

STRUCTURES DE DONNÉES

Jérémy Cabessa
Laboratoire DAVID, UVSQ

STRUCTURES DE DONNÉES

- ▶ Une **structures de données (data structure)** permet de stocker en machine des *ensembles dynamiques de données*.
- ▶ Ensemble *dynamique* de données: qui peut évoluer au cours du temps. Insérer, modifier, supprimer des éléments...
- ▶ Exemples (Python): listes, tuples, dictionnaires, sets.

STRUCTURES DE DONNÉES

- ▶ Une **structures de données (data structure)** permet de stocker en machine des *ensembles dynamiques de données*.
- ▶ Ensemble *dynamique* de données: qui peut évoluer au cours du temps. Insérer, modifier, supprimer des éléments...
- ▶ Exemples (Python): listes, tuples, dictionnaires, sets.

STRUCTURES DE DONNÉES

- ▶ Une **structures de données (data structure)** permet de stocker en machine des *ensembles dynamiques de données*.
- ▶ Ensemble *dynamique* de données: qui peut évoluer au cours du temps. Insérer, modifier, supprimer des éléments...
- ▶ **Exemples (Python):** listes, tuples, dictionnaires, sets.

TABLEAUX

- Un **tableau (array)** est une *structure de données*.

indices	0	1	2	3	4	5	6	7
clés	12	-7	33	11	-42	1	0	8

- Représente un *ensemble dynamique, ordonné et de taille fixe* d'éléments de même types.
- Usage: rapidité de traitement en ce qui concerne l'accès aux éléments.

TABLEAUX

- Un **tableau (array)** est une *structure de données*.

indices	0	1	2	3	4	5	6	7
clés	12	-7	33	11	-42	1	0	8

- Représente un *ensemble dynamique, ordonné et de taille fixe* d'éléments de même types.
- Usage: rapidité de traitement en ce qui concerne l'accès aux éléments.

TABLEAUX

- Un **tableau (array)** est une *structure de données*.

indices	0	1	2	3	4	5	6	7
clés	12	-7	33	11	-42	1	0	8

- Représente un *ensemble dynamique, ordonné et de taille fixe* d'éléments de même types.
- Usage: rapidité de traitement en ce qui concerne l'accès aux éléments.

TABLEAUX

- ▶ Taille fixe.
- ▶ Accès aux éléments de manière *directe*, par adressage des cellules via des *indices* (simple).
`t[3]`, `t[3:7]`, ...
- ▶ Insertion et suppression d'éléments nécessitent la création d'un nouveau tableau (coûteux, procédure de "realloc").
`t.append(14)`, `t.pop()`, ...

TABLEAUX

- ▶ Taille fixe.
- ▶ Accès aux éléments de manière *directe*, par adressage des cellules via des *indices* (simple).

`t[3]`, `t[3:7]`, ...

- ▶ Insertion et suppression d'éléments nécessitent la création d'un nouveau tableau (coûteux, procédure de "realloc").

`t.append(14)`, `t.pop()`, ...

TABLEAUX

- ▶ Taille fixe.
- ▶ Accès aux éléments de manière *directe*, par adressage des cellules via des *indices* (simple).
`t[3]`, `t[3:7]`, ...
- ▶ Insertion et suppression d'éléments nécessitent la création d'un nouveau tableau (coûteux, procédure de “realloc”).
`t.append(14)`, `t.pop()`, ...

TABLEAUX

- ▶ **Principe:** les éléments sont stockés en mémoire de façon *contigüe et accessibles via un indice*.
- ▶ Les *éléments* du tableaux sont des *objets* qui possèdent une *clé* et un *indice* (et une adresse mémoire).
- ▶ Les clés sont accessibles via leurs indices.



TABLEAUX

- ▶ **Principe:** les éléments sont stockés en mémoire de façon *contigüe et accessibles via un indice*.
- ▶ Les *éléments* du tableaux sont des *objets* qui possèdent une *clé* et un *indice* (et une adresse mémoire).
- ▶ Les clés sont accessibles via leurs indices.

élément x

x.cle	x.indice
-------	----------

TABLEAUX

- ▶ **Principe:** les éléments sont stockés en mémoire de façon *contigüe et accessibles via un indice*.
- ▶ Les *éléments* du tableaux sont des *objets* qui possèdent une *clé* et un *indice* (et une adresse mémoire).
- ▶ Les clés sont accessibles via leurs indices.

élément x

x.cle	x.indice
-------	----------

TABLEAUX: RECHERCHER, INSERER, SUPPRIMER EN $O(n)$

```
1 def rechercher(T,x):
2     for i in range(len(T)):
3         if T[i] == x:
4             return x
5     return None
```

```
1 def inserer(T, x, y):
2     """On suppose que T a la place pour inserer..."""
3     i = T.index(y)
4     for j in range(len(T)-1, i, -1):
5         T[j] = T[j-1]
6     T[i] = x
7     return T
```

```
1 def supprimer(T, x):
2     i = T.index(x)
3     for j in range(i, len(T)-1):
4         T[j] = T[j+1]
5     T[len(T)-1] = None
6     return T
```

TABLEAUX: RECHERCHER, INSERER, SUPPRIMER EN $O(n)$

```
1 def rechercher(T,x):
2     for i in range(len(T)):
3         if T[i] == x:
4             return x
5     return None
```

```
1 def inserer(T, x, y):
2     """On suppose que T a la place pour inserer..."""
3     i = T.index(y)
4     for j in range(len(T)-1, i, -1):
5         T[j] = T[j-1]
6     T[i] = x
7     return T
```

```
1 def supprimer(T, x):
2     i = T.index(x)
3     for j in range(i, len(T)-1):
4         T[j] = T[j+1]
5     T[len(T)-1] = None
6     return T
```

