6.- Implémentations et applications des ensembles et des dictionnaires

Niveau : MP2I/MPI selon les parties

Prérequis : Bases des structures de données

Motivation : Les ensembles et les dictionnaires sont des structures de données couramment utilisées. Elles sont par exemple présentes dans les librairies standard de Python et OCaml.

I) Interfaces des ensembles et dictionnaires

Definition 1 : Un ensemble est une structure de données permettant de manipuler une collection d'éléments distincts. Les opérations de base sur les ensembles sont :

- rechercher(S, e) : vérifie si $e \in S$
- insérer(S, e) : insère l'élément e dans S.
- supprimer(S, e) : supprime e de S.

Définition 2 : Soient C un ensemble de clefs et V un ensemble de valeurs. Un dictionnaire est un tableau associatif de C vers V. Les opérations de base sur les dictionnaires sont :

- rechercher(T, c) : s'il existe, renvoie $v \in V$ tel que $T(c) = v$.
- insérer(T, c, v) : assigne à c la valeur v dans T
- supprimer(T, c) : supprime la valeur associée à c dans T.

Remarque 3 : On peut comparer naturellement les ensembles à ceux que l'on rencontre en mathématiques, et les dictionnaires à ceux que l'on retrouve pour les langues.

Remarque 4: Un dictionnaire peut être vu comme un ensemble et inversement : on peut implémenter un dictionnaire comme un ensemble de couples (clef, valeur) tel qu'une même clef n'apparaît jamais plus d'une fois, et un ensemble par un dictionnaire dont l'ensemble de départ est l'ensemble initial et l'ensemble d'arrivée les booléens. On retrouve cette similarité dans leurs implémentations : en Python, set et dict sont implémentées de la même façon.

II) Implémentations par des tables

1.) Tables à adressage direct

Définition 5: Soit D un dictionnaire sur l'ensemble des clefs $[\![0;m-1]\!]$.
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 clef de D, alors $T[i]=$ NULL.

Propriété 6: Une table à adressage direct implémente un dictionnaire (et par extension un ensemble), avec une complexité en temps de $O(1)$ pour les opérations de base, mais une complexité mémoire de $O(m)$.

2.) Tables de hachage

Définition 7: Une fonction de hachage est une fonction d'un ensemble C à $[\![0;m-1]\!]$ pour un m fixé.

Proposition 8: Soit D un dictionnaire. Pour tout couple $(c,v)\in D$, on veut stocker v dans un tableau T de taille m à l'indice $h(c)$. On appelle cela le hachage.

Définition 9: On appelle collision le fait que deux clefs c_1 et c_2 distinctes soient telles que $h(c_1)=h(c_2)$.

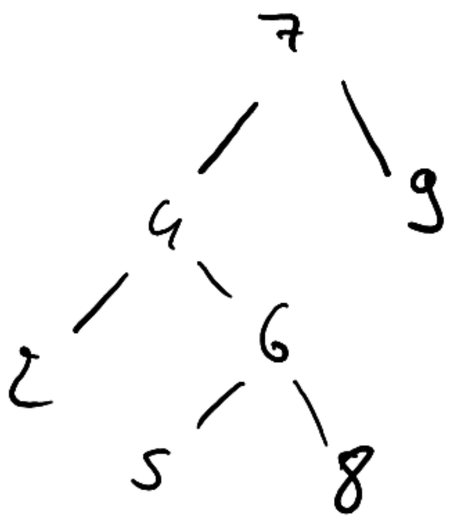
Activité 10 : TP d'implémentation de tables de hachage en C et OCaml. On gèrera les collisions par des listes chaînées et on mettra en avant le choix d'une bonne fonction de hachage.

III) Implémentation par des arbres

1.) Arbres binaires de recherche

Definition 11: Un arbre binaire de recherche est un arbre binaire où chaque noeud est supérieur à tous les éléments de son sous-arbre gauche et inférieur à tous les éléments de son sous-arbre droit.

Exemple 12: Arbre binaire de recherche



n'est pas un arbre binaire de recherche car $8 > 7$

est un arbre binaire de recherche.

