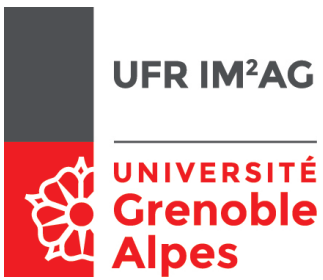
Algorithmique et Analyse d'Algorithmes

L3 Info

Cours 7 : Structures de données dynamiques

Benjamin Wack



2020- 2021

La dernière fois

- Structures arborescentes
- Type Abstrait Arbre Binaires (et autres)
- Partition Binaire de l'Espace

Aujourd'hui

- Structure dynamique
- Arbre Binaire de Recherche (ABR)
- Opérateurs de base
- Coûts des opérateurs

Plan

Plan

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

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)

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

Définition

- On reprend la méthodologie TAD.
- Par commodité, les fonctions d'insertion, suppression, mise à jour sont souvent des fonctions à effet de bord.
- Les propriétés quantitatives à vérifier sont ajoutées aux axiomes.

Méthodologie

Définition

- On reprend la méthodologie TAD.
- Par commodité, les fonctions d'insertion, suppression, mise à jour sont souvent des fonctions à effet de bord.
- Les propriétés quantitatives à vérifier sont ajoutées aux axiomes.

Évaluation des coûts

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

Méthodologie

Définition

- On reprend la méthodologie TAD.
- Par commodité, les fonctions d'insertion, suppression, mise à jour sont souvent des fonctions à effet de bord.
- Les propriétés quantitatives à vérifier sont ajoutées aux axiomes.

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)

Plan

Caractéristiques recherchées

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

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	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

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

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	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

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**.

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

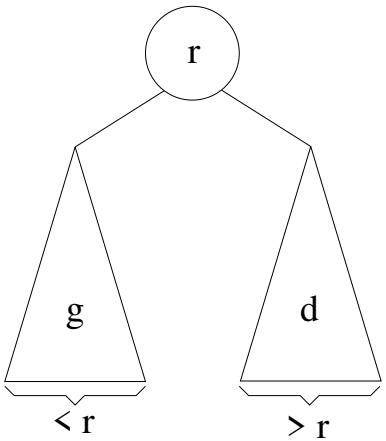
Le principe des Arbres Binaires de Recherche

- 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.

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.

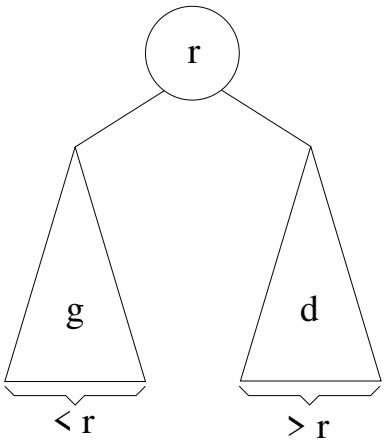
$\Rightarrow$ permet une recherche de type dichotomique : comparer e à la racine suffit à déterminer dans quel sous-arbre il se trouve.



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.

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

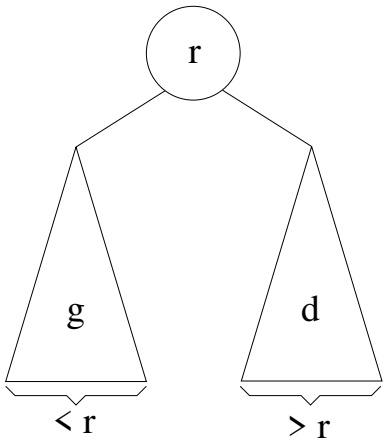


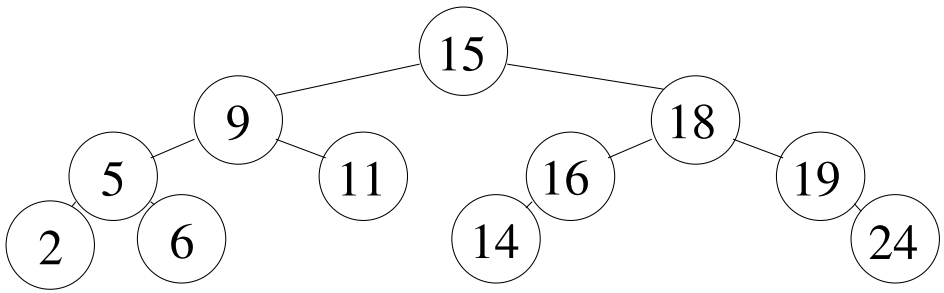
Enfin ce principe s'applique récursivement : le fils gauche et le fils droit sont tous deux des Arbres Binaires de Recherche.

Le 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.

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



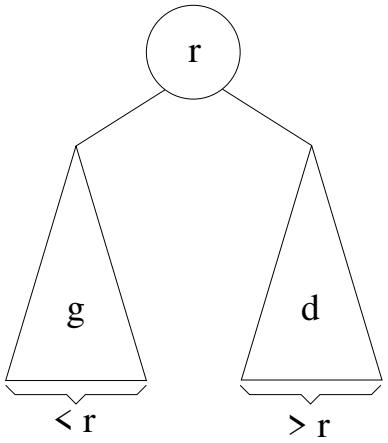


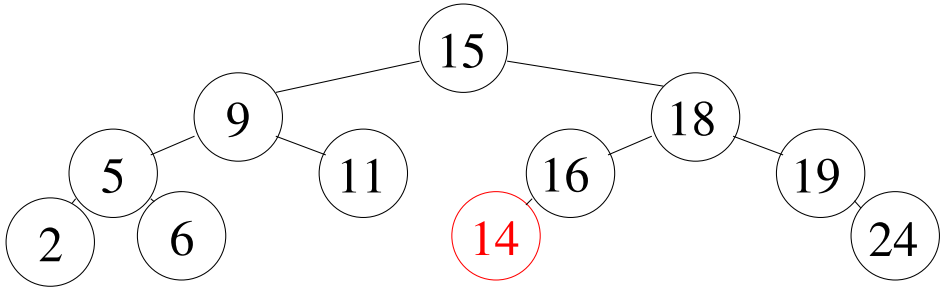
Enfin ce principe s'applique récursivement : le fils gauche et le fils droit sont tous deux des Arbres Binaires de Recherche.

Le 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.

