Structures de données. Exemples et applications

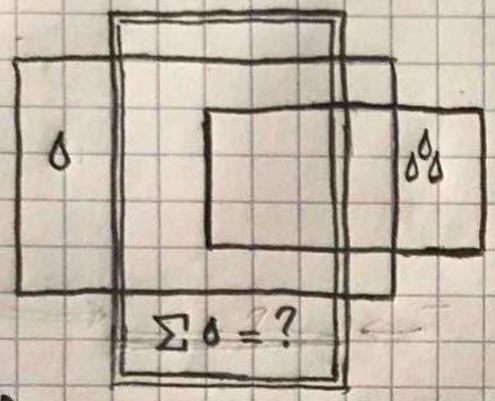
Prérequis: Algorithmie élémentaire, dont manipulation de tableaux, tris naifs. Probabilités élémentaires.

I) Structures statiques: Interface, efficacité

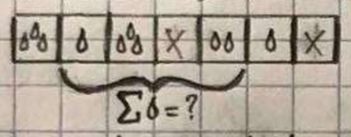
Problème:

Simulateur d'agriculture:

- Averses rectangulaires
- Calculs de somme sur fenêtres rectangulaires



1▷ Simplification: - Monde en 1D
- Pluie avant les calculs



Sol 1: Tableau + calcul linéaire

Interface: ‖ Build(init) ‖ GetSum(start, end)

S1	S2	S3
Θ(size)	Θ(size²)	Θ(size)
Θ(end-start)	Θ(1)	Θ(1)

Sol 2: Mémoïsation dans tableau 2D

Sol 2 est redondante par Chasles: $\sum_{i=a}^{b-1} t_i = \sum_{i=0}^{b-1} t_i - \sum_{i=0}^{a-1} t_i = P_b - P_b$

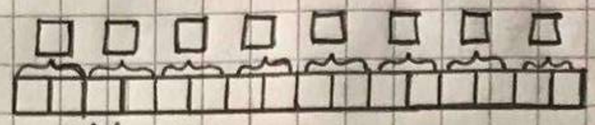
Sol 3: Calcul d'une primitive

⚠ Les données d'origine ne sont plus stockées directement.

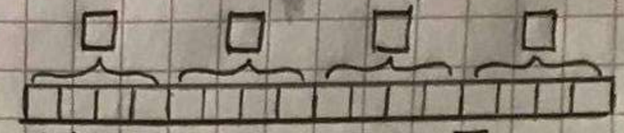
2▷ Rajout à l'interface: ‖ AddPoint(position, value) ⇒ Θ(n − position) [S3*]

↳ Trop long pour solution "étendue" Sol 3*

Idée: faire des précalculs dont peu doivent être mis à jour



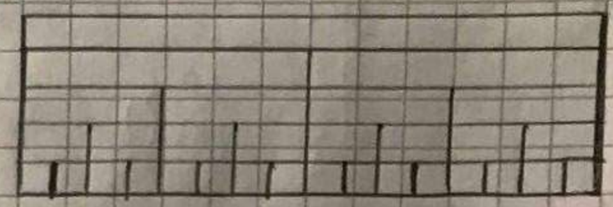
Sol 4: une case par paire
⇒ gain d'un facteur 2 par rapport à Sol 1



Sol 5: une case par groupe de $\sqrt{n}$
⇒ gain d'un facteur $\sqrt{n}$ / grands intervalles

Sol 6: comme Sol 4, mais récursivement

Build(init)	Θ(size)
GetSum(start, end)	O(log(size))
AddPoint(position, value)	Θ(log(size))



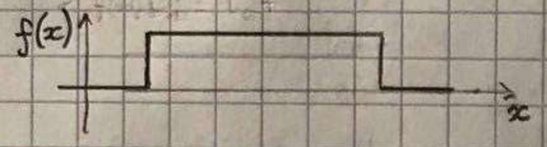
3▷ Rajout à l'interface: ‖ AddInterval(start, end, value) ⇒ Θ((end-start)+log(size)) [S6*]

