

Algorithmique et Analyse d'Algorithmes

L3 Info

Cours 8 : Arbres partiellement ordonnés
Diviser pour régner

Benjamin Wack



2024 – 2025

La dernière fois

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

Aujourd'hui

- ▶ Arbre partiellement ordonné, tassé
- ▶ Structure de tas
- ▶ Application à la FAP
- ▶ Diviser pour régner

Plan

Arbre partiellement ordonné, tassé

Arbre partiellement ordonné

Arbre tassé

Structure de tas

Définition et applications

Opérations

Diviser pour Régner

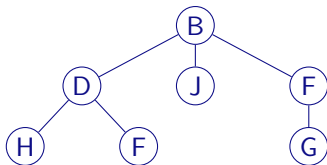
Définition

On suppose qu'on sait comparer les clés choisies pour les nœuds.

Arbre ordonné

Un arbre est **ordonné** si tout nœud a une clé inférieure ou égale à celles de chacun de ses fils (s'ils existent).

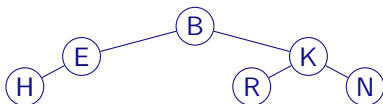
On convient qu'un arbre vide est ordonné.



On se limite ici à des arbres binaires :

- ▶ les algorithmes et propriétés présentés ici restent valables
- ▶ mais l'implantation est facilitée (cf TD)

Propriétés



- ▶ Chaque sous-arbre d'un arbre ordonné est lui-même un arbre ordonné.
(par définition)
- ▶ Dans tout chemin de l'arbre (d'un nœud vers une feuille), les clés sont en ordre croissant.
(par définition)
- ▶ La racine d'un arbre ordonné contient la clé de valeur minimum parmi les clés de l'arbre.
(récurrence structurelle sur l'arbre)

Propriétés

Structure

- ▶ Tous les niveaux sont complets, sauf éventuellement le dernier.
- ▶ Un arbre tassé contient au max. un nœud unaire et dans ce cas :
 - ▶ ce nœud n'a qu'un fils gauche ;
 - ▶ n est pair.

Hauteur et nombre de nœuds

- ▶ Le nombre de nœuds de chaque niveau complet i est 2^i .
- ▶ Le nombre total de nœuds est

$$n = \begin{array}{l} 2^h - 1 \\ + x \end{array} \quad \begin{array}{l} \text{(niveaux complets)} \\ \text{où } 0 < x \leq 2^h \text{ (dernier niveau)} \end{array}$$

- ▶ Par conséquent $h = \lfloor \log_2 n \rfloor$.

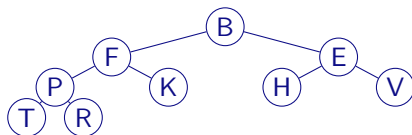
Définition

Tas binaire

Un **tas** (*heap* en anglais) est un arbre binaire **tassé** et **ordonné**.

Remarque

Ne pas confondre avec la zone d'allocation dynamique en mémoire.



Comme pour les ABR :

- ▶ on s'appuie sur le type abstrait Arbre Binaire
- ▶ on construit ensuite les opérations utiles

Mais on a encore le choix de la représentation de l'arbre en mémoire (pas forcément sous forme chaînée notamment).

Applications

Réalisation d'une **file à priorités** efficace

- ▶ On ordonne les nœuds de l'arbre par priorité (quitte à prendre un *tas max* si nécessaire)
- ▶ Le nœud prioritaire (racine) est accessible en temps constant
- ▶ Pas d'information superflue à maintenir en cas d'insertion / suppression

On retrouvera donc un tas dans tous les algorithmes qui demandent de gérer des priorités :

- ▶ Algorithme de **Huffman**
- ▶ Algorithmes de graphes (**Prim**, **Dijkstra**)

Application « évidente » : **tri par tas** (*heapsort*)

- ▶ Insérer les éléments à trier dans un tas, puis les extraire un par un.
- ▶ Version plus efficace en place avec un bon choix de représentation.

Ensemble minimal d'opérations

Tas_Vide : $void \rightarrow Tas$

Est_Vide : $Tas \rightarrow bool$

Insérer : $Element \times Tas \rightarrow void$

Ajoute (*par effet de bord*) l'élément à ceux déjà présents.
Si besoin, réorganise les éléments pour conserver un tas.

