Cours

- Donner la définition inductive d'un arbre. D'un arbre binaire.
- Qu'est-ce qu'une feuille ? Noeud ? Racine ?
- Donner la défintion de la hauteur d'un arbre.
- Définition d'un arbre complet. Localement complet. Parfait.
- Donner la définition d'un tas min (et d'un tas max)
- Rapeller le principe du tri par tas et sa complexité.
- Donner la définition d'un arbre binaire de recherche.
- Arbre rouge-noir. Ajout dans un arbre rouge-noir.

Questions de programmation du cours:

Pour chacune des fonctions suivantes, donne le code en OCaml et en C. Les signatures sont données en OCaml:

- Coder la fonction `let hauteur (t: 'a tree): int` qui prend un arbre en entrée et qui renvoie sa hauteur.
- Coder la fonction `let count (t: 'a tree): int` qui prend un arbre en entrée et qui renvoie son nombre de sommets.
- Coder la fonction `let prefixe (t: 'a tree): 'a list` qui prend un arbre en entrée et qui renvoie son parcours préfixe. *Le code C renverra un tableau au lieu d'une liste*
- Coder la fonction `let infixe (t: 'a tree): 'a list` qui prend un arbre en entrée et qui renvoie son parcours infixe. *Le code C renverra un tableau au lieu d'une liste*
- Coder la fonction `let suffixe (t: 'a tree): 'a list` qui prend un arbre en entrée et qui renvoie son parcours suffixe. *Le code C renverra un tableau au lieu d'une liste*
- Coder la fonction `let add (t: int tree) (x:int): int tree` qui ajoute à un arbre binaire de recherche `t` un élément `x`.
- Coder la fonction `let remove (t: int tree) (x:int): bool` qui retire à un arbre binaire de recherche `t` un élément `x`, et renvoie s'il existait.
- Coder la fonction `let push (t: tas) (x:int): tas` qui ajoute à un tas min `t` un élément `x`.
- Coder la fonction `let pop (t: tas): int` qui retire d'un tas min `t` son minimum et le renvoie.

Exercices à ajouter

- Comptage de sous-arbres: https://diplome.di.ens.fr/informatique-ens/annales/2017_InfoU-exercices.pdf
- https://en.wikipedia.org/wiki/Merkle_tree (mélange hash et arbres)

Nombre de sommets

Montrer que pour A un arbre binaire, si on note $|A|$ son nombre de noeuds et $h(A)$ sa hauteur, montrer que $|A| \leq 2^{h(A)+1} - 1$

Plus de feuilles

Soit T un arbre tel que chaque sommet qui n'est pas une feuille a un degré au moins 3. Montrer que T a plus de feuilles que de nœuds internes.

Contour d'un arbre

On considère la fonction contour suivante en OCaml:

```
type 'a tree = F | N of 'a tree * 'a * 'a tree;
let rec contour t = match t with
  | F -> []
  | N (g,x,d) -> [x] @ (contour g) @ [x] @ (contour d) @ [x]
```

1. Quel est le type de la fonction `contour` ?
2. Proposer une implémentation avec un accumulateur qui est terminal récursive de `contour`.
3. Donner le code de `val to_prefixe: 'a list -> 'a list` tel que `to_prefixe (to_contour t)` renvoie le parcours préfixe de `t`.
4. Montrer que pour chaque noeud x de parent p, on a $[x, p]$ et $[p, x]$ qui apparaissent dans le contour.

Arbres d'intervalles

Un arbre d'intervalles est un arbre binaire de recherche dont tous les nœuds contiennent un intervalle de la forme $[a; b]$, dont les clefs sont dans l'ordre lexicographique définie par la relation d'ordre $\preceq$:

$$(a, x) \preceq (b, y) \iff a < b \vee (a = b \wedge x \leq y)$$

On se donne le type suivant en OCaml:

```
type intervalle = int * int;;
type arbre_int = F | N of intervalle * arbre_int * arbre_int;;
```

Question 1 Qu'est-ce qu'un arbre binaire de recherche ? Proposer une structure en C similaire à celle proposée pour représenter un arbre d'intervalles.

