Structures de Données

Alain Camanes

alain.camanes@free.fr

Stanislas

Option Informatique
2021-2022

Plan

Types abstraits & Structures de données

↪ Les algorithmes opèrent sur des *données*.

↪ *Représenter* & *Manipuler* les données.

↪ 2 niveaux de représentation.

- Abstrait ou logique (type abstrait de données).
- Implantation (structure de données).

↪ Objectif : séparation des niveaux : modularité.
Manipuler l'objet par son interface abstraite, *indépendamment* des détails de l'implémentation.

Définition

↪ Type abstrait.

- Ensemble d'objets.
- Opérations sur ces objets (*Primitives*).

↪ Hiérarchie de types : Types élémentaires, ...

↪ Structure de données.

- Représentation d'un type abstrait dans la mémoire d'un ordinateur.
- Implantation des opérations sur cette représentation (*Complexité*).

↪ Distinguer : Persistant & Mutable

Exemples

↪ Tableaux.
Taille fixe, Éléments de même nature, Repérés par un index.

- Longueur : temps *constant*.
- Lecture d'un élément connaissant son rang : temps *constant*.
- Mutable.

↪ Listes.
Un élément = Une valeur + 1 pointeur vers la suite, Liste vide.

- Lecture de la tête : *constant*.
- Accès à la queue : *constant*.
- Non mutable.

↪ Autres. Piles (stack, LIFO), Files (queue, FIFO), Arbres,. . .

↪ Primitives communes (données mutables).

- Construction de la pile / file vide.
- Ajout d'un élément.
- Suppression et renvoi d'un élément.
- Tester si la pile / file est vide.

↪ Modules Caml.

- Stack - Type 'a Stack.t.

```
# let p = Stack.create ();;
val p : '_a Stack.t = <abstr>
```

- Queue - Type 'a Queue.t.

```
# let q = Queue.create ();;
val q : '_a Queue.t = <abstr>
```

Plan

1. Types abstraits

2. Piles
   - Type
   - Application
   - Implantation

3. Files

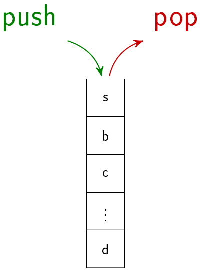
4. Dictionnaires

Pile

↪ Last In First Out. *stacks*.

↪ Exemples.

- Assiettes propres au self.
- Correction de copies.
- Pile d'exécution d'un programme.



↪ Sommet. Dernier élément ajouté.

↪ Modèle. Vide (ε) ou Suite d'éléments de type `'a` (`a:l`).

↪ Type *mutable*.

↪ Création. `Stack.create : unit -> 'a t`

↪ Test. `Stack.is_empty : 'a t -> bool`
`est_vide` ε renvoie `true`
`est_vide a:l` renvoie `false`

↪ Empile. *Push* - `Stack.push : 'a -> 'a t -> unit`
`empile e l` modifie `l` en `e::l`

↪ Dépile. *Pop* - `Stack.pop : 'a t -> 'a`
`depile e:l` renvoie `e` et modifie `e:l` en `l`
`depile` ε renvoie `Error`

$\hookrightarrow$ Expression algébrique. Ensemble d'objets défini inductivement par :

- Variables atomiques : `int`.
- Constructeurs d'arité 2 : $+$, $\times$.

$\hookrightarrow$ Représentation. $((1+2)+(-6))\times 4$.

- Arbre.
- Notation linéaire : postfixe (*polonaise inversée* - Hewlett Packard).
  `1 2 + (-6) + 4 *`

$\hookrightarrow$ Donnée.

```
type operations = Plus | Opp | Mult;;
type element = Op of operations | Nat of int;;
let exp = [Nat 1; Nat 2; Op Plus; Nat (-6); Op Plus;
           Nat 4; Op Mult];;
```

Expression algébrique : évaluation

```
let evalue exp =
  let p = Stack.create () in
  let rec aux exp p =
    match exp with
    | [] -> pop p
    | (Nat e)::q -> push e p; aux q p
    | _ -> let o1 = pop p and o2 = pop p in
           let (Op a)::q = exp in
           if a = Plus then push (o1 + o2) p
           else push (o1 * o2) p;
           aux q p
  in aux exp p;;
```

Implantation par tableau

↪ Type.

```
type 'a pile_tab = {pile : 'a array;
  mutable sommet : int};;
```

↪ Primitives.

```
let new n a =
  {pile = Array.make n a; sommet = -1};;

let isempty p = (p.sommet = -1);;

let push a p = (* Ajouter les erreurs *)
  p.sommet <- p.sommet + 1;
  p.pile.(p.sommet) <- a;;

let pop p = (* Ajouter les erreurs *)
  let sommet = p.pile.(p.sommet) in
  p.sommet <- p.sommet - 1; sommet;;
```

Implantation par liste mutable

↪ Type.

```
type 'a pile_list = {mutable liste : 'a list};;
```

↪ Primitives.

