ITI 1521. Introduction à l'informatique II*

Marcel Turcotte
École d'ingénierie et de technologie de l'information

Version du 14 février 2011

Résumé

– Implémentation d'une pile à l'aide d'une liste d'éléments chaînés

*. Ces notes de cours ont été conçues afin d'être visualiser sur un écran d'ordinateur.

Résumé

L'accès aux éléments d'un tableau est très rapide, il nécessite toujours un nombre constant d'opérations.

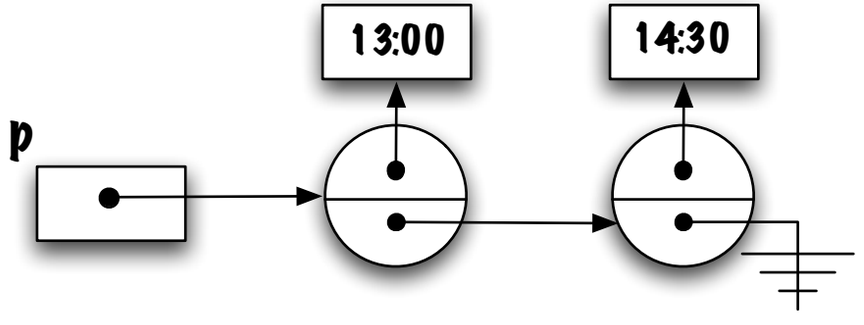
Cependant, puisque les tableaux ont une taille fixe, il y a certaines applications pour lesquelles ils ne sont pas appropriés.

Une technique fréquemment utilisée, afin de contourner cette limitation, consiste à copier les éléments du tableau dans un nouveau tableau, plus grand, et de remplacer l'ancien par le nouveau.

Par contre, cela rend les insertions plus coûteuses (par rapport au temps d'exécution parce qu'il faut copier tous les éléments de l'ancien tableau vers le nouveau) et l'utilisation de mémoire est accrue parce que la taille **physique** de la structure de données sera généralement plus grande que sa taille **logique**.

Structures chaînées

Considérons maintenant certaines structures de données utilisant toujours une quantité de mémoire proportionnelle au nombre d'éléments contenu dans la structure.



Ces structures sont efficaces, au niveau du temps d'exécution (pour certaines opérations), parce qu'elles évitent de recopier les éléments.

Les structures considérées ici sont **linéaires**, c.-à-d. chaque élément possède un prédécesseur et un successeur (sauf pour le premier et le dernier élément).

Au contraire des structures de données à base de tableaux, les éléments de ces structures ne sont pas contigus en mémoire.

Introduction

Étudiez la déclaration suivante (comme toujours, au début les variables d'instance sont ≪public≫, nous corrigerons ce problème sous peu) :

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

Qu'y a-t-il de particulier avec la définition d'**Elem** ?

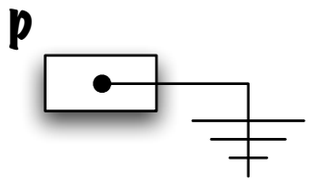
La variable d'instance **next** est une référence vers un objet de la classe **Elem**.

Est-ce valide ?

(essayez par vous même)

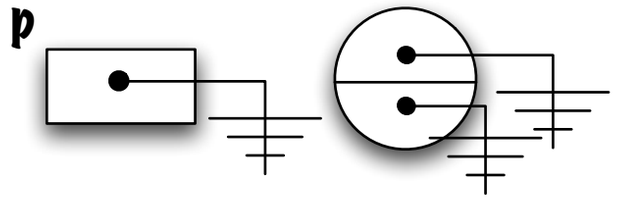
Oui, c'est valide, bien que cette définition semble circulaire.

$\Rightarrow$ À quoi diable cela peut-il bien servir ?



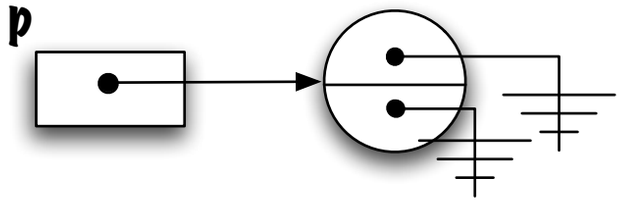
Elem p ;

$\Rightarrow$ Déclaration d'une variable de type (référence) **Elem** ; une variable qui pointe vers un objet de la classe Elem ; la valeur de défaut pour une référence est **null**, ici représentée par le symbole de mise à terre.



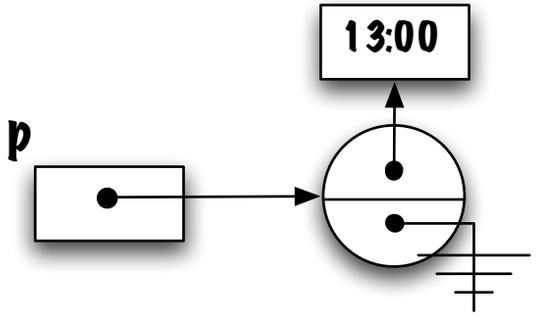
new Elem() ;

$\Rightarrow$ Création d'un objet (instance de) **Elem**. J'utiliserai toujours des cercles afin de représenter les objets de la classe **Elem**. Par convention, la partie du haut représente la variable d'instance **value** alors que la partie du bas représente la variable **next**.



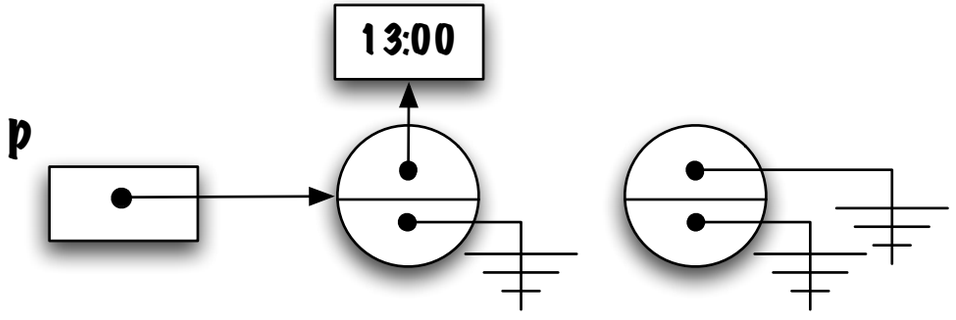
p = new Elem() ;

$\Rightarrow$ Affectation de la référence à la variable **p**. Comment change-t-on le contenu de la variable d'instance **value** de l'objet nouvellement créé ?



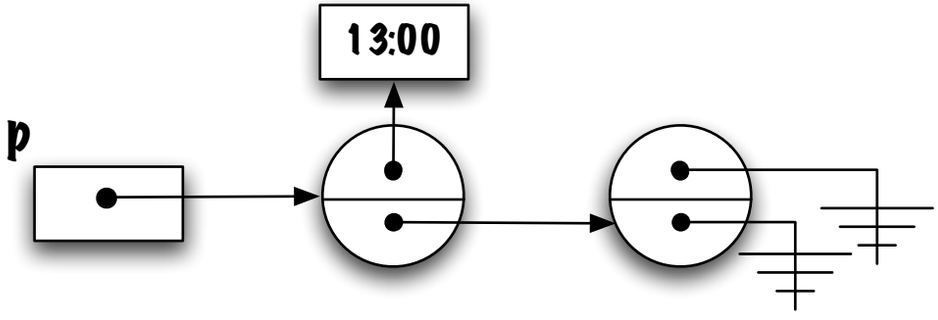
p.value = new Time(13, 0, 0) ;

$\Rightarrow$ La variable **value** de l'objet désigné par **p** désigne l'objet nouvellement créé, (**new Time(13, 0, 0)**). Il faut bien entendu que la visibilité de la variable **value** soit ≪public≫ bien entendu.



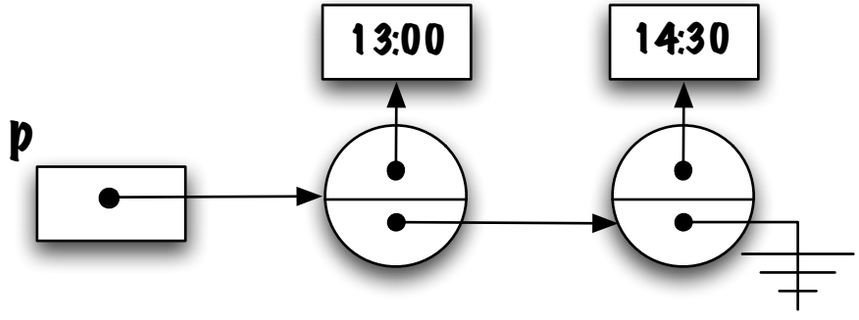
new Elem() ;

⇒ Création d'un objet de la classe **Elem**. Comment chaîne-t-on ces éléments les uns aux autres ?



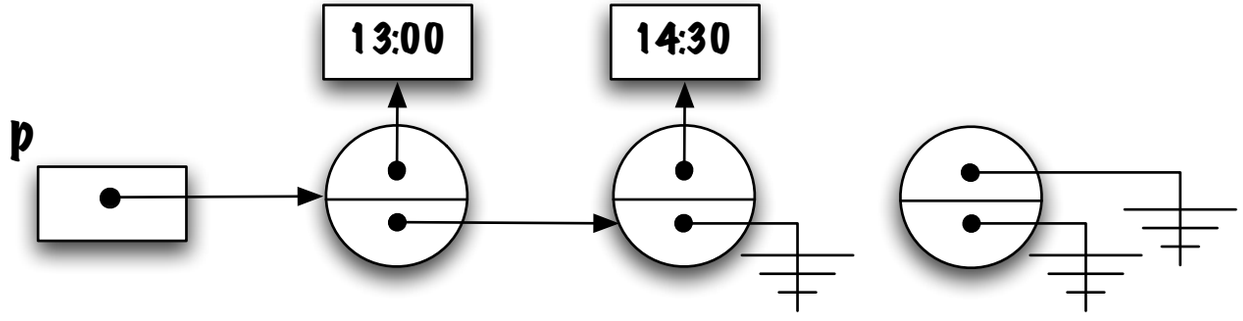
p.next = new Elem() ;

$\Rightarrow$ Affectation de la référence à la variable d'instance **p.next** (une variable de type **Elem**). On souhaite maintenant sauvegarder un objet de la classe **Time** dans cet élément. Comment fait-on ?



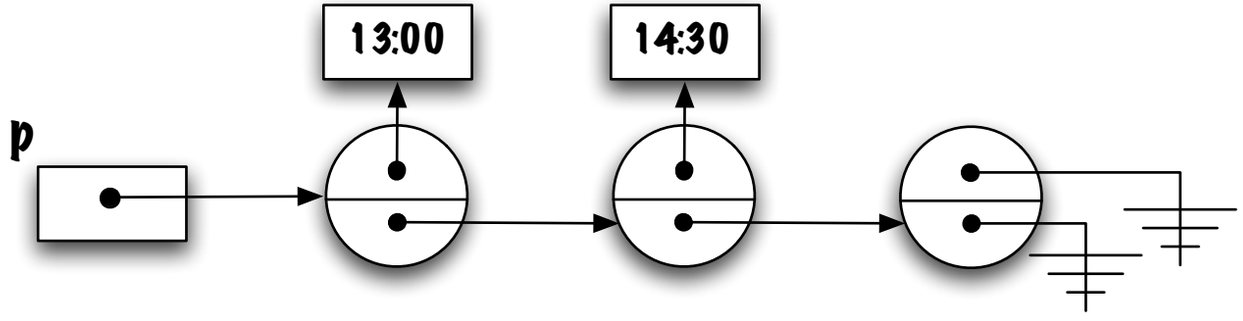
p.next.value = new Time(14, 30, 0) ;

$\Rightarrow$ On change la valeur de la variable **value**, de l'instance pointée par **p.next**.



