Implémentation de tris classiques

DIU Enseigner l'Informatique au Lycée

E. Jahier, L. Rieg, C. Parent-Vigouroux, B. Wack

UFR IM2AG, Université Grenoble Alpes

juillet 2020

Plan

Drapeau Hollandais

Tri par fusion
- Gestion de la mémoire

Tri par tas
- Une structure de données spécifique
- Opérations

Plan

Drapeau Hollandais

Tri par fusion
- Gestion de la mémoire

Tri par tas
- Une structure de données spécifique
- Opérations

Le drapeau hollandais (1976)

E.W. Dijkstra (1930-2002)

- un des fondateurs de la science informatique
- algorithme de recherche de plus court chemin
- pile de récursivité pour ALGOL-60
- Turing Award 1972

Le drapeau hollandais (1976)

E.W. Dijkstra (1930-2002)

- un des fondateurs de la science informatique
- algorithme de recherche de plus court chemin
- pile de récursivité pour ALGOL-60
- Turing Award 1972

Tableau de *n* éléments, chacun coloré en bleu, blanc ou rouge.

Objectif : réorganiser le tableau pour que :

- les éléments bleus soient sur la partie gauche
- les éléments blancs au centre
- les rouges en fin de tableau

1	2	3	4	5	6	7	8
A	B	C	D	E	F	G	H

→

1	2	3	4	5	6	7	8
H	G	D	B	E	F	C	A

Le drapeau hollandais (1976)

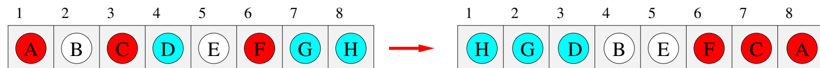
E.W. Dijkstra (1930-2002)

- un des fondateurs de la science informatique
- algorithme de recherche de plus court chemin
- pile de récursivité pour ALGOL-60
- Turing Award 1972

Tableau de n éléments, chacun coloré en bleu, blanc ou rouge.

Objectif : réorganiser le tableau pour que :

- les éléments bleus soient sur la partie gauche
- les éléments blancs au centre
- les rouges en fin de tableau



Contrainte : utiliser un minimum de mémoire supplémentaire (en place)

