Explicitér AVL

901

Structures de données, exemples et application.

Les types (concrets) sont utiles pour structurer les données et repérer certaines erreurs de programmation. Les types abstraits encapsulent les types concrets, permettant de les utiliser comme des boîtes noires, c-à-d sans en connaître l'implémentation.

Mais l'efficacité dépend de l'implémentation, c'est pourquoi nous présentons ici des structures de données efficaces respectant le cahier des charges qu'est le type abstrait.

Déf Un type concret est la description d'un format de représentation interne des données en machine. [BBC] p 37

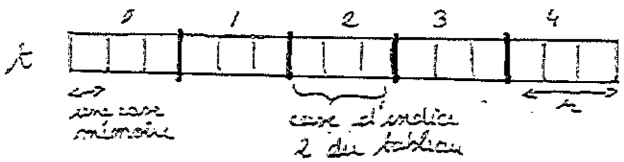
Déf Un type abstrait (de données) est la description d'un ensemble de données et des opérations que l'on peut y appliquer. [BBC] p 38.

Déf Une structure de données pour un type abstrait est la donnée d'un type concret et des implémentations des différentes fonctions associées.

I. Structures de base

1) Tableaux

(*) → Faire simple !



exemple de tableau de taille 5 pour un type codé sur 3 cases mémoire

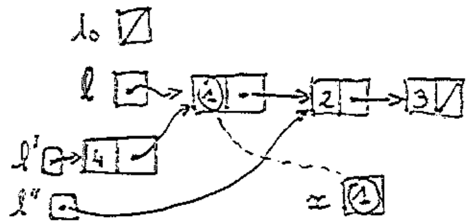
↳ On peut accéder à la i-ème case du tableau en temps constant car il suffit de lire i×n cases plus loin.

Rq En doublant la taille du tableau quand il est plein, et en le divisant par deux quand moins d'un quart du tableau est plein, on implémente de manière efficace une table dynamique. La lecture et l'écriture ont alors une complexité amortie de $O(1)$.

2) Listes

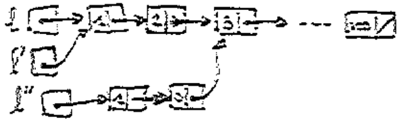
liste simplement chaînée.

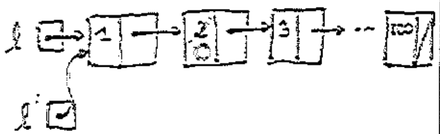
l_0 = crée_liste_vide()
l = [1;2;3]
l' = ajoute(4, l)
l'' = queue(l)
x = tête(l)



Rq

PERSISTANT	MUTABLE
l=[1;2…100]; l'=l; l''=remplace(l,2,0)	l=[1;2;…100]; l'=l; remplace(l,2,0)





Deux listes persistantes ayant la même fin peuvent partager de la mémoire car une modification sur l'une n'affecte pas l'autre.

Si deux listes mutables partagent de la mémoire, une modification sur l'une peut affecter l'autre car ici on ne recopie pas, on change.

→ Efficace

Rq Existent aussi les listes doublement chaînées, cycliques... cf annexe 1.

Rq Une liste paresseuse p est donnée par une fonction f qui en énumère les éléments, et une liste mutable l qui contient les éléments déjà calculés. Pour faire une opération sur p on la fait sur l en lui ajoutant si possible et si nécessaire des éléments calculés par f.

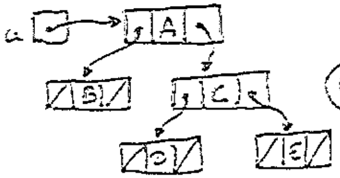
trop avancé

3) Arbres binaires

ex:

2 implémentations possibles.

[Froi] p 112. p 117.



OU

0	1	2	3	4	5	6	7	
	A		C	B		D	E	élmt
	4		6	/		/	/	fils gauche
	3		7	/		/	/	fils droit

{0, 2, 5, … } liste des cases libres.

car arbre pas complet

implémentation avec des pointeurs

implémentation avec un tableau d'indices.

⚠ Choisir ses propres définitions

[Froi] p 74
[BBC] p 40

II Ordonnancement

1) Piles

créer_vide () → pile
empiler (x: X; p: pile) → pile
est_vide (p: pile) → booléen
sommet (p: pile) → X
dépiler (p: pile) → X

- implémentation avec une liste
  ↳ toutes les opérations se font alors en temps constant
- application au parcours en profondeur (de graphes).

[Froi] p 77
[BBC] p 42

2) Files

créer_vide () → file
enfiler (x: X, f: file) → file
est_vide (f: file) → booléen
tête (f: file) → X
défiler (f: file) → X

- implémentation avec une liste conservant un pointeur vers la dernière cellule [cf annexe 2
  ↳ toutes les opérations se font alors en temps constant.
- applica° → parcours en largeur

3) Files de priorité (MIN)

Une file de priorité représente un ensemble d'élément munis de clés dans un ensemble totalem^t ordonné.
Ici nous nous intéressons aux files de priorité MIN, dans lesquels l'élément de clé minimale est privilégié.

Def ?

