DATA STRUCTURE 1 REPORT

Math Interpreter

DR.SALEH ELSHEHABY

OMAR MOHAMED – YEHIA SALAH-YOUSSEF HANY-KARIM ANANI-AHMED ELFAHAM

Table of Contents

Abstract:

The Interpreter implementation is based on the use of functions (caller/callee concept-modules), data structure (BST, Stacks) and headers.
Our main objectives were:

- Containerization of the code thus guarantee the reusability of the modules.
- Error handling and cover all the possible cases
- Reaching the maximum optimization and performance possible.

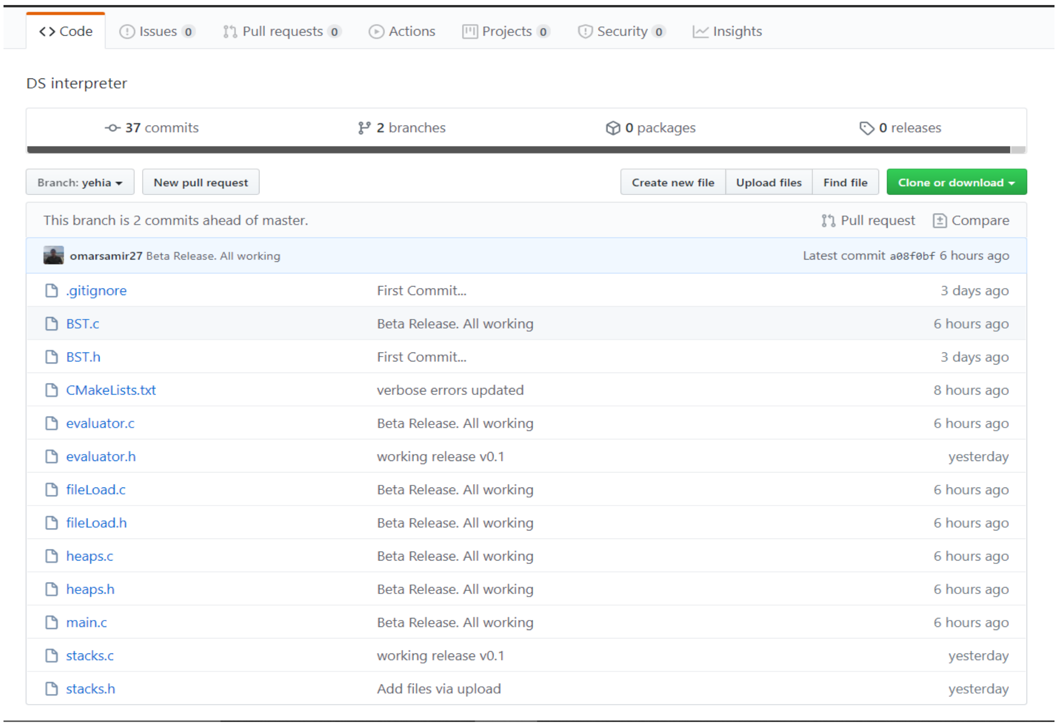
Implementation steps:

- Write pseudo code of the main modules and i/o charts.
- Split the project into main Files (sources + headers).
- The modules implementation of each file.
- Enhancing time complexity (reduced from O(n^2) to O(n*Log[n]).
- Exceptions handling and covering all possible scenarios.
- Code annotation, improving program features and make it user friendly.

Teamwork methodology:

"Adding manpower to a late software project makes it later"

- Split code files on members (every member works individually on assigned file).
- Set the communications between the code containers and preset the functions prototypes.
- Combining all project files.
- Use of version control platform (GitHub) to deliver the code.



DS interpreter

37 commits · 2 branches · 0 packages · 0 releases

This branch is 2 commits ahead of master.

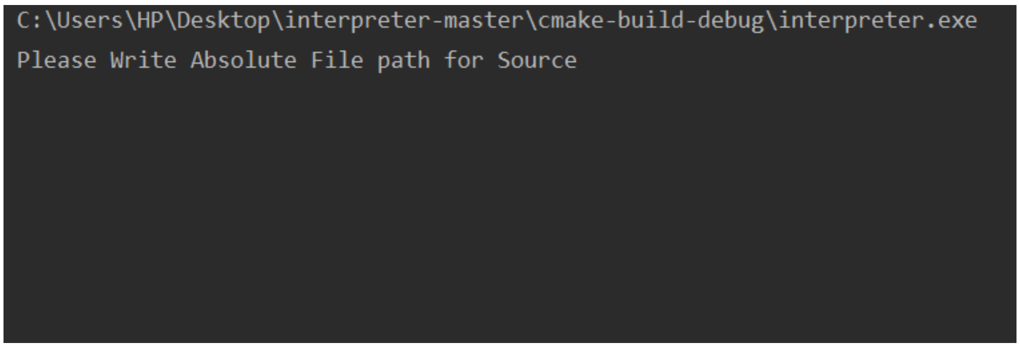
omarsamir27 Beta Release. All working — Latest commit a08f0bf 6 hours ago

.gitignore	First Commit...	3 days ago
BST.c	Beta Release. All working	6 hours ago
BST.h	First Commit...	3 days ago
CMakeLists.txt	verbose errors updated	8 hours ago
evaluator.c	Beta Release. All working	6 hours ago
evaluator.h	working release v0.1	yesterday
fileLoad.c	Beta Release. All working	6 hours ago
fileLoad.h	Beta Release. All working	6 hours ago
heaps.c	Beta Release. All working	6 hours ago
heaps.h	Beta Release. All working	6 hours ago
main.c	Beta Release. All working	6 hours ago
stacks.c	working release v0.1	yesterday
stacks.h	Add files via upload	yesterday

Functionality:

Overview:

The programs id divided into six main files:



```
C:\Users\HP\Desktop\interpreter-master\cmake-build-debug\interpreter.exe
Please Write Absolute File path for Source
```

* console picture

1. Console interface (main)

Contains the main of the program and the console interface code as pic. above.

2. File load

Load the data from the file and return a BST and heap to main.

3. Evaluator

Rebuild the RHS string and process an omni-check and evaluate it unless corrupted.

4. Stacks

Convert RHS from infix to postfix and evaluate it.

