DUT SRC – IUT de Marne-la-Vallée
12/02/2014
M2202 - Algorithmique

Cours 3
Tris, structures de données

UNIVERSITÉ PARIS-EST MARNE-LA-VALLÉE

iut Marne-la-Vallée

Philippe Gambette

Résumé de l'épisode précédent

- **Tris**

Pour ranger un ensemble d'éléments tous comparables deux à deux selon un certain ordre.

Cas de base : tri d'un **tableau d'entiers** pour l'ordre **≤**

Plusieurs algorithmes possibles :
- tri à bulles (cours + TD3)
- tri par sélection (cours)
- tri par insertion (TD3)
- ...

plus ou moins rapide... compter le nombre de comparaisons, d'échanges, etc.

Sources

- Cours de Jean-François Berdjugin à l'IUT de Grenoble
http://berdjugin.com/enseignements/inf/inf220/

- Cours de Xavier Heurtebise à l'IUT de Provence
http://x.heurtebise.free.fr

- *Le livre de Java premier langage*, d'A. Tasso

- http://xkcd.com, http://xkcd.free.fr

Plan du cours 3 – Tris, structures de données

- Le tri par sélection
- Complexité des tris
- Listes
- Piles et files
- Arbres

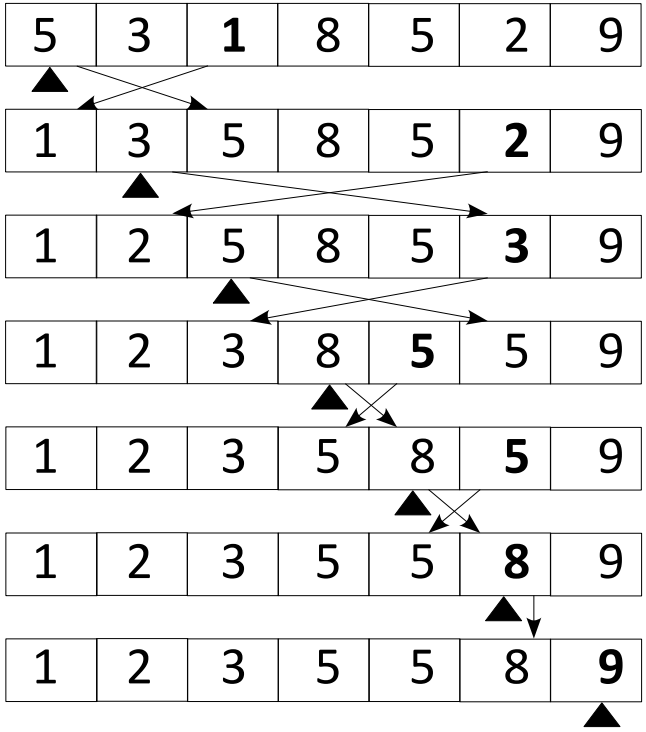
Plan du cours 3 – Tris, structures de données

- Le tri par sélection
- Complexité des tris
- Listes
- Piles et files
- Arbres

Tri par sélection (ou tri par extraction)

Idée : à la *i*-ième étape, on prend le plus petit élément à partir de la *i*-ième case (comprise) et on l'échange avec l'élément de la *i*-ième case.

Exemple avec un tableau d'entiers :



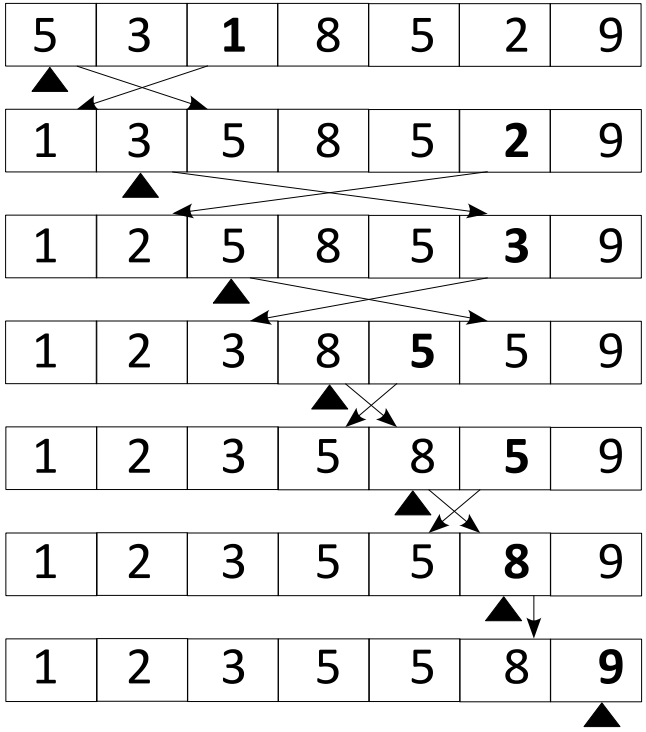
Algorithme **positionMinimum**
Entrée : tableau d'entiers *tab*, entier *debut*
Type de sortie : entier
Variables : entiers *position, i* et *min*
Début

Fin

Tri par sélection (ou tri par extraction)

Idée : à la *i*-ième étape, on prend le plus petit élément à partir de la *i*-ième case (comprise) et on l'échange avec l'élément de la *i*-ième case.

Exemple avec un tableau d'entiers :

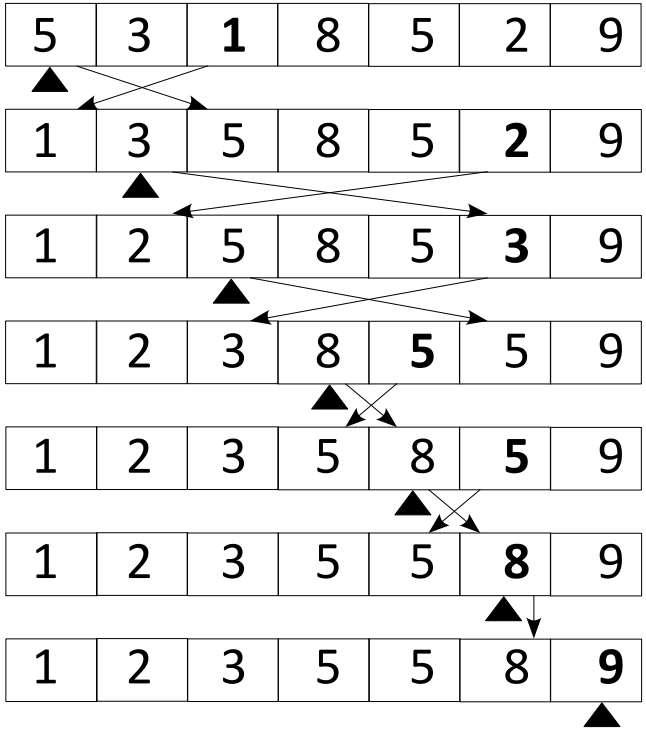


Algorithme **positionMinimum**
Entrée : tableau d'entiers *tab*, entier *debut*
Type de sortie : entier
Variables : entiers *position, i* et *min*
Début
  position ← *debut*
  min ← **Case**(*tab*,*debut*)
  i ← *debut*
  Tant que *i*≤**Longueur**(*tab*) faire :
    Si **Case**(*tab*,*i*)<*min* alors :
      position ← *i*
      min ← **Case**(*tab*,*i*)
    Fin si
    i ← *i*+1
  Fin Tant que
  renvoyer *position*
Fin

Tri par sélection (ou tri par extraction)

Idée : à la *i*-ième étape, on prend le plus petit élément à partir de la *i*-ième case (comprise) et on l'échange avec l'élément de la *i*-ième case.