Question 2 Donner en C la fonction `int hauteur(arbre* a)` donnant la hauteur d'un arbre d'intervalle.

Question 3 Dessiner puis donnez en OCaml un arbre complet contenant les intervalles $\{[0; 2]; [0; 1]; [1; 3]; [4; 5]; [3; 5]; [3; 3]\}$

Question 4 Donner `val trouver: arbre_int -> intervalle -> intervalle` tel que `trouver a i` retourne un intervalle de l'arbre `a` intersectant `i` en $O(h)$ avec h la hauteur de `a`.

Question 5 Définir les opérations de rotations sur les arbres binaire de recherche. Donnez la fonction `val rotg: arbre_int -> arbre_int` effectuant l'opération de rotation gauche. On supposera écrite la fonction `val rotd: arbre_int -> arbre_int` l'opération de rotation droite.

Question 6 Donner `val ajouter: arbre_int -> intervalle -> arbre_int` tel que `ajouter a i` ajoute à un arbre équilibré `a` l'intervalle `i`. On veillera à ce que `a` reste équilibré.

Accès dans un arbre parfait

On se donne le type C suivant

```
typedef struct Noeud *arb;
struct Noeud {
  int valeur;
  arb fils_g;
  arb fils_d;
};
```

1. Donnez une définition équivalente de ce type en OCaml. Toujours en OCaml, donnez la fonction `val hauteur: arbre -> int` donnant pour un arbre quelconque sa hauteur.
2. Dessinez un arbre parfait à 7 nœuds. Tout les arbres complets sont-ils parfaits ?
3. Démontrez que tout arbre parfait de hauteur h possède $2^{h+1} - 1$ nœuds.
4. Donnez `bool est_parfait(arb a)` renvoyant vrai si l'arbre `a` est parfait.
5. Donnez `arb arb_trouve(arb a, int k)` renvoyant le `k`-ème élément dans l'ordre préfixe de l'arbre. On suppose ici que `a` est parfait et que $0 \leq k < n$ avec n le nombre de nœuds.
6. Discutez de la complexité de `arb_trouve` et de potentiels moyens de l'améliorer. On pourra chercher un algorithme en $O(\ln n)$

Arbres 2-dimensionnels

On cherche à créer une structure de données similaire à un arbre binaire de recherche pour des couples de points. On pose le type suivant pour des arbres 2-dimensionnels

```
type tree = F | N of tree * int * int * tree
```

On pose sur $\mathbb{N}^2$ la relation d'ordre $(x, y) \leq_2 (x', y') \Leftrightarrow x + y \leq x' + y'$.

1. Montrer que $\leq_2$ est bien une relation d'ordre bien fondé.
2. Donner la fonction `add (a:tree) (x: int * int): tree` qui ajoute à un arbre binaire de recherche `a` l'élément `x` selon la relation d'ordre $\leq_2$
3. On cherche à obtenir tous les éléments dans l'ordre trié selon la première composante. Quelle est la complexité d'un algorithme qui retourne ça avec notre structure, en supposant l'arbre équilibré ? Et si on avait utilisé la relation d'ordre $(x, y) \leq' (x', y) \Leftrightarrow x \leq x'$ à la place ?

Pour être capable de renvoyer tous les élément dans l'ordre trié selon la première composante (ou la deuxième, au choix), on considère des arbres 2 dimensionels: pour savoir si (x, y) sera fils gauche ou droit de la racine (x', y'), on regarde en fonction de la profondeur de (x', y') :

- Si la profondeur est paire, on compare selon la première composante
- Si la profondeur est impaire, on compare selon la seconde.

4. Donner un arbre binaire 2-dimensionnels contenant les couples $[(1, 3), (5, 1), (6, 0), (2, 4), (7, 5), (0, 6), (3, 2)]$ respectant la définition du dessus.
5. Donner les fonctions d'ajout dans un tel arbre 2-dimensionel.
6. En supposant l'arbre équilibré, quelle est la complexité d'un algorithme qui renvoie la liste des points stoqué dans l'arbre trié selon la première composante ?

Reconstruire l'arbre avec des parcours

Pour T un arbre *binaire* contenant des entiers tous distincts, on note $T_{\text{pref}}, T_{\text{inf}}, T_{\text{post}}$ respectivement les listes des parcours préfixe, infixe et postfixe de T.

