

Algorithmique — 1. Structures de données

3. Tableaux dynamiques et arbres binaires de recherche

Bruno Grenet



<https://www-verimag.imag.fr/~grenetb/Algorithmique.html>

Université Grenoble Alpes – IM²AG
L3 Mathématiques et Informatique

Table des matières

1. Tableaux dynamiques

2. Arbres binaires de recherche

Table des matières

1. Tableaux dynamiques

2. Arbres binaires de recherche

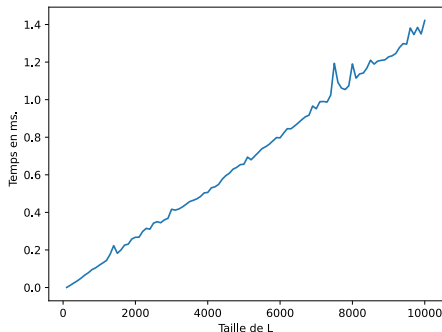
Objectifs

- ▶ Combiner l'accès en temps $O(1)$ des tableaux...
- ▶ ... avec le caractère *dynamique* des listes

▶ Réalisation des TAD Pile, File, File de priorité

Exemples des list Python

```
def test(n):  
    L = []  
    for i in range(n):  
        L.append(i)  
    for i in range(1,n):  
        L[i],L[n-i-1] = L[n-i-1],L[i]
```



Le TAD tableau dynamique

Opérations

- ▶ $\text{NVTABLEAUDYNAMIQUE}(n)$, $\text{TAILLE}(\mathcal{T})$ et $\mathcal{T}[i]$: identiques à un tableau standard
- ▶ $\text{AJOUTER}(\mathcal{T}, x)$: ajoute l'élément x à la fin de $\mathcal{T}$ et met à jour sa taille
- ▶ $\text{SUPPRIMER}(\mathcal{T})$: supprime le dernier élément de $\mathcal{T}$ et met à jour sa taille

Objectifs

- ▶ Complexité *amortie* $O(1)$ pour chaque opération
- ▶ Espace mémoire $O(n)$ où $n = \text{TAILLE}(\mathcal{T})$

Principes de la réalisation

- ▶ Un tableau de taille N pour représenter n éléments $N \geq n$
- ▶ Si le tableau devient trop petit : créer un nouveau tableau plus grand
- ▶ Ne pas gâcher d'espace : si le tableau est trop grand, en créer un nouveau plus petit

Réalisation du TAD tableau dynamique

Représentation

- ▶ Deux données : $\mathcal{T} = (T, n)$ avec
 - ▶ T : tableau (*physique*) T de taille N
 - ▶ n : taille (*utile*) de $\mathcal{T}$
- ▶ Éléments stockés en cases $T_{[0]}, \dots, T_{[n-1]}$
- ▶ Règle à respecter : $n \leq N < 4n$



Opérations identiques à un tableau standard : $O(1)$

- ▶ `NVTABLEAUDYNAMIQUE` : crée T de taille n et renvoie $\mathcal{T} = (T, n)$
- ▶ Accès en lecture/écriture : $\mathcal{T}_{[i]}$ renvoie $T_{[i]}$
- ▶ `TAILLE( $\mathcal{T}$ )` renvoie n

`NVTABLEAUDYNAMIQUE( $n$ )`

1. $T \leftarrow \text{NVTABLEAU}(n)$
2. Renvoyer (T, n)

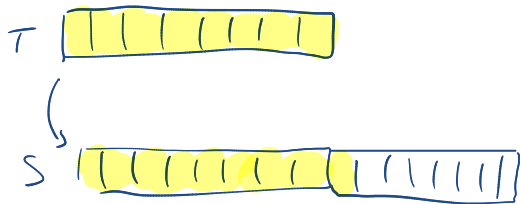
Remarques pour `NVTABLEAUDYNAMIQUE`

- ▶ Pour simplifier, on suppose qu'on crée toujours T de taille $N \geq 1$ (même si $n = 0$)
- ▶ On peut aussi choisir $N = 2n$ initialement

Ajout et suppression

Ajout d'un élément x à la fin

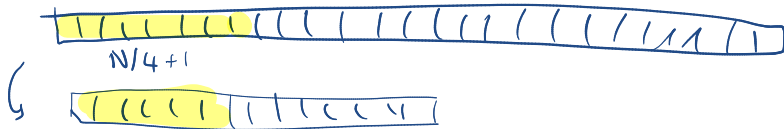
1. Si $N \leq n$, il faut créer de la place :
 - ▶ Créer un nouveau tableau S de taille $2N$
 - ▶ Recopier les N cases de T dans S
 - ▶ Remplacer T par S
2. Ajouter x en case n de T et incrémenter n