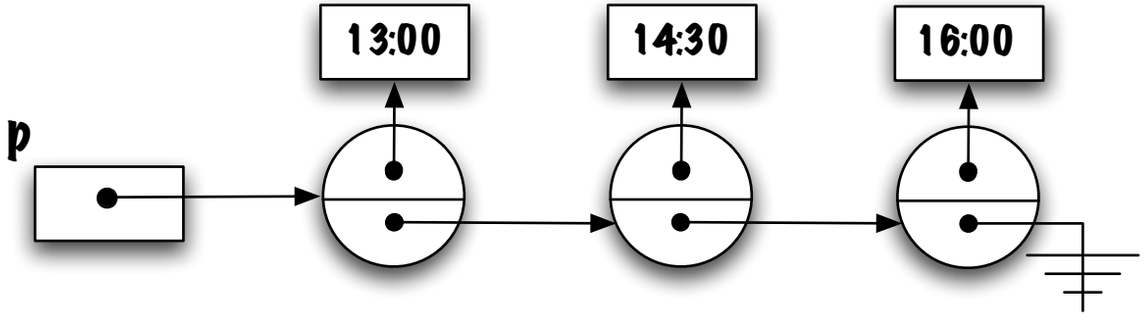
new Elem()

$\Rightarrow$ Création d'un nouveau noeud (on appelle noeuds les éléments d'une structure chaînée comme celle-ci). Comment raccorde-t-on cet élément à la chaîne existante ?



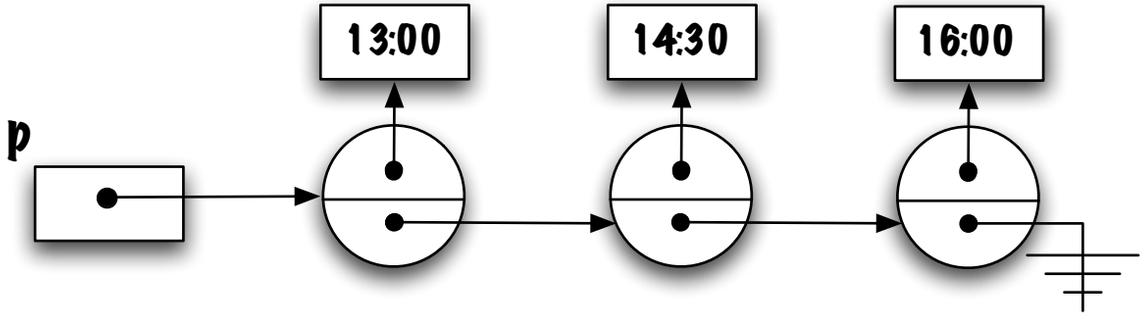
p.next.next = new Elem() ;

$\Rightarrow$ La référence vers ce nouveau noeud est mise dans la variable d'instance **p.next.next**. On souhaite maintenant sauvegarder un objet dans ce noeud, comment fait-on ?



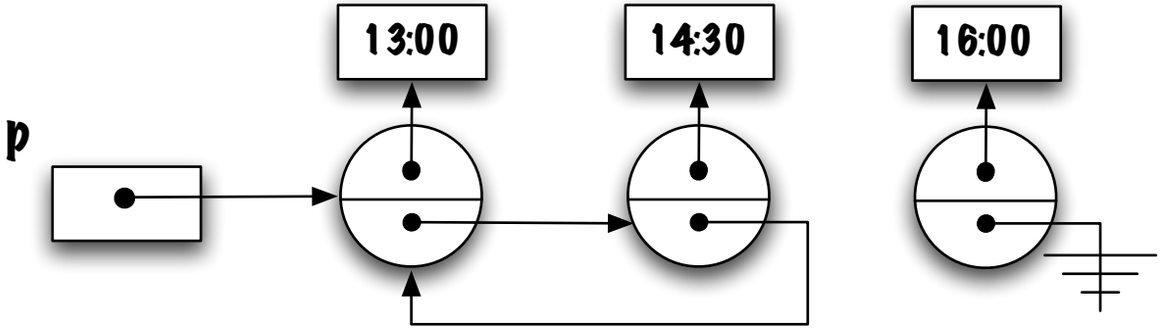
p.next.next.value = new Time(16, 0, 0) ;

$\Rightarrow$ Change le contenu de variable value, de l'objet nouvellement créé.



Qu'adviendra-t-il de chaîne ci-haut si l'énoncé suivant est exécuté ?

p.next.next = p ;



p.next.next = p ;

Hum . . .
– Une structure circulaire a été créée !
– Le dernier élément n'est plus accessible ;
– Il sera récupéré par le gestionnaire de mémoire ; **gc()**.

$\Rightarrow$ Tout ceci constitue la base des structures chaînées : des **informations** (valeurs) sont liées les unes aux autres par des **liens** (références).

Structures (de données) chaînées

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

Les structures de données chaînées, telles que celle-ci, nous permettent :
- de représenter des structures de données linéaires, telles que les piles, les files et les listes ;
- elles utilisent toujours une quantité de mémoire proportionnelle au nombre d'éléments ;
- tout ceci est rendu possible parce que la classe déclare une variable d'instance dont le type est une référence vers un objet de cette même classe.

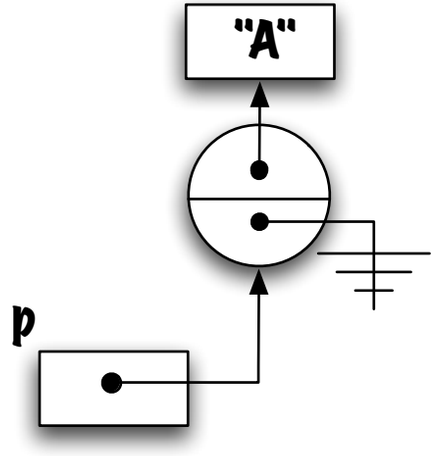
$\Rightarrow$ Lorsque les structures sont linéaires comme celles-ci, on parle alors de listes chaînées.

Le constructeur usuel de la classe Elem,

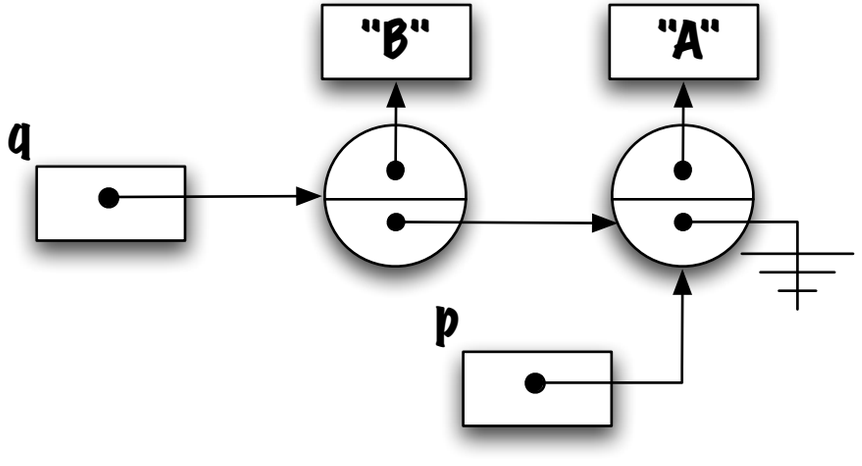
```
public class Elem {

  public Object value;
  public Elem next;

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

et son usage habituel,

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




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



Piège !

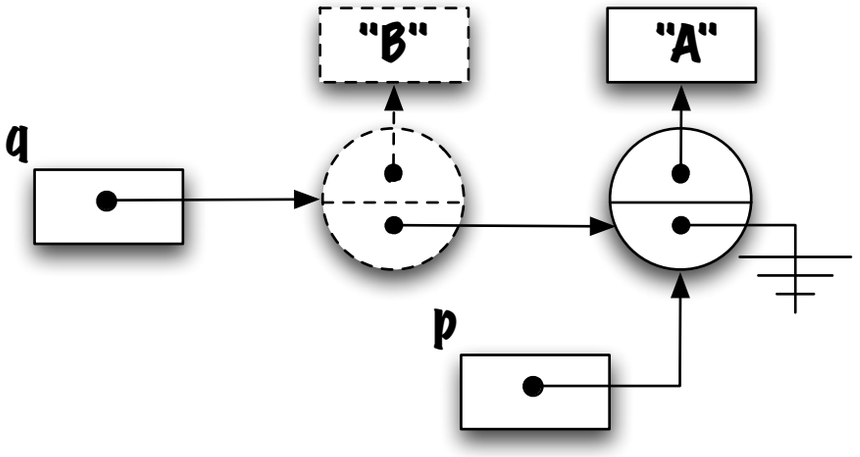
L'exemple suivant illustre une erreur fréquemment observée :

```
Elem p = null;
Elem q = null;

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

q.next = p;
```

bien que les énoncés ci-haut soient syntaxiquement corrects, ils causeront une erreur à l'exécution.

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

$\Rightarrow$ Pourquoi ?



Le diagramme illustre la situation. L'intention était, peut-être, de créer une chaîne telle que **q.next** désigne le même objet que **p**, cependant, **q** ne désigne aucun objet, alors l'accès **q.next** causera l'exception **NullPointerException**.

Solutions :

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

ou

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

Précaution

De façon générale, lorsqu'on utilise une variable d'instance d'un objet, il est plus prudent de s'assurer d'abord de l'existence de l'objet.

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

$\Rightarrow$ Cette construction reviendra souvent dans nos programmes.

Stack : structure chaînée

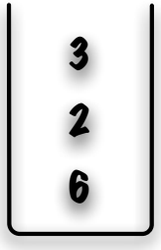
Utilisons des éléments chaînés afin de réaliser l'interface Stack.

```
public class LinkedStack implements Stack {

    public boolean empty() {

    }
    public void push( Object o ) {

    }
    public Object peek() {

    }
    public Object pop() {

    }
}
```

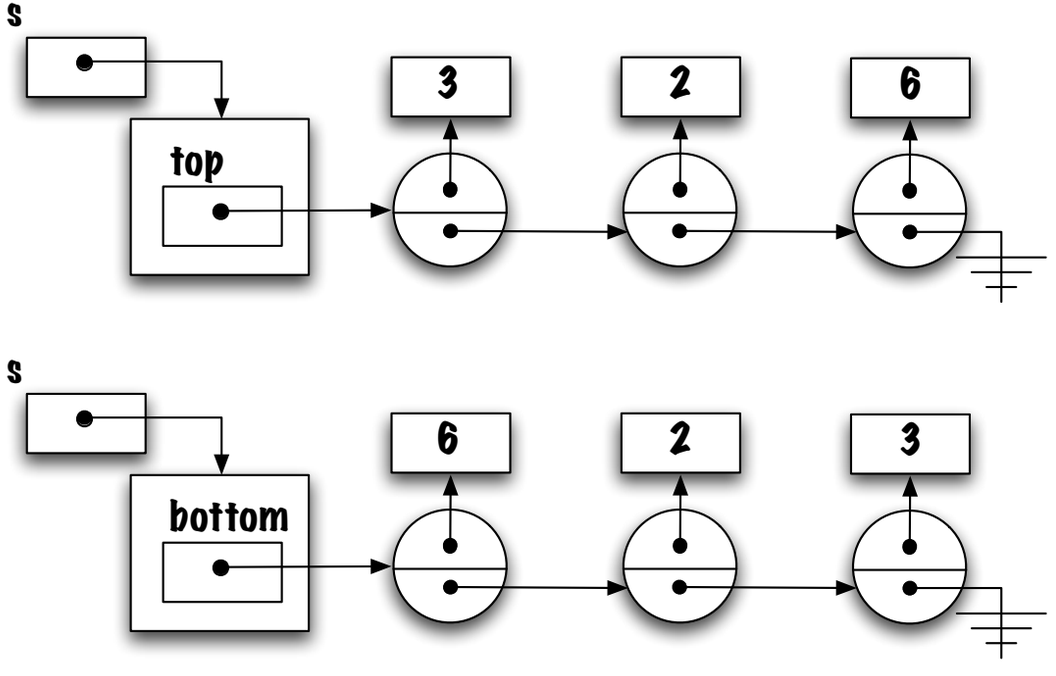
Stack : structure chaînée

Quelles sont les variables d'instances ?

