Algorithmique des structures arborescentes

L2 Info et Math-info, 2017–18

Marc Zeitoun

1er février 2018

Moodle

- Exercices Moodle des semaines 1, 2, 3 **clôturés le 11 février**.
- **Lire attentivement** les définitions (polycopié, diapos).
- Se référer aux énoncés **sous Moodle** uniquement.
- **Supprimer** la définition initiale (et incorrecte) des fonctions !

Plan

Complexité des algorithmes

Arbres binaires de recherche

Plan

Complexité dans le cas le pire d'un algorithme

Il est souhaitable qu'un algorithme soit correct **et efficace**.

- La **complexité** d'un algorithme mesure son **efficacité**.

Complexité dans le cas le pire d'un algorithme

Il est souhaitable qu'un algorithme soit correct **et efficace**.

- La **complexité** d'un algorithme mesure son **efficacité**.
- Un algorithme doit fonctionner sur de **multiples entrées**.
- Intuitivement, plus une entrée est grande, plus l'algorithme risque de mettre du temps pour la traiter.

Complexité dans le cas le pire d'un algorithme

Il est souhaitable qu'un algorithme soit correct **et efficace**.

- La **complexité** d'un algorithme mesure son **efficacité**.
- Un algorithme doit fonctionner sur de **multiples entrées**.
- Intuitivement, plus une entrée est grande, plus l'algorithme risque de mettre du temps pour la traiter.
- La **complexité dans le cas le pire** d'un algorithme est une fonction qui à $n \in \mathbb{N}$ associe le nombre **maximal** d'opérations élémentaires faites par l'algorithme sur les entrées **de taille** n.
- Opérations considérées comme **élémentaires** : affectation, arithmétique $+, -, *, /$ et comparaisons d'entiers, etc.

Évaluer la complexité d'un algorithme

Comment évaluer une complexité ?

- Exemple : tri par insertion

Tri par insertion, version récursive

Suppose: i et j sont 2 positions de t

▷ *Trie t entre les positions i et j*

1: **TRI-INSERTION**(t, i, j) :
2: **si** i $<$ j **alors**
3: Tri-Insertion(t, i, j $-$ 1)
▷ *Insère t[j] à sa place*
4: INSERTION(t, i, j)
5: **fin si**

Tri par insertion, version récursive

Suppose: i et j sont 2 positions de t

▷ *Trie t entre les positions i et j*

1: **TRI-INSERTION**(t, i, j) :
2: **si** i $<$ j **alors**
3: Tri-Insertion(t, i, j $-$ 1)
▷ *Insère t[j] à sa place*
4: INSERTION(t, i, j)
5: **fin si**

Il faut d'abord évaluer la complexité de l'insertion.

Insertion

Suppose: t est déjà trié entre i et $j-1$

▷ *Insère t[j] à sa place*

INSERTION(t, i, j) :

clé $\leftarrow$ t[j]

tant que $i < j$ et $t[j-1] >$ clé **faire**

$t[j] \leftarrow t[j-1]$

$t[j-1] \leftarrow$ clé

$j \leftarrow j-1$

fin tant que

Pour comprendre cet algorithme : le faire tourner pas à pas.

Visualisation de l'algorithme

Sites de **visualisation** d'exécutions d'algorithmes.

- `https://visualgo.net/en/`.
  - `https://visualgo.net/en/sorting`
- `https://www.cs.usfca.edu/~galles/visualization`
  - `https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html`

Nombreux algorithmes sur les **arbres**.

Tri par insertion récursif : complexité

- $c(n) \stackrel{\text{def}}{=}$ nombre d'échanges pour trier t, dans le cas le pire, quand i $= 1$ et j $= n$.

Tri par insertion récursif : complexité

- $c(n) \stackrel{\text{def}}{=}$ nombre d'échanges pour trier t, dans le cas le pire, quand i $= 1$ et j $= n$.
- Au pire, on doit ramener t[n] en position 1 $\Rightarrow (n-1)$ échanges.

Tri par insertion récursif : complexité

- $c(n) \stackrel{\text{def}}{=}$ nombre d'échanges pour trier t, dans le cas le pire, quand i $= 1$ et j $= n$.
- Au pire, on doit ramener t[n] en position 1 $\Rightarrow (n-1)$ échanges.
- Relation vérifiée par c ?

Tri par insertion récursif : complexité

- $c(n) \stackrel{\text{def}}{=}$ nombre d'échanges pour trier t, dans le cas le pire, quand i $= 1$ et j $= n$.
- Au pire, on doit ramener t[n] en position 1 $\Rightarrow (n-1)$ échanges.
- Relation vérifiée par c ?

$$c(n) = \underbrace{c(n-1)}_{\substack{\text{Appel récursif} \\ \text{(ligne 3)}}} + \underbrace{(n-1)}_{\substack{\text{nombre d'échanges} \\ \text{dans INSERTION}}}$$

Remarque : cette relation est obtenue en suivant **l'algorithme**.

Tri par insertion récursif : complexité

- $c(n) \stackrel{\text{def}}{=}$ nombre d'échanges pour trier t, dans le cas le pire, quand i $= 1$ et j $= n$.
- Au pire, on doit ramener t[n] en position 1 $\Rightarrow (n-1)$ échanges.
- Relation vérifiée par c ?

$$c(n) = \underbrace{c(n-1)}_{\substack{\text{Appel récursif} \\ \text{(ligne 3)}}} + \underbrace{(n-1)}_{\substack{\text{nombre d'échanges} \\ \text{dans INSERTION}}}$$

Remarque : cette relation est obtenue en suivant **l'algorithme**.

- Par récurrence : $c(n) = \dfrac{n(n-1)}{2}$.

