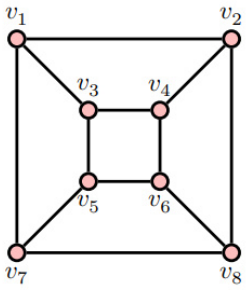
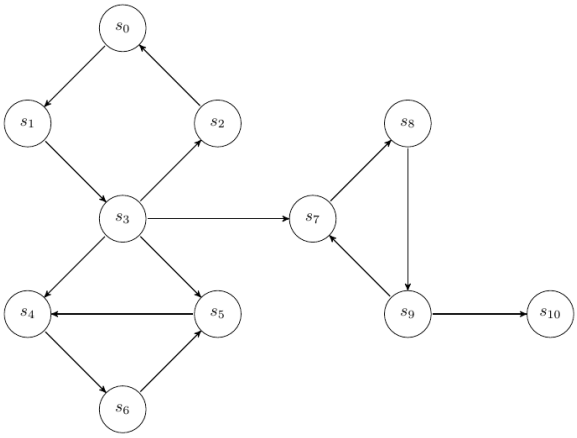
DS INFO N°5 - Corrigé

Exercice 1 (Questions - Applications directes du cours.).

1. Montrer que le graphe ci-dessous est biparti.



2. Redessiner et entourer (ou colorier) les composantes fortement connexes du graphe ci-dessous. Donner le résultat obtenu en empilant successivement les étiquettes des sommets terminés lors d'un parcours en profondeur, en partant du sommet s_3. On fera l'hypothèse que la liste des successeurs est triée par étiquettes croissantes des successeurs. On pourra détailler l'exécution du tri en profondeur récursif pour ne pas se tromper. S'agit-il d'un tri topologique ?

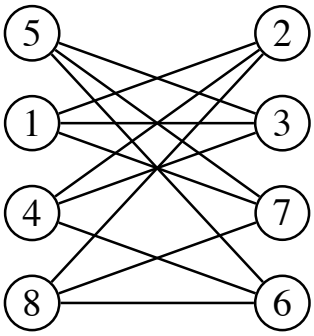


3. Montrer qu'un graphe non orienté ayant un nombre fini de sommets et dont tout sommet est de degré supérieur ou égal à 2 possède au moins un cycle.

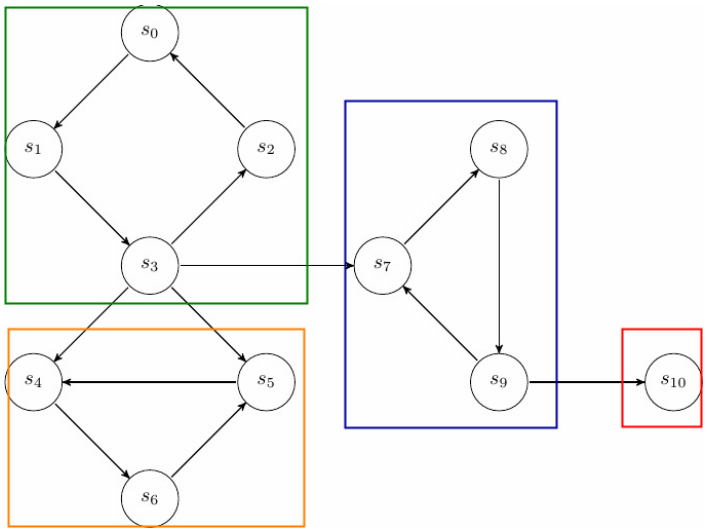
Corrigé de l'exercice 1.

[Retour à l'énoncé]

1. Le graphe peut être mis sous la forme suivante :



2. Les composantes fortement connexes sont encadrées en différentes couleurs :



Le parcours postfixe obtenu, en partant de s_3 et en traitant comme le demande l'énoncé la liste des successeurs par étiquettes croissantes, est $[3, 7, 8, 9, 10, 4, 6, 5, 2, 0, 1]$. Il ne s'agit pas d'un tri topologique car le graphe possède des cycles.

3. Soit $g = (V, E)$ un graphe non orienté fini dont tout sommet est de degré supérieur ou égal à 2. On choisit un sommet source $v_0$et on construit la suite de sommets $v_0, v_1, \ldots$ de la façon suivante :

— $v_i \notin \{v_1 v_2, \ldots, v_{i-1}\}$;

— v_{i-1} et v_i sont voisins.

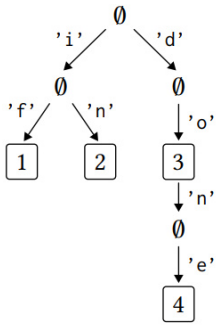
Le graphe ayant un nombre de sommets fini par hypothèse, une suite de ce type ne peut être infinie. Considérons le dernier sommet v_k d'une telle suite. Son degré est supérieur ou égal à 2 et il ne possède pas de voisins en dehors de $\{v_1, \ldots, v_{k-1}\}$ puisque c'est le dernier sommet que l'on a pu construire. Comme il est de degré 2 par hypothèse, il possède bien deux voisins distincts mais ceux-ci sont dans forcément dans la liste $\{v_1, \ldots, v_{k-1}\}$: le voisin v_{k-1} et un sommet $v_j \neq v_{k-1}$. On a donc trouvé un cycle, formé par la suite de sommets $(v_k, v_j, v_{j+1}, \ldots, v_{k-1}, v_k)$.

Ainsi, tout graphe fini dont les sommets sont de degré supérieur ou égal à 2 possède au moins un cycle.

L'exercice suivant présente une structure d'arbre permettant d'implémenter un dictionnaire dont les clés sont des chaînes de caractères.
Ces arbres sont appelés **arbres préfixes**, et plus connu en anglais sous le nom *trie*. [1]

1. Le mot *trie* vient du mot re**trie**val, rechercher, extraire.

Voici par exemple l'arbre représentant le dictionnaire associant des chaînes de caractères à des entiers et contenant les entrées suivantes : {`"if"` $\mapsto 1$, `"in"` $\mapsto 2$, `"do"` $\mapsto 3$, `"done"` $\mapsto 4$}
Dans ces arbres, chaque branche est étiquetée par un caractère. Un nœud contient une valeur si la séquence de lettres menant de la racine de l'arbre à ce nœud est une entrée dans le dictionnaire. Si cette séquence de caractères n'est pas une entrée, le nœud peut exister mais n'a pas de valeur associée, ce que l'on note $\emptyset$. Par exemple, dans l'arbre ci-dessus, la clé `"don"` n'est pas une entrée du dictionnaire. Le nœud se trouvant au bout du chemin associé à cette clé existe (car il a été crée lors de l'ajout de la clé `"done"`), mais est vide.