Propriété 13: Un arbre binaire de recherche implémente un ensemble avec une complexité en temps de $O(h)$ pour les opérations de base, où h est la hauteur de l'arbre.

Implémention 14: Partie du code OCaml implémentant les arbres binaires de recherche.

```
type 'a abr = F | N of 'a abr * 'a * 'a abr

let rec find (a:'a abr) (x:'a) :bool = match a with
  | F -> false
  | N(l,y,r) -> x = y || if x<y then find l x else find r x
```

Propriété 15 : La hauteur minimale d'un arbre binaire de recherche à n noeuds est $\lfloor \log_2(n) \rfloor + 1$.

2.) Arbres AVL

Définition 16 : Un noeud d'un arbre est équilibré si la hauteur de ses deux sous-arbres diffère d'au plus 1.

Propriété 17 : Un arbre dont tous les noeuds sont équilibrés est de hauteur logarithmique en son nombre de noeuds.

DEV : Définition et implémentation des arbres AVL en OCaml

3.) Applications

Proposition 18 : 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 $O(\log(n))$ où n est le nombre d'éléments.

Remarque 19 : Cette complexité est supérieure à celles d'implémentations utilisant des tables de hachage, ayant une complexité temporelle moyenne de $O(1)$. Pourquoi un tel choix ?

Exercice 20 : On souhaite manipuler plusieurs ensembles, par exemple $\{1,2,3\}$, $\{4,5,6\}$ et $\{1,2,3,4,5,6\}$. S'ils sont représentés par des tables de hachage, chaque élément sera stocké deux fois en mémoire. Comment éviter ce problème à l'aide des arbres binaires de recherche ?

Solution : $\{1,2,3\}$ → 2, $\{4,5,6\}$ → 6, $\{1,2,3,4,5,6\}$ → 4 ; 1, 3, 5, 7 sont les feuilles.

On parle de structures de données partagées.

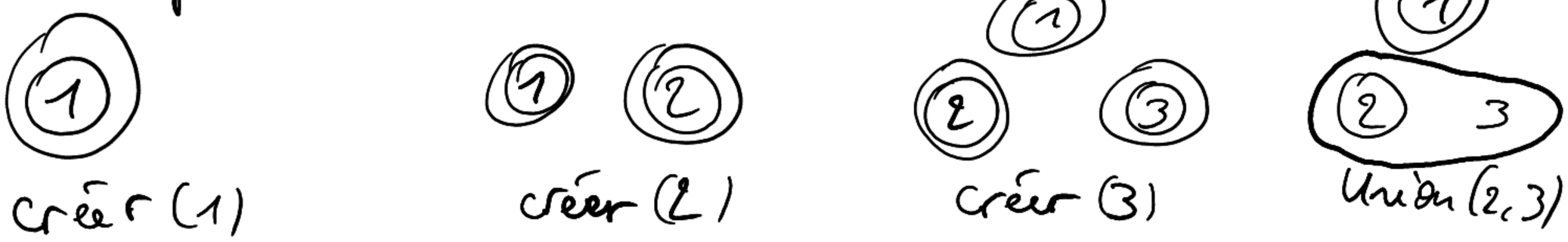
Remarque 21 : Les arbres binaires de recherche permettent également de retrouver facilement les éléments de manière ordonnée par un parcours infixe.

IV) Ensembles disjoints

1. Interface des ensembles disjoints

Définition 22 : Une structure d'**ensemble disjoints** gère une collection S d'ensembles dynamiquement disjoints. Chacun de ces ensembles est identifié par un **représentant**, qui est un élément de cet ensemble. Les opérations de base sont :
- créer(x) : renvoie un ensemble contenant uniquement x.
- trouver(x) : renvoie le représentant de l'unique ensemble contenant x
- union(x,y) : réunit les deux ensembles contenant x et y.

Exemple 23 : Opérations sur les ensembles disjoints
Les représentants sont entourés.



2.) Implémentation avec des tableaux

Propriété 24 : Un tableau p implémente une structure d'ensembles disjoints, où $p[x]$ est le représentant de x. On a alors créer et trouver avec une complexité temporelle de $O(1)$, et union de $O(n)$ où n est la taille de p.

