Arbres tournois, tas-max

Judicaël Courant

2018-W37-1

1 Arbres tournois

1.1 Introduction

Définition 1.1.1. *Un arbre binaire étiqueté par des nombres (ou plus généralement par les éléments d'un ensemble ordonné) est dit* arbre tournoi *si l'étiquette de tout nœud majore l'ensemble de celles de ses descendants.*

Exercice 1.1.2. *Voici trois affirmations :*

1. *On peut tester si un arbre est tournoi en temps quadratique en la taille de l'arbre.*
2. *Un arbre est tournoi si et seulement si, pour chaque nœud de l'arbre, chaque fils non-vide a une racine inférieure ou égale à celle du nœud considéré.*
3. *On peut tester si un arbre est tournoi en temps linéaire en la taille de l'arbre.*

Combien de ces affirmations sont vraies ?

A Zéro

B Une

C Deux

D Trois

1.2 Réalisation d'une file de priorité

— Extraction du maximum : on réalise une *percolation* descendante ;
— Ajout d'un élément : *percolation* montante.

(on utilise également le terme *tamisage* ; en anglais *sink*)

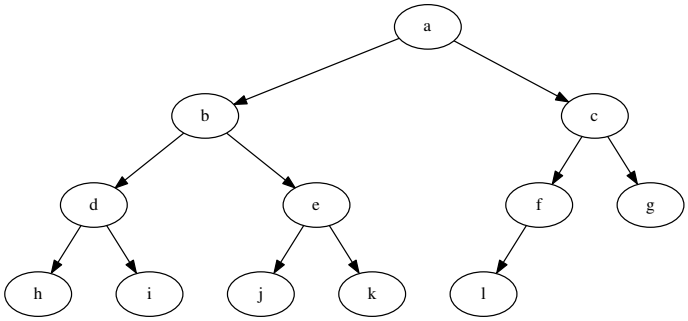
Complexité : $O(h)$ où h est la hauteur de l'arbre considéré.

2 Arbres quasi-complets

2.1 Introduction

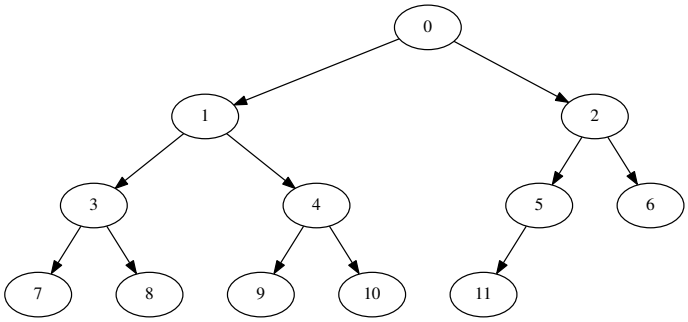
2.1.1 Définition

Définition 2.1.1. *Un arbre (quasi-)complet est un arbre de hauteur h où toutes les feuilles sont à profondeur $h-1$ ou h, les feuilles de profondeur h étant bien tassées sur la gauche. Un arbre quasi-complet est parfois appelé également* tas binaire.



2.2 Représentation par un tableau

Numérotons les nœuds d'un arbre quasi-complet par un parcours en largeur d'abord puis de gauche à droite (parcours militaire) en commençant par 0.



2.3 Propriété de la numérotation

Théorème 2.3.1. *Notons h la hauteur du tas considéré et n son nombre de nœuds.*
Alors les numéros des nœuds situés à la profondeur k sont les éléments de
— $[\![2^k-1, 2^{k+1}-1[\![$ *pour* $k \in [\![0, h-1[\![$.
— $[\![2^{h-1}-1, n[\![$ *pour* $k = h-1$.
En posant $q = 2p+1$ (resp. $2p+2$), le fils gauche (resp. droit) du nœud numéro p :

— *existe ssi* $q < n$ *;*
— *a pour numéro* q *s'il existe.*

2.4 Démonstration

1. c'est vrai pour les nœuds les plus à gauche de chaque rangée (numéro de la forme $2^k - 1$);
2. on effectue une récurrence pour le reste des nœuds de la rangée.

On peut alors représenter un tas de taille n par un tableau de taille n — voire par un tableau de taille supérieure (seuls les n premiers éléments étant pris en compte).

