Mise en œuvre des listes

Pratique de la programmation orientée-objet

Michel Schinz – 2014-03-17

Mise en œuvre des listes

Nous allons examiner deux mises en œuvre simplifiées des listes, similaires à celles offertes par la bibliothèque Java, à savoir :

1. les tableaux-listes, mis en œuvre par la classe `ArrayList`,
2. les listes chaînées, mises en œuvre par la classe `LinkedList`.

Interface `List`

L'interface `List` implémentée par nos mises en œuvre est une simplification de celle de l'API Java. Elle contient toutefois les méthodes les plus importantes.

```
public interface List<E> {
  boolean isEmpty();
  int size();
  void add(E newElem);
  void remove(int i);
  E get(int i);
  void set(int i, E newElem);
  Iterator<E> iterator();
}
```

Interface `Iterator`

L'interface `Iterator` implémentée par nos mises en œuvre des itérateurs est, quant à elle, identique à celle de l'API Java :

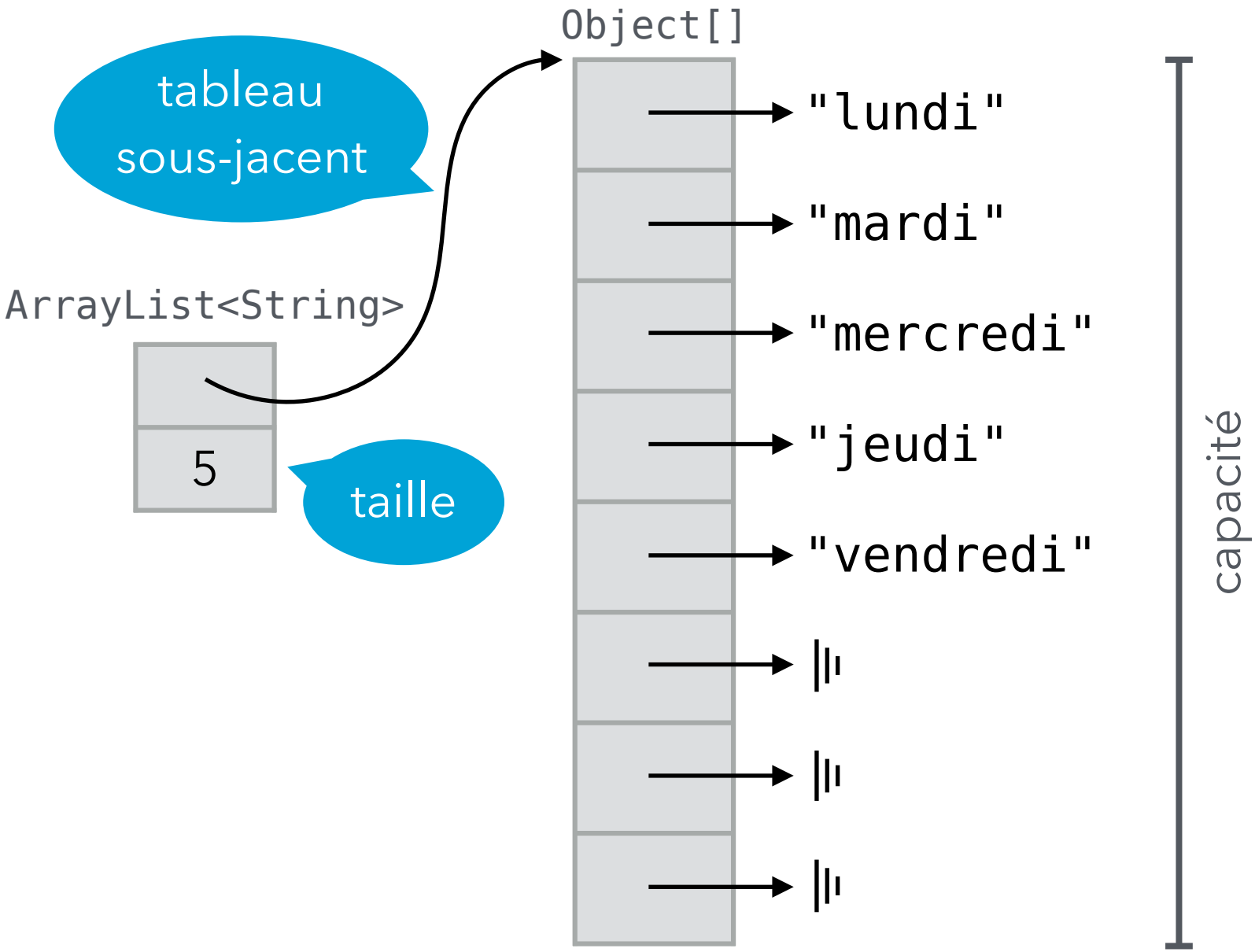
```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
}
```

Tableau-liste

Tableau-liste

Un tableau-liste est une liste dont les éléments sont stockés dans un tableau, que nous appellerons **tableau sous-jacent**. La **capacité** d'un tableau-liste est la taille de son tableau sous-jacent. Il s'agit du nombre d'éléments que le tableau-liste peut stocker sans qu'il soit nécessaire de redimensionner (par copie) son tableau sous-jacent.

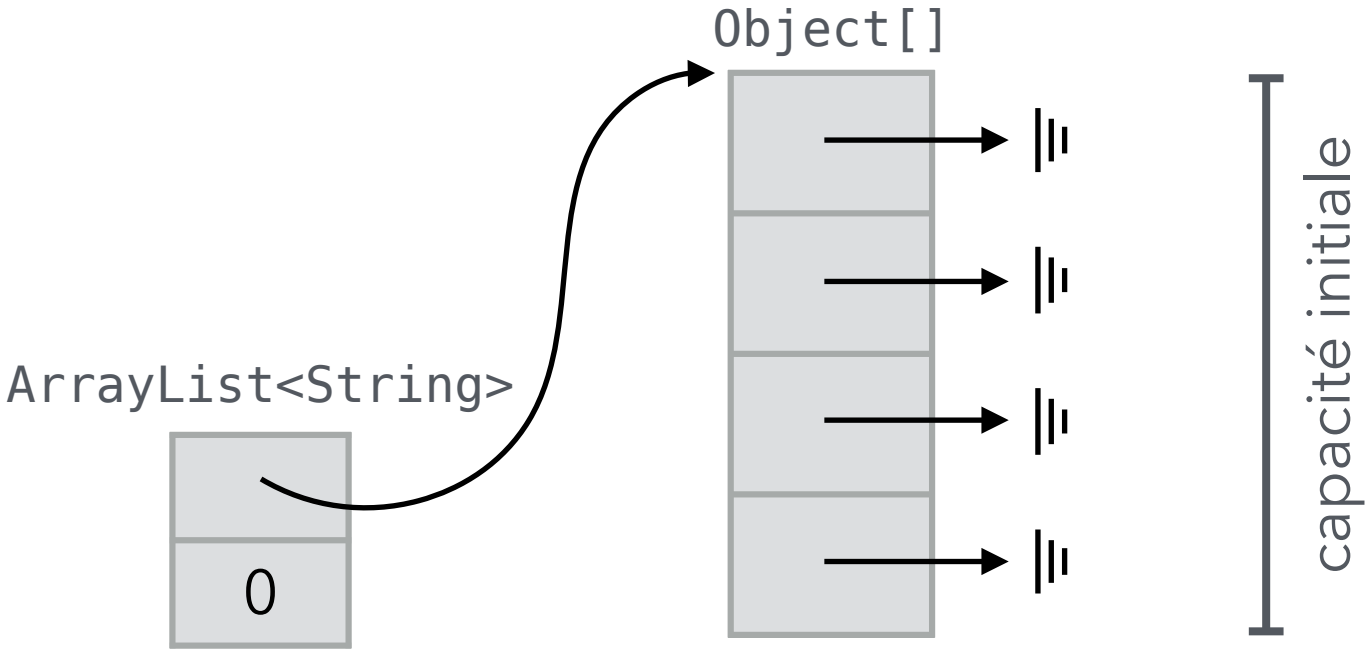
Tableau-liste



Création

A sa création, un tableau-liste est équipé d'un tableau sous-jacent de petite taille, la **capacité initiale**, non nulle.
Il peut être judicieux d'offrir la possibilité de spécifier la capacité initiale lors de la création.

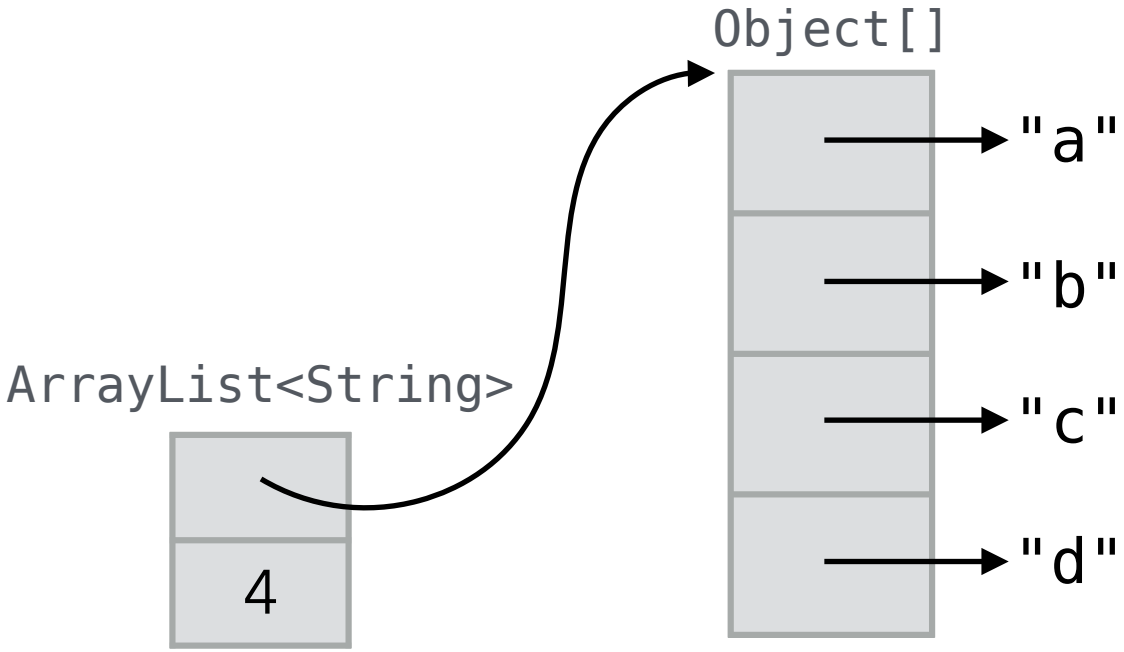
Création



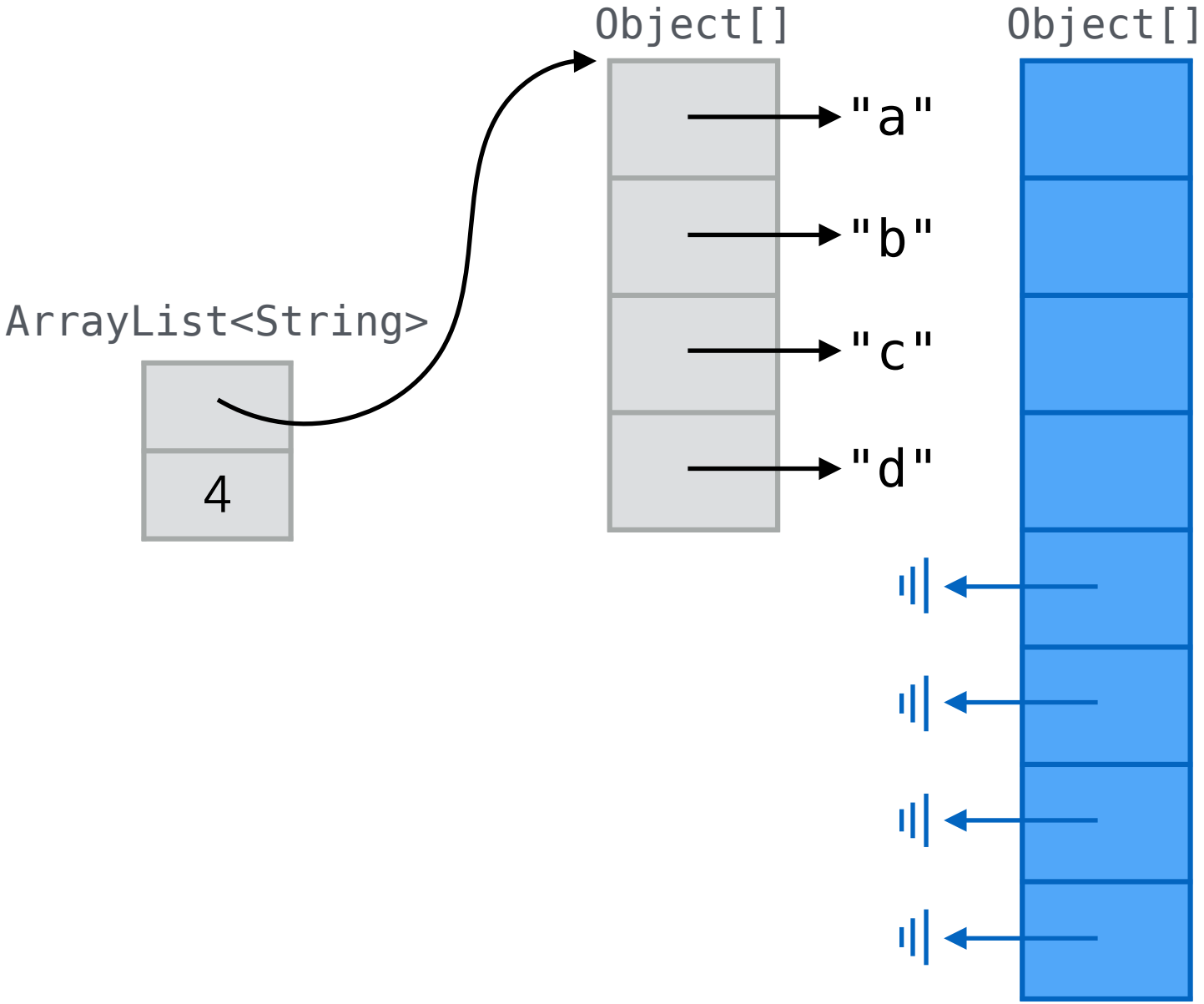
Redimensionnement

Lorsque toute la capacité du tableau sous-jacent a été utilisée et que de nouveaux éléments doivent être ajoutés au tableau-liste, il est nécessaire de « redimensionner » le tableau sous-jacent.
Cela se fait par création d'un nouveau tableau sous-jacent, dans lequel les éléments de l'ancien sont copiés.
La taille du nouveau tableau sous-jacent est un multiple de celle de l'ancien, p.ex. le double, afin de pouvoir amortir le coût de la copie sur plusieurs opérations d'insertion.

