Chapitre 4 : Complexité II

Alexandre Blondin Massé

Laboratoire d'informatique formelle
Université du Québec à Chicoutimi

27 janvier 2014
Cours 8INF809
Département d'informatique et mathématique

Table des matières

Table des matières

Rappels sur les différentes complexités

- La complexité dans le **pire cas**, qui donne une **borne supérieure** ;

Rappels sur les différentes complexités

- La complexité dans le **pire cas**, qui donne une **borne supérieure** ;
- La complexité **moyenne**, qui nécessite le calcul des **probabilités** ;

Rappels sur les différentes complexités

- La complexité dans le **pire cas**, qui donne une **borne supérieure** ;
- La complexité **moyenne**, qui nécessite le calcul des **probabilités** ;
- La complexité **amortie** donne une **borne supérieure** pour le pire cas lorsqu'une **suite d'opérations** sont effectuées.

Rappels sur les différentes complexités

- La complexité dans le **pire cas**, qui donne une **borne supérieure** ;
- La complexité **moyenne**, qui nécessite le calcul des **probabilités** ;
- La complexité **amortie** donne une **borne supérieure** pour le pire cas lorsqu'une **suite d'opérations** sont effectuées.
- Nous allons voir **trois méthodes** permettant de parler de complexité **amortie** :

Rappels sur les différentes complexités

- La complexité dans le **pire cas**, qui donne une **borne supérieure** ;
- La complexité **moyenne**, qui nécessite le calcul des **probabilités** ;
- La complexité **amortie** donne une **borne supérieure** pour le pire cas lorsqu'une **suite d'opérations** sont effectuées.
- Nous allons voir **trois méthodes** permettant de parler de complexité **amortie** :
  - L'analyse **cumulée** ;

Rappels sur les différentes complexités

- La complexité dans le **pire cas**, qui donne une **borne supérieure** ;
- La complexité **moyenne**, qui nécessite le calcul des **probabilités** ;
- La complexité **amortie** donne une **borne supérieure** pour le pire cas lorsqu'une **suite d'opérations** sont effectuées.
- Nous allons voir **trois méthodes** permettant de parler de complexité **amortie** :
  - L'analyse **cumulée** ;
  - La méthode **comptable** ;

Rappels sur les différentes complexités

- La complexité dans le **pire cas**, qui donne une **borne supérieure** ;
- La complexité **moyenne**, qui nécessite le calcul des **probabilités** ;
- La complexité **amortie** donne une **borne supérieure** pour le pire cas lorsqu'une **suite d'opérations** sont effectuées.
- Nous allons voir **trois méthodes** permettant de parler de complexité **amortie** :
  - L'analyse **cumulée** ;
  - La méthode **comptable** ;
  - La méthode du **potentiel**.

Analyse cumulée

- Il suffit de montrer le temps **total** $T(n)$ pris pour effectuer n **opérations** ;

Analyse cumulée

- Il suffit de montrer le temps **total** $T(n)$ pris pour effectuer n **opérations** ;
- On peut alors en conclure que le **coût amorti** par opération est $T(n)/n$;

Analyse cumulée

- Il suffit de montrer le temps **total** $T(n)$ pris pour effectuer n **opérations** ;
- On peut alors en conclure que le **coût amorti** par opération est $T(n)/n$;
- Contrairement aux méthodes **comptable** et **du potentiel**, on considère que **toutes les opérations** ont le même coût amorti ;

Analyse cumulée

- Il suffit de montrer le temps **total** $T(n)$ pris pour effectuer n **opérations** ;
- On peut alors en conclure que le **coût amorti** par opération est $T(n)/n$;
- Contrairement aux méthodes **comptable** et **du potentiel**, on considère que **toutes les opérations** ont le même coût amorti ;
- Dans ces deux cas, il est possible de **distinguer** le coût **selon l'opération**.

Extension de la pile

- Reprenons la **pile** et ses opérations :

Extension de la pile

- Reprenons la **pile** et ses opérations :
  - PileVide() : retourne une pile vide ;
  - P.EstVide() : indique si la pile est vide ou non ;
  - P.Empiler(x) : ajoute x au sommet de la pile ;
  - P.Sommet() : retourne l'élément au sommet de la pile ;
  - P.Dépiler() : retire l'élément du sommet de la pile.

Extension de la pile

- Reprenons la **pile** et ses opérations :
  - PileVide() : retourne une pile vide ;
  - P.EstVide() : indique si la pile est vide ou non ;
  - P.Empiler(x) : ajoute x au sommet de la pile ;
  - P.Sommet() : retourne l'élément au sommet de la pile ;
  - P.Dépiler() : retire l'élément du sommet de la pile.
- On ajoute un **opération supplémentaire** :

Extension de la pile

- Reprenons la **pile** et ses opérations :
  - PileVide() : retourne une pile vide ;
  - P.EstVide() : indique si la pile est vide ou non ;
  - P.Empiler(x) : ajoute x au sommet de la pile ;
  - P.Sommet() : retourne l'élément au sommet de la pile ;
  - P.Dépiler() : retire l'élément du sommet de la pile.
- On ajoute un **opération supplémentaire** :
  - P.DépilerPlusieurs(k) : retire les k premiers éléments de la pile ou jusqu'à ce que la pile soit vide ;

Extension de la pile

- Reprenons la **pile** et ses opérations :
  - PileVide() : retourne une pile vide ;
  - P.EstVide() : indique si la pile est vide ou non ;
  - P.Empiler(x) : ajoute x au sommet de la pile ;
  - P.Sommet() : retourne l'élément au sommet de la pile ;
  - P.Dépiler() : retire l'élément du sommet de la pile.
- On ajoute un **opération supplémentaire** :
  - P.DépilerPlusieurs(k) : retire les k premiers éléments de la pile ou jusqu'à ce que la pile soit vide ;
- Toutes les opérations se font en $\mathcal{O}(1)$ sauf la dernière qui se fait en $\mathcal{O}(\min(|P|, k))$.

Coût d'une suite d'opérations

- Supposons qu'on effectue n **opérations** parmi Empiler, Dépiler et DépilerPlusieurs ;

Coût d'une suite d'opérations

- Supposons qu'on effectue n **opérations** parmi Empiler, Dépiler et DépilerPlusieurs ;
- Quel est le **coût maximum** possible ?

Coût d'une suite d'opérations

- Supposons qu'on effectue n **opérations** parmi Empiler, Dépiler et DépilerPlusieurs ;
- Quel est le **coût maximum** possible ?
- Comme la **taille de la pile** est au plus n et que l'opération DépilerPlusieurs se fait en temps $\mathcal{O}(n)$, alors le coût total sera $\mathcal{O}(n^2)$;

Coût d'une suite d'opérations

- Supposons qu'on effectue n **opérations** parmi Empiler, Dépiler et DépilerPlusieurs ;
- Quel est le **coût maximum** possible ?
- Comme la **taille de la pile** est au plus n et que l'opération DépilerPlusieurs se fait en temps $\mathcal{O}(n)$, alors le coût total sera $\mathcal{O}(n^2)$;
- L'analyse précédente est **correcte**, mais pas **assez précise** ;

Coût d'une suite d'opérations (suite)

- En fait, si on effectue **plusieurs opérations** DépilerPlusieurs consécutive, alors elles ne peuvent pas **toutes** êtres coûteuses ;

Coût d'une suite d'opérations (suite)

- En fait, si on effectue **plusieurs opérations** DépilerPlusieurs consécutive, alors elles ne peuvent pas **toutes** êtres coûteuses ;
- Plus précisément, si on commence avec une **pile vide**, alors le nombre total d'opérations Empiler est **plus grand ou égal** que le nombre total d'opérations Dépiler ;

Coût d'une suite d'opérations (suite)

- En fait, si on effectue **plusieurs opérations** DépilerPlusieurs consécutive, alors elles ne peuvent pas **toutes** êtres coûteuses ;
- Plus précisément, si on commence avec une **pile vide**, alors le nombre total d'opérations Empiler est **plus grand ou égal** que le nombre total d'opérations Dépiler ;
- On en conclut que le temps total pris est $\mathcal{O}(n)$, ce qui est plus **précis** que $\mathcal{O}(n^2)$;

Coût d'une suite d'opérations (suite)

- En fait, si on effectue **plusieurs opérations** DépilerPlusieurs consécutive, alors elles ne peuvent pas **toutes** êtres coûteuses ;
- Plus précisément, si on commence avec une **pile vide**, alors le nombre total d'opérations Empiler est **plus grand ou égal** que le nombre total d'opérations Dépiler ;
- On en conclut que le temps total pris est $\mathcal{O}(n)$, ce qui est plus **précis** que $\mathcal{O}(n^2)$;
- Le **coût moyen** d'une opération est donc $\mathcal{O}(n)/n = \mathcal{O}(1)$;

Coût d'une suite d'opérations (suite)

- En fait, si on effectue **plusieurs opérations** DépilerPlusieurs consécutive, alors elles ne peuvent pas **toutes** êtres coûteuses ;
- Plus précisément, si on commence avec une **pile vide**, alors le nombre total d'opérations Empiler est **plus grand ou égal** que le nombre total d'opérations Dépiler ;
- On en conclut que le temps total pris est $\mathcal{O}(n)$, ce qui est plus **précis** que $\mathcal{O}(n^2)$;
- Le **coût moyen** d'une opération est donc $\mathcal{O}(n)/n = \mathcal{O}(1)$;
- À noter que le **coût amorti** ne se base pas sur des **probabilités**, mais considère **tous les cas possibles**.

Compteur binaire

Valeur	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0

Incrémentation d'un compteur binaire

```
1:  procedure INCRÉMENTER(A : tableau de bits)
2:      i ← 0
3:      tant que i < |A| et A[i] = 1 faire
4:          A[i] ← 0
5:          i ← i + 1
6:      fin tant que
7:      si i < |A| alors
8:          A[i] ← 1
9:      fin si
10: fin procedure
```

Incrémentation d'un compteur binaire

```
1:  procedure INCRÉMENTER(A : tableau de bits)
2:      i ← 0
3:      tant que i < |A| et A[i] = 1 faire
4:          A[i] ← 0
5:          i ← i + 1
6:      fin tant que
7:      si i < |A| alors
8:          A[i] ← 1
9:      fin si
10: fin procedure
```

- Soit $k = |A|$;

Incrémentation d'un compteur binaire

```
1: procedure INCRÉMENTER(A : tableau de bits)
2:     i ← 0
3:     tant que i < |A| et A[i] = 1 faire
4:         A[i] ← 0
5:         i ← i + 1
6:     fin tant que
7:     si i < |A| alors
8:         A[i] ← 1
9:     fin si
10: fin procedure
```

- Soit $k = |A|$;
- Alors une incrémentation se fait en $\mathcal{O}(k)$;

Incrémentation d'un compteur binaire

```
1:  procedure INCRÉMENTER(A : tableau de bits)
2:      i ← 0
3:      tant que i < |A| et A[i] = 1 faire
4:          A[i] ← 0
5:          i ← i + 1
6:      fin tant que
7:      si i < |A| alors
8:          A[i] ← 1
9:      fin si
10: fin procedure
```

- Soit $k = |A|$;
- Alors une incrémentation se fait en $\mathcal{O}(k)$;
- Une suite de n **incrémentations** a un coût total de $\mathcal{O}(nk)$.

Compteur binaire (suite)

Valeur	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0

Coût d'une suite d'incrémentations

- Supposons qu'on effectue n **incrémentations** successives en commençant par la valeur 0 ;

Coût d'une suite d'incrémentations

- Supposons qu'on effectue n **incrémentations** successives en commençant par la valeur 0 ;
- Alors

Coût d'une suite d'incrémentations

- Supposons qu'on effectue n **incrémentations** successives en commençant par la valeur 0 ;
- Alors
  - le bit $A[0]$ est inversé **chaque fois** ;

Coût d'une suite d'incrémentations

- Supposons qu'on effectue n **incrémentations** successives en commençant par la valeur 0 ;
- Alors
  - le bit $A[0]$ est inversé **chaque fois** ;
  - le bit $A[1]$ est inversé $\lfloor n/2 \rfloor$ fois ;

Coût d'une suite d'incrémentations

- Supposons qu'on effectue n **incrémentations** successives en commençant par la valeur 0 ;
- Alors
  - le bit $A[0]$ est inversé **chaque fois** ;
  - le bit $A[1]$ est inversé $\lfloor n/2 \rfloor$ fois ;
  - le bit $A[2]$ est inversé $\lfloor n/4 \rfloor$ fois ;

Coût d'une suite d'incrémentations

- Supposons qu'on effectue n **incrémentations** successives en commençant par la valeur 0 ;
- Alors
  - le bit $A[0]$ est inversé **chaque fois** ;
  - le bit $A[1]$ est inversé $\lfloor n/2 \rfloor$ fois ;
  - le bit $A[2]$ est inversé $\lfloor n/4 \rfloor$ fois ;
- Le **coût total** est donc

Coût d'une suite d'incrémentations

- Supposons qu'on effectue n **incrémentations** successives en commençant par la valeur 0 ;
- Alors
  - le bit $A[0]$ est inversé **chaque fois** ;
  - le bit $A[1]$ est inversé $\lfloor n/2 \rfloor$ fois ;
  - le bit $A[2]$ est inversé $\lfloor n/4 \rfloor$ fois ;
- Le **coût total** est donc

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor$$

Coût d'une suite d'incrémentations

- Supposons qu'on effectue n **incrémentations** successives en commençant par la valeur 0 ;
- Alors
  - le bit $A[0]$ est inversé **chaque fois** ;
  - le bit $A[1]$ est inversé $\lfloor n/2 \rfloor$ fois ;
  - le bit $A[2]$ est inversé $\lfloor n/4 \rfloor$ fois ;
- Le **coût total** est donc

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i}$$

Coût d'une suite d'incrémentations

- Supposons qu'on effectue n **incrémentations** successives en commençant par la valeur 0 ;
- Alors
  - le bit $A[0]$ est inversé **chaque fois** ;
  - le bit $A[1]$ est inversé $\lfloor n/2 \rfloor$ fois ;
  - le bit $A[2]$ est inversé $\lfloor n/4 \rfloor$ fois ;
- Le **coût total** est donc

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

- Le **coût amorti** est par conséquent $\mathcal{O}(1)$ par incrémentation.

La méthode comptable

- La **méthode comptable** consiste à affecter à **chaque opération** un **coût amorti** ;

La méthode comptable

- La **méthode comptable** consiste à affecter à **chaque opération** un **coût amorti** ;
- Si le **coût amorti** d'une opération est **plus grand** que son **coût réel**, alors le **supplément** est appelé **crédit** ;

La méthode comptable

- La **méthode comptable** consiste à affecter à **chaque opération** un **coût amorti** ;
- Si le **coût amorti** d'une opération est **plus grand** que son **coût réel**, alors le **supplément** est appelé **crédit** ;
- Le **crédit** permet de « **payer** » pour des **opérations subséquentes** ;

La méthode comptable (suite)

- Lorsqu'on choisit les **coûts amortis**, on doit s'assurer que **toute suite** de n opérations donne toujours un **coût amorti** supérieur au **coût réel** :

La méthode comptable (suite)

- Lorsqu'on choisit les **coûts amortis**, on doit s'assurer que **toute suite** de n opérations donne toujours un **coût amorti** supérieur au **coût réel** :

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i,$$

où c_i est le **coût réel** et $\hat{c}_i$ est le **coût amorti** de la i**-ème opération** ;

La méthode comptable (suite)

- Lorsqu'on choisit les **coûts amortis**, on doit s'assurer que **toute suite** de n opérations donne toujours un **coût amorti** supérieur au **coût réel** :

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i,$$

où c_i est le **coût réel** et $\hat{c}_i$ est le **coût amorti** de la i**-ème opération** ;

- Autrement dit, le **crédit total**

$$\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$$

doit être **positif en tout temps**.

Retour sur la pile

- Les **coûts réels** des opérations sur une pile sont :

Empiler	1,
Dépiler	1,
DépilerPlusieurs	$\min(k, \|P\|)$.

Retour sur la pile

- Les **coûts réels** des opérations sur une pile sont :

Empiler	1,
Dépiler	1,
DépilerPlusieurs	$\min(k, \lvert P \rvert)$.

- On choisit comme **coûts amortis** les valeurs suivantes :

Empiler	2,
Dépiler	0,
DépilerPlusieurs	0.

Retour sur la pile

- Les **coûts réels** des opérations sur une pile sont :

Empiler	1,
Dépiler	1,
DépilerPlusieurs	$\min(k, \lvert P \rvert)$.

- On choisit comme **coûts amortis** les valeurs suivantes :

Empiler	2,
Dépiler	0,
DépilerPlusieurs	0.

- Clairement, le **crédit total** est toujours positif, puisqu'il est égal au **nombre d'éléments** dans la pile à chaque instant.

Retour sur le compteur binaire

- On peut **décomposer** l'incrémentation en **deux opérations** élémentaires :

Retour sur le compteur binaire

- On peut **décomposer** l'incrémentation en **deux opérations** élémentaires :
  - Changer un bit de 0 vers 1 ;

Retour sur le compteur binaire

- On peut **décomposer** l'incrémentation en **deux opérations** élémentaires :
  - Changer un bit de 0 vers 1 ;
  - Changer un bit de 1 vers 0.

Retour sur le compteur binaire

- On peut **décomposer** l'incrémentation en **deux opérations** élémentaires :
  - Changer un bit de 0 vers 1 ;
  - Changer un bit de 1 vers 0.
- On affecte alors un **coût amorti** de 2 pour changer un bit à 1 et un coût de 0 pour changer un bit à 0 ;

Retour sur le compteur binaire

- On peut **décomposer** l'incrémentation en **deux opérations** élémentaires :
  - Changer un bit de 0 vers 1 ;
  - Changer un bit de 1 vers 0.
- On affecte alors un **coût amorti** de 2 pour changer un bit à 1 et un coût de 0 pour changer un bit à 0 ;
- Le **crédit total** est **toujours positif**, puisqu'il correspond au **nombre de bits égaux à** 1 ;

Retour sur le compteur binaire

- On peut **décomposer** l'incrémentation en **deux opérations** élémentaires :
  - Changer un bit de 0 vers 1 ;
  - Changer un bit de 1 vers 0.
- On affecte alors un **coût amorti** de 2 pour changer un bit à 1 et un coût de 0 pour changer un bit à 0 ;
- Le **crédit total** est **toujours positif**, puisqu'il correspond au **nombre de bits égaux à** 1 ;
- De plus, comme **chaque incrémentation** change **au plus** un bit de 0 vers 1, son **coût amorti** est $\mathcal{O}(1)$.

