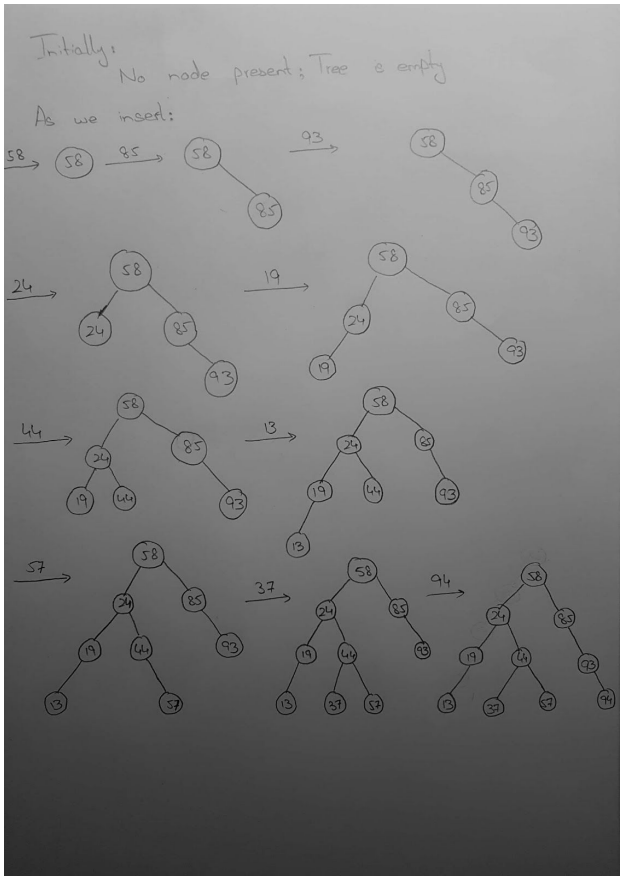
Question 1

a) Insertion:

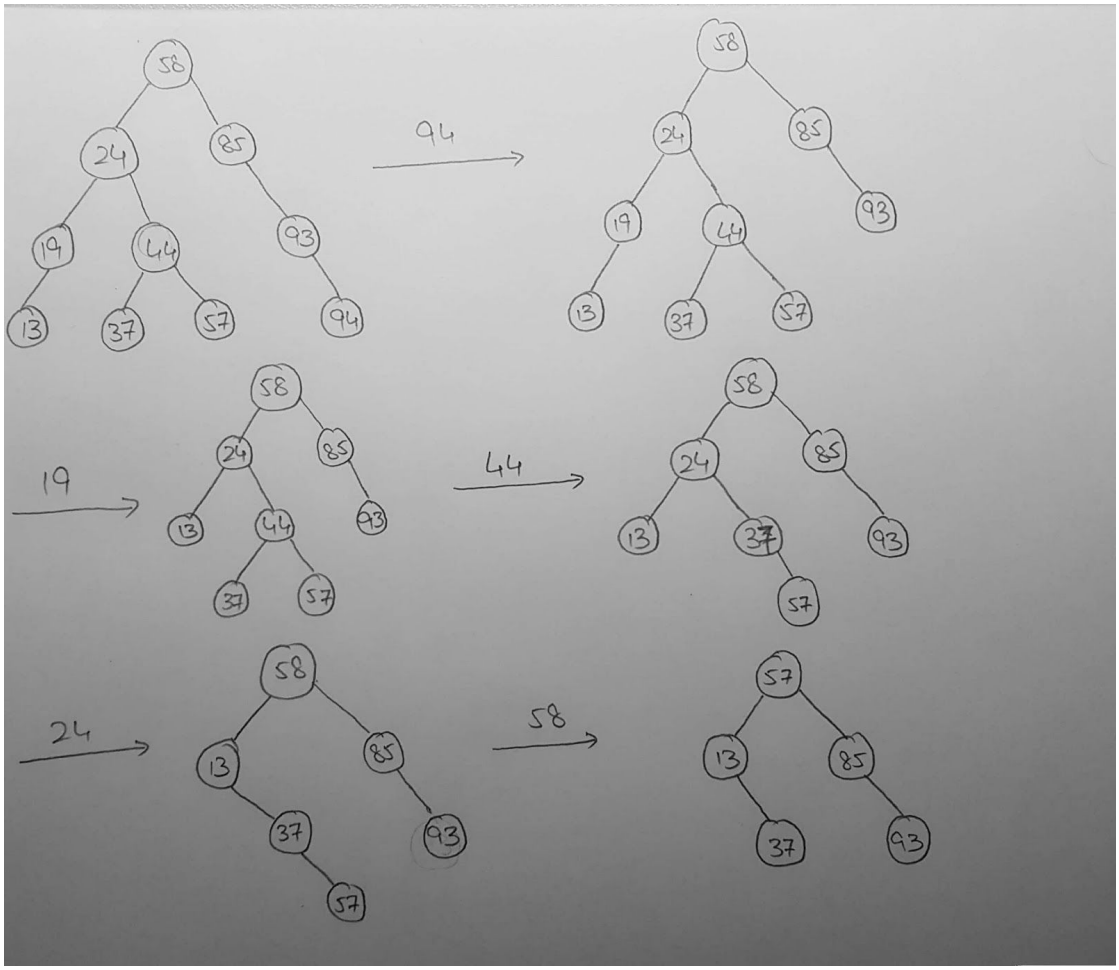


b) **Preorder**:
58, 24, 19, 13, 44, 37, 57, 85, 93, 94

Postorder:
13, 19, 37, 57, 44, 24, 94, 93, 85, 58

Inorder:
13, 19, 24, 37, 44, 57, 58, 85, 93, 94

c) Deletion:



