ITI 1121. Introduction to Computing II *

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of February 23, 2013

Abstract

- Implementing a stack using linked elements

*These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

Summary

We have seen that arrays are efficient data structures. Accessing any element of an array necessitates a constant number of operations.

For some applications, particularly if the number of elements to be stored is not known in advance or varies during the execution of the program, arrays are not suitable.

A technique frequently used consists of copying the elements of the array to a new one that's larger and substituting the new array for the old one.

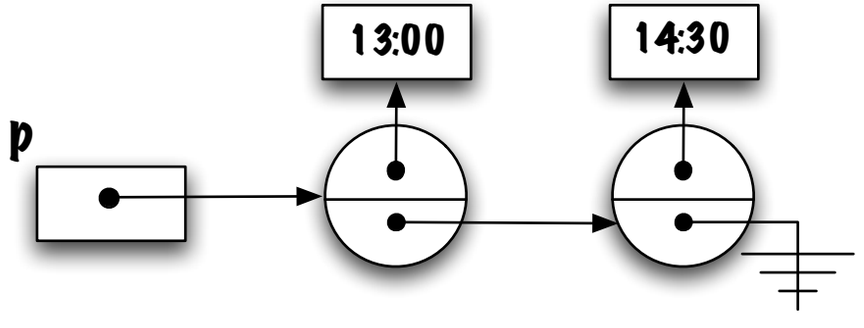
The consequences are that 1) insertions into the array are now slower (because of the need to copy the elements to the new array) and 2) the amount of physical memory used is larger than the actual logical size required by the application; i.e. memory is reserved but not necessarily used.

Linked structures

We now consider a data structure that always uses the exact amount of memory required by the application.

This data structure grows one element at a time **without** copying any elements!

We will need to create "containers" such that each "container" will hold one element.



The price to pay will be that certain elements will be more accessible than others, as the above figure suggests.

Linked structures

Access to the first element will be fast, but the access to the subsequent elements will necessitate "traversing" the data structure up to the element of interest.

Just like arrays, each element has a single predecessor and successor (except for the first and last elements). We say that the structure is **linear**.

Unlike the arrays, this data structure will not be implemented with contiguous memory cells.

How to implement such structure in Java?

Study the following class definition (as usual, we start with the visibility of the instance variables being public, we will fix that later):

```
public class Elem {
    public Object value;
    public Elem next;
}
```

Do you notice anything peculiar?

Yes, the type of the reference **next** is the name of the class that we are currently defining.

Hum, is this legal?

Let's try compiling this definition:

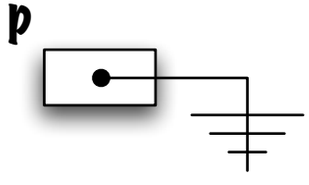
```
> javac Elem.java
```

$\Rightarrow$ it works!

What does it do?

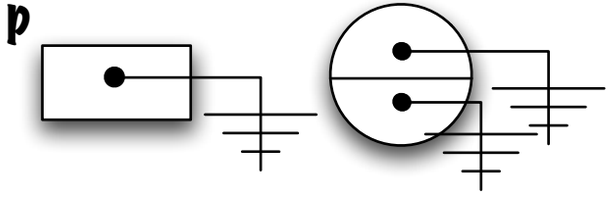
Let's see what we can do with it, let's declare a reference of type **Elem**:

```
Elem p;
```

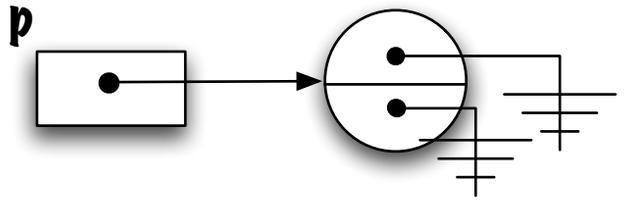


Creating an instance of the class **Elem**.

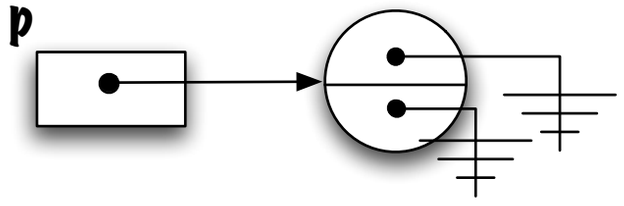
```
new Elem();
```



Let's assign the reference of the newly created **Elem** to **p**:

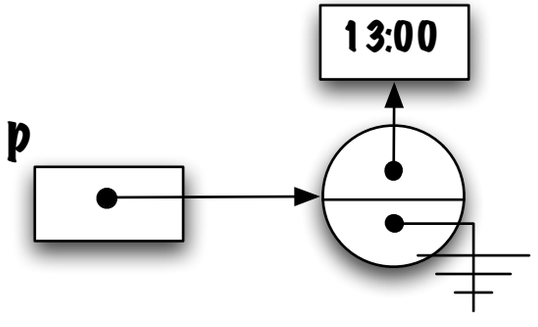


Convention. I will be using circles to represent the memory diagrams of all the objects of the class **Elem**. The top part represents the instance variable **value**. The bottom par represents the instance variable **next**.



How would you change the content of the instance variable **value**?

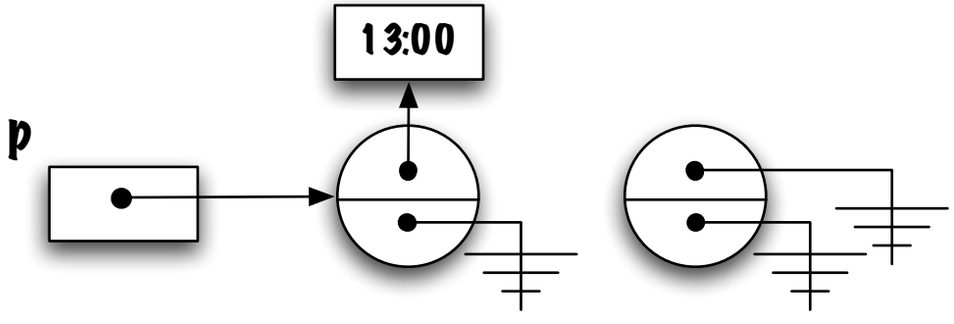
```
p.value = new Time( 13, 0, 0 );
```



We use the "dot" notation to access the attribute; here, this works because the instance variable is public (more about this later).

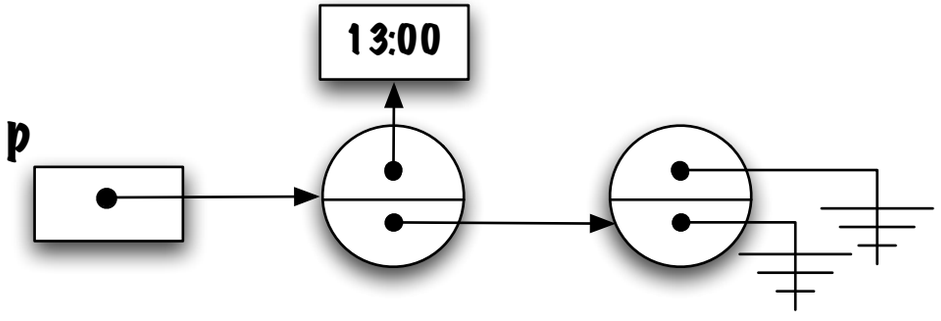
Let's create another **Elem** object:

```
new Elem();
```

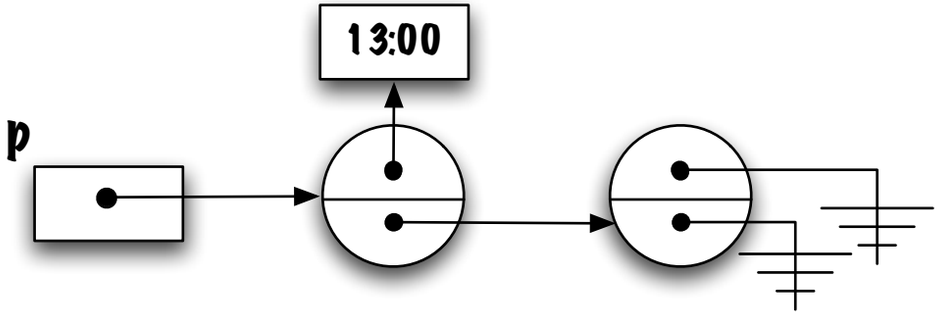


⇒ How are we going to connect the two together?

```
p.next = new Elem();
```

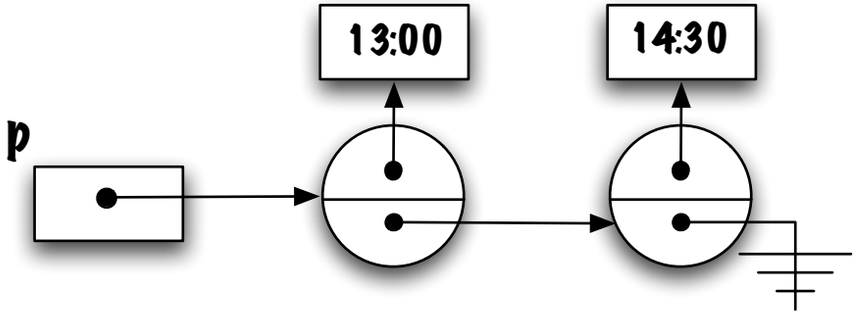


$\Rightarrow$ The reference to the newly created **Elem** object is assigned to **p.next** (a variable of type **Elem**).



How are we going to assign a value to that node (the elements of a linked structure are often called **nodes**)?

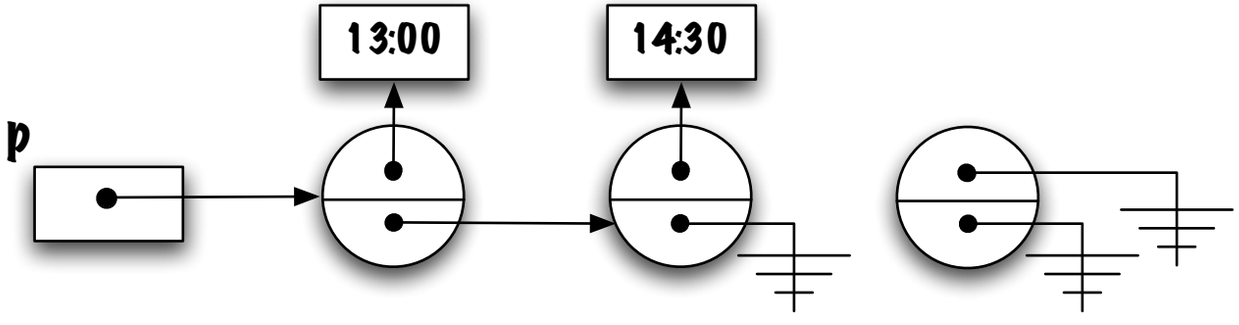
```
p.next.value = new Time( 14, 30, 0 );
```



$\Rightarrow$ To change the **value** of that node, we follow the reference **p.next**.

