CMPT 435 - Fall 2020 - Dr. Labouseur

Assignment Four – Graphs and BSTs
Graphs, Searching Graphs, Their Representations, and Binary Search Trees

Sam Alcosser
Samuel.Alcosser1@Marist.edu

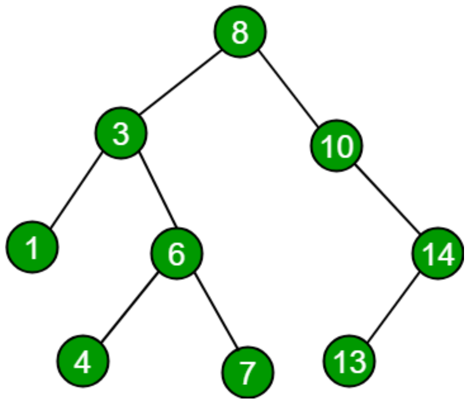
November 15, 2020

Contents

1 Introduction

This document describes the implementations of Binary Search Trees and unweighted graphs, along with the methods associated with each. As seen in the table of contents, this document will follow the high level structure of first defining each of the data structure's implementations in detail, followed by an asymptotic running time analysis of the searching algorithms within each of the data structures.

2 Binary Search Trees



In a previous document, I described binary search as a way of strategically checking specific points in a set in order to close in on the desired value. Binary Search trees implement a similar strategy. With a binary Search tree, we connect our nodes in such a way that they usually have a parent node, and either, both, or neither a left and right node just as seen in the image above. The root node has no parent, we use this as the starting point for most of our operations. Below you see the header file for a node in a binary search tree I call `binNode.h`.

```
#pragma Once
#include <string>
class BinNode
{
public:
        std::string key;
        BinNode* left = nullptr;
    BinNode* right = nullptr;
        BinNode* parent = nullptr;
};
```

As you can see, this is similar to the nodes seen in previous assignments, although now there are more pointers. One ”pointing” to its parent, one down and to the left, and one down to the right. Just as seen in the graphic above, each node does not need to have both or even any child nodes, but all must have a parent node unless they are the root node.

2.1 `BinTree.h`: Outline of Our Binary Search Tree Class

To begin, it is best to have a general understanding of some of the things that can be done with a binary search tree.

```cpp
#pragma once
#include <string>
#include <iostream>
#include "BinNode.h"

class BinTree{
    public:
    BinNode static search(BinNode * root, std::string key, int &count);
    void  insert(std::string key);
    BinNode * root;
    void pullBatch(std::string items[], int len);
};
```

- `BinTree::search()` is the method we use to search our tree
- `BinTree::insert()` is used to insert a `BinNode` into the tree
- Our `root` pointer keeps tabs on where the start of the binary search tree is
- `BinTree::pullBatch()` is a function created to make large batch searches, such as the one in this assignment's case, much easier

2.2 `BinTree::insert()`: Inserting Into the BST

The process of inserting a node into the tree is probably the most logically complex operation that will be done on the tree.

```
1  void BinTree::insert(std::string key)
2  {
3      BinNode *node = new BinNode();
4      node->key = key;
5      BinNode *ancestorNode = nullptr;
6      BinNode *parent = root;
7      while (parent != nullptr)
8      {
9          ancestorNode = parent;
10         if (node->key < parent->key)
11         {
12             parent = parent->left;
13         }
14         else
15         {
16             parent = parent->right;
17         }
18     }
19     node->parent = ancestorNode;
20     if (ancestorNode == nullptr)
21     {
22         root = node;
23     }
24     else if (node->key < ancestorNode->key)
25     {
26         ancestorNode->left = node;
27         //  std::cout  << "placed node left"<< std::endl;
28     }
29     else
30     {
31         ancestorNode->right = node;
32         // std::cout  << "placed node right"<< std::endl;
33     }
34 }
```

1. First, a pointer to a new node is created with the string passed in by the `key` argument (lines 3,4).

2. The trailing pointer `ancestorNode` and the parent pointer `parent` are initialized to `nullptr` and the root of the tree respectivley (lines 5,6).

