Algorithmique 1
Les séquences triées

Olivier Markowitch

Séquences triées

Nous allons nous intéresser à l'usage et la manipulation de séquences de données triées.

Le type de donnée « séquence triée » sera notre type de données abstrait

Interface des éléments formant une séquence triée

L'interface du type de donnée « séquence triée » est composée des primitives suivantes :

- getNext : donne accès à l'élément de la séquence triée qui suit l'élément courant
- getPrevious : donne accès à l'élément de la séquence triée qui précède l'élément courant
- getData : indique la valeur contenue dans l'élément courant dans la séquence triée

Interface d'une séquence triée

Interface du type de donnée « séquence triée » (suite) :

- getFirst : donne accès au premier élément de la séquence triée
- getLast : donne accès au dernier élément de la séquence triée
- `insert` : insère un élément d'une valeur donnée dans la séquence triée
- `remove` : supprime l'élément courant de la séquence triée
- `find` : donne accès à un élément de la séquence triée, s'il existe, d'une valeur donnée

Séquences triées implémentées au moyen d'un tableau

Considérons une séquence triée d'éléments stockés de manière contigüe en mémoire (dans un tableau par exemple)

Une telle séquence permet de rechercher efficacement un élément, grâce à une recherche dichotomique

Par contre, la gestion des insertions et suppressions d'éléments est lourde car elle entraîne des décalages d'éléments respectivement vers la droite et la gauche

Séquences triées implémentées au moyen de pointeurs

Supposons à présent une liste chaînée triée

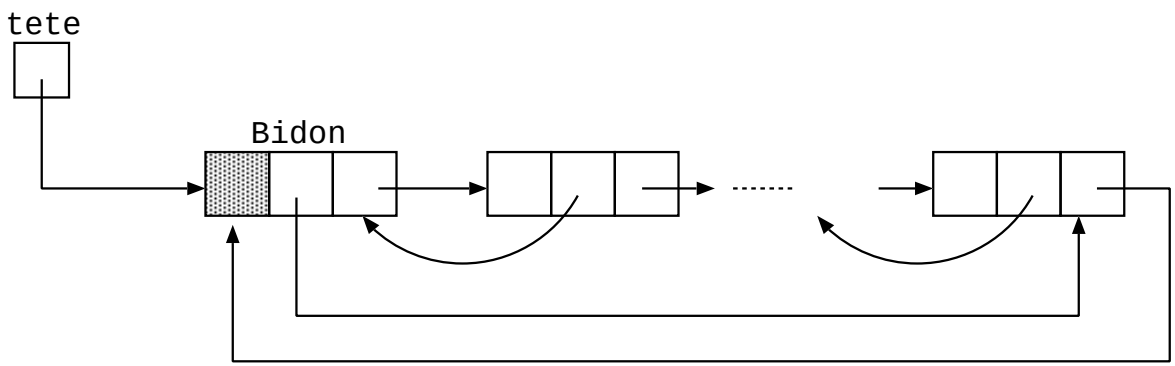
Cette fois-ci ce seront les opérations d'insertion et de suppression d'éléments qui seront aisées à réaliser

Par contre la recherche d'un élément dans la séquence sera d'une complexité en temps linéaire

De plus, les opérations d'insertion et de suppression pourront éventuellement aussi devoir être précédées d'une recherche dans la liste avant de pouvoir être réalisées

Séquences triées implémentées au moyen de pointeurs

La liste triée peut être bidirectionnelle, circulaire et avec un élément bidon en tête de liste



Liste circulaire bidirectionnelle : nœud

```
class NodeBidir:
    def __init__(self,initdata):
        self.data = initdata
        self.next = None
        self.previous = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def getPrevious(self):
        return self.previous
```

Liste circulaire bidirectionnelle : nœud

```
    def setData(self,newdata):
        self.data = newdata

    def setNext(self,newnext):
        self.next = newnext

    def setPrevious(self,newprevious):
        self.previous = newprevious
```

Liste circulaire bidirectionnelle

```
class ListeCircBidir:
    def __init__(self):
        self.head = NodeBidir(-1)
        self.head.setNext(self.head)
        self.head.setPrevious(self.head)
        self.count = 0

    def isEmpty(self):
        return self.head.getNext() == self.head

    def add(self,item):
        temp = NodeBidir(item)
        temp.setNext(self.head.getNext())
        temp.setPrevious(self.head)
        self.head.getNext().setPrevious(temp)
        self.head.setNext(temp)
        self.count = self.count + 1
```

Liste circulaire bidirectionnelle

```
    def addAfter(self,base,item):
        temp = NodeBidir(item)
        temp.setNext(base.getNext())
        temp.setPrevious(base)
        base.getNext().setPrevious(temp)
        base.setNext(temp)
        self.count = self.count + 1

    def length(self):
        return self.count

    def tete(self):
        return self.head.getNext()

    def fin(self):
        return self.head.getPrevious()
```