La méthode du potentiel

- C'est la méthode la **plus utilisée** en pratique ;

La méthode du potentiel

- C'est la méthode la **plus utilisée** en pratique ;
- Une **fonction de potentiel** Φ est une fonction qui **affecte** à une **structure de données** un **nombre réel** ;

La méthode du potentiel

- C'est la méthode la **plus utilisée** en pratique ;
- Une **fonction de potentiel** Φ est une fonction qui **affecte** à une **structure de données** un **nombre réel** ;
- Analogue au concept d'**énergie potentielle** ;

La méthode du potentiel

- C'est la méthode la **plus utilisée** en pratique ;
- Une **fonction de potentiel** Φ est une fonction qui **affecte** à une **structure de données** un **nombre réel** ;
- Analogue au concept d'**énergie potentielle** ;
- Soient D_0, D_1, ... une suite de **structures de données** obtenues par **application successive** d'opérations ;

La méthode du potentiel

- C'est la méthode la **plus utilisée** en pratique ;
- Une **fonction de potentiel** Φ est une fonction qui **affecte** à une **structure de données** un **nombre réel** ;
- Analogue au concept d'**énergie potentielle** ;
- Soient D_0, D_1, ... une suite de **structures de données** obtenues par **application successive** d'opérations ;
- Le **coût amorti** $\hat{c}_i$ de la i**-ème** opération par rapport à Φ est défini par

La méthode du potentiel

- C'est la méthode la **plus utilisée** en pratique ;
- Une **fonction de potentiel** Φ est une fonction qui **affecte** à une **structure de données** un **nombre réel** ;
- Analogue au concept d'**énergie potentielle** ;
- Soient D_0, D_1, ... une suite de **structures de données** obtenues par **application successive** d'opérations ;
- Le **coût amorti** $\hat{c}_i$ de la i**-ème** opération par rapport à Φ est défini par

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

La méthode du potentiel (suite)

- Le **coût amorti total** d'une suite de n **opérations** est

La méthode du potentiel (suite)

- Le **coût amorti total** d'une suite de n **opérations** est

$$\sum_{i=1}^{n} \hat{c}_i \quad = \quad \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

La méthode du potentiel (suite)

- Le **coût amorti total** d'une suite de n **opérations** est

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0).\end{aligned}$$

La méthode du potentiel (suite)

- Le **coût amorti total** d'une suite de n **opérations** est

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0).\end{aligned}$$

- Si on choisit Φ de sorte que $\Phi(D_n) \geq \Phi(D_0)$, alors le coût amorti total est une **borne supérieure** du coût réel total ;

La méthode du potentiel (suite)

- Le **coût amorti total** d'une suite de n **opérations** est

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0).\end{aligned}$$

- Si on choisit Φ de sorte que $\Phi(D_n) \geq \Phi(D_0)$, alors le coût amorti total est une **borne supérieure** du coût réel total ;
- En pratique, on a souvent $\Phi(D_0) = 0$ et il suffit de montrer que $\Phi(D_i) \geq 0$ pour tout i.

La pile avec la méthode du potentiel

- Si on reprend l'exemple de la **pile**, la fonction **potentiel** la plus simple est

La pile avec la méthode du potentiel

- Si on reprend l'exemple de la **pile**, la fonction **potentiel** la plus simple est

$$\Phi(D_i) = \textbf{nombre d'éléments} \text{ dans la pile } D_i.$$

La pile avec la méthode du potentiel

- Si on reprend l'exemple de la **pile**, la fonction **potentiel** la plus simple est

$$\Phi(D_i) = \textbf{nombre d'éléments} \text{ dans la pile } D_i.$$

- Clairement,

La pile avec la méthode du potentiel

- Si on reprend l'exemple de la **pile**, la fonction **potentiel** la plus simple est

$$\Phi(D_i) = \textbf{nombre d'éléments} \text{ dans la pile } D_i.$$

- Clairement,
  - $\Phi(D_0) = 0$ quand on commence avec une **pile vide** et

La pile avec la méthode du potentiel

- Si on reprend l'exemple de la **pile**, la fonction **potentiel** la plus simple est

$$\Phi(D_i) = \textbf{nombre d'éléments} \text{ dans la pile } D_i.$$

- Clairement,
  - $\Phi(D_0) = 0$ quand on commence avec une **pile vide** et
  - $\Phi(D_i) \geq 0$ **pour tout** i.

La pile avec la méthode du potentiel

- Si on reprend l'exemple de la **pile**, la fonction **potentiel** la plus simple est

$$\Phi(D_i) = \textbf{nombre d'éléments} \text{ dans la pile } D_i.$$

- Clairement,
  - $\Phi(D_0) = 0$ quand on commence avec une **pile vide** et
  - $\Phi(D_i) \geq 0$ **pour tout** i.
- Il ne nous reste qu'à calculer le **coût amorti** de chaque opération.

Coût amorti des opérations sur une pile

- Si la **i-ème opération** est EMPILER, alors le coût amorti est

Coût amorti des opérations sur une pile

- Si la ***i*-ème opération** est EMPILER, alors le coût amorti est

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Coût amorti des opérations sur une pile

- Si la ***i*-ème opération** est Empiler, alors le coût amorti est

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (|D_{i-1}| + 1) - |D_{i-1}| \end{aligned}$$

Coût amorti des opérations sur une pile

- Si la **i-ème opération** est EMPILER, alors le coût amorti est

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (|D_{i-1}| + 1) - |D_{i-1}| \\ &= 1 + 1\end{aligned}$$

Coût amorti des opérations sur une pile

- Si la **i-ème opération** est EMPILER, alors le coût amorti est

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= 1 + (|D_{i-1}| + 1) - |D_{i-1}| \\
&= 1 + 1 \\
&= 2.
\end{aligned}$$

Coût amorti des opérations sur une pile

- Si la i-**ème opération** est EMPILER, alors le coût amorti est

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (|D_{i-1}| + 1) - |D_{i-1}| \\ &= 1 + 1 \\ &= 2.\end{aligned}$$

- Si la i-**ème opération** est DÉPILERPLUSIEURS (qui inclut DÉPILER quand $k = 1$), alors le coût amorti est

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Coût amorti des opérations sur une pile

- Si la i-**ème opération** est EMPILER, alors le coût amorti est

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (|D_{i-1}| + 1) - |D_{i-1}| \\ &= 1 + 1 \\ &= 2.\end{aligned}$$

- Si la i-**ème opération** est DÉPILERPLUSIEURS (qui inclut DÉPILER quand $k = 1$), alors le coût amorti est

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k + (|D_{i-1}| - k) - |D_{i-1}|\end{aligned}$$

Coût amorti des opérations sur une pile

- Si la i-ème **opération** est EMPILER, alors le coût amorti est

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (|D_{i-1}| + 1) - |D_{i-1}| \\ &= 1 + 1 \\ &= 2.\end{aligned}$$

- Si la i-ème **opération** est DÉPILERPLUSIEURS (qui inclut DÉPILER quand $k = 1$), alors le coût amorti est

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k + (|D_{i-1}| - k) - |D_{i-1}| \\ &= k - k\end{aligned}$$

Coût amorti des opérations sur une pile

- Si la i-**ème opération** est EMPILER, alors le coût amorti est

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (|D_{i-1}| + 1) - |D_{i-1}| \\ &= 1 + 1 \\ &= 2.\end{aligned}$$

- Si la i-**ème opération** est DÉPILERPLUSIEURS (qui inclut DÉPILER quand $k = 1$), alors le coût amorti est

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k + (|D_{i-1}| - k) - |D_{i-1}| \\ &= k - k \\ &= 0,\end{aligned}$$

Coût amorti des opérations sur une pile

- Si la i-**ème opération** est EMPILER, alors le coût amorti est

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (|D_{i-1}| + 1) - |D_{i-1}| \\ &= 1 + 1 \\ &= 2.\end{aligned}$$

- Si la i-**ème opération** est DÉPILERPLUSIEURS (qui inclut DÉPILER quand $k = 1$), alors le coût amorti est

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k + (|D_{i-1}| - k) - |D_{i-1}| \\ &= k - k \\ &= 0,\end{aligned}$$

La méthode du potentiel avec le compteur binaire

- Dans le cas du **compteur binaire**, la fonction **potentiel** est définie par

La méthode du potentiel avec le compteur binaire

- Dans le cas du **compteur binaire**, la fonction **potentiel** est définie par

$$\Phi(D_i) = \textbf{nombre de } 1 \text{ affichés par le compteur } D_i.$$

La méthode du potentiel avec le compteur binaire

- Dans le cas du **compteur binaire**, la fonction **potentiel** est définie par

$$\Phi(D_i) = \textbf{nombre de } 1 \text{ affichés par le compteur } D_i.$$

- Soit u_i le nombre de 1 affichés par le compteur après la **i-ème incrémentation** ;

La méthode du potentiel avec le compteur binaire

- Dans le cas du **compteur binaire**, la fonction **potentiel** est définie par

$$\Phi(D_i) = \textbf{nombre de } 1 \text{ affichés par le compteur } D_i.$$

- Soit u_i le nombre de 1 affichés par le compteur après la **i-ème incrémentation** ;
- Soit z_i le nombre de bits qui sont affectés à la **valeur** 0 après la **i-ème incrémentation** ;

La méthode du potentiel avec le compteur binaire

- Dans le cas du **compteur binaire**, la fonction **potentiel** est définie par

$$\Phi(D_i) = \textbf{nombre de } 1 \text{ affichés par le compteur } D_i.$$

- Soit u_i le nombre de 1 affichés par le compteur après la i**-ème incrémentation** ;
- Soit z_i le nombre de bits qui sont affectés à la **valeur** 0 après la i**-ème incrémentation** ;
- Alors le **coût réel** d'une incrémentation est au plus $z_i + 1$, en incluant le bit qui est affecté à 1 ;

La méthode du potentiel avec le compteur binaire

- Clairement,

La méthode du potentiel avec le compteur binaire

- Clairement,

$$u_i = u_{i-1} - z_i + 1.$$

La méthode du potentiel avec le compteur binaire

- Clairement,

$$u_i = u_{i-1} - z_i + 1.$$

- Par conséquent,

La méthode du potentiel avec le compteur binaire

- Clairement,

$$u_i = u_{i-1} - z_i + 1.$$

- Par conséquent,

$$\begin{aligned} \hat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= (z_i + 1) + (u_{i-1} - z_i + 1) - u_{i-1} \\ &= 2. \end{aligned}$$

La méthode du potentiel avec le compteur binaire

- Clairement,

$$u_i = u_{i-1} - z_i + 1.$$

- Par conséquent,

$$\begin{aligned} \hat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= (z_i + 1) + (u_{i-1} - z_i + 1) - u_{i-1} \\ &= 2. \end{aligned}$$

- Ainsi, le **coût amorti** d'une incrémentation est $\mathcal{O}(1)$.

Redimensionnement de tableaux

```
1:  procedure INSÉRER(T : tableau, i : indice, x : élément)
2:      si T.capacité = 0 alors
3:          Redimensionner le tableau à une taille 1
4:          T.capacité ← 1
5:      sinon si T.utilisé = T.capacité alors
6:          Créer un tableau de taille 2 · T.capacité
7:          Recopier tous les éléments dans le nouveau tableau
8:          Libérer l'espace mémoire utilisé
9:          T.capacité ← 2 · T.capacité
10:     fin si
11:     T[i] ← x
12:     T.utilisé ← T.utilisé + 1
13: fin procedure
```

Analyse de la complexité du redimensionnement

- Une fonction **potentielle** qui permet de calculer la **complexité amortie** est

Analyse de la complexité du redimensionnement

- Une fonction **potentielle** qui permet de calculer la **complexité amortie** est

$$\Phi(T) = 2 \cdot T.\text{utilisé} - T.\text{capacité}.$$

Analyse de la complexité du redimensionnement

- Une fonction **potentielle** qui permet de calculer la **complexité amortie** est

$$\Phi(T) = 2 \cdot T.\text{utilisé} - T.\text{capacité}.$$

- Cette fonction est **toujours positive**, car au moins la **moitié** du tableau est **occupé** (on suppose qu'il n'y a pas de **suppressions possibles**) ;

Analyse de la complexité du redimensionnement

- Une fonction **potentielle** qui permet de calculer la **complexité amortie** est

$$\Phi(T) = 2 \cdot T.\text{utilisé} - T.\text{capacité}.$$

- Cette fonction est **toujours positive**, car au moins la **moitié** du tableau est **occupé** (on suppose qu'il n'y a pas de **suppressions possibles**) ;
- Aussi, $\Phi(T) = 0$ si on vient tout juste de **redimensionner** le tableau ;

Analyse de la complexité du redimensionnement

- Une fonction **potentielle** qui permet de calculer la **complexité amortie** est

$$\Phi(T) = 2 \cdot T.\text{utilisé} - T.\text{capacité}.$$

- Cette fonction est **toujours positive**, car au moins la **moitié** du tableau est **occupé** (on suppose qu'il n'y a pas de **suppressions possibles**) ;
- Aussi, $\Phi(T) = 0$ si on vient tout juste de **redimensionner** le tableau ;
- Soit u_i le nombre de cases **utilisées** par le tableau après la i**-ème opération** ;

Analyse de la complexité du redimensionnement

- Une fonction **potentielle** qui permet de calculer la **complexité amortie** est

$$\Phi(T) = 2 \cdot T.\text{utilisé} - T.\text{capacité}.$$

- Cette fonction est **toujours positive**, car au moins la **moitié** du tableau est **occupé** (on suppose qu'il n'y a pas de **suppressions possibles**) ;
- Aussi, $\Phi(T) = 0$ si on vient tout juste de **redimensionner** le tableau ;
- Soit u_i le nombre de cases **utilisées** par le tableau après la i**-ème opération** ;
- Soit s_i la **capacité** du tableau après la i**-ème opération**.

Coût amorti d'un redimensionnement

- Si la ***i*-ème insertion** ne provoque pas de **redimensionnement**, alors

Coût amorti d'un redimensionnement

- Si la i-**ème insertion** ne provoque pas de **redimensionnement**, alors

$$\hat{c}_i \quad = \quad c_i + \Phi(T_i) - \Phi(T_{i-1})$$

Coût amorti d'un redimensionnement

- Si la i-**ème insertion** ne provoque pas de **redimensionnement**, alors

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\ &= 1 + (2u_i - s_i) - (2u_{i-1} - s_{i-1}) \end{aligned}$$

Coût amorti d'un redimensionnement

- Si la *i*-**ème insertion** ne provoque pas de **redimensionnement**, alors

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= 1 + (2u_i - s_i) - (2u_{i-1} - s_{i-1}) \\
&= 1 + (2u_i - s_i) - (2(u_i - 1) - s_i)
\end{aligned}$$

Coût amorti d'un redimensionnement

- Si la i-**ème insertion** ne provoque pas de **redimensionnement**, alors

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= 1 + (2u_i - s_i) - (2u_{i-1} - s_{i-1}) \\
&= 1 + (2u_i - s_i) - (2(u_i - 1) - s_i) \\
&= 3.
\end{aligned}$$

Coût amorti d'un redimensionnement

- Si la i-**ème insertion** ne provoque pas de **redimensionnement**, alors

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= 1 + (2u_i - s_i) - (2u_{i-1} - s_{i-1}) \\
&= 1 + (2u_i - s_i) - (2(u_i - 1) - s_i) \\
&= 3.
\end{aligned}$$

- Si elle en **provoque** un, on a plutôt

Coût amorti d'un redimensionnement

- Si la i-**ème insertion** ne provoque pas de **redimensionnement**, alors

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\ &= 1 + (2u_i - s_i) - (2u_{i-1} - s_{i-1}) \\ &= 1 + (2u_i - s_i) - (2(u_i - 1) - s_i) \\ &= 3.\end{aligned}$$

- Si elle en **provoque** un, on a plutôt

$$\hat{c}_i = c_i + \Phi(T_i) - \Phi(T_{i-1})$$

Coût amorti d'un redimensionnement

- Si la i-**ème insertion** ne provoque pas de **redimensionnement**, alors

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\ &= 1 + (2u_i - s_i) - (2u_{i-1} - s_{i-1}) \\ &= 1 + (2u_i - s_i) - (2(u_i - 1) - s_i) \\ &= 3.\end{aligned}$$

- Si elle en **provoque** un, on a plutôt

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\ &= u_i + (2u_i - s_i) - (2u_{i-1} - s_{i-1})\end{aligned}$$

Coût amorti d'un redimensionnement

- Si la *i*-**ème insertion** ne provoque pas de **redimensionnement**, alors

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\ &= 1 + (2u_i - s_i) - (2u_{i-1} - s_{i-1}) \\ &= 1 + (2u_i - s_i) - (2(u_i - 1) - s_i) \\ &= 3.\end{aligned}$$

- Si elle en **provoque** un, on a plutôt

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\ &= u_i + (2u_i - s_i) - (2u_{i-1} - s_{i-1}) \\ &= u_i + (2u_i - 2(u_i - 1)) - (2(u_i - 1) - (u_i - 1))\end{aligned}$$

Coût amorti d'un redimensionnement

- Si la *i*-**ème insertion** ne provoque pas de **redimensionnement**, alors

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\ &= 1 + (2u_i - s_i) - (2u_{i-1} - s_{i-1}) \\ &= 1 + (2u_i - s_i) - (2(u_i - 1) - s_i) \\ &= 3.\end{aligned}$$

- Si elle en **provoque** un, on a plutôt

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\ &= u_i + (2u_i - s_i) - (2u_{i-1} - s_{i-1}) \\ &= u_i + (2u_i - 2(u_i - 1)) - (2(u_i - 1) - (u_i - 1)) \\ &= u_i + 2 - (u_i - 1)\end{aligned}$$

Coût amorti d'un redimensionnement

- Si la *i*-ème **insertion** ne provoque pas de **redimensionnement**, alors

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\ &= 1 + (2u_i - s_i) - (2u_{i-1} - s_{i-1}) \\ &= 1 + (2u_i - s_i) - (2(u_i - 1) - s_i) \\ &= 3.\end{aligned}$$

- Si elle en **provoque** un, on a plutôt

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\ &= u_i + (2u_i - s_i) - (2u_{i-1} - s_{i-1}) \\ &= u_i + (2u_i - 2(u_i - 1)) - (2(u_i - 1) - (u_i - 1)) \\ &= u_i + 2 - (u_i - 1) \\ &= 3.\end{aligned}$$

Tableaux dynamiques

- En résumé, le coût **amorti** d'une **insertion** dans un tableau **dynamique** est constant ;

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to ir the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in *amortized constant time*, that linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.

