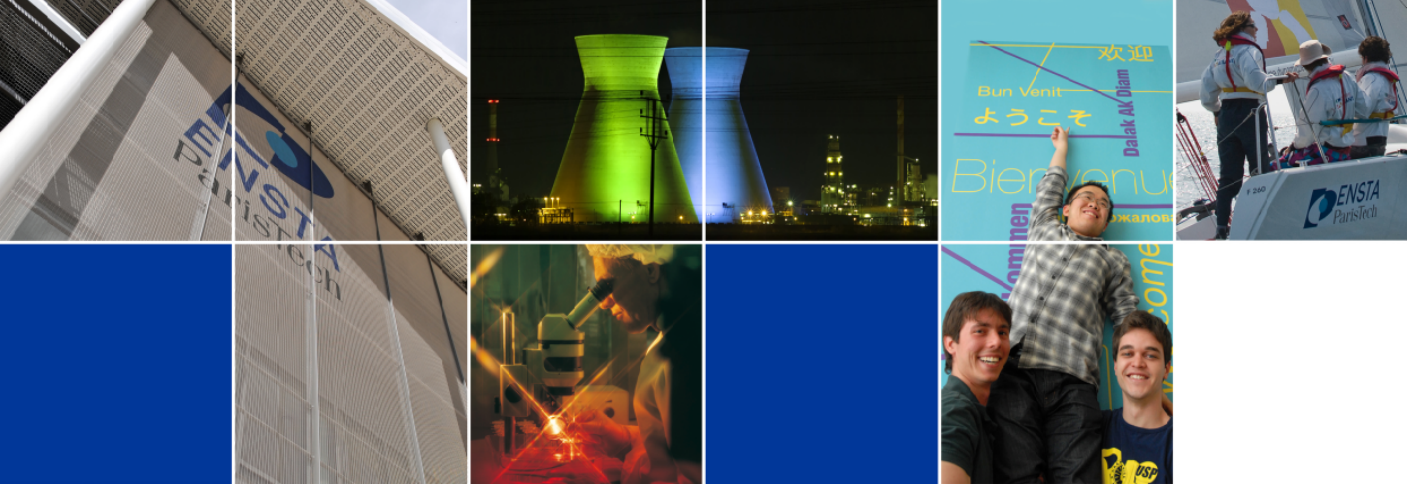


Structures de données en C

Stage Liesse

François Pessaux

https://perso.ensta-paris.fr/~pessaux/teaching/liesse

Laboratoire U2IS
ENSTA Paris
2021

ENSTA

IP PARIS

Bienvenue à toutes et tous. . .

- Le sujet :

  examiner l'implantation en C des structures de données usuelles
- Le présentateur :
  - Enseignant-chercheur en informatique à l'ENSTA depuis +9 ans.
  - Responsable de l'informatique en $1^{\text{ère}}$ année.
  - Cours en 1A et 2A.
  - Responsable de parcours de 3A.
- Le plan :
  - Listes chaînées
  - Tables de hachage
  - Piles
  - Files
  - Arbres
  - Files de priorité
  - Graphes

Listes chaînées

Structure de liste

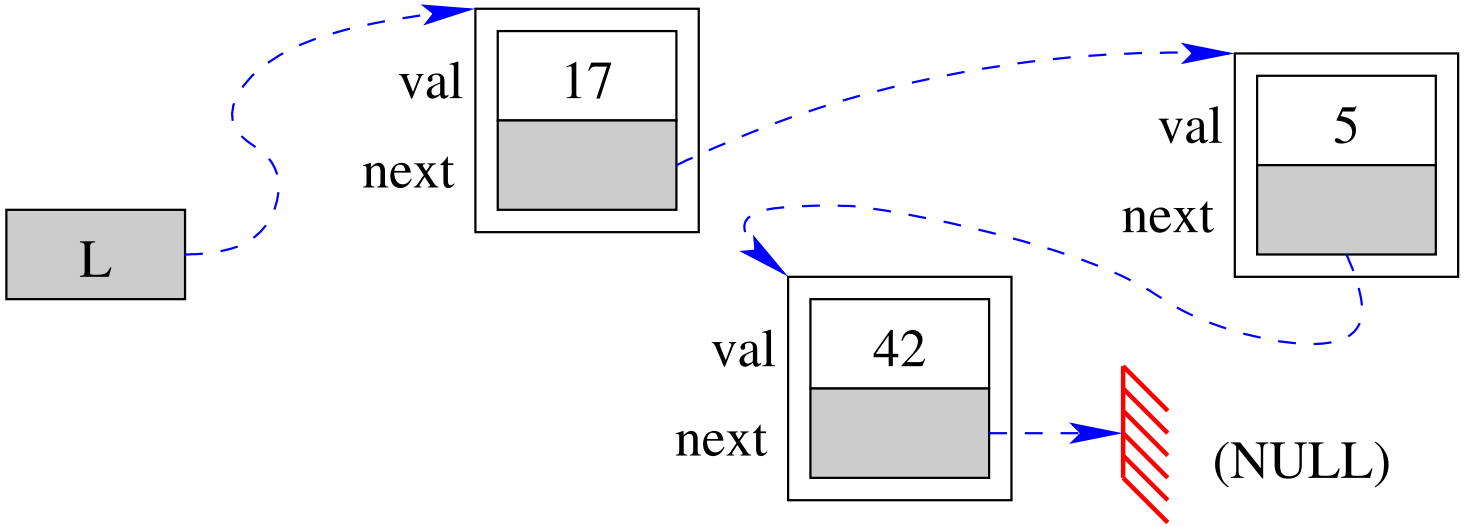
- Une liste c'est :
  - une liste vide
  - **ou** un élément suivi d'une liste.
- En C, pas de types inductifs ⇒ encodage à base de pointeurs :
  - On forme une « *chaîne* ».
  - Chaque élément est un maillon de la chaîne.
  - On passe d'un élément au suivant (précédent) en suivant un pointeur.

```
struct list_t {
  int data;
  struct list_t *next;
};
```

- On ne sait pas efficacement accéder au $i^{\text{ème}}$ élément.
- On « *tient* » une liste par le pointeur vers son premier élément.
- Le champ `next` du dernier élément pointe vers NULL.

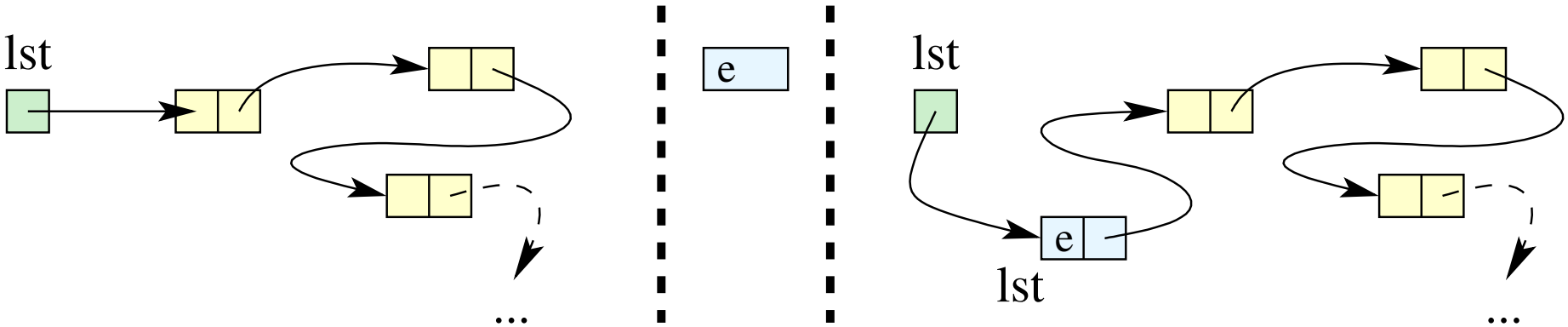
Organisation mémoire

- En mémoire, éléments dispersés ⇒ ≠ tableau où éléments contigus.
- Impossible d'accéder directement au $i^{\text{ème}}$ élément.
- Nécessité de suivre toute la chaîne (parcourir la liste).
- ⇒ Toujours garder accès à la tête de la liste.



Implantation en C : ajout en tête (« *insert_head* »)

```c
struct list_t* insert_head (int elem, struct list_t *lst) {
  struct list_t *tmp = malloc (sizeof (struct list_t)) ;
  if (tmp == NULL) return NULL ;
  tmp->data = elem ;
  tmp->next = lst ;
  return tmp ;
}
```



Ajout en tête : les trucs à ne **pas** faire !

```c
struct list_t* insert_head (int elem, struct list_t *lst) {
  struct list_t *tmp; /* malloc nécessaire ! */
  tmp->data = elem ;
  tmp->next = lst ;
  return tmp ;
}
```

- Avoir un pointeur ne signifie pas avoir de la mémoire !
- Il faut l'initialiser avec une adresse de zone allouée.

```c
struct list_t* insert_head (int elem, struct list_t *lst) {
  struct list_t tmp; /* Variable locale ! */
  tmp.data = elem ;
  tmp.next = lst ;
  return &tmp ;
}
```

- Retourne l'adresse d'une variable locale qui meurt à la fin de la fonction.

Ajout en tête : les trucs à ne **pas** faire également !

```
struct list_t insert_head (int elem, struct list_t* lst) {
   ...
}
```

ou

```
struct list_t insert_head (int elem, struct list_t lst) {
   ...
}
```

- . . . ne marchent pas non plus !
- Écrasement mémoire, adresse de paramètre (qui meurt à la fin de la fonction), et plus si affinités . . .

Implantation en C : impression

- Écriture itérative :

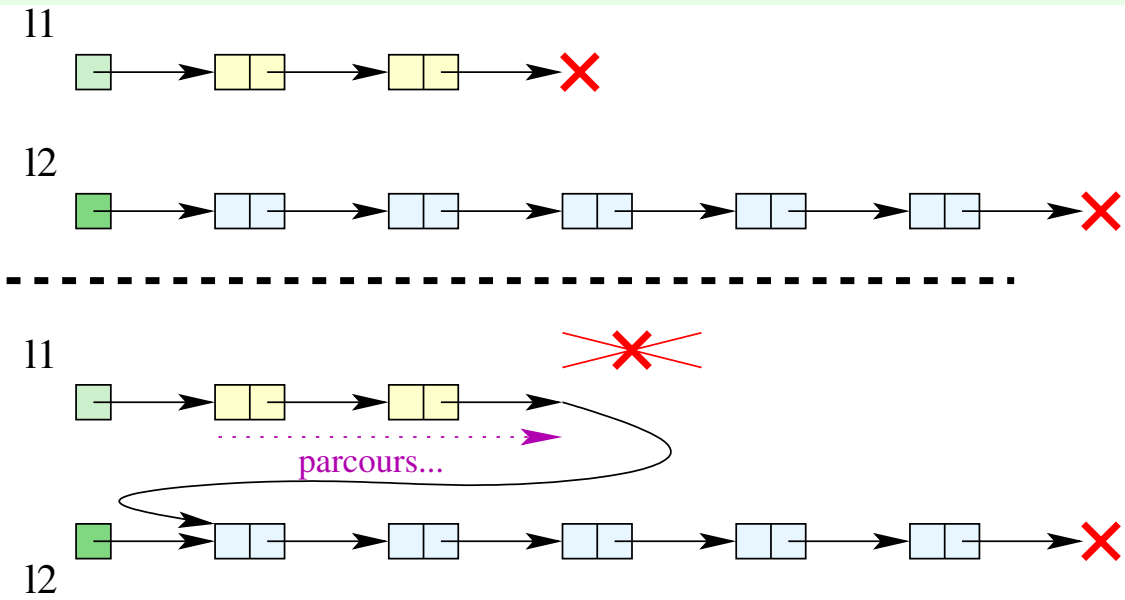
```
void print (struct list_t *lst) {
  printf ("[") ;
  while (lst != NULL) {
    printf (" %d", lst->data) ;
    lst = lst->next ;
  }
  printf (" ]") ;
}
```

- Écriture récursive (plus coûteuse en pile si compilé naïvement) :

```
void rec_print (struct list_t *lst) {
  if (lst != NULL) {
    printf (" %d", lst->data) ;
    rec_print (lst->next) ;
  }
}
```

Implantation en C : concaténation (« *append* »)

```
struct list_t* append (struct list_t *l1, struct list_t *l2) {
  struct list_t *tmp = l1 ;
  if (tmp == NULL) return l2 ;
  while (tmp->next != NULL) tmp = tmp->next ;
  tmp->next = l2 ;
  return l1 ;
}
```



Implantation en C : libération

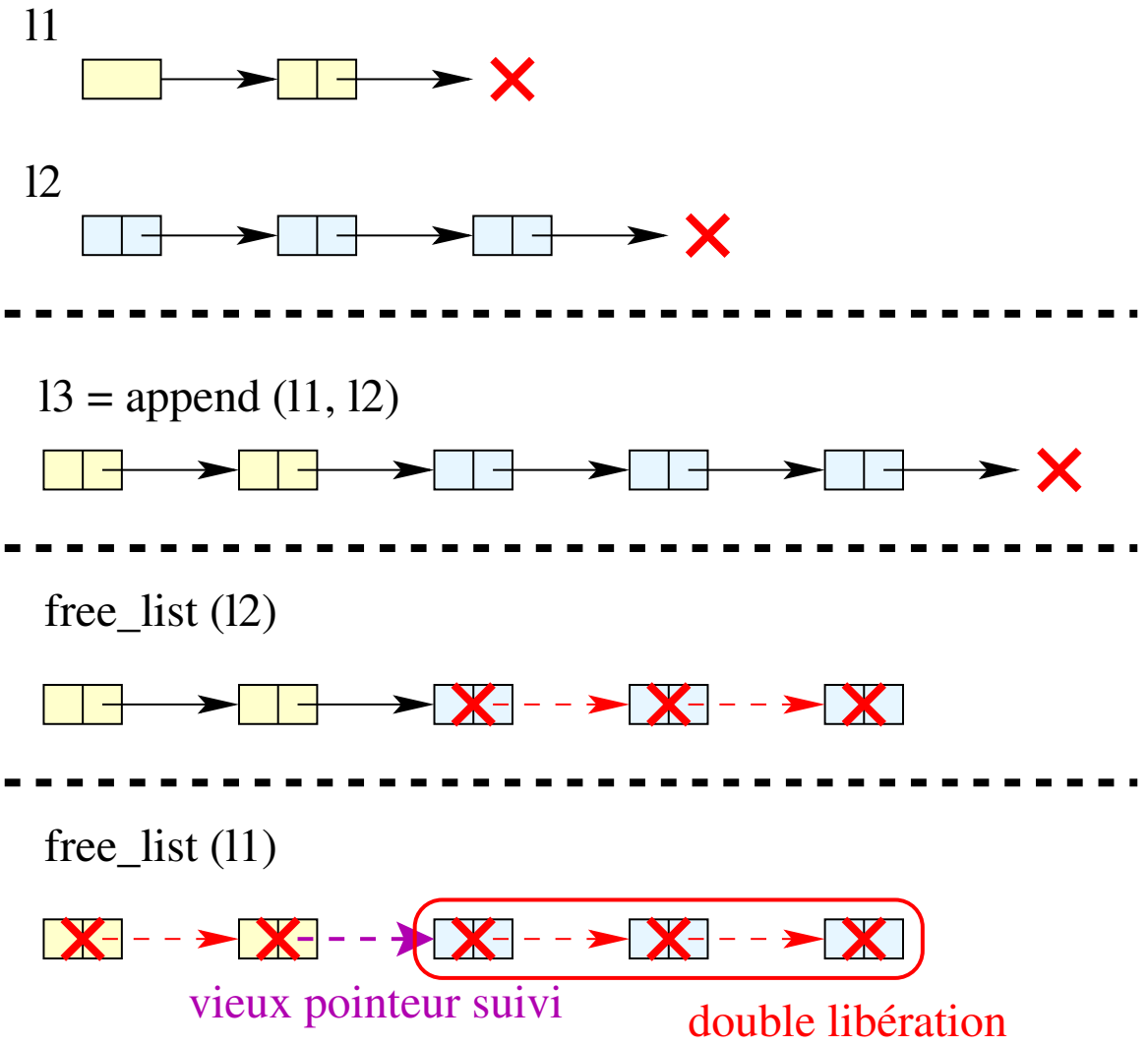
- Écriture itérative :

```
void free_list (struct list_t *lst) {
  struct list_t *tmp ;
  while (lst != NULL) {
    tmp = lst->next ;
    free (lst) ;
    lst = tmp ;
  }
}
```