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



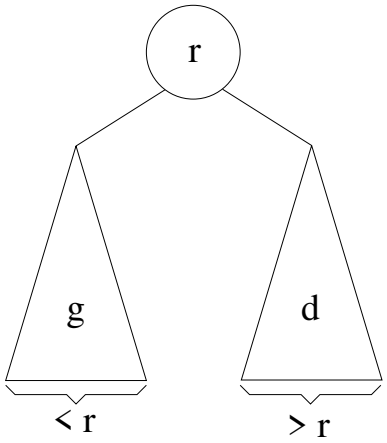


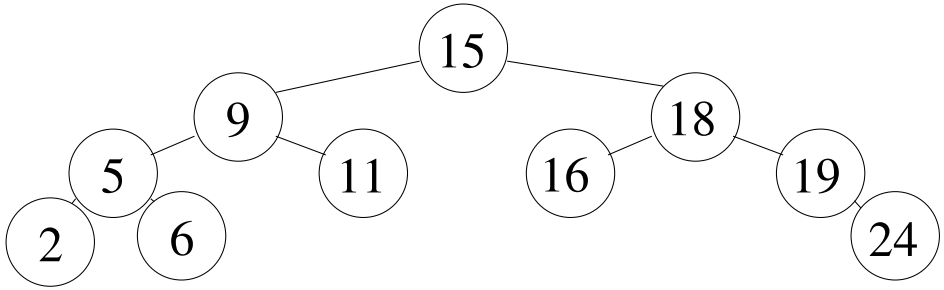
Enfin ce principe s'applique récursivement : le fils gauche et le fils droit sont tous deux des Arbres Binaires de Recherche.

Le 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.

$\Rightarrow$ 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 abstrait Arbre Binaire de Recherche

Préconditions Idem Arbre binaire

Axiomes Idem Arbre binaire +

Le type abstrait Arbre Binaire de Recherche

Préconditions Idem Arbre binaire

Axiomes Idem Arbre binaire +

$$\begin{aligned} \forall x \in a : \quad \forall y \in \text{FilsGauche}(x),\ \text{Clé}(y) &\leq \text{Clé}(x) \\ \forall z \in \text{FilsDroit}(x),\ \text{Clé}(x) &\leq \text{Clé}(z) \end{aligned}$$

Le type abstrait Arbre Binaire de Recherche

Préconditions Idem Arbre binaire

Axiomes Idem Arbre binaire +

$$\begin{aligned} \forall x \in a : \quad \forall y \in \text{FilsGauche}(x),\ \text{Clé}(y) &\leq \text{Clé}(x) \\ \forall z \in \text{FilsDroit}(x),\ \text{Clé}(x) &\leq \text{Clé}(z) \end{aligned}$$

Ou de façon équivalente :

$$\begin{aligned} \forall y \in \text{FilsGauche}(a),\ \text{Clé}(y) &\leq \text{Clé}(a) \\ \forall z \in \text{FilsDroit}(a),\ \text{Clé}(a) &\leq \text{Clé}(z) \\ \text{FilsGauche}(a) &\text{ est un ABR} \\ \text{FilsDroit}(a) &\text{ est un ABR} \end{aligned}$$

Le type abstrait Arbre Binaire de Recherche

Préconditions Idem Arbre binaire

Axiomes Idem Arbre binaire +

$$\begin{aligned} \forall x \in a : \quad \forall y \in \text{FilsGauche}(x),\ \text{Clé}(y) &\leq \text{Clé}(x) \\ \forall z \in \text{FilsDroit}(x),\ \text{Clé}(x) &\leq \text{Clé}(z) \end{aligned}$$

Ou de façon équivalente :

$$\begin{aligned} \forall y \in \text{FilsGauche}(a),\ \text{Clé}(y) &\leq \text{Clé}(a) \\ \forall z \in \text{FilsDroit}(a),\ \text{Clé}(a) &\leq \text{Clé}(z) \\ \text{FilsGauche}(a) &\text{ est un ABR} \\ \text{FilsDroit}(a) &\text{ est un ABR} \end{aligned}$$

$\forall x \in a$, $\text{Clé}(\text{FilsGauche}(x)) \leq \text{Clé}(x)$ ne suffit pas.

Opérations supplémentaires : modificateurs

Opérations

Opérations supplémentaires : modificateurs

Opérations	Rechercher	:	$Element \times ABR \rightarrow ABR$
	Minimum	:	$ABR \rightarrow ABR$
	Insérer	:	$Element \times ABR \rightarrow ABR$
	Supprimer	:	$Element \times ABR \rightarrow ABR$

Opérations supplémentaires : modificateurs

Opérations

$$\begin{aligned} \text{Rechercher} &: \mathit{Element} \times \mathit{ABR} \rightarrow \mathit{ABR} \\ \text{Minimum} &: \mathit{ABR} \rightarrow \mathit{ABR} \\ \text{Insérer} &: \mathit{Element} \times \mathit{ABR} \rightarrow \mathit{ABR} \\ \text{Supprimer} &: \mathit{Element} \times \mathit{ABR} \rightarrow \mathit{ABR} \end{aligned}$$

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

$$\begin{aligned} \text{Minimum}(a) &: \mathit{non}\ \text{EstArbreVide}(a) \\ \text{Supprimer}(e, a) &: \mathit{non}\ \text{EstVide}(\text{Rechercher}(e, a)) \end{aligned}$$

Opérations supplémentaires : modificateurs

Opérations

Rechercher : $Element \times ABR \rightarrow ABR$
Minimum : $ABR \rightarrow ABR$

Insérer : $Element \times ABR \rightarrow ABR$
Supprimer : $Element \times ABR \rightarrow ABR$

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

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

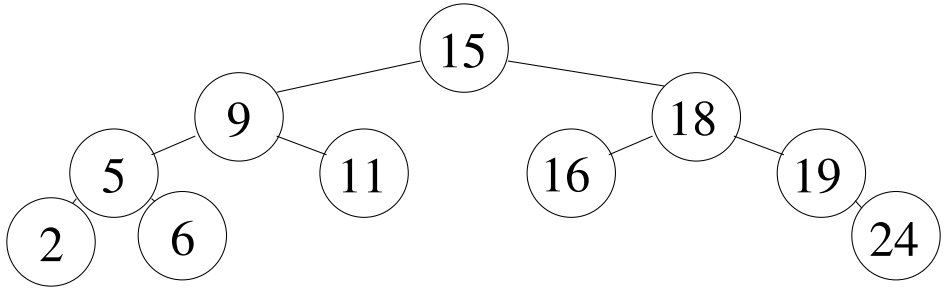
Axiomes Pour toutes les opérations : l'arbre renvoyé est un ABR

Rechercher(e, ArbreVide()) = ArbreVide()
Rechercher(e, N(g, r, d)) = N(g, r, d) si $e = r$
= Rechercher(e, g) si $e < r$
= Rechercher(e, d) si $r < e$
= ArbreVide() sinon
⋮

Opérations dérivées : on peut les coder à partir des opérations de base.
Mais ce sont elles qui maintiennent les caractéristiques de la structure.