Each `ArrayList` instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

An application can increase the capacity of an `ArrayList` instance before adding a large number of elements using the `ensureCapacity` operation.

Tableaux dynamiques

- En résumé, le coût **amorti** d'une **insertion** dans un tableau **dynamique** est constant ;
- En pratique, c'est souvent **implémenté**.

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to ir the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in *amortized constant time*, that linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.

Each `ArrayList` instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

An application can increase the capacity of an `ArrayList` instance before adding a large number of elements using the `ensureCapacity` operation.

Table des matières

Définition

- C'est une **structure de données** qui représente une **collection** d'**ensembles dynamiques**

Définition

- C'est une **structure de données** qui représente une **collection** d'**ensembles dynamiques**

$$\mathcal{S} = \{S_1, S_2, \ldots, S_k\}.$$

Définition

- C'est une **structure de données** qui représente une **collection** d'**ensembles dynamiques**

$$\mathcal{S} = \{S_1, S_2, \ldots, S_k\}.$$

- Chaque **ensemble** est identifié par un **représentant** ;

Définition

- C'est une **structure de données** qui représente une **collection** d'**ensembles dynamiques**

$$\mathcal{S} = \{S_1, S_2, \ldots, S_k\}.$$

- Chaque **ensemble** est identifié par un **représentant** ;
- Dans certains cas, le représentant peut être **quelconque** ;

Définition

- C'est une **structure de données** qui représente une **collection** d'**ensembles dynamiques**

$$\mathcal{S} = \{S_1, S_2, \ldots, S_k\}.$$

- Chaque **ensemble** est identifié par un **représentant** ;
- Dans certains cas, le représentant peut être **quelconque** ;
- Parfois, le représentant est **prescrit**, comme l'élément **minimum**.

Opérations

On considère **trois** opérations de base :

Opérations

On considère **trois** opérations de base :

- CréerEnsemble(x) construit un nouvel ensemble d'**un seul** élément, qui est x. À ce moment-là, le **représentant** est x ;

Opérations

On considère **trois** opérations de base :

- CréerEnsemble(x) construit un nouvel ensemble d'**un seul** élément, qui est x. À ce moment-là, le **représentant** est x ;
- Réunir(x, y) **fusionne** les ensembles S_x et S_y qui contiennent respectivement les éléments x et y. Le **représentant** de $S_x \cup S_y$ est alors n'importe quel élément de cet ensemble.

Opérations

On considère **trois** opérations de base :

- CréerEnsemble(x) construit un nouvel ensemble d'**un seul** élément, qui est x. À ce moment-là, le **représentant** est x ;
- Réunir(x, y) **fusionne** les ensembles S_x et S_y qui contiennent respectivement les éléments x et y. Le **représentant** de $S_x \cup S_y$ est alors n'importe quel élément de cet ensemble.
- Représentant(x) retourne le **représentant** de l'ensemble qui contient x.

Analyse des ensembles disjoints

- Lorsqu'on analyse la **structure de données** d'ensembles disjoints, on se base sur 2 paramètres ;

Analyse des ensembles disjoints

- Lorsqu'on analyse la **structure de données** d'ensembles disjoints, on se base sur 2 paramètres ;
- On désigne par n le **nombre d'éléments** qui forment les ensembles disjoints au début ;

Analyse des ensembles disjoints

- Lorsqu'on analyse la **structure de données** d'ensembles disjoints, on se base sur 2 paramètres ;
- On désigne par n le **nombre d'éléments** qui forment les ensembles disjoints au début ;
- Ensuite, on dénote par m le nombre d'opérations, incluant le coût n d'**initialisation** ;

Analyse des ensembles disjoints

- Lorsqu'on analyse la **structure de données** d'ensembles disjoints, on se base sur 2 paramètres ;
- On désigne par n le **nombre d'éléments** qui forment les ensembles disjoints au début ;
- Ensuite, on dénote par m le nombre d'opérations, incluant le coût n d'**initialisation** ;
- En particulier, on effectue **au plus** $n - 1$ opérations RÉUNIR puisqu'alors il ne reste qu'un **unique** ensemble.

Composantes connexes d'un graphe

- Supposons que nous avons calculé les **composantes connexes** d'un graphe non orienté ;

Composantes connexes d'un graphe

- Supposons que nous avons calculé les **composantes connexes** d'un graphe non orienté ;
- On aimerait savoir si les sommets u et v appartiennent à la **même composante** ;

Composantes connexes d'un graphe

- Supposons que nous avons calculé les **composantes connexes** d'un graphe non orienté ;
- On aimerait savoir si les sommets u et v appartiennent à la **même composante** ;
- Un algorithme **inefficace** consiste à vérifier si $\{u, v\}$ **est un sous-ensemble** de chacune des composantes connexes ;

Composantes connexes d'un graphe

- Supposons que nous avons calculé les **composantes connexes** d'un graphe non orienté ;
- On aimerait savoir si les sommets u et v appartiennent à la **même composante** ;
- Un algorithme **inefficace** consiste à vérifier si $\{u, v\}$ **est un sous-ensemble** de chacune des composantes connexes ;
- On peut faire beaucoup mieux en utilisant les **ensembles disjoints**.

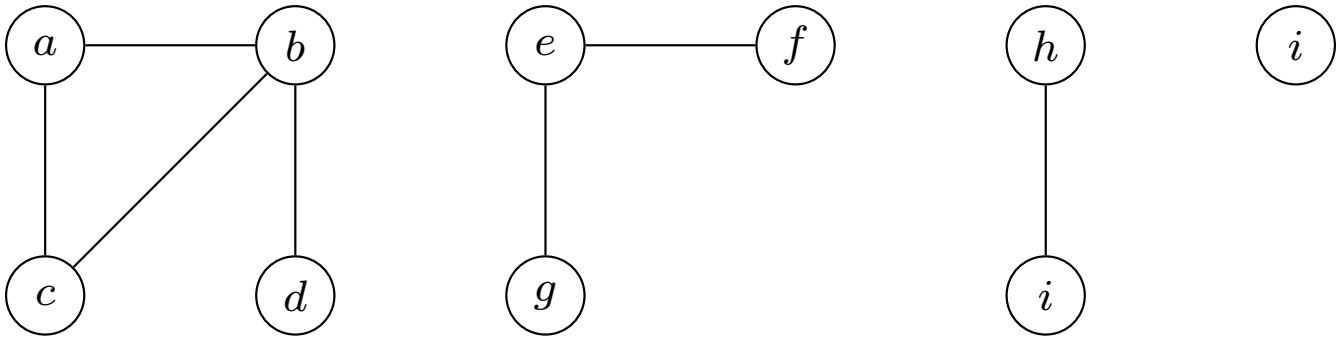
Pseudocode

```
1: fonction ComposantesConnexes(G : graphe) : ensembles disjoints
2:     Soit D une structure de données d'ensembles disjoints
3:     pour v ∈ G.V faire
4:         D.CréerEnsemble(v)
5:     fin pour
6:     pour (u, v) ∈ G.edges faire
7:         si D.Représentant(u) = D.Représentant(v) alors
8:             D.Réunir(u, v)
9:         fin si
10:    fin pour
11:    retourner D
12: fin fonction
```

Pseudocode

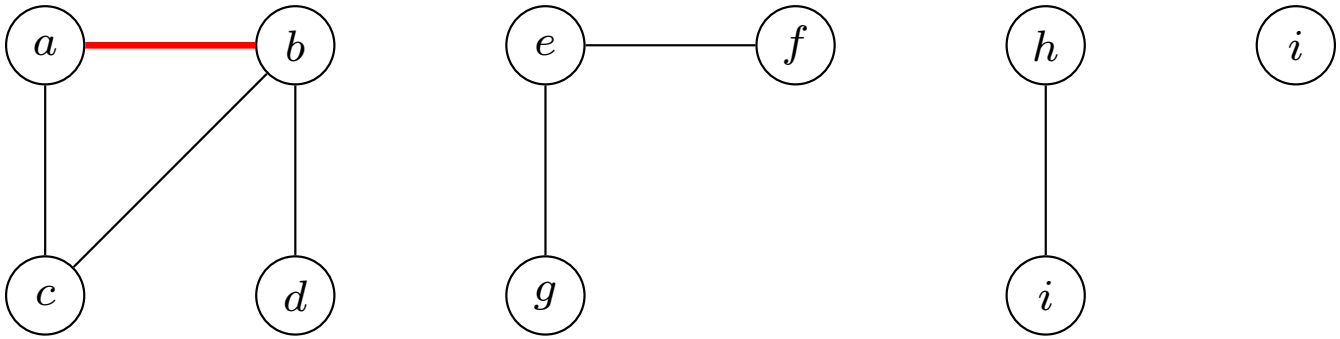
```
1: fonction ComposantesConnexes(G : graphe) : ensembles disjoints
2:     Soit D une structure de données d'ensembles disjoints
3:     pour v ∈ G.V faire
4:         D.CréerEnsemble(v)
5:     fin pour
6:     pour (u, v) ∈ G.edges faire
7:         si D.Représentant(u) = D.Représentant(v) alors
8:             D.Réunir(u, v)
9:         fin si
10:    fin pour
11:    retourner D
12: fin fonction

13: fonction MêmeComposante(D : ensembles disjoints, u, v : sommets)
14:    si D.Représentant(u) = D.Représentant(v) alors
15:        retourner vrai
16:    sinon
17:        retourner faux
18:    fin si
19: fin fonction
```

Exemple



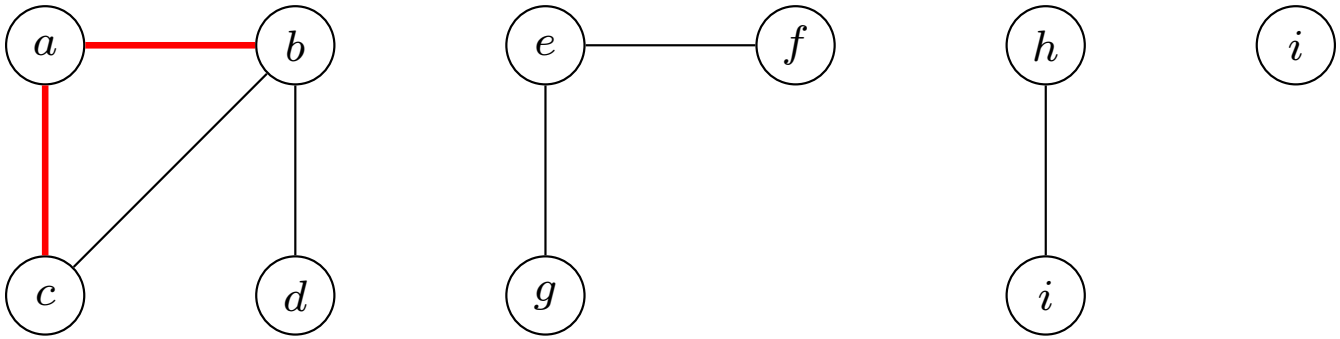
$$\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}, \{j\}$$

Exemple



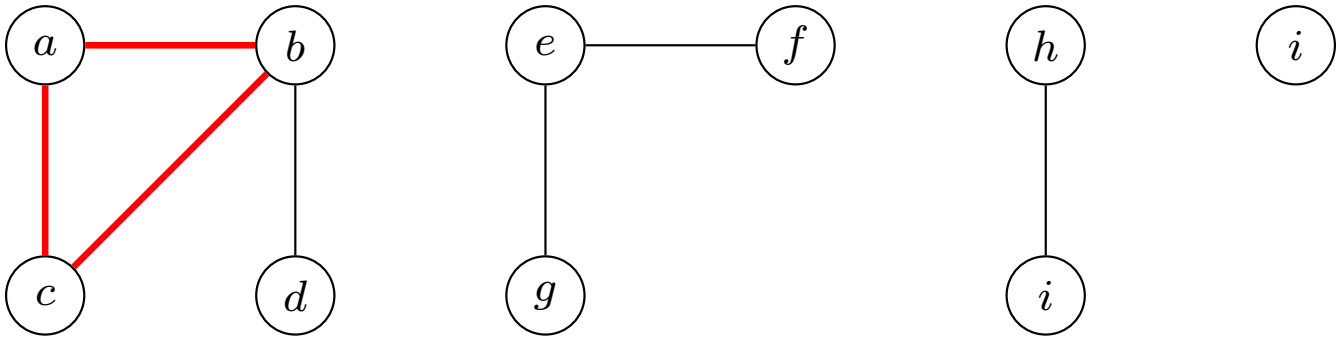
$$\{a, b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}, \{j\}$$

Exemple



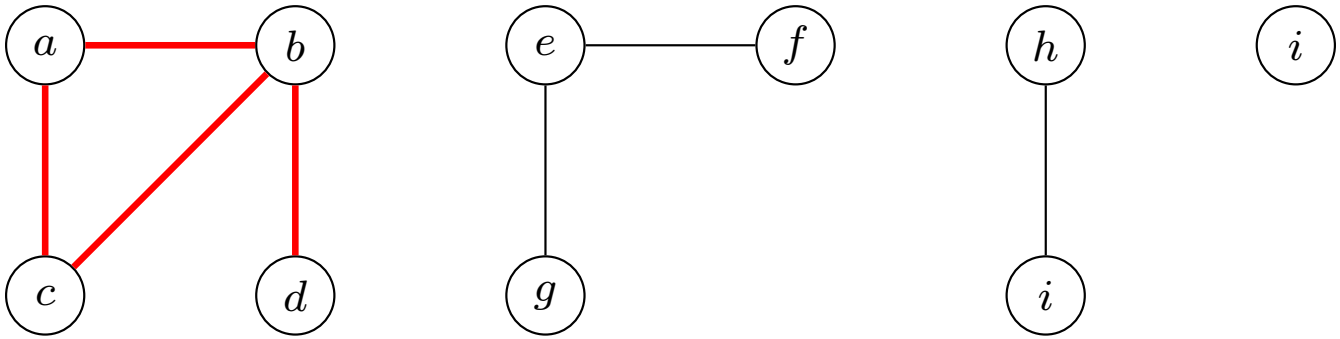
$$\{a, b, c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}, \{j\}$$

Exemple



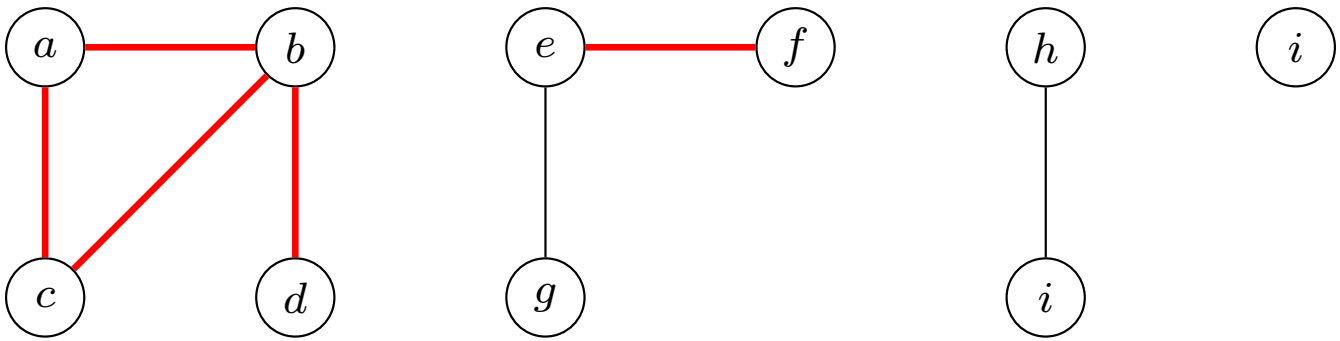
$$\{a, b, c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}, \{j\}$$

Exemple



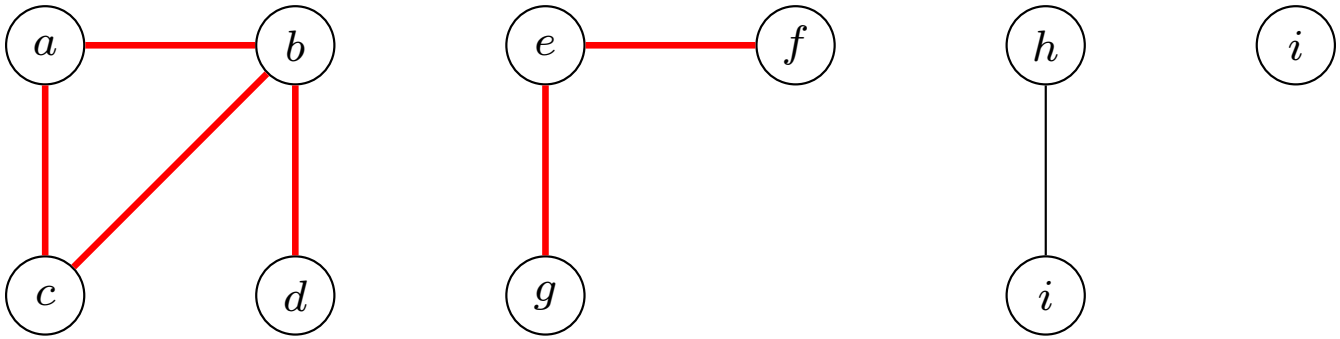
$$\{a, b, c, d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}, \{j\}$$

Exemple



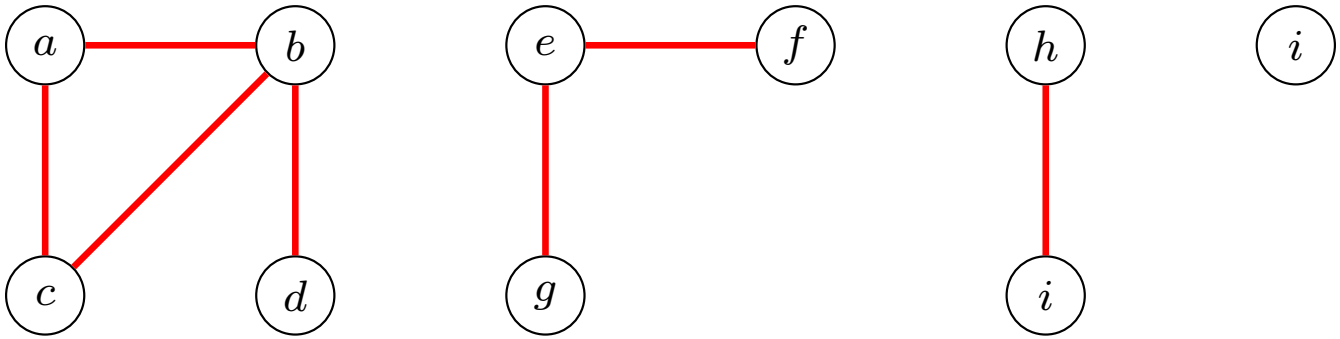
$$\{a, b, c, d\}, \{e, f\}, \{g\}, \{h\}, \{i\}, \{j\}$$

Exemple



$$\{a, b, c, d\}, \{e, f, g\}, \{h\}, \{i\}, \{j\}$$

Exemple



$$\{a, b, c, d\}, \{e, f, g\}, \{h, i\}, \{j\}$$

Représentation par listes chaînées

- La façon la plus **simple** de représenter des **ensembles disjoints** est de représenter chaque ensemble par une **liste chaînée** ;

Représentation par listes chaînées

- La façon la plus **simple** de représenter des **ensembles disjoints** est de représenter chaque ensemble par une **liste chaînée** ;
- Chaque élément pointe

Représentation par listes chaînées

- La façon la plus **simple** de représenter des **ensembles disjoints** est de représenter chaque ensemble par une **liste chaînée** ;
- Chaque élément pointe
  - vers le **suivant** et

Représentation par listes chaînées

- La façon la plus **simple** de représenter des **ensembles disjoints** est de représenter chaque ensemble par une **liste chaînée** ;
- Chaque élément pointe
  - vers le **suivant** et
  - vers l'**ensemble** ;

Représentation par listes chaînées

- La façon la plus **simple** de représenter des **ensembles disjoints** est de représenter chaque ensemble par une **liste chaînée** ;
- Chaque élément pointe
  - vers le **suivant** et
  - vers l'**ensemble** ;
- Le **représentant** est le **premier élément** de la liste ;

Représentation par listes chaînées

- La façon la plus **simple** de représenter des **ensembles disjoints** est de représenter chaque ensemble par une **liste chaînée** ;
- Chaque élément pointe
  - vers le **suivant** et
  - vers l'**ensemble** ;
- Le **représentant** est le **premier élément** de la liste ;
- Les éléments d'une **même liste** peuvent apparaître dans n'**importe quel ordre**.

