Les exercices 1 et 2 font parti du cours. Vous devez donc savoir répondre aux questions de ces exercices. Dans tout le TP, les arbres binaires de recherche (ABR) seront représentés en OCaml par le type « `'a abr` ».

```
type 'a abr =
  | Vide
  | N of  'a * 'a abr * 'a abr;;
```

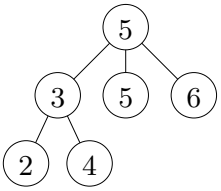

FIGURE 1

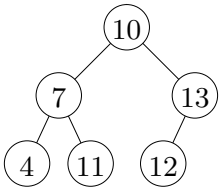

FIGURE 2

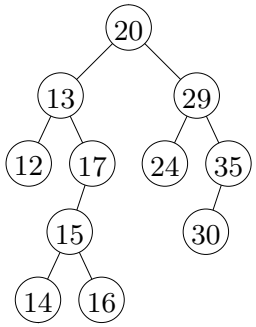

FIGURE 3

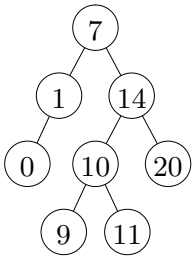

FIGURE 4 – `d1`

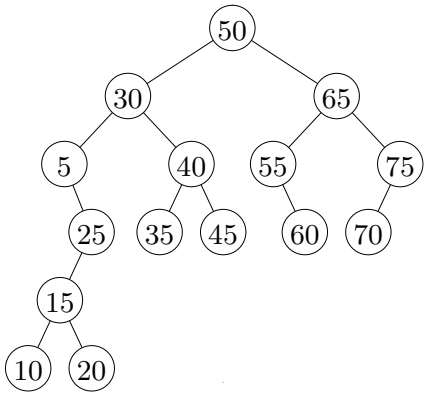

FIGURE 5 – `d2`

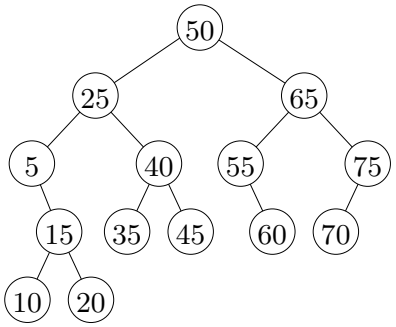

FIGURE 6

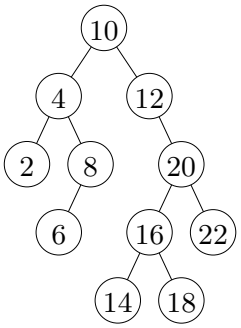

FIGURE 7

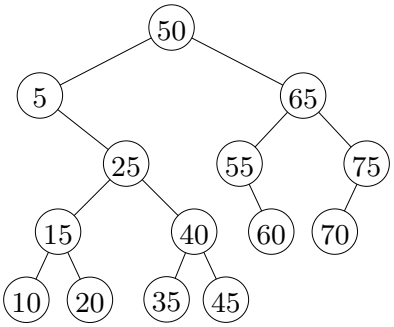

FIGURE 8

Exercice 1.

1. Pour chacune des figures 1, 2, 3, dire s'il s'agit d'un arbre binaire de recherche.
2. Donner un arbre binaire de recherche de hauteur 3 avec 12 noeuds étiquetés par $1, 2, 3, \ldots, 12$.

Exercice 2. Dictionnaires et arbres binaires de recherche

Le but de l'exercice est de construire des dictionnaires à l'aide d'ABR. On rappelle qu'un dictionnaire est une structure de données permettant de stocker des éléments composés d'une clé et d'une valeur. Les clés doivent appartenir à un ensemble totalement ordonné. Les opérations sur les dictionnaires qu'on programmera sont : la création du dictionnaire vide, l'ajout d'un couple (clé, valeur), la recherche de la valeur associée à une clé donnée et la suppression de l'élément associé à une clé. Notez qu'en pratique, les dictionnaires basés sur les tables de hachage sont plus performants que ceux basés sur les ABR.

1. Expliquer comment représenter un dictionnaire avec un ABR. À partir du type « `'a abr` », définir un type `dict` pour représenter les dictionnaires dont les clés sont des entiers et les valeurs des chaînes de caractères. Attention : `dict` doit être une structure de données persistante (statique et non mutable).

Le fichier `TP11-annexe.ml` disponible sur la page du cours contient une fonction `print_dict` permettant d'afficher des arbres. Utilisez cette fonction pour tester vos programmes. Ce fichier contient également les dictionnaires `d1` et `d2`

des figures 4 et 5 à utiliser pour tester vos fonctions. Par soucis de concision, les valeurs des dictionnaires ne sont pas représentées sur les figures du début du sujet, seules les clés y apparaissent.

