Représentation chainée des séquences

1. Notion de pointeur
2. Représentation d'une séquence à l'aide de pointeurs : Listes chaînées
3. *Simulation de listes chainées dans un tableau*

Notion de pointeur

- Nous introduisons la notion de pointeur pour décrire la création dynamique d'informations en mémoire.
- Nous considérons que l'espace mémoire est géré par le système.

Notion de pointeur

- Un pointeur est une information contenant une adresse en mémoire. Nous allons donc pouvoir :
  - manipuler des adresses sans connaître leurs valeurs,
  - définir des variables dont les valeurs sont des adresses,
  - disposer de primitives de gestion de l'espace mémoire.
- On complète la notation algorithmique par le constructeur de type de la forme :

  pointeur de T où T est un nom de type

Utilisation du constructeur pointeur de T

Définitions utilisées dans les lexiques :

- Variable x : pointeur de Nomdetype
- Type AdT : type pointeur de Nomdetype

Exemple :

Lexique partagé

AdEntier : type pointeur d'entier

Lexique d'un algorithme

a : AdEntier // a est l'adresse d'un entier qui sera créé dynamiquement

- Une valeur particulière d'adresse de nom NIL (du latin nihil) permet de noter un lien vers nulle part (adresse nulle)

Opérations sur les pointeurs

- On utilise l'opérateur ↑ pour décrire l'accès indirect à une valeur, c'est-à-dire l'accès par l'intermédiaire du pointeur.
  - Soit a le nom d'une variable de type pointeur de T
  - a↑ désigne la valeur de type T pointée par a

    en langage C, noté *a
- L'opération ↑ est appelée « déréférencement »
- Inversement, on utilise l'opérateur @ pour définir l'adresse associée à un nom :
  - Soit v le nom d'une variable de type T
  - @v désigne l'adresse associée à v, une valeur de type pointeur de T

    en langage C, noté &v
- L'opération @ est appelée « référencement »
- Propriétés :
  - a de type pointeur de T, on a : $a \equiv @(a\uparrow)$
  - b de type T, on a : $b \equiv (@b)\uparrow$

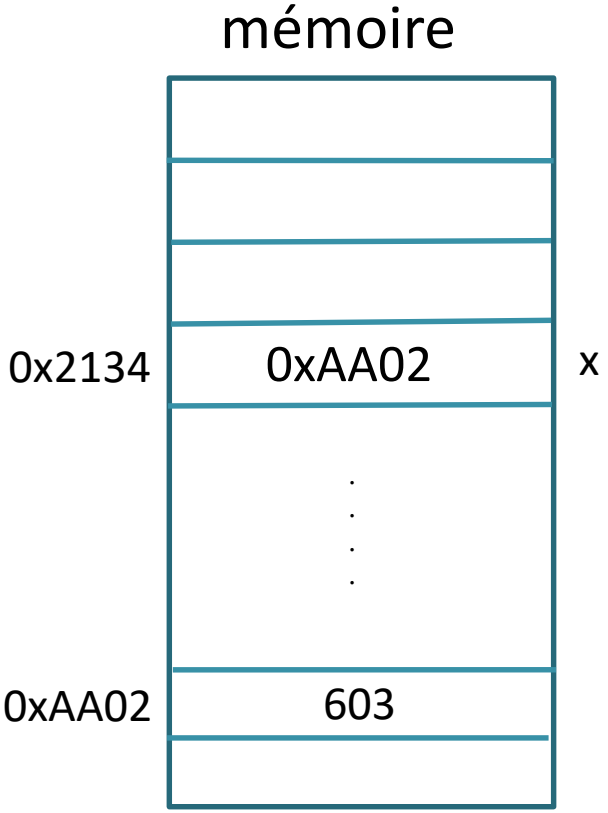
Opérations sur les pointeurs

x : pointeur d'entier

@x = 0x2134

x = 0xAA02

x↑ = 603



Primitives de d'allocation et de libération de la mémoire : créer et détruire

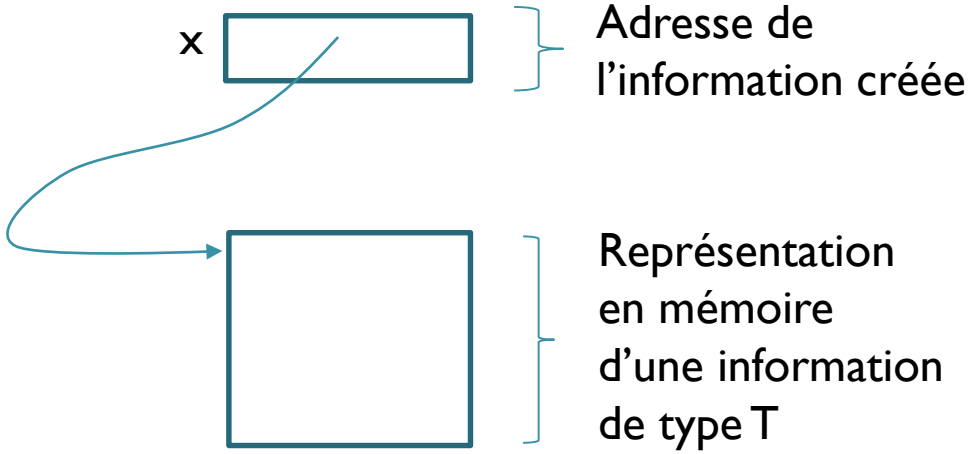
action **créer** (Elaboré x : pointeur de T)
// Effet : crée en mémoire une information de type T et mémorise dans x son adresse
// E.I. : indifférent
// E.F.: x = adresse en mémoire de l'information de type T créée

action **détruire** (Consulté x : pointeur de T)
// Effet : détruit l'information de type T pointée par x
// E.I. : x = adresse de l'information de type T à détruire
// E.F.: x inchangé, l'espace mémoire qui était occupé par l'information pointée par x est libéré

Opérations sur les pointeurs

Allocation et libération de la mémoire : créer et détruire

x : pointeur de T



créer(x)

détruire(x)

Représentation d'une séquence à l'aide de pointeurs

- La représentation contiguë d'une séquence ordonnée (table) nécessite des décalages pour réaliser les insertions et suppressions.
- Le coût d'un décalage varie selon la position de modification.
- Pour éviter les déplacements d'éléments dans l'espace mémoire, et pour rendre le coût indépendant de la position de modification, on supprime la contrainte de contiguïté :
  - les éléments de la séquence sont dispersés en mémoire
  - on tabule l'adresse du successeur au lieu de la calculer comme précédemment : à chaque élément de la séquence est associée l'adresse de son successeur, s'il existe. De plus, on connaît l'adresse du premier élément.
  - On peut alors réaliser les insertions et les suppressions uniquement par modification de l'adresse du successeur, également appelée lien de chainage. Ainsi, il n'y a plus de déplacements d'éléments dans l'espace mémoire.

Notion de chainage

- Pour représenter une séquence en mémoire, on mémorise l'adresse du successeur de chaque élément.
- Les éléments de la séquence sont représentés en mémoire par des Doublets composés d'un Elément et d'une Adresse :

```
Doublet : type agrégat
        el : Elément
        suiv : Adresse  // pointe sur le doublet suivant
fagrégat
```

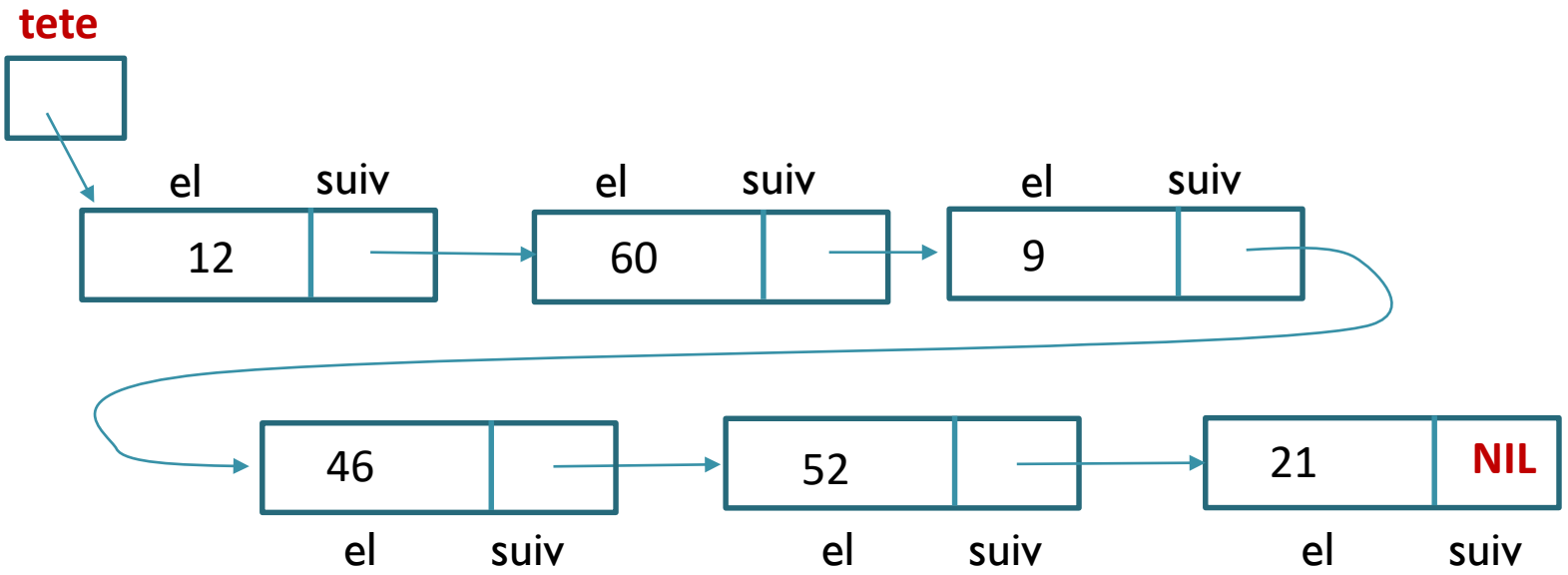
- Chaque élément de la séquence est chainé (lié) à son successeur.
- Une telle séquence est appelée liste chainée.

Représentation d'une séquence à l'aide de pointeurs : liste chainée

- Un Elément e de l'ensemble est rangé dans un Doublet d'adresse a.
- On a alors :
  - a↑.el = e
  - a↑.suiv = b, où b est défini comme suit :
    - Si e n'est pas le dernier élément de la séquence, b est l'adresse du Doublet contenant le successeur de e dans la séquence.
    - Si e est le dernier élément, b a la valeur NIL, utilisée pour noter la fin de la séquence.
- Pour pouvoir accéder au premier élément de la séquence, on définit dans le lexique tete une variable contenant l'adresse du premier élément de la séquence. On l'appelle tête de la séquence.
- Une séquence vide est caractérisée par tete = NIL.

tete : pointeur de Doublet

Schématisation d'une séquence chaînée d'entiers



Définition d'une liste chainée

Lexique partagé :

Doublet : type agrégat
el : Elément
suiv : AdDoublet
fagrégat

AdDoublet : type pointeur de Doublet

Lexique de l'algorithme qui manipule la liste

tete : AdDoublet // tête d'une liste de Doublets

Accès à la liste chaînée

Lexique partagé

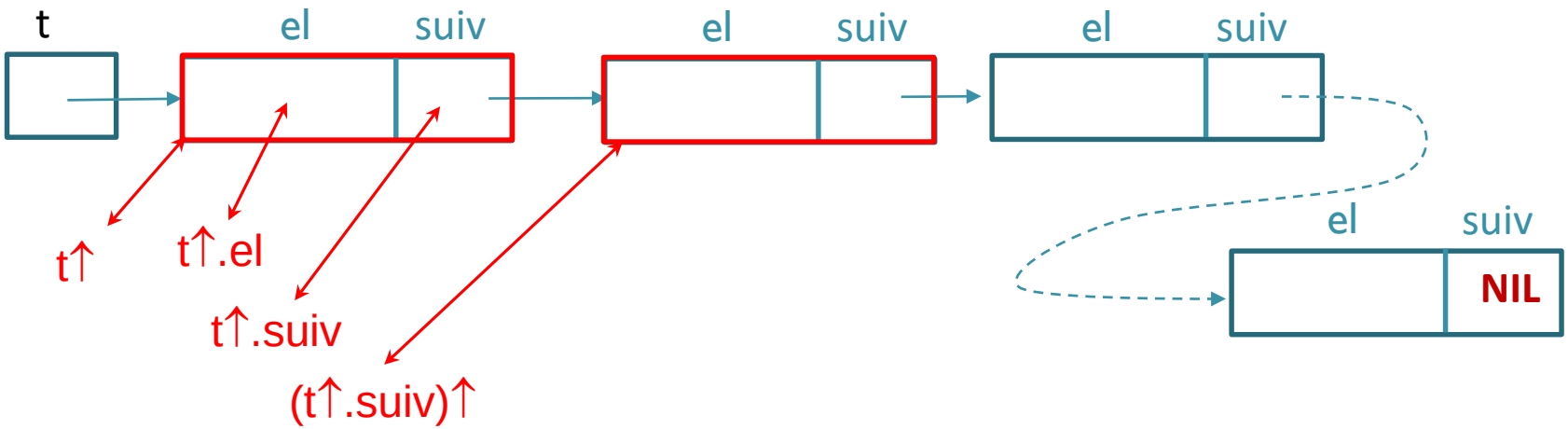
Elément : type …

AdDoublet : type pointeur de Doublet

Doublet : type agrégat el : Elément ; suiv : AdDoublet fagrégat

Lexique principal

t : AdDoublet // tête de la liste

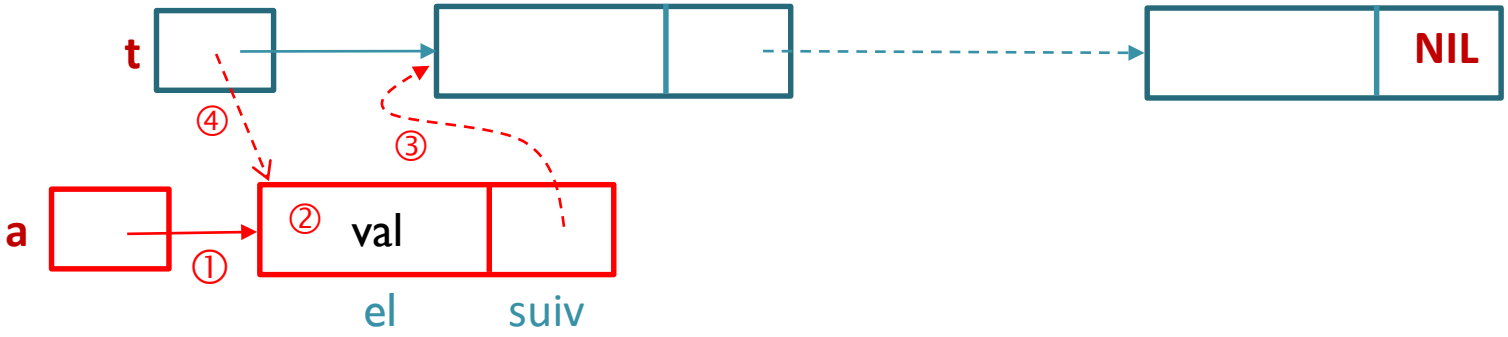


Schémas de modification d'une liste chainée

La liste représentant un ensemble d'éléments organisés en séquence, nous retrouvons les opérations habituelles associées :

