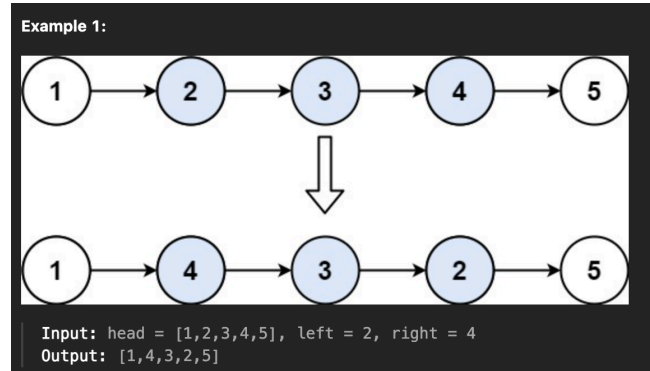


COMP 352 Algorithm and solved Problems

Problem 1:(Linked Lists)

You are given the head of a singly linked list and two integers one called “left” and the other called “right” such that $\text{left} \leq \text{right}$. Using this information, reverse the list nodes from position left to position right and return the linked list.



Example 2:

Input: head = [5], left = 1, right = 1
Output: [5]

Constraints:

- The number of nodes in the list is n .
- $1 \leq n \leq 500$
- $-500 \leq \text{Node.val} \leq 500$
- $1 \leq \text{left} \leq \text{right} \leq n$

SOLUTION: The solutions are written in Python which is quite similar to writing a pseudo-code. The idea is to create a Queue (first in last out) and traverse the linked list until the index = left. When that happens we enqueue the values in the Queue technically $O(1)$ but insert is $O(n)$. After that, we loop through the queue and place every value from that position (left to right). Lastly, return the linked list.

```
def reverseBetween(self, left: int, right: int):
    if not self.head: # if the head of the linked list is already null
        return self.head # return the null value
    else:
        values = []
        current = self.head
        position = None
        index = 1 # index set as identifier for left and right

        while current != None: # Traverse the linked list
            if index == left:
                position = current # store the reference in position
                while index <= right:
                    values.insert(0, current.val) # insert at index 0 => O(n)
                    current = current.next
                    index += 1
                for value in values: # once the array is filled
                    position.val = value # we reverse the values in each node
                    position = position.next
            if current:
                current = current.next
            index += 1

        return self.head # return the head of the linked list
```

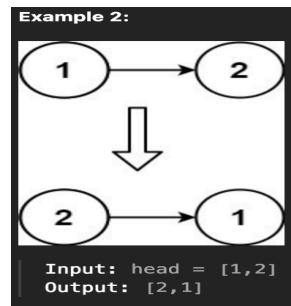
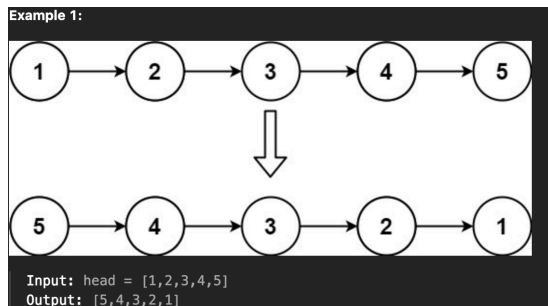
Time complexity is $O(n)$ because we have a linked list of n nodes and traverse it from head to end.

Space complexity is also $O(n)$ because of the queue (array) we created that could have n value

COMP 352 Algorithm and solved Problems

Problem 2:(Linked Lists)

You are given the head of a singly linked list, reverse the linkedList and return it



SOLUTION:

The idea is to either iterate through the linked list and put the elements in a stack then start popping them while changing the elements of the linked list or you could use recursion.

We create two sub methods `traverseList(head)` which fills the stack with $O(1)$ (push) and its base case is when the head is None.

Then create another sub method called `reverseList(head)` which traverses the linked list and changes the values with the popped element from the stack.

Time complexity $O(n+n) = O(n)$
Space Complexity $O(n)$ [Stack]

```
def reverseList(self):  
    stack = []  
    def traverseList(head:ListNode):  
        if head is None:  
            return None  
        else:  
            stack.append(head.val)  
            return traverseList(head.next) #O(n)  
    traverseList(self.head)  
    def reverse(head: ListNode):  
        if head is None:  
            return None  
        else:  
            if len(stack)<0:  
                return head  
            else:  
                head.val = stack.pop()  
                return reverse(head.next)  
    reverse(self.head) #O(n)  
    return self.head
```

COMP 352 Algorithm and solved Problems

Problem 3: (Linked Lists)

Given the head of a singly linked list, return true if it is a Palindrome or false otherwise.
A palindrome is a sequence that reads the same forward and backward.

Example 1:



Input: head = [1,2,2,1]

Output: true

Example 2:



Input: head = [1,2]

Output: false

SOLUTION:

The idea is to store these values of every linked list in an array and then verify if it is palindrome. Because we can't access the nodes by index its better to do so with an array.

The first method
traverse(head) accumulate the values of every node and puts them in an array

The second method operates at $O(n/2)$
It checks the lower bound and upper and does the comparison then +1 for lower and -1 for upper until lower > upper

Time Complexity : $O(n)$

Space Complexity: $O(n)$

```
def isPalindrome(self) ->bool:
    arr = [] #Space  $O(n)$ 

    def traverse(head: ListNode)->None:
        #Recursively traverse the linked list
        if head is None:
            return None
        else:
            arr.append(head.val) #  $O(1)$  adding to the array
            return traverse(head.next)

    traverse(self.head)
    if arr==[]:
        return True
    def isPalindrome(arr: list, lower: int, upper:int) ->bool:
        #Operate at  $O(n/2)$  and validate
        if lower>upper:
            return True
        if arr[lower]!= arr[upper]:
            return False
        return isPalindrome(arr, lower+1, upper-1)

    return isPalindrome(arr, 0, len(arr)-1)
```

COMP 352 Algorithm and solved Problems

Problem 4: (Linked Lists)

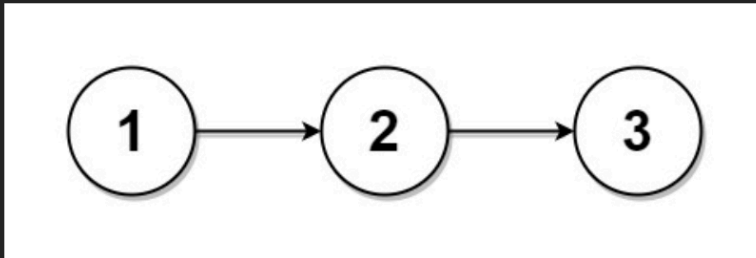
Given the head of a singly linked list and an integer k, split the linked list into k consecutive linked list parts.

The length of each part should be as equal as possible: no two parts should have a size differing by more than one. This may lead to some parts being null.

The parts should be in the order of occurrence in the input list, and parts occurring earlier should always have a size greater than or equal to parts occurring later.

Return an array of the k parts.

Example 1:



Input: head = [1,2,3], k = 5

Output: [[1],[2],[3],[],[]]

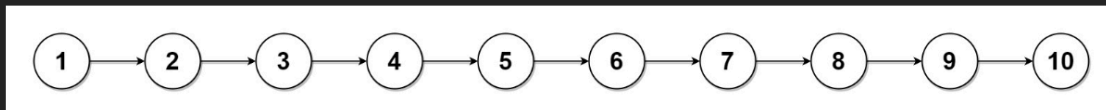
Explanation:

The first element output[0] has output[0].val = 1, output[0].next = null.

The last element output[4] is null, but its string representation as a ListNode is [].

Description: FYI, [1] is the node with value 1 and next Null so the array contains the nodes And [] indicates null value

Example 2:



Input: head = [1,2,3,4,5,6,7,8,9,10], k = 3

Output: [[1,2,3,4],[5,6,7],[8,9,10]]

Explanation:

The input has been split into consecutive parts with size difference at most 1, and earlier parts are a larger size than the later parts.

Description: [1,2,3,4] indicates a linked list with head value 1, next 2, next 3, next 4, next Null

COMP 352 Algorithm and solved Problems

Idea:

Think of using $N // k$ and $N \% k$ where N is the length of the linked list

Basically $N // k$ indicates the number of elements that every node in the array contains and $N \% k$ indicates the number of elements in the array with an extra node.

Example:

head = [1,2,3], $k = 5$, $N = 3 \Rightarrow 3 // 5 = 0$ and $3 \% 5 = 3$ Therefore every element of the array should be null except the first three elements of the array that have 1+0 node

In this case the result is [[1] , [2], [3], [], []]

SOLUTION:

The solution involves first storing the values of every node in an array called group.

This part is the easiest one to do (You probably have a good idea for it if you did at least 1 or 2 problems so far). Just a few simple steps

```
result = [] # Space complexity  $O(k)$  or  $O(n)$  depending on what is greater k or n
group = [] # Space complexity  $O(n)$ 
current = self.head

while current != None:
    group.append(current.val)
    current = current.next #  $O(n)$ 

N = len(group)
num_of_element_per_list = N // k # Number of elements in the list except the first part
num_of_added_lists = N % k # Number of lists with additional element
```

In the next part, we have two cases to cover when $k > N$ and when $k < N$. If you have noticed that when $k > N$ then $N \% k = 0$ (always true and we will take advantage of that)

In case $k > N$: traverse the linkedList and as you do create separate Nodes with the values and append them to the array result and for whatever is left keep adding null from 0 to $k - N$

In case $k < N$: We're going to have to handle the $N \% k$ first. Traversing the linked list again, if $N \% k > 0$ then using a while loop to create a chain of nodes, appending them to the array result and decrement the number $N \% k$. Once that's done, move to $N // k$ Using the same principle.

COMP 352 Algorithm and solved Problems

Full source code:

```
def splitListToParts(self, k:int)->list[list]:
    result = [] # Space complexity  $O(k)$  or  $O(n)$  depending on what is greater k or n
    group = [] # Space complexity  $O(n)$ 
    current = self.head

    while current!=None:
        group.append(current.val)
        current = current.next # $O(n)$ 

    N = len(group)
    num_of_element_per_list = N // k # Number of elements in the list except the first part
    num_of_added_lists= N % k # Number of lists with additional element

    current = self.head
    if k>N:
        while current!=None:
            node = ListNode(val=current.val)
            result.append(node)
            current = current.next
        for i in range(0, k-N):
            result.append(None)
    else:
        while current!=None:
            if num_of_added_lists>0:
                while num_of_added_lists>0:
                    node = ListNode(val = current.val)
                    temp = node
                    for i in range(1, num_of_element_per_list+1):
                        current = current.next
                        temp.next = ListNode(val = current.val)
                        temp = temp.next
                    result.append(node)
                    num_of_added_lists-=1
                    current = current.next

                if num_of_element_per_list>0:
                    node = ListNode(val = current.val)
                    temp= node
                    for i in range(1, num_of_element_per_list):
                        current = current.next
                        temp.next = ListNode(val = current.val )
                        temp = temp.next
                    result.append(node)
                else:
                    result.append(None)

            current= current.next

    return result
```

COMP 352 Algorithm and solved Problems

Problem 5: (Binary Trees)

You are given the root of a binary tree. Determine if it is a valid binary search Tree (BST)

Recall the definition of a binary search tree:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

SOLUTION:

Time Complexity is $O(n)$ and Space Complexity is $O(h)$ where h is the height of the tree

```
def isValidBST(self):
    #Sub method
    def traverse(node: TreeNode, low: float, heigh: float) ->bool:
        if node is None:
            return True
        if not (low < node.val < heigh):
            return False # base case
        return traverse(node.left, low, node.val) and traverse(node.right, node.val, heigh)
    return traverse(self.root, float('-inf'), float('+inf'))
```

Problem 6: (Trees)

You are given the root of a binary tree. Write a pseudoCode for pre, inOrder and post traversal and accumulate the values in an array

```
def preOrderTraverse(self):
    result = []
    def traverse(node: TreeNode):
        if node is None:
            return
        result.append(node.val)
        traverse(node.left)
        traverse(node.right)
    traverse(self.root)
    return result
```

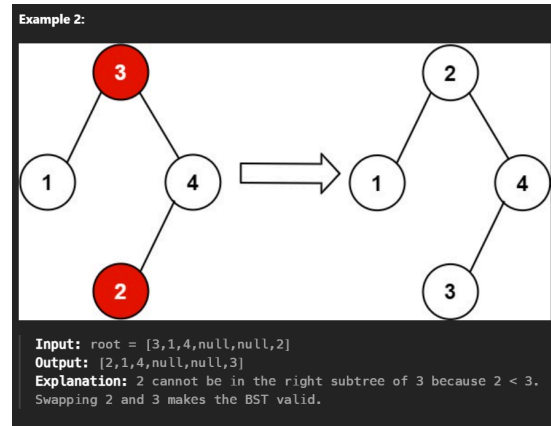
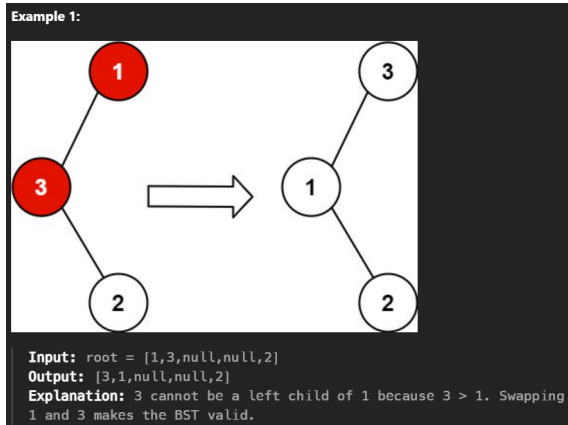
```
def inOrderTraverse(self):
    result = []
    def traverse(node: TreeNode):
        if node is None:
            return
        traverse(node.left)
        result.append(node.val)
        traverse(node.right)
    traverse(self.root)
    return result
```

```
def postOrderTraverse(self):
    result = []
    def traverse(node: TreeNode):
        if node is None:
            return
        traverse(node.left)
        traverse(node.right)
        result.append(node.val)
    traverse(self.root)
    return result
```

COMP 352 Algorithm and solved Problems

Problem 6: (BST)

You are given the root of a binary search tree (BST), where the values of exactly two nodes of the tree were swapped by mistake. Recover the tree without changing its structure.



Idea: Think of using a traversal you know (preOrder, inOrder, postOrder) and figure out the reason why to do so.

SOLUTION:

The idea here is to use an ADT that can act as both stack and queue (double-ended queue) We want to traverse the Binary tree in inOrderTraversal and push the values to the double-ended queue.

Then Sort the array in ascending order.

Lastly traverse in inOrderTraversal and deque every element while updating the value of every node in the tree.

Space Complexity $O(n)$ because of the double ended queue

Time Complexity $O(n)$ traversing n nodes in the tree.

```
def recoverTree(self):
    arr = [] #Space of  $O(n)$ 

    def inOrderTraverse(node: TreeNode) ->None:
        if node is None:
            return None
        inOrderTraverse(node.left)
        arr.append(node.val) #  $O(1)$ 
        inOrderTraverse(node.right)

    inOrderTraverse(self.root)
    arr = sorted(arr)

    def recoverTree(node:TreeNode, lower:int) ->None:
        if node:
            recoverTree(node.left)
            node.val = arr.pop(0) # popping the first element from the array
            recoverTree(node.right)

    recoverTree(self.root)
```


COMP 352 Algorithm and solved Problems

Problem 7: (Linked Lists)

Given the head of a linked list, you are asked to remove the nth node from the end of the linked list and return its head.

Example 1:

Given the following linked list:

[1] ⇒ [2] ⇒ [3] ⇒ [4] ⇒ [5] ⇒ NULL and n = 2

Output: [1] ⇒ [2] ⇒ [3] ⇒ [5] ⇒ NULL (go backwards from the end of the linked list 2 steps, land on 4 and remove it)

Example 2:

Given the following linked list:

[1] ⇒ Null and n = 1

Output: Null

Example 3:

Given the following linked list:

[1] ⇒ [2] ⇒ Null and n = 2

Output : [2] ⇒ Null

Idea:

Can you do this with a space complexity of $O(1)$ and time complexity of $O(n)$?

Remember that in a singly linked list, you can only go from head to tail. You don't have the ability to traverse from the end of the list to the head.

SOLUTION:

Because we only can do one traverse from head to the end of the linked list, think of getting the number of nodes and store them in N.

N: length of linked list

Num_of_traverse = N - n (number of traverse from left to right)

Case 1: N-n<=0

If N-n ==0 it means remove head and make the new head, head.next
Otherwise return null

Case 2: N-n>0: traverse and stop one node before the one to delete and check if the next and next.next are not null then proceed.

```
def removeNthFromEnd(self, n:int):
    N = 0
    current= self.head
    while current!=None:
        N+=1
        current = current.next

    num_of_traverse = N-n

    if num_of_traverse<=0 :
        if N == n:
            self.head = self.head.next
            return self.head
        return None
    else:
        current = self.head
        while num_of_traverse>1:
            current= current.next
            num_of_traverse-=1

        if current.next and current.next.next:
            current.next = current.next.next
        else:
            current.next = None
        return self.head
```

COMP 352 Algorithm and solved Problems

Problem 8: (Binary Trees)

Given the root of a binary tree, you are asked to find the maximum depth of a binary tree. The Maximum depth is the number of nodes along from the root of the tree down to the farthest leaf.

Example:

root = [3,9,20,null,null,15,7]
Output = 3

Input: root = [1,null,2]
Output: 2

Idea:

Perhaps draw the trees given the arrays of input and figure out what you need to do to get the maximum depth

SOLUTION:

I used a sub Function but its really not that necessary
If you were to write a pseudo code then:

```
FUNCTION maxDepth( root: TreeNode) ->Integer:  
    IF root is null then  
        Return 0 // setting this to 0 is important because you need to calculate  
    Return max( maxDepth(root.left) +1, maxDepth(root.right) +1)  
END OF FUNCTION
```

```
def maxDepth(self) ->int:  
    def traverse(node: TreeNode)->int:  
        if node is None:  
            return 0  
        return max(traverse(node.left) +1, traverse(node.right)+1)  
    return traverse(self.root)
```

Explanation:

Let's consider this root = [3,9,20,null,null,15,7]

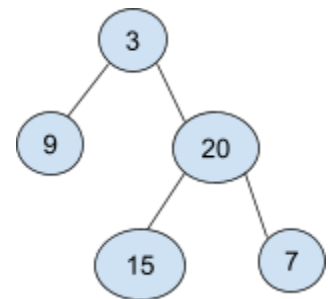
When calling **maxDepth**(root)

- First it checks if the node is null (FALSE)
- It traverses root.left and root.right: **Return max(maxDepth(root.left) +1, maxDepth(root.right) +1)**

Root.left = 9 and then root.left (9) = null => returns 0

And at the same time it checks root.right (9) == null => return 0

So from the left side we have obtain the value 0 and it does the same thing on the right side until we are left at the end with max(2,3) = 3



COMP 352 Algorithm and solved Problems

Problem 9: (Binary Trees)

Given the root of a binary tree, find the minimum depth and return it as an integer.

Example:

root = [3,9,20,null,null,15,7]
Output = 2

root = [2,null,3,null,4,null,5,null,6]
output = 4

Idea:

If you have already solved problem 8, you might be thinking of using the same algorithm and changing the max to min. That would be a correct idea, but what about the case where the tree looks like a chain of nodes (similar to a linked list). The minimum depth is the length in that case.

SOLUTION :

Algorithm: Time complexity is $O(n)$ and space is $O(h)$ where h is the height of the tree

FUNCTION minDepth(root: TreeNode) ->Integer:

IF root is null then

Return 0 // setting this to 0 is important because you need to calculate

IF root.left is null then

Return minDepth(root.right) +1

ELSE IF root.right is null then

Return minDepth(root.left)+1

ELSE

Return min(maxDepth(root.left) +1, maxDepth(root.right) +1)

END OF FUNCTION

```
def minDepth(self) ->int:
    def traverse(node:TreeNode) ->int:
        if node is None:
            return 0

        if node.left is None:
            return traverse(node.right)+1
        elif node.right is None:
            return traverse(node.left)+1
        else:
            return min(traverse(node.left)+1, traverse(node.right)+1)

    return traverse(self.root)
```

COMP 352 Algorithm and solved Problems

Problem 10: (Binary Trees)

You are given the root of a binary tree, perform a level order traversal and return it as an array of arrays of values.

Recall that a level order traversal is the same as Breadth-First-search (BFS)

Example:

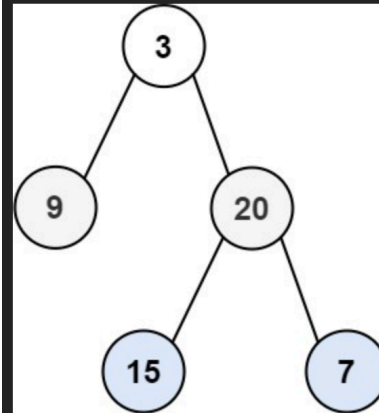
Root = [3,9,20,null,null,15,7]

output = [[3],[9,20],[15,7]]

Idea:

Think of creating a sub method that does a recursive call and pushes the elements into an array while it does it's traversing

Example 1:



Input: root = [3,9,20,null,null,15,7]
Output: [[3], [9,20], [15,7]]

SOLUTION:

If you were to write the Pseudo code:

FUNCTION breadthFirstSearch(root: Root of Tree) -> List[List[Integers]]

result = [] // Assign an empty array

FUNCTION traverse(node: TreeNode, level: Integer)

IF node is null Then

Return nothing

IF length(result) <= level:

result.append([]) // append empty array

result[level].append(node.value)

traverse(node.left, level +1)

traverse(node.right, level+1)

END OF SUB FUNCTION

traverse(root, level = 0)

Return result

END OF MAIN FUNCTION

Demonstration:

For this example [3,9,20,null,null,15,7]

First we call traverse(root(3), 0) => this will check the IF statements (first = False, second = True) => result = [[]] then it appends the node.value => result [[3]].

Then it does a recursive call so traverse(node.left, 0+1) the array will be [[3] [9]]

COMP 352 Algorithm and solved Problems

Then it does another traverse to the left it finds that there are no children nodes => return nothing

So now we have traverse(node.right, 0+1) => result will be [[3] [9, 20]]

Then it performs traverse on node.left => result will be [[3] [9,20] [15]]

Then again node.left and finds nothing => return

Go to node.right finds 7 => append and result is [[3] [9,20] , [15, 7]]

```
def breadthFirstSearch(self) -> list[list[int]]:    #Breadth-First-search (BFS)
    result = []
    def traverse(node:TreeNode, level):
        if node is None:
            return
        if len(result)<=level:
            result.append([])

        result[level].append(node.val)

        traverse(node.left, level+1)
        traverse(node.right, level+1)
    traverse(self.root, 0)
    return result
```

Problem 11: (Binary Trees)

You are given an array of sorted integers, convert it to a height-balanced binary tree.

Example:

Input: head = [-10,-3,0,5,9]

Output: [0,-3,9,-10,null,5]

Explanation: One possible answer is [0,-3,9,-10,null,5], which represents the shown height balanced BST.

SOLUTION:

Time and space complexity are both $O(n)$.

Pseudo code:

ALGORITHM sortedArrayToBst(nums: Array[Integer], lower: Integer, upper: Integer)

IF lower > upper Then

Return nothing //Base case

ELSE

Mid = (lower +upper) // 2

Node = Create TreeNode with Value nums[mid]

Node.left = sortedArrayToBst(nums, lower, mid-1)

Node.right = sortedArrayToBst(nums, mid+1, upper)

Return Node

COMP 352 Algorithm and solved Problems

```
def sortedArrayToBST(nums: list[int], lower: int, upper: int):  
    if lower > upper:  
        return None  
    else:  
        mid = (lower+upper)//2  
        node = TreeNode(val = nums[mid])  
        node.left = sortedArrayToBST(nums, lower, mid-1)  
        node.right = sortedArrayToBST(nums, mid+1, upper)  
        return node
```

Problem 12: (Heaps)

Given an integer array nums and an integer k, return the kth largest element in the array.

Note that it is the kth largest element in the sorted order, not the kth distinct element.

(Do not sort the array for this one)

Examples:

Input: nums = [3,2,1,5,6,4], k = 2

Output: 5

Input: nums = [3,2,3,1,2,4,5,5,6], k = 4

Output: 4

Idea:

Think of using min or Max heap to solve this one:

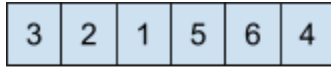
COMP 352 Algorithm and solved Problems

SOLUTION:

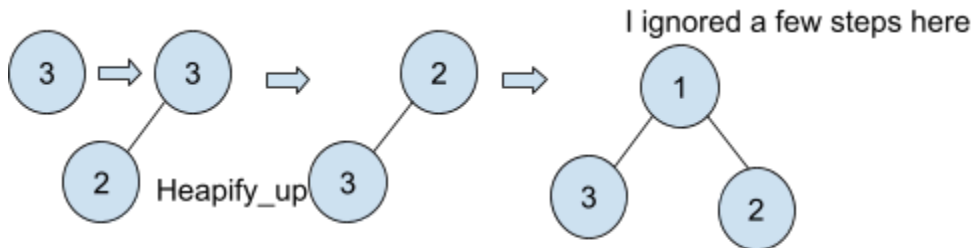
Using a Min heap, we'll tackle this by a quick demonstration.

Input: nums = [3,2,1,5,6,4], k = 2

Demonstration:

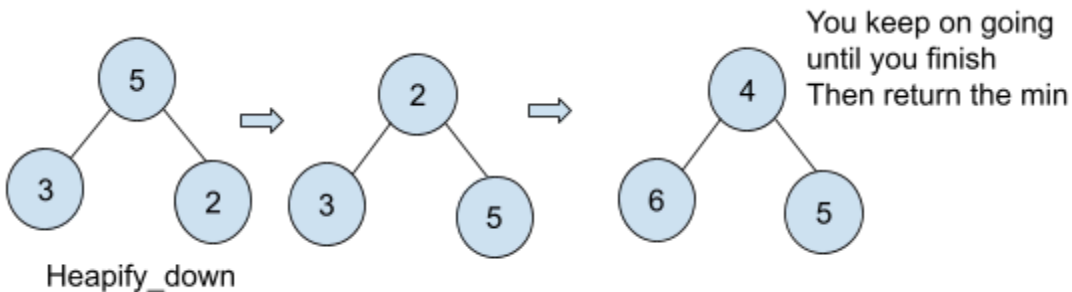


For $i = 0$ to $k - 1$, meaning we're looping from 0 to 1



For $i = k$ to $\text{len}(\text{arr})$

Check if $\text{min} < \text{value}$ then replace value with min



Pseudo Code:

```
ALGORITHM findKthLargest(nums: Array [ Integers], k: Integers) : Returns Integer
    Initialize MinHeap called heap
    For i from 0 to k-1 do begin
        heap.push( nums[ i ] )
        heap.heapify_up( length(heap) -1)
    For i from k to length(nums) do begin
        Min = heap.peek() // returns the minimum
        If Min < nums [ i ] then
            heap.replace(Min, nums[i] ) // Replace the min with the value from num
            heap.heapify_down(0)
    Return heap.peek()
END OF ALGORITHM
```

COMP 352 Algorithm and solved Problems

```
def findKthLargest(nums: list[int], k: int) -> int:
    heap = minHeap()
    for value in nums[:k]:
        heap.push(value)
    for value in nums[k:]:
        Min = heap.peak()
        if Min < value:
            heap.replace(value)
    return heap.peak()
```

Problem 13: (Heaps)

Given an array of integers nums, sort the array in ascending order and return it.

Criteria: Your solution must be in time complexity of $O(n\log(n))$. You can't use any sorting algorithms. The only allowance is to use heaps (min, Max).

Example

Input: nums = [5,2,3,1]

Output: [1,2,3,5]

Input: nums = [5,1,1,2,0,0]

Output = [0,0,1,1,2,5]

SOLUTION:

ALGORITHM sortArray(nums: Array[Integer]) -> sorted array

Initialize a min heap called " heap"

heap = minHeap()

For val in nums do begin // $O(n)$

heap.push(val) // $O(1)$

heap.heapify_up(len(heap)-1) // $O(\log(n))$

Initialize an empty array result

For val in heap do begin

result.append(heap.pop()) // Pop the min and add push it to the array $O(1)$

heap.heapify_down(0) // $O(\log(n))$ of index 0

Return result

End of Algorithm

Time complexity $O(n \log(n))$

Space Complexity $O(n)$

COMP 352 Algorithm and solved Problems

Problem 14: (Binary Trees)

You are given the root of a binary tree. invert the tree, and return its root.

Example:

Draw the tree to understand

Input: root = [4,2,7,1,3,6,9]

Output: [4,7,2,9,6,3,1]

Input: root = [2,1,3]

Output: [2,3,1]

Idea:

depth-first traversal (DFS)

SOLUTION:

Time complexity is $O(n)$ (traversing both right and left side)

Space Complexity $O(n)$ for the swapping (if the tree is balanced then it could be $O(\log(n))$)

ALGORITHM invertTree(root: Root of the Tree) => root

If root is null then

Return null

swap(root.left, root.right) // swap the left with the right

invertTree(root.left)

invertTree(root.right)

Return root

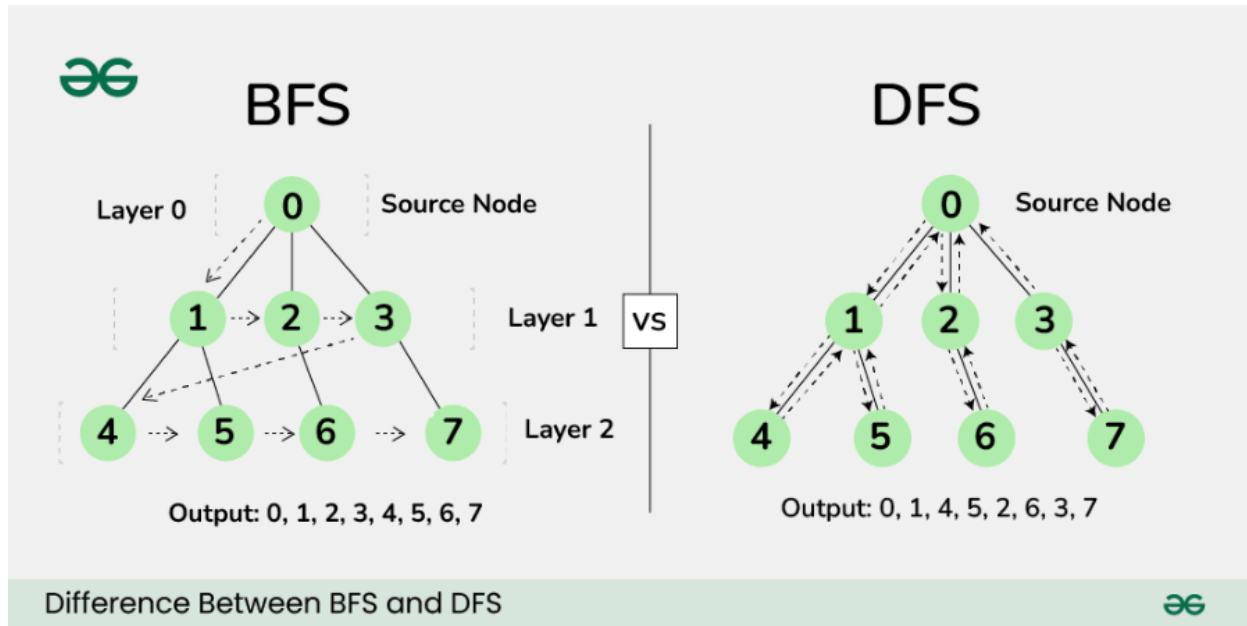
End of Algorithm

```
def invertTree(self):  
    def invertTree(node: TreeNode):  
        if node is None:  
            return None  
        node.right, node.left = node.left, node.right  
  
        invertTree(node.left)  
        invertTree(node.right)  
    invertTree(self.root)  
    return self.root
```

COMP 352 Algorithm and solved Problems

Problem 15: (Binary Trees)

Given the root of a tree. Write an algorithm for a Breadth-first and Depth-first search.



Difference between BFS and DFS

Idea:

Recall that: “BFS(Breadth First Search) uses Queue data structure for finding the shortest path.”

Recall that: “DFS(Depth First Search) uses Stack data structure”

Additional information for you:

“BFS is used in various applications such as bipartite graphs, shortest paths, etc. If weight of every edge is same, then BFS gives shortest path from source to every other vertex.”

“DFS is used in various applications such as acyclic graphs and finding strongly connected components etc. There are many applications where both BFS and DFS can be used like Topological Sorting, Cycle Detection, etc”

Further Information: <https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/>

(BFS) Algorithm:

ALGORITHM *BFS*(root: *TreeNode*)

Initialize a double ended queue = root

Initialize an empty array result

While queue do begin

Current = queue.popleft()

result.push(current.value)

If Current.left

COMP 352 Algorithm and solved Problems

```
        Then queue.push(current.left)
    If current.right
        Then queue.push(current.right)
Return result
```

```
def return_BFS(self):
    result = []
    queue = deque([self.root])
    while queue:
        current = queue.popleft()
        result.append(current.val)

        if current.left:
            queue.append(current.left)
        if current.right:
            queue.append(current.right)
    return result
```

DFS is same is a doing a pre order traversal

ALGORITHM DFS(root: TreeNode)

```
    If root is null Then
        Return nothing
    print(root.value)
    DFS(root.left)
    DFS(root.right)
```

END

Problem 16: (Graphs)

Information to know:

Vertex: a point in that graph that represents an entity

Edge:(Arc) a connection between two nodes that is either direct or indirect

Path: a sequence of vertices

Cycle: a path that starts and ends with the same vertex.

Given a reference of a node in a connected undirected graph.

Return a deep copy (clone) of the graph.

COMP 352 Algorithm and solved Problems

To simplify this, here is the class of both Node and Graph

Class **Node**:

Public Integer val

Public Object neighbors

*Define a **constructor**(This, val = 0, neighbors = null)*

This.val = val

This.neighbors = neighbors if neighbors is not null else [] (empty array)

End of class **Node**

Class **Graph**:

Public Object graph

*Define a **Constructor**(This, graph=null)*

This.graph = graph

End **Graph**

Idea:

For a good explanation, read the given information. So we want to create a deep copy of the entire graph and not a shallow copy.

Recall that a **deep copy** creates a new Object and recursively copies all objects it refers to.

A **shallow copy** creates a new object but does not recursively copy the object to the original object it refers to. It only copies the references

```
# Create nodes
node1 = Node(val=1)
node2 = Node(val=2)
node3 = Node(val=3)
node4 = Node(val=4)

# Define neighbors
node1.neighbors = [node2, node3]
node2.neighbors = [node1, node4]
node3.neighbors = [node1, node4]
node4.neighbors = [node2, node3]

# At this point, the graph is created

# ( 1 )----- ( 2 )
#   |               |
#   |               |
#   |               |
#   |               |
# ( 3 )----- ( 4 )

g = Graph(node1)
```

COMP 352 Algorithm and solved Problems

SOLUTION:

The idea here is to perform a depth first search and create a new Node for each recursively in order to deep copy the entire graph.

DFS (graph) = 1 -> 2 -> 4 -> 3

Further Explanation

In this example if we want to display our graph it would be in this form:

Graph = { 1: [2,3] , 2: [1,4], 4: [2,3], 3: [1,4]}

Graph.keys() = {1,2,3,4}

Graph.items() = { [2,3] , [1,4] , [2,3] , [1,4] }

Basically for the keys (1) = Node(val = 1, neighbors= [Node(val=2, neighbors= [...Node]) , Node(val=3, neighbors = [...Node])])

The way to stop your code from looping is if you create a map and store these nodes in it and only account for the ones that are not in the map.

ALGORITHM cloneGraph(node: Node)

If node is null Then

Return null

Initialize and empty map called visited = { }

Function DepthFirstSearch(node:Node)

If node in visited Then

Return visited[node]

Clone = Node(val = node.val, neighbors = [])

visited [node] = clone

For neighbor in node.neighbors do begin

clone.neighbors.append(DepthFirstSearch(neighbor))

Return clone

Return DepthFirstSearch(node)

End of Algorithm

```
def cloneGraph(self):
    if not self.graph:
        return None
    visited = {}
    def dfs(node:Node):
        if node in visited:
            return visited[node]
        clone = Node(val = node.val, neighbors=[])
        visited[node] = clone
        for neighbor in node.neighbors:
            clone.neighbors.append(dfs(neighbor))

        return clone
    return dfs(self.graph)
```

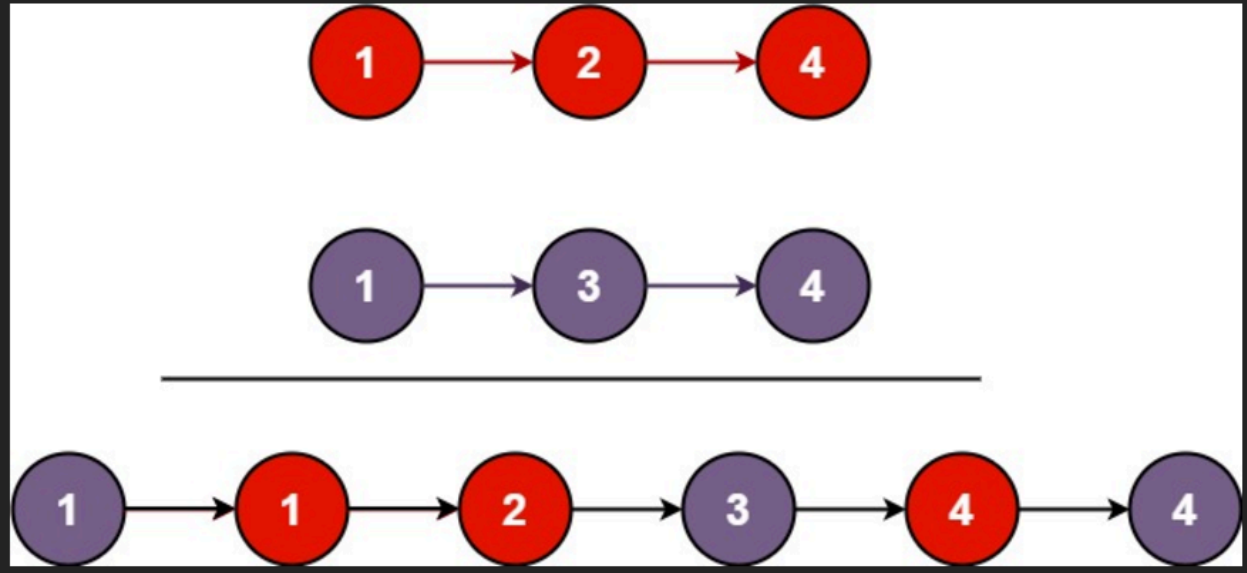
COMP 352 Algorithm and solved Problems

Problem 17: (Merging Linked List)

You are given the heads of two nodes, node1 and node2. Merge the two linked lists into one sorted linked list. The two nodes are already sorted

ALGORITHM mergeTwoLists(node1: ListNode, node2: ListNode) → ListNode

Example 1:

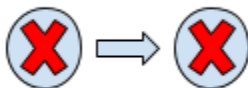


Idea:

node1



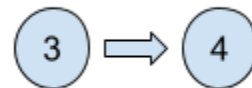
node2



Original node1



Original node2



$1 < 3 \Rightarrow$ create new Node and move head to (2) and repeat



COMP 352 Algorithm and solved Problems

SOLUTION:

ALGORITHM *mergeTwoLists*(node1: ListNode, node2: ListNode) → ListNode

If node1 is null Then

Return node2

If node2 is null Then

Return node1

If node1.value < node2.value Then

Merged = ListNode(node1.value)

Marged = mergeTwoLists(node1.next, node2)

Else

Merged = ListNode(node2.value)

Marged = mergeTwoLists(node1, node2.next)

Return Merged

End of Algorithm

Time and space complexity are both $O(n)$

Time for node1 of n_1 length and node2 of n_2 length

The time is $O(n_1+n_2)$

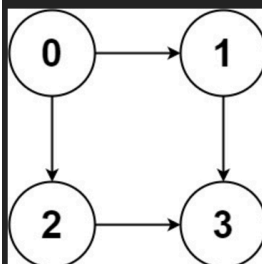
Space Complexity is for the creation of the ListNode and chaining it.

Problem 18: (Graph)

Given a directed acyclic graph (DAG) of n nodes labeled from 0 to $n - 1$, find all possible paths from node 0 to node $n - 1$ and return them in any order.

The graph is given as follows: $\text{graph}[i]$ is a list of all nodes you can visit from node i (i.e., there is a directed edge from node i to node $\text{graph}[i][j]$).

Example 1:

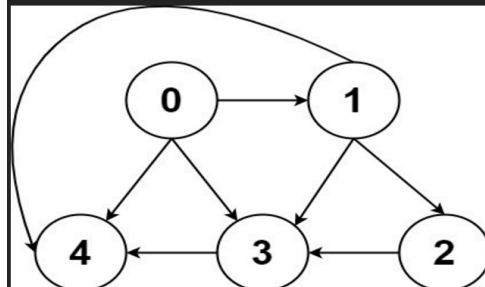


Input: $\text{graph} = [[1,2],[3],[3],[]]$

Output: $[[0,1,3],[0,2,3]]$

Explanation: There are two paths: $0 \rightarrow 1 \rightarrow 3$ and $0 \rightarrow 2 \rightarrow 3$.

Example 2:

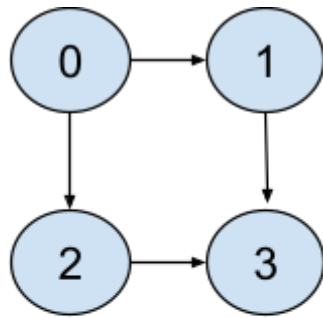


Input: $\text{graph} = [[4,3,1],[3,2,4],[3],[4],[]]$

Output: $[[0,4],[0,3,4],[0,1,3,4],[0,1,2,3,4],[0,1,4]]$

COMP 352 Algorithm and solved Problems

Idea:



In order to solve this problem, we'll need to perform a depth first search



SOLUTION

*ALGORITHM **allPathsSourceTarget** (graph: Array[Array[Integers]]) : returns an Array[Array[Integers]]*

Paths = [] // empty array

*FUNCTION **firstDepthSearch**(path, index)*

If index == Length(graph) - 1

Paths.push (path)

Return Nothing

For neighbor in graph[index]

firstDepthSearch(path + [neighbor], neighbor)

END of Function

firstDepthSearch([0], 0)

Return Paths

END OF ALGORITHM

Explanation:

Let's consider the following graph:

graph = [[1,2],[3],[3],[]]

Index = 0, we start with [0]

Check index == len(graph) -1 (FALSE)

For neighbor in graph[0] we have [1,2]

Call firstDepthSearch([0] + [1], 1)

Index = 1

Check index == len(graph) -1 (FALSE)

COMP 352 Algorithm and solved Problems

For neighbor in graph[1] we have [3]

Call firstDepthSearch([0,1] + [3], 3)

Index = 3

Check index == len(graph) - 1(TRUE) add a path to Paths and so on.

