Course: ECE-35900

Name: Elijah Basile

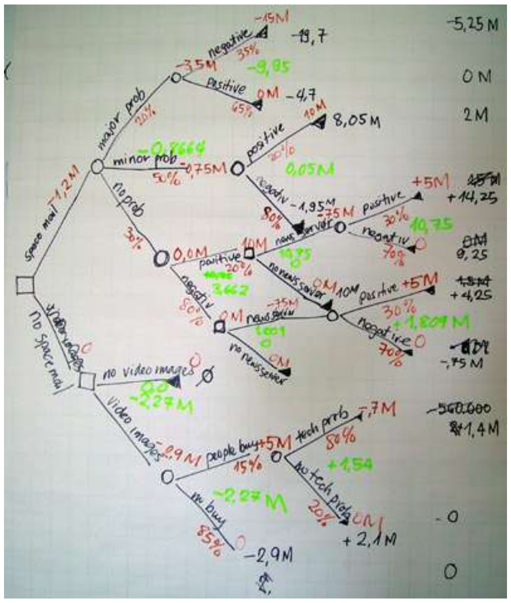
Honors Project

Date: Spring 2019 Semester

Summary:

For my honors project this semester, my goal was to construct a functioning Decision Tree application to make meaningful predictions on datasets without having to store the entirety of them in memory.

The first step I took was learning about Decision Trees. This exploit took me across the web, landing me in the knowledge oasis of the web (most commonly referred to as Wikipedia). There, I learned about traditional decision trees and their application in basic logic. Below is a picture of an example Decision tree.



Observing this decision tree example, I took to applying that same model to a dataset given to me by a graduate student under Dr. Zina Ben Miled: Álvaro Soares de Boa Esperança. He provided me a collection of information drawn from individual's tweets. All have one thing in common: they mention Diabetes (or diabetic) at one point.

Now differentiating whether a tweet signaled the individual actually had diabetes became my mission as I began constructing and applying what I later would learn was not a decision tree, but an ontology tree.

To begin, let me explain what an ontology tree is and compare it to what a supposed decision tree program would look like.

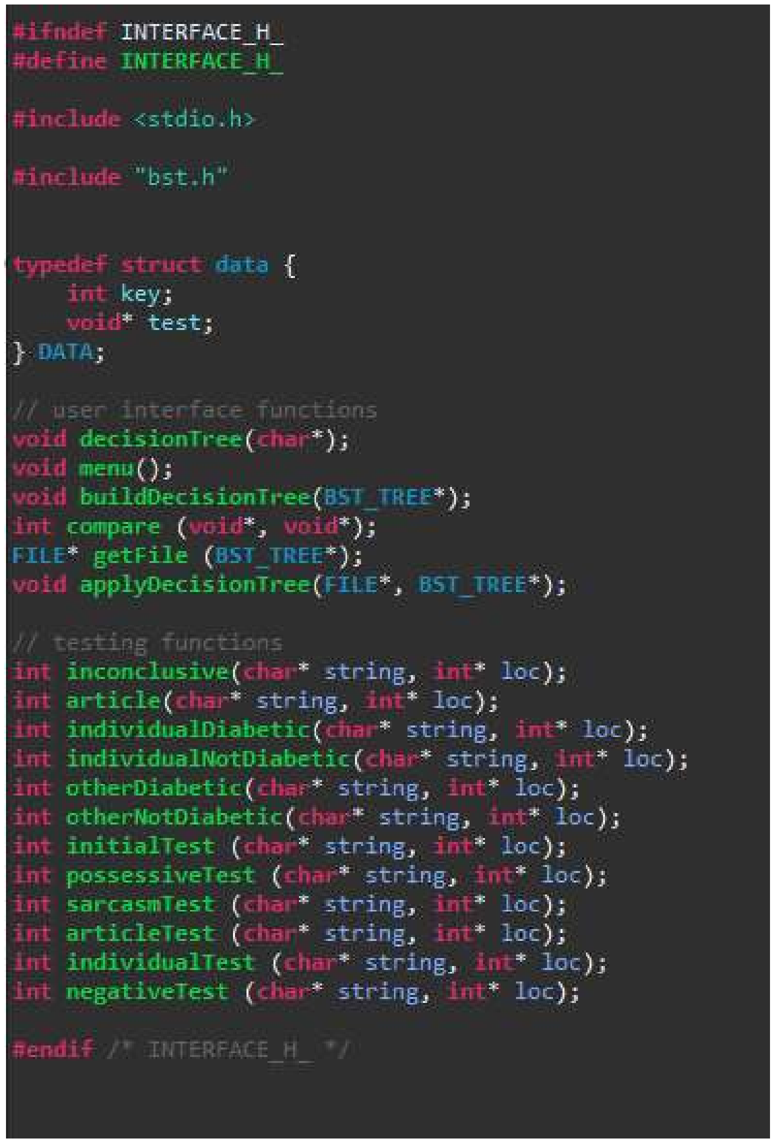
An ontology tree program takes a dataset (i.e. the past weather conditions over the past ten years), and draws conclusions based on derived results done by those familiar with the dataset (i.e. it will rain today due to x, y, z conditions that mirror past scenarios that scientists have tested). A decision tree program takes an entirely different approach, scraping the dataset and each result for the raw data and making conclusions based on the basic components leading to each particular result. The individual that is constructing and applying a decision tree to a dataset does not need to know what the set contains, only how to accurately parse the dataset so the program can "learn" from it and mirror those conclusions based on the contents of the set.