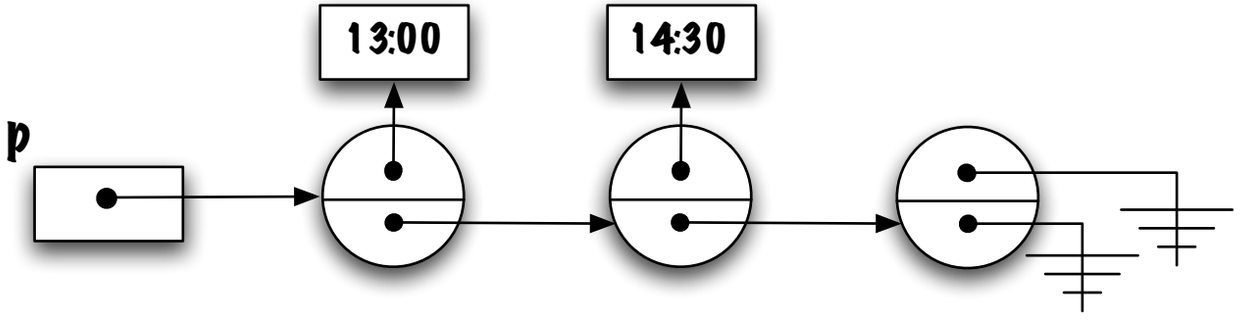
Let's create another node:

```
new Elem()
```

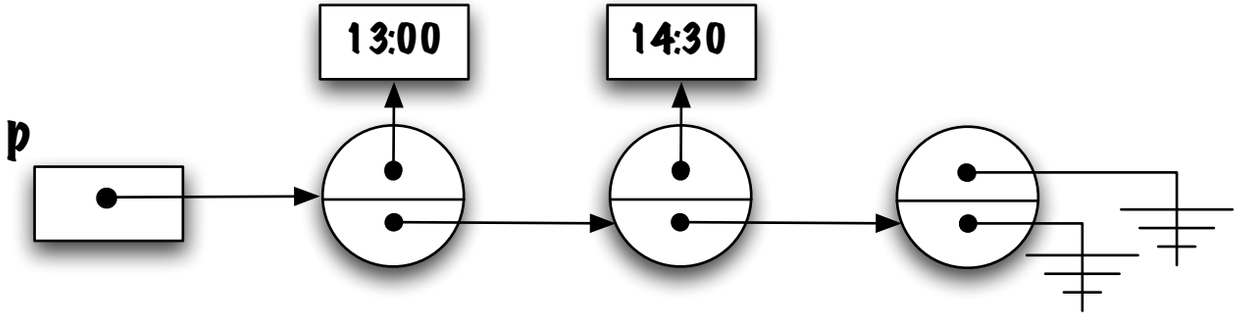


⇒ Again, how is it going to be "linked" to the other nodes.

```
p.next.next = new Elem();
```

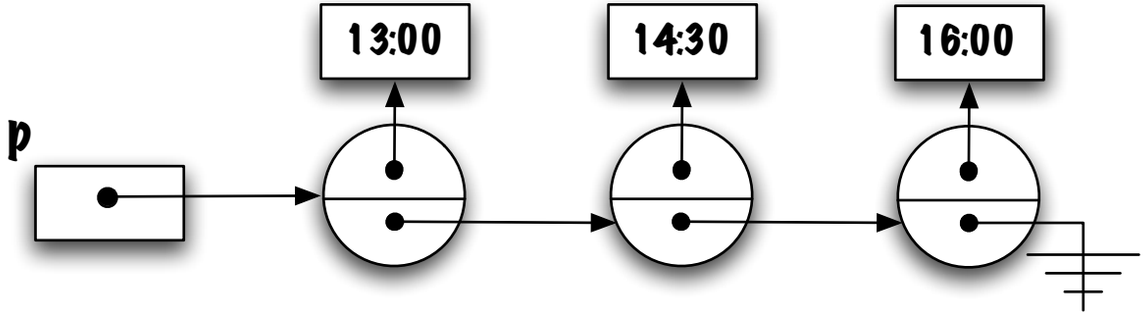


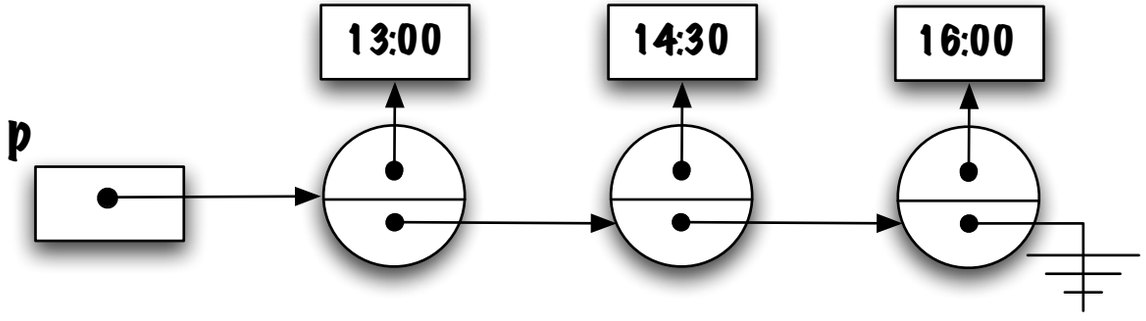
The reference of the newly created node has been assigned to the instance variable **p.next.next**.



Let's assign a value to the newly inserted element. How?

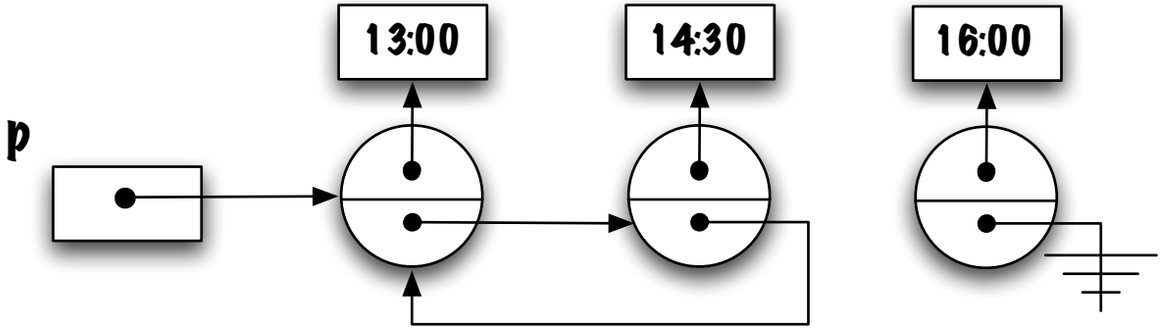
```
p.next.next.value = new Time( 16, 0, 0 );
```





What does the following statement do?

```
p.next.next = p;
```




```
p.next.next = p;
```

Hum . . .

- A circular data structure has been created!
- The element whose value is 16:00 is not accessible anymore;
- The node will be recycled by the garbage collector; gc().

$\Rightarrow$ All this is the basis of linked data structures: values are linked one to another with help of reference (pointer) variables.

Linked data structures

```
public class Elem {
    public Object value;
    public Elem next;
}
```

Linked structures allow to:

- Represent linear data structures such as lists, stacks and queues;
- They always use the "right" amount of memory;
- All this is made possible because of this self reference; the declaration of next of type **Elem** within the class **Elem** itself.

$\Rightarrow$ A data structure such as the one we have just described is called a linked list, to be more precise this is **singly linked list** because each node has a single link (the instance variable next).

Summary

Linked data structures are an alternative to arrays to store a “collection of values”.

Linked structures always use the required amount of memory; because each element is stored in its own “container”, which we called **Elem**. Each container is linked to its successor with help of a reference.

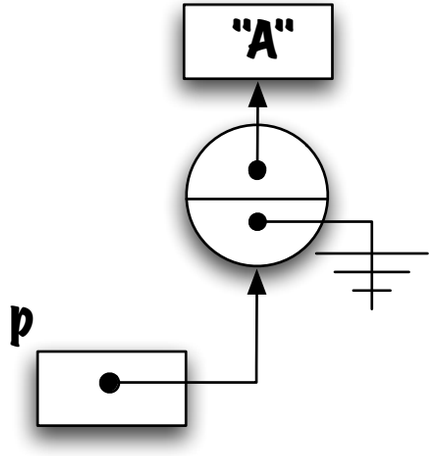
For now, we limit ourselves to linear data structures but linked structures are also used to represent graphs and trees.

Typical constructor for the class **Elem**:

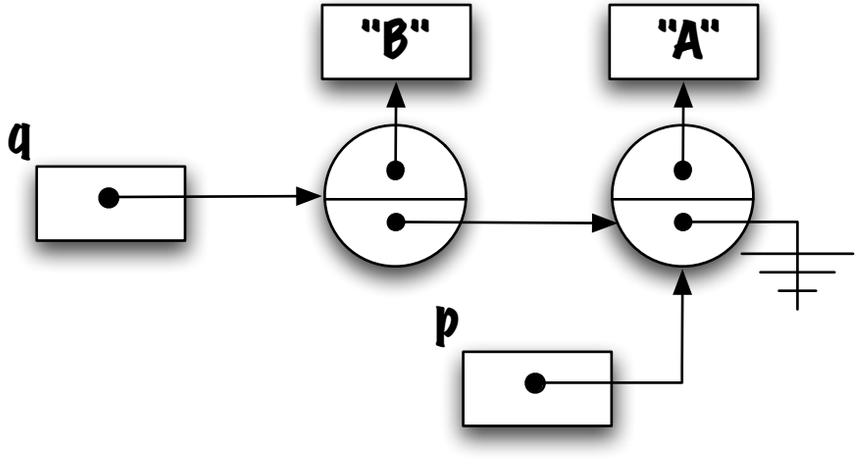
```
class Elem {

  Object value;
  Elem next;

  Elem( Object value, Elem next ) {
      this.value = value;
      this.next = next;
  }
}
```

and its typical usage:

```
p = new Elem( "A", null );
```




```
q = new Elem( "B", p );
```



Pitfall

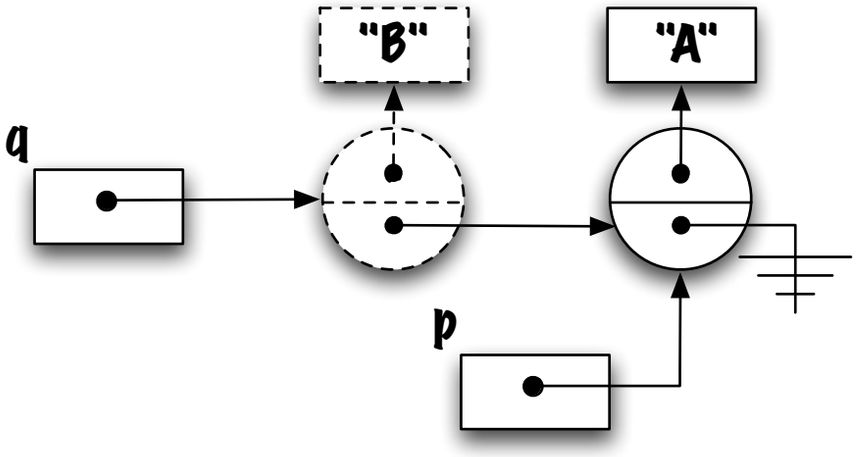
The following example illustrates a kind of error that is observed frequently when working with linked data structures:

```
Elem p = null, q = null;
p = new Elem( 1, null );
q.next = p;
```

The statements is syntactically correct but will produce the following error at runtime:

```
Exception in thread "main" java.lang.NullPointerException
        at T01.main(T01.java:8)
```

$\Rightarrow$ why?



The above diagram illustrates the problem. The intent, maybe, was to create a linked list so that **q.next** designate the element currently designated by **p**, however, the object shown as a dashed line has never been created.

The appropriate statements should have been:

```
p = new Elem( "A", null );
q = new Elem( "B", null );
q.next = p;
```

or

```
p = new Elem( "A", null );
q = new Elem( "B", p );
```

Workaround

As a general rule, whenever an attribute of an object is used, it's a good idea to first make sure (implicitly or explicitly) that the object exists:

```
if ( q != null )
   q.next = ...
```

$\Rightarrow$ you'll see that testing for **null** will become an idiom!

