Chapitre 1

Arbres

1.1 Quelques rappels sur les arbres binaires

Commençons par rappeler la définition d'un arbre binaire en informatique :

Définition 1.1.1
Soit $\mathcal{N}$ un ensemble. Alors l'ensemble $\mathcal{T}$ des arbres est le plus petit ensemble tel que

-
-

Les éléments de $\mathcal{N}$ sont appelés

Autrement dit, toutes les feuilles sont des arbres, et étant donnés deux arbres et un nœud, on peut construire un nouvel arbre.

EXEMPLE

Considérons l'ensemble $\mathcal{L} = \mathbb{N}$. Alors sont des arbres :

-
-

-

Évidemment, cette représentation est difficile à suivre ; on représentera donc les arbres comme des arbres inversés (on omet d'écrire les nil en bas de l'arbre). Ces deux derniers arbres seront donc définis par

Définition 1.1.2
Soit $t = (t_1, n, t_2)$ un arbre. Alors

- n *est appelé*
- t_1 *est appelé*
- t_2 *est appelé*

Définition 1.1.3
Un nœud dont les deux fils sont nil *est appelé*

Regardons des implémentations possibles en OCaml :

```
type 'a arbre =




let racine a =
```

EXERCICE

Écrire les fonctions donnant les fils d'un arbre.

1.1.1 Induction structurelle

On prouvera la plupart des résultats sur les arbres par induction strucurelle ; les arbres étant définis par induction, c'est le type de preuve "naturel".

Théorème 1.1.4
Soit P une propriété portant sur les arbres. Si

-
-

alors pour tout arbre t,

De la même façon, on peut définir inductivement des fonctions sur l'ensemble des arbres.

Théorème 1.1.5
Soit E un ensemble non vide. Soient et
Alors il existe une unique fonction $\varphi : \mathcal{T} \to E$ telle que

- $\varphi(\text{nil}) =$
- *Pour tous $t_1, t_2 \in \mathcal{T}$ et $n \in \mathcal{N}$, $\varphi((t_1, n, t_2)) =$.*

1.1.2 Taille d'un arbre

Il y a plusieurs moyens pour mesurer la taille d'un arbre. Les plus évidents sont de compter soit le nombre de nœuds, soit le nombre de feuilles.

```
let rec nombre_feuilles a =









let rec nombre_noeuds a =
```

On notera souvent $|t|$ le nombre de nœuds de t.

Une autre méthode de mesure plus subtile est de considérer la hauteur de l'arbre.

Définition 1.1.6
On définit la hauteur d'un arbre par induction :

- *La hauteur de l'arbre vide est*
- *La hauteur d'un arbre est*

On notera souvent $h(t)$ la hauteur d'un arbre.

```
let rec hauteur a =
```

Proposition 1.1.7
Soit t un arbre binaire. Alors

Démonstration. Par induction :

- Si $t = \text{nil}$, alors
- Sinon, on a $t = (t_1, n, t_2)$, l'inégalité étant vérifiée par t_1 et t_2.
  Alors $|t| =$ et $h(t) =$.
  Donc

□

1.1.3 Remarque sur l'équilibrage

On verra dans la suite que beaucoup d'algorithmes sur les arbres ont une complexité dépendant de la hauteur. L'idéal serait donc d'avoir $h(t) =$ dans l'inégalité précédente.

Ce cas optimal correspond au cas d'un arbre complet, *i.e.* tel que les hauteurs des fils de tout nœud soient égales. On cherchera donc de tels arbres, et on appellera *équilibré* tout arbre vérifiant $h(t) =$.

On peut par exemple regarder le cas des arbres AVL *.

Définition 1.1.8
On appelle déséquilibre d'un arbre $t = (t_1, n, t_2)$ *la différence*

Un tel arbre est dit arbre AVL *s'il vaut* nil *ou si*

*. Des noms Adelson, Velsky et Landis

-
-

Notons dans la suite (f_n) la suite de Fibonacci.

Proposition 1.1.9
Soit t un arbre AVL. Alors

Démonstration. Par induction structurelle :

- Si $t = \text{nil}$, alors
- Si $t = (t_1, n, t_2)$ et que t_1, t_2 vérifient bien l'inégalité, alors l'un au moins a pour hauteur , et donc contient au moins nœuds. Mais l'arbre étant équilibré, l'autre est de hauteur au moins , et donc contient au moins nœuds.

  In fine, t contient au moins

□

Or on sait que f_n est de l'ordre de φ^n, et donc on a bien

$$h(t) =$$

L'autre cas extrême est le cas où On est alors dans le cas d'un arbre avec une seule branche, qui se comporte en fait comme une liste.

1.2 Arbres binaires de recherche

Dans toute la suite, nous supposerons que l'ensemble $\mathcal{N}$ est totalement ordonné par un ordre $\leqslant$.

