Types de données abstraits & Structures de données

Emmanuel Grolleau
Observatoire de Paris – LESIA – Service d'Informatique Scientifique

Inspiré de :
Cours d'Olivier Raynaud - Université Blaise Pascal
Algorithms and Data Structures - Douglas Wilhelm Harder

12/2015

Sommaire

- Type de Données Abstrait (TDA) / Structure de Données (SD)
- TDA Container
- TDA Liste
  - SD Vecteur
- TDA Pile
  - SD Liste simplement chainée
- TDA File
  - SD Liste doublement chainée
- TDA Double-ended queue
- TDA File de priorité
  - SD TAS Binaire
- TDA Ensemble
  - SD Table de hachage
- TDA Multi ensemble
- TDA Dictionnaire
  - SD Arbre équilibré
  - SD Arbre équilibré rouge et noir
- TDA Multimap

Sommaire

- **Type de Données Abstrait (TDA) / Structure de Données (SD)**
- TDA Container
- TDA Liste
  - SD Vecteur
- TDA Pile
  - SD Liste simplement chainée
- TDA File
  - SD Liste doublement chainée
- TDA Double-ended queue
- TDA File de priorité
  - SD TAS Binaire
- TDA Ensemble
  - SD Table de hachage
- TDA Multi ensemble
- TDA Dictionnaire
  - SD Arbre équilibré
  - SD Arbre équilibré rouge et noir
- TDA Multimap

Types de Données Abstraits (TDA)

- En génie logiciel, un **type de données abstrait** est une spécification mathématique d'un ensemble de données et de l'ensemble des opérations qu'elles peuvent effectuer.

- On qualifie d'abstrait ce type de données car il correspond à un cahier des charges qu'une **structure de données** doit ensuite implémenter.

- Abstract data types – ADT en anglais

Types de Données Abstraits (TDA)

- La donnée est vue du point de vue de **l'utilisateur**

- Le TDA définit :
  - les valeurs possibles des données
  - les opérations possibles sur les données

Types de Données Abstraits (TDA)

TDA courants :

- **Sans relation entre les objets**
  - Container ou collection
  - Ensemble, (Set), *non ordonné, élément unique*
  - Dictionnaire (Map), *non ordonné, élément unique*

- **Relation d'ordre linéaire**
  - Liste, pas de contrainte d'unicité
  - Pile, LIFO, (Stack), *pas de contrainte d'unicité*
  - File, FIFO (Queue), *pas de contrainte d'unicité*
  - Deque
  - Priority queue
- Multimap
- Multiset
- …

Structures de données (SD)

- Une **structure de données** est une structure logique destinée à contenir des données, afin de leur donner une organisation permettant de simplifier leur traitement.

- Une structure de données **implémente** concrètement un **type de données abstrait** TDA

Structures de données

- C'est le point de vue de l'implémentation du Type de Données Abstrait, donc de l'informaticien.

- Exemple : Tableau (Array), c'est fondamentalement une structure de données, car sa taille est limitée.

Structures de données

- Tableau, tableau cyclique
- Vecteur (dynamic array)
- Liste chainée, liste doublement chainée,
- Arbre de recherche,
- Arbre équilibré (Self-balancing binary tree), implémentation des listes
- Tas (heap)
- Table de hachage (hash table)
- …

Sommaire

- Type de Données Abstrait (TDA) / Structure de Données (SD)
- **TDA Container**
- TDA Liste
  - SD Vecteur
- TDA Pile
  - SD Liste simplement chainée
- TDA File
  - SD Liste doublement chainée
- TDA Double-ended queue
- TDA File de priorité
  - SD TAS Binaire
- TDA Ensemble
  - SD Table de hachage
- TDA Multi ensemble
- TDA Dictionnaire
  - SD Arbre équilibré
  - SD Arbre équilibré rouge et noir
- TDA Multimap

Container/Collection

TDA : Sans relation entre les objets

Représente une collection d'objets.

Les conteneurs sont utilisés pour stocker des objets sous une forme organisée qui suit des règles d'accès spécifiques.

Opérations :

- **SIZE** : renvoie le nombre d'éléments stockés dans le container
- **EMPTY** : renvoie « vrai » si la liste est vide, « faux » sinon.
- **ADD** : ajoute un élément dans le container
- **REMOVE** : supprime un élément du container
- **CLEAR** : supprime tous les objets du container
- **FIND** : détermine si un objet est présent dans le container.

Container/Collection

TDA : Sans relation entre les objets

C'est un TDA de haut niveau dont découle plusieurs autres TDA.

Il existe deux types de conteneurs :

- Les conteneurs séquentiels (une seule valeur) :
  - Listes, piles, files, ensembles (set)
- Les conteneurs associatifs (correspondance clef/valeurs):
  - Les dictionnaires (map)

Sommaire

- Type de Données Abstrait (TDA) / Structure de Données (SD)
- TDA Container
- **TDA Liste**
  - SD Vecteur
- TDA Pile
  - SD Liste simplement chainée
- TDA File
  - SD Liste doublement chainée
- TDA Double-ended queue
- TDA File de priorité
  - SD TAS Binaire
- TDA Ensemble
  - SD Table de hachage
- TDA Multi ensemble
- TDA Dictionnaire
  - SD Arbre équilibré
  - SD Arbre équilibré rouge et noir
- TDA Multimap

Liste

TDA : Relation d'ordre linéaire

Une liste représente une séquence de valeurs (hétérogène), où la même valeur peut apparaitre plusieurs fois.

Opérations de base :

- **ADD** : ajoute un élément dans la liste.
- **REMOVE** : supprimer un élément de la liste.
- **EMPTY** : renvoie « vrai » si la liste est vide, « faux » sinon.
- **SIZE** : renvoie le nombre d'éléments dans la liste.

Liste

TDA : Relation d'ordre linéaire

<u>Opérations auxiliaires fréquemment rencontrées :</u>

- **FIRST** : retourne le premier élément dans la liste.
- **LAST** : retourne le dernier élément dans la liste.
- **NEXT** : retourne le prochain élément dans la liste.
- **PREVIOUS** : retourne l'élément qui précède dans la liste.
- **FIND** : cherche si un élément précis est contenu dans la liste et le retourne.

Liste

TDA : Relation d'ordre linéaire

Structure de données d'implémentation :

- Liste chainée
- Vecteur (dynamic array) (efficace pour trouver une valeur à un index précis mais pas pour les tests d'appartenance.
- Arbre équilibré (Self-balancing binary tree), si l'on souhaite faire des tests d'appartenance.

Liste

TDA : Relation d'ordre linéaire

Langage :

- Python : liste, a= [3,4,2,3,'toto'].
  - La liste python est indexée, il s'agit donc d'une implémentation de liste de type vecteur, temps de recherche par index constant. **Elle ne stocke qu'une référence aux objets, sa taille n'augmente qu'en fonction du nombre d'objet** mais pas de la taille des objets stockés.
  - Mauvaise performance pour les tests d'appartenance.
- C++ :
  - std::list : liste doublement chainée, temps d'insertion constant.
  - std::forward_list : Liste simplement chainée
- JAVA
  - Type abstrait : java.util.AbstractList<E>
    - java.util.LinkedList : liste doublement chainée
    - java.util.ArrayList<E> : Vecteur

Liste

TDA : Relation d'ordre linéaire

Python:

La liste python est indexée

A = [**2**,4,'titi', 5, **2**, True] => Out: [2, 4, 'titi', 5, 2, True]

A[2:4] => Out : ['titi', 5]

A.append(10) => Out: [2, 4, 'titi', 5, 2, True, **10**]

Vecteur (dynamic array)

Structure de données

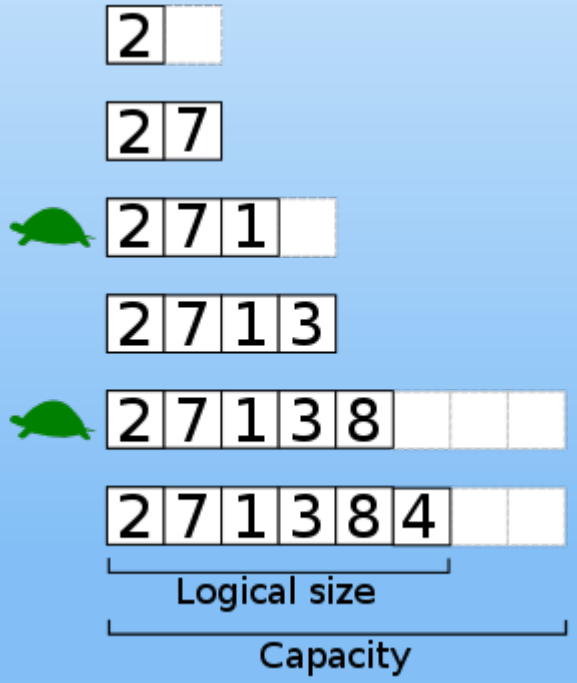
- En informatique, un **vecteur** désigne un conteneur d'éléments ordonnés et accessibles par des indices, dont la taille est dynamique : elle est mise à jour automatiquement lors d'ajouts ou de suppressions d'éléments.

- Typiquement, il s'agit d'un tableau à taille fixe au départ dont on alloue un espace supplémentaire en cas de besoin correspondant à un facteur de croissance fixé (classiquement growth factor = 2).

Vecteur (dynamic array)

Structure de données

- Exemple avec un facteur 2



Vecteur (dynamic array)

Structure de données

Implementation	Growth Factor (a)
Java ArrayList	3/2
Python PyListObject	9/8
Microsoft VC++ 2003 vector	3/2
GCC 5.2.0	2
Clang 3.6	2
Facebook folly/FBVector	3/2

Vecteur (dynamic array)

Structure de données

- **Complexité**

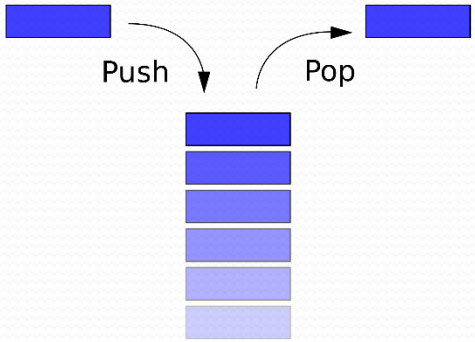
Opérations	Complexité	Justifications
ADD FIRST	**O(n)**	Obligation de déplacer tous les éléments suivants
ADD LAST	O(1)	
REMOVE FIRST	**O(n)**	Obligation de déplacer tous les éléments suivants
REMOVE LAST	O(1)	
PEEK FIRST	O(1)	
PEEK LAST	O(1)	
ADD/REMOVE élément au milieu	O(n)	Obligation de déplacer tous les éléments suivants
GET élément au milieu	O(1)	Accès direct par index
FIND	O(n) pire cas	

Sommaire

- Type de Données Abstrait (TDA) / Structure de Données (SD)
- TDA Container
- TDA Liste
  - SD Vecteur
- **TDA Pile**
  - SD Liste simplement chainée
- TDA File
  - SD Liste doublement chainée
- TDA Double-ended queue
- TDA File de priorité
  - SD TAS Binaire
- TDA Ensemble
  - SD Table de hachage
- TDA Multi ensemble
- TDA Dictionnaire
  - SD Arbre équilibré
  - SD Arbre équilibré rouge et noir
- TDA Multimap

Pile, LIFO (Stack)

TDA : Relation d'ordre linéaire



Opérations

- **ADD, ENQUEUE (push) :** ajoute un élément en queue de file
- **REMOVE, DEQUEUE (pop) :** supprime l'élément en tête de file (sans argument)
- **EMPTY** : renvoie « vrai » si la file est vide, « faux » sinon.
- **SIZE** : renvoie le nombre d'éléments de la file.

Opérations auxiliaires fréquemment rencontrées :

- **PEEK** : renvoie l'élément en haut de la pile sans le supprimer

Pile, LIFO (Stack)

TDA : Relation d'ordre linéaire

Structure de données d'implémentation :

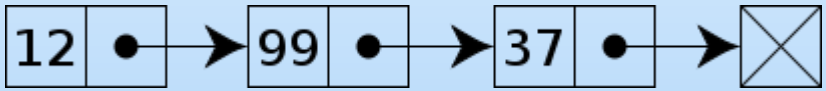
- Tableau cyclique : mais taille de la file définie à la création
- Liste simplement chainée : Efficace pour l'insertion et la suppression du dernier élément inséré.