Stack: linked implementation

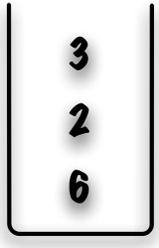
We will be using linked elements to implement the interface **Stack**.

```
public class LinkedStack implements Stack {

    public boolean empty() {

    }
    public void push( Object o ) {

    }
    public Object peek() {

    }
    public Object pop() {

    }
}
```

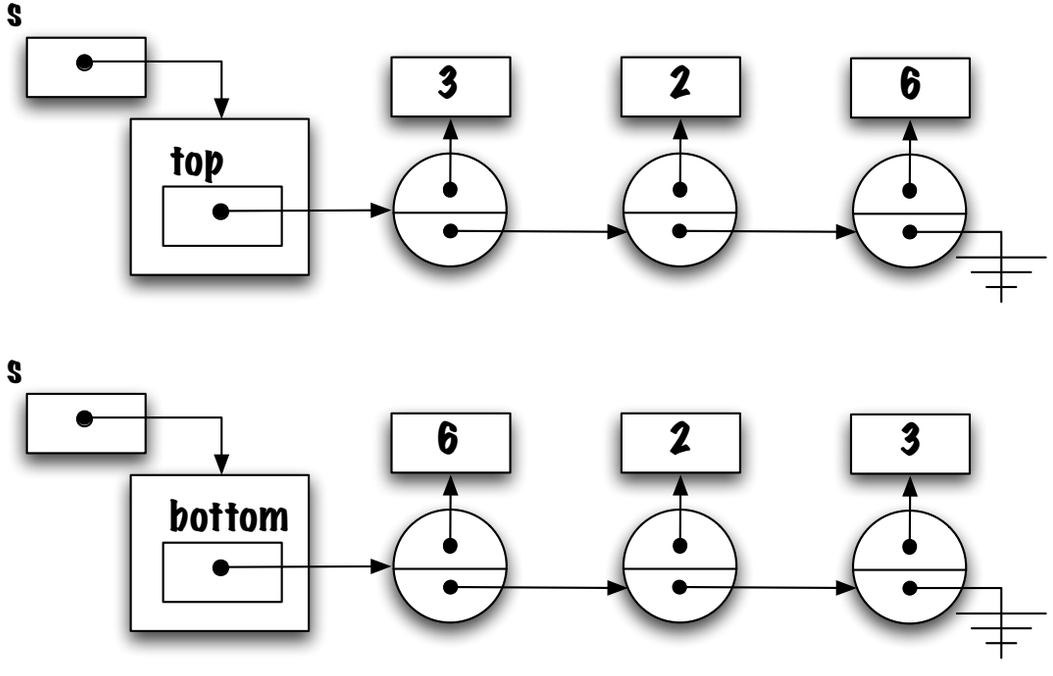
LinkedStack

What are the instance variables?

```
public class LinkedStack implements Stack {

    private Elem bottom; // or is it top?

    public boolean empty() { ... }
    public void push( Object o ) { ... }
    public void push( Object o ) { ... }
    public Object pop() { ... }
}
```



Compare the following two implementation:



The implementation where most accessible element (the first one) is top should be preferred since all the operations (e.g. push and pop) are accessing the same extremity.

In the other implementation, the entire chain should be scanned in order to add or remove an element.

The more elements there are in the stack, the more costly the push and pop operations will become.

Class Elem (0/3)

In the current implementation, the instance variables of the class are public which violates the **principle of encapsulation**, this is not a good design strategy. What are the options?

Class Elem (1/3)

```
public class Elem {
  private Object value;
  private Elem next;
  public Elem( Object value, Elem next ) {
      this.value = value;
      this.next = next;
  }
  public void setValue( Object value ) {
      this.value = value;
  }
  public void setNext( Elem next ) {
      this.next = next;
  }
  public Object getValue() {
      return value;
  }
  public Elem getNext() {
      return next;
  }
}
```

Class Elem (2/3)

```
class Elem {

  protected Object value;
  protected Elem next;

  protected Elem( Object value, Elem next ) {
      this.value = value;
      this.next = next;
  }
}
```

Elem is a first level class whose visibility is "package". If **LinkedList** and **Elem** are members of the same "package" then **LinkedStack** has access to the instance variables of the class **Elem**.

Class Elem (3/3)

```
public class LinkedStack implements Stack {

    private static class Elem {
      private Object value;
      private Elem next;
      private Elem( Object value, Elem next ) {
          this.value = value;
          this.next = next;
      }
    }

    private Elem top;

    // ...
}
```

Class Elem (3/3)

Elem is a **nested** class of the class **LinkedStack**.

Although the visibility of the class is private, and the visibility of its instance variable is private, **LinkedStack** has access to the instance variables of **Elem**, because **Elem** is a nested class of **LinkedStack**.

For now, all our nested classes will be "static". Such classes can be used as if they were top level classes except that their definition is nested and the outer implementation has access to the implementation of the nested class.

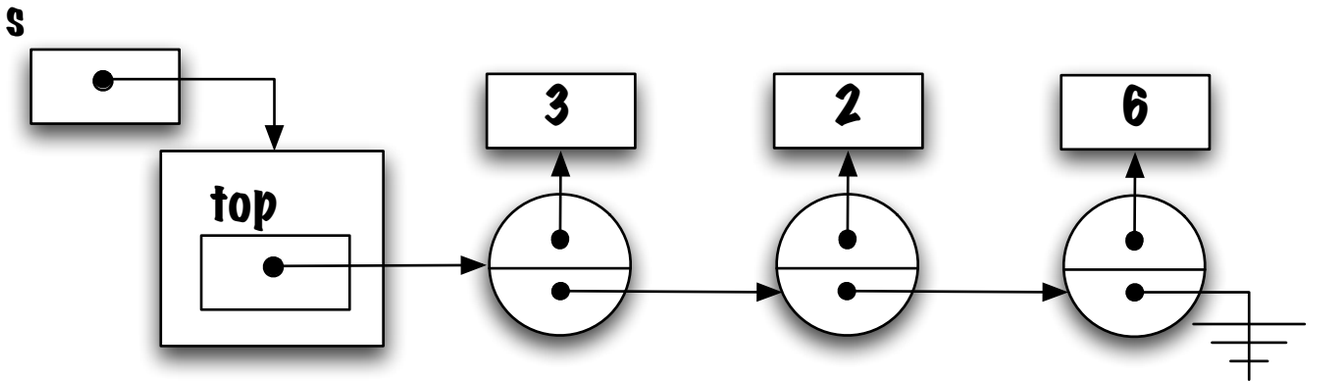
Later, we will consider an other kind of nested class.

Generics

```
public class LinkedStack<T> implements Stack<T> {

    private static class Elem<E> {
        private E info;
        private Elem<E> next;

        private Elem( E info, Elem<E> next) {
            this.info = info;
            this.next = next;
        }
    }
    private Elem<T> top; // Instance variable

    public boolean isEmpty() { ... }
    public void push( T info ) { ... }
    public T peek() { ... }
    public T pop() { ... }
}
```

```
public class LinkedStack implements Stack {

    private static class Elem { ... }

    private Elem top;

    public LinkedStack() {


    }
    public boolean isEmpty() {

    }
    public Object peek() {
        // pre-conditions


    }
    public Object pop() {
        // pre-conditions:
```

```
    }
    public void push(Object o) {
        // pre-conditions:



    }
}
```



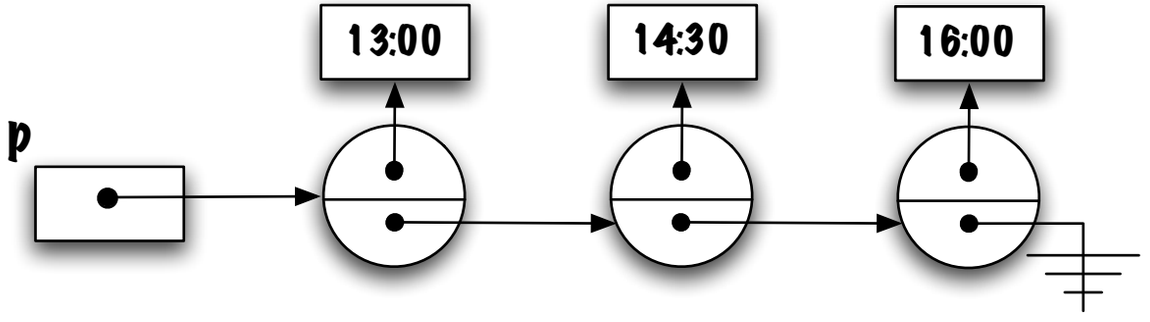
s.peek() should return the value 3..

```
public Objecct peek () {
    return _________________ ;
}
```

Complete the implementation.

top.value;

Detecting the end of a linked structure



Look at the above picture, how can you detect the end of a linked structure?

What distinguishes the last node from the others is the fact that its attribute **next** contains the value *null*.

Traversal

Let's override the method **toString**:

```
public String toString();
```

In order to assemble the **String** one needs to scan the list of elements from one end to the other; in the case of a linked list, this is called list **traversal**.

Similarly, to compare two lists for equality one would need to traverse both lists simultaneously.

In the array-based implementation of a Stack, a "for" loop and an index would have been used to "traverse" the array.

Array-based implementation

```
public String toString() {
    String res = "[";
    if ( size > 0 ) {
        int p = 0;
        res = res +  elems[ p ];
        p = p + 1;
        while ( p < size )
            res = res + ", " + elems[ p ];
            p = p + 1;
    }
    res = res + "]";
    return res;
}
```

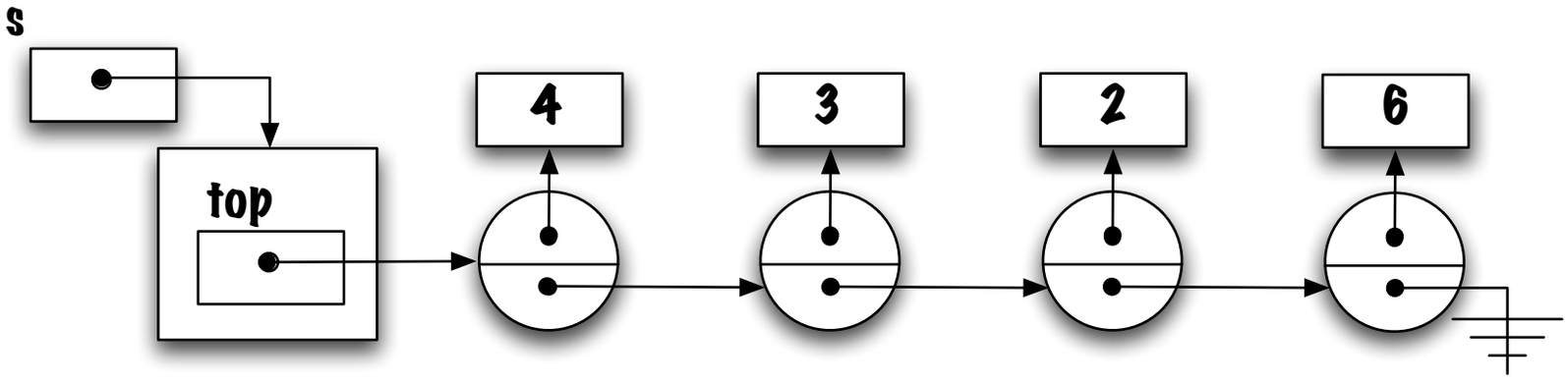
Complete

```
public String toString() {
    String res = "[";
    if ( ____________ ) {
        ____ p = ____;
        res = res + _______;
        p = ______;
        while ( _________ ) {
            res = res + ", " + _______;
            p = ______;
        }
    }
    res = res + "]";
    return res;
}
```

Complete

```
public String toString() {
    String res = "[";
    if ( ____________ ) {
        ____ p = ____;
        res = res + _______;
        p = ______;
        while ( _________ ) {
            res = res + ", " + _______;
            p = ______;
        }
    }
    res = res + "]";
    return res;
}
```

What test can be used to determine if the stack is empty?

