

Définition : structure de données

- En informatique, une **structure de données** est une façon d'organiser, de gérer et de stocker des données permettant d'accéder et de modifier ces données de façon efficace.

Définition : structure de données

- En informatique, une **structure de données** est une façon d'organiser, de gérer et de stocker des données permettant d'accéder et de modifier ces données de façon efficace.

Les tableaux fixes du C sont un exemple de structure de données.

Définition : structure de données

- En informatique, une **structure de données** est une façon d'organiser, de gérer et de stocker des données permettant d'accéder et de modifier ces données de façon efficace.

Les tableaux fixes du C sont un exemple de structure de données.

- L'**interface** de la structure de données est l'ensemble des opérations accessibles à un utilisateur de la structure de données.

Définition : structure de données

- En informatique, une **structure de données** est une façon d'organiser, de gérer et de stocker des données permettant d'accéder et de modifier ces données de façon efficace.

Les tableaux fixes du C sont un exemple de structure de données.

- L'**interface** de la structure de données est l'ensemble des opérations accessibles à un utilisateur de la structure de données.

Par exemple, la notation `[]` permet d'accéder à un élément du tableau, pour le lire ou le modifier. Par contre, en C, la taille du tableau ne fait pas partie de l'interface.

Définition : structure de données

- En informatique, une **structure de données** est une façon d'organiser, de gérer et de stocker des données permettant d'accéder et de modifier ces données de façon efficace.

Les tableaux fixes du C sont un exemple de structure de données.

- L'**interface** de la structure de données est l'ensemble des opérations accessibles à un utilisateur de la structure de données.

Par exemple, la notation `[]` permet d'accéder à un élément du tableau, pour le lire ou le modifier. Par contre, en C, la taille du tableau ne fait pas partie de l'interface.

- L'**implémentation** de la structure de données est la façon dont elle est représentée et codée en mémoire et n'est pas forcément accessible à l'utilisateur.

Définition : structure de données

- En informatique, une **structure de données** est une façon d'organiser, de gérer et de stocker des données permettant d'accéder et de modifier ces données de façon efficace.

Les tableaux fixes du C sont un exemple de structure de données.

- L'**interface** de la structure de données est l'ensemble des opérations accessibles à un utilisateur de la structure de données.

Par exemple, la notation `[]` permet d'accéder à un élément du tableau, pour le lire ou le modifier. Par contre, en C, la taille du tableau ne fait pas partie de l'interface.

- L'**implémentation** de la structure de données est la façon dont elle est représentée et codée en mémoire et n'est pas forcément accessible à l'utilisateur.

On peut utiliser les listes de OCaml via leur interface sans savoir comment elles sont représentées en mémoire par le langage.

Caractérisation par l'interface

- La différence entre **interface** et **implémentation** est fondamentale et doit être bien comprise. En effet une même structure de données peut avoir plusieurs implémentations. L'idée est que l'utilisation de la structure de données doit se faire indépendamment de son implémentation ce qui permet la séparation des programmes en composants indépendants (modularité).
On utilise la même interface (les opérations arithmétiques) pour manipuler les entiers du C et de Python mais ils ne sont pas implémentés de la même manière.

Caractérisation par l'interface

- La différence entre **interface** et **implémentation** est fondamentale et doit être bien comprise. En effet une même structure de données peut avoir plusieurs implémentations. L'idée est que l'utilisation de la structure de données doit se faire indépendamment de son implémentation ce qui permet la séparation des programmes en composants indépendants (modularité).
On utilise la même interface (les opérations arithmétiques) pour manipuler les entiers du C et de Python mais ils ne sont pas implémentés de la même manière.
- Lorsqu'on définit un *cahier des charges* pour une structure de données (ensemble des données et opérations possibles), on définit ce qu'on appelle un **type abstrait de données**. Ainsi, une structure de données peut être vue comme une implémentation d'un type abstrait de données.

Caractérisation par l'interface

- La différence entre **interface** et **implémentation** est fondamentale et doit être bien comprise. En effet une même structure de données peut avoir plusieurs implémentations. L'idée est que l'utilisation de la structure de données doit se faire indépendamment de son implémentation ce qui permet la séparation des programmes en composants indépendants (modularité).

On utilise la même interface (les opérations arithmétiques) pour manipuler les entiers du C et de Python mais ils ne sont pas implémentés de la même manière.

- Lorsqu'on définit un *cahier des charges* pour une structure de données (ensemble des données et opérations possibles), on définit ce qu'on appelle un **type abstrait de données**. Ainsi, une structure de données peut être vue comme une implémentation d'un type abstrait de données.
- La définition complète d'un type abstrait de données inclut généralement la complexité des opérations de l'interface.

L'ajout d'un élément en tête d'une liste de OCaml est une opération en $O(1)$.

Opérations de l'interface

- La création d'une structure de données se fait à l'aide d'une opération de l'interface appelé **constructeur**

Opérations de l'interface

- La création d'une structure de données se fait à l'aide d'une opération de l'interface appelé **constructeur**

Par exemple en C, **double** tab[10];

Opérations de l'interface

- La création d'une structure de données se fait à l'aide d'une opération de l'interface appelé **constructeur**
Par exemple en C, `double tab[10];`
- La récupération d'une valeur dans la structure se fait à l'aide d'**accesseur** (en anglais *getter*).

Opérations de l'interface

- La création d'une structure de données se fait à l'aide d'une opération de l'interface appelé **constructeur**
Par exemple en C, `double tab[10];`
- La récupération d'une valeur dans la structure se fait à l'aide d'**accesseur** (en anglais *getter*).
Par exemple en C, `double e = tab[3];`

Opérations de l'interface

- La création d'une structure de données se fait à l'aide d'une opération de l'interface appelé **constructeur**
Par exemple en C, `double tab[10];`
- La récupération d'une valeur dans la structure se fait à l'aide d'**accesseur** (en anglais *getter*).
Par exemple en C, `double e = tab[3];`
- La modification d'une valeur dans la structure se fait à l'aide de **transformateur** (en anglais *setter*).

Opérations de l'interface

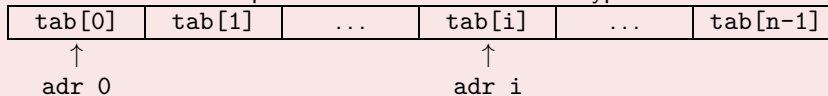
- La création d'une structure de données se fait à l'aide d'une opération de l'interface appelé **constructeur**
Par exemple en C, `double tab[10];`
- La récupération d'une valeur dans la structure se fait à l'aide d'**accesseur** (en anglais *getter*).
Par exemple en C, `double e = tab[3];`
- La modification d'une valeur dans la structure se fait à l'aide de **transformateur** (en anglais *setter*).
Par exemple en C, `tab[3] = 7.5;`

Opérations de l'interface

- La création d'une structure de données se fait à l'aide d'une opération de l'interface appelé **constructeur**
Par exemple en C, `double tab[10];`
- La récupération d'une valeur dans la structure se fait à l'aide d'**accesseur** (en anglais *getter*).
Par exemple en C, `double e = tab[3];`
- La modification d'une valeur dans la structure se fait à l'aide de **transformateur** (en anglais *setter*).
Par exemple en C, `tab[3] = 7.5;`
⚠ On distingue les structures de données mutables (comme les tableaux du C) des structures de données immuables (comme les listes de OCaml). En cas de non mutabilité, *pour modifier une structure de données on doit en construire une nouvelle.*

Définition

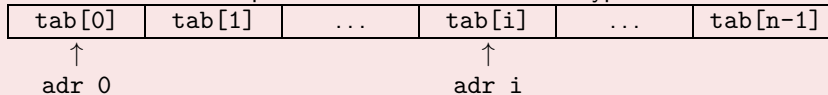
un tableau est une séquence de n valeurs de même type consécutives en mémoire.



Complexité des opérations

Définition

un tableau est une séquence de n valeurs de même type **consécutives en mémoire**.

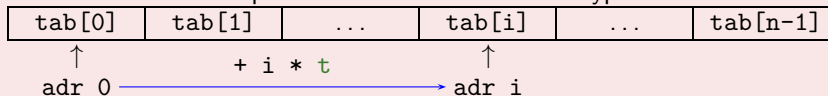


Complexité des opérations

- L'accès à un élément se fait **temps constant**, en effet il suffit de connaître la taille t d'une case et de disposer de l'adresse du premier élément du tableau adr_0 . L'adresse de l'élément d'indice i s'obtient alors en ajoutant à l'adresse du premier élément i fois la taille d'une case.

Définition

un tableau est une séquence de n valeurs de même type **consécutives en mémoire**.

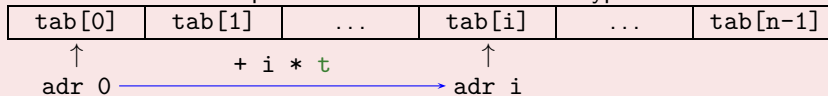


Complexité des opérations

- L'accès à un élément se fait **temps constant**, en effet il suffit de connaître la taille t d'une case et de disposer de l'adresse du premier élément du tableau $adr0$. L'adresse de l'élément d'indice i s'obtient alors en ajoutant à l'adresse du premier élément i fois la taille d'une case.