Chaque nœud est repéré de manière unique par son chemin depuis la racine, ce chemin étant représenté par le mot w formé par tous les caractères associés aux branches étiquetées successivement empruntées dans l'arbre pour arriver jusqu'à ce nœud. En particulier, la racine correspond au mot vide. Le nœud contient soit $\emptyset$ s'il n'y a pas de valeur associée à w, soit la valeur associée à w, ici encadrée.

Exercice 2 (Arbres préfixes - Exercice de programmation).
On propose le type OCaml suivant pour représenter de tels arbres :

```
type 'v trie =
 {
    mutable node_val : 'v option;
    branches : (char, 'v trie) Hashtbl.t
 };;
```

1. Expliquer en quelques phrases le type OCaml proposé.
2. Écrire une fonction `trie_add: 'v trie -> string -> 'v -> unit` qui insère une entrée dans notre dictionnaire implémenté avec un arbre préfixe. Si la clé était déjà présente, la fonction se contente de remplacer la valeur associée à la clé par la nouvelle valeur. On utilisera la fonction `Hashtbl.find_opt` et la fonction `Hashtbl.add`.
3. Écrire une fonction `trie_find_opt: 'v trie -> string -> 'v option` qui cherche une valeur associée à une clé dans notre dictionnaire et gère le cas où cette valeur n'est pas présente en renvoyant un type `option None`. On utilisera la fonction `Hashtbl.find_opt`.
4. Écrire une fonction `trie_remove: 'v trie -> string -> bool` qui cherche une valeur associée à une clé dans un arbre préfixe et supprime la valeur associée dans l'arbre. Si la clé n'a pas été trouvée, la fonction renvoie la valeur `false`.
5. Citer un algorithme où vous avez utilisé des arbres préfixes. Quel était le type des clés dans ce cas ?

Corrigé de l'exercice 2.

[Retour à l'énoncé]

1. La valeur associée à chaque nœud est implémentée par un type `option`, qui permet de gérer proprement le fait que cette valeur peut être vide dans le cas où le chemin menant au nœud ne correspond à aucune clé présente dans le dictionnaire.
   Chaque nœud peut avoir un nombre arbitraire de sous-arbres comme dans les arbres généraux, et l'accès à chacun de ces sous-arbres se fait par un arc étiqueté. Ainsi, cet ensemble de sous-arbre est représenté par un dictionnaire implémenté par une table de hachage, appelée `branches` : ce dictionnaire permet d'associer à chaque branche, représentée par son étiquette de type `char`, un sous-arbre de type `'v trie`.
2. On crée d'abord une fonction auxiliaire `aux_add` qui prend en entrée l'indice `i` du caractère courant. A chaque appel récursif de cette fonction, on lit le caractère `i` de la clé `k`. Si la branche associée au caractère `k.(i)` n'existe pas, on crée un sous-arbre vide `st` et on ajoute une branche étiquetée par `k.(i)` vers ce sous-arbre, puis on appelle récursivement la fonction, en incrémentant de 1 l'indice de lecture des caractères de la clé.. Sinon, si la branche existe déjà, on se contente d'appeler récursivement la fonction sur le sous arbre associé à cette branche, toujours en incrémentant de 1 l'indice de lecture des caractères de la clé.
   La descente récursive prend fin lorsque l'on a lu tous les caractères de la clé (`i = String.length k`). Dans ce cas, on a atteint le nœud destination - qu'il ait en fait déjà été présent dans l'arbre ou qu'il ait été construit à la volée par ce processus récursif - et on stocke la valeur `v` associée à la clé `k` dans ce nœud, avec un constructeur `Some` car il doit s'agir d'une valeur de type `option`.

```
# let trie_add t k v =
  let rec aux_add t k v i =
    if i = String.length k then
        t.node_val <- Some(v)
    else
      let subt = Hashtbl.find_opt t.branches k.[i] in
      match subt with
      |None -> let st = (trie_create () ) in
               (Hashtbl.add t.branches k.[i] st; aux_add st k v (i+1) )
      |Some(st) -> aux_add st k v (i+1)
  in
  aux_add t k v 0;;
val trie_add : 'a trie -> string -> 'a -> unit = <fun>
```

   Cette fonction auxiliaire est appelée en démarrant la lecture de la clé au caractère d'indice 0, donc avec `i` valant 0.
3. Cette fonction est très similaire et plus simple que la précédente. Elle navigue dans les branches selon le même principe, en lisant la clé caractère par caractère à chaque appel récursif, grâce à l'incrémentation de `i`.

```
# let trie_find_opt t k =
  let rec aux_find t k i =
    if (i = String.length k) then
      t.node_val
    else
      let subt = Hashtbl.find_opt t.branches k.[i] in
      match subt with
      | None -> None
      | Some(st) -> aux_find st k (i+1)
in
aux_find t k 0;;
val trie_find_opt : 'a trie -> string -> 'a option = <fun>
```

4. Pour supprimer un entrée dans l'arbre, on navigue dans l'arbre comme précédemment, au suivant les différentes branches du chemin indiqué par la clé, caractère après caractère. Si le chemin indiqué par la clé n'existe pas entièrement, on ne fait rien et on renvoie `false`. Sinon, la suppression se fait sur le nœud atteint à l'issue de ce parcours, après avoir lu tous les caractères. S'il n'y a plus aucune branche sous ce nœud, on réinitialise éventuellement la table de hachage des branches avec `Hashtbl.reset` (facultatif). Puis on regarde s'il y avait une valeur dans ce nœud. S'il n'y en avait pas, c'est que l'entrée n'était pas présente (même s'il y avait un chemin correspondant à cette clé, ce qui peut arriver) et on renvoie `false`. Sinon, on change la valeur de ce nœud pour le mettre à vide en utilisant le constructeur `None` et on renvoie la valeur `true` pour indiquer que l'on a bien effectuer une suppression d'entrée dans l'arbre.

```
# let trie_remove t k =
  let rec aux_remove t k i =
    if (i = String.length k) then
      (if (Hashtbl.length t.branches = 0) then
         Hashtbl.reset t.branches;
       match t.node_val with
       |None ->false
       |Some(_) -> (t.node_val <- None; true)
      )
    else
      let subt = Hashtbl.find_opt t.branches k.[i] in
      match subt with
      | None -> false
      | Some(st) -> aux_remove st k (i+1)
in
aux_remove t k 0;;
val trie_remove : 'a trie -> string -> bool = <fun>
```

5. Nous avons utilisé des arbres préfixe dans l'algorithme de compression/décompression de Huffman. Les clés étaient des chaînes de caractères formées uniquement des caractères `0` et `1`. En fait, la structure d'arbre préfixe peut être généralisée à tout type de clés pouvant être vu comme une suite de lettres, quelle que soit la nature de ces lettres. C'est le cas par exemple pour une liste. C'est aussi le cas d'un entier, si on voit ses bits comme formant un mot avec les lettres `0` et `1`. Nous l'avons d'ailleurs utilisé sous cette forme dans l'algorithme de Huffman.

