Algorithmique Avancée

Michèle Soria

Michele.Soria@lip6.fr

Master Informatique M1-STL

http://www-master.ufr-info-p6.jussieu.fr/2012
http://www-master.ufr-info-p6.jussieu.fr/2012/algav

Année 2012-2013
Transparents de cours – Partie 1

Organisation

- **L'équipe pédagogique** :
  - **Cours** : Michèle Soria (mercredi 10h45, amphi Astier)
  - **TD-TME** : Antoine Genitrini, Maryse Pelletier, Philippe Trébuchet, Binh-Minh Bui Xuan.
- **L'organisation** :
  - 1er cours le mercredi 19 septembre
    TDs : Lundi, mardi, mercredi, à partir du **24 septembre**.
  - Session 1
    - ecrit réparti 1 (E1) : 05–09 novembre 2012
    - fin des enseignements le 14 décembre 2012
    - ecrit réparti 2 (E2) : 07–11 Janvier 2013
    - note Session 1 = 0.2 E1 + 0.2 D + 0.6 E2
    - D = devoir-projet : distribué fin octobre, soutenu et rendu semaine du 10 décembre
  - Session 2
    - examen Session 2 (SS) : 13–20 mai 2013
    - note Session 2 = SS

Plan du cours

Objectifs : Complexité des algorithmes $\rightarrow$ Comparer, Optimiser

- **Structures Arborescentes : Files de priorités**
  - Files Binomiales et Files de Fibonacci
  - Coût amorti et Coût moyen
- **Structures Arborescentes pour la Recherche**
  - Arbres de Recherche équilibrés
  - Recherche externe
- **Méthodes de Hachage**
  - Hachage interne et externe
  - Hachage universel
  - Hachage cryptographique
- **Algorithmes de compression**
  - Compression statistique, statique et dynamique
  - Compression par dictionnaires
- **Algorithmique géométrique**
  - Enveloppe convexe, statique et dynamique
  - Analyse et implantation

Bibliographie

- T. Cormen, C. Leiserson, R. Rivest, C. Stein
  Introduction à l'algorithmique,
- C. Froidevaux, M-C. Gaudel, M. Soria
  Types de données et algorithmes
- D. Beauquier, J. Berstel, P. Chrétienne
  Eléments d'algorithmique
- D. Salomon
  Data Compression : The Complete Reference
- M. Nelson
  La Compression de données : texte, images, sons
- M. Crochemore, C. Hancart, T. Lecroq
  Algorithmique du texte

CHAPITRE 0 : INTRODUCTION COMPLEXITE

- Théorie de la complexité et classification de problèmes
- Problèmes pôlynomiaux : tri, recherche, géométrie, arithmétique, ...
- Analyse des *algorithmes* ; opération(s) *fondamentale(s)*
- Coût (temps, espace) fonction de la taille des données
- Cas pire, cas moyen, coût amorti
- Ordre de grandeur

CHAPITRE 1 : FILES de PRIORITÉ

CHAPITRE 1 : Files de Priorités : binomiales et Fibonacci

- Opérations sur les files de priorités
- Arbres binomiaux : définition et propriétés
- Files binomiales : définition et propriétés
- Union de 2 files binomiales en temps logarithmique
- Autres opérations sur les files binomiales
- Analyse en Coût amorti

Opérations sur les files de priorités

- **Ensemble d'éléments**
  - Chaque élément identifié par une clé
  - Ordre total sur les clés
- **Opérations**
  - Ajouter un élément
  - Supprimer l'élément de clé minimale
  - Union de 2 files de priorités
  - *Construction*
  - *Modification d'une clé*

Représentations et Efficacité

Nombre de comparaisons dans le pire des cas

	Liste triée	Tas (Heap)	File Binomiale
Supp Min (n)	$O(1)$	$O(\log n)$	$O(\log n)$
Ajout (n)	$O(n)$	$O(\log n)$	$O(\log n)$
Construction (n)	$O(n^2)$	$O(n)$	$O(n)$
Union (n, m)	$O(n+m)$	$O(n+m)$	$O(\log(n+m))$

Applications des Files de priorité

- Tri heapsort
- Sur les graphes
  - plus court chemin à partir d'une source (Dijkstra)
  - plus court chemin entre tous les couples de sommets (Johnson)
  - arbre couvrant minimal (Prim)
- Interclassement de listes triées
- Code de Huffmann (compression)

Arbre binomial- Définition

Un arbre binomial pour chaque entier positif.

Définition par récurrence

- B_0 est l'arbre réduit à un seul nœud,
- Étant donnés 2 arbres binomiaux B_k, on obtient B_{k+1} en faisant de l'un des B_k le premier fils à la racine de l'autre B_k.

Exemples : dessiner B_0, B_1, B_2, B_3, B_4

Arbre binomial - Propriétés

Propriétés de B_k, $(k \geq 0)$

1. B_k a 2^k nœuds
2. B_k a $2^k - 1$ arêtes
3. B_k a hauteur k
4. Le degré à la racine est k
5. Le nombre de nœuds à profondeur i est $\binom{k}{i}$
6. La forêt à la racine de B_k est $< B_{k-1}, B_{k-2}, \ldots, B_1, B_0 >$

Arbre binomial - Preuves

1. $n_0 = 1$ et $n_k = 2n_{k-1}$
2. arbre : x nœuds $\Rightarrow$ $x - 1$ arêtes
3. $h_0 = 0$ et $h_k = 1 + h_{k-1}$
4. $d_0 = 0$ et $d_k = 1 + d_{k-1}$
5. $n_{k,0} = 1$, $n_{k,l} = 0$ pour $l > k$, et $n_{k,i} = n_{k-1,i} + n_{k-1,i-1}$, pour $i = 1, \ldots, k$
6. propriété de décomposition, par récurrence sur k

File Binomiale

Tournoi Binomial

Un *tournoi binomial* est un arbre binomial étiqueté croissant (croissance sur tout chemin de la racine aux feuilles)

File Binomiale

Une *file binomiale* est une suite de tournois binomiaux de tailles strictement décroissantes

Exemples :

- $FB_{12} =< TB_3, TB_2 >$,
- $FB_7 =< TB_2, TB_1, TB_0 >$

Représentation d'une file de priorité

Représentation d'une file de priorité $\mathcal{P}$ de n éléments

- si $n = 2^k$, $\mathcal{P}$ tournoi binomial
- sinon $\mathcal{P}$ file binomiale, suite de tournois correspondants aux bits égaux à 1 dans la représentation binaire de n.