3. Within the while loop, `ancestorNode` stays one step behind `parent` (line 9) as they continue moving down left or right. This decision is made whether or not the new `binNode`'s `key` value is bigger or smaller than the parent's current position (left being smaller, right being bigger). This

process continues until parent finds a `nullptr` on a node where the child can go.(lines 7-18).

4. Now, the necessity to maintain a trailing `ancestorNode` is extremely apparent. Since it was constantly one step behind the `parent`, it just so happens that it is maintaining the pointer of the new node's parent pointer value which it is now assigned (line 19).

5. If the `ancestorNode` is a `nullptr`, meaning that the tree is empty, assign the `root` pointer to the new node.(lines 20-23)

6. Otherwise, if the `node`'s `key` value is smaller than that of the `ancestorNode`, assign the new node to the `left` pointer of `ancestorNode`. Otherwise, assign it to the right. (lines 24-33)

2.3 `BinTree::search()`: Searching The Binary Search Tree

As you may have realized, the way in which BSTs are set up makes them fairly easy to logically find values. The method I have implemented is a recursive way of searching for values within the tree.

```
1  BinNode BinTree::search(BinNode *rNode, std::string key, int &count)
2  {
3      if (rNode == nullptr || key == rNode->key)
4      {
5          count++;
6          return *rNode;
7      }
8      else if (key < rNode->key)
9      {
10         count++;
11         return search(rNode->left, key, count);
12     }
13     else
14     {
15         count++;
16         return search(rNode->right, key, count);
17     }
18 }
```

1. In the rare case that the root node is a `nullptr`, or that the `key` passed into the function as an argument, return the root node from the function. This is derived by dereferencing the pointer to the parent node passed into the function initially.(lines 3-7)

2. If the key is smaller than the key of the root node, continue the search starting at the left pointer of the current root node, using the same key.(lines 8-12)

3. Otherwise, we can assume that the key is bigger, and therefore we will continue the search from the right pointer of the current parent node. (lines 13-17)

2.3.1 Asymptotic Running Time of `BinTree::search()`

Hold on, how is the searching function the simplest operation for us? Well, If we start from the root node, we are simply looking if our key is bigger or smaller, then move down, wash, rinse, repeat. This is the elegance of the Binary Search Tree. When set up correctly, it can simplify the searching process greatly, as the process of inserting each element is effectivley sorting it as well.

With this, the complexity of this search algorithm is only $O(log_2(n)$. The reason being that, just as with binary search in arrays, every time that we are comparing, we are effectively cutting our possible range in half.

Although, one must be careful. If the order in which the binary search tree is filled is set in a particularly unlucky way, the complexity can climb up to $O(n)$. This can occur if the tree is filled in such a way that everything is just linked by the left pointer or only the right, effectively creating a linear search.