Notons que cette représentation est impérative : on peut modifier un arbre existant (contrairement aux arbres classiques en Caml).

```
type 'a tas = {
  mutable nb_elem : int;
  mutable donnees : 'a array
};;
(* cree_tas : int -> 'a -> 'a tas
   (cree_tas n x) crée un tas (arbre quasi-complet) initialement vide
   pouvant contenir jusqu'à n éléments du type de x.
*)
let cree_tas n x = {
  nb_elem = 0;
  donnees = Array.make n x
};;
```

On peut représenter un sous-arbre par un couple (tas, numéro du noeud où s'enracine le sous-arbre) ou tout simplement par le numéro du noeud. On choisit cette dernière solution :

```
type noeud = int;;
let racine = 0;;
let fils_gauche p = 2*p + 1;;
let fils_droit p = 2*p + 2;;
let pere p = (p - 1) / 2;; (* div. euclidienne *)
let est_racine p = (p = 0);;

(* elem : 'a tas -> noeud -> 'a
   retourne l'étiquette d'un noeud. *)
let elem t p = t.donnees.(p);;

(* est_vide : noeud -> 'a tas -> bool *)
let est_vide p t = (p >= t.nb_elem);;

(* swap : 'a tas -> noeud -> noeud -> unit
   échange les étiquettes des noeuds. *)
let swap t n1 n2 =
```



```
  let u = t.donnees in
  let x1 = u.(n1) and x2 = u.(n2) in
  u.(n1) <- x2;
  u.(n2) <- x1
;;
```

Il n'est pas difficile de reconstituer un arbre_bin à partir d'un tas binaire :

```
(* arbre_de_tas : noeud -> 'a tas -> 'a arbre_bin
   (arbre_de_tas n t) retourne le sous-arbre
   enraciné au noeud n du tas t.
*)
let rec arbre_de_tas p t =
  if est_vide p t then Vide
  else let fg = arbre_de_tas (fils_gauche p) t in
  let fd = arbre_de_tas (fils_droit p) t in
  N(t.(p), fg, fd)
;;
```

3 Structure de tas max

3.1 Définition

Définition 3.1.1. *Un* tas max *est un arbre quasi-complet qui est également un arbre tournoi. Les* tas max *sont parfois appelés simplement* tas binaire *(il y a donc risque de confusion entre le tas binaire qui désigne un tas max et le tas binaire qui désigne un arbre quasi-complet).*

3.2 Opérations usuelles

```
(* maximum : 'a tas -> 'a
   précondition : le tas n'est pas vide
   et vérifie la propriété de tas max *)
let maximum t = t.donnees.(racine);;
```

Percolation descendante sur un sous-arbre :

```
(* percole_descendant : 'a tas -> int -> unit
   (percole_descendant t p) réorganise le sous-arbre
   enraciné au noeud p par percolation, de façon
   qu'il respecte les conditions de tas max.
   Précondition : le noeud p existe et les éventuels
   fils de p respectent les conditions de tas-max.
*)

let rec percole_descendant t p =
  let fg = fils_gauche p in
```



```
  let fd = fils_droit p in
  let m = if not(est_vide fg t)&&elem t fg > elem t p
    then fg else p in
  let m' = if not(est_vide fd t)&&elem t fd > elem t m
    then fd else m in
  if m' <> p then begin
    swap t m' p;
    percole_descendant t m'
  end;;
```

NB : N'est pas facile à écrire avec une boucle `while` (sans `break` ni `return`).

```
(* extrait_max : 'a tas -> unit
   Modifie le tas pour en enlever le maximum.
   Le tas résultant vérifie encore la propriété de
   tas-max.
   Précondition : le tas n'est pas vide et
   la propriété de tas-max est vérifiée.
*)
let rec extrait_max t =
  let v = elem t (t.nb_elem - 1) in
  t.nb_elem <- t.nb_elem - 1;
  t.donnees(racine) <- v;
  percole_descendant t racine;;
```

