901 : Structures de données : exemples et applications

BOLLE Quentin

* Cormen
* Froidevaux, Type de données et algorithmes

Motivation : - travailler sur des ensembles dynamiques
- améliorer les algorithmes en ayant des opérations élémentaires peu coûteuses

Méthode : - envisager un type abstrait de données
- chercher ses représentations concrètes

I) Ensembles

Motivation : structure pauvre, pour les situations où l'on ne cherche que la présence/absence d'un élément

Opérations :

Ensemble_vide : → Ensemble # Renvoie $\emptyset$
Ajout : Elément x Ensemble → Ensemble # $S := S \cup \{a\}$
Supprimer : Elément x Ensemble → Ensemble # $S := S \setminus \{a\}$
_∈_ : Elément x Ensemble → Booléen # $a \in S$?

Représentations :

* Si $E \subseteq \{a_1, \dots, a_n\}$, on représente E par le tableau T avec $T[a_i] =$ Vrai ssi $a_i \in E$
Ajout $(x, E) \xrightarrow{\text{implémentation}} T[x] :=$ Vrai
Avantage : opérations en coût constant
Inconvénient : Si $n \to +\infty$, trop de coût mémoire

* On peut aussi représenter E par une liste de ses n éléments (il y a n! listes possibles)
Avantage : plus de coût mémoire exorbitant
Inconvénient : opérations en coût linéaire

Commentaire : sauf pour un petit nombre d'éléments, la structure d'ensemble se révèle être mal-adaptée à de nombreuses situations

II) Structures séquentielles

1) Liste

Motivation : Structure linéaire, pour traiter des données séquentiellement

Déf : 1) Une liste est une suite finie, éventuellement vide, d'éléments de E.
2) L'ensemble L des listes sur E est défini récursivement par $L = \emptyset + E \times L$

Opérations (récursives) :

Liste_vide : → Liste # Renvoie $\emptyset$
Tête : Liste → Element # Renvoie a si $L = \{a\} \times L'$
Queue : Liste → Liste # Renvoie L' si $L = \{a\} \times L'$
Ajout : Elément x Liste → Liste # $L := \{a\} \times L$

Commentaire : on peut ajouter des opérations plus complexes, comme la concaténation ou la recherche d'un élément.

Représentations :

* On représente la liste par un tableau surdimensionné dont la i-ème case est la i-ème place de la liste
Avantage : chercher le k-ième élément en $O(1)$
Inconvénients : les opérations récursives Queue et Ajout sont difficiles à programmer

* On peut utiliser des pointeurs pour chaîner les éléments successifs.
Avantage : pas de longueur maximum ; les opérations récursives sont faciles à programmer et en $O(1)$
Inconvénient : besoin de mémoire pour les pointeurs ; Accès au k-ème élément en $O(k)$.

2) Piles et Files

a) Piles

Motivation : Liste où l'on autorise les insertions et suppressions qu'à une seule extrémité, appelée sommet de pile (ex: pile d'assiettes)

Opérations :

- Pile-vide : $\rightarrow$ Pile
- est-vide : Pile $\rightarrow$ Booléen
- Empiler : Pile $\times$ Element $\rightarrow$ Pile
- Dépiler : Pile $\rightarrow$ Pile
- Sommet : Pile $\rightarrow$ Element

Représentations :

* La représentation par un tableau est plus simple que pour une liste "classique"
* La représentation par chaîne n'est pas modifiée.

b) Files

Motivation : Liste où l'on fait les adjonctions à une extrémité, les accès et les suppressions à l'autre extrémité (ex: file d'attente)

Opérations :

- File-vide : $\rightarrow$ File
- est-vide : File $\rightarrow$ Booléen
- Enfiler : File $\times$ Element $\rightarrow$ File
- Défiler : File $\rightarrow$ File
- Premier : File $\rightarrow$ Element

Représentations :

* pour un tableau, il faut conserver l'indice du 1er élément et celui du dernier. Tant qu'il n'y a pas de débordement, la programmation est simple
* Pour une chaîne, on rajoute un pointeur du dernier élément vers le 1er. Simple à programmer.

III) Structures arborescentes

1) Arbre binaire

Motivation : structure très riche permettant de nombreuses opérations (ex: organisation des fichiers dans un système d'exploitation.