On cherche ici à reconstruire T à partir de T_{inf} et T_{post}

Question 2 Montrer que le parcours T_{pref} et T_{suff} ne suffisent pas forcément pour reconstruire T

Question 3 Donner un algorithme récursif pour reconstruire T à partir de T_{pref} et T_{inf}

Question 3 Quelle est la complexité de l'algorithme ?

Question 4 Donner le code d'une fonction C `arb_t *get_arb(int n, int *pref, int *post);` qui retourne l'arbre T associé à T_{inf} et T_{post}

Question 5 Montrer que si l'arbre est localement complet (chaque nœud à 2 ou 0 enfants) alors on peut reconstruire T à partir de T_{pref} et T_{post}

Arbre canonique[1]

On définit une structure d'arbre:

```
type tree = F | N of tree * tree;;
```

Un arbre binaire strict (ou localement complet) est dit canonique si pour A et B deux feuilles, on a A moins profond que B ssi A arrive avant B dans un parcours préfixe.

Un arbre canonique peut-être représenté par un tableau qui à chaque hauteur associe son nombre de feuilles.

Question 1 Donner une définition équivalente de ce type en C. Toujours en C, donnez la fonction `void parcours(arbre* arb)` qui affiche (`print`) le parcours préfixe de `arb`.

Question 2 Donner les arbres canoniques des tableaux $[0; 2]$, $[0; 0; 3; 2]$, $[0; 1; 1; 1; 1; 2]$.

Question 3 Démontrer que le tableau d'un arbre canonique de longueur plus grande que 1 doit se terminer avec un nombre pair.

Question 4 Donner une fonction OCaml `val to_array: tree -> int array` qui à un arbre canonique associe son tableau d'entiers le représentant.

Question 5 Donner une fonction OCaml `val canonical: int array -> tree` qui à un tableau associe son arbre canonique.

---

[1] ENS 2023 MPI Info A

Préfixe vers Suffixe

On se fixe le type suivant pour les arbres:

```
type 'a arb = | V | N of 'a * 'a arb * 'a arb
```

Question 1 Donner le code d'une fonction OCaml `let prefix: 'a arb -> 'a list` qui renvoie le parcours préfixe d'un arbre.

On cherche à modifier un arbre T en un arbre T' tel que le parcours préfixe de T est le parcours suffixe inversé de T'

Question 2 Donner un exemple d'un arbre T qui possède le même parcours préfixe que le parcours suffixe inversé. Donner l'exemple d'un arbre ou ils diffèrent.

Question 3 Donner le code `val to_suff: 'a arb -> 'a arb` tel que `to_suff t` donne un arbre T' tel que le parcours préfixe de T est le parcours suffixe inversé de T'

Question 4 Montrer la correction de votre algorithme

Arbres généraux

On considère les deux type d'arbres suivant en OCaml:

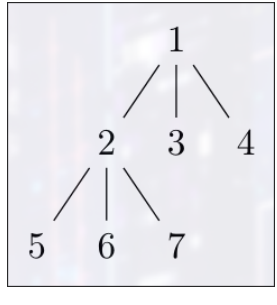
```
type bintree = | V | N of bintree * int * bintree
type tree = | N of int * tree list
```

Les arbres `bintree` représente des arbres binaire tandis que `tree` représente des arbres généraux.

1. Proposer une fonction `val to_tree (a:bintree): tree` tel que `to_tree a` convertit un arbre binaire non vide en un `tree` représentant le même arbre.

Pour transformer un arbre général en arbre binaire on utilisera la méthode *LCRS (Left child, right sibling)* décrite si dessous. Pour x un noeud, on note E_x la liste des enfants de x. Maintenant, pour x un noeud de parent N (avec donc $E_N[i] = x$) on aura que les deux enfants de x dans l'arbre binaire transformé seront à gauche $E_x[0]$ et à droite $E_N[i+1]$.