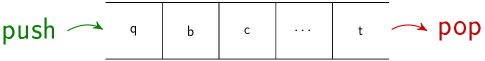
```
let new () = {liste = []};;

let isempty p = (p.liste = []);;

let push x p = p.liste <- x::p.liste;;

let pop p = match p.liste with
  | [] -> failwith "Pile vide"
  | t::q -> (p.liste <- q ; t);;
```

Plan

→ First In First Out. *queues*.

→ Exemples.

- File d'attente au self.
- File de priorité d'un processeur (buffers d'entrée/sortie).



→ Tête. Premier élément ajouté non encore retiré.

→ Queue. Dernier élément ajouté.

→ Modèle. Vide (ε) ou Suite d'éléments de type `'a` (`a:l`).

→ Type *mutable*.

Type abstrait : Opérations

↪ Création. `Queue.create : unit -> 'a t`

↪ Test. `Queue.is_empty : 'a t -> int`

`est_vide` ε renvoie `true`

`est_vide t:q` renvoie `false`

↪ Enfile. *Push* - `Queue.push : 'a -> 'a t -> unit`

`enfile a l` modifie `l` en `l:[a]`

↪ Défile. *Pop* - `Queue.pop : 'a t -> 'a`

`defile e:l` renvoie `e` et modifie `e:l` en `l`

`retire_file` ε renvoie `Error`

↪ Arbre. Parcourir les sommets par couches.

↪ Algorithme. *File*

Le nœud entre en tant qu'enfant et sort en tant que parent.

- Soit une file vide.
- On ajoute la racine.
- Tant que la file n'est pas vide, on enlève la tête, on parcourt le nœud correspondant et on stocke ses enfants.

Implantation par tableau - Cherchez l'ambiguïté

```
type 'a file_tab = {file : 'a array ;
                    mutable t : int; mutable q : int};;

let cree n a = {file = Array.make n a;
                t = 0; q = 0};;

let estvide f = (f.t = f.q);;
let empile a f =
 if f.q = (f.t + 1) mod n then failwith "Plein"
 else (f.file.(f.t) <- a; f.t <- (f.t + 1) mod n);;

let depile f =
  if f.q = f.t then failwith "File vide"
  else (let queue = f.file.(f.q) in
   f.q <- (f.q + 1) mod n); queue;;
```

Implantation par tableau - Cherchez l'ambiguïté

```
type 'a file_tab = {file : 'a array ;
               mutable t : int; mutable q : int};;

let cree n a = {file = Array.make n a;
          t = 0; q = 0};;

let estvide f = (f.t = f.q);;
let empile a f =
 if f.q = (f.t + 1) mod n then failwith "Plein"
 else (f.file.(f.t) <- a; f.t <- (f.t + 1) mod n);;

let depile f =
  if f.q = f.t then failwith "File vide"
  else (let queue = f.file.(f.q) in
   f.q <- (f.q + 1) mod n); queue;;
```

Distinguer les listes *pleine* et *vide* en imposant une distance de 1 entre la tête et la queue.

File : Implantation par deux listes chaînées

```ocaml
type 'a file_list =
  {mutable queue : 'a list;
   mutable tete : 'a list};;

let cree () = {queue = []; tete = []};;

let isempty f = f.queue = [] && f.tete = [];;

let push a f = f.queue <- a::f.queue;;

let pop f =
  if f.tete = [] then
    (f.tete <- List.rev f.queue; f.queue <- []);
  if f.tete = [] then failwith "File vide"
  else (let a::q = f.tete in
        f.tete <- q; a);;
```

Plan

$\hookrightarrow$ Permet de représenter et manipuler des ensembles clé / valeur.

$\hookrightarrow$ Opérations. Recherche, Insertion, Suppression.

$\hookrightarrow$ Éléments accessibles par une *clé* (unique).

$\hookrightarrow$ Élément contient une *valeur*.

Dictionnaires : Type abstrait

↪ Couple de valeurs (clef, valeur)

↪ **Création.** - `Hashtbl.create : int -> ('a, 'b) t`

↪ **Test** de vacuité - `Hashtbl.length : ('a, 'b) t -> int` (temps constant)

↪ **Recherche.** `Hashtbl.find : ('a, 'b) t -> 'a -> 'b`
`cherche dic k` renvoie `v` t.q. `(k, v)` $\in$ `dic`
Renvoie `Error` si `k` absente.

