Elements d'Algorithmique
CMTD6: Listes chaînées

Ensembles dynamiques

Un ensemble d'objets manipulés par un algorithme change généralement pendant son exécution. De tels ensembles sont appelés dynamiques.

Ensembles dynamiques

Un ensemble d'objets manipulés par un algorithme change généralement pendant son exécution. De tels ensembles sont appelés dynamiques.

Le cours d'aujourd'hui : qu'est-ce qu'une bonne structure de données pour représenter des ensembles dynamiques ordonnés d'objets de même type ?

Des opérations standard sur un tel ensemble S:

- RECHERCHER$(S, clé)$
- INSERTION(S, x)
- SUPPRESSION(S, x)

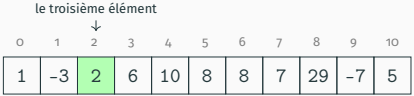
Pourquoi pas des tableaux ?

Pour les tableaux l'ordre linéaire est déterminé par les indices.

Pourquoi pas des tableaux ?

Pour les tableaux l'ordre linéaire est déterminé par les indices.

Advantage : accès en temps constant à un élément du tableau !



Pourquoi pas des tableaux ?

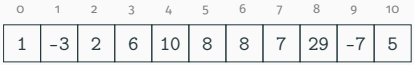
Gros inconvénients

- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.

Pourquoi pas des tableaux ?

Gros inconvénients

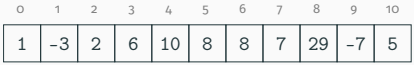
- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



Pourquoi pas des tableaux ?

Gros inconvénients

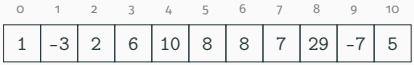
- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



Pourquoi pas des tableaux ?

Gros inconvénients

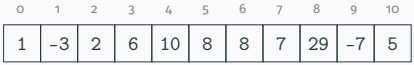
- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



Pourquoi pas des tableaux ?

Gros inconvénients

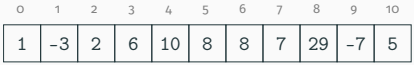
- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



Pourquoi pas des tableaux ?

Gros inconvénients

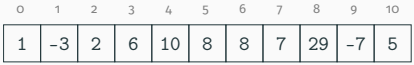
- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



Pourquoi pas des tableaux ?

Gros inconvénients

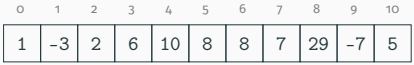
- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



Pourquoi pas des tableaux ?

Gros inconvénients

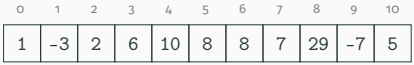
- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



Pourquoi pas des tableaux ?

Gros inconvénients

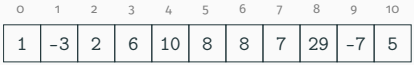
- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



Pourquoi pas des tableaux ?

Gros inconvénients

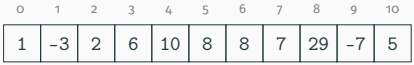
- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



Pourquoi pas des tableaux ?

Gros inconvénients

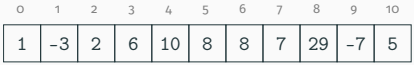
- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



Pourquoi pas des tableaux ?

Gros inconvénients

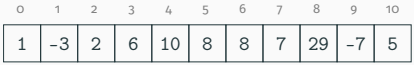
- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



Pourquoi pas des tableaux ?

Gros inconvénients

- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



Pourquoi pas des tableaux ?

Gros inconvénients

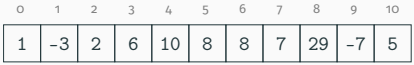
- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.

- Il est difficile et inefficace d'insérer un nouvel élément

Pourquoi pas des tableaux ?

Gros inconvénients

- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.