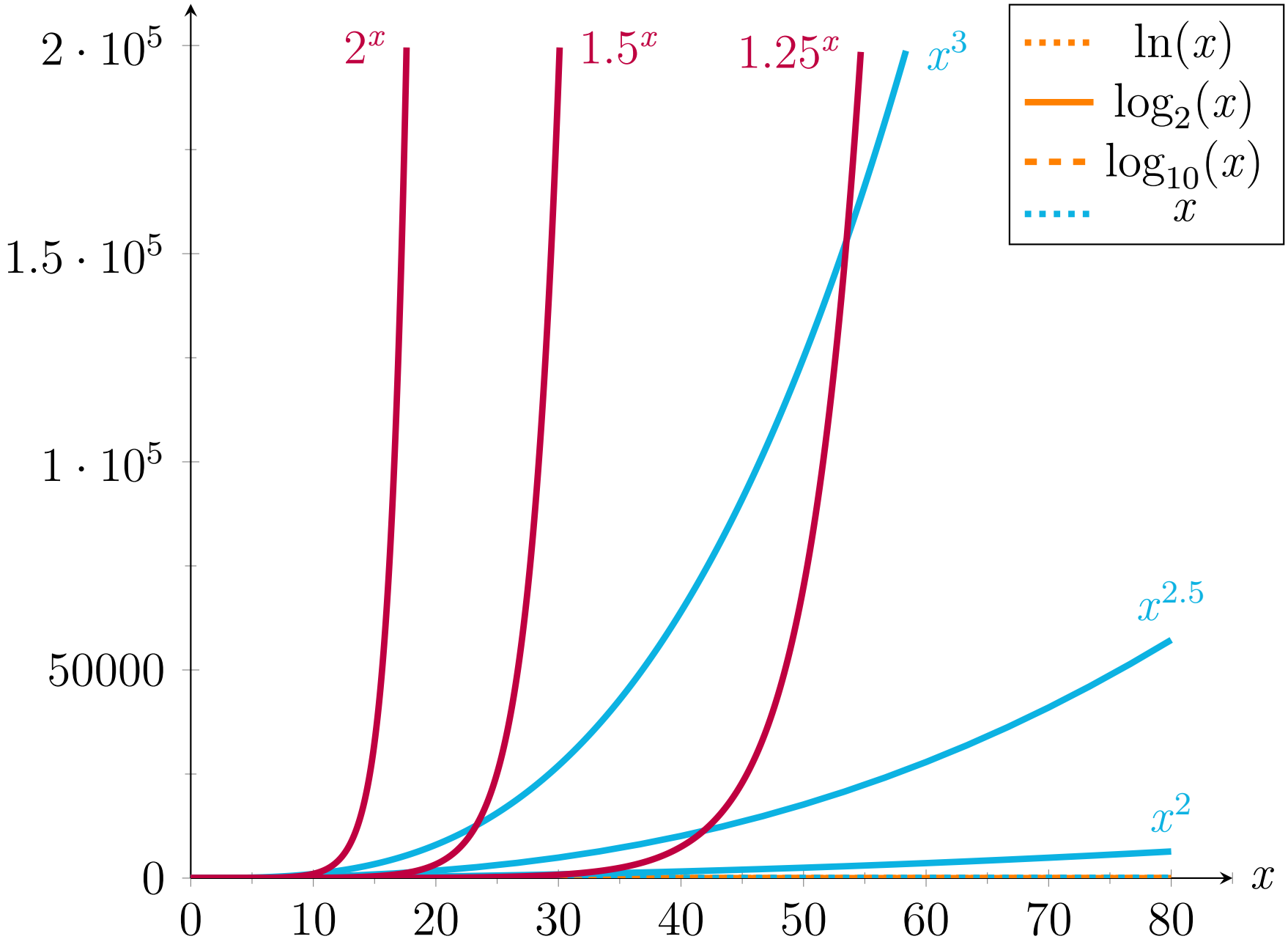
Fonctions utiles

Dans ce cours, les fonctions de complexité sont construites à l'aide :

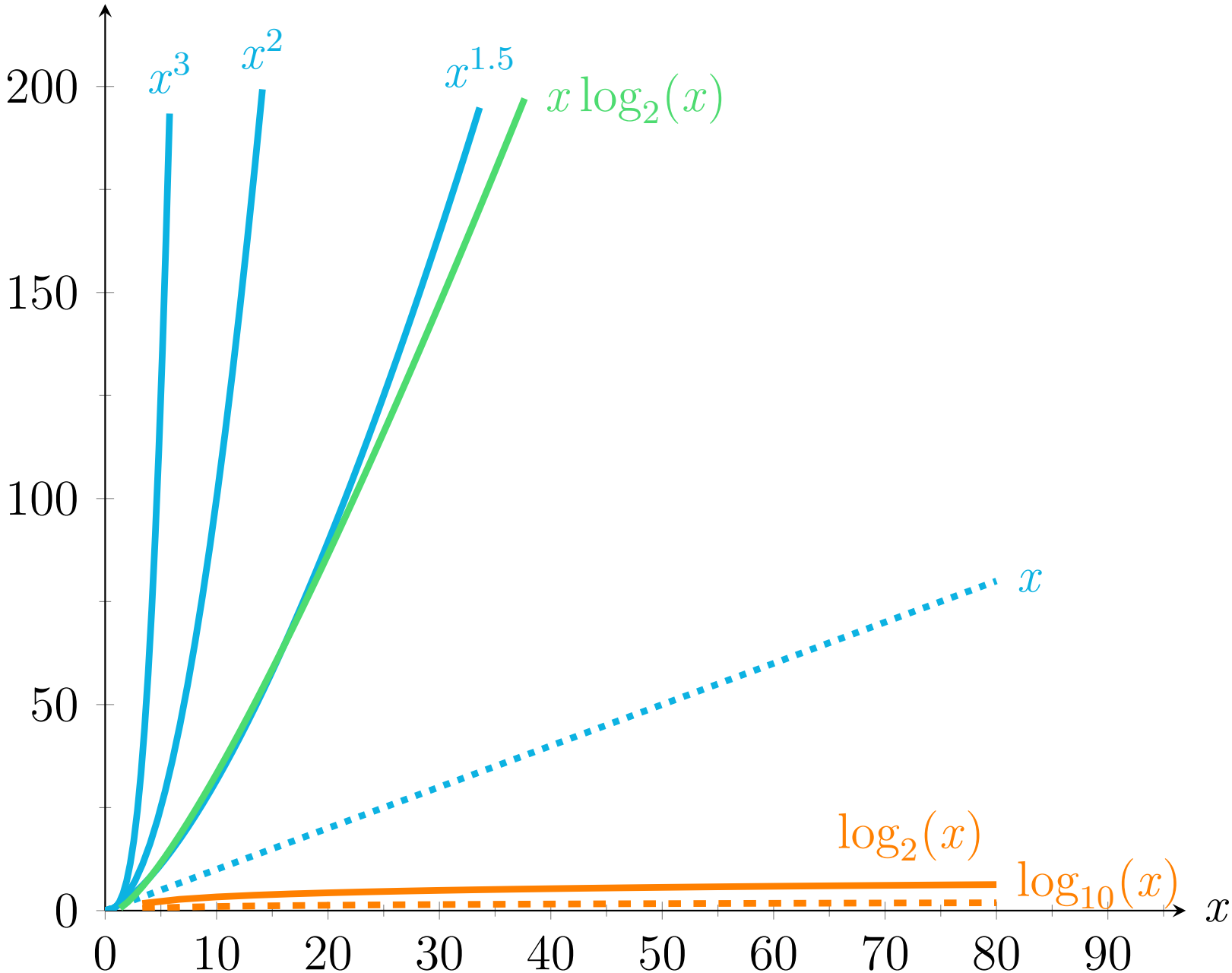
- des fonctions puissance $n \mapsto n^k$, souvent pour $k \in \mathbb{N}$,
- des fonctions logarithme $n \mapsto \log_2(n)$ ou $n \mapsto \log_{10}(n)$.
- des fonctions exponentielle $n \mapsto a^n$ pour $a > 1$, souvent entier.