Exercice 3 (Logique propositionnelle).

On définit le connecteur de Sheffer, ou d'incompatibilité, par $x_1 \diamond x_2 = \neg x_1 \vee \neg x_2$.

1. Construire la table de vérité du connecteur de Sheffer.
2. Exprimer ce connecteur en fonction de $\neg$ et $\wedge$.
3. Vérifier que $\neg x_1 \equiv x_1 \diamond x_1$.
4. En déduire une expression des connecteurs $\wedge$, $\vee$ et $\rightarrow$ en fonction du connecteur de Sheffer. Justifier en utilisant des équivalences avec les formules propositionnelles classiques.
5. Démontrer par induction sur les formules propositionnelles que l'ensemble de connecteurs $\mathcal{C} = \{\diamond\}$ est un système complet.
6. Application : soit $\mathcal{F}$ la formule propositionnelle $x_1 \vee (\neg x_2 \wedge x_3)$. Donner une forme logiquement équivalente de $\mathcal{F}$ utilisant uniquement le connecteur de Sheffer.

Corrigé de l'exercice 3.

[Retour à l'énoncé]

1. La table de vérité du connecteur de Sheffer est

x_1	x_2	$x_1 \diamond x_2$
F	F	V
F	V	V
V	F	V
V	V	F

2. D'après la table de vérité :

$$x_1 \diamond x_2 \equiv \neg (x_1 \wedge x_2)$$

3. En utilisant la question précédente et la loi de Morgan :

$$x_1 \diamond x_1 \equiv \neg x_1 \vee \neg x_1 = \neg x_1$$

4. Pour toutes formules propositionnelles $\mathcal{F}_1$ et $\mathcal{F}_2$,

$$\begin{aligned}
\mathcal{F}_1 \wedge \mathcal{F}_2 &\equiv \neg\neg (\mathcal{F}_1 \wedge \mathcal{F}_2) && \text{(double négation)}\\
&\equiv \neg (\mathcal{F}_1 \diamond \mathcal{F}_2) && \text{(question 2)}\\
&\equiv (\mathcal{F}_1 \diamond \mathcal{F}_2) \diamond (\mathcal{F}_1 \diamond \mathcal{F}_2) && \text{(question 3)}
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}_1 \vee \mathcal{F}_2 &\equiv \neg\neg (\mathcal{F}_1 \vee \mathcal{F}_2) && \text{(double négation)}\\
&\equiv \neg (\neg\mathcal{F}_1 \wedge \neg\mathcal{F}_2) && \text{(loi de Morgan)}\\
&\equiv \neg\mathcal{F}_1 \diamond \neg\mathcal{F}_2 && \text{(question 2)}\\
&\equiv (\mathcal{F}_1 \diamond \mathcal{F}_1) \diamond (\mathcal{F}_2 \diamond \mathcal{F}_2) && \text{(question 3)}
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}_1 \to \mathcal{F}_2 &\equiv \neg\mathcal{F}_1 \vee \mathcal{F}_2 && \text{(définition de } \to)\\
&\equiv (\neg\mathcal{F}_1 \diamond \neg\mathcal{F}_1) \diamond (\mathcal{F}_2 \diamond \mathcal{F}_2) && \text{(résultat juste au-dessus)}\\
&\equiv (\neg\neg\mathcal{F}_1) \diamond (\mathcal{F}_2 \diamond \mathcal{F}_2) && \text{(question 3)}\\
&\equiv \mathcal{F}_1 \diamond (\mathcal{F}_2 \diamond \mathcal{F}_2) && \text{(double négation)}
\end{aligned}$$

Il est possible d'obtenir d'autres résultats beaucoup plus longs lorsque l'on s'y prend mal. Par exemple, dans l'expression de $\vee$ où l'on utilise exclusivement la question 3 et le résultat pour $\wedge$, on obtient de façon tout aussi juste

$$\begin{aligned}
\mathcal{F}_1 \vee \mathcal{F}_2 \equiv& \neg (\neg\mathcal{F}_1 \wedge \neg\mathcal{F}_2) = \neg ((\mathcal{F}_1 \diamond \mathcal{F}_1) \wedge (\mathcal{F}_2 \diamond \mathcal{F}_2))\\
\equiv& \neg ((\mathcal{F}_1 \diamond \mathcal{F}_1) \diamond (\mathcal{F}_2 \diamond \mathcal{F}_2)) \diamond ((\mathcal{F}_1 \diamond \mathcal{F}_1) \diamond (\mathcal{F}_2 \diamond \mathcal{F}_2))\\
\mathcal{F}_1 \vee \mathcal{F}_2 \equiv& [((\mathcal{F}_1 \diamond \mathcal{F}_1) \diamond (\mathcal{F}_2 \diamond \mathcal{F}_2)) \diamond ((\mathcal{F}_1 \diamond \mathcal{F}_1) \diamond (\mathcal{F}_2 \diamond \mathcal{F}_2))]\\
&\diamond [((\mathcal{F}_1 \diamond \mathcal{F}_1) \diamond (\mathcal{F}_2 \diamond \mathcal{F}_2)) \diamond ((\mathcal{F}_1 \diamond \mathcal{F}_1) \diamond (\mathcal{F}_2 \diamond \mathcal{F}_2))]
\end{aligned}$$

5. Considérons le prédicat $\mathcal{P}$: « la formule $\mathcal{F}$ est équivalente à une formule où seul le connecteur $\diamond$ est utilisé.» Rappelons que l'ensemble $\{\neg, \vee, \wedge\}$ est un système complet de connecteurs. Soit $\mathcal{F}$ une formule propositionnelle.

— Si $\mathcal{F}$ est réduite à $\top, \bot$ ou une variable, elle n'utilise aucun connecteur donc elle vérifie $\mathcal{P}$.

— Si $\mathcal{F} = \neg\mathcal{F}_1$ avec $\mathcal{F}_1$ vérifiant $\mathcal{P}$, alors elle vérifie $\mathcal{P}$ aussi car $\mathcal{F} = \mathcal{F}_1 \diamond \mathcal{F}_1$ d'après la question 4 , valable également pour les formules propositionnelles.

