IN 101 - Cours 10

25 novembre 2011

présenté par
Matthieu Finiasz

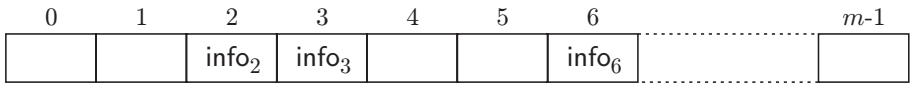
Le problème de la recherche en table

- Le problème : retrouver une information à partir d'une clef qui lui est associée.
- Par exemple :
  - dictionnaire : mot ⇾ définition,
  - annuaire : nom ⇾ adresse, téléphone,
  - bibliothèque : numéro ⇾ ouvrage.
- Il faut les fonctionnalités suivantes :
  - recherche,
  - insertion,
  - suppression.

⇾ Quelle structure de données utiliser ?

- Les clefs forment un ensemble de m éléments indexés de 0 à $m-1$
  - pour des mots de 10 lettres : $m = 26^{10} \approx 2^{47}$.
- On veut gérer n paires de la forme (clef,information)
  - jamais deux clefs identiques,
  - $n \leq m$ mais en général $n \ll m$.
- Nous allons voir plusieurs solutions. On s'intéresse à :
  - la complexité spatiale du stockage,
  - la complexité temporelle de chaque opération
    ⇾ insertion, recherche, suppression.

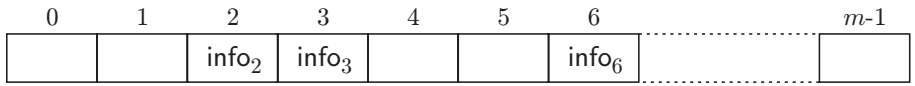
Table à adressage direct



- On utilise un tableau `tab` de taille m
  - complexité spatiale en $\Theta(m)$.
  - en fait $m \times$ taille_info ou $m + n \times$ taille_info avec des pointeurs.
- Recherche → `return tab[clef];`
  - complexité $\Theta(1)$.
- Insertion → `tab[clef] = info;`
  - complexité $\Theta(1)$.
- Suppression → `tab[clef] = NULL;`
  - complexité $\Theta(1)$.

IN101 - 2011-2012

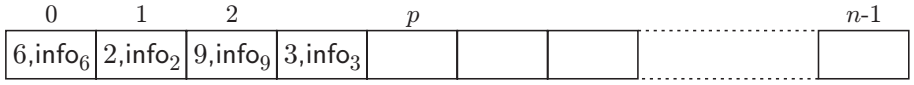
Table à adressage direct



- On utilise un tableau `tab` de taille m
  - complexité spatiale en $\Theta(m)$.
  - en fait $m \times$ taille_info ou $m + n \times$ taille_info avec des pointeurs.
- mots de 10 lettres : $m = 2^{47}$
  - même avec un seul bit d'information → 16 To de stockage...
- Une complexité spatiale en $\Theta(m)$ est irréalisable en pratique.

IN101 - 2011-2012

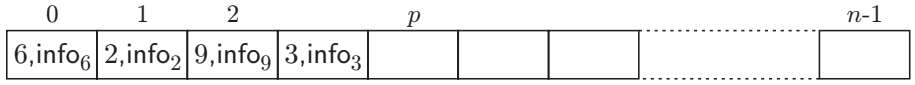
Recherche séquentielle



- On utilise un tableau `tab` de taille n et on ajoute les éléments dans l'ordre d'arrivée. L'indice p est la première case libre.
  - complexité spatiale en $\Theta(n)$ → on ne peut pas faire moins.
- Recherche → parcours de `tab` jusqu'à trouver la clef
  - complexité $\Theta(n)$.
- Insertion → `tab[p] = (clef,info);` recherche préalable : $\Theta(n)$
  - complexité $\Theta(1)$.
- Suppression → recherche, puis décalage
  - complexité $\Theta(n)$.

