

IN 101 - Cours 12

9 décembre 2011



présenté par
Matthieu Finiasz

méthode	stockage	recherche	insertion	modif.	suppr.
adressage direct	$\Theta(m)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
recherche séquentielle	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
recherche dichotomique	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
tables de hachage	$\Theta(k + n)$	$\Theta(\frac{n}{k})$	$\Theta(1)$	$\Theta(\frac{n}{k})$	$\Theta(\frac{n}{k})$
avec $k \simeq n$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
ABR	$\Theta(n)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

- ✗ On aimerait garantir la complexité des ABR dans le pire cas
 - ✗ il faut des arbres tels que $h = \Theta(\log n)$ dans tous les cas.

Un problème concret Indexation de bases de données

- ✗ Une base de donnée peut se voir comme une grande table avec plusieurs colonnes :

Nom	Prénom	Promo	Filière	Prépa	Téléphone	Sexe	Naissance	login
AMALRIC	Marie	2013	CC_MP	Paris : Lycée Louis-le-Grand	0675xxxx09	F	23/07/1989	amalric
BALZER	Anne-Sophie	2013	CC_MP	Toulouse : Lycée Bellevue	0645xxxx68	F	30/06/1990	balzer
BARHOUMI	Rania	2013	CC_MP	Tunisie : IPEST	0640xxxx30	F	10/07/1989	barhoumi
BASSIR KAZERUNI	Neda	2013	CC_PSI	Paris : Lycée Louis-le-Grand	0619xxxx65	F	31/05/1990	bassir
BENABDELJELIL	Kelthoum	2013	CC_MP		0636xxxx75	F	08/06/1990	benabdeljelil
DZIEDZIC	Amandine	2013	CC_PSI	Grenoble : Lycée Champollion	0670xxxx10	F	13/01/1989	dziedzic
FERON	Amélie	2013	CC_PC	Lycée Faidherbe, Lille	0678xxxx21	F	17/01/1990	feron
GARCIA	Maylis	2013	CC_PSI	Nantes : Lycée Clémenceau	0671xxxx16	F	22/09/1990	mgarcia
HEBERT	Odiè	2013	CC_PC	Lyon : Lycée du Parc	0662xxxx97	F	16/06/1989	hebert
LEDUCQ	Camille	2013	CC_MP	Paris : Lycée Stanislas	0621xxxx59	F	10/12/1990	leducq
MADELENAT	Jill	2013	CC_PSI	Marseille : Lycée Thiers	0679xxxx69	F	10/11/1990	madelenat
MENISSIER	Laure	2013	CC_PSI	Lyon : Lycée du Parc	0673xxxx45	F	26/08/1990	menissier
PELLETIER	Fanny	2013	CC_PC	Grenoble : Lycée Champollion	0677xxxx77	F	06/12/1991	pelletier
PESUDO	Laure	2013	CC_PC	Toulouse : Lycée de Fermat	0626xxxx91	F	27/11/1989	pesudo
PETITET	Marie	2013	CC_PSI	Lyon : Lycée du Parc	0680xxxx43	F	23/01/1989	petitet
PORE	Anne-Sophie	2013	CC_PSI	Paris : Lycée Saint-Louis	0631xxxx77	F	31/12/1990	pore
PUREN	Lucie	2013	CC_MP		0683xxxx14	F	16/05/1990	puren
REGNAUT	Anne-Charlotte	2013	CC_PSI	Lyon : Lycée du Parc	0643xxxx76	F	16/05/1991	regnaut
SOURDIN	Charlotte	2013	CC_MP		0679xxxx33	F	23/08/1989	sourdin
ZAIEM	Yasmine	2013	CC_MP	Paris : Lycée Louis-le-Grand	0667xxxx68	F	28/04/1989	zaiem

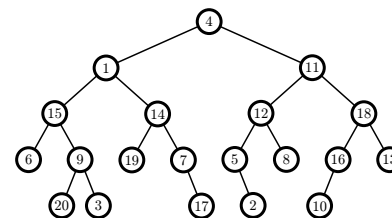
- ✗ Par rapport à un dictionnaire, on veut faire des recherches sur différentes clefs :
 - Nom, prénom, date de naissance...

Un problème concret Indexation de bases de données

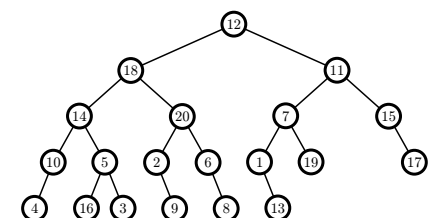
- ✗ On attribue à chaque ligne un identifiant unique : la clef primaire
 - ✗ pour chaque colonne on construit un index
 - un ABR équilibré triant les clefs primaires.

Clef	Nom	Prénom	Promo	Filière	Prépa	Téléphone	Sexe	Naissance	login
1	AMALRIC	Marie	2013	CC_MP	Paris : Lycée Louis-le-Grand	0675xxxx09	F	23/07/1989	amalric
2	BALZER	Anne-Sophie	2013	CC_MP	Toulouse : Lycée Bellevue	0645xxxx68	F	30/06/1990	balzer
3	BARHOUMI	Rania	2013	CC_MP	Tunisie : IPEST	0640xxxx30	F	10/07/1989	barhoumi
4	BASSIR KAZERUNI	Neda	2013	CC_PSI	Paris : Lycée Louis-le-Grand	0619xxxx65	F	31/05/1990	bassir
5	BENABDELJELIL	Kelthoum	2013	CC_MP		0636xxxx75	F	08/06/1990	benabdeljelil
6	DZIEDZIC	Amandine	2013	CC_PSI	Grenoble : Lycée Champollion	0670xxxx10	F	13/01/1989	dziedzic

Date de naissance



Téléphone



Le tas

Un premier exemple d'arbre équilibré

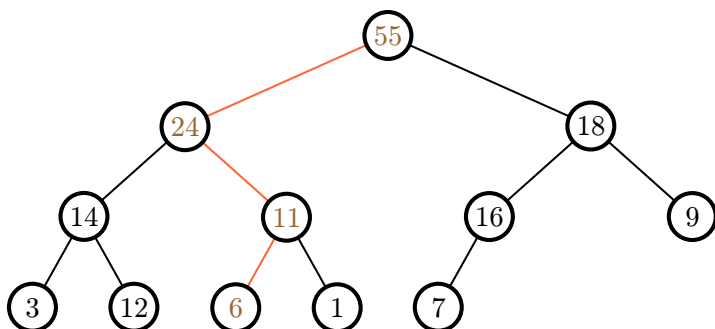
IN101 - 2011-2012

Définition

- × Un tas est un arbre binaire ordonné, mais pas un ABR.
 - × C'est un arbre binaire complet :
 - tous les niveaux (sauf le dernier) sont remplis,
 - sur le dernier, tous les nœuds sont à gauche.
 - × La clef d'un nœud est supérieure aux clefs de tous ses descendants
 - la clef la plus grande est à la racine.
- × Permet de gérer une file de priorité :
 - × chaque élément a une priorité (sa clef)
 - × on veut pouvoir efficacement :
 - insérer un élément,
 - rechercher l'élément de priorité maximale → la racine,
 - supprimer l'élément de priorité maximale.

