Implémentation de tris classiques

DIU Enseigner l'Informatique au Lycée

L. Mounier, C. Parent-Vigouroux, A. Rasse, B. Wack

UFR IM2AG, Université Grenoble Alpes

mai 2019

Plan

Tri par sélection
- Itératif VS récursif
- Validation et test

Tri par fusion
- Gestion de la mémoire

Tri par tas
- Une structure de données spécifique
- Opérations
- Un peu de programmation orientée objet

Itératif VS récursif

Beaucoup d'algorithmes peuvent s'incarner aussi bien en un programme itératif que récursif.

```
def exp_iter(x, n):
    r = 1
    while n > 0:
        if n%2 == 1:
            r = r * x
        x = x**2
        n = n//2
    return r
```

```
def exp_rec(x, n):
    if n == 0:
        return 1
    else:
        r = exp_rec(x, n//2)
        if n%2 == 0:
            return r * r
        else:
            return x * r * r
```

À essayer aujourd'hui : tri par sélection sans utiliser de boucle !

Indication : utiliser un argument supplémentaire pour indiquer sur quelle portion du tableau on travaille.

Validation expérimentale d'un programme

Valider un programme

= vérifier que ce qu'il fait **correspond à sa spécification**.

Pour quoi faire ?

- « Trouver des erreurs » plutôt que « vérifier que ça marche »
- Attention, en général un ou plusieurs test(s) positif(s) ne constituent pas une **preuve** de validité
- On ne cherche pas spécialement à vérifier que le programme « ne plante pas » (robustesse)

Dans notre cas, on se contentera de tests (mais d'autres méthodes existent : model-checking, analyse statique...)

Bonnes pratiques de test

1. identifier les données du programme, et parmi elles le domaine de valeurs sur lequel on souhaite tester le programme
2. écrire un jeu d'essai (**plusieurs** tests) tel que :
   - chaque test soit **pertinent**
   - on **couvre** bien le domaine recherché
   - si on connaît le code du programme (boîte blanche), tous les chemins d'exécution soient testés
   - éviter de tester des valeurs hors du domaine, notamment ne remplissant pas les préconditions (robustesse encore)
3. éventuellement, générer le jeu d'essai automatiquement
4. pour chaque test du jeu d'essai :
   - exécuter et vérifier à la main la validité du résultat
   - ou mieux, écrire un oracle : un programme vérifiant que le résultat est conforme

Revenons au problème du tri

Quel domaine souhaite-t-on tester ?

- Listes de différentes tailles
- Éventuellement varier les contenus aussi
  (éléments non consécutifs, doublons...)
- Différentes permutations, dont certaines particulières

Un oracle pour le tri

Deux possibilités :

- Partir d'une liste triée et la mélanger
  (`from random import shuffle`)
  (on sait quel sera le résultat avant de générer la donnée)
- Ou vérifier que le résultat convient
  (attention, il ne suffit pas de vérifier qu'il est ordonné !)

Rappel : principe du tri par fusion

```
def tri_fusion_rec(L, g, d):
    """ trie le segment L[g:d] """
    if g < d-1:
        m = (g+d)//2
        tri_fusion_rec(L, g, m)
        tri_fusion_rec(L, m, d)
        fusion(L, g, m, d)

def tri_fusion(L):
    tri_fusion_rec(L, 0, len(L))
```

Problème : la fusion est extrêmement difficile à réaliser en place.

Gestion de l'espace mémoire supplémentaire pour la fusion

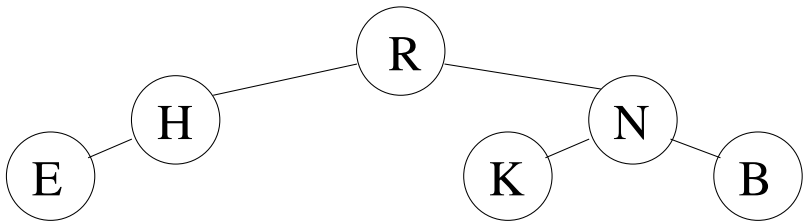
Plusieurs solutions, de la plus coûteuse à la moins coûteuse :

