A. Other Data Structures

A1. Stack

from linkedlist import *

class Stack(LinkedList):
    def __init__(self):
        super().__init__()

    def push(self, item):
        self.prepend(item)

    def pop(self):
        temp = self.head.value
        self.head = self.head.next  # get rid of top node
        return temp  # return the value of the former top

    def empty(self):
        return self.head == None  # or length == 0

    def __str__(self):
        return "TOP==>" + LinkedList.__str__(self)

Write a swap() method that uses push() and pop() and does not use any of the LinkedList methods, only the Stack methods.

def peek(self):
    if self.head != None:
        return self.head.value
    else:
        return None

def swap(self):
    if self.empty():
        return None
    else:
        temp = self.pop()
        self.push(temp)
        return temp
A2. Queues

A queue is a FIFO, first-in first-out, like a double ended container. (A stack is a one-ended container.)

How to remember how spell queue:

Kyoo – yoo –EE – yoo - EE

Queues are used in many simulations, and to keep track of lists of tasks to do next. The OS uses them, as well as any GUI system where the events that occur and need to be handled (eventually) as kept in a queue.

```python
from linkedlist import *

class Queue(LinkedList):
    ' classical FIFO data structure implemented using a LL. 
    The head is the first node in the list. comes out
    The tail is the last node in the list.'

    def __init__(self):
        super().__init__()  

    def enqueue(self, value):
        ' adds a new node to the end with value as its value''
        # self.append(value)

    def dequeue(self):
        ' returns the value at the head & removes nodes ''
        temp = self.head
        self.head = self.head.next  
        return temp.value

    def empty(self):
        return self.head == None  
 # or length == 0

    def __str__(self):
        return "FRONT==>" + LinkedList.__str__(self) + "<==END"
```

Fundamental operations are

enqueue() add something to the end of the queue

dequeue() remove the node (or value) at the head of the queue

Let's write those:
Applications of queues
- Mostly simulations
- Operating systems
- Telecommunications system
- Anything where you need to shove stuff in at one end of a "pipe" and get stuff out at the other end of the "pipe"
- Breadth-first traversal of a tree (coming up!)

An interesting one is the "mutual suicide pact" or even hot potato routing.

A3. Deques

Deques are double ended queues, where you can delete/insert at either end. You just can’t peek inside the deque. Merely add the push(), pop(), enqueue() and dequeue() methods together. Can also "alias" them.

A deque is a hybrid structure, i.e. a combination of several other structures.

```python
def addFront(self, value):
    self.enqueue(value)

def addEnd(self, value):
    self.enqueue(value)

def delFront(self):
    temp = self.pop()
    return temp

def delEnd(self):
    return self.pop
```

```
class Deque(Stack, Queue):
    def push(self, item):
        Stack.push(item)
    def pop(self):
        return Stack.pop()
```
B. Applications of Stacks and Queues and Dequeues

We’ve already seen many stack applications.

Queues are used in simulations, fairness contests, scheduling. Also used in tree searches which we’ll see later after we touch on trees.

We could think of merging sorted lists using queues:

L1 = [“ape”, “bear”, “dog”, “lion”, “zebra”]
L2 = [“cat”, “elephant”, “frog”, “gazelle”, “manatee”, “yeti”]

Q1 = tail of queue -> [“zebra”, “lion”, “dog”, “bear”, “ape”] <-head of queue
Q2 = tail of queue -> [“yeti”, “manatee”, “gazelle”, “frog”, “elephant”, “cat”] <-head of queue

Imagine we started at the beginning of each list and inserted into the queue, so the first thing put in will be the first thing coming out.

Then

```
while not Q1.empty() and not Q2.empty():
    if Q1.peek() < Q2.peek():
        Q3.enqueue(Q1.dequeue())
    else:
        Q3.enqueue(Q2.dequeue())

if not Q1.empty():
    while not Q1.empty():
        Q3.enqueue(Q1.dequeue())
else:
    while not Q2.empty():
        Q3.enqueue(Q2.dequeue())
```

Q3 is now a Queue in the right order and read to merge with another!

Another application using queues: Josephus’ method of avoiding suicide

Applications using deques

Palindrome checker: (See picture from Miller and Ranum)
C. Python's list data type

How Python implements its lists: it is complicated.