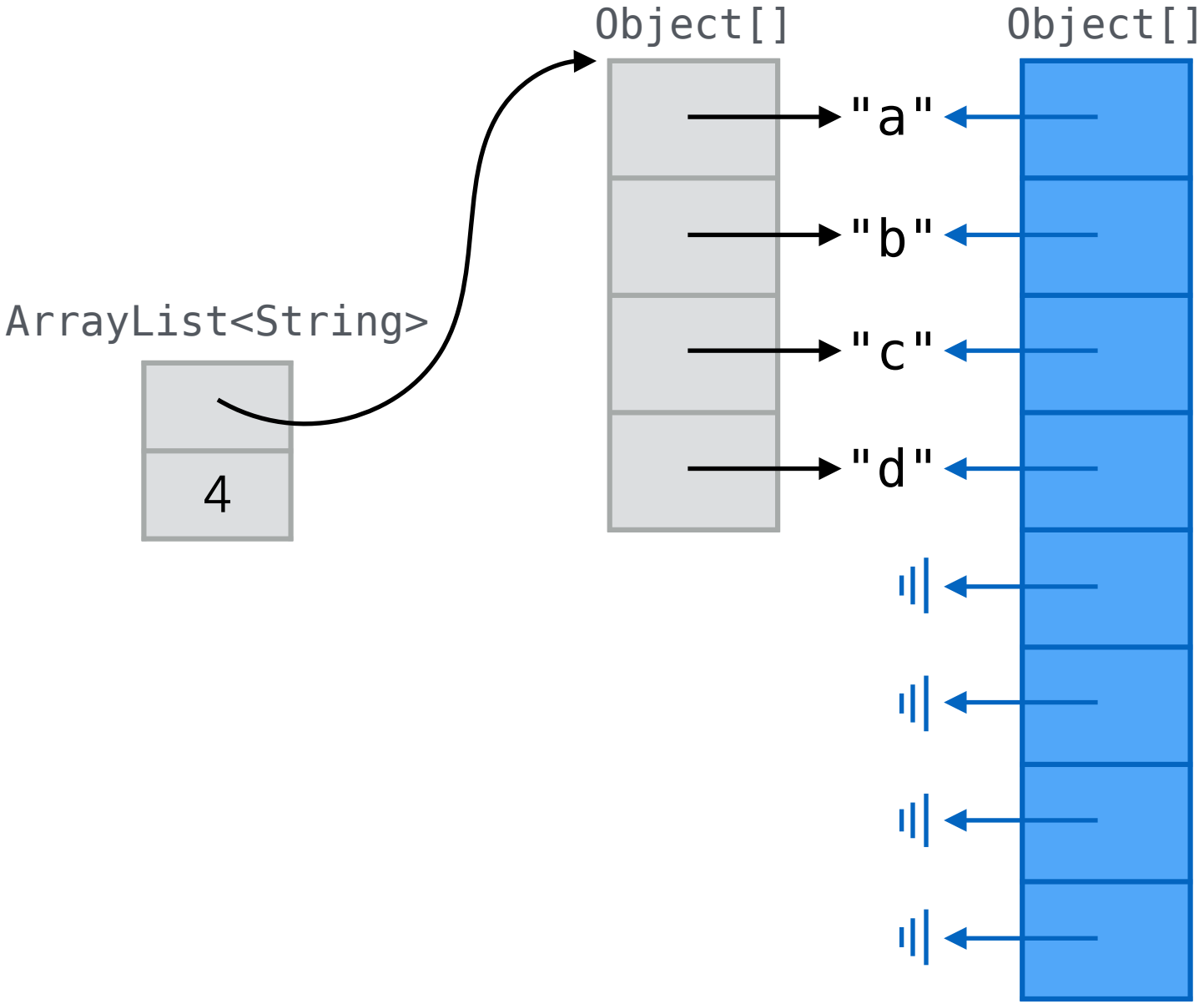
Redimensionnement



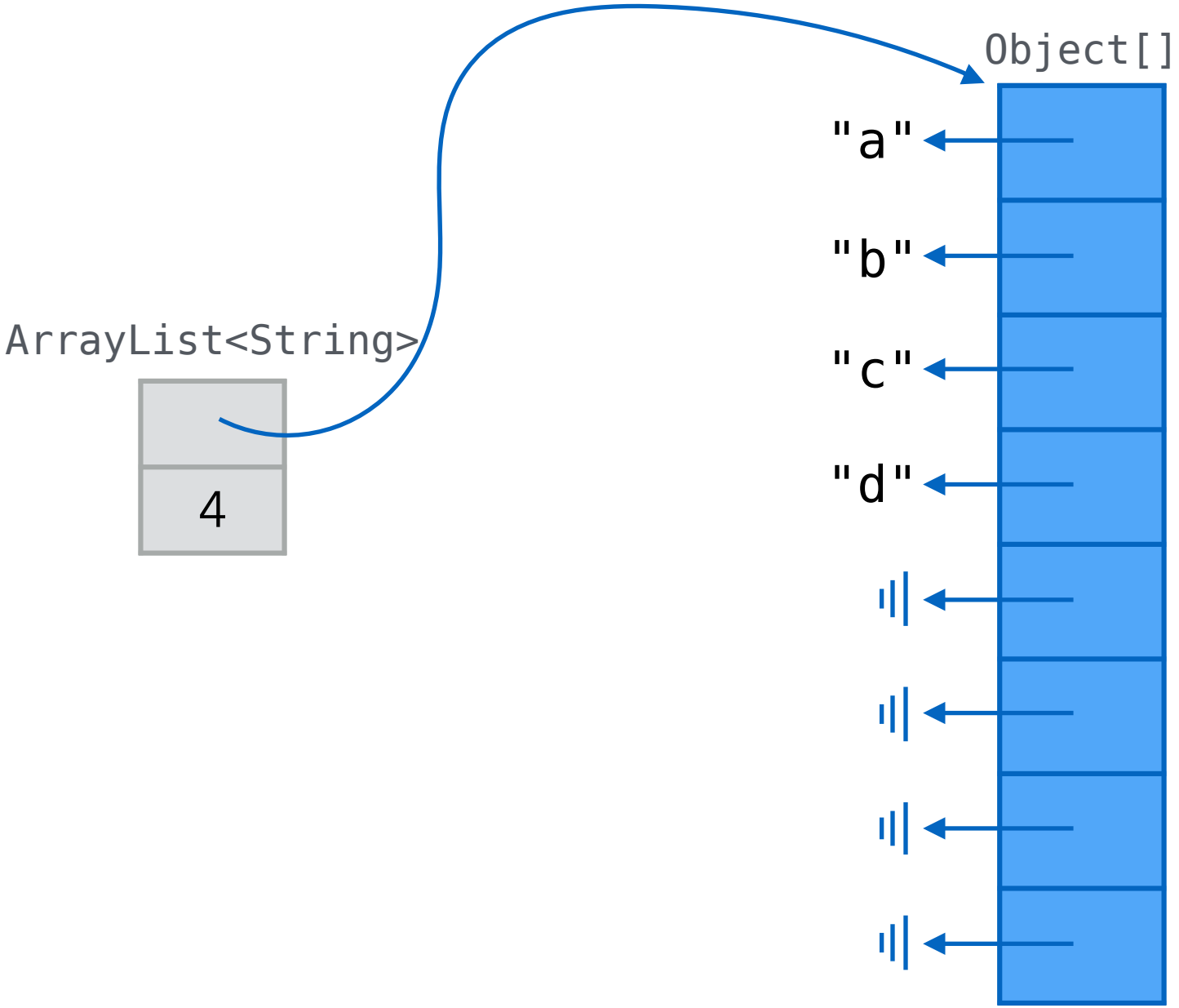
Redimensionnement



Redimensionnement



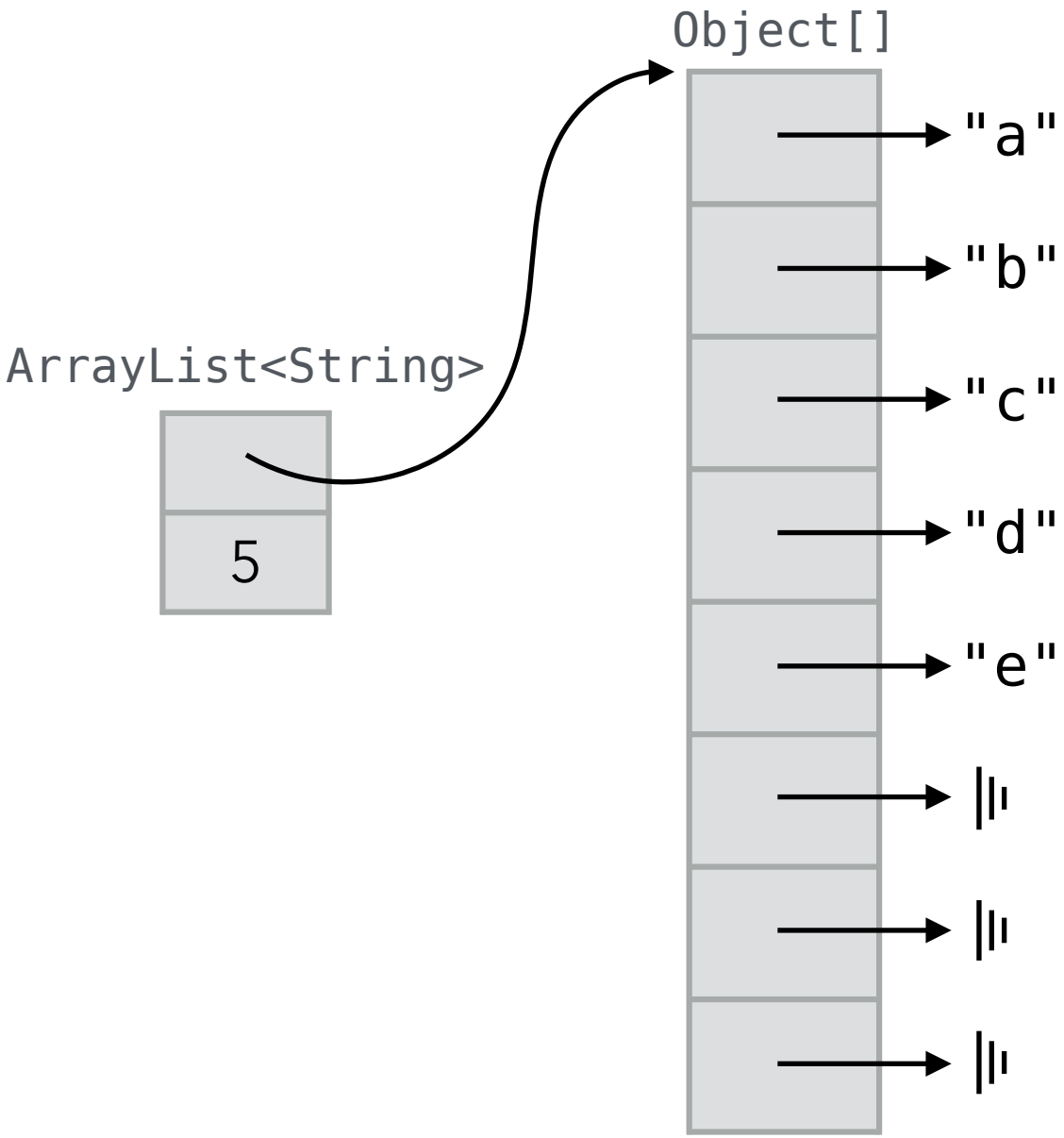
Redimensionnement



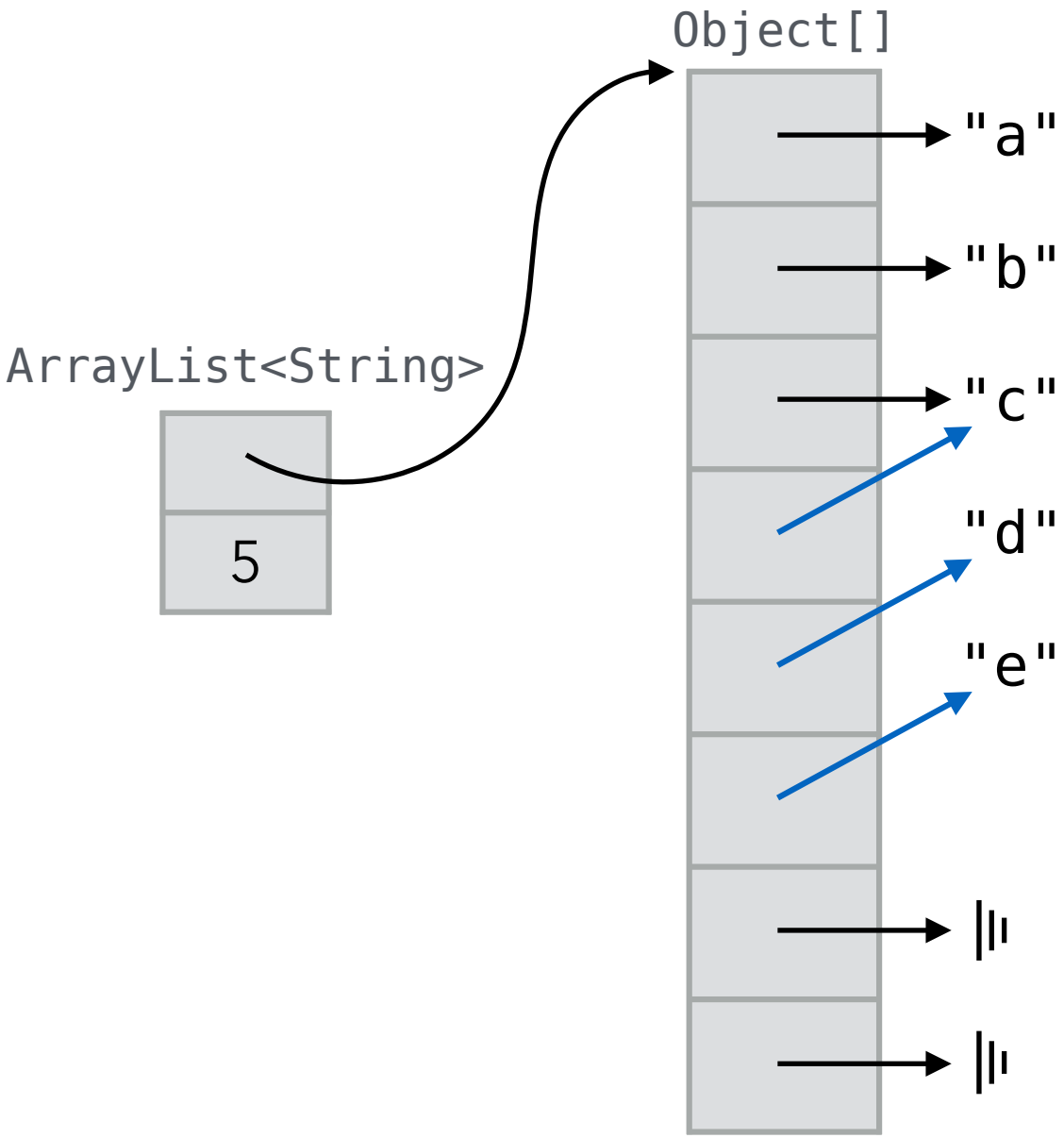
Insertion

L'insertion d'un élément dans un tableau-liste peut demander un redimensionnement préalable si la capacité du tableau sous-jacent est totalement utilisée.
Une fois l'éventuel redimensionnement fait, il faut faire une place pour le nouvel élément en décalant vers le haut tous ceux d'indice supérieur à l'indice d'insertion, puis mettre le nouvel élément à sa place.