TABLEAUX: RECHERCHER, INSERER, SUPPRIMER EN $O(n)$

```
1 def rechercher(T,x):
2     for i in range(len(T)):
3         if T[i] == x:
4             return x
5     return None
```

```
1 def inserer(T, x, y):
2     """On suppose que T a la place pour inserer..."""
3     i = T.index(y)
4     for j in range(len(T)-1, i, -1):
5         T[j] = T[j-1]
6     T[i] = x
7     return T
```

```
1 def supprimer(T, x):
2     i = T.index(x)
3     for j in range(i, len(T)-1):
4         T[j] = T[j+1]
5     T[len(T)-1] = None
6     return T
```


REMARQUES

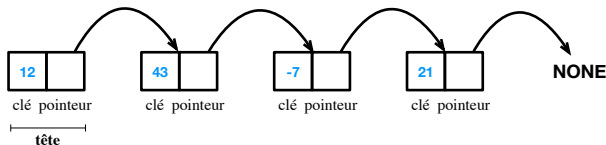
- ▶ L'accès aux éléments est simple!
- ▶ Les procédures de recherche, insertion et suppression sont complexes.

REMARQUES

- ▶ L'accès aux éléments est simple!
- ▶ Les procédures de recherche, insertion et suppression sont complexes.

LISTES CHAÎNÉES

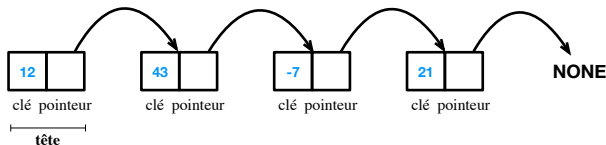
- Une liste chaînée (linked list) est une structure de données.



- Représente un *ensemble dynamique, ordonné et de taille arbitraire* d'éléments de même types.
- Usage: rapidité de traitement, lorsque l'ordre des éléments est important et que les insertions et les suppressions d'éléments sont plus fréquentes que les accès.

LISTES CHAÎNÉES

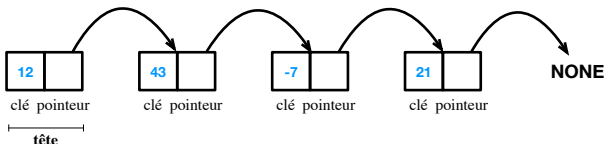
- Une liste chaînée (linked list) est une structure de données.



- Représente un *ensemble dynamique, ordonné et de taille arbitraire* d'éléments de même types.
- Usage: rapidité de traitement, lorsque l'ordre des éléments est important et que les insertions et les suppressions d'éléments sont plus fréquentes que les accès.

LISTES CHAÎNÉES

- ▶ Une liste chaînée (linked list) est une structure de données.



- ▶ Représente un *ensemble dynamique, ordonné et de taille arbitraire* d'éléments de même types.
- ▶ Usage: rapidité de traitement, lorsque l'ordre des éléments est important et que les insertions et les suppressions d'éléments sont plus fréquentes que les accès.

LISTES CHAÎNÉES

- ▶ Taille arbitraire, limitée par la mémoire.
- ▶ Accès aux éléments de manière *séquentielle* via une suite de *pointeurs* (coûteux).
- ▶ Insertion et suppression en début de liste d'éléments en temps constant (simple).

LISTES CHAÎNÉES

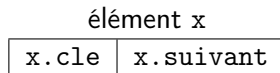
- ▶ Taille arbitraire, limitée par la mémoire.
- ▶ Accès aux éléments de manière *séquentielle* via une suite de *pointeurs* (coûteux).
- ▶ Insertion et suppression en début de liste d'éléments en temps constant (simple).

LISTES CHAÎNÉES

- ▶ Taille arbitraire, limitée par la mémoire.
- ▶ Accès aux éléments de manière *séquentielle* via une suite de *pointeurs* (coûteux).
- ▶ Insertion et suppression en début de liste d'éléments en temps constant (simple).

LISTES CHAÎNÉES

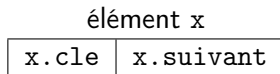
- **Principe:** les *éléments* de la liste sont des *objets* qui possèdent une *clé* et un *pointeur* vers l'élément suivant dans la liste.



- Le dernier élément pointe vers `NONE`.
- La liste chaînée peut comporter un attribut `tete` qui désigne le premier élément de la liste.

LISTES CHAÎNÉES

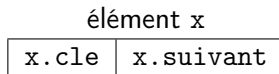
- ▶ **Principe:** les *éléments* de la liste sont des *objets* qui possèdent une *clé* et un *pointeur* vers l'élément suivant dans la liste.



- ▶ Le dernier élément pointe vers NONE.
- ▶ La liste chaînée peut comporter un attribut *tete* qui désigne le premier élément de la liste.

LISTES CHAÎNÉES

- **Principe:** les *éléments* de la liste sont des *objets* qui possèdent une *clé* et un *pointeur* vers l'élément suivant dans la liste.



- Le dernier élément pointe vers NONE.
- La liste chaînée peut comporter un attribut *tete* qui désigne le premier élément de la liste.

