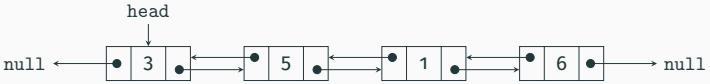
Elements d'Algorithmique
CMTD7 : Listes doublement chaînées et Piles

L2 UFR Informatique, Université Paris-Cité

Une liste doublement chaînée est une structure de données contenant des objets arrangés linéairement, telle que chaque nœud de la liste comprend :

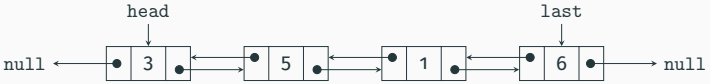
- un champ clé
- deux champs de pointeur, `next` et `prev`, pour respectivement, le nœud suivant et le nœud précédent de la liste.



On a un pointeur `head` vers le premier élément de la liste, appelé tête de liste.

Une liste doublement chaînée est une structure de données contenant des objets arrangés linéairement, telle que chaque nœud de la liste comprend :

- un champ clé
- deux champs de pointeur, `next` et `prev`, pour respectivement, le nœud suivant et le nœud précédent de la liste.



On a un pointeur `head` vers le premier élément de la liste, appelé tête de liste.

Il est parfois utile de maintenir un pointeur `last` vers le dernier élément de la liste.

Avantages par rapport à une liste chaînée simple

- Une liste doublement chaînée peut être parcourue aussi bien en avant qu'en arrière.
- L'opération de suppression dans une liste doublement chaînée est plus efficace si le pointeur vers le nœud à supprimer est donné.
- On peut rapidement insérer un nouveau nœud avant un nœud donné.

Avantages par rapport à une liste chaînée simple

- Une liste doublement chaînée peut être parcourue aussi bien en avant qu'en arrière.
- L'opération de suppression dans une liste doublement chaînée est plus efficace si le pointeur vers le nœud à supprimer est donné.
- On peut rapidement insérer un nouveau nœud avant un nœud donné.

Inconvénients par rapport à une liste chaînée simple

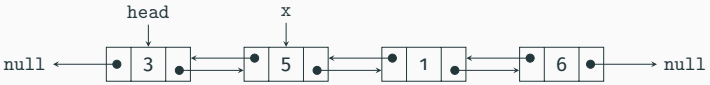
- Chaque nœud nécessite de la mémoire supplémentaire pour un pointeur `prev` vers le nœud précédent.
- Toutes les opérations nécessitent le maintien d'un pointeur supplémentaire.

Suppression dans une liste doublement chaînée

Entrée : une liste doublement chaînée `L` et un pointeur sur `x`

Sortie : la liste dans laquelle `x` a été supprimé

```
1: fonction SUPPRIMER(L, x)
2:     si x.prev ≠ null alors
3:         x.prev.next ← x.next
4:     sinon
5:         L.head ← x.next
6:     si x.next ≠ null alors
7:         x.next.prev ← x.prev
```

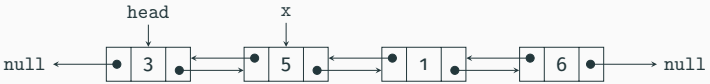


Suppression dans une liste doublement chaînée

Entrée : une liste doublement chaînée `L` et un pointeur sur `x`

Sortie : la liste dans laquelle `x` a été supprimé

```
1: fonction SUPPRIMER(L, x)
2:     si x.prev ≠ null alors
3:         x.prev.next ← x.next
4:     sinon
5:         L.head ← x.next
6:     si x.next ≠ null alors
7:         x.next.prev ← x.prev
```

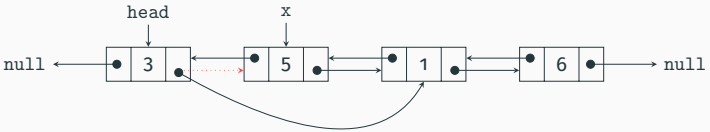


Suppression dans une liste doublement chaînée

Entrée : une liste doublement chaînée `L` et un pointeur sur `x`

Sortie : la liste dans laquelle `x` a été supprimé

```
1: fonction SUPPRIMER(L, x)
2:     si x.prev ≠ null alors
3:         x.prev.next ← x.next
4:     sinon
5:         L.head ← x.next
6:     si x.next ≠ null alors
7:         x.next.prev ← x.prev
```

