Assignment2

Sorin Macaluso

October 2023

1 Note

All code with comments is at the bottom in the appendix. The cout's in the original code had to be commented out or turned into a descriptive comment to explain what happened. All double quotes or single quotes where taken off and I added a comment with the data type next to it.

2 Main

Main is very similar to the main files of the past two labs. There is the same file read in section to read in the 666 lines of the magicitems.txt file into a array to be passed around to be manipulated and changed as see fit by me. This time we are passing the array into the Binary Search Tree (BST) functions. For this section we are making the magicitems.txt file into a BST as well as the graph1.txt file into three different kinds of graphs. The graph representations are Matrix, Adjacency List, and Linked Objects (LinkedObJ), these will be explained in later sections of this report. The main file is a bit different from the other labs, because there has been a lot more output and logic built into the main file to check and print out all the data to explain these representations to whom ever the reader is.

2.1 Read File BST

```
    cout << \n;

    int count = 0;

    int magicIteamCount = 0;
    string magicIteam[arraysize];

    string readInString;

    ifstream File (magicitems.txt);

    if (File.is_open()){

        while (File.good()){

            getline(File, readInString);

            magicIteam[magicIteamCount] = readInString;
            magicIteamCount++;


        }
        File.close();
    }else{
        cout << Unable to open file;
    }
```

As shown above the code for reading in the data in the magicitems.txt file is the same in practice as the other reports, the only difference is the names of all the file and variable names.

2.2 Read File Graph

```
    int vertexs = 0;
    string start = 0;
    string end = 0;
    vector<int> StartList;
    vector<int> EndList;
    int Gcount = 0;

    for(string i : GraphVector){
        if(i.find(--) == std::string::npos){

            if(i.find(new) != std::string::npos || i == GraphVector.back()){

                MatrixGraph(vertexs, StartList, EndList);

                AdjacencyList(vertexs, StartList, EndList, Gcount);

                LinkedObjs(vertexs, StartList, EndList, Gcount);

                vertexs = 0;
                StartList.clear();
                EndList.clear();

            }else if(i.find(vertex) != std::string::npos){

                vertexs++;

            }else if(i.find(edge) != std::string::npos){

                std::istringstream iss(i);
                std::string token;

                bool addToVectorA = true;

                while (iss >> token) {
                    if (token == add || token == edge || token == -) {
                        continue;
                    }

                    int num;

                    if (std::istringstream(token) >> num) {
                        if (addToVectorA) {
                            StartList.push_back(num);
                        } else {
                            EndList.push_back(num);
                        }

                        addToVectorA = !addToVectorA;

                        start at 0 instead of 1
                        if(num == 0){
                            Gcount = 5;
                        }
                    }
                }
            }
        }
```

As you can already tell the file reading for the graph is much more complex than the file read for the BST. This is because we have to take much more data, and the data is formatted differently from the magicitems.txt file than the graph1.txt file. In the graph1.txt file there is comments, command for new graph, command for the addition of a vertex, and the command for the addition of edges. All of this information needs to be store and passed to the graph representations so that the functions are able to work properly.

From line 11 is the code above to the bottom of this code block is the logic that is needed in order to check for the different pieces of data that we will need in order to make the graph representations. The for loop goes through everything in the vector GraphVector and checks if the string at i contains 1 of the 4 following things; new, vertex, edge, or –. Base on what is found in the string a different action is performed. If vertex is found then 1 is added the the vertex amount, this total after all processing is done will show the total number of vertices for the graph. If edge is found a string stream is used to skip over the words that are not needed and look at the 2 numbers that will signify the start and end edge. Then those two different numbers are added to either StartList vector (if the start), or EndList vector (if the end). Also if there is a edge that is only 0 then we know that the graph starts at 0 and not 1. Fun Fact: I chose 5 as my counter number, just like how in floating point values 0 is + and 5 is -. If new is found then that means a new graph is going to be created. So we pass all the data to the different representations and then reset all the values that are used for data. Finally the code checks if a – is not seen, if – is found it is not checked and is skipped.

3 Binary Search Tree (BST): Creation

3.1 BST: Creation Code

```
struct BST{
    string data;
    BST* left = nullptr;
    BST* right = nullptr;
    BST* parent = nullptr;
};

string BSTTreeInsert(BST*& root, string value){
    BST* trailing = nullptr;
    BST* current = root;
    string path = NULL;

    BST* node = new BST;
    node->data = value;

    while(current != nullptr){
        trailing = current;
        if(node->data < current->data){
            current = current->left;
            path = path + L;
        }else{
            current = current->right;
            path = path + R;
        }
    }

    node->parent = trailing;
    if(trailing == nullptr){
        root = node;
    }else{
        if(node->data < trailing->data){
            trailing->left = node;
        }else{
            trailing->right = node;
        }
    }
     return path;
```

```
38 }
```

The code above shows how the BST is created. The root of the tree is made from the struct BST. This struct contains the actual data, a left, and a right, as well as a pointer to the parent, or the node that is directly before the node being inserted. Then on line 8 is the BSTTreeInsert function. This function takes in two parameters, one being the root of the tree, or the start, and the value that we are inserting into the BST. Then the program creates a new BST node and puts the data for that node as the value that was passed to the function. Then until we get to the bottom (a nullptr) we go left or right based on if the passed value is greater than or less than the current node that. Before we do that the trailing node is set to the current node so that we are always one step behind, this makes sense since the trailing node is suppose to keep track of the parent node (or the node that is one before the insert node). We also concatenate a string together called path that stores L for a left movement and R for a right movement. This will be used to show the path once the program has found where to insert the new item into.