Exemple avec un tableau d'entiers :



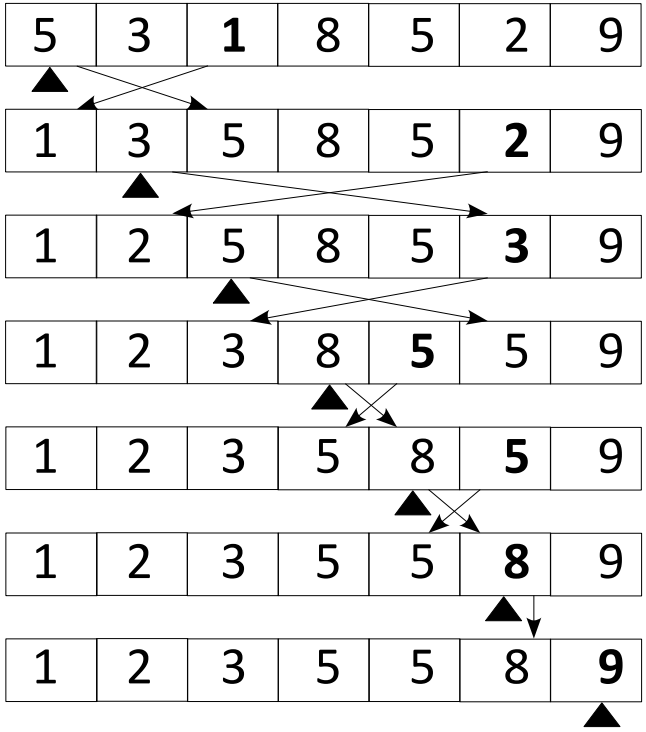
Algorithme **triSelection**
Entrée : tableau d'entiers *tab*
Type de sortie : tableau d'entiers
Variables : entiers *i, temp*, *posMin*
Début

Fin

Tri par sélection (ou tri par extraction)

Idée : à la *i*-ième étape, on prend le plus petit élément à partir de la *i*-ième case (comprise) et on l'échange avec l'élément de la *i*-ième case.

Exemple avec un tableau d'entiers :



Algorithme **triSelection**
Entrée : tableau d'entiers *tab*
Type de sortie : tableau d'entiers
Variables : entiers *i, temp*, *posMin*
Début
  i ← 1
  Tant que *i*<**Longueur**(*tab*) faire :
    posMin ← **positionMinimum**(*tab*,*i*)
    //*posMin* peut être égal à *i*
    temp ← **Case**(*tab*,*i*)
    Case(*tab*,*i*) ← **Case**(*tab*,*posMin*)
    Case(*tab*,*posMin*) ← *temp*
    i ← *i*+1
  Fin Tant que
  renvoyer *tab*
Fin

Plan du cours 3 – Tris, structures de données

- Le tri par sélection
- Complexité des tris
- Listes
- Piles et files
- Arbres

Complexité des tris

Combien de comparaisons ? (lié au **temps d'exécution** de l'algorithme)

Sur un exemple ou dans le pire des cas, pour *n* éléments.

Tri à bulles

Tri par sélection

Complexité des tris

Combien de comparaisons ? (lié au **temps d'exécution** de l'algorithme)

Sur un exemple ou dans le pire des cas, pour *n* éléments.

Tri à bulles

Dans tous les cas : ≤ *n* étapes, *n*-1 comparaisons par étape
donc ≤ *n* x (*n*-1) comparaisons au total

Tri par sélection

Pour *n* éléments : *n* étapes, *n-i* comparaisons par étape pour trouver le min

Complexité des tris

Combien de comparaisons ? (lié au **temps d'exécution** de l'algorithme)

Sur un exemple ou dans le pire des cas, pour *n* éléments.

Tri à bulles

Dans tous les cas : ≤ *n* étapes, *n*-1 comparaisons par étape
donc ≤ *n* x (*n*-1) comparaisons au total

Tri par sélection

Pour *n* éléments : *n* étapes, *n-i* comparaisons par étape pour trouver le min

étape 1 : *n*-1 comparaisons
étape 2 : *n*-2 comparaisons

…

étape n-2 : 2 comparaisons
étape n-1 : 1 comparaison
étape n : 0 comparaison

Complexité des tris

Combien de comparaisons ? (lié au **temps d'exécution** de l'algorithme)

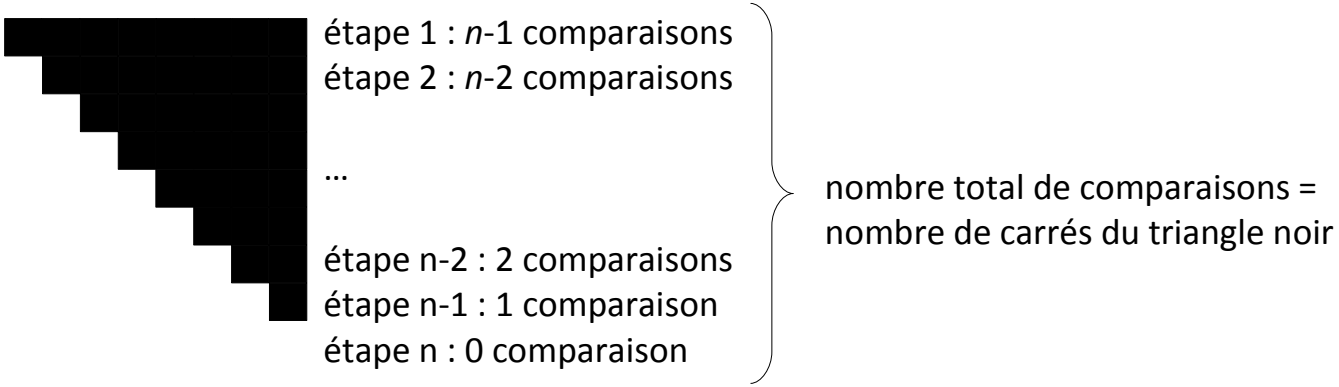
Sur un exemple ou dans le pire des cas, pour *n* éléments.

Tri à bulles

Dans tous les cas : ≤ *n* étapes, *n*-1 comparaisons par étape
donc ≤ *n* x (*n*-1) comparaisons au total

Tri par sélection

Pour *n* éléments : *n* étapes, *n-i* comparaisons par étape pour trouver le min



Complexité des tris

Combien de comparaisons ? (lié au **temps d'exécution** de l'algorithme)

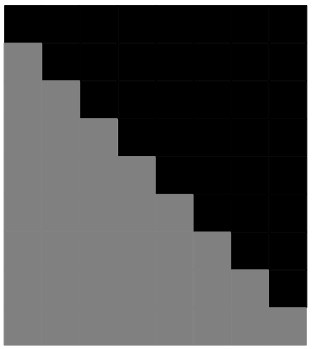
Sur un exemple ou dans le pire des cas, pour *n* éléments.

Tri à bulles

Dans tous les cas : ≤ *n* étapes, *n*-1 comparaisons par étape
donc ≤ *n* x (*n*-1) comparaisons au total

Tri par sélection

Pour *n* éléments : *n* étapes, *n-i* comparaisons par étape pour trouver le min



étape 1 : *n*-1 comparaisons
étape 2 : *n*-2 comparaisons

…

étape n-2 : 2 comparaisons
étape n-1 : 1 comparaison
étape n : 0 comparaison

nombre total de comparaisons =
nombre de carrés du triangle noir =
nombre de carrés du rectangle divisé par 2 =

Complexité des tris

Combien de comparaisons ? (lié au **temps d'exécution** de l'algorithme)

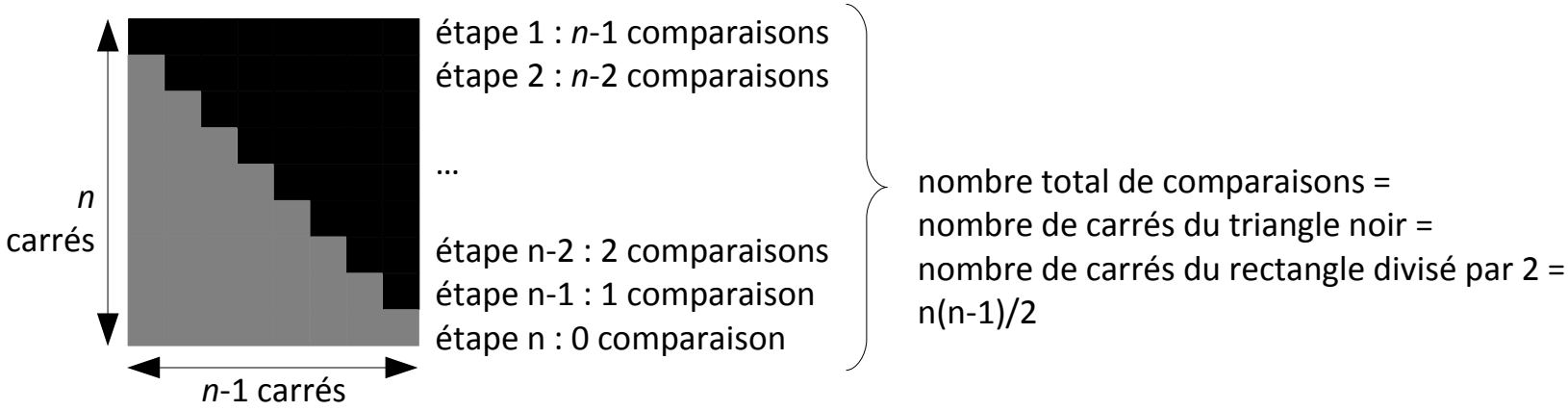
Sur un exemple ou dans le pire des cas, pour *n* éléments.

