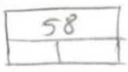
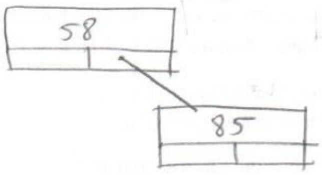
Question 1

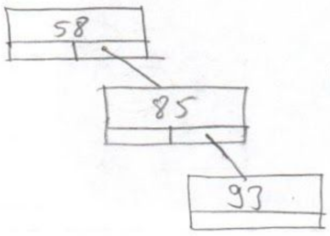
a) Insert 58 →



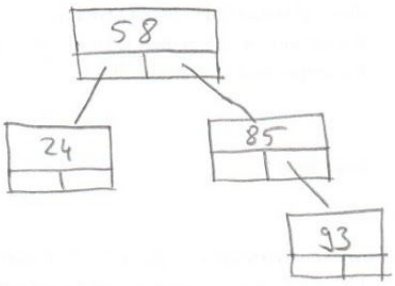
Insert 85 →



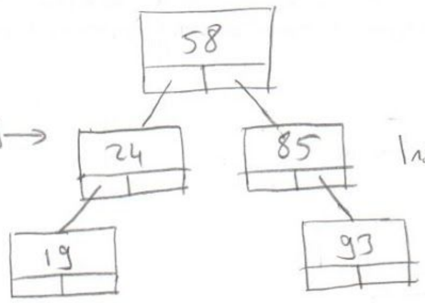
Insert 93 →



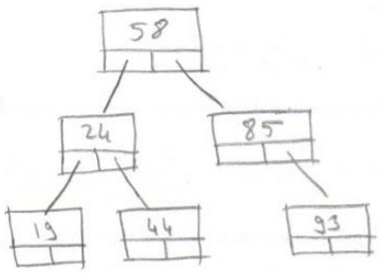
Insert 24 →



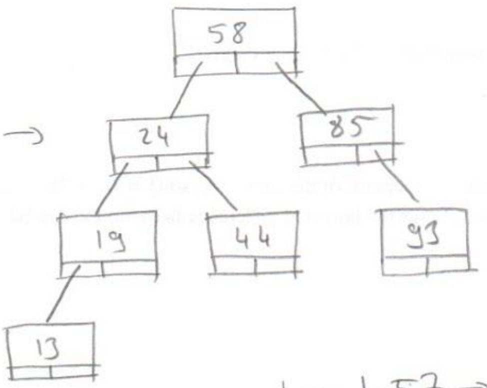
Insert 19 →



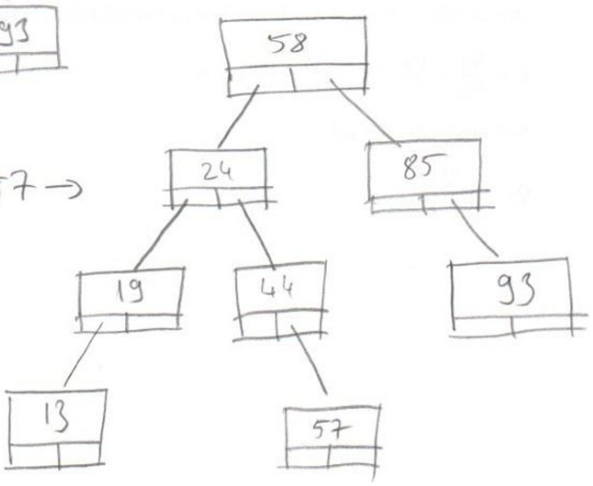
Insert 44 →

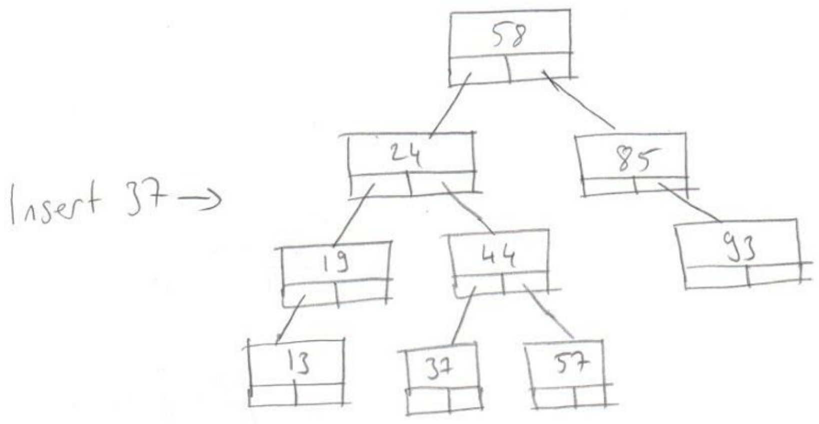


