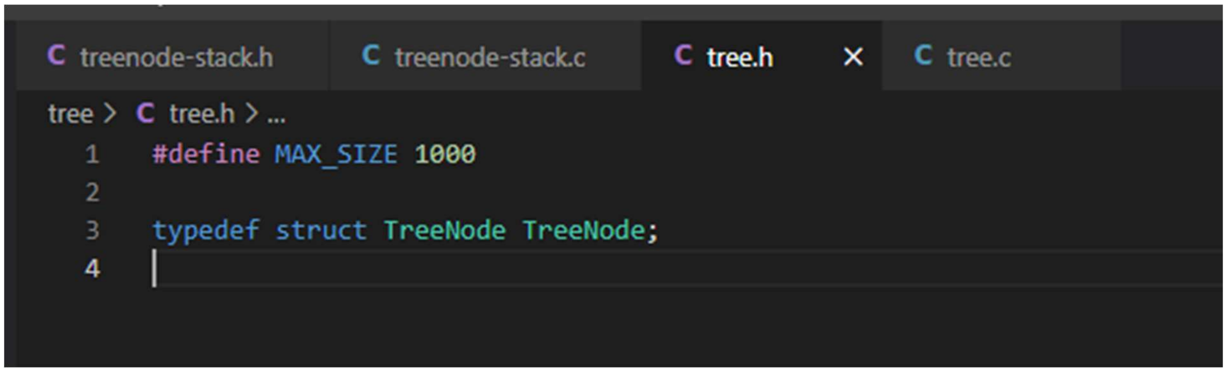
Tree vs Bubble Sort

Jacob Tucker

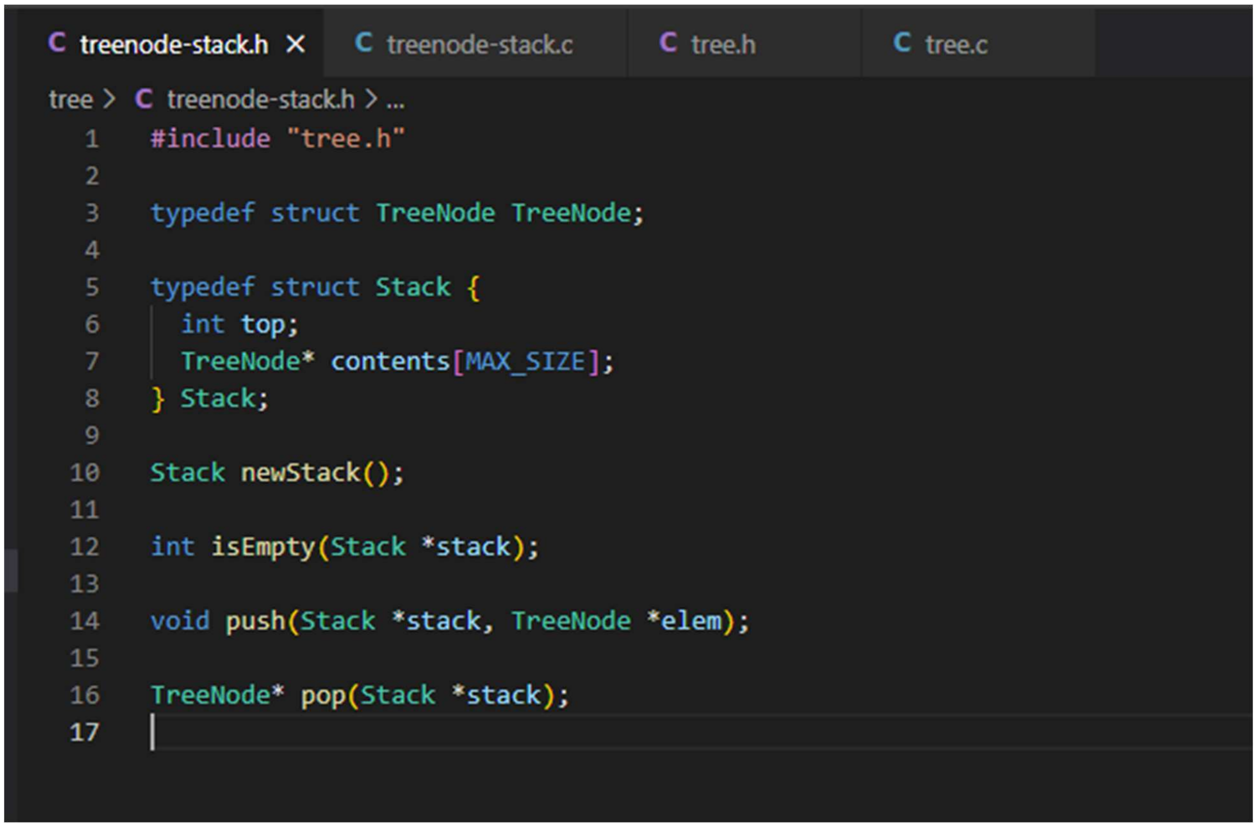
CMPS 390

4/8/23

This program defines a binary tree and provides functions to build a simple binary search tree from an array of integers, print the tree in-order, and bubble sort the same array of integers. It is split into 2 C files, tree.c and treenode-stack.c. The treenode-stack.c file