# 210CT Coursework 2

Contents

[210CT Coursework 2 1](#_Toc437791562)

[Task 1 -Graph Building 2](#_Toc437791563)

[Pseudo Code 2](#_Toc437791564)

[Python Code 2](#_Toc437791565)

[Adding Weighted vertices 3](#_Toc437791566)

[Task 2 - Graph Traversal 4](#_Toc437791567)

[Python Code 4](#_Toc437791568)

[Task 3 - Binary Tree Search 5](#_Toc437791569)

[Python Code 5](#_Toc437791570)

[Task 4 - Binary Tree Node Delete 6](#_Toc437791571)

[Python Code 6](#_Toc437791572)

[Task 5 - Signal Processor 7](#_Toc437791573)

[Python Code 7](#_Toc437791574)

[Bibliography 9](#_Toc437791575)

## Task 1 -Graph Building

### Pseudo Code

1. #This code uses adjacency list to created a unweighted graph
2. listOfNodes ← []
4. CLASS NODES()
5. #This function adds a node to the list of nodes
6. ADDNODE(nodeToAdd)
7. listOfNodes.append(nodeToAdd)
8. FOR i IN nodeToAdd
9. listOfNodes[i].append(length of listOfNodes)
11. #This function adds a edge (connections to a new node) to a node
12. ADDEDGE(firstNode,secondNode)
13. listOfNodes[firstNode].append(secondNode)
14. listOfNodes[secondNode].append(firstNode)

### Python Code

For the structure of the graph I decided to create an adjacency list by using a list of lists. I used this approach because it would make the addNode and addEdge functions easier to create since manipulating multidimensional lists are easy to navigate­. For the adding of a new node the code appends new node to the list of lists, then it updates the old nodes with the connection to the new node. For the adding of a new edge, it adds the edge to the second node to the first node and then the edge to the first node to the second node. The output method simply outputs the list of nodes in a readably format.

1. # nodeList is a adjacency list of nodes and edges that connect the nodes
2. nodeList = [[1,2,3],[0,2,4],[0,1,3,4],[0,2,4],[1,2,3,5],[4]]
3. """
4. the nodes class is what operates on the nodes
5. """
6. **class** nodes():
7. """
8. addNode will add new node the the end of nodeList, it will then update the nodes that the new node is connected to with the new edges. It must be
9. """
10. **def** addNode(self,nodeConnections):
11. nodeList.append(nodeConnections)# appends the connections(edges) of the new node to the node list
12. **for** self.i **in** nodeConnections:# loops though the new connections
13. nodeList[self.i].append(len(nodeList)-1) #for the connections in the new node it adds the corresoponding connections to the old nodes
14. """
15. addEdge will add ned connections/edges to alreay existing nodes, you hand it the 2 nodes that you would like a new edge between and it will update them
16. """
17. **def** addEdge(self,nodeFrom,nodeTo):
18. nodeList[nodeFrom].append(nodeTo) # append the new connection in 1st node
19. nodeList[nodeTo].append(nodeFrom) # append the new connection in 2nd node
20. """
21. This simply outputs the nodeList so I can check that its has been updated correctly
22. """
23. **def** output(self):
24. self.a = 0 # assigns a to zero
25. **for** self.i **in** nodeList: # loops through list of node
26. **print** "Node",self.a,":", self.i # print the list number and the value in the list
27. self.a += 1 # increments a
28. #return nodeList
29. b = nodes()
30. b.addNode([0,5])
31. b.addEdge(3,1)
32. b.output()

### Adding Weighted vertices

A weighted graph is a graph where each of edge has a numerical value (weight) assigned to it (Weisstein, 2015), this allows for shortest distance/rout algorithms to be performed on the graph. An example of this is Dijkstra’s algorithm (Auckland University, 1998). Dijkstra’s algorithm is a commonly known and used algorithm that works by follow these steps:

* Label the starting node with 0
* Choose the edge with the smallest value
* Temporarily label all the nodes connection to the permanent labelled nodes with the distance from the start point
* Choose the temporary label with the smallest value
* Repeat until steps until arrival at destination node
* Retrace the shortest route back through the network to the starting node

(WikiHow, 2015)

To add weighted graph to my code I would change the list of lists to a list of lists of dictionaries, this way I would have a value for the node it is and the edge weight. Here is what the list of lists used in my code would be changed to act as I am suggesting:

1. weightedGraph =[
2. [
3. {"node":1,"edgeWeight":3},
4. {"node":2,"edgeWeight":4},
5. {"node":3,"edgeWeight":5}
6. ],
7. [
8. {"node":0,"edgeWeight":3},
9. {"node":2,"edgeWeight":6},
10. {"node":4,"edgeWeight":2}
11. ],
12. [
13. {"node":0,"edgeWeight":4},
14. {"node":1,"edgeWeight":6},
15. {"node":3,"edgeWeight":4},
16. {"node":4,"edgeWeight":7}
17. ],
18. [
19. {"node":0,"edgeWeight":5},
20. {"node":2,"edgeWeight":4},
21. {"node":4,"edgeWeight":8}
22. ],
23. [
24. {"node":1,"edgeWeight":2},
25. {"node":2,"edgeWeight":7},
26. {"node":3,"edgeWeight":8},
27. {"node":5,"edgeWeight":1}
28. ],
29. [
30. {"node":4,"edgeWeight":1}
31. ]
32. ]

Changing it to be like this would add the ability to find out what node the edge is connected to and what the weight of the edge is.

## Task 2 - Graph Traversal

For the graph traversal I decided to use a dictionary of sets instead of a list of lists (like in the graph building task). I decided to do this because the sets worked better with stack that is used with depth first search. (Python Sets, 2015)  
At first I tried to do it with the list of lists but I found it to be over complicated. The graph traversal works by being handed a dictionary of nodes and the starting node, a list is created to hold the path and stack is created by adding the set of the starting node. The while loop is looped until the stack is empty. The while loop contains an if that adds if the current node (current node found by popping it off the stack) is in the list of visited nodes. I used a simple join statement to output the lists returned by the graphTraverse method so the output was more readably

### Python Code

1. nodeDict = {0: set([1,3]),1: set([0]),2: set([5]),3: set([0,4]),4: set([3,5]),5: set([2,4])}

4. **class** nodes():
5. """
6. This method must be handed a graph in the form of a dictionary of sets and which node to start on.
7. """
8. **def** graphTraverse(self,nodeDict,startNode):
9. self.path = [] #empty list for the path to he added to
10. self.visitedNodes, self.stack = set(), [startNode]#created the stack and visited nodes list
11. **while** self.stack: #loops though the stack until it is empty (meaning all nodes are visited)
12. self.node = self.stack.pop()#makes the current node the top of the stack
13. **if** self.node **not** **in** self.visitedNodes: #if that node has not already been visited
14. self.visitedNodes.add(self.node) #adds the current node to the visited nodes
15. self.stack.extend(nodeDict[self.node] - self.visitedNodes)  #adds the edges in the node to the stack minus the alead visited nodes
16. self.path.append(self.node)  # add the current node to the path
17. **return** self.path,self.visitedNodes # returns the path and the nodes visited

20. (i,j) = nodes().graphTraverse(nodeDict,4) # calls the method with the parameters graph and starting node. makes it equal to i and j so the 2 items returned by the method can be accessed individually
21. **print** "Path used:",", ".join(str(a) **for** a **in** i) # prints out the path and removes square brackets
22. **print** "Nodes visited:",", ".join(str(a) **for** a **in** j) # prints out the visited nodes and removes the square brackets

## Task 3 - Binary Tree Search

For the binary tree search used a while loop to loop if the tree handed to the method was not empty, then if it finds the value it is looking for it returns true(in the first if), if it is smaller or larger it moves left or right. If it goes off the end of the tree the value will become none so “None” will be returned, using the same return that would have returned “None” if the tree was empty. If the tree is empty none of the if statements are stratified and there is no else so is goes straight to the return none.

