Cours d'Analyse d'algorithmes. FLIN408

Chapitre 1. La problématique

Un problème peut être résolu par plusieurs algorithmes, encore plus de programmes (chaque algorithme pouvant être implémenté dans plusieurs langages et de différentes façons).

Comment comparer les diverses solutions ? Quels sont les critères importants pour comparer et donc choisir entre plusieurs algorithmes ou plusieurs programmes ?

Diverses propriétés ou paramètres physiques peuvent caractériser l'efficacité d'un programme (supposé juste) : place mémoire nécessaire, durée d'exécution, simplicité du code, ….

Dans ce cours (faute de temps et pour des raisons de simplification) nous allons nous limiter, aux deux premiers paramètres et nous intéresser plus aux aspects algorithmiques que programmation (dans la mesure du possible car il est évident que la réflexion autour de la mise en œuvre d'un algorithme à un impact parfois déterminant sur son efficacité).

Comment mesurer la place mémoire ? : le nombre de bits utilisés. Comment mesurer la durée d'exécution ? : le nombre d'unités de temps pour la durée d'exécution. Mais alors le temps d'exécution dépend de la donnée, du compilateur, de l'ordinateur, du langage utilisé … Bref la mesure devient difficilement utilisable. Il faut simplifier, modéliser. En effet, des énoncés du type : L'algorithme A implémenté dans le langage P sur l'ordinateur O, et exécuté sur la donnée D utilise k secondes et j bits de mémoire, sont d'une portée très limitée. Que se passe-t-il si l'on exécute sur la donnée D' ? puis si l'on change d'ordinateur ? Nous allons donc chercher des informations plus générales : l'algorithme A est toujours meilleur que l'algorithme B dès que D est grand.

En fait, à chaque problème, on peut en général associer une ou des **opérations élémentaires** : c'est-à-dire une (ou des) opération caractéristique ou élémentaire que l'on exécutera au mois autant de fois que les autres. Par exemple si on recherche un élément dans un tableau le nombre de comparaisons entre éléments sera l'opération élémentaire, si on multiplie deux matrices les opérations élémentaires seront les additions et les multiplications d'entiers, si l'on trie des éléments les opérations élémentaires seront les comparaisons entre éléments et les déplacements d'éléments etc.…

Axiome : **le temps d'exécution de l'algorithme est proportionnel au nombre d'exécutions de l'opération fondamentale (ou élémentaire).**
On se contentera donc à chaque fois après l'avoir déterminé, de compter le nombre de fois où cette opération est effectuée.

Exemple : la recherche d'un élément dans un tableau :

```
i = 1
tant que t[i] #x faire i=i+1 fin tq
retourner i
```

Remarque : on fait l'hypothèse que $x \in t$.

On remarque que le nombre de comparaisons dépend de la taille des données et de leur organisation (l'élément recherché est-il absent ou présent dans l'une des cases et s'il est présent dans laquelle ?). D'où la nécessité d'introduire la notion de fonctions qui prennent en paramètre la taille des données et donnent comme valeur le nombre d'opérations effectuées : soit le nombre moyen ou le nombre maximum. Le calcul du nombre moyen de comparaisons est souvent un problème très difficile (Quelle est la probabilité d'avoir la donnée d ?). Dans la suite nous limiterons ce problème en faisant l'hypothèse que toutes les données de même taille ont la même probabilité (distribution uniforme). Ainsi à chaque algorithme A, on essayera d'associer une fonction f_A de N dans N qui associe le nombre d'exécutions de l'opération élémentaire si l'on exécute A sur une donnée de taille n (l'unité de taille pourra être le bit, l'entier, le flottant, …).

Sur l'exemple, il est facile de voir que le nombre de comparaisons d'éléments est au mieux 1 (l'élément x à la première case), au pire n (l'élément x à la dernière case), en moyenne (n+1)/2 (i comparaisons si l'élément x est dans la case i, avec une probabilité 1/n). Pour traiter le cas où $x \notin t$: on ajoute le test i<n.