Then once the path is found and the new node is inserted the function repeats for all 666 items in the magicitems.txt file. But before the function has another value inserted into the BST the node parent value is set to the trailing value. Then we have to decided again weather to go to the left or right from the trailing value and then insert the new node. Finally the path is returned to use for output in the main.cpp file.

The time complexity of this can be related to the current height of the BST or O(h). The new thing that is being inserted will have to travel to the bottom of the BST from the top (root). This means that every time something is inserted unless it is going to the empty right or empty left value of the parent node then the height is increased. Meaning that the insert will need to travel at worst the height of the BST.

4 Binary Search Tree (BST): Search & In Order Print

4.1 BST: Search Code

```
BST* BSTSearch(BST*& node, string key){

     if(node->data == key){
         cout << :- << node->data <<  is done searching. Comparisons =  << comparisons
    << \n;
         comparisons = 0;
         return node;
    }else if(key < node->data){
         comparisons++;
         cout << L;
         return BSTSearch(node->left, key);
    }else{
         comparisons++;
         cout << R;
         return BSTSearch(node->right, key);
    }
}
```

The action of searching in a BST is similar to how the insert method works. The program will start at the root node and go left or right depending on if the key value is lesser than (left) or greater than (right). Once the program decides to go left or right that pointer is passed back to the fucntion again. It will recursively call with the new pointer going deeper and deeper in the BST until the data is equal to the key, or the key does not match anything in the BST. Once it is found we return the node that it was found at. Along with this is the turn that the program took, either left or right, it is outputted by the program.

The time complexity of the search for the BST is 0(log(n)). This is because if it has to to search for the greatest value that the BST at each of the choices going left or right the BST would be eliminating half of the items that it would have to choose from. Making the time complex for a BST search similar to Binary search from the previous assignment. Though in a worst case scenario the time complexity would be 0(n). The worst case scenario is that the BST becomes unbalance, unbalanced to the fact that the BST is just one line going left or right. This can happen when the data inputted for the BST to be built off is already in a sorted order.

4.2 BST: In Order Print (IOP) Code

```
void InOrderPrint(BST* node){

    if(node != nullptr){
       if(node->left != nullptr){
            InOrderPrint(node->left);
        }

        cout << node->data << \n;

       if(node->right != nullptr){
            InOrderPrint(node->right);
        }

    }else{
        cout << The Tree is Empty! << \n;
    }


}
```

The InOrderPrint function is very simple in the that is does also start at the root of the tree. Then it goes to the most left item in the BST and start printing from there. This effectively make it so that the print out starts at the most left (or smallest value) and then prints out the right most value or the greatest value that the BST is currently holding. Or the BST is printted out in order from least to greatest.

The time complexity of this is 0(n). This is because the InOrderPrint function will go through everything in the BST in order and print out the data. So it will have to go through all 666 items in the BST in order to print everything out.

5 Graphs

5.1 Matrix

```
void MatrixGraph(int vertexs, vector<int> start, vector<int> end) {
    int Matrix[vertexs][vertexs];
    int VECTOR_SIZE = 0;

    if(start.size() == end.size()){
        VECTOR_SIZE = start.size();
    }else if(start.size() > end.size()){
        VECTOR_SIZE = start.size();
    }else{
        VECTOR_SIZE = end.size();
    }


    for(int i = 0; i < vertexs; i++){
        for(int j = 0; j < vertexs; j++){
            Matrix[i][j] = 0;
        }
    }

    for(int i = 0; i < VECTOR_SIZE; i++){

        if(start[i] == 0){
            Matrix[start[i]][end[i]-1] = 1;
            Matrix[end[i]-1][start[i]] = 1;
        }else if(end[i] == 0){
            Matrix[start[i]-1][end[i]] = 1;
            Matrix[end[i]][start[i]-1] = 1;
        }else{
```

```
            Matrix[start[i]-1][end[i]-1] = 1;
            Matrix[end[i]-1][start[i]-1] = 1;
        }
    }

    if(vertexs > 0){

        cout << \n;

        cout << Matrix:  << \n;

        for(int a = 0; a < vertexs; a++){

            for(int b = 0; b < vertexs; b++){
                cout << Matrix[a][b] << //" " ;
            }

            cout << endl;
        }
    }
}
```

The Matrix is the first graph representation that we will be talking about. The function takes in the the vertex, this is the number of vertices for this particular graph. Then there are two vectors, start holds all the edges that are the beginning of the edge, and the end vector holds all the edges that are the end of the edge. These three values are used to generate what the graph will look like.

Vertex will be used to set the size of the 2d array. It will set it to the amount of vertices so that the edges are all able to be inputted into the 2d array. Then start and end are used to tell the matrix array where to put a 1 instead of a 0 (meaning no edge) to signify that there is a edge at those two points. This will be done by going to the Matrix 2d array at [start][end] and then again at [end][start] this is because these graphs are bi directional. Meaning that the edges go both ways for the graph. Also in the code there is a -1 on specific things. This is to make sure that we are not doing -1 from 0 which will cause a error, and if the code is to input the data in correctly arrays are in base 0 not 1 so we have to subtract a 1 from the start and end to get the correct array input place.

