1. Merge two sorted lists:

class Node:  
  
 def \_\_init\_\_(self, data = 0):  
 self.data = data  
 self.next = None  
  
  
  
def merge\_lists(self, l1, l2):  
  
 storage = Node()  
 content = storage  
  
 while l1 and l2:  
 if l1.value < l2.value:  
 content.next = l1  
 l1 = l1.next  
  
 else:  
 content.next = l2  
 l2 = l2.next  
  
 content = content.next  
  
 if l1:  
 content.next = l1  
 elif l2:  
 content.next = l2  
  
  
  
print(merge\_lists([1,2], [3,4]))

2. Duplicate number

def removeDuplicates(nums):  
  
 stanga = 1  
  
 for dreapta in range(1, len(nums)):  
 if nums[dreapta] != nums[dreapta-1]:  
 nums[stanga] = nums[dreapta]  
  
 stanga += 1  
  
 return [stanga]  
  
print(removeDuplicates([0,0,1,1,1,2,2]))

3. Length of last word

def lengthOfLastWord(s):  
  
 #pointers  
 index = len(s) - 1  
 lungime = 0  
  
 while s[index] == " ":  
 index -= 1  
  
 while index > 0 and s[index] != " ":  
 lungime += 1  
 index -= 1  
  
 return lungime  
  
print(lengthOfLastWord(" text "))

4. Plus one

def plusOne(digits):  
  
 digits = digits[::-1]  
 unul = 1  
 index = 0  
  
 while unul:  
 if index < len(digits):  
 if digits[index] == 9:  
 digits[index] = 0  
  
 else:  
 digits[index] += 1  
 unul = 0  
  
 elif index > len(digits):  
 digits.append(1)  
 unul = 0  
  
 index+= 1  
  
 return digits[::-1]  
  
  
  
print(plusOne([1,2,9]))

5. SQRT

def mySqrt(x):  
  
 stanga = 1  
 dreapta = x  
  
 while stanga <= dreapta:  
 mijloc = (dreapta + stanga) / 2  
  
 if mijloc \* mijloc == x:  
 return mijloc  
  
 elif mijloc \* mijloc > x:  
 dreapta = mijloc - 1  
  
 else:  
 stanga = mijloc + 1  
  
 return round(dreapta)  
  
  
print(mySqrt(15))

**Linked List problems:**

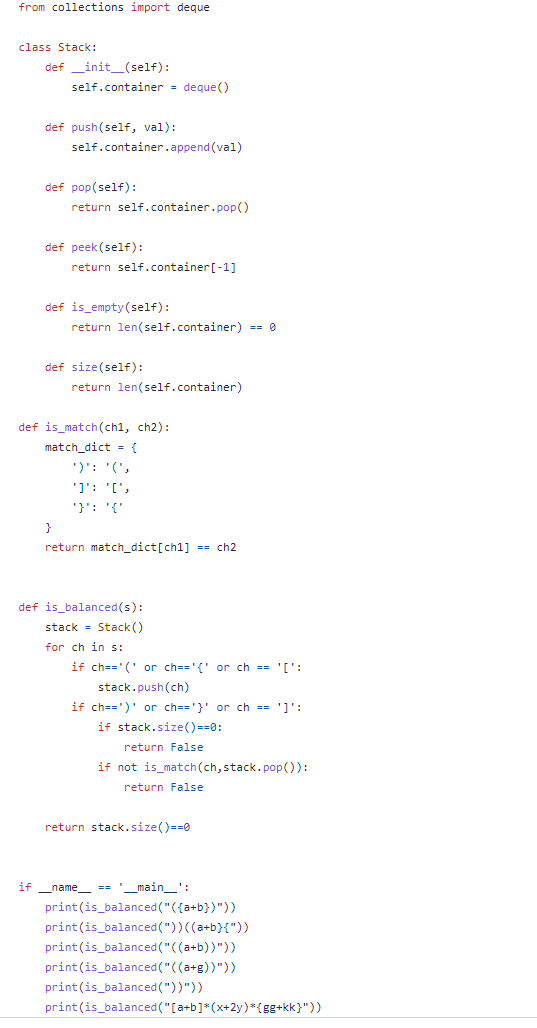
Append and Print

class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.next = None  
  
  
class Linked\_List:  
 def \_\_init\_\_(self):  
 self.head = None  
  
  
 def add\_node(self, new\_information):  
  
 new\_node = Node(new\_information)  
  
 if self.head is None:  
 self.head = Node(new\_information)  
 else:  
  
 current = self.head  
 while current.next:  
 current = current.next  
  
 current.next = new\_node  
  
  
  
  
 def print\_llist(self):  
  
 if self.head is None:  
 print("The LL is empty")  
  
 current = self.head  
  
 while current:  
  
 print(current.data)  
 current = current.next  
  
  
test = Linked\_List()  
test.add\_node(2)  
test.add\_node(3)  
test.add\_node(4)  
test.add\_node(5)  
test.print\_llist()

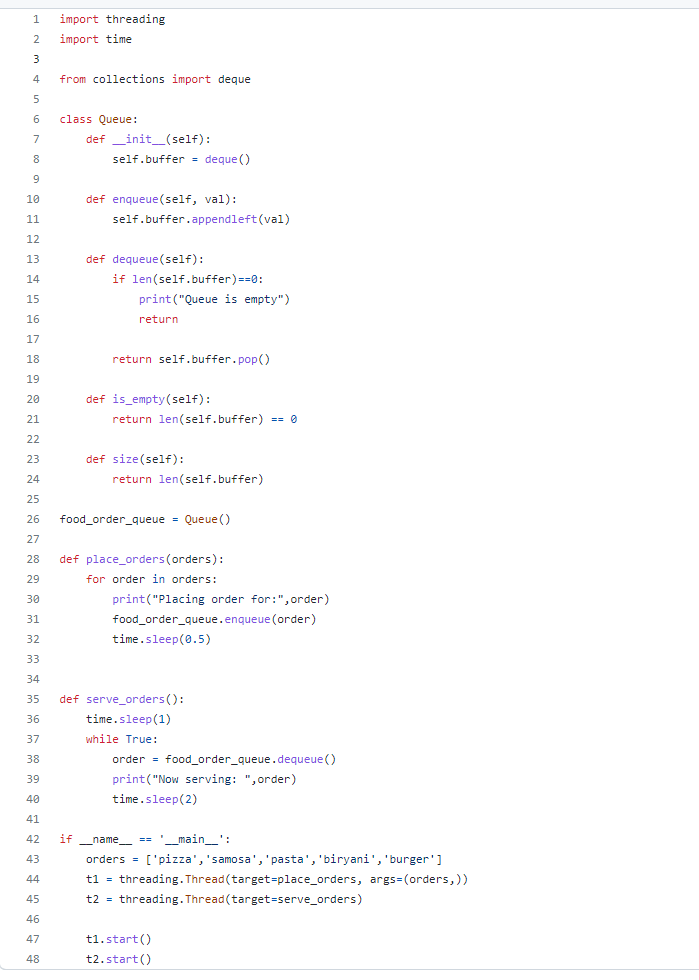
**Insert at an index**

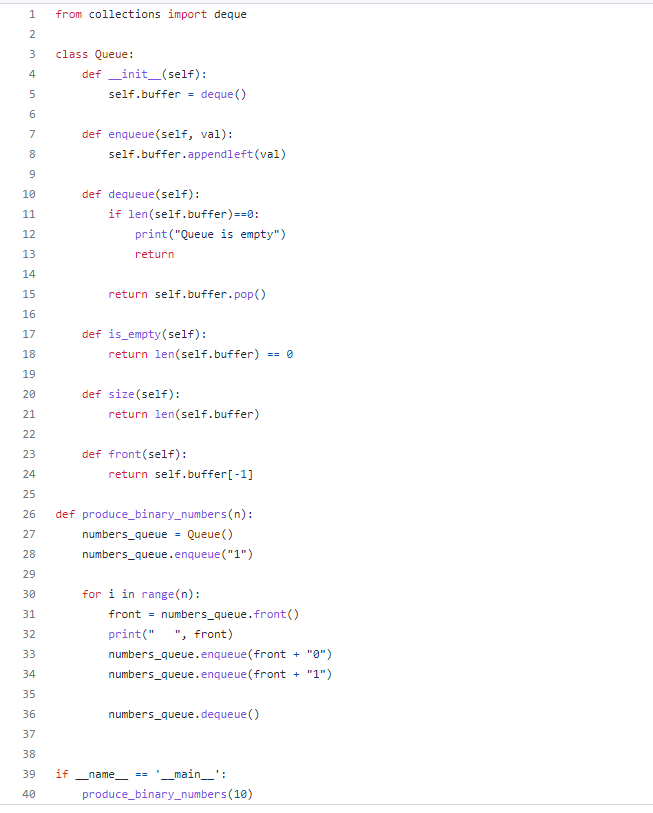
def insert\_at(self, index\_no, new\_info):  
 new\_node = Node(new\_info, self.head)  
 if index\_no < 0 or index\_no > self.get\_length():  
 print("Out of range")  
  
