901

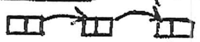
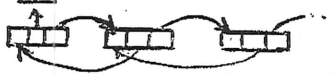
Structures de données : exemples et applications

Motivations : choisir une structure de données adaptée à un problème algorithmique permet de faciliter la conception et/ou d'améliorer sa complexité.

I) Types de données [Be] + [Fr]

Déf : Les types constituent une description du format de représentation interne des données en machine.

Intérêt : ne pas se soucier de la représentation physique des données.

Exemples : * les types de base donnés par le langage de programmation sont généralement : entiers, booléens, caractères...
* certains langages proposent des types plus avancés :
  - → tableau  (taille finie)
  - → liste chaînée 
  - → liste doublement chaînée 

Déf : * Un type de donnée abstrait est la description d'un ensemble organisé d'objets et des opérations de manipulation sur cet ensemble.
* Une structure de donnée correspond à la description des types abstraits et à leur implémentation.

→ Description des types abstraits :
- Signature { sortes : nom d'ensemble de valeurs ; opérations et leur profil }
- Préconditions = domaine de déf. des opérations
- Axiomes = propriétés des opérations

Exemple : pour le type "dictionnaire"
* sorte : dico ; utilise booléen et chaîne caractère
* opérations : dicovide : → dico
  - est-dicovide : dico → booléen
  - chercher : ch. carac × dico → booléen
  - insérer : ch. carac × dico → dico
  - supprimer : ch. carac × dico → dico
* Préconditions : • supprimer(x, d) ssi chercher(x, d) = vrai
* Axiomes : • est-dicovide (dicovide) = vrai

→ Implémentation

Le concept de type abstrait permet de concevoir un algorithme seulement à partir des signatures des types de données utilisés, et de leur donner ensuite une représentation. (démarche dite descendante)

Rq : la conception est alors plus simple puisqu'on n'a pas à prendre en compte les détails de représentation.

II) Structures adaptées à la collection d'objets

But : insérer et supprimer des éléments d'un ensemble.

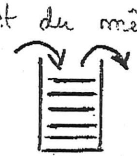
1) Structures séquentielles [Fr]

* **Liste** : c'est une suite finie d'éléments où les insertions et suppressions se font aux extrémités et à l'intérieur de la liste.

→ Signature : sorte : liste, utilise booléen, élément
opérations : accès $k^{ème}$ élt, insérer élt $k^{ème}$ pos, supprimer $k^{ème}$ élt.

→ Implémentation : • tableau (⊕ accès $k^{ème}$ élt ; ⊖ mémoire, ajout)
• liste chaînée (⊕ pas de longueur max ; ⊖ ajout, suppression, accès)

* **Pile** : c'est une suite finie d'éléments où les insertions et suppressions se font du même côté (LIFO)



Ex : pile d'assiettes

→ Implémentation :

- tableau avec un indice de sommet de pile
- liste chaînée

* **File** : les insertions se font à une extrémité et les suppressions à l'autre. (FIFO)

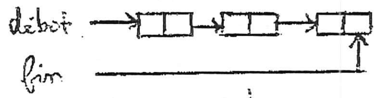
Ex : file d'attente.

→ Implémentation :

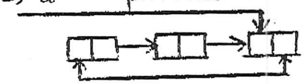
- tableau avec deux indices début et fin

début fin

- liste chaînée : → à deux pointeurs



→ à un pointeur



Appl : les parcours de graphe en profondeur (resp largeur) nécessitent un stockage dans une pile (resp file)

2) Structures arborescentes [F1]

* **Graphe** : Un graphe orienté G (resp. non orienté) est un couple (S, A) où S est un ensemble fini de sommets et A un ensemble fini d'arcs (resp. arêtes)

Ex : réseau routier, ordonnance de tâches

→ Signature : sorte : Graphe
opérations : est_vide, ajouter sommet/arc, supprimer sommet/arc, est_un_sommet/arc, précédent, successeur.

→ Implémentation :

- Matrice d'adjacence $M_{ij} = \begin{cases} 1 & \text{s'il existe un arc } (i,j) \\ 0 & \text{sinon} \end{cases}$
- Liste d'adjacence : à chaque sommet, on associe la liste de ses successeurs.

