# Structures de données listes, piles, files, arbres binaires

Jacques Le Maitre

UNIVERSITÉ du SUD
Toulon-Var

# Bibliographie


- Robert Sedgewick, *Algorithmes en langage C*, InterEditions.
- Ouvrage collectif (coordination Luc Albert), *Cours et exercices d'informatique*, Vuibert.
- Donald E. Knuth, *The Art of Computer Programming*, Volume 1, 2 et 3, Addison-Wesley.

# Définition et implantation d'un type de données

1. Définir formellement les données de ce type.
2. Définir l'**interface** du type de données :
   - un jeu d'opérations pour manipuler ces données en commençant par les opérations **primitives** : celles à partir desquelles les autres peuvent être définies.
3. Choisir une **représentation** de ces données.
4. Définir les opérations relativement à cette représentation.
5. Choisir un langage de programmation.
6. Définir, en termes de ce langage, la représentation des données et les opérations.
7. Regrouper ces définitions dans un module constituant l'**implantation** du type de données.

# Structures de données étudiées

- Les **listes linéaires** et deux spécialisations importantes de ces listes :
  - les **piles**
  - les **files**
- Les **arbres binaires.**

# Implantation des structures de données

- Deux modes d'implantation seront étudiés :
  - structures « **modifiables en place** », pour les listes linéaires,
  - structures non modifiables évoluant par **reconstruction**, pour les arbres binaires.
- Le premier mode implique une programmation par effets de bord c.-à-d. où les changements d'état de la mémoire sont explicites.
- Le second mode est parfaitement adapté à la programmation fonctionnelle.
- Le premier mode est plus efficace en temps et en mémoire.
- Le second mode privilégie une programmation déclarative.
- Le langage de programmation choisi est le langage C.

# LISTES LINÉAIRES

# Définition formelle

- Soit $T$ un type de données.
- Une **liste linéaire** de valeurs de type $T$ est une suite de $n$ ($n \geq 0$) valeurs $v_1$, ..., $v_n$ rangées de façon à ce que :
  - si $n > 0$, $v_1$ est la première valeur dans ce rangement et $v_n$ est la dernière,
  - si $1 < k < n$, $v_k$ est précédée par la valeur $v_{k-1}$ et suivie par la valeur $v_{k+1}$.

  (Donald E. Knuth, *The Art of Computer Programming*, Volume 1, *Fundamental Algorithms*, Third Edition, p. 238)
- Notation :
  - [] liste linéaire vide
  - [$v_1$, ..., $v_n$] liste linéaire non vide

# Opérations primitives sur les listes linéaires (1)

- On note :
  - *Vide* le type dont l'unique valeur est la valeur rien notée ()
  - *T* : un type de données,
  - *Liste*(*T*) le type des listes linéaires de valeurs de type *T*,
  - *Position* le type des positions d'une valeur dans une liste linéaire,
  - *Booléen* le type d'extension {vrai, faux},
  - *positions*(*l*) l'ensemble des positions des valeurs de la liste linéaire *l*,
  - *pos(v)* la position de la valeur *v* dans la liste considérée,
  - ⇝ la modification en place d'une liste.

# Opérations primitives sur les listes linéaires (2)

- Création d'une liste vide
  - *créer-liste* : Vide → *Liste*(*T*)
  - *créer-liste* : () ↦ []
- Test de liste vide
  - *liste-vide* : *Liste*(*T*) → *Booléen*
  - *liste-vide* : [] ↦ vrai
  - *liste-vide* : [...*v*...] ↦ faux

# Opérations primitives sur les listes linéaires (3)

- Insertion d'une valeur en début de liste
  - $insérer\_au\_début : Liste(T) \times T \longrightarrow Vide$
  - $insérer\_au\_début : ([...], v) \mapsto ()$

    effet de bord : $[...] \rightsquigarrow [v...]$
- Insertion d'une valeur après une position donnée
  - $insérer\_après : Liste(T) \times Position \times T \longrightarrow Liste(T)$
  - $insérer\_après : (l, p, v) \mapsto$ erreur ! (si $p \notin positions(l)$)
  - $insérer\_après : ([...w...], p, v)$ où $p = pos(w) \mapsto ()$

    effet de bord : $[...w...] \rightsquigarrow [...w, v...]$

# Opérations primitives sur les listes linéaires (4)

- Suppression d'une valeur dont la position est donnée
  - *supprimer* : $Liste(T) \times Position \rightarrow Vide$
  - *supprimer* : $(l, p) \mapsto$ erreur ! (si $p \notin positions(l)$)
  - *supprimer* : ([...$v$...], $p$) où $pos(v) = p \mapsto ()$

    effet de bord : [...$v$...] $\rightsquigarrow$ [...]

# Opérations primitives sur les listes linéaires (5)

- Sélection de la valeur dont la position est donnée
  - $valeur : Liste(T) \times Position \rightarrow T$
  - $valeur : (l, p) \mapsto$ erreur ! (si $p \notin positions(l)$)
  - $valeur : ([...v...], p)$ où $pos(v) = p \mapsto v$
- Remplacement de la valeur dont la position est donnée
  - $remplacer : Liste(T) \times Position \times T \rightarrow Liste(T)$
  - $remplacer : (l, p, w) \mapsto$ erreur ! (si $p \notin positions(l)$)
  - $remplacer : ([...v...], p, \mathrm{w})$ où $pos(v) = p \mapsto ()$

    effet de bord : $[...v...] \rightsquigarrow [...w...]$

# Types paramétrés et polymorphisme

- Le type *Liste*(*T*) est un **type paramétré**. Il englobe tous les types de listes obtenus en remplaçant le paramètre de type *T* par un type effectif : *Liste*(*Entier*), *Liste*(*Flottant*), *Liste* (*Point*), etc.
- Les opérations sur le type *Liste*(*T*) sont dites **polymorphes** car elles s'appliquent à toutes les listes quelque soit *T*.
- Par exemple, la fonction :
  - *liste_vide* : *Liste*(*T*) → *Booléen*

  s'applique aussi bien aux listes de types : *Liste*(*Entier*), *Liste*(*Flottant*), *Liste*(*Point*), etc.
- Il n'est pas nécessaire d'écrire une définition pour chacun de ces types : une seule définition, dite **générique**, est suffisante quelque soit *T*.

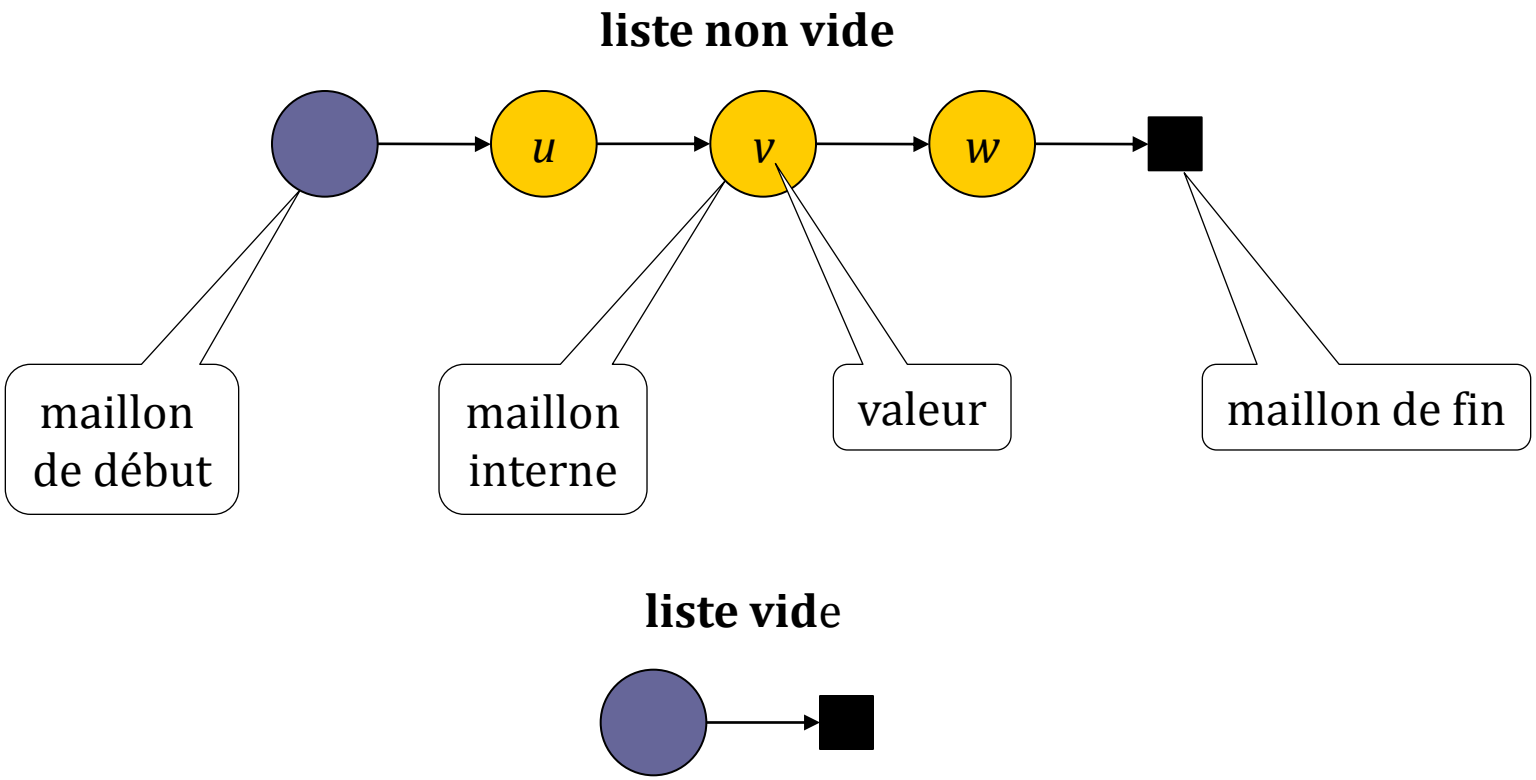
# Représentations d'une liste linéaire

- Plusieurs représentations des listes linéaires ont été proposées.
- La plupart consistent à enregistrer chaque valeur dans une cellule de la mémoire et à chaîner ces cellules entre elles.
- Elles se différencient principalement par :
  - le mode de mémorisation des cellules : dans un tableau ou bien dans une zone mémoire allouée dynamiquement,
  - le mode de de marquage du début ou de la fin de la liste,
  - le mode de chaînage des cellules : unidirectionnel ou bidirectionnel.

# Notre représentation d'une liste linéaire

- Une liste linéaire est représentée comme une chaîne de **maillons** composée :
  - d'un **maillon de début**,
  - d'une suite éventuellement vide de **maillons internes,**
  - d'un **maillon de fin**.
- Chaque maillon a un **identifiant**.
- Le maillon de début contient l'identifiant du 1$^{er}$ maillon interne.
- Le $i^e$ maillon interne contient la $i^e$ valeur de la liste linéaire et l'identifiant du maillon contenant la $(i + 1)^e$ valeur.
- Le maillon de fin a un identifiant nul.
- Le maillon suivant du maillon de début d'une **liste linéaire vide** est le maillon de fin.
- Une liste est identifiée par l'identifiant de son maillon de début.

# Notre représentation d'une liste linéaire

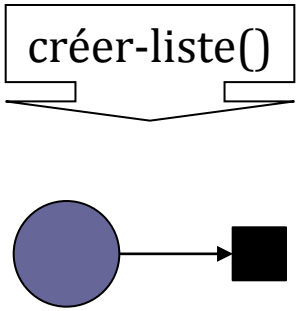


# Propriétés de cette représentation

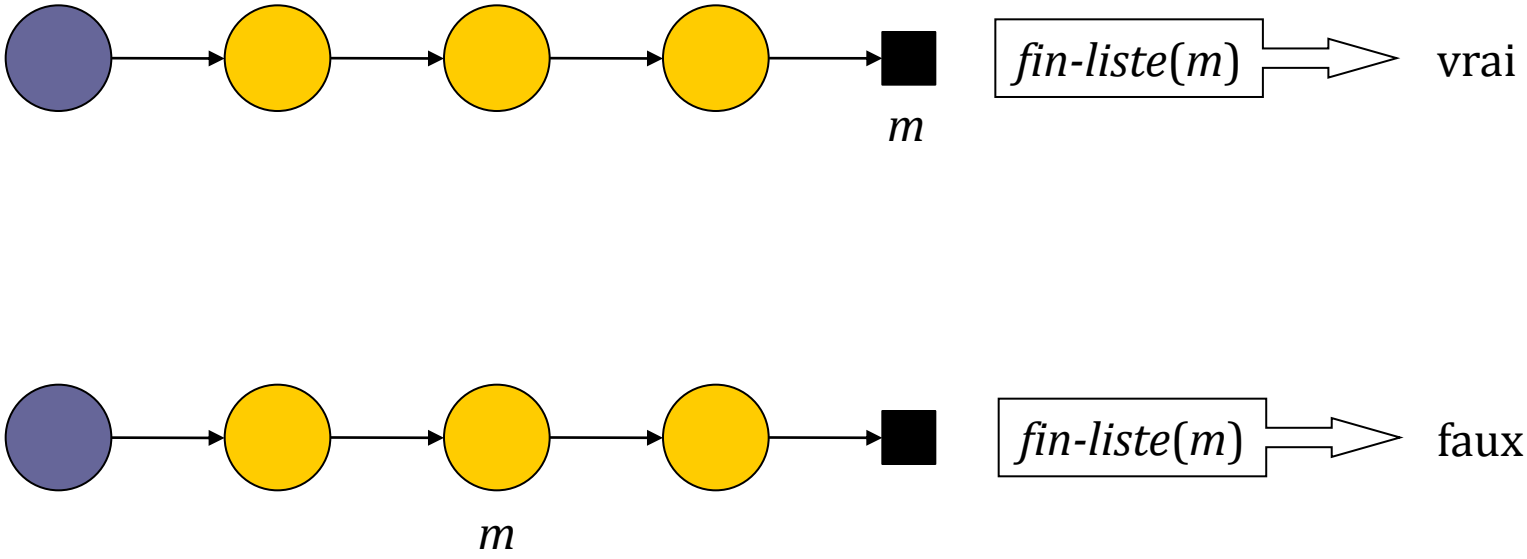
- La position d'une valeur est l'identifiant du maillon qui la contient.
- Le chaînage est unidirectionnel et dirigé de la première vers la dernière valeur :
  - la 1ère valeur est atteinte à partir du maillon de début,
  - la $k^e$ valeur $(k > 1)$ est atteinte à partir du maillon contenant la $(k - 1)^e$ valeur.
- Une opération *suivant* est nécessaire pour passer d'un maillon à l'autre.
- L'opération *supprimer* est remplacée par une opération *supprimer-suivan*t.
- L'opération *insérer-au-début* est redondante et donc non définie, car :
  - *insérer-au-début*$(l, v) \equiv$ *insérer-après*$(l, v)$

  puisque $l$ est l'identifiant du maillon de début.
- Une opération *fin-liste* est nécessaire pour tester que le maillon de fin a été atteint.