- effectuer systématiquement la fusion dans un tableau temporaire, qu'on recopie ensuite dans L
- même principe mais on alloue le tableau une fois pour toutes au début du tri
  (et donc on le passe en argument)
- éviter les recopies inutiles : selon la profondeur de l'appel récursif, on fusionne L dans M ou bien M dans L
  (là encore, jouer sur les arguments)

Arbre ordonné

Arbre ordonné

Un arbre est ordonné si tout nœud a une clé supérieure ou égale à celles de chacun de ses fils (s'ils existent).



Propriétés :

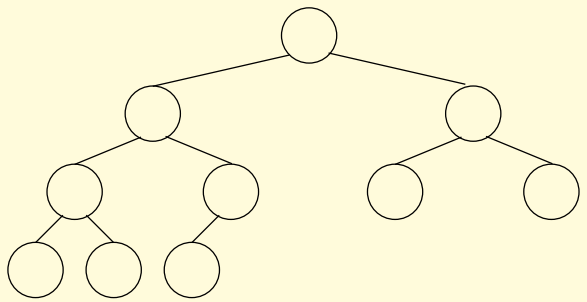
- Les sous-arbres sont eux-mêmes des arbres ordonnés.
- Dans tout chemin de père en fils, les clés sont en ordre décroissant.
- La racine porte la valeur maximale des clés de l'arbre.

Arbre tassé

Arbre tassé

Un arbre binaire de hauteur h est tassé si :

- Tous les nœuds *internes* ont deux fils, sauf éventuellement le **dernier** dans l'ordre par niveaux.



- Les feuilles de niveau h sont « tassées à gauche ».

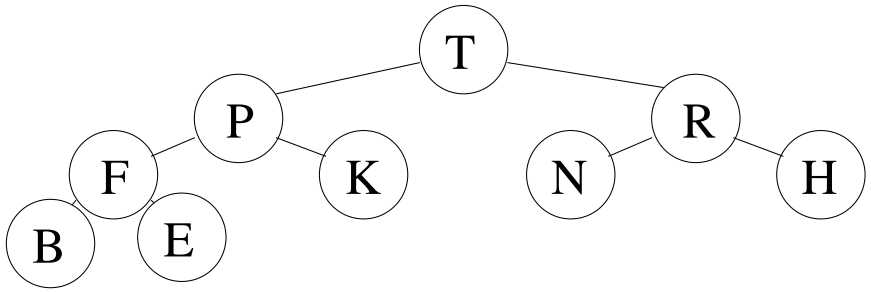
Propriétés :

- Tous les niveaux sont complets, sauf éventuellement le dernier
- Un arbre tassé contient au max. un nœud unaire (si n pair)
- Le nombre total de nœuds est $n = 2^h - 1 + x \quad$ où $\quad 0 < x \leq 2^h$
- D'où $h = \lfloor \log_2 n \rfloor$

Structure de tas

Tas binaire

Un tas (*heap* en anglais) est un arbre binaire **tassé** et **ordonné**.



Sert en particulier à réaliser une File à Priorités efficace :

- nœud de priorité maximale accessible en temps constant
- pas d'information superflue à maintenir

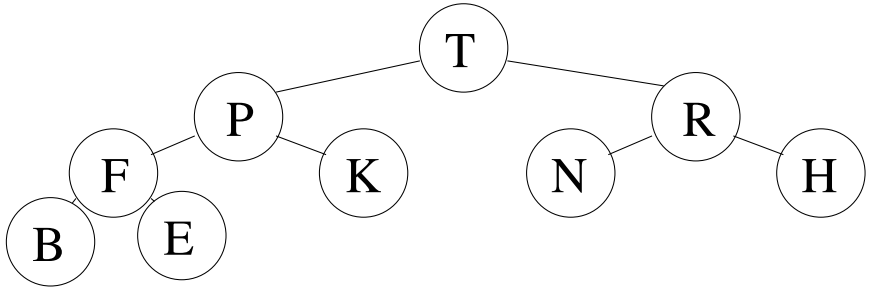
Application « évidente » : **tri par tas** (*heapsort*)

- Insérer les éléments dans un tas, puis les extraire un par un

Remarque

Ne pas confondre avec la zone d'allocation dynamique en mémoire.

