## Un arbre ternaire

Nous avons exploré les séquences d'éléments avec des structures de données immuables `List` et `Queue`, mais une autre classe de structure de donnée n'a pas été abordée : les tables associatives.

Comme son nom l'indique une table associative lie une clé à une valeur. Fournir la clé permet d'obtenir la valeur.

Posons nous la question d'une représentation efficace d'un *index* de table associative. Pour simplifier le problème nous allons travailler avec des clés qui sont toujours représentées par des chaînes de caractères en minuscules contenant uniquement des lettres.

### Arbres n-aires

Une première technique pourrait consister à coder nos clés par avec un *arbre de préfixes*. L'idée consiste à représenter chaque nœud de l'arbre sous la forme d'un tableau de 26 entrées représentant les 26 lettres de l'alphabet.

Le nœud racine représente la première lettre des clés. Les nœuds enfant de la racine les secondes lettres, etc. Chaque case identifie donc une lettre d'une clé et pointe vers un autre nœud pour la lettre suivante si la lettre apparaît dans le clé. On associe aux lettres un deuxième tableau de 26 valeurs stockées avec les clés (ou un marqueur s'il n'y a pas de valeur). Voici un exemple avec les clés "bug" et "buz" et "bol" :

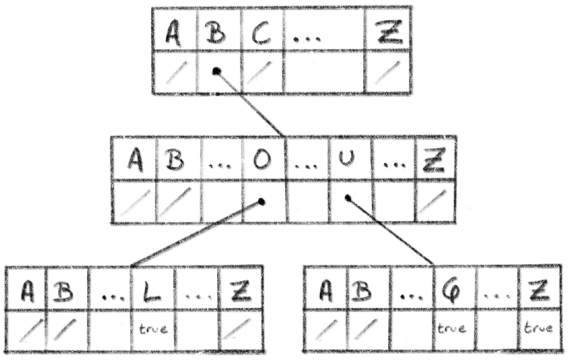


**Figure 1:** Arbre n-aire

La recherche et l'insertion dans de tels arbres sont suffisamment rapides pour les rendre très attractifs, et leur représentation sous forme de `case class` immuable certainement très pratique.

Cependant, il est facile de voir qu'une telle structure consomme rapidement énormément d'espace.

Par exemple avec notre alphabet de 26 symboles, et des clés de 5 caractères, l'arbre pourrait contenir jusqu'à 26^5 nœuds (à peu près 12 millions…)… ce qui le rend donc impraticable pour nos besoins.

### Arbres ternaires

Cependant, nous pourrions économiser de l'espace, au détriment d'une petite perte de temps lors de la recherche et de l'insertion en utilisant des nœuds non pas de 26 entrées, mais seulement de trois. On appelle donc la structure de donnée un *arbre ternaire*.

Le principe consiste à noter que certaines cases des nœuds de l'arbre précédent ne pointent sur rien. Et ceci devient de plus en plus vrai quand on descend dans l'arbre. On va donc *compresser* ce dernier en représentant dans les nœuds en tout cinq éléments :

1. La lettre associée à un indice dans la clé (la première lettre pour la racine, etc.);
2. La valeur stockée (s'il y en a une);
3. un pointeur `left` vers une lettre précédente dans l'alphabet (pas forcément directement précédente), mais au même indice dans la clé;
4. un pointeur `right` vers une lettre suivante dans l'alphabet, mais au même indice dans la clé.
5. un pointeur `next` vers la lettre d'indice suivant de la clé.