Suppression du dernier élément

- ▶ Presque rien à faire : $n \leftarrow n - 1$!
- ▶ Pour garder $N \leq 4n$, il faut (parfois) réduire la taille de T
 - ▶ Idée 1: si $n < N/2$ on réduit de moitié $\rightarrow$ mauvaise idée!
 - ▶ Idée 2: si $n < N/4$ on réduit de moitié $\rightarrow$ bonne idée!

pourquoi ?



Les algorithmes

AJOUTER($\mathcal{T}, x$):

1. $(T, n) \leftarrow \mathcal{T}; N \leftarrow \text{TAILLE}(T)$
2. Si $n < N$:
3. $T[n] \leftarrow x$
4. Renvoyer $(T, n+1)$
5. $S \leftarrow \text{NVTABLEAU}(2N)$
6. Pour $i = 0$ à $N - 1$: $S[i] \leftarrow T[i]$
7. $S[n] \leftarrow x$
8. Renvoyer $(S, n+1)$

SUPPRIMER($\mathcal{T}$):

1. $(T, n) \leftarrow \mathcal{T}; N \leftarrow \text{TAILLE}(T)$
2. Si $n = 1$ ou $n - 1 \geq N/4$:
3. Renvoyer $(T, n-1)$
4. $S \leftarrow \text{NVTABLEAU}(N/2)$
5. Pour $i = 0$ à $n - 2$: $S[i] \leftarrow T[i]$
6. Renvoyer $(S, n-1)$

$$N < 4(n-1)$$

Dans le pire cas, AJOUTER et SUPPRIMER effectuent chacun $O(n)$ affectations

Analyse amortie : cas simplifié

Coût de m ajouts dans un tableau dynamique initialement vide ?

Analyse pire cas

- ▶ Un ajout dans un tableau de taille k effectue $\leq k + 1$ affectations
- ▶ Coût total : $\leq \sum_{k=0}^{m-1} (k+1) = \frac{m(m+1)}{2} = \Theta(m^2)$

Théorème (analyse amortie)

Le nombre total d'affectations pour m ajouts est $< 3m$, donc < 3 par ajout

Preuve

Deux types d'affectations :

- "vraie" affectation ($T[n] \leftarrow x$) : m

- réaffectation $\lfloor \log_2 m \rfloor (S[i] \leftarrow T[i])$: lorsque N est une puissance de 2

$$\sum_{k=0}^{\lfloor \log_2 m \rfloor} 2^k = \frac{2^{\lfloor \log_2 m \rfloor + 1} - 1}{2 - 1} \leq 2m - 1$$

Analyse amortie : cas général avec m opérations AJOUTER/SUPPRIMER

Analyse pire cas

- Une opération sur un tableau de taille k effectuée $\leq k + 1$ affectations $\rightarrow$ coût $O(m^2)$

Comment analyser le coût globalement ?

- Après un redimensionnement, $n \simeq N/2$
 - Avant le prochain agrandissement de T , $\geq n$ ajouts
 - Avant le prochain rétrécissement de T , $\geq n/2$ suppressions
- Le coût des redimensionnements est *amorti* par le nombre d'opérations



Difficulté

- Ajouts et suppressions mélangés
- Coût global de m opérations : borne supérieure sur plein de situations possibles

Une méthode de preuve : la fonction potentiel

Idée : Revenir à une analyse *locale*, opération par opération

Fonction potentiel

- ▶ On attribue un potentiel $\Phi_i \in \mathbb{Z}_{\geq 0}$ à $\mathcal{T}$ après la $i^{\text{ème}}$ opération
- ▶ On définit le coût *amorti* de la $i^{\text{ème}}$ opération par : c_i : vrai coût

$$a_i := c_i + (\Phi_i - \Phi_{i-1})$$

Conséquence

$$\text{coût total} = \sum_{i=1}^m c_i = \sum_{i=1}^m (a_i + \Phi_{i-1} - \Phi_i) = \left(\sum_{i=1}^m a_i \right) + (\Phi_0 - \Phi_m)$$

- ▶ Si on peut montrer que $a_i \leq A$ pour tout i , alors le coût total est $\leq A \times m + \Phi_0$

