

Algorithmique et Analyse d'Algorithmes

L3 Info

Cours 7 : Structures de données dynamiques

Benjamin Wack



2025 - 2026

La dernière fois

- ▶ Preuve d'algorithmes récursifs
- ▶ Dichotomie
- ▶ Logique de Hoare

Aujourd'hui

- ▶ Structure dynamique
- ▶ Arbre Binaire de Recherche (ABR)
- ▶ Coûts des opérateurs
- ▶ ABR améliorés

Plan

Notion de structure dynamique

Arbre Binaire de Recherche

Algorithmes pour les opérations des ABR

Analyses de coût

Garantir des coûts logarithmiques

Motivation

Comportement **dynamique**

On cherche à construire une structure de données qui permet de :

- ▶ ajouter de nouveaux éléments
- ▶ retirer des éléments existants
- ▶ éventuellement mettre à jour un élément

De plus cette structure doit garantir un certain nombre de propriétés :

- ▶ qualitatives (par exemple : données triées, sans doublon...)
- ▶ quantitatives (contraintes de complexité)

Exemples

- ▶ Annuaire : recherche rapide, éventuellement par numéro
- ▶ Réseau social : facilité d'exploration des liens successifs

Méthodologie

Construction de la structure dynamique

- ▶ On s'appuie sur un Type Abstrait de Données connu.
- ▶ On définit de nouvelles fonctions qui garantissent les propriétés voulues.
- ▶ On s'interdit ensuite d'utiliser directement (une partie) des méthodes du type abstrait de départ.

Évaluation des coûts

- ▶ La « forme » de la structure dépend des opérations subies dynamiquement.
- ▶ Donc le coût des opérations aussi.

Deux possibilités :

- ▶ déterminer les caractéristiques **moyennes** d'une structure et évaluer le coût d'une opération sur cette base
- ▶ OU effectuer une analyse de coût **amorti** (non traité ici)

Caractéristiques recherchées

- ▶ La structure dynamique voulue doit permettre d'accéder à tout élément rapidement.
- ▶ On dispose d'un ordre total sur le type *Element*.

Représentation	Recherche	Insertion	Suppression
Tableau non ordonné	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Tableau ordonné	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Liste chaînée non ordonnée	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Liste chaînée ordonnée	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Seul espoir de recherche rapide : maintenir un ensemble **trié**.

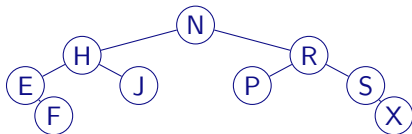
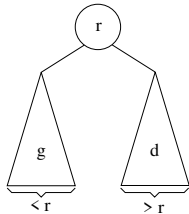
Mais comment assurer également une insertion et une suppression efficaces ?

- ▶ Utilisation d'une structure arborescente pour accéder (et modifier) rapidement tout point de la structure

Le principe des Arbres Binaires de Recherche

- ▶ La clé de la racine est supérieure à celle de **tous** les nœuds du fils gauche.
- ▶ La clé de la racine est inférieure à celle de **tous** les nœuds du fils droit.

⇒ permet une recherche de type dichotomique : comparer e à la racine suffit à déterminer dans quel sous-arbre il se trouve.



Enfin ce principe s'applique récursivement : le fils gauche et le fils droit sont tous deux des Arbres Binaires de Recherche.

Le type Arbre Binaire de Recherche

Type Abstrait de base Arbre binaire

Adaptation de l'interface

ArbreVide	:	$void \rightarrow ABR$	} immédiat
Nœud	:	$ABR \times Element \times ABR \rightarrow ABR$	
			peut casser les propriétés
EstArbreVide	:	$ABR \rightarrow bool$	} sans danger
Clé	:	$ABR \rightarrow Element$	
FilsGauche	:	$ABR \rightarrow ABR$	} par construction
FilsDroit	:	$ABR \rightarrow ABR$	

