Defne Betül Çiftci

21802635

Section 2

Assignment 2

double calculateEntropy(const int* classCounts, const int numClasses);

I used a direct approach in here, defining three doubles which are entropy, sum, and eachCalculation. sum held the value for the sum of classes inside the classCounts array, eachCalculation held the value to add to the variable entropy throughout a loop going through elements of classCounts. I also imported <bits/stdc++.h> here to be able to use the global log(int base) function. I assume its time complexity to be O(n) because I used two for loops starting from 0 to numClasses, incremented by one.

double calculateInformationGain(const bool** data, const int* labels,const int numSamples, const int numFeatures, const bool* usedSamples,const int featureId);

I used a helper function for this which has a prototype **void countClasses(const int numSamples, int*& classCountsAll, int& numClassesAll, const int*& labels , int*& countClass0, int& count0, int*& countClass1, int& count1, bool* columnWithFeature, const bool* usedSamples)**. I first define and initialize three integer arrays, one for the parent and two for the children. Then, since I used reference parameters for these arrays and the counts (numClassesAll is the count to be used for numClasses and count0 and count1 were used to get their probability, which are count0/numSamples and count1/numSamples. classCounts parameter of calculateEntropy was similarly replaced by classCountsAll, countClass0 and countClass1. Inside the helper method, I first define a dynamically allocated array called *newLabels* in which I put the values for each new label name we have come through, and I checked the each of the classCounts variables and arrays using the newLabels array. Another helper function I used here was **void incrementSize(int*& arr, int size)** because I first initialized arrays of size 10 (which I defined as a const in the header file) and if I needed to increment it, I would need to use this function.

I assume the time complexity to be O(n^2) because I used nested for-loops and I did not use any recursion.

int DecisionTree::predict(const bool* data);

I initialized a DecisionTreeNode* called current, and used a while loop which goes through each element of the current until its isLeaf() function returns true. I assume the time complexity to be O(n).

double DecisionTree::test(const bool** data, const int* labels, const int numSamples);

The function uses a for loop to look at each prediction (calls the predict function) throughout the data arrays in the parameter, and counts the times in which the prediction is true. It then returns (counts of each true guess) / (number of samples). I assume the time complexity to be O(n) because of the for-loop.

void DecisionTree::train(const bool** data, const int* labels, const int numSamples, const int numFeatures);

train function creates two Boolean arrays, usedSamples of size numSamples in which every element is initialized as true and parentNodes of size numFeatures in which every element is assigned to be false. Then the function calls a recursive function, through this line:

root = decideOnRoot((bool**) data, labels, numSamples, numFeatures, usedSamples, parentNodes);

which I will talk about more in depth now.

DecisionTreeNode* DecisionTree::decideOnRoot(bool** data, const int* labels, const int numSamples, const int numFeatures, const bool* usedSamples, bool* parentNodes);

The function takes the following parameters: data, labels, numFeatures and numSamples as passed from the original train function. These values do not change inbetween recursive calls. Then the usedSamples and parentNodes Boolean arrays, through which we determine which part of the data array we will analyse.

Let's assume we are on the first call of the function. usedSamples is all true and parentNodes is all false. We get the maximum information gain of the featureId's while checking if that index we are getting the information gain from is true or false, and since it is false for all of it, we get a featureId for the first node. Then we go on to the checking part for the leaf node.

```
bool isPure = true;
 int label = -1;
int i;
  for( i = 0; i < numSamples; i++) {
    if(usedSamples[i]) {
      label = labels[i];
      break;
    }
  }
```

In the above loop, we will get the first element in which usedSamples is true, in this case the first one, and we will give a temporary integer named *label* the class value of that index and we will continue accordingly:

```
  for( ; i < numSamples; i++) {
    if( usedSamples[i] && labels[i] != label) {
      isPure = false;
      break;
    }
  }
```

Above, we check if there are any class values different than the temporary one. If it is a leaf node, isPure should never be false for the entirety of numSamples.

```
if( isPure && label != -1) {
    node->setAsLeaf( label); //this will also assign the labelData
    return node;
}
```

Here, we check for two things: if isPure is true (as we will assign the labelData accordingly) and if the temporary class value is not -1. This is done purely on the assumption that negative integers will not be used as class labels, but if they are used, this part of the code can be adjusted for label to hold any value that is not held as a class label. “Return node;” line is there to end the recursive loop.