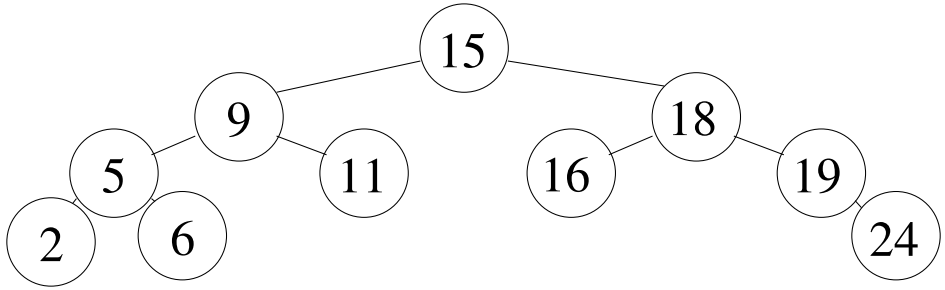
Plan

Rechercher



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

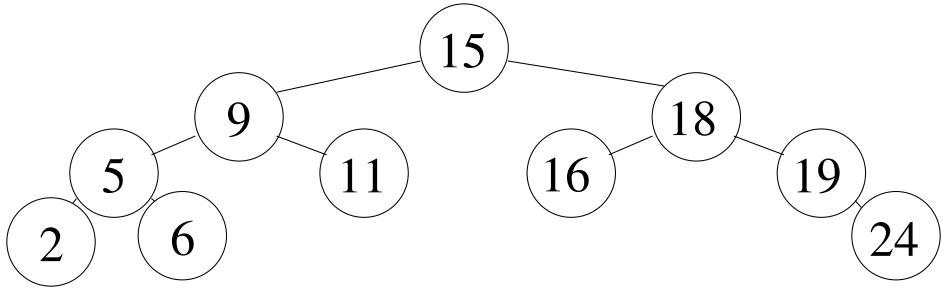
Rechercher



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

else if $e = Clé(a)$
 ∟ **return** a

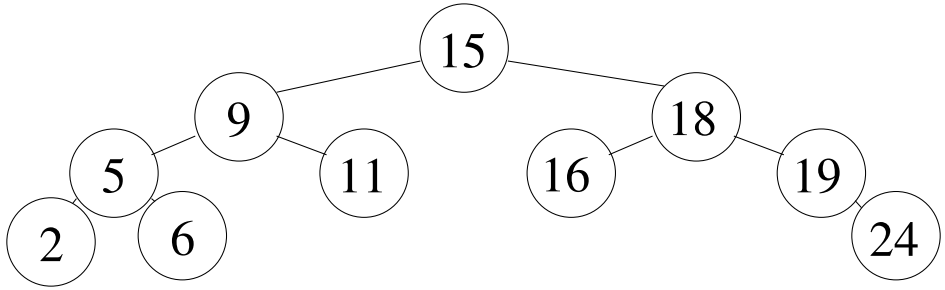
Rechercher



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

else if $e = Clé(a)$
  return a
else if $e < Clé(a)$
  return *RECHERCHER(e, FilsGauche(a))*

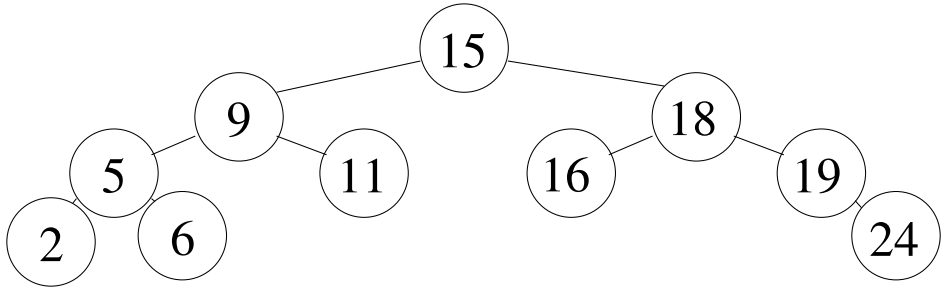
Rechercher



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

else if $e = Clé(a)$
  return a
else if $e < Clé(a)$
  return *RECHERCHER(e, FilsGauche(a))*
else // `e > Clé(a)`
  return *RECHERCHER(e, FilsDroit(a))*

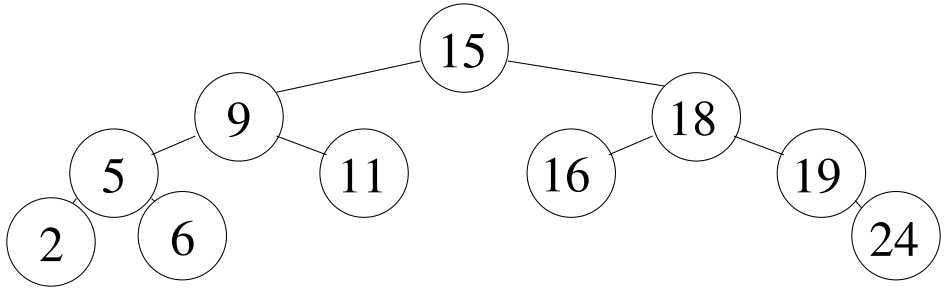
Rechercher



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

if *EstVide(a)*
  return *ArbreVide()*
else if $e = $ *Clé(a)*
  return a
else if $e <$ *Clé(a)*
  return *RECHERCHER(e, FilsGauche(a))*
else // $e >$ Clé(a)
  return *RECHERCHER(e, FilsDroit(a))*

Rechercher

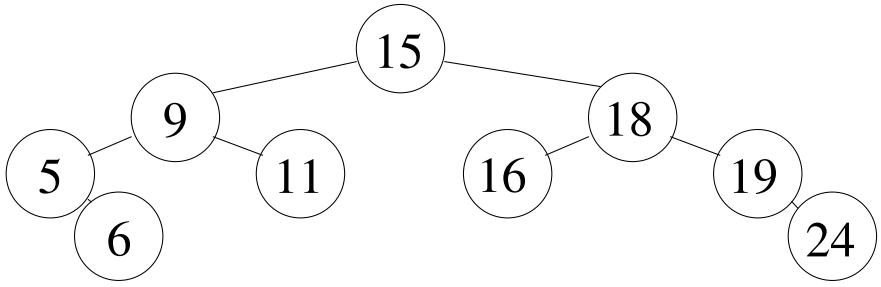


RECHERCHER(e, a)
Données : Une clé e, un ABR a
Résultat : Si possible un nœud de a dont la clé est e, sinon *ArbreVide*()

if *EstVide(a)*
  return *ArbreVide()*
else if $e = $ *Clé(a)*
  return a
else if $e <$ *Clé(a)*
  return *RECHERCHER(e, FilsGauche(a))*
else // $e >$ Clé(a)
  return *RECHERCHER(e, FilsDroit(a))*

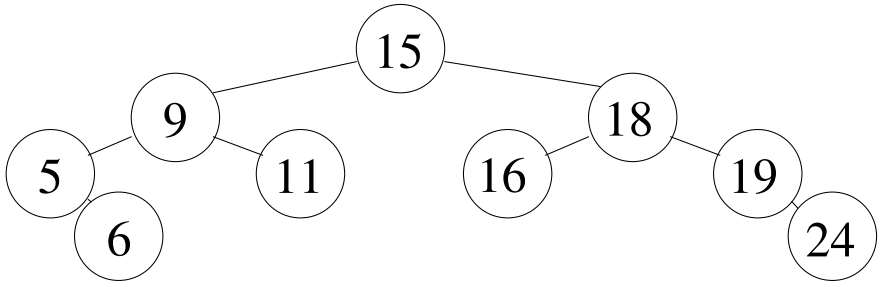
Variante : renvoyer une information associée à la clé plutôt que le nœud