Exploitation de la fonction :

En fait les valeurs de ces fonctions ne sont évidemment pas égales à la durée d'exécution du programme (la mesure qui intéresse l'utilisateur) par contre elles ont de bonnes chances de représenter une valeur proportionnelle à la durée d'exécution. C'est-à-dire ici proportionnelle à n. Ainsi si pour n=100, l'algorithme s'exécute en 1 ms, il mettra environ 10 ms pour n=1000.

Rappel sur les ordres de grandeurs et sur les relations de comparaison :

O et Θ. Soit f et g, 2 fonctions de N dans N.
$f = O(g)$ si $\exists\, c \in R^{*+}\ \exists\, n_0 \in N$ tel que $\forall n > n_0$, $c*g(n) \geq f(n)$. Dans ce cas f croit moins vite que g.
$f = \Theta(g)$ si $\exists$ c et d $\in R^{*+}\ \exists\, n_0 \in N$ tel que $\forall n > n_0$, $d*g(n) \geq f(n) \geq c*g(n)$. Dans ce cas f et g croissent à la même vitesse.
Si on reprend l'exemple de la recherche, le nombre de comparaisons d'éléments en moyenne (n+1)/2 donc en Θ(n). On peut donc en déduire que la durée de l'exécution de l'algorithme sera proportionnelle à n. Ainsi si en moyenne on met 1ms pour chercher un élément dans un tableau de taille 100, on mettra environ 1s pour chercher un élément dans un tableau de taille 100000.
Exercice : $f(n)=2n^2+3n$ et $g(n)= n^2+7n$. On a $f=\Theta(g)$ en prenant c=1, d=3, $n_0=5$.

Un exemple plus complet :

Exemple : la recherche du plus grand élément dans un tableau (retourne l'élément le plus grand du tableau).

```
m = t[1] ;
pour i de 2 à n faire
    si t[i] > m alors m = t[i] fin si
fin pour ;
retourner m ;
```

L'algorithme nécessite (n-1) comparaisons d'éléments : une comparaison à chaque passage dans la boucle. Ce résultat est a priori optimal car il y a (n-1) éléments qui ne sont pas optimaux et pour le savoir il faut comparer chacun avec un élément plus grand. Si l'on désire calculer le nombre d'affectations, le problème est plus délicat car le résultat ne dépend pas que du nombre d'éléments. Pour l'analyse on suppose que les éléments sont tous disjoints et que toutes les positions pour les éléments dans le tableau sont équiprobables (ainsi si le tableau contient les éléments 2,3 et 5 alors il y a 6 tableaux possibles :

2	3	5	Nombre d'affectations égal à 3
2	5	3	Nombre d'affectations égal à 2
3	2	5	Nombre d'affectations égal à 2
3	5	2	Nombre d'affectations égal à 2
5	2	3	Nombre d'affectations égal à 1
5	3	2	Nombre d'affectations égal à 1

Seul l'ordre des éléments est important ainsi on peut limiter l'analyse au cas où le tableau ne contient que les entiers de 1 à n (on peut donc se limiter dans l'analyse aux permutations des entiers de 1 à n).

Et chacune apparaît avec la même probabilité : 1/6. Dans le cas général, il y a donc n ! possibilités (le nombre de permutations), chacune ayant la probabilité 1/n ! d'apparaître. On remarque donc que le nombre minimum d'affectations est égal à 1 (avec la probabilité 1/n, le plus grand élément est dans la case 1), le nombre maximum est égal à n (avec la probabilité 1/n !, les éléments sont triés par ordre croissant) et le nombre moyen est égal H_n (à l'itération i, il y a une probabilité 1/i d'avoir une affectation) c'est à dire $\approx \log(n)$.