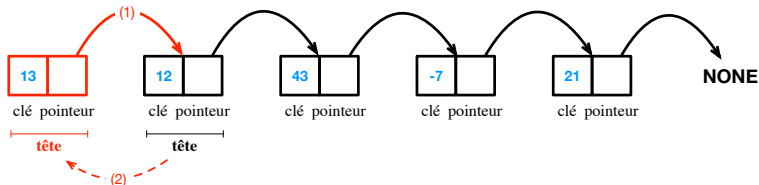
IMPLÉMENTATION: ÉLÉMENT DE LC (CELLULE)

```
1 class Cellule:
2     def __init__(self, valeur, suivant=None):
3         """creation d'une cellule
4
5         Args:
6             valeur (quelconque): valeur stockee
7             suiv (Cellule): cellule suivante dans la LC
8         """
9
10        self.valeur = valeur
11        self.suiv = suivant
```

IMPLÉMENTATION: LC

```
1 class LC:
2
3     def __init__(self):
4         """creation d'une liste chainee"""
5
6         self.tete = None # premier element de la liste
7
8     def est_vide(self):
9         """renvoie un booléen indiquant si
10        la liste est vide"""
11
12        return self.tete is None
```

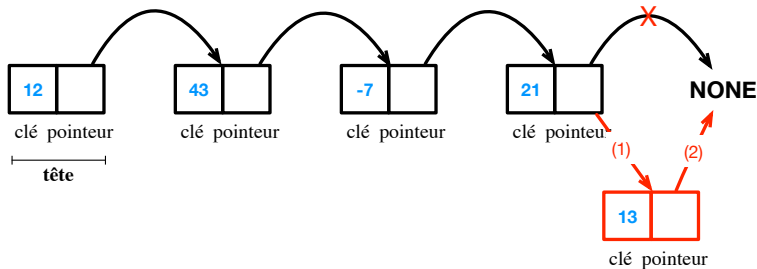
LISTES CHAÎNÉES: AJOUTER_DEBUT



LISTES CHAÎNÉES: AJOUTER_DEBUT

```
1  def ajouter_debut(self, x):  
2      """Ajoute x en tete de la liste"""  
3  
4      x.suiv = self.tete  
5      self.tete = x
```

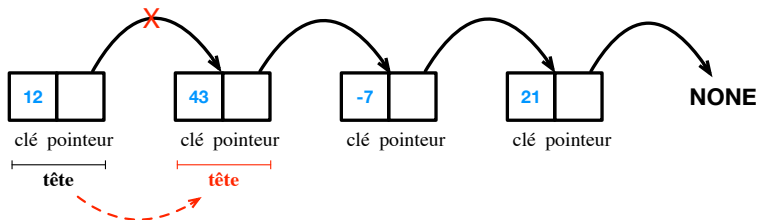
LISTES CHAÎNÉES: AJOUTER_FIN



LISTES CHAÎNÉES: AJOUTER_FIN

```
1  def ajouter_fin(self, x):  
2      """Ajoute le x en queue de la liste"""  
3  
4      m = self.tete  
5      while m.suiv is not None:  
6          m = m.suiv  
7      m.suiv = x # x.suiv = None par default
```

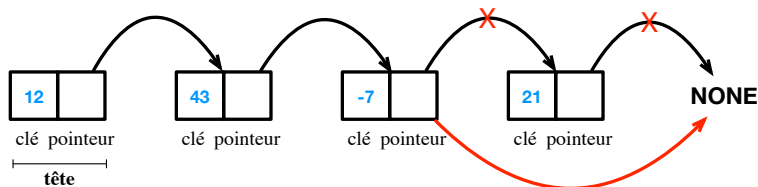
LISTES CHAÎNÉES: SUPPRIMER_DEBUT



LISTES CHAÎNÉES: SUPPRIMER_DEBUT

```
1  def supprimer_debut(self):  
2      """Supprime le 1er element de la liste  
3      et le renvoie"""  
4  
5      if self.est_vide():  
6          raise Exception("La liste est vide !")  
7      t = self.tete  
8      self.tete = t.suiv  
9      return t
```

LISTES CHAÎNÉES: SUPPRIMER_FIN



LISTES CHAÎNÉES: SUPPRIMER_FIN

```
1  def supprimer_fin(self):
2      """Supprime le dernier element de la liste
3      et le renvoie"""
4
5      if self.est_vide():                # pile vide
6          raise Exception("La pile est vide !")
7      elif self.tete.suiv is None:      # 1 seul element
8          e = self.tete
9          self.tete = None
10         return e
11     else:                              # au moins 2 elements
12         e = self.tete
13         e_courant = self.tete
14         while e.suiv is not None:
15             e_courant = e
16             e = e.suiv
17         e_courant.suiv = None
18         return e
```

LISTES CHAÎNÉES: RECHERCHER

```
1  def rechercher(self, valeur):
2      """Recherche valeur dans la liste
3      et renvoie l'element associe"""
4
5      if self.est_vide():
6          raise Exception("La pile est vide !")
7      else:
8          e = self.tete
9          while e is not None:
10             if e.valeur == valeur:
11                 return e
12             else:
13                 e = e.suiv
```

REMARQUES

- ▶ Les insertions et suppression en tête de liste sont simples!
- ▶ Des piles et files *de tailles arbitraires* peuvent être implémentées par des listes chaînées (TP).
- ▶ Dans le cas de la file, plus besoin du concept d'assignation circulaire des éléments (puisque taille arbitraire).

REMARQUES

- ▶ Les insertions et suppression en tête de liste sont simples!
- ▶ Des piles et files *de tailles arbitraires* peuvent être implémentées par des listes chaînées (TP).
- ▶ Dans le cas de la file, plus besoin du concept d'assignation circulaire des éléments (puisque taille arbitraire).

REMARQUES

- ▶ Les insertions et suppression en tête de liste sont simples!
- ▶ Des piles et files *de tailles arbitraires* peuvent être implémentées par des listes chaînées (TP).
- ▶ Dans le cas de la file, plus besoin du concept d'assignation circulaire des éléments (puisque taille arbitraire).