Tri à bulles

Dans tous les cas : ≤ *n* étapes, *n*-1 comparaisons par étape
donc ≤ *n* x (*n*-1) comparaisons au total

Tri par sélection

Pour *n* éléments : *n* étapes, *n-i* comparaisons par étape pour trouver le min



Complexité des tris

Combien de comparaisons ? (lié au **temps d'exécution** de l'algorithme)

Sur un exemple ou dans le pire des cas, pour *n* éléments.

Tri à bulles

Dans le pire des cas (tableau trié dans le sens inverse) : $n \times (n-1)$ comparaisons
Dans le meilleur cas (tableau trié) : $n-1$ comparaisons

Tri par sélection

Dans tous les cas : $n \times (n-1)/2$ comparaisons

Plan du cours 3 – Tris, structures de données

- Le tri par sélection
- Complexité des tris
- Listes
- Piles et files
- Arbres

Intérêt des listes

Le problème des tableaux :

- taille fixe

- insertion “difficile” d'un élément

- suppression “difficile” d'un élément

Intérêt des listes

L'intérêt des listes :

- taille variable

- insertion facile d'un élément

- suppression facile d'un élément

Mais :

- accès “difficile” au *i*-ième élément

Définition récursive d'une liste “simplement chaînée”

Une liste “simplement chaînée” est :

- soit une liste vide

- soit une première case qui contient une valeur, et qui pointe vers une liste

Définition récursive d'une liste d'entiers

Une liste d'entiers est :

- soit une liste vide

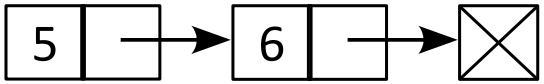
- soit une première case qui contient un entier, et qui pointe vers une liste d'entiers

Définition illustrée d'une liste d'entiers

Une liste d'entiers est :

- soit une liste vide

- soit une première case qui contient un entier, et qui pointe vers une liste d'entiers

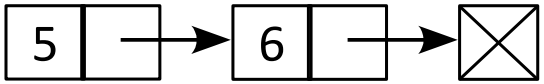


Définition illustrée d'une liste d'entiers

Une liste d'entiers est :

- soit une liste vide

- soit une première case qui contient un entier, et qui pointe vers une liste d'entiers



Liste “simplement chaînée” : la chaîne d'une ancre

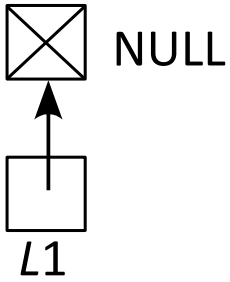
Quand l'ancre est jetée et toute la chaîne déroulée, on a facilement accès au premier maillon (sur le bateau). Pour les suivants il faut remonter l'ancre.
Quand on a un maillon sous les yeux on peut facilement le supprimer (le casser et accrocher le précédent au suivant), en insérer un...

Source image : http://virginiejmr.free.fr/serie_salle_de_bain.htm

Définition illustrée d'une liste d'entiers

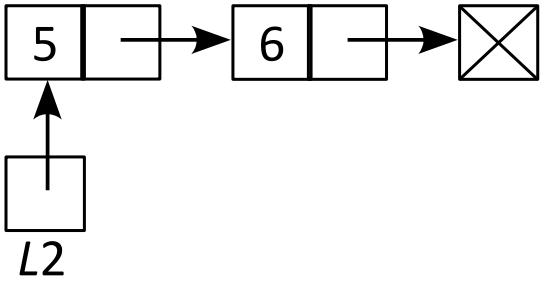
Une liste d'entiers est :

- soit une liste vide



L1 ← NULL
L2 ← NULL
L3 ← L1

- soit une première case qui contient un entier, et qui pointe vers une liste d'entiers

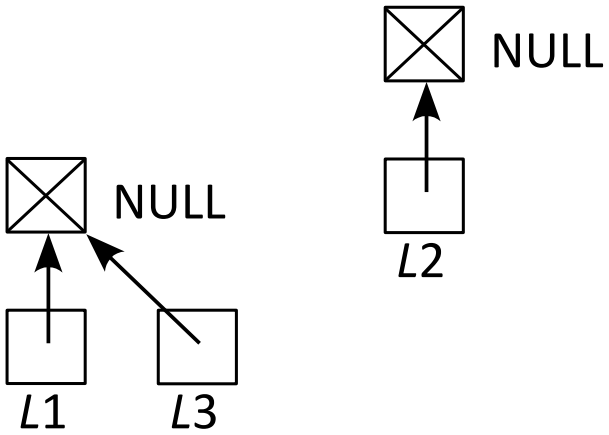


L2 ← **creeListe**(5,**creeListe**(6, NULL))

Définition illustrée d'une liste d'entiers

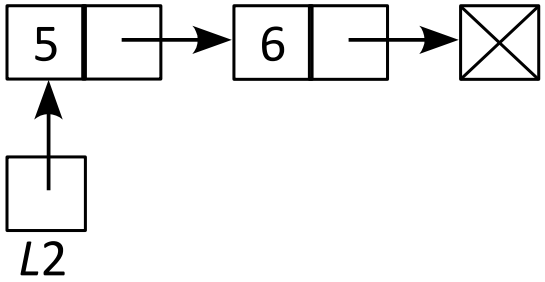
Une liste d'entiers est :

- soit une liste vide



$L1 \leftarrow$ NULL
$L2 \leftarrow$ NULL
$L3 \leftarrow L1$

- soit une première case qui contient un entier, et qui pointe vers une liste d'entiers

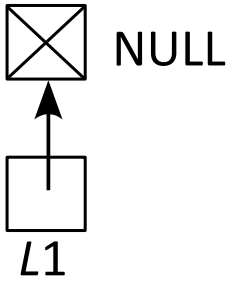


$L2 \leftarrow$ **creeListe**(5,**creeListe**(6, NULL))

Fonctions de base sur les listes

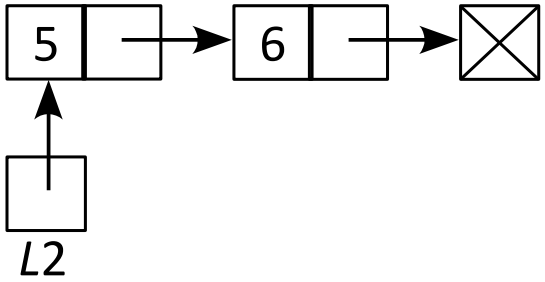
Une liste d'entiers est :

- soit une liste vide



$L1 \leftarrow$ NULL

- soit une première case qui contient un entier, et qui pointe vers une liste d'entiers



$L2 \leftarrow$ **creeListe**(5,**creeListe**(6, NULL))

Sur une liste $L1$ ou $L2$, on peut :

- savoir si elle est vide : $L1$ = NULL ?

si $L2$ est non vide :

- accéder à la valeur de son premier élément : **tete**($L2$)

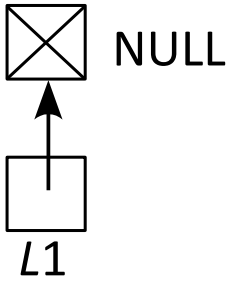
si $L2$ est non vide :

- accéder à la liste qui commence à la case suivante : **suivant**($L2$)

Fonctions de base sur les listes

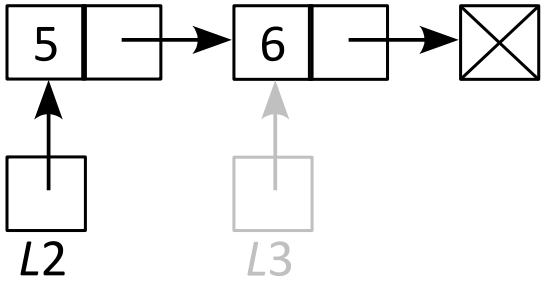
Une liste d'entiers est :

- soit une liste vide



$L1 \leftarrow$ NULL

- soit une première case qui contient un entier, et qui pointe vers une liste d'entiers



$L2 \leftarrow$ **creeListe**(5,**creeListe**(6, NULL))

Sur une liste $L1$ ou $L2$, on peut :

- savoir si elle est vide : $L1$ = NULL ?

si $L2$ est non vide :

- accéder à la valeur de son premier élément : **tete**($L2$) renvoie 5

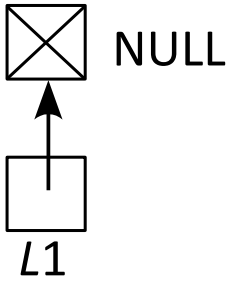
si $L2$ est non vide :