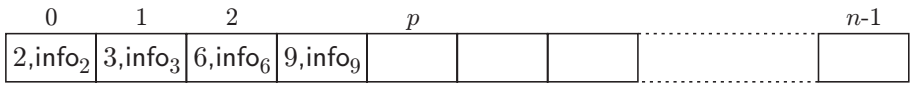
IN101 - 2011-2012

Recherche séquentielle



- On utilise un tableau `tab` de taille n et on ajoute les éléments dans l'ordre d'arrivée. L'indice p est la première case libre.
  - complexité spatiale en $\Theta(n)$ → on ne peut pas faire moins.
- La recherche est ici trop lente :
  - cette technique n'est bonne que si l'on ne fait qu'insérer des éléments sans les lire !
  - avec des comparaisons complexes cela devient trop coûteux.

Recherche dichotomique

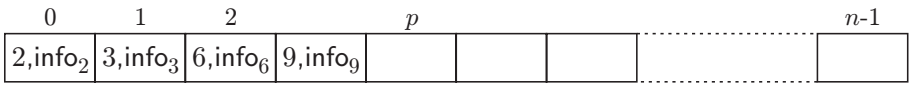


- On utilise un tableau de taille n dont les entrées sont triées.

- Recherche ⟶ parcours dichotomique de `tab` jusqu'à trouver la clef
  - complexité $\Theta(\log n)$.
- Insertion ⟶ recherche, puis décalage
  - complexité $\Theta(n)$.
- Suppression ⟶ recherche, puis décalage
  - complexité $\Theta(n)$.

IN101 - 2011-2012

Recherche dichotomique



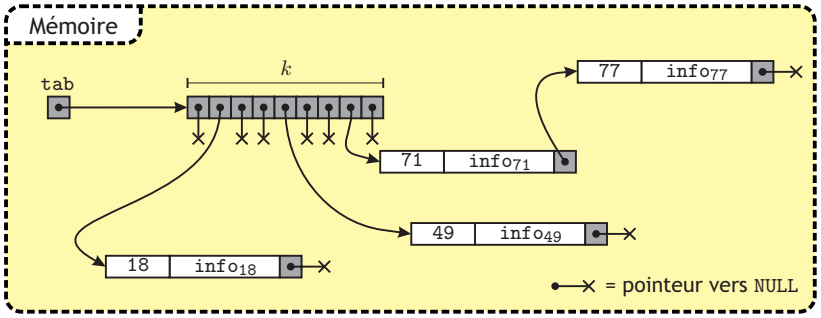
- On utilise un tableau de taille n dont les entrées sont triées.

- La complexité de la recherche est bonne,
  - l'insertion est trop chère ⟶ trier l'ensemble d'un seul coup.
  - on rend l'insertion/suppression logarithmique avec des ABR.

IN101 - 2011-2012

Table de hachage

- On définit une fonction de hachage `h` de $[0, m-1]$ dans $[0, k-1]$:
  - on utilise un tableau `tab` de taille k,
  - chaque (clef,information) va dans la case `tab[h(clef)]`
    ⟶ chaque case contient une liste de (clef,information).



IN101 - 2011-2012

Table de hachage
Complexités

- Les complexités dépendent du facteur de remplissage ρ de la table.
  - en moyenne chaque case contient $\rho = \frac{n}{k}$ éléments.

- On obtient les complexités (en moyenne) :
  - spatiale : $\Theta(k+n)$,
  - recherche : $\Theta(\rho)$,
  - insertion : $\Theta(1)$,
  - suppression : $\Theta(\rho)$.

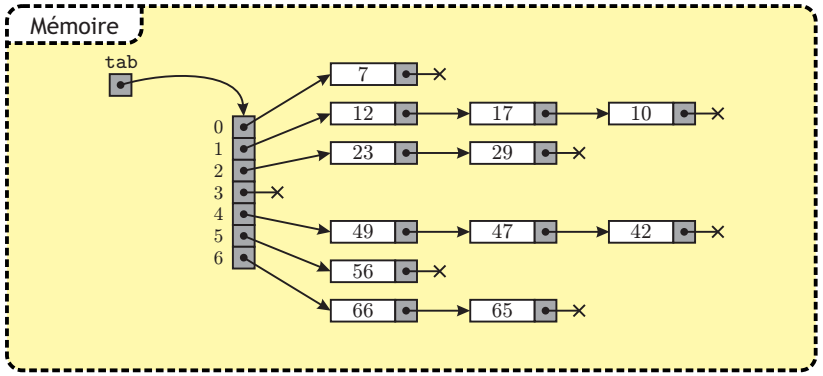
- Pour un bon fonctionnement il faut :
  - une bonne fonction de hachage (non biaisée sur les entrées)
  - connaître n à l'avance pour choisir $k \approx n$.

