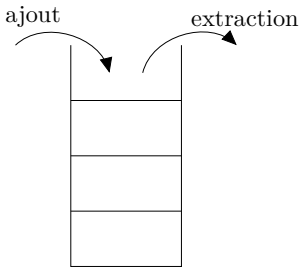
- Une structure de données **abstraite** est la description des opérations que doit comporter cette structure (exemple : dictionnaire).
  Une **implémentation** d'une structure de données est la réalisation pratique de cette structure. Il peut y avoir plusieurs implémentations de la même structure de données abstraite (exemple : un dictionnaire peut être implémenté par un arbre binaire ou une table de hachage).

- Une structure de données est **mutable** si on peut modifier ses éléments. Sinon, elle est **immuable** (ou : persistante).
  Par exemple, un algorithme de tri sera de type `'a list -> 'a list` pour une liste (on ne peut pas modifier la liste en argument donc on en renvoie une nouvelle) et `'a array -> unit` pour un tableau (la modification d'un tableau à l'intérieur de la fonction se répercute à l'extérieur).
  - Mutable : `array`, `ref`, `mutable` dans un type enregistrement.
  - Immuable : `list`, `string`, `tuple`, `arbre`.

  Les structures immuables sont plus adaptées à la programmation récursive et les structures mutables à la programmation impérative (boucles, références).

- Une **pile** est une structure de donnée possédant trois opérations :
  - `push` : Ajout d'un élément au dessus de la pile.
  - `pop` : Extraction (suppression et renvoi) de l'élément au dessus de la pile. Ainsi, c'est toujours le dernier élément rajouté qui est extrait.
  - `is_empty` : Test pour savoir si la pile est vide.



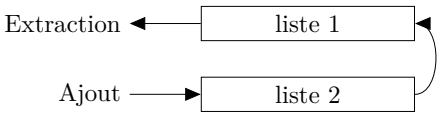
  C'est une structure LIFO : Last In First Out (dernier entré, premier sorti).
  On peut implémenter une pile avec une liste, où la tête de liste est le dessus de la pile.

```
let push x s = x::s

(* pop s renvoie (x,s') où x est l'élément extrait
et s' le reste de la pile *)
let pop s = match s with
    | [] -> failwith "pop"
    | x::s -> (x,s)

let is_empty s = s = []
```

  Applications :
  - Pile d'appel qui stocke les appels de fonctions en cours d'exécution.
  - Passer de récursif à itératif en simulant des appels récursifs (ex : DFS).

- Une **file** est une structure de donnée possédant trois opérations:
  - Ajout d'un élément à la fin de la file.
  - Extraction (suppression et renvoi) de l'élément au début de file. Ainsi, l'élément extrait est l'élément le plus ancien.
  - Test pour savoir si la file est vide.

  C'est une structure FIFO : First In First Out (premier entré, premier sorti).
  Implémenter une file avec une liste OCaml ne serait pas efficace, car accéder au dernier élément est en $O(n)$.
  À la place, on peut implémenter une file avec deux listes `f1` et `f2`, en remplaçant `f1` par `f2` (inversé) si `f1` est vide :

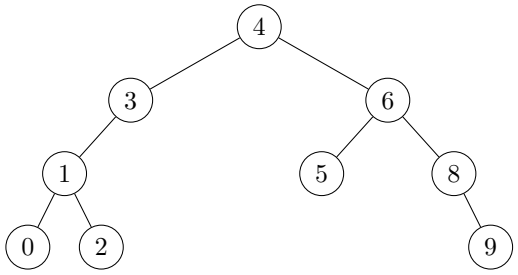



```
type 'a file = 'a list * 'a list

let empty = ([], []) (* file vide *)

let is_empty (f1,f2) = f1 = [] && f2 = []

let add x (f1, f2) = (f1, x::f2)

(* suppose que la file n'est pas vide *)
let rec extract (f1, f2) = match f1 with
    | [] -> extract ([], List.rev f2)
    | x::q1 -> (x, (q1, f2))
```

  Dans le pire des cas, un appel à `extract` sera en $O(n)$ car `List.rev` est en $O(n)$...
  Par contre, la complexité amortie (c'la complexité moyennée sur plusieurs appels) est meilleure. En effet, si on effectue n ajouts et n extractions dans un ordre quelconque (en partant d'une file vide), chaque élément est « renversé » exactement une fois, donc la complexité totale des `List.rev` est $O(n)$, d'où une complexité amortie de $O(1)$.
  Remarque : On peut aussi donner une implémentation mutable d'une file avec une tableau.
  Application : Parcours en largeur.