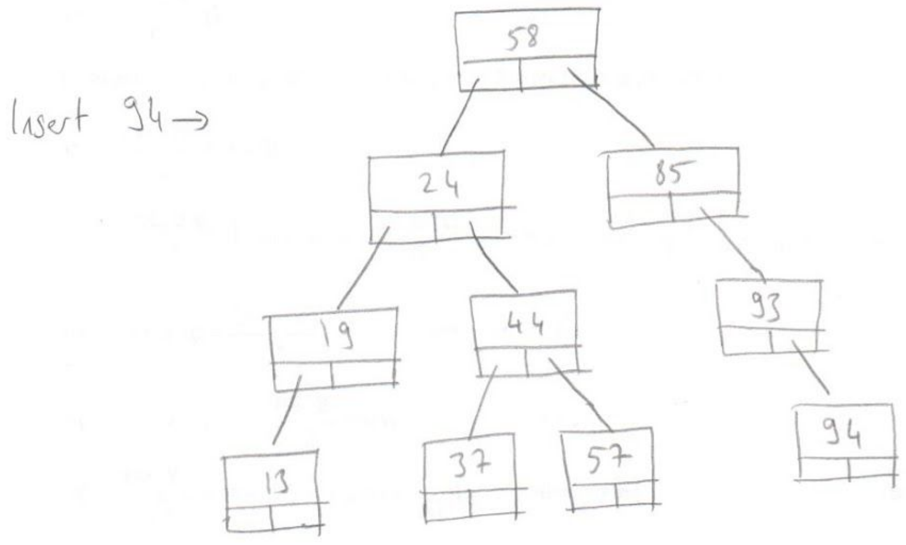
Insert 13 →



Insert 57 →







Question 1

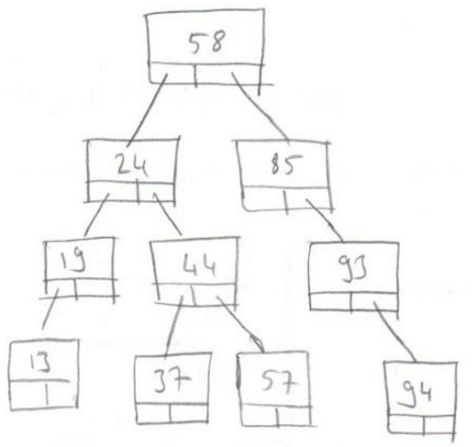
b) Preorder Traversal (Root - Left Child - Right Child)
58 - 24 - 19 - 13 - 44 - 37 - 57 - 85 - 93 - 94

Inorder Traversal (Left Child - Root - Right Child)
13 - 19 - 24 - 37 - 44 - 57 - 58 - 85 - 93 - 94

Postorder Traversal (Left Child - Right Child - Root)
13 - 19 - 37 - 57 - 44 - 24 - 94 - 93 - 85 - 58

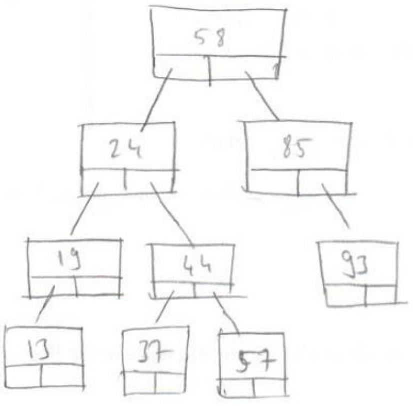
Question 1

c)



