Contents

Introduction

In this project, I was tasked to build a program using the object-oriented programming paradigm. The main aspects of OOP are;

- **Class & Objects**: OOP is based on classes, which define the logic, attributes and implications of objects and how they interact with other parts of the program.
- **Encapsulation**: Technique used to keep data where it will not interfere with other data and their implementations. It also prevents the misuse of code.
- **Polymorphism**: An important aspect of OOP. Polymorphism enabling objects to have multiple forms, so it can be used in different scenarios.
- **Inheritance**: Mechanism that allows objects to acquire properties and methods from another class called a base class.

The project that I have programmed includes the use of all these aspects, and much more;

Error Handling:

Used the try and catch blocks for all of the functions inside the main .cpp file.

Operator overloading:

Mainly used in the variable class. Operator overloading allows objects to be able to operate on each other.

Data parsing:

Used to separate the string input from the user to be able to define what function they would like to call. The first 4 words of the input are assigned to a variable each. This allows the program to compare and figure what method to call. As I have mentioned, only the first 4 words are assigned. This is the case because the maximum length of a function call is 4 words. But the algorithm is flexible, so any number of words can be assigned to separate variables.

Mapping:

Since the objects are created at run time, the concept of data paring had to be used to be able to relate the name of the data structure acquired from the user to its memory reference. Mapping was also used for pointers created by the users for the lists.

Data conversion:

Conversion of data. Mainly input string from user to integer.

Usage of header files:

Each class implementation has a header and a .cpp file, which makes everything much more organized as the program is very big.

Dynamic allocation for objects:

All objects are created at the user's command and the memory allocated for all objects are managed.

Memory allocation of multiple pointers for lists (DLL & SLL):

Ability to traverse through the lists using multiple pointers with different names. The pointers are all created at run time.

Dynamic casting:

Used to access operator overloading of sub classes as a pointer of base class was used to access sub classes.

STL functions:

A lot of different functions were used from the standard library. Some include the algorithm library to use sort and many more.

Problem Statement

Program and design a program using the object-oriented paradigm to create data structures that have are a part of a class with their own methods, as well as inherited methods from a base class to achieve the complete use of the object-oriented paradigm.

Functional Requirements

The concepts of data structures that includes methods such as;

Class	Functionality
Base	• Create a named data structure • Delete a named data structure • Display contents of a data structure • Search for an element • Sort the elements • Clone a data structure

Stack	• Push an item into the stack. • Remove an item from the stack. • Print out the value at the top of the stack.

	• Method to check if stack is full. • Method to check if stack is empty.
Queue	• Push an item into the queue. • Remove an item from the stack. • Print out the value at the front of the queue. • Print out the value at the back of the queue. • Method to check if queue is full. • Method to check if queue is empty
Binary Search Tree	• Add item to tree • Remove an item from the tree. • print out the root of the tree. • traverse the tree (in-order.) • traverse the tree (post-order.) • traverse the tree (pre-order.)
Singly Linked	• Add a node to the end. • Delete a node from the end. • Add a node to the front. • Delete a node from the front. • Insert a value after a node. • Point to the beginning of the list. • Get the value at a pointer and store it in variable. • Move a pointer forward. • Check if pointer reached the end of list.

	• Print out a message based on a Boolean value.
Doubly Linked List	• Add a node to the end. • Delete a node from the end. • Add a node to the front. • Delete a node from the front. • Insert a value after a node. • Point to the beginning of the list. • Get the value at a pointer and store it in variable. • Move a pointer forward. • Move a pointer backward. • Check if pointer reached the end or head of the list setting the value of a Boolean variable.
Vector	• Declare a variable • Assign a value to a variable • Assign a variable to another variable
	• Add a value to a variable • Add a variable to another variable • Subtract value from a variable • Subtract a variable from another variable • Multiply variable by a value • Multiply variable by another variable • Divide var by value • Divide var by another variable • Mod var by value

Use of Inheritance & Polymorphism

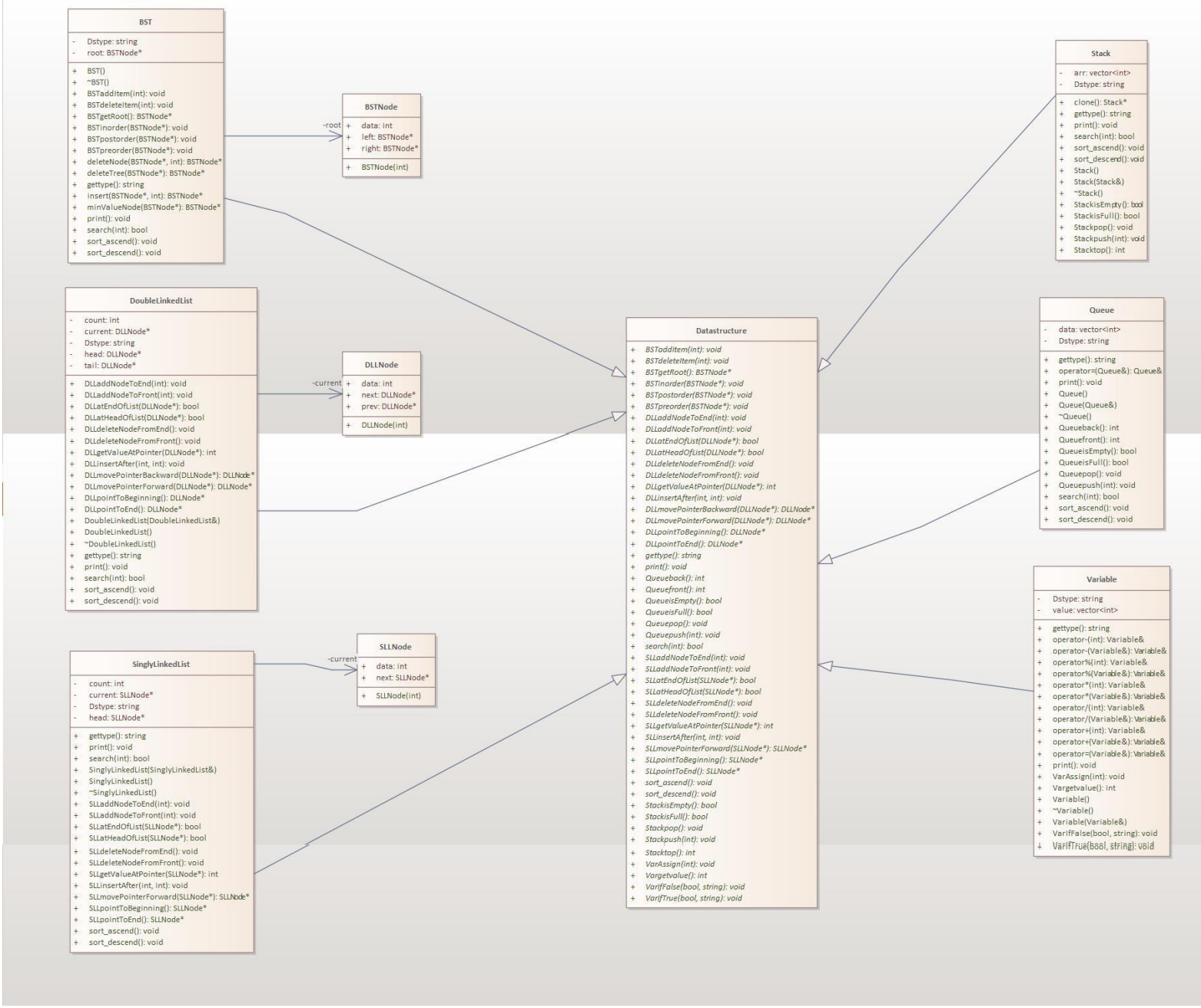
Polymorphisms:

Every single object created uses run-time polymorphism. A virtual function for almost all the methods of each class is contained inside the base class. These virtual methods are called at run-time and are overridden by their respective object for its specific use. Furthermore, pure virtual methods of the same name (print, sort ascend/descend, gettype) are overridden during runtime as well, as all the data structures contain these methods.

Inheritance:

All data structures are children of a base class called “Datastructure”. This base class contains all the methods of respective classes as virtual methods, and some methods as pure virtual, as they are available inside every data structure. An example of this would be the “Print” method which prints out the contents of a data structure.

UML DIAGRAM

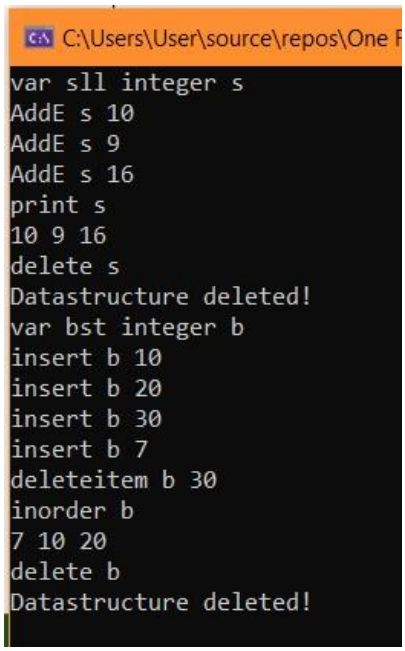


For full quality image please click this google drive link (MMU STAFF Only).

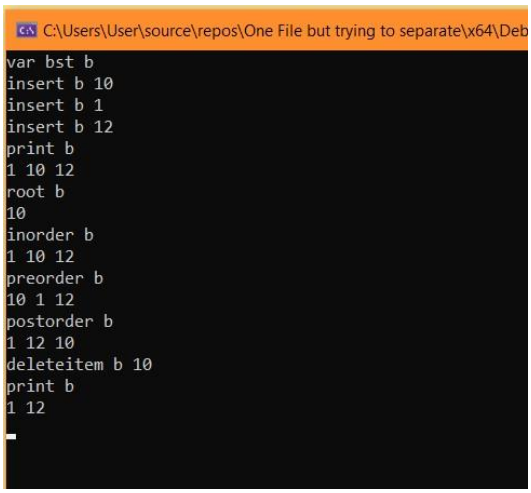
For personal email access please click here.

