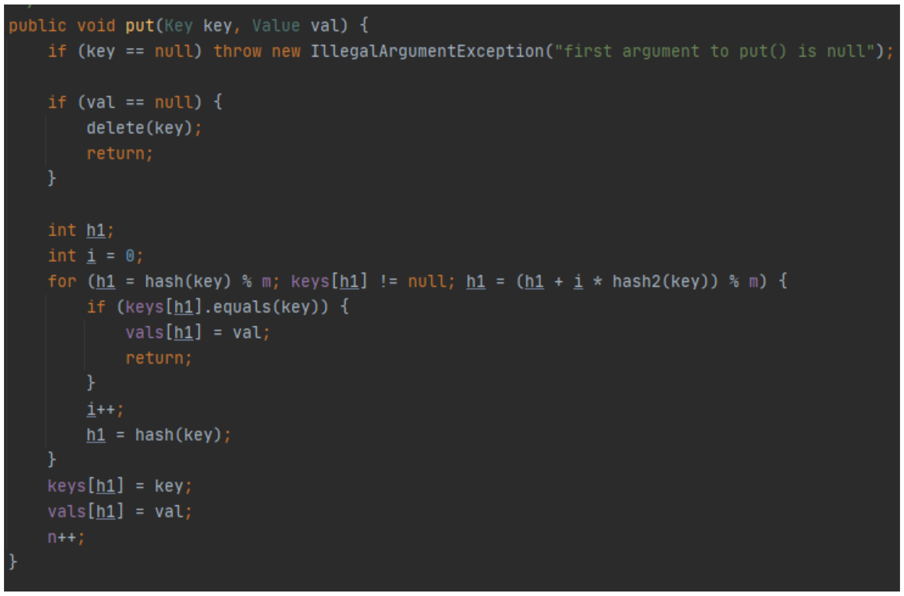

Rotate G to right
Flip Colors

Question 6:

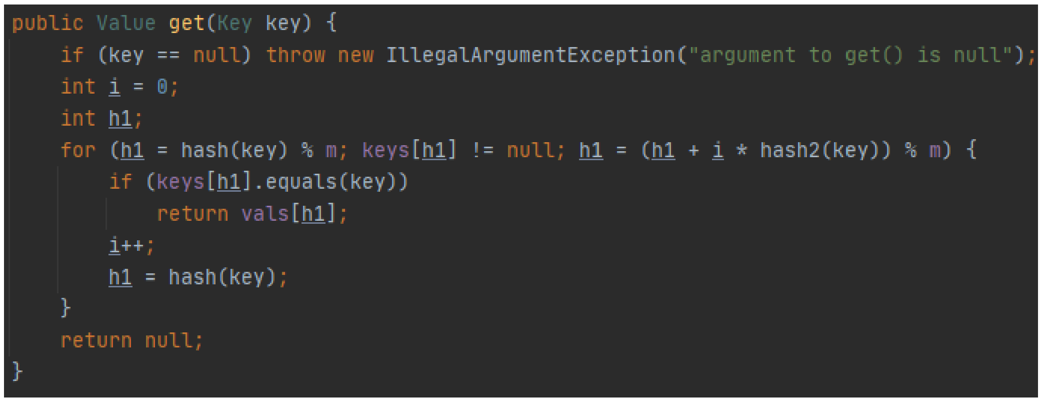
In the implementation that was changed from the given code, the first and second hash functions that were added were of the form that was seen in midterm 2 where h1 was made to be k, and h2 was made to be 1 + k mod (m-1). It can also be seen that while the inputs are taken in the form of alphabets, the implementation makes use of the alternate version where we map the corresponding letters to their equivalent number as described in the assignment specification, which can be seen in the h1 and h2 methods. It can also be seen that the resize code was removed from the original code as it was not needed. Lastly, some changes were made in the for loop for the put and get methods in order to incorporate for the second hash function.



```
String [] Letters = {"A","B","C","D","E", "F", "G", "H","I","J","K","L","M","N","O","P","Q","R","S","T","U","V","W","X","Y","Z"};
private int h1(Key key) {
    for (int i = 0; i < Letters.length; i++) {
        if (Letters[i] == key){
            return i + 1;
        }
    }
    return 0;
}

private int h2(Key key) {
    int ind = 0;
    for (int i = 0; i < Letters.length; i++) {
        if (Letters[i] == key){
            ind = i + 1;
        }
    }
    return (1 + ind % (m - 1));
}
```



```
public void put(Key key, Value val) {
    if (key == null) throw new IllegalArgumentException("first argument to put() is null");

    if (val == null) {
        delete(key);
        return;
    }

    int h1;
    int i = 0;
    for (h1 = hash(key) % m; keys[h1] != null; h1 = (h1 + i * hash2(key)) % m) {
        if (keys[h1].equals(key)) {
            vals[h1] = val;
            return;
        }
        i++;
        h1 = hash(key);
    }
    keys[h1] = key;
    vals[h1] = val;
    n++;
}
```



```
public Value get(Key key) {
    if (key == null) throw new IllegalArgumentException("argument to get() is null");
    int i = 0;
    int h1;
    for (h1 = hash(key) % m; keys[h1] != null; h1 = (h1 + i * hash2(key)) % m) {
        if (keys[h1].equals(key))
            return vals[h1];
        i++;
        h1 = hash(key);
    }
    return null;
}
```

