**Important.** Une grande importance sera apportée à la clarté, la lisibilité des raisonnements et des programmes. Dans ce sujet, on rédigera les programmes dans le langage OCaml.

**Remarque.** On pourra toujours librement utiliser une fonction demandée à une question précédente, même si cette question n'a pas été traitée.

Ce sujet est composé de 3 parties totalement indépendantes.

# 1 Arbres binaires en C

**Important.** Dans cette partie, on rédigera les programmes dans le langage C. On suppose que les librairies standards `stdbool.h`, `stdio.h` et `stdlib.h` sont déjà importées et vous n'avez pas besoin de le préciser.

On considère les types suivants :

```c
struct node {
  int key;
  struct node *left;
  struct node *right;
};
typedef struct node node;

struct abr {
  node *root;
  int size;
};
typedef struct bst bst;
```

On maintiendra les invariants suivants :

- un nœud vide est représenté par le pointeur `NULL` ;
- un ABR (Binary Search Tree en anglais) vide est représenté par un pointeur vers un `bst` dont le champ `root` vaut `NULL` et le champ `size` vaut 0 ;
- si $n$ est un nœud non vide de clé $k$, alors les clés présentes dans son sous-arbre gauche sont **strictement inférieures** à $k$, et celles dans sont sous-arbre droit sont **strictement supérieures** à $k$ ;
- si `t` est un `bst*`, alors son champ `size` est égal au nombre de nœuds non vides dans l'arbre dont la racine est le nœud `t->root`.

**Exercice 1 (Création et libération) :**

1. Écrire une fonction `node *node_new(int key);` créant un nœud ayant l'étiquette donnée en argument, et deux fils vides.
2. Écrire une fonction `bst *bst_new(void);` renvoyant un pointeur vers un nouveau `bst` vide.
3. Écrire une fonction `void node_free(node *n);` libérant toute la mémoire associée à un nœud (et au sous-arbre enraciné en ce nœud).
4. Écrire une fonction `void bst_free(bst *t);` libérant toute la mémoire associée à un ABR.

**Exercice 2 (Fonctions élémentaires) :**

5. Écrire une fonction `bool node_member(node *n, int key);` déterminant si une clé apparaît dans le sous-arbre enraciné en un nœud.
6. (a) Écrire une fonction `bool bst_member(bst *t, int key);` déterminant si une clé apparaît dans un ABR.
   (b) Donner la complexité de cette fonction.

7. Écrire une fonction `node *node_insert(node *n, int key);` qui ajoute une clé à un sous-arbre. Cette fonction renverra la racine su sous-arbre modifié, et n'aura aucun effet si l'élément à ajouter était déjà présent dans le sous-arbre.
8. (a) Écrire une fonction `void bst_insert(bst *t, int key);` qui ajoute une clé à un arbre binaire de recherche. On n'oubliera pas de mettre à jour le champ `size`.
   (b) Donner la complexité de cette fonction.

**Exercice 3 (Parcours et équilibrage) :**

9. Écrire une fonction `void node_to_array(node *n, int arr[], int *start_pointer);` qui écrit les étiquettes présentes dans un sous-arbre dans le tableau fourni :
   - les étiquettes seront écrites dans l'ordre croissant ;
   - la valeur initiale de `*start_pointer` indique l'indice du tableau dans lequel il faudra écrire la première (plus petite) étiquette.
   - la fonction devra avoir l'**effet de bord** suivant : après l'appel, `*start_pointer` contient l'indice immédiatement après celui de la dernière case dans laquelle on a écrit au cours de l'appel.

   On pourra supposer sans le vérifier (on ne peut pas, de toute façon) que le tableau est assez grand pour recevoir toutes les étiquettes du sous-arbre en commençant à les écrire à l'indice indiqué.
10. Écrire une fonction `int *bst_to_array(bst *t);` qui renvoie un pointeur vers un bloc alloué contenant les étiquettes de l'arbre passé en paramètre, dans l'ordre croissant.
11. Écrire une fonction `node *node_balanced(int arr[], int start, int end);` ayant la spécification suivante :
    **Précondition** : `arr` est strictement croissant, `start` est positif ou nul, `end` est inférieur ou égal à la longueur de `arr`.
    **Sortie** : un sous-arbre de hauteur minimale ayantles éléments de `arr` d'indice compris entre `start` (inclus) et `end` (exclu) comme étiquettes.
    **Remarque** : on ne demande pas de prouver le caractère minimal de la hauteur du sous-arbre renvoyé.
12. (a) Écrire une fonction `void bst_rebalance(bst *t);` qui modifie un ABR pour qu'il soit parfaitement équilibré, sans changer son nombre d'étiquettes.
    (b) Déterminer la complexité de la fonction `bst_rebalance`.

