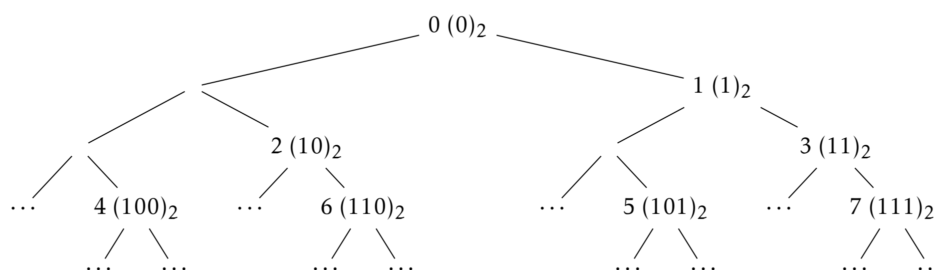
Travaux pratiques

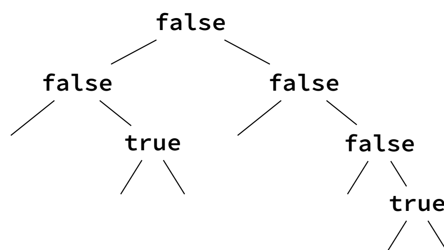
Représentation d'ensembles par arbres radix

On se propose de décrire une structure de données pour représenter les ensembles finis d'entiers positifs ou nuls appelée *arbres radix* (*radix tree* en anglais). L'idée consiste en une structure arborescente basée sur l'écriture binaire de l'entier : si l'entier vaut 0 il est stocké à la racine ; sinon on le stocke dans le sous-arbre gauche si son dernier chiffre binaire est 0 (donc s'il est pair), et dans le sous-arbre droit si son dernier chiffre binaire est 1 (s'il est impair), et l'on poursuit avec les chiffres suivants.
À titre indicatif, voici où se retrouvent placés les huit premiers entiers (leur écriture binaire est indiquée entre parenthèses) :



Notez que les chiffres de l'écriture binaire sont utilisés de droite à gauche (du moins significatif au plus significatif).

On choisit de coder les arbres radix en Caml par des arbres avec des feuilles vides et des nœuds contenant un booléen indiquant la présence ou l'absence de l'entier correspondant dans l'ensemble représenté par l'arbre.
L'ensemble $\{2,7\}$ sera donc représenté par l'arbre suivant :



Par la suite, on conviendra d'identifier un ensemble d'entiers avec l'arbre radix qui le représente.

On définit donc le type :

```
type arbre = Nil | Noeud of (bool * arbre * arbre) ;;
```

Question 1. Écrire en Caml une fonction **cherche** qui prend en arguments un entier n et un ensemble E (représenté par un arbre radix) et qui renvoie un booléen déterminant l'appartenance de n à E.

```
cherche : int -> arbre -> bool
```

On rappelle que si `n` est de type *int*, alors `n / 2` et `n mod 2` calculent respectivement le quotient et le reste de la division euclidienne de n par 2.

Question 2. Écrire en Caml une fonction **ajoute** qui prend en arguments un entier n et un ensemble E et qui retourne l'ensemble $E \cup \{x\}$.

```
ajoute : int -> arbre -> arbre
```

Question 3. En utilisant la fonction obtenue à la question précédente, écrire en Caml une fonction **construit** construisant un ensemble à partir de la liste des entiers qu'il contient.

```
construit : int list -> arbre
```

Question 4. Rédiger en Caml une fonction **supprime** prenant en arguments un entier n et un ensemble E et retournant l'ensemble $E \setminus \{n\}$ (si n n'appartient pas à E, cette fonction ne doit pas échouer mais retourner le même ensemble E).

```
supprime : int -> arbre -> arbre
```

Question 5. Écrire en Caml une fonction **union** réalisant l'union de deux ensembles.

```
union : arbre -> arbre -> arbre
```

Question 6. Écrire en Caml une fonction **intersection** réalisant l'intersection de deux ensembles.

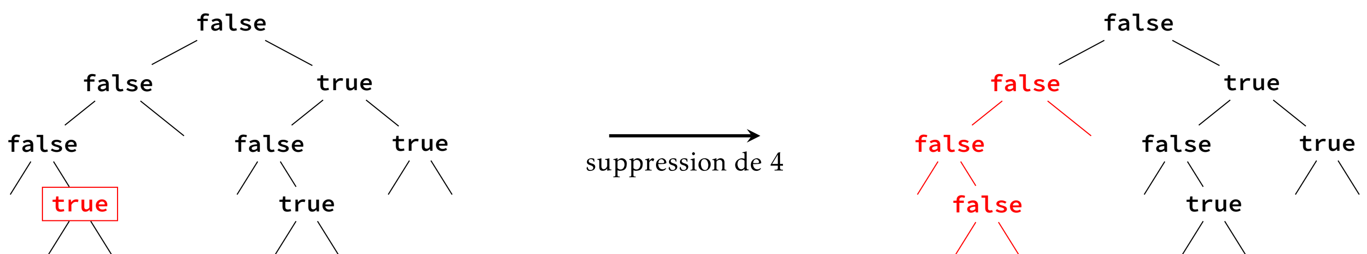
```
intersection : arbre -> arbre -> arbre
```

Et pour les plus rapides

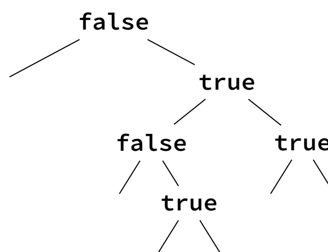
Question 7. (difficile) Rédiger en Caml une fonction **elements** retournant la liste de tous les éléments d'un ensemble (l'ordre des éléments importe peu, mais chaque élément doit apparaître une fois et une seule dans la liste).

```
elements : arbre -> int list
```

Question 8. (difficile) Les fonctions **supprime** et **intersection** ne sont pas satisfaisantes car elles peuvent faire apparaître des « branches mortes ». C'est le cas par exemple lorsqu'on supprime 4 de l'arbre radix représentant l'ensemble $\{1, 3, 4, 5\}$:



L'ensemble restant $\{1, 3, 5\}$ gagnerait à être représenté par l'arbre :



et c'est pourquoi on impose l'invariant suivant : l'arbre ne contient aucun nœud étiqueté par **false** dont les deux sous-arbres sont vides, c'est à dire de la forme

Rédiger en Caml une fonction **elague** qui prend en argument un arbre radix et qui retourne un arbre radix représentant le même ensemble et respectant l'invariant énoncé ci-dessus. En d'autres termes, cette fonction doit élaguer de l'arbre toutes ses branches mortes.

```
elague : arbre -> arbre
```