2. *Types et structures*

NOTRE PAIN ET NOTRE BEURRE dans ce cours sont les structures de données et les types abstraits. Le type abstrait définit une interface pour l'employer dans un algorithme, et la structure implémente le type — il y a une multitude de structures possibles pour le même type.

2.1 *Type abstrait*

Définition 2.1. *Un **type** est un ensemble (possiblement infini) de valeurs et d'opérations sur celles-ci. L'ensemble d'opérations s'appelle l'*interface.

Définition 2.2. *Un **type abstrait** (TA) est un type accessible* uniquement *à travers une interface.*

Notions.

- ⋆ **client** : le programme qui utilise le TA
- ⋆ **implantation**/implémentation : le programmme qui définit la représentation des valeurs, et l'exécution des opérations dans le le TA
- ⋆ **interface** : contrat entre le client et l'implantation

Collections

Les types abstraits fondamentaux du cours servent premièrement à stocker des collections d'éléments. Les opérations de l'interface spécifient la manière d'ajouter, de supprimer, et d'examiner des éléments dans l'ensemble ou collection representée. Par défaut, il y a toujours une opération pour tester si la collection est vide. Souvent, mais non pas toujours, il y a également une manière de parcourir (énumérer) tous les éléments de la collection.

Le type le plus simple qu'on considère, le sac, ne permet que l'ajout et parcours d'éléments. Une **file générique** est un type abstrait pour une collection d'éléments avec deux opérations principales : une opération pour ajouter un élément, et une autre pour retirer un élément. La règle du choix de l'élément à retirer fait partie de la définition de l'interface : queue (first-in-first-out), pile (last-in-first-out), file de priorité (élément de valeur maximale/minimale). Sacs et files simples peuvent être implémentés facilement par des tableaux. Par contre, les types abstraits qui permettent des opérations sophistiquées, comme la recherche ou la sélection, demandent des structures soigneusement conçues.

TAs		***opérations principales***						***structures typiques***
		vide?	parcourir	ajouter	retirer	rechercher	min	
sac (*bag*)		+	+	add	-	-	-	tableau ou liste chaînée
files	pile (*stack*)		-	push	pop			
	queue			enqueue	dequeue			
	file à priorités (*priority queue*)		-	add	deleteMin	-	+	tas (*heap*)
ensemble (*set*)			+/-	add ou -	delete ou -	contains	-	tableau de hachage
table de symboles (*symbol table / map*)			+	add	delete	get		
dictionnaire ordonné (*sorted dictionary*)							+	arbre binaire de recherche

FIG. 1: Types abstraits fondamentaux et leur usage.

Sac. Un **sac** (*bag*) contient des éléments (qui ne sont pas nécessairement distincts). Il y a une opération principale pour ajouter un élément, et on a le droit de parcourir les éléments du sac.

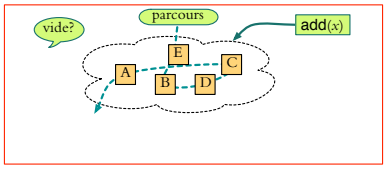

FIG. 2: Sac (*bag*)

Pile. Dans une **pile** (*stack*), l'élément le plus récemment ajouté est celui qui est retiré avant les autres (dernier entré, premier sorti). Les opérations de base s'appellent push («empiler») et pop («dépiler»).

Queue. Dans une **queue** ou file FIFO, l'élément le plus ancient sera retiré avant les autres (premier entré, premier sorti). Les opérations de base s'appellent enqueue («enfiler») et dequeue («défiler»).

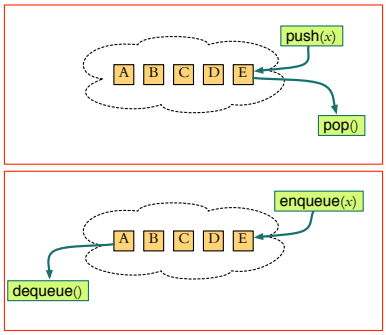

FIG. 3: Pile (en haut) et queue FIFO (en bas).

Recherche. Une **table de symboles**, ou **dictionnaire** est un ensemble d'objets avec clés uniques. Ce type supporte la **recherche** par clé, qui finit par être *fructueuse* (il y a un élément avec la clé), ou non. Un ensemble correspond au cas quand la clé est l'élément soi-même.

Sélection. S'il y a un ordre parmi les éléments de la collection, une opération de **sélection** est de choisir le minimum ou le maximum. Dans un cas plus générique on peut s'intéresser à retirer la médiane ou le k-ème élément, ou bien à parcourir les éléments selon l'ordre. Une **file de priorité** (*priority queue*) est une file avec sélection : on peut retirer le minimum.