2. Donner la transformation de l'arbre suivant (on ordonne les enfant de droite à gauche) :



3. Programmer la fonction `val to_bintree (a:tree): bintree`
4. On note $\Delta(A)$ le nombre d'enfants maximal d'un des noeuds de A pour A un arbre général. Montrer que $h(\text{to\_bintree}(A)) \leq \Delta(A) + h(A)$

Tableaux tri-coloré

Soit T un tableau de taille $2N$ dont les valeurs sont dans $\{0, 1, 2\}$. On dit que T est *tricoloré* si

$$\forall 0 \leq i < N, T[i] \neq T[2i+1] \neq T[2i+2] \neq T[i]$$

Dénombrer le nombre de tableau tricoloré de longueur $2^k - 1$

Permutation triable par pile[2]

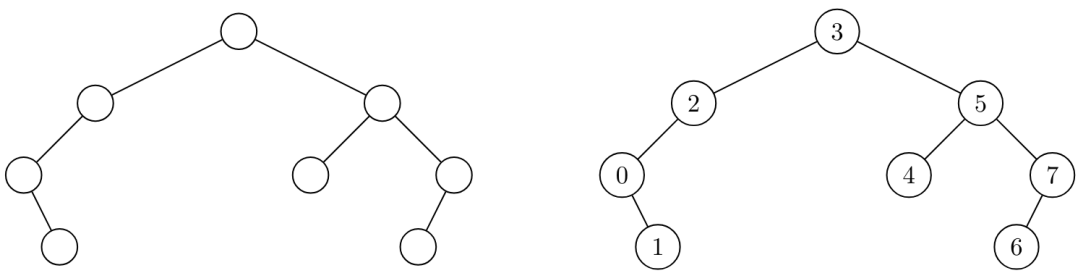
Dans cet exercice on s'interdit d'utiliser les traits impératifs du langage OCaml (références, tableaux, champs mutables, etc.)

On représente en OCaml une permutation σ de $[\![0; n-1]\!]$ par la liste d'entier $[\sigma(0); \sigma(1); ...; \sigma(n-1)]$. 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 :

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

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.

Par example, pour l'arbre suivant (gauche), son étiquettage infixe est donné à droite, et la permutation associé est $[3; 2; 0; 1; 5; 4; 7; 6]$.



1. Ecrire une fonction parcours `val 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.
2. Ecrire une fonction `val 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.
   Indication : on pourra utiliser une fonction auxiliaire de type `arbre -> int -> arbre * int` *qui prend en paramètres un arbre et la prochaine étiquette à mettre et qui renvoie le couple formé par l'arbre étiqueté et la nouvelle prochaine étiquette à mettre.*

Une permutation σ de $[\![0; n-1]\!]$ est dite *triable avec une pile* s'il est possible de trier la liste $[\sigma(0), ..., \sigma(n-1)]$ en utillisant uniquement une structure de pile comme espace de stockage interne. On considère l'algorithme 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 sortie
> **Fin tant que**
> Empiler l'élement en entrée dans la pile
> **Fin pour chaque**
> Dépiler tous les éléments restants dans la pile vers la sortie.

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. On admet qu'une permutation est triable par pile si et seulement cet algorithme permet de la trier correctement.

4. Dérouler l'exécution de cet algorithme sur la permutation associée à l'arbre exemple de la figure 1 et vérifier qu'elle est bien triable par pile
5. Ecrire une fonction `val 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 s'évaluer en la liste $[7; 6; 5; 4; 3; 2; 1; 0]$. On utilisera directement une liste pour implémenter une pile.
   Indication : écrire une fonction auxiliaire de type `int list -> int list -> int list -> int list` *qui prend en paramètre une liste d'entrée, une pile et une liste de sortie, et qui, en fonction de la forme de la liste d'entrée et de la pile, applique une étape élémentaire avant de procèder récursivement.*
6. 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.
7. On se propose de montrer que les permutations de$[\![0; n-1]\!]$ triables par une pile sont en bijection avec les arbres binaires non étiqueté à n nœuds.
   - 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.
   - Montrer qu'une permutation triable par pile est une permutation associée à un arbre binaire.
     Indication : on peut prendre $\sigma(0)$ comme racine, puis procéder récursivement avec les $\sigma(0) - 1$ éléments pour construire le fils gauche et avec le reste pour le fils droit.