5. BST

Store the variable name and its key using BST, sorted alphabetically

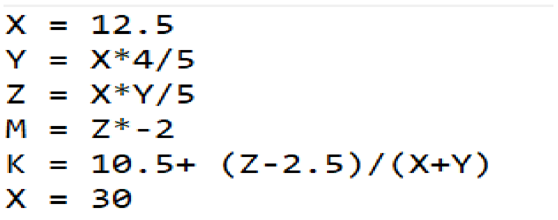
6. Heaps

Convert the BST into array and heapsort it.

Files:

File (.txt) is used as data buffer.

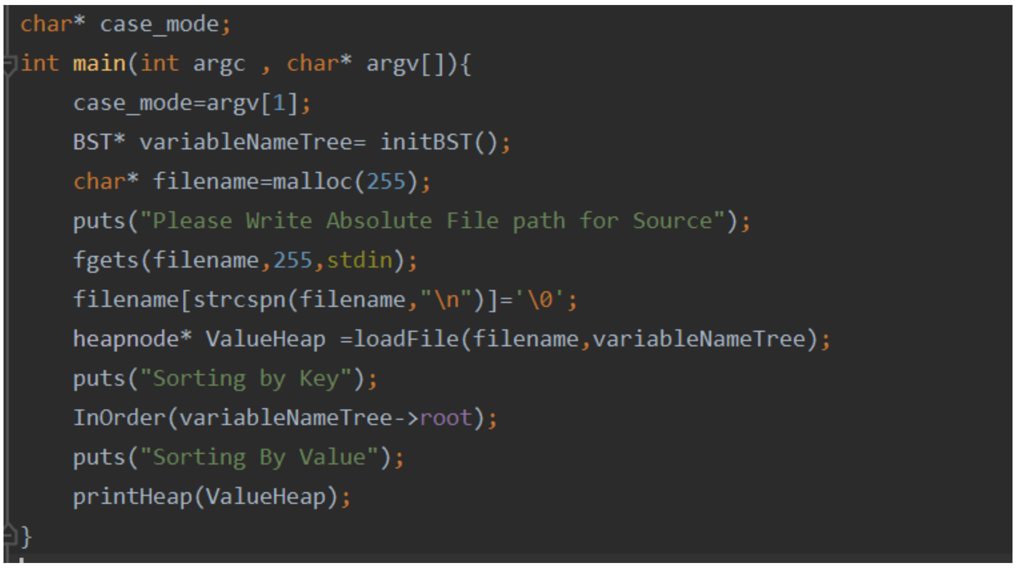
The Text contains the equations (data) that will be interpreted.


```
X = 12.5
Y = X*4/5
Z = X*Y/5
M = Z*-2
K = 10.5+ (Z-2.5)/(X+Y)
X = 30
```

** In-depth view of the code**

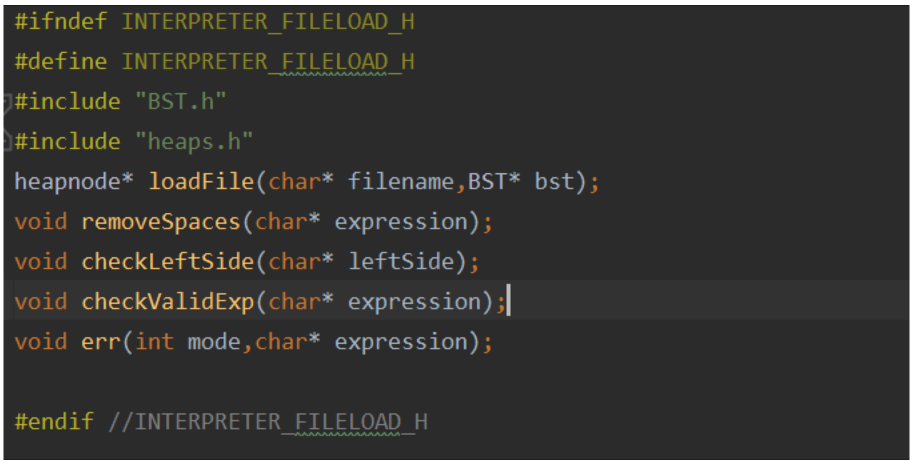
A. Console interface (main)

The main is the start function it prints the user-console interface, take the file path and sends it to loadfile function... It takes argument 'case' from the user if he wants the variables to be case-sensitve.


```
char* case_mode;
int main(int argc , char* argv[]){
    case_mode=argv[1];
    BST* variableNameTree= initBST();
    char* filename=malloc(255);
    puts("Please Write Absolute File path for Source");
    fgets(filename,255,stdin);
    filename[strcspn(filename,"\n")]='\0';
    heapnode* ValueHeap =loadFile(filename,variableNameTree);
    puts("Sorting by Key");
    InOrder(variableNameTree->root);
    puts("Sorting By Value");
    printHeap(ValueHeap);
}
```

