


# Synthèse sur les structures de données



HMIN215  
Département Informatique  
Faculté des sciences  
Université de Montpellier



# Plan

- Les principales structures de données
- Le cas de Java
- Choisir une structure de données
  - Par les opérations proposées
  - Par une approche de la complexité

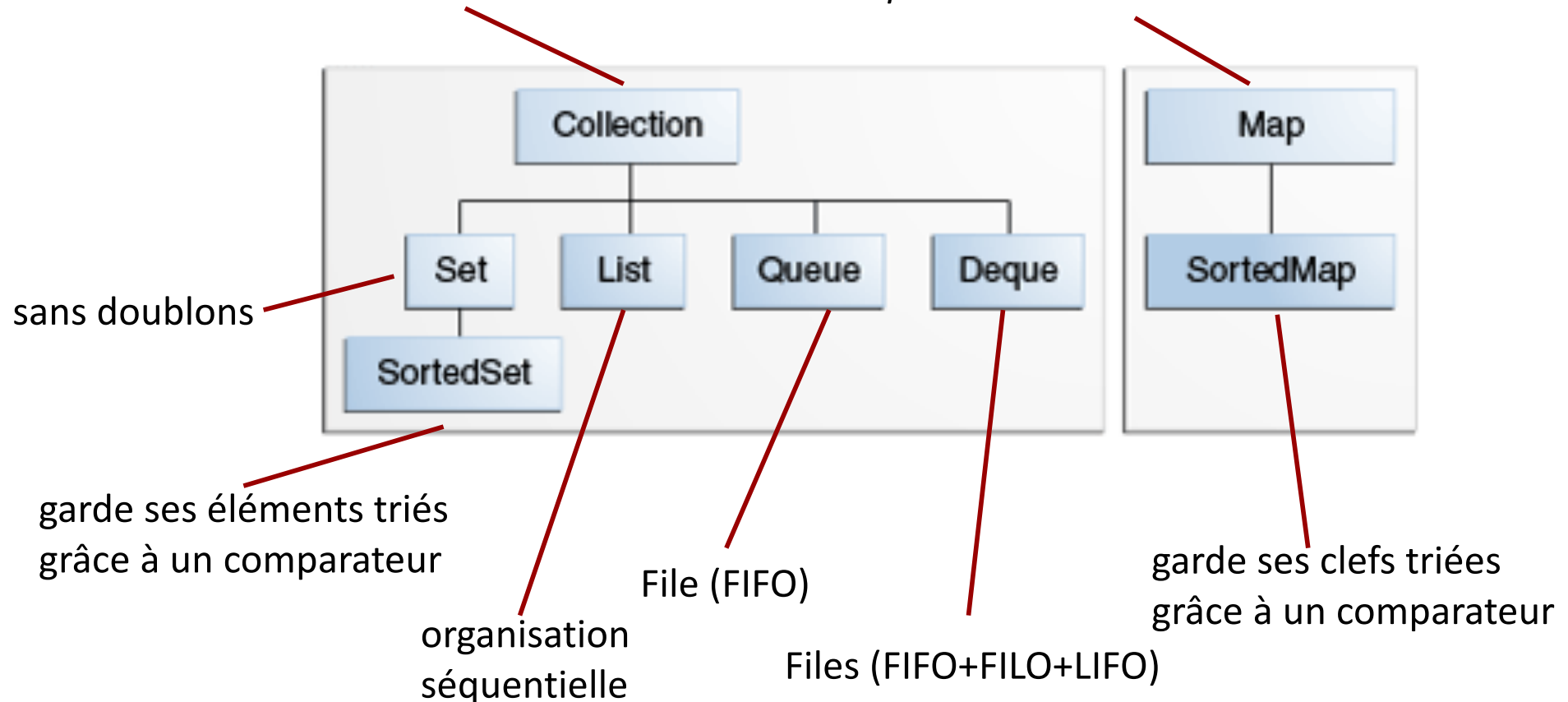
# Les principales structures de données

- **Séquences** : tableau, pile, files, liste
- **Arbres** : arbres binaires, arbres binaires de recherche, arbres rouges et noirs, arbres n-aires, B-arbres, arbres préfixes (trie)
- **Tas** : arbre binaire implémenté dans un tableau, tas binomiaux, tas de Fibonacci (ensembles d'arbres particuliers)
- **Tables de hachage**, filtres de bloom
- **Graphes** (listes d'adjacence ou table d'incidence)

# Les principales interfaces de Java

*isEmpty, size*  
*add, remove*  
*contains, iterator*

associe des clefs à des valeurs,  
pas de doublon de clefs  
*put*



# Les principales implémentations

Implémentation par hashtable

Implémentation par tableau

Implémentation par arbres équilibrés

Implémentation par liste chaînée

Set	List	Queue	Deque	Map
HashSet				HashMap
	ArrayList	ArrayDeque	ArrayDeque	
TreeSet		PriorityQueue		TreeMap
	LinkedList	LinkedList	LinkedList	
LinkedHashSet				LinkedHashMap