In conclusion, while my program did not end up acting like or even resembling like a decision tree *machine learning* program, I did gain a valuable experience in the high expense, tediousness and possible inaccuracy that an ontology program may have. The advantages of an effective decision tree program soon become clear.

Methodology:

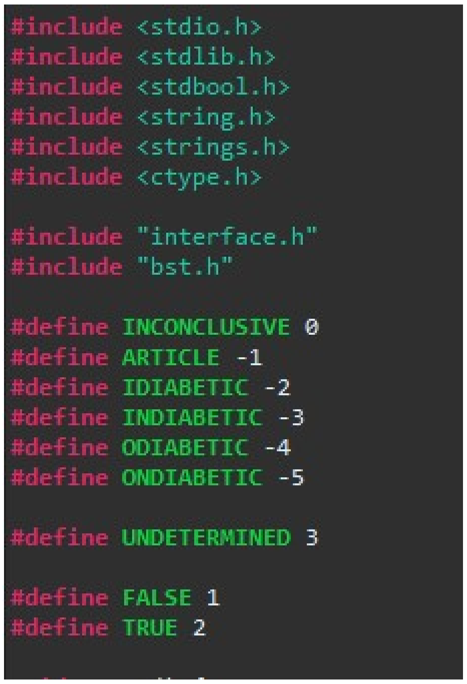
Below, you will find my approach in constructing my ontology tree. Despite the tedious nature of writing such a program, I gained further insight in the power of void* and functions as well as the standard string manipulation libraries.

I will start by saying I used a basic binary search tree (BST) data structure to base my tree. You may refer to the generic code I used in bst.c and bst.h if you are curious on my implementation; however, I will spend most of the speaking on my approach to implementing the ontology tree.



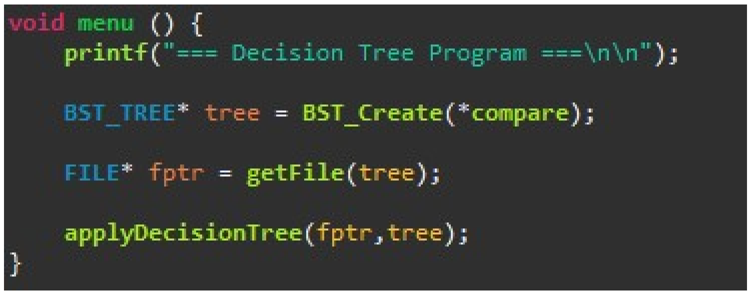
```c
#ifndef INTERFACE_H_
#define INTERFACE_H_

#include <stdio.h>

#include "bst.h"


typedef struct data {
    int key;
    void* test;
} DATA;

// user interface functions
void decisionTree(char*);
void menu();
void buildDecisionTree(BST_TREE*);
int compare (void*, void*);
FILE* getFile (BST_TREE*);
void applyDecisionTree(FILE*, BST_TREE*);

// testing functions
int inconclusive(char* string, int* loc);
int article(char* string, int* loc);
int individualDiabetic(char* string, int* loc);
int individualNotDiabetic(char* string, int* loc);
int otherDiabetic(char* string, int* loc);
int otherNotDiabetic(char* string, int* loc);
int initialTest (char* string, int* loc);
int possessiveTest (char* string, int* loc);
int sarcasmTest (char* string, int* loc);
int articleTest (char* string, int* loc);
int individualTest (char* string, int* loc);
int negativeTest (char* string, int* loc);

#endif /* INTERFACE_H_ */
```

