901 — Structures de données, exemples et applications.

Les types (concrets) sont utiles pour structurer les données et repérer certaines erreurs de programmation. Les types abstraits encapsulent les types concrets, permettant de les utiliser comme des boîtes noires, c-à-d sans en connaître l'implémentation.

Mais l'efficacité dépend de l'implémentation, c'est pourquoi nous présentons ici des structures de données efficaces respectant le cahier des charges qu'est le type abstrait.

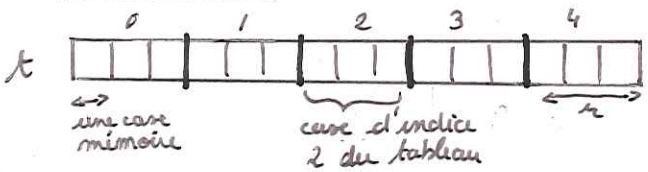
Déf Un type concret est la description d'un format de représentation internes des données en machine [BBC] p 37

Déf Un type abstrait (de données) est la description d'un ensemble de données et des opérations que l'on peut y appliquer. [BBC] p 38.

Déf Une structure de données pour un type abstrait est la donnée d'un type concret et des implémentations des différentes fonctions associées.

I. Structures de base

1) Tableaux



exemple de tableau de taille 5 pour un type codé sur 3 cases mémoire

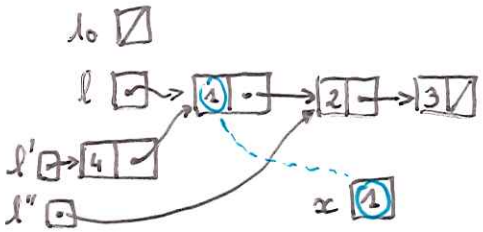
↪ On peut accéder à la i-ème case du tableau en temps constant, car il suffit de lire $i \times s$ cases plus loin.

Rq En doublant la taille du tableau quand il est plein, et en la divisant par deux quand moins d'un quart du tableau est plein on implémente de manière efficace une table dynamique. La lecture et l'écriture ont alors une complexité amortie de $O(1)$.

2) Listes

liste simplement chainée.

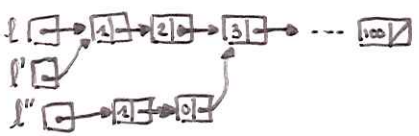
l0 = créer_liste_vide()
l = [1;2;3]
l' = ajouter(4, l)
l'' = queue(l)
x = tête(l)



Rq

PERSISTANT

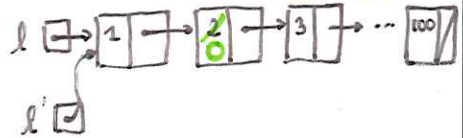
l = [1;2;...100] ; l' = l ; l'' = remplacer(l, 2, 0)



Deux listes persistantes ayant la même fin peuvent partager de la mémoire car une modification sur l'une n'affecte pas l'autre.

MUTABLE

l = [1;2;...100] ; l' = l ; remplacer(l, 2, 0)



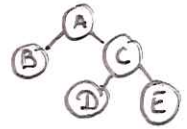
Vs deux listes mutables partagent de la mémoire, une modification sur l'une peut affecter l'autre car ici on ne recopie pas, on change.

Rq Existent aussi les listes doublement chaînées, cycliques... Vs annexe 1.

Rq Une liste paresseuse p est donnée par une fonction f qui en énumère les éléments, et une liste mutable l qui contient les éléments déjà calculés. Pour faire une opération sur p on la fait sur l en lui ajoutant si possible et si nécessaire des éléments calculés par f.

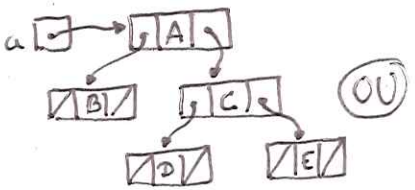
3) Arbres binaires

ex



2 implémentations possibles.

[Froi] p 112. p 111.



	0	1	2	3	4	5	6	7	
	A			C	B		D	E	élémᵗ
	4			6	/		/	/	fils gauche
	3			7	/		/	/	fils droit

{0, 2, 5, ...} liste des cases libres.

implémentation avec des pointeurs

implémentation avec un tableau d'indices.

