Structures de Données

TD2 – Les listes

Sébastien Da Silva

Telecom Nancy

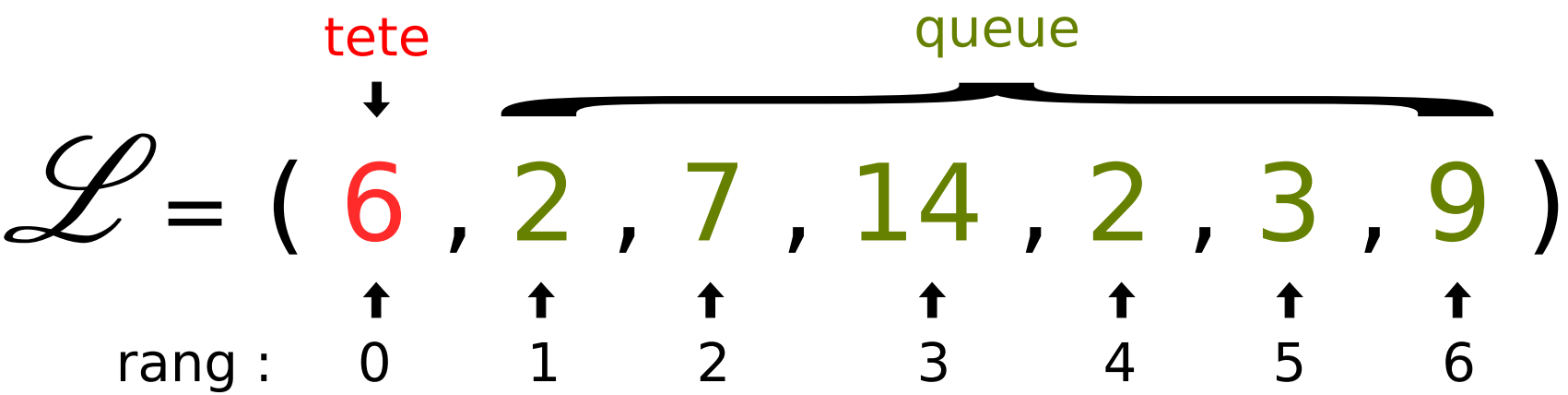
Semaine 12 - Mars 2016

Plan

1. Présentation des listes
2. Spécification algébrique
3. Implantations de $MyList[E]$
4. Performance et complexité

Plan

Vocabulaire



Tête 6 est la tête de L.

Queue $(2, 7, 14, 2, 3, 9)$ est la queue de L.

Rang L'élément de rang 2 est 7.

Sous-liste $(7, 14, 2, 3)$ est une sous-liste de L.

Préfixe $(6, 2, 7)$ est un préfixe de L.

Suffixe $(3, 9)$ est un suffixe de L.

À vous !

$$\mathscr{L} = (\,4\,,\,7\,,\,9\,,\,9\,,\,3\,,\,7\,,\,8\,,\,12\,)$$

Tête ?

Queue ?

Rang ?

Sous-liste ?

Préfixe ?

Suffixe ?

À vous !

$$\mathscr{L} = (\ 4\ ,\ 7\ ,\ 9\ ,\ 9\ ,\ 3\ ,\ 7\ ,\ 8\ ,\ 12\ )$$

Tête ? 4 est la tête de L.

Queue ? $(7, 9, 9, 3, 7, 8, 12)$ est la queue de L.

Rang ? L'élément de rang 2 est 9.

Sous-liste ? $(3, 7, 8)$ est une sous-liste de longueur 3.

Préfixe ? $(4, 7, 9, 9)$ est le préfixe de longueur 4.

Suffixe ? $(9, 9, 3, 7, 8, 12)$ est le suffixe de longueur 6.

Plan

Opérations

$$
\begin{aligned}
empty &: && \rightarrow MyList[E] \\
first &: MyList[E] && \rightarrow E \\
queue &: MyList[E] && \rightarrow MyList[E] \\
get &: MyList[E] \times Integer && \rightarrow E \\
contains &: MyList[E] \times E && \rightarrow Boolean \\
size &: MyList[E] && \rightarrow Integer \\
isEmpty &: MyList[E] && \rightarrow Boolean \\
indexOf &: MyList[E] \times E && \rightarrow Integer \\
addFirst &: MyList[E] \times E && \rightarrow MyList[E] \\
addLast &: MyList[E] \times E && \rightarrow MyList[E] \\
add &: MyList[E] \times E \times Integer && \rightarrow MyList[E] \\
add &: MyList[E] \times MyList[E] && \rightarrow MyList[E] \\
remove &: MyList[E] \times E && \rightarrow MyList[E] \\
remove &: MyList[E] \times Integer && \rightarrow MyList[E] \\
set &: MyList[E] \times Integer \times E && \rightarrow MyList[E]
\end{aligned}
$$

Opérations

$$
\begin{array}{rll}
empty : & & \rightarrow MyList[E] \\
first : & MyList[E] & \rightarrow E \\
queue : & MyList[E] & \rightarrow MyList[E] \\
get : & MyList[E] \times Integer & \rightarrow E \\
contains : & MyList[E] \times E & \rightarrow Boolean \\
size : & MyList[E] & \rightarrow Integer \\
isEmpty : & MyList[E] & \rightarrow Boolean \\
indexOf : & MyList[E] \times E & \rightarrow Integer \\
addFirst : & MyList[E] \times E & \rightarrow MyList[E] \\
addLast : & MyList[E] \times E & \rightarrow MyList[E] \\
add : & MyList[E] \times E \times Integer & \rightarrow MyList[E] \\
add : & MyList[E] \times MyList[E] & \rightarrow MyList[E] \\
remove : & MyList[E] \times E & \rightarrow MyList[E] \\
remove : & MyList[E] \times Integer & \rightarrow MyList[E] \\
set : & MyList[E] \times Integer \times E & \rightarrow MyList[E]
\end{array}
$$

Identifier les opérations internes et les observateurs du type $MyList[E]$

Opérations

$$
\begin{array}{rll}
empty : & & \rightarrow MyList[E] \\
first : & MyList[E] & \rightarrow E \\
queue : & MyList[E] & \rightarrow MyList[E] \\
get : & MyList[E] \times Integer & \rightarrow E \\
contains : & MyList[E] \times E & \rightarrow Boolean \\
size : & MyList[E] & \rightarrow Integer \\
isEmpty : & MyList[E] & \rightarrow Boolean \\
indexOf : & MyList[E] \times E & \rightarrow Integer \\
addFirst : & MyList[E] \times E & \rightarrow MyList[E] \\
addLast : & MyList[E] \times E & \rightarrow MyList[E] \\
add : & MyList[E] \times E \times Integer & \rightarrow MyList[E] \\
add : & MyList[E] \times MyList[E] & \rightarrow MyList[E] \\
remove : & MyList[E] \times E & \rightarrow MyList[E] \\
remove : & MyList[E] \times Integer & \rightarrow MyList[E] \\
set : & MyList[E] \times Integer \times E & \rightarrow MyList[E]
\end{array}
$$

Opérations

$$
\begin{array}{rll}
\mathit{empty} : & & \to \mathit{MyList}[E] \\
\mathit{first} : & \mathit{MyList}[E] & \to E \\
\mathit{queue} : & \mathit{MyList}[E] & \to \mathit{MyList}[E] \\
\mathit{get} : & \mathit{MyList}[E] \times \mathit{Integer} & \to E \\
\mathit{contains} : & \mathit{MyList}[E] \times E & \to \mathit{Boolean} \\
\mathit{size} : & \mathit{MyList}[E] & \to \mathit{Integer} \\
\mathit{isEmpty} : & \mathit{MyList}[E] & \to \mathit{Boolean} \\
\mathit{indexOf} : & \mathit{MyList}[E] \times E & \to \mathit{Integer} \\
\mathit{addFirst} : & \mathit{MyList}[E] \times E & \to \mathit{MyList}[E] \\
\mathit{addLast} : & \mathit{MyList}[E] \times E & \to \mathit{MyList}[E] \\
\mathit{add} : & \mathit{MyList}[E] \times E \times \mathit{Integer} & \to \mathit{MyList}[E] \\
\mathit{add} : & \mathit{MyList}[E] \times \mathit{MyList}[E] & \to \mathit{MyList}[E] \\
\mathit{remove} : & \mathit{MyList}[E] \times E & \to \mathit{MyList}[E] \\
\mathit{remove} : & \mathit{MyList}[E] \times \mathit{Integer} & \to \mathit{MyList}[E] \\
\mathit{set} : & \mathit{MyList}[E] \times \mathit{Integer} \times E & \to \mathit{MyList}[E]
\end{array}
$$

Les 5 préconditions

Déterminer les préconditions pour les opérations du type $MyList[E]$

Les 5 préconditions

$add(l, e, i)$

Les 5 préconditions

$$add(l, e, i)$$
$$first(l)$$

Les 5 préconditions

$$add(l, e, i)$$
$$first(l)$$
$$remove(l, i)$$

Les 5 préconditions

$$add(l, e, i)$$
$$first(l)$$
$$remove(l, i)$$
$$get(l, i)$$

Les 5 préconditions

$$add(l, e, i)$$
$$first(l)$$
$$remove(l, i)$$
$$get(l, i)$$
$$set(l, i, e)$$

Les 5 préconditions

$add(l, e, i)$	defini ssi
$first(l)$	defini ssi
$remove(l, i)$	defini ssi
$get(l, i)$	defini ssi
$set(l, i, e)$	defini ssi

Les 5 préconditions

$$
\begin{array}{rll}
add(l, e, i) & \text{defini ssi} & 0 \leq i \leq size(l) \\
first(l) & \text{defini ssi} & \\
remove(l, i) & \text{defini ssi} & \\
get(l, i) & \text{defini ssi} & \\
set(l, i, e) & \text{defini ssi} &
\end{array}
$$

Les 5 préconditions

$$
\begin{array}{rll}
add(l, e, i) & \text{defini ssi} & 0 \leq i \leq size(l) \\
first(l) & \text{defini ssi} & non\ isEmpty(l) \\
remove(l, i) & \text{defini ssi} & \\
get(l, i) & \text{defini ssi} & \\
set(l, i, e) & \text{defini ssi} &
\end{array}
$$

Les 5 préconditions

$$
\begin{array}{rll}
add(l, e, i) & \text{defini ssi} & 0 \leq i \leq size(l) \\
first(l) & \text{defini ssi} & non\ isEmpty(l) \\
remove(l, i) & \text{defini ssi} & 0 \leq i < size(l) \\
get(l, i) & \text{defini ssi} & \\
set(l, i, e) & \text{defini ssi} &
\end{array}
$$

Les 5 préconditions

$$
\begin{array}{rll}
add(l, e, i) & \text{defini ssi} & 0 \leq i \leq size(l) \\
first(l) & \text{defini ssi} & non\ isEmpty(l) \\
remove(l, i) & \text{defini ssi} & 0 \leq i < size(l) \\
get(l, i) & \text{defini ssi} & 0 \leq i < size(l) \\
set(l, i, e) & \text{defini ssi} &
\end{array}
$$

Les 5 préconditions

$$
\begin{array}{rll}
add(l, e, i) & \text{defini ssi} & 0 \leq i \leq size(l) \\
first(l) & \text{defini ssi} & non\ isEmpty(l) \\
remove(l, i) & \text{defini ssi} & 0 \leq i < size(l) \\
get(l, i) & \text{defini ssi} & 0 \leq i < size(l) \\
set(l, i, e) & \text{defini ssi} & 0 \leq i < size(l)
\end{array}
$$

Intégrer les préconditions

Trois façons d'envisager les **préconditions** en Java :

1. **supposer** qu'elles seront respectées
2. **s'assurer** qu'elles seront respectées
   $\rightarrow$ programmation défensive
3. **prévenir** lorsqu'elles ne sont pas respectées
   $\rightarrow$ programmation par contrat

Intégrer les préconditions

Trois façons d'envisager les **préconditions** en Java :

1. **supposer** qu'elles seront respectées
2. **s'assurer** qu'elles seront respectées
   → programmation défensive
3. **prévenir** lorsqu'elles ne sont pas respectées
   → programmation par contrat

```
1  // Attention : denominateur doit être différent de 0 !
2  public int diviser( int numerateur, int denominateur )
3  {
4
5    return numerateur/denominateur;
6
7  }
```

Intégrer les préconditions

Trois façons d'envisager les **préconditions** en Java :

1. **supposer** qu'elles seront respectées
2. **s'assurer** qu'elles seront respectées
   → programmation défensive
3. **prévenir** lorsqu'elles ne sont pas respectées
   → programmation par contrat

```
1  public int diviser( int numerateur, int denominateur )
2                                  throws IllegalArgumentException
3  {
4    if ( denominateur == 0 )
5      throw new IllegalArgumentException("denominateur==0");
6    return numerateur/denominateur;
7  }
```

Intégrer les préconditions

Trois façons d'envisager les **préconditions** en Java :

1. **supposer** qu'elles seront respectées
2. **s'assurer** qu'elles seront respectées
   → programmation défensive
3. **prévenir** lorsqu'elles ne sont pas respectées
   → programmation par contrat

```
1  public int diviser( int numerateur, int denominateur )
2  {
3    assert (denominateur != 0) :
4      "Violation précondition : denominateur == 0";
5    return numerateur/denominateur;
6  }
```

Exécution avec `java -ea`

Axiomes avec *first*

Axiomes avec *first*

$$first(addFirst(l, e)) \quad =$$

Axiomes avec *first*

$$first(addFirst(l, e)) \quad = \quad e$$

Axiomes avec *first*

$$first(addFirst(l, e)) = e$$

$$first(addLast(l, e)) =$$

Axiomes avec *first*

$$first(addFirst(l, e)) = e$$

$$first(addLast(l, e)) = \begin{cases} e & \text{si } l = empty() \\ first(l) & \text{sinon} \end{cases}$$

Axiomes avec *first*

$$first(addFirst(l, e)) = e$$

$$first(addLast(l, e)) = \begin{cases} e & \text{si } l = empty() \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l, e, i)) =$$

Axiomes avec *first*

$$first(addFirst(l, e)) = e$$

$$first(addLast(l, e)) = \begin{cases} e & \text{si } l = empty() \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l, e, i)) = \begin{cases} e & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

Axiomes avec *first*

$$first(addFirst(l, e)) = e$$

$$first(addLast(l, e)) = \begin{cases} e & \text{si } l = empty() \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l, e, i)) = \begin{cases} e & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l_1, l_2)) =$$

Axiomes avec *first*

$$first(addFirst(l, e)) = e$$

$$first(addLast(l, e)) = \begin{cases} e & \text{si } l = empty() \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l, e, i)) = \begin{cases} e & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l_1, l_2)) = \begin{cases} first(l_2) & \text{si } l_1 = empty() \\ first(l_1) & \text{sinon} \end{cases}$$

Axiomes avec *first*

$$first(addFirst(l, e)) = e$$

$$first(addLast(l, e)) = \begin{cases} e & \text{si } l = empty() \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l, e, i)) = \begin{cases} e & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l_1, l_2)) = \begin{cases} first(l_2) & \text{si } l_1 = empty() \\ first(l_1) & \text{sinon} \end{cases}$$

$$first(remove(l, i)) =$$

Axiomes avec *first*

$$first(addFirst(l, e)) = e$$

$$first(addLast(l, e)) = \begin{cases} e & \text{si } l = empty() \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l, e, i)) = \begin{cases} e & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l_1, l_2)) = \begin{cases} first(l_2) & \text{si } l_1 = empty() \\ first(l_1) & \text{sinon} \end{cases}$$

$$first(remove(l, i)) = \begin{cases} first(queue(l)) & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

Axiomes avec *first*

$$first(addFirst(l, e)) = e$$

$$first(addLast(l, e)) = \begin{cases} e & \text{si } l = empty() \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l, e, i)) = \begin{cases} e & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l_1, l_2)) = \begin{cases} first(l_2) & \text{si } l_1 = empty() \\ first(l_1) & \text{sinon} \end{cases}$$

$$first(remove(l, i)) = \begin{cases} first(queue(l)) & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(remove(l, e)) =$$

Axiomes avec *first*

$$first(addFirst(l, e)) = e$$

$$first(addLast(l, e)) = \begin{cases} e & \text{si } l = empty() \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l, e, i)) = \begin{cases} e & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l_1, l_2)) = \begin{cases} first(l_2) & \text{si } l_1 = empty() \\ first(l_1) & \text{sinon} \end{cases}$$

$$first(remove(l, i)) = \begin{cases} first(queue(l)) & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(remove(l, e)) = \begin{cases} first(queue(l)) & \text{si } indexOf(l, e) = 0 \\ first(l) & \text{sinon} \end{cases}$$

Axiomes avec *first*

$$first(addFirst(l, e)) = e$$

$$first(addLast(l, e)) = \begin{cases} e & \text{si } l = empty() \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l, e, i)) = \begin{cases} e & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l_1, l_2)) = \begin{cases} first(l_2) & \text{si } l_1 = empty() \\ first(l_1) & \text{sinon} \end{cases}$$

$$first(remove(l, i)) = \begin{cases} first(queue(l)) & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(remove(l, e)) = \begin{cases} first(queue(l)) & \text{si } indexOf(l, e) = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(set(l, i, e)) =$$

Axiomes avec *first*

$$first(addFirst(l, e)) = e$$

$$first(addLast(l, e)) = \begin{cases} e & \text{si } l = empty() \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l, e, i)) = \begin{cases} e & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l_1, l_2)) = \begin{cases} first(l_2) & \text{si } l_1 = empty() \\ first(l_1) & \text{sinon} \end{cases}$$

$$first(remove(l, i)) = \begin{cases} first(queue(l)) & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(remove(l, e)) = \begin{cases} first(queue(l)) & \text{si } indexOf(l, e) = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(set(l, i, e)) = \begin{cases} e & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

Axiomes avec *first*

$$first(addFirst(l, e)) = e$$

$$first(addLast(l, e)) = \begin{cases} e & \text{si } l = empty() \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l, e, i)) = \begin{cases} e & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l_1, l_2)) = \begin{cases} first(l_2) & \text{si } l_1 = empty() \\ first(l_1) & \text{sinon} \end{cases}$$

$$first(remove(l, i)) = \begin{cases} first(queue(l)) & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(remove(l, e)) = \begin{cases} first(queue(l)) & \text{si } indexOf(l, e) = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(set(l, i, e)) = \begin{cases} e & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(queue(l)) =$$

Axiomes avec *first*

$$first(addFirst(l, e)) = e$$

$$first(addLast(l, e)) = \begin{cases} e & \text{si } l = empty() \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l, e, i)) = \begin{cases} e & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(add(l_1, l_2)) = \begin{cases} first(l_2) & \text{si } l_1 = empty() \\ first(l_1) & \text{sinon} \end{cases}$$

$$first(remove(l, i)) = \begin{cases} first(queue(l)) & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(remove(l, e)) = \begin{cases} first(queue(l)) & \text{si } indexOf(l, e) = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(set(l, i, e)) = \begin{cases} e & \text{si } i = 0 \\ first(l) & \text{sinon} \end{cases}$$

$$first(queue(l)) = get(l, 1)$$

Axiomes avec *empty*() et *addFirst*()

Axiomes avec *empty*() et *addFirst*()

$$get(empty(), e) =$$

Axiomes avec *empty*() et *addFirst*()

$$get(empty(), e) = \text{Violation de précondition !}$$

Axiomes avec *empty*() et *addFirst*()

$$get(empty(), e) = \text{Violation de précondition !}$$

$$get(addFirst(l, e), i) =$$

Axiomes avec *empty*() et *addFirst*()

$$get(empty(), e) = \text{Violation de précondition !}$$

$$get(addFirst(l, e), i) = \begin{cases} e & \text{si } i = 0 \\ get(l, i-1) & \text{sinon} \end{cases}$$

Axiomes avec *empty*() et *addFirst*()

$$get(empty(), e) = \text{Violation de précondition !}$$

$$get(addFirst(l, e), i) = \begin{cases} e & \text{si } i = 0 \\ get(l, i-1) & \text{sinon} \end{cases}$$

$$contains(empty(), e) =$$

Axiomes avec *empty*() et *addFirst*()

$$
\begin{aligned}
get(empty(), e) &= \text{Violation de précondition !} \\
get(addFirst(l, e), i) &= \begin{cases} e & \text{si } i = 0 \\ get(l, i-1) & \text{sinon} \end{cases} \\
contains(empty(), e) &= faux
\end{aligned}
$$

Axiomes avec *empty*() et *addFirst*()

$$
\begin{aligned}
get(empty(), e) &= \text{Violation de précondition !} \\
get(addFirst(l, e), i) &= \begin{cases} e & \text{si } i = 0 \\ get(l, i-1) & \text{sinon} \end{cases} \\
contains(empty(), e) &= faux \\
contains(addFirst(l, e_1), e_2) &=
\end{aligned}
$$

Axiomes avec *empty*() et *addFirst*()

$$get(empty(), e) = \text{Violation de précondition !}$$

$$get(addFirst(l, e), i) = \begin{cases} e & \text{si } i = 0 \\ get(l, i-1) & \text{sinon} \end{cases}$$

$$contains(empty(), e) = faux$$

$$contains(addFirst(l, e_1), e_2) = \begin{cases} vrai & \text{si } e_1 = e_2 \\ contains(l, e_2) & \text{sinon} \end{cases}$$

Axiomes avec *empty*() et *addFirst*()

$$get(empty(), e) = \text{Violation de précondition !}$$

$$get(addFirst(l, e), i) = \begin{cases} e & \text{si } i = 0 \\ get(l, i-1) & \text{sinon} \end{cases}$$

$$contains(empty(), e) = faux$$

$$contains(addFirst(l, e_1), e_2) = \begin{cases} vrai & \text{si } e_1 = e_2 \\ contains(l, e_2) & \text{sinon} \end{cases}$$

$$size(empty()) =$$

Axiomes avec *empty*() et *addFirst*()

$$
\begin{aligned}
get(empty(), e) &= \text{Violation de précondition !} \\
get(addFirst(l, e), i) &= \begin{cases} e & \text{si } i = 0 \\ get(l, i-1) & \text{sinon} \end{cases} \\
contains(empty(), e) &= faux \\
contains(addFirst(l, e_1), e_2) &= \begin{cases} vrai & \text{si } e_1 = e_2 \\ contains(l, e_2) & \text{sinon} \end{cases} \\
size(empty()) &= 0
\end{aligned}
$$

