Algorithmique Avancée

Nicolas Labroche

Cours 2 : dictionnaires, tuples et couples

Université de Tours

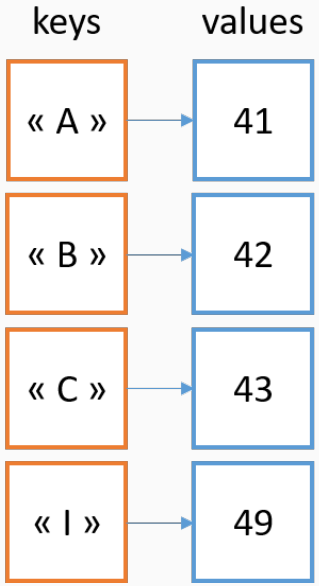
Objectifs du cours

- dictionnaires et listes d'association
- tuples
- couples

Dictionnaires

Dictionnaire

- un dictionnaire est une structure composée de clés qui permettent chacune d'identifier de manière unique un élément ou valeur
  - par analogie avec un *dictionnaire* (le livre) qui associe une chaîne de caractères à une définition
  - exemple **table ASCII** de traduction de caractères vers hexadécimal



Notion de tableau associatif

- synonyme de dictionnaire
- d'après *Wikipedia*, généralisation du tableau
  - *"le tableau traditionnel associe des entiers consécutifs à des valeurs"*
  - *"le tableau associatif associe des clefs d'un type arbitraire à des valeurs d'un autre type"*
- quelques exemples de tableau associatif / dictionnaire
  - dictionnaire classique : clé = mot, valeur = définition
  - annuaire téléphonique : clé = nom + prénom + ville, valeur = numéro de téléphone
  - table de BD : clé = index, valeur = tuple

Clé et dictionnaire

Toutes les opérations sont réalisées à l'aide d'une clé pour accéder ou modifier une valeur

- le stockage d'un élément dans le dictionnaire passe par l'attribution d'une nouvelle clé
- la recherche d'un élément se fait également par le biais d'une clé
- les clés peuvent être de n'importe quel type Java dès lors qu'une méthode `equals()` existe pour comparer les clés
- un dictionnaire ne peut contenir qu'une seule fois une clé

Barrière d'abstraction du dictionnaire

Opérations classiques sur un dictionnaire

- recherche d'une valeur par sa clé
- nouvelle association clé / valeur
- test présence clé ou valeur
- suppression d'une valeur par sa clé
- aspects programmation (taille, `toString()`)

```
public interface Dict<K,V> {
    public V get(K key);
    public void replace(K key,↩
        V value);
    public void put(K key, V ↩
        value);
    public boolean containsKey↩
        (K key);
    public boolean ↩
        containsValue(V value↩
        );
    public V remove(K key);
    public LList<K> keySet();
    public int size();
    public String toString();
}
```

Représentation simple du dictionnaire

- utilisation d'une **liste d'association**
- idée naïve : deux listes chaînées gérées en parallèle
  - une liste gère les clés
  - une liste gère les valeurs
- ou bien une liste de couples (clé, valeur) (voir après)
- avantages
  - recherche en $O(n)$ d'un élément dans une liste de taille n
  - pas d'ordre imposé sur les clés et donc pas de fonction pour obtenir la valeur à partir de la clé ($\neq$ table de hachage)
  - insertion possible en temps constant (en 1ère position uniquement)

Implémentation d'un dictionnaire `SDict`

Comment représenter **simplement** un dictionnaire en Java ?

- sous la forme de deux listes `LList` construites en parallèle
- un dictionnaire est défini en fonction de deux types génériques `<K,V>`
- une liste contient les clés de type générique `K`
- une liste contient les valeurs de type générique `V`

Implémentation d'un dictionnaire `SDict`

```
public class SDict<K,V> implements Dict<K,V> {
    LList<K> keys;
    LList<V> values;
    int size;

    public SDict(){
        keys = new SList<>();
        values = new SList<>();
        size = 0;
    }
```

Ajout d'une association clé valeur

- utilisation de la barrière d'abstraction fournie par l'interface `LList`

```
@Override
public void put(K key, V value) {
    keys.add(key);
    values.add(value);
    size++;
}
```

- problème : pas la solution la plus optimale
- ici `add` réalise un ajout en fin de liste, l'insertion n'a donc pas un coût constant
- mais implémentation de la barrière d'abstraction très simple à coder

Comment ajouter une clé et sa valeur ?

