Arbres préfixe et arbres des suffixes

Vincent Picard, enseignant CPGE au Lycée Claude Fauriel

Nous présentons les arbres préfixe servant à représenter de manière relativement compacte un ensemble de mots. Nous introduisons ensuite un cas particulier qui est l'arbre des suffixes d'un texte. Les arbres préfixe et les arbres des suffixes sont des structures de données utiles et très utilisées en informatique : auto-complétion, recherche dans un texte, compression, *etc*.

Nous commençons par écrire cette petite fonction utile.

Question 1

Écrire une fonction `liste_lettres : string -> char list` décomposant un mot en une liste de lettres.

1 Dictionnaire

Nous aurons besoin de la structure de donnée abstraite de dictionnaire. Il existe de nombreuses solutions pour l'implémentation concrète : tables de hachage, arbres binaires de recherche, *etc*. Pour des raisons de simplicité, nous utiliserons la structure de liste associative avec clefs triées.

Ainsi, le dictionnaire sera une liste de couples (clef, valeur) qui sera triée par ordre croissant de clefs. On utilisera l'interface suivante

```
type ('a, 'b) dico = ('a * 'b) list
dexiste : ('a, 'b) dico -> 'a -> bool
dajoute : ('a, 'b) dico -> 'a -> 'b -> ('a, 'b) dico
dtrouve : ('a, 'b) dico -> 'a -> 'b
dchange : ('a, 'b) dico -> 'a -> 'b -> ('a, 'b) dico
dsupprime : ('a, 'b) dico -> 'a -> ('a, 'b) dico
```

On pourra supposer que le dictionnaire ne contient pas plusieurs occurrences d'une même clef. Lorsqu'une fonction prend en argument une clef qui n'existe pas dans le dictionnaire, l'exception `Not_found` sera déclenchée à l'aide de la commande `raise Not_found`. On essaiera d'optimiser le code en utilisant bien le fait que les clefs de la liste sont triées (elles seront comparées avec l'opérateur polymorphe `<=`).

Question 2

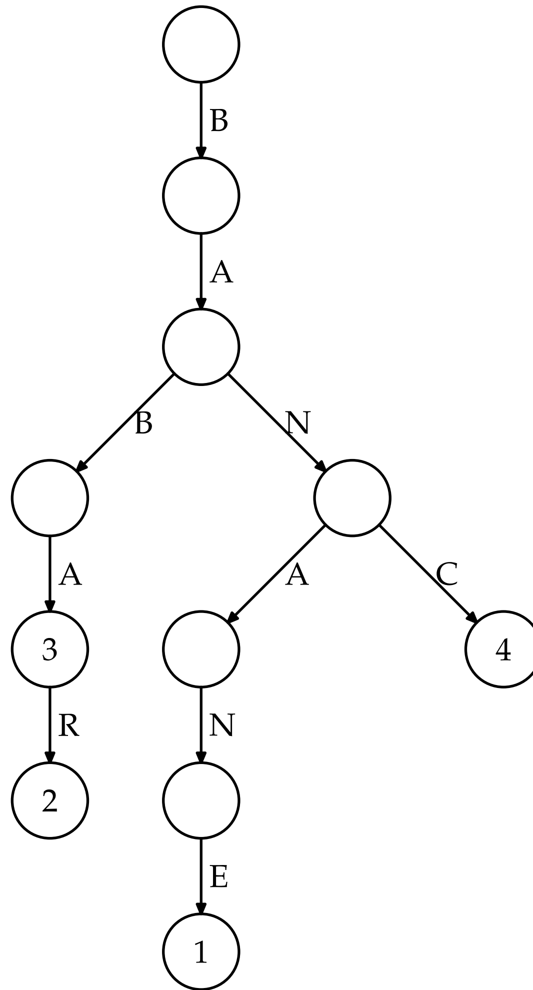
Programmer la structure de données dictionnaire.