Insertion



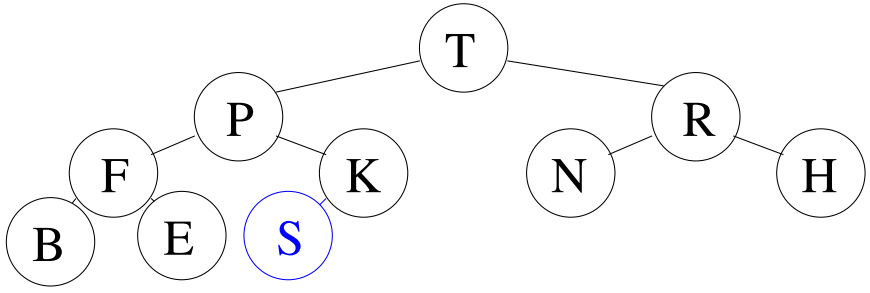
Idée générale : il faut que l'arbre reste **tassé** et **ordonné**.

- **Tassé**. On ne peut ajouter le nouveau nœud que :
  - au dernier niveau, après la dernière feuille ;
  - ou si le dernier niveau est complet, au tout début du prochain niveau.
- **Ordonné**. On procède par échange de clés *sans modifier la structure de l'arbre*, le long du chemin de la nouvelle feuille à la racine.

INSERER(e, t)
Créer une nouvelle feuille n de clé e après la dernière feuille de t
$p = \text{Père}(n)$
while *n n'est pas la racine et clé(p) $<$ clé(n) :*
  Échanger les clés de p et de n
  $n = p$
  $p = \text{Père}(n)$

(Percolation vers le haut)

Insertion



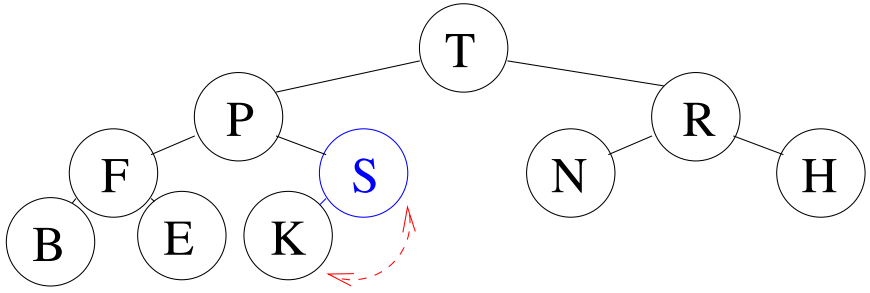
Idée générale : il faut que l'arbre reste **tassé** et **ordonné**.

- **Tassé**. On ne peut ajouter le nouveau nœud que :
  - au dernier niveau, après la dernière feuille ;
  - ou si le dernier niveau est complet, au tout début du prochain niveau.
- **Ordonné**. On procède par échange de clés *sans modifier la structure de l'arbre*, le long du chemin de la nouvelle feuille à la racine.

INSERER(e, t)
Créer une nouvelle feuille n de clé e après la dernière feuille de t
$p = \text{Père}(n)$
while *n n'est pas la racine et clé(p) $<$ clé(n) :*
    Échanger les clés de p et de n
    $n = p$
    $p = \text{Père}(n)$

(Percolation vers le haut)

