Implémentations et applications des ensembles et des dictionnaires

Niveau : MP2I/MPI, selon les parties Prérequis : structures de données basiques.

Motivation : Les ensembles et dictionnaires sont des structures de données couramment utilisées. Celles-ci sont directement incorporées dans Python et font partie de la librairie standard d'Ocaml.

I Ensembles et dictionnaires : abstractions

Définition 1 Un ensemble est une structure de données permettant de gérer une collection S d'éléments distincts. Les opérations de base sur un tel ensemble sont :
- RECHERCHER(e) : vérifie si $e \in S$
- INSERER(e) : insère e dans l'ensemble S.
- SUPPRIMER(e) : supprime e de l'ensemble S.

Définition 2 : Soit C un ensemble de clés et V un ensemble de valeurs, Un dictionnaire est un tableau associatif T de C vers V. Les opérations de base sur un dictionnaire sont :
- RECHERCHER(c) : s'il existe, retourne $v \in V$ tel que $T(c) = v$
- INSERER(c, v) : assigne à T(c) la valeur v
- SUPPRIMER(c) : supprime la valeur de T(c)

Remarque 3 : Un dictionnaire peut aussi être vu comme un ensemble de couples (clé, valeur) tel qu'une clé $c \in C$ n'apparaisse qu'au plus une fois dans l'ensemble.
Cette similarité entre ensemble et dictionnaire se retrouve aussi dans leurs implémentations respectives : par exemple set() et dict() en Python sont implémentés de la même manière.

II Tables de hachage

1) Tables à adressage direct

On se place dans le cadre de l'implémentation d'un dictionnaire dont les clés sont des entiers.

Définition 4 : Soit D un dictionnaire sur l'ensemble de clés $[\![0, m-1]\!]$.
Une table à adressage direct est un tableau T de taille m tel que pour tout $(c,v) \in D$, $T[c]$ est un pointeur vers v, et si $i \in [\![0, m-1]\!]$ n'est pas une clé de D, alors $T[c] = \text{NULL}$.

Propriétés : Une table à adressage direct implémente un dictionnaire, et par extension un ensemble, avec une complexité en temps de $\mathcal{O}(1)$ pour les opérations de base, mais une complexité en mémoire de $\mathcal{O}(m)$.

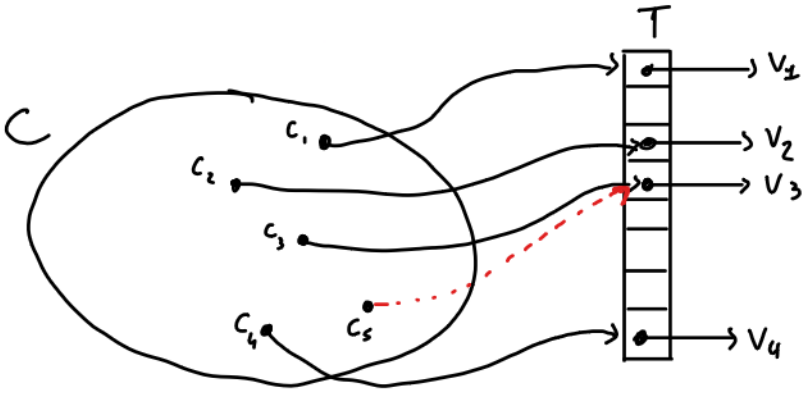
2) Tables de hachage

On considère maintenant un ensemble de clés quelconques C.

Définition 6 : Une fonction de hachage est une application
$$h : C \to [\![0, m-1]\!]$$

Principe 7 : Etant donné un dictionnaire D, pour tout couple $(c,v) \in D$, on veut stocker v dans un tableau T de taille m, à l'indice $h(c)$. C'est ce qu'on appelle le hachage

Problème : Que se passe-t-il lorsque deux clés distinctes vérifient $h(c_1) = h(c_2)$? C'est ce que l'on appelle une collision



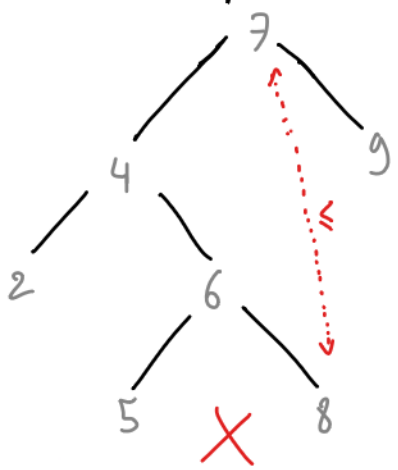
Activité 8 : Séance de TP, implémentation en C/Ocaml d'une table de hachage avec chaînage pour gérer les collisions. Mise en avant de l'importance d'un bon choix de fonction de hachage.