2.4 `BinTree::pullBatch()`: A Streamlined Way For Batches Of Searches

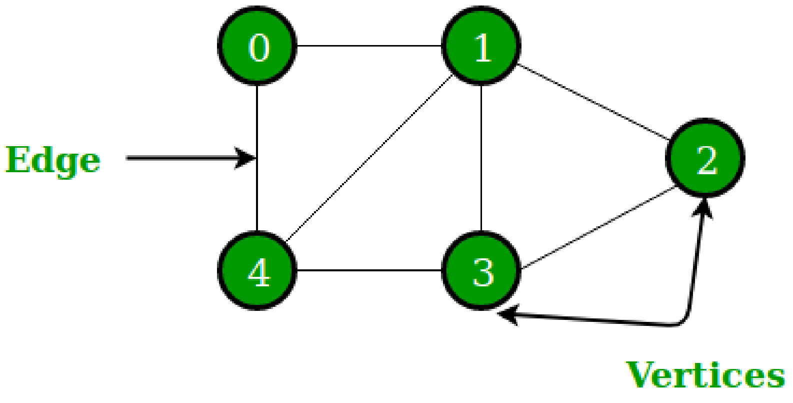
This function is relativley simple logically speaking, and therefore it doesn't require an in depth step by step analysis

```
1  void BinTree::pullBatch(std::string items[], int len)
2  void BinTree::pullBatch(std::string items[], int len)
3  {
4
5      int tCount = 0;
6      for (int i = 0; i < len; i++)
7      {
8          int count = 0;
9          auto res = search(root, items[i], count);
10         std::cout << "Item "<< i+1 << " was found after " << count << " comparisons." << std::endl;
11         tCount += count;
12     }
13     std::cout << "Total comparions to find all items: " << tCount << std::endl;
14     int avg = tCount / len;
15     std::cout << "Average comparisons to reach each item: " << avg << std::endl;
16 }
```

For every index in the array of items to find (line 6), search for it in the BST (line 9) and print and log the comparisons needed to find it (lines 10,11). Then when all of the items have been found, give the average comparisons by dividing the total comparisons by the number of items found(lines 13-15).

3 Undirected Graphs

3.1 Introduction To Graphs



The data structure of an undirected graph is a set of vertices that can be connected to each other by edges. In order to represent the edges, each vertex on eah side of the edge reference eachother in their own array of neighbor vertices. In other words, if vertex a and vertex b share an edge, vertex a would have a reference to vertex b, and vertex b would have a reference to vertex a, both within their respective neighbors arrays.

3.2 `Vertex.h`: The Smallest Unit Of A Graph

As seen below, a single vertex is quite simple having only three parts.

```
#pragma once
#include <iostream>
#include <vector>
class Vertex{
    public:
    int id;
    bool processed;
    std::vector<Vertex*> neighbors;
};
```

- `id` holds the id of the vertex in an int
- `processed` is a sort of flag variable used when searching (more on this later)
- `std::vector<Vertex*> neighbors` is a used to hold a set of `Vertex` pointers as the neighbors of the vertex

3.3 `GraphManager.h`: Maintaining The Graph

The reason why I created a structure called `GraphManager` and not separate graph and searching and managing classes was simple. Since the vertices hold all of the necessary information within themselves, all there is to do to keep them together is to group them into a sort of list or array of some kind. Therefore, The whole class would only really be a vector. So, I instead decided to structure the whole set of operations within one class.

```cpp
#pragma once
#include <iostream>
#include <string>
#include <vector>
#include "Vertex.h"
class GraphManager{
    public:
    std::vector<Vertex*> graph;
    void Overlord(std::string fileName);
    void printAJList();
    void printMatrix();
    void BFSearch(Vertex* startV);
    void DFSearch(Vertex* startV);
    void resetVList();

};
```

- `graph` holds a set of pointers to every vertex in the graph
- `Overlord()` is the function used to run all parts of the graph's functionality (this may seem like bad programming practice but I will explain later why this is done)
- `printAJList()` prints an adjacency list of the different vertices
- `printMatrix()` prints a matrix of the graph's vertices
- `BFSearch()` performs a breadth first search on the graph
- `DFSearch()` performs a depth first search on the graph
- `resetVList()` resets the `processed` flags on every vertex so that another search can be performed

3.4 Printing Methods

These methods are used to show the data in the graphs in two different ways.

3.4.1 `GraphManager::printAJList()`: Printing the Adjacency List

```
void GraphManager::printAJList()
{
        std::cout << "---Printing Adjacency List---" << std::endl;
        for (int i = 0; i < graph.size(); i++)
        {
                std::cout << "[" << graph[i]->id << "]";
                for (int j = 0; j < graph[i]->neighbors.size(); j++)
                {
                        std::cout << " " << graph[i]->neighbors[j]->id;
                }
                std::cout << std::endl;
        }
}
```

For each element in the graph as maintained by the `graph` vector, print out the id of that vertex and all of its neighbors next to it.