Screenshots

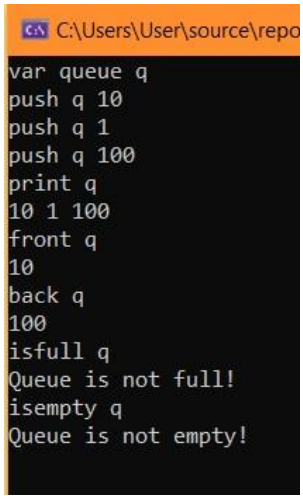
SLL:



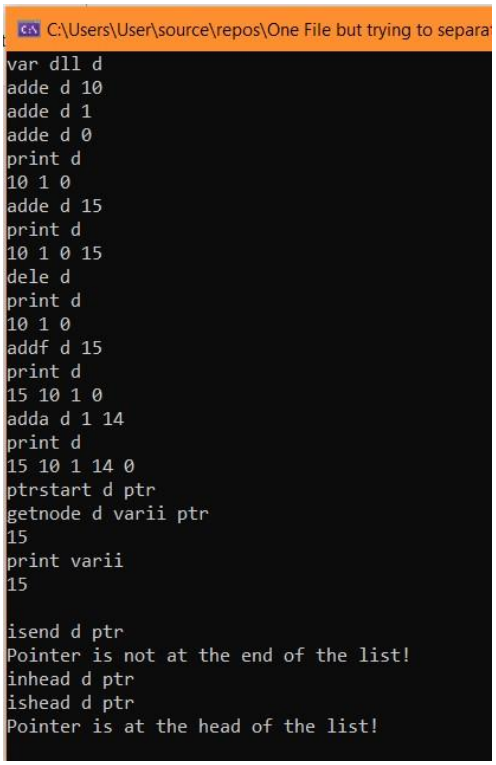
BST:

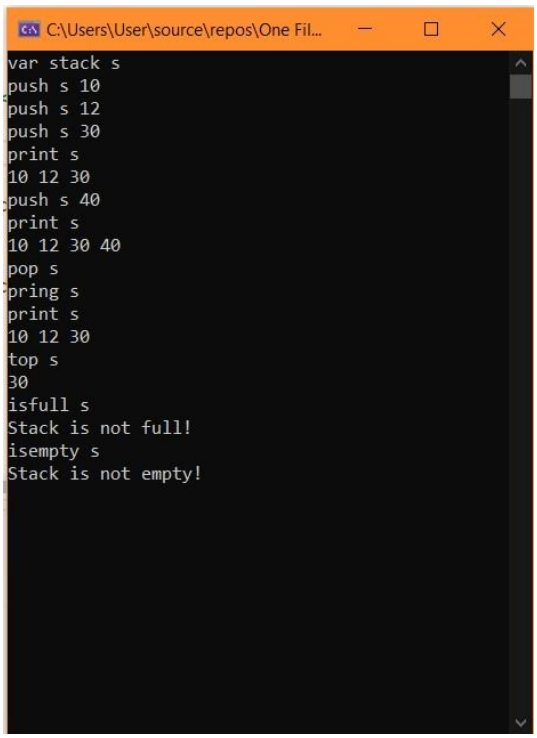


Queue:



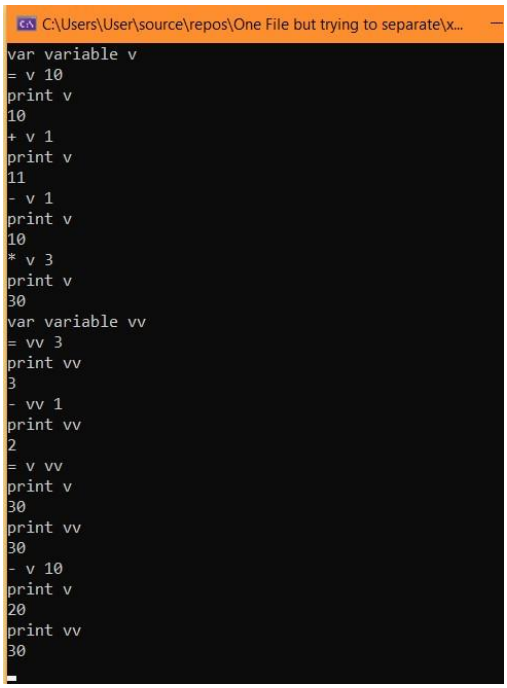
DLL:





Stack:

Variable:



Documentation & Usage:

Function	Data Structure	Sample
Data structure declaration	Stack	Var stack integer [name]
Data structure declaration	Variable	Var Variable integer [name]
Data structure declaration	Queue	Var Queue integer [name]
Data structure declaration	BST	Var Bst integer [name]
Data structure declaration	DLL	Var DLL integer [name]
Data structure declaration	SLL	Var SLL integer [name]
Clear Screen		cls
Delete data structure	ALL	Delete [DS name]
Clone data structure	ALL	Clone [DS Name] [DS Clone Name]
Print	ALL	Print [DS Name]
Sort	ALL	Sort ascend/descend [DS Name]
Assigning variable to value	Variable	= varii 3
Assigning variable to variable	Variable	= vari varii

Any operation on variable. Ex: *, +, 0 etc.	Variable	[Operation (+, -, /)] [Operand] [Operator] ex: (= Varii vari)
Push value	Stack & Queue	Push [Qname] [Value] Ex: push qqq 3
Pop value	Stack & Queue	Pop [DSname] Ex: pop qqq
Print front	Queue	Front [DS name]
Print back	Queue	Front [DS name]
Check if full	Stack & Queue	Isfull [DS name]
Check if empty	Stack & Queue	Isempty [DS name]
Insert	BST	Insert [DS name] [Value]
Delete an item from BST	BST	Deleteitem [DS Name] [Value]
Return BST root	BST	Root [DS Name]
Postorder	BST	Postorder mybst
Preorder	BST	Preorder mybst
inorder	BST	Inorder mybst
Add node to end	DLL & SLL	AddE [DS Name] [Value] ex: Adde hello 3
Add node to front	DLL & SLL	Addf [DS Name] [Value] Ex: addf mydll 340
Delete node from end	DLL & SLL	Dele [DS Name] [Value] ex: dele haggg
Insert after a value	DLL & SLL	Adda [DS Name] [AfterthisValue] [Value]
Point to beginning (creates a pointer)	DLL & SLL	Ptrstart [DS name] [pointer name]
Get value at node	DLL & SLL	Getnode [DS name] [Variable name] [Ptr name]
		Note: pointer must be made before getting the value at the pointer. Value will be stored in the variable name
Point to next node	DLL & SLL	Nextnode [DS Name] [ptr name] Disclaimer: Pointer must have been created using ptrstart method.
Check if ptr is at the end or head of list.	DLL & SLL	Isend/ishead [DS name] [PTR name]

Get value at node sample:

Structure of program:

About SLL & DLL:

There are a few things that should be known about SLL & DLL. Firstly, when “Ptrstart” method is used, a new pointer will be made and added to either of these dictionaries;

```
unordered_map<string, DLLNode* > DLLpointer;
unordered_map<string, SLLNode* > SLLpointer;
```

Syntax for reference; **Ptrstart [DS name] [pointer name]**.

The user will pick a name for the pointer. This would be just like creating a new data structure.

This allows for the ability to create multiple pointers within DLL & SLL. So, another pointer could be created with a different name that would also point at the beginning. Methods like; Nextnode & Isend could be used on any pointer of choice.

Syntax of NextNode for reference: **Nextnode [DS Name] [ptr name]**.

As shown above in the reference, the user would have to input the data structure name and the pointer name. The pointer name is the name they picked when they created the pointer at Ptrstart. This is just another feature that allows user to create as many pointers as they want.

When cloning data structures:

Whenever the clone method is called, the copy constructor of each data structure will be used. Although, the copy constructor cannot be called using the base class pointer, since all data structures created are pointer at using a base class pointer. To get around this, down-casting is used to get a pointer of sub class to be able to call the copy constructor.

How objects are created:

```
Datastructure* newBST() {
    Datastructure* ptr = new BST();
    return ptr;
};
```

As shown in the image, there are functions for creating a new of each data structure. All of them are stored using a “Datastructure” pointer, which is the base class.

Whenever the “var” method is used to create a new object, the name the user picks will be used as a key

```
}
try {
    if (contains_number) {
        num = stod(word3);//getting pointer to sub class and assigning it to itself and subtracting the number specified in the input.
        dynamic_cast<Variable*>(DS[word2])->VarAssign(dynamic_cast<Variable*>(DS[word2])->Vargetvalue() - num);
    }
```

to the memory location of the objects.

```
    if (word1 == "var") {
        if (word2 == "bst") {
            DS[word3] = newBST();
        }
        else if (word2 == "dll") {
            DS[word3] = newDLL();
        }
        else if (word2 == "sll") {
            DS[word3] = newSLL();
        }
        else if (word2 == "stack") {
            DS[word3] = newStack();
        }
        else if (word2 == "queue") {
            DS[word3] = newQueue();
        }
        else if (word2 == "variable") {
            DS[word3] = newVariable();

        };
    }
```

Syntax for reference: **var bst [DS Name]**.

DS is the dictionary that stores all the references to the object. So, as soon as a new object is made, they are added to the DS dictionary by making it equal to the functions that were shown before, the functions that return the base class pointer and creates a new object dynamically. DS dictionary declaration;

```
unordered_map<string, Datastructure* > DS;
```

As you can see, the key is a string (the name of the data structure) and the value stored is a “Datastructure “pointer, which is basically the reference we have to the newly created object.

Variable data structure:

The parts that require the use of operation overloading for the Variable data structure all use downcasting to be able to access the operation overloading for the data structure, as the objects reference is stored using a base class pointer to achieve polymorphism.

