Outils logiques et algorithmiques

Thibaut Balabonski @ Université Paris-Saclay
Édition 2024.

Troisième partie

Arbres

8 T :: r :: i :: e :: r :: []

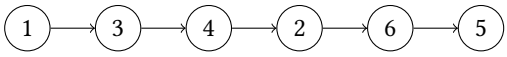
8.1 Problème : tri d'une liste chaînée

On revient à la question du tri déjà explorée sur les tableaux aux chapitres 2 et 3. Cependant, on s'intéresse maintenant à une autre forme de structure séquentielle : la liste chaînée. Cette structure est composée de *cellules*, dont chacune contient deux choses :

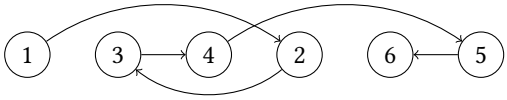
— un élément de la séquence, et

— un pointeur vers la cellule suivante.

Chaque cellule est donc un maillon d'une chaîne, et on peut parcourir la séquence d'éléments en partant de la première cellule et en suivant à chaque étape le pointeur vers la cellule suivante, jusqu'à arriver au bout de la chaîne (lorsqu'il n'y a plus de suivante).



Trier une liste chaînée, c'est produire une nouvelle configuration de ces pointeurs vers les cellules suivantes, de sorte à ce que suivre ces nouveaux pointeurs fasse parcourir les éléments en ordre croissant.



Trier une telle structure *en place* consisterait à conserver toutes les cellules et modifier leurs pointeurs. Cependant, les programmes basés sur des manipulations de pointeurs sont généralement complexes à mettre au point et à analyser, et un tel tri en place ne ferait pas exception [3]. On se restreint donc ici au cas des *listes immuables*, plus simples et souvent préférables. On voudra ainsi produire une *nouvelle liste*, composée de nouvelles cellules, comportant les mêmes éléments que la liste d'origine rangés par ordre croissant, *sans modifier la liste d'origine.*

8.2 Caractérisation récursive des listes chaînées

Une liste chaînée a une structure naturellement récursive, que l'on peut faire ressortir en reformulant la description ainsi :

Une liste non vide est une paire, formée par un élément et une liste.

Autrement dit, une liste est une structure de données pouvant prendre deux formes :

1. la liste vide, que l'on notera [],
2. une paire $e :: \ell$ formée par un élément e (la *tête*) et une liste ℓ (la *queue*).

On qualifie cette structure de *récursive* car une liste (non vide) contient elle-même une liste. En inversant le point de vue, à partir d'une liste ℓ donnée, on peut construire une liste plus grande en ajoutant un élément e en tête.

En appelant *longueur* d'une liste le nombre d'éléments qu'elle contient, on peut rendre encore plus précise la caractérisation précédente. Une liste est soit :

1. la liste vide [] (de longueur zéro),
2. une liste $e :: \ell$ de longueur $n > 0$, dont la queue ℓ est une liste de longueur $n - 1$.

3. Pour les curieux, l'une des sections d'approfondissement y est dédiée.

Notre liste exemple contenant la séquence d'éléments 1, 3, 4, 2, 6, 5 a une longueur de 6, et pourra être notée

$$1 :: (3 :: (4 :: (2 :: (6 :: (5 :: [])))))$$

Par convention, on donne aux opérations :: enchaînées un parenthésage implicite à droite, ce qui permet de simplifier la notation précédente en $1 :: 3 :: 4 :: 2 :: 6 :: 5 :: []$.

En java. Pour représenter une structure de liste chaînée immuable en java, il suffit de définir une unique classe comportant deux champs `head` et `tail`, respectivement pour la tête et la queue (et de déclarer ces deux champs immuables).

```
class List {
  final int  head;
  final List tail;

  List(int x, List l) {
    this.head = x;
    this.tail = l;
  }
}
```

On prend en plus comme convention que la liste vide est simplement représentée par le pointeur **`null`**, et on peut définir une liste contenant les éléments 1, 2, 3 par :

```
List l = new List(1, new List(2, new List(3, null)))
```

En caml. Les listes chaînées immuables sont un type de base en caml, utilisant les mêmes notations que ci-dessus, à savoir `[]` pour la liste vide et `::` pour l'ajout d'un élément en tête d'une liste. On peut donc définir la liste contenant les éléments 1, 2, 3 par :

```
let l = 1 :: (2 :: (3 :: []))
```

ou encore avec l'une des deux formes abrégées

```
let l = 1 :: 2 :: 3 :: []
let l = [1; 2; 3]
```

Si on voulait redéfinir manuellement un clone de ce type des listes chaînées, on utiliserait une définition de type algébrique, avec un constructeur `Nil` pour la liste vide, et un constructeur `Cons` prenant en paramètres un élément et une liste pour l'ajout d'un élément.

```
type list =
  | Nil
  | Cons of int * list
```

La liste avec les éléments 1, 2, 3 serait alors définie par :

```
let l = Cons(1, Cons(2, Cons(3, Nil)))
```

8.3 Fonctions récursives sur des listes chaînées

La caractérisation des listes en deux cas ([] ou $e :: \ell$) peut être utilisée pour définir des fonctions manipulant les listes. Dans les cas les plus simples, il suffit de deux équations pour définir une fonction f : une donnant le résultat $f([])$ pour la liste vide, et une exprimant le résultat $f(e :: \ell)$ pour une liste non vide. La deuxième équation pourra faire intervenir la valeur $f(\ell)$ prise par f sur la queue ℓ de la liste considérée.

Ainsi, une fonction longueur calculant la longueur d'une liste peut être définie par

$$\begin{cases} \mathsf{longueur}([]) &= 0 \\ \mathsf{longueur}(e :: \ell) &= 1 + \mathsf{longueur}(\ell) \end{cases}$$

De même, une fonction concat renvoyant la concaténation de deux listes ℓ_1 et ℓ_2, c'est-à-dire une liste formée en faisant se suivre, dans l'ordre, les éléments de ℓ_1 et les éléments de ℓ_2, peut être définie par

$$\begin{cases} \mathsf{concat}([], \ell_2) &= \ell_2 \\ \mathsf{concat}(e :: \ell_1, \ell_2) &= e :: \mathsf{concat}(\ell_1, \ell_2) \end{cases}$$

On pourrait alors observer les calculs détaillés suivants.

longueur(1 :: (2 :: (3 :: [])))	concat(1 :: (2 :: (3 :: [])), ℓ)
= 1 + longueur(2 :: (3 :: []))	= 1 :: concat(2 :: (3 :: []), ℓ)
= 2 + longueur(3 :: [])	= 1 :: (2 :: concat(3 :: [], ℓ))
= 3 + longueur([])	= 1 :: (2 :: (3 :: concat([], ℓ)))
= 3	= 1 :: (2 :: (3 :: ℓ))

En java. De telles équations donnent un modèle pour écrire un programme réalisant la fonction f. Le code repose sur un test permettant de différencier les deux formes possibles de la liste `l` donnée en argument : la liste est-elle vide, ou non ?

```
static int length(List l) {
  if (l == null) return 0;
  else           return 1 + length(l.tail);
}
static List concat(List l1, List l2) {
  if (l1 == null) return l2;
  else            return new List(l1.head, concat(l1.tail, l2));
}
```

À noter : le calcul de `length(l)` lorsque `l` n'est pas vide, repose sur un appel récursif sur la queue de la liste. Ce code, obtenu à partir d'équations récursives, est lui-même récursif ! Pour `concat`, on obtient de même une fonction récursive, qui travaille sur le premier des deux arguments. Dans cette deuxième fonction, le **`new`** `List` rend explicite la création d'une nouvelle cellule : la cellule de tête de `l1` est consultée, mais pas modifiée.

En caml. Des équations définissant une fonction sur une liste, on déduit aussi un code caml, centré sur une opération de filtrage énumérant les formes possibles de la liste.

```
let rec length l = match l with
  | []      -> 0
  | e :: l' -> 1 + length l'

let rec concat l1 l2 = match l1 with
  | []     -> l2
  | e :: l -> e :: concat l l2
```

Le code prend cette fois directement la forme d'une série d'équations. À noter : le *motif de filtrage* `e :: l'` ne fait pas que tester la forme « une liste dotée d'une tête et d'une queue », il attribue aussi des noms `e` et `l'` aux différents constituants (on peut utiliser à la place d'une telle lettre le symbole _ si on n'a pas besoin de nommer un constituant). On pourrait écrire également les deux versions allégées ci-dessous de la fonction `length` pour le même effet.

```
let rec length l = match l with
  | []     -> 0
  | _ :: l -> 1 + length l
```

```
let rec length = function
  | []     -> 0
  | _ :: l -> 1 + length l
```

Approfondissement : éviter les débordements de pile. Dans cet exemple, le code obtenu en traduisant directement les équations contient des appels récursifs inutilement coûteux. En effet, le simple fait d'appeler une fonction (récursive ou non) a un coût, qui vient s'ajouter au coût du travail réalisé par la fonction. Ce coût intrinsèque de l'appel étant modeste, nous n'en avons pas tenu compte dans les chapitres précédents. Cependant, la répétition due à la récurrence peut finir par le rendre visible. Appliquées à de grandes listes, les fonctions précédentes peuvent même provoquer une interruption du programme avec une erreur de « dépassement de pile » (*stack overflow*). Et ceci aussi bien en java qu'en caml. Des programmeurs avisés préféreraient certainement les solutions suivantes.

— En java, écrire la fonction `length` avec une boucle énumérant les cellules de la liste et incrémentant une variable `r`.

```
static int length(List l) {
  int r = 0;
  for (; l != null; l = l.tail) { r++; }
  return r;
}
```

— En caml, faire la récursion avec une fonction auxiliaire récursive terminale prenant deux paramètres : la longueur `r` du préfixe déjà parcouru, et la liste `l` restant à parcourir. Cette fonction auxiliaire est appelée en initialisant `r` à zéro, et `l` à la liste complète.

```
let length l =
  let rec loop r l = match l with
    | []     -> r
    | _ :: l -> loop (r+1) l
  in loop 0 l
```

L'appel récursif à `loop` est *terminal* : son résultat est directement transmis comme résultat de l'appel principal. Le compilateur caml reconnaît cette situation et l'optimise, de sorte à nous dispenser du coût de l'appel lui-même [4].

Le paramètre supplémentaire `r` donné à la fonction récursive terminale dans le code caml peut être comparé à la variable `r` qui est mise à jour dans le code java avec boucle. Ces deux versions améliorées ont d'ailleurs, à l'exécution, des comportements très similaires.

Nous verrons bientôt d'autres cas, dans lesquels même des programmeurs aguerris ne trouveraient rien à redire au code directement traduit des équations récursives.

8.4 Raisonnement récursif

En association avec le caractère récursif des listes chaînées et des fonctions les manipulant, il est possible de raisonner par récurrence sur les listes.

Principe de récurrence structurelle. Pour montrer qu'une certaine propriété $P(\ell)$ est valable pour toute liste ℓ, il suffit de s'assurer que la propriété est vraie pour la liste vide, et est préservée par ajout d'un élément en tête. C'est-à-dire qu'il suffit de justifier que :

— $P(\texttt{[]})$ est vraie (cas de base),

— pour toute liste ℓ et tout élément e, si $P(\ell)$ est valide alors $P(e :: \ell)$ est valide également (cas récursif).

On en déduit que la propriété P est valide pour toute liste que l'on puisse construire en ajoutant successivement des éléments en tête de la liste vide, c'est-à-dire pour toute liste.

Exemple. Supposons que l'on souhaite montrer la propriété suivante sur la concaténation de deux listes :

$$\forall \ell_1, \ell_2, \quad \texttt{longueur}(\texttt{concat}(\ell_1, \ell_2)) = \texttt{longueur}(\ell_1) + \texttt{longueur}(\ell_2)$$

Pour raisonner par récurrence structurelle sur ℓ_1, on s'intéresse à la propriété $P(\ell_1)$ définie par la formule

$$\forall \ell_2, \quad \texttt{longueur}(\texttt{concat}(\ell_1, \ell_2)) = \texttt{longueur}(\ell_1) + \texttt{longueur}(\ell_2)$$

Détaillons les deux cas de la récurrence structurelle sur ℓ_1.

— Cas de base, où ℓ_1 est la liste vide : il faut montrer $P(\texttt{[]})$, c'est-à-dire

$$\forall \ell_2, \quad \texttt{longueur}(\texttt{concat}(\texttt{[]}, \ell_2)) = \texttt{longueur}(\texttt{[]}) + \texttt{longueur}(\ell_2)$$

Comme souvent ce cas est facile. Ici il suffit de remarquer que, par définition de `concat` on a $\texttt{concat}(\texttt{[]}, \ell_2) = \ell_2$, et que par définition de `longueur` on a $\texttt{longueur}(\texttt{[]}) = 0$.

— Cas récursif, où on suppose que $P(\ell_1)$ est vraie et où on veut démontrer que $P(e :: \ell_1)$ est vraie. Pour une certaine liste ℓ_1, on suppose donc que

$$\forall \ell_2, \quad \texttt{longueur}(\texttt{concat}(\ell_1, \ell_2)) = \texttt{longueur}(\ell_1) + \texttt{longueur}(\ell_2)$$

et on veut montrer que

$$\forall \ell_2, \quad \texttt{longueur}(\texttt{concat}(e :: \ell_1, \ell_2)) = \texttt{longueur}(e :: \ell_1) + \texttt{longueur}(\ell_2)$$

4. Cette optimisation n'existe que dans peu de langages. En plus de caml, on peut citer C.

La première propriété est appelée *hypothèse de récurrence*. On peut détailler le calcul ainsi

$$
\begin{aligned}
&\texttt{longueur}(\texttt{concat}(e :: \ell_1, \ell_2)) \\
=\ &\texttt{longueur}(e :: (\texttt{concat}(\ell_1, \ell_2))) && \text{par déf. de } \texttt{concat} \\
=\ &1 + \texttt{longueur}(\texttt{concat}(\ell_1, \ell_2)) && \text{par déf. de } \texttt{longueur} \\
=\ &1 + (\texttt{longueur}(\ell_1) + \texttt{longueur}(\ell_2)) && \text{par hyp. de récurrence} \\
=\ &(1 + \texttt{longueur}(\ell_1)) + \texttt{longueur}(\ell_2) \\
=\ &\texttt{longueur}(e :: \ell_1) + \texttt{longueur}(\ell_2) && \text{par déf. de } \texttt{longueur}
\end{aligned}
$$

On peut rapprocher le raisonnement précédent d'un raisonnement par récurrence classique sur les nombres entiers :

- dans le cas de base on considère la liste vide, qui est la seule liste de longueur 0, et
- dans le cas récursif on montre que la propriété est vraie pour une liste de longueur n+1, en partant de l'hypothèse qu'elle est vraie pour une liste de longueur n.

Deux preuves pour s'exercer : $\forall \ell,\ \texttt{concat}(\ell, \texttt{[]}) = \ell$ et $\forall \ell_1, \ell_2, \ell_3,\ \texttt{concat}(\ell_1, \texttt{concat}(\ell_2, \ell_3)) = \texttt{concat}(\texttt{concat}(\ell_1, \ell_2), \ell_3)$.

8.5 Algorithme : tri insertion

Voici une réalisation en caml du tri par insertion de listes. Note : contrairement à ce qu'on a vu jusque-là sur les tableaux, mais comme annoncé au début du chapitre, ce tri n'est pas « en place », c'est-à-dire qu'il ne modifie pas la liste d'origine. À la place, on renvoie une nouvelle liste, contenant les éléments dans l'ordre. Il se trouve que ce fait, allié à la récursion, rend le raisonnement nettement plus simple.

```
let rec insert x l = match l with
  | []      -> [x]
  | y :: l' -> if x <= y then x :: l
                         else y :: insert x l'

let rec insertion_sort = function
  | []     -> []
  | x :: l -> insert x (insertion_sort l)
```

Spécifications de ces deux fonctions :

- `insertion_sort` renvoie une permutation triée de la liste donnée en argument.
- `insert` prend en paramètres un élément x et une liste triée ℓ, et renvoie une permutation triée de $x :: \ell$.

Savez-vous le faire avec des fonctions récursives terminales ? Serait-ce utile ici ?

Preuves de correction. On peut démontrer par récurrence que nos deux fonctions correspondent bien à ces spécifications. Pour `insert` :

- Cas de base : $\texttt{insert}(x, \texttt{[]}) = \texttt{[}x\texttt{]}$, et $\texttt{[}x\texttt{]}$ est bien une permutation triée de $x :: \texttt{[]}$.
- Cas récursif : on suppose que $y :: \ell$ est triée et on veut montrer que $\texttt{insert}(x, y :: \ell)$ est une permutation triée de $x :: y :: \ell$. Deux cas en fonction de la comparaison de x et y.
  - Si $x \leq y$ alors $\texttt{insert}(x, y :: \ell) = x :: y :: \ell$ et cette liste est triée et est bien une permutation d'elle-même.
  - Si $x > y$ alors $\texttt{insert}(x, y :: \ell) = y :: \texttt{insert}(x, \ell)$. Par hypothèse de récurrence $\texttt{insert}(x, \ell)$ est une permutation triée de $x :: \ell$. La liste $y :: \ell$ était supposée triée : pour tout $z \in \ell$ on a donc $y \leq z$. Donc $y :: \texttt{insert}(x, \ell)$ est triée. En outre, cette liste est une permutation de $y :: x :: \ell$, et donc également une permutation de $x :: y :: \ell$.

Pour `insertion_sort` :

- Cas de base : $\texttt{insertion\_sort}(\texttt{[]}) = \texttt{[]}$, résultat immédiat.
- Cas récursif : on veut montrer que $\texttt{insertion\_sort}(x :: \ell)$ est une permutation triée de $x :: \ell$. Le code donne $\texttt{insertion\_sort}(x :: \ell) = \texttt{insert}(x, \texttt{insertion\_sort}(\ell))$. Par hypothèse de récurrence $\texttt{insertion\_sort}(\ell)$ est une permutation triée de ℓ. En particulier $\texttt{insertion\_sort}(\ell)$ est triée et `insert` s'applique bien. Par spécification de `insert`, $\texttt{insert}(x, \texttt{insertion\_sort}(\ell))$ est une permutation triée de $x :: \texttt{insertion\_sort}(\ell)$, c'est-à-dire une permutation triée de $x :: \ell$.

Du côté des complexités, on pourrait calculer assez facilement que le tri insertion sur des listes chaînées, comme le tri insertion d'un tableau, est quadratique en la longueur de la séquence à trier.

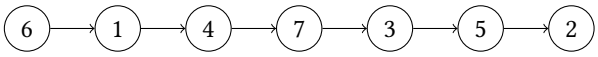
Arrivez-vous bien à le faire ?

En java. Pour information, voici une version java des fonctions précédentes. Les preuves de correction pour cette version seraient identiques à celles du paragraphe précédent.

```
static List insert(int x, List l) {
  if (l == null)        return new List(x, null);
  else if (l.head >= x) return new List(x, l);
  else                  return new List(l.head, insert(x, l.tail));
}

static List insertionSort(List l) {
  if (l == null) return null;
  else           return insert(l.head, insertionSort(l.tail));
}
```

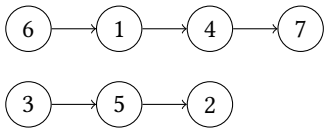
8.6 Algorithme : tri fusion

On peut trier une liste chaînée en un temps linéarithmique en la longueur de la liste. On présente pour cela l'algorithme de *tri fusion*[5].