On pourra vérifier le bon fonctionnement à l'aide du code suivant :

```
let d = make_dico ();;
let d1 = dajoute d 'C' 10;;
let d2 = dajoute d1 'A' 20;;
let d3 = dajoute d2 'B' 30;;
dexiste d3 'B';;
dexiste d3 'Z';;
dtrouve d3 'C';;
let d4 = dchange d3 'C' 40;;
dtrouve d4 'C';;
let d5 = dsupprime d4 'C';;
dexiste d5 'C'
```

2 Arbre préfixe

Un arbre préfixe est un arbre enraciné n-aire permettant de stocker un ensemble fini de mots. Les arêtes sont étiquetées par des lettres. Les mots sont obtenus en lisant les lettres rencontrées lorsqu'on descend dans l'arbre. Voici l'arbre préfixe correspondant aux mots `BANANE`, `BABAR`, `BABA`, `BANC`.



On voit que certains nœuds sont numérotés pour signaler la présence d'un mot (chaque mot à un numéro différent). Les nœuds sans numéro ne correspondent pas un mot (mais à un préfixe d'un ou plusieurs mots représentés).

Pour représenter un tel arbre on utilisera le type suivant :

```
type ptree =
    | Feuille of int
    | Noeud of int * (char, ptree) dico
```

On constate l'utilisation du dictionnaire pour représenter l'ensemble des fils d'un nœud. Les clefs du dictionnaire représentent la lettre sur l'arête et la valeur le sous-arbre correspondant. Les feuilles et les nœuds peuvent avoir un numéro. Lorsqu'il n'y a pas de numéro on utilisera la valeur `-1`.

Les fonctions suivantes manipuleront les mots sous forme de listes de lettres.

```
type mot = char list
```

On donne la fonction suivante permettant de créer un arbre préfixe représentant l'ensemble de mots vide :

```
let make_ptree () = Feuille (-1)
```

Question 3

Écrire une fonction `ptrouve : ptree -> mot -> int` cherchant si un mot appartient à un arbre prefixe et si oui retournant le numéro correspondant. Dans le cas contraire l'exception `Not_found` est déclenchée.

Question 4

Écrire une fonction `pajoute : ptree -> mot -> int -> ptree` ajoutant un nouveau mot dans un arbre prefixe en lui associant le numéro passé en argument. Cette fonction fournit un nouvel arbre prefixe.

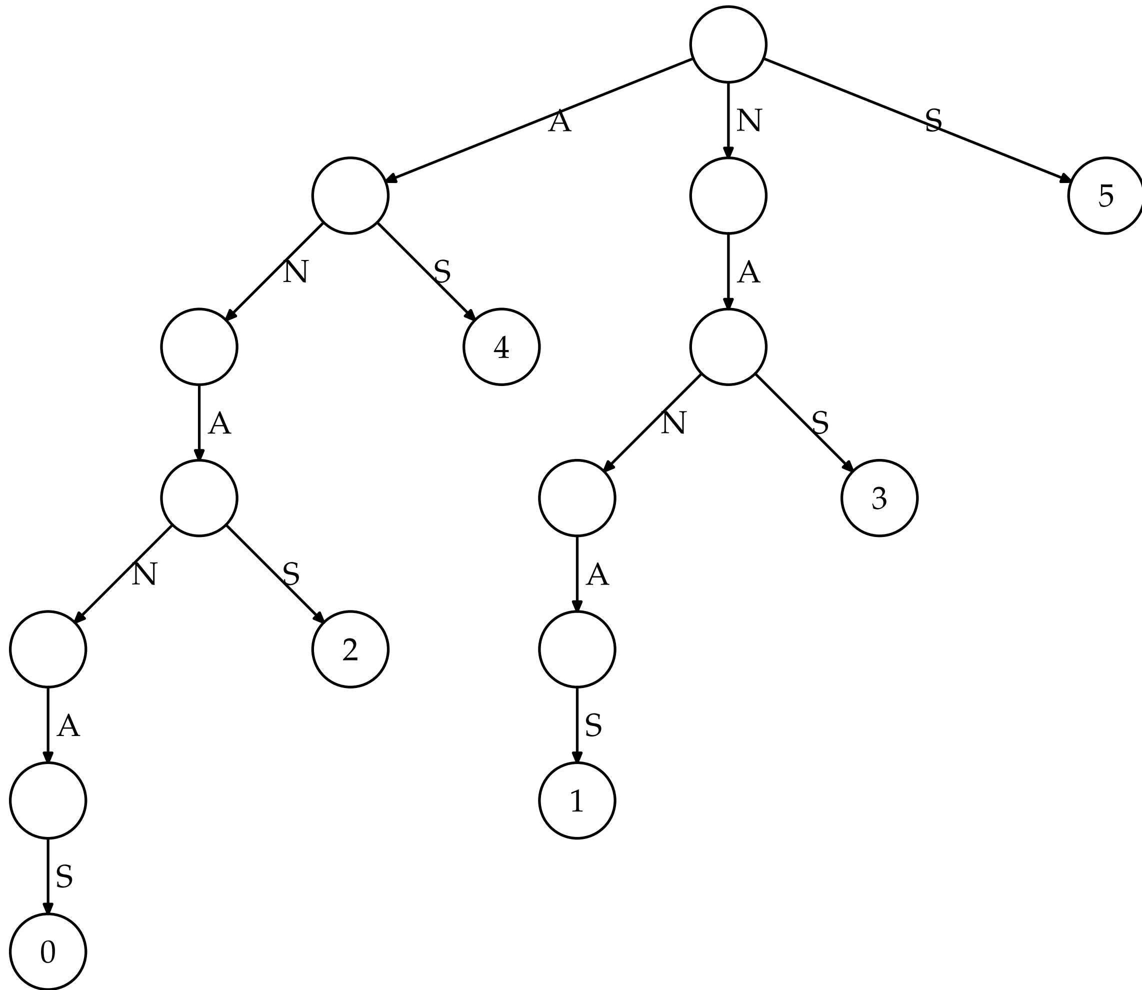
On pourra vérifier que le programme suivant construit bien l'arbre préfixe donné en exemple.

```
let p1 = make_ptree ();;
let p2 = pajoute p1 (liste_lettres "BANANE") 1;;
let p3 = pajoute p2 (liste_lettres "BABAR") 2;;
let p4 = pajoute p3 (liste_lettres "BABA") 3;;
let p5 = pajoute p4 (liste_lettres "BANC") 4;;
```