Une autre application des tables de hachage
Le tri par paquets

- On veut trier un tableau de n nombres compris entre 0 et $m-1$
  - cela peut se faire en temps $\Theta(n)$ en autorisant plus que des comparaisons...
- On crée une table de hachage dont la fonction de hachage :
  - possède $k = \Theta(n)$ valeurs de sortie,
  - préserve l'ordre : $x \leq y \quad \rightarrow \quad \mathtt{h}(x) \leq \mathtt{h}(y)$.
- Pour trier :
  - on insère les n éléments dans la table,
  - on trie chaque liste de la table avec un algorithme simple,
  - on met les listes bout à bout.

Une autre application des tables de hachage
Le tri par paquets

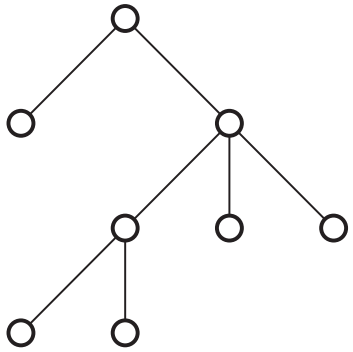
- La complexité totale est en $\Theta(n\rho^2)$, donc si k (et donc ρ) sont adaptés, on trouve bien du $\Theta(n)$.
- Par exemple, avec la fonction de hachage $\mathtt{h}(x) = \lfloor \frac{x}{10} \rfloor$:



Les arbres

Les arbres
Définitions

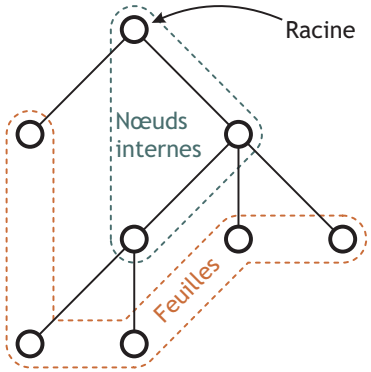
Exemple d'arbre :
- Des nœuds et des arrêtes.
- Pas de « boucles. »



Les arbres
Définitions

Vocabulaire :

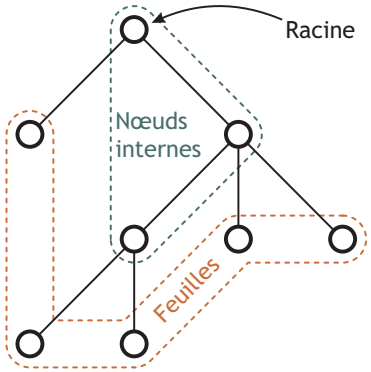
- Racine, nœud interne, feuille.
- Sous-arbre, descendant, ascendant.
- Fils, frère, père.
- Degré d'un nœud, arité d'un arbre.
- Arbre binaire, arbre k-aire.



Les arbres
Définitions

Exemples d'utilisation :

- Représentation d'expressions
  - → arithmétique, programme...
- Stockage de données hiérarchisées
  - → répertoires et fichiers...
- Plein d'autres choses !

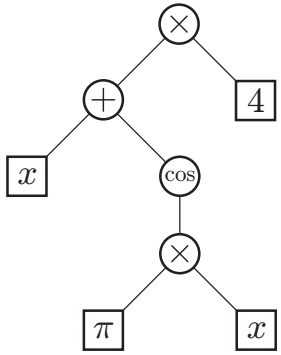


Représentation d'expressions

- On peut représenter une expression par un arbre :
  - cela permet de l'évaluer,
  - permet d'autres calculs : dérivation, simplification... (voir TD11)

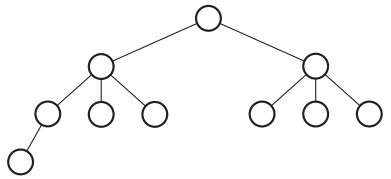
- Exemple :

$$(x + \cos(\pi \times x)) \times 4.$$



Arbres généraux
Représentation

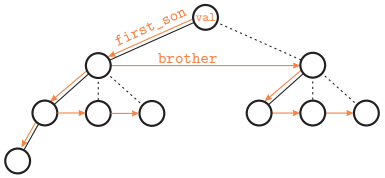
- Chaque nœud a un nombre quelconque de fils



Arbres généraux
Représentation

- Chaque nœud a un nombre quelconque de fils : on utilise une structure de liste chaînée pour les frères et on ne garde qu'un pointeur vers le premier fils.

