IN 101 - Cours 09

18 novembre 2011

présenté par

Matthieu Finiasz

Structures dynamiques

Qu'est-ce qu'une structure dynamique

- Une structure dynamique est une structure :
  - destinée à stocker des données,
  - dont la taille (quantité de données) varie au cours du temps
    - → on veut pouvoir ajouter/enlever des éléments.

- Il existe de nombreux types de structures dynamiques utiles en algorithmiques
  - on va regarder les plus simples pour l'instant.

- Il faut distinguer :
  - la définitions de la structure
    - → ses propriétés algorithmiques,
  - l'implémentation de la structure
    - → la façon dont elle sera représentée en mémoire...

Les tableaux dynamiques

- Le tableau dynamique est la structure dynamique la plus simple
  - on veut les mêmes propriétés qu'un tableau, sans limite de taille,
    → accès direct, en $\Theta(1)$, au i-ème élément
  - naturellement, on l'implémente en C avec des tableaux...

- Il n'est pas toujours possible de rallonger un bloc de mémoire allouée
  - souvent la zone mémoire suivante est déjà occupée.
- Pour agrandir un tableau il faut donc :
  - allouer un nouveau bloc mémoire (plus grand),
  - recopier l'ancien tableau dedans,
  - libérer le bloc mémoire occupé par l'ancien tableau.

Les tableaux dynamiques

Implémentation en C

- Il est nécessaire de garder une trace de la taille du tableau :
  - on utilise une structure C pour cela,
  - il faut des fonctions spéciales pour lire et écrire.

```
1  typedef struct {
2    unsigned int len;
3    int* val;
4  } table;
5
6  table* table_init() {
7    table* new = (table*) malloc(sizeof(table));
8    new->len = 0;           // on choisit des tableaux de
9    new->val = NULL;        // taille 0 par défaut.
10   return new;
11 }
```

```
1  int table_read(table* tab, unsigned int pos) {
2    if (pos >= tab->len) {  // on évite les dépassements
3      printf("Erreur !\n");
4      return -1;
5    }
6    return tab->val[pos];   // comme dans un tableau normal
7  }
8
9  void table_write(table* tab, unsigned int pos, int val) {
10   unsigned int new_len;
11   if (pos >= tab->len) {  // si on dépasse
12     new_len = 2*len;      // on double la taille
13     if (pos >= new_len) {
14       new_len = pos+1;    // et plus si nécessaire
15     }
16     tab->val = realloc(tab->val, new_len*sizeof(int));
17     if (tab->val == NULL) {
18       printf("Erreur !\n");
19       return;
20     }
21   }
22   tab->val[pos] = val;
23 }
```

Les tableaux dynamiques
Implémentation en C

- On connait la taille courante du tableau dynamique :
  - on peut tester les dépassements,
  - on utilise `realloc` pour réallouer le bloc mémoire
    → exactement comme `malloc`, `memcpy`, puis `free`.

- Quand on réalloue, il est important de doubler au minimum :
  - réallouer coûte cher car il faut recopier,
  - en doublant, les cases sont recopiées en moyenne 1 fois
    → remplir un tableau de n valeurs coûte $\Theta(n)$.

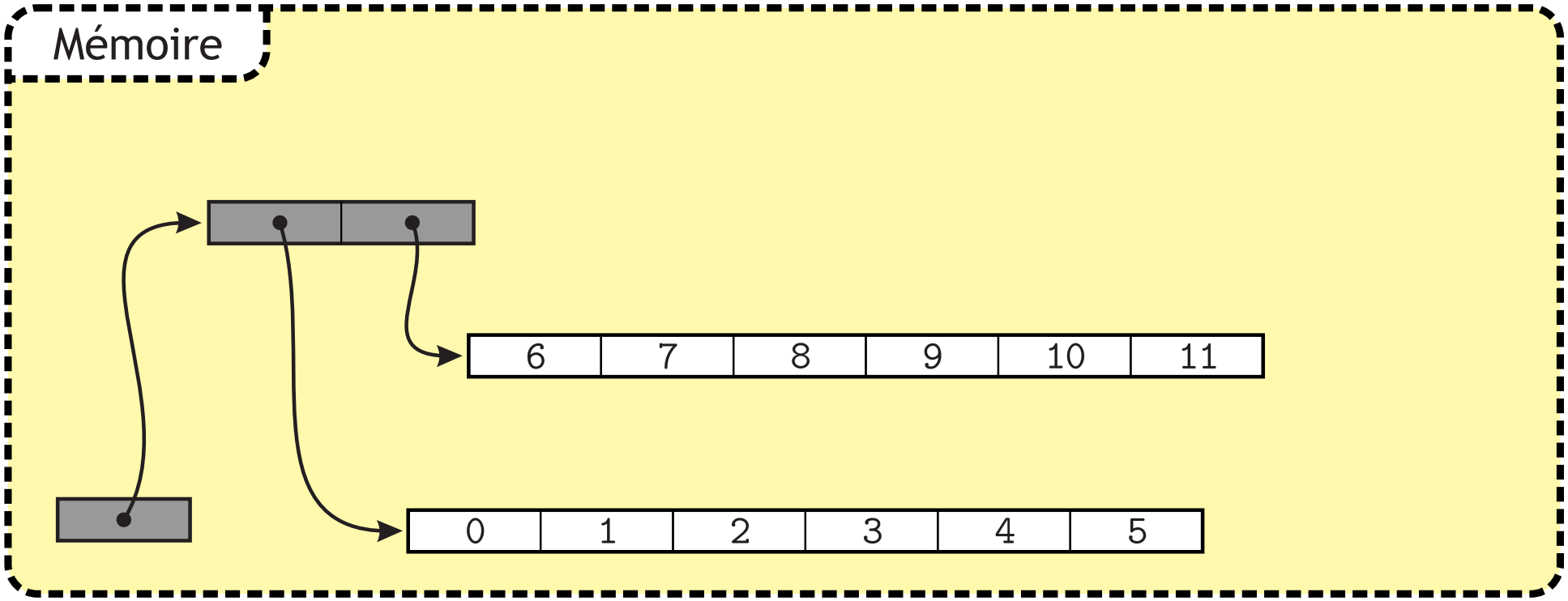
- Faire un test à chaque lecture/écriture est cher :
  - cela ne change pas la complexité, mais le temps d'exécution,
  - on peut faire sans, si on est certain de où on lit/écrit.

Tableaux dynamiques avec des tableaux à deux dimensions

- La technique précédente marche mal pour les très grands tableaux
  - difficile d'allouer des très grands blocs mémoire,
  - les recopies utilisent beaucoup de mémoire
    - → la somme des 2 tailles (avant et après),
  - les recopies sont chères.

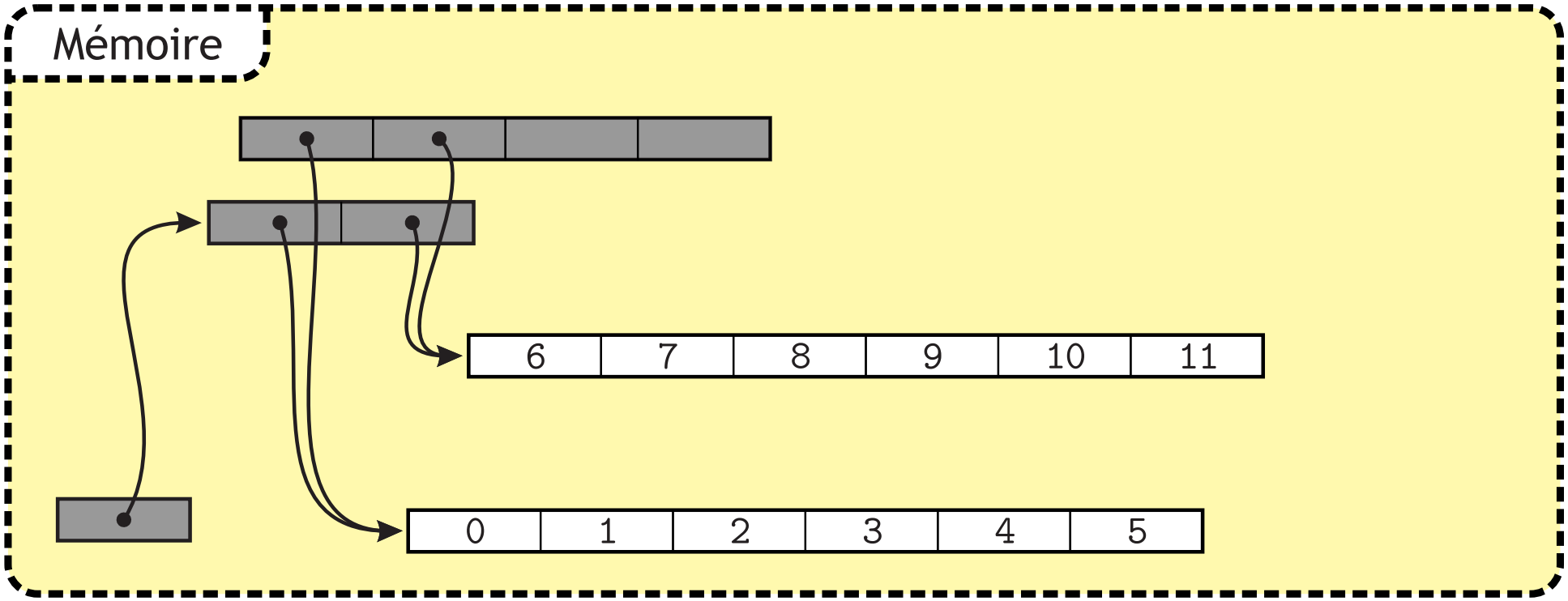
- On peut utiliser un tableau à 2 dimensions : un tableau dynamique de pointeurs vers des tableaux de 1000 cases (par exemple).
  - on n'a que des petits blocs mémoire,
  - on ne recopie que les pointeurs en réallouant,
  - les lectures/écritures sont un peu plus lentes
    - → utile pour des tableaux de plusieurs millions de cases.

Tableaux dynamiques avec des tableaux à deux dimensions



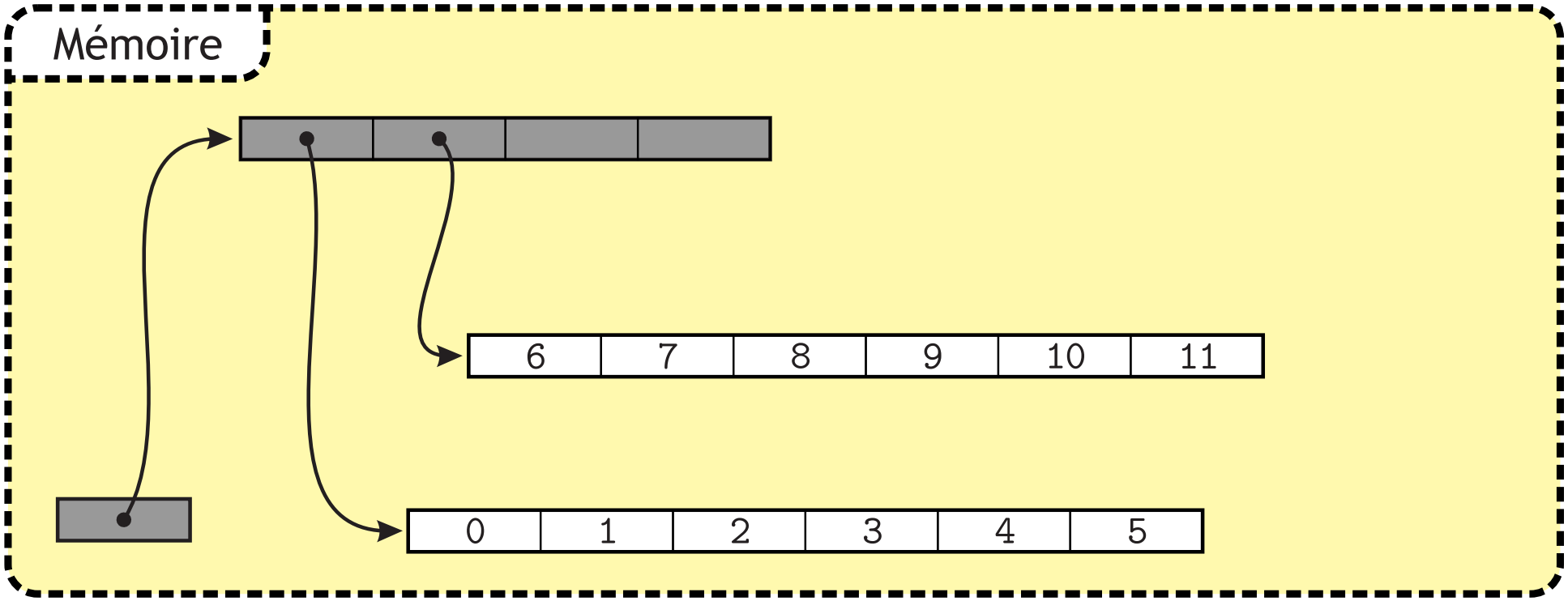
- En mémoire le tableau dynamique à deux dimensions est un pointeur vers un tableau de pointeurs vers des grands tableaux.

Tableaux dynamiques avec des tableaux à deux dimensions



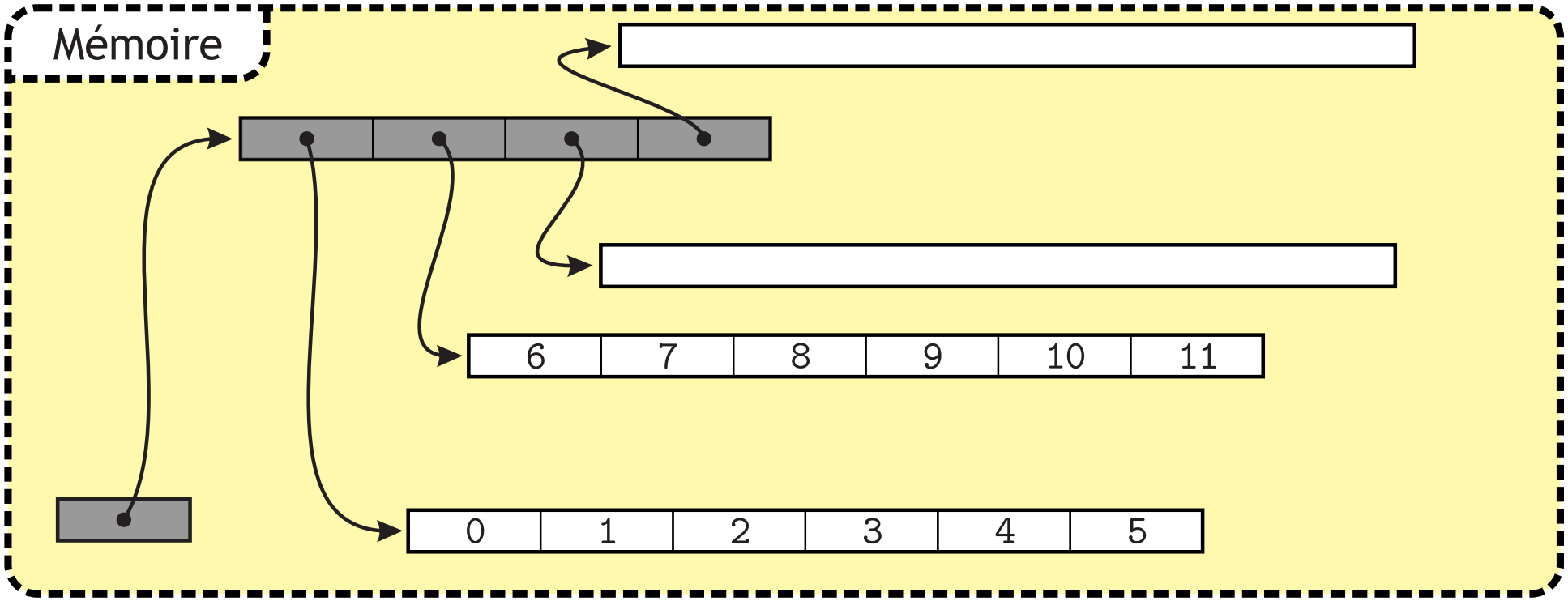
- Quand on veut augmenter la capacité, on réalloue uniquement le tableau de pointeurs :
  - on double la taille,
  - on recopie les pointeurs vers les grands tableaux.

Tableaux dynamiques avec des tableaux à deux dimensions



- Puis on pointe sur le nouveau tableau et on libère l'ancien.
- Sur notre exemple, on a recopié 2 éléments au lieu de 12.

Tableaux dynamiques avec des tableaux à deux dimensions



- On finit en allouant de nouveaux grands tableaux :
  - il n'est pas nécessaire de tout allouer d'un coup,
  - on peut faire ça au fur et à mesure des besoins,
    - → on utilise le minimum nécessaire de mémoire.

Piles et files

Qu'est-ce qu'une pile ou une file ?

- Comme dans la vie courante, pile et file servent à stocker des choses
  - comme une pile de livres ou une file d'attente.

- On veut pouvoir efficacement (en $\Theta(1)$) :
  - ajouter un élément à la pile/file,
  - tester si la pile/file est vide,
  - extraire un élément de la pile/file.