For the print out the program will run through every column row and print out everything at that column. This happens when the a goes through everything from 0 to the total vertex amount. Then does the same for the columns. Prints out everything from 0 to the total vertex amount. This will print out everything in the Matrix since the matrix is total vertex size so it will print out all rows and columns of the Matrix.

5.2 Adjacency List

```
void AdjacencyList(int vertexs, vector<int> start, vector<int> end, int count){
    vector <int> neighbors[vertexs];
    int VECTOR_SIZE = 0;

    if(start.size() == end.size()){
        VECTOR_SIZE = start.size();
    }else if(start.size() > end.size()){
        VECTOR_SIZE = start.size();
    }else{
        VECTOR_SIZE = end.size();
    }

    for(int i = 0; i < VECTOR_SIZE; i++){

        if(start[i] == 0){
            neighbors[start[i]].push_back(end[i]);
            neighbors[end[i]-1].push_back(start[i]);
        }else if(end[i] == 0){
```

```
            neighbors[start[i]-1].push_back(end[i]);
            neighbors[end[i]].push_back(start[i]);
        }else{
            neighbors[start[i]-1].push_back(end[i]);
            neighbors[end[i]-1].push_back(start[i]);
        }
    }

    if(vertexs > 0){
        if(count == 5){
            cout << \n;

            cout << Adjacency List:  << \n;


            for (int i = 0; i < vertexs; i++) {
                cout << Vertex  << i << : ;
                for (int j = 0; j < neighbors[i].size(); j++) {
                    cout << neighbors[i][j] << //" ";
                }
                cout << endl;
            }

        }else{
            cout << \n;

            cout << Adjacency List:  << \n;

            for (int i = 0; i < vertexs; i++) {

                cout << Vertex  << i + 1 << : ;

                for (int j = 0; j < neighbors[i].size(); j++) {
                    cout << neighbors[i][j] << //" ";
                }
                cout << endl;
            }

        }

    }

}
```

The Adjacency List is the second graph representation that we will be talking about. For this graph representation we will make a array of vectors called neighbors. The neighbors array will have a vector at each of the indexes of the array this will be used to hold the end value that the edge is at. The start value would be the index of the vector we are adding the end edge too.

The same at the Matrix graph representation the start and end vector also have the start and end values for the edges. The count is used to tell weather the start of the graph is 0 or not. To store the representation of the graph in this context we go to neighbors array sub start and add the end part of the edge to the vector that is in that array. As well as the reverse since it is a bi directional graph. This also has the same error checking if the graph starts at 0 or 1.

Then for the print out the program goes through every item in the vector at neighbors array[i]. The i value goes through the whole neighbor array from i starting at 0 to the vertex amount. Then print out everything in that vector. This is form j starting at 0 to the size of the vector at neighbors[i].

5.3 Link Objects (LinkedObj): Queue

```
struct QueueNode{
```

```
    int data;
    QueueNode* link;
};

struct Queue{
    QueueNode* front;
    QueueNode* back;

    Queue(){
        front = nullptr;
        back =  nullptr;
    }

    void EnQueue(int info) {
        struct QueueNode * ptr;

        ptr = (struct QueueNode * ) malloc(sizeof(struct QueueNode));

        ptr->data = info;
        ptr->link = NULL;

        if ((front == NULL) && (back == NULL)) {
            front = back = ptr;
        } else {
            back-> link = ptr;
            back = ptr;
        }

    }

    /* Diagram:
        head -> link
        tail -> link (new iteam)
        iteam -> link (becomes new tail) ...
    */

    int DeQueue(){
        if (front == NULL) {
            return 0;
        }

       QueueNode* temp = front;

        front = front->link;

        int data =  temp->data;

        delete temp;

        if(front == NULL){
            back = NULL;
        }

       return data;

    }

    bool isEmptyQueue() {
        return front == nullptr;
    }
```

```
62
63     /* Diagram:
64         head -> link
65         tail -> link (new iteam)
66         iteam -> link (becomes new tail) ...
67
68         same thing as above but in reverse
69
70         head -> link (takes out this item)
71         item -> link (this becomes new head)
72         tail -> link
73
74     */
75 };
```

I had to change how my original Queue worked. The Queue from Assignment 1 did not properly allocate the temp pointer that would be used to store the new data. When every I used my old Queue the back would be over written the front would be set to pointer 0xbAAdf00dbAAdf00d. This is a special memory address that is used in vscode to show that memory allocation has gone bad in the program. I added line 18, 20 and 21 so that there is a proper allocation of the memory that will then be put at the front if the back and front is NULL, or to the back if the back is null. Other than that the DeQeueu is the same as the old lab. I have simplified the isEmptyQueue function to return the true or false if the front is equal to nullptr.

5.4 Link Objects (LinkedObj)

```
struct LinkedObj{
    string node;
    vector<int> neighbors;
    bool IsProcessed =  false;
};

void LinkedObjs(int vertexs, vector<int> start, vector<int> end, int count) {

    LinkedObj Vertecies[vertexs];
    int VECTOR_SIZE = 0;

    if(start.size() == end.size()){
        VECTOR_SIZE = start.size();
    }else if(start.size() > end.size()){
        VECTOR_SIZE = start.size();
    }else{
        VECTOR_SIZE = end.size();
    }

    for(int i = 0; i < vertexs; i++){
        Vertecies[i].node = to_string(i+1);
    }

    for(int i = 0; i < VECTOR_SIZE; i++){

        if(start[i] == 0){
            Vertecies[start[i]].neightbors.push_back(end[i]);
            Vertecies[end[i]-1].neightbors.push_back(start[i]);
        }else if(end[i] == 0){
            Vertecies[start[i]-1].neightbors.push_back(end[i]);
            Vertecies[end[i]].neightbors.push_back(start[i]);
        }else{
            Vertecies[start[i]-1].neightbors.push_back(end[i]);
            Vertecies[end[i]-1].neightbors.push_back(start[i]);
        }
```

```
    }
}
```

The Linked Objects is the third graph representation that we will be talking about. There is another array that is set to the size of the total amount of vertices. As well as the start and the end for the edges and the count to tell if this graph starts at 0 or 1.