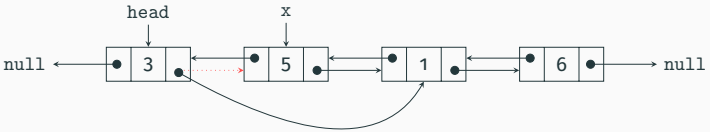


Suppression dans une liste doublement chaînée

Entrée : une liste doublement chaînée `L` et un pointeur sur `x`

Sortie : la liste dans laquelle `x` a été supprimé

```
1: fonction SUPPRIMER(L, x)
2:     si x.prev ≠ null alors
3:         x.prev.next ← x.next
4:     sinon
5:         L.head ← x.next
6:     si x.next ≠ null alors
7:         x.next.prev ← x.prev
```

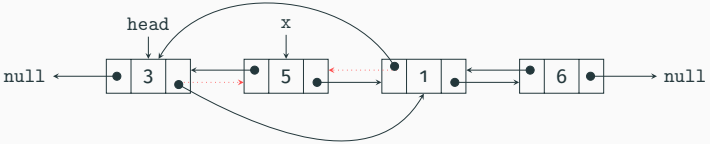


Suppression dans une liste doublement chaînée

Entrée : une liste doublement chaînée `L` et un pointeur sur `x`

Sortie : la liste dans laquelle `x` a été supprimé

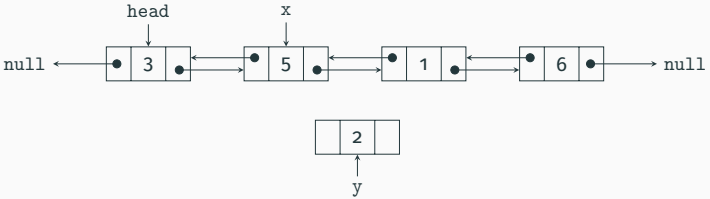
```
1: fonction SUPPRIMER(L, x)
2:     si x.prev ≠ null alors
3:         x.prev.next ← x.next
4:     sinon
5:         L.head ← x.next
6:     si x.next ≠ null alors
7:         x.next.prev ← x.prev
```



Insérer un nœud après un nœud donné

Entrée : une liste d.c. `L`, un pointeur `x` sur un nœud non null de `L` et un pointeur `y`
Sortie : la liste dans laquelle `y` a été inséré après `x`

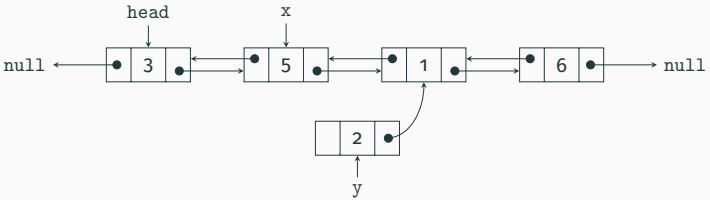
1: **fonction** INSERER(`L`, `x`, `y`)
2: `y.next` ← `x.next`
3: `y.prev` ← `x`
4: **si** `x.next` ≠ `null` **alors**
5: `x.next.prev` ← `y`
6: `x.next` ← `y`



Insérer un nœud après un nœud donné

Entrée : une liste d.c. `L`, un pointeur `x` sur un nœud non null de `L` et un pointeur `y`
Sortie : la liste dans laquelle `y` a été inséré après `x`

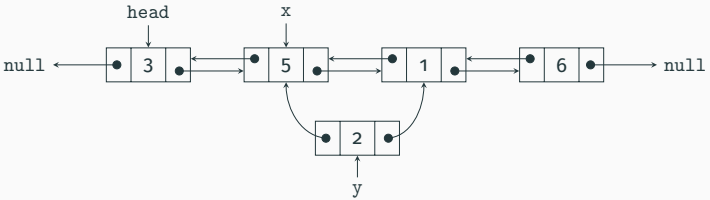
```
1: fonction INSERER(L, x, y)
2:     y.next ← x.next
3:     y.prev ← x
4:     si x.next ≠ null alors
5:         x.next.prev ← y
6:     x.next ← y
```



Insérer un nœud après un nœud donné

Entrée : une liste d.c. `L`, un pointeur `x` sur un nœud non null de `L` et un pointeur `y`
Sortie : la liste dans laquelle `y` a été inséré après `x`

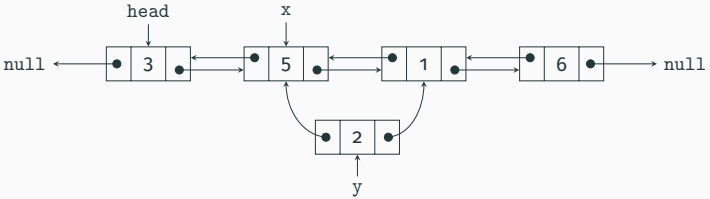
1: **fonction** INSERER(`L`, `x`, `y`)
2: `y.next` ← `x.next`
3: `y.prev` ← `x`
4: **si** `x.next` ≠ `null` **alors**
5: `x.next.prev` ← `y`
6: `x.next` ← `y`