Parsing algorithm:

```
list<string> parseInput(const string& input) {
    list<string> words;
    string word = "";
    for (const char& c : input) {
        if (c != ' ') {
            word += c;
        }
        else {
            transform(word.begin(), word.end(), word.begin(), ::tolower);
            words.push_back(word);
            word = "";
        }
    }
    transform(word.begin(), word.end(), word.begin(), ::tolower);
    words.push_back(word);
    return words;
}
```

The parsing algorithm stores the first four words as word1-word4 using a list. It also transforms the words to lowercase so they can be used with ease.

Catch blocks and error handling:

There are multiple catch blocks in the main function. Mainly for catching exceptions thrown for when the user tries to access a data structure that has not been created (accessing a key in the map without it existing.) There are also if statements that don't allow exceptions to be thrown in the first place, incase they are handled and an error message would be shown.

Example:

```
else if (word1 == "adde") {
    if (DS.count(word2) == 0) {
        cout << "Error: The key " << word2 << " is not found in the map" << endl;
        continue;
```

How to read from file:

Type any command in the txt file (input.txt) and save it. Then type "file" in the program, and it will read the file.

Example:

Var bst b

Save file type “file”
in program

Command is executed

Problems encountered

There were numerous problems I encountered while making this program. There have been over 15 versions of the project, building from just the queue data structure to slowly increasing the program.

The main problems that were encountered are;

- Implementation all object functions to be done at run time.
- Memory management.
- Making a connection between the data structure name and the data structure reference.
- Methods of type data structure (for example Stack returnvalue();) cannot be a virtual method in the base class (although of type data structure pointer could be virtual). This caused me a lot of problems. There were numerous compiler errors that were impossible to understand.
- I ran in to a lot of problems while trying to point methods of sub classes using base class pointer. They were small at first, but became a big problem as the program kept growing.
- Problems encountered when trying to access the operator overloading of variable class. Its impossible to access operator overloading of sub classes using base classes. Dynamic casting had to be used to get a pointer of sub class from base class pointer.
- By far, the biggest problem was managing the whole program myself. The program has over 2000 lines and it took a lot of versions to get to where it is. Getting lost in the program was quite easy. After a lot of parts were connected together, a single change in the program was able to create multiple compiler errors.
- Circular definition: while trying to separate the program from one cpp file into different files. I came across the weirdest compiler errors. Program would work fine, and then I would add a new data structure to it and it would stop working. I remove the newly added data structure and

  the problem was still there, even though that wasn’t the case before adding the new data structure.
- Trying to access class constructors using base class pointers.

Conclusion:

Overall. I came across so many problems and obstacles. But, I was able to overcome almost all of them. These problems were the building blocks to my initiative to learn more. One problem would lead to learning multiple things that were related to it, and other problems.

The outcome of this project was a whole lot more than just a program. I was able to learn a lot about OOPD and inshallah a lot more in the future.

Having this project was a great curve ball for me. Maybe if it weren't for this project, I would not have ever had the will to try this hard for anything.

Reference:

Application used for UML Diagram: Enterprise Architect.

IDE used: Visual Studio.

Some of the video references (Playlist):
https://youtube.com/playlist?list=PLoWvJO8072bGOz_2h42KwanoxaoGwjNNQ