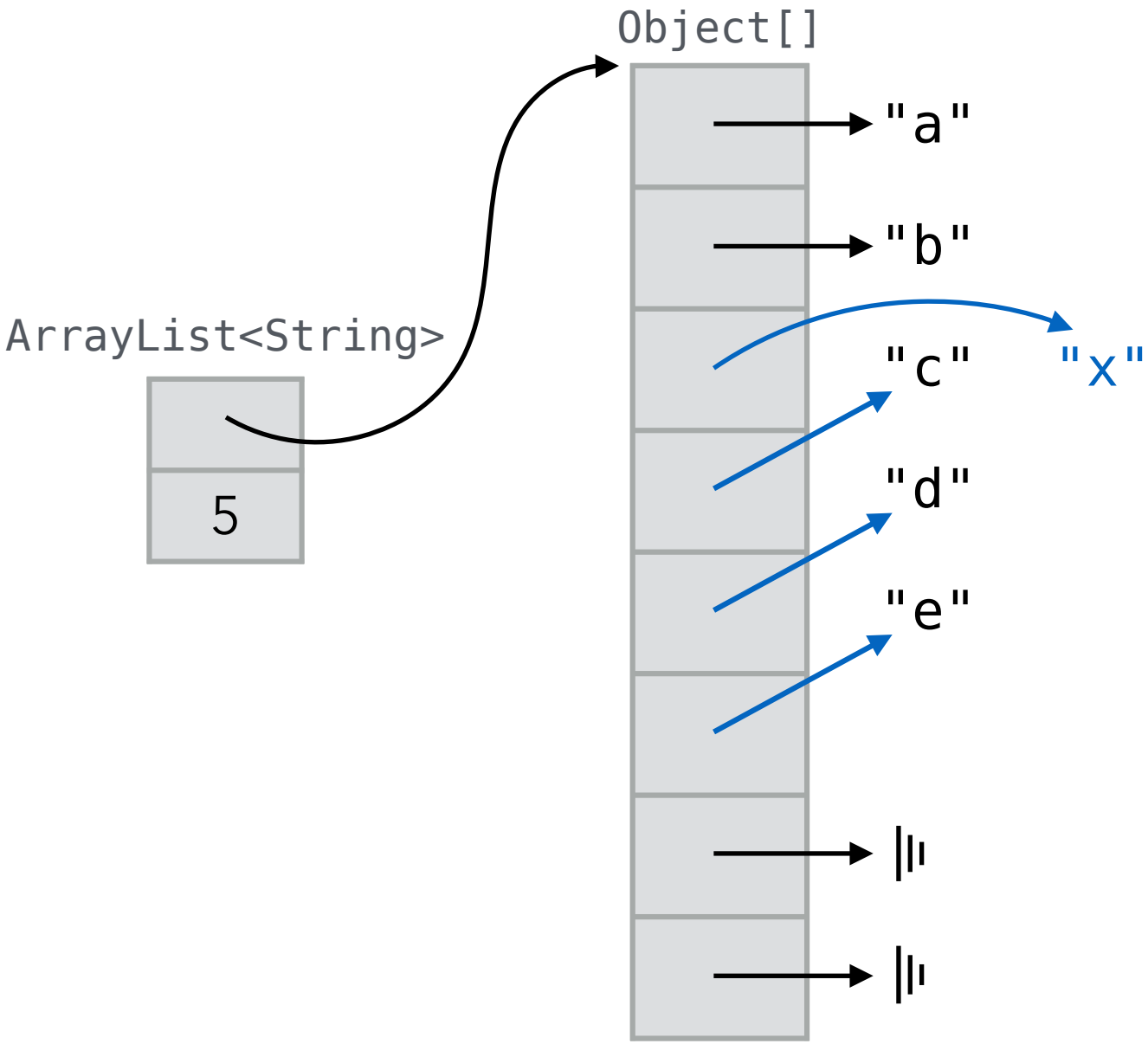
Insertion



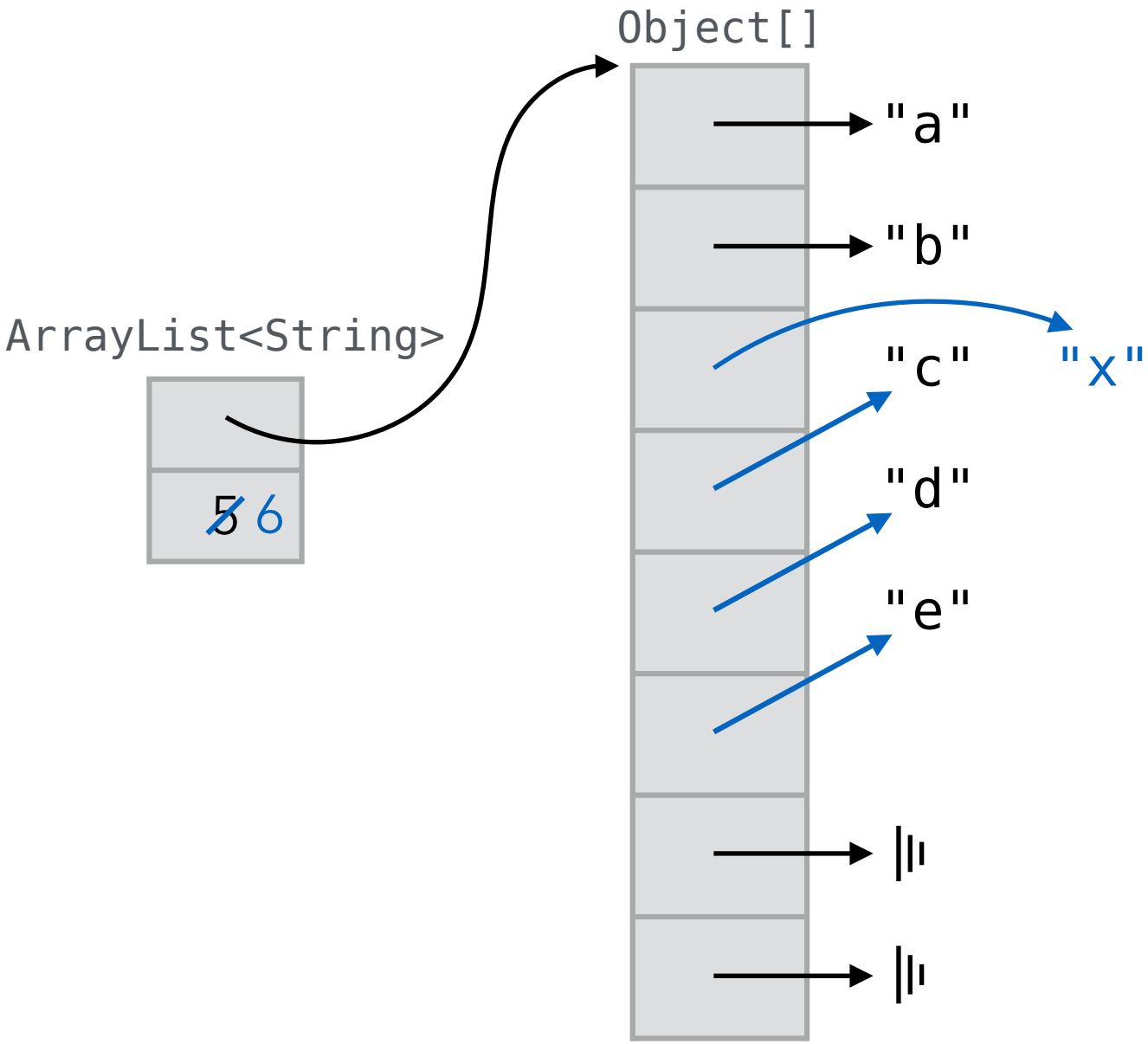
Insertion



Insertion



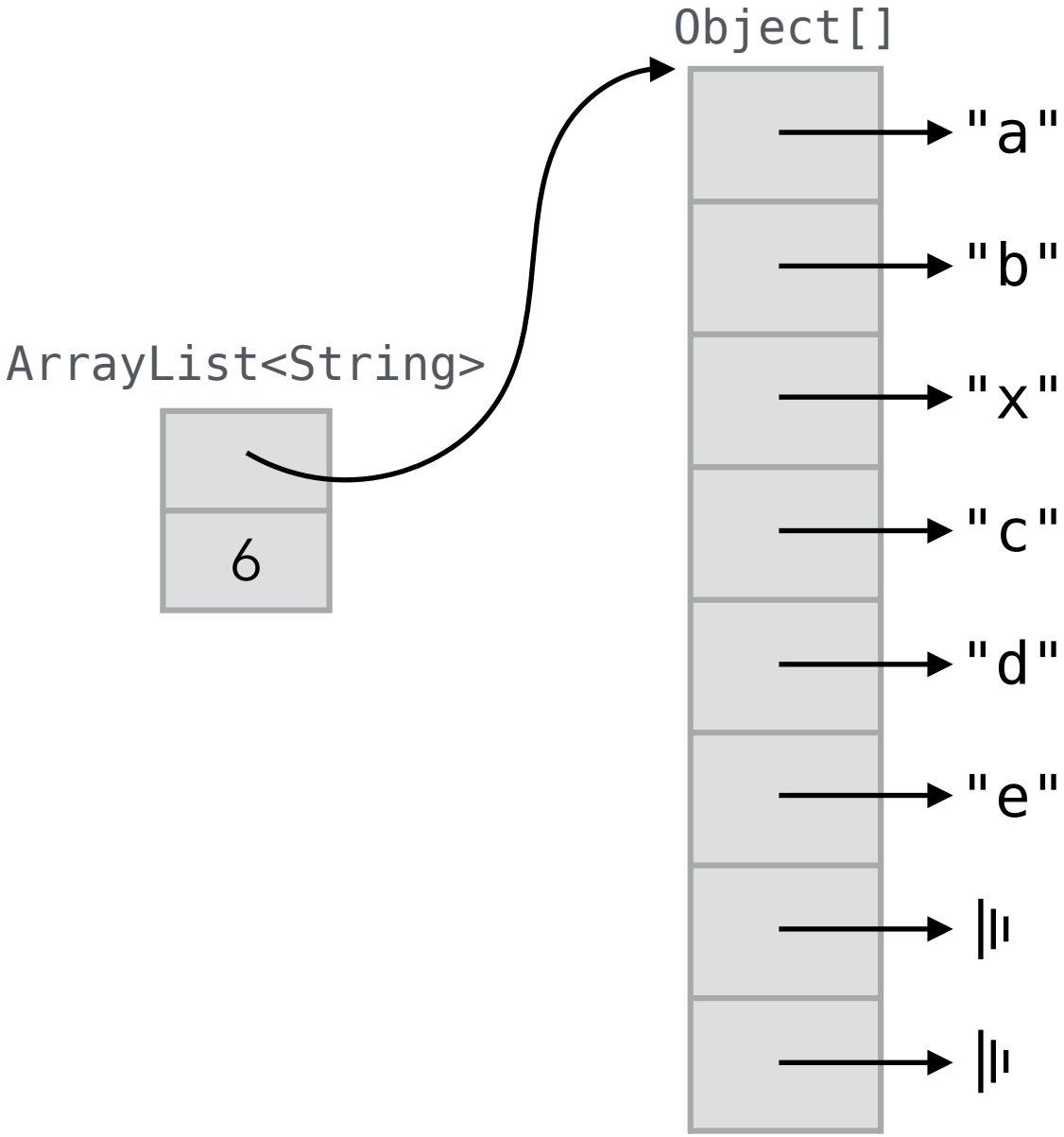
Insertion



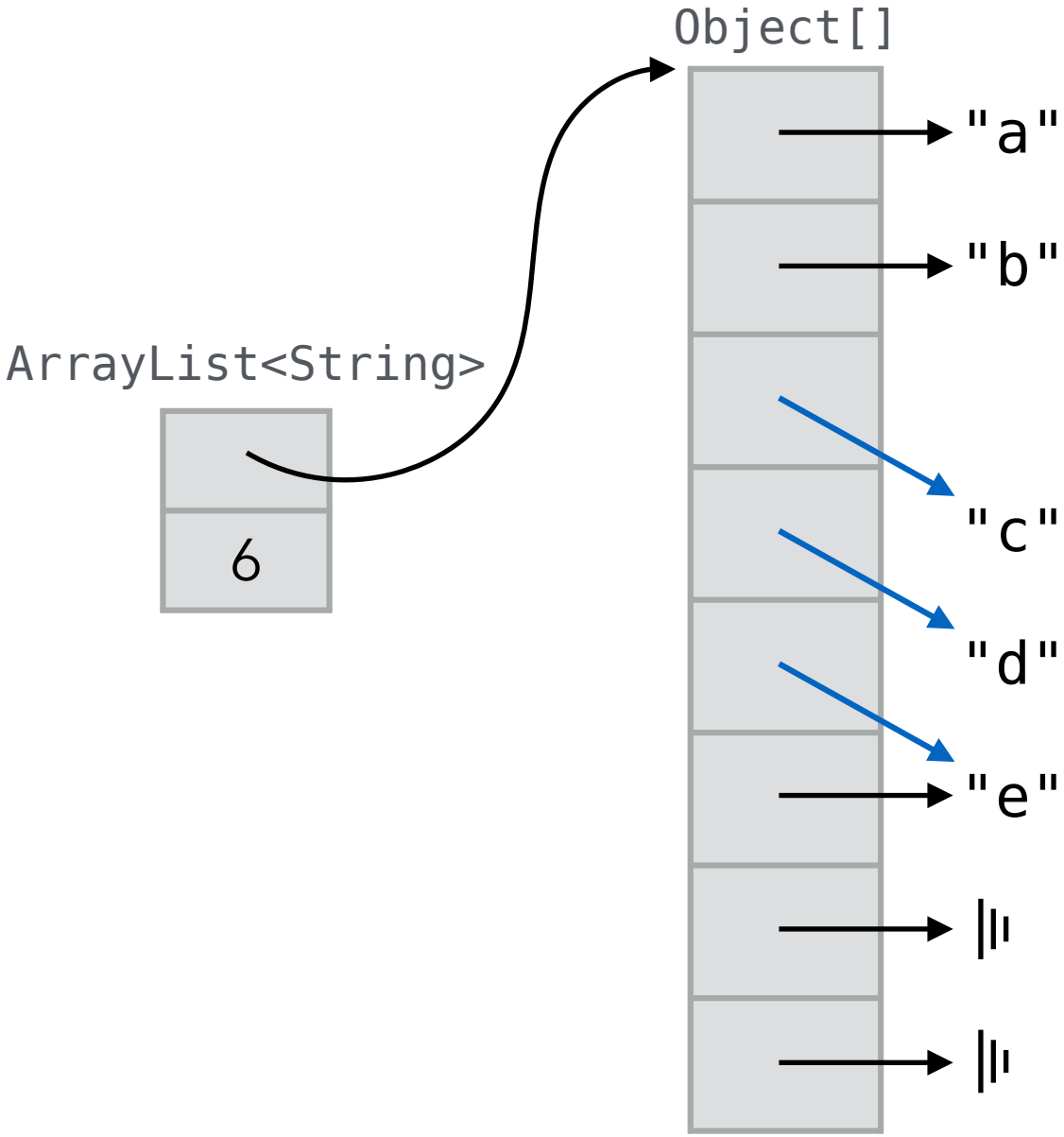
Suppression

La suppression d'un élément d'un tableau-liste se fait par décalage vers le bas de tous les éléments d'indice supérieur à l'élément à supprimer.
Cela fait, il faut effacer l'élément désormais inutilisé du tableau sous-jacent, pour éviter de conserver une référence inutilisée à un objet.