Langage :

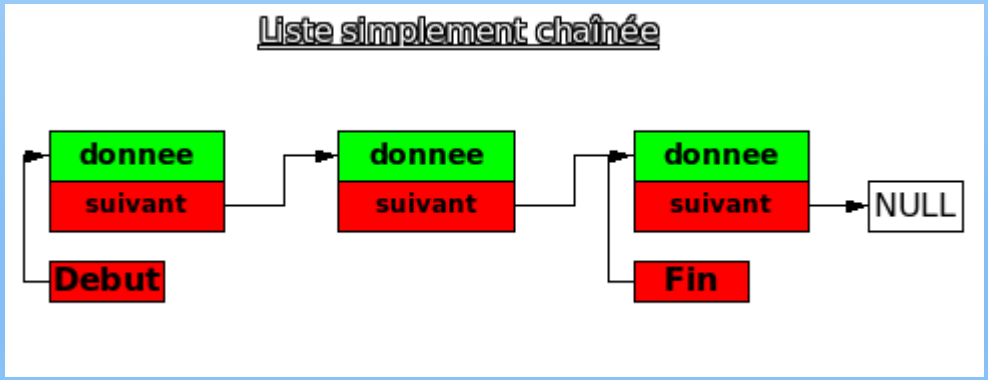
- Python
  - Queue.LifoQueue, implémentation par ?
- C++
  - std::stack, implementation par défaut : Vecteur
- Java
  - java.util.Stack<E>, file implémentée par un tableau

Liste simplement chainée

Structure de données



Les listes chaînées sont des structures de données semblables aux tableaux sauf que l'accès à un élément ne se fait pas par *index* mais à l'aide d'un *pointeur*.



Liste simplement chainée

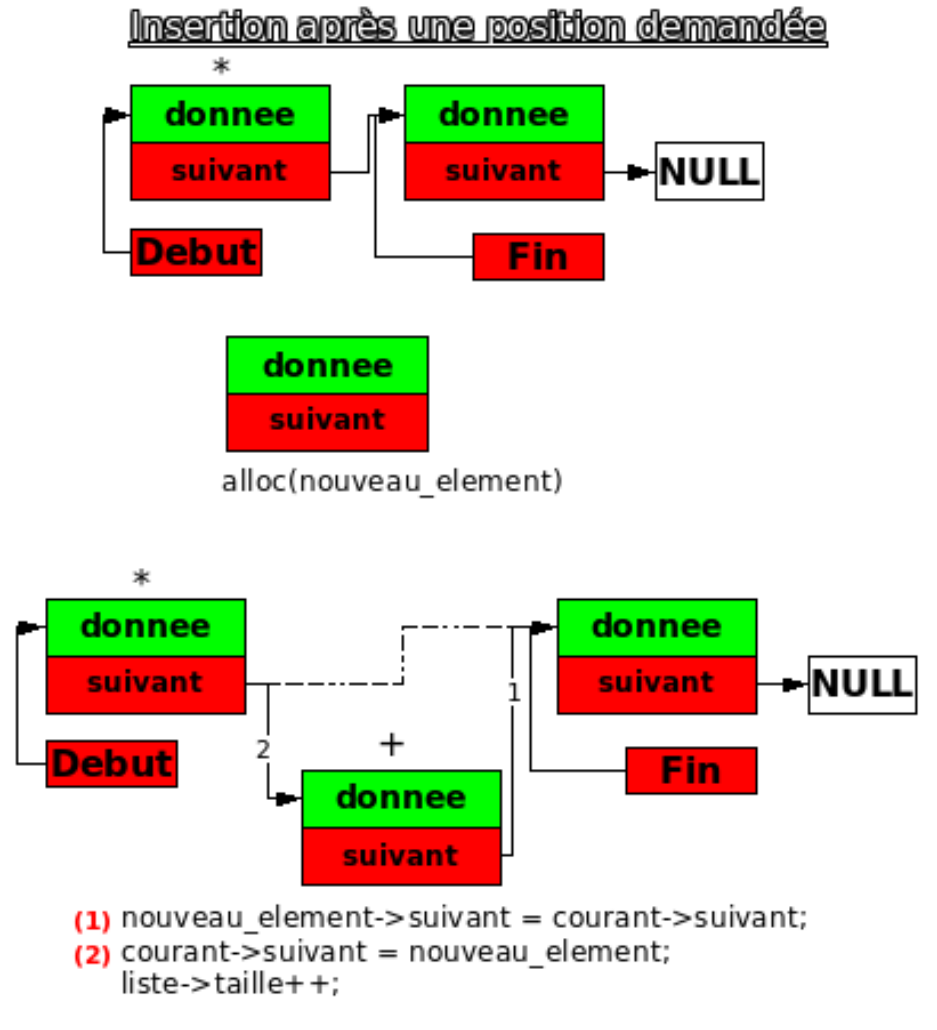
Structure de données

- Par rapport aux tableaux où les éléments sont contigus dans la mémoire, les éléments d'une liste sont éparpillés dans la mémoire.
  La liaison entre les éléments se fait grâce à un pointeur.
  Dans la mémoire la représentation est aléatoire en fonction de l'espace alloué.

- Pour accéder à un élément la liste est parcourue en commençant avec la tête, le pointeur suivant permettant le déplacement vers le prochain élément.
  Le déplacement se fait **dans une seule direction**, du premier vers le dernier élément.

Liste simplement chainée

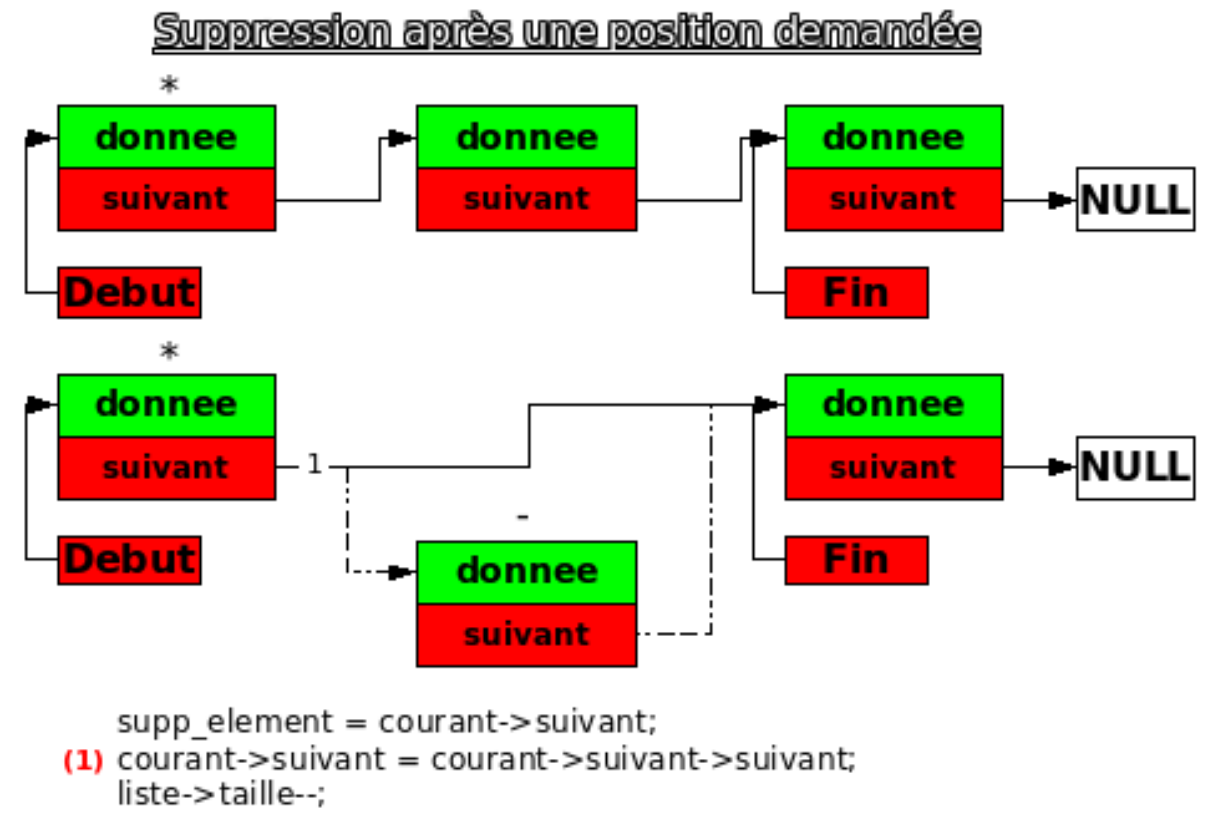
Structure de données



- **ADD** n'importe où dans la liste
- Complexité : O(1), moyennant le coût de la recherche de l'emplacement

Liste simplement chainée

Structure de données



- **REMOVE** n'importe où dans la liste
- Complexité : O(1), moyennant le coût de la recherche de l'emplacement

Liste simplement chainée

Structure de données

- **Complexité**

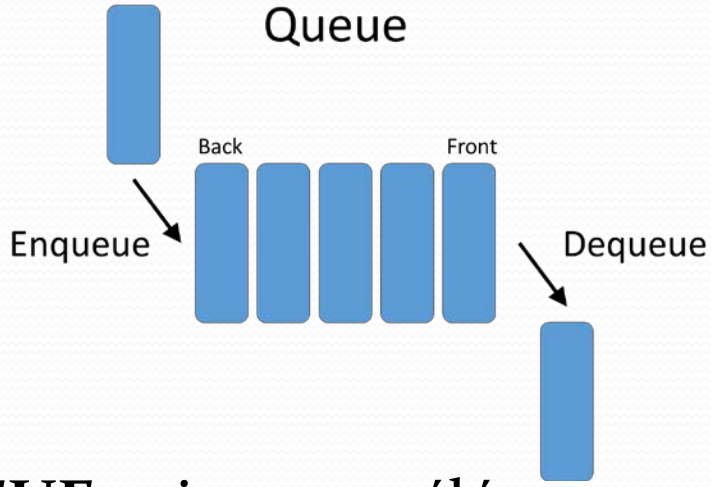
Opérations	Complexité	Justifications
ADD FIRST	**O(n)**	On doit parcourir toute la liste pour trouver le premier élément, puis insertion en O(1)
ADD LAST	O(1)	
REMOVE FIRST	**O(n)**	On doit parcourir toute la liste pour trouver le premier élément, puis suppression en O(1)
REMOVE LAST	O(1)	
PEEK FIRST	**O(n)**	On doit parcourir toute la liste pour trouver le premier élément
PEEK LAST	O(1)	
ADD/REMOVE élément au milieu	**O(n)**	On doit parcourir la liste pour trouver l'élément, puis opération en O(1)
GET élément au milieu	**O(n)**	On doit parcourir la liste pour trouver l'élément

Sommaire

- Type de Données Abstrait (TDA) / Structure de Données (SD)
- TDA Container
- TDA Liste
  - SD Vecteur
- TDA Pile
  - SD Liste simplement chainée
- **TDA File**
  - SD Liste doublement chainée
- TDA Double-ended queue
- TDA File de priorité
  - SD TAS Binaire
- TDA Ensemble
  - SD Table de hachage
- TDA Multi ensemble
- TDA Dictionnaire
  - SD Arbre équilibré
  - SD Arbre équilibré rouge et noir
- TDA Multimap

File, FIFO (Queue)

TDA : Relation d'ordre linéaire



Opérations

- **ADD, ENQUEUE :** ajoute un élément en queue de file
- **REMOVE, DEQUEUE :** supprime l'élément en tête de file (sans argument)
- **EMPTY** : renvoie « vrai » si la file est vide, « faux » sinon.
- **SIZE** : renvoie le nombre d'éléments de la file.

