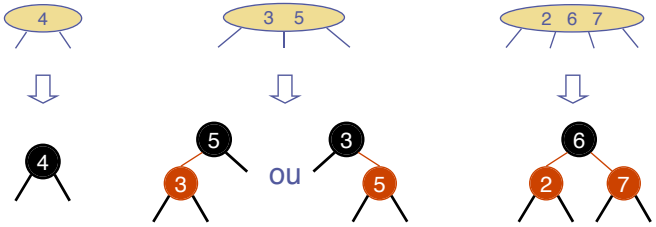
Arbres Rouges-Noirs

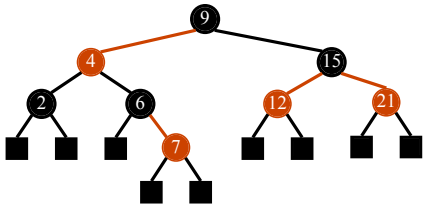
Correspondance entre les arbres (2,4) et les arbres rouges-noirs

- On peut voir un arbre rouge-noir comme étant la représentation d'un arbre (2,4) par un arbre binaire de recherche dont les noeuds sont colorés rouge ou **noir**
- En comparaison avec les arbres (2,4), les arbres rouges-noirs ont
  - La même complexité logarithmique
  - Plus simple à implémenter car on a une seule sorte de noeud



Arbres rouges-noirs

- Un arbre rouge-noir peut aussi être défini comme étant un arbre binaire de recherche qui satisfait les propriétés suivantes:
  - Propriété de racine: La racine est **noir**
  - Propriété externe: Les noeuds externes sont **noirs**
  - Propriété interne: Les enfants d'un noeud **rouge** sont **noirs**
  - Propriété de profondeur: Tous les noeuds externes ont la même profondeur **noir**, qui est définie comme étant le nombre d'ancêtre interne **noir**

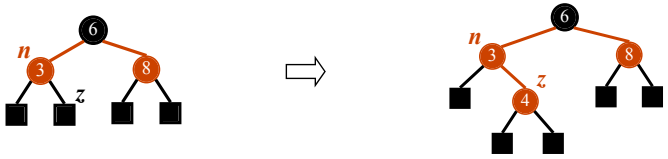


Hauteur d'un arbre rouge-noir

- La hauteur d'un arbre rouge-noir gardant en mémoire n éléments est en $O(\log n)$
  - Preuve: La hauteur d'un arbre rouge-noir est au plus le double de la hauteur de l'arbre (2,4) lui correspondant, qui est de $O(\log n)$
- L'algorithme de recherche dans un arbre rouge-noir est exactement le même algorithme que pour la recherche dans un arbre binaire de recherche
- On a donc que la complexité en temps de la recherche dans un arbre rouge-noir est en $O(\log n)$

Insertion dans un arbre rouge-noir

- Pour insérer un élément (k,v) dans un arbre rouge-noir, on exécute l'algorithme d'insertion d'un arbre binaire de recherche et on colore **rouge** le nouveau noeud z, sauf si z est la racine
- Exemple: Insérer un élément de clé 4
  - Les propriétés de racine et de profondeur, de même que la propriété externe sont préservées
  - Si le parent n de z est **noir**, on préserve aussi la propriété interne et on a terminé l'algorithme d'insertion
  - Sinon, on a un **double rouge** et on doit modifier l'arbre pour rétablir la propriété interne.



Remédier à un double rouge

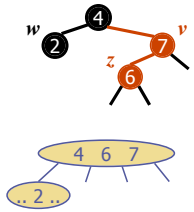
- Considérons un **double rouge**: v, le parent **rouge**, z, le fils **rouge** et considérons w, le frère de v.

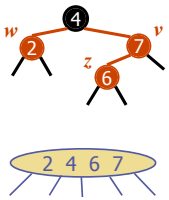
Cas 1: w est **noir**

- on peut voir le double rouge comme un remplacement incorrect d'un 4-noeud
- Restructuration: On exécute le bon remplacement du 4-noeud

Cas 2: w est **rouge**

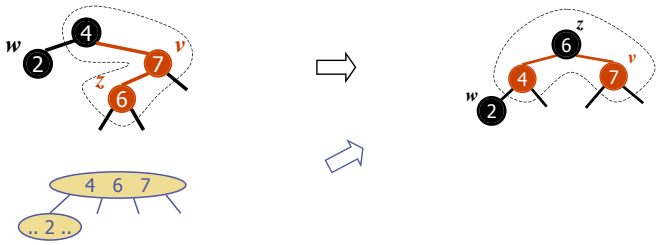
- on peut voir le double rouge comme un débordement dans un arbre (2,4)
- Recoloration: On performe l'équivalent d'un fractionnement