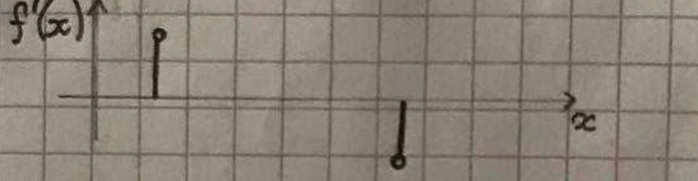
↳ Trop long pour solution "étendue" Sol 6*

Nouvelle interface:

	objectif
Build(init)	O(size)
GetValue(position)	O(log(size))
AddInterval(start, end, value)	O(log(size))

Idée: $f(x) = f(0) + \int_0^x f'(x)\,dx$





↳ Sol 7: Sol 6 appliquée à Δp

Retour à l'interface complète

4▷ Même travail en 2D

} Développement 1
Implémentation en Python

II) Structures dynamiques: principe de référence

Rappel: tri par insertion: long (Θ(n²) en pire des cas) à cause de l'insertion

référence: position d'un élément (ex: adresse mémoire, indice dans tableau, etc)

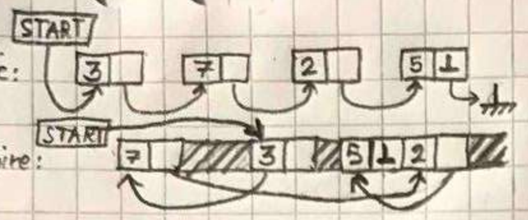
Idée: pour déformer une structure, stocker des références dedans

I) Listes

1) Liste chaînée :

1 élément ⇔ { 1 valeur ; 1 ref vers un élément }

Abstraite :



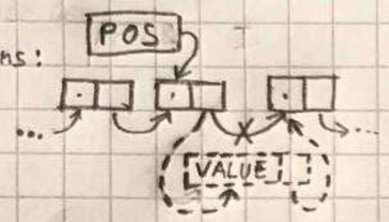
En mémoire :

Interface :

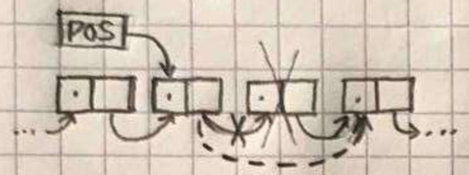
Build(init)	
InsertAfter(refPosition, value)	H (1)
Enumerate()	H (size)
AccessKth(k)	⚠ H (k) ⚠
DeleteAfter(refPosition)	H (1)

(H (size))

Opérations :



InsertAfter(POS, VALUE)

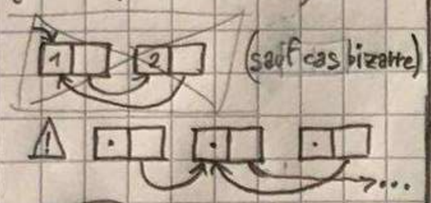


DeleteAfter(POS)

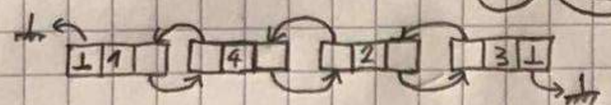
Tri par insertion : pas mieux que $O(n^2)$

Tri fusion : insérer, en un parcours, $\lfloor \frac{n}{2} \rfloor$ éléments (déjà triés, récursivement) dans une liste chaînée de $\lceil \frac{n}{2} \rceil$ éléments (déjà triés, récursivement) } $O(\log(n))$

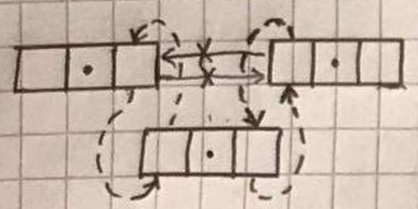
⚠ Seules certaines listes sont bien formées.
Attention aux objets non référencés,
et à ceux plusieurs fois référencés.

