901

Structure de données. Exemples et applications.

Motivation : Stoker et manipuler (efficacement) un ensemble de données dynamique.

Définition : [1, p.37] Un type abstrait de données est la description d'un ensemble de données et les opérations que l'on peut y appliquer.

Définition : [1, p.37] Un type concret est la description d'un format de representations internes des données en machine.

Définition : [1, p.37] Une structure de données pour un type abstrait est la donnée d'un type concret et des implémentations des différentes fonctions associées.

I - Le type abstrait de base.

A - Les opérations souhaitées.

* find : trouver un élément de notre structure.
* create : créer la structure.
* union : unifier deux structures.
* insert : ajouter un élément à notre structure.
* delete : enlever un élément de notre structure.
* find-min : trouver le plus petit élément selon un ordre donné.
* find-earliest : trouver le dernier élément ajouter.

B - Le tableau : une structure de données contiguë.

Type concret : un espace mémoire contiguë de taille fixe et un entier conservant le nombre de données dans la structure.

Implémentation :

* insert : quand le tableau est plein, on en crée un nouveau qui vaut deux fois la taille du premier.
* delete : on ramène le dernier élément sur celui qu'on supprime.
* union : création d'un tableau de taille des deux tableaux et dont les valeurs sont celles des deux tableaux.
* find-earliest : impossible.

Le reste des opérations se fait de manière usuelle en parcourant le tableau séquentiellement.

C - La liste : l'apparition des pointeurs.

Type concret : une liste chainée à l'aide des pointeurs.

Implémentation :

* insert : insertion en tête.
* delete : manipulation de pointeurs
* union : le dernier élément va pointer sur le premier de la deuxième (un pointeur existe sur le dernier élément)
* find-earliest : le premier élément de la liste.

Le reste des opérations se fait de manière habituelle en parcourant la liste en suivant les pointeurs.

Proposition : Comparaison des complexités selon le type concret. On note n le nombre d'éléments dans la structure et m celui dans une deuxième structure si besoin.

Opérations	Tableau	Liste	Opérations	Tableau	Liste
find	$O(n)$	$O(n)$	create	$O(n)$	$O(1)$
union	$O(n+m)$	$O(1)$	insert	$O(n)$	$O(1)$
find-min	$O(n)$	$O(n)$	delete	$O(n)$	$O(n)$
find-earliest	X	$O(1)$			

Remarque : En moyenne, l'insertion dans un tableau se fait en $O(1)$.

Remarque : Dans la suite nous allons réduire le nombre des opérations pour les spécialisées.

II - Les structures de données d'ordonnancement.

A - La pile et la file : des structures séquentielles

Type abstrait : Pile : insert, create, find-earliest, delete-earliest
File : insert, create, find-oldest, delete-oldest

Application : parcours en profondeur
Complexité : $O(|E|+|V|)$.

Algorithme : entrée (G, o)
create S (S une file)
tant que $S \neq \emptyset$
  (p, v) = find-oldest S
  delete-oldest S
  si p non marqué alors
    marquer p
    pour $(p, v) \in G.E$
      insert(p, v)
où $G = (V, E)$ et $o \in V$.

Type concret : Les listes doublement chaînées.

Implémentation :

* insert : au début de la liste.
* find : donner le premier (ou le dernier) élément de la liste.
* delete : supprimer le premier (ou le dernier) élément.

Proposition : Avec ce type concret, les opérations sur les piles et les files s'effectue en $O(1)$.

B - La file de priorité : une généralisation sur l'ordre.

Type abstrait : File de priorité : insert, create, find-max, delete-max.

Remarque : On a une file de priorité analogue où on cherche le min à la place du max.

Application : algorithme de Dijkstra du plus court chemin [2, P.600]
Complexité : dépend de celle de la file de priorité.

Algorithme : entrée : G, w, s
initialisation (G, s)
$E = \emptyset$ (E ensemble)
$F = G.S$ (F file de priorité)
tant que $F \neq \emptyset$
  u = find - min F
  $E = E \cup \{u\}$
  pour $(u, v) \in E$
    relacher (u, v, w)
où $G = (V, E)$, $s \in V$, w un poid.