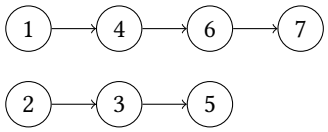


L'idée est la suivante :

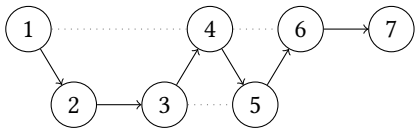
1. Séparer la liste en deux sous-listes de tailles égales (à un arrondi près si la liste à séparer a une taille impaire).



2. Trier chacune des deux sous-listes. Récursivement, bien sûr.



3. Reformer une unique liste triée à partir des deux sous-listes triées. C'est cette dernière étape qu'on appelle *fusion*.



En caml. On propose d'abord deux fonctions `take` et `chop`, qui respectivement, prennent les `n` premiers d'une liste ou ne conservent d'une liste que les éléments situés après les `n` premiers. Ces deux fonctions, appliquées avec un même `n` ciblant la moitié de la liste, permettent de la séparer en deux parties égales. Note : dans le code de la première fonction, `List.hd` et `List.tl` sont deux fonctions de la bibliothèque standard qui extraient respectivement la tête et la queue d'une liste.

```
let rec take n l =
  if n=0 then []
  else (List.hd l) :: (take (n-1) (List.tl l))

let rec chop n l =
  if n=0 then l
  else chop (n-1) (List.tl l)
```

5. Algorithme aperçu dans le TD3 dans le cas du tri en place d'un tableau.

La fonction principale `mergesort` découpe la liste donnée en argument en deux à l'aide de `take` et `chop`, trie les deux moitiés récursivement, puis les assemble à l'aide d'une autre fonction auxiliaire `merge`. À moins que la liste d'origine soit si petite qu'il soit certain qu'elle est déjà triée (c'est-à-dire : si elle ne contient que zéro ou un élément).

```
let rec mergesort l =
  let n = length l in
  if n < 2 then l
  else let k = (n+1)/2 in
       let l1, l2 = take k l, chop k l in
       merge (mergesort l1) (mergesort l2)
```

La fonction de fusion prend en paramètres deux listes `l1` et `l2` supposées triées, et produit une unique liste triée entrelaçant les éléments de `l1` et `l2`. Elle est également récursive : elle compare le premier élément de `l1` et le premier élément de `l2`, sélectionne le plus petit des deux, puis fusionne ce qui reste. Le processus s'arrête lorsque l'une des deux listes à fusionner est vide : il suffit alors de conserver l'autre intégralement.

Pour lesquelles de ces fonctions serait-il préférable d'utiliser une variante récursive terminale ? Sauriez-vous les écrire ?

```
let rec merge l1 l2 = match l1, l2 with
  | [], l | l, []    -> l
  | x1::tl1, x2::tl2 -> if x1 <= x2 then x1 :: (merge tl1  l2)
                                    else x2 :: (merge  l1 tl2)
```

En java. Voici une version java du tri fusion présenté ci-dessus en caml. Les deux vont s'analyser exactement de la même façon.

```
static List take(int n, List l) {
  if (n == 0) return null;
  else        return new List(l.head, take(n-1, l.tail));
}
static List chop(int n, List l) {
  if (n == 0) return l;
  else        return chop(n-1, l.tail);
}

static List merge(List l1, List l2) {
  if (l1 == null) return l2;
  if (l2 == null) return l1;
  if (l1.head <= l2.head) return new List(l1.head, merge(l1.tail, l2     ));
  else                    return new List(l2.head, merge(l1     , l2.tail));
}

static List mergeSort(List l) {
  int n = length(l);
  if (n < 2) return l;
  int k = (n+1)/2;
  List l1 = take(k, l), l2 = chop(k, l);
  return merge(mergeSort(l1), mergeSort(l2));
}
```

8.7 Approfondissement : analyse du tri fusion

Spécification. Précisons les spécification des quatre fonctions formant notre tri fusion.

- Un appel `chop n l` renvoie la liste formée en retirant à `l` ses `n` premiers éléments. Précondition : la liste `l` doit contenir au moins `n` éléments.
- Un appel `take n l` renvoie la liste formée avec les `n` premiers éléments de `l`, dans l'ordre. Précondition : la liste `l` doit contenir au moins `n` éléments.
- Un appel `merge l1 l2` renvoie une liste triée formée des éléments des listes `l1` et `l2`. Précondition : `l1` et `l2` doivent elle-mêmes être triées.
- Un appel `mergesort l` renvoie une liste triée formée des éléments de la liste `l`.

Terminaison. Nos quatre fonctions récursives terminent à coup sûr sur toute entrée. On le vérifie à l'aide de la technique du variant. Rappel : adaptée pour justifier la terminaison d'une fonction récursive f, la technique demande d'identifier une quantité positive dépendant des paramètres de f, qui décroît strictement à chaque appel récursif. On peut choisir les variants suivants :

- Pour `take` et `chop`, le paramètre `n` est lui-même un variant : c'est un entier supposé positif ou nul, et l'unique appel récursif présent dans le code de `take` (ou `chop`) utilise la valeur `n-1`.
- Pour `mergesort`, on prend comme variant la longueur n de la liste `l` donnée en paramètre. Les appels récursifs se font avec des listes de longueurs $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor$, et uniquement dans le cas où $n \geq 2$: on a bien décroissance.
- Pour `merge`, on prend comme variant la somme des longueurs des deux listes `l1` et `l2`. Il s'agit bien d'un entier positif, car les longueurs sont elle-mêmes des entiers positifs. En outre, chaque appel récursif s'applique à : l'une des deux listes telle qu'elle a été donnée, et la queue de l'autre liste. On ne sait pas à l'avance quelle liste va décroître, mais on sait qu'une des deux le fait : la somme va bien décroître (en l'occurrence, de précisément 1).

Correction. Les variants identifiés dans l'analyse précédente nous disent également comment organiser notre raisonnement par récurrence, si l'on veut démontrer que nos fonctions récursives sont bien correctes.

- Pour `take` et `chop`, une récurrence simple sur `n` suffit.
- Pour `merge`, on peut faire une récurrence simple sur la somme des longueurs des deux listes.
- Pour `mergesort`, il faut faire une récurrence forte sur la longueur de la liste.

Complexité. Les fonctions `take` et `chop` ont manifestement une complexité proportionnelle à `n`. La fonction `merge` a une complexité linéaire en la somme des longueurs des deux listes passées en argument. Lors d'un appel `mergesort l` avec `l` de taille n, on a les opérations suivantes.

- Découpage de la liste `l` en deux moitiés avec `take` et `chop` : coût $\mathcal{O}(n)$.
- Deux appels récursifs sur des listes de tailles $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor$.
- Fusion des deux listes obtenues avec `merge`. La somme des longueurs de ces deux listes est n : le coût de cette étape est encore $\mathcal{O}(n)$.

La relation de récurrence pour la complexité $C(n)$ de `mergesort` sur les listes de longueur n a donc la forme

$$C(n) \quad = \quad 2C(\frac{n}{2}) + f(n)$$

où $f(n) = \mathcal{O}(n)$. On a déjà vu une telle équation dans le cas le plus favorable du tri rapide.

$$C(n) \quad = \quad \Theta(n \log(n))$$

Conclusion : le tri fusion de listes chaînées a une complexité linéarithmique.

8.8 Approfondissement : tri fusion, version récursive terminale (caml)

On améliore le tri fusion présenté en caml en s'assurant que les fonctions de découpage et de fusion sont récursives terminales. Au passage, on combine les fonctions `take` et `chop` en une seule fonction `split_at` renvoyant une paire de listes. Cette fonction utilise une fonction auxiliaire récursive terminale `split` telle que `split n l1 l2` place les `n` premiers éléments de `l1` en tête de `l2` (dans l'ordre inverse), et renvoie la paire formée par : ce qui reste de `l1`, et la version étendue de `l2`. Il suffit alors d'appliquer cette fonction auxiliaire en prenant pour `l1` la liste à découper et pour `l2` la liste vide.

```
let split_at n l =
  let rec split n l1 l2 = match n, l1 with
    | 0, _     -> l1, l2
    | n, x::tl -> split (n-1) tl (x :: l2)
  in split n l []
```

Remarque : en transférant les éléments en tête de `l1` vers `l2` on renverse leur séquence, mais cela ne perturbe pas le tri qui va suivre.

La nouvelle fonction de fusion est de même basée sur une fonction récursive terminale `merge l l1 l2` où : `l1` et `l2` sont ce qui reste à fusionner, et `l` est la séquence déjà fusionnée. À noter cependant : comme on n'ajoute les éléments qu'en tête, la séquence `l` est triée à l'envers. Ainsi, lorsque l'une des deux listes `l1` ou `l2` est épuisée et que l'on veut concaténer `l` et la liste restante, on le fait en renversant `l`. La fonction `rev_append` réalise cela [6].

```
let rec rev_append l1 l2 = match l1 with
  | []    -> l2
  | x::tl -> rev_append tl (x::l2)

let merge l1 l2 =
  let rec merge l l1 l2 = match l1, l2 with
    | [], l' | l', []  -> rev_append l l'
    | x1::tl1, x2::tl2 -> if x1 <= x2 then merge (x1::l) tl1  l2
                                      else merge (x2::l)  l1 tl2
  in merge [] l1 l2
```

La fonction `mergesort` ne change pas, si ce n'est pour s'adapter au remplacement de `take` et `chop` par l'unique fonction `split_at`. On ne peut pas rendre `mergesort` récursive terminale car le tri récursif des deux moitiés est suivi d'une opération de fusion. Cependant, on ne risque pas de dépassement de pile ici puisque le nombre d'appels emboîtés à `mergesort` est logarithmique en la taille de la liste.

```
let rec mergesort l =
  let n = length l in
  if n < 2 then l
  else let l1, l2 = split_at ((n+1)/2) l in
       merge (mergesort l1) (mergesort l2)
```

8.9 Approfondissement : tri fusion en place (java)

Listes mutables. Reprenons notre définition des listes en java, et autorisons cette fois la mutation du pointeur `tail` vers la cellule suivante.

```
class List {
  final int head;
  List tail;
  ...
}
```

Certaines des fonctions déjà écrites, comme `length`, n'ont pas vocation à changer. On peut en revanche imaginer une nouvelle spécification pour la fonction `concat` : au lieu de créer une nouvelle liste réunissant les éléments de ses deux paramètres `l1` et `l2`, on peut lui demander de *modifier* physiquement `l1` pour que le pointeur `tail` de sa dernière cellule soit maintenant dirigé vers `l2`.

```
static void concat(List l1, List l2) {
  assert (l1 != null);
  for(; l1.tail != null; l1 = l1.tail) {}
  l1.tail = l2;
}
```

Note : par sa spécification même, cette version n'a de sens que si `l1` contient bien au moins une cellule ! Pour une variante fonctionnant également avec `l1` vide, il faut encore une autre spécification. En voici une possible :

— si `l1` n'est pas vide, la fonction doit rediriger le dernier pointeur `tail` de `l1` vers `l2`,

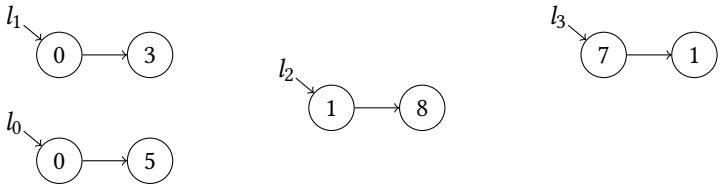
— et dans tous les cas, la fonction doit renvoyer la première cellule de la liste fusionnée.

Cette nouvelle spécification change même le type de la fonction, qui doit maintenant renvoyer (un pointeur vers) une cellule. Il s'agira en général de la première cellule de `l1`, sauf dans le cas où `l1` est vide, où on renvoie à la place la première cellule de `l2`.

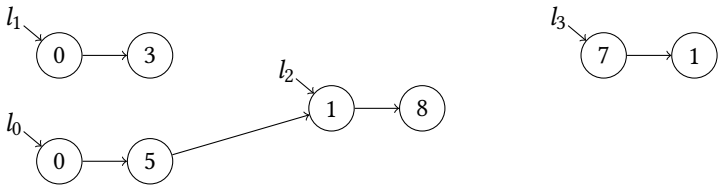
6. Fonction présente dans la bibliothèque standard, et analysée prochainement en TD.

```
static List concat(List l1, List l2) {
  if (l1 == null) return l2;
  List c = l1;
  for(; c.tail != null; c = c.tail) {}
  c.tail = l2;
  return l1;
}
```

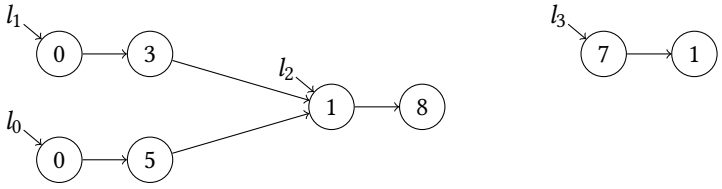
Pièges des listes mutables. Lorsqu'elles sont autorisées, les modifications de listes peuvent rapidement devenir acrobatiques, car elles introduisent des phénomènes de *partage*, ou *aliasing* : au fil du temps, plusieurs fragments de listes peuvent être reliés, et une modification d'une liste peut avoir des répercussions sur une autre. Par exemple, partons de quatre listes `l0`, `l1`, `l2` et `l3` contenant respectivement les séquences `{0, 5}`, `{0, 3}`, `{1, 8}` et `{7, 1}`.



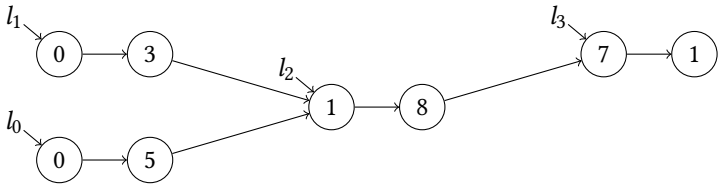
Avec la concaténation `concat(l0, l2)`, on connecte la dernière cellule de `l0` à la première cellule de `l2`. La liste `l0` contient maintenant la séquence `0, 5, 1, 8`.



Si on enchaîne avec la concaténation `concat(l1, l2)`, de manière similaire on lie les deux listes.



Si enfin on appelle `concat(l1, l3)`, la liste `l1` grandit encore pour contenir la séquence `0, 3, 1, 8, 7, 1`.



Cependant la liste `l2` a été modifiée également, même si elle n'était pas mentionnée dans l'opération : elle contient maintenant la séquence `1, 8, 7, 1`. En effet, cette liste était physiquement intégrée à `l1`, et a donc subi des effets des modifications appliquées à `l1`. De même, la liste `l0` contenait la liste `l2` et a également été affectée : elle contient maintenant la séquence `0, 5, 1, 8, 7, 1` que l'on n'aurait peut-être pas souhaité voir.

Ces phénomènes rendent les programmes manipulant des listes chaînées modifiables difficiles à mettre au point et à analyser.

Tri fusion en place, en java. On veut un tri fusion ne créant pas de nouvelles cellules, mais modifiant les pointeurs `tail` de la liste passée en paramètre. Dans un tel contexte, on peut déjà remplacer la paire de fonctions `take`/`chop` par une unique fonction `split` effectuant une coupure dans la liste donnée : la liste `l` après coupure n'est plus que la première partie de la liste d'origine, et on demande à `split` de renvoyer un pointeur vers la deuxième partie. Note : après action de `split`, la liste `l` est définitivement raccourcie.

```
static List split(int n, List l) {
  for(; n > 1; n--) l = l.tail;
  List res = l.tail;
  l.tail = null;
  return res;
}
```

La fonction `merge` fusionne deux listes `l1` et `l2` en modifiant les pointeurs `tail` des cellules. Encore plus manifestement que la précédente, cette opération est « destructrice », dans le sens où les listes d'origine sont perdues (leurs cellules sont recyclées pour former la liste fusionnée). La fonction `merge` renvoie un pointeur vers la première cellule de la liste fusionnée, qui sera la première cellule de l'une des deux anciennes listes `l1` ou `l2`. La boucle principale de la fonction maintient une cellule `last`, qui est la dernière cellule courante de la liste fusionnée, et donc la prochaine dont on va redéfinir le pointeur `tail`.

```
static List merge(List l1, List l2) {
  if (l1 == null) return l2;
  if (l2 == null) return l1;
  // Initialisation du résultat
  List res;
  if (l1.head <= l2.head) { res = l1; l1 = l1.tail; }
  else                    { res = l2; l2 = l2.tail; }
  // Dernière cellule du résultat partiel...
  List last = res;
  // ... que l'on met à jour tant que l1 et l2 ne sont pas épuisées.
  while (l1 != null && l2 != null) {
    if (l1.head <= l2.head) { last.tail = l1; l1 = l1.tail; }
    else                    { last.tail = l2; l2 = l2.tail; }
    last = last.tail;
  }
  // Lorsque l'une des listes est épuisée on concatène l'autre et on conclut.
  if (l1 == null) { last.tail = l2; }
  else            { last.tail = l1; }
  return res;
}
```

Il ne reste alors plus qu'à combiner ces deux premières fonctions pour réaliser le tri `mergeSort`. Comme avec la fonction `merge`, on renvoie un pointeur vers la première cellule de la liste produite.

```
static List mergeSort(List l) {
  int n = length(l);
  if (n < 2) return l;
  List l2 = split((n+1)/2, l);
  return merge(mergeSort(l), mergeSort(l2));
}
```

Petit quizz : après la définition et le tri suivant, vers quelle séquence pointe la variable `l` *?*

```
List l = new List(3, new List(2, new List(5, new List(1, new List(4, null)))));
List r = mergeSort(l);
```

9 Chercher ses clés

9.1 Problème : le mot le plus fréquent

Considérons un texte français d'une taille respectable. Par exemple, *Notre-Dame de Paris* de Victor Hugo. Question : quels en sont les dix mots les plus fréquents ?

Pour répondre simplement à cette question, on peut lire l'intégralité du texte et tenir à jour un lexique, dans lequel on note les mots rencontrés et le nombre d'apparitions de chacun. À la fin, ne reste plus qu'à parcourir ce lexique pour prélever les mots ayant le plus grand nombre d'occurrences. Cependant, pour que ceci soit viable nous avons besoin d'une structure de données :

- qui permet de stocker des mots, et d'associer une information à chaque mot stocké,
- dans laquelle on peut efficacement :
  - tester la présence ou l'absence d'un mot,
  - ajouter un nouveau mot,
  - obtenir ou mettre à jour l'information associée à un mot.

Une telle structure associant des *valeurs* (ici : les nombres d'occurrences) à des *clés* (ici : les mots) est appelée un *tableau associatif*. Nous allons voir l'une des manières efficaces de réaliser un tel tableau associatif, basée sur une structure d'*arbre binaire*, c'est-à-dire une structure de données récursive qui, à chaque étape, se scinde en deux branches.

