Implémentations et Applications des ensembles et dictionnaire

I. Interface et Implémentations Naïves [SED] [TOR]

Définition 1 **Un ensemble** est une structure de données abstraite permettant de stocker un rassemblement d'objets distincts.

Définition 2 **Un dictionnaire** , ou table d'association est une structure de données abstraite permettant de représenter des associations clé-valeur.

Remarque 3 **Équivalence Ensemble Dictionnaire** Un ensemble est un dictionnaire dont les valeurs sont des booléens et un dictionnaire est un ensemble de couple clés valeurs. Dans la suite de la leçon, on se limitera à implémenter des ensembles.

Remarque 4 On donnera les complexités des implémentations d'ensemble en fonction de « n » le cardinal des ensembles pris en entrée.

Définition 5 **Deux Interfaces d'ensemble** [SED p.489]

```
val vide    : ens
val ajoute  : ens -> 'a -> ens
val retire  : ens -> 'a -> ens
val contient : ens -> 'a -> bool
```

```
val vide    : unit -> ens
val ajoute  : ens -> 'a -> unit
val retire  : ens -> 'a -> unit
val contient : ens -> 'a -> bool
```

Remarque 6 **Une implémentation Mutable ou Immutable** [TOR 7.4 p.422] peuvent être proposé pour les ensembles. Le compromis entre mémoire utilisée, performance, facilité d'utilisation sera à prendre en compte.

Définition 7 **Un multi ensemble** [TOR] est un ensemble qui peut posséder plusieurs fois le même élément.

A. Liste Chainée

Définition 8 **Une liste Chainée** peut décrire un ensemble d'éléments de façon très primitive dont la valeur de chaque chainon est un élément de l'ensemble.

Complexité 9 Pour savoir si un élément est dedans on va devoir potentiellement parcourir toute la liste donc `ajoute` se fait en $\mathcal{O}(1)$ et `contient` en $\mathcal{O}(n)$.

Exemple 10 [SED p.] $\{1, 2, 3, 4, 5\}$ peut être réprésenté comme ceci :

$$\boxed{5} \rightarrow \boxed{3} \rightarrow \boxed{2} \rightarrow \boxed{1} \rightarrow \boxed{4}$$

B. Tableau Trié

Définition 11 **Un tableau trié** sans doublons est une manière simple d'implémenter un ensemble qui permet une recherche logarithmique par dichotomie.

Complexité 12 `ajoute` en $\mathcal{O}(n)$, `contient` en $\mathcal{O}(\log(n))$

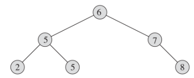
II. Implémentations classiques [SED]

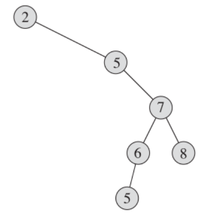
A. Arbre binaire de recherche

Définition 13 **Un Ensemble d'étiquettes** Ens d'un arbre binaire peut être défini par induction comme :

- $\mathrm{Ens}(\texttt{Vide}) = \emptyset$
- $\mathrm{Ens}(\texttt{N(g, e, d)}) = \{\texttt{e}\} \cup \mathrm{Ens}(\texttt{g}) \cup \mathrm{Ens}(\texttt{d})$

Exemple 14 [COR3 12.1]





Définition 15 **Un Arbre binaire de recherche** (ABR) est un arbre binaire où, l'étiquette `e` de chaque noeud `N(g, e, d)` est plus grande que n'importe quel élément de $\mathrm{Ens}(\texttt{g})$ et plus petite que n'importe quel élément de $\mathrm{Ens}(\texttt{d})$.

Implémentation 16 On peut implémenter un ABR de manière mutable et non-mutable. Dans le premier cas, les opérations d'insertion et de suppression effectuent des modifications en place. Dans le second cas, elles renvoient donc un nouvel ensemble. TODO « Remarque sur -ensemble- » ?

Complexité 17 Pour un ABR de hauteur h, la recherche, l'insertion et la suppression se font en $\mathcal{O}(h)$. Si l'arbre est équilibré, on a donc des complexités en $\mathcal{O}(\log n)$; cependant, dans le pire cas, on reste en $\mathcal{O}(n)$.

Application 18 **Les `Set` en OCaml** sont implémentés par des arbres binaires équilibrés. En particulier ils utilisent des AVL relachés.

Propriété 19 Avec cette implémentation, on dispose des opérations `min` et `max` en $\mathcal{O}(h)$ et on peut itérer sur l'ensemble en $\mathcal{O}(n)$ au total.

Définition 20 **Un Arbre bicolore** (ou arbre rouge-noir) est un ABR dont les nœuds sont colorés rouge ou noir, satisfaisant :

- chaque nœud est soit rouge, soit noir
- la racine et les feuilles (`Vide`) sont noires
- les enfants d'un nœud rouge sont noirs
- le chemin de la racine à n'importe quelle feuille contient toujours le même nombre de nœuds noirs. On appelle ce nombre la hauteur noire.