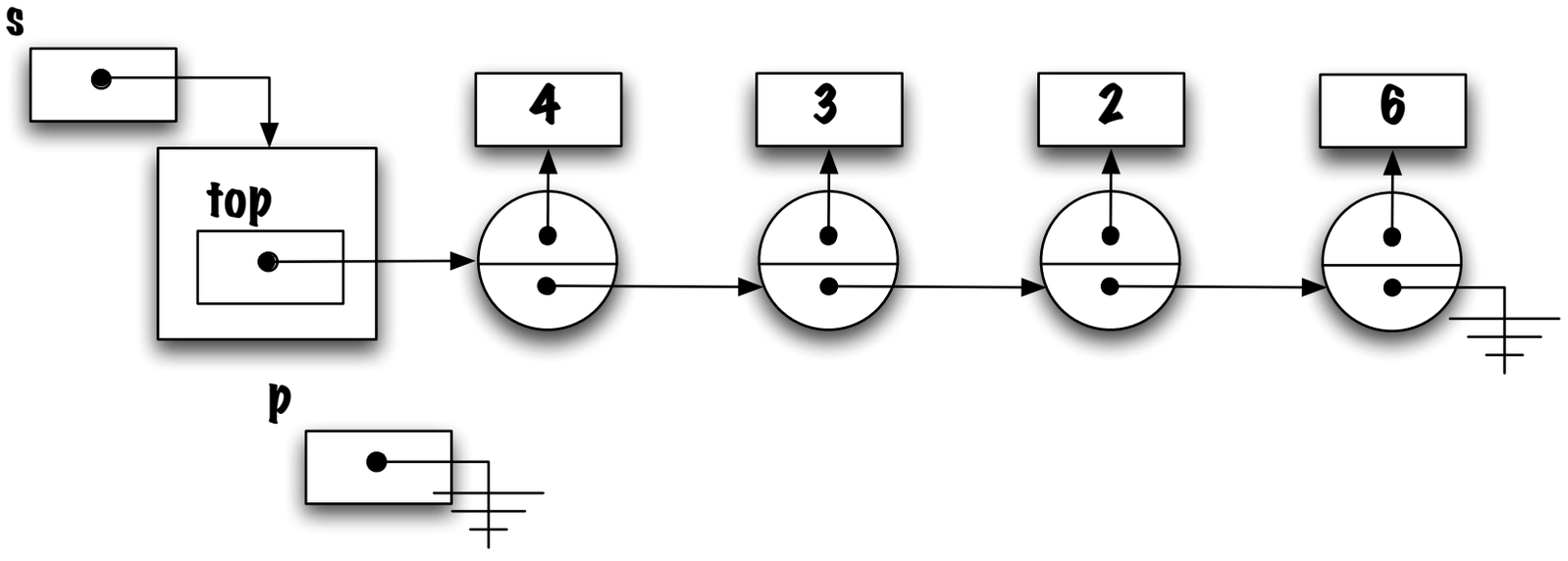


top == null

Complete

```
public String toString() {
    String res = "[";
    if ( top != null ) {
        ____ p = ____;
        res = res + _______;
        p = ______;
        while ( _________ ) {
            res = res + ", " + _______;
            p = ______;
        }
    }
    res = res + "]";
    return res;
}
```

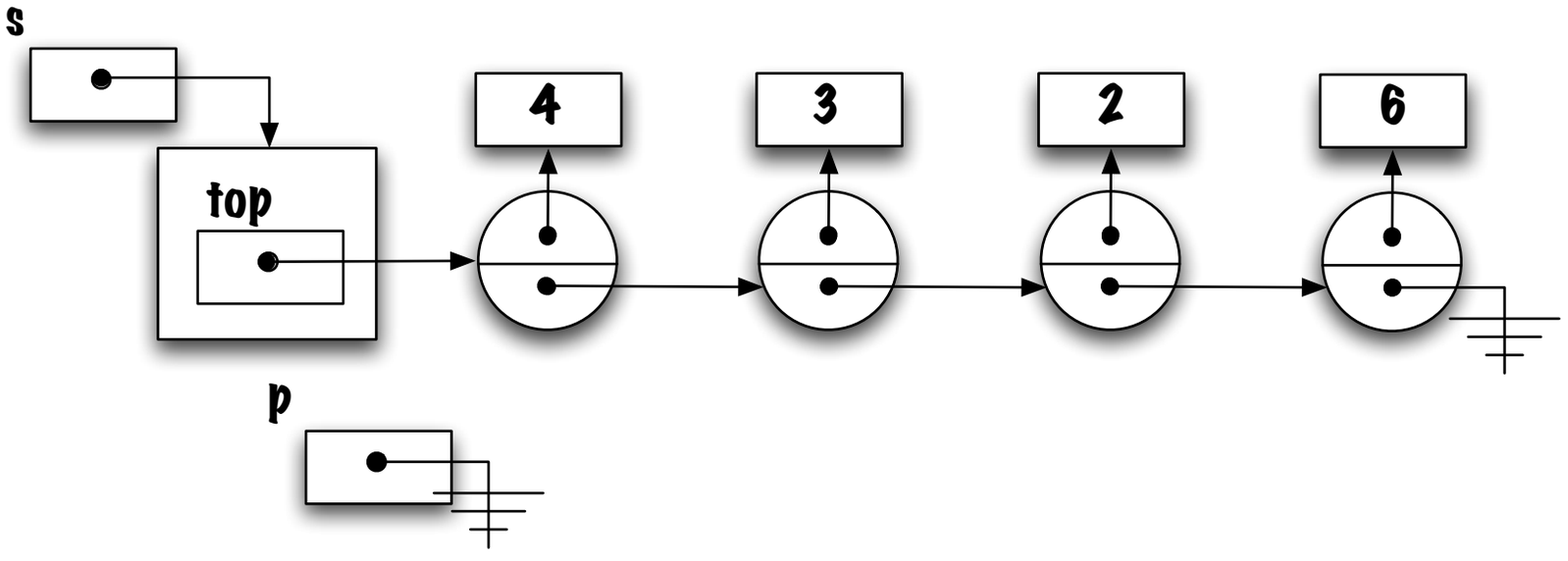
The variable **p** plays a similar role to that of the index in the array-based implementation. The variable is used to access the elements one at time. What is the type of the variable **p**? **Elem**.

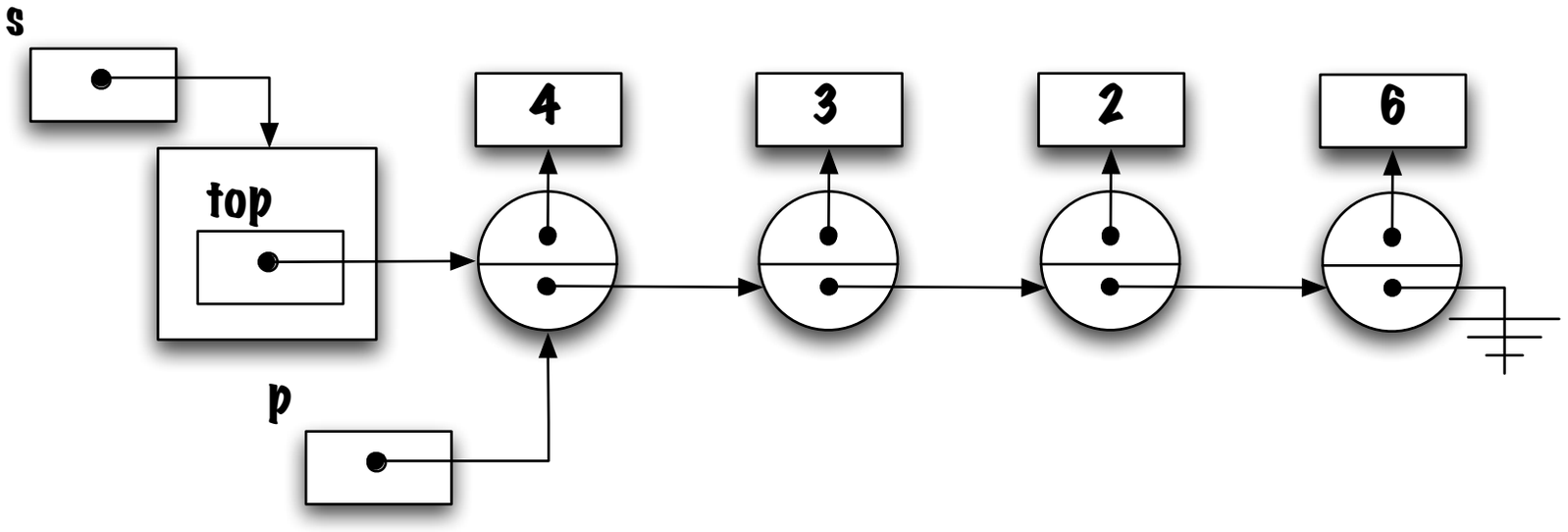

s
top
4
3
2
6
p

Complete

```
public String toString() {
    String res = "[";
    if ( top != null ) {
        Elem p = ____;
        res = res + _______;
        p = ______;
        while ( _________ ) {
            res = res + ", " + _______;
            p = ______;
        }
    }
    res = res + "]";
    return res;
}
```

What is its initial value?


s
top
4
3
2
6
p

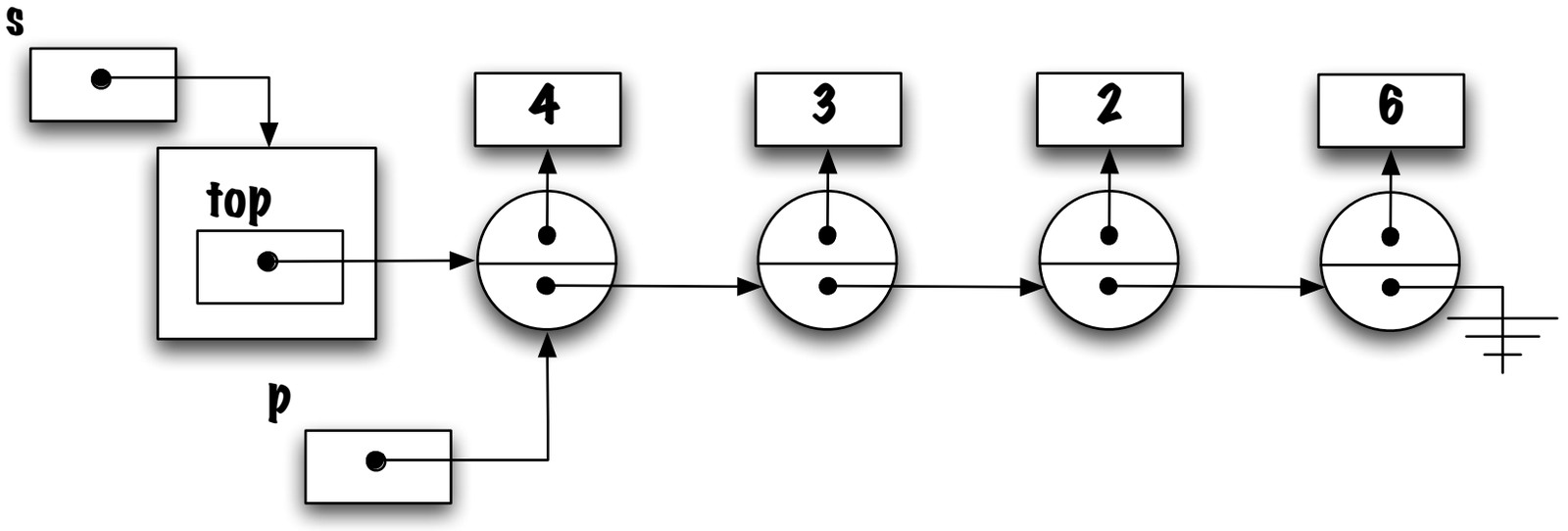


top, therefore **p** and **top** both designate the same element, the top one.

Complete

```
public String toString() {
    String res = "[";
    if ( top != null ) {
        Elem p = top;
        res = res + _______;
        p = ______;
        while ( _________ ) {
            res = res + ", " + _______;
            p = ______;
        }
    }
    res = res + "]";
    return res;
}
```

How to access the value of the top element?