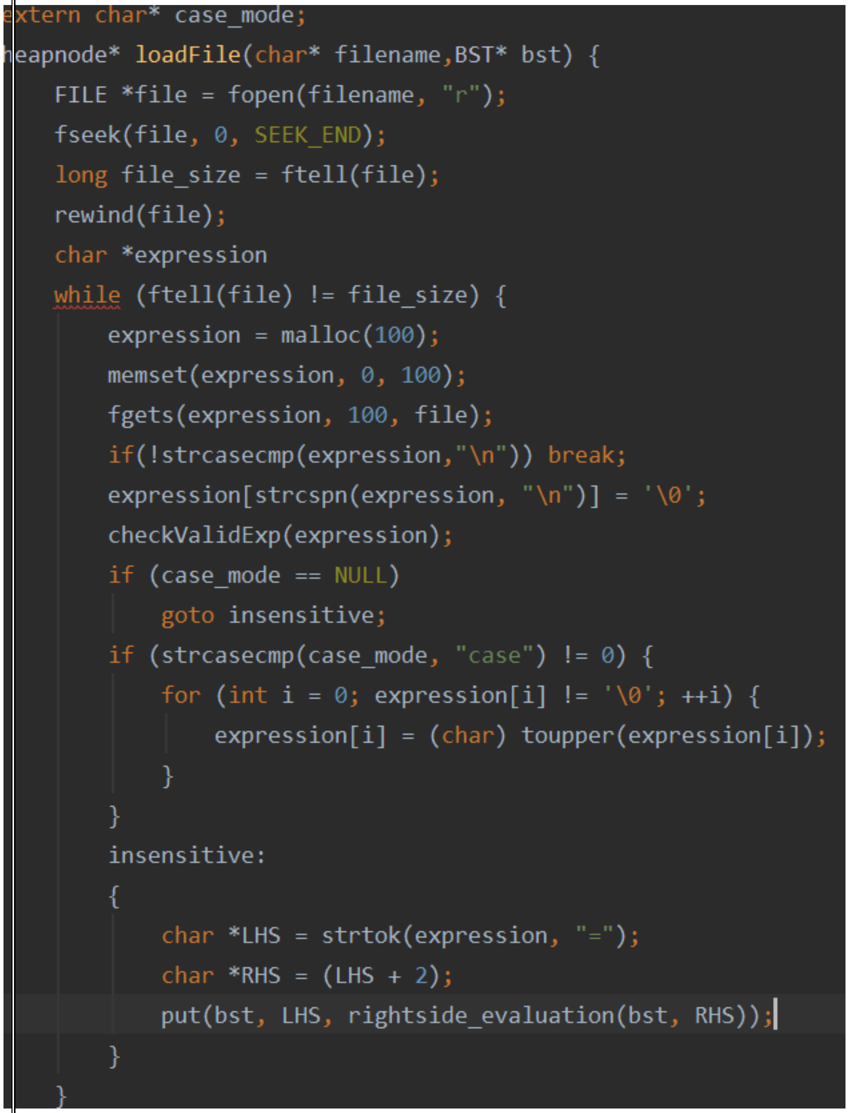
B. *File load*

Fileload file has 5 functions, its objective is to open the .txt file, collects the data, evaluate LHS, fill the BST, and send the RHS to evaluator.



```c
#ifndef INTERPRETER_FILELOAD_H
#define INTERPRETER_FILELOAD_H
#include "BST.h"
#include "heaps.h"
heapnode* loadFile(char* filename,BST* bst);
void removeSpaces(char* expression);
void checkLeftSide(char* leftSide);
void checkValidExp(char* expression);
void err(int mode,char* expression);

#endif //INTERPRETER_FILELOAD_H
```

The main function of the code block: -

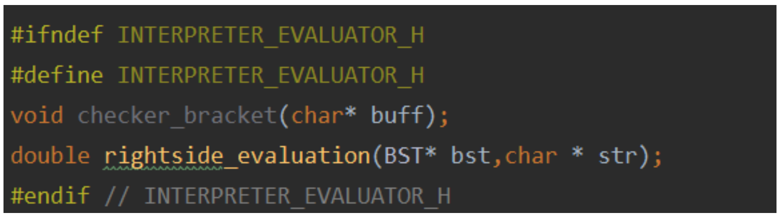


```c
xtern char* case_mode;
eapnode* loadFile(char* filename,BST* bst) {
    FILE *file = fopen(filename, "r");
    fseek(file, 0, SEEK_END);
    long file_size = ftell(file);
    rewind(file);
    char *expression
    while (ftell(file) != file_size) {
        expression = malloc(100);
        memset(expression, 0, 100);
        fgets(expression, 100, file);
        if(!strcasecmp(expression,"\n")) break;
        expression[strcspn(expression, "\n")] = '\0';
        checkValidExp(expression);
        if (case_mode == NULL)
            goto insensitive;
        if (strcasecmp(case_mode, "case") != 0) {
            for (int i = 0; expression[i] != '\0'; ++i) {
                expression[i] = (char) toupper(expression[i]);
            }
        }
        insensitive:
        {
            char *LHS = strtok(expression, "=");
            char *RHS = (LHS + 2);
            put(bst, LHS, rightside_evaluation(bst, RHS));
        }
    }
```

- Load file function return a *heapnode* pointer(heap array after being filled).
- It loops on the file line by line : check the validity of LHS then send it *rightside_evaluation* function
- If LHS and RHS are valid then it create a tree node of the corresponding variable in LHS.
- After the completion of the tree it calls heapsort function to create the heap arr and sort it.

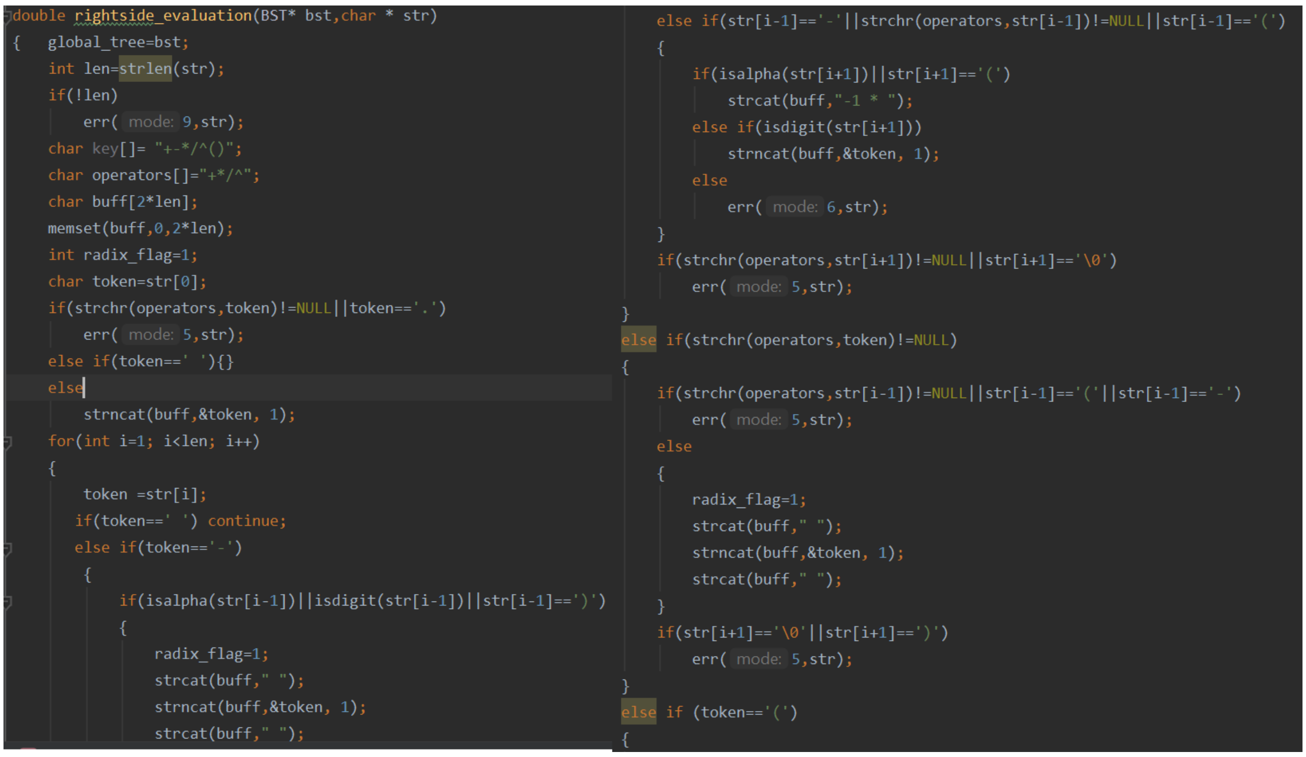
C. Evaluator:

Evaluator file checks the RHS string, rebuild it and send it to be evaluated.



```
#ifndef INTERPRETER_EVALUATOR_H
#define INTERPRETER_EVALUATOR_H
void checker_bracket(char* buff);
double rightside_evaluation(BST* bst,char * str);
#endif // INTERPRETER_EVALUATOR_H
```

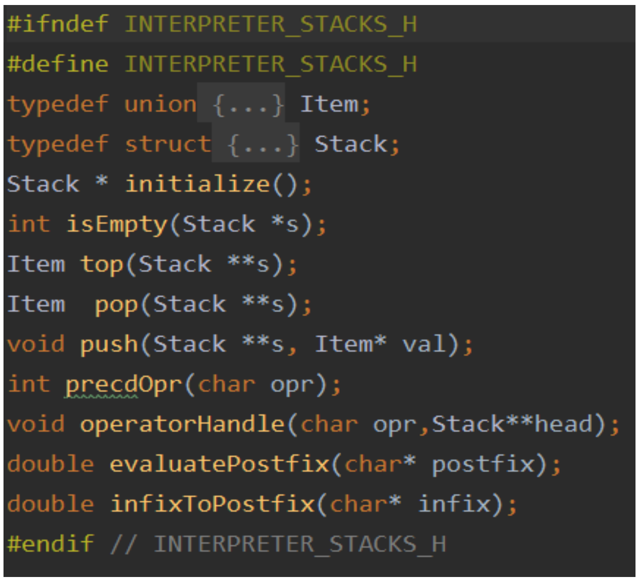
The main function of the code block: -



```
double rightside_evaluation(BST* bst,char * str)
{   global_tree=bst;
    int len=strlen(str);
    if(!len)
        err( mode: 9,str);
    char key[]= "+-*/^()";
    char operators[]="+*/^";
    char buff[2*len];
    memset(buff,0,2*len);
    int radix_flag=1;
    char token=str[0];
    if(strchr(operators,token)!=NULL||token=='.')
        err( mode: 5,str);
    else if(token==' '){}
    else
        strncat(buff,&token, 1);
    for(int i=1; i<len; i++)
    {
        token =str[i];
       if(token==' ') continue;
       else if(token=='-')
        {
            if(isalpha(str[i-1])||isdigit(str[i-1])||str[i-1]==')')
            {
                radix_flag=1;
                strcat(buff," ");
                strncat(buff,&token, 1);
                strcat(buff," ");
            else if(str[i-1]=='-'||strchr(operators,str[i-1])!=NULL||str[i-1]=='(')
            {
                if(isalpha(str[i+1])||str[i+1]=='(')
                    strcat(buff,"-1 * ");
                else if(isdigit(str[i+1]))
                    strncat(buff,&token, 1);
                else
                    err( mode: 6,str);
            }
            if(strchr(operators,str[i+1])!=NULL||str[i+1]=='\0')
                err( mode: 5,str);
        }
        else if(strchr(operators,token)!=NULL)
        {
            if(strchr(operators,str[i-1])!=NULL||str[i-1]=='('||str[i-1]=='-')
                err( mode: 5,str);
            else
            {
                radix_flag=1;
                strcat(buff," ");
                strncat(buff,&token, 1);
                strcat(buff," ");
            }
            if(str[i+1]=='\0'||str[i+1]==')')
                err( mode: 5,str);
        }
        else if (token=='(')
        {
```

- *Rightside_evaluation* function checks the RHS string if valid, then pass it to *infixtopostfix* function to be evaluated and return the result.
- The function structure is based on nested if condition to check each character of the string and compare it to the char around it.
- The function loops on every character, check it, and rebuilt a new string with comma delimiter.
- If space it skips, it calls error function if double operator, unbalance parentheses, radix point error, inappropriate varname are found.

