Structures de données et algorithmes

Joëlle Cohen

28 mars 2018

Dans ce document, la description des classes de l'API ne prétend aucunement être exhaustive. Reportez-vous à l'API en question pour connaître tous les détails de cette classe.

Chapitre I – Structures linéaires

- Introduction
- Généralités
- Les tableaux
- Les piles
- les files
- Les listes

Chapitre I – Structures linéaires

- Introduction
- Généralités
- Les tableaux
- Les piles
- les files
- Les listes

Chapitre I – Structures linéaires

- Introduction
- Généralités
- Les tableaux
- Les piles
- les files
- Les listes

Chapitre I – Structures linéaires

Chapitre I – Structures linéaires

Chapitre I – Structures linéaires

Algorithme : méthode de résolution d'un problème

Structure de données : méthodes pour stocker et manipuler des données.

Algorithme + Structure de données = programme.

Pour un problème donné, il peut y avoir plusieurs algorithmes qui peuvent utliser des structures de données différentes d'un algorithme à l'autre.
Par exemple un tri peut utiliser un tableau (tri à bulle, quicksort, ...), une liste chaînée (tri fusion), un arbre binaire de recherche, un tas (tri par tas).

Le but du cours est de savoir manipuler un ensemble de données selon un contexte imposé (par le problème ou par l'algorithme résolvant ce problème).
Pour un problème donné, on cherche à déterminer la meilleure façon de manipuler les données mises en jeu dans ce problème.
Pour cela on va

- *spécifier* $\Rightarrow$ construire un modèle abstrait du réel et décrire ce modèle en vue d'assurer l'adéquation entre ce qui est produit et ce qu'on attend,
- *utiliser* une structure de données

structure de données $\Rightarrow$ modèle qui décrit le comportement d'un ensemble d'informations muni de ses propriétés.
Une structure de données peut être vue de l'extérieur (modèle abstrait) et de l'intérieur (réalisation concrète).

Le but du cours est de savoir manipuler un ensemble de données selon un contexte imposé (par le problème ou par l'algorithme résolvant ce problème).
Pour un problème donné, on cherche à déterminer la meilleure façon de manipuler les données mises en jeu dans ce problème.
Pour cela on va

- *spécifier* = construire un modèle abstrait du réel et décrire ce modèle en vue d'assurer l'adéquation entre ce qui est produit et ce qu'on attend,
- *utiliser* une structure de données

structure de données = modèle qui décrit le comportement d'un ensemble d'informations muni de ses propriétés.
Une structure de données peut être vue de l'extérieur (modèle abstrait) et de l'intérieur (réalisation concrète).

Le but du cours est de savoir manipuler un ensemble de données selon un contexte imposé (par le problème ou par l'algorithme résolvant ce problème).
Pour un problème donné, on cherche à déterminer la meilleure façon de manipuler les données mises en jeu dans ce problème.
Pour cela on va

- *spécifier* = construire un modèle abstrait du réel et décrire ce modèle en vue d'assurer l'adéquation entre ce qui est produit et ce qu'on attend,
- *utiliser* une structure de données

structure de données = modèle qui décrit le comportement d'un ensemble d'informations muni de ses propriétés.
Une structure de données peut être vue de l'extérieur (modèle abstrait) et de l'intérieur (réalisation concrète).

Le but du cours est de savoir manipuler un ensemble de données selon un contexte imposé (par le problème ou par l'algorithme résolvant ce problème).
Pour un problème donné, on cherche à déterminer la meilleure façon de manipuler les données mises en jeu dans ce problème.
Pour cela on va

- *spécifier* = construire un modèle abstrait du réel et décrire ce modèle en vue d'assurer l'adéquation entre ce qui est produit et ce qu'on attend,
- *utiliser* une structure de données

structure de données = modèle qui décrit le comportement d'un ensemble d'informations muni de ses propriétés.
Une structure de données peut être vue de l'extérieur (modèle abstrait) et de l'intérieur (réalisation concrète).

étapes

- *spécification fonctionnelle* (ou *abstraite*) : on fait la liste des opérations (avec leurs définitions et restrictions) qui agiront sur les données. C'est le document de référence de l'utilisateur,
- *spécification opérationnelle* : elle consiste en deux étapes

Remarque : pour un même spécification fonctionnelle, il peut y avoir plusieurs plusieurs spécifications opérationnelles dans le même langage.

étapes

- *spécification fonctionnelle* (ou *abstraite*) : on fait la liste des opérations (avec leurs définitions et restrictions) qui agiront sur les données. C'est le document de référence de l'utilisateur,
- *spécification opérationnelle* : elle consiste en deux étapes
  - description logique : on organise la structure de données en utilisant des structures informatiques classiques (par exemple des tableaux),
  - représentation physique : c'est l'implémentation complète faite dans un langage de programmation fixé.

Remarque : pour un même spécification fonctionnelle, il peut y avoir plusieurs plusieurs spécifications opérationnelles dans le même langage.

étapes

- *spécification fonctionnelle* (ou *abstraite*) : on fait la liste des opérations (avec leurs définitions et restrictions) qui agiront sur les données. C'est le document de référence de l'utilisateur,
- *spécification opérationnelle* : elle consiste en deux étapes
  - description logique : on organise la structure de données en utilisant des structures informatiques classiques (par exemple des tableaux),
  - représentation physique : c'est l'implémentation complète faite dans un langage de programmation fixé.

Remarque : pour un même spécification fonctionnelle, il peut y avoir plusieurs plusieurs spécifications opérationnelles dans le même langage.

étapes

- *spécification fonctionnelle* (ou *abstraite*) : on fait la liste des opérations (avec leurs définitions et restrictions) qui agiront sur les données. C'est le document de référence de l'utilisateur,
- *spécification opérationnelle* : elle consiste en deux étapes
  - description logique : on organise la structure de données en utilisant des structures informatiques classiques (par exemple des tableaux),
  - représentation physique : c'est l'implémentation complète faite dans un langage de programmation fixé.

Remarque : pour un même spécification fonctionnelle, il peut y avoir plusieurs plusieurs spécifications opérationnelles dans le même langage.

étapes

- *spécification fonctionnelle* (ou *abstraite*) : on fait la liste des opérations (avec leurs définitions et restrictions) qui agiront sur les données. C'est le document de référence de l'utilisateur,
- *spécification opérationnelle* : elle consiste en deux étapes
  - description logique : on organise la structure de données en utilisant des structures informatiques classiques (par exemple des tableaux),
  - représentation physique : c'est l'implémentation complète faite dans un langage de programmation fixé.

Remarque : pour un même spécification fonctionnelle, il peut y avoir plusieurs plusieurs spécifications opérationnelles dans le même langage.

étapes

- *spécification fonctionnelle* (ou *abstraite*) : on fait la liste des opérations (avec leurs définitions et restrictions) qui agiront sur les données. C'est le document de référence de l'utilisateur,
- *spécification opérationnelle* : elle consiste en deux étapes
  - description logique : on organise la structure de données en utilisant des structures informatiques classiques (par exemple des tableaux),
  - représentation physique : c'est l'implémentation complète faite dans un langage de programmation fixé.

Remarque : pour un même spécification fonctionnelle, il peut y avoir plusieurs plusieurs spécifications opérationnelles dans le même langage.

pour la spécification fonctionnelle (ou abstraite) on va définir un *type de données abstrait* avec :

1. TYPE : les noms des types définis
2. UTILISE : les types abstraits déjà définis utilisés
3. OPÉRATIONS : les opérations avec leur signature[1]
4. PRÉCONDITIONS : les restrictions éventuelles d'utilisation des opérations

1. la signature d'une opération est sa description syntactique

pour la spécification fonctionnelle (ou abstraite) on va définir un *type de données abstrait* avec :

1. TYPE : les noms des types définis
2. UTILISE : les types abstraits déjà définis utilisés
3. OPÉRATIONS : les opérations avec leur signature[1]
4. PRÉCONDITIONS : les restrictions éventuelles d'utilisation des opérations

1. la signature d'une opération est sa description syntactique

pour la spécification fonctionnelle (ou abstraite) on va définir un *type de données abstrait* avec :

1. TYPE : les noms des types définis
2. UTILISE : les types abstraits déjà définis utilisés
3. OPÉRATIONS : les opérations avec leur signature [1]
4. PRÉCONDITIONS : les restrictions éventuelles d'utilisation des opérations

1. la signature d'une opération est sa description syntactique

pour la spécification fonctionnelle (ou abstraite) on va définir un *type de données abstrait* avec :

1. TYPE : les noms des types définis
2. UTILISE : les types abstraits déjà définis utilisés
3. OPÉRATIONS : les opérations avec leur signature [1]
4. PRÉCONDITIONS : les restrictions éventuelles d'utilisation des opérations

1. la signature d'une opération est sa description syntactique

Structures de données linéaires

Une structure de données est linéaire lorsque les données sont en quelque sorte les unes derrière les autres : chaque donnée a une donnée successeur et une donnée prédécesseur exception faite éventuellement des données qui sont aux extrémités.
Une fois les données rangées, la structure sera amenée à évoluer par l'*ajout* de nouvelles données ou la *suppression* de données appartenant à la structure. Selon la façon de procéder à ces modifications, on distinguera plusieurs types de structures.

Tableaux

Un tableau est une structure de données linéaire dans laquelle on a un accès direct à chaque donnée par son indice dans le tableau.
Les problèmes généralement posés sur les tableaux sont

- la recherche d'un valeur
- la recherche de la valeur la plus grande/petite
- le tri des valeurs dans l'ordre croissant
- le calcul de la valeur moyenne
- ...

Tableaux

Un tableau est une structure de données linéaire dans laquelle on a un accès direct à chaque donnée par son indice dans le tableau.
Les problèmes généralement posés sur les tableaux sont

- la recherche d'un valeur
- la recherche de la valeur la plus grande/petite
- le tri des valeurs dans l'ordre croissant
- le calcul de la valeur moyenne
- ...

Tableaux

Un tableau est une structure de données linéaire dans laquelle on a un accès direct à chaque donnée par son indice dans le tableau.
Les problèmes généralement posés sur les tableaux sont

- la recherche d'un valeur
- la recherche de la valeur la plus grande/petite
- le tri des valeurs dans l'ordre croissant
- le calcul de la valeur moyenne
- ...

Tableaux

Un tableau est une structure de données linéaire dans laquelle on a un accès direct à chaque donnée par son indice dans le tableau.
Les problèmes généralement posés sur les tableaux sont

- la recherche d'un valeur
- la recherche de la valeur la plus grande/petite
- le tri des valeurs dans l'ordre croissant
- le calcul de la valeur moyenne
- ...

Tableaux

Un tableau est une structure de données linéaire dans laquelle on a un accès direct à chaque donnée par son indice dans le tableau.
Les problèmes généralement posés sur les tableaux sont

- la recherche d'un valeur
- la recherche de la valeur la plus grande/petite
- le tri des valeurs dans l'ordre croissant
- le calcul de la valeur moyenne
- ...

2-somme

Problème : étant données n valeurs entières, trouver tous les couples de valeurs dont la somme est nulle.

2-somme

Problème : étant données n valeurs entières, trouver tous les couples de valeurs dont la somme est nulle.

algorithme : tester toutes les sommes, afficher les sommes nulles

2-somme

Problème : étant données n valeurs entières, trouver tous les couples de valeurs dont la somme est nulle.

données : les n valeurs sont stockées dans un tableau d'entiers t.
algorithme : pour chaque indice k du tableau, calculer toutes les sommes t(k)+t(i) pour i variant dans les indices du tableau, les afficher si elles sont nulles

2-somme

Problème : étant données n valeurs entières, trouver tous les couples de valeurs dont la somme est nulle.

données : les n valeurs sont stockées dans un tableau d'entiers t.
algorithme : pour chaque indice k du tableau, pour chaque indice i du tableau supérieur à k, calculer t(k)+t(i), l'afficher si nulle

programme

```
public class DeuxSomme{
public static void afficherDeuxSomme(int[] t) {
    for (int k = 0; k < t.length; k++) {
        for (int i = k; i < t.length; i++) {
            if (t[k] + t[i] == 0) {
                System.out.println(t[k]+ "  "+ t[i]);
            }}}}
public static void main(String[] args) {
. . .
}
```

temps d'exécution

Cette procédure exécute $\frac{n(n-1)}{2}$ itérations donc la complexité est en $O(n^2)$.
Peut-on tester le temps d'exécution ? oui
Peut-on faire mieux ? oui

temps d'exécution

Cette procédure exécute $\frac{n(n-1)}{2}$ itérations donc la complexité est en $O(n^2)$.
Peut-on tester le temps d'exécution ? oui
Peut-on faire mieux ? oui

temps d'exécution

Cette procédure exécute $\frac{n(n-1)}{2}$ itérations donc la complexité est en $O(n^2)$.
Peut-on tester le temps d'exécution ? oui
Peut-on faire mieux ? oui

temps d'exécution

Cette procédure exécute $\frac{n(n-1)}{2}$ itérations donc la complexité est en $O(n^2)$.
Peut-on tester le temps d'exécution ? oui
Peut-on faire mieux ? oui

mesurer le temps d'exécution

```
public class Chronometre {
private final long debut = System.currentTimeMillis();
public double tempsEcoule() {
long maintenant = System.currentTimeMillis();
return (maintenant - debut) / 1000.0;}}
```

La méthode `currentTimeMillis()` de la classe `System` donne en milliseconde le temps écoulé depuis le 1er Janvier 1970 minuit jusqu'à l'invocation de la méthode.
Une instance de la classe `Chronometre` a donc l'attribut `debut` initialisé à cette durée au moment de l'appel au constructeur de la classe.
La méthode `tempsEcoule` mesure en seconde le temps écoulé entre l'appel au constructeur de la classe et l'invocation de la méthode.

mesurer le temps d'exécution

```
public static void main(String[] args) {
int[] t ;
// initialiser et remplir t
// utiliser la classe Random du paquetage java.util
Chronometre c = new Chronometre();
afficherDeuxSomme(t);
System.out.println
("temps d'exécution " + c.tempsEcoule()+ " secondes");
}
```

exercice 1 : : faire des tests sur des tableaux de dimension 10, 100, 1000, 2000, 4000, 8000.