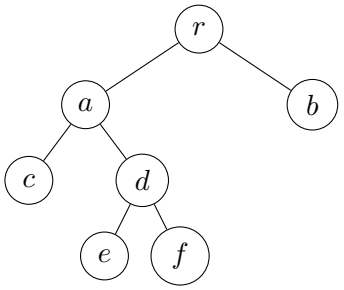
## 2 Arbres parfaits et ensemble de nombres

**Important.** Dans cette partie, on rédigera les programmes dans le langage OCaml.

**Définition.** Soit $p$ une séquence finie de 0 et de 1 et $t$ un arbre binaire.
On définit le chemin associé à $p$ dans $t$ de la manière suivante (en partant de la racine de $t$) :

- si $p = \varepsilon$ ($\varepsilon$ dénote la séquence vide), alors le chemin est vide, et on reste sur le nœud courant ;
- si $p = 0 \cdot p'$, alors on descend au fils gauche du nœud courant, et on continue avec le chemin $p'$ ;
- si $p = 1 \cdot p'$, alors on descend au fils droit du nœud courant, et on continue avec le chemin $p'$.

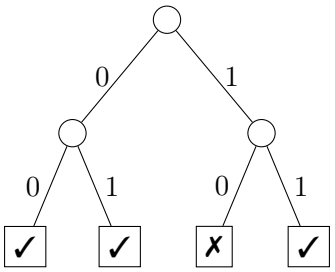
Par exemple, sur l'arbre ci-contre :




- $p = \varepsilon$ mène au nœud $r$ ;
- $p = 0$ mène au nœud $a$ ;
- $p = 1$ mène au nœud $b$ ;
- $p = 00$ mène au nœud $c$ ;
- $p = 01$ mène au nœud $d$ ;
- $p = 010$ mène au nœud $e$ ;
- $p = 011$ mène au nœud $f$ ;
- $p = 10$ n'est pas un chemin valide.

**Définition.** Un arbre binaire est dit **parfait** si toutes ses branches sont de même longueur.

**Exercice 4 :** On souhaite représenter un ensemble d'entiers par un arbre binaire parfait dont les nœuds sont non étiquetés et les feuilles contiennent un booléen.




Dans l'arbre ci-dessus ✓ représente le booléen `true`, ✗ représente le booléen `false`. Un arbre binaire de hauteur $k$ représente le sous-ensemble des entiers de $[\![0, 2^k - 1]\!]$ dont l'écriture en base 2 du bit de poids faible au bit de poids fort conduit à une feuille marquée `true`. Par exemple l'arbre ci-dessus représente le sous-ensemble de $[\![0, 3]\!]$ contenant les entiers dont l'écriture en base 2 est 00, 10, et 11, mais pas 01, l'arbre représente donc l'ensemble $\{0, 2, 3\}$.

Afin de représenter de tels arbres en OCaml on définit le type `type int_set` ci-dessous. L'arbre de l'exemple ci-dessus est représenté par la valeur OCaml `ex` définie ci-dessous. On remarque que les exemples dessinés font apparaître des 0 et 1 pour identifier les arêtes, mais que ceux-ci sont inutiles (et donc non présents) dans la représentation en machine. Finalement, on suppose défini les exceptions `Not_perfect` et `No_fit`.

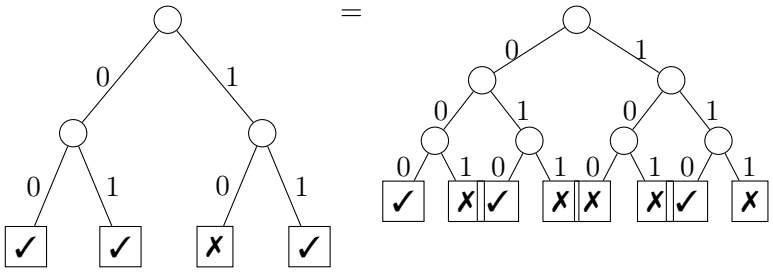
```
type int_set =
  | N of int_set * int_set
  | E of bool
;;

  let ex =
      N(N(E(true),
          E(true)),
        N(E(false),
          E(true)))
  ;;

  exception Not_perfect ;;
  exception No_fit ;;
```

13. Définir une fonction `hauteur_parfait : int_set -> int` prenant en argument un `int_set` et retournant sa hauteur si l'arbre est parfait et levant l'exception `Not_perfect` sinon.
**Attention :** l'arbre `E(...)` est de hauteur 0.
14. *En déduire* une fonction `est_parfait : int_set -> bool` prenant en argument un `int_set` et testant si celui-ci est parfait.
15. Définir une fonction `mem : int -> int_set -> bool` permettant de tester si un entier est représenté dans un ensemble.