- Un **dictionnaire** associe à chaque **clé** une **valeur** avec les opérations :
  - Ajouter une association (clé, valeur)
  - Supprimer une association (clé, valeur)
  - Recherche de la valeur associée à une clé

- On peut implémenter un dictionnaire par arbre binaire de recherche.
  Un **arbre binaire de recherche** (ABR) est un arbre binaire tel que, pour chaque noeud d'étiquette r et de sous-arbres g et d, r est supérieur à toutes les étiquettes de g et inférieur à toutes les étiquettes de d.
  Il faut donc une relation d'ordre sur les étiquettes.



Exemple d'ABR

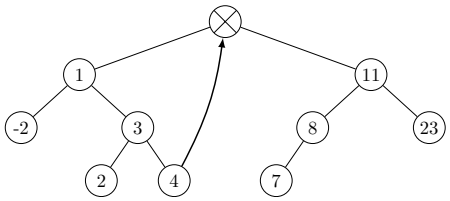
Pour chercher si un élément e appartient à un ABR $N(r, g, d)$, il suffit de regarder dans g si $e < r$ ou dans d si $e > r$.

```
let rec appartient e a = match a with
    | V -> false
    | Noeud (r, g, d) ->
        if e = r then true
        else if e < r then appartient e g
        else appartient e d
```

`appartient` est en $O(h)$ où h est la hauteur de l'arbre, car on ne parcourt qu'une branche de l'arbre.
Pour ajouter un élément e à un ABR $N(r, g, d)$, on peut soit le placer dans g si $e < r$ soit dans d si $e > r$, de façon à conserver la propriété d'ABR.

```
let rec ajoute e = function
    | V -> N(e, V, V) (* ajoute e comme feuille *)
    | N(r, g, d) when e < r -> N(r, ajoute e g, d)
    | N(r, g, d) -> N(r, g, ajoute e d)
```

On peut supprimer un élément en le remplaçant par le maximum de son sous-arbre gauche (qui est tout à droite de l'arbre), de façon à conserver la propriété d'ABR.




```
(* supprimer_max a renvoie le maximum de l'arbre
et l'arbre sans ce maximum *)
let rec supprimer_max = function
    | V -> min_int, V
    | N(r, g, V) -> r, g
    | N(r, g, d) -> let m, d' = del_max d in
                    m, N(r, g, d')

(* supprimer e a renvoie un ABR contenant les
mêmes éléments que a sauf e *)
let rec supprimer e = function
    | E -> E
    | N(r, g, d) when e = r -> let m, g' = del_max g in
                               N(m, g', d)
    | N(r, g, d) when e < r -> N(r, del e g, d)
    | N(r, g, d) -> N(r, g, del e d)
```

où chaque noeud contient un couple (clé, valeur) et les sous-arbres gauche et droit sont les sous-dictionnaires associés aux clés inférieures et supérieures à la clé du noeud.

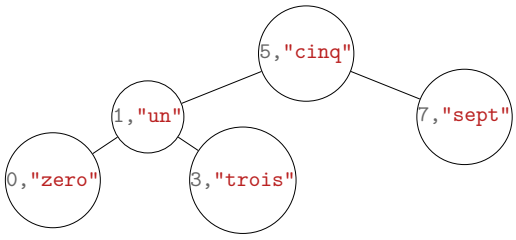
- On peut implémenter un dictionnaire avec un ABR dont les noeuds contiennent des couples (clé, valeur) où les étiquettes sont comparées sur les clés :

```
type ('k, 'v) dict =
    V | N of ('k * 'v) * ('k, 'v) dict * ('k, 'v) dict
(* 'k est le type des clés et 'v le type des valeurs *)
```

Les fonctions d'ajout et de suppression sont alors données par `ajoute`, `supprime`, et on peut adapter `appartient` pour pouvoir chercher une clé :

```
(* recherche k d renvoie None si k n'est pas une clé une d,
et Some v si k est associée à la valeur v *)
let rec recherche k = function
    | V -> None
    | N((k', v), g, d) when k = k' -> Some v
    | N((k', v), g, d) when k < k' -> recherche k g
    | N((k', v), g, d) -> recherche k d
```

Remarque : On a utilisé le type prédéfini `type 'a option = None | Some of 'a`.



Exemple de dictionnaire implémenté par ABR

- Implémenter un dictionnaire par ABR demande une relation d'ordre sur les clés et toutes les fonctions ci-dessus sont en $O(h)$. Une autre possibilité est d'utiliser une table de hachage où les complexités peuvent être $O(1)$ en moyenne, mais il faut bien choisir la fonction de hachage.