Faire mieux ?

Supposons le tableau classé par ordre croissant.

0	1	2	3	4	5	6	7	8	9	10	11
-7	-5	-4	-2	-1	2	3	6	7	8	11	15

Faire mieux ?

Supposons le tableau classé par ordre croissant.

0	1	2	3	4	5	6	7	8	9	10	11
-7	-5	-4	-2	-1	2	3	6	7	8	11	15

Faire mieux ?

0	1	2	3	4	5	6	7	8	9	10	11
-7	-5	-4	-2	-1	2	3	6	7	8	11	15

Faire mieux ?

0	1	2	3	4	5	6	7	8	9	10	11
-7	-5	-4	-2	-1	2	3	6	7	8	11	15

Faire mieux ?

0	1	2	3	4	5	6	7	8	9	10	11
-7	-5	-4	-2	-1	2	3	6	7	8	11	15

Faire mieux ?

0	1	2	3	4	5	6	7	8	9	10	11
-7	-5	-4	-2	-1	2	3	6	7	8	11	15

-7 et 7

Faire mieux ?

0	1	2	3	4	5	6	7	8	9	10	11
-7	-5	-4	-2	-1	2	3	6	7	8	11	15

-7 et 7

Faire mieux ?

0	1	2	3	4	5	6	7	8	9	10	11
-7	-5	-4	-2	-1	2	3	6	7	8	11	15

-7 et 7

Faire mieux ?

0	1	2	3	4	5	6	7	8	9	10	11
-7	-5	-4	-2	-1	2	3	6	7	8	11	15

-7 et 7

Faire mieux ?

0	1	2	3	4	5	6	7	8	9	10	11
-7	-5	-4	-2	-1	2	3	6	7	8	11	15

-7 et 7

Faire mieux ?

0	1	2	3	4	5	6	7	8	9	10	11
-7	-5	-4	-2	-1	2	3	6	7	8	11	15

-7 et 7

Faire mieux ?

0	1	2	3	4	5	6	7	8	9	10	11
-7	-5	-4	-2	-1	2	3	6	7	8	11	15

-7 et 7
-2 et 2

Faire mieux ?

0	1	2	3	4	5	6	7	8	9	10	11
-7	-5	-4	-2	-1	2	3	6	7	8	11	15

-7 et 7
-2 et 2
arrêt car même signe

exercice 2 : implémenter cette méthode
`afficherDeuxSommeRapide`

Tri par insertion

C'est le principe du joueur de carte :
on trie au fur et à mesure que l'on découvre une nouvelle valeur en l'insérant parmi celles déjà triées.
Par exemple : on a déjà

i	1	2	3	4	5	6	7	8	9	10
$t(i)$	-75	-54	15	34	56	74				

Tri par insertion

C'est le principe du joueur de carte :
on trie au fur et à mesure que l'on découvre une nouvelle valeur en l'insérant parmi celles déjà triées.
Par exemple : on a déjà

i	1	2	3	4	5	6	7	8	9	10
$t(i)$	-75	-54	15	34	56	74				

et on donne la valeur 25 à insérer

i	1	2	3	4	5	6	7	8	9	10
$t(i)$	-75	-54	15	34	56	74				

Tri par insertion

C'est le principe du joueur de carte :
on trie au fur et à mesure que l'on découvre une nouvelle valeur en l'insérant parmi celles déjà triées.
Par exemple : on a déjà

i	1	2	3	4	5	6	7	8	9	10
$t(i)$	-75	-54	15	34	56	74				

On décale les trois valeurs supérieures à 25 d'un rang vers la droite puis on place 25

i	1	2	3	4	5	6	7	8	9	10
$t(i)$	-75	-54	15	25	34	56	74			

En pratique, le tableau est déjà rempli. Donc on commence par insérer la deuxième valeur dans la partie gauche du tableau réduite à la première valeur, puis la troisième valeur dans la partie gauche du tableau composée des deux premières valeurs déjà triées et ainsi de suite jusqu'à insérer la dernière valeur.

L'insertion peut se réaliser de façons

En pratique, le tableau est déjà rempli. Donc on commence par insérer la deuxième valeur dans la partie gauche du tableau réduite à la première valeur, puis la troisième valeur dans la partie gauche du tableau composée des deux premières valeurs déjà triées et ainsi de suite jusqu'à insérer la dernière valeur.
L'insertion peut se réaliser de façons

En pratique, le tableau est déjà rempli. Donc on commence par insérer la deuxième valeur dans la partie gauche du tableau réduite à la première valeur, puis la troisième valeur dans la partie gauche du tableau composée des deux premières valeurs déjà triées et ainsi de suite jusqu'à insérer la dernière valeur.
L'insertion peut se réaliser de façons

- on décale d'un rang vers la droite toutes les valeurs de rang inférieur qui sont supérieures à la valeur à insérer
- on recule la valeur à insérer par échange de la valeur avec sa voisine de gauche jusqu'à ce sa voisine lui soit inférieure

En pratique, le tableau est déjà rempli. Donc on commence par insérer la deuxième valeur dans la partie gauche du tableau réduite à la première valeur, puis la troisième valeur dans la partie gauche du tableau composée des deux premières valeurs déjà triées et ainsi de suite jusqu'à insérer la dernière valeur.
L'insertion peut se réaliser de façons

- on décale d'un rang vers la droite toutes les valeurs de rang inférieur qui sont supérieures à la valeur à insérer
- on recule la valeur à insérer par échange de la valeur avec sa voisine de gauche jusqu'à ce sa voisine lui soit inférieure

Complexité des tris de tableau

Les tris simples (insertion, à bulle, par sélection) ont une complexité en $O(n^2)$.
Donc la méthode `DeuxSommeRapide` n'apporte rien si on ne peut pas améliorer les tris.
Les tris rapide, fusion (voir *listes*) ou par tas (voir *file de priorité*) ont une complexité en moyenne de $O(n \ln(n))$.

Tri rapide

Le tri rapide (Quicksort) est basé sur une méthode `pivoter` qui partage le tableau en deux parties (non nécessairement équilibrées).
A la frontière des deux parties se trouve une valeur de référence, le *pivot*, qui aura servi à faire ce partage.
La méthode `pivoter` est alors rappelée sur les deux parties de façon récursive (avec un pivot propre à chaque partie).

exercice 3 : Ecrire les méthodes `pivoter` puis `triRapide`

exercice 4 : tester le temps d'exécution de `tri rapide` + `afficherDeuxSommeRapide`.

Autres tris

Tous les tris vus précédemment sont basés sur des comparaisons des valeurs du tableau deux à deux. Il existe d'autres méthodes de tris basées sur d'autres principes :

- tri comptage (counting sort) : cette méthode nécessite que l'on connaisse les bornes des valeurs à trier, et que l'intervalle de ces valeurs ne soit pas trop « grand »
- tri radix (radix sort) : cette méthode est basée sur l'écriture en base 2 (voire 10) des nombres
- tri par paquets (bucket sort) : cette méthode nécessite que les valeurs à trier soient répaties uniformément sur une intervalle connu

Tri comptage

On suppose que les valeurs à trier appartiennent à l'intervalle $[0, N]$.
On utilise alors un tableau `tIndex` indexé de 0 à N.
Principe :

- on parcourt le tableau à trier `t` ; si `t`$[i]$ vaut k alors on incrémente `tIndex`$[k]$.
- on parcourt le tableau `tIndex` ; on place la valeur k `tIndex`$[k]$ fois successivement dans `t`

Exemple :

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	4	7	4	5	4	5	7	1	2	1	7

k	0	1	2	3	4	5	6	7
$tIndex(k)$	0	0	0	0	0	0	0	0

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	4	7	4	5	4	5	7	1	2	1	7

k	0	1	2	3	4	5	6	7
$tIndex(k)$	0	0	0	0	1	0	0	0

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	4	7	4	5	4	5	7	1	2	1	7

k	0	1	2	3	4	5	6	7
$tIndex(k)$	0	0	0	0	1	0	0	1

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	4	7	4	5	4	5	7	1	2	1	7

k	0	1	2	3	4	5	6	7
$tIndex(k)$	0	0	0	0	2	0	0	1

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	4	7	4	5	4	5	7	1	2	1	7

k	0	1	2	3	4	5	6	7
$tIndex(k)$	0	0	0	0	2	1	0	1

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	4	7	4	5	4	5	7	1	2	1	7

k	0	1	2	3	4	5	6	7
$tIndex(k)$	0	0	0	0	3	1	0	1

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	4	7	4	5	4	5	7	1	2	1	7

k	0	1	2	3	4	5	6	7
$tIndex(k)$	0	0	0	0	3	2	0	1

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	4	7	4	5	4	5	7	1	2	1	7

k	0	1	2	3	4	5	6	7
$tIndex(k)$	0	0	0	0	3	2	0	2

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	4	7	4	5	4	5	7	1	2	1	7

k	0	1	2	3	4	5	6	7
$tIndex(k)$	0	1	0	0	3	2	0	2

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	4	7	4	5	4	5	7	1	2	1	7

k	0	1	2	3	4	5	6	7
$tIndex(k)$	0	1	1	0	3	2	0	2

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	4	7	4	5	4	5	7	1	2	1	7

k	0	1	2	3	4	5	6	7
$tIndex(k)$	0	2	1	0	3	2	0	2

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	4	7	4	5	4	5	7	1	2	1	7

k	0	1	2	3	4	5	6	7
$tIndex(k)$	0	2	1	0	3	2	0	3

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	1	1	4	5	4	5	7	1	2	1	7

k	0	1	2	3	4	5	6	7
$tIndex(k)$	0	2	1	0	3	2	0	3

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	1	1	2	5	4	5	7	1	2	1	7

k	0	1	2	3	4	5	6	7
$tIndex(k)$	0	2	1	0	3	2	0	3

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	1	1	2	4	4	4	7	1	2	1	7

k	0	1	2	3	4	5	6	7
$tIndex(k)$	0	2	1	0	3	2	0	3

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	1	1	2	4	4	4	5	5	2	1	7

k	0	1	2	3	4	5	6	7
$tIndex(k)$	0	2	1	0	3	2	0	3

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	1	1	2	4	4	4	5	5	7	7	7

k	0	1	2	3	4	5	6	7
$tIndex(k)$	0	2	1	0	3	2	0	3

Tri radix

Le principe présenté s'adapte à d'autres bases ou même à l'alphabet.
Principe : on classe les nombres par ordre croissant selon les chiffres qui le composent depuis le moins significatif (unité) jusqu'au plus significatif tout en conservant l'ordre obtenu à l'itération précédente pour les nombres ayant le même chiffre significatif considéré.
Exemple :

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	473	70	24	651	42	55	709	19	2	192	97

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	70	651	42	2	192	473	24	55	97	19	709

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	02	709	19	24	42	651	55	70	473	192	97

i	0	1	2	3	4	5	6	7	8	9	10
$t(i)$	002	019	024	042	055	070	097	192	473	651	709

En pratique ce tri nécessite 10 tableaux de même longueur que le tableau d'origine `t`, chacun est prévu pour recevoir les valeurs dont le chiffre significatif est respectivement 0, ... 9. A chaque itération, la tableau `t` est lu par ordre croissant d'indice et on remplit ces 10 tableaux, puis on remet les valeurs dans le tableau `t` depuis les tableaux intermédiaires en partant du tableau `t0` jusqu'au tableau `t9`. La complexité de ce tri est en O(n) mais nécessite de la mémoire auxiliaire.