La solution précédente **n'est ni complète ni correcte**

- aucune vérification de l'unicité de la clé qui est ajoutée !
- pour s'assurer de l'unicité de la clé et insérer le nouvel élément, il faut
  - parcourir l'ensemble des clés de la liste `keys`
  - si la clé recherchée est trouvée, il faut modifier la valeur qui lui est associée
  - sinon on ajoute le nouveau couple "clé, valeur" en fin de liste et on incrémente la taille du dictionnaire

Affichage d'un dictionnaire

- parcours en parallèle des deux listes `keys` et `values`

```
public String toString() {
    String res = "";
    for (int i = 0; i < size; i++){
        res += " { " + keys.get(i).toString() + " , " + ↩
            values.get(i).toString() + "}";
        if (i < size-1) { res += ", "; }
    }
    return "[" + res + "]"; }
```

- complexité quadratique en nombre d'opération !
  - 1 opération pour le 1er élément, 2 pour le second, etc.
  - au total la complexité
    $str(n) = 1 + \ldots + n = \frac{n \times (n+1)}{2} \approx O(n^2)$
- on observe l'importance du choix des modèles pour implémenter une barrière d'abstraction !

Récupération d'une valeur

- recherche à partir d'un identifiant de clé
- parcours de tous les éléments de la liste de clés
- récupération de l'index correspondant s'il existe
- recherche dans la liste de valeurs
- complexité dans le pire des cas
  $get(n) = n + n = 2 \times n \approx O(n)$

```
public V get(K key) {
    int index = keys.indexOf(key);
    if (index > -1){
        return values.get(index);
    } else return null;
}
```

Calcul de l'index d'une clé

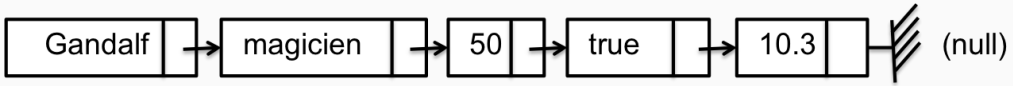
On utilise la méthode `indexOf` de `LList<T>` définie comme suit :

```
@Override
public int indexOf(T elem) {
    int index = 0;
    Node<T> p = head;
    while (p != null){
        if (p.value.equals(elem)) return index;
        else {
            p = p.next;
            index ++;
        }
    }
    return -1;
}
```

Tuples, couples et listes d'associations

Tuple

- un tuple ou n-uplet est une collection ordonnée de **n objets** qui sont des éléments du n-uplet
- il permet la représentation de structures de données avec plusieurs champs
- par exemple :



- un tuple est généralement immutable, il ne peut être modifié après sa création
- un tuple est représenté en Java par un **tableau d'objets** de façon à pouvoir contenir tout type d'élément

Définition d'un tuple en Java

- on va définir une classe appropriée pour gérer les tuples
- on utilise le mot-clé `final` pour rendre le tableau de valeurs immutable
- on utilise des `var-args` au niveau du constructeur pour permettre un nombre indifférencié de paramètres séparés par des virgules

Définition d'un tuple en Java

```
public class Tuple {
  final Object[] values;

  public Tuple(Object... elements) { // ... => var-args
    values = elements;
  }

  public <E> E get(int index){
    E tmp = (E) values[index];
    return tmp;
  }
}
```

Exemple de création d'un tuple

- exemple avec la création d'un personnage `perso` en définissant les éléments dans le constructeur :

```
// Dans la methode main
Tuple test = new Tuple("Gandalf", "Magicien", 50, true, 10.3);
System.out.println(test);

  public String toString(){
    StringBuilder res = new StringBuilder();
    for (int i = 0; i < values.length; i++){
      res.append(values[i]);
      if (i < values.length-1) { res.append(", "); }
    }
    return "{" + res + "}";
  }
```

Couples

Un couple est une structure contenant deux éléments :

- en maths : $(1, 2)$
- en Java : implémentation sous la forme d'une structure dédiée
- un couple peut être vu comme un cas particulier de tuple **avec 2 éléments**

```
public class Pair<L, R> {
    public final L left;
    public final R right;

    public Pair(L left, R right) {
        this.left = left;
        this.right = right;
    }
}
```

Accès aux éléments d'un couple

Implémentation de 2 *getters* très simples !

```
public L getLeft() {
    return left;
}
public R getRight() {
    return right;
}
```

Comparaison avec un autre couple

```
public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return ↩
            false;
        Pair<?, ?> pair = (Pair<?, ?>) o;
        return Objects.equals(getLeft(), pair.getLeft()) &&
                Objects.equals(getRight(), pair.getRight());
    }
```

Exemple de mise en œuvre (1)

- on souhaite définir une méthode **récursive** de calcul de la moyenne des éléments d'une liste
- solution classique (récursive) : faire deux parcours : un pour calculer la somme et l'autre le nombre d'éléments dans la liste
- solution alternative : utiliser un couple de type `Pair` pour stocker ces deux informations
  - la partie gauche `left` du couple est de type `Integer` : taille de la liste
  - la partie droite `right` du couple est de type `Double` : somme des éléments de la liste
- le type retenu est donc `Pair<Integer, Double>`