- accéder à la liste qui commence à la case suivante : **suivant**($L2$) renvoie $L3$

Fonctions de base sur les listes

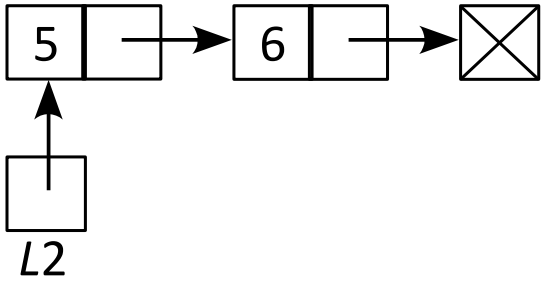
Une liste d'entiers est :

- soit une liste vide



$L1 \leftarrow$ NULL

- soit une première case qui contient un entier, et qui pointe vers une liste d'entiers



$L2 \leftarrow$ **creeListe**(5,**creeListe**(6, NULL))
$L3 \leftarrow$ **suivant**($L2$)
$L5 \leftarrow$ **suivant**($L2$)
tete($L5$) $\leftarrow$ 4

Sur une liste $L1$ ou $L2$, on peut :

- savoir si elle est vide : $L1$ = NULL ?

si $L2$ est non vide :

- accéder à la valeur de son premier élément : **tete**($L2$) renvoie 5

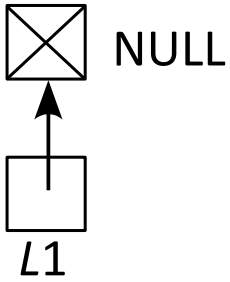
si $L2$ est non vide :

- accéder à la liste qui commence à la case suivante : **suivant**($L2$)

Fonctions de base sur les listes

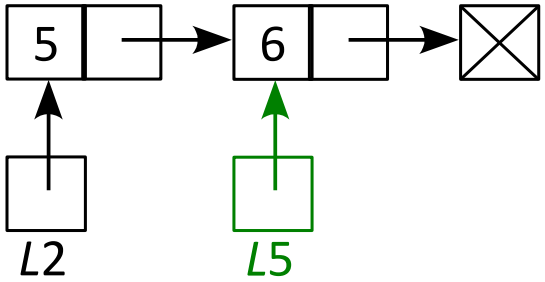
Une liste d'entiers est :

- soit une liste vide



$L1 \leftarrow$ NULL

- soit une première case qui contient un entier, et qui pointe vers une liste d'entiers



$L2 \leftarrow$ **creeListe**(5,**creeListe**(6, NULL))
$L5 \leftarrow$ **suivant**($L2$)

Sur une liste $L1$ ou $L2$, on peut :

- savoir si elle est vide : $L1$ = NULL ?

si $L2$ est non vide :

- accéder à la valeur de son premier élément : **tete**($L2$) renvoie 5

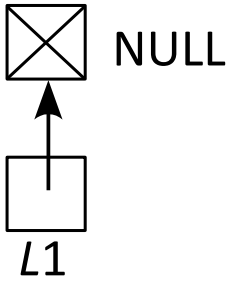
si $L2$ est non vide :

- accéder à la liste qui commence à la case suivante : **suivant**($L2$)

Fonctions de base sur les listes

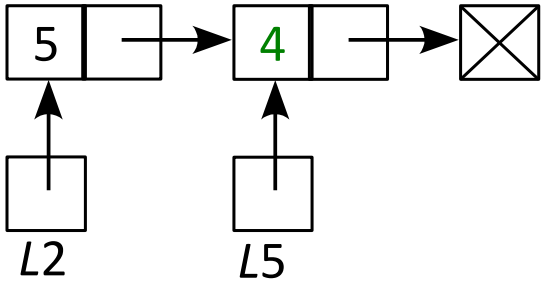
Une liste d'entiers est :

- soit une liste vide



$L1 \leftarrow$ NULL

- soit une première case qui contient un entier, et qui pointe vers une liste d'entiers



$L2 \leftarrow$ **creeListe**(5,**creeListe**(6, NULL))
$L5 \leftarrow$ **suivant**($L2$)
tete($L5$) $\leftarrow$ 4

Sur une liste $L1$ ou $L2$, on peut :

- savoir si elle est vide : $L1$ = NULL ?

si $L2$ est non vide :

- accéder à la valeur de son premier élément : **tete**($L2$) renvoie 5

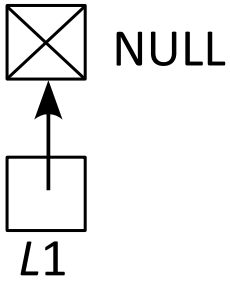
si $L2$ est non vide :

- accéder à la liste qui commence à la case suivante : **suivant**($L2$)

Fonctions de base sur les listes

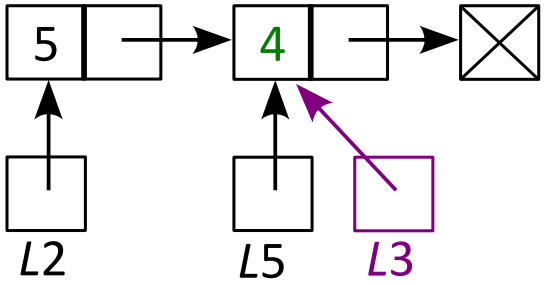
Une liste d'entiers est :

- soit une liste vide



$L1 \leftarrow$ NULL

- soit une première case qui contient un entier, et qui pointe vers une liste d'entiers



$L2 \leftarrow$ **creeListe**(5,**creeListe**(6, NULL))
$L5 \leftarrow$ **suivant**($L2$)
tete($L5$) $\leftarrow$ 4

Sur une liste $L1$ ou $L2$, on peut :

- savoir si elle est vide : $L1$ = NULL ?

si $L2$ est non vide :

- accéder à la valeur de son premier élément : **tete**($L2$) renvoie 5

si $L2$ est non vide :

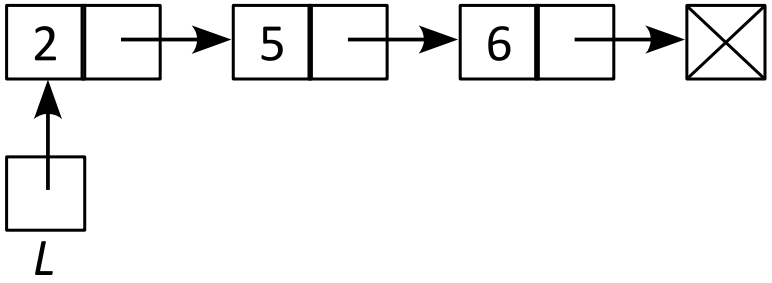
- accéder à la liste qui commence à la case suivante : **suivant**($L2$) renvoie $L3$

Premières fonctions sur les listes

Modification de l'élément en tête d'une liste

Ecrire l'algorithme **remplaceTete** qui prend en entrée une liste d'entiers *L* et un entier *a*, et renvoie la liste *L* dont le premier élément est remplacé par *a*

Algorithme **remplaceTete**
Entrées : liste d'entiers *L*, entier *a*
Type de sortie : liste d'entiers
Début



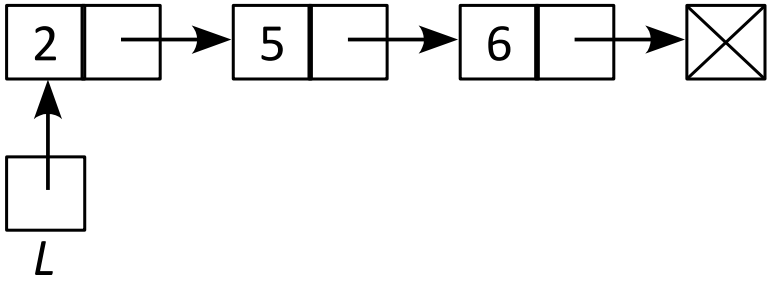
Fin

Premières fonctions sur les listes

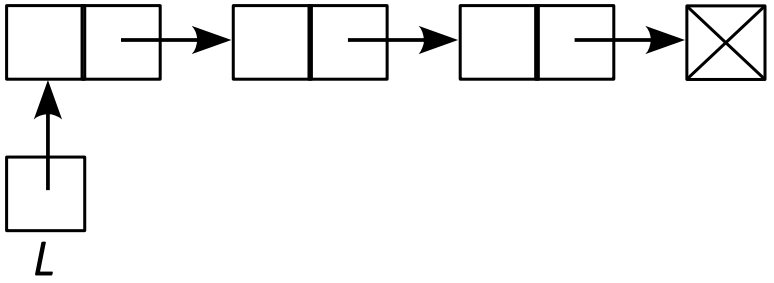
Modification de l'élément en tête d'une liste