— Si $\mathcal{F} = \mathcal{F}_1 \vee \mathcal{F}_2$ ou $\mathcal{F} = \mathcal{F}_1 \wedge \mathcal{F}_2$ avec $\mathcal{F}_1$ et $\mathcal{F}_2$ vérifiant $\mathcal{P}$, on conclut que $\mathcal{F}$ vérifie $\mathcal{P}$ d'après la question 4.

Par induction structurelle, le prédicat $\mathcal{P}$ est vérifié pour toute formule. Ainsi, l'ensemble $\{\diamond\}$ est un système complet de connecteurs.

6. Les négations sont transformées en fin de calcul :

$$\begin{aligned} x_1 \vee (\neg x_2 \wedge x_3) &\equiv \neg\left(\neg x_1 \wedge \neg\left(\neg x_2 \wedge x_3\right)\right) \quad \text{(loi de Morgan)} \\ &\equiv \neg x_1 \diamond \neg\left(\neg x_2 \wedge x_3\right) \quad \text{(question 2)} \\ &\equiv \neg x_1 \diamond \left(\neg x_2 \diamond x_3\right) \quad \text{(question 2)} \\ &\equiv \left(x_1 \diamond x_1\right) \diamond \left(\left(x_2 \diamond x_2\right) \diamond x_3\right) \quad \text{(question 3)} \end{aligned}$$

Exercice 4 (Modélisation).

Un voyageur perdu dans le désert arrive à une bifurcation à partir de laquelle sa piste se sépare en deux. Chaque piste peut soit mener à une oasis, soit se perdre dans un désert profond. Chaque piste est gardée par un sphinx. Les données du problème sont les suivantes :

A. le sphinx de droite dit : " Une au moins des 2 pistes conduit à une oasis "

B. le sphinx de gauche dit : " La piste de droite se perd dans le désert "

C. soit les deux sphinx disent la vérité, soit ils mentent tous les deux

Le voyageur aimerait bien savoir s'il y a une oasis au bout d'un des deux chemins et si oui quelle direction prendre.

1. Introduire deux variables propositionnelles pour modéliser le problème (expliciter ce qu'elles représentent) et traduire les trois données en formules propositionnelles ;

2. Par la méthode de votre choix, résoudre l'énigme en justifiant votre réponse.

Corrigé de l'exercice 4.

[Retour à l'énoncé]

On introduit deux variables propositionnelles qui représentent le fait qu'il y a une oasis à gauche (variable g), et à droite (variable d).

1. Les données se traduisent par les formules propositionnelles suivantes :

A. $g \vee d$

B. $\neg d$

C. $(g \vee d) \Leftrightarrow \neg d$

2. On regarde les interprétations dans lesquelles la troisième formule est vraie

g	d	$g \vee d$	$\neg d$	$(g \vee d) \Leftrightarrow \neg d$
V	V	V	F	F
V	F	V	V	V
F	V	V	F	F
F	F	F	V	F

La seule situation possible est que l'oasis est au bout de la route de gauche.

Problème - Listes triables par pile

Dans cet exercice on s'interdit d'utiliser les traits impératifs du langage OCaml (références, tableaux, champs mutables, etc.).

On représente en OCaml une permutation σ de $[\![0, n-1]\!]$ par la liste d'entier $[\sigma_0; \sigma_1; \ldots; \sigma_{n-1}]$.

Un arbre binaire étiqueté est soit un arbre vide, soit un nœud formé d'un sous-arbre gauche, d'une étiquette et d'un sous-arbre droit.

On représente un arbre binaire non étiqueté par un arbre binaire étiqueté en ignorant simplement les étiquettes. On étiquette un arbre binaire non étiqueté à n nœuds par $[\![0; n-1]\!]$ en suivant l'ordre infixe de son parcours en profondeur. La permutation associée à cet arbre est donnée par le parcours en profondeur par ordre préfixe. La figure 1 propose un exemple (on ne dessine pas les arbres vides).

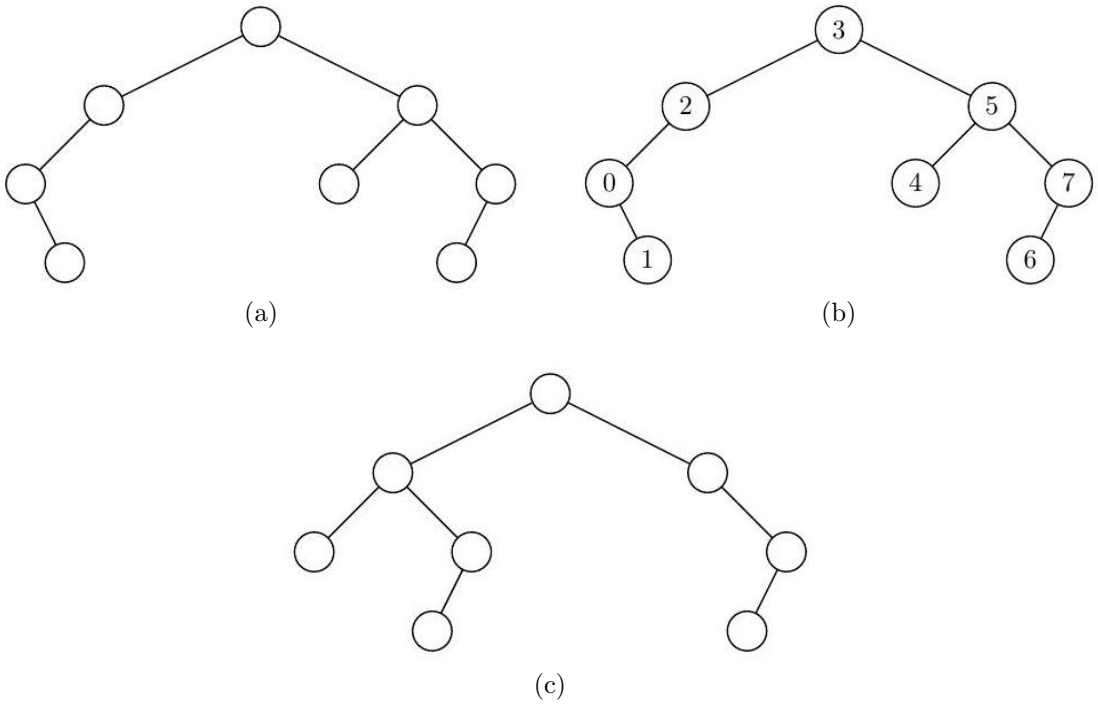


FIGURE 1 – (a) un arbre binaire non étiqueté ; (b) son étiquetage en suivant un ordre infixe, la permutation associée est `[3; 2; 0; 1; 5; 4; 7; 6]` ; (c) un autre arbre binaire non étiqueté

1. Définir un type OCaml `arbre` permettant de représenter un arbre binaire étiqueté par des entiers.
2. Étiqueter l'arbre (c) de la figure 1 et donner la permutation associée.
3. Écrire une fonction `parcours_prefixe : arbre -> int list` qui renvoie la liste des étiquettes d'un arbre dans l'ordre préfixe de son parcours en profondeur. On pourra utiliser l'opérateur @ et on ne cherchera pas nécessairement à proposer une solution linéaire en la taille de l'arbre.
4. Écrire une fonction `etiquette : arbre -> arbre` qui prend en paramètre un arbre dont on ignore les étiquettes et qui renvoie un arbre identique mais étiqueté par les entiers de $[\![0, n-1]\!]$ en suivant l'ordre infixe d'un parcours en profondeur.