### Python Code

1. **class** BinTreeNode(object):
3. **def** \_\_init\_\_(self, value):
4. self.value=value
5. self.left=None
6. self.right=None
8. **def** tree\_insert( tree, item):
9. **if** tree==None:
10. tree=BinTreeNode(item)
11. **else**:
12. **if**(item < tree.value):
13. **if**(tree.left==None):
14. tree.left=BinTreeNode(item)
15. **else**:
16. tree\_insert(tree.left,item)
17. **else**:
18. **if**(tree.right==None):
19. tree.right=BinTreeNode(item)
20. **else**:
21. tree\_insert(tree.right,item)
22. **return** tree
24. **def** bin\_tree\_find(tree,target):
25. **while** tree != None: # will loop while the is something in the tree
26. **if** tree.value == target: # if the value selected is the target then value is returned
27. **return** tree.value
28. **elif** tree.value > target: # if the value selected is more than the target then the value selected is moved to the left
29. tree = tree.left
30. **else**: #if anything else (aiming for selected value less than the target) the selected value is moved right
31. tree = tree.right
32. **return** None # returns none if cant be found or tree is empty
34. **def** postorder(tree):
35. **if**(tree.left!=None):
36. postorder(tree.left)
37. **if**(tree.right!=None):
38. postorder(tree.right)
39. **print** tree.value
41. **def** in\_order(tree):
42. **if**(tree.left!=None):
43. in\_order(tree.left)
44. **print** tree.value
45. **if**(tree.right!=None):
46. in\_order(tree.right)
48. **if** \_\_name\_\_ == '\_\_main\_\_':
49. t=tree\_insert(None,6);
50. tree\_insert(t,10)
51. tree\_insert(t,5)
52. tree\_insert(t,2)
53. tree\_insert(t,3)
54. tree\_insert(t,4)
55. tree\_insert(t,11)
56. in\_order(t)
57. **print** (bin\_tree\_find(t,4))

## Task 4 - Binary Tree Node Delete

This is a node delete algorithm, I have implemented the delete function into the code provided, the code works by handing the delete function the node that need to be deleted, if it is the head then head gets changed to the next node, if it is the tail it is changed to the previous node. If it is a node in the middle, it is changed to the previous or next node depending on where it is on the list and if the previous or next node are not empty.

### Python Code

1. **class** Node(object):
2. **def** \_\_init\_\_(self, value):
3. self.value=value
4. self.next=None
5. self.prev=None
7. **class** List(object):#inserts the given parameters to the list
8. **def** \_\_init\_\_(self):
9. self.head=None
10. self.tail=None
11. **def** insert(self,n,x):
12. **if** n !=None:
13. x.next=n.next
14. n.next=x
15. x.prev=n
16. **if** x.next!=None:
17. x.next.prev=x
18. **if** self.head==None:
19. self.head=self.tail=x
20. x.prev=x.next=None
21. **elif** self.tail==n:
22. self.tail=x
23. **def** delete(self,n): #this will delete a Node
24. **if** n.prev != None: #if previous n isnt empty then the previous next n will become the next n
25. n.prev.next = n.next
26. **else**: #if not head will become the next n
27. self.head = n.next
28. **if** n.next != None: #if next n is empty then the next previous n will become the previous n
29. n.next.prev = n.prev
30. **else**: #if not tail will become the previous n
31. self.tail = n.prev
32. **def** display(self): #outputs the list to the screen
33. values=[]
34. n=self.head
35. **while** n!=None:
36. values.append(str(n.value))
37. n=n.next
38. **print** "List: ",",".join(values)
40. **if** \_\_name\_\_ == '\_\_main\_\_':
41. l=List()
42. l.insert(None, Node(4))
43. l.insert(l.head,Node(6))
44. l.insert(l.head,Node(8))
45. l.display()
46. l.delete(l.head)
47. l.display()

## Task 5 - Signal Processor

For signal processor I used the math library to allowed me use sin so in the genSeries method no matter what values are inputted by the incrementor the values returned will be between -1 and 1 (Python Math, 2015). The list of the series of numbers (self.series) and the integer (self.serieslength) that defines the length of the series are created in the constructor, this allows them to used by the other methods. The series generation method (genSereis) works by having a value that is incremented by 10 for defined length of the series (self.serieslength), each loop the incremented value is put through sin to create a value between -1 and 1. That value is then appended to a list that is returned by the method. The value change (valchange) method gets a series (self.series) from the genSeries method then uses a lambda function make all the values positive so the wave produced is all in the positive rather than going in and out of negative and positive like the series normally does. Using a lambda function allows me to create a small function that will only be used in that one place without having to create a full defined function (Sahu, 2014). The adjusted values are then appended to a list that is returned by the method. The display method works by being handed a list, then if the value is positive it fills half the scale with white space until the mid-point then it will but the number of hashes that the value is. If it is negative it fill with white space up to where the hashes should start, where hashes start is calculated by adding the value in the list (which will be negative) to the scale.