3.4.2 `GraphManager::printMatrix()`: Printing the Matrix

```cpp
void GraphManager::printMatrix()
{
        int size = graph.size();
        std::cout << "about to do matrix stuff" << std::endl;
        std::vector<std::vector<char>> matrix(size, std::vector<char>(size, '.'));
        std::cout << "--Printing Matrix--" << std::endl;

        //populating
        for (int i = 0; i < graph.size(); i++)
        {
                for (int j = 0; j < graph[i]->neighbors.size(); j++)
                {

                        matrix[i][graph[i]->neighbors[j]->id] = '1';
                }
        }
        //printing
        std::cout << "    ";
        //table headings
        for (int n = 0; n < graph.size(); n++)
        {
                if (graph[n]->id < 10)
                {
                        std::cout << graph[n]->id << "  ";
                }
                else
                {
                        std::cout << graph[n]->id << " ";
                }
        }
        std::cout << std::endl;

        for (int i = 0; i < graph.size(); i++)
        {
                if (graph[i]->id < 10)
                {
                        std::cout << " " << graph[i]->id << "| ";
                }
                else
                {
                        std::cout << graph[i]->id << "| ";
                }
                //printing row by row

                for (int h = 0; h < matrix[i].size(); h++)
                {
                        std::cout << matrix[i][h] << "  ";
                }
49                  std::cout << std::endl;
50          }
51  }
```

1. Initialize a 2D vector with a length and width equal to the number of vertices with periods to be used as a matrix. (lines 3,5)
2. Use two nested for loops for populating. (lines 9-11).
3. Use the outer loops iterator `i` and the inner loops `j` as vertex and neighbor indicators respectively. For every vertex `i`, place a 1 in the column of the neighbor at index `j`. (lines 9-16)
4. print out each vertex's' id inline, leaving a single space if the id is 1 digit for consistency (lines 20-31)
5. Start a for loop to iterate over each parent index in the matrix, as well as the index of the `graph` vector. (line 33)
6. Start each line by giving a header of the id of the vertex at that index, leaving room again for single digit ids. (lines 35-42)
7. For each character in the sub-vector of the current index, print out each value inline.(lines 45-49)

3.5 `Overlord()`: Running it all

`Overlord()` is a function used to tie everything for the graphs together. Upon first glance, it may seem as if this function breaks a cardinal sin of programming by doing more than one thing in one function, although it is necessary. This function is more like a file reader, as its main purpose is to read in the text file and create a graph. Although, depending on the information on each line, it will add a vertex or edge, do nothing, or perform both print and searching processes on the graph. Then, the complexity of these functions are abstracted out into their own methods.

```
1   void GraphManager::Overlord(std::string fileName)
2   {
3
4           std::fstream newfile;
5
6           std::string ln;
7           int count = 0;
8           bool started = false;
9           newfile.open(fileName, std::ios::in);
10          if (newfile.is_open())
11          {
12                  while (getline(newfile, ln)) //read each line of the file
13                  {
14
```

```
				if (ln.find("new graph") != std::string::npos)
				{

					if (started)
					{
						std::cout << "got to the printing" << std::endl;
						printAJList();
						printMatrix();
						BFSearch(graph[0]);
						resetVList();
						std::cout << "Printing elements as seen through Depth First Search" << s
						DFSearch(graph[0]);
						graph.clear();
					}
					started = true;
				}
				else if (ln.find("add vertex") != std::string::npos)
				{
					int id = std::stoi(ln.substr(11));
					Vertex *vertex = new Vertex();
					vertex->id = id;
					graph.push_back(vertex);
				}
				else if (ln.find("add edge") != std::string::npos)
				{

					int hyph = ln.find("-");

					int num1 = stoi(ln.substr((ln.find("edge") + 5), 2));
					int num2 = stoi(ln.substr(hyph + 2));

					for (int i = 0; i < graph.size(); i++)
					{
						if (graph[i]->id == num1)
						{
							for (int j = 0; j < graph.size(); j++)
							{
								if (graph[j]->id == num2)
								{
									graph[i]->neighbors.push_back(graph[j]);
									graph[j]->neighbors.push_back(graph[i]);
								}
							}
						}
					}
				}
			}
		}
		newfile.close();