Représentation binaire de n

$$n = \sum_{i=0}^{\lfloor \log_2 n \rfloor} b_i 2^i, \quad \text{avec} \quad b_i \in \{0,1\}, \; b_{\lfloor \log_2 n \rfloor} = 1$$

$\nu(n) = \sum_i b_i$: # bits à 1 dans représentation binaire de n.

File binomiale - Propriétes

Propriétés de FB_n

1. FB_n a n nœuds
2. FB_n a $n - \nu(n)$ arêtes
3. Le plus grand arbre de la file est $B_{\lfloor \log_2 n \rfloor}$ (hauteur $\lfloor \log_2 n \rfloor$ et nombre de nœuds $2^{\lfloor \log_2 n \rfloor}$)
4. Le nombre d'arbres de la file est $\nu(n)$ (avec $\nu(n) \leq 1 + \lfloor \log_2 n \rfloor$)
5. Le minimum de la file est à la racine de l'un des arbres

1. $n = \sum b_i 2^i$,
2. $n - \nu(n) = \sum b_i (2^i - 1)$,
3. $\nu(n) = \sum_i b_i$

Union de files binomiales

1. **Union de 2 tournois de tailles différentes** :
   $TB_{k1} \cup TB_{k2} \longrightarrow F_{2^{k1}+2^{k2}} =< TB_{k1}, TB_{k2} >$
   Exemple : $TB_1 \cup TB_2$
2. **Union de 2 tournois de même taille** :
   $TB_k \cup TB_{k'} \longrightarrow TB_{k+1}$,
   avec $rac(TB_{k+1}) = min(rac(TB_k), (rac(TB_{k'})))$
   Exemple : $TB_2 \cup TB_2'$
3. **Union de 2 files binomiales** $\equiv\equiv$ addition binaire
   Exemple : $FB_5 \cup FB_7$

Union de deux files

1. interclasser les 2 files en partant des tournois de degré minimum
2. lorsque 2 tournois de taille k on engendre un tournoi de taille $k+1$
3. à chaque étape au plus 3 tournois de même taille à fusionner (1 dans chacune des files + 1 retenue de la fusion à l'étape précédente)
4. lorsque 3 tournois de taille k on en retient 2 pour engendrer un tournoi de taille $k+1$, et l'on garde le troisième comme tournoi de taille k.

Primitives sur les tournois binomiaux

EstVide : TournoiB $\rightarrow$ booleen
 renvoie *vrai ssi le tournoi est vide*

Degre : TournoiB $\rightarrow$ entier
 renvoie *le dégré du tournoi*

UTid : TournoiB * TournoiB $\rightarrow$ TournoiB
 renvoie *l'union de 2 tournois de même taille* $T_k * T_k \mapsto T_{k+1}$

Decapite : TournoiB $\rightarrow$ FileB
 renvoie *la file binomiale obtenue en enlèvant la racine du tournoi* $T_k \mapsto < T_{k-1}, T_{k-2}, ..., T_1, T_0 >$

File : TournoiB $\rightarrow$ FileB
 renvoie *la file binomiale réduite au tournoi* $T_k \mapsto < T_k >$

Primitives sur les files binomiales

EstVide : FileB $\rightarrow$ booleen
 renvoie *vrai ssi la file est vide*

MinDeg : FileB $\rightarrow$ TournoiB
 renvoie *le tournoi de degré minimal de la file*

Reste : FileB $\rightarrow$ FileB
 renvoie *la file privée de son tournoi de degré minimal*

AjoutMin : TournoiB * FileB $\rightarrow$ FileB
hypothèse *: le tournoi est de degré inférieur au MinDeg de la file*
 renvoie *la file obtenue en ajoutant le tournoi comme tournoi de degré minimal de la file initiale*

Algorithme d'Union

*UnionFile : FileB * FileB* $\longrightarrow$ *FileB*
renvoie *la file binomiale union des deux files F1 et F2*
Fonction UnionFile(F1, F2)
 Retourne UFret(F1, F2, Ø)
FinFonction *UnionFile*

*UFret : FileB * FileB * TournoiB* $\longrightarrow$ *FileB*
renvoie *la file binomiale union de deux files et d'un tournoi*
Fonction UFret(F1, F2, T)

Fonction UFret(F1, F2, T)
 Si EstVide(T) *; pas de tournoi en retenue*
  Si EstVide(F1) **Retourne** F2
  Si EstVide(F2) **Retourne** F1
  Soient T1=MinDeg(F1) et T2=MinDeg(F2) *;tourn deg min*
   Si Degre(T1)<Degre(T2)
    Retourne AjoutMin(T1,UnionFile(reste(F1),F2))
   Si Degre(T2)<Degre(T1)
    Retourne AjoutMin(T2,UnionFile(F1,reste(F2)))
   Si Degre(T2)=Degre(T1)
    Retourne UFret(reste(F1),reste(F2),UTid(T1,T2))

$\longrightarrow$

 Sinon *; un tournoi en retenue*
  Si EstVide(F1) **Retourne** UnionFile(File(T), F2)
  Si EstVide(F2) **Retourne** UnionFile(File(T), F1)
  Soient T1=MinDeg(F1) **Et** T2=MinDeg(F2)
   Si Degre(T)<Degre(T1) **Et** Degre(T)<Degre(T2)
    Retourne AjoutMin (T,UnionFile(F1,F2)))
   Si Degre(T)=Degre(T1)=Degre(T2)
    Retourne
     AjoutMin(T,UFret(reste(F1), reste(F2), UTid(T1,T2)))
   Si Degre(T)=Degre(T1) *; et* $<$ *Degre(T2)*
    Retourne UFret(reste(F1),F2,UTid(T1,T))
   Si Degre(T)=Degre(T2) *; et* $<$ *Degre(T1)*
    Retourne UFret(F1,reste(F2),UTid(T2,T))
FinFonction *UFret*

Analyse de complexité

Union de 2 files binomiales FB_n et FB_m en $O(\log_2(n+m))$

- Hypothèse :
  toutes les primitives ont une complexité en $O(1)$
- Critère de complexité : *nombre de comparaisons entre clés*
- Complexité dans le *pire des cas*
- idée
  1 union de 2 tournois de même taille $\longrightarrow$ 1 comparaison entre clés et ajoute une arête dans la file résultat.
- Conséquence : nombre de comparaisons pour faire l'union de 2 files égale nombre d'arêtes de la file union diminué du nombre d'arêtes des files de départ