Type concret : tas binaire [2, P.140]
Définition : Un tas binaire est un arbre binaire dont les descendants d'un noeud sont tous supérieurs ou égale à ce noeud. Dans un arbre binaire tous les noeuds ont uniquement deux fils.

Implémentation : [2, P.140] On voit le tas comme un tableau.
* insert : insère l'élément le plus à gauche possible et tant que cet élément n'est pas racine et que son père a une valeur inférieure effectuer l'échange.
* delete - max : supprimer la racine et l'échanger avec la valeur la plus à gauche (dernière valeur du tableau) et tant que celui-ci a des fils et qu'il est inférieur à ses fils. On l'échange avec le maximum de ses fils.
* find - max : prendre la racine.
* create : insert une a une des valeurs dans un tas vide.

Application : [2, P.140] Tri par tas :
construction du tas et tant que celui-ci est non vide find-max et delete-max.

Proposition : Comparaison des complexités en fonctions de types concrets pour une file de priorité. On note n le nombre de données.

Type concret	create	insert	find - max	delete - max
Liste	$O(n)$	$O(1)$	$O(n)$	$O(1)$
Liste triée	$O(n \log n)$	$O(n)$	$O(1)$	$O(1)$
Tas binaire	$O(n \log n)$	$O(\log n)$	$O(1)$	$O(\log n)$

C - Le graphe : des relations non séquentielles.
Type abstrait : le graphe est un type abstrait relationnelle où le test principal consiste à savoir s'il y a une connexion entre deux sommets (a)→(b). Le graphe est non-orienté s'il y a symétrie dans le test principal. Sinon, il est orienté.
Remarque : le graphe induit un ordre non total dans les données.

Types concrets : On a deux types concrets majeurs.
* liste d'adjacence : souple pour l'ajout et la suppression de sommets mais en $O(n)$ pour le test principal.
* matrice d'adjacence : tests principal en $O(1)$ mais l'ajout et la suppression des sommets demande la création d'une nouvelle table.

Application : [2, P.567] Le tri topologique d'un graphe acyclique orienté.
Algorithme : entrée $G = (V, E)$
Pour chaque sommet faire un parcours en profondeur pour calculer les dates de fin de traitement.
A chaque fin de traitement, insérer l'élément dans une liste
Retourner cette liste.
Complexité : $\Theta(|V| + |E|)$.

III - Les structures de recherche : dictionnaire.

Type abstrait : dictionnaire : insert, delete, find.

A - Les types concrets arborescents.

Définition : [2, P.267] Un arbre binaire de recherche est un arbre dont le sous-arbre gauche d'un noeud est inférieur et le sous-arbre droit dont le sous-arbre droit lui est supérieur.

Implémentation : Quelque soit le type concret, on a
* find : tant qu'on a pas trouver si la valeur est inférieure au noeud courant aller à gauche, sinon aller à droite.
Pour insert ou delete celui-ci dépend du type concret.

Proposition : Dans le pire cas, chacune des opérations se fait en $O(h)$ où h est la hauteur de l'arbre.

→ **Les arbres équilibré : AVL.** [2, P.310]
* insert et delete : se fait comme find, à la racine mais peut déséquilibré l'arbre qu'il faut alors rééquilibré.
* rééquilibrage : rotation gauche et rotation droite.
* complexité : $\Theta(\log n)$

→ **Les arbres optimaux** [2, P.368]
Lorsque les recherches sur les données ne sont pas uniforme, un arbre non équilibré qui laisse accessible les clés les plus cherchées près de la racine. Un type concret est alors un arbre optimal construit aléatoirement suivant la fréquence de recherche.

Proposition : La création d'un tel arbre se fait en $\Theta(n^3)$ où n est le nombre de clés.

DEV1

B- Un type concret séquentiel : table de hachage. [2, p.235]

Une table de hachage est une généralisation d'un tableau où la case de stockage d'une valeur est donnée par une fonction de hachage souvent surjective.

Définition : Une collision entre deux données implique que leur valeurs de hachage (par la fonction) est égale.