Rq : la matrice est plus utile pour un graphe "plein" et la liste d'adjacence l'est pour un graphe à peu d'arcs.

Appl : • calcul des composantes fortement connexes (Kosaraju)
- plus court chemin : algo. de Djikstra
- arbre couvrant minimal (algo de Prim et de Kruskal)

* **Arbre** : c'est un graphe connexe sans cycle dont l'un des sommets est appelé racine et est distingué

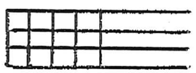
Ex : généalogie, fichiers système

Cas particulier : arbres binaires, définis récursivement
→ soit vide
→ soit de la forme (O, B_1, B_2) où O est la racine et B_1, B_2 deux AB disjoints.

→ Signature : Sorte : arbre, noeud
opérations : arbre_vide, constructeur, racine, fils gauche, fils droit, contenu_noeud

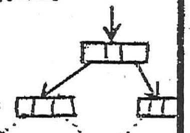
→ Implémentation :

- tableau



← contenu du noeud
← indice fg
← indice fd

- pointeurs vers les ss.arbres gauches et droits



Appl : · tri par tas
· Codage de Huffman
- file de priorité

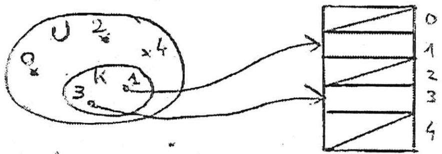
III) Structures de données adaptées à la recherche

But : En plus de l'insertion et de la suppression, on souhaite pouvoir rechercher un élément efficacement.

1) Table de hachage [Co]

→ dans un ensemble d'objets sans ordre

* Table à adressage direct : Soit U un univers de clés et $K \subseteq U$



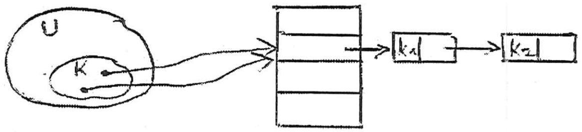
Chaque élément est associé à un indice de la table.

Les opérations se font en $O(1)$, mais si $|U|$ est grand ou si $|K|$ est petit comparé à $|U|$, cela ne s'avère pas efficace.

* On utilise donc une fonction de hachage h pour calculer l'alvéole à partir de la clé : h établit une correspondance entre l'univers des clés U et la table de hachage $T[0 ; .. m-1]$.

Si $m < |U|$, h n'est pas injective : plusieurs clés ont même image par h, entraînant une collision.

* Résolution par chaînage : les éléments d'une même alvéole sont placés dans une même liste chaînée.



Une recherche d'élément se fait alors en un temps moyen $O(1+\alpha)$ si l'on suppose que chaque élément a une chance égale d'être haché vers l'une des alvéoles, (hachage uniforme) et où $\alpha = \frac{n}{m}$ le facteur de remplissage.

Application : hachage parfait [DVP 1]

2) Arbre binaire de recherche [Co]

→ dans un ensemble d'objets totalement ordonné

Déf : Un arbre binaire de recherche (ABR) est un arbre binaire tel que pour tout noeud x :

- si y est un noeud du sous-arbre gauche de x, alors $clé[y] \leq clé[x]$
- si y est un noeud du sous-arbre droit de x, alors $clé[y] \geq clé[x]$

Les opérations rechercher, minimum, maximum, successeur et prédécesseur peuvent se faire en temps $O(h)$ où h est la hauteur de l'ABR.

Rq : c'est en $O(n)$ dans le pire des cas pour un arbre classique à n noeuds.

L'idée est donc d'équilibrer l'arbre du mieux possible pour assurer une hauteur minimale, $h = O(\log_2 n)$

Application : [arbres AVL et arbres rouges et noirs : [DEV 2]
comparaison de la hauteur minimale

Conclusion : il faut choisir une structure de donnée selon l'utilité que l'on en a.

Il existe d'autres structures de plus en plus complexes ou adaptées à un problème précis (ex : les tables dynamiques pour pallier à la rigidité d'un tableau, la structure "union-find" pour représenter les classes d'équivalence, ...)

Références : - Cormen [Co], Froidevaux [Fr], Beauquier [Be]