Ecrire l'algorithme **remplaceTete** qui prend en entrée une liste d'entiers *L* et un entier *a*, et renvoie la liste *L* dont le premier élément est remplacé par *a*

Algorithme **remplaceTete**
Entrées : liste d'entiers *L*, entier *a*
Type de sortie : liste d'entiers
Début



remplaceTete(*L*,5) :



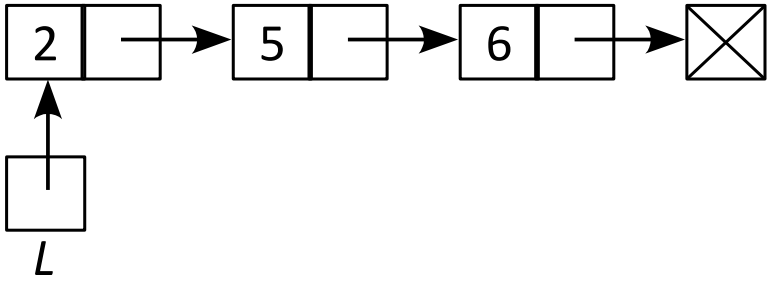
Fin

Premières fonctions sur les listes

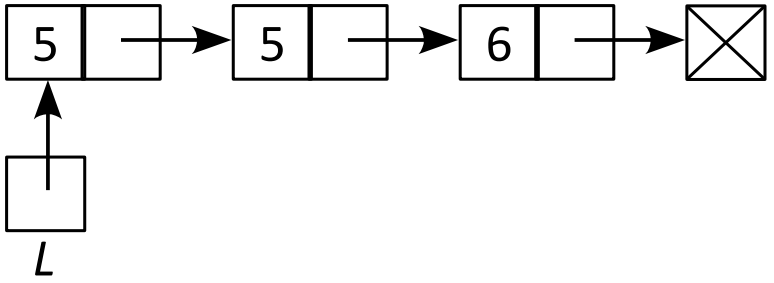
Modification de l'élément en tête d'une liste

Ecrire l'algorithme **remplaceTete** qui prend en entrée une liste d'entiers *L* et un entier *a*, et renvoie la liste *L* dont le premier élément est remplacé par *a*

Algorithme **remplaceTete**
Entrées : liste d'entiers *L*, entier *a*
Type de sortie : liste d'entiers
Début
    …
    tete(*L*) ← *a*
    renvoyer *L*
    …
Fin



remplaceTete(*L*,5) :

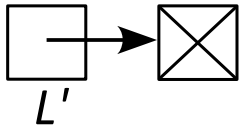


Premières fonctions sur les listes

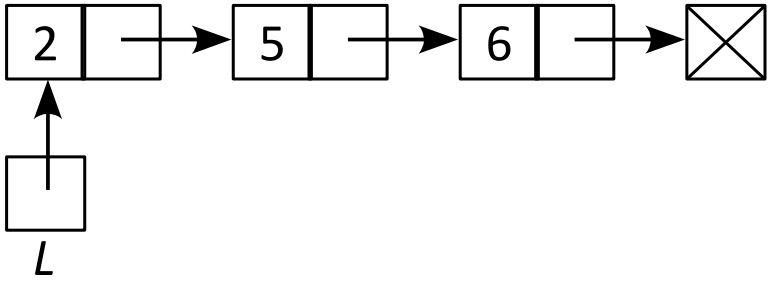
Modification de l'élément en tête d'une liste

Ecrire l'algorithme **remplaceTete** qui prend en entrée une liste d'entiers *L* et un entier *a*, et renvoie la liste *L* dont le premier élément est remplacé par *a*

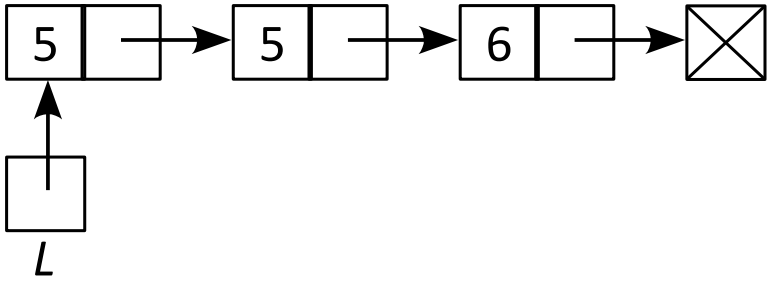
Algorithme **remplaceTete**
Entrées : liste d'entiers *L*, entier *a*
Type de sortie : liste d'entiers
Début
    …
    tete(*L*) ← *a*
    renvoyer *L*
    …
Fin



ne fonctionne pas sur la liste vide *L'*



remplaceTete(*L*,5) :

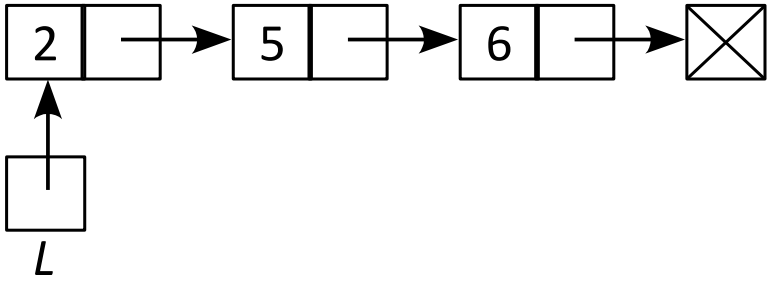


Premières fonctions sur les listes

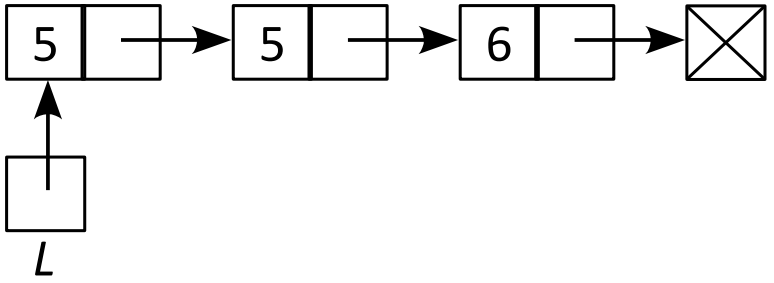
Modification de l'élément en tête d'une liste

Ecrire l'algorithme **remplaceTete** qui prend en entrée une liste d'entiers *L* et un entier *a*, et renvoie la liste *L* dont le premier élément est remplacé par *a*

Algorithme **remplaceTete**
Entrées : liste d'entiers *L*, entier *a*
Type de sortie : liste d'entiers
Début
    si *L* ≠ NULL alors :
        tete(*L*) ← *a*
    Fin Si
    renvoyer *L*
Fin



remplaceTete(*L*,5) :

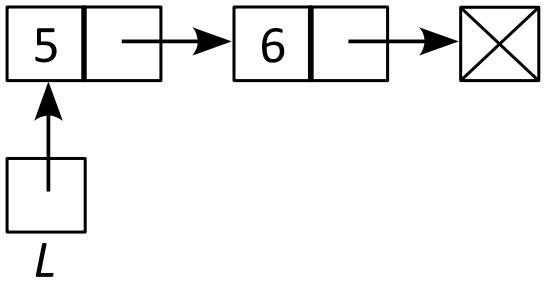


Facile pour les listes, facile pour les tableaux

Premières fonctions sur les listes

Suppression de l'élément en tête d'une liste

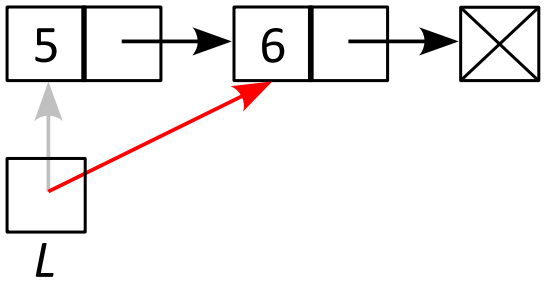
Par exemple, supprimer 5 au début de la liste *L*=**creeListe**(5,**creeListe**(6, NULL))



Premières fonctions sur les listes

Suppression de l'élément en tête d'une liste

Par exemple, supprimer 5 au début de la liste *L*=**creeListe**(5,**creeListe**(6, NULL))