[BBC] p 56 →

tas binaire
↓
[Froi] p 343.

créer_vide () → file de priorité (FDP)
insérer (x: X; f: FDP) → FDP
est_vide (f: FDP) → booléen
min (f: FDP) → X
extraire_min (f: FDP) → X
diminuer_clé (c: clé, x: X; f: FDP) → FDP

- implémentation par $\boxed{\text{tas binaire}}$
  ↳ créer_vide, est_vide, min en $O(1)$
  ↳ insérer, extraire_min en $O(\lg(n))$
  ↳ diminuer clé en $O(\log(n))$ si on sait aller d'un élmt à son nœud en $O(1)$ — avec un tableau par exemple si les élm^t sont $[1..n]$ — en $O(n)$ sinon

Applications
- Construction d'un arbre couvrant de coût min avec Prim [C] p 583
- Plus court chemin à une source avec Dijkstra [C] p 609

Rq La structure de tas sert aussi pour le tri par tas, mais pour utiliser pleinement cette structure il faut ajouter à l'interface une fonction de création de tas à partir d'une liste, ce qui se fait alors en O(longueur de la liste). [DVP]

Rq Avec les tas de Fibonacci diminuer_clé a une complexité amortie en $O(1)$ (ADTIS) [C] p 457

III Ensembles et ~~dictionnaires~~ + app: dictionnaire

[Froi] p 85

DICTIONNAIRE

créer_vide () → dictionnaire
est_vide (d: dico) → booléen
chercher (c: clé, d: dico) → X
ajouter (x: X, c: clé, d: dico) → unité
supprimer (c: clé; d: dico) → unité

Le dictionnaire, aussi appelé tableau associatif, représente une fonction partielle des clés vers les valeurs, ici de type X.

Un ensemble, vu comme ensemble de définition de cette fonction partielle, est un dictionnaire dont les valeurs n'importent pas. En effet en remplaçant X par unité ci-dessus, on obtient le type ensemble.

1) Implémentation naïve

[Froi] p 88

On peut représenter un ensemble par une liste, auquel cas il faut vérifier à chaque ajout qu'on ne va pas créer de doublons.
De même avec une liste de couple (clé, valeur) on implémente dictionnaire.

2) Cas des clés entières

[Froi] p 89

Si l'on sait que les clés sont des entiers entre 0 et N, on peut implémenter le dictionnaire par un tableau de valeurs. De même pour un ensemble inclus dans [0..N] on peut utiliser un tableau de booléens indiquant si l'élément est ou non dans l'ensemble.

Ex avec $N = 7$, $S = \{1, 2, 5\}$ $f = \begin{pmatrix} 1 \to A \\ 2 \to C \\ 5 \to B \end{pmatrix}$

S :

0	1	1	0	0	1	0	0

f :

/	A	C	/	/	B	/	/

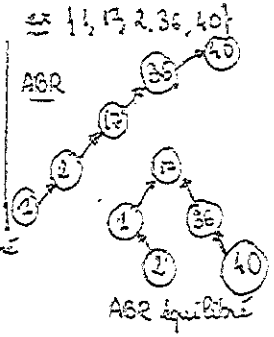
↓
connaître mais PAS la place ...

⚠ Tjs donner l'implémentation naïve.

pas besoin de détailler / pas très intéressant

3) Cas des clés ordonnées

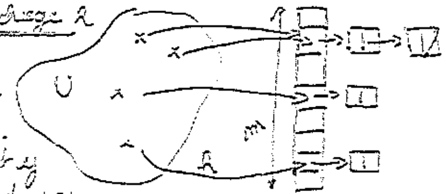
Si les clés sont dans un ensemble totalement ordonné, on peut utiliser des arbres binaires de recherche, c-à-d arbre de clés dans lequel chaque nœud est plus grand que son fils gauche et strictement plus petit que son fils droit. Il existe des améliorations des ABR comme les AVL qui s'attachent à garder un arbre équilibré.



4) Tables de hachage

Lorsque les clés peuvent s'étaler dans un ensemble U, très grand, la solution du tableau n'est pas satisfaisante, voire impossible si U est infini.

On utilise alors une fonction de hachage h qui envoie U dans $[1..m]$.
Pour une clé $k \in U$ on mettra la valeur correspondante dans l'alvéole $h(k)$.



Comme h n'est pas injective il peut y avoir des collisions. On les résoud par chaînage, c-à-d que chaque alvéole contient une liste chaînée à laquelle on peut ajouter des éléments sans écraser les précédents.
Si X est une variable aléatoire à valeur dans U telle que $\forall i \in [1..m]$ $P(h(X) = i) = \frac{1}{m}$ on dit qu'elle vérifie l'hypothèse de hachage simple.
Après l'insertion de $X_1, X_2 \dots X_n$ clés aléatoires vérifiant cette hypothèse, la longueur moyenne d'une liste est $E[n_j] \leq \frac{n}{m} = \alpha :=$ le taux de remplissage.
Pour maintenir ce taux bas, on est amené à agrandir la table (la augmentation)
↳ complexité amortie comme pr table dynamique

5) Complexités comparées
→ En temps !

très important [Se]

	liste	tableau de N cases	ABR / AVL	table de hachage
espace mémoire	$O(n)$	$O(N)$	$O(n)$	$O(n+m)$
créer-vide	$O(1)$	$O(N)$	$O(1)$	$O(m)$
est-vide	$O(1)$	$O(1)$	$O(1)$	$O(m)$ donc $O(n)$
chercher	$O(n)$	$O(1)$	$O(h)$ / $O(\log(n))$	$O(1 + n/m)$ en moyenne
ajouter*	$O(1)$	$O(1)$	$O(h)$ / $O(\log(n))$	$O(1)$ amorti
supprimer	$O(n)$	$O(1)$	$O(h)$ / $O(\log(n))$	$O(1 + n/m)$ en moyenne

⚠ ajouter doit être appliqué pour un élém^t non déjà présent.