My interface.h file contained the function prototypes of all of my functions, but more importantly it contains the data struct I used in my BST. The key was an integer determining its place in the BST, while the void* contained the address to a testing function for the string passed into it.



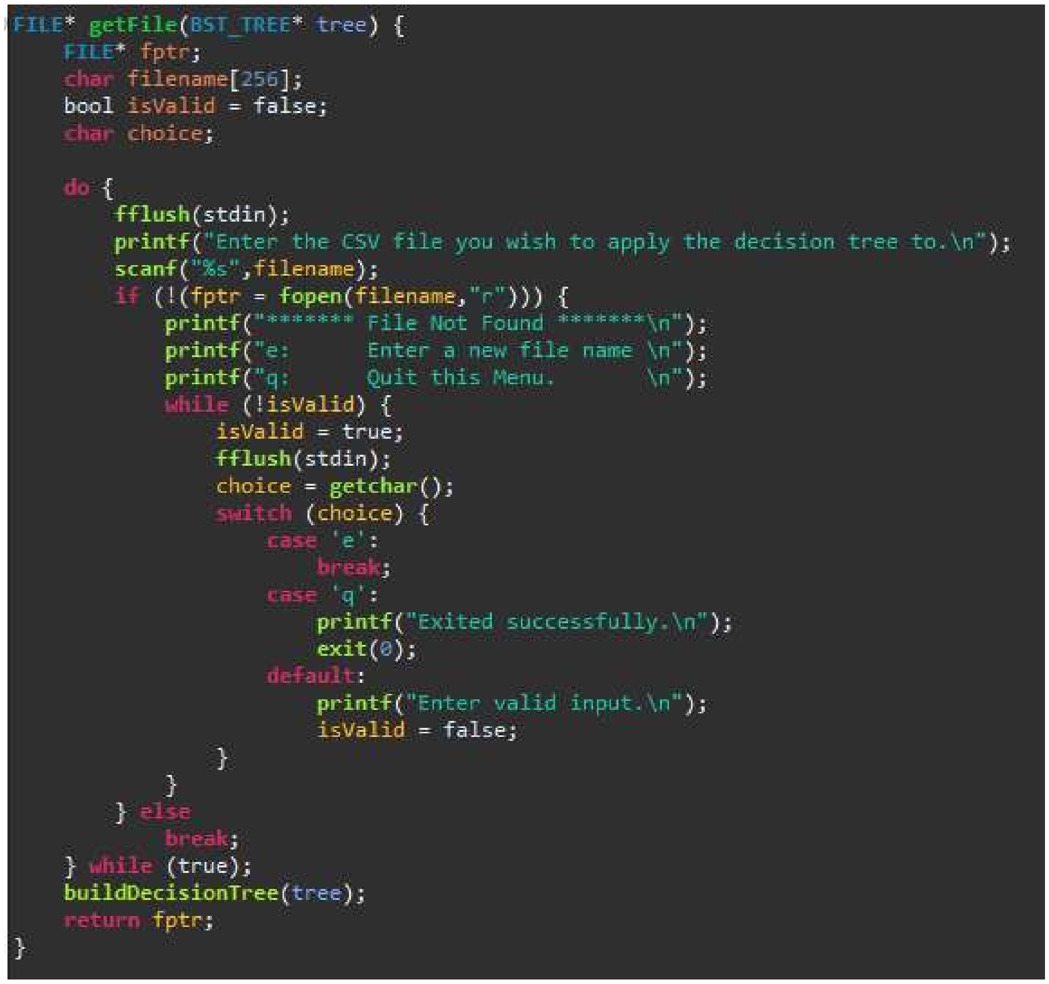
```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <strings.h>
#include <ctype.h>

#include "interface.h"
#include "bst.h"

#define INCONCLUSIVE 0
#define ARTICLE -1
#define IDIABETIC -2
#define INDIABETIC -3
#define ODIABETIC -4
#define ONDIABETIC -5

#define UNDETERMINED 3

#define FALSE 1
#define TRUE 2
```

