Algorithmique E3

Structures de données

Daniel Vaz

daniel.ramosvaz@esiee.fr

2025-12-15

ESIEE PARIS

Notation

Opérations :

et ($\wedge$)	$x > 1 \wedge x < 3$	$\leftrightarrow$	$x > 1$ et $x < 3$	(C : `&&`)
ou ($\vee$)	$x < 0 \vee x > 2$	$\leftrightarrow$	$x < 0$ ou $x > 2$	(C : `||`)
non ($\neg$)	$\neg(x < 0)$	$\leftrightarrow$	non$(x < 0)$	(C : `!(x < 0)`)
égal ($=$)	$x = 0$	$\leftrightarrow$	x est égal à 0 ?	(C : `x == 0`)
affecter ($\leftarrow$)	$x \leftarrow 0$	$\leftrightarrow$	x prend la valeur 0	(C : `x = 0`)

Spécification d'intervalles :

- for $i \leq 0 < n$ do

for i de x à y

- `tableau`$[a : b]$: éléments de indice a à $b - 1$

3 premiers + [0 : 3]

Révision

La structure d'un tableau

Tableau : une suite contiguë d'éléments de même type

Les éléments sont indexés par la position (entier)

- Élément k du tableau t se notera $t[k]$
- La numérotation commence à partir de zéro
- Accès à un élément en temps constant

La limite du tableau est spécifiée :

- soit en précisant la longueur,
- soit en précisant une sentinelle ($'\backslash 0'$ pour les chaînes en C)

Les tableaux dans la mémoire

Tableaux peuvent être alloués dans la mémoire pile ou tas

Mémoire pile : Stocke information sur l'exécution des fonctions

- Exemple : `int tbl[10];`
- Mémoire alloué pour une fonction et libéré à la sortie
- Facile à utiliser, normalement limité à 1–8 MB

Mémoire tas : Mémoire allouée manuellement

- Exemple : `int* tbl = (int*) malloc(10*sizeof(int));`
- Permet d'utiliser grandes quantités de mémoire
- Reste accessible dans toutes les fonctions
- Nécessite une libération manuelle
  - si on perd le pointeur $\rightarrow$ *memory leak*

Itération sur les tableaux – Avantages

Tableau avec longueur :

- \+ Longueur en temps $\Theta(1)$
- \+ Itération en sens inverse
- – Toujours 2 paramètres

Tableau avec sentinelle (chaines)

- \+ Un seule paramètre
- \+ Facile à modifier
- – Longueur en temps $\Theta(n)$

void afficher(int* tbl, int n)

Itération en précisant la longueur

Algorithme EstTrie

Données: tableau t, longueur n de t

Résultat: **true** si t est trié, sinon **false**

```
1: for 0 ≤ i < n do
2:     if t[i − 1] > t[i] then
3:         return false
4:     end if
5: end for
6: return true
```

Comment passer un seul paramètre ?

Redimensionner un tableau

Et si nous voulons des opérations plus complexes ?

Exercice : Donner une fonction `redim(int* tbl, int n, int m)` qui redimensionne un tableau de taille n en taille m. ($m > n$?)

allouer un tableau
copier les éléments
libérer

Redimensionner un tableau

Et si nous voulons des opérations plus complexes ?

Exercice : Donner une fonction `redim(int* tbl, int n, int m)` qui redimensionne un tableau de taille n en taille m.

```
1 int* redim(int* tbl, int n, int m) {
2     int* nouveau_tbl = (int*) malloc(m*sizeof(int));
3     if (!nouveau_tbl) return NULL;
4     for (int i=0; i<n; i++)
5         nouveau_tbl[i] = tbl[i];
6     free(tbl);
7     return nouveau_tbl;
8 }
```

Redimensionner un tableau

Et si nous voulons des opérations plus complexes ?