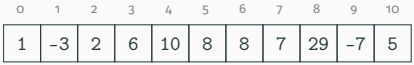


- Il est difficile et inefficace d'insérer un nouvel élément

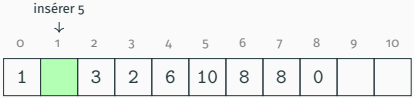
Pourquoi pas des tableaux ?

Gros inconvénients

- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



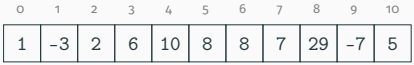
- Il est difficile et inefficace d'insérer un nouvel élément



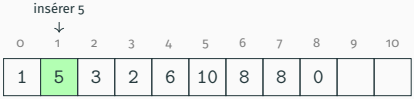
Pourquoi pas des tableaux ?

Gros inconvénients

- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



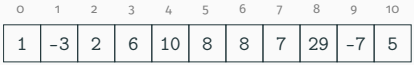
- Il est difficile et inefficace d'insérer un nouvel élément



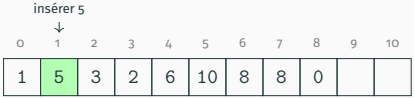
Pourquoi pas des tableaux ?

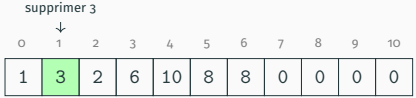
Gros inconvénients

- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



- Il est difficile et inefficace d'insérer un nouvel élément et de supprimer des éléments existants.

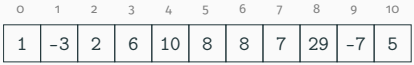




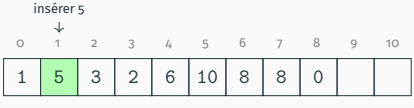
Pourquoi pas des tableaux ?

Gros inconvénients

- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



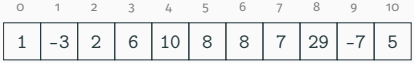
- Il est difficile et inefficace d'insérer un nouvel élément et de supprimer des éléments existants.



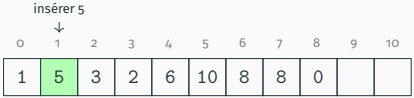
Pourquoi pas des tableaux ?

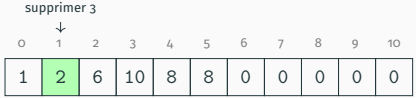
Gros inconvénients

- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



- Il est difficile et inefficace d'insérer un nouvel élément et de supprimer des éléments existants.

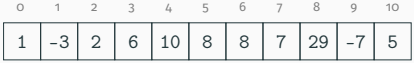




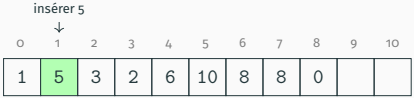
Pourquoi pas des tableaux ?

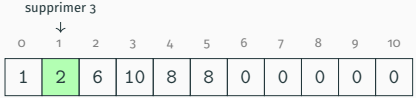
Gros inconvénients

- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



- Il est difficile et inefficace d'insérer un nouvel élément et de supprimer des éléments existants. (complexité : $O(n)$).

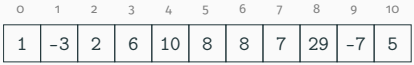




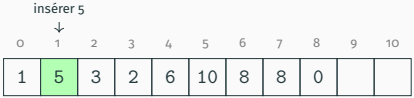
Pourquoi pas des tableaux ?

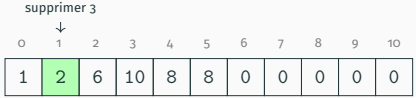
Gros inconvénients

- La taille d'un tableau est fixe, pour ajouter plus d'éléments, on peut avoir besoin de créer un nouveau tableau plus grand et de copier les éléments existants.



- Il est difficile et inefficace d'insérer un nouvel élément et de supprimer des éléments existants. (complexité : $O(n)$).