Il faut avoir une intuition de leur rapidité de croissance.

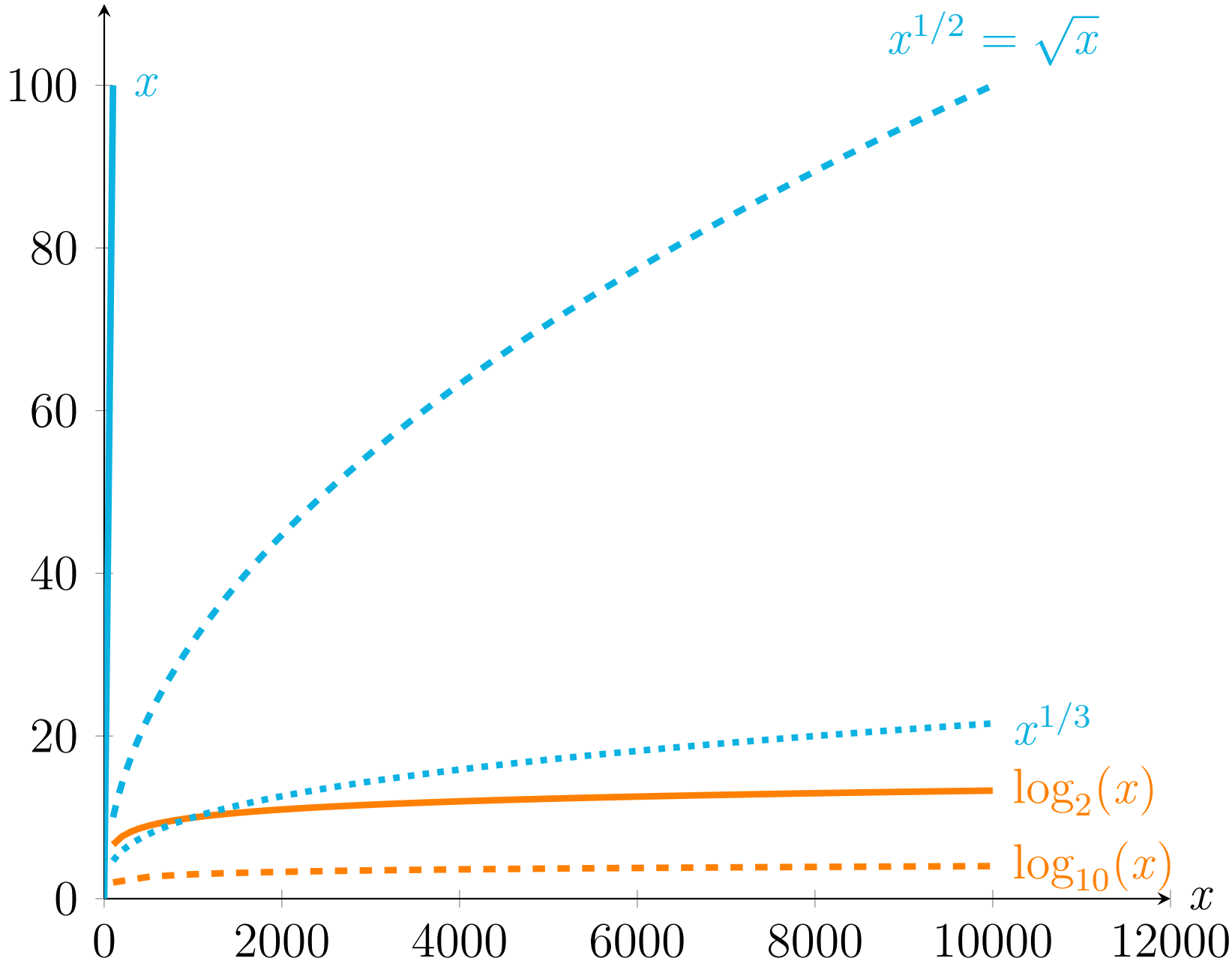
Croissance des fonctions classiques



Croissance des fonctions classiques (2)



Croissance des fonctions classiques (3)



Fonction logarithme

A retenir :

1. $\lfloor \log_{10}(x) \rfloor$ = (nombre de chiffres de x, moins 1).
   où $\lfloor x \rfloor$ = partie entière de x ($\lfloor 3\,,14 \rfloor = \lfloor 3\,,9 \rfloor = \lfloor 3 \rfloor = 3$).
   Par exemple : $6 \leqslant \log_{10}(1234567) < 7$.

Fonction logarithme

A retenir :

1. $\lfloor \log_{10}(x) \rfloor$ = (nombre de chiffres de x, moins 1).
   où $\lfloor x \rfloor$ = partie entière de x ($\lfloor 3\,,14 \rfloor = \lfloor 3\,,9 \rfloor = \lfloor 3 \rfloor = 3$).
   Par exemple : $6 \leqslant \log_{10}(1234567) < 7$.