Définition : Un tableau de la taille de l'espace d'arrivée de la fonction de hachage (souvent plus petit que celui de départ) et une liste simplement chaînée pour chacune de ces cellules.

Type concret :

Implémentation :

* insert : calculer la valeur par la fonction de hachage et l'insérer dans la liste correspondante à cette valeur.
* find : calculer la valeur par la fonction et rechercher l'élément dans la liste correspondant.
* delete : calculer la valeur par la fonction et supprimer l'élément dans la liste correspondante.

IV - Partitionner un ensemble : union - find. [2, p.519]

Objectif : On veut une structure de données constituée d'un ensemble de structures de données permettant de savoir (efficacement) à quelle structure (vue comme une classe d'équivalence) chaque donnée appartient et unifier deux de ces structures.

Type abstrait : Union - find : create, find, union.

Application : L'algorithme de Kruskal et les arbres couvrant de poids minimal. [2, p.584]

Algorithme : entrée : (G, w) où $G = (V, E)$ et w une fonction poids

```
E = ∅
pour chaque v ∈ G.V
    create (v)
trier les arêtes de G.E par ordre croissant
de poids w
pour chaque (u, v) ∈ G.E pris par ordre croissant
    si find (u) ≠ find (v)
        E = E ∪ {(u, v)}
        union (u, v)
renvoie E.
```

Complexité : À l'aide d'une structure union - find optimale, on a une complexité en $O(|E| \log |V|)$

Type concret :
* liste chaînée et union par rang
* forêt et union par rang, compression de chemin. DEV2

Hypothèse : La structure union - find ne touche pas aux données : l'ensemble des données admettent un pointeur (et réciproquement) vers leur copie dans union - find.

Proposition : Une séquence de m opérations find, create et union telles qu'il y a n opérations create se fait en $O(m + n \log n)$ pour une liste chaînée avec union par rang.

Proposition : Une séquence de m opérations find, create et union telles qu'il y a n opérations create se fait en $O(m\,\alpha(n))$ pour une forêt avec union par rang et compression de chemin.

V - Stoker et manipuler des données en grands nombres : la Base de données relationnelle. [2, p.447]

Objectif : On souhaite une structure de données qui permet de stocker et de manipuler des données en grand nombre efficacement. Elle doit donc gérer un stockage externe efficace.

Type abstrait : Base de données relationnelle : find ; create ; insert ; delete.

Application : recherche dans un ensemble de donnée présente sur un disque externe.

Type concret : les B-arbres.

Définition : Un B-arbre T possède les propriétés suivantes :
* chaque noeuds possèdent les attributs
  + $x.n$ le nombre de clés ;
  + $x.clé_1 \leq \dots \leq x.clé_n$ les n clés ;
  + $x.feuille$ = VRAI si le noeud est une feuille ;
  + $x.c_i$ les $(n+1)$ pointeurs vers ses fils (si ce n'est pas une feuille)
* si k_i est stocké dans l'arbre de racine $x.c_i$ alors $x.c_{i-1} \leq k_i \leq x.c_i$
* les feuilles sont toutes à la même hauteur
* t est le degré minimal de T : $t - 1 \leq x.n \leq 2t - 1$ pour tout noeud sauf la racine.

Proposition : Toutes les opérations sur un B-arbre (sauf create) s'effectue en $O(\log_t n)$ où t est le degré minimal et n est le nombre de données. DEV 3

Figure 1 : Un tableau .

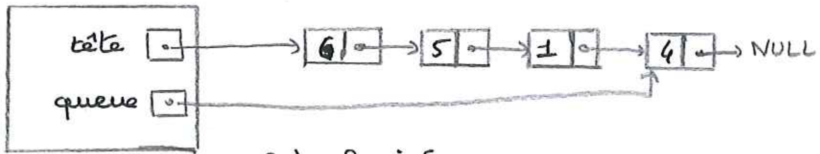


Figure 2 : Une liste chaînée.

