Gianna Julio

# CMPT435 - Algorithm Analysis and Design
## Assignment 4 - Graphs and Trees

# Assignment4.cpp

## Matrix

```
class Matrix
{
public:
    int matrix[MAX_N][MAX_N] = {};
    int size; // actual size and number of vertices

    void addEdge(int v1, int v2)
    {
        matrix[v1][v2]++;
    }

    void print()
    {
        // print heading
        std::cout << "  |";
        for (int i = 1; i <= size; i++)
        {
            std::cout << " " << i;
        }
        std::cout << "\n"; // new line

        // print contents of matrix
        for (int i = 1; i <= size; i++)
        {
            std::cout << i << " |";
            for (int j = 1; j <= size; j++)
            {
                std::cout << " " << matrix[i-1][j-1];
            }
            std::cout << "\n"; // new line
        }
    }
};
```

This class holds all of the information to display the undirected graphs as a matrix. The matrix is initialized as all zeros, and each time an edge is added the value at those coordinates in the matrix is updated. This class also includes a print function.

## Adjacency List

```
class AdjacencyList
{
```

```
public:
    int adjacencyList[MAX_N][MAX_N] = {};
    int size; // actual size and number of vertices

    void addEdge(int v1, int v2)
    {
        // for each vector connected to v1
        for (int v : adjacencyList[v1])
        {
            // find the next empty space
            if (v == 0)
            {
                // assign new edge connection
                v = v2;
                break;
            }
        }
    }

    void print()
    {
        for (int i = 1; i <= size; i++)
        {
            std::cout << i << " : ";

            for (int j = 0; j < size; j++)
            {
                if (adjacencyList[i - 1][j] != 0)
                {
                    std::cout << adjacencyList[i - 1][j] << ", ";
                }
                else
                {
                    break;
                }
            }
            std::cout << "\n"; // new line
        }
    }
};
```

Similar to the matrix class, the adjacency list class also has functions to add an edge and display the list.

## Linked Objects

```
class Node
{
public:
    int val;
    Node* pointer;
};
```

For the linked objects functionality, I repurposed the simple Node class that holds a value and a pointer to the next value.

### Binary Search Tree

```
// definition of the AVL tree node
struct AVLNode {
    string data;
    AVLNode* left;
    AVLNode* right;
    int height;
};

// function to get the height of a node
int height(AVLNode* node) {
    if (node == nullptr) {
        return 0;
    }
    return node->height;
}

// function to get the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// function to create a new node with a given data
AVLNode* newNode(string data) {
    AVLNode* node = new AVLNode();
    node->data = data;
    node->left = nullptr;
    node->right = nullptr;
    node->height = 1;
    return node;
}

// function to perform a right rotation on a node
AVLNode* rightRotate(AVLNode* y) {
    AVLNode* x = y->left;
    AVLNode* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    // Return the new root
    return x;
}

```

```
// function to perform a left rotation on a node
AVLNode* leftRotate(AVLNode* x) {
    AVLNode* y = x->right;
    AVLNode* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    // Return the new root
    return y;
}

// function to get the balance factor of a node
int getBalance(AVLNode* node) {
    if (node == nullptr) {
        return 0;
    }
    return height(node->left) - height(node->right);
}

// function to insert a new node into the tree
AVLNode* insertNode(AVLNode* node, string data) {
    // Perform the normal BST insertion
    if (node == nullptr) {
        return newNode(data);
    }

    if (data < node->data) {
        node->left = insertNode(node->left, data);
    }
    else if (data > node->data) {
        node->right = insertNode(node->right, data);
    }
    else {
        // Duplicate keys are not allowed
        return node;
    }

    // Update the height of the ancestor node
    node->height = 1 + max(height(node->left), height(node->right));

    // Get the balance factor of the ancestor node
    int balance = getBalance(node);

    // If the node is unbalanced, then there are 4 possible cases

    // Left Left Case
    if (balance > 1 && data < node->left->data) {
        return rightRotate(node);
```

```
    }

    // Right Right Case
    if (balance < -1 && data > node->right->data) {
        return leftRotate(node);
    }

    // Left Right Case
    if (balance > 1 && data > node->left->data) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && data < node->right->data) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    // Return the (unchanged) node pointer
    return node;
}

// function to print the tree in order
void inorderTraversal(AVLNode* node) {
    if (node != nullptr) {
        inorderTraversal(node->left);
        cout << node->data << " ";
        inorderTraversal(node->right);
    }
}
```

The AVLNode structure defines the basics of the binary search tree. It contains the data, pointers to a left and right node, and the height of the current node. The following functions are all helper functions to enable the functionality of the binary search tree. There is a function to find height, find the maximum for comparisons, add a new node, rotate right or left to balance the tree after an addition, to check if the tree is balanced, insert a node, and to traverse the tree.