Exercice : Donner une fonction `redim(int* tbl, int n, int m)` qui redimensionne un tableau de taille n en taille m.

```
1 int* redim(int* tbl, int n, int m) {
2     int* nouveau_tbl = (int*) malloc(m*sizeof(int));
3     if (!nouveau_tbl) return NULL;
4     for (int i=0; i<n; i++)
5         nouveau_tbl[i] = tbl[i];
6     free(tbl);
7     return nouveau_tbl;
8 }
```

Complexité ? $\Theta(n)$, dépend de `malloc` (mais rapide en général)

Structures de données

Qu'est ce qu'une structure de données ?

Façon de stocker et organiser des données, comportant :

- les données et les relation entre les données,
- des fonctions et opérateurs qui on applique aux données.

Exemple : tableau dynamique Python, list , Java : ArrayList

- Tableau qui change sa taille quand nécessaire
- Permet d'insérer et supprimer des éléments
- données : un tableau t et sa longueur n ;

Qu'est ce qu'une structure de données ?

Façon de stocker et organiser des données, comportant :

- les données et les relation entre les données,
- des fonctions et opérateurs qui on applique aux données.

Exemple : tableau dynamique

- Tableau qui change sa taille quand nécessaire
- Permet d'insérer et supprimer des éléments
- données : un tableau t et sa longueur n ;
- stocke aussi m, le nombre d'éléments alloués ; (capacité)

Qu'est ce qu'une structure de données ?

Façon de stocker et organiser des données, comportant :

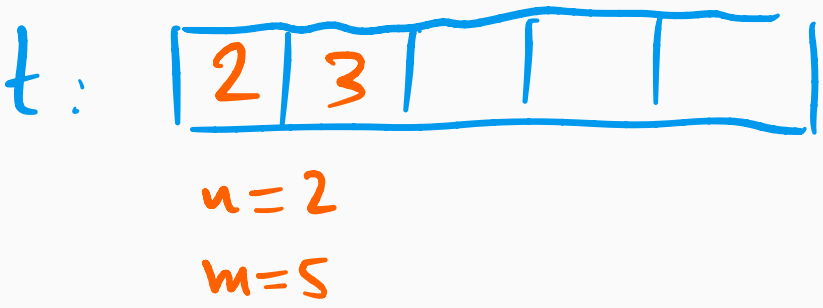
- les données et les relation entre les données,
- des fonctions et opérateurs qui on applique aux données.

Exemple : tableau dynamique

- Tableau qui change sa taille quand nécessaire
- Permet d'insérer et supprimer des éléments
- données : un tableau t et sa longueur n ;
- stocke aussi m, le nombre d'éléments alloués ;
- opérations : insertion, redimensionnement, ...

longueur

Comment organiser une structure de données ?



On utilise une structure C pour grouper les données.

```
1 typedef struct {
2     int* t;
3     int n;
4     int m;
5 } tbl_int;
```

Opérations : Créer un tableau

```
1 tbl_int tbl_int_nouveau(int m) {
2     tbl_int tbl = {NULL, 0, 0};
3     if (m>0)
4         tbl_int_redim(&tbl, m);
5     return tbl;
6 }
```

t n m

Opérations : Redimensionner

```
1  void tbl_int_redim(tbl_int * tbl, int m) {
2      int* nouveau_t = (int*) malloc(m*sizeof(int));
3      if (!nouveau_t[illegible]) return;
4      if (tbl->t != NULL) {
5          for (int i=0; i<tbl->n; i++)
6              nouveau_t[i] = tbl->t[i];
7          free(tbl->t);
8      }
9      tbl->t = nouveau_t;
10     tbl->m = m;
11 }
```

! nouveau_t : nouveau_t ≠ NULL

tbl->t : attribut t de tbl (mémoire alloué pour stocker les éléments)

tbl.m : tbl_int tbl

tbl->m : tbl_int* tbl

(*tbl).m

Opérations : Insérer un élément

Comment insérer un élément ? Pas comme ça !

```
1 void tbl_int_inserer(tbl_int * tbl, int el, int pos) {
2     for (int i=pos; i<tbl->n; i++) {
3         int z=tbl->t[i];
4         tbl->t[i] = el;
5         el = z;
6     }
7     tbl->t[tbl->n] = el;
8     (tbl->n)++;
9 }
```