Insérer un nœud après un nœud donné

Entrée : une liste d.c. `L`, un pointeur `x` sur un nœud non null de `L` et un pointeur `y`
Sortie : la liste dans laquelle `y` a été inséré après `x`

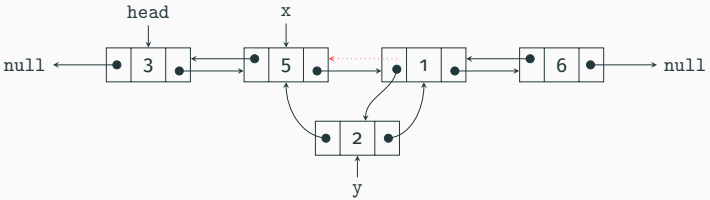
1: **fonction** INSERER(`L`, `x`, `y`)
2: `y.next` ← `x.next`
3: `y.prev` ← `x`
4: **si** `x.next` ≠ `null` **alors**
5: `x.next.prev` ← `y`
6: `x.next` ← `y`



Insérer un nœud après un nœud donné

Entrée : une liste d.c. `L`, un pointeur `x` sur un nœud non null de `L` et un pointeur `y`
Sortie : la liste dans laquelle `y` a été inséré après `x`

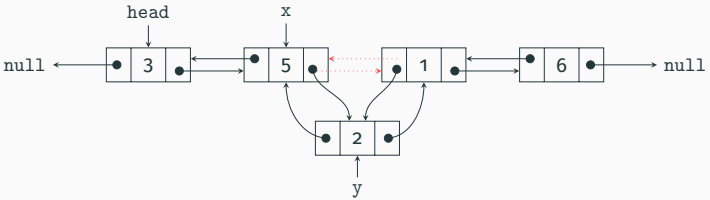
1: **fonction** INSERER(`L`, `x`, `y`)
2: `y.next` ← `x.next`
3: `y.prev` ← `x`
4: **si** `x.next` ≠ `null` **alors**
5: `x.next.prev` ← `y`
6: `x.next` ← `y`



Insérer un nœud après un nœud donné

Entrée : une liste d.c. `L`, un pointeur `x` sur un nœud non null de `L` et un pointeur `y`
Sortie : la liste dans laquelle `y` a été inséré après `x`

1: **fonction** INSERER(`L`, `x`, `y`)
2: `y.next` ← `x.next`
3: `y.prev` ← `x`
4: **si** `x.next` ≠ `null` **alors**
5: `x.next.prev` ← `y`
6: `x.next` ← `y`



Recherche dans une liste triée

Quiz. Quelle est la complexité de la recherche d'un élément dans une liste doublement chaînée triée ?

Recherche dans une liste triée

Quiz. Quelle est la complexité de la recherche d'un élément dans une liste doublement chaînée triée ?

Réponse : Toujours linéaire. Dans le pire des cas, nous devons parcourir toute la liste.

Recherche dans une liste triée

Quiz. Quelle est la complexité de la recherche d'un élément dans une liste doublement chaînée triée ?

Réponse : Toujours linéaire. Dans le pire des cas, nous devons parcourir toute la liste.

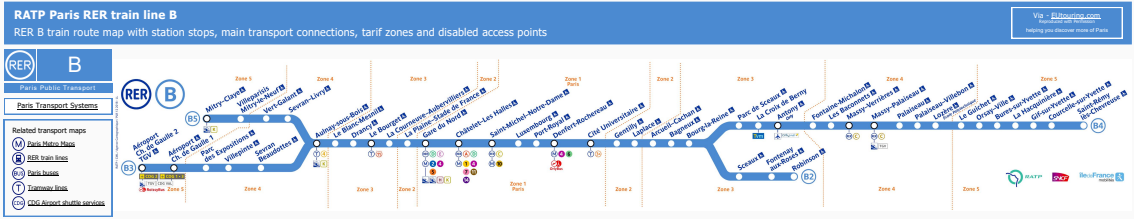
Peut-on faire mieux ? Par exemple, si nous utilisons deux listes ?

Recherche dans une liste triée

Quiz. Quelle est la complexité de la recherche d'un élément dans une liste doublement chaînée triée ?

Réponse : Toujours linéaire. Dans le pire des cas, nous devons parcourir toute la liste.

Peut-on faire mieux ? Par exemple, si nous utilisons deux listes ?