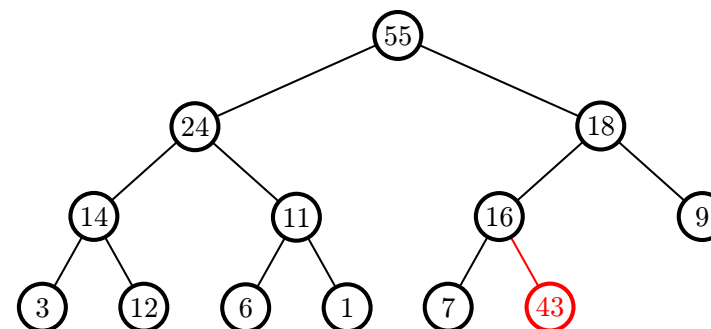
Exemple de tas

- × Les clefs sont en ordre décroissant au sein d'une même branche
 - × aucune relation entre des nœuds de branches différentes.
- × Les nœuds sont tassés à gauche.



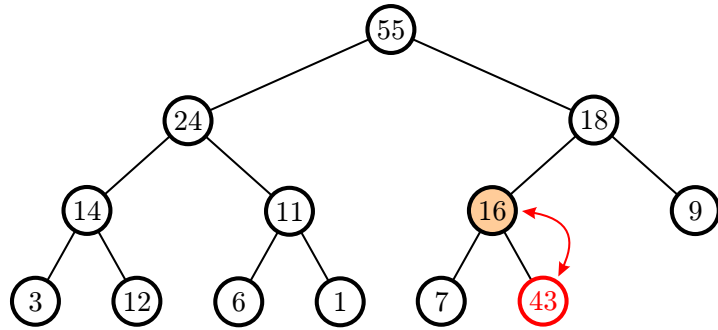
Insertion d'un élément

- × Il faut conserver les deux propriétés du tas :
 - × la complétude,
 - × l'ordre des clefs.
- × On insère à la première place libre.



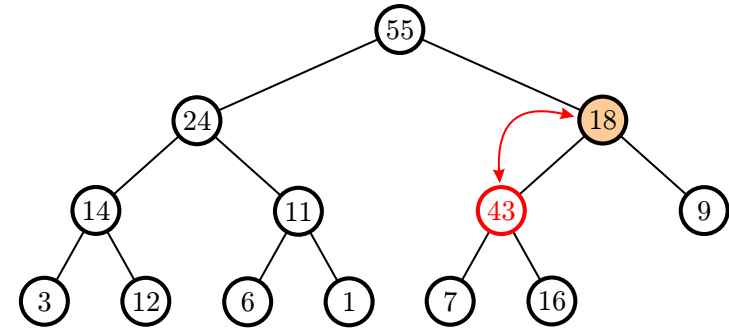
Insertion d'un élément

- ✖ Il faut conserver les deux propriétés du tas :
 - ✖ la complétude,
 - ✖ l'ordre des clefs.
- ✖ On fait remonter le nœud.



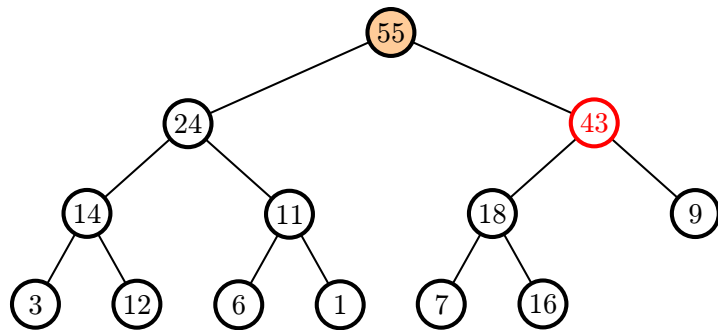
Insertion d'un élément

- ✖ Il faut conserver les deux propriétés du tas :
 - ✖ la complétude,
 - ✖ l'ordre des clefs.
- ✖ On fait remonter le nœud.



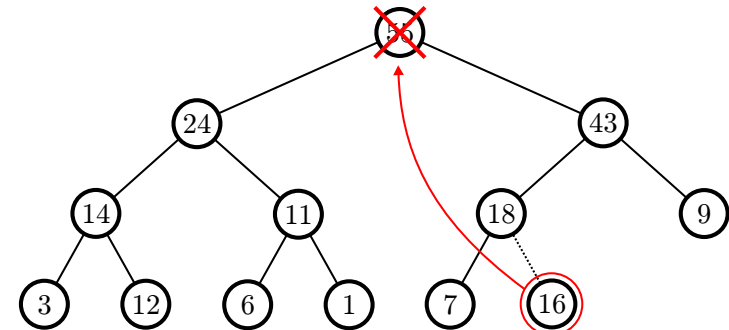
Insertion d'un élément

- ✖ Il faut conserver les deux propriétés du tas :
 - ✖ la complétude,
 - ✖ l'ordre des clefs.
- ✖ On fait remonter le nœud → jusqu'à ce que l'ordre soit bon.



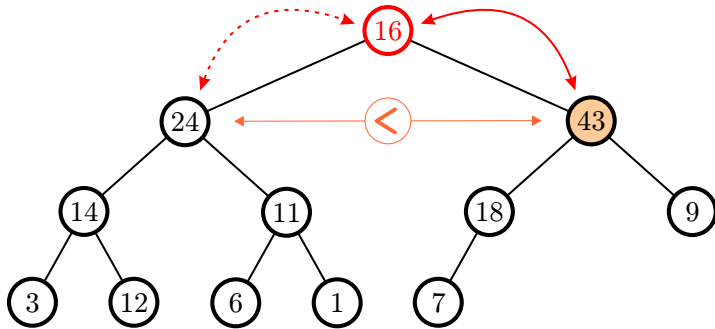
Suppression de la racine

- ✖ On supprime toujours la racine :
 - ✖ conserver un arbre complet.



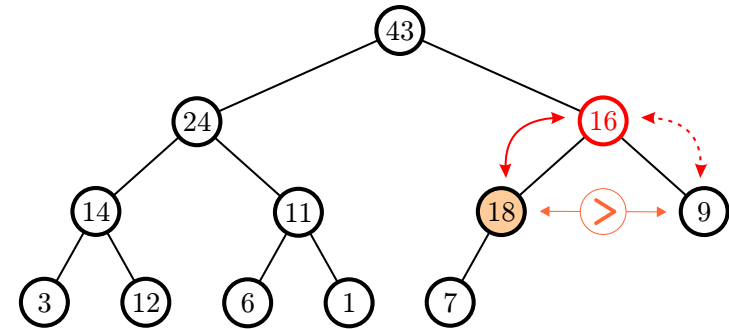
Suppression de la racine

- × On supprime toujours la racine :
 - × conserver un arbre complet,
 - × rétablir l'ordre
- on échange avec le fils **le plus grand**.