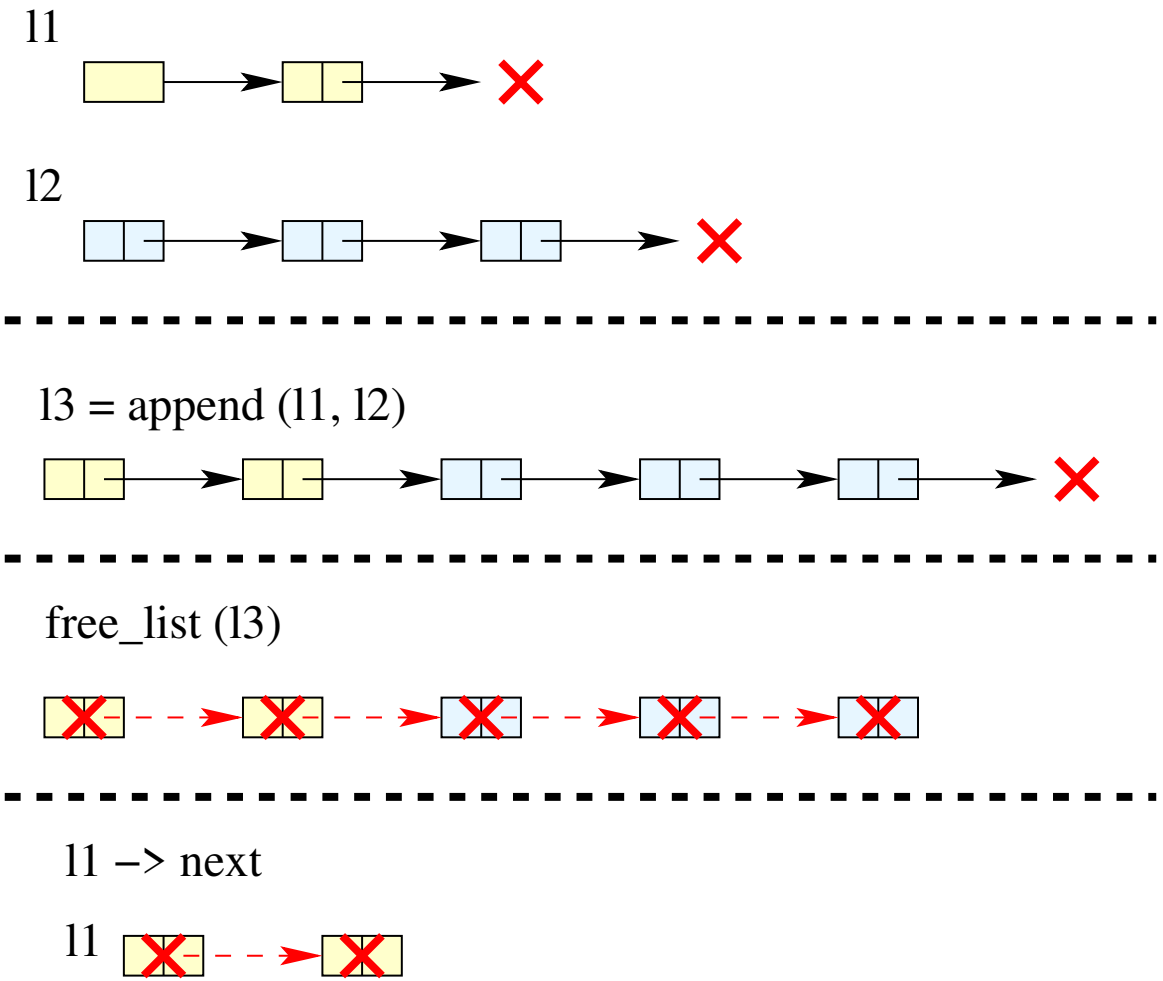
- Écriture récursive (plus coûteuse en pile) :

```
void rec_free_list (struct list_t *lst)
{
  if (lst != NULL) {
    rec_free_list (lst->next) ;
    free (lst) ;
  }
}
```

Implantation en C : double libération



Implantation en C : libération anticipée



Variations sur les listes

- Beaucoup de variantes :
  - Doublement chaînées (pointeur `prev`) pour reculer.
  - Listes circulaires : dernier élément pointe sur le premier.
  - Listes circulaires doublement chaînées ...
- La liste peut être plus que le pointeur vers son premier élément :
  - Peut contenir le nombre d'éléments.
  - Peut contenir le pointeur vers le dernier élément.
  - ...

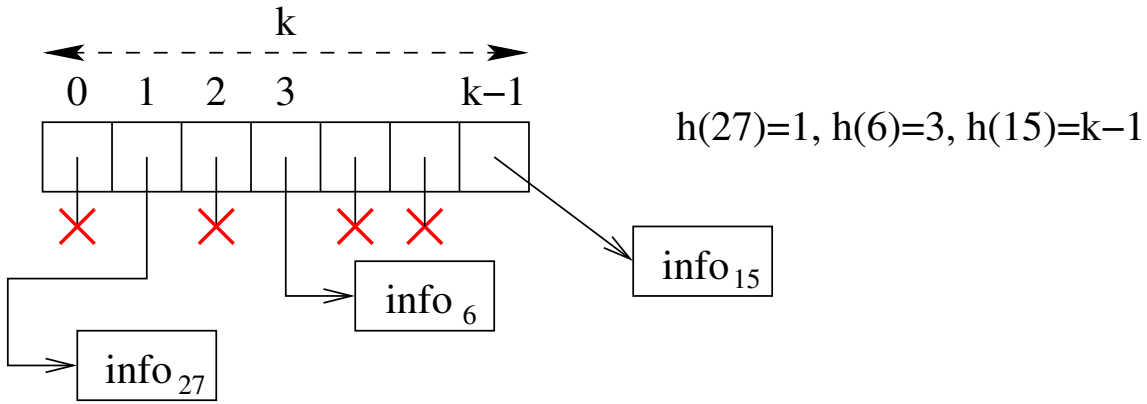
```c
struct list_t {
  int nb_elems ;
  struct cell_t *head ;
  struct cell_t *last ;
};
```

```c
struct cell_t {
  int data1 ;
  int data2 ;
  struct cell_t *next ;
};
```

Tables de hachage

Table de hachage, dictionnaire (« *hastable* »)

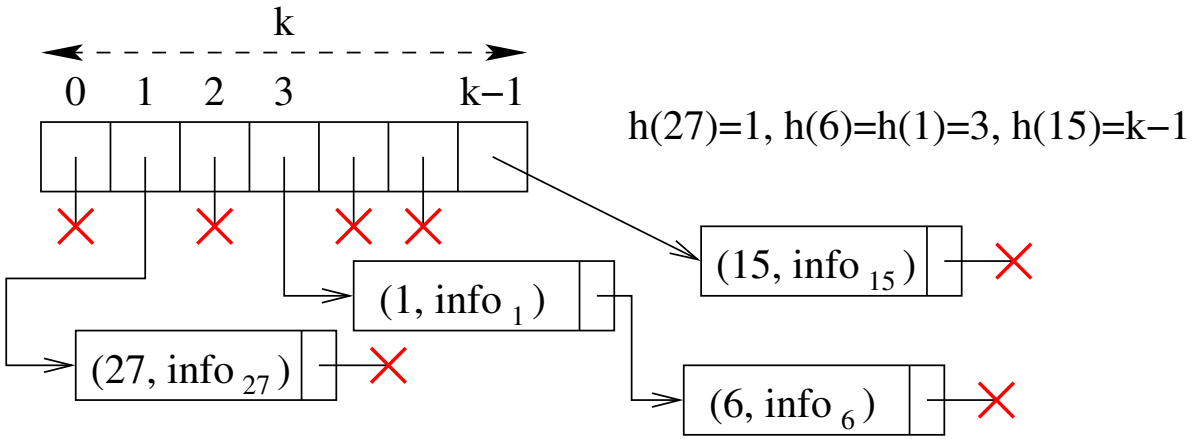
- But : retrouver une information à partir d'une clef m.
- Réduire l'espace des clefs possibles, m, à un espace plus petit, k (celui des clefs réellement utiles).
- Fonction de hachage $h : [0; m-1] \to [0; k-1]$ avec $k < m$.
- On utilise un tableau de taille k.
- On stocke chaque information dans le tableau à l'indice $h(clef)$.
  - Chaque case contient l'information dont le hachage de la clef vaut l'indice de cette case.



- Si 2 clefs ont le même « hash » : collision.

Table de hachage et collisions

- Collision : plusieurs infos avec le même hachage de clef.
- Au lieu de mettre 1 info dans la table, on met une liste chaînée d'infos.
- Toutes les infos d'une liste ont le même hachage de leur clef.



- Autre solution : adressage ouvert → utiliser la 1ère case contiguë libre.

Collisions et adressage ouvert

```c
typedef char* key_t ;    /* Type des clefs. */
typedef int value_t ;    /* Type des valeurs. */

/* Type des cellules de table. */
struct cell_t {
  key_t key ;     /* Clef. Pour gérer les collisions. */
  value_t val ;   /* Valeur associée. */
};

/* Type des tables de hachage. */
struct hash_tbl_t {
  unsigned int size ;          /* Nombre max de cellules. */
  struct cell_t **storage ;    /* Tableau des cellules. */
};
```

Insertion en adressage ouvert

```
bool add (struct hash_tbl_t *tbl, key_t key, value_t val) {
  unsigned int i ;
  if (tbl == NULL || key == NULL) return false ;
  unsigned int hashcode = hash (key, tbl->size) ;
  i = hashcode ;
  while (tbl->storage[i] != NULL &&
         strcmp (tbl->storage[i]->key, key) != 0 &&
         i < tbl->size)
    i++ ;
  if (i == tbl->size) {
    i = 0 ;
    while (tbl->storage[i] != NULL &&
           strcmp (tbl->storage[i]->key, key) != 0 &&
           i < hashcode)
      i++ ;
    if (i == hashcode) return false ;
  }
  if (tbl->storage[i] == NULL) {
    tbl->storage[i] = malloc (sizeof (struct cell_t)) ;
    if (tbl->storage[i] == NULL) exit (EXIT_FAILURE) ;
    tbl->storage[i]->key = malloc (sizeof (char) * (strlen (key) + 1)) ;
    if (tbl->storage[i]->key == NULL) exit (EXIT_FAILURE) ;
    strcpy (tbl->storage[i]->key, key) ;
    tbl->storage[i]->val = val ;
  }
  else {
    tbl->storage[i]->val = val ;
  }
  return true ;
}
```

Recherche en adressage ouvert

```
bool find (struct hash_tbl_t *tbl, key_t key, value_t *ret_val) {
  unsigned int i ;
  if (tbl == NULL || key == NULL) return false ;
  unsigned int hashcode = hash (key, tbl->size) ;
  if (tbl->storage[hashcode] != NULL &&
      strcmp (tbl->storage[hashcode]->key, key) == 0) {
    *ret_val = tbl->storage[hashcode]->val ;
    return true ;
  }
  i = hashcode ;
  while (i < tbl->size) {
    if (tbl->storage[i] != NULL && strcmp (tbl->storage[i]->key, key) == 0) {
      *ret_val = tbl->storage[i]->val ;
      return true ;
    }
    i++ ;
  }
  i = 0 ;
  while (i < hashcode) {
    if (tbl->storage[i] != NULL && strcmp (tbl->storage[i]->key, key) == 0) {
      *ret_val = tbl->storage[i]->val ;
      return true ;
    }
    i++ ;
  }
  return false ;
}
```

- ⚠ : si suppression, besoin de mémoriser les éléments « mal logés » ou parcourir la table.