[Co] p 235

[Co] p 241

IV Graphes

créer_vide() → graphe
ajouter_sommet (g: graphe, s: X) → unité
ajouter_arc (g: graphe, s: X, s: X) → unité
retirer_sommet (g: graphe, s: X) → unité
retirer_arc (g: graphe, s: X, s: X) → unité
est_sommet (g: graphe, s: X) → booléen
est_arc (g: graphe, s: X, s: X) → booléen
prédécesseurs (g: graphe, s: X) → liste de X
successeurs (g: graphe, s: X) → liste de X

$G = (S, A)$ $n = \#S$, $m = \#A$ [Froi] p143

- implémentation par une matrice d'adjacence
$$(M_{ij})_{(i,j) \in S^2} = \begin{cases} 0 & \text{si } (i,j) \notin A \\ 1 & \text{si } (i,j) \in A \end{cases}$$
- implémentation par une table dynamique d'ensembles de successeurs implémentés par des AVL.
$\forall i \in S$, $T[i] = \{j \in S \mid (i,j) \in A\}$

Rq. Lorsque le graphe est trop gros, si par exemple c'est le graphe des états d'un jeu, mais qu'on sait calculer les successeurs, c-à-d les configurations suivantes possibles, on choisit une implémentation implicite c-à-d qu'on ne stocke pas le graphe mais qu'on sait l'explorer.

matrices d'adjacence ...

- Si l'on dispose de deux implémentations d'un même type abstrait on peut les combiner en une troisième dont les opérations de lecture prennent la meilleure complexité des deux, et celles d'écriture la pire (cf ⓐ et ⓑ)

opération \ structure	matrice d'adjacence	tableau ⓐ d'ensembles de successeurs	tableau ⓑ d'ensembles de prédécesseurs	ⓐ et ⓑ
espace mémoire	$O(n^2)$	$O(n+m)$	$O(n+m)$	$O(n+m)$
s_i est-il successeur de s_j / s_i —— prédécesseur ——	$O(1)$	$O(\log(\deg^+))$	$O(\log(\deg^-))$	$O(\log(\min(\deg^+, \deg^-)))$
parcours des succ. / parcours des préd.	$O(n)$	$O(\deg^+)$ / $O(n+m)$	$O(n+m)$ / $O(\deg^-)$	$O(\deg^+)$ / $O(\deg^-)$
ajouter arc	$O(1)$	$O(\log(\deg^+))$	$O(\log(\deg^-))$	$O(\log(\max(\deg^+, \deg^-)))$
ajouter sommet	$O(n)$ amorti	$O(1)$ amorti	$O(1)$ amorti	$O(1)$ amorti

V Partition

créer_vide () → partition
créer_classe (x: X, p: partition) → unité
trouver (x: X, p: partition) → X
union (x: X, y: X, p: partition) → unité

[BBC] p61
[Co] p526

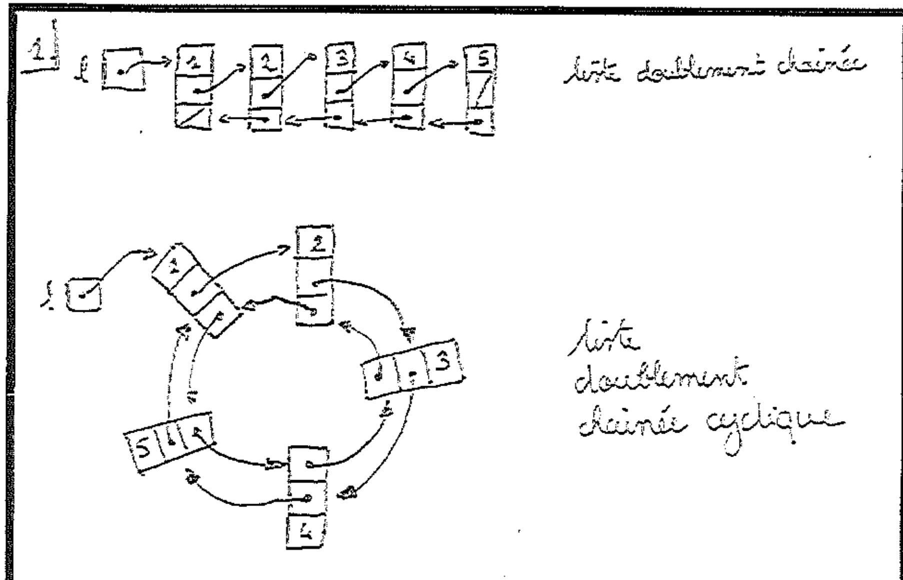
- implémentation par une forêt d'arbres ascendants cf annexe 3.
- amélioration: compression de chemin ← DEV
- application: algorithme de Kruskal