Représentation par listes chaînées (suite)

- L'opération CréerEnsemble est réalisée en temps $\mathcal{O}(1)$: initialisation d'une liste vide ;

Représentation par listes chaînées (suite)

- L'opération CréerEnsemble est réalisée en temps $\mathcal{O}(1)$: initialisation d'une liste vide ;
- L'opération Représentant est aussi $\mathcal{O}(1)$, puisqu'il suffit, pour chaque élément, de consulter le **pointeur** vers l'ensemble et ensuite accéder au **premier élément** de l'ensemble ;

Représentation par listes chaînées (suite)

- L'opération CréerEnsemble est réalisée en temps $\mathcal{O}(1)$: initialisation d'une liste vide ;
- L'opération Représentant est aussi $\mathcal{O}(1)$, puisqu'il suffit, pour chaque élément, de consulter le **pointeur** vers l'ensemble et ensuite accéder au **premier élément** de l'ensemble ;
- L'opération Réunir est **plus coûteuse** :

Représentation par listes chaînées (suite)

- L'opération CréerEnsemble est réalisée en temps $\mathcal{O}(1)$: initialisation d'une liste vide ;
- L'opération Représentant est aussi $\mathcal{O}(1)$, puisqu'il suffit, pour chaque élément, de consulter le **pointeur** vers l'ensemble et ensuite accéder au **premier élément** de l'ensemble ;
- L'opération Réunir est **plus coûteuse** :
  - On **fusionne** les deux listes en $\mathcal{O}(1)$;

Représentation par listes chaînées (suite)

- L'opération CréerEnsemble est réalisée en temps $\mathcal{O}(1)$: initialisation d'une liste vide ;
- L'opération Représentant est aussi $\mathcal{O}(1)$, puisqu'il suffit, pour chaque élément, de consulter le **pointeur** vers l'ensemble et ensuite accéder au **premier élément** de l'ensemble ;
- L'opération Réunir est **plus coûteuse** :
  - On **fusionne** les deux listes en $\mathcal{O}(1)$;
  - On doit mettre à jour les **pointeurs vers le représentant** en $\mathcal{O}(n)$;

Représentation par listes chaînées (suite)

- L'opération CréerEnsemble est réalisée en temps $\mathcal{O}(1)$: initialisation d'une liste vide ;
- L'opération Représentant est aussi $\mathcal{O}(1)$, puisqu'il suffit, pour chaque élément, de consulter le **pointeur** vers l'ensemble et ensuite accéder au **premier élément** de l'ensemble ;
- L'opération Réunir est **plus coûteuse** :
  - On **fusionne** les deux listes en $\mathcal{O}(1)$;
  - On doit mettre à jour les **pointeurs vers le représentant** en $\mathcal{O}(n)$;
- En **annexant** la liste la **plus courte** à la liste la plus longue, on peut améliorer légèrement la complexité.

Analyse de la représentation par liste chaînée

Théorème

En implémentant des ensembles disjoints à l'aide de **listes chaînées** et en favorisant la **fusion** de la plus courte liste avec la plus longue, si on effectue une suite de m **opérations** parmi CréerEnsemble, Représentant et Réunir, dont n **d'entre elles** sont des opérations CréerEnsemble, alors le temps total pris est $\mathcal{O}(m + n \log n)$.

Analyse de la représentation par liste chaînée

Théorème

En implémentant des ensembles disjoints à l'aide de **listes chaînées** et en favorisant la **fusion** de la plus courte liste avec la plus longue, si on effectue une suite de m **opérations** parmi CréerEnsemble, Représentant et Réunir, dont n **d'entre elles** sont des opérations CréerEnsemble, alors le temps total pris est $\mathcal{O}(m + n \log n)$.

- Nous allons voir plus tard qu'on peut faire **mieux** en utilisant une représentation par **forêt** ;

Table des matières

Retour sur les ensembles disjoints

- Nous avons vu **plus tôt** qu'il était possible d'implémenter les ensembles disjoints à l'aide de **listes chaînées** ;

Retour sur les ensembles disjoints

- Nous avons vu **plus tôt** qu'il était possible d'implémenter les ensembles disjoints à l'aide de **listes chaînées** ;
- Une représentation **beaucoup plus efficace** consiste à représenter les ensembles par des **arbres** dont la **racine** est le **représentant** ;

Retour sur les ensembles disjoints

- Nous avons vu **plus tôt** qu'il était possible d'implémenter les ensembles disjoints à l'aide de **listes chaînées** ;
- Une représentation **beaucoup plus efficace** consiste à représenter les ensembles par des **arbres** dont la **racine** est le **représentant** ;
- On obtient alors les **complexités** suivantes pour les opérations :

Retour sur les ensembles disjoints

- Nous avons vu **plus tôt** qu'il était possible d'implémenter les ensembles disjoints à l'aide de **listes chaînées** ;
- Une représentation **beaucoup plus efficace** consiste à représenter les ensembles par des **arbres** dont la **racine** est le **représentant** ;
- On obtient alors les **complexités** suivantes pour les opérations :
  - CréerEnsemble en temps $\mathcal{O}(1)$;

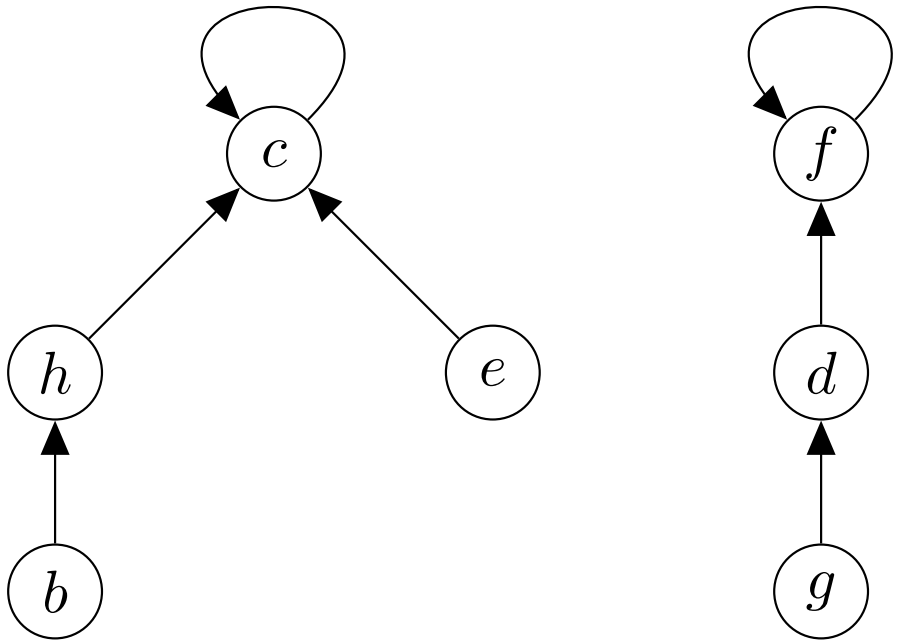
Retour sur les ensembles disjoints

- Nous avons vu **plus tôt** qu'il était possible d'implémenter les ensembles disjoints à l'aide de **listes chaînées** ;
- Une représentation **beaucoup plus efficace** consiste à représenter les ensembles par des **arbres** dont la **racine** est le **représentant** ;
- On obtient alors les **complexités** suivantes pour les opérations :
  - CréerEnsemble en temps $\mathcal{O}(1)$;
  - Réunir en temps $\mathcal{O}(1)$ et

Retour sur les ensembles disjoints

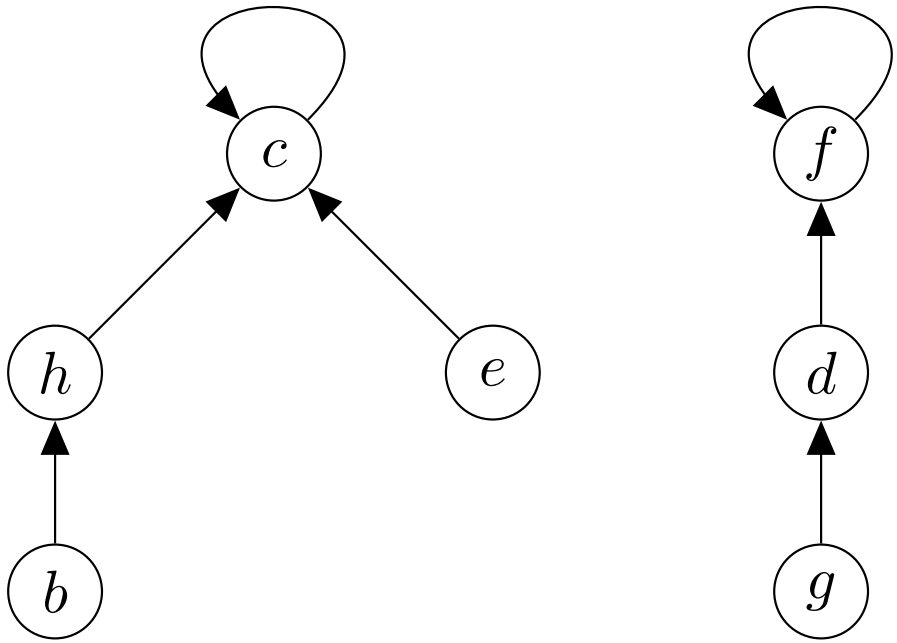
- Nous avons vu **plus tôt** qu'il était possible d'implémenter les ensembles disjoints à l'aide de **listes chaînées** ;
- Une représentation **beaucoup plus efficace** consiste à représenter les ensembles par des **arbres** dont la **racine** est le **représentant** ;
- On obtient alors les **complexités** suivantes pour les opérations :
  - CréerEnsemble en temps $\mathcal{O}(1)$;
  - Réunir en temps $\mathcal{O}(1)$ et
  - Représentant en temps **amorti** $\mathcal{O}(\alpha(n))$, où $\alpha(n)$ est une fonction qui **croît très lentement**.

Exemple



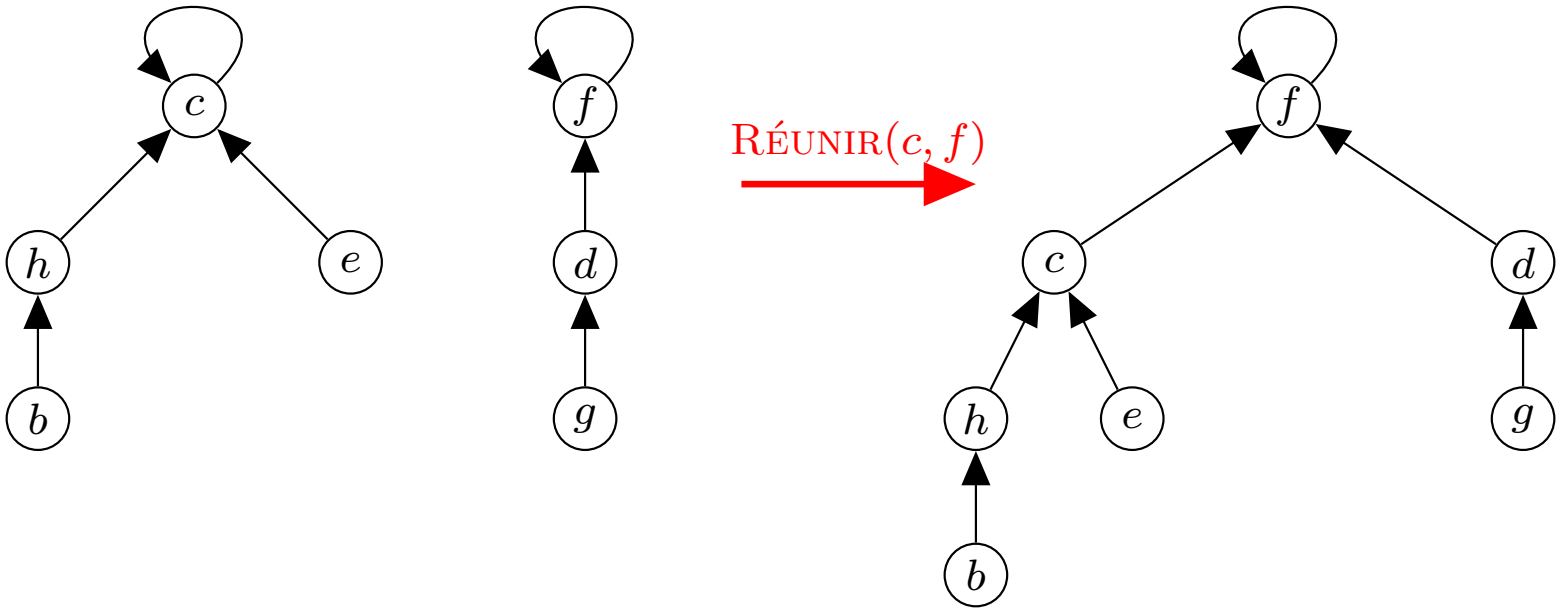
- On représente les **ensembles disjoints** $\{b, c, e, h\}$ et $\{d, f, g\}$ par une **forêt** ;

Exemple



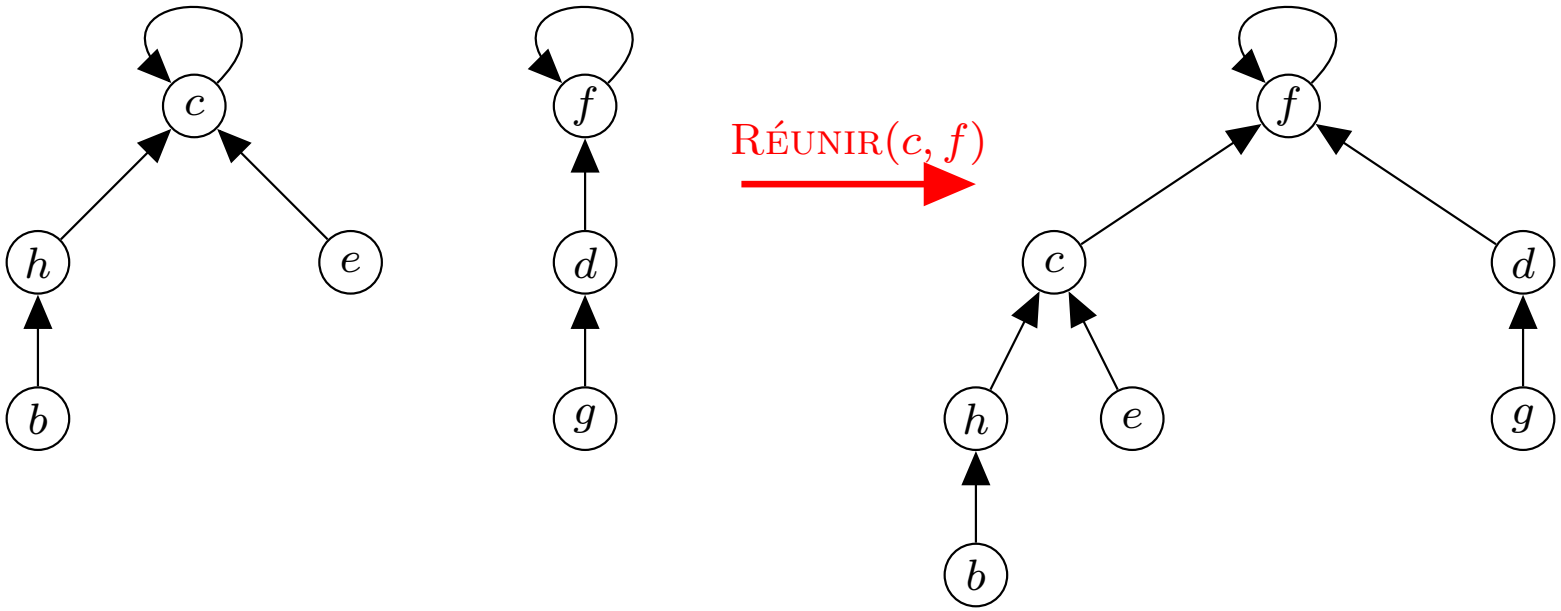
- On représente les **ensembles disjoints** $\{b, c, e, h\}$ et $\{d, f, g\}$ par une **forêt** ;
- Les **représentants** sont c **et** f.