Problem 19: (Graph)

(Information for you: bi-directional graphs are also indirect graphs $A \Leftrightarrow B$)

Find if Path Exists in Graph. There is a bi-directional graph with n vertices, where each vertex is labeled from 0 to $n - 1$ (inclusive). The edges in the graph are represented as a 2D integer array edges, where each edges[i] = [ui, vi] denotes a bi-directional edge between vertex ui and vertex vi. Every vertex pair is connected by at most one edge, and no vertex has an edge to itself.

You want to determine if there is a valid path that exists from vertex source to vertex destination.

Given edges and the integers n , source, and destination, return true if there is a valid path from source to destination, or false otherwise.

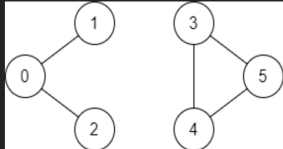
ALGORITHM `validPath(n: int, edges: List[List[int]], source: int, destination: int) -> bool:`

Example 1:



Input: $n = 3$, edges = [[0,1],[1,2],[2,0]], source = 0, destination = 2
Output: true
Explanation: There are two paths from vertex 0 to vertex 2:
- 0 → 1 → 2
- 0 → 2

Example 2:



Input: $n = 6$, edges = [[0,1],[0,2],[3,5],[5,4],[4,3]], source = 0, destination = 5
Output: false
Explanation: There is no path from vertex 0 to vertex 5.

Let's look at Example 1 for a better explanation:

edges = [[0,1],[1,2],[2,0]]

[0,1] Indicates Node (0) bidirectionally connected to Node (1)

[1,2] Indicates Node (1) bidirectionally connected to Node (2)

[2,0] Indicates Node (2) bidirectionally connected to Node (0)

Idea:

The idea here is to perform either a *Breadth first search* using a queue or a *Depth First search* to find if the path from a given source to the destination exists.

Think also about turning the edges to a graph.

SOLUTION

We'll use a Depth First search as it can be done recursively and quickly.

COMP 352 Algorithm and solved Problems

First we'll need to turn this edges = $[[0,1],[1,2],[2,0]]$ to this graph = $\{0: [1,2], 1:[0,2], 2:[0,1]\}$

ALGORITHM validPath(n, edges, source, destination) -> bool:

n: Integer

edges: Array of Array of Integers [[u,v], [u,v], [u,v]]

source: Integer

destination: Integer

Return value [true, false]

Define a map called graph = { }

For u,v in edges do begin

If u not exist in graph

graph[u] = [] // empty array

If v not exist in graph

graph[v] = [] // empty array

graph[u].append(v)

graph[v].append(u)

End of for

Define an empty set called visited

FUNCTION dfs(source):

If source == destination then

Return TRUE

visited.add(source)

For neighbor in graph[source]

If neighbor not in visited

If dfs(neighbor)

Return True

Return False

END OF FUNCTION

Return dfs(source)

END of ALGORITHM

EXPLANATION:

Define a map called graph = { }

For u,v in edges do begin

If u not exist in graph

graph[u] = [] // empty array

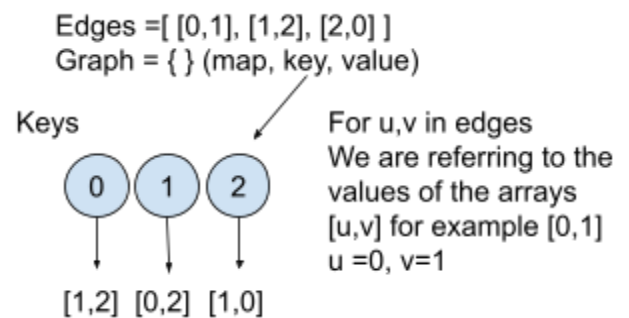
If v not exist in graph

graph[v] = [] // empty array

graph[u].append(v)

graph[v].append(u)

End of for



COMP 352 Algorithm and solved Problems

Define an empty set called visited

```
FUNCTION dfs(source):
    If source == destination then
        Return TRUE
    visited.add(source)
    For neighbor in graph[source]
        If neighbor not in visited
            If dfs(neighbor)
                Return True
    Return False
END OF FUNCTION
Return dfs(source)
```

Here we perform a depth-first-search using the set visited to ensure that we don't visit nodes more than one time otherwise, it will be an endless loop.

Let's consider this example:

graph = {0: [1,2], 1:[0,2], 2:[0,1]}

We call dfs(source = 0)

```
0 == 2 (FALSE)
visited.add(0)
For neighbor in [1,2]
    1 is not in visited
        Check dfs(1)
            1 == 2 False
            visited.add(1)
            For neighbor in [0,2]
                0 is in visited so move to 2
                dfs(2)
                2 == 2 Return True
            End
```

Let's consider this example:

graph = {0: [1,2], 1:[0], 2:[0], 3:[4,5], 4:[3,5], 5:[3,4]} (second example given)

We call dfs(source = 0)

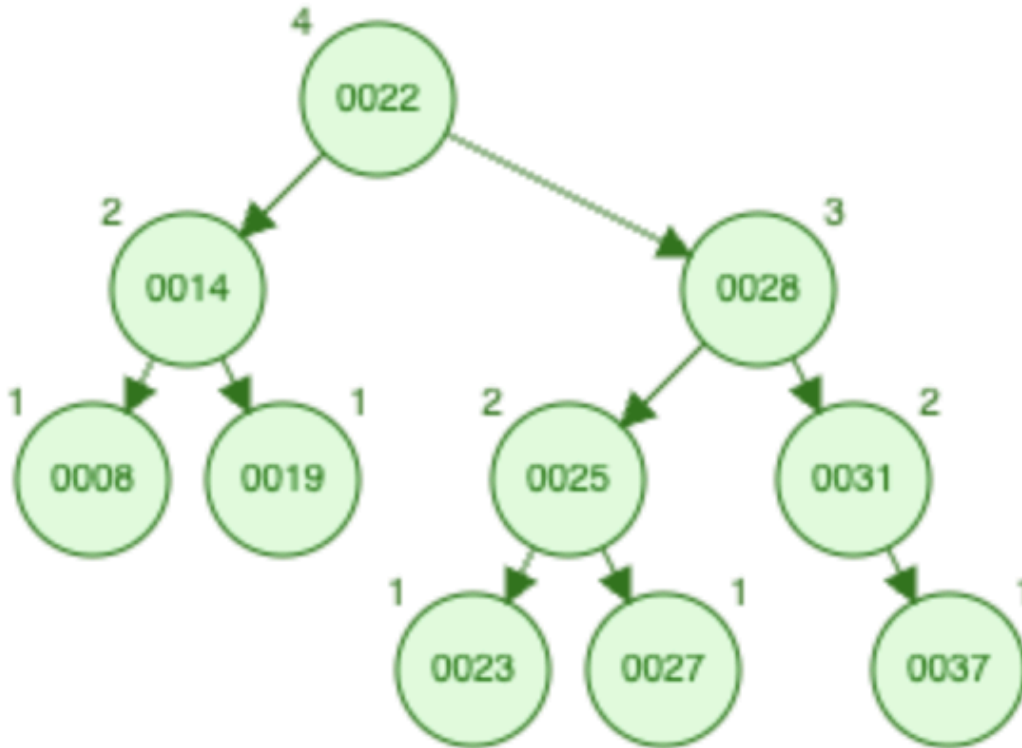
```
0 == 2 (False)
visited.add(0)
For neighbor in [1,2]
    1 is not in visited
        Check if dfs(1)
            1 == 2 False
            visited.add(1)
            For neighbor in [0]
                0 is in Visited So we skip the for loop and return False
```