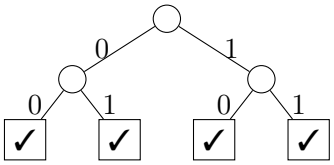
Exemple

```
# mem 2 ex ;;
- : bool = false
# mem 1 ex ;;
- : bool = true
```

16. Définir une fonction `increase_height : int_set -> int_set` augmentant de 1 la hauteur de l'arbre passé en argument, tout en laissant stable l'ensemble des entiers représentés.
**Exemple :** `increase_height` =

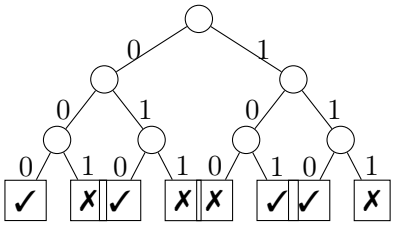


17. Définir une fonction `insert_exn : int_set -> int -> int_set` telle que `insert_exn s x` calcule l'ensemble d'entiers obtenu après insertion de `x` dans `s`.
Si `x` ne peut être inséré dans `s` pour (des raisons de hauteur) alors on lèvera l'exception `No_fit`.
**Exemple :**
- `insert_exn ex 1` =

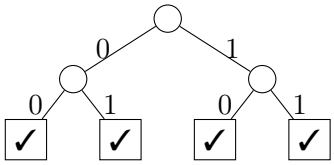


- `insert_exn ex 25` lève l'exception `No_fit`.

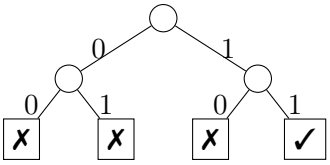
18. Déduire, des deux questions précédentes, une fonction `insert : int_set -> int -> int_set` telle que `insert s x` calcule l'ensemble d'entiers obtenu après insertion de `x` dans `s`. La taille l'arbre sera augmentée au besoin.
**Exemple :**
- `insert ex 5` =



- `insert ex 1` =



- `insert (E false) 3` =



19. Définir une fonction `fold : (int -> 'a -> 'a) -> int_set -> 'a -> 'a` telle que `fold f t a` calcule la valeur :

$$(f\ x_n\ (f\ x_{n-1}\ (\dots\ (f\ x_1\ a)\dots))$$

lorsque `t` représente l'ensemble $\{x_1, \dots, x_{n-1}, x_n\}$. On n'accordera aucune importance à l'ordre dans lequel la fonction `fold` itère sur les éléments de l'ensemble.

Exemple

```
# fold (fun x y -> x + y) ex 0  ;;
- : int = 5
```

20. *En déduire* une fonction `union : int_set -> int_set -> int_set` calculant l'union de deux ensembles d'entiers.

# 3 Arbres binomiaux

**Rappels sur les arbres généraux.**

**Ensembles $\mathcal{G}_S$, et $\mathcal{F}_S$.** Dans cette partie, $\mathcal{G}_S$ dénote l'ensemble des arbres généraux étiquetés par des éléments de $S$, $\mathcal{F}_S$ dénote l'ensemble des forêts d'arbres généraux (à savoir une suite finie d'arbres généraux). Dans toute la suite on suppose de plus le type suivant déjà défini :

```
type 'a gtree = GN of 'a * ('a gtree list)
```

**Tailles : `s` et `sl`.** On rappelle que les fonctions `s` donnant la taille d'un arbre général et `sl` donnant la taille d'une forêt d'arbre généraux peuvent être définies par la récurrence mutuelle suivante :

- $\texttt{s}(\texttt{GN}(x, l)) = 1 + \texttt{sl}(l)$ ;
- $\texttt{sl}(g :: l) = \texttt{s}(g) + \texttt{sl}(l)$ lorsque $g \in \mathcal{G}_S$ et $l \in \mathcal{F}_S$ ;
- $\texttt{sl}([]) = 0$.