- La différence entre pile et file est l'ordre d'extraction :
  - dans la pile, on retire en premier les éléments arrivés en derniers,
    - → en anglais LIFO : Last In First Out,
  - dans la file, on retire les éléments dans l'ordre d'arrivée
    - → en anglais FIFO : First In First Out.

Utilisation des piles et files

- Les piles et files s'utilisent de la même manière, mais pour des problèmes différents :
  - la pile mémoire est une vraie structure de pile,
  - les files sont utilisées dans les routeurs pour stocker les paquets en attente,
  - les deux sont utilisées dans de nombreux algorithmes.

- Elles implémentent en général deux fonctions :
  - `push` pour ajouter un élément,
  - `pop` pour extraire un élément.

- Il existe plusieurs façons de les implémenter :
  - avec un tableau,
  - avec une liste chaînée.

Implémentation d'une pile

Avec un tableau

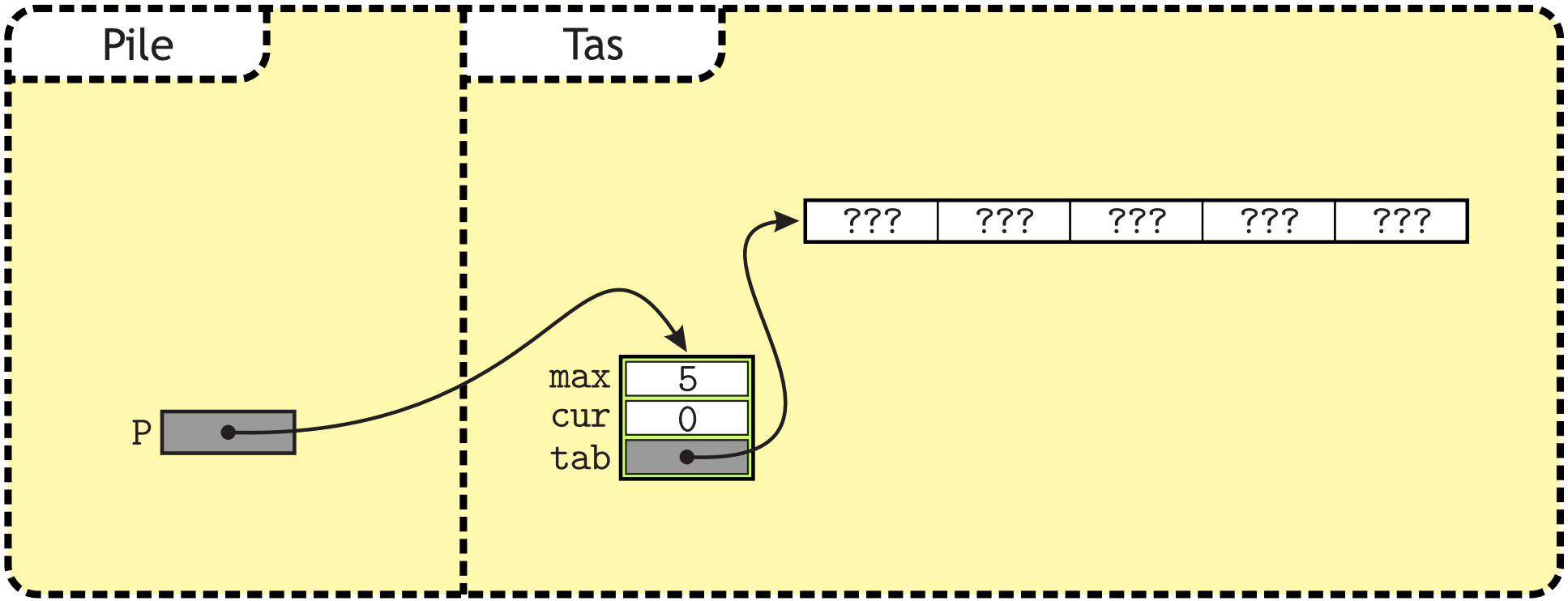
- On considère ici une pile avec une capacité maximale
  - on peut utiliser un tableau dynamique pour enlever cette contrainte.

```
1 typedef struct {
2   unsigned int max;       // capacité max
3   unsigned int cur;       // nombre d'éléments
4   int* tab;
5 } stack;
```

- La pile est donc juste un tableau `tab` :
  - `max` est la capacité maximale,
  - `cur` est le nombre d'éléments stockés (à mettre à jour).
- En permanence les éléments sont au début du tableau :
  - on ajoute et on retire par la fin,
  - c'est tout simple !

Fonctionnement d'une pile

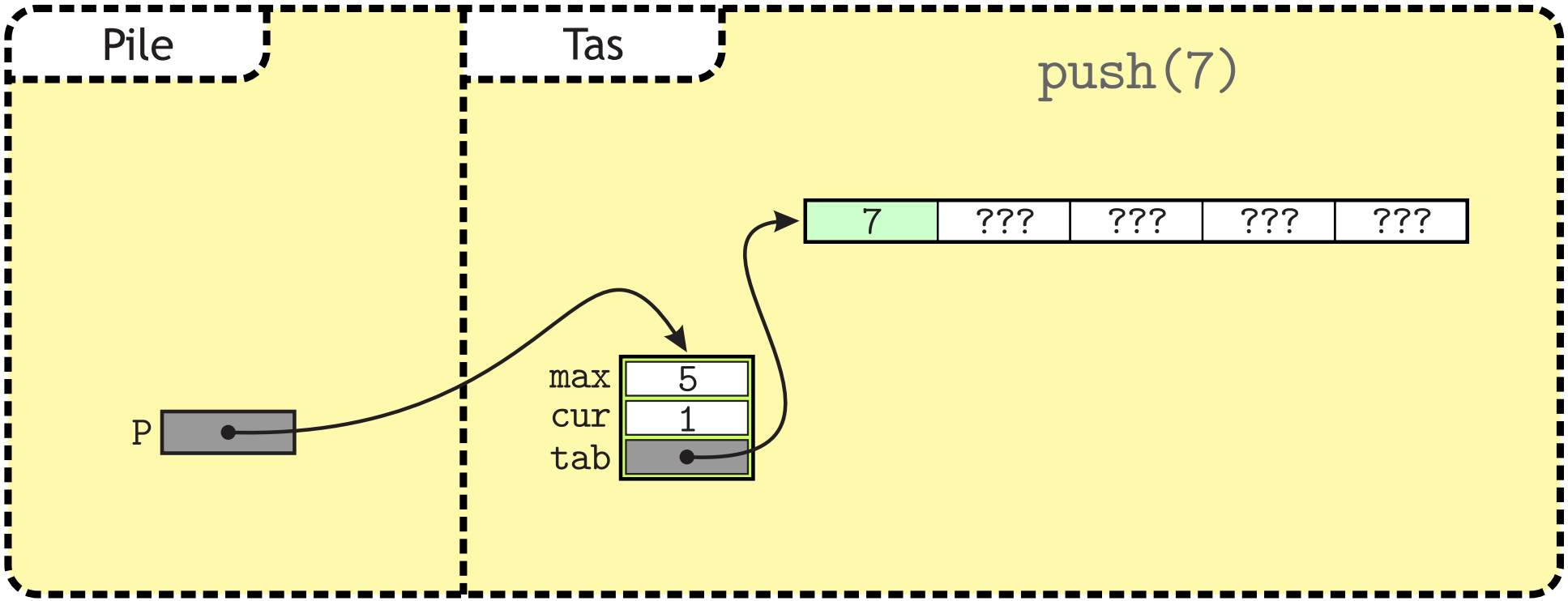
Avec un tableau



- La pile est vide au départ : `cur` vaut 0.

Fonctionnement d'une pile

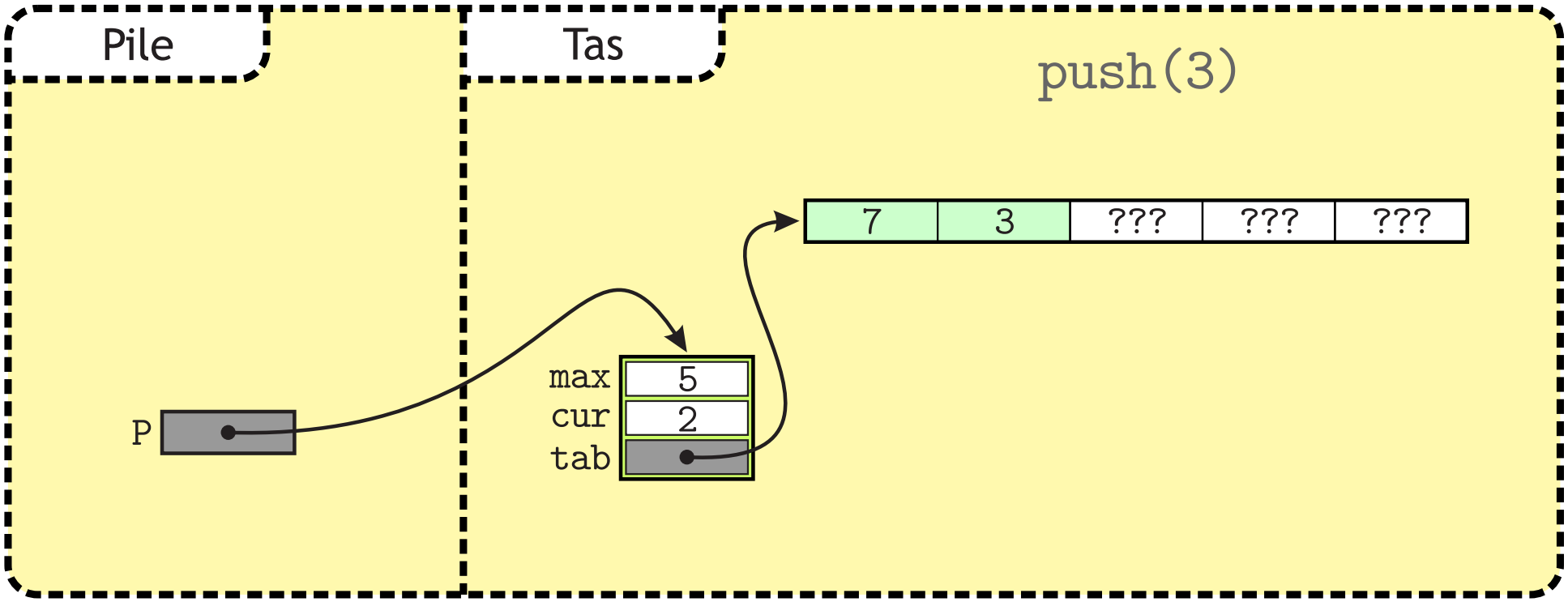
Avec un tableau



- Chaque `push` ajoute un élément dans le tableau et incrémente `cur`.

Fonctionnement d'une pile

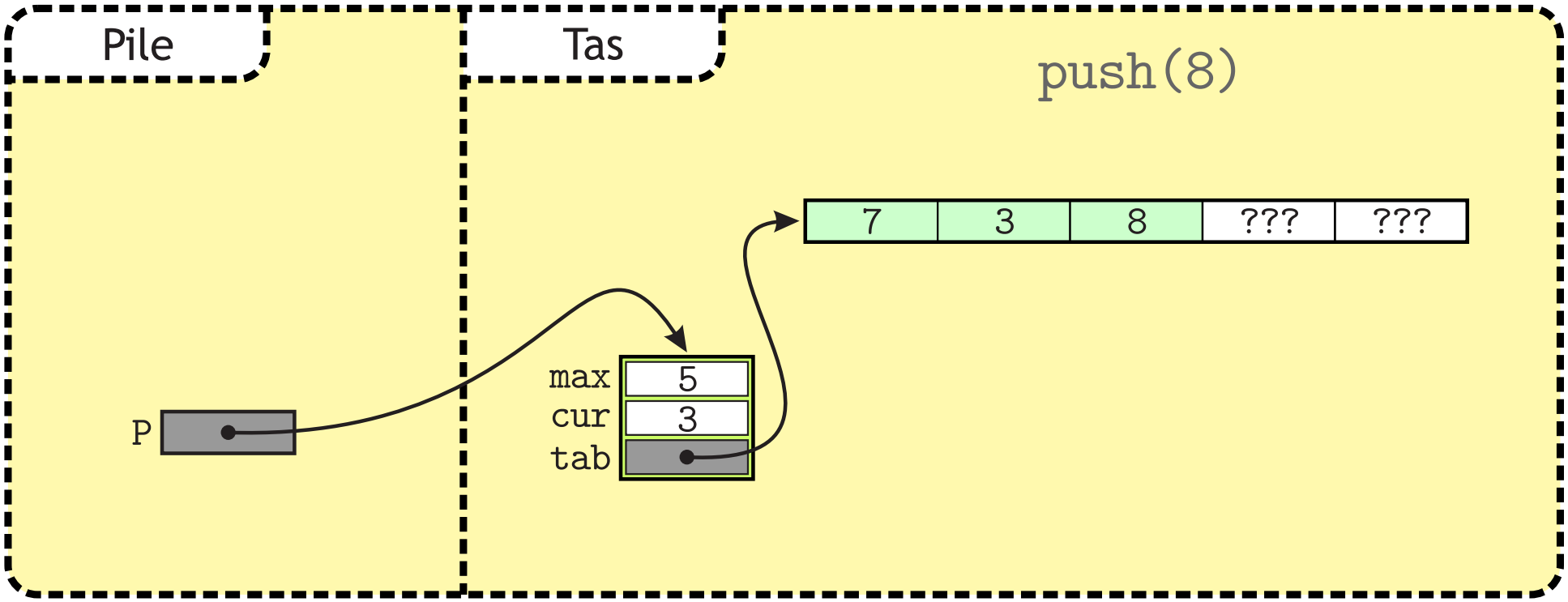
Avec un tableau



- Chaque `push` ajoute un élément dans le tableau et incrémente `cur`.

Fonctionnement d'une pile

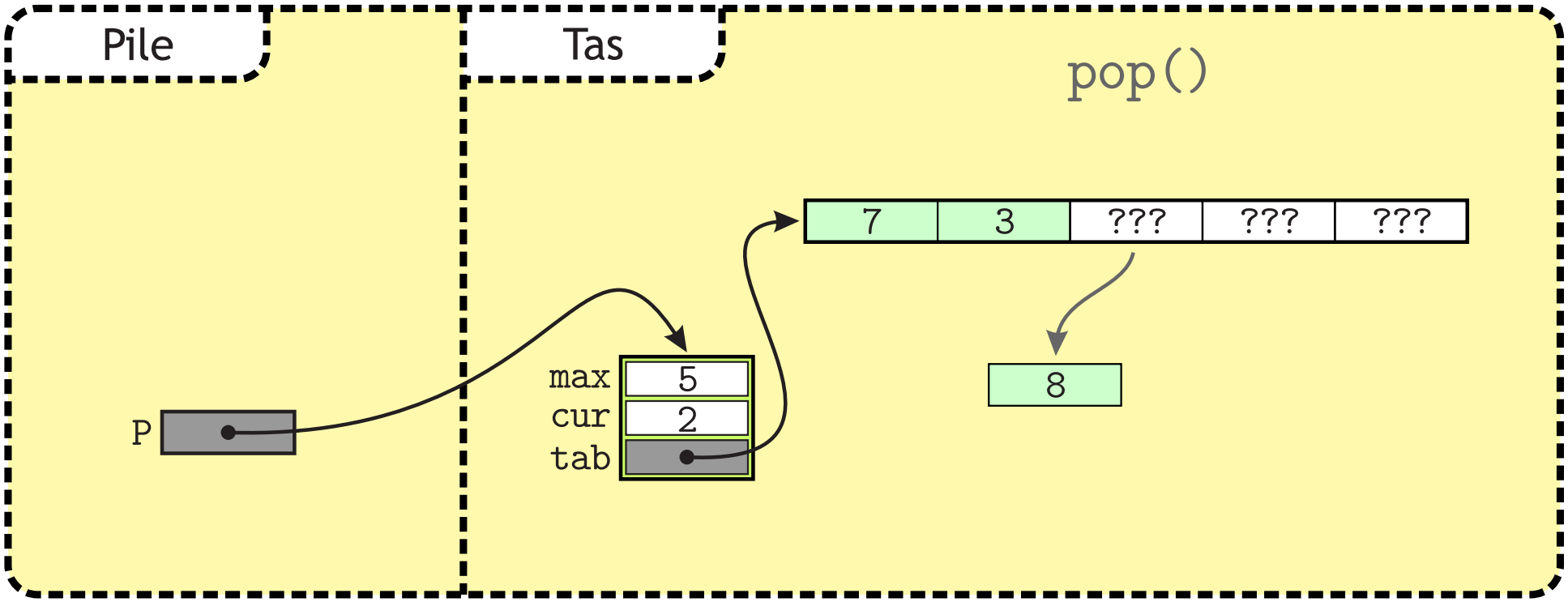
Avec un tableau



- Chaque `push` ajoute un élément dans le tableau et incrémente `cur`.