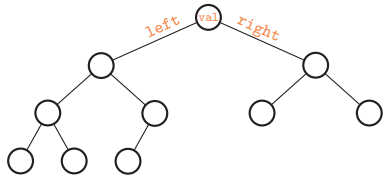
```
1 typedef struct st_node {
2   int val;
3   struct st_node* first_son;
4   struct st_node* brother;
5 } node;
```



Les arbres binaires
Définition

- Chaque nœud a au plus deux fils : un fils gauche et un fils droit.

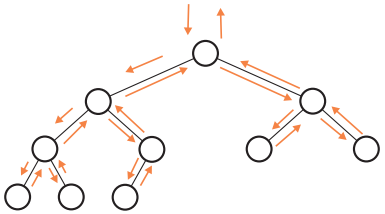
```
1 typedef struct st_node {
2   int val;
3   struct st_node* left;
4   struct st_node* right;
5 } node;
```



Les arbres binaires
Parcours en profondeur

- On descend tout au fond de l'arbre dès le début
  - différent du parcours par niveau (*breadth first*).

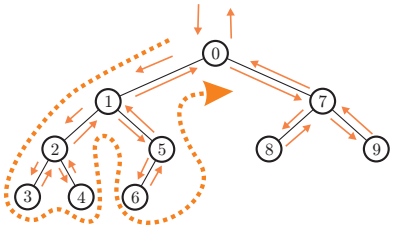
```
1 depth_first (node* T) {
2  if (T != NULL) {
3   depth_first(T->left);
4   depth_first(T->right);
5  }
6 }
```



Les arbres binaires
Parcours préfixe

- On a trois endroits pour insérer le traitement du nœud courant
  - soit au début → parcours préfixe.

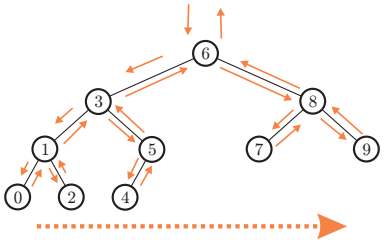
```
1 prefix (node* T) {
2  if (T != NULL) {
3   printf("%d ", T->val);
4   prefix(T->left);
5   prefix(T->right);
6  }
7  printf("\n");
8 }
```



Les arbres binaires
Parcours préfixe

- On a trois endroits pour insérer le traitement du nœud courant
  - soit au milieu → parcours infixe.
  - correspond à l'écriture normale pour une arbre d'évaluation.

```
1 infix (node* T) {
2  if (T != NULL) {
3   infix(T->left);
4   printf("%d ", T->val);
5   infix(T->right);
6  }
7  printf("\n");
8 }
```

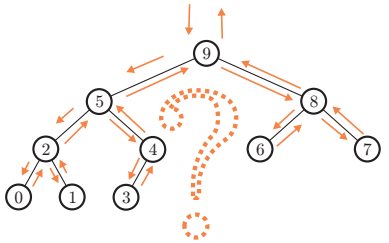


IN101 - 2011-2012

Les arbres binaires
Parcours préfixe

- On a trois endroits pour insérer le traitement du nœud courant
  - soit à la fin → parcours postfixe.
  - correspond à l'écriture polonaise inverse.

```
1 postfix (node* T) {
2  if (T != NULL) {
3   postfix(T->left);
4   postfix(T->right);
5   printf("%d ", T->val);
6  }
7  printf("\n");
8 }
```

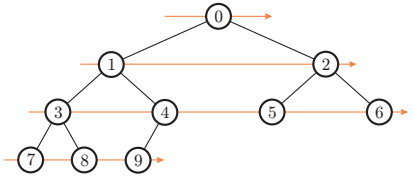


IN101 - 2011-2012

Parcours par niveaux
Parcours en largeur

- On utilise une file dans laquelle on insère/récupère les nœuds dans l'ordre. On insère la racine, puis à chaque étape on extrait un nœud, on le traite, et on insère tous ses fils.

```
1 largeur (node* T) {
2  push(T);
3  while ((T = pop()) != NULL) {
4   printf("%d ", T->val);
5   if (T->left != NULL) {
6    push(T->left);
7   }
8   if (T->right != NULL) {
9    push(T->right);
10  }
11 }
12 printf("\n");
13 }
```