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

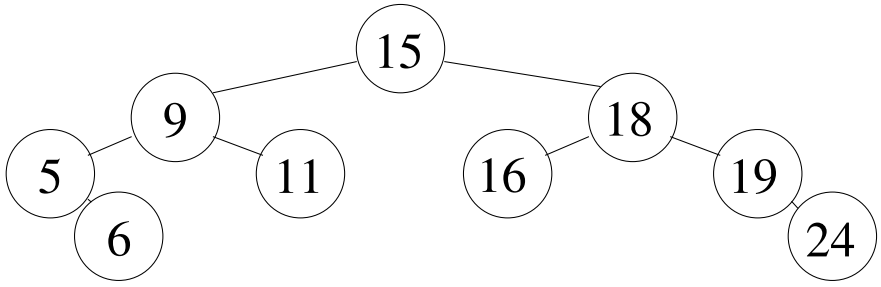
Minimum



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

else
  return *MINIMUM(FilsGauche(a))*

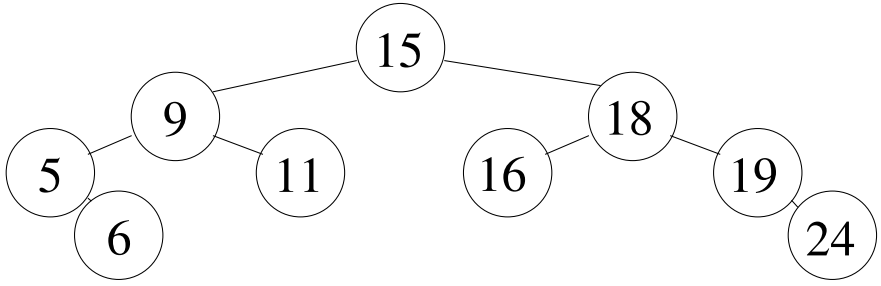
Minimum



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*

if *EstVide(FilsGauche(a))*
  return *a*
else
  return *MINIMUM(FilsGauche(a))*

Minimum



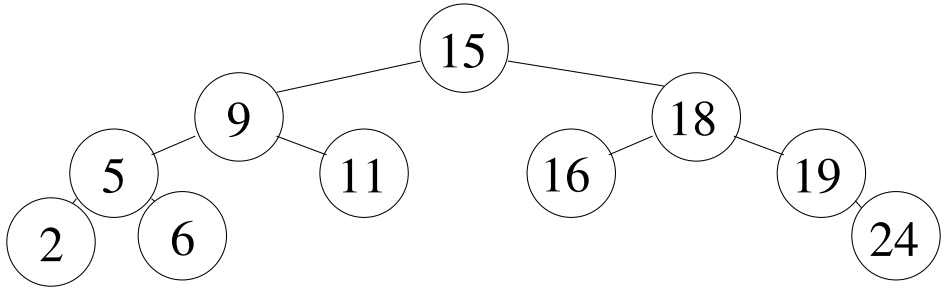
MINIMUM(a)
Données : Un ABR a non vide
Résultat : Un nœud de a dont la clé est inférieure à toutes les autres clés de a

if *EstVide(FilsGauche(a))*
  return a
else
  return *MINIMUM(FilsGauche(a))*

Remarque

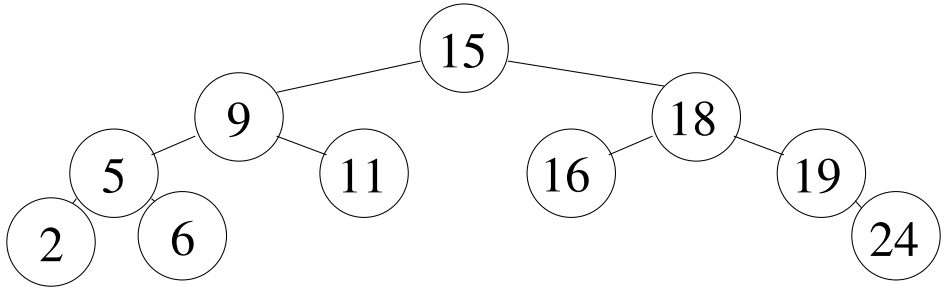
Cet algorithme et le précédent s'écrivent aussi bien sous forme impérative sans pile.

Insérer



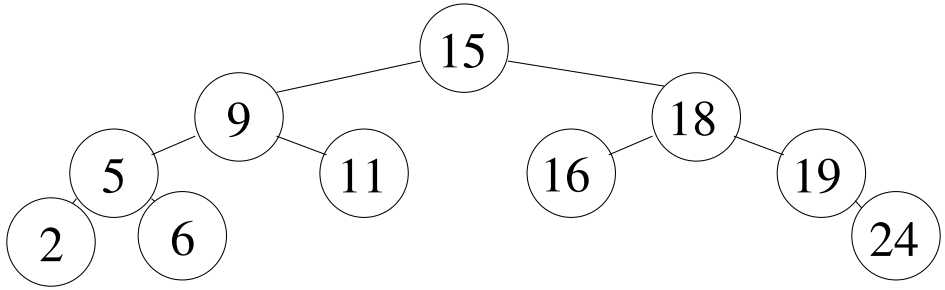
INSERER(e, a)
Données : Une clé e, un ABR a
Résultat : Un ABR dont les clés sont : e et les clés de a

Insérer



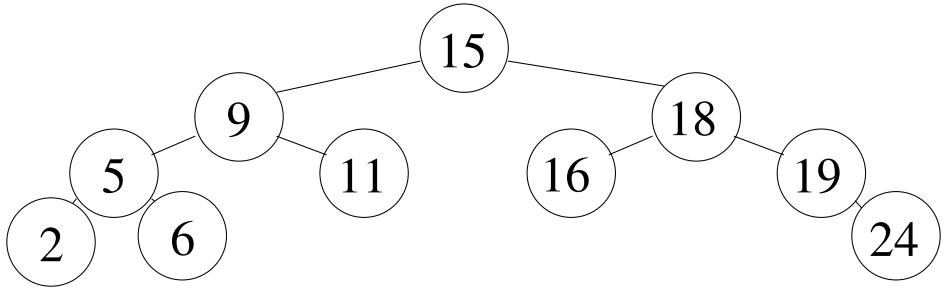
INSERER(*e*, *a*)
Données : Une clé *e*, un ABR *a*
Résultat : Un ABR dont les clés sont : *e* et les clés de *a*
if *EstVide(a)*
  return *N(ArbreVide(), e, ArbreVide())*

