A. Containers

<table>
<thead>
<tr>
<th>List</th>
<th>ordered, heterogeneous, not unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>unordered, hetero-, unique</td>
</tr>
<tr>
<td>Bag</td>
<td>unordered, hetero-, not unique</td>
</tr>
<tr>
<td>Limited Bag</td>
<td>like a Bag but there are limits on</td>
</tr>
<tr>
<td></td>
<td>the number of instances (applies to</td>
</tr>
<tr>
<td></td>
<td>all or a different limit for each</td>
</tr>
<tr>
<td></td>
<td>value?)</td>
</tr>
<tr>
<td>Ordered Set</td>
<td>ordered, hetero-, unique</td>
</tr>
<tr>
<td>Poset</td>
<td>partially-ordered set</td>
</tr>
</tbody>
</table>

These are all characterized by their external appearance:
1. How many copies of each different thing can there be?
2. Is there an ordering or not?

What is a pointer (a reference)? It is a memory address.

The Node class: used by many dynamic data structures in programming languages. Needed because arrays were of fixed size.

```python
class Node:
    def __init__(self):
        self.value = 0
        self.next = None
```

Note that "next" is a pointer. The empty pointer is None in Python (null in Java.)

headptr is a Python variable in your program. The other boxes are Nodes. They are all "anonymous," i.e., do not have a variable name in your python program.

---

the heap — no discipline
What it really looks like in memory:

The pointers are really just memory addresses (32-bit addresses are shown above, in hex, which is 8 hights.) None is really just 0. You can never use address 0 for your own code.

More than one thing can point to the same node at a given point in your program's time:

Why all of this? Because memory is really just a one-dimensional array of "words" which are 8-bits, 16-bits or some other fixed-size bit cells. It is really showing binary contents:
Let's write some code to run through a linked list, looking for a value:

```python
def find(headNode, target):
    currentNode = headNode
    while currentNode != None and currentNode.value != target:
        currentNode = currentNode.next
    if currentNode == None:
        print("Not found")
    else:
        print("Found!")
```

It is quite common to have a runner loop like this that goes through the list. Here's another version:

```python
def find(headNode, target):
    currentNode = headNode
    while currentNode != None:
        if currentNode.value == target:
            print("Found")
            break
        currentNode = currentNode.next
    else:
        print("Not found")
```

There are many challenges:

1. How do you copy a linked list?

2. How do you add a new node to a linked list?
   a.) where, at the end?
   b.) where, at the front?
   c.) where, in the middle somewhere?
   d.) where in order to keep the list sorted?

3. How do you delete a node from a linked list?
   a.) where, at the end?
   b.) where, at the front?
   c.) where, in the middle somewhere?

In strongly typed languages, like Java or C, the nodes come from a pool of free memory areas called the **heap**. When a node is pointed to by a variable in your code, then it is considered "live" or "active" or "in use." If a node is not pointed to by a variable in your code, nor is it pointed to by a node that is ultimately pointed to by a variable in your code, then it is "garbage" or "dead." The memory areas can be reclaimed and reused then.
For example, if you wrote:

```plaintext
headptr = None
```

and assuming that no other variable points to "John"'s node, then "John" is garbage. But "Susan" is then not pointed to by anything that is not garbage. So she becomes garbage and will have to be collected, and so forth. This whole process is called garbage collection and takes a lot of time. Some languages like C or Java delay GC or make it manual (i.e. you the programmer have to do it!) That avoids the situation where your program suddenly freezes because it is rushing through all memory regions to find and reclaim garbage nodes.
Next step: write a class to represent a linked list:

```python
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):
        pass

    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
        else:
            return None

# main code
mylist = LinkedList()
mylist.prepend("stuff")
mylist.prepend("apples")
mylist.prepend("oranges")
mylist.prepend("bananas")
mylist.print()
```

This was the day 9 last time.
Python, Ruby, LISP, Prolog

Java → C#

C++, C

↑ Machine Lang (assembler)
```python
def print_list(headptr):
    runner = headptr
    while runner != None:
        print(headptr.value)
        runner = runner.next
    print("--- at the end ---")

headptr = Node()
headptr.value = "John"
temp = Node()
temp.value = "Susan"
headptr.next = temp
temp = Node()
temp.value = "Marcia"
headptr.next.next = temp
```
```python
def length(headNode):
    counter = 0
    runner = headNode
    while runner != None:
        counter += 1
        runner = runner.next
    return counter
```
headNode 1000
target Mark
currentNode 1000
output not found

headptr 3000

List

Tree

Graph: most general

Variable # of pts in each node

<---- bidirectional