b)
```
h₁(k) = k, n₂(k) = 1 + (kmod(m-1))
m = 11
h(k,i) = (k + i(1 + (kmod(m-1)))
```
Insert E → 5
```
h(5,0) = (5+0)mod11 = 5
```
Insert A → 1
```
h(1,0) = (1+0)mod11 = 1
```
Insert S → 19
```
h(19,0) = (19+0)mod11 = 8
```
Insert Y → 25
```
h(25,0) = (25+0)mod11 = 3
```
Insert Q → 17
```
h(17,0) = (17+0)mod11 = 6
```
Insert U → 21
```
h(21,0) = (21+0)mod11 = 10
```
Insert T → 20
```
h(20,0) = (20+0)mod11 = 9
```
Insert I → 9
```
h(9,0) = (9+0)mod11 = 9 (Taken increment i)
h(9,1) = (9+1(1+9mod10))mod11 = 8 (Taken increment i)
h(9,2) = (9+2(1+9mod10))mod11 = 7
```
Insert O → 15
```
h(15,0) = (15+0)mod11 = 4
```
Insert N → 14
```
h(14,0) = (14+0)mod11 = 3
h(14,1) = (14+1(1+14mod10))mod11 = 8(Taken increment i)
h(14,2) = (14+2(1+14mod10))mod11 = 2(Taken increment i)
```

Final output Array

0	1	2	3	4	5	6	7	8	9	10
	A	N	Y	O	E	Q	I	S	T	U

Question 7:

A connected graph is such that there is a path connecting every node in the graph to every other node in the graph. To prove that every node in the graph has at least one vertex whose removal will not affect the graph, meaning every node that was connected to each other before will still be connected to each other after the removal, we can take the following example. We must take two nodes in the graph, a, c, such that the distance between them is the maximum possible distance there can be between any two nodes in the graph. Once we have found two such nodes, we can take any single other node, b, in the graph at random, and compare its distance to c. We can see that as the distance between a and c was maximized, the distance between b and c will either have to be less than or equal to the distance between a and c. As such, there are two possible situations that can arise from this scenario.

If it is less than the distance, which it will be in most cases, it is safe to remove the node a as it will not affect connectivity. This is because the path between node b and node c is unaffected by the presence or absence of node a as it is a smaller distance between node a and c, meaning it is impossible for node a to be in the path, since the distance was originally maximised from the entire graph. As such, the removal of such a node will not affect connectivity.

If the distances are equal, that would imply that removing either node a or b would be equivalent, as we can apply the same logic from the first scenario. If we have two nodes a and b such that they are the same maximum possible distance away from another node c, that would mean that node a is not present in node b's path and vice versa. For that reason, the removal of either node a or b would be equivalent, and this logic can be applied for any number of nodes that are an equivalent maximum distance away from a node c. As such, from this we can see that there will always be a node whose removal will be possible in a graph to maintain connectivity. Similarly, we have also seen that if there is more than 1 such pair with the same maximum distance, removing a node from any such pair will suffice and maintain connectivity.

This logic is applied in our DFS method below which identifies all possible paths in the graph with the longest length, and removes the first node from the first path it identifies.

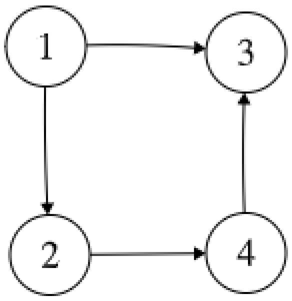
Due to the nature of the problem, it does not matter if the graphs are directed or undirected, as this will not have an impact on connectivity.

```
procedure DFS_method(G, vrtx, visited, path):
    visited += [vrtx]
    paths = []
    for adj in G[vrtx]:
        if !(adj in visited):
            adj_path = path + [adj]
            paths += [adj_path]
            temp = visited.copy()
            paths += DFS(G, adj, temp, adj_path)
    return paths
```

```
procedure single_vertex(G):
    every_path = []
    for i in [0..len(G)]:
        if G[i] != []:
            every_path += DFS(G, i, [], [i])
    max_len = 0
    for path in every_path:
        if len(path) > max_len:
            max_len = len(path)
    max_paths = []
    for n_path in every_path:
        if len(n_path) == max_len:
            max_paths += [n_path]
            return n_path[0]
```

Question 8:

The algorithm does not guarantee a topological error mainly due to the fact we are searching using BFS rather than DFS. Unlike DFS, BFS ensures that we exhaust all possibilities that are directly connected with our respective source node, and then recursively do the same to all the nodes thereafter. As such, that would create an order where all the starting nodes are closest to the source node respectively, or increasing distance from the source node, as described in the question. This does not guarantee a topological order as this searching could lead to a node that is dependent on a node further down, but comes after in the ordering, hence not making it a topological order. This situation can be described in the following image. Taking 1 as the source node, we can see that following the BFS, we will first go to the 2, and then 3, as we must go to all the nodes attached to 1, adding them to the queue. After all nodes directly connected to 1 have been exhausted, we will then go to the next value that is stored in the queue, which is 2. From 2 we can only go to 4, which will give us the final order of 1,2,3,4. This however is not a topological order as we can see that 3 is dependent on both 1 and 4, meaning it should come after both these nodes in a topological ordering, but 4 comes after due to the BFS. The proper topological ordering would be 1,2,4,3, which would be achieved through a DFS.


$$\begin{tikzcd} 1 \arrow[r] \arrow[d] & 3 \\ 2 \arrow[r] & 4 \arrow[u] \end{tikzcd}$$