Insertion



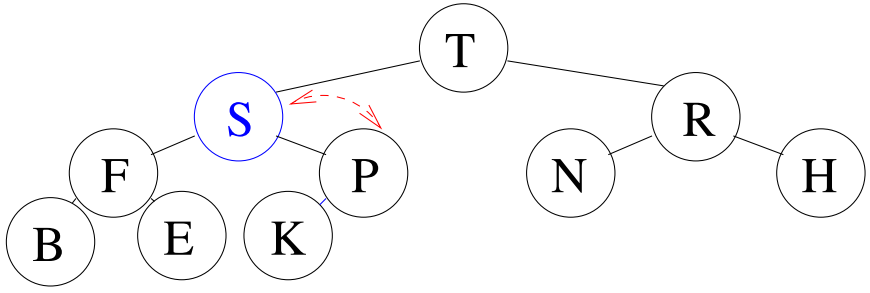
Idée générale : il faut que l'arbre reste **tassé** et **ordonné**.

- **Tassé**. On ne peut ajouter le nouveau nœud que :
  - au dernier niveau, après la dernière feuille ;
  - ou si le dernier niveau est complet, au tout début du prochain niveau.
- **Ordonné**. On procède par échange de clés *sans modifier la structure de l'arbre*, le long du chemin de la nouvelle feuille à la racine.

INSERER(e, t)
Créer une nouvelle feuille n de clé e après la dernière feuille de t
$p = \text{Père}(n)$
while *n n'est pas la racine et clé(p) $<$ clé(n) :*
    Échanger les clés de p et de n
    $n = p$
    $p = \text{Père}(n)$

(Percolation vers le haut)

Insertion



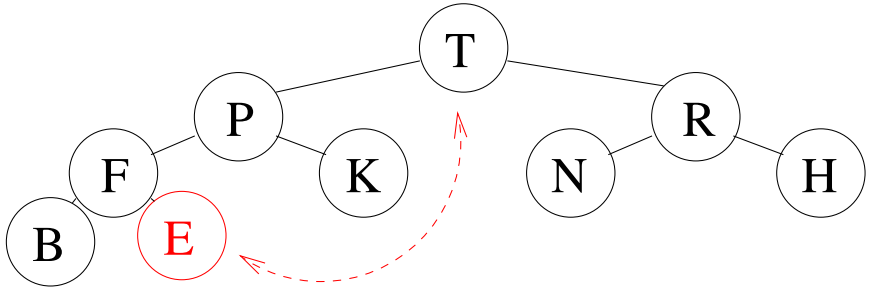
Idée générale : il faut que l'arbre reste **tassé** et **ordonné**.

- **Tassé**. On ne peut ajouter le nouveau nœud que :
  - au dernier niveau, après la dernière feuille ;
  - ou si le dernier niveau est complet, au tout début du prochain niveau.
- **Ordonné**. On procède par échange de clés *sans modifier la structure de l'arbre*, le long du chemin de la nouvelle feuille à la racine.

INSERER(e, t)
Créer une nouvelle feuille n de clé e après la dernière feuille de t
$p = \text{Père}(n)$
while *n n'est pas la racine et clé(p) $<$ clé(n) :*
    Échanger les clés de p et de n
    $n = p$
    $p = \text{Père}(n)$

(Percolation vers le haut)

Extraction du maximum



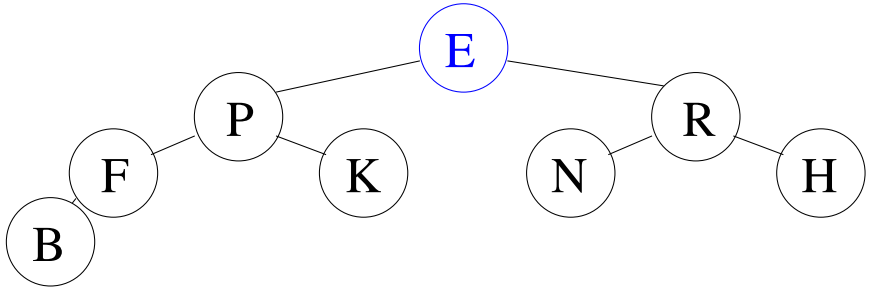
Même principe :

- **Tassé**. On doit supprimer la dernière feuille du dernier niveau. Cependant le maximum est à la racine : on commence par échanger.
- **Ordonné**. On procède par échange de clés *sans modifier la structure de l'arbre*, en descendant cette fois vers les feuilles.