Opérations : Insérer un élément

Comment insérer un élément ?

```
1  void tbl_int_inserer(tbl_int * tbl, int el, int pos) {
2      if (tbl->n == tbl->m)
3          tbl_int_redim(tbl, 1+2*tbl->m);
4      for (int i=pos; i<tbl->n; i++) {
5          int z=tbl->t[i];
6          tbl->t[i] = el;
7          el = z;
8      }
9      tbl->t[tbl->n] = el;
10     (tbl->n)++;
11 }
```

Opérations : Insérer un élément

Comment insérer un élément ?

```
1 void tbl_int_inserer(tbl_int * tbl, int el, int pos) {
2     if (tbl->n == tbl->m)
3         tbl_int_redim(tbl, 1+2*tbl->m);
4     for (int i=tbl->n; i>pos; i--)
5         tbl->t[i] = tbl->t[i-1];
6     tbl->t[pos] = el;
7     (tbl->n)++;
8 }
```

Fichier d'en-tête

tbl_int.h

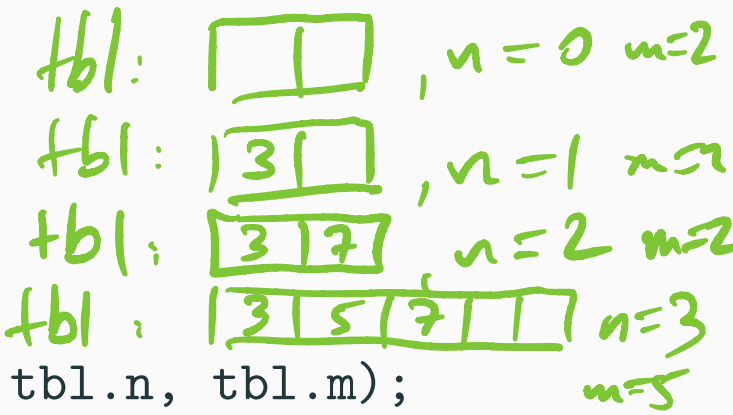
```
1  // tbl_int.h
2  typedef struct {
3      int* t;
4      int n;
5      int m;
6  } tbl_int;
7
8  tbl_int tbl_int_nouveau(int m);
9  void tbl_int_redim(tbl_int * tbl, int m);
10 void tbl_int_inserer(tbl_int * tbl, int el, int pos);
```

Utilisation

```c
#include "tbl_int.h"

int main() {
    tbl_int tbl = tbl_int_nouveau(2);
    inserer(&tbl, 3, 0);
    inserer(&tbl, 7, 1);
    inserer(&tbl, 5, 1);
    printf("tbl.n: %d, tbl.m: %d\n", tbl.n, tbl.m);
    printf("%d %d %d %d\n", tbl.t[0], tbl.t[1],
        tbl.t[2], tbl.t[3]);
    return 0;
}
```

tbl_int



3 5 7 ?

Opérations

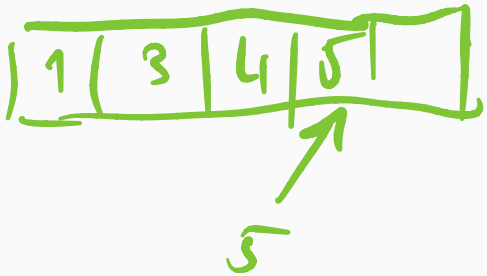
Autres opérations possibles

- `void tbl_int_trier(tbl_int* tbl)`
- `int tbl_int_rechercher(tbl_int* tbl, int x)`
- `int tbl_int_inverser(tbl_int* tbl)`

Complexité

Complexité des opérations :

- Créer : (vide) $\Theta(1)$
- Accès à l'élément à la position i : $\Theta(1)$
- Redimensionnement : $\Theta(n)$
- Insertion ou suppression : $\Theta(n)$
- Insertion ou suppression du dernier élément : $\Theta(1)$
  - ~~(en moyenne)~~ (si $m > n$)