Opérations auxiliaires fréquemment rencontrées :

- **PEEK** : renvoie l'élément en tête de file sans le supprimer

File, FIFO (Queue)

TDA : Relation d'ordre linéaire

Structure de données d'implémentation :

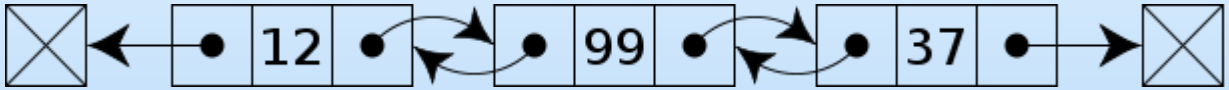
- Tableau cyclique : impose une taille de la file définie à la création
- Liste doublement chainée : Efficace pour l'insertion et la suppression des 2 côtés

Langage :

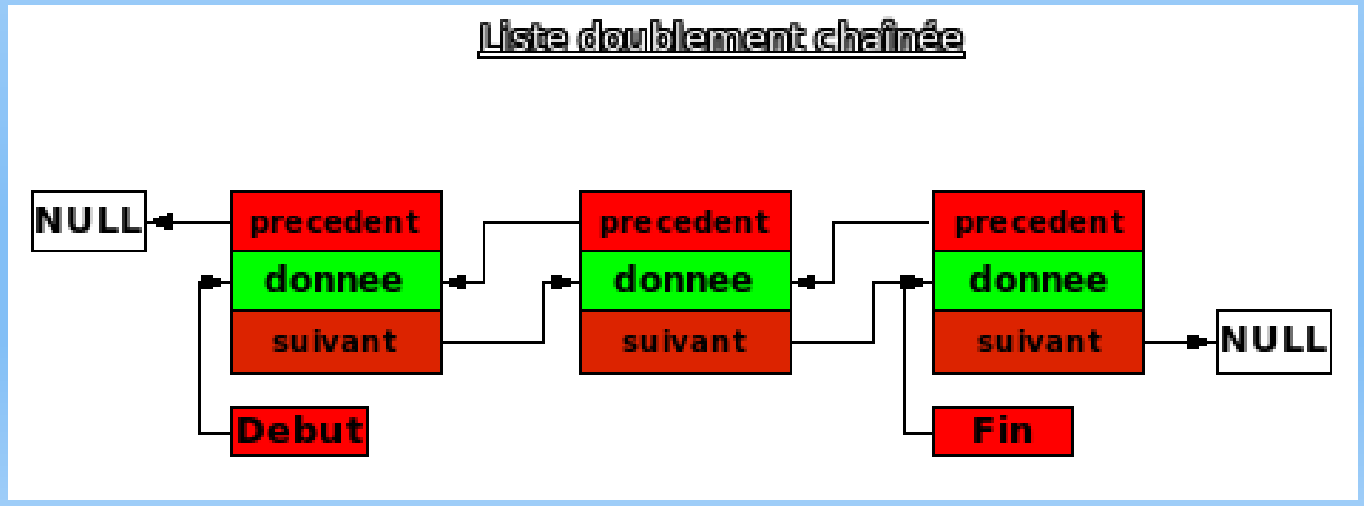
- Python
  - Queue.Queue, implémentation par ? probablement liste doublement chainée
- C++
  - std::queue, implementation par défaut : Vecteur
- Java
  - Type abstrait : java.util.AbstractQueue<E>
    - ArrayBlockingQueue<E>, file implémentée par un tableau, si la file est pleine l'insertion est bloquante (typiquement buffer de taille limitée).
    - LinkedBlockingQueue<E>, file implémentée par liste chainée

Liste doublement chainée

Structure de données



Les listes doublement chaînées sont des structures de données semblables aux listes simplement chaînées . En revanche la liaison entre les éléments se fait grâce **à deux pointeurs** (un qui pointe vers l'élément précédent et un qui pointe vers l'élément suivant).



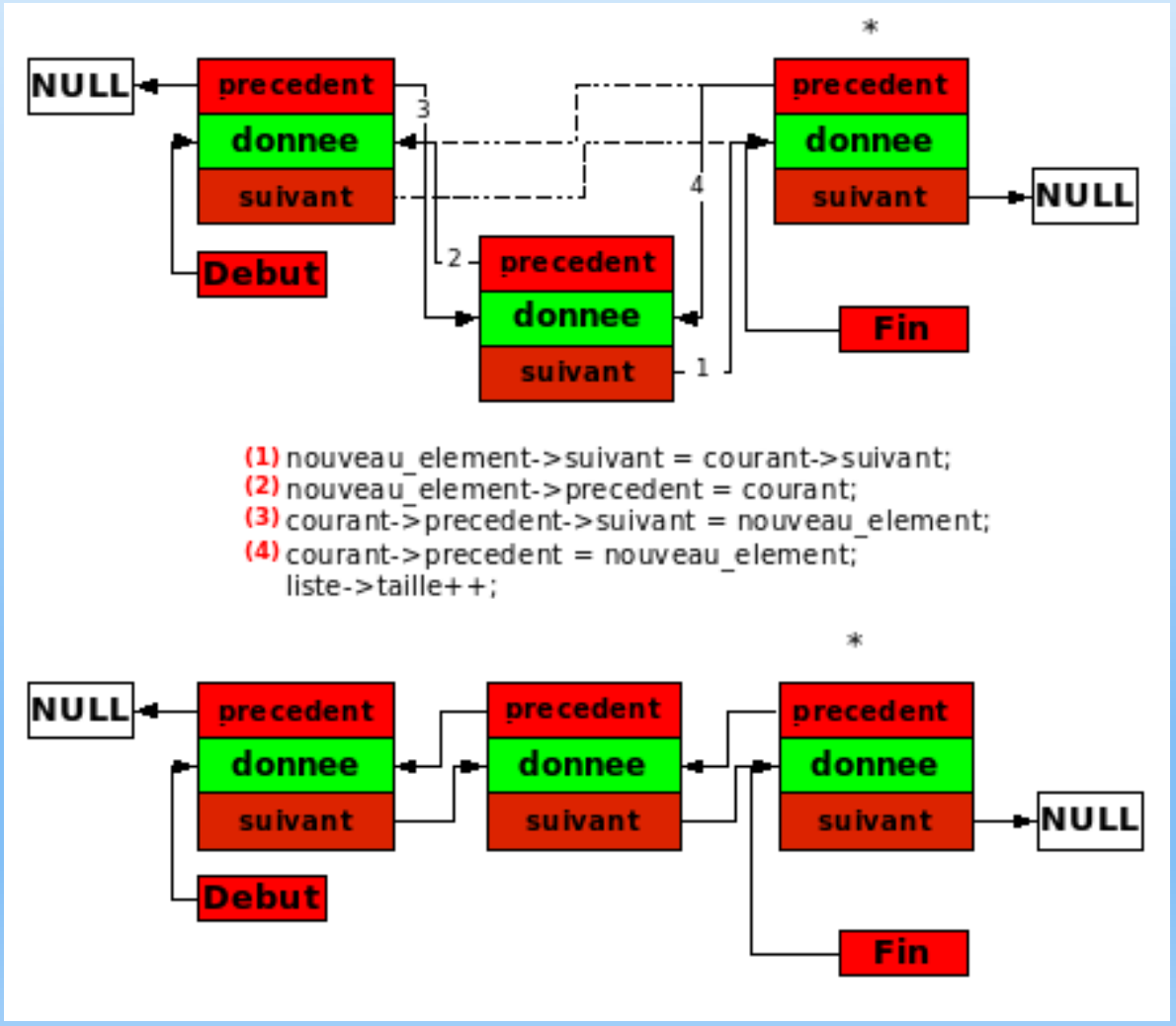
Liste doublement chainée

Structure de données

- Pour accéder à un élément la liste peut être parcourue **dans les deux sens** :
  - en commençant avec la tête, le pointeur suivant permettant le déplacement vers le prochain élément.
  - en commençant avec la queue, le pointeur précédent permettant le déplacement vers l'élément précédent.

Liste doublement chainée

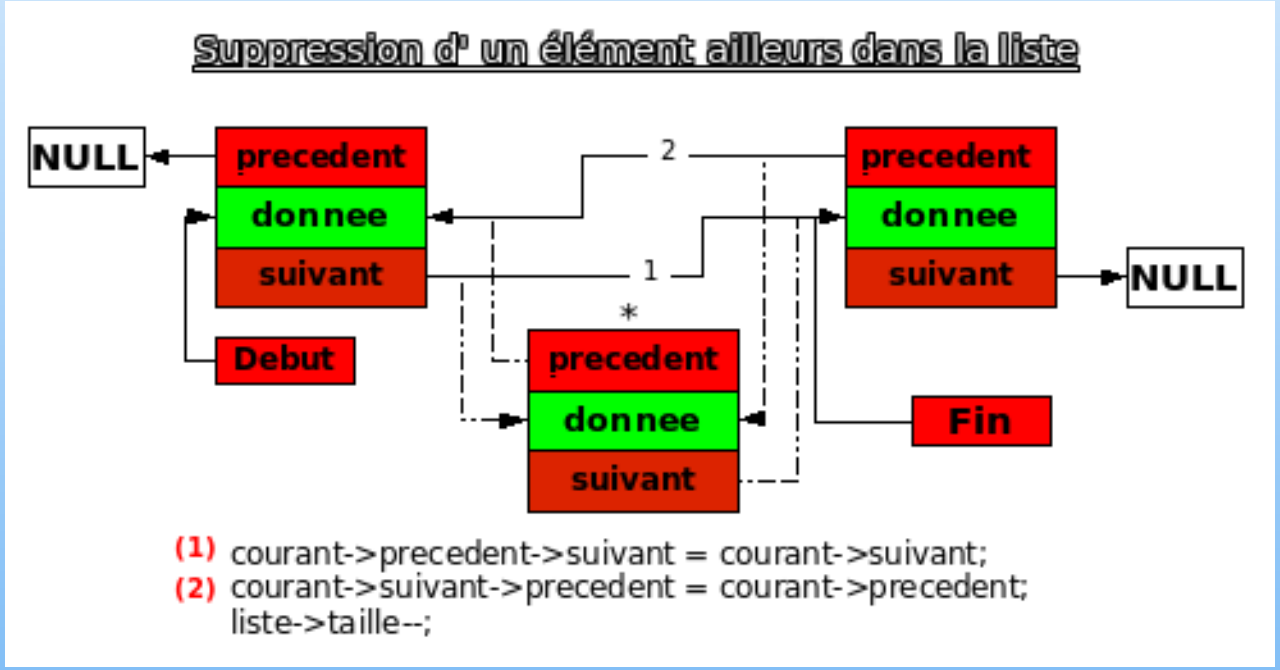
Structure de données



- **ADD** n'importe où dans la liste
- Complexité : O(1), moyennant le coût de la recherche de l'emplacement

Liste doublement chainée

Structure de données



- **REMOVE** n'importe où dans la liste
- Complexité : O(1), moyennant le coût de la recherche de l'emplacement

Liste doublement chainée

Structure de données

- **Complexité**

Opérations	Complexité	Justifications
ADD FIRST	O(1)	
ADD LAST	O(1)	
REMOVE FIRST	O(1)	
REMOVE LAST	O(1)	
PEEK FIRST	O(1)	
PEEK LAST	O(1)	
ADD/REMOVE élément au milieu	**O(n)**	On doit parcourir la liste pour trouver l'élément, puis opération en O(1)
GET élément au milieu	**O(n)**	On doit parcourir la liste pour trouver l'élément

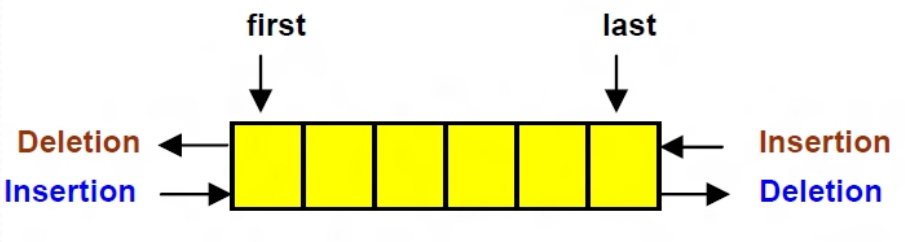
Sommaire

- Type de Données Abstrait (TDA) / Structure de Données (SD)
- TDA Container
- TDA Liste
  - SD Vecteur
- TDA Pile
  - SD Liste simplement chainée
- TDA File
  - SD Liste doublement chainée
- **TDA Double-ended queue**
- TDA File de priorité
  - SD TAS Binaire
- TDA Ensemble
  - SD Table de hachage
- TDA Multi ensemble
- TDA Dictionnaire
  - SD Arbre équilibré
  - SD Arbre équilibré rouge et noir
- TDA Multimap

Double-ended queue (deque)

TDA : Relation d'ordre linéaire

- Généralise la File avec opérations d'ajout et de suppression possibles sur les 2 extrémités.



Opérations

- **ADD FIRST, ADD LAST:** ajoute un élément dans la file respectivement en tête et en queue de file.
- **REMOVE FIRST, REMOVE LAST:** supprime l'élément respectivement en tête et en queue de file.
- **PEEK LAST, PEEK FIRST**: renvoie l'élément respectivement en tête et en queue de file.