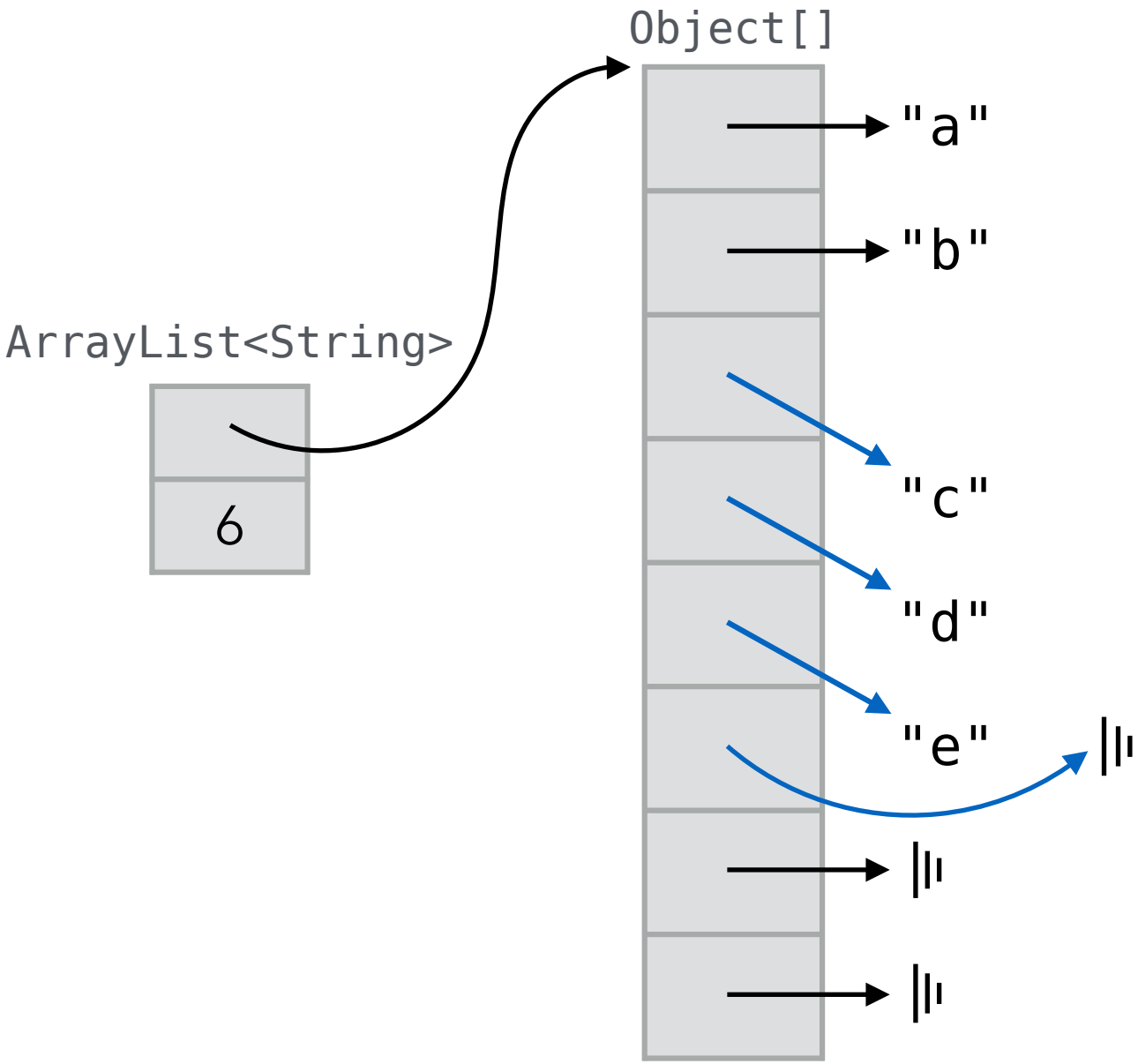
Suppression



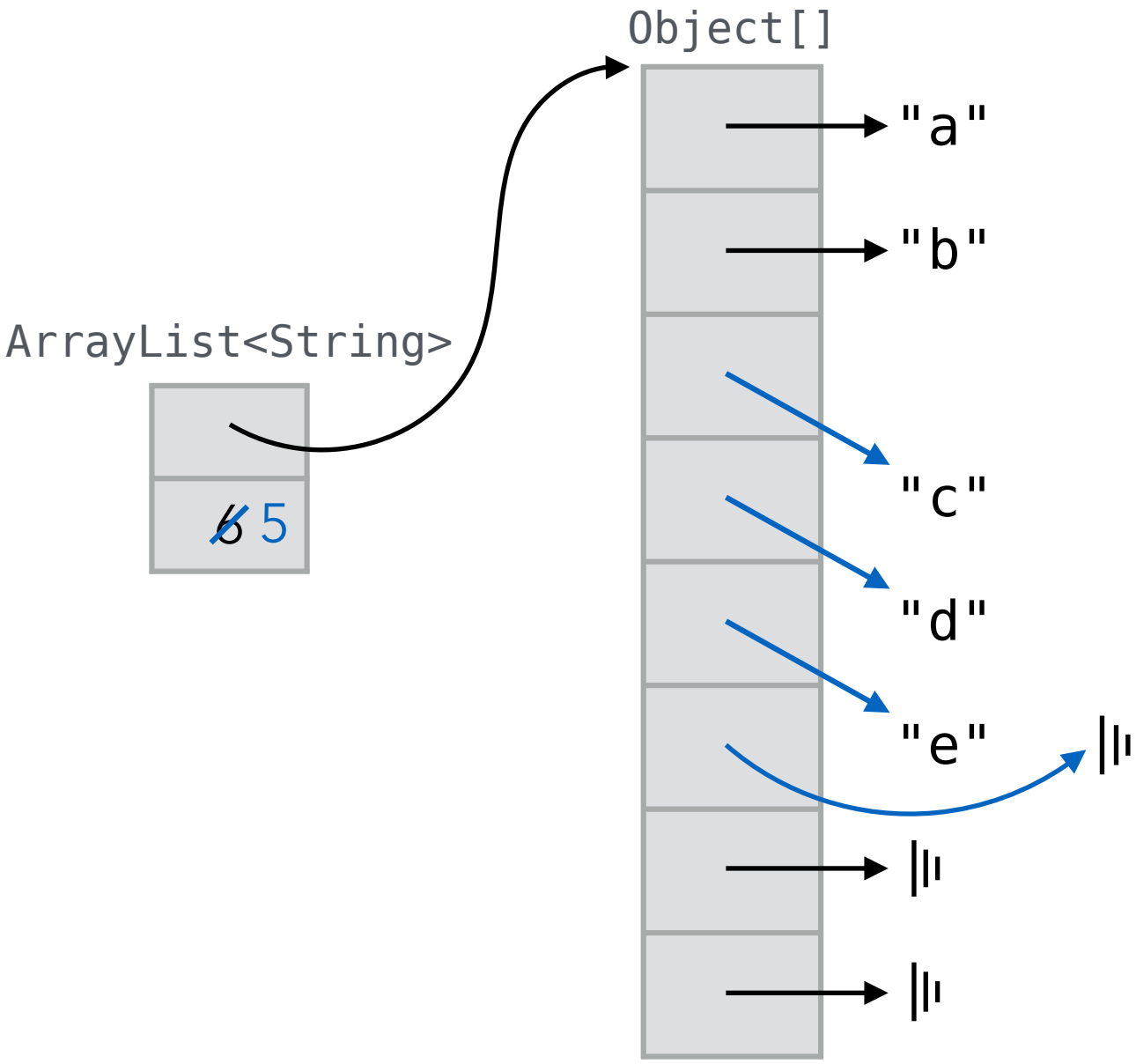
Suppression



Suppression



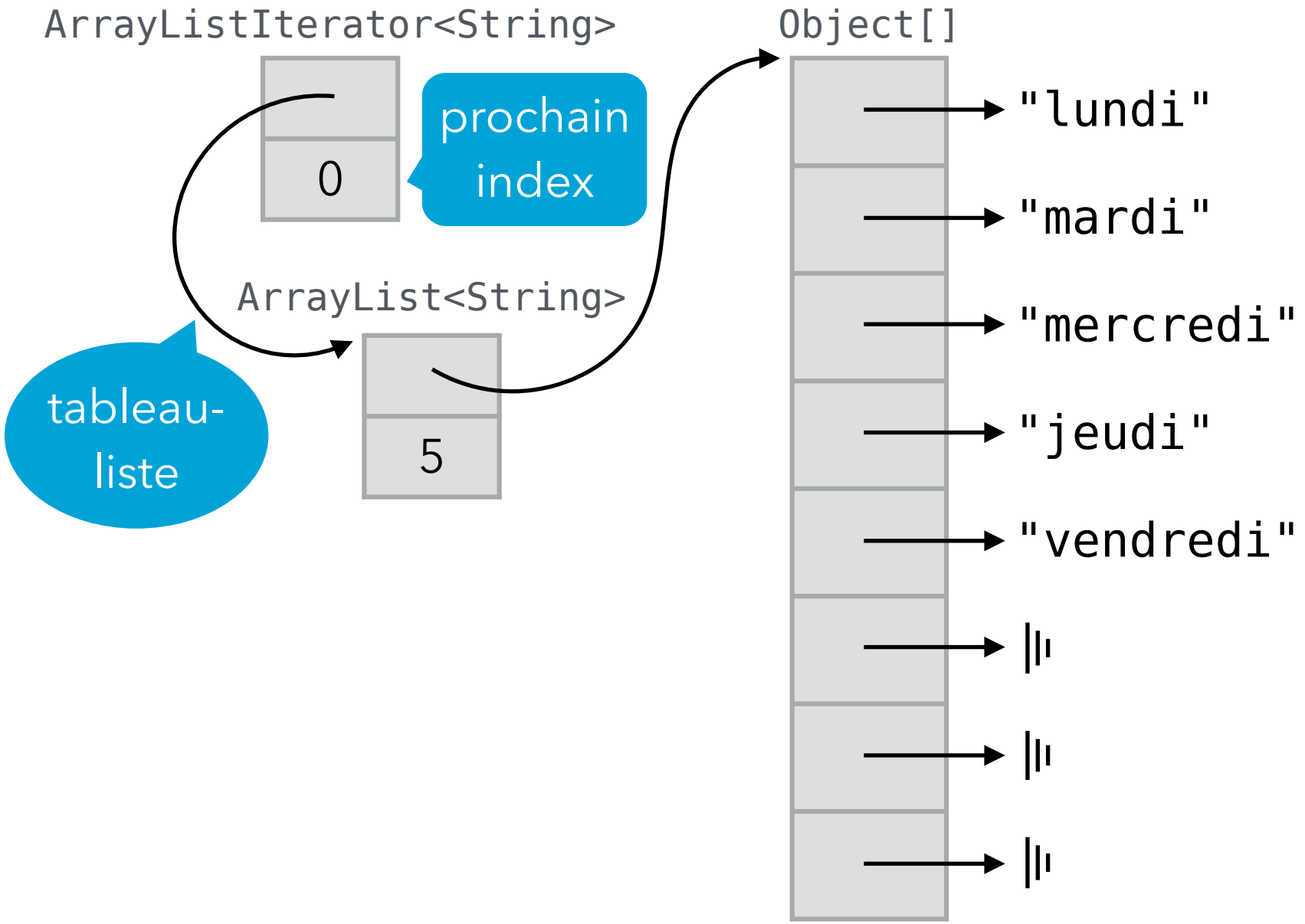
Suppression



Itération

Un tableau-liste est à accès aléatoire, c-à-d qu'il est possible d'obtenir un élément étant donné son index en O(1).
Cette caractéristique rend la mise en œuvre d'un itérateur sur un tableau-liste très simple, puisqu'il suffit de stocker une référence vers le tableau-liste et l'index du prochain élément.
La méthode `next` de l'itérateur utilise la méthode `get` du tableau-liste pour obtenir le prochain élément, puis incrémente l'index.

Itération


ArrayListIterator<String>
0
prochain index
ArrayList<String>
5
tableau-liste
Object[]
"lundi"
"mardi"
"mercredi"
"jeudi"
"vendredi"

Exercice (en classe)

Ecrivez la classe `ArrayList`.