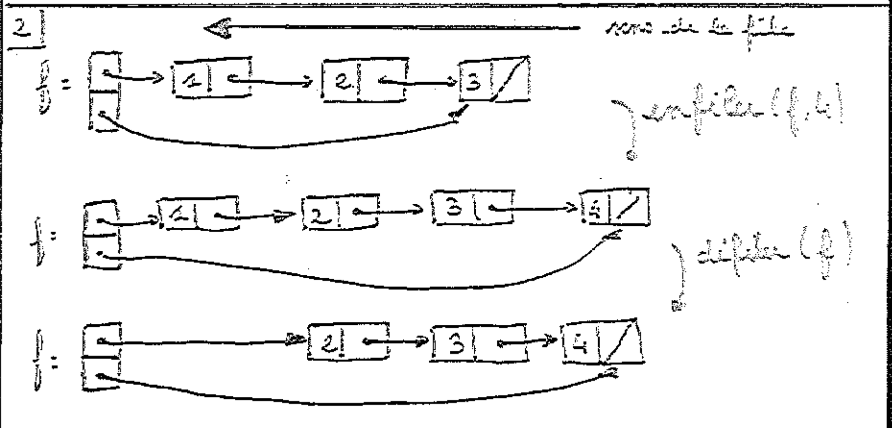
[Papa] p132

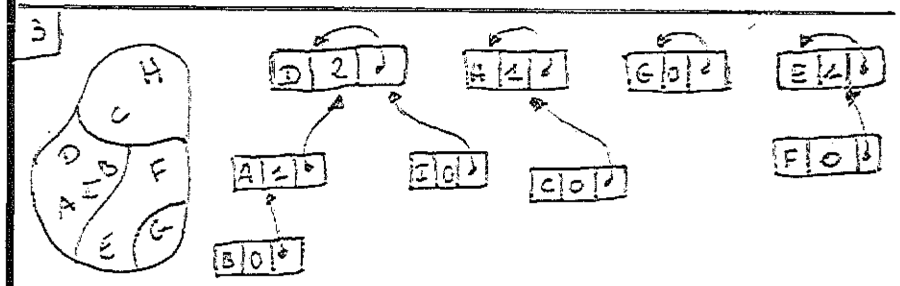
Kruskal → [Co] p 583

Si dvpt: Il faut écrire l'algo de base ds le plan / descript° pas assez précise

[Co] p219







Cas trié :

Tableau = bien pr la recherche dichotomique

Liste chainée : Mieux pr ajouter et supprimer, bof pr la recherche

⚠ Ce qui est présenté est très spécifique au C. (*)

Persistant / mutable : pour l'efficacité(1) MAIS quand on écrit un pgme, est-ce qu'il y a d'autres préoccupations ? ⇝ CORRECTION

Écrire : "facilite la preuve de correction". ↳ P

II ⇝ Piles / files : autres applications ?

Rq : Graphes : très important MAIS en APPLICATions

Rq : On est pas obligé d'écrire tous les types abstraits

Biblio

[BBC] Éléments d'algorithmique, Beauquier, Berstel, Chrétienne

[Co] Algorithmique, Cormen, Leiserson, Rivest, Stein

[Fr] Types de données et algorithmes, Froidevaux, Gaudel, Soria

[Papa] Algorithms, Dasgupta, Papadimitriou, Vazirani.

[Se] Algorithms, Sedgewick, Wayne.

Tables de hachage [Se] ?

⚠ Aux dvpts sur les tables de hachage - les bouquins sont bof

LE LOGARITHME ÉTOILE

cf. [illegible] de Papadimitriou p 136

Déf $\log^* = \begin{pmatrix} \mathbb{N}^* \longrightarrow \mathbb{N} \\ n \longmapsto \inf\{k \in \mathbb{N} \mid \log_2^{k}(n) \leq 1\} \end{pmatrix}$

ex $\left.\begin{array}{l} \log_2(3) > \log_2(2) = 1 \\ \log_2(\log_2(3)) \leqslant \log_2(\log_2(4)) = \log_2(2) = 1 \end{array}\right] \to \log_*(3) = 2.$

Pté $\log^*$ est croissante

Preuve Si $n \leq p$, puisque le $\log_2$ est st $\nearrow$ et que ses itérés aussi, on a $\forall k \in \mathbb{N}$ $\log_2{}^{k}(n) \leq \log_2{}^{k}(p)$.

En particulier $\log_2{}^{\log_*(p)}(n) \leq \log_2{}^{\log_*(p)}(p) \leq 1$

Donc néc. $\log_*(n) \leq \log_*(p)$.

Notation On note $(2^{\boxed{k}})_{k \in \mathbb{N}}$ la suite définie par $\begin{cases} 2^{\boxed{0}} = 1 \\ \forall k \in \mathbb{N}\ 2^{\boxed{k+1}} = 2^{2^{\boxed{k}}} \end{cases}$

"$2^{\boxed{k}} = 2^{2^{\cdot^{\cdot^{2}}}}$ k fois"

Pté

- $\log_2(2^{\boxed{0}}) = 0$ et $\forall k \in \mathbb{N}$ $\log_2(2^{\boxed{k}}) = 2^{\boxed{k-1}}$. (a)
- $\forall k \in \mathbb{N}$ $\log_2{}^{k}(2^{\boxed{k}}) = 1$ (b)
- $\forall k \in \mathbb{N}$ $\log_*(2^{\boxed{k}}) = k$. (c)

a) clair. b) découle de a) en itérant la 2^{nde} relation.

c) Comme $\log_2{}^{k}(2^{\boxed{k}}) = 1$, néc. $\log_*(2^{\boxed{k}}) \leq k$.

Or $\forall i \in [0..k-1]$, $\underset{\substack{\uparrow \\ \text{en itérant (a)}}}{\log_2{}^{i}(2^{\boxed{k}}) =} 2^{\boxed{k-i}} \underset{\substack{\uparrow \\ \text{car } k-i \geqslant 1}}{\geqslant} 2 > 1$,

D'où $\log_*(2^{\boxed{k}}) = k$.

Pté Avec $I_0 = \{1\}$ $\forall k \in \mathbb{N}^*$ $I_k =]2^{\boxed{k-1}} + 1 .. 2^{\boxed{k}}]$

on a $\forall k \in \mathbb{N}, \forall n \in I_k$, $\log_*(n) = k$.

Preuve

- Si $n \in I_0$, $n = 1$ et $id(n) = 1 \leq 1$ donc $\log_*(n) = 0$.
- Si $k > 0$ et si $n \in [2^{\boxed{k-1}}+1 .. 2^{\boxed{k}}]$.