Définition

un tableau est une séquence de n valeurs de même type **consécutives en mémoire**.



Complexité des opérations

- L'accès à un élément se fait **temps constant**, en effet il suffit de connaître la taille t d'une case et de disposer de l'adresse du premier élément du tableau $\text{adr}0$. L'adresse de l'élément d'indice i s'obtient alors en ajoutant à l'adresse du premier élément i fois la taille d'une case.
- La suppression ou l'insertion d'un élément demande par contre la recopie des éléments et ce sont donc des opérations en $\mathcal{O}(n)$.

Remarques

- Les tableaux sont des structures de données mutables par nature, ceux de OCaml seront donc vus dans les aspects impératifs de ce langage. On notera cependant la syntaxe pour l'accès à un élément : `tab.(i)` pour l'élément d'indice `i` du tableau `tab`.

Remarques

- Les tableaux sont des structures de données mutables par nature, ceux de OCaml seront donc vus dans les aspects impératifs de ce langage. On notera cependant la syntaxe pour l'accès à un élément : `tab.(i)` pour l'élément d'indice `i` du tableau `tab`.
- En C, l'accès au i ème par addition à l'adresse du premier de $i \times t$ où t est la taille d'une case est masqué par le « sucre syntaxique » `tab[i]`. On notera cependant, que les deux syntaxes suivantes sont tout à fait équivalentes ! :

```
1  int main(){
2      int tab[] = {2, 3, 5, 7, 11, 13, 17, 19};
3      // Avec sucre syntaxique
4      int elt5 = tab[5];
5      // En utilisant la représentation en mémoire
6      int elt5 = *(tab + 5) ;}
```

Tableaux dynamiques

- Les tableaux ont une taille fixée au moment de leur création de façon à réserver l'espace mémoire nécessaire.

Tableaux dynamiques

- Les tableaux ont une taille fixée au moment de leur création de façon à réserver l'espace mémoire nécessaire.
- On peut créer des tableaux dynamiques (ou redimensionnables) à la façon des listes de Python, une implémentation en C sera vue en TP.

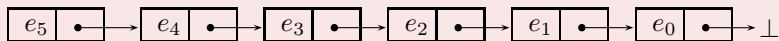
Tableaux dynamiques

- Les tableaux ont une taille fixée au moment de leur création de façon à réserver l'espace mémoire nécessaire.
- On peut créer des tableaux dynamiques (ou redimensionnables) à la façon des listes de Python, une implémentation en C sera vue en TP.
- On retiendra dès maintenant que dans le cas d'un redimensionnement, la stratégie efficace est alors de **doubler** la taille courante du tableau car cela conduit à une complexité **amortie** en $\mathcal{O}(1)$ lors de l'ajout.

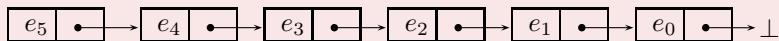
C8 Structures de données séquentielles

3. Liste chaînées

Définition

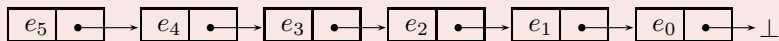


Définition



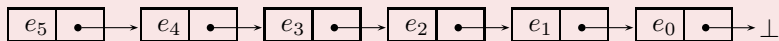
- Contrairement aux tableaux, les différentes valeurs ne sont pas stockés de façon contiguës en mémoire.

Définition



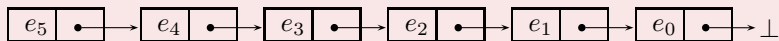
- Contrairement aux tableaux, les différentes valeurs ne sont pas stockés de façon contiguës en mémoire.
- Avec chaque élément, on stocke aussi dans un « maillon » l'emplacement de son successeur. Le successeur du dernier élément se note \perp (**NULL** en C).

Définition



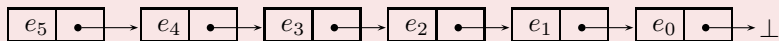
- Contrairement aux tableaux, les différentes valeurs ne sont pas stockés de façon contiguës en mémoire.
- Avec chaque élément, on stocke aussi dans un « maillon » l'emplacement de son successeur. Le successeur du dernier élément se note \perp (**NULL** en C).

Définition



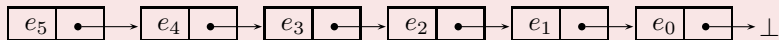
- Contrairement aux tableaux, les différentes valeurs ne sont pas stockés de façon contiguës en mémoire.
- Avec chaque élément, on stocke aussi dans un « maillon » l'emplacement de son successeur. Le successeur du dernier élément se note \perp (**NULL** en C).

Définition



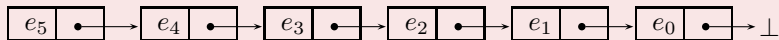
- Contrairement aux tableaux, les différentes valeurs ne sont pas stockés de façon contiguës en mémoire.
- Avec chaque élément, on stocke aussi dans un « maillon » l'emplacement de son successeur. Le successeur du dernier élément se note \perp (**NULL** en C).
- Les listes chaînées peuvent être définies de façon **récursive** :

Définition



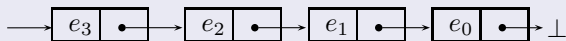
- Contrairement aux tableaux, les différentes valeurs ne sont pas stockés de façon contiguës en mémoire.
- Avec chaque élément, on stocke aussi dans un « maillon » l'emplacement de son successeur. Le successeur du dernier élément se note \perp (**NULL** en C).
- Les listes chaînées peuvent être définies de façon **récursive** :
 - Une liste est soit vide (référence vers \perp)

Définition

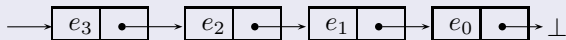


- Contrairement aux tableaux, les différentes valeurs ne sont pas stockés de façon contiguës en mémoire.
- Avec chaque élément, on stocke aussi dans un « maillon » l'emplacement de son successeur. Le successeur du dernier élément se note \perp (NULL en C).
- Les listes chaînées peuvent être définies de façon **récursive** :
 - Une liste est soit vide (référence vers \perp)
 - Soit c'est la donnée d'un maillon constitué d'une valeur et d'une référence vers une liste.

Complexités des opérations



Complexités des opérations



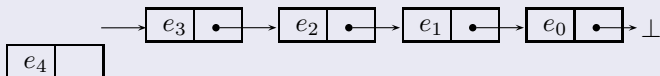
- Pour accéder à un élément, on doit parcourir tout ceux qui le précèdent. L'accès à un élément est donc une opération en $\mathcal{O}(n)$.

Complexités des opérations



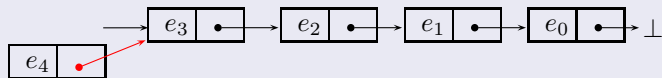
- Pour accéder à un élément, on doit parcourir tout ceux qui le précèdent. L'accès à un élément est donc une opération en $\mathcal{O}(n)$.
- L'ajout ou la suppression en tête de liste est en $\mathcal{O}(1)$ car aucune copie n'est nécessaire :

Complexités des opérations



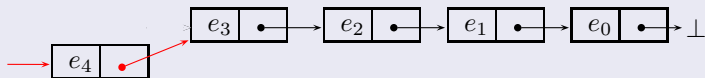
- Pour accéder à un élément, on doit parcourir tout ceux qui le précèdent. L'accès à un élément est donc une opération en $\mathcal{O}(n)$.
- L'ajout ou la suppression en tête de liste est en $\mathcal{O}(1)$ car aucune copie n'est nécessaire :
 - 1 On crée un maillon avec la nouvelle valeur.

Complexités des opérations



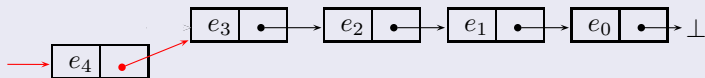
- Pour accéder à un élément, on doit parcourir tout ceux qui le précèdent. L'accès à un élément est donc une opération en $\mathcal{O}(n)$.
- L'ajout ou la suppression en tête de liste est en $\mathcal{O}(1)$ car aucune copie n'est nécessaire :
 - 1 On crée un maillon avec la nouvelle valeur.
 - 2 Le suivant de ce maillon est l'ancienne tête.

Complexités des opérations



- Pour accéder à un élément, on doit parcourir tout ceux qui le précèdent. L'accès à un élément est donc une opération en $\mathcal{O}(n)$.
- L'ajout ou la suppression en tête de liste est en $\mathcal{O}(1)$ car aucune recopie n'est nécessaire :
 - 1 On crée un maillon avec la nouvelle valeur.
 - 2 Le suivant de ce maillon est l'ancienne tête.
 - 3 La nouvelle tête pointe sur le nouveau maillon.