Insérer



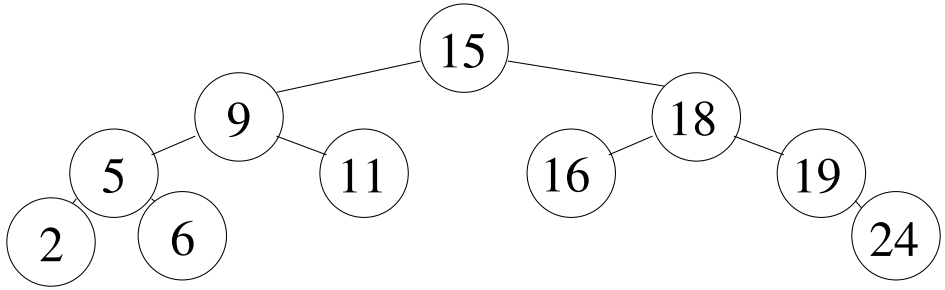
INSERER(e, a)
Données : Une clé e, un ABR a
Résultat : Un ABR dont les clés sont : e et les clés de a
if *EstVide(a)*
  return *N(ArbreVide(), e, ArbreVide())*
else if $e \leq$ *Clé(a)*
  return *N(INSERER(e, FilsGauche(a)), Clé(a), FilsDroit(a))*

Insérer



INSERER(e, a)
Données : Une clé e, un ABR a
Résultat : Un ABR dont les clés sont : e et les clés de a
if *EstVide(a)*
  return *N(ArbreVide(), e, ArbreVide())*
else if $e \leq$ *Clé(a)*
  return *N(INSERER(e, FilsGauche(a)), Clé(a), FilsDroit(a))*
else // $e >$ `Clé(a)`
  return *N(FilsGauche(a), Clé(a), INSERER(e, FilsDroit(a)))*

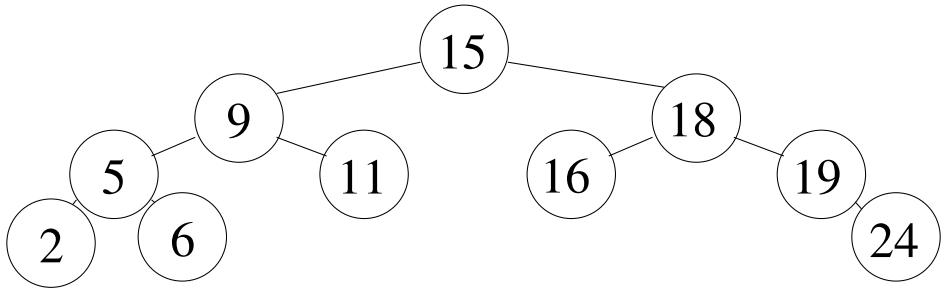
Insérer



INSERER(e, a)
Données : Une clé e, un ABR a
Résultat : Un ABR dont les clés sont : e et les clés de a
if *EstVide(a)*
  return *N(ArbreVide(), e, ArbreVide())*
else if $e \leq$ *Clé(a)*
  return *N(INSERER(e, FilsGauche(a)), Clé(a), FilsDroit(a))*
else // $e >$ `Clé(a)`
  return *N(FilsGauche(a), Clé(a), INSERER(e, FilsDroit(a)))*

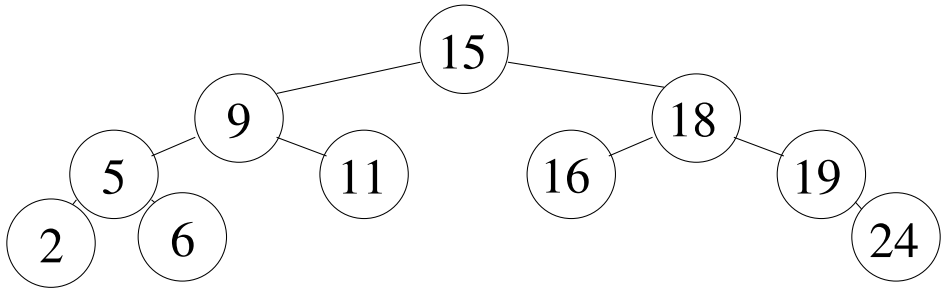
Variante : chaque clé est unique, si $e =$ *Clé*(a) on met à jour le nœud.

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

Supprimer

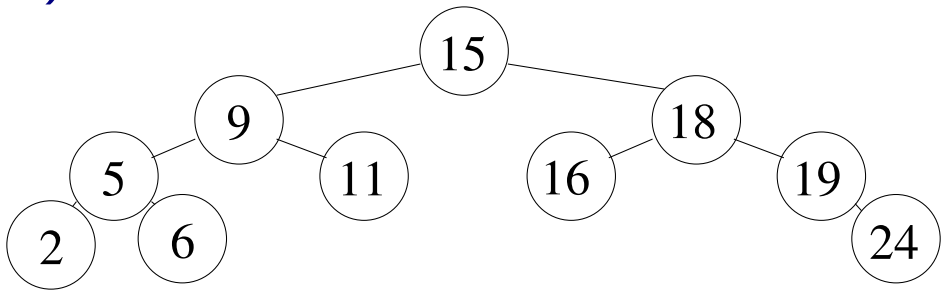


SUPPRIMER(e, a)
Données : Un élément e présent dans a, un ABR a
Résultat : Un ABR contenant les mêmes clés que a sauf e

... comme RECHERCHER mais...

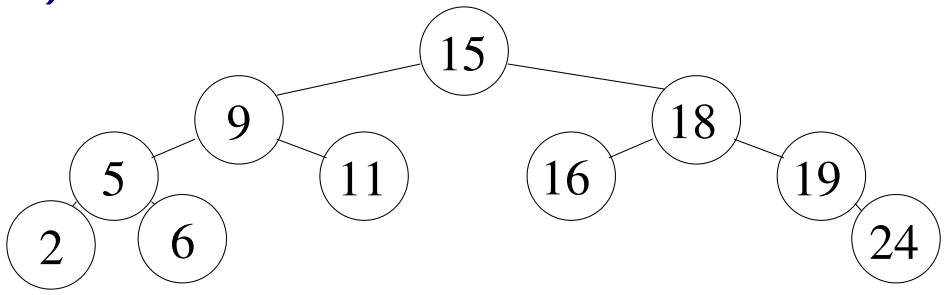
⋮
if $e = Clé(a)$
  return *SUPPRIMER_RACINE(a)*
⋮

Supprimer (2)



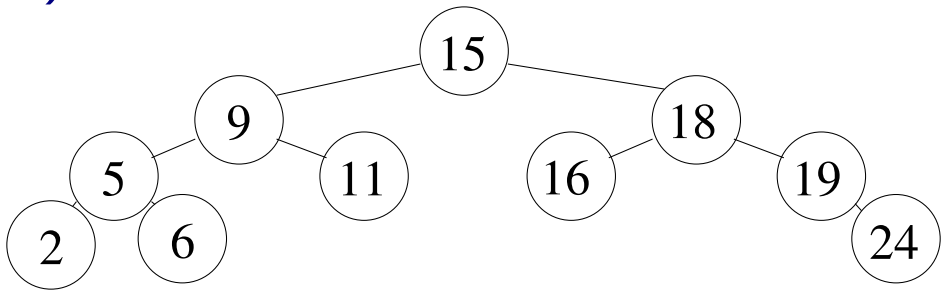
SUPPRIMER_RACINE(x)

Supprimer (2)