It could implement using a linked list.

What are the costs and trade-offs?

- extra space
- extra time – a lot of time to traverse the list

The problem is the supposed "direct access" using the [i] indexes. It would take a lot of time to walk through the list.

But a raw Array doesn't work, either, because it can't expand.

So a compromise solution exists, which is a lot more work for the programmers, but it finds a halfway point. It creates fixed-size Arrays in blocks or segments, and links those together. This is typical. Our project 4 is a compromise, or hybrid.

C1. Hierarchy of loosening restrictions

List = least restrictive, look anywhere, delete/insert anywhere
Deque ≠ cannot look anywhere, cannot delete/insert anywhere, only at the (two) ends
Stack / Queue = same level, like a dequeue but can only insert/delete at the designated end

D. Taxonomy of Data Structures

There is a disconnect between the logical data structure and its implementation. You can implement a regular list, Array, stack, queue, deque, etc. using a bare linked list. But logically they are different things.

Ultimately, EVERYTHING can be implemented by using a bare fixed-size Array!
or by using a linked list of nodes.

Graph = the most general DS (data structure), nodes pointing to other nodes

You start customizing or restricting graphs in various ways:
(choose one or more, and combine)
- two-way pointers (same thing as an undirected graph, i.e. no direction, just wires)
- fixed # of pointers per node (versus unlimited # of pointers)
- are there cycles? can a node point to itself? can a node link back to itself via another node?

DAG = graph with no cycles, but only one-way pointers
Tree = DAG where each node has exactly one incoming array (except for the head or root)

Many different kinds of trees:
- binary, ternary or variable (how many children nodes)
- balanced vrs. unbalanced
- sorted
- heap property

Trees are used very widely! All over the place. They model nested structures perfectly.

Nested structures in the real world:
- book, chapters within a book, paragraphs with chapters
- web page elements, e.g. a div with paragraphs inside it, and words inside the paragraphs
- prog. lang. control structures: while loops which contain ifs, which could contain other while loops
- hierarchical organizations: divisions, departments, etc.
- The army: 6 legions
  - each legion had 10 cohorts
  - some cohorts had 5 centuria
  - each centuria had 100 men

The important thing is that the smaller divisions are completely contained in their larger units. They do not overlap at all!

E. Trees

Simple binary tree. We will impose the sorted restriction (or property) so that we form a binary search tree. You will see that the tree can no longer be guaranteed to be balanced. However there is a form of a tree that maintains both the sorted property and the balanced property. They are called AVL trees but they are a lot more work!

A tree is a linked node structure such that
- there is one special node call the root.
- No other node points to the root (no parent for the root)
- every node can have 0, 1 or 2 pointers to other nodes

bushy tree many more than 2 children
• No node can point to itself (no cycles)
• No node can be pointed to more than once (only one parent for each node)

The sorted property adds:
• For the payload of each node, the left child's payload is less than its parent's payload
• And the right child's payload is greater than or equal to its parent's payload

Here is a DAG that is not a tree.

Directed Acyclic Graph

A linked list is a degenerate tree!

In this case, the root is the tail of the list. It could be arranged to be backwards.

The heap property is a weaker form of the sorted property:

• For the payload of each node, the left child's payload is less than its parent's payload
• And the right child's payload is less than or equal to its parent's payload

In other words, the two subtrees of a node are not sorted themselves, but the parent is definitely greater than both the children.
Let's write code:

class Node:
    def __init__(self, value):
        self.value = value
        self.lchild = None
        self.rchild = None

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

Tree code tends to be recursive because it is very easy to talk about it.

- A tree can be one node
- Every node of a tree points to either
  - Nothing
  - another tree (whose nodes are distinct from all other parts of the tree, i.e. the root node is not included in this subtree) – called the left child
  - another tree – call the right child

Both left child and right child are trees. We sometimes call them subtrees.

Let's write some methods:

def __add2left(self, value):
1. Single inheritance
   at most one superclass

2. Multiple inheritance
   0, 1, 2... superclasses

Java
Python
C++

Animals
   \[ \text{Vertebrate} \]
   \[ \text{Invertebrate} \]

Vertebrate
   \[ \text{Carnivores} \]
   \[ \text{Herbivores} \]
   \[ \text{Feline} \]