Une permutation σ de $[\![0, n-1]\!]$ est triable avec une pile s'il est possible de trier par ordre décroissant la liste $[\sigma_0; \sigma_1; \ldots; \sigma_{n-1}]$ en utilisant uniquement une structure de pile comme espace de stockage interne. On utilise une liste résultat pour accumuler le résultat du tri au cours de l'algorithme.
L'algorithme est le suivant, énoncé ici dans un style impératif :

- Initialiser une pile vide ;
- Pour chaque élément en entrée :
  - Tant que l'élément est plus grand que le sommet de la pile, dépiler le sommet de la pile vers la liste résultat ;
  - Empiler l'élément en entrée dans la pile ;
- Dépiler tous les éléments restant dans la pile vers la liste résultat

Par exemple, pour la permutation `[3; 2; 0; 1; 5; 4; 7; 6]`, on empile $3, 2, 0$, on dépile 0, on empile 1, on dépile $1, 2, 3$, on empile $5, 4$, on dépile $4, 5$, on empile $7, 6$, on dépile $6, 7$.
On obtient la liste triée `[7; 6; 5; 4; 3; 2; 1; 0]` en supposant avoir ajouté en sortie les éléments dans une liste, en insérant à chaque fois en tête de liste comme cela est le cas naturellement en OCaml. On admet qu'une permutation est triable par pile si et seulement cet algorithme permet de la trier correctement.
On remarquera que la pile intermédiaire est toujours triée par ordre croissant (de la tête à la queue).

5. Dérouler l'exécution de cet algorithme sur la permutation associée à l'arbre (c) de la figure 1 et vérifier qu'elle est bien triable par pile. On représentera chaque étape de l'algorithme sur une ligne et on décrira, sur chaque ligne : l'état courant de la liste donnée en argument, l'état de la pile intermédiaire et l'état de la liste servant à accumuler le résultat.
6. Écrire une fonction `trier : int list -> int list` qui implémente cet algorithme dans un style fonctionnel. Par exemple, `trier [3; 2; 0; 1; 5; 4; 7; 6]` doit renvoyer la liste `[7; 6; 5; 4 ; 3 ; 2 ; 1 ; 0]`. On utilisera directement une liste pour implémenter une pile.
7. Montrer que s'il existe $0 \leq i < j < k \leq n-1$ tels que $\sigma_k < \sigma_i < \sigma_j$, alors σ n'est pas triable par une pile.
8. On se propose de montrer que les permutations de $[\![0, n-1]\!]$ triables par une pile sont en bijection avec les arbres binaires non étiquetés à n nœuds.
   (a) Montrer que la permutation associée à un arbre binaire est triable par pile. On pourra remarquer le lien entre le parcours préfixe et l'opération empiler d'une part et le parcours infixe et l'opération dépiler d'autre part.
   (b) Montrer qu'une permutation triable par pile est une permutation associée à un arbre binaire. Indication : on peut prendre σ_0 comme racine, puis procéder récursivement avec les $\sigma_0 - 1$ éléments pour construire le fils gauche et avec le reste pour le fils droit.
   (c) On note AB_n l'ensemble des arbres binaires non étiquetés à n nœuds et P_n l'ensemble des permutations de $[\![0, n-1]\!]$. On note enfin Ptp_n l'ensemble des permutations de $[\![0, n-1]\!]$ triables par pile. Nous avons étudié dans les questions précédentes une application $f : AB_n \to P_n$.
   Quel est l'ensemble image I de cette fonction ?
   Que reste-t-il à montrer sur la fonction $\tilde{f} : AB_n \to I$ pour prouver que les

permutations de $[\![0, n-1]\!]$ triables par une pile sont en bijection avec les arbres binaires non étiquetés à n nœuds ?
Proposez une démonstration pour cette dernière étape.

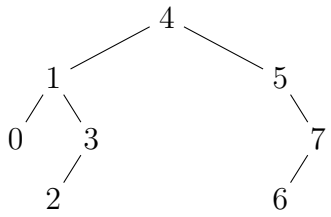
Corrigé de l'exercice 5.

[Retour à l'énoncé]

1. On propose le type OCaml suivant :

```
# type arbre =
  | V
  | N of arbre * int * arbre;;
```

2. Voici l'arbre (c) étiqueté en suivant un étiquetage infixe :



La permutation associée s'obtient en lisant ces étiquettes dans l'ordre préfixe : `[4; 1; 0; 3; 2; 5; 7; 6]`

3. Voici une proposition de code pour la fonction `parcours_prefixe : arbre -> int list` qui énumère les étiquettes d'un arbre par ordre préfixe :

```
# let rec parcours_prefixe a =
  match a with
  | V -> []
  | N(g,x,d) -> x::(parcours_prefixe g)@(parcours_prefixe d);;
val parcours_prefixe : arbre -> int list = <fun>
```

On affiche considère d'abord la valeur au nœud courant, puis on traite récursivement les nœuds du sous-arbre gauche et enfin ceux du sous-arbre droit, ce qui apparaît clairement ici dans la manière dont les appels récursifs sont concaténés.

4. Voici une proposition de code pour la fonction `etiquette : arbre -> arbre` qui étiquette tous les nœuds d'un arbre par leur numéro d'ordre infixe. On a d'abord déclaré une fonction auxiliaire, qui permet de d'attribuer récursivement une étiquette à chaque noeud en numérotant selon le parcours infixe.

```
# let rec aux_etiquette a i =
    match a with
    | V -> (V, i)
    | N(g, _, d) -> let (ng, j) = aux_etiquette g i
                    in let (nd, j2) = aux_etiquette d (j+1) in
                       (N(ng, j, nd), j2);;
val aux_etiquette : arbre -> int -> arbre * int = <fun>
```

Cette fonction `aux_etiquette` prend en entrée un argument de plus de type entier, appelé `i`, qui représente la prochaine étiquette à attribuer. Cette fonction reconstruit l'arbre de manière récursive en le re-étiquetant grâce à cet entier i qui permet compter les étiquettes attribuées et de redémarrer chaque étiquetage à partir de la prochaine étiquette disponible.

