A. Deleting a node from a linked list

Cases:
1. Delete the node whose value is **mongoose**. (Head of the list)
2. Delete the node whose value is **orca**. (Middle of the list)
3. Delete the node whose value is **pelican**. (Tail of the list)

*A1.* Write a method called `delete(somevalue)` which unlinks the node that has somevalue as its value. Handle all three cases above. If you want, you can create helper functions with names such as `delete_from_end()`, `delete_from_middle()`, etc.

```python
def delete(somevalue):
    if headptr.value == somevalue:  # case 1
        headptr = headptr.next
        return

    runner = headptr
    prev = None
    while runner.value != somevalue:
        prev = runner
        runner = runner.next
    prev.next = runner.next
```

runner == ?  
None
Here's the original linked list class:

class LinkedList:

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

    def __init__(self):
        self.head = None  # empty linked list

    def append(self, something):
        newnode = self.Node(something)
        if self.head == None:
            self.head = newnode
        else:
            runner = self.head
            while runner.next != None:
                runner = runner.next
            runner.next = newnode

    def prepend(self, something):
        newnode = self.Node(something)
        if self.head == None:
            self.head = newnode
        else:
            newnode.next = self.head
            self.head = newnode

    def print(self):
        runner = self.head
        while runner != None:
            print(runner.value, end=", ")
            runner = runner.next
        print()

    def find(self, target):
        currentNode = self.head
        while currentNode != None:
            if currentNode.value == target:
                return currentNode
                break
            currentNode = currentNode.next
        return None

    def __str__(self):
        res = ""
        runner = self.head
        while runner != None:
            res += str(id(runner)) + ": " + str(runner.value) + ", "
            runner = runner.next
        return res

# main code
mylist = LinkedList()
mylist.append("stuff")
mylist.append("apples")
mylist.append("oranges")
mylist.append("bananas")
print(mylist)
mylist.print()  # 2 different ways
R. Copying

shallow vrs. deep copies

shallow = copy the pointers only
depth = copy the actual things, make new, separate copies of them

One very shallow copy would be

    newlist = headptr

Now if you change

    newlist.next.next.value = "snake"

Notice that the corresponding thing has change in headptr, because they point to the same actual memory location!

To do a deep copy, you would need to make a whole new node for each node in headptr's list.

*B1. Write a method called deep_copy() that takes in a pointer to the first node in a list and returns a pointer to a new node, but copy all the other nodes, too.

    def deep_copy(orig_pointer):
        """ returns a pointer to a new linked list where all the nodes have new ids, but their values are the same as the old """
def deep copy (orig-pointer):
    newlist = self.Node (orig-pointer, value)
    runner2 = newlist
    runner = orig-pointer.next
    while runner != None:
        tempnode = self.Node (runner.value)
        runner2.next = tempnode
        runner = runner.next
        runner2 = runner2.next
    return newlist
non-volatile memory
- disk, SSD, flash drive
- tape, punched cards

volatile mem (1000000x faster)
RAM - main memory of computer

cost
E. How files are linked lists

Disk drives are organized by platters, sides, tracks and sectors. The smallest unit that is read/written in one operation is a sector, usually 512 bytes. There are often other aggregates like clusters.

Sectors are usually the smallest hardware unit. Each has a unique address, often starting at 0, but really organized by platter, track within platter and sector within track. In order to accommodate small # of disk addresses (e.g. FAT 32, which only permits 4 billion addresses), they gang up powers of 2 number of sectors, (e.g. 4 or 16 or 64) and call that a cluster. Then each cluster gets an address starting at 0, then 1, etc. That's a way to use small address size with huge disks.

Contiguous allocation is a bad idea because you can't increase a file's size without entirely recopying it. Also, dead space naturally occurs between files and wastes your disk drive.

So linked lists are used! Sometimes called block chains. Maximum flexibility in using the storage on a disk.

Downside is that a file may consist of these sectors scattered all over. Defragmentation and disk optimization is needed to reorganize so that most files are mostly contiguous because moving around to find the right sector is a mechanical process, hence very slow. (Two movements: the rotations of the entire disk and the stepper motor to move the head to the right track.)

Also, the next pointer must be encoded somewhere, usually inside a sector so it eats up some data room.

Finally, if a sector goes bad then the block chain is interrupted and the file's later parts are unreachable. Disk fixing tools will often move these orphan chains to temporary files so they can be recovered but often it doesn't work right, esp. with encoded or encrypted files.
You always trade off one benefit for another!

Trade-off: File's sectors can be scattered, takes time.

defrag the disk
C. Stack

Stack and Queue

```python
from linkedlist import *

class Stack(LinkedList):
    """ First node of LL is the top of this stack. """
    def __init__(self):
        super(Stack, self).__init__()
        self.head = None

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

    def pop(self):
        pass  # need to write this

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

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

*C1. Write the pop() method. It both changes the stack and returns the old top element.*

```python
def pop(self):
    temp = self.head.value
    self.head = self.head.next
    return temp
```

*C2. Write a method called swap() that switches around the order of the top two elements. If there is only 1 or 0 elements, swap() does nothing. It does not return anything.*