At the beginning of my c interface file, I included the following macros and libraries for easier readability of my code.



```c
void menu () {
    printf("=== Decision Tree Program ===\n\n");

    BST_TREE* tree = BST_Create(*compare);

    FILE* fptr = getFile(tree);

    applyDecisionTree(fptr,tree);
}
```

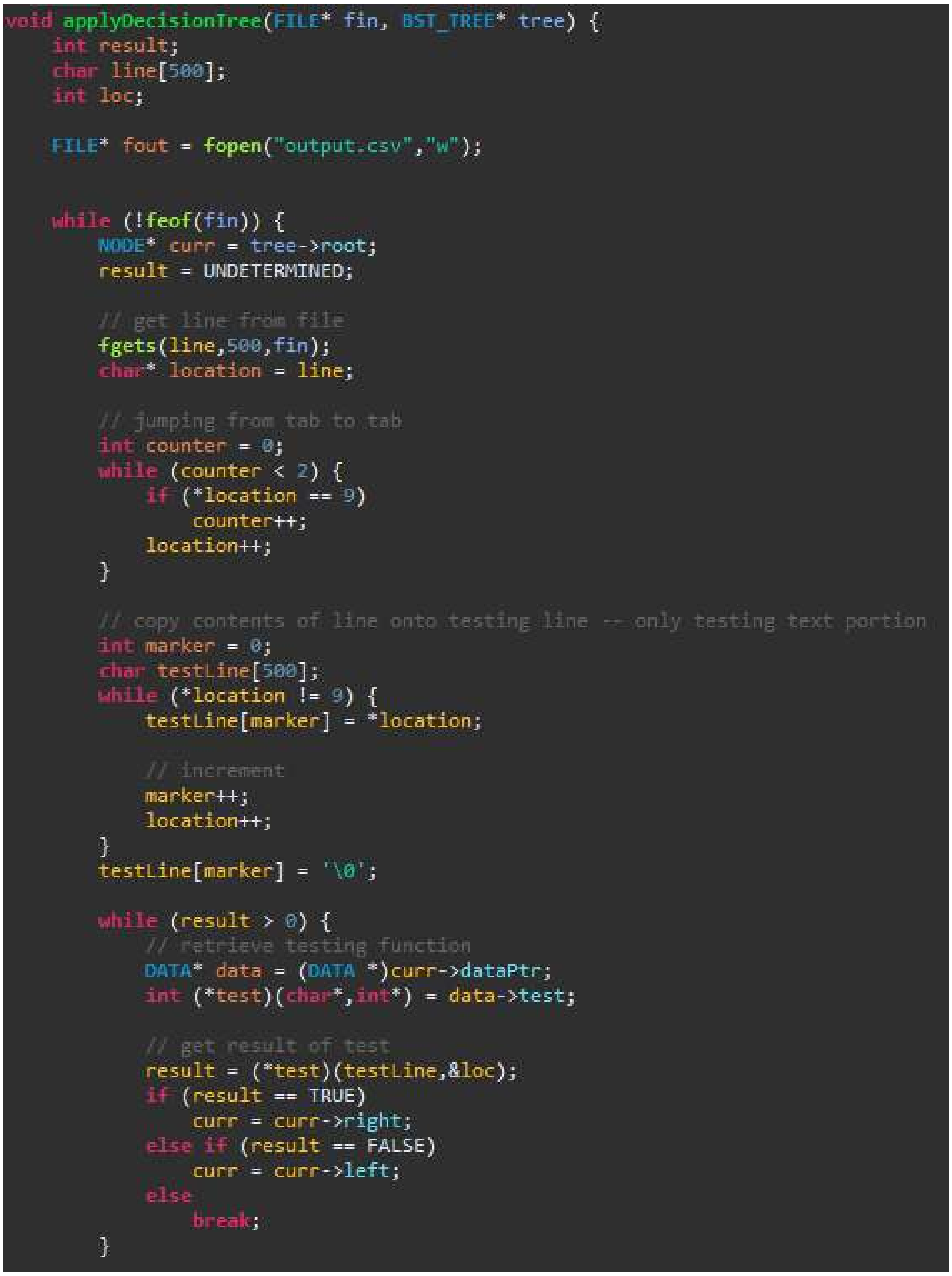
My menu function acted as the basic skeletal call for the execution of my code.



