AP Computer Science with Data Structures

Linked Lists in Java Town Homework

As is the case for **all** homework assignments in this course, you are expected to do this independently. That means that this **is not** a group exercise.

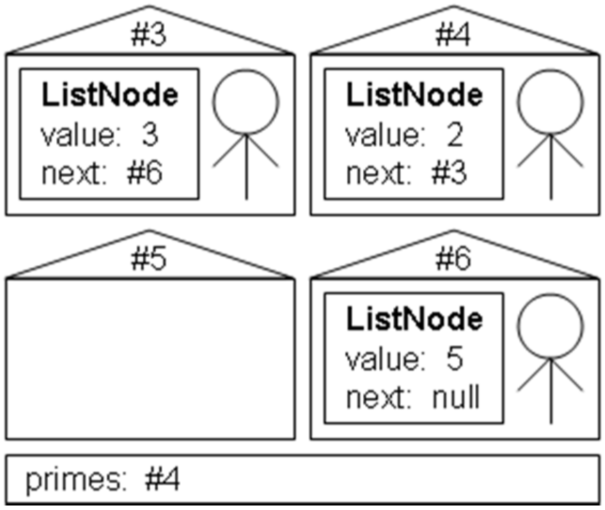
In order to create linked lists, you will need to define the following `ListNode` class.

```
public class ListNode
{
  private value;
  private next;

  public ListNode(initValue, initNext)
  {    value = initValue; next = initNext; }

  public getValue() { return value; }
  public getNext() { return next; }

  public setValue(theNewValue) { value = theNewValue; }
  public setNext(theNewNext) { next = theNewNext; }
}
```

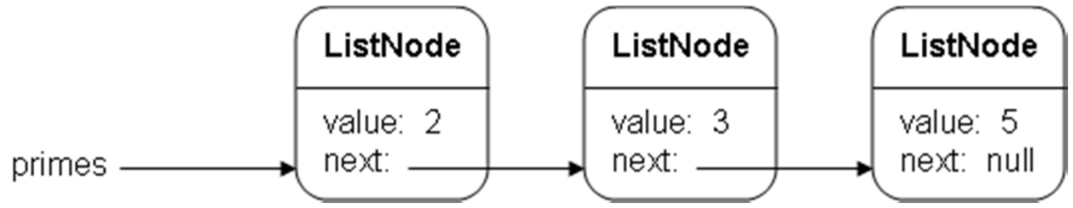
Now, let's use this class to create a linked list of three elements as follows.

```
primes = new ListNode(2, new ListNode(3,
                                new ListNode(5, null)));
```

Something like the following will appear in JavaTown.



It turns out, though, that JavaTown is overwhelming us with information. In particular, we don't actually care which street address houses which `ListNode`. And we certainly don't want to have to follow all the address numbers to find the list (although this is precisely what JavaTown and Java actually do). Instead of thinking of the `next` field as containing #3, for example, it will be far more useful to imagine that the `next` field *points to* an object corresponding to the one in house #3. We'll represent such relationships between instances by drawing an instance diagram like the following.



Instance diagrams will be invaluable to us in AP Computer Science. Each box represents one `ListNode` object—a *node* in linked list terminology. The left part of each box points to the node's value, and the right part points to the next node. The keyword `null` is used here as the end-of-list marker. We can think of it as representing an empty list—one with no elements.

We'll write all of our methods for working with linked lists in a class called `ListUtil.`

Let's add an ***example*** method to the class `ListUtil` which traverses a list of numbers and adds up all of the values. (Traversing a list means that we will look at every value in the list). The key to this method is realizing that it is recursive.

```
public class ListUtil
{
  public addValues(list)
  {
      if(list == null) return 0;
    return list.getValue() +
                this. addValues(list.getNext());
      }
  }
```

The method `addValues` takes in a list as a parameter. It then gets the value and makes a recursive call to itself, passing a list that is one node less. Try this in Java Town, and you will see the thought balloons representing the stack frames appear one on top of another until the entire list is processed.

The beautiful thing about a linked list is that it is a *recursive* data structure. Each node's `next` field points to the beginning of a smaller linked list. Therefore, procedures that traverse linked lists are naturally recursive, like the `addValues` method, which returns a value corresponding to the sum of all the values in a list.

Linked-list traversing procedures tend to fit into the following general form.

```
public f(list [, other arguments])
{
  if (list == null)
    return [base case];
  return [use list.getValue() and this.f(list.next())];
}
```

By the way, linked lists can be a bit like an optical illusion. If you look at them one way, a linked list *consists of* nodes. From another perspective, a node *is* a linked list, and `null` *is* the empty list. So, we can think of the `ListNode` constructor as taking in a value and another `ListNode` (or `null`), and returning a new `ListNode`. But it is often more helpful to think of the `ListNode` constructor as taking in a value and a *list*, and returning a new *list*.

Homework Exercises – Write your answers in your notebook

1. Assume that the variable `list` points to the beginning of a linked list. For each of the following, determine what expression will return the requested value.

    a) the first element of `list`
    b) the second element of `list`
    c) the third element of `list`
    d) a list of all but the first two elements of `list`

2. Draw an instance diagram showing `x`, `y`, `z`, and `w`, and the values they point to after the following code is executed.

```
x = new ListNode(1, null);
x2 = x;
y = new ListNode(x, x2);
w = new ListNode(1, null);
w.setNext(new ListNode(2, w));
```

Use your diagram to predict the values of the following expressions. Write down your predictions and use JavaTown to check your predictions.

a) `x == new ListNode(1, null);`
b) `w.getNext().getValue();`
c) `w.getNext().getNext().getValue();`
d) `w == w.getNext();`
e) `w == w.getNext().getNext();`