Propriétés souhaitées

$$\forall y \in \text{FilsGauche}(a), \text{Clé}(y) \leq \text{Clé}(a)$$

$$\forall z \in \text{FilsDroit}(a), \text{Clé}(a) \leq \text{Clé}(z)$$

Si a est un ABR, FilsGauche(a) est un ABR

Si a est un ABR, FilsDroit(a) est un ABR

⇒ Il nous faut une opération supplémentaire pour insérer une valeur dans un ABR

Opérations supplémentaires

Opérations

Insérer	:	$Element \times ABR \rightarrow ABR$	} structure dynamique
Supprimer	:	$Element \times ABR \rightarrow ABR$	
Rechercher	:	$Element \times ABR \rightarrow ABR$	} exploite la structure
Minimum	:	$ABR \rightarrow ABR$	

Préconditions Pour toutes les opérations : l'argument est un ABR

Minimum(a) : *non* EstArbreVide(a)

Supprimer(e, a) : *non* EstVide(Rechercher(e, a))

Axiomes Pour toutes les opérations : l'arbre renvoyé est un ABR

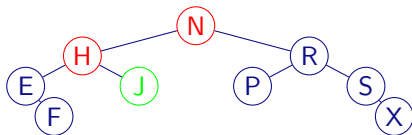
Rechercher(e, a) = un sous-arbre $N(g, r, d)$ avec $e = r$
s'il existe

Rechercher(e, a) = ArbreVide() sinon

⋮

Reste à **coder ces opérations** à partir des opérations de base sur les arbres
puis on **n'utilisera plus** le constructeur Noeud (encapsulation).

Rechercher (J)



RECHERCHER(e, a)

Données : Une clé e , un ABR a

Résultat : Si possible un nœud de a dont la clé est e , sinon *ArbreVide()*

si *EstVide(a)*

└ renvoyer *ArbreVide()*

sinon si $e < \text{Clé}(a)$

└ renvoyer *RECHERCHER(e, FilsGauche(a))*

sinon si $e > \text{Clé}(a)$

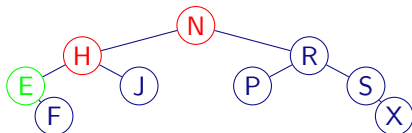
└ renvoyer *RECHERCHER(e, FilsDroit(a))*

sinon // $e = \text{Clé}(a)$

└ renvoyer a

Variante : renvoyer une information associée à la clé plutôt que le nœud

Minimum



MINIMUM(a)

Données : Un ABR a non vide

Résultat : Un nœud de a dont la clé est inférieure à toutes les autres clés de a

si *EstVide*(FilsGauche(a))

└ renvoyer a

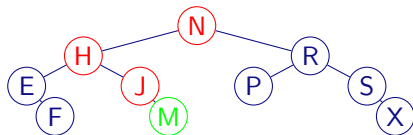
sinon

└ renvoyer MINIMUM(FilsGauche(a))

Remarque

Cet algorithme et le précédent s'écrivent aussi bien sous forme impérative sans pile.

Insérer (M)



INSERER(e, a)

Données : Une clé e , un ABR a

Résultat : Un ABR dont les clés sont : e et les clés de a

si $EstVide(a)$

└ renvoyer $N(ArbreVide(), e, ArbreVide())$

sinon si $e \leq Clé(a)$

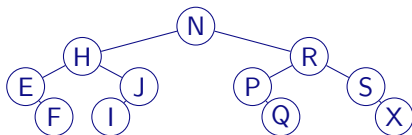
└ renvoyer $N(INSERER(e, FilsGauche(a)), Clé(a), FilsDroit(a))$

sinon // $e > Clé(a)$

└ renvoyer $N(FilsGauche(a), Clé(a), INSERER(e, FilsDroit(a)))$

Variante : chaque clé est unique, si $e = Clé(a)$ on met à jour le nœud.

Supprimer



SUPPRIMER(e, a)

Données : Un élément e présent dans a , un ABR a

Résultat : Un ABR contenant les mêmes clés que a sauf e

... comme RECHERCHER mais...