Collisions et chaînage

```c
typedef char* key_t ;    /* Type des clefs. */
typedef int value_t ;    /* Type des valeurs. */

/* Type des cellules chaînées de table. */
struct cell_t {
  key_t key ;     /* Clef. Pour gérer les collisions. */
  value_t val  ;          /* Valeur mémorisée. */
  struct cell_t *next ;  /* Cellule suivante. */
};

/* Type des tables de hachage. */
struct hash_tbl_t {
  unsigned int size ;          /* Nombre max de cellules. */
  struct cell_t **storage ;    /* Tableau des cellules. */
};
```

Insertion avec chaînage

```
struct cell_t* insert_or_modify (struct cell_t *head,
                                 key_t key, value_t val) {
  struct cell_t *tmp = head ;
  while (tmp != NULL) {
    if (tmp->key == key) {
      tmp->val = val ;
      return head ;
    }
    tmp = tmp->next ;
  }
  return (insert_head (key, val, head)) ;
}

bool add (struct hash_tbl_t *tbl, key_t key, value_t val) {
  if (tbl == NULL || key == NULL) return false ;
  unsigned int hashcode = hash (key, tbl->size) ;
  struct cell_t *new_head =
    insert_or_modify (tbl->storage[hashcode], key, val) ;
  if (new_head == NULL) return false ;
  tbl->storage[hashcode] = new_head ;
  return true ;
}
```

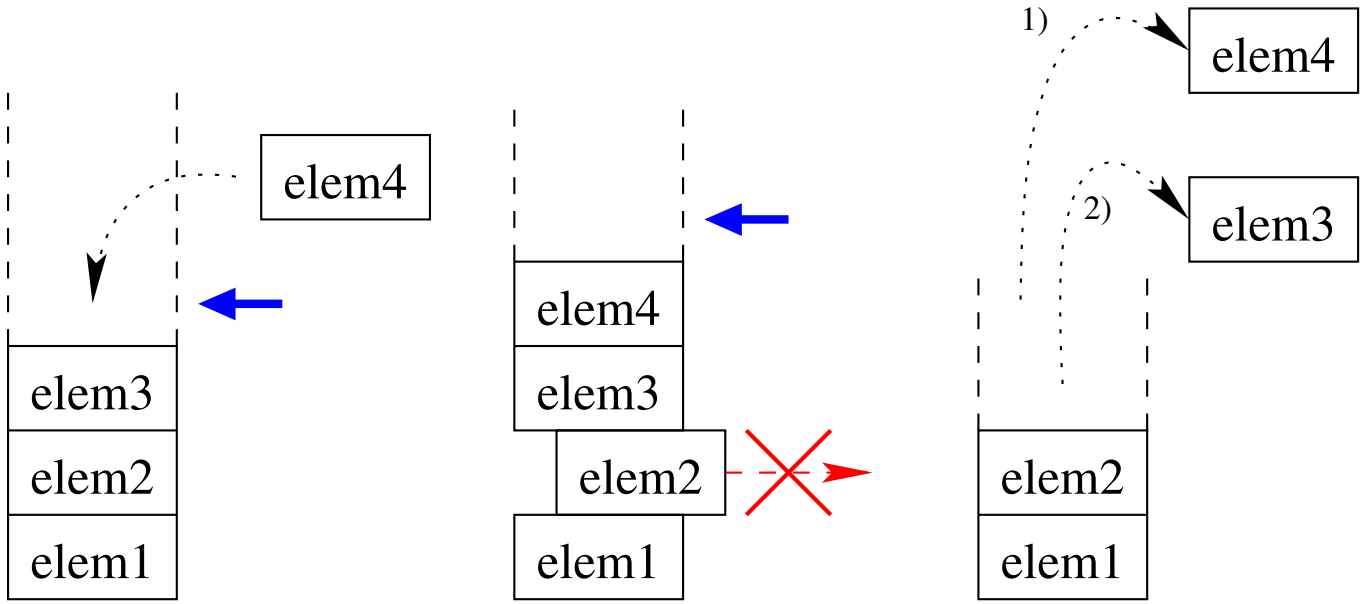
Recherche avec chaînage

```
bool find (struct hash_tbl_t *tbl, key_t key, value_t *ret_val)
    {
  if (tbl == NULL || key == NULL) return false ;
  unsigned int hashcode = hash (key, tbl->size) ;
  struct cell_t *lst = tbl->storage[hashcode] ;
  while (lst != NULL) {
    if (strcmp (lst->key, key) == 0) {
      *ret_val = lst->val ;
      return true ;
    }
    lst = lst->next ;
  }
  return false ;
}
```

Piles

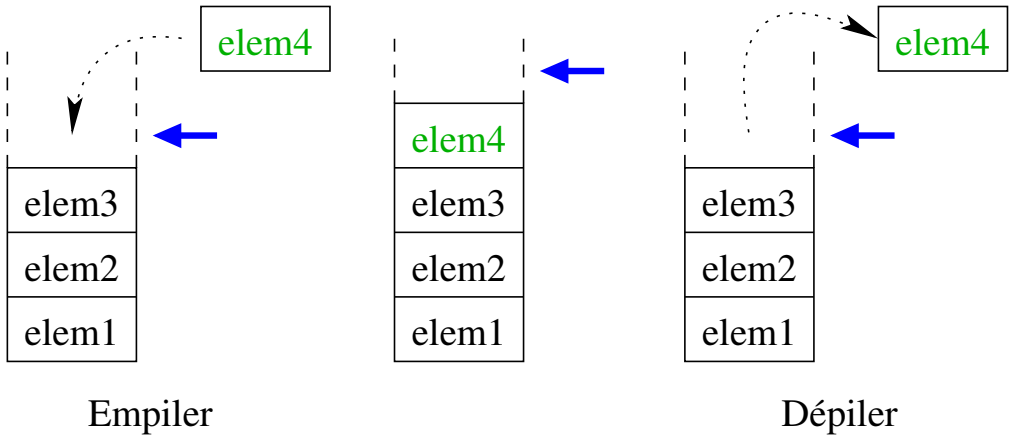
« Quand utiliser une pile ? » (« stack »)

- Comme dans la vie courante : stocker des choses comme une pile d'assiettes.
- ⇒ Dernier posé au sommet → premier retiré.
- En Anglais : LIFO (« **L***ast* **I***n* **F***irst* **O***ut* »).
- On peut consulter les éléments sous le sommet mais pas les retirer tant que ceux au-dessus sont dans la pile.



Opérations sur une pile

- **Empiler** (« *push* ») : Ajouter un élément au sommet.
  - `push` : (elem × stack) → stack
- **Dépiler** (« *pop* ») : Retirer l'élément du sommet.
  - `pop` : stack → (elem × stack)
- **Consulter** (« *peek* ») : Accéder à un élément sans le retirer.
  - `peek` : (stack × int) → elem
- Le « *pointeur de pile* » indique la prochaine place libre.
- Le « *pointeur de pile* » varie au cours des opérations (sauf `peek`).
- On peut tester si la pile est vide.



Pile : implantation avec un tableau de taille fixe

- Ici, pile de **int**s avec une taille maximale MAX_SIZE
  - Utiliser un tableau dynamique pour supprimer cette restriction.

```c
#define MAX_SIZE 64
struct stack_t {
  unsigned int sp ;          /* Pointeur de pile. */
  int storage[MAX_SIZE] ; /* Mémoire de pile. */
};
```

- Pile ≡ simple tableau.
- « *Pointeur* » de pile = entier positif :
  - Représente l'indice de la prochaine « *case* » libre.
  - ⇒ Représente le nombre d'éléments présents dans la pile.
- On ajoute et on retire toujours « *par le haut* ».

Pile : implantation en C

```c
bool pop (struct stack_t *st, value_t *res) {
  if ((st != NULL) && (st->sp > 0)) {
    st->sp-- ;
    *res = st->storage[st->sp] ;
    return true ;
  }
  else return false ;
}

bool push (struct stack_t *st, value_t val) {
  if ((st != NULL) && (st->sp < MAX_STACK_SIZE)) {
    st->storage[st->sp] = val ;
    st->sp++ ;
    return true ;
  }
  else return false ;
}
```

Pile : implantation en C avec un tableau dynamique

- Taille maximale n'est plus une constante
  - ⇒ Appartient à la structure de pile.
  - « *Tableau dynamique* » ⇒ pointeur.

```
struct stack_t {
  unsigned int size ;  /* Taille de la pile. */
  int *storage ;       /* Mémoire de pile. */
  unsigned int sp ;    /* Pointeur de pile. */
};
```

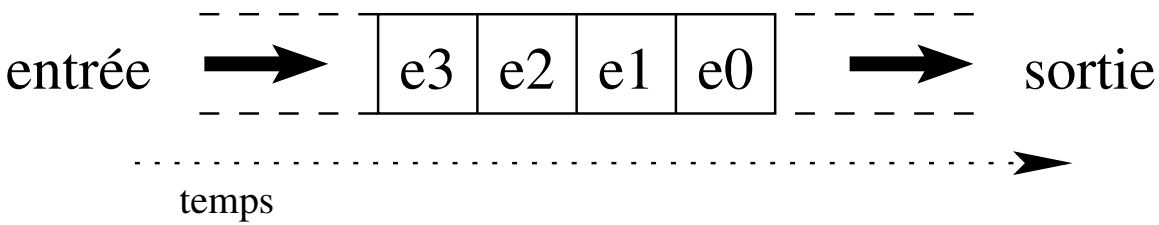
- Seul `push` change : en cas de pile pleine, on agrandit le tableau.

```
void push (struct stack_t *st, int val) {
  if (st == NULL) return false ;
  if (st->sp == st->size) {
    st->size *= 2 ;
    st->storage = realloc (st->storage, st->size) ;
  }
  st->data[st->sp] = val ;
  st->sp++ ;
}
```

Files

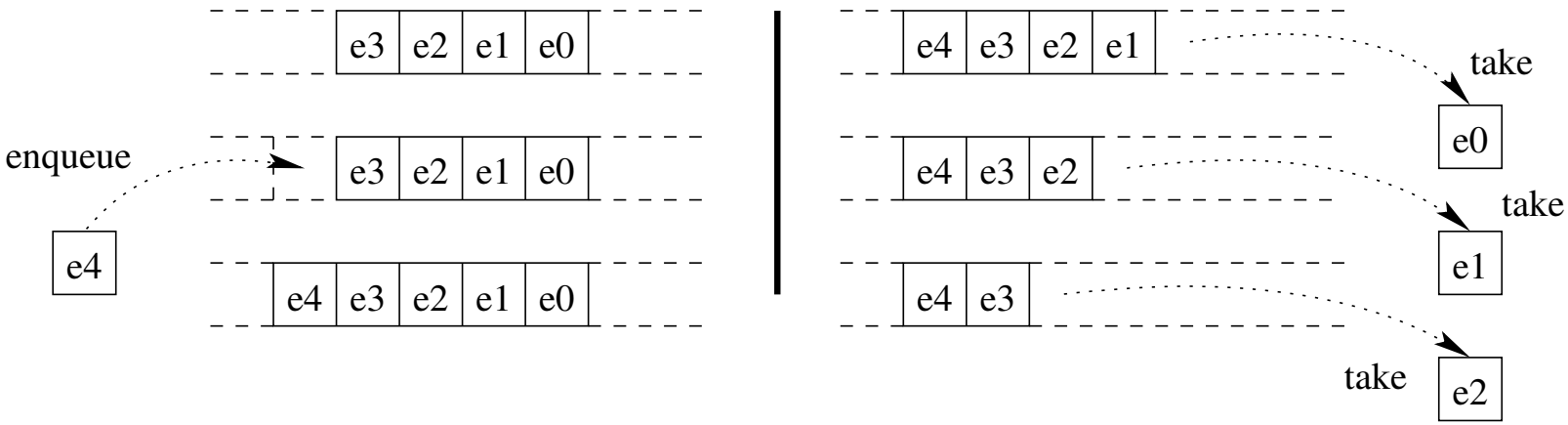
« Quand utiliser une file ? »

- Stocker des choses à traiter dans l'ordre d'arrivée (≈ file d'attente).
- ⇒ Premier inséré en queue → premier retiré en tête.
- En Anglais : FIFO (« **F***irst* **I***n* **F***irst* **O***ut* »).
- On peut consulter les éléments dans la file mais pas les retirer tant que ceux de devant sont dans la file.



Opérations sur une file

- **Ajouter** (« *enqueue* ») : Ajouter un élément en queue.
  - `enqueue` : (elem × file) → file
- **Retirer** (« *take* ») : Retirer l'élément de tête.
  - `take` : file → (elem × file)
- **Consulter** (« *peek* ») : Accéder à un élément sans le retirer.
  - `peek` : (file × int) → elem
- On peut **tester** si la file est **vide**.



File : implantation avec un tableau de taille fixe

- Utilisation d'un tableau avec taille bornée.
  - Comme les piles : on peut utiliser un tableau dynamique.
- Ajout des éléments en fin du tableau.
- Retrait des éléments en début de tableau.
- **Mais** on ne veut pas décaler tout le tableau (impossible en $\theta(1)$).
  - $\Rightarrow$ Utilisation du tableau de manière cyclique.
  - Arrivé au bout du tableau, on recommence au début.
- Besoin de garder l'indice du 1er élément (le + ancien encore vivant).

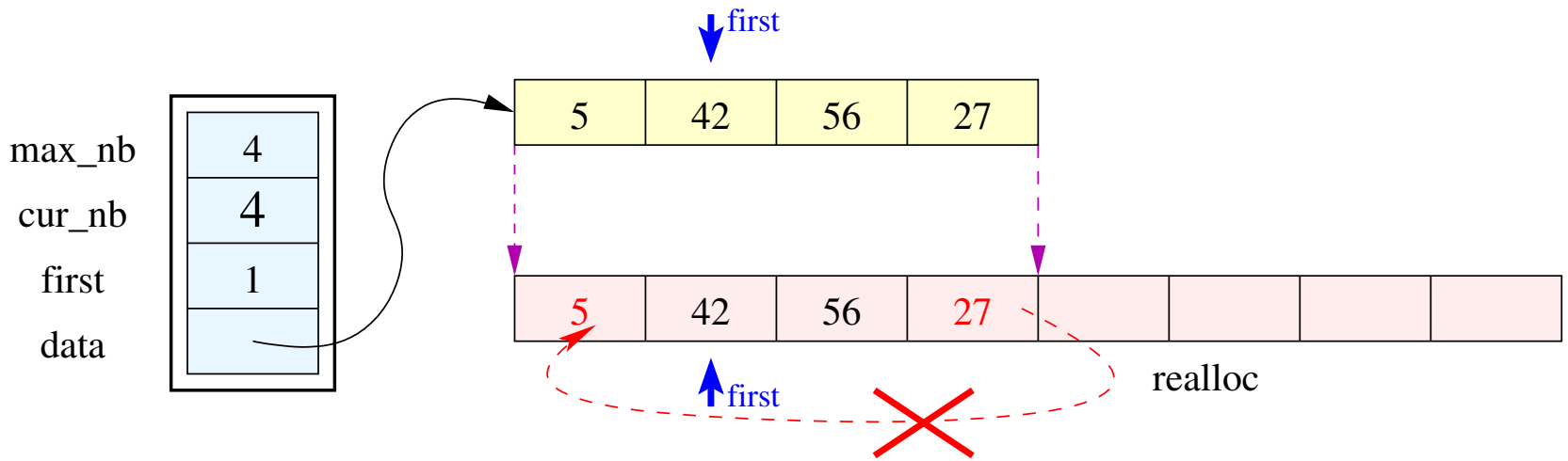
```
typedef int value_t ;     /* Infos mémorisées dans la file. */

/* Taille maximale de la pile. */
#define MAX_QUEUE_SIZE (6)

struct queue_t {
  unsigned int max_nb ;       /* Nombre maximal d'éléments. */
  unsigned int cur_nb ;       /* Nombre actuel d'éléments. */
  unsigned int first ;        /* Indice du premier élément. */
  value_t storage[MAX_QUEUE_SIZE] ;
};
```

File : implantation en C

```c
bool take (struct queue_t *q, value_t *res) {
  if (q == NULL || q->cur_nb == 0) return false ;
  *res = q->storage[q->first] ;
  q->first = (q->first + 1) % q->max_nb ;
  q->cur_nb-- ;
  return true ;
}

bool enqueue (struct queue_t *q, value_t val)
{
  if (q == NULL || q->cur_nb == q->max_nb) return false ;
  q->storage[(q->first + q->cur_nb) % q->max_nb] = val ;
  q->cur_nb++ ;
  return true ;
}
```

Utiliser l'aspect dynamique pour éviter la file pleine (1)

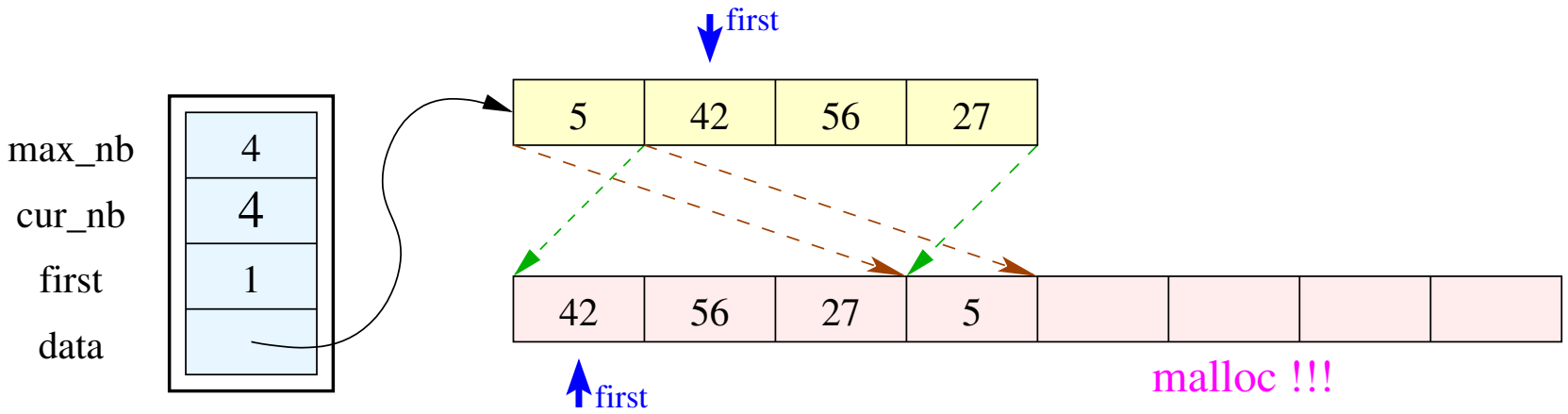
- Traitement à intercaler en cas de file pleine (cur_nb == max_nb).
- Plus compliqué que pour les piles : `realloc` ne convient pas.