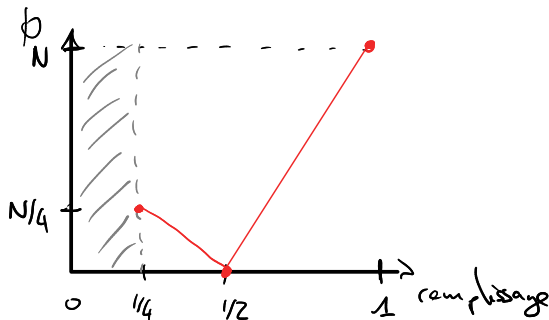
Analyse amortie : potentiel pour les tableaux dynamiques

Fonction potentiel

$$\Phi_i = \begin{cases} 2n_i - N_i & \text{si } n_i \geq N_i/2 \\ N_i/2 - n_i & \text{si } n_i \leq N_i/2 \end{cases}$$

où

- n_i : nombre d'éléments dans $\mathcal{T}$ après la $i^{\text{ème}}$ opération
- N_i : taille de $\mathcal{T}$ après la $i^{\text{ème}}$ opération



Preuve de l'analyse amortie

Lemme

Le coût amorti $a_i = c_i + \Phi_i - \Phi_{i-1}$ de la $i^{\text{ème}}$ opération est ≤ 3 pour tout i

AJOUTER / SUPPRIMER

- Agrandissement :

$$\left. \begin{array}{l} c_i = n+1 \\ \Phi_{i-1} = n \\ \Phi_i = 2 \end{array} \right\} a_i = 3$$

- rétrécissement :

$$\left. \begin{array}{l} c_i = n-1 \\ \Phi_{i-1} = n \\ \Phi_i \approx 0 \end{array} \right\} \leq 0$$

- Pas de rétrécissement :

$$\left. \begin{array}{l} c_i \leq 1 \\ \Phi_i - \Phi_{i-1} \leq 2 \end{array} \right\} a_i \leq 3$$

Preuve de l'analyse amortie

Lemme

Le coût amorti $a_i = c_i + \Phi_i - \Phi_{i-1}$ de la $i^{\text{ème}}$ opération est ≤ 3 pour tout i

Corollaire

Les opérations AJOUTER & SUPPRIMER ont un coût amorti $O(1)$ (≤ 3)

m opérations ont un coût total $\leq \sum_i a_i + \Phi_0 \leq 3m + 0 = 3m$

Donc par opération, le coût est $\leq \frac{3m}{m} = 3$

Bilan sur les tableaux dynamiques

Principes

- ▶ Tableau de taille variable
 - ▶ Mémoire *allouée* supérieure à celle utilisée
 - ▶ Taux de remplissage entre $\frac{1}{4}$ et 1
 - ▶ Taille doublée ou divisée par deux quand nécessaire
- ▶ Accès direct et AJOUTER/SUPPRIMER *en fin de tableau* en temps *amorti* constant

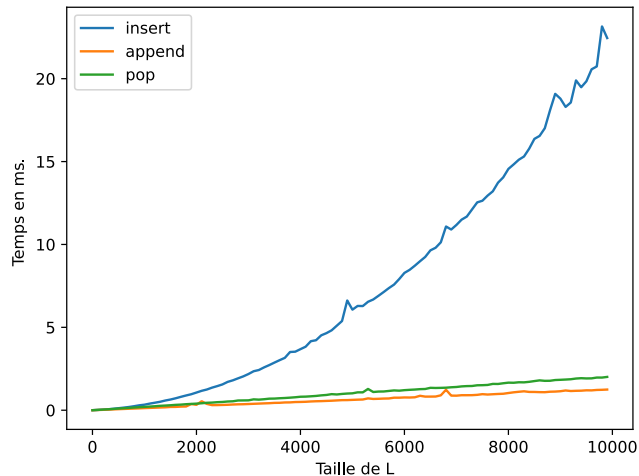
Complexité amortie

- ▶ m opérations coûtent $\leq 3m$ affectations $\rightarrow$ coût *amorti* constant
- ▶ Mais tout de même : si on connaît à l'avance la taille, coût triplé !