Trouver_Min : $Tas \rightarrow Element$

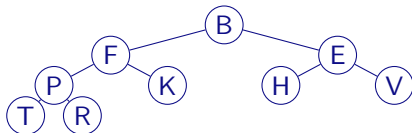
Permet de consulter l'élément minimum sans modifier le tas (pas d'effet de bord, il suffit de consulter la racine).

Extraire_Min : $Tas \rightarrow void$

Supprime (*par effet de bord*) l'élément de clé minimale.
Si besoin, réorganise les éléments pour conserver un tas.
S'il y a plusieurs minimums, un seul est supprimé.

(En pratique on ordonne souvent selon une clé (priorité) à laquelle on associe une valeur.)

Insertion



Idée générale : il faut que l'arbre reste **tassé** et **ordonné**.

- ▶ **Tassé.** On ne peut ajouter le nouveau nœud que :
 - ▶ au dernier niveau, après la dernière feuille ;
 - ▶ ou si le dernier niveau est complet, au tout début du prochain niveau.
- ▶ **Ordonné.** On procède par échange de clés *sans modifier la structure de l'arbre*, le long du chemin de la nouvelle feuille à la racine.

INSERER(e, t)

Créer une nouvelle feuille n de clé e après la dernière feuille de t

$p := \text{Père}(n)$

tant que n n'est pas la racine et $\text{clé}(p) > \text{clé}(n)$

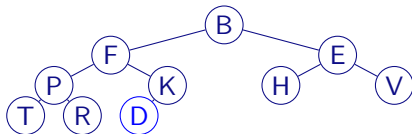
Échanger les clés de p et de n

$n := p$

$p := \text{Père}(n)$

(Percolation
vers le haut)

Insertion



Idée générale : il faut que l'arbre reste **tassé** et **ordonné**.

- ▶ **Tassé.** On ne peut ajouter le nouveau nœud que :
 - ▶ au dernier niveau, après la dernière feuille ;
 - ▶ ou si le dernier niveau est complet, au tout début du prochain niveau.
- ▶ **Ordonné.** On procède par échange de clés *sans modifier la structure de l'arbre*, le long du chemin de la nouvelle feuille à la racine.

INSERER(e, t)

Créer une nouvelle feuille n de clé e après la dernière feuille de t

$p := \text{Père}(n)$

tant que n n'est pas la racine et $\text{clé}(p) > \text{clé}(n)$

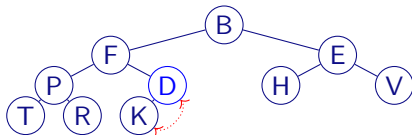
Échanger les clés de p et de n

$n := p$

$p := \text{Père}(n)$

} (Percolation
vers le haut)

Insertion



Idée générale : il faut que l'arbre reste **tassé** et **ordonné**.

- ▶ **Tassé.** On ne peut ajouter le nouveau nœud que :
 - ▶ au dernier niveau, après la dernière feuille ;
 - ▶ ou si le dernier niveau est complet, au tout début du prochain niveau.
- ▶ **Ordonné.** On procède par échange de clés *sans modifier la structure de l'arbre*, le long du chemin de la nouvelle feuille à la racine.

INSERER(e, t)

Créer une nouvelle feuille n de clé e après la dernière feuille de t

$p := \text{Père}(n)$

tant que n n'est pas la racine et $\text{clé}(p) > \text{clé}(n)$

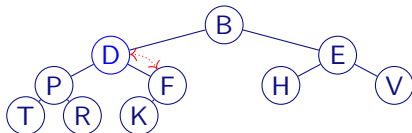
Échanger les clés de p et de n

$n := p$

$p := \text{Père}(n)$

(Percolation
vers le haut)

Insertion



Idée générale : il faut que l'arbre reste **tassé** et **ordonné**.

- ▶ **Tassé**. On ne peut ajouter le nouveau nœud que :
 - ▶ au dernier niveau, après la dernière feuille ;
 - ▶ ou si le dernier niveau est complet, au tout début du prochain niveau.
- ▶ **Ordonné**. On procède par échange de clés *sans modifier la structure de l'arbre*, le long du chemin de la nouvelle feuille à la racine.