Comme $2^{\boxed{k-1}} < n \leq 2^{\boxed{k}}$, par croissance de $\log_*$ on a $k-1 = \log_*(2^{\boxed{k-1}}) \leq \log_*(n) \leq \log_*(2^{\boxed{k}}) = k$.

Or puisque $\log_2$ est st. croissant $\log_2{}^{k}(n) > \log_2{}^{k}(2^{\boxed{k}}) = 1$ donc $\log_*(n) \neq k$. D'où $\log_*(n) = k$.

Ex

$I_0 = \{1\}$

$I_1 = \{2\}$

$I_2 = \{3, 4\}$

$I_3 = [5 .. 2^4 = 16]$

$I_4 = [17 .. 2^{16} = 65\,536]$

$I_5 = [65537 .. 2^{65536}]$.

En pratique on n'ira pas plus loin.

Pté Si $n > 2$, $\log_*(\log_2(n)) = \log_*(n) - 1$

Preuve Si $\log_*(n) = k > 1$, $n \in I_k = [2^{\boxed{k-1}}+1 .. 2^{\boxed{k}}]$

On a alors $2^{2^{\boxed{k-2}}} < n \leq 2^{2^{\boxed{k-1}}}$, donc par croissance stricte de $\log_2$

on a $2^{\boxed{k-2}} < \log_2(n) \leq 2^{\boxed{k-1}}$ soit $\log_2(n) \in I_{k-1}$

donc $\log_*(\log_2(n)) = k-1 = \log_*(n) - 1$.

"c'est gonflé comme dupt"

UNION - FIND

[Papa] → Algorithms p 132 → 137.

Idée Gérer des partitions, notamment des classes d'équivalences, dans le cas où les classes ne font que grossir.

Type abstrait

PARTITION

CRÉER_PARTITION_VIDE ()	partition
CRÉER_CLASSE (P: partition, x: élm^t)	unité
TROUVER (P: partition, x: élm^t)	élm^t
UNION (P: partition, x: élm^t, y: élm^t)	unité

Implémenta° avec des arbres

↳ Une classe va être représentée par un arbre, d'arité quelconque, dont les nd seront les élm^t, et la racine un représentant canonique

↳ Pour "trouver" un élm^t, c-à-d pour identifier sa classe on donne ce rep. canonique. Il faut donc remonter à la racine. On a tout intérêt à minimiser la hauteur de ces arbres.

↳ Pour savoir comment faire l'union de deux classes avec une hauteur minimale, on a besoin de connaître la hauteur. Pour cela on ajoute à chaque noeud un rg qui doit dans un premier temps être compris comme la hauteur du sous-arbre engendré.

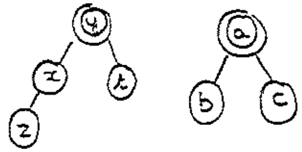
Rq une feuille est considérée de hauteur nulle.

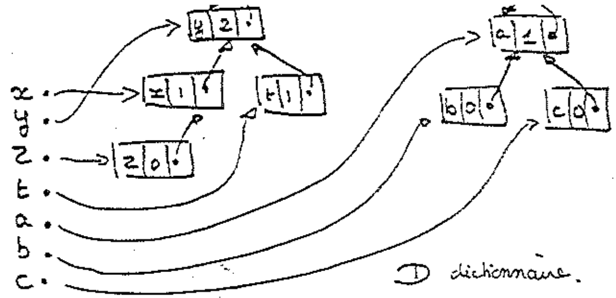
↳ Contrairem^t à d'habitude les nd vont ici stocker leur (unique) père, et non leurs (multiples) fils, puisqu'on va chercher à remonter l'arbre

1 noeud = 1 élm^t + son rg + son père.

↳ ⚠ Cependant il faut pouvoir étant donné un élm^t retrouver le nd qui correspond ce qui n'est pas évident avec cette structure ascendante. On utilise pour cela un dictionnaire.

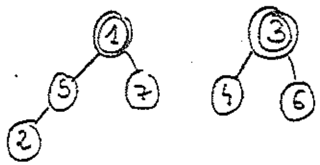
ex P = {{x, y, z, t}, {a, b, c}}





↳ A partir de maintenant on se place dans le cas où les élm^t sont les entiers de 1 à n, où n est fixé initialem^t.
L dictionnaire s'implémente alors simplem^t comme un tableau.

ex P = {{1, 2, 5, 7} {3, 4, 6}}



	1	2	3	4	5	6	7
Π	1	5	3	3	1	3	1
rg	2	0	1	0	1	0	0

↳ Dans ce cadre on implémente le type abstrait par les f° suivantes.

CRÉER PARTITION_VIDE (n)
- P.Π = tableau de taille n
- P.rg = ———
- retourner P

CRÉER CLASSE (P, x)
- P.Π [x] = x
- P.rg [x] = 0

TROUVER (P, x)
- t = x
- tant que P.Π[t] ≠ t
  - t ← P.Π[t]
- retourner t.

UNION (P, x, y)
- rx = TROUVER (P, x)
- ry = ——— (P, y)
- Si rx = ry alors fini (cas 1)
- Si P.rg [rx] < P.rg [ry] alors P.Π [rx] = ry ; fini (cas 2)
- Si P.rg [rx] > P.rg [ry] alors P.Π [ry] = rx ; fini (cas 2 bis)
- // si P.rg [rx] = P.rg [ry]
  - P.Π [rx] = ry
  - P.rg [ry] ← P.rg [ry] + 1. (cas 3).