II Ordonnancement

1) Piles

[Froi] p 74
[BBC] p 40

```
créer_vide () -> pile
empiler (x: X ; p: pile) -> pile
est_vide (p: pile) -> booléen
sommet (p: pile) -> X
dépiler (p: pile) -> X
```

- implémentation avec une liste
  ↳ toutes les opérations se font alors en temps constant
- application au parcours en profondeur (de graphes).

2) Files

[Froi] p 77
[BBC] p 42

```
créer_vide() -> file
enfiler (x: X, f: file) -> file
est_vide (f: file) -> booléen
tête (f: file) -> X
défiler (f: file) -> X
```

- implémentation avec une liste conservant un pointeur vers la dernière cellule (If annexe 2)
  ↳ toutes les opérations se font alors en temps constant.
- applica° -> parcours en largeur

3) Files de priorité (MIN)

Une file de priorité représente un ensemble d'éléments munis de clés dans un ensemble totalem^t ordonné.
Ici nous nous intéressons aux files de priorité MIN, dans lesquels l'élément de clé minimale est privilégié.

[BBC] p 56 → tas binaire → [Froi] p 343

```
créer_vide () -> file de priorité (FDP)
insérer (x: X ; f: FDP) -> FDP
est_vide (f: FDP) -> booléen
min (f: FDP) -> X
extraire_min (f: FDP) -> X
diminuer_clé (c: clé, x: X ; f FDP) -> FDP
```

- implémentation par tas binaire
  ↳ creer_vide, est_vide, min en $O(1)$
  ↳ insérer, extraire_min en $O(\log(n))$
  ↳ diminuer clé en $O(\log(n))$ si on sait aller d'un élmt à son noeud en $O(1)$ — avec un tableau par exemple si les élmts sont $[1..m]$ — en $O(n)$ sinon

Applications
- Construction d'un arbre couvrant de coût MIN avec Prim [Co] p 583 [Lo] p 609
- Plus court chemin à une source avec Dijkstra

Rq La structure de tas sert aussi pour le tri par tas, mais pour utiliser pleinement cette structure il faut ajouter à l'interface une fonction de création de tas à partir d'une liste, ce qui se fait alors en O(longueur de la liste).

Rq Avec les tas de Fibonnacci diminuer_clé a une complexité amortie en $O(1)$ (ADNiS) [Co] p 467

III Ensembles et dictionnaires

DICTIONNAIRE
```
créer_vide() -> dictionnaire
est_vide (d: dico) -> booléen
chercher (c: clé, d: dico) -> X
ajouter (x: X, c: clé, d: dico) -> unité
supprimer (c: clé ; d: dico) -> unité
```

Le dictionnaire, aussi appelé tableau associatif, représente une fonction partielle des clés vers les valeurs, ci contre de type X. [Froi] p 85

Un ensemble, vu comme ensemble de définition de cette fonction partielle, est un dictionnaire dont les valeurs n'importent pas. En effet en remplaçant X par unité ci-dessus, on obtient le type ensemble.

1) Implémentation naive

On peut représenter un ensemble par une liste, auquel cas il faut vérifier à chaque ajout qu'on ne va pas créer de doublons. [Froi] p 88
De même avec une liste de couple (clé, valeur) on implémente dictionnaire.

2) Cas des clés entières

Si l'on sait que les clés seront des entiers entre 0 et N, on peut implémenter le dictionnaire par un tableau de valeurs. De même pour un ensemble inclus dans $[0..N]$ on peut utiliser un tableau de booléens indiquant si l'élément est ou non dans l'ensemble. [Froi] p 89

ex avec $N=7$, $S=\{1,2,5\}$ $f=\begin{pmatrix}1\to A\\2\to C\\5\to B\end{pmatrix}$

S :

0	1	1	0	0	1	0	0

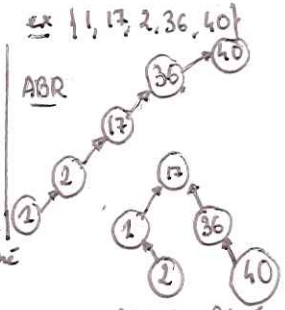
f :

/	A	C	/	/	B	/	/

3) Cas des clés ordonnées

Si les clés sont dans un ensemble totalement ordonné, on peut utiliser des autres binaires de recherche, c-à-d autre des clés dans lequel chaque noeud est plus grand que son fils gauche et strictement plus petit que son fils droit. Il existe des améliorations des ABR comme les AVL qui s'attachent à garder un autre équilibré

ex : {1, 17, 2, 36, 40}



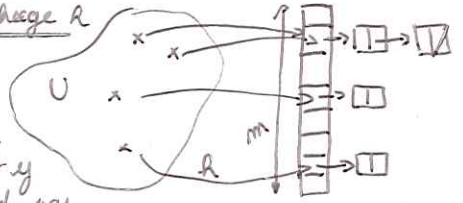
4) Tables de hachage

[Co] p 235

Lorsque les clés peuvent s'étaler dans un ensemble U très grand, la solution des tableaux n'est pas satisfaisante, voire impossible si U est infini.