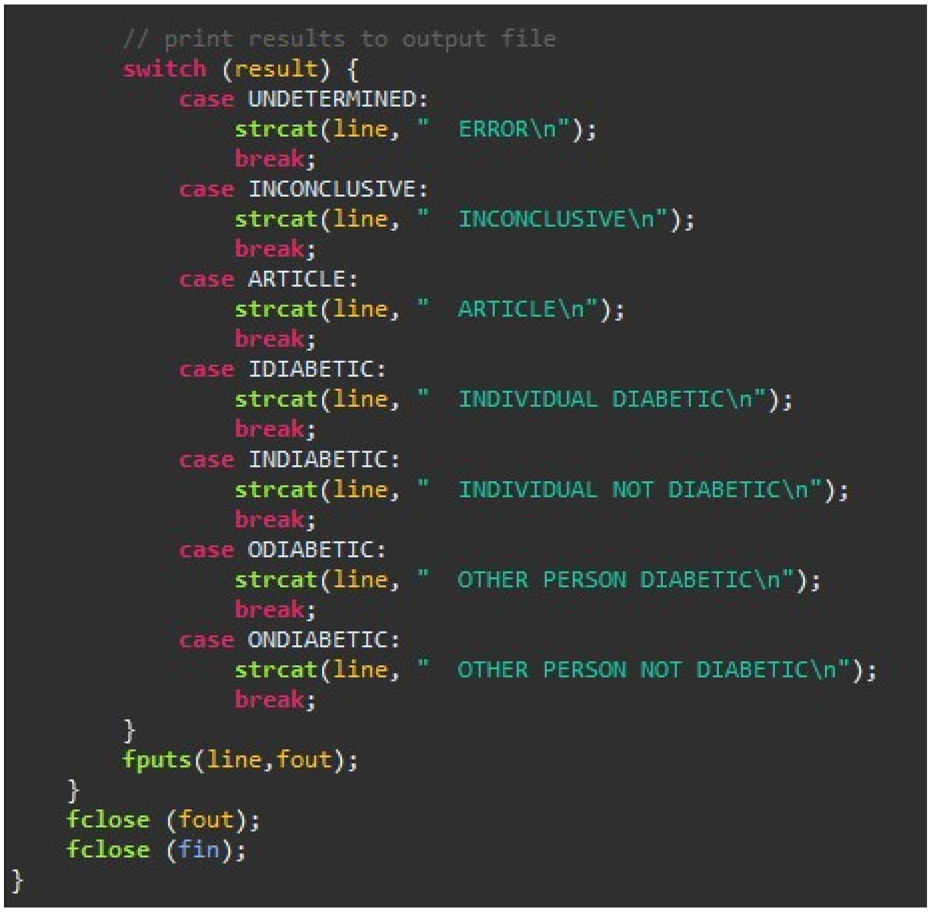
```
FILE* getFile(BST_TREE* tree) {
    FILE* fptr;
    char filename[256];
    bool isValid = false;
    char choice;

    do {
        fflush(stdin);
        printf("Enter the CSV file you wish to apply the decision tree to.\n");
        scanf("%s",filename);
        if (!(fptr = fopen(filename,"r"))) {
            printf("******* File Not Found *******\n");
            printf("e:      Enter a new file name \n");
            printf("q:      Quit this Menu.       \n");
            while (!isValid) {
                isValid = true;
                fflush(stdin);
                choice = getchar();
                switch (choice) {
                    case 'e':
                        break;
                    case 'q':
                        printf("Exited successfully.\n");
                        exit(0);
                    default:
                        printf("Enter valid input.\n");
                        isValid = false;
                }
            }
        } else
            break;
    } while (true);
    buildDecisionTree(tree);
    return fptr;
}
```

The above getFile function merely asks the user to enter the file name in question and tests if it exists. The decision tree is constructed following that and a file pointer to the beginning of the open file is returned.