Source code

Main.cpp

```
#include "Datastructure.h"

#include "BST.h"

#include "DoubleLinkedList.h"

#include "SinglyLinkedList.h"

#include "Stack.h"

#include "Queue.h"

#include "Variable.h"

#include <unordered_map>

#include <cmath>

#include <sstream>

#include <cctype>
```

```
#include <list> //used for parsing algorithm
#include <cstdlib>
#include <fstream> using
namespace std;



//parsing algorithm. Storing values in a list and transforming the words to lowercase.


list<string> parseInput(const string& input) {
list<string> words;     string word = "";     for
(const char& c : input) {
       if (c != ' ') {
          word += c;
       }
       else {
          transform(word.begin(), word.end(), word.begin(), ::tolower);
words.push_back(word);
          word = "";
       }
    }
    transform(word.begin(), word.end(), word.begin(), ::tolower);
words.push_back(word);     return words;
}



//functions to create each DS dynamically. Each of them gets a pointer of the sub class and assigns it to
pointer of base class (Datastructure)
//This is where run time polymorphism is used
```

```
Datastructure* newBST() {
   Datastructure* ptr = new BST();
   return ptr;
};

Datastructure* newDLL() {
   Datastructure* ptr = new DoubleLinkedList();
   return ptr;

};

Datastructure* newSLL() {
   Datastructure* ptr = new SinglyLinkedList();
   return ptr;
};

Datastructure* newStack() {
Datastructure* ptr = new Stack();
   return ptr;
};

Datastructure* newQueue() {
Datastructure* ptr = new Queue();
   return ptr;
};

Datastructure* newVariable() {
Datastructure* ptr = new Variable();
   return ptr;
```

```
};

int main() {
   //map for storing all datastructure addresses. One map is only used because all datastructure addresses are stored using the BASE class.
   unordered_map<string, Datastructure* > DS;

   //maps for pointers. The pointers store the current location of a user defined pointer in either a SLL or a DLL. Map is used because the user
   //can make as many pointers as they want for the SLL and DLL, and use each pointer to traverse the datastructures.
   //The map stores the address of each location and uses its name as a key.
   unordered_map<string, DLLNode* > DLLpointer;
unordered_map<string, SLLNode* > SLLpointer;


   string input;
double num;
double num2;    bool
DLLisend;    bool
SLLisend;    bool
DLLishead;    bool
SLLishead;

   fstream inputfile;
```

```
   //while loop that continues until the input is "end"
while (input != "end") {



      getline(cin, input);
if (input == "end") {
         break;
      }
      list<string> words = parseInput(input);
string word1, word2, word3, word4;        int i = 1;


      //adds first 4 words, the rest are ignored.
      for (auto& word : words) {
         switch (i) {
case 1:              word1
= word;
            break;
case 2:              word2
= word;
            break;
case 3:              word3
= word;
            break;
case 4:              word4
= word;              break;
         }
         if (i == 4) break;
```

```
        i++;
    }


    //reading from a file and changing values for word1-word4.
    if (word1 == "file") {
inputfile.open("Input.txt", ios::in); if
(inputfile.is_open()) {
        string line; while
(getline(inputfile, line)) {
list<string> words = parseInput(line);
            int i = 1;
            for (auto& word : words) {
              switch (i) {
case 1:                 word1
= word;                  break;
case 2:                 word2
= word;                  break;
case 3:                 word3
= word;
                  break;
case 4:                 word4
= word;
                  break;
}
              if (i == 4) break;
i++;
            }
```

```
            }


};
        inputfile.close();


    }
```

```
    //creating a datastructure. Depending on the user's choice, the right function for making a new DS is called and the datastructure
    //is created at run time.
if (word1 == "var") {           if
(word2 == "bst") {
            DS[word3] = newBST();
        }
        else if (word2 == "dll") {
            DS[word3] = newDLL();
        }
        else if (word2 == "sll") {
            DS[word3] = newSLL();
        }
        else if (word2 == "stack") {
            DS[word3] = newStack();
        }
        else if (word2 == "queue") {
            DS[word3] = newQueue();
        }
        else if (word2 == "variable") {
```

```
            DS[word3] = newVariable();


        };
    }

    //clear screen       else
if (word1 == "cls") {
system("cls");

    }

    //deleting datastructure.       else if (word1 == "delete") {          if
(DS.count(word2) == 0) {              cout << "Error: The key " << word2 << " is not
found in the map" << endl;
            continue;
        }
try {
            delete DS[word2];              cout <<
"Datastructure deleted!" << endl;
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
        catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }
```

```
    //printing datastructure        else if (word1 == "print") {            if
(DS.count(word2) == 0) {                cout << "Error: The key " << word2 << " is not
found in the map" << endl;
            continue;
        }
try {
            if (DS.count(word2) > 0)
            {
                DS[word2]->print(); cout << endl;
            }
else
                cout << "Datastructure does not exist" << endl;
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }



    //sorting datastructure. Syntax is: sort ascend/descend [DS NAME]        else if
(word1 == "sort") {            if (DS.count(word2) == 0) {                cout << "Error: The
key " << word2 << " is not found in the map" << endl;
            continue;
        }
try {
            if (word2 == "ascend") {
```

```
            DS.at(word3)->sort_ascend();
        }
        else if (word2 == "descend") {
            DS.at(word3)->sort_descend();
        }
    }
    catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
    }
    catch (...) {
        cout << "An unknown error occurred" << endl;
    }
  }


  //cloning datastructure.    else if (word1 == "clone") {        if
(DS.count(word2) == 0) {          cout << "Error: The key " << word2 << " is not
found in the map" << endl;
        continue;
    }
    if (DS[word2]->gettype() == "bst") {
        BST* derivedPtr = dynamic_cast<BST*>(DS[word2]);
        DS[word3] = new BST(*derivedPtr);
    }

    else if (DS[word2]->gettype() == "dll") {
        DoubleLinkedList* derivedPtr = dynamic_cast<DoubleLinkedList*>(DS[word2]);
DS[word3] = new DoubleLinkedList(*derivedPtr);
```

```
        }
        else if (DS[word2]->gettype() == "sll") {
            SinglyLinkedList* derivedPtr = dynamic_cast<SinglyLinkedList*>(DS[word2]);
DS[word3] = new SinglyLinkedList(*derivedPtr);
```

```
        }

        else if (DS[word2]->gettype() == "stack") {
            Stack* derivedPtr = dynamic_cast<Stack*>(DS[word2]);
            DS[word3] = new Stack(*derivedPtr);
        }
        else if (DS[word2]->gettype() == "queue") {
            Queue* derivedPtr = dynamic_cast<Queue*>(DS[word2]);
            DS[word3] = new Queue(*derivedPtr);
        }

        else if (DS[word2]->gettype() == "var") {
            Variable* derivedPtr = dynamic_cast<Variable*>(DS[word2]);
DS[word3] = new Variable(*derivedPtr);

        };

    }

    //---------------------------------------Methods for the variable datastructure; adding, making equal to, subtracting, dividing, multiplying & modding.

    else if (word1 == "=") {        if (DS.count(word2) == 0) {            cout <<
"Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        }
        bool contains_number = false;
for (int i = 0; i < word3.length(); i++) {
if (isdigit(word3[i])) {
contains_number = true;
```

```
                break;
            }
}           try
{
             if (contains_number) {
num = stod(word3);
                DS.at(word2)->VarAssign(num);
            }
else {
                Variable* derivedPtr = dynamic_cast<Variable*>(DS[word3]); //downcasting to access copy
constructor
                DS[word3] = derivedPtr;
                DS[word3]->VarAssign(DS[word2]->Vargetvalue());
            }
          }
          catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
          }
          catch (...) {
             cout << "An unknown error occurred" << endl;
          }
       }



      else if (word1 == "+") {            if (DS.count(word2) == 0) {                cout <<
"Error: The key " << word2 << " is not found in the map" << endl;
             continue;
          bool contains_number = false;
```

```
        }


        for (int i = 0; i < word3.length(); i++) {
if (isdigit(word3[i])) {
contains_number = true;
            break;
          }
}          try
{
           if (contains_number) {
num = stod(word3);
              dynamic_cast<Variable*>(DS.at(word2))-
>VarAssign(dynamic_cast<Variable*>(DS.at(word2))->Vargetvalue() + num);
          }
else {
              dynamic_cast<Variable*>(DS.at(word2))-
>VarAssign(dynamic_cast<Variable*>(DS.at(word2))->Vargetvalue() +
dynamic_cast<Variable*>(DS.at(word3))->Vargetvalue());
          }
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
catch (...) {
          cout << "An unknown error occurred" << endl;
        }
      }
```

```
else if (word1 == "-") {            if (DS.count(word2) == 0) {                cout <<
"Error: The key " << word2 << " is not found in the map" << endl;
                continue;
            }
            bool contains_number = false;
            for (int i = 0; i < word3.length(); i++) { //checks if word3 contains a number. if its a number, an
assumption is made that the user wants to subtract a value to the varaible.
                if (isdigit(word3[i])) {
contains_number = true;
                    break;
                }
}           try
{
                if (contains_number) {
                    num = stod(word3);//getting pointer to sub class and assigning it to itself and subtracting the
number specified in the input.
                    dynamic_cast<Variable*>(DS[word2])-
>VarAssign(dynamic_cast<Variable*>(DS[word2])>Vargetvalue() - num);
                }
else {
                    //using dynamic casting to be able to access the sub class pointers. And then using Varassign
to assign the values retrieved from word2(operand)  and word3(operator)
                    dynamic_cast<Variable*>(DS[word2])-
>VarAssign(dynamic_cast<Variable*>(DS[word2])>Vargetvalue() - dynamic_cast<Variable*>(DS[word3])-
>Vargetvalue());
                }
            }
            catch    (const    out_of_range&    e)    {
cout << "Error: " << e.what() << endl; catch (...)
{           cout << "An unknown error occurred"
<< endl;
```

```
        }


        }
    }



    else if (word1 == "*") {        if (DS.count(word2) == 0) {            cout <<
"Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        }
        bool contains_number = false;
        for (int i = 0; i < word3.length(); i++) { //checks if word3 contains a number. if its a number, an
assumption is made that the user wants to subtract a value to the varaible.
            if (isdigit(word3[i])) {
contains_number = true;
                break;
            }
}         try
{
            if (contains_number) {
                num = stod(word3);//getting pointer to sub class and assigning it to itself and subtracting the
number specified in the input.
                dynamic_cast<Variable*>(DS[word2])-
>VarAssign(dynamic_cast<Variable*>(DS[word2])>Vargetvalue() * num);
            }
else {
                //using dynamic casting to be able to access the sub class pointers. And then using Varassign
to assign the values retrieved from word2(operand)  and word3(operator)
dynamic_cast<Variable*>(DS[word2])->VarAssign(dynamic_cast<Variable*>(DS[word2])>Vargetvalue() *
dynamic_cast<Variable*>(DS[word3])->Vargetvalue());
            }
        }
```

```
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
        catch (...) {
            cout << "An unknown error occurred" << endl;
        }

    }

    else if (word1 == "/") {        if (DS.count(word2) == 0) {            cout <<
"Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        }
        bool contains_number = false;
        for (int i = 0; i < word3.length(); i++) { //checks if word3 contains a number. if its a number, an
assumption is made that the user wants to subtract a value to the varaible.
            if (isdigit(word3[i])) {
contains_number = true;
                break;
            }
}         try
{
            if (contains_number) {
                num = stod(word3);//getting pointer to sub class and assigning it to itself and subtracting the
number specified in the input.
```

```
                dynamic_cast<Variable*>(DS[word2])-
>VarAssign(dynamic_cast<Variable*>(DS[word2])>Vargetvalue() / num);

            }

else {

                //using dynamic casting to be able to access the sub class pointers. And then using Varassign
to assign the values retrieved from word2(operand)  and word3(operator)

                dynamic_cast<Variable*>(DS[word2])-
>VarAssign(dynamic_cast<Variable*>(DS[word2])>Vargetvalue() / dynamic_cast<Variable*>(DS[word3])-
>Vargetvalue());

            }

        }

        catch (const out_of_range& e) {

cout << "Error: " << e.what() << endl;

        }

        catch (...) {

            cout << "An unknown error occurred" << endl;

        }


    }


    else if (word1 == "%") {           if (DS.count(word2) == 0) {               cout <<

"Error: The key " << word2 << " is not found in the map" << endl;

            continue;

        }

        int intnum = stod(word3);

bool contains_number = false;

        for (int i = 0; i < word3.length(); i++) { //checks if word3 contains a number. if its a number, an
assumption is made that the user wants to subtract a value to the varaible.

            if (isdigit(word3[i])) {

contains_number = true; break;
```

```
            }
}           try
{
            if (contains_number) {
                num = stod(word3);//getting pointer to sub class and assigning it to itself and subtracting the number specified in the input.
                dynamic_cast<Variable*>(DS[word2])->VarAssign(dynamic_cast<Variable*>(DS[word2])>Vargetvalue() % intnum);
            }
else {
                //using dynamic casting to be able to access the sub class pointers. And then using Varassign to assign the values retrieved from word2(operand)  and word3(operator)
                dynamic_cast<Variable*>(DS[word2])->VarAssign(dynamic_cast<Variable*>(DS[word2])>Vargetvalue() % dynamic_cast<Variable*>(DS[word3])->Vargetvalue());
            }
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
catch (...) {
            cout << "An unknown error occurred" << endl;
        }


    }



    //--------------------------------------Methods for DLL & SLL
```

```
    else if (word1 == "adde") {          if (DS.count(word2) == 0) {              cout <<
"Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        }
try {
            num = stod(word3);              if
(DS[word2]->gettype() == "dll") {
               DS[word2]->DLLaddNodeToEnd(num);
            }
            else if (DS[word2]->gettype() == "sll") {
               DS[word2]->SLLaddNodeToEnd(num);
            }
else {
               cout << "This datastructure does not support the method called" << endl;
            };
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
        catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }

    else if (word1 == "dele") {          if (DS.count(word2) == 0) {              cout <<
"Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        }
```

```
          try
{
            if (DS[word2]->gettype() == "dll") {
               DS[word2]->DLLdeleteNodeFromEnd();
            }
            else if (DS[word2]->gettype() == "sll") {
               DS[word2]->SLLdeleteNodeFromEnd();
            }
          }
          catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
          }
          catch (...) {
            cout << "An unknown error occurred" << endl;
          }
       }

      else if (word1 == "delf") {          if (DS.count(word2) == 0) {              cout <<
"Error: The key " << word2 << " is not found in the map" << endl;
            continue;
          }
try {
            if (DS[word2]->gettype() == "dll") {
               DS[word2]->DLLdeleteNodeFromFront();
            }
            else if (DS[word2]->gettype() == "sll") {
               DS[word2]->SLLdeleteNodeFromFront();
            }
            else {
```

```
            cout << "This datastructure does not support the method called" << endl;
        };
    }
    catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
    }
    catch (...) {
        cout << "An unknown error occurred" << endl;
    }
  }


  else if (word1 == "addf") {        if (DS.count(word2) == 0) {            cout <<
"Error: The key " << word2 << " is not found in the map" << endl;
        continue;
    }
try {
        if (DS[word2]->gettype() == "dll") {
num = stod(word3);
            DS[word2]->DLLaddNodeToFront(num); //needs to be casted
        }
        else if (DS[word2]->gettype() == "sll") {
num = stod(word3);
            DS[word2]->SLLaddNodeToFront(num); // needs to be casted
        }
else { cout <<
"This
datastructure
```

```
does not
support the
method
called" <<
endl;
            };
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }



    else if (word1 == "adda") {          if (DS.count(word2) == 0) {              cout <<
"Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        }
try {
            if (DS[word2]->gettype() == "dll") {
num = stod(word3);                  num2 =
stod(word4);
               DS[word2]->DLLinsertAfter(num2, num);
            }
```

```
            else if (DS[word2]->gettype() == "sll") {
num = stod(word3);                  num2 =
stod(word4);
                DS[word2]->SLLinsertAfter(num2, num);
            }
            else {
                cout << "This datastructure does not support the method called" << endl;
            };
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
        catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }



    // creates new pointer with name of (word3) and points at the start.
    else if (word1 == "ptrstart") {          if (DS.count(word2) == 0) {              cout
<< "Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        }
try {
            if (DS[word2]->gettype() == "dll") {
                DLLpointer[word3] = DS[word2]->DLLpointToBeginning(); //returns a ptr address that is at the
beginning of list
            }
```

```
            else if (DS[word2]->gettype() == "sll") {
               SLLpointer[word3] = DS[word2]->SLLpointToBeginning();
            }
else { cout <<
"This
datastructure
does not
support the
method
called" <<
endl;
            };
         }
         catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
         }
catch (...) {
            cout << "An unknown error occurred" << endl;
         }

      }


      // creates new pointer with name of (word3) and points at the start.
      else if (word1 == "ptrend") {          if (DS.count(word2) == 0) {              cout
<< "Error: The key " << word2 << " is not found in the map" << endl;
            continue;
```

```
        }
try {
            if (DS[word2]->gettype() == "dll") {
                DLLpointer[word3] = DS[word2]->DLLpointToEnd(); //returns a ptr address that is at the
beginning of list
            }
            else if (DS[word2]->gettype() == "sll") {
                SLLpointer[word3] = DS[word2]->SLLpointToEnd();
            }
else {
                cout << "This datastructure does not support the method called" << endl;
            };
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
        catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }
    //!!!new syntax of getnode : getnode [DSName] [VariableName] [PTRName]!!!!

    else if (word1 == "getnode") {          if (DS.count(word2) == 0) {              cout
<< "Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        }
try {
            if (DS[word2]->gettype() == "dll") {
```

```
                DS[word3] = newVariable();//new variable made based on input
                DS[word3]->VarAssign(DS[word2]->DLLgetValueAtPointer(DLLpointer[word4]));
                cout << DLLpointer[word4]->data;
cout << endl;
            }
            else if (DS[word2]->gettype() == "sll") {
                DS[word3] = newVariable();//new variable made based on input
                DS[word3]->VarAssign(DS[word2]->SLLgetValueAtPointer(SLLpointer[word4]));
cout << SLLpointer[word4]->data;

                cout << endl;
            }
            else {
                cout << "This datastructure does not support the method called" << endl;
            };
        }

        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
        catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }



    //must specify DSname to able to move ptr --> (nextnode [dll_DSname] [ptr name]).Disclaimer: Ptr must have had to be made by calling ptrstart.
    else if (word1 == "nextnode") {          if (DS.count(word2) == 0) {
cout << "Error: The key " << word2 << " is not found in the map" << endl;
```

```
            continue;
        }
try {
            if (DS[word2]->gettype() == "dll") {
                DLLpointer[word3] = DS[word2]->DLLmovePointerForward(DLLpointer[word3]);
            }

            else if (DS[word2]->gettype() == "sll") {
                SLLpointer[word3] = DS[word2]->SLLmovePointerForward(SLLpointer[word3]);

}              else
{
                cout << "This datastructure does not support the method called" << endl;
            };
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
        catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }



    //is end of the list. syntax --> isend [DSName] [PTR Name] Disclaimer: Ptr must have had to be made by calling ptrstart.
    else if (word1 == "isend") {          if (DS.count(word2) == 0) {              cout <<
"Error: The key " << word2 << " is not found in the map" << endl;
```

```
            continue;
        }
try {
            if (DS[word2]->gettype() == "dll") {                    if
(DS[word2]->DLLatEndOfList(DLLpointer[word3]))                    {
cout << "Pointer is at the end of the list!" << endl;
                }
```

```
                else cout << "Pointer is not at the end of the list!" << endl;
                ;
            }
            else if (DS[word2]->gettype() == "sll") {                    if
(DS[word2]->SLLatEndOfList(SLLpointer[word3]))                    {
cout << "Pointer is at the end of the list!" << endl;
                }
                cout << "Pointer is not at the end of the list!" << endl;
            }
else {
                cout << "This datastructure does not support the method called" << endl;
            };
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }




    else if (word1 == "ishead") {
if (DS.count(word2) == 0) {



        }
```

```
            cout << "Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        try {
            if (DS[word2]->gettype() == "dll") {            if (DS[word2]-
>DLLatHeadOfList(DLLpointer[word3])) {                      cout <<
"Pointer is at the head of the list!" << endl;
                DLLisend = true;
            }

            else cout << "Pointer is not at the head of the list!" << endl; DLLishead = false;
            ;
          }
            else if (DS[word2]->gettype() == "sll") {               if
(DS[word2]->SLLatHeadOfList(SLLpointer[word3])) {
cout << "Pointer is at the head of the list!" << endl;
            }
            cout << "Pointer is not at the head of the list!" << endl;
          }
else {
            cout << "This datastructure does not support the method called" << endl;
          };
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
catch (...) {

        }
```

```
            cout << "An unknown error occurred" << endl;
        }
    }


    //-------------------------------------Methods for Queue & Stack:


    else if (word1 == "push") {          if (DS.count(word2) == 0) {              cout <<
"Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        }
try {
            num = stod(word3);              if
(DS[word2]->gettype() == "queue") {
                DS[word2]->Queuepush(num);
            }
            else if (DS[word2]->gettype() == "stack") {
                DS[word2]->Stackpush(num);
            }
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
        catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }
```

```
      else if (word1 == "pop") {
if (DS.count(word2) == 0) {
            cout << "Error: The key " << word2 << " is not found in the map" << endl;
            continue;

          try {
            if (DS[word2]->gettype() == "queue") {
               DS[word2]->Queuepop();
            }
            else if (DS[word2]->gettype() == "stack") {
               DS[word2]->Stackpop();
            }
          }
          catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
          }
catch (...) {
            cout << "An unknown error occurred" << endl;
          }
       }

      else if (word1 == "front") {           if (DS.count(word2) == 0) {               cout <<
"Error: The key " << word2 << " is not found in the map" << endl;
            continue;
          }
try {
            cout << DS[word2]->Queuefront() << endl;


          }
```

```
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
        catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }

    else if (word1 == "back") {          if (DS.count(word2) == 0) {              cout <<
"Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        }
try {
            cout << DS[word2]->Queueback() << endl;
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }



    else if (word1 == "top") {          if (DS.count(word2) == 0) {              cout <<
"Error: The key " << word2 << " is not found in the map" << endl;
```

```
            continue;
        }
        try {
            cout << DS[word2]->Stacktop() << endl;
        catch (const out_of_range& e) {
            cout << "Error: " << e.what() << endl;
        }
        catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }


    else if (word1 == "isfull") {          if (DS.count(word2) == 0) {              cout <<
"Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        }
try {
            if (DS[word2]->gettype() == "queue") {
if (DS[word2]->QueueisFull()) {
cout << "Queue is full!" << endl;
                }
                cout << "Queue is not full!" << endl;
            }



        }
```

```
            if (DS[word2]->gettype() == "stack") {
if (DS[word2]->StackisFull()) {                    cout
<< "Stack is full!" << endl;
                }
                else cout << "Stack is not full!" << endl;
              }
          }
```

```
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
        catch (...) {
            cout << "An unknown error occurred" << endl;
        }
      }
```

```
    else if (word1 == "isempty") {          if (DS.count(word2) == 0) {              cout
<< "Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        }
try {
            if (DS[word2]->gettype() == "queue") {
if (DS[word2]->QueueisEmpty()) {
cout << "Queue is empty!" << endl;
                }
                else cout << "Queue is not empty!" << endl;
            }
            if (DS[word2]->gettype() == "stack") {
if (DS[word2]->StackisEmpty()) {                cout
<< "Stack is empty!" << endl;
                }
                else cout << "Stack is not empty!" << endl;
            }
        }
```

```
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }



    //--------------------------------------Methods for BST


    else if (word1 == "insert") {          if (DS.count(word2) == 0) {              cout
<< "Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        }
try {
            num = stod(word3);
            DS[word2]->BSTaddItem(num);
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
        catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }
```

```
    //deleteitem from BST        else if (word1 == "deleteitem") {           if
(DS.count(word2) == 0) {               cout << "Error: The key " << word2 << " is not
found in the map" << endl;
            continue;
        }
try {
            num = stod(word3);
            DS[word2]->BSTdeleteItem(num);
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
        catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }


    else if (word1 == "root") {           if (DS.count(word2) == 0) {               cout <<
"Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        }
try {
            cout << DS[word2]->BSTgetRoot()->data << endl; //returns BSTNode pointer and uses that
pointer to retrieve the attribute data(which is the value stored in the node)
        }
        catch    (const    out_of_range&    e)    {
cout  <<  "Error:  "  <<  e.what()  <<  endl;  }
catch (...) {
            cout << "An unknown error occurred" << endl;
```

```
        }
    }

    else if (word1 == "postorder") {        if (DS.count(word2) == 0) {
cout << "Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        }
try {
            DS[word2]->BSTpostorder(DS[word2]->BSTgetRoot());
            cout << endl;
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }

    else if (word1 == "preorder") {        if (DS.count(word2) == 0) {
cout << "Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        }
try {
DS[word2]
-
>BSTpreor
der(DS[wo
rd2]-
```

```
>BSTgetRo
ot());
            cout << endl;
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }

    else if (word1 == "inorder") {          if (DS.count(word2) == 0) {              cout
<< "Error: The key " << word2 << " is not found in the map" << endl;
            continue;
        }
try {
            DS[word2]->BSTinorder(DS[word2]->BSTgetRoot());
            cout << endl;
        }
        catch (const out_of_range& e) {
cout << "Error: " << e.what() << endl;
        }
catch (...) {
            cout << "An unknown error occurred" << endl;
        }
    }
  }
```

```
}
```