- Création d'une liste vide
- Ajout d'un élément (doublet)
- Suppression d'un élément
- Modification d'un élément

Ajout d'un élément en tête



① créer(a)

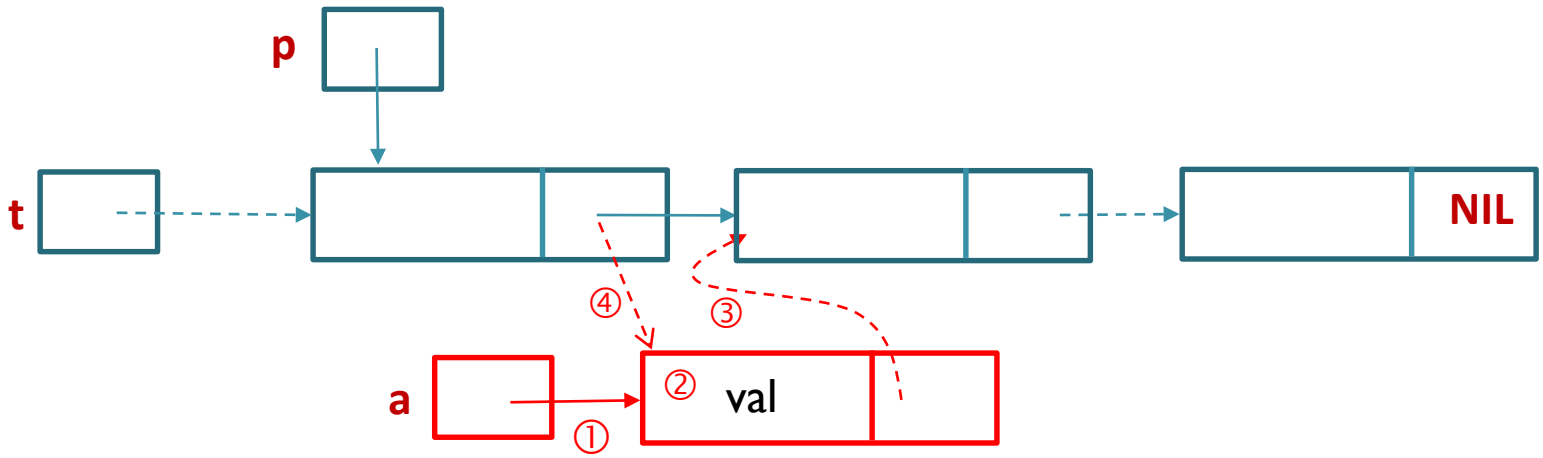
② a↑.el ← val

③ a↑.suiv ← t ;

④ t ← a

Le cas où t = NIL (liste vide) entre dans la cas général : a↑.suiv prend la valeur NIL

Ajout d'un élément après p



① créer(a)

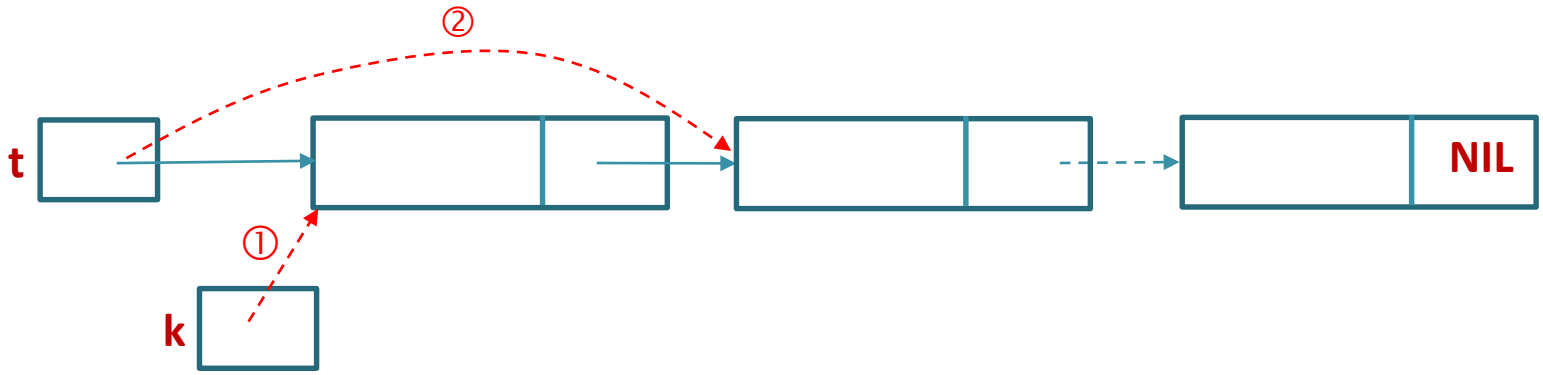
② a↑.el ← val

③ a↑.suiv ← p↑.suiv

④ p↑.suiv ← a

Le cas où p pointe sur la dernière cellule de la liste entre dans le cas général : a↑.suiv prend la valeur NIL

Suppression d'un élément en tête

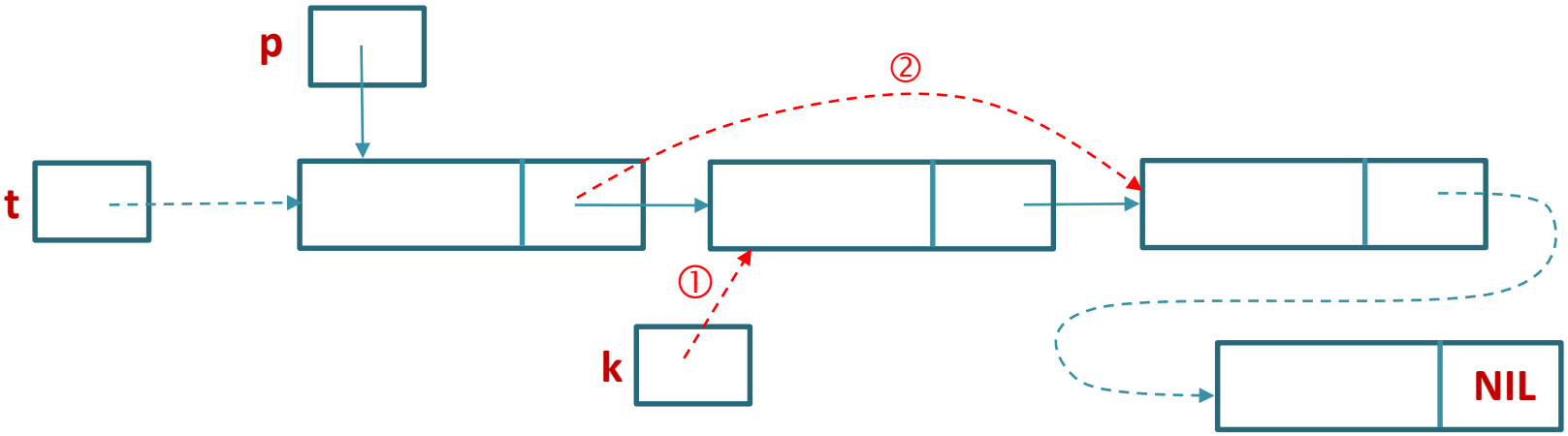


k ← t

t ← t↑.suiv

detruire(k)

Suppression de l'élément successeur de p



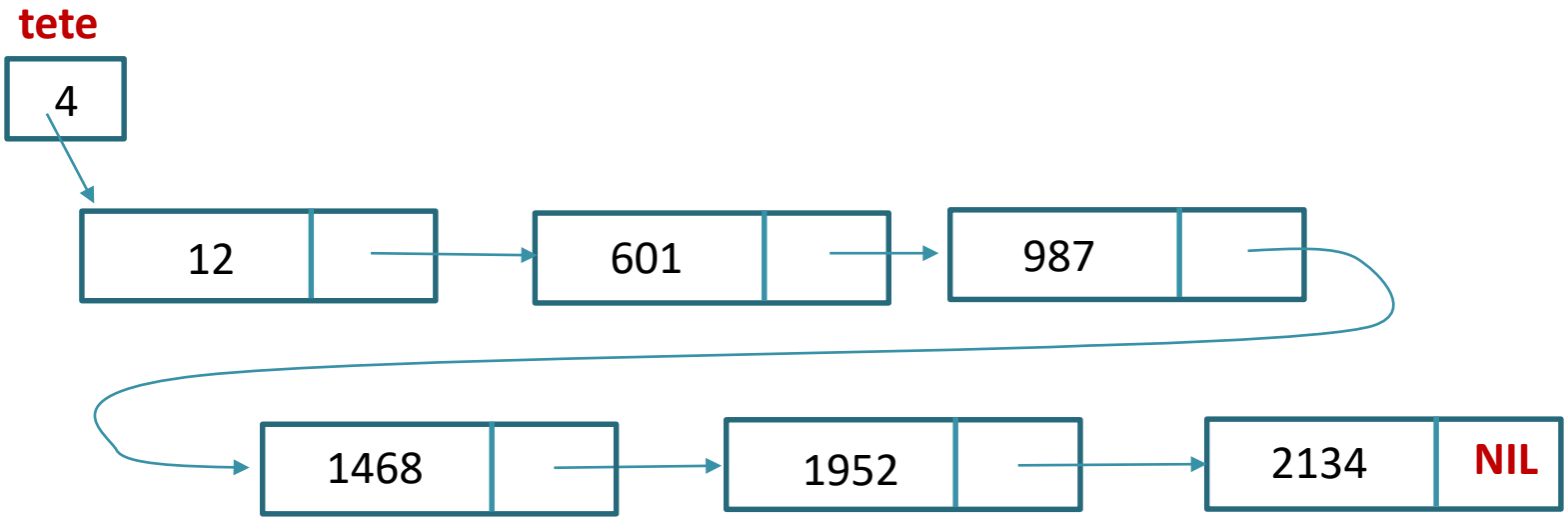
k ← p↑.suc

p↑.suc ← k↑.suc

detruire(k)

Le cas où p pointe l'avant dernière cellule (suppression de la dernière cellule) entre dans le cas général :
p↑.suc prend la valeur NIL

Schématisation d'une séquence chaînée d'entiers triée en ordre croissant

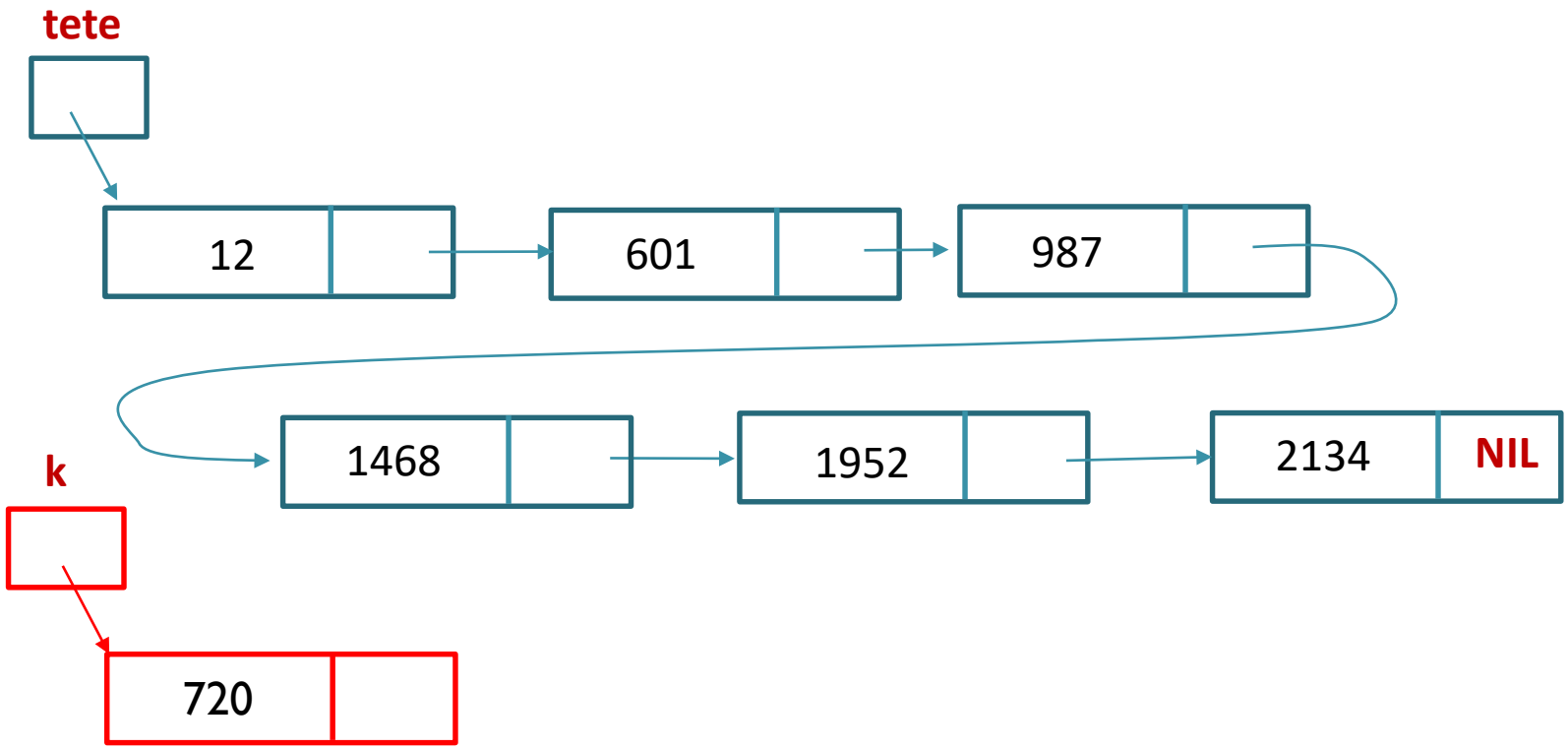


Insertion d'un élément dans la séquence triée

- L'**insertion** d'un nouvel élément **e** consiste à :
  1. Créer un Doublet d'adresse **k** et y placer e
  2. Déterminer l'adresse **p** du Doublet contenant l'élément qui doit précéder e dans la séquence
  3. Définir la valeur du **successeur de k** : l'ancien successeur de **p**
  4. Définir la valeur du **successeur de p** : k

