❑ 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

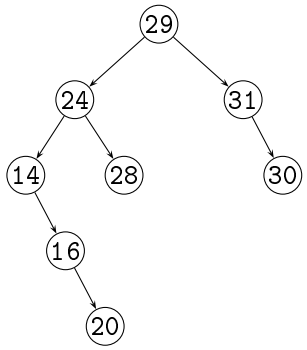
   En utilisant le résultat de la question précédente, il suffit de calculer la taille de l'arbre, on traite le cas de l'arbre vide avec un `failwith` :

```
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 :

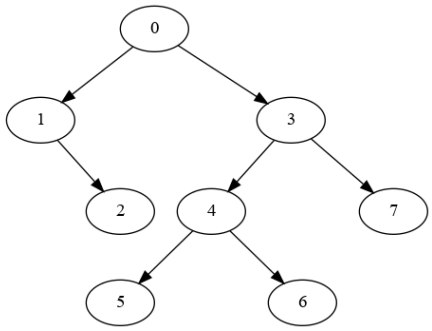


   — préfixe : `[ 29; 24; 14; 16; 20; 28; 31; 30 ]`
   — infixe : `[ 14; 16; 20; 24; 28; 29; 31; 30 ]`
   — postfixe : `[ 20; 16; 14; 28; 24; 30; 31; 29 ]`

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.

```
let rec separe_valeur liste valeur =
  match liste with
  | [] -> failwith "L'entier n'est pas dans la liste"
  | h::t -> if h=valeur then [],t else let l1,l2 = separe_valeur t valeur
  ↪  in h::l1,l2

let rec separe_nb  liste nb =
  match liste, nb with
  | l , 0 -> [], l
  | [], _ -> failwith "Erreur d'index"
  | h::t, n -> let l1,l2 = separe_nb t (n-1) in h::l1, l2

let rec reconstruit prefixe infixe =
  match prefixe, infixe with
  | [],[] -> Vide
  | [],_ | _, [] -> failwith "Ne sont pas des parcours prefixe et infixe"
  | h::t, infi ->
    let infixe1, infixe2 = separe_valeur infixe h  in
    let prefixe1, prefixe2 = separe_nb t (List.length infixe1)  in
    let p1 = reconstruit prefixe1 infixe1 in
    let p2 = reconstruit prefixe2 infixe2 in
      Noeud(p1, h ,p2);;
```

❑ **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.

   Structure de données séquentielle de type LIFO (dernier entré, premier sorti)
   - — Empiler (ajouter un élément au sommet de la pile)
   - — Dépiler (retirer l'élément situé au sommet de la pile)
   - — Est_vide (qui permet de savoir si la pile est vide)

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.

```
open Stack;;

type parenthese = {ouvrante : char; fermante : char}

let couples = [
    {ouvrante = '('; fermante=')'};
    {ouvrante = '['; fermante=']'};
    {ouvrante = '{'; fermante='}'}
    ];;

let rec list_of_string str =
  if str = "" then [] else str.[0]::list_of_string (String.sub str 1
  ↪  ((String.length str) -1));;


let bien_parenthesee expr =
  let my_stack = Stack.create() in
  let lexp = list_of_string expr in
  let rec aux_bp lexp stack =
    match lexp with
    | [] -> Stack.is_empty my_stack
    | a :: t -> if List.mem a (List.map (fun x -> x.ouvrante) couples) then
    ↪  (Stack.push a my_stack; aux_bp t stack)  else
                if List.mem a (List.map (fun x -> x.fermante) couples) then
                ↪  (
                  if Stack.is_empty my_stack then false else
                    let s = Stack.pop my_stack in
                    List.mem {ouvrante = s; fermante = a} couples && aux_bp
                    ↪  t stack
                )
                  else aux_bp t stack
  in
  aux_bp lexp my_stack;;
```

❑ Exercice 4

1. Rappeler la définition du type abstrait *file* et donner les fonctions de son interface.

   Structure de données séquentielle de type FIFO (premier entré, premier sorti)
   - Enfiler (ajouter un élément à la file)
   - Défiler (retirer un élément de la file)
   - Est_vide (qui permet de savoir si la file est vide)

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`.

— enfiler (on doit vérifier que la file n'est pas pleine en comparant `size` et `N`). En notant `x` l'élément à enfiler et on affecte `tab[(next+size)%N]=x` et on incrémente `size`.

— défiler on renvoie `tab[next]`, on incrémente `next` et on décrémente `size`.

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.

tab	size	next
[2]	1	0
[2; 3]	2	0
[2; 3]	1	1
[2; 3; 4]	2	1
[2; 3; 4]	1	2
[7; 3; 4]	2	2
[7; 8; 4]	3	2

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
  }
```

```
let enfiler file elt =
  if file.capacity=file.size then false else
    (
      file.data.((file.next + file.size) mod file.capacity) <- elt;
      file.size <- file.size +1;
      true
    )

let defiler file =
  if file.size = 0 then failwith "File vide" else
    file.size <- file.size - 1;
    file.next <- file.next+1;
    file.data.(file.next-1);;
```

❑ **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.