is used to create a stack that has elements that are of type *TreeNode**. Since both C files use the *TreeNode* struct, they both have header files that use a forward declaration for *TreeNode* so that they can both use it without there being definition conflicts. The header files are as follows:

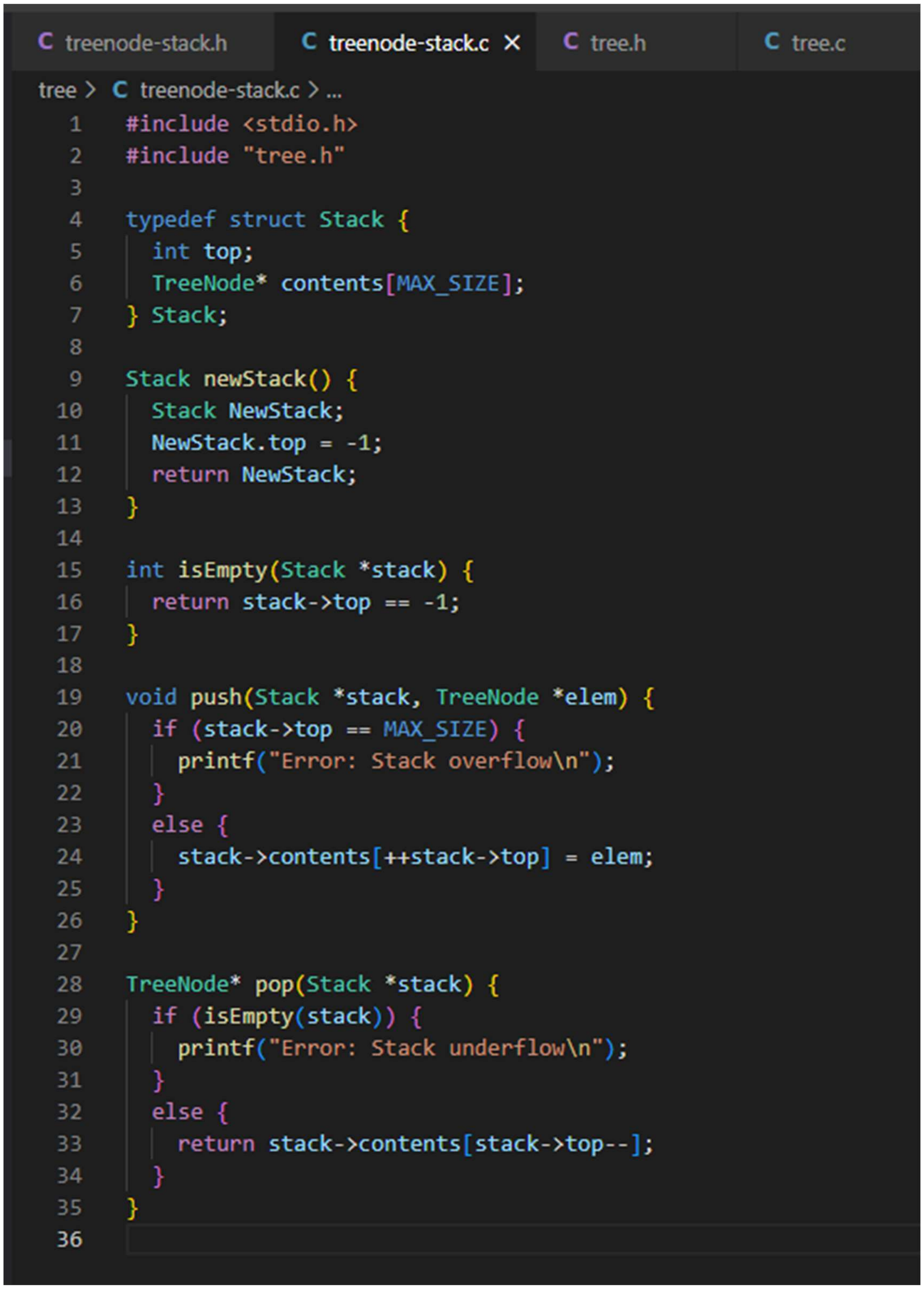


```c
#define MAX_SIZE 1000

typedef struct TreeNode TreeNode;