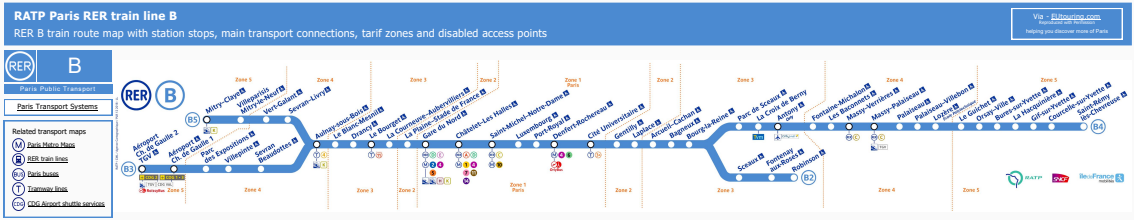
Il existe deux types de trains : normaux et express. Les trains express ne s'arrêtent pas à chaque station !

Recherche dans une liste triée

Quiz. Quelle est la complexité de la recherche d'un élément dans une liste doublement chaînée triée ?

Réponse : Toujours linéaire. Dans le pire des cas, nous devons parcourir toute la liste.

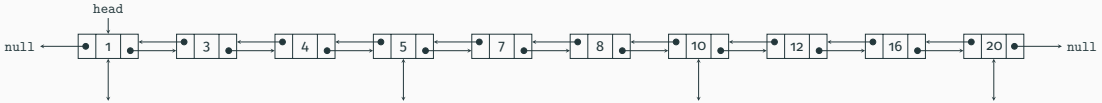
Peut-on faire mieux ? Par exemple, si nous utilisons deux listes ?



Il existe deux types de trains : normaux et express. Les trains express ne s'arrêtent pas à chaque station ! Pour aller de CDG à Arcueil, il est préférable de prendre un train express, disons jusqu'à Laplace, puis un train normal ensuite.

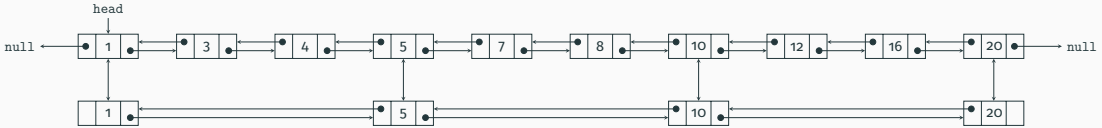
Recherche d'un élément en utilisant de deux listes chaînées

Nous pouvons utiliser une deuxième liste, dans laquelle nous copions certains des éléments de la première, répartis de manière uniforme.



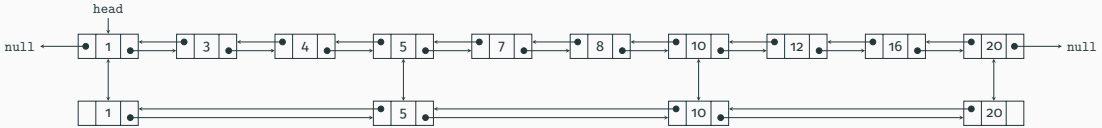
Recherche d'un élément en utilisant de deux listes chaînées

Nous pouvons utiliser une deuxième liste, dans laquelle nous copions certains des éléments de la première, répartis de manière uniforme.



Recherche d'un élément en utilisant de deux listes chaînées

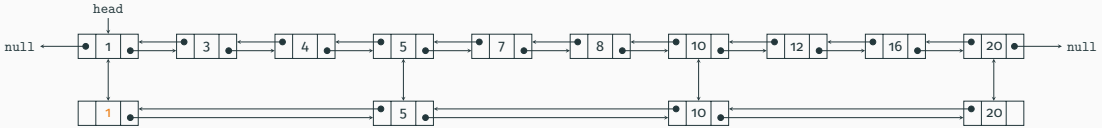
Nous pouvons utiliser une deuxième liste, dans laquelle nous copions certains des éléments de la première, répartis de manière uniforme.



Exemple : Recherchons 8 !

Recherche d'un élément en utilisant de deux listes chaînées

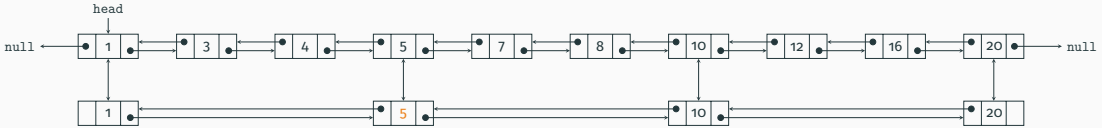
Nous pouvons utiliser une deuxième liste, dans laquelle nous copions certains des éléments de la première, répartis de manière uniforme.



