MODULE CODE AND NAME:

DA3306 – DATA STRUCTURES AND ALGORTIHMS

TITLE OF ASSIGNMENT:

Assignment 2: *Binary Search Tree and Directed Graph*

NAME:

Muhammad Hidayat Bin Mohd Yusof (20FTT2910)

Mohammad Izzuddin Bin Abdullah Rosmalishah (20FTT1913

SUBMISSION DATE:

22nd April 2022

MODULE LECTURER:

Norfarrah Muhd Masdi

NAME OF INSTITUTE:

School of Information, Communication and Technology, Polytechnic Brunei

PROGRAMME:

Level 5 Diploma in Data Analytics

Session:

2021 – 2023

Table of Contents

1. Introduction

The purpose of this report is to explain the definition of a Binary Search Tree and a Directed Graph including, their implementation through python code. They are implemented through the use of pre-coded processes to achieve a certain goal called methods or functions. These methods of both topics will be explained in detail of their functionalities as well as their differences from each other.

Additionally, each topic has their own sub-topics with their own specific variations of methods where although similar, would have unique alterations to their code which changes the results they bring.

Hence, by the end of this report the reader will:

- Understand the definition of Binary Search Tree and Directed Graph.
- Understand their methods and how they are applied.
- Understand the differences of each variation within each topic.
- Understand the benefits brought by the created data structure.

2. Binary Search Tree

2.1. Definition –

Binary Search Tree are node-based tree data structures where the left and right subtree are also binary search tree. The left subtree of a Binary Search Tree contains nodes or elements that are less than the node's main element or key node while the right subtree contains nodes or element that are greater than the node's main element or key node.

2.2. Binary Tree vs Binary Search Tree -

Binary Tree is a non-linear data structure where each node can only have at most two child nodes		**Binary Search Tree** is a node-based data structure where the left and right subtree are also Binary Search Tree
Binary Tree is unordered hence slower in process of		Insertion, deletion and searching of an element is faster in **Binary**

insertion, deletion, and searching.		**Search Tree** than **Binary Tree** due to the ordered characteristics
In **Binary Tree**, there is no ordering in terms of how the nodes are arranged		In **Binary Search Tree** the left subtree has elements less than the nodes element and the right subtree has elements greater than the nodes element.

2.3. Methods

2.3.1. Insert(items)

```
def insert(self, value):

        if self.value:
            if value < self.value:
                if self.left is None:
                    self.left = BinarySearchTree(value)
                else:
                    self.left.insert(value)
            elif value > self.value:
                if self.right is None:
                    self.right = BinarySearchTree(value)
                else:
                    self.right.insert(value)
        else:
            self.value = value
```

This method adds new items as child nodes to the tree. It will check the value added is less than the node and if it is, it will add this value to the left child of the tree and if it's more than the node, it will add this value to the right child of the tree.

2.3.2. Delete(item)

2.3.3. deleteMax()

2.3.4. deleteMin()

2.3.5. Search(item)

```
def search(self, value):
      if self.key == value:
        print("The node is present")
        return
      if value < self.key:
        if self.leftchild:
            self.leftchild.search(value)
        else:
          print("The node is empty in the tree!")
      else:
        if self.rightchild:
            self.rightchild.search(value)
            return true
        else:
            print("The node is empty in the tree!")
```

This method will return a Boolean result if the item or value is found within the tree. The function checks if a value is the same as the key value or not. It will check the left child tree whether it is empty or contains nodes for the same value. If not found, it will check the right child tree for this value and if not found, it will return true to give output as" The node is empty in the tree." If it is found, it will return true to give output" the node is present."

2.3.6. getHeight()

```
def height(root):
    if root is None:

        return 0
    leftAns = height(root.left)
    rightAns = height(root.right)
    return max(leftAns, rightAns) + 1

print("Height of the binary tree is: " + str(height(root)))
```

This method will give the height of the tree. The function first checks if the binary tree is empty or not. If true, it will return the number 0 as there is no height. If the tree exists, it will recursively recall the height of each node. It will return the function at each iteration and will give output of the height of the binary tree.

2.3.7. Inorder()

```
def inorder(root):
    if root:
        inorder(root.left)
        print(root.val)
        inorder(root.right)
```

This method will print the values or node in the tree in ascending order. The Function will recur on the left child tree as it is less the key node and then recur on the right child tree where it is more than key node hence it will give output of nodes in an ascending order.

2.3.8. getParent(item)

```
def getParent(node : Node,
              val : int,
              parent : int) -> None:
    if (node is None):
        return
    if (node.data == val):
        print(parent)
    else:
        getParent(node.left,
                  val, node.data)
        getParent(node.right,
                  val, node.data)
```

This method will return the parent value or parent node of tree. This function is a recursive function used to find the parent of a given node. If the current node is the required node, it will return it and print out the

node. Otherwise, it will recursively call for the child nodes or child tree and make the current node its new parent node.