Définition 1.2.1
On appelle arbre binaire de recherche *ou ABR tout arbre binaire t qui est soit vide, soit tel que*

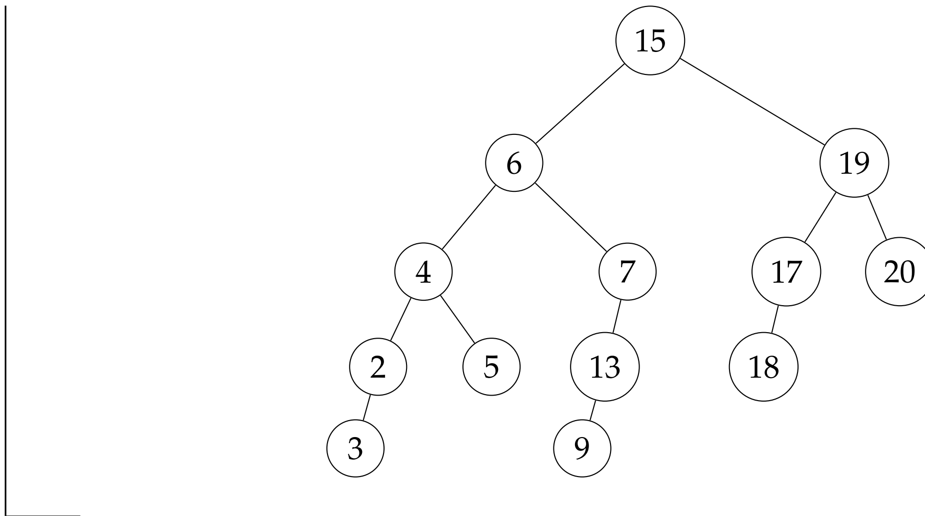
Commençons par écrire une fonction qui teste si un arbre binaire est un ABR :

```
let rec est_inferieur_egal t x =





let rec est_superieur t x =





let rec est_ABR t =
```

NOTA

On a ici un problème d'ordre esthétique : la définition n'est pas symétrique entre le fils gauche et le fils droit. En fait, elle le devient si on oblige les nœuds à être tous distincts.

EXEMPLE



Les arbres binaires permettent en particulier d'afficher toutes les valeurs dans l'ordre croissant, en faisant simplement : supposons que affiche est une fonction de type 'a -> unit

```
let rec parcours_infixe t affiche =
```

L'opération justifiant le plus cette structure de données est la fonction recherche, donnée par

```
let rec recherche t x =
```

Proposition 1.2.2
La complexité au pire de l'algorithme de recherche est en

Il sera donc intéressant d'avoir des arbres équilibrés. Dans le pire cas, on retrouve l'algorithme classique de recherche dans une liste.

On va regarder d'autres fonctions classiques : l'insertion, la suppression et la recherche de minimum et maximum.

```
let rec inserer x t =
```

Un problème de cette fonction est qu'elle ne peut insérer qu'au niveau des feuilles. Une insertion d'éléments toujours plus grand va donc créer un gros déséquilibre.

Pour le minimum, il suffit de noter qu'il est nécessairement
(méthode similaire pour le maximum).

```
let rec minimum t =

let rec maximum t =
```

Là encore, si l'arbre est bien équilibré, on a une complexité plus faible que la recherche de maximum ou minimum dans une liste.

La fonction de suppression est plus compliquée, principalement si la valeur à supprimer est une racine.

L'idée est la suivante : si l'élément à supprimer n'a pas de fils droit, on renvoie tout simplement le fils gauche. Sinon, la nouvelle racine de l'arbre sera la plus petite valeur du fils droit.

```
let rec supprimer x t =
```

Proposition 1.2.3
La fonction `supprimer x t`

Démonstration. Seul le dernier cas est à vérifier. Il suffit de se convaincre que

□

1.2.1 Structure persistante de dictionnaire

Rappelons qu'un dictionnaire est une structure de données permettant d'accéder à des valeurs *via* des clefs.

Plus précisément, si C est un ensemble ordonné et V un ensemble de valeurs, alors un dictionnaire est

Un ABR bien équilibré permet donc d'avoir un accès rapide à la bonne clef.

Notons que la structure est dite *persistante* :

1.3 Structure de tas

On considère maintenant un ensemble de nœuds totalement ordonné.

Définition 1.3.1
On appelle tas *tout arbre t tel que*

NOTA

On a en fait défini la structure de tas-max. On peut aussi construire des tas-min en remplaçant "inférieurs" par "supérieurs".

Les tas sont particulièrement utiles pour implémenter la structure de *file de priorité*. C'est un type abstrait sur des couples (priorité, valeur) permettant trois opérations :

-
-
-

On ne considérera dans la suite que des tas *parfaits*, *i.e.* des arbres complets, sauf éventuellement la dernière ligne qui doit être remplie de gauche à droite. On prendra, pour simplifier, des arbres sur les entiers naturels.

Cette contrainte va nous permettre de représenter efficacement les tas avec un tableau, en utilisant la méthode de généalogie de Sosa-Stradonitz :

-
-
-

Dans l'autre sens, on voit que le père d'un nœud de numéro k porte le numéro

```
type tas = int array
```

Après modification d'un élément du tas, il se peut qu'un nœud ne soit plus à la bonne place ; il faut le faire monter ou descendre dans l'arbre.