Complexité

pire cas

Complexité des opérations :

- Créer (vide $n=0$) : $\Theta(1)$
- Accès à l'élément à la position i : $\Theta(1)$
- Redimensionnement : $\Theta(n)$
- Insertion ou suppression : $\Theta(n)$
- Insertion ou suppression du dernier élément : $\Theta(1)$
  - ~~(en moyenne)~~ ($n > n$)

Structures de données

- Façon de stocker et organiser des données,
- Permet de distribuer facilement des fonctionnalités,
- Généralement, un fichier `.h` avec les structures et en-têtes,
- Le code est dans un fichier `.c` séparé.

Les listes chaînées

Les listes chaînées

Structure de données

Les listes chaînées

Une liste chaînée est une structure de données :

- Pour stocker une séquence d'éléments (d'un même type)
- Organisé de manière non-contiguë dans la mémoire
- Permet d'insérer et supprimer des éléments efficacement
- Peut servir de base à d'autres structures
  - piles, files, ...
- Opérations principales : insertion, suppression, recherche

Structure d'une liste chaînée

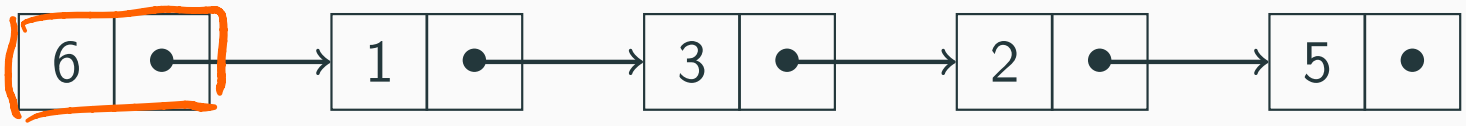
Structure :

- un ensemble d'éléments (les maillons)
- chaque élément contient :
  - un valeur (contenu) et
  - un pointeur (chaînage) vers l'élément suivant

Structure d'une liste chaînée

Structure :

- un ensemble d'éléments (les maillons)
- chaque élément contient :
  - un valeur (contenu) et
  - un pointeur (chaînage) vers l'élément suivant
- Petites détails :
  - `NULL` : liste vide / sentinelle
  - `CONTENU`(m) : contenu, `SUIVANT`(m) : élément suivant de m.



séq : 6, 1, 3, 2, 5

Structure d'une liste chaînée

```c
// liste_chainee.h
typedef struct _maillon_int {
    int val;
    struct _maillon_int* suiv;
} maillon_int, *liste_int;
```

Structure d'une liste chaînée

```c
// liste_chainee.h
typedef struct _maillon_int {
    int val;
    struct _maillon_int* suiv;
} maillon_int, *liste_int;
```

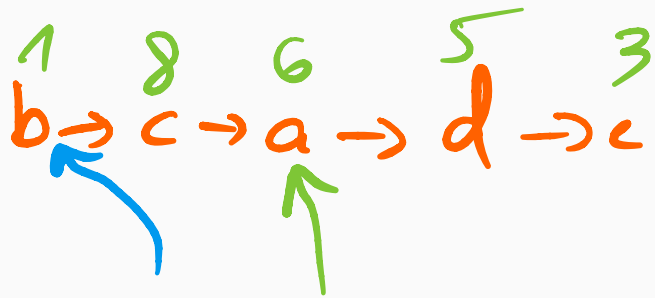
↳ liste_int pointeur vers maillon_int.

struct _maillon_int m;

maillon_int m

Exemple (trop) basique

```
1  int main() {
2      maillon_int ma={6, NULL}, mb={1, NULL};
3      maillon_int mc={8, NULL}, md={5, NULL};
4      maillon_int me={3, NULL};
5
6      ma.suiv = &md; mb.suiv = &mc;
7      mc.suiv = &ma; md.suiv = &me;
8
9      liste_int lst = &ma;
10     liste_int lst2 = &mb;
11 }
```