Percolation montante :

```
(* percole_montant : 'a tas -> unit
   Réorganise le tas par percolation, de façon qu'il
   respecte les conditions de tas max.
   Précondition : le tas est non-vide et respecte
   les conditions de tas max sauf peut-être entre le
   dernier noeud du tas (le plus en bas à droite) et
   son père.
*)

let percole_montant t =
  let pos = ref (t.nb_elem - 1) in
  while !pos <> racine && elem t !pos > elem t (pere !pos) do
    (* les contraintes de tas sont vérifiées partout,
       sauf entre !pos et son père. *)
    let p = pere !pos in
    swap t p !pos;
    pos := p;
  done;;
```



```
(* ajoute_elem : 'a tas -> 'a -> unit
   Ajoute l'élément donné au tas.
   Conserve la propriété de tas-max.
   Précondition : le nombre d'éléments du tas est
   strictement inférieur à la taille du tableau
   utilisé *)
let ajoute_elem t x =
  let n = t.nb_elem in
  t.donnees.(n) <- x;
  t.nb_elem <- n + 1;
  percole_montant t
;;
```

3.3 Réalisation d'une file de priorité (impérative)

```
let cree_file_priorite = cree_tas;;
let insere = ajoute_elem;;
```

3.4 Construction en $O(n)$ (par forêt)

La construction naïve d'un tas-max de n éléments (par ajout successifs) a un coût temporel $O(n \log n)$ dans le cas le pire.

Exercice 3.4.1. *Démontrer que cette borne est serrée.*

Exercice 3.4.2. *Quelle est la complexité de cette construction dans le cas le meilleur ?*

La construction du tas-max de n éléments par ajout successifs des éléments se fait donc en temps $\Theta(n \log n)$ dans le cas le pire.

Mais on peut faire mieux : on peut commencer par construire un tas de profondeur h à partir des n éléments (sans que ce soit un tas-max), puis effectuer des percolations sur les sous-arbres.

Pour les nœuds situés profondeur $h-1$, il n'y a rien à faire.

On effectue des percolations descendantes des nœuds de numéros dans $[\![2^{h-2}-1, 2^{h-1}-1[\![$ (deux appels récursifs par nœud au maximum), puis des nœuds de numéros dans $[\![2^{h-3}-1, 2^{h-2}[\![$ (trois appels récursifs au plus), etc.

À une étape donnée du calcul, on a donc fait en sorte que les 2^k arbres enracinés à la profondeur k aient la propriété de tas-max.

Au total, le nombre total d'appels sera donc majoré par

$$\sum_{k=2}^{h} k2^{h-k} \leq 2^h \sum_{k=2}^{+\infty} k2^{-k} = O(2^h) = O(n)$$

On peut donc construire le tas en $O(n)$.

Un ensemble d'arbre est appelé une *forêt*, d'où :

Définition 3.4.3. *On appelle construction du tas par* forêt *cette construction.*

3.5 Tri par tas

3.5.1 Principe

Il consiste à trier un tableau par la méthode suivante :

1. Construire un tas à partir du tableau ;
2. Extraire de ce tas les maximums successifs et les mettre dans le tableau résultat.

Fait remarquable : on peut utiliser le même tableau pour représenter le tableau de sortie et le tas.

3.5.2 Implantation

On peut alors écrire tout le code sans même faire référence au type `tas` :

```
let swap t i j =
  let tmp = t.(i) in (t.(i) <- t.(j); t.(j) <- tmp);;
let percole_descendant p t n =
  let pos = ref p in
  let c = ref false in
  while !c do
    let g = 2 * !pos + 1 in
    let d = g + 1 in
    let next = if d<n && t.(d)>t.(g) then d else g in
    if next < n && t.(next) > t.(!pos) then begin
      swap t !pos next;
      pos := next;
    end else c := true
  done;;

let tri_tas t =
  let n = vect_length t in
  for i = (n-2) / 2 (* pere(n-1) *) downto 0 do
    (* pour j>i, t[j] est un tas-max *)
    percole_descendant i t n;
  done;
  for i = n-1 downto 1 do
    (* t[i+1], ..., t[n-1] est trié, et t[0],...,t[i]
       est un tas-max d'éléments plus petits *)
    swap t racine i;
    percole_descendant racine t i
  done;;
```

NB : Pas besoin de percoler montante.

Complexité en temps : $O(n \log n)$.

Complexité en espace : $O(1)$.

3.5.3 Avantages du tri par tas

— Rapide ;
— En place ;
— Raisonnablement court ;

3.5.4 Inconvénients

— Non stable ;
— Incompréhensible sans les concepts.