INSERER(e, t)

Créer une nouvelle feuille n de clé e après la dernière feuille de t

$p := \text{Père}(n)$

tant que n n'est pas la racine et $\text{clé}(p) > \text{clé}(n)$

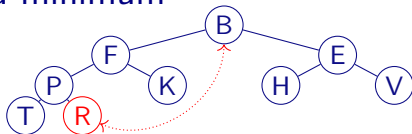
Échanger les clés de p et de n

$n := p$

$p := \text{Père}(n)$

(Percolation
vers le haut)

Extraction du minimum



Même principe : l'arbre devra rester...

- ▶ **Tassé.** On doit supprimer la dernière feuille du dernier niveau. Cependant le minimum est à la racine : on commence par échanger.
- ▶ **Ordonné.** On procède par échange de clés *sans modifier la structure de l'arbre*, en descendant cette fois vers les feuilles.

EXTRAIRE_MIN(t)

f := Dernière feuille de t

n := Racine de t

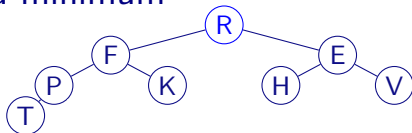
Échanger les clés de f et de n

Supprimer f

tant que n n'est pas une feuille et $\text{clé}(n) > \text{clé d'un fils de } n$ } *Percolation vers le bas*

m := Fils de n de clé minimale
 Échanger les clés de n et de m
 n := m

Extraction du minimum



Même principe : l'arbre devra rester...

- ▶ **Tassé.** On doit supprimer la dernière feuille du dernier niveau. Cependant le minimum est à la racine : on commence par échanger.
- ▶ **Ordonné.** On procède par échange de clés *sans modifier la structure de l'arbre*, en descendant cette fois vers les feuilles.

EXTRAIRE_MIN(t)

f := Dernière feuille de t

n := Racine de t

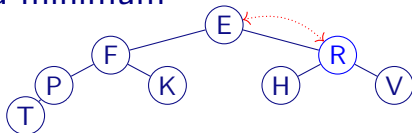
Échanger les clés de f et de n

Supprimer f

tant que n n'est pas une feuille et $\text{clé}(n) > \text{clé d'un fils de } n$ } *Percolation vers le bas*

m := Fils de n de clé minimale
 Échanger les clés de n et de m
 n := m

Extraction du minimum



Même principe : l'arbre devra rester...

- ▶ **Tassé.** On doit supprimer la dernière feuille du dernier niveau. Cependant le minimum est à la racine : on commence par échanger.
- ▶ **Ordonné.** On procède par échange de clés *sans modifier la structure de l'arbre*, en descendant cette fois vers les feuilles.

EXTRAIRE_MIN(t)

f := Dernière feuille de t

n := Racine de t

Échanger les clés de f et de n

Supprimer f

tant que n n'est pas une feuille et $\text{clé}(n) > \text{clé d'un fils de } n$

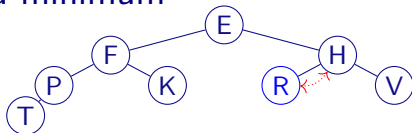
m := Fils de n de clé minimale

 Échanger les clés de n et de m

$n := m$

} Percolation
vers le bas

Extraction du minimum



Même principe : l'arbre devra rester...

- ▶ **Tassé.** On doit supprimer la dernière feuille du dernier niveau. Cependant le minimum est à la racine : on commence par échanger.
- ▶ **Ordonné.** On procède par échange de clés *sans modifier la structure de l'arbre*, en descendant cette fois vers les feuilles.

EXTRAIRE_MIN(t)

f := Dernière feuille de t

n := Racine de t

Échanger les clés de f et de n

Supprimer f

tant que n n'est pas une feuille et $\text{clé}(n) > \text{clé d'un fils de } n$

m := Fils de n de clé minimale

 Échanger les clés de n et de m

n := m

} Percolation
vers le bas

Complexité

Les opérations qui peuvent être coûteuses sont la comparaison et l'échange de clés.

Les deux opérations **Insérer** et **Extraire_min** sont constituées d'une boucle effectuant à chaque itération :

- ▶ une comparaison (ou deux)
- ▶ un échange

le long d'un seul chemin : leur coût est majoré par la **hauteur de l'arbre**.

Celui-ci étant tassé :

Complexité de la mise à jour du tas

La complexité des opérations **Insérer** et **Extraire_min** est en $\mathcal{O}(\log_2 n)$.