Fonctionnement d'une pile

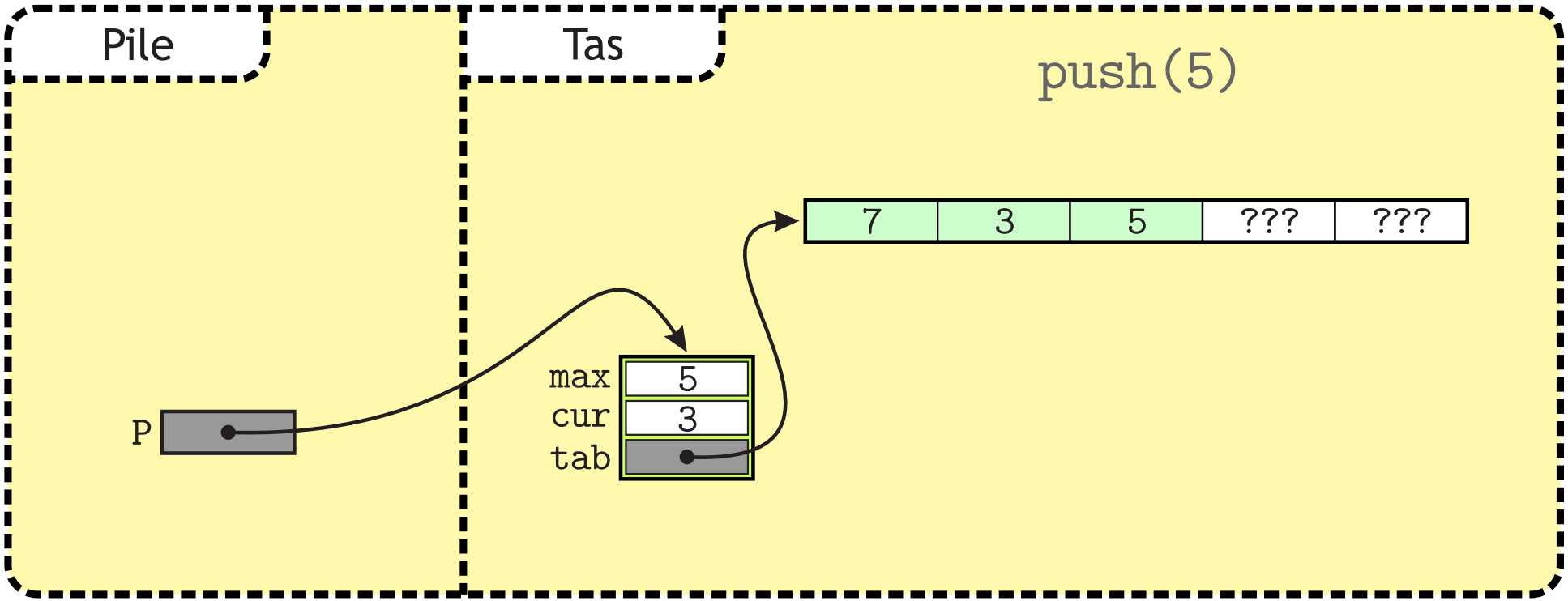
Avec un tableau



- Chaque `pop` extrait le dernier élément du tableau et décrémente `cur`.

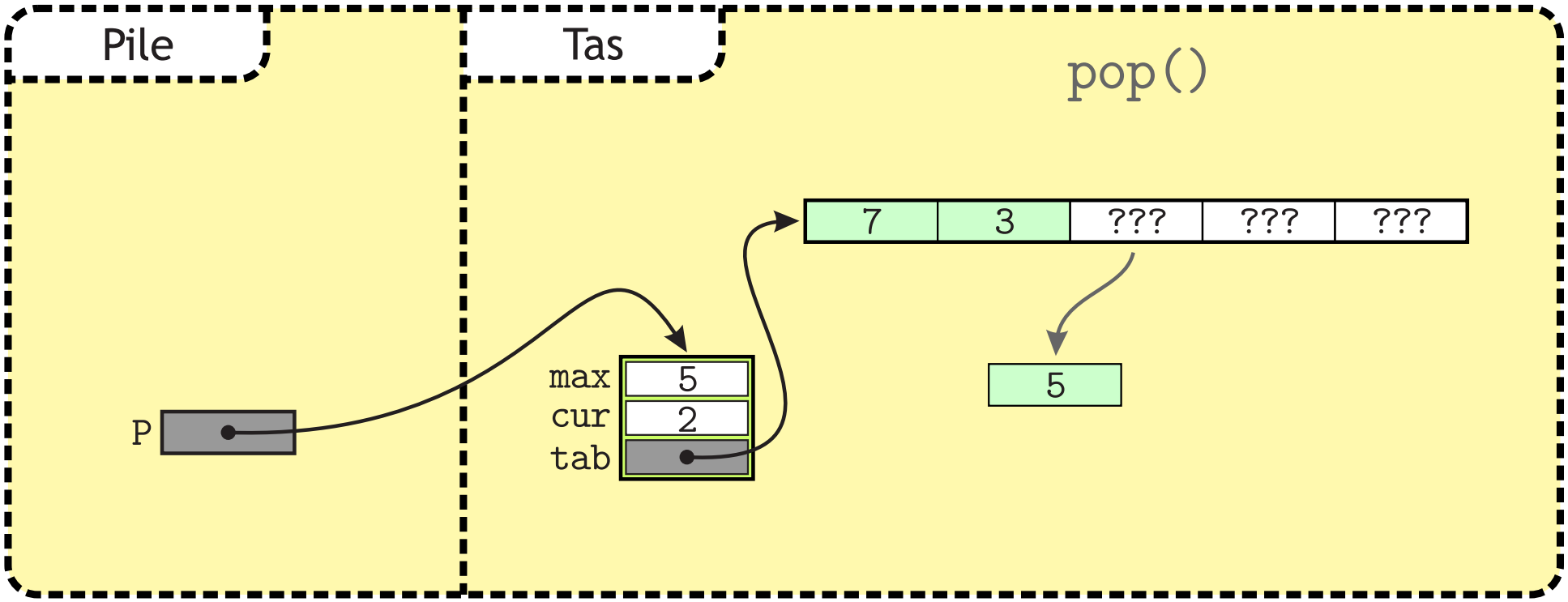
Fonctionnement d'une pile

Avec un tableau



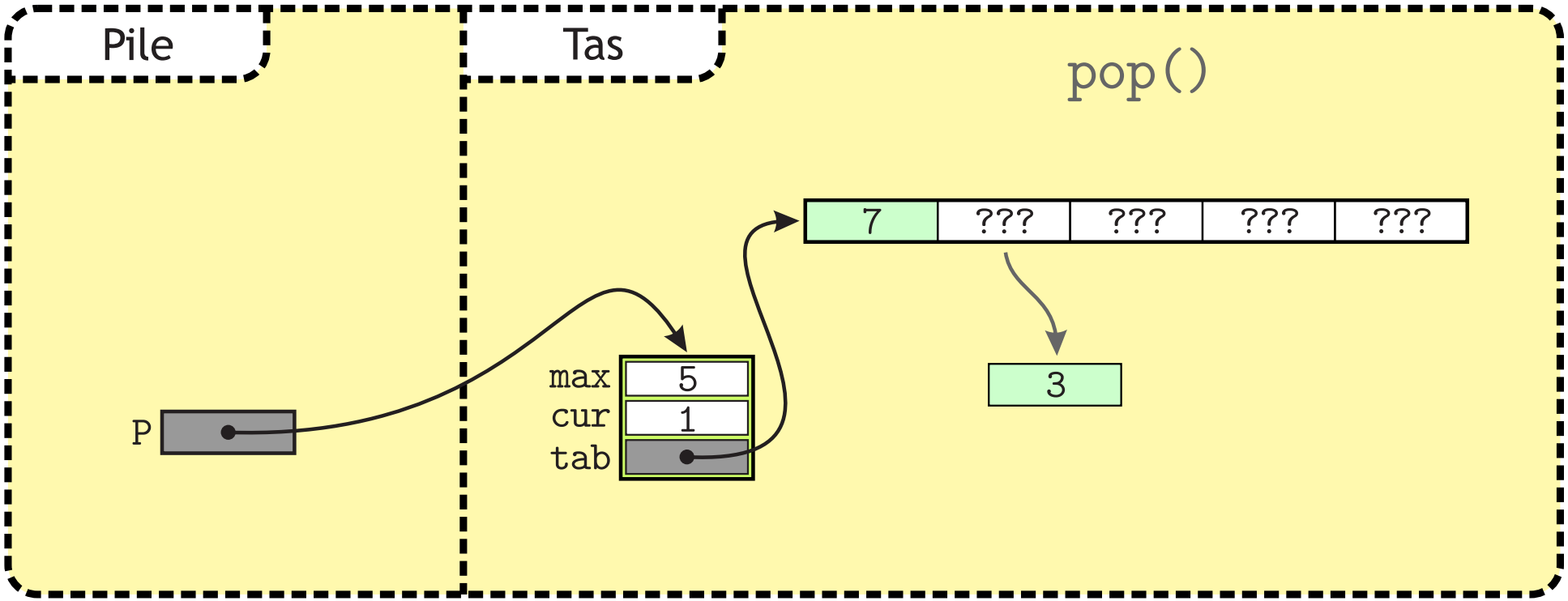
Fonctionnement d'une pile

Avec un tableau



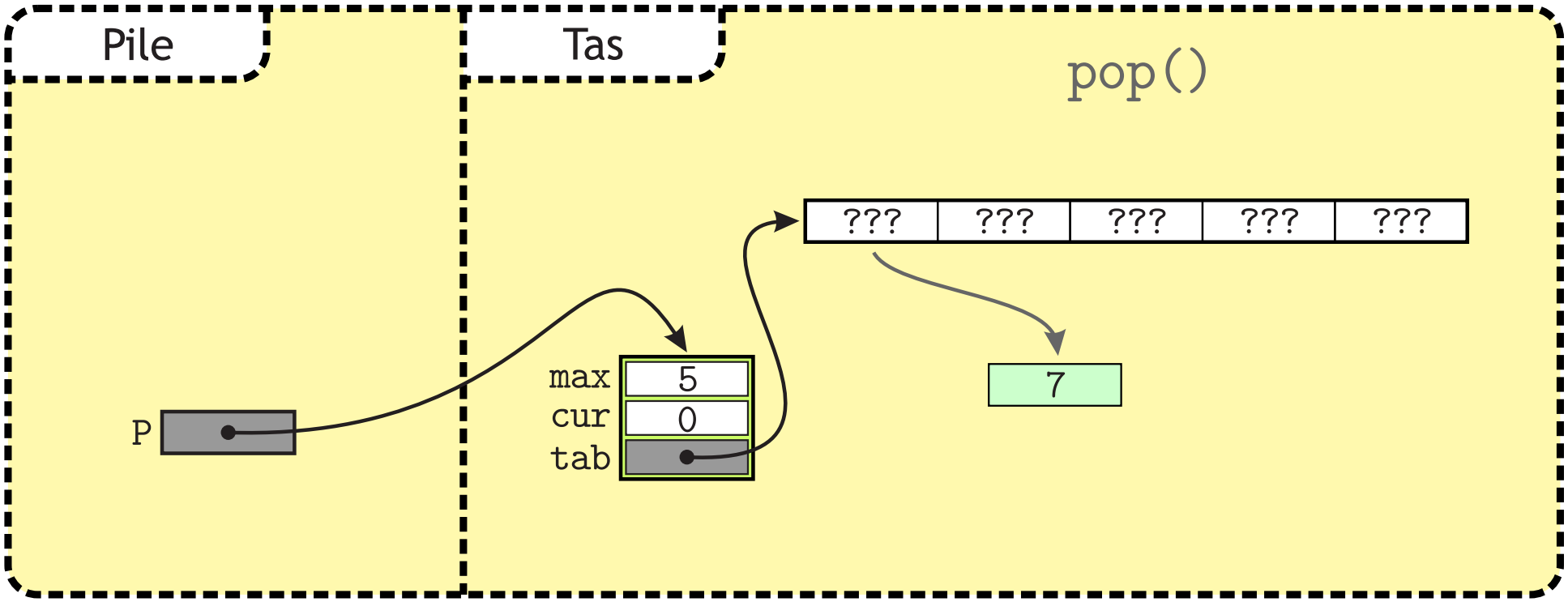
Fonctionnement d'une pile

Avec un tableau



Fonctionnement d'une pile

Avec un tableau



- Le fonctionnement est donc très simple,
  - on peut aussi remplacer `tab` par un tableau dynamique
    - → plus de limite de capacité.

Implémentation d'une pile en C

Avec un tableau

```
1  int pop(stack* P) {
2    if (P->cur == 0) {
3      printf("Pile vide !\n");
4      return -1;
5    }
6    P->cur--;
7    return P->tab[P->cur];
8  }
9
10 void push(stack* P, int val) {
11   if (P->cur == P->max) {
12     printf("Pile pleine !\n");
13     return;
14   }
15   P->tab[P->cur] = val;
16   P->cur++;
17 }
```

Implémentation d'une pile en C

Avec un tableau dynamique

```
1  int pop(stack* P) {
2    if (P->cur == 0) {
3      printf("Pile vide !\n");
4      return -1;
5    }
6    P->cur--;
7    return P->tab[P->cur];
8  }
9
10 void push(stack* P, int val) {
11   if (P->cur == P->max) {
12     P->max *= 2;
13     P->tab = (int*) realloc(P->tab, P->max);
14   }
15   P->tab[P->cur] = val;
16   P->cur++;
17 }
```

Implémentation d'une file

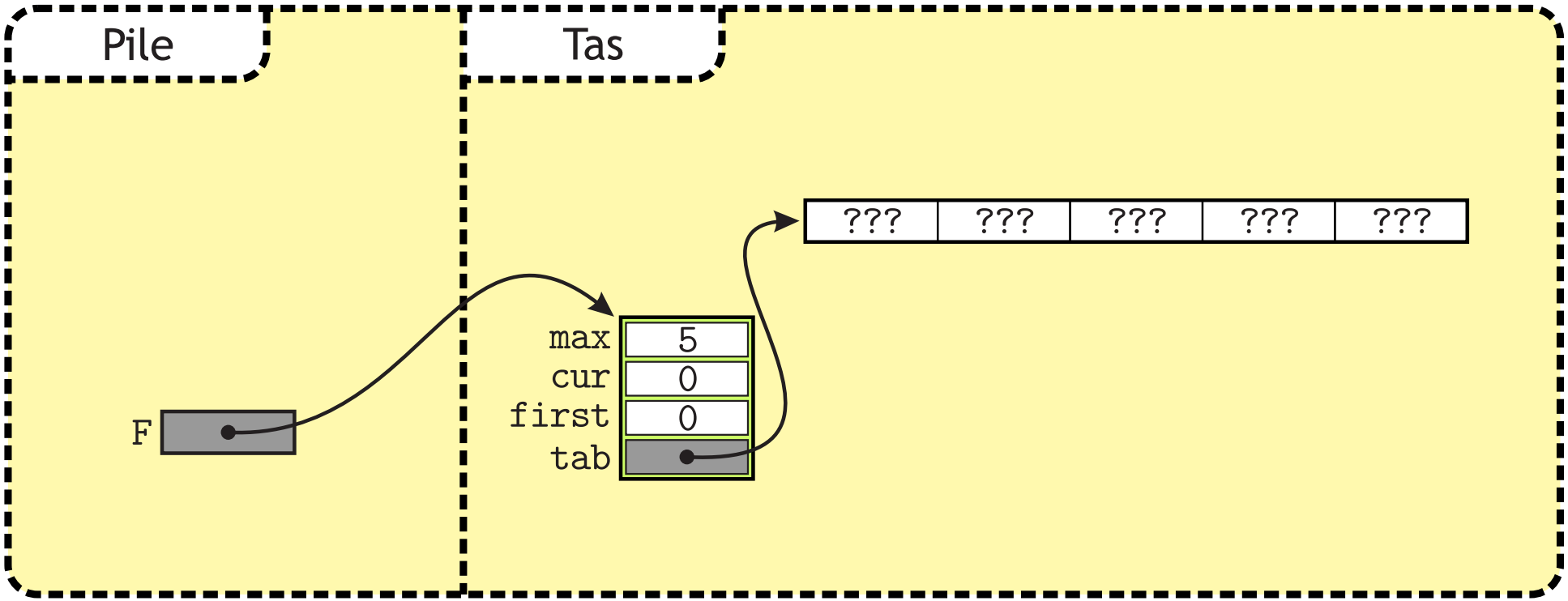
Avec un tableau

- On implémente la file de façon similaire à la pile
  - toujours un tableau avec capacité maximale,
  - on ajoute encore les éléments à la fin du tableau.
- Il faut en revanche enlever les éléments par le début :
  - on ne peut pas décaller tout le tableau (ne se fait pas en $\Theta(1)$),
  - on utilise le tableau de façon cyclique
    - → quand on arrive au bout du tableau on recommence du début.
- On doit donc aussi garder l'indice du premier élément (le plus ancien encore présent).

```
1 typedef struct {
2   unsigned int max;        // capacité max
3   unsigned int cur;        // nombre d'éléments
4   unsigned int first;      // premier élément
5   int* tab;
6 } queue;
```

Fonctionnement d'une file

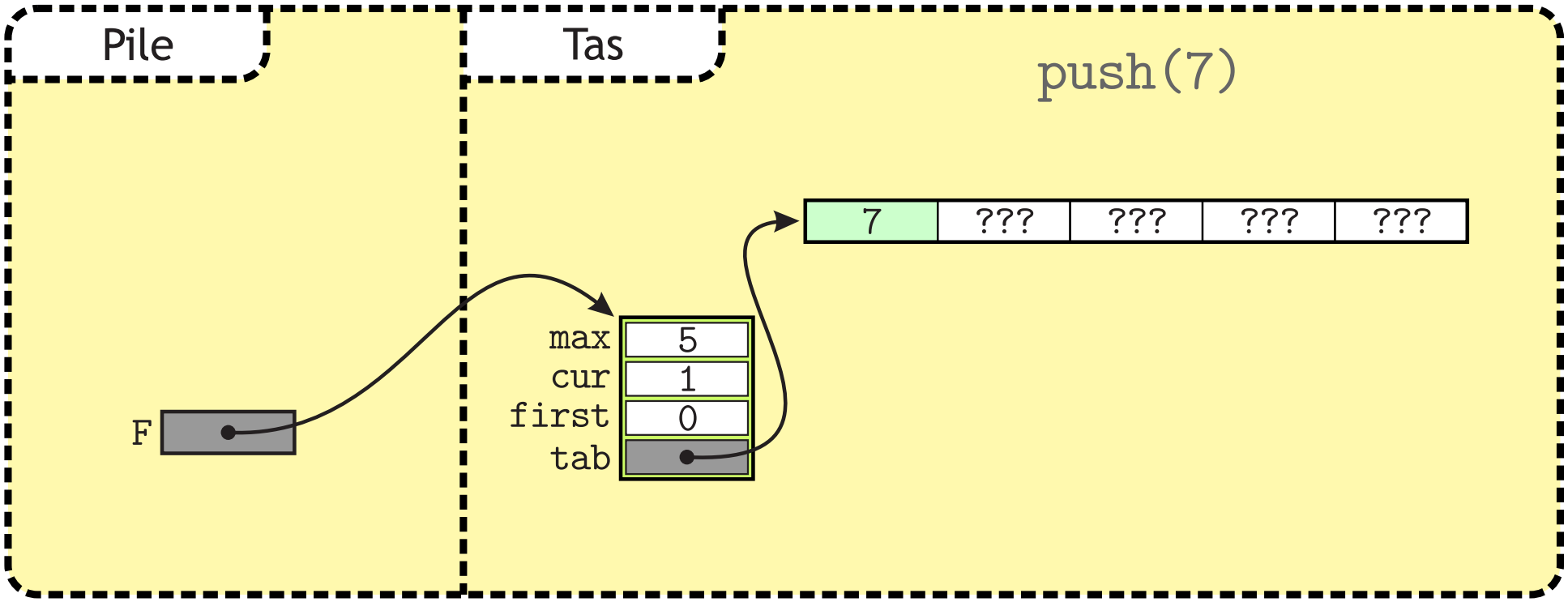
Avec un tableau



- La structure est similaire à celle d'une pile : `cur` et `first` sont initialisés à 0.