L'opération la plus simple est la remontée, il suffit d'échanger le nœud voulu avec son père autant de fois que nécessaire.

```
let swap t i j =




let monte t k =
```

Pour la descente, on échange le nœud souhaité avec son plus grand fils, s'il existe. Pour cela, on introduit un indice n qui représente la première case vide du tableau. On remarque alors que le nœud portant le numéro i a

- deux fils si
- un seul fils (gauche, donc) si
- aucun fils si

```
let descend t n k =
```

```

```

L'opération de montée nous permet donc de transformer tout tableau en tas : il suffit de faire remonter chaque élément du tableau dans le tas défini par les élements à gauche :

```
let array_to_tas a =
```

Ceci nous donne donc un nouvel algorithme de tri : le tri par tas (*heap sort*).

```
let tri_tas a =
```

Il fonctionne de la façon suivante :

-
-
-
-
-

Finalement, le tri par tas a une complexité en

1.3.1 Files de priorités impératives

Comme on l'a dit, les tas nous permettent d'implémenter des files de priorités.

On définit donc les types suivants :

```
type 'a data = {Priorite: int; Valeur: 'a}
type 'a tas = {mutable N: int; Tas: 'a array}
exception Vide
exception Plein
```

Les éléments de la file sont donc des couples (Priorité, Valeur), et un tas inclut maintenant un entier N, qui représentera la première case vide du tableau.

On crée aussi deux exceptions pour les cas où on veut supprimer une valeur d'un tas vide, ou insérer dans un tas plein.

EXERCICE

Modifier les fonctions `monte` et `descend` pour tenir compte des modifications de types.

Pour créer une file de priorité vide de taille n, il suffit de mettre n'importe quel valeur dedans :

```
let file_vide n v =
```

Pour ajouter un élément, on peut alors l'ajouter

```
let ajoute (p,v) t =
```

Pour extraire l'élément de plus grande priorité (qui est nécessairement à la racine), on le recupère puis on le remplace

```
let extrait t =
```

Pour augmenter la priorité d'un élément, il suffit de

```
let augmente t k p =
```

1.3.2 Files de priorité persistante

On peut reprocher à l'implémentation précedente de ne pas être persistante : les files de priorités sont modifiées, et il est donc impossible d'en garder une trace.

Nous verrons en TD comment faire une file de priorité fonctionnelle et persistante, en utilisant la structure d'arbre.

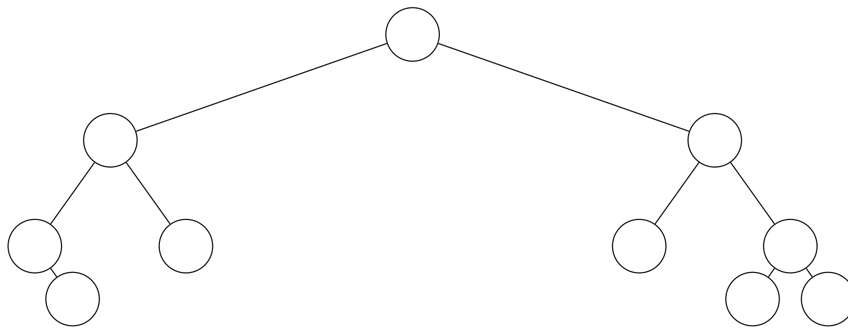
1.4 Exercices

Arbres binaires

Exercice 1. On appelle *arbre rouge-noir* tout arbre binaire auquel on rajoute une donnée à chaque nœud : sa couleur, rouge ou noire, et tel que

- La racine est noire
- Le parent d'un rouge est noir
- Tous les chemins d'une feuille à la racine comporte le même nombre de nœuds noirs.

(i) Colorier l'arbre ci-dessous



(ii) Montrer que certains arbres ne peuvent pas être coloriés.

(iii) Soit $b(a)$ le nombre de nœuds noirs des chemins d'une feuille à la racine d'un arbre rouge-noir. Montrer que

$$b(a) \leqslant h(a) + 1 \leqslant 2b(a).$$

(iv) Montrer que $|a| \geqslant 2^{b(a)} - 1$.

(v) En déduire qu'un arbre rouge-noir est équilibré.

Arbres binaires de recherche

Exercice 2. Écrire une fonction qui vérifie qu'un arbre binaire est un ABR.

Exercice 3. On a vu comment insérer une nouvelle valeur au niveau des feuilles. Écrire une fonction permettant l'insertion à la racine.

Exercice 4. On veut fusionner deux ABR a et b. Pour cela, on insère la racine de b dans a, puis fusionner le fils gauche (resp. droit) de b avec le fils gauche (resp. droit) de l'arbre obtenu.

Écrire le code correspondant.

Tas

Exercice 5. Écrire une fonction qui teste si un arbre binaire est un tas.

Exercice 6. Montrer qu'un tas à n nœuds a exactement $\lceil n/2 \rceil$ feuille.