921

Algorithmes de recherche et structures de données associées.

Soit E un ensemble d'éléments ayant chacun une clé. On note U l'ensemble des clés possibles. Quelles structures de données utiliser pour rechercher efficacement dans E ?

I Tableaux et listes

1) Vecteurs de bits ($U \subset \mathbb{N}$)

On représente E par un tableau T de taille $|U|$ tel que

$T[i] = \begin{cases} 1 & \text{si } \exists x \in E, \text{clé}(x)=i \\ 0 & \text{sinon} \end{cases}$. Savoir si $x \in E$ revient à savoir si $T[\text{clé}(x)]=1$. On a donc une recherche en $O(1)$ mais cette idée est inutilisable si $|U|$ est grand.

2) Recherche naïve

On représente E par la liste L des clés de ses éléments. Rechercher x dans E revient à parcourir L jusqu'à trouver $\text{clé}(x)$. D'où une complexité au pire en $O(|E|)$.

3) Recherche dichotomique (U ordonné)

On représente E par le tableau trié de ses clés. On effectue alors une recherche dichotomique en $O(\log|E|)$

II Hachage [Cor]

1) Principe général

Si $|U|$ est grand, I.1 est inutilisable. On cherche ici une structure de taille $O(|E|)$ telle que la recherche soit $O(1)$.

On choisit un tableau T de taille $m < |U|$ et $h : U \to [\![0, m-1]\!]$. On stocke alors $x \in E$ à la place $T[h(\text{clé}(x))]$. Comme $m < |U|$, h n'est pas injective. Si deux clés ont même image par h, on parle de collision (Figure 1).

2) Résolution par chaînage

On considère T comme un tableau de listes. Ainsi $T[i]$ contient tous les éléments $x \in E$ tel que $h(\text{clé}(x)) = i$

Notons alors $n_i = |T[i]|$ et $\alpha = |E|/m$ le coefficient de remplissage.

Si h est mal choisi, toutes les clés peuvent être envoyées sur la même case. Dans ce cas la recherche se fait en $\Theta(|E|)$. On va voir comment choisir h pour éviter ça.

Hypothèse : On suppose que h vérifie l'hypothèse de hachage uniforme simple (HHUS) :

$\mathbb{P}(h(x)=r) = 1/m \quad \forall x \in U \ \forall r \in [\![0, m-1]\!]$.

thm : Sous HHUS, toute recherche infructueuse est au pire en $\Theta(1+\alpha)$ et toute recherche réussie est en moyenne en $\Theta(1+\alpha)$.

3) Hachage universel et hachage parfait

déf : Soit $\mathcal{H}$ une classe de fonctions de hachage. $\mathcal{H}$ est dite universelle ssi

$\forall k \neq l \in U, \ |\{h \in \mathcal{H}, h(k)=h(l)\}| \leq |\mathcal{H}|/m$.

ex : Si $p > |U|$ est premier, la classe $\{k \mapsto ((ak+b) \bmod p) \bmod m, \ a \in (\mathbb{Z}/p\mathbb{Z})^*, \ b \in \mathbb{Z}/p\mathbb{Z}\}$ est universelle

thm : Si $\mathcal{H}$ est universelle et $h \in \mathcal{H}$, $\mathbb{E}(n_{h(k)}) \leq 1+\alpha \ \forall k \in U$.

app : Soit E un ensemble fixé, i.e. on ne fait plus d'insertion ni de suppression, seulement des recherches. On peut construire une structure de taille $O(|E|)$ où la recherche est $O(1)$ au pire.
DEV

app :
- implémentation des dictionnaires
- utilisation de CD-Rom
- tables des symboles/mots réservés dans les compilateurs

III Structures arborescentes [Cor][Bear]

1) Arbres de recherche.

def : Un arbre de recherche est un arbre étiqueté tel que tout noeud ayant $k+1$ sous-arbres $A_0 \dots A_k$ contient un k-uplet $x_1 < \dots < x_k$ tel que pour toute composante x d'une étiquette d'un noeud de A_i, $x_i < x < x_{i+1}$.

ex : Figure 2.

def : Un arbre binaire de recherche (ABR) est un arbre de recherche où tout noeud non terminal a deux fils.

On a un algorithme de recherche linéaire en la hauteur de l'arbre, donc entre $\log|E|$ et $|E|$.

Si pour chaque clé, on connaît la probabilité qu'on la recherche, on peut s'intéresser à des ABR optimaux, i.e. qui minimisent l'espérance du temps de recherche d'une clé. (DEV2).