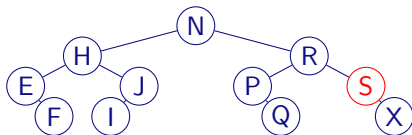
⋮

si $e = \text{Clé}(a)$

 └ renvoyer SUPPRIMER_RACINE(a)

⋮

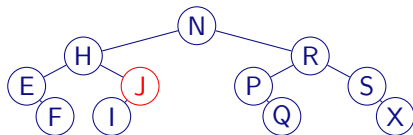
Supprimer (2)



SUPPRIMER_RACINE(x)

si *EstVide(FilsGauche(x))* renvoyer *FilsDroit(x)*

Supprimer (2)

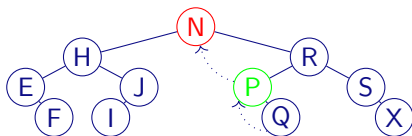


SUPPRIMER_RACINE(x)

si *EstVide(FilsGauche(x))* **renvoyer** *FilsDroit(x)*

sinon si *EstVide(FilsDroit(x))* **renvoyer** *FilsGauche(x)*

Supprimer (2)



SUPPRIMER_RACINE(x)

si *EstVide*(FilsGauche(x)) **renvoyer** FilsDroit(x)

sinon si *EstVide*(FilsDroit(x)) **renvoyer** FilsGauche(x)

sinon

$m := \text{MINIMUM}(\text{FilsDroit}(x))$

renvoyer $N(\text{FilsGauche}(x), m, \text{SUPPR_MIN}(\text{FilsDroit}(x)))$

SUPPR_MIN(x)

... (presque) comme MINIMUM mais ...

⋮

si *EstVide*(FilsGauche(x)) **renvoyer** FilsDroit(x)

⋮

Premier constat

Il est facile de voir que toutes les opérations sont en $\mathcal{O}(h)$ où h est la hauteur de l'arbre parcouru.

On s'intéresse donc à la hauteur d'un ABR de n nœuds **construit par n insertions successives** dans l'arbre vide.

Pire cas

- ▶ Au pire $h = n$ (hauteur maximale d'un arbre à n nœuds).
- ▶ Concrètement, ce pire cas est atteint quand les données sont insérées par croissant (ou décroissant).

Meilleur cas

- ▶ Au mieux $h = \lfloor \log_2 n \rfloor$ (hauteur minimale d'un arbre à n nœuds).
- ▶ Concrètement, ce meilleur cas est atteint quand... ?

Arbre moyen

On cherche à déterminer la structure moyenne d'un ABR de n nœuds **construit par n insertions successives** dans l'arbre vide.

Hypothèse de répartition des données : les $n!$ permutations des n clés introduites sont équiprobables.

Exemple : écrire toutes les permutations de A,B,C,D.

Construire les ABR correspondant à 3 ordres d'insertion parmi les 24.

La racine de l'ABR :

- ▶ est toujours la première clé insérée
- ▶ peut être chaque clé de façon équiprobable

Si la racine est la i^{e} clé, alors :

- ▶ le sous-arbre gauche contient $i - 1$ nœuds
- ▶ le sous-arbre droit contient $n - i$ nœuds

Profondeur totale des nœuds dans un ABR

Somme des profondeurs

On note $P(a)$ la somme des profondeurs de tous les nœuds de a .

$$P_{totale}(a) = \sum_{x \in a} dist(x, racine(a))$$

$$\begin{aligned} P_{totale}(a) &= \sum_{x \in g} dist(x, racine(a)) + \sum_{x \in d} dist(x, racine(a)) + 0 \\ &= \sum_{x \in g} (dist(x, racine(g)) + 1) + \sum_{x \in d} (dist(x, racine(d)) + 1) \\ &= n - 1 + \sum_{x \in g} dist(x, racine(g)) + \sum_{x \in d} dist(x, racine(d)) \end{aligned}$$

$$P_{totale}(a) = n - 1 + P_{totale}(g) + P_{totale}(d)$$

Profondeur moyenne des nœuds dans un ABR

On s'intéresse à P_n la valeur moyenne de $P_{totale}(a)$ pour **tous** les arbres a de n nœuds :

- ▶ la même équation de récurrence s'applique
- ▶ on a supposé que les n cas de la forme « $i - 1$ nœuds dans g , $n - i$ nœuds dans d » sont équiprobables.