Autre exemple : la recherche d'un élément X dans un tableau trié t par le recherche dichotomique.

```
i = 1 ; j = n ;
tant que i ≠ j faire
    aux = (i+j) div 2 ;
    si X > t[aux] alors i = aux + 1
                  sinon j = aux
    fin si
fin tq
si X=t[i] alors retourner i
          sinon «  X n'est pas dans le tableau »
fin si ;
```

Pour simplifier supposons que $n=2^k$. Le nombre de comparaisons est égal à k, c'est-à-dire $\log_2(n)$, si l'on ne compte pas la comparaison extérieure à la boucle tant que. Ainsi une trentaine de comparaisons suffisent pour chercher un élément dans un tableau contenant 10^9 éléments. Ce résultat est à comparer à celui de la recherche séquentielle (de l'ordre de n comparaisons dans le pire des cas). Améliorer la recherche séquentielle en utilisant le fait que les éléments sont triés ne permet de gagner qu'un facteur 2 en moyenne.

Peut-on améliorer la recherche dichotomique dans le pire des cas ? A priori non car il y a n conclusions possibles et donc il est nécessaire d'avoir n exécutions possibles et chaque comparaison ne peut multiplier au plus par deux le nombre d'exécutions différentes.

De façon générale : si on a un algorithme basé sur les comparaisons avec p exécutions différentes nécessaires alors le nombre de comparaisons au pire est supérieur à $\log_2(p)$. Une façon de se convaincre de ce résultat est de construire l'arbre de décision. L'arbre de décision est un arbre binaire qui représente toutes les exécutions possibles d'une certaine taille. Les feuilles de l'arbre indiquent les résultats des différentes exécutions (remarque : deux exécutions différentes peuvent donner le meme résultat). Les noeuds interne correspond à la comparaison de deux éléments effectuées par l'algorithme. Si la comparaison est VRAI (resp. FAUX) alors le sous arbre gauche (resp. droit) représente la suite de l'exécution.

Le tableau ci-dessous permet d'avoir une idée de la durée d'exécution d'un algorithme en fonction de l'ordre de grandeur de son exécution. Si la complexité est inférieure à log (n) alors il n'y a aucune contrainte sur la taille des données, si la complexité est inférieure à n, seules des données de taille très grande peuvent poser des problèmes (données codées sur quelques dizaines de bits), si la complexité est inférieure à n^k avec k>1 seules les données de taille moyenne peuvent être traitées, si la complexité est exponentielle seules les données de petite taille (quelques dizaines) peuvent être traitées.

Complexité de l'algorithme	log (n)	$\sqrt{n}$	n	n log(n)	n^2	2^n
1	1 µs	1 µs	1 µs	1 µs	1 µs	1 µs
1000	10 µs	32 µs	1 ms	10 ms	1 s	∞
10^6	20 µs	1 ms	1 s	20 s	> 1 j	∞
10^9	30 µs	32 ms	18 mn	≈10 h	>20 siècles	∞

Estimation des temps d'exécution d'un algorithme en fonction de sa complexité (hypothèse où un problème de taille 1 est effectué en 1 microseconde).

Chapitre 2. Comment calculer la complexité d'un algorithme ? : Quelques règles.

C'est un problème complexe (en fait il est indécidable, notion qui sera vue en Master). Prenons par exemple la fonction suivante f(1)=1, f(n)=n/2 si n est pair, f(n)=3n+1 sinon. On conjecture que cette fonction est définie pour tout n et que sa valeur est 1 pour tout n. Donc à l'heure actuelle personne ne connait la complexité du calcul de cette fonction (en utilisant le programme récursif induit par la définition). On ne sait même pas si elle boucle ou non.

Soit P(X) : le nombre de fois que l'on exécute l'opération élémentaire sur la structure X. Dans la suite nous allons voir quelques règles (malheureusement non complètes) qui permettront souvent de calculer P(X).

X = X1 ; X2 alors P(X) = P(X1) + P(X2).

X = si C alors X1 sinon X2 alors $P(X) \leq P(C) + \max(P(X1), P(X2))$

Et si p(C) est la probabilité que C soit vrai alors

$$moy(P(X)) = p(C)moy(P(X1)) + (1-p(C)),moy(P(X2))$$

Si X est une boucle pour i allant de a à b faire Xi alors

$$P(X) = \Sigma P(Xi)$$

Lors d'un appel à une fonction avec comme paramètre une donnée d, l'on ajoute le nombre d'exécutions de l'opération élémentaire effectuées pendant l'appel. Si l'appel est récursif alors nous sommes confrontés à la résolution d'une équation de récurrence.