Double-ended queue (deque)

TDA : Relation d'ordre linéaire

Structure de données d'implémentation :

- Vecteur (dynamic array)
- Liste doublement chainée (head-tail linked list)

Langage :

- Python
  - from collections import deque
- *C++*
  - *Std:deque, implémentation hybride*
- *JAVA*
  - *java.util.ArrayDeque<E>*, Vecteur

Vecteur vs liste chainée

Structure de données

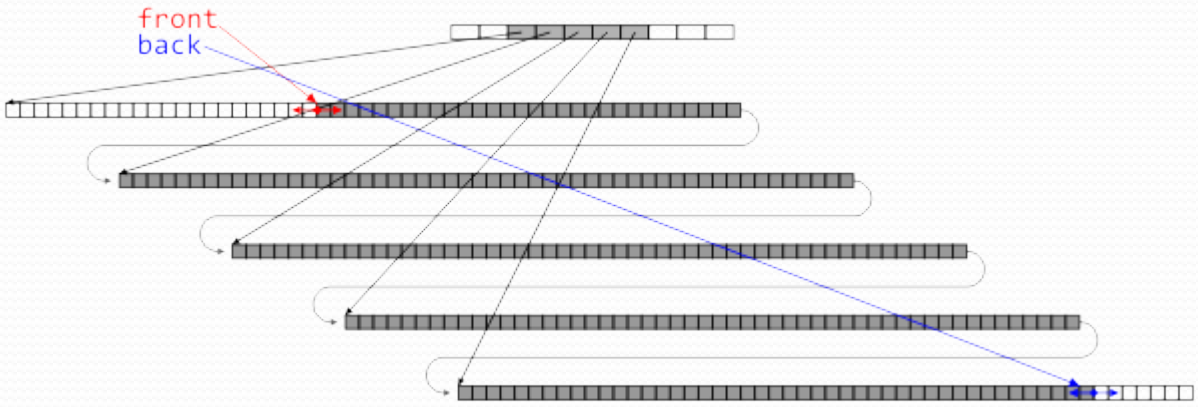
- **Complexité**

Opérations	Vecteur	Liste chainée	Liste doublement chainée
ADD FIRST	**O(n)**	**O(n)**	O(1)
ADD LAST	O(1)	O(1)	O(1)
REMOVE FIRST	**O(n)**	**O(n)**	O(1)
REMOVE LAST	O(1)	O(1)	O(1)
PEEK FIRST	O(1)	**O(n)**	O(1)
PEEK LAST	O(1)	O(1)	O(1)
ADD/REMOVE élément au milieu	**O(n)**	**O(n)**	**O(n)**
GET élément au milieu	O(1)	**O(n)**	**O(n)**

Double-ended queue (deque)

TDA : Relation d'ordre linéaire

- Spécificité de l'implémentation en C : Hybride tableau dynamique chainé. (A l'opposé du type Vecteur = uniquement tableau dynamique).

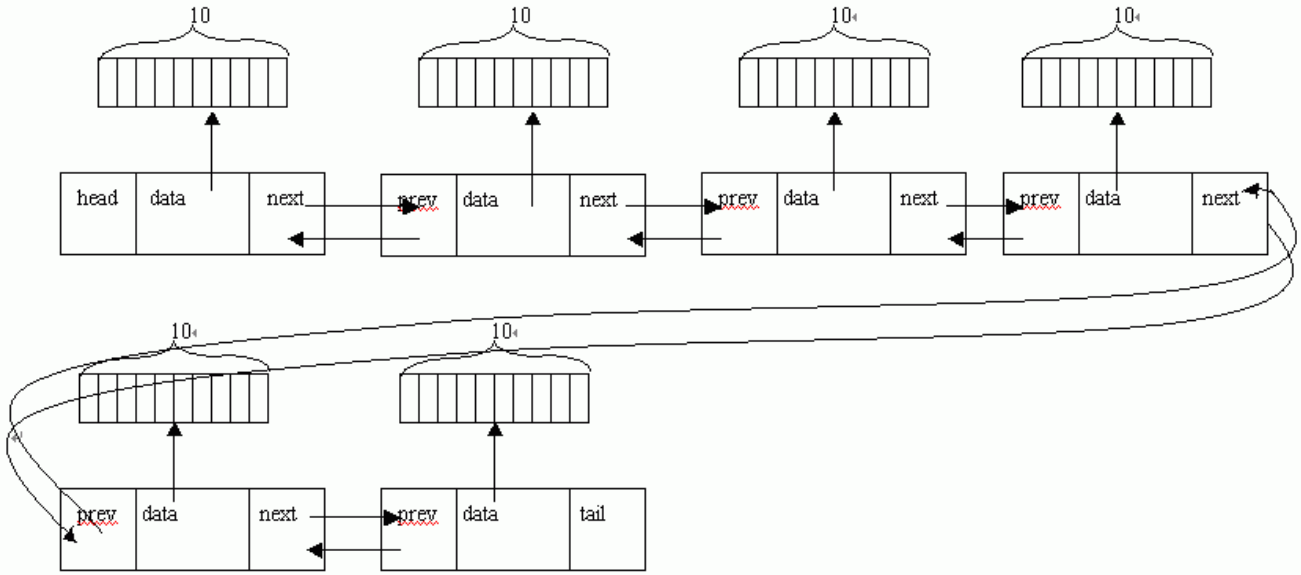


- Permet d'avoir moins d'allocations mémoires à chaque insertion/suppression (on alloue un nouveau tableau si nécessaire). Mauvaises performances si l'on doit insérer un élément au milieu de la file (a priori non nécessaire dans un TDA Deque).

Double-ended queue (deque)

TDA : Relation d'ordre linéaire

- Autre implémentation

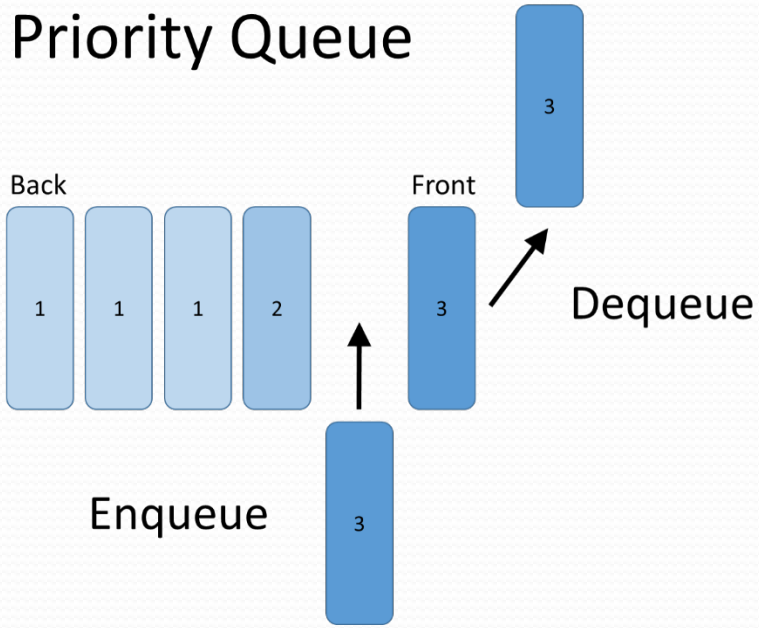


Sommaire

- Type de Données Abstrait (TDA) / Structure de Données (SD)
- TDA Container
- TDA Liste
  - SD Vecteur
- TDA Pile
  - SD Liste simplement chainée
- TDA File
  - SD Liste doublement chainée
- TDA Double-ended queue
- **TDA File de priorité**
  - SD TAS Binaire
- TDA Ensemble
  - SD Table de hachage
- TDA Multi ensemble
- TDA Dictionnaire
  - SD Arbre équilibré
  - SD Arbre équilibré rouge et noir
- TDA Multimap

File de priorité (Priority queue)

TDA : Relation d'ordre linéaire



- C'est une file qui possède pour chaque élément un attribut : Priorité.

File de priorité (Priority queue)

TDA : Relation d'ordre linéaire

Opérations

- **ADD, ENQUEUE (with priority) :** ajoute un élément dans la file avec une priorité, la position de l'élément correspond à sa priorité.
- **REMOVE, DEQUEUE (pull_highest_priority_element) :** supprime l'élément qui a la plus grande priorité et le retourne
- **EMPTY** : renvoie « vrai » si la file est vide, « faux » sinon.
- **SIZE** : renvoie le nombre d'éléments de la file.

Opérations auxiliaires fréquemment rencontrées :

- **PEEK** : renvoie l'élément de plus haute priorité sans modifier la file

Remarque : Une **file** peut-être vue comme une file de priorité où chaque élément est inséré avec une priorité monotone **décroissante**

Une **pile** peut-être vue comme une file de priorité où chaque élément est inséré avec une priorité monotone **croissante**.

File de priorité (Priority queue)

TDA : Relation d'ordre linéaire

Structure de données d'implémentation :

- Vecteur, liste, TAS (Binary Heap)

Langage :

- Python
  - Queue.PriorityQueue : TAS (module heapq)
- C++
  - TAS basé sur un tableau
- Java
  - java.util.PriorityQueue<E>, implémentation non précisée, mais :
    - add, remove en $O(\log_2(n))$,
    - peek, size en $O(1)$

TAS binaire (Binary Heap)

Structure de données

- Un **TAS binaire** est une structure de données de type arbre binaire, généralement stockée sous forme d'un tableau.
  - c'est un **arbre binaire parfait** : tous les niveaux excepté le dernier doivent être totalement remplis et si le dernier n'est pas totalement remplis alors il doit être rempli de gauche à droite
  - c'est un **tas** : l'étiquette (qu'on appelle aussi clé ou key) de chaque nœud doit être supérieure ou égale (resp. inférieure ou égale) aux étiquettes de chacun de ses fils (la signification de supérieur ou égal dépend de la relation d'ordre choisie)

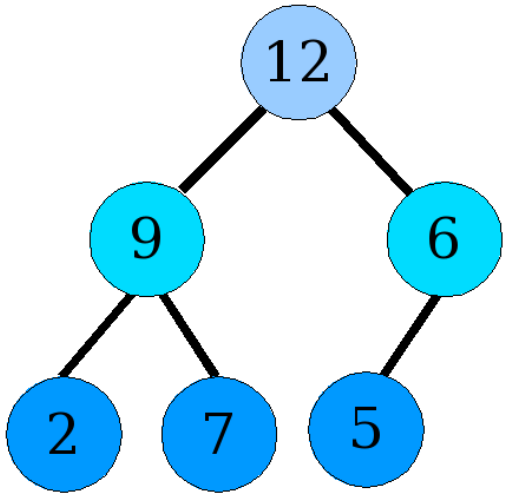
TAS binaire (Binary Heap)

Structure de données

Stockage d'un tas dans un tableau :

0	1	2	3	4	5
12	9	6	2	7	5

Représentation équivalente sous forme d'arbre :



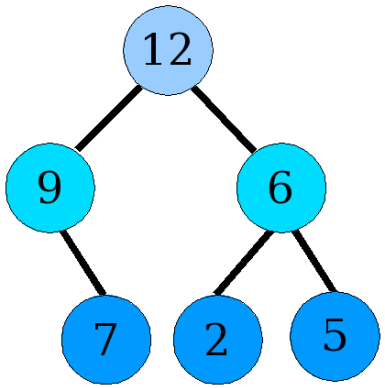
Pour tous nœuds A et B de l'arbre tels que B soit un fils de A :
clé(A) ≥ clé(B)

TAS binaire (Binary Heap)

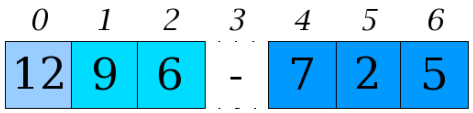
Structure de données

- Contre exemple :

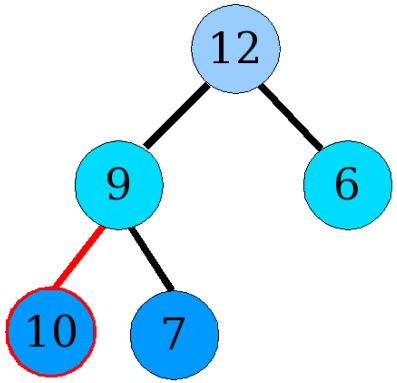
Cet arbre n'est pas un tas, car il n'est pas complet calé à gauche :



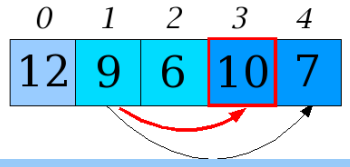
La représentation en tableau comporte un « trou », ce qui est interdit :



Cet arbre n'est pas un tas, car il viole la propriété de tas : un noeud de valeur 10 ne devrait pas être fils d'un noeud de valeur 9.



Représentation en tableau : valeur(filsgauche(1)) > valeur(1)



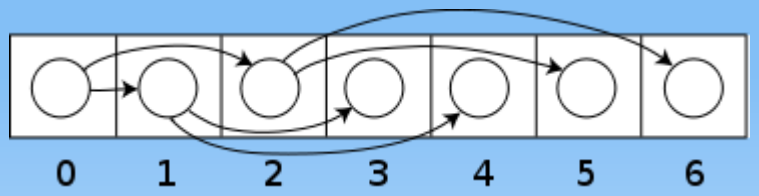
TAS binaire (Binary Heap)

Structure de données

Implémentation

Un tas binaire étant un arbre binaire parfait, on peut donc l'implémenter de manière compacte avec un tableau dynamique.

- La racine se situe à l'index 0
- Soit un nœud à l'index i alors son fils gauche est à l'index 2i+1 et son fils droit à 2i+2
- Soit un nœud à l'index i>0 alors son père est à l'index (i-1)/2



TAS binaire (Binary Heap)

Structure de données

Cette structure permet de retrouver directement l'élément que l'on veut traiter en priorité. En effet, on est sûr d'avoir en racine la clé maximale (dans le cas d'un *TAS maximal -max heap*) ou minimale (dans le cas d'un Tas minimal - *min heap*).