```



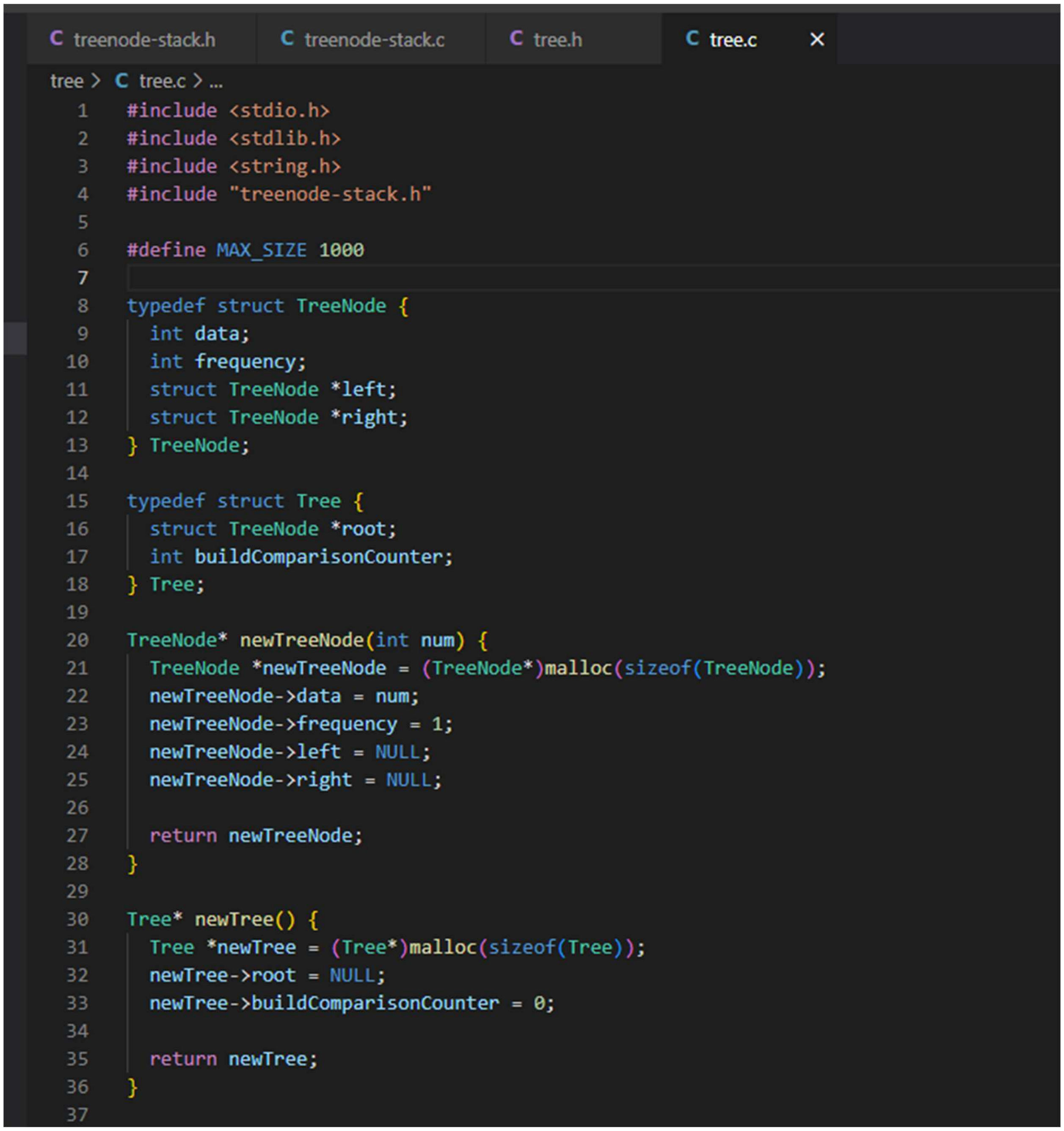
```c
#include "tree.h"

typedef struct TreeNode TreeNode;

typedef struct Stack {
    int top;
    TreeNode* contents[MAX_SIZE];
} Stack;

Stack newStack();

int isEmpty(Stack *stack);

void push(Stack *stack, TreeNode *elem);

TreeNode* pop(Stack *stack);

```

The implementation of the treenode-stack is just a very simple stack, similar to the one done in a previous assignment but with less features since they weren't needed.