Calcul

Nombre de comparaisons pour faire l'union d'une file binomiale de n éléments et d'une file binomiale de m éléments.

$$\begin{aligned}
\#cp(FB_n \cup FB_m) &= n+m-\nu(n+m)-(n-\nu(n))-(m-\nu(m)) \\
&= \nu(n)+\nu(m)-\nu(n+m) \\
&< \lfloor \log_2 n \rfloor + 1 + \lfloor \log_2 m \rfloor + 1 \\
&\leq 2\lfloor \log_2 (n+m) \rfloor + 2 \\
&= O(\log_2(n+m))
\end{aligned}$$

Exemples :

- $FB_{21} \cup FB_{11}$
- $FB_{21} \cup FB_{10}$

Ajout d'un élément x à une file FB_n

Algorithme :
Créer une file binomiale FB_1 contenant uniquement x.
Puis faire l'union de FB_1 et FB_n

Complexité : $\nu(n) + 1 - \nu(n+1) \longrightarrow$ entre 0 et $\nu(n)$

Exemples :

- $FB_1 \cup FB_8$
- $FB_1 \cup FB_7$

Construction

Complexité de la construction d'une file binomiale par **adjonctions successives** de ses n éléments.

$$\begin{aligned} \#cp(FB_n) &= \nu(n-1) + 1 - \nu(n) \\ &+ \nu(n-2) + 1 - \nu(n-1) \\ &+ \dots \\ &+ \nu(1) + 1 - \nu(2) \\ &= n - \nu(n) \end{aligned}$$

Donc le nombre moyen de comparaisons pour 1 ajout est $1 - \nu(n)/n < 1$.
Coût amorti d'une opération dans une série d'opérations : $couttotal/nbreop$

Suppression du minimum de FB_n

Recherche du minimum
Le minimum de la file est à la racine d'un des tournois la composant
$\longrightarrow \nu(n) - 1$ comparaisons = $O(\log n)$

Suppression du minimum

- Déterminer l'arbre B_k de racine minimale
- Supprimer la racine de $B_k \longrightarrow$ File $< B_{k-1}, \dots, B_0 >$
- Faire l'union des files $FB_n - B_k$ et $< B_{k-1}, \dots, B_0 >$

Complexité : $O(\log n)$

Diminuer une clé

N.B. **Accès direct au nœud dont il faut diminuer la clé**

- modifier la clé
- échanger le nœud avec son père jusqu'à vérifier l'hypothèse de croissance ($\equiv$ tas)

Le nombre maximum de comparaisons est la hauteur de l'arbre ($O(\log n)$)

Coût amorti

Files Binomiales :

- ajout d'un élément et recherche du minimum en $O(1)$
- suppression du minimum et union de 2 files en $O(\log n)$

Files de Fibonacci :

- ajout d'un élément et union de 2 files en $O(1)$
- suppression du minimum en $O(\log n)$

Remarque : on ne peut pas espérer avoir $O(1)$ pour ajout et suppression du minimum, car alors on serait en contradiction avec les résultats de borne inférieure en $O(n \log n)$ pour le tri par comparaisons.

Coût amorti

Définition

- *Coût amorti* d'une opération dans une suite d'opérations $=$ coût moyen d'une opération dans le pire cas.
- coût amorti = coût total / nombre d'opérations

Méthodes

- méthode par agrégat
- méthode du potentiel
- autres...

Coût amorti : méthode par agrégat

- **Principe :** majorer le coût total d'une suite de n opérations et diviser par n.
- **Exemple :** opérations sur les piles
  - `empiler(S,x)` $\rightarrow$ coût 1
  - `dépiler(S)` $\rightarrow$ coût 1
  - `multidépiler(S,k)` $\rightarrow$ coût $\leq k$

  Suite de n opérations :
  - coût maximal d'une opération $O(n)$
  - mais coût amorti de chaque opération en $O(1)$:
  on ne dépile que les éléments empilés $\rightarrow$ coût de n opérations en $O(n)$.

Coût amorti : méthode du potentiel

- **Principe :**
  - structure de données D_i,
  - fonction *potentiel* Φ vérifiant $\Phi(D_i) \geq \Phi(D_0)$
  - *coût amorti* de la i-ème opération :
    $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
    (c_i coût réel de la i-ème opération)
  - coût amorti = $\sum_{i=1}^{n} \hat{c}_i / n$
- **Exemple :** opérations sur les piles
  - $\Phi(D_i) =$ nombre d'objets de D_i
  - coût amorti de chaque opération en $O(1)$

CHAPITRE 2

RECHERCHE ARBORESCENTE

Plan du Chapitre 2

- Arbres binaires de recherche
- AVL
- Arbres 2-3-4
- Arbres B

Problème de Recherche

Bases de données

- Ensemble d'éléments
  - Chaque élément a une clé
  - (Ordre total sur les clés)
- Opérations
  - Rechercher un élément
  - Ajouter un élément
  - Supprimer un élément
  - *Construction*
  - *Rechercher tous les éléments dans un intervalle*

Structures concurrentes

- Arbres de Recherche
- Hachage

Structures-Efficacité

Nombre de comparaisons en moyenne//au pire

	ABR	ABR-Equilibré	Hachage
Recherche(n)	$O(\log n)//O(n)$	$O(\log n)//\ O(\log n)$	$O(1)//\ O(n)$
Ajout(n)	$O(\log n)//O(n)$	$O(\log n)//\ O(\log n)$	$O(1)//\ O(n)$
Suppression(n)	$O(\log n)//O(n)$	$O(\log n)//\ O(\log n)$	————

Arbres de Recherche

1. **Arbres binaires de recherche**
   - en moyenne hauteur en $O(\log n)$, mais au pire dégénère en liste $O(n)$
   - algorithmes recherche, ajout et suppression : parcours d'une branche
   - algorithmes simples : modifications minimes
2. **Arbres de recherche équilibrés**
   - hauteur toujours en $O(\log n)$
   - algorithmes recherche, ajout et suppression : sur une branche
   - algorithmes sophistiqués : modifications locales (branche) rotations, éclatement, ... pour maintenir la hauteur en $O(\log n)$