Création, recherche et insertion d'un élément

2. Écrire une fonction « `create: unit -> dict` » qui renvoie un dictionnaire vide.
3. Écrire une fonction « `find: dict -> int -> string` » qui prend en entrée un dictionnaire `d` ainsi qu'une clé `c` et renvoie la valeur associée à `c` dans `d`. Si `c` n'est pas une clé de `d`, votre fonction déclenchera une erreur. Testez sur l'arbre vide ainsi que sur `d1` et `d2`. Faites des tests pour lesquels la clé appartient à l'arbre et des tests pour lesquels la clé n'appartient pas à l'arbre.
4. Écrire une fonction « `add: dict -> int -> string -> dict` » qui ajoute un couple (clé, valeur) dans un dictionnaire. Si la clé apparaît déjà dans le dictionnaire, la valeur associée sera mise à jour. Sinon, le nouvel élément sera ajouté au niveau des feuilles de l'arbre. Testez votre fonction.

Suppression d'un noeud (méthode 1)

Pour supprimer un noeud dans un arbre binaire de recherche, on utilise la méthode vue en cours. Par exemple, lorsqu'on supprime la clé 30 dans le dictionnaire `d2` de la figure 5, on obtient l'arbre de la figure 6. Dans les questions qui suivent, on pourra supposer que les clés de l'arbre sont distinctes deux à deux.

5. Écrire une fonction « `get_max: dict -> (int * string) * dict` » qui prend en entrée un dictionnaire `d`, et renvoie l'élément de clé maximale de `d` ainsi que le dictionnaire où cet élément a été supprimé. Par exemple, si on note `d3` (resp. `d4`) le sous-arbre de la figure 5 (resp. figure 6) dont la racine est étiquetée par 5, alors « `get_max d3` » vaut `((25,"vingt-cinq"), d4)`.
6. En déduire une fonction « `remove: dict -> int -> dict` » qui prend en entrée un dictionnaire `d` ainsi qu'une clé `c`, et supprime l'élément associé à `c` dans `d`. Votre fonction renverra l'arbre initial si la clé n'apparaît pas dans `d`.
7. Donner une méthode similaire pour supprimer un élément en utilisant la clé minimale du sous-arbre droit à la place de la clé maximale du sous-arbre gauche.

Complexités des fonctions

8. Quelles sont les complexités de vos fonctions ?

Dictionnaires modifiables

Pour terminer, on souhaite créer un autre type `dict2` correspondant à une structure de données dynamique de dictionnaires (contrairement à `dict` qui est persistante).

9. Définir `dict2` ainsi que les fonctions associées. Vous devez réutiliser les fonctions précédentes.

Exercice 3. Tris et arbres binaires de recherche

1. Proposer un moyen de trier une liste d'entiers en utilisant des arbres binaires de recherche.
2. Quelles sont les complexités pire cas et meilleur cas ?

Exercice 4. Nombre d'arbres binaires de recherche

On souhaite compter le nombre d'ABR ayant un nombre de noeuds et une hauteur donnés.

1. Dessiner les 5 ABR ayant pour étiquettes $\{1, 2, 3\}$.

Pour $n \in \mathbb{N}$, on note c_n le nombre d'ABR ayant pour étiquettes $\{1, 2, \dots, n\}$. Les entiers c_n s'appellent les ***nombres de Catalan***.

2. Donner une formule de récurrence pour calculer c_n.
3. Écrire une fonction qui prend en entrée $n \in \mathbb{N}$ et renvoie c_n. Vérifiez que $c_{20} = 6\ 564\ 120\ 420$.

Pour $n \in \mathbb{N}$ et $h \in \{-1\} \cup \mathbb{N}$, on note $u_{n,h}$ le nombre d'arbre binaires de recherche de hauteur h ayant pour étiquettes $\{1, 2, \dots, n\}$.

4. Soit $n \in \mathbb{N}$.
   (a) Trouver un entier M_n tel que pour tout $h \geqslant M_n$, on ait $u_{n,h} = 0$.
   (b) Trouver un entier m_n tel que pour tout $h \leqslant m_n$, on ait $u_{n,h} = 0$.

5. Trouver une formule de récurrence pour calculer $u_{n,h}$.
6. Écrire une fonction qui prend en entrée n et h, et renvoie $u_{n,h}$. Vérifiez que :

$$u_{6,2} = 4, \quad u_{10,-1} = 0, \quad u_{10,2} = 0, \quad u_{10,3} = 116,$$
$$u_{10,9} = 512, \quad u_{10,10} = 0, \quad u_{25,12} = 699\ 971\ 274\ 240.$$

Exercice 5. Successeur dans un ABR

Soit A un arbre binaire de recherche étiqueté par des entiers distincts. Étant donné un entier k, on souhaite déterminer le ***successeur*** de k dans A, c'est à dire la plus petite étiquette de A qui soit strictement supérieure à k. Dans le cas où k est l'étiquette maximale de A, ou bien si k n'est pas une étiquette de A, on considérera que son successeur n'est pas défini.