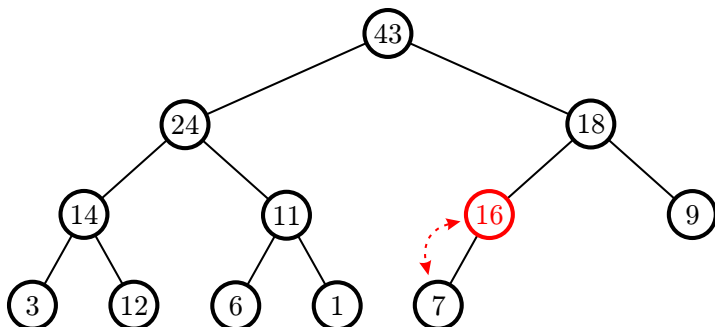
Suppression de la racine

- × On supprime toujours la racine :
 - × conserver un arbre complet,
 - × rétablir l'ordre
- on échange avec le fils **le plus grand**.



Suppression de la racine

- × On supprime toujours la racine :
 - × conserver un arbre complet,
 - × rétablir l'ordre
- on échange avec le fils **le plus grand**.



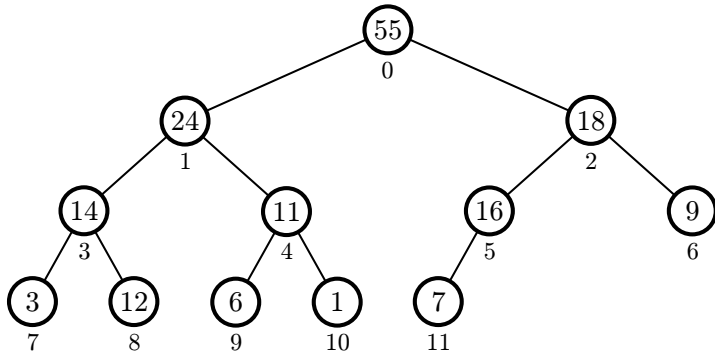
Complexité des opérations

- × Avec les algorithmes précédents on obtient les complexités suivantes :
 - × recherche du maximum : $\Theta(1)$
 - × insertion : $\Theta(\log n)$
 - × suppression : $\Theta(\log n)$
- × Ce sont des complexités **dans le pire cas**.
- × On peut implémenter cela avec un arbre binaire :
 - × il faut ajouter à chaque nœud un lien vers son père,
 - × il faut une façon simple d'accéder à la première place libre et à la dernière place occupée :
 - garder un pointeur sur le dernier élément.
- × Ça marche, mais c'est un peu lourd...

Implémentation d'un tas à l'aide d'un tableau

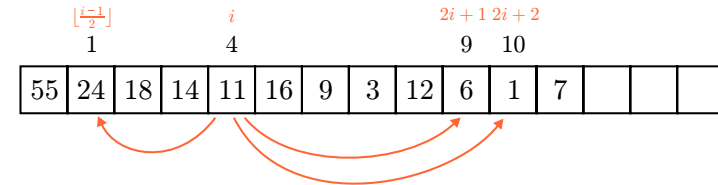
- ✘ En général, on utilise un simple tableau :
 - ✘ les nœuds sont rangés par niveaux,
 - ✘ comme l'arbre est complet, les nœuds sont au début du tableau.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
55	24	18	14	11	16	9	3	12	6	1	7			



Implémentation d'un tas à l'aide d'un tableau

- ✘ En général, on utilise un simple tableau :
 - ✘ les nœuds sont rangés par niveaux,
 - ✘ comme l'arbre est complet, les nœuds sont au début du tableau.
- ⚠ Cela introduit une limite au nombre d'éléments dans le tas.
- ✘ En revanche, on accède facilement à tous les nœuds sans pointeurs :
 - ✘ la racine est en 0,
 - ✘ le **fil gauche** du nœud i en $2i + 1$,
 - ✘ le **fil droit** du nœud i en $2i + 2$,
 - ✘ le **père** du nœud i en $\lfloor \frac{i-1}{2} \rfloor$.



Implémentation d'un tas à l'aide d'un tableau

- ✘ Utiliser un tableau ne modifie pas les complexités précédentes :
 - ✘ on ne fait qu'échanger des éléments du tas
 - échanger des éléments du tableau, pas de décalages,
 - ✘ accéder au fils ou au père se fait toujours en temps constant,
 - ✘ on gagne en revanche un **accès direct au dernier élément**
 - il suffit de savoir le nombre d'éléments présents dans le tas.

```

1 typedef struct {
2     int max;      /* capacité totale du tas      */
3     int n;       /* nombre de noeuds dans le tas */
4     int* tab;    /* tableau des noeuds           */
5 } heap;
    
```

(voir TD 12)

Tri par tas Heapsort

- ✘ Insérer n éléments dans un tas coûte $\Theta(n \log n)$.
- ✘ Les extraire en **ordre décroissant** coûte aussi $\Theta(n \log n)$.
- ✘ On peut trier n éléments en $\Theta(n \log n)$ en moyenne, comme dans le pire cas.
 - ✘ comportement similaire au tri fusion,
 - ✘ en revanche, on peut implémenter ce tri **en place**.
 - en pratique, moins rapide que le tri rapide (en moyenne).

Exemple d'application

Sélection des 10 plus grands éléments

- ✘ Un programme reçoit une longue série d'éléments :
 - ✘ il ne doit conserver que les 10 plus grands parmi ces éléments,
 - ✘ il a une mémoire très limitée
 → ne peut pas stocker tout ce qui arrive.

→ Comment faire ?

Exemple d'application

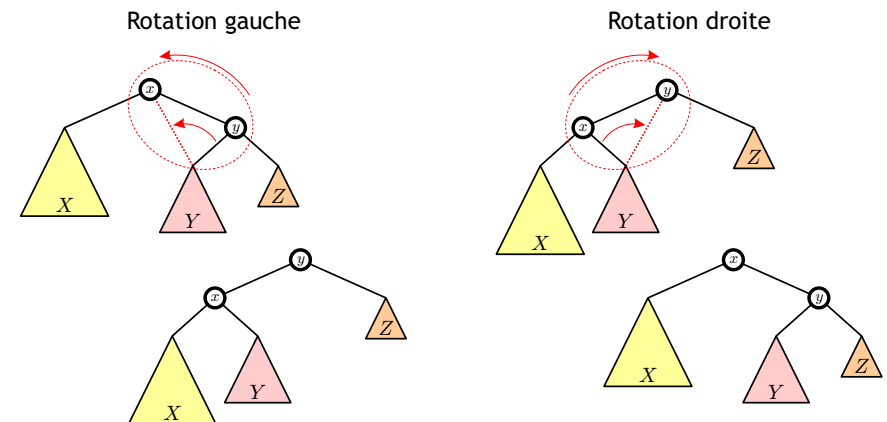
Sélection des 10 plus grands éléments

- ✘ Un programme reçoit une longue série d'éléments :
 - ✘ il ne doit conserver que les 10 plus grands parmi ces éléments,
 - ✘ il a une mémoire très limitée
 → ne peut pas stocker tout ce qui arrive.
- ✘ Il suffit de garder en mémoire un tas contenant en permanence les 10 plus grands éléments reçus :
 - ✘ on insère les 10 premiers éléments reçus,
 - ✘ chaque nouvel élément est comparé au plus petit
 - il faut un tas ayant le **minimum** à la racine,
 - ✘ si le nouvel élément est plus grand :
 - ✘ on enlève la racine,
 - ✘ on insère le nouvel élément.
- ✘ Dans le pire cas, on fait $n \log 10$ comparaisons.