Propriété 21 Pour un arbre rouge-noir à n noeuds, de hauteur h, on a : $h \leq 2\log_2(n+1)$. Les opérations de recherche sont donc en $\mathcal{O}(\log n)$.

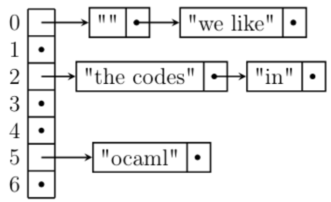
Propriété 22 On peut implémenter les opérations d'insertion et de suppression dans un arbre rouge-noir en $\mathcal{O}(\log n)$.

Application 23 **L'Algorithme d'ordonnancement de Linux** utilise un arbre rouge-noir dans l'algorithme CFS d'ordonnancement des processus.

B. Table de Hachage

Définition 24 **Une table de Hachage** par chaînage est un tableau de listes chaînées contenant des couples clé/valeur, muni d'une fonction de hachage associant à chaque clé une alvéole (cellule) du tableau.

Exemple 25 [SED] On considère des chaînes de caractères comme clés, et on prend h(w) = |w| % 7 comme fonction de hash. La table de hachage ci-contre ne contient que des clés, elle représente donc l'ensemble `{"", "we like", "the codes", "in", "ocaml"}`.



Implémentation 26 **Mutabilité** La structure de tableau est mutable, rendant ces ensemble mutables. Les opérations d'insertion et de suppression agissent en place.

Définition 27 **Les collisions** interviennent dans une table de hachage si deux éléments différents ont la même valeur de hache. Dans ce cas la recherche dans la table prendra d'autant plus de temps qu'il y a de collisions.

Remarque 28 **Les fonctions de hachages** ne se font pas en $\mathcal{O}(1)$ pour des entrées de taille quelconques. En pratique nos entrées sont bornées et nos fonctions de hachage sont très rapides donc considérés comme fait en temps constant.

Application 29 Une table de hachage est une structure très utile et flexible utilisée pour implémenter le type `Hashtbl` OCaml ou encore le type `dict` Python.

Application 30 **Les matrices éparses** contenant majoritairement des 0. Elles peuvent être représenter par l'ensemble des valeurs non nulles permet de s'éviter la majeure partie des calculs lors qu'une multiplication par exemple.

Définition 31 **Un hachage parfait** est une fonction de hachage qui ne provoque pas de collisions (accès en $\mathcal{O}(1)$ dans le pire cas). Elle nécessite cependant, au préalable, l'ensemble des éléments susceptibles d'être hashés.

Complexité 32 Si la fonction de hachage `h` a un coût constant, alors la recherche, l'insertion et la suppression sont en $\mathcal{O}(k)$ avec k le nombre maximal de collisions dans la table. Si h permet de borner k par une constante, on a donc un coût en $\mathcal{O}(1)$.

III. Implémentations particulière

A. Probabilité sur les éléments

Idée 33 Même avec un ABR équilibré, si certains noeuds sont plus utilisés que d'autres alors un arbre adapté à l'utilisation de chaque élément pourrait être plus rapide.

Définition 34 **ABR Optimaux** [COR3] En connaissant la probabilité de requête de chaque élément, on peut construire un ABR optimal, qui aura un coût en $\mathcal{O}(\log n)$ pour la recherche. DEV

Application 35 **Un serveur DNS** répond à de nombreuses requêtes sont faites avec assez peu de modification sur l'espace des adresses. On pourrait construire un ABR optimal en approximant les probabilités de chaque demande à sa fréquence.

B. Sous structure commune [OCA 11.4]

Idée 36 Certains éléments d'un ensemble peuvent contenir des sous structures égales. On va pouvoir réutiliser ses sous structures communes pour limiter notre empreinte mémoire et potentiellement mémoïser des calculs. DEV

Définition 37 **Le partage maximal** (Hash Concing) est une technique consistant à mémoïser les fonctions de créations d'ensembles. Cette méthode souvent utilisés dans des languages fonctionnels permet de limiter la mémoire utilisés.

C. Partition d'ensembles [COR3]

Idée 38 Si on travaille sur un ensemble global que l'on souhaite partionner, on peut simplement relier un élément à son parent. En remonttant la chaine de parent on trouve le représentant d'une des partitions de l'ensemble.

Définition 39 **La structure Unir et Trouver** (Union-Find) permet de travailler sur les partitions d'un ensemble. Elle dispose des opérations Unir, qui fait l'union de deux sous-ensembles et Trouver qui trouve le représentant d'un élément. DEV

Implémentation 40 On implémente classiquement cette structure avec une forêt dont les arbres correspondent aux ensembles disjoints. Une telle forêt peut être représentée par un tableau de parenté.