# Autres caractéristiques

- Des méthodes statiques (factories) permettent d'obtenir des implémentations ayant des propriétés de :
  - Synchronisation
    - en programmation multi-threads, les opérations ne peuvent être interrompues, ce qui préserve l'intégrité des données (Vector)
    - ex. la méthode **synchronizedList** retourne une liste synchronisée
  - Immutabilité
    - collections non modifiables, pour limiter l'accès en lecture seule par certains programmes clients
    - ex. la méthode **unmodifiableList** retourne une liste non modifiable

# Algorithmes polymorphes

- Implémentés par des méthodes statiques
- Trier (**sort**), mélanger (shuffle)
- Inverser (reverse), remplir (fill), copier (copy), échanger (swap)
- Rechercher (binarySearch) dans une liste triée, **Min, Max**
- Fréquence d'un élément (frequency), savoir si deux collections sont disjointes (disjoint)
- D'autres peuvent être programmés grâce aux itérateurs ou aux streams

# Complexité

- Introduction à la complexité
  - en place mémoire utilisée
  - en temps de calcul

Complexité **en place mémoire utilisée**

*En simplifiant*

- pour un booléen, caractère, nombre borné : 1
- pour un tableau ou un ensemble :

*nombre d'éléments  $\times$  taille d'un élément*



# Complexité

- Introduction à la complexité
  - en temps de calcul
  - en place utilisée

Complexité **en temps de calcul**

*pour simplifier ....*

La complexité **théorique** d'un algorithme est le nombre d'opérations élémentaires exécutées par l'algorithme en fonction de la taille de la donnée et dans le plus mauvais des cas.

# Complexité

## Opération élémentaire

Opération dont le temps d'exécution est borné par une constante (ce temps est indépendant de la donnée)

## Exemples

- Affectation de variables simples
- Accès à un élément de tableau, à un attribut
- Opérations booléennes, comparaisons
- Opérations arithmétiques ordinaires

# Complexité

Exemple d'un calcul de moyenne

```
public static double moyenne(double a, double b){  
    double somme = a+b;  
    return somme/2;  
}
```

Evaluation de la complexité : constante

➤ 4 opérations élémentaires

1 affectation, 2 opérations arithmétiques, 1 opération "retourner"

➤ 4 emplacements mémoire de la taille d'un double

# Complexité

Exemple d'une recherche dans un tableau de taille  $n$

```
public static boolean recherche(int[] tab, int v){  
    for (int i=0; i<tab.length; i++)  
        if (tab[i]==v) return true;  
    return false;  
}
```

Evaluation de la complexité : linéaire par rapport à  $n$

- opérations élémentaires : au plus  $5n+1$ 
  - $< n$  affectations de valeurs à  $i$ ,  $< n$  comparaisons de  $i$  avec  $\text{tab.length}$ ,
  - $< n$  incrémentations de  $i$ ,  $< n$  accès à  $\text{tab}$ ,  $< n$  comparaisons,
  - 1 opération "retourner"
- emplacements mémoire de la taille d'un int :  $n+2$  et 1 booléen

# Complexité

Recherche dans un tableau de taille n

```
public static boolean recherche(int[] tab, int v){  
    for (int i=0; i<tab.length; i++)  
        if (tab[i]==v) return true;  
    return false;  
}
```

Recherche de 16 dans [2, 21, 8, 9, 16, 32, 14]

16 == 2 ? [2, 21, 8, 9, 16, 32, 14]

16 == 21 ? [2, 21, 8, 9, 16, 32, 14]

16 == 8 ? [2, 21, 8, 9, 16, 32, 14]

16 == 9 ? [2, 21, 8, 9, 16, 32, 14]

16 == 16 ? [2, 21, 8, 9, 16, 32, 14] → true

# Complexité

Exemple de recherche dans un tableau **trié** de taille  $n$  (par dichotomie)