### Loading Items From File

```
void loadGraphs()
{
    // initialize file
    fstream itemFile;
    itemFile.open("graphs1.txt", ios_base::in);

    if (itemFile.is_open())
    {
        // counter
        int i = 0;

        // initialize objects
        Matrix m;
```

```
        AdjacencyList adj;
        int n = 0; // size

        // while file still has items to read
        while (itemFile.good())
        {
            string line; // initialize item string
            getline(itemFile, line); // get line

            if (line.find("--") != std::string::npos)
            {
                // ignore this line
                //std::cout << "ignore\n";
            }

            else if (line.find("graph") != std::string::npos)
            {
                m.print();      // print old matrix
                adj.print();    // print old adjacency list
                n = 0;          // reset size
            }

            else if (line.find("vertex") != std::string::npos)
            {
                // adjust sizing for each new vertex
                n++;
                m.size = n;
                adj.size = n;
                //std::cout << "vertex" << n << "\n";
            }

            else if (line.find("edge") != std::string::npos)
            {
                // find v1 and v2
                int v1 = 0;
                int v2 = 0;
                m.addEdge(v1, v2);
                adj.addEdge(v1, v2);
                //std::cout << "edge\n";
            }

            i++; // increment counter
        }
        itemFile.close();
    }
}

// load magic items into array from file
void loadItems()
{
    // initialize file
    fstream itemFile;
    itemFile.open("magicitems.txt", ios_base::in);

```

```
256     if (itemFile.is_open())
257     {
258         // counter
259         int i = 0;
260 
261         // while file still has items to read
262         while (itemFile.good())
263         {
264             string item; // initialize item string
265             getline(itemFile, item); // get line
266             transform(item.begin(), item.end(), item.begin(), ::tolower); // transform whole string to lowercase
267             items[i] = item; // store item string
268             i++; // increment counter
269         }
270         itemFile.close();
271     }
272 }
```

The loadGraphs function goes line by line and loads each vertex and edge into a matrix and adjacency list. When a new entry is detected, the matrix an list are cleared and the loop continues until all graphs are read and processed. The loadItems function loads all of the items on the magic list into an array of strings. It opens the file, gets each item, saves it to the global array, then closes the file. I added the statement on line 33 to convert the entire string to lowercase in order to be able to accurately sort the strings.

### Main Function

```
276 int main()
277 {
278     // load and sort items
279     loadItems();
280     mergeSort(items, 0, 666);
281 
282     // create and fill hash table
283     HashTable h;
284     for (string i : items)
285     {
286         h.insert(i);
287     }
288 
289     // initialize random seed
290     srand(time(NULL));
291 
292     // initialize totals - used to compute averages later
293     double totalComparesLinear = 0;
294     double totalComparesBinary = 0;
295     double totalComparesHashes = 0;
296 
297     // loops to pick a new random item, perform all searches, and compile results
298     for (int i = 0; i < 42; i++)
299     {
300         // choose random item
```

```
        string randItem = items[rand() % 666];

        // linear
        double linearComparisons = linearSearch(items, 666, randItem);
        totalComparesLinear += linearComparisons; // add to running total

        // binary
        double binaryComparisons = binarySearch(items, 666, randItem);
        totalComparesBinary += binaryComparisons; // add to running total

        // hash table
        double hashComparisons = h.lookup(randItem);
        totalComparesHashes += hashComparisons; // add to running total

    }

    // compute averages
    double avgComparesLinear = totalComparesLinear / 42.0;
    double avgComparesBinary = totalComparesBinary / 42.0;
    double avgComparesHashes = totalComparesHashes / 42.0;

    // display summary
    display("Linear Search", avgComparesLinear);
    display("Binary Search", avgComparesBinary);
    display("Hash Table with chaining", avgComparesHashes);
}
```

The main function calls the loadItems function first to load the array with all of the magic items, then sorts them with merge sort. The hash table is initialized and all of the items are inserted utilizing the hash table insert function. The total amount of comparisons for each type of search is initialized so the average can be computed later. The random seed is also initialized here to ensure 42 different random items are generated each time. The for loop is executed 42 times, and each time, it randomly selects an item and does a linear, binary, and hashing search for that item, adding each respective number of comparisons to the running totals. After the loop finishes iterating, the averages are computed and displayed with the formatting function.

## Sample Output

```
  | 1 2 3 4 5 6 7
1 | 0 1 0 0 1 1 0
2 | 1 0 1 0 1 1 0
3 | 0 1 0 1 0 0 0
4 | 0 0 1 0 1 0 0
5 | 1 1 0 1 0 1 1
6 | 1 1 0 0 1 0 1
7 | 0 0 0 0 1 1 0

1 : 2, 5, 6
2 : 1, 3, 5, 6
3 : 2, 4
4 : 3, 5
5 : 1, 2, 4, 6, 7
6 : 1, 2, 5, 7
```

```
7 : 5, 6
```

This is a sample of the output after the program is run to display a sample output matrix and adjacency list.