The apply decision tree was by far the longest, and yet most integral part of the program. The function began by opening the output file, and then for each iteration grabbed one line of the csv file. A new string of text would be formed from that string to only include the text from the tweet. The “testing string” would then be passed into the initial testing function of the decision tree and would concurrently move from node to node down the tree until a return value of something other than TRUE or FALSE was found.

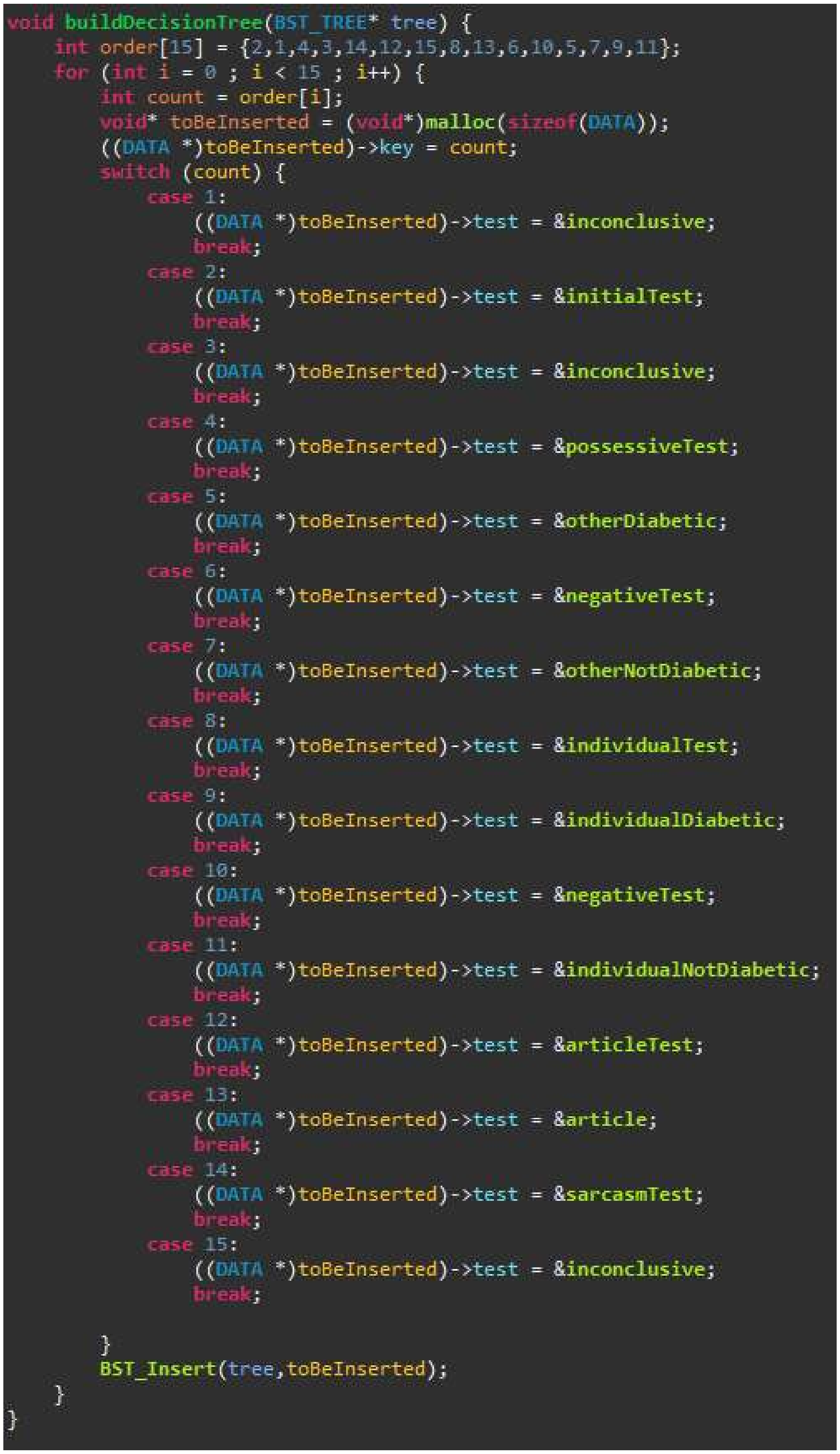


```c
void applyDecisionTree(FILE* fin, BST_TREE* tree) {
    int result;
    char line[500];
    int loc;

    FILE* fout = fopen("output.csv","w");


    while (!feof(fin)) {
        NODE* curr = tree->root;
        result = UNDETERMINED;

        // get line from file
        fgets(line,500,fin);
        char* location = line;

        // jumping from tab to tab
        int counter = 0;
        while (counter < 2) {
            if (*location == 9)
                counter++;
            location++;
        }

        // copy contents of line onto testing line -- only testing text portion
        int marker = 0;
        char testLine[500];
        while (*location != 9) {
            testLine[marker] = *location;

            // increment
            marker++;
            location++;
        }
        testLine[marker] = '\0';

        while (result > 0) {
            // retrieve testing function
            DATA* data = (DATA *)curr->dataPtr;
            int (*test)(char*,int*) = data->test;

            // get result of test
            result = (*test)(testLine,&loc);
            if (result == TRUE)
                curr = curr->right;
            else if (result == FALSE)
                curr = curr->left;
            else
                break;
        }
```



```
        // print results to output file
        switch (result) {
            case UNDETERMINED:
                strcat(line, "  ERROR\n");
                break;
            case INCONCLUSIVE:
                strcat(line, "  INCONCLUSIVE\n");
                break;
            case ARTICLE:
                strcat(line, "  ARTICLE\n");
                break;
            case IDIABETIC:
                strcat(line, "  INDIVIDUAL DIABETIC\n");
                break;
            case INDIABETIC:
                strcat(line, "  INDIVIDUAL NOT DIABETIC\n");
                break;
            case ODIABETIC:
                strcat(line, "  OTHER PERSON DIABETIC\n");
                break;
            case ONDIABETIC:
                strcat(line, "  OTHER PERSON NOT DIABETIC\n");
                break;
        }
        fputs(line,fout);
    }
    fclose (fout);
    fclose (fin);
}
```