Fonctionnement d'une file

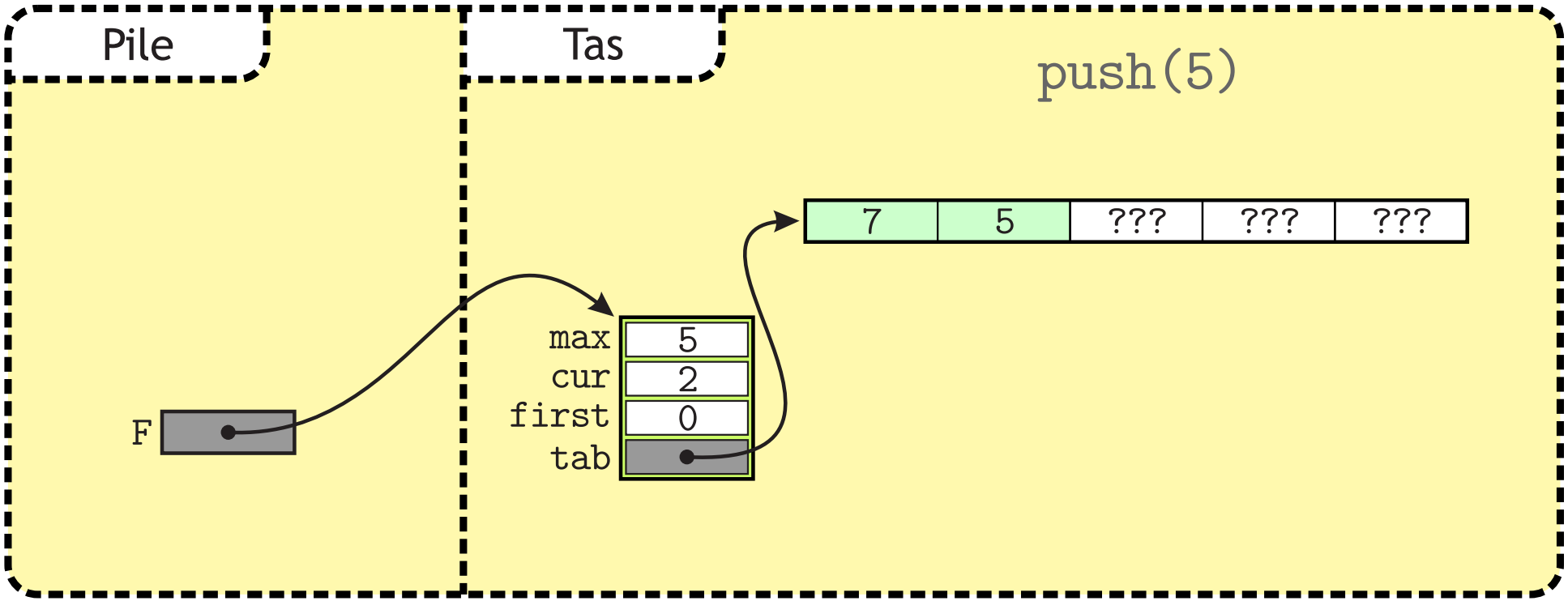
Avec un tableau



- Les `push` fonctionnent comme avec la pile.

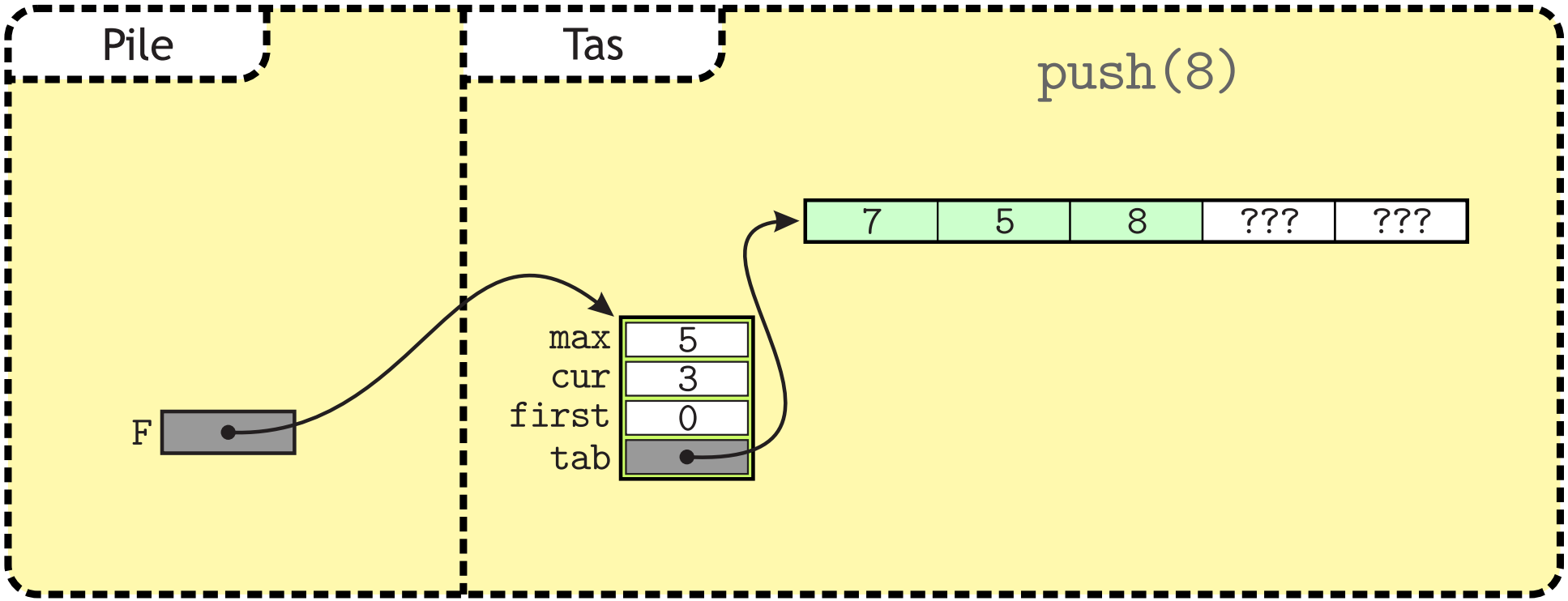
Fonctionnement d'une file

Avec un tableau



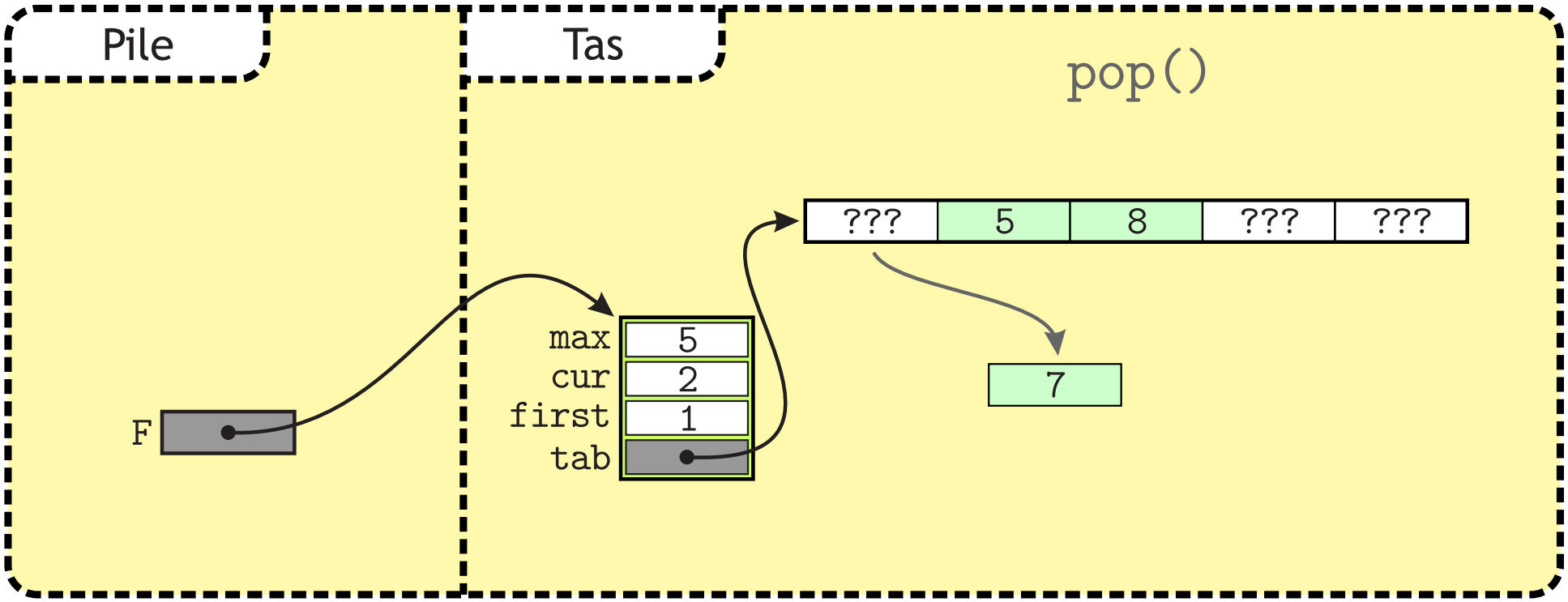
Fonctionnement d'une file

Avec un tableau



Fonctionnement d'une file

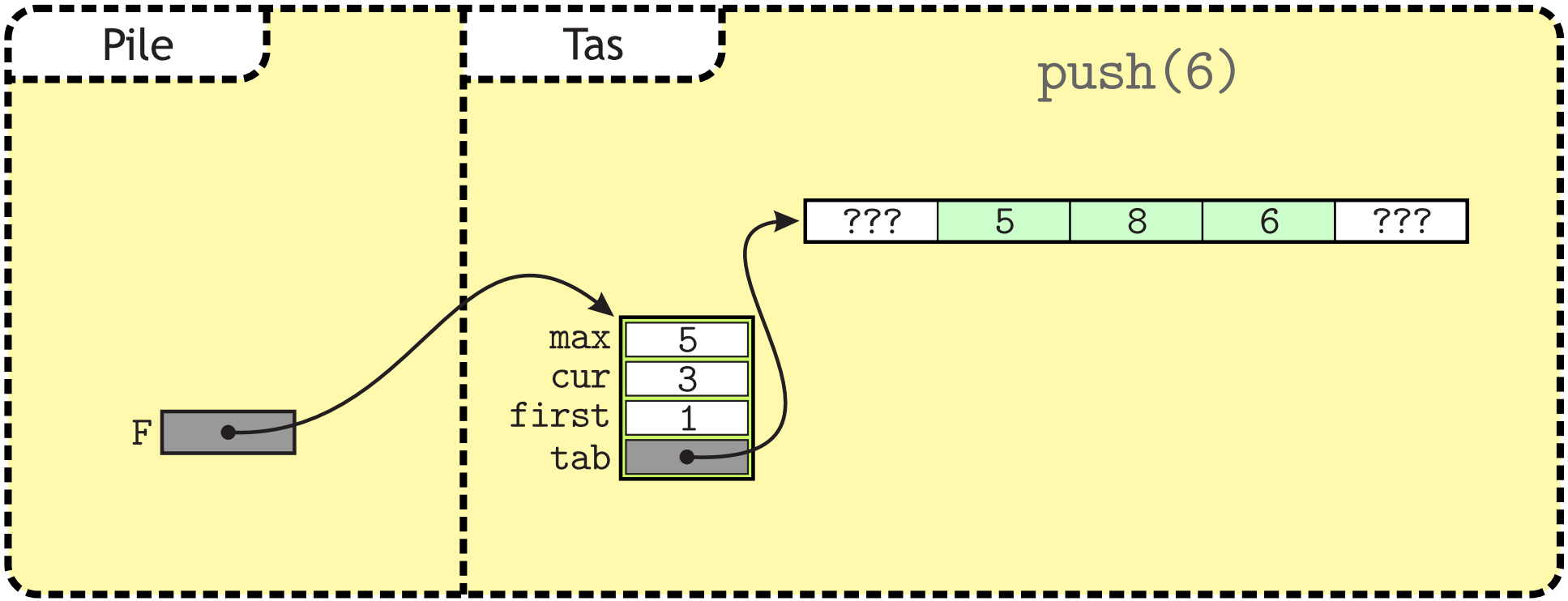
Avec un tableau



- Les pop sont en revanche différents de ceux de la pile :
  - on extrait l'élément le plus ancien (à la position `first`),
  - on incrément `first`,
  - on décrémente `cur`.