2. Provient de : $\log_{10}(10^x) = x$.

Fonction logarithme

A retenir :

1. $\lfloor \log_{10}(x) \rfloor$ = (nombre de chiffres de x, moins 1).
   où $\lfloor x \rfloor$ = partie entière de x ($\lfloor 3\,,14 \rfloor = \lfloor 3\,,9 \rfloor = \lfloor 3 \rfloor = 3$).
   Par exemple : $6 \leqslant \log_{10}(1234567) < 7$.

2. Provient de : $\log_{10}(10^x) = x$.

3. $\log_2(x) = \log_2(10) \cdot \log_{10}(x) \simeq 3\,,32 \log_{10}(x)$.
   - Les fonctions $\log_2$ et $\log_{10}$ sont proportionnelles.
     $\Rightarrow$ On parle souvent du « **log** » sans préciser la base.

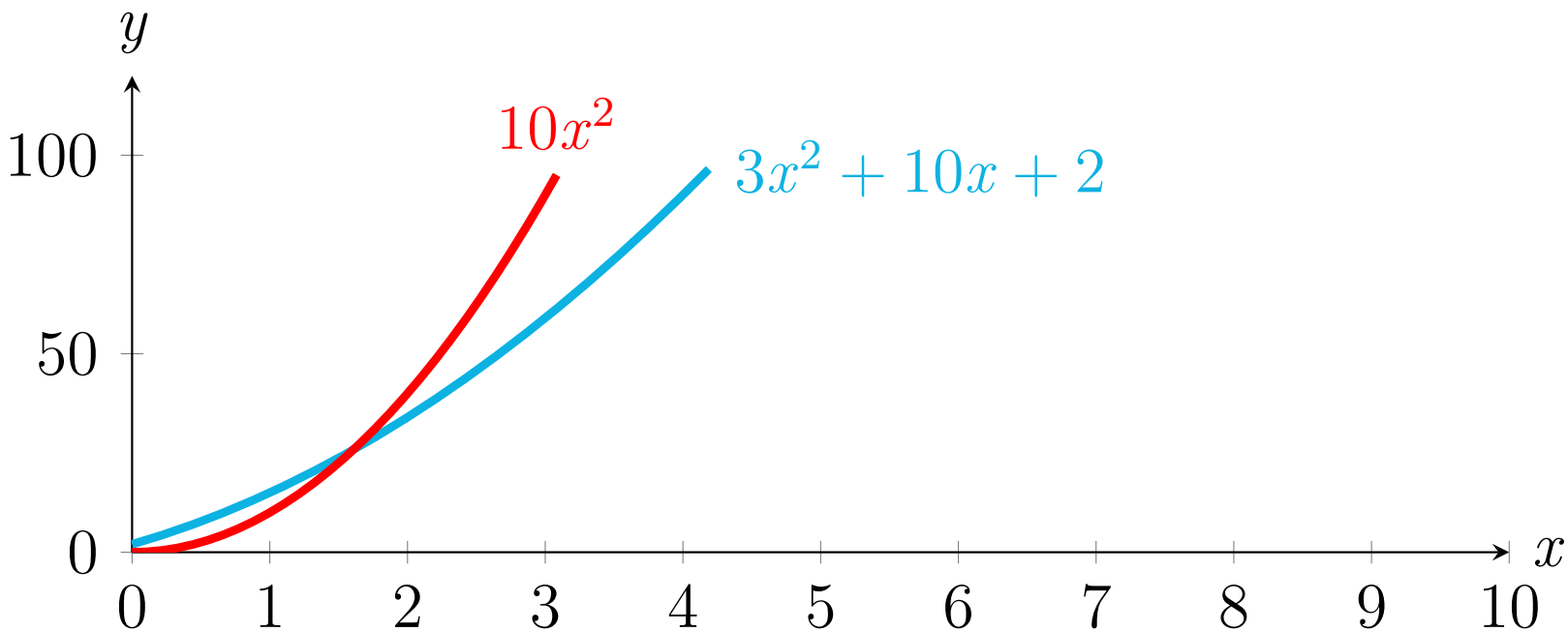
4. $\log_2(x)$ croît **moins vite que toute fonction** x^a avec $a > 0$.
   - Formellement, $\lim\limits_{x \to \infty} \dfrac{\log_2^b(x)}{x^a} = 0$ si $a > 0$.

Notation O()

Si f, g sont des fonctions de $\mathbb{N}$ dans $\mathbb{N}$, on écrit $f = O(g)$ si

$$\exists C > 0,\ \exists N > 0,\ \forall n, \quad n \geqslant N \implies f(n) \leqslant Cg(n)$$

- **Exemple** : $3n^2 + 10n + 2 = O(n^2)$ car à partir de $N = 2$, $3n^2 + 10n + 2 \leqslant 10n^2$.



Utilisation de la notation $O()$

Sert à **simplifier** des expressions de façon **raisonnable**.

- Si $n \geqslant 1$, on a $7n^2 + 2n + 1 \leqslant 7n^2 + 2n^2 + n^2 = 10n^2$ donc

Utilisation de la notation $O()$

Sert à **simplifier** des expressions de façon **raisonnable**.

- Si $n \geqslant 1$, on a $7n^2 + 2n + 1 \leqslant 7n^2 + 2n^2 + n^2 = 10n^2$ donc

$$7n^2 + 2n + 1 = O(n^2)$$

- De même, si $n \geqslant 1$:
  $25n^7 + 12n^3 + 4n\log_2(n) + 1 \leqslant 25n^7 + 12n^7 + 4n^7 + n^7 = 42n^7$,
  donc

Utilisation de la notation $O()$

Sert à **simplifier** des expressions de façon **raisonnable**.