Pour réaliser les Pile, File et File de priorité

- ▶ Remplacement du tableau par un tableau dynamique
- ▶ Plus de souci de taille maximale fixée
- ▶ Complexités *amorties* aussi bonnes

Performance des list Python



- Insertion en début de tableau
- Insertion en fin de tableau
- Suppression en fin de tableau

Table des matières

1. Tableaux dynamiques

2. Arbres binaires de recherche

Dictionnaires

(ou tableau associatif, ou table de symboles)

Définition informelle

- ▶ Structure dynamique pour représenter un ensemble de couples (clé, valeur)
- ▶ Une clé ne peut être présente qu'une seule fois
- ▶ Accès rapide à la valeur associée à une clé

Exemples

- ▶ *dictionaries* en Python, .NET, Swift, C#, ...
- ▶ *maps* en C++, Java, OCaml, Haskell, ...
- ▶ *tables* en Lua, Maple, ...
- ▶ *arrays* en PHP

Quelques utilisations

- ▶ (vrai) dictionnaire, annuaire, ...
- ▶ types et valeurs des variables pour un compilateur
- ▶ base de données type NoSQL

Le TAD Dictionnaire

Opérations

- ▶ $\text{NVDICTIONNAIRE}()$: crée un dictionnaire vide
- ▶ $\text{INSÉRER}(D, c, v)$: associe la valeur v à la clé c (avec écrasement) $D[c] \leftarrow v$
- ▶ $\text{SUPPRIMER}(D, c)$: supprime c de D si $c \in D$; *erreur* sinon
- ▶ $\text{RECHERCHER}(D, c)$: renvoie la valeur associée à c si $c \in D$; *erreur* sinon $v \leftarrow D[c]$

Remarque

- ▶ On peut ajouter facilement un test d'appartenance : est-ce que $c \in D$?

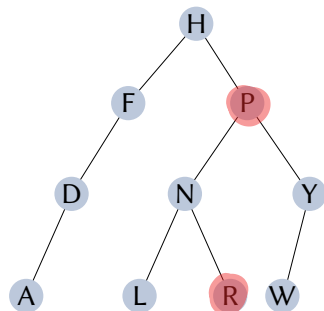
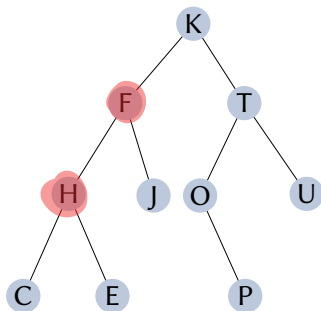
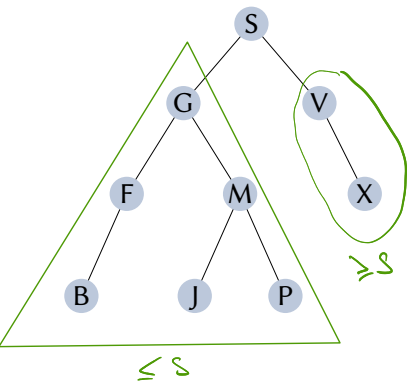
Principales réalisations

- ▶ Arbres binaires de recherche et extensions *ce cours*
- ▶ Tables de hachage *cours 4*

Définition

Un **arbre binaire de recherche** (ABR) est un arbre binaire dont chaque nœud possède une clé, qui est

- ▶ supérieure aux clés des nœuds de son sous-arbre gauche
- ▶ inférieure aux clés des nœuds de son sous-arbre droit



Réalisation du TAD Dictionnaire avec un ABR

Représentation des données

- ▶ Un arbre binaire de recherche $\mathcal{A}$
- ▶ Chaque nœud possède aussi une valeur
 - ▶ Un nœud est un couple (clé, valeur)
 - ▶ Structure d'ABR sur les clés
- ▶ $\text{NVDICTIONNAIRE}() \rightsquigarrow \text{NVARBRE}()$

À résoudre

- ▶ RECHERCHER une clé donnée
- ▶ INSÉRER une nouvelle clé
- ▶ SUPPRIMER une clé

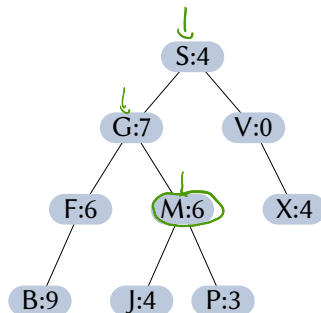
Remarque

- ▶ On manipule les nœuds *via* leurs clés $\rightsquigarrow$ « clé de $\text{RACINE}(\mathcal{A})$ »

Recherche dans un ABR

RECHERCHER($\mathcal{A}, c$):