EXTRAIRE_MAX(t)
f = Dernière feuille de t
n = Racine de t
Échanger les clés de f et de n
Supprimer f
while *n n'est pas une feuille et clé(n) < clé d'un fils de n :*
    m = Fils de n de clé maximale
    Échanger les clés de m et de n
    $n = m$

(Percolation vers le bas)

Extraction du maximum



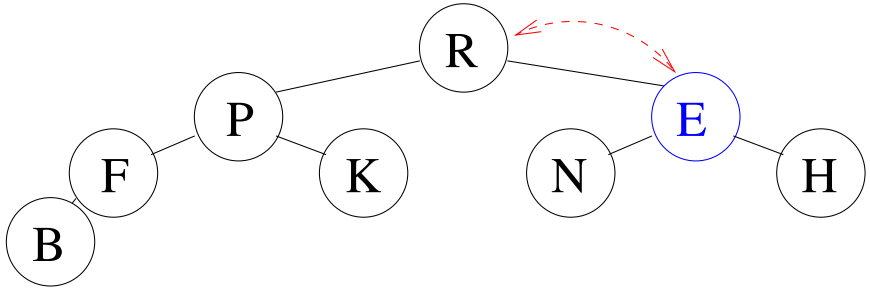
Même principe :

- **Tassé**. On doit supprimer la dernière feuille du dernier niveau. Cependant le maximum est à la racine : on commence par échanger.
- **Ordonné**. On procède par échange de clés *sans modifier la structure de l'arbre*, en descendant cette fois vers les feuilles.

EXTRAIRE_MAX(t)
f = Dernière feuille de t
n = Racine de t
Échanger les clés de f et de n
Supprimer f
while *n n'est pas une feuille et clé(n) < clé d'un fils de n :*
    m = Fils de n de clé maximale
    Échanger les clés de m et de n
    $n = m$

(Percolation vers le bas)

Extraction du maximum



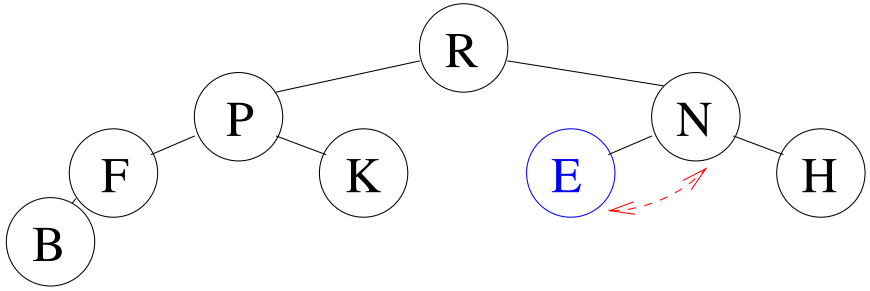
Même principe :

- **Tassé**. On doit supprimer la dernière feuille du dernier niveau. Cependant le maximum est à la racine : on commence par échanger.
- **Ordonné**. On procède par échange de clés *sans modifier la structure de l'arbre*, en descendant cette fois vers les feuilles.

EXTRAIRE_MAX(t)
f = Dernière feuille de t
n = Racine de t
Échanger les clés de f et de n
Supprimer f
while *n n'est pas une feuille et clé(n) < clé d'un fils de n :*
    m = Fils de n de clé maximale
    Échanger les clés de m et de n
    $n = m$

(Percolation vers le bas)

Extraction du maximum



Même principe :

- **Tassé**. On doit supprimer la dernière feuille du dernier niveau. Cependant le maximum est à la racine : on commence par échanger.
- **Ordonné**. On procède par échange de clés *sans modifier la structure de l'arbre*, en descendant cette fois vers les feuilles.