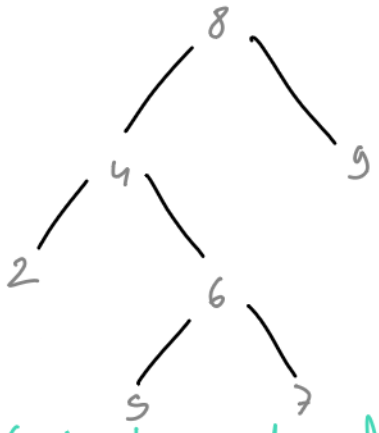
II) Implémentations avec des arbres

1) Arbres binaires de recherche

On se place dans le cadre de l'implémentation d'un ensemble S dont les éléments sont muni

Définition 9 : Un arbre binaire de recherche est un arbre binaire dont la racine est inférieure à tous les éléments du sous-arbre droit et supérieure à tous les éléments du sous-arbre gauche





✓ arbre binaire de recherche

Propriété 10 : Un arbre binaire de recherche implimente un ensemble S avec une complexité en temps de $O(h)$ pour les opérations de base, où h est la hauteur de l'arbre.

```
let rec find (x:'a) (a:'a arbre) = match a with
  | V -> raise Not_found
  | N(l,e,r) -> if e==x then true else begin
      if e<x then find x r else find x l end

let rec delete_left (a:'a arbre) = match a with
  | N(V,e,r) -> (r,e)
  | N(l,e,r) -> let l',rem = delete_left l in (N(l',e,r), rem)

let delete_racine (a:'a arbre) = match a with
  | N(l,e,V) -> l
  | N(l,e,r) -> let (r',e') = delet_left r in N(l,e',r')
```

Propriété 11 La hauteur minimale d'un arbre à n noeuds est $\lfloor \log_2(n) \rfloor + 1$.

2) Arbres AVL

Définition 12 : Un noeud d'un arbre est dit équilibré si la hauteur de ses deux sous-arbres diffèrent au plus de 1.

Propriété 13 : Un arbre dont tous les noeuds sont équilibrés est de hauteur logarithmique.

DEV Définition et implémentation d'arbres AVL en Ocaml

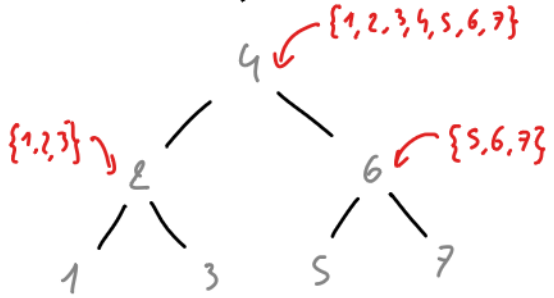
3) Applications

- Le module Set d'Ocaml utilise une structure de donnée très proche des AVL, et implémente donc les opérations de base sur les ensembles en $O(\log n)$

Remarque 14 : Cette complexité est supérieure à celles d'implémentations utilisant des tables de hachage, atteignant une complexité en moyenne en $O(1)$. Pourquoi un tel choix ?

Exercice Si on veut manipuler plusieurs ensembles, par exemple $\{1,2,3\}$, $\{5,6,7\}$ et $\{1,2,3,4,5,6,7\}$ et qu'ils sont implémentés par des table de hachage, les éléments 1, 2, 3, 5, 6 et 7 sont stockés chacun deux fois.
Expliquer en quoi une implémentation avec des arbres binaires de recherche permet d'éviter ce problème.

Solution



On parle de structure de données partagées !

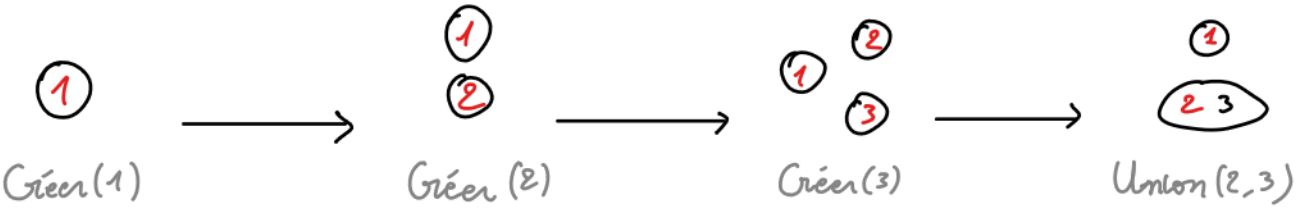
Un autre avantage des arbres binaires de recherche est qu'ils maintiennent les éléments ordonnés. Par exemple, on peut énumérer les élément dans l'ordre croissant en temps linéaire.

II] Ensembles disjoints

1) Définition

Définition 15 : Une structure d'ensembles disjoints gère une collection S d'ensemble dynamiques disjoints. Chacun de ces ensembles est identifié par un représentant, qui est un élément de cet ensemble. Les 3 opérations de base de cette structure sont :

- Créer(x) : crée un ensemble contenant x uniquement.
- Trouver(x) : renvoie le représentant de l'unique ensemble contenant x.
- Union(x,y) : réuni les deux ensembles contenant x et y.