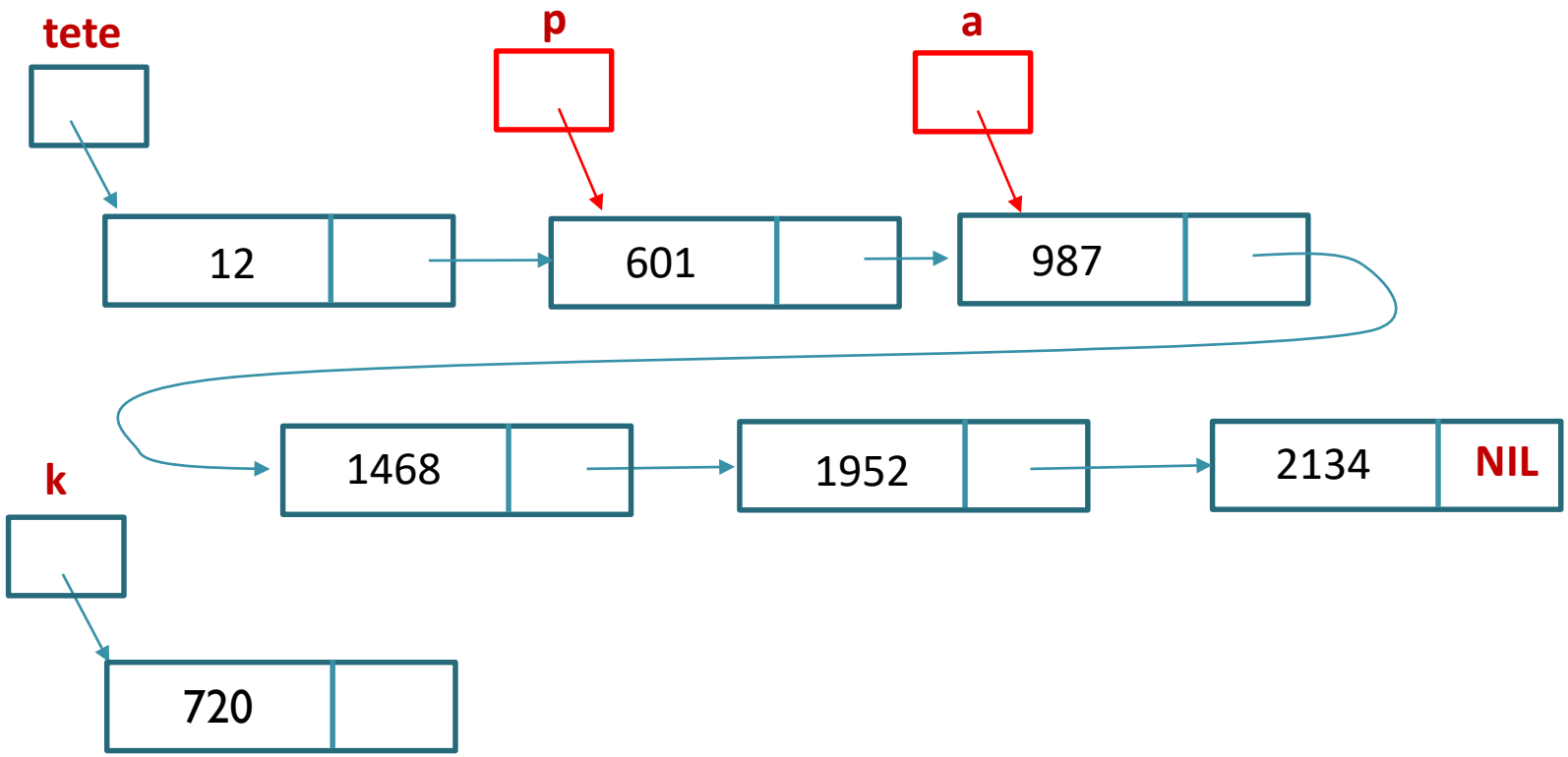
Ajout de l'élément 720 à la séquence

1 - Allouer un doublet d'adresse **k** et y ranger 720



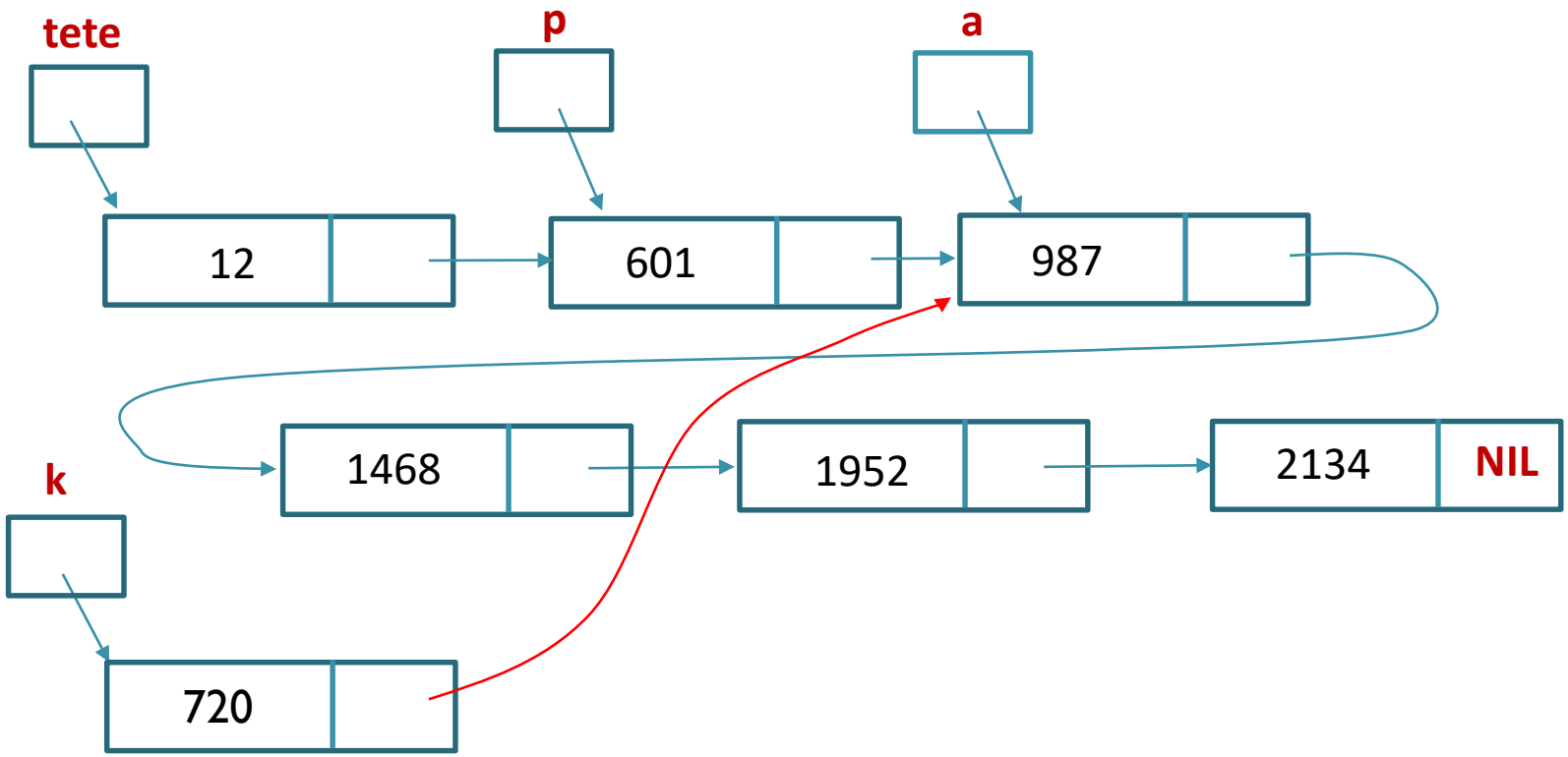
Ajout de l'élément 720 à la séquence

2 - déterminer **p** l'adresse du doublet qui précède l'élément inséré, et **a** l'adresse du doublet qui le suit



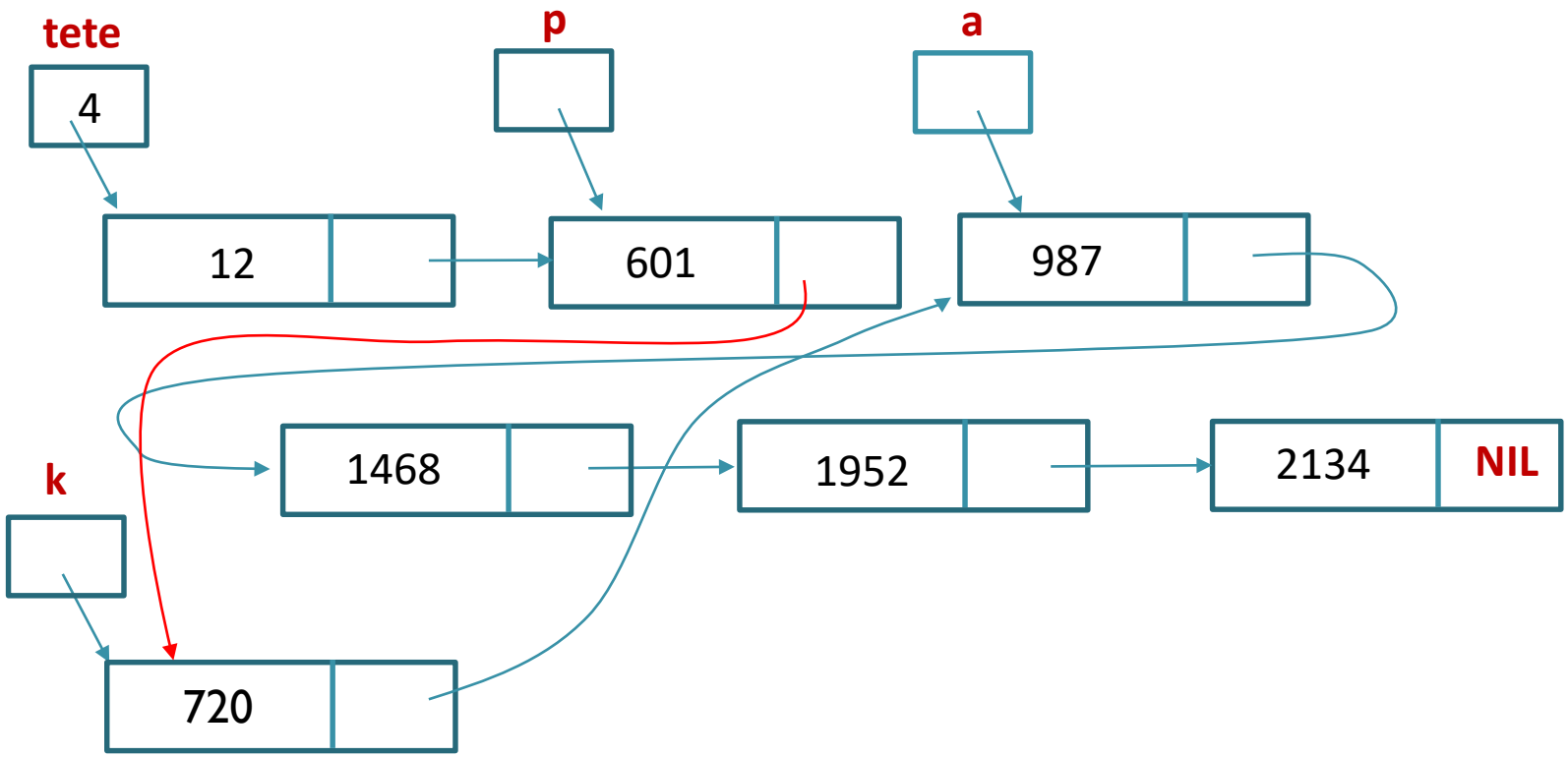
Ajout de l'élément 720 à la séquence

3 - déterminer la valeur du successeur de k : l'ancien successeur de p



Ajout de l'élément 720 à la séquence

4 – déterminer la valeur du successeur de p : k



Algorithme d'insertion d'un élément **e** dans la séquence triée

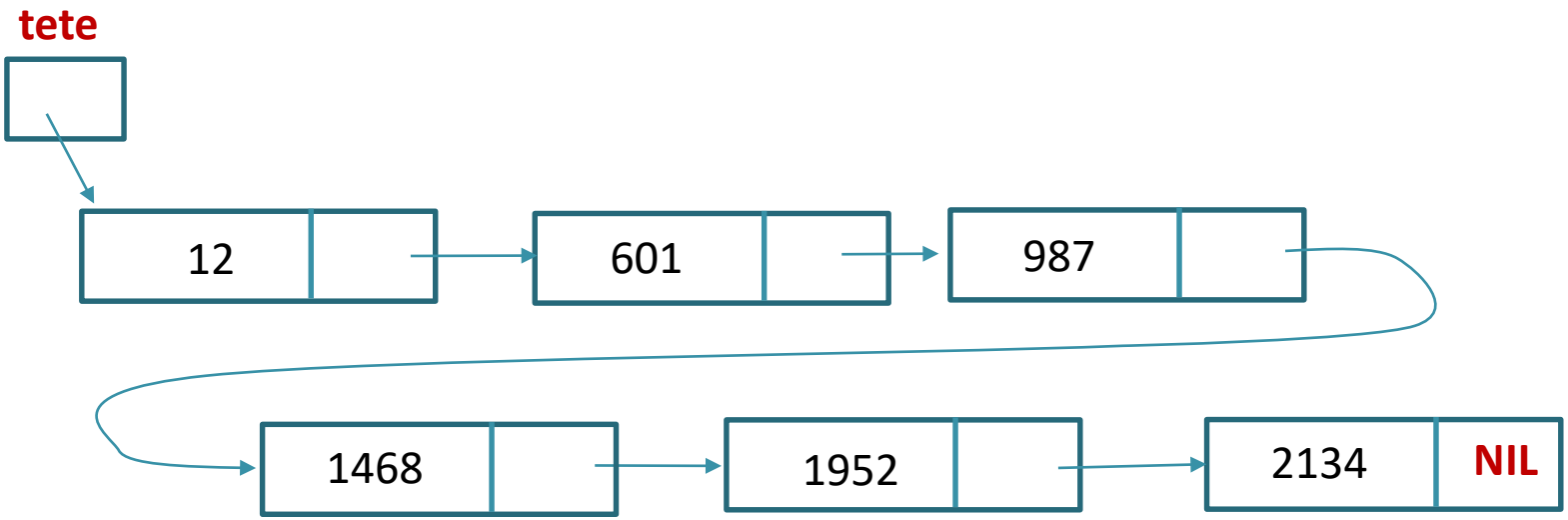
```
action insérer(Consulté e : Element)
// Effet : ajout de l'élément e à la séquence triée (en ordre croissant) de tête tete
// E.I. : tete = adresse du premier élément de la liste ou NIL si liste vide
// E.F.: l'élément e a été ajouté à la liste de tête tete
lexique de insérer
  a, p : AdDoublet  // pointeurs sur l'élément courant, l'élément précédent
  k : AdDoublet     // point sur le Doublet inséré dans la liste

algorithme de insérer
      créer(k)
      k↑.el ← e
      a ← tete ; p ← NIL
      tantque a ≠ NIL etpuis  a↑.el < e faire
          p ← a ; a ← a↑.suiv
      ftq
      // a = NIL oualors a↑.el ≥ e : on place e avant a => après p
      selon p
        p = NIL : k↑.suiv ← tete ; tete ←  k  // insertion en tête de la liste
        p ≠ NIL : k↑.suiv ← a                 // a = p↑.suiv
                  p↑.suiv ← k                 // insertion après le doublet d'adresse p
      fselon
```

