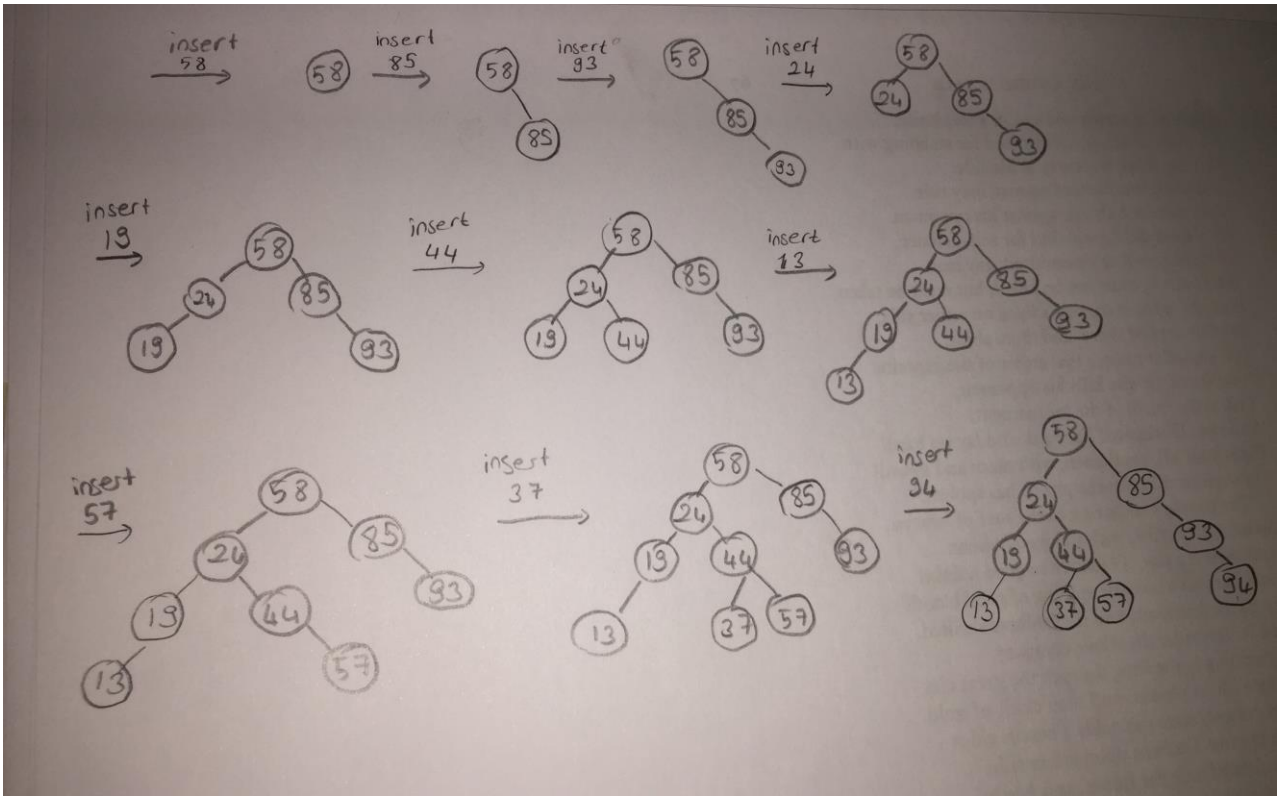
/**
* Author: İlknur Baş
* ID: 21601847
* Section: 2
* Assignment: 2
* /