Complexités des opérations



- Pour accéder à un élément, on doit parcourir tout ceux qui le précèdent. L'accès à un élément est donc une opération en $\mathcal{O}(n)$.
- L'ajout ou la suppression en tête de liste est en $\mathcal{O}(1)$ car aucune recopie n'est nécessaire :
 - 1 On crée un maillon avec la nouvelle valeur.
 - 2 Le suivant de ce maillon est l'ancienne tête.
 - 3 La nouvelle tête pointe sur le nouveau maillon.
- La taille de la structure de données n'est pas fixée à la construction contrairement aux tableaux.

C8 Structures de données séquentielles

3. Liste chaînées

Implémentation en C

On peut définir un maillon comme un `struct` avec les champs valeur et pointeur vers un maillon :

Implémentation en OCaml

C8 Structures de données séquentielles

3. Liste chaînées

Implémentation en C

On peut définir un maillon comme un `struct` avec les champs valeur et pointeur vers un maillon :

```
1 struct maillon_s
2 {
3     int valeur;
4     struct maillon_s * suivant;
5 };
6 typedef struct maillon_s maillon;
7 typedef maillon* liste;
```

Implémentation en OCaml

C8 Structures de données séquentielles

3. Liste chaînées

Implémentation en C

On peut définir un maillon comme un `struct` avec les champs valeur et pointeur vers un maillon :

```
1 struct maillon_s
2 {
3     int valeur;
4     struct maillon_s * suivant;
5 };
6 typedef struct maillon_s maillon;
7 typedef maillon* liste;
```

Implémentation en OCaml

- Le type `'a list` est prédéfini dans le langage

C8 Structures de données séquentielles

3. Liste chaînées

Implémentation en C

On peut définir un maillon comme un `struct` avec les champs valeur et pointeur vers un maillon :

```
1 struct maillon_s
2 {
3     int valeur;
4     struct maillon_s * suivant;
5 };
6 typedef struct maillon_s maillon;
7 typedef maillon* liste;
```

Implémentation en OCaml

- Le type `'a list` est prédéfini dans le langage
- Attention, les listes de OCaml ne sont pas mutables

Implémentation en C

On peut définir un maillon comme un `struct` avec les champs valeur et pointeur vers un maillon :

```
1 struct maillon_s
2 {
3     int valeur;
4     struct maillon_s * suivant;
5 };
6 typedef struct maillon_s maillon;
7 typedef maillon* liste;
```

Implémentation en OCaml

- Le type `'a list` est prédéfini dans le langage
- Attention, les listes de OCaml ne sont pas mutables
- Tous les éléments doivent être du même type

C8 Structures de données séquentielles

3. Liste chaînées

Interface des list de OCaml

Fonction	Rôle
List.mem a -> 'a list -> bool	Teste l'appartenance à la liste
List.length : 'a list -> int	Renvoie la longueur de la liste
List.map : ('a -> 'b) -> 'a list -> 'b list	Application d'une fonction
List.for_all : ('a -> bool) -> 'a list -> bool	Vérification d'un prédicat
List.rev : 'a list -> 'a	Retourne la liste
List.filter : ('a -> bool) -> 'a list -> 'a list	Filtrage suivant un prédicat
List.iter : ('a -> unit) -> 'a list -> unit	Applique une fonction renvoyant ()

Toutes ces fonctions sont de complexité $\mathcal{O}(n)$ où n est la longueur de la liste.

Interface des list de OCaml

Fonction	Rôle
List.mem a -> 'a list -> bool	Teste l'appartenance à la liste
List.length : 'a list -> int	Renvoie la longueur de la liste
List.map : ('a -> 'b) -> 'a list -> 'b list	Application d'une fonction
List.for_all : ('a -> bool) -> 'a list -> bool	Vérification d'un prédicat
List.rev : 'a list -> 'a	Retourne la liste
List.filter : ('a -> bool) -> 'a list -> 'a list	Filtrage suivant un prédicat
List.iter : ('a -> unit) -> 'a list -> unit	Applique une fonction renvoyant ()

Toutes ces fonctions sont de complexité $\mathcal{O}(n)$ où n est la longueur de la liste.
La concaténation de deux listes @ : 'a list -> 'a list -> 'a list est de complexité $\mathcal{O}(n_1)$ où n_1 est la longueur de la *première* liste.

C8 Structures de données séquentielles

3. Liste chaînées

Interface des list de OCaml

Fonction	Rôle
List.mem a -> 'a list -> bool	Teste l'appartenance à la liste
List.length : 'a list -> int	Renvoie la longueur de la liste
List.map : ('a -> 'b) -> 'a list -> 'b list	Application d'une fonction
List.for_all : ('a -> bool) -> 'a list -> bool	Vérification d'un prédicat
List.rev : 'a list -> 'a	Retourne la liste
List.filter : ('a -> bool) -> 'a list -> 'a list	Filtrage suivant un prédicat
List.iter : ('a -> unit) -> 'a list -> unit	Applique une fonction renvoyant ()

Toutes ces fonctions sont de complexité $\mathcal{O}(n)$ où n est la longueur de la liste.

La concaténation de deux listes @ : 'a list -> 'a list -> 'a list est de complexité $\mathcal{O}(n_1)$ où n_1 est la longueur de la *première* liste.

On rappelle que List.fold_left et List.fold_right permettent de parcourir une liste (depuis la gauche ou depuis la droite) en accumulant les résultats successifs trouvés.

Exercice

On rappelle l'implémentation d'une liste chaînée en C ;

```
1 struct maillon_s
2 {
3     int valeur;
4     struct maillon_s * suivant;
5 };
6 typedef struct maillon_s maillon;
7 typedef maillon* liste;
```

Exercice

On rappelle l'implémentation d'une liste chaînée en C ;

```
1 struct maillon_s
2 {
3     int valeur;
4     struct maillon_s * suivant;
5 };
6 typedef struct maillon_s maillon;
7 typedef maillon* liste;
```

- 1 Ecrire une fonction affiche de signature `void affiche(liste l)` qui affiche les éléments de la liste chaînée `l`.

Exercice

On rappelle l'implémentation d'une liste chaînée en C ;

```
1 struct maillon_s
2 {
3     int valeur;
4     struct maillon_s * suivant;
5 };
6 typedef struct maillon_s maillon;
7 typedef maillon* liste;
```

- 1 Ecrire une fonction affiche de signature `void affiche(liste l)` qui affiche les éléments de la liste chaînée `l`.
- 2 Donner la signature d'une fonction ajoute qui ajoute une valeur en tête d'une liste.

Exercice

On rappelle l'implémentation d'une liste chaînée en C ;

```
1 struct maillon_s
2 {
3     int valeur;
4     struct maillon_s * suivant;
5 };
6 typedef struct maillon_s maillon;
7 typedef maillon* liste;
```

- 1 Ecrire une fonction affiche de signature `void affiche(liste l)` qui affiche les éléments de la liste chaînée `l`.
- 2 Donner la signature d'une fonction ajoute qui ajoute une valeur en tête d'une liste.
- 3 Donner l'implémentation de cette fonction.
 - 🎯 On pourra traduire en C les étapes de l'ajout : création du nouveau maillon en le faisant pointer vers l'ancienne tête, modification de la liste dont la tête est le nouveau maillon.

Piles

- Au niveau sémantique, une **pile** est semblable à une pile d'objet dans la vie de tous les jours.

Piles

- Au niveau sémantique, une **pile** est semblable à une pile d'objet dans la vie de tous les jours.

elt4
elt3
elt2
elt1

Piles

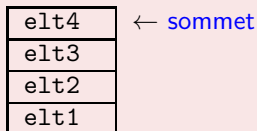
- Au niveau sémantique, une **pile** est semblable à une pile d'objet dans la vie de tous les jours.

elt4
elt3
elt2
elt1

- L'élément situé en haut de la pile s'appelle le **sommet**.

Piles

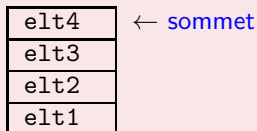
- Au niveau sémantique, une **pile** est semblable à une pile d'objet dans la vie de tous les jours.



- L'élément situé en haut de la pile s'appelle le **sommet**.

Piles

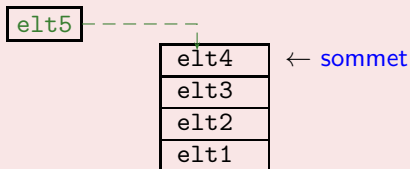
- Au niveau sémantique, une **pile** est semblable à une pile d'objet dans la vie de tous les jours.



- L'élément situé en haut de la pile s'appelle le **sommet**.
- **Empiler** signifie ajouter un élément au sommet de la pile

Piles

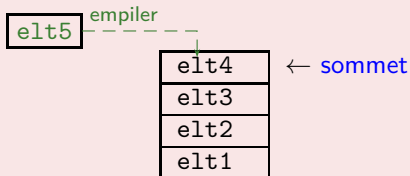
- Au niveau sémantique, une **pile** est semblable à une pile d'objet dans la vie de tous les jours.