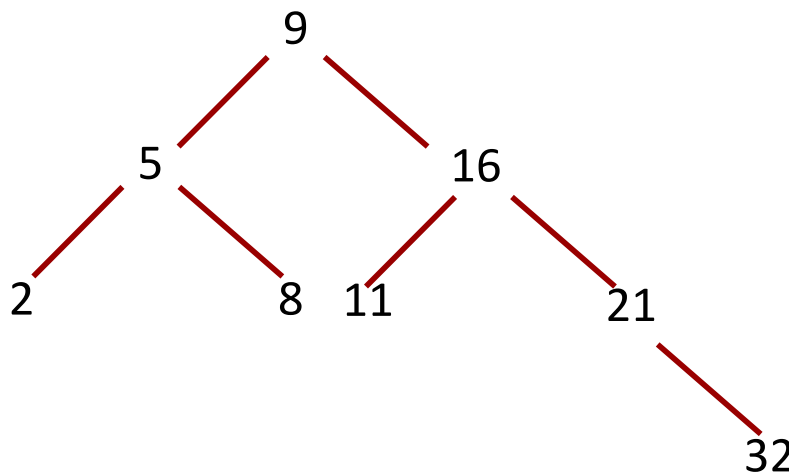
```
public static boolean rechercheDicho(int [] t, int v) {  
    boolean stop=false, res=false;  
    int indi=0, indf=t.length-1; int indm, valm;  
    while ( stop == false ) {  
        if ( indi > indf ) stop = true;  
        else {  
            indm = (indi+indf)/2;  
            valm = t[indm];  
            if ( valm == v ) {stop = true; res = true; } // on a trouvé !  
            else  
                if ( v < valm ) indf = indm-1; // chercher à gauche  
                else indi = indm+1; } } // chercher à droite  
    return res ;  
}
```

➤ Evaluation de la complexité : de l'ordre de  $\log_2(n)$

# Complexité

Exemple de recherche dans un tableau **trié** de taille  $n$  (par dichotomie)

Recherche dans [2, 5, 8, 9, 11, 16, 21, 32]



$$n = 2^h \quad h = \log_2(n)$$

*Dans un arbre binaire complet de hauteur  $h$*   
**nombre de nœuds =  $\text{floor}(2^{h+1}-1)$**

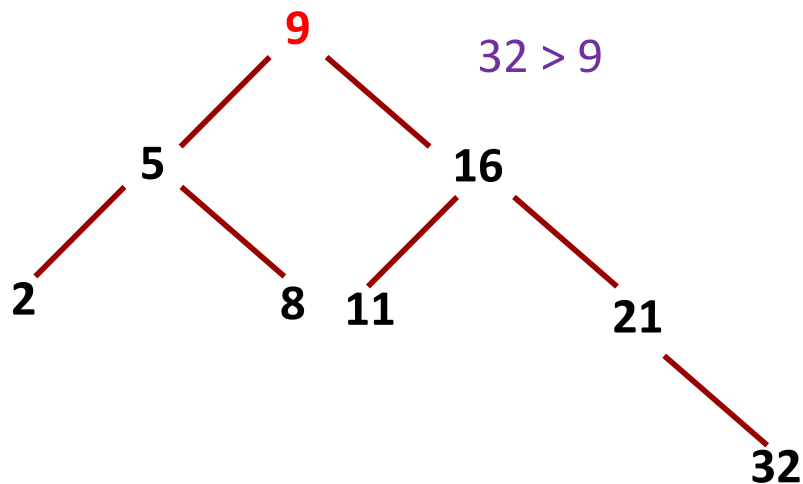
On descend dans le pire des cas sur une branche de la profondeur de l'arbre

Evaluation de la complexité : de l'ordre de  $\log_2(n)$

# Complexité

Exemple de recherche dans un tableau **trié** de taille  $n$  (par dichotomie)

Recherche de 32 dans [2, 5, 8, 9, 11, 16, 21, 32]

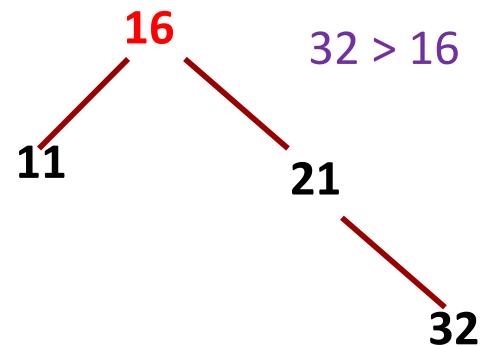




# Complexité

Exemple de recherche dans un tableau **trié** de taille  $n$  (par dichotomie)

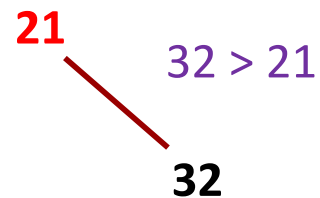
Recherche de 32 dans [11, 16, 21, 32]



# Complexité

Exemple de recherche dans un tableau **trié** de taille  $n$  (par dichotomie)

Recherche de 32 dans [21, 32]



# Complexité

Exemple de recherche dans un tableau **trié** de taille n (par dichotomie)

Recherche de 32 dans [32]

32 = 32

32

# Complexité

Exemple du tri par sélection d'un tableau de taille n

```
public static void triSelection(int []arr){
    int indiceDuMin = 0;
    for(int i = 0; i < arr.length; i++) {
        indiceDuMin = i;
        for(int j = i + 1; j < arr.length; j++)
            if(arr[j] < arr[indiceDuMin])
                indiceDuMin = j;
        int temp = arr[i]; arr[i] = arr[indiceDuMin]; arr[indiceDuMin] = temp;
    }
}
```