Primitives sur les arbres binaires

`ArbreVide :` $\rightarrow$ arbre bin
  renvoie *l'arbre vide*
`ArbreBinaire :` *elt* $\times$ *arbre bin* $\times$ *arbre bin* $\rightarrow$ *arbre bin*
  ArbreBinaire(x,G,D) ***renvoie*** *l'arbre binaire dont la racine a pour contenu x et dont les sous-arbres gauche et droit sont G et D*
`EstArbreVide :` *arbre bin* $\rightarrow$ *booléen*
  renvoie *vrai ssi l'arbre binaire est vide*
`Racine :` *arbre bin* $\rightarrow$ *elt*
  renvoie *le contenu de la racine de l'arbre binaire*
`SousArbreGauche :` *arbre bin* $\rightarrow$ *arbre bin*
  renvoie *une copie du sous-arbre gauche de l'arbre*
`SousArbreDroit :` *arbre bin* $\rightarrow$ *arbre bin*
  renvoie *renvoie une copie du sous-arbre droit de de l'arbre*

Opérations sur les ABR

recherche, ajout, suppression
*Abr-ajout : elt * arbre bin* $\longrightarrow$ *arbre bin*
renvoie *l'arbre binaire de recherche résultant de l'ajout de x*
Fonction Abr-ajout (x, ABR)
  Si EstArbreVide (ABR)
    Retourne ArbreBinaire(x, ArbreVide, ArbreVide)
  Si x= Racine(ABR) **Retourne** ABR
  Si x< Racine(ABR)
    Retourne ArbreBinaire(Racine(ABR),
      abr-ajout (x,SousArbreGauche(ABR)),
      SousArbreDroit(ABR))
  Sinon Retourne ArbreBinaire(Racine(ABR),
      SousArbreGauche(ABR),
      abr-ajout (x, SousArbreDroit(ABR)))
Fin Fonction *Abr-ajout*

Arbres de recherche équilibrés

Idéal ABR parfait : hauteur toujours $\sim \log n$
Mais il faut pouvoir réorganiser l'arbre en $O(\log n)$ après un ajout ou une suppression (ex : ajouter 1 dans l'ABR parfait contenant 2,3,4,5,6).
Assouplir contraintes sur forme des arbres en autorisant déséquilibre

- soit en hauteur $\rightarrow$ Arbres AVL
- soit en largeur $\rightarrow$ Arbres B

Arbres AVL

Définition d'un AVL

Un *AVL* (Adelson–Velskii, Landis) est un ABR t.q. en chaque nœud, la hauteur du sous-arbre gauche et celle du sous-arbre droit diffèrent au plus de 1.

Hauteur d'un AVL

Soit h la hauteur d'un AVL avec n nœuds :
$\log_2(n+1) \leq h+1 < 1,44 \log_2 n$

Au pire les arbres de Fibonacci :
$F_0 =< \bullet, , >, F_1 =< \bullet, F_0, >, F_n =< \bullet, F_{n-1}, F_{n-2} >$

Rotations

Rotations pour rééquilibrer, tout en gardant la propriété d'ABR

1. $A = < q, < p, U, V >, W > \Longrightarrow$
   $RD(A) = < p, U, < q, V, W >>$
2. $A = < p, U, < q, V, W >> \Longrightarrow$
   $RG(A) = < q, < p, U, V >, W >$
3. $A = < r, < p, T, < q, U, V >>, W > \Longrightarrow$
   $RDG(A) = < q, < p, T, U >, < r, V, W >>$
4. $A = < r, T, < p, < q, U, V >, W >> \Longrightarrow$
   $RGD(A) = < q, < r, T, U >, < p, V, W >>$

Opérations sur les AVL

- primitive `Hauteur` **:** *arbre bin → nat*
- fonctions de rotation : `RG, RD, RGD, RGD`
- fonction de rééquilibrage d'un arbre
  `Equilibrage` **:** *arbre bin → AVL*
  renvoie *l'arbre AVL obtenu en rééquilibrant l'arbre initial*
  hypothèse *: T est un arbre de recherche, les sous-arbres de T sont des arbres AVL et leurs hauteurs diffèrent d'au plus 2*
- recherche, ajout, suppression

Ajout dans un AVL

*AVL-ajout : elt * AVL ⟶ AVL*
renvoie *l'AVL résultant de l'ajout de x à A*
Fonction AVL-ajout (x, A)
  Si EstArbreVide (A)
    Retourne ArbreBinaire(x, ArbreVide, ArbreVide)
  Si x= Racine(A) **Retourne** A
  Si x< Racine(A) **Retourne**
    Equilibrage (ArbreBinaire(Racine(A),
        AVL-ajout (x,SousArbreGauche(A)),
        SousArbreDroit(A))
  Sinon Retourne
    Equilibrage (ArbreBinaire(Racine(A),
        SousArbreGauche(A),
        AVL-ajout (x, SousArbreDroit(A)))
Fin Fonction *AVL-ajout*

Arbre de recherche général

Arbre de recherche général

Dans un *arbre de recherche général*

- chaque nœud contient un k-uplet $(e_1 < \ldots < e_k)$ d'éléments distincts et ordonnés,
- et chaque nœud a $k+1$ sous-arbres $A_1, \ldots, A_{k+1}$ tels que
  - tous les éléments de A_1 sont $\leq e_1$,
  - tous les éléments de A_i sont $> e_{i-1}$ et $\leq e_i$, pour $i = 2, \ldots, k$
  - tous les éléments de A_{k+1} sont $> e_k$

Arbres 2-3-4

Définition d'un arbre 2-3-4

Un *arbre 2-3-4* est un arbre de recherche

- dont les nœuds contiennent des k-uplets de soit 1, soit 2, soit 3 éléments,
- et dont toutes les feuilles sont situées au même niveau

Hauteur d'un arbre 2-3-4

Soit h la hauteur d'un arbre 2-3-4 avec n éléments :
$h = \Theta(\log n)$

- arbre qui ne contient que des 2-nœuds : $h + 1 = \log_2(n + 1)$,
- vs. arbre qui ne contient que des 4-nœuds : $h + 1 = \log_4(3n + 1)$

