Chapitre 3 : Tables

3.1 Définition et terminologie

Une **table** est une structure de donnée qui permet d'associer une **clé** (ou entrée) à une **valeur** (pas nécessairement un nombre). Exemples : association d'un mot à sa définition, association d'un numéro de carte bleue à un compte bancaire, association d'une personne à un ensemble de numéros de téléphone.

Les opérations principales d'une table permettent de :

- consulter la valeur associée à une clé,
- ajouter une clé et sa valeur associée,
- vérifier l'existence d'une clé,
- supprimer une clé et sa valeur associée.

Des opérations supplémentaires permettent de consulter les informations relatives à la taille de la table : consulter le nombre de clés et tester si celle-ci est vide.

Cette abstraction donne lieu à la définition du type abstrait Dictionary[K,V].

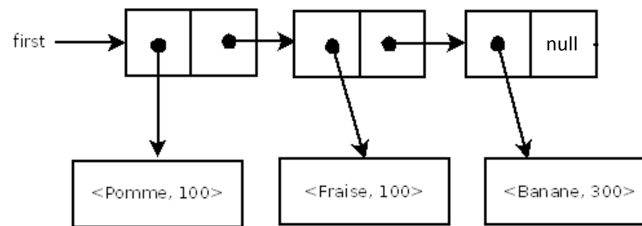
3.2 Spécification algébrique

```
Type Dictionary[K,V]

Opérations

       empty : -> Dictionary[K,V] -- créer une table vide

       add : Dictionary[K,V] X K X V -> Dictionary[K,V]
                                 -- ajouter une clé avec sa valeur

       has : Dictionary[K,V] X K -> Boolean
                                 -- vrai si la clé est présente

       value : Dictionary[K,V] X K -> V      -- valeur attachée à une clé

       size : Dictionary[K,V] -> Integer     -- nombre de clés

       isEmpty : Dictionary[K,V] -> Boolean  -- table vide ?

       delete : Dictionary[K,V] X K -> Dictionary[K,V]
                                 -- supprimer une clé et sa valeur
```

> ☺ **Exercice 3.A :** identifier les opérations internes et les observateurs du type Dictionary[K,V], puis compléter la spécification algébrique du type Dictionary[K,V] en indiquant les préconditions et les axiomes correspondants.

3.3 Implantations de Dictionary[K,V]

L'implantation d'une table peut être envisagée sous de multiples facettes. Au premier abord, nous pouvons imaginer représenter les couples <clé, valeur> dans une liste simplement chaînée. L'opération add se fait en temps constant, alors que les opérations value, has et delete exigent une itération qui s'exécute en un temps proportionnel à la taille de la table.



On diminue le temps de recherche (pour value, has et delete) si la liste est maintenue triée (à condition de disposer d'un ordre total sur les clés) ; en contrepartie, l'opération add ne se fait plus en temps constant, puisqu'il faut trouver la bonne place où insérer. On peut améliorer un peu cette implantation si on utilise une liste doublement chaînée, avec un rang courant, car alors, le parcours peut se faire dans les deux sens ; on y perd bien sûr en espace mémoire.

> ☺ **Exercice 3.B :** soient l'interface `Dictionary<K,V>`, la classe abstraite `AbstractDictionary<K,V>` et la classe `Association<K,V>` permettant de représenter les couples <clé, valeur> décrites ci-dessous. Ecrire brièvement les méthodes d'ajout et suppression de la classe `ListDictionary<K,V>` qui hérite de la classe abstraite `AbstractDictionary<K,V>` et utilise une `ArrayList` pour stocker les associations.

```
public interface Dictionary<K, V> {

    public abstract void add (K key, V value) ;
    public boolean has (K key) ;
    public abstract V value (K key) ;
    public abstract boolean isEmpty () ;
    public abstract int size() ;
    public abstract void delete (K key) ;

}
```

```
public abstract class AbstractDictionary<K, V> implements  Dictionary<K, V>
{
      public boolean isEmpty () {return (size()==0) ;}
}
```

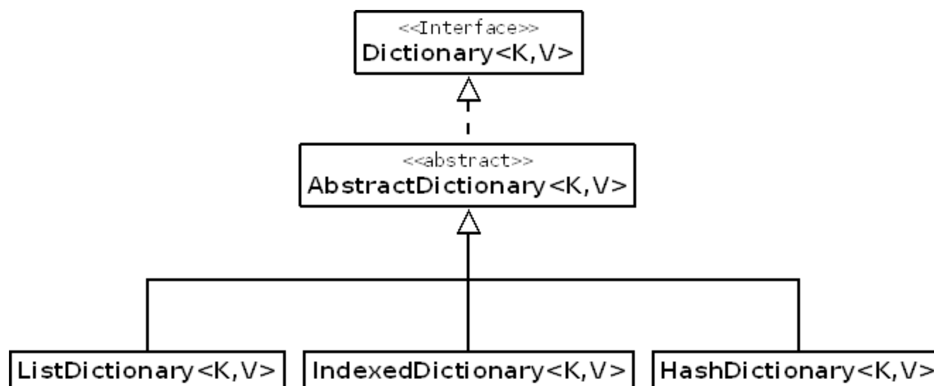
```
public class Association<K, V> {

    protected K key ;  // clé
    protected V value ; // valeur

    public Association(K k, V v) { key = k ; value = v ;}
    public K getKey () { return key ;}
    public V getValue () {return value ;}
    public String toString () { return "<"+key+","+value+">" ;} }
```