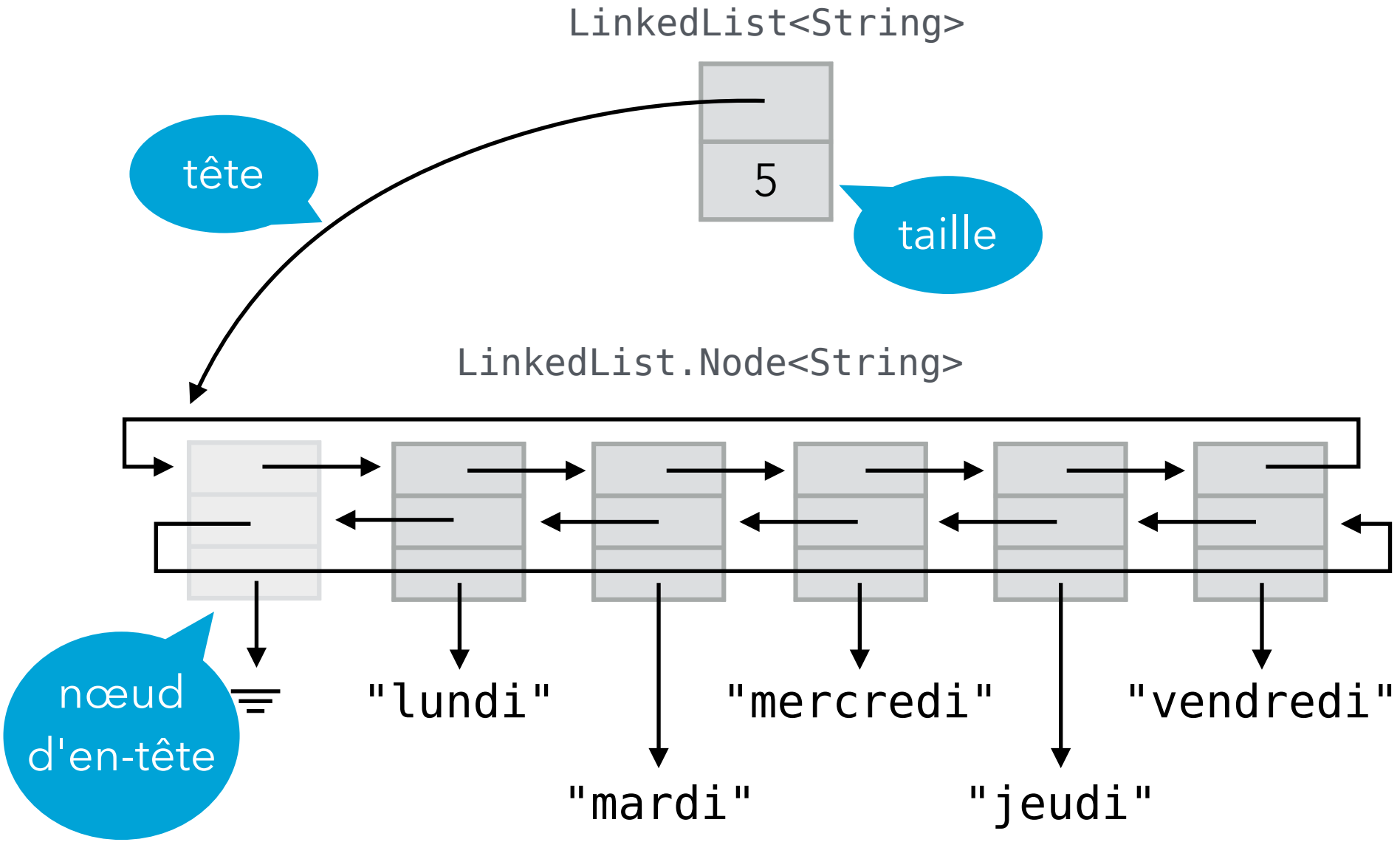
Liste chaînée

Liste chaînée

Une **liste chaînée** stocke ses éléments dans des **nœuds** (un par élément) qui sont chaînés entre-eux.
Une liste est **simplement chaînée** si chaque nœud ne possède une référence que vers l'un de ses voisins, et **doublement chaînée** si chaque nœud possède une référence vers ses deux voisins. Elle est **circulaire** si le premier et le dernier nœud sont chaînés entre-eux.
Pour faciliter la programmation, une liste chaînée peut être équipée d'un **nœud d'en-tête** qui ne stocke aucun élément mais garantit la présence d'un nœud dans la liste.

Liste chaînée

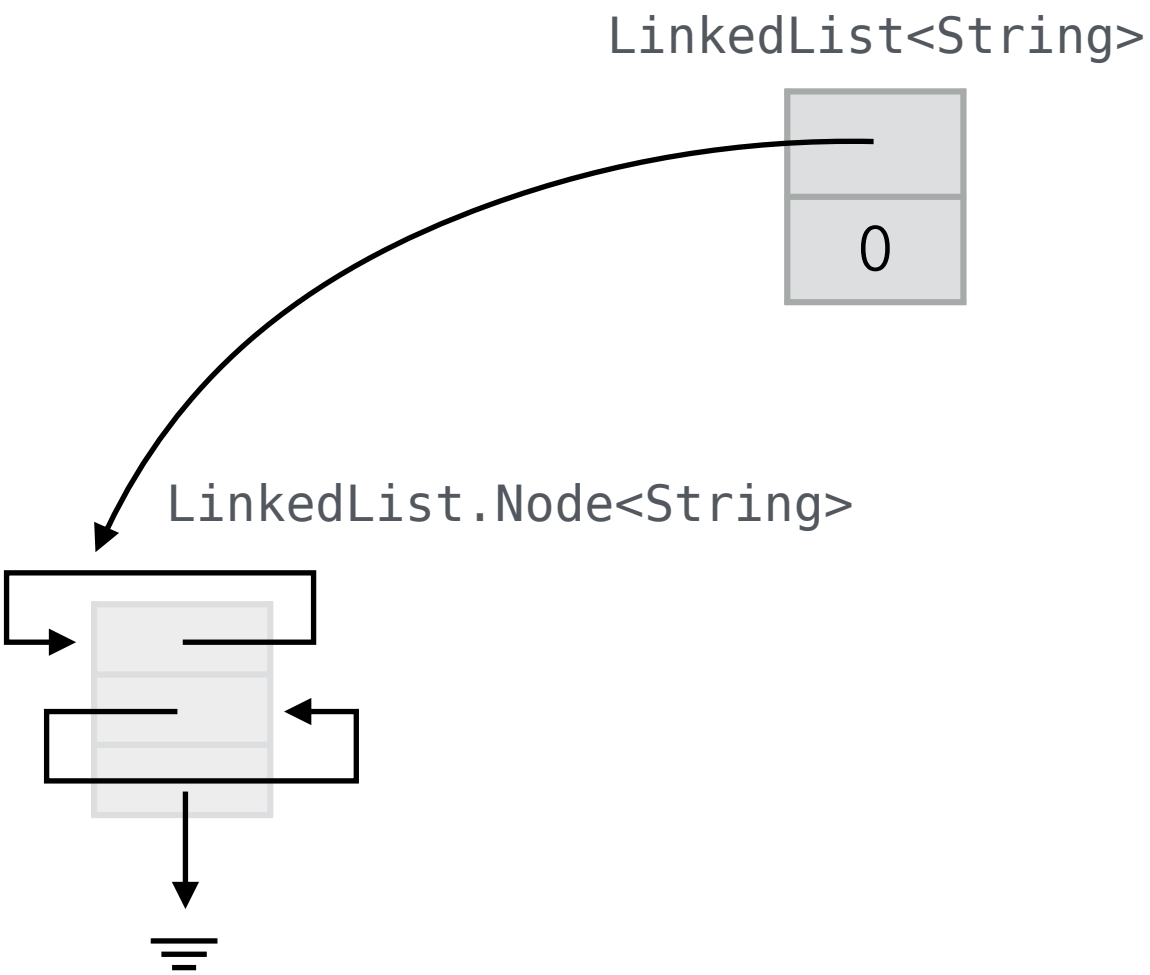
(Liste doublement chaînée circulaire avec nœud d'en-tête)



Création

A sa création, une liste chaînée (doublement et avec nœud d'en-tête) ne possède aucun autre nœud que celui d'en-tête.