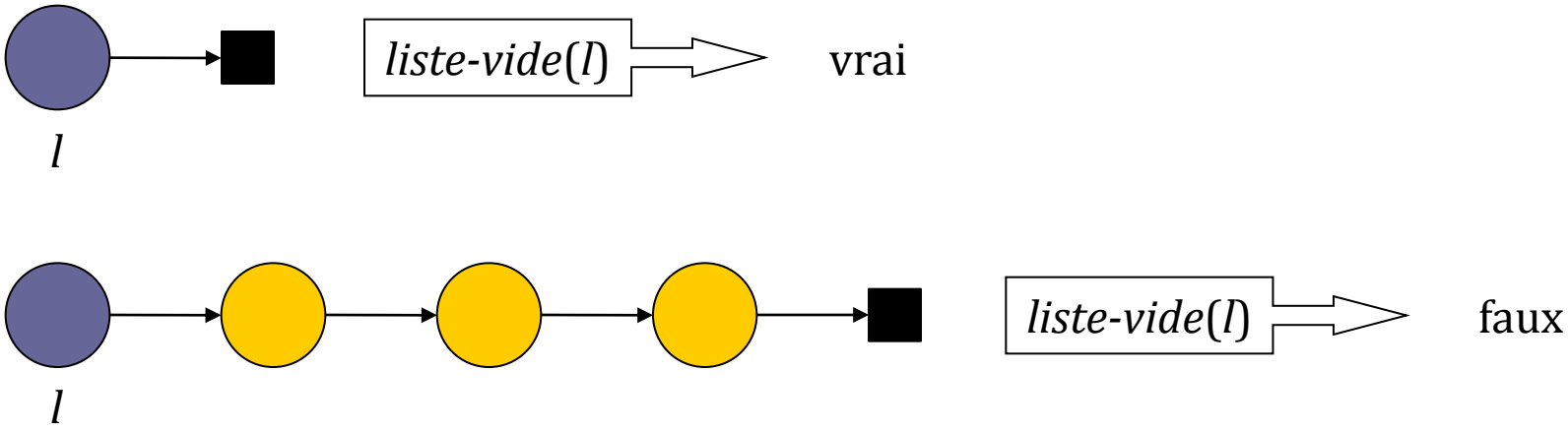
# *créer-liste*



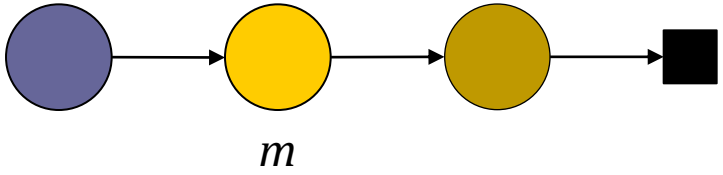
# *fin-liste*



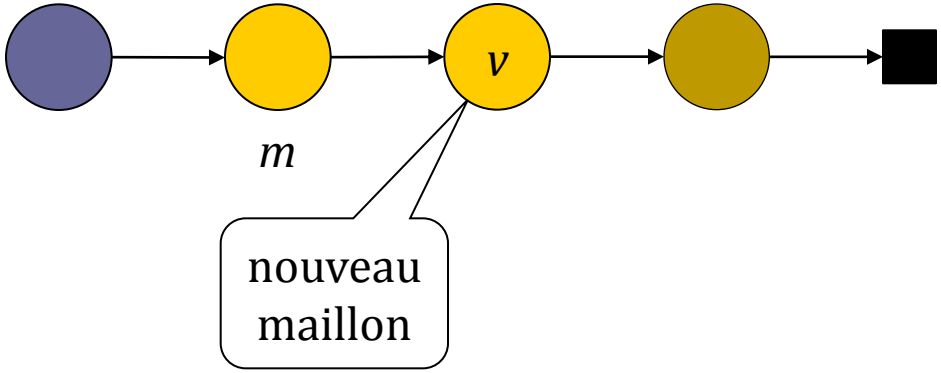
# *liste-vide*



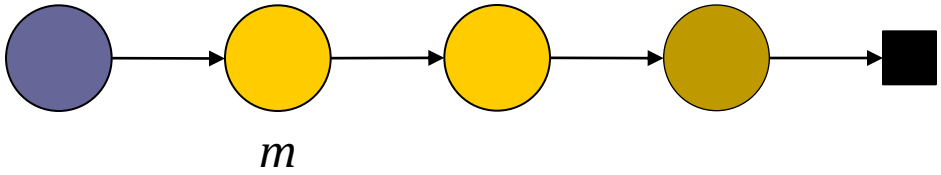
# *insérer-après*



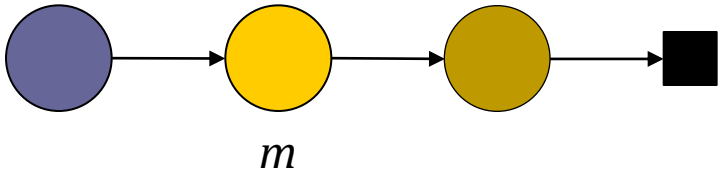

*insérer-après*$(m, v)$



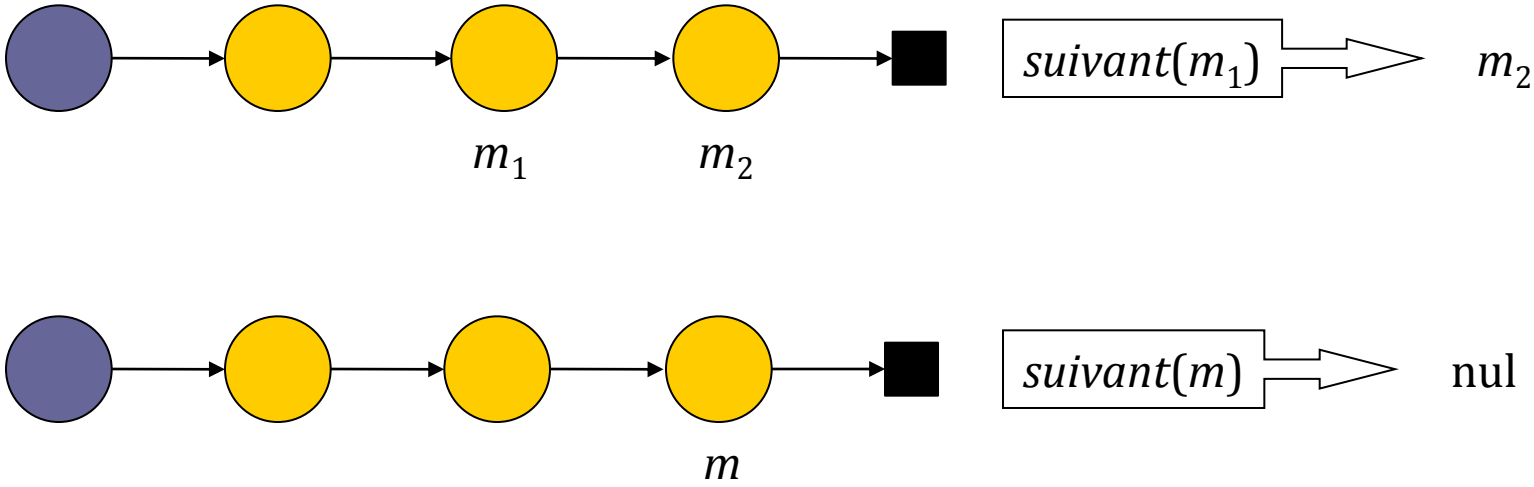
# *supprimer-suivant*



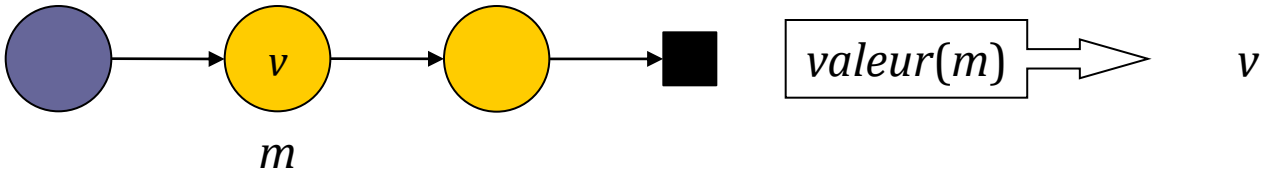

*supprimer-suivant*(*m*)



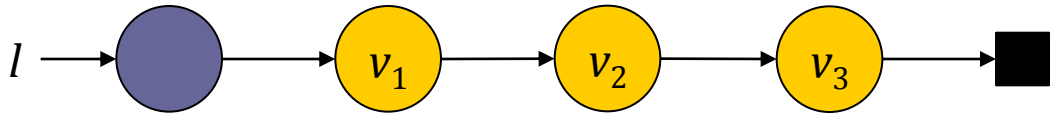
# *suivant*

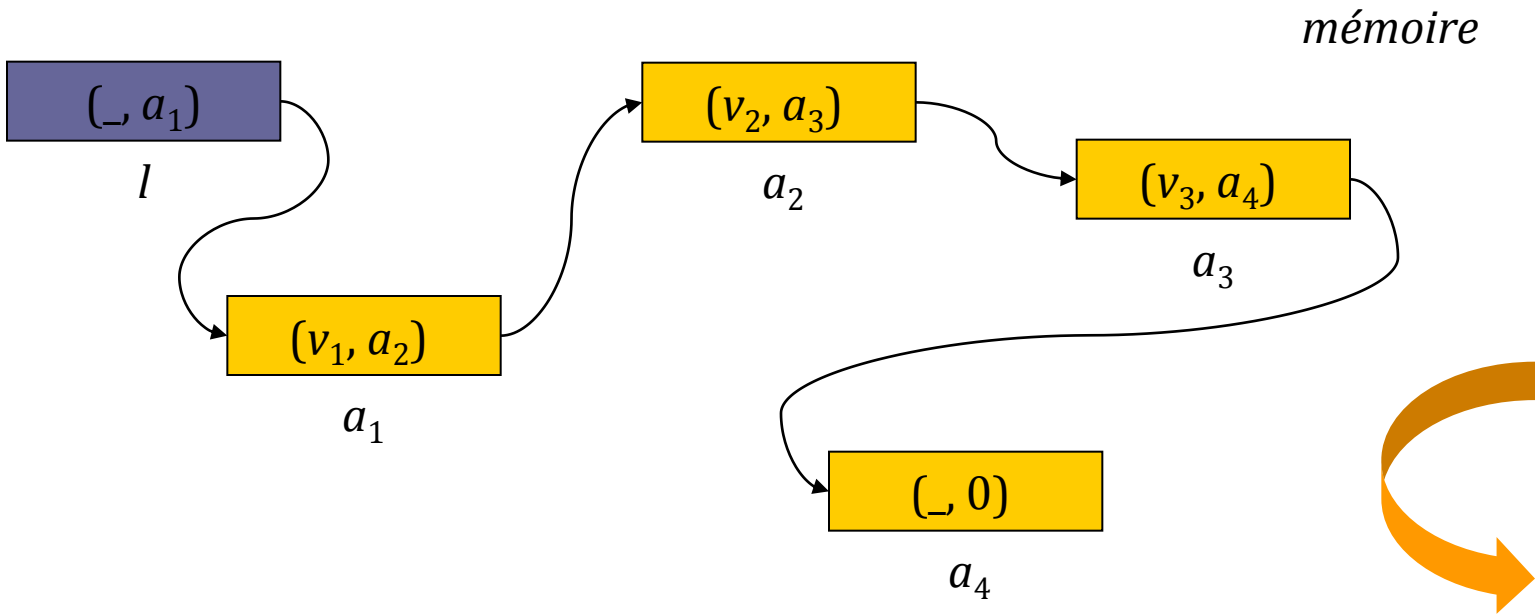


# *valeur*

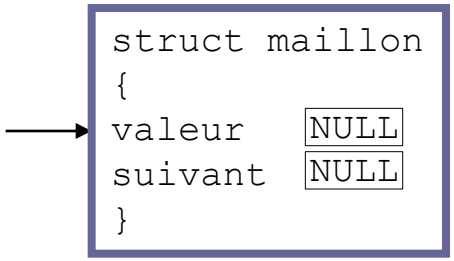


# Représentation chaînée par pointeurs



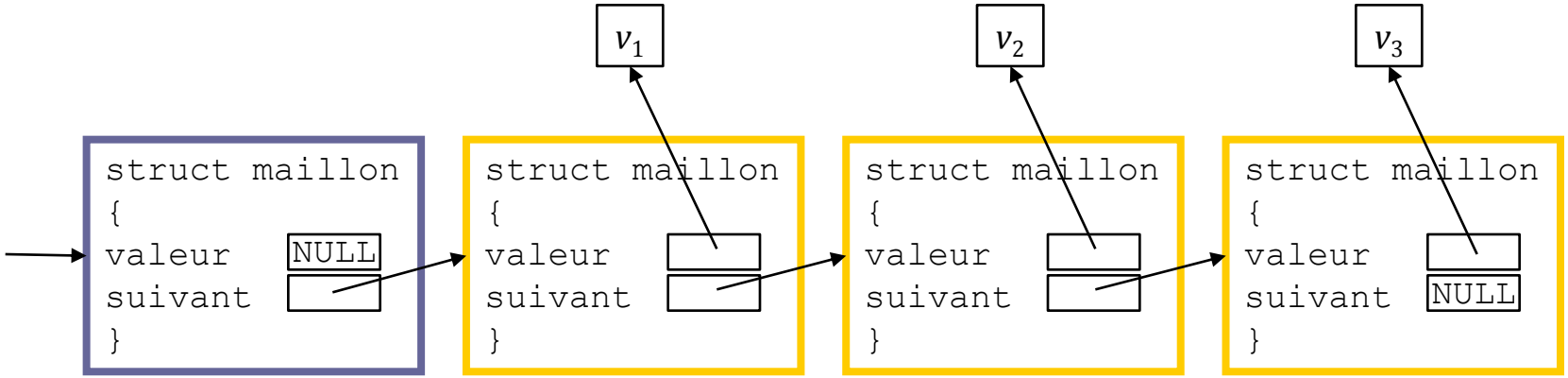



# Représentation d'une liste linéaire en C




[]

```
typedef struct maillon Maillon;
typedef Maillon *Liste;
struct maillon
   {
   void *valeur;
   Maillon *suivant;
   }
```




$[v_1, v_2, v_3]$

# Interface du type Liste linéaire en C

- Les déclarations des types `Maillon` et `Liste` ainsi que les déclarations des fonctions de l'interface sont enregistrées dans le fichier `TD-liste-lineaire.h`.
- Les définitions des fonctions de l'interface sont enregistrées dans le fichier `TD-liste-lineaire.c`.

# Polymorphisme

- Les fonctions de cet interface sont polymorphes, c.-à-d. qu'elles s'appliquent à toute liste de valeurs quelque soit le type *T* de ces valeurs.
- Cela est possible parce que :
  - ce ne sont pas directement les valeurs qui sont contenues dans les maillons, mais les adresses des variables contenant ces valeurs,
  - le champ `valeur` d'un maillon est de type `void *`, on peut donc lui affecter n'importe quel type d'adresse par définition du type `void *` en C, et donc l'adresse d'une variable de type *T*.
- Pour accéder à la valeur de la variable dont l'adresse est contenue dans un maillon, il faudra au préalable convertir cette adresse en une adresse de variable de type *T*.

# Exemple

- Supposons que l'on veuille créer et manipuler une liste de personnes décrites par leur nom et leur âge.
- Supposons que la description d'une personne soit une instance du type `Personne` défini par :
  - ```
    typedef struct
        {
        char *nom,
        int age;
        } Personne
    ```
- La description d'une personne à insérer dans une liste devra être affectée à une variable de type `Personne` (allouée par un appel à `malloc`, par exemple) :
  - C'est l'adresse de cette variable qui sera affectée dans le maillon correspondant à la place de cette personne dans cette liste.
- Pour obtenir la description de la personne dont l'adresse est contenue dans un maillon il faudra au préalable convertir cette adresse en une adresse de type `Personne *`.
    ```

# Fichier TD-liste-lineaire.h (1)

```
#ifndef TD_LISTE_LINEAIRE
#define TD_LISTE_LINEAIRE

#include <stdlib.h>
#include <stddef.h>
#include "erreur.h"

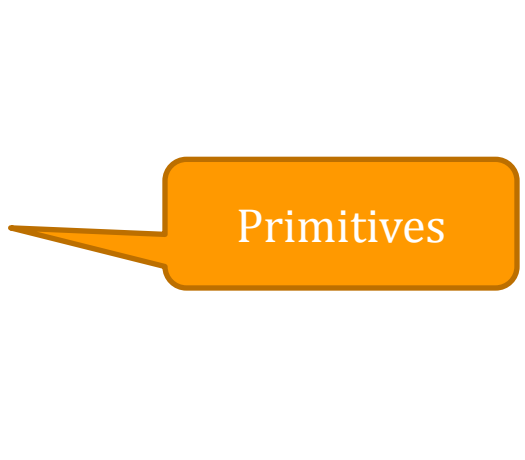
typedef struct maillon Maillon;
typedef Maillon *Liste;

struct maillon
   {
   void *valeur;
   Maillon *suivant;
   };
```

# Fichier TD-liste-lineaire.h (2)

```
Liste creer_liste(void);
int fin_liste(Maillon *m);
int liste_vide(Liste l);
Maillon *inserer_apres(Maillon *m, void *v);
void supprimer_suivant(Maillon *m);
Maillon *suivant(Maillon *m);
void *valeur(Maillon *m);

#endif
```



# Créer une liste linéaire

```
Liste creer_liste(void)
   {
   Maillon *d, *f;
   d = (Maillon *) malloc(sizeof(Maillon));
   if (d == NULL)
      erreur("creer_liste");
   d->suivant = NULL;
   return d;
   }
```

# Le maillon `m` est-il le maillon de fin de liste ?

```
int fin_liste(Maillon *m)
    {
    return m == NULL;
    }
```

# La liste `l` est-elle vide ?