2) Equilibrage et AVL.

def : Un AVL est un arbre binaire de recherche tel que pour tout noeud $x \in T$ de sous-arbres G et D, $|h(G)-h(D)| \leq 1$.

thm : Soit T un AVL à n noeuds, de hauteur h. Alors
$$\log_2(n+1) \leq h+1 \leq 1.44 \log_2(n+1)$$

cor : Les opérations de recherche dans un AVL sont en $\Theta(\log(n))$.

rmq : Les opérations d'insertion et suppression sont plus complexes à cause des rééquilibrages mais restent $O(\log n)$

3) Arbres 2-3-4.

def : Un arbre 2-3-4 est un arbre de recherche où tous les noeuds internes ont 2,3 ou 4 fils, et dont toutes les feuilles sont à la même hauteur

thm : La hauteur h d'un arbre 2-3-4 contenant n clés vérifie
$$\log_4(n+1) \leq h+1 \leq \log_2(n+1).$$

cor : La recherche est encore en $\Theta(\log n)$.

4) Arbres rouge-noir.

def : Un arbre rouge/noir est un ABR vérifiant :
- * tout noeud est rouge ou noir
- * la racine est noire
- * les feuilles sont noires
- * les fils d'un noeud rouge sont noirs
- * $\forall x$, les chemins issus de x et allant jusqu'aux feuilles ont même nombre de noeuds noirs.

thm : Un arbre rouge/noir à n noeuds a une hauteur plus petite que $2\log_2(n+1)$.

cor : La recherche est en $O(\log n)$.

rmq : On peut passer d'un arbre rouge/noir à un arbre 2-3-4 et réciproquement via la figure 3.

5) Arbres persistants et duplication de chemins

On veut gérer une base de données qui évolue dans le temps. On veut savoir si x appartient à la structure au temps t. (i.e. après t opérations insérer/supprimer)

1ère idée : recopiage intégral à chaque opération (Figure 4)

2ème idée : on utilise des arbres et on duplique chaque noeud dont un descendant est modifié (Figure 5).

La recherche de x à l'instant t revient à chercher x dans l'arbre dont la racine est pointée par $T[t]$. L'insertion, la suppression et la recherche sont en $O(\ln n)$ où n est le plus grand nombre de clés présentes dans la structure à l'instant t.

6) Tas et files de priorité

déf : • Un tas est un arbre binaire dont tous les niveaux sont complets sauf éventuellement le dernier, où tous les noeuds sont tassés à gauche.
• Un tas max est un tas où chaque noeud est plus grand que ses fils.

rmq : • On représente un tas par un tableau (Figure 6)
• Le maximum d'un tas max est sa racine.

Si A est un tas, $x \in A$ dont les sous-arbres G et D sont des tas max. Alors on peut obtenir un tas max contenant x, G et D en faisant descendre x tant qu'il viole la propriété de tas max. On parle d'entassement.

app : • Construction d'un tas max à partir d'un tas en temps linéaire
• On utilise l'entassement pour récupérer un tas max quand on extrait la racine d'un tas max.

On peut implémenter une file de priorité grâce aux tas max :
• défilage par extraction du maximum : on extrait la racine puis on entasse pour obtenir un tas max ⇝ $O(\ln n)$
• enfilage : on ajoute un noeud (correspondant à une case à la fin du tableau) qui contient la nouvelle priorité et on fait remonter le noeud jusqu'à obtenir un tas max ⇝ $O(\ln n)$.

app : • algorithme de Dijkstra
• ordonnancement de tâches/processus.
• codage de Huffman.

[Cor] [Beau] [Pap] [Baa]

IV Union Find, union par rang et compression de chemins

On veut pouvoir gérer des classes d'équivalences de $[\![0, n-1]\!]$ avec deux opérations : • déterminer la classe d'un élément
• fusionner deux classes.

On représente chaque classe par un arbre et la partition par une forêt. On prend un tableau T tel que $T[i]$ est le père de i. Si i est racine, on pose par convention $T[i] = i$.
On a alors les deux algorithmes

Trouver(x)
  Si $x \neq T[x]$, Trouver($T[x]$)
  Sinon renvoyer x
Union(x, y)
  $T[\text{Trouver}(x)] \leftarrow \text{Trouver}(y)$

problème : On peut se retrouver avec des arbres peigne
Trouver est alors en $O(n)$

Union par rang :
→ On calcule un tableau Rang tel que Rang$[x]$ majore la hauteur du sous-arbre issu de x.
→ Lors d'une union, on fait pointer le noeud de plus petit rang parmi Trouver(x) et Trouver(y) vers l'autre
→ On actualise Rang

Compression de chemins :
→ Quand on appelle Trouver(x), on fait pointer tous les noeuds parcourus vers Trouver(x)
(Figure 7)

thm : Si on fait m opérations, leur complexité est $O(m\,\alpha(n))$ donc la complexité amortie est $O(\alpha(n))$ où α est une fonction à croissance très lente ($\alpha(n) \leq 5$ pour tous les n utilisés en pratique).

app : * algorithme de Kruskal
* composantes connexes d'un graphe
* unification
* Kelmetaire congruent.