BST.cpp

```
#include "BST.h"
//----------------------------BST:
//Constructor
BST::BST() { root = nullptr; Dstype = "bst"; }

string BST::gettype() {
return Dstype;
};

//returns the node.
BSTNode* BST::BSTgetRoot() { return root; }

//used internally by the BSTaddItem method
BSTNode* BST::insert(BSTNode* node, int value) {
if (node == nullptr) {         return new
BSTNode(value);
    }
    if (value < node->data) {
        node->left = insert(node->left, value);
    }
else {
        node->right = insert(node->right, value);
    }     return
node;
}

//add value to the BST void
BST::BSTaddItem(int value) {
root = insert(root, value);
}



//only used internally by the delete item method
BSTNode* BST::deleteNode(BSTNode* node, int value) {
if (node == nullptr) {         return node;     }
    if (value < node->data) {
        node->left = deleteNode(node->left, value);
```

```
    }
    else if (value > node->data) {          node-
>right = deleteNode(node->right, value);      }
else {
        if (node->left == nullptr) {
BSTNode* temp = node->right;
delete node;              return temp;
        }
        else if (node->right == nullptr) {
BSTNode* temp = node->left;
delete node;              return temp;
        }
        BSTNode* temp = minValueNode(node->right);
node->data = temp->data;
        node->right = deleteNode(node->right, temp->data);
    }     return
node;
}

//delete an item
void BST::BSTdeleteItem(int value) {
root = deleteNode(root, value);
}

//only used internally
BSTNode* BST::minValueNode(BSTNode* node) {
BSTNode* current = node;     while
(current->left != nullptr) {
current = current->left;
    }     return
current;
}
void BST::BSTinorder(BSTNode* node) {
if (node != nullptr) {
BSTinorder(node->left);          cout
<< node->data << " ";
        BSTinorder(node->right);
    }
}
void BST::BSTpostorder(BSTNode* node) {
if (node != nullptr) {
BSTpostorder(node->left);
BSTpostorder(node->right);          cout
<< node->data << " ";
    }
}
void BST::BSTpreorder(BSTNode* node) {
if (node != nullptr) {          cout <<
node->data << " ";
BSTpreorder(node->left);
        BSTpreorder(node->right);
    }
}  void
BST::print() {
BSTinorder(root);
```

```
}

void BST::sort_ascend() {
BSTinorder(root);     cout
<< endl;
}
void BST::sort_descend() {
BSTpostorder(root);     cout
<< endl;
}



BST::BST(const BST& other) {
root = nullptr;     Dstype =
other.Dstype;     root =
copyTree(other.root);
}

//helper method. Used by the copy constructor
BSTNode* BST::copyTree(BSTNode* node) {
if (node == nullptr) {         return
nullptr;
    }
    BSTNode* newNode = new BSTNode(node->data);
newNode->left = copyTree(node->left);     newNode-
>right = copyTree(node->right);     return
newNode;
}







//returns true if found, otherwise returns false.
 //It iterates through the tree using a while loop and moves to the left or right --
 //child based on the comparison of the search key with the current node's value.
bool BST::search(int value) {     BSTNode* current = root;     while (current !=
nullptr) {         if (current->data == value) {             return true;
        }
        if (current->data < value) {
current = current->right;
        }
else {
            current = current->left;
        }
}
    return false;
}

BST ::  ~BST() {
    root = deleteTree(root);
}
```

```
//used internally only to delete tree.
BSTNode* BST::deleteTree(BSTNode* node) {
if (node == nullptr) {         return
nullptr;
    }
    node->left = deleteTree(node->left);
node->right = deleteTree(node->right);
delete node;     return nullptr;
}
```

