DATA STRUCTURES IMPLEMENTATION - **PYTHON**

Table of Contents

JOYDEEP BASU

(For more, please visit my career site)

1. Linked List

1.1 Singly Linked List (SLL)

```
# Create a Singly LinkedList with below Properties (1.Head,2.Tail,3.Length)
# methods (1.Push,2.Pop,3.Shift,4.Unshift,5.Get,6.Set,7.Insert,8.Remove,9.Reverse)
import treevizer
class Node:
    def __init__(self,val):
        self.data = val
        self.next = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.length = 0

    def print(self):
        treevizer.to_png(self.head, structure_type="ll", dot_path="sll.dot",
png_path="sll.png")

    def push(self,nval):
        nd = Node(nval)
        if self.head is None:
            self.head = nd
            self.tail = self.head
        else:
            self.tail.next = nd
            self.tail = nd
        self.length +=1

    def pop(self):
        if self.length == 0: return None
        if self.length == 1:
            self.head = None
            self.tail = None
        else:
            cNode = self.head
            for x in range(1, self.length-1):
                cNode = cNode.next
            self.tail = cNode
            cNode.next =None

        self.length -=1
        return self

    def get(self, index):
        if self.length > index and index >= 0:
            cNode = self.head
            if index == 0: return cNode.data
            for x in range(1, index+1):
                cNode = cNode.next
            return cNode.data
        return None
```

```
    def set(self, index,nval):
        if self.length > index and index >= 0:
            cNode = self.head
            if index == 0:
                cNode.data = nval
                return True
            for x in range(1, index+1):
                cNode = cNode.next
            cNode.data = nval
            return True
        return False

    def reverse(self):
        prevNode, curNode = None, self.head
        self.tail = curNode
        while (curNode):
            nextNode = curNode.next
            curNode.next = prevNode
            prevNode = curNode
            curNode = nextNode
        self.head = prevNode


    def reversePos(self,start,end):
        if start > 0 and end <self.length -1:
            sNode = eNode = self.head
            for x in range(1,start):
                sNode = sNode.next
            for x in range(1,end+1):
                eNode = eNode.next

            headNode = sNode
            tailNode = eNode.next

            prevNode = tailNode
            curNode = headNode.next
            while(curNode is not tailNode):
                nextNode = curNode.next
                curNode.next = prevNode
                prevNode = curNode
                curNode = nextNode
            headNode.next = prevNode

    def printSll(self):
        cNode = self.head
        lst = []
        while(cNode):
            lst.append(cNode.data)
            cNode = cNode.next
        print(lst)

if __name__ == '__main__':
    sll = SinglyLinkedList()
    for x in range(1,11): sll.push(x)
    sll.print()

    sll.printSll()
    sll.reversePos(1,7)
    sll.printSll()
    print('Head [', sll.head.data,']')
    print('Tail [',sll.tail.data,']')
```

1.2 Doubly Linked List (DLL)

```
# Create a Doubly LinkedList with below properties (1.Head,2.Tail,3.Length)
# Problem: DLL FLattening with ChildDLLs or SubChildDLLs
import treevizer

class Node:
    def __init__(self,val=None):
        self.next = None
        self.prev = None
        self.data = val
        self.child = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.length = 0

    def print(self):
        treevizer.to_png(self.head, structure_type="ll", dot_path="dll.dot",
png_path="dll.png")

    def addChildDLL(self,index,chlDLL):
        if index<0 or index>self.length-1: return None
        if index==0:
            cNode = self.head
        elif index==self.length-1:
            cNode = self.tail
        else:
            cNode = self.head
            for x in range(1,index+1): cNode = cNode.next

        cNode.child = chlDLL.head

    def flattenDLL(self):
        psNode = self.head
        while(psNode):
            if psNode.child is not None:
                peNode = psNode.next
                csNode = psNode.child
                while(csNode): ceNode = csNode; csNode = csNode.next
                # ------------------------
                psNode.next = psNode.child
                psNode.child.prev = psNode
                # ------------------------
                ceNode.next = peNode
                peNode.prev = ceNode
                # ------------------------
                psNode.child = None
                #------------------------
            psNode = psNode.next
```

```
    def push(self,nval):
        nd = Node(nval)
        if self.head is None:
            self.head = nd
            self.tail = self.head
        else:
            self.tail.next = nd
            nd.prev = self.tail
            self.tail = nd
        self.length +=1

    def printDll(self):
        # nd=self.head
        # while nd: print(nd.stage,end='<->'); nd=nd.next
        # print("\n")
        print(self.buildMap(self.head))

    def buildMap(self,pNode):
        tempMap = {}
        while(pNode):
            if pNode.child is not None:
                tempMap[pNode.data] = self.buildMap(pNode.child)
            else:
                tempMap[pNode.data] = {}
            pNode = pNode.next
        return tempMap

if __name__ == '__main__':
    dll = DoublyLinkedList(); dll1 = DoublyLinkedList(); dll3 = DoublyLinkedList()
    dll11 = DoublyLinkedList(); dll33 = DoublyLinkedList()

    for x in range(0,10): dll.push(x)
    for x in range(10,15): dll1.push(x)
    for x in range(30,40): dll3.push(x)
    for x in range(120,125): dll11.push(x)
    for x in range(330,333): dll33.push(x)

    dll1.addChildDLL(2,dll11)
    dll3.addChildDLL(3,dll33)
    dll.addChildDLL(1,dll1); dll.addChildDLL(3,dll3)

    dll.printDll()
    dll.flattenDLL()    # DLL Flattening Test
    dll.printDll()
```

2. Stack

2.1 Using LinkedList (Node)

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

class Stack:
    def __init__(self):
        self.top = None
        self.size = 0

    def is_empty(self):
        return self.top is None

    def push(self, value):
        nd = Node(value)
        nd.next = self.top
        self.top = nd
        self.size+=1

    def pop(self):
        if self.top is None: return None
        value = self.top.value
        self.top = self.top.next
        self.size -= 1
        return value

    def peek(self):
        if self.top is None: return None
        return self.top.value

    def length(self): return self.size

    def print(self):
        cnode = self.top
        while cnode:
            print(cnode.value,end=" -> ")
            cnode=cnode.next
        print("None")

if __name__ == '__main__':
    stk = Stack()
    for i in range(1,11,2): stk.push(i)
    stk.print()
    stk.push(100)
    stk.print()
    print(stk.length())
    print(stk.peek())
    stk.pop()
    stk.print()
    while stk.length(): val = stk.pop(); print(val)
```

2.2 Using List

```
class stack():
    def __init__(self):
        self.arr = []
        self.size = 0

    def push(self,val):
        self.arr.append(val)
        self.size = len(self.arr)

    def pop(self):
        self.arr.pop()
        self.size = len(self.arr)

    def peek(self):
        if self.size >0:
            return (self.arr[self.size - 1])
        else:
            return None

    def lookup(self, val):
        print(self.arr.index(val)) if val in self.arr else print('Not found')

    def printStack(self):
        print(self.arr)


if __name__ == "__main__":
    stk = stack()
    stk.push('Joy')
    stk.push('deep')
    stk.push('Basu')
    stk.lookup('Basu')
    print(stk.peek())
    stk.pop()
    print(stk.peek())
```

3. Queue

3.1 Using LinkedList

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

class Queue:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def enqueue(self,val):
        nd = Node(val)
        if (self.head is None):
            self.head = nd
            self.tail = self.head
        else:
            self.tail.next = nd
            self.tail = nd
        self.size +=1

    def dequeue(self):
        if self.size == 0: return None
        val = self.head.val
        if self.head.next:
            self.head = self.head.next
        else:
            self.head = None
            self.tail = None
        self.size -= 1
        return val

    def peek(self):
        return self.head.val if self.head else None

# ---------------- Optional ----------------------
    def printQ(self):
        if self.size == 0: return None
        cNode = self.head
        while(cNode):
            print(cNode.val, end=' <- ')
            cNode = cNode.next
        print('\n')

if __name__ == '__main__':
    q = Queue()
    q.enqueue('Joy'); q.enqueue('Deep'); q.enqueue('Basu')
    print(q.peek())
    q.printQ()

print(q.dequeue());print(q.dequeue());print(q.dequeue());print(q.dequeue());print(q.dequ
eue());
    q.printQ()
    q.enqueue('Joy'); print(q.peek()); print(q.size)
    print(q.lookup('Basu'))
```

3.2 Using Two Stacks

```
class queueFromStack():
    def __init__(self):
        self.stk1 = [] # stack enqueue or push operation
        self.stk2 = [] # stack dequeue or pop operation
        self.size = 0

    def isEmpty(self):
        return self.size==0

    def enqueue(self,item):
        self.stk1.append(item)
        self.size +=1

    def dequeue(self):
        if self.size==0: return None
        if not self.stk2:
            while self.stk1: self.stk2.append(self.stk1.pop())
        self.size -=1
        return self.stk2.pop()

    def peek(self):
        if self.size == 0: return None
        if not self.stk2:
            while self.stk1: self.stk2.append(self.stk1.pop())
        return self.stk2[-1]

    def print(self):
        print(" <- ".join(self.stk1))


if __name__ == '__main__':
    qs = queueFromStack()
    for i in range(3): qs.enqueue(i)
    print("Stack1: ", qs.stk1, "\nStack2: ", qs.stk2)
    print(qs.dequeue())
    print("Stack1: ", qs.stk1, "\nStack2: ", qs.stk2)
    print(qs.isEmpty())
    qs.enqueue(3);
    qs.enqueue(4);
    print("Stack1: ", qs.stk1, "\nStack2: ", qs.stk2)
    print(qs.dequeue())
    print("Stack1: ", qs.stk1, "\nStack2: ", qs.stk2)
    print(qs.dequeue())
    print("Stack1: ", qs.stk1, "\nStack2: ", qs.stk2)
    print(qs.dequeue())
    print("Stack1: ", qs.stk1, "\nStack2: ", qs.stk2)
    print(qs.dequeue())
    print("Stack1: ", qs.stk1, "\nStack2: ", qs.stk2)
    print(qs.dequeue())
    print("Stack1: ", qs.stk1, "\nStack2: ", qs.stk2)
    ##########################################
    for i in range(3): qs.enqueue(i)
    qs.dequeue()
    print(qs.peek())
    print(qs.isEmpty())
```

3.3 Using Deque

```
from collections import deque

class Queue:
    def __init__(self):
        self.q = deque()

    def enqueue(self, item):
        self.q.append(item)

    def dequeue(self):
        return self.q.popleft() if self.q else None

    def peek(self):
        return self.q[0] if self.q else None

    def lookup(self, key):
        for item in self.q:
            if item == key: return True
        return False

    def print(self): print("<-".join(map(str,self.q)))


q = Queue()
for x in range(1,10): q.enqueue(x)
q.print()
q.dequeue(); print(q.dequeue())
q.print()
print(q.peek())
print(q.lookup(7))
print(q.peek())
```

4. Binary Search Tree (BST)

4.1 Using Node class

```
# Design a Binary Search Tree (BST) with methods
# 1. Insert, 2. Lookup, 3. Remove, 4. BFS, 5. DFS_InOrder, 6. DFS_PreOrder, 7.
DFS_PostOrder
from tree import drawTree
from collections import deque

class binaryNode:
    def __init__(self, val):
        self.value = val
        self.left = None
        self.right = None
#########################################################################
class binarySearchTree:
    def __init__(self): self.root = None
    def print(self): drawTree(self.root)

    def insert(self, val):
        bNode = binaryNode(val)
        if self.root is None: self.root = bNode
        curNode = self.root
        while curNode:
            if bNode.value == curNode.value: return
            if bNode.value > curNode.value:
                if curNode.right: curNode = curNode.right; continue
                curNode.right = bNode; return
            else:
                if curNode.left: curNode = curNode.left; continue
                curNode.left = bNode; return

    def search(self, val):
        if self.root is None: return None
        curNode = self.root
        while curNode:
            if curNode.value == val: return curNode
            if val > curNode.value: curNode = curNode.right; continue
            if val < curNode.value: curNode = curNode.left; continue
        return None

    def remove(self, input_val):  ## Not working ##
        def helper(node, searchVal):
            if node is None: return None
            if searchVal < node.value:
                node.left = helper(node.left, searchVal)
            elif searchVal > node.value:
                node.right = helper(node.right, searchVal)
            else:
                if node.left is None: return node.right
                if node.right is None: return node.left
                # Node with two children: Get the in-order successor (smallest in the
right subtree)
                min_node = node.right
                while min_node.left: min_node = min_node.left
                node.right = helper(min_node, min_node.value)
            return node

        self.root = helper(self.root, input_val)
```

```
    def BFS(self):
        q, arr = deque(), []
        if self.root is None: return []
        q.append(self.root)
        while q:
            curNode = q.popleft()
            arr.append(curNode.value)
            if curNode.left: q.append(curNode.left)
            if curNode.right: q.append(curNode.right)
        return arr

    def DFS_InOrder(self):
        def traverseInOrder(node, arr):  # InOrder: Left Node (recursive) -> Parent Node
-> Right Node (recursive)
            if node.left: traverseInOrder(node.left, arr)
            arr.append(node.value)
            if node.right: traverseInOrder(node.right, arr)
            return arr
        return traverseInOrder(self.root, [])  # recursive call from stage

    def DFS_PreOrder(self):
        def traversePreOrder(node, arr):  # PreOrder: Parent Node -> Left Node (R) ->
Right Node (R)
            arr.append(node.value)
            if node.left: traversePreOrder(node.left, arr)
            if node.right: traversePreOrder(node.right, arr)
            return arr
        return traversePreOrder(self.root, [])  # recursive call from stage

    def DFS_PostOrder(self):
        def traversePostOrder(node, arr):  # PreOrder: Left Node (R) -> Right Node (R) -
> Parent Node
            if node.left: traversePostOrder(node.left, arr)
            if node.right: traversePostOrder(node.right, arr)
            arr.append(node.value)
            return arr
        return traversePostOrder(self.root, [])  # recursive call from stage

######################################################################

if __name__ == '__main__':
    myBST = binarySearchTree()
    for num in [20, 40, 8, 45, 43, 36, 19, 1, 47, 29, 42, 44, 30]: myBST.insert(num)
    # for num in [x for x in range(10)]: myBST.insert(num)
    myBST.print()
    # exit(0)
    print('BFS list:', myBST.BFS())
    print('DFS In Order: ', myBST.DFS_InOrder())
    print('DFS Pre Order: ', myBST.DFS_PreOrder())
    print('DFS Post Order: ', myBST.DFS_PostOrder())
    print('Looking up for value (28):', myBST.search(30))

    for num in [40, 42, 20]: myBST.remove(num)
    myBST.print()
    print(myBST.depth())
```

5. Heap

5.1 Max Heap

```
from draw_arr2tree import arr2tree

class MaxHeap:
    def __init__(self): self.heap = []

    def insert(self,value):
        self.heap.append(value)
        self._heapifyUp(len(self.heap)-1)

    def extractMax(self):
        if len(self.heap) == 0: return None
        if len(self.heap) == 1: return self.heap.pop()
        maxVal = self.heap[0]
        self.heap[0] = self.heap.pop()
        self._heapifyDown(0)
        return maxVal

    def Max_heapifyArr(self,arr):
        self.heap = list(arr)
        for index in range((len(arr)//2)-1,-1,-1):  # reverse looping on "non-leaf"
nodes --> heapifyDown()
            self._heapifyDown(index)

    def _swapIndex(self, idx1, idx2): self.heap[idx1], self.heap[idx2] =
self.heap[idx2], self.heap[idx1]

    def _heapifyUp(self,index):     # Any child (left/right) to its parent comparison
        pIndex = (index-1)//2
        if index>0 and self.heap[index] > self.heap[pIndex]:
            self._swapIndex(index,pIndex)
            self._heapifyUp(pIndex)

    def _heapifyDown(self,index):   # parent to both children (left,right) comparison
        largest = index
        lcIndex, rcIndex = 2*index+1, 2*index+2
        if lcIndex < len(self.heap) and self.heap[lcIndex] > self.heap[largest]:
largest=lcIndex
        if rcIndex < len(self.heap) and self.heap[rcIndex] > self.heap[largest]: largest
= rcIndex
        if index!=largest:
            self._swapIndex(index, largest)
            self._heapifyDown(largest)

    def print(self): print(self.heap)

if __name__ == "__main__":
    mxhp = MaxHeap()
    mxhp.Max_heapifyArr([3, 8, 5, 2, 7, 6, 4, 1]); mxhp.print(); arr2tree(mxhp.heap)

    mxhp.insert(0); mxhp.print(); arr2tree(mxhp.heap)
    mxhp.insert(100); mxhp.print(); arr2tree(mxhp.heap)

    print(mxhp.extractMax()); mxhp.print(); arr2tree(mxhp.heap)
    print(mxhp.extractMax()); mxhp.print(); arr2tree(mxhp.heap)
```

5.2 Min Heap

```
from draw_arr2tree import arr2tree

class MinHeap:
    def __init__(self): self.heap = []

    def insert(self,value):
        self.heap.append(value)
        self._heapifyUp(len(self.heap)-1)

    def extractMin(self):
        if len(self.heap) == 0: return None
        if len(self.heap) == 1: return self.heap.pop()
        minVal = self.heap[0]
        self.heap[0] = self.heap.pop()
        self._heapifyDown(0)
        return minVal

    def Min_heapifyArr(self,arr):
        self.heap = list(arr)
        for index in range((len(arr)//2)-1,-1,-1):  # reverse looping on "non-leaf"
nodes --> heapifyDown()
            self._heapifyDown(index)

    def _swapIndex(self, idx1, idx2): self.heap[idx1], self.heap[idx2] =
self.heap[idx2], self.heap[idx1]

    def _heapifyUp(self,index):     # Any child (left/right) to its parent comparison
        pIndex = (index-1)//2
        if index>0 and self.heap[index] < self.heap[pIndex]:
            self._swapIndex(index,pIndex)
            self._heapifyUp(pIndex)

    def _heapifyDown(self,index):   # parent to both children (left,right) comparison
        smallest = index
        lcIndex, rcIndex = 2*index+1, 2*index+2
        if lcIndex < len(self.heap) and self.heap[lcIndex] < self.heap[smallest]:
smallest=lcIndex
        if rcIndex < len(self.heap) and self.heap[rcIndex] < self.heap[smallest]:
smallest = rcIndex
        if index!=smallest:
            self._swapIndex(index, smallest)
            self._heapifyDown(smallest)

    def print(self): print(self.heap)

if __name__ == '__main__':
    mnhp = MinHeap()
    mnhp.Min_heapifyArr([3,8,5,2,7,6,4,1]);  mnhp.print(); arr2tree(mnhp.heap)

    mnhp.insert(0); mnhp.print(); arr2tree(mnhp.heap)
    mnhp.insert(100); mnhp.print(); arr2tree(mnhp.heap)

    print(mnhp.extractMin()); mnhp.print(); arr2tree(mnhp.heap)
    print(mnhp.extractMin()); mnhp.print(); arr2tree(mnhp.heap)
```

6. Graph

```
class graph:
    def __init__(self): self.graph = {}
    def print(self): print(self.graph)
    def sort(self): return {k:v for k,v in sorted(self.graph.items())}

    def addVertex(self, key):
        if key not in self.graph: self.graph[key] = []

    def removeVertex(self, key):
        if key in self.graph:  # removing the node
            self.graph.pop(key)
            for v in self.graph:  # Removing all connections of deleted node
                if key in self.graph[v]: self.graph[v].remove(key)

    def addEdge(self, fromKey, toKey):
        if fromKey == toKey: return
        if fromKey in self.graph and toKey in self.graph:
            self.graph[fromKey].append(toKey)
            self.graph[toKey].append(fromKey)

    def deleteEdge(self, fromKey, toKey):
        if fromKey == toKey: return None
        if fromKey in self.graph and toKey in self.graph:
            if toKey in self.graph[fromKey]: self.graph[fromKey].remove(toKey)
            if fromKey in self.graph[toKey]: self.graph[toKey].remove(fromKey)

    def BFS(self):
        q, arr, visited = Queue(), [], []
        q.enqueue(min(self.graph.keys()))
        node = None
        while q.size > 0:
            node = q.dequeue()
            arr.append(node); visited.append(node)
            for key in self.graph[node]:
                if key not in visited: q.enqueue(key)
        return print('BFS order: ', arr)

    def DFS(self):
        def traversalDFS(node, arr, visited):
            if node not in visited:
                arr.append(node); visited.append(node)
                for key in self.graph[node]: traversalDFS(key, arr, visited)
            return arr
        return print('DFS order:',traversalDFS(min(self.graph.keys()),[],[]))

if __name__ == '__main__':
    grp = graph()
    grp.addVertex(10); grp.addVertex(100); grp.print()
    grp.removeVertex(100); grp.removeVertex(89); grp.print()

    for item in [0, 1, 2, 3, 4, 5, 6, 7, 8]: grp.addVertex(item)
    grp.addEdge(0, 1); grp.addEdge(0, 3); grp.addEdge(3, 2); grp.addEdge(3, 4)
    grp.addEdge(3, 5); grp.addEdge(2, 8); grp.addEdge(4, 6); grp.addEdge(6, 7)
    grp.print();  grp.deleteEdge(4, 100); grp.removeVertex(4); grp.print()

    sgrp = grp.sort(); print(f"Sorted Graph: {sgrp}")
    grp.BFS(); grp.DFS()
```

7. Trie

```
import treevizer
class TrieNode:
    def __init__(self,val,stop=False):
        self.children={}
        self.value = val
        self.stop = False

class Trie:
    def __init__(self):
        self.root=TrieNode(val="")
        self.length=0  # This tracks the total number of nodes of a Trie
        self.keys = 0 # This tracks unique keys/words count of a Trie

    def print(self): treevizer.to_png(self.root, structure_type="trie",
dot_path="trie.dot", png_path="trie.png")
    def wordsCount (self): return self.keys

    ########################################################################
    def insert(self,word):
        node = self.root
        for ch in word:
            if ch not in node.children: node.children[ch] = TrieNode(ch); self.length+=1
            node=node.children[ch]
        if not node.stop: node.stop=True; self.keys += 1

    ###################### Word/Prefix search ######################
    def search(self,key,type):
        node = self.root
        for ch in key:
            if ch not in node.children: return False
            node=node.children[ch]
        # if type='prefix' always True returned, but for word search node.stop value is
returned
        return node.stop if type=='word' else True

    ##############################################################
    def delete(self,key):
        stack, node = [], self.root
        for ch in key:
            if ch not in node.children: return False
            node=node.children[ch]
            stack.append(node)
        if node.stop: node.stop = False; self.keys -=1
        if node.children: return

        # to recursively delete parent node hierarchy if no siblings/other key found in
the hierarchy
        stack.insert(0,self.root)
        while stack:
            node = stack.pop()
            pNode = stack[-1]
            del pNode.children[node.value]; self.length -=1
            # exit loop if siblings (pNode.children) or other key (pNode.stop) is
present
            if (pNode is self.root) or pNode.stop or pNode.children: return

    ################## Autofill Suggestions ##########################
    def autofill_suggestions(self,startswith):
        node, stack, suggestions = self.root,[],[]
```

```
        for ch in startswith:
            if ch not in node.children: return []
            node = node.children[ch]

        for ch in node.children: stack.append(("",node.children[ch]))
        while stack:
            prefix, node = stack.pop()
            prefix += node.value
            if node.stop: suggestions.append(prefix)
            for ch in node.children: stack.append((prefix, node.children[ch]))
        return suggestions
################################################################################

if __name__ == '__main__':
    t = Trie()
    for wd in
['Cap','Capstone1','Capstone2','Capstone3','Capital','Caps','Caterpillar']: t.insert(wd)
    t.print()

    print(f"Word (Capstone) search: {t.search('Capstone', 'word')}")
    print(f"Word (Capstone1) search: {t.search('Capstone1', 'word')}")
    print(f"Prefix/Starts_with (Capta) search: {t.search('Capta', 'prefix')}")
    print(f"Prefix/Starts_with  (Capita) search: {t.search('Capita', 'prefix')}")

    print(f"Distinct Key count: {t.wordsCount()}")
    print(f"Total Node count: {t.length}")
    print(f"Prompting that starts with: 'Cap' >> {t.autofill_suggestions('Cap')}")

    t.delete('Capital'); t.print()

    # print(t.search('Cats'))
    # print(t.search('Catz'))
    # print(t.length)
```

8. Matrix

```
class Matrix:
    def __init__(self, rows, cols):
        self.rows = rows
        self.cols = cols
        self.data = [[0 for _ in range(cols)] for _ in range(rows)]

    def get(self, row, col):
        if 0 <= row < self.rows and 0 <= col < self.cols:
            return self.data[row][col]
        return None

    def set(self, row, col, value):
        if 0 <= row < self.rows and 0 <= col < self.cols:
            self.data[row][col] = value
        else:
            print("Index out of bounds")
```

```
    def transpose(self):
        # Time Complexity: O(rows * cols)
        transposed_matrix = Matrix(self.cols, self.rows)
        for i in range(self.rows):
            for j in range(self.cols):
                transposed_matrix.set(j, i, self.data[i][j])
        return transposed_matrix

    def add(self, other_matrix):
        # Time Complexity: O(rows * cols)
        if self.rows != other_matrix.rows or self.cols != other_matrix.cols:
            print("Matrix dimensions do not match for addition.")
            return None

        result_matrix = Matrix(self.rows, self.cols)
        for i in range(self.rows):
            for j in range(self.cols):
                result_matrix.set(i, j, self.data[i][j] + other_matrix.get(i, j))
        return result_matrix

    def multiply(self, other_matrix):
        # Time Complexity: O(self.rows * self.cols * other_matrix.cols)
        if self.cols != other_matrix.rows:
            print("Matrix dimensions are not compatible for multiplication.")
            return None

        result_matrix = Matrix(self.rows, other_matrix.cols)
        for i in range(self.rows):
            for j in range(other_matrix.cols):
                dot_product = 0
                for k in range(self.cols):
                    dot_product += self.data[i][k] * other_matrix.get(k, j)
                result_matrix.set(i, j, dot_product)
        return result_matrix

    def display(self):
        for row in self.data:
            print(row)

# Example usage:
matrix1 = Matrix(2, 3)
matrix1.set(0, 0, 1)
matrix1.set(0, 1, 2)
matrix1.set(0, 2, 3)
matrix1.set(1, 0, 4)
matrix1.set(1, 1, 5)
matrix1.set(1, 2, 6)

matrix2 = Matrix(3, 2)
matrix2.set(0, 0, 7)
matrix2.set(0, 1, 8)
matrix2.set(1, 0, 9)
matrix2.set(1, 1, 10)
matrix2.set(2, 0, 11)
matrix2.set(2, 1, 12)

result = matrix1.multiply(matrix2)
result.display()
```

9. Sorting Algorithms

9.1 Bubble Sort

```
def bubbleSort(self, arr):
    for x in range(0, len(arr) - 1):  # Outer loop to reduce right boundary of inner
loop by 1
        end = len(arr)
        for pt in range(1, end):  # Inner loop to compare adjacent elements (always
checks from beginning)
            if arr[pt - 1] > arr[pt]:
                arr[pt - 1], arr[pt] = arr[pt], arr[pt - 1]  # Swapping adjacent
elements
        end = end - 1
    return arr
```

9.2 Selection Sort

```
def selectionSort(self, arr):
    for pt1 in range(0, len(arr) - 1):  # Outer loop to increase left boundary of inner
loop by 1
        smallest = pt1  # pt1 starts with 0, then increases until length of array - 1
        for pt2 in range(pt1 + 1, len(arr)):  # loop to find the smallest number after
arr[pt1] to replace with arr[pt1]
            if arr[pt2] < arr[smallest]: smallest = pt2
        if arr[smallest] < arr[pt1]:
            arr[pt1], arr[smallest] = arr[smallest], arr[pt1]
    return arr
```

9.3 Insertion Sort

```
def insertionSort(self, arr):
    for pt1 in range(1, len(arr)):
        for pt2 in range(0, pt1):
            if arr[pt1] < arr[pt2]:
                arr.insert(pt2, arr.pop(pt1))
                break
    return arr
```

9.4 Merge Sort

```
def mergeSort(self, arr):
    def merge(lArr, rArr):
        lpt, rpt, result = 0, 0, []
        while (lpt < len(lArr) and rpt < len(rArr)):
            if lArr[lpt] < rArr[rpt]:
                result.append(lArr[lpt]); lpt += 1
            else:
                result.append(rArr[rpt]); rpt += 1
        while (lpt < len(lArr)): result.append(lArr[lpt]); lpt += 1
        while (rpt < len(rArr)): result.append(rArr[rpt]); rpt += 1
        return result
    #-----------------------------------------------------------------------------
    if len(arr) <= 1: return arr
    mid = len(arr) // 2
    left = self.mergeSort(arr[0:mid])
    right = self.mergeSort(arr[mid:])
    return merge(left, right)
```

9.5 Quick Sort

```
def quickSort(self, arr):
    if len(arr) <= 1: return arr
    leftArr, rightArr, pivot = [], [], arr.pop()
    for x in arr: leftArr.append(x) if x < pivot else rightArr.append(x)
    return self.quickSort(leftArr) + [pivot] + self.quickSort(rightArr)
```

9.6 Radix Sort

```
def radixSort(self, arr):
    maxPos = max([len(str(y)) for y in arr])
    def getDigit(num, pos):  # Helper function to get digit of specified position
        return 0 if len(str(num)) < pos else int(str(num)[len(str(num)) - pos])

    for pos in range(1, maxPos + 1):
        resDict = {x: [] for x in range(10)}  # Resetting dictionary to empty list for
keys (0-9)
        ####################################
        while arr:
            num = arr.pop(0)
            resDict[getDigit(num, pos)].append(num)  # Pop & add array element to dict
(arr positional values = dictionary key)
        ####################################
        for key in resDict: arr = arr + resDict[key]  # Re-adding elements to arr as per
dict key order
    return arr
```

9.7 Heap Sort

```
def heapsort(self, arr):
    n = len(arr)
    hp.heapify(arr)
    return [hp.heappop(arr) for i in range(n)]
```