Liste circulaire bidirectionnelle

```
    def search(self,item):
        current = self.head.getNext()
        while current != self.head and current.getData() ↩
            != item:
            current = current.getNext()
        if current == self.head:
            return None
        else:
            return current

    def remove(self,base):
        base.getPrevious().setNext(base.getNext())
        base.getNext().setPrevious(base.getPrevious())
        self.count = self.count - 1
```

Séquence triée par une liste chaînée

```
class SeqTriee:
    def __init__(self):
        self.liste=ListeCircBidir()

    def getFirst(self):
        return self.liste.tete()

    def getLast(self):
        return self.liste.fin()

    def length(self):
        return self.liste.length()

    def isEmpty(self):
        return self.liste.isEmpty()
```

Séquence triée par une liste chaînée

```
    def insert(self,item):
        if self.liste.isEmpty():
            self.liste.add(item)
        else:
            current = self.liste.tete()
            fin = current == self.liste.fin()
            while current.getData() < item and not fin:
                current = current.getNext()
                fin = current == self.liste.fin()
            if current.getData() < item:
                self.liste.addAfter(current,item)
            else:
                self.liste.addAfter(current.getPrevious(),↩
                    item)

    def remove(self,base):
        self.liste.remove(base)

    def find(self,item):
        return self.liste.search(item)
```

Séquences triées sur mesure

L'idéal serait d'avoir une implémentation de la structure de données abstraite d'une séquence triée qui combine à la fois

- l'efficacité de la recherche dichotomique dans un vecteur trié
- la souplesse et la simplicité de l'insertion et de la suppression d'un élément dans une liste triée

Séquences triées sur mesure

Une telle implémentation est réalisable en utilisant un arbre binaire particulier dont les nœuds sont les éléments de la séquence triée et respectant la propriété suivante :

Propriété

soit x l'information présente dans le nœud courant, on impose que le sous-arbre gauche du nœud courant ne contienne que des éléments dont la valeur est inférieure ou égale à x et que le sous-arbre droit ne contienne que des éléments dont la valeur est supérieure ou égale à x