Question 1

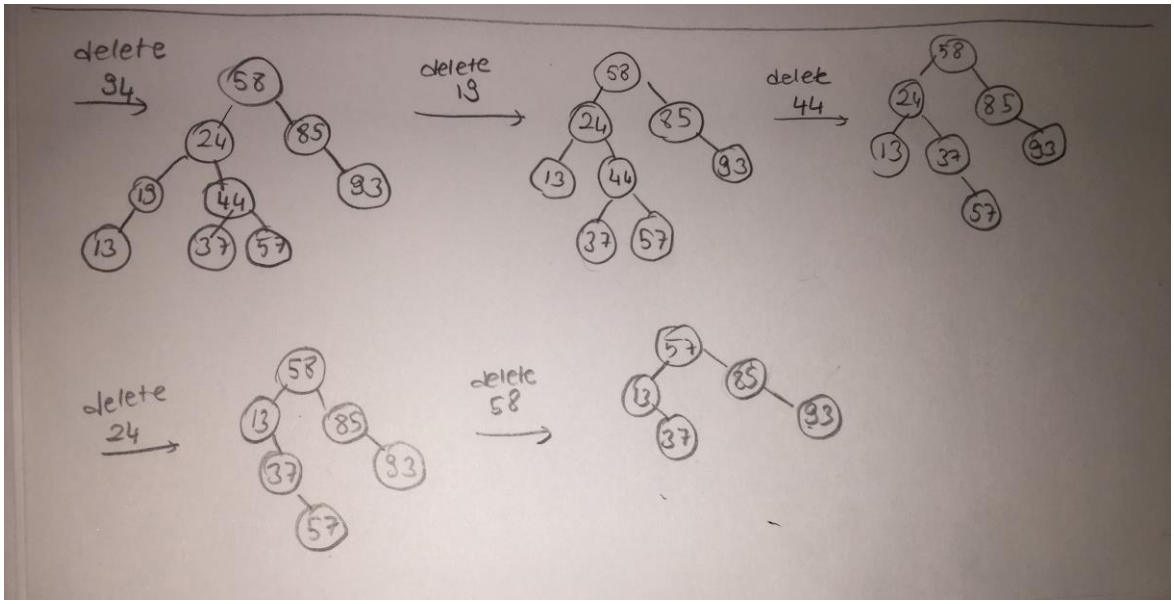


Insertion

a)

b)
Preorder: 58 24 19 13 44 37 57 85 93 94
Inorder: 13 19 24 37 44 57 58 85 93 94
Postorder: 13 19 37 57 44 24 94 93 85 58

c)



Deletion

Question 2

(a)
In order to calculate entropy, we need the value of the parameters (will be explain in detail later) which are come from *calculateInformationGain* function. In the **double calculateEntropy(const int* classCounts, const int numClasses) function**, basic calculations are done, like the given example in the end of the pg4 of the assignment. Since in my code, there are 2 separate for loops and each of these time complexity is $O(n)$, in total *calculateEntropy* function's time complexity is $O(n)$ where n is *numClasses*.

(b)
***calculateInformationGain(const bool** data, const int* labels, const int numSamples, const int numFeatures, const bool* usedSamples, const int featureId)* function** calculates information gain for a given feature id in order to split a node. In the assignment it is written that parent entropy and left and right child entropy for a potential split must be known in order to calculate information gain.
This piece of code finds the number of each class of the samples for a parent node in the *usedSamples[]* array which is used by *calculateEntropy* function.

```
for (int i=0; i< numSamples;i++) {
     if (usedSamples [i] == true ){
        if ( labels[i] == 1 ) //class no
           classCountsParent[0] ++;
        else if(labels[i] == 2 ) {
           classCountsParent[1] ++;
        }else
           classCountsParent[2] ++;
```

```
    }
}
```

Similarly, *int *classCountsLeft* and *int *classCountsRight* are initialized in order to find the entropy of the potential split with given *featureId*. In the left child, the samples's values with given *featureId* must be 0, in the right 1. Time complexity of this function again because of the for loops, $O(n)$ where n is *numSamples*. (*calculateEntropy* function is called in this function however we can ignore its time complexity since numSamples> numClasses)

(Note: Also I am aware of that this function won't execute correctly when there is more than 3 classes.)

(c)
In the DecisionTree.cpp class, I did the implementation of the *bool isLeaf() const* function which is used to determine whether a node is leaf. Also did the implementation getter&setter functions.

(d)

- In the ***void train(const string, const int, const int) function***, *ifstream* has been used in order to parse the text file. Then the datas are sent to *bool **traindata inn order to form* 2D array. Since this will done using nested loops, this parts time complexity will be $O(numFeatures * numSamples)$. But there is also another function that is called *train((const bool**)traindata,labels, numSamples, numFeatures)* its time complexity should be included (it is calculated in the following lines more detailed) which is $O(nlogn)$ where n is number of samples. Hence, this function's time complexity is $O(numFeatures * numSamples)$.

(Note: Also in this function, where the reading and parsing the text file happens, some error can occur due to the fact that my code only works when there is 3 different classes.)