Opérations supplémentaires

Pour certains algorithmes, on a besoin d'une opération supplémentaire

Modifier_priorité (Noeud n , Priorité p)

Priorité(n) := p

si clé(n) < clé(Parent(n))

└ Percolation vers le haut

si clé(n) > clé d'un fils de n

└ Percolation vers le bas

On peut également définir des opérations dérivées des précédentes :

- ▶ **Tri_Par_Tas** (Tableau t)
- ▶ **Créer_tas** (Tableau t de taille n)
 - ▶ Version naïve par n insertions successives en $\mathcal{O}(n \log n)$
 - ▶ Version efficace procédant des feuilles vers la racine en $\mathcal{O}(n)$

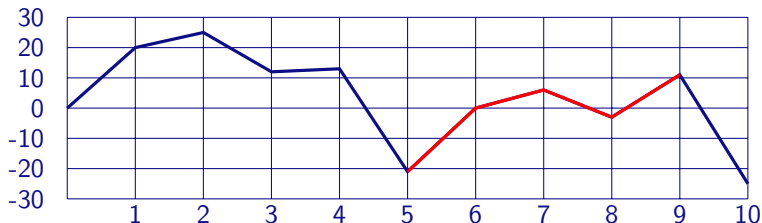
Un exemple : le problème du sous-tableau maximal

Soit un tableau $T[0..n-1]$ d'entiers relatifs.

On cherche deux indices i et j tels que la somme des entiers de $T[i..j]$ soit maximale.

0	1	2	3	4	5	6	7	8	9
20	5	-13	1	-34	21	6	-9	14	-36

Une visualisation possible : les sommes cumulées croissantes



Algorithme (très) naïf

On essaye tous les couples (i, j) et pour chacun on calcule la somme :

```

max := T[0] // Initialisation avec T[1..1]
(imax, jmax) := (0, 0)
pour i := 0 à n - 1
  pour j := i à n - 1
    s := 0
    pour k := i à j // Calcul de la somme de T[i..j]
      s := s + T[k]
    si s > max
      max := s
      (imax, jmax) := (i, j)
renvoyer (max, imax, jmax)
  
```

Complexité :

- ▶ On a trois « vraies » boucles imbriquées (i, j, k)
- ▶ On peut démontrer que la complexité est en $\mathcal{O}(n^3)$

Comment améliorer cet algorithme naïf ?

Clairement, il faut éviter de recalculer la somme de $T[i..j]$ à partir de 0 à chaque fois :

```

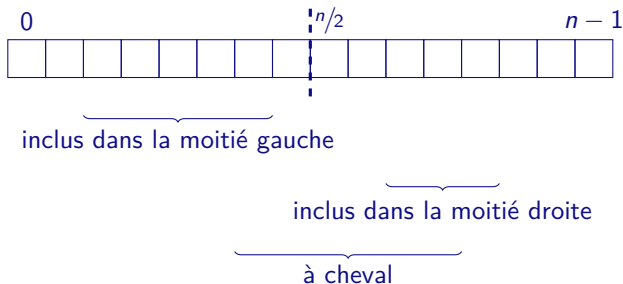
max := T[0]
(imax, jmax) := (0, 0)
pour i := 0 à n - 1
  s := 0
  pour j := i à n - 1
    s := s + T[j]           // Mise à jour continue de s = T[i..j]
    si s > max
      max := s
      (imax, jmax) := (i, j)
renvoyer (max, imax, jmax)
  
```

Complexité : $\sum_{i=0}^{n-1} (n - i) = \frac{n(n+1)}{2} = \mathcal{O}(n^2)$

Peut-on faire encore mieux ?

Nicolas Machiavel, *Le Prince* : « Divide et impera » (Divise et règne)

Imaginons qu'on coupe le tableau en deux par le milieu : le sous-tableau maximal est forcément dans un de ces trois cas



Algorithme récursif

Pour les deux cas où le tableau recherché est dans une des deux moitiés, un des appels récursifs nous donnera directement la réponse :

```

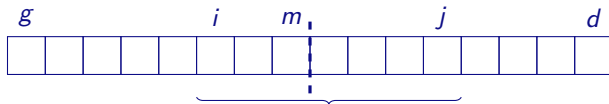
SOUSTABLEAUMAX(g, d)
/* Renvoie le triplet (somme, borne gauche, borne droite)      */
/* de valeur maximale dans T[g..d]                          */
si g = d                                                    // Cas de base
└ renvoyer ( T[g], g, g)
sinon
┌ milieu := (g + d) / 2
  (maxG, iG, jG) := SOUSTABLEAUMAX(g, milieu)           // 2 appels
  (maxD, iD, jD) := SOUSTABLEAUMAX(milieu + 1, d)     // récursifs

  (maxC, iC, jC) := CHEVALMAX(g, d)

└ Choisir le meilleur résultat parmi les 3 précédents.
  