- 5 ne suit pas 27 puisque espace libre après 27 !

Utiliser l'aspect dynamique pour éviter la file pleine (2)

- Idée :
  - Recopier depuis *first* → fin du tableau en début de nouveau tableau.
  - Recopier depuis 0 → *first* − 1 à la suite du nouveau tableau.
  - Remettre `first` au début du nouveau tableau (→ 0).



- Plus de `realloc` ⇒ gérer la libération de mémoire.

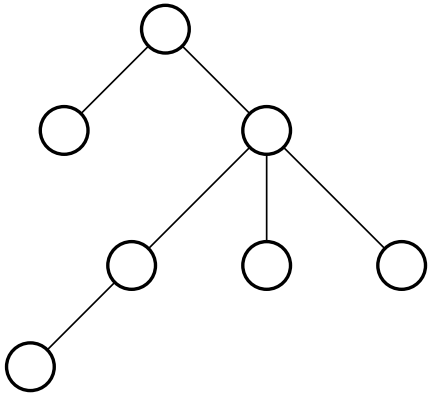
Utiliser l'aspect dynamique pour éviter la file pleine (3)

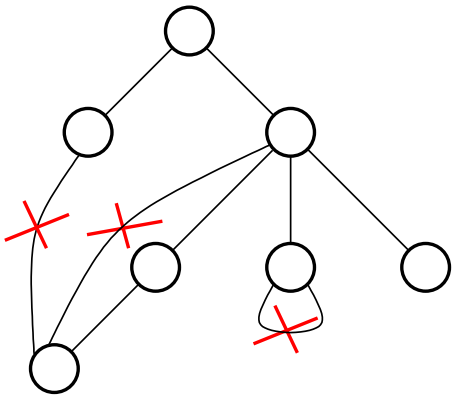
- memcpy (dest, src, size) : copie mémoire-mémoire.
- Nécessite #**include** <string.h>

```
bool enqueue (struct queue_t *q, value_t val) {
  if (q == NULL) return false ;
  if (q->cur_nb == q->max_nb) {
    value_t *new_storage = malloc (q->max_nb * 2 * sizeof (value_t)) ;
    memcpy
      (new_storage, &(q->storage[q->first]),
       (q->max_nb - q->first) * sizeof (value_t)) ;
    memcpy
      (&(new_storage[q->max_nb - q->first]), q->storage,
       q->first * sizeof (value_t)) ;
    free (q->storage) ;          /* Libération de l'ancien stockage. */
    q->storage = new_storage ;
    q->first = 0 ;
    q->max_nb *= 2 ;
  }
  q->storage[(q->first + q->cur_nb) % q->max_nb] = val ;
  q->cur_nb++ ;
  return true ;
}
```

Les arbres

Arbre

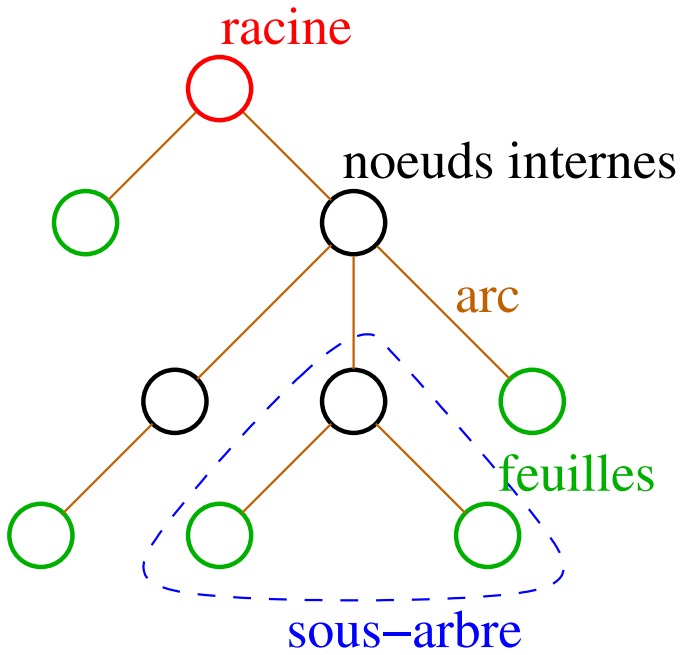


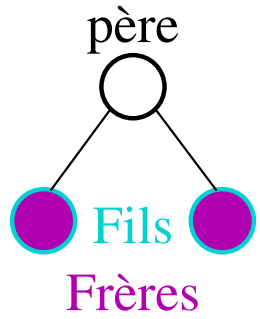


- Structure arborescente.
- Des nœuds, des arcs.
- Pas de « *boucle* ».
- Pas de « *raccourci* ».

Vocabulaire (1)

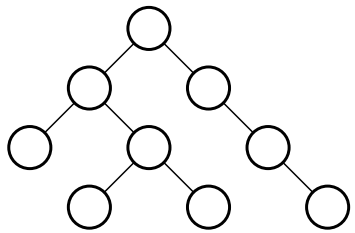
- Nœud (« *node* ») :
  - Racine (« *root* »)
  - Nœud interne
  - Feuille (« *leaf* »)
- Arc (« *edge* »)
- Sous-arbre (« *subtree* »)
- Parenté :
  - Père (« *parent* »)
  - Fils (« *child* »)
  - Frère (« *sibling* »)



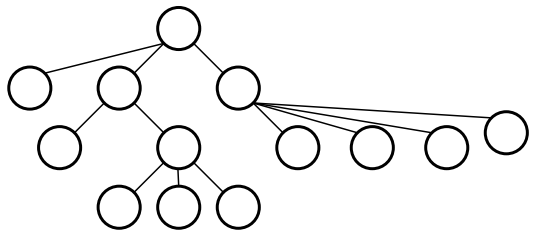


Vocabulaire (2)

- Degré d'un nœud : nombre de fils.
- Arité d'un arbre : nombre max de fils.
  - Certains nœuds peuvent en avoir moins.
- Arbre binaire : arité 2.

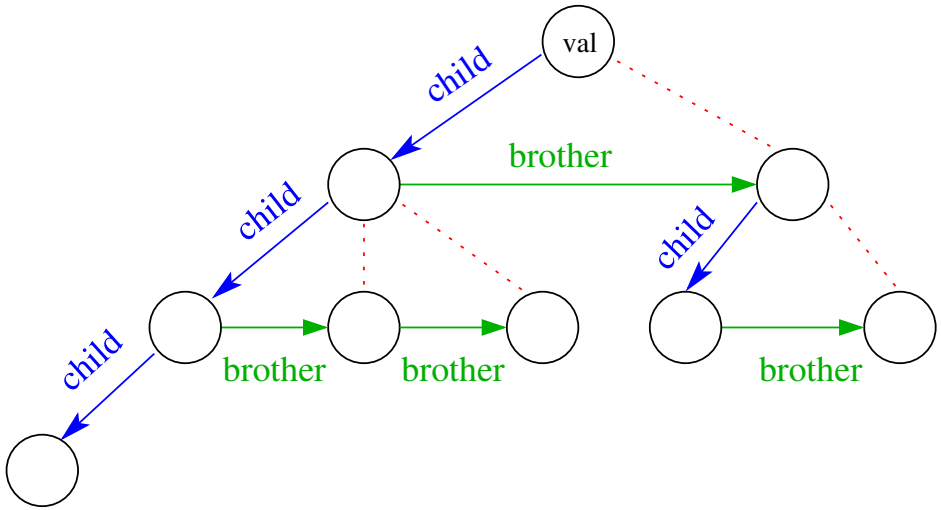


- Sinon, arbre n-aire.



Implantation par pointeurs : *Left-Child Right-Sibling*

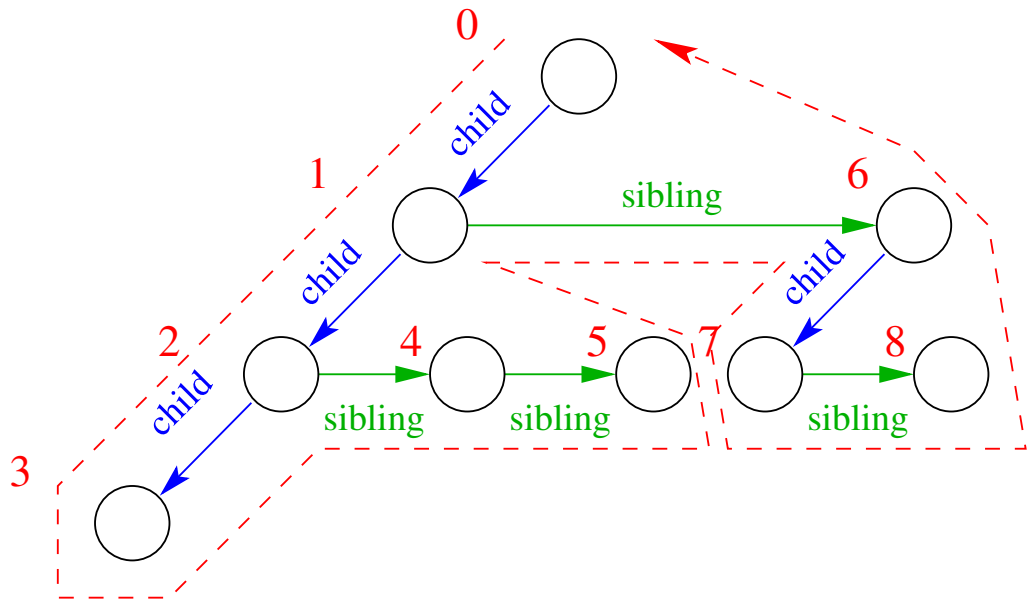
- Cas général : nombre quelconque de fils :
  - Liste chaînée de tous les fils (frères).
  - Le parent a un pointeur sur son 1er fils.




```
struct node_t {
  value_t data ;
  struct node_t *child ;
  struct node_t *sibling ;
};
```

Parcours en profondeur

- En Anglais : « *Depth First Search* » (DFS).
- Descente au plus profond en commençant par le fils gauche (ou droit).
- Parcours récursif (exemple parcours gauche-droite) :
  - À chaque nœud, on applique la descente sur le fils le plus à gauche.
  - Une fois le sous-arbre gauche exploré, on explore les fils à droite.
  - Une fois les sous-arbres droites explorés, on remonte et applique le parcours.
  - Fin : fin de l'exploration du sous-arbre le plus à droite.

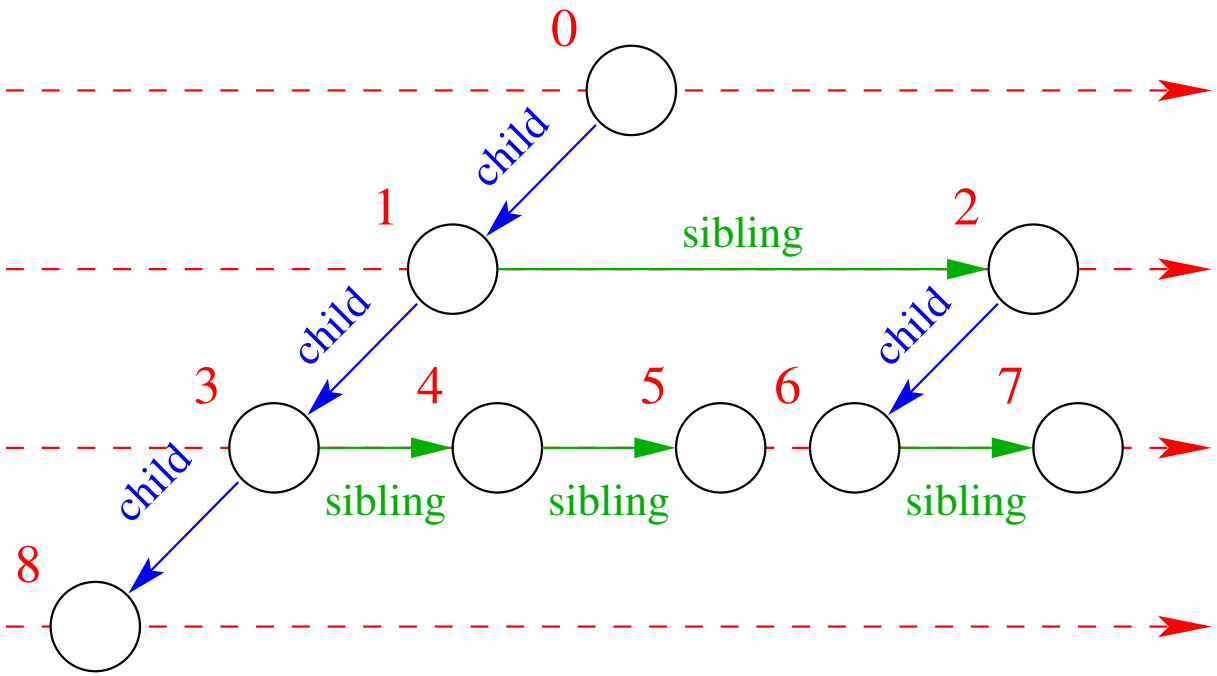


Parcours en profondeur : implantation

```c
void dfs (struct node_t *n) {
  if (n != NULL) {
    dfs (n->child) ;
    struct node_t *s = n->sibling ;
    while (s != NULL) {
      dfs (s) ;
      s = s->sibling ;
    }
  }
}
```

Parcours en largeur

- En Anglais : « *Breadth First Search* » (BFS).
- Parcours « *par niveaux* » (croissants).
- On visite d'abord tous les nœuds de même profondeur.
- Utilisation d'une file :
  - On extrait un nœud, on le traite, on insère tous ses fils.