Primitives des Arbres 2-3-4

- Notations
  2-nœud : $< (a), T_1, T_2 >$
  3-nœud : $< (a, b), T_1, T_2, T3 >$
  4-nœud : $< (a, b, c), T_1, T_2, T3, T4 >$
- Primitives
  `EstVide :` A2-3-4 → booleen
  `Degre :` A2-3-4 → entier
  `Contenu :` A2-3-4 → LISTE/de longueur 1 à 3/[entier]
  `EstDans :` entier*LISTE[entier] → booleen
  `Elem-i :` A2-3-4 → entier
  ***renvoie** le i-ème élément du nœud (et sinon $+\infty$)*
  `Ssab-i :` A2-3-4 → A2-3-4
  ***renvoie** le i-ème sous-arbre du nœud (et sinon ∅)*

Algorithme de recherche

*234Recherche : entier * A2-3-4 ⟶ booleen*
***renvoie** vrai ssi x est dans A*

Fonction 234Recherche(x, A)
  Si EstVide(A) **Retourne** FAUX
  Si EstDans(x,Contenu(A)) **Retourne** VRAI
  Si x<Elem-1(A) **Retourne** 234Recherche(x, Ssab-1(A))
  Si x<Elem-2(A) **Retourne** 234Recherche(x, Ssab-2(A))
  Si x<Elem-3(A) **Retourne** 234Recherche(x, Ssab-3(A))
  Retourne 234Recherche(x, Ssab-4(A))
Fin Fonction *234Recherche*

Complexité en nombre de comparaisons : $O(\log n)$

Ajout d'un élément

- Ajout aux feuilles (guidé par la recherche)
- un i-nœud se transforme en $(i + 1)$-nœud, par insertion dans la liste
- sauf lorsque la feuille contient déjà 3 éléments ! ! !

Exemple : Construire par adjonctions successives un arbre 2-3-4 contenant les éléments
$(4, 35, 10, 13, 3, 30, 15, 12, 7, 40, 20, 11, 6)$

Deux méthodes de rééquilibrage

- Eclatements en remontée (au pire en cascade sur toute une branche)
- Eclatements en descente (éclatement systématique de tout 4-nœud)

Comparaison des méthodes

Les deux méthodes ne donnent pas forcément le même arbre.
Elles opèrent toutes les deux en $O(\log n)$ comparaisons (transformations sur une branche)

Avantages de la méthode d'éclatements en descente

- parcours de branche uniquement de haut en bas
- transformation très locale : accès parallèles possibles

Inconvénients de la méthode d'éclatements en descente

- taux d'occupation des nœuds plus faible
- hauteur de l'arbre plus grande

Eclatements en descente

Transformations de rééquilibrage locales : sur le chemin de recherche, on éclate systématiquement les 4-nœuds

1. Le père du nœud à éclater ne peut pas être un 4-nœud
2. Le père du nœud à éclater est un 2-nœud
   $P2 =< (x), A1, A2 >$, avec $A1 =< (a, b, c), U1, U2, U3, U4 >$
   $\implies \quad P2 =< (b, x), < (a), U1, U2 >, < (c), U3, U4 >, A2 >$
3. Le père du nœud à éclater est un 3-nœud
   $P3 =< (x, y), A1, A2, A3 >$ et $A2 =< (a, b, c), U1, U2, U3, U4 >$
   $\implies \quad P3 =< (x, b, y), A1, < (a), U1, U2 >, < (c), U3, U4 >$
   $, A3 >$
4. autres cas analogues

Ne modifie pas la profondeur des feuilles (sauf lorsque la racine de l'arbre éclate, la profondeur est alors augmentée de 1)

Algorithme d'ajout

Fonction ajout(x, A)
  Si Degré(A)= 4 **Retourne** ajoutECR(x, A)
  Retourne ajoutSimple(x, A)
Fin Fonction *ajout*

*ajoutECR : entier * A2-3-4/racine4noeud / $\longrightarrow$ A2-3-4*
Fonction ajoutECR(x, A)
  Retourne ajoutSimple(x, A') *A' résulte de l'éclatement de la racine de A*
Fin Fonction *ajoutECR*

*ajoutSimple : entier * A2-3-4/racineNon4noeud/ $\longrightarrow$ A2-3-4*
Fonction ajoutSimple(x, A)
  Si Degré(A)< 4 **Retourne** ajoutSimple(x, Ui)
  Si Degré(Pere(A))=2 **Retourne** ajoutSimple(x, P2)
  Retourne ajoutSimple(x, P3)
Fin Fonction *ajoutSimple*

Remarques

- Ui est le sous-arbre dans lequel doit se poursuivre l'ajout (comme pour une recherche)
- P2 est le transformé2 du père de A (cf. éclatements)
- P3 est le transformé3 du père de A (cf. éclatements)

- Complexité en nombre de comparaisons : $O(\log n)$
- Implantation : représentation des arbres 2-3-4 par des arbres binaires bicolores (voir TD).

Arbres-B

- Recherche Externe : éléments stockés sur disque
- Mémoire secondaire paginée (allouer et récupérer les pages)
- Temps d'accès MS 100000 fois supérieur à MC.
- D'où organisation pour avoir peu de transferts de pages.

Un *B-arbre d'ordre m* est un arbre de recherche

- dont les nœuds contiennent des k-uplets d'éléments, avec $m \leq k \leq 2m$,
- sauf la racine, qui peut contenir entre 1 et $2m$ éléments,
- et dont toutes les feuilles sont situées au même niveau

Arbre Bm, n nœuds : $\log_{2m+1}(n+1) \leq h+1 \leq 1+\log_{m+1}[(n+1)/2]$

Algorithmes et complexité

- Algorithmes analogues à ceux des arbres 2-3-4
- Seul le nœud racine est en mémoire centrale $\Longrightarrow$ nombre d'accès à la mémoire secondaire $=$ hauteur de l'arbre
- hauteur inférieure à $\log_{m+1}[(n+1)/2] \Longrightarrow$ prendre m grand. $m = 250$ peut contenir 125.10^6 éléments dans un arbre de hauteur 2
- éclatement $\rightarrow$ écrire 2 pages en MS (# éclatements borné par hauteur)
- Analyse amortie : # éclatements dans construction par adjonctions successives d'un B-arbre d'ordre m compris entre $1/m$ et $1/2m$
- Analyse de frange : 1 éclatement pour $1,38m$ adjonctions
- Un B-arbre d'ordre m contenant n éléments aléatoires comporte $1,44n/m$ nœuds.