Axiomes avec *empty*() et *addFirst*()

$$
\begin{aligned}
get(empty(), e) &= \text{Violation de précondition !} \\
get(addFirst(l, e), i) &= \begin{cases} e & \text{si } i = 0 \\ get(l, i-1) & \text{sinon} \end{cases} \\
contains(empty(), e) &= faux \\
contains(addFirst(l, e_1), e_2) &= \begin{cases} vrai & \text{si } e_1 = e_2 \\ contains(l, e_2) & \text{sinon} \end{cases} \\
size(empty()) &= 0 \\
size(addFirst(l, e)) &=
\end{aligned}
$$

Axiomes avec *empty*() et *addFirst*()

$$
\begin{aligned}
get(empty(), e) &= \text{Violation de précondition !} \\
get(addFirst(l, e), i) &= \begin{cases} e & \text{si } i = 0 \\ get(l, i-1) & \text{sinon} \end{cases} \\
contains(empty(), e) &= faux \\
contains(addFirst(l, e_1), e_2) &= \begin{cases} vrai & \text{si } e_1 = e_2 \\ contains(l, e_2) & \text{sinon} \end{cases} \\
size(empty()) &= 0 \\
size(addFirst(l, e)) &= size(l) + 1
\end{aligned}
$$

Axiomes avec *empty*() et *addFirst*()

$$
\begin{aligned}
get(empty(), e) &= \text{Violation de précondition !} \\
get(addFirst(l, e), i) &= \begin{cases} e & \text{si } i = 0 \\ get(l, i-1) & \text{sinon} \end{cases} \\
contains(empty(), e) &= faux \\
contains(addFirst(l, e_1), e_2) &= \begin{cases} vrai & \text{si } e_1 = e_2 \\ contains(l, e_2) & \text{sinon} \end{cases} \\
size(empty()) &= 0 \\
size(addFirst(l, e)) &= size(l) + 1 \\
isEmpty(empty()) &=
\end{aligned}
$$

Axiomes avec *empty*() et *addFirst*()

$$
\begin{aligned}
get(empty(), e) &= \text{Violation de précondition !} \\
get(addFirst(l, e), i) &= \begin{cases} e & \text{si } i = 0 \\ get(l, i-1) & \text{sinon} \end{cases} \\
contains(empty(), e) &= faux \\
contains(addFirst(l, e_1), e_2) &= \begin{cases} vrai & \text{si } e_1 = e_2 \\ contains(l, e_2) & \text{sinon} \end{cases} \\
size(empty()) &= 0 \\
size(addFirst(l, e)) &= size(l) + 1 \\
isEmpty(empty()) &= vrai
\end{aligned}
$$

Axiomes avec *empty*() et *addFirst*()

$$
\begin{aligned}
get(empty(), e) &= \text{Violation de précondition !} \\
get(addFirst(l, e), i) &= \begin{cases} e & \text{si } i = 0 \\ get(l, i-1) & \text{sinon} \end{cases} \\
contains(empty(), e) &= faux \\
contains(addFirst(l, e_1), e_2) &= \begin{cases} vrai & \text{si } e_1 = e_2 \\ contains(l, e_2) & \text{sinon} \end{cases} \\
size(empty()) &= 0 \\
size(addFirst(l, e)) &= size(l) + 1 \\
isEmpty(empty()) &= vrai \\
isEmpty(addFirst(l, e)) &=
\end{aligned}
$$

Axiomes avec *empty*() et *addFirst*()

$$
\begin{aligned}
get(empty(), e) &= \text{Violation de précondition !} \\
get(addFirst(l, e), i) &= \begin{cases} e & \text{si } i = 0 \\ get(l, i-1) & \text{sinon} \end{cases} \\
contains(empty(), e) &= faux \\
contains(addFirst(l, e_1), e_2) &= \begin{cases} vrai & \text{si } e_1 = e_2 \\ contains(l, e_2) & \text{sinon} \end{cases} \\
size(empty()) &= 0 \\
size(addFirst(l, e)) &= size(l) + 1 \\
isEmpty(empty()) &= vrai \\
isEmpty(addFirst(l, e)) &= faux
\end{aligned}
$$

Axiomes avec *empty*() et *addFirst*()

$$
\begin{aligned}
get(empty(), e) &= \text{Violation de précondition !} \\
get(addFirst(l, e), i) &= \begin{cases} e & \text{si } i = 0 \\ get(l, i-1) & \text{sinon} \end{cases} \\
contains(empty(), e) &= faux \\
contains(addFirst(l, e_1), e_2) &= \begin{cases} vrai & \text{si } e_1 = e_2 \\ contains(l, e_2) & \text{sinon} \end{cases} \\
size(empty()) &= 0 \\
size(addFirst(l, e)) &= size(l) + 1 \\
isEmpty(empty()) &= vrai \\
isEmpty(addFirst(l, e)) &= faux \\
indexOf(empty(), e) &=
\end{aligned}
$$

Axiomes avec *empty*() et *addFirst*()

$$
\begin{aligned}
get(empty(), e) &= \text{Violation de précondition !} \\
get(addFirst(l, e), i) &= \begin{cases} e & \text{si } i = 0 \\ get(l, i-1) & \text{sinon} \end{cases} \\
contains(empty(), e) &= faux \\
contains(addFirst(l, e_1), e_2) &= \begin{cases} vrai & \text{si } e_1 = e_2 \\ contains(l, e_2) & \text{sinon} \end{cases} \\
size(empty()) &= 0 \\
size(addFirst(l, e)) &= size(l) + 1 \\
isEmpty(empty()) &= vrai \\
isEmpty(addFirst(l, e)) &= faux \\
indexOf(empty(), e) &= -1 \text{ (élément non trouvé)}
\end{aligned}
$$

Axiomes avec *empty*() et *addFirst*()

$$
\begin{aligned}
get(empty(), e) &= \text{Violation de précondition !} \\
get(addFirst(l, e), i) &= \begin{cases} e & \text{si } i = 0 \\ get(l, i-1) & \text{sinon} \end{cases} \\
contains(empty(), e) &= faux \\
contains(addFirst(l, e_1), e_2) &= \begin{cases} vrai & \text{si } e_1 = e_2 \\ contains(l, e_2) & \text{sinon} \end{cases} \\
size(empty()) &= 0 \\
size(addFirst(l, e)) &= size(l) + 1 \\
isEmpty(empty()) &= vrai \\
isEmpty(addFirst(l, e)) &= faux \\
indexOf(empty(), e) &= -1 \text{ (élément non trouvé)}
\end{aligned}
$$

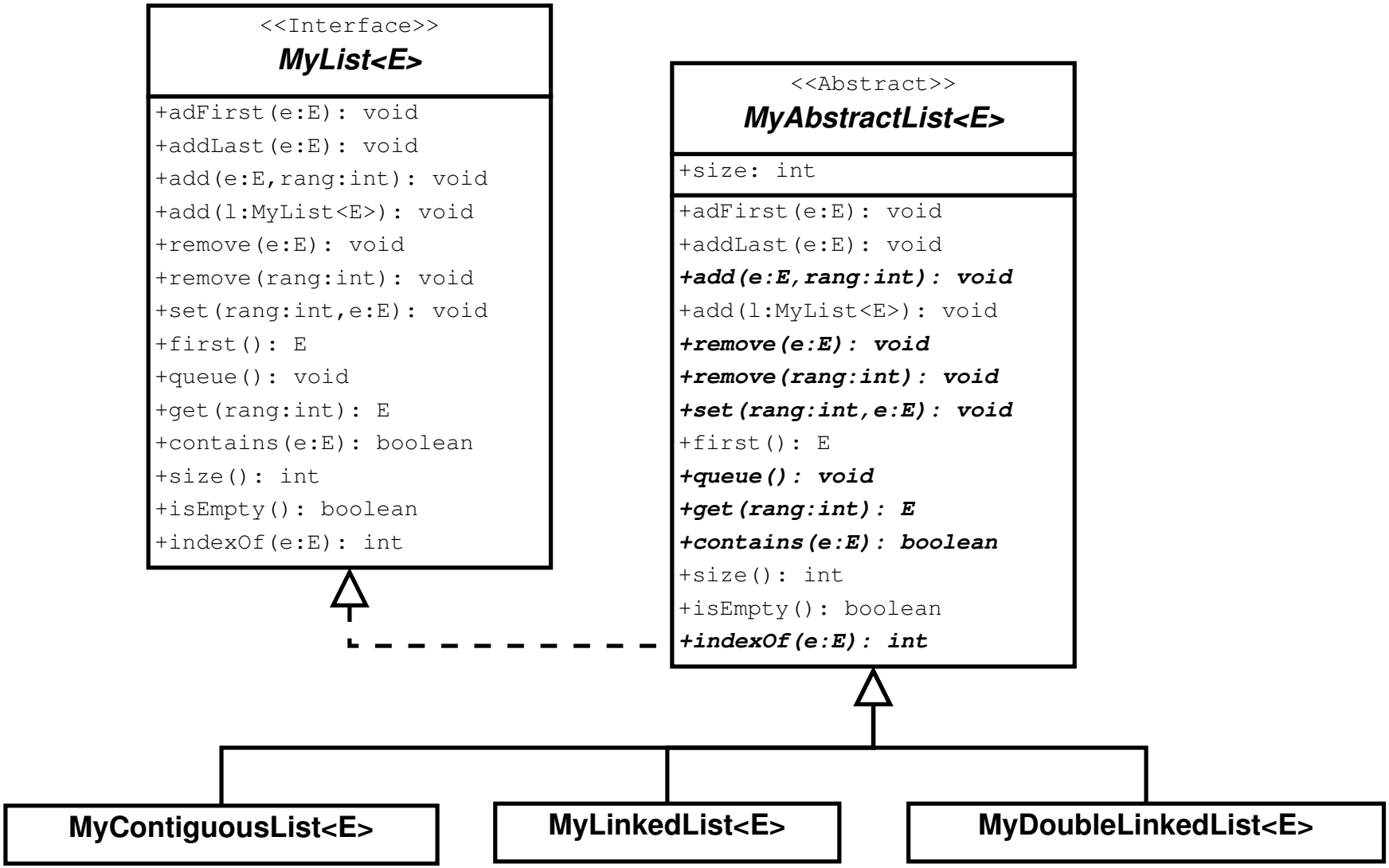
$$
indexOf(addFirst(l, e_1), e_2) =
$$

Axiomes avec *empty*() et *addFirst*()

$$
\begin{aligned}
get(empty(), e) &= \text{Violation de précondition !} \\
get(addFirst(l, e), i) &= \begin{cases} e & \text{si } i = 0 \\ get(l, i-1) & \text{sinon} \end{cases} \\
contains(empty(), e) &= faux \\
contains(addFirst(l, e_1), e_2) &= \begin{cases} vrai & \text{si } e_1 = e_2 \\ contains(l, e_2) & \text{sinon} \end{cases} \\
size(empty()) &= 0 \\
size(addFirst(l, e)) &= size(l) + 1 \\
isEmpty(empty()) &= vrai \\
isEmpty(addFirst(l, e)) &= faux \\
indexOf(empty(), e) &= -1 \text{ (élément non trouvé)} \\
indexOf(addFirst(l, e_1), e_2) &= \begin{cases} 0 & \text{si } e_1 = e_2 \\ indexOf(l, e_2) + 1 & \text{si } contains(l, e_2) \\ -1 & \text{sinon} \end{cases}
\end{aligned}
$$

Plan

Diagramme de classes



Exercice 2.C

Écrire la classe abstraite `MyAbstractList<E>` qui implante les fonctions :

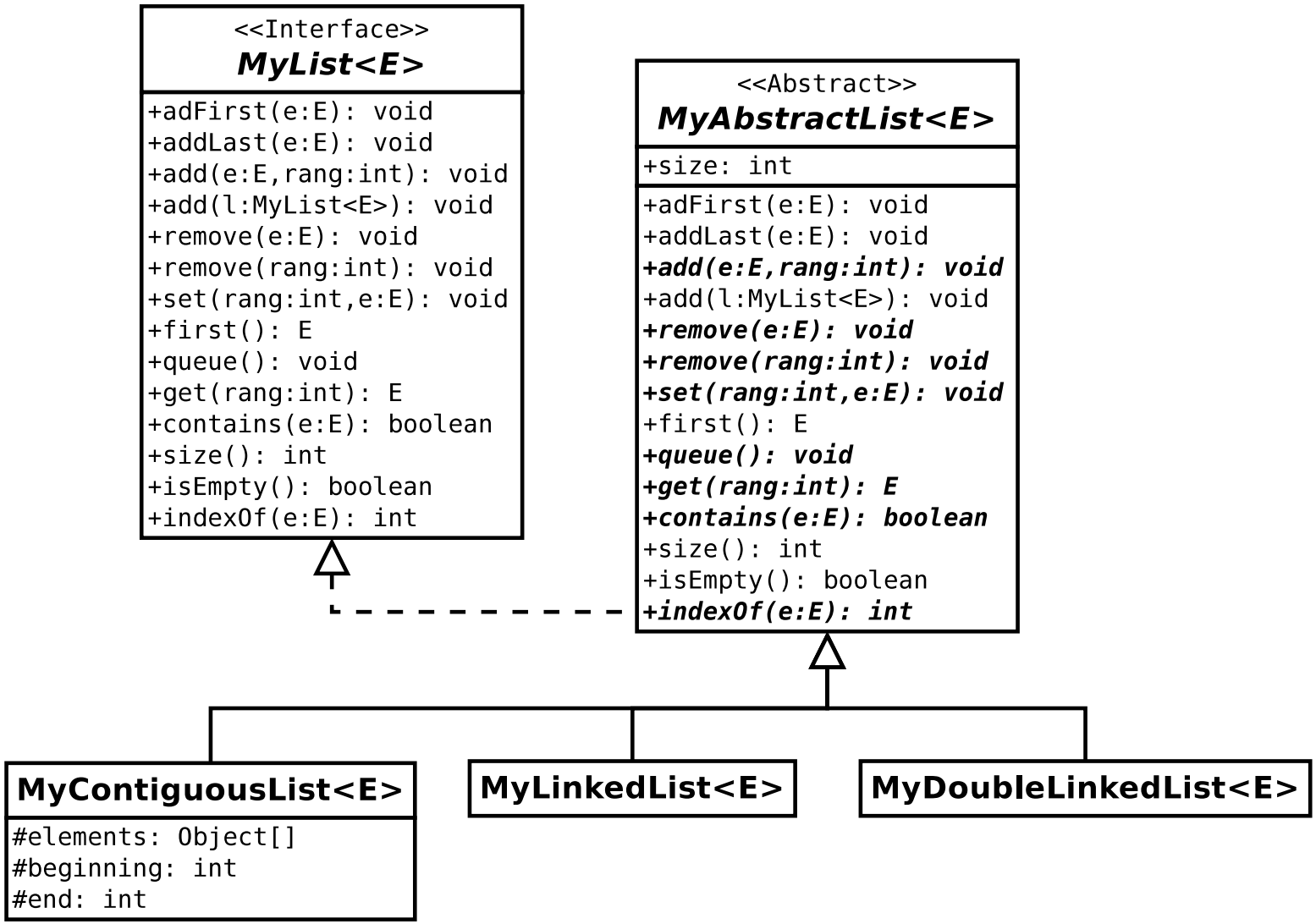
- `addFirst`
- `addLast`
- `add( MyList<E> l )`
- `size`
- `isEmpty`
- `first`

de l'interface `MyList<E>`. La taille de la liste est stockée grâce à l'attribut `int size;` dans la classe abstraite.

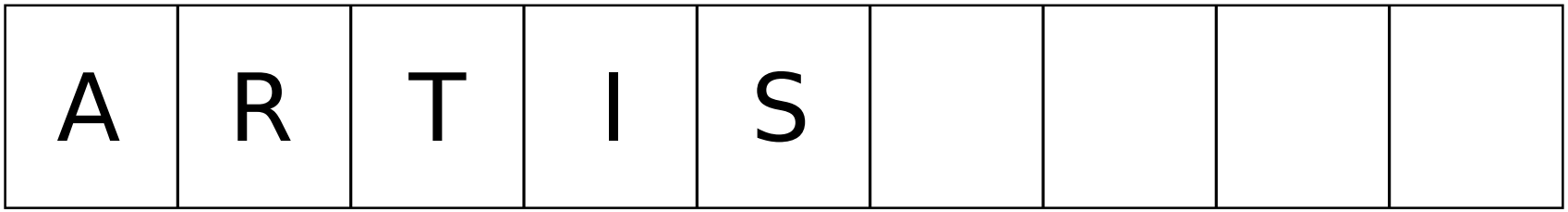
Solution

```
1  public abstract class MyAbstractList<E> implements MyList<E>
2   {
3       protected int size ;
4
5       public void addFirst( E e ) { this.add( e, 0 ); }
6       public void addLast( E e ) { this.add( e, size ); }
7       public E first() { return this.get( 0 ); }
8       public int size() { return size; }
9       public boolean isEmpty() { return size == 0 ; }
10
11      public abstract void add( E e, int rang );
12      public abstract void add( MyList<E> l );
13      public abstract void remove( E e );
14      public abstract void remove( int rang );
15      public abstract void set( int rang, E e );
16      public abstract MyList<E> queue();
17      public abstract E get( int rang );
18      public abstract boolean contains( E e );
19      public abstract int indexOf( E e );
20  }
```

Implantation contiguë



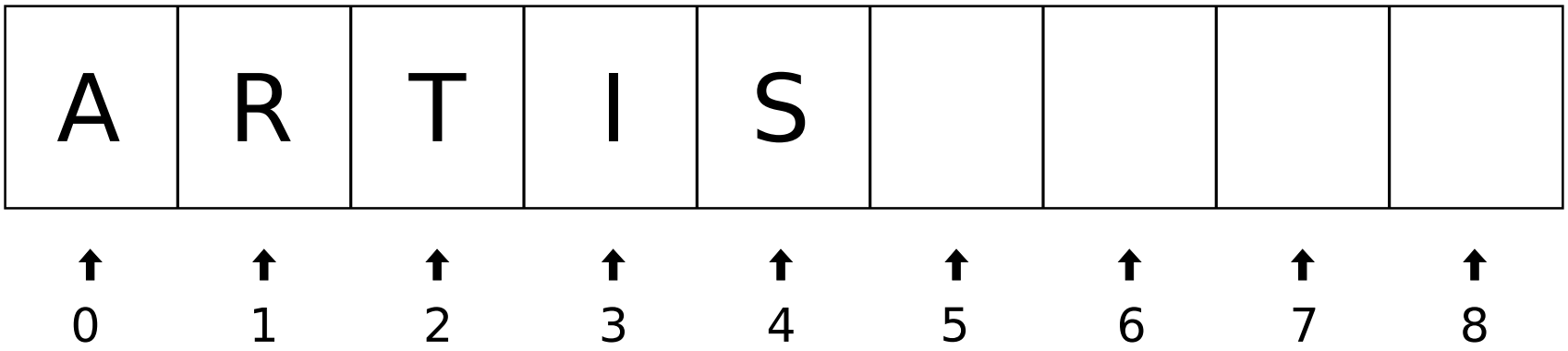
Présentation



Dans une implantation contiguë, les éléments sont rangés dans un tableau.

Deux solutions de stockage :

Présentation

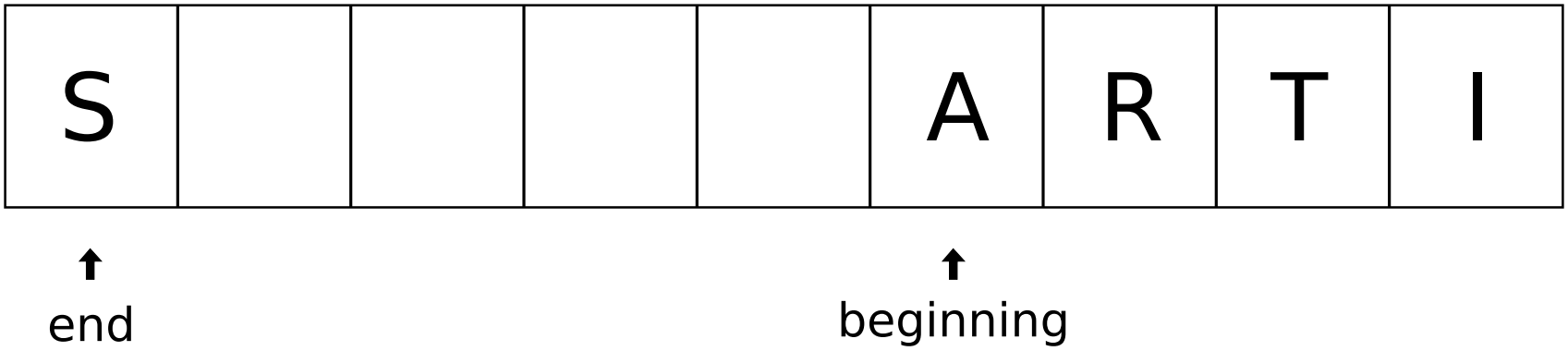


Dans une implantation contiguë, les éléments sont rangés dans un tableau.

Deux solutions de stockage :

- L'élément de rang N à la position N

Présentation



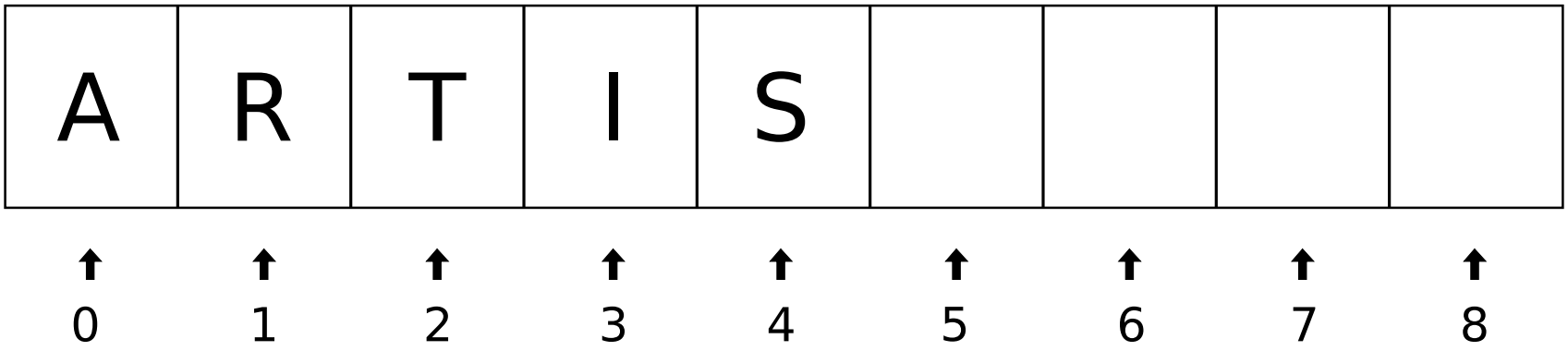
Dans une implantation contiguë, les éléments sont rangés dans un tableau.