On en déduit que :

$$\begin{aligned}P_n &= \frac{1}{n} \sum_{i=1}^n (P_{i-1} + P_{n-i} + n - 1) \\&= \frac{1}{n} \sum_{i=1}^n P_{i-1} + \frac{1}{n} \sum_{i=1}^n P_{n-i} + \frac{1}{n} \sum_{i=1}^n n - 1 \\P_n &= \frac{2}{n} \sum_{i=0}^{n-1} P_i + n - 1 \\n \times P_n &= 2 \sum_{i=0}^{n-1} P_i + n(n - 1)\end{aligned}$$

Profondeur moyenne des nœuds dans un ABR (2)

En éliminant la somme :

$$nP_n - (n-1)P_{n-1} = 2P_{n-1} + n(n-1) - (n-1)(n-2)$$

$$\frac{P_n}{n+1} = \frac{P_{n-1}}{n} + 2\frac{n-1}{n(n+1)}$$

$$\frac{P_n}{n+1} = \frac{P_{n-1}}{n} + \frac{4}{n+1} - \frac{2}{n}$$

$$\frac{P_n}{n+1} = \sum_{i=1}^n \left(\frac{4}{i+1} - \frac{2}{i} \right)$$

$$\frac{P_n}{n+1} = 2 \sum_{i=1}^n \frac{1}{i} + \frac{2}{n+1} - 4$$

$$\frac{P_n}{n+1} = 2 \log n + \text{un terme négligeable devant } \log n$$

D'où

$$P_n = \mathcal{O}(n \log n)$$

et

la profondeur moyenne d'un nœud est $\mathcal{O}(\log n)$.

Conséquences

- ▶ Le coût de la recherche d'une clé existant dans l'arbre est exactement sa profondeur :
Le coût moyen de la recherche est $\mathcal{O}(\log n)$.
- ▶ En l'absence de suppressions, le coût d'insertion de chaque nœud est exactement sa profondeur :
 - ▶ INSERTION suit le même chemin que RECHERCHE ;
 - ▶ les nœuds ne sont pas déplacés par les insertions suivantes.Le coût total moyen de l'insertion de n nœuds est $\mathcal{O}(n \log n)$.

Admis : la hauteur moyenne d'un ABR obtenu par insertions successives de n nœuds en ordre aléatoire est $\mathcal{O}(\log n)$.

Une application « évidente »

L'idée initiale de l'ABR était de maintenir une structure **triée** pour accélérer la recherche et l'accès **d'un élément** : comment récupérer **l'ensemble** trié ?

```
PARCOURS_PROFONDEUR_INFIXE(a)
```

```
si non EstArbreVide(a)  
┌ PARCOURS_PROFONDEUR_INFIXE ( FilsGauche(a) )  
  Récupérer( Clé(a) )  
└ PARCOURS_PROFONDEUR_INFIXE ( FilsDroit(a) )
```

D'où l'algorithme de tri par ABR :

```
tant que il reste des données  
┌ Insérer une donnée dans l'ABR  
Parcourir l'ABR en profondeur infixe
```

Complexité du tri par ABR

- ▶ Coût des n insertions successives :
 - ▶ Au pire : $\mathcal{O}(n^2)$
 - ▶ Au mieux **et en moyenne** : $\mathcal{O}(n \log n)$
- ▶ Coût du parcours en profondeur infixe : $\mathcal{O}(n)$

Lien avec le tri rapide

- ▶ Le premier élément e_1 inséré dans l'arbre est mis à sa place définitive.
- ▶ Tous les éléments $< e_1$ sont insérés à gauche.
- ▶ Tous les éléments $> e_1$ sont insérés à droite. } *segmentation*
- ▶ La construction des sous-arbres suit récursivement ce processus.

(En revanche la segmentation du tri rapide peut modifier l'ordre d'insertion.)

La complexité du tri rapide est au pire en $\mathcal{O}(n^2)$, en moyenne en $\mathcal{O}(n \log n)$.

Marge de progression

Les coûts moyens (et au mieux) de la recherche et de l'insertion sont logarithmiques, mais le cas pire est linéaire : peut-on l'éviter ?

Il faut (et il suffit) que $h = \mathcal{O}(\log n)$.