Rééquilibrage d'arbres

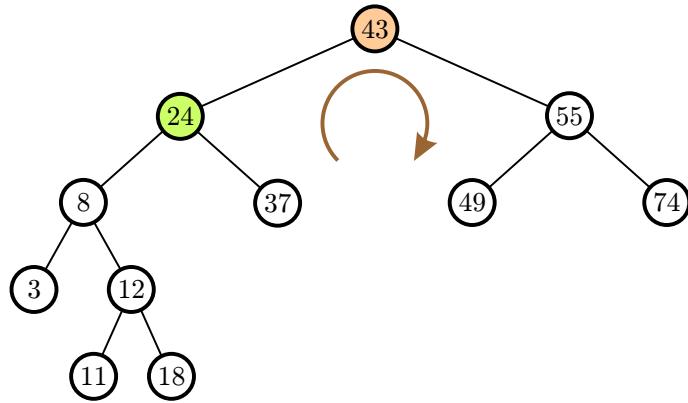
Rotations

- ✘ Une **rotation** permet de rééquilibrer **localement** un arbre.
 - ✘ préserve l'ordre **infixe** des clefs → préserve la structure d'ABR,
 - ✘ se réalise en **temps constant**.



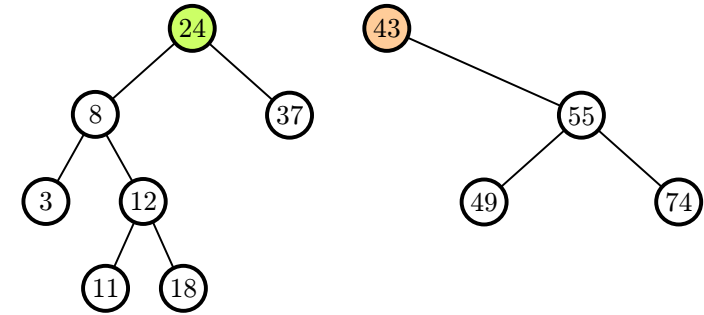
Exemple de rotation à droite

- ✖ On part d'un arbre déséquilibré :
 - ✖ le sous-arbre gauche est plus haut que le droit,
 - rotation à droite en 43.



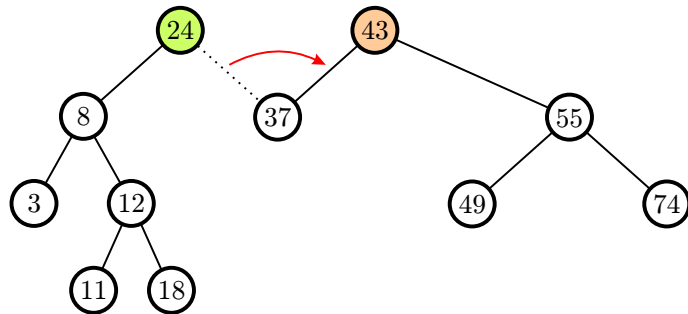
Exemple de rotation à droite

- ✖ La racine 24 du sous-arbre gauche sera la nouvelle racine.



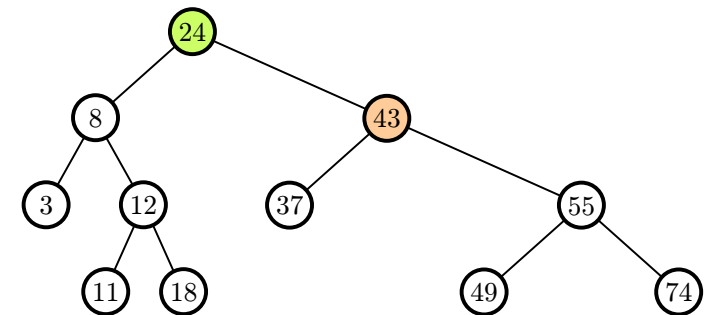
Exemple de rotation à droite

- ✖ Le sous-arbre droit de 24 devient fils gauche de 43.



Exemple de rotation à droite

- ✖ L'ancienne racine 43 devient fils droit de la nouvelle racine 24.
 - ✖ l'arbre est mieux équilibré qu'avant !
 - ✖ l'ordre ABR a été préservé (mouvements "verticaux" seulement).



Arbres équilibrés

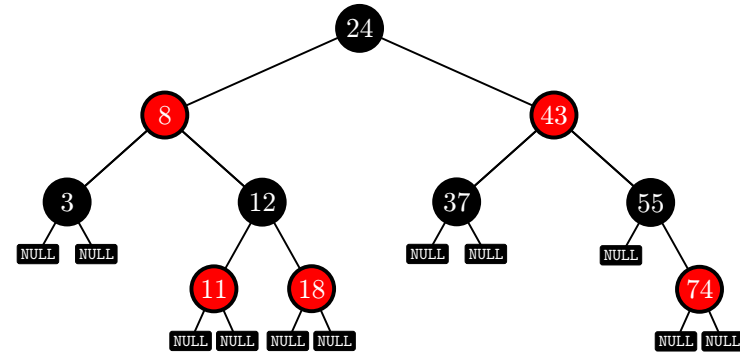
Les familles les plus connues

- ✖ Un arbre est **équilibré** si toutes ses feuilles sont situées à peu près à la même hauteur,
 - ✖ cela implique $h = \Theta(\log n)$.
- ✖ Plusieurs type d'ABR équilibrés existent, chacun avec des propriétés différentes :
 - ✖ arbres bicolores (ou arbres rouge/noir),
 - ✖ arbres 2-3,
 - ✖ arbres AVL...

Arbres rouge/noir

[Bayer – 1972]

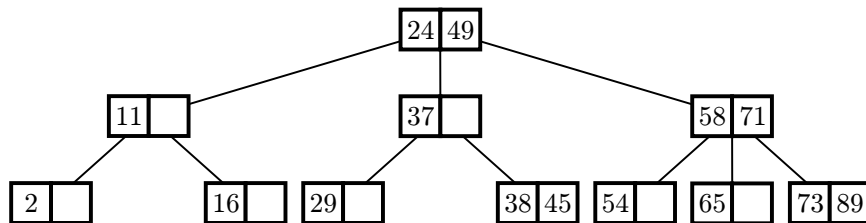
- ✖ Chaque nœud est soit rouge, soit noir :
 - ✖ la racine et les feuilles (vides) sont noires,
 - ✖ un nœud rouge n'a que des fils noirs,
 - ✖ même nombre de nœuds noirs entre la racine et chaque feuille.