Déf. Un arbre binaire est soit vide, soit de la forme $B = \langle o, B_1, B_2 \rangle$ où B_1 et B_2 sont deux arbres binaires. 'o' est un noeud appelé racine.

Opérations (élémentaires)

- Arbre_vide : $\rightarrow$ Arbre
- $\langle \_,\_,\_ \rangle$: Noeud $\times$ Arbre $\times$ Arbre $\rightarrow$ Arbre
- Racine : Arbre $\rightarrow$ Noeud
- G : Arbre $\rightarrow$ Arbre
- D : Arbre $\rightarrow$ Arbre
- Contenu : Noeud $\rightarrow$ Elément

Représentations :

Pour chaque noeud, on utilise 2 pointeurs : l'un vers le fils gauche et l'autre vers le fils droit. En plus, on peut ajouter un pointeur vers le noeud parent.

Commentaire : pour avoir des opérations plus complexes, il faut rajouter de la structure pour éviter les coûts linéaires.
On étudie deux cas particuliers : les files de priorité et les dictionnaires.

2) File de priorité et tas

Déf : Une file de priorité est une structure de données qui associe à chaque élément une valeur (appelé clé).
Une file de priorité min reconnait les opérations suivantes :

- Ajout(x, S)
- Minimum(S)
- Extraire_Min(S)
- Diminuer_Clé(x, S, k)

Commentaire : On définit de même une file de priorité max.

Def : Un tas est un arbre binaire où chaque niveau doit être rempli avant de remplir le suivant.
Un tas min (resp. tas max) est un tas où la valeur contenue dans un noeud doit être inférieur (resp. supérieur) à celles de ses fils.

Commentaire : la hauteur d'un tas à n noeud est en $\Theta(\log n)$

Opérations :

- Construire_tas_min(T) construit un tas min en $O(n)$ à partir d'un tableau T de taille n.
- Ajout_tas_min(x,T), Extraire_min_tas(T), Minimum(T) et Diminuer_Clé_Tas(x,T,k) s'exécutent en $O(\log n)$.

Commentaire : Une file de priorité min est donc bien représenté par un tas min.

Applications : Algorithme de Djikstra, algorithme de Prim

Application :
Un codage caractère par caractère est un arbre binaire où les caractères sont les feuilles.
Si un caractère c a $n(c)$ occurrences dans un texte, son coûte $n(c)h(c)$ où $h(c)$ est la hauteur de c dans le codage.

DEV1

Théorème : Si $((a_i, n(a_i)))_{i\in[\![1,N]\!]}$ est une suite de caractère avec leurs occurences dans un texte, le codage de Huffman minimise $\sum_{i=1}^{N} n(a_i)h(a_i)$ en temps $O(N\log N)$

3) **Dictionnaire et arbre de recherche**

Def : Un dictionnaire est une structure de donnée qui reconnait les opérations suivantes :
- Rechercher(x,S)
- Ajout(x,S)
- Supprimer(x,S)

Def : Un arbre (binaire) de recherche est un graphe étiqueté tel que pour tout noeud v de l'arbre, les éléments des noeuds du sous-arbre gauche (resp. droit) de v sont inférieurs (resp. supérieurs) à l'élément contenu dans v. On note ABR.

Opérations :
Rechercher_ABR(x,A), Ajout_ABR(x,A), Supprimer_ABR(x,A) s'exécutent en $O(h)$ où h est la hauteur de A