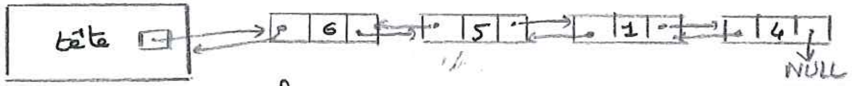


Figure 3 : Une pile.

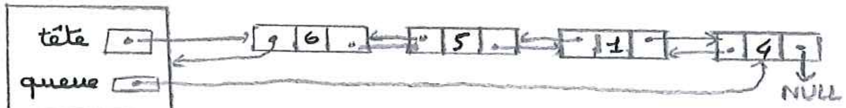


Figure 4 : Une file.

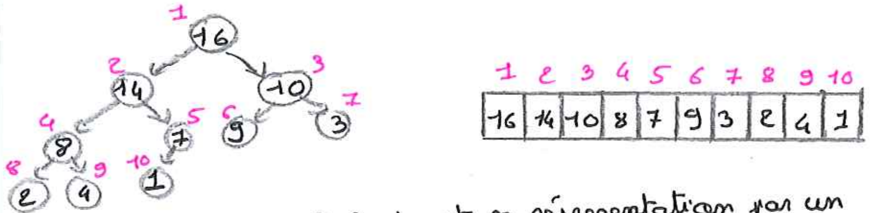
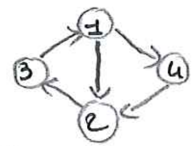


Figure 5 : Un tas binaire et sa représentation par un tableau.

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$: matrice d'adjacence



$\{(1,2) ; (1,4) ; (2,3) ; (3,1) ; (4,2)\}$: liste d'adjacence

Figure 6 : Un graphe orienté et ses types concrets.

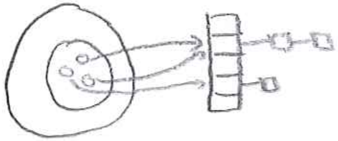
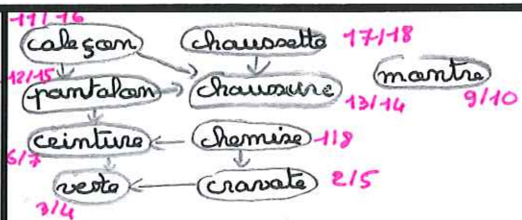


Figure 11 : Une table de hachage à listes simplement chaînées.



chaussette
caleçon
pantalon
chaussure
montre
chemise
ceinture
cravate
veste

Figure 7 : Tri topologique

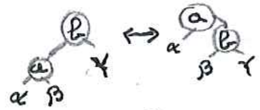


Figure 8 : Une rotation droite

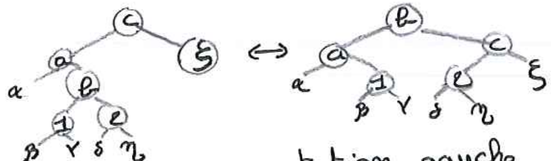
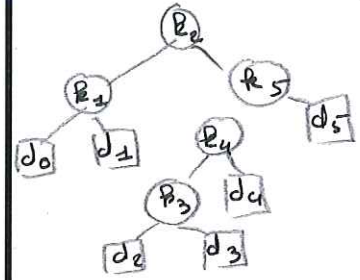


Figure 9 : Une rotation gauche



i	0	1	2	3
p_i		0,15	0,10	0,05
q_i	0,05	0,10	0,05	0,05

i	4	5
p_i	0,10	0,20
q_i	0,05	0,10

Figure 10 : Un arbre binaire de recherche optimal

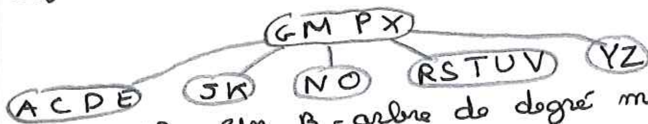


Figure 12 : Un B-arbre de degré minimal $t = 3$.

Références :

[1] Beauquier, Berstel, Chrétienne, Élément d'algorithmique
[2] Cormen, Algorithmique
[3] Froidevaux, Gaudel, Type de données et algorithmes