Arbres 2-3

[Bayer, Creight – 1972]

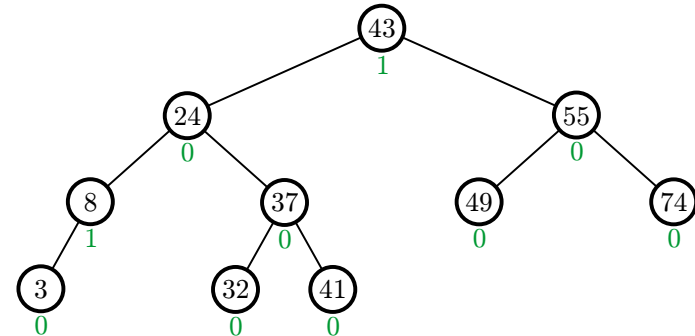
- ✖ Un nœud peut contenir une ou deux valeurs et a 2 ou 3 fils
 - ✖ les feuilles sont toutes à la même hauteur,
 - ✖ se généralise en B-arbres avec de L à U fils et L-1 à U-1 valeurs.
- ✖ Un peu compliqués à utiliser
(cf. <http://people.ksp.sk/~kuko/bak/index.html>)



Arbres AVL

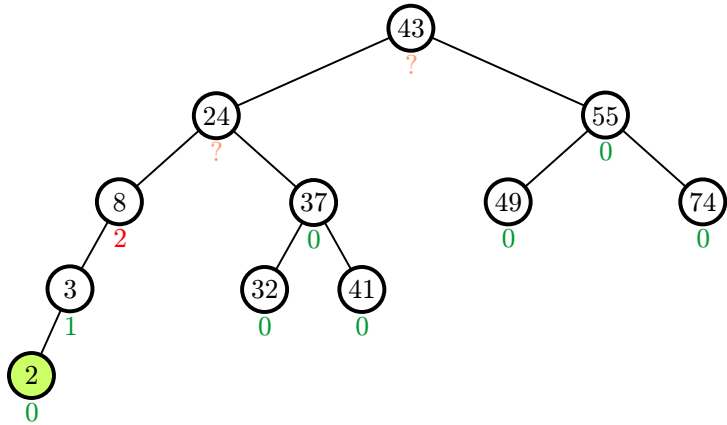
[Adelson, Velskii et Landis – 1962]

- ✖ ABR avec une **marque d'équilibrage** δ pour chaque nœud :
 - $\delta = h(\text{fils gauche}) - h(\text{fils droit})$ et $|\delta| \leq 1$.
- ✖ Recherche, insertion, suppression : comme dans un ABR standard
 - mise à jour de δ et rééquilibrage si nécessaire.



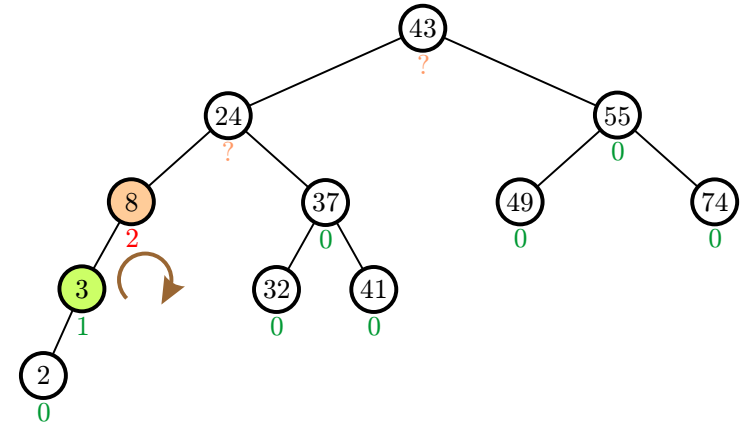
Insertion dans un AVL

✘ On insère le nœud 2 → déséquilibre à gauche.



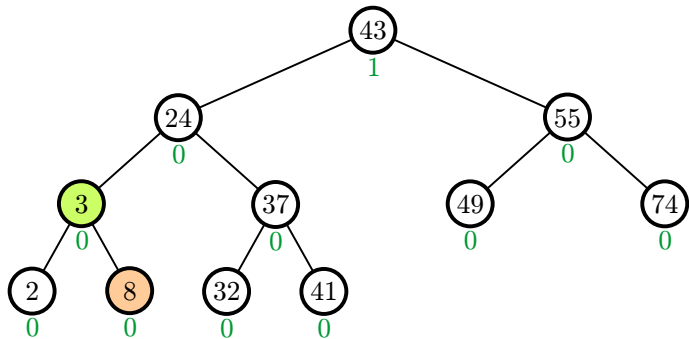
Insertion dans un AVL

✘ On insère le nœud 2 → déséquilibre à gauche
→ une rotation à droite permet de rétablir la propriété AVL.



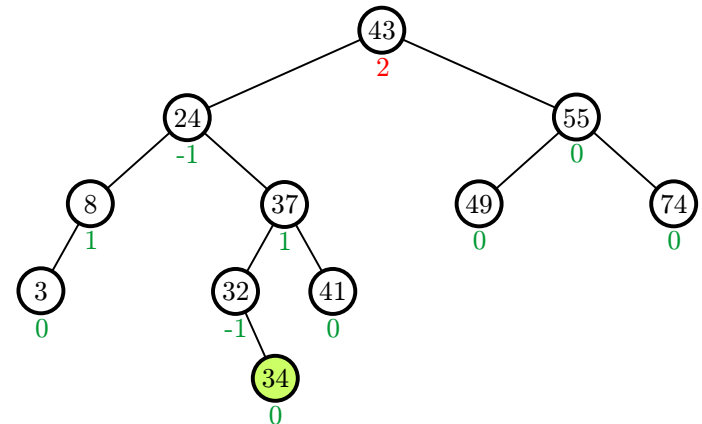
Insertion dans un AVL

✘ On insère le nœud 2 → déséquilibre à gauche
→ une rotation à droite permet de rétablir la propriété AVL.



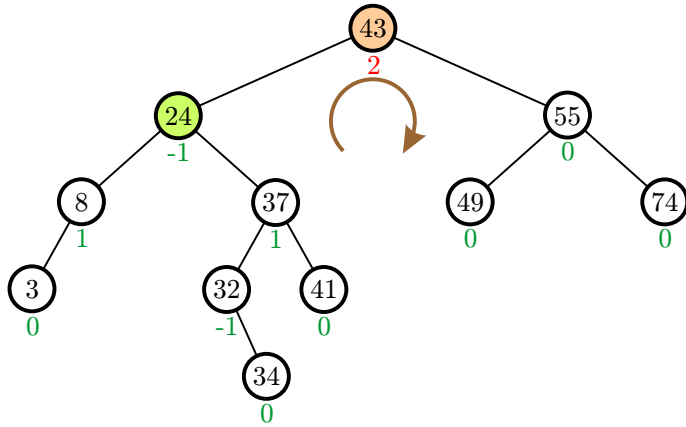
Insertion dans un AVL

✘ Si à la place on insère 34 → encore un déséquilibre à gauche.



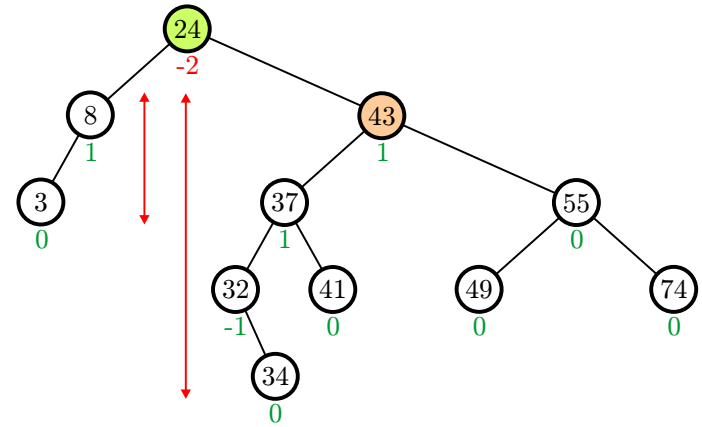
Insertion dans un AVL

- ✘ Si à la place on insère 34 → encore un déséquilibre à gauche
→ une rotation à droite **ne suffit pas** à rétablir la propriété AVL.