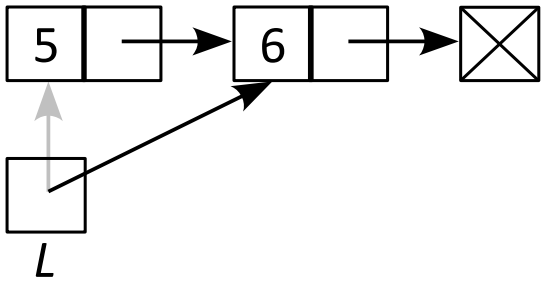


Premières fonctions sur les listes

Suppression de l'élément en tête d'une liste

Par exemple, supprimer 5 au début de la liste *L*=**creeListe**(5,**creeListe**(6, NULL))

$L \leftarrow$ **suivant**(*L*)

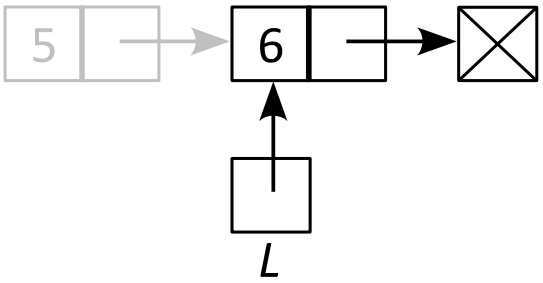


Premières fonctions sur les listes

Suppression de l'élément en tête d'une liste

Par exemple, supprimer 5 au début de la liste *L*=**creeListe**(5,**creeListe**(6, NULL))

$L \leftarrow$ **suivant**(*L*)

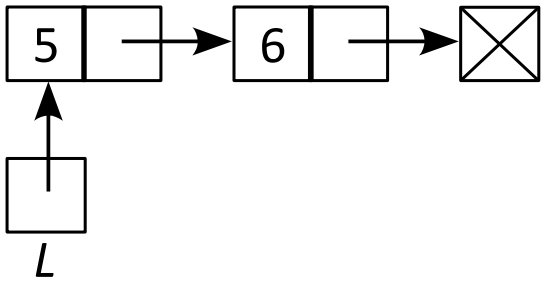


Facile pour les listes, “difficile” pour les tableaux

Premières fonctions sur les listes

Insertion d'un élément en tête d'une liste

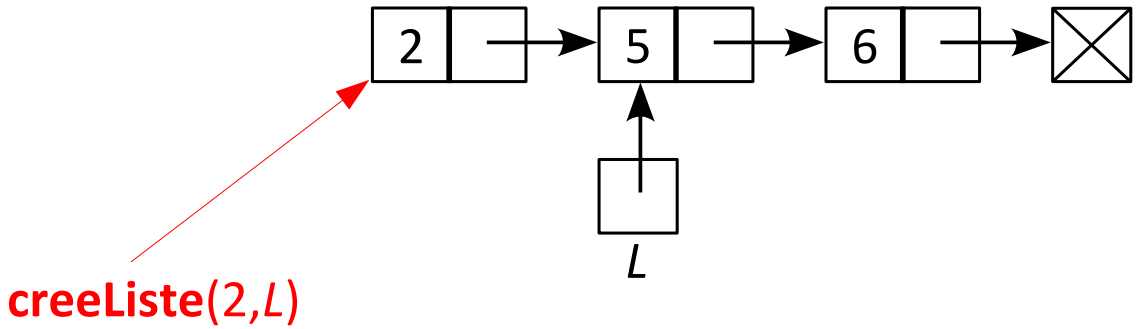
Par exemple, insérer 2 au début de la liste *L*=**creeListe**(5,**creeListe**(6, NULL))



Premières fonctions sur les listes

Insertion d'un élément en tête d'une liste

Par exemple, insérer 2 au début de la liste *L*=**creeListe**(5,**creeListe**(6, NULL))

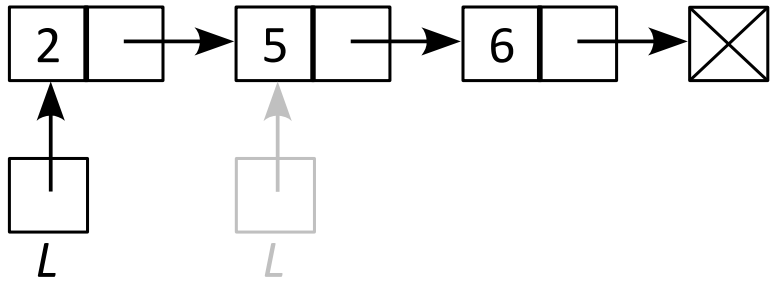


Premières fonctions sur les listes

Insertion d'un élément en tête d'une liste

Par exemple, insérer 2 au début de la liste *L*=**creeListe**(5,**creeListe**(6, NULL))

$L \leftarrow$ **creeListe**(2,*L*)



Facile pour les listes, “difficile” pour les tableaux

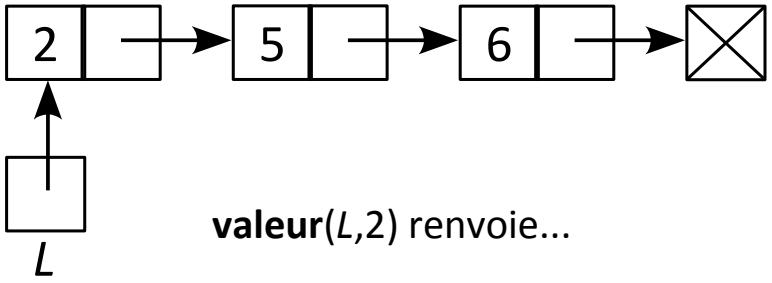
Premières fonctions sur les listes

Valeur du *i*-ième élément d'une liste

Nécessite de **parcourir la liste !**

Écrire la fonction **valeur** qui prend en entrée une liste d'entiers *L* et un entier *i*, et qui renvoie la valeur de la *i*-ième case de *L*, si elle existe, et -1 sinon.

Algorithme **valeur**
Entrées : liste d'entiers *L*, entier *i*
Type de sortie :
Variables :
Début



valeur(*L*,2) renvoie...

Fin

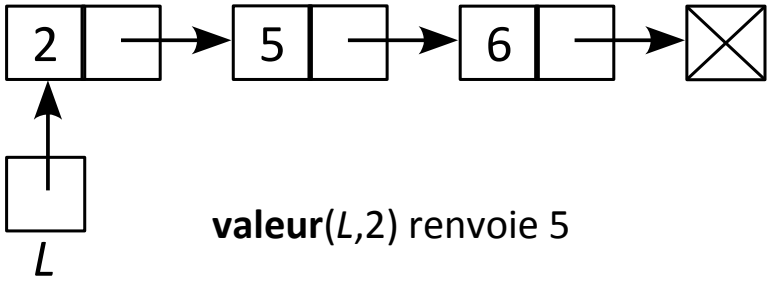
Premières fonctions sur les listes

Valeur du *i*-ième élément d'une liste

Nécessite de **parcourir la liste !**

Écrire la fonction **valeur** qui prend en entrée une liste d'entiers *L* et un entier *i*, et qui renvoie la valeur de la *i*-ième case de *L*, si elle existe, et -1 sinon.

Algorithme **valeur**
Entrées : liste d'entiers *L*, entier *i*
Type de sortie :
Variables :
Début



valeur(*L*,2) renvoie 5

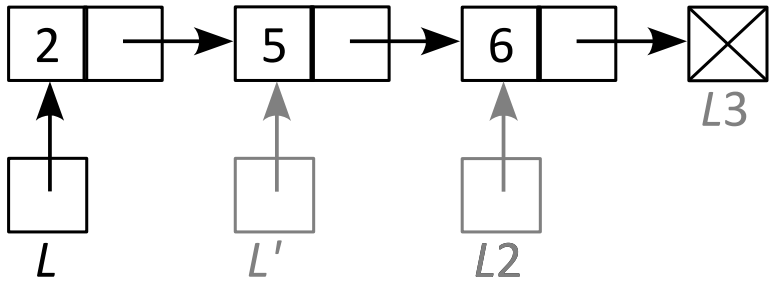
Fin

Premières fonctions sur les listes

Valeur du *i*-ième élément d'une liste

Nécessite de **parcourir la liste !**

Écrire la fonction **valeur** qui prend en entrée une liste d'entiers *L* et un entier *i*, et qui renvoie la valeur de la *i*-ième case de *L*, si elle existe, et -1 sinon.




```
Algorithme valeur
Entrées : liste d'entiers L, entier i
Type de sortie : entier
Variables : entier resultat
Début
    resultat ← -1
    Si L≠NULL alors :
        Si i=1 alors :
            resultat ← tete(L)
        Sinon :
            resultat ← valeur(suivant(L),i-1)
        Fin Si
    Fin Si
    renvoyer resultat
Fin
```