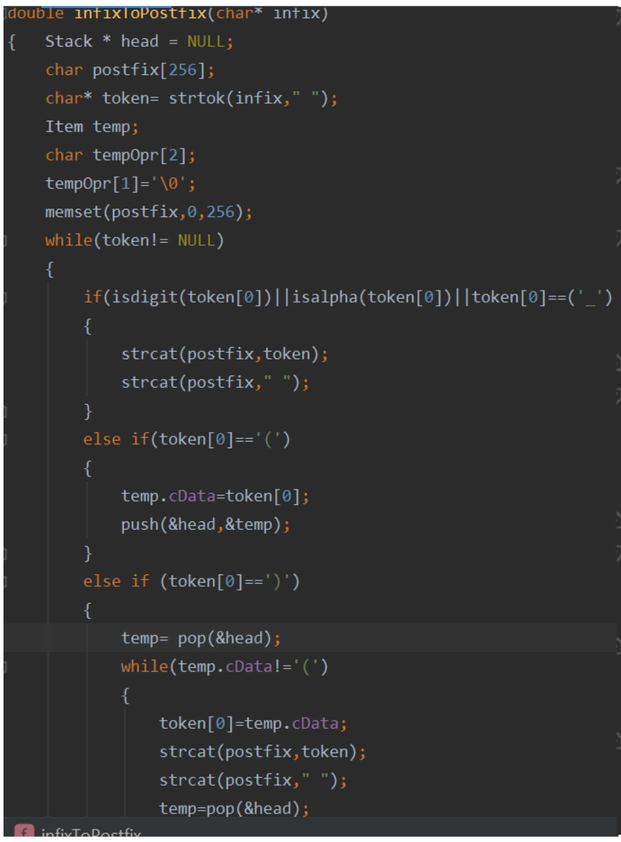
D. Stacks



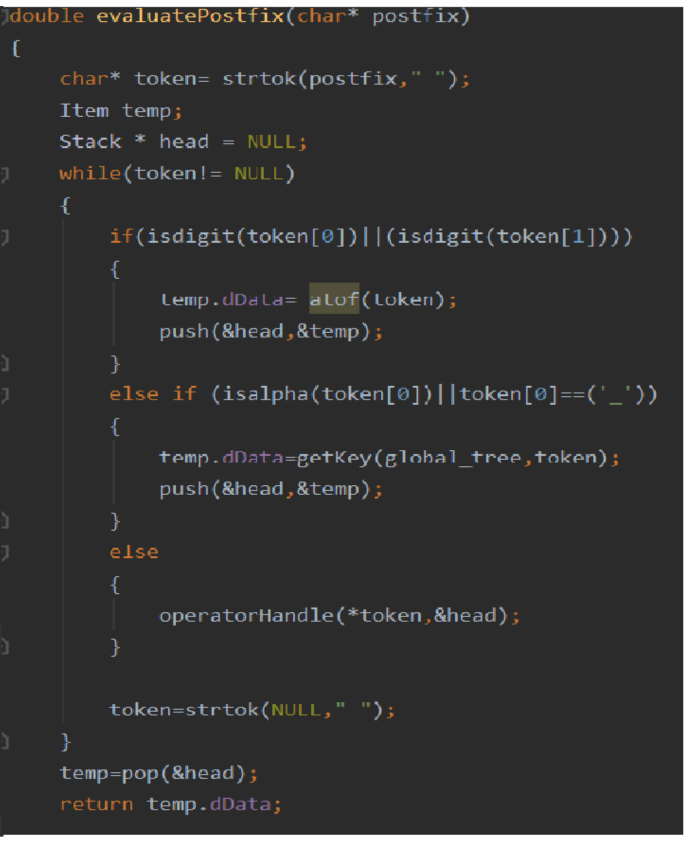
```
#ifndef INTERPRETER_STACKS_H
#define INTERPRETER_STACKS_H
typedef union {...} Item;
typedef struct {...} Stack;
Stack * initialize();
int isEmpty(Stack *s);
Item top(Stack **s);
Item  pop(Stack **s);
void push(Stack **s, Item* val);
int precdOpr(char opr);
void operatorHandle(char opr,Stack**head);
double evaluatePostfix(char* postfix);
double infixToPostfix(char* infix);
#endif // INTERPRETER_STACKS_H
```

Stacks file transform the RHS-edited string from infix to postfix then it evaluates the postfix and returns the results.

The main functions of the code block: -



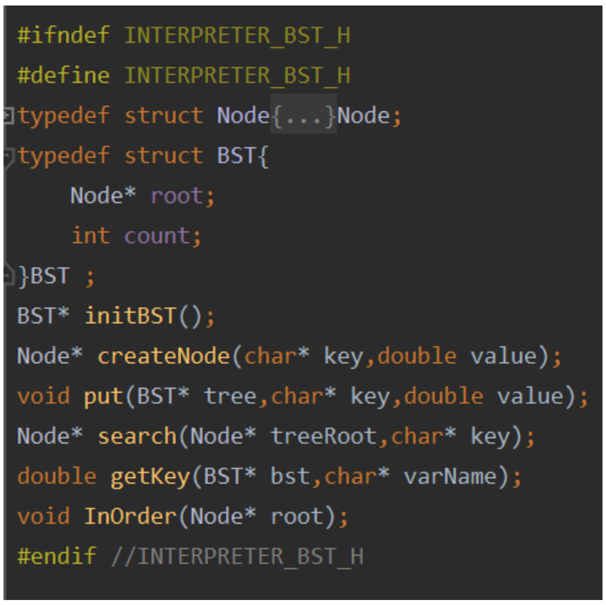
```
double infixToPostfix(char* infix)
{   Stack * head = NULL;
    char postfix[256];
    char* token= strtok(infix," ");
    Item temp;
    char tempOpr[2];
    tempOpr[1]='\0';
    memset(postfix,0,256);
    while(token!= NULL)
    {
        if(isdigit(token[0])||isalpha(token[0])||token[0]==('_'))
        {
            strcat(postfix,token);
            strcat(postfix," ");
        }
        else if(token[0]=='(')
        {
            temp.cData=token[0];
            push(&head,&temp);
        }
        else if (token[0]==')')
        {
            temp= pop(&head);
            while(temp.cData!='(')
            {
                token[0]=temp.cData;
                strcat(postfix,token);
                strcat(postfix," ");
                temp=pop(&head);
```



```
double evaluatePostfix(char* postfix)
{
    char* token= strtok(postfix," ");
    Item temp;
    Stack * head = NULL;
    while(token!= NULL)
    {
        if(isdigit(token[0])||(isdigit(token[1])))
        {
            temp.dData= atof(token);
            push(&head,&temp);
        }
        else if (isalpha(token[0])||token[0]==('_'))
        {
            temp.dData=getKey(global_tree,token);
            push(&head,&temp);
        }
        else
        {
            operatorHandle(*token,&head);
        }

        token=strtok(NULL," ");
    }
    temp=pop(&head);
    return temp.dData;
```

- *Infixtopostfix* function take a string in form of infix and transform it into postfix using shunt-yard algorithm.
- *Evaluatepostfix* take postfix and evaluate it using stacks and return the result.