- Si $n \geqslant 1$, on a $7n^2 + 2n + 1 \leqslant 7n^2 + 2n^2 + n^2 = 10n^2$ donc

$$7n^2 + 2n + 1 = O(n^2)$$

- De même, si $n \geqslant 1$:
  $25n^7 + 12n^3 + 4n\log_2(n) + 1 \leqslant 25n^7 + 12n^7 + 4n^7 + n^7 = 42n^7$,
  donc

$$25n^7 + 12n^3 + 4n\log_2(n) + 1 = O(n^7)$$

Utilisation de la notation $O()$ – (2)

- Si on a calculé une complexité en $n \log_2(n)$, on n'utiliserait pas $n \log_2(n) = O(n^2)$, même si c'est vrai, parce que
  - $n \log_2(n)$ est une expression simple,
  - on perd trop de précision en majorant par n^2.

Utilisation de la notation $O()$ – (2)

- Si on a calculé une complexité en $n \log_2(n)$, on n'utiliserait pas $n \log_2(n) = O(n^2)$, même si c'est vrai, parce que
  - $n \log_2(n)$ est une expression simple,
  - on perd trop de précision en majorant par n^2.

- Par contre, $\frac{1}{2}n^2 + 7n \log_2(n) + 12n + 8 \leqslant 28n^2$ si $n \geqslant 1$, donc

$$\frac{1}{2}n^2 + 7n \log_2(n) + 12n + 8 = O(n^2)$$

Le terme dominant était $\frac{1}{2}n^2$, on ne perd pas de précision en écrivant ici $n \log_2(n) = O(n^2)$.

Notation O() : formules utiles

- Complexité $O(1) =$ temps constant.
- Si $a < b$, on a $n^a = O(n^b)$.
- Si $a > 0$ et $c > 1$, on a $\log_c(n) = O(n^a)$.
- si $a > 0$ et $c > 1$, on a $n^a = O(c^n)$.

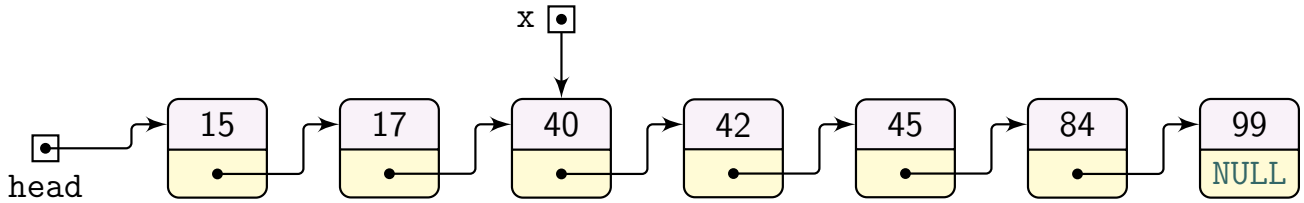
Plan

Tableaux vs. listes

	Tableau	**Tableau trié**	**Liste**	**Liste triée**
Recherche	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$
Minimum	$O(n)$	$O(1)$	$O(n)$	$O(1)$
Insertion	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Suppression	$O(n)$	$O(n)$	$O(n)$	$O(n)$

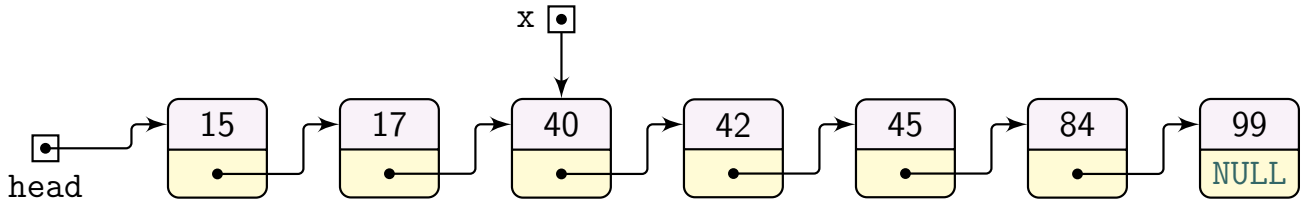
Suppression dans une liste en $O(n)$

Le coût pour supprimer une valeur provient de sa **recherche**.
Si on a déjà un pointeur sur la cellule à supprimer, le coût est $O(1)$.
De même, l'insertion coûte $O(n)$ si on veut un seul exemplaire de chaque valeur.

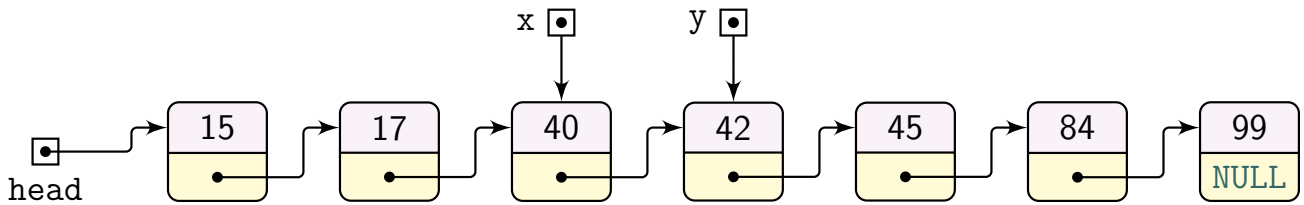


Suppression dans une liste en $O(n)$

Le coût pour supprimer une valeur provient de sa **recherche**.
Si on a déjà un pointeur sur la cellule à supprimer, le coût est $O(1)$.
De même, l'insertion coûte $O(n)$ si on veut un seul exemplaire de chaque valeur.