COMPLEXITÉ DES OPÉRATIONS

Opérations	Tableaux	Liste chaînées
rechercher	$O(n)$	$O(n)$
ajouter/insérer	$O(n)$	$O(1)/O(n)$ (début/fin de liste)
supprimer	$O(n)$	$O(1)/O(n)$ (début/fin de liste)
min/max	$O(n)$	$O(n)$
accéder	$O(1)$	$O(n)$

PILES ET FILES

- ▶ Une **pile (stack)** et une **file (queue)** sont des structures de données.

- ▶ Représentent des *ensembles dynamiques, ordonnés et de tailles arbitraires* d'éléments de même types.

- ▶ Une pile ou une file *S* comporte trois opérations (une requête et deux opérations pour être précis):

est_vide(S): teste si *S* est vide (requête);

ajouter(S,x): ajoute l'élément *x* dans *S* (opération);

supprimer(S): supprime l'élément de *S* (opération).

- ▶ **Remarque:** Pour *ajouter*, on spécifie l'élément *x* à ajouter. Pour *supprimer*, pas d'élément *x* spécifié. Déterminé par la nature intrinsèque de l'ensemble: pile ou file (cf. suite).

PILES ET FILES

- ▶ Une **pile (stack)** et une **file (queue)** sont des structures de données.
- ▶ Représentent des *ensembles dynamiques, ordonnés et de tailles arbitraires* d'éléments de même types.

- ▶ Une pile ou une file *S* comporte trois opérations (une requête et deux opérations pour être précis):

est_vide(S): teste si *S* est vide (requête);

ajouter(S,x): ajoute l'élément *x* dans *S* (opération);

supprimer(S): supprime l'élément de *S* (opération).

- ▶ **Remarque:** Pour *ajouter*, on spécifie l'élément *x* à ajouter. Pour *supprimer*, pas d'élément *x* spécifié. Déterminé par la nature intrinsèque de l'ensemble: pile ou file (cf. suite).

PILES ET FILES

- ▶ Une **pile (stack)** et une **file (queue)** sont des structures de données.
- ▶ Représentent des *ensembles dynamiques, ordonnés et de tailles arbitraires* d'éléments de même types.
- ▶ Une pile ou une file S comporte trois opérations (une requête et deux opérations pour être précis):

`est_vide(S)`: teste si S est vide (requête);

`ajouter(S,x)`: ajoute l'élément x dans S (opération);

`supprimer(S)`: supprime l'élément de S (opération).

- ▶ **Remarque:** Pour `ajouter`, on spécifie l'élément x à ajouter. Pour `supprimer`, pas d'élément x spécifié. Déterminé par la nature intrinsèque de l'ensemble: pile ou file (cf. suite).

PILES ET FILES

- ▶ Une **pile (stack)** et une **file (queue)** sont des structures de données.
- ▶ Représentent des *ensembles dynamiques, ordonnés et de tailles arbitraires* d'éléments de même types.
- ▶ Une pile ou une file S comporte trois opérations (une requête et deux opérations pour être précis):

`est_vide(S)`: teste si S est vide (requête);

`ajouter(S,x)`: ajoute l'élément x dans S (opération);

`supprimer(S)`: supprime l'élément de S (opération).

- ▶ **Remarque:** Pour `ajouter`, on spécifie l'élément x à ajouter. Pour `supprimer`, pas d'élément x spécifié. Déterminé par la nature intrinsèque de l'ensemble: pile ou file (cf. suite).

PILES ET FILES

- ▶ Une **pile (stack)** et une **file (queue)** sont des structures de données.
- ▶ Représentent des *ensembles dynamiques, ordonnés et de tailles arbitraires* d'éléments de même types.
- ▶ Une pile ou une file S comporte trois opérations (une requête et deux opérations pour être précis):

`est_vide(S)`: teste si S est vide (requête);

`ajouter(S,x)`: ajoute l'élément x dans S (opération);

`supprimer(S)`: supprime l'élément de S (opération).

- ▶ **Remarque:** Pour `ajouter`, on spécifie l'élément x à ajouter. Pour `supprimer`, pas d'élément x spécifié. Déterminé par la nature intrinsèque de l'ensemble: pile ou file (cf. suite).

PILES ET FILES

- ▶ Une **pile (stack)** et une **file (queue)** sont des structures de données.
- ▶ Représentent des *ensembles dynamiques, ordonnés et de tailles arbitraires* d'éléments de même types.
- ▶ Une pile ou une file S comporte trois opérations (une requête et deux opérations pour être précis):

`est_vide(S)`: teste si S est vide (requête);

`ajouter(S,x)`: ajoute l'élément x dans S (opération);

`supprimer(S)`: supprime l'élément de S (opération).

- ▶ **Remarque:** Pour `ajouter`, on spécifie l'élément x à ajouter. Pour `supprimer`, pas d'élément x spécifié. Déterminé par la nature intrinsèque de l'ensemble: pile ou file (cf. suite).