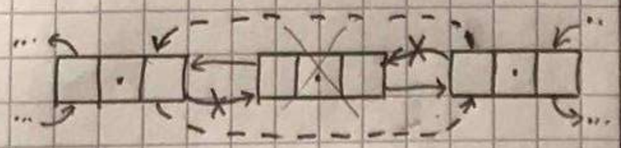


Listes doublement chainées :

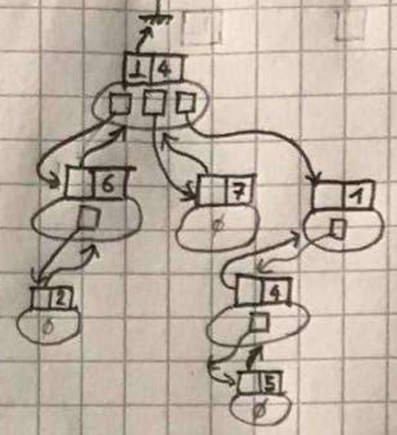


Chaque élément a deux références. Chaque référence a son inverse.
Plus maniable, mais plus lourd et plus long.





2) Arbre :



1 "noeud" ⇔ { 1 valeur ; 1 ref vers un noeud "parent" ; collection de noeuds "fils" }

Bien formé lorsque :
- ▷ parent(x) = y ⇔ x ∈ fils(y) ou y = ⊥
- ▷ parent(x) ≠ x ; parent(parent(x)) ≠ x ; etc.
- ▷ parent(x) = ⊥ pour exactement un noeud

• Dans certains cas, pas besoin de stocker { le parent ; la collection des fils } dits "implicites"
•

Arbre binaire lorsque exactement deux fils (qui peuvent être ⊥)

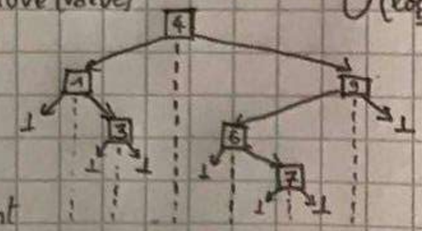
Exemple : les structures de la fin de partie I (parent et fils implicites)

II) Stratégies de complexité : analyse en moyenne et amortie

1) Problème : Ensemble d'objets triés. Interface :

	objectif
Insert(value)	O(log(size))
Contains?(value)	O(log(size))
EnumerateOrdered()	O(size)
Remove(value)	O(log(size))

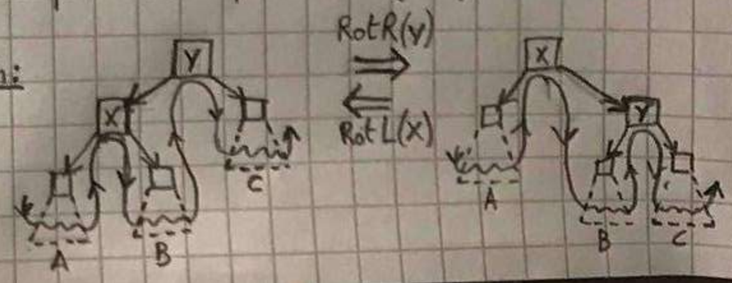
Idée : diviser pour régner ?
Arbre Binaire de Recherche
↳ ordre infixe



Recherche en O(profondeur) naïvement
Insertion naïvement, suppression aussi

Problème : profondeur peut être Θ(size).

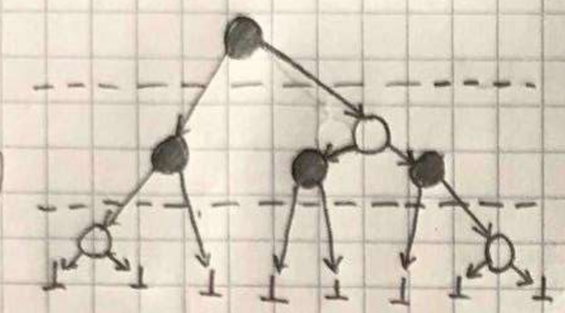
Rotation :



Propriétés: • Préserve l'ordre infixe
• Est réversible par une rotation
• Permet de passer de n'importe quel ABR à un autre avec les mêmes valeurs

Démonstration: par induction

↳ Utiliser les rotations pour forcer l'arbre à être équilibré
Exemple: arbre AVL
Exemple: arbre rouge-noir
opérations en $O(\log(n))$



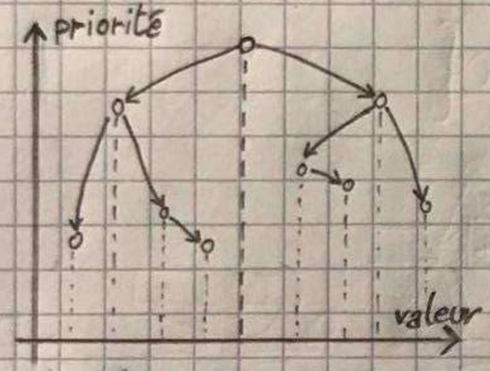
↳ détails d'implémentation en TD ou en TP ou en DM

2.4 Quand est-ce que la méthode naïve est efficace?

Définition: complexité en moyenne : espérance de la complexité dans un modèle proba.