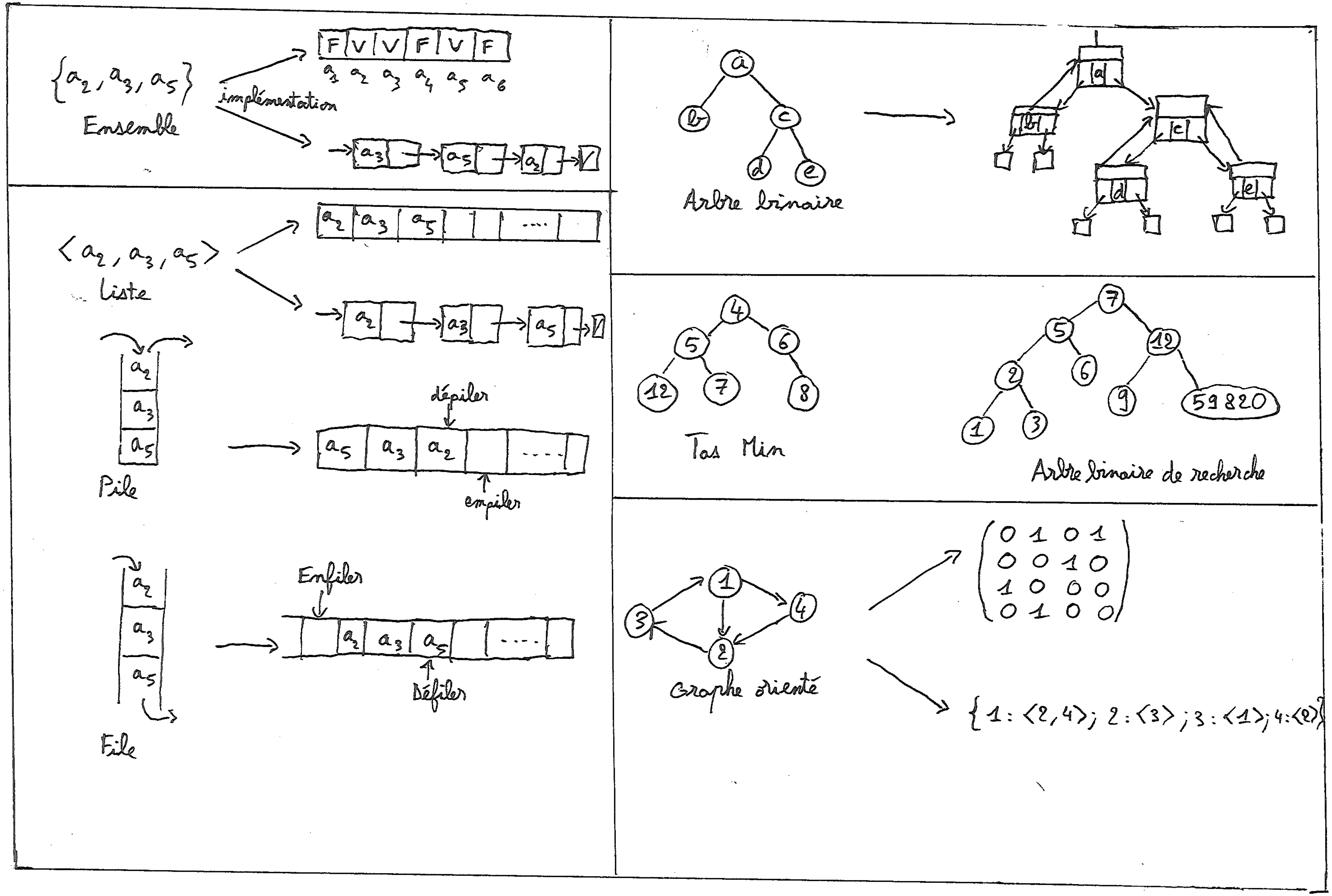
Commentaire : A moins de structure plus pointue, on peut avoir $h = n-1$ où n est le nombre de noeuds de A. Cependant, on a le résultat suivant :

Théorème : La hauteur moyenne d'un ABR à n clés distinctes construit aléatoirement est $O(\log n)$ ← DEV2

IV) Quelques mots sur les structures relationnelles

La structure de donnée principale est le graphe.
Le test principal consiste à voir s'il y a connexion entre deux sommets : $a \longrightarrow b$.
Le graphe est non-orienté s'il y a symétrie dans le test principal. Sinon, il est orienté.

Représentations : il y a 2 représentations majeures :
* la liste d'adjacence : souple pour ajouter/supprimer des sommets
* la matrice d'adjacence : le test principal est en $O(1)$.


{a2, a3, a5}
Ensemble
implémentation
F V V F V F
a1 a2 a3 a4 a5 a6
a3 a5 a2
⟨a2, a3, a5⟩
Liste
a2 a3 a5 ....
Pile
a5 a3 a2 ....
dépiler
empiler
File
Enfiler
a2 a3 a5 ....
Défiler
Arbre binaire
Tas Min
Arbre binaire de recherche
59820
Graphe orienté
0 1 0 1
0 0 1 0
1 0 0 0
0 1 0 0
{1: ⟨2, 4⟩; 2: ⟨3⟩; 3: ⟨1⟩; 4: ⟨2⟩}