PILES ET FILES

- ▶ Une **pile (stack)** et une **file (queue)** sont des structures de données.
- ▶ Représentent des *ensembles dynamiques, ordonnés et de tailles arbitraires* d'éléments de même types.
- ▶ Une pile ou une file S comporte trois opérations (une requête et deux opérations pour être précis):

`est_vide(S)`: teste si S est vide (requête);

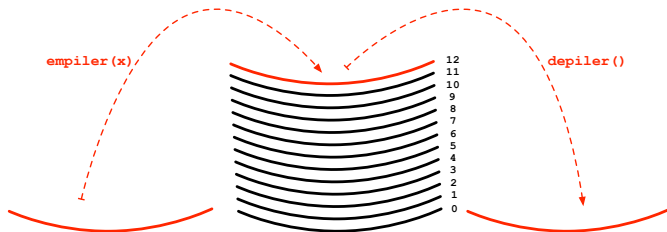
`ajouter(S,x)`: ajoute l'élément x dans S (opération);

`supprimer(S)`: supprime l'élément de S (opération).

- ▶ **Remarque:** Pour ajouter, on spécifie l'élément x à ajouter. Pour supprimer, pas d'élément x spécifié. Déterminé par la nature intrinsèque de l'ensemble: pile ou file (cf. suite).

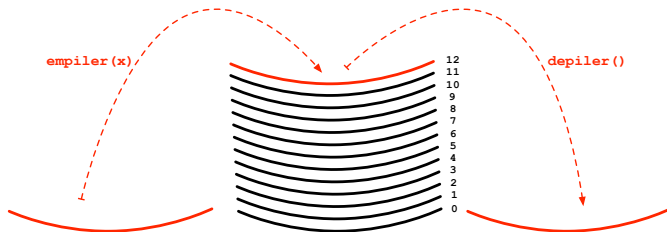
PILES

- ▶ Une **pile (stack)** implémente le principe:
dernier entré, premier sorti, Last In First Out, (LIFO)
- ▶ Comme des assiettes sales que l'on empile: la dernière assiette empilée sera la première à être retirée pour être lavée.



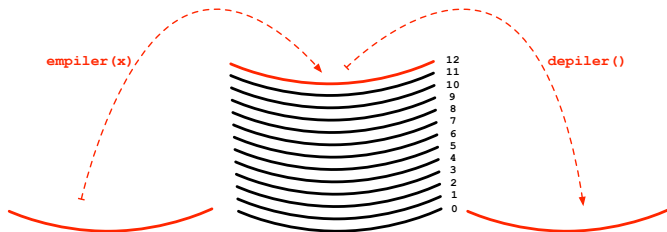
PILES

- ▶ Une **pile (stack)** implémente le principe:
dernier entré, premier sorti, Last In First Out, (LIFO)
- ▶ Comme des assiettes sales que l'on empile: la dernière assiette empilée sera la première à être retirée pour être lavée.



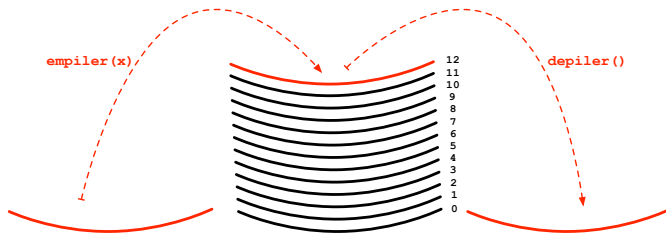
PILES

- ▶ Une **pile (stack)** implémente le principe:
dernier entré, premier sorti, Last In First Out, (LIFO)
- ▶ Comme des assiettes sales que l'on empile: la dernière assiette empilée sera la première à être retirée pour être lavée.



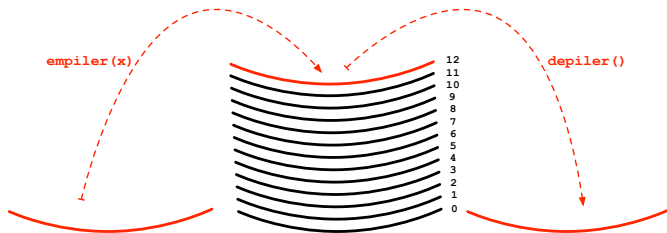
PILES: APPLICATIONS

- ▶ undo / redo dans des programmes;
- ▶ page précédente / page suivante dans une navigation internet;
- ▶ récursivité (pile d'appels de fonctions).



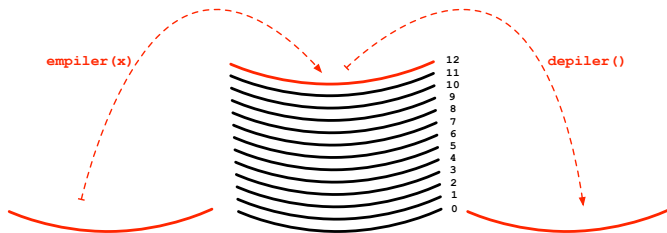
PILES: APPLICATIONS

- ▶ undo / redo dans des programmes;
- ▶ page précédente / page suivante dans une navigation internet;
- ▶ récursivité (pile d'appels de fonctions).



PILES: APPLICATIONS

- ▶ undo / redo dans des programmes;
- ▶ page précédente / page suivante dans une navigation internet;
- ▶ récursivité (pile d'appels de fonctions).



PILES

- ▶ Une **pile** d'au plus n éléments peut être implémentée par un tableau $P[0, \dots, n-1]$ ($0, \dots, n-1$ sont les indices des clés).
- ▶ La pile possède un attribut `P.sommet` qui est l'indice du dernier élément inséré (indice et non élément).
- ▶ Si `P.sommet == -1`, alors la pile est vide.

C'est la requête `est_vide`.

- ▶ Pour insérer un élément x dans P , on assigne $P[P.sommet] = x$ et on incrémente `P.sommet`.

C'est l'opération **empiler** ou **push**.

- ▶ Pour supprimer l'élément en haut de la pile P , on décrémente `P.sommet` et on retourne $P[P.sommet+1]$.