Propriété: Pour des insertions en ordre uniformément aléatoire, l'algo naïf est $O(\log(n))$ en moyenne pour chaque opération.
↳ Pas toujours suffisant ⚠

Définition: un arbre est un tas pour $\prec_p$ lorsque $\forall x,\ x \prec_p \text{parent}(x)$



Propriété: Si $\prec_p$ et $\prec_v$ sont des ordres totaux, il y a un unique arbre ABR pour $\prec_v$ et tas pour $\prec_p$
↳ Appelé arbre cartésien. Facile à faire.

Remarque: L'insertion naïve dans l'ordre inverse de $\prec_p$ donne le même ABR.

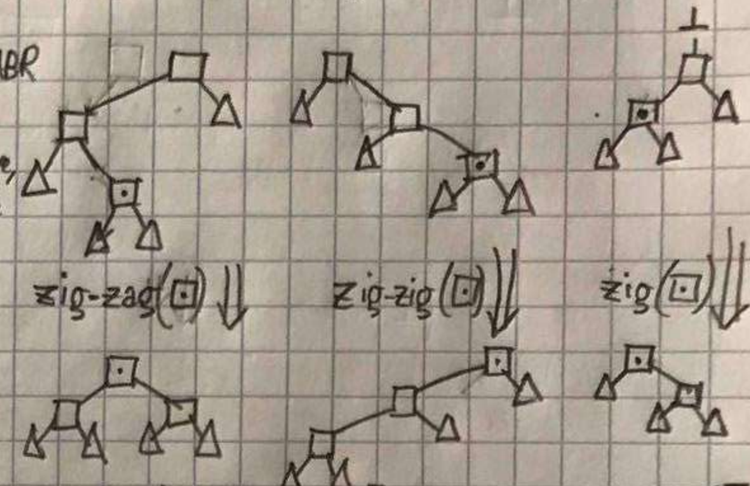
↳ Algorithme randomisé indépendamment des requêtes, avec la bonne complexité en moyenne.

3.4 Définition: complexité amortie: moyenne de complexité sur une séquence d'opérations (arbitrairement longue)

Exemple: Tableau à redimensionnement dynamique

Définition-propriété: Potentiel $\Phi(s)$ est fonction minorée par une constante.
Si pour toute opération $s \to s'$, son coût + $\underbrace{\Phi(s') - \Phi(s)}_{\text{amortissement}} \leq M$,
Alors le coût amorti total est $\leq M$

Définition: Arbre Splay: ABR



• après chaque opération naïve, le noeud concerné remonte à la racine par séquence de rotations.
Choix des quelles selon l'ascendance du noeud.

Potentiel $\Phi(T) = \sum_{x \in T} r(x,T) = \sum_{x \in T} \log_2\left[\text{Card}(\text{sousArbre}(T,x))\right]$

Propriété: coût amorti de (en # rotation)
$\text{zig}(x): T \to T'$ est $\leq 1 + 3\left(r(x,T) - r(x,T')\right)$
$\text{zigzig}(x): T \to T'$ est $\leq 3\left(r(x,T) - r(x,T')\right)$
$\text{zigzag}(x): T \to T'$ est $\leq 3\left(r(x,T) - r(x,T')\right)$

↳ Pareil pour splay. Démonstration par convexité de log

Utilisation: Link-cut Tree: manipule une forêt dynamique malgré des déséquilibres

Link(a, b)	$O(\log(\text{size}))$
Cut(a)	$O(\log(\text{size}))$
GetRoot(a)	$O(\log(\text{size}))$

↳ Objet du deuxième développement, avec implementation en C/C++ et analyse de complexité