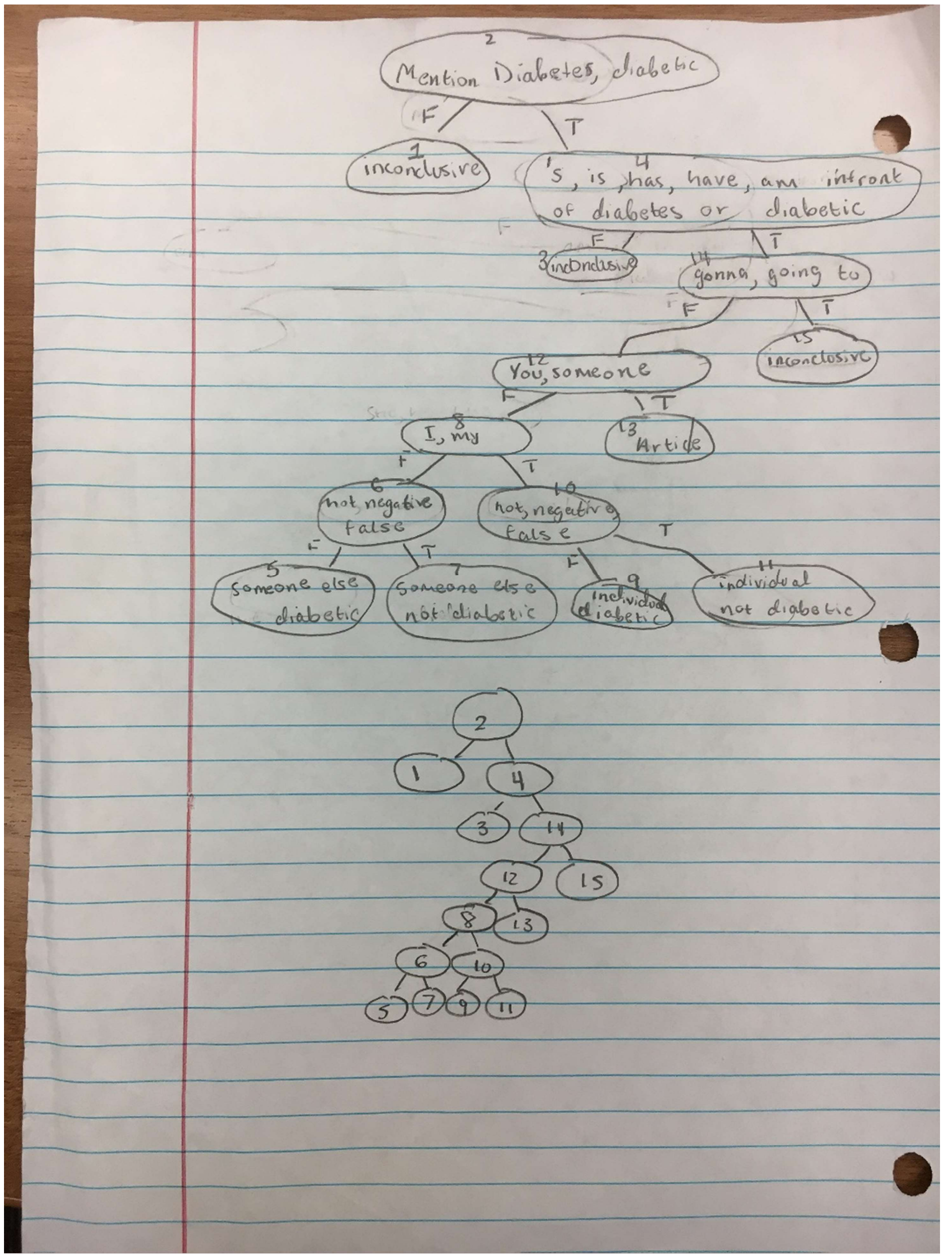
Based on that return value. The tree would append to the line in question and place the results right below the line in question.



```c
void buildDecisionTree(BST_TREE* tree) {
    int order[15] = {2,1,4,3,14,12,15,8,13,6,10,5,7,9,11};
    for (int i = 0 ; i < 15 ; i++) {
        int count = order[i];
        void* toBeInserted = (void*)malloc(sizeof(DATA));
        ((DATA *)toBeInserted)->key = count;
        switch (count) {
            case 1:
                ((DATA *)toBeInserted)->test = &inconclusive;
                break;
            case 2:
                ((DATA *)toBeInserted)->test = &initialTest;
                break;
            case 3:
                ((DATA *)toBeInserted)->test = &inconclusive;
                break;
            case 4:
                ((DATA *)toBeInserted)->test = &possessiveTest;
                break;
            case 5:
                ((DATA *)toBeInserted)->test = &otherDiabetic;
                break;
            case 6:
                ((DATA *)toBeInserted)->test = &negativeTest;
                break;
            case 7:
                ((DATA *)toBeInserted)->test = &otherNotDiabetic;
                break;
            case 8:
                ((DATA *)toBeInserted)->test = &individualTest;
                break;
            case 9:
                ((DATA *)toBeInserted)->test = &individualDiabetic;
                break;
            case 10:
                ((DATA *)toBeInserted)->test = &negativeTest;
                break;
            case 11:
                ((DATA *)toBeInserted)->test = &individualNotDiabetic;
                break;
            case 12:
                ((DATA *)toBeInserted)->test = &articleTest;
                break;
            case 13:
                ((DATA *)toBeInserted)->test = &article;
                break;
            case 14:
                ((DATA *)toBeInserted)->test = &sarcasmTest;
                break;
            case 15:
                ((DATA *)toBeInserted)->test = &inconclusive;
                break;

        }
        BST_Insert(tree,toBeInserted);
    }
}
```

This function constructs the decision tree and adds each testing function to each specific branch.

Also written were my testing functions. For clarity, I am just including my written-out decision tree. As you can see, I ran through several tests before finding definitive conclusions; otherwise, the program would return an nonconclusive signature to the line.


2
Mention Diabetes, diabetic
F
T
1
inconclusive
4
'S, is, has, have, am infront of diabetes or diabetic
F
T
3 inconclusive
14
gonna, going to
F
T
15
inconclusive
12
You, someone
F
T
8
I, my
13
Article
F
T
6
not negative false
10
not, negative false
F
T
F
T
5
Someone else diabetic
7
Someone else not diabetic
9
individual diabetic
11
individual not diabetic
2
1
4
3
14
12
15
8
13
6
10
5
7
9
11