Les listes chaînées

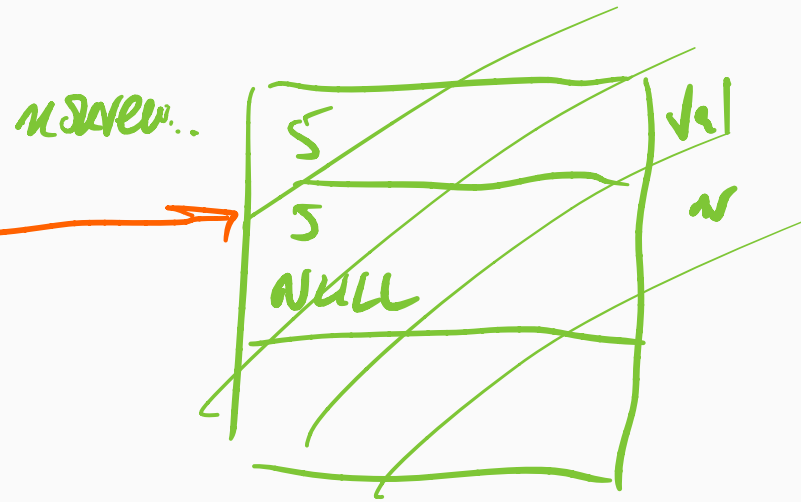
Opérations basiques

Opérations basiques

- Créer un maillon
- Insérer un nouveau maillon après un maillon m
- Insérer un nouveau maillon dans la position i
- Supprimer le maillon après un maillon m

Créer un maillon

```
1 maillon_int* nouveau_maillon_int(int val) {
2     maillon_int nv;
3     nv.val = val;
4     nv.suiv = NULL;
5     return &nv;
6 }
```



Créer un maillon

```
1 maillon_int* nouveau_maillon_int(int val) {
2     maillon_int nv;
3     nv.val = val;
4     nv.suiv = NULL;
5     return &nv;
6 }
```

Segmentation fault !

Mieux si on retourne `nv` ?

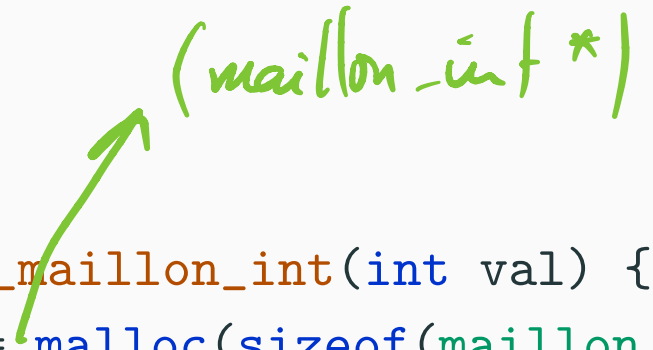
Créer un maillon

```
1 maillon_int* nouveau_maillon_int(int val) {
2     maillon_int nv;
3     nv.val = val;
4     nv.suiv = NULL;
5     return &nv;
6 }
```

Segmentation fault !

Mieux si on retourne `nv` ? Non !