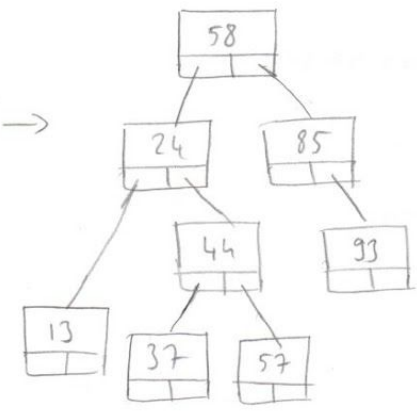
is the given tree.

Delete 94 →



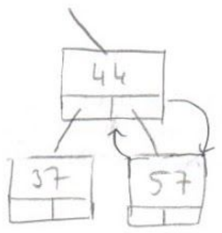
no swap need because 94 is leaf.

Delete 19 →



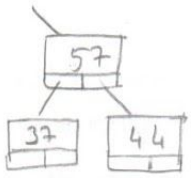
no swap need because 19 has only one child. 19's parent (24) is now pointing 19's only child which is 13.

Delete 44 →



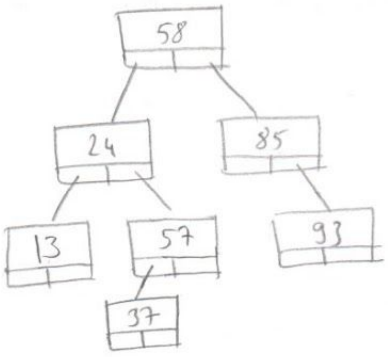
44 have 2 child. Therefore before deleting we need to swap with right child's left most node.

⟹



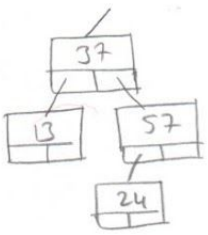
now we can delete 44.

⟹



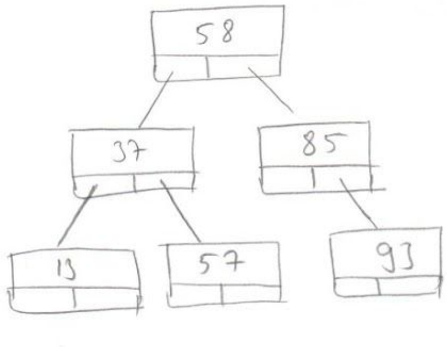
is the current tree.

Delete 24 → 24's right child's left most node is 37. Swap 24 with 37.



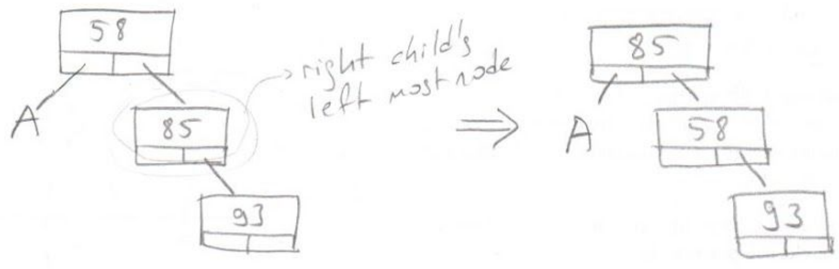
now delete 24.

⟹

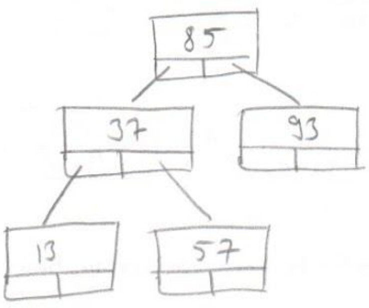


is the current tree.

Delete 58 → Again 58 has 2 child. We need to swap with right child's left most node.