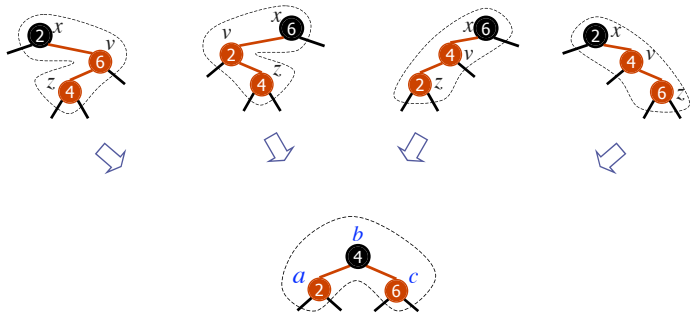
Cas 1: Restructuration

- Considérons un **double rouge**: v, le parent **rouge**, z, le fils **rouge** et considérer w, le frère de v. Lorsque w est **noir**, on exécute une restructuration
- Cela revient à exécuter le remplacement correct du 4-noeud
- La propriété interne est restorée et les autres propriétés sont préservées



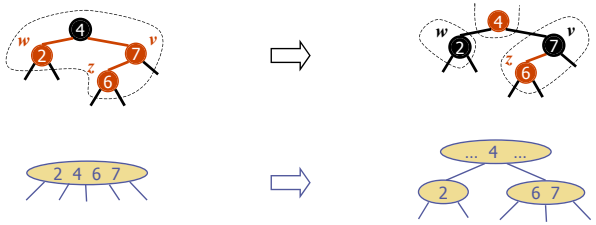
Cas 1: Restructuration (suite)

- Il y a 4 configurations possibles demandant une restructuration, dépendant de l'emplacement des deux noeuds rouges formant le **double rouge**



Cas 2: Recoloration

- Considérons un **double rouge**: v, le parent **rouge**, z, le fils **rouge** et considéroms w, le frère de v. Lorsque w est **rouge**, on exécute une recoloration
- On recolore le parent v et son frère w en **noir** et le grand-parent (le parent de v) devient **rouge**, sauf si c'est la racine
- Cela correspond à exécuter le fractionnement d'un 5-noeud
- Il est possible que le **double rouge** se propage chez le grand-parent



Complexité en temps d'une insertion

Algorithme *insérer*(k, v)

1. On exécute chercher(k) pour trouver le noeud d'insertion z
2. On insère le nouvel élément (k, v) dans le noeud z et on colore z rouge
3. **Tant que** *doubleRouge*(z)
   si *estNoir*(*frère*(*parent*(z)))
   $z \leftarrow$ *restructure*(z)
   sinon
   $z \leftarrow$ *recolore*(z)

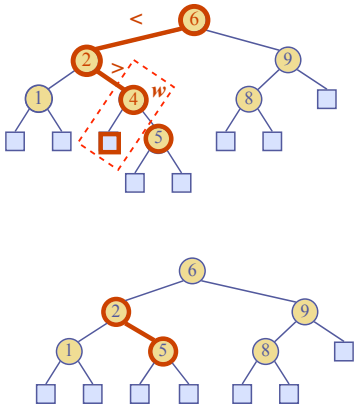
- La hauteur d'un arbre rouge-noir gardant en mémoire n éléments est en O(log n)
- L'étape 1 prend un temps O(log n), étant donné qu'on doit visiter O(log n) noeuds lors de la recherche
- L'étape 2 prend un temps $O(1)$
- L'étape 3 prend un temps O(log n), étant donné qu'on exécute au plus
  - O(log n) recolaration, chacune prenant un temps $O(1)$
  - Au plus une restructuration prenant un temps $O(1)$
- L'insertion d'un élément dans un arbre rouge-noir prend donc un temps O(log n)

Suppression dans un arbre rouge-noir

- Pour supprimer un élément de clé k dans un arbre rouge-noir, on exécute l'algorithme de suppression d'un arbre binaire de recherche

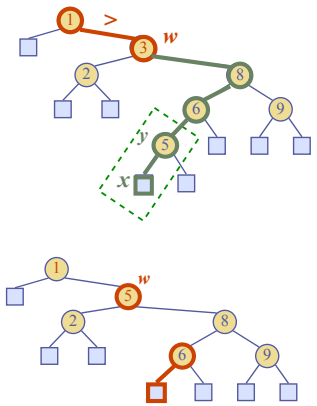
Supprimer dans un arbre binaire de recherche (rappel)

- Pour enlever un élément de clé k dans un arbre binaire de recherche, on commence par exécuter l'algorithme chercher(k).
- Exemple 1: Enlever(4)
- Si k est dans l'arbre l'algorithme chercher(k) se terminera dans un noeud interne w
- Si l'un des enfant de w est une feuille, on enlève cette feuille et w
- Sinon...