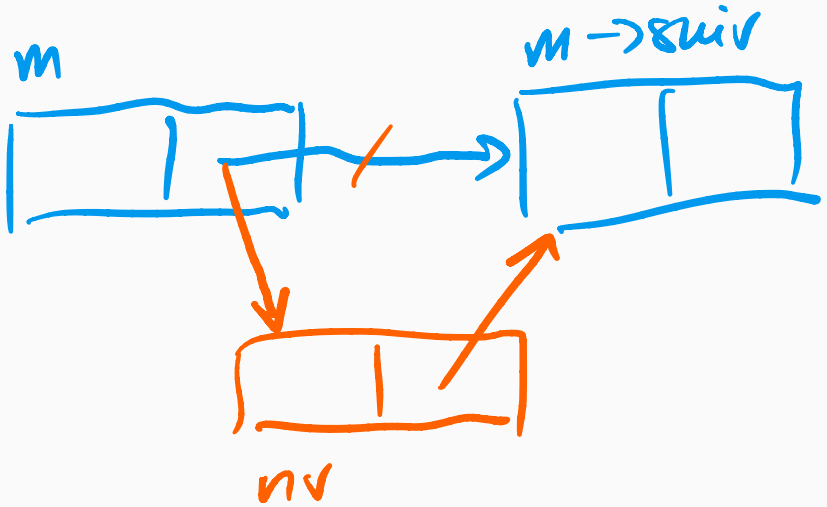
Créer un maillon




```
1 maillon_int* nouveau_maillon_int(int val) {
2     maillon_int* nv = malloc(sizeof(maillon_int));
3     if (nv == NULL) return NULL;
4     nv->val = val;
5     nv->suiv = NULL;
6     return nv;
7 }
```

Insérer après un maillon

```
1 maillon_int* liste_int_inserer_apres(maillon_int* m,
      int val) {
2     maillon_int* nv = nouveau_maillon_int(val);
3     if (nv == NULL) return NULL;
4     nv->suiv = m->suiv;
5     m->suiv = nv;
6     return nv;
7 }
```



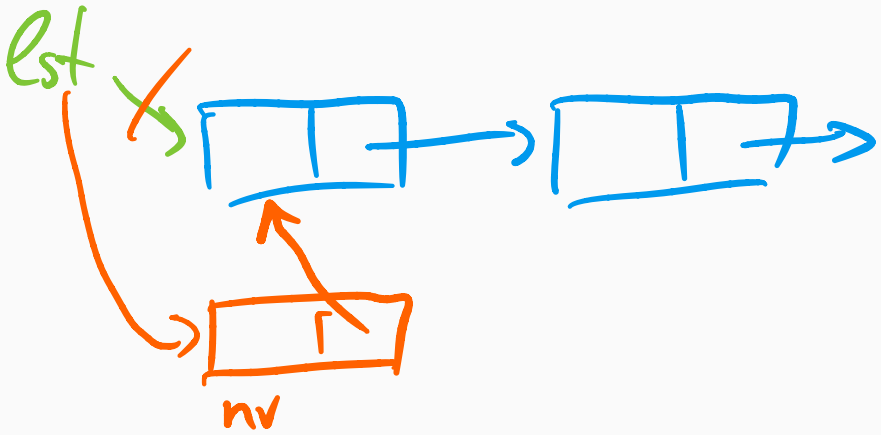
Insérer dans une position

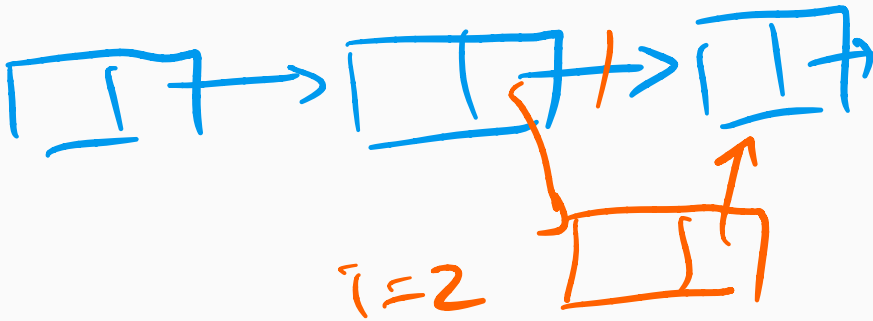
Algorithme InsererPosition

Données: Liste `lst`, position i, valeur v

Résultat: Adresse du nouveau maillon ; peut modifier `lst`

```
1: if i = 0 then
2:     nv ← NouveauMaillon(v)
3:     SUIVANT(nv) ← lst
4:     lst ← ADRESSE(nv)
5:     return nv
6: end if
7: p ← lst
8: for j = 1, ..., i − 1 do
9:     p ← SUIVANT(p)
10: end for
11: return InsererApres(p, v)
```