- L'élément situé en haut de la pile s'appelle le **sommet**.
- **Empiler** signifie ajouter un élément au sommet de la pile

Piles

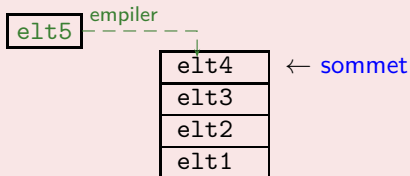
- Au niveau sémantique, une **pile** est semblable à une pile d'objet dans la vie de tous les jours.



- L'élément situé en haut de la pile s'appelle le **sommet**.
- **Empiler** signifie ajouter un élément au sommet de la pile

Piles

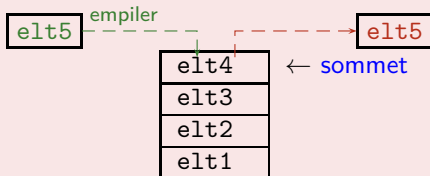
- Au niveau sémantique, une **pile** est semblable à une pile d'objet dans la vie de tous les jours.



- L'élément situé en haut de la pile s'appelle le **sommet**.
- **Empiler** signifie ajouter un élément au sommet de la pile
- **Dépiler** signifie retirer l'élément situé au sommet de la pile

Piles

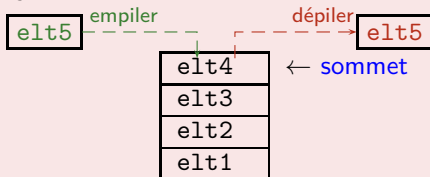
- Au niveau sémantique, une **pile** est semblable à une pile d'objet dans la vie de tous les jours.



- L'élément situé en haut de la pile s'appelle le **sommet**.
- **Empiler** signifie ajouter un élément au sommet de la pile
- **Dépiler** signifie retirer l'élément situé au sommet de la pile

Piles

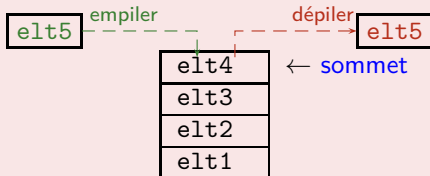
- Au niveau sémantique, une **pile** est semblable à une pile d'objet dans la vie de tous les jours.



- L'élément situé en haut de la pile s'appelle le **sommet**.
- **Empiler** signifie ajouter un élément au sommet de la pile
- **Dépiler** signifie retirer l'élément situé au sommet de la pile

Piles

- Au niveau sémantique, une **pile** est semblable à une pile d'objet dans la vie de tous les jours.



- L'élément situé en haut de la pile s'appelle le **sommet**.
- **Empiler** signifie ajouter un élément au sommet de la pile
- **Dépiler** signifie retirer l'élément situé au sommet de la pile
- Ainsi le premier élément entré dans la pile sera aussi le dernier à en sortir, on dit qu'une pile est une structure **LIFO** *Last In First Out*

Piles comme structures de données

L'interface d'une structure de données Pile se limite donc aux opérations suivantes :

Piles comme structures de données

L'interface d'une structure de données Pile se limite donc aux opérations suivantes :

- `est_vide` qui renvoie un booléen indiquant si la pile est vide ou non.

Piles comme structures de données

L'interface d'une structure de données Pile se limite donc aux opérations suivantes :

- `est_vide` qui renvoie un booléen indiquant si la pile est vide ou non.
- `empiler` (en anglais *push*) qui ajoute un élément au sommet de la pile.

Piles comme structures de données

L'interface d'une structure de données Pile se limite donc aux opérations suivantes :

- `est_vide` qui renvoie un booléen indiquant si la pile est vide ou non.
- `empiler` (en anglais *push*) qui ajoute un élément au sommet de la pile.
- `depiler` (en anglais *pop*) qui retire l'élément situé au sommet (cela n'est possible que si la pile n'est pas vide).

Piles comme structures de données

L'interface d'une structure de données Pile se limite donc aux opérations suivantes :

- `est_vide` qui renvoie un booléen indiquant si la pile est vide ou non.
- `empiler` (en anglais *push*) qui ajoute un élément au sommet de la pile.
- `depiler` (en anglais *pop*) qui retire l'élément situé au sommet (cela n'est possible que si la pile n'est pas vide).

Utilisation

En dépit de sa simplicité, cette structure de données a de nombreuses applications en informatique : pile d'appel récursif, pile d'évaluation d'une expression, ...

Manipulation de piles

Manipulation de piles

- On considère la pile : $P = |"A", "L", "I", "X">$ (le sommet est "X"). Quelle suite d'opération permet d'obtenir $P = |"A", "L", "E", "X">$?

Manipulation de piles

- On considère la pile : $P = | "A", "L", "I", "X" >$ (le sommet est "X"). Quelle suite d'opération permet d'obtenir $P = | "A", "L", "E", "X" >$?
- Un programmeur décide d'utiliser une pile afin de stocker une réponse entrée au clavier. Chaque caractère tapé doit être empiler. Traduire en terme d'opérations sur cette pile les actions suivantes :


Manipulation de piles

- On considère la pile : $P = | "A", "L", "I", "X" >$ (le sommet est "X"). Quelle suite d'opération permet d'obtenir $P = | "A", "L", "E", "X" >$?
- Un programmeur décide d'utiliser une pile afin de stocker une réponse entrée au clavier. Chaque caractère tapé doit être empiler. Traduire en terme d'opérations sur cette pile les actions suivantes :
 - 1 Appuie sur la touche "o"


Manipulation de piles

- On considère la pile : $P = | "A", "L", "I", "X" >$ (le sommet est "X"). Quelle suite d'opération permet d'obtenir $P = | "A", "L", "E", "X" >$?
- Un programmeur décide d'utiliser une pile afin de stocker une réponse entrée au clavier. Chaque caractère tapé doit être empiler. Traduire en terme d'opérations sur cette pile les actions suivantes :
 - 1 Appuie sur la touche "o"
 - 2 Appuie sur la touche "i"

Manipulation de piles

- On considère la pile : $P = | "A", "L", "I", "X" >$ (le sommet est "X"). Quelle suite d'opération permet d'obtenir $P = | "A", "L", "E", "X" >$?
- Un programmeur décide d'utiliser une pile afin de stocker une réponse entrée au clavier. Chaque caractère tapé doit être empiler. Traduire en terme d'opérations sur cette pile les actions suivantes :
 - 1 Appuie sur la touche "o"
 - 2 Appuie sur la touche "i"
 - 3 Appuie sur la touche  (*backspace*)

Manipulation de piles

- On considère la pile : $P = | "A", "L", "I", "X" >$ (le sommet est "X"). Quelle suite d'opération permet d'obtenir $P = | "A", "L", "E", "X" >$?
- Un programmeur décide d'utiliser une pile afin de stocker une réponse entrée au clavier. Chaque caractère tapé doit être empiler. Traduire en terme d'opérations sur cette pile les actions suivantes :
 - 1 Appuie sur la touche "o"
 - 2 Appuie sur la touche "i"
 - 3 Appuie sur la touche  (*backspace*)
 - 4 Appuie sur la touche "k"

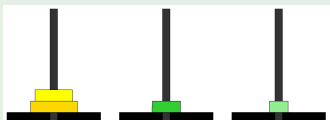
Manipulation de piles

Au jeu des tours des Hanoï, on gère les trois tours T1, T2 et T3 à l'aide de trois piles.

Manipulation de piles

Au jeu des tours des Hanoï, on gère les trois tours T1, T2 et T3 à l'aide de trois piles.

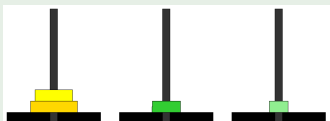
- 1 Quel est le contenu de chacune des piles dans la situation ci-dessous? (un disque est représenté dans la pile par sa taille)



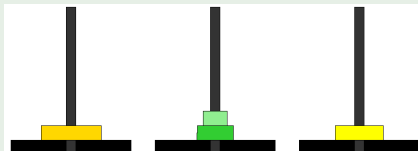
Manipulation de piles

Au jeu des tours des Hanoï, on gère les trois tours T1, T2 et T3 à l'aide de trois piles.

- 1 Quel est le contenu de chacune des piles dans la situation ci-dessous ? (un disque est représenté dans la pile par sa taille)



- 2 Ecrire les opérations permettant de passer la situation précédente à celle ci-dessous :



Implémentation des piles

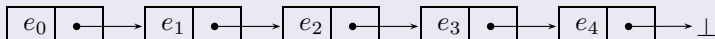
Plusieurs implémentations sont possibles :

- A l'aide d'une liste chaînée, la tête de la liste représente alors le sommet de la pile.

Implémentation des piles

Plusieurs implémentations sont possibles :

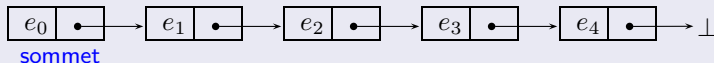
- A l'aide d'une liste chaînée, la tête de la liste représente alors le sommet de la pile.