Facile pour les tableaux, “difficile” pour les listes

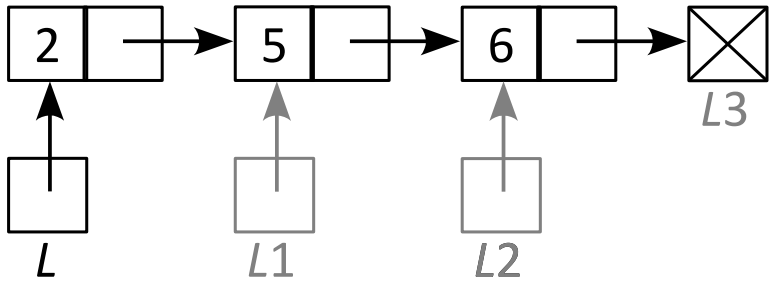
Premières fonctions sur les listes

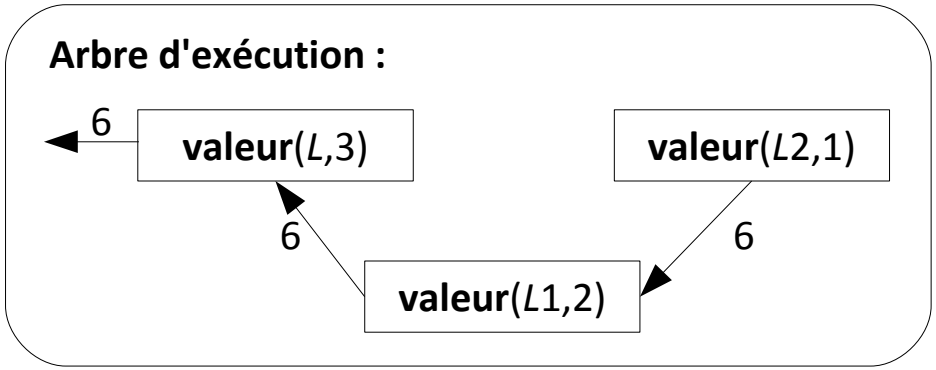
Valeur du *i*-ième élément d'une liste

Nécessite de **parcourir la liste !**

Écrire la fonction **valeur** qui prend en entrée une liste d'entiers *L* et un entier *i*, et qui renvoie la valeur de la *i*-ième case de *L*, si elle existe, et -1 sinon.

```
Algorithme valeur
Entrées : liste d'entiers L, entier i
Type de sortie : entier
Variables : entier resultat
Début
    resultat ← -1
    Si L≠NULL alors :
        Si i=1 alors :
            resultat ← tete(L)
        Sinon :
            resultat ← valeur(suivant(L),i-1)
        Fin Si
    Fin Si
    renvoyer resultat
Fin
```





Facile pour les tableaux, “difficile” pour les listes

Plan du cours 3 – Tris, structures de données

- Le tri par sélection
- Complexité des tris
- Listes
- Piles et files
- Arbres

Files et piles

Files

FIFO “First In First Out”

File d'attente à la Poste

Piles

LIFO “Last In First Out”

Pile de documents à travailler

Files et piles

Files

FIFO “First In First Out”

File d'attente à la Poste

Gestion des buffers

http://www.dessinateur.biz/blog/tag/file-dattente/
http://img.clubic.com/02989454-photo-cisco-crs3.jpg

Piles

LIFO “Last In First Out”

Pile de documents à travailler

http://fnhostile420.tumblr.com/post/1586779953/aporianonymous-i-have-a-to-do-pile
http://imageshack.us/photo/my-images/268/magicthegatheringback.jpg/

Files et piles

Files
FIFO “First In First Out”

File d'attente à la Poste

Piles
LIFO “Last In First Out”

Pile de documents à travailler

Pourquoi utiliser ces structures de données ?
- Plus simple à programmer qu'une liste
- Fonctions plus limitées, adaptées aux besoins

Gestion des buffers

http://www.dessinateur.biz/blog/tag/file-dattente/
http://img.clubic.com/02989454-photo-cisco-crs3.jpg

http://fnhostile420.tumblr.com/post/1586779953/aporianonymous-i-have-a-to-do-pile
http://imageshack.us/photo/my-images/268/magicthegatheringback.jpg/

Files et piles

Files
FIFO “First In First Out”

File d'attente à la Poste

Quatre fonctions :
- **créer** : créer une file vide
- **enfiler** : ajouter un élément en fin de file
- **défiler** : lire la valeur de l'élément au début de la file en l'enlevant de la file
- **tester si vide** : savoir si la file est vide

Gestion des buffers

http://www.dessinateur.biz/blog/tag/file-dattente/
http://img.clubic.com/02989454-photo-cisco-crs3.jpg

Piles
LIFO “Last In First Out”

Pile de documents à travailler

Quatre fonctions :
- **créer** : créer une pile vide
- **empiler** : ajouter un élément en haut de la pile
- **dépiler** : lire la valeur de l'élément en haut de la pile en l'enlevant de la pile
- **tester si vide** : savoir si la pile est vide

http://fnhostile420.tumblr.com/post/1586779953/aporianonymous-i-have-a-to-do-pile
http://imageshack.us/photo/my-images/268/magicthegatheringback.jpg/

Files et piles

implémentable avec liste ***doublement*** chaînée

Files

FIFO “First In First Out”

File d'attente à la Poste

Quatre fonctions :
- **créer** : créer une file vide
- **enfiler** : ajouter un élément en fin de file
- **défiler** : lire la valeur de l'élément au début de la file en l'enlevant de la file
- **tester si vide** : savoir si la file est vide

Gestion des buffers

http://www.dessinateur.biz/blog/tag/file-dattente/
http://img.clubic.com/02989454-photo-cisco-crs3.jpg

implémentable avec liste simplement chaînée

Piles

LIFO “Last In First Out”

Pile de documents à travailler

Quatre fonctions :
- **créer** : créer une pile vide
- **empiler** : ajouter un élément en haut de la pile
- **dépiler** : lire la valeur de l'élément en haut de la pile en l'enlevant de la pile
- **tester si vide** : savoir si la pile est vide

http://fnhostile420.tumblr.com/post/1586779953/aporianonymous-i-have-a-to-do-pile
http://imageshack.us/photo/my-images/268/magicthegatheringback.jpg/

Plan du cours 3 – Tris, structures de données

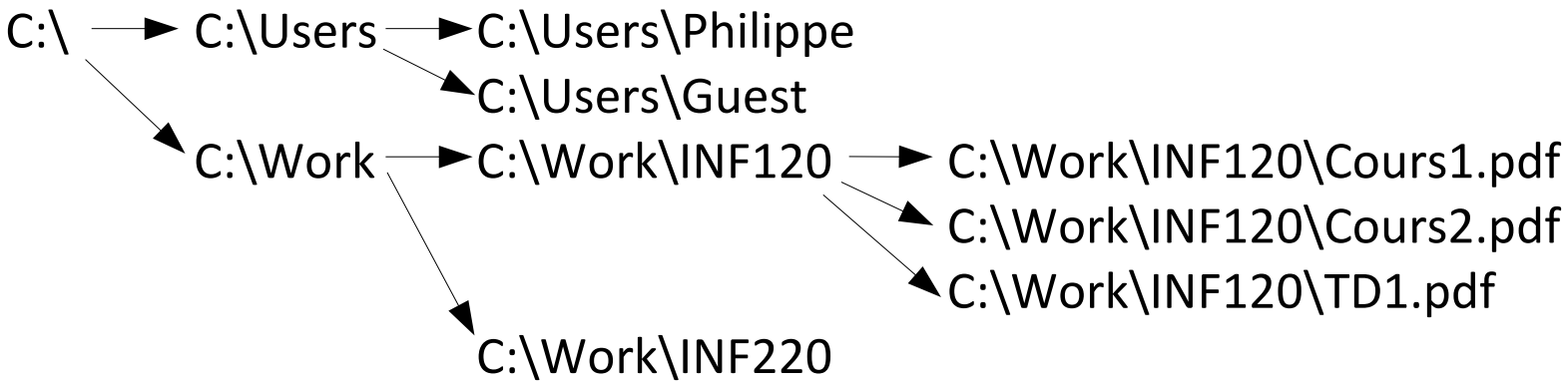
- Le tri par sélection
- Complexité des tris
- Listes
- Piles et files
- Arbres