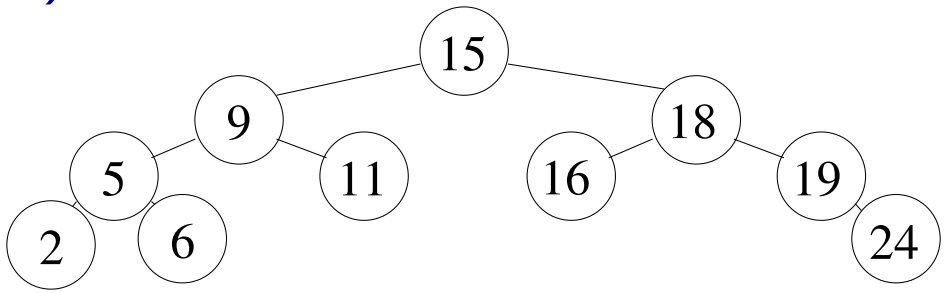
SUPPRIMER_RACINE(x)
if *EstVide(FilsGauche(x))* **return** *FilsDroit(x)*

Supprimer (2)



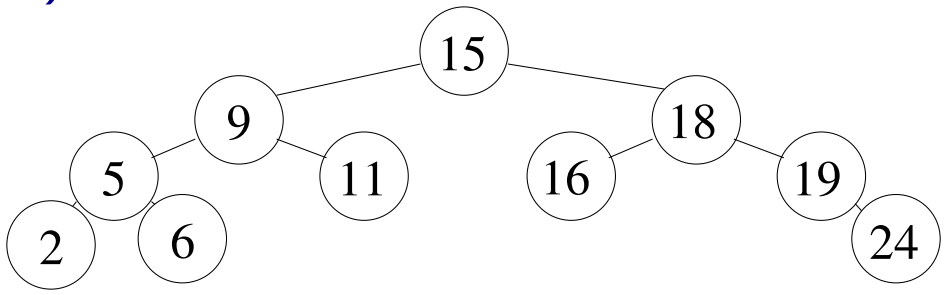
SUPPRIMER_RACINE(x)
if *EstVide(FilsGauche(x))* **return** *FilsDroit(x)*
else if *EstVide(FilsDroit(x))* **return** *FilsGauche(x)*

Supprimer (2)



SUPPRIMER_RACINE(x)
if *EstVide(FilsGauche(x))* **return** *FilsDroit(x)*
else if *EstVide(FilsDroit(x))* **return** *FilsGauche(x)*
else
  $m :=$ MINIMUM(FilsDroit(x))
  return *N(FilsGauche(x), m, SUPPR_MIN(FilsDroit(x)))*

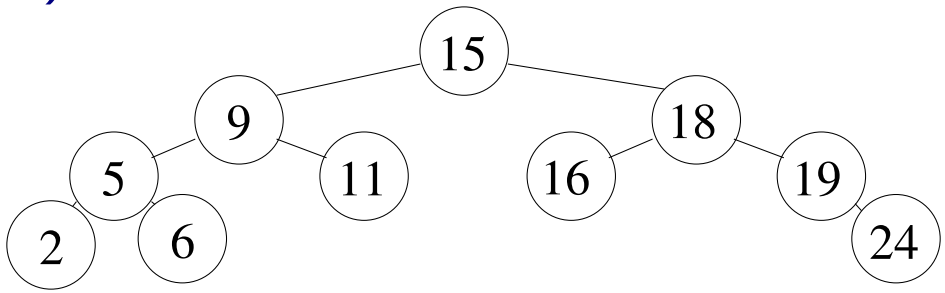
Supprimer (2)



SUPPRIMER_RACINE(x)
if *EstVide(FilsGauche(x))* **return** *FilsDroit(x)*
else if *EstVide(FilsDroit(x))* **return** *FilsGauche(x)*
else
  $m :=$ MINIMUM(FilsDroit(x))
  return *N(FilsGauche(x), m, SUPPR_MIN(FilsDroit(x)))*

SUPPR_MIN(x)
... (presque) comme MINIMUM mais ...

Supprimer (2)



SUPPRIMER_RACINE(x)
if *EstVide(FilsGauche(x))* **return** *FilsDroit(x)*
else if *EstVide(FilsDroit(x))* **return** *FilsGauche(x)*
else
  m := MINIMUM(FilsDroit(x))
  return *N(FilsGauche(x), m, SUPPR_MIN(FilsDroit(x)))*

SUPPR_MIN(x)
... (presque) comme MINIMUM mais ...
⋮
if *EstVide(FilsGauche(x))* **return** *FilsDroit(x)*
⋮

Plan

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.

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.

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).

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 détermine maintenant la structure moyenne d'un ABR de n nœuds construit par n insertions successives dans l'arbre vide.

Arbre moyen

On détermine maintenant 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.

En particulier la racine de l'ABR (qui est toujours la première clé insérée) peut porter de façon équiprobable la 1^{re}, la 2^{e}, ..., la n^{e} clé (par rapport à la liste triée).

Arbre moyen

On détermine maintenant 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.

En particulier la racine de l'ABR (qui est toujours la première clé insérée) peut porter de façon équiprobable la 1^{re}, la 2^{e}, ..., la n^{e} clé (par rapport à la liste triée).

Remarque : 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.

$$P(a) = \sum_{x \in a} d(x, racine(a))$$

Profondeur totale des nœuds dans un ABR

Somme des profondeurs

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

$$P(a) = \sum_{x \in a} d(x, racine(a))$$

Structure récursive

Pour tout arbre a de n nœuds,

$$\text{Si } a = N(g, r, d), \text{ alors } P(a) = P(g) + P(d) + n - 1.$$

Profondeur totale des nœuds dans un ABR

Somme des profondeurs

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

$$P(a) = \sum_{x \in a} d(x, racine(a))$$

Structure récursive

Pour tout arbre a de n nœuds,

$$\text{Si } a = N(g, r, d), \text{ alors } P(a) = P(g) + P(d) + n - 1.$$

$$P(a) \quad = \quad \sum_{x \in g} d(x, racine(a)) \; + \; \sum_{x \in d} d(x, racine(a)) \; + \; p(r)$$

Profondeur totale des nœuds dans un ABR

Somme des profondeurs

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

$$P(a) = \sum_{x \in a} d(x, racine(a))$$

Structure récursive

Pour tout arbre a de n nœuds,

$$\text{Si } a = N(g, r, d), \text{ alors } P(a) = P(g) + P(d) + n - 1.$$

$$\begin{aligned} P(a) &= \sum_{x \in g} d(x, racine(a)) + \sum_{x \in d} d(x, racine(a)) + p(r) \\ &= \sum_{x \in g} (d(x, racine(g)) + 1) + \sum_{x \in d} (d(x, racine(d)) + 1) + 0 \end{aligned}$$