- **Complexité**

- **PEEK :** O(1)

TAS binaire (Binary Heap)

Structure de données

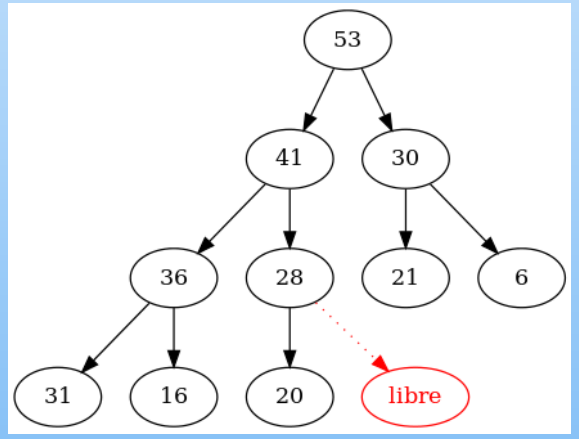
Complexité

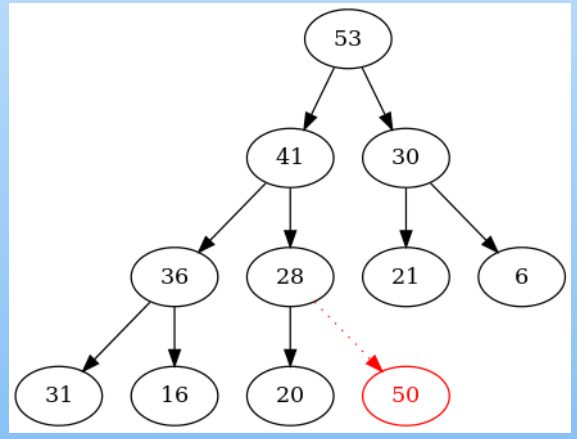
- **ADD :** $O(\log_2 n)$
- On insère x à la prochaine position libre (la position libre la plus à gauche possible sur le dernier niveau) puis on effectue l'opération suivante (que l'on appelle percolation vers le haut ou percolate-up) pour rétablir si nécessaire la propriété d'ordre du tas binaire :
  - tant que x n'est pas la racine de l'arbre et que x est strictement supérieur à son père on échange les positions entre x et son père.

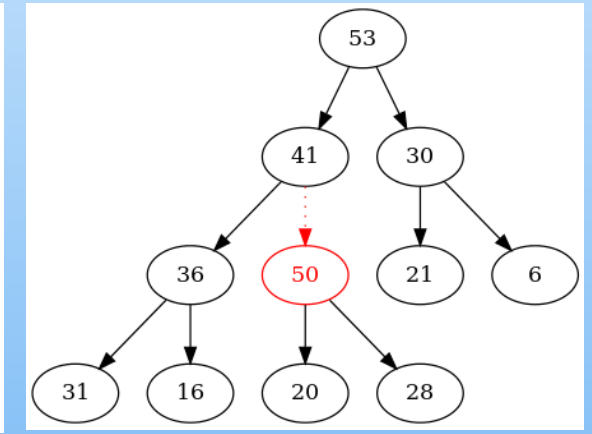
TAS binaire (Binary Heap)

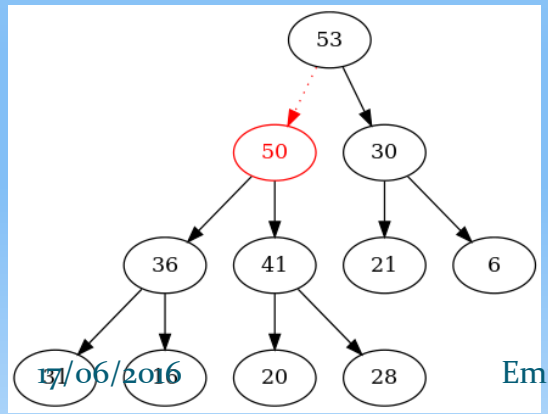
Structure de données

- **ADD :** $O(\log_2 n)$, exemple : on insère 50 dans un tas binaire max









Soit h la hauteur de notre tas binaire, lors de l'algorithme ci-dessus on effectue au plus h échanges. Or comme un tas binaire est un arbre binaire parfait on a $h <= \log_2(n)$ (où n est le nombre de nœuds du tas binaire donc la complexité est bien $O(\log_2 n)$.

TAS binaire (Binary Heap)

Structure de données

Complexité

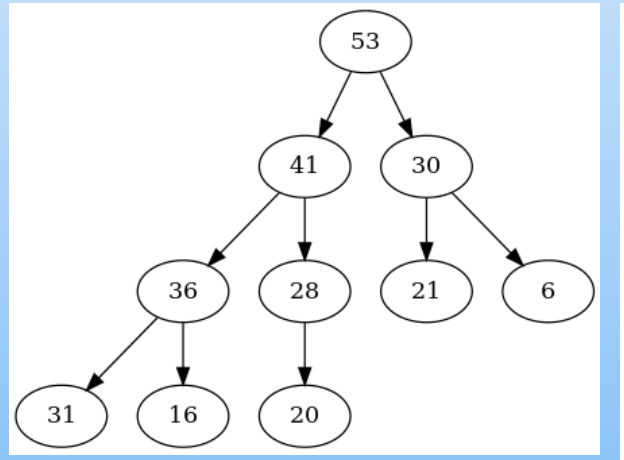
REMOVE : $O(\log_2 n)$

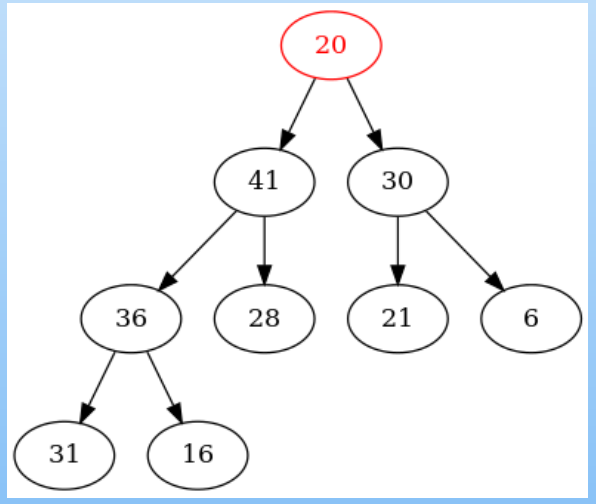
- On souhaite retirer la racine de notre tas binaire (c'est-à-dire le maximum de notre tas selon la relation d'ordre associée). Cependant il faut conserver la structure de tas binaire après la suppression, on procède donc de la manière suivante :
- On supprime la racine et on met à sa place le nœud qui était en dernière position de l'arbre binaire (donc le nœud le plus à droite sur le dernier niveau) que l'on notera x. On choisit ce nœud car nous souhaitons conserver la structure de TAS binaire. Puis on fait l'opération suivante (que l'on appelle percolation vers le bas ou percolate-down) :
  - tant que x a des fils et que x est strictement inférieur à un de ses fils, on échange les positions entre x et le plus grand de ses fils.

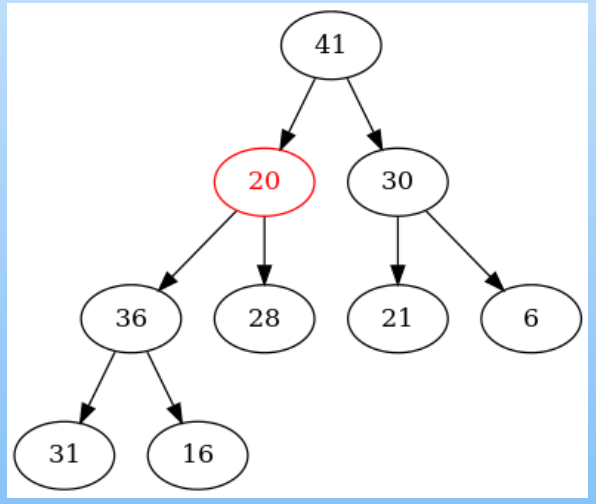
TAS binaire (Binary Heap)

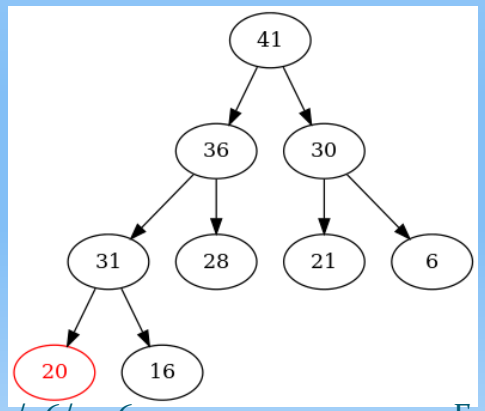
Structure de données

- **REMOVE:** O($\log_2 n$), exemple : On retire la racine









Par le même argument que pour l'algorithme de ADD, on fait au plus h échange donc la complexité est bien O(log n)

TAS binaire (Binary Heap)

Structure de données

<u>Pour info :</u>

Augmenter ou diminuer une clé :

- On peut augmenter ou diminuer la priorité (la clé) d'un nœud mais il faut ensuite satisfaire la contrainte d'ordre. Si l'on augmente la clé on fera donc un percolate-up à partir de notre nœud et si l'on diminue la clé on fera un percolate-down.
  - percolate-up : O(h(x)), où h(x) est la profondeur de x
  - percolate-down : O(h – h(x)), où h est la hauteur de l'arbre

Construire un TAS : O(n)

File de priorité (Priority queue)

TDA : Relation d'ordre linéaire

Structure de données	**Insérer()**	**Maximum()**	**extraireMax()**
Vecteur non ordonné	O(1)	O(n)	O(n)
Liste chainée non ordonnée	O(1)	O(n)	O(n)
Vecteur ordonné	O(n)	O(1)	O(1)
Liste chainée ordonnée	O(n)	O(1)	O(1)
Tas	$O(\log_2 n)$	O(1)	$O(\log_2 n)$

Complexité, pire cas

Sommaire

- Type de Données Abstrait (TDA) / Structure de Données (SD)
- TDA Container
- TDA Liste
  - SD Vecteur
- TDA Pile
  - SD Liste simplement chainée
- TDA File
  - SD Liste doublement chainée
- TDA Double-ended queue
- TDA File de priorité
  - SD TAS Binaire
- **TDA Ensemble**
  - SD Table de hachage
- TDA Multi ensemble
- TDA Dictionnaire
  - SD Arbre équilibré
  - SD Arbre équilibré rouge et noir
- TDA Multimap

TDA : Ensemble (Set)

TDA : Sans relation entre les objets

- Un **ensemble** (**set**) est un TDA qui peut stocker des valeurs, sans ordre particulier, et ne présentant aucun doublon.

- Les ensembles sont essentiellement utilisés pour tester l'appartenance d'une valeur à cet ensemble et non pour en extraire des données.

TDA : Ensemble (Set)

TDA : Sans relation entre les objets

Opérations de base :

- **SIZE** : renvoie le nombre d'éléments stockés dans l'ensemble
- **EMPTY** : renvoie « vrai » si l'ensemble est vide, « faux » sinon.
- **ADD** : ajoute un élément dans l'ensemble
- **REMOVE** : supprime un élément de l'ensemble
- **CLEAR** : supprime tous les éléments de l'ensemble container
- **MEMBER** : détermine si un élément est présent dans l'ensemble.