— Un sous-arbre vide correspond au cas d'arrêt : la récursivité s'arrête et on revoie un couple constitué d'un arbre vide et du numéro de la prochaine étiquette à poser.

— Pour un nœud quelconque :

  — On descend d'abord dans le sous-arbre gauche pour re-étiqueter. Cet appel récursif nous renvoie l'arbre gauche ré-étiqueté et la prochaine étiquette à poser `j`.

  — Puis on appelle récursivement l'algorithme dans le sous-arbre droit mais en démarrant à l'étiquette `j+1`, car l'étiquette `j` est réservée au nœud courant qui est celui du milieu dans le parcours infixe. On récupère le numéro `j2` de la prochaine étiquette à apposer après avoir re-étiqueté tout le sous arbre droit.

  — Enfin, on renvoie le couple formé du nœud courant `N(ng, j, nd)` formé par ses deux sous-arbres ré-étiqueté `ng` et `nd` et par son étiquette `j`, et le compteur d'étiquette mis à jour à `j2`.

Pour obtenir la fonction `etiquette`, on appelle cette fonction auxiliaire en initialisant le compteur d'étiquette `i` à 0. On renvoie simplement l'arbre `t` re-étiqueté et reconstruit par la remontée récursive, et on ignore la deuxième composante du couple renvoyé, qui correspond au nombre d'étiquettes aposées.

```
# let etiquette a =
  let (t, _) = aux_etiquette a 0 in t;;
val etiquette : arbre -> arbre = <fun>
#
```

On a testé notre fonction sur le cas test fourni par l'énoncé.

```
# let test1_desetiquete = N(N(N(V, 7, N(V, 3, V)),4, V), 8, N(N(V,5,V),6, N(N(V,1,V),3, V)));;
val test1_desetiquete : arbre =
  N (N (N (V, 7, N (V, 3, V)), 4, V), 8,
   N (N (V, 5, V), 6, N (N (V, 1, V), 3, V)))
# etiquette test1_desetiquete;;
- : arbre =
N (N (N (V, 0, N (V, 1, V)), 2, V), 3,
 N (N (V, 4, V), 5, N (N (V, 6, V), 7, V)))
```

5. Voici le déroulé de l'exécution sur la permutation associée à l'arbre (c). On réalise un suivi des trois listes/piles utilisées : la liste initiale fournie en entrée de l'algorithme, la pile intermédiaire, et la liste servant à accumuler la liste triée résultat :

Liste	Pile	Accumulation résultat
4 1 0 3 2 5 7 6	vide	vide
1 0 3 2 5 7 6	4	vide
0 3 2 5 7 6	1 4	vide
3 2 5 7 6	0 1 4	vide
3 2 5 7 6	1 4	0
3 2 5 7 6	4	1 0
2 5 7 6	3 4	1 0
5 7 6	2 3 4	1 0
5 7 6	3 4	2 1 0
5 7 6	4	3 2 1 0
5 7 6	vide	4 3 2 1 0
7 6	5	4 3 2 1 0
7 6	vide	5 4 3 2 1 0
6	7	5 4 3 2 1 0
vide	6 7	5 4 3 2 1 0
vide	7	6 5 4 3 2 1 0
vide	vide	7 6 5 4 3 2 1 0

6. Voici une proposition de code pour la fonction `trier :int list -> int list` qui implémente cet algorithme :

```
# let trier l =
  let rec aux l pile res =
    match (l, pile) with
    | ([], []) -> res
    | ([], h::t) -> aux l t (h::res)
    | (h::t, []) -> aux t [h] res
    | (h::t, h1::t1) when h < h1 -> aux t (h::pile) res
    | (_, h1::t1)  -> aux l t1 (h1::res)
  in aux l [] [];;
val trier : 'a list -> 'a list = <fun>
# trier [3;2;0;1;5;4;7;6];;
- : int list = [7; 6; 5; 4; 3; 2; 1; 0]
# trier [4;1;0;3;2;5;7;6];;
- : int list = [7; 6; 5; 4; 3; 2; 1; 0]
```

que l'on a testé sur les deux exemples traités dans l'énoncé.

On créer une fonction auxiliaire récursive faisant apparaître deux paramètres supplémentaires :

- un paramètre `pile` correspondant à l'état courant de la pile intermédiaire (liste OCaml = pile immuable) décrite dans l'algorithme
- un paramètre `res` correspond à une site servant d'accumulateur comme décrit dans l'algorithme.
- Si la liste restant à trier `l` est vide et la pile également (cas 1), alors l'algorithme est terminé, et on renvoie la liste `res` qui accumulé les éléments dans l'ordre décroissant.
- Si la liste restant à trier `l` est vide (plus rien à trier), mais que la pile intermédiaire n'est pas vide, on purge la pile intermédiaire en la dépilant entièrement par récursivement dans la liste `res` (cas 2).
- S'il reste des éléments dans la liste à trier `l`, on considère la tête `h` de la liste `l` à trier :
  - Si la `pile` est vide, on empile `h` dans la `pile` (cas 3)
  - Sinon, on compare l'élément `h` à la tête `h1` de la `pile` intermédiaire :
    - Tant que `h` $\geq$ `h1`, on dépile les éléments de la `pile` pour les mettre dans la liste de sortie `res` (cas 5)
    - Dès que `h < h1`, on empile h sur la `pile` (cas 4)

Enfin, on appelle cette fonction auxiliaire en initialisant la `pile` et la liste d'accumulation du résultat `res` comme des listes vides.

7. Supposons qu'il existe $0 \leq i < j < k \leq n-1$ tels que $\sigma_k < \sigma_i < \sigma_j$.

Lorsque l'on traite σ_j, on peut avoir 2 cas :

σ_i **est encore dans la pile.** Dans ce cas, comme la pile intermédiaire est toujours triée dans l'ordre croissant (admis dans l'énoncé), σ_i sortira de la pile car il est plus petit que σ_j et sera ajouté à la liste résultat. Il apparaîtra donc toujours à droite de σ_j dans la liste résultat.

σ_i **est déjà dans la liste résultat.** Il apparaîtra donc toujours à droite de σ_j dans la liste résultat.