Deux solutions de stockage :

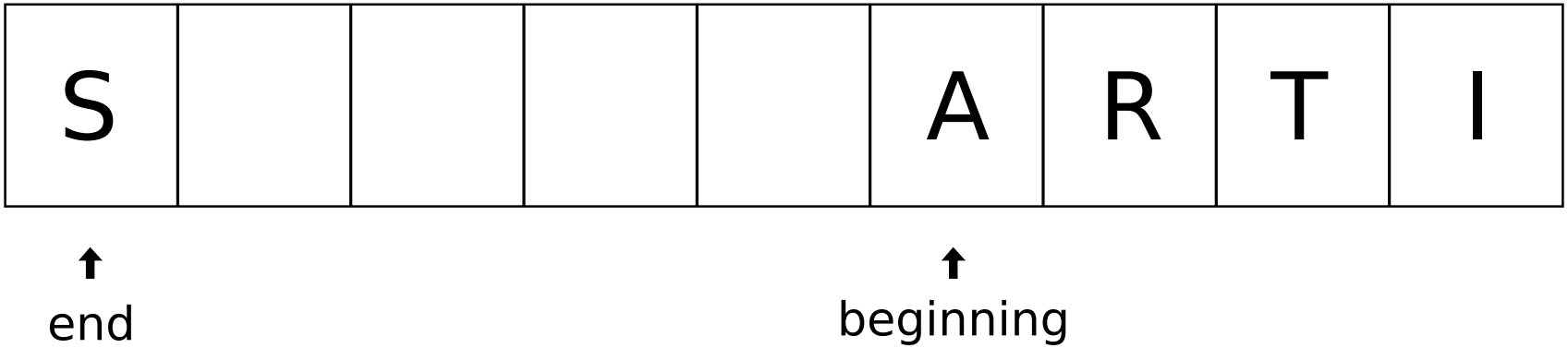
- L'élément de rang N à la position N
- Gestion circulaire

Comparaison indexée/circulaire : l'insertion en tête

Exemple : on souhaite insérer la lettre P au rang 0.



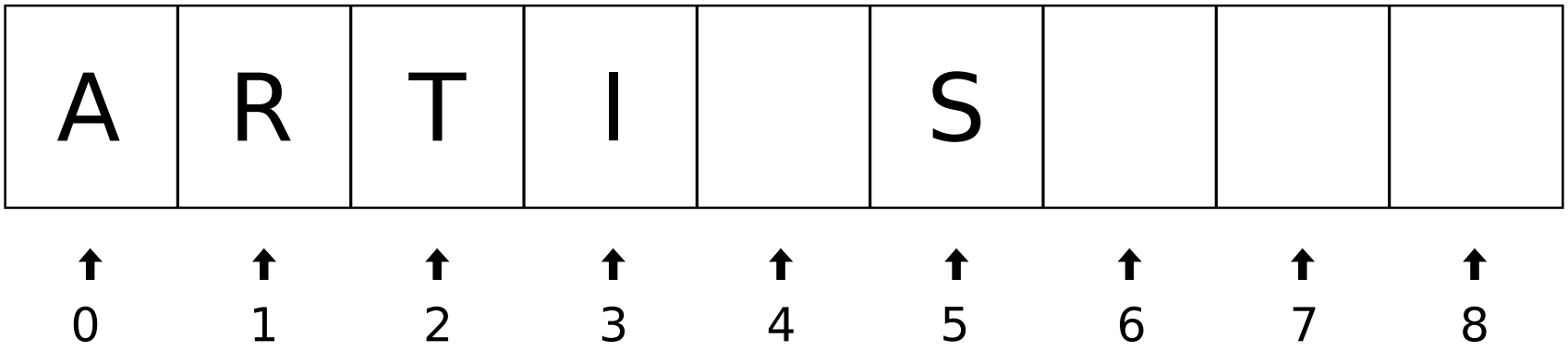
1. Avec l'élément de rang N à la position N.



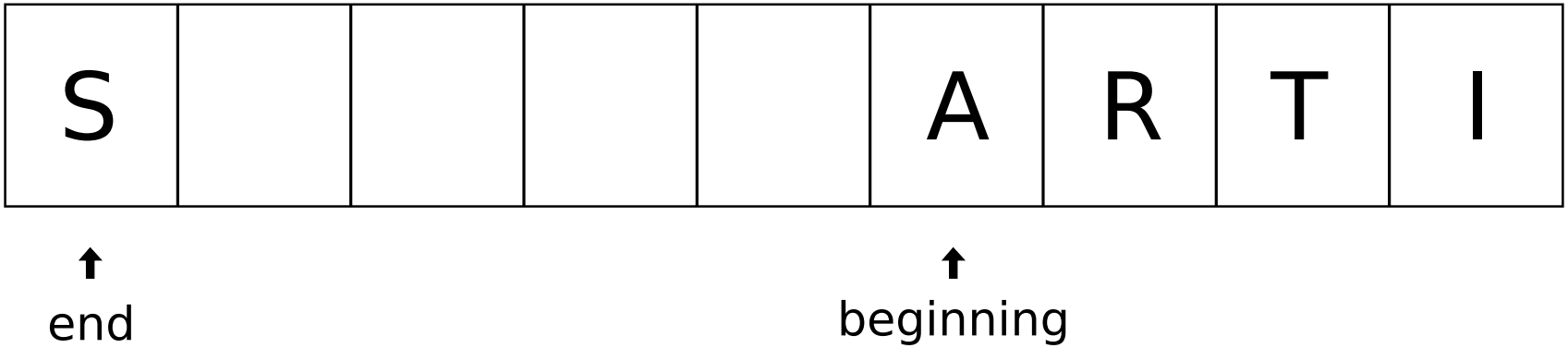
2. Avec gestion circulaire.

Comparaison indexée/circulaire : l'insertion en tête

Exemple : on souhaite insérer la lettre P au rang 0.



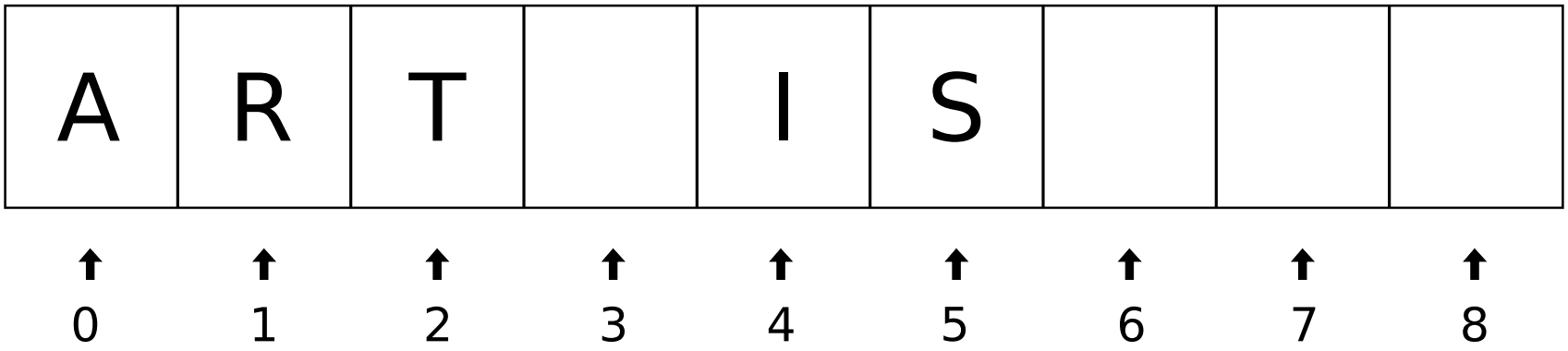
1. Avec l'élément de rang N à la position N.



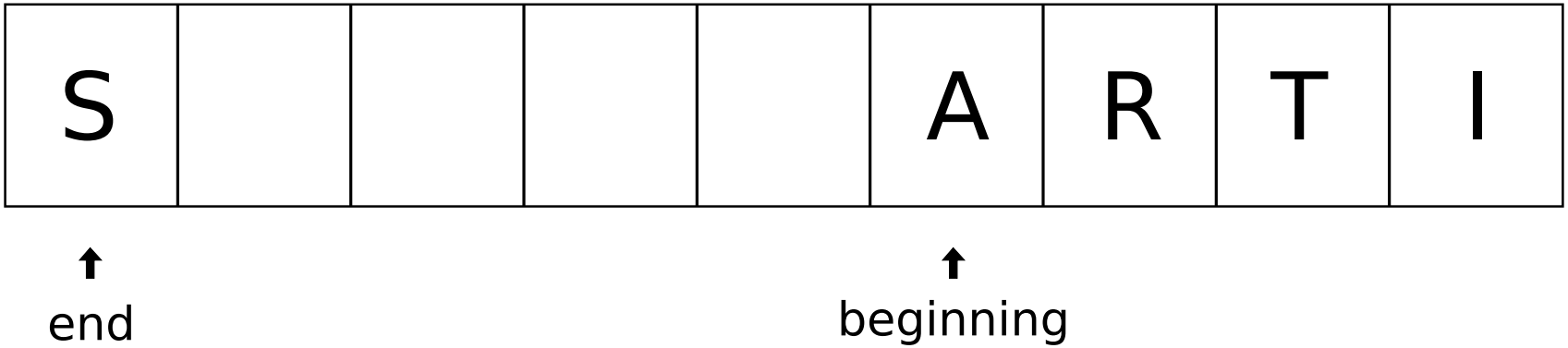
2. Avec gestion circulaire.

Comparaison indexée/circulaire : l'insertion en tête

Exemple : on souhaite insérer la lettre P au rang 0.



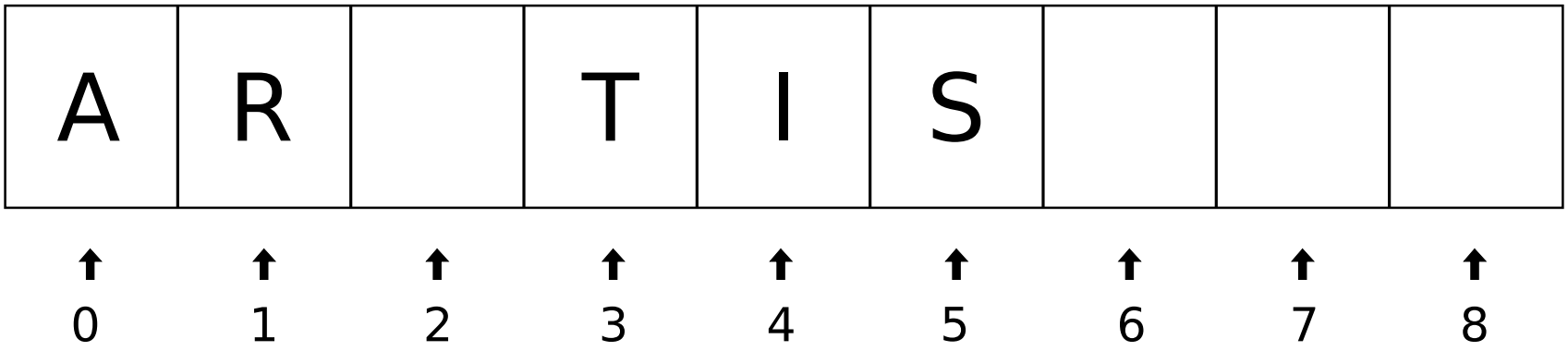
1. Avec l'élément de rang N à la position N.



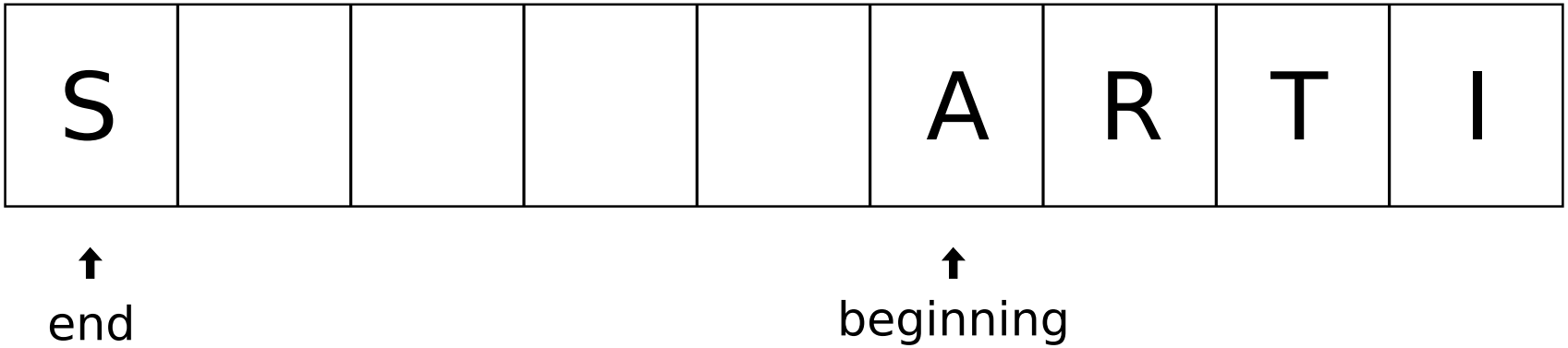
2. Avec gestion circulaire.

Comparaison indexée/circulaire : l'insertion en tête

Exemple : on souhaite insérer la lettre P au rang 0.



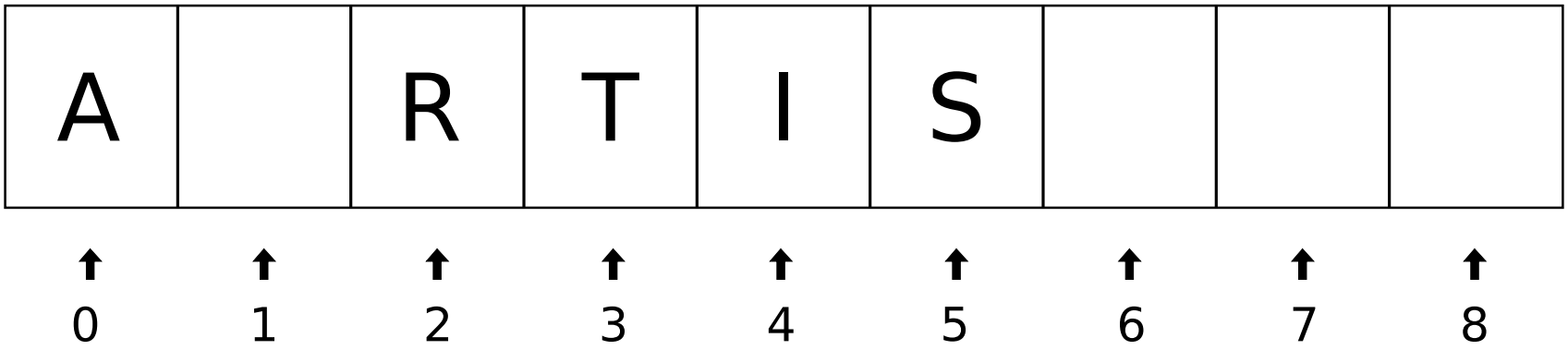
1. Avec l'élément de rang N à la position N.



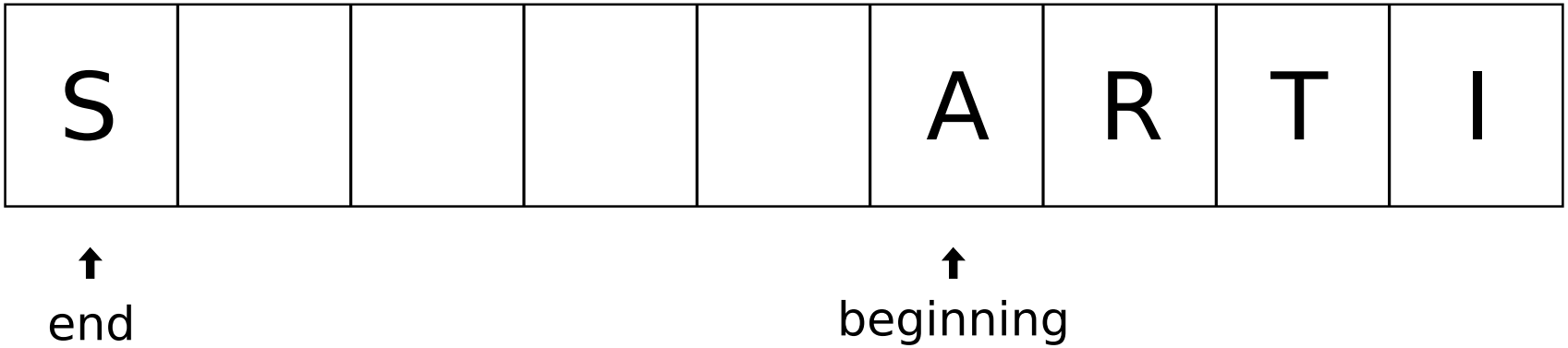
2. Avec gestion circulaire.

Comparaison indexée/circulaire : l'insertion en tête

Exemple : on souhaite insérer la lettre P au rang 0.



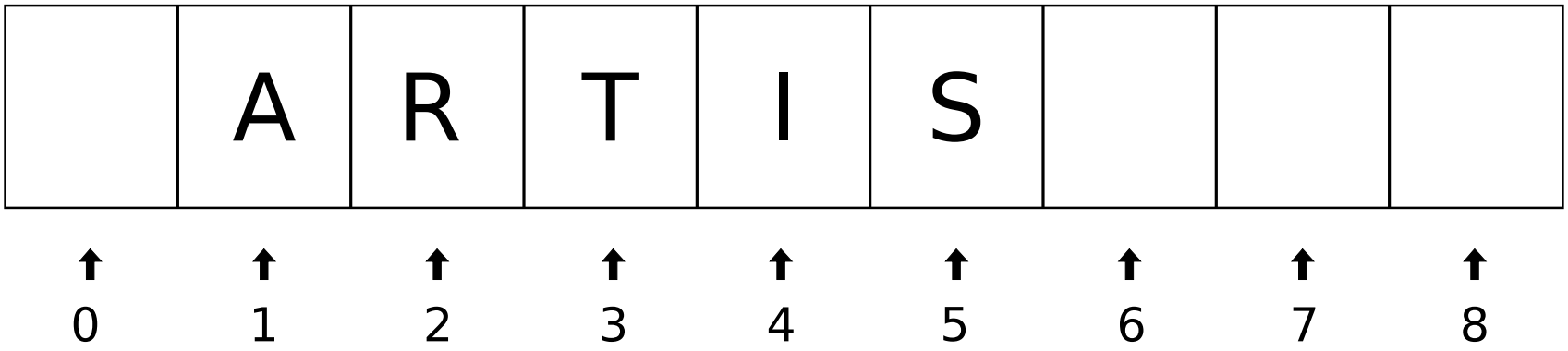
1. Avec l'élément de rang N à la position N.



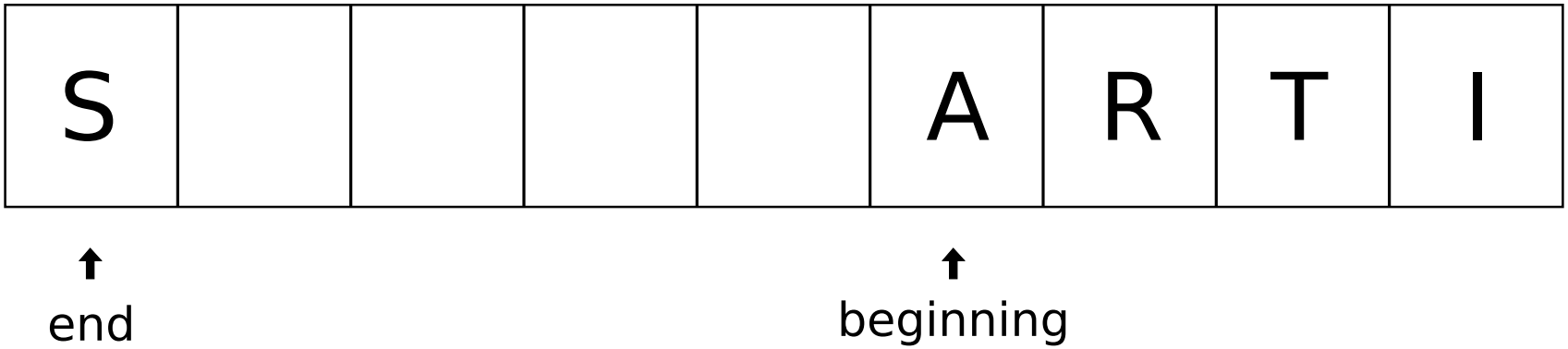
2. Avec gestion circulaire.

Comparaison indexée/circulaire : l'insertion en tête

Exemple : on souhaite insérer la lettre P au rang 0.



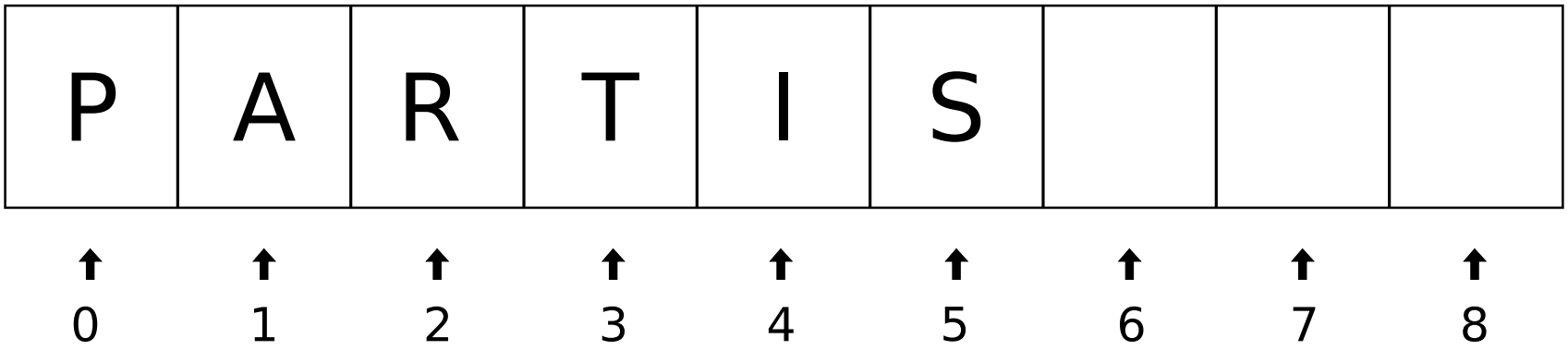
1. Avec l'élément de rang N à la position N.