Tri par paquets

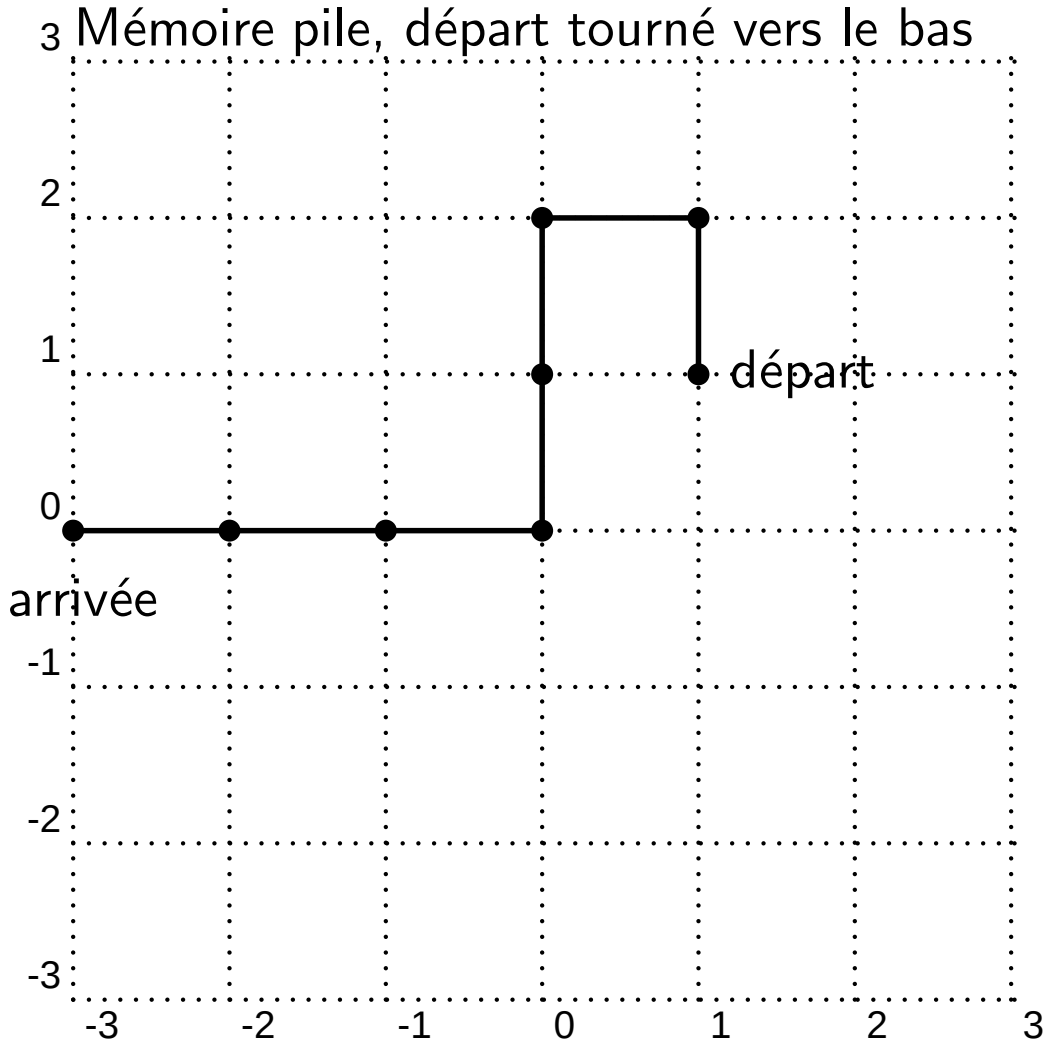
Ce tri fonctionne pour des valeurs réelles contenues dans un intervalle connu $[a, b[$. Soit N le nombre de ces valeurs.
On partitionne l'intervalle en N sous-intervalles de la forme suivante $I_j = [a + j\frac{b-a}{N}, a + (j+1)\frac{b-a}{N}[$ pour $0 \leq j < N$.
On crée N listes L_j pour $0 \leq j < N$.
Principe :

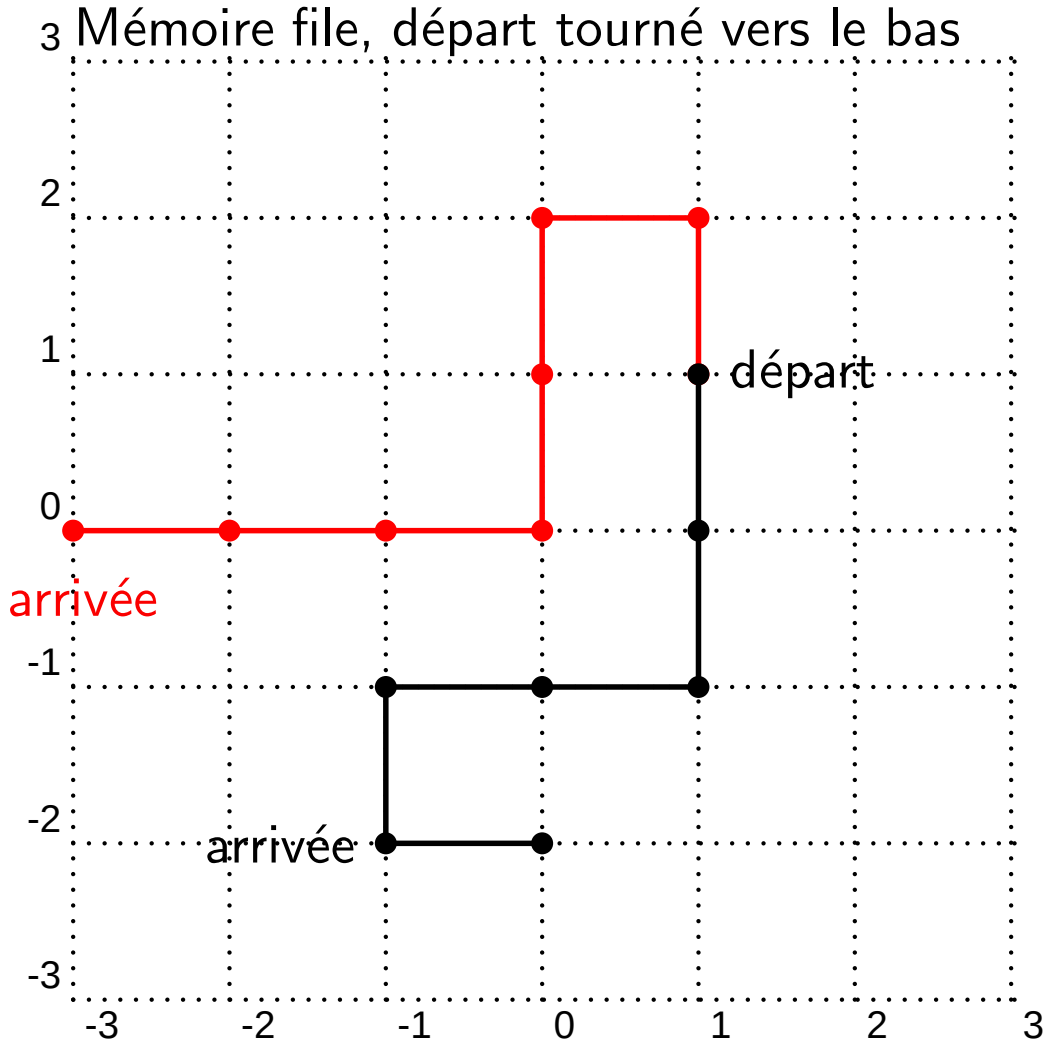
- Pour chaque valeur de t
  - calculer l'intervalle I_j contenant cette valeur
  - placer cette valeur dans la liste L_j
- Trier les N listes $L_0, \ldots L_{N-1}$.
- Copier dans t depuis l'indice 0 L_0 puis L_1 ... puis L_{N-1}.

Ce tri est performant en O(n) si les valeurs sont uniformément réparties ; en effet puisqu'il y a autant d'intervalles que de valeurs, une distribution uniforme garantit des listes de longueur 1 qui sont dès lors déjà triées.
Le pire des cas est une distribution qui placerait toutes les valeurs dans le même intervalle de la partition.

Autres structures linéaires : exemple

Un robot piloté à distance reçoit différents types d'instructions pour se déplacer : en avant, en arrière (demi-tour), à droite, à gauche. Il est de plus muni d'une mémoire « linéaire » qui enregistre la suite d'instructions et ne démarre qu'une fois toutes les instructions reçues.
Par exemple, en avant, en avant, à droite, en avant, à gauche, à gauche, en arrière.
Le choix du déplacement dépendra de la façon dont est gérée la mémoire : il choisit de lire tout d'abord soit la première instruction entrée soit la dernière.
Dans le premier cas sa mémoire fonctionne comme une file.
Dans ce dernier cas sa mémoire fonctionne comme une pile.





Les piles

Une pile est donc une structure de données linéaire dans laquelle ajout et suppression se font au même bout de la structure : son *sommet*.
On dit qu'on *empile* lorsque l'on ajoute un nouvel élément à la pile et cet ajout se fait au sommet de la pile.
On dit qu'on *dépile* lorsque l'on supprime un élément de la pile et puisque la suppression se fait aussi au sommet de la pile, l'élément supprimé est nécessairement le dernier élément à avoir été empilé.
Une pile est aussi désignée par l'acronyme anglais LIFO (**L**ast **I**n **F**irst **O**ut).
Une pile permet par exemple à un navigateur (Firefox, IE ...) de gérer le bouton « reculer d'une page » ou pour un logiciel de gérer la fonctionnalité « undo ».

spécification fonctionnelle

Elle est donc définie par le type de données abstrait PILE suivant :
type PILE
utilise type E, **type** BOOLÉEN
signature

- *initilialiser* : $\rightarrow$ PILE
- *empiler* : (PILE,E)$\rightarrow$ PILE
- *dépiler* : PILE $\rightarrow$ PILE
- *sommet* : PILE $\rightarrow$ E
- *vide* : PILE $\rightarrow$ BOOLÉEN

préconditions

- *dépiler*(p) est défini *si et seulement sivide*(p)= Faux
- *sommet*(p) est défini *si et seulement sivide*(p)= Faux

axiomes

- *vide*(*initilialiser*)= Vrai
- *vide*(*empiler*(p,E))=Faux
- *dépiler*(*empiler*(p,E))=p
- *sommet*(*empiler*(p,E))= E

remarques :

- les préconditions précisent la restriction du domaine de définition des deux opérations *dépiler* et *sommet*
- les axiomes assurent le fonctionnement LIFO.

Interface Pile

Pour traduire en partie les spécifications fonctionnelles on va utliser la notion d'interface Java.

```
/**
* Interface pour le type pile
*@author moi
*@version 0.1
*/
public Interface Pile{
boolean vide();
void empiler(int n);
void depiler();
int sommet();
}
```

remarque ? ; on a créé une interface spéciale pour les piles d'entiers alors que des piles de flottants ou autre objets se comportent de la même façon.
Solutions ? ?

Interface Pile

Pour traduire en partie les spécifications fonctionnelles on va utliser la notion d'interface Java.

```java
/**
* Interface pour le type pile
*@author moi
*@version 0.1
*/
public Interface Pile{
boolean vide();
void empiler(int n);
void depiler();
int sommet();
}
```

remarque ? : on a créé une interface spéciale pour les piles d'entiers alors que des piles de flottants ou autre objets se comportent de la même façon.
Solutions ? ?

Interface Pile

Pour traduire en partie les spécifications fonctionnelles on va utliser la notion d'interface Java.

```
/**
* Interface pour le type pile
*@author moi
*@version 0.1
*/
public Interface Pile{
boolean vide();
void empiler(int n);
void depiler();
int sommet();
}
```

remarque ? : on a créé une interface spéciale pour les piles d'entiers alors que des piles de flottants ou autre objets se comportent de la même façon.
Solutions ? ?

Interface Pile

Avec le type `Object`

```
/**
* Interface pour le type pile
*@author moi
*@version 0.2
*/
public interface Pile{
boolean vide();
void empiler(Object o);
void depiler();
Object sommet();
}
```

inconvénient : utilisation importante du transtypage.
avantage : on peut empiler des objets instances de classes différentes
mais après on ne saura pas les manipuler finement.

Interface Pile

Avec le type `Object`

```
/**
* Interface pour le type pile
*@author moi
*@version 0.2
*/
public interface Pile{
boolean vide();
void empiler(Object o);
void depiler();
Object sommet();
}
```

inconvénient : utilisation importante du transtypage.
avantage : on peut empiler des objets instances de classes différentes mais après on ne saura pas les manipuler finement.

Interface Pile générique

La généricité est une notion de *polymorphisme paramétrique* qui permet de définir ici des piles contenant des objets de type (uniforme) mais arbitraire.

```
/**
* Interface pour le type pile générique
*@author moi
*@version 1.0
*/
public Interface Pile<T>{
/** T est un paramètre de type de cette interface */
/** T représente le type des éléments de la pile */
boolean vide();
void empiler(T o);
void depiler();
T sommet();
}
```

et maintenant l'implémentation, des idées ? ? ?

Interface Pile générique

La généricité est une notion de *polymorphisme paramétrique* qui permet de définir ici des piles contenant des objets de type (uniforme) mais arbitraire.

```
/**
* Interface pour le type pile générique
*@author moi
*@version 1.0
*/
public Interface Pile<T>{
/** T est un paramètre de type de cette interface */
/** T représente le type des éléments de la pile */
boolean vide();
void empiler(T o);
void depiler();
T sommet();
}
```

et maintenant l'implémentation, des idées ? ? ?

avec un tableau dynamique

```
import java.util.ArrayList;
public class PileArrayList<T> implements Pile<T>{
private int top=-1;
private ArrayList<T> tabPile;
// constructeur
public PileArrayList(){ tabPile= new ArrayList<T>();}
// implémentation de l'interface
boolean vide(){return this.top==-1;}
void empiler(T o){tabPile.add(o); top ++;}
void depiler(){top --;}
T sommet(){return tabPile.get(top); }
}
```

utilisation

```
public class EssaiPileArrayList {
public static void main(String[] args) {
 Pile<Integer> p = new PileArrayList<Integer>();
   for (int i=1; i<=10;i++)  p.empiler(i);
   p.depiler();
   p.depiler();
   System.out.println
   ("le sommet de la pile p est " + p.sommet());

 Pile<String> p2 = new PileArrayList<String>();
   for (int i=1; i<=10;i++)
     p2.empiler("toto " + i);
   p2.depiler();
   System.out.println
   ("le sommet de la pile p2 est " + p2.sommet());
  }
}
```

```
le sommet de la pile p est 8
le sommet de la pile p2 est toto 9
```

utilisation

```
public class EssaiPileArrayList {
public static void main(String[] args) {
 Pile<Integer> p = new PileArrayList<Integer>();
   for (int i=1; i<=10;i++)  p.empiler(i);
   p.depiler();
   p.depiler();
   System.out.println
   ("le sommet de la pile p est " + p.sommet());

 Pile<String> p2 = new PileArrayList<String>();
   for (int i=1; i<=10;i++)
     p2.empiler("toto " + i);
   p2.depiler();
   System.out.println
   ("le sommet de la pile p2 est " + p2.sommet());
  }
}
```

```
le sommet de la pile p est 8
le sommet de la pile p2 est toto 9
```

autre implémentation

```
public class PileAuto<T> implements Pile<T>{
//classe interne privée
    private class Cell{
    private T val;
    private Cell suiv;
    Cell(T v, Cell c){val=v; suiv=c;}
    T getVal(){return val;}
    Cell getSuiv(){return suiv;}
  }
//1 unique attribut
   private Cell top;
// les méthodes de l'interface Pile
   public boolean vide(){
     return top == null;
   }
   public void empiler(T o) {
     top=new Cell(o,top);
   }
   public void depiler(){
     top=top.getSuiv();
   }
   public T sommet(){
     return this.top.getVal();
   }}
```

```
public class EssaiPileArrayListEtPileAuto {
public static void main(String[] args) {
 Pile<Integer> p = new PileArrayList<Integer>();
   for (int i=1; i<=10;i++)  p.empiler(i);
   p.depiler();
   p.depiler();
   System.out.println
   ("le sommet de la pile p est " + p.sommet());

 Pile<Float> p3= new PileAuto<Float>();
  for (int i=10; i>=1;i--)  p3.empiler(i+20.0f);
  p3.depiler();
  System.out.println
  ("le sommet de la pile p3 est " + p3.sommet());
  }
}
```

```
le sommet de la pile p est 8
le sommet de la pile p3 est 22.0
```