Malgré certaines optimisations, cette dernière implantation n'est que faiblement performante et ne baisse pas sensiblement le temps d'exécution des méthodes d'accès. Les techniques classiques d'implantation de tables les plus efficaces appliquent le principe de diviser pour régner, qui consiste à découper la table en sous-tables, ou plus précisément le domaine de la table en sous-domaines.

Ainsi, la recherche d'une clé se décompose en deux étapes successives :

- **Etape 1** : recherche de la sous-table susceptible de contenir la clé,
- **Etape 2** : recherche de la clé dans la sous-table.

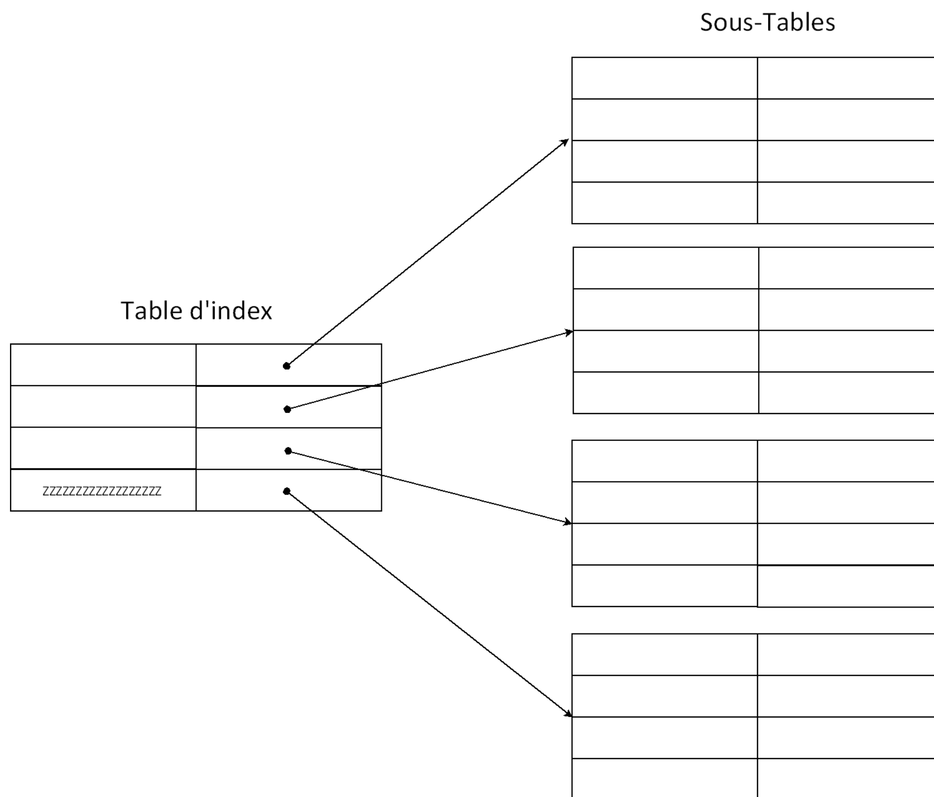
Il existe deux grandes catégories de découpage du domaine. Le découpage peut être fait en fonction de la relation d'ordre sur les clés (**représentation à base d'index**), ou être fait selon la valeur (hashcode) retournée par une fonction de hachage appliquée aux clés (**représentation par hashcode**).



3.3.1 Représentation indexée

Le domaine est découpé en sous-domaines de taille identique (sauf éventuellement pour le dernier sous-domaine), en respectant l'ordre des clés. Il faut donc disposer de l'ensemble des clés pour réaliser ce découpage, ce qui signifie que la table ne peut pas être créée vide, puis remplie par adjonctions successives ; elle doit être créée à partir d'une autre collection (une liste de couples <clé, valeur>, par exemple). C'est le principal inconvénient de cette implantation. Le nombre de sous-domaines est ici fixé a priori, mais ce nombre doit être choisi au mieux pour optimiser les performances. On gère une table d'index qui permet d'accéder spécifiquement à chaque sous-table.