Evaluation de la complexité : de l'ordre de  $n^2$ , quadratique par rapport à n

Examen des comparaisons :

i=0 : n-1 comparaisons de arr[j] et de arr[indiceDuMin]

i=1 : n-2 comparaisons de arr[j] et de arr[indiceDuMin]

..... environ  $n(n+1)/2$  comparaisons

# Complexité

Exemple du tri par sélection d'un tableau de taille n

Les parties colorées sont triées

[11, 8, 2, 32] Départ – rien n'est trié

Min=2 [11, 8, 2, 32] [2, 11, 8, 32] Le Min est placé à la fin de la partie triée

Min=8 [2, 11, 8, 32] [2, 8, 11, 32] Le Min est placé à la fin de la partie triée

Min=11 [2, 8, 11, 32] Le Min est placé à la fin de la partie triée

Min=32 [2, 8, 11, 32] Le Min est placé à la fin de la partie triée

# Complexité

Exemple de la génération de tous les sous-ensembles d'un ensemble de taille n  
inspiré de : [http://rosettacode.org/wiki/Power\\_set](http://rosettacode.org/wiki/Power_set)

```
public static <T> List<List<T>> powerset(Collection<T> list) {  
    List<List<T>> ps = new ArrayList<List<T>>();  
    ps.add(new ArrayList<T>()); // add the empty set  
    // for every item in the original list  
    for (T item : list) {  
        List<List<T>> newPs = new ArrayList<List<T>>();  
        for (List<T> subset : ps) {  
            // add the subset (without the current item)  
            newPs.add(subset);  
            // add the subset with the current item  
            List<T> newSubset = new ArrayList<T>(subset);  
            newSubset.add(item);  
            newPs.add(newSubset);  
        }  
        // powerset is now powerset of list.subList(0, list.indexOf(item)+1)  
        ps = newPs;  
    }  
    return ps;  
}
```

# Complexité

Exemple de la génération de tous les sous-ensembles d'un ensemble de taille  $n$

inspiré de : [http://rosettacode.org/wiki/Power\\_set](http://rosettacode.org/wiki/Power_set)

Valeurs de  $ps$  :

`[]`

`[], [a]`

`[], [b], [a], [a, b]`

`[], [c], [b], [b, c], [a], [a, c], [a, b], [a, b, c]`

Evaluation de la complexité : de l'ordre de  $2^n$

# Complexité

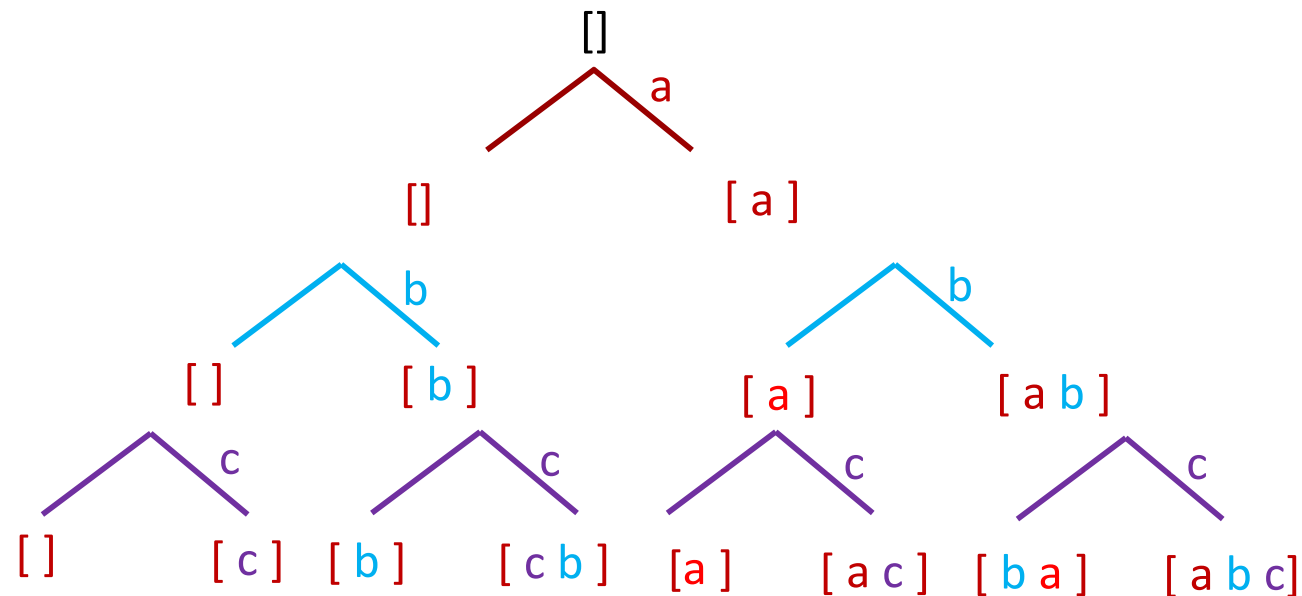
Génération de tous les sous-ensembles d'un ensemble de taille  $n$