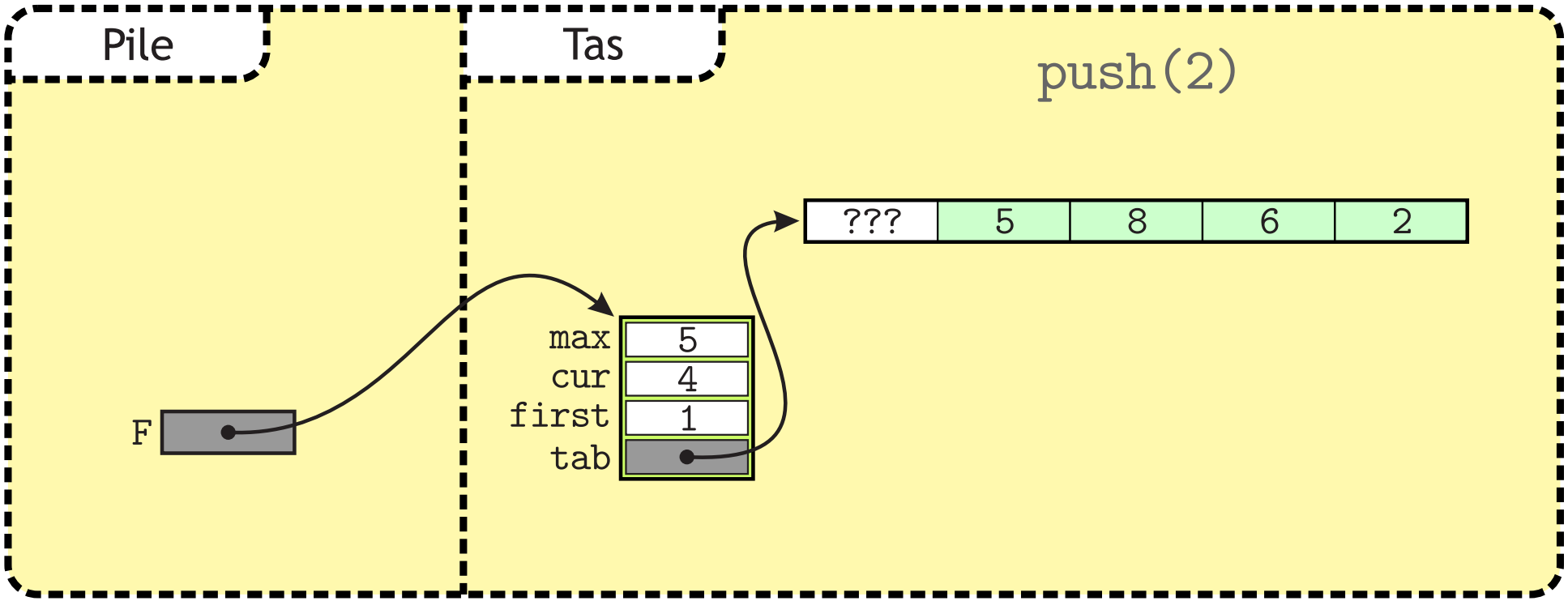
Fonctionnement d'une file

Avec un tableau



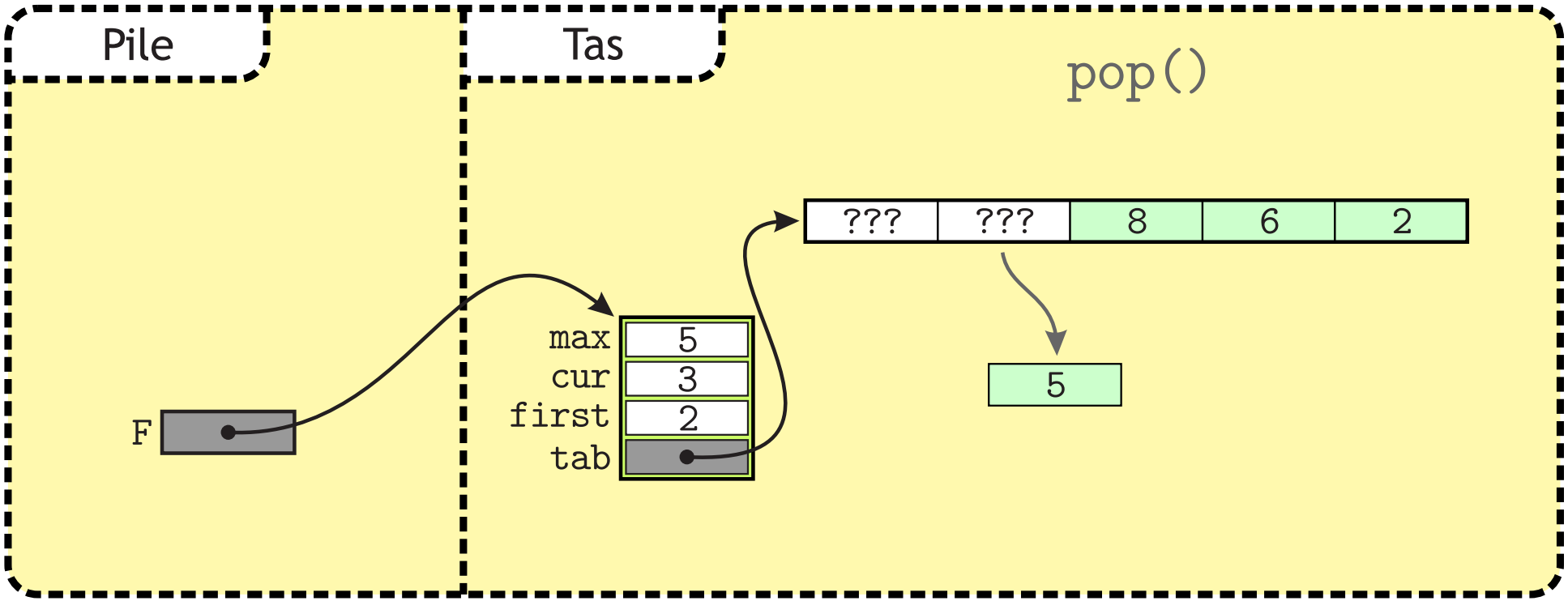
Fonctionnement d'une file

Avec un tableau



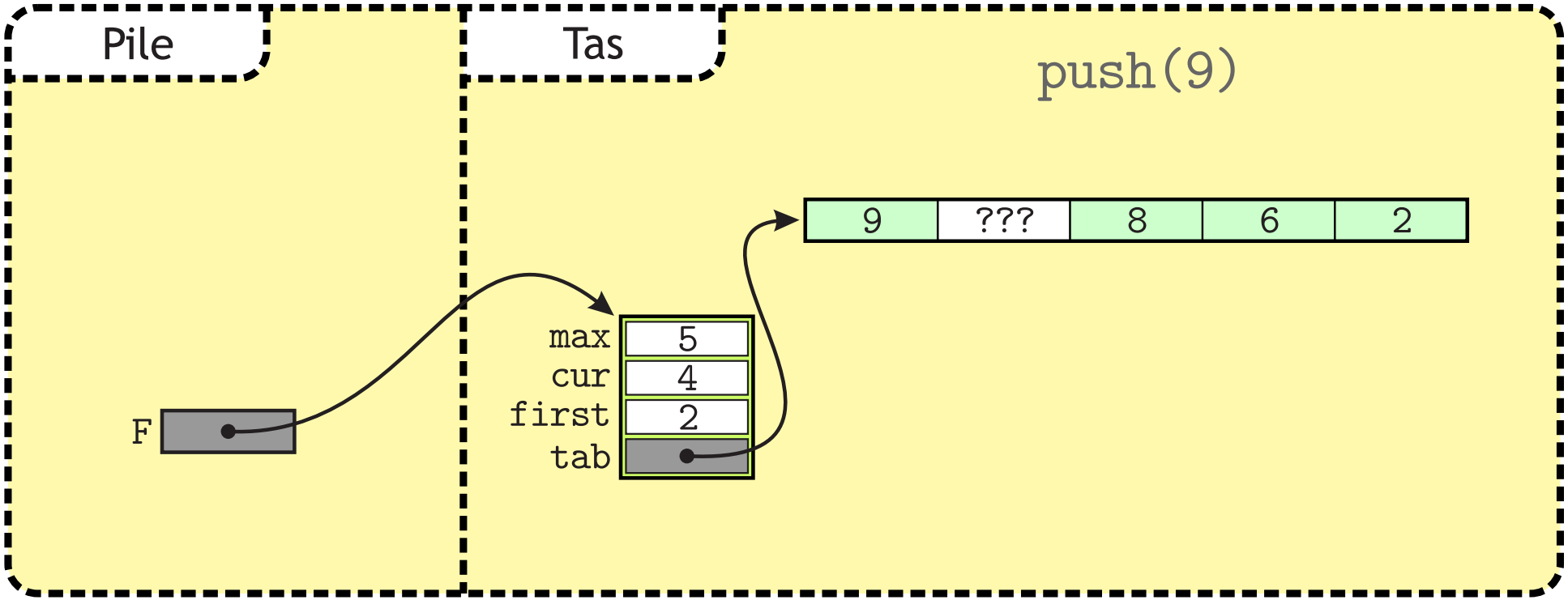
Fonctionnement d'une file

Avec un tableau



Fonctionnement d'une file

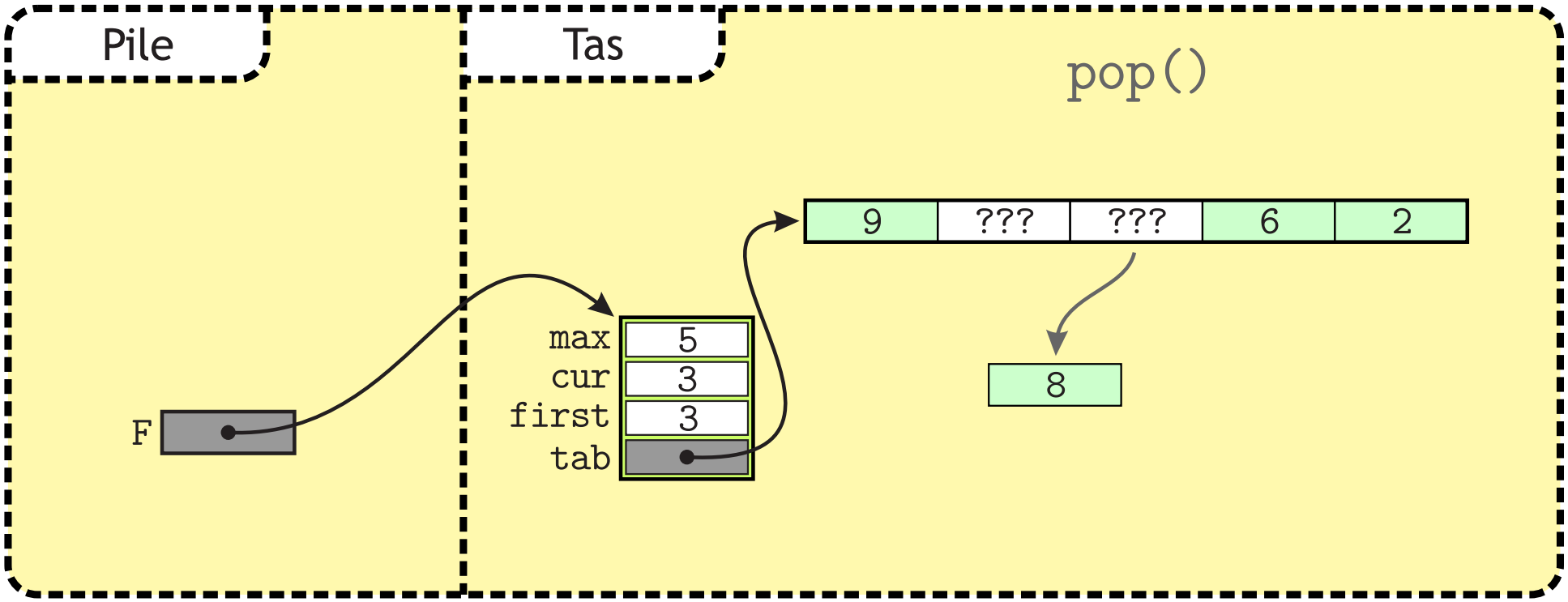
Avec un tableau



- Quand on arrive au bout du tableau, la file n'est pas forcément pleine :
  - les premiers éléments sont "vides",
  - on peut donc faire un cycle et réinsérer au début.

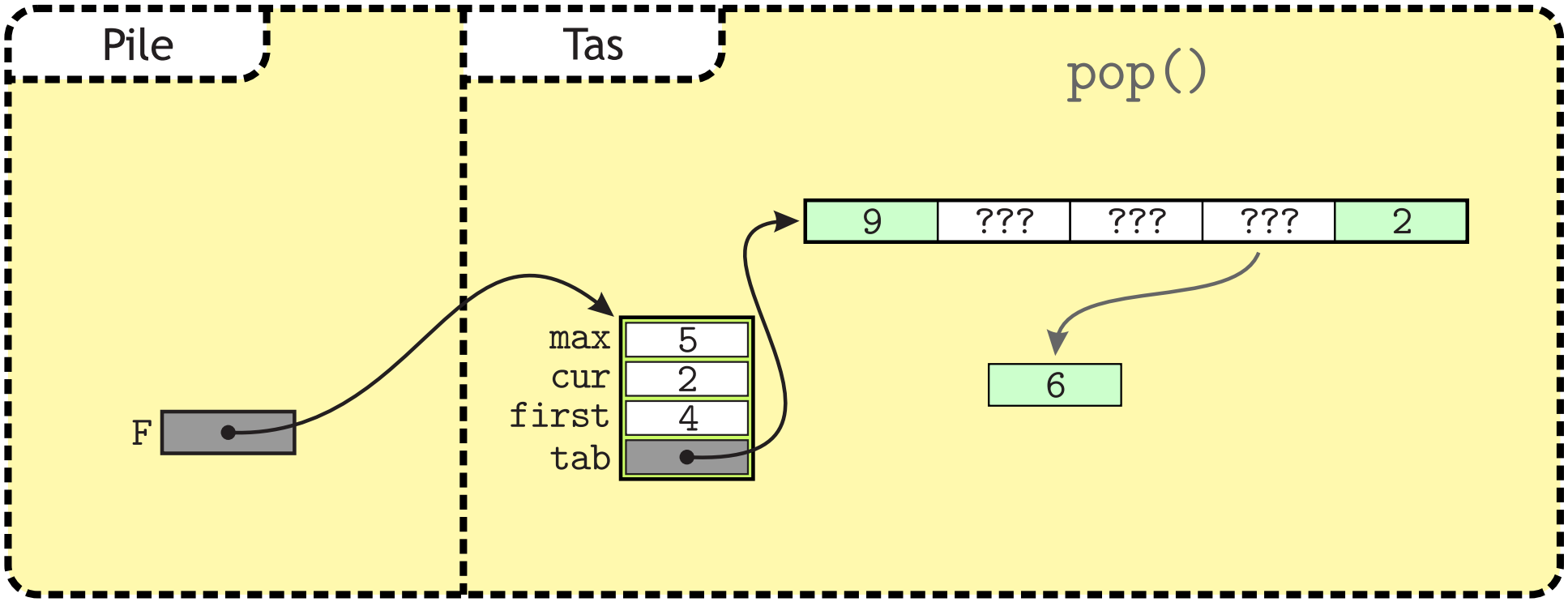
Fonctionnement d'une file

Avec un tableau



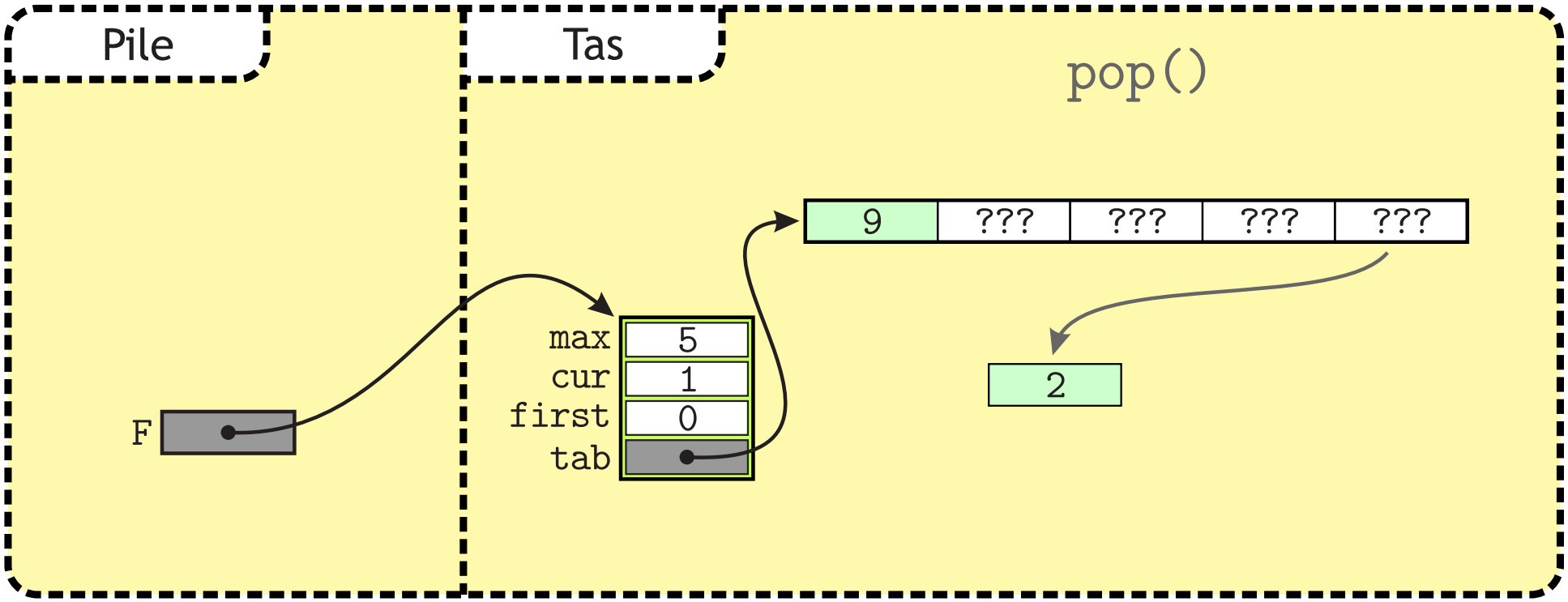
Fonctionnement d'une file

Avec un tableau



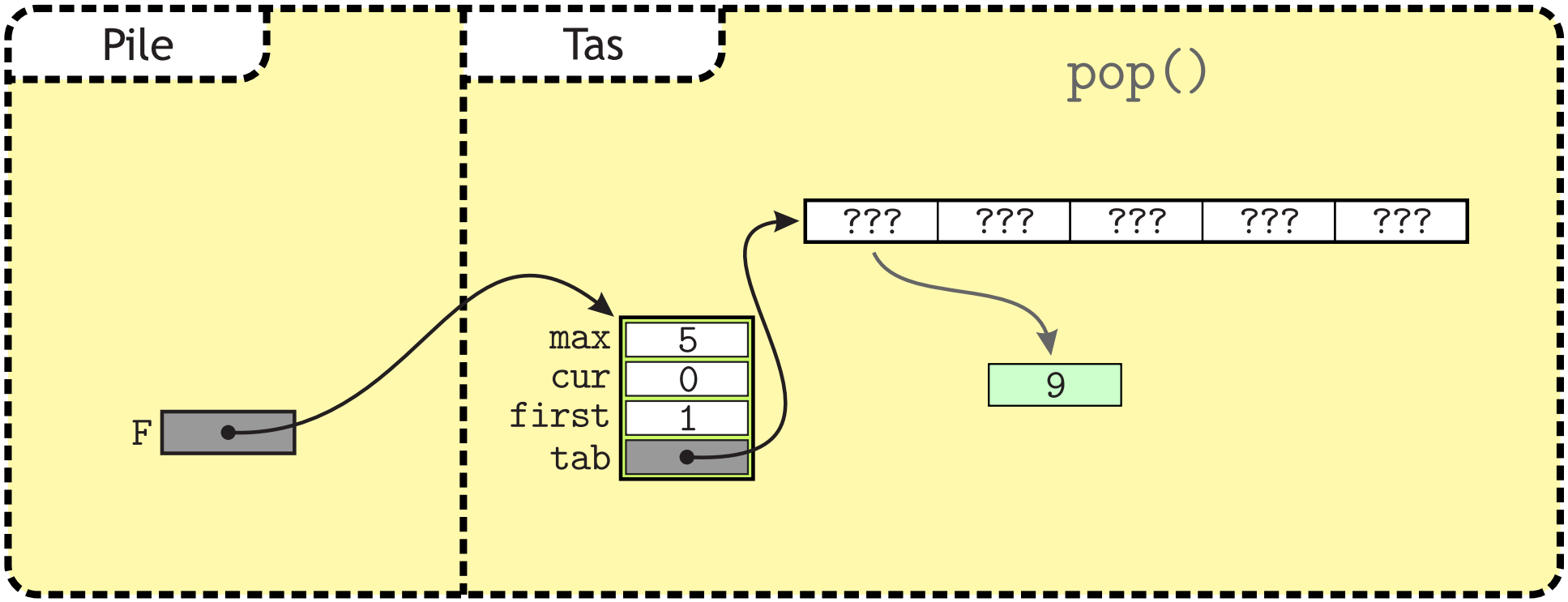
Fonctionnement d'une file

Avec un tableau



Fonctionnement d'une file

Avec un tableau



- L'extraction aussi s'effectue de façon cyclique :
  - il suffit de calculer les indices d'insertion et d'extraction modulo `max`.