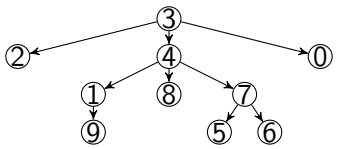
**Hauteur : `h` et `hl`.** On rappelle que la fonction $\texttt{h}(s)$ donnant la hauteur d'un arbre général et `hl` donnant la hauteur d'une forêt d'arbres généraux peuvent être définies par la récurrence mutuelle suivante :

- $\texttt{h}(\texttt{GN}(x, l)) = 1 + \texttt{hl}(l)$ ;
- $\texttt{hl}(g :: l) = \max(\texttt{h}(g), \texttt{hl}(l))$ lorsque $g \in \mathcal{G}_S$ et $l \in \mathcal{F}_S$ ;
- $\texttt{hl}(\varepsilon) = -1$.

**Nœuds et chemins : paths.** On étend la notion de chemin aux arbres généraux :

- $\mathsf{paths}(\texttt{GN}(x, l)) = \{\varepsilon\} \cup \{i \cdot \mathsf{paths}(l_i) \mid i \in [\![0, |l| - 1]\!]\}$

Ainsi un nœud d'un arbre général peut-être vu comme son chemin depuis de la racine de l'arbre. La fonction paths associe alors à un arbre général, l'ensemble de ses nœuds. Par exemple pour l'arbre $t_{\mathsf{ex}}$ suivant :




on a $\mathsf{paths}(t_{\mathsf{ex}}) = \{\varepsilon, \texttt{0}, \texttt{1}, \texttt{2}, \texttt{10}, \texttt{100}, \texttt{11}, \texttt{12}, \texttt{120}, \texttt{121}, \texttt{2}\}$. De plus, l'étiquette du nœud $\varepsilon$ est 3, l'étiquette du nœud `1` est 4, l'étiquette du nœud `12` est 7, l'étiquette du nœud `120` est 5.

**Profondeur : prof.** Étant donné un nœud $n$ d'un arbre général $t$ (vu comme son chemin dans $t$), sa profondeur est la longueur de ce chemin, dénotée $\mathsf{prof}(n)$ ou plus simplement $|n|$.

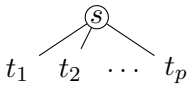
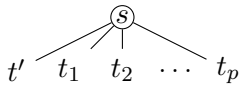
**Exercice 5 (Définitions, vocabulaires, propriétés) :**

**Définition des arbres binomiaux d'ordre $k$.** On définit inductivement l'ensemble des *arbres binomiaux d'ordre $k \in \mathbb{N}$, noté $\mathbb{B}_k$* comme étant un sous-ensemble de l'ensemble des arbres généraux $\mathcal{G}_S$ :

- $\mathbb{B}_0$ contient les arbres généraux de la forme $\texttt{GN}(s, \varepsilon)$ pour $s \in S$, à savoir les arbres généraux qui sont des feuilles ;
- si $b_1 = \texttt{GN}(s_1, l_1) \in \mathbb{B}_k$ et $b_2 \in \mathbb{B}_k$ alors $\texttt{GN}(s_1, b_2 :: l_1) \in \mathbb{B}_{k+1}$.

La définition ci-dessus se traduit schématiquement par :

- $\forall s \in S, \textcircled{s} \in \mathbb{B}_0$.

- si  $\in \mathbb{B}_k$, et $t' \in \mathbb{B}_k$, alors  $\in \mathbb{B}_{k+1}$

Ou encore en OCaml par :

- quelque soit `x`, `GN(x, [])` est un arbre binomial ;
- si `GN(s, l)` est un arbre binomial et `b` est un arbre binomial alors `GN(s, b::l)` est un arbre binomial.

On remarque que cette définition inductive assure que si $b \in \mathbb{B}_k$ alors :

- soit $k = 0$, auquel cas il existe $s \in S$ tel que $b = \texttt{GN}(s, \varepsilon)$ ;
- soit $k \neq 0$, auquel cas il existe $s \in S$, $b_1 = \texttt{GN}(s, l) \in \mathbb{B}_{k-1}$ et $b_2 \in \mathbb{B}_{k-1}$ tels que $b = \texttt{GN}(s, b_2 :: l)$.

Cet exercice est consacré à l'étude de quelques propriétés de arbres binomiaux, au vu des derniers chapitres on s'efforcera de fournir des preuves rigoureuses des résultats énoncés. On précisera par exemple les espaces sur lesquels sont menés les raisonnements par induction.