CHAPITRE 3

MÉTHODES DE HACHAGE

Plan du Chapitre 3

- Fonctions de hachage et Gestion des collisions
- Hachage avec chaînage et hachage avec calcul
- Hachage dynamique et hachage extensible
- Comparaison avec la recherche arborescente
- Hachage universel et hachage parfait
- Hachage cryptographique

Méthodes de hachage

Table T de taille m contenant des clés.
Opérations : Rechercher, Insérer, Supprimer une clé x dans T
Fonction de hachage $h : \mathcal{U} \longrightarrow \{0, \ldots, m-1\}$
Accès direct dans T selon la valeur de hachage

*H-ajout : elt * Table * fonctionH $\longrightarrow$ Table*
***renvoie** la table résultant de l'ajout de x*
Fonction H-ajout (x, T, h)
  soit v = h(x)
   Si EstVide T(v) alors T(v) := x
   Sinon Si T(v)$\neq$ x alors GérerCollision(x,T)
  Retourne T
Fin Fonction *H-ajout*

Fonctions de hachage

Fonction de hachage $h : \mathcal{C} \subset \mathcal{U} \longrightarrow \{0, \ldots, m-1\}$

- calcul de fonction de hachage
  - division : $h(x) = x \mod m$, (m premier)
  - multiplication : $h(x) = \lfloor m.frac(\lambda x) \rfloor$, $\lambda = \frac{\sqrt{5}-1}{2}$
- but : obtenir *répartition uniforme* des clés
  $\forall x \in \mathcal{C}, \forall i \in \{0, \ldots, m-1\}, \quad \Pr(h(x) = i) = \frac{1}{m}$
- mais il y a *toujours des collisions* : $x \neq y$ et $h(x) = h(y)$.

  $\Pr(0\ collision) = \frac{\#injections\ de\ [n]\ dans\ [m]}{\#fonctions\ de\ [n]\ dans\ [m]} = \frac{m(m-1)\ldots(m-n+1)}{m^n}$

  $\rightarrow$ nécessité de gérer les collisions.

Analyse du nombre de collisions primaires

- Hypothèse d'uniformité de la fonction de hachage
  $\forall x \in \mathcal{C}, \forall i \in \{0, \ldots, m-1\}, \quad \Pr(h(x) = i) = \frac{1}{m}$
- Probabilité que k clés aient même valeur de hachage v
  $\Pr(X = k) = \Pr(h^{-1}(v) = k) = \binom{n}{k} \frac{1}{m^k} \left(\frac{m-1}{m}\right)^{n-k}$
- Moyenne nb clés par case :
  $E(|h^{-1}(v)|) = \sum k \times \Pr(X = k) = \frac{n}{m}$
- Variance : Var $(|h^{-1}(v)|) = E((X - E(X))^2) \frac{n}{m} (1 - \frac{1}{m})$

Paradoxe des anniversaires

$P =$ Probabilité pour qu'il y ait au moins 1 collision

$$\begin{aligned} P &= 1 - \prod_{i=0}^{n-1} (1 - \frac{i}{m}) \\ &= 1 - \exp \sum_{i=0}^{n-1} \log(1 - 1/i) \sim 1 - e^{-\frac{n^2}{2m}} \end{aligned}$$

$\Rightarrow$ pour $p = 0,5$, $n \sim \sqrt{2mLog2}$, ($m = 365 \rightarrow n = 23$)

Paradoxe des anniversaires : étant donné un groupe de plus de 23 personnes, la probabilité qu'au moins 2 d'entre elles aient la même date anniversaire est supérieure à 1/2.

Gestion des collisions

Différentes méthodes pour gérer les collisions

- Hachage Chainage Séparé : toutes les clés ayant la même valeur de hachage sont dans une structure extérieure (LC)
  - "coût" moyen d'une recherche négative :
    $\frac{1}{m} \sum_{i=1..m} L_i = \frac{n}{m} = \alpha,$
  - coût moyen d'une recherche positive :
    $\frac{1}{n} \sum_{i=0..n-1} 1 + \frac{i}{m} \sim 1 + \frac{\alpha}{2}$
  - coût pire $\sim n$: toutes clés même valeur de hachage
- à l'intérieur de la table
  - par calcul (HLinéaire, HDouble)
  - par chaînage (HCoalescent)

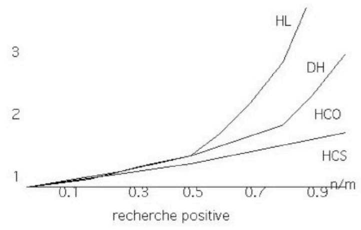
Caractéristiques des méthodes de Hachage

- *Taux de remplissage* $\alpha = \frac{n}{m} < 1$, (sauf pour HCS).
- Très bien adaptées à la gestion d'ensembles statiques (choisir $\alpha \sim 60\%$)
- Mal adaptées aux suppressions ($\Rightarrow$ sup "logiques")
- Si table trop pleine il faut réorganiser en augmentant la mémoire allouée, et en rehachant tous les éléments ($\Rightarrow$ indisponibilité provisoire)

Comparaison des performances

Opération fondamentale = comparaison entre clés (on ne compte pas coût hachage)
Sous l'hypothèse de répartition uniforme des clés

- HChainageSéparé a les meilleures performances, et la suppression est simple.
- HCoalescent très bon mais utilise place pour chaînages internes.
- DoubleH meilleur que HLinéaire, mais nécessite calcul de 2 fonctions de hachage.



Pour $\alpha < 60\%$, toutes méthodes en moyenne moins de 2 essais (mais au pire n).

Hachage Externe

- Pour ne pas avoir à "rehacher"
- Pour traiter la recherche externe (M.S.)

La table est remplacée par un index (arbre digital binaire), et les feuilles adressent vers des pages en mémoire secondaire.

Hachage dynamique et Hachage extensible

- analogie avec les B-arbres : éléments dans pages, qui éclatent quand elles sont pleines
- analogie avec Accès Séquentiel indexé : on maintient un index pour guider vers la page (et si index trop grand, on le pagine lui aussi)
- méthodes de hachage (fonctions de hachage pour disperser les clés) : clé $x \to h(x)$
- utilisent propriétés binaires des valeurs de hachage des clés (index sous la forme d'un arbre digital)

Accès Séquentiel Indexé

Liste des clés triées.
Eléments rangés séquentiellement dans les pages.