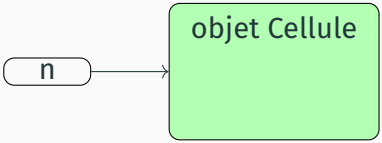


- Un tableau Java 1-D doit occuper une mémoire contiguë. Lorsque l'on stocke de grandes quantités de données, il peut être impossible de la trouver.

Listes chaînées

Une liste chaînée est une structure de données dans laquelle
les objets sont arrangés linéairement,
mais dont l'ordre est déterminé par un pointeur dans chaque objet.

Références des objets



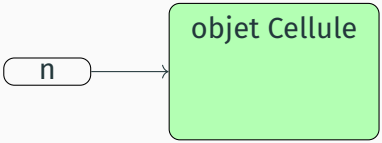
En Java, on considère une classe «Cellule».

```
class Cellule {

}


Cellule n = new Cellule();
```

Références des objets



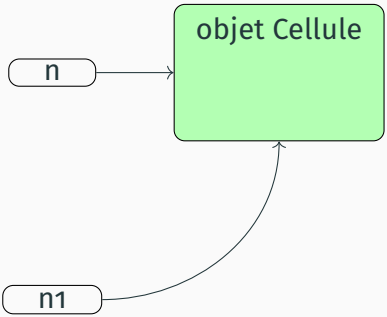
En Java, on considère une classe «Cellule».

```
class Cellule {

}


Cellule n = new Cellule();
Cellule n1 = n;
```

Références des objets



En Java, on considère une classe «Cellule».

```
class Cellule {

}


Cellule n = new Cellule();
Cellule n1 = n;
```

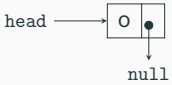
Enchainer des objets Cellule

En Java, on considère une classe «Cellule» et une classe «Liste».

```
class Cellule {
  int key;
  Cellule next;
}

class Liste {
  Cellule head;
}
```

```
Cellule head = new Cellule();
head.key = 4;
head.next = new Cellule();
head.next.key = 1;
Cellule n3  = new Cellule();
n3.key = 2;
head.next.next = n3;
Liste l = new Liste(head);
```



Enchainer des objets Cellule

En Java, on considère une classe «Cellule» et une classe «Liste».

```
class Cellule {
  int key;
  Cellule next;
}

class Liste {
  Cellule head;
}
```

```
Cellule head = new Cellule();
head.key = 4;
head.next = new Cellule();
head.next.key = 1;
Cellule n3  = new Cellule();
n3.key = 2;
head.next.next = n3;
Liste l = new Liste(head);
```

Enchainer des objets Cellule

En Java, on considère une classe «Cellule» et une classe «Liste».

```
class Cellule {
  int key;
  Cellule next;
}

class Liste {
  Cellule head;
}
```

```
Cellule head = new Cellule();
head.key = 4;
head.next = new Cellule();
head.next.key = 1;
Cellule n3  = new Cellule();
n3.key = 2;
head.next.next = n3;
Liste l = new Liste(head);
```

Enchainer des objets Cellule

En Java, on considère une classe «Cellule» et une classe «Liste».

```
class Cellule {
  int key;
  Cellule next;
}

class Liste {
  Cellule head;
}
```

```
Cellule head = new Cellule();
head.key = 4;
head.next = new Cellule();
head.next.key = 1;
Cellule n3  = new Cellule();
n3.key = 2;
head.next.next = n3;
Liste l = new Liste(head);
```

Enchainer des objets Cellule

En Java, on considère une classe «Cellule» et une classe «Liste».

```
class Cellule {
  int key;
  Cellule next;
}

class Liste {
  Cellule head;
}
```

```
Cellule head = new Cellule();
head.key = 4;
head.next = new Cellule();
head.next.key = 1;
Cellule n3  = new Cellule();
n3.key = 2;
head.next.next = n3;
Liste l = new Liste(head);
```