Profondeur totale des nœuds dans un ABR

Somme des profondeurs

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

$$P(a) = \sum_{x \in a} d(x, racine(a))$$

Structure récursive

Pour tout arbre a de n nœuds,

$$\text{Si } a = N(g, r, d), \text{ alors } P(a) = P(g) + P(d) + n - 1.$$

$$\begin{aligned} P(a) &= \sum_{x \in g} d(x, racine(a)) + \sum_{x \in d} d(x, racine(a)) + p(r) \\ &= \sum_{x \in g} (d(x, racine(g)) + 1) + \sum_{x \in d} (d(x, racine(d)) + 1) + 0 \\ &= P(g) + nb\_noeuds(g) + P(d) + nb\_noeuds(d) \end{aligned}$$

Profondeur totale des nœuds dans un ABR

Somme des profondeurs

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

$$P(a) = \sum_{x \in a} d(x, racine(a))$$

Structure récursive

Pour tout arbre a de n nœuds,

$$\text{Si } a = N(g, r, d), \text{ alors } P(a) = P(g) + P(d) + n - 1.$$

$$\begin{aligned} P(a) &= \sum_{x \in g} d(x, racine(a)) + \sum_{x \in d} d(x, racine(a)) + p(r) \\ &= \sum_{x \in g} (d(x, racine(g)) + 1) + \sum_{x \in d} (d(x, racine(d)) + 1) + 0 \\ &= P(g) + nb\_noeuds(g) + P(d) + nb\_noeuds(d) \\ &= P(g) + P(d) + n - 1 \end{aligned}$$

Profondeur moyenne des nœuds dans un ABR

On s'intéresse à P_n la valeur moyenne de $P(a)$ pour tous les arbres a de n nœuds :

- l'équation de récurrence s'applique encore
- l'hypothèse d'équirépartition assure que les n cas de la forme « $i-1$ nœuds dans g, $n-i$ nœuds dans d » sont équiprobables.

Profondeur moyenne des nœuds dans un ABR

On s'intéresse à P_n la valeur moyenne de $P(a)$ pour tous les arbres a de n nœuds :

- l'équation de récurrence s'applique encore
- l'hypothèse d'équirépartition assure que les n cas de la forme « $i-1$ nœuds dans g, $n-i$ nœuds dans d » sont équiprobables.

On en déduit que :

$$P_n = \frac{1}{n}\sum_{i=1}^{n}(P_{i-1} + P_{n-i} + n - 1)$$

Profondeur moyenne des nœuds dans un ABR

On s'intéresse à P_n la valeur moyenne de $P(a)$ pour tous les arbres a de n nœuds :

- l'équation de récurrence s'applique encore
- l'hypothèse d'équirépartition assure que les n cas de la forme « $i-1$ nœuds dans g, $n-i$ nœuds dans d » sont équiprobables.

On en déduit que :

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

Profondeur moyenne des nœuds dans un ABR

On s'intéresse à P_n la valeur moyenne de $P(a)$ pour tous les arbres a de n nœuds :

- l'équation de récurrence s'applique encore
- l'hypothèse d'équirépartition assure que les n cas de la forme « $i-1$ nœuds dans g, $n-i$ nœuds dans d » sont équiprobables.

On en déduit que :

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

Profondeur moyenne des nœuds dans un ABR

On s'intéresse à P_n la valeur moyenne de $P(a)$ pour tous les arbres a de n nœuds :

- l'équation de récurrence s'applique encore
- l'hypothèse d'équirépartition assure que les n cas de la forme « $i-1$ nœuds dans g, $n-i$ nœuds dans d » sont équiprobables.

On en déduit que :

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

Profondeur moyenne des nœuds dans un ABR (2)

En éliminant la somme (cf détail au tableau) :

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

Profondeur moyenne des nœuds dans un ABR (2)

En éliminant la somme (cf détail au tableau) :

$$\begin{aligned} nP_n - (n-1)P_{n-1} &= 2P_{n-1} + n(n-1) - (n-1)(n-2) \\ \frac{P_n}{n+1} &= \frac{P_{n-1}}{n} + 2\frac{n-1}{n(n+1)} \end{aligned}$$

Profondeur moyenne des nœuds dans un ABR (2)

En éliminant la somme (cf détail au tableau) :

$$
\begin{aligned}
nP_n - (n-1)P_{n-1} &= 2P_{n-1} + n(n-1) - (n-1)(n-2) \\
\frac{P_n}{n+1} &= \frac{P_{n-1}}{n} + 2\frac{n-1}{n(n+1)} \\
\frac{P_n}{n+1} &= \frac{P_{n-1}}{n} + \frac{4}{n+1} - \frac{2}{n}
\end{aligned}
$$

Profondeur moyenne des nœuds dans un ABR (2)

En éliminant la somme (cf détail au tableau) :

$$
\begin{aligned}
nP_n - (n-1)P_{n-1} &= 2P_{n-1} + n(n-1) - (n-1)(n-2) \\
\frac{P_n}{n+1} &= \frac{P_{n-1}}{n} + 2\frac{n-1}{n(n+1)} \\
\frac{P_n}{n+1} &= \frac{P_{n-1}}{n} + \frac{4}{n+1} - \frac{2}{n} \\
\frac{P_n}{n+1} &= \sum_{i=1}^{n} \left(\frac{4}{i+1} - \frac{2}{i}\right)
\end{aligned}
$$

Profondeur moyenne des nœuds dans un ABR (2)

En éliminant la somme (cf détail au tableau) :

$$
\begin{aligned}
nP_n - (n-1)P_{n-1} &= 2P_{n-1} + n(n-1) - (n-1)(n-2) \\
\frac{P_n}{n+1} &= \frac{P_{n-1}}{n} + 2\frac{n-1}{n(n+1)} \\
\frac{P_n}{n+1} &= \frac{P_{n-1}}{n} + \frac{4}{n+1} - \frac{2}{n} \\
\frac{P_n}{n+1} &= \sum_{i=1}^{n} \left(\frac{4}{i+1} - \frac{2}{i}\right) \\
\frac{P_n}{n+1} &= 2\sum_{i=1}^{n} \frac{1}{i} + \frac{2}{n+1} - 4
\end{aligned}
$$

Profondeur moyenne des nœuds dans un ABR (2)

En éliminant la somme (cf détail au tableau) :

$$
\begin{aligned}
nP_n - (n-1)P_{n-1} &= 2P_{n-1} + n(n-1) - (n-1)(n-2) \\
\frac{P_n}{n+1} &= \frac{P_{n-1}}{n} + 2\frac{n-1}{n(n+1)} \\
\frac{P_n}{n+1} &= \frac{P_{n-1}}{n} + \frac{4}{n+1} - \frac{2}{n} \\
\frac{P_n}{n+1} &= \sum_{i=1}^{n} \left(\frac{4}{i+1} - \frac{2}{i}\right) \\
\frac{P_n}{n+1} &= 2\sum_{i=1}^{n} \frac{1}{i} + \frac{2}{n+1} - 4 \\
\frac{P_n}{n+1} &= 2\log n + \text{un terme négligeable devant } \log n
\end{aligned}
$$

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 $\mathcal{O}(\log n)$.

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 $\mathcal{O}(n \log n)$.

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 ?