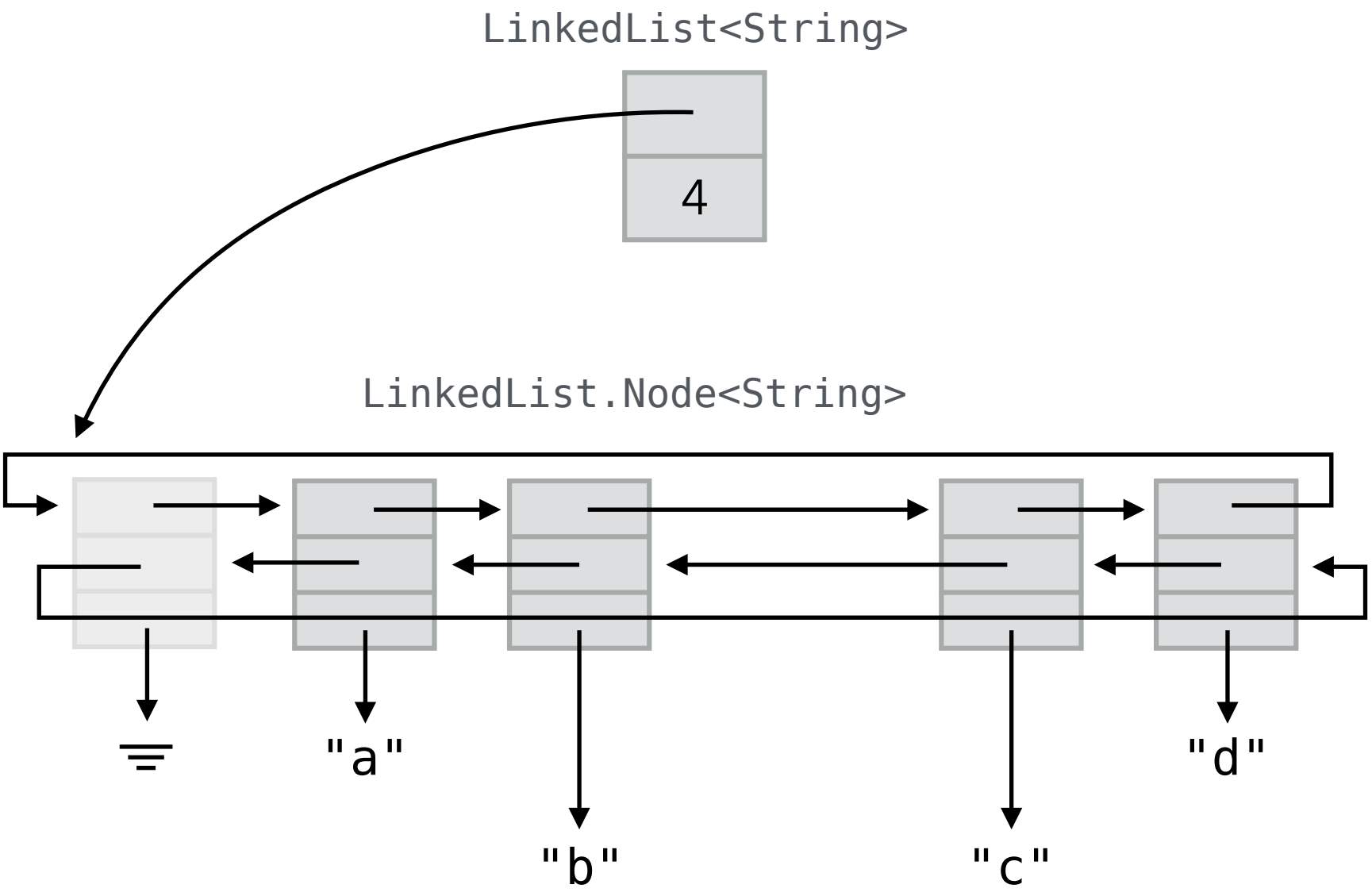
Création



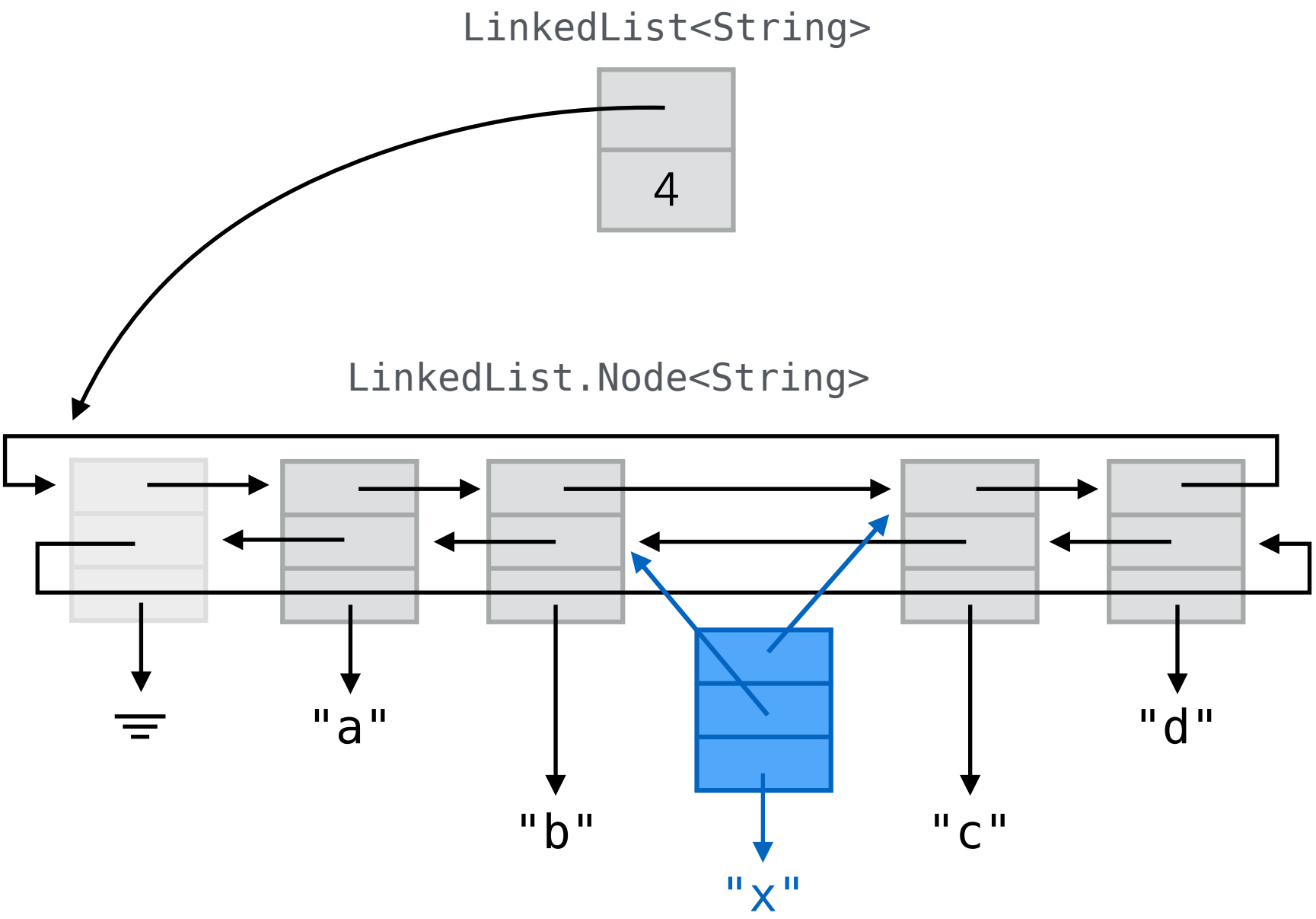
Insertion

L'insertion d'un élément dans une liste chaînée se fait par création d'un nouveau nœud le référençant et par chaînage de celui-ci.
Dans une liste doublement chaînée circulaire avec nœud d'en-tête, l'existence d'un successeur et d'un prédécesseur au nouveau nœud est garantie, rendant l'insertion particulièrement simple à mettre en œuvre. De plus, la présence d'un nœud d'en-tête garantit que la référence vers la tête de liste ne change jamais, même en cas d'insertion en première position.