Voici un exemple stockant uniquement une clé, "chien", puis après l'ajout de "chat", puis "coq" et enfin "pie" :

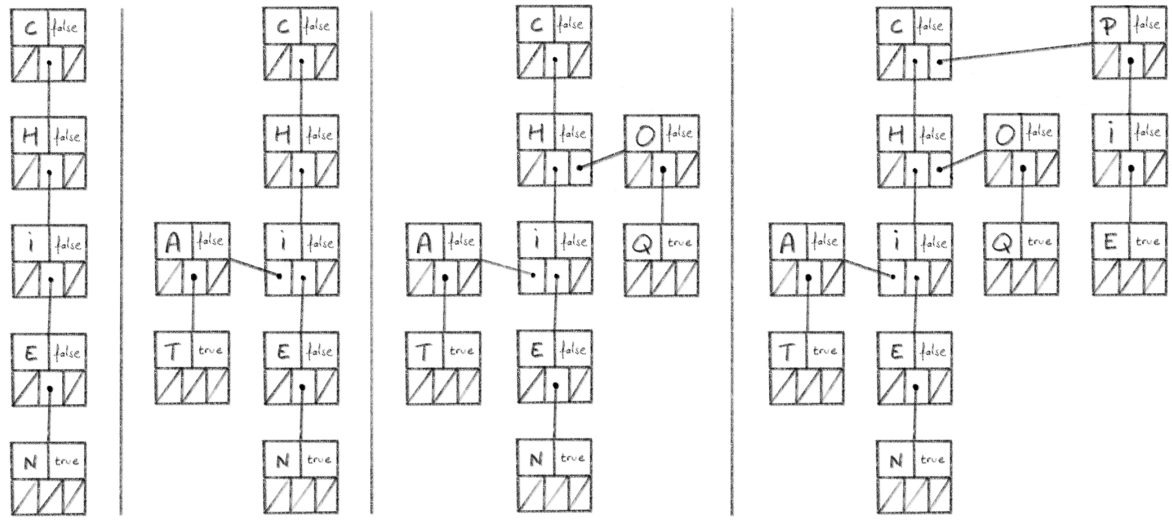


**Figure 2:** Arbre ternaire

On associe à chaque clé la valeur **`true`** pour indiquer simplement qu'elle existe, sinon **`false`**.

### Implantation

Une fois de plus nous allons implanter la structure de donnée sous la forme d'une interface `trait Tree`, et de deux sous éléments : une **`case class`** `Node` pour les nœuds et un **`case`** `object Leaf` pour indiquer un lien vide :

```
sealed trait Tree[+A]
case object Leaf extends Tree[Nothing]
case class Node[A](value:Option[A],
                   char:Char,
                   left:Tree[A],
                   next:Tree[A],
                   right:Tree[A]) extends Tree[A]
```

Notez les cinq champs de `Node`. Le champ `value` est une option puisqu'un nœud peut ne pas contenir de valeur. Les objets `Leaf` ne servent qu'à indiquer qu'il n'y a pas de pointeur pour les champs `left`, `right` et `next`. Comparez les au `Nil` des listes, il est indiqué par des cases barrées dans les diagrammes ci-dessus.

Nous allons placer les méthodes générales dans le `trait Tree`, mais comme nous allons souvent "itérer" sur les caractères de la clé, ces méthodes ne seront qu'une redirection vers les fonctions récursives ayant besoin de plus de paramètres au sein d'un `object Tree` compagnon du `trait`. C'est une technique que l'on utilise souvent pour cacher des paramètres utilitaires seulement.

Ajoutons une méthode `insert` dans le trait :

```
sealed trait Tree[+A] {

    import Tree._

    // Associe la valeur `value` à la clé `key` dans l'arbre.
    def insert[B >: A](key:String, value:B):Tree[B] = Tree.insert(this,
        key, value, 0)
}
```

Qui redirige vers une fonction de l'objet compagnon :

```
case object Tree {

    def apply[A]():Tree[A] = Leaf

    def insert[A](root:Tree[A], key:String, value:A, n:Int):Tree[A] =
       root match {
        case Leaf =>
            insert(Node(None, key.charAt(n), Leaf, Leaf, Leaf), key,
               value, n)
        case node:Node[A] if(node.char > key.charAt(n)) =>
            node.copy(left = insert(node.left, key, value, n))
```

```
10          case node:Node[A] if(node.char < key.charAt(n)) =>
11              node.copy(right = insert(node.right, key, value, n))
12          case node:Node[A] if(n < key.length - 1) =>
13              node.copy(next = insert(node.next, key, value, n + 1))
14          case node:Node[A] =>
15              node.copy(value = Some(value))
16
17      }
18  }
```

Nous avons aussi au passage ajouté une méthode `apply()` permettant de créer un arbre vide.

