

□ Exercice 1 : Nombre d'arêtes

1. Rappeler la définition d'un arbre binaire.

Voir cours

2. Soit a un arbre binaire à n noeuds ($n \geq 1$), 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 distinguer le cas où l'un des sous arbre est vide.

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

```
1 type ab =
2   | Vide
3   | Noeud of ab * int * ab;;
```

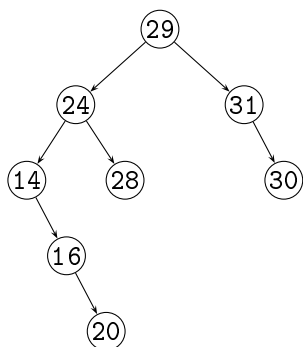
En utilisant cette implémentation, écrire une fonction `nb_aretes` de signature `ab -> int` et qui renvoie le nombre d'arêtes d'un arbre binaire

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` :

```
1 let rec taille ab =
2   match ab with
3   | Vide -> 0
4   | Noeud (g, r, d) -> 1 + taille g + taille d
5
6 let nb_aretes ab =
7   match ab with
8   | Vide -> failwith "L'arbre est vide"
9   | a -> taille a - 1
```

□ Exercice 2 : Parcours d'arbre binaires

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



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

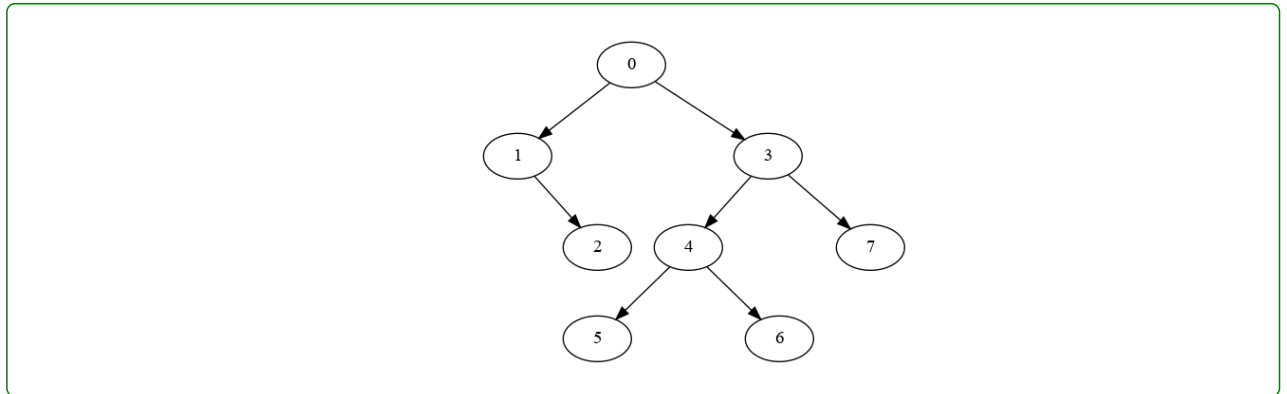
— infixe : [14; 16; 20; 24; 28; 29; 31; 30]

— 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 préfixe 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 préfixe 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 préfixe 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 préfixe 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 éléments 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.

```

1  let rec separe_valeur liste valeur =
2      match liste with
3      | [] -> failwith "L'entier n'est pas dans la liste"
4      | h::t -> if h=valeur then [],t else let l1,l2 = separe_valeur t valeur
5          ↪ in h::l1,l2
6
6  let rec separe_nb liste nb =
7      match liste, nb with
8      | l , 0 -> [], l
9      | [], _ -> failwith "Erreur d'index"
10     | h::t, n -> let l1,l2 = separe_nb t (n-1) in h::l1, l2
11
12 let rec reconstruit prefixe infixe =
13     match prefixe, infixe with
14     | [], [] -> Vide
15     | [], _ | _, [] -> failwith "Ne sont pas des parcours prefixe et infixe"
16     | h::t, infi ->
17         let infixe1, infixe2 = separe_valeur infixe h in
18         let prefixe1, prefixe2 = separe_nb t (List.length infixe1) in
19         let p1 = reconstruit prefixe1 infixe1 in
20         let p2 = reconstruit prefixe2 infixe2 in
21         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_parenthesesee` de signature `str -> bool` qui renvoie `true` lorsque la chaîne 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.

```

1  open Stack;;
2
3  type parenthese = {ouvrante : char; fermante : char}
4
5  let couples = [
6    {ouvrante = '('; fermante=')'};
7    {ouvrante = '['; fermante=']'};
8    {ouvrante = '{'; fermante='}'}
9  ];;
10
11 let rec list_of_string str =
12   if str = "" then [] else str.[0]::list_of_string (String.sub str 1
13   ↪ ((String.length str) -1));;
14
15 let bien_parenthesee expr =
16   let my_stack = Stack.create() in
17   let lexp = list_of_string expr in
18   let rec aux_bp lexp stack =
19     match lexp with
20     | [] -> Stack.is_empty my_stack
21     | a :: t -> if List.mem a (List.map (fun x -> x.ouvrante) couples) then
22       ↪ (Stack.push a my_stack; aux_bp t stack) else
23         if List.mem a (List.map (fun x -> x.fermante) couples) then
24           ↪ (
25             if Stack.is_empty my_stack then false else
26             let s = Stack.pop my_stack in
27             List.mem {ouvrante = s; fermante = a} couples && aux_bp
28             ↪ t stack
29           )
30         else aux_bp t stack
31   in
32   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 :

```

1 type 'a file = {
2   capacity : int;
3   data : 'a array;
4   mutable size : int;
5   mutable next : int
6 }
```

```

1 let enfiler file elt =
2   if file.capacity=file.size then false else
3   (
4     file.data.((file.next + file.size) mod file.capacity) <- elt;
5     file.size <- file.size +1;
6     true
7   )
8
9 let defiler file =
10  if file.size = 0 then failwith "File vide" else
11  file.size <- file.size - 1;
12  file.next <- file.next+1;
13  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 syllabe il désigne son suivant immédiat dans le cercle puis il avance d'un enfant à chaque syllabe 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

```

1  open Queue;;
2
3  let dernier nb_enfants longueur =
4      let cercle = Queue.create () in
5      let enfant = ref 0 in
6      for i=0 to nb_enfants-1 do
7          Queue.add i cercle
8      done;
9      for i=1 to nb_enfants-1 do
10         for j=0 to longueur-1 do
11             enfant := Queue.take cercle;
12             Queue.push !enfant cercle;
13         done;
14         ignore (Queue.take cercle);
15     done;
16     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.

□ 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 chaîne 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 chaîne "AB".
2. Montrer qu’il existe deux chaînes 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 .
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 chaînes de 8 caractères produisant une collision et le vérifier.