Exemple : Recherchons 8 !

Recherche d'un élément en utilisant de deux listes chaînées

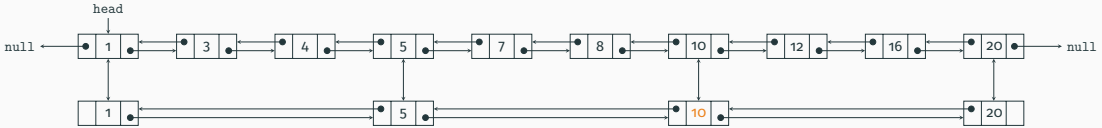
Nous pouvons utiliser une deuxième liste, dans laquelle nous copions certains des éléments de la première, répartis de manière uniforme.



Exemple : Recherchons 8 !

Recherche d'un élément en utilisant de deux listes chaînées

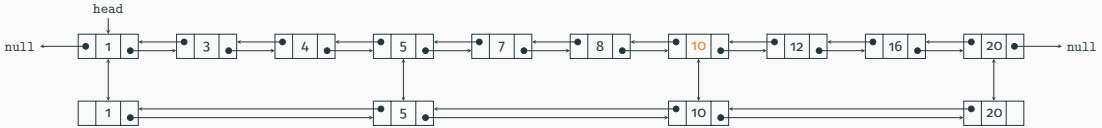
Nous pouvons utiliser une deuxième liste, dans laquelle nous copions certains des éléments de la première, répartis de manière uniforme.



Exemple : Recherchons 8 !

Recherche d'un élément en utilisant de deux listes chaînées

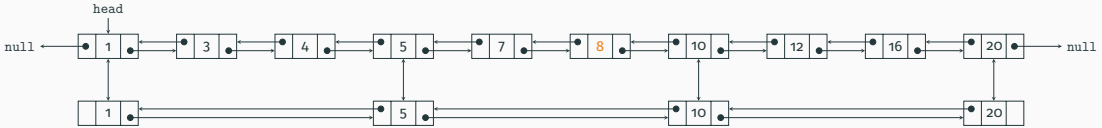
Nous pouvons utiliser une deuxième liste, dans laquelle nous copions certains des éléments de la première, répartis de manière uniforme.



Exemple : Recherchons 8 !

Recherche d'un élément en utilisant de deux listes chaînées

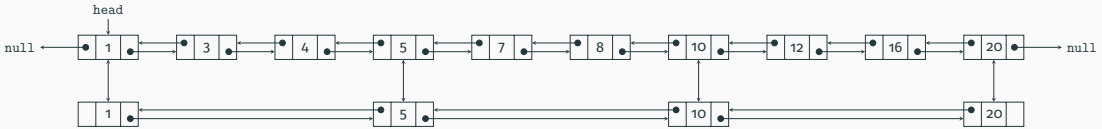
Nous pouvons utiliser une deuxième liste, dans laquelle nous copions certains des éléments de la première, répartis de manière uniforme.



Exemple : Recherchons 8 !

Recherche d'un élément en utilisant de deux listes chaînées

Nous pouvons utiliser une deuxième liste, dans laquelle nous copions certains des éléments de la première, répartis de manière uniforme.

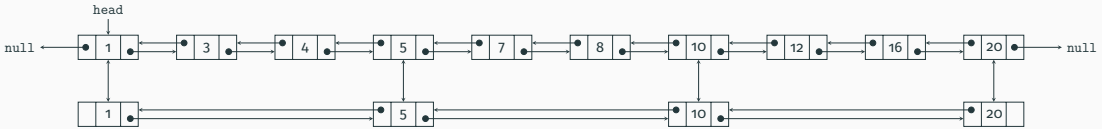


Exemple : Recherchons 8 !

Si la première liste a n éléments, et la seconde m éléments, alors la recherche d'un élément a une complexité $\frac{n}{m} + m$.

Recherche d'un élément en utilisant de deux listes chaînées

Nous pouvons utiliser une deuxième liste, dans laquelle nous copions certains des éléments de la première, répartis de manière uniforme.

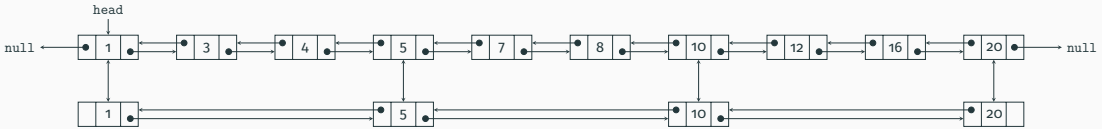


Exemple : Recherchons 8 !