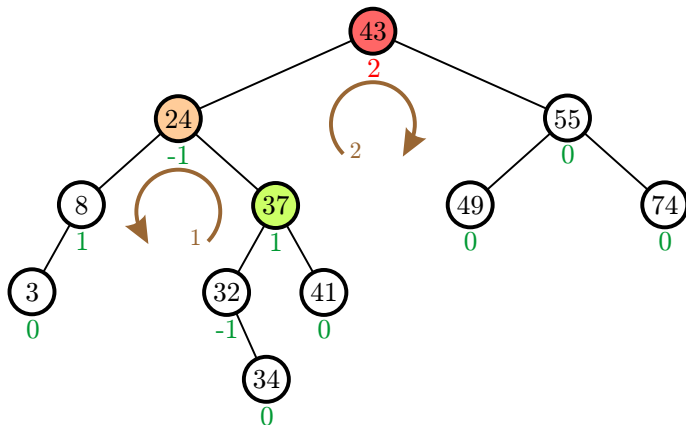
Insertion dans un AVL

- ✘ Si à la place on insère 34 → encore un déséquilibre à gauche
→ une rotation à droite **ne suffit pas** à rétablir la propriété AVL.



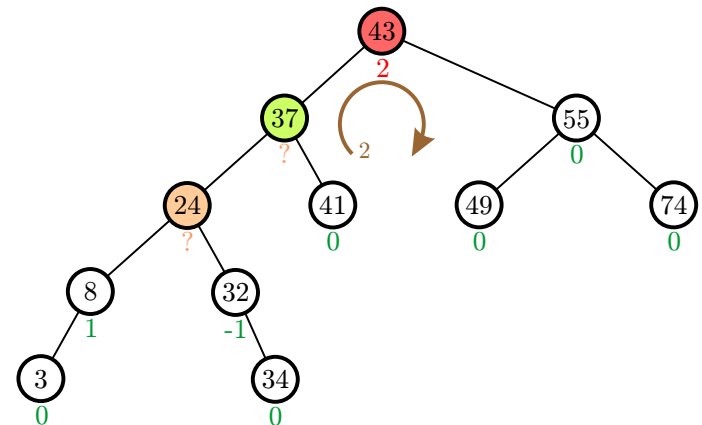
Insertion dans un AVL

- ✘ Si à la place on insère 34 → encore un déséquilibre à gauche
→ on doit effectuer une **double rotation**.



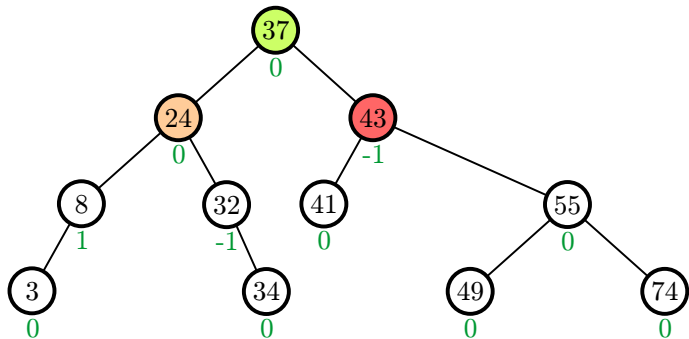
Insertion dans un AVL

- ✘ Si à la place on insère 34 → encore un déséquilibre à gauche
→ on doit effectuer une **double rotation**.



Insertion dans un AVL

- ✖ Si à la place on insère 34 → encore un déséquilibre à gauche
→ on doit effectuer une **double rotation**.

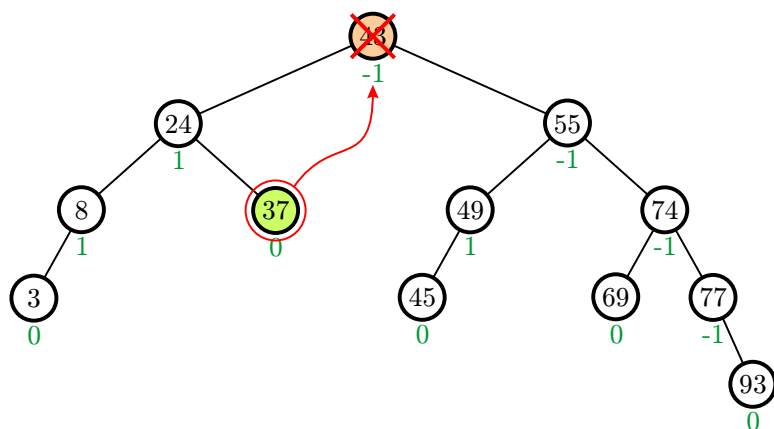


Implémentation d'un AVL

- ✖ Pour implémenter un AVL, il suffit de stocker dans chaque nœud la hauteur du sous-arbre dont il est racine.
 - ✖ lors d'une insertion/suppression on remet à jour les hauteurs : sur une seule branche → complexité en $\Theta(\log n)$ quand même.
- ✖ Ensuite, on parcourt la branche pour rechercher les déséquilibres :
 - ✖ insertion : une seule (double) rotation suffit,
 - ✖ suppression : jusqu'à h (double) rotations nécessaires.
- ✖ Les complexités restent les mêmes que pour un ABR standard :
→ tout coûte $\Theta(h)$, mais $h = \Theta(\log n)$.