Implémentation des piles

Plusieurs implémentations sont possibles :

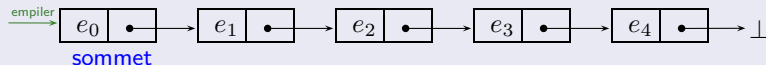
- A l'aide d'une liste chaînée, la tête de la liste représente alors le sommet de la pile.



Implémentation des piles

Plusieurs implémentations sont possibles :

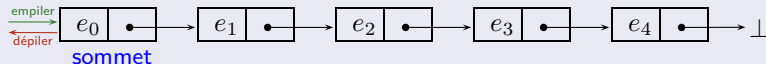
- A l'aide d'une liste chaînée, la tête de la liste représente alors le sommet de la pile.



Implémentation des piles

Plusieurs implémentations sont possibles :

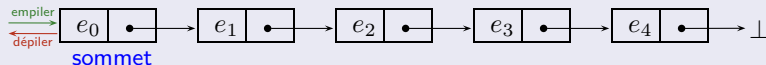
- A l'aide d'une liste chaînée, la tête de la liste représente alors le sommet de la pile.



Implémentation des piles

Plusieurs implémentations sont possibles :

- A l'aide d'une liste chaînée, la tête de la liste représente alors le sommet de la pile.

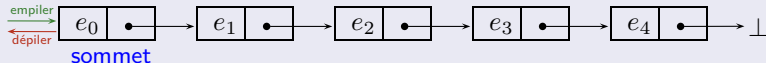


- Si la capacité de la pile n est bornée et connue en amont, on peut aussi utiliser un tableau de taille n et mémoriser la taille courante t de la pile.

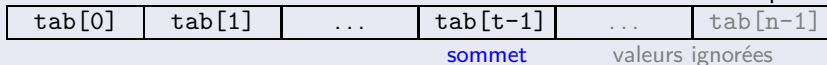
Implémentation des piles

Plusieurs implémentations sont possibles :

- A l'aide d'une liste chaînée, la tête de la liste représente alors le sommet de la pile.



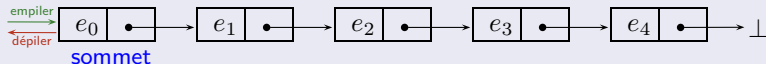
- Si la capacité de la pile n est bornée et connue en amont, on peut aussi utiliser un tableau de taille n et mémoriser la taille courante t de la pile.



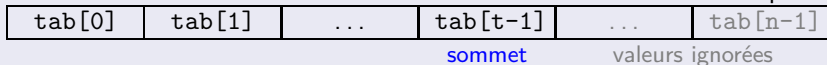
Implémentation des piles

Plusieurs implémentations sont possibles :

- A l'aide d'une liste chaînée, la tête de la liste représente alors le sommet de la pile.



- Si la capacité de la pile n est bornée et connue en amont, on peut aussi utiliser un tableau de taille n et mémoriser la taille courante t de la pile.

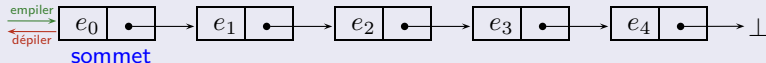


- pour tester si la pile est vide on teste si t est égal à 0,

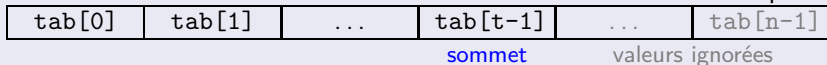
Implémentation des piles

Plusieurs implémentations sont possibles :

- A l'aide d'une liste chaînée, la tête de la liste représente alors le sommet de la pile.



- Si la capacité de la pile n est bornée et connue en amont, on peut aussi utiliser un tableau de taille n et mémoriser la taille courante t de la pile.

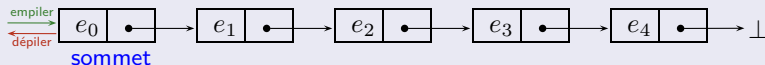


- pour tester si la pile est vide on teste si t est égal à 0,
- pour empiler une valeur v , on affecte $\text{tab}[t]=v$ et on incrémente t ,

Implémentation des piles

Plusieurs implémentations sont possibles :

- A l'aide d'une liste chaînée, la tête de la liste représente alors le sommet de la pile.



- Si la capacité de la pile n est bornée et connue en amont, on peut aussi utiliser un tableau de taille n et mémoriser la taille courante t de la pile.



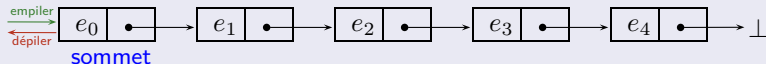
sommet $\xrightarrow{\text{empiler}}$ valeurs ignorées

- pour tester si la pile est vide on teste si t est égal à 0,
- pour empiler une valeur v , on affecte $\text{tab}[t]=v$ et on incrémente t ,

Implémentation des piles

Plusieurs implémentations sont possibles :

- A l'aide d'une liste chaînée, la tête de la liste représente alors le sommet de la pile.



- Si la capacité de la pile n est bornée et connue en amont, on peut aussi utiliser un tableau de taille n et mémoriser la taille courante t de la pile.



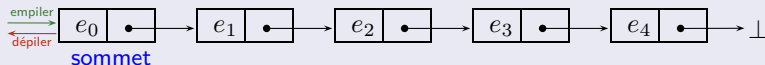
sommet $\xrightarrow{\text{empiler}}$ valeurs ignorées

- pour tester si la pile est vide on teste si t est égal à 0,
- pour empiler une valeur v , on affecte $\text{tab}[t]=v$ et on incrémente t ,
- pour dépiler on renvoie $\text{tab}[t-1]$ et décrémente t .

Implémentation des piles

Plusieurs implémentations sont possibles :

- A l'aide d'une liste chaînée, la tête de la liste représente alors le sommet de la pile.



- Si la capacité de la pile n est bornée et connue en amont, on peut aussi utiliser un tableau de taille n et mémoriser la taille courante t de la pile.



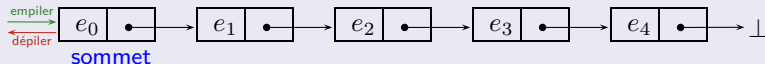
← dépiler sommet empiler → valeurs ignorées

- pour tester si la pile est vide on teste si t est égal à 0,
- pour empiler une valeur v , on affecte $\text{tab}[t]=v$ et on incrémente t ,
- pour dépiler on renvoie $\text{tab}[t-1]$ et décrémente t .

Implémentation des piles

Plusieurs implémentations sont possibles :

- A l'aide d'une liste chaînée, la tête de la liste représente alors le sommet de la pile.



- Si la capacité de la pile n est bornée et connue en amont, on peut aussi utiliser un tableau de taille n et mémoriser la taille courante t de la pile.



← dépiler sommet empiler → valeurs ignorées


- pour tester si la pile est vide on teste si t est égal à 0,
- pour empiler une valeur v , on affecte $\text{tab}[t]=v$ et on incrémente t ,
- pour dépiler on renvoie $\text{tab}[t-1]$ et décrémente t .

Ainsi, le sommet de la pile est toujours l'élément d'indice $t-1$ du tableau.


Remarques

- ⚠ Une pile peut-être implémentée par un tableau ou par une liste chaînée, mais une pile n'est ni l'un ni l'autre !


Remarques

-  Une pile peut-être implémentée par un tableau ou par une liste chaînée, mais une pile n'est ni l'un ni l'autre !
Par exemple en OCaml, même si la pile est implémentée par une `list`, on n'utilisera pas `List.length` car la taille ne fait pas partie de l'interface d'une pile.

Remarques

-  Une pile peut-être implémentée par un tableau ou par une liste chaînée, mais une pile n'est ni l'un ni l'autre !
Par exemple en OCaml, même si la pile est implémentée par une `list`, on n'utilisera pas `List.length` car la taille ne fait pas partie de l'interface d'une pile.
- En Ocaml, on se limite pour le moment à des structures de données *non mutables*, par conséquent, la modification d'une pile renvoie une nouvelle pile.