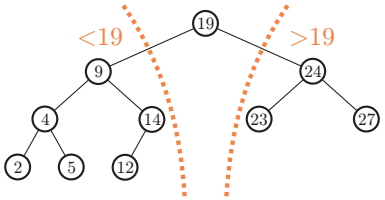


Arbres Binaires de Recherche
(ABR)

IN101 - 2011-2012

Les ABR
Définition

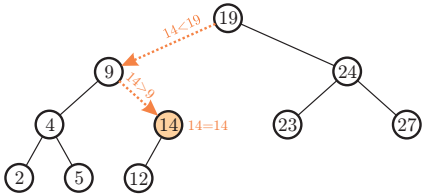
- Structure d'arbre binaire avec quelques propriétés en plus :
  - chaque nœud contient une clef
    - → élément d'un ensemble ordonné.
- La clef est d'un nœud est :
  - supérieure à celle de son fils gauche,
    - → celles du sous-arbre gauche.
  - inférieure à celle de son fils droit,
    - → celles du sous-arbre droit.



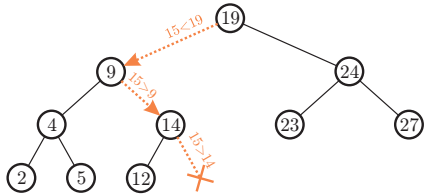
- Un parcours infixe affiche les nœuds dans l'ordre croissant.

Recherche d'une clef dans un ABR

Recherche de 14



Recherche de 15



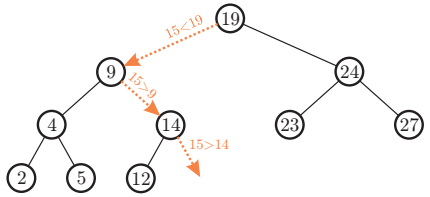
Recherche d'une clef dans un ABR

- Un algorithme qui retourne 1 si la clef v est présente dans T
  - trois cas sont possibles.

```
1  int search (int v, node* T) {
2    if (T == NULL) {
3      return 0;
4    }
5    if (v == T->val) {
6      return 1;
7    }
8    if (v < T->val) {
9      return search(v, A->left);
10   } else {
11     return search(v, A->right);
12   }
13 }
```

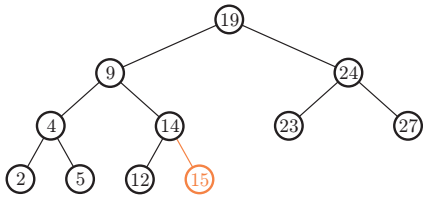

Insertion dans un ABR

- Insérer le nœud 15 dans l'arbre :
  - rechercher sa position.



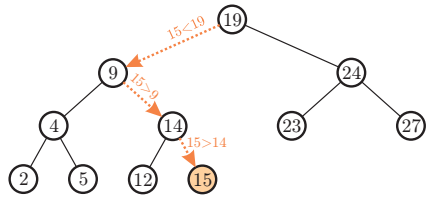
Insertion dans un ABR

- Insérer le nœud 15 dans l'arbre :
  - rechercher sa position.
  - insérer comme fils du dernier nœud rencontré.



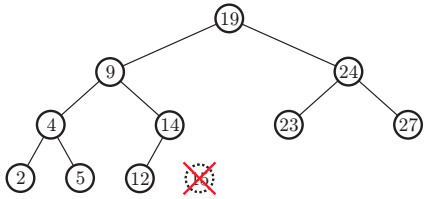
Suppression dans un ABR
Une feuille : le cas le plus simple

- Supprimer le nœud 15 :
  - rechercher sa position
  - il n'a pas de fils.



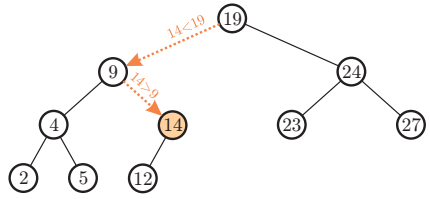
Suppression dans un ABR
Une feuille : le cas le plus simple

- Supprimer le nœud 15 :
  - rechercher sa position
  - il n'a pas de fils :
    - → on supprime le pointeur.
      (on désalloue avant)



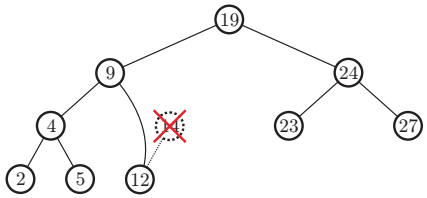
Suppression dans un ABR
Un seul fils : un autre cas simple

- Supprimer le nœud 14 :
  - rechercher sa position.
  - il n'a qu'un fils.