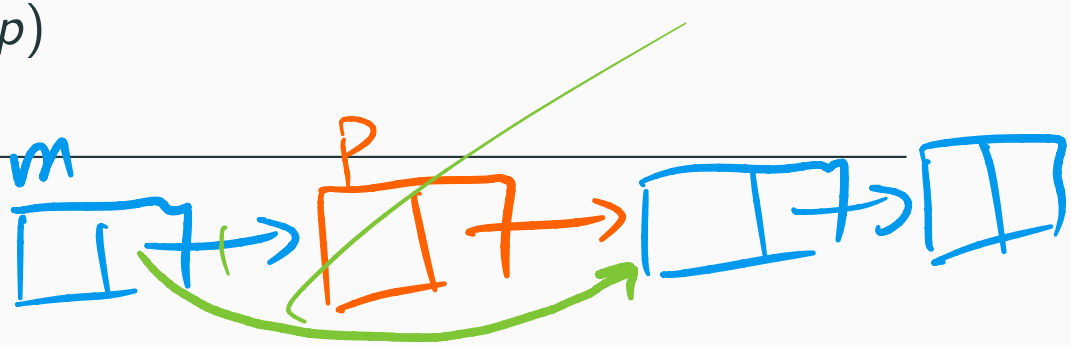
Supprimer après un maillon

Algorithme SupprimerApres

Données: maillon m

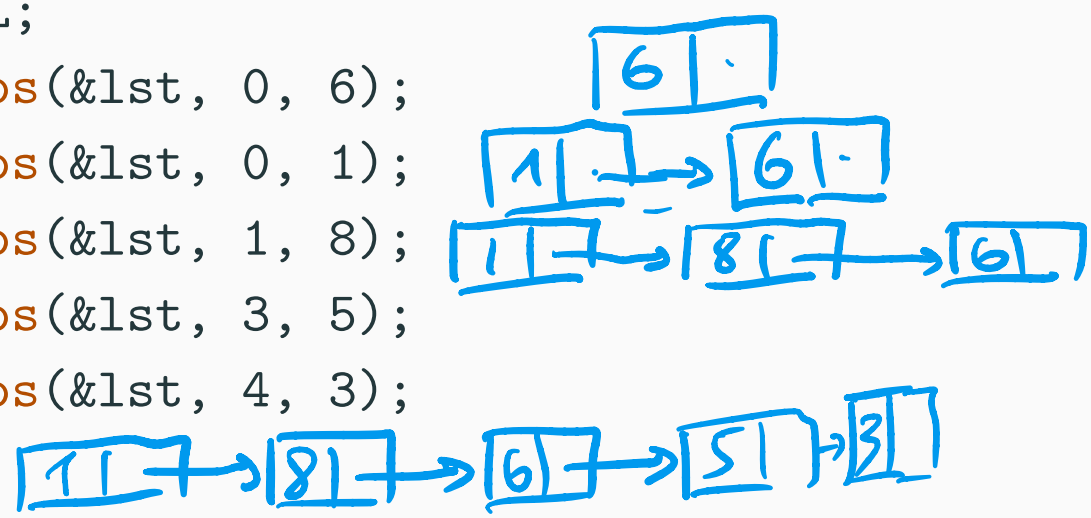
Résultat: maillon $\texttt{SUIVANT}(m)$ est supprimé de la liste

1: $p \leftarrow \texttt{SUIVANT}(m)$
2: $\texttt{SUIVANT}(m) \leftarrow \texttt{SUIVANT}(p)$
3: $\textsc{Free}(p)$



Exemple

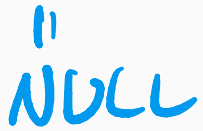
```
1  int main() {
2      liste_int lst = NULL;
3      liste_int_inserer_pos(&lst, 0, 6);
4      liste_int_inserer_pos(&lst, 0, 1);
5      liste_int_inserer_pos(&lst, 1, 8);
6      liste_int_inserer_pos(&lst, 3, 5);
7      liste_int_inserer_pos(&lst, 4, 3);
8
9      liste_int_afficher(lst);
10     liste_int_supprimer(lst);
11 }
```