- ***void train(const bool**, const int*, const int, const int*) function** is used as a helper function to carry out the recursion and also for printing the tree.
The piece of code is used for the following reason: At first, when no split is occurred, the all samples are must be consider for the calculation of information gain. Hence, the *usedSamples* array elements must be initialized with 1.

```
bool *usedSamples = new bool[numSamples];
for ( int i =0; i<numSamples; i++ ){
  usedSamples[i]=1;
}
```

Since, at the very beginning where no node is split, all feature ids is an option for a split decision of a node, to determine the selection of a feature id, I have defined an array when shows whether the specific feature id is used. This array is needed because, we cannot use same feature ids in the same path. The following code is written for this reason.

```
bool *checkLeafFeatureId=new bool[numFeatures];
for(int i=0;i<numFeatures;i++){
```

```
    checkLeafFeatureId[i]=1;
}
```

It calls the *train((const bool**)data,labels, numSamples, numFeatures, usedSamples, checkLeafFeatureId, root)* function in order to build the tree recursively in that function. Its time complexity determines according to the function that is called inside.(Each of these functions' (print() and void train(const bool** data, const int* labels, const int numSamples, const int numFeatures,bool *usedSamples,bool *checkLeafFeatureId, DecisionTreeNode* &root)) time complexities are explained in further lines.) Hence, $O(nlogn)$ where n is number of samples because of the train function that is called.

- Inside the **void train(const bool** data, const int* labels, const int numSamples, const int numFeatures,bool *usedSamples,bool *checkLeafFeatureId, DecisionTreeNode* &root),** I have additionally define 2 functions called : ***calculateleftSample*** and ***calculateRightSample.*** The reason is that after finding the feature id with biggest information gain, split will occur. These two functions will determine the left samples and right samples after the split. I have need these function since we build the tree recursively by calling ***void train(const bool** data, const int* labels, const int numSamples, const int numFeatures,bool *usedSamples,bool *checkLeafFeatureId,DecisionTreeNode* &root)*** function. usedSamples parameter have to change every time the function calls itself. There are lots of for loops, in total we can write this function's time complexity as follows $T(n) = 2T(n/2) + O(n)$ where n is number of samples. So its complexity is $O(nlogn)$.(assuming number of features is less than number of samples.)

(e)

int predict(const bool*) is used as a helper function to do the actual prediction. Its time complexity will be determined by the function that is called inside.I have noticed that I need a node as a parameter in order to trace the decision tree from the root. For that reason, the ***int predict(const bool* data,DecisionTreeNode* a)*** function is called for finding the class of a given sample which are extracted from the test data text file according to the tree that has been built. When it is reached to the leaf(reaching node is a must), that means the prediction of a sample is done. Finding the class is done recursively as it can be seen from the following code block.

```
if ( data[a->featureId]==false ) {
   prediction= predict((const bool*)data, a->leftChildPtr);
}else{
   prediction =predict((const bool*)data, a->rightChildPtr);
}
```

I can say that its time complexity will be determined by the height of the tree. Since the tree is tracked node by node and eventually will assign a class to a sample, its time complexity will be $O(logn)$ where n is numOFFeatures in average/worst case. The path for prediction can be decrease due to the numOFFeatures that are used in this path so the time complexity can be changed. Assume after first split, left pointer reaches a leaf. In this case time complexity can be $O(1)$.

(f)

•***double test(const string, const int)*** function reads and parses the test data text file and calls the *double test(const bool**, const int*, const int)* function. I needed the number of features (Note: maybe with another more efficient algorithm, there may be no need). *Reading text file* will done using nested loops, this parts time complexity will be $O(numFeatures * numSamples)$. But there is also another function that is called inside it which is double test(const bool**, const int*, const int).Its time complexity should be consider.In total, ***double test(const string, const int)*** function's time complexity is $O(numFeatures * numSamples)$.
(Note: Also in this function, where the reading and parsing the text file happens, some error can occur due to the fact that my code only works when there is 3 different classes.)

•In ***double test(const bool**, const int*, const int) function***, nested loops are used for finding the prediction for each sample as it can be seen from below piece of code. In other words, inside of the for loop predict function is called. Predict class's time complexity is $O(logn)$ and the inside for loop's is $O(m)$ where m is number of features. Like I mentioned, this is nested loop, so ***double test(const bool**, const int*, const int) function 's*** time complexity is $O(numFeatures * numSamples)$.

```
for (int i=0;i<numSamples;i++) {
     dataEachSample = new bool [getNoOfFeatures()];
     for (int j=0; j<getNoOfFeatures();j++){
        dataEachSample[j] = data[i][j];
     }

     int prediction = predict((const bool*)dataEachSample);
     delete[]dataEachSample;
     cout<<"prediction TEST:"<<prediction <<endl;
     if ( prediction == labels [i] ){
        truePrediction ++;
        cout<<"TRUE"<<truePrediction;
     }
}
```

(g)

void print() *function* is used as a helper method for the actual printing. void ***printTree(DecisionTreeNode* decisionTreeNode, int height)*** *function* is where the actual printing happens recursively. Since it is recursive and there occurs a for loop in the function, we can write its time complexity as follows $T(n) = 2T(n/2) + O(n)$ which is $O(nlogn)$**.**