Remarques

-  Une pile peut-être implémentée par un tableau ou par une liste chaînée, mais une pile n'est ni l'un ni l'autre !
Par exemple en OCaml, même si la pile est implémentée par une `list`, on n'utilisera pas `List.length` car la taille ne fait pas partie de l'interface d'une pile.
- En Ocaml, on se limite pour le moment à des structures de données *non mutables*, par conséquent, la modification d'une pile renvoie une nouvelle pile.
Par exemple, en implémentant les piles avec le type `'a list` la fonction `dépiler` renvoie un élément dépilé et une nouvelle pile `dépiler` : `'a list -> 'a * 'a list`

Remarques

- **⚠** Une pile peut-être implémentée par un tableau ou par une liste chaînée, mais une pile n'est ni l'un ni l'autre !
Par exemple en OCaml, même si la pile est implémentée par une `list`, on n'utilisera pas `List.length` car la taille ne fait pas partie de l'interface d'une pile.
- En Ocaml, on se limite pour le moment à des structures de données *non mutables*, par conséquent, la modification d'une pile renvoie une nouvelle pile.
Par exemple, en implémentant les piles avec le type `'a list` la fonction `dépiler` renvoie une l'élément dépilé et une nouvelle pile `dépiler` : `'a list -> 'a * 'a list`
- En Ocaml, le module `Stack` accessible avec `open Stack` sera vu plus tard et permet de manipuler des piles mutables via l'interface :

Remarques

- **⚠** Une pile peut-être implémentée par un tableau ou par une liste chaînée, mais une pile n'est ni l'un ni l'autre !
Par exemple en OCaml, même si la pile est implémentée par une `list`, on n'utilisera pas `List.length` car la taille ne fait pas partie de l'interface d'une pile.
- En Ocaml, on se limite pour le moment à des structures de données *non mutables*, par conséquent, la modification d'une pile renvoie une nouvelle pile.
Par exemple, en implémentant les piles avec le type `'a list` la fonction `dépiler` renvoie un élément dépilé et une nouvelle pile `dépiler : 'a list -> 'a * 'a list`
- En Ocaml, le module `Stack` accessible avec `open Stack` sera vu plus tard et permet de manipuler des piles mutables via l'interface :
 - `Stack.create` de signature `() -> 'a t` qui crée une pile vide d'éléments de type `'a`.

Remarques

- **⚠** Une pile peut-être implémentée par un tableau ou par une liste chaînée, mais une pile n'est ni l'un ni l'autre !
Par exemple en OCaml, même si la pile est implémentée par une `list`, on n'utilisera pas `List.length` car la taille ne fait pas partie de l'interface d'une pile.
- En Ocaml, on se limite pour le moment à des structures de données *non mutables*, par conséquent, la modification d'une pile renvoie une nouvelle pile.
Par exemple, en implémentant les piles avec le type `'a list` la fonction `dépiler` renvoie un élément dépilé et une nouvelle pile `dépiler : 'a list -> 'a * 'a list`
- En Ocaml, le module `Stack` accessible avec `open Stack` sera vu plus tard et permet de manipuler des piles mutables via l'interface :
 - `Stack.create` de signature `() -> 'a t` qui crée une pile vide d'éléments de type `'a`.
 - `Stack.push` de signature `'a 'a t -> ()` qui empile un élément.

Remarques

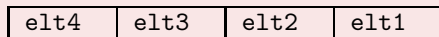
- **⚠** Une pile peut-être implémentée par un tableau ou par une liste chaînée, mais une pile n'est ni l'un ni l'autre !
Par exemple en OCaml, même si la pile est implémentée par une `list`, on n'utilisera pas `List.length` car la taille ne fait pas partie de l'interface d'une pile.
- En Ocaml, on se limite pour le moment à des structures de données *non mutables*, par conséquent, la modification d'une pile renvoie une nouvelle pile.
Par exemple, en implémentant les piles avec le type `'a list` la fonction `dépiler` renvoie une l'élément dépilé et une nouvelle pile `dépiler : 'a list -> 'a * 'a list`
- En Ocaml, le module `Stack` accessible avec `open Stack` sera vu plus tard et permet de manipuler des piles mutables via l'interface :
 - `Stack.create` de signature `() -> 'a t` qui crée une pile vide d'éléments de type `'a`.
 - `Stack.push` de signature `'a 'a t -> ()` qui empile un élément.
 - `Stack.pop` de signature `'a t -> 'a` qui dépile un élément.

Files

- Au niveau sémantique, une **file** est semblable à une file d'attente dans la vie de tous les jours.

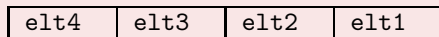
Files

- Au niveau sémantique, une **file** est semblable à une file d'attente dans la vie de tous les jours.



Files

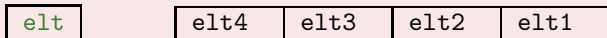
- Au niveau sémantique, une **file** est semblable à une file d'attente dans la vie de tous les jours.



- **Enfiler** signifie ajouter un élément en fin de file

Files

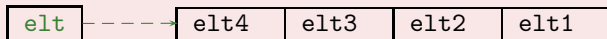
- Au niveau sémantique, une **file** est semblable à une file d'attente dans la vie de tous les jours.



- **Enfiler** signifie ajouter un élément en fin de file

Files

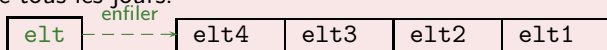
- Au niveau sémantique, une **file** est semblable à une file d'attente dans la vie de tous les jours.



- **Enfiler** signifie ajouter un élément en fin de file

Files

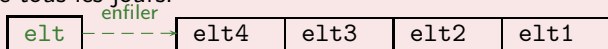
- Au niveau sémantique, une **file** est semblable à une file d'attente dans la vie de tous les jours.



- **Enfiler** signifie ajouter un élément en fin de file

Files

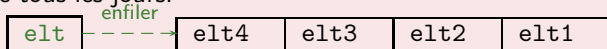
- Au niveau sémantique, une **file** est semblable à une file d'attente dans la vie de tous les jours.



- **Enfiler** signifie ajouter un élément en fin de file
- **Défiler** signifie retirer l'élément situé au début de la file.

Files

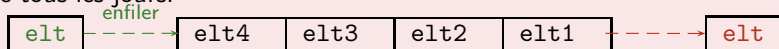
- Au niveau sémantique, une **file** est semblable à une file d'attente dans la vie de tous les jours.



- **Enfiler** signifie ajouter un élément en fin de file
- **Défiler** signifie retirer l'élément situé au début de la file.

Files

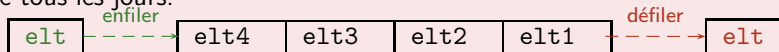
- Au niveau sémantique, une **file** est semblable à une file d'attente dans la vie de tous les jours.



- **Enfiler** signifie ajouter un élément en fin de file
- **Défiler** signifie retirer l'élément situé au début de la file.

Files

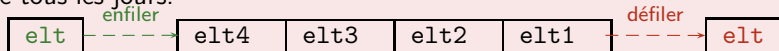
- Au niveau sémantique, une **file** est semblable à une file d'attente dans la vie de tous les jours.



- **Enfiler** signifie ajouter un élément en fin de file
- **Défiler** signifie retirer l'élément situé au début de la file.

Files

- Au niveau sémantique, une **file** est semblable à une file d'attente dans la vie de tous les jours.



- **Enfiler** signifie ajouter un élément en fin de file
- **Défiler** signifie retirer l'élément situé au début de la file.
- Ainsi le premier élément entré dans la file sera aussi le premier à en sortir, on dit qu'une file est une structure **FIFO** *First In First Out*

Files comme structures de données

L'interface d'une structure de données File se limite donc aux opérations suivantes :

Files comme structures de données

L'interface d'une structure de données File se limite donc aux opérations suivantes :

- `est_vide` qui renvoie un booléen indiquant si la file est vide ou non.

Files comme structures de données

L'interface d'une structure de données File se limite donc aux opérations suivantes :

- `est_vide` qui renvoie un booléen indiquant si la file est vide ou non.
- `enfiler` qui ajoute un élément à la fin de la file.

Files comme structures de données

L'interface d'une structure de données File se limite donc aux opérations suivantes :

- `est_vide` qui renvoie un booléen indiquant si la file est vide ou non.
- `enfiler` qui ajoute un élément à la fin de la file.
- `defiler` qui retire l'élément situé au début de la file (cela n'est possible que si la file n'est pas vide).

Files comme structures de données

L'interface d'une structure de données File se limite donc aux opérations suivantes :

- `est_vide` qui renvoie un booléen indiquant si la file est vide ou non.
- `enfiler` qui ajoute un élément à la fin de la file.
- `defiler` qui retire l'élément situé au début de la file (cela n'est possible que si la file n'est pas vide).

Utilisation

Comme pour les piles, cette structure de données a de nombreuses applications en informatique : file d'attente d'une imprimante, simulation de files d'attentes réelles, ...

Manipulation de files

Manipulation de files

- On considère la file : $F = \langle "E", "L", "S", "A" \rangle$. Quelle suite d'opération permet d'obtenir $F = \langle "N", "O", "E", "L" \rangle$?

Manipulation de files

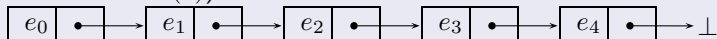
- On considère la file : $F = \langle "E", "L", "S", "A" \rangle$. Quelle suite d'opération permet d'obtenir $F = \langle "N", "O", "E", "L" \rangle$?
- On simule la file d'attente d'une imprimante à l'aide d'une file. A quelle opération sur cette file correspond l'envoi d'une nouvelle impression ? La fin de l'impression en cours ? Dans quel ordre seront effectuées les impressions ?

Implémentation avec une liste chaînée

L'une des implémentations possibles, afin d'obtenir des opérations en $\mathcal{O}(1)$ est de disposer d'un accès au premier maillon (pour défiler en $\mathcal{O}(1)$) et au dernier maillon (pour enfiler en $\mathcal{O}(1)$).