Exemple : Encyclopedia Universalis

Index : Page i contient clés jusqu'à $F_i \Rightarrow$ Pour rechercher la clé x à partir de l'index D, on calcule (dichotomie) l'indice i tel que $F_{i-1} < x \leq F_i$, et on renvoie sur la page i.
Index sur plusieurs niveaux (Index des disques et sur chaque disque index des pages)
Performant pour la recherche, mais un ajout peut nécessiter le **recalcul de toutes les pages** ! ! !

Arbre lexicographique

Représentation arborescente des mots d'un dictionnaire en évitant de répéter les préfixes communs
(Ex : complétion automatique, vérificateur d'orthographe)

Alphabet de 26 lettres -> arbres avec noeuds d'arité 26
Exemple : a, bac, balle, ballon, bas, base, bus, sac
Recherche, adjonction et suppression par parcours d'une branche en "épelant" le mot.

Alphabet binaire -> arbre binaire digital
Exemple : 0, 01,001, 01101, 11, 110, 111
Recherche, adjonction et suppression par parcours d'une branche (aiguillage à gauche si 0 et à droite si 1)

Hachage dynamique

- Table de hachage remplacé par index = arbre digital
- Index fabriqué par raffinnements successifs d'une fonction de hachage. clé $x \to h(x) \in \{0,1\}^*$

Pour rechercher un élément x

- on suit un chemin dans l'arbre digital, qui mène à une feuille pointant sur la page contenant x,
- le chemin suivi est guidé par la valeur de hachage de x.

Ajouts augmentent le nombre d'éléments dans une page

- allouer de nouvelles pages en MS
- répartir les éléments dans les pages en utilisant plus ou moins d'informations de la valeur binaire de leur fonction de hachage (selon qu'il y a plus ou moins de collisions)
- nouvelles pages référencées par nouvelles feuilles de l'arbre digital (qui croît).

Exemple

Hachage dynamique des clés $E, X, T, F, R, N, C, L, S, G, B$, avec pages de capacité $C = 4$
$h(E) = 00101, h(X) = 11000, h(T) = 10100, h(F) = 00110,$
$h(R) = 10010,\ h(N) = 01110, h(C) = 00011, h(L) = 01100,$
$h(S) = 10011, h(G) = 00111, h(B) = 00010$

Tri des éléments dans une page :
temps négligeable par rapport au temps d'accès.

Hachage Extensible

- index = arbre digital parfait $\Rightarrow$ representable par un tableau de 2^d mots
- chaque mot est une suite de d bits (nombre entre 0 et $2^d - 1$) qui adresse vers une page
- Pour rechercher la clé x, on utilise les d premiers bits de $h(x)$, pour accéder à une page
- plusieurs mots peuvent adresser la même page : (2^{d-k} si les clés de cette page ont les k mêmes premiers bits pour leurs valeurs de hachage)

Lors d'un ajout les modif. peuvent être à plusieurs niveaux :

- insertion dans une page (si elle n'est pas pleine)
- éclatement d'une page avec redistribution des clés (et l'index n'est pas modifié)
- doublement de l'index

Exemple

Hachage extensible des clés
$E, X, T, E, R, N, A, L, S, E, A, R, C, H, I, N, G, E, X, A, M, P, L, E,$
avec pages de capacité $C = 4$.
$h(E) = 00101, h(X) = 11000, h(T) = 10100, h(R) = 10010,$
$h(N) = 01110, h(A) = 00001, h(L) = 01100, h(S) = 10011,$
$h(C) = 00011, h(H) = 01000, h(I) = 01001, h(G) = 00111,$
$h(M) = 01101, h(P) = 10000$

Hachage/Arbres de Recherche

- Mémoire centrale : Complexité en nombre de comparaisons

	ABR-Equilibré	Hachage
Recherche, ajout	$O(\log n)$// $O(\log n)$	$O(1)$// $O(n)$
Suppression	$O(\log n)$// $O(\log n)$	————
Recherche par intervalle (ou tris)	$O(\log n)$// $O(\log n)$	————

- Mémoire secondaire paginée :
  - Complexité en nombre d'accès aux pages
    temps d'accès $>>$ temps de traitement
  - Taux de remplissage des pages ($\alpha = n/Cm$)

	B-arbres	H-Extensible
Recherche, ajout, suppression	$O(1)$	$O(1)$
Taux de remplissage	70%	70%
Lectures séquentielles (tris)	Oui	Non
Accès concurrents	complexes	plus simples

Hachage universel

Choix aléatoire fonction de hachage $\rightarrow \sim$ h uniforme

Ensemble universel de fonctions de hachage

- $\mathcal{H} : \{h : \mathcal{U} \longrightarrow \{0, \ldots, m-1\}\}$ ensemble fini ;
- $\mathcal{H}$ est *universel* ssi $\forall x, y \in \mathcal{U}, x \neq y,$
  $|\{h \in \mathcal{H}; h(x) = h(y)\}| = \frac{|\mathcal{H}|}{m}$

D'où pour $h \in \mathcal{H}$ aléatoire : $\forall x \neq y \in \mathcal{U}^2, \mathbb{P}(h(x) = h(y)) = \frac{|\mathcal{H}|}{m} \cdot \frac{1}{|\mathcal{H}|} = \frac{1}{m}$.

Propriété

Soit h fonction aléatoire dans $\mathcal{H}$ universel ; si h répartit n clés dans une table de taille m, alors $\forall x$, le nombre moyen de clés y telles que $h(y) = h(x)$ est inférieur à n/m.

$\sum_{y \neq x} \mathbb{P}(h(x) = h(y)) = \frac{n-1}{m}$

Tirer aléatoirement une fonction de hachage

On dispose d'une fonction de hachage uniforme
$h : E \rightarrow [0, m-1]$,
on peut créer alors une autre fonction h',

$$h'(x) = (a \times h(x) + b) \mod m$$

avec a, b deux entiers tirés aléatoirement entre 0 et $m - 1$

En pratique : technique peu coûteuse + bons résultats

On appelle ces fonctions 1-universelle (voir TD).

Construction d'un ensemble universel

- on suppose m premier ;
- décomposition des clés en chiffres m-aires : $\forall x \in \mathcal{U}, x = (x_0, x_1, \ldots, x_r)$,
- randomisation : choisir $a = (a_0, a_1, \ldots, a_r)$, avec chaque a_i aléatoire dans $\{0, 1, \ldots, m-1\}$

Théorème

Soit $h_a : x \rightarrow \sum_{i=0}^{r} a_i x_i \mod m$.
L'ensemble $\mathcal{H} = \{h_a\}$ est universel, de cardinal m^{r+1}.

