❑ Exercice 1 : *Nombre d'arêtes*

1. Rappeler la définition d'un arbre binaire.

   Voir cours

2. Soit a un arbre binaire à n noeuds ($n \geqslant 1$), montrer que a possède $n-1$ arêtes.

   Preuve par récurrence forte sur la taille de l'arbre (k noeuds dans le sous arbre gauche et $n-k-1$ noeuds dans le sous arbre droit), il faut distignuer le cas où l'un des sous arbre est vide.

3. On rappelle l'implémentation des arbres en OCaml utilisée en cours :

```
type ab =
 |Vide
 |Noeud of ab *  int * ab;;
```

   En utilisant cette implémentation, écrire une fonction `nb_aretes` de signature `ab -> int` et qui renvoie le nombre d'arêtes d'un arbre binaire

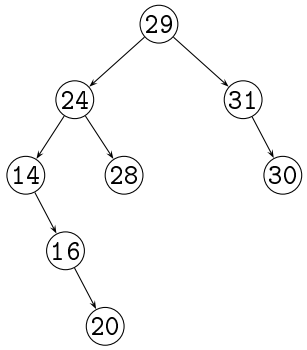
   En utilisant le résultat de la question précédente, il suffit de calculer la taille de l'arbre, on traite le cas de l'arbre vide avec un `failwith` :

```
let rec taille ab =
  match ab with
  | Vide -> 0
  | Noeud (g, r, d) -> 1 + taille g + taille d

let nb_aretes ab =
  match ab with
  | Vide -> failwith "L'arbre est vide"
  | a -> taille a - 1
```

❑ Exercice 2 : *Parcours d'arbre binaires*

1. Donner l'ordre des noeuds lors des parcours prefixe, infixe et postfixe de l'arbre suivant :

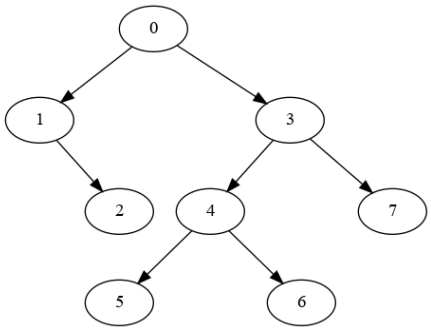


   — préfixe : `[ 29; 24; 14; 16; 20; 28; 31; 30 ]`
   — infixe : `[ 14; 16; 20; 24; 28; 29; 31; 30 ]`
   — postfixe : `[ 20; 16; 14; 28; 24; 30; 31; 29 ]`

2. On cherche à présent à reconstruire un arbre en connaissant un ou plusieurs de ses parcours. Montrer sur un exemple que deux arbres ayant les mêmes parcours prefixe et postfixe peuvent être différents.

   On peut prendre un peigne

3. Montrer (en le construisant) qu'un seul arbre a pour préfixe [0 ; 1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7] et pour parcours infixe [1 ; 2 ; 0 ; 5 ; 4 ; 6 ; 3 ; 7]



4. Comment savoir si deux listes de valeurs correspondent aux parcours prefixe et infixe d'un arbre binaire ?

   On note $p = [p_0, \dots p_{n-1}]$ et $i = [i_0, \dots i_{n-1}]$ les listes représentant les parcours prefixe et infixe. Comme ce sont les parcours du même arbre, i est une permutation de p. On extrait deux sous listes i_g et i_d de i, i_g sont les éléments situés à gauche de p_0 et i_d ceux situés à droite. On extrait de même deux sous listes de p, p_g contient les $|i_g|$ éléments situés après p_0 et p_d le reste de la liste. La liste p_g doit être une permutation de i_g et p_d doit être une permutation de i_d. Et cette propriété doit rester vraie en reproduisant ce processus récursivement sur les deux listes extraites.

5. Ecrire une fonction qui prend en argument deux listes (le parcours prefixe et le parcours infixe) et qui renvoie l'arbre binaire correspondant. On supposera que les étiquettes de l'arbre sont des entiers tous différents. 🛟 On pourra commencer par écrire :
   - Une fonction `separe_valeur` de signature `int -> int list -> int list * int list` qui prend en argument un entier, une liste contenant cet entier et renvoie un couple de liste : les élément situés avant (resp après) cette valeur
   - Une fonction `separe_nb` de signature `int -> int list -> int list * int list` qui prend en argument un entier `n` et une liste et renvoie un couple de listes : les `n` éléments situés après le premier puis le reste de la liste.

❑ **Exercice 3** : *Expression bien parenthésée*

On considère dans cet exercice un parenthésage avec les couples (), {} et []. On dira qu'une expression est bien parenthésée si chaque symbole ouvrant correspond à un symbole fermant et si l'expression contenue à l'intérieur est elle-même bien parenthésée.

1. Les expressions suivantes sont-elles bien parenthésée ?
   - $3 + [5 - 4 \div (3 + 2)]\} + 10$
   - $\{(3 + 2) \times 5$
   - $5) - 4 \times 2($
   - $[(3 + 2) \times (5 - 3)]$
2. Rappeler les fonctions de l'interface d'une pile.
3. Ecrire une fonction `bien_parenthesee` de signature `str -> bool` qui renvoie `true` lorsque la chaine de caractère donnée en argument est une expression bien parenthésée
   🛟 on utilisera le module `Stack` de OCaml afin de disposer d'une structure de pile *mutable*. On rappelle ci-dessous les fonctions principales de ce module :
   - `Stack.create` de signature `() -> 'a t` qui crée une pile vide d'éléments de type `'a`. Par exemple `let mapile = Stack.create ()`
   - `Stack.push` de signature `'a 'a t -> ()` qui empile un élément. Par exemple `Stack.push 5 mapile` empile l'entier 5 sur `mapile` (le type option 'a est alors le type `int`).