Valeurs de  $p_s$  :

$[\ ]$

$[\ ], [a]$

$[\ ], [b], [a], [a, b]$

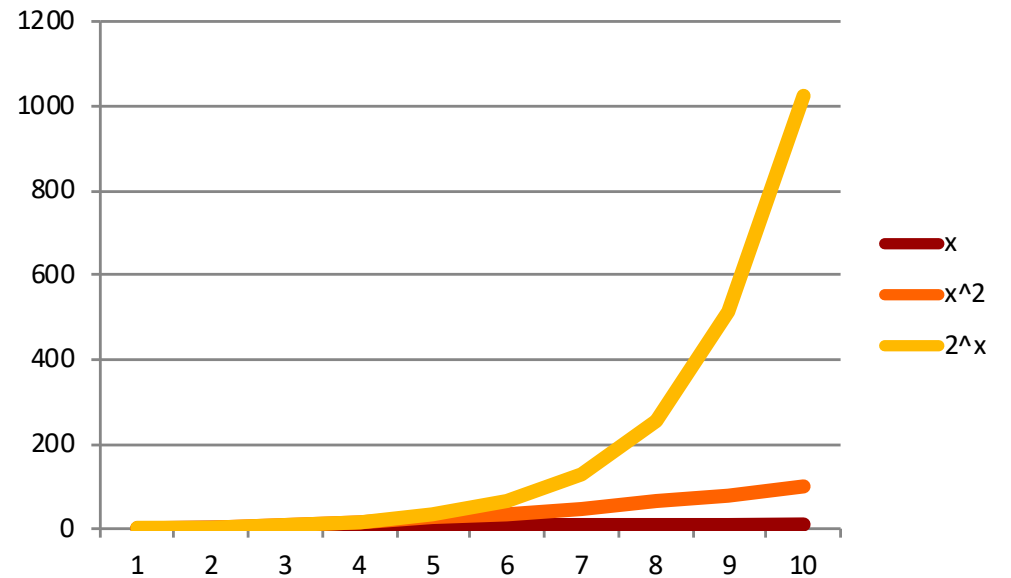
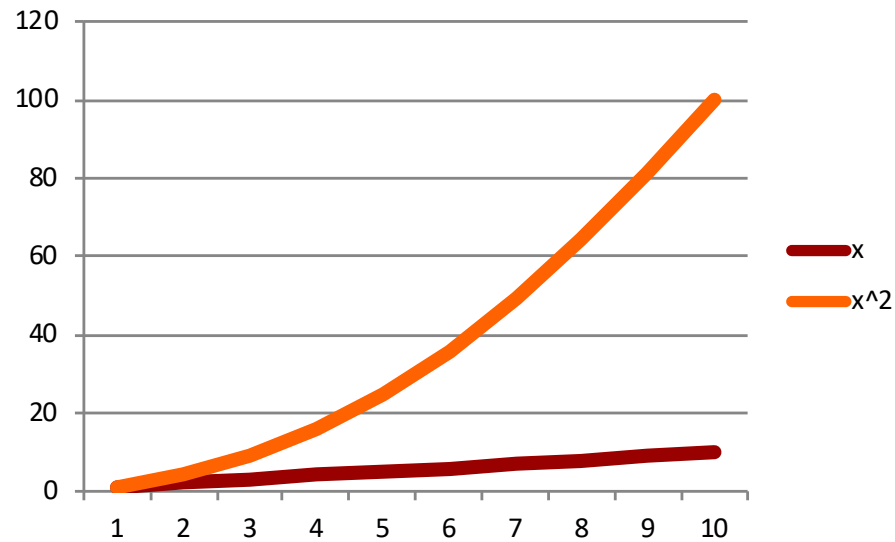
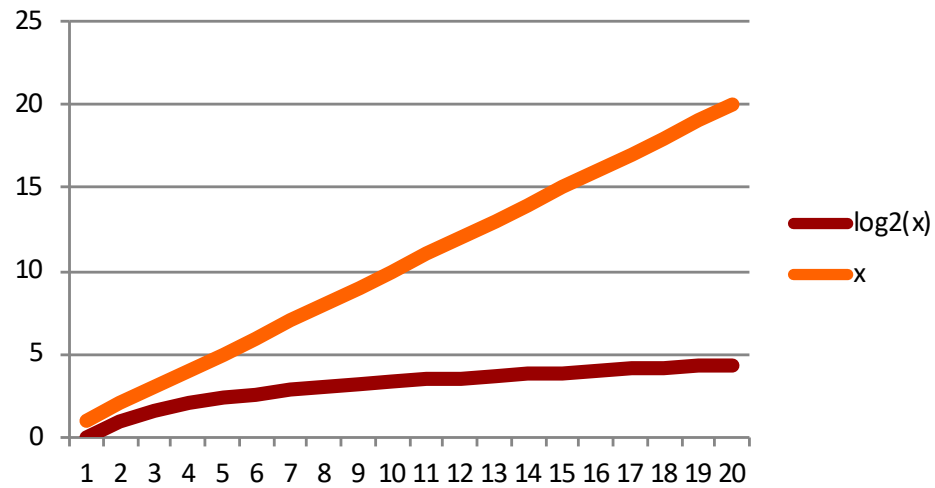