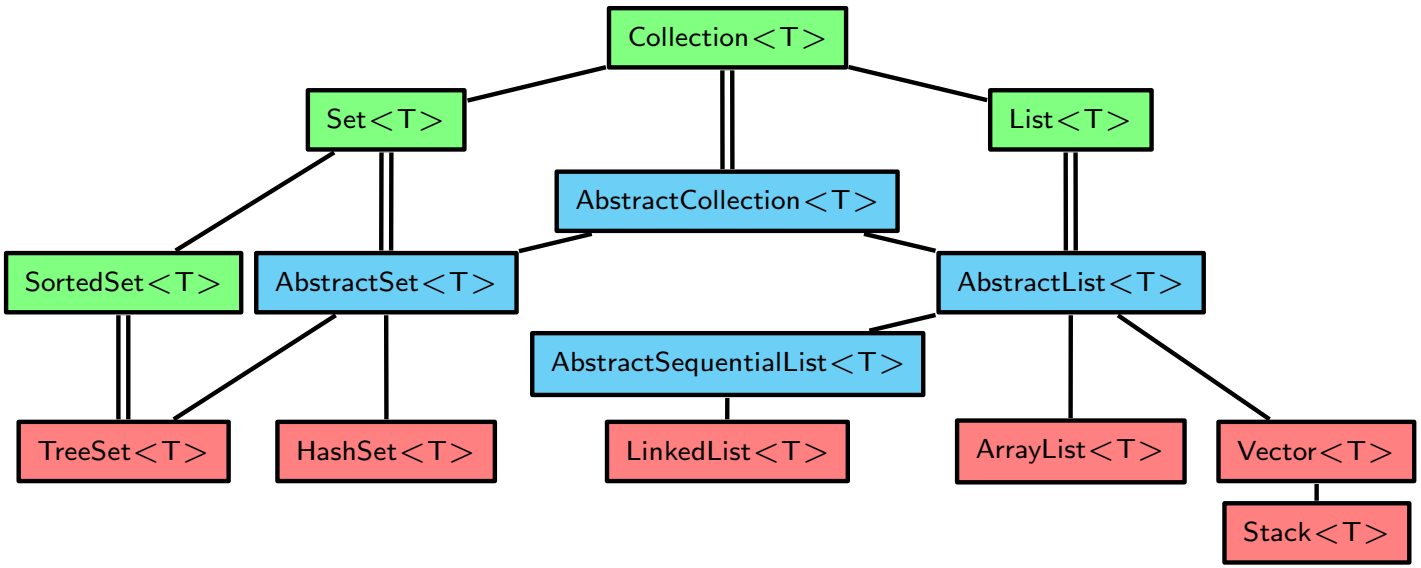
```java
public class EssaiPileArrayListEtPileAuto {
public static void main(String[] args) {
 Pile<Integer> p = new PileArrayList<Integer>();
   for (int i=1; i<=10;i++)  p.empiler(i);
   p.depiler();
   p.depiler();
   System.out.println
   ("le sommet de la pile p est " + p.sommet());

 Pile<Float> p3= new PileAuto<Float>();
  for (int i=10; i>=1;i--)  p3.empiler(i+20.0f);
  p3.depiler();
  System.out.println
  ("le sommet de la pile p3 est " + p3.sommet());
  }
}
```

```
le sommet de la pile p est 8
le sommet de la pile p3 est 22.0
```

une classe toute prête

Dans le paquetage `java.util`



la classe Stack

Il existe aussi la classe `Stack` : c'est une sous-classe de la classe `Vector` qui implémente entre autres les interfaces `List<T>` et `Collection<T>`. Elle étend la classe `Vector` avec 5 opérations qui permettent de traiter un objet `Vector` comme une pile.

- `public boolean empty()` ⇜⇝ `public boolean vide()`
- `public T push(T item)` : empile item et renvoie item
- `public T pop()` : dépile le sommet et renvoie le sommet
- `public T peek()` ⇜⇝ `public T sommet()`
- `public int search(Object o)` : renvoie la position de l'objet o dans la pile avec la convention suivante
  - si o n'est pas dans la pile renvoie -1
  - sinon renvoie le numéro d'ordre depuis le sommet jusqu'à la première occurrence de o en "descendant" (le sommet ayant le numéro 1)

Exercices

exercice 5 : Ecrire l'algorithme et une méthode qui inverse une pile en utilisant uniquement les fonctionnalités des piles

exercice 6 : Ecrire l'algorithme et une méthode d'instance qui renvoie une pile d'entiers dans laquelle les valeurs positives de l'objet sont au-dessus des valeurs négatives de l'objet, dans le même ordre relatif.

exercice 7 : Réécrire la méthode `equals()` pour une égalité en profondeur.

exercice 8 : Réécrire la méthode `clone()` pour un clonage en profondeur.

exercice 9 : Utiliser les piles pour résoudre le problème des tours Hanoï.

Les files

Une file est donc une structure de données linéaire dans laquelle l'ajout se fait à une extrêmité - la queue - et la suppression se fait à l'autre extrêmité de la structure - la tête.
On dit qu'on *enfile* lorsque l'on ajoute un nouvel élément à la file.
On dit qu'on *défile* lorsque l'on supprime un élément de la file et puisque la suppression se fait en tête de la file, l'élément supprimé est nécessairement le premier élément à avoir été enfilé. Une file est aussi désignée par l'acronyme anglais FIFO (**F**irst **I**n **F**irst **O**ut).
Une file permet par exemple à un serveur de gérer les requêtes qu'il reçoit.

spécification fonctionnelle

Elle est donc définie par le type de données abstrait FILE suivant :

type FILE

utilise TYPE E, TYPE BOOLÉEN

signature

- *initilialiser* : $\rightarrow$ FILE
- *enfiler* : (FILE,E)$\rightarrow$ FILE
- *défiler* : FILE $\rightarrow$ FILE
- *premier* : FILE $\rightarrow$ E
- *vide* : FILE $\rightarrow$ BOOLÉEN

préconditions

- *défiler*(f) est défini *si et seulement si vide*(f)= Faux
- *premier*(f) est défini *si et seulement si vide*(f)= Faux

axiomes

- *vide*(*initilialiser*)= Vrai
- *vide*(*enfiler(f,E)*)=Faux
- $premier(enfiler(f,E)) = \begin{cases} E & \text{si } vide(\text{f}) \\ premier(f) & \text{sinon} \end{cases}$
- $défiler(enfiler(f,E)) = \begin{cases} initilialiser(f) & \text{si } vide(\text{f}) \\ enfiler(défiler(f), E) & \text{sinon} \end{cases}$

Interface File

Pour traduire en partie les spécifications fonctionnelles on va utiliser la notion d'interface Java.

```
/**
* Interface pour le type file générique
*@author moi
*@version 1.0
*/
public interface File<T>{
/** T est un paramètre de type de cette interface */
/** T représente le type des éléments de la file */
boolean vide();
void enfiler(T o);
void defiler();
T premier();
}
```

implémentation

```
public class FileAuto<T> implements File<T>{
//classe interne privée
    private class Cell{
    private T val;
    private Cell suiv;
    Cell(T v, Cell c){val=v; suiv=c;}
    T getVal(){return val;}
    Cell getSuiv(){return suiv;}
    void setSuiv(Cell c){suiv =c;}
  }
//1 unique attribut
   private Cell last;
// les méthodes de l'interface File
   public boolean vide(){
     return last == null;
   }
```

suite

```
public void enfiler(T o) {
if (first == null) {
 last=new Cell(o,first);
 last.setSuiv(first);}
 else {
 Cell c=new Cell(o,last.getSuiv());
 last.setSuiv(c);
 last=c;
}
public void defiler(){
if(last.getSuiv()==last) last=null;
else last.setSuiv(last.getSuiv().getSuiv());
}
public T premier(){
  return this.last.getSuiv().getVal();
}}
```

```
public class EssaiFileAuto {
public static void main(String[] args) {
 File<Integer> f = new FileAuto<Integer>();
   for (int i=1; i<=25;i++)  f.enfiler(i);
   f.defiler();
   f.defiler();
   System.out.println
   ("la tête de la file f est " + f.premier());

File<Float> f2= new FileAuto<Float>();
  for (int i=25; i>=1;i--)  f2.enfiler(i+20.0f);
  f2.defiler();
  System.out.println
  ("la tête de la file f2 est " + f2.premier());
  }
}
```

```
la tête de la file f est 3
la tête de la file f2 est 44.0
```

```
public class EssaiFileAuto {
public static void main(String[] args) {
 File<Integer> f = new FileAuto<Integer>();
   for (int i=1; i<=25;i++)  f.enfiler(i);
   f.defiler();
   f.defiler();
   System.out.println
   ("la tête de la file f est " + f.premier());

File<Float> f2= new FileAuto<Float>();
  for (int i=25; i>=1;i--)  f2.enfiler(i+20.0f);
  f2.defiler();
  System.out.println
  ("la tête de la file f2 est " + f2.premier());
  }
}
```

```
la tête de la file f est 3
lla tête de la file f2 est 44.0
```

La classe `LinkedList`

La classe `LinkedList<T>` du package `java.util` implémente l'interface `Collection<T>` et possède notamment les méthodes suivantes :

- T getFirst() : retourne le premier élément de la liste
- boolean add(T o) : ajoute l'objet o à la fin de la liste
- T removeFirst() : supprime et retourne le premier élément de la liste

La classe `FileLL<T>` utilise alors un attribut de type `LinkedList<T>` pour implémenter une file.

```
import java.util.LinkedList;
public class FileLL<T> implements File<T> {
private int lg=0;
private LinkedList<T> list;
public FileLL(){list = new LinkedList<T>();}
public boolean vide(){return lg==0;}
public void enfiler(T o){list.add(o); lg++;}
public T premier(){assert !vide(); return list.getFirst();}
public void defiler(){ list.removeFirst(); lg--; }
}
```

La classe `LinkedList`

La classe `LinkedList<T>` du package `java.util` implémente l'interface `Collection<T>` et possède notamment les méthodes suivantes :

- T getFirst() : retourne le premier élément de la liste
- boolean add(T o) : ajoute l'objet o à la fin de la liste
- T removeFirst() : supprime et retourne le premier élément de la liste

La classe `FileLL<T>` utilise alors un attribut de type `LinkedList<T>` pour implémenter une file.

```
import java.util.LinkedList;
public class FileLL<T> implements File<T> {
private int lg=0;
private LinkedList<T> list;
public FileLL(){list = new LinkedList<T>();}
public boolean vide(){return (lg==0);}
public void enfiler(T o){list.add(o); lg++;}
public T premier(){assert !vide(); return list.getFirst();}
public void defiler(){ list.removeFirst(); lg--; }
}
```

La classe `LinkedList`

La classe `LinkedList<T>` du package `java.util` implémente l'interface `Collection<T>` et possède notamment les méthodes suivantes :

- T getFirst() : retourne le premier élément de la liste
- boolean add(T o) : ajoute l'objet o à la fin de la liste
- T removeFirst() : supprime et retourne le premier élément de la liste

La classe `FileLL<T>` utilise alors un attribut de type `LinkedList<T>` pour implémenter une file.

```
import java.util.LinkedList;
public class FileLL<T> implements File<T> {
private int lg=0;
private LinkedList<T> list;
public FileLL(){list = new LinkedList<T>();}
public boolean vide(){return (lg==0);}
public void enfiler(T o){list.add(o); lg++;}
public T premier(){assert !vide(); return list.getFirst();}
public void defiler(){ list.removeFirst(); lg--; }
}
```

La classe `LinkedList`

La classe `LinkedList<T>` du package `java.util` implémente l'interface `Collection<T>` et possède notamment les méthodes suivantes :

- T getFirst() : retourne le premier élément de la liste
- boolean add(T o) : ajoute l'objet o à la fin de la liste
- T removeFirst() : supprime et retourne le premier élément de la liste

La classe `FileLL<T>` utilise alors un attribut de type `LinkedList<T>` pour implémenter une file.

```
import java.util.LinkedList;
public class FileLL<T> implements File<T> {
private int lg=0;
private LinkedList<T> list;
public FileLL(){list = new LinkedList<T>();}
public boolean vide(){return (lg==0);}
public void enfiler(T o){list.add(o); lg++;}
public T premier(){assert !vide(); return list.getFirst();}
public void defiler(){ list.removeFirst(); lg--; }
}
```

La classe `LinkedList`

La classe `LinkedList<T>` du package `java.util` implémente l'interface `Collection<T>` et possède notamment les méthodes suivantes :

- T getFirst() : retourne le premier élément de la liste
- boolean add(T o) : ajoute l'objet o à la fin de la liste
- T removeFirst() : supprime et retourne le premier élément de la liste

Le classe `FileLL<T>` utilise alors un attribut de type `LinkedList<T>` pour implémenter une file

```
import java.util.LinkedList;
public class FileLL<T> implements File<T> {
private int lg=0;
private LinkedList<T> list;
public FileLL(){list = new LinkedList<T>();}
public boolean vide(){return lg==0;}
public void enfiler(T o){list.add(o); lg++;}
public T premier(){return list.getFirst();}
public void defiler(){ list.removeFirst();lg--; }
}
```

Exercices

exercice 10 : Ecrire l'algorithme et une méthode qui inverse une file en utilisant uniquement une piles et réécrire l'inversion d'une pile en utilisant une file.

exercice 11 : Réécrire la méthode `equals()` pour une égalité en profondeur.

exercice 12 : Réécrire la méthode `clone()` pour un clonage en profondeur.