- [0 ; 1 ; 2 ; 3 ; 4] 2 quitte le cercle et 3 récite la comptine
- [0 ; 1 ; 3 ; 4] 1 quitte le cercle et 3 récite la comptine
- [0 ; 3 ; 4] 4 quitte le cercle et 0 récite la comptine
- [0 ; 4] 4 quitte le cercle

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

```
open Queue;;

let dernier nb_enfants longueur =
  let cercle = Queue.create () in
  let enfant = ref 0 in
  for i=0 to nb_enfants-1 do
    Queue.add i cercle
  done;
  for i=1 to nb_enfants-1 do
    for j=0 to longueur-1 do
      enfant := Queue.take cercle;
      Queue.push !enfant cercle;
    done;
      ignore (Queue.take cercle);
  done;
  Queue.take cercle;;
```

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

1. Rappeler la définition d'un arbre binaire de recherche

   Pour tous les noeuds de l'arbres, les valeurs du sous arbre gauche (resp. droit) sont strictement inférieures (resp. supérieures) à l'étiquette du noeud.

2. Proposer deux méthodes de complexité linéaire permettant de vérifier qu'un arbre est bien un ABR.

   — Effectuer un parcours infixe de l'arbre en complexité linéaire et vérifier si la liste des valeurs obtenues est triée dans l'ordre croissant
   — Parcourir l'arbre en donnant l'intervalle de valeurs dans lequel doit se trouver les éléments. Initialement l'intervalle est celui des entiers représentables, puis à chaque fois qu'on descend à gauche (resp. à droite) on met à jour la borne droite (resp gauche) de l'intervalle

3. Donner l'implémentation de l'une au moins des méthodes.

— Méthode 1 : attention, le parcours infixe utilisant l'opérateur de concaténation de liste `@` n'est *pas* de complexité linéaire (la preuve a été faite en TD). On utilise ici un parcours infixe avec un accumulateur de façon à avoir une complexité linéaire. Il faut ensuite vérifier que la liste obtenue est triée.

```
let infixe t =
  let rec aux_infixe t acc =
    match t with
    | Vide -> acc
    | Noeud (g, v, d) -> aux_infixe g (v::(aux_infixe d acc)) in
  aux_infixe t [];;

let rec est_croissant liste =
  match liste with
  | [] -> true
  | h::[] -> true
  | h1::h2::t -> h1<h2 && est_croissant (h2::t);;

let est_abr_v1 tabr = est_croissant (infixe tabr);;
```

— Méthode 2 : On initialise les deux bornes de l'intervalle dans lequel doivent se trouver les valeurs rencontrées avec le plus petit (`Int.min_int`) et le plus grand (`Int.max_int`) entier représentable.

```
let rec est_abr_v2 tabr =
  let rec est_aux tabr vmin vmax=
  match tabr with
  | Vide -> true
  | Noeud (g, v, d) -> est_aux g vmin v && est_aux d v vmax in
  est_aux tabr Int.min_int Int.max_int;;
```

❑ **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

— Non valide ! 487, 503, 911, 954, 499, 651, 672, 668, 666 :
— Valide 951, 812, 803, 798, 751, 670, 589, 652, 653, 666
— Valide 985, 112, 251, 306, 444, 503, 574, 602, 605, 681, 666
— Non valide ! 844, 511, 845, 603, 702, 651, 699, 660, 670, 665, 666
— Valide 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 $O(n)$.

   Initialement, les valeurs rencontrées doivent être comprises en 0 et 999. Puis pour tout $i = 0 \dots n$, si u_i est supérieure (resp. inférieure) à la valeur cherchée on met à jour la borne supéreure (resp. inférieure). On renvoie une erreur si la valeur n'est pas comprise entre les deux bornes

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`

   L'énoncé ne précise pas si la séquence est fournie sous forme de tableau ou de liste (laissé au choix de l'élève). Ci-dessous les corrections dans les deux cas de figure

```
let valide_liste seq target =
  let rec valid_aux seq target vmin vmax =
    match seq with
    | [] -> false
    | [target] -> true
    | hseq::tseq -> vmin < hseq && hseq<vmax && if hseq<target then
    ↪  valid_aux tseq target hseq vmax else  valid_aux tseq target vmin
    ↪  hseq
  in
  valid_aux seq target 0 999;;

let valide_tableau tseq =
  let target = tseq.(Array.length tseq -1) in
  let vmin = ref 0 in
  let vmax = ref 999 in
  let error = ref false in
  let index = ref 0 in
  while not !error && !index < (Array.length tseq -1) do
    if tseq.(!index) < !vmin || tseq.(!index) > !vmax then error := true
    ↪  else
      if tseq.(!index) > target then vmax := tseq.(!index) else vmin :=
      ↪  tseq.(!index);
    index := !index +1;
    done;
  not !error;;
```

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.