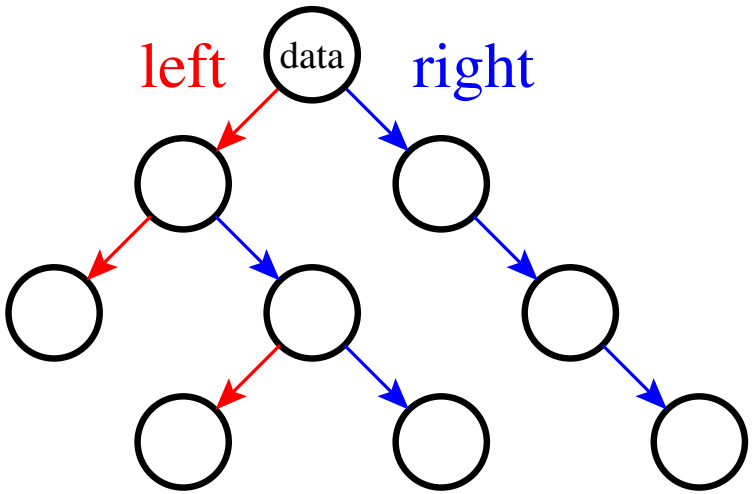
Parcours en largeur : implantation

```c
struct queue_t { ... } ;
bool take (struct queue_t *q, struct node_t **n) ;
bool enqueue (struct queue_t *q, struct node_t *n) ;

void bfs (struct node_t *n, struct queue_t *q) {
  if (n == NULL || q == NULL) return ;
  enqueue (q, n) ;
  struct node_t *curr_n ;
  while (take (q, &curr_n)) {
    struct node_t *s = curr_n->sibling ;
    while (s != NULL) {
      enqueue (q, s) ;
      s = s->sibling ;
    }
  }
}
```

Arbres binaires

Arbres binaires

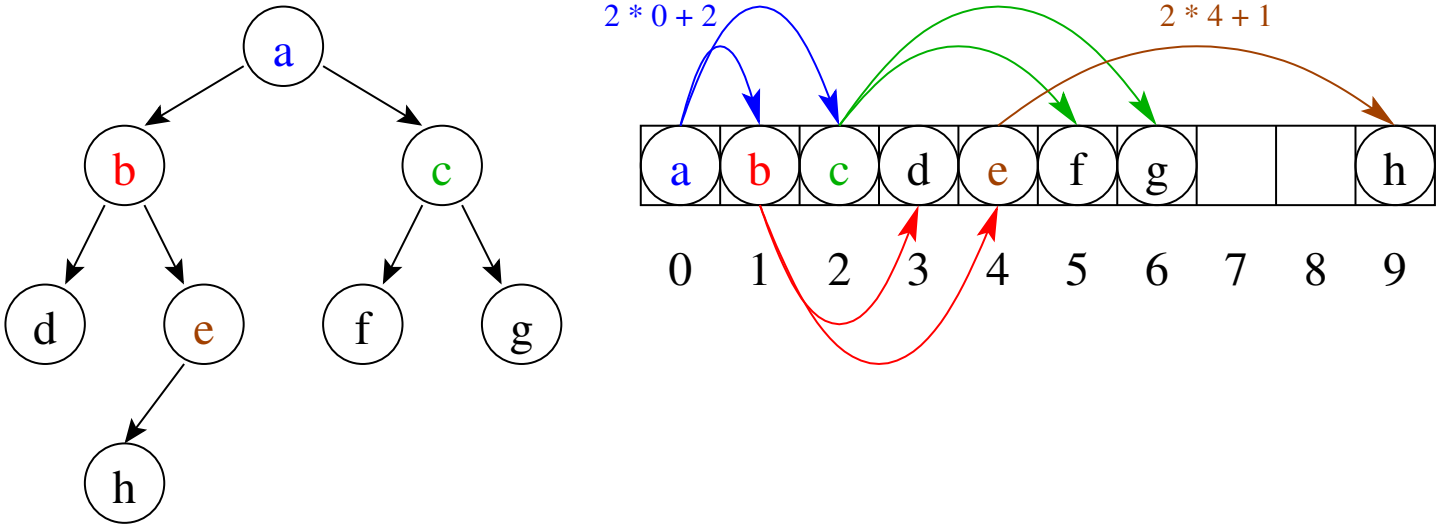
- Un nœud a au plus 2 fils :
  - 1 fils « *droit* »
  - 1 fils « *gauche* »




```
struct node_t {
  value_t data ;
  struct node_t *left ;
  struct node_t *right ;
};
```

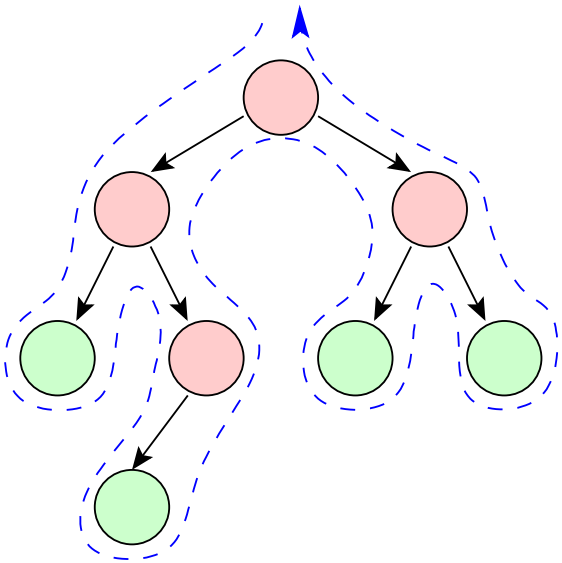
Arbres binaires : implantation par tableau

- Un nœud a un indice i.
- Ses fils sont aux indices $2i+1$ et $2i+2$.
- Son père est à l'indice $\lfloor((i-1)/2)\rfloor$.
- ⇒ Pas besoin de pointeurs.
- ⇒ Si arbre binaire complet, pas de perte de place.



Arbres binaires : parcours en profondeur

- Descente au plus profond en commençant par le fils gauche (ou droit).



```c
void dfs (struct node_t *n) {
  if (n != NULL) {
    dsf (n->left) ;
    dsf (n->right) ;
  }
}
```

- Ici, on a fait que parcourir, aucun traitement...

Arbres binaires : parcours en profondeur préfixe

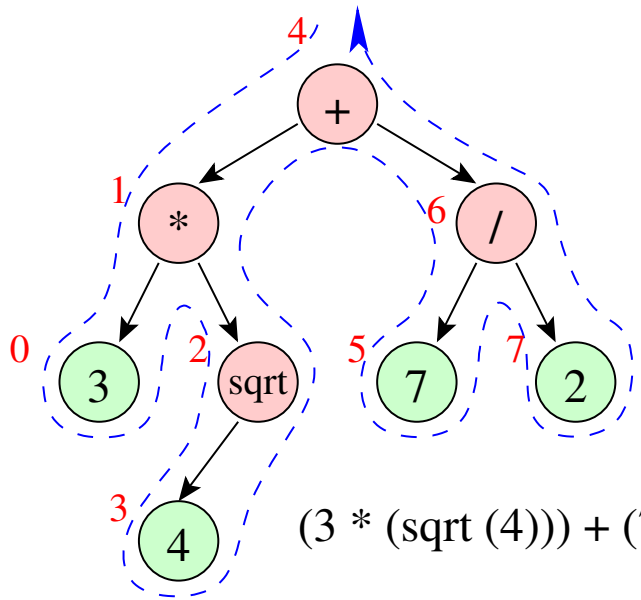
- 3 endroits pour insérer du traitement du nœud courant.
  - Au début.

```
void prefix (node_t *n) {
  if (n != NULL) {
    do_something (n->data) ;
    prefix (n->left) ;
    prefix (n->right) ;
  }
}
```

Arbres binaires : parcours en profondeur infixe

- 3 endroits pour insérer du traitement du nœud courant.
  - Au milieu.
  - Si `do_something` est l'impression : correspond à l'écriture *« normale »* d'une expression arithmétique.

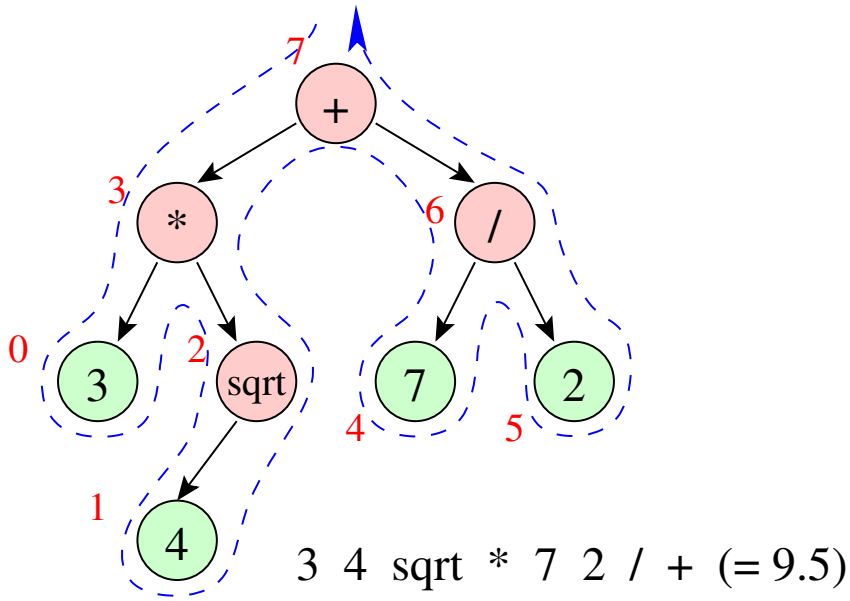
```
void infix (struct node_t *n) {
  if (n != NULL) {
    infix (n->left) ;
    do_something (n->data) ;
    infix (n->right) ;
  }
}
```



Arbres binaires : parcours en profondeur postfixe

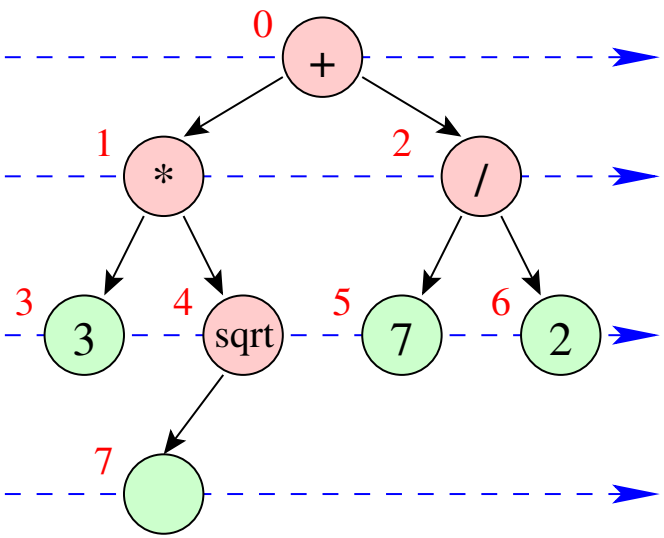
- 3 endroits pour insérer du traitement du nœud courant.
  - À la fin.
  - Si `do_something` est l'impression : correspond à l'écriture RPN (notation polonaise inverse).

```
void postfix (struct node_t *n) {
  if (n != NULL) {
    postfix (n->left) ;
    postfix (n->right) ;
    do_something (n->data) ;
  }
}
```



Arbres binaires : parcours en largeur

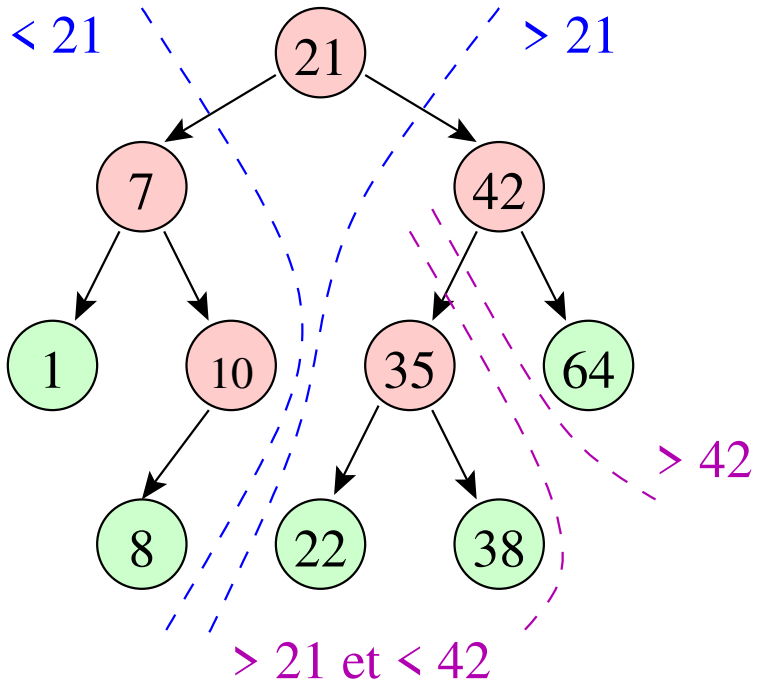
- On visite d'abord tous les nœuds de même profondeur.
- Utilisation d'une file :
  - On extrait un nœud, on le traite, on insère ses 2 fils.