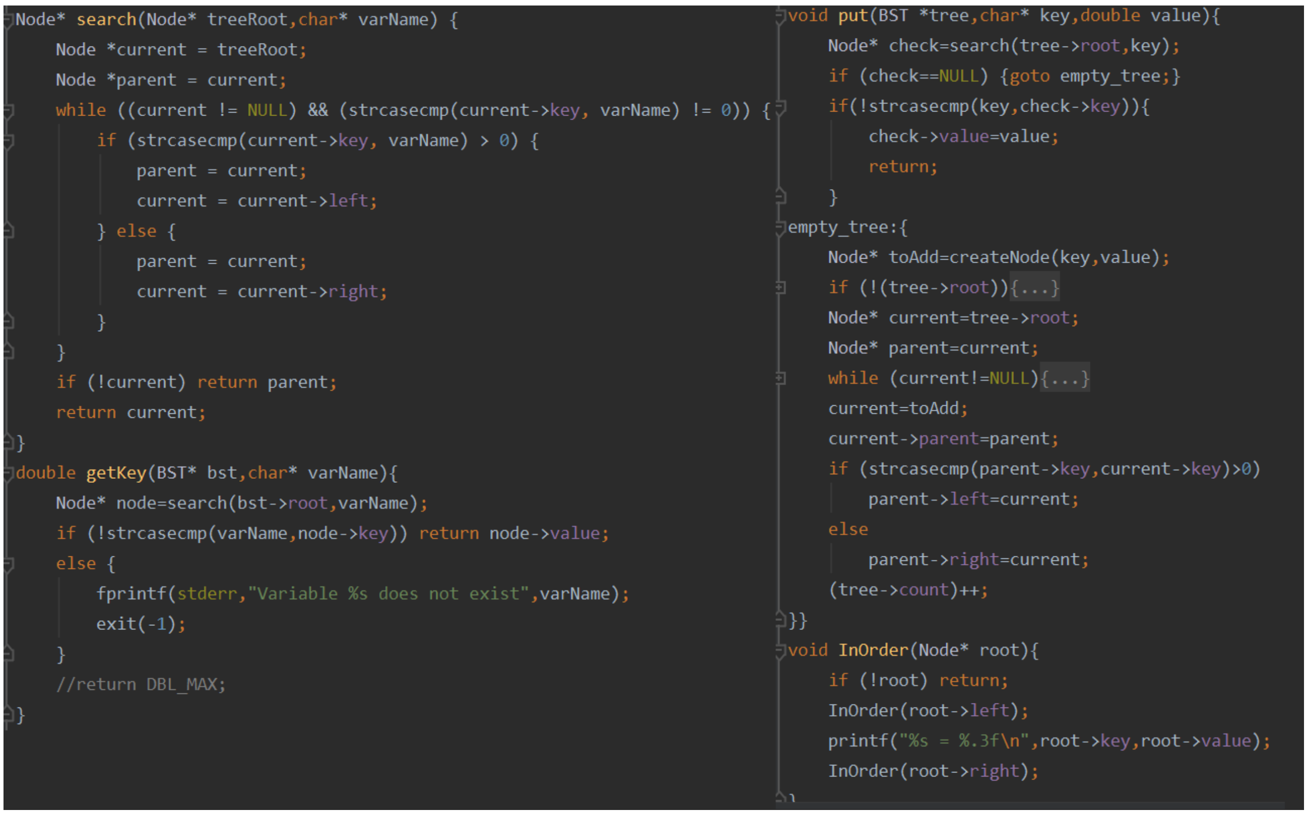
7. BST



```c
#ifndef INTERPRETER_BST_H
#define INTERPRETER_BST_H
typedef struct Node{...}Node;
typedef struct BST{
    Node* root;
    int count;
}BST ;
BST* initBST();
Node* createNode(char* key,double value);
void put(BST* tree,char* key,double value);
Node* search(Node* treeRoot,char* key);
double getKey(BST* bst,char* varName);
void InOrder(Node* root);
#endif //INTERPRETER_BST_H
```

BST file creates a bst tree, its objective is to insert a tree node for every variable, overwrite it if inserted for more than once. Search for a node and return the value of it

The main functions of the code block: -

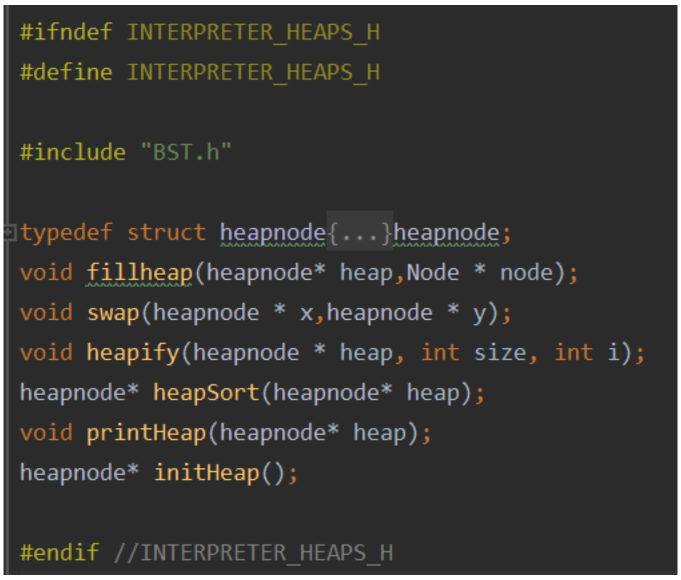


```c
Node* search(Node* treeRoot,char* varName) {
    Node *current = treeRoot;
    Node *parent = current;
    while ((current != NULL) && (strcasecmp(current->key, varName) != 0)) {
        if (strcasecmp(current->key, varName) > 0) {
            parent = current;
            current = current->left;
        } else {
            parent = current;
            current = current->right;
        }
    }
    if (!current) return parent;
    return current;
}
double getKey(BST* bst,char* varName){
    Node* node=search(bst->root,varName);
    if (!strcasecmp(varName,node->key)) return node->value;
    else {
        fprintf(stderr,"Variable %s does not exist",varName);
        exit(-1);
    }
    //return DBL_MAX;
}
```

```c
void put(BST *tree,char* key,double value){
    Node* check=search(tree->root,key);
    if (check==NULL) {goto empty_tree;}
    if(!strcasecmp(key,check->key)){
        check->value=value;
        return;
    }
empty_tree:{
    Node* toAdd=createNode(key,value);
    if (!(tree->root)){...}
    Node* current=tree->root;
    Node* parent=current;
    while (current!=NULL){...}
    current=toAdd;
    current->parent=parent;
    if (strcasecmp(parent->key,current->key)>0)
        parent->left=current;
    else
        parent->right=current;
    (tree->count)++;
}}
void InOrder(Node* root){
    if (!root) return;
    InOrder(root->left);
    printf("%s = %.3f\n",root->key,root->value);
    InOrder(root->right);
```

- Put function creates a tree node, it takes key and value as argument …if the node is found it overwrites it.
- Get key function return the value of the varnode searched for.
- In Order function prints the nodes of tree In Order (from a to z)