Si la première liste a n éléments, et la seconde m éléments, alors la recherche d'un élément a une complexité $\frac{n}{m} + m$. Quelle est la valeur optimale de m ?

Recherche d'un élément en utilisant de deux listes chaînées

Nous pouvons utiliser une deuxième liste, dans laquelle nous copions certains des éléments de la première, répartis de manière uniforme.

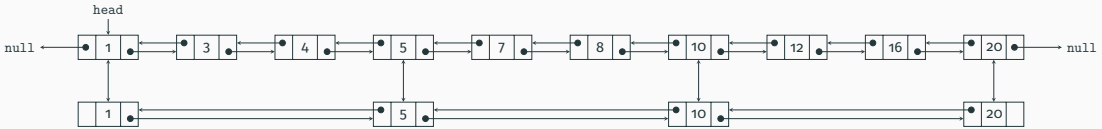


Exemple : Recherchons 8 !

Si la première liste a n éléments, et la seconde m éléments, alors la recherche d'un élément a une complexité $\frac{n}{m} + m$. Quelle est la valeur optimale de m ? Réponse : $\sqrt{n}$

Recherche d'un élément en utilisant de deux listes chaînées

Nous pouvons utiliser une deuxième liste, dans laquelle nous copions certains des éléments de la première, répartis de manière uniforme.



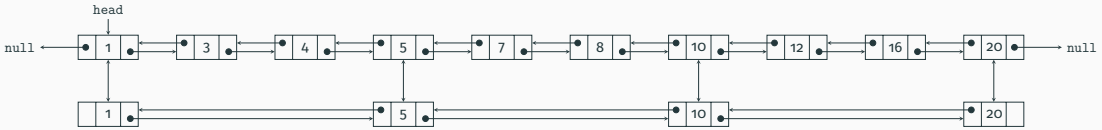
Exemple : Recherchons 8 !

Si la première liste a n éléments, et la seconde m éléments, alors la recherche d'un élément a une complexité $\frac{n}{m} + m$. Quelle est la valeur optimale de m ? Réponse : $\sqrt{n}$

La recherche d'un élément a une complexité $O(\sqrt{n})$ si l'on utilise une liste supplémentaire comme ci-dessus.

Recherche d'un élément en utilisant de deux listes chaînées

Nous pouvons utiliser une deuxième liste, dans laquelle nous copions certains des éléments de la première, répartis de manière uniforme.



Exemple : Recherchons 8 !

Si la première liste a n éléments, et la seconde m éléments, alors la recherche d'un élément a une complexité $\frac{n}{m} + m$. Quelle est la valeur optimale de m ? Réponse : $\sqrt{n}$

La recherche d'un élément a une complexité $O(\sqrt{n})$ si l'on utilise une liste supplémentaire comme ci-dessus.

Cette idée peut être répétée et on obtient une structure de données plus complexe : skip list, pour laquelle la recherche a une complexité $O(log(n))$.

Piles

Les piles sont des structures de données dynamiques représentant une séquence d'éléments ordonnés selon leur ordre d'insertion et dont nous ne pouvons supprimer que l'élément qui a été inséré le plus récemment.

La pile met en œuvre le principe dernier entré, premier sorti, ou LIFO (last in, first out).

Piles

Les piles sont des structures de données dynamiques représentant une séquence d'éléments ordonnés selon leur ordre d'insertion et dont nous ne pouvons supprimer que l'élément qui a été inséré le plus récemment.

La pile met en œuvre le principe dernier entré, premier sorti, ou LIFO (last in, first out).

Une pile est une structure de données abstraite sur laquelle sont définies trois opérations :

Piles

Les piles sont des structures de données dynamiques représentant une séquence d'éléments ordonnés selon leur ordre d'insertion et dont nous ne pouvons supprimer que l'élément qui a été inséré le plus récemment.

La pile met en œuvre le principe dernier entré, premier sorti, ou LIFO (last in, first out).

Une pile est une structure de données abstraite sur laquelle sont définies trois opérations :

- `empty(P)` qui teste si la pile `P` est vide ;

Piles

Les piles sont des structures de données dynamiques représentant une séquence d'éléments ordonnés selon leur ordre d'insertion et dont nous ne pouvons supprimer que l'élément qui a été inséré le plus récemment.

La pile met en œuvre le principe dernier entré, premier sorti, ou LIFO (last in, first out).

Une pile est une structure de données abstraite sur laquelle sont définies trois opérations :

- `empty(P)` qui teste si la pile `P` est vide ;
- `push(x,P)` qui ajoute un élément `x` au sommet de la pile `P`. Cette opération est également appelée `empiler` ;

Piles

Les piles sont des structures de données dynamiques représentant une séquence d'éléments ordonnés selon leur ordre d'insertion et dont nous ne pouvons supprimer que l'élément qui a été inséré le plus récemment.