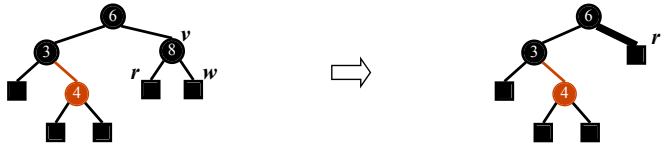
Supprimer dans un arbre binaire de recherche (suite)

- Si k est dans l'arbre, l'algorithme chercher(k) se terminera dans un noeud interne w. Si les fils de w sont tous les deux des noeuds internes alors
- Exemple 2: Enlever(3)
  - On trouve le noeud interne y qui suit w lors d'un parcours symétrique de l'arbre et son fils gauche x
  - On enlève l'entrée dans w et on la remplace par l'entrée dans y
  - On enlève les noeuds y et x



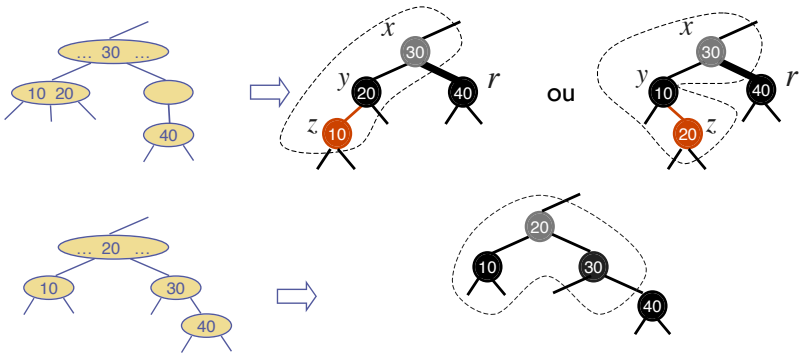
Suppression dans un arbre rouge-noir

- Pour supprimer un élément de clé k d'un un arbre rouge-noir, on exécute l'algorithme de suppression d'un arbre binaire de recherche
- Soit v, le noeud interne et w, le noeud externe enlever lors de la suppression. Soit r, le frère de w.
  - Si soit v ou r était **rouge**, on colore r **noir** et on a terminé
  - Sinon (v et r était **noir**), enlever v va causer une violation de la propriété de profondeur et demander une restructuration de l'arbre. On appelera cette situation **double noir** au noeud r.



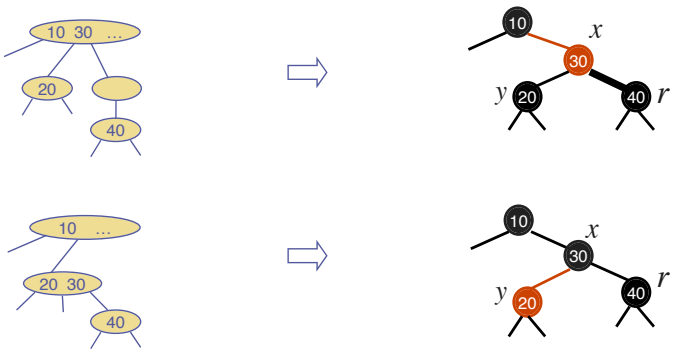
Remédier à un double noir

- Soit r le noeud **double noir**. Soit y, le frère de r.
  - Cas 1: Si y est **noir** et a un fils z **rouge**, on performe une restructuration qui équivaut à un transfert dans l'arbre (2,4) correspondant. À la suite de cette restructuration, toutes les propriétés des arbres rouges-noirs sont rétablies.



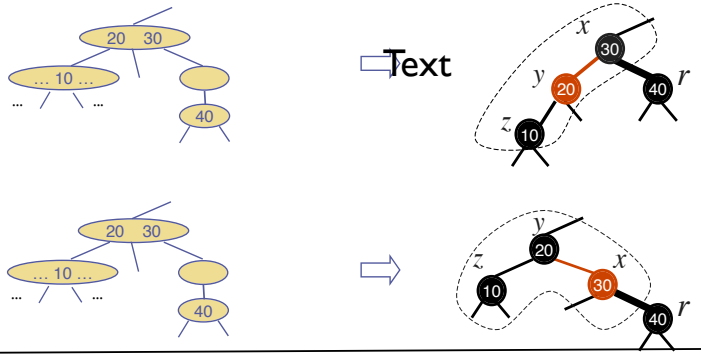
Remédier à un double noir

- Soit r le noeud **double noir**. Soit y, le frère de r.
  - Cas 2: Si y est **noir** et les deux fils de y sont **noirs**, on performe une recoloration qui équivaut à une fusion dans l'arbre (2,4) correspondant. La recolaration peut causer un problème de **double noir** chez le parent de r.



Remédier à un double noir

- Soit r le noeud **double noir**. Soit y, le frère de r.
  - Cas 3: y est **rouge**, on performe un ajustement qui équivaut à choisir une représentation différente d'un 3-noeud dans l'arbre (2,4) correspondant. L'ajustement va nous ramener soit dans le cas 1, soit dans le cas 2



Insertion: Remédier à un double rouge		
Opérations arbres rouge-noir	Opérations arbres (2,4)	Résultats
Restructuration	Changement de représentation d'un 4-noeud	Le double rouge est enlevé
Recoloration	Fractionnement	Le double rouge est enlevé ou il se propage vers le haut

Suppression: Remédier à un double noir		
Opérations arbres rouge-noir	Opérations arbres (2,4)	Résultats
Restructuration	Transfert	Le double noir est enlevé
Recoloration	Fusion	Le double noir est enlevé ou il se propage vers le haut
Ajustement	Changement de représentation d'un 3-noeud	Suivi d'une restructuration ou d'une recoloration