

Les énoncés des travaux pratiques seront disponibles sur internet quelques jours avant la séance correspondante, à l'adresse indiquée ci-dessus. Les questions en rapport avec les TP, Caml Light ou l'informatique sont les bienvenues et peuvent être envoyées à victor.nicollet@ens.fr

Un arbre est un objet mathématique défini récursivement de la manière suivante: un arbre est une liste d'arbres (qu'on appelle ses sous-arbres). Ici, nous travaillerons sur des arbres binaires: ceux-ci sont soit des feuilles (une liste vide) soit des noeuds (une liste de deux sous-arbres binaires, le gauche et le droit).

1 ARBRE BINAIRE DE RECHERCHE

Un arbre binaire de recherche est soit une feuille, soit une paire d'arbres binaires de recherche associée à un entier. Tous les entiers associés au sous-arbre gauche sont inférieurs à cet entier, et tous les entiers associés au sous-arbre droit sont supérieurs à cet entier. On représentera les arbres binaires de recherche à l'aide du type ci-contre.

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

Ainsi, un arbre binaire de recherche est intéressant pour des recherches dichotomiques, car celles-ci ne nécessitent qu'un nombre limité de comparaisons avant de trouver un élément. On appellera profondeur d'un arbre binaire de recherche le nombre maximal de comparaisons nécessaires pour déterminer si un élément arbitraire se trouve dans cet arbre.

▷ **Question 1.** Écrire une fonction `profondeur` qui calcule la profondeur d'un arbre binaire de recherche. Écrire une fonction `valeurs` qui renvoie la liste triée des valeurs se trouvant dans l'arbre. ◀

Pour construire l'arbre, on travaille à l'aide d'une opération appelée insertion. Celle-ci consiste à remplacer un arbre par un nouvel arbre contenant l'élément inséré. Si l'arbre est une feuille, il est remplacé par une paire de feuilles associée à l'entier inséré. Si l'arbre est un noeud, alors le bon sous-arbre est modifié pour contenir l'élément.

▷ **Question 2.** Écrire une fonction `insere` qui insère un élément dans un arbre binaire de recherche. Écrire une fonction `cherche` qui retourne vrai si et seulement si l'arbre binaire de recherche contient un certain entier. ◀

Il est également possible, quoique plus difficile, de supprimer des valeurs de l'arbre. La difficulté consiste à remplacer la valeur supprimée par une autre. C'est facile si celle-ci le sous-arbre droit est une feuille (il suffit alors de remplacer par son sous-arbre gauche). Si le sous-arbre droit n'est pas une feuille, alors le seul candidat est valeur minimale du sous-arbre droit, qu'il faut retirer à cet arbre (facile, puisque cette valeur a une feuille comme sous-arbre gauche) et replacer au bon endroit dans l'arbre.

▷ **Question 3.** Écrire une fonction `supprime_minimal` qui extrait la valeur minimale d'un arbre, et renvoie cette valeur et l'arbre privé de cette valeur. En déduire une fonction `supprime` qui supprime de l'arbre une valeur donnée. ◀

2 ARBRE BINAIRE ALÉATOIRE

On choisit maintenant une structure d'arbre binaire simple, sans y associer d'entiers. On appelle *taille* d'un tel arbre le nombre d'éléments `Deux` dans cet arbre (de plus, si un arbre est de taille n , il contient $n + 1$ éléments `Zero`). On se fixe pour objectif de construire aléatoirement un arbre de taille n donnée, de façon à ce que la loi de probabilité soit uniforme (chaque arbre a la même probabilité d'être choisie que les autres).

```
type alea =
  | Zero
  | Deux of alea * alea
```

Pour cet algorithme, on a besoin d'être capable de compter combien il existe d'arbres de taille n : nous allons commencer par des résultats théoriques à ce propos, en établissant une bijection entre les arbres

binaires de taille n et les mots de Dyck de taille $2n$, et prouver que ce nombre correspond à la suite de Catalan:

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

On note que, pour la calculer plus simplement, on peut utiliser la relation de récurrence:

$$C_{n+1} = \frac{2(2n+1)}{n+2} C_n \quad C_0 = 1$$

Un mot de Dyck de taille $2n$ est une permutation du mot $a^n b^n$ telle que chaque préfixe de ce mot contienne au moins autant de a que de b (préfixe de Dyck). Si on assimile a et b à des parenthèses ouvrante et fermante, les mots de Dyck sont des suites de parenthèses correctement associées.

On se donne une transformation I des permutations de $a^n b^n$ qui ne sont pas des mots de Dyck: on prend le plus grand préfixe de Dyck p de ce mot, qui s'écrit alors pbq . Alors $I(pbq) = pbq'$ où q' est le mot où chaque b de q a été remplacé par un a et vice-versa. I est à image dans les permutations de $a^{n-1} b^{n+1}$.

▷ **Question 4.** Montrer que I est une bijection. En déduire que la taille de l'ensemble des mots de Dyck de taille $2n$ est le nombre de Catalan C_n . ◀

On considère maintenant l'algorithme de construction J , qui lit un mot de Dyck lettre par lettre et construit un arbre à partir d'un noeud initial. En rencontrant un a , J ajoute un fils au noeud courant, et ce fils devient le nouveau noeud courant. En rencontrant un b , J interrompt définitivement le travail sur le noeud courant et remonte au père de ce noeud, qui devient le nouveau noeud courant. Le résultat de J est donc un arbre quelconque d'au moins un noeud.

▷ **Question 5.** Montrer que J est une bijection. Trouver une bijection permettant de transformer un arbre quelconque de $n+1$ noeuds en un arbre binaire de n noeuds et $n+1$ feuilles. En déduire qu'il y a C_n tels arbres binaires. ◀

Nous sommes maintenant capables de compter le nombre d'arbres d'une taille donnée. On construit un arbre de taille n en construisant deux sous-arbres aléatoires de taille i et $n-i-1$, rendant la construction récursive. Il faut seulement faire attention à choisir i de manière uniforme.

▷ **Question 6.** Écrire une fonction pour calculer c_n . Utiliser la fonction `random__int` pour tirer un entier aléatoire, et construire récursivement un arbre aléatoire de taille n . ◀

3 QUESTION DIFFICILE

Le problème de la représentation des arbres binaires de recherche vue dans la première partie est que n'importe quelle modification de l'arbre demande de recréer une grande quantité d'objets, alors même que souvent les modifications sont très limitées, voir même n'impliquent pas la création de nouveaux objets.

Pour cette raison, on se propose de travailler à l'aide de structures mutables (soit des champs mutables, soit des références), par exemple en représentant les arbres avec le type produit ci-contre. L'intérêt d'utiliser des références plutôt que des labels mutables pour `gauche` et `droite` est de pouvoir considérer qu'un arbre est en réalité un objet de type `m_arbre option ref`, et donc de pouvoir écrire des fonctions travaillant sur ce type.

```
type m_arbre =
{
  gauche : m_arbre option ref;
  droite : m_arbre option ref;
  mutable valeur : int;
};;
```

▷ **Question 7.** Réécrire les fonctions de la première partie, à savoir `profondeur`, `valeurs`, `insérer`, `chercher`, `supprimer_minimal` et `supprimer`, avec ce nouveau type d'arbre. Faire en sorte que chaque opération ne crée de nouveaux objets de type `m_arbre` que si c'est absolument nécessaire. ◀