$[\ ], [c], [b], [b, c], [a], [a, c], [a, b], [a, b, c]$

Evaluation de la complexité : de l'ordre de  $2^n$  (ici  $2^4-1$ ) en  $O(2^n)$

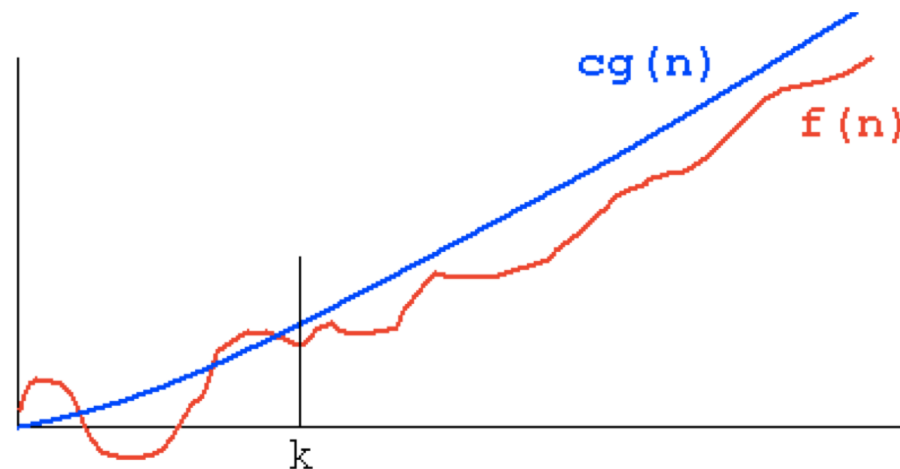


# Complexité



# Complexité

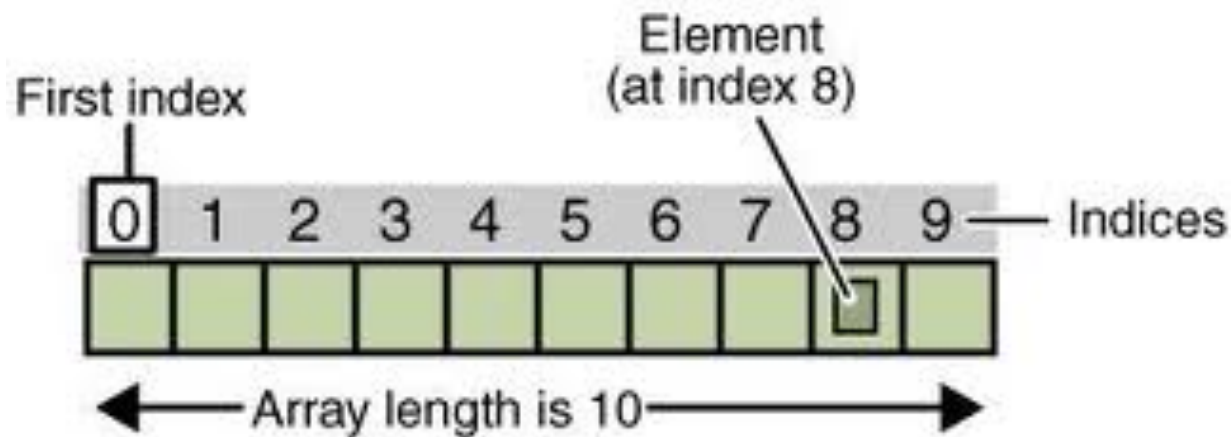
- Des courbes, on tire que ce qui est important pour l'évaluation et la comparaison des algorithmes est la tendance quand la taille de la donnée augmente
- Pour  $g(n)$ ,  $O(g(n))$  est l'ensemble des fonctions  $f(n)$  telles qu'il existe un réel  $c > 0$  et un entier  $k > 0$  tel que pour tout  $n > k$ ,  $f(n) < c \times g(n)$ .