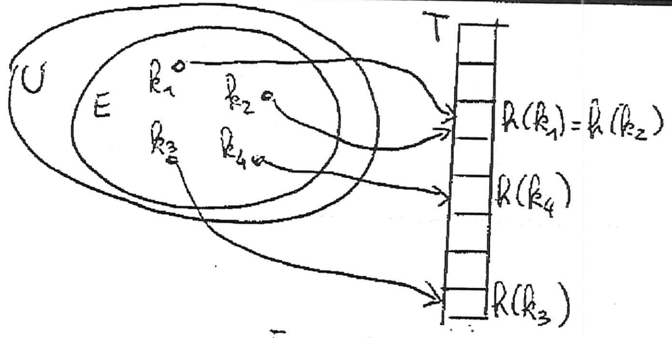

Figure 1.

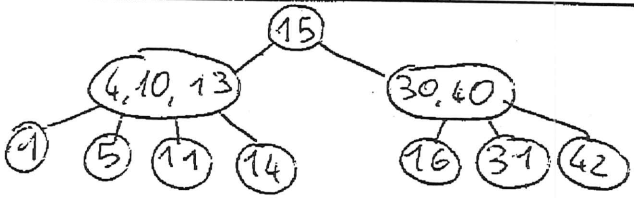

Figure 2

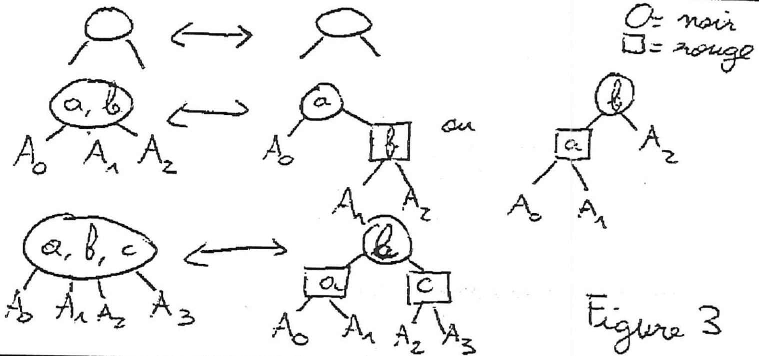

Figure 3

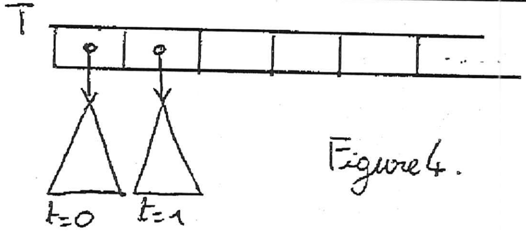

Figure 4.

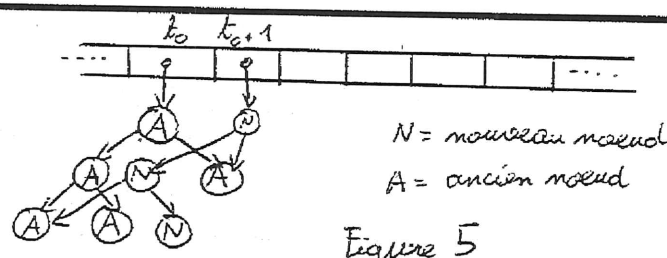

Figure 5

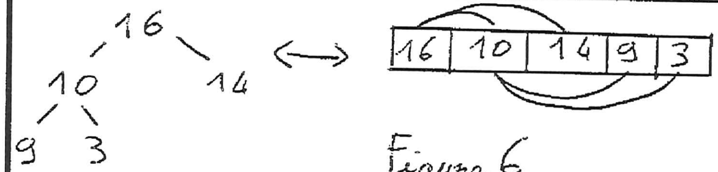

Figure 6

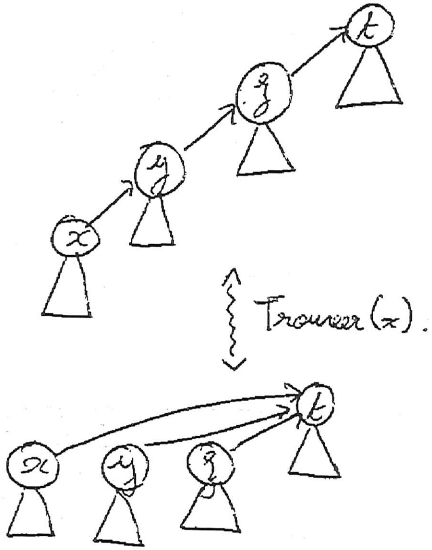

Figure 7.

[Cor] Carmen
[Beau] Beauquier
[Pap] Papadimitriou, Algorithms
[Baa] Baader Nipkow, Term rewriting systems and all that