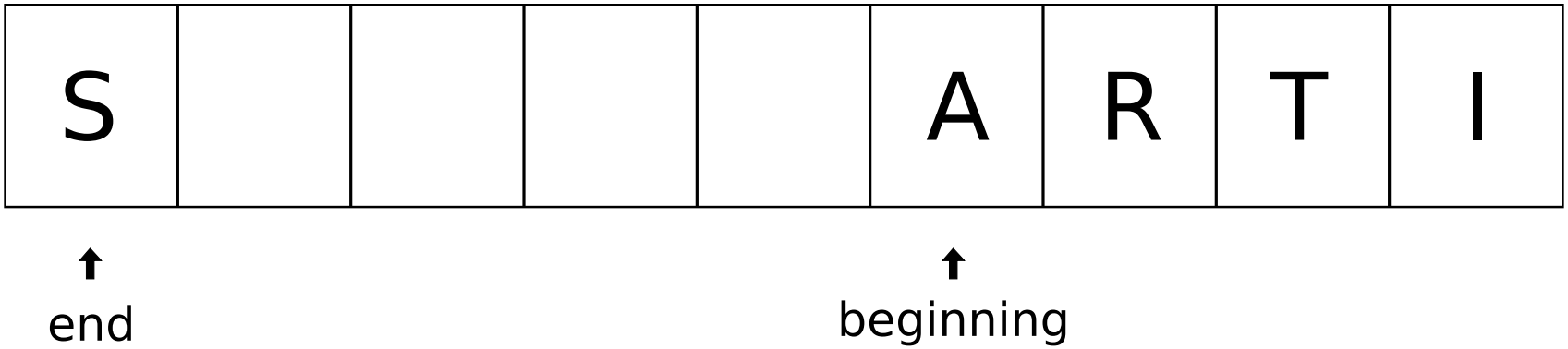
2. Avec gestion circulaire.

Comparaison indexée/circulaire : l'insertion en tête

Exemple : on souhaite insérer la lettre P au rang 0.



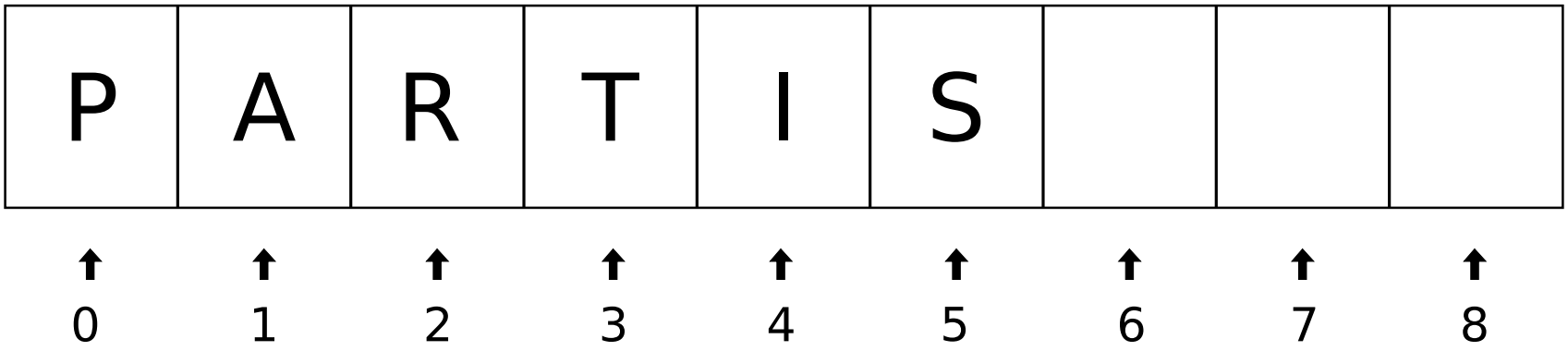
1. Avec l'élément de rang N à la position N.



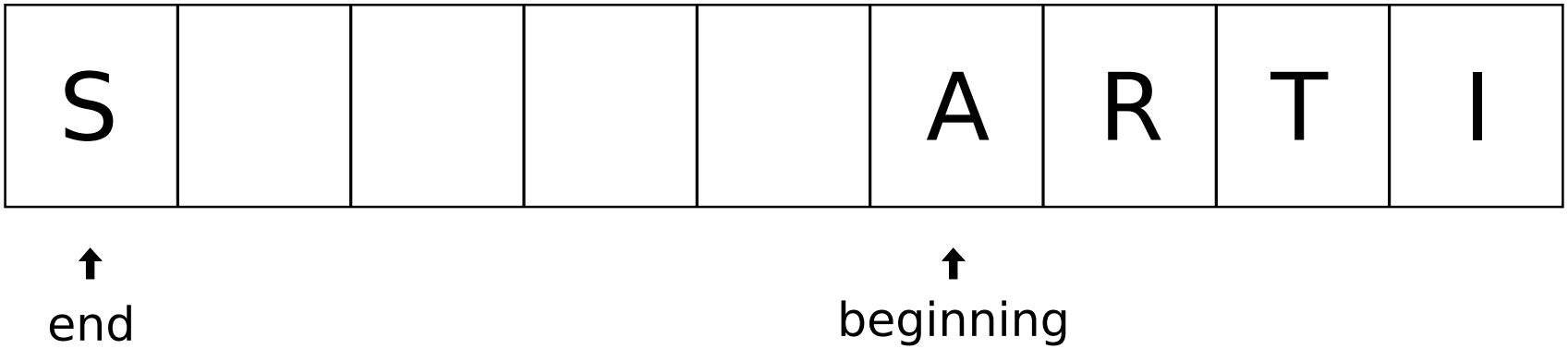
2. Avec gestion circulaire.

Comparaison indexée/circulaire : l'insertion en tête

Exemple : on souhaite insérer la lettre P au rang 0.



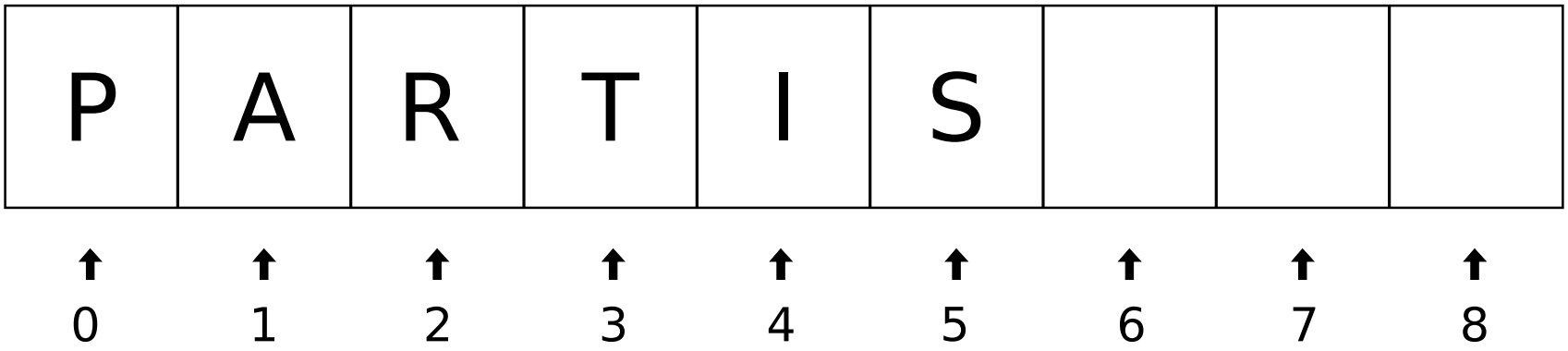
1. Avec l'élément de rang N à la position N. **5 décalages !**



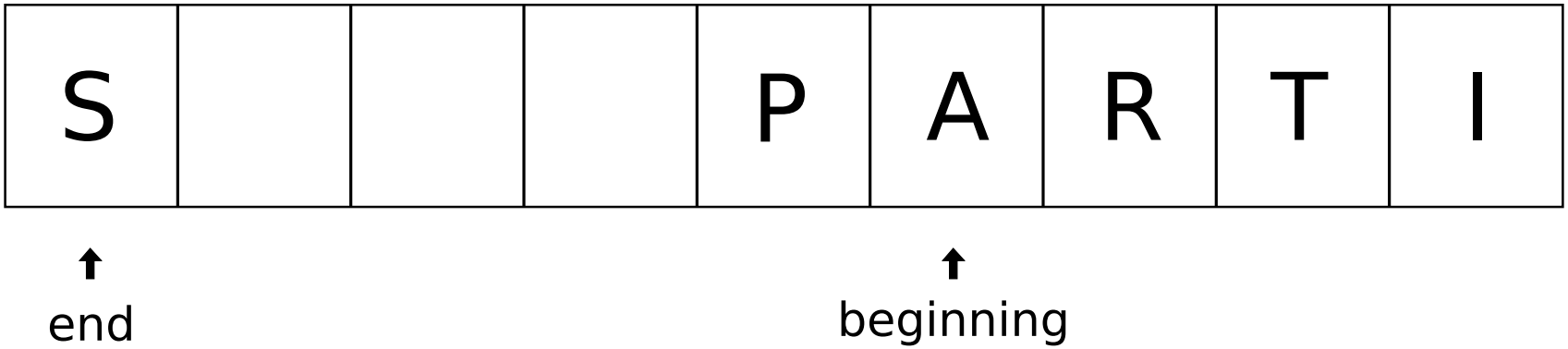
2. Avec gestion circulaire.

Comparaison indexée/circulaire : l'insertion en tête

Exemple : on souhaite insérer la lettre P au rang 0.



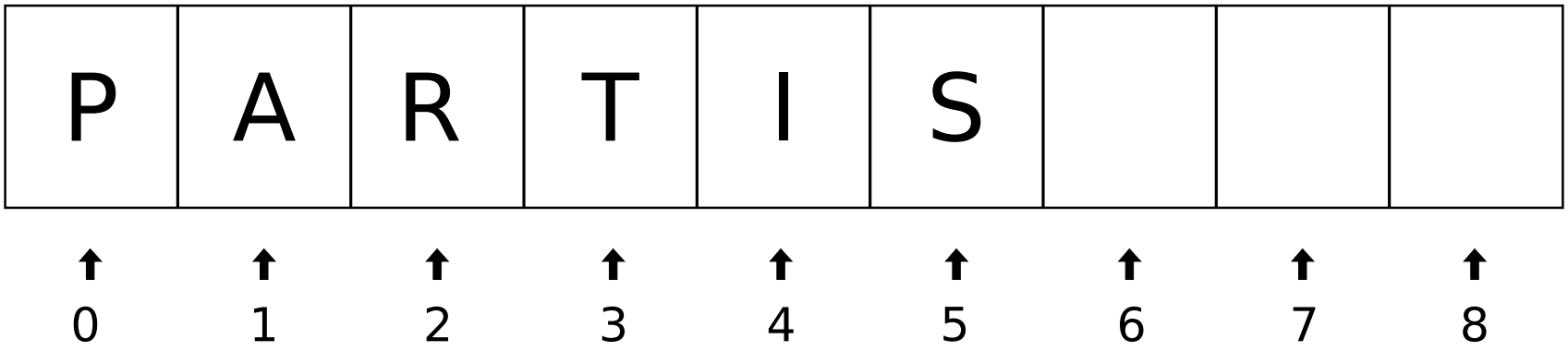
1. Avec l'élément de rang N à la position N. **5 décalages !**



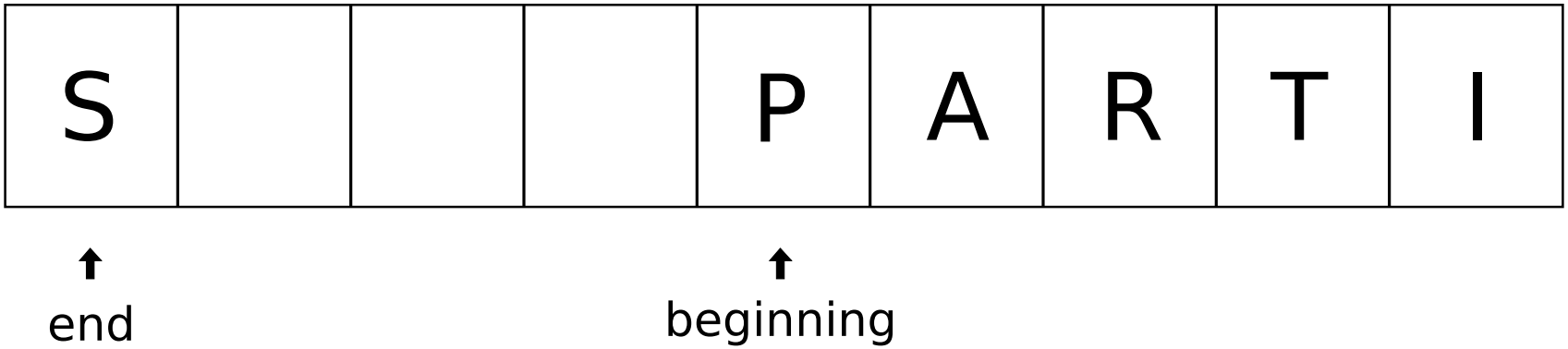
2. Avec gestion circulaire.

Comparaison indexée/circulaire : l'insertion en tête

Exemple : on souhaite insérer la lettre P au rang 0.



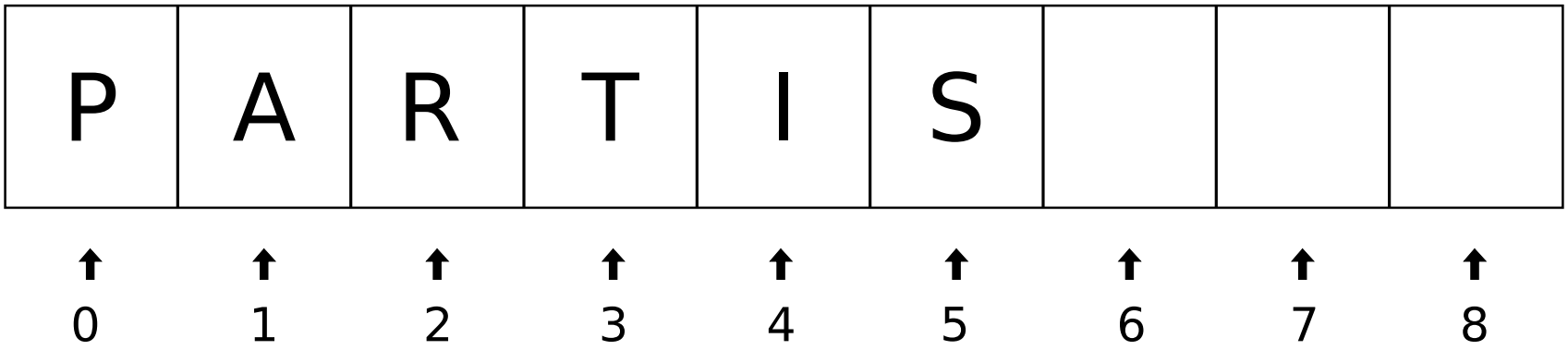
1. Avec l'élément de rang N à la position N. **5 décalages !**



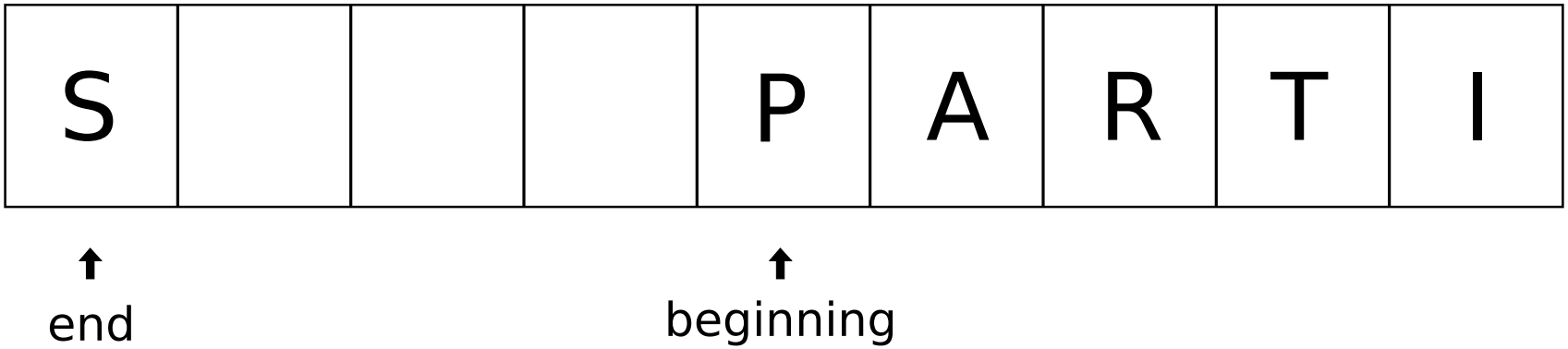
2. Avec gestion circulaire.

Comparaison indexée/circulaire : l'insertion en tête

Exemple : on souhaite insérer la lettre P au rang 0.



1. Avec l'élément de rang N à la position N. **5 décalages !**



2. Avec gestion circulaire. **0 décalage !**

Exercice 2.D

Écrire la classe `MyContiguousList<E>`, implantation **contiguë** de l'interface `MyList<E>` avec une **gestion circulaire**.

Cette implantation doit contenir une fonction `int position( int rang )` qui retourne la position dans le tableau de l'élément dont le rang est passé en argument.

La méthode `add` doit distinguer les quatre cas (à déterminer) rencontrés lors de l'ajout d'un élément.

Solution des attributs

```
1  public class MyContiguousList<E> extends MyAbstractList<E>
2  {
3      protected E[] elements ; // Gestion circulaire
4      protected int beginning;
5      protected int end;
6
7      //Constructeurs avec capacite et constructeur vide
8      ...
9  }
```

Solution du constructeur

```
1  public class MyContiguousList<E> extends MyAbstractList<E>
2  {
3      protected E[] elements ; // Gestion circulaire
4      protected int beginning;
5      protected int end;
6
7      public MyContiguousList( int capacity )
8      {
9          elements = new E[capacity];
10         beginning = capacity / 2;
11         end = beginning-1;
12         size = 0;
13     }
14
15     public MyContiguousList() { this(100); }
16
17     // Methode position
18     ...
19 }
```

Solution de la méthode *position*

```
1  public class MyContiguousList<E> extends MyAbstractList<E>
2  {
3      protected E[] elements ; // Gestion circulaire
4      protected int beginning;
5      protected int end;
6
7      ...
8
9      public int position( int rang )
10     {
11         return (beginning + rang) % elements.length;
12     }
13
14     // Methode add
15     ...
16 }
```

Solution de la méthode *add*

```
1  public class MyContiguousList<E> extends MyAbstractList<E>
2  {
3      protected E[] elements ; // Gestion circulaire
4      protected int beginning;
5      protected int end;
6      ...
7      public void add( E e , int rang ) {
8          ...
9          if ( rang<demiSize ) {
10             if ( beginning==0 ) { /* Cas 1 */ }
11             else { /* Cas 2 */ }
12         } else {
13             if ( end==elements.length-1 ) { /* Cas 3 */ }
14             else { /* Cas 4 */ }
15         }
16         ...
17     }
18 }
```

Solution de la méthode *add* : Cas 1

```
1  public class MyContiguousList<E> extends MyAbstractList<E>
2  {
3      protected E[] elements ; // Gestion circulaire
4      protected int beginning;
5      protected int end;
6      ...
7      public void add( E e , int rang ) {
8          ...
9          if ( rang<demiSize ) {
10             if ( beginning==0 ) { /* Cas 1 */
11                 elements[elements.length-1] = elements[0];
12                 for (int k=0 ; k<position(rang)-1 ; k++) {
13                     elements[k] = elements[k+1];
14                 }
15                 beginning = elements.length - 1;
16             } else { /* Cas 2 */ }
17         } else { /* Cas 3 et 4 */ }
18         ...
19     }
20 }
```

Solution de la méthode *add* : Cas 2

```
1  public class MyContiguousList<E> extends MyAbstractList<E>
2  {
3      protected E[] elements ; // Gestion circulaire
4      protected int beginning;
5      protected int end;
6      ...
7      public void add( E e , int rang ) {
8          ...
9          if ( rang<demiSize ) {
10             if ( beginning==0 ) { /* Cas 1 */ }
11             else {/* Cas 2 (si position(rang)>beginning) */
12                 for (int k=beginning-1 ;
13                                 k<position(rang)-1 ; k++) {
14                     elements[k] = elements[k+1];
15                 }
16                 beginning--;
17             }
18         } else { /* Cas 3 et 4 */ }
19         ...
20     }
21 }
```

Solution de la méthode *add* : Cas 3

```
1  public class MyContiguousList<E> extends MyAbstractList<E>
2  {
3      protected E[] elements ; // Gestion circulaire
4      protected int beginning;
5      protected int end;
6      ...
7      public void add( E e , int rang ) {
8          ...
9          if ( rang<demiSize ) { /* Cas 1 et 2 */ }
10         } else {
11             if ( end==elements.length-1 ) { /* Cas 3 */
12                 elements[0] = elements[elements.length-1];
13                 for (int k=elements.length-1 ;
14                                 k>position(rang)+1 ; k--) {
15                     elements[k] = elements[k-1];
16                 }
17                 end = 0;
18             } else { /* Cas 4 */ }
19         }
20         ...
21 }
```

Solution de la méthode *add* : Cas 4

```
1  public class MyContiguousList<E> extends MyAbstractList<E>
2  {
3      protected E[] elements ; // Gestion circulaire
4      protected int beginning;
5      protected int end;
6      ...
7      public void add( E e , int rang ) {
8          ...
9          if ( rang<demiSize ) { /* Cas 1 et 2 */ }
10         } else {
11             if ( end==elements.length-1 ) { /* Cas 3 */ }
12             else { /* Cas 4 (si position(rang)<end) */
13                 for (int k=end+1 ;
14                                 k>position(rang)+1 ; k--) {
15                     elements[k] = elements[k-1];
16                 }
17                 end++;
18             }
19         }
20         ...
21 }
```