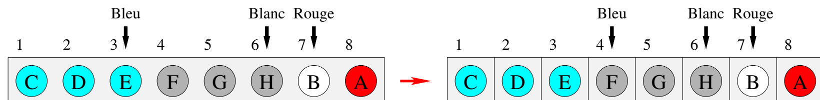
Les trois cas

Trois indices mémorisant où placer le prochain élément de chaque couleur

Les trois cas

Trois indices mémorisant où placer le prochain élément de chaque couleur

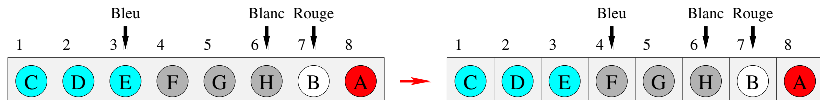
- Cas bleu



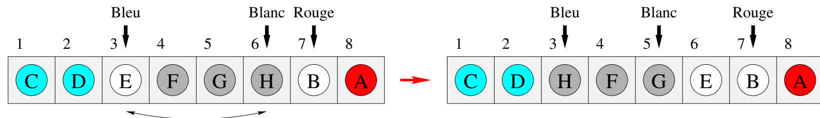
Les trois cas

Trois indices mémorisant où placer le prochain élément de chaque couleur

- Cas bleu



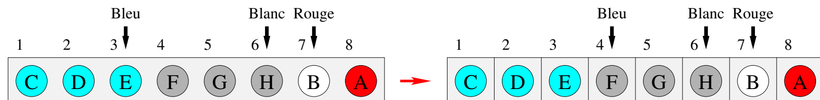
- Cas blanc



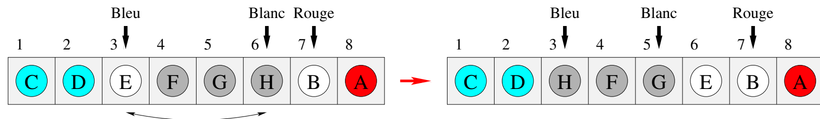
Les trois cas

Trois indices mémorisant où placer le prochain élément de chaque couleur

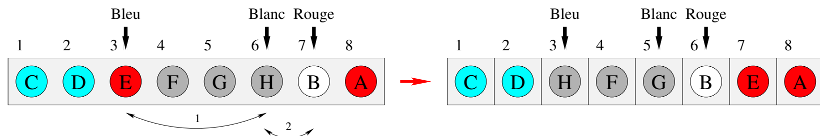
- Cas bleu



- Cas blanc



- Cas rouge



Application au tri

Application au tri rapide

Partition(T, g, d)
Valeur de retour : ?
Effet de bord : Les éléments de T entre les indices :

- g et i sont tous inférieurs à $pivot = T[g]$.
- i et d sont tous supérieurs à $pivot$.

Application au tri rapide

Partition(T, g, d)
Valeur de retour : ?
Effet de bord : Les éléments de T entre les indices :

- g et i sont tous inférieurs à $pivot = T[g]$.
- i et d sont tous supérieurs à $pivot$.

$i = g$
$pivot = T[g]$
while
  if $T[i+1] \leq pivot$
  else

return i

Application au tri rapide

Partition(T, g, d)
Valeur de retour : ?
Effet de bord : Les éléments de T entre les indices :

- g et i sont tous inférieurs à $pivot = T[g]$.
- i et d sont tous supérieurs à $pivot$.

```
i = g
pivot = T[g]
while
    if T[i + 1] ≤ pivot
        i = i + 1
    else

return i
```

Application au tri rapide

Partition(T, g, d)
Valeur de retour : ?
Effet de bord : Les éléments de T entre les indices :

- g et i sont tous inférieurs à $pivot = T[g]$.
- i et d sont tous supérieurs à $pivot$.

```
i = g
pivot = T[g]
while
    if  T[i+1] ≤ pivot
        i = i + 1
    else
        Échanger T[d] et T[i+1]
        d = d − 1

return i
```

Application au tri rapide

Partition(T, g, d)
Valeur de retour : ?
Effet de bord : Les éléments de T entre les indices :

- g et i sont tous inférieurs à $pivot = T[g]$.
- i et d sont tous supérieurs à $pivot$.

```
i = g
pivot = T[g]
while i < d
    if T[i+1] ≤ pivot
        i = i + 1
    else
        Échanger T[d] et T[i+1]
        d = d - 1

return i
```

Application au tri rapide

Partition(T, g, d)
Valeur de retour : ?
Effet de bord : Les éléments de T entre les indices :

- g et i sont tous inférieurs à $pivot = T[g]$.
- i et d sont tous supérieurs à $pivot$.

```
i = g
pivot = T[g]
while i < d
    if  T[i + 1] ≤ pivot
        i = i + 1
    else
        Échanger T[d] et T[i + 1]
        d = d − 1
Échanger T[g] et T[i]
return i
```

Plan

Drapeau Hollandais

Tri par fusion
- Gestion de la mémoire

Tri par tas
- Une structure de données spécifique
- Opérations

Rappel : principe du tri par fusion

```python
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))
```

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)

Plan

Drapeau Hollandais

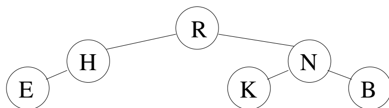
Tri par fusion
- Gestion de la mémoire

Tri par tas
- Une structure de données spécifique
- Opérations

Arbre ordonné

Arbre ordonné

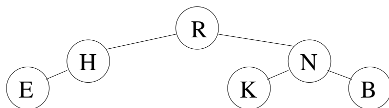
Un arbre est ordonné si tout nœud a une clé supérieure ou égale à celles de chacun de ses fils (s'ils existent).



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 ».

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 ».