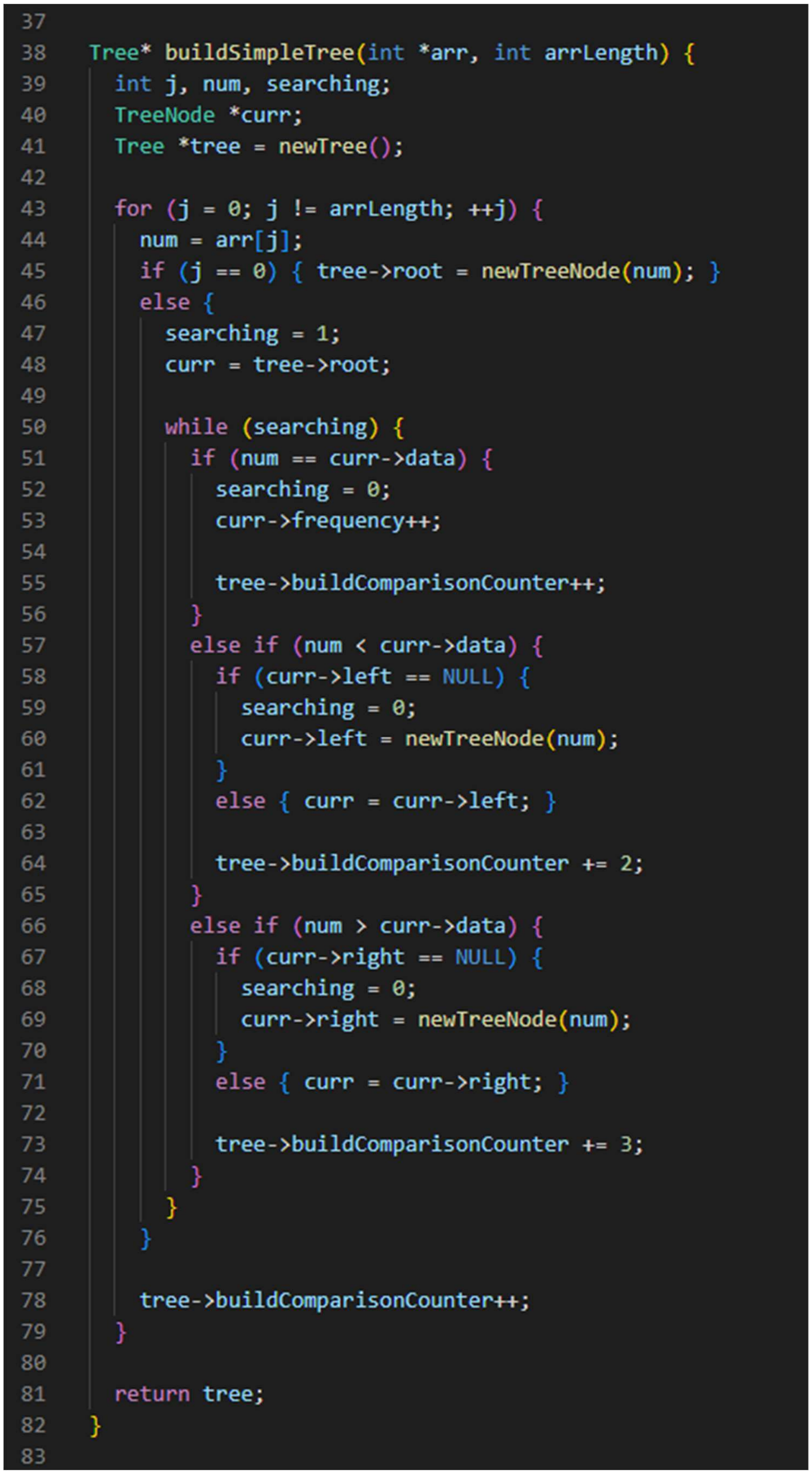
```c
#include <stdio.h>
#include "tree.h"

typedef struct Stack {
    int top;
    TreeNode* contents[MAX_SIZE];
} Stack;

Stack newStack() {
    Stack NewStack;
    NewStack.top = -1;
    return NewStack;
}

int isEmpty(Stack *stack) {
    return stack->top == -1;
}

void push(Stack *stack, TreeNode *elem) {
    if (stack->top == MAX_SIZE) {
        printf("Error: Stack overflow\n");
    }
    else {
        stack->contents[++stack->top] = elem;
    }
}

TreeNode* pop(Stack *stack) {
    if (isEmpty(stack)) {
        printf("Error: Stack underflow\n");
    }
    else {
        return stack->contents[stack->top--];
    }
}

```

For tree.c, the *TreeNode* and *Tree* structs are defined and initialized as follows:



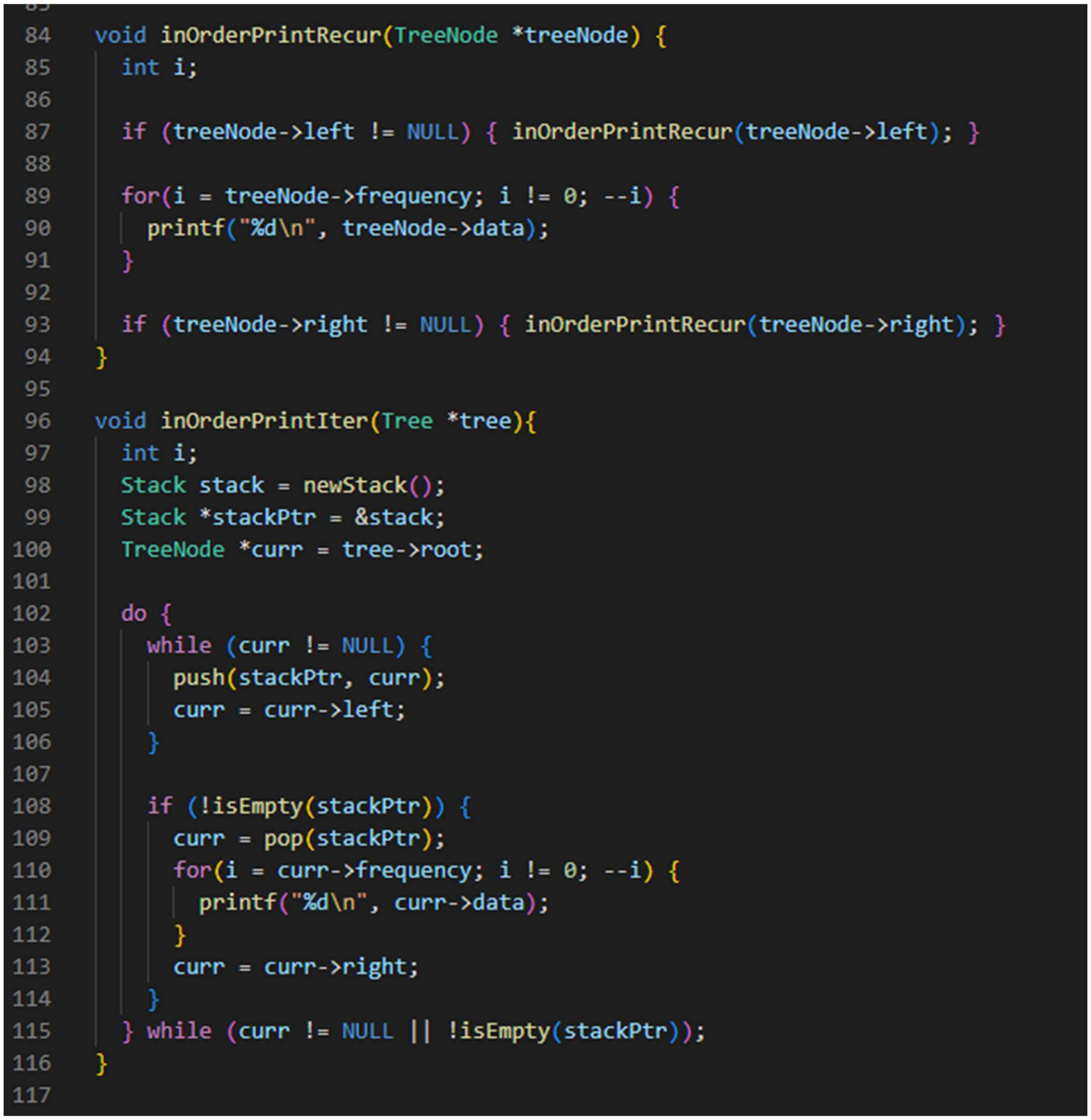
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "treenode-stack.h"

#define MAX_SIZE 1000

typedef struct TreeNode {
  int data;
  int frequency;
  struct TreeNode *left;
  struct TreeNode *right;
} TreeNode;

typedef struct Tree {
  struct TreeNode *root;
  int buildComparisonCounter;
} Tree;

TreeNode* newTreeNode(int num) {
  TreeNode *newTreeNode = (TreeNode*)malloc(sizeof(TreeNode));
  newTreeNode->data = num;
  newTreeNode->frequency = 1;
  newTreeNode->left = NULL;
  newTreeNode->right = NULL;

  return newTreeNode;
}

Tree* newTree() {
  Tree *newTree = (Tree*)malloc(sizeof(Tree));
  newTree->root = NULL;
  newTree->buildComparisonCounter = 0;