UNION-FIND (SUITE 1)

Rq • x est une racine dans P ssi $P.\Pi[x] = x$.

• Dès qu'un nd arrête d'être racine son rang est fixé, et jamais plus il ne sera racine ; de plus on ne lui ajoutera pas de descendants.

• Un nd obtient son rang k par union de 2 classes de "rg" $k-1$.

• Ici le rg est la hauteur du sous-arbre engendré.

Pté ☆1 Si x n'est pas racine, alors ~~$P.rg[P.\Pi[x]] > P.rg[x]$~~ on notera $rg(\Pi(x)) > rg(x)$

☆2 Chaque nd de rang k a au moins 2^k élm^t dans l'arbre qu'il engendre

☆3 Avec $\underline{n_k = \text{nbr de nd de rg } k}$ $\quad n_k \leq \frac{n}{2^k}$ et $\forall x \; rg(x) \leq \log_2(n)$

Preuve ☆1 Un noeud x arrête d'être racine dans le cas 3 de UNION, moment auquel on lui affecte un nouveau père de rang un de plus. Après cela ni son rang ni son père ne changent, et le rang de son père ne peut que croître d'où ☆1

☆2 La pté est vraie au rang 0 car le noeud lui m̂ compte (et $2^0 = 1$). Supposons la propriété vraie au rang k. Si x est un noeud de rang $k+1$, c'est qu'il représente l'union de 2 classes de rg k (rq3) c-à-d + précisém^t de classes dont la racine est de rg k. Par HR chacune d'elle a au moins 2^k élm^t, donc le sous arbre engendré par x a au moins $2^k + 2^k = 2^{k+1}$ élm^t. On conclut à ☆2 par récurrence.

☆3 D'après la rq 1, lorsqu'on remonte de père en père le rang croît strictem^t. Chaque nd a donc au plus un ancêtre de rang k.
En comptant tous les descendants (au sens large) des nds de rang k, on compte alors d'après ☆2, au moins $n_k \times 2^k$ élm^t. Or il n'y en a que n.
D'où $n_k \times 2^k \leq n$. Par suite si $k > \log_2(n)$ ie si $2^k > n$ soit $\frac{n}{2^k}$, $n_k < 1$ donc $n_k = 0$.
Cela traduit bien qu'au plus le rg d'un nd est $\log_2(n)$.

Cplxité (a) • CRÉER_PARTITION_VIDE et CRÉER_CLASSE se font en $\underline{\text{tps constant}}$.
(b) • et donc UNION se font en $\underline{O(\log_2(n))}$

(a) clair (b) Corollaire direct du fait que le rang est borné par $\log_2(n)$ (☆3) et donc la hauteur des arbres (d'après rq 4) aussi.

Compression de chemins

Idée Si on calcule TROUVER pour un élém^t, pourquoi le laisser en bas dans l'arbre alors qu'on peut le remonter juste sous la racine.

TROUVER*(P, x)
  si $P.\Pi[x] \neq x$
  alors $P.\Pi[x]$ = TROUVER*($P.\Pi[x]$)
  retourner $P.\Pi[x]$

. TROUVER* est définie récursivem^t

. On notera rg* le rang obtenu en utilisant TROUVER* plutôt que TROUVER, et Π^* le père.

Rq • Pour tout élém^t x $\; rg^*(x) = rg(x)$
En faisant la m̂ suite d'opérations, avec TROUVER ou TROUVER* les élm^t finissent avec le m̂ rg. C'est leur père, et donc la forme des arbres qui change.

. TROUVER* n'agit ni sur les racines, ni sur leurs fils.

• UNION à l'inverse n'agit que sur les racines.

Pté ☆1' Si x n'est pas racine. $rg^*(\Pi^*(x)) > rg^*(x)$

☆2' Chaque $\underline{\text{racine}}$ de rg* k a au moins 2^k élm^t ds le ss arbre engendré

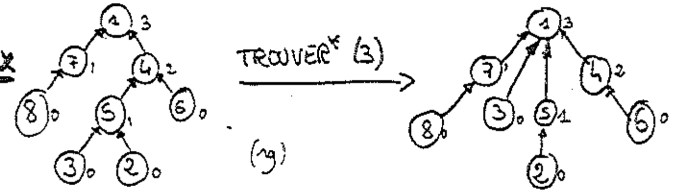
☆3' Avec n_k^* = nbr de nd de rg* k on a $n_k^* \leq \frac{n}{2^k}$ et $\forall x \; rg^*(x) \leq \log_2(n)$

Preuve ☆1' Le $rg^*(x)$ est le m̂ que $rg(x)$. $\Pi^*(x)$ peut être différent de $\Pi(x)$ si TROUVER*(x) a été appelé, auquel cas $\Pi(x)$ a été remplacé par la racine qui est néc de rang plus grand d'après ☆1. D'où $rg^*(\Pi^*(x)) = rg(\Pi^*(x))$
$\geq rg(\Pi(x))$
$> rg(x) = rg^*(x)$.

☆2 Ce n'est plus vrai si on n'est pas racine car l'action de TROUVER* a pu faire perdre des enfants à un nd, au profit de la racine.
Mais comme TROUVER* n'affecte pas les racines, ☆2 reste vrai pour elles.

☆3. Simplem^t parce que le rg = rg* donc $n_k = n_k^*$.

Ex



UNION-FIND (SUITE 2)

Analyse amortie

L'avantage de TROUVER* c'est qu'il profite de son exécution pour améliorer la structure, ce qui réduit la complexité des suivants.
On ne peut donc, pour prendre en compte cette amélioration, se restreindre à étudier la complexité d'un appel, on doit au contraire faire une analyse de la complexité amortie.