21. Donner un exemple d'éléments de $\mathbb{B}_4$ étiquetés par des entiers naturels ($S = \mathbb{N}$).

On remarque en fait que les arbres binomiaux d'ordre $k$ ne se distinguent les uns des autres que par leurs étiquettes : l'ensemble des nœuds de tous les arbres de $\mathbb{B}_k$ sont identiques, à savoir $\forall b, b' \in \mathbb{B}_k, \mathsf{paths}(b) = \mathsf{paths}(b')$.

22. Donner et prouver une relation entre $\texttt{s}(b)$ et $k$ lorsque $b \in \mathbb{B}_k$.
23. Donner et prouver une relation entre $\texttt{h}(b)$ et $k$ lorsque $b \in \mathbb{B}_k$.
24. On rappelle qu'un nœud est une *feuille* si il n'a aucun fils. Donner une définition inductive du nombre de feuilles d'un arbre général, trouver une relation entre le nombre de feuilles d'un arbre de $\mathbb{B}_k$ et $k$, prouver cette relation.

**Définition des arbres mystères d'ordre $k$.** Étant donné un ensemble $S$ d'étiquettes, on définit inductivement l'ensemble des *arbres mystères d'ordre $k$, noté* $\mathbb{M}_k$ comme étant un sous-ensemble de l'ensemble des arbres généraux :

- $\mathbb{M}_0$ contient les arbres généraux de la forme $\texttt{GN}(s, \varepsilon)$ pour $s \in S$ ;
- si $t_{k-1} \in \mathbb{M}_{k-1}$, $t_{k-2} \in \mathbb{M}_{k-2}$, …, $t_0 \in \mathbb{M}_0$ et $s \in S$ alors : $\texttt{GN}(s, t_{k-1} :: t_{k-2} :: \cdots :: t_0) \in \mathbb{M}_k$.

25. Démontrer que $\forall k \in \mathbb{N}, \mathbb{M}_k = \mathbb{B}_k$.
26. En déduire que si $t \in \mathbb{B}_k$ alors la racine de $t$ a exactement $k$ fils.

**Exercice 6 (Fabrication d'arbres binomiaux) :**

27. Définir une fonction `est_binomial: 'a gtree -> int -> bool` permettant de tester que l'arbre général passé en argument est bien un arbre binomial d'ordre l'entier passé en argument.

Exemple

```
# est_binomial (GN(0, [GN(2, [GN(3, [])]); GN(1, [])])) 2 ;;
- : bool = true
# est_binomial (GN(0, [GN(2, [GN(3, [])])])) 2 ;;
- : bool = false
```

28. Définir une fonction `map : ('a -> 'b) -> 'a gtree -> 'b gtree` prenant en argument une fonction `f` et un arbre général `t` et calculant l'arbre obtenu par application de `f` sur chacune des étiquettes de `t`.

Exemple

```
# map (fun x -> string_of_int x) (GN(0, [GN(2, [GN(3, [])]); GN(1, [])])) ;;
- : string gtree = GN ("0", [GN ("2", [GN ("3", [])]); GN ("1", [])])
# map (fun x -> 2*x) (GN(1, [GN(3, [GN(4, [])])])) ;;
- : int gtree = GN (2, [GN (6, [GN (8, [])])])
```

29. En déduire une fonction `generate : int -> int gtree` prenant en argument un entier $k$ et calculant un arbre binomial d'ordre $k$ dont les étiquettes font apparaître une et une seule fois chaque entier de $[\![0, 2^k - 1]\!]$.
On pourra remarquer que l'application de la fonction $x \mapsto 2x$ sur toutes les étiquettes d'un arbre binomial étiqueté par l'intervalle $[\![0, 2^p - 1]\!]$ produit un arbre binomial dont les étiquettes sont dans l'intervalle $[\![0, 2^{p+1} - 1]\!]$.

Exemple

```
# generate 0 ;
- : int gtree = GN(0, [])
# generate 0 ;
- : int gtree = GN(0, [GN(1, [])])
# generate 2 ;
- : int gtree = GN(0, [GN(1, [GN(3, [])]); GN(2, [])])
```

**Exercice 7 (Tas binomiaux) :** Dans cet exercice on se propose d'utiliser les arbres binomiaux définis ci-avant pour implémenter une structure de type de données abstrait : file de priorité.