Solution “pré” et “post” traitements

```
1  public class MyContiguousList<E> extends MyAbstractList<E>
2  {
3      protected E[] elements ; // Gestion circulaire
4      protected int beginning;
5      protected int end;
6      ...
7      public void add( E e , int rang ) {
8          // Pre-traitements
9          if ( rang<demiSize ) {
10             ...
11         } else {
12             ...
13         }
14         // Post-traitements
15     }
16 }
```

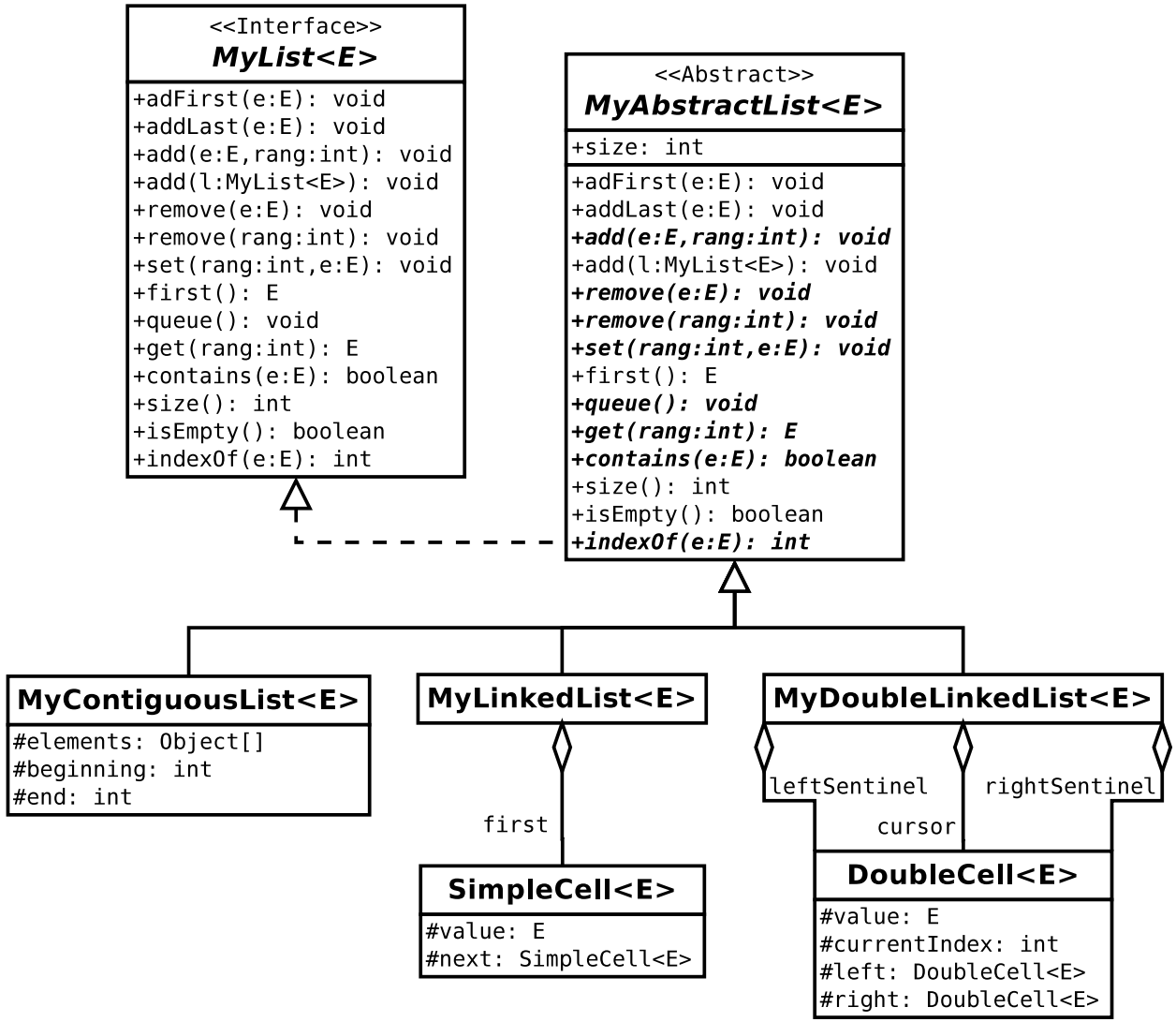
Solution pré-traitements

```
1  public class MyContiguousList<E> extends MyAbstractList<E>
2  {
3      protected E[] elements ; // Gestion circulaire
4      protected int beginning;
5      protected int end;
6      ...
7      public void add( E e , int rang ) {
8          // Pre-traitements
9          int demiSize = size / 2;
10         if ( rang<demiSize ) {
11             ...
12         } else {
13             ...
14         }
15         // Post-traitements
16     }
17 }
```

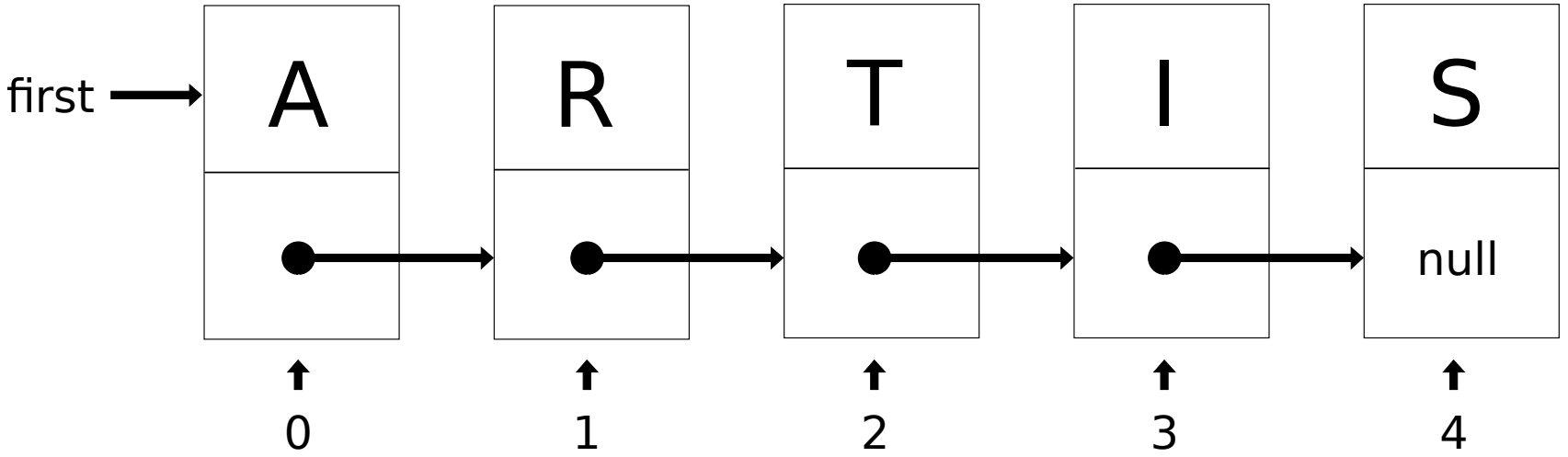
Solution post-traitements

```
1  public class MyContiguousList<E> extends MyAbstractList<E>
2  {
3      protected E[] elements ; // Gestion circulaire
4      protected int beginning;
5      protected int end;
6      ...
7      public void add( E e , int rang ) {
8          // Pre-traitements
9          int demiSize = size / 2;
10         if ( rang<demiSize ) {
11             ...
12         } else {
13             ...
14         }
15         // Post-traitements
16         elements[position(rang)] = e;
17         size++;
18     }
19 }
```

MyLinkedList<E> - Diagramme de classe



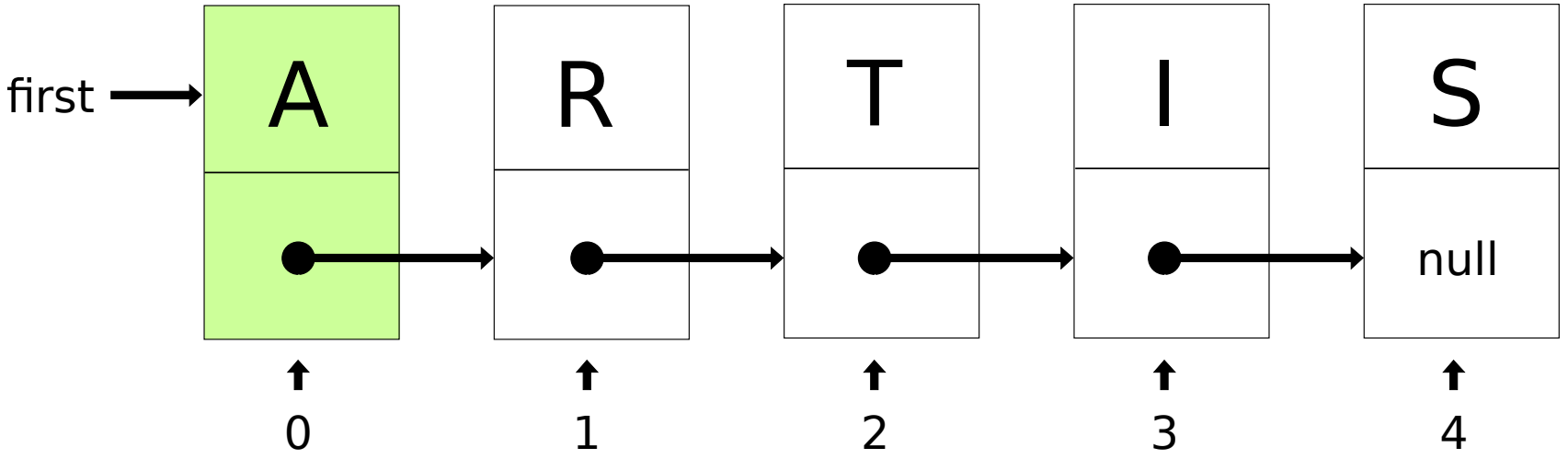
MyLinkedList<E> - Présentation



Dans une liste simplement chainee, les éléments connaissent leur suivant.

Un peu de vocabulaire :

MyLinkedList<E> - Présentation

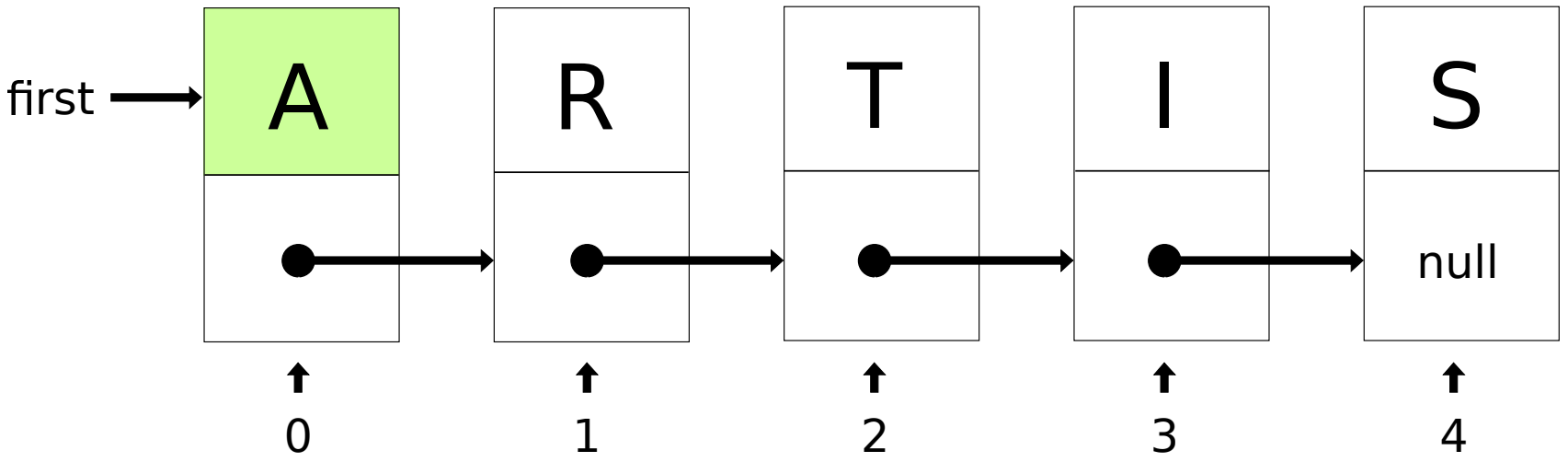


Dans une liste simplement chainee, les éléments connaissent leur suivant.

Un peu de vocabulaire :

- Une **cellule**...

MyLinkedList<E> - Présentation

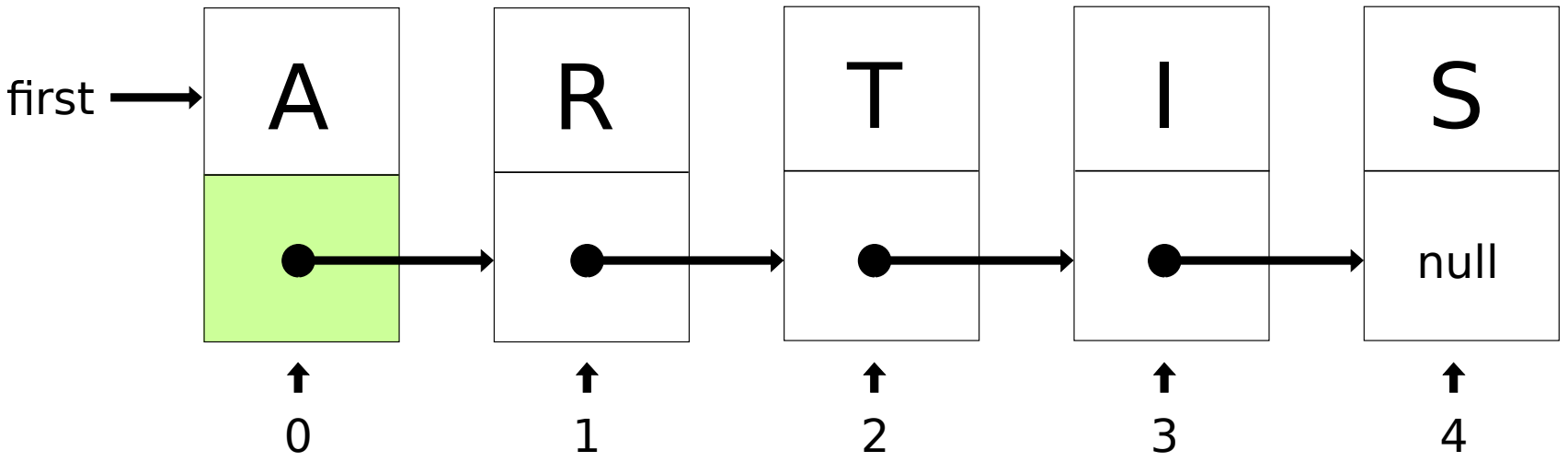


Dans une liste simplement chainee, les éléments connaissent leur suivant.

Un peu de vocabulaire :

- Une **cellule**...
- ... contient une **valeur**...

MyLinkedList<E> - Présentation



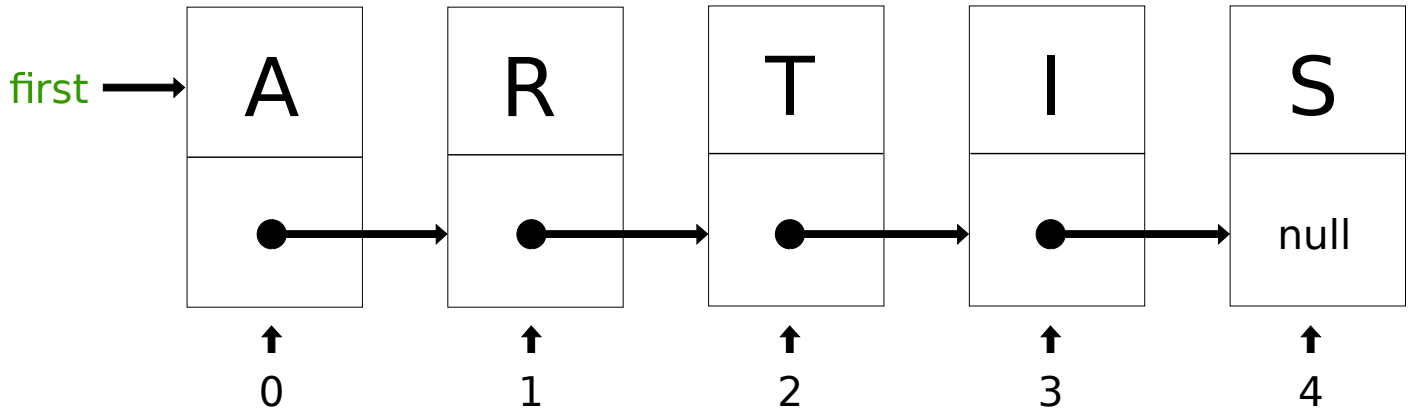
Dans une liste simplement chainee, les éléments connaissent leur suivant.

Un peu de vocabulaire :

- Une **cellule**...
- ... contient une **valeur**...
- ... et l'élément **suivant**.

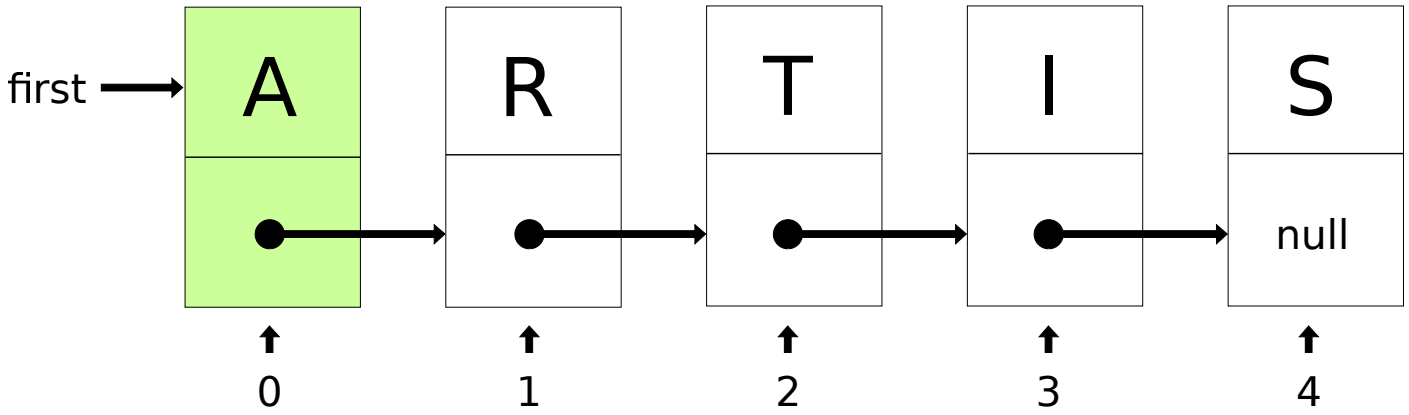
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



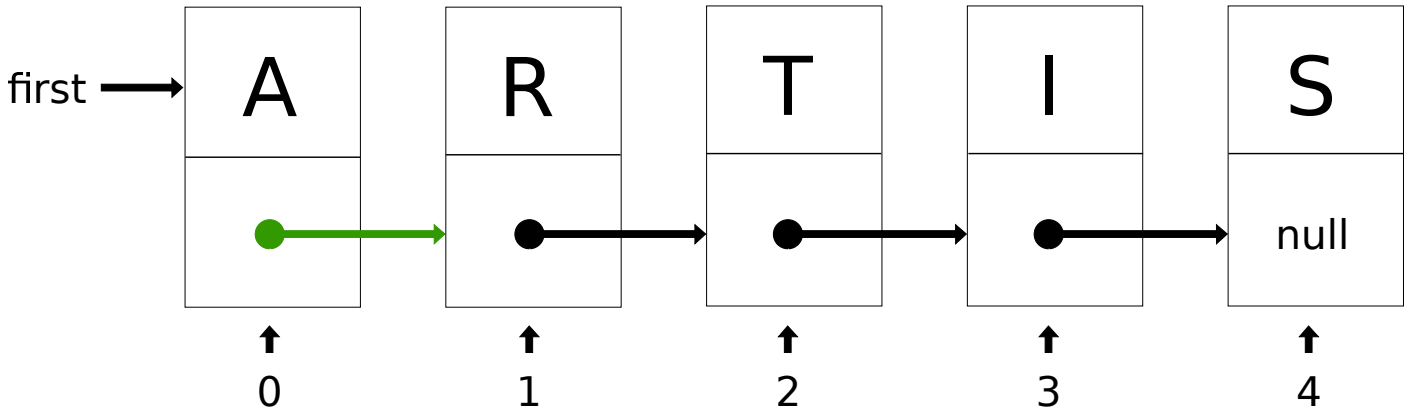
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



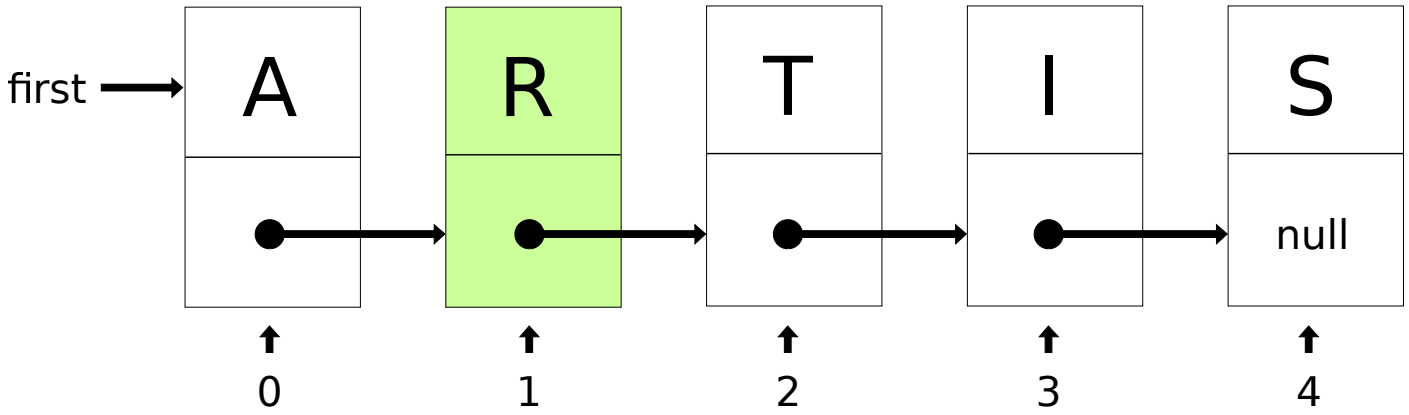
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