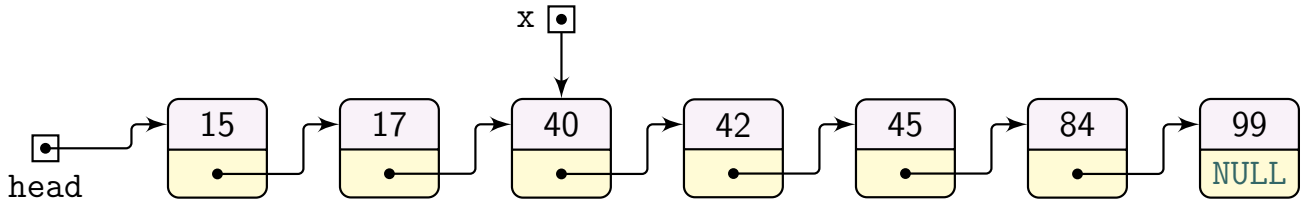



```
y = x->next;
```

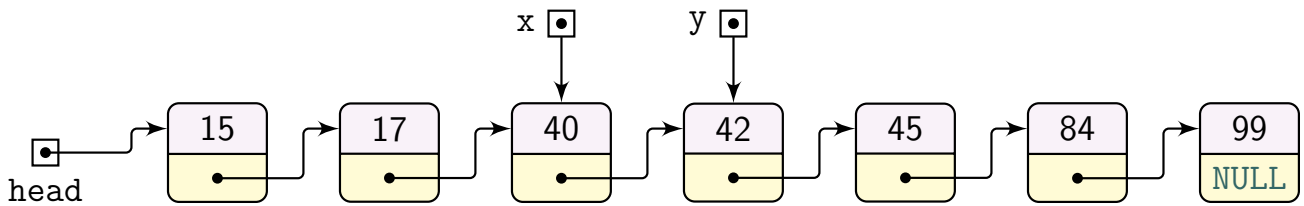


Suppression dans une liste en $O(n)$

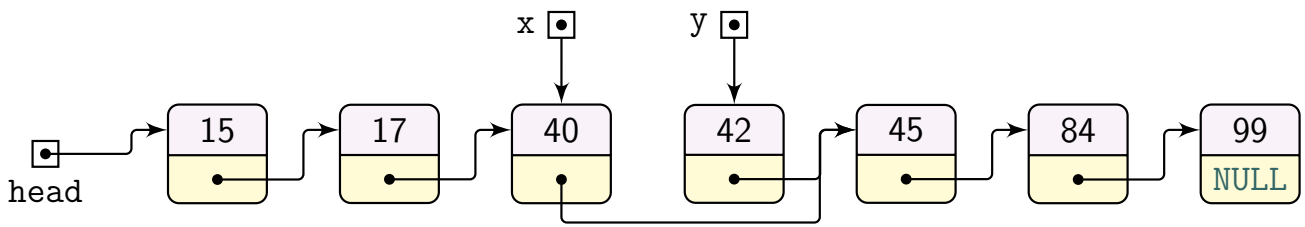
Le coût pour supprimer une valeur provient de sa **recherche**.
Si on a déjà un pointeur sur la cellule à supprimer, le coût est $O(1)$.
De même, l'insertion coûte $O(n)$ si on veut un seul exemplaire de chaque valeur.




```
y = x->next;
```

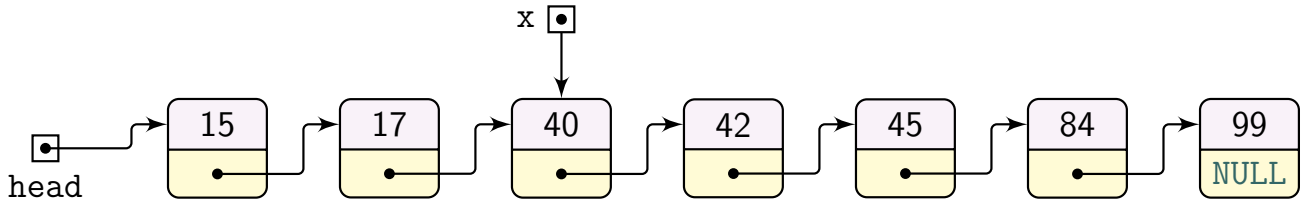



```
x->next = y->next;
```

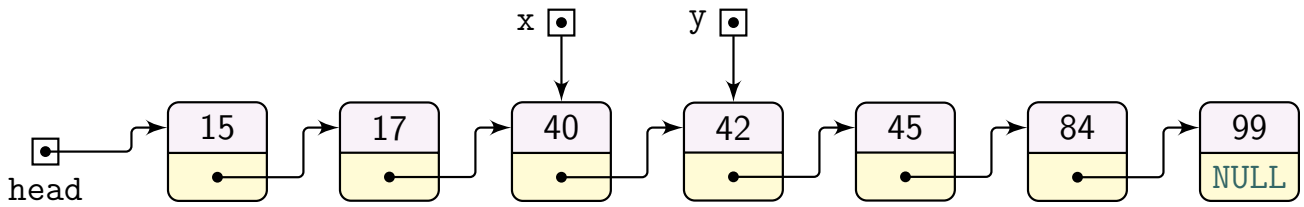


Suppression dans une liste en $O(n)$

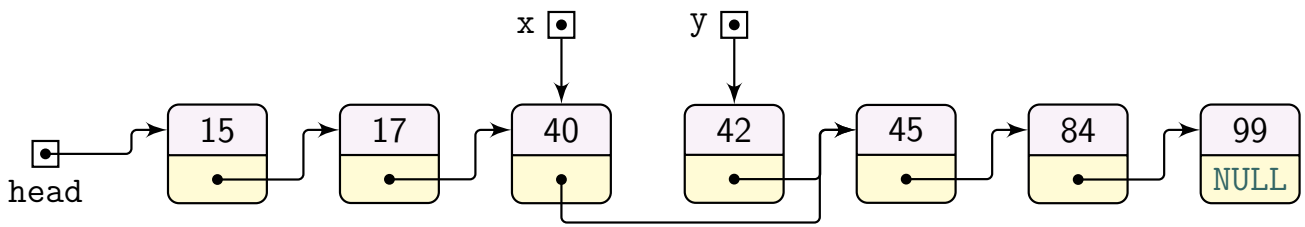
Le coût pour supprimer une valeur provient de sa **recherche**.
Si on a déjà un pointeur sur la cellule à supprimer, le coût est $O(1)$.
De même, l'insertion coûte $O(n)$ si on veut un seul exemplaire de chaque valeur.