```
public class LinkedStack implements Stack {

    private Elem bottom; // ou encore top?

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



Laquelle des deux stratégies suivantes est préférable ?



La première implémentation est préférable parce tous les accès (**push** et **pop**) se fond à une seule extrémité.

Dans le second cas, il faudrait traverser toute la liste afin d'ajouter ou de retirer un élément.

Plus il y a d'éléments, plus ça serait coûteux !

Classe Elem (0/3)

La visibilité **public** des variables value et next n'est pas acceptable. Quelles solutions s'offrent à nous ?

Classe 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;
  }
}
```

Classe Elem (2/3)

```
class Elem {

  protected Object value;
  protected Elem next;

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

Elem est une classe de premier niveau dont la visibilité est ≪package≫. Si les classes **LinkedStack** et **Elem** font partie du même ≪package≫ alors la classe **LinkedStack** aura accès aux variables d'instance de la classe **Elem**.

Classe 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;

    // ...
}
```

Classe Elem (3/3)

Elem est une classe **imbriquée** de la classe **LinkedStack**.

Bien que la visibilité de la classe et de ses variables soit **private**, la classe **LinkedStack** a accès aux variables d'instance de la classe **Elem** parce que son implémentation est imbriquée.

Pour l'instant, les classes imbriquées seront ≪static≫. Nous les utiliserons comme si elles étaient des classes de premier niveau sauf que 1) la déclaration est imbriquée et 2) l'implémentation est accessible à la classe extérieure.

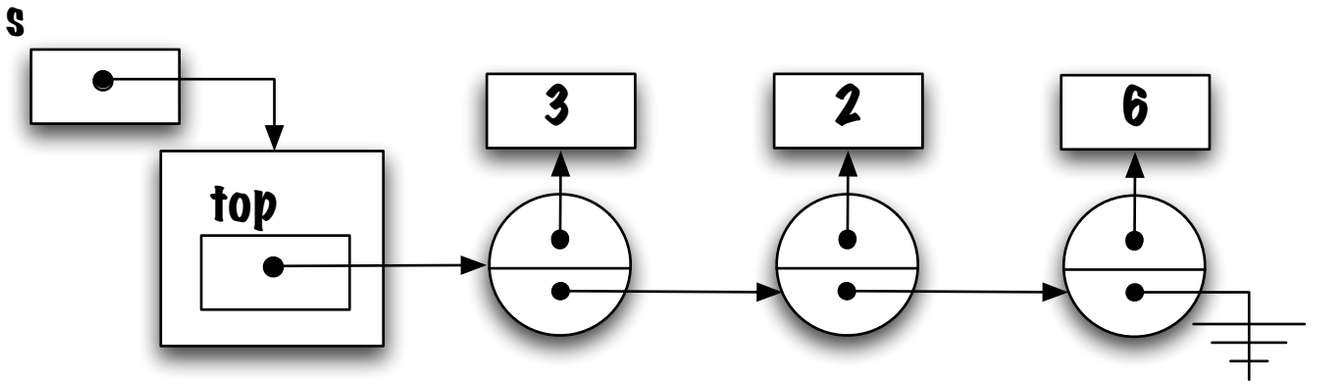
Plus tard, nous verrons qu'il existe une seconde catégorie de classes imbriquées.

≪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:



    }
}
```



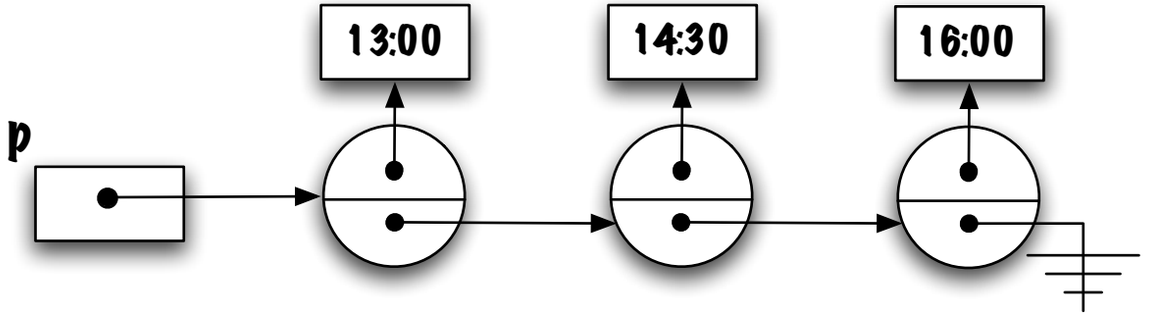
s.peek() devrait retourner la valeur 3.

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

Complétez l'implémentation.

top.value ;

Détecter la fin d'une liste



Regardez bien le diagramme et suggérez une expression permettant de déterminer la fin de la structure chaînée.

Ce qui distingue le noeud en position terminale des autres, c'est la valeur **null** de la variable **next**.

Traverser une liste

Implémentation de la méthode toString().

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

Afin de créer une chaîne de caractères représentant le contenu d'un objet de cette classe, il nous faut parcourir la liste d'un bout à l'autre, ce que nous appellerons ≪traverser≫ la liste.

De même, afin de comparer deux listes il nous faut traverser deux listes simultanément (voir méthode **equals(Object other)**).

Comparons l'implémentation à base de tableau et la liste.

Tableau

```
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;
}
```

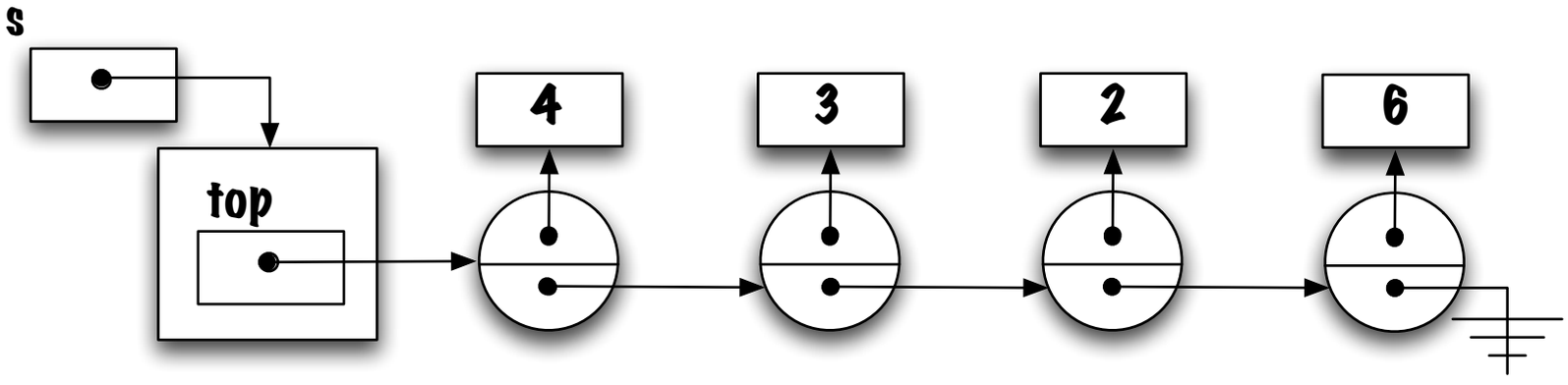
Complétez

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

Complétez

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

Quelle expression déterminera si la pile est vide ?