☺ **Exercice 3.C :** soient les couples suivants : <Tilleul, Lime tree>, <Chene, Oak>, <Epicea, Spruce>, <Sapin, Fir>, <Erable, Maple>, <Feuille, Leaf>, <Conifere, Conifer>, <Feuillu, Broad-leave Tree>, <Marronier, Chestnut>, <Mimosa, Mimosa>, <Peuplier, Poplar>, <Arbre, Tree>, <Pin, Pine tree>, <Sequoia, Sequoia>, <Tige, Stem>. Compléter la table d'index et les sous-tables à l'aide du schéma ci-dessous. L'accès se fait à partir de la dernière clé de chaque sous-table. Dans la dernière entrée de l'index, on stocke la plus grande des clés, de façon à pouvoir gérer, comme les autres sous-tables, les adjonctions dans la dernière sous-table.



Dans cet exemple, chaque sous-table est gérée séparément (il y a d'autres solutions) : chaque adjonction se fait dans la sous-table qui lui correspond, sans jamais modifier la table d'index.

☺ **Exercice 3.D :** évaluer les performances au pire cas de la recherche d'une clé dans une représentation à base d'index, en prenant en compte le nombre total d'éléments (noté n) et la taille de la table d'index (noté m).

3.3.2 Représentation par hashcode

Le découpage du domaine de la table se fait selon une fonction de hachage qui retourne un hashcode. Cette fonction transforme la clé en une adresse dans une zone de mémoire contigue, ou plus simplement en un entier qui sert d'indice dans un tableau. Comme il existe la plupart du temps une infinité de clés possibles, plusieurs éléments peuvent avoir le même hash-code (problème de **collisions**). Ils sont alors rangés dans une même sous-table.

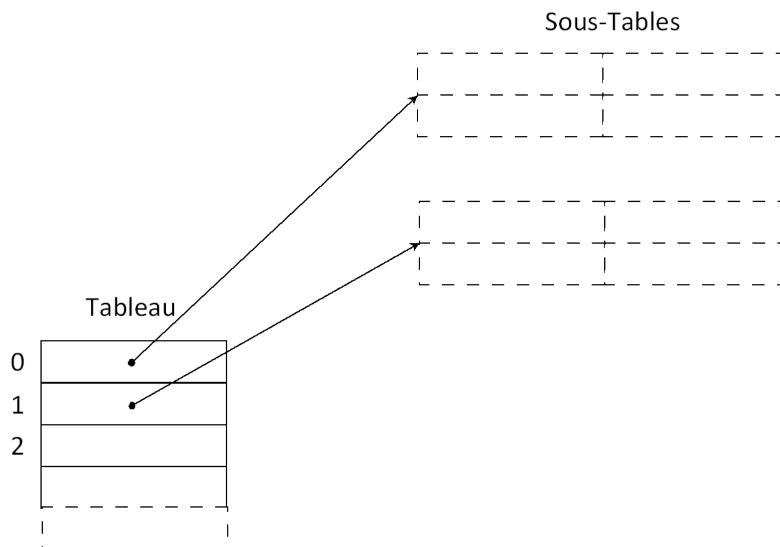
Les performances de la table sont étroitement liées à la fonction de hachage, qui doit fournir une répartition autant que possible uniforme sur l'ensemble des clés, de sorte que la répartition des éléments entre les différentes sous-tables soit équilibrée.

Formulé mathématiquement : soit une fonction de hachage h : K -> L (avec K, l'ensemble des clés et L, l'ensemble des hashcodes), alors quels que soient $x \in K$ et $i \in L$, la probabilité $P(h(x) = i)$ doit aussi proche que possible de 1/m.

Par exemple, si la clé est un mot du dictionnaire français, prendre comme fonction de hachage, le code unicode de la première lettre n'est pas satisfaisant puisque la sous-table des mots commençant par la lettre 'Z' va être petite au regard de celle des mots commençant par la lettre 'A', ceci provoquant un déséquilibrage. Bien souvent, lorsque la clé est un mot, on préfère une fonction qui intègre le code de toutes les caractères.

☺ **Exercice 3.E :** en reprenant les couples de l'exercice 3.C, compléter le schéma ci-dessous correspondant à une représentation par hashcode. Nous considérerons la fonction de hachage h(k) qui calcule le hashcode de la clé k de la manière suivante :

1) Attribuer aux lettres A, B, C … Z (indépendamment de la casse) les valeurs 1,2,3 … 26,
2) Sommer les valeurs des lettres de la clé k,
3) Ajouter au nombre obtenu le nombre de lettres de k.
4) Calculer ce dernier nombre modulo 10 pour obtenir le hashcode.



Les sous-tables peuvent bien évidemment être implantées de manière différentes : implantation contiguë, implantation à l'aide de listes, utilisation d'une seconde fonction de hachage donnant un accès calculé à un élément à partir du précédent.

☺ **Exercice 3.F :** réaliser l'implantation de la classe `HashDictionary<K,V>` (héritant de `AbstractDictionary<K ,V>`) qui implante la représentation par hashcode, avec les sous-tables implantées à l'aide d'une `ArrayList`. Vous pourrez exploiter la fonction `hashCode` héritée de la classe `Object`.