Exemple 1 : le calcul de n !

```
fact(n)
si n=0 alors retourner 1
        sinon retourner n*fact(n-1)
fin
```

Soit t_n le nombre de multiplications pour calculer fact(n) (c'est évidemment l'opération fondamentale pour le calcul de n !).

On a $t_0 = 0$ et pour n>0 $t_n = 1 + t_{n-1}$.

La résolution est immédiate et la solution est $t_n = n$.

Exemple 2 : le calcul de Fibonacci de n

```
fibo(n)
si n<2 alors retourner n
        sinon retourner fibo(n-1)+fibo(n-2)
fin
```

Soit t_n le nombre d'additions pour calculer fibo(n) (c'est évidemment l'opération fondamentale pour le calcul de Fibonacci de n).

On a $t_0 = t_1 = 0$ et pour n>1 $t_n = 1 + t_{n-1} + t_{n-2}$.

La résolution est moins triviale que pour la formule précédente mais elle est faisable : $t_n = \Theta(\rho^n)$ où ρ est le nombre d'or $((1+\sqrt{5})/2)$.

En pratique dans de nombreux cas on ne sait pas résoudre l'équation de récurrence et l'on se contente de l'encadrer ou de la majorer.

Exemple 3 : le calcul de mystère de n

```
mystere (n)
si n<2 alors retourner 1
        sinon retourner mystere(n-1)+mystere(n div 2)+1
fin
```

Soit t_n le nombre d'additions pour calculer mystere(n) (c'est évidemment l'opération fondamentale pour le calcul de mystère de n).
On a $t_0 = t_1 = 0$ et pour n>1 $t_n = 2 + t_{n-1} + t_{n \text{ div } 2}$.
Le résultat sera compris entre les 2 résultats précédents (on peut évidemment améliorer cet encadrement).

Rappel sur la résolution des équations de récurrence : équation linéaire d'ordre 1 et d'ordre 2, diviser pour régner.

Quelques exemples de résolution simple :

Exercice 1**.** calcul du nombre de nœuds d'un arbre binaire complet de hauteur n avec 2 stratégies différentes : niveau par niveau ou une racine plus 2 arbres binaires parfait de hauteur n-1 :
$u_0 = 1$
$u_n = u_{n-1} + 2^n$ si n>0

ou

$u_0 = 1$
$u_n = 2 * u_{n-1} + 1$ si n>0

Ce sont des équations de récurrence linéaire d'ordre 1 : résolution par substitution ou par récurrence si l'on a une idée de la solution.

Exercice 2 :
$u_1 = c$
$u_n = a * u_{n/2} + b$ si n>0

Indication : on pose $n=2^k$ et l'on se ramène à une équation d'ordre 1.

Exercice 3 :
résolution de $u_n = a\, u_{n-1} + b\, u_{n-2}$ avec $(u_0, u_1, a, b) = (1, 2, 2, 3)$ et $(2, 4, 4, 4)$
et de $u_n = a\, u_{n-1} + b\, u_{n-2} + 4$ avec $(u_0, u_1, a, b) = (1, 3, 2, 3)$.

Indication : on calcule les racine de $P(x) = x^2 - ax - b$. Si les 2 racines x_1 et x_2 sont différentes alors la solution est égal à $\alpha\, x_1^n + \beta\, x_2^n$ sinon à $(\alpha\, n + \beta)\, x_1^n$. Les valeurs initiales de u permettent de calculer α et β.