```
int liste_vide(Liste l)
   {
   return l->suivant == NULL;
   }
```

# Insérer un maillon contenant la valeur v après le maillon m

```
Maillon *inserer_apres(Maillon *m, void *v)
   {
   Maillon *s;
   if (fin_liste(m))
      erreur("inserer_apres");
   s = (Maillon *) malloc(sizeof(Maillon));
   if (s == NULL)
      erreur("inserer_apres");
   s->valeur = v;
   s->suivant = m->suivant;
   m->suivant = s;
   return s;
   }
```

On retourne l'adresse du maillon inséré.

# Supprimer le maillon suivant le maillon `m`

```
void supprimer_suivant(Maillon *m)
   {
   if (fin_liste(m))
      erreur("supprimer_suivant");
   if (!fin_liste(m->suivant))
      m->suivant = m->suivant->suivant;
   }
```

Si le maillon suivant de *m* est la fin de la liste, on ne fait rien.

# Maillon suivant le maillon `m`

```
Maillon *suivant(Maillon *m)
   {
   if (fin_liste(m))
      erreur("suivant");
   return m->suivant;
   }
```

# Valeur contenue dans le maillon `m`

```
void *valeur(Maillon *m)
   {
   if (fin_liste(m))
      erreur("valeur");
   return m->valeur;
   }
```

# Autres opérations

- Longueur d'une liste
- $n^{e}$ valeur d'une liste
- Copie d'une liste
- Fusion de 2 listes d'entiers triées par ordre croissant

# n^ième maillon de la liste l

```
Maillon *nieme(Liste l, int n)
    {
    Maillon *m = l;
    int i = 0;
    while (!fin_liste(m) && i < n)
        {
        i++;
        m = suivant(m);
        }
    if (i != n)
        erreur("nieme");
    return m;
    }
```

Si *i* est différent de *n*, c'est que *n* est inférieur à 0 ou supérieur à la longueur de la liste : **erreur, c'est au programme appelant de vérifier que le rang demandé est valide !**

# Longueur de la liste `l`

```
int longueur_liste(Liste l)
   {
   Maillon *m;
   int i = 0;
   m = suivant(l);
   while (!fin_liste(m))
      {
      i++;
      m = suivant(m);
      }
   return i;
   }
```

# Copier la liste l1

```
Maillon *copier_liste(Liste l1)
   {
   Liste l2;
   Maillon *m1, *m2;
   l2 = creer_liste();
   m1 = suivant(l1);
   m2 = l2;
   while (!fin_liste(m1))
      {
      m2 = inserer_apres(m2, valeur(m1));
      m1 = suivant(m1);
      }
   return l2;
   }
```

# Fusion de 2 listes d'entiers l1 et l2 ordonnées (1)

```
int entier(void *v)
   {
   return *((int *) v);
   }
```

entier pointé par v

```
Maillon *fusion_listes(Liste l1, Liste l2)
   {
   Liste l3;
   Maillon *m1, *m2, *m3;
   l3 = creer_liste();
   m1 = suivant(l1);
   m2 = suivant(l2);
   m3 = l3;
   …
```

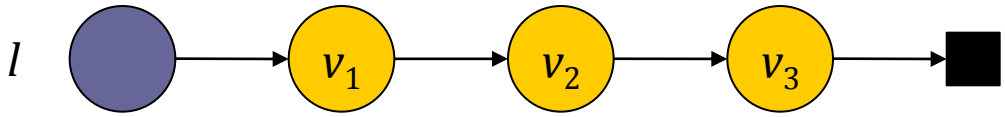
# Fusion de 2 listes d'entiers l1 et l2 ordonnées (2)

```
…
while (!fin_liste(m1) && !fin_liste(m2))
    {
    if (entier(valeur(m1)) < entier(valeur(m2)))
        {
        m3 = inserer_apres(m3, valeur(m1));
        m1 = suivant(m1);
        }
    else
        {
        m3 = inserer_apres(m3, valeur(m2));
        m2 = suivant(m2);
        }
    }
…
```

# Fusion de 2 listes d'entiers `l1` et `l2` ordonnées (3)

```
…
while (!fin_liste(m1))
    {
    m3 = inserer_apres(m3, valeur(m1));
    m1 = suivant(m1);
    }
while (!fin_liste(m2))
    {
    m3 = inserer_apres(m3, valeur(m2));
    m2 = suivant(m2);
    }
return l3;
}
```

# Allocation et libération de maillons (1)




| | | |
|---|---|---|
| | 0 | (_, *adresse case* 4) |
| | 1 | (_, ***adresse nulle***) |
| *l* | 2 | (_, *adresse case* 3) |
| | 3 | ($v_1$, *adresse case* 6) |
| | 4 | (_, *adresse case* 9) |
| | 5 | (_, *adresse case* 1) |
| | 6 | ($v_2$, *adresse case* 7) |
| | 7 | ($v_3$, ***adresse nulle***) |
| *libre* | 8 | (_, *adresse case* 0) |
| | 9 | (_, *adresse case* 5) |

*maillons*

# Allocation et libération de maillons(2)

```
#include "TD-liste-lineaire.h"

#define TAILLE 1000
static Maillon maillons[TAILLE];
static int configurer_liste_maillons = 1;
static Maillon *libre;

static Maillon *allouer_maillon(void)
static void liberer_maillon(Maillon *m)

Liste creer_liste(void)
int fin_liste(Maillon *m)
int liste_vide(Liste l)
Maillon *inserer_apres(Maillon *m, void *v)
void supprimer_suivant(Maillon *m)
Maillon *suivant(Maillon *m)
void *valeur(Maillon *m)
```

Gestion de la mémoire

Dans les fonctions `creer_liste`, `inserer_apres` et `supprimer_suivant` les fonctions `allouer_maillon` et `liberer_maillon` seront utilisées au lieu de `malloc` et `free`.

# Allocation et libération de maillons (3)

```
static Maillon *allouer_maillon(void)
   {
   int i;
   Maillon *m;
   if (configurer_liste_maillons)
      {
      for (i = 0; i < TAILLE - 1; i++)
         maillons[i].suivant = maillons + i + 1;
      maillons[i].suivant = NULL;
      libre = maillons;
      configurer_liste_maillons = 0;
      }
   if (libre == NULL)
      return NULL;
   m = libre;
   libre = m->suivant;
   return m;
   }
```

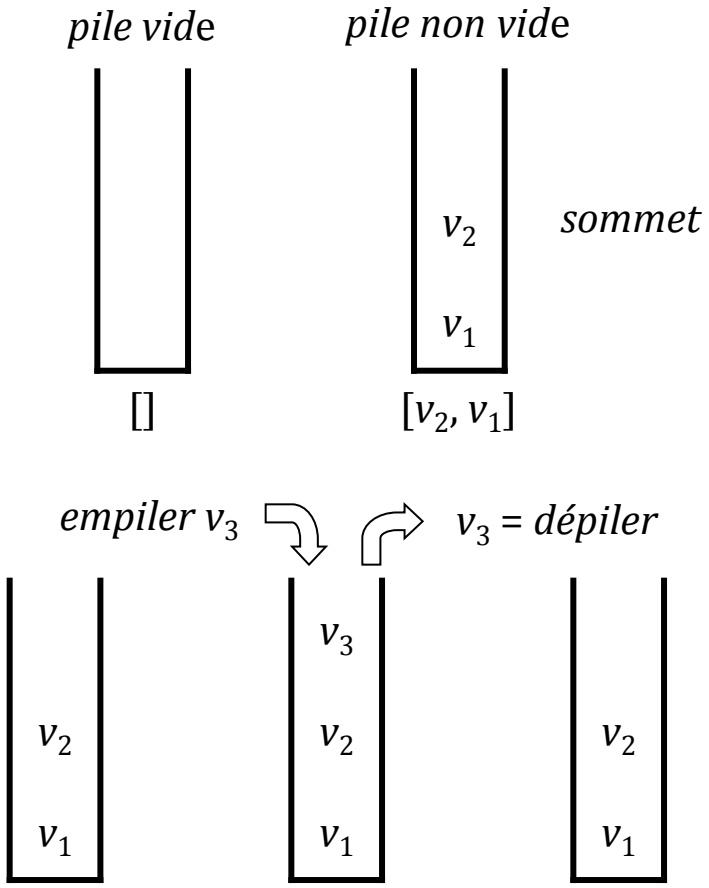
# Allocation et libération de maillons(4)

```
static void liberer_maillon(Maillon *m)
   {
   m->suivant = libre;
   libre = m;
   }
```

# PILES

# Définition formelle

- Une **pile** est une liste linéaire *l* pour laquelle les insertions et les suppressions sont effectuées au début de *l* :
  - la première valeur de *l* est le **sommet** de la pile,
  - **empiler** une valeur c'est l'insérer au début de *l*,
  - **dépiler** une valeur, si *l* est non vide, c'est sélectionner la première valeur de *l* et la supprimer de *l*.



# Opérations primitives sur les piles (1)

- On note :
  - *Pile*($T$) : le type des piles de valeurs de type $T$
- Création d'une pile
  - *créer-pile* : Vide → *Pile*($T$)
  - *créer-pile* : () ↦ []
- Test de pile vide
  - *pile-vide* : *Pile*($T$) → *Booléen*
  - *pile-vide* : [] ↦ vrai
  - *pile-vide* : [$v$...] ↦ faux

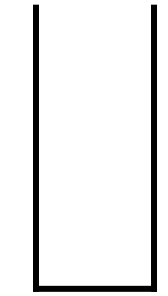
# Opérations primitives sur les piles (2)

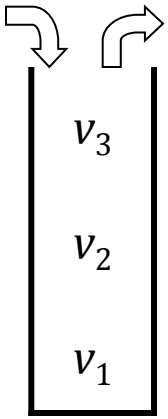
- Empiler une valeur
  - *empiler* : $Pile(T) \times T \rightarrow Vide$
  - *empiler* : ([...], $v$) $\mapsto$ ()

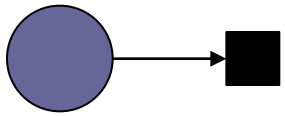
    effet de bord : [...] $\rightsquigarrow$ [$v$...]
- Dépiler une valeur
  - *dépiler* : $Pile(T) \rightarrow T$
  - *dépiler* : [] $\mapsto$ erreur !
  - *depiler* : [$v$...] $\mapsto$ $v$

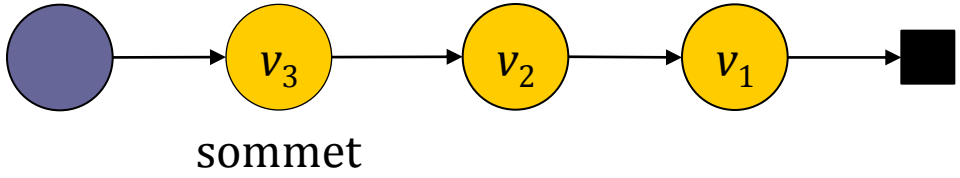
    effet de bord : [$v$...] $\rightsquigarrow$ [...]
- Sélectionner la valeur située au sommet
  - *sommet* : $Pile(T) \rightarrow T$
  - *sommet* : [] $\mapsto$ erreur !
  - *sommet* : [$v$, ...] $\mapsto$ $v$

# Représentation d'une pile

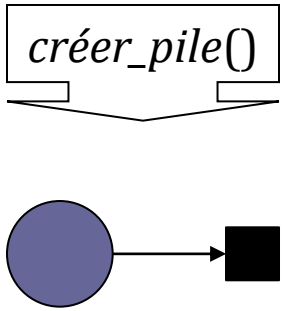




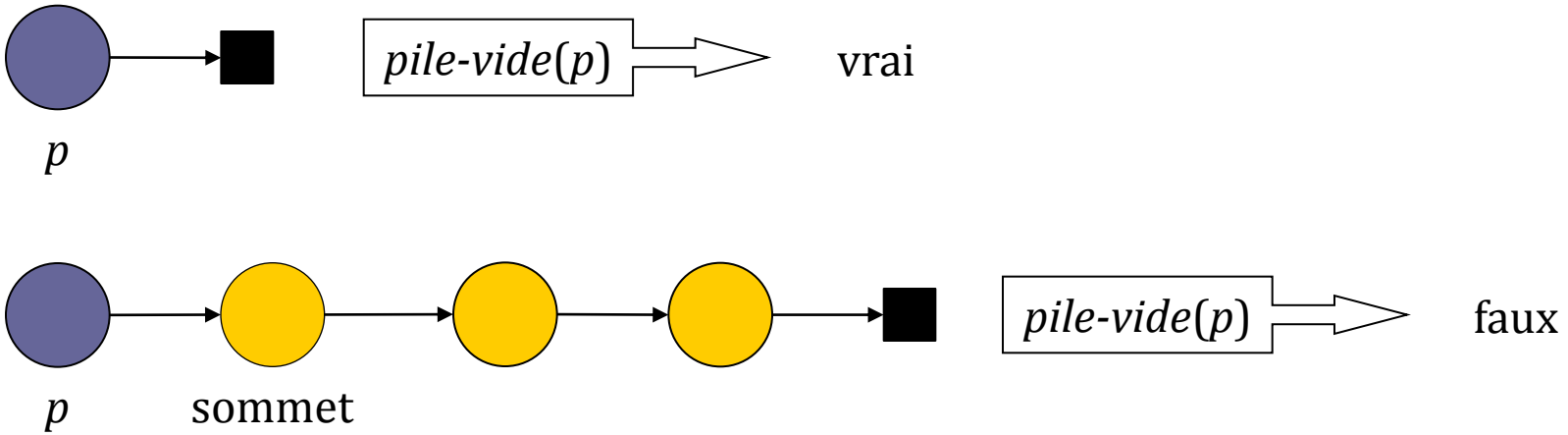





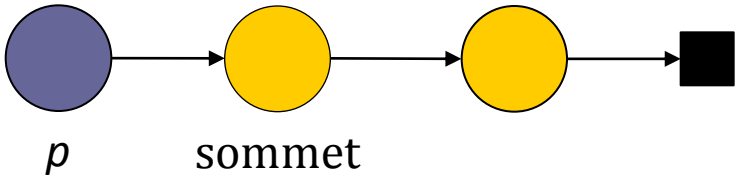
# *créer-pile*

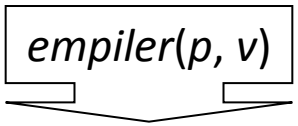


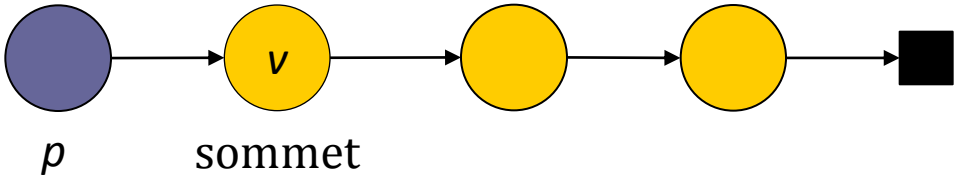
# *pile-vide*



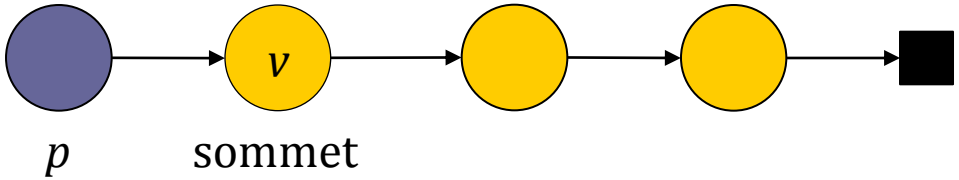
# *empiler*



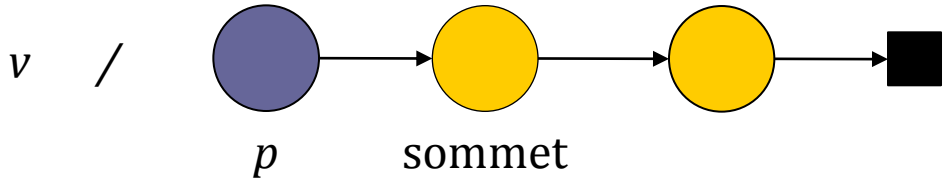



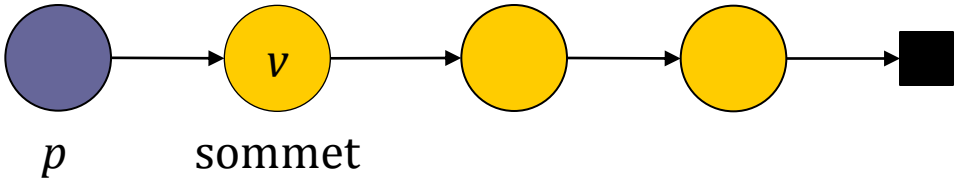
# *dépiler*




*dépiler*(*p*)



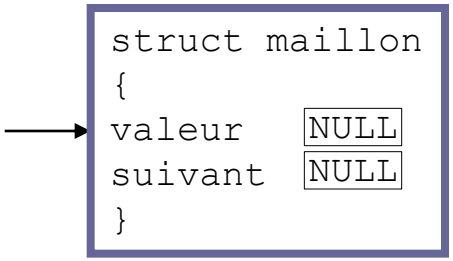
# *sommet*





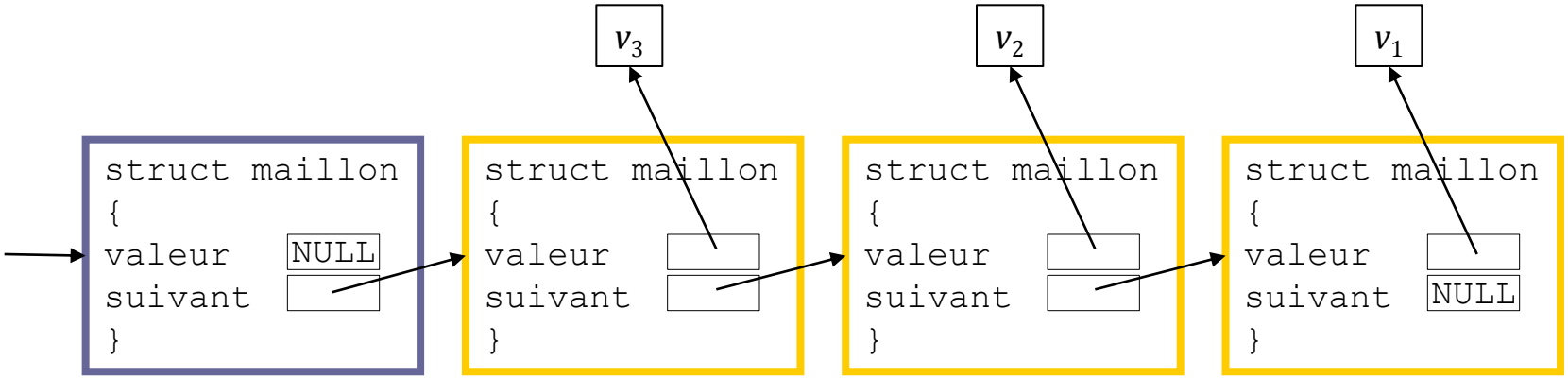
$v$

# Représentation d'une pile en C