1. (a) Expliquer comment trouver le successeur de k en commençant par calculer le parcours infixe de A.
   (b) Quelle est la complexité de cette implémentation ?

En fait, il est possible d'obtenir le successeur de k en temps $\mathcal{O}(h)$ où h est la hauteur de l'arbre. L'idée est d'écrire une fonction récursive en traitant séparément chacun des cas suivants :

→ L'arbre A est vide.

→ L'arbre A est non vide. On a alors trois sous-cas :

- La racine est d'étiquette k. On a alors deux sous-sous-cas :
  - Le sous-arbre droit est vide.
  - Le sous-arbre droit est non vide.
- La racine est d'étiquette $e > k$. On a alors trois sous-sous-cas :
  - k ne se trouve pas dans le sous-arbre gauche.
  - k est l'étiquette maximale du sous-arbre gauche.
  - Le successeur de k dans le sous-arbre gauche est k'.
- La racine est d'étiquette $e < k$. On a alors trois sous-sous-cas :
  - k ne se trouve pas dans le sous-arbre droit.
  - k est l'étiquette maximale du sous-arbre droit.
  - Le successeur de k dans le sous-arbre droit est k'.

2. Pour chacun des cas ci-dessus, déterminer le successeur de k dans A. On rappelle que si k n'apparaît pas dans A ou bien si k est l'étiquette maximale de A, alors son successeur n'est pas défini.

Afin de programmer la recherche du successeur en OCaml, on utilise le type « `int abr` » et on définit les deux exceptions `Absent` et `Maximum`. On utilisera l'exception `Absent` pour indiquer que l'étiquette k n'apparaît pas dans A et l'exception `Maximum` lorsque k est l'étiquette maximale de A. On rappelle que pour déclencher l'exception `Absent`, on utilise la syntaxe « `raise Absent` ». Pour rattraper les exceptions, on utilise la syntaxe « `try ... with ...` »

```
exception Absent;;
exception Maximum;;
```

```
let a = try <expression 0> with
        | Absent  -> <expression 1>
        | Maximum -> <expression 2> ;;
```

Avec le programme ci-dessus, OCaml essaye d'abord d'évaluer l'expression `<expression 0>` puis :

- Si `<expression 0>` s'évalue sans erreur, la variable `a` prend la valeur de cette expression.
- Si l'exception `Absent` est déclenchée, la variable `a` prend la valeur de `<expression 1>`
- Si l'exception `Maximum` est déclenchée, la variable `a` prend la valeur de `<expression 2>`.

3. Écrire une fonction « `successeur: abr -> int -> int` » qui prend en entrée A et k, et renvoie le successeur de k dans A. Tester pour l'arbre de la figure 7 et $k \in \{2, 6, 7, 8, 12, 14, 17, 18, 20, 22\}$.

Une autre possibilité pour trouver le successeur consiste à utiliser une fonction auxiliaire qui prend en argument un couple (A', e) où :

→ A' est un sous-arbre de A dont on note r' la racine.

→ e est le plus petit ancêtre de r' dans A qui soit supérieur à r'.

4. Écrire une fonction « `successeur_bis: abr -> int -> int` » qui prend en entrée A et k, et renvoie le successeur de k dans A.

Exercice 6. Suppression d'un noeud (méthode 2)

Une autre procédure possible pour supprimer un noeud d'un arbre binaire de recherche consiste à remplacer le noeud par la fusion du sous-arbre gauche `G` et du sous-arbre droit `D`. Ici, "fusion" signifie que l'on créé un nouvel arbre binaire de recherche contenant toutes les étiquettes de `G` et toutes les étiquettes de `D`. Cette fusion peut être faite de manière simple en exploitant le fait que les étiquettes de `G` sont inférieures aux étiquettes de `D`. En effet, il suffit de repérer dans `G` le noeud d'étiquette maximale (ce noeud se trouve en bas à droite de `G`) et de remplacer son fils droit (qui vaut `Vide`) par `D`. Par exemple, si on souhaite supprimer le noeud d'étiquette 30 dans l'arbre `d2` de la figure 5, on doit fusionner le sous-arbre de racine 5 et le sous-arbre de racine 40. On obtient alors l'arbre de la figure 8.