Then I assign the featureId (since if it was a leaf node it should have been going back), I assign usedSamples0 and usedSamples1 arrays for 1 and 0 values and then through these lines I call the recursive function again:

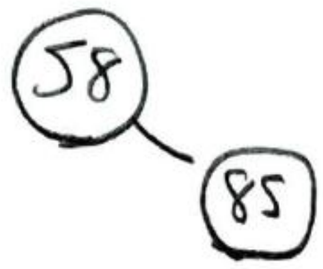
```
node->leftChild = decideOnRoot( data, (const int*) labels, numSamples, numFeatures, usedSamples0, parentNodes);//recursive for training
node->rightChild = decideOnRoot( data, (const int*) labels, numSamples, numFeatures, usedSamples1, parentNodes );
```

The function returns the node at the end.

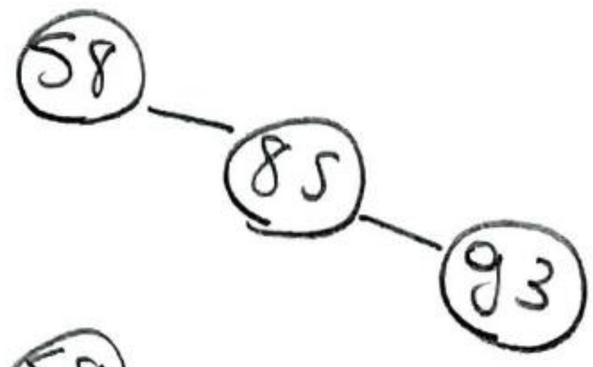
9)

insert 58:

insert 85:

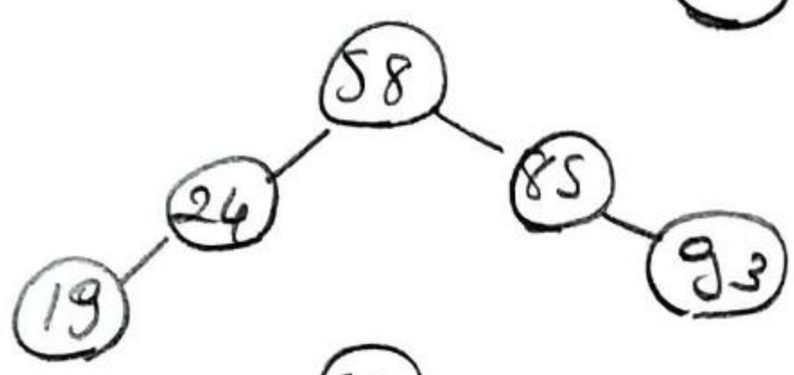


insert 93:

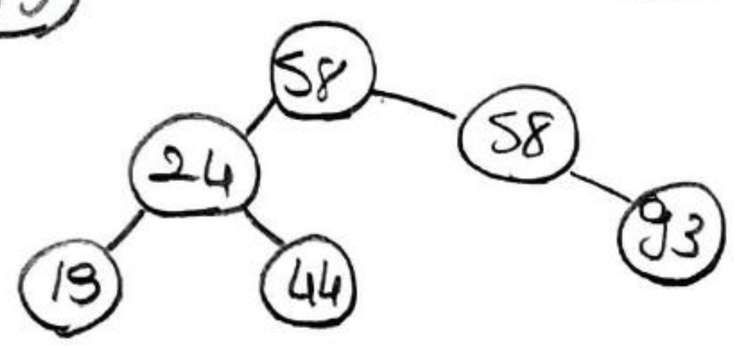


insert 24:

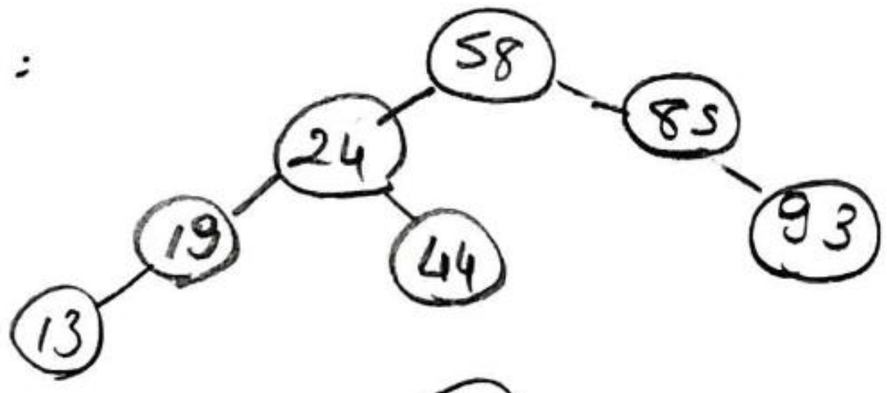
insert 19:



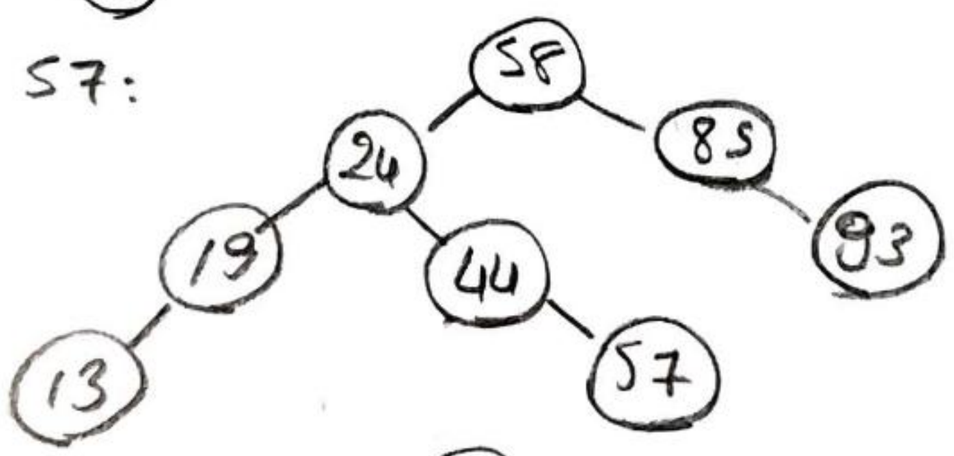
insert 44:



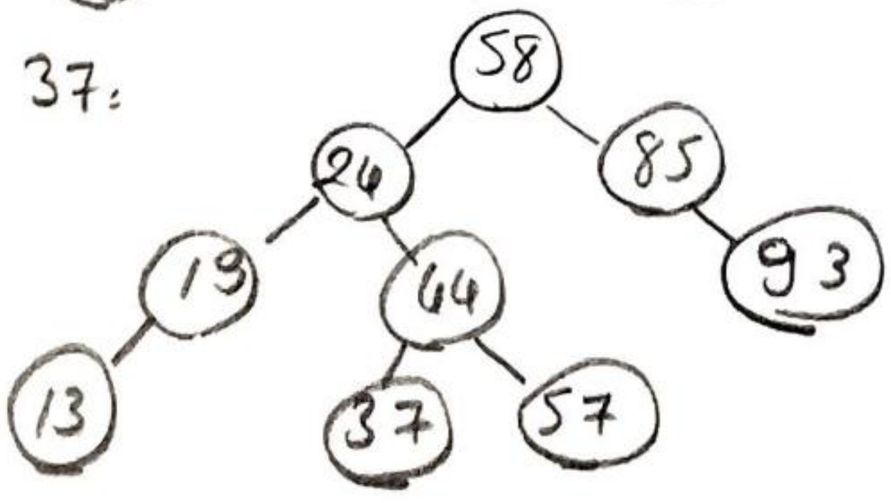
insert 13:



insert 57:



insert 37:



Inorder Traversal:

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

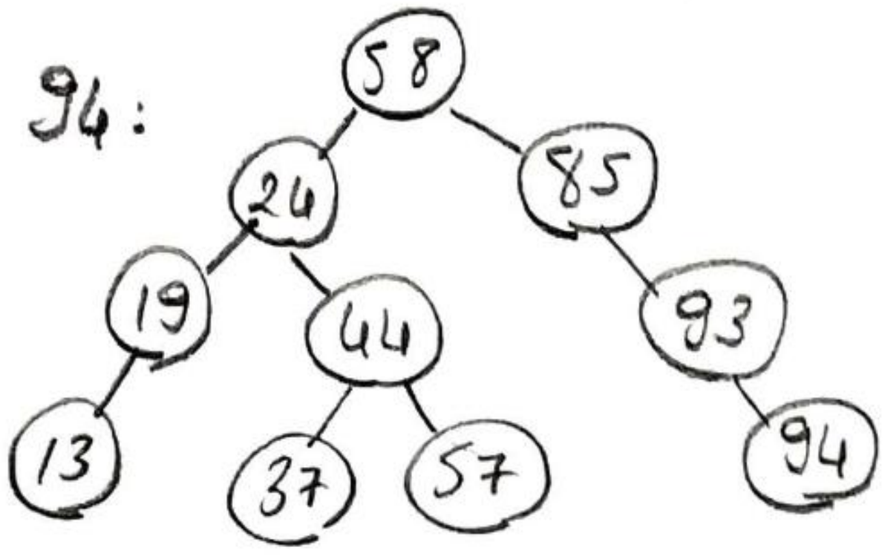
Preorder traversal:

(6) 58, 24, 19, 13, 44, 37, 57, 85, 93, 94

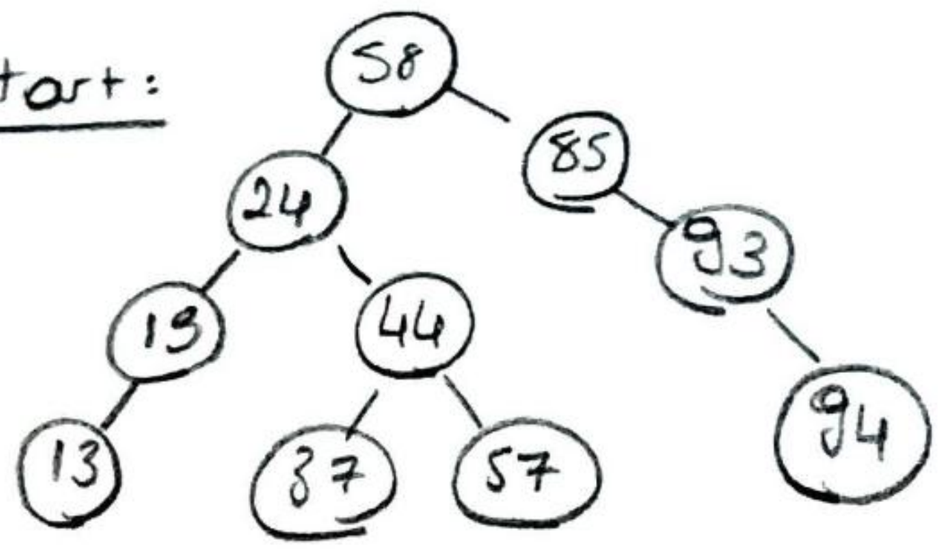
Postorder Traversal:

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

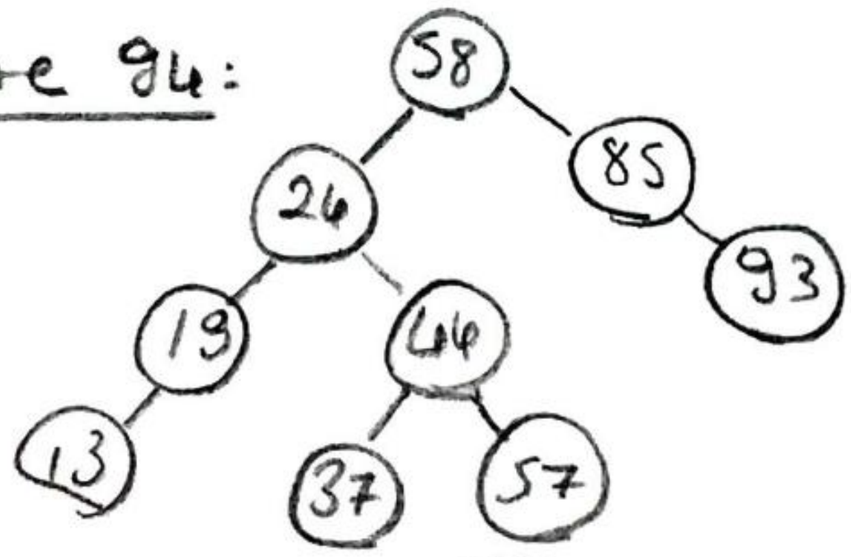
insert 94:



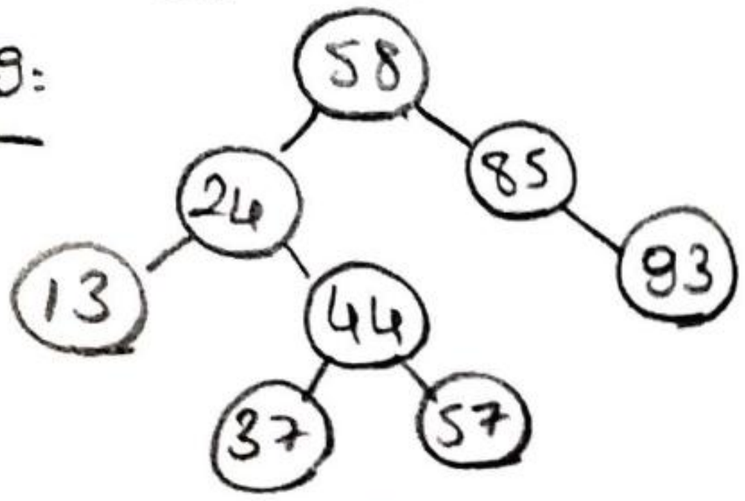
Start:



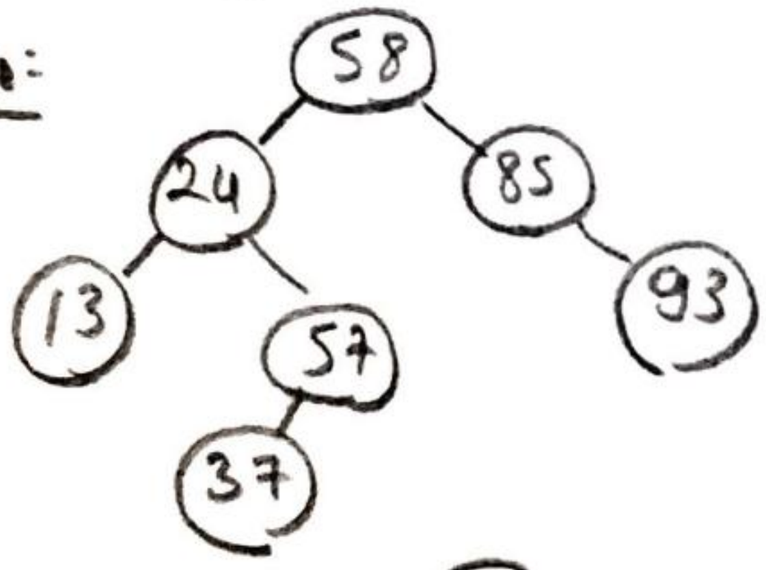
Delete 94:



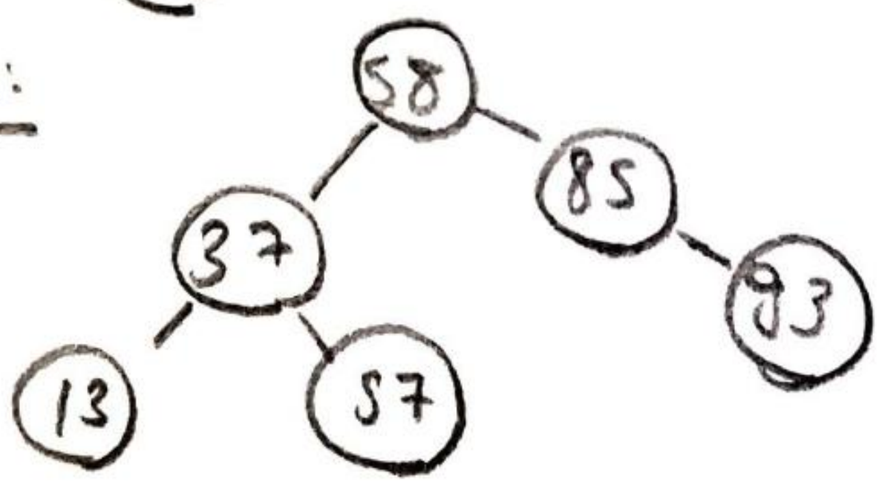
Delete 19:



Delete 44:



Delete 24:



Delete 58: