merged.py

```
BinarySearchTree.py
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    def insert(self, newData):
        if newData <= self.data:
            if self.left is None:
                self.left = Node(newData)
            else:
                self.left.insert(newData)
        else:
            if self.right is None:
                self.right = Node(newData)
            else:
                self.right.insert(newData)

    def contains(self, target):
        if target == self.data:
            return True
        elif target < self.data:
            if self.left is None:
                return False
            else:
                return self.left.contains(target)
        else:
            if self.right is None:
                return False
            else:
                return self.right.contains(target)

    def printInOrder(self):
        if self.left:
            self.left.printInOrder()
        print(self.data)
        if self.right:
            self.right.printInOrder()

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

    def insert(self, data):
        if self.root is None:
            self.root = Node(data)
        else:
            self.root.insert(data)

    def contains(self, target):
        if self.root is None:
            return False
        return self.root.contains(target)

    def print(self):
        if self.root is None:
            return
        self.root.printInOrder()

myBinarySearchTree = BinarySearchTree()

print(myBinarySearchTree.contains(1))
myBinarySearchTree.insert(20)
myBinarySearchTree.insert(2)
myBinarySearchTree.insert(3)
myBinarySearchTree.insert(40)
myBinarySearchTree.insert(5)
print(myBinarySearchTree.contains(1))
```

```
print(myBinarySearchTree.contains(5))
print(myBinarySearchTree.contains(2))
print(myBinarySearchTree.contains(10))
print(myBinarySearchTree.print())

DisjointSet.py
class DisjointSet:
    def __init__(self, items):
        self.items = items
        self.parent = [i for i in range(len(items))]
        self.rank = [0]*len(self.items)

    def findRoot(self, i):
        if self.parent[i] != i:
            self.parent[i] = self.findRoot(self.parent[i])
        return self.parent[i]

    def union(self, i, j):
        iRoot = self.findRoot(i)
        jRoot = self.findRoot(j)

        if iRoot == jRoot:
            return

        if self.rank[iRoot] < self.rank[jRoot]:
            self.parent[iRoot] = jRoot
        elif self.rank[iRoot] > self.rank[jRoot]:
            self.parent[jRoot] = iRoot
        else:
            self.parent[jRoot] = iRoot
            self.rank[iRoot] += 1


myDisjointSet = DisjointSet([1,2,3,4])
myDisjointSet.union(0,1)

for i in range(4):
    print(myDisjointSet.findRoot(i))

Heap.py
class Heap:
    def __init__(self):
        self.capacity = 1
        self.size = 0
        self.items = [None]*100

    def __len__(self):
        return self.size

    def increaseCapacity(self):
        if self.size == self.capacity:
            self.items += [None]*self.capacity
            self.capacity *= 2

    def isEmpty(self):
        return len(self) == 0

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

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

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

    def getLeftChild(self, index):
        return self.items[self.getLeftChildIndex(index)]

    def getRightChild(self, index):
        return self.items[self.getRightChildIndex(index)]

    def getParent(self, index):
```

```
        return self.items[self.getParentIndex(index)]

    def hasLeftChild(self, index):
        return self.getLeftChildIndex(index) <= len(self)

    def hasRightChild(self, index):
        return self.getRightChildIndex(index) <= len(self)

    def hasParent(self, index):
        return self.getParentIndex(index) >= 0

    def swap(self, i, j):
        tmp = self.items[i]
        self.items[i] = self.items[j]
        self.items[j] = tmp

    def put(self, data):
        self.increaseCapacity()
        self.items[self.size] = data
        self.size += 1
        self.rise(self.size-1)

    def peek(self):
        assert not self.isEmpty(), "Heap is empty"
        return self.items[0]

    def poll(self):
        assert not self.isEmpty(), "Heap is empty"
        item = self.items[0]
        self.items[0] = self.items[self.size-1]
        self.size -= 1
        self.sink(0)
        return item

    def rise(self, index):
        while self.hasParent(index) and self.getParent(index) > self.items[index]:
            self.swap(index, self.getParentIndex(index))
            index = self.getParentIndex(index)

    def sink(self, index):
        while self.hasLeftChild(index):
            smallerChildIndex = self.getLeftChildIndex(index)

            if self.hasRightChild(index) and self.getRightChild(index) < self.getLeftChildIndex(index):
                smallerChildIndex = self.getRightChildIndex(index)

            if self.items[index] <= self.items[smallerChildIndex]:
                break
            else:
                self.swap(index, smallerChildIndex)
            index = smallerChildIndex

myHeap = Heap()
myHeap.put(5)
myHeap.put(4)
myHeap.put(3)
myHeap.put(2)
myHeap.put(1)
print(myHeap.poll())
print(myHeap.poll())
print(myHeap.poll())
print(myHeap.poll())
print(myHeap.poll())

LinkedList.py