  return newTree;
}

```

The *TreeNode* has members: *data, frequency, left, and right* so that it can store data, the number of times that specific data has been inserted into a list rather than making more tree nodes to store duplicates, and pointers to the left and right. The *Tree* struct simply points to a “root” node to begin with and has a counter to track the number of data value comparisons made when building the tree so that its efficiency can be compared with other ordering algorithms. Both the *Tree* and *TreeNode* struct implementations involve manually allocating memory, and so will be deallocated later in the program.

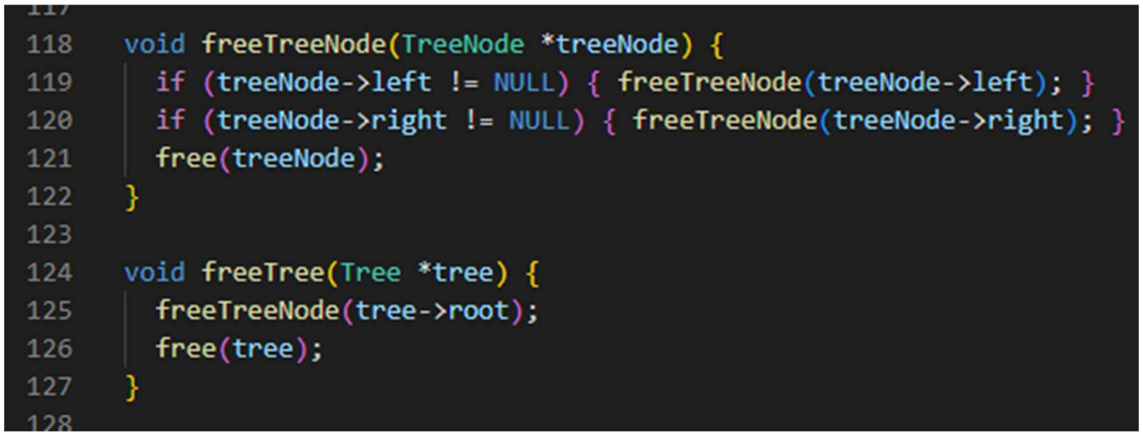


```c

Tree* buildSimpleTree(int *arr, int arrLength) {
  int j, num, searching;
  TreeNode *curr;
  Tree *tree = newTree();

  for (j = 0; j != arrLength; ++j) {
    num = arr[j];
    if (j == 0) { tree->root = newTreeNode(num); }
    else {
      searching = 1;
      curr = tree->root;

      while (searching) {
        if (num == curr->data) {
          searching = 0;
          curr->frequency++;

          tree->buildComparisonCounter++;
        }
        else if (num < curr->data) {
          if (curr->left == NULL) {
            searching = 0;
            curr->left = newTreeNode(num);
          }
          else { curr = curr->left; }

          tree->buildComparisonCounter += 2;
        }
        else if (num > curr->data) {
          if (curr->right == NULL) {
            searching = 0;
            curr->right = newTreeNode(num);
          }
          else { curr = curr->right; }

          tree->buildComparisonCounter += 3;
        }
      }
    }

    tree->buildComparisonCounter++;
  }

  return tree;
}

```

The *buildSimpleTree()* function takes in a pointer to an array and the arrays length and builds a binary tree out of the array's elements. The function starts by creating an empty tree, and then it iterates over each integer in the array. For each integer, it either creates a new node in the tree or increments the frequency of an existing node if the integer is already present in the tree. The function keeps track of the number of comparisons it makes during the tree-building process using the *buildComparisonCounter* counter.

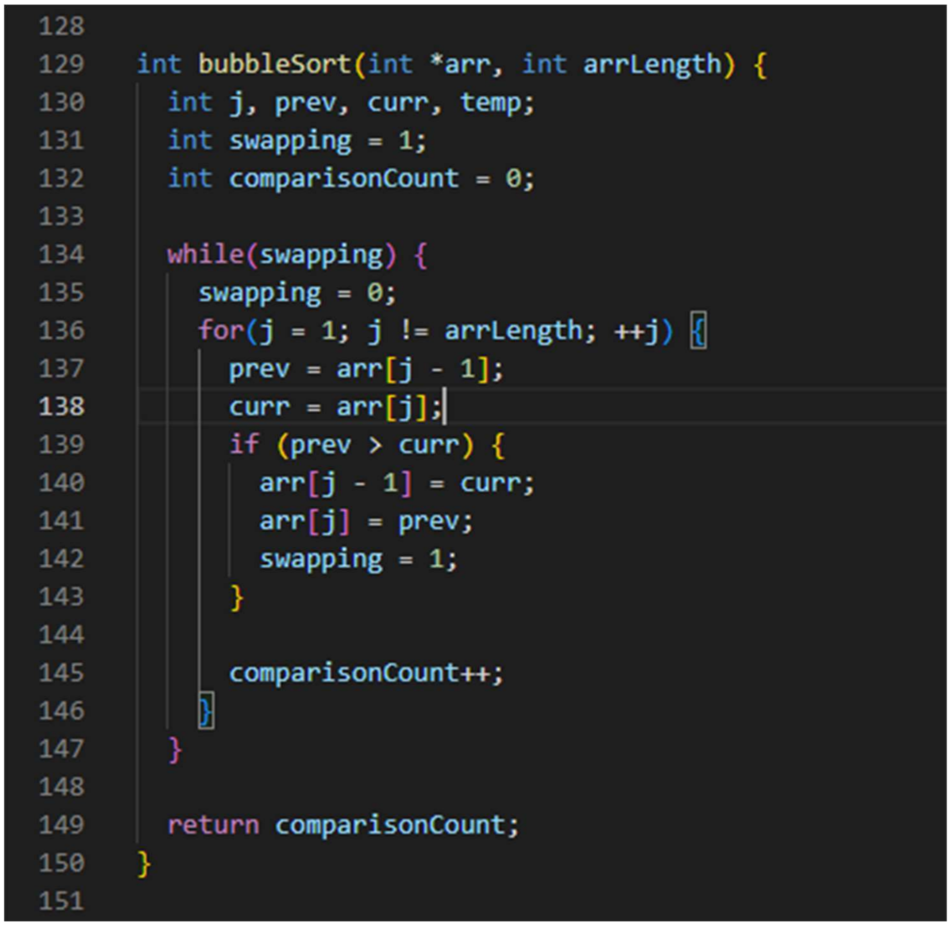