BST.h

```
#pragma once
#include "Datastructure.h"
#include "BSTNode.h" class BST
:public Datastructure { private:
BSTNode* root;     string Dstype;
public:     BST();
    string gettype();

    BSTNode* BSTgetRoot();

    BSTNode* insert(BSTNode* node, int value);

    void BSTaddItem(int value);

    //only used internally by the delete item method
    BSTNode* deleteNode(BSTNode* node, int value);

    void BSTdeleteItem(int value);

    BSTNode* minValueNode(BSTNode* node);

    void BSTinorder(BSTNode* node);

    void BSTpostorder(BSTNode* node);

    void BSTpreorder(BSTNode* node);

    void print();

    void sort_ascend();
void sort_descend();


    BST(const BST& other);
```

```
BSTNode* copyTree(BSTNode* node);
```

```
    bool search(int value);

    ~BST();
    BSTNode* deleteTree(BSTNode* node);

};
```

BSTNode.h

```
#pragma once
class BSTNode { //can either be a struct inside of the class BST or a separate
class. This way is more reliable. public:
    int data;
BSTNode* left;
    BSTNode* right;
BSTNode(int value) {
data = value;         left
= nullptr;         right =
nullptr;
    } };
```

Datastructure.h

```
#pragma once
#include "BSTNode.h"
#include "DLLNode.h"
#include "SLLNode.h"
#include <iostream>
#include <string>
#include <algorithm>
#include <vector> using
namespace std;

class Datastructure { public:
    //BST Methods;
    virtual BSTNode* BSTgetRoot() {
        cout << "This datastructure does not support the method called" << endl;
return nullptr;
    };
    virtual void BSTdeleteItem(int value) {
        cout << "This datastructure does not support the method called" << endl;
                                                                        << endl;
```

```
    };
    virtual void BSTaddItem(int value) {
        cout << "This datastructure does not support the method called"
    };
    virtual void BSTinorder(BSTNode* node) {
        cout << "This datastructure does not support the method called" << endl;
    };
    virtual void BSTpostorder(BSTNode* node) {
        cout << "This datastructure does not support the method called" << endl;
    };
    virtual void BSTpreorder(BSTNode* node) {
        cout << "This datastructure does not support the method called" << endl;
};
    // ALl methods have;
virtual void print() = 0;

    virtual void sort_ascend() {
    };
    virtual void sort_descend() {
    };
    virtual bool search(int value) {
return true;;
    };


    virtual string gettype() = 0;

    //DLL Methods;
    virtual void DLLaddNodeToEnd(int data) {
        cout << "This datastructure does not support the method called" << endl;
    };
    virtual void DLLdeleteNodeFromEnd() {
        cout << "This datastructure does not support the method called" << endl;
    };
    virtual void DLLaddNodeToFront(int data) {
        cout << "This datastructure does not support the method called" << endl;
    };
    virtual void DLLdeleteNodeFromFront() {
        cout << "This datastructure does not support the method called" << endl;
    };
    virtual void DLLinsertAfter(int data, int value) {
        cout << "This datastructure does not support the method called" << endl;
    };
    virtual DLLNode* DLLpointToBeginning() {
        cout << "This datastructure does not support the method called" << endl;
return nullptr;
    };
    virtual DLLNode* DLLpointToEnd() {
        cout << "This datastructure does not support the method called"
return nullptr;
                                                                    << endl;
```

```
    };

    virtual int DLLgetValueAtPointer(DLLNode* pointer) {
        cout << "This datastructure does not support the method called" << endl;
return 0;
    };
    virtual DLLNode* DLLmovePointerForward(DLLNode* pointer) {
        cout << "This datastructure does not support the method called" << endl;
return nullptr;
    };
    virtual DLLNode* DLLmovePointerBackward(DLLNode* pointer) {
        cout << "This datastructure does not support the method called" << endl;
return nullptr;
    };
    virtual bool DLLatEndOfList(DLLNode* pointer) {
        cout << "This datastructure does not support the method called" << endl;
return true;
    };
    virtual bool DLLatHeadOfList(DLLNode* pointer) {
        cout << "This datastructure does not support the method called" << endl;
return true;
    };

    //SLL Methods;

    virtual void SLLaddNodeToEnd(int data) {
        cout << "This datastructure does not support the method called" << endl;
    };
    virtual  void SLLdeleteNodeFromEnd() {
        cout << "This datastructure does not support the method called" << endl;
    };
    virtual void SLLaddNodeToFront(int data) {
        cout << "This datastructure does not support the method called" << endl;
    };
    virtual void SLLdeleteNodeFromFront() {
        cout << "This datastructure does not support the method called" << endl;
    };
    virtual void SLLinsertAfter(int data, int value) {
        cout << "This datastructure does not support the method called" << endl;
    };
    virtual SLLNode* SLLpointToBeginning() {
        cout << "This datastructure does not support the method called" << endl;
return nullptr;
    };
    virtual SLLNode* SLLpointToEnd() {
        cout << "This datastructure does not support the method called"
return nullptr;
    };


                                                                    << endl;
```

```
    virtual int SLLgetValueAtPointer(SLLNode* pointer) {
        cout << "This datastructure does not support the method called" << endl;
return 0;
    };
    virtual SLLNode* SLLmovePointerForward(SLLNode* pointer) {
        cout << "This datastructure does not support the method called" << endl;
return nullptr;
    };
    virtual bool SLLatEndOfList(SLLNode* pointer) {
        cout << "This datastructure does not support the method called" << endl;
return 0;
    };

    virtual bool SLLatHeadOfList(SLLNode* pointer) {
        cout << "This datastructure does not support the method called" << endl;
return 0;
    };

    //stack methods;

    virtual void Stackpush(int item) {
        cout << "This datastructure does not support the method called" << endl;
    };
    virtual  void Stackpop() {
        cout << "This datastructure does not support the method called" << endl;
    };
    virtual int Stacktop() {
        cout << "This datastructure does not support the method called" << endl;
return 0;
    };
    virtual  bool StackisFull() {
        cout << "This datastructure does not support the method called" << endl;
return false;
    };
    virtual bool StackisEmpty() {
        cout << "This datastructure does not support the method called" << endl;
return false;
    };



    //Queue methods;

    virtual void Queuepush(int value) {
        cout << "This datastructure does not support the method called" << endl;

    };
    virtual void Queuepop() {
        cout << "This datastructure does not support the method called" << endl;
    };
```

```
    virtual int Queuefront() {
        cout << "This datastructure does not support the method called" << endl;
return 0;
    };
    virtual int Queueback() {
        cout << "This datastructure does not support the method called" << endl;
return 0;
    };
    virtual  bool QueueisFull() {
        cout << "This datastructure does not support the method called" << endl;
return 0;
    };
    virtual bool QueueisEmpty() {
        cout << "This datastructure does not support the method called" << endl;
return 0;
    };

    //Variable methods;

    virtual void VarAssign(int value) {
        cout << "This datastructure does not support the method called" << endl;
    };
    virtual int Vargetvalue() {
        cout << "This datastructure does not support the method called" << endl;
return 0;
    };
    virtual void VarIfTrue(bool condition, string message) {
        cout << "This datastructure does not support the method called" << endl;
    };
    virtual void VarIfFalse(bool condition, string message) {
        cout << "This datastructure does not support the method called" << endl;
    };

};
```