2.3.9. getChild(item)

```
def numberOfChildren( root, x):
    
    numChildren = 0

    if (root == None):
        return 0
    q = []
    q.append(root)

    while (len(q) > 0) :
        n = len(q)
        while (n > 0):
            p = q[0]
            q.pop(0)
            if (p.key == x) :
                numChildren = numChildren + len(p.child)
                return numChildren
            
            i = 0
            
            while ( i < len(p.child)):
                q.append(p.child[i])
                i = i + 1
            n = n - 1

    return numChildren
```

This Method will return the child value of the item. This function works to calculate the number of child nodes of a given node. It will first initialize the number as zero and will also return zero if it checks the tree does not exist. It will start a queue and appends the root. If the node has child nodes, it will dequeue the item from the queue and check if it is equal to the given node and if yes it will return the number of children. As each child nodes returns the number, the initialized number of children will be added by one until all child nodes are accounted for.

3. Directed Graph

3.1. Definition

A graph consists of a set of objects called vertices that are connected in pairs through connections called edges. The purpose of a graph is to represent relationships between two objects. A directed graph is a type of

graph where each edge has arrows directing form predecessor vertex to the successor vertex now called arcs.

3.2. Directed Graph vs Undirected Graph

Compared to its Directed counterpart, Edges in an Undirected Graph have no direction or no arrows hence, they are not called arcs. This means traversal between the two nodes have no order enabling travel from and to any vertex of the pair of vertices connected by edge possible. For example, in an undirected graph, a pairwise edge connection between vertices A and B would allow traversal from A to B as well as B to A.

3.3. Methods

3.3.1. Add(items)

```
def Add(self,item):
    self.numVertices = self.numVertices + 1
    newVertex = Vertex(item)
    self.vertList[item] = newVertex
    return newVertex
```

This method adds items into the graph as vertices. Every single time this method is called self.numVertices or the number of vertices in the graph increases by 1. A variable newVertex contains the item stored as a Vertex allowing it to keep track of its own connections with other vertices in the graph. The newly made vertex is then stored in self.vertList dictionary with the item value as its key.

3.3.2. AddEdges(item1, item2)

```
def AddEdges(self,item1,item2):
    if item1 not in self.vertList:
        nv = self.Add(item1)
    if item2 not in self.vertList:
        nv = self.Add(item2)
    self.vertList[item1].addNeighbor(self.vertList[item2])
```

This method creates an edge between two vertices in the graph. If the any of the specified vertices does not exist then, they are created by calling the Add function on the non-existent item.

The connection is made by accessing item 1 vertex in the self.vertlist dictionary and using .addNeighbor method to update its connections to include item 2 vertex.

3.3.3. Search(item)

```
def Search(self,item):
    if item in self.vertList:
        return print(True)
    else:
        return print(False)
```

This method will return ‘True’ if the item is found in the graph as a vertex. Otherwise, ‘False’ is returned. As stated before, when items are added they are stored in self. vertList as a vertex with the item value as the key. This means it contains all vertices of the graph, so it is logical to search for the item there.

3.3.4. getEdges(m)

```
def getEdges(self,m):
    connectedtolist=[]
    for i in self.vertList[m].getConnections():
        connectedtolist.append(i.getId())
    print(m,'is connected to',connectedtolist)
```

This method lists down all vertices connected to vertex m. When called, all vertices in ‘self. connectedTo’ of vertex m is iterated one by one to retrieve their ids and append them to the ‘connectedtolist’ list. The list holds all the values the vertex m is connected to so is printed for the user to see.

3.3.5. Size ()

```
def Size(self):
    return print(self.numVertices)
```

Self.numVertices is created when graph class is instantiated. Its purpose is to keep count of the number vertices added to the graph object. Hence, to retrieve the number nodes in the graph(size) is to return a print of self. numVertices.

3.3.6. isPath(m, n)

```
def isPath(self,m,n):
    visited = [False] * self.numVertices
    queue= []
    queue.append(m)
    visited[m] = True

    while queue:
        m = queue.pop(0)
        if m==n:
            return print(True)
        else:
            for i in self.vertList[m].getConnections():
                if visited[i.getId()] == False:
                    queue.append(i.getId())
                    visited[i.getId()] = True
    return print(False)
```

This method checks if there is a path between vertices m and n. It returns true if a path is found or false if not found. It uses breadth first search (BFS), a traversal method for graphs that prioritizes visiting neighboring nodes. It first creates a list called 'visited' to mark all the vertices as unvisited (False). Another list called 'queue' is made to hold all nodes to be visited. When a node is visited, its value in the 'visited' list is changed from 'False' to 'True' to mark it as visited and is dequeued from the queue. This is done for the start node but for its neighboring nodes they are visited where the neighboring nodes of those nodes are also visited until there are no more unvisited nodes to visit or a node that matches n is visited. Visiting n means there is a path to it from m so the method would return 'True' otherwise, it would return 'False'.