Une autre structure que vous connaissez déjà pour réaliser cela est la table de hachage.

9.2 Structure : arbre binaire

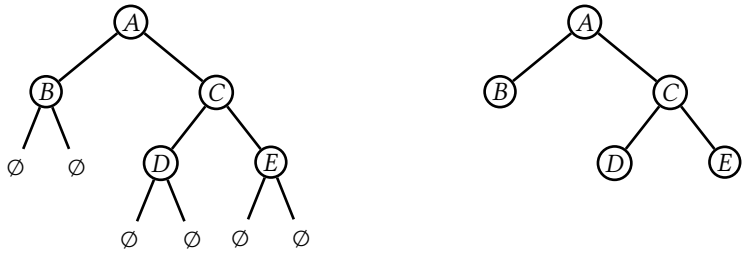
Caractérisation récursive. Un *arbre binaire* est une structure qui est :

- soit vide,
- soit formée d'un *nœud* principal appelé *racine* et de deux sous-structures appelées *sous-arbre gauche* et *sous-arbre droit*.

On a une structure récursive similaire à celle des listes chaînées. Différence : une liste chaînée non vide possède un pointeur vers une unique queue, tandis qu'un arbre binaire possède deux pointeurs vers deux sous-arbres (avec distinction d'un côté gauche et d'un côté droit).

Exemple. Voici un exemple d'arbre binaire, où l'arbre vide est représenté par ∅. Dans les dessins cependant, on ne fait en général pas figurer les sous-arbres vides. On utilisera donc couramment le schéma de droite. L'arbre de cet exemple a pour racine le nœud A, pour sous-arbre gauche l'arbre contenant uniquement le nœud B, et pour sous-arbre droit l'arbre contenant les nœuds C, D et E.

Oui, en info un arbre a une racine en haut et des branches qui s'étendent vers le bas.



Vocabulaire. Une *feuille* est un nœud dont les deux sous-arbres sont vides. Un *nœud interne* est un nœud qui n'est pas une feuille. Dans l'arbre ci-dessus, B, D et E sont des feuilles, et A et C sont des nœuds internes.

La *taille* d'un arbre binaire est le nombre de nœuds qu'il contient. La *profondeur* d'un nœud est le nombre de liens à suivre pour atteindre ce nœud en partant de la racine de l'arbre. La *hauteur* d'un arbre binaire non vide est la profondeur maximale de ses feuilles, plus un. La *hauteur* de l'arbre vide est zéro. Ainsi, l'arbre ci-dessus a une taille de 5. Le nœud A a la prondeur 0, le nœud B la prondeur 1 et le nœud D la prondeur 2. La hauteur de l'arbre est 3.

Dans un arbre binaire, le *fils gauche* (resp. *fils droit*) d'un nœud est la racine du sous-arbre gauche (resp. sous-arbre droit), si ce sous-arbre n'est pas vide. Chaque nœud en dehors de la racine est le fils (gauche ou droit) d'un unique nœud appelé son *parent*. Dans l'arbre ci-dessus, le nœud A a pour fils gauche le nœud B, et pour fils droit le nœud C. Le parent de D est C, et le parent de C est A.

Un arbre peut être vu comme un graphe, dont les sommets sont les nœuds de l'arbre et dont les arêtes sont les liens entre parents et fils. On pourra donc réemployer ici certaines notions vues sur les graphes. *Attention toutefois :* pour un arbre binaire, contrairement au cas des graphes habituels, le placement des nœuds est significatif : le sous-arbre gauche d'un nœud ne doit pas être confondu avec son sous-arbre droit. En particulier, les deux arbres binaires ci-dessous ne sont pas égaux : le nœud B est dans l'un des cas le fils gauche de A, et dans l'autre cas le fils droit.

Dans chacun de ces deux cas le nœud A a bien deux sous-arbres, mais l'un ou l'autre des deux sous-arbres est vide, et le nœud A n'a donc qu'un seul fils.

Dimensions. Voici quelques propriétés liant la taille et la hauteur d'un arbre binaire.

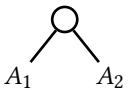
1. *Un arbre binaire a au plus 2^d nœuds à profondeur d.*
   Démonstration par récurrence sur d :
   - Cas $d = 0$: dans un arbre non vide, seule la racine a une profondeur de 0.
   - Cas héréditaire : on suppose que tout arbre binaire a au plus 2^d nœuds à profondeur d et on démontre que tout arbre binaire a au plus 2^{d+1} nœuds à profondeur $d+1$. Soit un arbre A. S'il est vide, il n'a aucun nœud à profondeur $d+1$. S'il est non vide, il possède deux sous-arbres A_1 et A_2. Les nœuds à profondeur $d+1$ de A sont à profondeur d dans A_1 ou dans A_2. Par hypothèse de récurrence il y en a au maximum 2^d dans A_1 et 2^d dans A_2. D'où en additionnant $2^d + 2^d = 2 \times 2^d = 2^{d+1}$ nœuds à profondeur $d+1$ dans A au total.
2. *Un arbre binaire de hauteur h a au plus $2^h - 1$ nœuds.*
   Démonstration : chaque nœud d'un arbre binaire de hauteur h a une profondeur $d < h$. Or il y a au plus 2^d nœuds à profondeur d. Le nombre total de nœuds est donc au plus $\sum_{0 \leq d < h} 2^d = 2^h - 1$.
3. *Un arbre binaire de hauteur h possède au moins h nœuds.*
   Démonstration : il faut au moins un nœud à chacune des profondeurs 0, 1, 2, jusqu'à $h-1$, soit au moins h nœuds au total.
4. *Un arbre binaire non vide avec n sommets a une hauteur h vérifiant $\lfloor log_2(n) \rfloor < h \leq n$.*
   Cette propriété est un corollaire des deux précédentes.

Bilan : lorsque les différents niveaux de profondeur d'un arbre sont « bien remplis », la hauteur est logarithmique en le nombre total de nœuds. Ce point sera la clé de l'efficacité des structures de données bâties sur des arbres [7].

9.3 Raisonnement par récurrence structurelle.

Plutôt que de se ramener à de la récurrence sur les entiers, on peut également tirer parti de la nature récursive des arbres binaires pour raisonner directement par récurrence sur la structure des arbres. Pour montrer qu'une certaine propriété P est valide pour tous les arbres binaires il suffit de vérifier que :

- P est valide pour l'arbre vide, et que
- dès que A_1 et A_2 sont deux arbres binaires pour lesquels P est valide, P reste valide pour l'arbre composé



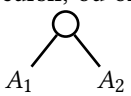
Exemple. Pour tout arbre A la hauteur h_A et la taille n_A vérifient $n_A < 2^{h_A}$.
On raisonne par récurrence structurelle. Comme dans un raisonnement par récurrence usuel, on a deux cas à traiter.

- Cas de base, pour l'arbre vide : il faut montrer que la hauteur $h_\emptyset$ et la taille $n_\emptyset$ de l'arbre vide vérifient $n_\emptyset < 2^{h_\emptyset}$. Or, par définition on a

  $$\begin{aligned} h_\emptyset &= 0 \\ n_\emptyset &= 0 \end{aligned}$$

  On a donc bien $n_\emptyset = 0 < 1 = 2^0 = 2^{h_\emptyset}$.

7. Plus d'information dans la section d'approfondissement sur les arbres équilibrés.

— Cas récursif, où on veut montrer que la propriété est valide pour un arbre A de la forme  en supposant qu'elle est bien valide pour les deux sous-arbres A_1 et A_2. Autrement dit, en notant h_1 et n_1 (resp. h_2 et n_2) la hauteur et la taille de A_1 (resp. A_2), on suppose que

$$\begin{aligned} n_1 &< 2^{h_1} \\ n_2 &< 2^{h_2} \end{aligned}$$

et on veut démontrer que

$$n_A < 2^{h_A}$$

Les deux suppositions sont appelées *hypothèses de récurrence.*
Détail du raisonnement :

$$\begin{aligned} n_A &= 1 + n_1 + n_2 && \text{par déf. de la taille} \\ &\leq 2^{h_1} + n_2 && \text{par hyp. de récurrence 1} \\ &< 2^{h_1} + 2^{h_2} && \text{par hyp. de récurrence 2} \\ &\leq 2^{\max(h_1,h_2)} + 2^{\max(h_1,h_2)} \\ &= 2^{1+\max(h_1,h_2)} \\ &= 2^{h_A} && \text{par déf. de la hauteur} \end{aligned}$$

9.4 Représentation des arbres binaires en caml

On peut définir un type de données caml pour représenter des arbres binaires en suivant la caractérisation récursive des arbres. Définissons donc un arbre comme étant :

— soit l'arbre vide, représenté par la constante `E`,
— soit un nœud, représenté par l'application `N(`l`, `r`)` du constructeur `N` à deux sous-arbres gauche et droit l et r.

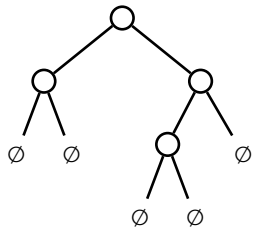
Déclaration d'un tel type en caml :

```
type tree =
  | E
  | N of tree * tree
```

Alors la définition

```
let t = N(N(E, E), N(N(E, E), E))
```

associe la variable `t` à l'arbre



On peut alors définir des fonctions manipulant les arbres binaires grâce aux mécanismes de filtrage. Par exemple pour calculer la taille d'un arbre, c'est-à-dire compter son nombre de nœuds, on prévoit un cas indiquant que l'arbre vide ne contient aucun nœud, et un cas indiquant que la taille d'un arbre non vide est obtenue en ajoutant un à la somme des tailles de ses deux sous-arbres.

Rappel : la ligne
`let rec size = function`
est ici équivalente à
`let rec size t = match t with`

```
let rec size = function
  | E       -> 0
  | N(l, r) -> 1 + size l + size r
```

Cette définition caml réalise les équations suivantes :

$$\begin{cases} \texttt{size}(\emptyset) = 0 \\ \texttt{size}\left(\begin{array}{c} \bigcirc \\ A_1 \quad A_2 \end{array}\right) = 1 + \texttt{size}(A_1) + \texttt{size}(A_2) \end{cases}$$

On peut calculer la hauteur d'une manière similaire.

```
let rec height = function
  | E       -> 0
  | N(l, r) -> 1 + max (height l) (height r)
```

Arbres binaire annotés. On peut associer des informations à chaque sommet d'un arbre pour représenter des objets intrinsèquement arborescents ou organiser des structures de données. Selon la structure représentée on peut utiliser pour cela des stratégies variées :

— mettre des données dans chaque nœud, ou

— ne mettre des données que dans les feuilles, ou

— mettre des informations de natures différentes dans les feuilles et dans les nœuds internes.

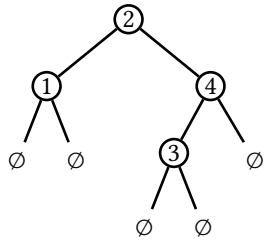
Considérons ici la version la plus simple : chaque nœud contient un nombre entier. On peut enrichir notre type caml pour ajouter ce troisième paramètre au constructeur `N` :

```
type tree =
  | E
  | N of tree * int * tree
```

La nouvelle définition

```
let t = N(N(E, 1, E), 2, N(N(E, 3, E), 4, E))
```

associe la variable `t` à l'arbre



où on a écrit dans chaque nœud la donnée associée.

Parcours d'arbre. La présence d'un élément x dans un arbre A peut être ramenée à deux équations simples :

— l'arbre vide ne contient aucun élément

$$x \notin \emptyset$$

— un élément x présent dans un arbre non vide A peut être à la racine, dans le sous-arbre gauche ou dans le sous-arbre droit

$$x \in \overset{v}{\underset{A_1 \quad A_2}{\diagup\diagdown}} \iff x = v \ \vee \ x \in A_1 \ \vee \ x \in A_2$$

Toujours en utilisant le filtrage, on peut directement traduire ces équations en du code caml :

```
let rec mem x t = match t with
  | E          -> false
  | N(l, v, r) -> x = v || mem x l || mem x r
```

Dans le cas d'un arbre vide, cette fonction renvoie `false`. Dans le cas d'un arbre non vide elle teste d'abord si l'élément cherché est à la racine, et dans le cas contraire elle poursuit la recherche dans le sous-arbre gauche, puis dans le sous-arbre droit.

Note : l'opérateur `||` est *paresseux*. Si sa condition de gauche est vraie, il renvoie `true` sans évaluer la condition de droite. La fonction précédente est donc équivalente à :

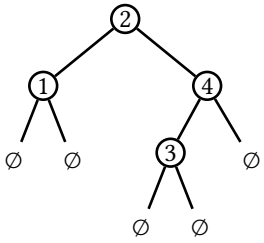
```
let rec mem x t = match t with
  | E          -> false
  | N(l, v, r) -> if x = v then true
                  else if mem x l then true
                  else mem x r
```

Autrement dit, cette fonction `mem` réalise un parcours en profondeur de l'arbre, et explore les branches de gauche à droite.

9.5 Représentation des arbres binaires en java

Comme on l'avait fait pour les listes, on peut représenter des arbres binaires à l'aide d'une unique classe représentant un nœud, et utiliser le pointeur nul pour l'arbre vide.

```
class N {
  final N l, r;
  final int v;
  N(N l, int v, N r) { this.l = l; this.v = v; this.r = r; }
  ...
```



On utilise alors la définition

```
N t = new N(new N(new N(null, 1, null), 2, new N(new N(null, 3, null), 4, null));
```

pour définir l'arbre exemple rappelé dans la marge.

On peut ensuite ajouter à la classe des définitions de fonctions récursives réalisant les fonctions `size`, `height` et `mem` déjà vues.

```
  ...
  static int size(N t) {
    if (t == null) { return 0; }
    else           { return 1 + size(t.l) + size(t.r); }
  }
  static int height(N t) {
    if (t == null) { return 0; }
    else           { return 1 + Math.max(height(t.l), height(t.r)); }
  }
  static boolean mem(int x, N t) {
    if (t == null) { return false; }
    else           { return x == t.v || mem(x, t.l) || mem(x, t.r); }
  }
}
```

9.6 Structure : arbres binaires de recherche

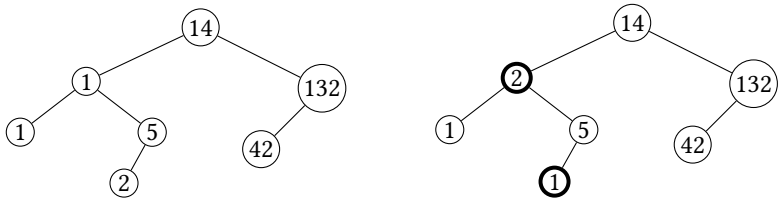
Dans un *arbre binaire de recherche*, on stocke une donnée dans chaque nœud d'une manière qui permet de localiser efficacement les données. Il s'agit de créer une structure d'arbre dans laquelle la recherche d'un élément peut être similaire à la recherche dichotomique :

- considérer un élément médian m,
- s'arrêter s'il s'agit de l'élément x cherché,
- sinon, chercher « à gauche » ou « à droite » selon que x est plus petit ou plus grand que m.

Avec une structure d'arbre binaire, on prend comme élément médian l'élément stocké dans la racine, et les directions « à gauche » et « à droite » consistent à poursuivre la recherche dans le sous-arbre gauche ou dans le sous-arbre droit.

L'invariant important des tableaux triés permettant la recherche dichotomique était : tous les éléments de la partie gauche sont inférieurs à l'élément médian, et tous les éléments de la partie droite sont supérieurs à l'élément médian. Ainsi par exemple, si on cherchait un élément plus petit que l'élément médian il était garanti qu'il ne se trouvait pas dans la partie droite, qui pouvait être ignorée dans la suite de la recherche. On peut transposer cette propriété dans le cadre des arbres.

Structure. Un *arbre binaire de recherche* (ABR) est un arbre binaire annoté dans lequel, pour tout nœud A portant un élément m, tous les éléments présents dans le sous-arbre gauche de A sont inférieurs à m et tous les éléments présents dans le sous-arbre droit de A sont supérieurs à m. L'arbre ci-dessous à gauche est un ABR. Celui de droite ne l'est pas : on y trouve une occurrence de 1 dans le sous-arbre droit d'un nœud portant la valeur 2.



Recherche et ajout. Dans de tels arbres, on peut transposer l'algorithme de recherche dichotomique. La condition d'arrêt selon laquelle on abandonne la recherche et on déclare que l'élément n'appartient pas à la collection lorsque la zone de recherche est vide correspond ici à tester la vacuité de l'arbre.

```
let rec mem x t = match t with
  | E                     -> false
  | N(_, m, _) when x = m  -> true
  | N(l, m, _) when x < m  -> mem x l
  | N(_, m, r) (* x > m *) -> mem x r
```

```
static boolean mem(int x, N t) {
  if (t == null)     { return false; }
  else if (x == t.v) { return true; }
  else if (x  < t.v) { return mem(x, t.l); }
  else /* x > t.v */ { return mem(x, t.r); }
}
```

Cet algorithme répond dans le pire cas en un temps proportionnel à la hauteur de l'arbre (logarithmique si l'arbre a une bonne forme, linéaire au pire).

Pour ajouter un élément dans un arbre binaire de recherche tout en préservant la propriété des ABR, il suffit de commencer par utiliser l'algorithme de recherche, puis :

— si l'élément a été trouvé, ne rien faire,

— sinon, on a atteint un sous-arbre vide, qu'on remplace par une feuille contenant l'élément à ajouter.

Dans cet algorithme, on a choisi de ne pas ajouter une deuxième copie d'un élément déjà présent.

```
let rec add x t = match t with
  | E                     -> N(E, x, E)
  | N(_, m, _) when x = m  -> t
  | N(l, m, r) when x < m  -> N(add x l, m, r)
  | N(l, m, r) (* x > m *) -> N(l, m, add x r)
```

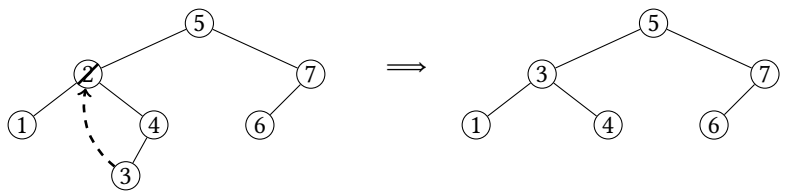
```
static N add(int x, N t) {
  if (t == null)     { return new N(null, x, null); }
  else if (x == t.v) { return t; }
  else if (x  < t.v) { return new N(add(x, t.l), t.v, t.r); }
  else /* x > t.v */ { return new N(t.l, t.v, add(x, t.r)); }
}
```

Approfondissement : retrait d'un élément. Pour retirer un élément d'un arbre binaire de recherche tout en préservant la propriété des ABR il faut commencer par aller chercher cet élément, puis on a différents cas de figure :

— Si l'élément à retirer est sur une feuille, il suffit de retirer la feuille.

— Si l'élément à retirer est sur un nœud ayant un seul fils (gauche ou droit), il suffit de retirer ce nœud et de le remplacer par son unique fils.