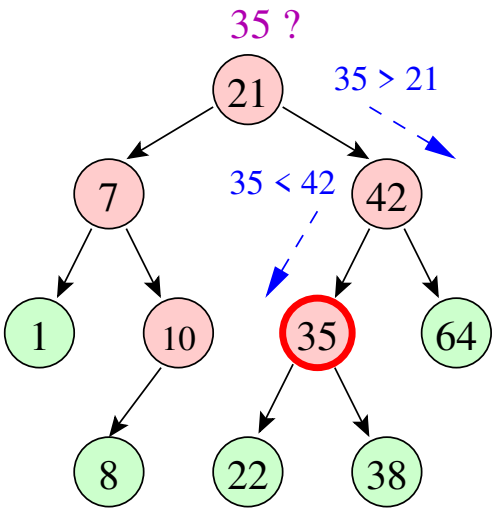

```
struct queue_t ;
bool take (struct queue_t *q, struct
     node_t **n) ;
bool enqueue (struct queue_t *q, struct
     node_t *n) ;

void bfs (struct node_t *n, struct
     queue_t *q) {
  if (n == NULL || q == NULL) return ;
  enqueue (q, n) ;
  struct node_t *curr_n ;
  while (take (q, &curr_n)) {
    do_something (curr_n->data);
    if (curr_n->left != NULL)
      enqueue (curr_n->left) ;
    if (curr_n->right != NULL)
      enqueue (curr_n->right) ;
   }
}
```

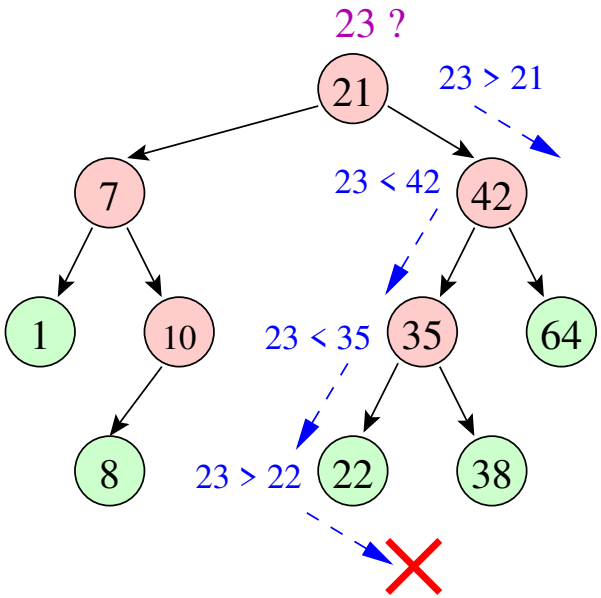
Arbres binaires de recherche

- Structure d'arbre binaire telle que :
  - Chaque nœud contient une clef.
  - La clef d'un nœud est $>$ à celle de son fils gauche.
    - Inductivement $>$ à celles du sous-arbre gauche.
  - La clef d'un nœud est $<$ à celle de son fils droit.
    - Inductivement $<$ à celles du sous-arbre droite.



Arbres binaires de recherche : recherche

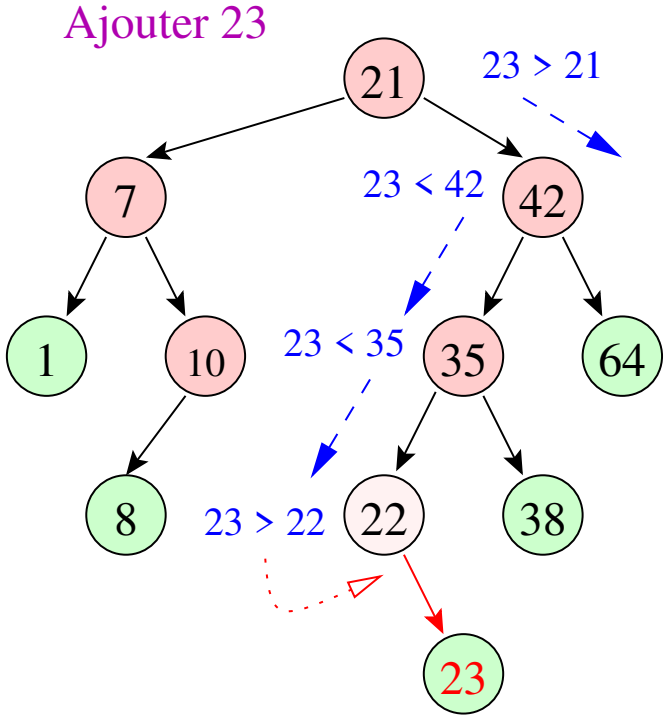





```
bool find (struct node_t *n, value_t
    val) {
  if (n == NULL) return false ;
  if (n->data == val) return true ;
  if (val < n->data)
    return find (n->left, val) ;
  return find (n->right, val) ;
}

bool find_iter (struct node_t *n,
    value_t val) {
  while (n != NULL) {
    if (n->data == val) return true ;
    if (val < n->data) n = n->left ;
    else n = n->right ;
  }
  return false ;
}
```

Arbres binaires de recherche : insertion (version naïve)

- Rechercher la position du nœud dans l'arbre.
- L'ajouter comme fil du dernier nœud rencontré.



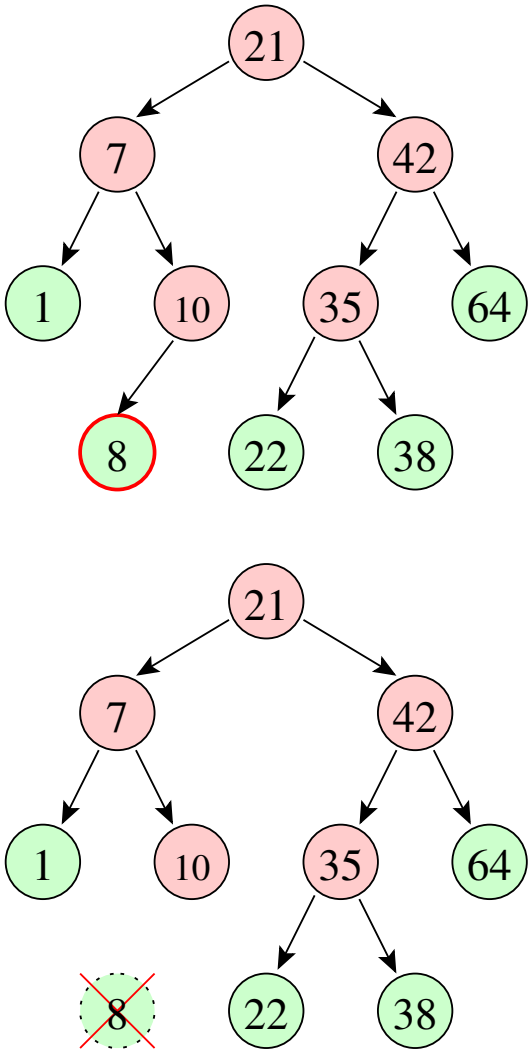

```
struct node_t* insert (struct node_t *n,
     value_t val) {
  if (n == NULL) {
    struct node_t *nn =
      malloc (sizeof (struct node_t)) ;
    if (nn == NULL) return NULL ;
    nn->data = val ;
    nn->left = NULL ;
    nn->right = NULL ;
    return nn ;
  }
  if (val < n->data)
    n->left = insert (n->left, val) ;
  else if (val > n->data)
    n->right = insert (n->right, val) ;
  return n ;
}
```

Arbres binaires de recherche : suppression

- On commence par rechercher le nœud à supprimer dans l'arbre.
- 3 cas dont 2 simples :
  - Suppression d'une feuille (→ facile).
  - Suppression d'un nœud avec 1 seul fils (→ facile).
  - Suppression d'un nœud avec 2 fils (→ plus compliqué).

```
struct node_t* __max_node_left_subtree (struct node_t *n) ;
struct node_t* __remove_root (struct node_t *n) ;

struct node_t* remove_node (struct node_t *n, value_t val) {
  if (n == NULL) return NULL ;
  if (n->data == val) return __remove_root (n) ;
  if (val < n->data) n->left = remove_node (n->left, val) ;
  else n->right = remove_node (n->right, val) ;
  return n ;
}
```

ABR, suppression : 1 feuille

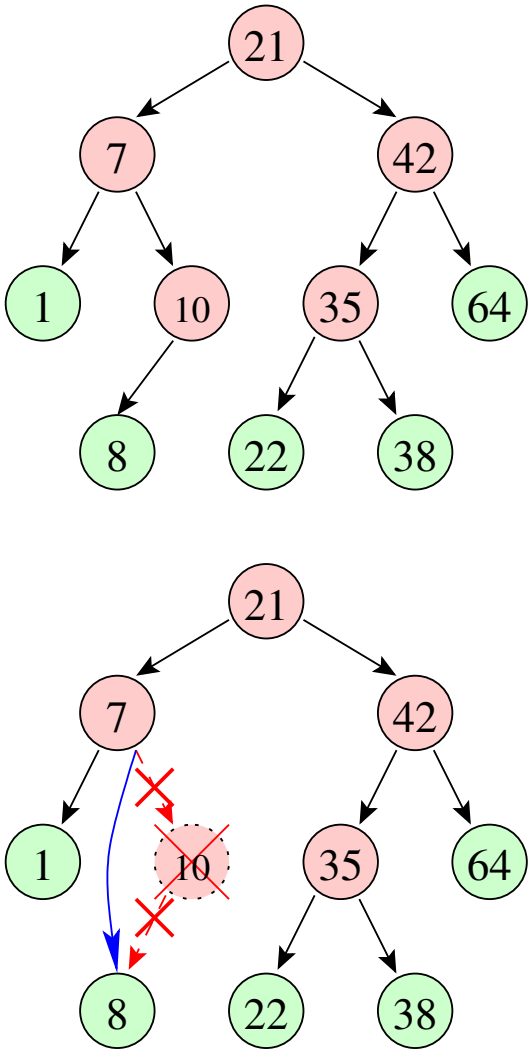


Suppression du nœud 8 ...

- Mettre le pointeur droit du nœud 10 à `NULL`.
- Libérer la mémoire occupée par le nœud 8.

```
struct node_t* __remove_root (struct node_t *n) {
  if (n->left == NULL) {
    struct node_t *ret_n = n->right ;
    free (n) ;
    return ret_n ;
  }
  if (n->right == NULL) {
    struct node_t *ret_n = n->left ;
    free (n) ;
    return ret_n ;
  }
  struct node_t* child =
     __max_node_left_subtree (n->left) ;
  n->data = child->data ;
  n->left = remove_node (n->left, child->data) ;
  return n ;
}
```

ABR, suppression : 1 seul fils

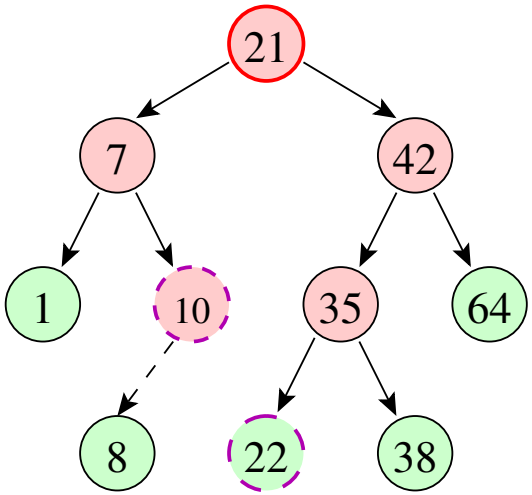


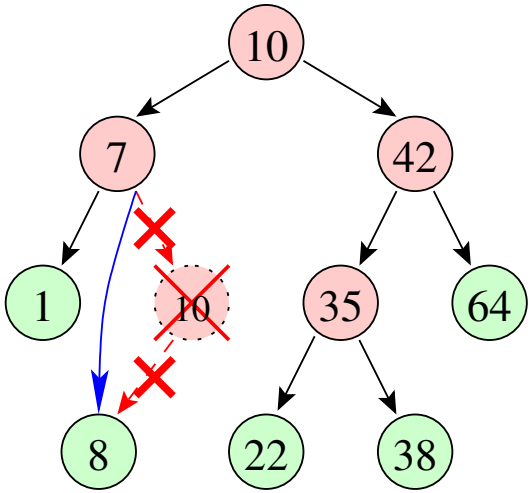
Suppression du nœud 10 ...

- Faire pointer son père (7) vers son fils (8).
- Libérer la mémoire occupée par le nœud 10.

```
struct node_t* __remove_root (struct node_t *n) {
  if (n->left == NULL) {
    struct node_t *ret_n = n->right ;
    free (n) ;
    return ret_n ;
  }
  if (n->right == NULL) {
    struct node_t *ret_n = n->left ;
    free (n) ;
    return ret_n ;
  }
  struct node_t* child =
     __max_node_left_subtree (n->left) ;
  n->data = child->data ;
  n->left = remove_node (n->left, child->data) ;
  return n ;
}
```

ABR, suppression : 2 fils





Suppression du nœud 21 . . .

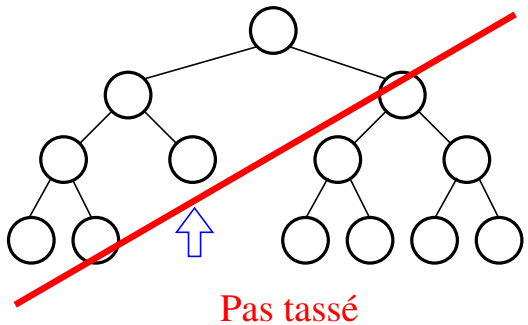
- Remplacement par le min du sous-arbre droit ou le max du sous-arbre gauche.
  - Successeur ou prédécesseur le plus proche.
- Choisir un candidat.
- On recopie sa clef et ses données pour remplacer le nœud 21.
- Ensuite, supprimer le nœud 10
  - 10 étant « le max », il n'a pas de fils droit,
  - ⇒ tombe dans les 2 cas précédents.

```
struct node_t* __remove_root (struct node_t *n) {
  if (n->left == NULL) ...
  if (n->right == NULL) ...
  struct node_t* child =
      __max_node_left_subtree (n->left) ;
  n->data = child->data ;
  n->left = remove_node (n->left, child->data) ;
  return n ;
}
```

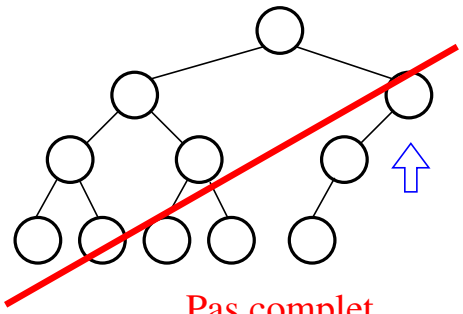
Les files de priorité

Rappels : arbres binaires et tas

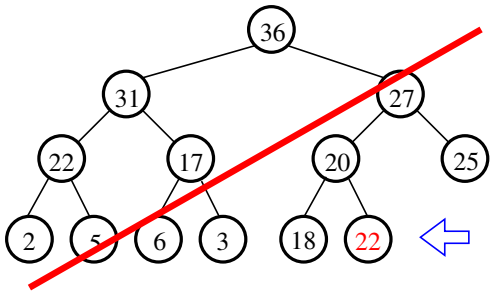
- Arbre binaire « tassé » si :
  - tous les niveaux (sauf éventuellement le dernier) sont complets
  - et le dernier niveau est rempli à gauche.
- Tas : arbre binaire « tassé »
  - dont les nœuds sont étiquetés par des clefs,
  - tout noeud possède une clef supérieure ou égale aux clefs de ses fils (i.e. des nœuds de ses sous-arbres gauche et droit)



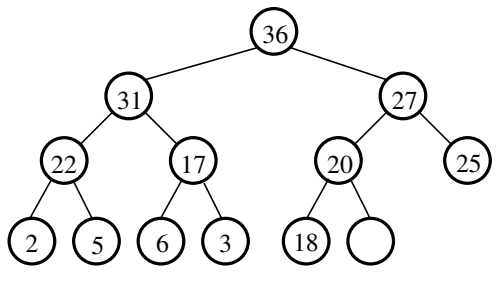
Pas tassé



Pas complet



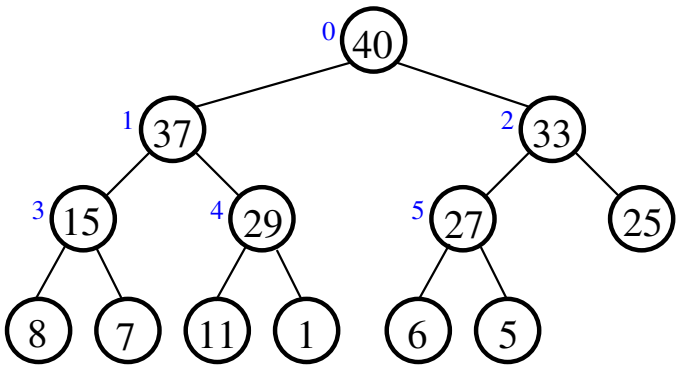
Mal ordonné



Tas

Représentation d'un tas

- Tableau : comme vu pour les arbres binaires.
- Fils gauche et droit aux indices $2i+1$ et $2i+2$.
- Père à l'indice $\lfloor((i-1)/2)\rfloor$.
- Tableau sans « trous ».