Exemple de mise en œuvre (2)

- deux méthodes doivent être définies
- une méthode publique `average()` qui s'assure que la liste contient des nombres sur lesquels la moyenne peut être calculée

Exemple de mise en œuvre (3)

```
    public double average(){
        if (head == null) return 0;
        else {
            Class c = head.value.getClass();
            if (Number.class.isAssignableFrom(c)){
                Pair<Integer, Double> res = sum_length(head);
                return res.getRight() / res.getLeft();
            } else {
                System.out.println("This instance of list does↩
                     not contain values that can be averaged"↩
                    );
                return 0;
            }
        }
    }
```

Exemple de mise en œuvre (4)

- une méthode privée récursive `sum_length` qui réalise le traitement demandé
- cette méthode travaille sur le type récursif `Node`

```
private Pair<Integer, Double> sum_length(Node<T> node){
    if (node == null) return new Pair(0,0);
    else {
        Pair<Integer, Double> tmp = sum_length(node.next);
        int left = tmp.getLeft()+1;
        double right = ((Number)tmp.getRight()).↩
            doubleValue() + ((Number) node.value).↩
            doubleValue(); //
        return new Pair<Integer, Double>(left, right);
    }
}
```

Liste d'association avec des couples

- une association est un **couple** contenant une clé et une valeur
- la clé permet de retrouver la (ou les) valeur(s) associées
- exemple :

```
(Cle Valeur), ("Gandalf", "magicien")
```

- une liste d'association est une liste chaînée dont chaque élément est une association

Liste d'association avec des couples

```
public class SAssoc<K,V> implements Dict<K,V> {

    protected Node<K,V> head;
    protected int size;

    public SAssoc(){
        head = null;
        size = 0;
    }
```

Adaptation de la classe `Node`

- on reprend l'idée du `Node` utilisé dans la classe de liste chaînée `SList`

```
public static class Node<K,V> {
    Pair<K,V> value;
    Node<K,V> next;

    public Node(K key , V value){
        this.value = new Pair<K,V>(key, value);
        this.next = null;
    }

    public Node(K key , V value, Node<K,V> _next){
        this.value = new Pair<K,V>(key, value);
        this.next = _next;
    }
}
```

Affichage d'une liste d'association

- il s'agit d'un parcours de liste classique dont les éléments sont des couples `Pair<K,V>`
- la complexité redevient donc linéaire en $O(n)$

```
public String toString() {
    StringBuilder sb = new StringBuilder("[");
    Node<K,V> p = head;
    while (p != null){
        sb.append(p.value); // p.value de type Pair<K,V> !
        if (p.next != null) sb.append(",");
        p = p.next;
    }
    sb.append("]");
    return sb.toString();
}
```

Récupération d'une valeur à partir d'une clé

- par rapport au dictionnaire `SDict`, opération (un peu) plus complexe
- il faut récupérer pour chaque couple sa clé
- si elle correspond à ce qui est cherché alors retourner le second élément du couple

```
public V get(K key) {
    Node<K,V> p = head;
    while (p != null){
        if (Objects.equals(p.value.getLeft(), key))
            return p.value.getRight();
        else p = p.next;
    }
    return null;
}
```

Récupération de la liste de clés

- par rapport au dictionnaire `SDict`, opération plus complexe
- il faut récupérer pour chaque couple sa clé et l'insérer dans une nouvelle liste

```
public LList<K> keySet() {
    SList<K> tmp = new SList<>();
    Node<K,V> p = head;
    while (p != null){
        tmp.add(p.value.getLeft());
        p = p.next;
    }
    return tmp;
}
```

Exemple d'exécution

- que fait le code suivant ?

```
SAssoc<String, String> persos = new SAssoc();
persos.put("Gandalf", "Magicien");
persos.put("Sauron", "Sorcier");
persos.put("Aragorn", "Rodeur"); // ajout de 3 couples

System.out.println(persos); // affichage

persos.put("Gandalf", "Le Blanc");
// suppression du 1er Gandalf
// remplacement par le nouveau
// pas de doublon

System.out.println(persos.keySet()); // affichage des ↩
    cles
System.out.println(persos.get("Sauron")); // "Sorcier"
System.out.println(persos.get("Legolas")); // null
```

Pour aller plus loin : les tables de hachage

- les tables de hachage sont des structures de type tableau associatif sans ordre
- elles sont efficaces pour stocker des associations clés / valeurs indexées dans des alvéoles (ou buckets / slots en anglais)
- principe :
  - chaque clé est encodée par une fonction de hachage qui sert d'empreinte numérique pour indexer le couple dans la structure
  - si la fonction de hachage n'est pas injective, alors il peut survenir des collisions : plusieurs couples (clé, valeur) correspondent à la même empreinte dans la structure

Illustration des tables de hachage