1. Si ESTVIDE($\mathcal{A}$) : renvoyer une erreur (*non trouvé*)
2. $c' \leftarrow$ clé de RACINE($\mathcal{A}$)
3. Si $c = c'$: renvoyer la valeur de RACINE($\mathcal{A}$)
4. $(\mathcal{G}, \mathcal{D}) \leftarrow (\text{ENFANTG}(\mathcal{A}), \text{ENFANTD}(\mathcal{A}))$
5. Si $c < c'$: renvoyer RECHERCHER($\mathcal{G}, c$)
6. renvoyer RECHERCHER($\mathcal{D}, c$)



Théorème

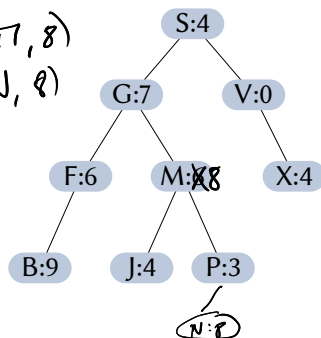
RECHERCHER($\mathcal{A}, c$) est correct et a une complexité $O(h_{\mathcal{A}})$ où $h_{\mathcal{A}}$ = hauteur de $\mathcal{A}$.

Insertion d'un élément

INSÉRER($\mathcal{A}$, c , v):

1. Si ESTVIDE($\mathcal{A}$):
2. $\mathcal{A} \leftarrow \text{NVARBRE}((c, v), \emptyset, \emptyset)$
3. Sinon :
4. $c' \leftarrow \text{clé de RACINE}(\mathcal{A})$
5. $(\mathcal{G}, \mathcal{D}) \leftarrow (\text{ENFANTG}(\mathcal{A}), \text{ENFANTD}(\mathcal{A}))$
6. Si $c' = c$: $\mathcal{A} \leftarrow \text{NVARBRE}((c, v), \mathcal{G}, \mathcal{D})$
7. Sinon si $c < c'$: INSÉRER($\mathcal{G}$, c , v)
8. Sinon : INSÉRER($\mathcal{D}$, c , v)

INSÉRER($A_0, 17, 8$)
— ($A_0, N, 8$)



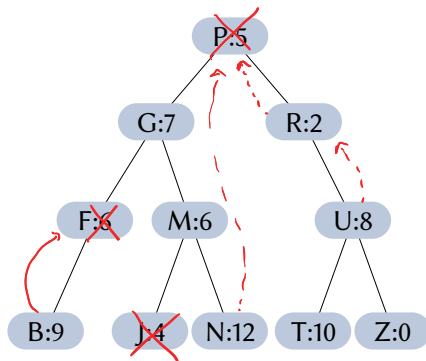
Théorème

INSÉRER a une complexité $O(h_{\mathcal{A}})$ et conserve la structure d'ABR

Suppression d'un élément

Exemple

1. Supprimer *J*
2. Supprimer *F*
3. Supprimer *P*



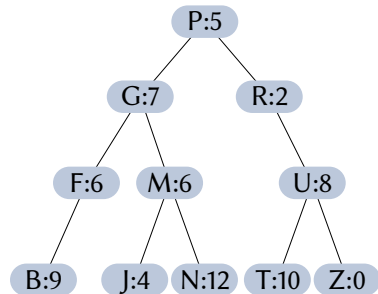
Algorithme pour supprimer le nœud *N* de clé *c*

1. Si *N* n'a pas d'enfant : le supprimer
2. Si *N* a un seul enfant : le remplacer par son unique enfant
3. Sinon : remplacer *N* par le minimum de son sous-arbre droit

Trouver et supprimer le minimum

SUPPRIMERMIN($\mathcal{A}$):

1. Si $\text{ENFANTG}(\mathcal{A}) = \emptyset$:
2. $(c, v) \leftarrow \text{RACINE}(\mathcal{A})$
3. $\mathcal{A} \leftarrow \text{ENFANTD}(\mathcal{A})$
4. Renvoyer (c, v)
5. Renvoyer SUPPRIMERMIN($\text{ENFANTG}(\mathcal{A})$)



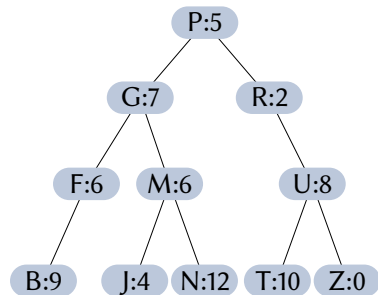
Lemme

SUPPRIMERMIN a une complexité $O(h_{\mathcal{A}})$ et est correct

Algorithme de suppression

SUPPRIMER($\mathcal{A}, c$):