Implémentation d'une file en C

Avec un tableau

```
1  int pop(queue* F) {
2    int res;
3    if (F->cur == 0) {
4      printf("File vide !\n");
5      return -1;
6    }
7    res = F->tab[F->first];
8    F->first = (F->first+1) % F->max;  // incrémentation "cyclique"
9    F->cur--;
10   return res;
11 }
12 void push(queue* F, int val) {
13   if (F->cur == F->max) {
14     printf("File pleine !\n");
15     return;
16   }
17   F->tab[(F->first+F->cur) % F->max] = val;
18   F->cur++;
19 }
```

Implémentation d'une file en C

Avec un tableau dynamique

```
1  void push(queue* F, int val) {
2    int* new_tab;
3    if (F->cur == F->max) {  // realloc ne marche pas ici...
4      new_tab = (int*) malloc(2*F->max*sizeof(int));
5      // on utilise : memcpy(dest, source, taille);
6      // on doit le faire en 2 blocs
7      memcpy(new_tab, &(F->tab[F->first]), (F->max-F->first)*sizeof(int));
8      memcpy(&(new_tab[F->max-F->first]), F->tab, F->first*sizeof(int));
9      free(F->tab);           // on libère l'ancien tab
10     F->tab = new_tab;
11     F->first = 0;
12     F->max *= 2;
13   }
14   F->tab[(F->first+F->cur) % F->max] = val;
15   F->cur++;
16 }
```

- Ça marche bien, mais c'est un peu compliqué à gérer...

Listes chaînées

Qu'est-ce qu'une liste chaînée ?

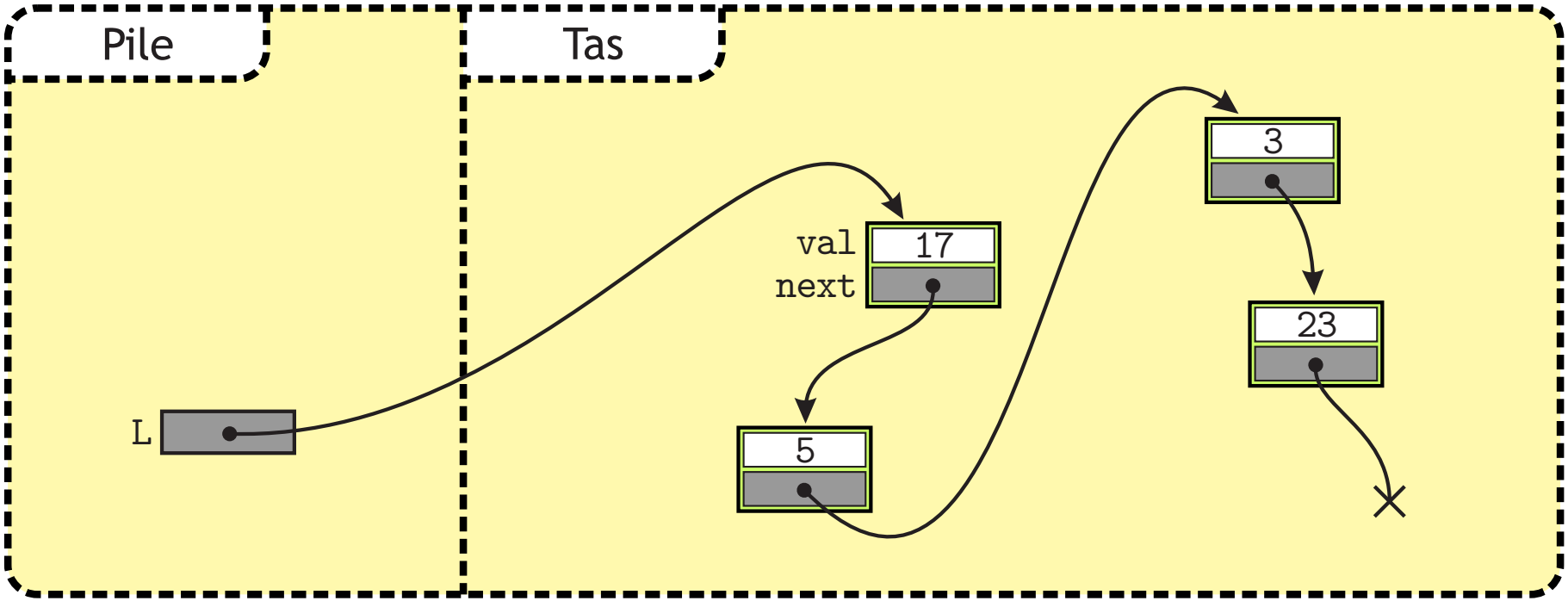
- Une liste chaînée est une structure similaire à une chaîne :
  - chaque élément est un maillon de la chaîne
  - on ne sait pas efficacement accéder au i-ème élément,
  - on sait passer d'un élément au suivant (ou précédent).
- En C, chaque élément d'une liste chaînée contient des données et un pointeur vers l'élément suivant :

```
1 typedef struct cell_st {
2   int val;       // le contenu n'est pas forcément un int
3   struct cell_st* next;
4 } cell;
```

- La liste chaînée à proprement parler est simplement un pointeur vers le premier élément.
  → le dernier élément pointe vers `NULL`.

Qu'est-ce qu'une liste chaînée ?

- En mémoire les éléments de la liste sont dispersés :
  - on ne peut pas savoir directement où est le *i*-ème,
  - il faut suivre toute la chaîne pour parcourir la liste.
- Pour compter les éléments il faut aussi tout parcourir.



Opérations de base sur une liste chaînée

- La structure de liste chaînée est très flexible et permet de nombreuses opérations en temps constant $\Theta(1)$:
  - insérer un élément au début,
  - supprimer le premier élément,
  - avancer d'un élément au suivant,
  - insérer un élément après un élément dont on connait l'adresse,
  - supprimer l'élément suivant un élément dont on connait l'adresse,
  - insérer un élément à la fin
    - → si on garde en plus un pointeur sur le dernier élément

- En plus de ces opérations "standards", on peut faire toutes sortes d'opérations bizarres en ne changeant qu'un ou deux pointeurs :
  - échanger la fin de deux listes,
  - coller deux listes bout à bout,
  - créer deux listes avec une fin commune...

Utilisation de listes chaînées

Exemples en C

insertion

```
1  typedef struct cell_st {
2    int val;
3    struct cell_st* next;
4  } cell;
5
6  int main() {
7    cell* L = NULL;       // liste vide
8    cell* tmp;
9
10   // insertion de 4 en début de liste
11   tmp = (cell*) malloc(sizeof(cell));
12   tmp->val = 4;
13   tmp->next = L;
14   L = tmp;
15
16   // insertion de 7
17   tmp = (cell*) malloc(sizeof(cell));
18   tmp->val = 7;
19   tmp->next = L;
20   L = tmp;
21 }
```

Utilisation de listes chaînées

Exemples en C

- On peut aussi utiliser une fonction d'insertion
  - ⚠ il faut modifier une variable de la fonction appelante
    → passage par adresse.

insertion

```
1  void insert(cell** L, int v) {
2    cell* new = (cell*) malloc(sizeof(cell));
3    new->val = v;
4    new->next = *L;
5    *L = new;
6  }
7
8  int main() {
9    cell* L = NULL;
10   insert(&L, 4);
11   insert(&L, 7);
12 }
```

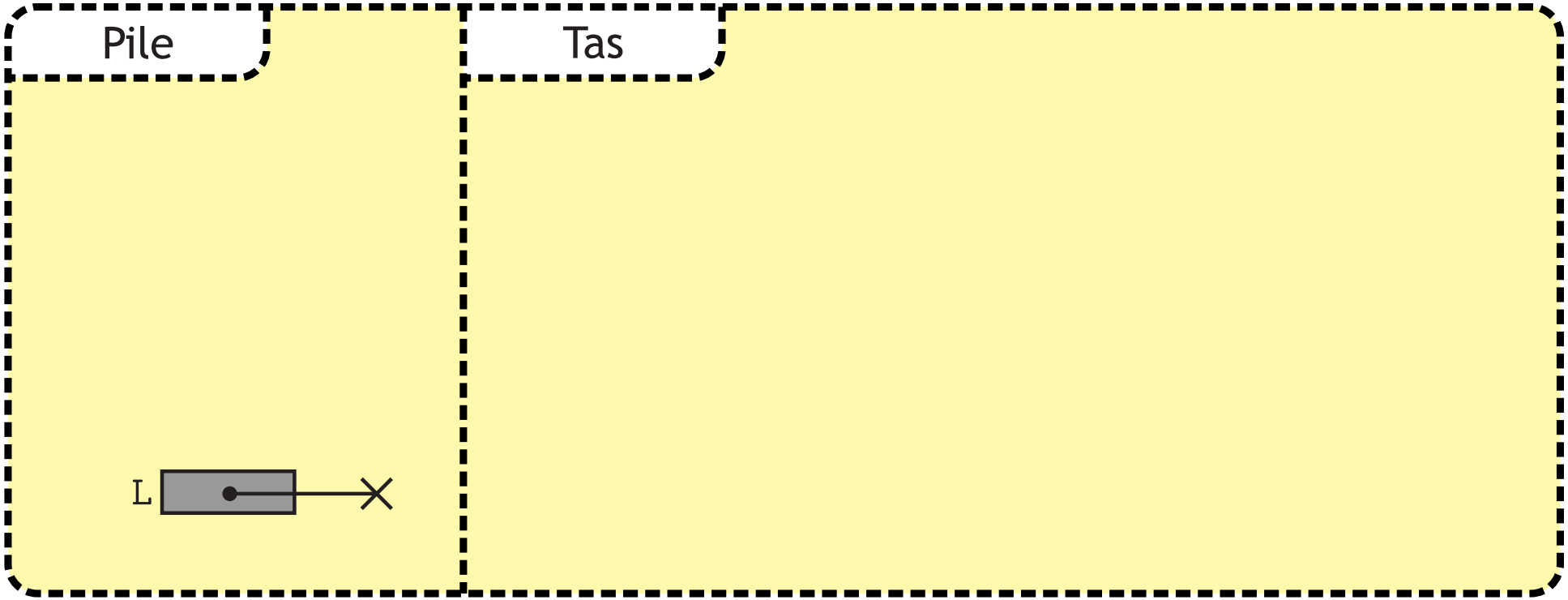
- Cette même fonction permet d'insérer n'importe où dans la liste
  - il suffit de lui passer `&(cur->next)` pour insérer après `cur`.

Utilisation de listes chaînées

Exemples en C

```
1 void insert(cell** L, int v) {
2   cell* new = malloc(sizeof(cell));
3   new->val = v;
4   new->next = *L;
5   *L = new;
6 }
```

```
7 int main() {
8 ▶ cell* L = NULL;
9   insert(&L, 4);
10  insert(&L, 7);
11 }
```

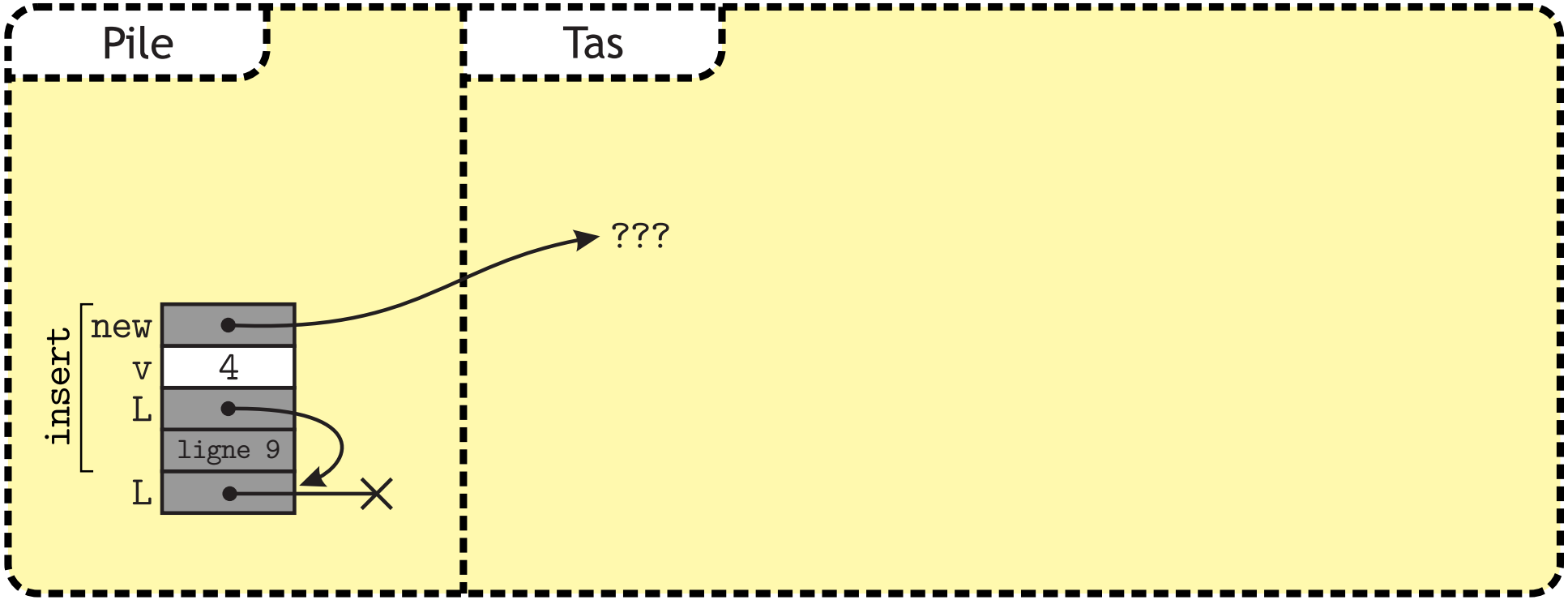


Utilisation de listes chaînées

Exemples en C

```
1 void insert(cell** L, int v) {
2   cell* new = malloc(sizeof(cell));
3   new->val = v;
4   new->next = *L;
5   *L = new;
6 }
```

```
7 int main() {
8   cell* L = NULL;
9 ▶ insert(&L, 4);
10  insert(&L, 7);
11 }
```