Chapitre 3. Les Tris

Problème classique (par exemple pour faire des recherches dichotomiques qui sont optimales) :

10	12	8	7	20	3	6

3	6	7	8	10	12	20

On supposera toujours que les éléments sont disjoints pour effectuer l'analyse. Comme on utilise des algorithmes basés sur les comparaisons, on peut supposer que le tableau ne comporte que des éléments de 1 à n.. Ainsi si l'on part du tableau :

5	6	4	3	7	1	2

On aura la même exécution pour arriver au tableau trié que pour le tableau précédent (l'élément qui était au départ en case 6 se retrouvera en case 1, celui en case 4 se retrouvera en case 3 etc …). Ce qui compte c'est l'ordre des éléments pas leur valeur. Sans perte de généralités dans l'analyse, on peut supposer que le tableau de départ est donc une permutation de n éléments et deux permutations induisent donc deux exécutions différentes. Ainsi si chaque permutation est équiprobable (chacune ayant une probabilité 1/n !), il y a au moins $\log_2(n\,!)$ comparaisons nécessaires dans le pire des cas soit une complexité dans le pire des cas en $\Theta(n \log(n))$ au mieux.

Tri par sélection ordinaire :

Stratégie : pour trier n éléments :
Si n>1 alors sélection du plus grand élément (avec n-1 comparaisons)
le mettre à la case n (grâce à un échange)
continuer avec les (n-1) éléments

Une analyse permet d'avoir le nombre de comparaisons en $\Theta(n^2)$ au pire et le nombre d'échanges en $\Theta(n)$.

En effet, si t_n est le nombre de comparaisons on a $t_n = (n-1) + t_{n-1}$ si n>1 et donc t_n est en $\Theta(n^2)$. Dans la formule le (n-1) correspond au nombre de comparaisons pour trouver le plus grand élément et le t_{n-1} à l'appel récursif pour trier (n-1) éléments.

Le tri bulle est une variante qui sera vue en TD.

Tri par insertion :

A l'étape i les éléments dans les cases de 1 à i sont triés. Le but est de mettre l'élément de la case (i+1) à sa place :

- x = t[i+1]
- on cherche la case j telle que $t[j] \geq x$ et si $j \neq 1$ $x > t[j-1]$

- on décale les éléments de la case i à la case j (en décrémentant) de un cran vers la droite
- on met x à la case j

L'analyse montre que si j est cherché par dichotomie alors le nombre de comparaisons est au pire en $\Theta(n \log(n))$ par contre le nombre d'affectations est en $\Theta(n^2)$.

En effet la recherche de la case j où insérer x peut se faire avec $\log_2(n)$ comparaisons (en adaptant la recherche dichotomique). Le nombre de décalages pour insérer x est en moyenne égal à n/2.

Notons l'importance de l'opération fondamentale et la non trivialité de son choix qui était pour l'algorithme précédent le nombre de comparaisons et qui devient le nombre de déplacements d'éléments.

Tri fusion :

On divise en deux parties égales (ou quasi égales) l'ensemble des éléments à trier on trie chacune des parties et on fusionne les 2 parties triés.

Pour fusionner deux listes triées et obtenir une seule liste triée il faut au pire $n_1 + n_2 - 1$ comparaisons si n_1 et n_2 sont les longueurs des 2 listes. Ici on obtient donc comme formule de récurrence si t_n est le nombre maximum de comparaisons :

$t_0 = t_1 = 0$ et pour $n>1$ $t_n = 2\, t_{n \text{ div } 2}.+ n-1$. Soit t_n est en $\Theta(n \log(n)$

Cet algorithme est donc optimal en ordre de grandeur pour trier n éléments.

Tri rapide : l'étude de ce tri sera approfondi en travaux dirigés.

Pour trier un ensemble S de n éléments avec n>1 :

- on choisit un élément x (appelé le pivot) de S (a priori au hasard)
- on construit l'ensemble S1 des éléments de S plus petits que x
- on construit l'ensemble S2 des éléments de S plus grands que x
- tri(S) = tri(S1), x, tri(S2)

Remarque : pour construire les ensembles S1 et S2, il faut (n-1) comparaisons, chaque élément étant comparé au pivot. Si x est choisit au hasard, il a autant de chance d'être le $p^{\text{ième}}$ élément pour tout p de 1 à n c'est-à-dire avec une probabilité de 1/n pour chaque valeur. Dans ce cas S1 contient (p-1) éléments et S2 (n-p) éléments. Ainsi si t_n est le nombre moyen de comparaisons on a la formule suivante :

$t_0 = t_1 = 0$ et si $n>1$, $t_n = 1/n\ \Sigma_{p=1..n}\ (t_{p-1} + t_{n-p}) + n-1$

La résolution donne un ordre de grandeur en $\Theta(n \log(n))$. Dans le pire des cas, le pivot est toujours le plus petit élément ou le plus grand et la complexité devient du $\Theta(n^2)$.