Question 3

Functions are analyzed separately. In case one function calls another, the time complexity of the caller is determined in accordance with the callee function.

The DecisionTreeNode class is a simple implementation of a node of a decision tree. It contains the feature index used for decision, a class label (applicable if it is a leaf node), and pointers to the left and right child of the node. It has simple constant time functions. The constructor simply initializes each variable. isLeaf() checks if the node is a leaf, setter functions for each data variable sets the respective data variable as the parameter of the function defines, and the getters return the values of the variables. All these operations are constant time operations and therefore these are O(1).

The DecisionTree class implements the decision tree. It contains functions required to be implemented in the homework, as well as additional helper functions. A destructor is

implemented to prevent memory leaks. The destructor calls a helper function destory(DecisionTreeNode* root) that destroy a binary tree recursively by calling the function for each subtree until a leaf is reached. Since this operation requires visiting each node, the time complexity for the operation is O(N). calculateEntropy simply calculates the entropy at each node based on the entropy formula provided in the homework question ($H(P) = -\sum_i P_i logP_i$). If the number of classes is equal to C, the time complexity of the calulateEntropy function is O(C). caclulateInformationGain function computes the information gain for a specific feature by using the equation for information gain provided in the homework question ($H(S) = P_{left}H(left) + P_{right}H(right)$). At each node, it calculates the entropy for that node and the entropy of a potential split. The entropy at each node is calculated by the number of samples at that node, and the entropy for a potential split is calculated based on the number of samples on the left and right node after the potential split would occur. As far as the time complexity for this function goes, it is O(S) + O(C), where S is the number of samples and C is the number of classes in the data set. To implement the train method, at first a usedSamples array is created that keeps track of all the samples that have been used to train so far. Then, for each unused feature, the information gain is calculated and the maximum information gain is identified. The index of the feature in the features array which provides the maximum information gain is stored. This feature is used to split at the node, since it provides the maximum information gain. After creating the root, a featuresUsedSoFar array is created to keep track of the used features. Hence, the feature that is used to split at the node is marked with a boolean true in the featuresUsedSoFar array. After this, we create two separate arrays for the left used samples and the right used samples. We also create two boolean flags namely, leftLabelCheck and rightLabelCheck, and assign their values to -1. As we iterate through the samples, it is checked if the feature value at this node is 0. If it is 0, we go to the left node, and if it is one we go to the right node. In the same loop, the purity of the sample is checked. If the samples on the left have the same label, the leftSampleIsPure is set to true, otherwise false. We do the same for the right subtree. If the feature value is 1, we go to the right and repeat the same as we did for the left subtree. After checking if the samples are pure, the values of the flags leftSampleIsPure and rightSampleIsPure are used to check if the samples are pure. If they are not pure, we call the trainHelper function for the samples at the node. This function carries out the same operations as when the root was created for a feature, except that while computing information gain it does not take into account the features that have already been used. This is to ensure that once a feature is used, it is not used again. If the sample at the end of this function is not pure, the function is called recursively to split the dataset further until the samples are pure. When we reach the point where the samples at each node are pure, we set the leaf classes and the decision tree is made. The time complexity for the train operation is O(FC) + O(F) + O(S), so O(FC) + O(S), where F is the number of features, C is the number of classes, and S is the number of samples. The predict function is implemented by checking the feature the node uses to make a split. If the feature is 0, we traverse the left subtree and check the next feature; however, if the feature is 1, we traverse the right subtree and check the next feature. The time complexity for the predict function is O(N), which might happen if we have all nodes as either all left or all right descendants of the root. The test function iterates over all the

samples, and for each sample calls the predict function. The number of correctly predicted samples are counted and is divided by the number of samples to get the accuracy. The time complexity for the test function is O(NS), where N is the number of nodes and S is the number of samples. The test function with the file name as a parameter, parses the file and calls the previous test function. The time complexity for this function is O(SF) + O(NS), where S is the number of samples, F is the number of features, and N is the number of nodes. The print function of the decision tree calls a printHelper function that takes in the root and the number of tabs as parameters. The printHelper function recursively prints the tree using a preorder traversal. We check if the node is a leaf, in which case we print the class label, and if the node is a splitting node, we print the feature index used to split the dataset. Since each node is visited, the time complexity for the printHelper is O(N), where N is the number of nodes. The time complexity for the print function is also O(N).