class SinglyNode:
    def __init__(self, data):
        self.data = data
        self.next = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None
```

```
        self.size = 0

    def append(self, data):
        newNode = SinglyNode(data)
        cur = self.head

        self.size += 1

        if not cur:
            self.head = newNode
            return

        while cur.next != None:
            cur = cur.next

        cur.next = newNode

    def prepend(self, data):
        newNode = SinglyNode(data)
        cur = self.head

        self.size += 1

        if not cur:
            self.head = newNode
            return

        newNode.next = self.head
        self.head = newNode

    def __str__(self):
        outputString = "["

        cur = self.head

        if cur:
            outputString += str(cur.data)
            cur = cur.next
            while cur != None:
                outputString += ", " + str(cur.data)
                cur = cur.next

        outputString += "]"
        return outputString

myList = SinglyLinkedList()

print(myList)
myList.append(1)
myList.append(2)
myList.append(3)
myList.append(4)
myList.prepend(5)
myList.prepend(6)
myList.prepend(7)
myList.prepend(8)
print(myList)

Queue.py
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    def __str__(self):
        return str(self.data)

class Queue:
    def __init__(self):
        self.front = None
        self.back = None

    def isEmpty(self):
```

```
        return self.front is None

    def append(self, data):
        newNode = Node(data)
        if self.front is None:
            self.front = newNode
            self.back = newNode
            return

        # otherwise, append to the back, and set the new node as the back
        self.back.next = newNode
        self.back = newNode

    def peek(self):
        return self.front

    def serve(self):
        if self.isEmpty():
            return None
        data = self.front.data
        self.front = self.front.next
        if self.front is None:
            self.back = None
        return data

    def __str__(self):
        outputString = "["
        cur = self.front

        if cur is not None:
            outputString += str(cur)
            cur = cur.next
            while cur != None:
                outputString += ", " +str(cur)
                cur = cur.next

        outputString += "]"
        return outputString

myQueue = Queue()

print(myQueue)
myQueue.append(1)
myQueue.append(2)
myQueue.append(3)
myQueue.append(4)
myQueue.append(5)
print(myQueue)
myQueue.serve()
print(myQueue)

Stack.py
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    def __str__(self):
        return str(self.data)

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

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

    def push(self, data):
        # otherwise, create node that points to current top, set as top
        newNode = Node(data)
        newNode.next = self.top
        self.top = newNode
```

```
    def peek(self):
        if self.isEmpty():
            return None
        return self.top.data

    def pop(self):
        # if empty, return None
        if self.isEmpty():
            return None

        # otherwise, return top's data and set top to next
        data = self.top.data
        self.top = self.top.next

        return data

    def __str__(self):
        outputString = "Stack:\n"

        cur = self.top

        if cur is not None:
            while cur is not None:
                outputString += str(cur) + "\n"
                cur = cur.next

        return outputString

myStack = Stack()

print(myStack)
myStack.push(1)
myStack.push(2)
myStack.push(3)
myStack.push(4)
myStack.push(5)
print(myStack)
myStack.pop()
print(myStack)

Trie.py
def charToIndex(char):
    return ord(char) - ord('a')

class Node:
    def __init__(self):
        self.children = [None]*26
        self.noOfChildren = 0

    def getNode(self, char):
        return self.children[charToIndex(char)]

    def setNode(self, char, node):
        self.children[charToIndex(char)] = node

    def add(self, string, index):
        self.noOfChildren += 1
        if index == len(string):
            return
        curChar = string[index]
        child = self.getNode(curChar)

        if child is None:
            self.setNode(curChar, Node())
            child = self.getNode(curChar)
        child.add(string, index+1)

    def getNoOfAppearances(self, string, index):
        if (index == len(string)):
            return self.noOfChildren
        curChar = string[index]
        child = self.getNode(curChar)
```

```
            if child is None:
                return 0
            return child.getNoOfAppearances(string, index+1)

        def __str__(self):
            return "A WILD NODE APPEARS"

    class Trie:
        def __init__(self):
            self.root = Node()

        def add(self, string):
            self.root.add(string,0)

        def getNoOfAppearances(self, prefix):
            return self.root.getNoOfAppearances(prefix, 0)

    myTrie = Trie()
    myTrie.add("abcde")
    myTrie.add("apple")
    # should print 2
    print(myTrie.getNoOfAppearances("a"))

    # should print 1
    print(myTrie.getNoOfAppearances("ab"))

    # should print 0
    print(myTrie.getNoOfAppearances("b"))
```