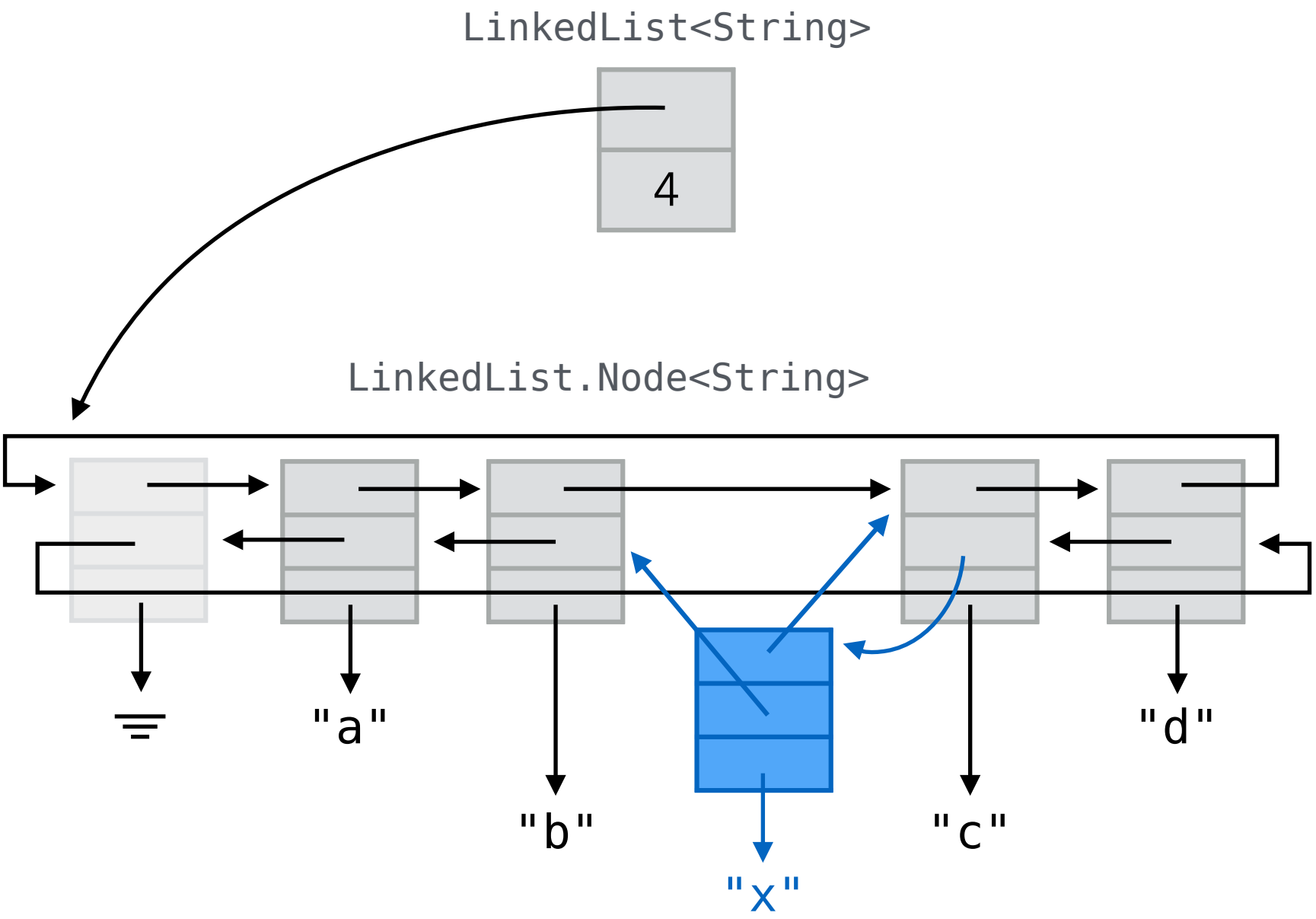
Insertion



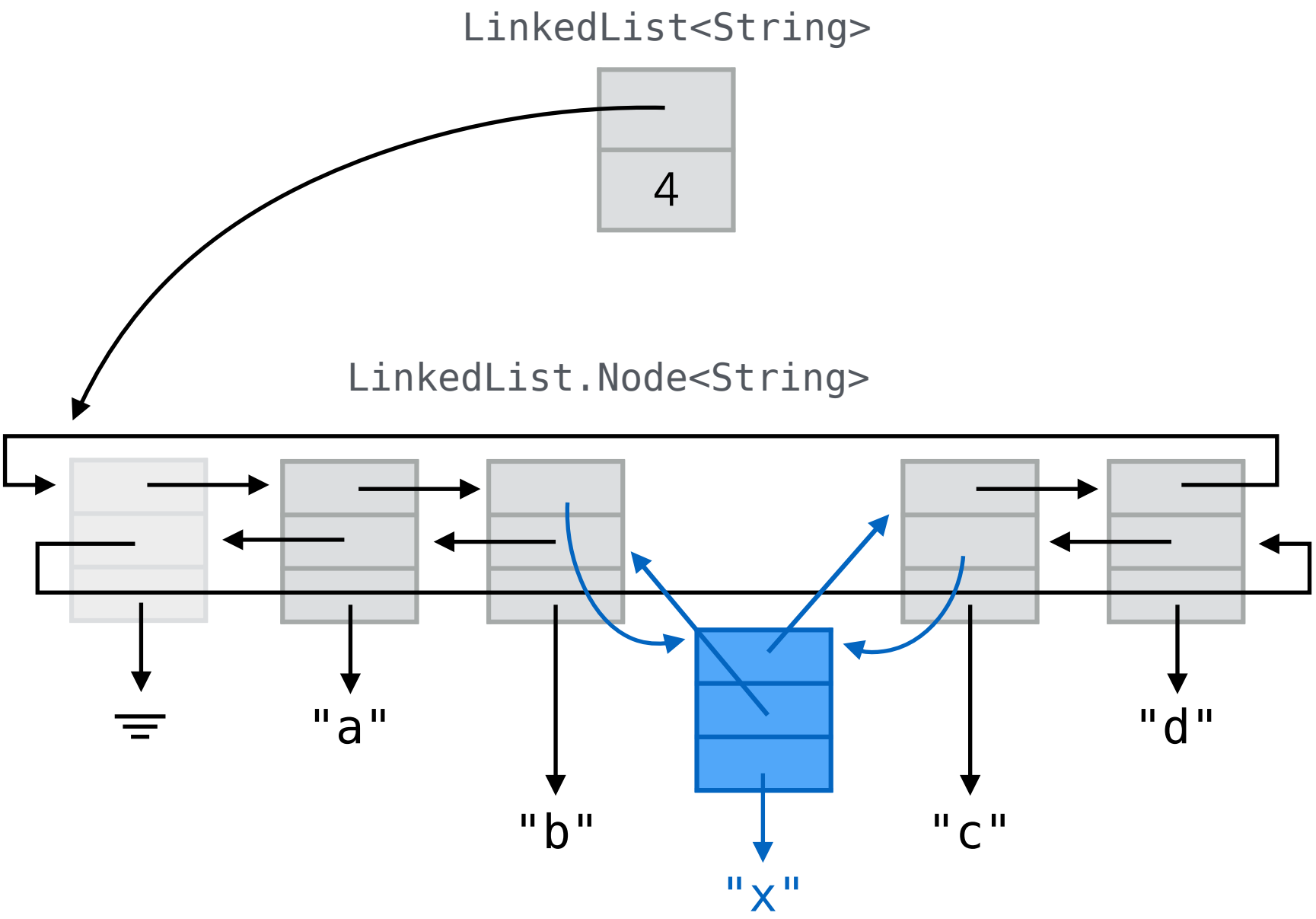
Insertion



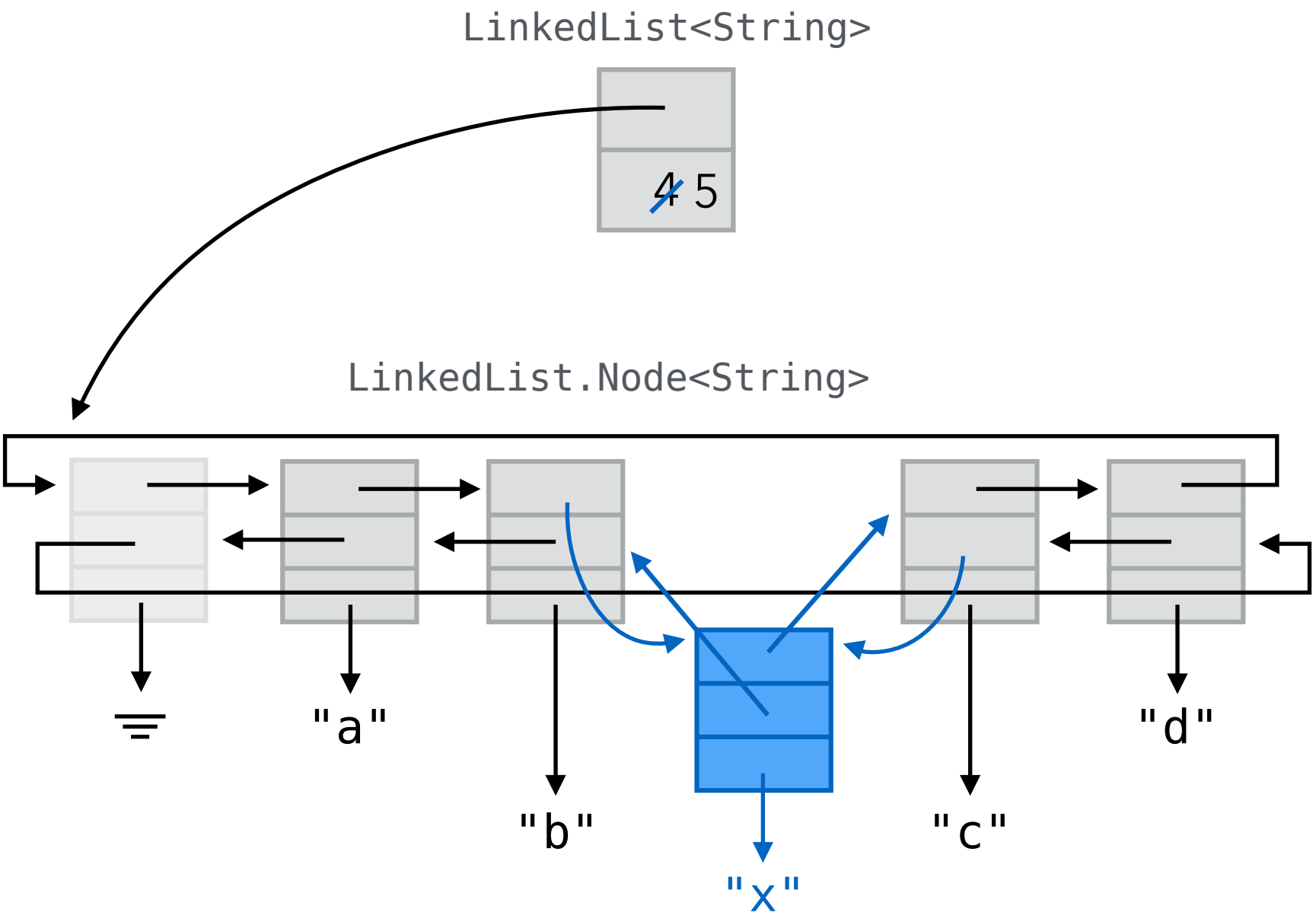
Insertion



Insertion



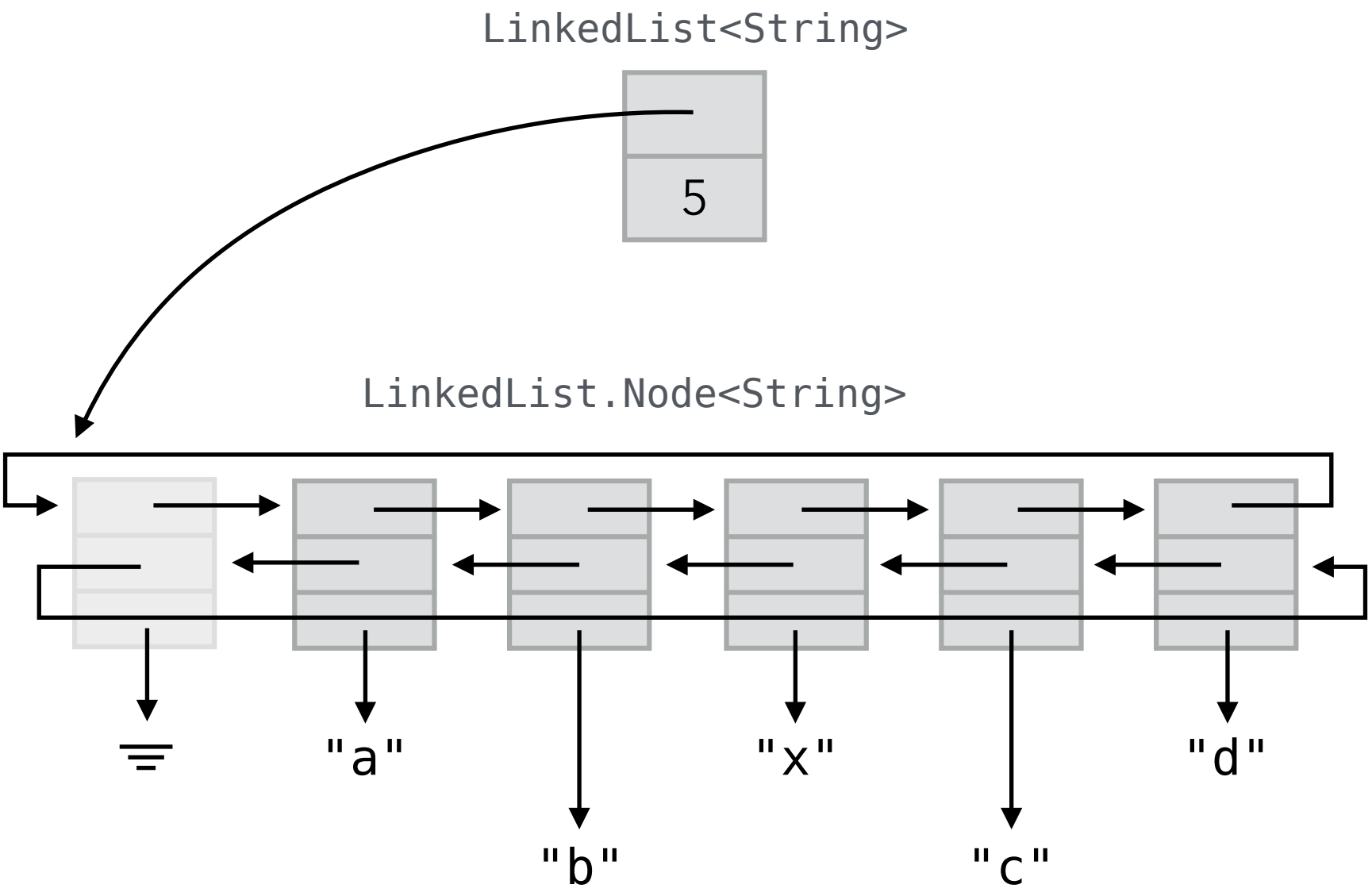
Insertion



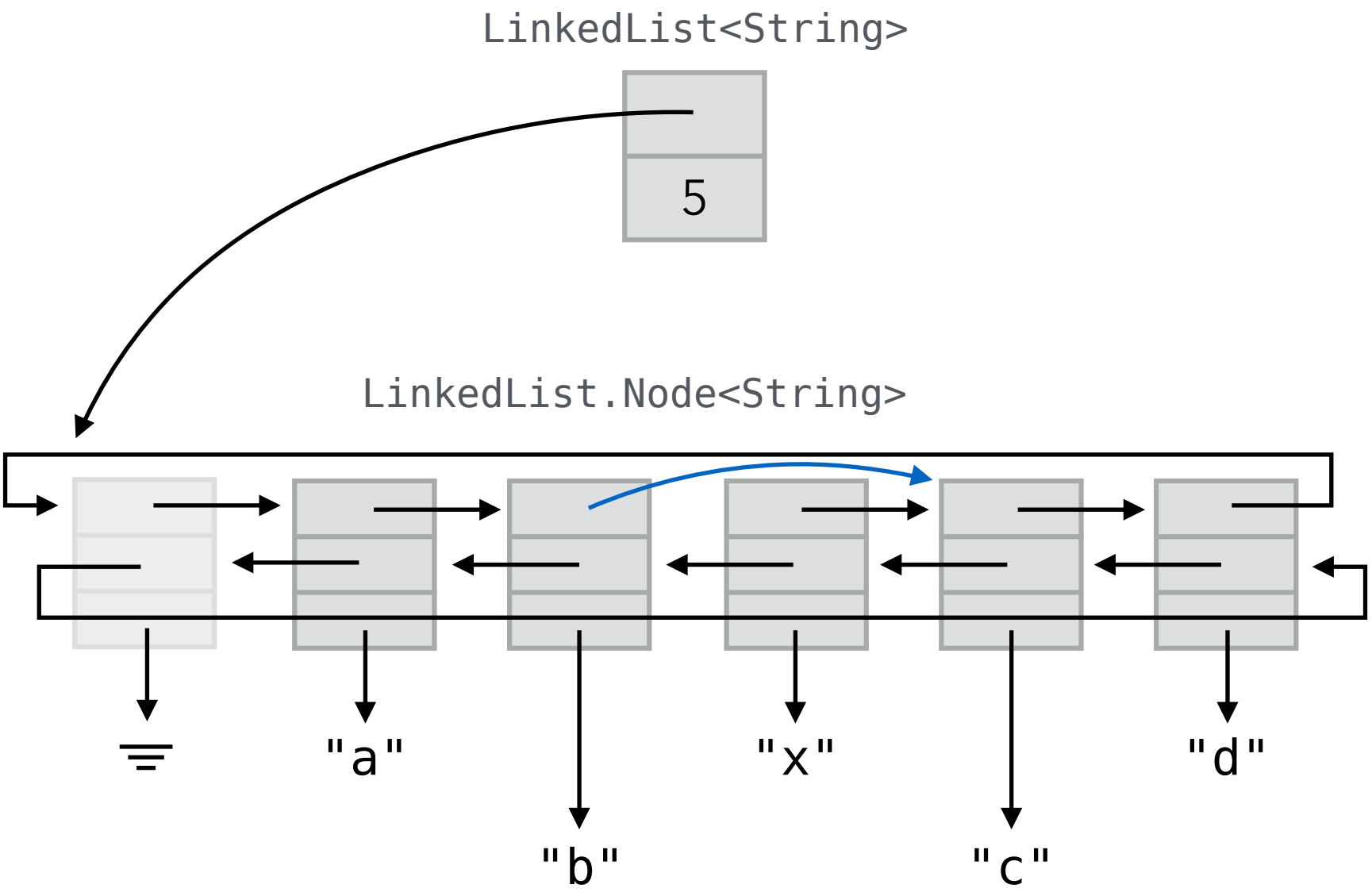
Suppression

La suppression d'un élément d'une liste chaînée se fait par dé-chaînage de son nœud, c-à-d par ajustement des liens de ses voisins.
Le chainage double circulaire garantit l'existence de ces deux voisins, et la présence d'un nœud d'en-tête garantit que la référence vers la tête de la liste ne doit jamais être changée, même en cas de suppression du premier élément.
Ces caractéristiques rendent la suppression très simple.