— `Stack.pop` de signature `'a t -> 'a` qui renvoie l'élément situé au sommet de la pile en le dépilant.

❑ Exercice 4

1. Rappeler la définition du type abstrait *file* et donner les fonctions de son interface.
2. On rappelle que lorsque la file a une capacité bornée N, on peut l'implémenter en utilisant un tableau `tab` de taille N qu'on traite de façon circulaire. On maintient alors à jour :
   — une variable `size` contenant le nombre d'élément de la file
   — une variable `next` contenant l'indice du prochain élément à défiler

   Expliciter les opérations enfiler et défiler en terme de modification sur `tab`, `size` et `next`.
3. Dans le cas où $N = 3$, décrire le contenu de `tab` et des variables `size` et `next` lorsqu'on effectue les opérations suivante : enfiler 2, enfiler 3, défiler, enfiler 4, défiler, enfiler 7, enfiler 8.
4. Donner une implémentation en OCaml en utilisant le type suivant :

```
type 'a file = {
  capacity : int;
  data : 'a array;
  mutable size : int;
  mutable next : int
  }
```

❑ Exercice 5 : *Comptine enfantine*

Certaines comptines enfantines ont pour objectif de désigner une personne « au hasard », un exemple bien connu est « *Am, stram, gram, pic et pic et colégram* ». On suppose que N enfants numérotés de 0 à $N-1$ sont assis en cercle et que l'un d'entre eux (le numéro k) récite une comptine contenant S syllabes. A la première syllable il désigne son suivant immédiat dans le cercle puis il avance d'un enfant à chaque syllable jusqu'à la fin de la comptine. L'enfant désigné à la fin de la comptine doit quitter le cercle et le processus recommence à partir de son suivant immédiat jusqu'à ce qu'un seul enfant reste.

1. Donner une illustration de ce processus avec $N = 5$ et $S = 7$, en supposant que l'enfant 0 commence.
2. Implémenter en OCaml, un programme exécutant ce processus et donnant le numéro de l'enfant restant. On pourra utiliser le module `Queue` de OCaml. Dont on rappelle ci-dessous les fonctions principales :
   — `Queue.create` qui crée une file vide d'éléments de type `'a`.
   — `Queue.add` qui enfile un élément.
   — `Queue.take` qui défile

❑ Exercice 6 : *Tester si un arbre est un* ABR

1. Rappeler la définition d'un arbre binaire de recherche
2. Proposer deux méthodes de complexité linéaire permettant de vérifier qu'un arbre est bien un ABR.
3. Donner l'implémentation de l'une au moins des méthodes.

❑ Exercice 7 : *Recherche dans un* ABR

1. Rappeler la définition d'un arbre binaire de recherche
2. On suppose maintenant qu'on a inséré dans un ABR initialement vide tous les entiers compris en 0 et 999. On effectue la recherche de l'entier 666 dans cet arbre. Parmi les séquences de valeurs suivantes, lesquelles peuvent être la séquence de noeuds parcourus jusqu'à atteindre 666 ? :
   — 487, 503, 911, 954, 499, 651, 672, 668, 666
   — 951, 812, 803, 798, 751, 670, 589, 652, 653, 666
   — 985, 112, 251, 306, 444, 503, 574, 602, 605, 681, 666
   — 844, 511, 845, 603, 702, 651, 699, 660, 670, 665, 666
   — 303, 404, 541, 752, 749, 742, 592, 603, 666

3. Proposer un algorithme qui prend en entrée une séquence d'entiers $u_0, \dots u_n$ avec u_n la valeur cherchée et vérifie que cette séquence peut effectivement constituer la suite de noeuds visités lors de la recherche réussie d'un nombre dans un tel ABR. L'algorithme doit avoir une complexité temporelle en $O(n)$.
4. En fournir une implémentation en OCaml, en supposant que la séquence est donnée sous la forme d'un tableau d'entiers de OCaml. La signature de votre fonction sera donc `int array -> bool`
5. Soit t un tableau représentant la suite de valeurs obtenue lors de la recherche réussie d'un élément dans un ABR, proposer un algorithme *de complexité linéaire* permettant de trier ce tableau. En donner l'implémentation en OCaml.

❑ **Exercice 8** : *Collision dans une table de hachage*
Pour une chaine de caractères $s = c_0 \dots c_{n-1}$, on considère la fonction de hachage :

$$h(s) = \sum_{i=0}^{n-1} 31^i \times c_i$$

1. Calculer le hash de la chaine `"AB"`.
2. Montrer qu'il existe deux chaines de caractères de longueur 2, formées de lettres minuscules (code 97 à 177) ou majuscules (code 65 à 90) et produisant la même valeur pour h.
3. En déduire une façon de construire un nombre arbitraire de chaînes de caractères de longueurs quelconques ayant la même valeur pour la fonction h.
4. Pour implémenter cette fonction en langage C, on propose une fonction de signature `int hash(char *s)`. Qu'en pensez-vous ?
5. Proposer une implémentation *efficace* pour cette fonction en langage C.
6. Déterminer, grâce à la question 3, deux chaines de 8 caractères produisant une collision et le vérifier.