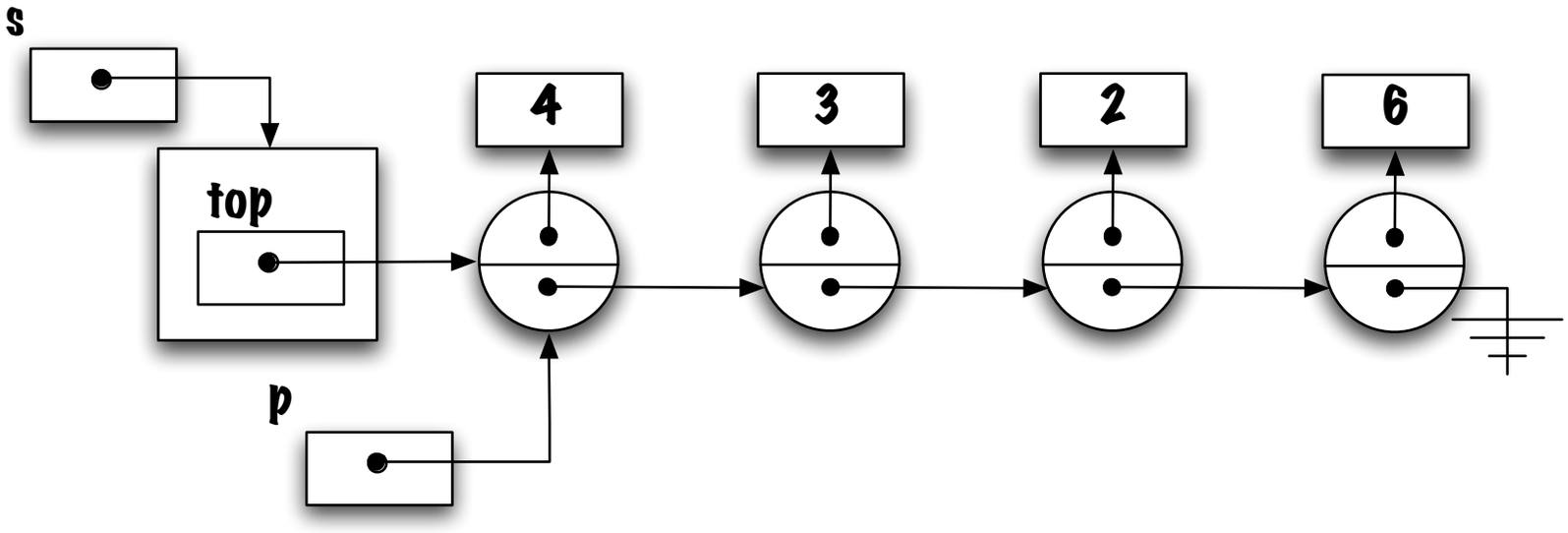


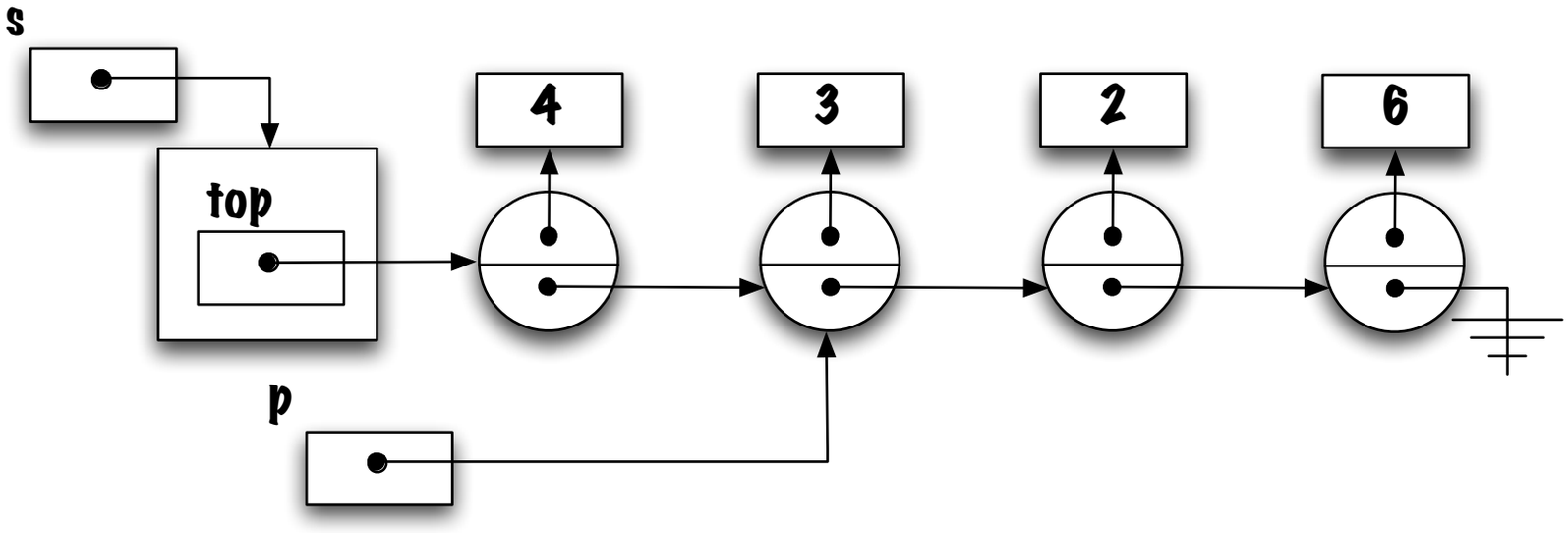
top.value.toString(), **p.value.toString()**, or simply **top.value** or **p.value**.

Complete

```
public String toString() {
    String res = "[";
    if ( top != null ) {
        Elem p = top;
        res = res + p.value;
        p = ______;
        while ( _________ ) {
            res = res + ", " + _______;
            p = ______;
        }
    }
    res = res + "]";
    return res;
}
```

We now need to move the reference **p** one position forward (the equivalent of the **i = i + 1** in the array-based implementation). How to?


s
top
p
4
3
2
6

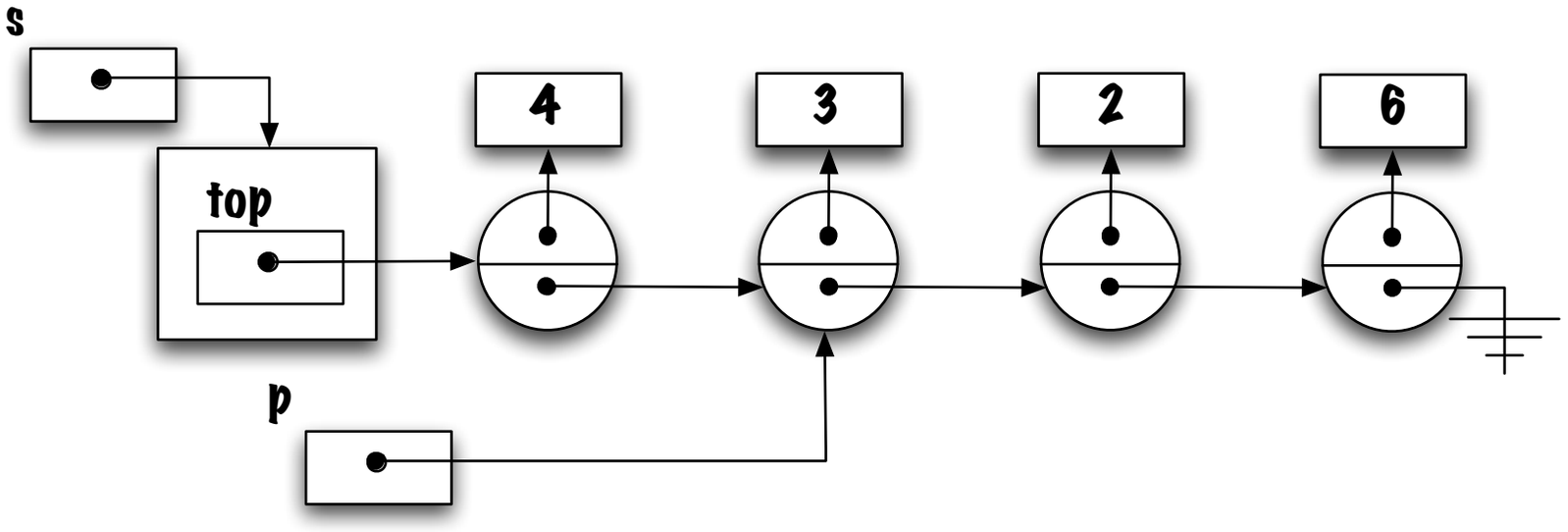


p = p.next.

Complete

```
public String toString() {
    String res = "[";
    if ( top != null ) {
        Elem p = top;
        res = res + p.value;
        p = p.next;
        while ( _________ ) {
            res = res + ", " + _______;
            p = ______;
        }
    }
    res = res + "]";
    return res;
}
```

The body of the loop will be visited as long as the end of the linked structure has not been detected. What is the test?



p != null

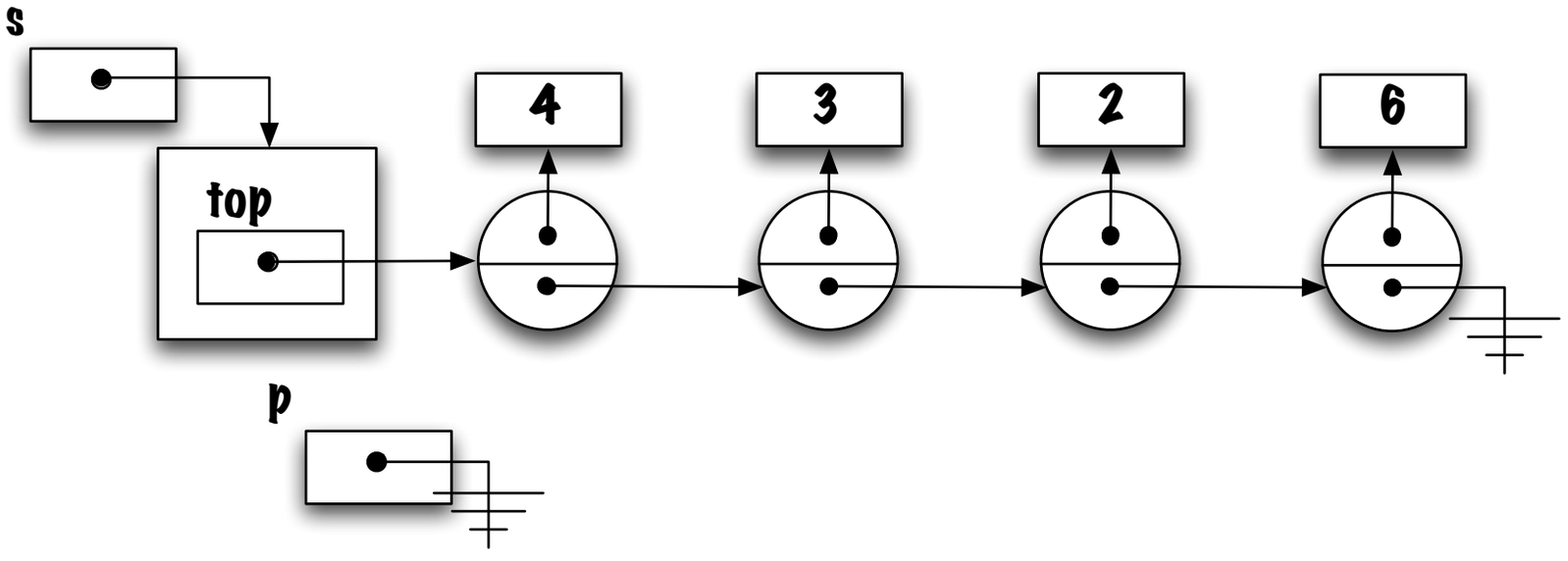
Complete

```
public String toString() {
    String res = "[";
    if ( top != null ) {
        Elem p = top;
        res = res + p.value;
        p = p.next;
        while ( p != null ) {
            res = res + ", " + _______;
            p = ______;
        }
    }
    res = res + "]";
    return res;
}
```

Complete

```
public String toString() {
    String res = "[";
    if ( top != null ) {
        Elem p = top;
        res = res + p.value;
        p = p.next;
        while ( p != null ) {
            res = res + ", " + p.value;
            p = ______;
        }
    }
    res = res + "]";
    return res;
}
```