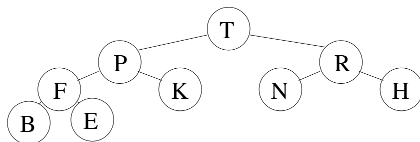
Propriétés :

- 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 $n = 2^h - 1 + x$ où $0 < x \leq 2^h$
- D'où $h = \lfloor \log_2 n \rfloor$

Structure de tas

Tas binaire

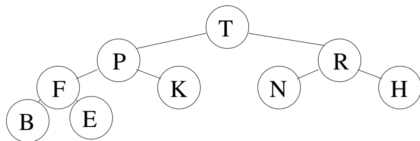
Un tas (*heap* en anglais) est un arbre binaire **tassé** et **ordonné**.



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.

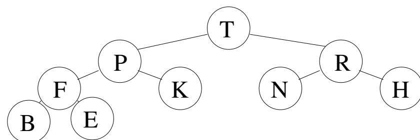
Plan

Drapeau Hollandais

Tri par fusion
Gestion de la mémoire

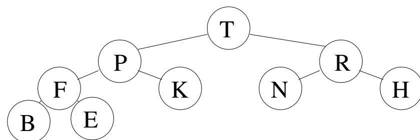
Tri par tas
Une structure de données spécifique
Opérations

Insertion



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

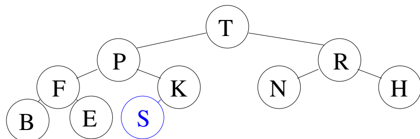
Insertion



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

- **Tassé**. On ne peut ajouter le nouveau nœud que :

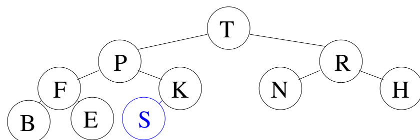
Insertion



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.

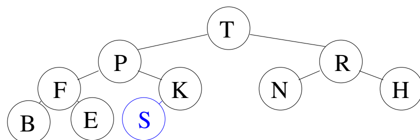
Insertion



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**.

Insertion

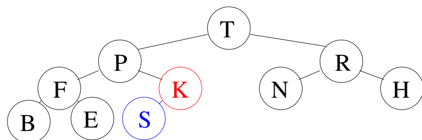


Idée générale : il faut que l'arbre reste **tassé** et **ordonné**.

- **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 = 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 = Père(n)

Insertion

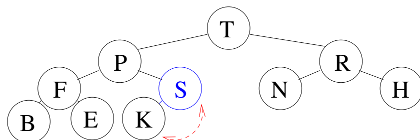


Idée générale : il faut que l'arbre reste **tassé** et **ordonné**.

- **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
$p =$ Père(n)
while *n n'est pas la racine et clé(p) $<$ clé(n) :*
    Échanger les clés de p et de n
    $n = p$
    $p =$ Père(n)

Insertion

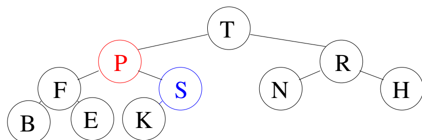


Idée générale : il faut que l'arbre reste **tassé** et **ordonné**.

- **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
  $n = p$
  $p = \text{Père}(n)$

Insertion

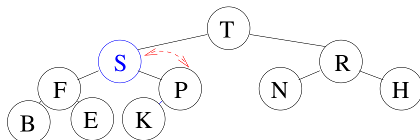


Idée générale : il faut que l'arbre reste **tassé** et **ordonné**.

- **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 = 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 = Père(n)

Insertion

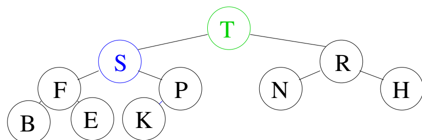


Idée générale : il faut que l'arbre reste **tassé** et **ordonné**.

- **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 = 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 = Père(n)

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)

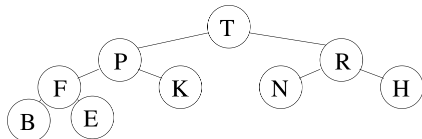
Créer une nouvelle feuille n de clé e après la dernière feuille de t

$p = \text{Père}(n)$

while *n n'est pas la racine et clé(p) < clé(n) :*
- Échanger les clés de p et de n
- $n = p$
- $p = \text{Père}(n)$

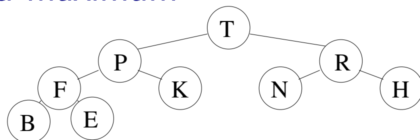
(Percolation vers le haut)