Intérêt des arbres

Intérêt des arbres :

• représentation d'un objet qui a une structure arborée :

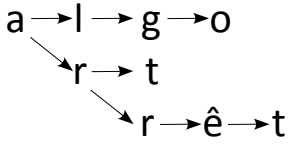
→ contenu d'un disque dur :



• stockage astucieux d'éléments

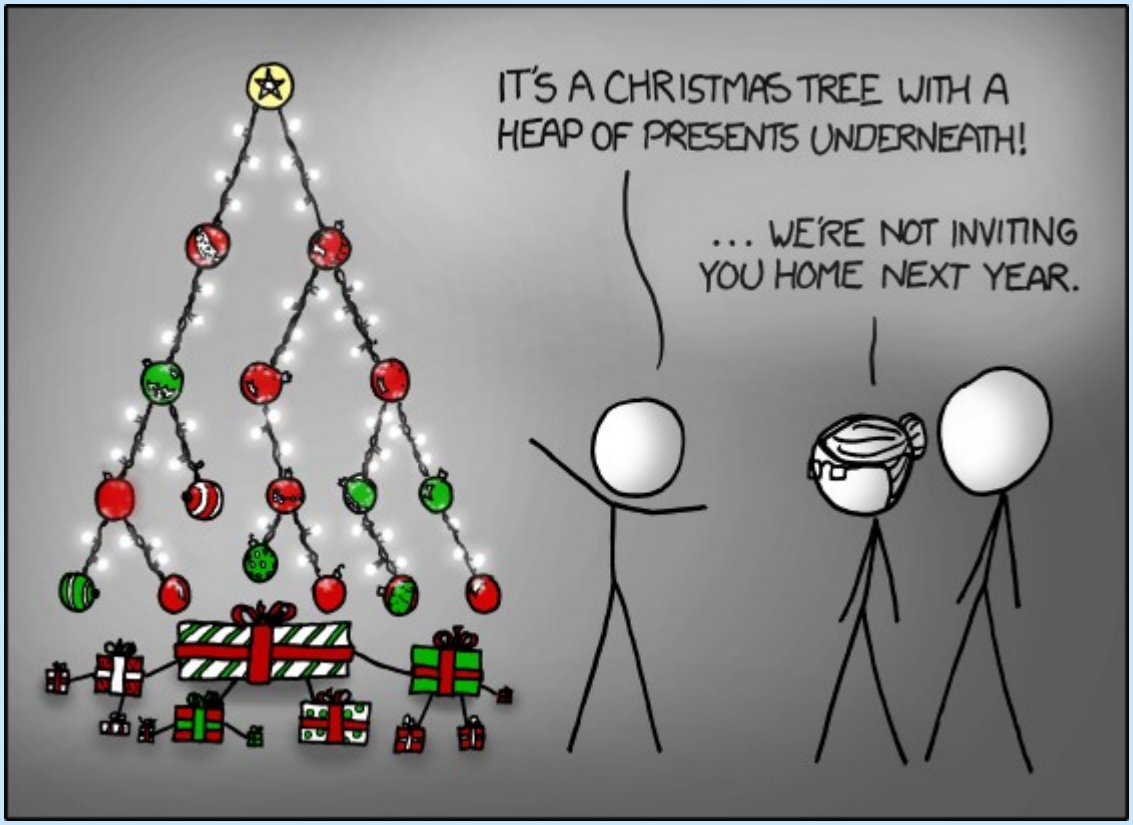
→ trouver rapidement le minimum des valeurs stockées

→ stocker un dictionnaire de manière compacte :



Arbre

La “minute xkcd” - Arbre



http://xkcd.com/835
http://xkcd.free.fr/?id=835

- Voici un **arbre** de noël avec un **tas** de cadeaux en dessous !

- … L'année prochaine, on ne t'invitera pas à la maison.

Définition récursive d'un arbre

Un **arbre** :

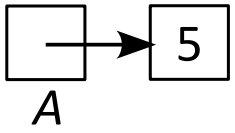
- soit une **feuille**, qui contient une valeur, mais n'a pas d'enfant

- soit un **noeud** qui contient une valeur, et un tableau d'**enfants** dont chaque case pointe vers un **arbre**

Définition illustrée d'un arbre

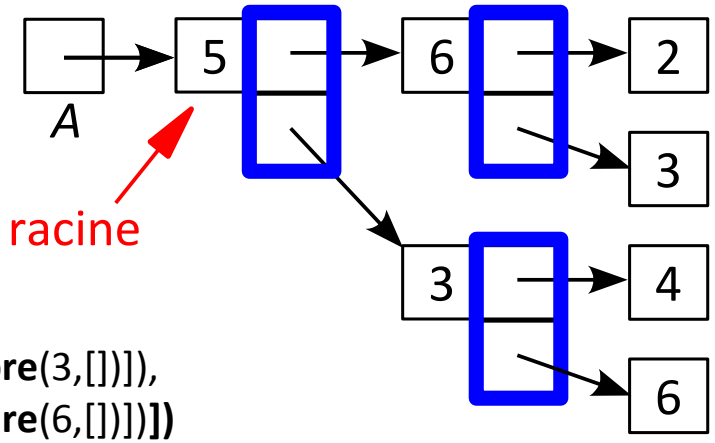
Un **arbre** :

- soit une **feuille**, qui contient une valeur, mais n'a pas d'enfant



$A \leftarrow$ **CreeArbre**(5,[])

- soit un **noeud** qui contient une valeur, et un tableau d'**enfants** dont chaque case pointe vers un **arbre**

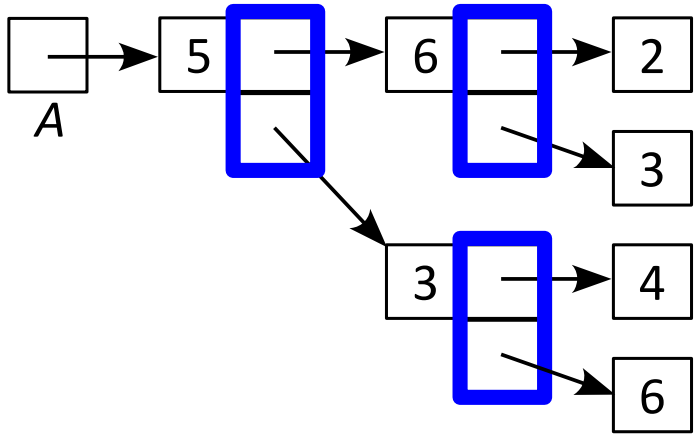


$A \leftarrow$ **CreeArbre(**5,**[CreeArbre**(6,[**CreeArbre**(2,[]),**CreeArbre**(3,[])]),
CreeArbre(3,[**CreeArbre**(4,[]),**CreeArbre**(6,[])])**])**

Si chaque tableau d'enfants a au plus deux cases, c'est un **arbre binaire**.

Fonctions de base sur les arbres

Tester si l'arbre est une feuille :
estFeuille(*A*)

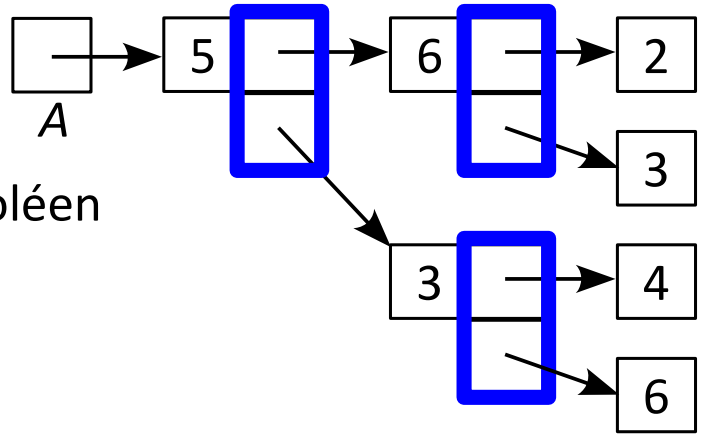


Récupérer la valeur de la racine de l'arbre
racine(*A*)

Récupérer le tableau des enfants de la racine de l'arbre
enfants(*A*)

Fonctions de base sur les arbres

Tester si l'arbre est une feuille :
estFeuille(*A*)
Type de sortie pour un arbre d'entiers : un booléen



Récupérer la valeur de la racine de l'arbre
racine(*A*)
Type de sortie pour un arbre d'entiers : un entier

Récupérer le tableau des enfants de la racine de l'arbre
enfants(*A*)
Type de sortie pour un arbre d'entiers : un tableau d'arbres d'entiers