Suppression d'un élément

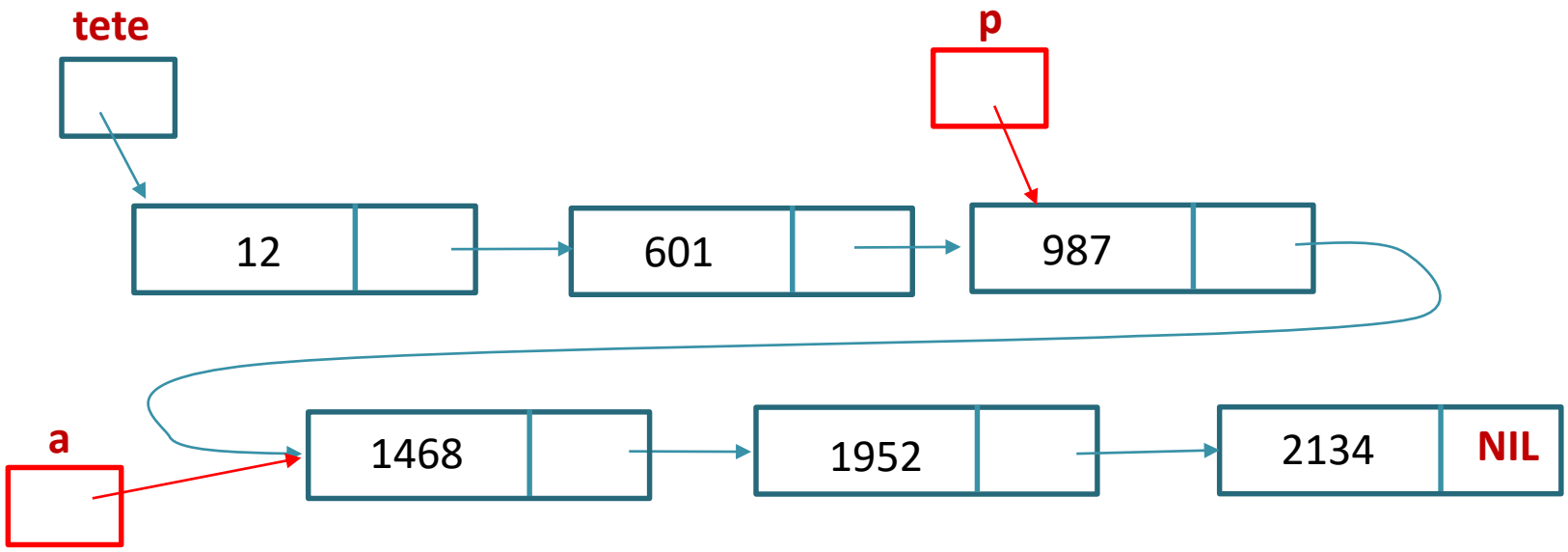
- La **suppression** d'un élément **e** consiste à :
  1. Déterminer l'adresse **a** du Doublet contenant e et l'adresse son prédécesseur **p**
  2. Redéfinir la valeur du **successeur de p** : l'ancien successeur de a
  3. Détruire le Doublet a

Supprimer l'élément 1468 de la séquence



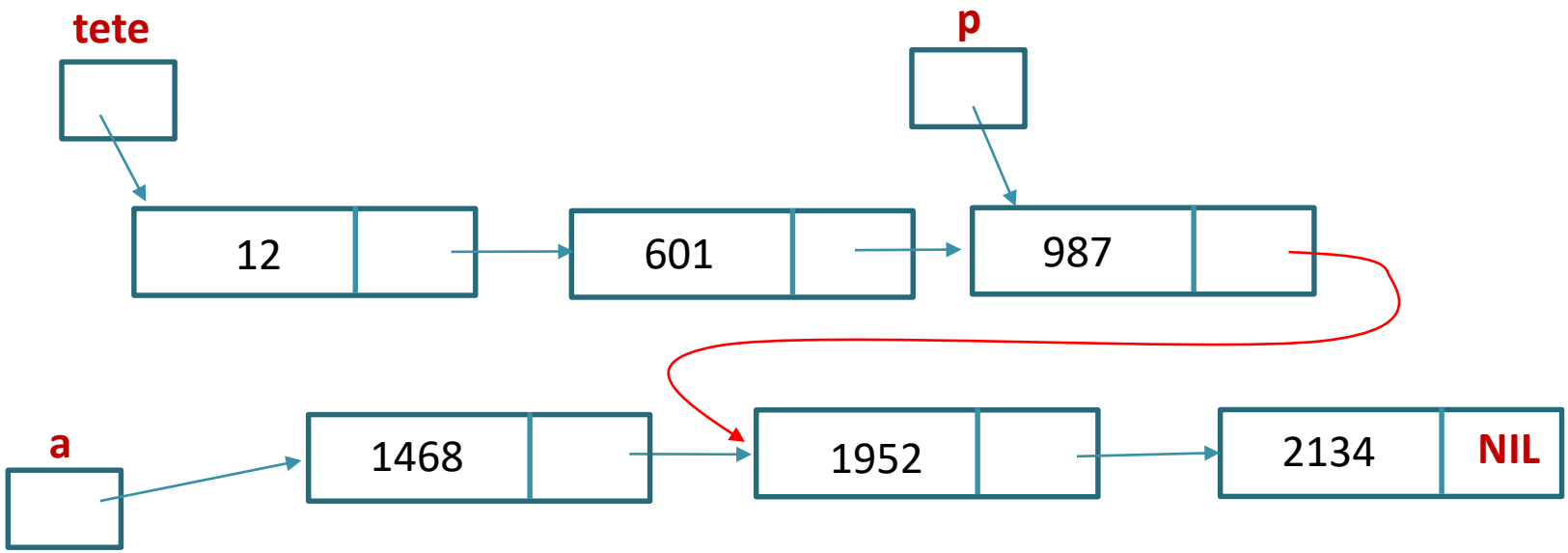
Supprimer l'élément 1468 de la séquence

1 – Déterminer l'adresse **a** du doublet à supprimer et l'adresse de son prédécesseur **p**



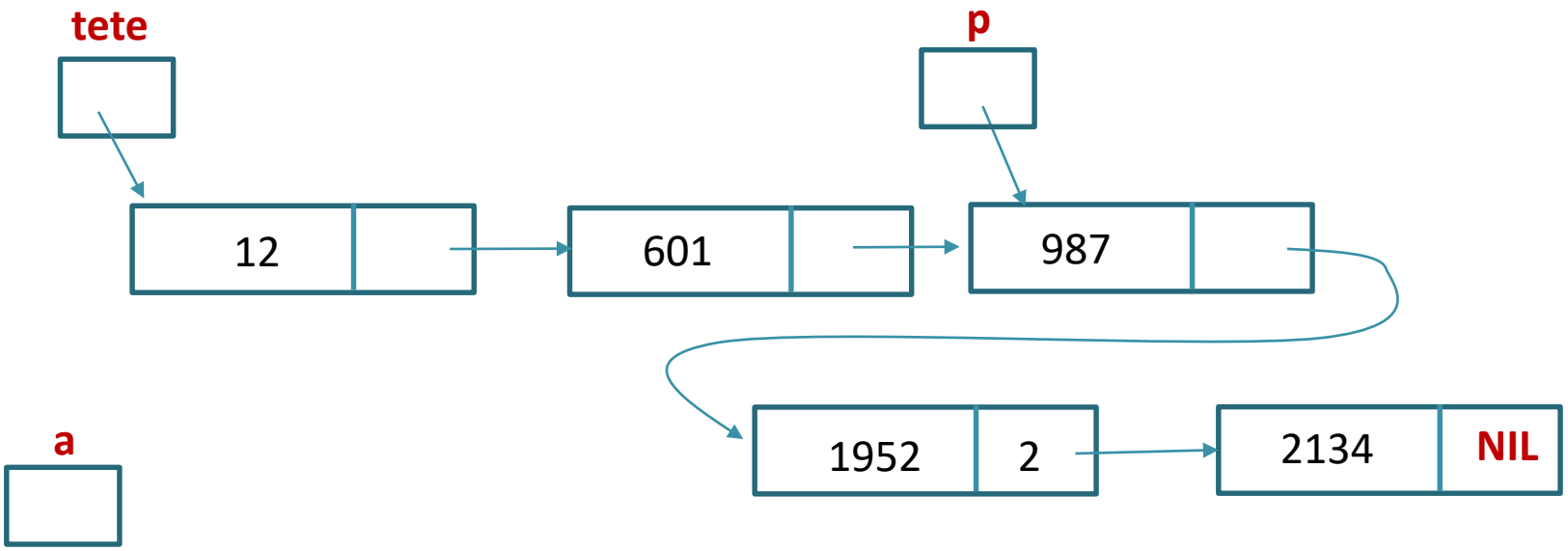
Supprimer l'élément 1468 de la séquence

2 – Redéfinir la valeur du **successeur de p** : l'ancien successeur de a



Supprimer l'élément 1468 de la séquence

3 – Détruire le Doublet d'adresse a

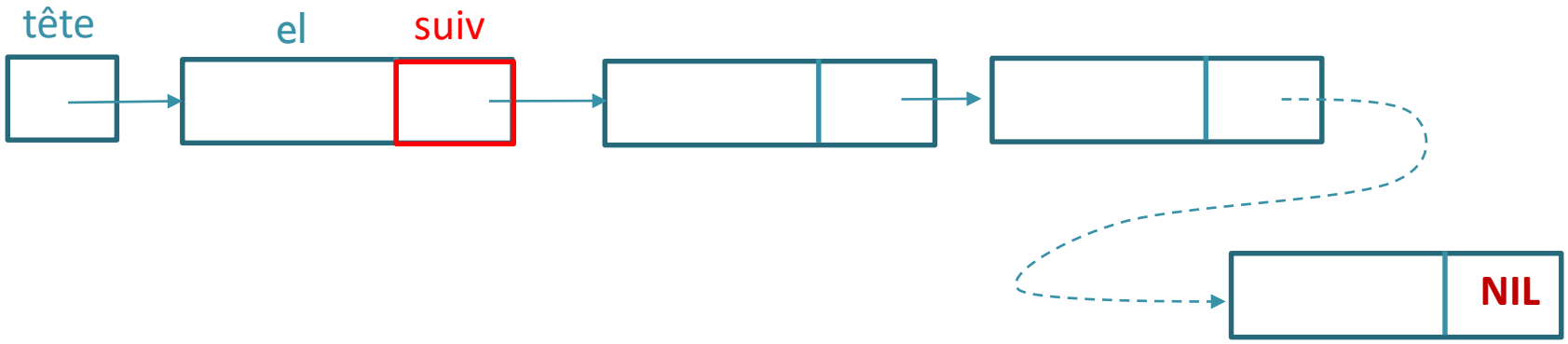


Algorithme de suppression d'un élément **e** de la séquence

```
action supprimer(Consulté e : Elément)
// Effet : supprime l'élément e de la séquence triée de tête tete
// E.I. : indifférent
// E/F.: l'élément e est supprimé de la liste triée,
//         si e n'est pas trouvé, la liste est inchangée
lexique de supprimer
  a, p : AdDoublet   // pointeurs sur l'élément courant, l'élément précédent
algorithme de supprimer
      a ← tete ; p ← NIL
      tantque a ≠ NIL etpuis a↑.el < e faire
          p ← a ; a ← a↑.suiv
      ftq
      // a = NIL oualors a↑.el ≥ e
      si a ≠ NIL etpuis a↑.el = e
      alors
          selon p
            p = NIL : tete ← a↑.suiv ;    // suppression en tête de la liste (a = tete)
            p ≠ NIL : p↑.suiv ← a↑.suiv ;  // suppression de l'élément a↑
          fselon
          detruire(a)
      fsi
```

Algorithmes **récursifs** de manipulation d'une séquence chaînée

- Pour écrire les versions récursives des algorithmes de manipulation de séquences chaînées, on se base sur la définition récursive des séquences chaînées :
- Base : la séquence vide : tête = NIL
- Récurrence : tête pointe sur un Doublet composé :
  - d'un élément (el) premier élément de la séquence
  - d'une Adresse (suiv) représentant la tête de la séquence qui suit le premier élément
- Les algorithmes récursifs comporteront donc toujours un paramètre de plus (modifié) : la tête de la séquence sur laquelle porte l'opération.