↪ **Insertion.** `Hashtbl.add : ('a, 'b) t -> 'a -> 'b -> unit`
`insere dic k v` renvoie `dic` $\cup$ `(k,v)`.

↪ **Suppression.** `Hashtbl.remove : ('a, 'b) t -> 'a -> unit`
`supprime dic k` renvoie `dic` $\backslash\{$`(k, v)`$\}$.

Implantation par une liste

↪ Dictionnaire `('a, 'b) list`

Liste contenant les couples clé / valeur.

↪ Complexités.

	meilleur	pire
Recherche	1	n
Insertion	1	1
Suppression	1	n

Implantation par un tableau non trié

↪ Dictionnaire `('a, 'b) array, int`

- Tableau contenant les couples clé / valeur.
- Entier stockant la taille.

↪ Complexités.

	meilleur	pire
Recherche	1	n
Insertion	1	1
Suppression	1	n

Implantation par un tableau *trié*

↪ Dictionnaire `('a, 'b) array, int`

- Tableau contenant les couples clé / valeur.
- Entier stockant la taille.

↪ Clés triées.

↪ Complexités.

	meilleur	pire
Recherche dicho	1	log n
Insertion	1	n
Suppression	1	n

Tables de hachage

$\hookrightarrow$ Fonction de hachage. $h : C \to [\![0, m-1]\!]$.

C : Ensemble des clés, $n = |C|$.

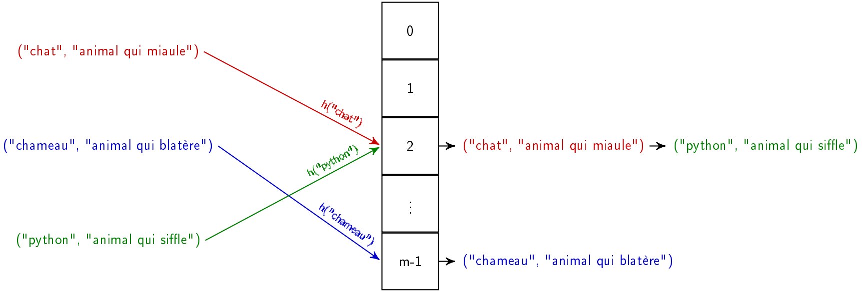
`tab` : tableau de taille $m \ll n$.

Le couple (c, v) est stocké à l'indice $h(c)$ dans `tab`.

$\hookrightarrow$ Inconvénient. Risques de collisions.

$\hookrightarrow$ Contraintes.

- Fonction facile à calculer.
- Distribution uniforme en les entrées.



Collisions : Listes chaînées

↪ Paradoxe des anniversaires.

- k : nombre de clés distinctes,
- distribution *uniforme* des clés sur $[\![0, m-1]\!]$.
- p_k la probabilité qu'il y ait au moins une collision.

$$p_k = 1 - \frac{m!}{m^k(m-k)!}$$

$m = 10^6$, $k = 2500$ clés distinctes : $p_k \geqslant 95\%$.

↪ Chaînage. Chaque case est une liste chaînée dont chaque case contient la valeur (k, v).

↪ Complexité.
Ajout : $\Theta(1)$. Recherche : case $\Theta(1)$ + liste (linéaire).

$\hookrightarrow$ Clés sous forme d'entiers. $s : C \to \mathbb{N}$.
Chaîne de caractère ASCII $\to$ Nombre en base 127
$s(c_0 \cdots c_p) = \sum_{k=0}^{p} s_k 127^k$.

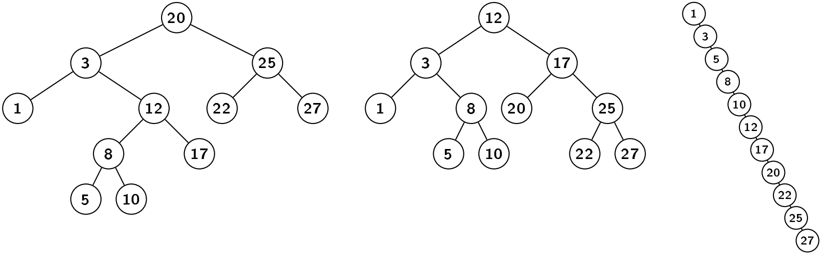
$\hookrightarrow$ Exemple I. $h : n \mapsto n \pmod m$.

- Si $m = 2^p$, seuls les p plus petits bits comptent.
- Si clés périodiques, choisir m premier. Alors, $\{a + bi \pmod m, i \in [\![0, m-1]\!]\} = [\![0, m-1]\!]$.
- *Choix* de m : premier, loin d'une puissance de 2.