Tri par tas :

C'est une sorte de tri par sélection. En fait dans un tri par sélection on doit trouver l'élément le plus petit ou le plus grand à chaque itération. L'idée d'utiliser une structure qui permet de faire cette opération rapidement. Il s'agit du tas : sa construction se fait en temps linéaire en n, on trouve le minimum en temps constant, on retire le minimum du tas en temps logarithmique. Le tri consiste donc à rechercher et à retirer n fois le minimum du tas dans un temps qui est donc en O(n log(n)).

Définition : un arbre partiellement ordonné est un arbre étiquetés par des éléments d'un ensemble muni d'un ordre total et tel que l'élément contenu dans tout nœud est inférieur ou égal aux éléments contenus dans les fils de ce nœud.

Un arbre parfait est un arbre binaire dont toutes les feuilles sont situées sur au plus deux niveaux, au plus seul le dernier niveau n'est pas saturé, et sur ce dernier niveau les feuilles sont calées le plus à gauche possible.

Un tas est un arbre parfait partiellement ordonné. Un tas comportant n nœuds est de hauteur $\lfloor \log_2 n \rfloor$.

Dans un tas le minimum est forcément à la racine.

Comme un tas est un arbre parfait, on peut facilement le représenter par un tableau dont les entrées sont numérotés de 1 à n où n est la taille du tas. La racine du tas est dans la case 1, les fils de l'élément de la case i sont dans les cases 2i et 2i+1 (si ces valeurs sont inférieures ou égales à n), le père de l'élément à la case i ($i \neq 1$) est dans la case i div 2.

Avec cette représentation on peut facilement écrire la procédure d'ajout d'un élément dans un tas : Si le tas contient p éléments avant l'ajout, on incrémente p, on met le nouveau élément dans la case p (pour conserver un arbre parfait) et on fait remonter l'élément vers la racine tant qu'il est de valeur inférieur à son père.

On obtient le code suivant :

```
Ajouter (t, p, x) :

p :=p + 1 ;
t[p] := x ;
i := p ;
tant que i>1 et t[i]<t[i div 2] faire
        echanget (t[i],t[i div 2]) ;
        i := i div 2
fin tq
```

Pour supprimer le minimum, on met l'élément y de la case p à la case 1 (pour conserver un arbre parfait), on décrémente la valeur de p et on fait descendre l'élément y à sa place pour obtenir un arbre partiellement ordonné : si l'un des deux fils a une valeur plus petite, on échange y avec le fils ayant la plus petite valeur.

On obtient le code suivant :

```
SupprimerMin (t, p) ;
```

```
t[1] := t[p] ;
p := p-1 ;
i := 1 ;
tant que p div 2  ≥ i faire
            j := 2*i ;
            si j≠p et t[j+1]<t[j] alors  j := j+1 fin si
            si t[i]>t[j] alors echanger (t[i],t[j]) ; i :=j
                      sinon i :=p
            fin si ;
fin tq ;
```

La construction du tas se fait simplement par

```
p := 1 ;
tant que p <> taille(t) faire
        ajouter (t,p,t[p+1])
fin tq ;
```

La complexité de cette construction est dans le pire des cas en Θ(n log(n).

Chapitre 4. Les algorithmes de recherche

On a vu au premier chapitre un algorithme de recherche optimal : la recherche dichotomique. Cet algorithme nécessite un précalcul pour trier les données et donc son utilisation peut poser des problèmes sur des données très dynamiques (ajout et suppression de données très fréquentes). Les algorithmes suivants permettent de résister à ces opérations ou de gérer des données non ordonnées de façon efficace.