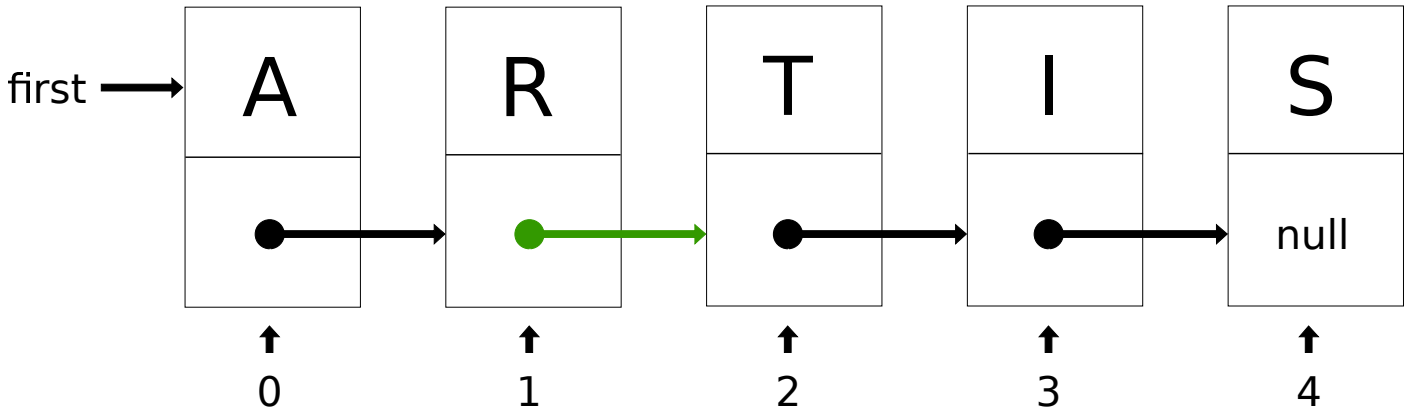
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



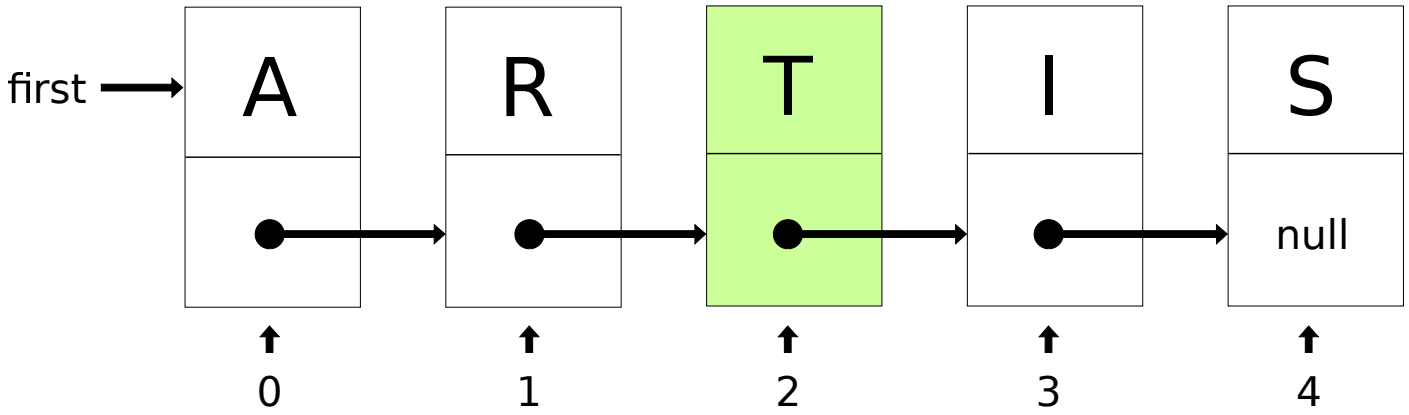
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



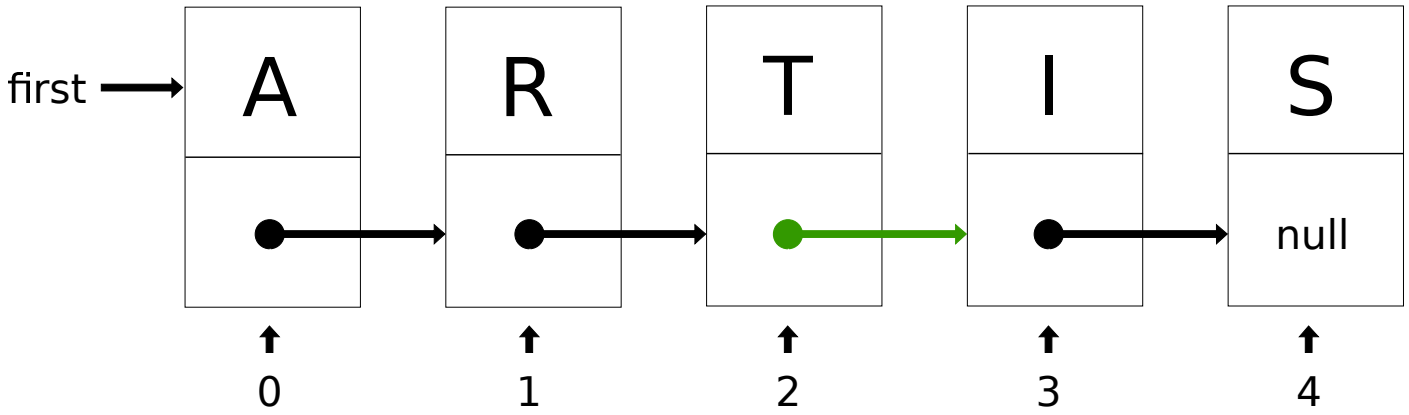
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



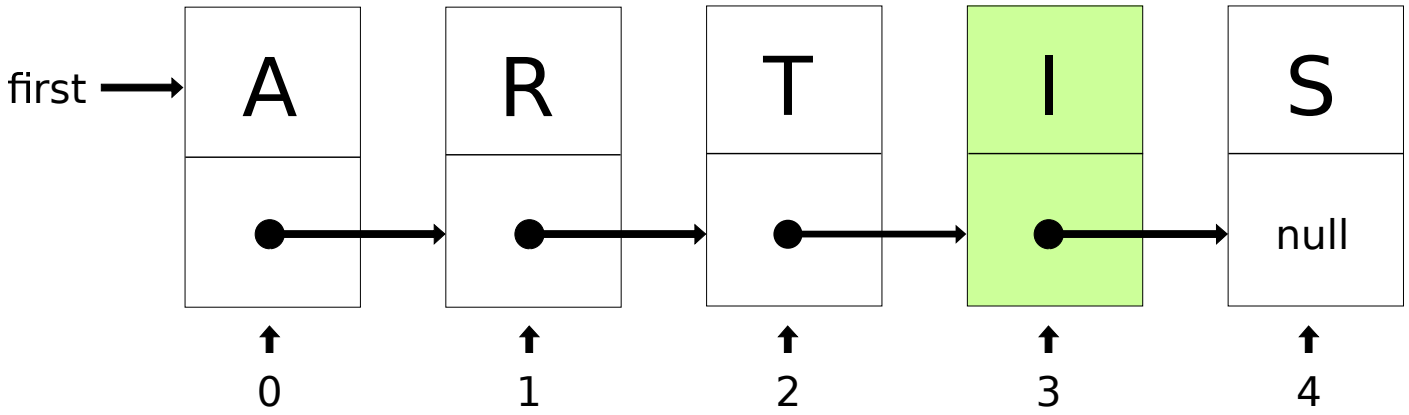
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



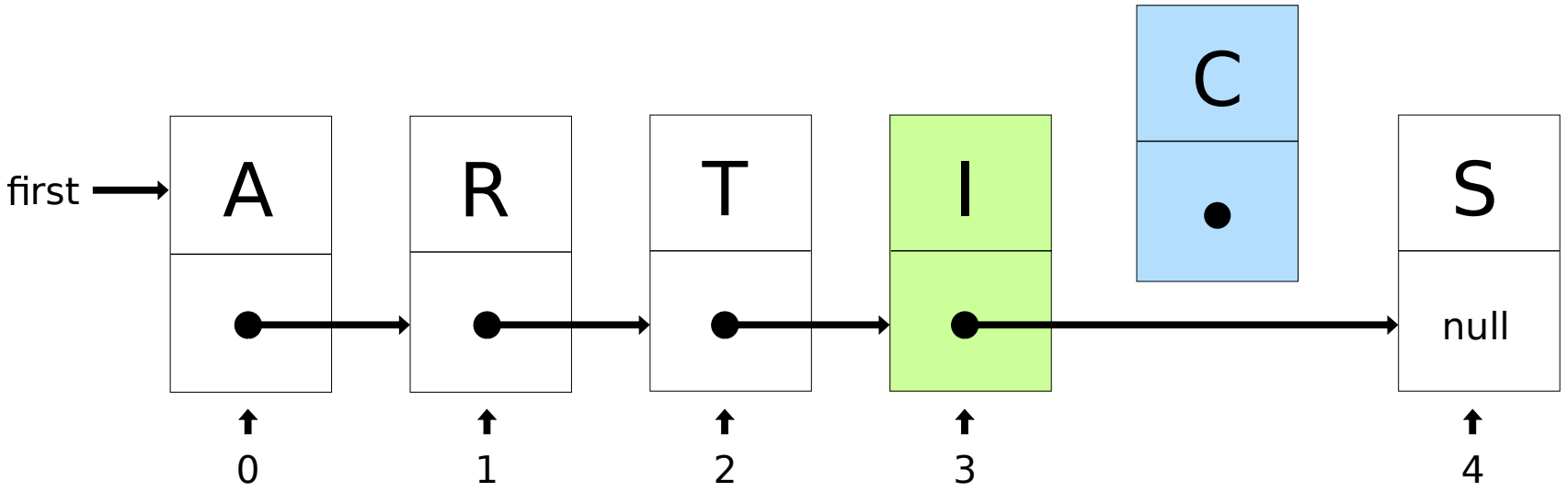
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



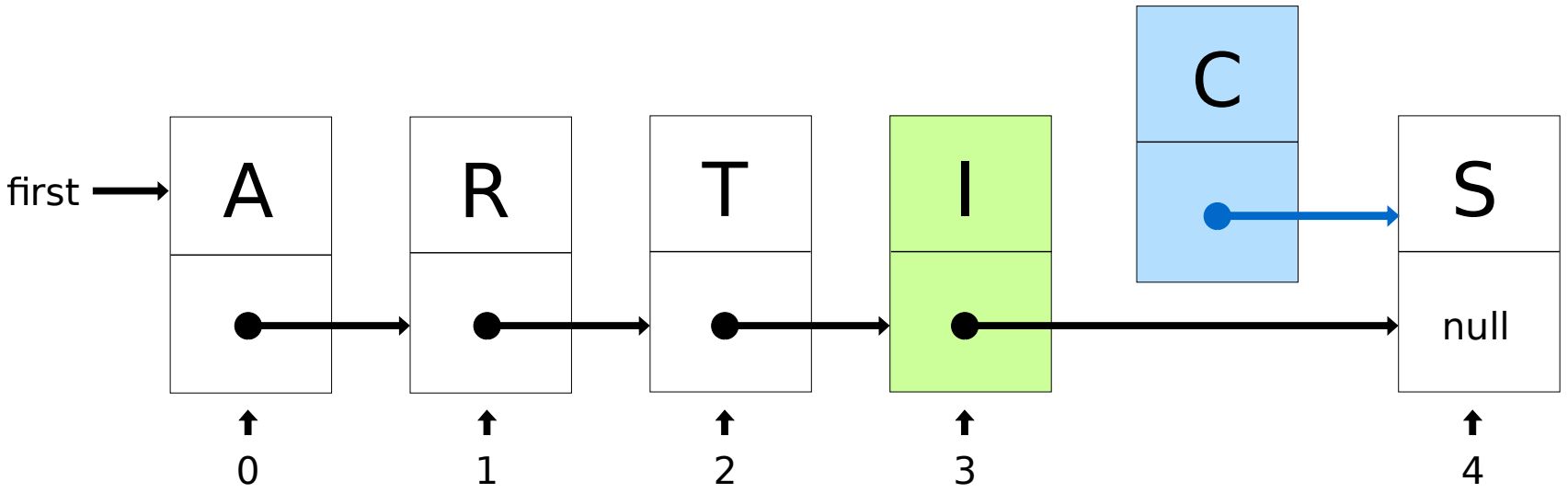
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



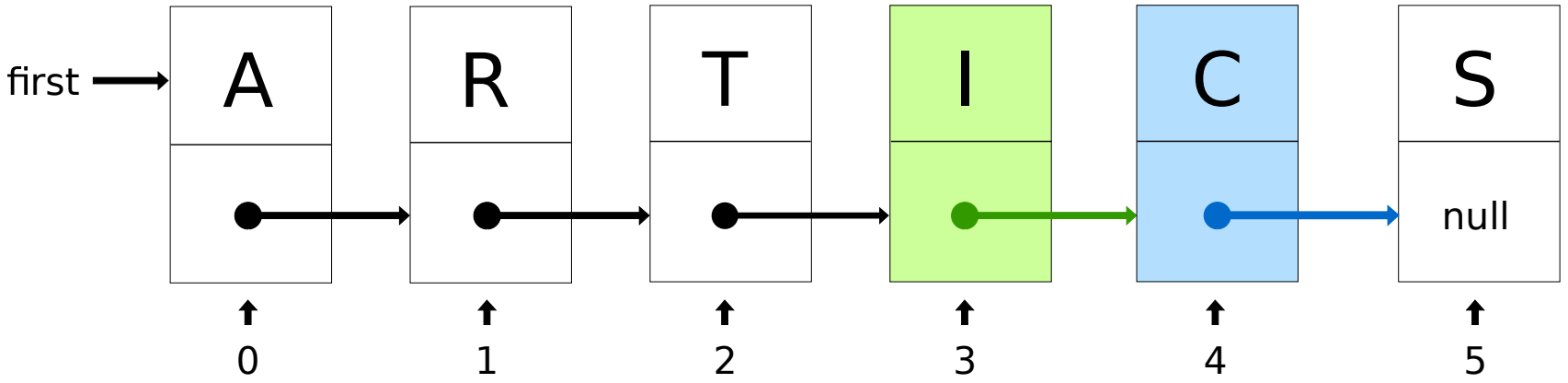
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



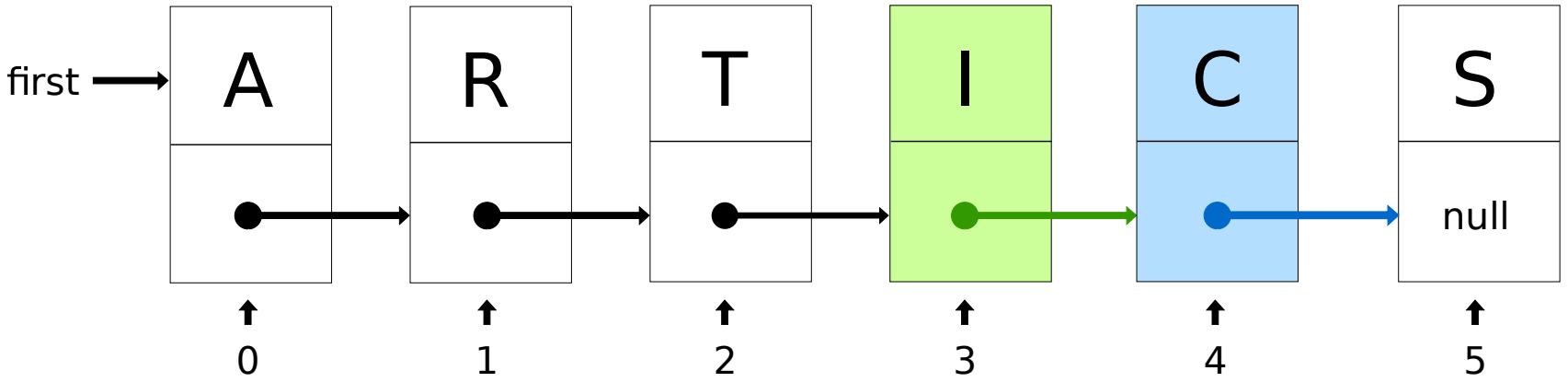
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



4 déplacements !

Exercice 2.D

Soit la classe `SimpleCell<E>`, représentant une cellule d'une liste chainée simple.

Complétez la classe `MyLinkedList<E>` en implantant les fonction d'ajout et de suppression d'un élément.

Classe SimpleCell<E>

```
1  public class SimpleCell<E>
2  {
3      protected E value ;
4      protected SimpleCell<E> next;
5
6      public SimpleCell( SimpleCell<E> next, E value ) {
7          this.value = value;
8          this.next = next;
9      }
10
11     public E getValue() { return value ; }
12     public SimpleCell<E> getNext() { return next; }
13
14     public void setValue( E value ) { this.value = value; }
15     public SimpleCell<E> setNext( SimpleCell<E> next ) {
16         this.next = next ;
17     }
18 }
```

Classe MyLinkedList<E>

```
1  public class MyLinkedList<E> extends MyAbstractList<E>
2  {
3      protected SimpleCell<E> first;
4
5      public MyLinkedList() { size = 0; }
6
7      public void add( E e, int rang )
8      {
9          // À compléter
10     }
11
12     public void remove( int rang )
13     {
14         // À compléter
15     }
16
17     ...
18 }
```

Solution de add

```
1  public class MyLinkedList<E> extends MyAbstractList<E>
2  {
3      protected SimpleCell<E> first;
4      ...
5      public void add( E e, int rang )
6      {
7          SimpleCell<E> nouveau = new SimpleCell<E>(null,e);
8          if ( rang == 0 ) { ... }
9          else if ( rang == size() ) { ... }
10         else { ... }
11     }
12     ...
13 }
```

Solution de add

```
1  public class MyLinkedList<E> extends MyAbstractList<E>
2  {
3      protected SimpleCell<E> first;
4      ...
5      public void add( E e, int rang )
6      {
7          SimpleCell<E> nouveau = new SimpleCell<E>(null,e);
8          if ( rang == 0 )
9          {
10             nouveau.setNext( first );
11             first = nouveau;
12         }
13         else if ( rang == size() ) { ... }
14         else { ... }
15     }
16     ...
17 }
```

Solution de add

```
1  public class MyLinkedList<E> extends MyAbstractList<E>
2  {
3      protected SimpleCell<E> first;
4      ...
5      public void add( E e, int rang )
6      {
7          SimpleCell<E> nouveau = new SimpleCell<E>(null,e);
8          if ( rang == 0 ) { ... }
9          else if ( rang == size() )
10         {
11             getCell(size()-1).setNext(nouveau);
12         }
13         else { ... }
14     }
15     ...
16 }
```

Solution de getCell

```
1  public class MyLinkedList<E> extends MyAbstractList<E>
2  {
3      protected SimpleCell<E> first;
4      ...
5      public SimpleCell<E> getCell( int rang )
6      {
7          SimpleCell<E> courant = first;
8          int rangCourant = 0;
9          while ( rangCourant != rang )
10         {
11             courant = courant.getNext();
12             rangCourant++;
13         }
14         return courant;
15     }
16     ...
17 }
```

Solution de add (suite)

```
1  public class MyLinkedList<E> extends MyAbstractList<E>
2  {
3      protected SimpleCell<E> first;
4      ...
5      public void add( E e, int rang )
6      {
7          SimpleCell<E> nouveau = new SimpleCell<E>(null,e);
8          if ( rang == 0 ) { ... }
9          else if ( rang == size() ) { ... }
10         else
11         {
12             SimpleCell<E> courant = getCell(rang-1);
13             nouveau.setNext(courant.getNext());
14             courant.setNext(nouveau);
15         }
16     }
17     ...
18 }
```

Solution de add (suite)

```
1  public class MyLinkedList<E> extends MyAbstractList<E>
2  {
3      protected SimpleCell<E> first;
4      ...
5      public void add( E e, int rang )
6      {
7          SimpleCell<E> nouveau = new SimpleCell<E>(null,e);
8          if ( rang == 0 ) { ... }
9          else if ( rang == size() ) { ... }
10         else { ... }
11         size++;
12     }
13     ...
14 }
```

Classe MyLinkedList<E>

```
1  public class MyLinkedList<E> extends MyAbstractList<E>
2  {
3      protected SimpleCell<E> first;
4
5      public MyLinkedList() { size = 0; }
6
7      public void add( E e, int rang )
8      {
9          // Ok !
10     }
11
12     public void remove( int rang )
13     {
14         // À compléter
15     }
16
17     ...
18 }
```

Solution de remove

```
1  public class MyLinkedList<E> extends MyAbstractList<E>
2  {
3      protected SimpleCell<E> first;
4      ...
5      public void remove( int rang )
6      {
7          if ( rang == 0 ) { ... }
8          else if ( rang == size()-1 ) { ... }
9          else { ... }
10     }
11     ...
12 }
```

Solution de remove

```
1  public class MyLinkedList<E> extends MyAbstractList<E>
2  {
3      protected SimpleCell<E> first;
4      ...
5      public void remove( int rang )
6      {
7          if ( rang == 0 )
8          {
9              first = first.getNext();
10         }
11         else if ( rang == size()-1 ) { ... }
12         else { ... }
13     }
14     ...
15 }
```

Solution de remove

```
1  public class MyLinkedList<E> extends MyAbstractList<E>
2  {
3      protected SimpleCell<E> first;
4      ...
5      public void remove( int rang )
6      {
7          if ( rang == 0 ) { ... }
8          else if ( rang == size()-1 )
9          {
10             getCell(size()-2).setNext(null);
11         }
12         else { ... }
13     }
14     ...
15 }
```