COMP 352 Algorithm and solved Problems

Problem 19: (AVL Trees)

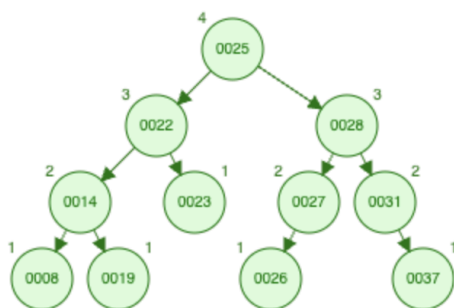
<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

You are given the following AVL tree, insert these values and perform what's necessary 26, 34, 40, 29, 33, 32

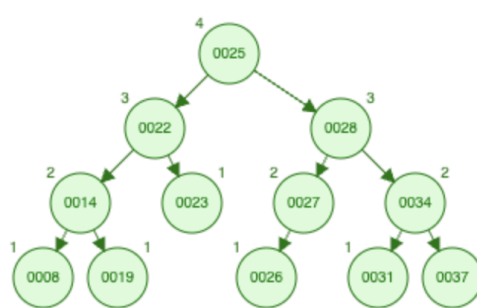


SOLUTION:

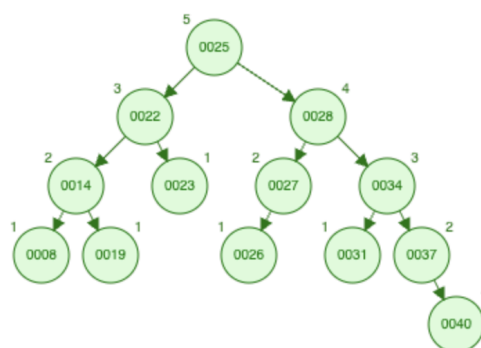
Insert(26)



insert(34)

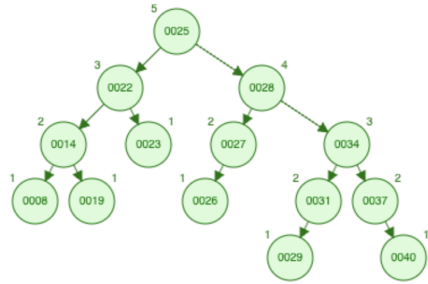


insert(40)(Nothing)

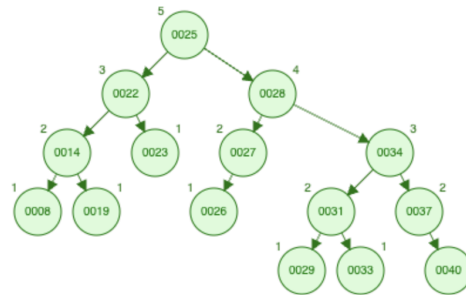


COMP 352 Algorithm and solved Problems

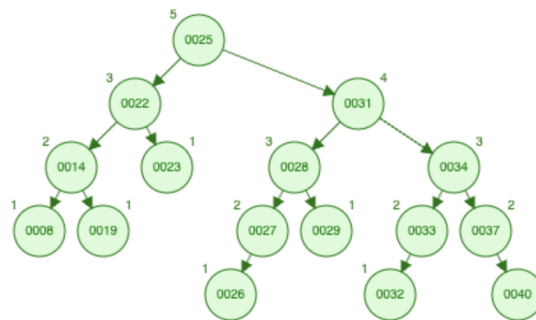
insert(29) (Nothing occurs)



Insert(33) (Nothing occurs)



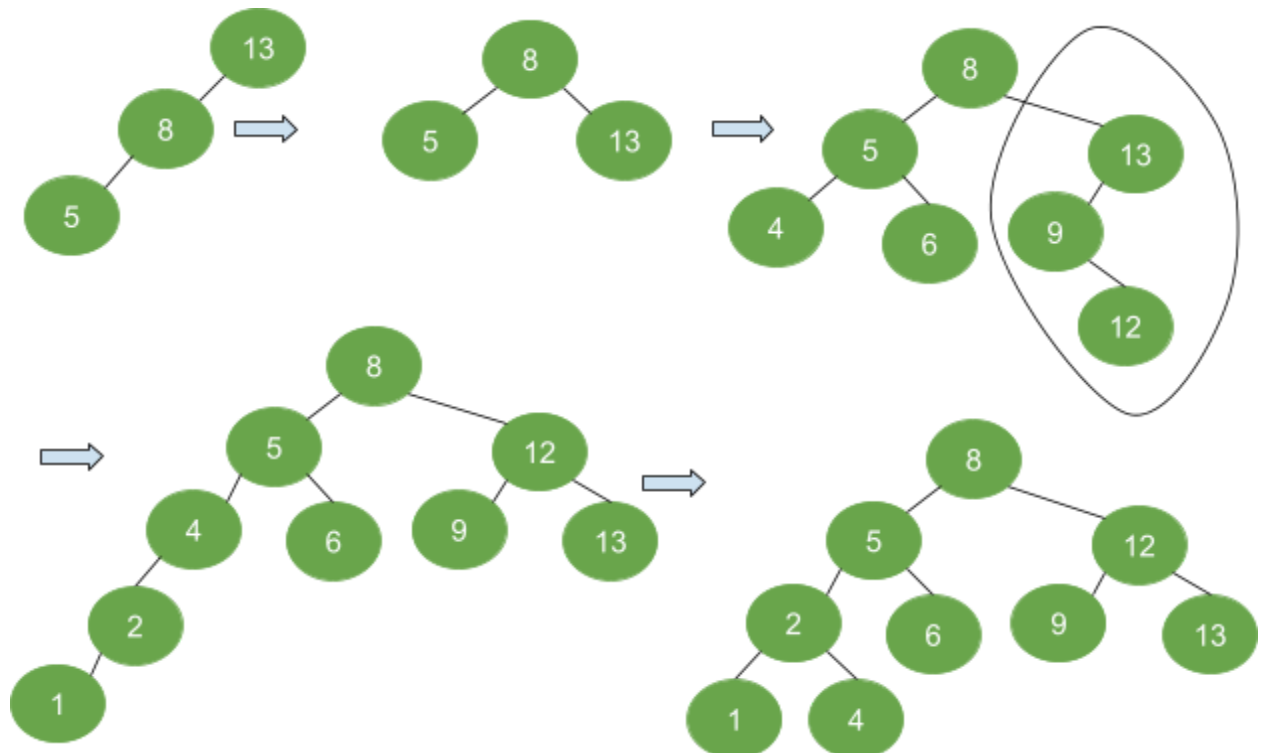
insert(32)



If you need help understanding the steps for each, reach out to me. Otherwise, if you're all good, then congrats.

Problem 20: (AVL Trees)

Insert the following nodes and show the steps for each 13, 8, 5, 9, 4, 6, 12, 2, 1 and 3



COMP 352 Algorithm and solved Problems