On voit bien comment un tel arbre peut servir à écrire un algorithme d'autocomplétion. Si on commence à lire les lettres `BAB`, il suffit de regarder tous les numéros dans le sous-arbre correspondant pour obtenir la liste des propositions de complétion.

3 Arbre des suffixes

L'arbre des suffixes d'un mot (ou d'un texte) est l'arbre préfixe de tous ses suffixes propres. Voici par exemple l'arbre des suffixes du mot `ANANAS` :



Dans cet arbre chacun des suffixes a été numéroté du plus grand au plus petit de 0 à $n-1$ où n est la longueur du mot. Le numéro correspond donc aussi à l'indice de la première lettre du suffixe dans le mot.

Question 5

Écrire une fonction `make_stree : string -> ptree` construisant l'arbre des suffixes d'un mot. On s'aidera évidemment des fonctions précédentes.

Question 6

Définir de manière simultanée (avec la syntaxe `let rec a1 = b1 and a2 = b2`) les fonctions

```
numeros_arbre : ptree -> int list
numeros_liste : ptree list -> int list
```

retournant la liste des numéros présents dans un `ptree` ou dans une liste de `ptree`. Pour cette question on pourra supposer que l'on connaît la représentation du dictionnaire sous forme de liste associative.

Question 7

En déduire une fonction `occurrences : ptree -> string -> int list` prenant en entrée un arbre des suffixes d'un texte, un motif et retournant la liste des positions des occurrences exactes du motif dans le texte. Que dire de la complexité de cette fonction par rapport à un algorithme de recherche classique ?

Les arbres des suffixes sont très utilisés en bioinformatique. Voici par exemple une recherche de motifs exacts au sein d'une séquence ADN :

```
let adn = "ATTGCATATAGCACTATAGCATATGCTATAGCTAT";;
let adn_sf = make_stree adn;;
occurrences adn_sf "TATA";;
occurrences adn_sf "TA";;
```

Notre algorithme de construction de l'arbre des suffixes et très élémentaire et coûteux mais il existe des algorithmes (complexes) de construction fonctionnant en temps linéaire par rapport à la longueur du texte.