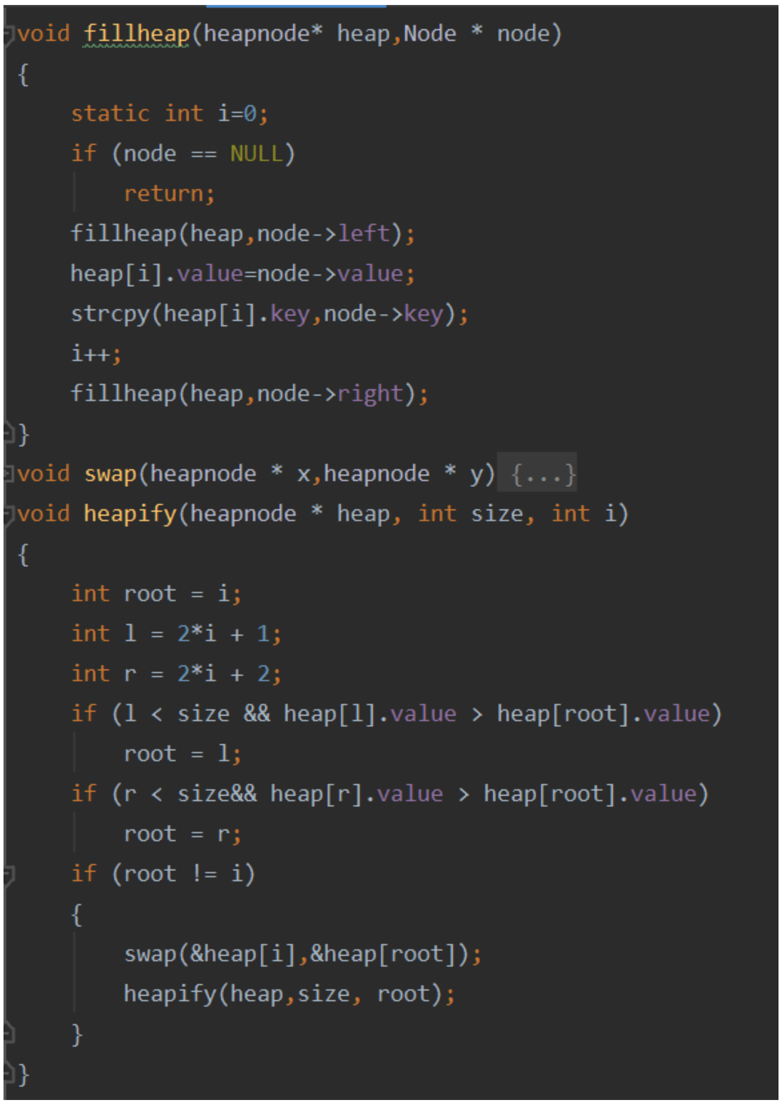
8. Heaps



```
#ifndef INTERPRETER_HEAPS_H
#define INTERPRETER_HEAPS_H

#include "BST.h"

typedef struct heapnode{...}heapnode;
void fillheap(heapnode* heap,Node * node);
void swap(heapnode * x,heapnode * y);
void heapify(heapnode * heap, int size, int i);
heapnode* heapSort(heapnode* heap);
void printHeap(heapnode* heap);
heapnode* initHeap();

#endif //INTERPRETER_HEAPS_H
```

Heaps file creates an array from BST nodes and sorts it using heapsort

The main functions of the code block: -



```
void fillheap(heapnode* heap,Node * node)
{
    static int i=0;
    if (node == NULL)
        return;
    fillheap(heap,node->left);
    heap[i].value=node->value;
    strcpy(heap[i].key,node->key);
    i++;
    fillheap(heap,node->right);
}
void swap(heapnode * x,heapnode * y) {...}
void heapify(heapnode * heap, int size, int i)
{
    int root = i;
    int l = 2*i + 1;
    int r = 2*i + 2;
    if (l < size && heap[l].value > heap[root].value)
        root = l;
    if (r < size&& heap[r].value > heap[root].value)
        root = r;
    if (root != i)
    {
        swap(&heap[i],&heap[root]);
        heapify(heap,size, root);
    }
}
```

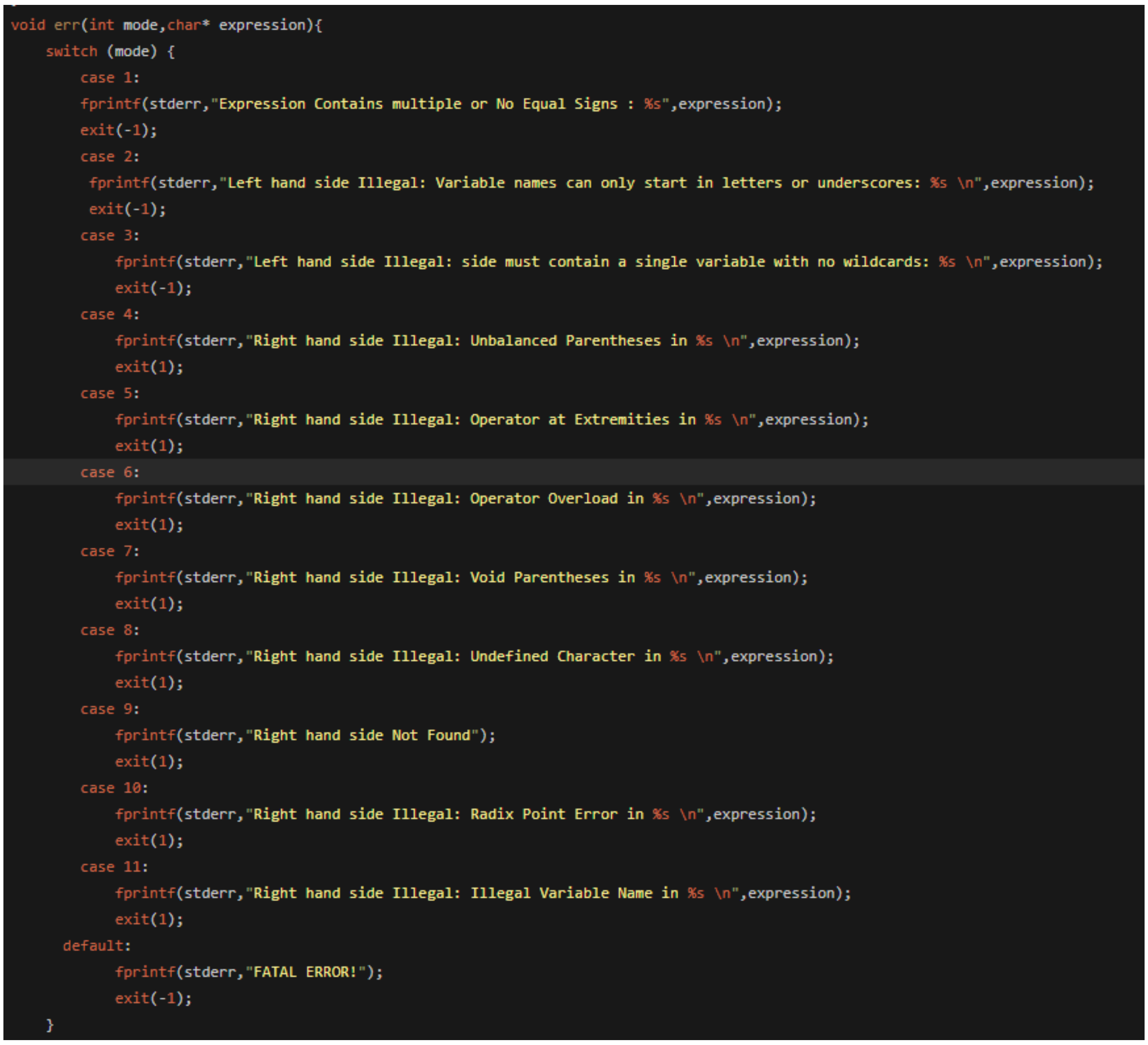
- *Fillheap* fills the array using a recursive method (same as inorder traversal for traverse all nodes in tree).
- Static int is used to overcome recursive method and i value does not change in recursion.
- *Heapify* function rearrange a heap to maintain the heap property, the key of the root node is more extreme (greater or less) than or equal to the keys of its children.