EXTRAIRE_MAX(t)
f = Dernière feuille de t
n = Racine de t
Échanger les clés de f et de n
Supprimer f
while *n n'est pas une feuille et clé(n) < clé d'un fils de n :*
    m = Fils de n de clé maximale
    Échanger les clés de m et de n
    $n = m$

(Percolation vers le bas)

Complexité

Les opérations potentiellement coûteuses sont la comparaison et l'échange de clés.

Les deux opérations **Insérer** et **Extraire_max** sont constituées d'une boucle effectuant à chaque itération :

- une comparaison
- un échange

le long d'un seul chemin : leur coût est majoré par la **hauteur de l'arbre**.

Celui-ci étant tassé :

Complexité de la mise à jour du tas

La complexité des opérations **Insérer** et **Extraire_max** est en $\mathcal{O}(\log_2 n)$.

Implémentation dans une liste

Principe

On place les étiquettes des nœuds dans une liste, dans l'ordre du parcours par niveaux :

- du niveau 0 au niveau h ;
- de gauche à droite dans chaque niveau.

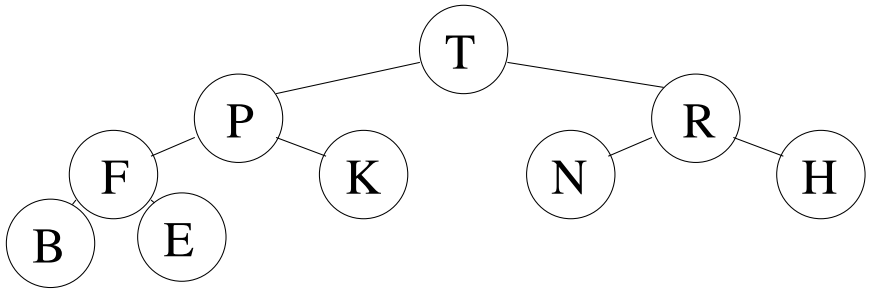
Comme la forme de l'arbre est complètement déterminée par n, une liste représente un unique tas.

L'insertion et la suppression ont toujours lieu en fin de tableau : on profite donc de `pop` et `append`.

Dans un autre langage

On prévoirait un tableau d'une taille suffisante $nmax$, et on mémoriserait le nombre d'éléments effectif dans une variable $n \in [0 \dots nmax]$.

Exemple et précisions



Le tas ci-dessus est représenté par la liste :

T	P	R	F	K	N	H	B	E

On remarque que :

- La racine est à l'indice 0
- Si un nœud est à l'indice i :
  - son fils gauche est à l'indice $2i + 1$
  - son fils droit est à l'indice $2i + 2$
  - son père est à l'indice $(i - 1)//2$
- on peut aussi déterminer si un nœud est ou non une feuille en étudiant son indice

Un exemple

Pour aller plus loin, on peut rassembler les fonctions du tas dans une classe.

Exemple : une classe pour les nombres complexes

```
class Complexe:

    def module(self):
        return math.sqrt( self.re**2 + self.im**2 )

    def argument(self):
        return math.atan( self.im / self.re )
```

Instanciation d'une classe

On construit un objet en appelant la classe comme une fonction :
`c = Complexe()`

- crée un nouvel objet *vide*
- n'alloue pas de mémoire pour le stockage de l'objet (attributs, ...)
- mais, si elle est définie dans la classe, la fonction `__init__` est immédiatement exécutée ; elle joue donc le rôle de *constructeur* de la classe

```
class Complexe:
    def __init__(self, re, im):
        self.re = re
        self.im = im

    def module(self): ...

    def argument(self): ...

c = Complexe(1.,2.)
```

Écriture et appel aux méthodes

- Si une méthode est définie comme `meth(self, arg1, arg2)`
- et si `x` est une instance de la classe,

on appelle cette méthode sous la forme `x.meth(v1, v2)`
ce qui revient à exécuter `meth(x, v1, v2)`.

Par convention, on appelle donc toujours `self` le premier argument de la méthode : c'est l'objet lui-même.

```
c = Complexe(2, 1)
c.module()
c.translater(1, 1)
c.argument()
```