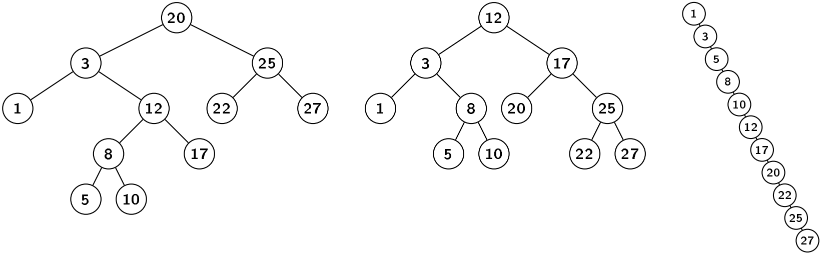
$\hookrightarrow$ Exemple II. $\alpha \in]0, 1[$.
$h : n \mapsto \lfloor m \cdot (n \cdot \alpha \pmod 1) \rfloor$.

$\hookrightarrow$ Arbre Binaire de Recherche. Chaque nœud possède une clé. Les clés du *sous-arbre gauche* lui sont *inférieures*. Les clés du *sous-arbre droit* luis sont *supérieures*.

$\hookrightarrow$ *Exemples.* Étiquettes : `[1;3;5;8;10;12;17;20;22;25;27]`

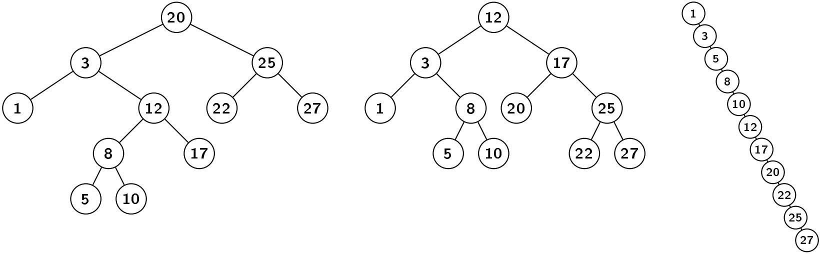


↪ *Exemples.*



↪ Que dire du parcours infixe ?

↪ *Exemples.*

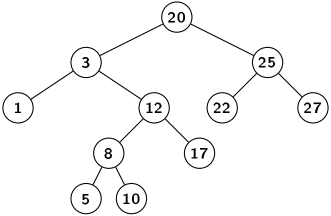


↪ Quel est le minimum des étiquettes ?

↪ Quel est le maximum des étiquettes ?

↪ L'étiquette 23 est-elle présente ?

Insérer

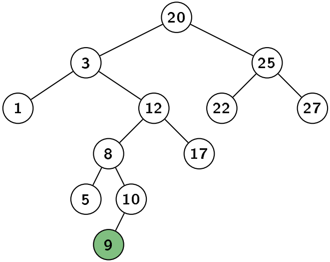


$\hookrightarrow$ Insérer (dans une feuille) la clé 9 ?

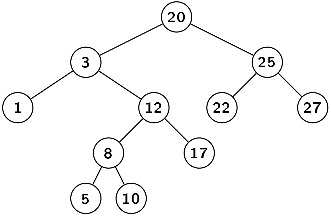
Si $T = (T_g, x, Td)$.

- Si $9 < x$, insérer dans T_g.
- Si $9 > x$, insérer dans T_d.

Si T = `Vide`, retourner (`Vide`, 9, `Vide`).

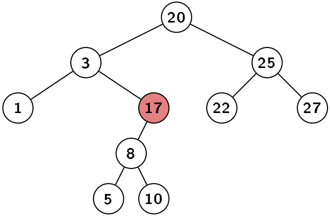


Supprimer



$\hookrightarrow$ Supprimer la clé 12 ? Notons $T = (T_g, x, Td)$.

- Si $12 < x$, supprimer dans T_g.
- Si $12 > x$, supprimer dans T_d.
- Si $12 = x$,
    - Si $T_g =$ `Vide`, renvoyer T_d.
    - Si $T_d =$ `Vide`, renvoyer T_g.
    - Sinon, supprimer un minimum m de T_d pour obtenir T'_d et renvoyer (T_g, m, T'_d).



Équilibrer (H.P.)

↪ Recherche et Insertion dans un dictionnaire, via un ABR, en

$$\Theta(\log n) \ ?$$

Maintenir une structure d'arbre qui garantisse l'équilibre.

↪ Arbres Adelson-Velsky Landis.
Pour supprimer ou insérer, utiliser des *rotations* pour maintenir l'équilibre.

↪ Arbres Rouge-Noir.
Colorier les nœuds (maintenir un attribut supplémentaire). Pour supprimer ou insérer, maintenir une *coloration* des nœuds qui garantit l'équilibre de l'arbre.