[2] Oral blanc MPI CCINP

Initialisation d'un Tas

On encode un tas max sous la forme d'un tableau d'entiers tel que les 2 fils de `T[i]` sont en position `T[2*i]` et `T[2*i+1]`

Question 1 Donner la fonction `void add_tas(int* T, int n, int k);` qui ajoute un entier `k` au tas `T`, représenté par un tableau de longueur `n`.

Question 2 Quelle est la complexité de cette fonction ? Proposer une meilleure structure que des tableaux pour avoir une complexité en moyenne $O(\log n)$

On cherche maintenant à initialiser un corriger un tableau `T` qui n'est **pas** un tas pour qu'il en devienne un.

Question 3 On suppose que pour tous les indices $j > i, T[j] \geq \max(T[2j], T[2j+1])$. Donner un algorithme `void correct(int* T, int n, int i);` qui corrige $T[i]$. Quelle est la complexité de cet algorithme ?

Pour i un noeud, on note $h(i)$ sa hauteur, donnée comme sa distance maximale aux feuilles, aka $h(T) - p(i)$ avec $p(i)$ la profondeur. On considère le code suivant:

```
void to_tas_2(int* T, int n){
  for(int i=n-1; i>=0;i--) {
    correct(T, n, i);
  };
};
```

Question 4 Montrer que

$$\sum_{0 \leq i < N} h(i) = \Theta(n)$$

En déduire que `to_tas_2` est en $\Theta(n)$

Mots d'arbres

On se fixe un ensemble fini de lettres $\Sigma = \{a, b\}$. On définit un *mot* comme étant une suite finie de lettres de Σ. Le mot vide (la suite vide) sera noté ε, et la concaténation de deux mots u, v sera noté par la concaténation uv. Pour w un mot, on note $|w|_a$ le nombre de a et $|w|_b$ le nombre de b. On définit par induction l'ensemble A par:

- $a \in A$
- Si $u, v \in A$, alors $buv \in A$

Question 1 Montrer que $\forall w \in A, |w|_a = 1 + |w|_b$

Question 2 Soit $w_1 ... w_n$ un mot de A, montrer que pour tout $i < n$, on a que $|w_1 ... w_i|_a \leq |w_1 ... w_i|_b$. En déduire que A est non ambiguë.

Question 3 Pour un $w \in A$, on essaye d'associer un arbre binaire T_w tel que T_{buv} soit l'abre contenant T_u et T_v comme enfants. Proposer un algorithme pour effectuer cette transformation.

Question 4 En déduire une manière de stoquer des arbres d'entiers positif sous la forme d'un tableau. Quel est l'avantage par rapport aux représentations que vous connaissez déjà ?

Ranger des boîtes

On cherche à trouver une structure de données efficace pour un magasin de vente de boîte en bois. Chaque boîte a deux paramètre: sa longueur (l) et sa largeur (w), les deux en centimètres.

Question 1 Proposer une définition d'un type `box_t` en C pour représenter une boîte.

On définit un type d'arbre de recherche de boîtes spécial:

```
struct box_arb {
  box_t* box;
  struct box_arb childs[4];
};
typedef struct box_arb box_arb_t;
```

Ou l'idée est que pour E un enfant de P :

- si $E = P.\text{childs}[0]$ alors $E.w < P.w$ et $E.l < P.l$
- si $E = P.\text{childs}[1]$ alors $E.w < P.w$ et $E.l \geq P.l$
- si $E = P.\text{childs}[2]$ alors $E.w \geq P.w$ et $E.l < P.l$
- si $E = P.\text{childs}[3]$ alors $E.w \geq P.w$ et $E.l \geq P.l$

Question 2 Donner un arbre équilibré contenant les boîtes de dimensions $2 \times 8, 4 \times 5, 4 \times 7, 5 \times 8, 7 \times 4$

Question 3 Proposer le code d'une fonction `bool is_box_arb(box_arb_t *arb);` qui teste si un arbre respecte la condition de l'arbre de recherche que l'on souhaite crée.

Question 4 Donner un code C pour effectuer la recherche d'une boîte b dans l'arbre. Quel est sa complexité ?

