LinkedList

In [4]:
```
class Node :
    def __init__(self,value):
        self.value = value
        self.next = None

class linkedList:
    def __init__(self,value):
        new_node = Node(value)
        self.head = new_node
        self.tail = new_node
        self.length = 1

    def print_list(self):
        temp = self.head
        while temp is not None:
            print(temp.value)
            temp = temp.next

    def append(self,value):
        new_node = Node(value)
        if self.length == 0:
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            self.tail = new_node
        self.length +=1
        return True

    def pop(self):
        if self.length == 0:
            return None
        if self.length == 1:
            self.head = None
            self.tail = None
        else:
            temp = self.head
            prev = self.head
            while temp.next is not None:
                prev = temp
                temp = temp.next
            self.tail = prev
            self.tail.next = None
        self.length -=1

        return temp.value

    def prepend(self,value):
        new_node = Node(value)
        if self.length == 0:
            self.head = new_node
```

```
            self.tail = new_node
        else:
            new_node.next = self.head
            self.head = new_node
        self.length +=1
        return True

    def pop_first(self):
        if self.length == 0:
            return None
        temp = self.head
        self.head = self.head.next
        temp.next = None
        self.length -=1
        if self.length == 0:
            self.head == None
            self.tail = None
        return temp.value

    def get(self,index):
        if index<0 and index>=self.length:
            return None
        temp = self.head
        for _ in range(index):
            temp = temp.next
        return temp


    def set_value(self,index,value):
        temp = self.get(index)
        if temp :
            temp.value= value
            return True
        return False

    def insert(self,index,value):
        if index<0 and index>self.length:
            return None
        if index == 0:
            return self.prepend(value)
        if index == self.length:
            return self.append(value)
        new_node = Node(value)
        temp = self.get(index-1)
        new_node.next = temp.next
        temp.next = new_node
        self.length+=1
        return True

    def remove(self,index):
        if index < 0 and index >=self.length:
            return None
        if index == 0:
            return self.pop_first()
        if index == self.length-1:
            return self.pop()
```

```
        prev = self.get(index-1)
        temp = prev.next
        prev.next = temp.next
        self.length -=1
        return temp

    def reverse(self):
        prev = None
        current = self.head
        while current:
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node
        self.head = prev
        return True

    #find_middle_node without length attribute

    def find_middle_node(self):
        fast = self.head
        slow = self.head
        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
        return slow

    #detect whether LL has loop or not
    def has_loop(self):
        slow = self.head
        fast = self.head
        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
            if fast == slow:
                return True
        return False

    #find kth element from end without length attribute

    def find_kth_from_end(ll,k):
        slow = ll.head
        fast = ll.head
        for _ in range(k):
            if fast is None:
                return None
            fast = fast.next

        while fast :
            slow = slow.next
            fast = fast.next
        return slow

    # Partition List

    def partition_list(self,x):
```

```
        if self.head ==None:
            return None
        dummy1 = Node(0)
        dummy2 = Node(0)
        prev1 = dummy1
        prev2 = dummy2
        current = self.head
        while current is not None:
            if current.value < x:
                prev1.next = current
                prev1 = current
            else:
                prev2.next = current
                prev2 = current
            current = current.next
        prev2.next = None
        prev1.next = None
        prev1.next = dummy2.next
        self.head = dummy1.next

    #removes dublicat

    def remove_duplicates(self):
        values = set()
        prev = None
        curr = self.head
        while curr is not None:
            if curr.value in values:
                prev.next = curr.next
                self.length -= 1
            else:
                values.add(curr.value)
                prev = curr
            curr = curr.next

    #binary to decimal

    def binary_to_decimal(self):
        num = 0
        current = self.head
        while current:
            num = num * 2 + current.value
            current = current.next
        return num

    #reverse between m and n

    def reverse_between(self,m,n):
        if self.length<=1:
            return None
        dummy= Node(0)
        dummy.next = self.head
        prev = dummy
        for i in range(m):
            prev = prev.next
        curr = prev.next
```

```
        for i in range(n-m):
            # temp' will point to the next node in line that we want to reve
            temp = curr.next
            # Disconnect 'temp' from the list and point 'current' to the noc
            curr.next = temp.next
            # Prepare to insert 'temp' to its new position. Connect 'temp' t
            temp.next = prev.next
             # Now, connect 'prev' to 'temp' completing its new placement ir
            prev.next = temp
            # If m was 0, then we reversed from the start and we need to upc
        self.head = dummy.next
```

interview question

1. Find middle node

In [ ]:
```
def find_middle_node(self):
        fast = self.head
        slow = self.head
        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
        return slow
```

2. has loop

In [ ]:
```
def has_loop(self):
        slow = self.head
        fast = self.head
        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
            if fast == slow:
                return True
        return False
```

3. find_kth_from_end

In [22]:
```
def find_kth_from_end(ll,k):
        slow = ll.head
        fast = ll.head
        for _ in range(k):
            if fast is None:
                return None
            fast = fast.next

        while fast :
```

```
            slow = slow.next
            fast = fast.next
        return slow
```

4. Partition List

In [ ]:
```
def partition_list(self,x):
        if self.head ==None:
            return None
        dummy1 = Node(0)
        dummy2 = Node(0)
        prev1 = dummy1
        prev2 = dummy2
        current = self.head
        while current is not None:
            if current.value < x:
                prev1.next = current
                prev1 = current
            else:
                prev2.next = current
                prev2 = current
            current = current.next
        prev2.next = None
        prev1.next = None
        prev1.next = dummy2.next
        self.head = dummy1.next
```

5. remove dublicates

In [ ]:
```
def remove_duplicates(self):
        values = set()
        prev = None
        curr = self.head
        while curr is not None:
            if curr.value in values:
                prev.next = curr.next
                self.length -= 1
            else:
                values.add(curr.value)
                prev = curr
            curr = curr.next
```

6. binary to decimal

In [ ]:
```
def binary_to_decimal(self):
        num = 0
        current = self.head
        while current:
            num = num * 2 + current.value
            current = current.next
        return num
```

7. Reverse Between

In [ ]:
```python
def reverse_between(self,m,n):
        if self.length<=1:
            return None
        dummy= Node(0)
        dummy.next = self.head
        prev = dummy
        for i in range(m):
            prev = prev.next
        curr = prev.next
        for i in range(n-m):
            # temp' will point to the next node in line that we want to reve
            temp = curr.next
            # Disconnect 'temp' from the list and point 'current' to the noc
            curr.next = temp.next
            # Prepare to insert 'temp' to its new position. Connect 'temp' t
            temp.next = prev.next
             # Now, connect 'prev' to 'temp' completing its new placement ir
            prev.next = temp
            # If m was 0, then we reversed from the start and we need to upc
        self.head = dummy.next
```

Doubly Linked List

In [176…
```python
class Node:
    def __init__(self,value):
        self.value = value
        self.next = None
        self.prev = None

class doublyLinkedList:
    def __init__(self,value):
        new_node = Node(value)
        self.head = new_node
        self.tail = new_node
        self.length = 1

    def print_list(self):
        temp = self.head
        while temp is not None:
            print(temp.value)
            temp = temp.next

    def append(self,value):
        new_node = Node(value)
        if self.length == 0:
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node
```

```
        self.length+=1
        return True

    def pop(self):
        if self.length == 0:
            return None
        temp = self.tail
        if self.length == 1:
            self.head = None
            self.tail = None
        else:
            self.tail = self.tail.prev
            self.tail.next = None
            temp.prev = None
        self.length -=1
        return temp

    def prepend(self,value):
        new_node = Node(value)
        if self.length == 0:
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
        self.length +=1
        return true

    def pop_first(self):
        temp = self.head
        if self.length == 0:
            return None
        elif self.length == 1:
            self.head = None
            self.tail = None
        else:
            self.head = self.head.next
            self.head.prev = None
            temp.next = None
        self.length -=1
        return temp

    def get(self,index):
        if index<0 and index>=self.length:
            return False
        temp = self.head
        if index < self.length/2:
            for _ in range(index):
                temp = temp.next
        else:
            temp = self.tail
            for _ in range(self.length -1,index,-1):
                temp = temp.prev
        return temp.value
```

```
    def set_value(self,index,value):
        temp = self.get(index)
        if temp:
            temp.value = value
            return True
        return False

    def insert(self,index,value):
        if index<0 and index>self.length:
            return False
        if index == 0:
            return self.prepend(value)
        if index == self.length:
            return self.append(value)
        new_node = Node(value)
        before = self.get(index-1)
        after = before.next
        new_node.prev = before
        new_node.next = after
        before.next = new_node
        after.prev = new_node
        self.length += 1
        return True

    def remove(self,index):
        if index<0 and index>=self.length:
            return None
        if index == 0:
            return self.pop_first()
        if index == self.length-1:
            return self.pop()
        temp = self.get(index)
        temp.next.prev = temp.prev
        temp.prev.next = temp.next
        temp.next = None
        temp.prev = None
        self.length -= 1
        return temp
```

interview question

1. Swap the values of the first and last node

In [ ]:
```
def swap_first_last(self):
        if self.head is None or self.head == self.tail:
            return
        self.head.value,self.tail.value = self.tail.value,self.head.value
```

2. Reverses the order of the nodes in the list

In [ ]:
```
def reverse(self):
        # Initialize 'current_node' to the starting node of the doubly linke
        curr = self.head
        # Traverse through each node of the doubly linked list.
        while curr:
            # Swap the 'next' and 'prev' pointers of the current node.This e
            # direction of the node's pointers.
            curr.prev,curr.next = curr.next,curr.prev
            # Since the 'next' and 'prev' pointers of the 'current_node'
            # have been swapped, we move to what was originally the 'prev'
            # node to continue the reversal.
            # Note: In a reversed scenario, 'prev' becomes 'next', hence we
            curr = curr.prev
        # After all nodes have been reversed, the original head becomes the
        # and the original tail becomes the head. Swap the 'head' and 'tail
        self.head,self.tail = self.tail,self.head
```

3. Palindrome Checker

In [ ]:
```
def is_palindrome(self):
        if self.length <=1:
            return True
        else:
            forward_node = self.head
            backward_node = self.tail
            for _ in range(self.length//2):
                if forward_node.value != backward_node.value:
                    return False
                forward_node = forward_node.next
                backward_node = backward_node.prev
        return True
```

4. Swap Nodes in Pairs

In [ ]:
```
#for doublylinked list having next and prev
def swap_pairs(self):
        dummy = Node(0)
        dummy.next = self.head
        prev = dummy
        while self.head and self.head.next:
            first_node = self.head
            second_node = self.head.next
            prev.next = second_node
            first_node.next = second_node.next
            second_node.next = first_node
            second_node.prev = prev
            first_node.prev = second_node
            if first_node.next:
                first_node.next.prev = first_node
            self.head = first_node.next
            prev = first_node
        self.head = dummy.next
```

```
        if self.head:
            self.head.prev = None
```

In [ ]:
```
# for linked list having only next    (NEETCODE IO)
def swap_pairs(self):
        dummy = Node(0)
        curr = self.head
        prev = dummy
        while curr and curr.next:

            #save ptrs
            nxtPair = curr.next.next
            second = curr.next

            #reverse pair
            second.next = curr
            curr.next = nxtPair
            prev.next = second

            #update ptrs
            prev = curr
            curr = nxtPair

        return dummy.next
```

Stack

In [217…
```
class Node :
    def __init__(self,value):
        self.value = value
        self.next = None

class Stack:
    def __init__(self,value):
        new_node = Node(value)
        self.top = new_node
        self.height = 1

    def print_stack(self):
        temp = self.top
        while temp is not None:
            print(temp.value)
            temp = temp.next

    def push(self,value):
        new_node = Node(value)
        if self.height == 0:
            self.top = new_node
        else:
            new_node.next = self.top
            self.top = new_node
        self.height += 1

    def pop(self):
```

```
        if self.height ==0:
            return None
        temp = self.top
        if self.height == 1:
            self.top = None
        else:
            self.top = self.top.next
            temp.next = None
        self.height -=1
        return temp
```

Queue

In [278…

```
class Node :
    def __init__(self,value):
        self.value = value
        self.next = None

class Queue:
    def __init__(self,value):
        new_node = Node(value)
        self.first = new_node
        self.last = new_node
        self.length = 1

    def print_queue(self):
        temp = self.first
        while temp:
            print(temp.value)
            temp = temp.next


    def enqueue(self,value):
        new_node = Node(value)
        if self.length == 0:
            self.first = new_node
            self.last = new_node
        else:
            self.last.next = new_node
            self.last = new_node
        self.length += 1


    def dequeue(self):
        if self.length == 0:
            return None
        temp = self.first
        if self.length == 1:
            self.first = None
            self.last = None
        else:
            self.first = self.first.next
            temp.next = None
```

```
        self.length -=1
        return temp
```

Interview question

1. Implement Stack Using a List

In [295…

```
class Stack:
    def __init__(self):
        self.stack_list = []

    def print_stack(self):
        for i in range(len(self.stack_list)-1, -1, -1):
            print(self.stack_list[i])

    def push(self,value):
        self.stack_list.append(value)

    def peek(self):
        if self.is_empty():
            return None
        else:
            return self.stack_list[-1]

    def is_empty(self):
        return len(self.stack_list) == 0

    def pop(self):
        if not self.is_empty():
            return self.stack_list.pop()
        else:
            return None
```

2. Parentheses Balanced using stack class

In [ ]:

```
def is_balanced_parentheses(p):
    stack = Stack()
    for ch in p:
        if ch == "(":
            stack.push(ch)
        elif ch == ")":
            if stack.is_empty():
                return False
            stack.pop()
    return stack.is_empty()
```

3. Reversed string using stack class

In [ ]:
```
def reverse_string(str):
    stack = Stack()
    for ch in str:
        stack.push(ch)
    reversed = ""
    while not stack.is_empty():
        reversed += stack.pop()
    return reversed
```

4. Sort Stack

In [ ]:
```

```

5. Enqueue using Stacks

In [ ]:
```
def enqueue(self, value):
        for _ in range(len(self.stack1)):
            self.stack2.append(self.stack1.pop())
        self.stack1.append(value)
        for _ in range(len(self.stack2)):
            self.stack1.append(self.stack2.pop())
```

TREES

In [40]:
```
class Node:
    def __init__(self,value):
        self.value = value
        self.right = None
        self.left = None

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self,value):
        new_node = Node(value)
        if self.root == None:
            self.root = new_node
            return True
        temp = self.root
        while (True) :
            if new_node == temp.value:
                return False
            if new_node.value < temp.value:
                if temp.left is None:
                    temp.left = new_node
                    return True
                temp = temp.left
            else:
                if temp.right is None:
                    temp.right = new_node
                    return True
```

```
                temp = temp.right


    def contains(self,value):
        temp = self.root
        while temp is not None:
            if value < temp.value:
                temp = temp.left
            elif value > temp.value:
                temp = temp.right
            else:
                return True
        return False

    #Breadth First Search

    def BFS(self):
        current_node = self.root
        queue = []
        result = []
        queue.append(current_node)
        while len(queue)>0:
            current_node = queue.pop(0)
            result.append(current_node.value)
            if current_node.left is not None:
                queue.append(current_node.left)
            if current_node.right is not None:
                queue.append(current_node.right)
        return result

    #Depth First Search (Pre-order)

    def DFS_pre_order(self):
        results = []
        #Traverse recursion method
        def traverse(current_node):
            results.append(current_node.value)
            if current_node.left is not None:
                traverse(current_node.left)
            if current_node.right is not None:
                traverse(current_node.right)
        traverse(self.root)
        return results

    #Depth First Search (post-order)

    def DFS_post_order(self):
        results = []
        #Traverse recursion method
        def traverse(current_node):
            if current_node.left is not None:
                traverse(current_node.left)
            if current_node.right is not None:
                traverse(current_node.right)
            results.append(current_node.value)
        traverse(self.root)
```

```
        return results

    #Depth First Search (in-order)

    def DFS_in_order(self):
        results = []
        #Traverse recursion method
        def traverse(current_node):
            if current_node.left is not None:
                traverse(current_node.left)
            results.append(current_node.value)
            if current_node.right is not None:
                traverse(current_node.right)
        traverse(self.root)
        return results
```

Interview

A) BST: Validate BST

In [ ]:
```
def is_valid_bst(self):
        bst = self.dfs_in_order()
        for i in range(1,len(bst)):
            if bst[i] <= bst[i-1]:
                return False
        return True
```

B) BST: Kth Smallest Node

In [ ]:
```
def kth_smallest(self, k):
        stack = []
        node = self.root

        while stack or node:
            while node:
                stack.append(node)
                node = node.left
            node = stack.pop()
            k -= 1
            if k == 0:
                return node.value
            node = node.right
        return None
```

In [41]:
```
my_ll = BinarySearchTree()
my_ll.insert(47)
my_ll.insert(21)
my_ll.insert(76)
my_ll.insert(18)
my_ll.insert(27)
my_ll.insert(52)
my_ll.insert(82)
```

Out[41]: True

In [42]:
```
my_ll.DFS_post_order()
```

Out[42]: [18, 27, 21, 52, 82, 76, 47]

Hash Table

In [414…
```
class HashTable:
    def __init__(self, size = 7):
        self.data_map = [None] * size

    def __hash(self,key):
        my_hash = 0
        for letter in key:
            my_hash = (my_hash + ord(letter) * 23) % len(self.data_map)
        return my_hash

    def print_table(self):
        for i ,val in enumerate(self.data_map):
            print(i,": ", val)

    def set_item(self,key,value):
        index = self.__hash(key)
        if self.data_map[index] == None:
            self.data_map[index] = []
        self.data_map[index].append([key, value])

    def get_item(self,key):
        index = self.__hash(key)
        if self.data_map[index] is not None:
            for i in range(len(self.data_map[index])):
                if self.data_map[index][i][0] == key:
                    return self.data_map[index][i][1]
        return None

    def keys(self):
        all_keys = []
        for i in range(len(self.data_map)):
            if self.data_map[i] is not None:
                for j in range(len(self.data_map[i])):
                    all_keys.append(self.data_map[i][j][0])
        return all_keys
```

Interview Question

1. Sub Array sum

In [430…
```
def subarray_sum(nums,target):
    hashmap = {}
    currsum = 0
,n in enumerate(nums):
```

```
        currsum += n
        if currsum == target:
            return [0,i]
        if currsum - target in hashmap:
            return [hashmap[currsum - target] + 1 ,i]
        hashmap[currsum] = i
    return []




nums = [1, 2, 3, 4, 5]
target = 9
print ( subarray_sum(nums, target) )

nums = [-1, 2, 3, -4, 5]
target = 0
print ( subarray_sum(nums, target) )

nums = [2, 3, 4, 5, 6]
target = 3
print ( subarray_sum(nums, target) )

nums = []
target = 0
print ( subarray_sum(nums, target) )
```

```
[1, 3]
[0, 3]
[1, 1]
[]
```

Sets

A) Sets are similar to dictionaries except that instead of having key/value pairs they only have the keys but not the values.

B) Sets can only contain unique elements (meaning that duplicates are not allowed).

C)They are useful for various operations such as finding the distinct elements in a collection and performing set operations such as union and intersection.

D)They are defined by either using curly braces {} or the built-in set()

Graph

In [468…

```
class Graph:
    def __init__(self):
        self.adj_list = {}

    def print_graph(self):
        for vertex in self.adj_list:
```

```
            print(vertex, ":" ,self.adj_list[vertex])

    def add_vertex(self,vertex):
        if vertex not in self.adj_list.keys():
            self.adj_list[vertex] = []
            return True
        return False

    def add_edge(self,v1,v2):
        if v1 in self.adj_list.keys() and v2 in self.adj_list.keys():
            self.adj_list[v1].append(v2)
            self.adj_list[v2].append(v1)
            return True
        return False

    def remove_edge(self,v1,v2):
        if v1 in self.adj_list.keys() and v2 in self.adj_list.keys():
            try:
                self.adj_list[v1].remove(v2)
                self.adj_list[v2].remove(v1)
            except ValueError:
                pass
            return True
        return False

    def remove_vertex(self,vertex):
        if vertex in self.adj_list.keys():
            for other_vertex in self.adj_list[vertex]:
                self.adj_list[other_vertex].remove(vertex)
            del self.adj_list[vertex]
            return True
        return False
```

In [469…
```
my_graph = Graph()
my_graph.add_vertex('A')
my_graph.add_vertex('B')
my_graph.add_vertex('C')
my_graph.add_vertex('D')
my_graph.add_edge('A','B')
my_graph.add_edge('B','C')
my_graph.add_edge('C','A')
my_graph.add_edge('A','D')
```