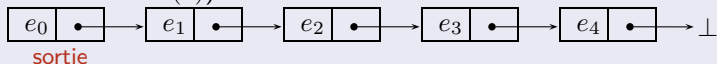
Implémentation avec une liste chaînée

L'une des implémentations possibles, afin d'obtenir des opérations en $\mathcal{O}(1)$ est de disposer d'un accès au premier maillon (pour défiler en $\mathcal{O}(1)$) et au dernier maillon (pour enfiler en $\mathcal{O}(1)$).



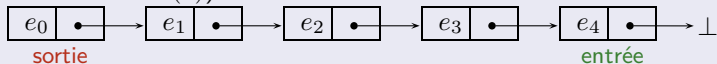
Implémentation avec une liste chaînée

L'une des implémentations possibles, afin d'obtenir des opérations en $\mathcal{O}(1)$ est de disposer d'un accès au premier maillon (pour défiler en $\mathcal{O}(1)$) et au dernier maillon (pour enfiler en $\mathcal{O}(1)$).



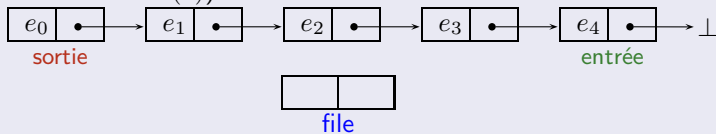
Implémentation avec une liste chaînée

L'une des implémentations possibles, afin d'obtenir des opérations en $\mathcal{O}(1)$ est de disposer d'un accès au premier maillon (pour défiler en $\mathcal{O}(1)$) et au dernier maillon (pour enfiler en $\mathcal{O}(1)$).



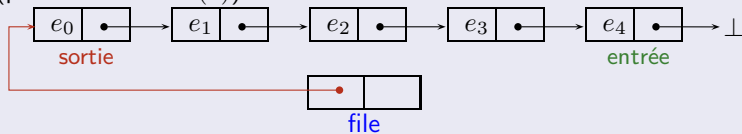
Implémentation avec une liste chaînée

L'une des implémentations possibles, afin d'obtenir des opérations en $\mathcal{O}(1)$ est de disposer d'un accès au premier maillon (pour défiler en $\mathcal{O}(1)$) et au dernier maillon (pour enfiler en $\mathcal{O}(1)$).



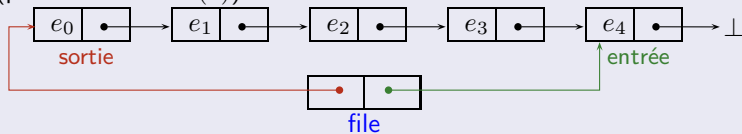
Implémentation avec une liste chaînée

L'une des implémentations possibles, afin d'obtenir des opérations en $\mathcal{O}(1)$ est de disposer d'un accès au premier maillon (pour défiler en $\mathcal{O}(1)$) et au dernier maillon (pour enfiler en $\mathcal{O}(1)$).



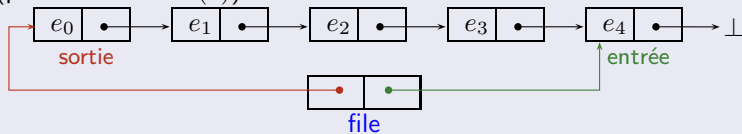
Implémentation avec une liste chaînée

L'une des implémentations possibles, afin d'obtenir des opérations en $\mathcal{O}(1)$ est de disposer d'un accès au premier maillon (pour défiler en $\mathcal{O}(1)$) et au dernier maillon (pour enfiler en $\mathcal{O}(1)$).



Implémentation avec une liste chaînée

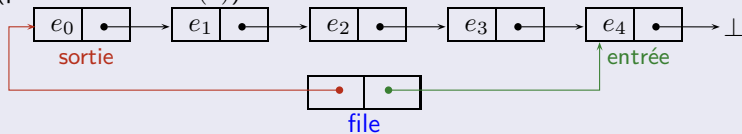
L'une des implémentations possibles, afin d'obtenir des opérations en $\mathcal{O}(1)$ est de disposer d'un accès au premier maillon (pour défiler en $\mathcal{O}(1)$) et au dernier maillon (pour enfiler en $\mathcal{O}(1)$).



! Le sens de parcours de la file est l'inverse de celui des maillons.

Implémentation avec une liste chaînée

L'une des implémentations possibles, afin d'obtenir des opérations en $\mathcal{O}(1)$ est de disposer d'un accès au premier maillon (pour défiler en $\mathcal{O}(1)$) et au dernier maillon (pour enfiler en $\mathcal{O}(1)$).



! Le sens de parcours de la file est l'inverse de celui des maillons.

```

1 struct file_s
2 {
3     maillon *tete;
4     maillon *queue;
5 };
6 typedef struct file_s file;

```

Implémentation avec un tableau

Si la capacité de la file n est bornée, on peut utiliser un tableau f qu'on traite circulairement (*ring buffer*). Pour cela, on maintient à jour une variable t contenant le nombre d'éléments de la file et une variable d contenant l'indice du prochain élément à défiler.

Implémentation avec un tableau

Si la capacité de la file n est bornée, on peut utiliser un tableau f qu'on traite circulairement (*ring buffer*). Pour cela, on maintient à jour une variable t contenant le nombre d'éléments de la file et une variable d contenant l'indice du prochain élément à défiler.

- pour défiler, on renvoie l'élément d'indice d , on décrémente t et on incrémente d (modulo n).

Implémentation avec un tableau

Si la capacité de la file n est bornée, on peut utiliser un tableau f qu'on traite circulairement (*ring buffer*). Pour cela, on maintient à jour une variable t contenant le nombre d'éléments de la file et une variable d contenant l'indice du prochain élément à défiler.

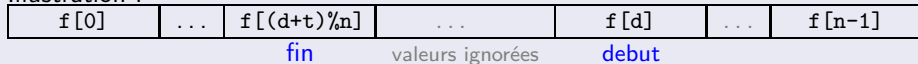
- pour défiler, on renvoie l'élément d'indice d , on décremente t et on incrémente d (modulo n).
- pour enfiler une valeur v lorsque la file n'est pas pleine, on affecte $tab[(d+t)\%n]=v$ et on incrémente t (modulo n).

Implémentation avec un tableau

Si la capacité de la file n est bornée, on peut utiliser un tableau f qu'on traite circulairement (*ring buffer*). Pour cela, on maintient à jour une variable t contenant le nombre d'éléments de la file et une variable d contenant l'indice du prochain élément à défiler.

- pour défiler, on renvoie l'élément d'indice d , on décrémente t et on incrémente d (modulo n).
- pour enfileur une valeur v lorsque la file n'est pas pleine, on affecte $tab[(d+t)\%n]=v$ et on incrémente t (modulo n).

Illustration :



Exemple introductif

- Rappeler les opérations de l'interface usuelle d'une pile et leur complexité.

Exemple introductif

- Rappeler les opérations de l'interface usuelle d'une pile et leur complexité.
- On suppose qu'on ajoute à cette interface une opération `multiPop` qui dépile la totalité des éléments de la pile. En notant n le nombre d'éléments de la pile, donner un grand \mathcal{O} de cette nouvelle opération.

Exemple introductif

- Rappeler les opérations de l'interface usuelle d'une pile et leur complexité.
- On suppose qu'on ajoute à cette interface une opération `multiPop` qui dépile la totalité des éléments de la pile. En notant n le nombre d'éléments de la pile, donner un grand \mathcal{O} de cette nouvelle opération.
- On considère une pile initialement vide S_0 , sur laquelle on effectue une suite d'opérations p_1, \dots, p_m , donner un \mathcal{O} de cette suite d'opérations. Que penser de cette majoration ?

Définitions

- On considère une structure de données \mathcal{S} munie d'un ensemble d'opérations \mathcal{P} . Sur une instance \mathcal{S}_0 de cette structure de données on effectue une suite d'opérations $\{p_1, \dots, p_n\}$ où $p_i \in \mathcal{P}$ pour $i \in \llbracket 1; n \rrbracket$:

$$\mathcal{S}_0 \xrightarrow{p_1} \mathcal{S}_1 \xrightarrow{p_2} \mathcal{S}_2 \dots \xrightarrow{p_n} \mathcal{S}_n$$

Définitions

- On considère une structure de données \mathcal{S} munie d'un ensemble d'opérations \mathcal{P} . Sur une instance \mathcal{S}_0 de cette structure de données on effectue une suite d'opérations $\{p_1, \dots, p_n\}$ où $p_i \in \mathcal{P}$ pour $i \in \llbracket 1; n \rrbracket$:

$$\mathcal{S}_0 \xrightarrow{p_1} \mathcal{S}_1 \xrightarrow{p_2} \mathcal{S}_2 \dots \xrightarrow{p_n} \mathcal{S}_n$$

En notant c_i le coût de p_i , la **complexité amortie** de chacune des opérations p_i est la moyenne arithmétique des $(c_i)_{1 \leq i \leq n}$:

$$\frac{1}{n} \sum_{k=1}^n c_k$$

Définitions

- On considère une structure de données \mathcal{S} munie d'un ensemble d'opérations \mathcal{P} . Sur une instance \mathcal{S}_0 de cette structure de données on effectue une suite d'opérations $\{p_1, \dots, p_n\}$ où $p_i \in \mathcal{P}$ pour $i \in \llbracket 1; n \rrbracket$:

$$\mathcal{S}_0 \xrightarrow{p_1} \mathcal{S}_1 \xrightarrow{p_2} \mathcal{S}_2 \dots \xrightarrow{p_n} \mathcal{S}_n$$

En notant c_i le coût de p_i , la **complexité amortie** de chacune des opérations p_i est la moyenne arithmétique des $(c_i)_{1 \leq i \leq n}$:

$$\frac{1}{n} \sum_{k=1}^n c_k$$

- Une fonction qui associe à chacune des instances \mathcal{S}_i de la structure de données un nombre réel *positif* et telle que $\Phi(\mathcal{S}_0) = 0$ est appelée **fonction de potentiel**.