Interface Liste<T>

On va fabriquer notre version des listes pour les manipuler « à la LISP » c'est-à-dire en privilégiant les fonctions récursives.

```
public interface Liste<T extends Comparable> {
  boolean vide();
  Liste<T> ajout(T o);
  Liste<T> queue();
  T tete();
  Liste<T> fusion(Liste<T> l);
  public Liste<T> triFusion();
  int longueur();
}
```

```
public class ListeAuto<T extends Comparable<T>> implements Liste<T>{
  private class Cell{
    private T val;
    private Cell suiv;
    Cell(){};
    Cell(T v, Cell c){val=v; suiv=c;}
    void setVal(T v){val=v;}
    void setSuiv(Cell c){suiv=c;}
    T getVal(){return val;}
    Cell getSuiv(){return suiv;}
  }
   private Cell top=null;
   public ListeAuto(){}
   private ListeAuto(Cell c){
     top=c;
   }

   public boolean vide(){return top==null;}
```

```
public Liste<T> ajout(T o){
  Cell c= new Cell(o,this.top);
  ListeAuto<T> l=new ListeAuto<T>(c);
  return l;}

public Liste<T> queue(){
  Cell c= top.getSuiv();
  ListeAuto<T> l=new ListeAuto<T>(c);
  return l;}

public T tete(){
  return top.getVal();}

public  int longueur(){
  Liste<T> l=this;
  if (l.vide()) return 0;
  else  {
    l=l.queue();
    return(1+l.longueur());
  }}
```

Tri fusion

Le tri fusion est un algorithme efficace permettant de trier une liste de nombres x1, . . ., xn par comparaison et échange : il repose sur un post-traitement de la liste de nombres - que l'on appelera fusion - qui consiste à prendre deux listes déjà classées pour en faire une unique liste classée.
On partage donc en deux listes l1 et l2 la liste initiale. On trie l1 et l2 puis on les fusionne.
Pour trier l1 et l2, on va appliquer le même principe : c'est donc un algorithme récursif.
La récursivité s'arrête lorsque la liste est vide ou de longueur 1.

Tri fusion

Le tri fusion est un algorithme efficace permettant de trier une liste de nombres x1, . . ., xn par comparaison et échange : il repose sur un post-traitement de la liste de nombres - que l'on appelera fusion - qui consiste à prendre deux listes déjà classées pour en faire une unique liste classée.
On partage donc en deux listes l1 et l2 la liste initiale. On trie l1 et l2 puis on les fusionne.
Pour trier l1 et l2, on va appliquer le même principe : c'est donc un algorithme récursif.
La récursivité s'arrête lorsque la liste est vide ou de longueur 1.

Tri fusion

Le tri fusion est un algorithme efficace permettant de trier une liste de nombres x1, . . ., xn par comparaison et échange : il repose sur un post-traitement de la liste de nombres - que l'on appelera fusion - qui consiste à prendre deux listes déjà classées pour en faire une unique liste classée.
On partage donc en deux listes l1 et l2 la liste initiale. On trie l1 et l2 puis on les fusionne.
Pour trier l1 et l2, on va appliquer le même principe : c'est donc un algorithme récursif.
La récursivité s'arrête lorsque la liste est vide ou de longueur 1.

Exercices

exercice 13 : Ecrire les fonctions fusion puis triFusion

exercice 14 : Ecrire l'algorithme récursif et la fonction qui renvoie le miroir d'une liste.

exercice 15 : Ecrire l'algorithme récursif et la fonction qui renvoie la concaténation de deux listes.

exercice 16 : Ecrire l'algorithme et la fonction qui teste si une liste donnée est le début d'une liste donnée en paramètre.

exercice 17 : Ecrire l'algorithme et la fonction qui teste si une liste donnée est contenue dans une liste donnée en paramètre.

Chapitre II – Structures arborescentes

Chapitre II – Structures arborescentes

Chapitre II – Structures arborescentes

Structures de données arborescentes

De façon informelle, on parlera de structure de données arborescente lorsqu'une donnée a plusieurs successeurs mais un seul prédécesseur, sauf pour exactement une donnée qui est sans prédécesseur et appelée *racine*.
On s'intéressera tout d'abord aux arbres binaires dans lesquels chaque donnée a au plus deux successeurs : un fils droit et un fils gauche.

Spécification fonctionnelle

On définit le type de données abstrait Arbre Binaire suivant :
type Arbre Binaire
utilise type E, **type** booléen
signature

- *initilialiser* : → Arbre Binaire
- *faire_arbre* : (E,Arbre Binaire,Arbre Binaire)→ Arbre Binaire
- *fils_gauche* : Arbre Binaire → Arbre Binaire
- *fils_droit* : Arbre Binaire → Arbre Binaire
- *valracine* : Arbre Binaire → E
- *feuille* : Arbre Binaire → booléen
- *vide* : Arbre Binaire → booléen

préconditions

- *valracine*(a) est défini *si et seulement sivide*(a)= Faux
- *fils_gauche*(a) est défini *si et seulement sivide*(a)= Faux
- *fils_droit*(a) est défini *si et seulement sivide*(a)= Faux

axiomes

- *vide*(*initilialiser*)= Vrai
- *vide*(*faire_arbre(E,ag,ad)*)= Faux
- *valracine*(*faire_arbre(E,ag,ad)*)= E
- *fils_gauche*(*faire_arbre(E,ag,ad)*) = ag
- *fils_droit*(*faire_arbre(E,ag,ad)*) = ad
- *feuille*(*faire_arbre(E,vide,vide)*) = Vrai
- *feuille*(vide) = Faux

Interface ArbreBinaire

```
/**
* Interface pour le type arbre binaire générique
*@author moi
*@version 1.0
*/
public Interface ArbreBinaire<T>{
/** T est un paramètre de type de cette interface */
/** T représente le type des éléments de l' ArbreBinaire */
boolean vide();
boolean feuille();
T valRacine();
ArbreBinaire<T> filsGauche();
ArbreBinaire<T> filsDroit();
}
```

```
public class AB<T> implements ArbreBinaire<T>{
//classe interne privée
  private class Noeud{
  private T val;
  private Noeud fg,fd;
  Noeud(T v, Noeud cg, Noeud cd){val=v; fg=cg; fd=cd;}
  T getVal(){return val;}
  Noeud getFilsGauche(){return fg;}
  Noeud getFilsDroit(){return fd;}
}
//1 unique attribut
 private Noeud racine;
//Constructeurs
public AB(){racine=null;}
public AB(T item, AB<T> g, AB<T> d)
{Noeud c=new Noeud(item, g.racine, d.racine);
 this.racine = c;}
//les méthodes de l'interface ArbreBinaire
 public boolean vide(){
   return racine == null;
 }
 public T valRacine(){
   return this.racine.getVal();
 }
```

```
public AB<T> filsGauche(){
  AB<T> a= new  AB<T>();
a.racine =  this.racine.fg;
return a;}
public AB<T> filsDroit(){
  AB<T> a= new  AB<T>();
  a.racine =  this.racine.fd;
  return a;
}
public boolean feuille() {
  if (this.vide()) return false;
  if(this.racine.fg== null && this.racine.fg==null)
    return true;
  else return false;
  }
  }
```

Exercices

exercice 18 : Ecrire la méthode qui renvoie la hauteur d'un arbre binaire

exercice 19 : Ecrire la méthode qui renvoie la taille d'un arbre binaire

exercice 20 : Ecrire la méthode qui renvoie le nombre de feuilles d'un arbre binaire

exercice 21 : Réécrire la méthode `equals` (mêmes valeurs aux mêmes noeuds)

Parcours des arbres binaires

On va étudier deux méthodes de visites des arbres binaires :

- en profondeur : à partir de la racine de l'arbre on va toujours le plus à gauche possible. Lorsque le sous-arbre gauche d'un noeud est vide :
  - si le sous-arbre droit n'est pas vide on va sur sa racine puis on reprend le chemin le plus à gauche possible.
  - si le sous-arbre droit est vide le noeud est une feuille et on va sur le père de ce noeud, puis à la racine de son sous-arbre droit (frère du noeud feuille) pour reprendre à partir de là le chemin le plus à gauche possible.
- en largeur : niveau par niveau, de gauche à droite en commençant par la racine ; on parle aussi de parcours hiérarchique.

Parcours en profondeur

Dans ce parcours, chaque noeud est vu trois fois, dans l'ordre suivant :

1. en descente : préfixe
2. en montée depuis la gauche : infixe
3. en montée depuis la droite : suffixe

Ce principe de parcours en profondeur est **récursif**.

exercice 22 : Ecrire trois méthodes d'affichage préfixe, infixe et suffixe d'un arbre binaire.

parcours en largeur

On parcourt l'arbre binaire selon sa numérotation hiérarchique : la racine a le numéro 1 puis son fils gauche et son fils droit ont respectivement les numéros 2 et 3, ainsi de suite, niveau par niveau.
On remarque ainsi que que le fils gauche d'un noeud numéro k a le numéro $2k$ et son fils le numéro $2k + 1$.
Pour chaque noeud il faudra visiter ses deux fils (dans l'ordre gauche - droite) seulement après avoir vu tous les autres noeuds du même niveau que lui et tous les noeuds du niveau de ses fils qui se trouvent à leur gauche.
Pour pouvoir conserver cet ordre de visite, on utilisera une *file de noeuds* dans laquelle on enfilera successivement les fils de l'élément de tête de la file juste avant de le défiler (supprimer de la file).

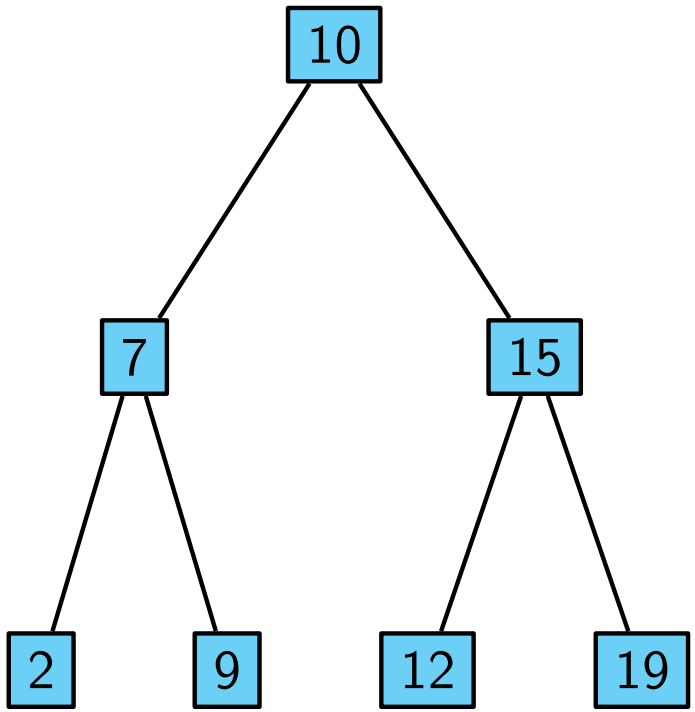
exercice 23 : Ecrire la méthode qui affiche les noeuds selon l'ordre du parcours en largeur.

Arbres binaires de recherche

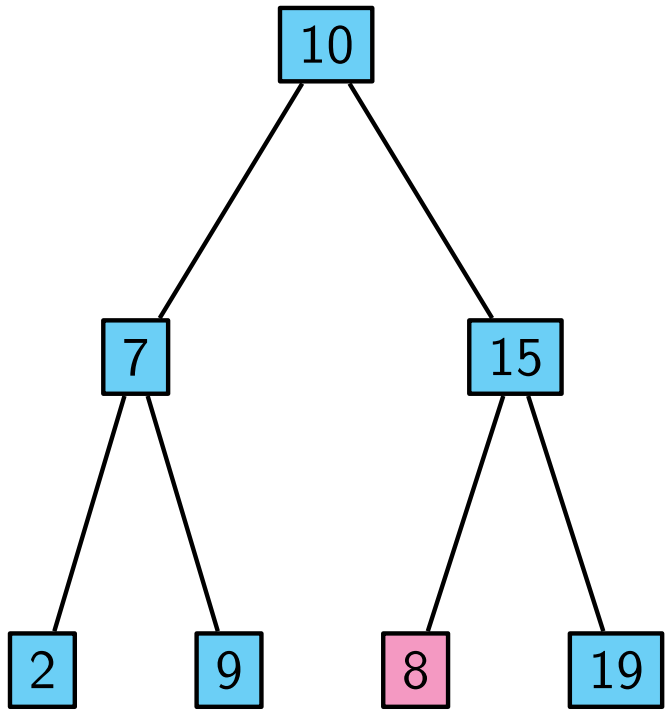
Définition

un *arbre binaire de recherche* est un arbre binaire dans lequel **pour chaque noeud**, si sa valeur est v, les valeurs contenues dans son sous-arbre gauche sont inférieures à v et les valeurs contenues dans son sous-arbre droit sont supérieures à v.

exemples



exemples



Avantages des ABR

- les valeurs sont « triées »
- le parcours infixe d'ABR fournit la liste des valeurs triées en ordre croissant
- la plus petite valeur est nécessairement sur le premier noeud du parcours infixe qui n'a pas de sous-arbre gauche soit le noeud qui est le plus à gauche.
- de même, la plus grande valeur est nécessairement sur le dernier noeud du parcours infixe qui n'a pas de sous-arbre droit soit le noeud qui est le plus à droite.
- la recherche est rapide et se fait en temps proportionnel à