Arbres binaires de recherche :

Un arbre binaire de recherche est un arbre étiqueté (les nœuds ont une valeur). Tous les nœuds du sous-arbre gauche ont une valeur inférieure ou égale à celle de la racine et ceux du sous-arbre droit ont une valeur strictement supérieure à celle de la racine. Cette organisation permet d'avoir des ajouts d'éléments, des suppressions d'éléments et des recherches d'éléments en un temps proportionnel à la profondeur moyenne des nœuds et cette valeur est en $\Theta(\log n)$.

Donc pour rechercher un élément, on le compare à la racine. S'il est différent de la valeur à la racine, on continue sur le sous-arbre gauche (s'il est inférieur) ou sur le sous-arbre droite (s'il est supérieur).

Une stratégie simple pour ajouter un nœud consiste à le rechercher et dès que l'on trouve l'endroit ou théoriquement il devrait se trouver on ajoute une feuille avec sa valeur. Pour la suppression même stratégie et quand on trouve l'élément :

- on le supprime simplement si c'est une feuille,
- on le remplace par son sous-arbre droit si le sous-arbre gauche est vide,
- on le remplace par le plus grand élément du sous-arbre gauche (qui est supprimé du sous-arbre gauche ce qui ne pose aucun problème puisque c'est forcément une feuille).

Malheureusement dans le pire des cas la profondeur de l'arbre peut être en $\Theta(n)$. L'idée est donc de définir des stratégie d'ajouts et de suppressions qui garantissent de conserver une profondeur logarithmique.

Arbres équilibrés

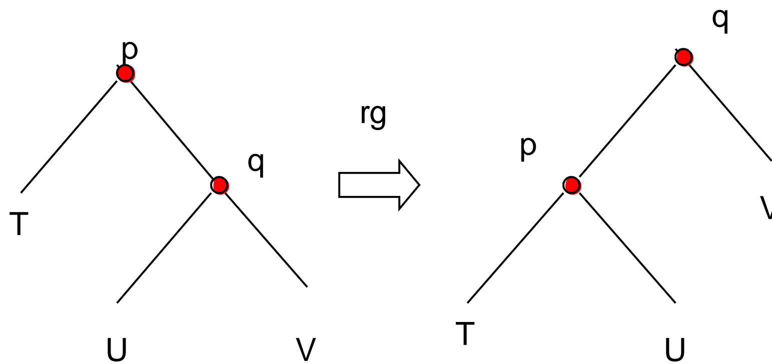
Une façon simple de conserver des ABR de profondeur logarithmique est de garantir que le déséquilibre (différence de hauteur entre le sous-arbre gauche et le sous arbre droit) reste en tout nœud inférieur à 1 en valeur absolue. Lors d'une ajout ou une suppression dans un ABR le déséquilibre peut devenir 2 dans ce cas on fait une rotation qui permet de corriger ce déséquilibre. Il existe 4 types de rotations :

Ci-dessous la rotation gauche, rg, (définie si le sous-arbre droit est non vide). La rotation droite, rd, est la transformation inverse (celle qui permet de passer de l'arbre à droite à l'arbre de gauche). La rotation gauche-droite, rgd, consiste à faire une rotation gauche dans le sous arbre gauche puis une rotation droite. La rotation droite-gauche consiste à faire une rotation droite dans le sous arbre droit puis une rotation gauche.

On peut remarquer que les rotations sont des transformations internes aux ABR (la rotation d'un ABR est un ABR). De plus pour la rotation gauche on remarque que le niveau des

nœuds dans V diminue, reste le même pour U et augmente pour T. Une adjonction dans V qui déséquilibre l'ABR (déséquilibre de –2) peut ainsi être complétée par une rotation gauche pour corrigée ce déséquilibre.

La rotation gauche :



On appelle donc les AVL des ABR qui conservent la propriété d'avoir un déséquilibre de 1, 0 ou -1 en tout nœud grâce à l'exécution de la rotations adaptée lors d'adjonction ou de suppression. Ainsi si le déséquilibre en un nœud devient –2 (resp. 2) alors que le déséquilibre est –1 (resp. 1) pour le sous-arbre droit (resp. sous-arbre gauche) on exécute une rotation gauche (resp. droite). Si le déséquilibre en un nœud devient –2 (resp. 2) alors que le déséquilibre est 1 (resp.-1) pour le sous-arbre droit (resp. sous-arbre gauche) on exécute une rotation droite gauche (resp. gauche droite). On peut montrer qu'au plus une seule rotation est nécessaire lors d'un ajout ou de la suppression d'un élément pour conserver un arbre équilibré.