```
typedef Liste Pile;
```

$\Leftrightarrow []$




$\Leftrightarrow [v_3, v_2, v_1]$

# Interface

- La déclaration du type `Pile` ainsi que les déclarations des fonctions de l'interface sont enregistrés dans le fichier `TD-pile.h`.
- Les définitions des fonctions de l'interface sont enregistrées dans le fichier `TD-pile.c`.

# Fichier TD-pile.h

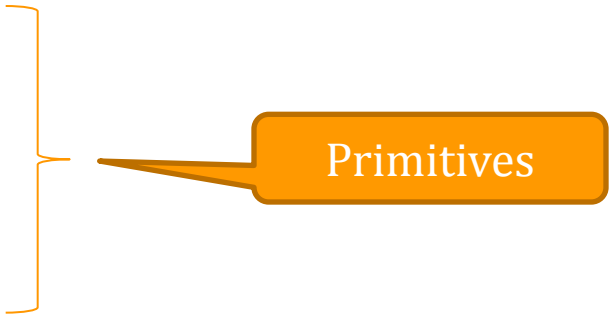
```
#ifndef TD_PILE
#define TD_PILE

#include "TD-liste-lineaire.h"
#include "erreur.h"

typedef Liste Pile;

Pile creer_pile(void);
int pile_vide(Pile p);
void empiler(Pile p, void *v);
void *depiler(Pile p);
void *sommet(Pile p);

#endif
```



# Créer une pile

```
Pile creer_pile(void)
   {
   return creer_liste();
   }
```

# La pile p est-elle vide ?

```
int pile_vide(Pile p)
    {
    return liste_vide(p);
    }
```

# Empiler une valeur v au sommet d'une pile p

```
void empiler(Pile p, void *v)
    {
    inserer_apres(p, v);
    }
```

# Dépiler la valeur au sommet de la pile p

```
void *depiler(Pile p)
   {
   void *v;
   if (pile_vide(p))
      erreur("depiler");
   v = valeur(suivant(p));
   supprimer_suivant(p);
   return v;
   }
```

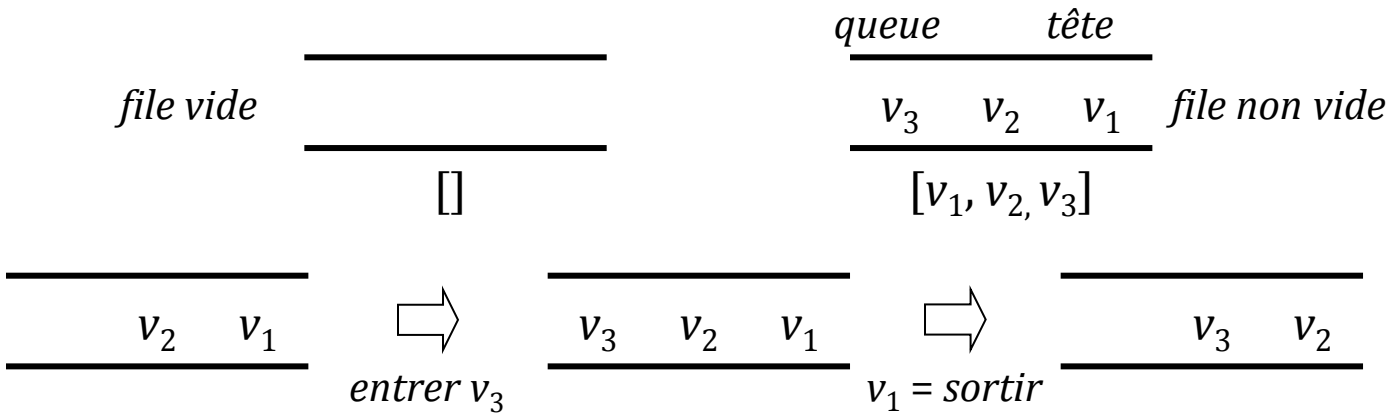
# Valeur au sommet de la pile p

```
void *sommet(Pile p)
   {
   if (pile_vide(p))
      erreur("sommet");
   return valeur(suivant(p));
   }
```

# FILES

# Définition formelle

- Une **file** est une liste linéaire *l* pour laquelle les insertions sont réalisées à la fin de *l* et les suppressions sont effectuées au début de *l* :
  - la première valeur de *l* est la **tête** de la file et la dernière est la **queue** de la file,
  - **entrer** une valeur c'est l'ajouter à la fin de de *l*,
  - **sortir** une valeur, si *l* est non vide, c'est sélectionner la première valeur de *l* et la supprimer de *l*.



# Opérations primitives sur les files (1)

- On note :
  - $File(T)$ : le type des files de valeurs de type $T$
- Création d'une file vide
  - *créer-file* : $Vide \rightarrow File(T)$
  - *créer-file* : () ↦ []
- Test de file vide
  - *file-vide* : $File(T) \rightarrow Booléen$
  - *file-vide* : [] ↦ vrai
  - *file-vide* : [*v*...] ↦ faux

# Opérations primitives sur les files (2)

- Entrer une valeur dans une file
  - $entrer : File(T) \times T \rightarrow Vide$
  - $entrer : ([...], v) \mapsto ()$

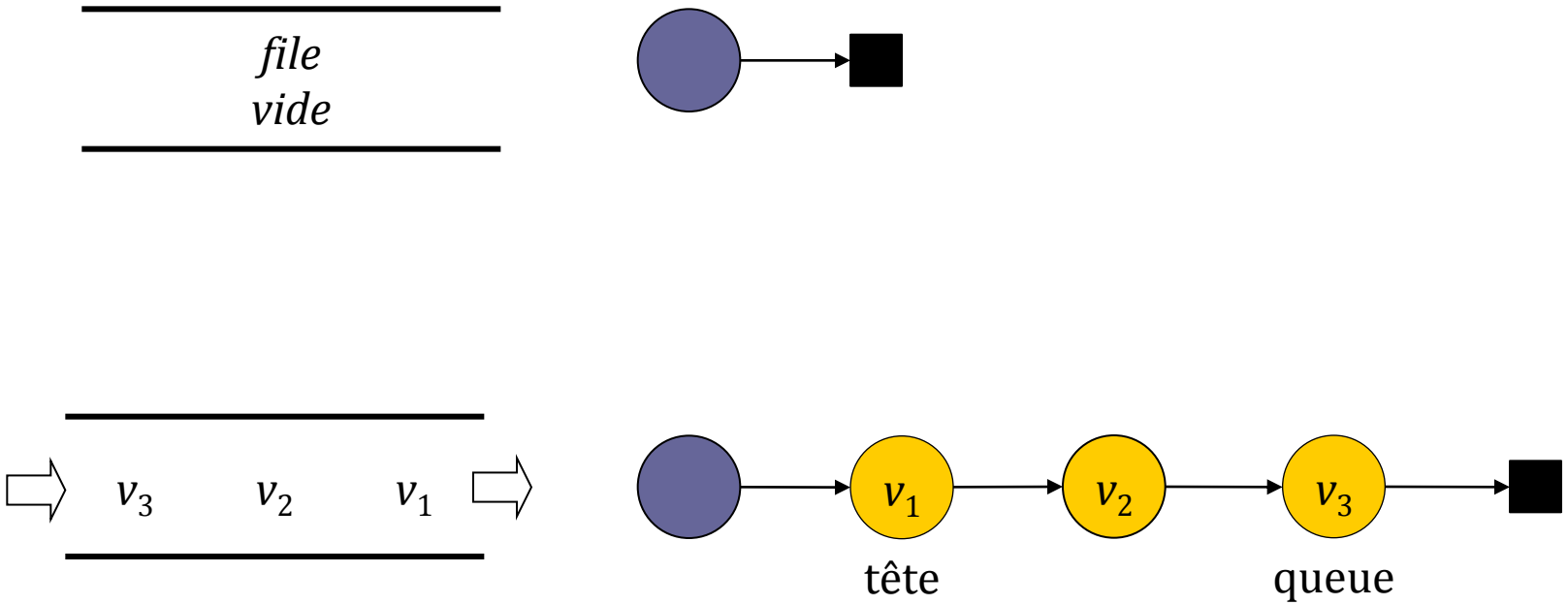
    <u>effet de bord</u> : $[...] \rightsquigarrow [...v]$
- Sortir une valeur d'une file
  - $sortir : File(T) \rightarrow T$
  - $sortir : [] \mapsto$ erreur !
  - $sortir : [v...] \mapsto v$

    <u>effet de bord</u> : $[v...] \rightsquigarrow [...]$

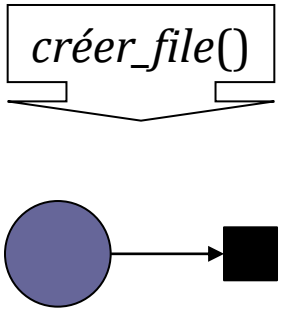
# Opérations primitives sur les files (3)

- Sélectionner la valeur en tête d'une file
  - $tête : File(T) \rightarrow T$
  - $tête : [v...] \mapsto v$
  - $tête : [] \mapsto$ erreur !
- Sélectionner la valeur en queue d'une file
  - $queue : File(T) \rightarrow T$
  - $queue : [...v] \mapsto v$
  - $queue : [] \mapsto$ erreur !

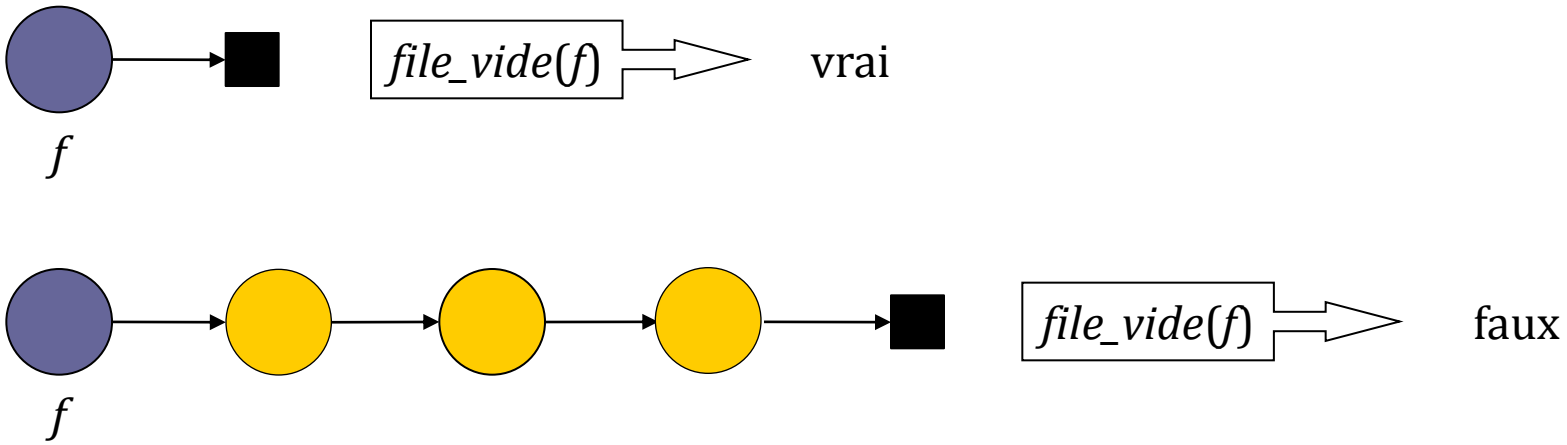
# Représentation d'une file



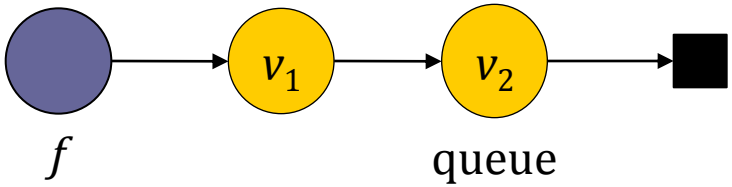
# *créer-file*



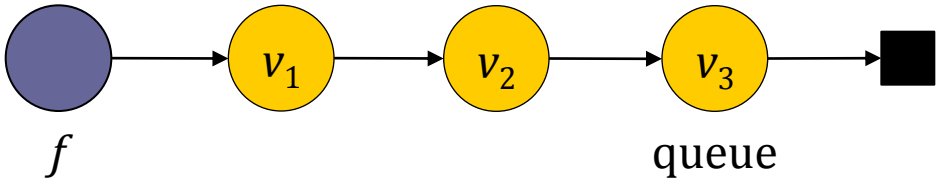
# *file-vide*



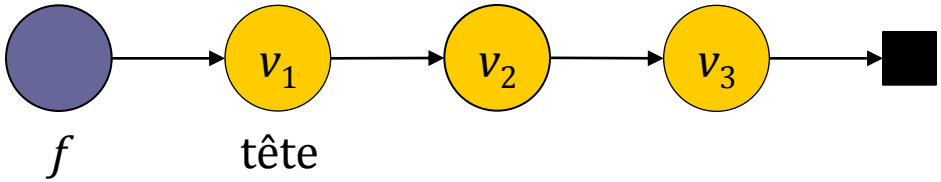
# *entrer*



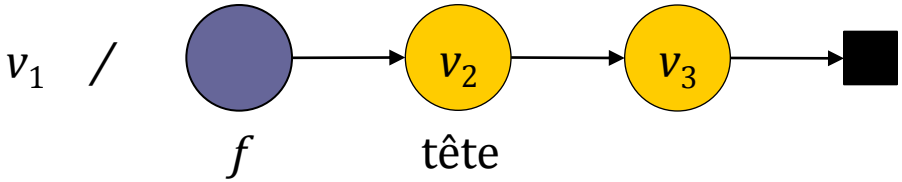

$entrer(f, v_3)$



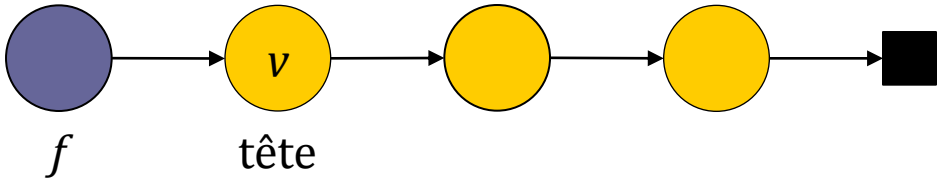
# *sortir*

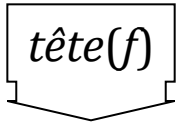






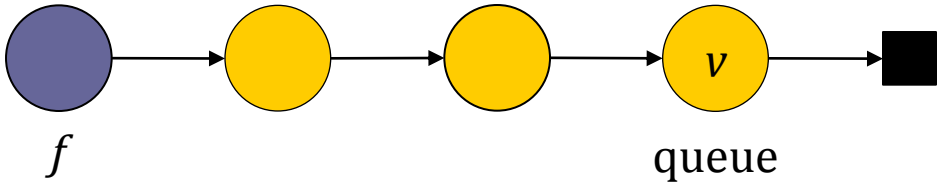
# *tête*

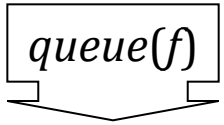



$v$

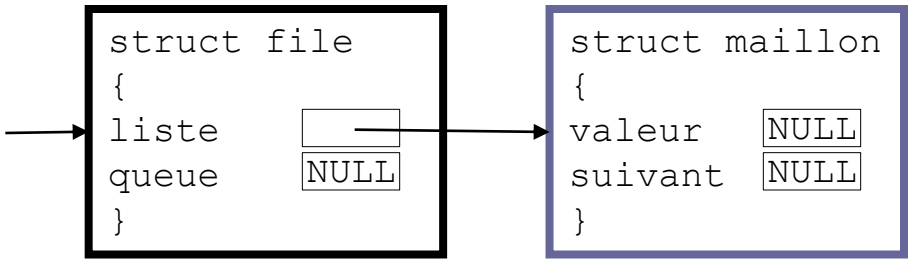
# *queue*



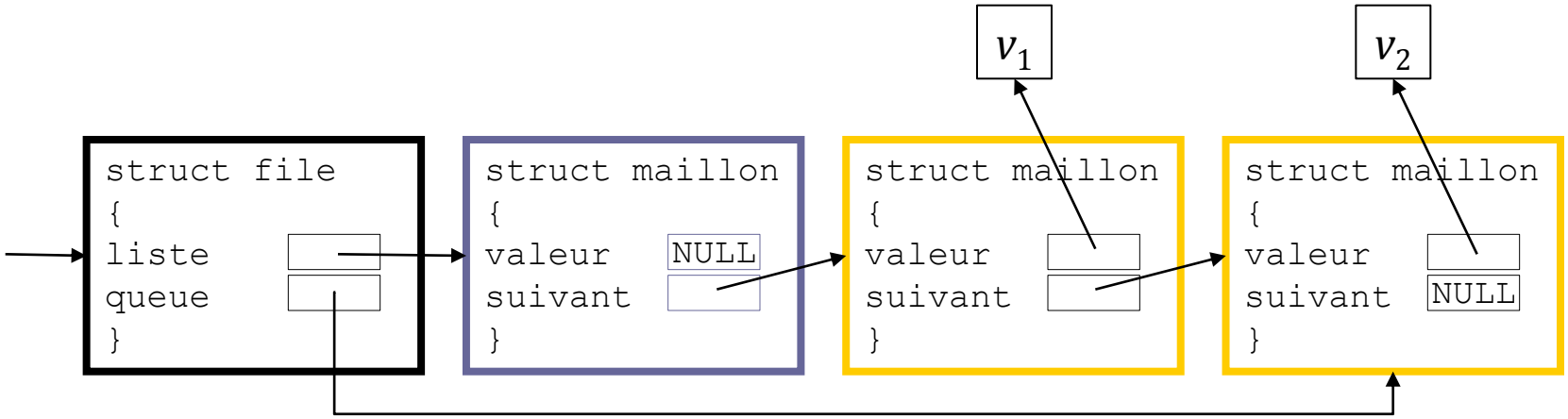

$v$

# Représentation d'une file en C