Avantages des ABR

- les valeurs sont « triées »
- le parcours infixe d'ABR fournit la liste des valeurs triées en ordre croissant
- la plus petite valeur est nécessairement sur le premier noeud du parcours infixe qui n'a pas de sous-arbre gauche soit le noeud qui est le plus à gauche.
- de même, la plus grande valeur est nécessairement sur le dernier noeud du parcours infixe qui n'a pas de sous-arbre droit soit le noeud qui est le plus à droite.
- la recherche est rapide et se fait en temps proportionnel à

Avantages des ABR

- les valeurs sont « triées »
- le parcours infixe d'ABR fournit la liste des valeurs triées en ordre croissant
- la plus petite valeur est nécessairement sur le premier noeud du parcours infixe qui n'a pas de sous-arbre gauche soit le noeud qui est le plus à gauche.
- de même, la plus grande valeur est nécessairement sur le dernier noeud du parcours infixe qui n'a pas de sous-arbre droit soit le noeud qui est le plus à droite.
- la recherche est rapide et se fait en temps proportionnel à

Avantages des ABR

- les valeurs sont « triées »
- le parcours infixe d'ABR fournit la liste des valeurs triées en ordre croissant
- la plus petite valeur est nécessairement sur le premier noeud du parcours infixe qui n'a pas de sous-arbre gauche soit le noeud qui est le plus à gauche.
- de même, la plus grande valeur est nécessairement sur le dernier noeud du parcours infixe qui n'a pas de sous-arbre droit soit le noeud qui est le plus à droite.
- la recherche est rapide et se fait en temps proportionnel à

Avantages des ABR

- les valeurs sont ≪ triées ≫
- le parcours infixe d'ABR fournit la liste des valeurs triées en ordre croissant
- la plus petite valeur est nécessairement sur le premier noeud du parcours infixe qui n'a pas de sous-arbre gauche soit le noeud qui est le plus à gauche.
- de même, la plus grande valeur est nécessairement sur le dernier noeud du parcours infixe qui n'a pas de sous-arbre droit soit le noeud qui est le plus à droite.
- la recherche est rapide et se fait en temps proportionnel à la hauteur de l'ABR

Avantages des ABR

- les valeurs sont ≪ triées ≫
- le parcours infixe d'ABR fournit la liste des valeurs triées en ordre croissant
- la plus petite valeur est nécessairement sur le premier noeud du parcours infixe qui n'a pas de sous-arbre gauche soit le noeud qui est le plus à gauche.
- de même, la plus grande valeur est nécessairement sur le dernier noeud du parcours infixe qui n'a pas de sous-arbre droit soit le noeud qui est le plus à droite.
- la recherche est rapide et se fait en temps proportionnel à la hauteur de l'ABR

Avantages des ABR

- les valeurs sont ≪ triées ≫
- le parcours infixe d'ABR fournit la liste des valeurs triées en ordre croissant
- la plus petite valeur est nécessairement sur le premier noeud du parcours infixe qui n'a pas de sous-arbre gauche soit le noeud qui est le plus à gauche.
- de même, la plus grande valeur est nécessairement sur le dernier noeud du parcours infixe qui n'a pas de sous-arbre droit soit le noeud qui est le plus à droite.
- la recherche est rapide et se fait en temps proportionnel à la hauteur de l'ABR

Inconvénients des ABR

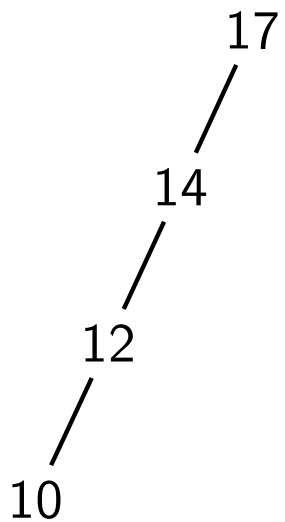
- la manipulation est délicate
- un ABR peut être très déséquilibré, le bénéfice de la recherche dichotomique est alors nul

Inconvénients des ABR

- la manipulation est délicate
- un ABR peut être très déséquilibré, le bénéfice de la recherche dichotomique est alors nul

Inconvénients des ABR

- la manipulation est délicate
- un ABR peut être très déséquilibré, le bénéfice de la recherche dichotomique est alors nul



Exercices

exercice 24 : Ecrire la classe

`ABR<T extends Comparable<T>>`

qui hérite de la classe `Arbrebinaire<T>`

Arbres équilibrés AVL

FIGURE: *Georgy Maximovich Adelson Velski et Evgueni Mikhailovitch Landis*

Arbres équilibrés AVL

Définition

On définit la fonction d'équilibre δ d'un arbre binaire par

- si α est vide alors $\delta(\alpha) = 0$
- sinon $\delta(\alpha) = h(g(\alpha)) - h(d(\alpha))$

Arbres équilibrés AVL

Définition

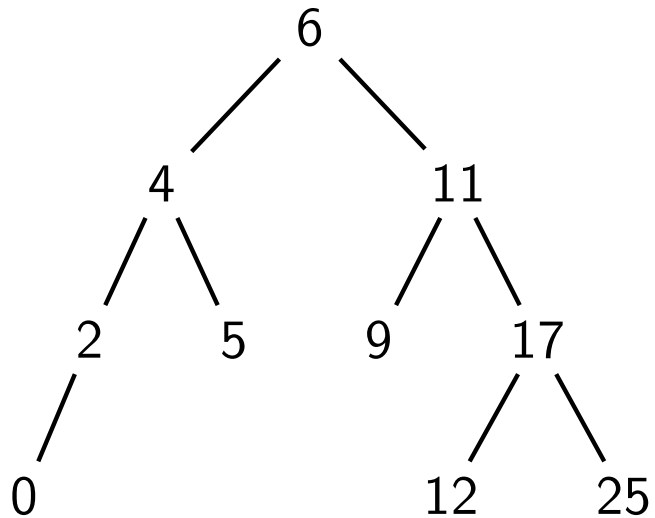
On définit la fonction d'équilibre δ d'un arbre binaire par

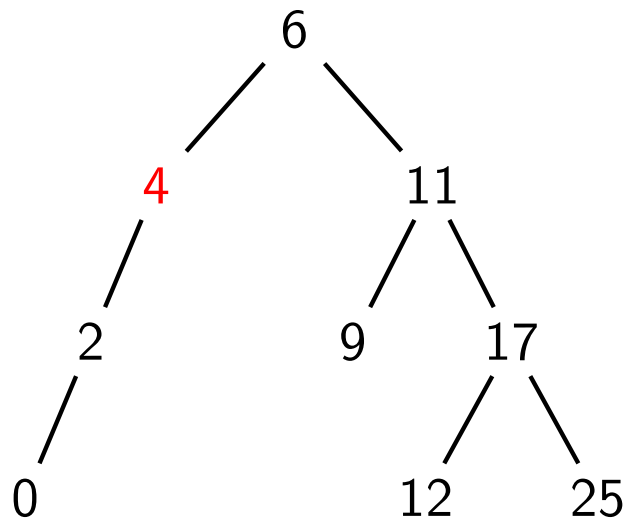
- si α est vide alors $\delta(\alpha) = 0$
- sinon $\delta(\alpha) = h(g(\alpha)) - h(d(\alpha))$

Définition

Un arbre AVL [a] α est un ABR tel que pour tout noeud β de α, $\delta(\beta) \in \{-1, 0, 1\}$.

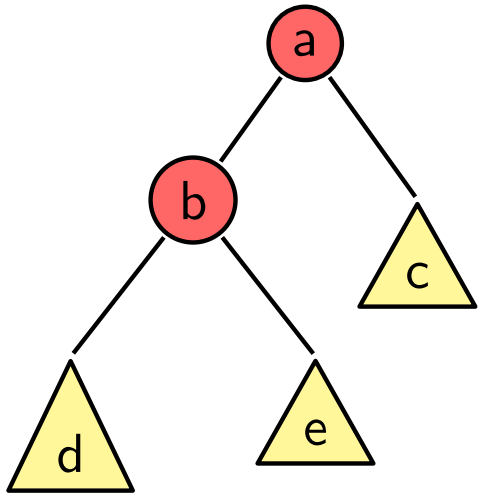
a. Addelson-Velski, Landis



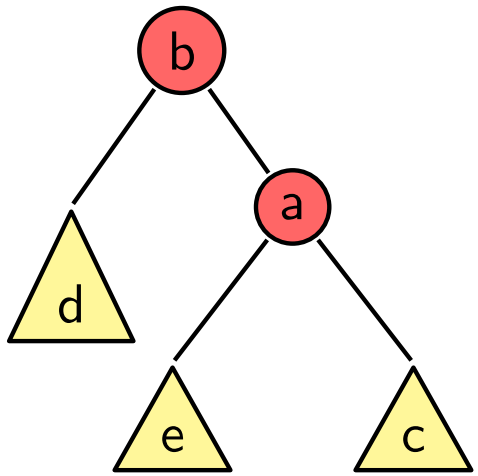


Pour implémenter un AVL, on utilisera donc les fonctionnalités des ABR auxquelles on ajoutera ces deux procédures de « rééquilibrages » (des rotations) ainsi qu'une fonction de test d'équilibre.
Ces procédures de rééquilibrages seront utilisées après chaque insertion d'une valeur sur le sous-arbre qui ne sera plus équilibré du fait de cette insertion.
On disposera deux types de rotations : rotation gauche et rotation droite.

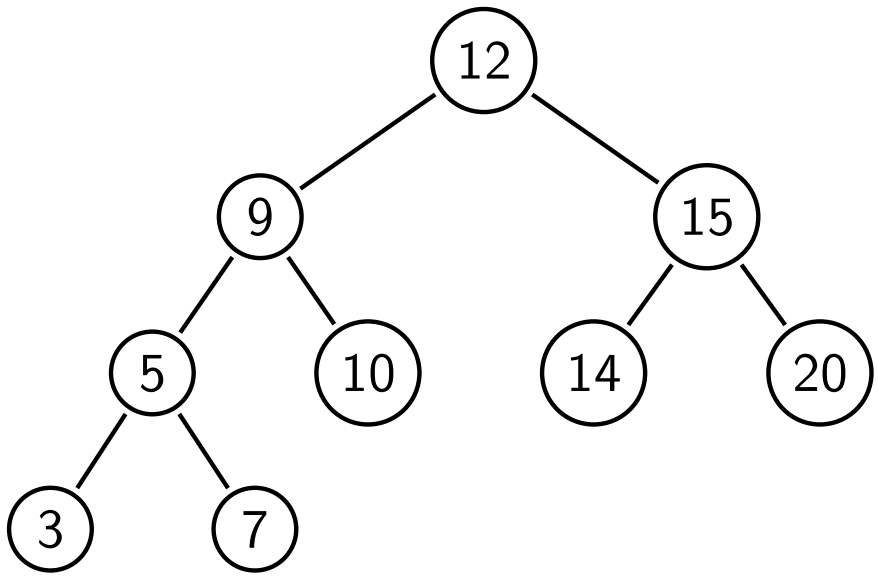
Dans l'arbre ABR ci-dessous on a $d < b < e < a < c$ et le déséquilibre provient de la différence de hauteur entre le sous-arbre droit de a et son sous-arbre gauche.



On va donc effectuer une rotation droite qui va faire monter b d'un niveau et baisser a d'un niveau.

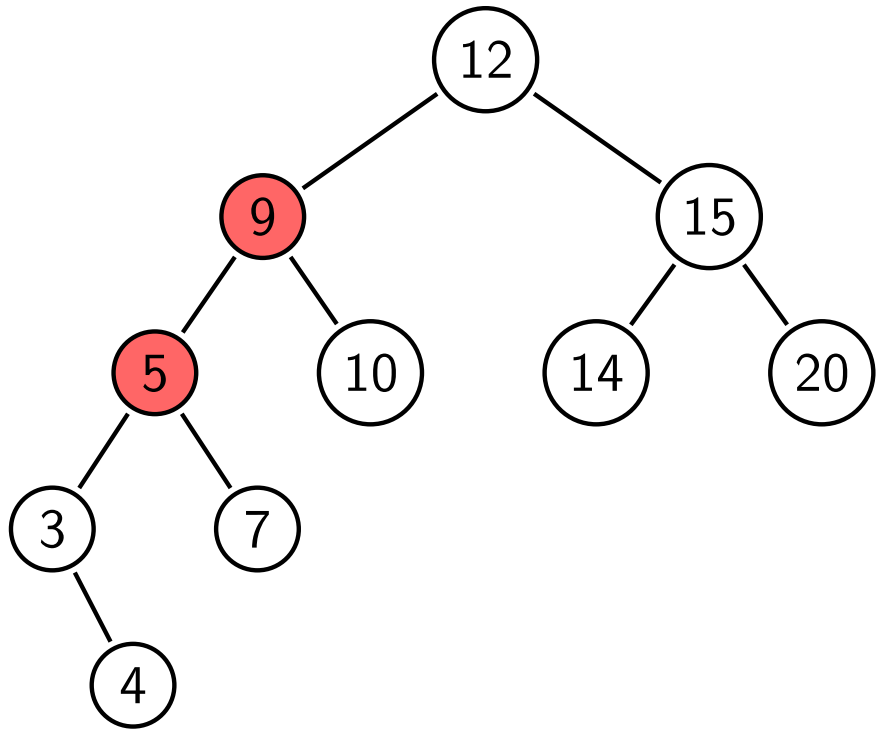


exemple de rotation droite

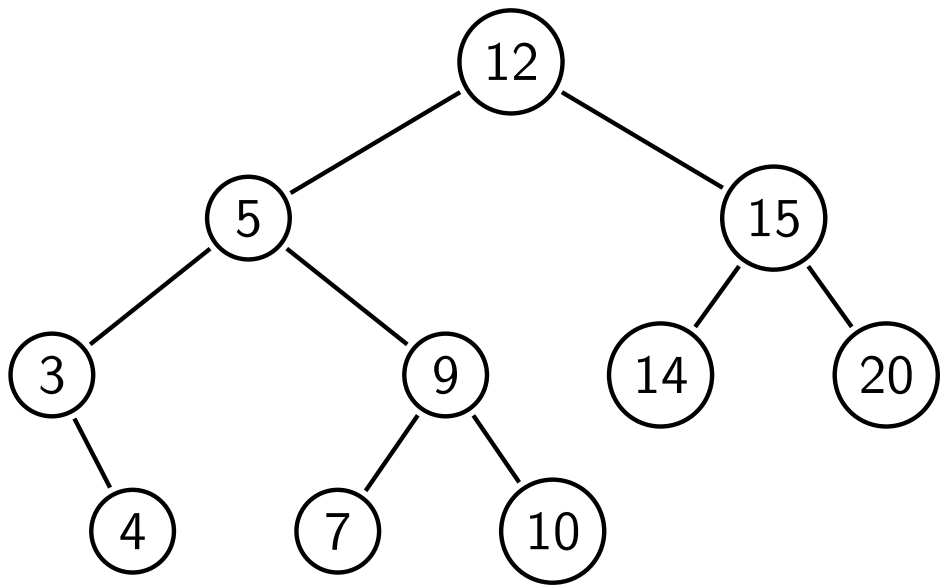


exemple de rotation droite

On ajoute la valeur 4, le sous-arbre 9 est déséquilibré.