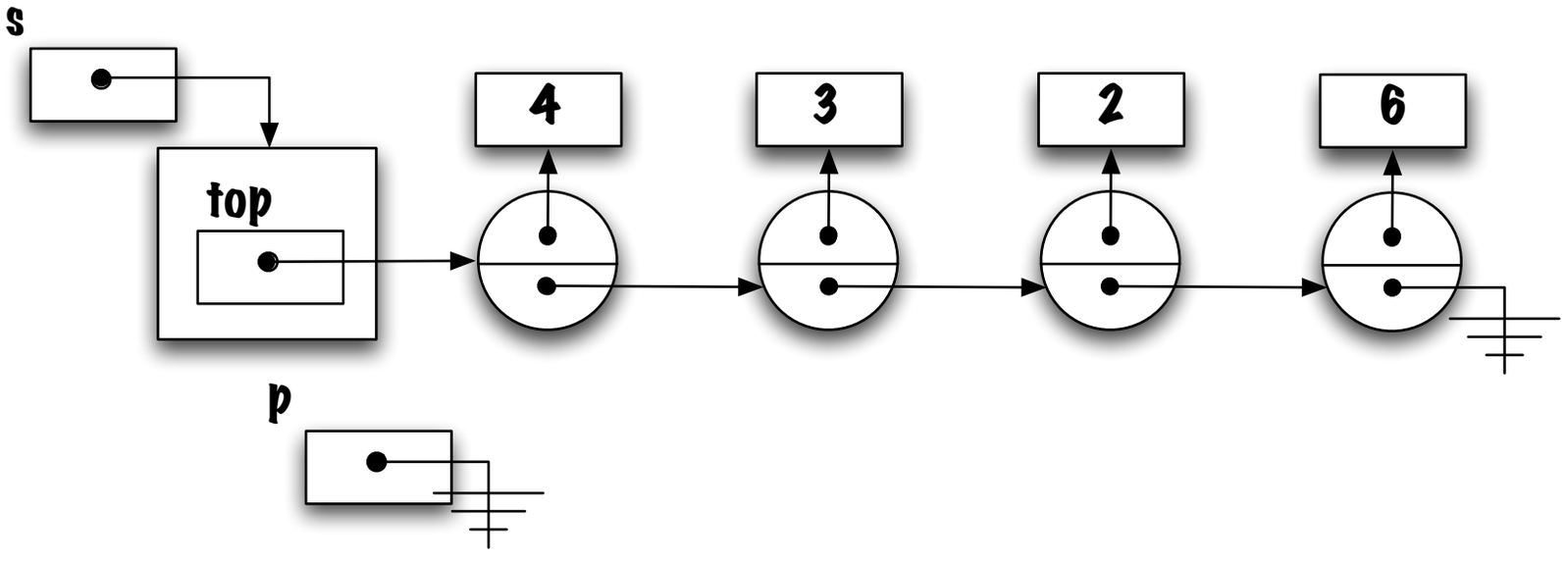


top == null

Complétez

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

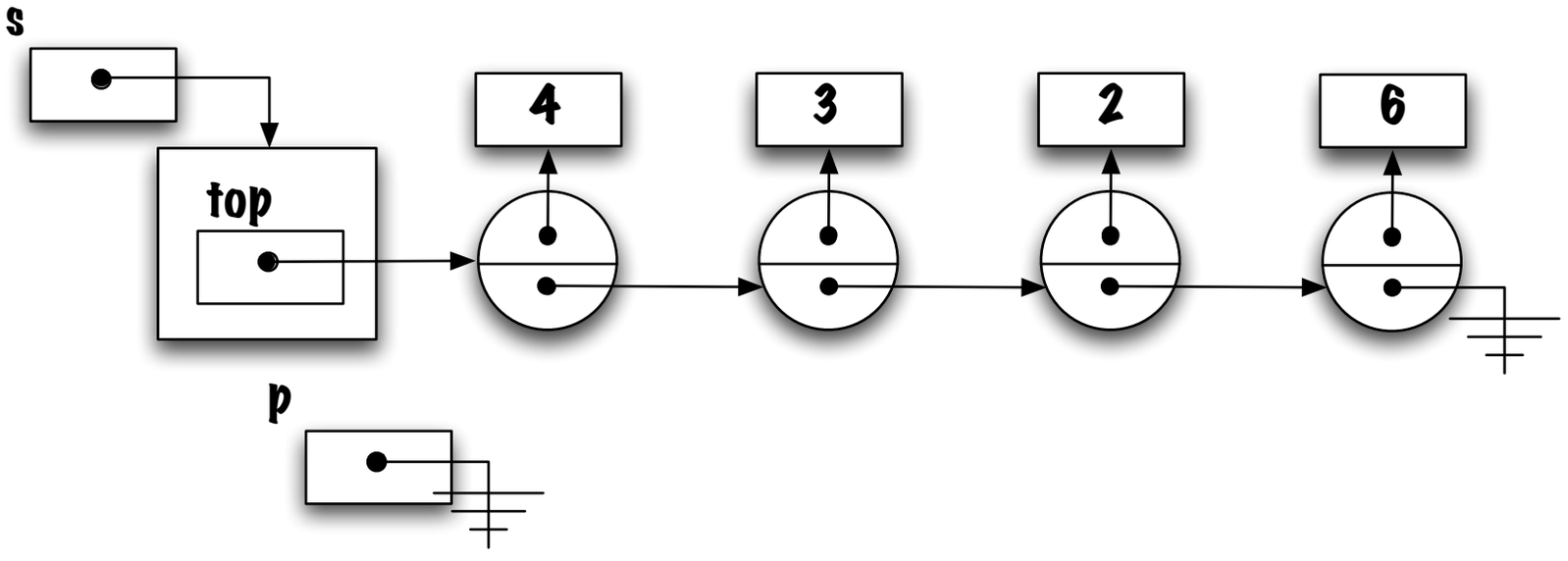
La variable **p** aura un rôle semblable à l'index dans l'implémentation à l'aide d'un tableau. Elle nous permettra d'accéder aux éléments de la pile, un à un. Quel est sont type ? **Elem**.

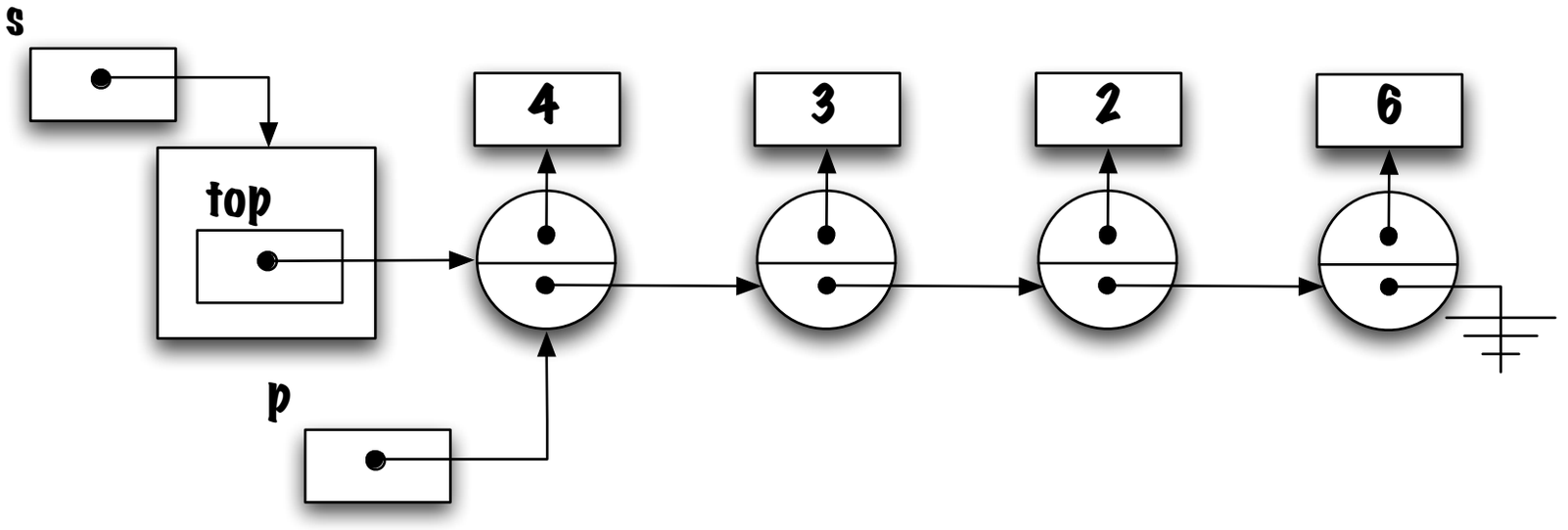

s
top
4
3
2
6
p

Complétez

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

Quelle est sa valeur initiale ?


s
top
4
3
2
6
p

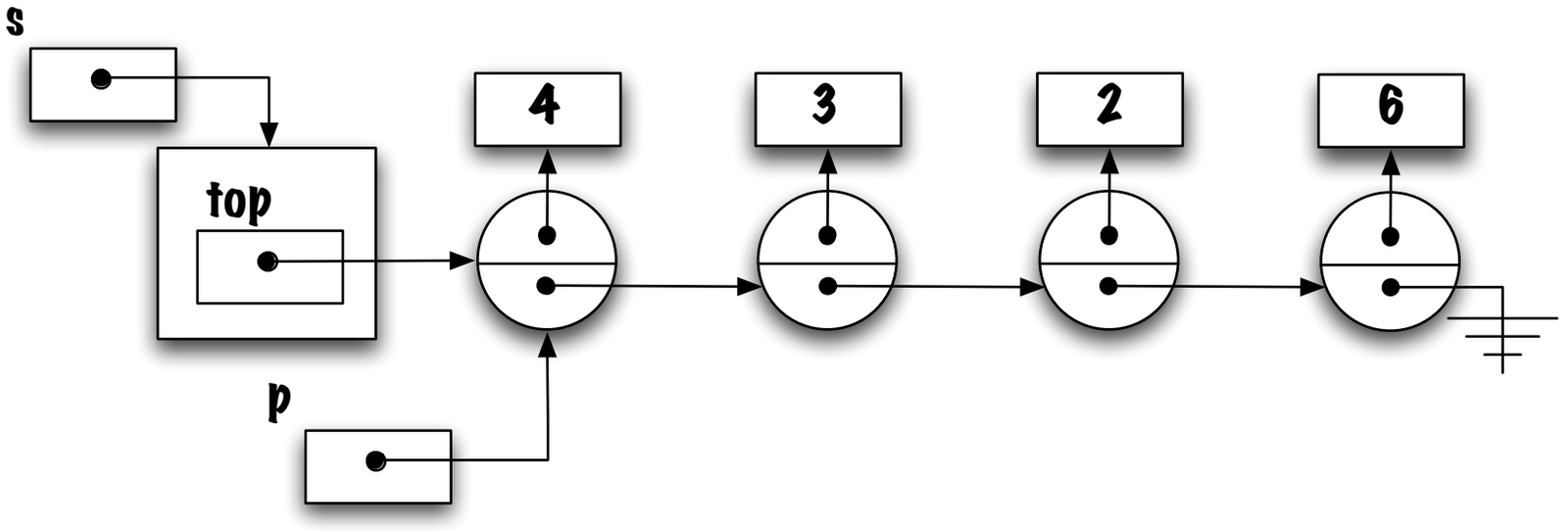


top, ainsi **p** et **top** désignent toutes les deux le même objet, celui du dessus.

Complétez

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

Comment accède-t-on à la valeur de l'élément du dessus ?