```
typedef struct file *File;
struct file
   {
   Liste liste;
   Maillon *queue;
   };
```

$\Leftarrow [] \Leftarrow$




$\Leftarrow [v_1, v_2] \Leftarrow$

# Interface

- La déclaration du type `File` ainsi que les déclarations des fonctions de l'interface sont enregistrés dans le fichier `TD-file.h`.
- Les définitions des fonctions de l'interface sont enregistrées dans le fichier `TD-file.c`.

# Fichier TD-file.h (1)

```
#ifndef TD_FILE
#define TD_FILE

#include <stdlib.h>
#include <stddef.h>
#include "TD-liste-lineaire.h"
#include "erreur.h"

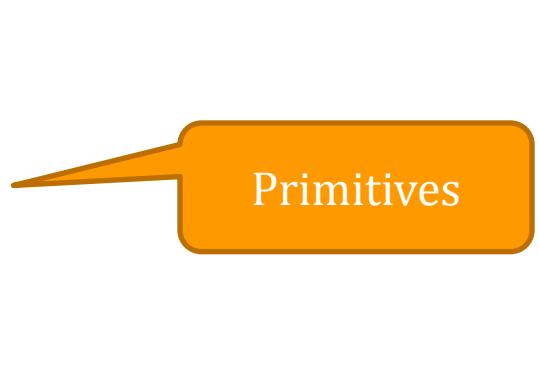
typedef struct file *File;

struct file
   {
   Liste liste;
   Maillon *queue;
   };
```

# Fichier `TD-file.h` (2)

```
File creer_file(void);
int file_vide(File f);
void entrer(File f, void *v);
void *sortir(File f);
void *tete(File f);
void *queue(File f);

#endif
```



# Créer une file

```
File creer_file(void)
   {
   File f;
   Liste l;
   f = (File) malloc(sizeof(struct file));
   if (f == NULL)
      erreur("creer_file");
   l = creer_liste();
   f->queue = l;
   f->liste = l;
   return f;
   }
```

# La file `f` est-elle vide ?

```
int file_vide(File f)
    {
    return liste_vide(f->liste);
    }
```

# Entrer une valeur `v` dans une file `f`

```
void entrer(File f, void *v)
    {
    f->queue = inserer_apres(f->queue, v);
    }
```

# Sortir une valeur d'une file `f`

```
void *sortir(File f)
   {
   void *v;
   if (file_vide(f))
      erreur("sortir");
   v = valeur(suivant(f->liste));
   supprimer_suivant(f->liste);
   if (file_vide(f))
      f->queue = f->liste;
   return v;
   }
```

# Valeur en tête d'une file `f`