Opérations auxiliaires fréquemment rencontrées :

Toutes les operation sur les ensembles (Union, Intersection, difference)

TDA : Ensemble (Set)

TDA : Sans relation entre les objets

Structure de données d'implémentation :

- Arbre,
- Table de hachage

Langage :

- Python : set : différent des listes car non indexé (pas indexation, slices, …).
  - Implémentation par table de hachage où la clef et l'élément sont confondus
  - Optimisé pour les tests d'appartenance
- C++
  - std::unordered_set, table de hachage où la clef et l'élément sont confondus
- Java
  - java.util. AbstractSet
    - java.util.TreeSet<E> : implémentation par un arbre rouge et noir, trié par ordre naturel
    - java.util.HashSet<E> : implémentation par une table de hachage

TDA : Ensemble (Set)

TDA : Sans relation entre les objets

- **Python**
  - Création (2 méthodes)
    - S1 = {1, 3, 'aaa', 2 } => Out : set([1, 3, 'aaa', 2])
    - S2 = set([1,3,3,2, 'toto']) => Out: set([1, 2, 3, 'toto'])
  - Membre
    - 3 in S2 => Out: True
  - Différence
    - S1 – s2 => Out : set(['aaa'])
  - Union
    - S1 | S2 => Out : set([1, 2, 3, 'aaa', 'toto'])
  - Intersection
    - S1 & S2 => Out : set([1, 2, 3])

Table de hachage (hash table)

Structure de données

- Une **table de hachage** est une structure de données qui permet une association clé-élément.
- On accède à chaque élément de la table via sa **clé**.
- L'accès à un élément se fait en transformant la clé en une valeur de hachage par l'intermédiaire d'une **fonction de hachage**.
- La valeur de hachage est un nombre qui permet la localisation des éléments dans le tableau, typiquement le hachage est l'index de l'élément dans le tableau.
- Les algorithmes **MD5** et **SHA1** sont des fonctions de hachage célèbres

Table de hachage (hash table)

Structure de données

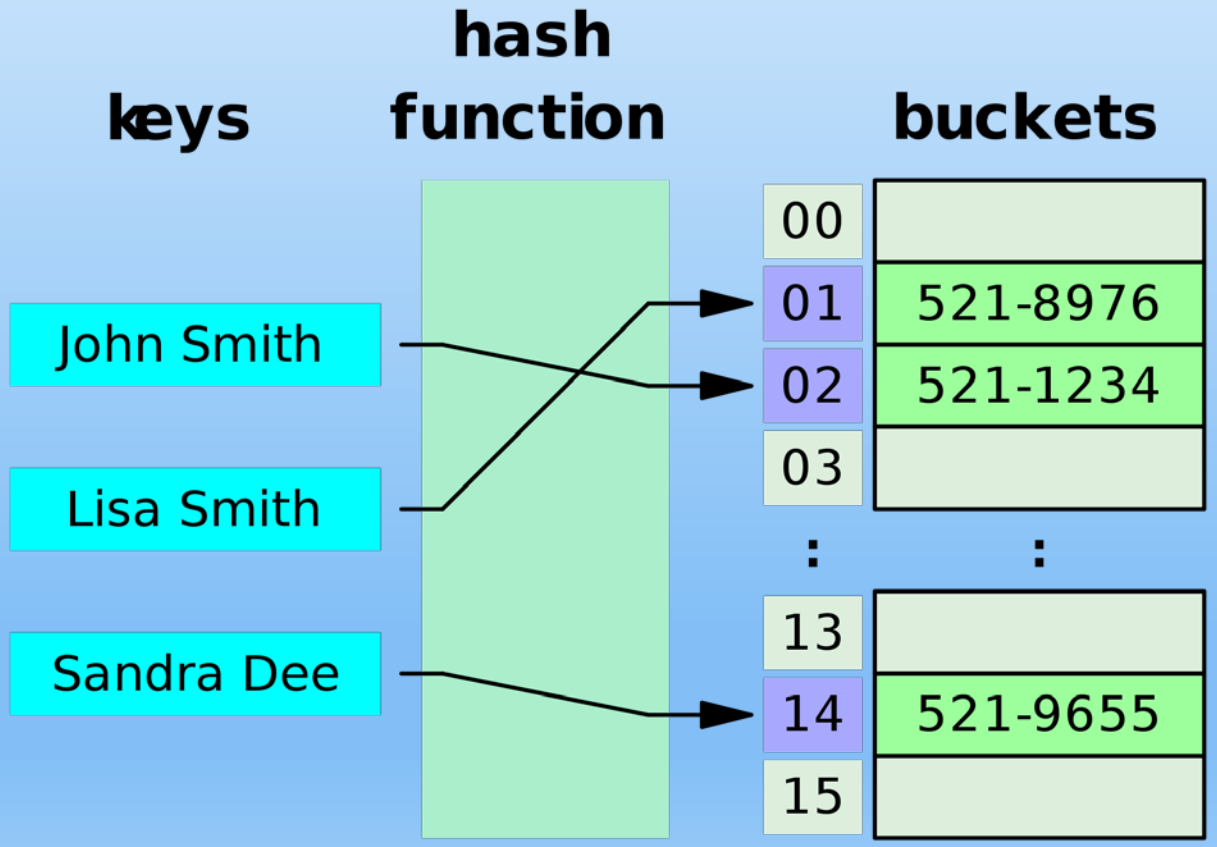


Table de hachage (hash table)

Structure de données

- Fonction de hachage
  - Une bonne fonction de hachage vérifie l'hypothèse de hachage uniforme simple (chaque clef a autant de chance d'être hachée dans l'une quelconque des m alvéoles.)

Exemple :

$$H(k) = k \text{ modulo } m$$

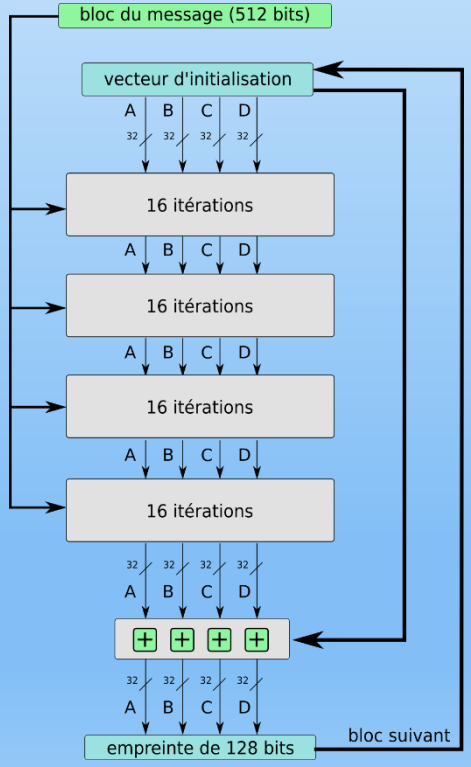
Si on prend m = 10000 (par exemple on utilise un tableau de taille 10000)

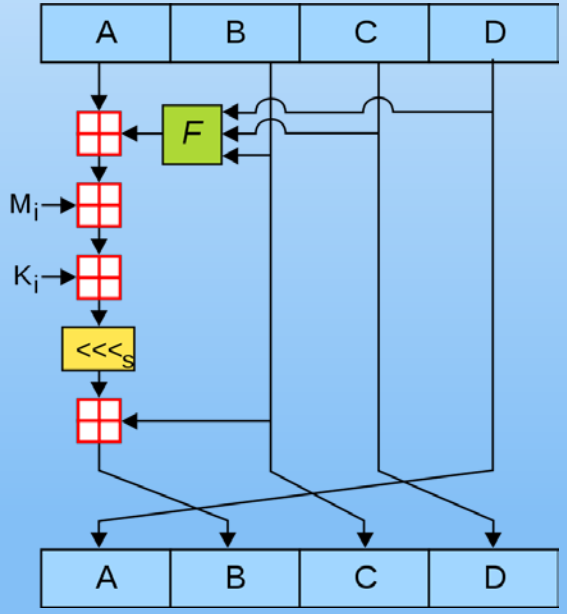
$$H(4\ 325\ 678) = H(25\ 678) = 5678$$

Table de hachage (hash table)

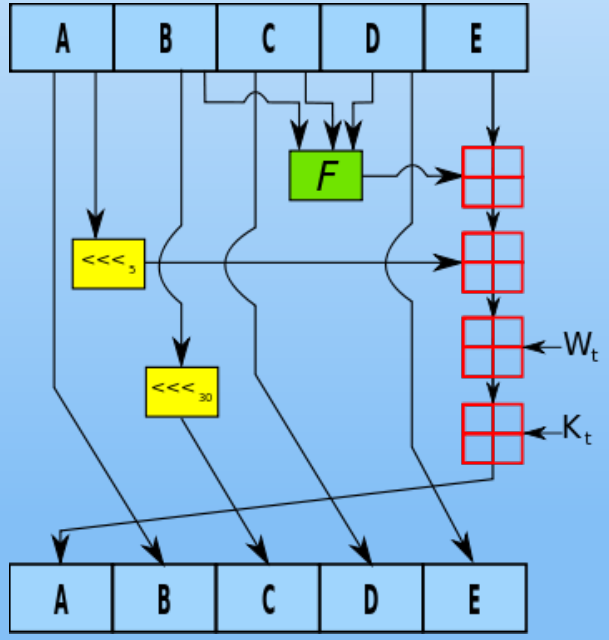
Structure de données

- Fonction de hachage





MD5



Un des 80 tours de **SHA1**

Table de hachage (hash table)

Structure de données

- Collision : Quand la fonction de hachage renvoie le même nombre pour deux clés différentes, on dit qu'il y a **collision**.
  - Le calcul probabiliste montre que *même si la fonction de hachage a une distribution parfaitement uniforme, il y a 95 % de chances d'avoir une collision dans une table de taille 1 million avant même qu'elle ne contienne 2 500 éléments*

- Résolution d'une collision : **le chaînage**
  - Une solution consiste à créer une **liste chaînée** à l'emplacement de la collision. Vous avez deux données (ou plus) à stocker dans la même case ? Utilisez une liste chaînée et créez un pointeur vers cette liste depuis le tableau.

Table de hachage (hash table)

Structure de données

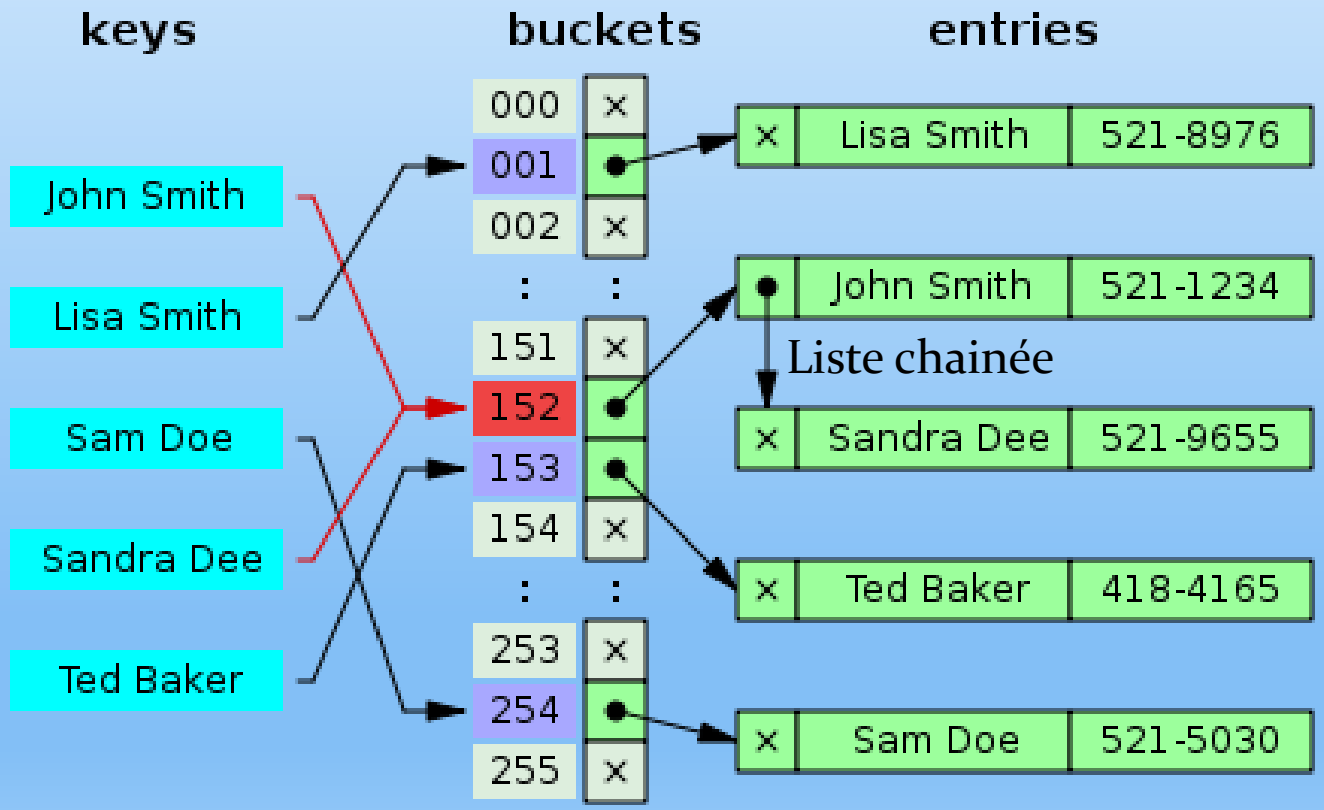


Table de hachage (hash table)

Structure de données

- **Complexité**
  - Soit α (facteur de remplissage) = $\frac{\text{nombre de clefs stockés}}{\text{taille de la table}} = \frac{n}{m}$
    = nombre moyen de clefs par case

Opérations	Complexité moyenne	Complexité dans le pire des cas (n collisions)
ADD	O(1)	O(n)
REMOVE	O(1+α)	O(n)
MEMBER	O(1+α)	O(n)
MAXIMUM	O(n)*	O(n)

* Sauf si les index renvoyés par la fonction de hachage sont dans le même ordre que les clefs, ce qui n'a aucune raison d'être le cas car il faudrait que l'espace des clefs soit aussi grand que celui des valeurs.