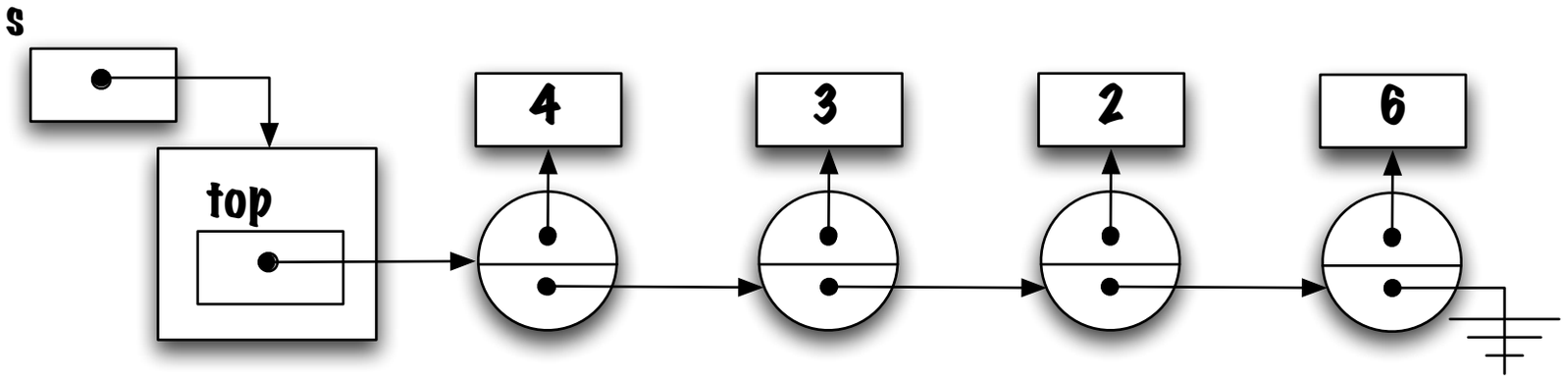
How to move forward? (the equivalent of **i = i + 1** in the array-based implementation)

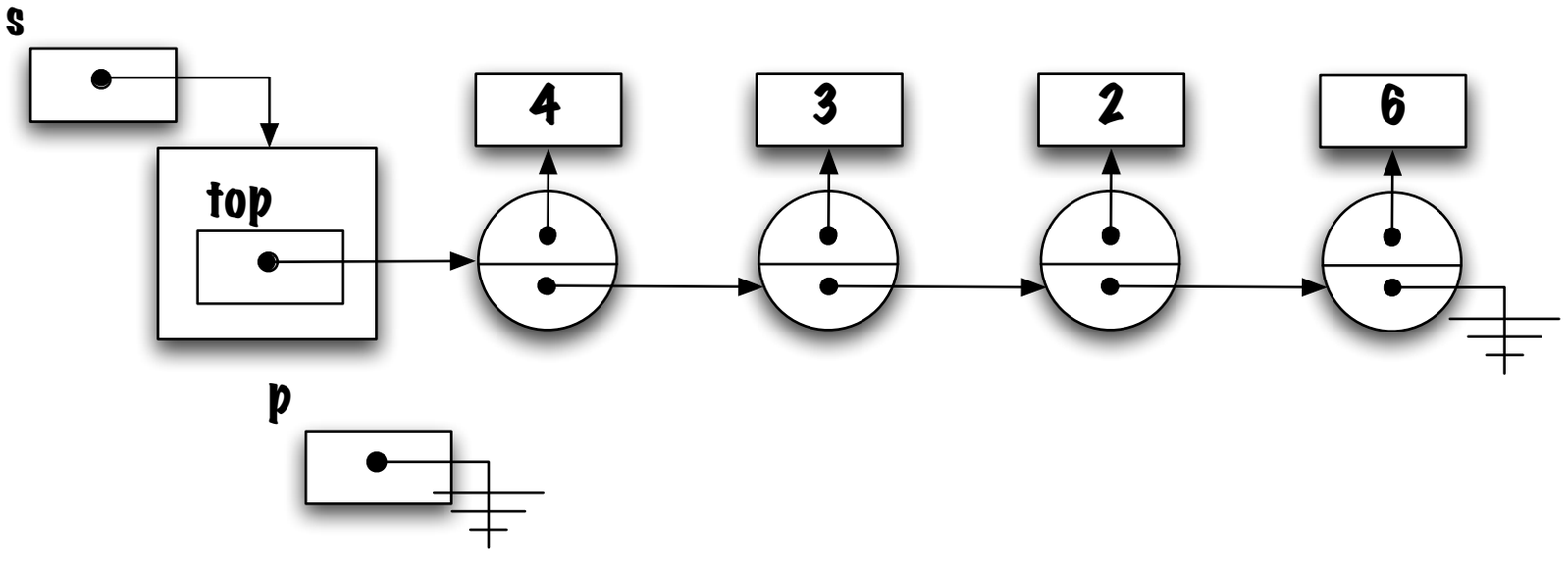


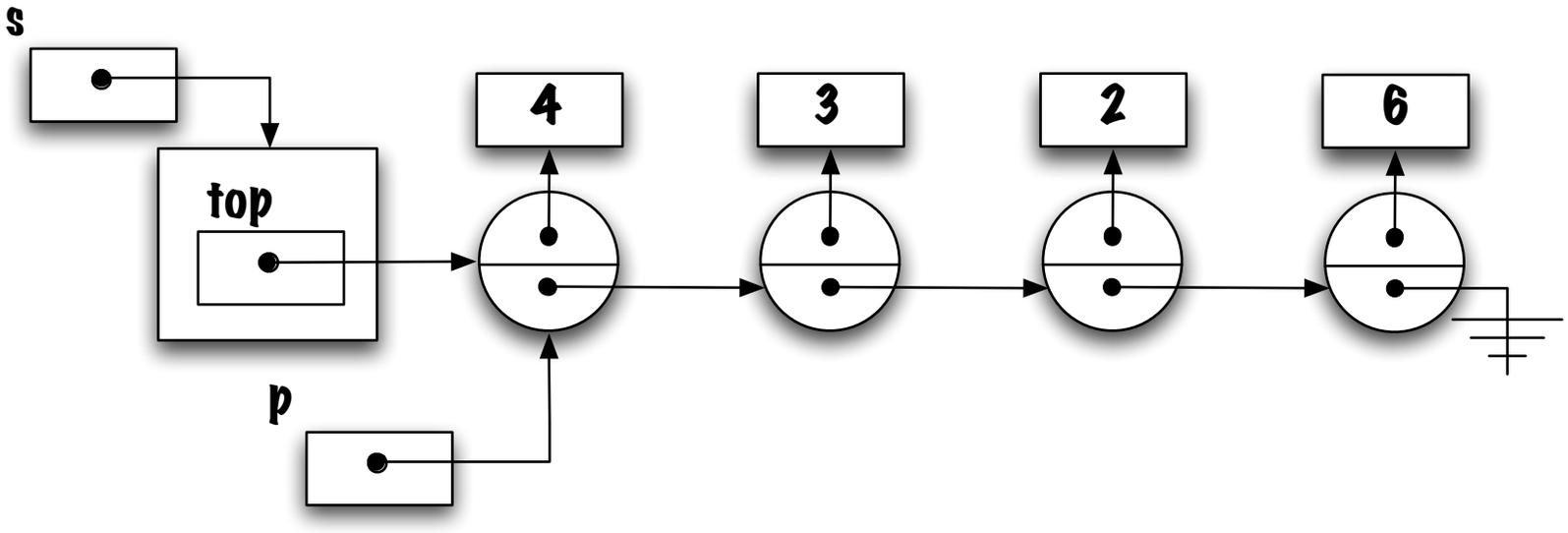
p = p.next

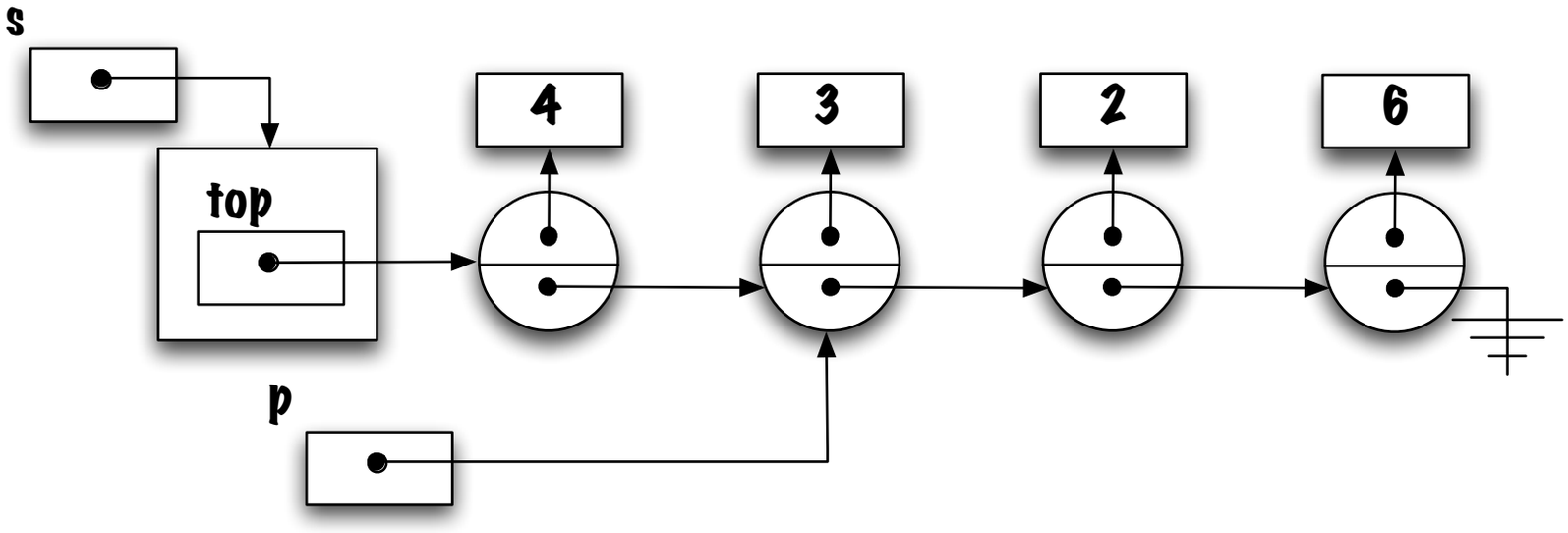
Solution

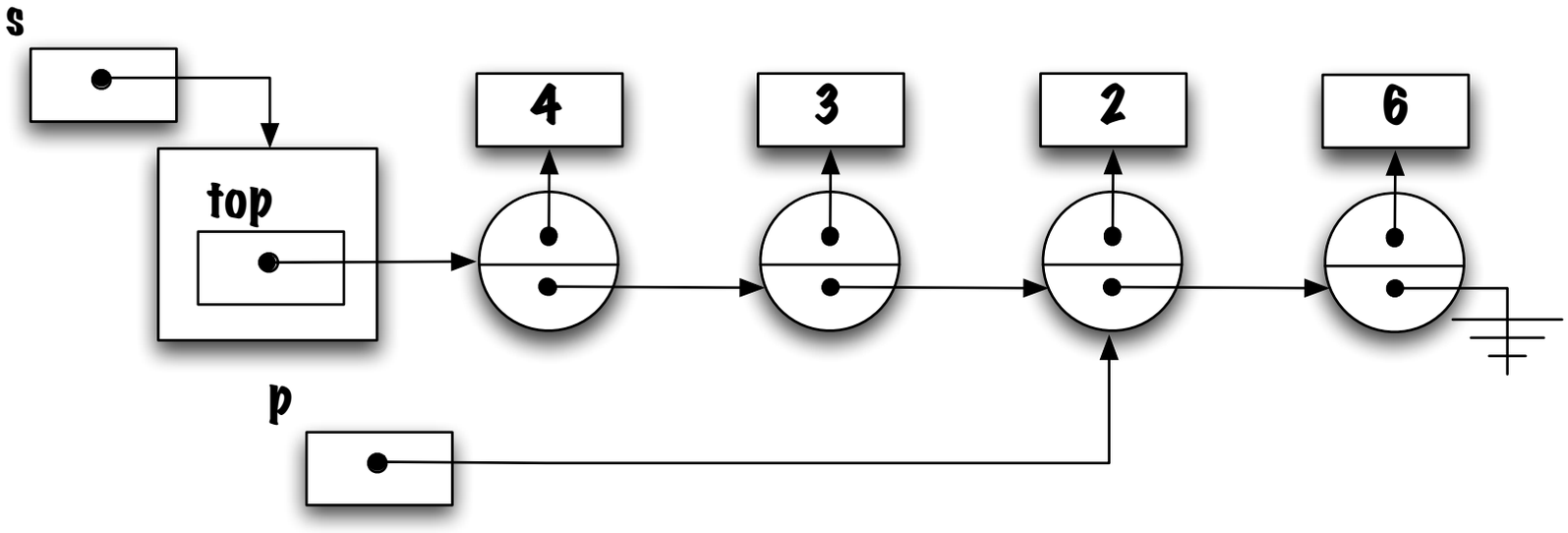
```
public String toString() {
    String res = "[";
    if ( top != null ) {
        Elem p = top;
        res = res + p.value;
        p = p.next;
        while ( p != null ) {
            res = res + "," + p.value;
            p = p.next;
        }
    }
    res = res + "]";
    return res;
}
```