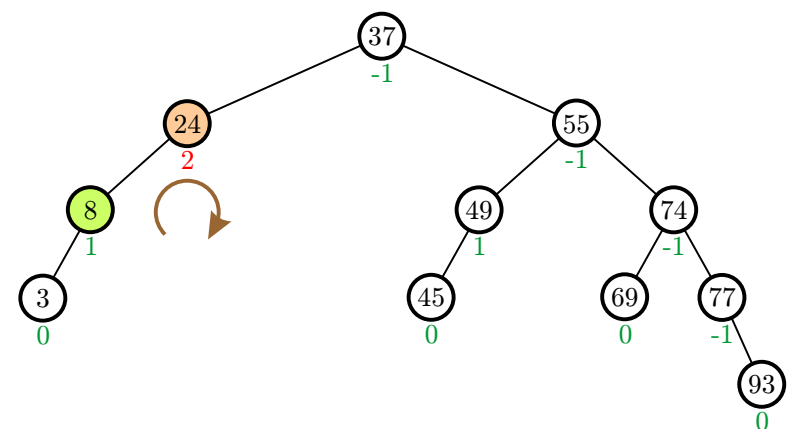
Exemple de suppression

Jusqu'à h rotations peuvent être nécessaires



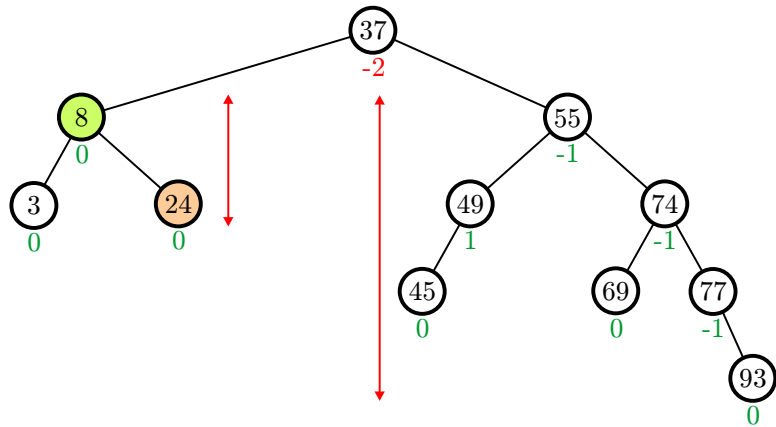
Exemple de suppression

Jusqu'à h rotations peuvent être nécessaires



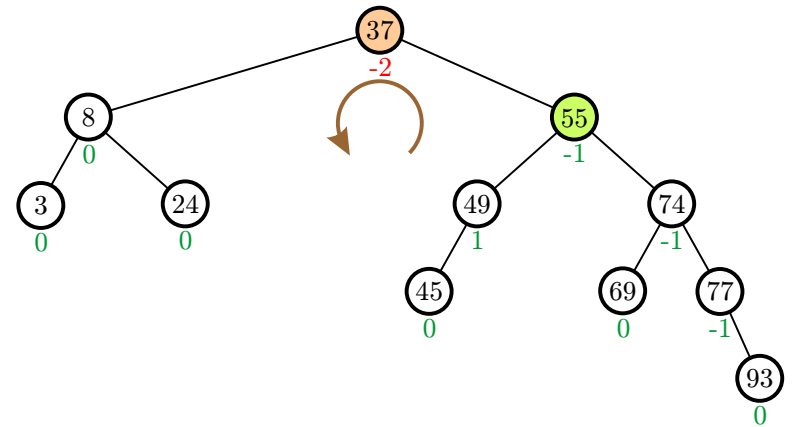
Exemple de suppression

Jusqu'à h rotations peuvent être nécessaires



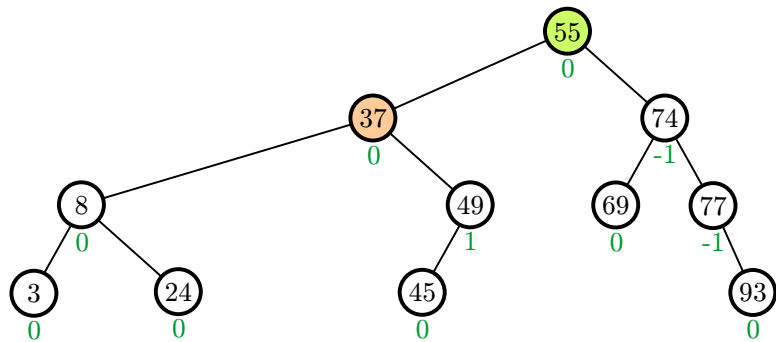
Exemple de suppression

Jusqu'à h rotations peuvent être nécessaires



Exemple de suppression

Jusqu'à h rotations peuvent être nécessaires



Implémentation en C
Manipulation avancée de pointeurs

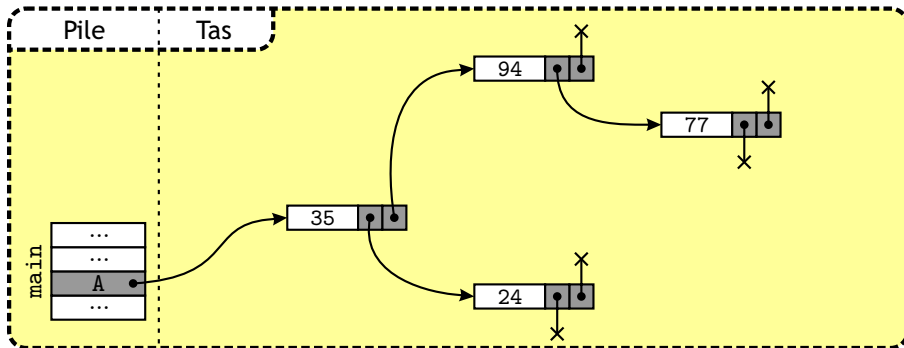
La rotation

Mauvaise façon de faire

```

1 int main() {
2   node* A = (node*) malloc(sizeof(node));
3   ....
4   rot_g(A);
5 }

```



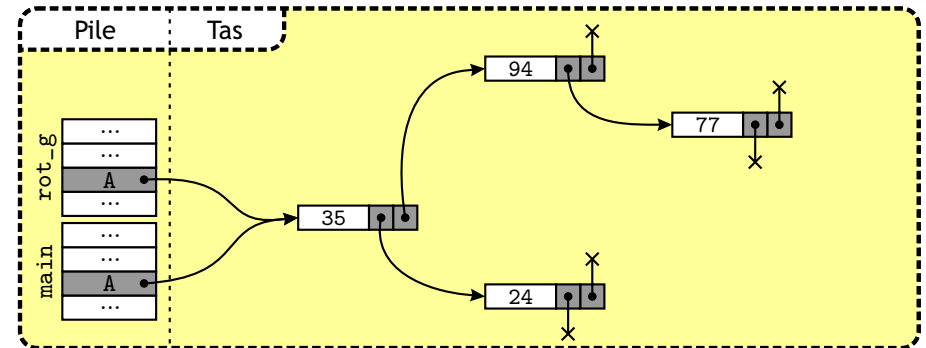
La rotation

Mauvaise façon de faire

```

1 int main() {
2   node* A = (node*) malloc(sizeof(node));
3   ....
4   rot_g(A);
5 }

```



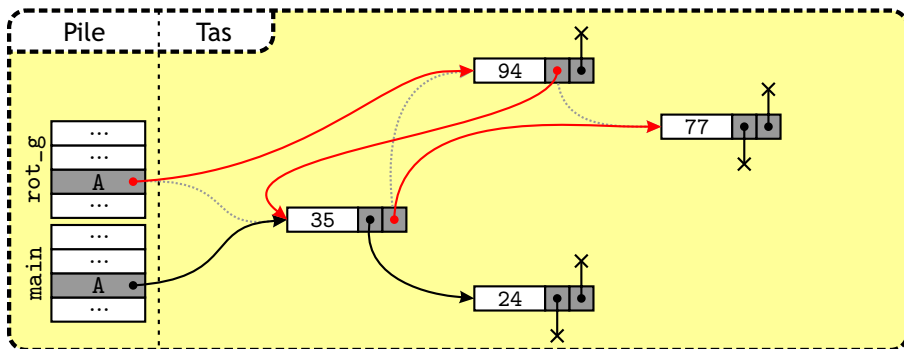
La rotation

Mauvaise façon de faire

```

1 int main() {
2   node* A = (node*) malloc(sizeof(node));
3   ....
4   rot_g(A);
5 }