Lorsque l'on traite σ_k, on peut avoir 2 cas :

Soit σ_j est déjà sorti de la pile et est dans la liste résultat. Dans ce cas, σ_k, lorsqu'il sortira de la pile, sera à gauche de σ_j dans la liste résultat, qui aura cette forme :

$$[\ldots \sigma_k; \ldots; \sigma_j; \ldots \sigma_i; \ldots]$$

ce qui ne correspond pas à une liste triée puisque par hypothèse $\sigma_k < \sigma_i < \sigma_j$

Soit σ_j est encore dans la pile. Dans ce cas, σ_k, sortira de la pile avant σ_j et on aura une liste résultat avec :

$$[\ldots \sigma_j; \ldots; \sigma_k; \ldots \sigma_i; \ldots]$$

ce qui ne correspond pas à une liste triée puisque par hypothèse $\sigma_k < \sigma_i < \sigma_j$.

L'algorithme ne fonctionne donc pas sur une telle permutation.

(a) La permutation associée aux étiquettes infixe d'un arbre binaire est telle que cet arbre binaire vérifie la propriété d'arbre binaire de recherche :

—toutes les étiquettes du sous-arbre gauche d'un nœud sont inférieures strictement à l'étiquette de ce nœud, et

—toutes les étiquettes de son sous-arbre droit sont supérieures strictement.

Montrons par induction structurelle la propriété suivante, où n est le nombre de nœuds de l'arbre binaire :

$\mathcal{P}(t)$: Si t est un arbre binaire étiqueté par son parcours infixe et σ la permutation associée, alors σ est triable par pile.

Cas de base : Si $t = V$ est l'arbre vide, c'est évident

Si $t = N(V, \sigma_0, V)$, c'est également évident

Vérification de l'hypothèse d'induction : Soit $t \neq V$ un arbre étiqueté par sa numérotation infixe et supposons la propriété vraie pour tout sous arbre de t.

Comme t est non vide, il s'écrit : $t = N(g, \sigma_0, d)$.

t est un ABR car il est étiqueté par sa numérotation infixe.

—Le parcours préfixe comme par la racine σ_0 et la met dans la pile, qui est vide.

—Lorsque le parcours préfixe (celui qui sert à écrire la permutation) nous amène dans un sous-arbre gauche g on ne rencontre que des étiquettes plus petites $\sigma_i < \sigma_0$

$$[\ldots \sigma_i \ldots \sigma_0]$$

Il peut y aura des empilements et dépilements intermédiaires, mais ce dont on est certains, c'est que σ_0 restera au fond de la pile qui il est plus grand que toutes les étiquettes de ses fils gauches. On applique dans l'hypothèse d'induction sur le sous-arbre gauche g et la permutation qui lui est associée : l'algorithme va travailler dans la pile, au dessus de σ_0 et stocker les étiquettes du sous arbre gauche triées dans la liste résultat.

—Lorsque le parcours préfixe passe au sous-arbre droit, il ne reste plus que σ_0 dans la pile. Comme l'arbre est ABR, dès l'analyse du premier nœud du sous-arbre droit, on rencontre une étiquette plus grande que σ_0, qui va donc être dépilé et mis dans la liste résultat.

— Puis l'hypothèse d'induction s'applique au sous arbre droit et à la permutation associée, qui vont travailler dans la pile. L'algorithme va trier ce sous-arbre droit, par hypothèse d'induction, et stocker les étiquettes du sous arbre triées dans la liste résultat, à gauche de σ_0 et des étiquettes triées du sous-arbre gauche qui avaient déjà été triées et rangées.

Finalement, la liste résultat correspond bien à un tri de la permutation associée à l'arbre de départ :

$$[\ \underbrace{\ldots}_{\text{étiquettes du sous arbre droit } (>\sigma_0) \text{ triées}}\ \sigma_0;\ \underbrace{\ldots}_{\text{étiquettes du sous arbre gauche } (<\sigma_0) \text{ triées}}\ ]$$

Conclusion : Nous avons montré, par induction, que toute permutation de $[\![0, n-1]\!]$ correspondant à la lecture préfixe d'un arbre binaire dont les sommets sont étiquetés dans l'ordre infixe est triable par pile.

(b) Nous allons montrer par récurrence forte :

$\mathcal{P}(n)$: Soit $[\sigma_0; \ldots \sigma_{n-1}]$ une permutation de $[\![0, n-1]\!]$ triable par pile. Alors cette permutation est associée à un arbre binaire étiqueté par sa numérotation infixe.

Initialisation. Pour $n = 0$, on a une permutation d'un ensemble vide, qui est triable par pile et est bien associée à... un arbre vide (donc étiqueté par sa numérotation infixe vide) Pour $n = 1$ c'est trivialement vrai
Pour $n = 2$, deux cas peuvent se produire :

— $[0; 1]$ qui est associé à l'arbre binaire étiqueté suivant :

— Ou $[1; 0]$ qui est associé à l'arbre binaire étiqueté suivant :

Vérification de l'hypothèse de récurrence. Supposons la propriété vraie au rang k pour tout $k < n$. Et soit $\sigma = [\sigma_0; \ldots; \sigma_{n-1}]$ une permutation de $[\![0, n-1]\!]$ triable par pile.

Si $\sigma_0 < \sigma_j$ pour tout $1 \leq j \leq n-1$: alors on applique l'hypothèse de récurrence à la permutation $[\sigma_1; \ldots \sigma_{n-1}]$ de taille $n-1 < n$: elle correspond à un arbre binaire d étiqueté par sa numérotation infixe. L'arbre $N(V, \sigma_0, d)$ correspond bien à un arbre binaire étiqueté par sa numérotation infixe valide, car d est un arbre valide, étiqueté par sa numérotation infixe, et σ_0 est plus petit que toutes les étiquettes $[\sigma_1; \ldots \sigma_{n-1}]$ du sous-arbre droit. Comme le sous-arbre gauche est vide dans ce cas, il n'y pas de problème.

Si $\sigma_0 > \sigma_j$ pour tout $1 \leq j \leq n-1$: alors on applique l'hypothèse de récurrence à la permutation $[\sigma_1; \ldots \sigma_{n-1}]$ de taille $n-1 < n$: elle correspond à un arbre binaire g étiqueté par sa numérotation infixe. L'arbre $N(g, \sigma_0, V)$ correspond bien à un arbre binaire étiqueté par sa numérotation infixe valide, car g est un arbre valide, étiqueté par sa numérotation infixe, et σ_0 est plus grand que toutes les étiquettes $[\sigma_1; \ldots \sigma_{n-1}]$ du sous-arbre gauche. Comme le sous-arbre droit est vide dans ce cas, il n'y pas de problème.