# Complexité

- Si on s'intéresse au pire des cas
  - Recherche dichotomique  $O(\log_2(n))$
  - Recherche séquentielle  $O(n)$
  - Tri par sélection  $O(n^2)$
  - Génération des sous-ensembles  $O(2^n)$
- On peut faire aussi des analyses
  - en moyenne (ex. coût moyen d'une recherche)
  - amorties (ex. coût de l'ajout de  $n$  éléments)

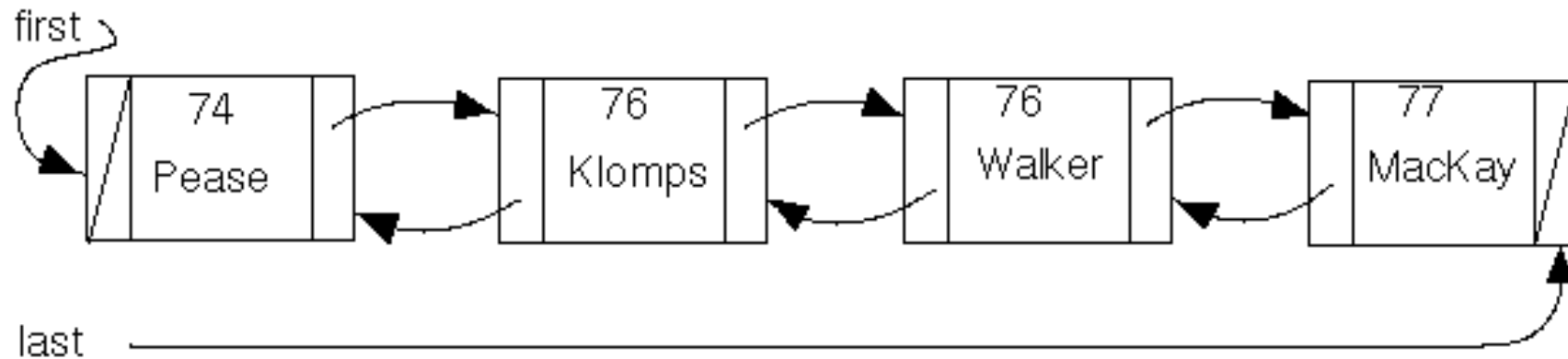
# ArrayList



***size, isEmpty, get, set*** : en temps constant (accès à une valeur ou à une case)  **$O(1)$**

***add/remove*** : dans le pire des cas il faut décaler des éléments, dans le pire des cas, on devra décaler tous les éléments ( **$O(n)$** )

# LinkedList



***size, isEmpty, add, remove, set, get*** : temps constant

***add(i, elt), get(i), remove(i)*** : parcours de la liste jusqu'à l'élément *i* (dans le pire des cas toute la longueur de la liste) en  **$O(n)$**

# HashSet (ou HashMap)

élément	hash(element)
-----	-----
"beer"	5
"afterlife"	9
"wisdom"	4
"politics"	10
"schools"	1
"fear"	3

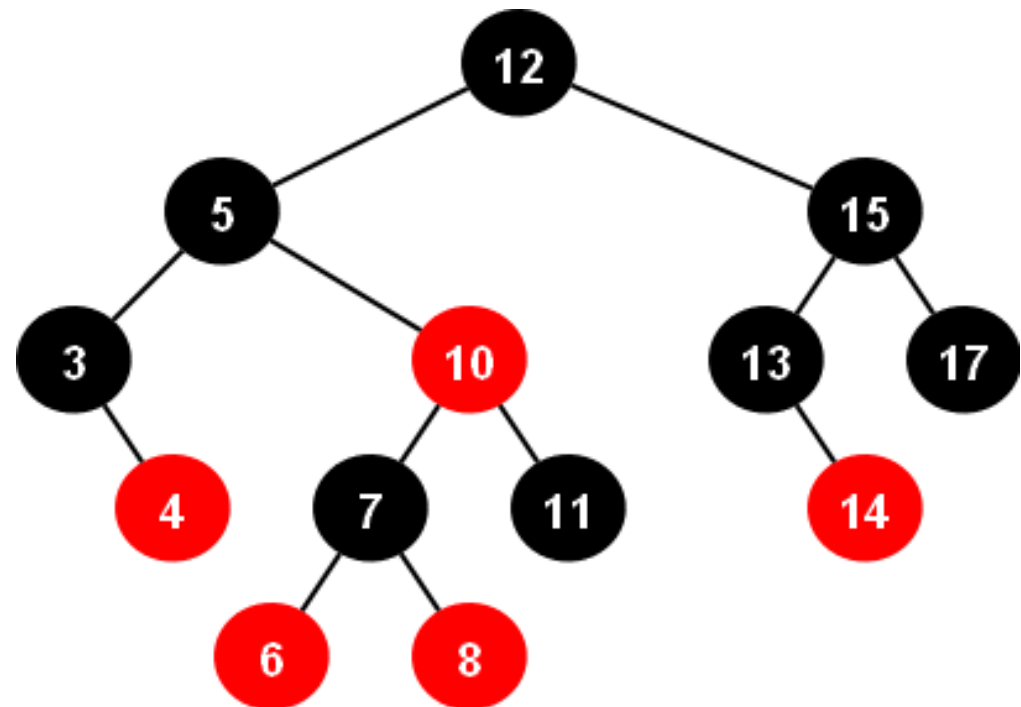
***add, remove, contains, size* :  $O(1)$**   
considérés comme en temps constant,  
accès par la valeur résultat de hash