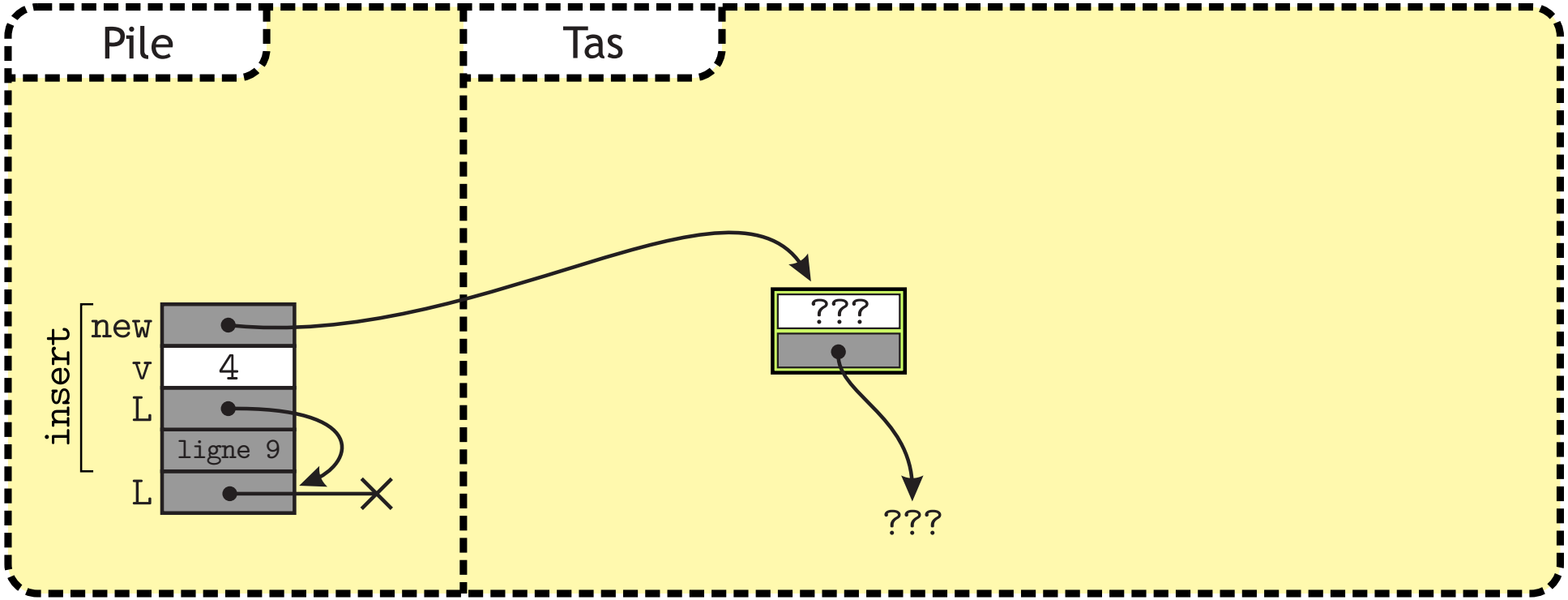


Utilisation de listes chaînées

Exemples en C

```
1 void insert(cell** L, int v) {
2 ▶ cell* new = malloc(sizeof(cell));
3   new->val = v;
4   new->next = *L;
5   *L = new;
6 }
```

```
7 int main() {
8   cell* L = NULL;
9   insert(&L, 4);
10  insert(&L, 7);
11 }
```

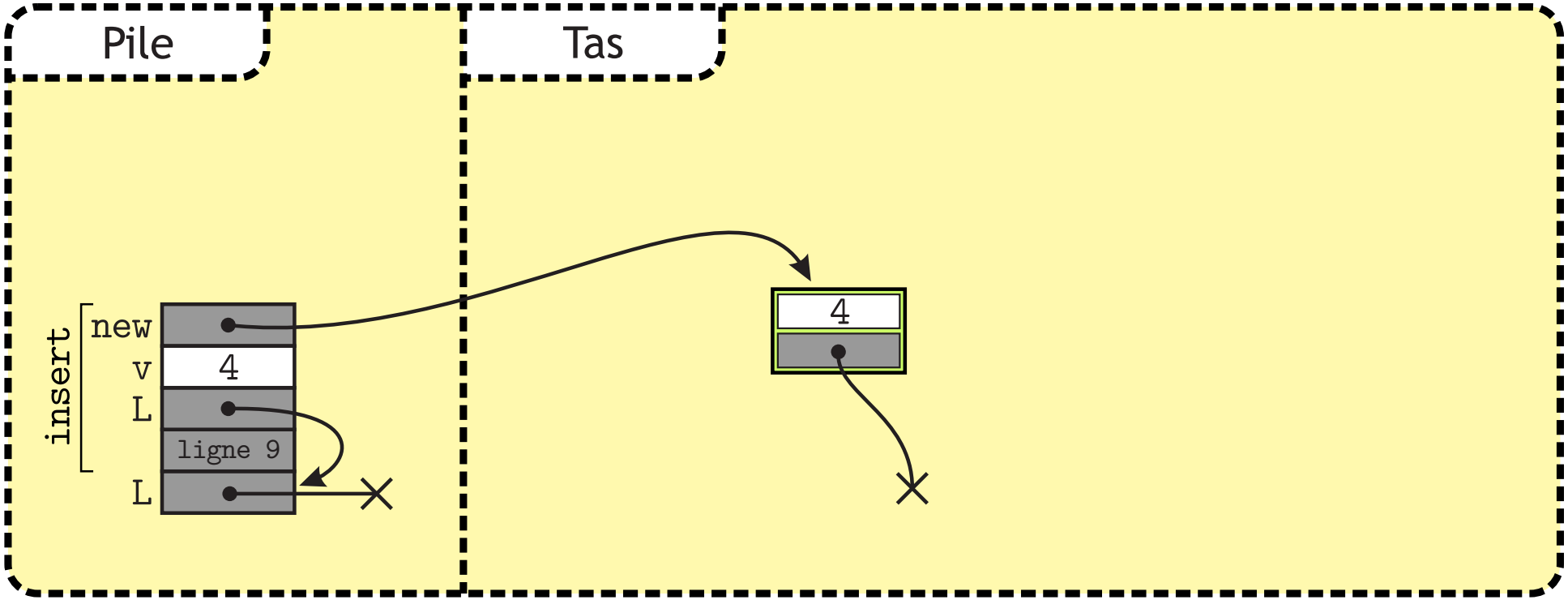


Utilisation de listes chaînées

Exemples en C

```
1 void insert(cell** L, int v) {
2   cell* new = malloc(sizeof(cell));
3 ▶ new->val = v;
4   new->next = *L;
5   *L = new;
6 }
```

```
7 int main() {
8   cell* L = NULL;
9   insert(&L, 4);
10  insert(&L, 7);
11 }
```

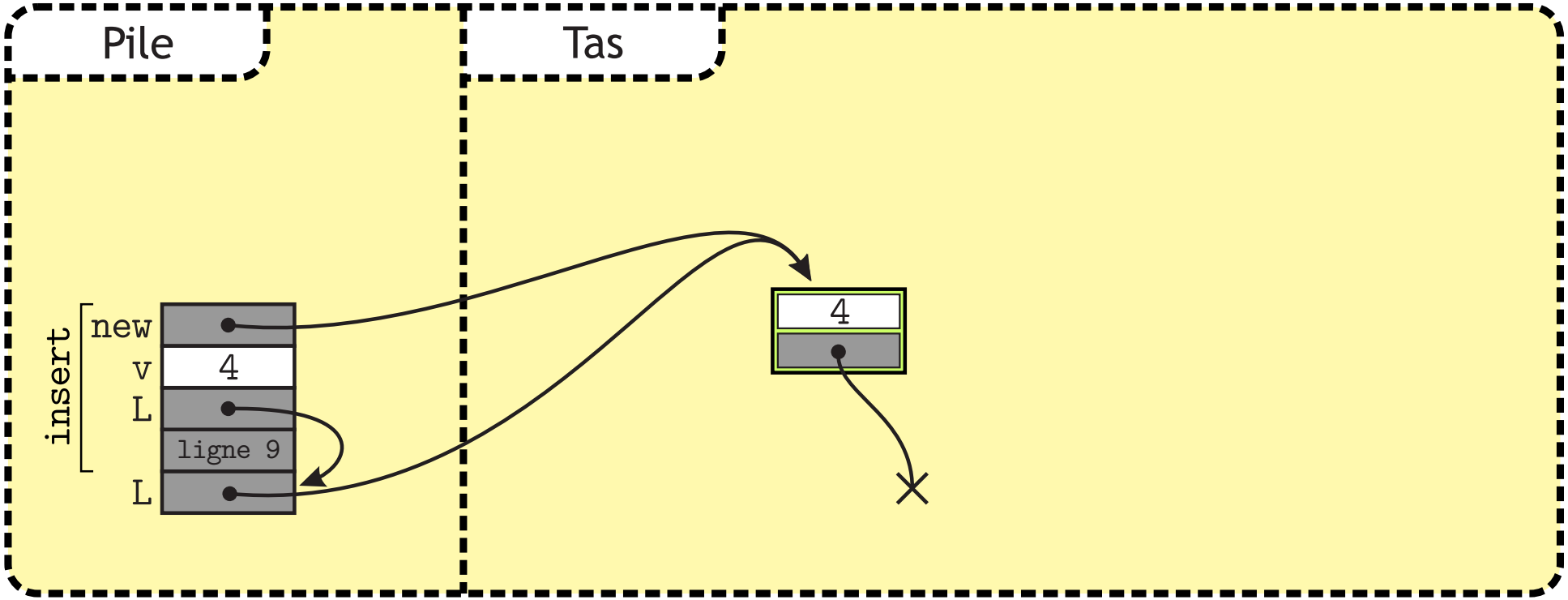


Utilisation de listes chaînées

Exemples en C

```
1 void insert(cell** L, int v) {
2   cell* new = malloc(sizeof(cell));
3   new->val = v;
4   new->next = *L;
5 ▶ *L = new;
6 }
```

```
7 int main() {
8   cell* L = NULL;
9   insert(&L, 4);
10  insert(&L, 7);
11 }
```

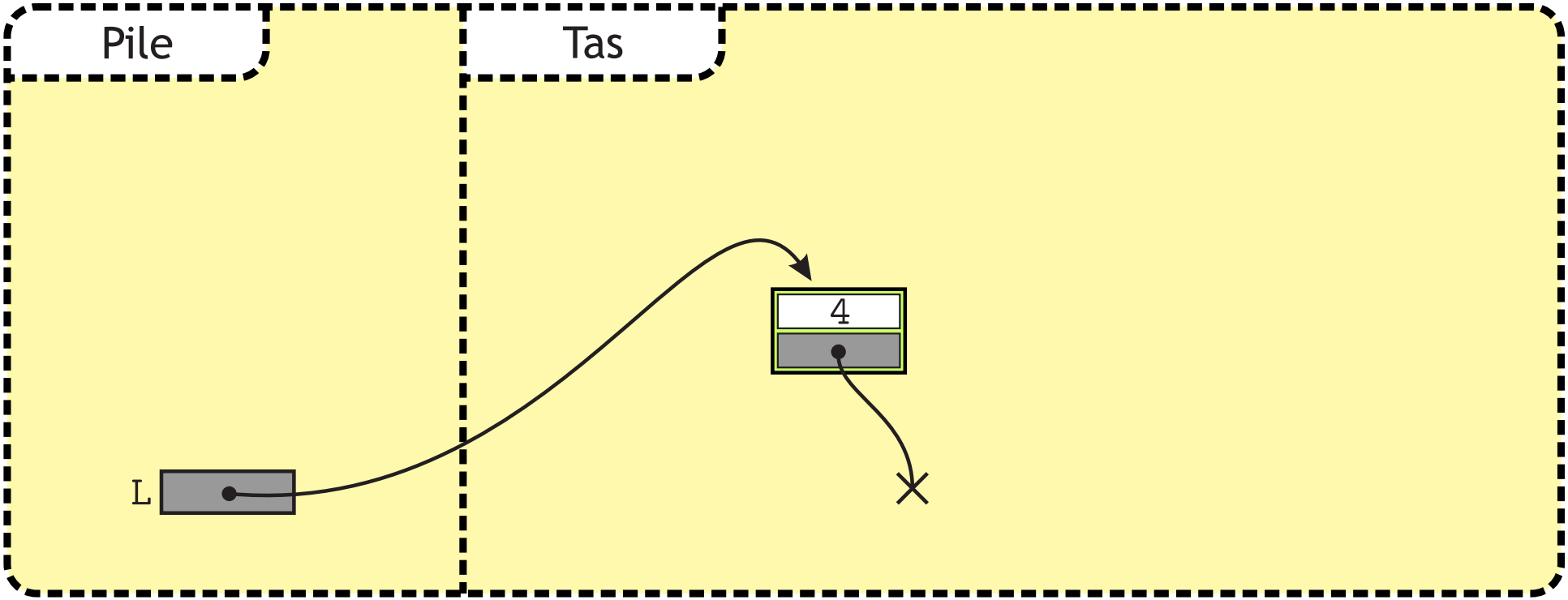


Utilisation de listes chaînées

Exemples en C

```
1  void insert(cell** L, int v) {
2    cell* new = malloc(sizeof(cell));
3    new->val = v;
4    new->next = *L;
5    *L = new;
6  }
```

```
7  int main() {
8    cell* L = NULL;
9    insert(&L, 4);
10 ▶ insert(&L, 7);
11 }
```

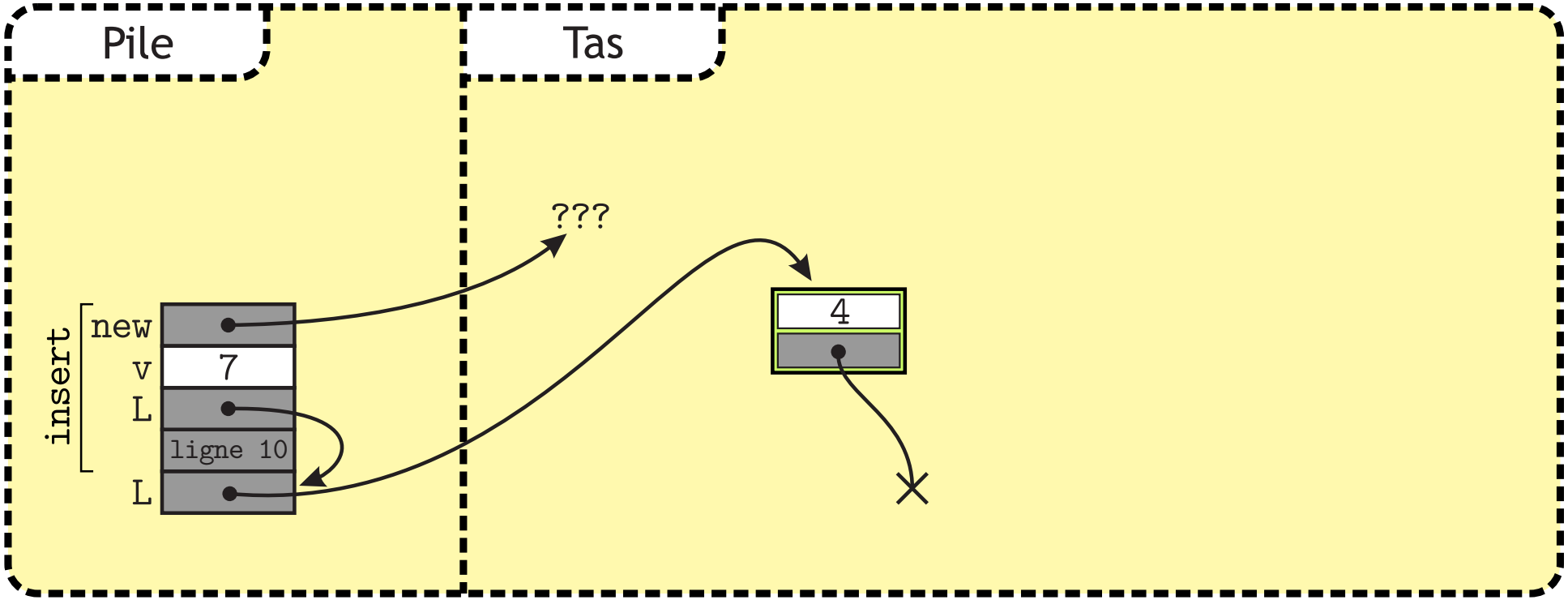


Utilisation de listes chaînées

Exemples en C

```
1  void insert(cell** L, int v) {
2    cell* new = malloc(sizeof(cell));
3    new->val = v;
4    new->next = *L;
5    *L = new;
6  }
```

```
7  int main() {
8    cell* L = NULL;
9    insert(&L, 4);
10 ▶ insert(&L, 7);
11 }
```