But for this instead of the array having vectors to store everything we have the LinkedObj class to store it all. There is the node, this will be the name of the node. The neighbors vector, this will hold all the neighbors that we will have. Finally there is the IsProcessed, this is used to for depth first search (DFS) and breath first search (BFS). For the input the program will go to the array[start] value and look at the neighbors vector and add the end to it. As well as the reverse since it is a bi directional graph. This also has the same error checking if the graph starts at 0 or 1.

5.5 Link Objects (LinkedObj): Print

```
cout << \n;

cout << Linked Objects:  << \n;

for (int i = 0; i < vertexs; i++) {

    cout << Neighbors of Node  << stoi(Vertecies[i].node) << : ;

        for (int neighbor : Vertecies[i].neightbors) {
            cout << neighbor << //" ";
        }
    cout << endl;
}

int id = 0;

cout << \n;

cout << Depth First Search:  << \n;

DepthFirstSearch(Vertecies, id, count);

for(int i = 0; i < vertexs; i++){
    Vertecies[i].IsProcessed = false;
}

BreathFirstSearch(Vertecies, id, count);

cout << \n;

cout << Not Connected Nodes:  << \n;

for(int i = 0; i < vertexs; i++){
    if(Vertecies[i].IsProcessed == false){
         cout << stoi(Vertecies[i].node) <<  is not connect :( << \n;
    }
}
```

Then for the print out the program goes through every item in the Vertecies array. Then for all neighbors in the neighbors part of the linked object at Vertecies[i], print out the neighbor in there. BFS and DFS handle there own print out. At the end of the print out is something simple to show all unprocessed vertices. This was added because a vertex can be unprocessed if it has no neighboring value or connection to it. All the print out does it go through he LinkedObj and check to see if the IsProcessed value is false or not, if so then it was never processed and will be printed out.

5.5.1 Depth First Search (DFS)

```
void DepthFirstSearch(LinkedObj Vertecies[], int id, int count){
    if (Vertecies[id].IsProcessed == false) {

        Vertecies[id].IsProcessed = true;

        if(count == 5){
             cout << Visited node  << stoi(Vertecies[id].node) - 1 << endl;
        }else{
             cout << Visited node  << stoi(Vertecies[id].node) << endl;
        }

        for (int neighbor : Vertecies[id].neightbors) {
            DepthFirstSearch(Vertecies, neighbor - 1, count);
        }
    }
}
```

Depth first search (DFS) is the first type of search that is we cover. This type of search goes deep into the graph and then goes wide outputting all the vertex ids as it goes along through the graph. This is performed with by getting the first vertex id (0). Before the current vertex is printed out make sure to make the vertex you are on IsProcessed value to be true. Then we get that node from the Vertecies[i] and output the id for that node in the array. Then for all neighbors that are in Vertecies[id] recursively send those to DepthFirstSearch. The recursion causes a stack data structure to be implemented into the program, this stack is the run time stack that all programming languages have. This effectively is the graph going to each vertex using the edges to travel from vertex to vertex. This means the the time complexity of this search is O(v + e), where v is the collection of vertices and e is the collection of edges. These are both added together to get the total amount of things check because in order for the graph to be fully traversed the program has to visit all vertices and go to each vertex via there respective neighbor connections / edges.

5.5.2 Breath First Search (BFS)

```
void BreathFirstSearch(LinkedObj Vertecies[], int id, int count){

    Queue BFSQueue;

    BFSQueue.EnQueue(id);

    Vertecies[id].IsProcessed = true;

    cout << \n;

    cout << Breath First Search:  << \n;

    while(BFSQueue.isEmptyQueue() == false){

        int current =  BFSQueue.DeQueue();

        if(count == 5){
            cout << Visited Node:  << stoi(Vertecies[current].node) - 1 << endl;
        }else{
            cout << Visited Node:  << Vertecies[current].node << endl;
        }

        for(int neighbor: Vertecies[current].neightbors){
            if(!Vertecies[neighbor-1].IsProcessed){
                Vertecies[neighbor-1].IsProcessed = true;
                BFSQueue.EnQueue(neighbor-1);
            }
        }
    }

}
```

Breath first search (BFS) is the second type of search that is we cover. Does the opposite of the DFS, for this search the traversal goes wide on the graph first then goes deeper into the graph. This anomaly happens because we use a Queue rather than a stack. Like DFS, BFS is given the first node to take a look at, this is 0. It will go to the Vertecies array[0] and pull the id from it and put that id into the Queue, and make the id's IsProcessed value true. Then there is a while loop that will only stop once the Queue is empty. Then the program takes out the first item in the Qeue and a simple print out is used. Finally the for loop will check all the neighbors of the Vertecies[current], current being the node that you are currently processing, the program will go through the whole neighbors array for the current value, process them as true and then put that id into the Queue with proper assignment. This is effectively making the program go wider, since the program is processing all the neighbors and putting them in the Queue to be printed out first rather than in DFS where it will automatically process the next unprocessed vertex via recursion.