Pté Considérons une suite d'opérations qui appelle m fois TROUVER* sans compter les appels récursifs internes.
Sa complexité est en $O(m \log^*(n)) + O(n \log^*(n))$

On comprend cette complexité comme un coût de TROUVER* en $O(\log^*(n))$, à ceci près qu'on a un surcoût en $O(n \log^*(n))$, relatif à la taille de la structure.

Pour montrer cela on utilise un modèle budgétaire

-> Chaque étape de calcul coûte 1 \$

-> TROUVER* propose un forfait de $\log^*(n)+2$ étapes par appel, qu'on paye d'avance pour les m étapes avec un budget $B_1 = m \times (\log^*(n)+2)$

-> les opérations supplémentaires éventuellem^t nécessaires seront payées par les noeuds concernés, auxquels on aura globalem^t attribué un budget B_2.

. Comment répartir le budget sur les noeuds ?

Lorsqu'un noeud arrête d'être racine, il a atteint son rg définitif.
Si ce rang est dans I_k où $k>0$, on attribue $2^{\boxed{k}}$ \$ à ce noeud.

. Calculer B_2 Notons a_i l'ensemble des nd de rg i à la fin ; $n_i = \# a_i \le \frac{n}{2^i}$
A_k ——— de I_k ——— ; $N_k = \# A_k$.

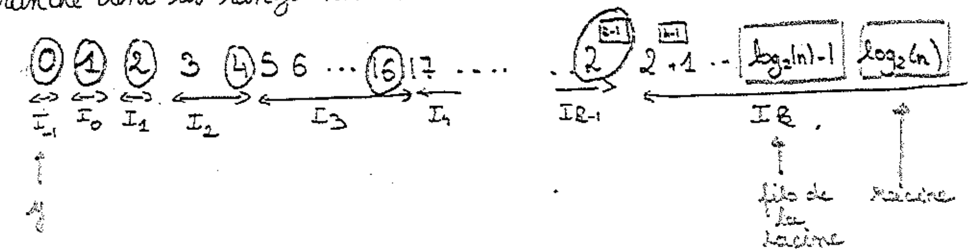
$$B_2 = \sum_{x \text{ nd}} \text{budget}(x) = \sum_{k=-1}^{\log_*(\log_2(n))} \sum_{x \in A_k} \text{budget}(x) = \sum_{x \in A_{-1}} 0 + \sum_{x \in A_0} 0 + \sum_{k=1}^{\log_*(n)-1} \sum_{x \in A_k} \underbrace{\text{budget}(x)}_{\le 2^{\boxed{k}}}$$

$$\le \sum_{k=1}^{\log_*(n)-1} N_k \times 2^{\boxed{k}} \quad \text{or } N_k = \sum_{i=2^{\boxed{k-1}}+1}^{2^{\boxed{k}}} n_i \le \sum_{i=2^{\boxed{k-1}}+1}^{+\infty} \frac{n}{2^i} = \frac{n}{2^{2^{\boxed{k-1}}}} \sum_{j=1}^{+\infty} \frac{1}{2^j} = \frac{n}{2^{\boxed{k}}} \times 1.$$

D'où finalem^t $B_2 \le \sum_{k=1} \frac{n}{2^{\boxed{k}}} \times 2^{\boxed{k}} \le n \log^*(n)$.

. Vérifier qu'on pourra bien payer toutes les étapes

Considérons l'appel de TROUVER*(y).
Il demande autant d'étapes de calcul qu'il y a de nd sur la branche remontant de y à la racine, branche le long de laquelle le rang croît strictem^t.
Le pire cas, celui de la branche la plus longue possible, est alors une branche dont les rangs sont :



On fait rentrer dans le forfait les calculs des noeuds dont le père a son rang dans un intervalle plus grand $\bigcirc$, et ceux pour le fils de la racine et la racine $\square$.
Il y a au plus - ie dans ce pire cas - $k+1$ noeuds du 1er type où $k = \log^*(\log_2(n)) = \log^*(n) - 1$. On rentre bien dans le forfait des $\log^*(n)+2$ calculs pour cet appel.
Il reste à s'assurer que les autres noeuds ont de quoi payer.
Ils ne sont pas racine et leur noeud est de rang dans un I_k avec $k>0$, ils ont donc bien reçu $2^{\boxed{k}}$ \$.
Comme à chaque fois qu'un noeud paye il change de père pour un père de rang strictem^t plus grand, quand un noeud n'a plus de budget son père est nec de rg dans un intervalle plus grand, il sera alors dispensé de payer ie rentrera dans le forfait.

La complexité est de même que le budget

$$\underbrace{O(m \log^*(n))}_{B_1} + \underbrace{O(n \log^*(n))}_{B_2}.$$

Il faut numéroter à partir de 1.

Développement : Tri par tas [Cormen]

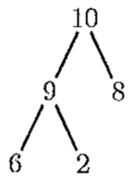
PIERRON Théo – LACOSTE Cyril

13 avril 2014

Définition Un tas est un arbre binaire dont tous les niveaux sont complètement remplis sauf éventuellement le dernier. De plus, toutes les feuilles doivent être le plus à gauche possible.

On peut le représenter par un tableau A et un entier *taille* tels que les éléments du tas soient dans $A[1 \dots taille]$. ↳ parce que l'arbre binaire est presque complet + à gauche.