3.3.7. findAllPath(m, n)

```
def findAllPathsUtil(self, m, n, visited, path):
    visited[m]= True
    path.append(m)
    if m == n:
        print (path,end=" ")
    else:
        for i in self.vertList[m].getConnections():
            if visited[i.getId()]== False:
                self.findAllPathsUtil(i.getId(), n, visited, path
    path.pop()
    visited[m]= False

def findAllpath(self,m,n):
    visited =[False]*(self.numVertices)
    path = []
    self.findAllPathsUtil(m, n, visited, path)
```

This method returns all paths from m to n. It uses depth first search (DFS) which is a traversal method where it prioritizes visiting new nodes instead of adjacent nodes (BFS). Similar to isPath() Running findAllpath() would create a new list containing False objects to represent each unvisited node called ‘visited’. However, instead of creating a queue next, a new list called ‘path’ is created to be used as a stack list. Since DFS uses recursion, the recursive function ‘findAllPathUtil’ is called whose purpose is to stack each visited node into the ‘path’ list. It happens within the findAllpath() itself with m, n, visited list and path list as its arguments, carrying them over. Once called, the start node is marked as visited and added to the path list. Next, an if-else statement is executed where unless the currently visited node m matches the destination node n then, a neighboring node of m is visited and also experiences the same check. This continues on until node value n is visited.

3.3.8. isCyclic ()

•

```
def isCyclicUtil(self, v, visited, recStack):
    visited[v] = True
    recStack[v] = True

    for neighbour in self.vertList[v].getConnections():
        if visited[neighbour.getId()] == False:
            if self.isCyclicUtil(neighbour.getId(), visited, recStack) == Tru
                return True
        elif recStack[neighbour.getId()] == True:
            return True

    recStack[v] = False
    return False

def isCyclic(self):
    visited = [False] * (self.numVertices + 1)
    recStack = [False] * (self.numVertices + 1)
    for node in range(self.numVertices):
        if visited[node] == False:
            if self.isCyclicUtil(node,visited,recStack) == True:
                print("")
                return print(True)
    print("")
    return print(False)
```

This method will return true if there is a cycle in the graph. If there is a path with at least one edge that leads from the start node to itself, that would mean there is a cycle within the graph. This method also uses DFS so would require its own recursive function.

When isCyclic() is called, each vertex is marked as unvisited and not added yet to the stack. As per usual each node in the graph is visited using the isCyclicUtil() recursive function where visiting them would mark them as visited and added to the stack list but under the condition that they have not been visited yet. If it has already been visited, as in visited[neighbour.getId()] is not equal to false, an elif statement is made asking if the has already been added to the stack. If it was already

added, then the recursive function would return true for a cycle. Otherwise, it would return False for a cycle in the graph.

3.3.9. allConnected ()

```
def DFSUtil(self, temp, v, visited):
    visited[v] = True

    temp.append(v)

    for i in self.vertList[v].getConnections():
        if visited[i.getId()] == False:

            temp = self.DFSUtil(temp, i.getId(), visited)
    return temp

def allConnected(self):
    visited = []
    cc = []
    for i in range(self.numVertices):
        visited.append(False)
    for v in range(self.numVertices):
        if visited[v] == False:
            temp = []
            cc.append(self.DFSUtil(temp, v, visited))
    if len(cc)!=1:
        print("")
        return print(False)
    else:
        print("")
        return print(True)
```

This method returns True if all nodes are connected or False if there is at least one that is unconnected. This method also uses DFS so a recursive function called DFSUtil() is used to visit each node and they

are added to temp as a single path. Once each node has been visited all paths recorded in temp is appended to the cc list. An if-else statement is used on the list where if is more than 1 paths in cc list then the method allconnected() would return False for all nodes being connected. Else, it would return True for all nodes being connected.

4. Conclusions

Binary search tree and directed graph are far more complicated than their basic variants. Both utilizing various conditions and complex traversal methods such as Breath-first or Depth-first. Such complexity is necessary for this data structure to achieve great feats in its application. For example, binary search tree may be used in managing large libraries of data due to its nature of fast insertion removal and retrieval of items (baeldung, 2021). An example for Directed graphs is its use in analyzing social relationships, electric circuits, networks and much more. Therefore, there are limitless possibilities for what they can used for.

References

baeldung. (2021, August 25). *A Quick Guide to Binary Search Trees*. Retrieved from Baeldung: https://www.baeldung.com/cs/binary-search-trees#:~:text=Simply%20put%2C%20a%20binary%20search,iterate%20them%20in%20sorted%20order.

Geek For Geeks. (2022, April 22). *Binary Search Tree Data Structure*. Retrieved from Geek For Geeks: https://www.geeksforgeeks.org/binary-search-tree-data-structure/?ref=shm

Geek For Geeks. (2022, February 01). *graph and its representations*. Retrieved from Geek For Geeks: https://www.geeksforgeeks.org/graph-and-its-representations/?ref=lbp