Exemple

Résultat :

```
1 Maillon @A169B0 : valeur  1, suivant  A169D0
2 Maillon @A169D0 : valeur  8, suivant  A16990
3 Maillon @A16990 : valeur  6, suivant  A169F0
4 Maillon @A169F0 : valeur  5, suivant  A16A10
5 Maillon @A16A10 : valeur  3, suivant  0
```



Les listes chaînées

Parcours d'une liste

Parcours itératif – AficherListe

Algorithme AfficherListe

Données: Liste `lst`

```
1: p ← lst
2: while p ≠ NULL do
3:     Print("Maillon", p, " : ",
           "valeur", CONTENU(p), ",",
           "suivant", SUIVANT(p))
4:     p ← SUIVANT(p)
5: end while
```

Parcours récursif – SupprimerListe

Algorithme SupprimerListe

Données: Liste `lst`

```
1: if lst ≠ NULL then
2:     SupprimerListe(SUIVANT(lst))
3:     Free(lst)
4: end if
```

Les listes chaînées

Exercices

Exercice 1

Algorithme Func1

Données: Liste `lst`

```
1: if lst = NULL then
2:     return 0
3: else
4:     return 1 + FUNC1(SUIVANT(lst))
5: end if
```

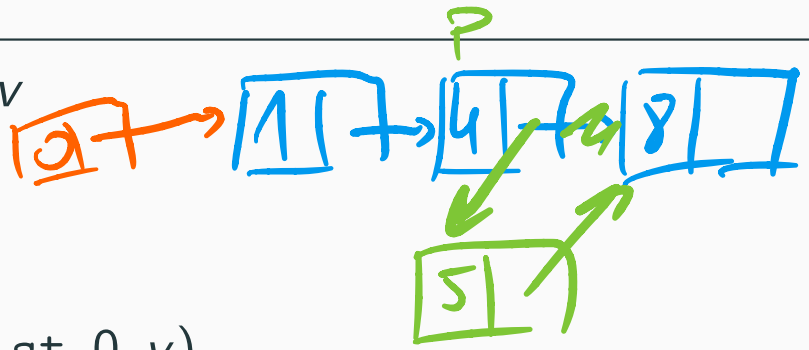
$\Theta(n)$

↳ où n est le nombre d'éléments.

Exercice 2

Algorithme Func2

Données: Liste `lst`, entier v



1: $p \leftarrow$ `lst`
2: **if** $v <$ CONTENU(p) **then**
3: INSERERPOSITION(`lst`, 0, v)
4: **else**
5: **while** SUIVANT(p) $\neq$ NULL $\wedge$ $v \geq$ CONTENU(SUIVANT(p)) **do**
6: $p \leftarrow$ SUIVANT(p)
7: **end while**
8: INSERERAPRES(p, v)
9: **end if**

$\Theta(n)$

Exercice 3

Supprimer le premier maillon avec contenu supérieur ou égal à x, dans une liste triée.

Listes – Complexité

Complexité des opérations sur des listes :

- Créer : $\Theta(1)$ lst = NULL
- Accès à l'élément à la position i : $\Theta(i) \in O(n)$
- Insertion ou suppression :
  - Après un maillon donné : $\Theta(1)$
  - À la position i : $\Theta(i)$
  - Du dernier élément : $\Theta(n)$
- Trier :

Listes – Complexité

Complexité des opérations sur des listes :

- Créer : $\Theta(1)$
- Accès à l'élément à la position i : $\Theta(i)$
- Insertion ou suppression :
  - Après un maillon donné : $\Theta(1)$
  - À la position i : $\Theta(i)$
  - Du dernier élément : $\Theta(n)$
- Trier : $\Theta(n \log n)$