```

```
65          return;
66  };
```

1. Begin processing the inputted file line by line. (line 1-12, 63)
2. If the current line says ”new graph” check if the graph has been started (line 15)
3. If the graph has been started, meaning there is something in the graph, print the graph as an adjacency list, a matrix, then perform a breadth first and a depth first search while resetting the vertices processed flag in between the two.(lines 22-26)
4. Clear the `graph` vector to begin a new graph.(line 27)
5. If the line says ”add vertex”, take the number on that line and make a vertex with that id. Then add it to the `graph` vector(lines 31-37)
6. If the line says add edge, begin by parsing out both numbers. (lines 38-44)
7. Using two nested for loops, find the vertex with the id of the first number (`i`) and then in an inner for loop locate the vertex with the id of the second number (`j`).(lines 46-53)
8. Add vertex with id `j` as a neighbor to vertex with id `i`, and vise versa.(lines 54,55)

3.6 `GraphManager::BFSearch`: Breadth First Search

Breadth first search operates in a relatively simple fashion. As it progresses, it keeps iterating over each of the currently processing vertex’s neighbors, seeing if those are processed, and if they are not, then putting them into a queue to be processed themselves.

```
void GraphManager::BFSearch(Vertex *startV)
{
        std::cout << "Printing elements as seen through Breadth First Search" << std::endl;
        Queue q = Queue();
        q.enQueue(startV);
        startV->processed = true;
        while (!q.isEmpty())
        {
                Vertex *cv = q.deQueue().data;
                std::cout << cv->id << std::endl;
                for (int i = 0; i < cv->neighbors.size(); i++)
                {
                        if (!cv->neighbors[i]->processed)
                        {
                                q.enQueue(cv->neighbors[i]);
                                cv->neighbors[i]->processed = true;
                        }
18                  }
19          }
20          std::cout << "------------------------------" << std::endl;
21  }
```

1. Begin a `Queue` and place the starting vertex `startV` on it. (lines 4, 5)
2. Mark the starting vertex as processed. (line 6)
3. Open a while loop that will run until the queue is empty. (line 7)
4. Set our current vertex `cv` to the value of the `data` field in the next `node` to be `deQueued` from the queue.(line 9)
5. Print the id of the `vertex`.(line 10)
6. Begin a for loop that will iterate over each index in the vector of neighbors on the current vertex.(line 11)
7. if the neighbor of the current vertex indicated by the currently operating index is not yet processed, add said neighbor to the queue and set it to processed. (lines 13-17)

3.6.1 Asymptotic Running Time Analysis Of `BFSearch()`

By having a while loop and a for loop nested inside of it, upon first glance one may think that this process is relatively complex in terms of running time. Although there are two things to keep in mind. First of all, most of the operations in this function operate in constant time. Also, in reality we are not operating with a set of sets of data as it may seem, but instead we are operating with a simple list of elements that are all interconnected.

So, when the process is going through each vertex, each of its neighbors, and each of its neighbors and so on, it is all the same set of data. Because of this, breadth first search is actually $O(V + E)$, as the worst case would be that the algorithm has to go to every vertex and every edge. Therefore, this operation's runtime is linear.