Doublelinkedlist.cpp

```
#include "DoubleLinkedList.h"
//----------------------------DLL:
//constructor
DoubleLinkedList::DoubleLinkedList() {
head = nullptr;     tail = nullptr;
current = nullptr;     count = 0;
Dstype = "dll";
}

//copy constructor
DoubleLinkedList::DoubleLinkedList(const DoubleLinkedList& other) {
```

```
    head = nullptr;     tail =
nullptr;     current = nullptr;
count = 0;     DLLNode* temp =
other.head;     while (temp !=
nullptr) {
DLLaddNodeToEnd(temp->data);
temp = temp->next;
    }
}
 //returns DS type string
DoubleLinkedList::gettype() {
return Dstype;
};

DoubleLinkedList :: ~DoubleLinkedList() {
DLLNode* temp = head;     while (temp !=
nullptr) {         DLLNode* next = temp-
>next;         delete temp;         temp
= next;
    }
}
void DoubleLinkedList::print() {
DLLNode* temp = head;     while
(temp != nullptr) {         cout
<< temp->data << " ";         temp
= temp->next;
    }
}
bool DoubleLinkedList::search(int value) {
DLLNode* temp = head;     while (temp !=
nullptr) {         if (temp->data ==
value) {             return true;
}         temp = temp->next;
    }     return
false;
}
void DoubleLinkedList::sort_ascend() {
DLLNode* temp = head;     while (temp !=
nullptr) {         DLLNode* next = temp-
>next;         while (next != nullptr) {
if (temp->data > next->data) {
int tempData = temp->data;
temp->data = next->data;
next->data = tempData;
            }             next
= next->next;
        }         temp =
temp->next;
    }
}
void DoubleLinkedList::sort_descend() {
DLLNode* temp = head;     while (temp !=
nullptr) {         DLLNode* next = temp-
>next;         while (next != nullptr) {
if (temp->data < next->data) {
```

```
int tempData = temp->data;
temp->data = next->data;
next->data = tempData;
            }
            next = next->next;
        }
        temp = temp->next;
    }
}


//add node to the end
void DoubleLinkedList::DLLaddNodeToEnd(int data) {
DLLNode* newNode = new DLLNode(data);     if (head
== nullptr) {         head = newNode;         tail
= newNode;
    }     else {
tail->next = newNode;
newNode->prev = tail;
tail = newNode;
    }
count++;
}


//delete node from the end void
DoubleLinkedList::DLLdeleteNodeFromEnd() {
if (head == nullptr) {         return;     }
if (head == tail) {         delete tail;
head = nullptr;         tail = nullptr;
current = nullptr;
    }
else {
        DLLNode* temp = tail->prev;
temp->next = nullptr;
delete tail;         tail = temp;
}     count--;
}

//add node to the front
void DoubleLinkedList::DLLaddNodeToFront(int data) {
    DLLNode* newNode = new DLLNode(data);
    if (head == nullptr) {
head = newNode;         tail
= newNode;
    }
else {
        newNode->next = head;
head->prev = newNode;         head
= newNode;
    }
count++;
```

```
}

//delete node from the front of the list void
DoubleLinkedList::DLLdeleteNodeFromFront() {
if (head == nullptr) {         return;     }
    if (head == tail) {
delete head;         head
= nullptr;         tail =
nullptr;
    }
else {
        DLLNode* temp = head->next;
delete head;         head = temp;
head->prev = nullptr;
    }
count--;
}

//Insert value after a node value. void
DoubleLinkedList::DLLinsertAfter(int data, int value) {
DLLNode* temp = head;     while (temp != nullptr) {
if (temp->data == value) {
            DLLNode* newNode = new DLLNode(data);
newNode->prev = temp;             newNode->next =
temp->next;             temp->next = newNode;
if (temp == tail) {                 tail =
newNode;
            }
else {
                newNode->next->prev = newNode;
            }
count++;
return;         }
        temp = temp->next;
    }
}

////point a pointer to the beginning. (A new pointer will be created whenever this
is callled. the pointer name will be decided on the user's PTRName input)
DLLNode* DoubleLinkedList::DLLpointToBeginning() {
current = new DLLNode(0); current = head;
    return current;
}


//point a pointer to the end. (A new pointer will be created whenever this is
callled. the pointer name will be decided on the user's PTRName input)
DLLNode* DoubleLinkedList::DLLpointToEnd() {
current = new DLLNode(0);     current =
tail;     return current;
}

//get the value at pointer and print it out
```

```
int DoubleLinkedList::DLLgetValueAtPointer(DLLNode* pointer) {
return pointer->data;
}
//move a pointer forward
DLLNode* DoubleLinkedList::DLLmovePointerForward(DLLNode* pointer) {
if (pointer->next != nullptr) {         pointer = pointer->next;
return pointer;
    }
    return nullptr;
}

//move a pointer backward
DLLNode* DoubleLinkedList::DLLmovePointerBackward(DLLNode* pointer) {
if (pointer->prev != nullptr) {         pointer = pointer->prev;
return pointer;
    }
    return nullptr;
}

//Checks if pointer is at the end of the list bool
DoubleLinkedList::DLLatEndOfList(DLLNode* pointer) {
if (pointer == tail) {         return true;
    }
    return false;
}

//Checks if pointer is at the head of the list bool
DoubleLinkedList::DLLatHeadOfList(DLLNode* pointer) {
if (pointer == head) {         return true;
    }
    return false;
}
```

Doublelinkedlist.h

```
#pragma once
#include "Datastructure.h" #include
"DLLNode.h"
class DoubleLinkedList : public Datastructure {
private:
DLLNode* head;
    DLLNode* tail;
DLLNode* current;
int count;     string
Dstype; public:
    //copy constructor
    DoubleLinkedList(const DoubleLinkedList& other);

    DoubleLinkedList();
```

```
    string gettype();

    ~DoubleLinkedList();

    void print();

    bool search(int value);

    void sort_ascend();

    void sort_descend();

    void DLLaddNodeToEnd(int data);

    void DLLdeleteNodeFromEnd();

    void DLLaddNodeToFront(int data);

    void DLLdeleteNodeFromFront();

    void DLLinsertAfter(int data, int value);

    DLLNode* DLLpointToBeginning();

    DLLNode* DLLpointToEnd();

    int DLLgetValueAtPointer(DLLNode* pointer);

    DLLNode* DLLmovePointerForward(DLLNode* pointer);

    DLLNode* DLLmovePointerBackward(DLLNode* pointer);

    bool DLLatEndOfList(DLLNode* pointer);

    bool DLLatHeadOfList(DLLNode* pointer);
};
```

DLLNode.h

```
#pragma once class
DLLNode { public:
int data;
DLLNode* next;
DLLNode* prev;
DLLNode(int data)
{         this-
>data = data;
this->next =
```

```
nullptr;
this->prev =
nullptr;
    } };
```

Queue.cpp

```
#include "Queue.h"
//----------------------------queue;
//Constructor
Queue::Queue() {
    Dstype = "queue";
};  string
Queue::gettype() {
return Dstype;
};


//copy constructor Queue ::
Queue(const Queue& other) {     data
= vector<int>(other.data);
}

 // print queue void Queue::print() {
for (int i = 0; i < data.size(); i++) {
cout << data[i] << " ";
    }
}
 // search queue bool Queue::search(int
value) {     for (int i = 0; i <
data.size(); i++) {         if (data[i] ==
value) {             return true;
        }     }
return false;
}

// sort queue in ascending void
Queue::sort_ascend() {
sort(data.begin(), data.end());
}
// sort queue in ascending void
Queue::sort_descend() {
sort(data.rbegin(), data.rend());
}

// push (add item) void
Queue::Queuepush(int item) {
data.push_back(item);
```

```
}

// pop (pop item) void
Queue::Queuepop() {     if
(!data.empty()) {
data.erase(data.begin());
    }
}

// front (return front of queue)
int Queue::Queuefront() {     if
(!data.empty()) {         return
data.front();
    }
return -1;
}

// back (return back of queue)
int Queue::Queueback() {
if (!data.empty()) {
return data.back();
    }
return -1;
}

// isFull (return a bool value) bool
Queue::QueueisFull() {
    return false; // queues can dynamically resize, so it can never be full
}

// isEmpty (return a bool value)
bool Queue::QueueisEmpty() {
return data.empty(); }
```

Queue.h

```
#pragma once
#include "Datastructure.h" class
Queue : public Datastructure {
public:
    Queue(); // default constructor

    string gettype();

    // copy constructor
    Queue(const Queue& other);

    // assignment operator
    Queue& operator=(const Queue& other) {
data = vector<int>(other.data);         return
*this;
```

```
    }


    // destructor
~Queue() {
data.clear();     }     //
print queue     void
print();      // search
queue     bool search(int
value);

    // sort queue in ascending
void sort_ascend();     //
sort queue in ascending
void sort_descend();
    // push (add item)
void Queuepush(int value);
    // pop (pop item)
void Queuepop();

    // front (return front of queue)
int Queuefront();

    // back (return back of queue)
int Queueback();

    // isFull (return a bool value)
bool QueueisFull();

    // isEmpty (return a bool value)
bool QueueisEmpty();
 private:
    vector<int> data;
string Dstype;
};
```