La pile met en œuvre le principe dernier entré, premier sorti, ou LIFO (last in, first out).

Une pile est une structure de données abstraite sur laquelle sont définies trois opérations :

- `empty(P)` qui teste si la pile `P` est vide ;
- `push(x,P)` qui ajoute un élément `x` au sommet de la pile `P`. Cette opération est également appelée `empiler` ;
- `pop(P)` qui enlève la valeur au sommet de la pile `P` et la renvoie. Cette opération est aussi appelée `dépiler`.

Pile: structure de données abstraite et axiomes

Ensembles et fonctions:

Pile est l'ensemble des piles sur un ensemble E.

- `pilevide` $\in E$ est une constante
- `push`: $Pile \times E \longrightarrow Pile$
- `pop`: $Pile \longrightarrow Pile$
- `top`: $Pile \longrightarrow E$
- `empty`: $Pile \longrightarrow boolean$

Axiomes:

$\forall p \in Pile; \forall e \in E$:

- $\texttt{empty}(\texttt{pilevide}) = true$
- $\texttt{empty}(\texttt{push}(p, e) = false$
- $\texttt{pop}(\texttt{push}(p, e)) = p$
- $\texttt{top}(\texttt{push}(p, e)) = e$
- $\texttt{pop}(\texttt{pilevide})$ n'est pas défini
- $\texttt{top}(\texttt{pilevide})$ n'est pas défini

On peut vérifier que ces axiomes et définitions définissent entièrement une pile abstraite sans faire référence à une représentation.

Une pile concrète (par exemple en réalisant la pile avec une liste chaînée ou un tableau) est alors une réalisation de la pile qui satisfait les axiomes.

Piles : exemple

```
p:= new Pile();
p.push(1);
p.push(2);
print(p.pop());
p.push(3);
p.push(4);
p.push(1);
while(!p.empty()) {
    print(p.pop())
}
```

>

Piles : exemple

```
p:= new Pile();
p.push(1);
p.push(2);
print(p.pop());
p.push(3);
p.push(4);
p.push(1);
while(!p.empty()) {
    print(p.pop())
}
```

1

>

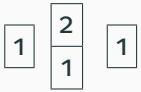
Piles : exemple

```
p:= new Pile();
p.push(1);
p.push(2);
print(p.pop());
p.push(3);
p.push(4);
p.push(1);
while(!p.empty()) {
    print(p.pop())
}
```

>

Piles : exemple

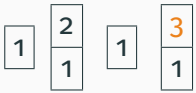
```
p:= new Pile();
p.push(1);
p.push(2);
print(p.pop());
p.push(3);
p.push(4);
p.push(1);
while(!p.empty()) {
    print(p.pop())
}
```



>2

Piles : exemple

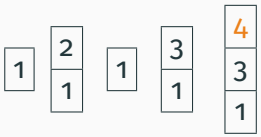
```
p:= new Pile();
p.push(1);
p.push(2);
print(p.pop());
p.push(3);
p.push(4);
p.push(1);
while(!p.empty()) {
    print(p.pop())
}
```



>2

Piles : exemple

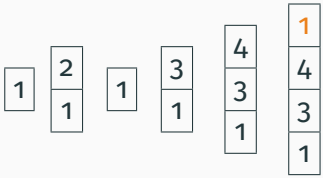
```
p:= new Pile();
p.push(1);
p.push(2);
print(p.pop());
p.push(3);
p.push(4);
p.push(1);
while(!p.empty()) {
    print(p.pop())
}
```



>2

Piles : exemple

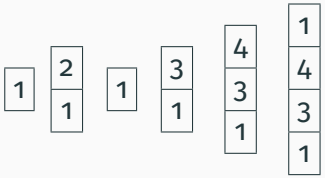
```
p:= new Pile();
p.push(1);
p.push(2);
print(p.pop());
p.push(3);
p.push(4);
p.push(1);
while(!p.empty()) {
    print(p.pop())
}
```



>2

Piles : exemple

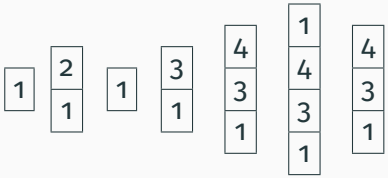
```
p:= new Pile();
p.push(1);
p.push(2);
print(p.pop());
p.push(3);
p.push(4);
p.push(1);
while(!p.empty()) {
    print(p.pop())
}
```