On utilise alors une fonction de hachage h qui envoie U dans $[1..m]$.

Pour une clé $k \in U$ on mettra la valeur correspondante dans l'alvéole $A(k)$.



Comme h n'est pas injective il peut y avoir des collisions. On les résout par chaînage, c-à-d que chaque alvéole contient une liste chainée à laquelle on peut ajouter des éléments sans écraser les précédents.

Si K est une variable aléatoire à valeur dans U telle que $\forall i \in [0..m]\ \mathbb{P}(h(K)=i) = \frac{1}{M}$ on dit qu'elle vérifie l'hypothèse de hachage simple.

Après d'insertion de $K_1, K_2 \dots K_n$ clés aléatoires vérifiant cette hypothèse, la longueur moyenne d'une liste est $\mathbb{E}[n_j] \leq \frac{n}{m} =: \alpha =:$ le taux de remplissage.

Pour maintenir ce taux bas, on est amené à agrandir la table (ie augmenter m)
↳ complexité amortie comme pr table dynamique

5) Complexités comparées

	liste	tableau de N connu	ABR / AVL	table de hachage
espace mémoire	$O(n)$	$O(N)$	$O(n)$	$O(n+m)$
créer-vide	$O(1)$	$O(N)$	$O(1)$	$O(m)$
est-vide	$O(1)$	$O(1)$	$O(1)$	$O(m)$ donc $O(n)$
chercher	$O(n)$	$O(1)$	$O(h)$ / $O(\log(n))$	$O(1+n/m)$ en moyenne
ajouter *	$O(1)$	$O(1)$	$O(h)$ / $O(\log(n))$	$O(1)$ amorti
supprimer	$O(n)$	$O(1)$	$O(h)$ / $O(\log(n))$	$O(1+n/m)$ en moyenne

⚠ ajouter doit être appelé pour un élément non déjà présent.

IV Graphes

[Froi] p143

```
créer_vide() -> graphe
ajouter_sommet (g: graphe, s:X) => unité
ajouter_arc( g: graphe, u:X, v:X) -> unité
retirer_sommet (g: graphe, s:X) -> unité
retirer_arc( g: graphe, u:X, v:X) -> unité
est_sommet (g: graphe, s:X) -> booléen
est_arc (g: graphe, u:X, v:X) -> booléen
prédécesseurs (g: graphe, s:X) -> liste de X
successeurs (g: graphe, s:X) -> liste de X
```

$G=(S,A)$ $n=\#S$, $m=\#A$

- implémentation par une matrice d'adjacence

$$(M_{i,j})_{(i,j)\in S^2} = \begin{cases} 0 \text{ si } (i,j) \notin A \\ 1 \text{ si } (i,j) \in A \end{cases}$$

- implémentation par une table dynamique d'ensembles de successeurs implémentés par des AVL.

$\forall i \in S,\ T[i] = \{j \in S \mid (i,j) \in A\}$

Rq Lorsque le graphe est trop gros, ni par exemple c'est le graphe des états d'un jeu, mais qu'on sait calculer les successeurs, c-à-d les configurations suivantes possibles, on choisit une implémentation implicite c-à-d qu'on ne stocke pas le graphe mais qu'on sait l'explorer.