Question 5 Donner un algorithme pour obtenir la liste trié des boîtes par longueur à partir d'un T. Quelle est sa complexité ?

Arbre et oracle

Soit $A = (S, E)$ un graphe représentant un arbre.

On se fixe une fonction $f : \mathcal{P}(S) \longrightarrow \{\text{true}, \text{false}\}$ qui à tout sous-ensemble de sommets de S qui **forment une composante connexe** renvoie si oui ou non un certain sommet s fixé à l'avance, inconnu pour nous, y appartient.

1. Donner un algorithme qui en $O(|S|)$ appels à f trouve le sommet s.
2. Donner un algorithme qui en $O(\log|S|)$ appels à f trouve le sommet s dans le cas ou l'arbre est un arbre binaire parfait.
3. Donner un algorithme qui en $O(\log|S|)$ appels à f trouve le sommet s dans le cas général.

Indépendant dans les arbres

On se donne en OCaml le type d'arbre suivant stockant des entiers sur chaque noeuds (que on apellera poids) :

```
type abr = F | N of int * abr * abr;;
```

On dit qu'un ensemble de noeuds S d'un arbre est indépendant si aucun n'est enfant direct d'un autre. Autrement dit, pour tout $p \in S$, on a les enfants de p qui ne sont pas dans S.

On dit qu'il est fortement indépendant si pour tout $p \in S$, on a tout le sous arbre de p qui n'est pas dans S.

Question 1 Donner une fonction OCaml `val get_max: abr -> int` qui trouve le poids maximal et le retourne.

Question 2 Montrer que le poids maximal d'un ensemble indépendant est supérieur ou égal au poids maximal d'un ensemble fortement indépendant

Question 3 Proposer un algorithme OCaml qui calcule le poids maximal d'un ensemble fortement indépendant.

Question 4 Proposer un algorithme OCaml qui calcule le poids maximal d'un ensemble indépendant.

Arbre α-équilibré

Pour T un arbre, on note $g(T)$ et $d(T)$ respectivement son fils gauche et droit. On note $|T|$ son nombre de noeud. On dit qu'un arbre binaire de recherche est α-équilibré si pour tout noeud x, on a

$$|g(x)| \geq \alpha |d(x)| \text{ et } |d(x)| \geq \alpha |g(x)|$$

Le type `bintree` représente les arbres usuels. On représente cette donnée par le type suivant en OCaml tel que $\text{EN}(g, x, n, d)$ corresponde à l'arbre T stockant x à la racine avec $|T| = n$, $g = g(T)$ et $d = d(T)$:

```
type 'a bintree = | F | N of 'a bintreee * 'a * 'a bintree
type 'a etree = | EF | EN of 'a etree * 'a * int * 'a etree
```

Question 1 Donner le code d'une fonction `val to_etree: 'a bintree -> 'a etree` qui converti un arbre binaire de recherche en un `etree`. Pour l'instant l'on ne ce soucie pas de la condition d'α-équilibrage

Question 2 Donner le code d'une fonction `val is_balanced: float -> 'a etree -> bool` telle que `is_balanced alpha t` renvoie `true` si t est α-balanced et `false` sinon.

On suppose que on possède une fonction

```
val join: float -> 'a etree -> 'a -> 'a etree -> 'a etree
```

telle que `join alpha t1 x t2` renvoie l'abr α-équilibré contenant les noeuds de t_1, t_2 et x, pour t_1, t_2 deux abr α-équilibré et x plus petit que tous les élements de t_2 et plus grand que tous les éléments de t_1.

Question 3 Donner le code d'une fonction `let split (alpha:float) (t:'a etree) (x:'a etree): 'a etree * 'a etree` telle que pour $\alpha \in]0; 1[$ et t un abr α-équilibré, `split alpha t x` renvoie un couple de deux abr α-équilibré contenant respectivement tous les éléments plus petits que x et tous les éléments plus grand que x.

Question 4 En supposant que `join` est en $O(\log(|t_1| + |t_2|))$, quel est la complexité de `split` ?

Question 5 (*) Pour $\alpha \in]\frac{2}{11}; 1 - \frac{1}{\sqrt{2}}[$, donner le code de `join`.