### Python Code

1. **import** math
2. **class** numStore():
4. """
5. self iniateing method that creates the series list and the length of the series list, it must be handed the lengh of the series as a integer
6. """
7. **def** \_\_init\_\_(self,seriesLength):
8. self.series = [] #list the other methods use for the series of numbers
9. self.seriesLength = seriesLength #defines the length of the series from the parameter that is handed to the method
11. """
12. This method generates a series of numbers between -1 and 1 using sin. It will the the length that is set in the constructor
13. """
14. **def** genSeries(self):
15. self.incrementor = 10 #screates the value that will be incremented
16. **for** self.i **in** range (0,self.seriesLength): #loops for the required length of the series of numbers
17. self.series.append(math.sin(math.radians(self.incrementor)))# appends a sin value between -1 and 1
18. self.incrementor += 10 # increments the value the sin uses by 10
19. **return** self.series #returns the series
21. """
22. This method will display a list(what the other functuons return) is a readable format (lines of hashes)
23. """
24. **def** display(self,listToOutput):
25. self.scale = 30 # this sets the scale for the graph
26. **for** self.i **in** listToOutput:#loops for the length of the list handed to the method
27. self.j = int(self.i\*self.scale) # changes the values to ints from a floats to integers
28. self.str = "" #creates the string to fill with spaces or #
29. **if** self.j > 0: # if the value is value is positive
30. **for** self.a **in** range (0,self.scale): #for 30 spaces a white space will be created
31. self.str = self.str + " "
32. **for** self.a **in** range(0,self.j): # for the length of the value hashs will be printed
33. self.str = self.str + "#"
34. **else**: # if its is a negative number
35. **for** self.a **in** range (0,self.j+self.scale): #for the empty space print white space
36. self.str = self.str + " "
37. **for** self.a **in** range(0,-self.j): # for the remaining space between white space created and the mid point, print hashs
38. self.str = self.str + "#"
39. **print** self.str #print the line of hashs produced
40. """
41. This method get a series of numbers from genSeries() and then alters them using a lambda so they are all in the positive region
42. """
43. **def** valChanger(self):
44. self.listToReturn =[] #creates the list that will be returned
45. self.values = numStore(self.seriesLength).genSeries() #gets a series of values from the genSeries
46. f = **lambda** x : x+1 **if** x<=1 **else** x #lambda that makes adds 1 to any value less than 1 which will create a wave only in positive
47. **for** self.i **in** self.series: #for the length of the series
48. self.listToReturn.append(f(self.i)) # appends the value that is changed by the lambda to the list that is returned
49. **return** self.listToReturn #returns the list
51. a = numStore(100) #makes the class equal to a, send 10 as the series length
52. a.display(a.genSeries()) # displays what is returned by genSeries()
53. **print** " "
54. a.display(a.valChanger())# displays what is returned by valChanger()

# Bibliography

Auckland University, 1998. *Data Structures and Algorithms 10.2 Dijkstra's Algorithm.* [Online]   
Available at: https://www.cs.auckland.ac.nz/software/AlgAnim/dijkstra.html  
[Accessed 8 12 2015].

Python Math, 2015. *Python Documentation 9.2. math.* [Online]   
Available at: https://docs.python.org/2/library/math.html  
[Accessed 10 12 2015].

Python Random, 2015. *Python Documentation Random 9.6.* [Online]   
Available at: https://docs.python.org/2/library/random.html  
[Accessed 7 12 2015].

Python Sets, 2015. *Python Documentation Sets 8.7.* [Online]   
Available at: https://docs.python.org/2/library/sets.html  
[Accessed 5 12 2015].

Sahu, J., 2014. *Lambda Expressions ine Java 8: Why and How to Use Them.* [Online]   
Available at: http://www.nagarro.com/de/de/blog/post/26/Lambda-Expressions-in-Java-8-Why-and-How-to-Use-Them  
[Accessed 9 12 2015].

Weisstein, E. W., 2015. *Wolfram Mathworld Weighted Graph.* [Online]   
Available at: http://mathworld.wolfram.com/WeightedGraph.html  
[Accessed 5 12 2015].

WikiHow, 2015. *How to Use Dijkstra's Algorithm.* [Online]   
Available at: http://www.wikihow.com/Use-Dijkstra's-Algorithm  
[Accessed 9 12 2015].