C'est l'opération **dépiler** ou **pop**.

PILES

- ▶ Une **pile** d'au plus n éléments peut être implémentée par un tableau $P[0, \dots, n-1]$ ($0, \dots, n-1$ sont les indices des clés).
- ▶ La pile possède un attribut `P.sommet` qui est l'indice du dernier élément inséré (indice et non élément).
- ▶ Si `P.sommet == -1`, alors la pile est vide.

C'est la requête `est_vide`.

- ▶ Pour insérer un élément x dans P , on assigne `P[P.sommet] = x` et on incrémente `P.sommet`.

C'est l'opération **empiler** ou **push**.

- ▶ Pour supprimer l'élément en haut de la pile P , on décrémente `P.sommet` et on retourne `P[P.sommet+1]`.

C'est l'opération **dépiler** ou **pop**.

PILES

- ▶ Une **pile** d'au plus n éléments peut être implémentée par un tableau $P[0, \dots, n-1]$ ($0, \dots, n-1$ sont les indices des clés).
- ▶ La pile possède un attribut `P.sommet` qui est l'indice du dernier élément inséré (indice et non élément).
- ▶ Si `P.sommet == -1`, alors la pile est vide.

C'est la requête `est_vide`.

- ▶ Pour insérer un élément x dans P , on assigne `P[P.sommet] = x` et on incrémente `P.sommet`.

C'est l'opération **empiler** ou **push**.

- ▶ Pour supprimer l'élément en haut de la pile P , on décrémente `P.sommet` et on retourne `P[P.sommet+1]`.

C'est l'opération **dépiler** ou **pop**.

PILES

- ▶ Une **pile** d'au plus n éléments peut être implémentée par un tableau $P[0, \dots, n-1]$ ($0, \dots, n-1$ sont les indices des clés).
- ▶ La pile possède un attribut `P.sommet` qui est l'indice du dernier élément inséré (indice et non élément).
- ▶ Si `P.sommet == -1`, alors la pile est vide.

C'est la requête **est_vide**.

- ▶ Pour insérer un élément x dans P , on assigne $P[P.sommet] = x$ et on incrémente `P.sommet`.

C'est l'opération **empiler** ou **push**.

- ▶ Pour supprimer l'élément en haut de la pile P , on décrémente `P.sommet` et on retourne $P[P.sommet+1]$.

C'est l'opération **dépiler** ou **pop**.

PILES

- ▶ Une **pile** d'au plus n éléments peut être implémentée par un tableau $P[0, \dots, n-1]$ ($0, \dots, n-1$ sont les indices des clés).
- ▶ La pile possède un attribut `P.sommet` qui est l'indice du dernier élément inséré (indice et non élément).
- ▶ Si `P.sommet == -1`, alors la pile est vide.

C'est la requête **est_vide**.

- ▶ Pour insérer un élément x dans P , on assigne $P[P.sommet] = x$ et on incrémente `P.sommet`.

C'est l'opération **empiler** ou **push**.

- ▶ Pour supprimer l'élément en haut de la pile P , on décrémente `P.sommet` et on retourne $P[P.sommet+1]$.

C'est l'opération **dépiler** ou **pop**.

PILES

- ▶ Une **pile** d'au plus n éléments peut être implémentée par un tableau $P[0, \dots, n-1]$ ($0, \dots, n-1$ sont les indices des clés).
- ▶ La pile possède un attribut `P.sommet` qui est l'indice du dernier élément inséré (indice et non élément).
- ▶ Si `P.sommet == -1`, alors la pile est vide.

C'est la requête **est_vide**.

- ▶ Pour insérer un élément x dans P , on assigne $P[P.sommet] = x$ et on incrémente `P.sommet`.

C'est l'opération **empiler** ou **push**.

- ▶ Pour supprimer l'élément en haut de la pile P , on décrémente `P.sommet` et on retourne $P[P.sommet+1]$.

C'est l'opération **dépiler** ou **pop**.

PILES

- ▶ Une **pile** d'au plus n éléments peut être implémentée par un tableau $P[0, \dots, n-1]$ ($0, \dots, n-1$ sont les indices des clés).
- ▶ La pile possède un attribut $P.sommet$ qui est l'indice du dernier élément inséré (indice et non élément).
- ▶ Si $P.sommet == -1$, alors la pile est vide.

C'est la requête **est_vide**.

- ▶ Pour insérer un élément x dans P , on assigne $P[P.sommet] = x$ et on incrémente $P.sommet$.

C'est l'opération **empiler** ou **push**.

- ▶ Pour supprimer l'élément en haut de la pile P , on décrémente $P.sommet$ et on retourne $P[P.sommet+1]$.

C'est l'opération **dépiler** ou **pop**.

PILES

- ▶ Une **pile** d'au plus n éléments peut être implémentée par un tableau $P[0, \dots, n-1]$ ($0, \dots, n-1$ sont les indices des clés).
- ▶ La pile possède un attribut `P.sommet` qui est l'indice du dernier élément inséré (indice et non élément).
- ▶ Si `P.sommet == -1`, alors la pile est vide.

C'est la requête **est_vide**.

- ▶ Pour insérer un élément x dans P , on assigne `P[P.sommet] = x` et on incrémente `P.sommet`.

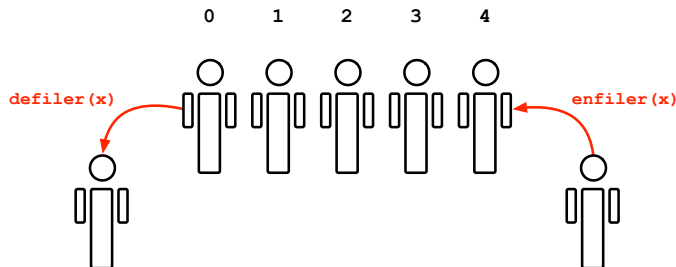
C'est l'opération **empiler** ou **push**.