Enchainer des objets Cellule

En Java, on considère une classe «Cellule» et une classe «Liste».

```
class Cellule {
  int key;
  Cellule next;
}

class Liste {
  Cellule head;
}
```

```
Cellule head = new Cellule();
head.key = 4;
head.next = new Cellule();
head.next.key = 1;
Cellule n3  = new Cellule();
n3.key = 2;
head.next.next = n3;
Liste l = new Liste(head);
```

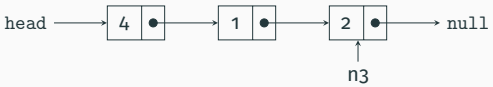
Enchainer des objets Cellule

En Java, on considère une classe «Cellule» et une classe «Liste».

```
class Cellule {
  int key;
  Cellule next;
}

class Liste {
  Cellule head;
}
```

```
Cellule head = new Cellule();
head.key = 4;
head.next = new Cellule();
head.next.key = 1;
Cellule n3  = new Cellule();
n3.key = 2;
head.next.next = n3;
Liste l = new Liste(head);
```



Enchainer des objets Cellule

En Java, on considère une classe «Cellule» et une classe «Liste».

```
class Cellule {
  int key;
  Cellule next;
}

class Liste {
  Cellule head;
}
```

```
Cellule head = new Cellule();
head.key = 4;
head.next = new Cellule();
head.next.key = 1;
Cellule n3  = new Cellule();
n3.key = 2;
head.next.next = n3;
Liste l = new Liste(head);
```

Nous obtenons une **liste chaînée** !

Listes chaînées: avantages et inconvénients

Avantages

- Taille flexible: une liste chaînée peut croître de façon dynamique !

Listes chaînées: avantages et inconvénients

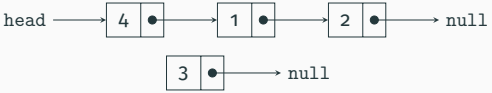
Avantages

- Taille flexible: une liste chaînée peut croître de façon dynamique !
- Pas besoin de mémoire contiguë !

Listes chaînées: avantages et inconvénients

Avantages

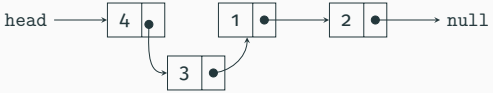
- Taille flexible: une liste chaînée peut croître de façon dynamique !
- Pas besoin de mémoire contiguë !
- L'insertion d'un élément ne nécessitent que la mise à jour de certains pointeurs



Listes chaînées: avantages et inconvénients

Avantages

- Taille flexible: une liste chaînée peut croître de façon dynamique !
- Pas besoin de mémoire contiguë !
- L'insertion d'un élément ne nécessitent que la mise à jour de certains pointeurs

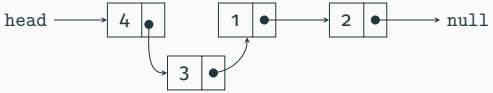


- Pareil pour la suppression d'un élément

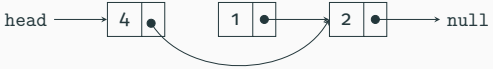
Listes chaînées: avantages et inconvénients

Avantages

- Taille flexible: une liste chaînée peut croître de façon dynamique !
- Pas besoin de mémoire contiguë !
- L'insertion d'un élément ne nécessitent que la mise à jour de certains pointeurs



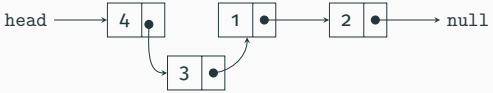
- Pareil pour la suppression d'un élément



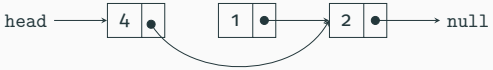
Listes chaînées: avantages et inconvénients

Avantages

- Taille flexible: une liste chaînée peut croître de façon dynamique !
- Pas besoin de mémoire contiguë !
- L'insertion d'un élément ne nécessitent que la mise à jour de certains pointeurs



- Pareil pour la suppression d'un élément

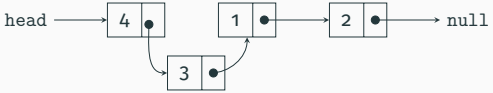


- Ajout et suppression en tête de liste en temps constant $O(1)$.