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)

```
if non EstArbreVide(a)
    PARCOURS_PROFONDEUR_INFIXE ( FilsGauche(a) )
    Traiter( Clé(a) )
    PARCOURS_PROFONDEUR_INFIXE ( FilsDroit(a) )
```

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)

if *non EstArbreVide(a)*
  PARCOURS_PROFONDEUR_INFIXE (FilsGauche(a))
  Traiter(Clé(a))
  PARCOURS_PROFONDEUR_INFIXE (FilsDroit(a))

D'où l'algorithme de tri par ABR :

while *Il reste des données*
  Insérer une donnée dans l'ABR
Parcourir l'ABR en profondeur infixe.

Complexité du tri par ABR

- Coût du parcours en profondeur infixe :

Complexité du tri par ABR

- Coût du parcours en profondeur infixe : $\mathcal{O}(n)$
- Coût des n insertions successives :

Complexité du tri par ABR

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

Complexité du tri par ABR

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

Lien avec le tri rapide

- Le premier élément e_1 inséré dans l'arbre est mis à sa place définitive.

Complexité du tri par ABR

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

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.

Complexité du tri par ABR

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

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*

Complexité du tri par ABR

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

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.

Complexité du tri par ABR

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

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 :

- L'ABR procède de haut en bas, le tri rapide de bas en haut.

Complexité du tri par ABR

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

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 :

- L'ABR procède de haut en bas, le tri rapide de bas en haut.
- La segmentation du tri rapide peut bouleverser l'ordre d'insertion.

Complexité du tri par ABR

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

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 :

- L'ABR procède de haut en bas, le tri rapide de bas en haut.
- La segmentation du tri rapide peut bouleverser l'ordre d'insertion.

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

Plan

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 ?

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 $h = \mathcal{O}(\log n)$.

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

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 $f_1, \ldots f_i$ avec $t \leqslant i \leqslant 2t$
- ces i fils sont séparés par $i-1$ clés $k_1 < k_2 < \ldots k_{i-1}$
- 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

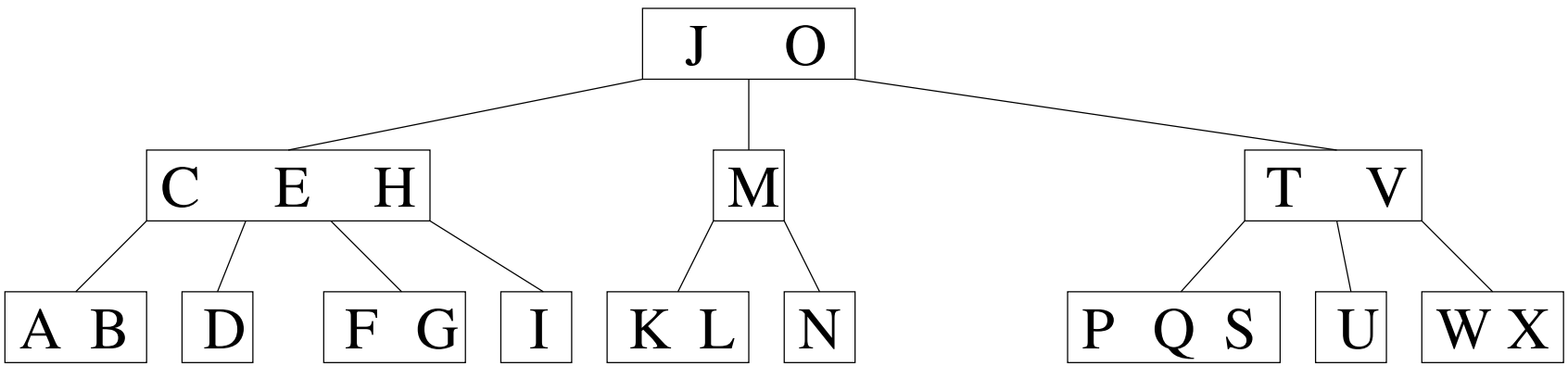
Les B-arbres

Structure

Similaire à un ABR mais :

- chaque nœud peut porter i fils $f_1, \ldots f_i$ avec $t \leqslant i \leqslant 2t$
- ces i fils sont séparés par $i-1$ clés $k_1 < k_2 < \ldots k_{i-1}$
- 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. Comme dans un ABR, on descend jusqu'à la feuille adaptée pour accueillir la nouvelle clé. $\mathcal{O}(h \times t)$
   - (à chaque nœud on parcourt la liste des clés pour trouver le bon fils)

Insertion

1. Comme dans un ABR, on descend jusqu'à la feuille adaptée pour accueillir la nouvelle clé. $\mathcal{O}(h \times t)$
   - (à chaque nœud on parcourt la liste des clés pour trouver le bon fils)
2. On insère la nouvelle clé à sa place dans ce nœud. $\mathcal{O}(t)$

Insertion

1. Comme dans un ABR, on descend jusqu'à la feuille adaptée pour accueillir la nouvelle clé. $\mathcal{O}(h \times t)$
   - (à chaque nœud on parcourt la liste des clés pour trouver le bon fils)
2. On insère la nouvelle clé à sa place dans ce nœud. $\mathcal{O}(t)$
3. Si après cette opération le nœud n'est pas plein ($i \leqslant 2t$), ok.

Insertion

1. Comme dans un ABR, on descend jusqu'à la feuille adaptée pour accueillir la nouvelle clé. $\mathcal{O}(h \times t)$
   - (à chaque nœud on parcourt la liste des clés pour trouver le bon fils)
2. On insère la nouvelle clé à sa place dans ce nœud. $\mathcal{O}(t)$
3. Si après cette opération le nœud n'est pas plein ($i \leqslant 2t$), ok.
4. Sinon ($i = 2t + 1$), on coupe ce nœud 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.

Insertion

1. Comme dans un ABR, on descend jusqu'à la feuille adaptée pour accueillir la nouvelle clé. $\mathcal{O}(h \times t)$
   - (à chaque nœud on parcourt la liste des clés pour trouver le bon fils)
2. On insère la nouvelle clé à sa place dans ce nœud. $\mathcal{O}(t)$
3. Si après cette opération le nœud n'est pas plein ($i \leqslant 2t$), ok.
4. Sinon ($i = 2t + 1$), on coupe ce nœud 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.
5. Si le nœud père est lui-même plein, on coupe à nouveau, autant de fois qu'il le faut en remontant vers la racine. $\mathcal{O}(h \times t)$

Insertion

1. Comme dans un ABR, on descend jusqu'à la feuille adaptée pour accueillir la nouvelle clé. $\mathcal{O}(h \times t)$
   - (à chaque nœud on parcourt la liste des clés pour trouver le bon fils)
2. On insère la nouvelle clé à sa place dans ce nœud. $\mathcal{O}(t)$
3. Si après cette opération le nœud n'est pas plein ($i \leqslant 2t$), ok.
4. Sinon ($i = 2t + 1$), on coupe ce nœud 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.
5. Si le nœud père est lui-même plein, on coupe à nouveau, autant de fois qu'il le faut en remontant vers la racine. $\mathcal{O}(h \times t)$
6. 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 de codage
- Entropie
- Algorithme de Huffman