Table de hachage (hash table)

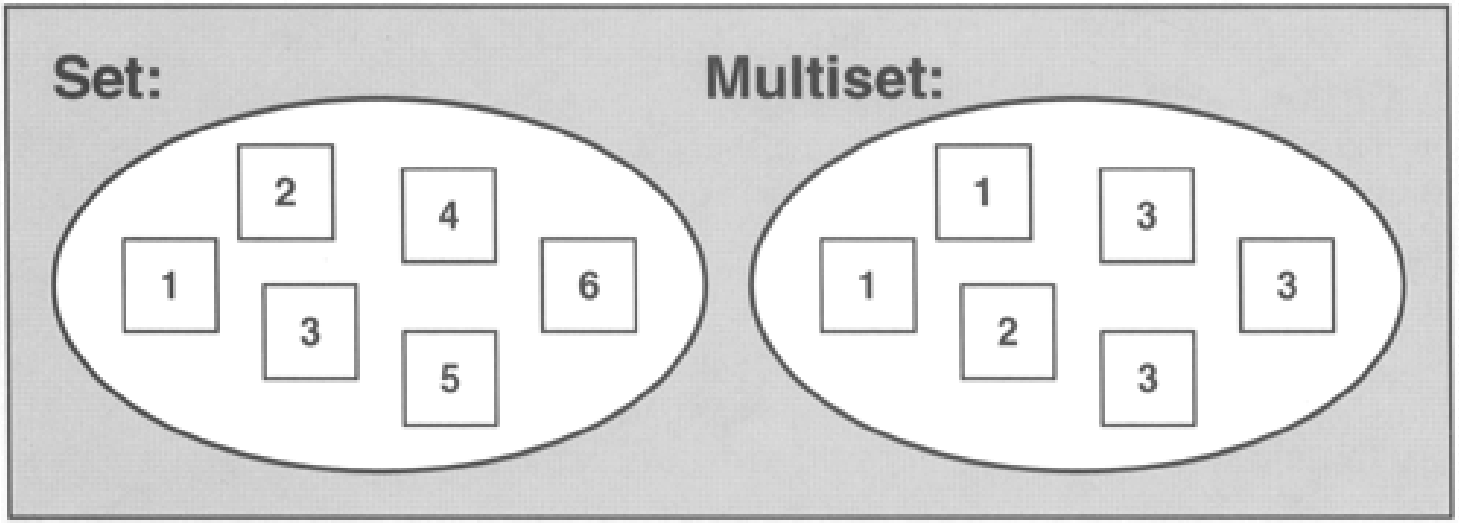
Structure de données

- Pourquoi les Ensembles sont-ils implémentés par table de hachage ?
- Parce que l'on souhaite le meilleur temps possible pour l'opération MEMBER, on ne souhaite pas d'opération sur l'ordre (MAX, MIN).

TDA : Multi-Ensemble (Multiset)

TDA : Sans relation entre les objets

- Un multi ensemble est un ensemble dans lequel chaque élément peut apparaitre plusieurs fois. Pas de contrainte d'unicité.



Sommaire

- Type de Données Abstrait (TDA) / Structure de Données (SD)
- TDA Container
- TDA Liste
  - SD Vecteur
- TDA Pile
  - SD Liste simplement chainée
- TDA File
  - SD Liste doublement chainée
- TDA Double-ended queue
- TDA File de priorité
  - SD TAS Binaire
- TDA Ensemble
  - SD Table de hachage
- TDA Multi ensemble
- **TDA Dictionnaire**
  - SD Arbre équilibré
  - SD Arbre équilibré rouge et noir
- TDA Multimap

TDA : Dictionnaire (Map)

TDA : Sans relation entre les objets

- Un dictionnaire est un TDA associant à un ensemble de clefs un ensemble correspondant de valeurs, sans ordre particulier, et ne présentant aucun doublon de clef.

- On parle également de tableau associatif.

TDA : Dictionnaire (Map)

TDA : Sans relation entre les objets

Opérations de base (Les mêmes qu'un ensemble + MODIFY) :

- **SIZE** : renvoie le nombre d'éléments stockés dans l'ensemble
- **EMPTY** : renvoie « vrai » si l'ensemble est vide, « faux » sinon.
- **ADD** : ajoute un élément dans l'ensemble
- **REMOVE** : supprime un élément de l'ensemble
- **CLEAR** : supprime tous les éléments de l'ensemble container
- **MEMBER** : détermine si un élément est présent dans l'ensemble.
- **MODIFY** (Key) : Modifie un objet à partir de sa clef (seule opération différente du SET)

TDA : Dictionnaire (Map)

TDA : Sans relation entre les objets

Structure de données d'implémentation :

- Table de hachage
- Arbre équilibré (typiquement rouge et noir)
- Liste chainée (valable uniquement si dictionnaire de petite taille)

Langage :

- Python : dictionaries
  - Implémentation par table de hachage (la clef peut-être un nombre ou une chaine)
  - Optimisé pour les tests d'appartenance
- C++
  - std::map, : implémenation pr arbre binaire Rouge et Noir
- Java
  - java.util. AbstractMap
    - java.util.HashMap <K,V> : implémentation par une table de hachage
    - java.util.TreeMap<K,V> : implémentation par un arbre équilibré rouge et noir et trié par clef
    - java.util.LinkedHashMap<K,V>: implémentation par une liste doublement chainée

TDA : Dictionnaire (Map)

TDA : Sans relation entre les objets

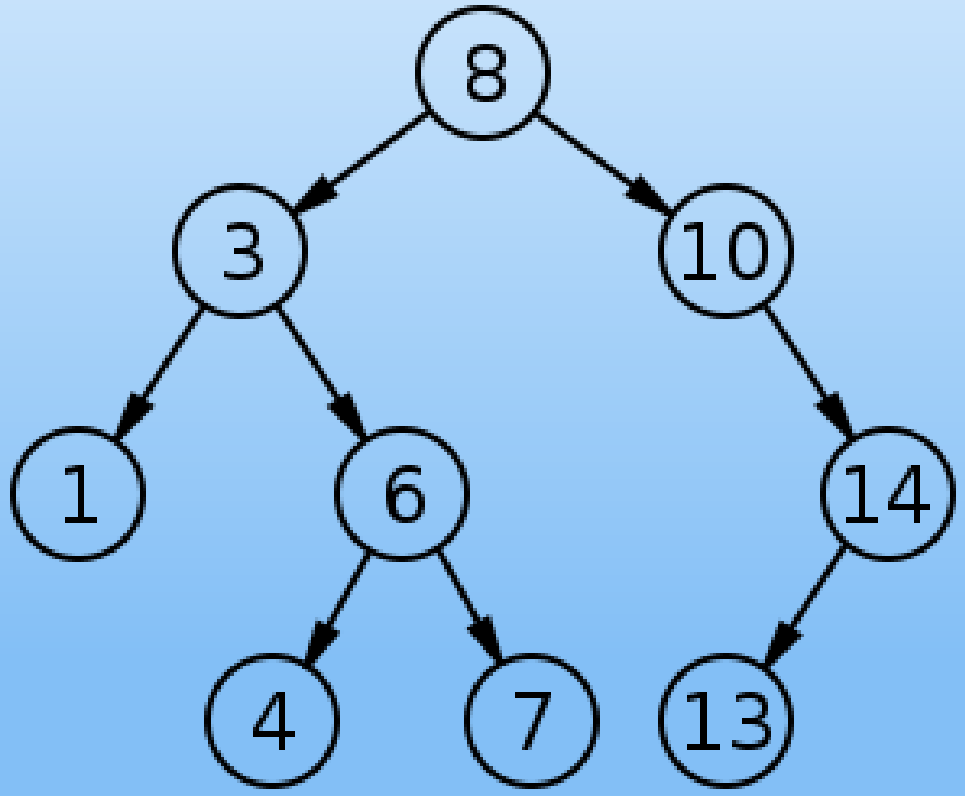
- **Python**
  - Création
    - D1 = {'toto' : 40, 'titi' : 20, 222 : 3}
    - D2 = dict([('toto',40), ('titi',20), (222, 3)]) => liste de tupple
  - Membre
    - 'toto' in D1 => Out: True => interrogation porte sur les clefs
  - Get value
    - Print D1['titi']=> Out : 20
  - Suppression
    - del D1['titi'] => {'toto': 40, 222: 3}
  - Toutes les clefs/valeurs/couples
    - D1.keys() / D1.values() / D1.items()
  - Fonction de hashage Python :
    - Tester : hash(clef)

Arbre binaire de recherche (Binary search tree) *Structure de données*

Un **arbre binaire de recherche** est un arbre binaire dans lequel chaque nœud possède une clé, telle que :

- chaque nœud du sous-arbre ***gauche*** ait une clé **inférieure** ou égale à celle du nœud considéré
- chaque nœud du sous-arbre ***droit*** possède une clé **supérieure** ou égale à celle-ci

Arbre binaire de recherche (Binary search tree) *Structure de données*

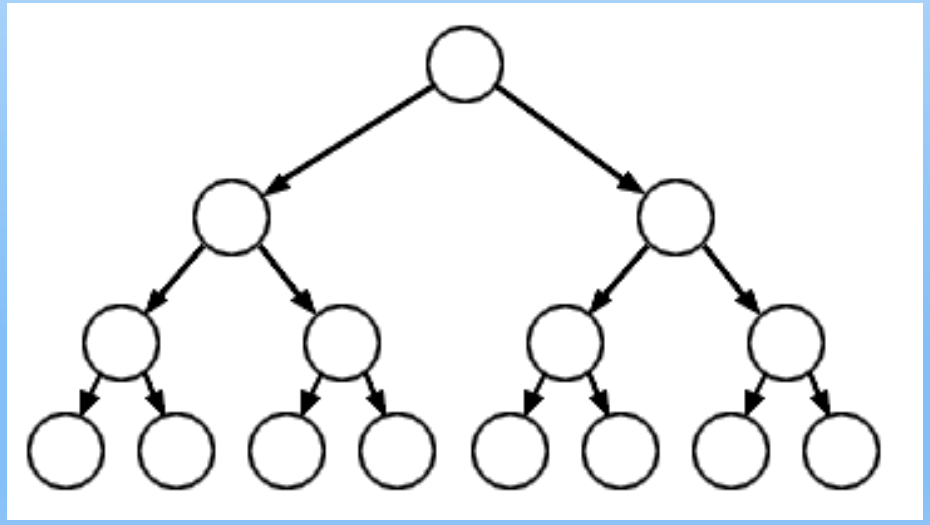


Arbre binaire de recherche (Binary search tree) *Structure de données*

La complexité des opérations dépend de la profondeur de l'arbre qui <u>en moyenne</u> est $\log_2(n+1) \cong \mathbf{\log_2(n)}$

$n = 1+2+4+8 = 15$, hauteur $\cong \log_2(16) = 4$

H = 4



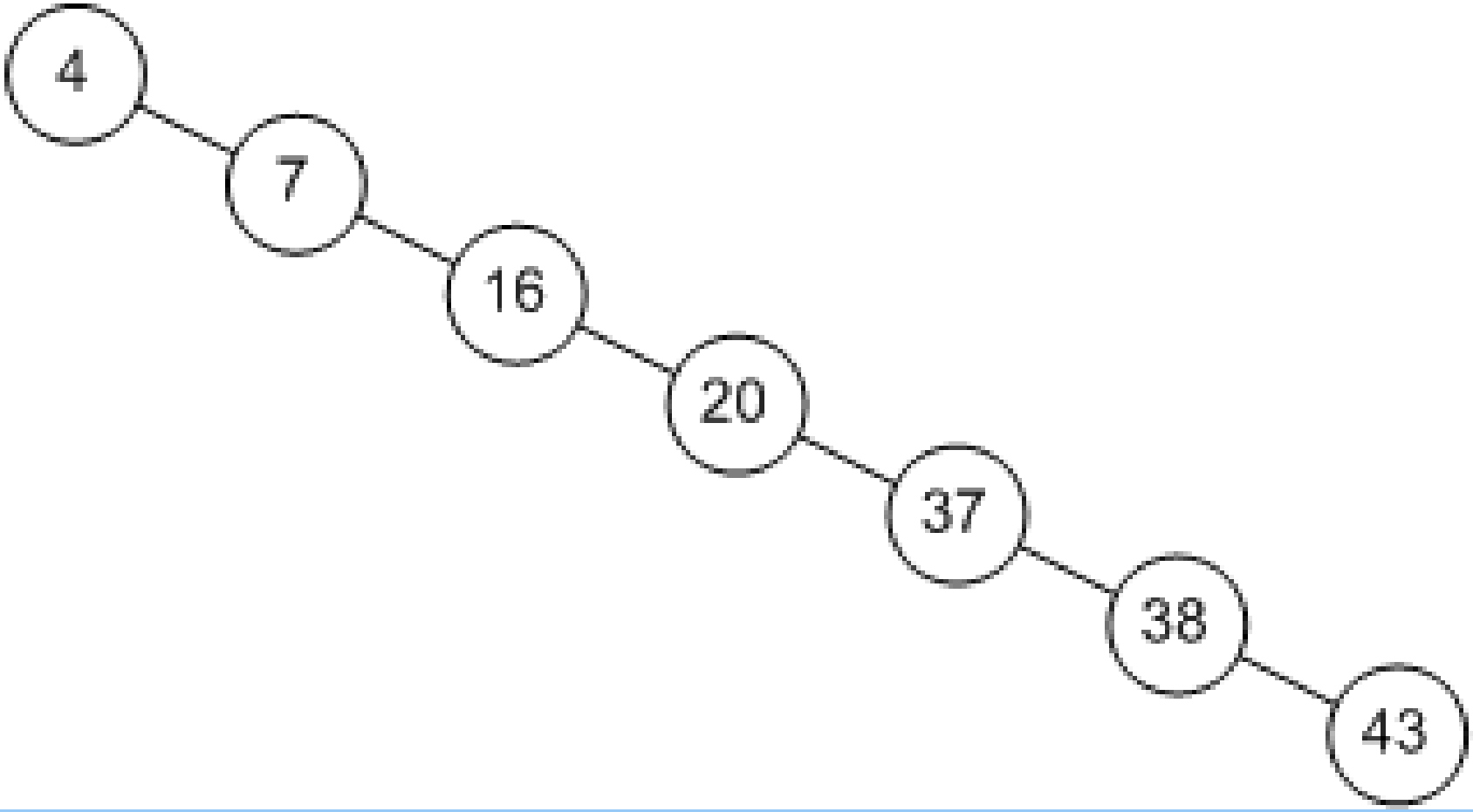
Arbre binaire de recherche (Binary search tree) *Structure de données*

Opérations	Complexité moyenne	Complexité dans le pire des cas
ADD	O(log2(n))	O(n)
REMOVE	O(log2(n))	O(n)
SEARCH	O(log2(n))	O(n)

Le problème est que lors de l'ajout de nœuds, rien ne contraint l'arbre binaire de recherche dans sa défintion à grossir de manière équilibrée.