Définitions

- On considère une structure de données \mathcal{S} munie d'un ensemble d'opérations \mathcal{P} . Sur une instance \mathcal{S}_0 de cette structure de données on effectue une suite d'opérations $\{p_1, \dots, p_n\}$ où $p_i \in \mathcal{P}$ pour $i \in \llbracket 1; n \rrbracket$:

$$\mathcal{S}_0 \xrightarrow{p_1} \mathcal{S}_1 \xrightarrow{p_2} \mathcal{S}_2 \dots \xrightarrow{p_n} \mathcal{S}_n$$

En notant c_i le coût de p_i , la **complexité amortie** de chacune des opérations p_i est la moyenne arithmétique des $(c_i)_{1 \leq i \leq n}$:

$$\frac{1}{n} \sum_{k=1}^n c_k$$

- Une fonction qui associe à chacune des instances \mathcal{S}_i de la structure de données un nombre réel *positif* et telle que $\Phi(\mathcal{S}_0) = 0$ est appelée **fonction de potentiel**.
- On définit le **coût amorti** \hat{c}_i d'une opération p_i par :
$$\hat{c}_i = c_i + \Phi(\mathcal{S}_i) - \Phi(\mathcal{S}_{i-1})$$

Théorème d'amortissement

La somme des coût des opérations $(p_i)_{1 \leq i \leq n}$ est majoré par la somme de leur coût amorti :

$$\sum_{k=1}^n c_i \leq \sum_{k=1}^n \hat{c}_i$$

Théorème d'amortissement

La somme des coût des opérations $(p_i)_{1 \leq i \leq n}$ est majoré par la somme de leur coût amorti :

$$\sum_{k=1}^n c_i \leq \sum_{k=1}^n \hat{c}_i$$

Exemple de la pile avec multipop

On définit la fonction de potentiel $\Phi(S_i) =$ nombre d'éléments dans S_i (noté n_i), Φ est bien à valeur positives et $\Phi(S_0) = 0$. Alors, puisque $\hat{c}_i = \Phi(S_i) - \Phi(S_{i-1})$,

Théorème d'amortissement

La somme des coût des opérations $(p_i)_{1 \leq i \leq n}$ est majoré par la somme de leur coût amorti :

$$\sum_{k=1}^n c_i \leq \sum_{k=1}^n \hat{c}_i$$

Exemple de la pile avec multipop

On définit la fonction de potentiel $\Phi(S_i) =$ nombre d'éléments dans S_i (noté n_i), Φ est bien à valeur positives et $\Phi(S_0) = 0$. Alors, puisque $\hat{c}_i = \Phi(S_i) - \Phi(S_{i-1})$,

- Si $c_i = \text{push}$, $\hat{c}_i = 1 + (n_i - n_{i-1}) = 2$

C8 Structures de données séquentielles

6. Complexité amortie

Théorème d'amortissement

La somme des coût des opérations $(p_i)_{1 \leq i \leq n}$ est majoré par la somme de leur coût amorti :

$$\sum_{k=1}^n c_i \leq \sum_{k=1}^n \hat{c}_i$$

Exemple de la pile avec multipop

On définit la fonction de potentiel $\Phi(S_i) =$ nombre d'éléments dans S_i (noté n_i), Φ est bien à valeur positives et $\Phi(S_0) = 0$. Alors, puisque $\hat{c}_i = \Phi(S_i) - \Phi(S_{i-1})$,

- Si $c_i = \text{push}$, $\hat{c}_i = 1 + (n_i - n_{i-1}) = 2$
- Si $c_i = \text{pop}$, $\hat{c}_i = 1 + (n_i - n_{i-1}) = 0$

Théorème d'amortissement

La somme des coût des opérations $(p_i)_{1 \leq i \leq n}$ est majoré par la somme de leur coût amorti :

$$\sum_{k=1}^n c_i \leq \sum_{k=1}^n \hat{c}_i$$

Exemple de la pile avec multipop

On définit la fonction de potentiel $\Phi(S_i) =$ nombre d'éléments dans S_i (noté n_i), Φ est bien à valeur positives et $\Phi(S_0) = 0$. Alors, puisque $\hat{c}_i = \Phi(S_i) - \Phi(S_{i-1})$,

- Si $c_i = \text{push}$, $\hat{c}_i = 1 + (n_i - n_{i-1}) = 2$
- Si $c_i = \text{pop}$, $\hat{c}_i = 1 + (n_i - n_{i-1}) = 0$
- Si $c_i = \text{multipop}$, $\hat{c}_i = n_{i-1} + (0 - n_{i-1}) = 0$

C8 Structures de données séquentielles

6. Complexité amortie

Théorème d'amortissement

La somme des coût des opérations $(p_i)_{1 \leq i \leq n}$ est majoré par la somme de leur coût amorti :

$$\sum_{k=1}^n c_i \leq \sum_{k=1}^n \hat{c}_i$$

Exemple de la pile avec multipop

On définit la fonction de potentiel $\Phi(S_i) =$ nombre d'éléments dans S_i (noté n_i), Φ est bien à valeur positives et $\Phi(S_0) = 0$. Alors, puisque $\hat{c}_i = \Phi(S_i) - \Phi(S_{i-1})$,

- Si $c_i = \text{push}$, $\hat{c}_i = 1 + (n_i - n_{i-1}) = 2$
- Si $c_i = \text{pop}$, $\hat{c}_i = 1 + (n_i - n_{i-1}) = 0$
- Si $c_i = \text{multipop}$, $\hat{c}_i = n_{i-1} + (0 - n_{i-1}) = 0$

Par conséquent, la complexité amortie d'une suite de m opérations sur cette structure de données est proportionnel au nombre d'opérations (c'est un $\mathcal{O}(m)$), tout se passe comme si chaque opération avait un coût constant.

Exemple : file implémentée par deux piles

On implémente une file à l'aide de deux piles initialement vides, P_e (pile d'entrée) et P_s (pile de sortie) de la façon suivante :

- L'opération `enfiler` consiste à empiler un élément dans P_e ,
- L'opération `défiler` consiste à dépiler un élément de P_s , si cette pile est vide alors on dépile la totalité de P_e dans P_s .

Exemple : file implémentée par deux piles

On implémente une file à l'aide de deux piles initialement vides, P_e (pile d'entrée) et P_s (pile de sortie) de la façon suivante :

- L'opération `enfiler` consiste à empiler un élément dans P_e ,
 - L'opération `défiler` consiste à dépiler un élément de P_s , si cette pile est vide alors on dépile la totalité de P_e dans P_s .
- 1 Executer quelques opérations `enfiler` et `defiler` (illustrer éventuellement par un schéma).

Exemple : file implémentée par deux piles

On implémente une file à l'aide de deux piles initialement vides, P_e (pile d'entrée) et P_s (pile de sortie) de la façon suivante :

- L'opération `enfiler` consiste à empiler un élément dans P_e ,
 - L'opération `défiler` consiste à dépiler un élément de P_s , si cette pile est vide alors on dépile la totalité de P_e dans P_s .
- 1 Executer quelques opérations `enfiler` et `defiler` (illustrer éventuellement par un schéma).
 - 2 Quelle est la complexité dans le pire des cas des opérations ?

Exemple : file implémentée par deux piles

On implémente une file à l'aide de deux piles initialement vides, P_e (pile d'entrée) et P_s (pile de sortie) de la façon suivante :

- L'opération `enfiler` consiste à empiler un élément dans P_e ,
 - L'opération `défiler` consiste à dépiler un élément de P_s , si cette pile est vide alors on dépile la totalité de P_e dans P_s .
- 1 Exécuter quelques opérations `enfiler` et `defiler` (illustrer éventuellement par un schéma).
 - 2 Quelle est la complexité dans le pire des cas des opérations ?
 - 3 Etudier la complexité amortie d'une suite d'opérations sur cette structure de données.
 - 🌀 On pourra considérer la fonction Φ renvoyant le nombre d'éléments de P_e .