```c
void inOrderPrintRecur(TreeNode *treeNode) {
  int i;

  if (treeNode->left != NULL) { inOrderPrintRecur(treeNode->left); }

  for(i = treeNode->frequency; i != 0; --i) {
    printf("%d\n", treeNode->data);
  }

  if (treeNode->right != NULL) { inOrderPrintRecur(treeNode->right); }
}

void inOrderPrintIter(Tree *tree){
  int i;
  Stack stack = newStack();
  Stack *stackPtr = &stack;
  TreeNode *curr = tree->root;

  do {
    while (curr != NULL) {
      push(stackPtr, curr);
      curr = curr->left;
    }

    if (!isEmpty(stackPtr)) {
      curr = pop(stackPtr);
      for(i = curr->frequency; i != 0; --i) {
        printf("%d\n", curr->data);
      }
      curr = curr->right;
    }
  } while (curr != NULL || !isEmpty(stackPtr));
}
```

The two functions do the same thing, except one is a recursive implementation and the other is iterative. The recursive implementation makes use of the call-stack to print nodes in order, while the iterative version uses my simple stack. The iterative version is also one of the rare times a do…while loop makes sense. They both go as far left down the tree as they can to get the next lowest data value, print it according to its frequency, go right if they can, and repeat.



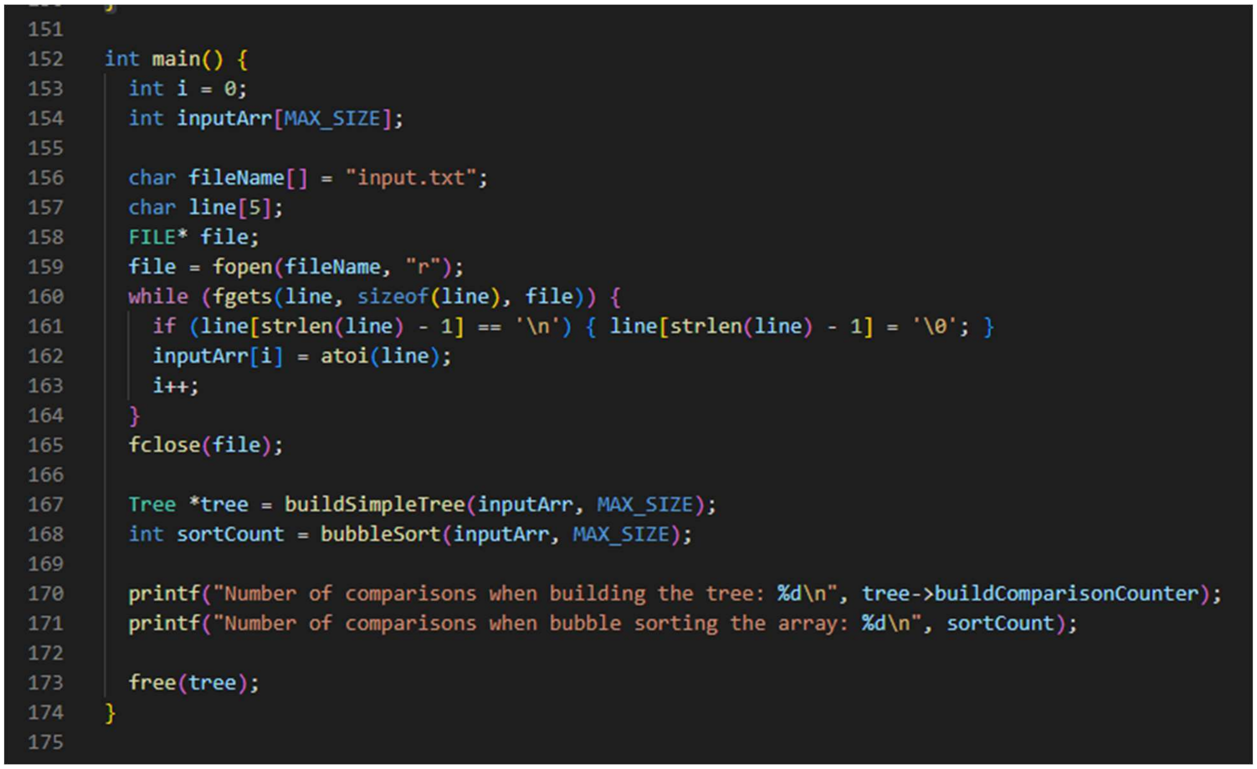
```
void freeTreeNode(TreeNode *treeNode) {
  if (treeNode->left != NULL) { freeTreeNode(treeNode->left); }
  if (treeNode->right != NULL) { freeTreeNode(treeNode->right); }
  free(treeNode);
}