⇒ 58 has only one child. 58's parent (85) is now pointing 58's only child which is 93.

⇒  will be the final binary search tree.

Question 3

In question 2 our objective is to find the best split for the data. To do this, we need to find the diversity of the data. According to that diversity we need to find the not previously used most significant feature for the data. Therefore, for finding the best split, we have to calculate Information Gain. With choosing the feature that gives maximum Information Gain for the current data will be the best split.

Before explaining the functions, we know that the labels are bigger than 0 and feature starts from 0. Thus, rather than using new variable we can use negative numbers for classes and other numbers for features. Hence, in DecisionTreeNode we can find whether the node is leaf or not. In DecisionTreeNode:

Our members are:

```
        DecisionTreeNode* left; // which points to the left child (if feature is 0)
        DecisionTreeNode* right; // which points to the right child (if feature is 1)
        int value; // holds the feature and class Id.
bool DecisionTreeNode::nodeDecide(){
   if(value < 0) // It's a class because value is negative
      return true;

   return false;
}
```

Now, let's look at the train functions.

```
void DecisionTree::train(const bool** data,.....){
        // Create new bool with the size of numFeatures to check whether the feature
        // is used or not. Initially featureUse[i] = false; - non of the features are used
        bool* featureUse = new bool[numFeatures];

        // Create another bool array for which data is currently on the node
        // Initialize an int to indicate whether all the features are used or not

        // call the trainTree function with train's parameteres, array's, number of
        // available features and root

        trainTree(data, labels, numSamples, numFeatures, numFeatures, used,
        featureUse, root);
}
.
.
.
```

```
void trainTree(const bool** data, const int* labels, const int numSamples, const int
numFeatures, int current, bool* used, bool* featureUse, DecisionTreeNode* node){

    // Check whether there is available feature. If not make this node a class with
    // most common data
    if(current <= 0) {
        // find the maximum repeated label and make it class
        for( i = 0 □ numSamples){
            if(used[i]){ // data reach to that node
                //increment the label and find the maximum repeated label
            }
        }
        node□value = -1 * maximum; // It's a leaf node
        return; // We finished
    }

    // Check if the labels are same, node is a leaf node.
    // if(labels are equal to each other){
    //      node->value = -1 * label;
    //      return;
    // }

    // Now we need to find the maximum Information Gain

    for( i = 0 -> numFeatures){
        if( !featueUse[i] ) { // feature is available
            double x = calculateInformationGain(....);
            if( x > max ){
                max = x;
                featureId = i;
            }
        }
    }
    // maximum Information Gain occurs when we choose featureId

    featureUse[featureId] = true;
    node->value = featureId; // we choose this feature for that node

    // Create new bool array for right child and left child
    for( int i = 0; i < numSamples; i++){
        if( used[i] ){ // check data can be used for further steps or not
```

```
                        If (data[i][featureId]) {
                                rightChild[i] = true;
                                leftChild[i] = false;
                        }
                        else{
                                rightChild[i] = false;
                                leftChild[i] = true;
                        }
                }
        }
        // Create node’s child and repeat for right and left child
        node->right = new DecisionTreeNode;
        node->left = new DecisionTreeNode;
        trainTree(data, labels, numSamples, numFeatures, current - 1, rightChild,
                featureUse, node->right);

        trainTree(data, labels, numSamples, numFeatures, current - 1, leftChild,
                featureUse, node->left);
}
.
.
.
void train(const string fileName, ....){
        // Open the file
        // Read line by line
        // Last number represents the label
}
```

For predict function:

```
int DecisionTree::predict(const bool* data){
        DecisionTreeNode* cur = root;

        while(cur->value >= 0){       // Node is not class, it holds a feature
                if(data[cur->value])  // Split to the right
                        cur = cur->right;
                else                          // Split to the left
                        cur = cur->left;
        }
        return -1 * cur->value; // Because class is represented with negative numbers
}
```