Sinon, si la permutation est triable par pile, d'après la question 6, il existe forcément un indice j vérifiant $0 < j < n-1$ tel que : $\sigma = [\sigma_0; \underbrace{\sigma_1; \ldots; \sigma_{j-1}}_{<\sigma_0}; \underbrace{\sigma_j; \ldots; \sigma_{n-1}}_{>\sigma_0}]$

On applique alors l'hypothèse de récurrence à la permutation $[\sigma_1; \dots \sigma_{j-1}]$ de taille $j-1 < n$: elle correspond à un arbre binaire g étiqueté par sa numérotation infixe.

On applique également l'hypothèse de récurrence à la permutation $[\sigma_j; \dots \sigma_{n-1}]$ de taille $n-j < n$: elle correspond à un arbre binaire d étiqueté par sa numérotation infixe.

L'arbre construit à partir de ces deux sous-arbres de la manière suivante $N(g, \sigma_0, d)$ correspond bien à un arbre binaire étiqueté par sa numérotation infixe valide, car g et d le sont et σ_0 est plus grand que toutes les étiquettes $[\sigma_1; \dots \sigma_{j-1}]$ du sous arbre gauche, et plus petit que toutes les étiquettes $[\sigma_j; \dots \sigma_{n-1}]$ du sous-arbre droit.

Conclusion. Par principe de récurrence forte, on a démontré que tout permutation de $[\![0, n-1]\!]$ triable par pile est associée à un arbre binaire étiqueté par sa numérotation infixe.

La fonction f correspond à la permutation obtenue en accumulant selon un parcours préfixe les étiquettes d'un arbre étiqueté selon un parcours infixe :

$$\begin{array}{rccl} f: & AB_n & \to & P_n \\ & t & \mapsto & [\![\sigma_0, \dots, \sigma_{n-1}]\!] \end{array}$$

où $[\![\sigma_0, \dots, \sigma_{n-1}]\!]$ est le parcours préfixe de l'arbre t, qui a été étiqueté par numérotation infixe.

Nous avons montré à la question 8 (a) que l'ensemble image de cette fonction est $T = Ptp_n$.

La question 8 (b) nous permet d'affirmer que :

$$\begin{array}{rccl} \tilde{f}: & AB_n & \to & I = Ptp_n \\ & t & \mapsto & [\![\sigma_0, \dots, \sigma_{n-1}]\!] \end{array}$$

est surjective.

Pour montrer que f est une bijection, il reste donc à démontrer que f est injective. Pour cela, il suffit de démontrer que deux arbres différents, c'est-à-dire n'ayant pas la même structure, donneront deux permutations différentes. Nous allons le montrer par récurrence forte sur le nombre de nœuds de l'arbre :
$\mathcal{P}(n)$: Si $t, t' \in AB_n$ et $t \neq t'$, alors $f(t) \neq f(t')$

Initialisation : Pour $n = 0$ ou $n = 1$, il n'existe pas deux arbres différents structurellement, donc la propriété, s'appliquant sur un ensemble vide, est trivialement vraie.

Pour $n = 2$, il y a deux arbres non étiquetés différents possibles :

—l'arbre binaire étiqueté infixe suivant, qui correspond à la permutation (lecture préfixe) $[0; 1]$

—l'arbre binaire étiqueté infixe suivant, qui correspond à la permutation (lecture préfixe) $[1; 0]$

La propriété est donc bien vérifiée car les deux seuls arbres ont pour image deux permutations triables par pile différentes.

Vérification de l'hypothèse de récurrence : Soit $t = N(\ell, r) \neq t' = N(\ell', r')$ deux arbres non étiquetés à n nœuds et supposons la propriété $\mathcal{P}(k)$ vraie pour tout $k < n$. Nous allons démontrer la validité de l'hypothèse de récurrence en procédant par l'absurde. Supposons donc que ces deux arbres t et t' différents aboutissent à la même permutation $[\![\sigma_0, \ldots, \sigma_{n-1}]\!]$ par $\tilde{f}$. σ_0 correspond à l'étiquette de la racine de ces deux arbres car $\tilde{f}$ correspond à une lecture préfixe. Comme pour la précédente démonstration, on distingue 3 cas :

Si $\sigma_0 > \sigma_j$ pour tout $1 \leq j \leq n-1$, alors les sous-arbres droite r et r' de t et t' sont vides et on applique l'hypothèse de récurrence aux deux sous-arbres gauches ℓ et ℓ' qui ont au plus $n-1$ nœuds. Comme $t = N(\ell, E) \neq t' = N(\ell', E)$, nécessairement, $\ell \neq \ell'$. Or, si t et t' sont associés à la même permutation $[\![\sigma_0, \ldots, \sigma_{n-1}]\!]$, ces deux sous-arbres sont également associés à la même permutation $[\![\sigma_1, \ldots, \sigma_{n-1}]\!]$. On a donc une contradiction, et on en déduit que t et t' ne peuvent pas être associés à la même permutation dans ce cas.

Si $\sigma_0 < \sigma_j$ pour tout $1 \leq j \leq n-1$, on fait le même raisonnement en disant que ce sont cette fois les sous-arbres gauches qui sont vides.

Sinon, il existe forcément un indice j vérifiant $0 < j < n-1$ tel que :

$$\sigma = [\sigma_0; \underbrace{\sigma_1; \ldots; \sigma_{j-1}}_{<\sigma_0}; \underbrace{\sigma_j; \ldots; \sigma_{n-1}}_{>\sigma_0}]$$

La même permutation $[\![\sigma_1, \ldots, \sigma_{j-1}]\!]$ est associée au sous-arbres gauches ℓ et ℓ' de t et de t' et la même permutation $[\![\sigma_j, \ldots, \sigma_{n-1}]\!]$ est associées à aux sous-arbres droit r et r' de t et de t' . Par hypothèse de récurrence, on en déduit forcément que $\ell = \ell'$ et $r = r'$, et donc, finalement, que $t = t'$, contradiction. Donc t et t' ne peuvent pas être associés à la même permutation dans ce cas non plus.

On a donc montré par l'absurde, avec disjonction des 3 cas, la validité de l'hypothèse de récurrence.

Conclusion : Par principe de récurrence, on a montré la propriété $\mathcal{P}(n)$, c'est-à-dire l'injectivité de $\tilde{f}$

On a donc finalement montré que les arbres binaires non étiquetés à n nœuds et les permutations triables de $[\![\sigma_0, \ldots, \sigma_{n-1}]\!]$ sont en bijection par $\tilde{f}$.