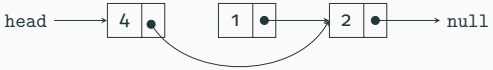
Listes chaînées: avantages et inconvénients

Avantages

- Taille flexible: une liste chaînée peut croître de façon dynamique !
- Pas besoin de mémoire contiguë !
- L'insertion d'un élément ne nécessitent que la mise à jour de certains pointeurs



- Pareil pour la suppression d'un élément



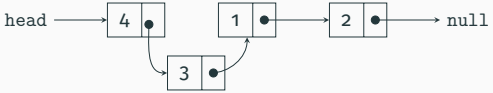
- Ajout et suppression en tête de liste en temps constant $O(1)$.

Inconvénient: Pour accéder au $i^{ème}$ élément, il faut parcourir tous les éléments qui le précèdent

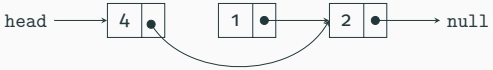
Listes chaînées: avantages et inconvénients

Avantages

- Taille flexible: une liste chaînée peut croître de façon dynamique !
- Pas besoin de mémoire contiguë !
- L'insertion d'un élément ne nécessitent que la mise à jour de certains pointeurs



- Pareil pour la suppression d'un élément



- Ajout et suppression en tête de liste en temps constant $O(1)$.

Inconvénient: Pour accéder au $i^{ème}$ élément, il faut parcourir tous les éléments qui le précèdent (complexité $O(n)$) .

Recherche dans une liste chaînée

Entrée : une liste chaînée L et une clé x

Sortie : c tel que $c.key = x$ ou `null` si non trouvé

```
1: fonction RECHERCHE(L, x)
2:     c ← L.head
3:     tant que c ≠ null faire
4:         si c.key = x alors
5:             retourne c
6:         c ← c.next
7:     retourne non trouvé
```

Exemple : RECHERCHE$(L, 5)$:

Recherche dans une liste chaînée

Entrée : une liste chaînée L et une clé x

Sortie : c tel que $c.key = x$ ou `null` si non trouvé

```
1: fonction RECHERCHE(L, x)
2:     c ← L.head
3:     tant que c ≠ null faire
4:         si c.key = x alors
5:             retourne c
6:         c ← c.next
7:     retourne non trouvé
```

Exemple : RECHERCHE$(L, 5)$:

Recherche dans une liste chaînée

Entrée : une liste chaînée L et une clé x

Sortie : c tel que $c.key = x$ ou `null` si non trouvé

```
1: fonction RECHERCHE(L, x)
2:     c ← L.head
3:     tant que c ≠ null faire
4:         si c.key = x alors
5:             retourne c
6:         c ← c.next
7:     retourne non trouvé
```

Exemple : RECHERCHE$(L, 5)$:

Recherche dans une liste chaînée

Entrée : une liste chaînée L et une clé x

Sortie : c tel que $c.key = x$ ou `null` si non trouvé

```
1: fonction RECHERCHE(L, x)
2:     c ← L.head
3:     tant que c ≠ null faire
4:         si c.key = x alors
5:             retourne c
6:         c ← c.next
7:     retourne non trouvé
```

Exemple : RECHERCHE$(L, 5)$:

Recherche dans une liste chaînée

Entrée : une liste chaînée L et une clé x

Sortie : c tel que $c.key = x$ ou `null` si non trouvé

```
1: fonction RECHERCHE(L, x)
2:     c ← L.head
3:     tant que c ≠ null faire
4:         si c.key = x alors
5:             retourne c
6:         c ← c.next
7:     retourne non trouvé
```