Algorithme récursif d'insertion d'un élément **e** dans la séquence

```
action insérer(Consulté e : Element, Modifié tet : AdDoublet)
// Effet : ajout de l'élément e à la séquence triée (en ordre croissant) de tête tet
// E.I. : indifférent
// E.F.: e est inséré dans la liste de tête t
lexique de insérer
//  e :   Element      paramètre : élément à insérer dans la liste de tête tet
//  tet : AdDoublet   paramètre : tête de la liste dans laquelle on insère e
  a : AdDoublet    // intermédiaire : adresse du Doublet créé

algorithme de insérer
   si tet ≠ NIL etpuis e > tet↑.el
   alors  // on insère e dans la liste de tête tet↑.suiv
      insérer(e, tet↑.suiv)
   sinon // on insère en tête de liste
      créer(a)
      a↑.el ← e ; a↑.suiv ← tet
      tet ← a
   fsi
```

Algorithme de récursif suppression d'un élément **e** de la séquence

```
action supprimer(Consulté e : Elément , Modifié tet : AdDoublet)
// Effet : supprime l'élément e de la séquence triée (en ordre croissant) de tête tet
// E.I. : indifférent
// E.F.: l'élément e est supprimé de la liste triée, si e n'est pas trouvé, la liste est
         inchangée
lexique de supprimer
//  e :  Element        paramètre : élément à supprimer de la liste de tête tet
//  tet : AdDoublet     paramètre : tète de la liste de laquelle on supprime e
  a : AdDoublet      // intermédiaire : adresse de l'élément à supprimer

algorithme de supprimer
      si tet ≠ NIL etpuis tet↑.el < e
      alors
         supprimer(e, tet↑.suiv)
      sinon
            // tet = NIL oualors tet↑.el ≥ e
            si tet ≠ NIL etpuis tet↑.el = e  // e trouvé => on le supprime
            alors
                  a ← tet  ; tet ← tet↑.suiv ;       // suppression en tête de la liste
                  detruire(a)
             fsi
      fsi
```

Enumération des éléments de la liste chaînée

Lexiques (rappel)

Elément : type …

AdDoublet : type pointeur de Doublet

Doublet : type agrégat el : Elément ; suiv : AdDoublet fagrégat

t : AdDoublet // tête de la liste

a : AdDoublet // adresse du Doublet contenant l'élément courant

Pour énumérer les éléments de la liste chainée, nous devons définir :

- Elément courant $a\uparrow.el$
- Démarrer $a \leftarrow t$
- Avancer $a \leftarrow a\uparrow.suiv$
- Fin de Séquence $a = NIL$

Schémas de parcours et de recherche

Lexique
Elément : type …
AdrDoublet : type pointeur de Doublet
Doublet : type agrégat el : Elément ; suiv : AdrDoublet fagrégat
t : AdrDoublet // tête de la liste
a : AdrDoublet // adresse du Doublet contenant l'élément courant

Algorithme de parcours
a ← t
Initialisation traitement
tantque a ≠ NIL faire
    traiter a↑.el
    a ← a↑.suiv
ftq
terminaison traitement

Algorithme de recherche du 1er élément vérifiant P
a ← t
tantque a ≠ NIL etpuis non P(a↑.el) faire
    a ← a↑.suiv
ftq
selon a
  a = NIL : // non trouvé
  a ≠ NIL : // a↑.el vérifie P
fselon

Nombre d'entiers positifs

Lexique partagé
Elément : type …
AdrCell : type pointeur de Cellule
Cellule : type agrégat el : Elément ; suc : AdrCell fagrégat

fonction **nbPos**(t : AdrCell) → entier ≥ 0
// nbPos(t) renvoie le nombre de valeurs positives présentes dans la liste de tête t
Lexique de nbPos
// t : AdrCell paramètre : tête de la liste
a : AdrCell // adresse de la cellule contenant l'élément courant
nbp : entier ≥ 0 // nombre d'entiers positifs de la partie parcourue
Algorithme de nbPos
a ← t
nbp ← 0
tantque a ≠ NIL faire
si a↑.el > 0 alors npb ← nbp + 1 fsi
a ← a↑.suc
ftq
renvoyer(nbp)

Création d'une liste chainée
(observation des états successifs)

Construire une liste chaînée contenant les entiers de 1 à n (liste vide si n = 0)

Lexique

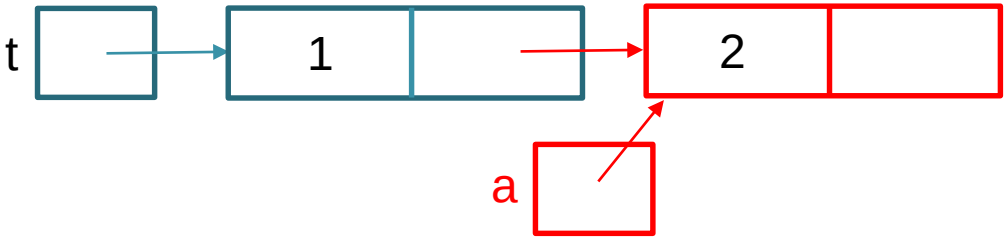
Elément : type entier

AdrCell : type pointeur de Cellule

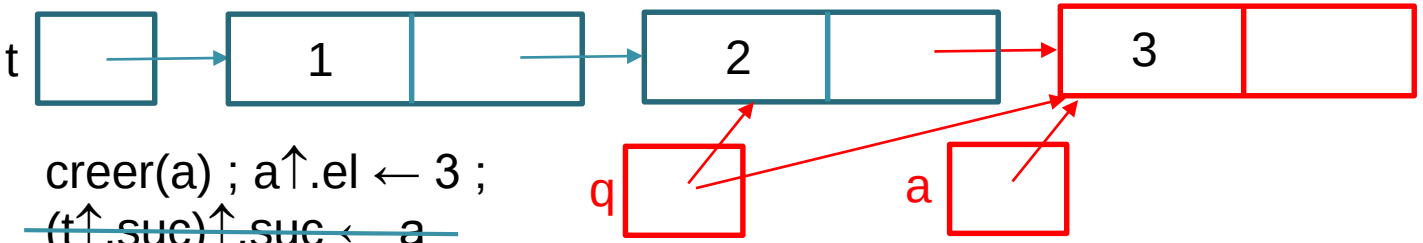
Cellule : type agrégat el : Elément ; suc : AdrCell fagrégat

t : AdrCell // tête de la liste créée

créer(t) ; t↑.el ← 1



creer(a) ; a↑.el ← 2 ;
t↑.suc ← a



creer(a) ; a↑.el ← 3 ;
~~(t↑.suc)↑.suc ← a~~
q↑.suc ← a
q ← a

Création d'une liste chainée

Construire une liste chaînée contenant les entiers de 1 à n (liste vide si n = 0)

Lexique
Elément : type entier
AdrCell : type pointeur de Cellule
Cellule : type agrégat el : Elément ; suc : AdrCell fagrégat
t : AdrCell // tête de la liste créée
a, q : AdrCell // adresses cellule créée, queue de liste
n : entier ≥ 0 // nombre de cellules à créer, supposé initialisé à n_0
i : entier > 0 // valeur à placer dans la cellule créée

Algorithme
// $n = n_0$
si n = 0 alors t ← NIL
sinon
    créer(t) ; t↑.el ← 1 ; q ← t
    i ← 2
    tantque i ≤ n faire
        créer(a) ; a↑.el ← i
        q↑.suc ← a
        q ← a
        i ← i + 1
    ftq
    q↑.suc ← NIL
fsi

Création d'une liste chainée version 1bis : (on se passe de la variable a)

Construire une liste chaînée contenant les entiers de 1 à n (liste vide si n = 0)

Lexique
Elément : type entier
AdrCell : type pointeur de Cellule
Cellule : type agrégat el : Elément ; suc : AdrCell fagrégat
t : AdrCell // tête de la liste créée
q : AdrCell // queue de la liste
n : entier ≥ 0 // nombre de cellules à créer, supposé initialisé à n_0
i : entier > 0 // valeur à placer dans la cellule créée

Algorithme
// n = n_0
si n = 0 alors t ← NIL
sinon
    créer(t) ; t↑.el ← 1 ; q ← t
    i ← 2
    tantque i ≤ n faire
        créer(q↑.suc) ;
        q ← q↑.suc
        q↑.el ← i
        i ← i + 1
    ftq
    q↑.suc ← NIL
fsi

Création d'une liste chainée par insertions en tête (version 2)

Construire une liste chaînée contenant les entiers de 1 à n (liste vide si n = 0)

Principe de la création de la liste :

- On effectue des ajouts successifs en tête de liste
- Pas de cas particulier, pas de pointeur de queue
- On commence par créer une liste vide
- On parcourt les entiers i de n à 1 et on ajoute i en tête de liste

Ajouts en queue : on préserve l'ordre d'arrivée des éléments
Ajouts en tête : on inverse l'ordre d'arrivée des éléments

Création d'une liste chainée par insertions en tête (version 2)

Construire une liste chaînée contenant les entiers de 1 à n (liste vide si n = 0)

Lexique

Elément : type entier

AdrCell : type pointeur de Cellule

Cellule : type agrégat el : Elément ; suc : AdrCell fagrégat

t : AdrCell // tête de la liste créée

n : entier ≥ 0 // nombre de cellules à créer, supposé initialisé à n_0

i : entier > 0 // valeur à placer dans la cellule créée

Algorithme

// n = n_0

t ← NIL

i ← n

tantque i > 0 faire

    créer(a) ;

    a↑.el ← i

    a↑.suc ← t

    t ← a

    i ← i - 1

ftq

Création d'une liste chainée

- Deux méthodes possibles
  - Construction par ajouts successifs en queue. L'ordre d'arrivée des éléments est conservé dans la liste créée.
    - Initialiser la liste avec le premier élément
    - Initialiser la queue
    - Chaque élément de la liste est ajouté en queue de la liste créée
  - Construction par ajouts successifs en tête de liste. L'ordre d'arrivée des éléments est inversé dans la liste créée.
    - Initialiser la liste par une liste vide
    - Chaque élément de la liste est ajouté en tête de la liste créée

Création d'une liste chainée par une action récursive (version 3)

Construire une liste chaînée contenant les entiers de 1 à n (liste vide si n = 0)

Lexique partagé
Elément : type entier
AdrCell : type pointeur de Cellule
Cellule : type agrégat el : Elément ; suc : AdrCell fagrégat

Création de la liste par insertions en queue

```
action creerListe(consultés i,n : entier ≥ 0 ; élaboré t : AdrCell)
// Effet : ajoute à la liste de tête t les entiers de i à n
Lexique
// t : AdrCell          paramètre : tête de la liste modifiée
// i, n : entier ≥ 0    paramètres : intervalle des entiers à ajouter à la liste
Algorithme
si i > n alors t ← NIL
sinon
    créer(t) ; t↑.el ← i
    creerListe(i+1, n, t↑.suc)
fsi
```

Appel : creerListe(1, n, tete)

Création d'une liste chainée par une action récursive (version 4)

Construire une liste chaînée contenant les entiers de 1 à n (liste vide si n = 0)

Lexique partagé
Elément : type entier
AdrCell : type pointeur de Cellule
Cellule : type agrégat el : Elément ; suc : AdrCell fagrégat
tete : AdrCell // tête de la liste créée

Création de la liste par insertions en tête

action creerListe(consultés n : entier ≥ 0 ; modifié t : AdrCell)
// Effet : ajoute à la liste de tête t les entiers de 1 à n
Lexique
// t : AdrCell paramètre : tête de la liste créée
// n : entier ≥ 0 borne supérieure des entiers à ajouter à la liste
Algorithme
si n > 0
alors
    créer(a) ;
    a↑.el ← n ; a↑.suc ← t ; t ← a
    creerListe(n-1, t)
fsi

Appel : tete ← NIL ; creerListe(n, tete)

Destruction d’une liste

Exercice : écrire un algorithme qui détruit totalement une liste donnée

- l’ordre des éléments n’a pas d’influence sur l’algorithme
- on peut donc procéder par une succession des suppressions en tête

Destruction d'une liste

```
action détruireListe(modifié t :AdrCell)
// Effet supprime la liste de tête t
Lexique
// t :AdrCell          paramètre : tête de la liste supprimée
// a :AdrCell          adresse du doublet à supprimer
Algorithme itératif
tantque t != NIL faire
            a ← t
            t ← t^.suc
            détruire(a)
ftq

Algorithme récursif
si t ≠ NIL
alors
            detruireListe(t^.suc)  // destruction des la fin de la liste
            detruire(t)            // destruction de la tête
            t ← NIL
fsi
```

Exercice

- Appliquer cette représentation à l'exemple de la course contre la montre

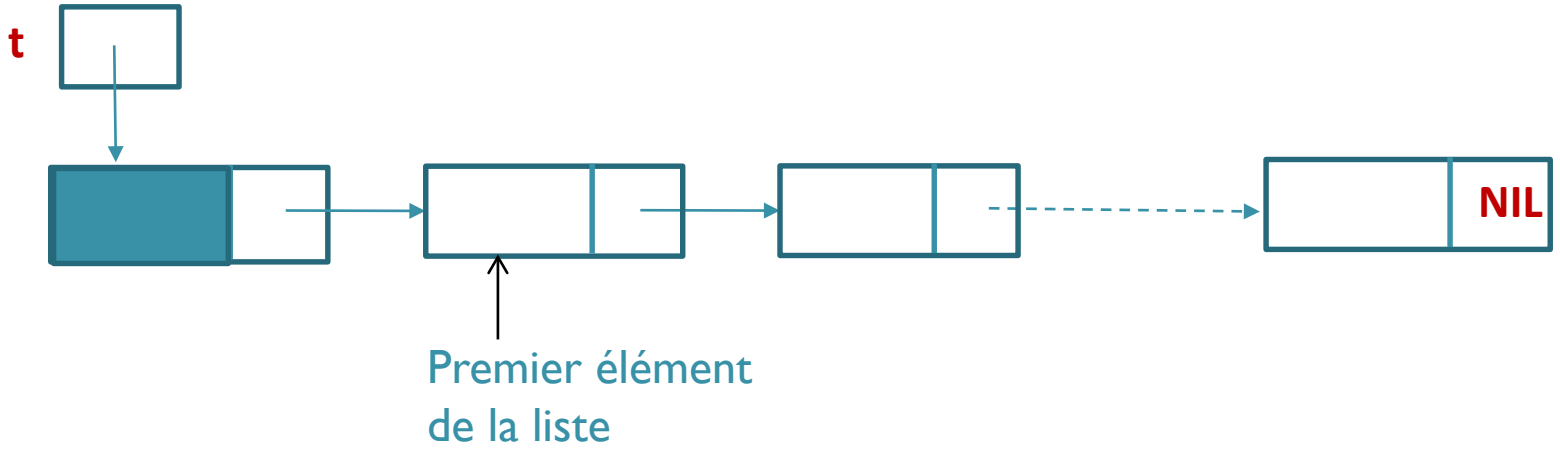
Variantes de représentation des séquences chaînées