```

Il reste à coder la fonction CHEVALMAX, qui cherche un sous-tableau maximal à cheval sur les deux moitiés.

Recherche d'un sous-tableau maximal « à cheval »



$T[i..j]$ contient forcément $T[i..m]$ et $T[m..j]$:
 on cherche **séparément** le meilleur i puis le meilleur j .

CHEVALMAX(g, d)

$m := (g + d) / 2$

$s := T[m]$

$maxG := s$

$imax := m$

pour $i := m - 1$ à g

$s := s + T[i]$

si $s > maxG$

$maxG := s$

$imax := i$

// Puis on fait pareil pour $jmax$ entre $milieu + 1$ et d

Complexité de la méthode récursive

- ▶ Complexité de CHEVALMAX
Deux boucles successives de *milieu* à *g* puis de *milieu* à *d*
donc une complexité en $\mathcal{O}(d - g)$ (taille du segment considéré)
- ▶ Complexité de SOUSTABLEAUMAX
Notons $C(n)$ le coût de SOUSTABLEAUMAX sur un segment de taille n :

$$C(n) = 2C(n/2) + \mathcal{O}(n) + \mathcal{O}(1)$$

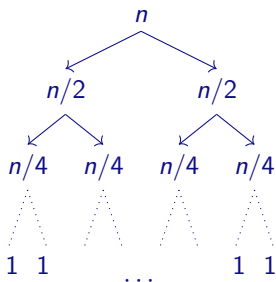
appels récursifs

CHEVALMAX

choix de la solution

Résolution de l'équation de récurrence

Intuition : arbre des appels récursifs, avec la taille des sous-tableaux



- ▶ arbre de hauteur $\log_2 n$
- ▶ chaque niveau coûte $\mathcal{O}(n)$

Conjecture

SOUSTABLEAUMAX a un coût
 $C(n) = kn \log_2 n + kn$.

où kn est le coût de CHEVALMAX.

En effet : supposons cette propriété vraie pour tous les rangs $< n$, alors

$$\begin{aligned}
 C(n) &= 2C(n/2) + kn \\
 &= 2k \frac{n}{2} \log_2 \frac{n}{2} + 2k \frac{n}{2} + kn \\
 &= kn(\log_2 n - 1) + kn + kn \\
 &= kn \log_2 n - kn + kn + kn \\
 &= kn \log_2 n + kn
 \end{aligned}$$

D'où $C(n) = \mathcal{O}(n \log n)$.

Quand divise-t-on pour régner ?

On peut parler de diviser pour régner si :

- ▶ le problème initial est découpé en parties de **taille équivalente** ;
- ▶ chaque partie peut être résolue **récurivement** ;
- ▶ la solution globale peut être **reconstruite** à partir de la solution de chaque partie.



John von Neumann
(1903-1957)

Par exemple, ces algorithmes relèvent-ils du Diviser pour Régner ?

- ▶ tri rapide **parties inégales** (cf tri par fusion au semestre 6)
- ▶ drapeau hollandais **non, pas récursif**
- ▶ recherche dichotomique **oui**
- ▶ fonctions sur un ABR **oui s'il est équilibré**
- ▶ vérification du parenthésage **non, pas d'indépendance des parties**
- ▶ insertion dans un tas **non**

En résumé

Aujourd'hui

- ▶ Pour accéder **rapidement au maximum** d'un ensemble, il suffit de maintenir une structure **partiellement ordonnée**.
- ▶ La structure de **tas** convient et elle permettra de réaliser une **File à Priorités** efficace.
- ▶ La stratégie **diviser pour régner** permet d'accélérer certains algorithmes **à condition de pouvoir traiter récursivement une fraction des données**.

La prochaine fois

- ▶ Arbre de codage
- ▶ Entropie
- ▶ Algorithme de Huffman