Notamment si on ajoute par exemple successivement une suite de chiffre déjà triée, on obtient une profondeur = O(n)

Arbre binaire de recherche (Binary search tree) *Structure de données*

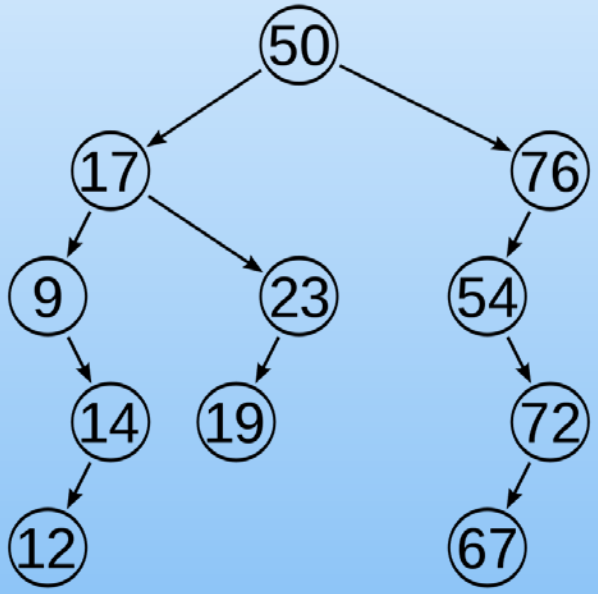


Arbre équilibré (Self-balancing binary search tree) *Structure de données*

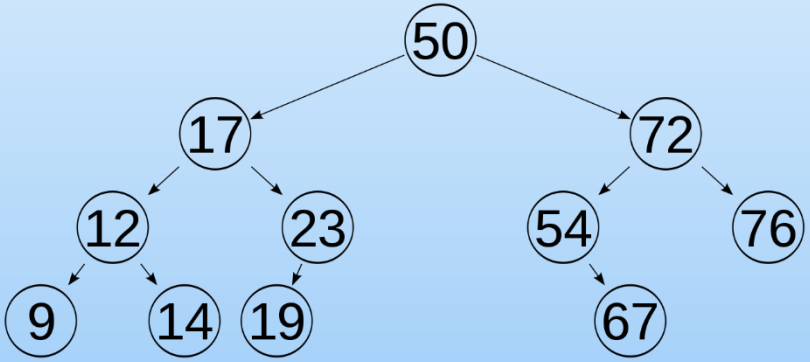
- Un **arbre équilibré**, aussi appelé **arbre à critère d'équilibre**, est un arbre qui maintient une *profondeur* équilibrée entre ses branches.

- Cela a l'avantage que le nombre de pas pour accéder à la donnée d'une clé est en moyenne minimisé, et ce nombre est égal (+/- 1) quelle que soit la clé.

Arbre équilibré (Self-balancing binary search tree) *Structure de données*



Arbre non équilibré, le parcours de la racine à un nœud quelconque utilise 3.27 nœuds en moyenne.



Le même arbre mais équilibré.
Le parcours de la racine à un nœud quelconque utilise 3 nœuds en moyenne.

Arbre équilibré rouge et noir

(Red–black tree) Structure de données

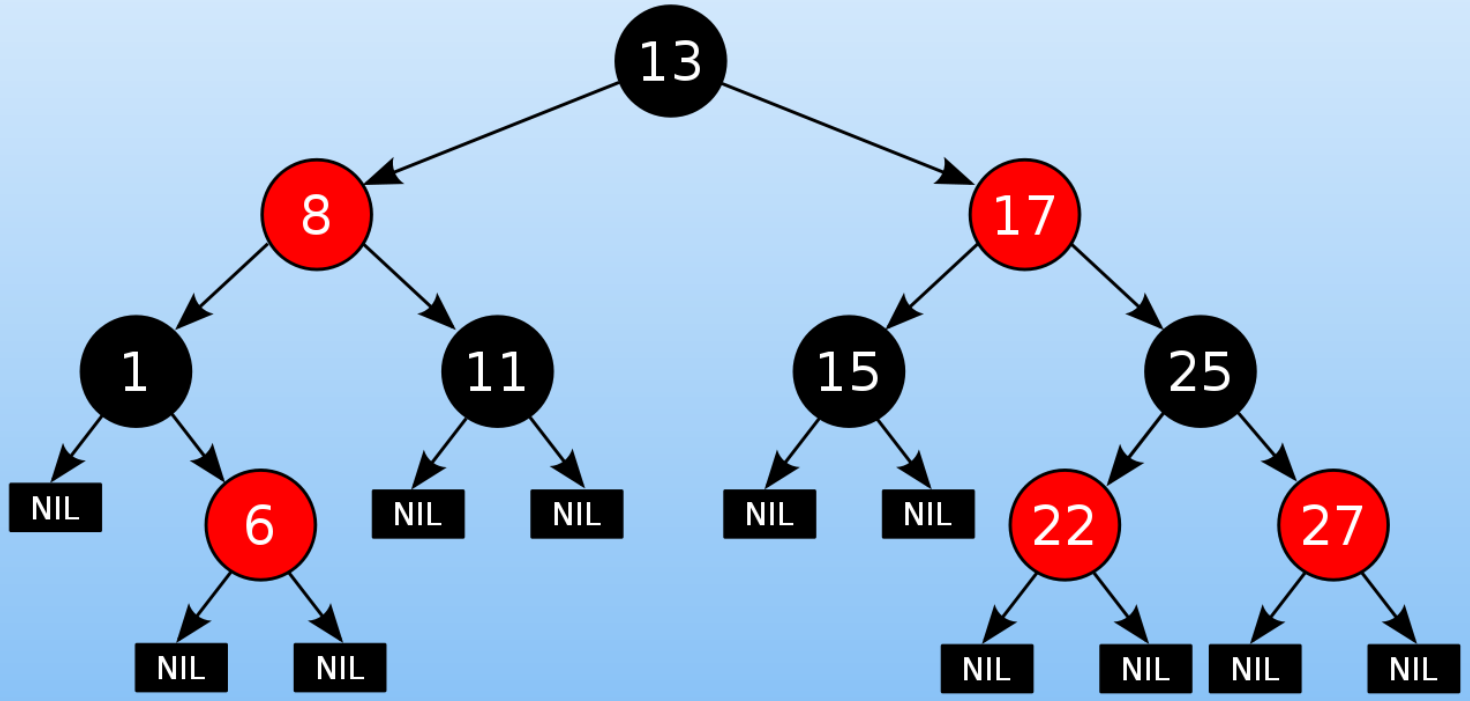
- Un **arbre équilibré rouge et noir** est un arbre binaire de recherche équilibré dont chaque nœud possède un attribut supplémentaire : sa couleur, qui est soit **rouge** soit **noire**.
  - Chaque nœud du sous-arbre gauche possède une clé inférieure ou égale à celle du nœud parent, et chaque nœud du sous-arbre droite possède une clé supérieure ou égale à celle du nœud parent (Propriété des **Arbres Binaires de Recherche**)

  +

  - Un nœud est soit rouge soit noir ;
  - La racine est noire ;
  - Toutes les feuilles sont noires ;
  - Le parent d'un nœud rouge est noir (ou dit autrement, si un nœud est rouge, alors ses deux enfants sont noirs);
  - Le chemin de chaque feuille à la racine contient le même nombre de nœuds noirs (ou dit autrement, pour chaque nœud, tous les chemins reliant le nœud à des feuilles contiennent le même nombre de nœud noirs) .

Arbre équilibré rouge et noir

(Red–black tree) Structure de données



Ces contraintes impliquent une propriété importante des arbres rouges et noirs :

- **le chemin le plus long possible d'une racine à une feuille (sa hauteur) ne peut être que deux fois plus long que le plus petit chemin possible entre une racine et une feuille**.
- On a ainsi un arbre presque équilibré.

TDA : Dictionnaire (Map)

TDA : Sans relation entre les objets

- **Complexité, comparaison**
  - Soit α (facteur de remplissage) = $\frac{\text{nombre de clefs stockés}}{\text{taille de la table}} = \frac{n}{m}$
    = nombre moyen de clefs par case

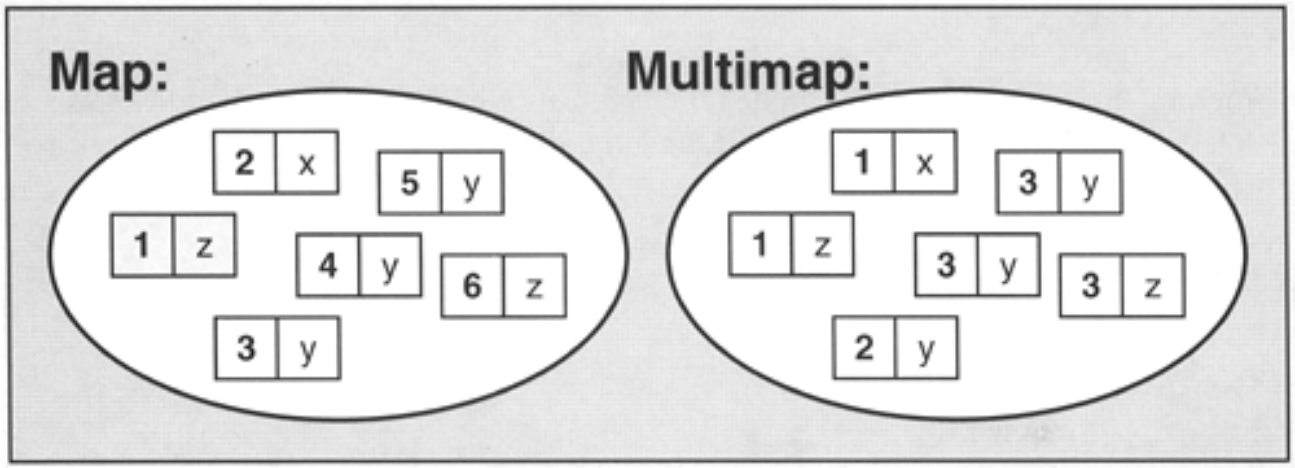
Opérations	Complexité moyenne		Complexité dans le pire des cas	
	Table Hachage	Arbre Rouge-Noir	Table Hachage	Arbre Rouge-Noir
ADD	O(1)	$O(\log_2 n)$	O(n)	$O(\log_2 n)$
REMOVE	O(1+α)	$O(\log_2 n)$	O(n)	$O(\log_2 n)$
MEMBER	O(1+α)	$O(\log_2 n)$	O(n)	$O(\log_2 n)$

Les tables de hachage ont une meilleure complexité en moyenne alors que les arbres équilibrés ont une meilleure complexité dans le pire des cas.

TDA : Multi-Dictionnaire (Multimap)

TDA : Sans relation entre les objets

- Un multi dictionnaire est un dictionnaire dans lequel chaque clef peut apparaitre plusieurs fois avec une valeur différente. Pas de contrainte d'unicité.
- Lors de la recherche d'une clef, on peut obtenir plusieurs valeurs en retour.



Bibliographie

- **Introduction à l'algorithmique**
  - Auteur(s) : Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein
  - Editeur(s) : Dunod
  - Collection : Sciences sup
- https://ece.uwaterloo.ca/~dwharder/aads/Abstract_data_types/