— Si l'élément à retirer est sur un nœud ayant deux fils, on le remplace par un nœud portant la plus petite valeur du sous-arbre droit (cette dernière étant alors retirée du sous-arbre droit).



En caml on introduit déjà une fonction `min` qui cherche le plus petit élément d'un arbre binaire de recherche (cet élément est « le plus à gauche »), et une fonction `remove_min` qui retire le plus petit élément d'un arbre binaire de recherche. L'une et l'autre ont comme précondition que l'arbre doit être non vide. On peut ensuite en déduire une fonction `remove` qui applique notre stratégie.

```
let rec min = function
  | E          -> assert false
  | N(E, v, _) -> v
  | N(l, _, _) -> min l

let rec remove_min = function
  | E          -> assert false
  | N(E, _, r) -> r
  | N(l, v, r) -> N(remove_min l, v, r)

let rec remove x t = match t with
  | E                     -> E
  | N(l, v, r) when x < v -> N(remove x l, v, r)
  | N(l, v, r) when x > v -> N(l, v, remove x r)
  | N(E, _, r) (* x=v *)  -> r
  | N(l, _, E) (* x=v *)  -> l
  | N(l, _, r) (* x=v *)  -> N(l, min r, remove_min r)
```

```
static int min(N t) {
  if (t == null)        { throw new IllegalArgumentException(); }
  else if (t.l == null) { return t.v; }
  else                  { return min(t.l); }
}
static N removeMin(N t) {
  if (t == null)        { throw new IllegalArgumentException(); }
  else if (t.l == null) { return t.r; }
  else                  { return new N(removeMin(t.l), t.v, t.r); }
}
static N remove(int x, N t) {
  if (t == null)    { return null; }
  else if (x < t.v) { return new N(remove(x, t.l), t.v, t.r); }
  else if (x > t.v) { return new N(t.l, t.v, remove(x, t.r)); }
  else /* x == v */
       if      (t.l == null) { return t.r; }
       else if (t.r == null) { return t.l; }
       else                  { return new N(t.l, min(t.r), removeMin(t.r)); }
}
```

9.7 Approfondissement : table associative

La structure d'arbre binaire de recherche peut directement être adaptée pour représenter une table associative. Il suffit de :

— faire porter deux éléments à chaque nœud : une clé et la valeur associée,

— ordonner les nœuds en fonction de leurs clés.

Si nos clés sont des chaînes de caractères, comme dans notre problème de recensement des mots d'un texte, on peut naturellement comparer deux clés à l'aide de l'ordre lexicographique (celui du dictionnaire). C'est le critère qui est utilisé dans les bibliothèque standard de caml et java (et de nombreux langages).

En caml. On crée un nouveau type d'arbre, où chaque nœud porte une chaîne de caractères (une clé) et un entier (une valeur).

```
type abr =
  | E
  | N of abr * string * int * abr
```

Les fonctions déjà vues s'adaptent naturellement. La recherche par exemple se fait à l'aide d'une clé, et répond à la question « cette clé est-elle présente dans la table ? ».

```
let rec mem k = function
  | E              -> false
  | N(l, k', _, r) -> if k < k' then mem k l
                      else if k > k' then mem k r
                      else true
```

Note : la fonction `mem` n'a jamais besoin de consulter les valeurs. On pourrait de même écrire une fonction `remove` retirant une clé d'un arbre (et la valeur associée), et une fonction `add` ajoutant une clé et une valeur un arbre.

En plus des fonctions déjà connues, on peut ajouter des fonctions manipulant les valeurs. Par exemple une fonction `find` renvoyant la valeur associé à une clé dans une table, avec la même structure que `mem`

Pourrait-on aussi écrire une fonction `mem_v`, qui indique si une valeur `v` est présente dans la table ? Avec quelle complexité ?

```
let rec find k = function
  | E              -> raise Not_found
  | N(l, k', v, r) -> if k < k' then find k l
                      else if k > k' then find k r
                      else v
```

ou encore une fonction `update` renvoyant une table mise à jour avec une nouvelle association clé/valeur (similaire à `add` mais modifiant une éventuelle valeur déjà présente).

```
let rec update k v = function
  | E               -> N (E, k, v, E)
  | N(l, k', v', r) -> if k < k' then N(update k v l, k', v', r)
                       else if k > k' then N(l, k', v', update k v r)
                       else N(l, k, v, r) (* on écrase la valeur précédente *)
```

9.8 Approfondissement : représentation alternative en java

Jusque-là, on a utilisé en java une classe unique pour les nœuds, et le pointeur nul pour l'arbre vide. En programmation objet, on peut alternativement utiliser plusieurs classes pour les différentes formes possibles d'une structure. On définit d'abord une interface commune, fixant le type général des arbres et déclarant les méthodes qu'on voudra pouvoir appliquer.

```
interface Tree {
  int size();
  int height();
  boolean mem(int x);
}
```

Puis on fournit une classe concrète pour chaque forme que peut prendre un arbre, avec définition de chacune des méthodes pour le cas correspondant. La classe concrète `E` va représenter l'arbre vide (on ne manipule donc plus de pointeurs nuls). Cette classe n'a pas besoin d'attributs et peut se contenter du constructeur par défaut. Elle contient en revanche des définitions pour chacune des méthodes, correspondant au cas `t == null` dans la version précédente.

```
class E implements Tree {
  public int size() { return 0; }
  public int height() { return 0; }
  public boolean mem(int x) { return false; }
}
```

La classe concrète `N` va représenter un nœud. On lui donne trois attributs représentant la valeur portée par le nœud et ses deux sous-arbres gauche et droit, et un constructeur adapté. Les définitions des différentes méthodes correspondent cette fois au cas récursif, et les appels récursifs sont remplacés par des appels des méthodes des sous-arbres.

```
class N implements Tree {
  public final Tree l, r;
  public final int v;
  public N(Tree l, int v, Tree r) { this.l = l; this.v = v; this.r = r; }

  public int size() { return 1 + l.size() + r.size(); }
  public int height() { return 1 + Math.max(l.height(), r.height()); }
  public boolean mem(int x) {
    if (x == v)      { return true; }
    else if (x < v)  { return l.mem(x); }
    else /* x > v */ { return r.mem(x); }
  }
}
```

Dans cette version, la définition de chaque fonction sur les arbres est donc répartie entre les différentes classes concrètes (vu dans l'autre sens : chaque classe est responsable, dans chaque fonction, de la partie qui la concerne). Pour ajouter une nouvelle fonction on va donc étendre l'interface, puis ajouter une définition à chaque classe. Pour ajouter une fonction `add` par exemple :

```
interface Tree {
  ...
  Tree add(int x);
}

class E extends Tree {
  ...
  public Tree add(int x) { return new N(new E(), x, new E()); }
}

class N extends Tree {
  ...
  public Tree add(int x) {
    if (x == v)      { return this; }
    else if (x < v)  { return new N(l.add(x), v, r); }
    else /* x > v */ { return new N(l, v, r.add(x)); }
  }
}
```

Avantage en revanche : on va gagner plus de souplesse pour définir des structures arborescentes plus riches, dans lesquelles les nœuds n'ont pas tous la même forme [8].

9.9 Approfondissement : arbres binaires de recherche équilibrés

Arbres binaires équilibrés. Un arbre binaire est *équilibré* si pour chaque nœud N de cet arbre les hauteurs des sous-arbres gauche et droit ne diffèrent pas de plus de 1.

Un arbre binaire équilibré de hauteur h a au moins $F_{h+2} - 1$ sommets, où $(F_k)_{k\in\mathbb{N}}$ est la suite de Fibonacci.

Rappel de la définition de la suite de Fibonacci.

$$\begin{cases} F_0 &= 0 \\ F_1 &= 1 \\ F_{n+2} &= F_{n+1} + F_n \end{cases}$$

Démonstration par récurrence forte sur h. On considère un arbre binaire équilibré A de hauteur h, et on suppose que tout arbre binaire équilibré de hauteur $k < h$ possède au moins $F_{k+2} - 1$ nœuds. Raisonnement par cas sur la forme de A.

- Si A est vide, on a $h = 0$ et 0 sommet, et on a bien $F_2 - 1 = 1 - 1 = 0$.
- Si A est formé par une racine et par deux sous-arbres A_1 et A_2, alors l'un des deux sous-arbres a la hauteur $h - 1$. Supposons qu'il s'agit de A_1 (l'autre cas sera symétrique). Comme $h - 1 < h$, par hypothèse de récurrence A_1 possède au moins $F_{(h-1)+2} - 1 = F_{h+1} - 1$ nœuds. Comme A est équilibré, la hauteur du deuxième sous-arbre A_2 diffère d'au plus 1 avec la hauteur $h-1$ de A_1. Donc la hauteur de A_2 est au moins $h-2$. Comme $h - 2 < h$, par hypothèse de récurrence A_2 a au moins $F_{(h-2)+2} - 1 = F_h - 1$ nœuds. Le nombre de nœuds total est donc au moins $1+(F_{h+1}-1)+(F_h-1) = F_{h+1}+F_h-1 = F_{h+2}-1$.

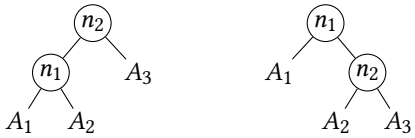
La suite de Fibonacci ayant une croissance exponentielle, on en déduit qu'un *arbre binaire équilibré a une hauteur logarithmique* en son nombre de nœuds.

AVL comme Adelson-Velski et Landis, les inventeurs.

Arbres AVL. Les arbres AVL sont des arbres binaires de recherche qui sont toujours équilibrés. Les opérations de recherche, d'insertion et de suppression d'un élément utilisent les mêmes algorithmes que ceux déjà vus pour les arbres de recherche, à une différence près : à chaque fois qu'un nœud est ajouté à ou retiré d'un sous-arbre, on vérifie l'équilibrage de ce sous-arbre. Si le nœud ajouté ou retiré amène un déséquilibre, alors on réarrange le sous-arbre concerné pour rétablir l'équilibre.

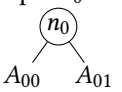
Le rééquilibrage est basé sur une opération de *rotation*, qui transforme l'arbre ci-dessous à gauche en celui à droite (et inversement).

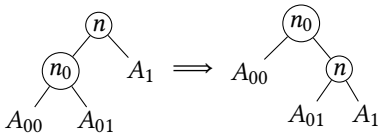
8. Vous en verrez des exemples : en TP, dans un chapitre ultérieur, mais aussi en PIL.

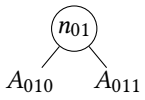


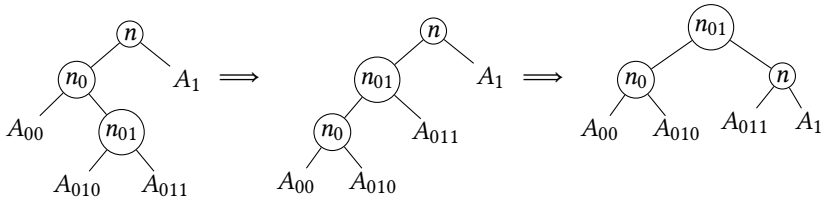
Une telle rotation, appliquée à un arbre binaire de recherche, préserve sa qualité d'arbre de recherche. En revanche, selon les profondeurs respectives des sous-arbres A_1, A_2 et A_3 une telle rotation peut modifier l'équilibre de l'arbre (en l'occurrence, on utilisera des rotations pour corriger un équilibre menacé).

Cas d'étude. On part d'un arbre équilibré et on ajoute un élément x à son sous-arbre gauche. On obtient un arbre  dans lequel on peut avoir un déséquilibre du côté gauche si l'ajout d'un élément a fait augmenter la hauteur du sous-arbre gauche. En notant h_0 et h_1 les hauteurs des deux sous-arbres A_0 et A_1 le risque est donc que $h_0 = h_1 + 2$.

Dans cette situation, au moins un des deux sous-arbres de A_0 =  a une hauteur de $h_1 + 1$. S'il s'agit du sous-arbre de gauche A_{00}, une rotation simple suffit à rétablir l'équilibre.



S'il s'agit du sous-arbre de droite A_{01} =  , on enchaîne deux rotations.



Réalisation en caml. On enrichit le type des arbres pour étiqueter chaque nœud avec sa hauteur.

```
type avl =
  | E
  | N of avl * int * avl * int
```

On obtient la hauteur d'un arbre non vide en lisant le champ correspondant.

```
let height = function
  | E            -> 0
  | N(_, _, _, h) -> h
```

Pour alléger la création des nœuds on utilise la fonction suivante, qui renseigne elle-même la hauteur.

```
let node l n r =
  N(l, n, r, 1 + max (height l) (height r))
```

L'ajout d'un élément se fait comme précédemment, à ceci près que chaque nouveau nœud est créé par une fonction auxiliaire qui fait le rééquilibrage si nécessaire. On utilise deux fonctions de rééquilibrage aux fonctionnements symétriques, pour les ajouts à gauche ou à droite.

```
let rec add x t = match t with
  | E                      -> node E x E
  | N(_, n, _, _) when x=n  -> t
  | N(l, n, r, _) when x<n  -> rebalance_left (add x l) n r
  | N(l, n, r, _) (* x>n *) -> rebalance_right l n (add x r)
```

La fonction chargée de corriger les déséquilibres à gauche est une traduction directe des rotations décrites plus haut.

```
let rebalance_left a0 n a1 =
  if height a0 > height a1 + 1 then
    match a0 with
    | N(a00, n0, a01, _) when height a00 >= height a01 ->
       node a00 n0 (node a01 n a1)
    | N(a00, n0, N(a010, n01, a011, _), _) ->
       node (node a00 n0 a010) n01 (node a011 n a1)
    | _ -> assert false
  else
    node a0 n a1
```

La fonction `rebalance_right` s'écrirait de manière symétrique, et ces deux fonctions pourraient de même être utilisées pour que la fonction `remove` préserve l'équilibre d'un AVL.

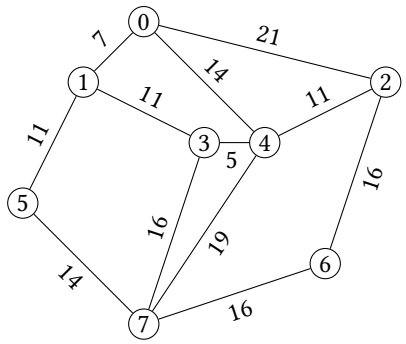
10 Trouver un raccourci

10.1 Problème : l'itinéraire le plus court

Imaginons une carte routière avec ses routes, ses villes et ses carrefours, et supposons que l'on cherche à déterminer un itinéraire le plus court possible entre une ville de départ et une ville d'arrivée.

On peut imaginer résumer notre carte par un graphe, dont les sommets seraient les villes et les arêtes les routes reliant deux villes. Nous avons déjà vu que les parcours en profondeur et en largeur permettaient de trouver des chemins depuis un sommet source vers les autres sommets accessibles d'un graphe, et également que le parcours en largeur trouvait systématiquement des chemins utilisant un nombre minimal d'arêtes. Cependant, l'itinéraire empruntant le nombre minimal de routes n'est pas nécessairement celui qui nous intéresse ici : toutes les routes n'ont pas la même longueur. Autrement dit : certaines arêtes de notre graphe représentent un chemin plus long que d'autres, et deux arêtes « courtes » peuvent être préférables à une arête « longue ».

On enrichit notre modèle de graphe pour ajouter à chaque arête un « coût », (appelé traditionnellement *poids*), sous la forme d'un nombre qui sera d'autant plus faible que l'arête sera intéressante. Ici, ce coût représentera directement la longueur du chemin représenté par l'arête. On cherche alors des *chemins les plus courts*, c'est-à-dire des chemins pour lesquels la somme des poids des arêtes est minimale.



Ces *graphes pondérés* ont une interface très similaire à l'interface déjà utilisée pour les graphes ordinaires : on y ajoute principalement une fonction `int weight(int s, int t)` donnant le poids de l'arête allant du sommet `s` au sommet `t` (en supposant qu'une telle arête existe bien). Pour le code java de ce chapitre, on utilisera ainsi l'interface suivante.

```
interface WGraph {
  int size();                    // nombre de sommets
  Iterable<Integer> succ(int s); // voisins de s
  int weight(int s, int t);      // poids de l'arête s --> t
}
```

On peut en imaginer une réalisation inspirée des listes d'adjacence, où la listes des voisins d'un sommet `s` est remplacée par une table de hachage associant chaque voisin `t` de `s` à sa distance.

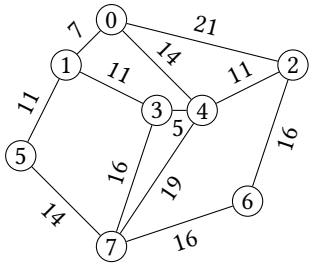
```
class WGraphAdj implements WGraph {
  private final int size;
  private ArrayList<HashMap<Integer, Integer>> adj;

  WGraphAdj(int size) {
    this.size = size;
    this.adj = new ArrayList<>(size);
    for (int s=0; s<size; s++) { adj.add(new HashMap<>()); }
  }
  void addEdge(int s, int t, int d) { adj.get(s).put(t, d); }

  public int size() { return this.size; }
  public Iterable<Integer> succ(int s) { return adj.get(s).keySet(); }
  public int weight(int s, int t) { return adj.get(s).get(t); }
}
```

10.2 Algorithme de Dijkstra

L'algorithme de Diskstra est une adaptation du parcours en largeur qui permet de trouver les chemins les plus courts dans un graphe pondéré [9]. Dans l'un et l'autre, on souhaite explorer les sommets par ordre croissant de « distance » à la source. Les deux diffèrent par la notion de distance considérée, et la structure utilisée pour stocker les sommets en attente.

— Dans le parcours en largeur, la distance est évaluée en nombre d'arêtes. On utilise une file FIFO, et on explore les sommets dans l'ordre de leur découverte.

— Dans l'algorithme de Dijkstra, la distance est évaluée par le poids des chemins. L'ordre de découverte des sommets ne correspond plus nécessairement aux distances croissantes à la source. À chaque nouvelle étape on choisit, parmi les sommets en attente, celui qui est à la plus courte distance de la source.