1. Si ESTVIDE($\mathcal{A}$): renvoyer une erreur (*non trouvé*)
2. $c' \leftarrow$ clé de RACINE($\mathcal{A}$)
3. $(\mathcal{G}, \mathcal{D}) \leftarrow (\text{ENFANTG}(\mathcal{A}), \text{ENFANTD}(\mathcal{A}))$
4. Si $c = c'$:
 5. Si $\mathcal{G} = \emptyset$: $\mathcal{A} \leftarrow \mathcal{D}$
 6. Sinon si $\mathcal{D} = \emptyset$: $\mathcal{A} \leftarrow \mathcal{G}$
 7. Sinon :
 8. $(c_m, v_m) \leftarrow \text{SUPPRIMERMIN}(\mathcal{D})$
 9. $\mathcal{A} \leftarrow \text{NVARBRE}((c_m, v_m), \mathcal{G}, \mathcal{D})$
10. Sinon si $c < c'$: SUPPRIMER($\mathcal{G}, c$)
11. Sinon : SUPPRIMER($\mathcal{D}, c$)



Théorème

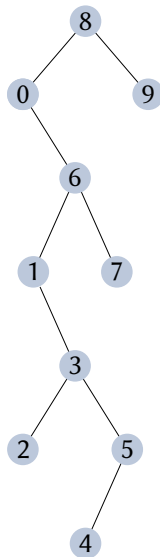
SUPPRIMER a une complexité $O(h_{\mathcal{A}})$, et conserve la structure d'ABR de $\mathcal{A}$

Quelle efficacité alors ?

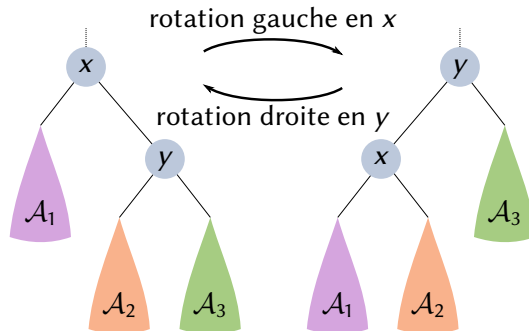
Rappel des complexités

- RECHERCHER, INSÉRER et SUPPRIMER : $O(h_{\mathcal{A}})$
- Dans un arbre binaire, $\lfloor \log \# \mathcal{A} \rfloor \leq h_{\mathcal{A}} \leq \# \mathcal{A}$

Un ABR est une structure de donnée efficace s'il est **équilibré**, c'est-à-dire si $h_{\mathcal{A}} = O(\log(\# \mathcal{A}))$.



Outil de base : les rotations



Théorème

Les rotations s'effectuent en temps $O(1)$ et conservent la structure d'ABR

Utilisation des rotations

- ▶ Augmentation de la hauteur d'un côté, diminution de l'autre $\rightsquigarrow$ meilleur équilibre
- ▶ Nombreuses techniques d'équilibrage : arbre rouge-noir, AVL, B, déployé, tarbre, ...
 - ▶ hors programme de ce cours

Bilan sur les ABR

- ▶ **Une** réalisation possible du TAD Dictionnaire
- ▶ INSÉRER/SUPPRIMER, RECHERCHER, ... : $O(h_{\mathcal{A}})$
 - ▶ Efficace uniquement si $h_{\mathcal{A}} = O(\log(\#\mathcal{A}))$
 - ▶ Vrai si insertion en ordre aléatoire
 - ▶ Techniques d'équilibrage basées sur les rotations

TAD Dictionnaire *ordonné*

- ▶ Mêmes opérations que Dictionnaire, plus
 - ▶ $\text{MIN}(D)$, $\text{MAX}(D)$: clés minimale et maximale dans D
 - ▶ $\text{SUIVANT}(D, c)$: plus petite clé $> c$ dans D
 - ▶ $\text{PRÉCÉDENT}(D, c)$: plus grande clé $< c$ dans D
- ▶ Les ABR réalisent le TAD Dictionnaire ordonné

opérations en $O(h_{\mathcal{A}})$

Autres utilisations des ABR

- ▶ Tri $\rightarrow$ parcours infixe
- ▶ Files de priorité
- ▶ TAD Ensemble ou Ensemble ordonné

clés uniquement