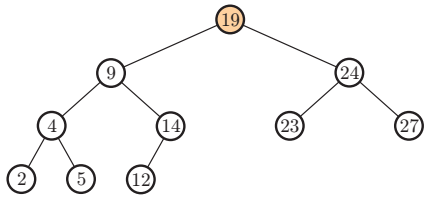
Suppression dans un ABR
Un seul fils : un autre cas simple

- Supprimer le nœud 14 :
  - rechercher sa position.
  - il n'a qu'un fils :
    → on fait pointer son père vers son fils (unique).
    (on désalloue avant)



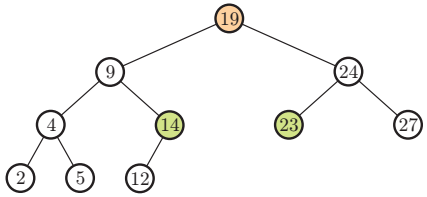
Suppression dans un ABR
Deux fils : le cas compliqué

- Supprimer le nœud 19 :
  - rechercher sa position.
  - il a deux fils.



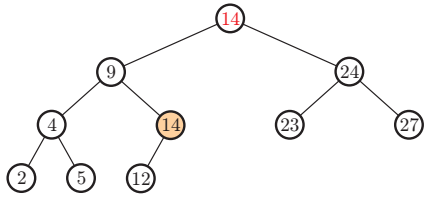
Suppression dans un ABR
Deux fils : le cas compliqué

- Supprimer le nœud 19 :
  - rechercher sa position.
  - il a deux fils.
  - le min du fils droit ou le max du fils gauche peuvent le prendre sa place.



Suppression dans un ABR
Deux fils : le cas compliqué

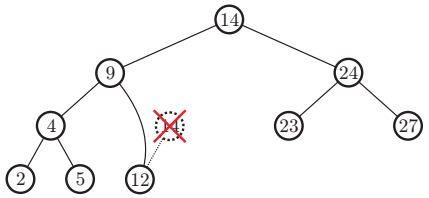
- Supprimer le nœud 19 :
  - rechercher sa position.
  - il a deux fils.
  - le min du fils droit ou le max du fils gauche peuvent le prendre sa place.
  - on recopie sa clef et on le supprime.



Suppression dans un ABR

Deux fils : le cas compliqué

- Supprimer le nœud 19 :
  - rechercher sa position.
  - il a deux fils.
  - le min du fils droit ou le max du fils gauche peuvent le prendre sa place.
  - on recopie sa clef et on le supprime :
    → comme précédemment.



Complexité des opérations sur les ABR

- Stockage en $\Theta(n)$.
- Recherche en $\Theta(h)$ (h est la hauteur de l'arbre).
- Insertion en $\Theta(h)$.
- Suppression en $\Theta(h)$.

- En moyenne $h = \Theta(\log n)$:
  → bonne solution au problème de recherche en table.
- Dans le pire cas $h = n$:
  → importance d'avoir des arbres équilibrés.

- Par rapport aux tables de hachage :
  - pas besoin de connaître n à l'avance,
  - pas de mémoire gaspillée,
  - les ABR équilibrés sont une solution efficace dans le pire cas.

Recherche en table

Récapitulatif

méthode	stockage	recherche	insertion	modif.	suppr.
adressage direct	$\Theta(m)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
recherche séquentielle	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
recherche dichotomique	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
tables de hachage	$\Theta(k+n)$	$\Theta(\frac{n}{k})$	$\Theta(1)$	$\Theta(\frac{n}{k})$	$\Theta(\frac{n}{k})$
avec $k \simeq n$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
ABR	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

- Il faut donc souvent opter pour des tables de hachage :
  ⚠ si les entrées sont très biaisées
  - si n est inconnu/fortement variable
    → utiliser des arbres binaires de recherche (équilibrés).