exemple de rotation droite



Files de priorité

Définition

Une *file de priorité* est une stucture de données dans laquelle

- les éléments sont **ordonnées**
- le premier élément supprimé est le maximum (priorité la plus haute)

Par exemple un serveur recevant ces requêtes ayant chacune un certain niveau de priorité doit servir en premier la requêtes la plus prioritaire.
Remarque : le premier supprimé peut être le minimum selon la définition de la priorité.

spécification fonctionnelle

type FILEDEPRIORITÉ
utilise type ordonné E, **type** BOOLÉEN
utilise fonction priorité : (E,E) → BOOLÉEN
signature

- *initilialiser* : → FILEDEPRIORITÉ
- *ajouter* : (FILEDEPRIORITÉ,E)→ FILEDEPRIORITÉ
- *retirer* : FILEDEPRIORITÉ → FILEDEPRIORITÉ
- *premier* : FILEDEPRIORITÉ → E
- *vide* : FILEDEPRIORITÉ → BOOLÉEN

préconditions

- *retirer*(fp) est défini *si et seulement sivide*(fp)= Faux
- *premier*(fp) est défini *si et seulement sivide*(fp)= Faux

axiomes

- *vide*(*initilialiser*)= Vrai
- *vide*(*ajouter*(fp,e))=Faux
- *premier*(fp) = e *si et seulement si*priorité(e,x) = Vrai pour tout $x \in fp$
- *priorité*(*premier*(fp),*premier*(retirer(fp))) = Vrai

Représentation

On peut utiliser

- une liste : *ajouter* se fera en $O(1)$ mais *retirer* et *premier* en $O(n)$
- une liste (ou une tableau ordonné) : *retirer* et *premier* se feront en $O(1)$ mais ajouter en $O(n)$
- un tas : *premier* se fera en $O(1)$ et *ajouter* et *retirer* se feront en $O(\ln n)$

Tas (heap)

Définition

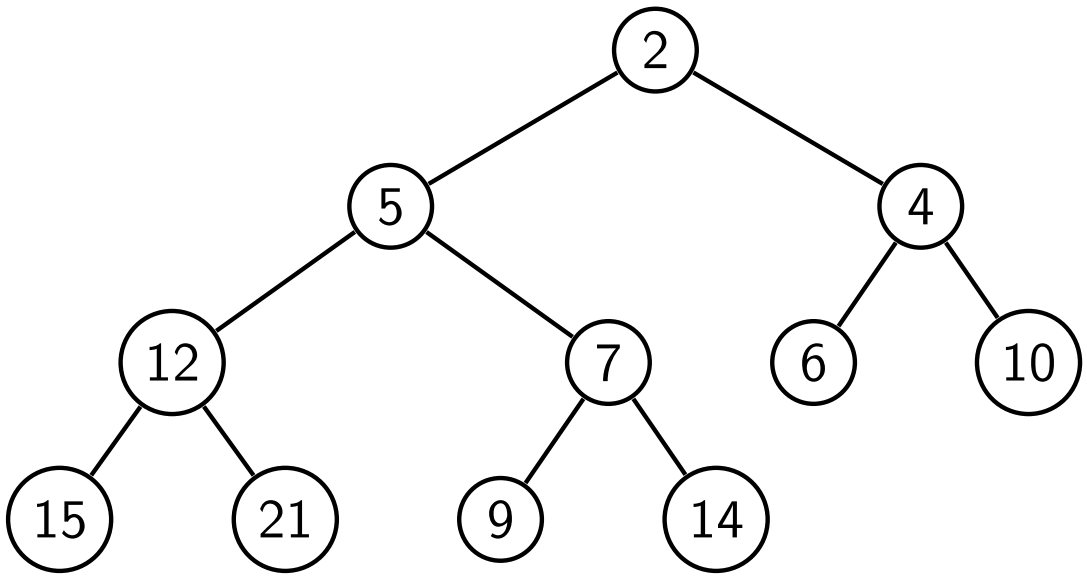
Un *tas* est un arbre binaire ayant les deux caractéristiques suivantes :

- arbre parfait : c'est un arbre binaire dont tous les niveaux hiérarchiques sont remplis, sauf éventuellement le dernier dans lequel toutes les feuilles sont le plus à gauche possible.
- arbre tournoi : c'est un arbre binaire dont toutes les valeurs sont croissantes depuis la racine vers les feuilles.

Un tas (heap en anglais) est un arbre binaire **parfait** et **tournoi**.

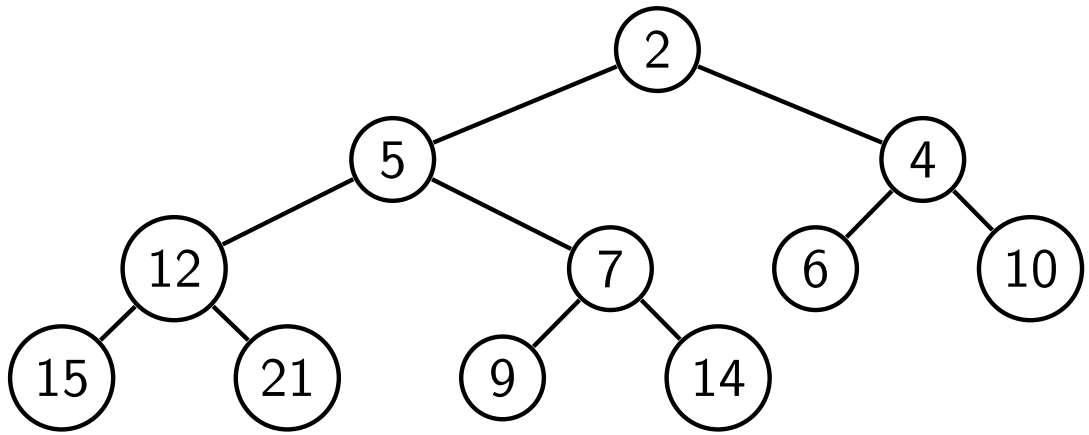
exemple

Remarque : on peut aussi décider que les valeurs seront *décroissantes* depuis la racine vers les feuilles, selon la fonction de priorité considérée.
Exemple :



Représentation physique

Le gros avantage d'un arbre parfait est sa compacité qui permet d'utiliser la *numérotation hiérarchique* pour représenter cet arbre dans un *tableau* dont l'indexage correspond au numéro hiérarchique.
Exemple :



L'arbre ci-dessus est représenté par le tableau suivant :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	...
$t(i)$	2	5	4	12	7	6	10	15	21	9	14	...	...	...

Remarques

- la valeur prioritaire d'un tas est toujours à la racine, d'indice 0
- le fils gauche d'un noeud k a le numéro $2k + 1$
- le fils droit d'un noeud k a le numéro $2k + 2$
- le père d'un noeud k, pour $k >= 1$ a le numéro $\lfloor (k-1)/2 \rfloor$

Remarques

- la valeur prioritaire d'un tas est toujours à la racine, d'indice 0
- le fils gauche d'un noeud k a le numéro $2k + 1$
- le fils droit d'un noeud k a le numéro $2k + 2$
- le père d'un noeud k, pour $k \geq 1$ a le numéro $\lfloor (k-1)/2 \rfloor$

Remarques

- la valeur prioritaire d'un tas est toujours à la racine, d'indice 0
- le fils gauche d'un noeud k a le numéro $2k + 1$
- le fils droit d'un noeud k a le numéro $2k + 2$
- le père d'un noeud k, pour $k \geq 1$ a le numéro $\lfloor (k-1)/2 \rfloor$

Remarques

- la valeur prioritaire d'un tas est toujours à la racine, d'indice 0
- le fils gauche d'un noeud k a le numéro $2k + 1$
- le fils droit d'un noeud k a le numéro $2k + 2$
- le père d'un noeud k, pour $k > 1$ a le numéro $\lfloor (k - 1)/2 \rfloor$

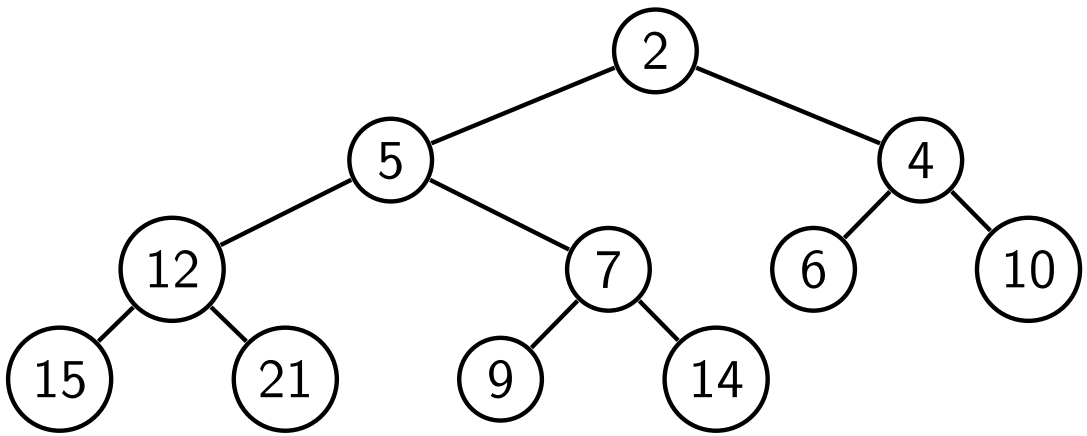
retirer

On sait qui va être retiré puisque c'est l'élément qui a priorité, ici dans le tas c'est le premier élément du tableau. Mais cette suppression doit se faire en maintenant la structure de tas : on va

1. supprimer la dernière feuille (au dernier niveau, la plus à droite) pour maintenir un arbre **parfait**
2. **remplacer** la racine de l'arbre par l'élément restant le plus prioritaire.

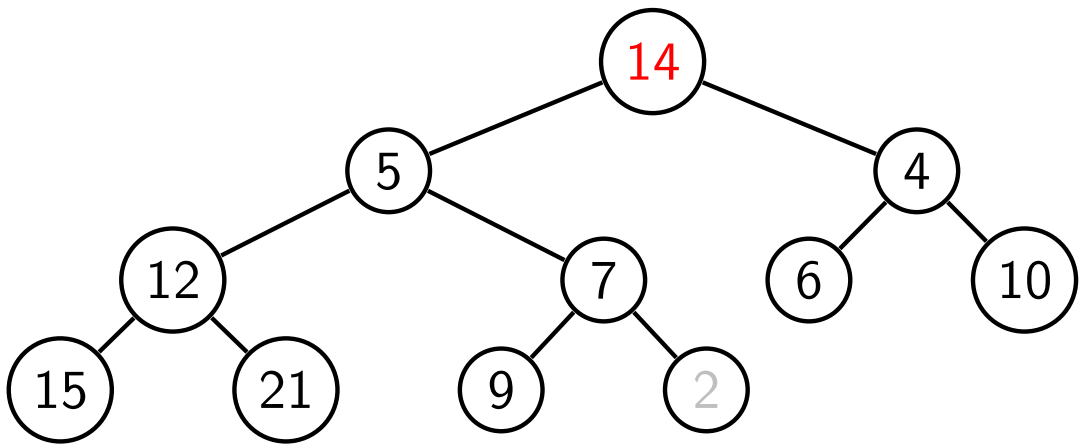
retirer

la valeur 2 doit être retirée :



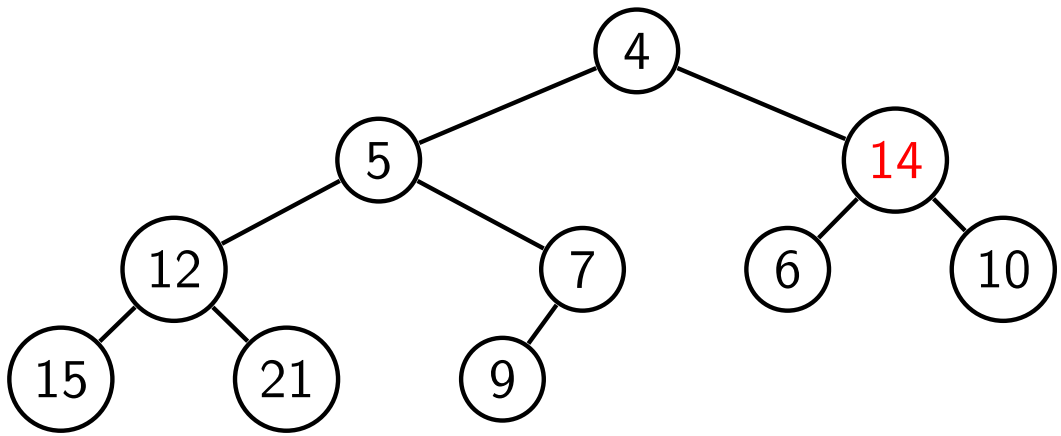
retirer

on supprime la dernière feuille en échangeant la valeur 2 avec la valeur de la dernère feuille