En programmation fonctionnelle il n'est bien sûr pas possible de modifier les champs d'une classe. Vous savez que les **`case class`** sont en effet immuables. La méthode `copy()` est définie automatiquement sur ces classes et permet de faire un clone de l'objet en spécifiant au passage un ou plusieurs champs de ce clone avec de nouvelles valeurs. Ainsi créer un clone d'un `Node` en ne changeant que le champ `value` se fait avec `node.copy(value = uneNouvelleValeur)`.

Notez aussi que `insert()` dans l'objet `Tree` reçoit un paramètre `n` qui sert à indiquer à quelle "profondeur" dans la clé nous sommes (et donc dans l'arbre). C'est la raison pour laquelle nous n'avons pas implanté `insert` dans le trait `Tree`. L'utilisateur final n'a pas à gérer les indices des clés.

Voici un début de code permettant de tester cette structure d'arbre :

```
1  object TestTree {
2      def main(args:Array[String]):Unit = {
3          val tree = Tree[Boolean]().insert("chien", true).insert("chat",
               true).insert("coq", true).insert("pie", true)
4
5          println(tree)
6      }
7  }
```

Nous allons stocker des booléens dans l'arbre, la valeur vaut "vrai" si ce nœud est la dernière lettre d'une clé valide.

### Travail demandé

Répondez aux questions suivantes, justifiez :

1. Avec la structure de démonstration stockant des booléens **`true`** pour indiquer la présence d'un objet nous simulons un ensemble (`Set`, sinon on simule une `Table` d'association). Ajoutez une méthode `insert(root:Tree[Boolean], key:String)` dans l'objet `Tree` qui associe automatiquement la valeur **`true`** avec une clé. Elle considère que l'on commence toujours à la racine de l'arbre. Avec cet ensemble, obtiendrons-nous un jour la valeur **`false`** ? Que se passe-t'il si on demande une clé non présente ?

2. Implantez une méthode `size` qui donne le nombre de clé/valeurs stockées (pas le nombre de nœuds).
3. Implantez une méthode `toList` qui retourne une liste des valeurs stockées. Note: vous pouvez concaténer des listes entre elles avec l'opérateur ++. Notez aussi que `Nil ++ Nil` donne `Nil` .
4. Ajoutez un accesseur `get` retournant la valeur associée à une clé passée en paramètre. Que faire si la clé n'existe pas ?
5. Pour simuler un `Set` ajoutez une méthode `contains` qui prend en paramètre une clé et retourne un booléen **`true`** si la clé existe.
6. Implantez une méthode `toKeyValueList` qui retourne une liste de tuples (clé, valeur).
7. Implantez une méthode `remove` qui retire un valeur dont la clé est passée en paramètre. Cette méthode doit retourner l'élément supprimé si il existe, ainsi qu'une référence vers la nouvelle valeur d'arbre modifié. Après une suppression il faut souvent réorganiser l'arbre pour une meilleure performance. Nous allons nous passer de cette optimisation, et donc la réorganisation n'est pas demandée. Cependant il ne doit pas rester de branches *mortes* (c'est-à-dire ne menant à aucune valeur). Voici quelques questions que l'on peut se poser : L'élément supprimé est il la seule valeur à retourner ? Que faire si on fait deux ou trois suppressions de suite ? Que faire si la valeur n'existe pas ? Peut-on supprimer des éléments qui ne sont pas en bas de l'arbre ? Peut-il rester des lettres qui ne sont pas utilisées dans un mot après suppression ? Cette méthode n'est pas triviale à mettre en place, expliquez en détail son fonctionnement.

Toutes les fonctions et méthodes doivent être *pures*, et si possible *totales*.

Complétez le `main()` de `TestTree` afin de tester exhaustivement chaque fonction et méthode implantée.

Le code doit être commenté, vous pouvez répondre aux questions dans les commentaires du code.