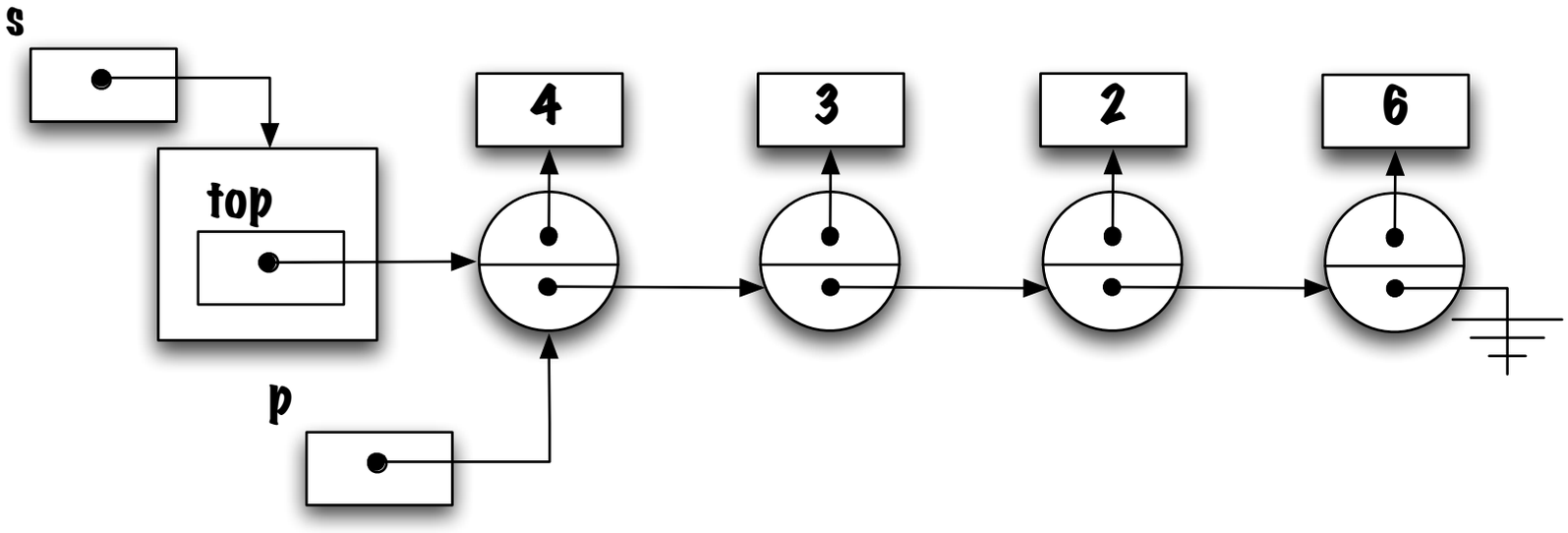


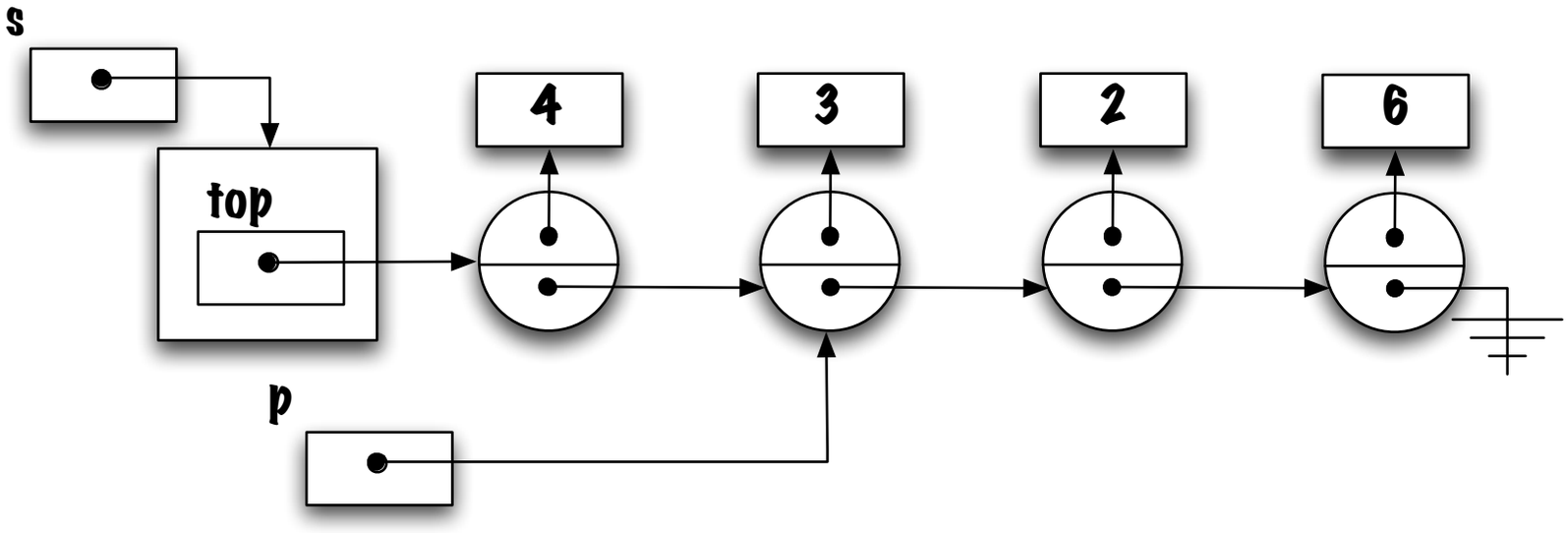
top.value.toString(), **p.value.toString()**, ou tout simplement **top.value** ou **p.value**.

Complétez

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

On souhaite maintenant déplacer la référence **p** d'une position vers l'avant, l'équivalent du **i = i+1**. Comment fait-on ?


s
top
p
4
3
2
6

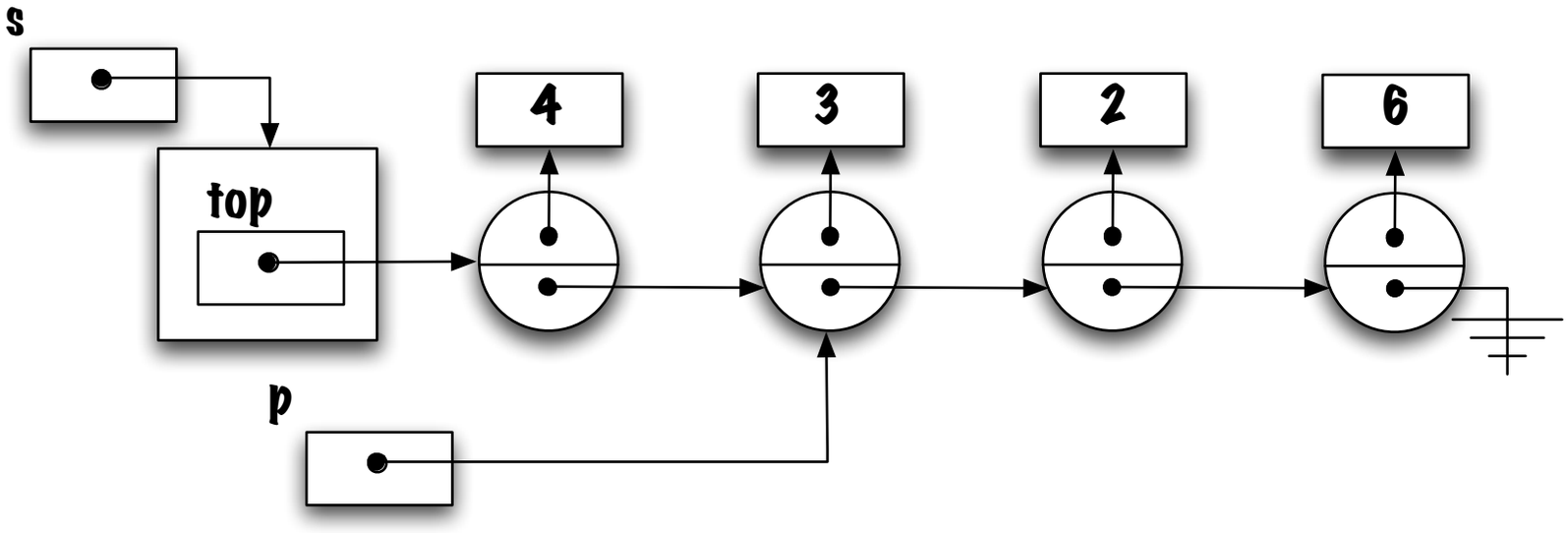


p = p.next.

Complétez

```
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;
}
```

Nous irons dans la boucle tant que la fin de structure chaînée n'a pas été détectée ? Quel test ?



p != null

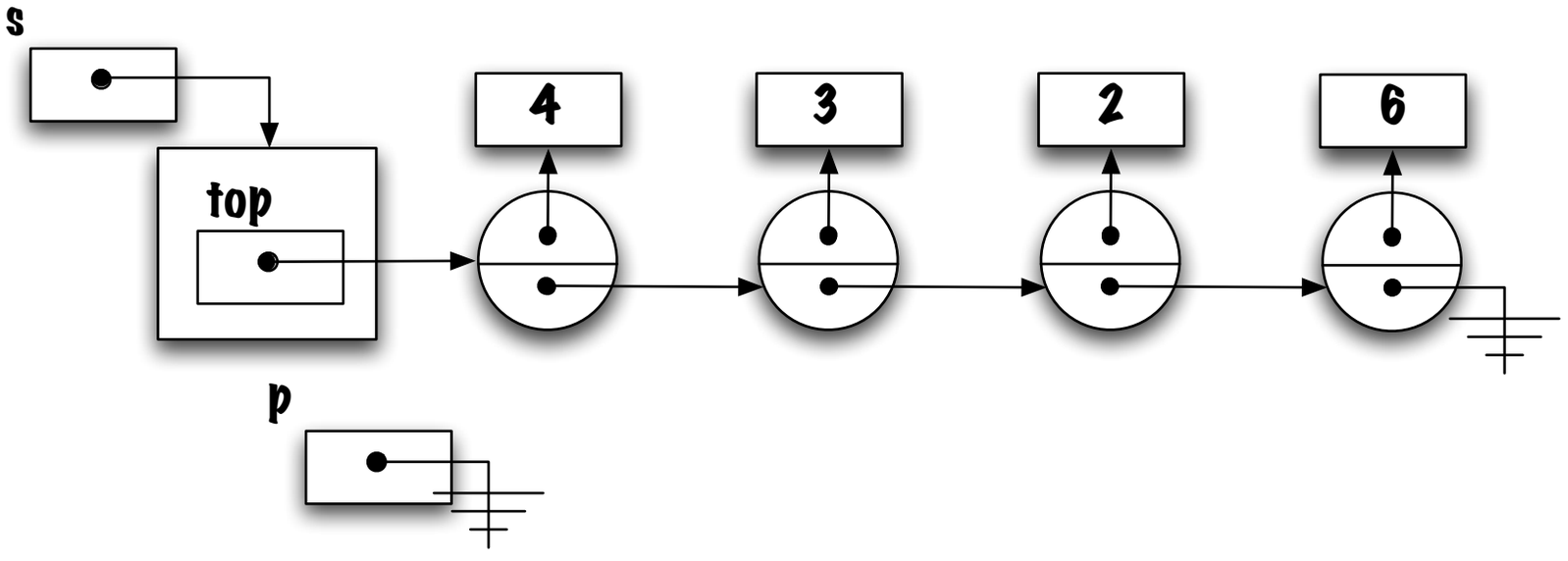
Complétez

```
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;
}
```

Complétez

```
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;
}
```

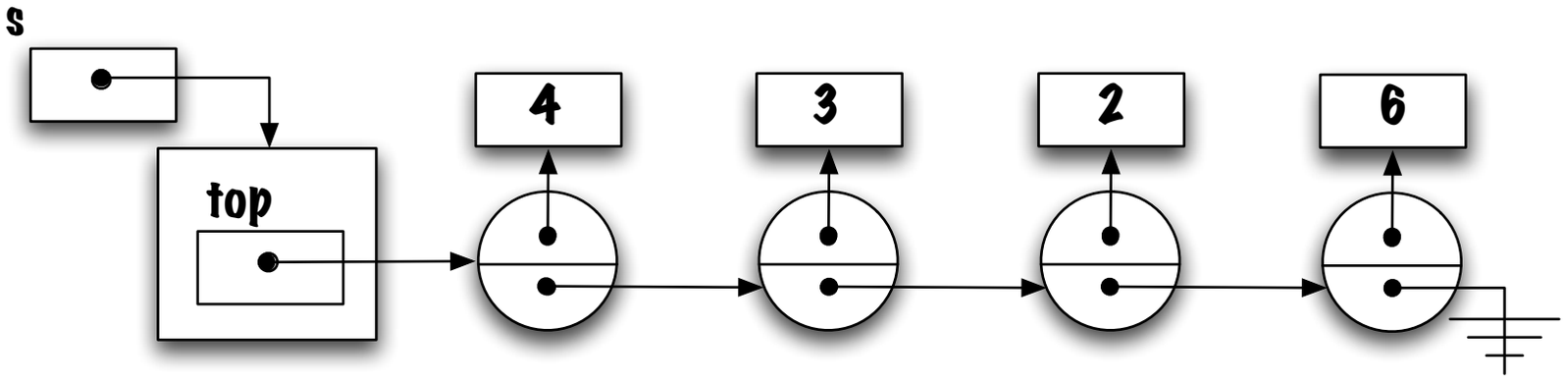
Comment se déplace-t-on vers l'avant (l'équivalent du $\mathbf{i = i + 1}$) ?