- ▶ Pour supprimer l'élément en haut de la pile P , on décrémente `P.sommet` et on retourne `P[P.sommet+1]`.

C'est l'opération **dépiler** ou **pop**.

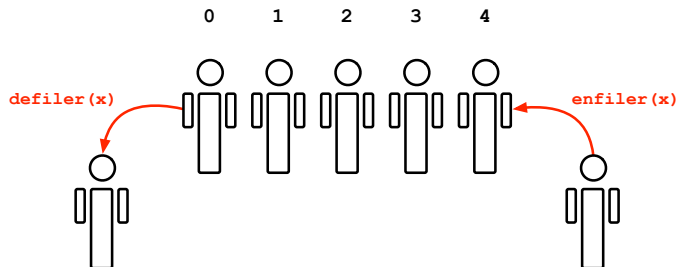
FILES

- ▶ Une **file** (**queue**) implémente le principe:
premier entré, premier sorti, First In First Out, (FIFO)
- ▶ Comme une file d'attente dans un magasin: la première personne arrivée sera la première servie.



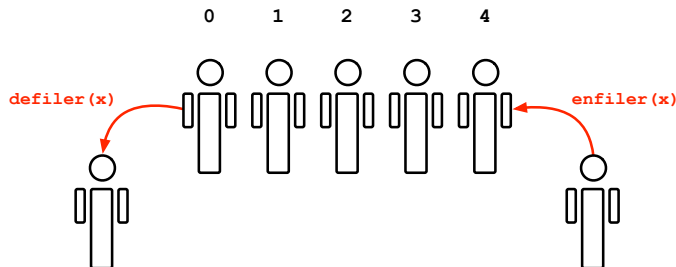
FILES

- ▶ Une **file** (**queue**) implémente le principe:
premier entré, premier sorti, First In First Out, (FIFO)
- ▶ Comme une file d'attente dans un magasin: la première personne arrivée sera la première servie.



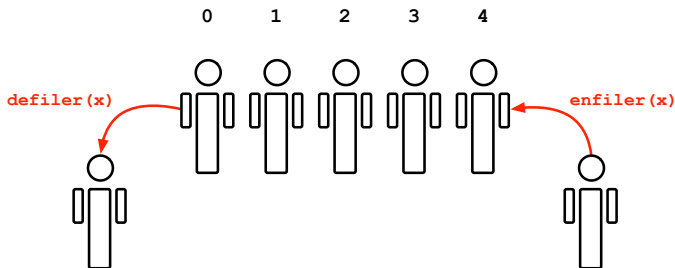
FILES

- ▶ Une **file** (**queue**) implémente le principe:
premier entré, premier sorti, First In First Out, (FIFO)
- ▶ Comme une file d'attente dans un magasin: la première personne arrivée sera la première servie.



FILES: APPLICATIONS

- Traiter des connexions, messages, processus, etc., dans leur ordre d'arrivée.



FILES

- ▶ Une **file** d'au plus $n-1$ éléments (et non n éléments) peut être implémentée par un tableau $F[0, \dots, n-1]$ ($0, \dots, n-1$ sont les indices des clés).
- ▶ La file possède deux attributs:
 - $F.tete$: indice du prochain indice du prochain élément à sortir (élément courant de la file)
 - $F.queue$: indice du prochain élément à entrer
- ▶ Les éléments de la file se trouvent aux emplacements $F.tete, F.tete + 1, \dots, F.queue - 1$ après quoi on boucle (l'emplacement 0 suit l'emplacement $n-1$).

FILES

- ▶ Une **file** d'au plus $n-1$ éléments (et non n éléments) peut être implémentée par un tableau $F[0, \dots, n-1]$ ($0, \dots, n-1$ sont les indices des clés).
- ▶ La file possède deux attributs:

$F.tete$: indice du prochain indice du prochain élément à sortir
(élément courant de la file)

$F.queue$: indice du prochain élément à entrer

- ▶ Les éléments de la file se trouvent aux emplacements
 $F.tete, F.tete + 1, \dots, F.queue - 1$
après quoi on boucle (l'emplacement 0 suit l'emplacement $n-1$).

FILES

- ▶ Une **file** d'au plus $n-1$ éléments (et non n éléments) peut être implémentée par un tableau $F[0, \dots, n-1]$ ($0, \dots, n-1$ sont les indices des clés).
- ▶ La file possède deux attributs:
 - $F.tete$: indice du prochain indice du prochain élément à sortir (élément courant de la file)
 - $F.queue$: indice du prochain élément à entrer
- ▶ Les éléments de la file se trouvent aux emplacements $F.tete, F.tete + 1, \dots, F.queue - 1$ après quoi on boucle (l'emplacement 0 suit l'emplacement $n-1$).

FILES

- ▶ Une **file** d'au plus $n-1$ éléments (et non n éléments) peut être implémentée par un tableau $F[0, \dots, n-1]$ ($0, \dots, n-1$ sont les indices des clés).
- ▶ La file possède deux attributs:
 - $F.tete$: indice du prochain indice du prochain élément à sortir (élément courant de la file)
 - $F.queue$: indice du prochain élément à entrer
- ▶ Les éléments de la file se trouvent aux emplacements $F.tete, F.tete + 1, \dots, F.queue - 1$ après quoi on boucle (l'emplacement 0 suit l'emplacement $n-1$).