```
void *tete(File f)
   {
   if (file_vide(f))
      erreur("tete");
   return valeur(suivant(f->liste));
   }
```

# Valeur en queue d'une file `f`

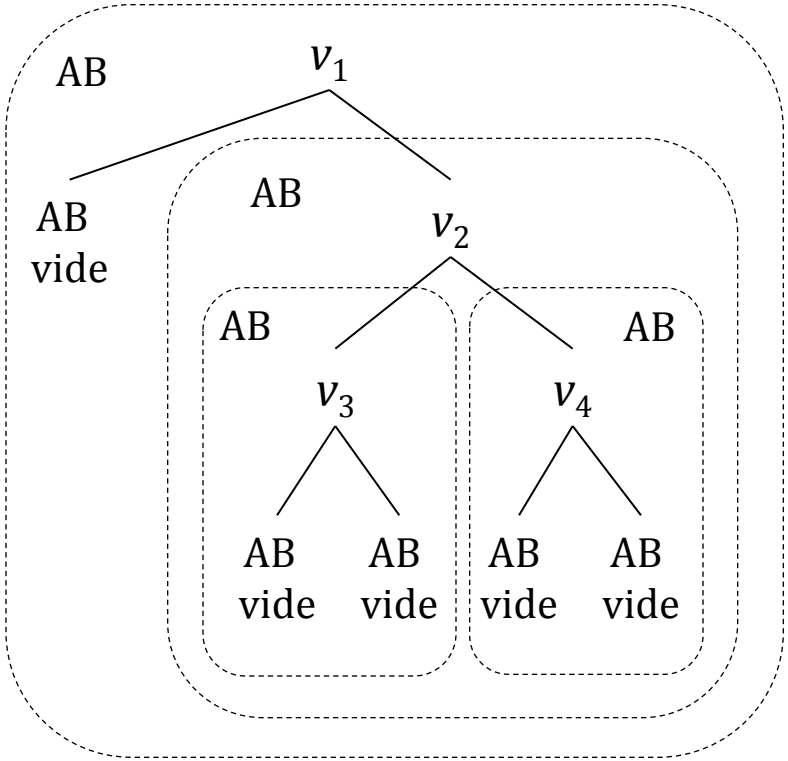
```
void *queue(File f)
   {
   if (file_vide(f))
      erreur("queue");
   return valeur(f->queue);
   }
```

# ARBRES BINAIRES

# Définition formelle

- Un **arbre binaire** est un ensemble fini de valeurs qui est :
  - soit **vide**,
  - soit constitué d'une **racine** et des valeurs de deux arbres binaires disjoints appelés **sous-arbre gauche** et **sous-arbre droit** de la racine.

  (Donald E. Knuth, *The Art of Computer Programming*, Volume 1, Fundamental Algorithms, 3rd Edition, p. 312)




$arbre(v_1, arbre(), arbre(v_2, arbre(v_3, arbre(), arbre()), arbre(v_4, arbre(), arbre())))$

# Opérations primitives sur les arbres binaires (1)

- On note :
  - $Arbre(T)$ : le type des arbres binaires de valeurs de type $T$

# Opérations primitives sur les arbres binaires (2)

- Création d'un arbre binaire vide
  - $cons\text{-}arbre\text{-}vide : Vide \rightarrow Arbre(T)$
  - $cons\text{-}arbre\text{-}vide : \mapsto ()$
- Création d'un arbre binaire non vide
  - $cons\text{-}arbre : T \times Arbre(T) \times Arbre(T) \rightarrow Arbre(T)$
  - $cons\text{-}arbre : (v, a_1, a_2) \mapsto arbre\ (v, a_1, a_2)$
- Test d'arbre vide
  - $arbre\text{-}vide : Arbre(T) \rightarrow Booléen$
  - $arbre\text{-}vide : arbre() \mapsto$ vrai
  - $arbre\text{-}vide : arbre(v, a_1, a_2) \mapsto$ faux

# Opérations primitives sur les arbres binaires (3)

- Racine d'un arbre
  - $racine : Arbre(T) \rightarrow T$
  - $racine : () \mapsto$ erreur !
  - $racine : arbre(v, a_1, a_2) \mapsto v$
- Sous-arbre gauche de la racine d'un arbre
  - $gauche : Arbre(T) \rightarrow Arbre(T)$
  - $gauche : () \mapsto$ erreur !
  - $gauche : arbre(v, a_1, a_2) \mapsto a_1$
- Sous-arbre droit de la racine d'un arbre
  - $droit : Arbre(T) \rightarrow Arbre(T)$
  - $droit : () \mapsto$ erreur !
  - $droit : arbre(v, a_1, a_2) \mapsto a_2$

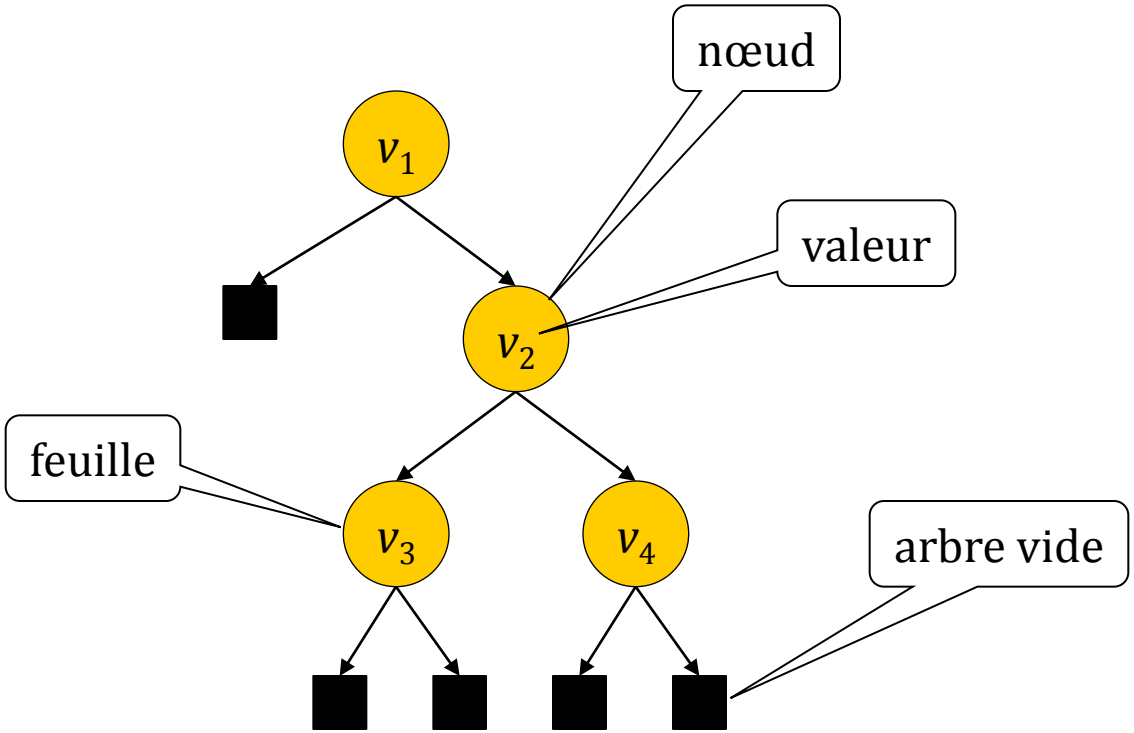
# Autres opérations

- Nous en étudierons 4 :
  - test de feuille (racine d'un arbre binaire dont les deux sous-arbres sont vides),
  - hauteur d'un arbre binaire,
  - construction d'un arbre binaire de recherche par insertion, dans un arbre binaire de recherche, d'une valeur relativement à une relation d'ordre donnée,
  - recherche d'une valeur dans un arbre binaire de recherche.

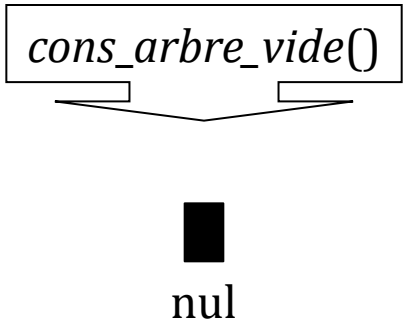
# Notre représentation d'un arbre binaire

- Un arbre binaire non vide de racine *v* est représenté par un **nœud** qui contient :
  - la valeur *v*,
  - l'identifiant du sous-arbre gauche,
  - l'identifiant du sous-arbre droit.
- Un nœud a un **identifiant**.
- Un arbre binaire a un identifiant qui est :
  - l'identifiant du nœud qui contient sa racine, si cet arbre n'est pas vide,
  - l'identifiant nul, si cet arbre est vide (on peut voir l'identifiant nul comme celui d'un nœud factice de contenu vide).

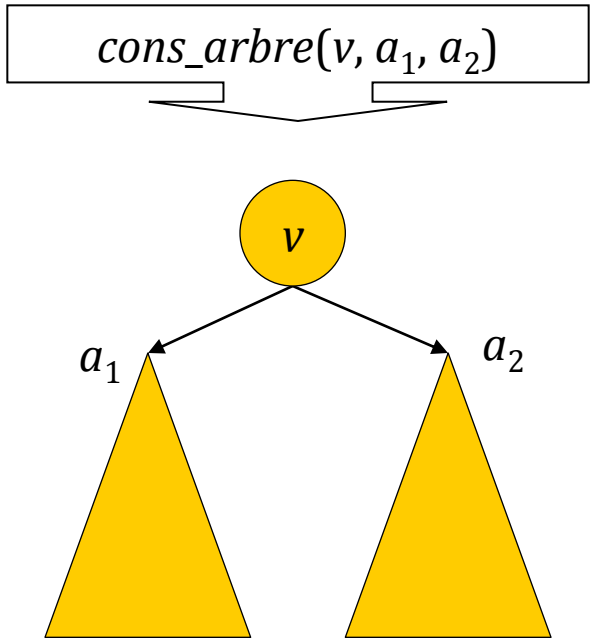
# Représentation d'un arbre binaire



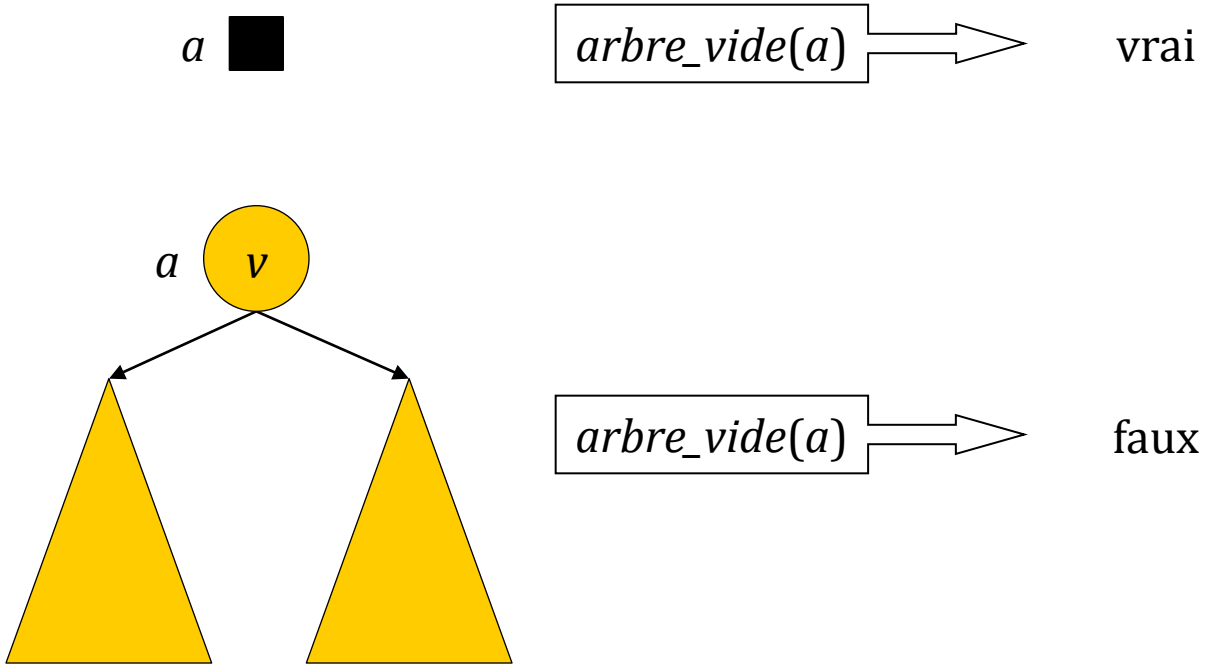
# *cons-arbre-vide*



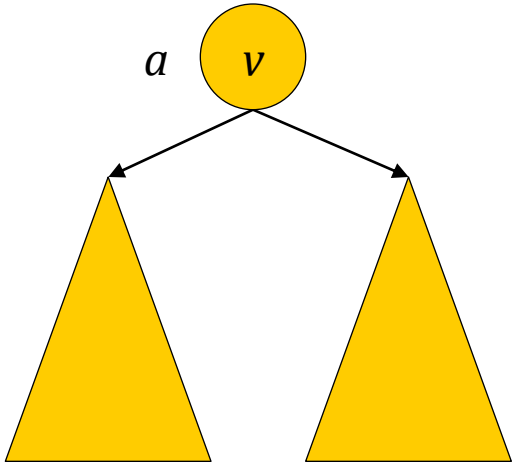
# *cons-arbre*



# *arbre-vide*

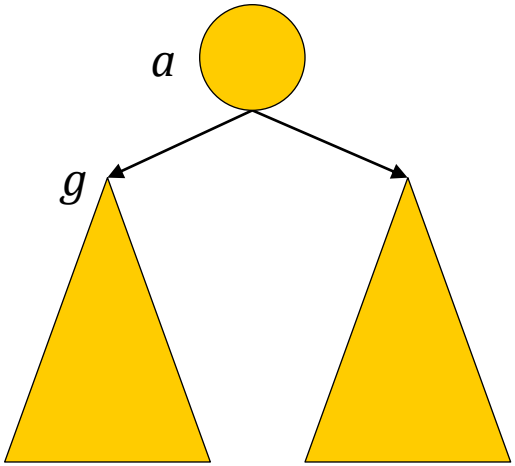


# *racine*



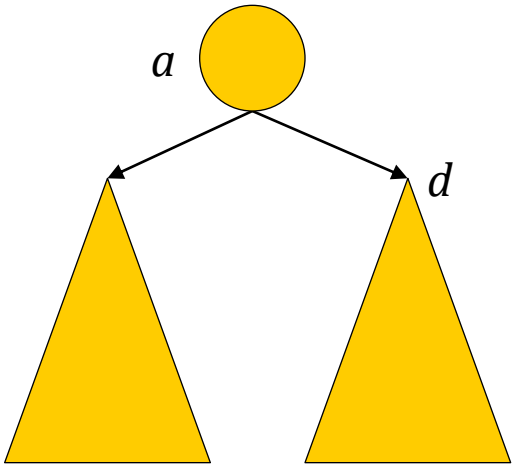

$racine(a)$ ⇨ $v$

# *gauche*




$gauche(a)$ ⇨ $g$

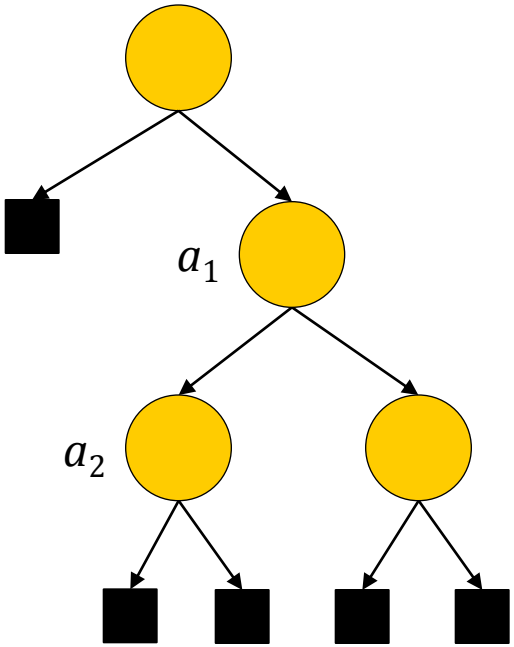
# *droit*




$droit(a)$ ⇨ $d$

# *feuille*

Une feuille est la racine d'un arbre binaire
dont le sous-arbre gauche et le sous-arbre droit sont vides.



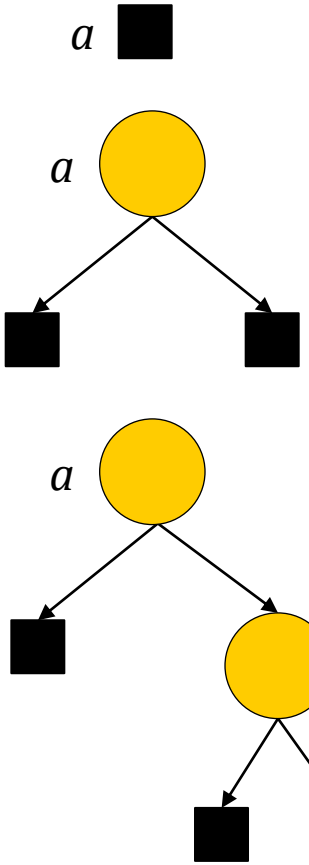

$feuille(a_1)$ → faux

$feuille(a_2)$ → vrai

# *hauteur-arbre*

La **hauteur** d'un arbre binaire est égale à 0 s'il est vide ou au nombre de nœuds − 1 de sa plus longue branche sinon.
On a :

- *hauteur*(*a*) = 0, si *a* est vide ou si la racine de *a* est une feuille,
- *hauteur*(*a*) = 1 + *max*(*hauteur*(*g*), *hauteur*(*d*)), si *a* est un arbre non vide de sous-arbre gauche *g* et de sous-arbre droit *d*.

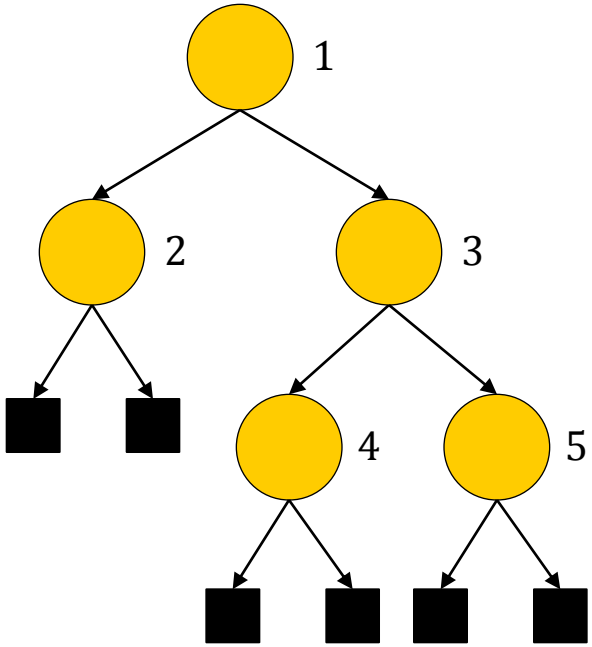



*hauteur_arbre*(*a*) → 0

*hauteur_arbre*(*a*) → 0

*hauteur_arbre*(*a*) → 1

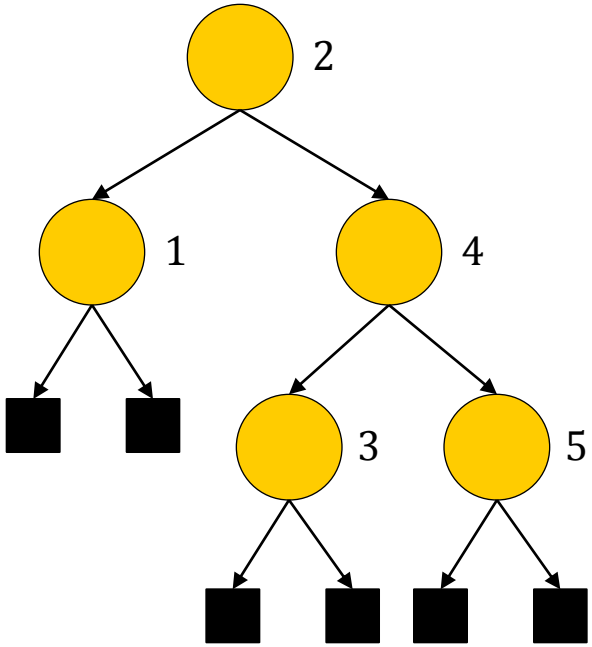
# Parcours préfixe d'un arbre binaire




**Parcours préfixe**
racine
sous-arbre gauche
sous-arbre droit

parcours-préfixe(a) =
- **si** ¬*arbre-vide*(*a*) **alors**
  - **traiter** *racine*(*a*)
  - *parcours-préfixe*(*gauche*(*a*))
  - *parcours-préfixe*(*droit*(*a*))

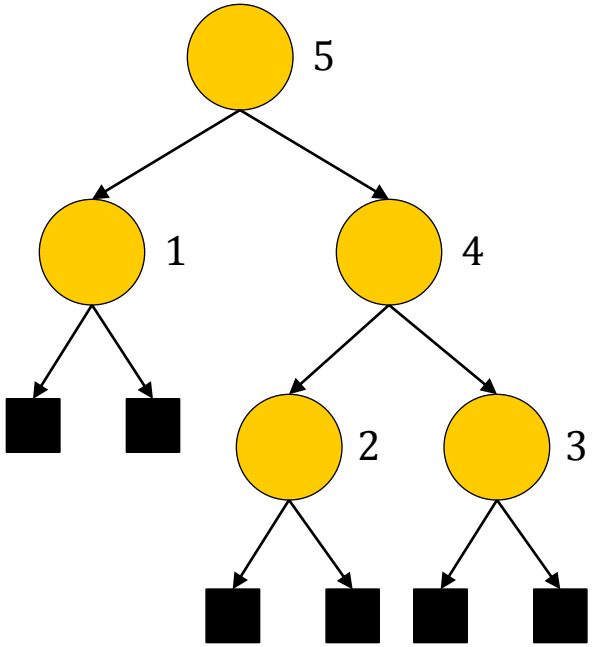
# Parcours infixe d'un arbre binaire




**Parcours infixe**
sous-arbre gauche
racine
sous-arbre droit

*parcours-infixe*(*a*) =
- **si** ¬*arbre-vide*(*a*) **alors**
  - *parcours-infixe*(*gauche*(*a*))
  - **traiter** *racine*(*a*)
  - *parcours-infixe*(*droit*(*a*))

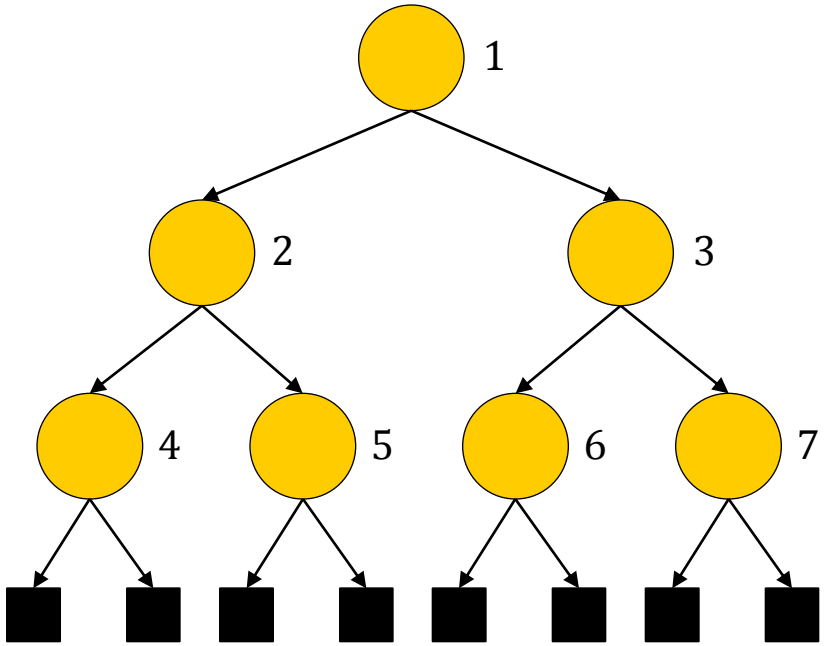
# Parcours postfixe d'un arbre binaire




**Parcours postfixe**
sous-arbre gauche
sous-arbre droit
racine

*parcours-postfixe*(*a*) =
- **si** ¬*arbre-vide*(*a*) **alors**
  - *parcours-postfixe*(*gauche*(*a*))
  - *parcours-postfixe*(*droit*(*a*))
  - **traiter** *racine*(*a*)

# Parcours en largeur d'un arbre binaire




**Parcours en largeur**
niveau par niveau

*parcours-en-largeur*(*a*) =
- **si** ¬*arbre-vide*(*a*) **alors**
- *f* = *file-vide*
- *entrer*(*f*, *a*)
- **faire**
  - *a* = *sortir*(*f*)
  - **traiter** *racine*(*a*)
  - *g* = *gauche*(*a*)
  - *d* = *droit*(*a*)
  - **si** ¬*arbre-vide*(*g*) **alors**
  - *entrer*(*f*, *g*)
  - **si** ¬*arbre-vide*(*d*) **alors**
  - *entrer*(*f*, *d*)

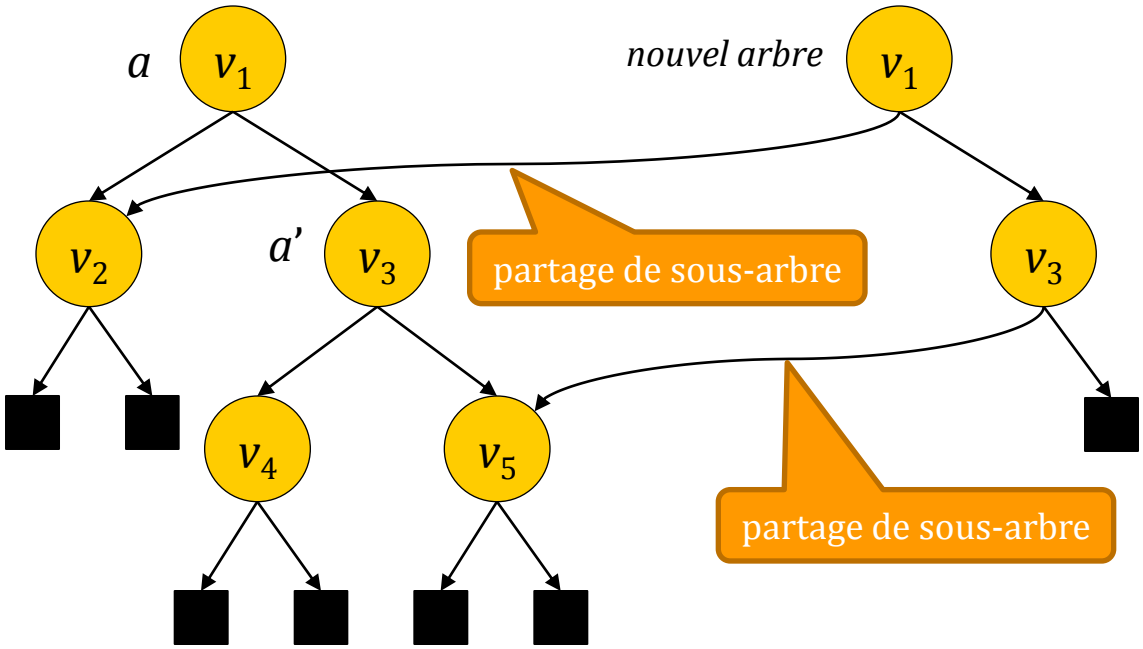
  **jusqu'à** *file-vide*(*f*)

# Modification d'un arbre binaire

- On notera qu'il n'y a pas de primitives d'insertion ou de suppression de sous-arbres, ou bien de remplacement de la valeur de la racine, comme c'est le cas pour les maillons d'une liste linéaire :
  - Il n'est donc pas possible de modifier une arbre « en place » .