void freeTree(Tree *tree) {
  freeTreeNode(tree->root);
  free(tree);
}
```

These functions free the allocated memory for the *Tree* and *TreeNode* data structures. The *freeTreeNode()* function traverses the tree very similarly as the recursive print function did, freeing the left and right child nodes if they exist, and then freeing the original node it was called with. The *freeTree()* function calls the *freeTreeNode()* function to free all the nodes and then frees the tree itself.



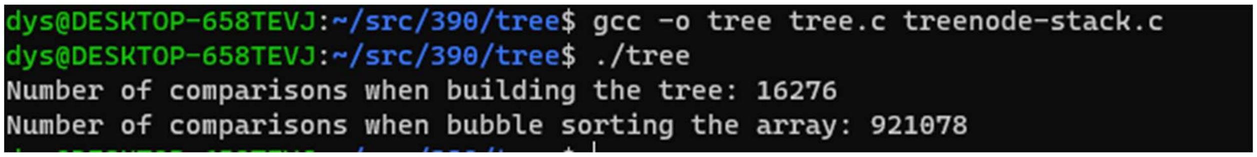
```c

int bubbleSort(int *arr, int arrLength) {
  int j, prev, curr, temp;
  int swapping = 1;
  int comparisonCount = 0;

  while(swapping) {
    swapping = 0;
    for(j = 1; j != arrLength; ++j) {
      prev = arr[j - 1];
      curr = arr[j];
      if (prev > curr) {
        arr[j - 1] = curr;
        arr[j] = prev;
        swapping = 1;
      }

      comparisonCount++;
    }
  }

  return comparisonCount;
}

```

The *bubbleSort()* function is used to sort an array and return the comparison counts. The function initializes a flag *swapping* to 1, indicating that at least one swap operation will occur during the first iteration. It then enters a loop that continues until no swaps are made during an iteration. In each iteration of the outer loop the function loops through the array comparing adjacent elements. If the element on the left is greater than the element on the right, the two elements are swapped and *swapping* is set to 1, indicating that a swap has been made. The function then returns the number of comparisons made during the sorting process.



```c
int main() {
    int i = 0;
    int inputArr[MAX_SIZE];

    char fileName[] = "input.txt";
    char line[5];
    FILE* file;
    file = fopen(fileName, "r");
    while (fgets(line, sizeof(line), file)) {
        if (line[strlen(line) - 1] == '\n') { line[strlen(line) - 1] = '\0'; }
        inputArr[i] = atoi(line);
        i++;
    }
    fclose(file);

    Tree *tree = buildSimpleTree(inputArr, MAX_SIZE);
    int sortCount = bubbleSort(inputArr, MAX_SIZE);

    printf("Number of comparisons when building the tree: %d\n", tree->buildComparisonCounter);
    printf("Number of comparisons when bubble sorting the array: %d\n", sortCount);

    free(tree);
}
```

The main function reads 1000 random integers from *input.txt*, converts the strings it reads to integers, and then appends them to an array. Then it builds the binary tree from that array, and then bubble sorts the array. Finally, it prints the comparison results and frees the allocated memory. Here are the results:



```
dys@DESKTOP-658TEVJ:~/src/390/tree$ gcc -o tree tree.c treenode-stack.c
dys@DESKTOP-658TEVJ:~/src/390/tree$ ./tree
Number of comparisons when building the tree: 16276
Number of comparisons when bubble sorting the array: 921078
```

Based on these results, ordering this type of data is clearly more efficient when done via building a binary tree than when bubble sorting an array.