Extraction du maximum



Même principe :

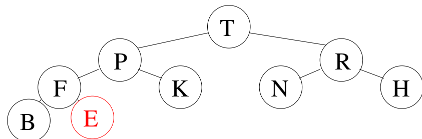
Extraction du maximum



Même principe :

- **Tassé**. On doit supprimer

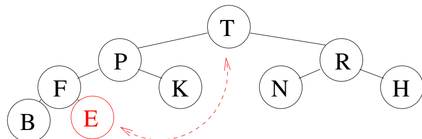
Extraction du maximum



Même principe :

- **Tassé**. On doit supprimer la dernière feuille du dernier niveau.

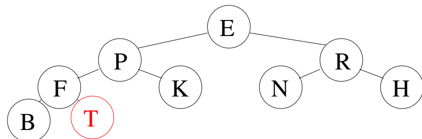
Extraction du maximum



Même principe :

- **Tassé**. On doit supprimer la dernière feuille du dernier niveau. Cependant le maximum est à la racine : on commence par échanger.

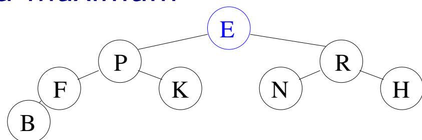
Extraction du maximum



Même principe :

- **Tassé**. On doit supprimer la dernière feuille du dernier niveau. Cependant le maximum est à la racine : on commence par échanger.

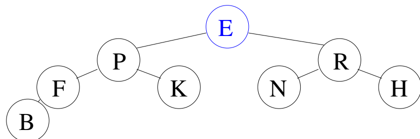
Extraction du maximum



Même principe :

- **Tassé**. On doit supprimer la dernière feuille du dernier niveau. Cependant le maximum est à la racine : on commence par échanger.
- **Ordonné**.

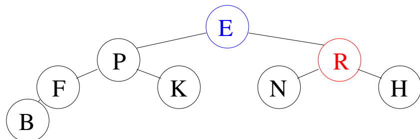
Extraction du maximum



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.

Extraction du maximum

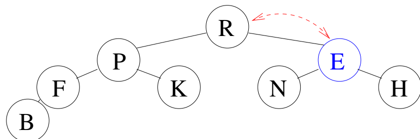


Même principe :

- **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
  $n = m$

Extraction du maximum

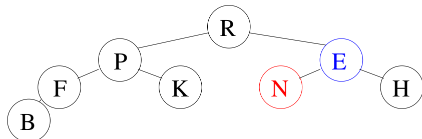


Même principe :

- **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
  $n = m$

Extraction du maximum

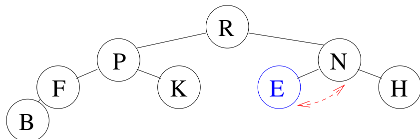


Même principe :

- **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
  $n = m$

Extraction du maximum

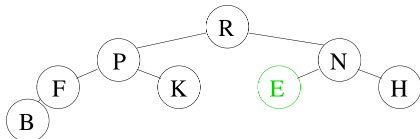


Même principe :

- **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
  $n = m$

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)

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.

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.

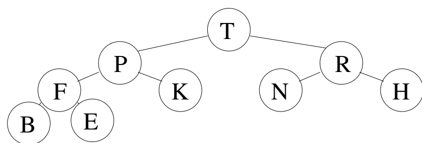
Comme la forme de l'arbre est complètement déterminée par n,
une liste représente un unique tas.

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

Dans un autre langage

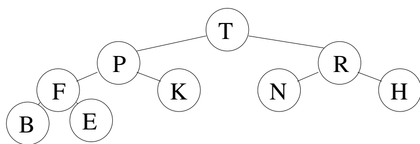
On prévoirait un tableau d'une taille suffisante $nmax$, et on mémoriserait le nombre d'éléments effectif dans une variable $n \in [0 \dots nmax]$.

Exemple et précisions



Le tas ci-dessus est représenté par la liste :

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 …?
  - son fils droit est à l'indice …?
  - son père est à l'indice …?
- on peut aussi déterminer si un nœud est ou non une feuille en étudiant son indice