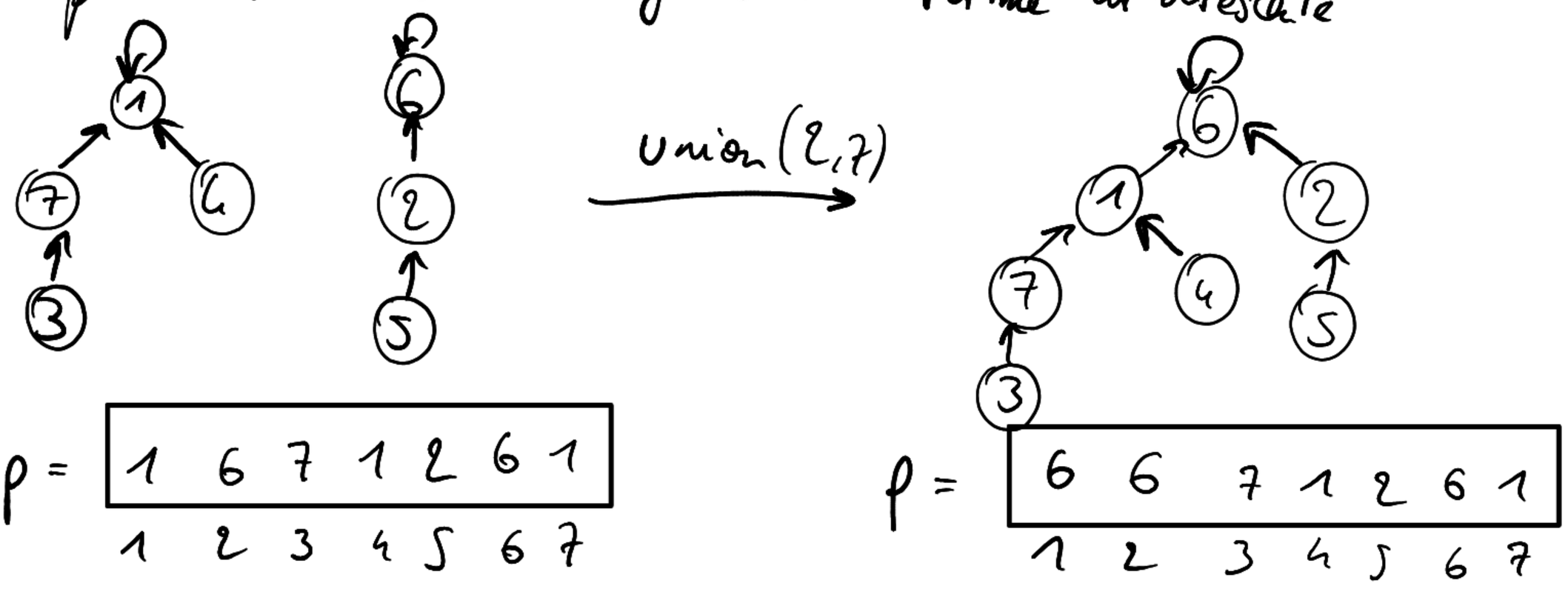
Implémentation 25: Pseudo-code de l'implémentation des ensembles disjoints par des tableaux:

```
Fonction créer(x)
    | p[x] ← x

Fonction trouver(x)
    | retourner p[x]

Fonction union(x,y)
    | rep-y ← trouver(y)
    | Pour i de 1 à n
    |     | Si p[i] = rep-y
    |     |     | p[i] = p[x]
```

2.) Implémentation avec des structures arborescentes:

Proposition 26: Pour représenter des ensembles disjoints, on peut représenter chaque ensemble par un arbre dont la racine est le représentant. En pratique, on maintient un tableau p de taille tel que p[x] est le parent de x dans l'arbre, ou x lui-même si p[x] est une racine.

Exemple 27: Ensembles disjoints sous forme arborescente



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

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

Propriété 28: Cette structure de données implémente des ensembles disjoints avec trouver(x) en $O(h(S_x))$ et union(x,y) en $O(\max(h(S_x), h(S_y)))$.