Even though the approach for both of these searches are different the time complexity is the same. BFS has time complexity O(v + e). This is because like DFS the program will still have to go to all vertices in the graph. Whilst it is doing that the program is using the neighbor connections/ edges to travel through the whole graph. This means that like DFS, BFS also will eventually travel through all vetices and edges in order to do the dropper work.

6 References

Geeks For Geeks Vector in C++
Geeks For Geeks Passing Vecotors
This lab was my first time using a c++ vector so I need a lot of help with this topic and got these sources to help me out.

Geeks For Geeks String Stream
This one along with the 9th link where what helped me to figure out how to use a string stream and what I can do with it.

Geeks For Geeks Vecotor Erase
Geeks For Geeks Vector Empty
This was used to figure out how to clear start and end vector for every time there is the new graph command is seen in the file input.

Geeks For Geeks Array of Vectors
Geeks For Geeks C++ Print 2d Array
This was used to show me how to correctly pass the data that I would need to properly represent all the graphs.

Stack Overflow Iterating through a C++ Vector
This was used to show how to do a proper iteration through a vector to do the print out for the graph representations.

Stack Overflow Extract int from String Stream
C++ String Find
This is used to find a specific string in a string. This is what I use to find the key words in the commands for the graph1.txt file.

Youtube In Order Print
This is the video that I used to create the in order print out for the BST.

Scaler Queue for Linked List
What I used to fix my Queue and the memory allocation problem that I had mentioned.

7 Appendix

7.1 Main.cpp

```cpp
//first two are librarys
#include <iostream> //object oriented library that allows input and output using streams
#include <fstream> //allows for the reading of a file in the library
```

```
#include <string> /* These three are used for the removing of a space for the strings */
#include <algorithm> /* These three are used for the removing of a space for the strings */
#include <cctype> /* These three are used for the removing of a space for the strings */
#include <iomanip> //used to set the amount of accuracy for the decmial points
#include <vector>
#include <sstream>
#include BST.hpp
#include Matrix.hpp
#include AdjacencyList.hpp
#include LinkedObjects.hpp

using namespace std;

const int arraysize = 666;
const int BSTsize = 42;
int palanplacecheck = 1;

//main functions
int main(){

    cout << \n;

    int count = 0;

    int magicIteamCount = 0;
    string magicIteam[arraysize];

    //start of the file stream studd
    string readInString;

    //opens the right file
    ifstream File (magicitems.txt);

    //checks if the file is open
    if (File.is_open()){

        //while file is open gets the line
        while (File.good()){

            getline(File, readInString);

            //add the now properly formatted line to the array
            magicIteam[magicIteamCount] = readInString;
            magicIteamCount++;


        }
        //closed the file at the end when all done
        File.close();
    }

    //error checking if the file is not opened
    else cout << Unable to open file;

    //used to creat the root of the tree
    BST* root;
    root = new BST;
```

```
    //goes through the whole array of magic items to see where they will be inserted into the binary tree
    for(int i = 0; i < arraysize; i++){
        //this string will output the path taken to where the magicitem was placed
        //it gets passed the tree's root to start and the item it is adding
        string insertPath = BSTTreeInsert(root , magicIteam[i]);
        cout << This is the path of  + magicIteam[i] +  is  + insertPath + . << \n;
    }

    //to split the two sections up
    cout << \n;

    count = 0;

    //makes a new array to be written too for the stuff that will looked for in the BinaryTree
    int BSTcount = 0;
    string BSTitem[BSTsize];

    //start of the file stream studd
    readInString;

    //opens the right file
    ifstream BST (magicitems-find-in-bst.txt);

    //checks if the file is open
    if (BST.is_open()){

        //while file is open gets the line
        while (BST.good()){

            getline(BST, readInString);

            //add the now properly formatted line to the array
            BSTitem[BSTcount] = readInString;
            BSTcount++;


        }
        //closed the file at the end when all done
        BST.close();
    }

    //error checking if the file is not opened
    else cout << Unable to open file;

    //call the BSTSearch function to search for the 42 items that where inputed from the magicitems-find-in-bst.txt file
    for(int i = 0; i < BSTcount; i++){
        BSTSearch(root, BSTitem[i]);
    }

    cout << \n;

    InOrderPrint(root);

    //start of the graph

    //makes a new array to be written too for the stuff that will looked for in the
```

```
BinaryTree
 vector<string> GraphVector;

 //start of the file stream studd
 readInString;

 //opens the right file
 ifstream Graph (graphs1.txt);

 //checks if the file is open
 if (Graph.is_open()){

     //while file is open gets the line
     while (Graph.good()){

         getline(Graph, readInString);

         //add the now properly formatted line to the array
         GraphVector.push_back(readInString);


     }
     //closed the file at the end when all done
     Graph.close();
 }

 //error checking if the file is not opened
 else cout << Unable to open file;

 //values that will be used to check things
 int vertexs = 0;
 string start = 0;
 string end = 0;
 vector<int> StartList;
 vector<int> EndList;
 int Gcount = 0;

 //will go throught the vector that contains all the lines in the graph1.txt file
 for(string i : GraphVector){
     //check if it does not see the comment line
     //if so start processing
     if(i.find(--) == std::string::npos){

         //if it sees new (for new graph) or the end of the vector then reset
everything
         if(i.find(new) != std::string::npos || i == GraphVector.back()){

             //print out as Matrix
             MatrixGraph(vertexs, StartList, EndList);

             //print out as Adjacency list
             AdjacencyList(vertexs, StartList, EndList, Gcount);

             //print out as linked objects
             LinkedObjs(vertexs, StartList, EndList, Gcount);

             //resets all data back to nothing to get the data of the next graph
             vertexs = 0;
             StartList.clear();
             EndList.clear();
```

```

            }else if(i.find(vertex) != std::string::npos){

                //if you find vertex in i that means a new vertex so add to the total amount
                vertexs++;

            }else if(i.find(edge) != std::string::npos){

                //if you find the edge then start pulling the start and end value for the edge to find out what two vertexes are connected to each other4

                //this will be used with the tokens to read and extract tokens out of the string i
                std::istringstream iss(i);
                std::string token;

                //will be used to switch between vectors
                bool addToVectorA = true;

                //checks the tokens if they are equal will skip them
                while (iss >> token) {
                    if (token == add || token == edge || token == -) {
                        continue;
                    }

                    int num;

                    //this will check if the token is a int and if so add it to the start list first then the second one to the end list
                    if (std::istringstream(token) >> num) {
                        if (addToVectorA) {
                            StartList.push_back(num);
                        } else {
                            EndList.push_back(num);
                        }

                        // Toggle between vectors
                        addToVectorA = !addToVectorA;

                        //if the graph starts at zero this value will be used to tell the out functions to start at 0 instead of 1
                        if(num == 0){
                            Gcount = 5;
                        }
                    }
                }
            }
        }
    }
}
```