>2

Piles : exemple

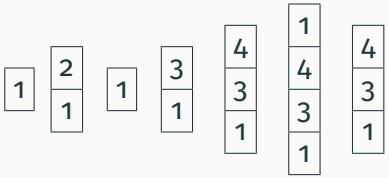
```
p:= new Pile();
p.push(1);
p.push(2);
print(p.pop());
p.push(3);
p.push(4);
p.push(1);
while(!p.empty()) {
    print(p.pop())
}
```




```
>2 1
```

Piles : exemple

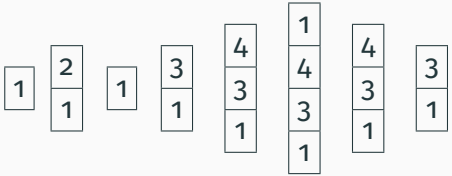
```
p:= new Pile();
p.push(1);
p.push(2);
print(p.pop());
p.push(3);
p.push(4);
p.push(1);
while(!p.empty()) {
    print(p.pop())
}
```



>2 1

Piles : exemple

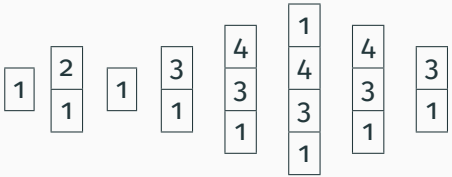
```
p:= new Pile();
p.push(1);
p.push(2);
print(p.pop());
p.push(3);
p.push(4);
p.push(1);
while(!p.empty()) {
    print(p.pop())
}
```



>2 1 4

Piles : exemple

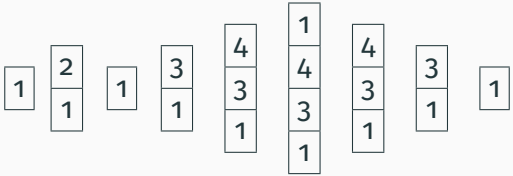
```
p:= new Pile();
p.push(1);
p.push(2);
print(p.pop());
p.push(3);
p.push(4);
p.push(1);
while(!p.empty()) {
    print(p.pop())
}
```



>2 1 4

Piles : exemple

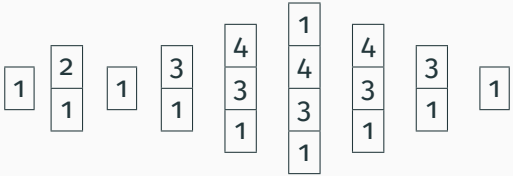
```
p:= new Pile();
p.push(1);
p.push(2);
print(p.pop());
p.push(3);
p.push(4);
p.push(1);
while(!p.empty()) {
    print(p.pop())
}
```



>2 1 4 3

Piles : exemple

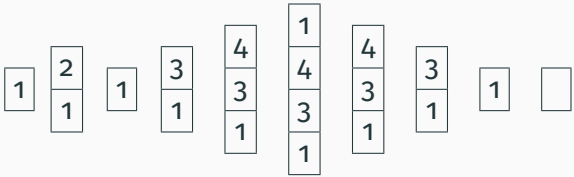
```
p:= new Pile();
p.push(1);
p.push(2);
print(p.pop());
p.push(3);
p.push(4);
p.push(1);
while(!p.empty()) {
    print(p.pop())
}
```



>2 1 4 3

Piles : exemple

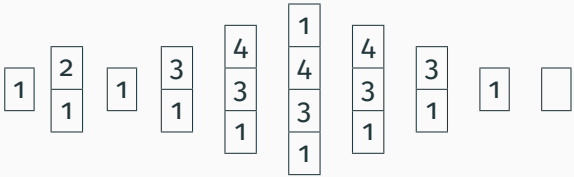
```
p:= new Pile();
p.push(1);
p.push(2);
print(p.pop());
p.push(3);
p.push(4);
p.push(1);
while(!p.empty()) {
    print(p.pop())
}
```



>2 1 4 3 1

Piles : exemple

```
p:= new Pile();
p.push(1);
p.push(2);
print(p.pop());
p.push(3);
p.push(4);
p.push(1);
while(!p.empty()) {
    print(p.pop())
}
```



>2 1 4 3 1

Piles

- Une pile est une structure de données dynamique avec des opérations (`pop`, `push`, `empty`) qui sont réalisées en temps constant
- C'est une structure de données facilement axiomatisable
- La pile est la structure de base qui permet de réaliser des appels de fonctions et la récursion
- La pile permet de mémoriser suivant l'ordre LIFO: inverser une chaine de caractère ou un tableau
- La pile permet l'exploration en profondeur et le backtracking