Exemple : RECHERCHE$(L, 5)$:

Recherche dans une liste chaînée

Entrée : une liste chaînée L et une clé x

Sortie : c tel que $c.key = x$ ou `null` si non trouvé

```
1: fonction RECHERCHE(L, x)
2:     c ← L.head
3:     tant que c ≠ null faire
4:         si c.key = x alors
5:             retourne c
6:         c ← c.next
7:     retourne non trouvé
```

Exemple : RECHERCHE$(L, 5)$:

Recherche dans une liste chaînée

Entrée : une liste chaînée L et une clé x

Sortie : c tel que $c.key = x$ ou `null` si non trouvé

```
1: fonction RECHERCHE(L, x)
2:     c ← L.head
3:     tant que c ≠ null faire
4:         si c.key = x alors
5:             retourne c
6:         c ← c.next
7:     retourne non trouvé
```

Exemple : RECHERCHE$(L, 5)$:

Recherche dans une liste chaînée

Entrée : une liste chaînée L et une clé x

Sortie : c tel que $c.key = x$ ou `null` si non trouvé

```
1: fonction RECHERCHE(L, x)
2:     c ← L.head
3:     tant que c ≠ null faire
4:         si c.key = x alors
5:             retourne c
6:         c ← c.next
7:     retourne non trouvé
```

Exemple : RECHERCHE$(L, 5)$:

Recherche dans une liste chaînée

Entrée : une liste chaînée L et une clé x

Sortie : c tel que $c.key = x$ ou `null` si non trouvé

```
1: fonction RECHERCHE(L, x)
2:     c ← L.head
3:     tant que c ≠ null faire
4:         si c.key = x alors
5:             retourne c
6:         c ← c.next
7:     retourne non trouvé
```

Exemple : RECHERCHE$(L, 5)$:

Recherche dans une liste chaînée

Entrée : une liste chaînée L et une clé x

Sortie : c tel que $c.key = x$ ou `null` si non trouvé

```
1: fonction RECHERCHE(L, x)
2:     c ← L.head
3:     tant que c ≠ null faire
4:         si c.key = x alors
5:             retourne c
6:         c ← c.next
7:     retourne non trouvé
```

Exemple : RECHERCHE$(L, 5)$:

Recherche dans une liste chaînée

Entrée : une liste chaînée L et une clé x

Sortie : c tel que $c.key = x$ ou `null` si non trouvé

```
1: fonction RECHERCHE(L, x)
2:     c ← L.head
3:     tant que c ≠ null faire
4:         si c.key = x alors
5:             retourne c
6:         c ← c.next
7:     retourne non trouvé
```

Exemple : RECHERCHE$(L, 5)$:

Recherche dans une liste chaînée

Entrée : une liste chaînée L et une clé x

Sortie : c tel que $c.key = x$ ou `null` si non trouvé

```
1: fonction RECHERCHE(L, x)
2:     c ← L.head
3:     tant que c ≠ null faire
4:         si c.key = x alors
5:             retourne c
6:         c ← c.next
7:     retourne non trouvé
```

Exemple : RECHERCHE$(L, 5)$:

Insertion d'un élément en tête de liste chaînée

Entrée : une liste chaînée L et une clé x

1: **fonction** InsertionTete(L, x)
2: $c \leftarrow$ `new` *Cellule*()
3: $c.key \leftarrow x$
4: $c.next \leftarrow L.head$
5: $L.head \leftarrow c$

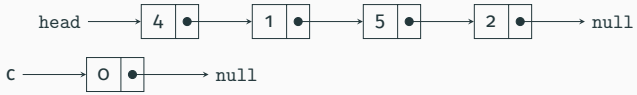
Exemple : InsertionTete($L, 3$):