Out[469… True

In [472…
```
my_graph.print_graph()
```

```
A : ['C', 'D']
C : ['A']
D : ['A']
```

Heap

In [487…
```
class MaxHeap:
    def __init__(self):
```

```
        self.heap = []

    def _left_child(self,index):
        return 2*index+1

    def _right_child(self,index):
        return 2 * index + 2

    def _parent(self,index):
        return (index - 1)// 2

    def _swap(self,index1,index2):
        self.heap[index1],self.heap[index2] = self.heap[index2],self.heap[ir

    def insert(self, value):
        self.heap.append(value)
        current = len(self.heap)-1
        while current>0 and self.heap[current] > self.heap[self._parent(curr
            self._swap(current,self._parent(current))
            current = self._parent(current)

    def _sink_down(self, index):
        max_index = index
        while True:
            left_index = self._left_child(index)
            right_index = self._right_child(index)

            if (left_index < len(self.heap) and self.heap[left_index] > self
                max_index = left_index

            if (right_index < len(self.heap) and self.heap[right_index] > se
                max_index = right_index

            if max_index != index:
                self._swap(index, max_index)
                index = max_index
            else:
                return

    def remove(self):
        if len(self.heap) == 0:
            return None
        if len(self.heap) == 1:
            return self.heap.pop()
        max_value = self.heap[0]
        self.heap[0] = self.heap.pop()
        self._sink_down(0)

        return max_value
```

Interview question

1. Heap: Kth Smallest Element in an Array

In [ ]:
```
def find_kth_smallest(nums, k):
    max_heap = MaxHeap()
    for num in nums:
        max_heap.insert(num)
        if len(max_heap.heap) > k:
            max_heap.remove()

    return max_heap.remove()
```

2. Heap: Maximum Element in a Stream

In [495…
```
def stream_max(nums):
    max_heap = MaxHeap()
    max_stream = []

    for num in nums:
        max_heap.insert(num)
        max_stream.append(max_heap.heap[0])

    return max_stream
```

Recursion

A) factorial

In [507…
```
def factorial(n):
    if n == 1:
        return n
    return n * factorial(n-1)
```

In [508…
```
factorial(5)
```

Out[508… 120

B) Recursive contains Binary Search Tree

In [510…
```
def __r_contains(self,current_node,value):
    if current_node == None:
        return False
    if value == current_node.value:
        return True
    if value < cuurent_node.value:
        return self.__r_contains(current_node.left,value)
    if value > current.node.value:
        return self.__r_contains(current_node.right,value)
```

```
def r_contains(self,value):
    return self.__r_contains(self.root,value)
```

C) Recursive insert Binary Search Tree

In [511…
```
def __r_insert(self,current_node,value):
    if current_node == None:
        return Node(value)
    if value < current_node.value:
        current_node.left = self.__r_insert(current_node.left,value)
    if value > current_node.value:
        current_node.right = self.__r_insert(current_node.right,value)
    return current_node

def r_insert(self,value):
    if self.root == None:
        self.root = Node(value)
    self.__r_insert(self.root,value)
```

D) Recursive Delete Binary Search Tree

In [ ]:
```
def __r_delete(self,current_node,value):
    if current_node == None:
        return None
    if value < current_node.value:
        current_node.left = self.__r_delete(self.current_node.left,value)
    elif value > current_node.value:
        current_node.right = self.__r_delete(self.currrent_node.right,value)
    else:
        return current_node #incomplete
```

Sorting

A) Bubble Sort

In [514…
```
def bubble_sort(my_list):
    for i in range(len(my_list)-1,0,-1):
        for j in range(i):
            if my_list[j] > my_list[j+1]:
                my_list[j],my_list[j+1] = my_list[j+1],my_list[j]
    return my_list
list = [4,5,6,3,1,2]
bubble_sort(list)
```

Out[514… `[1, 2, 3, 4, 5, 6]`

B) Selection Sort

In [528…
```
def selection_sort(my_list):
    for i in range(len(my_list)):
        min_index = i
        for j in range(i,len(my_list)):
            if my_list[j]<my_list[min_index]:
```

```
                min_index = j
        my_list[min_index],my_list[i] = my_list[i],my_list[min_index]
    return my_list

list = [4,5,6,3,1,2,5,5,9,8]
selection_sort(list)
```