```



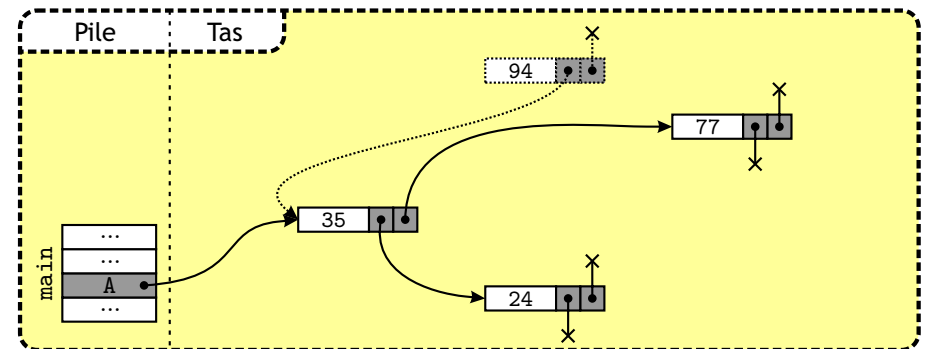
La rotation

Mauvaise façon de faire

```

1 int main() {
2   node* A = (node*) malloc(sizeof(node));
3   ....
4   rot_g(A);
5 }

```

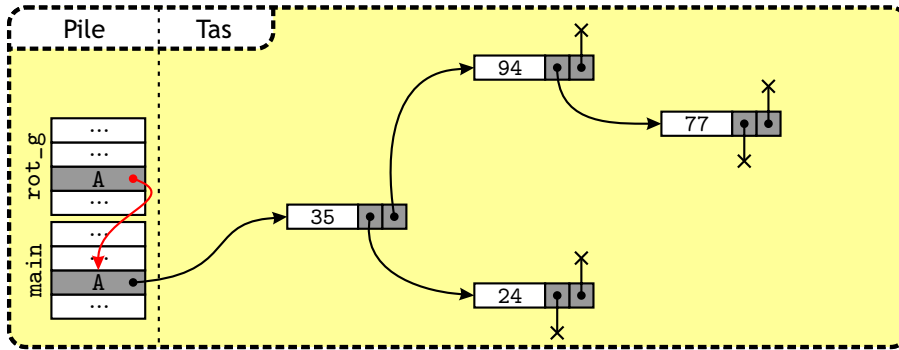


La rotation

Bonne façon de faire

```

1 int main() {
2   node* A = (node*) malloc(sizeof(node));
3   ....     // rot_g prend un node** en argument
4   rot_g(&A); // A du main et (*A) de rot_g sont
5 }           // la même case mémoire.
```

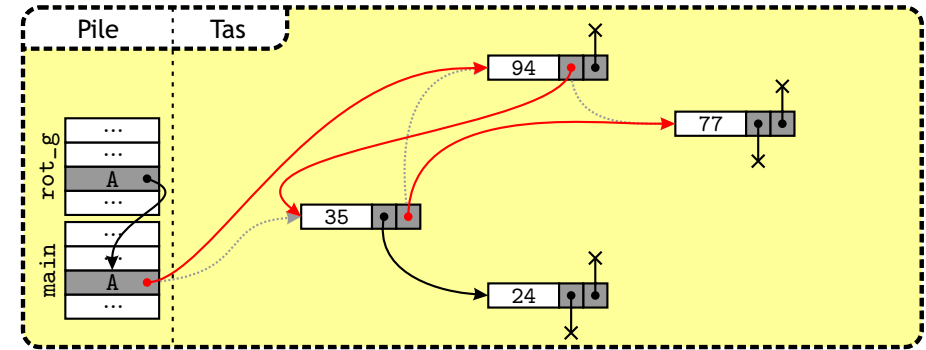


La rotation

Bonne façon de faire

```

1 int main() {
2   node* A = (node*) malloc(sizeof(node));
3   ....     // rot_g prend un node** en argument
4   rot_g(&A); // A du main et (*A) de rot_g sont
5 }           // la même case mémoire.
```

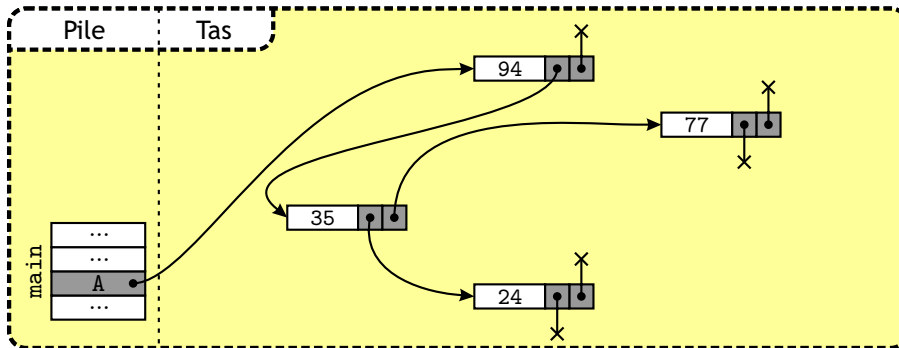


La rotation

Bonne façon de faire

```

1 int main() {
2   node* A = (node*) malloc(sizeof(node));
3   ....     // rot_g prend un node** en argument
4   rot_g(&A); // A du main et (*A) de rot_g sont
5 }           // la même case mémoire.
```



La rotation

Bonne façon de faire

- ✘ Dans la fonction rot_g on travaille toujours avec (*A).
- ✘ la dernière instruction modifie la racine de l'arbre,
- ✘ la variable locale A du main est aussi modifiée.

```

1 void rot_g(node** A) {
2   node* tmp;
3   if (((*A) != NULL) && ((*A)->fd != NULL)) {
4     tmp = (*A)->fd;
5     (*A)->fd = tmp->fg;
6     tmp->fg = (*A);
7     (*A) = tmp;
8   }
9 }
```

- ✘ En général, une insertion ne change pas la racine,
 - ✘ sauf pour un arbre vide...
- ✘ On veut une seule fonction pour tout faire :
 - ✘ cela permet d'avoir un main très propre.

```
1 int main() {  
2     node* A = NULL;  
3     insert(12,&A);  
4     insert(16,&A);  
5     insert(5,&A);  
6     insert(2,&A);  
7 }
```

- ✘ Une fois encore, on utilise tout le temps (*A).

```
1 void insert(int v, node** A) {  
2     if ((*A) == NULL) {  
3         node* n = (node*) malloc(sizeof(node));  
4         n->key = v;  
5         n->left = NULL;  
6         n->right = NULL;  
7         (*A) = n;  
8         return;  
9     }  
10    if (v < (*A)->key) {  
11        insert(v, &((*A)->left));  
12    } else {  
13        insert(v, &((*A)->right));  
14    }  
15 }
```

IN101 - 2011-2012

Ce qu'il faut retenir de ce cours

- ✘ Les arbres sont une structure très utile/utilisée en informatique
 - ✘ en général, un arbre équilibré est toujours plus efficace
 - un petit peu plus lourd à gérer, mais garantit l'efficacité.
 - ✘ les arbres AVL sont des ABR équilibrés faciles à implémenter
 - il suffit de savoir faire des rotations.
- ✘ Les tas sont un exemple un peu à part :
 - ✘ ce ne sont pas des ABR,
 - ✘ permettent de gérer une [file de priorité](#),
 - ✘ utilisés comme structure de base dans beaucoup d'algorithmes
 - un peu comme les piles/files.