Key Techniques:

1. **Modularization*:***
   The program is split into several functions each with one single responsibility to enable reuse and make bug tracking easier.

2. ***Error Handling:***

   err() function implemented to output appropriate verbose errors for the user so it is easier to find errors in source files. err() is invoked from appropriate functions when an error occurs. The arguments specify the error message and the source.



```c
void err(int mode,char* expression){
    switch (mode) {
        case 1:
        fprintf(stderr,"Expression Contains multiple or No Equal Signs : %s",expression);
        exit(-1);
        case 2:
         fprintf(stderr,"Left hand side Illegal: Variable names can only start in letters or underscores: %s \n",expression);
         exit(-1);
        case 3:
            fprintf(stderr,"Left hand side Illegal: side must contain a single variable with no wildcards: %s \n",expression);
            exit(-1);
        case 4:
            fprintf(stderr,"Right hand side Illegal: Unbalanced Parentheses in %s \n",expression);
            exit(1);
        case 5:
            fprintf(stderr,"Right hand side Illegal: Operator at Extremities in %s \n",expression);
            exit(1);
        case 6:
            fprintf(stderr,"Right hand side Illegal: Operator Overload in %s \n",expression);
            exit(1);
        case 7:
            fprintf(stderr,"Right hand side Illegal: Void Parentheses in %s \n",expression);
            exit(1);
        case 8:
            fprintf(stderr,"Right hand side Illegal: Undefined Character in %s \n",expression);
            exit(1);
        case 9:
            fprintf(stderr,"Right hand side Not Found");
            exit(1);
        case 10:
            fprintf(stderr,"Right hand side Illegal: Radix Point Error in %s \n",expression);
            exit(1);
        case 11:
            fprintf(stderr,"Right hand side Illegal: Illegal Variable Name in %s \n",expression);
            exit(1);
      default:
            fprintf(stderr,"FATAL ERROR!");
            exit(-1);
    }
```

3. ***Global Variables:***

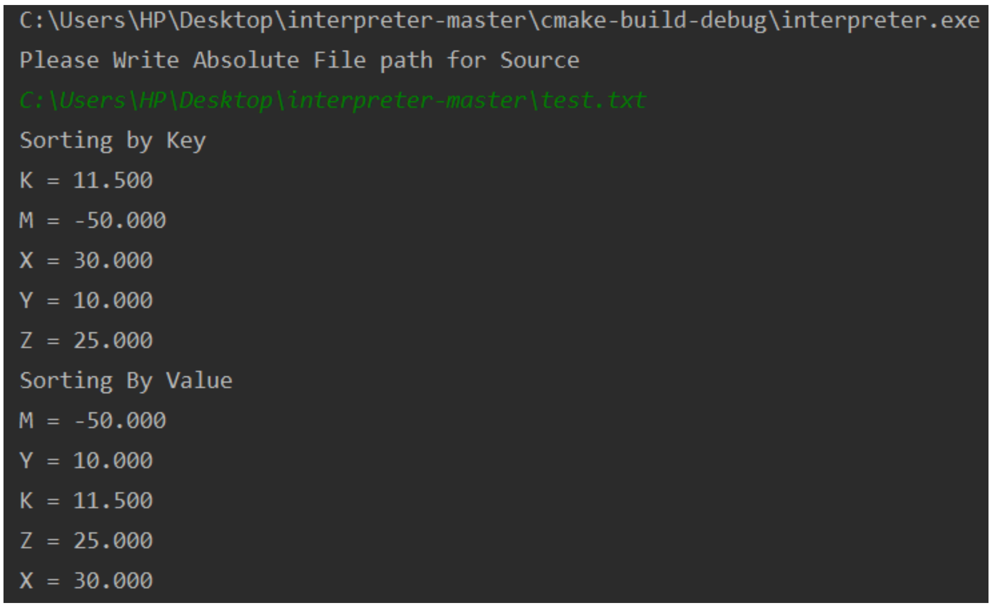
Global Variables were used to provide a global access point to source files and primary data structures on which the program flow and operaiton depend on.
A Global Variable is also used to determine the mode of operation of the program to be case sensitive or case insensitive.
The extern keyword is used to share global variables across several C source files.

Results:

The program works properly:

- All Functionality is tested and working.
- Special Cases are tested and working.

Running Demo:



```
C:\Users\HP\Desktop\interpreter-master\cmake-build-debug\interpreter.exe
Please Write Absolute File path for Source
C:\Users\HP\Desktop\interpreter-master\test.txt
Sorting by Key
K = 11.500
M = -50.000
X = 30.000
Y = 10.000
Z = 25.000
Sorting By Value
M = -50.000
Y = 10.000
K = 11.500
Z = 25.000
X = 30.000
```

References:

GUIDES.GITHUB.COM

STACKOVERFLOW.COM

TUTORIALSPOINT.COM

INTRODUCTION TO ALGORITHMS-THOMAS H. CORMEN

ALGORITHMS IN C-ROBERT SEDGEWICK