Voici un exemple d'exécution de l'algorithme. On explore le graphe donné dans la marge en prenant comme sommet s d'origine le sommet numéro 5. Le tableau montre à chaque ligne :

— le sommet `t` exploré,

— les sommets en attente, classés par ordre de distance croissante,

— un tableau `dist` tel que `dist[t]` est le poids du meilleur chemin connu vers `t`,

— un tableau `pred` tel que `pred[t]` précède `t` dans le meilleur chemin connu vers `t`,

— un tableau `visited` mémorisant les sommets déjà explorés.

		dist								pred								visited							
t	En attente	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
–	**5(0)**	–	–	–	–	–	**0**	–	–	–	–	–	–	–	**5**	–	–	–	–	–	–	–	–	–	–
5	**1(11)**, **7(14)**	–	**11**	–	–	–	0	–	**14**	–	**5**	–	–	–	5	–	**5**	–	–	–	–	–	✓	–	–
1	7(14), **0(18)**, **3(22)**	**18**	11	–	**22**	–	0	–	14	**1**	5	–	**1**	–	5	–	5	–	✓	–	–	–	✓	–	–
7	0(18), 3(22), **6(30)**, **4(33)**	18	11	–	22	**33**	0	**30**	14	1	5	–	1	**7**	5	**7**	5	–	✓	–	–	–	✓	–	✓
0	3(22), 6(30), 4(**32**), **2(39)**	18	11	**39**	22	**32**	0	30	14	1	5	**0**	1	**0**	5	7	5	✓	✓	–	–	–	✓	–	✓
3	4(**27**), 6(30), 2(39)	18	11	39	22	**27**	0	30	14	1	5	0	1	**3**	5	7	5	✓	✓	–	✓	–	✓	–	✓
4	6(30), 2(**38**)	18	11	**38**	22	27	0	30	14	1	5	**4**	1	3	5	7	5	✓	✓	–	✓	✓	✓	–	✓
6	2(38)	18	11	38	22	27	0	30	14	1	5	4	1	3	5	7	5	✓	✓	–	✓	✓	✓	✓	✓
2		18	11	38	22	27	0	30	14	1	5	4	1	3	5	7	5	✓	✓	✓	✓	✓	✓	✓	✓

À noter dans cet exemple d'exécution, un point qui diffère avec le parcours en largeur déjà connu : il est possible ici, à un moment où l'on connaît déjà des chemins vers un certain sommet `t`, de découvrir un nouveau chemin meilleur que ceux déjà connus. Dans ce cas, on met à jour les tableaux `dist` et `pred` pour ne mémoriser à chaque fois que le meilleur chemin connu. Dans l'exemple, cela apparaît par exemple avec le sommet s_4 : on découvre en premier un chemin de longueur 33 venant de s_7, puis un chemin de longueur 32 venant de s_0, et enfin un chemin de longueur 27 venant de s_3.
À la fin de l'exécution, en consultant `dist` et `pred` on apprend :

— que s_2 est à distance 38, en passant par s_4,

— que s_4 est à distance 27, en passant par s_3,

— que s_3 est à distance 22, en passant par s_1,

— que s_1 est à distance 11, directement depuis s_5.

Ainsi, la séquence 5, 1, 3, 4, 2 est le chemin le plus court de 5 vers 2 (de longueur totale 38).

Complexité ? Présenté ainsi, l'algorithme semble suffisamment similaire au parcours en largeur pour avoir la même complexité, linéaire en les nombres de sommets et d'arêtes du graphe. Cependant, un coût supplémentaire est caché à chaque étape dans la sélection du prochain sommet à explorer. On peut imaginer plusieurs scénarios.

1. Ranger les sommets à traiter par ordre croissant de distance dans un tableau ou une liste. La sélection du prochain sommet est immédiate, mais chaque insertion d'un nouveau sommet est linéaire (car il faut l'insérer à la bonne place).
2. Laisser les sommets à traiter dans le désordre, et les parcourir à la recherche du minimum à chaque nouvelle étape. L'insertion est efficace, mais la sélection est linéaire.
3. Trouver une nouvelle structure de données, dans laquelle l'insertion *et* la sélection seront toutes les deux efficaces !

Évidemment, nous allons regarder ici ce dernier scénario, avec la structure de *file de priorité*.

9. À condition que les poids des arêtes soient bien positifs ou nuls !

File de priorité. On fixe pour la file de priorité une interface `PQueue`, dont les principales méthodes sont : ajouter un nouvel élément à la file, en précisant son niveau de priorité, et retirer de la file l'élément le plus prioritaire. Ici, on représente le niveau de priorité par un entier positif d'autant plus petit que l'élément est prioritaire.

```
interface PQueue {
  boolean isEmpty();         // teste si la file est vide
  void insert(int x, int p); // insère l'élément x avec la priorité p
  int extractMin();          // retire et renvoie l'élément de priorité minimale
}
```

Voici une réalisation en java de l'algorithme de Dijkstra, avec une structure proche du parcours en largeur déjà vu, mais en utilisant une file de priorité à la place de la file FIFO. On utilise également les différents éléments déjà énumérés dans l'exemple d'exécution : `t` est le sommet en cours d'exploration, le tableau `visited` indique les sommets déjà explorés, les tableaux `dist` et `pred` donnent les informations des meilleurs chemins connus à un moment donné (on fixe `dist[u]` = `pred[u]` = -1 si on n'a pas encore découvert de chemin atteignant un sommet `u`). Lorsque l'on explore un sommet `t` on considère chacun de ses voisins. Si un voisin `v` n'avait encore jamais été découvert, *ou si le chemin menant à* `v` *en passant par* `t` *est meilleur que le meilleur chemin vers* `v` *déjà connu*, alors on renseigne les tableaux `dist` et `pred` avec les nouvelles informations et on ajoute `v` à la file de priorité. On donne comme « priorité » à `v` le poids du chemin vers `v` que l'on vient de découvrir. Ainsi, les sommets explorés par ordre de priorité croissante seront bien explorés par ordre de distance à la source.

```
static int[] dijkstra(WGraph g, int s) {
  // initialisation
  int n = g.size();
  boolean[] visited = new boolean[n];
  int[] dist = new int[n]; for (int t=0; t<n; t++) dist[t] = -1; // dist. source
  int[] pred = new int[n]; for (int t=0; t<n; t++) pred[t] = -1; // chemins

  // file de priorité
  PQueue pqueue = new ... ; // insérer ici la classe concrète utilisée

  // insertion de la source
  pqueue.insert(s, 0); dist[s] = 0; pred[s] = s;

  // exploration
  while (!pqueue.isEmpty()) {
    // on considère le sommet non visité à plus courte distance de la source
    int t = pqueue.extractMin();
    if (visited[t]) continue;
    visited[t] = true;
    for (int v: g.succ(t)) {
      // insertion d'un successeur dans la file si vu pour la première fois
      // ou si chemin plus court que le meilleur connu actuellement
      int d = dist[t] + g.weight(t, v);
      if (dist[v] < 0 || d < dist[v]) {
        pqueue.insert(v, d); dist[v] = d; pred[v] = t;
      }
    }
  }
  return pred;
}
```

À noter : notre interface de file de priorité ne propose pas de fonction permettant de « mettre à jour » la priorité d'un élément déjà présent. À la place, on ajoute l'élément à nouveau. Dans cet algorithme, il est donc possible qu'un même sommet `t` apparaisse plusieurs fois dans la file de priorité, avec plusieurs priorités correspondant chacune à la longueur de l'un des chemins de `s` à `t` découverts. Chacun ne sera cependant exploré qu'une fois, correspondant à la meilleure priorité : lorsque l'ordre de la file nous amène à une deuxième occurrence (moins prioritaire) le sommet a déjà été marqué dans le tableau `visited`.

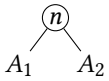
Les files de priorité avec possibilité de mise à jour existent ! Mais elles sont plus complexes que ce que nous allons voir ici.

Bilan. L'efficacité de l'algorithme de Dijkstra va dépendre de la structure utilisée pour la file de priorité. Dans la suite du chapitre, nous allons voir une nouvelle utilisation des arbres binaires permettant de réaliser une file de priorité efficace.

10.3 Structure : tas binaire

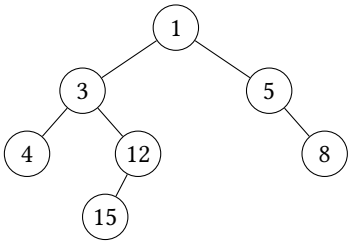
Un *tas binaire* est un arbre binaire dont chaque nœud contient un élément, et où l'élément porté par un nœud est inférieur ou égal aux éléments portés par ses fils. Autrement dit :

— l'arbre vide est un tas,

— un arbre composé de la forme 

est un tas si A_1 et A_2 sont tous deux des tas, et si n est inférieur ou égal aux racines de A_1 et A_2 (et si A_1 ou A_2 est vide, il n'y a rien à vérifier pour ce sous-arbre).

Propriété essentielle d'un tas : le plus petit élément est systématiquement à la racine.



Consulter l'élément minimal d'un tas est facile : c'est l'élément porté par la racine. On veut également des algorithmes efficaces pour retirer cet élément minimal (c'est-à-dire pour reformer un tas avec le reste), et pour ajouter un élément quelconque (à une position qui respecte la propriété d'ordre). On a plusieurs stratégies pour cela.

Première réalisation : *skew heap*. Un tas étant un arbre binaire, on peut reprendre directement l'une des défininitions déjà vues au chapitre précédent. Par exemple en caml.

```
type heap =
  | E
  | N of heap * int * heap
```

On a déjà dit que l'élément minimal d'un tas se trouvait à la racine. On l'obtient donc directement ainsi.

```
let min t = match t with
  | E          -> raise Not_found
  | N(_, x, _) -> x
```

Nous allons voir que l'ajout d'un élément quelconque ou le retrait de la racine peuvent tous deux être ramenés à une unique opération de fusion de deux tas. Supposons donc disposer d'une fonction merge qui prend deux tas t_1 et t_2 en paramètres, et qui renvoie un nouveau tas combinant les éléments de t_1 et t_2.

— Ajout d'un élément `x` à un tas `t`. Remarquons d'abord que l'arbre `N(E, x, E)` est un tas (quel que soit `x`). On peut donc réaliser l'ajout en fusionnant `t` avec le tas précédent contenant l'unique élément `x` :

$$\mathsf{add}(x, t) \quad = \quad \mathsf{merge}(\mathsf{N}(\mathsf{E}, x, \mathsf{E}), t)$$

Le code caml est immédiat.

```
let add x t =
  merge (N(E, x, E)) t
```

— Retirer la racine d'un tas revient à reformer un tas avec les éléments de ses deux sous-arbres, qui sont eux-même des tas. On peut prendre pour cela le résultat de la fusion de ces deux sous-arbres.

$$\mathsf{remove\_min}(\mathsf{N}(A_1, x, A_2)) \quad = \quad \mathsf{merge}(A_1, A_2)$$

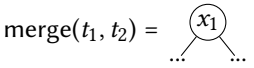
Le code caml s'obtient également directement.

```
let remove_min = function
  | E          -> raise Not_found
  | N(l, x, r) -> merge l r
```

Fusion de *skew heaps*. Ne reste plus qu'à savoir réaliser cette fusion. Si l'un ou l'autre des tas à fusionner est vide le résultat est immédiat : il suffit de prendre l'autre. Supposons alors avoir deux tas non vides.

On cherche à former un nouveau tas contenant tous les éléments de t_1 et de t_2. Ce tas à construire devra avoir son plus petit élément à la racine. Comme t_1 et t_2 sont tous deux des tas, on sait que x_1 est le plus petit élément de t_1, et x_2 le plus petit élément de t_2. Le plus petit élément de l'ensemble ne peut donc être que x_1 ou x_2. On compare donc x_1 et x_2.

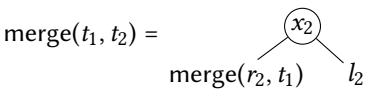
— Si $x_1 \leq x_2$, alors x_1 est le plus petit élément du tas fusionné, qui devra nécessairement avoir la forme



où les deux sous-arbres se partagent les éléments non encore utilisés, c'est-à-dire les deux sous-tas l_1, r_1 de t_1, et le tas t_2 entier. On sait que l_1, r_1 et t_2 sont trois tas et ne contiennent que des éléments supérieurs ou égaux à x_1 : chacun aurait le droit de constituer l'un des deux sous-arbres de notre résultat. Cependant, il n'y a que deux places pour trois : en supposant que l'on utilise directement l_1 ou r_1 ou t_2 comme sous-arbre droit, il faut fusionner les deux autres pour avoir un unique sous-arbre gauche. *N'importe quel choix de répartition formerait un tas correct.* En voici deux possibles :

Dans la version de gauche, on garde ce qui vient de t_1 d'un côté et ce qui vient de t_2 de l'autre. Dans la version de droite, on les entremêle (et on fait passer à gauche le sous-arbre droit r_1 de t_1 et à droite son sous-arbre gauche l_1). Dans les deux cas, la complexité de l'opération de fusion est proportionnelle à la hauteur des arbres considérés, et la fusion est donc efficace si les arbres sont équilibrés. Il se trouve que la solution de droite, en faisant tourner les sous-arbres à chaque opération, évite de tout le temps charger le même côté et permet ainsi de maintenir un bon équilibre *en moyenne*[10].

— Si à l'inverse $x_1 > x_2$, on peut faire une construction symétrique :



Résumé de tout ceci en quelques lignes de caml :

```
let rec merge t1 t2 = match t1, t2 with
  | E, t | t, E -> t
  | N(l1, x1, r1), N(l2, x2, r2) ->
     if x1 <= x2 then N (merge r1 t2, x1, l1)
                 else N (merge r2 t1, x2, l2)
```

Utilisation d'un tas pour l'algorithme de Dijkstra. On adapte directement la structure que l'on vient de voir, en stockant non pas un uniquement élément `int`, mais une paire formée d'un sommet et de sa priorité, et on remplace la comparaison `x1 <= x2` par une comparaison des priorités. En voici la transposition en java avec une classe `N` représentant un nœud d'arbre binaire (où la valeur est sockée dans un attribut `v`, et la priorité dans un attribut `p`). On réalise enfin l'interface `PQueue` avec une deuxième classe possédant pour unique attribut un (pointeur vers un) tel nœud, et qui modifie cet attribut à chaque fabrication d'un nouvel arbre.

10. Cette réalisation de la fusion ne produit *pas que* des arbres équilibrés. Une analyse fine permet cependant d'assurer que, sur la durée, les fusions successives appliquées à un même tas auront une complexité moyenne logarithmique : si certaines fusions s'avèrent ponctuellement très coûteuses, il est certain qu'elles seront compensées par de nombreuses fusions très peu coûteuses. On parle ici d'analyse de complexité *amortie*. C'est ce même concept qui est à l'œuvre lorsque l'on dit que l'ajout d'un élément à un tableau redimensionnable peut-être considéré comme une opération en temps constant, alors même que ponctuellement cet ajout peut nécessiter une copie de l'ensemble du tableau.

```
class N {
  final int v, p;
  final N l, r;
  N(N l, int v, int p, N r) {
    this.l = l; this.v = v; this.p = p; this.r = r;
  }

  static N add(int v, int p, N t) {
    return merge(new N(null, v, p, null), t);
  }
  static int min(N t) {
    if (t == null) { throw new IllegalArgumentException(); }
    else           { return t.v; }
  }
  static N removeMin(N t) {
    if (t == null) { throw new IllegalArgumentException(); }
    else           { return merge(t.l, t.r); }
  }
  static N merge(N t1, N t2) {
    if (t1 == null)        { return t2; }
    else if (t2 == null)   { return t1; }
    else if (t1.p <= t2.p) { return new N(merge(t1.r, t2), t1.v, t1.p, t1.l); }
    else                   { return new N(merge(t2.r, t1), t2.v, t2.p, t2.l); }
  }
}
```

```
class SkewQueue implements PQueue {
  private N heap;
  SkewQueue() { this.heap = null; }
  public boolean isEmpty() { return heap == null; }
  public void insert(int v, int p) { heap = N.add(v, p, heap); }
  public int extractMin() {
    int v = N.min(heap);
    heap = N.removeMin(heap);
    return v;
  }
}
```

Pour compléter notre code de l'algorithme de Dijkstra, il suffit maintenant de compléter la ligne manquante par

```
PQueue pqueue = new SkewQueue();
```

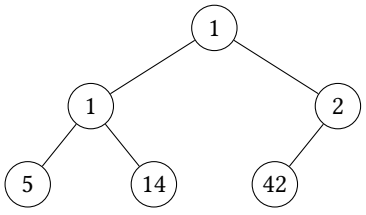
10.4 Approfondissement : tas binaire mutable

Dans certains cas, on peut représenter un arbre binaire par un tableau, où chaque case correspond à un nœud. L'idée est la suivante.

- La racine correspond à la case d'indice 0.
- Le fils gauche du nœud d'indice k est le nœud d'indice $2k + 1$.
- Le fils droit du nœud d'indice k est le nœud d'indice $2k + 2$.

Et donc le parent du nœud d'indice $k > 0$ a l'indice $\left\lfloor \frac{k-1}{2} \right\rfloor$.

Ainsi, l'arbre



est représenté par le tableau suivant.

0	1	2	3	4	5
1	1	2	5	14	42

On parle de représentation *implicite* de l'arbre binaire. *Attention cependant :* ceci n'est adapté que lorsque la forme de l'arbre s'y prête. En l'occurrence il faut un arbre bien équilibré et même dans l'idéal un arbre binaire *quasi-parfait*, c'est-à-dire un arbre dans lequel tous les niveaux de profondeur sont complètement remplis, sauf éventuellement le dernier niveau (dans lequel on demande que les nœuds soient serrés aussi à gauche que possible). En effet, dans le cas contraire le tableau contiendrait des « trous ».

On peut utiliser cette structure pour représenter un tas, dont le bon équilibre est contrôlé de manière stricte. Il s'agira cette fois d'un tas *mutable* : l'ajout ou l'extraction d'un élément modifie le tas en place. Aussi, pour pouvoir traiter l'algorithme de Dijkstra, on stocke dans le tableau des paires formées par une valeur et une priorité.

```
class BinaryQueue implements PQueue {
  // Classe représentant les paires (valeur, priorité)
  static class PrioPair {
    final int v, p;
    PrioPair(int v, int p) { this.v = v; this.p = p; }
  }

  // Support de la file de priorité, avec fonctions de comparaison et d'échange
  ArrayList<PrioPair> t;
  boolean prio(int i, int j) { return t.get(i).p < t.get(j).p; }
  void swap(int i, int j) { Collections.swap(t, i, j); }

  // Fonctions auxiliaires ciblant les voisins utiles
  int parent(int i) { return (i-1)/2; }
  int left(int i)   { return 2*i+1; }
  int right(int i)  { return 2*i+2; }

  BinaryQueue() { this.t = new ArrayList<>(); }
  public boolean isEmpty() { return t.size() == 0; }
```

On insère un nouvel élément en l'ajoutant à la fin du tableau, c'est-à-dire comme une nouvelle feuille. On fait ensuite remonter cet élément vers la racine jusqu'à ce qu'il atteigne une place légitime. Pour cela, la fonction auxiliaire `moveUp` permute l'élément ciblé avec son parent, tant qu'il est strictement plus prioritaire que son parent.

```
  void moveUp(int i) {
    if ( i > 0 && prio(i, parent(i)) ) {
      swap(i, parent(i));
      moveUp(parent(i));
    }
  }

  public void insert(int v, int p) {
    t.add(new PrioPair(v, p));
    moveUp(t.size() - 1);
  }
```

L'opération d'extraction est un peu plus complexe : on remplace l'élément retiré à la racine par l'élément de la dernière case, puis on le fait descendre dans l'arbre tant qu'au moins un de ses fils est plus prioritaire que lui. Point d'attention : lorsque l'on fait descendre un élément il faut l'échanger avec le plus petit de ses deux fils. Il faut aussi prévoir le cas, possible à la fin, où le nœud considéré a un seul fils, qui est alors nécessairement à gauche. La fonction auxiliaire `minChild` sélectionne le plus prioritaire des deux fils (ou l'unique fils le cas échéant), et la fonction auxiliaire `moveDown` réalise les échanges.

```
  int minChild(int i) {
    if ( right(i) < t.size() && prio(right(i), left(i)) ) { return right(i); }
    else                                                  { return left(i); }
  }
  void moveDown(int i) {
    if ( left(i) < t.size() && prio(minChild(i), i) ) {
      swap(i, minChild(i));
      moveDown(minChild(i));
    }
  }
```

```
  public int extractMin() {
    int r = t.get(0).v;
    swap(0, t.size()-1);
    t.remove(t.size()-1);
    moveDown(0);
    return r;
  }
}
```

10.5 Approfondissement : Dijkstra, en caml

On a vu que caml permettait de réaliser facilement une structure de tas. Voyons comment compléter la réalisation de l'algorithme de Dijkstra dans ce langage également.