Out[528… `[1, 2, 3, 4, 5, 5, 5, 6, 8, 9]`

C) Insertion Sort

In [547…
```
def insertion_sort(my_list):
    for i in range(len(my_list)):
        j = i-1
        while j >=0 and my_list[i]<my_list[j]:
            my_list[i],my_list[j] = my_list[j],my_list[i]
            i = j
            j-=1
    return my_list
my_list = [4,5,6,3,1,2,5,5,9,8]
insertion_sort(my_list)
```

Out[547… `[1, 2, 3, 4, 5, 5, 5, 6, 8, 9]`

Interview Question

A) Bubble sort : The method sorts the linked list in place.

In [548…
```
def bubble_sort(self):
        if self.length < 2:
            return
        sorted_until = None
        while self.head.next != sorted_until:
            curr = self.head
            while curr.next != sorted_until:
                next_node = curr.next
                if curr.value > next_node.value:
                    curr.value,next_node.value = next_node.value,curr.value
                curr = curr.next
            sorted_until = curr
```

B) Selection Sort : The method sorts the linked list in place.

In [550…
```
def selection_sort(self):
        if self.length < 2:
            return
        current = self.head
        while current.next is not None:
            smallest = current
            inner_current = current.next
            while inner_current is not None:
                if inner_current.value < smallest.value:
                    smallest = inner_current
                inner_current = inner_current.next
```

```
            if smallest != current:
                current.value, smallest.value = smallest.value, current.valu
            current = current.next
        self.tail = current
```

C) Insertion Sort

In [551…

```
def insertion_sort(self):
        if self.length < 2:
            return

        sorted_list_head = self.head
        unsorted_list_head = self.head.next
        sorted_list_head.next = None

        while unsorted_list_head is not None:
            current = unsorted_list_head
            unsorted_list_head = unsorted_list_head.next

            if current.value < sorted_list_head.value:
                current.next = sorted_list_head
                sorted_list_head = current
            else:
                search_pointer = sorted_list_head
                while search_pointer.next is not None and current.value > se
                    search_pointer = search_pointer.next
                current.next = search_pointer.next
                search_pointer.next = current

        self.head = sorted_list_head
        temp = self.head
        while temp.next is not None:
            temp = temp.next
        self.tail = temp
```

Merge Sort

In [1]:

```
def merge(list1,list2):
    combined = []
    i = 0
    j = 0
    while i<len(list1) and j <len(list2):
        if list1[i] < list2[j]:
            combined.append(list1[i])
            i+=1
        else:
            combined.append(list2[j])
            j+=1
    while i< len(list1):
        combined.append(list1[i])
        i+=1
    while j<len(list2):
        combined.append(list2[j])
        j+=1
    return combined
```

```
def merge_sort(my_list):
    if len(my_list) == 1:
        return my_list
    mid_index = int(len(my_list)/2)
    left = merge_sort(my_list[:mid_index])
    right = merge_sort(my_list[mid_index:])
    return merge(left,right)
```

In [2]:
```
merge_sort([1,2,7,8,3,4,5,6])
```

Out[2]:
```
[1, 2, 3, 4, 5, 6, 7, 8]
```

**Merge Sort Linked List

In [ ]:
```
def merge(self,other_list):
        other_head = other_list.head
        temp = self.head
        dummy = Node(0)
        current = dummy
        while temp is not None and other_head is not None:
            if temp.value < other_head.value:
                current.next = temp
                temp = temp.next
            else:
                current.next = other_head
                other_head = other_head.next
            current = current.next
        while temp is not None:
            current.next = temp
            temp = temp.next
            current = current.next
        while other_head is not None:
            current.next = other_head
            other_head = other_head.next
            current = current.next

        self.head = dummy.next
        self.length +=other_list.length

def find_middle_node(self):
        fast = self.head
        slow = self.head
        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
        return slow

#now merge_sort

    _sort(self):
```

```
    # Base case: If the list is empty or has only one element, it is already
    if self.head is None or self.head.next is None:
        return

    # Find the middle of the linked list
    middle = self.find_middle_node()

    # Split the linked list into two halves
    left_half = LinkedList()
    left_half.head = self.head
    right_half = LinkedList()
    right_half.head = middle.next
    middle.next = None  # Disconnect the two halves

    # Recursively sort the two halves
    left_half.merge_sort()
    right_half.merge_sort()

    # Merge the sorted halves back together
    self.head = self.merge(left_half.head, right_half.head)
```