Opérations



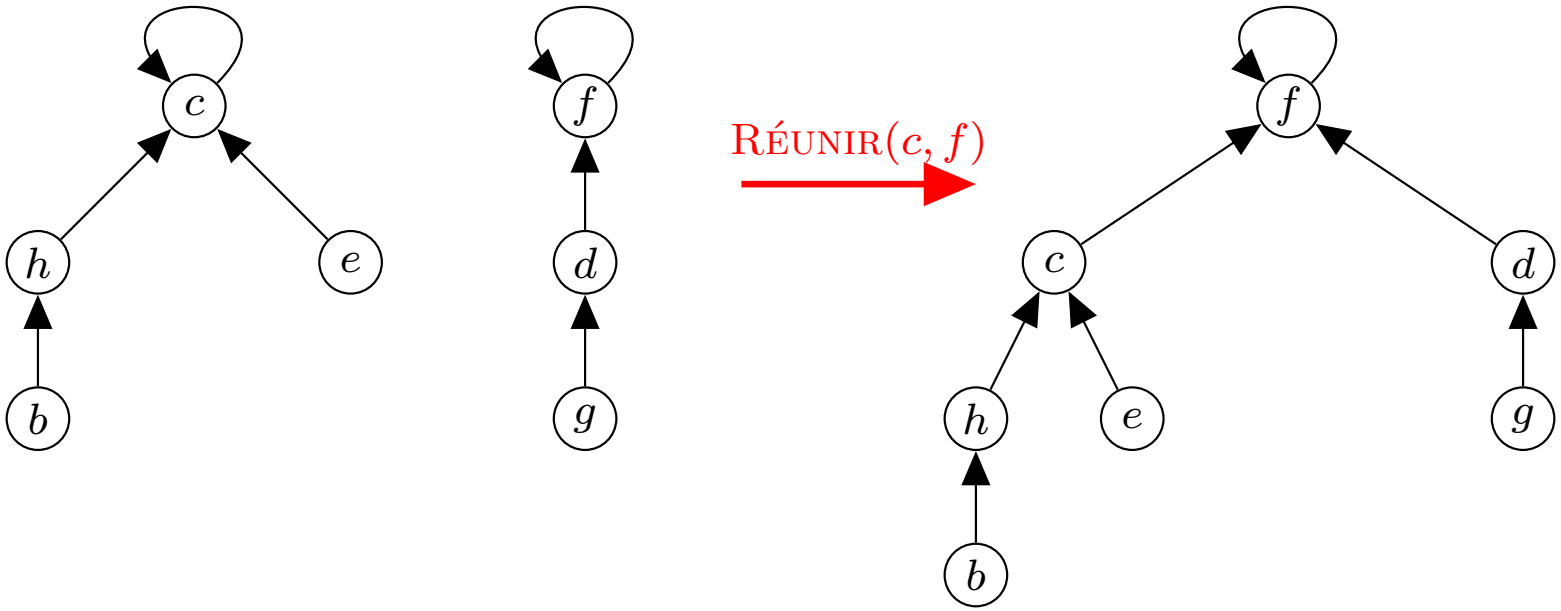
- La fonction CRÉERENSEMBLE est implémentée en créant un noeud qui **pointe vers lui-même** ;

Opérations



- La fonction CRÉERENSEMBLE est implémentée en créant un noeud qui **pointe vers lui-même** ;
- Pour chaque noeud, on calcule le **représentant** en remontant jusqu'à la **racine** ;

Opérations



- La fonction CRÉERENSEMBLE est implémentée en créant un noeud qui **pointe vers lui-même** ;
- Pour chaque noeud, on calcule le **représentant** en remontant jusqu'à la **racine** ;
- La **réunion** se fait en rattachant une des **racines** à l'autre.

Deux heuristiques pour améliorer la complexité

- Les **arbres** de la forêt peuvent être **profonds** ;

Deux heuristiques pour améliorer la complexité

- Les **arbres** de la forêt peuvent être **profonds** ;
- Pour **éviter cette situation**, on introduit **deux heuristiques** ;

Deux heuristiques pour améliorer la complexité

- Les **arbres** de la forêt peuvent être **profonds** ;
- Pour **éviter cette situation**, on introduit **deux heuristiques** ;
- La première consiste à **compresser le chemin** de chaque valeur sur laquelle on effectue une **requête** ;

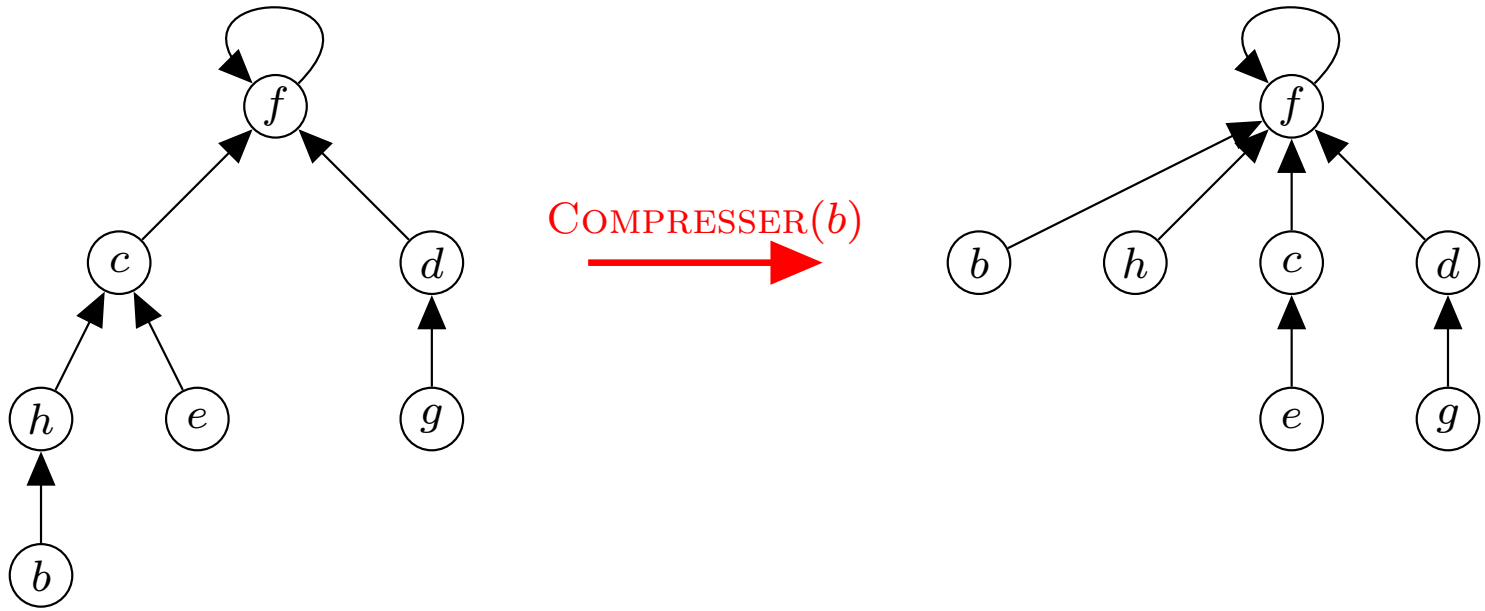
Deux heuristiques pour améliorer la complexité

- Les **arbres** de la forêt peuvent être **profonds** ;
- Pour **éviter cette situation**, on introduit **deux heuristiques** ;
- La première consiste à **compresser le chemin** de chaque valeur sur laquelle on effectue une **requête** ;
- La seconde **favorise** la racine de l'arbre le **moins compressé** comme **nouvelle racine** lors de la **réunion** ;

Deux heuristiques pour améliorer la complexité

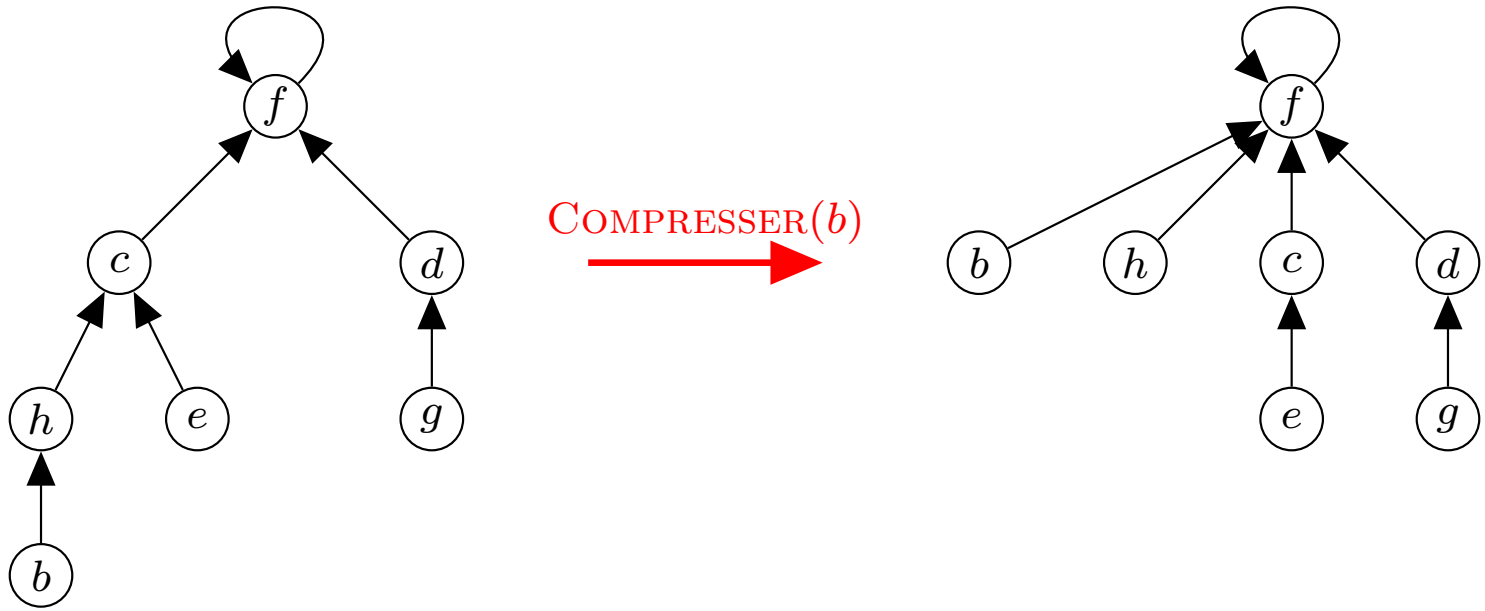
- Les **arbres** de la forêt peuvent être **profonds** ;
- Pour **éviter cette situation**, on introduit **deux heuristiques** ;
- La première consiste à **compresser le chemin** de chaque valeur sur laquelle on effectue une **requête** ;
- La seconde **favorise** la racine de l'arbre le **moins compressé** comme **nouvelle racine** lors de la **réunion** ;
- À noter que les **deux heuristiques** sont nécessaires pour que la complexité soit $\mathcal{O}(\alpha(n))$.

Heuristique 1 : Compression de chemin



- Supposons qu'on **interroge** la structure de données pour calculer le **représentant de** b ;

Heuristique 1 : Compression de chemin



- Supposons qu'on **interroge** la structure de données pour calculer le **représentant de** b ;
- Alors on en **profite** pour faire pointer **directement** vers f chacun des noeuds sur le **chemin de** b.

Heuristique 2 : Réunir par rang

- La deuxième heuristique permet de **réunir** deux arbres de telle sorte que le résultat ne soit pas un arbre **trop profond** ;

Heuristique 2 : Réunir par rang

- La deuxième heuristique permet de **réunir** deux arbres de telle sorte que le résultat ne soit pas un arbre **trop profond** ;
- Plus précisément, on **maintient** un attribut **rang** pour chaque noeud qui donne une **borne supérieure** sur sa **hauteur** ;

Heuristique 2 : Réunir par rang

- La deuxième heuristique permet de **réunir** deux arbres de telle sorte que le résultat ne soit pas un arbre **trop profond** ;
- Plus précisément, on **maintient** un attribut **rang** pour chaque noeud qui donne une **borne supérieure** sur sa **hauteur** ;
- Lors de la fusion, on fait pointer la racine du **plus petit rang** vers celle du **plus grand rang** ;

Heuristique 2 : Réunir par rang

- La deuxième heuristique permet de **réunir** deux arbres de telle sorte que le résultat ne soit pas un arbre **trop profond** ;
- Plus précisément, on **maintient** un attribut **rang** pour chaque noeud qui donne une **borne supérieure** sur sa **hauteur** ;
- Lors de la fusion, on fait pointer la racine du **plus petit rang** vers celle du **plus grand rang** ;
- Dans le cas où il y a **égalité des rangs**, alors on prend n'importe quelle racine et on **met à jour le rang**.

Pseudocode

```
1: procedure CréerEnsemble(x : élément)
2:     x.parent ← x
3:     x.rang ← 0
4: fin procedure

5: procedure Réunir(x, y : éléments)
6:     x ← Représentant(x)
7:     y ← Représentant(y)
8:     si x.rang > y.rang alors
9:         y.parent ← x
10:    sinon
11:        x.parent ← y
12:        si x.rang = y.rang alors
13:            y.rang ← y.rang + 1
14:        fin si
15:    fin si
16: fin procedure

17: fonction Représentant(x : élément)
18:    si x ≠ x.parent alors
19:        x.parent ← Représentant(x.parent)
20:    fin si
21:    retourner x.parent
22: fin fonction
```

Table des matières

Arbre splay (*splay tree*)

- Il s'agit d'un **arbre binaire** ;

Arbre splay (*splay tree*)

- Il s'agit d'un **arbre binaire** ;
- Plutôt que de **maintenir une hauteur bornée** à chaque étape, on accepte que l'arbre ne soit **pas toujours équilibré** ;

Arbre splay (*splay tree*)

- Il s'agit d'un **arbre binaire** ;
- Plutôt que de **maintenir une hauteur bornée** à chaque étape, on accepte que l'arbre ne soit **pas toujours équilibré** ;
- Par contre, aussitôt qu'un **noeud** est **consulté**, **inséré** ou **supprimé**, on le fait remonter à la **racine** par une suite de **rotations** ;

Arbre splay (*splay tree*)

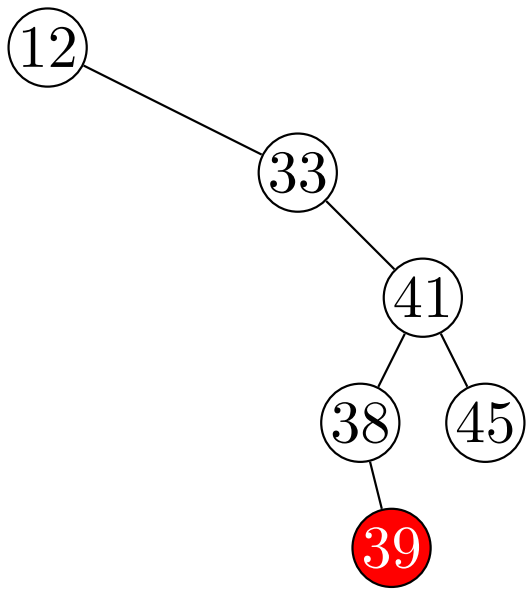
- Il s'agit d'un **arbre binaire** ;
- Plutôt que de **maintenir une hauteur bornée** à chaque étape, on accepte que l'arbre ne soit **pas toujours équilibré** ;
- Par contre, aussitôt qu'un **noeud** est **consulté**, **inséré** ou **supprimé**, on le fait remonter à la **racine** par une suite de **rotations** ;
- Structure de données introduite par **Tarjan et Sleator** en **1985**.