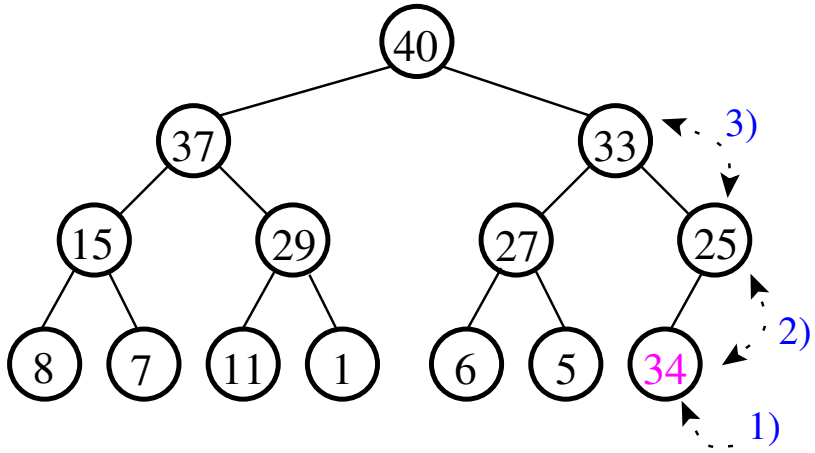


40	37	33	15	29	27	25	8	7	11	1	6	5		
0	1	2	3	4	5									14

```
struct prio_t {
  unsigned int max_size ;
  unsigned int curr_size ;
  value_t *storage ;
};
```

Insertion

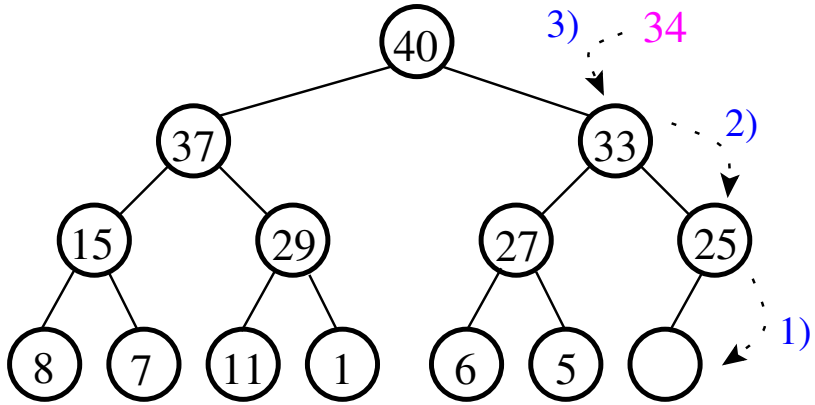
- Placer le noeud à la dernière position libre du tableau.
- Tant que parent plus petit, remonter en permutant avec le paren




```
bool insert (struct prio_t* q, value_t val) {
  if (q == NULL || q->curr_size == q->max_size) return false ;
  value_t *storage = q->storage ;
  storage[q->curr_size] = val ;
  int i = q->curr_size ;
  while (i > 0 && storage[parent_index (i)] <= storage[i]) {
    value_t tmp = storage[i] ;
    storage[i] = storage[parent_index (i)] ;
    storage[parent_index (i)] = tmp ;
    i = parent_index (i) ;
  }
  q->curr_size++ ;
  return true ;
}
```

Insertion : autre solution

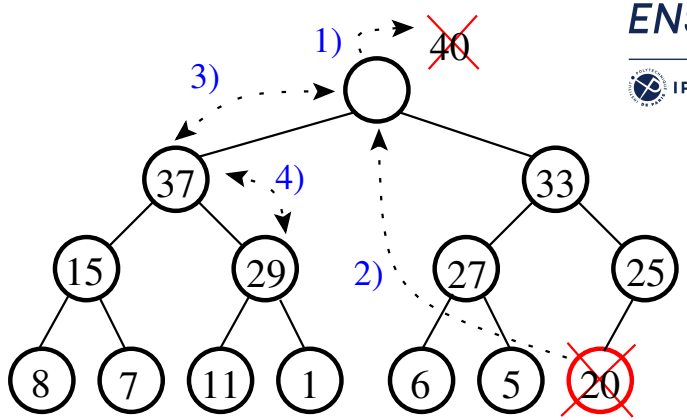
- Partir de la dernière case occupée du tableau.
- En remontant, faire redescendre les parents ≤ la priorité à insérer
- Quand on trouve un parent >, insérer la valeur au nœud courant.




```
bool insert (struct prio_t *q, value_t val) {
  if (q == NULL || q->curr_size == q->max_size) return false ;

  int i = q->curr_size ;
  q->curr_size++ ;
  value_t *storage = q->storage ;
  while (i > 0 && storage[parent_index (i)] <= val) {
    storage[i] = storage[parent_index (i)] ;
    i = parent_index (i) ;
  }
  storage[i] = val ;
  return true ;
}
```

Extraction

- Copier le dernier nœud dans la racine.
- Faire redescendre la valeur de la racine en permutant avec le plus grand des fils.
- Arrêter quand le plus grand des fils est ≤ priorité insérée.

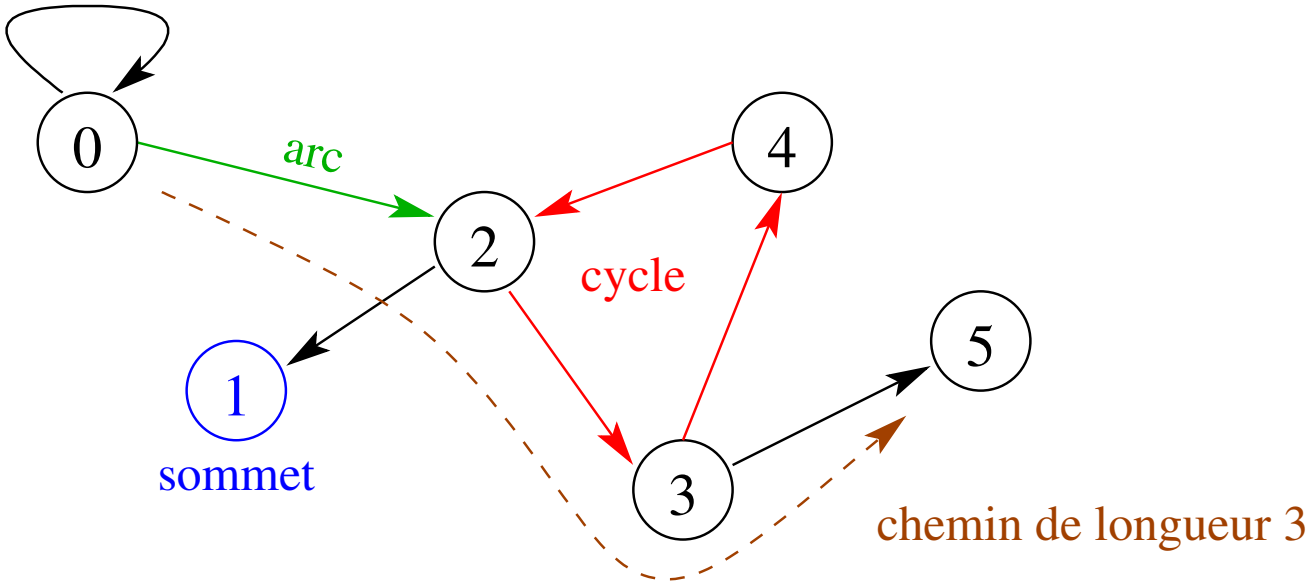



```
bool extract (struct prio_t* q, value_t *ret_val) {
  if (q == NULL || q->curr_size == 0) return false ;
  value_t *storage = q->storage ;
  *ret_val = storage[0] ;   /* Valeur retournée. */
  q->curr_size-- ;
  int v = storage[q->curr_size] ;
  storage[0] = v ;
  int i = 0 ;
  while (left_child_index (i) < q->curr_size) {
    int j = left_child_index (i) ;
    if (j + 1 < q->curr_size && storage[j + 1] > storage[j]) j++ ;
    if (v >= storage[j]) break ;
    storage[i] = storage[j] ;
    i = j ;
  }
  storage[i] = v ;
  return true ;
}
```

Les graphes

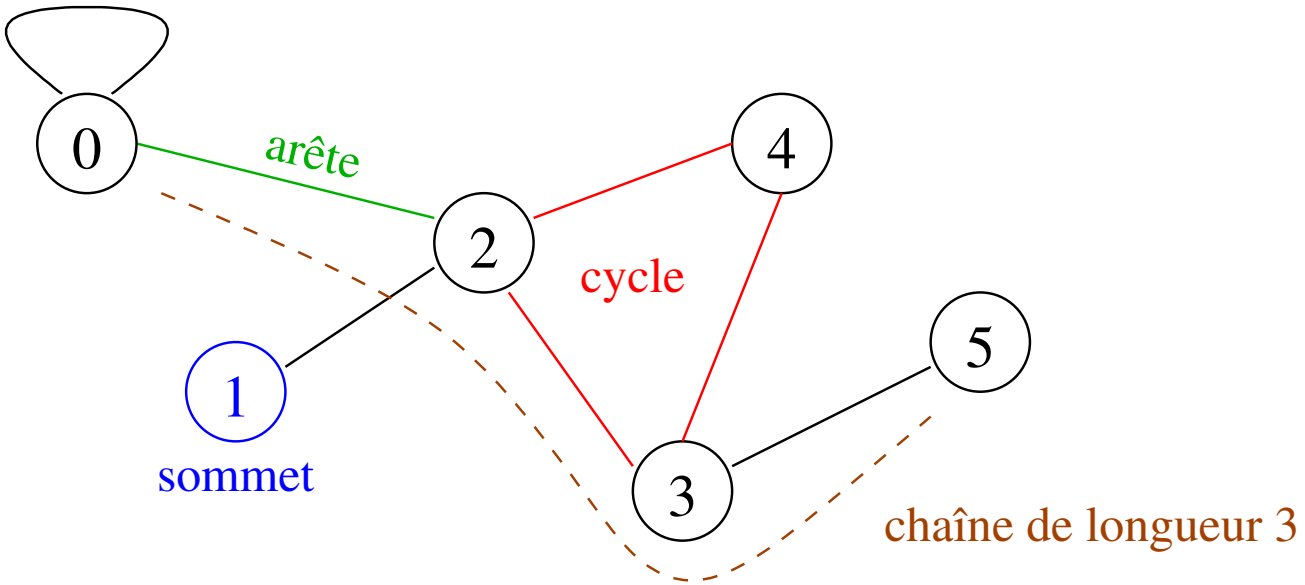
Définitions : graphe orienté

- Graphe orienté : couple (S, A)
  - S : ensemble de sommets (« *vertex/vertices* »).
  - A : ensemble d'arcs (« *edge* »), sous-ensemble de $S \times S$.
- Chemin : suite d'arcs consécutifs (« *path* »).
- Longueur d'un chemin : nombre d'arcs sur ce chemin.
- Chemin simple : chemin où aucun arc n'est parcouru plusieurs fois.
- Cycle : chemin qui boucle.



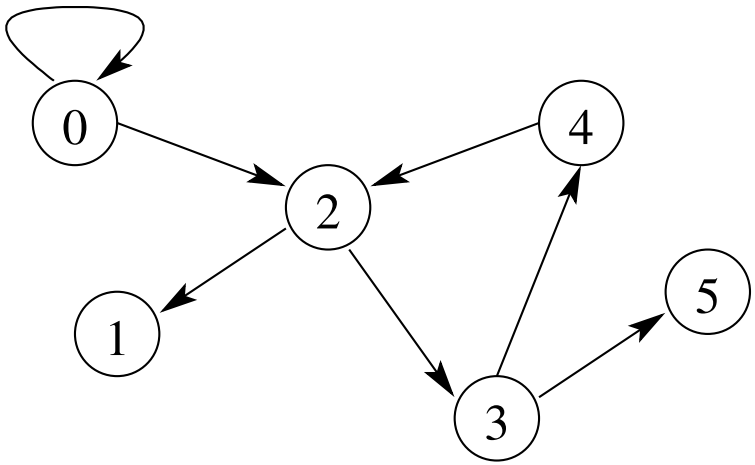
Définitions : graphe non-orienté

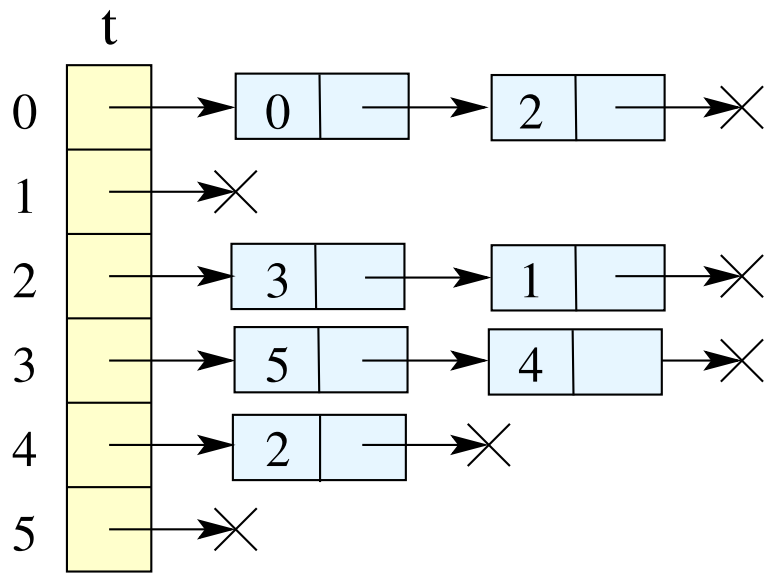
- Graphe non-orienté : comme un graphe orienté . . . sans orientation sur les « *liens* » entre sommets.
- Les « *arcs* » sont souvent appelés arêtes à la place.
- Les « *chemins* » sont souvent appelés chaînes à la place.