Hachage

Les méthodes de hachage permettent de stocker des éléments en calculant l'endroit où l'on va les placer en mémoire. Cette méthode permet de rechercher rapidement les éléments. Plus précisément si on veut stocker x, on calcule la valeur de h(x) qui donne une adresse (h est donc une fonction qui donne une valeur entière).

Par exemple si l'on veut stocker des prénoms, on peut faire la somme des rangs des lettres (le rang de a est 1, celui de b est 2 etc …) plus le nombre de lettres pour avoir un entier. Si on n'a que m places mémoires on fait tous les calculs modulo m. Ainsi si m=11, on a h(bob)=22=0, h(serge)=19+5+18+7+5+5=8+5+7+7+10=2+3-1=4

Si on ne connaît pas a priori les éléments que l'on doit stocker il est impossible de trouver une fonction injective. Le mieux que l'on puisse espérer que la probabilité que h(x)=h(y) si x#y soit la plus faible possible (c'est à dire que h soit uniforme proba(h(x)=i)=1/m).

Dans tous les cas, il existe des collisions c'est-à-dire des valeurs de x et y telles que $h(x)=h(y)$, c'est-à-dire qui théoriquement devraient être stockées au même endroit. Ces collisions arrivent vite (par exemple même si l'on suppose que les dates d'anniversaires sont équiprobables dès qu'il y a 23 personnes la probabilité d'en avoir 2 nées le même jour est supérieure à 1/2). Heureusement elles sont assez rares, ce qui rend efficace la méthode du hachage.

Les méthodes se distinguent par la méthode de résoudre les collisions.

Méthode de chainage séparé : l'entrée i de la mémoire est en fait une liste qui contient tous les éléments x tels que $h(x)=i$. Le coût de l'ajout est proportionnel à la longueur de la liste (pour éviter l'ajout d'élément déjà présent). Le coût de la recherche dépend aussi de la longueur de la liste (si l'élément n'est pas présent) et de sa position dans la liste si l'élément est présent. Avec cette méthode le coût d'une recherche positive (resp. négative) est inférieure (resp. égale) à $1+\alpha/2$ (resp. α) où α est le taux de remplissage de la mémoire (n/m où n est le nombre d'éléments stockés et m le nombre de cases du tableau, ici c'est le nombre de listes).

Les autres méthodes même si elles sont moins bonnes que la précédentes ont toutes la particularité d'être de complexité constante (et elles n'ont de sens que si $\alpha < 1$).

Méthode de hachage coaslescent : En cas de place occupée on cherche une place libre dans la mémoire (en général en partant de la fin) pour mettre l'élément et on laisse une information (pointeur) à l'endroit où on aurait du le mettre pour le retrouver facilement.

Méthode avec calcul en cas de collision : on dispose d'une suite de fonctions h_i , on calcule $h_1(x)$ si la place est occupé on calcule $h_2(x)$ etc … jusqu'à trouver une place libre. Dans le cas où i#j implique $h_i(x)$#$h_j(x)$, m fonctions suffisent.

C'est le cas pour le hachage linéaire $h_i(x)=h(x)+i-1$ ou le double hachage $h_i(x)=h(x)+(i-1)d(x)$ avec h(x) et d(x) deux fonctions de hachage uniforme (m doit etre premier pour le double hachage et d(x) différent de 0). Ces méthodes ont l'inconvénient de créer des blocs.

Une méthode efficace pour résoudre ce problème est de prendre des fonctions h_i indépendantes entre elles. Dans ce cas pour être sur de pouvoir faire l'ajout dans tous les cas si la mémoire n'est pas saturée, il faut une infinité de fonctions de hachage.