Démonstration

http://www.ibr.cs.tu-bs.de/courses/ss98/audii/applets/BST/SplayTree-Example.html

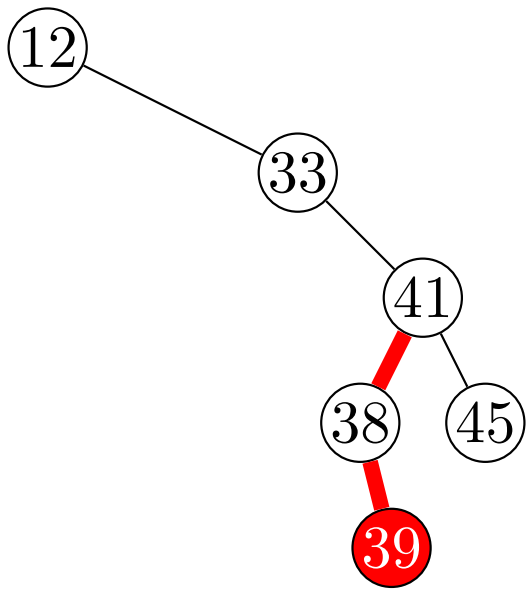
Déploiement (*splaying*)

- L'opération de base des **arbres splay** est appelée **déploiement** (*splaying*) ;
- Il s'agit de faire **remonter** un noeud jusqu'à la racine par une suite de **rotations** ;



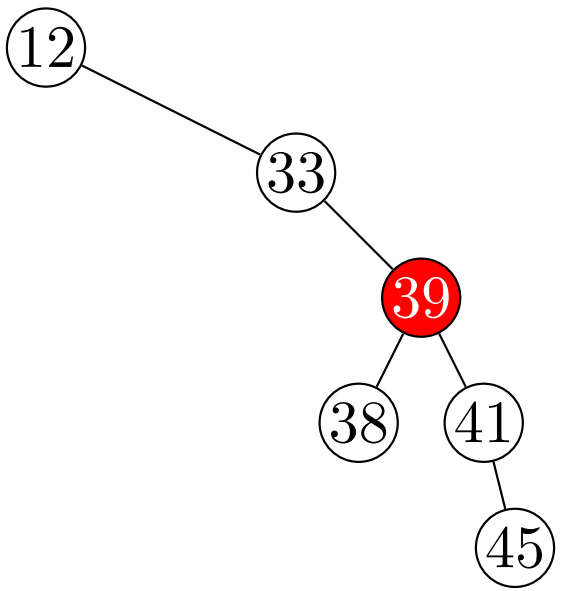
Déploiement (*splaying*)

- L'opération de base des **arbres splay** est appelée **déploiement** (*splaying*) ;
- Il s'agit de faire **remonter** un noeud jusqu'à la racine par une suite de **rotations** ;



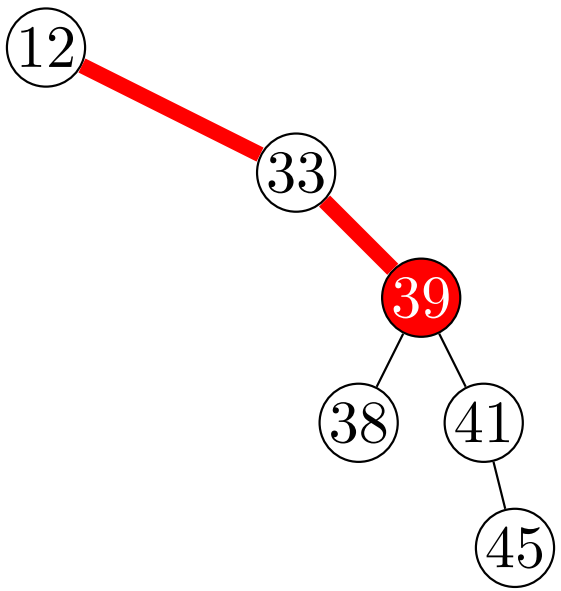
Déploiement (*splaying*)

- L'opération de base des **arbres splay** est appelée **déploiement** (*splaying*) ;
- Il s'agit de faire **remonter** un noeud jusqu'à la racine par une suite de **rotations** ;



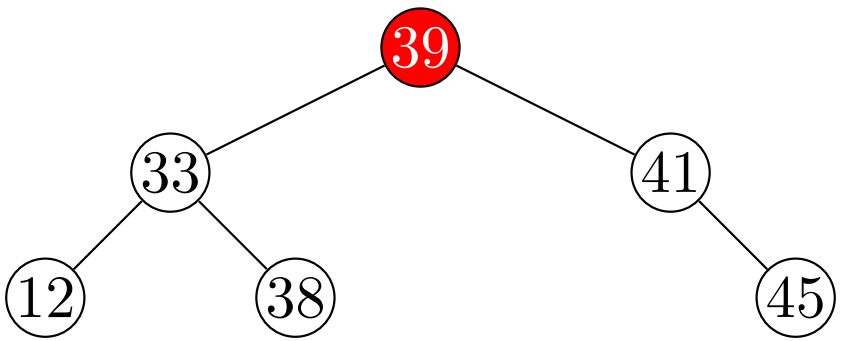
Déploiement (*splaying*)

- L'opération de base des **arbres splay** est appelée **déploiement** (*splaying*) ;
- Il s'agit de faire **remonter** un noeud jusqu'à la racine par une suite de **rotations** ;



Déploiement (*splaying*)

- L'opération de base des **arbres splay** est appelée **déploiement** (*splaying*) ;
- Il s'agit de faire **remonter** un noeud jusqu'à la racine par une suite de **rotations** ;



Autres opérations

- Lorsqu'une **fouille** est effectuée, l'algorithme est le même, mais on **déploie** (*splay*) le noeud **final**, que la requête ait abouti ou non ;

Autres opérations

- Lorsqu'une **fouille** est effectuée, l'algorithme est le même, mais on **déploie** (*splay*) le noeud **final**, que la requête ait abouti ou non ;
- Lors d'une **insertion**, l'algorithme de base est le même, mais on **déploie** le noeud **inséré** ;

Autres opérations

- Lorsqu'une **fouille** est effectuée, l'algorithme est le même, mais on **déploie** (*splay*) le noeud **final**, que la requête ait abouti ou non ;
- Lors d'une **insertion**, l'algorithme de base est le même, mais on **déploie** le noeud **inséré** ;
- Lors d'une suppression, il y a deux stratégies possibles :

Autres opérations

- Lorsqu'une **fouille** est effectuée, l'algorithme est le même, mais on **déploie** (*splay*) le noeud **final**, que la requête ait abouti ou non ;
- Lors d'une **insertion**, l'algorithme de base est le même, mais on **déploie** le noeud **inséré** ;
- Lors d'une suppression, il y a deux stratégies possibles :
  - **Déployer** le parent du noeud supprimé ;

Autres opérations

- Lorsqu'une **fouille** est effectuée, l'algorithme est le même, mais on **déploie** (*splay*) le noeud **final**, que la requête ait abouti ou non ;
- Lors d'une **insertion**, l'algorithme de base est le même, mais on **déploie** le noeud **inséré** ;
- Lors d'une suppression, il y a deux stratégies possibles :
  - **Déployer** le parent du noeud supprimé ;
  - **Déployer** le noeud à supprimer, puis ensuite le supprimer.

Fonction potentiel

- Soit T un **arbre binaire** ;

Fonction potentiel

- Soit T un **arbre binaire** ;
- Pour chaque noeud x de T, on définit

Fonction potentiel

- Soit T un **arbre binaire** ;
- Pour chaque noeud x de T, on définit
  - le **poids** $w(x)$ comme le **nombre de noeuds** dans le sous-arbre induit par x ;

Fonction potentiel

- Soit T un **arbre binaire** ;
- Pour chaque noeud x de T, on définit
  - le **poids** $w(x)$ comme le **nombre de noeuds** dans le sous-arbre induit par x ;
  - le **rang** $r(x) = \log(w(x))$.

Fonction potentiel

- Soit T un **arbre binaire** ;
- Pour chaque noeud x de T, on définit
  - le **poids** $w(x)$ comme le **nombre de noeuds** dans le sous-arbre induit par x ;
  - le **rang** $r(x) = \log(w(x))$.
- Le **potentiel** de T est alors défini par

$$\Phi(T) = \sum_{x \in T} r(x).$$

Exercice

Calculer le potentiel d'un arbre de 7 noeuds qui est

1. parfaitement équilibré ;
2. complètement déséquilibré.

Coût amorti des rotations

- On analyse ce qui se passe lorsqu'une **rotation** est effectuée ;

Coût amorti des rotations

- On analyse ce qui se passe lorsqu'une **rotation** est effectuée ;
- Soit

Coût amorti des rotations

- On analyse ce qui se passe lorsqu'une **rotation** est effectuée ;
- Soit
  - T l'arbre **avant** la rotation ;

Coût amorti des rotations

- On analyse ce qui se passe lorsqu'une **rotation** est effectuée ;
- Soit
  - T l'arbre **avant** la rotation ;
  - T' l'arbre **après** la rotation ;

Coût amorti des rotations

- On analyse ce qui se passe lorsqu'une **rotation** est effectuée ;
- Soit
  - T l'arbre **avant** la rotation ;
  - T' l'arbre **après** la rotation ;
  - $w(x)$ le poids de x **avant** la rotation ;

Coût amorti des rotations

- On analyse ce qui se passe lorsqu'une **rotation** est effectuée ;
- Soit
  - T l'arbre **avant** la rotation ;
  - T' l'arbre **après** la rotation ;
  - $w(x)$ le poids de x **avant** la rotation ;
  - $w'(x)$ le poids de x **après** la rotation ;

Coût amorti des rotations

- On analyse ce qui se passe lorsqu'une **rotation** est effectuée ;
- Soit
  - T l'arbre **avant** la rotation ;
  - T' l'arbre **après** la rotation ;
  - $w(x)$ le poids de x **avant** la rotation ;
  - $w'(x)$ le poids de x **après** la rotation ;
  - $r(x)$ le rang de x **avant** la rotation ;

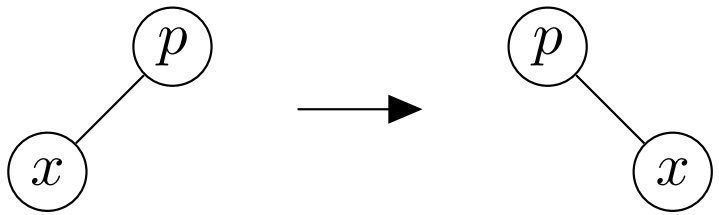
Coût amorti des rotations

- On analyse ce qui se passe lorsqu'une **rotation** est effectuée ;
- Soit
  - T l'arbre **avant** la rotation ;
  - T' l'arbre **après** la rotation ;
  - $w(x)$ le poids de x **avant** la rotation ;
  - $w'(x)$ le poids de x **après** la rotation ;
  - $r(x)$ le rang de x **avant** la rotation ;
  - $r'(x)$ le rang de x **après** la rotation.

Coût amorti des rotations

- On analyse ce qui se passe lorsqu'une **rotation** est effectuée ;
- Soit
  - T l'arbre **avant** la rotation ;
  - T' l'arbre **après** la rotation ;
  - $w(x)$ le poids de x **avant** la rotation ;
  - $w'(x)$ le poids de x **après** la rotation ;
  - $r(x)$ le rang de x **avant** la rotation ;
  - $r'(x)$ le rang de x **après** la rotation.
- Il faut considérer **trois rotations** possibles : (1) zig, (2) zig-zag et (3) zig-zig.

Premier cas : zig



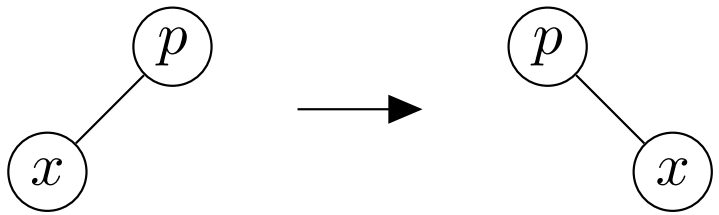
$r'(p) < r(p)$

$r'(x) > r(x)$

Le **coût amorti** d'une **rotation zig** est

$$\widehat{c_{\text{zig}}} = 1 + \Phi(T') - \Phi(T)$$

Premier cas : zig



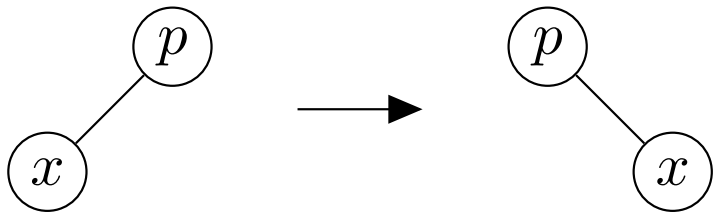
$r'(p) < r(p)$

$r'(x) > r(x)$

Le **coût amorti** d'une **rotation zig** est

$$\begin{aligned} \widehat{c_{\text{zig}}} &= 1 + \Phi(T') - \Phi(T) \\ &= 1 + (r'(x) + r'(p)) - (r(x) + r(p)) \end{aligned}$$

Premier cas : zig



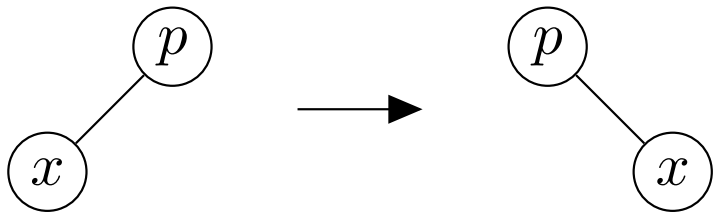
$r'(p) < r(p)$

$r'(x) > r(x)$

Le **coût amorti** d'une **rotation zig** est

$$\begin{aligned}
\widehat{c_{\text{zig}}} &= 1 + \Phi(T') - \Phi(T) \\
&= 1 + (r'(x) + r'(p)) - (r(x) + r(p)) \\
&= 1 + (r'(x) - r(x)) + (r'(p) - r(p))
\end{aligned}$$

Premier cas : zig



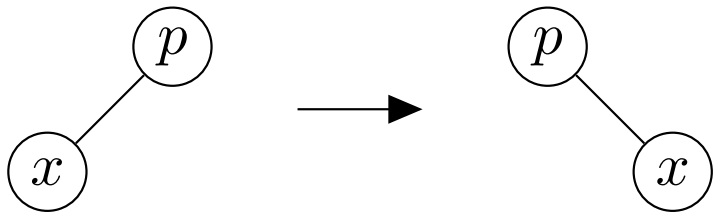
$r'(p) < r(p)$

$r'(x) > r(x)$

Le **coût amorti** d'une **rotation zig** est

$$\begin{aligned}
\widehat{c_{\text{zig}}} &= 1 + \Phi(T') - \Phi(T) \\
&= 1 + (r'(x) + r'(p)) - (r(x) + r(p)) \\
&= 1 + (r'(x) - r(x)) + (r'(p) - r(p)) \\
&\leq 1 + (r'(x) - r(x))
\end{aligned}$$

Premier cas : zig



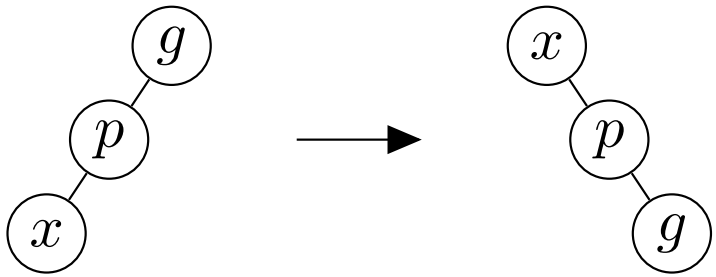
$r'(p) < r(p)$

$r'(x) > r(x)$

Le **coût amorti** d'une **rotation zig** est

$$\begin{aligned}
\widehat{c_{\text{zig}}} &= 1 + \Phi(T') - \Phi(T) \\
&= 1 + (r'(x) + r'(p)) - (r(x) + r(p)) \\
&= 1 + (r'(x) - r(x)) + (r'(p) - r(p)) \\
&\leq 1 + (r'(x) - r(x)) \\
&\leq 1 + 3(r'(x) - r(x)).
\end{aligned}$$

Deuxième cas : zig-zig

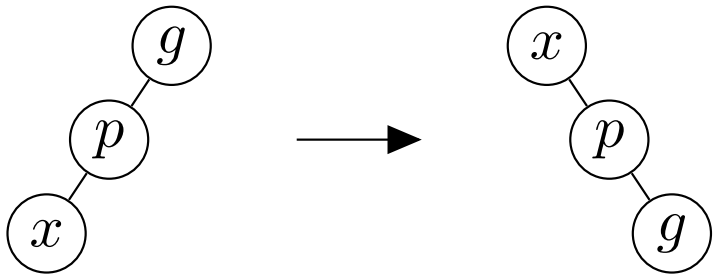


$r'(x) = r(g)$

$r'(x) \geq r'(p)$

$r(p) \geq r(x)$

Deuxième cas : zig-zig



$r'(x) = r(g)$

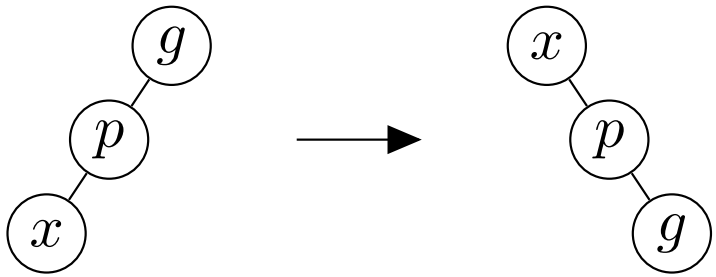
$r'(x) \geq r'(p)$

$r(p) \geq r(x)$

Le **coût amorti** d'une **rotation zig-zig** est

$$\widehat{c_{\text{zig-zig}}} = 2 + \Phi(T') - \Phi(T)$$

Deuxième cas : zig-zig



$r'(x) = r(g)$

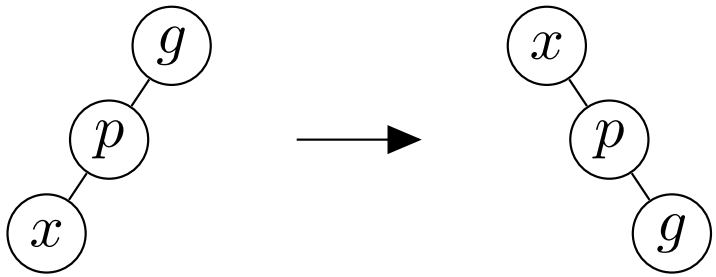
$r'(x) \geq r'(p)$

$r(p) \geq r(x)$

Le **coût amorti** d'une **rotation zig-zig** est

$$\begin{aligned} \widehat{c_{\text{zig-zig}}} &= 2 + \Phi(T') - \Phi(T) \\ &= 2 + [r'(x) + r'(p) + r'(g)] - [r(x) + r(p) + r(g)] \end{aligned}$$

Deuxième cas : zig-zig



$r'(x) = r(g)$

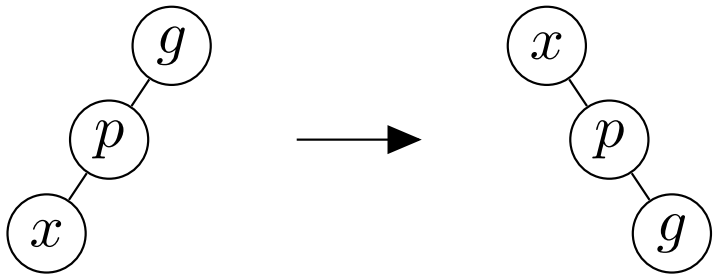
$r'(x) \geq r'(p)$

$r(p) \geq r(x)$

Le **coût amorti** d'une **rotation zig-zig** est

$$\begin{aligned}
\widehat{c_{\text{zig-zig}}} &= 2 + \Phi(T') - \Phi(T) \\
&= 2 + [r'(x) + r'(p) + r'(g)] - [r(x) + r(p) + r(g)] \\
&= 2 + [r'(p) + r'(g)] - [r(x) + r(p)]
\end{aligned}$$