Exemple : adresses IP – 132.227.74.253 – 4 champs de 8 bits : $x = (x_1, x_2, x_3, x_4)$.

Pour hacher ~ 250 adresses, choisir $m = 257$ et

$\mathcal{H} = \{h_a; h_a(x) = a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4, \quad \text{avec} \quad a_i \in [0..256]\}$

Preuve

Soient $x \neq y$, $x = (x_0, \ldots, x_r)$, $y = (y_0, \ldots, y_r)$, avec $x_0 \neq y_0$.

Montrer que le nombre de h_a tq $h_a(x) = h_a(y)$ est m^r (i.e. $\|\mathcal{H}\|/m$)

$h_a(x) = h_a(y) \Longrightarrow \sum_{i=0}^{r} a_i x_i \equiv \sum_{i=0}^{r} a_i y_i (\mod m)$
$\Longrightarrow a_O(x_0 - y_0) + \sum_{i=1}^{r} a_i(x_i - y_i) \equiv O(\mod m)$
$\Longrightarrow a_0 = \left(-\sum_{i=1}^{r} a_i(x_i - y_i)\right).(x_0 - y_0)^{-1}(\mod m)$ (lemme)
Donc $\forall a_1, a_2, \ldots, a_r$, il existe un seul a_0 tel que $h_a(x) = h_a(y)$.
Et le choix des $a_1, a_2, \ldots, a_r$ donne m^r fonctions possibles.
Donc $\mathcal{H}$ est universel.

Lemme : Si m est premier, alors $\forall z \neq 0 \in \mathbb{Z}/m\mathbb{Z}$, $\exists! z^{-1}$ tq. $z.z^{-1} = 1$

Ex. pour $m = 7$, $z = 1, 2, 3, 4, 5, 6$, $z^{-1} = 1, 4, 5, 2, 3, 6$.

Hachage parfait

Ensemble statique de n clés : Pire cas $O(1)$, mémoire $O(n)$

Idée : deux niveaux de hachage universel

- choisir m premier, $m \sim n$ et $h_1 \in \mathcal{H}$ universel.
- pour $i = 1..m$, soit n_i le nombre de clés tq $h_1(x) = i$
- pour chaque i, choisir m_i premier, $m_i \sim n_i^2$ et $h_{2,i} \in \mathcal{H}$ universel.

Exemple

Mémoire totale $O(n)$ en moyenne car $\sum E(n_i^2) = O(n)$.

Pas de collision au second niveau (proba$>1/2$)

Théorème

Soit $\mathcal{H}$ un ensemble universel pour une table de taille $m \sim n^2$;
le nb total de collisions pour hacher n clés est $< \frac{1}{2}$ en moyenne.

$\forall (x, y), \Pr(h(x) = h(y)) = \frac{1}{m} \sim \frac{1}{n^2}$. Et le nombre de couple est $\binom{n}{2}$. Donc le nombre moyen de collisions est $\binom{n}{2}\frac{1}{n^2} < \frac{1}{2}$

Corollaire

Dans ces conditions, $\Pr(\text{aucune collision}) > \frac{1}{2}$.

appliquer l'inégalité de Markov.

Pour le hachage parfait il suffit donc d'essayer des fonctions de $\mathcal{H}$ jusqu'à ce qu'il n'y ait pas de collision (prétraitement) ; et ensuite on travaille avec ensemble statique (recherches uniquement).

Hachage cryptographique

- Utilisation du hachage dans un contexte différent : Cryptage et Compression
- $\mathcal{M}$ messages de tailles qques, $\mathcal{S}$ signatures (empreintes) de taille fixe m et $h : \mathcal{M} \to \mathcal{S}$
- "identifier" le message et sa signature
- représentation compacte et paradoxe des anniversaires : si m= 2^k, il faut $\sim 2^{k/2}$ messages pour avoir une collision avec proba 0.5
- Applications
  - vérification de données (web, ftp)
  - authentification de messages

Cryptosystème à clé publique

- Chaque utilisateur a 2 clés $u \to (P_u, S_u)$, avec
  - inverses : $P_u(S_u(T)) = S_u(P_u(T)) = T, \forall T$ texte
  - P_u publique (tables) et S_u secrète (seul u la connaît)
- Opérations possibles :
  - Crypter message T de A vers B : A transmet $P_B(T)$ à B
  - Signature de A vérif. par tous : A transmet T et $S_A(T)$
  - Crypter et signer : A transmet $P_B(T; S_A(T))$
- Algorithme RSA pour calculer (P_u, S_u) :
  - u choisit p et q deux (grands) nbres premiers ; et $n = pq$
  - u choisit e petit, premier avec $(p-1)(q-1)$ et calcule d inverse de e modulo $(p-1)(q-1)$
  - $P_u = (e, n)$ et $S_u = d$
  - $P_u(S_u(T)) = S_u(P_u(T)) = T, \forall T$

Propriétés Hachage cryptographique

- h compresse : $\{0,1\}^\star \to \{0,1\}^m$
- h à sens unique : facile à calculer et très difficile à inverser
  - *facile* : calculable en temps-mémoire polynomial
  - *très difficile* : techniquement impossible

Propriétés recherchées

1. Préimage difficile : pour presque tout $y \in \{0,1\}^m$, il doit être *très difficile* de trouver $x \in \{0,1\}^\star$ tel que $h(x) = y$.
2. Collisions faibles difficile : étant donné $x \in \{0,1\}^\star$, il doit être *très difficile* de trouver $x' \neq x$ tel que $h(x) = h(x')$.
3. Collisions fortes difficile : il doit être *très difficile* de trouver (x, x') tels que $x \neq x'$ et $h(x) = h(x')$.

Méthodes de hachage cryptographique

- MD- Message Digest : Rivest
  - MD4 (1989), MD5 (1991) – 128 bits
  - faille en 96 et collision complète en 2004
  - vérification de téléchargements FTP
- SHA Secure Hash Algorithm : NSA
  - SHA-0 et SHA-1 – 160 bits
  - faille en 93 et collision complète en 2004
  - SHA-256, signature sur 256 bits ..., SHA-512
- RIPEMD-160, Whirlpool (512) : EU project