- Si l'on dispose de deux implémentations d'un même type abstrait on peut les combiner en une troisième dont les opérations de lecture prennent le meilleur complexité des deux, et celles d'écriture la pire (Cf ⓐ et ⓑ)

opération \ structure	matrice d'adjacence	tableau ⓐ d'ensembles de successeurs	tableau ⓑ d'ensembles de prédécesseurs	ⓐ et ⓑ
espace mémoire	$O(n^2)$	$O(n+m)$	$O(n+m)$	$O(n+m)$
N: est-il successeur de Nj / Ni: — prédécesseur —	$O(1)$	$O(\log(\deg^+))$	$O(\log(\deg^-))$	$O(\log(\min \deg^+, \deg^-))$
parcours des succ. / parcours des préd	$O(n)$	$O(\deg^+)$ / $O(n+m)$	$O(n+m)$ / $O(\deg^-)$	$O(\deg^+)$ / $O(\deg^-)$
ajouter_arc	$O(1)$	$O(\log(\deg^+))$	$O(\log(\deg^-))$	$O(\log(\max(\deg^+, \deg^-)))$
ajouter sommet	$O(n)$ amorti	$O(1)$ amorti	$O(1)$ amorti	$O(1)$ amorti

V Partition

[BBC] p61
[Co] p526

```
créer_vide () -> partition
créer_classe (x:X, p: partition) -> unité
trouver (x:X, p: partition) -> X
union (x:X, y:X, p: partition) -> unité
```

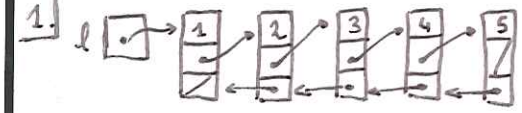
- implémentation par une forêt d'arbres ascendants Cf annexe 3.
- amélioration : compression de chemin ← DEV
- application : algorithme de Kruskal

[Papa] p132

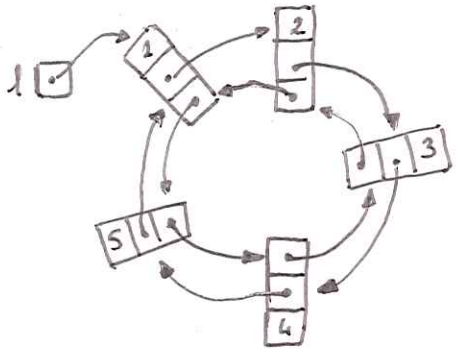
Kruskal → [Co] p583

[Co] p219

1.

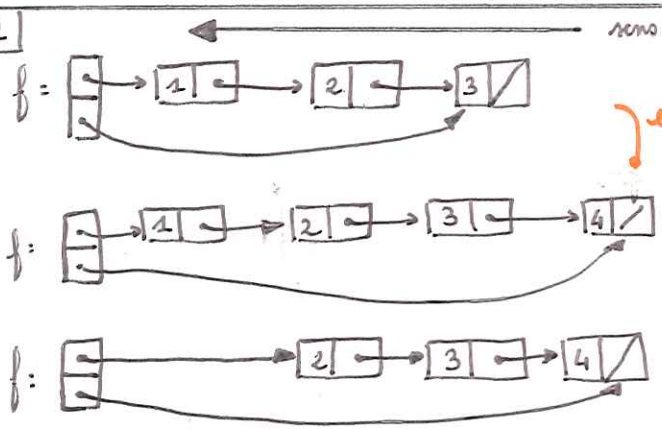


liste doublement chainée

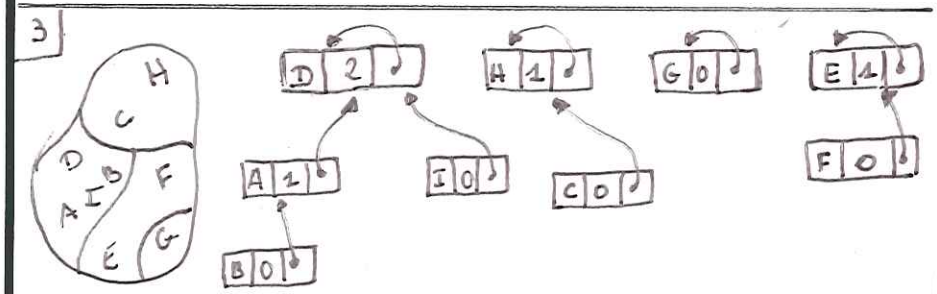


liste doublement chainée cyclique

2



3



Biblio

[BBC] Eléments d'algorithmiques, Beauquier, Berstel, Chrétienne

[Co] Algorithmique, Cormen, Leiserson, Rivest, Stein

[Fr] Types de données et algorithmes, Froidevaux, Gaudel, Soria

[Papa] Algorithms, Dasgupta, Papadimitriou, Vazirani.

[Se] Algorithms, Sedgewick, Wayne.