 if index\_no == 0:  
 self.head = new\_node  
   
 current = self.head  
 position = 0  
  
 while current:  
 new\_node = Node(new\_info, current.next)  
 if position == index\_no - 1:  
 current.next = new\_node  
 break  
  
 current = current.next  
 position +=1

**Stack**

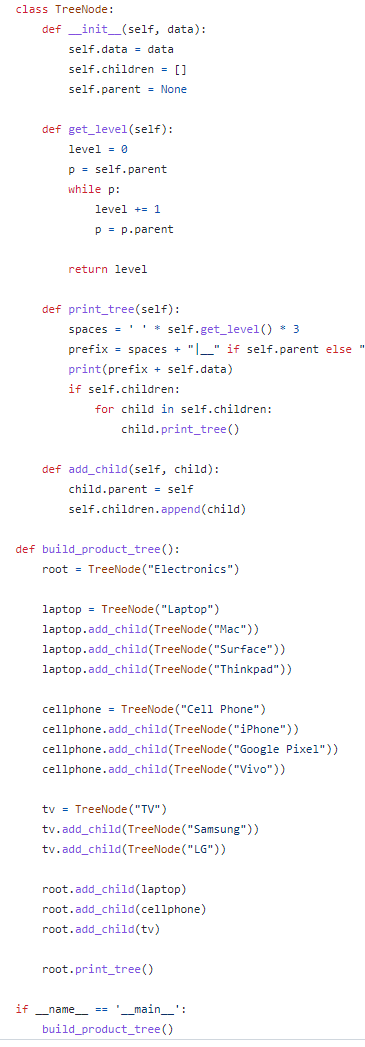


**Queue**

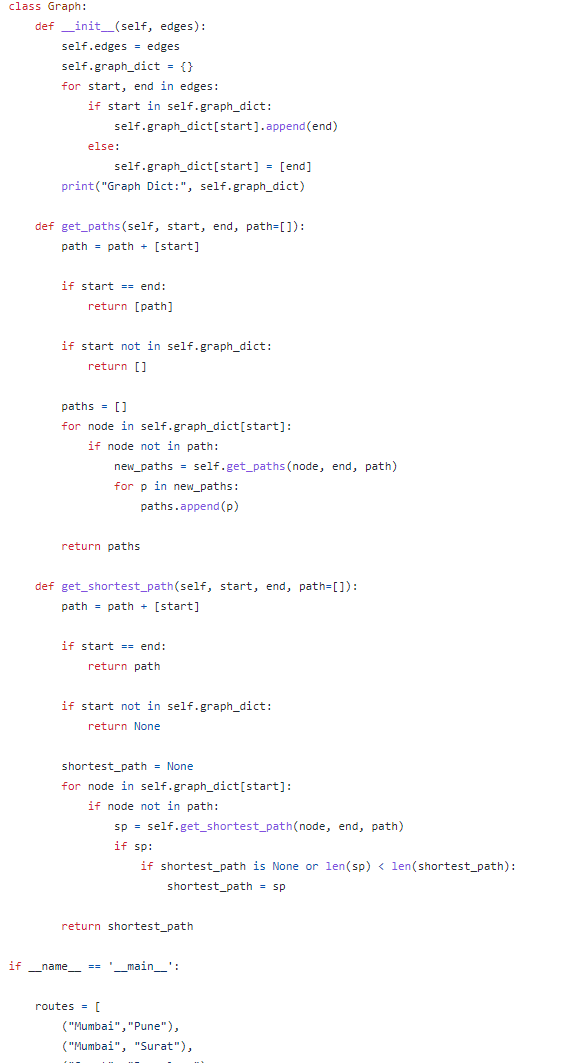




**Tree**



**Graph**





**Trie**

class Trie:  
 def \_\_init\_\_(self):  
 self.root = {}  
  
 def insert(self, word):  
 current\_node = self.root  
 for char in word:  
 if char not in current\_node:  
 current\_node[char] = {}  
 current\_node = current\_node[char]  
 current\_node['\*'] = True # mark the end of a word  
  
 def search(self, word):  
 current\_node = self.root  
 for char in word:  
 if char not in current\_node:  
 return False  
 current\_node = current\_node[char]  
 return '\*' in current\_node