Insertion d'un élément en tête de liste chaînée

Entrée : une liste chaînée L et une clé x

1: **fonction** InsertionTete(L, x)
2: $c \leftarrow$ `new` *Cellule*()
3: $c.key \leftarrow x$
4: $c.next \leftarrow L.head$
5: $L.head \leftarrow c$

Exemple : InsertionTete($L, 3$):

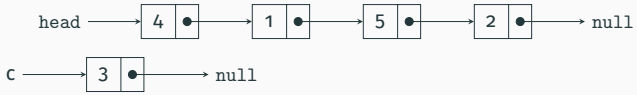


Insertion d'un élément en tête de liste chaînée

Entrée : une liste chaînée L et une clé x

1: **fonction** InsertionTete(L, x)
2: $c \leftarrow$ `new` *Cellule*()
3: $c.key \leftarrow x$
4: $c.next \leftarrow L.head$
5: $L.head \leftarrow c$

Exemple : InsertionTete($L, 3$):

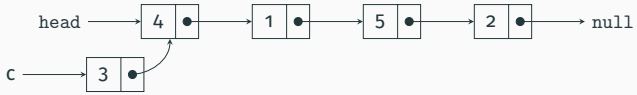


Insertion d'un élément en tête de liste chaînée

Entrée : une liste chaînée L et une clé x

1: **fonction** InsertionTete(L, x)
2: $c \leftarrow$ `new` *Cellule*()
3: $c.key \leftarrow x$
4: $c.next \leftarrow L.head$
5: $L.head \leftarrow c$

Exemple : InsertionTete($L, 3$):

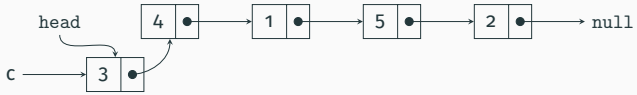


Insertion d'un élément en tête de liste chaînée

Entrée : une liste chaînée L et une clé x

1: **fonction** InsertionTete(L, x)
2: $c \leftarrow$ `new` *Cellule*()
3: $c.key \leftarrow x$
4: $c.next \leftarrow L.head$
5: $L.head \leftarrow c$

Exemple : InsertionTete($L, 3$):



Suppression d'un élément en tête de liste chaînée

Entrée : une liste chaînée L

1: **fonction** SUPPRESSIONELEMTETE(L)
2: $c \leftarrow L.head$
3: **si** $c \neq$ `null` **alors**
4: $L.head \leftarrow c.next$

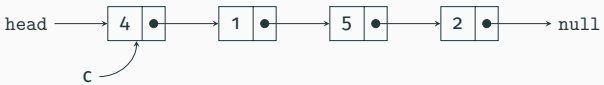
Exemple : SUPPRESSIONELEMTETE(L):

Suppression d'un élément en tête de liste chaînée

Entrée : une liste chaînée L

1: **fonction** SUPPRESSIONELEMTETE(L)
2: $c \leftarrow L.head$
3: **si** $c \neq$ `null` **alors**
4: $L.head \leftarrow c.next$

Exemple : SUPPRESSIONELEMTETE(L):

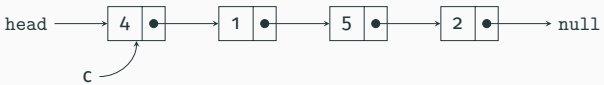


Suppression d'un élément en tête de liste chaînée

Entrée : une liste chaînée L

1: **fonction** SUPPRESSIONELEMTETE(L)
2: $c \leftarrow L.head$
3: **si** $c \neq$ `null` **alors**
4: $L.head \leftarrow c.next$

Exemple : SUPPRESSIONELEMTETE(L):



Suppression d'un élément en tête de liste chaînée

Entrée : une liste chaînée L

1: **fonction** SUPPRESSIONELEMTETE(L)
2: $c \leftarrow L.head$
3: **si** $c \neq$ `null` **alors**
4: $L.head \leftarrow c.next$

Exemple : SUPPRESSIONELEMTETE(L):