- On enrichit la représentation de la liste chaînée pour simplifier/faciliter l'écriture d'un algorithme, parfois de manière temporaire :
  - Pointeurs de queue
  - Elément fictif placé en tête
  - Listes circulaires
  - Listes doublement chaînées

Listes avec pointeurs de queue

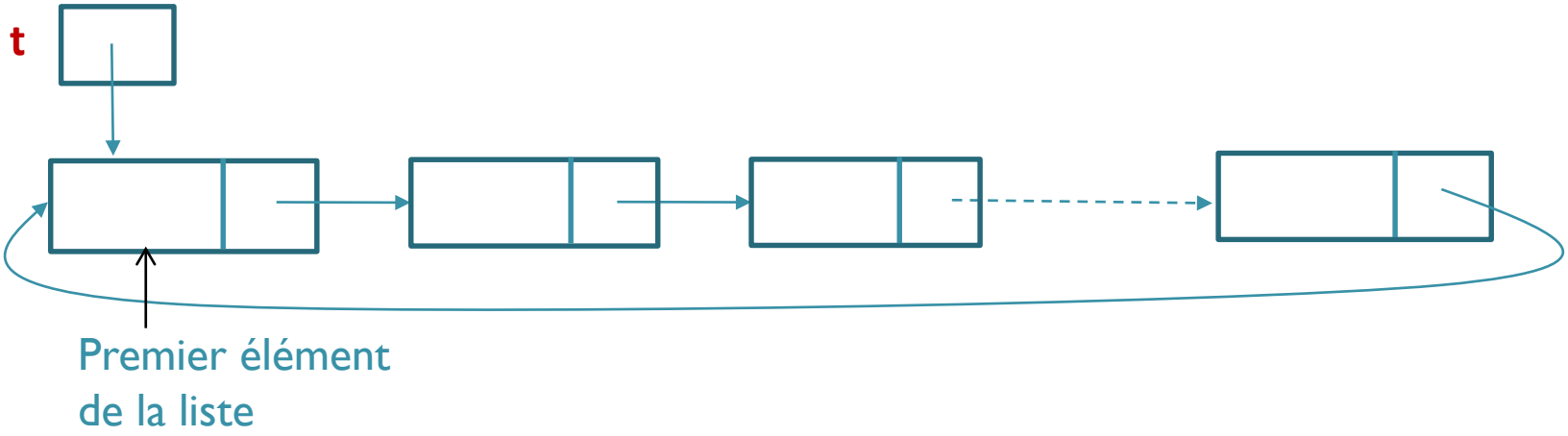
- Simplifie les ajouts en queue
- Utile pour la création par recopie d'une séquence
- Cas particuliers à traiter
  - Suppression en queue nécessite la mise à jour de la queue
  - Ajout dans une liste vide nécessite l'initialisation de la queue (différent de l'insertion en tête)

Elément fictif placé en tête



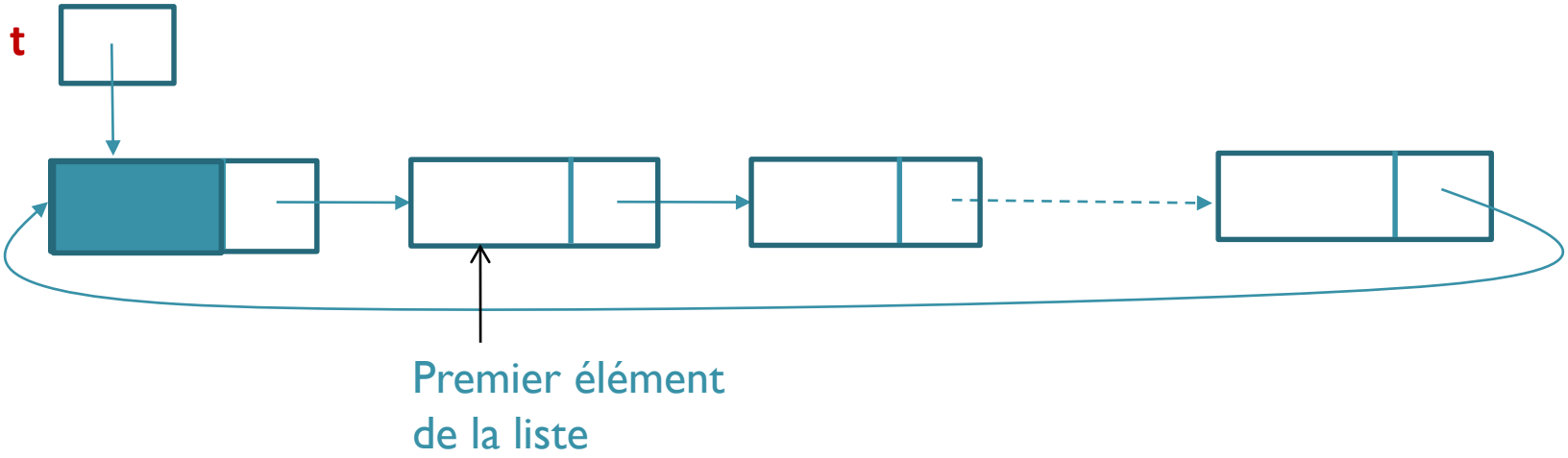
- On a toujours une liste non vide
- On accède à la liste par l'élément fictif, la tête t n'est jamais modifiée
- Permet d'unifier les opérations de modification de la liste
  - Ajout en tête = ajout après l'élément fictif
  - Suppression en tête = suppression de l'élément après l'élément fictif
- L'introduction d'un élément fictif peut être provisoire, pour les besoins d'un algorithme particulier
- Pas utile si l'on ne fait que des ajouts/suppressions en tête

Liste circulaire



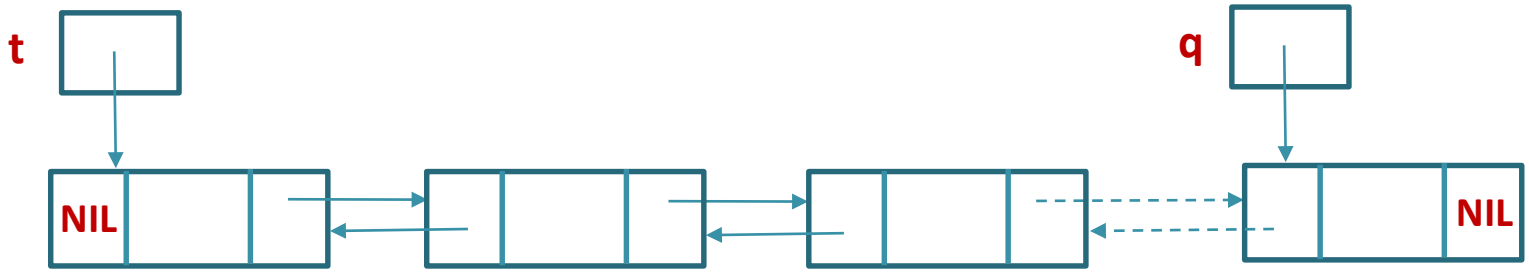
- Le dernier élément pointe sur le premier
- On peut avoir une liste vide
- Parfois on a un pointeur de queue plutôt que d'avoir un pointeur de tête
  - faciliter l'insertion en queue (file d'attente)
  - l'élément en tête et les suivant du dernier

Liste circulaire avec élément fictif



- Les dernier élément de la liste pointe sur l'élément fictif
- Liste circulaire jamais vide
- On accède à la liste par l'élément fictif, la tête t n'est jamais modifiée
- L'élément fictif sert à repérer la fin, on peut y placer une valeur sentinelle

Liste doublement chainée



- On utilise des cellules (triplets) comportant 2 pointeurs :

  AdrCell : pointeur de Cellule

  Cellule : type agrégat

  el : Element

  pred, suc : AdrCell

  fagrégat

- On peut bien entendu :
  - Introduire un élément fictif
  - Gérer des listes circulaires doublement chaînées

Listes chainées représentées dans un tableau

Listes chainées représentées dans un tableau

- Si la mémoire contenant la séquence est un tableau, un chainage (pointeur) est alors représenté par un indice de ce tableau.
- Le tableau est un ensemble de Doublets composés d'un Elément et d'une adresse :
- Doublet : type agrégat
  el : Elément
  suiv : Adresse // indice de tableau
  fagrégat

Tabulation de la fonction de succession

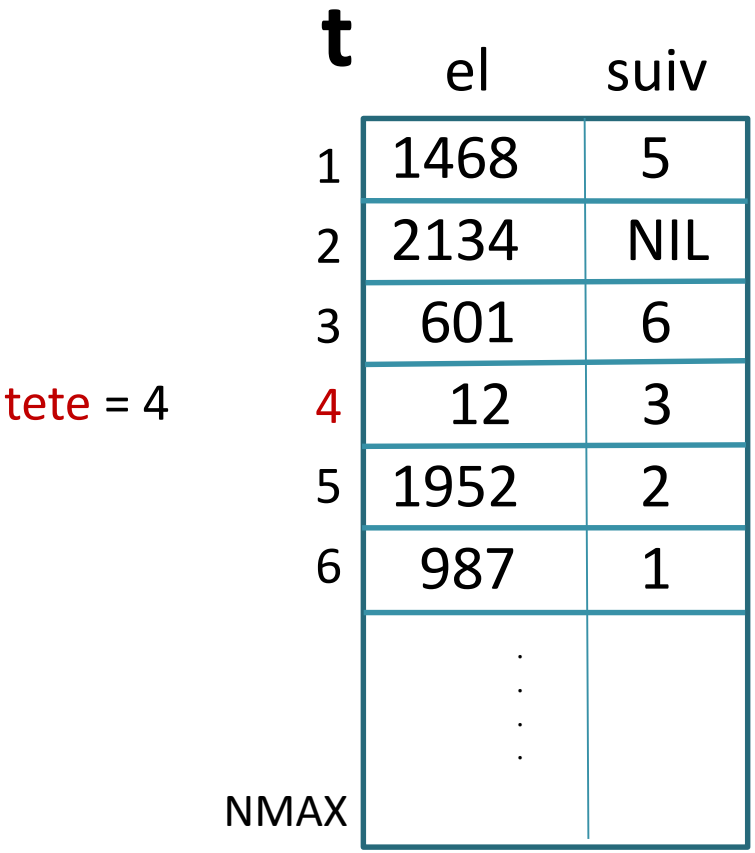
- Un Elément e de la séquence est rangé dans un doublet d'indice a du tableau **t**.
- On a alors :
  - t[a].el = e
  - t[a].suiv = b, où b est défini comme suit :
    - Si e n'est pas le dernier élément de la séquence, b est l'indice dans t du Doublet contenant le successeur de e dans la séquence.
    - Si e est le dernier élément, b a une valeur particulière nommée NIL, définie pour noter la fin de la séquence.
    - t[NIL] n'est pas défini
    - Une valeur de NIL souvent utilisée est -1.
- Pour pouvoir accéder au premier élément de la séquence, on définit dans le lexique tete une variable contenant l'indice dans t du premier élément de la séquence.
- Une séquence vide est caractérisée par tete = NIL.