Deuxième cas : zig-zig



$r'(x) = r(g)$

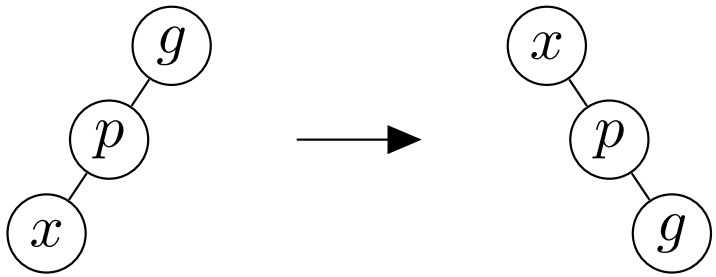
$r'(x) \geq r'(p)$

$r(p) \geq r(x)$

Le **coût amorti** d'une **rotation zig-zig** est

$$\begin{aligned}
\widehat{c_{\text{zig-zig}}} &= 2 + \Phi(T') - \Phi(T) \\
&= 2 + [r'(x) + r'(p) + r'(g)] - [r(x) + r(p) + r(g)] \\
&= 2 + [r'(p) + r'(g)] - [r(x) + r(p)] \\
&\leq 2 + [r'(x) + r'(g)] - [r(x) + r(x)]
\end{aligned}$$

Deuxième cas : zig-zig



$r'(x) = r(g)$

$r'(x) \geq r'(p)$

$r(p) \geq r(x)$

Le **coût amorti** d'une **rotation zig-zig** est

$$\begin{aligned}
\widehat{c_{\text{zig-zig}}} &= 2 + \Phi(T') - \Phi(T) \\
&= 2 + [r'(x) + r'(p) + r'(g)] - [r(x) + r(p) + r(g)] \\
&= 2 + [r'(p) + r'(g)] - [r(x) + r(p)] \\
&\leq 2 + [r'(x) + r'(g)] - [r(x) + r(x)] \\
&\leq 2 + r'(x) + r'(g) - 2r(x)
\end{aligned}$$

Intermède : la fonction logarithme est convexe

- Une propriété **connue** de la fonction log est sa **convexité** ;

Intermède : la fonction logarithme est convexe

- Une propriété **connue** de la fonction log est sa **convexité** ;
- Plus précisément, pour tous $a, b, c \in \mathbb{R}^+$, si $a + b \leq c$, alors $\log a + \log b \leq 2 \log c - 2$;

Intermède : la fonction logarithme est convexe

- Une propriété **connue** de la fonction log est sa **convexité** ;
- Plus précisément, pour tous $a, b, c \in \mathbb{R}^+$, si $a + b \leq c$, alors $\log a + \log b \leq 2 \log c - 2$;
- Pour **notre problème**, cela se traduit par la propriété suivante :

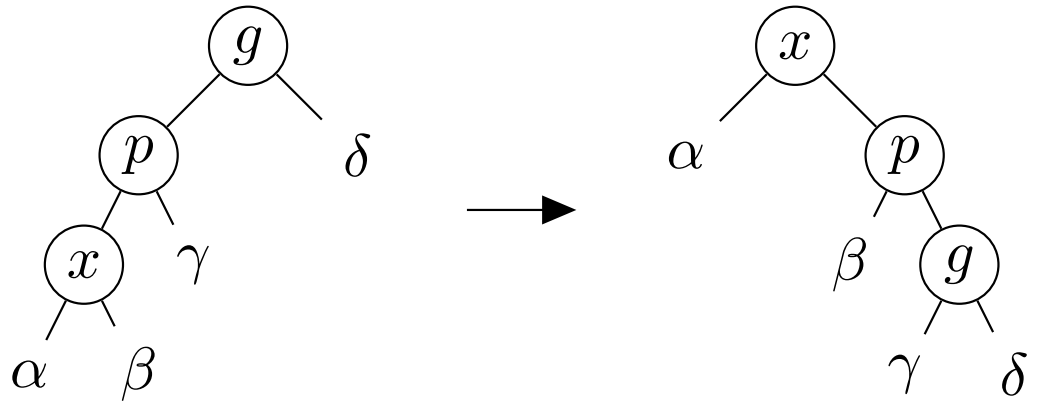
Intermède : la fonction logarithme est convexe

- Une propriété **connue** de la fonction log est sa **convexité** ;
- Plus précisément, pour tous $a, b, c \in \mathbb{R}^+$, si $a + b \leq c$, alors $\log a + \log b \leq 2 \log c - 2$;
- Pour **notre problème**, cela se traduit par la propriété suivante :
  - Si $w(x) + w(y) \leq w(z)$,

Intermède : la fonction logarithme est convexe

- Une propriété **connue** de la fonction log est sa **convexité** ;
- Plus précisément, pour tous $a, b, c \in \mathbb{R}^+$, si $a + b \leq c$, alors $\log a + \log b \leq 2 \log c - 2$;
- Pour **notre problème**, cela se traduit par la propriété suivante :
  - Si $w(x) + w(y) \leq w(z)$,
  - alors $r(x) + r(y) \leq 2r(z) - 2$,

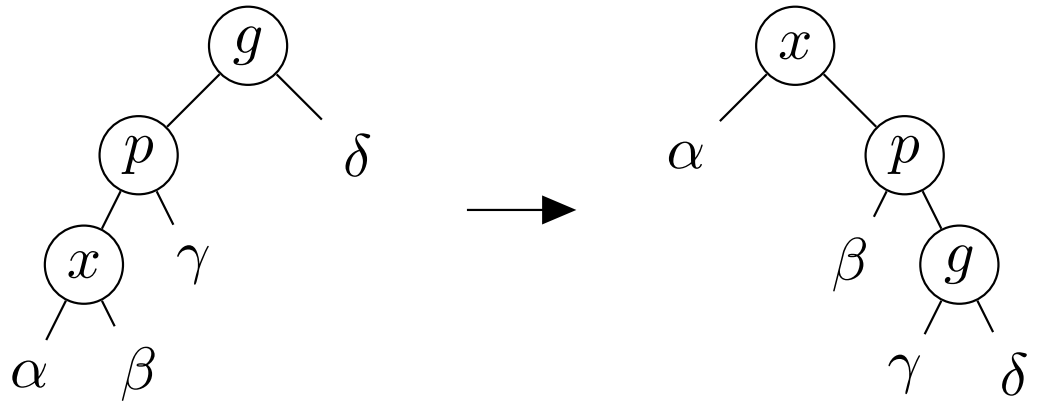
Deuxième cas : zig-zig (suite)



$w(x) + w'(g) \leq w'(x)$

$r(x) + r'(g) \leq 2r'(x) - 2$

Deuxième cas : zig-zig (suite)



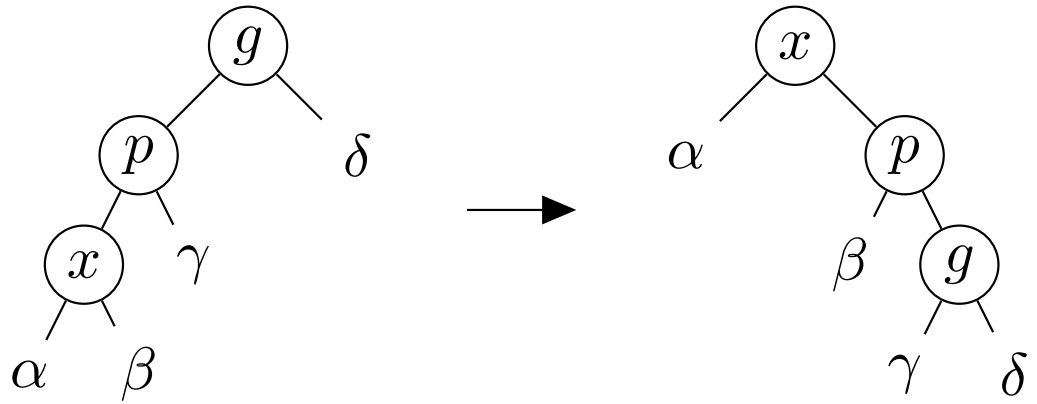
$w(x) + w'(g) \leq w'(x)$

$r(x) + r'(g) \leq 2r'(x) - 2$

Le **coût amorti** d'une **rotation zig-zig** est

$$\widehat{c_{\text{zig-zig}}} \leq 2 + r'(x) + r'(g) - 2r(x)$$

Deuxième cas : zig-zig (suite)



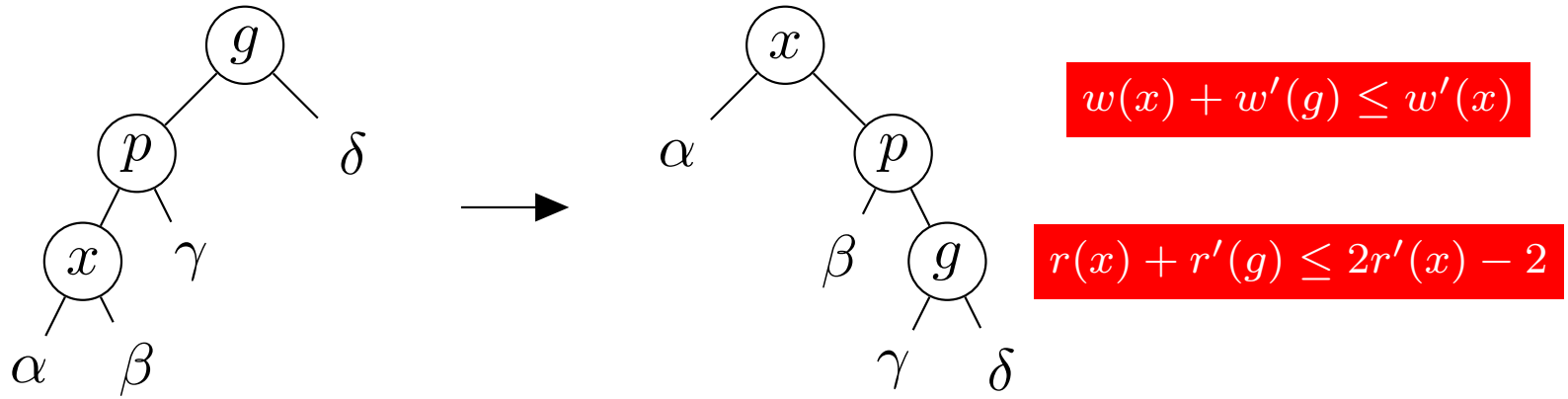
$w(x) + w'(g) \leq w'(x)$

$r(x) + r'(g) \leq 2r'(x) - 2$

Le **coût amorti** d'une **rotation zig-zig** est

$$\begin{aligned} \widehat{c_{\text{zig-zig}}} &\leq 2 + r'(x) + r'(g) - 2r(x) \\ &\leq 2 + [r(x) + r'(g)] + [r'(x) - 3r(x)] \end{aligned}$$

Deuxième cas : zig-zig (suite)



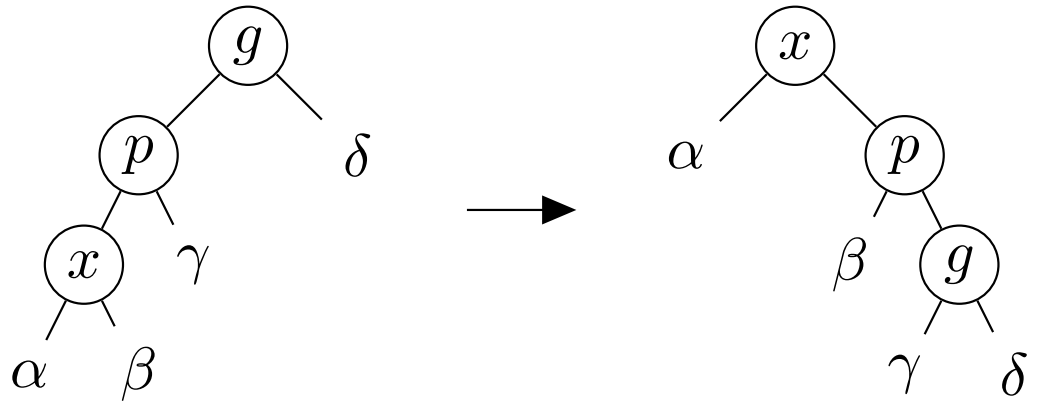
$w(x) + w'(g) \leq w'(x)$

$r(x) + r'(g) \leq 2r'(x) - 2$

Le **coût amorti** d'une **rotation zig-zig** est

$$\begin{aligned} \widehat{c_{\text{zig-zig}}} &\leq 2 + r'(x) + r'(g) - 2r(x) \\ &\leq 2 + [r(x) + r'(g)] + [r'(x) - 3r(x)] \\ &\leq 2 + [2r'(x) - 2] + [r'(x) - 3r(x)] \end{aligned}$$

Deuxième cas : zig-zig (suite)



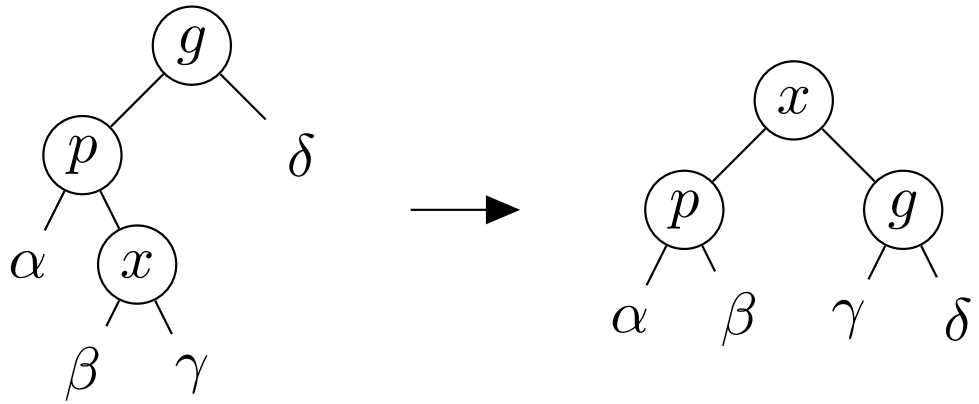
$w(x) + w'(g) \leq w'(x)$

$r(x) + r'(g) \leq 2r'(x) - 2$

Le **coût amorti** d'une **rotation zig-zig** est

$$\begin{aligned}
\widehat{c_{\text{zig-zig}}} &\leq 2 + r'(x) + r'(g) - 2r(x) \\
&\leq 2 + [r(x) + r'(g)] + [r'(x) - 3r(x)] \\
&\leq 2 + [2r'(x) - 2] + [r'(x) - 3r(x)] \\
&\leq 3(r'(x) - r(x)).
\end{aligned}$$

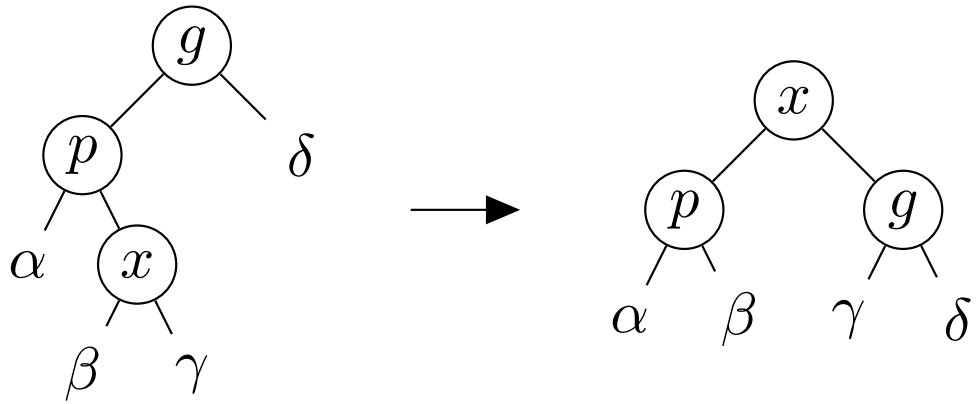
Troisième cas : zig-zag



$r'(x) = r(g)$

$r(p) \geq r(x)$

Troisième cas : zig-zag



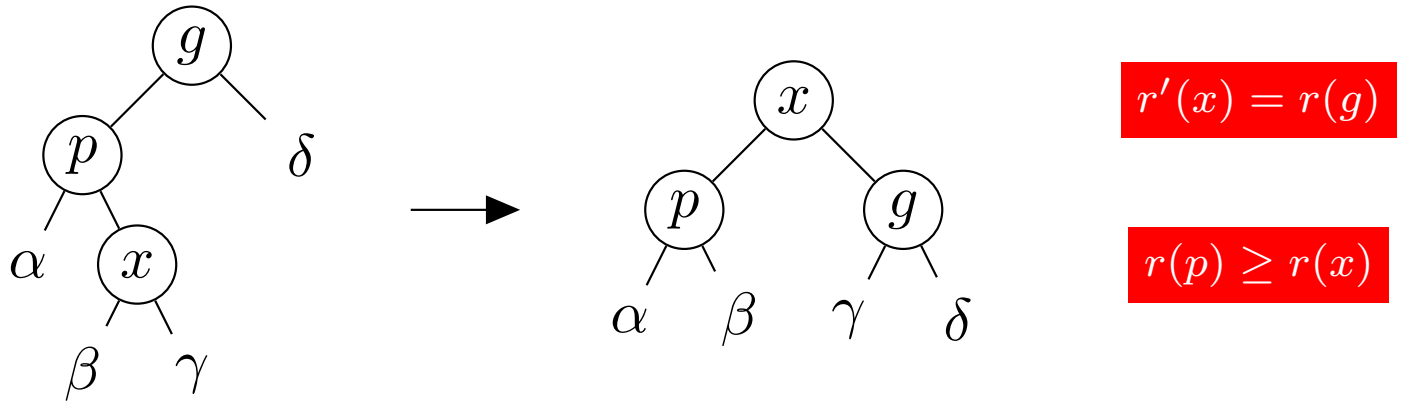
$r'(x) = r(g)$

$r(p) \geq r(x)$

Le **coût amorti** d'une **rotation zig-zag** est

$$\widehat{c_{\text{zig-zag}}} \leq 2 + \Phi(T') - \Phi(T)$$

Troisième cas : zig-zag



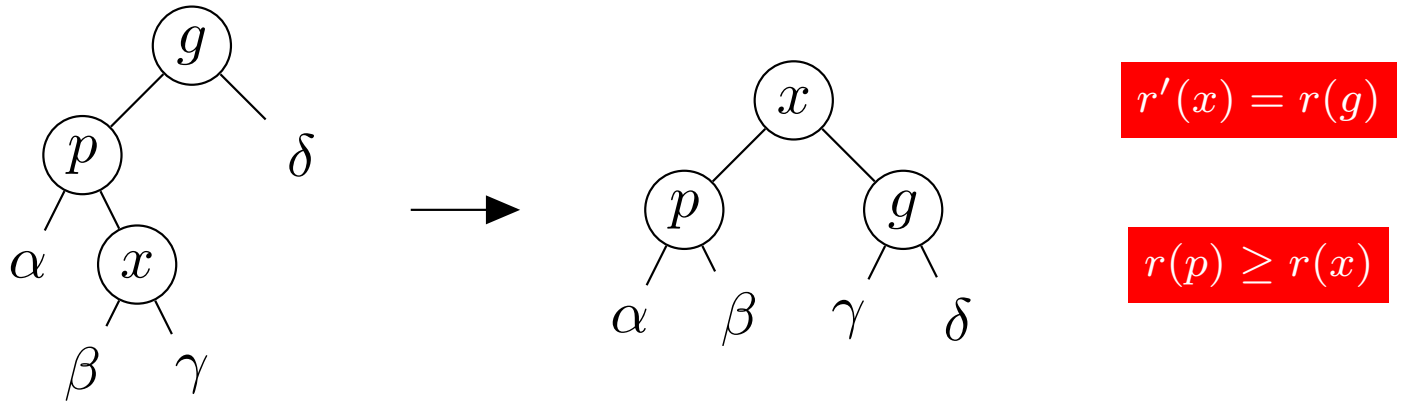
$r'(x) = r(g)$

$r(p) \geq r(x)$

Le **coût amorti** d'une **rotation zig-zag** est

$$\begin{aligned} \widehat{c_{\text{zig-zag}}} &\leq 2 + \Phi(T') - \Phi(T) \\ &= 2 + [r'(x) + r'(p) + r'(g)] - [r(x) + r(p) + r(g)] \end{aligned}$$