- La solution est de construire, à partir de l'arbre à modifier, un nouvel arbre comportant qui peut partager des sous-arbres avec l'arbre initial.

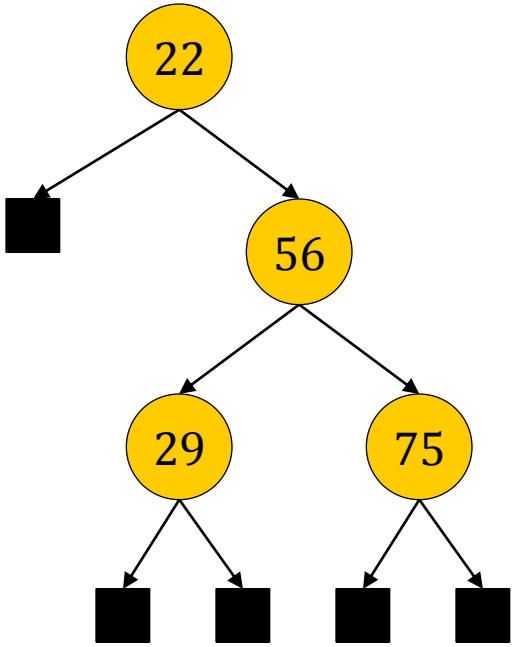
# Modification par reconstruction

$cons\_arbre(racine(a), gauche(a), cons\_arbre(racine(droit(a)), droit(droit(a)), arbre\_vide()));$



# Arbres binaires de recherche

- Un **arbre binaire de recherche** est un arbre binaire dans lequel les valeurs sont placées relativement à une **relation d'ordre** $\preccurlyeq$ , de la façon suivante.
- Pour tout sous-arbre de racine $r$ :
  - les valeurs contenues dans le sous-arbre gauche, sont les valeurs $v$ telles que $v \preccurlyeq r$,
  - les valeurs contenues dans le sous-arbre droit sont les valeurs $v$ telles que $v \succ r$.
- La recherche d'une valeur ne nécessite que le parcours de la branche à laquelle appartient cette valeur.




$\preccurlyeq$ est la relation $\leq$ sur les entiers

# Insertion et recherche d'une valeur dans un arbre binaire de recherche

- L'**insertion** et la **recherche** d'une valeur dans un arbre binaire de recherche sont réalisées par les fonctions *cons-arbre-recherche* et *chercher*.
- Dans le cas général, les valeurs de l'arbre sont structurées et sont ordonnées par rapport à la valeur de l'une de leurs composantes que nous appellerons la **clé**. Par exemple, le nom ou l'âge d'une personne.
- Lorsque les valeurs de l'arbre sont atomiques, des nombres par exemple, la clé est égale à la valeur elle-même.
- Soit $T$ le type des valeurs de l'arbre binaire de recherche et $K$ celui de leurs clés.
- En insertion, il faudra fournir une fonction de comparaison de deux valeurs, définie de la façon suivante :
  - $comp_{vv} : T \times T \rightarrow \{-1, 0, 1\}$
  - $comp_{vv} : (v_1, v_2) \mapsto -1$ si clé$(v_1) \prec$ clé$(v_2)$, 0 si clé$(v_1) =$ clé$(v_2)$, 1 si clé$(v_1) \succ$ clé$(v_2)$
- En recherche, il faudra fournir une fonction de comparaison d'une clé et d'une valeur, définie de la façon suivante :
  - $comp_{kv} : K \times T \rightarrow \{-1, 0, 1\}$
  - $comp_{kv} : (k, v) \mapsto -1$ si $k \prec$ clé$(v)$, 0 si $k =$ clé$(v)$, 1 si $k \succ$ clé$(v)$

# *cons-arbre-recherche* (1)

- *cons-arbre-recherche* : $Arbre(T) \times T \times (T \times T \rightarrow \{-1, 0, 1\}) \rightarrow Arbre(T)$
- *cons-arbre-recherche* $(a, v, comp_{vv})$ =

  **si** *arbre-vide(a)* **alors**

    *cons-arbre*(*v*, *cons-arbre-vide*(), *cons-arbre-vide*())

  **sinon**

    **soit** $a = (r, g, d)$ **dans**

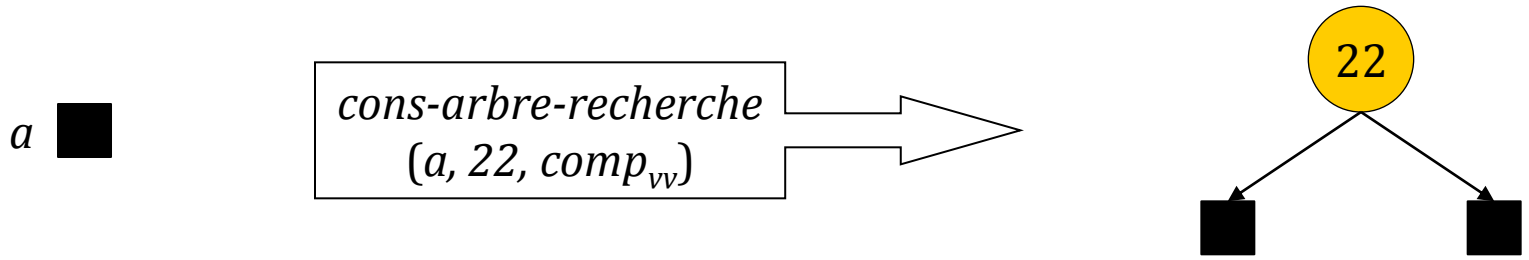
      **si** $comp_{vv}(v, r) \leq 0$ **alors**

        *cons-arbre*(*r*, *cons-arbre-recherche*(*g*, $comp_{vv}$, *v*), *d*)

      **sinon**

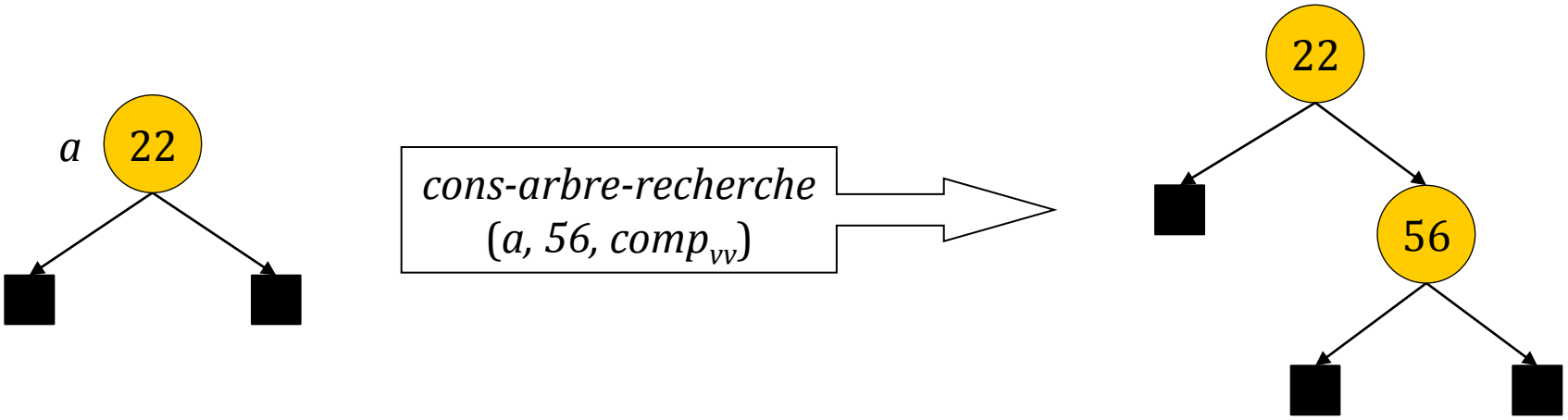
        *cons_arbre*(*r*, *g*, *cons-arbre-recherche*(*d*, $comp_{vv}$, *v*))

# *cons-arbre-recherche* (2)



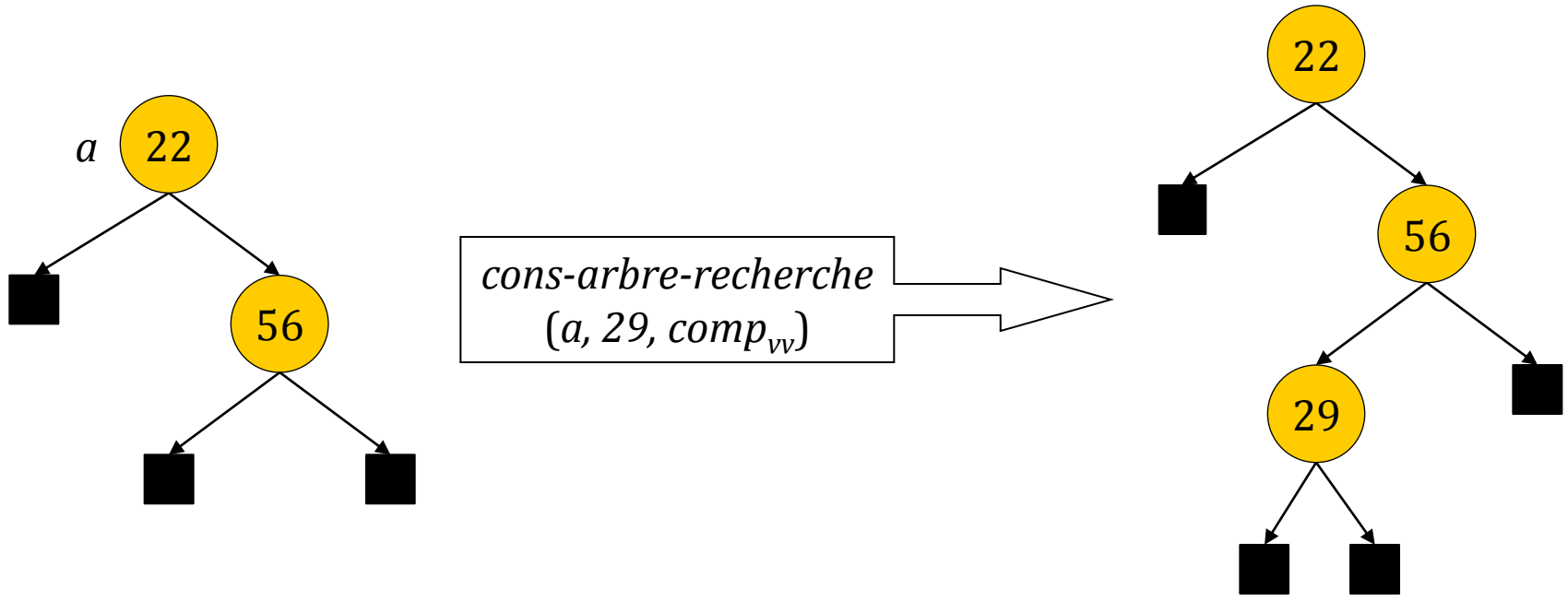

**Relation d'ordre** : ≤ sur les entiers

# *cons-arbre-recherche* (3)



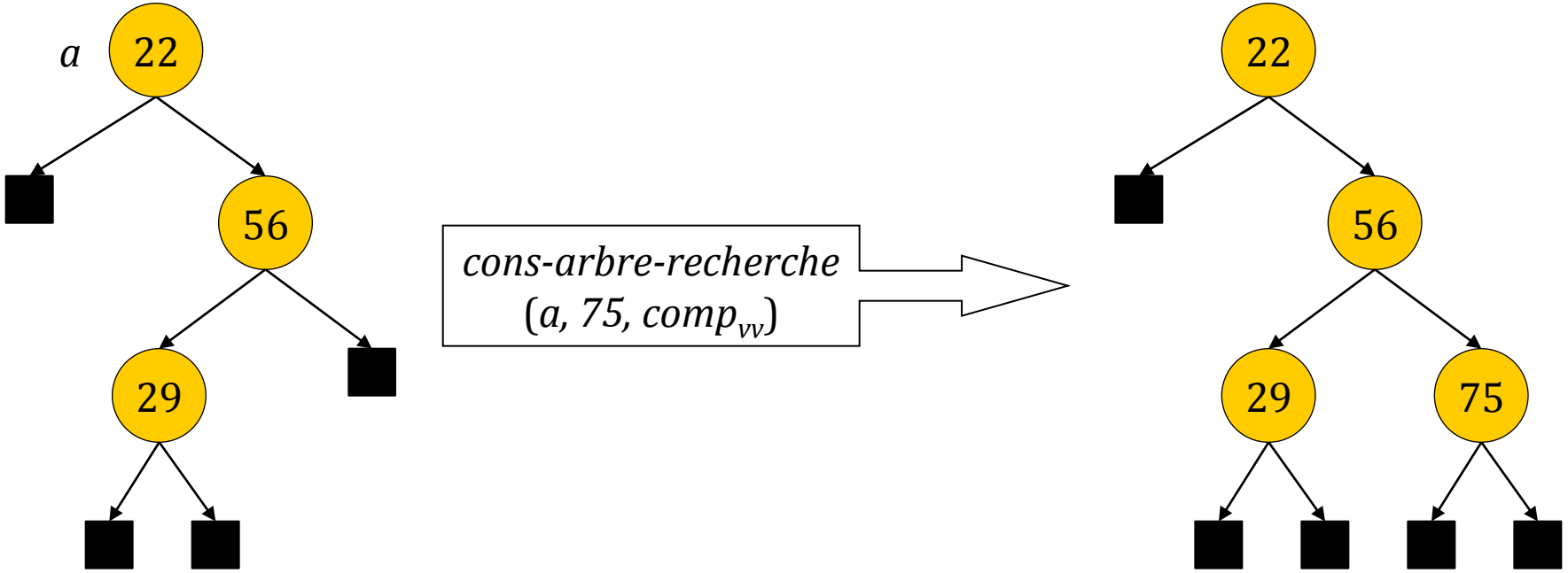

**Relation d'ordre** : ≤ sur les entiers

# *cons-arbre-recherche* (4)




**Relation d'ordre** : ≤ sur les entiers

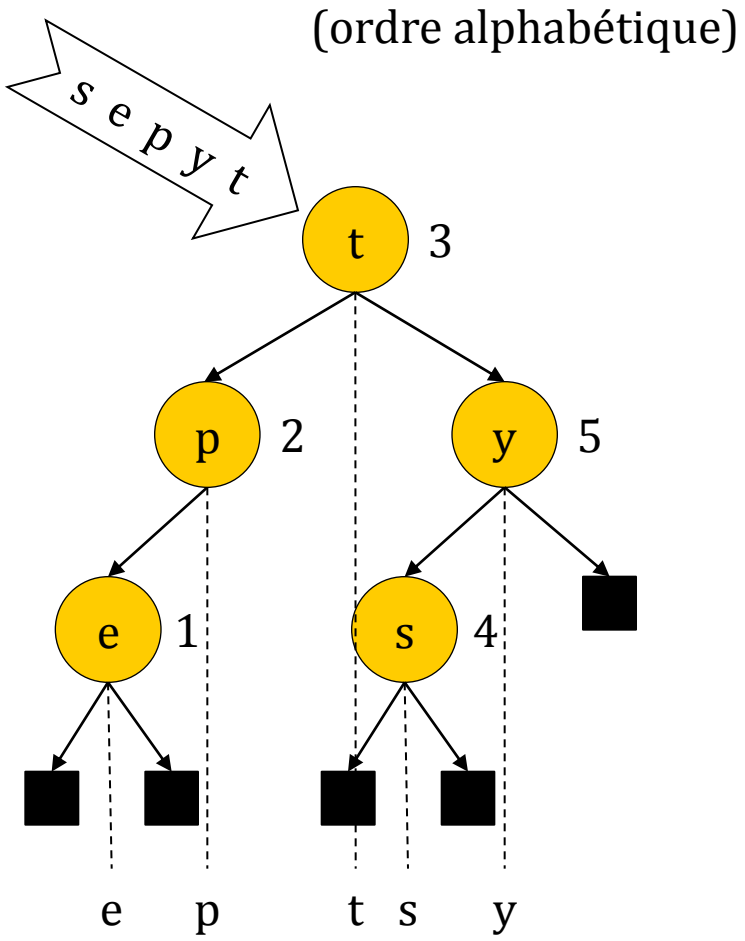
# *cons-arbre-recherche* (5)




**Relation d'ordre** : ≤ sur les entiers

# Parcours ordonné des valeurs d'un arbre binaire de recherche

- Pour accéder aux valeurs d'un arbre binaire de recherche dans l'ordre selon lequel cet arbre a été construit, il suffit de parcourir l'arbre en ordre « infixe ».



# *chercher* (1)

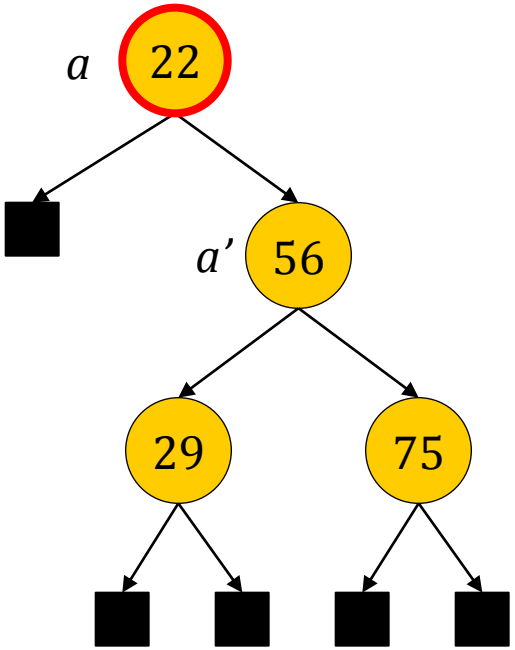
- $chercher : Arbre(T) \times K \times (K \times T \rightarrow \{-1, 0, 1\}) \rightarrow Arbre(T)$
- $chercher(a, k, comp_{kv})$ =

**si** $arbre\_vide(a)$ **alors**
  $a$
**sinon**
  **soit** $a = (r, g, d)$ **dans**
    **si** $comp_{kv}(k, r) < 0$ **alors**
      $chercher(g, k, comp_{kv})$
    **sinon**
      **si** $comp_{kv}(k, r) = 0$ **alors**
        $a$
      **sinon**
        $chercher(d, k, comp_{kv})$

# *chercher* (2)

- Si la recherche a réussi, la fonction *chercher* retourne le sous-arbre dont la racine a pour clé la clé recherchée.
- Si cette clé n'est pas unique, il faudra relancer la recherche dans le sous-arbre gauche de ce sous-arbre
- Si la recherche a échoué, la fonction *chercher* retourne un arbre vide.

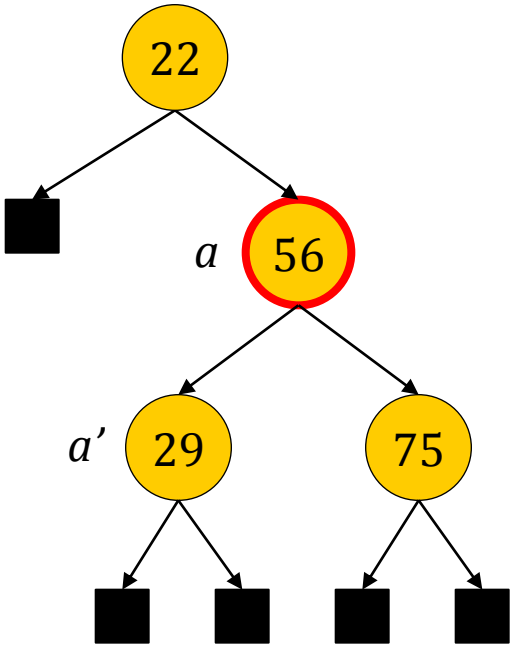
# *chercher* (3)




*chercher*(*a*, *29*) ⇒ *chercher*(*a'*, *29*)

**Relation d'ordre** : ≤ sur les entiers

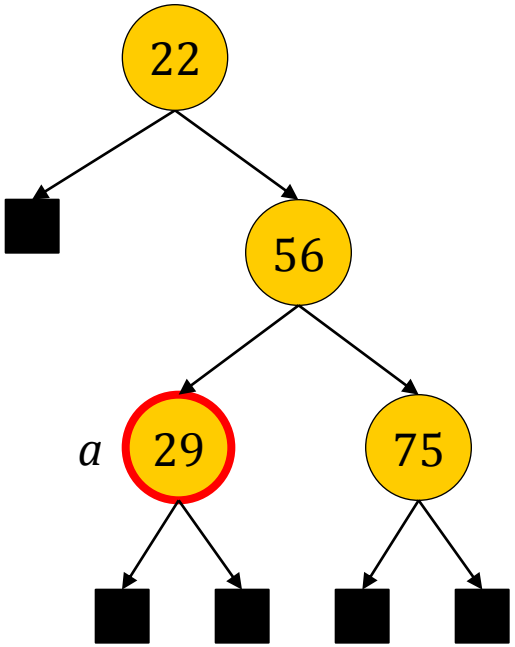
# *chercher* (4)




*chercher*(*a*, *29*) ⇒ *chercher*(*a'*, *29*)

**Relation d'ordre** : ≤ sur les entiers

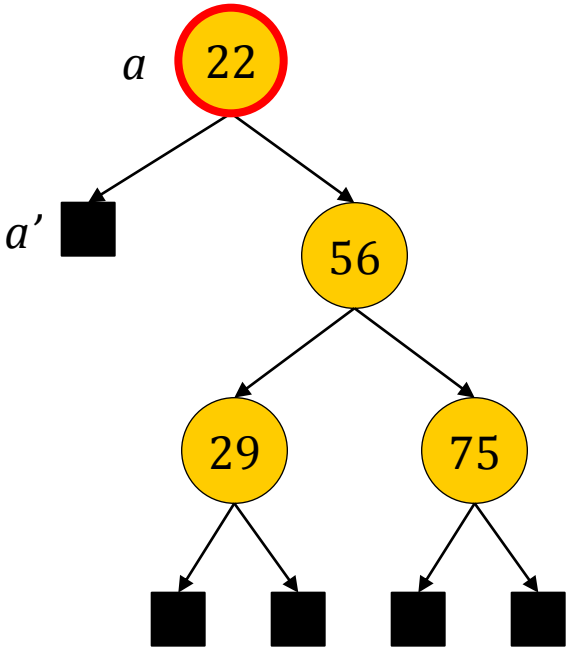
# *chercher* (5)




*chercher*(*a*, *29*) ⟹ *a*hercher(*a'*, *29*)

**Relation d'ordre** : ≤ sur les entiers

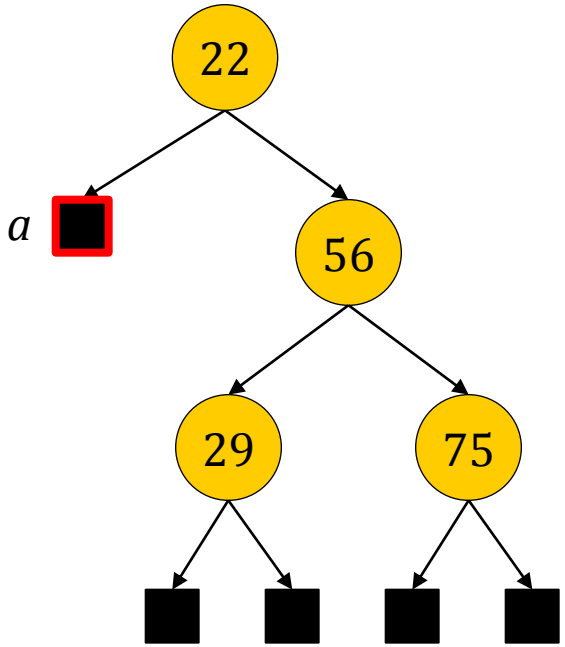
# *chercher* (6)




*chercher*(*a*, *12*) ⇒ *chercher*(*a'*, *12*)

**Relation d'ordre** : ≤ sur les entiers

# *chercher* (7)



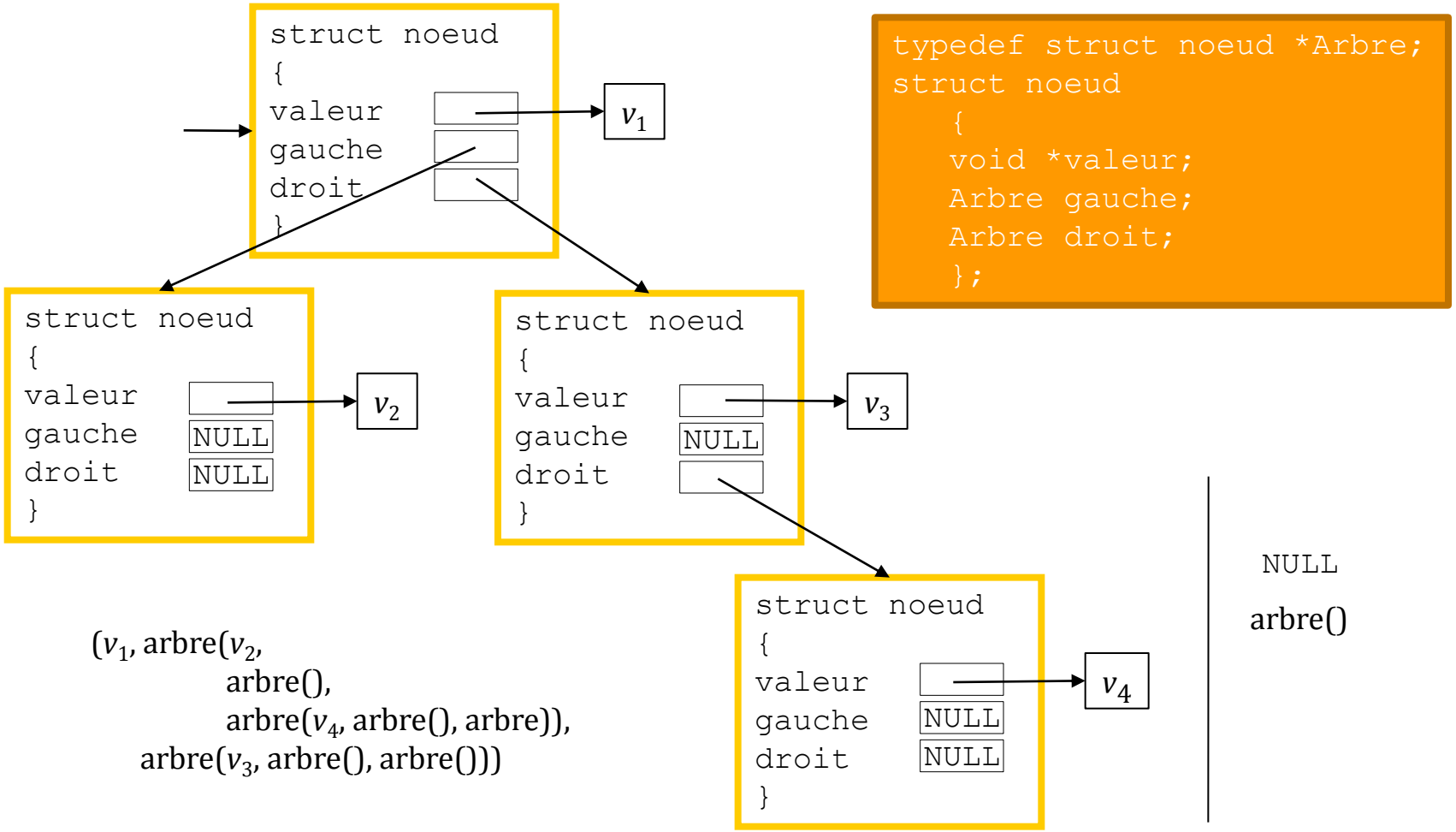

*chercher*(*a*, *12*) ⟹ *a* (échec)

**Relation d'ordre** : ≤ sur les entiers

# Coût d'une recherche

- Le coût d'une recherche peut-être mesuré en nombre de nœuds parcourus.
- La recherche d'une valeur se fait le long d'une seule branche.
- Le coût maximum est donc égal à $h + 1$, si $h$ est la hauteur de l'arbre.
- Si un arbre contient $n$ nœuds :
  - sa hauteur maximum est égale $n - 1$ lorsque l'arbre n'a qu'une seule branche.
  - sa hauteur minimum est égale à $\log_2(n + 1) - 1$ lorsque l'arbre est équilibré (toutes les branches ont la même longueur).
- Le coût de recherche est donc compris entre $n$ et $\log_2(n + 1)$. En moyenne, il est logarithmique ($\mathcal{O}(\log n)$).

# Représentation d'un arbre binaire en C



# Interface

- La déclaration du type `Arbre` ainsi que les déclarations des fonctions de l'interface sont enregistrés dans le fichier `TD-arbre-binaire.h`.
- Les définitions des fonctions de l'interface sont enregistrées dans le fichier `TD-arbre-binaire.c`.

# Fichier TD-arbre-binaire.h (1)