7.2 BST.hpp

```
//librarys that are always used in c++
#include <iostream>
using namespace std;

int comparisons = 0;
```

```cpp

//This class will be the nodes that are made to make the Binary Search Tree
//The data type for data is string since there will whole strings passed into it
struct BST{
    string data;
    //the left side for values lesser than the one before
    BST* left = nullptr;
    //the right side for values greater than the one before
    BST* right = nullptr;
    //to find the one that it came from
    BST* parent = nullptr;
};

string BSTTreeInsert(BST*& root, string value){
    //used to find the parent node this will be trailing one behind in the search
    BST* trailing = nullptr;
    //the one you are on will start at the root and then go up the tree
    BST* current = root;
    //the pathway that will be taken
    string path = //"";

    //a new node that will have the value passed as the data for it so we have a new node where the node gets place
    BST* node = new BST;
    node->data = value;

    //while the place you are at is not at the end check
    while(current != nullptr){
        //make the trailing the current so we are a step behind
        trailing = current;
        //if less then the current is the left side
        if(node->data < current->data){
            current = current->left;
            path = path + L;
        //if greater then the current is the right side
        }else{ // greater
            current = current->right;
            path = path + R;
        }
    }

    //the parent become the one behind it so we know where it came from
    node->parent = trailing;
    //if trailing is null than that node is the root
    //so the first node will always be the root
    if(trailing == nullptr){
        root = node;
    }else{
        //if not and the data is less then at the the end the new node is the left side
        if(node->data < trailing->data){
            trailing->left = node;
        //if not and the data is greater then at the the end the new node is the right side
        }else{ // greater
            trailing->right = node;
        }
    }

    //return the path string taken for print out
```

```
    return path;

}

//passes the key we are looking for and the node (the start or root)
BST* BSTSearch(BST*& node, string key){

    //if the node left and right is null then output (not there), if the data is the key also output (found)
    if(node->data == key){
        cout << :- << node->data <<  is done searching. Comparisons =  << comparisons << \n;
        comparisons = 0;
        return node;
    //if the key is less then data then return the left side to go down next
    }else if(key < node->data){
        comparisons++;
        cout << L;
        return BSTSearch(node->left, key);
    //if the key is greater then data then return the right side to go down next
    }else{ // greater
        comparisons++;
        cout << R;
        return BSTSearch(node->right, key);
    }
}

//performs a in order print out of all the things in the BST
void InOrderPrint(BST* node){

    if(node != nullptr){
        //starts at the left of the root node

        //goes left as long as the left is not null
        if(node->left != nullptr){
            InOrderPrint(node->left);
        }

        //prints out
        cout << node->data << \n;

        //goes right as long as the right is not null
        if(node->right != nullptr){
            InOrderPrint(node->right);
        }

    }else{
        //tells us if the tree is empty
        cout << The Tree is Empty! << \n;
    }


}
```

7.3 Matrix.hpp

```
//librarys that are always used in c++
#include <iostream>
#include <vector>
```

```
using namespace std;

void MatrixGraph(int vertexs, vector<int> start, vector<int> end) {
    //makes a 2d array of row vertex and colum vertex
    int Matrix[vertexs][vertexs];
    int VECTOR_SIZE = 0;

    //to check howe many edges there are (for is the start has more than the end)
    if(start.size() == end.size()){
        VECTOR_SIZE = start.size();
    }else if(start.size() > end.size()){
        VECTOR_SIZE = start.size();
    }else{
        VECTOR_SIZE = end.size();
    }


    //initilize everything so now outputing errors
    for(int i = 0; i < vertexs; i++){
        for(int j = 0; j < vertexs; j++){
            Matrix[i][j] = 0;
        }
    }

    //will go to Matrix array row start column end and make the value a 1 instead of a 0
    for(int i = 0; i < VECTOR_SIZE; i++){

        //checking if given a 0 vertex
        if(start[i] == 0){
            Matrix[start[i]][end[i]-1] = 1;
            Matrix[end[i]-1][start[i]] = 1;
        }else if(end[i] == 0){
            Matrix[start[i]-1][end[i]] = 1;
            Matrix[end[i]][start[i]-1] = 1;
        }else{
            Matrix[start[i]-1][end[i]-1] = 1;
            Matrix[end[i]-1][start[i]-1] = 1;
        }
    }

    if(vertexs > 0){

        //print out for the Matrix array
        cout << \n;

        cout << Matrix:  << \n;

        //goes through every row
        for(int a = 0; a < vertexs; a++){

            //and prints everything out in that column at that row
            for(int b = 0; b < vertexs; b++){
                cout << Matrix[a][b] << //" ";
            }

            cout << endl;
        }
    }