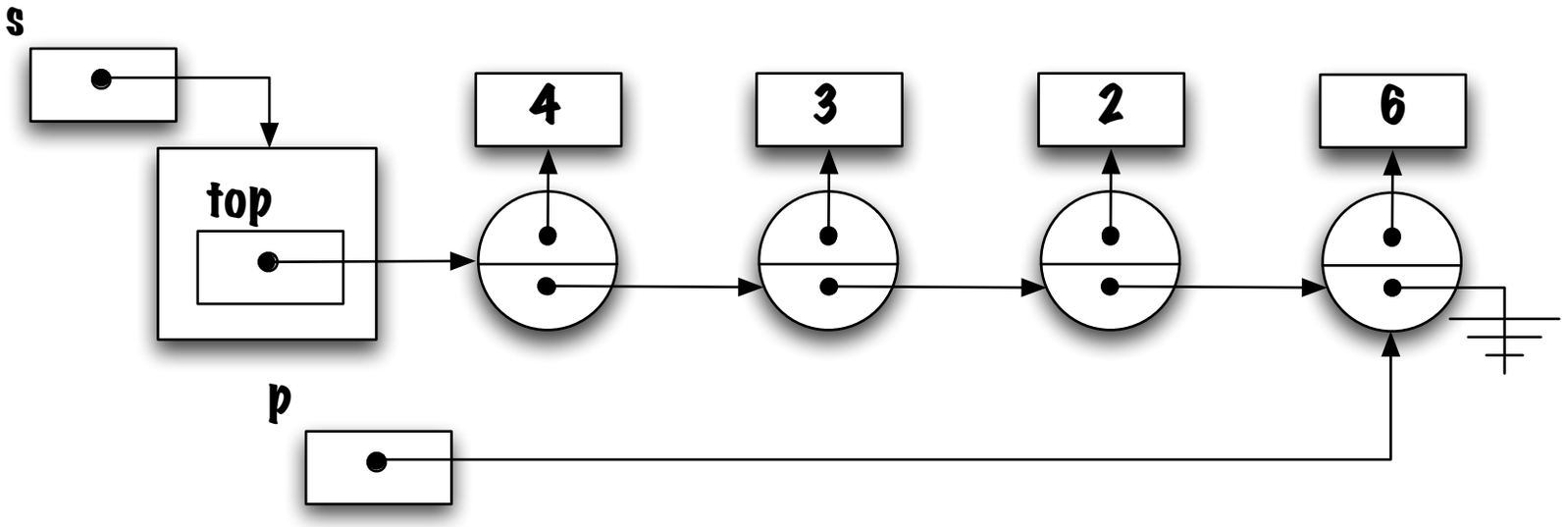

s
top
4
3
2
6

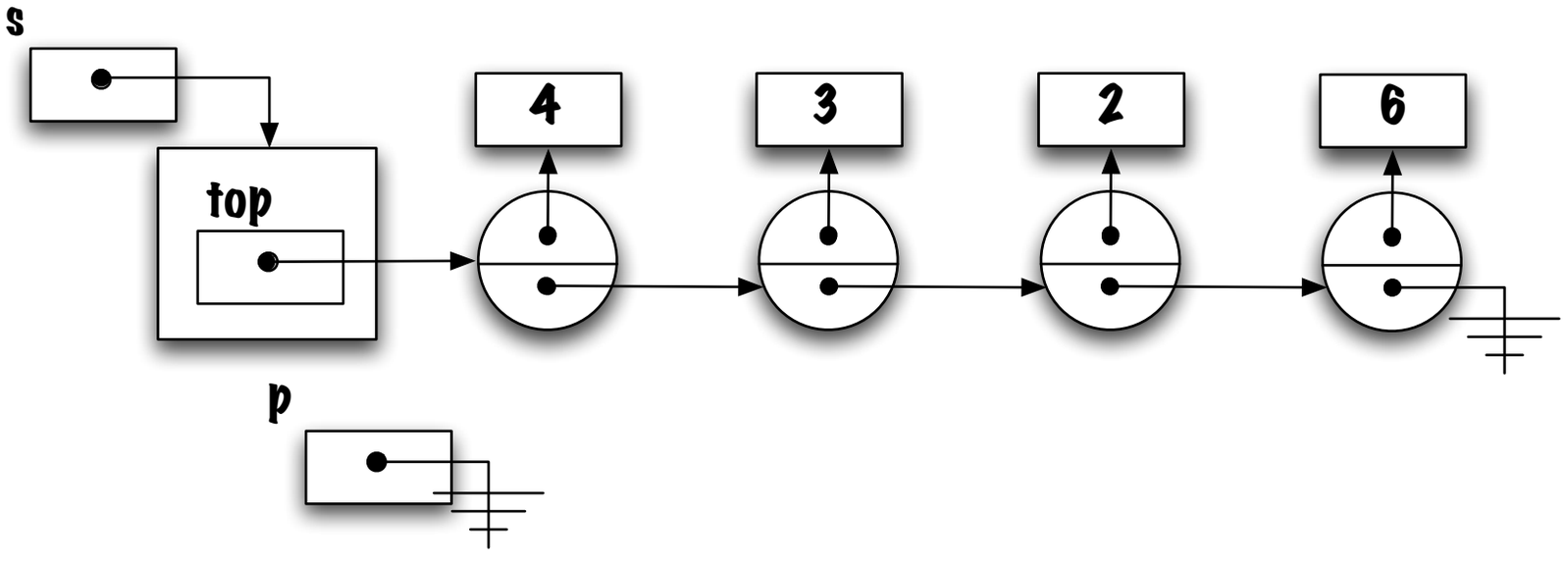

s
top
4
3
2
6
p


s
top
p
4
3
2
6


s
top
4
3
2
6
p


s
top
4
3
2
6
p

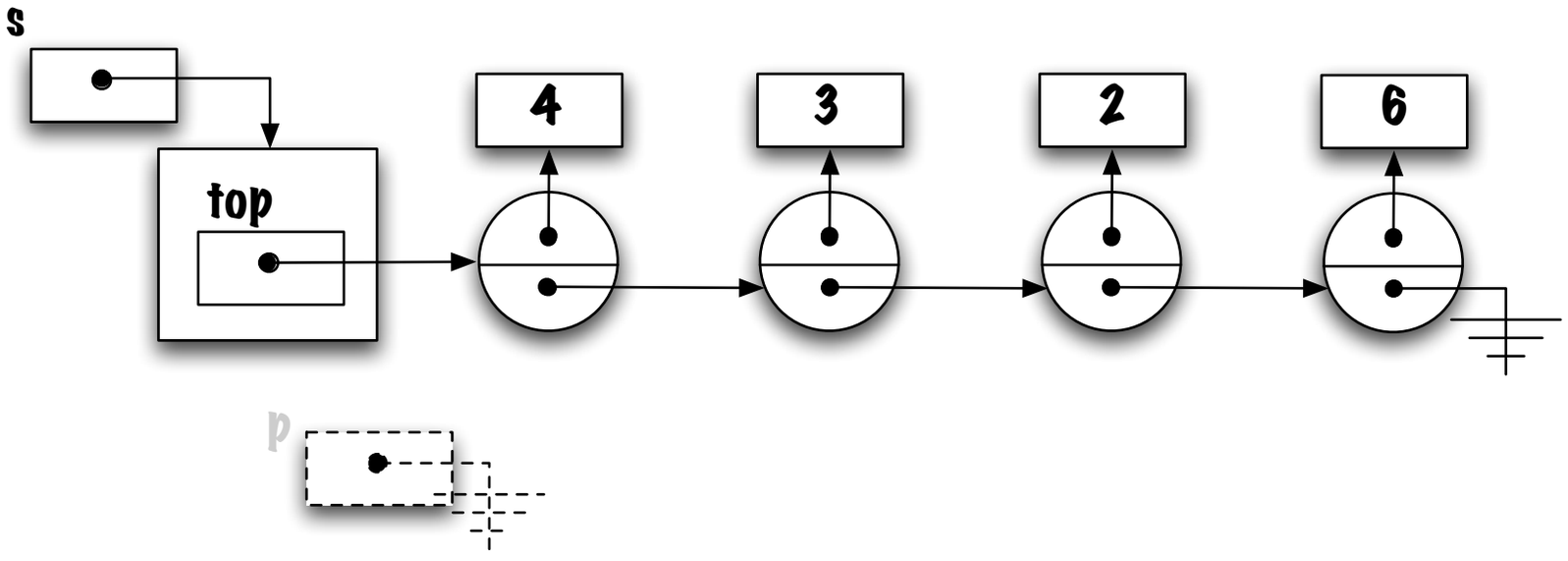

s
top
4
3
2
6
p

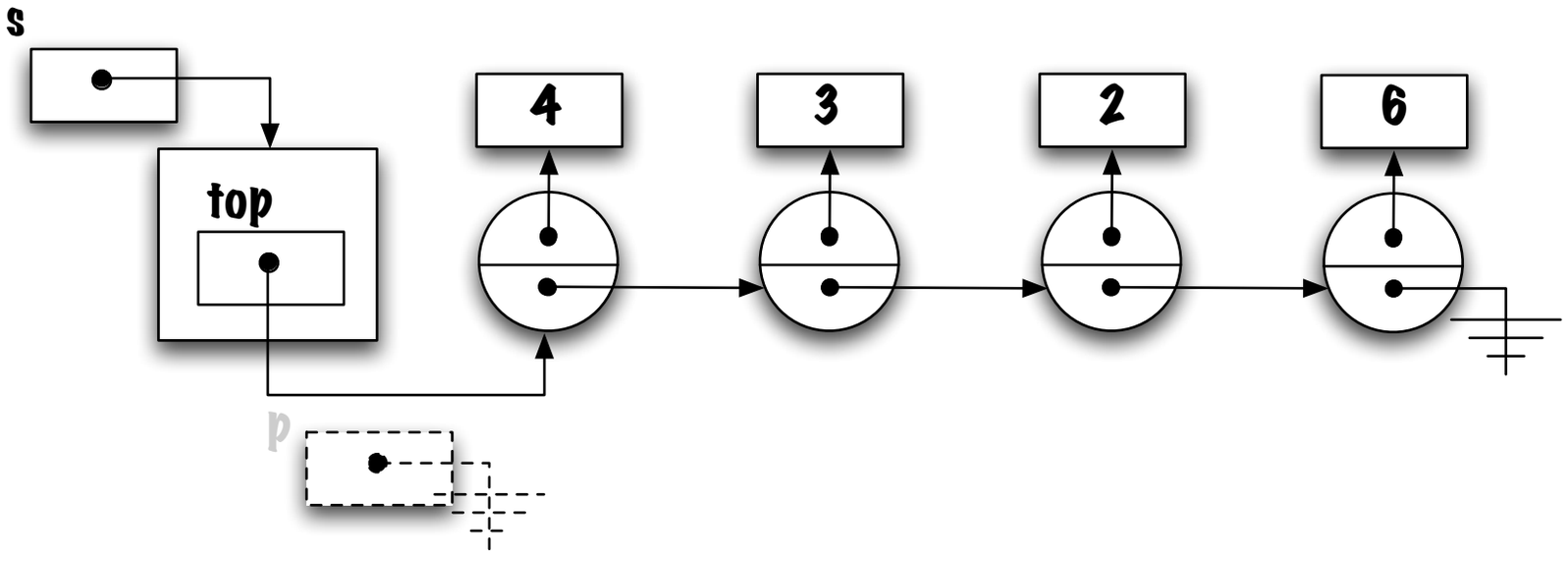

s
top
4
3
2
6
p

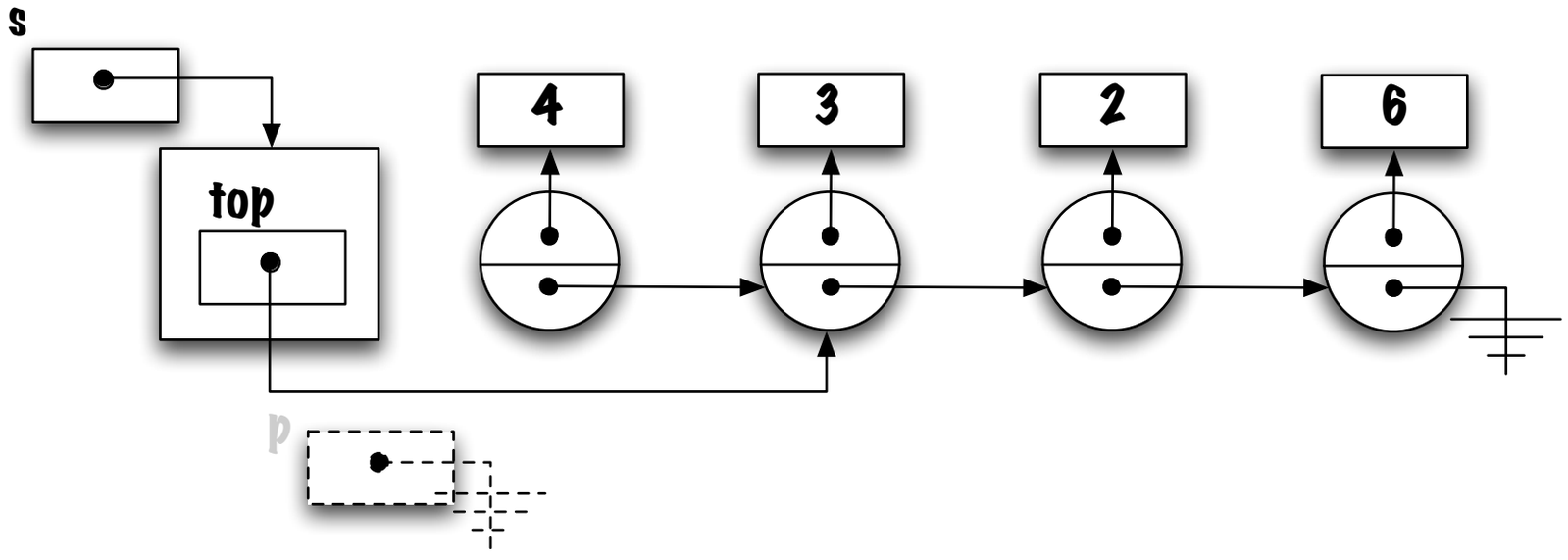
Pitfall!!!

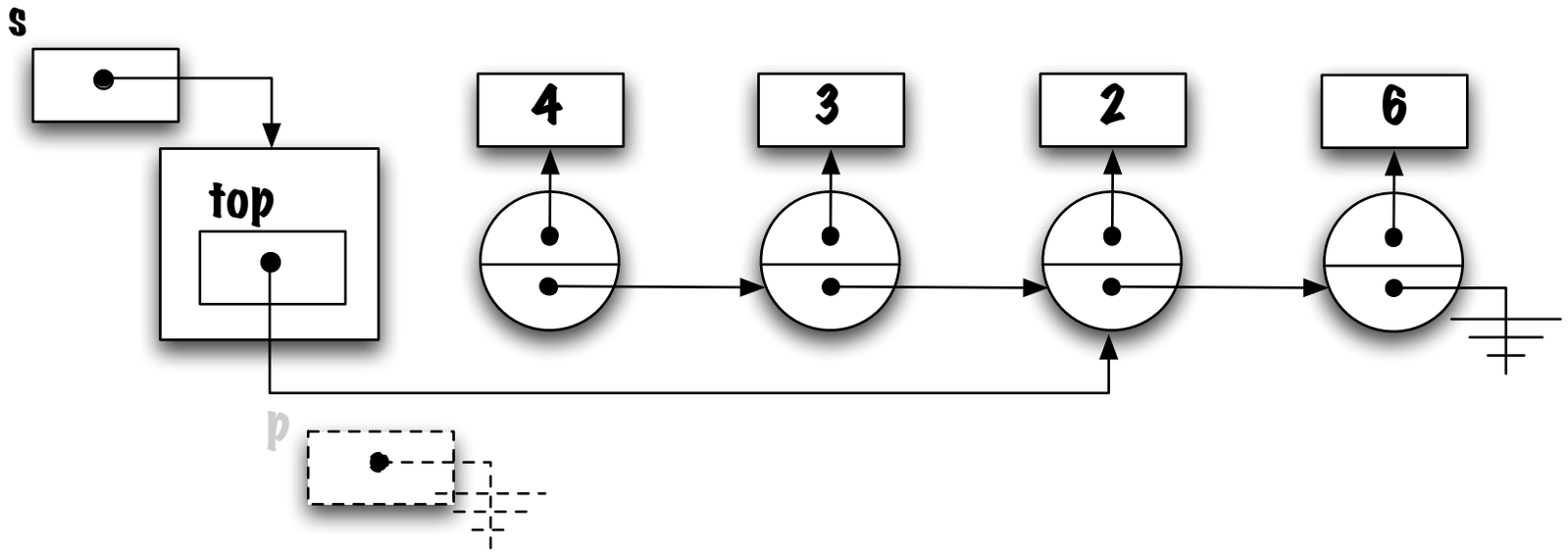
```
public String toString() {
    String res = "[";
    if ( top != null ) {
        res = res + top.value;
        top = top.next;
        while ( top != null ) {
            res = res + "," + top.value;
            top = top.next;