30. Rappeler les opérations du type de données abstrait file de priorité.

On ajoute au type de donnée abstrait file de priorité les deux opérations :

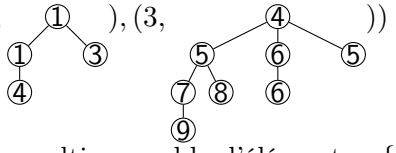
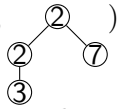
$$\mathsf{union} : \mathsf{fp} \times \mathsf{fp} \to \mathsf{fp}$$
$$\mathsf{singleton} : \mathsf{elem} \to \mathsf{fp}$$

union permet, à partir de deux files de priorité $f_1$ et $f_2$, de calculer la file de priorité contenant l'union (au sens des multi-ensembles) de tous les éléments de $f_1$ et $f_2$. singleton permet, de créer une file de priorité contenant uniquement l'élément passé en argument.

31. Montrer comment implémenter l'opération d'insertion des files de priorité au moyen des opérations union et singleton.

**Définition des tas binomiaux** On dit d'un arbre général qu'il est un *tournoi* si pour tout nœud de l'arbre ayant un père, l'étiquette du père est inférieure à l'étiquette du fils. Un tas binomial est défini comme une suite finie de couples $((o_i, b_i))_{i \in [\![0,p-1]\!]}$ telle que $(o_i)_{i \in [\![0,p-1]\!]}$ est une suite strictement croissante d'entiers naturels et $\forall i \in [\![0, p-1]\!], b_i \in \mathbb{B}_{o_i}$ et $b_i$ est un tournoi. Un tas $((o_i, b_i))_{i \in [\![0,p-1]\!]}$ représente le multi-ensemble des étiquettes des $b_i$.

**Exemples :**

- $e_1 = ((0, ②), (2,$  $), (3,$  $))$
qui représente le multi-ensemble d'éléments : $\{\{2, 1, 1, 3, 4, 4, 5, 6, 7, 8, 6, 9\}\}$.
- $e_2 = ((0, ⑤), (2,$  $)$
qui représente le multi-ensemble d'éléments : $\{\{5, 2, 2, 7, 3\}\}$.

On définit alors le type `type 'a tbin = (int * 'a gtree) list` permettant la représentation d'un tas binomial.

Par exemple le tas binomial $e_1$ ci-dessus est représenté par la valeur OCaml :

```
let ex1 = [
  (0, GN(2, []));
  (2, GN(1, [
       GN(1, [
           GN(4, [])]);
       GN(3, [])]));
  (3, GN(4, [
       GN(5, [
           GN(7, [
               GN(9, [])]);
           GN(8, [])]);
       GN(6, [
           GN(6, [])]);
       GN(5, [])]))
] ;;
```

32. Définir une fonction `est_tournoi : 'a gtree -> bool` testant si un arbre général est un tournoi. On assurera une complexité linéaire en $s(g)$.

Exemple

```
# est_tournoi (GN(1, [])) ;;
- : bool = true
# est_tournoi (GN(1, [GN(2, [])])) ;;
- : bool = true
# est_tournoi (GN(2, [GN(4, []); GN(1, [])])) ;;
- : bool = false
```

33. Définir une fonction `est_tas_bin : 'a tbin -> bool` testant si un tas binomial vérifie bien toutes les propriétés requises pour être un tas binomial.

Exemple

```
# est_tas_bin ex1 ;;
- : bool = true
# est_tas_bin [] ;;
- : bool = true
# est_tas_bin [(3, GN(1, []))] ;;
- : bool = false
# est_tas_bin [(1, GN(1, [GN(2, [])])); (0, GN(1, []))] ;;
- : bool = false
# est_tas_bin [(0, GN(2, [])); (0, GN(1, []))] ;;
- : bool = false
```

34. Définir la valeur `empty : 'a tbin` qui est l'unique tas binomial représentant le multi-ensemble vide d'étiquettes.

35. (a) Définir une fonction `minimum: 'a tbin -> 'a` calculant l'étiquette minimal se trouvant dans le tas binomial passé en argument si celui-ci est non vide.

    (b) Préciser la complexité dans le pire des cas en terme de $n$, le nombre d'éléments se trouvant dans le tas binomial.

36. (a) Définir une fonction `union_binom_tournoi : 'a gtree -> 'a gtree -> 'a gtree` prenant en argument deux arbres binomiaux $t_1$ et $t_2$ de même ordre $k$ et calculant un arbre binomial tournoi d'ordre $k+1$ dont l'ensemble d'étiquette est l'union des étiquettes de $t_1$ et de $t_2$.

    (b) Préciser la complexité dans le pire des cas.