FILES

- ▶ Une **file** d'au plus $n-1$ éléments (et non n éléments) peut être implémentée par un tableau $F[0, \dots, n-1]$ ($0, \dots, n-1$ sont les indices des clés).
- ▶ La file possède deux attributs:
 - $F.tete$: indice du prochain indice du prochain élément à sortir (élément courant de la file)
 - $F.queue$: indice du prochain élément à entrer
- ▶ Les éléments de la file se trouvent aux emplacements $F.tete, F.tete + 1, \dots, F.queue - 1$ après quoi on boucle (l'emplacement 0 suit l'emplacement $n-1$).

FILES

- ▶ Au départ, on a $F.tete = F.queue = 0$
- ▶ Si $F.tete = F.queue$, alors la file est vide.
C'est la requête **est_vide**.
- ▶ Si $F.tete = F.queue + 1$, alors la file est pleine.
- ▶ Pour insérer un élément x dans F , on assigne $F[F.queue] = x$ et on incrémente $F.queue$.
C'est l'opération **enfiler** ou **push**.
- ▶ Pour supprimer l'élément en premier de la file F , on décrémente $F.tete$ et on retourne $F[F.tete + 1]$.
C'est l'opération **défiler** ou **pop**.

FILES

- ▶ Au départ, on a $F.tete = F.queue = 0$
- ▶ Si $F.tete = F.queue$, alors la file est vide.
C'est la requête **est_vide**.
- ▶ Si $F.tete = F.queue + 1$, alors la file est pleine.
- ▶ Pour insérer un élément x dans F , on assigne $F[F.queue] = x$ et on incrémente $F.queue$.
C'est l'opération **enfiler** ou **push**.
- ▶ Pour supprimer l'élément en premier de la file F , on décrémente $F.tete$ et on retourne $F[F.tete + 1]$.
C'est l'opération **défiler** ou **pop**.

FILES

- ▶ Au départ, on a $F.tete = F.queue = 0$
- ▶ Si $F.tete = F.queue$, alors la file est vide.
C'est la requête **est_vide**.
- ▶ Si $F.tete = F.queue + 1$, alors la file est pleine.
- ▶ Pour insérer un élément x dans F , on assigne $F[F.queue] = x$ et on incrémente $F.queue$.
C'est l'opération **enfiler** ou **push**.
- ▶ Pour supprimer l'élément en premier de la file F , on décrémente $F.tete$ et on retourne $F[F.tete + 1]$.
C'est l'opération **défiler** ou **pop**.

FILES

- ▶ Au départ, on a $F.tete = F.queue = 0$
- ▶ Si $F.tete = F.queue$, alors la file est vide.
C'est la requête **est_vide**.
- ▶ Si $F.tete = F.queue + 1$, alors la file est pleine.
- ▶ Pour insérer un élément x dans F , on assigne $F[F.queue] = x$ et on incrémente $F.queue$.
C'est l'opération **enfiler** ou **push**.
- ▶ Pour supprimer l'élément en premier de la file F , on décrémente $F.tete$ et on retourne $F[F.tete + 1]$.
C'est l'opération **défiler** ou **pop**.

FILES

- ▶ Au départ, on a $F.tete = F.queue = 0$
- ▶ Si $F.tete = F.queue$, alors la file est vide.
C'est la requête **est_vide**.
- ▶ Si $F.tete = F.queue + 1$, alors la file est pleine.
- ▶ Pour insérer un élément x dans F , on assigne $F[F.queue] = x$ et on incrémente $F.queue$.

C'est l'opération **enfiler** ou **push**.

- ▶ Pour supprimer l'élément en premier de la file F , on décrémente $F.tete$ et on retourne $F[F.tete + 1]$.

C'est l'opération **défiler** ou **pop**.

FILES

- ▶ Au départ, on a $F.tete = F.queue = 0$
- ▶ Si $F.tete = F.queue$, alors la file est vide.
C'est la requête **est_vide**.
- ▶ Si $F.tete = F.queue + 1$, alors la file est pleine.
- ▶ Pour insérer un élément x dans F , on assigne $F[F.queue] = x$ et on incrémente $F.queue$.
C'est l'opération **enfiler** ou **push**.
- ▶ Pour supprimer l'élément en premier de la file F , on décrémente $F.tete$ et on retourne $F[F.tete + 1]$.
C'est l'opération **défiler** ou **pop**.

FILES

- ▶ Au départ, on a $F.tete = F.queue = 0$
- ▶ Si $F.tete = F.queue$, alors la file est vide.
C'est la requête **est_vide**.
- ▶ Si $F.tete = F.queue + 1$, alors la file est pleine.
- ▶ Pour insérer un élément x dans F , on assigne $F[F.queue] = x$ et on incrémente $F.queue$.
C'est l'opération **enfiler** ou **push**.
- ▶ Pour supprimer l'élément en premier de la file F , on décrémente $F.tete$ et on retourne $F[F.tete + 1]$.
C'est l'opération **défiler** ou **pop**.

FILES

- ▶ Au départ, on a $F.tete = F.queue = 0$
- ▶ Si $F.tete = F.queue$, alors la file est vide.
C'est la requête **est_vide**.
- ▶ Si $F.tete = F.queue + 1$, alors la file est pleine.
- ▶ Pour insérer un élément x dans F , on assigne $F[F.queue] = x$ et on incrémente $F.queue$.
C'est l'opération **enfiler** ou **push**.
- ▶ Pour supprimer l'élément en premier de la file F , on décrémente $F.tete$ et on retourne $F[F.tete + 1]$.
C'est l'opération **défiler** ou **pop**.