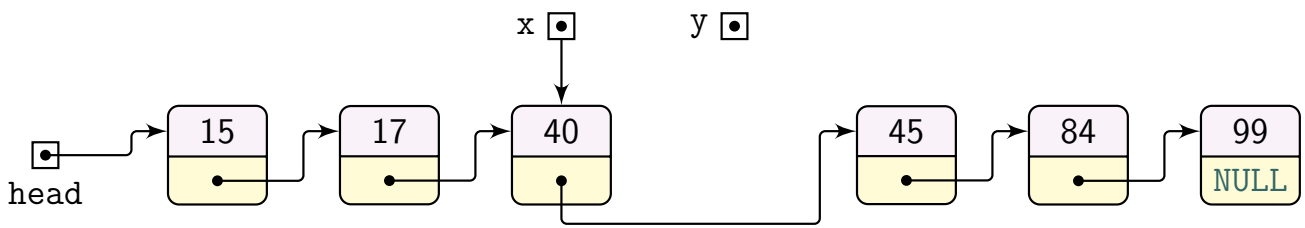

```
y = x->next;
```




```
x->next = y->next;
```




```
free(y);
```



Tableaux vs. listes

	Tableau	**Tableau trié**	**Liste**	**Liste triée**
Recherche	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$
Minimum	$O(n)$	$O(1)$	$O(n)$	$O(1)$
Insertion	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Suppression	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Objectif

Obtenir une complexité au pire $O(\log n)$
pour chacune de ces opérations.

Comparaison entre $O(n)$ à $\log_2(n)$

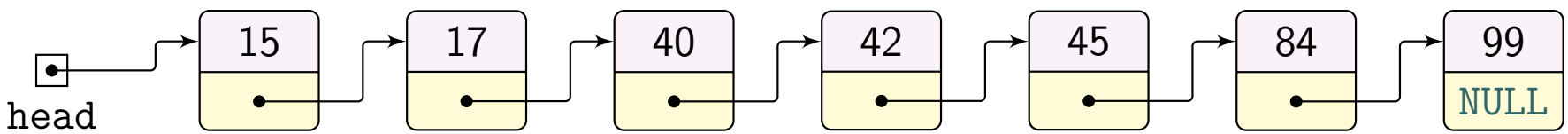
- On suppose 10^9 opérations par seconde (1GHz).
- Temps nécessaire pour 10^6 recherches en $O(n)$ et $O(\log_2(n))$.

Taille	10^6	10^9	10^{12}
n	16mn.	11.5 jours	31.5 ans
$\log_2(n)$	0,02s.	0,03s.	0,04s.

Des structures linéaires aux arbres

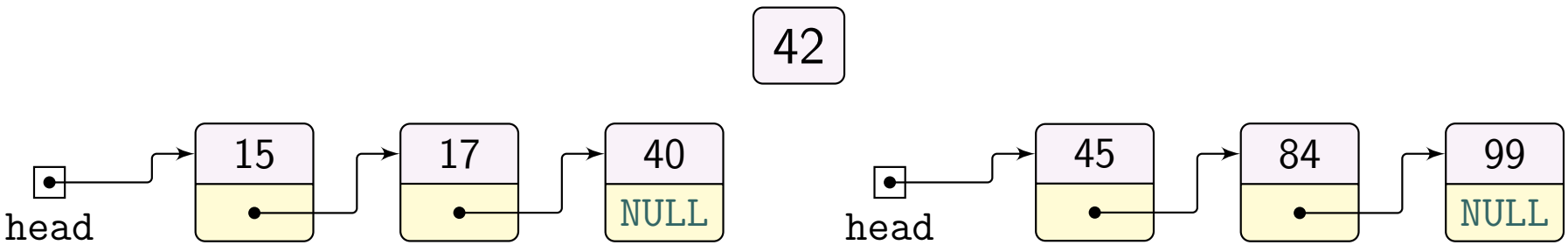
Objectif : avantages des listes et des tableaux.

- comme pour les listes : ajout, suppression d'1 élément efficace.
- comme pour les tableaux : dichotomie.