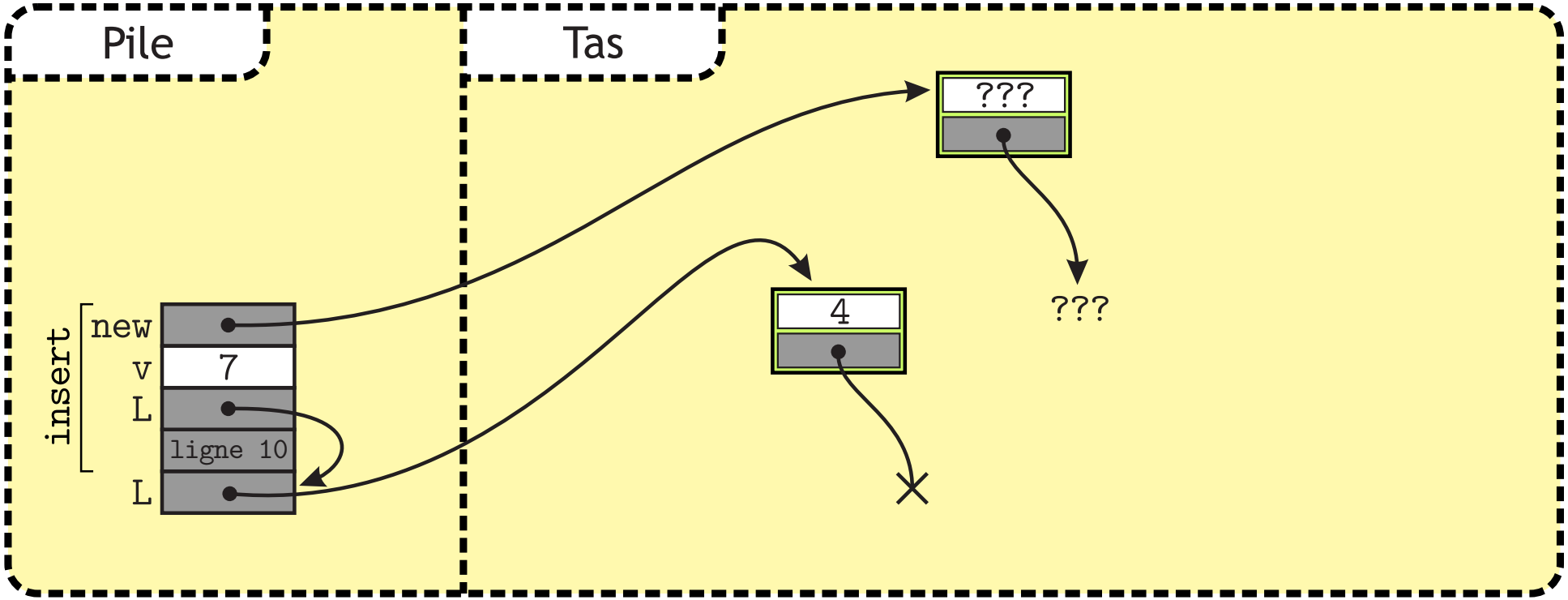


Utilisation de listes chaînées

Exemples en C

```
1  void insert(cell** L, int v) {
2  ▶ cell* new = malloc(sizeof(cell));
3    new->val = v;
4    new->next = *L;
5    *L = new;
6  }
```

```
7  int main() {
8    cell* L = NULL;
9    insert(&L, 4);
10   insert(&L, 7);
11 }
```

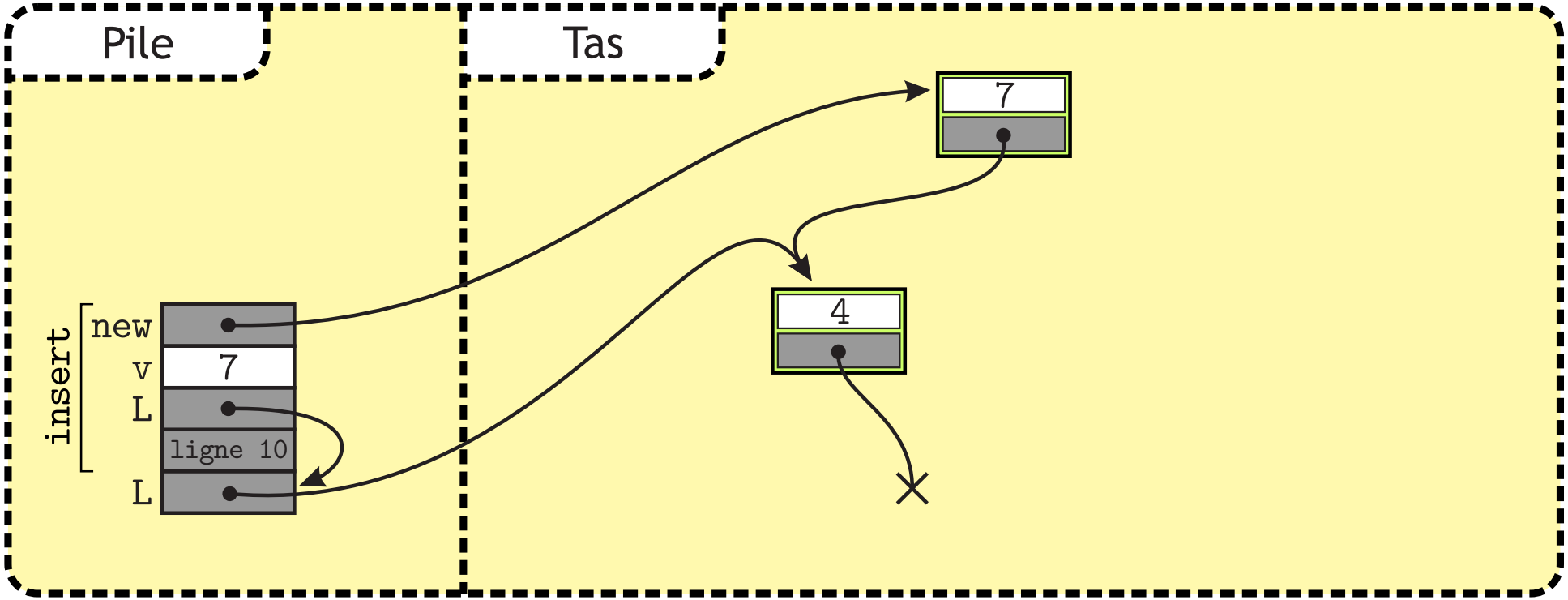


Utilisation de listes chaînées

Exemples en C

```
1  void insert(cell** L, int v) {
2    cell* new = malloc(sizeof(cell));
3  ▶ new->val = v;
4    new->next = *L;
5    *L = new;
6  }
```

```
7  int main() {
8    cell* L = NULL;
9    insert(&L, 4);
10   insert(&L, 7);
11 }
```

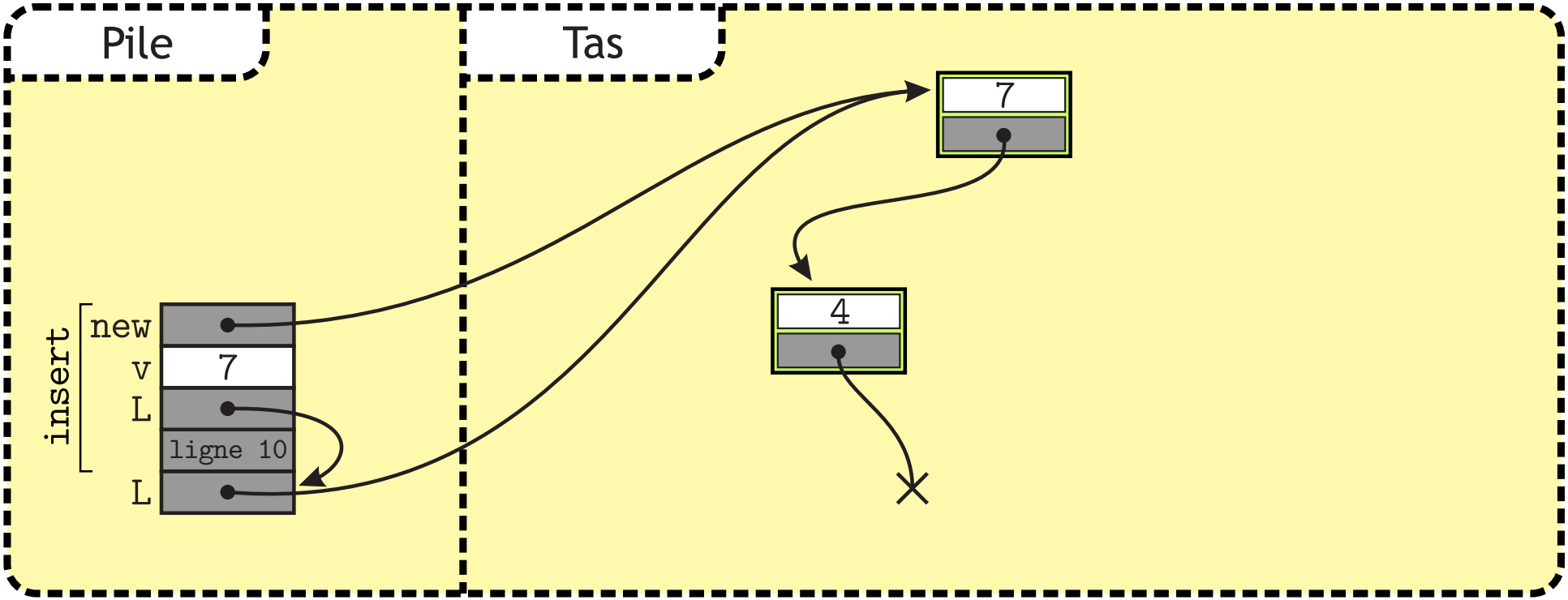


Utilisation de listes chaînées

Exemples en C

```
1  void insert(cell** L, int v) {
2    cell* new = malloc(sizeof(cell));
3    new->val = v;
4    new->next = *L;
5  ▶ *L = new;
6  }
```

```
7  int main() {
8    cell* L = NULL;
9    insert(&L, 4);
10   insert(&L, 7);
11 }
```

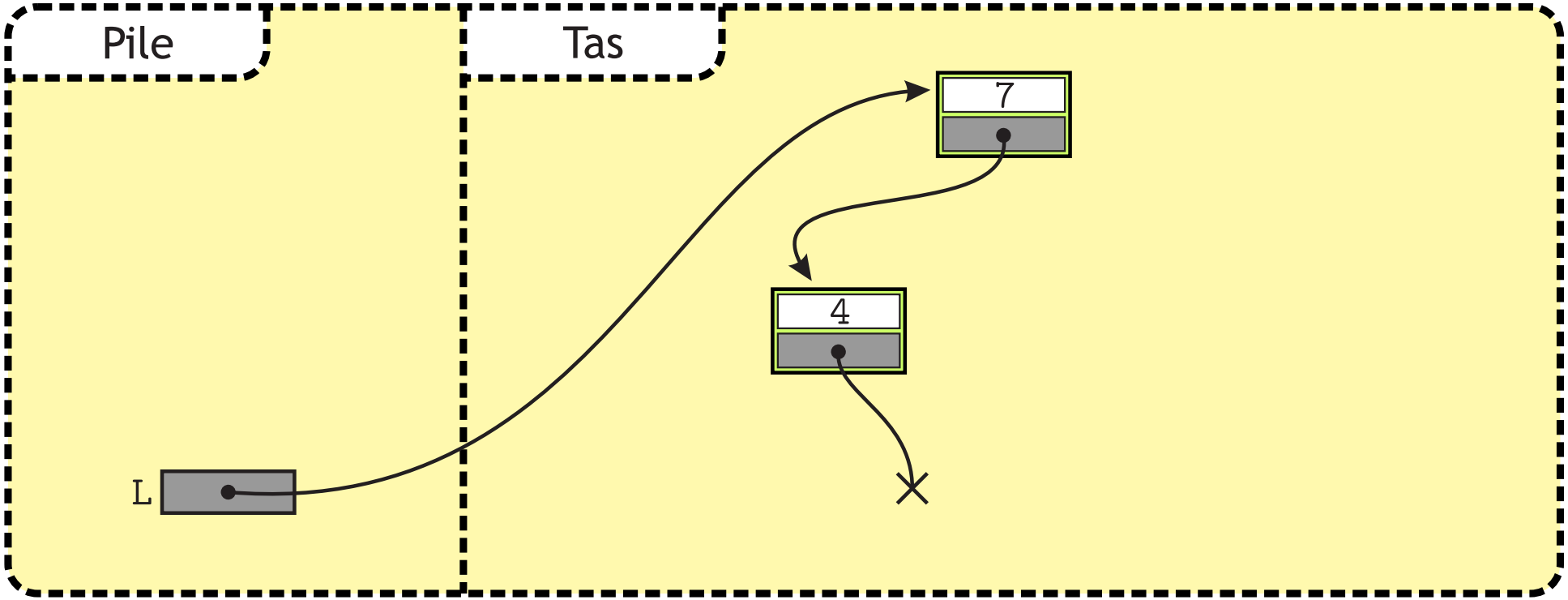


Utilisation de listes chaînées

Exemples en C

```
1  void insert(cell** L, int v) {
2    cell* new = malloc(sizeof(cell));
3    new->val = v;
4    new->next = *L;
5    *L = new;
6  }
```

```
7  int main() {
8    cell* L = NULL;
9    insert(&L, 4);
10   insert(&L, 7);
11 }
```



Utilisation de listes chaînées

Exemples en C

- La suppression se passe de façon similaire
  - ⚠ il ne faut pas oublier le `free` de la case supprimée
    - → sauvegarder un pointeur vers la case à libérer.

suppression

```
1  void delete_one(cell** L) {
2    cell* tmp = *L;
3    if ((*L) == NULL) {
4      printf("Liste vide !\n");
5      return;
6    }
7    *L = (*L)->next;        // la suppression est ici
8    free(tmp);
9  }
10
11 int main() {
12   cell* L = NULL;
13   insert(&L, 4); insert(&L, 7);
14   delete_one(&L);
15 }
```

Utilisation de listes chaînées

Exemples en C

- Pour afficher le contenu de la liste chaînée, il faut la parcourir :
  - soit avec une boucle `while` et un “curseur”,
  - soit de façon récursive (terminale).

affichage

```
1  void print_iter(cell* L) {
2    cell* cur = L;
3    while (cur != NULL) {
4      printf("%d, ", cur->val);
5      cur = cur->next;
6    }
7    printf("\n");
8  }
9  void print_rec(cell* L) {
10   if (L == NULL) {
11     printf("\n");
12     return;
13   }
14   printf("%d, ", L->val);
15   print_rec(L->next);
16 }
```

Utilisation de listes chaînées

Exemples en C

- Pour libérer entièrement le contenu de la liste chaînée :
  - on peut encore faire de l'itératif (très efficace),
  - mais la méthode récursive est plus courte à écrire

    ⚠ il faut bien faire l'appel récursif avant le `free`.

libération

```
1 void free_list(cell** L) {
2   if ((*L) == NULL) {
3     return;
4   }
5   free_list((*L)->next);
6   free(*L);
7   *L = NULL;
8 }
```

- L'algorithme commence par faire tous les appels récursifs
  - quand il arrive au bout de la liste il libère en partant de la fin,
  - cela utilise donc de la mémoire dans la pile.

Variantes des listes chaînées

- Il existe beaucoup de variantes de listes chaînées :
  - listes doublement chaînées : pour avancer et reculer, mais plus de pointeurs à mettre à jour,
  - listes circulaires : le dernier élément pointe sur le premier,
  - listes à sentinelles : au lieu de pointer vers `NULL` la liste peut contenir un dernier élément spécial appelé sentinelle.

- Si pour une application spécifique une liste simple ne suffit pas :
  - ne pas hésiter à ajouter des données dans chaque cellule,
  - la liste peut aussi contenir des données (nombre d'éléments...) en plus du pointeur vers le premier élément,
  - ⚠ il faut que les opérations de base restent efficaces (en $\Theta(1)$) !

Piles et files avec des listes chaînées

- Il est aussi possible d'implémenter efficacement une pile ou une file avec une liste chaînée
  - il n'y a pas de contraintes de taille comme avec un tableau.
- La pile est le plus simple :
  - `push` consiste à insérer un élément en début de liste,
  - `pop` lit le contenu du premier élément et le supprime,
    - → les opérations se font bien en $\Theta(1)$.
- Pour la file c'est un peu plus compliqué :
  - il faut insérer les éléments à un bout et les supprimer à l'autre,
  - on ne sait pas reculer d'un élément, juste passer au suivant.
    - → on insère à la fin (nécessite un pointeur supplémentaire) et on retire du début.

Piles et files avec des listes chaînées

Implémentation d'un file

queue.c

```
1  typedef struct {          // nouvelle structure de file
2    cell* beg;              // pointeur sur la première case
3    cell* end;              // pointeur sur la dernière case
4  } queue;
5
6  void push(queue* F, int v) {
7    cell* new = (cell*) malloc(sizeof(cell));
8    new->val = v;
9    new->next = NULL;
10   if (F->end == NULL) {   // si la file est vide
11     F->beg = new;         // le dernier est aussi le premier élément,
12     F->end = new;
13   } else {                // sinon :
14     F->end->next = new;   // - ajout de l'élément à la fin
15     F->end = new;         // - on met à jour la fin
16   }
17 }
```

Piles et files avec des listes chaînées

Implémentation d'un file

queue.c

```
1  int pop(queue* F) {
2    int res;
3    cell* tmp;
4    if (F->beg == NULL) {
5      printf("File vide !\n");
6      return -1;
7    } else if (F->beg == F->end) {  // s'il n'y a qu'une case
8      F->end = NULL;                // la file sera vide à la fin
9    }
10   res = F->beg->val;              // on sauvegarde la valeur
11   tmp = F->beg;                   // on sauvegarde le pointeur
12   F->beg = F->beg->next;          // on avance le début d'une case
13   free(tmp);                      // on libère la case extraite
14   return res;
15 }
```

- Le code parait compliqué, mais les opérations sont très simples
  - les listes sont très efficaces pour implémenter des files.

Ce qu'il faut retenir de ce cours

- Les structures dynamiques permettent de stocker une quantité variable d'éléments :
  - permettent de n'utiliser que la quantité de mémoire nécessaire,
  - pour les algorithmes dont le comportement est dur à prévoir.

- Il faut bien distinguer (même si les deux sont très liés) :
  - la description algorithmique de la structure
    - → la complexité des opérations possibles,
  - l'implémentation de la structure
    - → comment on se débrouille pour obtenir ces complexités.

- Les structures dynamiques les plus simples sont :
  - le tableau dynamique : accès direct au i-ème élément,
  - la liste chaînée : très flexible et utile partout,
  - la pile et la file : opérations `push` et `pop`.