Un tas max est un tas tel que la racine de tout sous-arbre est plus grande que les éléments du sous-arbre.



gauche $(i) = 2i$
droite $(i) = 2i+1$
père $(i) = \lfloor \frac{i}{2} \rfloor$

Prop: il est équilibré

FIGURE 1 – Tas max représenté par $[|10; 9; 8; 6; 2|]$

Soit un tas A est un indice i tels que les fils gauche et droit de i soient des tas max. On veut faire descendre $A[i]$ pour que l'arbre de racine i soit un tas max. On utilise pour ce faire l'algorithme suivant

Algorithme 1: Entasser(A, i, n)

Entrées : Un tas A et un entier i, $n \in \mathbb{N}^*$, n taille du sous-arbre.
Sorties : Une permutation de A tel que l'arbre de racine i soit un tas max

```
g := gauche(i)
d := droite(i)
max := i
si g ⩽ taille et A[g] > A[max] alors
    max := g
si d ⩽ taille et A[d] > A[max] alors
    max := d
si max ≠ i alors
    échanger A[i] et A[max]
    Entasser(A, max)
```

La complexité de Entasser(A, i) sur un noeud i de hauteur h est au pire $O(h)$.

Pour construire un tas max à partir d'un tableau A, on va appeler Entasser sur chaque élément du tableau. On remarque que les $\underbrace{\text{feuilles, ie les éléments de } A[\frac{|A|}{2}, \dots, |A|]}_{\text{gauche}(i) = 2i}$ sont déjà des

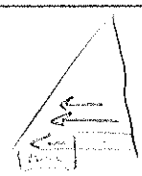
il faut le détailler

gauche $(i) = 2i$

tas max. Il suffit donc de considérer les éléments de $A[1, \ldots, \lfloor \frac{|A|}{2} \rfloor]$.

Algorithme 2: Construire(A)

Entrées : Un tableau A
Sorties : Une permutation de A correspondant à un tas max



taille := A
pour $i = \lfloor \frac{|A|}{2} \rfloor \ldots 1$ **faire**
  Entasser(A, i)

Quand on construit le tas, on appelle Entasser sur $\lfloor \frac{n}{2^{h+1}} \rfloor$ nœuds de hauteur h. La complexité est donc

bof

$$\sum_{i=0}^{\lfloor \log_2(n) \rfloor} \left\lfloor \frac{n}{2^{h+1}} \right\rfloor O(h) = O\left(n \sum_{i=0}^{\lfloor \log_2(n) \rfloor} \frac{h}{2^h}\right) = O(n)$$

cf [Cormen]

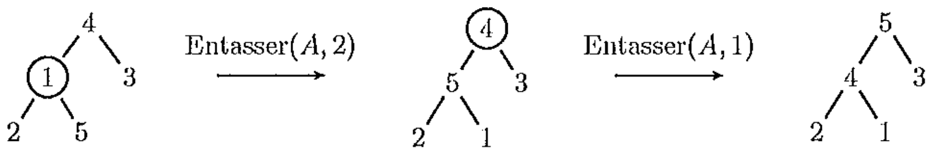


FIGURE 2 – Étapes de Construire([|4 ;1 ;3 ;2 ;5|])

On peut maintenant donner l'algorithme du tri par tas : on construit un tas puis on enlève la racine, on entasse, et on recommence.

Algorithme 3: Tri(A)

Entrées : Un tableau A
Sorties : Le tableau trié associé à A

Construire(A)
pour $i = |A| \ldots 2$ **faire**
  Échanger $A[1]$ et $A[i]$
  taille := *taille* − 1
  Entasser($A, 1$)   Entasser $(A[1, i], 1)$.

Le tri par tas a donc une complexité de $O(n) + n \times O(\log_2(n)) = O(n \ln(n))$ car la racine est de hauteur $\lfloor \log_2(n) \rfloor$.

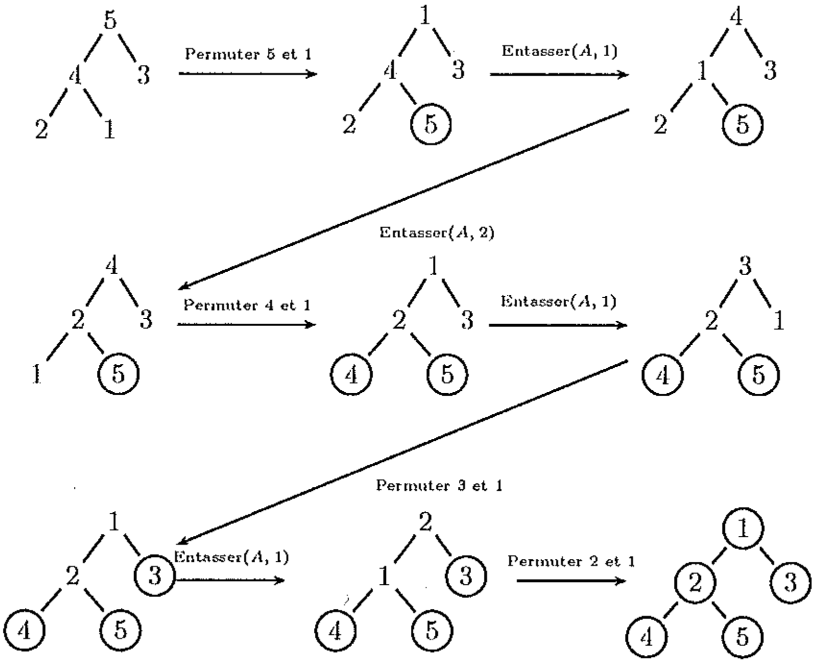


FIGURE 3 – Étapes de Tri([|4 ;1 ;3 ;2 ;5|])