Pivot & Quick Sort

In [17]:
```
def swap(my_list,index1,index2):
    temp = my_list[index1]
    my_list[index1] = my_list[index2]
    my_list[index2] = temp

def pivot(my_list,pivot_index,end_index):
    swap_index = pivot_index
    for i in range(pivot_index+1,end_index+1):
        if my_list[i]<my_list[pivot_index]:
            swap_index +=1
            swap(my_list,swap_index,i)
    swap(my_list,pivot_index,swap_index)
    return swap_index

def quick_sort_helper(my_list,left,right):
    if left<right:
        pivot_index = pivot(my_list,left,right)
        quick_sort_helper(my_list,left,pivot_index-1)
        quick_sort_helper(my_list,pivot_index+1,right)
    return my_list

def quick_sort(my_list):
    return quick_sort_helper(my_list,0,len(my_list)-1)

my_list = [4,7,3,1,2,6,5,9,3,5,7,8,10,34,56,21,23]
quick_sort(my_list)
```

Out[17]:
```
[1, 2, 3, 3, 4, 5, 5, 6, 7, 7, 8, 9, 10, 21, 23, 34, 56]
```


erview Question

A) List: Remove Element

(Given a list of integers nums and an integer val, write a function remove_element that removes all occurrences of val in the list in-place and returns the new length of the modified list.)

In [ ]:
```
def remove_element(nums,val):
    i = 0
    while i<len(nums):
        if nums[i] == val:
            nums.pop(i) #importent
        else:
            i+=1
    return len(nums)
```

B) Find Max Min

(Write a Python function that takes a list of integers as input and returns a tuple containing the maximum and minimum values in the list.)

In [ ]:
```
def find_max_min(mylist):
    maximum = minimum = mylist[0]
    for num in mylist:
        if num > maximum:
            maximum = num
        elif num < minimum:
            minimum = num
    return maximum,minimum
```

C) Find Longest String

Write a Python function called find_longest_string that takes a list of strings as an input and returns the longest string in the list

In [ ]:
```
def find_longest_string(str_list):
    longest_str = ""
    for str in str_list:
        if len(str) > len(longest_str):
            longest_str = str
    return longest_str
```

D) Remove Duplicates

(Given a sorted list of integers, rearrange the list in-place such that all unique elements appear at the beginning of the list, followed by the duplicate elements. Your function should return the new length of the list containing only unique

elements. Note that you should not create a new list or use any additional data structures to solve this problem. The original list should be modified in-place.)

In [ ]:
```
def remove_duplicates(nums):
    if not nums:
        return 0
    i = 1
    for j in range(1,len(nums)):
        if nums[j]!=nums[j-1]:
            nums[i] = nums[j]
            i+=1
    return i
```

E) Max Profit

You are given a list of integers representing stock prices for a certain company over a period of time, where each element in the list corresponds to the stock price for a specific day.

In [47]:
```
def max_profit(prices):
    l = 0
    r = 1
    maxProfit = 0
    while l<r and r<len(prices):
        while r<len(prices):
            if prices[l] < prices[r]:
                profit = prices[r] - prices[l]
                maxProfit = max(maxProfit,profit)
            else:
                l = r
            r+=1
        l+=1
        r+=1
    return maxProfit
```

F) Rotate

You are given a list of n integers and a non-negative integer k. Your task is to write a function called rotate that takes the list of integers and an integer k as input and rotates the list to the right by k steps.

In [48]:
```
def rotate(nums, k):
    k = k % len(nums)
    nums[:] = nums[-k:] + nums[:-k]
```

G) Max Sub Array

Given an array of integers nums, write a function max_subarray(nums) that finds the contiguous subarray (containing at least one number) with the largest sum and returns its sum. Remember to also account for an array with 0 items.

In [49]:
```python
def max_subarray(nums):
    if len(nums)== 0:
        return 0
    max_sum=current_sum = nums[0]
    for i in range(1,len(nums)):
        current_sum = max(nums[i],current_sum +nums[i])
        max_sum = max(max_sum,current_sum)
    return max_sum
```

In [ ]:
```python

```