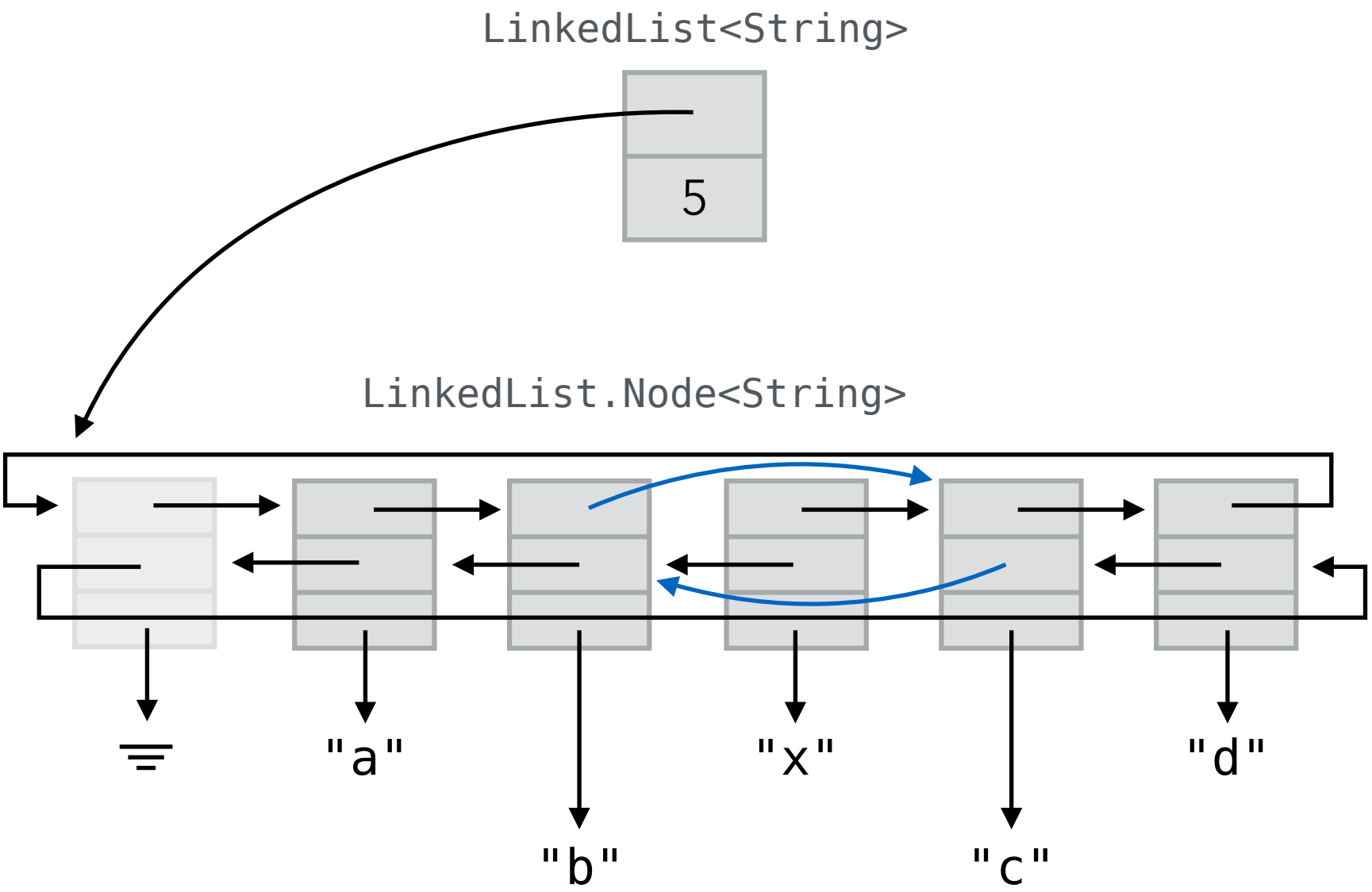
Suppression



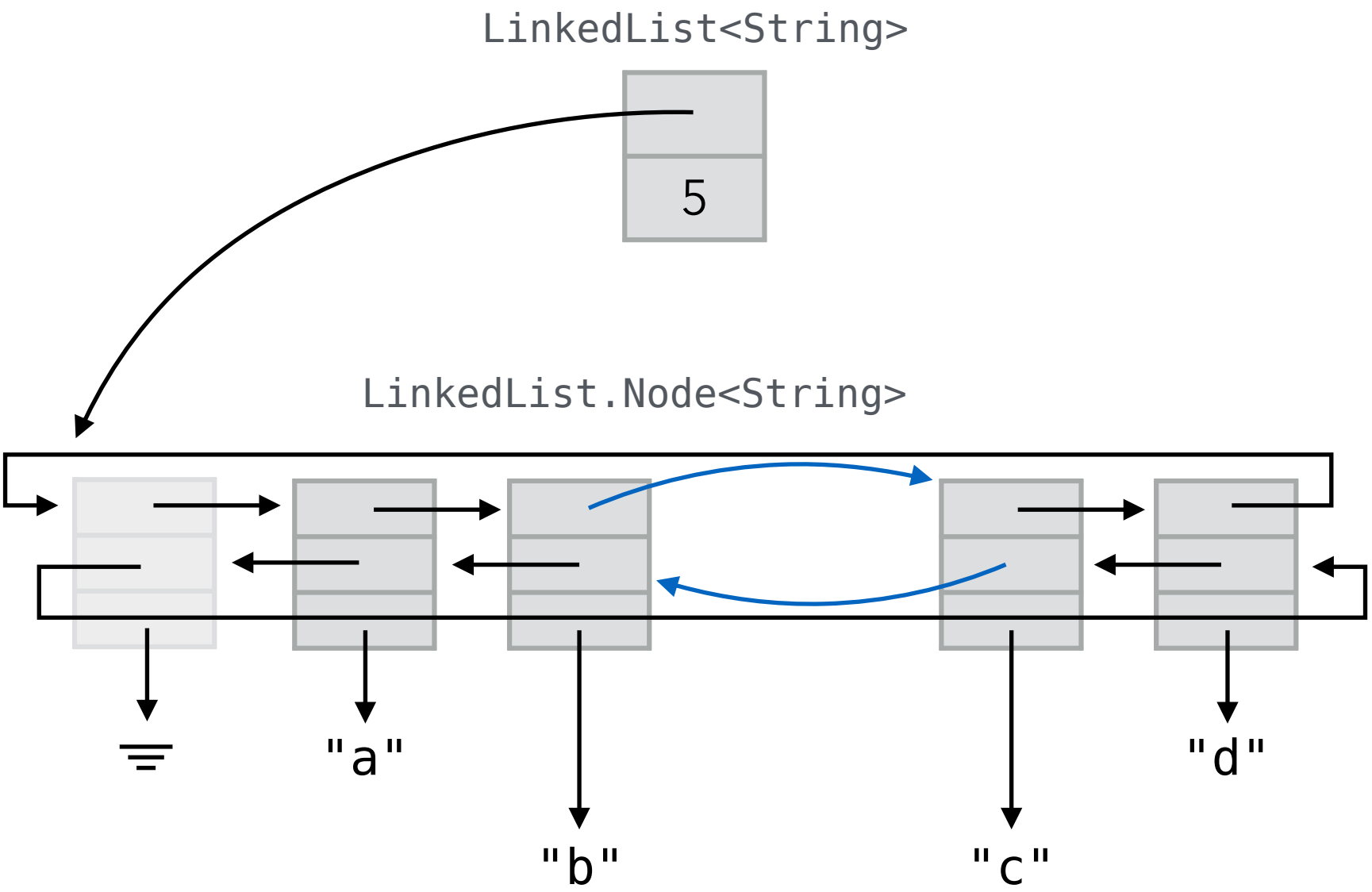
Suppression



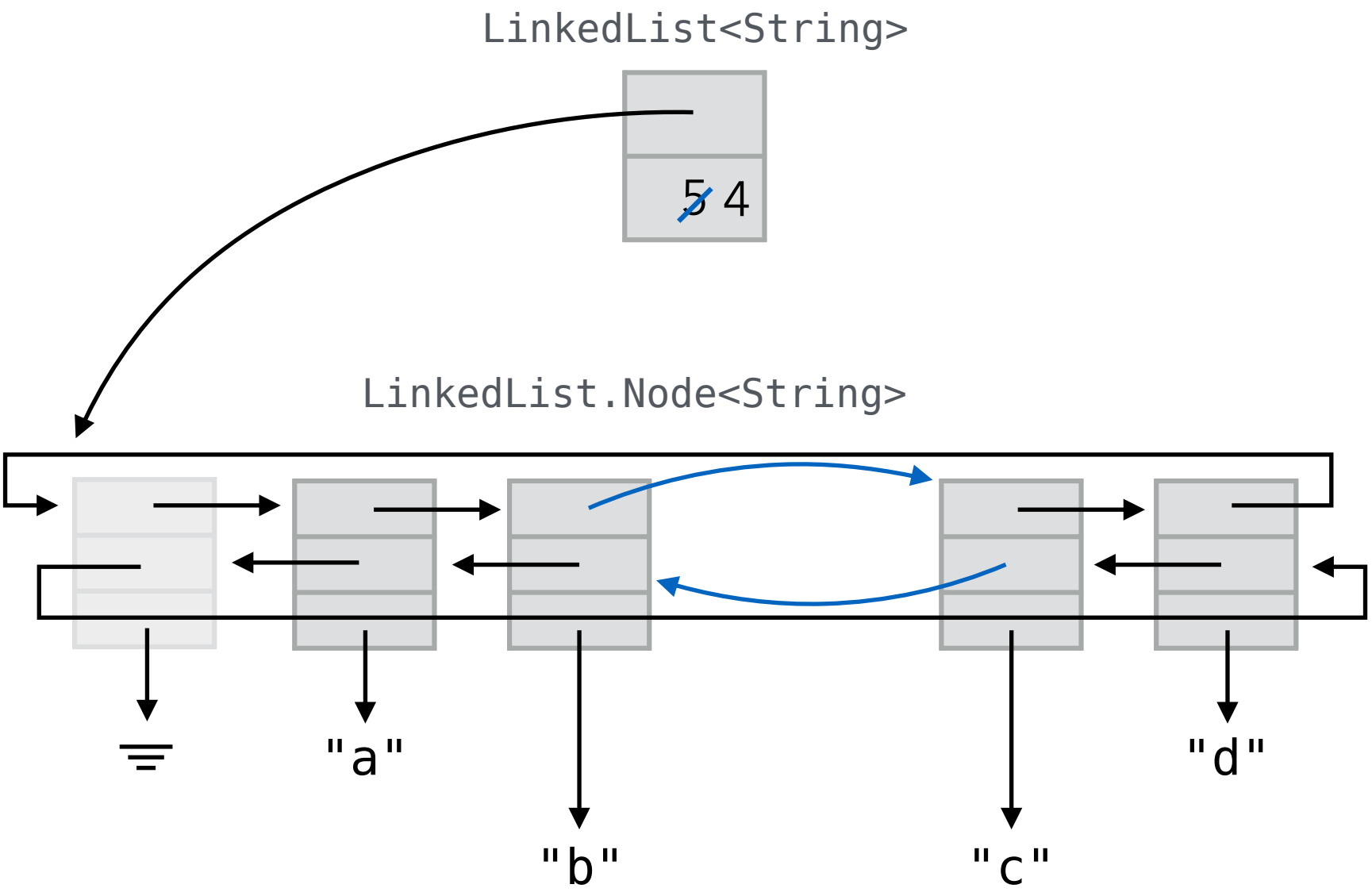
Suppression



Suppression



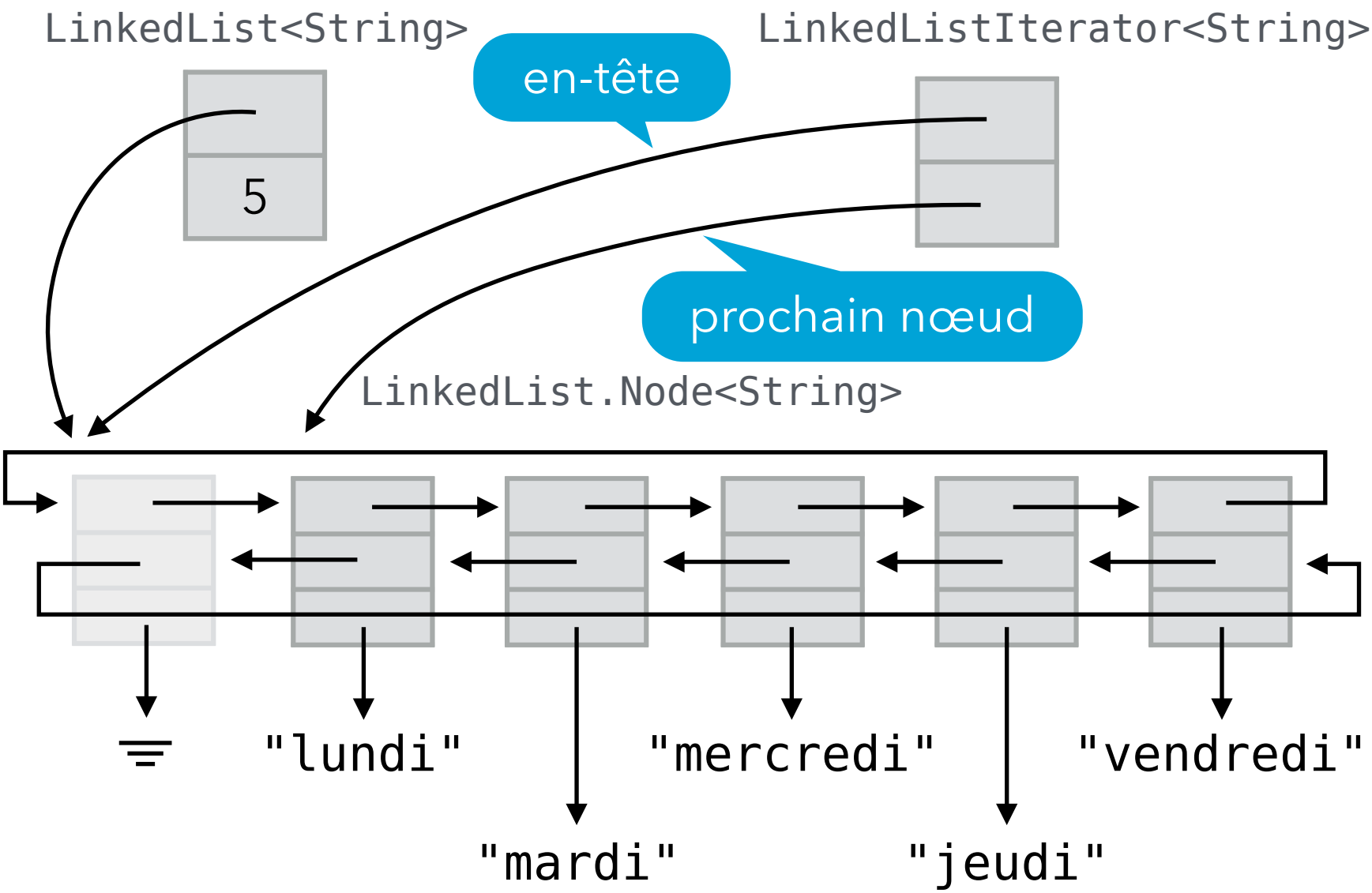
Suppression



Itération

Contrairement aux tableaux-listes, les listes chaînées ne doivent pas être parcourues au moyen d'un index et de la méthode `get`, pour des raisons de performance.
Un itérateur de liste chaînée doit donc avoir connaissance des nœuds. Il possède une référence vers le nœud d'en-tête et une vers le prochain nœud de l'itération.
La méthode `next` extrait (puis retourne) la valeur associée au prochain nœud, tout en avançant la référence vers son successeur. L'itération est terminée lorsque le prochain nœud est celui d'en-tête.

Itération


LinkedList<String>
LinkedListIterator<String>
en-tête
5
prochain nœud
LinkedList.Node<String>
"lundi"
"mercredi"
"vendredi"
"mardi"
"jeudi"

Exercice (en classe)

Ecrivez la classe `LinkedList`.

Itération par boucle *for-each*

Boucle *for-each*

Comme nous l'avons vu, la boucle *for-each* en Java est traduite en utilisation d'itérateurs.
Logiquement, Java offre la possibilité d'utiliser la boucle *for-each* pour toute collection que l'on peut parcourir au moyen d'un itérateur. Il faut toutefois que ce soit un itérateur Java, c-à-d qu'il ait le type `java.util.Iterator`.
Cette contrainte est exprimée par l'interface `Iterable`, qui doit être implémentée par toute collection que l'on désire pouvoir parcourir au moyen de la boucle *for-each*.

Interface `Iterable`

L'interface `java.lang.Iterable` ne déclare que la méthode `iterator`, qui permet d'obtenir un itérateur – de type `java.util.Iterator` – sur la collection à laquelle on l'applique.

Bien entendu, `Iterable` est générique, son paramètre de type donnant le type des éléments de la collection à parcourir.

```
public interface Iterable<E> {
  public Iterator<E> iterator();
}
```

Listes itérables

Pour rendre nos listes « itérables » au moyen de la boucle *for-each*, il suffit donc d'utiliser l'interface `Iterator` de `java.util` plutôt que la nôtre, et de faire implémenter à notre interface `List` l'interface `Iterable` :

```
public interface List<E>
  implements Iterable<E> { … }
```

On peut ensuite les parcourir au moyen de la boucle *for-each*, p.ex. :

```
List<String> l = …;
for (String e: l)
  System.out.println(e);
```