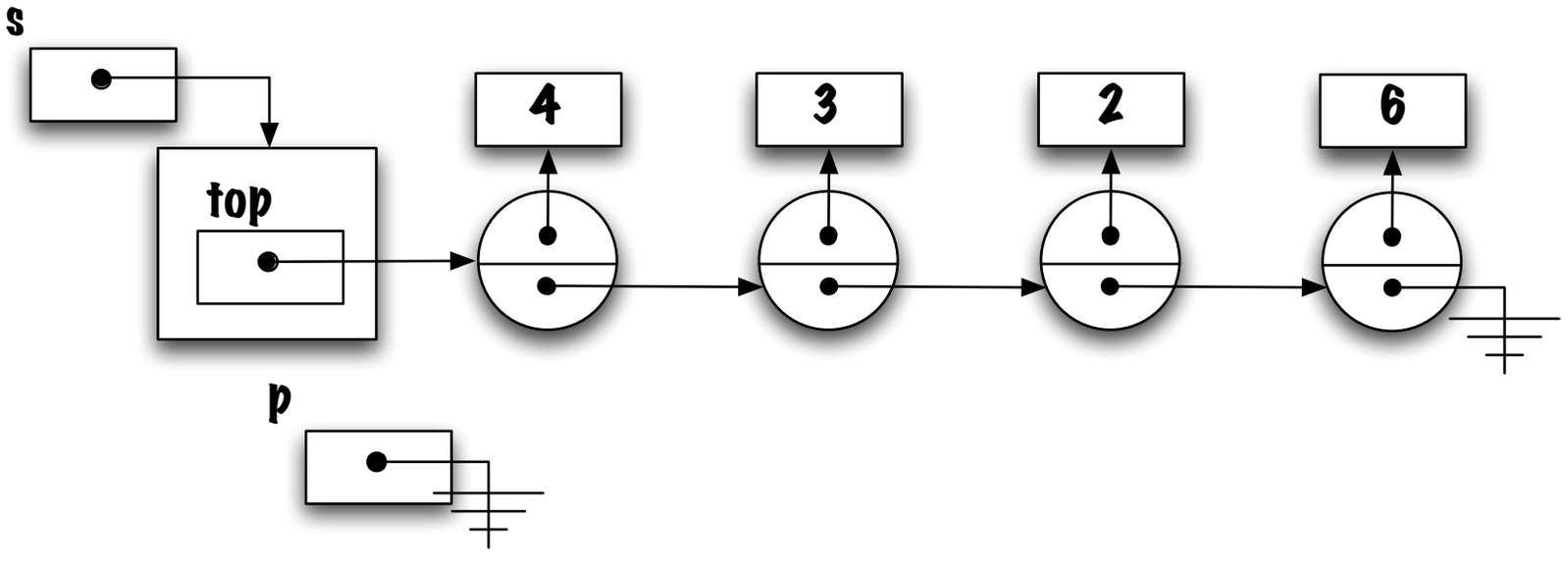


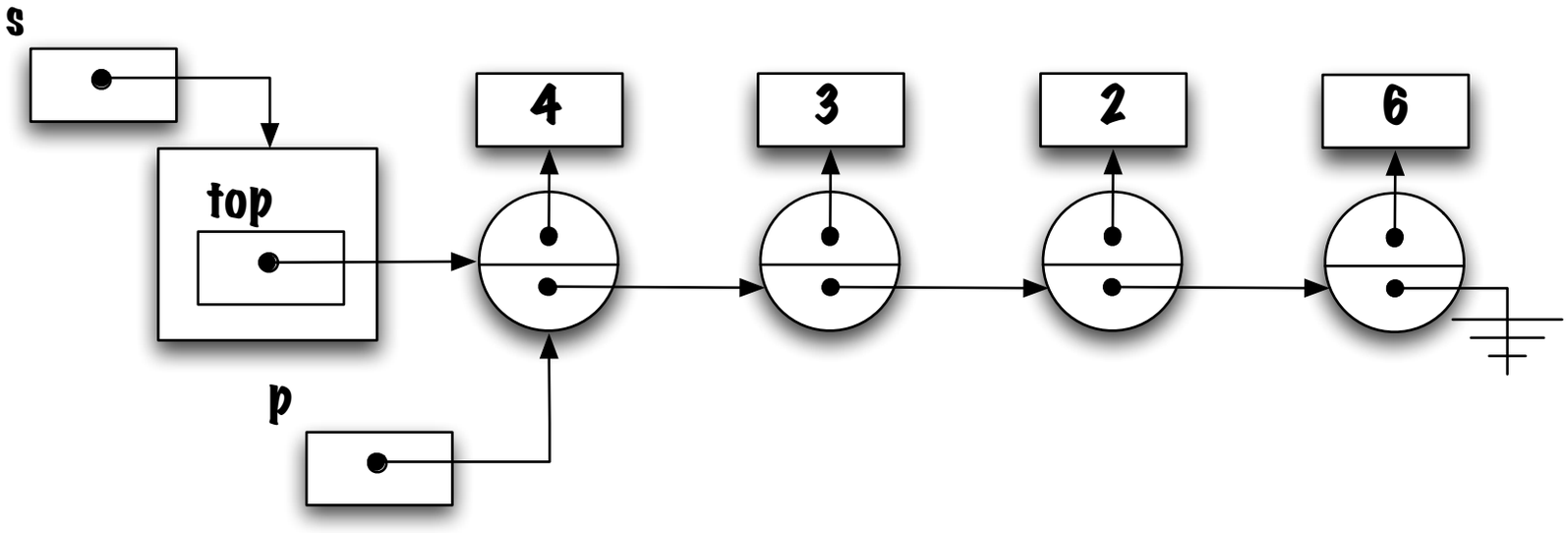
p = p.next

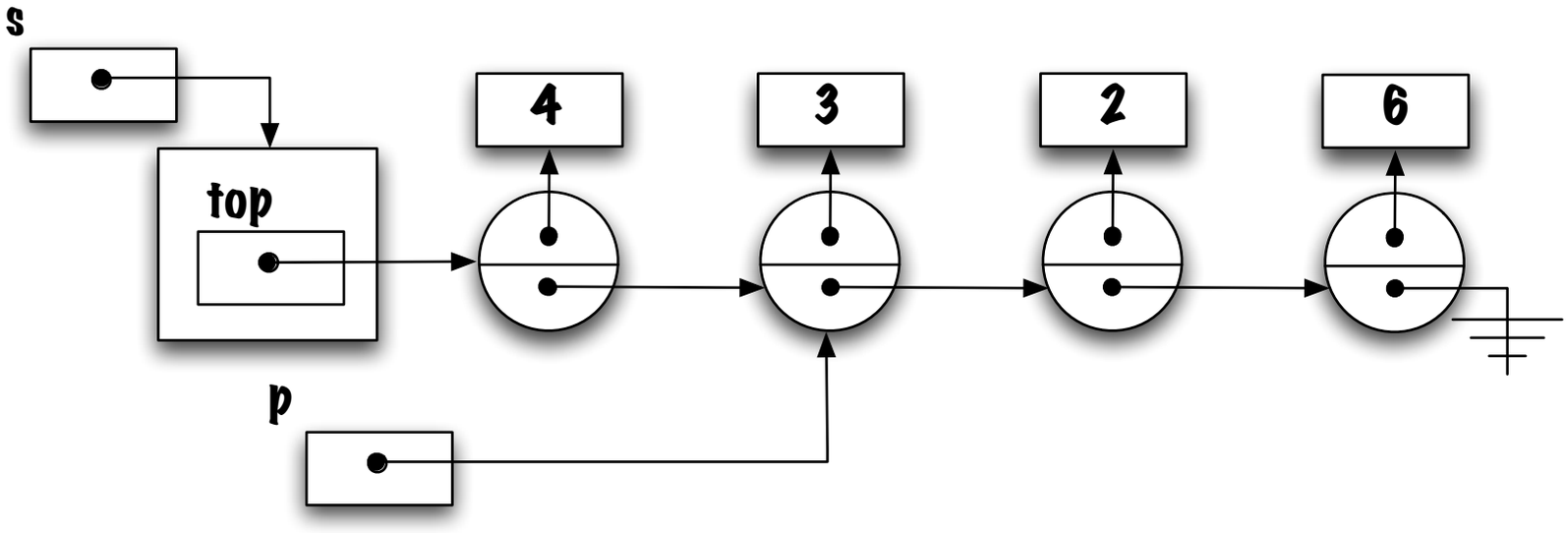
Solution

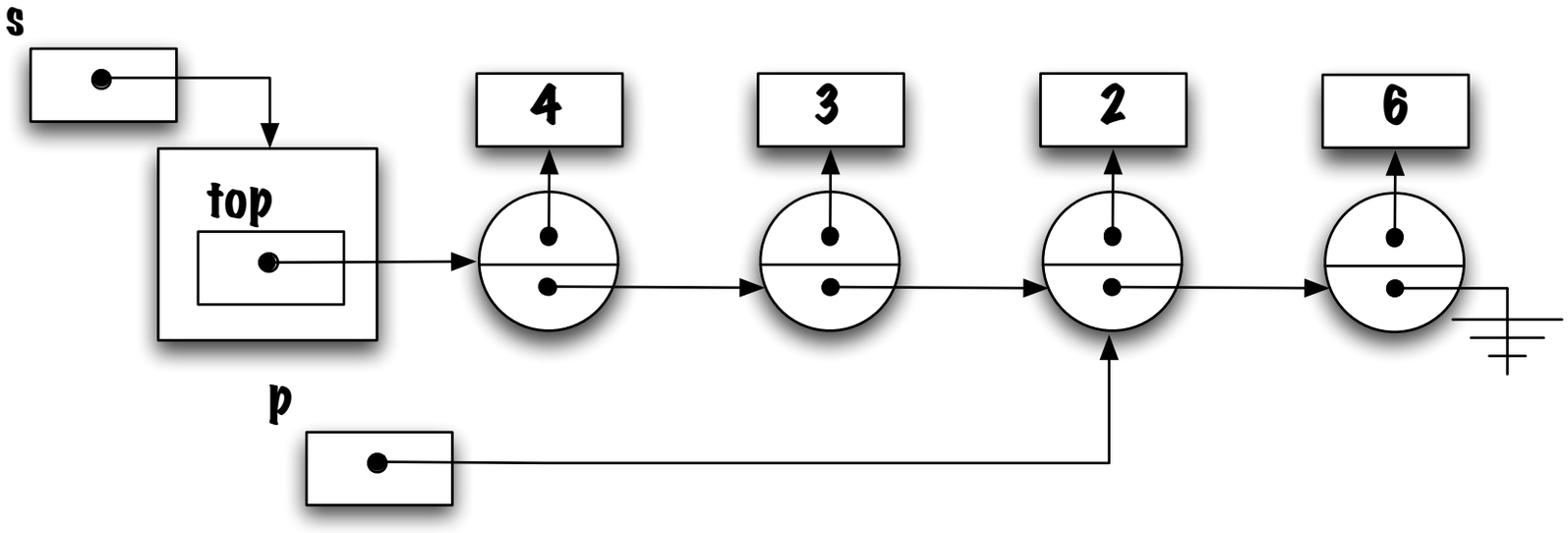
```
public String toString() {
    String res = "[";
    if ( top != null ) {
        Elem p = top;
        res = res + p.value;
        p = p.next;
        while ( p != null ) {
            res = res + "," + p.value;
            p = p.next;
        }
    }
    res = res + "]";
    return res;
}
```