For test functions:

```
double DecisionTree::test(const bool** data,....){
        int trueSoFar = 0;

        for(int i = 0; i < numSamples; i++){
                // Call predict function for every sample
                // Then compare with the label
                // if they are equal, test is true else false
                int x = predict(data[i]);
                if(x == labels[i])
                        trueSoFar++;
        }
        //Return the accuracy percentage
        double percentage = (double)trueSoFar / (double)numSamples;
        return percentage * 100;
}
.
.
.
double DecisionTree::test(const string fileName, const int numSamples){

        // We don't have the feature number but we know the last number in the digit is
        // used for labels
        // x = 0;
        // Create two arrays; bool** data and int* labels
        // Fill data
        // for(int i = 0; i < line.length() - 1; i+=2){
        //         if(line[i] == '1')
        //             data[x][i / 2] = true;
        //         else if(line[i] == '0')
        //             data[x][i / 2] = false;
        // }
        // and fill labels
        // labels[x] = line[line.length() - 2] - '0';
        // increment x;
        // Call test(const bool** data, labels, numSamples);
        // And return the answer
}
```

Finally, for print:

```
void print(){
        // PREORDER PRINT
        print(root, 0);
}

void print(DecisionTreeNode* node, int tabs){
        // Recursion will be stopped when it reaches to class leaf
        if(node->nodeDecide()){
                // Print number of tabs that is given
                // print “class=node->value * -1”
        }
        else{
                // Parent - Left - Right
                // Print number of tabs that is given
                // print “node->value”
                print(node->left, tabs + 1);
                print(node->right, tabs + 1);
        }
}
```

In calculateInformationGain and calculateEntropy:

We basically follow the formulas given in the assignment. In Information Gain, we create two arrays for right and left split, and then according to featureId and usedSamples, we find which data goes into which node.

We know that the Decision Tree is a binary tree and also we know that every parent has to have two children. Due to the fact that if the node is not split, which means either all the labels that reached to that node are equal or there is no feature left to check. Either way node is not split into two. Similarly, if the node split, which means data is either true or false. Therefore, we need to create two child nodes for each split. Furthermore, if we use all the features for splitting our height will be numFeatures which is the worst case because in order to find the class, we check every numFeatures . In optimal case our height will be log(numFeatures), because our best height could be one or two, it depends on the variety of the data and different features.

In **trainTree** we iterate numFeatures times calculateInformationGain and in this function we have a for, iterating numSamples.

Therefore, in optimal case our time complexity is:
O(numSamples * numFeatures) * **RECURSION**

As it mentioned before, our tree's optimal depth will be log(numFeatures) and therefore, our recursion's optimal complexity will be log(numFeatures). And similarly, worst case will be numFeatures complexity.

Therefore, trainTree's optimal time complexity will be
O(log(numFeatures) * numSamples * numFeatures)
Thus, worst case will be
O(numFeatures * numFeatures * numSamples)

For other train functions, we know that they call trainTree function.
train(const strint …) calls train(const bool data, …), and train(const bool** data,..) calls the trainTree.**
Even though each of the train functions have different iterations and different complexities regardless of trainTree, their maximum effect on the complexity can be:
O(numSamples * numFeatures) which is smaller than trainTree.

Let's find the train functions complexity:
O(numSamples * numFeatures) + O(trainTree) =
O(numSamples * numFeatures(log(numFeatures) + 1))
which is equal to
= O(numSamples * numFeatures * log(numFeatures))

For test function, our time complexity will depend on the predict function.
Time complexity:
O(numSamples) * **PREDICT**

We know that predict searches the tree and the worst search is the depth of the tree. For optimal tree our height will be log(numFeatures) and worst case will be numFeatures.

Therefore, time complexity of the test(const bool** data,...) is
O(numSamples * log(numFeatures))

For print function, our height will be important. Our complexity will be
O(log(numFeatures) * log(numFeatures)), because function always iterates currentHeight for tabs, before printing the feature. Hence, currentHeight will be recursively

iterated height times, print function’s time complexity will be O(logn * logn) which is O(log(numFeatures) * log(numFeatures)).