De part la propriété des ABR, les valeurs inférieures à celle recherchée sont rangées dans l'ordre décroissante dans la séquence et les valeur supérieures à la valeur recherchée sont rangées dans l'ordre croissant. On parcourt donc la séquence en créant deux listes (les valeurs supérieures et inférieure à la valeur cherchée) et on concatene ensuite ces listes.

```
let tri tseq target =
  let rec tri_aux tseq inf sup target =
  match tseq with
  | [] -> (List.rev inf) @ sup
  | h::t -> if h<target then tri_aux t (h::inf) sup target else tri_aux t
  ↪  inf (h::sup) target in
  tri_aux tseq [] [] target;;
```

On peut en donner une version un peu plus concise avec :

```
let tri2 tseq target =
  List.filter (fun x -> x<target) tseq @ List.rev (List.filter (fun x ->
  ↪  x>=target) tseq)
```

❑ **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"`.

   $31 \times 65 + 66 = 2081$

2. Montrer qu'il existe deux chaines de caractères de longueur 2, formées de lettres minuscules (code 97 à 122) ou majuscules (code 65 à 90) et produisant la même valeur pour h.

   En notant $E = [\![65; 90]\!] \cup [\![97; 122]\!]$, on cherche $(a, b) \in E^2$ et $(a', b') \in E^2$ tels que : $(a, b) \neq (a', b')$ et $31a + b = 31a' + b'$. Cette équation se ramène à $b - b' = 31(a' - a)$. Donc $b - b'$ est un multiple non nul de 31. l'écart entre b' et b étant au maximum en valeur absolue de $122 - 65 = 57$ les seules possiblités sont :

   - $b - b' = 31$ et donc $a' - a = 1$. Ce qui donne $a' = a + 1$ (donc a' et a sont dans le même intervalle) et $b' = b - 31$ donc ils ne sont pas dans le même intervalle. On peut prendre par exemple la solution $a = 65$ donc $a' = 66$ et $b = 97$ donc $b' = 66$ c'est à dire les chaines `"Aa"` et `"BB"`.
   - $b - b' = -31$ et donc $a' - a = -1$. Ce qui donne $a' = a - 1$ et $b' = b + 31$, par exemple la solution $a = 66$, $a' = 65$, $b = 67$ et $b' = 98$ convient (c'est à dire les chaines `"BC"` et `"Ab"`)

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.

   En concaténant plusieurs chaines de caractères de longueur deux ayant le même hash on obtient des chaine de caractère de longueur paire arbitrairement grande et ayant le même hash. Par exemple (en utilisant les exemples de longueur deux donnés à la question précédente), `"AaBC"` et `"BBAb"`. Pour la longueur impaire on rajoute le même caractère aux deux chaines.

4. Pour implémenter cette fonction en langage C, on propose une fonction de signature `int hash(char *s)`. Qu'en pensez-vous ?

Les chaines de caractères "connaissent" leur longueur en C grâce au caractère sentinelle, on a donc pas besoin de passer la longueur de la chaine de caractère en paramètre. Par contre, cette fonction risque fort de provoquer un dépassement de capacité sur le type `int` ce qui est un comportement indéfini en C. On devrait plutôt utiliser la librairie `stdint.h` et renvoyer un `uint`

5. Proposer une implémentation *efficace* pour cette fonction en langage C.

Le calcul par la méthode de Horner est de complexité linéaire (car évite le calcul explicite des puissances de 31)

```c
uint32_t hash(char *s)
{
    uint32_t h = 0;
    int index = 0;
    while (s[index]!='\0')
    {
        h = 31*h + s[index];
        index++;
    }
    return h;
}
```

6. Déterminer, grâce à la question 3, deux chaines de 8 caractères produisant une collision et le vérifier.

On peut simplement deux chaines de longueur 4 ayant le même hash avec elle-même :

```c
int main()
{
    char s1[] = "AaBCAaBC";
    char s2[] = "BBAbBBAb";
    printf("Hash de %s = %u\n",s1,hash(s1));
    printf("Hash de %s = %u\n",s1,hash(s2));
}
```