1. Écrire une fonction « `fusion: int abr -> int abr -> int arb` » qui prend en entrée deux arbres binaires de recherche `g` et `d` tels que toutes les étiquettes de `g` sont inférieures ou égales aux étiquettes de `d`, et renvoie la fusion des deux arbres.
2. En déduire une fonction « `supprimer_bis: int -> int abr -> int arb` » qui supprime une étiquette d'un arbre binaire de recherche. Votre fonction déclenchera une erreur dans le cas où l'étiquette n'apparaît pas dans l'arbre. L'arbre en sortie devra être un arbre binaire de recherche.
3. Donner une autre méthode pour fusionner `G` et `D`. Pour cela, on utilisera le noeud d'étiquette minimale dans `D` au lieu du noeud d'étiquette maximale dans `G`.

Exercice 7. Gestion d'intervalles

```
type ui  = (int*int) list;;
type uid = (int*int) list;;
```

```
type uid2 = | Vide
            | N of (int*int) * uid2 * uid2;;
```

Soit E un ensemble de réels défini comme une union d'intervalles :

$$E = [a_1, b_1] \cup \ldots \cup [a_n, b_n] \text{ où } a_i \leqslant b_i \text{ pour tout } i \text{ (les intervalles ne sont pas forcément disjoints).}$$

On admet qu'il est toujours possible d'écrire E sous la forme d'une union disjointe d'intervalles :

$$E = [a'_1, b'_1] \cup \ldots \cup [a'_p, b'_p] \text{ où } a'_1 \leqslant b'_1 < a'_2 \leqslant b'_2 < \ldots < a'_p \leqslant b'_p.$$

Par exemple, l'ensemble $E = [-2, 1] \cup [0, 2] \cup [-5, -3]$ s'écrit aussi $E = [-5, -3] \cup [-2, 2]$. En OCaml, on utilisera les types `ui` et `uid` pour représenter des unions d'intervalles. Lorsqu'une liste `[(a1, b1); ...; (an, bn)]` est de type `ui`, elle doit vérifier $\texttt{a}_\texttt{i} \leqslant \texttt{b}_\texttt{i}$ pour tout `i`. De plus, lorsqu'une liste `[(a'1, b'1); ...; (a'p, b'p)]` est de type `uid`, elle doit vérifier $\texttt{a}'_1 \leqslant \texttt{b}'_1 < \texttt{a}'_2 \leqslant \texttt{b}'_2 < \ldots < \texttt{a}'_\texttt{p} \leqslant \texttt{b}_\texttt{p}$.

1. (a) Écrire une fonction « `add: uid -> int*int -> uid` » qui prend en entrée un ensemble E ainsi que deux entiers a, b avec $a \leqslant b$, et renvoie l'ensemble $E \cup [a, b]$.
   (b) Quelle est la complexité de votre fonction ?
2. (a) En déduire une fonction « `uid_of_ui: ui -> uid` » qui prend en entrée un ensemble représenté par le type `ui`, et renvoie sa représentation par le type `uid`.
   (b) Quelle est la complexité de votre fonction ?

Partie facultative. On souhaite maintenant réécrire la fonction `uid_of_ui` en passant par des arbres binaires. Pour représenter une liste `[(a1, b1); ...; (an, bn)]` de type `uid`, on utilisera un arbre de type `uid2` contenant `n` noeuds et dont les étiquettes sont tous les couples $(\texttt{a}_\texttt{i}, \texttt{b}_\texttt{i})$. De plus, l'arbre devra vérifier deux propriétés :

- Soit (a, b) l'étiquette d'un noeud et G son sous-arbre gauche. Alors toute étiquette (a', b') dans G vérifie $b' < a$.
- Soit (a, b) l'étiquette d'un noeud et D son sous-arbre droit. Alors toute étiquette (a', b') dans D vérifie $b < a'$.

3. Soit E un ensemble et a un réel. On définit l'ensemble E' et le réel a' par :
   - Si $a \in E$, alors $a' = a''$ et $E' = E \cap \,]-\infty\,; a''[$ où $[a'', b'']$ est l'intervalle de E contenant a.
   - Si $a \notin E$, alors $a' = a$ et $E' = E \cap \,]-\infty\,; a[$.

   Écrire une fonction « `coupe_gauche: uid2 -> int -> uid2 * int` » qui prend entrée E et a, et renvoie le couple (E', a').
4. Soit E un ensemble et $a \leqslant b$ deux réels. Écrire une fonction « `add2: uid2 -> int*int -> uid2` » qui prend en entrée E ainsi que le couple (a, b), et renvoie $E \cup [a, b]$.
5. (a) En déduire une fonction « `uid_of_ui_bis: ui -> uid` » qui prend en entrée un ensemble représenté par le type `ui`, et renvoie sa représentation par le type `uid`.
   (b) Quelle est la complexité de votre fonction ?