tete : entier sur binf..bsup ∪ NIL

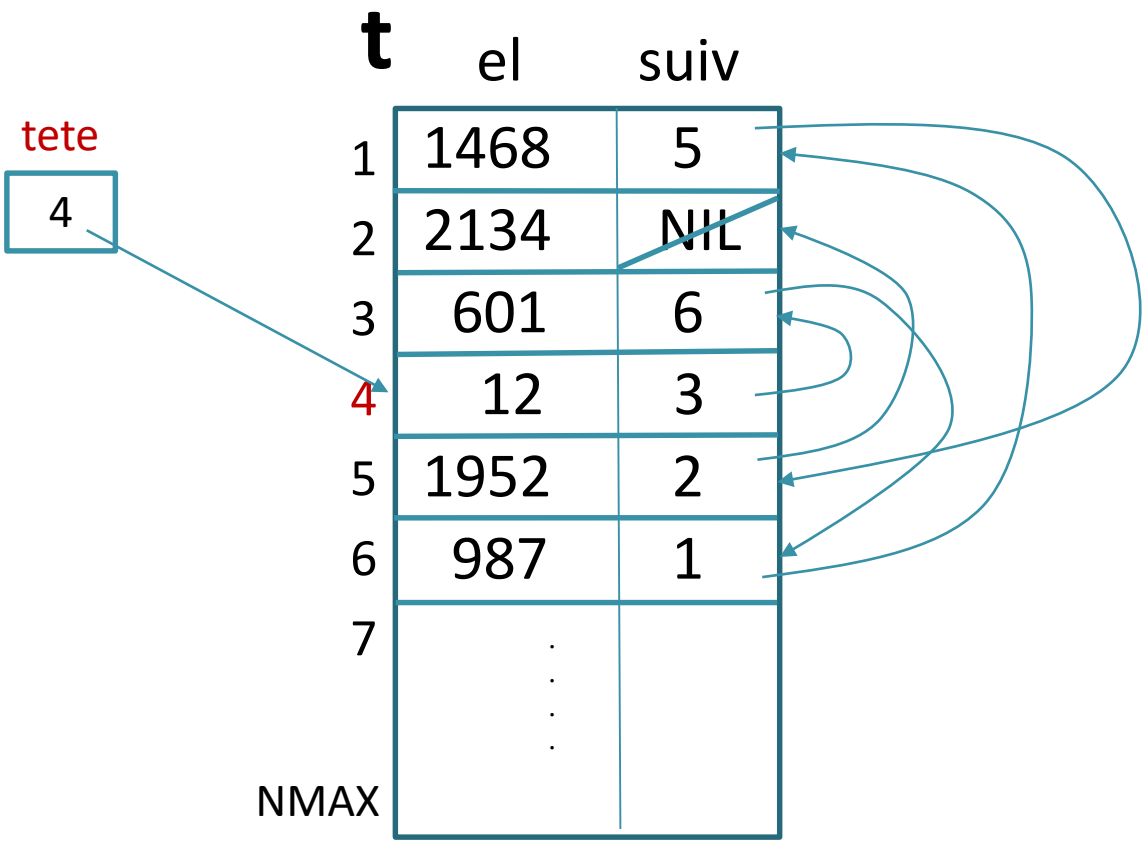
ou

tete : entier sur NIL..bsup // si binf = 0 et NIL = -1

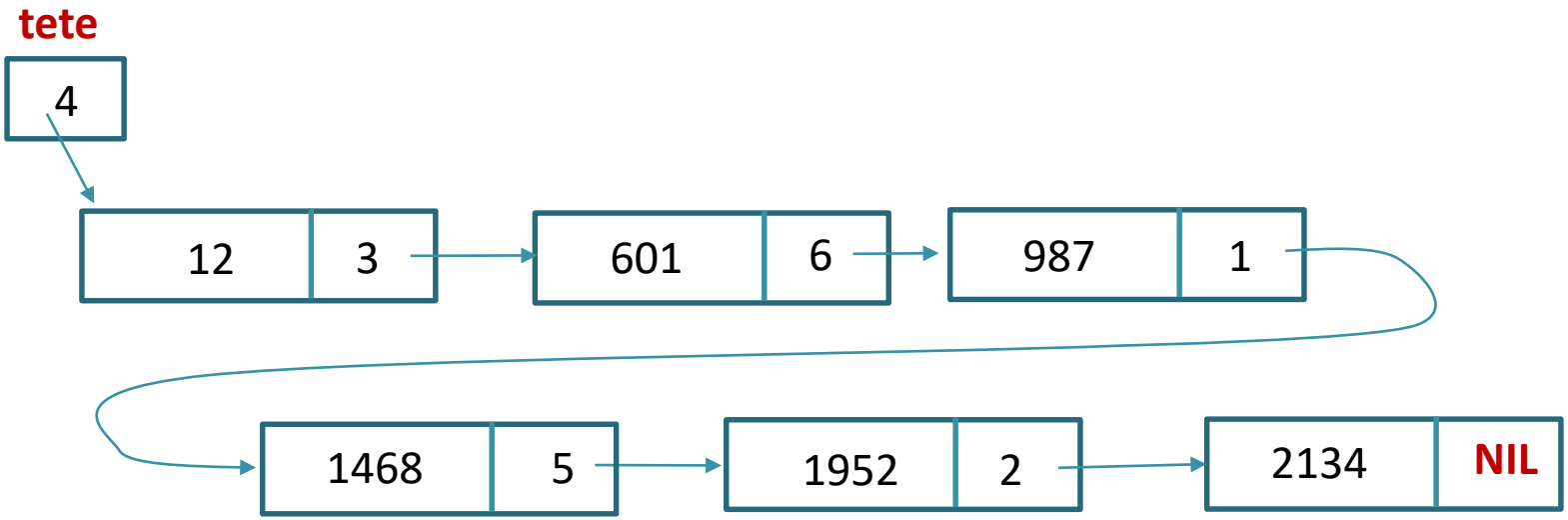
Schématisation d'une séquence chaînée : exemple séquence d'entiers triée



Schématisation d'une séquence chaînée : exemple séquence d'entiers triée



Schématisation de la séquence chaînée



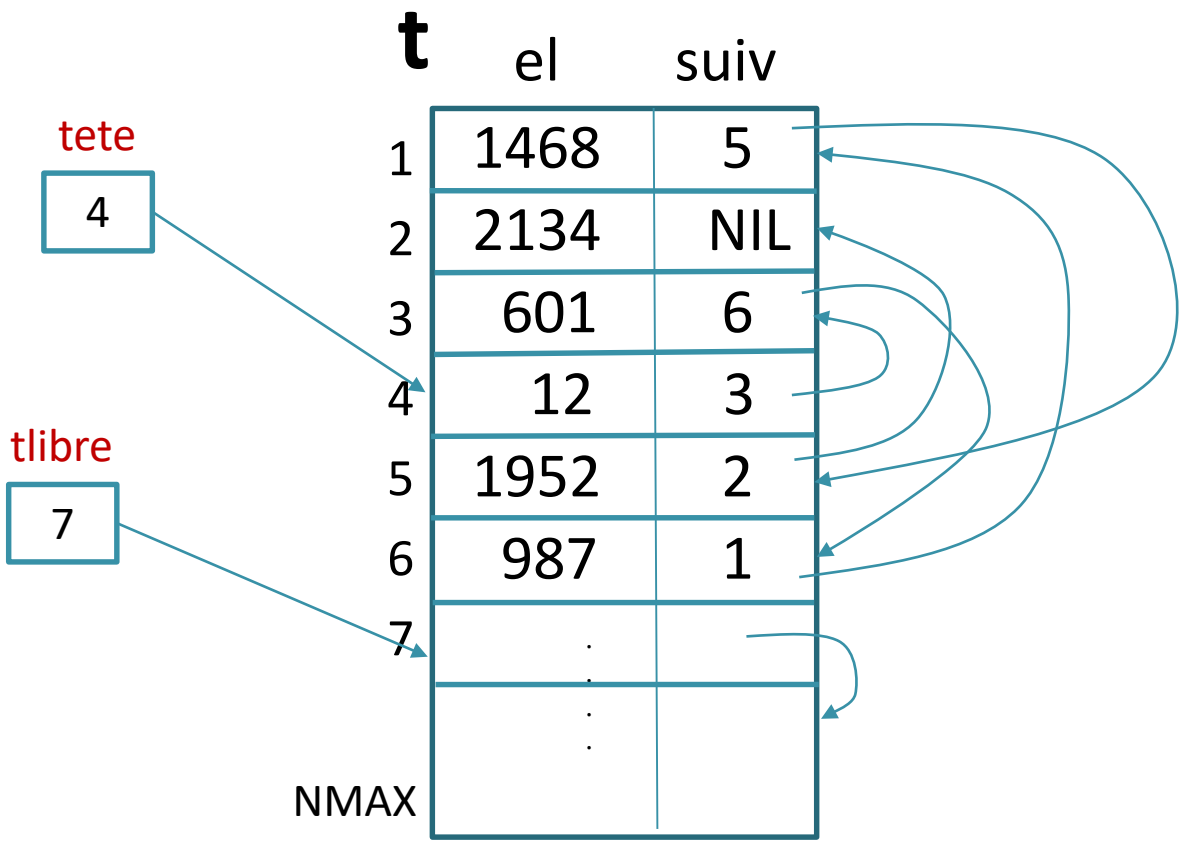
Gestion de l'espace mémoire (1)

- Le tableau comporte des doublets occupés par les éléments de la séquence, et des doublets libres.
- Les doublets libres sont dispersés dans le tableau: il faut pouvoir les atteindre en cas de besoin pour l'ajout d'un nouvel élément.
- Pour cela on forme une séquence chainée des doublets libres : la séquence libre.
- tlibre est la tête de la séquence libre.
- Le tableau est saturé lorsque la séquence libre est vide (tlibre = NIL)
- Pour ajouter un nouvel Elément e à la séquence, il faut le ranger dans un doublet libre.
- On doit donc en déterminer l'adresse : on parle de l'allocation d'un doublet à l'élément e.

Gestion de l'espace mémoire (2)

- Inversement, après avoir enlevé un élément de la séquence, le doublet d'indice a correspondant est restitué à la séquence libre : on parle de libération du doublet a.
- Lors de l'allocation d'un doublet, on choisit toujours le premier disponible, c'est-à-dire le doublet d'adresse tlibre.
- Celui-ci devient occupé, il faut par conséquent l'enlever de la séquence libre.
- L'initialisation du tableau consiste à mettre tous les doublets dans la séquence libre.

Schématisation d'une séquence chaînée : exemple séquence d'entiers triée



Représentation chainée d'une séquence dans un tableau

Lexique (partagé ou de la classe concernée)

NMAX : entier > 0 // nb d'éléments de t

NIL : l'entier -1

Adresse : type entier entre -1 et NMAX-1

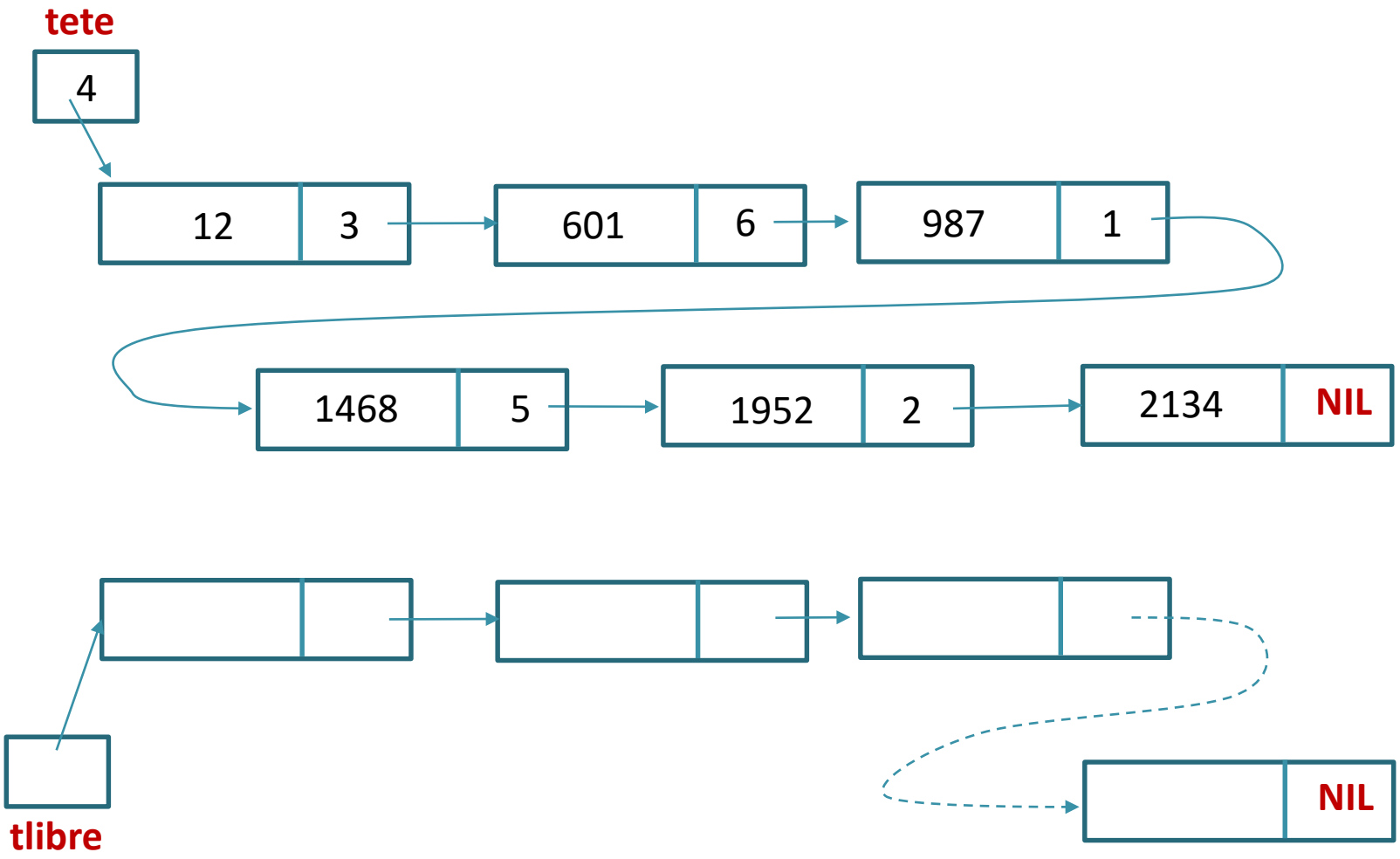
Doublet : type agrégat el : Elément ; suiv : Adresse fagrégat

t : tableau sur [0..NMAX-1] de Doublets

tete : Adresse // indice du premier élément de la séquence représentée

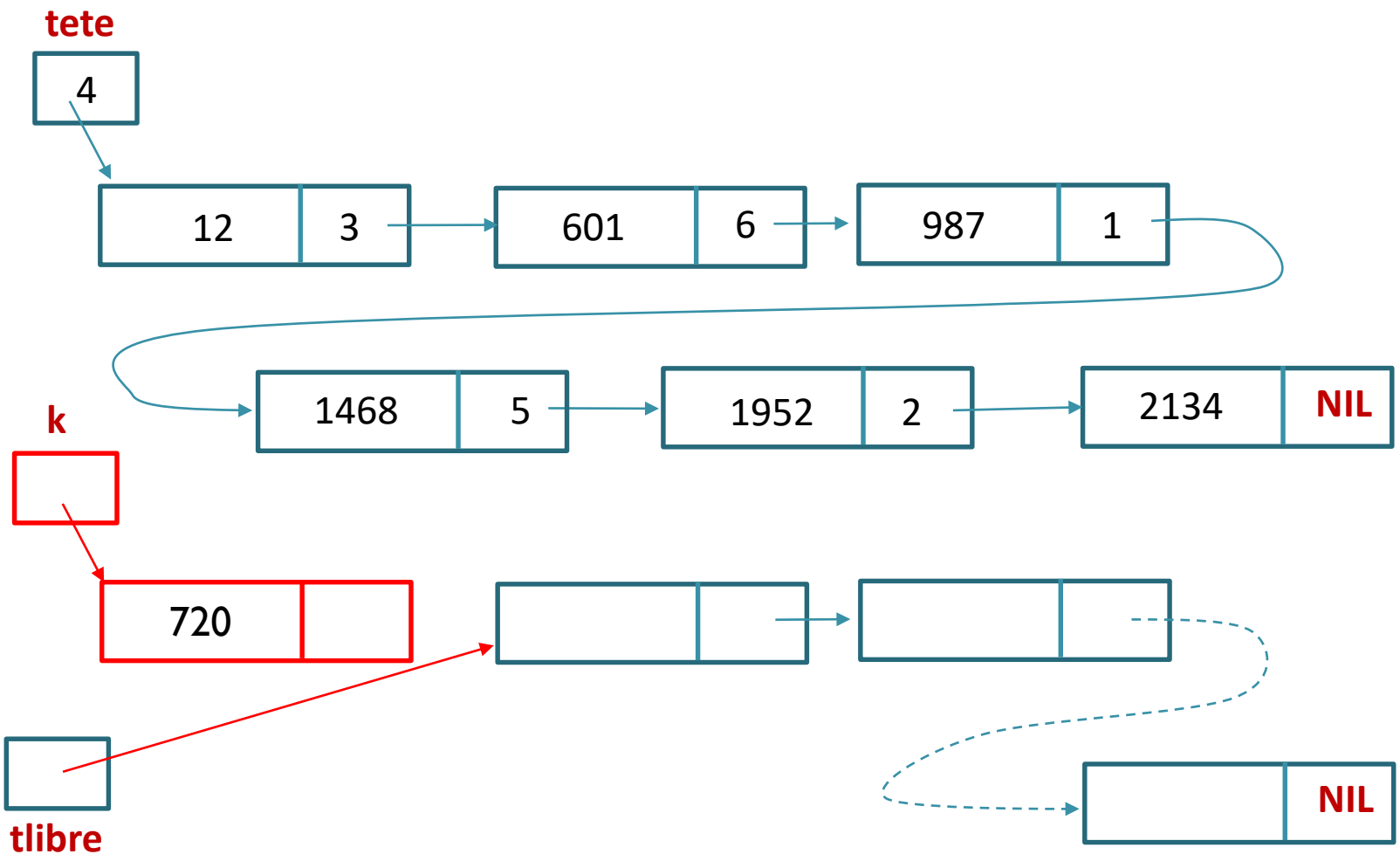
tlibre : Adresse // indice du premier emplacement libre dans t