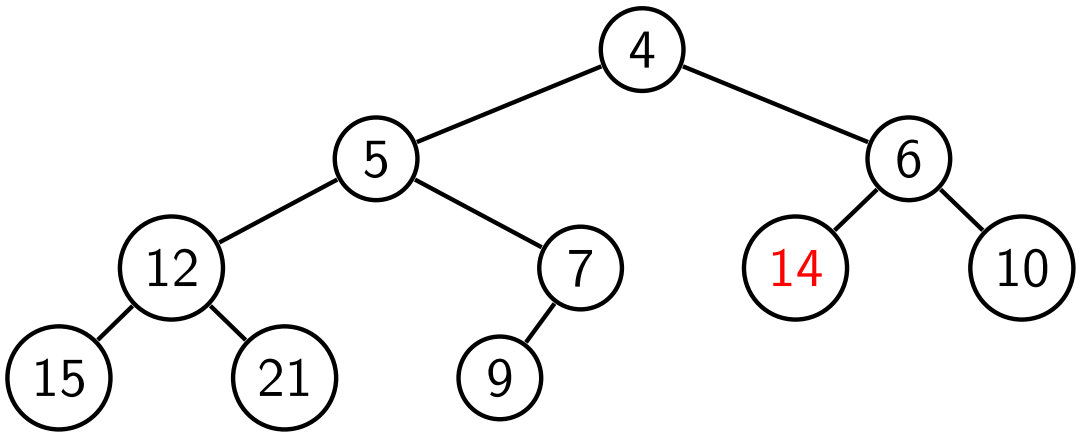
retirer

On rétablit l'ordre de haut en bas par des échanges



retirer

On rétablit l'ordre de haut en bas par des échanges entre la valeur du noeud et le plus petit de ces deux fils



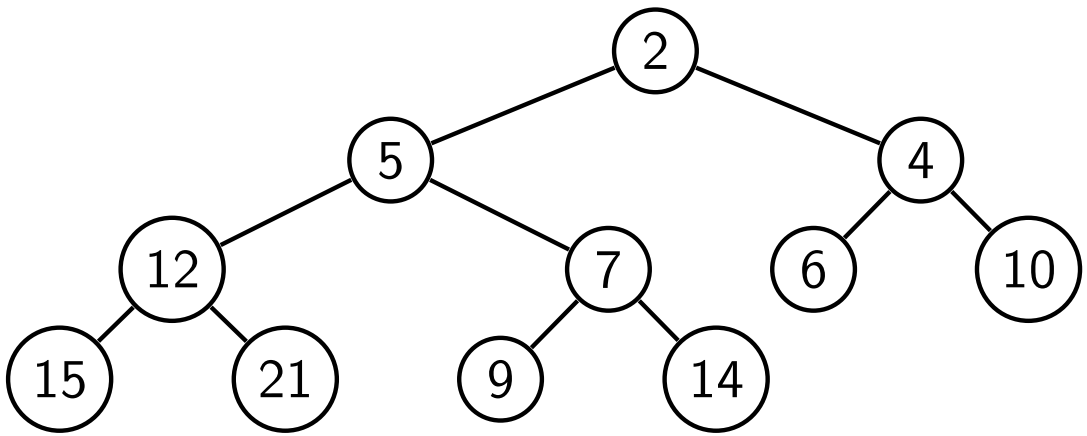
ajouter

Pour *ajouter*, on procède de même en deux étapes principales :

1. on crée une nouvelle feuille au dernier niveau juste à droite de la dernière feuille avec la valeur insérée pour maintenir un arbre parfait,
2. on rétablit l'ordre de haut en bas par échanges depuis la dernière feuille en remontant vers la racine

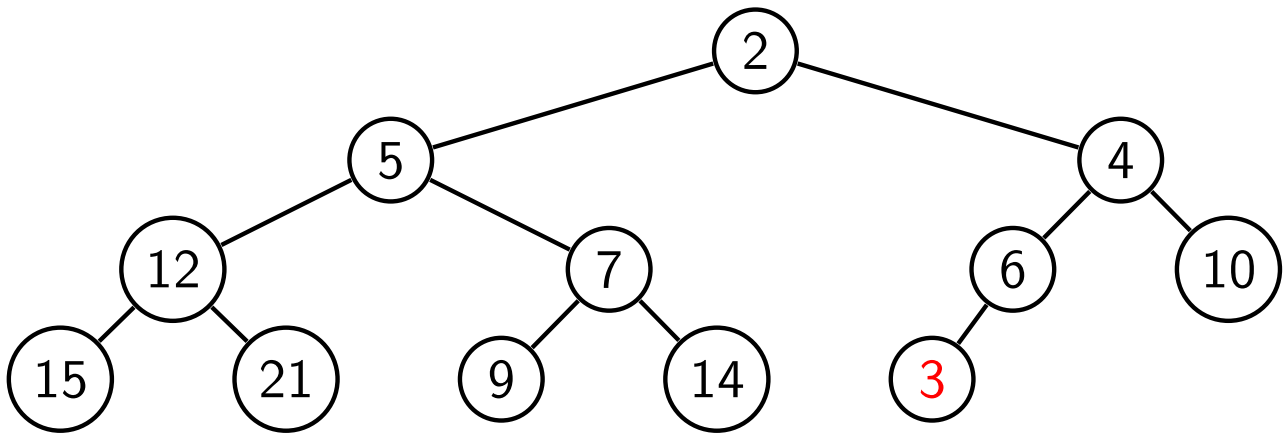
ajouter

On veut insérer la valeur 3 :



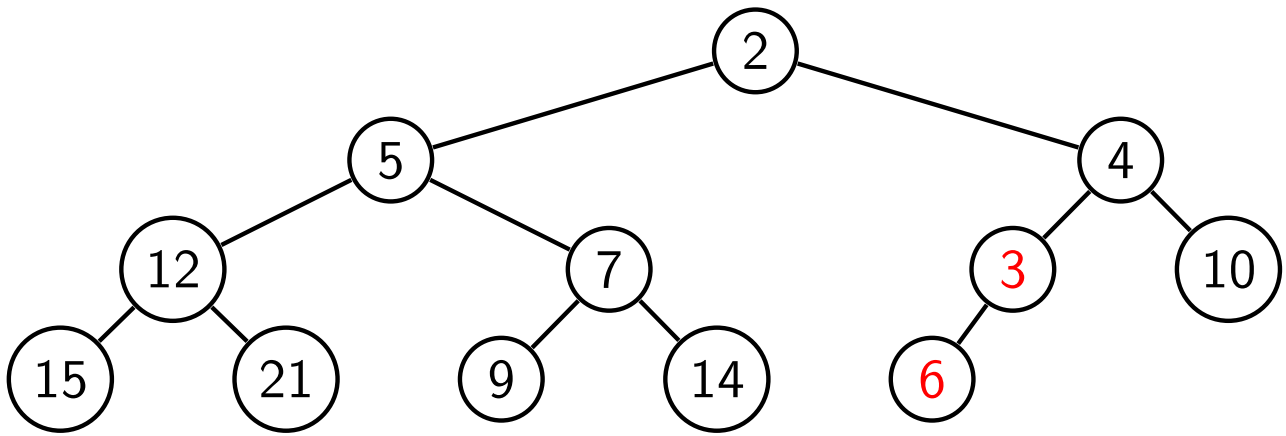
ajouter

on crée une feuille au dernier niveau pour y mettre la valeur 3



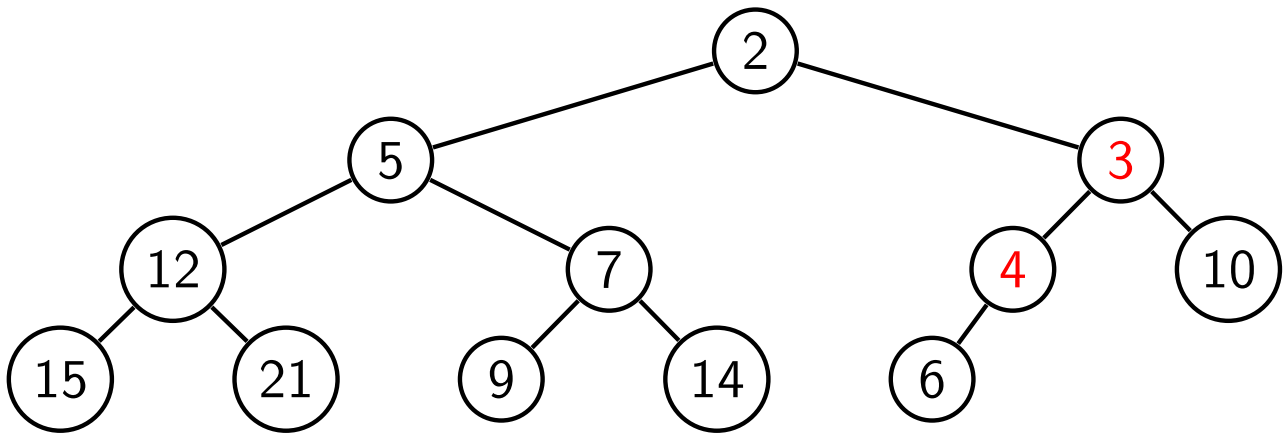
ajouter

on échange la valeur 3 avec son noeud père s'il est supérieur



ajouter

On rétablit l'ordre de haut en bas par des échanges



Efficacité

Pour un arbre parfait de hauteur h avec n valeurs, on a $2^h - 1 < n \leq 2^{h+1} - 1$

Sachant que les opérations ajouter et retirer prennent au maximum un temps proportionnel à la hauteur du tas, on peut conclure que, pour un tas de n valeurs, les deux opérations prennent un temps en $O(log_2 n)$.

Par exemple pour 1000000 de valeurs une opération de suppression ou d'insertion prend un temps de l'ordre de 20.

Interface FileDePriorite

```
/**
* Interface pour le type FileDePriorite
*@author moi
*@version 0.1
*/
public Interface FileDePriorite<T extends Comparable>{
boolean vide();
void ajouter(T o);
void retirer();
T premier();
int taille();
}
```

classe Tas

exercice 25 : Compléter la classe `Tas` qui implémente `FileDePriorite`

```
import java.util.ArrayList;
public class Tas<T extends Comparable<T>>)
                implements FileDePriorite<T>{
   int n;
   ArrayList<T> a;
   Tas(int k){ a= new ArrayList<T>(k); n=0;}
   public T premier(){ return this.a.get(0);}
   public boolean vide() {return n==0; }
   public int taille(){return n;}
   public void ajouter(T o) {   }
   public void retirer() {   }
}
```

Chapitre III – Documeter un projet

- Documenter un projet
- Générer le fichier html
- exemple

Chapitre III – Documeter un projet

Chapitre III – Documeter un projet

Annotations

On peut annoter son programme pour obtenir un fichier html de commentaires d'un format semblable à l'API Java.

- commencer par $/**$, terminer par $*/$ (entre les deux commencer chaque ligne par $*$).
- @auteur : il peut y avoir plusieurs auteurs
- @see : pour créer un lien vers un autre document
- @param : pour indiquer les paramètres d'une méthode
- @exception : pour indiquer quelle exception est levée
- @return : pour indiquer la valeur de retour d'une méthode
- @version : pour donner le numéro de la version du code
- @since : pour donner le numéro de la version initiale
- @deprecated : pour indiquer qu'une méthode ne devrait plus être utilisée, (cela crée un warning à la compilation)
- @serial : pour les objets sérialisables

en ligne de commande

Après compilation de `MonProgramme.java`, la commande
`javadoc MonProgramme.java`
engendre un fichier `MonProgramme.html`.
Options de la commande `javadoc` :

- -author : Indique que les commentaires tagés par `@author` devront être utilisées pour générer la documentation (par défaut, ces informations ne sont pas utilisées).
- -d repertoire : Permet de préciser le repertoire où javadoc placera les fichiers HTML générés (par défaut, répertoire courant).
- -public : Ne documente que les membres (méthodes, constructeurs, attributs) publics.
- -private : Documente tous les membres (méthodes, constructeurs, attributs), quelle que soit leur visibilité (par défaut membres publics et protected)

sous Eclipse

Dans le menu aller sur `Project` puis choisir Generate Javadoc

Un exemple

```java
import java.util.*;
/** Le premier exemple de programme  Java.
* Affiche une chaîne de caractères et la date du jour.
* @author moi
*@author http://www.lacl.u-pec/moi/
*@version 0.0  */
public class BonjourDate {
/** Unique point d'entrée de la classe et de l'application
*@param args tableau de paramètres sous forme de chaînes de caractères
*@return Pas de valeur de retour
*@exception exceptions Pas d'exceptions émises */
public static void main(String[] args) {
System.out.println("Bonjour, aujourd'hui: ");
System.out.println(new Date());}}
```

suite

voilà la liste des fichiers engendrés :

- index.html
- allclasses-frame.html
- allclasses-noframe.html
- constant-values.html
- deprecated-list.html
- BonjourDate.html
- help-doc.html
- index-all.html
- overview-tree.html
- package-frame.html
- package-summary.html
- package-tree.html
- package-list
- stylesheet.css

Autres formats

Il existe des outils (Doclet) pour exporter la documentation sous d'autres formats (XML, pdf, /dots)