On se limite au cas où les éléments sont dans $[\![1, n]\!]$

2) Implémentation naïve avec un tableau

Principe 16 : On maintient un tableau p de taille n tel que p[x] est le représentant de x.

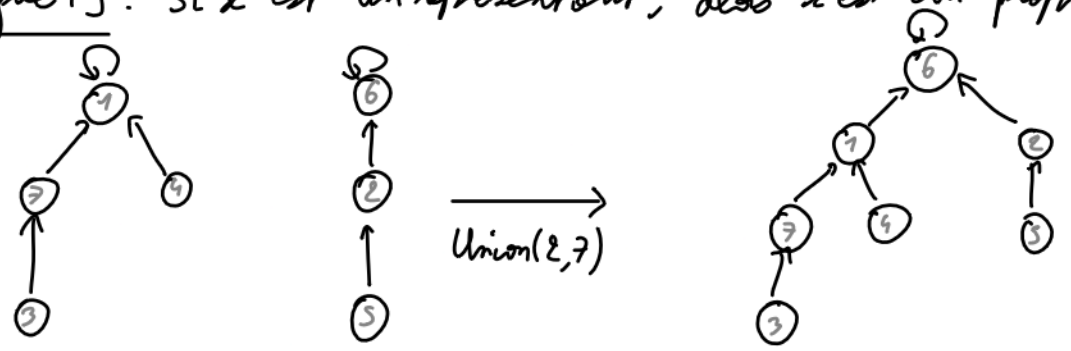
Propriété 17 : Cette structure implémente des ensembles disjoints de $[\![1, n]\!]$ avec Trouver en $O(1)$, Union en $O(n)$ et Créer en $O(1)$

```
Créer(x) :
    p[x] ← x

Trouver(x) :
    renvoyer p[x]

Union(x, y) :
    rep_y = Trouver(y)
    Pour i allant de 1 à n :
        Si p[i] = rep_y :
            p[i] = p[x]
```

3) Implémentation par des structures arborescentes

Principe 18 : On représente chaque ensemble par un arbre, dont la racine est le représentant. En pratique, on maintient un tableau p de taille n tel que p[x] est le parent de x dans l'arbre.

Remarque 19 : Si x est un représentant, alors x est son propre parent.



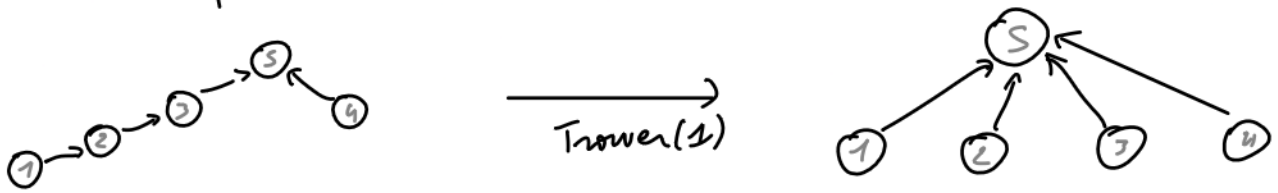
p = | 1 | 6 | 7 | 1 | 2 | 6 | 1 |
1 2 3 4 5 6 7

p = | 6 | 6 | 7 | 1 | 2 | 6 | 1 |
1 2 3 4 5 6 7

Propriété 20 : Cette structure de données implémente des ensembles disjoints avec Trouver(x) en $O(h(S_x))$ et Union(x,y) en $O(\max(h(S_x), h(S_y)))$

3) Union par rang et compression de chemins

Définition 21 : Lorsqu'on cherche le représentant de x, la compression de chemin consiste à remonter l'arbre afin de trouver $\hat{x}$, le représentant de x, et, de faire de $\hat{x}$ le parent de tous les noeuds rencontrés.



Définition 22 Lorsde l'union de deux ensembles ayant pour représentant x et y, l'union par rang consiste en la liaison de l'arbre le moins grand vers le plus grand. Comme le maintien de la hauteur de chaque représentant est trop couteux, on maintient un rang, majorant de cette hauteur.

```
Trouver(x):
    si x ≠ p[x]:
        p[x] = Trouver(p[x])
    renvoyer p[x]
```

```
Union(x,y):
    Lier(Trouver(x), Trouver(y))
```

```
Lier(x,y):
    Si rang[x] > rang[y]:
        p[y] ← x
    Sinon:
        p[x] ← y
        Si rang[x] = rang[y]:
            rang[y] += 1
```

DEV Complexité amortie de UnionFind et application à Kruskal.

plan inspiré (beaucoup) de celui présenté par Gaëlan Houser