Singlylinkedlist.cpp

```
#include "SinglyLinkedList.h"
//----------------------------SLL:

//constructor
SinglyLinkedList::SinglyLinkedList() {
head = nullptr;     current = nullptr;
count = 0;     Dstype = "sll";
```

```
}

//copy constructor
SinglyLinkedList::SinglyLinkedList(const SinglyLinkedList& other) {
head = nullptr;     current = nullptr;     count = 0;
    SLLNode* temp = other.head;
while (temp != nullptr) {
SLLaddNodeToEnd(temp->data);
temp = temp->next;
    }
}



//returns the type of DS

string SinglyLinkedList::gettype() {
return Dstype;
};



//destructor
SinglyLinkedList:: ~SinglyLinkedList() {
SLLNode* temp = head;     while (temp !=
nullptr) {         SLLNode* next = temp-
>next;         delete temp;         temp
= next;
    }
}
void SinglyLinkedList::print() {
SLLNode* temp = head;     while
(temp != nullptr) {         cout
<< temp->data << " ";         temp
= temp->next;
    }
}
bool SinglyLinkedList::search(int value) {
SLLNode* temp = head;     while (temp !=
nullptr) {         if (temp->data ==
value) {             return true;
        }
        temp = temp->next;
    }     return
false;
}
void SinglyLinkedList::sort_ascend() {
SLLNode* temp = head;     while (temp
!= nullptr)
    {
        for (SLLNode* i = temp->next; i != nullptr; i = i->next) {
            if (temp->data > i->data) {
int tempData = temp->data;
temp->data = i->data;                 i-
>data = tempData;
```

```
            }
}
        temp = temp->next;
    }
}
void SinglyLinkedList::sort_descend() {
SLLNode* temp = head;     while (temp
!= nullptr)
    {
        for (SLLNode* i = temp->next; i != nullptr; i = i->next) {
            if (temp->data < i->data) {
int tempData = temp->data;
temp->data = i->data;                 i-
>data = tempData;
            }
}
        temp = temp->next;
    }
}

//Add a node from the end
void SinglyLinkedList::SLLaddNodeToEnd(int data) {
SLLNode* newNode = new SLLNode(data);     if (head
== nullptr) {         head = newNode;
    }
else {
        SLLNode* temp = head;
while (temp->next != nullptr) {
temp = temp->next;
        }
        temp->next = newNode;
    }
count++;
}
//Delete a node from the end
void SinglyLinkedList::SLLdeleteNodeFromEnd() {
if (head == nullptr) {         return;     }
    if (head->next == nullptr) {
delete head;         head =
nullptr;         current =
nullptr;
    }
else {
        SLLNode* temp = head;         while
(temp->next->next != nullptr) {
temp = temp->next;
        }         delete
temp->next;         temp-
>next = nullptr;
    }
count--;
}
//Add a node from the front
```

```
void SinglyLinkedList::SLLaddNodeToFront(int data) {
SLLNode* newNode = new SLLNode(data);     newNode-
>next = head;     head = newNode;     count++;
}
//Delete a node from the front
void SinglyLinkedList::SLLdeleteNodeFromFront() {
if (head == nullptr) {         return;
    }
    SLLNode* temp = head;
head = head->next;
delete temp;     count--;
}
//insert a value after a node value
void SinglyLinkedList::SLLinsertAfter(int data, int value) {
SLLNode* temp = head;     while (temp != nullptr) {
if (temp->data == value) {
            SLLNode* newNode = new SLLNode(data);
newNode->next = temp->next;             temp-
>next = newNode;             count++;
return;         }
        temp = temp->next;
    }
}
//Point a poiter to the beginning of the list
SLLNode* SinglyLinkedList::SLLpointToBeginning() {
current = new SLLNode(0);     current = head;
return current;
}
//Point a poiter to the end of the list
SLLNode* SinglyLinkedList::SLLpointToEnd() {
current = new SLLNode(0);     current =
head;
    while (current->next != nullptr) {
current = current->next;
    }
    return current;
}
//getting value at pointer
int SinglyLinkedList::SLLgetValueAtPointer(SLLNode* pointer) {
return pointer->data;
}

//moving pointer forward
SLLNode* SinglyLinkedList::SLLmovePointerForward(SLLNode* pointer) {
if (pointer->next != nullptr) {         pointer = pointer->next;
return pointer;
    }
    return nullptr;
}

//Check if pointer is at the end of the list bool
SinglyLinkedList::SLLatEndOfList(SLLNode* pointer) {
if (pointer->next == nullptr) {         return true;
    }     else {
return false;
```

```
    }
}
//Check if pointer is at the head of the list bool
SinglyLinkedList::SLLatHeadOfList(SLLNode* pointer) {
if (pointer == head) {         return true;
    }     else {
return false;
    } }
```

Singlylinkedlist.h

```
#pragma once
#include "Datastructure.h" #include
"SLLNode.h"
class SinglyLinkedList : public Datastructure {
private:     SLLNode* head;     SLLNode*
current;     int count;     string Dstype;
public:
    string gettype();

    SinglyLinkedList(const SinglyLinkedList& other);

    SinglyLinkedList();

    ~SinglyLinkedList();

    void print();

    bool search(int value);

    void sort_ascend();

    void sort_descend();

    void SLLaddNodeToEnd(int data);

    void SLLdeleteNodeFromEnd();

    void SLLaddNodeToFront(int data);

    void SLLdeleteNodeFromFront();

    void SLLinsertAfter(int data, int value);

    SLLNode* SLLpointToBeginning();

    SLLNode* SLLpointToEnd();
```

```
    int SLLgetValueAtPointer(SLLNode* pointer);

    SLLNode* SLLmovePointerForward(SLLNode* pointer);

    bool SLLatEndOfList(SLLNode* pointer);

    bool SLLatHeadOfList(SLLNode* pointer);
};
```

SLLNode.h

```
#pragma once class SLLNode {
public:     int data;
SLLNode* next;
SLLNode(int data) {
this->data = data;
this->next = nullptr;
    } };
```

Stack.cpp

```
#include "Stack.h"
#include <iostream> using
namespace std;
//constructor
Stack::Stack() {
    Dstype = "stack";
};
string Stack::gettype() {
return Dstype;
};

//copy constructor
Stack::Stack(const Stack& other) {     for (int
i = 0; i < other.arr.size(); i++) {
arr.push_back(other.arr[i]);
    }
}


// Destructor Stack
:: ~Stack() {
arr.clear();
}

// Push item
void Stack::Stackpush(int item) {
arr.push_back(item);
```

```
}

// Pop item void
Stack::Stackpop() {
if (!arr.empty()) {
arr.pop_back();
    }
}

// Display top of the stack
int Stack::Stacktop() {
if (!arr.empty()) {
return arr.back();
    }
return -1;
}

// Check if stack is full bool
Stack::StackisFull() {
    return false; // vector can dynamically resize, so it can never be full
}

// Check if stack is empty bool
Stack::StackisEmpty() {
return arr.empty();
}

// Print stack void
Stack::print() {
    for (int i = 0; i < arr.size(); i++) {
cout << arr[i] << " ";
    }
}

// Search stack
bool Stack::search(int value) {     for
(int i = 0; i < arr.size(); i++) {
if (arr[i] == value) {             return
true;
        }
}
    return false;
}

// Sort stack in ascending order void
Stack::sort_ascend() {
sort(arr.begin(), arr.end());
}
```

```
// Sort stack in descending order
void Stack::sort_descend() {
sort(arr.rbegin(), arr.rend()); }
```

Stack.h

```
#pragma once #include
"Datastructure.h" class Stack
:public Datastructure { private:
    vector<int> arr;
string Dstype;
public:     //
Constructor
    Stack();
string gettype();
    // copy constructor
    Stack(const Stack& other);

    // Destructor
    ~Stack();
     // Push item     void
Stackpush(int item);     //
Pop item     void Stackpop();

    // Display top of the stack
int Stacktop();

    // Check if stack is full
bool StackisFull();

    // Check if stack is empty
bool StackisEmpty();      //
Print stack     void print();
// Search stack     bool
search(int value);

    // Sort stack in ascending order
void sort_ascend();

    // Sort stack in descending order
void sort_descend();
};
```

Variable.cpp

```
#include "Variable.h"
//--------------Variable class (vector)
//constructor

Variable::Variable() {
    { value.push_back(0); }
```

```
    Dstype = "var";
};
string Variable::gettype() {
return Dstype;
};

//copy constructor
Variable :: Variable(const Variable& other) {
value.push_back(other.value[0]);
}

// if boolean variable true print message void
Variable::VarIfTrue(bool condition, string message) {
if (condition) {         cout << message << endl;
    }
}

// if boolean variable not true print message void
Variable::VarIfFalse(bool condition, string message) {
if (!condition) {         cout << message << endl;
    }
}

int Variable::Vargetvalue() {
return value[0]; };
```

Variable.h

```
#pragma once
#include "Datastructure.h" class
Variable :public Datastructure {
private:
    vector<int> value;
string Dstype; public:
// constructor
    Variable();
    // destructor
    ~Variable() { value.clear(); }
    // copy constructor
    Variable(const Variable& other);

    string gettype();


    // declare variable
    void VarAssign(int val) { value[0] = val; }
```

```
    // operator overloading for assigning one variable to another      Variable&
operator=(const Variable& other) { value[0] = other.value[0]; return *this; }
    // operator overloading for adding value to variable
    Variable& operator+(int val) { value[0] += val; return *this; }
    // operator overloading for adding one variable to another
    Variable& operator+(const Variable& other) { value[0] += other.value[0]; return
*this; }
    // operator overloading for subtracting value from variable
    Variable& operator-(int val) { value[0] -= val; return *this; }
    // operator overloading for subtracting one variable from another
    Variable& operator-(const Variable& other) { value[0] -= other.value[0]; return
*this; }
    // operator overloading for multiplying variable by value
    Variable& operator*(int val) { value[0] *= val; return *this; }
    // operator overloading for multiplying one variable by another
    Variable& operator*(const Variable& other) { value[0] *= other.value[0]; return
*this; }
    // operator overloading for dividing variable by value
    Variable& operator/(int val) { value[0] /= val; return *this; }
    // operator overloading for dividing one variable by another
    Variable& operator/(const Variable& other) { value[0] /= other.value[0]; return
*this; }
    // operator overloading for mod variable by value
    Variable& operator%(int val) { value[0] %= val; return *this; }
    // operator overloading for mod variable by another variable
    Variable& operator%(const Variable& other) { value[0] %= other.value[0]; return
*this; }
    // print variable
    void print() { cout << value[0]; }

    int Vargetvalue();

    // if boolean variable true print message
void VarIfTrue(bool condition, string message);
// if boolean variable not true print message
void VarIfFalse(bool condition, string message);


};
```

Input.txt

Var bst b