```
#ifndef TD_ARBRE_BINAIRE
#define TD_ARBRE_BINAIRE

#include <stdlib.h>
#include <stddef.h>
#include "erreur.h"

typedef struct noeud *Arbre;

struct noeud
   {
   void *valeur;
   Arbre gauche;
   Arbre droit;
   };
```

# Fichier TD-arbre-binaire.h (2)

```
Arbre cons_arbre_vide(void);
Arbre cons_arbre(void *v, Arbre g, Arbre d);
void *racine(Arbre a);
Arbre gauche(Arbre a);
Arbre droit(Arbre a);
int arbre_vide(Arbre a);

int feuille(Arbre a);
int hauteur_arbre(Arbre a);
Arbre cons_arbre_recherche(Arbre a, void *v, int (*comp)(const
   void *, const void *));
Arbre chercher(Arbre a, void *cle, int (*comp)(const void *,
   const void *));

#endif
```

# Construire un arbre binaire vide

```
Arbre cons_arbre_vide(void)
    {
    return NULL;
    }
```

# Construire un arbre binaire de racine r, de sous-arbre gauche g et de sous-arbre droit d

```
Arbre cons_arbre(void *v, Arbre g, Arbre d)
   {
   Arbre a;
   a = (Arbre) malloc(sizeof(struct noeud));
   if (a == NULL)
      erreur("cons_arbre");
   a->valeur = v;
   a->gauche = g;
   a->droit = d;
   return a;
   }
```

# L'arbre binaire `a` est-il vide ?

```
int arbre_vide(Arbre a)
    {
    return a == NULL;
    }
```

# Racine d'un arbre binaire a

```
void *racine(Arbre a)
   {
   if (arbre_vide(a))
      erreur("racine");
   return a->valeur;
   }
```

# Sous-arbre gauche de la racine d'un arbre binaire a

```
Arbre gauche(Arbre a)
   {
   if (arbre_vide(a))
      erreur("gauche");
   return a->gauche;
   }
```

# Sous-arbre droit de la racine d'un arbre binaire a

```
Arbre droit(Arbre a)
   {
   if (arbre_vide(a))
      erreur("droit");
   return a->droit;
   }
```

# La racine de l'arbre binaire `a` est-elle une feuille ?

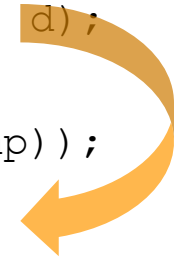
```
int feuille(Arbre a)
   {
   if (arbre_vide(a))
      erreur("feuille");
   return arbre_vide(gauche(a)) && arbre_vide(droit(a));
   }
```

# Hauteur de l'arbre binaire a

```
int hauteur_arbre(Arbre a)
   {
   int hg, hd;
   if (arbre_vide(a) || feuille(a))
      return 0;
   hg = hauteur_arbre(gauche(a));
   hd = hauteur_arbre(droit(a));
   if (hg > hd)
      return 1 + hg;
   else
      return 1 + hd;
   }
```

# Construire un arbre binaire de recherche par insertion d'une valeur `v` dans un arbre binaire de recherche `a` en utilisant la fonction `comp` de comparaison de deux valeurs

```
Arbre cons_arbre_recherche(Arbre a, void *v,
   int (*comp)(const void *, const void *))
   {
   void *r;
   Arbre g, d;
   if (arbre_vide(a))
      return cons_arbre(v, NULL, NULL);
   r = racine(a);
   g = gauche(a);
   d = droit(a);
   if (comp(v, r) <= 0)
      return cons_arbre(r, cons_arbre_recherche(g, v, comp), d);
   else
      return cons_arbre(r, g, cons_arbre_recherche(d, v, comp));
   }
```



# Chercher une valeur de clé `cle` dans un arbre binaire de recherche `a` en utilisant la fonction `comp` de comparaison d'une clé et d'une valeur

```
Arbre chercher(Arbre a, void *cle,
   int (*compvv)(const void *, const void *))
   {
   void *v;
   int c;
   if (arbre_vide(a))
      return a;
   c = comp(cle, racine(a));
   if (c < 0)
      return chercher(gauche(a), cle, comp);
   if (c == 0)
      return a;
   return chercher(droit(a), cle, comp);
   }
```

# Autres opérations

- Parcours  d'un arbre binaire
  - préfixe
  - infixe
  - suffixe
  - en largeur

# Parcours préfixe d'un arbre binaire a (racine – gauche – droit)

```
void parcours_prefixe(Arbre a)
   {
   if (!arbre_vide(a))
      {
      Traiter la valeur contenue dans la racine de l'arbre a
      parcours_prefixe(gauche(a));
      parcours_prefixe(droit(a));
      }
   }
```

# Parcours infixe d'un arbre binaire `a` (gauche – racine – droit)

```
void parcours_infixe(Arbre a)
   {
   if (!arbre_vide(a))
      {
      parcours_infixe(gauche(a));
      Traiter la valeur contenue dans la racine de l'arbre a
      parcours_infixe(droit(a));
      }
   }
```

# Parcours suffixe d'un arbre binaire `a` (gauche – droit - racine)

```
void parcours_suffixe(Arbre a)
   {
   if (!arbre_vide(a))
      {
      parcours_suffixe(gauche(a));
      parcours_suffixe(droit(a));
      Traiter la valeur contenue dans la racine de l'arbre a
      }
   }
```

# Parcours en largeur d'un arbre binaire a

```
void parcours_en_largeur(Arbre a)
   {
   File f;
   Arbre g, d;
   if (!arbre_vide(a))
      {
      f = creer_file();
      entrer(f, a);
      do
         {
         a = arbre(sortir(f));
         Traiter la valeur contenue dans la racine de l'arbre a
         g = gauche(a);
         d = droit(a);
         if (!arbre_vide(g))
            entrer(f, g);
         if (!arbre_vide(d))
            entrer(f, d);
         }
      while (!file_vide(f));
      }
   }
```