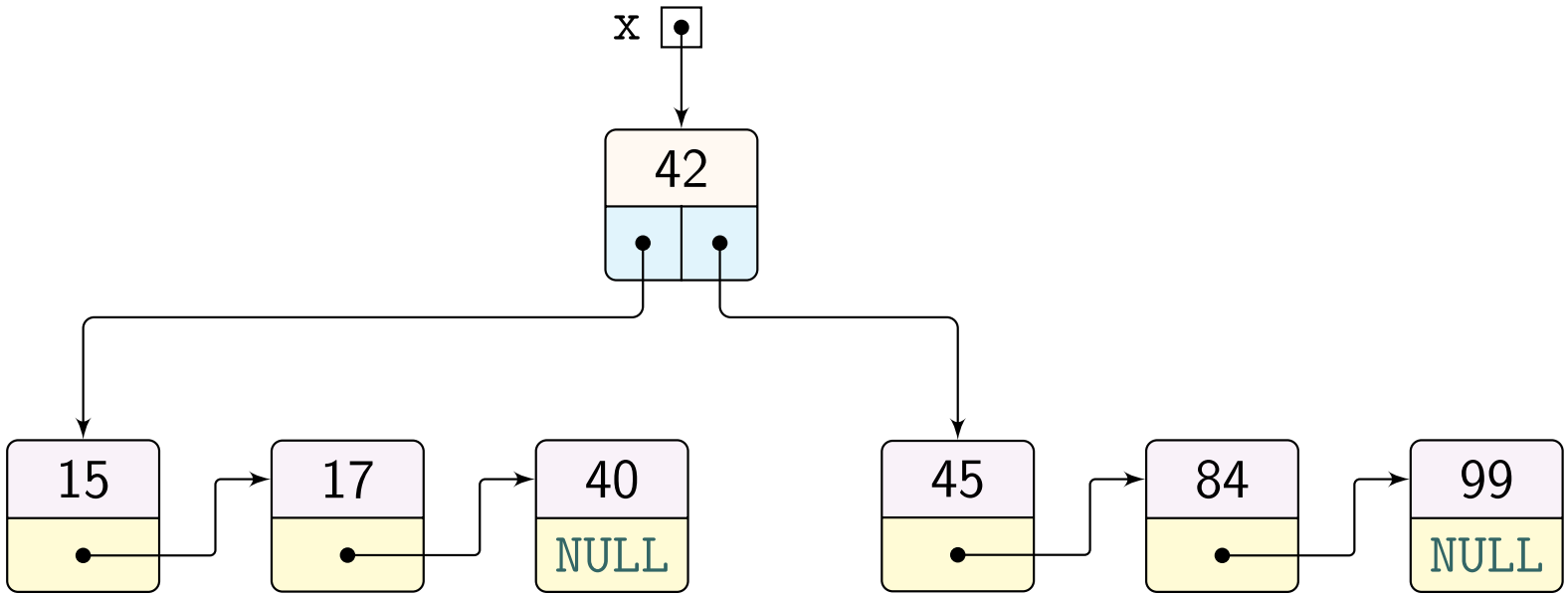


Dichotomie sur les listes

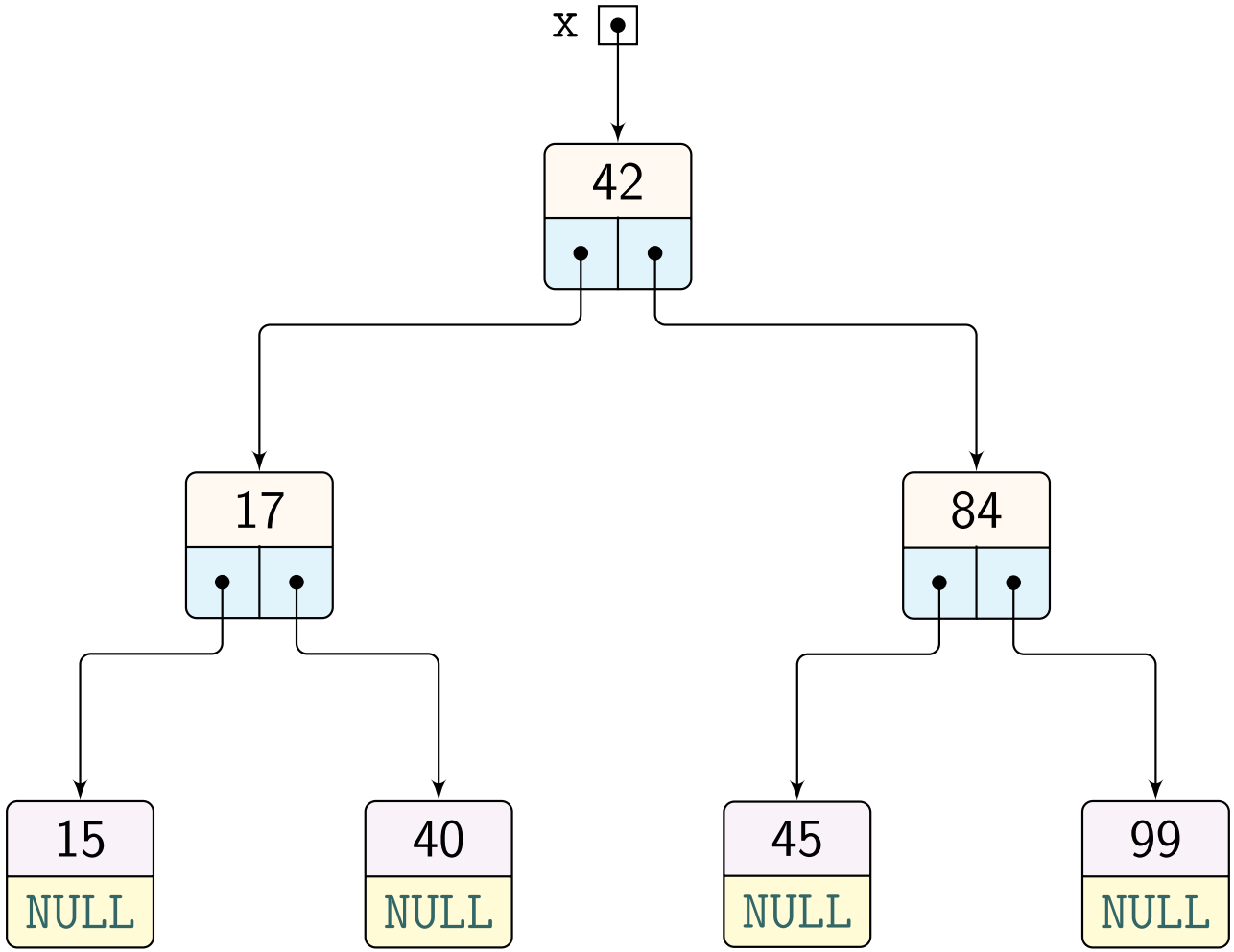
- Dichotomie : nécessite un accès efficace au milieu de liste.



Une organisation possible



Itérer cette idée conduit à un arbre



Les arbres binaires de recherche (ABR)

- Partant d'une liste **triée**, on obtient un **arbre particulier**.
- En **tout** nœud x :
  - si y est dans le sous-arbre **gauche** de x :

$$\text{étiquette}(y) \leqslant \text{étiquette}(x)$$

  - si y est dans le sous-arbre **droit** de x :

$$\text{étiquette}(y) \geqslant \text{étiquette}(x)$$

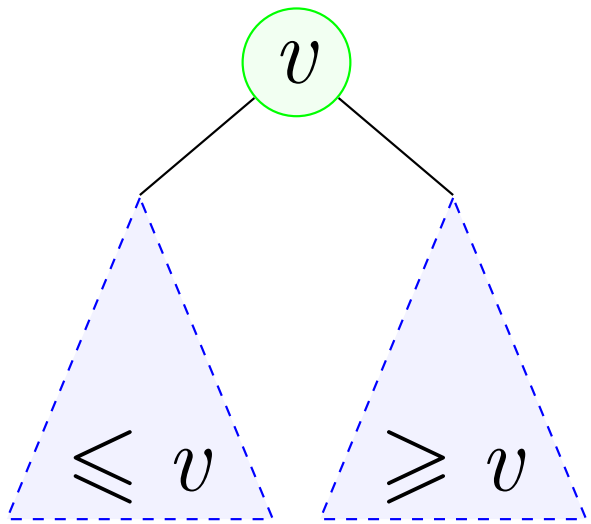
Cette propriété ne se vérifie pas uniquement « localement ».

ABR

La propriété d'être un arbre binaire de recherche n'est **pas locale**.

Pour **chaque nœud** v, elle implique :

- **tous les nœuds** du sous-arbre gauche de v,
- **tous les nœuds** du sous-arbre droit de v.



ABR : exemples