Graphes pondérés. Pour commencer, fixons un type pour les graphes pondérés. On peut utiliser un tableau de listes d'adjacence comme déjà vu en java. Ici plus précisément, on représente une liste d'adjacence avec une liste primitive de caml, qui contient des paires (voisin, longueur de l'arête). La fonction `succ` renvoie directement cette liste de paires (on n'aura donc pas besoin ici de la fonction `weight` vue dans l'interface java).

```
type wgraph = (int * int) list array

let size g = Array.length g
let succ s g = g.(s)
```

File de priorité polymorphe. Comme déjà vu avec java, notre file de priorité devra contenir à la fois des sommets et leurs priorités. Voici une définition *polymorphe* (ou *générique*) de la structure de tas qui va permettre cela. Le *paramètre de type* `'a` y désigne un type de données quelconque, et le type `'a heap` est un tas dont les nœuds portent des valeurs de type `'a`.

```
type 'a heap =
  | E
  | N of 'a heap * 'a * 'a heap
```

La version que nous avions déjà vue correspondait à `int heap`, mais tout le code donné pour les tas fonctionne également avec un type `'a` quelconque. En particulier, on voudra maintenant que ce `'a` ne représente pas un entier seul, mais une paire `int * int` contenant une priorité et un numéro de sommet (en caml, contrairement à java, les paires sont des types primitifs).

Petite subtilité sur la comparaison < en caml : elle est elle-même polymorphe. Elle ne s'applique donc pas seulement aux nombres entiers, mais à tout type de données. Lors de la comparaison de structures de données à plusieurs champs elle applique l'ordre lexicographique. En particulier, pour comparer deux paires (p_1, v_1) et (p_2, v_2), elle compare d'abord p_1 et p_2, et ne compare ensuite v_1 et v_2 qu'en cas d'égalité sur la composante p. Ici, nous allons donc utiliser des paires dans lesquelles le premier élément est la priorité, et le deuxième élément est le numéro du sommet (et ainsi le code déjà donné est encore valide).

Ensuite, on définit une file de priorité comme un pointeur vers un tel tas. Note caml : un type `'b ref` est le type d'une *référence*, c'est-à-dire d'un pointeur, vers un élément de type `'b`. Ici, `'b` est le type d'un tas, c'est-à-dire `'a heap`. On crée un nouveau pointeur avec l'opérateur `ref`, on lit la valeur associée avec l'opérateur `!` de *déréférencement*, et on modifie la valeur avec l'opérateur `:=` d'*affectation*.

Note : les parenthèses dans `('a heap) ref` sont facultatives.

```
type 'a pqueue = ('a heap) ref

let create ()  = ref E
let is_empty q = !q = E
let insert e q = q := add e !q
let extract_min q =
  let e = min !q in
  q := remove_min !q;
  e
```

Dijkstra. Le code a la même structure que son équivalent en java, avec quelques différences mineures : on définit une fonction auxiliaire locale `upd` qui regroupe les opérations d'insertion d'un sommet dans la file de priorité, et de mise à jour de ses informations dans `dist` et `pred` (le code correspondant n'est donc écrit qu'une fois, mais utilisé à deux endroits), lors de l'extraction d'un sommet de la pile on récupère directement sa distance (en java on ne récupérait que le numéro du sommet, puis on consultait `dist`), et l'itération sur les voisins d'un sommets se fait avec la fonction d'itération `iter` (en java on avait une boucle *for each*).

```
let dijkstra s g =
  (* Initialisation *)
  let n = WGraph.size g in
  let visited = Array.make n false in
  let dist = Array.make n (-1) in
  let pred = Array.make n (-1) in
  let pqueue = PQueue.create () in

  (* Fonction auxiliaire pour mettre à jour les informations d'un sommet
     et l'ajouter à la file de priorité *)
  let upd v d t =
    PQueue.insert (d, v) pqueue;
    dist.(v) <- d;
    pred.(v) <- t
  in
  upd s 0 s;

  while not (PQueue.is_empty pqueue) do
    let dt, t = PQueue.extract_min pqueue in
    if (not visited.(t)) then
      (visited.(t) <- true;
       (* Itération sur l'ensemble des voisins, pour insérer dans la file
          et mettre à jour ceux qui doivent l'être *)
       List.iter (fun (v, dtv) ->
           if (dist.(v) < 0 || dt+dtv < dist.(v)) then upd v (dt+dtv) t)
       (WGraph.succ t g))
  done;

  dist, pred
```

11 Ne pas se casser la tête

Un club écossais très *select* a fixé les règles suivantes pour autoriser l'adhésion de nouveaux membres.

1. Les membres non écossais doivent porter des chaussettes rouges.
2. Un membre sans kilt ne peut pas porter de chaussettes rouges.
3. Les membres mariés ne doivent pas sortir se promener le dimanche.
4. Un membre sort se promener le dimanche si et seulement s'il est écossais.
5. Tout membre portant un kilt doit être écossais et marié.

À se demander s'il est réellement possible d'entrer dans ce cercle... Pouvez-vous déterminer si certaines situations permettent réellement l'adhésion, et si oui lesquelles ?

Modélisation par des formules. Commençons par recenser les différentes caractéristiques des candidats qui sont concernées par ces règles, et associons une lettre à chacune.

E : Être écossais $\quad$ C : Porter des chaussettes rouges $\quad$ K : Porter le kilt
M : Être marié $\quad$ S : Sortir se promener le dimanche

Les règles d'adhésion peuvent ensuite être modélisées par des formules logiques liant ces différentes caractéristiques.

1. $\neg E \implies C$
2. $\neg K \implies \neg C$
3. $M \implies \neg S$
4. $E \iff S$
5. $K \implies (E \wedge M)$

On en déduit une traduction logique de notre problème : déterminer les combinaisons de valeurs de vérité pour les cinq variables propositionnelles E, C, K, M et S rendant vraie la formule d'adhésion résumée ici.

$$(\neg E \implies C) \ \wedge \ (\neg K \implies \neg C) \ \wedge \ (M \implies \neg S) \ \wedge \ (E \iff S) \ \wedge \ (K \implies (E \wedge M))$$

Résolution automatique ? Vous avez déjà appris à construire, à la main, une table de vérité pour une telle formule, qui permet de résoudre ce problème. Ici, nous aurions un tableau de 32 lignes recensant toutes les combinaisons possibles de valeurs de vérité pour nos 5 variables propotionnelles, et pour chacune la valeur de vérité de la formule complète. Nous allons maintenant voir comment traiter ce problème d'une manière plus algorithmique, en travaillant directement sur la structure des formules. Et cette structure va être... *un arbre !*

11.1 Structure récursive des formules logiques

Jusqu'ici, nous avons décrit une formule logique comme « une combinaison de propositions liées par des connecteurs ». Regardons de plus près la structure d'un tel objet. Une *formule* de la logique propositionnelle est :

- soit l'une des formules atomiques $\top$ (tautologie) ou $\bot$ (contradiction),
- soit une variable propositionnelle X,
- soit la négation $\neg F$ d'une formule F,
- soit la combinaison $F_1 \wedge F_2$ (conjonction), ou $F_1 \vee F_2$ (disjonction), ou $F_1 \implies F_2$ (implication) de deux formules F_1 et F_2 à l'aide d'un connecteur binaire.

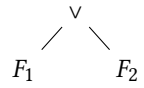
D'autres connecteurs, comme $\iff$, peuvent être encodés par des combinaisons des précédents.

On peut faire trois remarques à propos de cette description. D'une part, elle définit les formules logiques en décrivant comment on peut les *construire*. Une telle définition sous-entend deux choses :

- toute application de ces règles de construction produit une formule valide, et
- toute formule valide peut être obtenue à l'aide d'une telle construction.

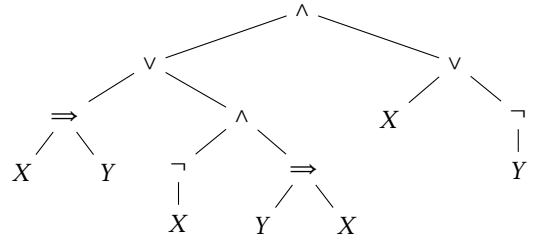
D'autre part, cette définition est récursive : les connecteurs $\neg$, $\wedge$, $\vee$ et $\implies$ construisent une formule propositionnelle *à partir de formules déjà construites*. Enfin, cette définition ressemble à une version enrichie de la définition d'un arbre binaire ! En conséquence de tout ceci, on pourra encore utiliser sur les formules les techniques de définition d'une fonction récursive par cas sur la forme d'une formule, et de raisonnement par récurrence structurelle.

Structure des formules, graphiquement. Pour faire ressortir plus clairement la structure arborescente d'une formule, on peut la représenter graphiquement. Dans le cas d'une formule composée, on place en haut le connecteur principal, et en-dessous de lui les sous-formules, reliées au connecteur, à la manière d'un arbre binaire.



Sur une formule plus riche, la connection se fait au niveau des connecteurs principaux des sous-formules, et certains nœuds ont un seul fils (car la négation est une opération *unaire*).

$$((X \implies Y) \vee (\neg X \wedge (Y \implies X))) \wedge (X \vee \neg Y)$$



Cette représentation graphique ressemble à nouveau à un graphe, avec une structure (un peu) plus variée qu'un arbre binaire. On y retrouve cependant encore la différence qu'on avait déjà évoquée entre graphes et arbres : le placement des nœuds est important. On le voit en particulier avec le connecteur d'implication : les deux formules suivantes n'ont pas la même signification !

Définition récursive de la taille d'une formule. La *taille* $|F|$ d'une formule F est, informellement, le nombre de symboles logiques qu'elle contient (variables propositionnelles incluses). On peut en donner une définition rigoureuse en fournissant une équation décrivant $|F|$ pour chaque forme possible de F. Comme lors de la définition de fonctions sur des listes chaînées ou des arbres binaires, nous avons des cas de base, pour les formules les plus simples, dans lesquels l'équation donne directement une valeur explicite,

$$\begin{aligned} |\top| &= 1 \\ |\bot| &= 1 \\ |X| &= 1 \end{aligned}$$

mais aussi des cas récursifs, pour les formules construites par combinaison de formules plus petites, dans lesquels l'équation fait intervenir la taille de ces sous-formules.

$$\begin{aligned} |\neg F| &= 1 + |F| \\ |F_1 \wedge F_2| &= 1 + |F_1| + |F_2| \\ |F_1 \vee F_2| &= 1 + |F_1| + |F_2| \\ |F_1 \implies F_2| &= 1 + |F_1| + |F_2| \end{aligned}$$

À noter : l'écriture d'une formule utilise des parenthèses pour éviter les ambiguïtés. Ces parenthèses ne font pas partie de la structure elle-même. On ne les a pas représentées dans l'arbre ci-dessus, et on ne les compte pas non plus dans la taille d'une formule. Par exemple :

$$\begin{aligned} |(\neg X \implies Y) \wedge \neg Y| &= 1 + |\neg X \implies Y| + |\neg Y| \\ &= 1 + (1 + |\neg X| + |Y|) + (1 + |Y|) \\ &= 1 + (1 + (1 + |X|) + 1) + (1 + 1) \\ &= 1 + (1 + (1 + 1) + 1) + 2 \\ &= 1 + 4 + 2 \\ &= 7 \end{aligned}$$

Variables d'une formule. Sur le même modèle, on peut donner une caractérisation récursive de l'ensemble $v(F)$ des variables présentes dans une formule F. En voici une dans laquelle on a regroupé les trois cas $F_1 \wedge F_2$, $F_1 \vee F_2$ et $F_1 \Longrightarrow F_2$ sous la forme commune $F_1 \circ F_2$, où le symbole $\circ$ représente un connecteur binaire quelconque.

$$\begin{array}{rcll} v(\top) & = & \emptyset & \\ v(\bot) & = & \emptyset & \\ v(X) & = & \{X\} & \\ v(\neg F) & = & v(F) & \\ v(F_1 \circ F_2) & = & v(F_1) \cup v(F_2) & \circ \in \{\wedge, \vee, \Longrightarrow\} \end{array}$$

Exemple :

$$\begin{array}{rcl} v((\neg X \Longrightarrow Y) \wedge \neg Y) & = & v(\neg X \Longrightarrow Y) \cup v(\neg Y) \\ & = & (v(\neg X) \cup v(Y)) \cup v(Y) \\ & = & (v(X) \cup v(Y)) \cup v(Y) \\ & = & \{X, Y\} \cup \{Y\} \\ & = & \{X, Y\} \end{array}$$

11.2 Représentation des formules en caml : types algébriques

Caml propose un mécanisme natif pour définir des structures de données construites ainsi récursivement : les types algébriques (que nous avons déjà utilisés pour les arbres binaires). On fournit pour cela un ensemble de *constructeurs*, correspondant chacun à l'une des formes possibles que peut prendre l'objet à construire. On peut noter par exemple :

- `True` pour la formule $\top$,
- `False` pour la formule $\bot$,
- `Not(f)` pour la négation de la formule f,
- `And(f1, f2)` pour la conjonction des formules f_1 et f_2,
- `Or(f1, f2)` pour la disjonction des formules f_1 et f_2,
- `Imp(f1, f2)` pour une implication de f_1 vers f_2.

Pour pouvoir différencier les variables, on donnera à chacune un numéro. Cela revient à considérer un ensemble $\{X_0, X_1, X_2, \ldots\}$ (infini, dénombrable) de variables propostionnelles. Il suffit alors de noter :

- `Var(k)` pour la variable X_k.

Avec une telle convention, la formule $\neg X_0 \vee (\neg X_1 \Longrightarrow X_0)$ sera écrite

```
Or(Not(Var(0)), Imp(Not(Var(1)), Var(0)))
```

ou encore de la manière suivante, où on a seulement ajouté un peu d'indentation pour rendre mieux visible la structure.

```
Or(Not(Var(0)),
   Imp(Not(Var(1)),
       Var(0)))
```

Définition d'un type algébrique. Pour introduire une telle structure en caml, on énumère les *constructeurs* que l'on souhaite définir. En l'occurrence : `True`, `False`, `Var`, `Neg`, `And`, `Or`, `Imp`. On précise également pour chacun les différents éléments auxquels il s'applique : rien pour `True` et `False`, un entier pour `Var`, une formule pour `Neg`, deux formules pour `And`, `Or`, `Imp`.

```
type fmla =            (* nom que l'on donne au type des formules *)
  | True               (* constructeur sans paramètre *)
  | False
  | Var of int         (* constructeur avec un paramètre entier *)
  | Not of fmla        (* constructeur s'appliquant à une formule *)
  | And of fmla * fmla (* constructeur s'appliquant à deux formules *)
  | Or  of fmla * fmla
  | Imp of fmla * fmla
```

L'aspect récursif de cette construction se manifeste par la mention du type `fmla` que l'on est en train de définir dans les paramètres de certains des constructeurs.

Définition d'une fonction par filtrage. On définit une fonction manipulant une telle structure à l'aide d'une opération de *filtrage*, notée **`match`**, qui permet d'énumérer les différentes formes possibles des formules, et de fournir un code adapté à chacune.

```
let rec size f = match f with
  | True        -> 1
  | False       -> 1
  | Var(_)      -> 1
  | Not(f)      -> 1 + size f
  | And(f1, f2) -> 1 + size f1 + size f2
  | Or (f1, f2) -> 1 + size f1 + size f2
  | Imp(f1, f2) -> 1 + size f1 + size f2
```

Factorisation de la définition, et simplifications de l'écriture. Dans les équations mathématique définissant la fonction v, on a jugé commode de factoriser les trois cas $F_1 \wedge F_2$, $F_1 \vee F_2$ et $F_1 \implies F_2$ en un unique cas $F_1 \circ F_2$, où $\circ$ peut représenter l'un ou l'autre des trois constructeurs. En effet, ces trois formes ont une structure similaire, et seront probablement régulièrement traitées de la même manière. On peut intégrer cette factorisation à notre définition du type algébrique. Pour cela, on remplace les trois constructions `And(f1, f2)`, `Or(f1, f2)` et `Imp(f1, f2)` par une unique construction `Bin(op, f1, f2)` regroupant les connecteurs binaires. Dans cette nouvelle construction, `f1` et `f2` sont toujours des formules, et `op` est une constante `And`, `Or` ou `Imp` désignant le connecteur.

```
type binop = And | Or | Imp
type fmla =
  | True
  | False
  | Var of int
  | Not of fmla
  | Bin of binop * fmla * fmla
```

Cette définition remplace la précédente (avec de nouvelles significations pour les symboles `And`, `Or` et `Imp`). On représente maintenant comme suit la formule $\neg X_0 \vee (\neg X_1 \implies X_0)$.

```
Bin(Or, Not(Var(0)), Bin(Imp, Not(Var(1)), Var(0)))
```

L'écriture d'une formule est légèrement plus lourd, mais le calcul de la taille est en revanche simplifié, puisqu'il regroupe les trois dernières lignes de la version précédente par

```
  | Bin(_, f1, f2) -> 1 + size f1 + size f2
```

Simplification d'écriture : les parenthèses ne sont pas nécessaires pour un constructeur prenant un unique paramètre, lorsque celui-ci est atomique. C'est le cas notamment pour `Var` lorsqu'on lui fournit une constante entière

```
Bin(Or, Not(Var 0), Bin(Imp, Not(Var 1), Var 0))
```

mais aussi pour `Not` ou `Var` utilisés dans un filtrage avec un identifiant ou un joker.

```
  | Not f -> 1 + size
  | Var _ -> 1
```

Simplification d'écriture : on peut grouper plusieurs cas de filtrage au même comportement.

```
  | True | False -> 1
```

En combinant tous ces éléments, voici la version finale de notre fonction de calcul de taille.

```
let rec size f = match f with
  | True | False | Var _  -> 1
  | Not f                -> 1 + size f
  | Bin (_, f1, f2)      -> 1 + size f1 + size f2
```

Autre exemple : fonction `varmax` donnant le numéro de la plus grande variable d'une formule.

```
let rec varmax f = match f with
  | True | False    -> -1    (* réponse -1 si aucune variable présente *)
  | Var i           -> i     (* i est supposé >= 0 *)
  | Not f           -> varmax f
  | Bin (_, f1, f2) -> max (varmax f1) (varmax f2)
```

11.3 Représentation des formules en java : abstraction et héritage

Dans un langage comme java, on peut facilement définir une structure de données récursive (on l'a vu avec les listes et les arbres aux chapitres précédents), mais on n'a pas de mécanisme dédié pour la définition et la manipulation d'un type algébrique [11]. Pour représenter une telle structure, une technique « orientée objet » traditionnelle consiste à : définir une interface ou une classe abstraite principale pour la structure de données, et une classe concrète implémentant cette interface pour chaque forme que peut prendre la structure.