Ajout de l'élément 720 à la séquence



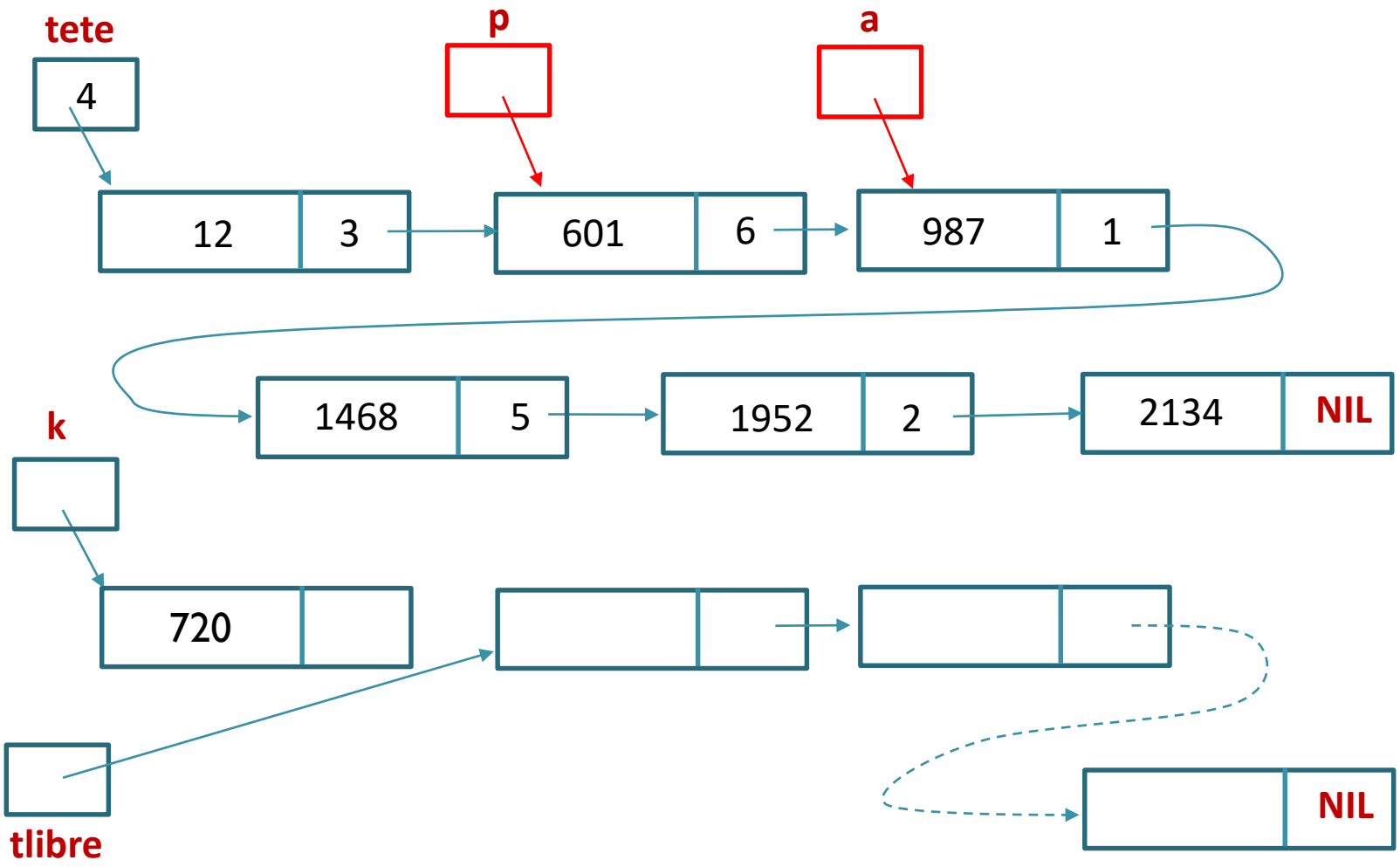
Ajout de l'élément 720 à la séquence

1 - Allouer un doublet libre d'indice **k** et y ranger 720



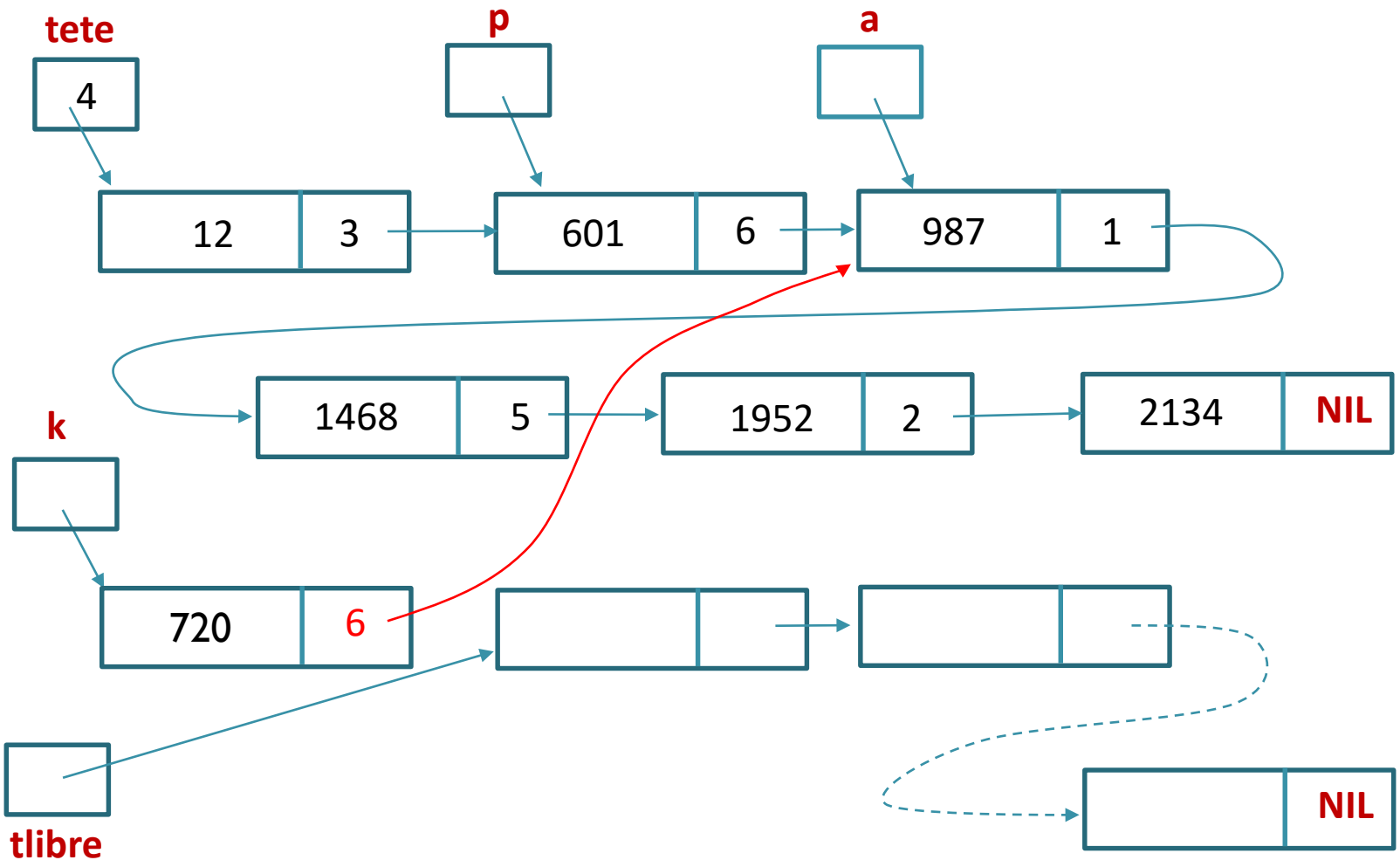
Ajout de l'élément 720 à la séquence

2 - déterminer **p** l'indice du doublet qui précède l'élément inséré, et **a** l'indice du doublet qui le suit



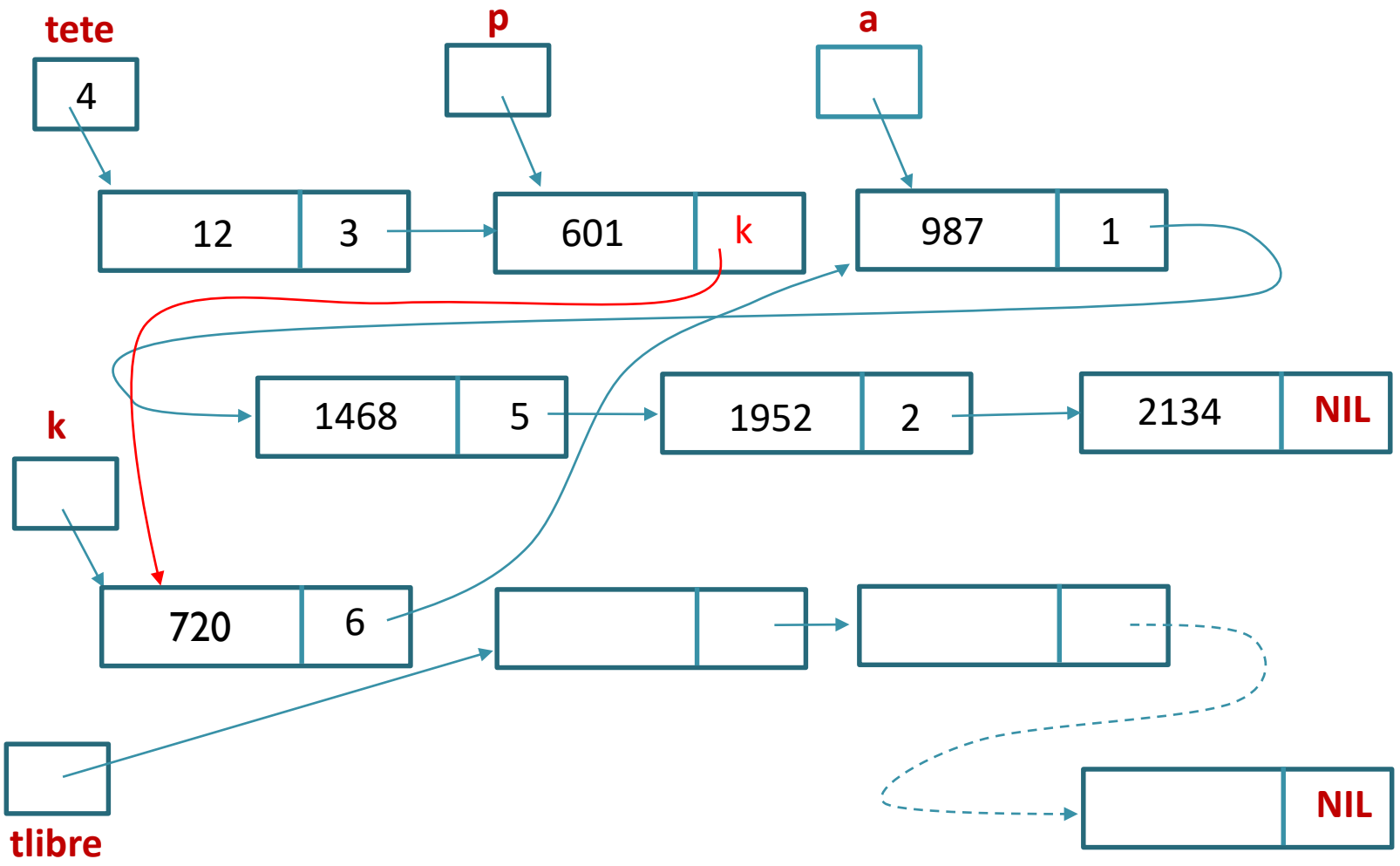
Ajout de l'élément 720 à la séquence

3 - déterminer la valeur du successeur de k : l'ancien successeur de p



Ajout de l'élément 720 à la séquence

4 – déterminer la valeur du successeur de p : k



Opérations de gestion de la liste libre

```
action allouer(Elaboré x : Adresse)
// Effet : alloue un doublet libre, soit x son adresse, et le supprime de la liste libre
// E.I. : indifférent
// E.F.: x = adresse du doublet alloué, s'il n'y avait pas d'emplacement libre, x = NIL
lexique de allouer
//  x : Adresse        paramètre : adresse du doublet alloué
algorithme de allouer
   x ← tlibre ;
   si tlibre ≠ NIL
   alors
      tlibre ← t[tlibre].suiv
   fsi

action libérer(Consulté x : Adresse)
// Effet : libère le doublet d'adresse x : l'ajoute en tête de la liste libre
// E.I. : x = adresse du doublet à libérer
// E.F.: le doublet t[x] a été placé en tête de la liste libre
lexique de libérer
//  x : Adresse      paramètre : adresse du doublet à libérer
algorithme de libérer
   t[x].suiv ← tlibre
   tlibre ← x
```

Initialisation du tableau

Au départ, tous les emplacements du tableau t sont des Doublets libres

```
action initialiser
// Effet : initialise t en une liste d'emplacements libres
// E.I. : indifférent
// E.F.: t représente la séquence libre constituée des nmax emplacements de t
lexique de initialiser
  i : Adresse      // indice de parcours de t
algorithme de initialiser
i ← 0
tlibre ← 0
tantque  i  < NMAX-1 faire
   t[i].suiv ← i+1
   i ← i+1
ftq
t[NMAX-1].suiv ← NIL
```