Origine des pires coûts

Déséquilibres de l'arbre, introduits par :

- ▶ insertion dans une branche déjà chargée
- ▶ suppression dans une branche déjà légère
OU réorganisation malencontreuse suite à la suppression

Deux pistes de solutions

Améliorer les opérations

pour rééquilibrer l'arbre au fur et à mesure

- ▶ en se basant sur quelle information ? il faut la stocker aussi...
- ▶ attention à ne pas payer un coût supplémentaire !

Arbres AVL (Adelson-Velsky et Landis) par exemple

Améliorer la structure

pour qu'il y ait « toujours de la place »

- ▶ la hauteur ne change pas, ou alors par la racine \Rightarrow équilibre parfait
- ▶ mais on perd l'aspect binaire

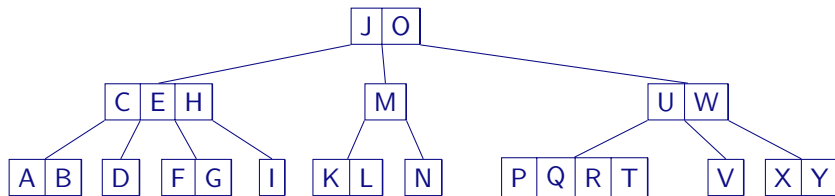
Les B-arbres

Structure

Similaire à un ABR mais :

- ▶ chaque nœud peut porter i fils f_1, \dots, f_i avec $t \leq i \leq 2t$
- ▶ ces i fils sont séparés par $i - 1$ clés $k_1 < k_2 < \dots < k_{i-1}$
- ▶ telles que la clé k_j est supérieure à toutes celles du fils f_j et inférieure à toutes celles du fils f_{j+1}
- ▶ tous les sous-arbres ont la même hauteur

Exemple :



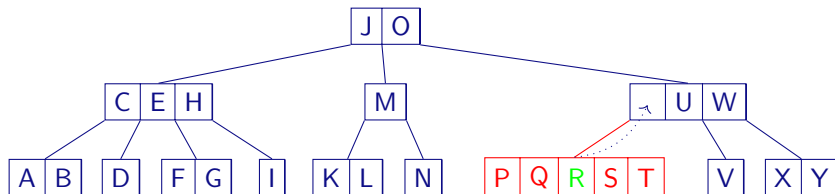
Les B-arbres

Structure

Similaire à un ABR mais :

- ▶ chaque nœud peut porter i fils f_1, \dots, f_i avec $t \leq i \leq 2t$
- ▶ ces i fils sont séparés par $i - 1$ clés $k_1 < k_2 < \dots < k_{i-1}$
- ▶ telles que la clé k_j est supérieure à toutes celles du fils f_j et inférieure à toutes celles du fils f_{j+1}
- ▶ tous les sous-arbres ont la même hauteur

Exemple :



Insertion

1. On insère la nouvelle clé dans une feuille, comme dans un ABR
 $\mathcal{O}(h \times t)$
2. Si après cette opération le nœud est plein ($i = 2t + 1$), on le coupe en deux : $\mathcal{O}(t)$
 - ▶ on obtient deux nœuds de t éléments chacun ;
 - ▶ la clé médiane est insérée dans le nœud père, où elle sert à « séparer » ces deux nœuds.
3. Si le nœud père est lui-même plein, on coupe à nouveau, autant de fois qu'il le faut en remontant vers la racine. $\mathcal{O}(h \times t)$
4. La hauteur augmente quand on arrive à une racine pleine : l'arbre « pousse en largeur d'abord, puis par la racine si nécessaire ».

En résumé

Aujourd'hui

- ▶ Une **structure dynamique** est un TAD qui **maintient des propriétés**
- ▶ Un **Arbre Binaire de Recherche** permet de maintenir un **ensemble trié** à moindre coût
- ▶ Les coûts de recherche, insertion et suppression restent **linéaires au pire**, mais ils sont **logarithmiques en moyenne**
- ▶ On peut **garantir un coût logarithmique** en **enrichissant** ou en **généralisant** la structure

La prochaine fois

- ▶ Arbre partiellement ordonné
- ▶ Structure de tas
- ▶ Application à la FàP