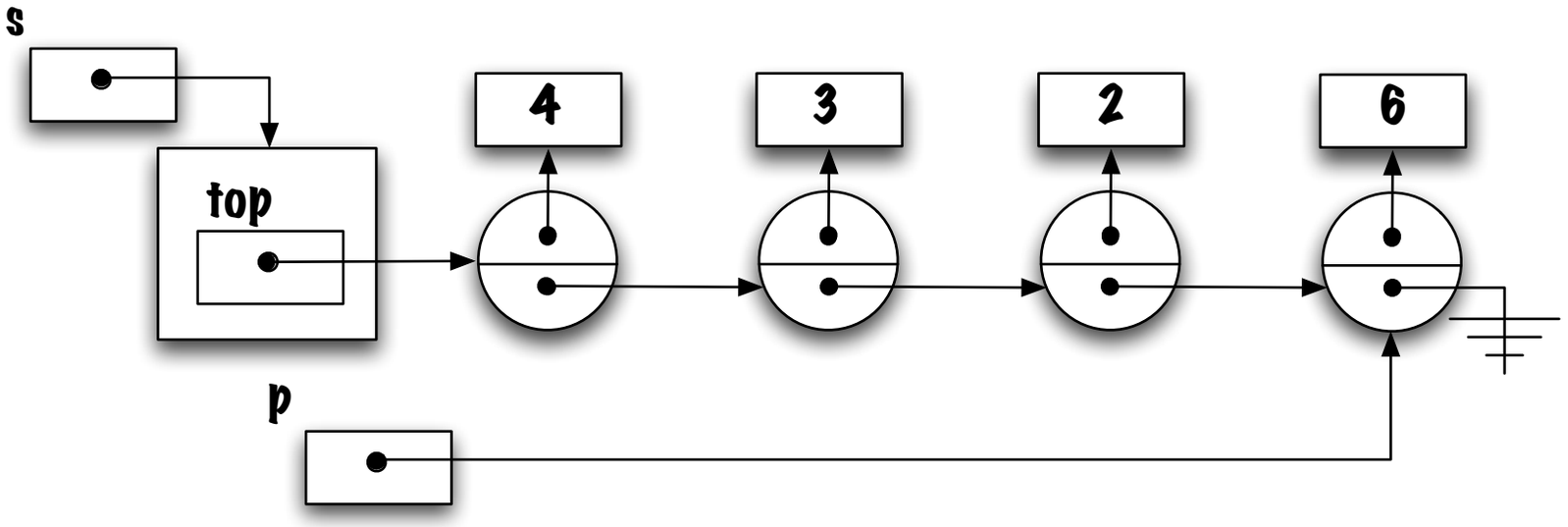

s
top
4
3
2
6

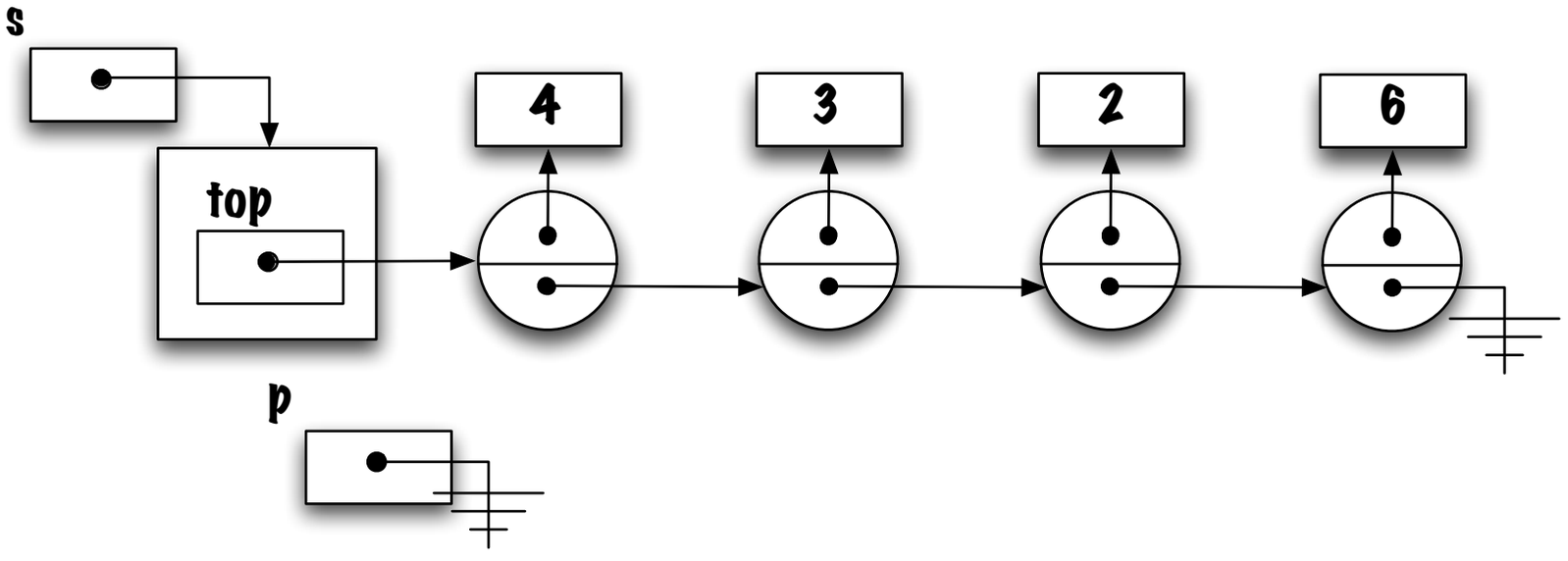

s
top
4
3
2
6
p


s
top
p
4
3
2
6


s
top
4
3
2
6
p


s
top
4
3
2
6
p

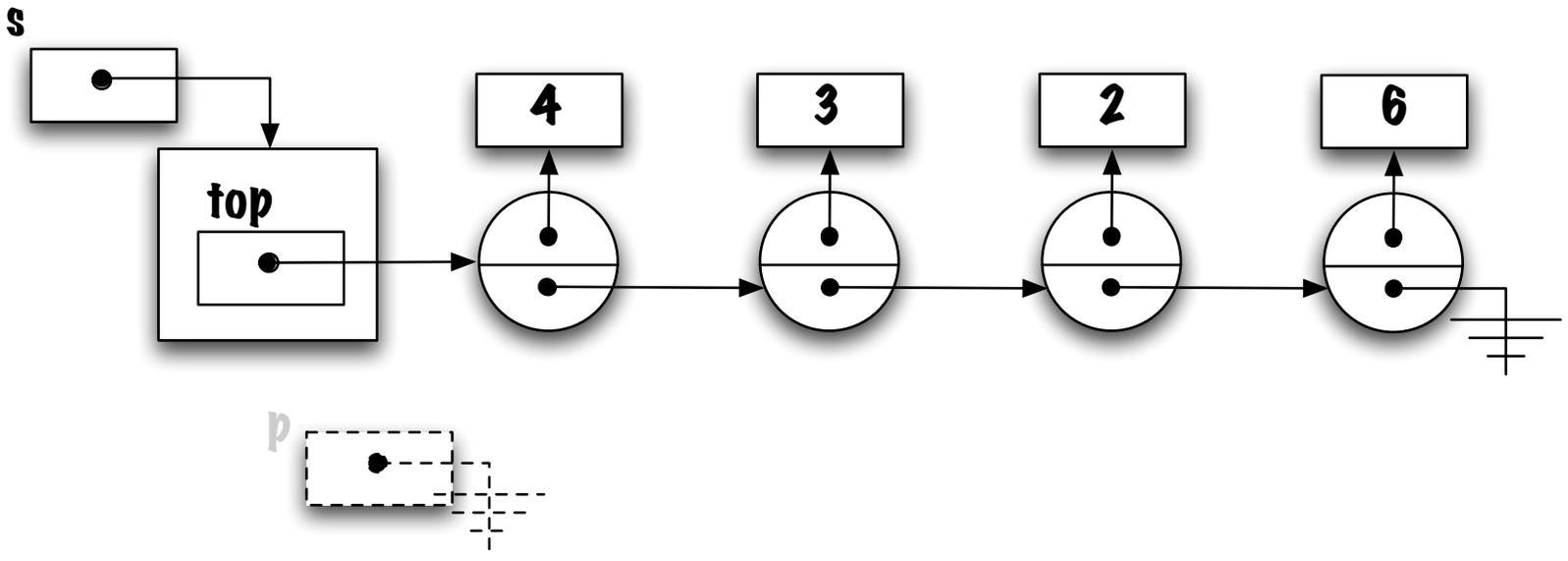

s
top
4
3
2
6
p

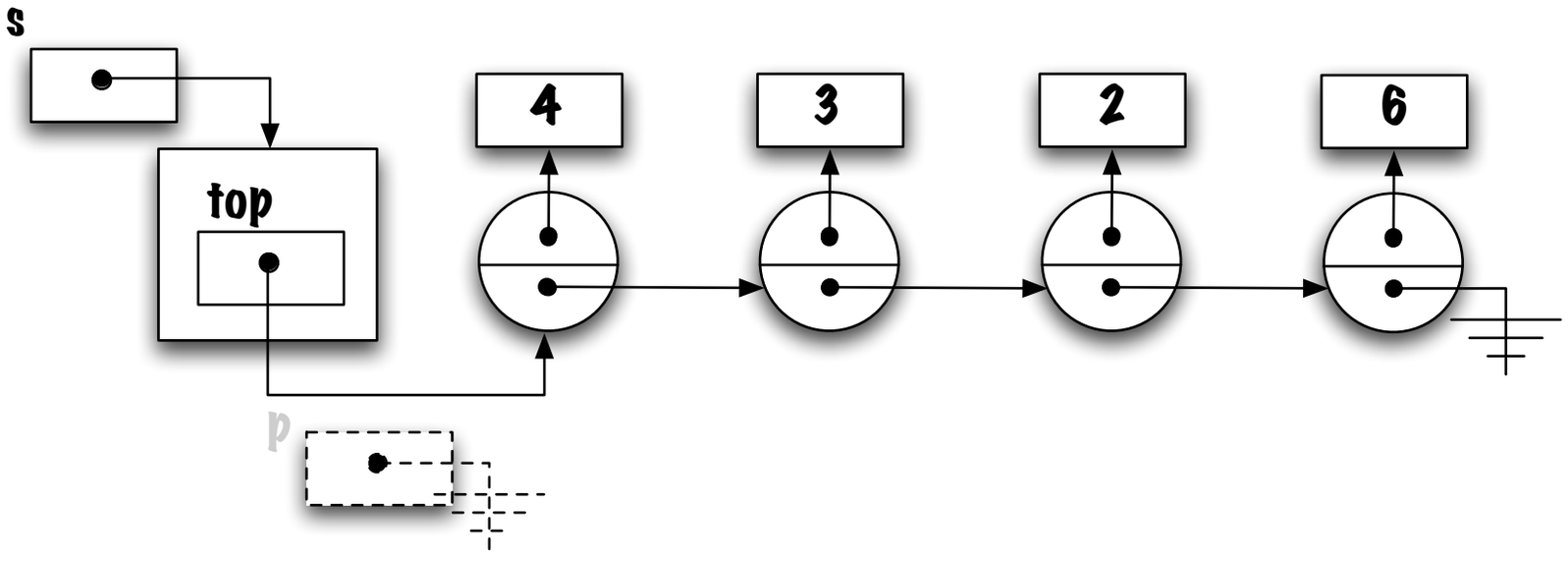

s
top
4
3
2
6
p

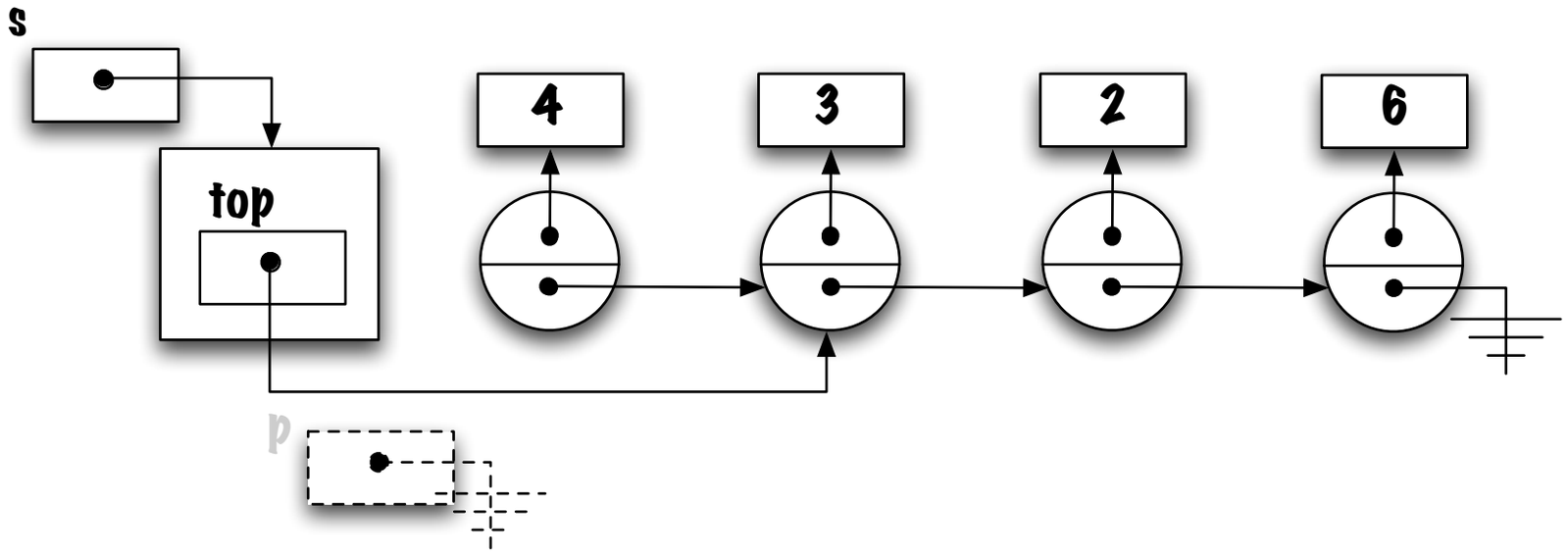
Piège

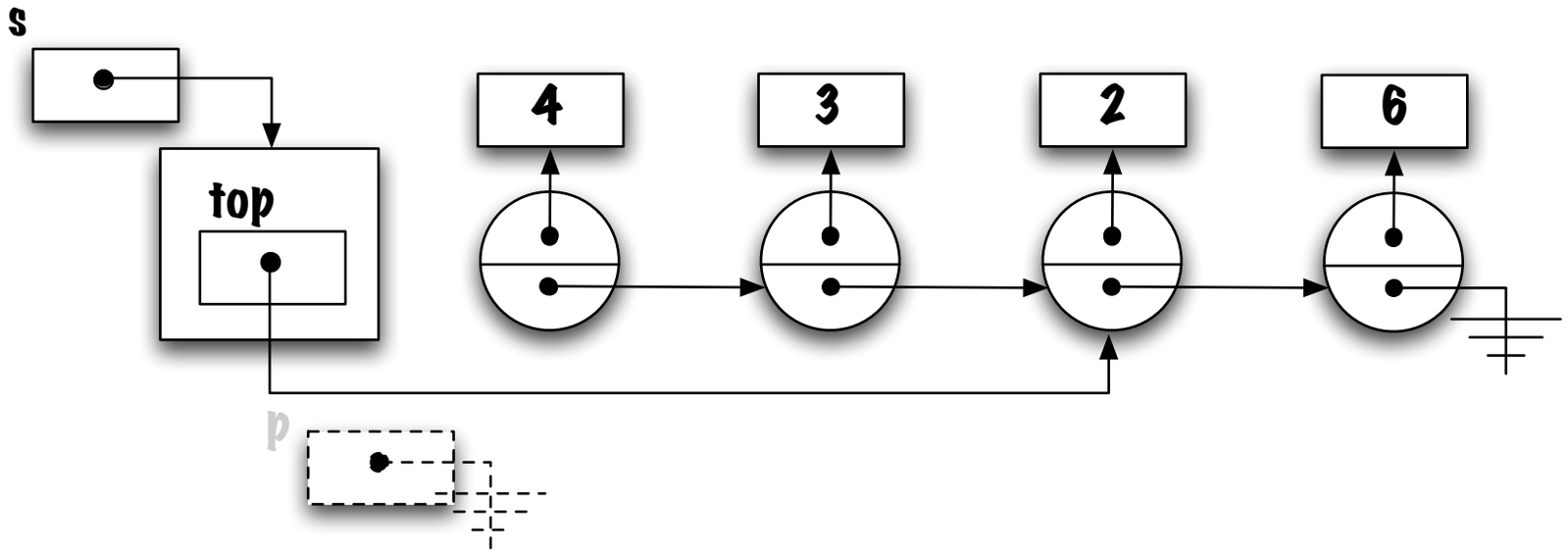
```
public String toString() {
    String res = "[";
    if ( top != null ) {
        res = res + top.value;
        top = top.next;
        while ( top != null ) {
            res = res + "," + top.value;
            top = top.next;
        }
    }
    res = res + "]";
    return res;
}
```

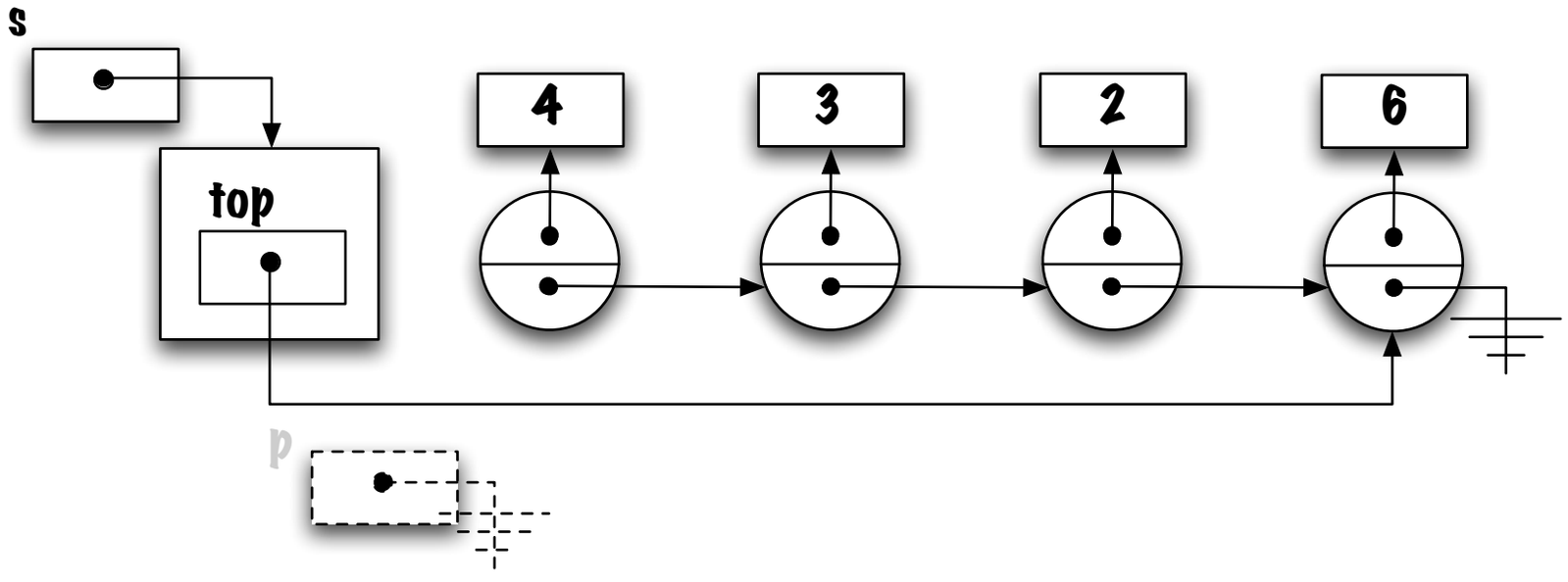
$\Rightarrow$ L'implémentation ci-haut illustre un problème récurrent. Quel est ce problème ? Quelles en sont les conséquences.

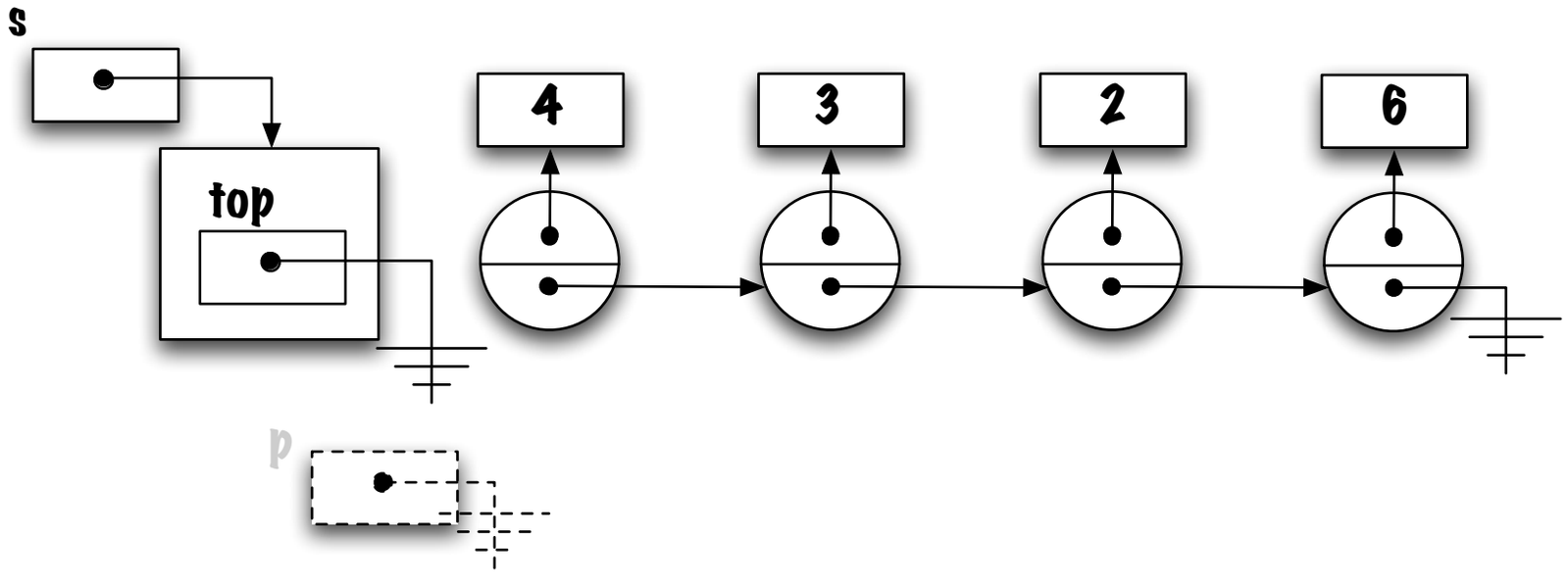

s
top
4
3
2
6
p


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

Remarques

Pour l'implémentation à l'aide d'un tableau le parcours de la structure de données est aussi facile dans un sens que dans l'autre.

Qu'en est-il de l'implémentation à l'aide d'une liste chaînée ?

Comparez le nombre d'opérations nécessaires afin d'accéder à un élément par position pour l'une ou l'autre des implémentations.

Résumé

Les listes chaînées constituent une alternative aux tableaux afin de sauvegarder une ≪collection d'éléments≫.

Ces structures utilisent toujours une quantité de mémoire qui est proportionnelle au nombre d'éléments ; c'est parce que chaque élément a son propre conteneur, appelé noeud (Elem), et que chaque conteneur est lié à son suivant à l'aide d'une référence.

Pour l'instant, nous nous limiterons aux structures linéaires, mais les mêmes principes s'applique pour créer des structures de graphes, arborescences et autres — mais ça, c'est une autre histoire.