Dans notre cas, on peut ainsi définir une interface `Fmla` couvrant l'ensemble des formules, dans laquelle on déclare les différentes méthodes que l'on voudra avoir.

```
interface Fmla {
  int size();
  int varmax();
}
```

Il reste ensuite à définir une classe concrète implémentant `Fmla` pour chaque connecteur logique ou forme atomique. Chacune de ces classes concrètes doit contenir :

- des attributs (immuables) pour chaque élément constituant une formule du type donné,
- un constructeur (sauf si le constructeur par défaut suffit),
- l'implémentation des méthodes `size` et `varmax` correspondant à la forme considérée.

Pour les formes atomiques $\top$ et $\bot$, il n'y a pas besoin d'attribut ni de constructeur spécifique, on indique simplement ce que doivent être la taille et la plus grande variable.

```
class True implements Fmla {
  public int size()   { return 1; }
  public int varmax() { return -1; }
}
```

Pour une variable, on introduit un attribut entier immuable désignant le numéro de la variable et un constructeur initialisant cet attribut (en plus des concrétisations des méthodes).

```
class Var implements Fmla {
  final int i;
  public Var(int i) { this.i = i; }
  public int size()   { return 1; }
  public int varmax() { return i; }
}
```

Pour une formule de la forme $\neg F$, on introduit un attribut pour la sous-formule F. Les définitions des méthodes `size` et `varmax` vont appeler les méthodes de la sous-formule.

```
class Not implements Fmla {
  final Fmla f;
  public Not(Fmla f) { this.f = f; }
  public int size()   { return 1 + f.size(); }
  public int varmax() { return f.varmax(); }
}
```

Pour une formule $F_1 \circ F_2$ avec un connecteur binaire $\circ$, on a deux attributs pour les sous-formules F_1 et F_2, et un troisième pour le connecteur lui-même. On définit l'ensemble des connecteurs utilisés à l'aide d'une énumération.

```
enum Binop { AND, OR, IMP }
class Bin implements Fmla {
  final Binop op;
  final Fmla f1, f2;
  public Bin(Binop op, Fmla f1, Fmla f2) {
    this.op = op; this.f1 = f1; this.f2 = f2;
  }
  public int size()   { return 1 + f1.size() + f2.size(); }
  public int varmax() { return Math.max(f1.varmax(), f2.varmax()); }     }
```

11. Depuis 2020, une grande part des nouveautés introduites par les mises à jour successives du langage java s'inscrivent dans un effort d'améliorer ceci, en intégrant à java certains des éléments que nous avons vus en caml. La version 21 de java (sept. 2023) en a stabilisé une bonne partie mais tout n'est pas encore terminé (certains aspects sont encore en cours de test, et d'autres restent à implémenter). Pour suivre l'évoluation de la situation, surveiller dans la documentation les mots-clés *sealed class*, *record class*, et *pattern matching*. `https://openjdk.org/jeps/0`
`https://docs.oracle.com/en/java/javase/21/language/java-language-changes.html`

Avec cette mise en place, chaque nouvelle fonction manipulant les formules se traduit par :

— une déclaration dans l'interface `Fmla`,

— une définition adaptée dans chaque classe concrète.

Ainsi, chaque classe concrète contient des définitions traduisant toutes les équations relatives à cette forme. Par exemple, `Bin.size` et `Bin.varmax` traduisent les deux équations

$$\begin{array}{rcl} |F_1 \circ F_2| & = & 1 + |F_1| + |F_2| \\ v(F_1 \circ F_2) & = & \max(v(F_1), v(F_2)) \end{array}$$

Avec cette famille de classes, note formule exemple $\neg X_0 \vee (\neg X_1 \Longrightarrow X_0)$ peut être définie par

```
Fmla f = new Bin(Binop.OR,
                 new Not(new Var(0)),
                 new Bin(Binop.IMP, new Not(new Var(1)), new Var(0)))
```

11.4 Évaluation et transformation de formules

Sémantique booléenne. *Évaluer* une formule propositionnelle, c'est lui associer une valeur de vérité booléenne (vrai ou faux). Une telle évaluation se fait en fonction d'une *interprétation* des variables propositionnelles, c'est-à-dire d'une fonction associant une valeur de vérité à chaque variable. À des fins de programmation, et avec nos variables numérotées, on peut résumer une interprétation des variables par un tableau de booléens. D'où la fonction `eval` en caml, qui prend en paramètres une formule `f` (type `fmla`) et une interprétation `v` (type `bool array`), et qui renvoie une valeur de vérité (type `bool`).

```
let rec eval f v = match f with
  | True -> true
  | False -> false
  | Var i -> v.(i)    (* accès à l'élément d'indice i du tableau v *)
  | Not f -> not (eval f v)
  | Bin (And, f1, f2) -> eval f1 v && eval f2 v
  | Bin (Or,  f1, f2) -> eval f1 v || eval f2 v
  | Bin (Imp, f1, f2) -> not (eval f1 v) || eval f2 v
```

En java, cette nouvelle fonction est déclarée dans l'interface, et son code est réparti entre les différentes classes.

— Dans l'interface `Fmla`, on déclare :

```
boolean eval(boolean[] v);
```

— Dans la classe `True`, la méthode est constante :

```
public boolean eval(boolean[] v) { return true; }
```

— Dans la classe `False`, de même :

```
public boolean eval(boolean[] v) { return false; }
```

— Dans la classe `Var`, on consulte l'interprétation pour la variable numéro **`this`**`.i` :

```
public boolean eval(boolean[] v) { return v[i]; }
```

— Dans la classe `Not`, on évalue la sous-formule :

```
public boolean eval(boolean[] v) { return !f.eval(v); }
```

— Dans la classe `Bin`, on évalue les sous-formules et on consulte le connecteur.

```
public boolean eval(boolean[] v) {
  switch (op) {
    case AND: return f1.eval(v) && f2.eval(v);
    case OR:  return f1.eval(v) || f2.eval(v);
    case IMP: return !f1.eval(v) || f2.eval(v);
    default: throw new IllegalArgumentException();
  }
}
```

Il y a précisément deux exceptions. Lesquelles ?

Simplifications de formules. Dans chaque formule où apparaît l'une des formules atomiques $\top$ ou $\bot$, on peut faire des simplifications. Voici les équations que l'on peut appliquer :

$$\begin{array}{lll} F \wedge \bot \equiv \bot \wedge F \equiv \bot & F \wedge \top \equiv \top \wedge F \equiv F & \neg\top \equiv \bot \\ F \vee \top \equiv \top \vee F \equiv \top & F \vee \bot \equiv \bot \vee F \equiv F & \neg\bot \equiv \top \\ F \Longrightarrow \top \equiv \top & \top \Longrightarrow F \equiv F & \neg(\neg F) \equiv F \\ \bot \Longrightarrow F \equiv \top & F \Longrightarrow \bot \equiv \neg F & \end{array}$$

Exemple :

$$\begin{array}{ll} & (\bot \Longrightarrow Z) \wedge \neg((Y \vee \top) \wedge (X \Longrightarrow \bot)) \\ \equiv & \top \wedge \neg((Y \vee \top) \wedge (X \Longrightarrow \bot)) \\ \equiv & \top \wedge \neg(\top \wedge (X \Longrightarrow \bot)) \\ \equiv & \top \wedge \neg(\top \wedge \neg X) \\ \equiv & \top \wedge \neg(\neg X) \\ \equiv & \top \wedge X \\ \equiv & X \end{array}$$

Pour simplifier au maximum une formule à l'aide de ces équations, on peut combiner deux fonctions : une fonction `try_simp` qui essaie d'appliquer l'une des équations à la racine de la formule, et une fonction `simp` qui itère cette fonction `try_simp` sur tous les nœuds de la formule. En caml :

```
let try_simp f = match f with
  | Not True   -> False
  | Not False  -> True
  | Not(Not f) -> f
  | Bin(And, False, _) | Bin(And, _, False) -> False
  | Bin(And, True,  f) | Bin(And, f, True ) -> f
  | Bin(Or,  True,  _) | Bin(Or,  _, True ) -> True
  | Bin(Or,  False, f) | Bin(Or,  f, False) -> f
  | Bin(Imp, False, _) | Bin(Imp, _, True ) -> True
  | Bin(Imp, True,  f) | Bin(Imp, Not f, False) -> f
  | Bin(Imp, f, False) -> Not f
  | _ -> f

let rec simp f = match f with
  | True | False | Var _ -> f
  | Not f                -> try_simp (Not (simp f))
  | Bin(op, f1, f2)      -> try_simp (Bin(op, simp f1, simp f2))
```

Validité des simplifications. Une formule F' obtenue par simplification est *équivalente* à la formule F d'origine, ce qu'on note $F' \equiv F$ et qu'on définit par :

Pour toute interprétation v, on a $\mathsf{eval}(F', v) = \mathsf{eval}(F, v)$.

On peut le montrer en deux temps.

1. D'abord, on vérifie que chaque équation est bien correcte (il suffit de faire des tables de vérité pour chaque). Cela suffit à justifier que `try_simp` produit une formule équivalente à la formule prise en argument.

   Pour toute formule F, on a $\mathsf{try\_simp}(F) \equiv F$.

2. Ensuite, on montre que les simplifications récursives réalisées par `simp` sont encore valides.

   Pour toute formule F, on a $\mathsf{simp}(F) \equiv F$.

   On le démontre ci-dessous par *récurrence structurelle* sur F.

Raisonnement par récurrence structurelle sur les formules. Pour démontrer qu'une propriété $P(F)$ est vraie pour toute formule F, il suffit de vérifier que les règles de construction des formules ne permettent de construire que des formules vérifiant P, c'est-à-dire d'une part que la propriété P est vraie pour les formules atomiques, et que d'autre part toute

construction d'une nouvelle formule à l'aide d'un connecteur *préserve* la propriété P : si les sous-formules prises comme briques de base satisfont P, alors la nouvelle formule construite doit encore satisfaire P.

Ainsi, il suffit de vérifier que :

1. la propriété $P(\top)$ est vraie,
2. la propriété $P(\bot)$ est vraie,
3. la propriété $P(X)$ est vraie pour toute variable propositionnelle X,
4. la propriété $P(\neg F)$ est vraie pour toute formule F telle que $P(F)$ est vraie,
5. la propriété $P(F_1 \circ F_2)$ est vraie pour tout connecteur binaire $\circ$ et toutes formules F_1 et F_2 telles que $P(F_1)$ et $P(F_2)$ sont vraies,

pour s'assurer que la propriété P est vraie *pour toutes les formules*. Les trois premiers points sont des cas de base, et les deux derniers des cas récursifs. Dans les deux derniers points, les hypothèses $P(F)$, $P(F_1)$ et $P(F_2)$ que l'on suppose pour justifier $P(\neg F)$ ou $P(F_1 \circ F_2)$ sont les *hypothèses de récurrence*. Il s'agit du même principe de raisonnement par récurrence structurelle que nous avons déjà vu sur les listes et les arbres binaires.

Application à la simplification. On montre que pour toute formule F, on a $\mathsf{simp}(F) \equiv F$. On vérifie les cinq points du principe de récurrence.

1. Cas $\top$. Par définition $\mathsf{simp}(\mathsf{True}) = \mathsf{True}$, et on a donc bien $\mathsf{simp}(\mathsf{True}) \equiv \mathsf{True}$.
2. Cas $\bot$. De même, $\mathsf{simp}(\mathsf{False}) = \mathsf{False} \equiv \mathsf{False}$.
3. Cas X. De même, $\mathsf{simp}(\mathsf{Var}(i)) = \mathsf{Var}(i) \equiv \mathsf{Var}(i)$.
4. Cas $\neg F$, en supposant que $\mathsf{simp}(F) \equiv F$. *(on veut montrer simp*$(\neg F) \equiv \neg F$*)*
   On calcule :

$$\begin{array}{lcll} \mathsf{simp}(\mathsf{Not}(F)) & = & \mathsf{try\_simp}(\mathsf{Not}(\mathsf{simp}(F))) & \text{par déf. de simp} \\ & \equiv & \mathsf{Not}(\mathsf{simp}(F)) & \text{par propriété de try\_simp} \\ & \equiv & \mathsf{Not}(F) & \text{par hypothèse de récurrence} \end{array}$$

5. Cas $F_1 \circ F_2$, en supposant que $\mathsf{simp}(F_1) \equiv F_1$ et $\mathsf{simp}(F_2) \equiv F_2$.
   (on veut montrer simp$(F_1 \circ F_2) \equiv F_1 \circ F_2$*)*
   On calcule :

$$\begin{array}{lcll} & & \mathsf{simp}(\mathsf{Bin}(\circ, F_1, F_2)) & \\ & = & \mathsf{try\_simp}(\mathsf{Bin}(\circ, \mathsf{simp}(F_1), \mathsf{simp}(F_2))) & \text{par déf. de simp} \\ & \equiv & \mathsf{Bin}(\circ, \mathsf{simp}(F_1), \mathsf{simp}(F_2)) & \text{par propriété de try\_simp} \\ & \equiv & \mathsf{Bin}(\circ, F_1, \mathsf{simp}(F_2)) & \text{par hyp. de récurrence sur } F_1 \\ & \equiv & \mathsf{Bin}(\circ, F_1, F_2) & \text{par hyp. de récurrence sur } F_2 \end{array}$$

11.5 Approfondissement : algorithme de *backtracking*

Partant d'une formule F, on cherche une interprétation des variables propositionnelles de F qui rende la formule vraie. Une option pour cela est d'*explorer* l'ensemble des combinaisons de valeurs de vérité. Dans cette approche, on choisit une valeur de vérité arbitraire pour une première variable, puis pour une deuxième et ainsi de suite jusqu'à aboutir, soit à une solution (les valeurs choisies suffisent à rendre la formule vraie), soit à une impasse (les valeurs choisies suffisent à rendre la formule fausse). Lorsque l'on arrive à une impasse, on « revient sur nos pas » (« *backtrack* ») jusqu'au choix le plus récent, et on essaie à nouveau avec la valeur opposée.

Est-ce une exploration en largeur, ou en profondeur ?

Algorithme. L'algorithme d'exploration part d'une formule F et d'une interprétation v d'*une partie* des variables de F (on parle d'*interprétation partielle*) et cherche à compléter v d'une manière qui rende F vraie.

- Si v suffit à ce que F s'évalue à vrai, alors v est une solution, on peut s'arrêter.
- Si v suffit à ce que F s'évalue à faux, alors aucune manière de compléter v ne pourra donner une solution : on abandonne cette voie.
- Sinon, on choisit une variable X_i de F pas encore interprétée dans v, et récursivement :
  - on explore avec l'interprétation étendue $v, X_i \mapsto \text{vrai}$,
  - et en cas d'échec on explore avec l'interprétation opposée $v, X_i \mapsto \text{faux}$.

  Si ces deux tentatives échouent, on abandonne cette voie.

Pour évaluer la valeur d'une formule F avec une interprétation partielle v, on propose la technique suivante :

1. dans F, on remplace X par $\top$ si $v(X)$ = vrai et par $\bot$ si $v(X)$ = faux,
2. on simplifie la formule ainsi obtenue,
3. on a un résultat immédiat si la simplification donne $\top$ ou $\bot$.

Exemple. Considérons la formule

$$(X \implies Y) \wedge (\neg X \vee (\neg Y \wedge \neg Z)) \wedge ((Y \implies \neg Z) \vee X)$$

et partons de l'interprétation vide. On considère d'abord la variable X.

— On explore d'abord en fixant $v(X)$ = vrai. La formule simplifiée est

$$\begin{aligned}&(\top \implies Y) \wedge (\neg\top \vee (\neg Y \wedge \neg Z)) \wedge ((Y \implies \neg Z) \vee \top)\\ \equiv\ & Y \wedge (\bot \vee (\neg Y \wedge \neg Z)) \wedge \top\\ \equiv\ & Y \wedge (\neg Y \wedge \neg Z)\end{aligned}$$

On considère ensuite la variable Y.

— Avec $v(Y)$ = vrai, on échoue avec la formule $\top \wedge (\neg\top \wedge \neg Z) \equiv \bot$.

— Avec $v(Y)$ = faux, on échoue avec la formule $\bot \wedge (\neg\bot \wedge \neg Z) \equiv \bot$.

Aucune valeur de Y ne permet d'aboutir. Échec sur cette voie, on revient à X.

— On explore maintenant en fixant $v(X)$ = faux. La formule simplifiée est

$$\begin{aligned}&(\bot \implies Y) \wedge (\neg\bot \vee (\neg Y \wedge \neg Z)) \wedge ((Y \implies \neg Z) \vee \bot)\\ \equiv\ & \top \wedge (\top \vee (\neg Y \wedge \neg Z)) \wedge (Y \implies \neg Z)\\ \equiv\ & Y \implies \neg Z\end{aligned}$$

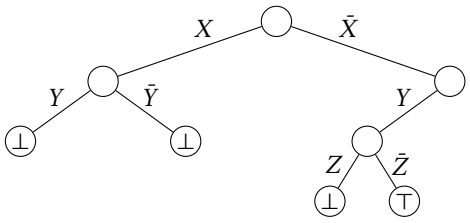
On considère ensuite la variable Y.

— Avec $v(Y)$ = vrai, on obtient la formule $\top \implies \neg Z \equiv \neg Z$. On considère alors Z.

— Avec $v(Z)$ = vrai, on échoue avec la formule $\neg\top \equiv \bot$.

— Avec $v(Z)$ = faux, on obtient la formule $\neg\bot \equiv \top$: on a trouvé une solution !

Finalement, on a déterminé que l'interprétation $\{X \mapsto$ faux, $Y \mapsto$ vrai, $Z \mapsto$ faux$\}$ satisfait la formule. L'ensemble de cette exploration peut être résumé par l'arbre des différents choix testés.



En caml. On se donne une fonction auxiliaire `subst` pour remplacer une variable X_i par une formule F_s dans une formule F :

```
let rec subst i fs f = match f with
  | Var j when i = j -> fs
  | Var _            -> f
  | True | False     -> f
  | Not f            -> Not (subst i fs f)
  | Bin(op, f1, f2)  -> Bin(op, subst i fs f1, subst i fs f2)
```

On peut ensuite définir la fonction d'exploration récursive, qui prend en paramètres une formule `f` et une interprétation partielle `v`, et qui renvoie soit `Some v'` si on a trouvé une extension `v'` de `v` satisfaisant `f`, soit `None` s'il n'y a pas de solution. Chaque appel récursif est fait après remplacement de la variable par `True` ou `False`.