```

```

}
```

7.4 AdjacencyList.hpp

```
//librarys that are always used in c++
#include <iostream>
#include <vector>
using namespace std;

void AdjacencyList(int vertexs, vector<int> start, vector<int> end, int count){
    //makes a array with vectors or size Vertexs.
    vector <int> neighbors[vertexs];
    int VECTOR_SIZE = 0;

    //to check howe many edges there are (for is the start has more than the end)
    if(start.size() == end.size()){
        VECTOR_SIZE = start.size();
    }else if(start.size() > end.size()){
        VECTOR_SIZE = start.size();
    }else{
        VECTOR_SIZE = end.size();
    }

    //will go through the VECTOR_SIZE amount and add to what every the start is the
    end value connected to and vis versa
    for(int i = 0; i < VECTOR_SIZE; i++){

        //checking if given a 0 vertex
        if(start[i] == 0){
            neighbors[start[i]].push_back(end[i]);
            neighbors[end[i]-1].push_back(start[i]);
        }else if(end[i] == 0){
            neighbors[start[i]-1].push_back(end[i]);
            neighbors[end[i]].push_back(start[i]);
        }else{
            neighbors[start[i]-1].push_back(end[i]);
            neighbors[end[i]-1].push_back(start[i]);
        }
    }

    if(vertexs > 0){
        //special print if the vertex start at 0 instead of 1
        if(count == 5){
            cout << \n;

            cout << Adjacency List:  << \n;


            for (int i = 0; i < vertexs; i++) {
                cout << Vertex  << i << : ;
                for (int j = 0; j < neighbors[i].size(); j++) {
                    cout << neighbors[i][j] << //" ";
                }
                cout << endl;
            }

        }else{
            //print out if the vertex starts a 1
```

```
            cout << \n;

            cout << Adjacency List:  << \n;

            //goes to every vertex in the neighbors array
            for (int i = 0; i < vertexs; i++) {

                //prints out the vertex
                cout << Vertex  << i + 1 << : ;

                //and all the neighbors at that specific vertex (kinda like a 2d array)
                for (int j = 0; j < neighbors[i].size(); j++) {
                    cout << neighbors[i][j] << //" ";
                }
                cout << endl;
            }

        }

    }

}
```