Troisième cas : zig-zag



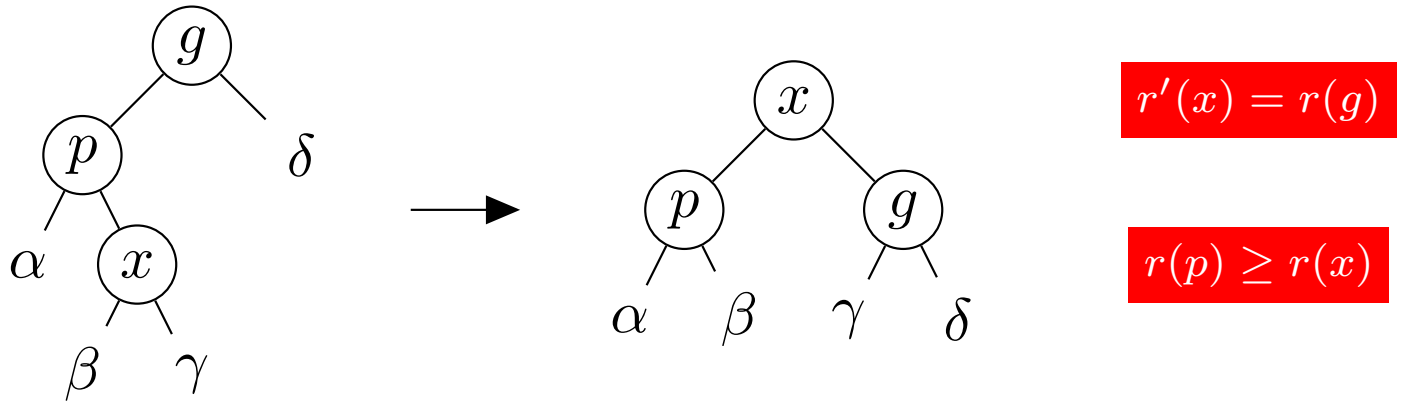
$r'(x) = r(g)$

$r(p) \geq r(x)$

Le **coût amorti** d'une **rotation zig-zag** est

$$\begin{aligned} \widehat{c_{\text{zig-zag}}} &\leq 2 + \Phi(T') - \Phi(T) \\ &= 2 + [r'(x) + r'(p) + r'(g)] - [r(x) + r(p) + r(g)] \\ &= 2 + [r'(p) + r'(g)] - [r(x) + r(p)] \end{aligned}$$

Troisième cas : zig-zag



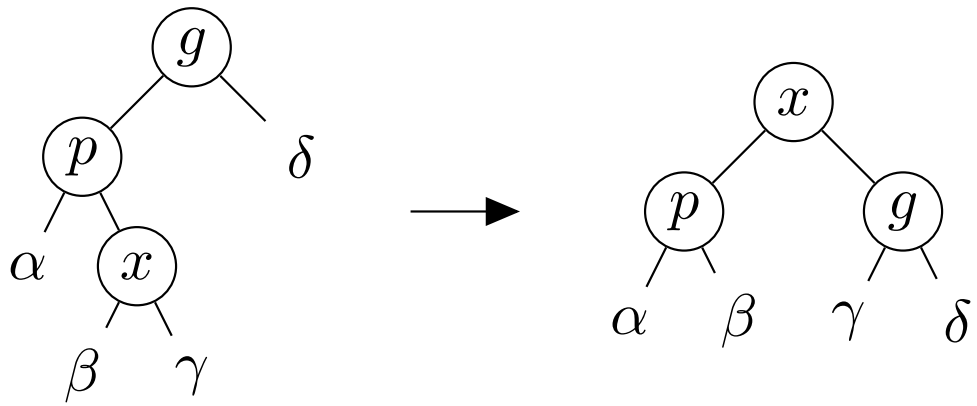
$r'(x) = r(g)$

$r(p) \geq r(x)$

Le **coût amorti** d'une **rotation zig-zag** est

$$\begin{aligned}
\widehat{c_{\text{zig-zag}}} &\leq 2 + \Phi(T') - \Phi(T) \\
&= 2 + [r'(x) + r'(p) + r'(g)] - [r(x) + r(p) + r(g)] \\
&= 2 + [r'(p) + r'(g)] - [r(x) + r(p)] \\
&\leq 2 + r'(p) + r'(g) - 2r(x)
\end{aligned}$$

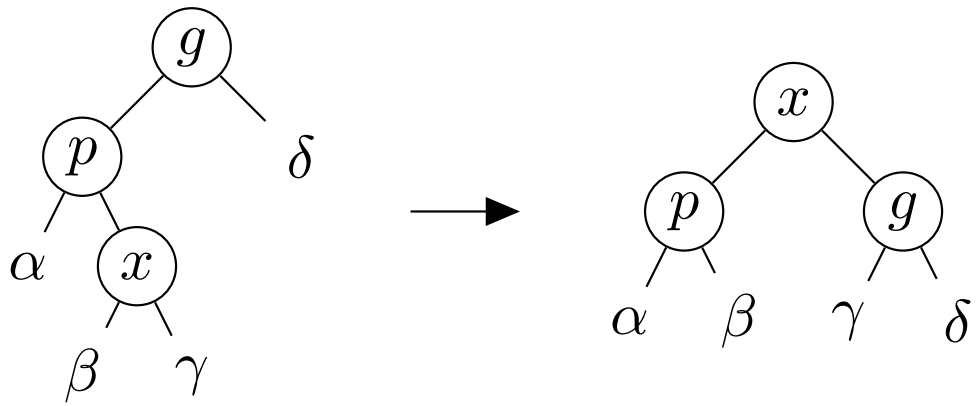
Troisième cas : zig-zag (suite)



$w'(p) + w'(g) \leq w'(x)$

$r'(p) + r'(g) \leq 2r'(x) - 2$

Troisième cas : zig-zag (suite)



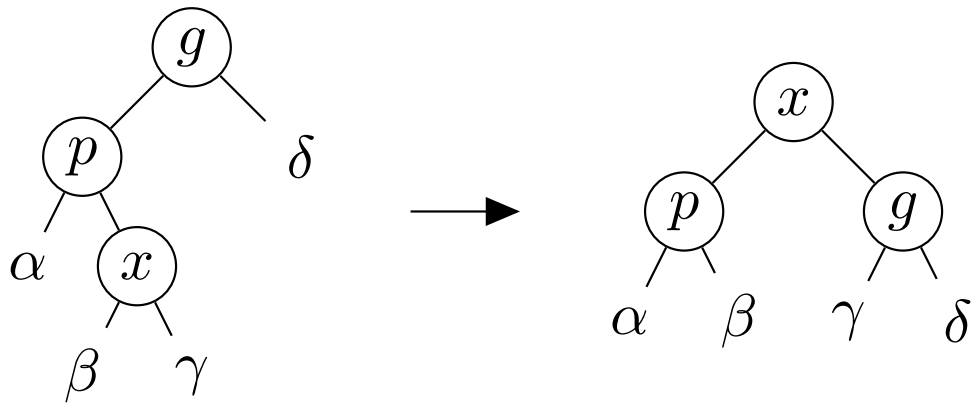
$w'(p) + w'(g) \leq w'(x)$

$r'(p) + r'(g) \leq 2r'(x) - 2$

Le **coût amorti** d'une **rotation zig-zag** est

$$\widehat{c_{\text{zig-zag}}} \leq 2 + r'(p) + r'(g) - 2r(x)$$

Troisième cas : zig-zag (suite)



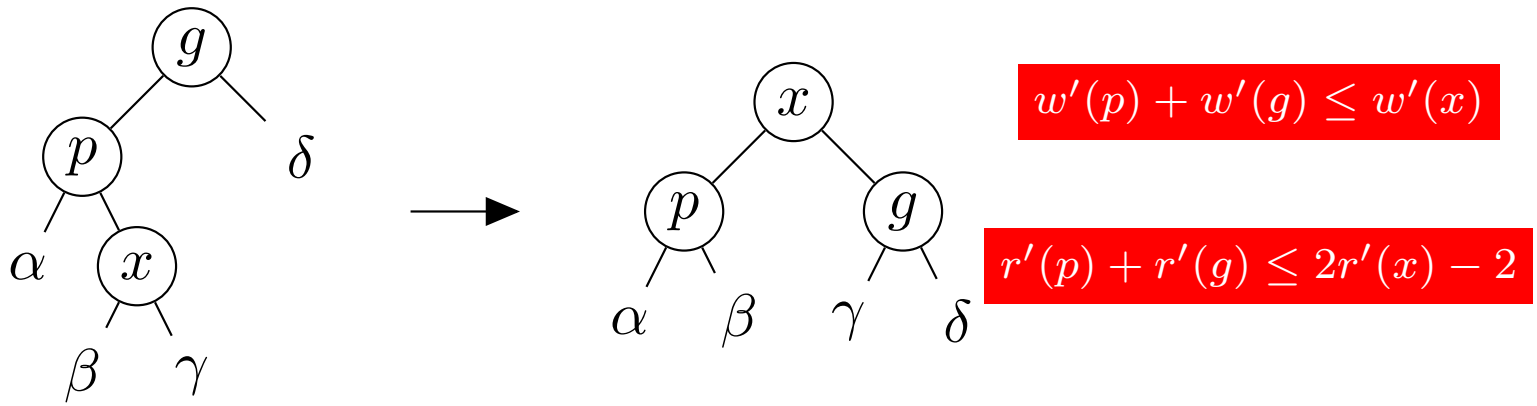
$w'(p) + w'(g) \leq w'(x)$

$r'(p) + r'(g) \leq 2r'(x) - 2$

Le **coût amorti** d'une **rotation zig-zag** est

$$\begin{aligned} \widehat{c_{\text{zig-zag}}} &\leq 2 + r'(p) + r'(g) - 2r(x) \\ &\leq 2 + 2r'(x) - 2 - 2r(x) \end{aligned}$$

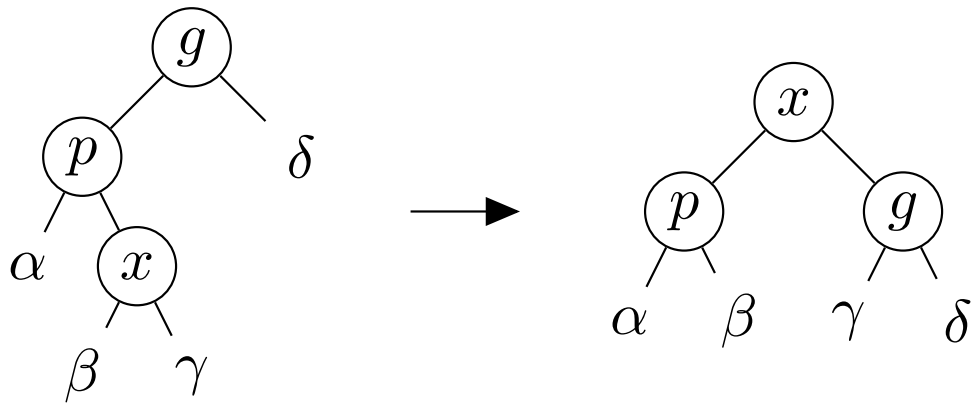
Troisième cas : zig-zag (suite)



Le **coût amorti** d'une **rotation zig-zag** est

$$\begin{aligned} \widehat{c_{\text{zig-zag}}} &\leq 2 + r'(p) + r'(g) - 2r(x) \\ &\leq 2 + 2r'(x) - 2 - 2r(x) \\ &\leq 2(r'(x) - r(x)) \end{aligned}$$

Troisième cas : zig-zag (suite)



$w'(p) + w'(g) \leq w'(x)$

$r'(p) + r'(g) \leq 2r'(x) - 2$

Le **coût amorti** d'une **rotation zig-zag** est

$$\begin{aligned} \widehat{c_{\text{zig-zag}}} &\leq 2 + r'(p) + r'(g) - 2r(x) \\ &\leq 2 + 2r'(x) - 2 - 2r(x) \\ &\leq 2(r'(x) - r(x)) \\ &\leq 3(r'(x) - r(x)) \end{aligned}$$

Coût amorti du « déploiement »

- En résumé,

$$\begin{aligned}
\widehat{c_{\text{zig}}} &\leq 3(r'(x) - r(x)) + 1 \\
\widehat{c_{\text{zig-zig}}} &\leq 3(r'(x) - r(x)) \\
\widehat{c_{\text{zig-zag}}} &\leq 3(r'(x) - r(x)).
\end{aligned}$$

Coût amorti du « déploiement »

- En résumé,

$$\begin{aligned} \widehat{c_{\text{zig}}} &\leq 3(r'(x) - r(x)) + 1 \\ \widehat{c_{\text{zig-zig}}} &\leq 3(r'(x) - r(x)) \\ \widehat{c_{\text{zig-zag}}} &\leq 3(r'(x) - r(x)). \end{aligned}$$

- Or, un déploiement est une **suite de rotations** ;

Coût amorti du « déploiement »

- En résumé,

$$\begin{aligned}
\widehat{c_{\text{zig}}} &\leq 3(r'(x) - r(x)) + 1 \\
\widehat{c_{\text{zig-zig}}} &\leq 3(r'(x) - r(x)) \\
\widehat{c_{\text{zig-zag}}} &\leq 3(r'(x) - r(x)).
\end{aligned}$$

- Or, un déploiement est une **suite de rotations** ;
- Soit $T_1, T_2, \ldots, T_n$ les arbres obtenus à chaque rotation et $r_i(x)$ le rang du noeud x dans l'arbre T_i ;

Coût amorti du « déploiement »

- En résumé,

$$\begin{aligned} \widehat{c_{\text{zig}}} &\leq 3(r'(x) - r(x)) + 1 \\ \widehat{c_{\text{zig-zig}}} &\leq 3(r'(x) - r(x)) \\ \widehat{c_{\text{zig-zag}}} &\leq 3(r'(x) - r(x)). \end{aligned}$$

- Or, un déploiement est une **suite de rotations** ;
- Soit $T_1, T_2, \ldots, T_n$ les arbres obtenus à chaque rotation et $r_i(x)$ le rang du noeud x dans l'arbre T_i ;
- Alors le **coût amorti** du déploiement est

$$\widehat{c} \leq 1 + \sum_{i=1}^{n-1} 3(r_{i+1}(x) - r_i(x)).$$

Coût amorti du « déploiement »(suite)

- Le **coût amorti** du déploiement est

$$\begin{aligned}
\widehat{c} &\leq 1 + \sum_{i=1}^{n-1} 3(r_{i+1}(x) - r_i(x)) \\
&\leq 1 + 3\sum_{i=1}^{n-1} (r_{i+1}(x) - r_i(x)) \\
&\leq 1 + r_n(x) - r_1(x) \\
&\leq 1 + \log(n) - 0,
\end{aligned}$$

Coût amorti du « déploiement »(suite)

- Le **coût amorti** du déploiement est

$$\begin{aligned}
\widehat{c} &\leq 1 + \sum_{i=1}^{n-1} 3(r_{i+1}(x) - r_i(x)) \\
&\leq 1 + 3\sum_{i=1}^{n-1} (r_{i+1}(x) - r_i(x)) \\
&\leq 1 + r_n(x) - r_1(x) \\
&\leq 1 + \log(n) - 0,
\end{aligned}$$

- Ainsi, $\widehat{c}$ est $\mathcal{O}(\log n)$.

Coût amorti du « déploiement »(suite)

- Le **coût amorti** du déploiement est

$$\begin{aligned}
\widehat{c} &\leq 1 + \sum_{i=1}^{n-1} 3(r_{i+1}(x) - r_i(x)) \\
&\leq 1 + 3\sum_{i=1}^{n-1} (r_{i+1}(x) - r_i(x)) \\
&\leq 1 + r_n(x) - r_1(x) \\
&\leq 1 + \log(n) - 0,
\end{aligned}$$

- Ainsi, $\widehat{c}$ est $\mathcal{O}(\log n)$.
- En tenant compte des **insertions** et **suppressions**, on peut démontrer que le coût amorti est aussi $\mathcal{O}(\log n)$.

Résumé

Il faut retenir que

- Une opération précise peut avoir un coût $\mathcal{O}(n)$;

Résumé

Il faut retenir que

- Une opération précise peut avoir un coût $\mathcal{O}(n)$;
- Par contre, si on effectue **n'importe quelle** suite de m opérations, alors le coût est $\mathcal{O}(m \log n)$.

Résumé

Il faut retenir que

- Une opération précise peut avoir un coût $\mathcal{O}(n)$;
- Par contre, si on effectue **n'importe quelle** suite de m opérations, alors le coût est $\mathcal{O}(m \log n)$.
- Intuitivement, ceci signifie que, même si une opération est **coûteuse**, les rotations qui sont effectuées permettent d'**accélérer** les opérations futures.