3.7 `GraphManager::DFSearch`: Depth First Search

Depth first search is different from breadth first in a couple of ways. First of all, as the name implies, the algorithm processes the vertices by going depth first, trying to go through all of the neighbors of one vertex instead of jumping around as breadth first search does.

```
void GraphManager::DFSearch(Vertex *startV)
{

        if (!startV->processed)
        {
                std::cout << startV->id << std::endl;
7                   startV->processed = true;
8           }
9           for (int i = 0; i < startV->neighbors.size(); i++)
10          {
11                  if (!startV->neighbors[i]->processed)
12                  {
13                          DFSearch(startV->neighbors[i]);
14                  }
15          }
16  }
```

1. If the starting vertex is not processed, print its' id and mark it as processed. (lines 4-8)
2. Start a for loop that will be used to iterate over every neighbor of the starting vertex (line 9)
3. If the neighbor in focus in this iteration is not processed, perform a depth first search starting at this neighbor. (lines 11-14)

3.7.1 Asymptotic Running Time Of `DFSearch`

Just like breadth first search, depth first search is also a linear time operation. The only difference again being that breadth first focuses on getting a hold of each part of the graph, while depth first attempts to fully process each vertex as much as it can. Despite these differences, depth first still has to walk over or process each vertex and edge, giving it the same runtime of $O(V+E)$.

4 `Main::main()`: Execution

`Main::main()` is in all honestly a very simple funcion. And for good reason! By properly abstracting my code, I was able to build a main function that is simple and not messy, yet I still am well aware of what is being executed.

```
int main(){
        setupMagic();
    BinTree * tree = new BinTree();
    populateTree(tree, arr, MAGIC_LEN);
        string randItems[CHOSEN_LEN];
        randPick(arr, randItems, MAGIC_LEN, CHOSEN_LEN);
        tree->pullBatch(randItems, CHOSEN_LEN);
        GraphManager * gm = new GraphManager();
        gm->Overlord("graphs1.txt");
        return 0;
}
```

- `setupMagic()` condenses all of the processes needed to initialize the array of 666 items into a single function. THis is not covered as the same code has been carried over with little to no alteration.

- `tree` becomes the tree we will use for our operations `populateTree()` goes through and calls the `BinTree::insert()` function on all of the items in the array of the magic items.
- Lines 5-7 create a list of 24 random items from the same array and search for them all using `pullBatch()`. This function simplifies the process at this point by making it easier to do all at once
- All that is left is to create a `GraphManager` and call `Overlord()` on it.(lines 8,9)

5 Conclusion

5.1 Results

Search Algorithm	Complexity/ $O()$
Breadth First Search	$O(V + E)$
Depth First Search	$O(V + E)$
Binary Tree Search	$O(log_2(n))$

Here I have put all of the searching algorithms in one place so it is easier to compare.

5.2 References

Cormen, Thomas H., et al. Introduction to Algorithms. MIT Press, 2007.

Geeks, Geeks For. “Check If given Sorted Sub-Sequence Exists in Binary Search Tree.” GeeksforGeeks, 21 May 2019, www.geeksforgeeks.org/check-if-given-sorted-sub-sequence-exists-in-binary-search-tree/.

Geek. “Graph Data Structure And Algorithms.” GeeksforGeeks, www.geeksforgeeks.org/graph-data-structure-and-algorithms/.

Closing Thoughts: A Story

The week prior to handing this in, I was having major troubles with my `Overlord()` function when I was trying to add edges. I just assumed it was an issue with pointers, as clearly that is usually my issue. I left it alone for a day, as I was stumped and could not find the answer. I even opened an online compiler to try and replicate my problem, but it worked! So I had completely lost hope. Then that night a friend from home who is trying to get into computer science texted me.

He wanted my help understanding how for loops work. I felt so high and mighty giving him an in depth explanation of how they work, being happy that I can be of help to him. Then the next day I returned to the same code and got mad at the screen for another hour or so. But then, I realized my mistake. A for loop inside of the function that should look like
`for (int j = 0; j < graph.size(); j++)` actually read
`for (int j = 0;graph.size(); j++)`. I doubt I'll ever make that mistake again.

as a wise person once said, " give a person a program and they'll be frustrated for a day, teach a person to program and they'll be frustrated for the rest of their life.". {*S.A.*}