7.5 LinkedObjects.hpp

```
//librarys that are always used in c++
#include <iostream>
#include <vector>
#include <string>
using namespace std;


//new class for the LinkedObj
struct LinkedObj{
    string node;
    vector<int> neightbors;
    bool IsProcessed =  false;
};

struct QueueNode{
    int data;
    QueueNode* link;
};

struct Queue{
    //Node pointer for the front and back
    QueueNode* front;
    QueueNode* back;

    Queue(){
        //sets the front and back to null
        front = nullptr;
        back =  nullptr;
    }

    void EnQueue(int info) {
        //creates a new point and make the data for it the id passed (called info)
        //and the link null
```

```
        struct QueueNode * ptr;

        ptr = (struct QueueNode * ) malloc(sizeof(struct QueueNode));

        ptr->data = info;
        ptr->link = NULL;

        //if the queue has nothing make the front the new thing and the back
        if ((front == NULL) && (back == NULL)) {
            front = back = ptr;
        } else { //else add the new data to the back
            back-> link = ptr;
            back = ptr;
        }

    }

    /* Diagram:
        head -> link
        tail -> link (new iteam)
        iteam -> link (becomes new tail) ...
    */

    int DeQueue(){
        //checks if front is null or at the top
        if (front == NULL) {
            return 0;
        }

        //creates a temp node pointer
        QueueNode* temp = front;

        //sets the front  variable to the link of front
        front = front->link;

        //set data of char type to the temp->datas data
        int data =  temp->data;

        //delets temp since not needed anymore
        delete temp;

        //makes it all null
        if(front == NULL){
            back = NULL;
        }

        //returns the data
        return data;

    }

    //checks if the Queue is empty
    bool isEmptyQueue() {
        return front == nullptr;
    }

    /* Diagram:
        head -> link
        tail -> link (new iteam)
        iteam -> link (becomes new tail) ...
```

```
        same thing as above but in reverse

        head -> link (takes out this item)
        item -> link (this becomes new head)
        tail -> link

    */
};

void DepthFirstSearch(LinkedObj Vertecies[], int id, int count){
    //will see if the node at the array of LinkeObjs is processed
    if (Vertecies[id].IsProcessed == false) {

        //if not make true and print out
        Vertecies[id].IsProcessed = true;

        //print out if first vertex is 0
        if(count == 5){
            cout << Visited node  << stoi(Vertecies[id].node) - 1 << endl;
        }else{ //if first vertex is 1
            cout << Visited node  << stoi(Vertecies[id].node) << endl;
        }

        //then will recurivly send the neighbors to the fucntion to have the same
    thing happen to them
        for (int neighbor : Vertecies[id].neightbors) {
            DepthFirstSearch(Vertecies, neighbor - 1, count);
        }
    }
}

void BreathFirstSearch(LinkedObj Vertecies[], int id, int count){

    //creates the queue
    Queue BFSQueue;

    //enques the first thing of the id of the first thing
    BFSQueue.EnQueue(id);

    //sets the processed value to true
    Vertecies[id].IsProcessed = true;

    cout << \n;

    cout << Breath First Search:  << \n;

    //while the queue is not empty do these actions
    while(BFSQueue.isEmptyQueue() == false){

        //then dequeus the first item and prints it out
        int current =  BFSQueue.DeQueue();

        if(count == 5){
            cout << Visited Node:  << stoi(Vertecies[current].node) - 1 << endl;
        }else{
            cout << Visited Node:  << Vertecies[current].node << endl;
        }

        //for all the neighbors in the place
```

```
        for(int neighbor: Vertecies[current].neightbors){
            if(!Vertecies[neighbor-1].IsProcessed){
                //if they are not processed then process them (set processed to true)
                Vertecies[neighbor-1].IsProcessed = true;
                //and enqueue it
                BFSQueue.EnQueue(neighbor-1);
            }
        }
    }

}

void LinkedObjs(int vertexs, vector<int> start, vector<int> end, int count) {

    //will create a array of LinkedObj called Vertecies to store them all
    LinkedObj Vertecies[vertexs];
    int VECTOR_SIZE = 0;

    //to check howe many edges there are (for is the start has more than the end)
    if(start.size() == end.size()){
        VECTOR_SIZE = start.size();
    }else if(start.size() > end.size()){
        VECTOR_SIZE = start.size();
    }else{
        VECTOR_SIZE = end.size();
    }

    //give the name to the node value of each of the LinkedObj
    for(int i = 0; i < vertexs; i++){
        Vertecies[i].node = to_string(i+1);
    }

    //will go to Vertecies sub start the neighbors vector and push_back the end value
    to it
    for(int i = 0; i < VECTOR_SIZE; i++){

        //checking if given a 0 vertex
        if(start[i] == 0){
            Vertecies[start[i]].neightbors.push_back(end[i]);
            Vertecies[end[i]-1].neightbors.push_back(start[i]);
        }else if(end[i] == 0){
            Vertecies[start[i]-1].neightbors.push_back(end[i]);
            Vertecies[end[i]].neightbors.push_back(start[i]);
        }else{
            Vertecies[start[i]-1].neightbors.push_back(end[i]);
            Vertecies[end[i]-1].neightbors.push_back(start[i]);
        }
    }

    if(vertexs > 0){

        //print out if the first vertex starts at 0
        if(count == 5){
            cout << \n;

            cout << Linked Objects:  << \n;

            for (int i = 0; i < vertexs; i++) {

                //goes to each of the Vertecies get the node and prints out
```

```
            cout << Neighbors of Node  << stoi(Vertecies[i].node) - 1 << : ;

                //then prints out all of thats nodes neighbors
                for (int neighbor : Vertecies[i].neightbors) {
                    cout << neighbor << //" ";
                }
            cout << endl;
        }

        //id is the first node we want to process
        int id = 0;

        cout << \n;

        cout << Depth First Search:  << \n;

        //then goes to depth first search to do that print out as well
        DepthFirstSearch(Vertecies, id, count);

        //resets values for the breath first search
        for(int i = 0; i < vertexs; i++){
            Vertecies[i].IsProcessed = false;
        }

        //then goes to breath first search to do that print out as well
        BreathFirstSearch(Vertecies, id, count);

        cout << \n;

        cout << Not Connected Nodes:  << \n;

        //will print out all no connected values
        for(int i = 0; i < vertexs; i++){
            //if not connected they where never processed so there value would be false
            if(Vertecies[i].IsProcessed == false){
                 cout << stoi(Vertecies[i].node)-1 <<  is not connect :( << \n;
            }
        }

    }else{
        //print out if the vertex starts a 1
        cout << \n;

        cout << Linked Objects:  << \n;

        for (int i = 0; i < vertexs; i++) {

            //goes to each of the Vertecies get the node and prints out
            cout << Neighbors of Node  << stoi(Vertecies[i].node) << : ;

                //then prints out all of thats nodes neighbors
                for (int neighbor : Vertecies[i].neightbors) {
                    cout << neighbor << //" ";
                }
            cout << endl;
        }

        //id is the first node we want to process
        int id = 0;
```

```
            cout << \n;

            cout << Depth First Search:  << \n;

            //then goes to depth first search to do that print out as well
            DepthFirstSearch(Vertecies, id, count);

            for(int i = 0; i < vertexs; i++){
                Vertecies[i].IsProcessed = false;
            }

            //then goes to breath first search to do that print out as well
            BreathFirstSearch(Vertecies, id, count);

            cout << \n;

            cout << Not Connected Nodes:  << \n;

            for(int i = 0; i < vertexs; i++){
                if(Vertecies[i].IsProcessed == false){
                     cout << stoi(Vertecies[i].node) <<  is not connect :( << \n;
                }
            }
        }

    }

}
```