Sur le problème du club écossais, on trouve l'interprétation

$E \mapsto$ vrai
$C \mapsto$ faux
$K \mapsto$ faux
$M \mapsto$ faux
$S \mapsto$ vrai

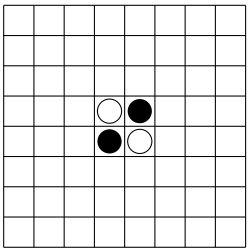
après avoir exploré 9 interprétations partielles.

```
let rec backtrack f v = match simplify f with
  | True  -> Some v
  | False -> None
  | f     -> let x = varmax f in
             let r = backtrack (subst x True f) ((x, true) :: v) in
             if Option.is_some r then r
             else backtrack (subst x False f) ((x, false) :: v)
let sat f = backtrack f []
```

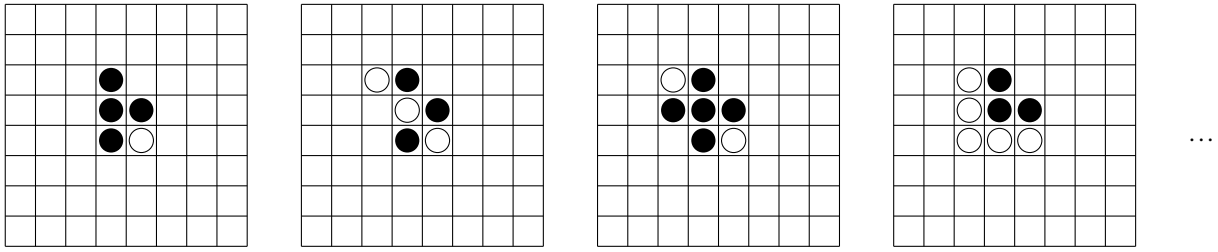
12 Se faire battre à tous les coups

12.1 Problème : algorithme gagnant pour Othello

Le jeu Othello, encore appelé Reversi, se joue sur un échiquier 8×8 avec des pions blancs d'un côté et noirs de l'autre. Initialement, quatre pions sont placés comme ceci au centre de l'échiquier :



Chaque joueur a une couleur et les noirs commencent. Un coup consiste à poser un pion de sa couleur sur une case libre de l'échiquier en réalisant au moins une « prise », c'est-à-dire une rangée de pions adverses (horizontale, verticale ou diagonale) encadrée par deux pions de sa couleur. Tous les pions de l'échiquier qui se retrouvent ainsi encadrés sont retournés et deviennent donc des pions du joueur qui vient de jouer. Voici un début de partie possible :



…

On note comment les blancs ont réalisé deux prises avec leur deuxième coup. Lorsqu'un joueur ne peut jouer, il passe son tour. La partie se termine lorsqu'aucun des deux joueurs ne peut jouer. Le joueur disposant alors du plus grand nombre de pions de sa couleur sur l'échiquier gagne la partie.

Dans ce chapitre, on cherche à définir un algorithme capable de nous battre à tous les coups au jeu d'Othello.

12.2 Graphe implicite et exploration

L'état courant d'une partie est défini par deux informations : la configuration des pions sur l'échiquier, et l'identification du joueur devant jouer le prochain coup. On peut alors voir le jeu comme un graphe dans lequel :

- chaque état possible de la partie correspond à un sommet,
- on a une arête de l'état s vers l'état t lorsque l'un des coups autorisés dans l'état s mène à l'état t.

Alors, chaque partie imaginable correspond à un chemin dans ce graphe, allant de l'état initial du jeu jusqu'à un état où la partie est déclarée terminée.

Contrairement à ce qu'on a fait jusque là, on ne construira pas explicitement un tel graphe en mémoire (sa taille serait prohibitive). On se contente à la place de fournir une structure de données décrivant un état du jeu, et une fonction calculant la liste des coups possibles. Voici la définition java de l'interface d'une telle structure.

```
public interface GameState {
  // caractéristiques d'un état
  char player();     // joueur dont c'est le tour
  boolean isFinal(); // partie terminée ?
  char outcome();    // si partie terminée, vainqueur ? (ou partie nulle ?)

  // les coups possibles, représentés par la liste des états où mènent ces coups
  List<GameState> moves();
}
```

12.3 Stratégies gagnantes

Pour jouer intelligemment à un tel jeu, on explore tout ou partie du graphe des coups possibles, pour déterminer les meilleurs coups. On cherche en particulier les *configurations gagnantes*, c'est-à-dire les états du jeu ayant l'une de ces trois propriétés :

— la partie est terminée, et le joueur gagne, ou

— c'est le tour du joueur, et l'un des coups mène à une configuration gagnante, ou

— c'est le tour de l'adversaire, et tous les coups mènent à une configuration gagnante.

Autrement dit, une configuration est gagnante pour le joueur si :

— tout coup de l'adversaire mène à une configuration telle que,

— il existe un coup du joueur menant à une configuration telle que,

— tout coup de l'adversaire mène à une configuration telle que,

— il existe un coup du joueur menant à une configuration telle que,

— ...

— la partie est terminée et le joueur gagne.

(on a pris ici comme point de départ une situation où c'est au tour de l'adversaire de jouer, il suffit d'en oublier la première ligne pour décrire la situation où c'est le tour du joueur). L'alternance entre les quantificateurs universels et existentiels traduit deux faits :

Le fait qu'il existe un chemin d'une configuration donnée à un état gagnant ne suffit pas à garantir la victoire, car on ne maîtrise pas les coups de l'adversaire !

— le joueur peut choisir son coup, il lui suffit donc qu'au moins un des coups possibles soit bon (à charge pour lui de déterminer lequel !),

— le joueur ne peut pas choisir les coups de son adversaires, il ne gagne donc à coup sûr que si aucun des coups que peut choisir l'adversaire ne prive le joueur de la victoire.

Une fonction très simple permet alors, partant d'un état du jeu `s` quelconque, d'explorer tous les états futurs possibles, pour déterminer si cet état `s` est gagnant pour le joueur `p`.

```
public static boolean winning(GameState s, char p) {
  if (s.isFinal()) {
    // Si la partie est terminée, le joueur gagne-t-il ?
    return (s.outcome() == p);
  }
  if (s.player() == p) {
    // Si c'est le tour du joueur, existe-t-il un coup gagnant ?
    for (GameState m : s.moves()) {
      if (winning(m, p)) return true;
    }
    return false;
  } else {
    // Si c'est le tour de l'adversaire, tous les coups sont-ils gagnants ?
    for (GameState m : s.moves()) {
      if (!winning(m, p)) return false;
    }
    return true;
  }
}
```

Problème : cette fonction n'est pas utilisable en pratique, à part pour les jeux les plus simples. En supposant qu'à chaque tour, le joueur ou l'adversaire a ne serait-ce que deux coups différents possibles, le nombre d'états à explorer est exponentiel en le nombre de coups joués (environ 60 pour Othello !) On peut bien sûr en économiser un peu en mémorisant les états déjà analysés, de la même manière qu'on marque les sommets d'un graphe rencontrés lors d'une exploration, mais cela ne sera jamais assez pour briser ce caractère exponentiel (le nombre d'états différents possibles à Othello est bien exponentiel en la taille du plateau).

12.4 Algorithme Min-Max : exploration à profondeur bornée

Pour un jeu comme Othello, il est impossible d'explorer jusqu'au bout l'ensemble des séquences de coups possibles jusqu'à la fin de la partie. On peut cependant déjà obtenir de très bon résultats en explorant seulement *quelques coups à l'avance*. L'exploration d'une séquence peut alors s'arrêter de deux manières différentes :

— si la séquence analysée va jusqu'à une partie terminée, on utilise comme précédemment la fonction `outcome` pour connaître l'éventuel vainqueur,

— si la séquence analysée s'arrête car on a déjà exploré le nombre de coups fixé, il nous faut *évaluer* à quel point l'état atteint est intéressant (ou au contraire catastrophique !) pour le joueur.

L'évaluation est *heuristique* : il ne s'agit pas d'un jugement absolu déterminant si l'état est gagnant ou non, mais seulement d'une indication que l'état *semble favorable*, ou au contraire défavorable, au joueur. Traditionnellement, l'évaluation prend la forme d'un score entier dans un intervalle fixé, par exemple l'intervalle $[-10000, 10000]$, où les nombres positifs sont favorables au joueur, les nombres négatifs défavorables, et où l'amplitude indique à quel point un état semble favorable ou défavorable.

À Othello, on pourra prendre les critères suivants pour évaluer un état :

— les jetons de la couleur du joueur dans les coins du plateau sont *très intéressants* (+9 par jeton),

— les jetons de la couleur du joueur sur les bords sont *intéressants* (+3 par jeton),

— les jetons de la couleur du joueur ailleurs sur le plateau sont *modérément intéressants* (+1 par jeton),

et pour chaque jeton adverse : les mêmes points en négatif. Si c'est le tour du joueur, on peut également compter quelques points supplémentaires pour chaque coup légal (avoir de nombreuses possibilités est généralement favorable). En voici une réalisation simple. Le plateau (`board`) est représenté par une chaîne de 64 caractères, avec : `'B'` pour un jeton noir, `'W'` pour un jeton blanc, `'.'` pour une case libre. Le joueur (`player`) est représenté par le caractère `'B'` ou `'W'`. La fonction `eval` calcule notre heuristique du point de vue d'un joueur `p`, et `other(p)` renvoie le joueur autre que `p`.

```
public class OthelloState implements GameState {
  final String board;
  final char player;

  public int eval(char p) {
    int value = (player == p)?moves().size():0;
    int[] weight = {
      9, 3, 3, 3, 3, 3, 3, 9,
      3, 1, 1, 1, 1, 1, 1, 3,
      3, 1, 1, 1, 1, 1, 1, 3,
      3, 1, 1, 1, 1, 1, 1, 3,
      3, 1, 1, 1, 1, 1, 1, 3,
      3, 1, 1, 1, 1, 1, 1, 3,
      3, 1, 1, 1, 1, 1, 1, 3,
      9, 3, 3, 3, 3, 3, 3, 9
    };
    for (int i=0; i<64; i++) {
      if (board.charAt(i) == p) value += weight[i];
      else if (board.charAt(i) == other(p)) value -= weight[i];
    }
    return value;
  }

  public static char other(char p) {
    switch (p) {
      case 'W': return 'B';
      case 'B': return 'W';
      default: throw new IllegalArgumentException();
    }
  }
}
```

L'évaluation d'un état *avec une profondeur d*, c'est-à-dire en tenant compte de toutes les séquences possibles de d prochains coups, suit alors les principes suivants.

— Si l'état est celui d'une partie terminée, on lui attribue l'un des scores +10000 (partie gagnée), -10000 (partie perdue) ou 0 (partie nulle).

— À profondeur 0, on attribue à l'état le score heuristique donné par la fonction `eval`.

— À profondeur $d > 0$, on calcule les scores à profondeur $d - 1$ de tous les états suivants possibles, puis :

— si c'était le tour du joueur, on sélectionne le score le plus favorable (le joueur choisit son coup, on suppose qu'il le fait au mieux de ses capacités),

— si c'était le tour de l'adversaire, on sélectionne le score le moins favorable au joueur (l'adversaire, s'il joue bien, optimisera sa propre situation au détriment de celle du joueur).

Cet algorithme est appelé *Min-Max*, puisqu'il évalue un score en alternant des minimisations (des scores à l'issue des tours de l'adversaire) et des maximisations (des scores à l'issue des tours du joueur).

```
static int minmax(GameState state, int d, char p) {
  // Fin de l'exploration
  if (state.isFinal()) {
    char o = state.outcome();
    if (o == p) return 10000;
    else if (o == '.') return 0;
    else return -10000;
  }
  if (d == 0) {
    return state.eval(p);
  }
  // Examen des coups possibles
  int r = (state.player() == p)?-10000:10000;
  for (GameState m : state.moves()) {
    int value = minmax(m, d-1, p);
    if (state.player() == p) { // maximisation
      if (value > r) { r = value; }
    } else { // minisation
      if (value < r) { r = value; }
    }
  }
  return r;
}
```

Pour sélectionner un coup, il suffit alors d'évaluer tous les coups possibles à une profondeur fixée (par exemple : 4), puis prendre le plus favorable.

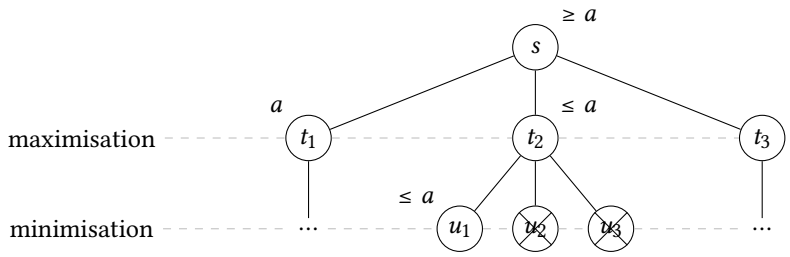
```
static GameState play(GameState state) {
  int bestScore = -10001;
  GameState bestMove = null;
  for (GameState m : state.moves()) {
    int score = minmax(m, DEPTH, state.player());
    if (score > bestScore) {
      bestScore = score;
      bestMove = m;
    }
  }
  assert (bestMove != null);
  return bestMove;
}
```

12.5 Algorithme Alpha/Beta : exploration optimisée

L'*élagage* α/β est une optimisation de l'algorithme Min-Max qui consiste à éviter l'exploration de certaines branches, dès lors l'on sait que ces branches ne peuvent pas modifier le score minimum (ou maximum) en cours de calcul pour une configuration donnée.

Le principe est le suivant. Supposons que l'on considère une position s du joueur, que l'on cherche à évaluer à profondeur d. On cherche donc la valeur maximale, parmi les évaluations à profondeur $d-1$ des configurations t_1, t_2, ..., t_n correspondant aux différents coups possibles du joueur. Supposons que l'on a déterminé une certaine valeur a pour t_1, et que l'on regarde maintenant la configuration t_2. Rappel : dans cette configuration t_2, c'est à l'opposant de

jouer, on cherche donc la valeur minimale parmi les coups possibles u_1, ..., u_k de l'opposant.



Dès que l'on a pu déterminer une valeur r pour l'un des coups u_i de l'opposant, on sait que le minimum sera inférieur ou égal à r, et donc que la valeur calculée pour t_2 ne dépassera pas r. Autrement dit, si on trouve pour l'un des u_i une valeur inférieur ou égale à a, alors on sait que l'évaluation de t_2 sera inférieur ou égale à l'évaluation a obtenue pour t_1, et donc que le coup t_2 ne sera pas celui donnant la valeur maximale cherchée dans l'évaluation de s (et donc que l'on peut arrêter là l'évaluation de t_2).

Voici donc un nouveau code, qui est une version étendue de la fonction `minmax` prenant deux paramètres supplémentaires `a` et `b`, qui indiquent les seuils auxquels interrompre respectivement une minimisation ou une maximisation. Remarque : lorsque l'on met à jour l'un des seuils `a` ou `b`, c'est dans l'optique d'élaguer des « neveux », c'est-à-dire les successeurs d'un des coups alternatifs au coup que l'on vient d'analyser.

```
static int alphabeta(GameState state, int d, int a, int b, char p) {
  // Fin de l'exploration
  if (state.isFinal()) {
    if (state.outcome() == p) return 10000;
    if (state.outcome() == '.') return 0;
    return -10000;
  }
  if (d == 0) return state.eval(p);
  // Examen des coups possibles
  int r = (state.player() == p)?-10000:10000;
  for (GameState m : state.moves()) {
    int value = alphabeta(m, d-1, a, b, p);
    if (state.player() == p) { // maximisation
      if (value > r) r = value;
      if (r >= b)    return r; // seuil maximisation dépassé
      if (r > a)     a = r;    // mise à jour seuil minisation
    } else { // mininisation
      if (value < r) r = value;
      if (r <= a)    return r; // seuil minimisation dépassé
      if (r < b)     b = r;    // mise à jour seuil maximisation
    }
  }
  return r;
}

// initialisation de alpha et beta à des valeurs extrèmes
static int alphabeta(GameState state, int d, char p) {
  return alphabeta(state, d, -10000, 10000, p);
}

// fonction play similaire à la précédente
```

12.6 Mémoïsation : conserver en mémoire les évaluations déjà faites

Notre algorithme d'exploration, tel qu'il est écrit, n'empêche pas d'évaluer plusieurs fois une même configuration (et donc d'explorer plusieurs fois les coups qui suivent une configuration déjà vue). La profondeur d'exploration étant limitée par le paramètre d, cela ne peut pas d'engendrer de boucle infinie, mais dans certaines situations cela peut rendre les explorations plus longues que nécessaire.

Les algorithmes d'exploration de graphes du chapitre précédent, à l'inverse, marquaient l'ensemble des sommets déjà explorés pour ne pas les traiter à nouveau. Ce marquage utilisait un tableau de booléens, et chaque sommet était associé à une case par son numéro. Cette

solution n'est pas envisageable ici : les configurations possibles d'un jeu comme Othello sont bien trop nombreuses pour qu'on puisse imaginer leur donner un numéro à chacune, puis manipuler un tableau prévoyant une case pour chacune.

Pour obtenir un effet similaire, on va utiliser une structure de données annexe dans laquelle, pour chaque configuration explorée, on mémorise l'évaluation qui a été calculée. Ainsi, au moment d'évaluer une configuration, la procédure devient la suivante.

1. Regarder si la configuration a déjà été évaluée.
2. Si oui, renvoyer directement l'évaluation enregistrée, sans calculer à nouveau.
3. Si non, évaluer normalement, *et enregistrer le résultat obtenu.*

Une structure adaptée pour cela est la *table de hachage*. Pour pouvoir l'utiliser, il faut simplement s'assurer qu'une fonction de hachage est bien disponible pour le type de données utilisé comme clé (ici : une configuration du jeu). On peut par exemple ajouter à notre classe `OthelloState` les définitions suivantes pour assurer cela.

```
public boolean equals(Object o) {
  if (this == o) return true;
  if (o == null || this.getClass() != o.getClass()) return false;
  OthelloState that = (OthelloState) o;
  return this.player == that.player && this.board.equals(that.board);
}

public int hashCode() {
  return Objects.hash(board, player);
}
```

Il suffit alors d'ajouter trois éléments à notre algorithme alpha/beta :

— Avant de commencer une exploration, initialiser une table de hachage :

```
// déclaration
private static HashMap<GameState, Integer> memo;

// initialisation avant exploration
public static int alphabeta(GameState state, int d, char p) {
  memo = new HashMap<>();
  return alphabeta(state, d, -10000, 10000, p);
}
```

— Au début de la fonction récursive, vérifier si la configuration analysée a déjà été traitée :

```
static int alphabeta(GameState state, int d, int a, int b, char p) {
  if (memo.containsKey(state)) return memo.get(state);
  ...
```

— À la fin de cette même fonction, enregistrer le résultat obtenu avant de le renvoyer.

```
  ...
  memo.put(state, r);
  return r;
}
```

On peut même imaginer avoir une seule table de hachage, que l'on réutilise et enrichit à chaque exploration faite au cours d'une partie. Attention cependant : dans ce cas, une même configuration est susceptible d'être évaluée plusieurs fois, à différentes profondeurs, et il faut en tenir compte dans la gestion de la table.