2.2 *Types et variables en Java*

Les types définissent une couche d'abstraction pour la manipulation de données dans nombreuses situations :

- ⋆ arithmétique et logique (entiers, nombres flottants, booléens)
- ⋆ entrée-sortie (Java : dans `java.io`, `java.net`,...)
- ⋆ encapsulation de données appartenant à un seul objet (personne, forme géométrique, ...)
- ⋆ collections d'éléments → structure de données efficace

FIG. 4: John von Neumann (1903–1957), inventeur de l'architecture des ordinateurs modernes

Variables. Rappel : une **variable** est l'abstraction d'un emplacement en mémoire (définition de von Neumann), et comprend

- ⋆ le nom
- ⋆ l'adresse en mémoire (*lvalue*)
- ⋆ la valeur (*rvalue*)
- ⋆ le type
- ⋆ la portée

En Java[1], les types peuvent être primitifs ou agrégés :

- ⋆ types primitifs (`int`, `double`, `boolean`, ...)
- ⋆ types agrégés (tableaux et ceux définis par les classes)

[1] Exemples : (a) type primitif int (entier sur 32, signé) :
- ⋆ valeurs possibles $-2^{31}..2^{31}-1$,
- ⋆ opérations +, ∗, etc. (définies par la spécification du langage)

(b) type agrégé String (chaîne de caractères) :
- ⋆ valeurs possibles : null et toute autre référence en mémoire
- ⋆ opérations `length`(), +, etc.

La valeur d'une variable de type agrégé est une référence. Une **référence** (ou pointeur) est une adresse d'emplacement mémoire contentant de l'information (ou elle est nulle).

Interface et `interface`.

- ⋆ L'interface d'une classe Java comprend (1) les variables non-privées, et (2) les signatures de méthodes et de constructeurs non-privés.
- ⋆ En Java, un type peut être déclaré sans implémentation en un `interface` ou en une `abstract class`. Mais on place l'interface et l'implémentation souvent dans le même fichier (classe avec méthodes publiques without `extends`).
- ⋆ Les clients (autres classes) ont des droits différents (sous-classe, package)
- ⋆ L'`interface` en Java n'est pas exactement l'interface de notre définition : en Java, on ne peut pas déclarer un constructeur sans implémentation tandis que dans notre discussion de types, on aime spécifier l'initialisation.

Java Collections

Java Collections Framework[2] : interfaces et implantations, dans le package `java.util`, incluant les types suivants (Fig. 5) :

[2] Tutoriel : (*Collections*)

- ⋆ liste (`java.util.List`)
- ⋆ ensemble (`java.util.Set`)
- ⋆ dictionnaire et ensemble (`java.util.Map`)
- ⋆ dictionnaire/ensemble ordonnés (`java.util.SortedMap` ou `java.util.SortedSet`)

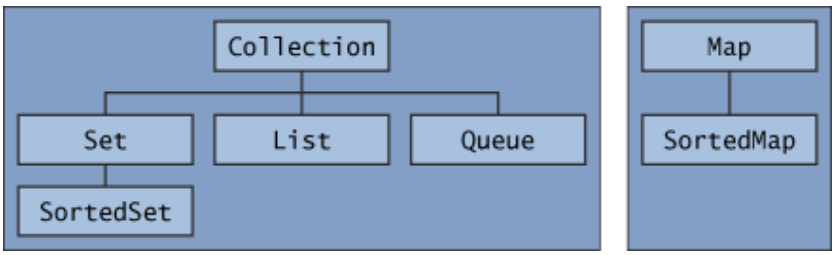

FIG. 5: Interfaces essentielles en Java.

1. Toute collection est `Iterable`. Pour parcourir ses éléments, on peut se servir alors d'une boucle **for**-each ou d'un itérateur[3] (v. Fig. 6).

2. Il existe plusieurs implémentations de la même interface (Fig. 7) : p.e., `HashMap` et `TreeMap` implantent l'interface `Map` : l'un par un tableau de hachage et l'autre par un arbre binaire de recherche. C'est une bonne idée de déclarer une variable du type aussi général que possible (type de l'interface) tandis qu'on choisit une implémentation lors d'affectation :

[3] Tutoriel : (*Traversing Collections*)

```
for (Object o : collection)
    System.out.println(o);
```

```
public interface Iterator<E>
{
    boolean hasNext();
    E next();
    void remove(); //optionnelle
}
```

FIG. 6: Boucle for-each (en haut), et les opérations de `Iterator`.

```
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.TreeSet;
...
Map<String, int[]> string_map = new HashMap<>();
Set<Integer> number_set = new TreeSet<>();
```

Ainsi, on peut tester des implémentations différentes à l'aise pendant le développment du code.

General-purpose Implementations

Interfaces	Implementations				
	Hash table	**Resizable array**	**Tree**	**Linked list**	**Hash table + Linked list**
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

FIG. 7: Implémentations différentes.

3. Les interfaces des collections de Java spécifient beacoup de méthodes (Fig. 8), avec des dépendences entre eux. Afin d'aider le développement de nouvelles structures, il existe aussi des classes abstraites qui implantent les nombreuses méthodes de l'interface sauf une ou deux que le/la programmeur/e doit fournir. En particulier, la classe `AbstractCollection`[4], ne laisse que deux méthodes abstraites à définir : `size()` et `iterator()`.

[4] Java API docs :`Collection` `AbstractCollection`

```
public interface Collection<E> extends Iterable<E>
    // contient des éléments de type E
{
    // opérations de base
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);         //optionnelle
    boolean remove(Object element); //optionnelle
    Iterator<E> iterator();

    // opérations de masse
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optionnelle
    boolean removeAll(Collection<?> c);        //optionnelle
    boolean retainAll(Collection<?> c);        //optionnelle
    void clear();                              //optionnelle

    // opérations de tableaux
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

FIG. 8: Opérations communes pour toute collection. «optionnelle» : on a le droit de jeter une exception `UnsupportedOperationException` au lieu de fournir la fonctionnalité.