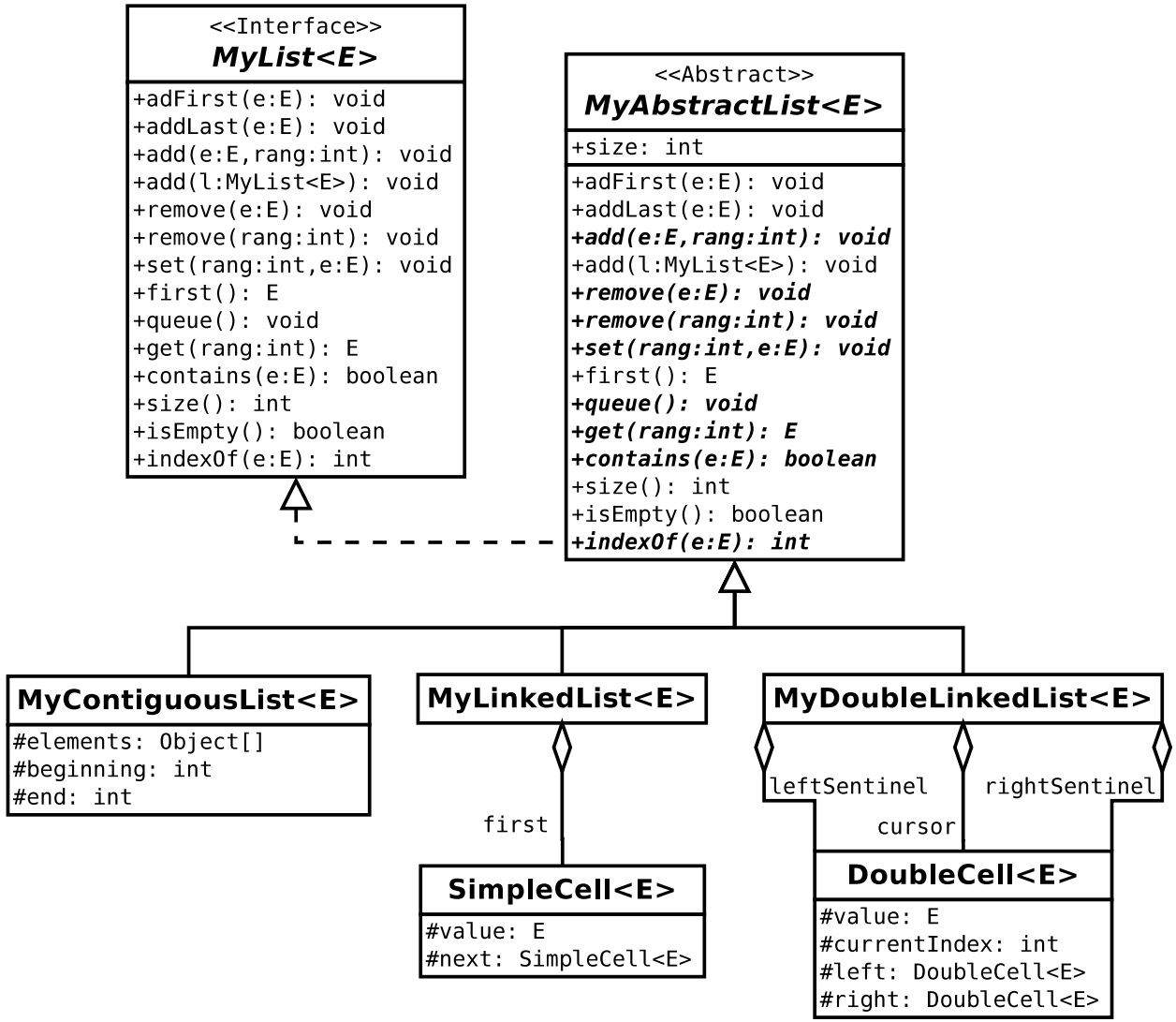
Solution de remove

```
1  public class MyLinkedList<E> extends MyAbstractList<E>
2  {
3      protected SimpleCell<E> first;
4      ...
5      public void remove( int rang )
6      {
7          if ( rang == 0 ) { ... }
8          else if ( rang == size()-1 ) { ... }
9          else
10         {
11             SimpleCell courant = getCell(rang-1);
12             courant.setNext(courant.getNext().getNext());
13         }
14     }
15     ...
16 }
```

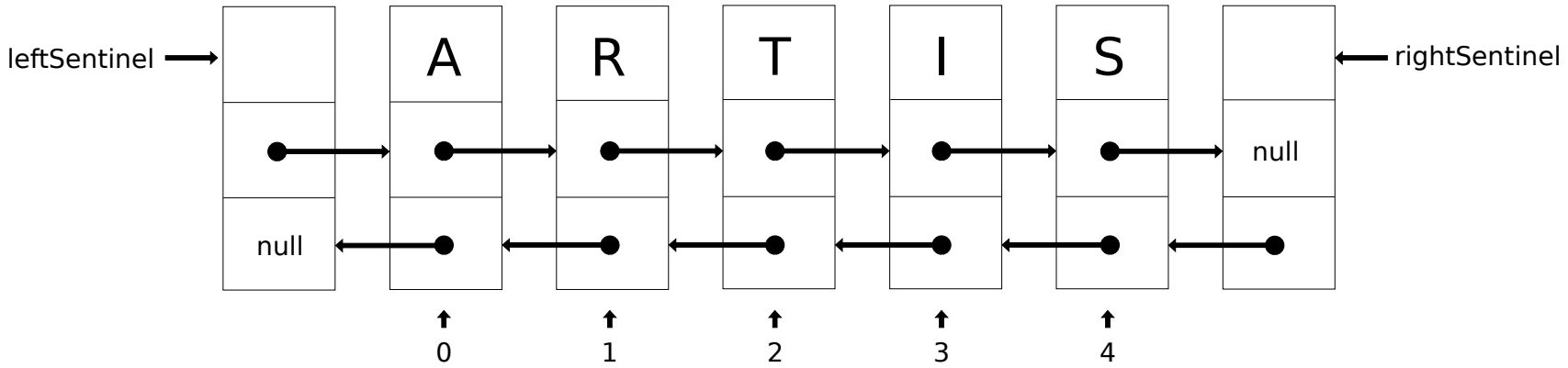
Solution de remove

```
1  public class MyLinkedList<E> extends MyAbstractList<E>
2  {
3      protected SimpleCell<E> first;
4      ...
5      public void remove( int rang )
6      {
7          if ( rang == 0 ) { ... }
8          else if ( rang == size()-1 ) { ... }
9          else { ... }
10         size--;
11     }
12     ...
13 }
```

MyDoubleLinkedList<E> - Diagramme de classe



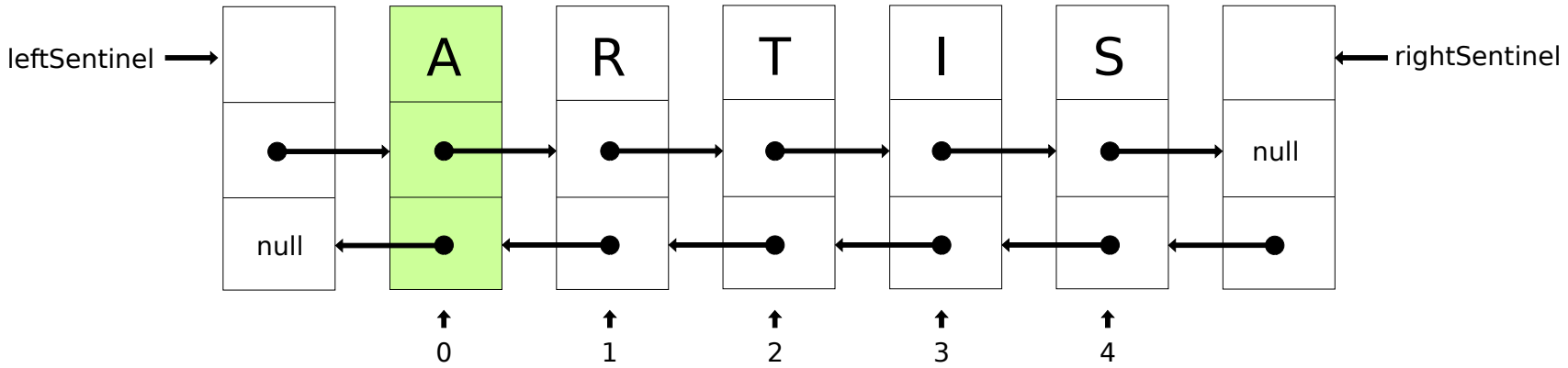
MyDoubleLinkedList<E> - Présentation



Dans une liste doublement chainee, les éléments connaissent leurs voisins de gauche ET de droite.

Un peu de vocabulaire

MyDoubleLinkedList<E> - Présentation

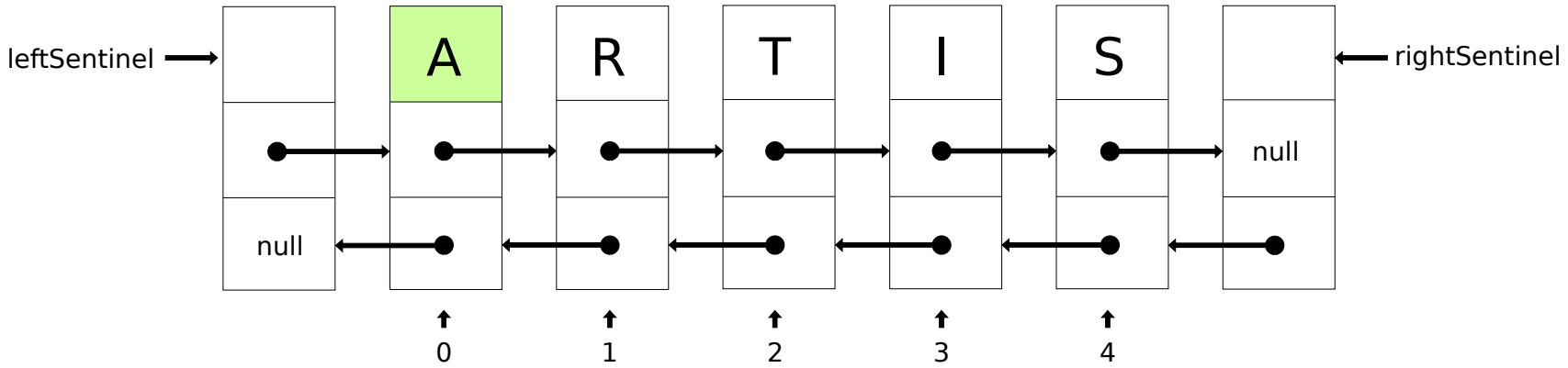


Dans une liste doublement chainee, les éléments connaissent leurs voisins de gauche ET de droite.

Un peu de vocabulaire

- Une **cellule**...

MyDoubleLinkedList<E> - Présentation

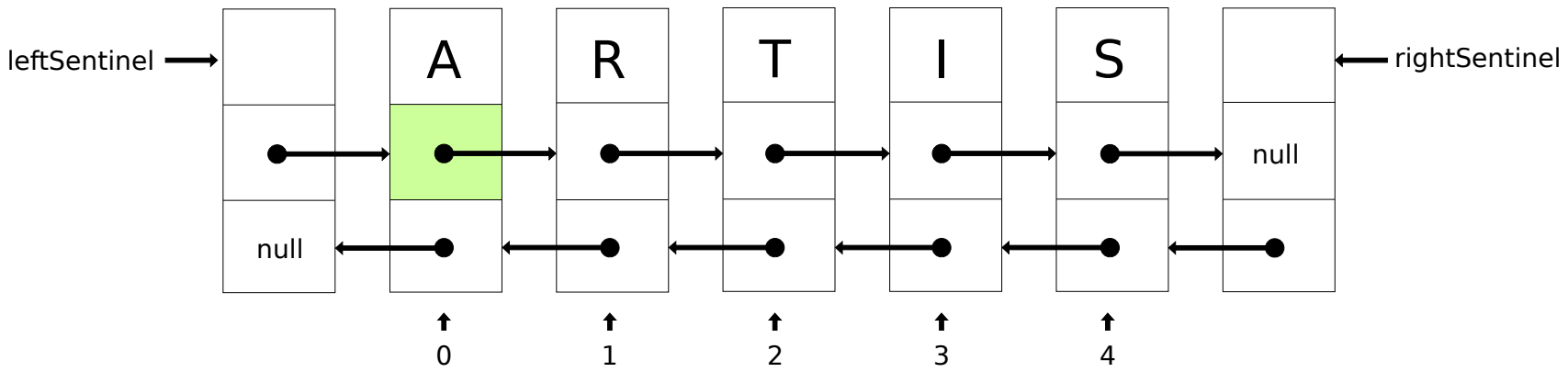


Dans une liste doublement chainee, les éléments connaissent leurs voisins de gauche ET de droite.

Un peu de vocabulaire

- Une **cellule**...
- ... contient une **valeur**...

MyDoubleLinkedList<E> - Présentation

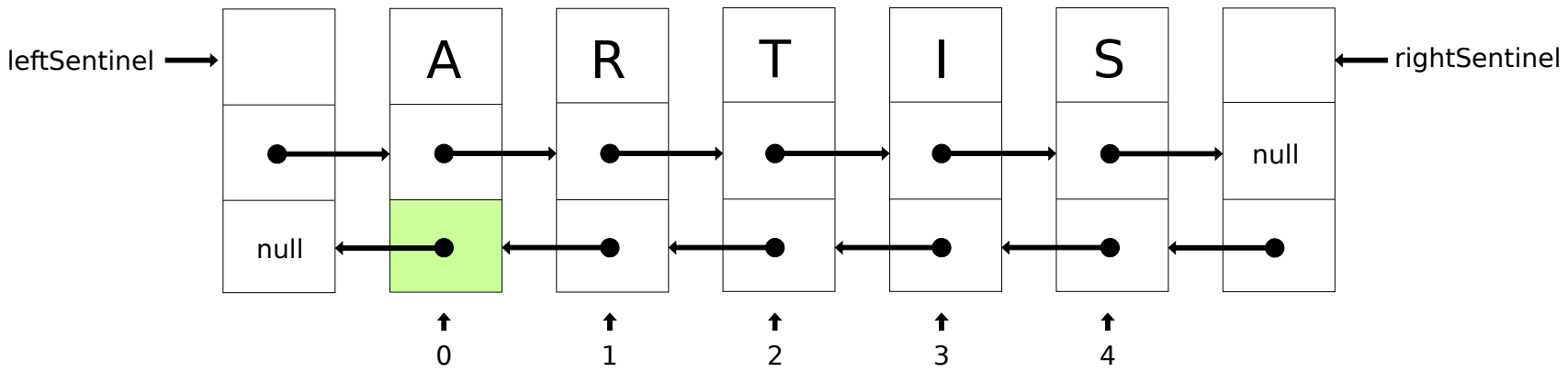


Dans une liste doublement chainee, les éléments connaissent leurs voisins de gauche ET de droite.

Un peu de vocabulaire

- Une **cellule**...
- ... contient une **valeur**...
- ... son **voisin de gauche**...

MyDoubleLinkedList<E> - Présentation



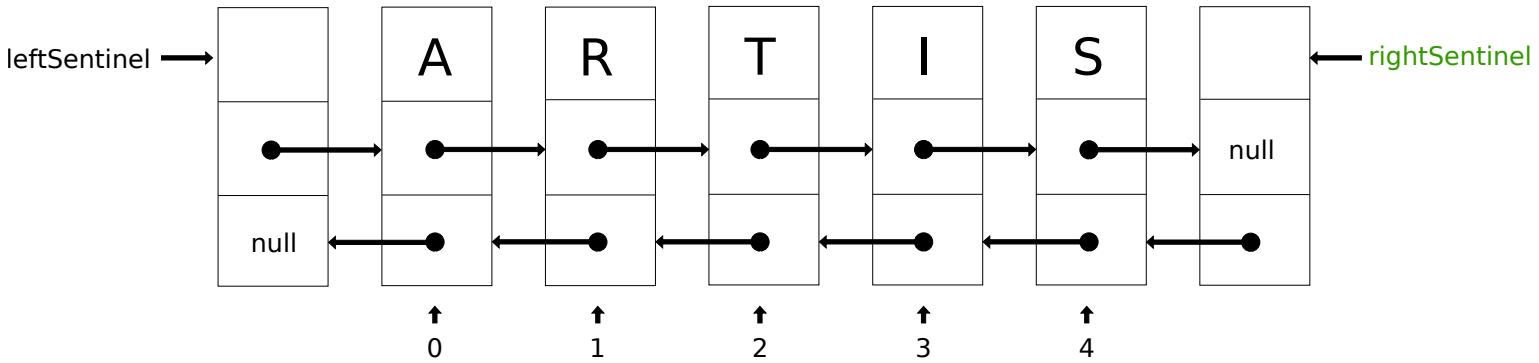
Dans une liste doublement chainee, les éléments connaissent leurs voisins de gauche ET de droite.

Un peu de vocabulaire

- Une **cellule**...
- ... contient une **valeur**...
- ... son **voisin de gauche**...
- ... et son **voisin de droite**.

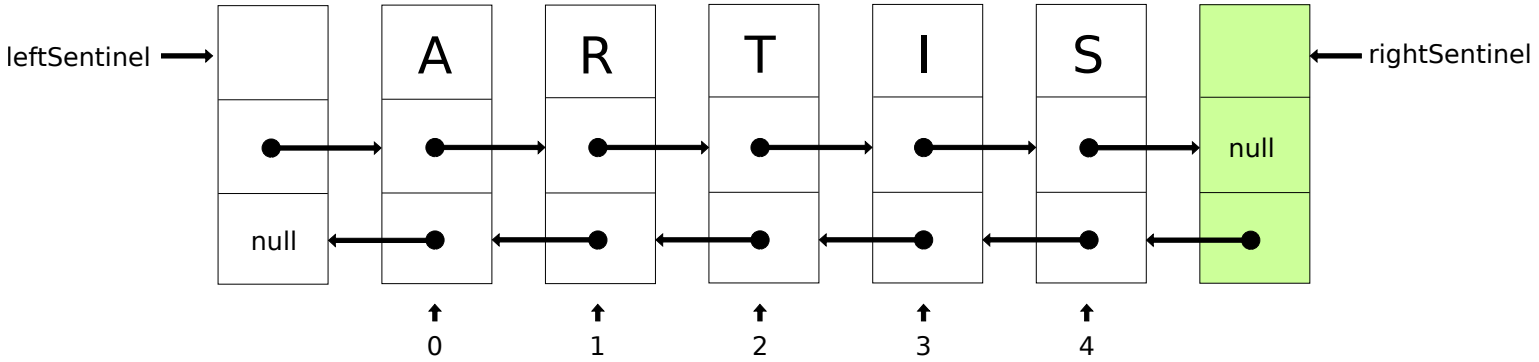
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



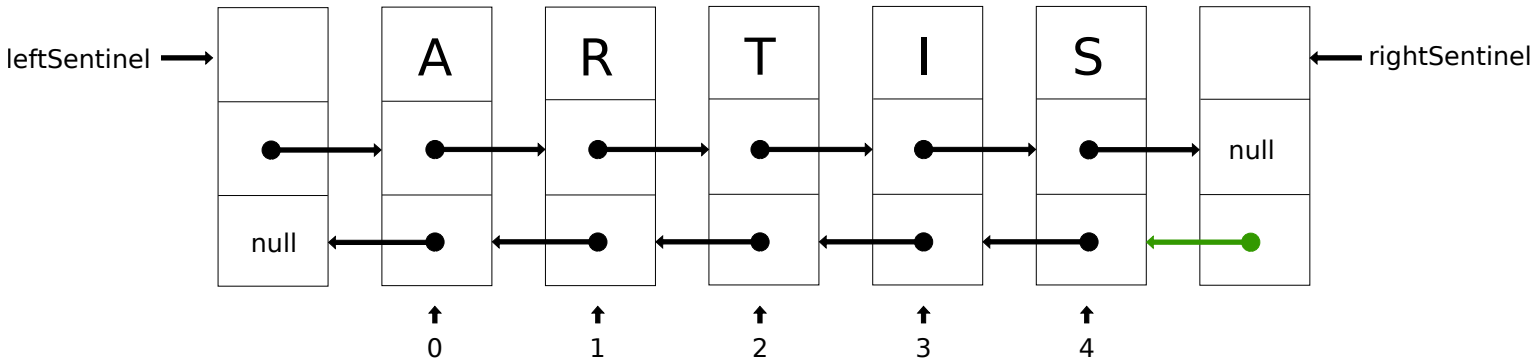
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



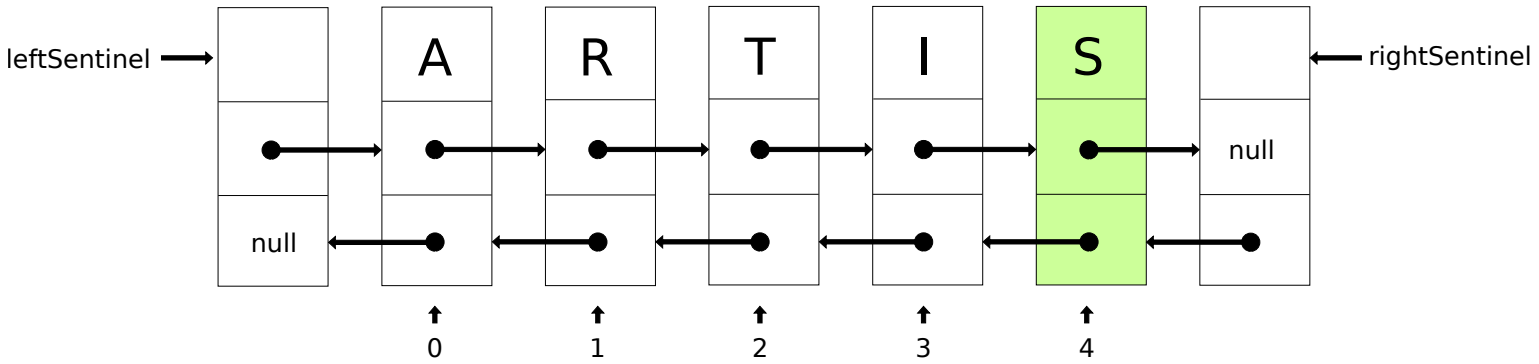
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