0	
1	"schools"
2	
3	"fear"
4	"wisdom"
5	"beer"
6	
7	
8	
9	"afterlife"
10	"politics"

# TreeSet (ou TreeMap)

**Arbre binaire de recherche  
(ordonné) équilibré**

si  $n$  est le nombre de  
nœuds, la profondeur ne  
dépassé jamais  $2 \log_2(n)$



***add, remove, contains, size :  $O(\log_2(n))$***

# LinkedHashSet

"music"

hash code: 104263205

array index: 2

"beer"

hash code: 3019824

array index: 5

"afterlife"

hash code: 1019963096

array index: 9

"wisdom"

hash code: -787603007

array index: 4

"politics"

hash code: 547400545

array index: 10

"theater"

hash code: -1350043631

array index: 2

"schools"

hash code: 1917457279

array index: 1

"painting"

hash code: 925981380

array index: 5

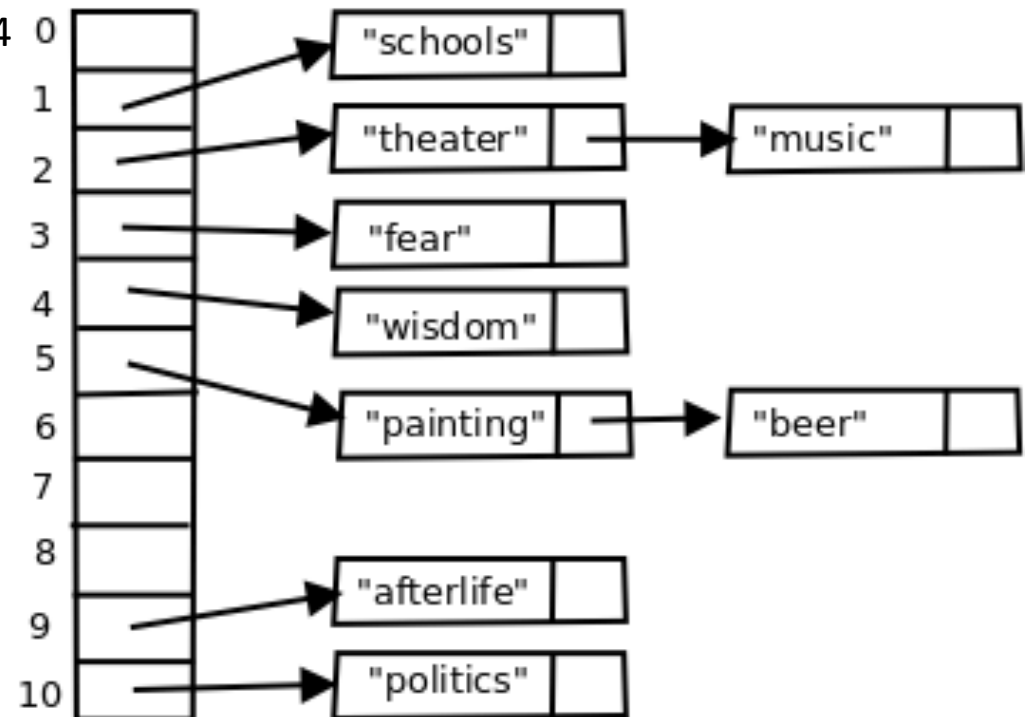
"fear"

hash code: 3138864

array index: 3

***add, remove, contains, size : O(1)***

considéré en temps constant,  
accès par la valeur résultat de hash  
et parcours courte liste pour les  
collisions





# Complexité des opérations en Java

	$O(1)$	$O(\log_2(n))$	$O(n)$
ArrayList (tableau)	size, isEmpty, get, set		add/remove ( $O(1)$ en complexité amortie)
LinkedList (cellules chaînées)	size, isEmpty, add, remove, set, get		méthodes faisant référence à un indice get(i), set(i)
HashSet/HashMap (table de hachage)	add, remove, contains size		
TreeSet/TreeMap red-black tree (arbre binaire de recherche équilibré)		add, remove, contains, size	
LinkedHashSet-Map (tableaux et cellules)	add, remove, contains, size		

# Conclusion

- Distinguer **TDA** et **SD**
  - **TDA** : type abstrait de données (spécification représentée par une interface et des assertions)
  - **SD** : structure de données (organisation des données, représentée par des classes)
- Analyser le besoin en opérations, la complexité en place et en temps pour le choix du TDA et de la SD
- Utiliser les outils de Java pour une bonne mise en œuvre
  - généricité, assertions, exceptions, itérateurs, streams (Java 1.8)
  - intégration dans l'API, usage des interfaces et classes existantes