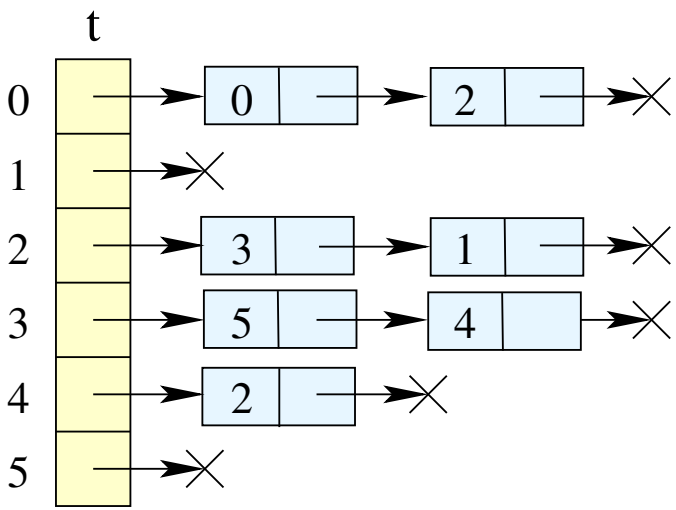
Représentation en machine : liste de successeurs (1)

- Méthodes par chaînage et manipulation de pointeurs.
- Plusieurs méthodes selon les besoins.
- Par exemple :
  - Tableau `t` des successeurs de chaque sommet.
  - $\Rightarrow$ `t[i]` pointe la liste chaînée des successeurs du sommet i.
  - Accès direct à un sommet via `t`.
  - Complexité spatiale optimale en $\theta(|S|+|A|)$.





Représentation en machine : liste de successeurs (2)

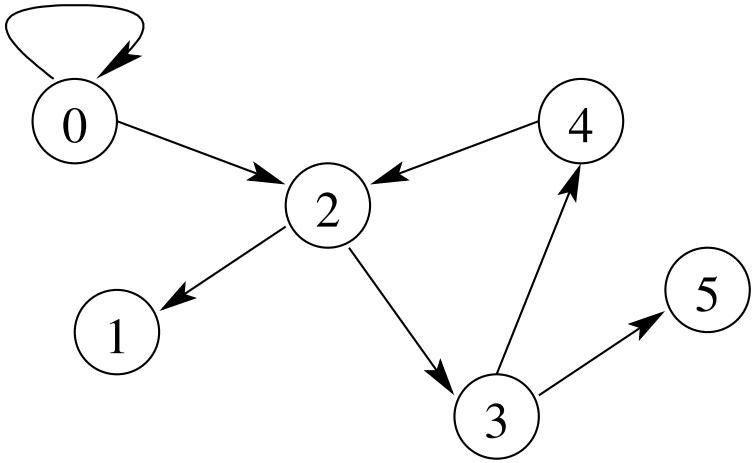


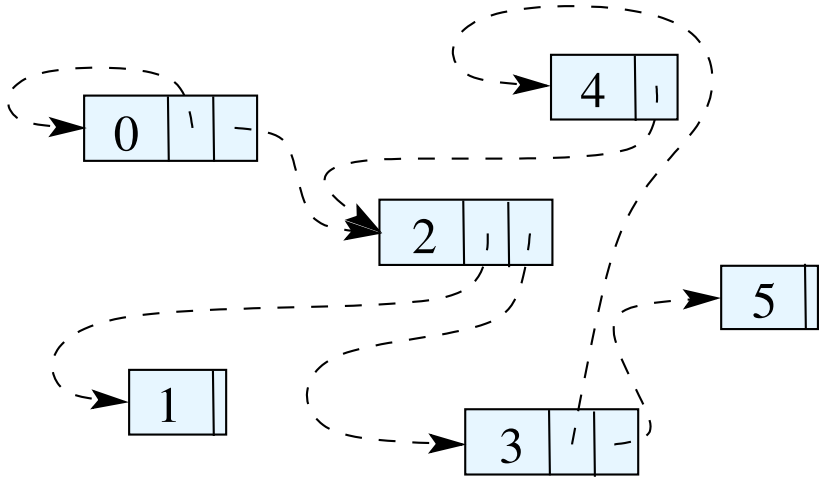

```c
typedef int edge_val ;
typedef int vertex_name_t ;

struct edge_list_t {
  int dest ;
  edge_val data ;
  struct edge_list_t *next ;
};

struct vertex_t {
  vertex_name_t name ;
  struct edge_list_t *edges ;
};

struct graph_t {
  int nb_vertices ;
  struct vertex_t *vertices[MAX_VERTICES] ;
};
```

- Remarque : ici, étiquette sur les arcs.
- Si nom $\neq$ index de tableau, besoin d'une fonction $nom \rightarrow index$.

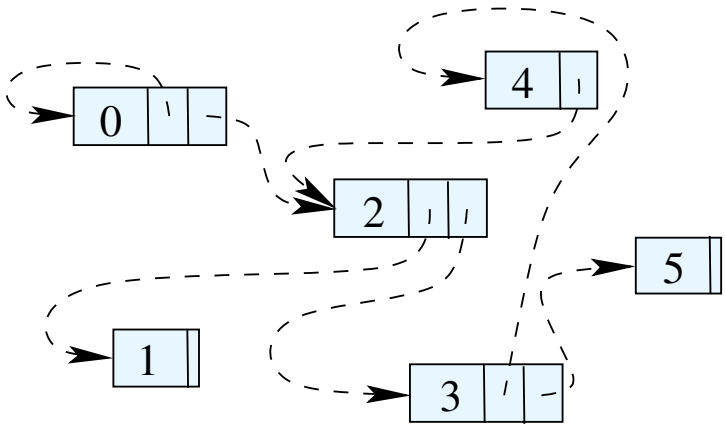
Représentation en machine : liste de successeurs (3)

- On peut aussi partager physiquement les sommets.
  - Chaque sommet est représenté 1 et 1 seule fois.
  - Chaque sommet a une liste dont les éléments pointent sur ses successeurs.





Représentation en machine : liste de successeurs (4)

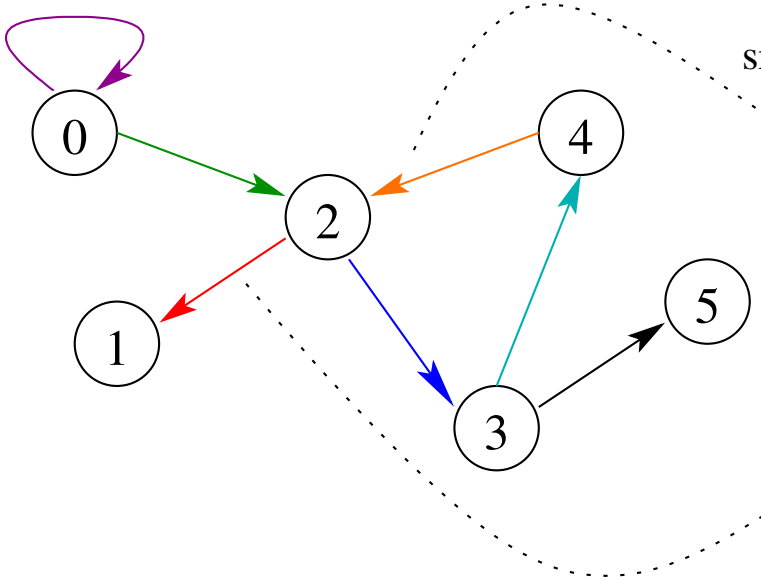



```
struct vertex_t {
  vertex_name_t name ;
  struct vertex_list_t *neighbours ;
  bool seen ;
};

struct vertex_list_t {
  struct vertex_t *vertex ;
  struct vertex_list_t *next ;
};

struct graph_t {
  int nb_vertices ;
  struct vertex_list_t *vertices ;
};
```

- Remarque : ici, pas d'étiquette sur les arcs.
- Marqueur intégré aux sommets pour les parcours (éviter de boucler).
- Marqueurs à réinitialiser entre chaque parcours.

Représentation en machine : matrice d'adjacence (1)

- Ensemble S des sommets indexés de 0 à $n-1$.
- Arcs $A \subset S \times S$.
- Matrice M de taille $n \times n$ telle que :
  - $M_{ij} = 1$ s'il existe un arc $i \to j$ (i.e. $(i,j) \in A$).
  - $M_{ij} = 0$ sinon.
- Complexité spatiale : $\Theta(n^2)$.

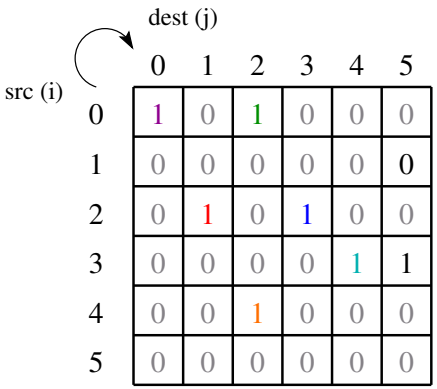


dest (j)

src (i)

	0	1	2	3	4	5
0	1	0	1	0	0	0
1	0	0	0	0	0	0
2	0	1	0	1	0	0
3	0	0	0	0	1	1
4	0	0	1	0	0	0
5	0	0	0	0	0	0

Représentation en machine : matrice d'adjacence (2)



- Tableau de taille fixe, 2 dimensions :
  - bool graph[NB_VERTICES][NB_VERTICES] ;
- Tableau dynamique, 2 dimensions :
  - bool **graph ;
- Tableau de taille fixe, 1 dimension :
  - bool graph[NB_VERTICES * NB_VERTICES] ;
  - Linéarisation des accès
    graph[i][j] → graph[i * NB_VERTICES + j]
- Tableau dynamique, 1 dimension :
  - bool *graph ;
  - Linéarisation des accès

Recouvrement de graphe par un arbre

- Un parcours sert parfois à produire un recouvrement d'un graphe.
- Arbre ⇒ chaque sommet a au plus 1 parent.
- Puisque c'est un arbre . . . pas de cycle.
- En quelque sorte, c'est le graphe « amputé » de certains arcs.
- But : « *simplifier* » un graphe
  - Garder seulement une information « *intéressante* ».

Parcours en largeur

- Équivalent du parcours par niveaux des arbres :
  - On part d'un sommet,
  - on visite tous les voisins,
  - on visite tous les voisins des voisins...

⇒ Sommets de distance d découverts avant ceux de distance $d+1$.

- Problèmes :
  - Ne pas boucler en visitant les cycles.
  - Ne pas visiter plusieurs fois le même sommet (en passant par des « raccourcis »).

⇒ Marquage des sommets lors du parcours :

- « *Blanc* » : non visité (ou « *non marqué* »).
- (« *Gris* » : en cours de visite, si besoin.)
- « *Noir* » : déjà visité (ou « *marqué* »).

Parcours en largeur : algorithme général

- Utilisation d'une file F :
  - contient initialement 1 seul sommet (racine).
- Complexité en $O(|A|)$ pour une représentation par listes.

```
Soit r la racine ;
Marquer r ;
Enfiler r dans la file F ;
Tant que F n'est pas vide {
  s = élément en tête de F ;
  Traiter (s) ;
  Pour chaque voisin v du sommet s {
    Si v n'est pas marqué alors {
      Marquer v ;
      Enfiler v dans la file F ;
    }
  }
}
```

Parcours en largeur : implantation

```
struct vertex_t {
  vertex_name_t name ;
  struct vertex_list_t *neighbours ;
  bool seen ;
};

struct vertex_list_t {
  struct vertex_t *vertex ;
  struct vertex_list_t *next ;
};

struct graph_t {
  int nb_vertices ;
  struct vertex_list_t *vertices ;
};

typedef struct vertex_t* value_t ;
struct queue_t {
  unsigned int max_nb ;
  unsigned int cur_nb ;
  unsigned int first ;
  value_t *storage ;
};

struct queue_t* empty_queue () ;
bool take (struct queue_t *q,
           value_t *res) ;
bool enqueue (struct queue_t *q,
              value_t val) ;
bool is_empty (struct queue_t *q) ;
```

```
void bfs (struct vertex_t *root) {
  struct queue_t *q = empty_queue () ;
  root->seen = true ;
  enqueue (q, root) ;
  while (! is_empty (q)) {
    struct vertex_t *n ;
    take (q, &n) ;
    printf ("%d\n", n->name) ;
    struct vertex_list_t *neighb =
      n->neighbours ;
    while (neighb != NULL) {
      if (neighb->vertex != NULL &&
          ! neighb->vertex->seen) {
        neighb->vertex->seen = true ;
        enqueue (q, neighb->vertex) ;
      }
      neighb = neighb->next ;
    }
  }
}
```

Parcours en profondeur : algorithme général

- Équivalent du parcours en profondeur des arbres :
  - On descend au plus profond en premier.
  - Algorithme naturellement récursif.
- Mêmes problèmes que pour le parcours en largeur ⇒ marquage.

```
DFS (sommet s) {
  Marquer s ;
  (Pré-traiter s ;) /* Selon le besoin. */
  Pour chaque voisin non marqué v du sommet s
    DFS (v) ;
  (Post-traiter s ;) /* Selon le besoin. */
  }
}
```

Parcours en profondeur : implantation (1)

```
struct edge_list_t {
  int dest ; edge_val data ; struct edge_list_t *next ;
};

struct vertex_t {
  vertex_name_t name ; struct edge_list_t *edges ;
};

struct graph_t {
  int nb_vertices ; struct vertex_t *vertices[MAX_VERTICES] ;
};

void __dfs (struct graph_t *g, int n_index, bool *seen) {
  seen[n_index] = true ;
  printf (" %d", g->vertices[n_index]->name) ;
  struct edge_list_t *e = g->vertices[n_index]->edges ;
  while (e != NULL) {
    if (! seen[e->dest]) {
      printf (" -%d->", e->data) ;
      __dfs (g, e->dest, seen) ;
    }
    e = e->next ;
  }
}

void dfs (struct graph_t *g, int root_index) {
  if (g->nb_vertices <= 0) return ;
  bool *seen = malloc (sizeof (bool) * g->nb_vertices) ;
  if (seen == NULL) return ;
  __dfs (g, root_index, seen) ;
}
```

Parcours en profondeur : implantation (2)

```
struct vertex_t {
  vertex_name_t name ; struct vertex_list_t *neighbours ; bool seen ;
};

struct vertex_list_t {
  struct vertex_t *vertex ; struct vertex_list_t *next ;
};

struct graph_t {
  int nb_vertices ; struct vertex_list_t *vertices ;
};

void dfs (struct vertex_t *v) {
  v->seen = true ;
  printf (" %d", v->name) ;
  struct vertex_list_t *neigh = v->neighbours ;
  while (neigh != NULL) {
    if (!neigh->vertex->seen) {
      printf (" ->") ;
      dfs (neigh->vertex) ;
    }
    neigh = neigh->next ;
  }
}
```

- Penser à effacer les marqueurs (champ `seen`).

Avez-vous des questions ?