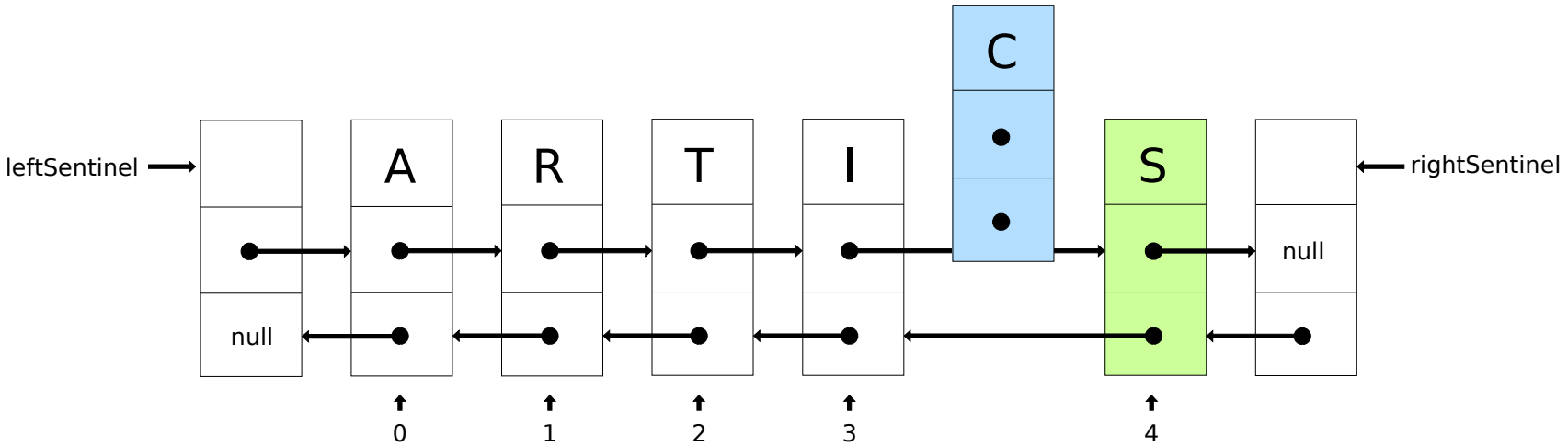
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



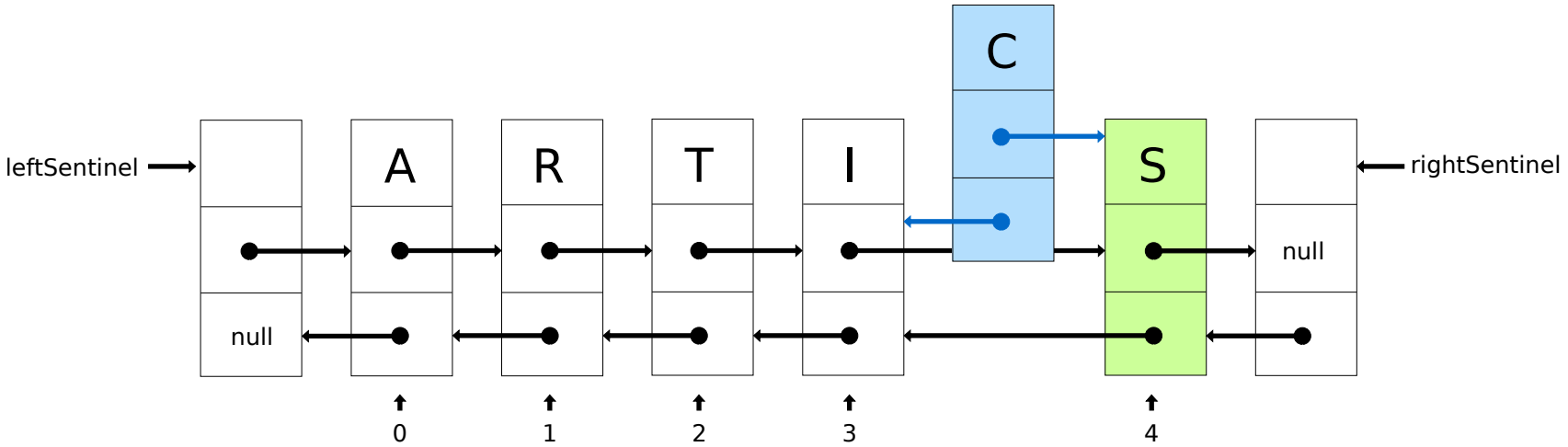
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



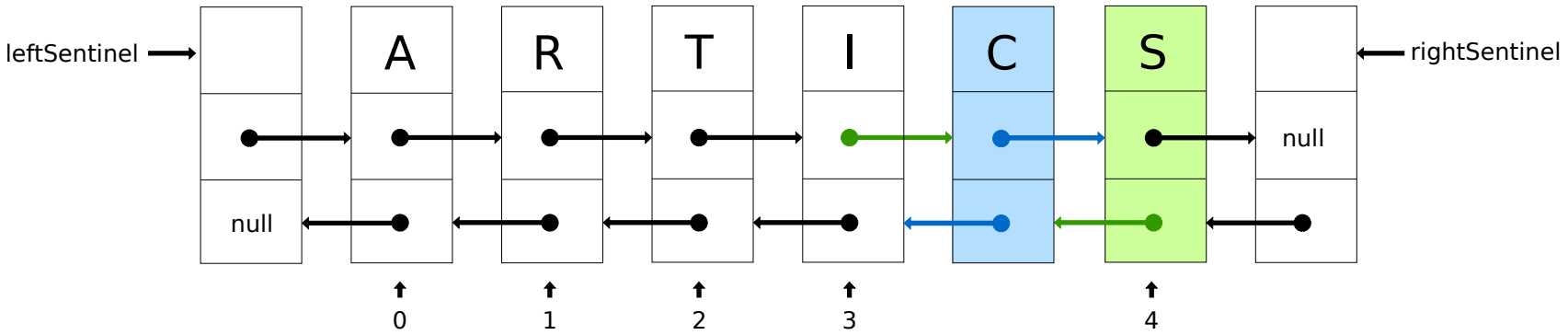
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



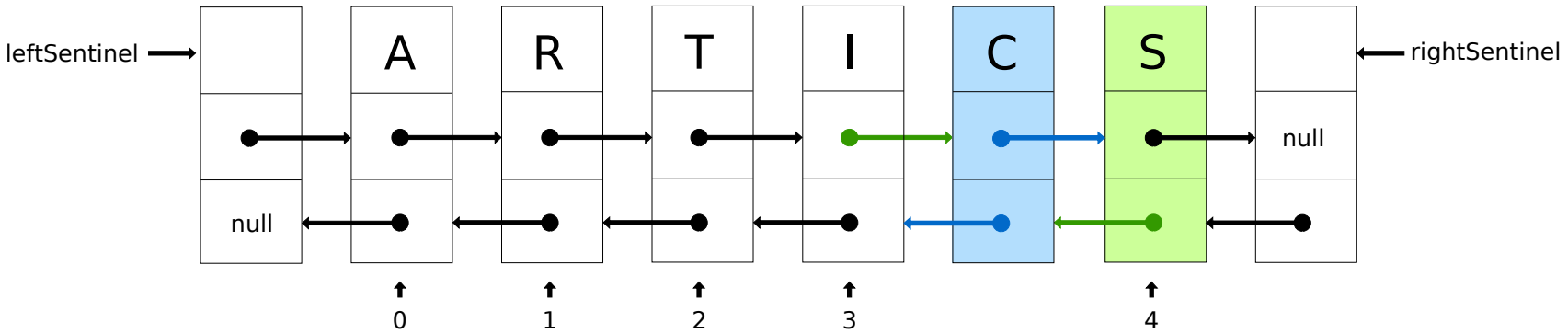
Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



Comment ça marche ?

Exemple : on souhaite insérer la lettre C au rang 4.



2 déplacements !

Exercice 2.E

Soit la classe `DoubleCell<E>`, représentant les cellules d'une liste doublement chainée.

Complétez la classe `MyDoubleLinkedList<E>` en implantant les fonction d'ajout et de suppression d'un élément.

Vous introduirez une méthode privée `positionCursor` qui permet de placer le curseur au rang donné en argument. Cela permet de minimiser le nombre de déplacement.

Classe DoubleCell<E>

```
1  public class DoubleCell<E>
2  {
3      protected E value ;
4      protected DoubleCell<E> left, right;
5      public DoubleCell( DoubleCell<E> left,
6                         DoubleCell<E> right, E value ) {
7          this.value = value;
8          this.left = left; this.right = right;
9      }
10     public E getValue() { return value ; }
11     public DoubleCell<E> getLeft() { return left; }
12     public DoubleCell<E> getRight() { return right; }
13     public void setValue( E value ) { this.value = value; }
14     public DoubleCell<E> setLeft( DoubleCell<E> left ) {
15         this.left = left;
16     }
17     public DoubleCell<E> setRight( DoubleCell<E> right ) {
18         this.right = right;
19     }
20 }
```

Compléter le constructeur

```
1  public class MyDoubleLinkedList<E>
2                                  extends MyAbstractList<E>
3  {
4      protected DoubleCell<E> leftSentinel, rightSentinel;
5      protected DoubleCell<E> cursor;
6      public int currentIndex; // rang du curseur
7
8      public MyDoubleLinkedList() {
9      {
10         // À compléter
11     }
12
13     public void add( E e, int rang ) { ... }
14     public void remove( int rang ) { ... }
15     private void positionCursor( int index ) { ... }
16     ...
17 }
```

Solution du constructeur

```
1  public class MyDoubleLinkedList<E> extends MyAbstractList<E
2  {
3      protected DoubleCell<E> leftSentinel, rightSentinel;
4      protected DoubleCell<E> cursor;
5      public int currentIndex; // rang du curseur
6
7      public MyDoubleLinkedList()
8      {
9          leftSentinel = new DoubleCell<E>();
10         rightSentinel = new DoubleCell<E>();
11         leftSentinel.setRight( rightSentinel );
12         rightSentinel.setLeft( leftSentinel );
13         cursor = leftSentinel;
14         currentIndex = -1;
15         size = 0;
16     }
17
18     ...
19 }
```

Complétez la fonction add

```
1  public class MyDoubleLinkedList<E> extends MyAbstractList<E
2  {
3      protected DoubleCell<E> leftSentinel, rightSentinel;
4      protected DoubleCell<E> cursor;
5      public int currentIndex; // rang du curseur
6
7      ...
8
9      public void add( E e, int rang )
10     {
11         // À compléter
12         // On suppose que positionCursor existe
13     }
14
15     ...
16 }
```

Solution de la fonction add

```
1  public class MyDoubleLinkedList<E> extends MyAbstractList<E
2  {
3      protected DoubleCell<E> leftSentinel, rightSentinel;
4      protected DoubleCell<E> cursor;
5      public int currentIndex; // rang du curseur
6      ...
7      public void add( E e, int rang )
8      {
9          DoubleCell<E> nouveau = new DoubleCell<E>();
10         nouveau.setValue(e);
11         this.positionCursor(rang);
12         nouveau.setLeft( cursor.getLeft() );
13         nouveau.setRight( cursor );
14         cursor.getLeft().setRight( nouveau );
15         cursor.setLeft( nouveau );
16         currentIndex++;
17         size++;
18     }
19     ...
20 }
```

Complétez la fonction add

```
1  public class MyDoubleLinkedList<E> extends MyAbstractList<E
2  {
3      protected DoubleCell<E> leftSentinel, rightSentinel;
4      protected DoubleCell<E> cursor;
5      public int currentIndex; // rang du curseur
6
7      ...
8
9      public void remove( int rang )
10     {
11         // À compléter
12         // On suppose que positionCursor existe
13     }
14
15     ...
16 }
```

Solution de la fonction remove

```
1  public class MyDoubleLinkedList<E> extends MyAbstractList<E
2  {
3      protected DoubleCell<E> leftSentinel, rightSentinel;
4      protected DoubleCell<E> cursor;
5      public int currentIndex; // rang du curseur
6      ...
7      public void remove( int rang )
8      {
9          this.positionCursor(rang);
10         cursor.getLeft().setRight( cursor.getRight() );
11         cursor.getRight().setLeft( cursor.getLeft() );
12         cursor = cursor.getRight();
13         size--;
14     }
15     ...
16 }
```

Plan

Listes à implantation contiguë

Avantages	Inconvénients

Listes à implantation contiguë

Avantages	Inconvénients
Place en mémoire	

Listes à implantation contiguë

Avantages	Inconvénients
Place en mémoire	
Accès aux données	

Listes à implantation contiguë

Avantages	Inconvénients
Place en mémoire	Place en mémoire
Accès aux données	

Listes à implantation contiguë

Avantages	Inconvénients
Place en mémoire	Place en mémoire
$\rightarrow$ *éléments seulement*	$\rightarrow$ *allocation initiale*
Accès aux données	

Listes à implantation contiguë

Avantages	Inconvénients
Place en mémoire	Place en mémoire
$\rightarrow$ *éléments seulement*	$\rightarrow$ *allocation initiale*
Accès aux données	Insertions/suppression

Listes à implantation contiguë

Avantages	Inconvénients
Place en mémoire	Place en mémoire
→ *éléments seulement*	→ *allocation initiale*
Accès aux données	Insertions/suppression

Une gestion cyclique permet de réduire le coût des insertions et suppressions.

Listes à implantation chainée

Avantages	Inconvénients

Listes à implantation chainée

Avantages	Inconvénients
Place en mémoire	

Listes à implantation chainée

Avantages	Inconvénients
Place en mémoire	
Insertion/suppression	

Listes à implantation chainée

Avantages	Inconvénients
Place en mémoire	Place en mémoire
Insertion/suppression	

Listes à implantation chainée

Avantages	Inconvénients
Place en mémoire	Place en mémoire
→ *allocation "à la volée"*	→ *utilisation de cellules*
Insertion/suppression	

Listes à implantation chainée

Avantages	Inconvénients
Place en mémoire	Place en mémoire
→ *allocation "à la volée"*	→ *utilisation de cellules*
Insertion/suppression	Accès aux données

Listes à implantation chainée

Avantages	Inconvénients
Place en mémoire	Place en mémoire
→ *allocation "à la volée"*	→ *utilisation de cellules*
Insertion/suppression	Accès aux données

Un double chainage permet de réduire le temps d'accès aux données.

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire		
Accès/Insertion au rang k		
Accès/Insertion en tête		
Accès/Insertion en queue		

Chainée	Simple	Double
Coût mémoire		
Accès/Insertion au rang k		
Accès/Insertion en tête		
Accès/Insertion en queue		

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	
Accès/Insertion au rang k		
Accès/Insertion en tête		
Accès/Insertion en queue		

Chainée	Simple	Double
Coût mémoire		
Accès/Insertion au rang k		
Accès/Insertion en tête		
Accès/Insertion en queue		

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	
Accès/Insertion au rang k Accès/Insertion en tête Accès/Insertion en queue	$\Theta(1)/\Theta(N)$	

Chainée	Simple	Double
Coût mémoire		
Accès/Insertion au rang k Accès/Insertion en tête Accès/Insertion en queue		

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	
Accès/Insertion au rang k	$\Theta(1)/\Theta(N)$	
Accès/Insertion en tête	$\Theta(1)/\Theta(N)$	
Accès/Insertion en queue		

Chainée	Simple	Double
Coût mémoire		
Accès/Insertion au rang k		
Accès/Insertion en tête		
Accès/Insertion en queue		

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	
Accès/Insertion au rang k	$\Theta(1)/\Theta(N)$	
Accès/Insertion en tête	$\Theta(1)/\Theta(N)$	
Accès/Insertion en queue	$\Theta(1)/\Theta(1)$	

Chainée	Simple	Double
Coût mémoire		
Accès/Insertion au rang k		
Accès/Insertion en tête		
Accès/Insertion en queue		

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	$T_T \times T_E$
Accès/Insertion au rang k	$\Theta(1)/\Theta(N)$	
Accès/Insertion en tête	$\Theta(1)/\Theta(N)$	
Accès/Insertion en queue	$\Theta(1)/\Theta(1)$	

Chainée	Simple	Double
Coût mémoire		
Accès/Insertion au rang k		
Accès/Insertion en tête		
Accès/Insertion en queue		

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	$T_T \times T_E$
Accès/Insertion au rang k	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(N/2)$
Accès/Insertion en tête	$\Theta(1)/\Theta(N)$	
Accès/Insertion en queue	$\Theta(1)/\Theta(1)$	

Chainée	Simple	Double
Coût mémoire		
Accès/Insertion au rang k		
Accès/Insertion en tête		
Accès/Insertion en queue		

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	$T_T \times T_E$
Accès/Insertion au rang k	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(N/2)$
Accès/Insertion en tête	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(1)$
Accès/Insertion en queue	$\Theta(1)/\Theta(1)$	

Chainée	Simple	Double
Coût mémoire		
Accès/Insertion au rang k		
Accès/Insertion en tête		
Accès/Insertion en queue		

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	$T_T \times T_E$
Accès/Insertion au rang k	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(N/2)$
Accès/Insertion en tête	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(1)$
Accès/Insertion en queue	$\Theta(1)/\Theta(1)$	$\Theta(1)/\Theta(1)$

Chainée	Simple	Double
Coût mémoire		
Accès/Insertion au rang k		
Accès/Insertion en tête		
Accès/Insertion en queue		

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	$T_T \times T_E$
Accès/Insertion au rang k	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(N/2)$
Accès/Insertion en tête	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(1)$
Accès/Insertion en queue	$\Theta(1)/\Theta(1)$	$\Theta(1)/\Theta(1)$

Chainée	Simple	Double
Coût mémoire		
Accès/Insertion au rang k		
Accès/Insertion en tête		
Accès/Insertion en queue		

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	$T_T \times T_E$
Accès/Insertion au rang k	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(N/2)$
Accès/Insertion en tête	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(1)$
Accès/Insertion en queue	$\Theta(1)/\Theta(1)$	$\Theta(1)/\Theta(1)$

Chainée	Simple	Double
Coût mémoire	$N \times (T_E + 1)$	
Accès/Insertion au rang k		
Accès/Insertion en tête		
Accès/Insertion en queue		

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	$T_T \times T_E$
Accès/Insertion au rang k	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(N/2)$
Accès/Insertion en tête	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(1)$
Accès/Insertion en queue	$\Theta(1)/\Theta(1)$	$\Theta(1)/\Theta(1)$

Chainée	Simple	Double
Coût mémoire	$N \times (T_E + 1)$	
Accès/Insertion au rang k	$\Theta(N)/\Theta(1)$	
Accès/Insertion en tête		
Accès/Insertion en queue		

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	$T_T \times T_E$
Accès/Insertion au rang k	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(N/2)$
Accès/Insertion en tête	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(1)$
Accès/Insertion en queue	$\Theta(1)/\Theta(1)$	$\Theta(1)/\Theta(1)$

Chainée	Simple	Double
Coût mémoire	$N \times (T_E + 1)$	
Accès/Insertion au rang k	$\Theta(N)/\Theta(1)$	
Accès/Insertion en tête	$\Theta(1)/\Theta(1)$	
Accès/Insertion en queue		

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	$T_T \times T_E$
Accès/Insertion au rang k	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(N/2)$
Accès/Insertion en tête	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(1)$
Accès/Insertion en queue	$\Theta(1)/\Theta(1)$	$\Theta(1)/\Theta(1)$

Chainée	Simple	Double
Coût mémoire	$N \times (T_E + 1)$	
Accès/Insertion au rang k	$\Theta(N)/\Theta(1)$	
Accès/Insertion en tête	$\Theta(1)/\Theta(1)$	
Accès/Insertion en queue	$\Theta(N)/\Theta(1)$	

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	$T_T \times T_E$
Accès/Insertion au rang k	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(N/2)$
Accès/Insertion en tête	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(1)$
Accès/Insertion en queue	$\Theta(1)/\Theta(1)$	$\Theta(1)/\Theta(1)$

Chainée	Simple	Double
Coût mémoire	$N \times (T_E + 1)$	$N \times (T_E + 2)$
Accès/Insertion au rang k	$\Theta(N)/\Theta(1)$	
Accès/Insertion en tête	$\Theta(1)/\Theta(1)$	
Accès/Insertion en queue	$\Theta(N)/\Theta(1)$	

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	$T_T \times T_E$
Accès/Insertion au rang k	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(N/2)$
Accès/Insertion en tête	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(1)$
Accès/Insertion en queue	$\Theta(1)/\Theta(1)$	$\Theta(1)/\Theta(1)$

Chainée	Simple	Double
Coût mémoire	$N \times (T_E + 1)$	$N \times (T_E + 2)$
Accès/Insertion au rang k	$\Theta(N)/\Theta(1)$	$\Theta(N/2)/\Theta(1)$
Accès/Insertion en tête	$\Theta(1)/\Theta(1)$	
Accès/Insertion en queue	$\Theta(N)/\Theta(1)$	

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	$T_T \times T_E$
Accès/Insertion au rang k	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(N/2)$
Accès/Insertion en tête	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(1)$
Accès/Insertion en queue	$\Theta(1)/\Theta(1)$	$\Theta(1)/\Theta(1)$

Chainée	Simple	Double
Coût mémoire	$N \times (T_E + 1)$	$N \times (T_E + 2)$
Accès/Insertion au rang k	$\Theta(N)/\Theta(1)$	$\Theta(N/2)/\Theta(1)$
Accès/Insertion en tête	$\Theta(1)/\Theta(1)$	$\Theta(1)/\Theta(1)$
Accès/Insertion en queue	$\Theta(N)/\Theta(1)$	

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	$T_T \times T_E$
Accès/Insertion au rang k	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(N/2)$
Accès/Insertion en tête	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(1)$
Accès/Insertion en queue	$\Theta(1)/\Theta(1)$	$\Theta(1)/\Theta(1)$

Chainée	Simple	Double
Coût mémoire	$N \times (T_E + 1)$	$N \times (T_E + 2)$
Accès/Insertion au rang k	$\Theta(N)/\Theta(1)$	$\Theta(N/2)/\Theta(1)$
Accès/Insertion en tête	$\Theta(1)/\Theta(1)$	$\Theta(1)/\Theta(1)$
Accès/Insertion en queue	$\Theta(N)/\Theta(1)$	$\Theta(1)/\Theta(1)$

Complexité au pire cas

Soit une liste de N éléments de taille T_E.
L'implantation contiguë se fait dans un tableau de taille T_T.

Contigue	Linéaire	Circulaire
Coût mémoire	$T_T \times T_E$	$T_T \times T_E$
Accès/Insertion au rang k	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(N/2)$
Accès/Insertion en tête	$\Theta(1)/\Theta(N)$	$\Theta(1)/\Theta(1)$
Accès/Insertion en queue	$\Theta(1)/\Theta(1)$	$\Theta(1)/\Theta(1)$

Chainée	Simple	Double
Coût mémoire	$N \times (T_E + 1)$	$N \times (T_E + 2)$
Accès/Insertion au rang k	$\Theta(N)/\Theta(1)$	$\Theta(N/2)/\Theta(1)$
Accès/Insertion en tête	$\Theta(1)/\Theta(1)$	$\Theta(1)/\Theta(1)$
Accès/Insertion en queue	$\Theta(N)/\Theta(1)$	$\Theta(1)/\Theta(1)$