Remarque 41 **Union par rang :** Lors de l'union, on choisit la racine parmi deux candidats de façon à limiter la hauteur maximale. Unir et Trouver sont alors de complexité $\mathcal{O}(\log(n))$ dans le pire cas.

Implémentation 42 **La compression de chemin** est un mécanisme effectué lorsqu'on trouve un « raccourci vers la racine », on peut remplacer le parent. La complexité amortie devient alors quasi-constante. On atteint alors une complexité amortie en $\mathcal{O}(\log *(n))$ (Admis).

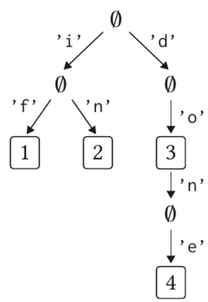
Application 43 L'algorithme de Kruskal utilise la structure Unir et Trouver pour construire itérativement un arbre couvrant minimal. Cette algorithme manipule une partition des noeuds d'un graphe d'où l'interêt de la structure.

D. Ensembles de mots [TOR 7.3.5]

Idée 44 On range de façon naturelle les mots par rapport à leur première lettre dans un dictionnaire. On peut implémenter un ensemble de mots avec un arbre qui parcoure les lettres au fur et à mesure.

Définition 45 **Un Arbre Préfixe** est un arbre dont chaque branche est étiquetée par une lettre et chaque nœud contient une valeur si la séquence de lettres menant de la racine à ce nœud est une entrée

Exemple 47 [TOR p412]



Implémentation 46 **Recherche/Ajout d'une clef s** La recherche consiste à descendre dans l'arbre en suivant les lettres de l'entrée `s`.

Complexité 48 Les opérations de recherche et d'ajout ont une complexité en $\mathcal{O}(|s|)$ la longueur de la clef `s`

Application 49 Les arbres préfixes sont utilisés dans l'autocomplétion, mais aussi en bio-informatique notamment pour implémenter des anti-dictionnaires.

Implémentations et Applications des ensembles et dictionnaire

I. Interface et Implémentations Naïves [SED] [TOR]

1 Def Un ensemble
2 Def Un dictionnaire
3 Rem Équivalence Ensemble Dictionnaire
4 Rem
5 Def Deux Interfaces d'ensemble [SED p.489]
6 Rem Une implémentation Mutable ou Immutable [TOR 7.4 p.422]
7 Def Un multi ensemble [TOR]

A. Liste Chainée

8 Def Une liste Chainée
9 Complex
10 Ex [SED p.]

B. Tableau Trié

11 Def Un tableau trié
12 Complex

II. Implémentations classiques [SED]

A. Arbre binaire de recherche

13 Def Un Ensemble d'étiquettes
14 Ex [COR3 12.1]
15 Def Un Arbre binaire de recherche
16 Implem
17 Complex
18 App Les `Set` en OCaml
19 Prop
20 Def Un Arbre bicolore
21 Prop
22 Prop
23 App L'Algorithme d'ordonnancement de Linux

B. Table de Hachage

24 Def Une table de Hachage
25 Ex [SED]
26 Implem Mutabilité
27 Def Les collisions
28 Rem Les fonctions de hachages
29 App
30 App Les matrices éparses
31 Def Un hachage parfait
32 Complex

III. Implémentations particulière

A. Probabilité sur les éléments

33 Idée
34 Def ABR Optimaux [COR3]
35 App Un serveur DNS

B. Sous structure commune [OCA 11.4]

36 Idée
37 Def Le partage maximal

C. Partition d'ensembles [COR3]

38 Idée
39 Def La structure Unir et Trouver
40 Implem
41 Rem Union par rang :
42 Implem La compression de chemin
43 App

D. Ensembles de mots [TOR 7.3.5]

44 Idée
45 Def Un Arbre Préfixe
47 Ex [TOR p412]
46 Implem Recherche/Ajout d'une clef s
48 Complex
49 App

Dans le programmes

- Arbre Bicolore
- Arbre binaire de recherche

Remarque

Idée à faire passer aux élèves
- Distinction Interface implémentation
- Mutable Immutable
- Les AVL ont été enlevé pour développer les arbres rouge-noir car ils sont au programme. ⚠️ Attention à bien connaitre les optimisations possibles des arbres rouges noir (insertion et suppression en $\log(n)$).

Questions

- Algorithme où on peut utiliser un multi-ensemble : tri comptage.
- Decrire les opérations d'optimisations d'un arbre bicolore.

Bibliographie

[SED] R. Sedgewick & K. Wayne, *Algorithms (4th edition)*.

[TOR] T. Balabonski & S. Conchon & J. Filliâtre & K. Nguyen & L. Sartre, *MP2I MPI, Informatique Cours et exercices corrigés*.

[COR3] T. H. Cormen, *Introduction à l'algorithmique (3rd édition)*.

[OCA] S. Conchon & J. Filliâtre, *Apprendre à programmer avec Ocaml*.

Paul Adam - Alexis Hamon