        }
    }
    res = res + "]";
    return res;
}
```

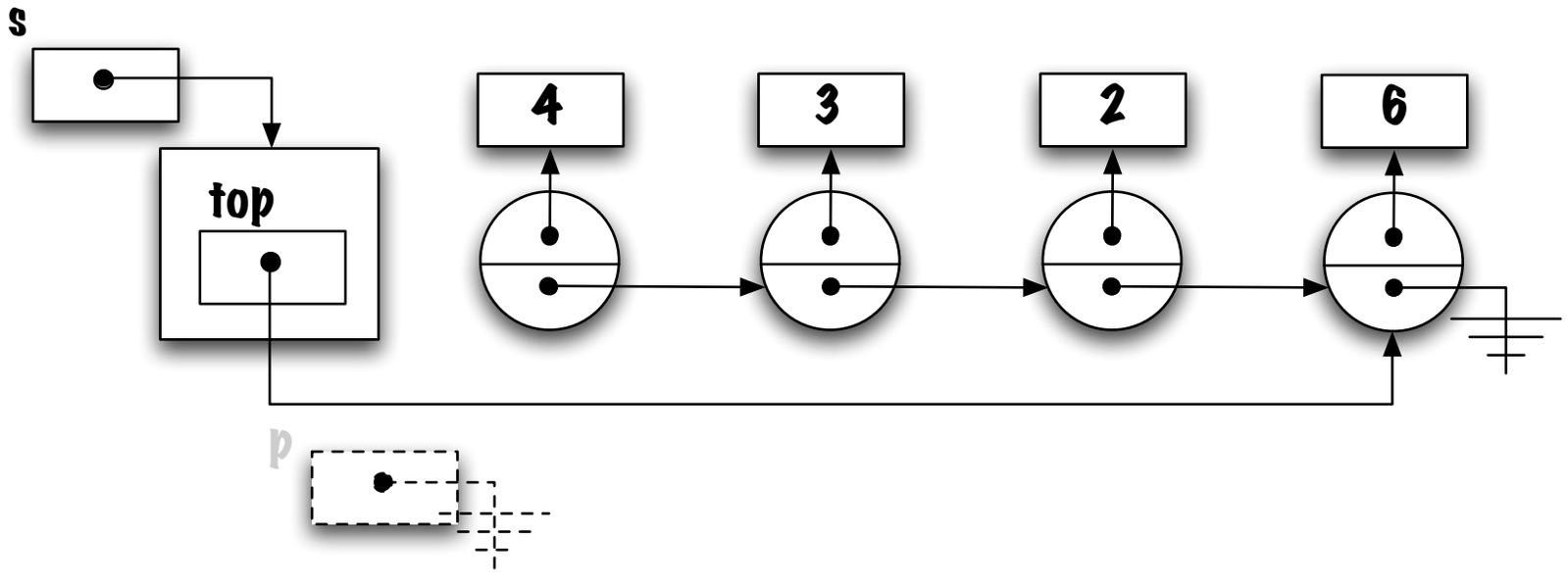
$\Rightarrow$ The above method illustrates a commonly occurring mistake, what is it? What are the consequences?

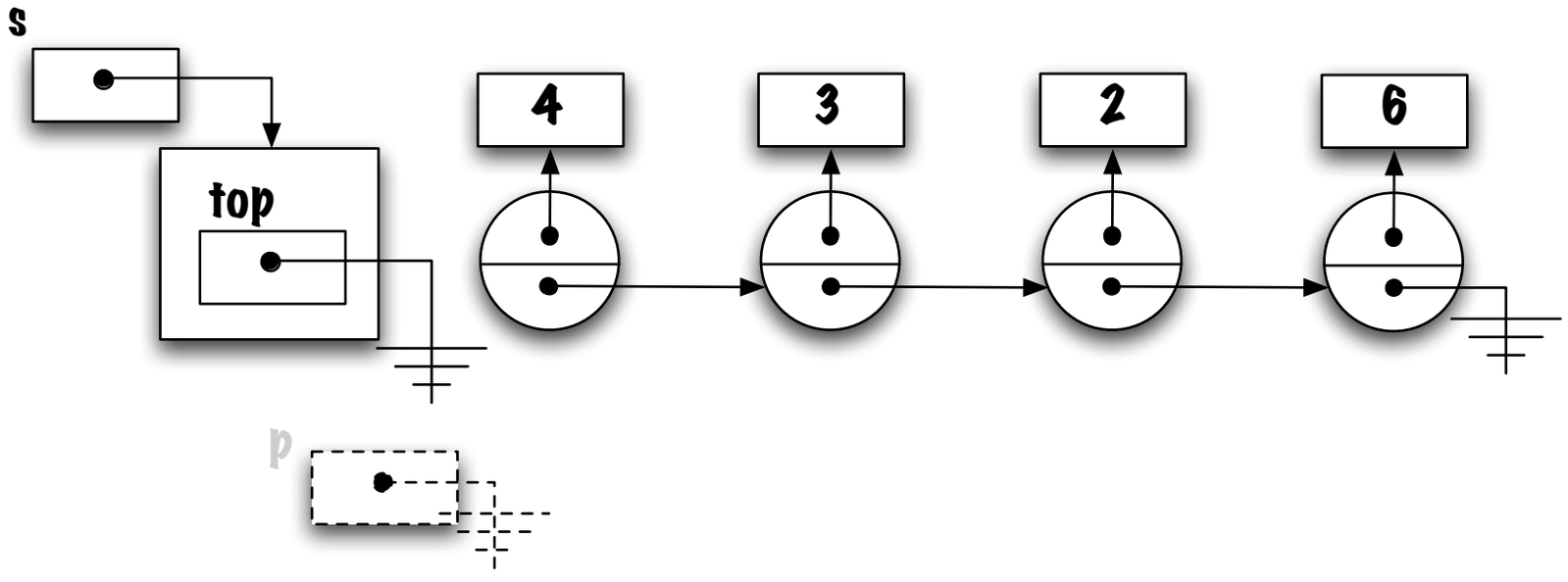

s
top
4
3
2
6
p


s
top
4
3
2
6
p


s
top
4
3
2
6
p


s
top
4
3
2
6
p


s
top
4
3
2
6
p


s
top
4
3
2
6
p

Remark(s)

In the array-based implementation it is equally easy to move in either directions. How about singly-linked lists?

Compare the number of operations that are necessary to retrieve an element by position in the case of an array and a list.