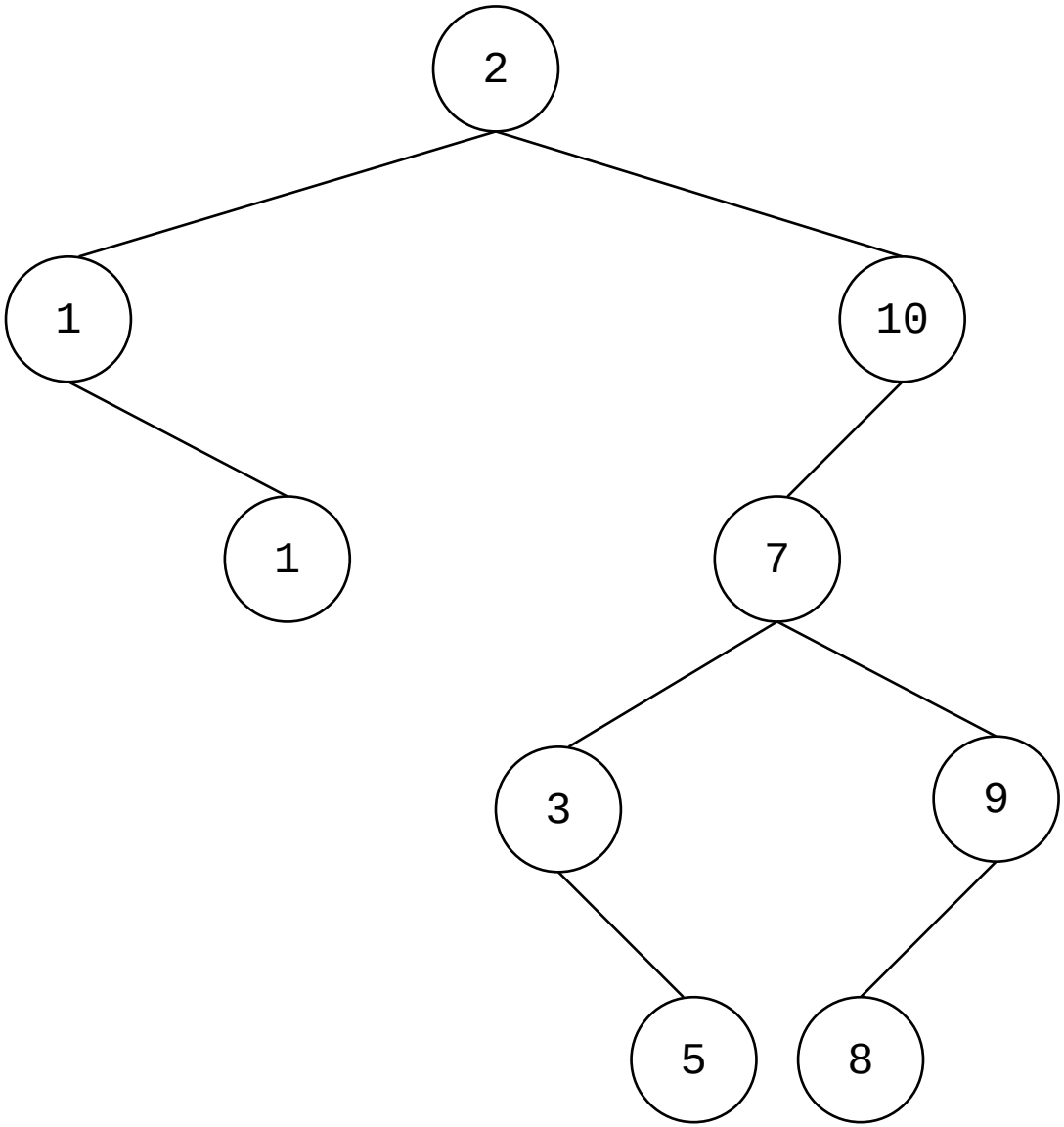
Séquences triées sur mesure

Soit x l'information présente dans le nœud courant, le sous-arbre gauche du nœud courant ne contient que des éléments dont la valeur est inférieure ou égale à x et le sous-arbre droit ne contient que des éléments dont la valeur est supérieure ou égale à x

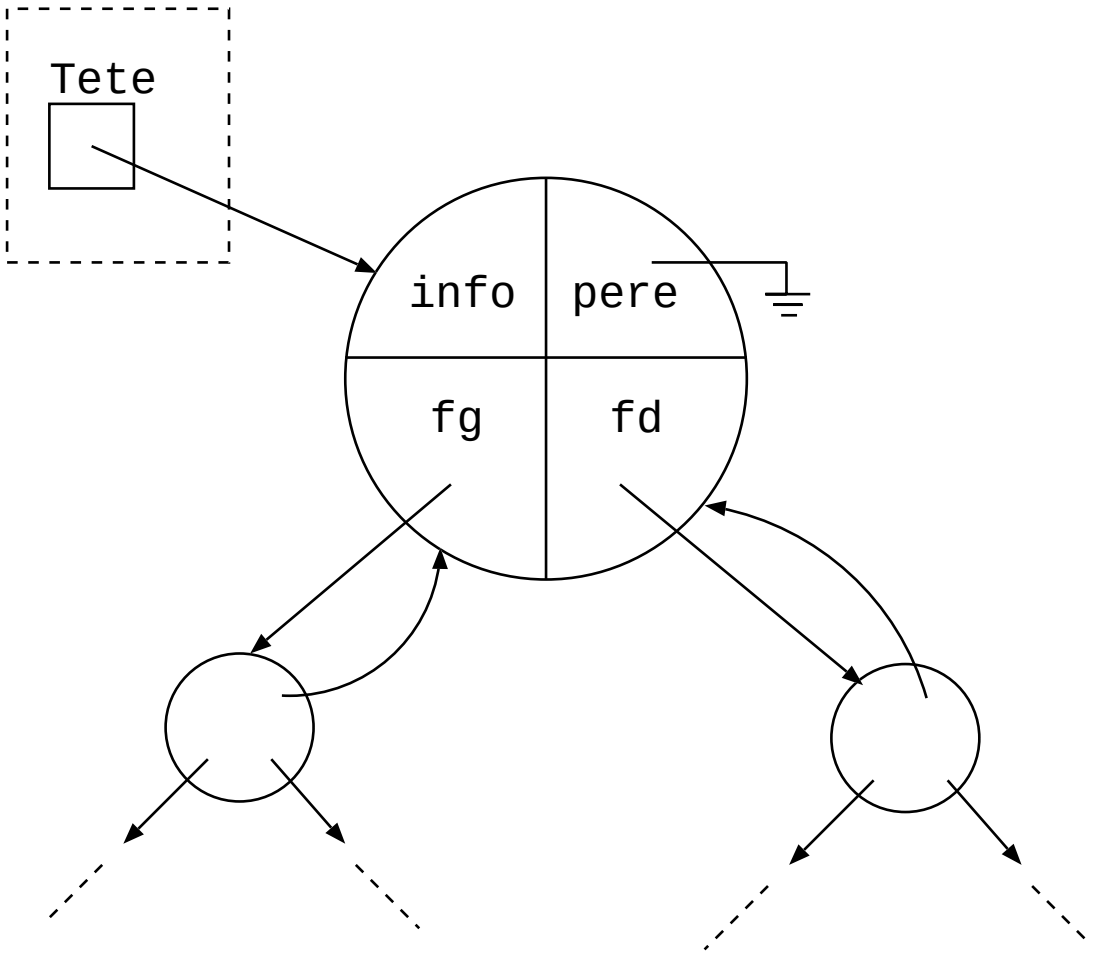
Cette règle doit être vérifiée pour tous les nœuds

Un tel arbre binaire est appelé arbre binaire de recherche (ABR) ou arbre binaire de tri

Arbre binaire de recherche



Arbre binaire de recherche



ABR : recherche d'un élément

La recherche d'un élément dans un arbre binaire de recherche ressemble à la recherche dichotomique dans un vecteur trié

Si on cherche la valeur x, on compare x à la valeur y présente dans la racine

Si x est strictement inférieure à y, on se dirige vers le sous-arbre gauche

Si x est strictement supérieure à y on se dirige vers le sous-arbre droit

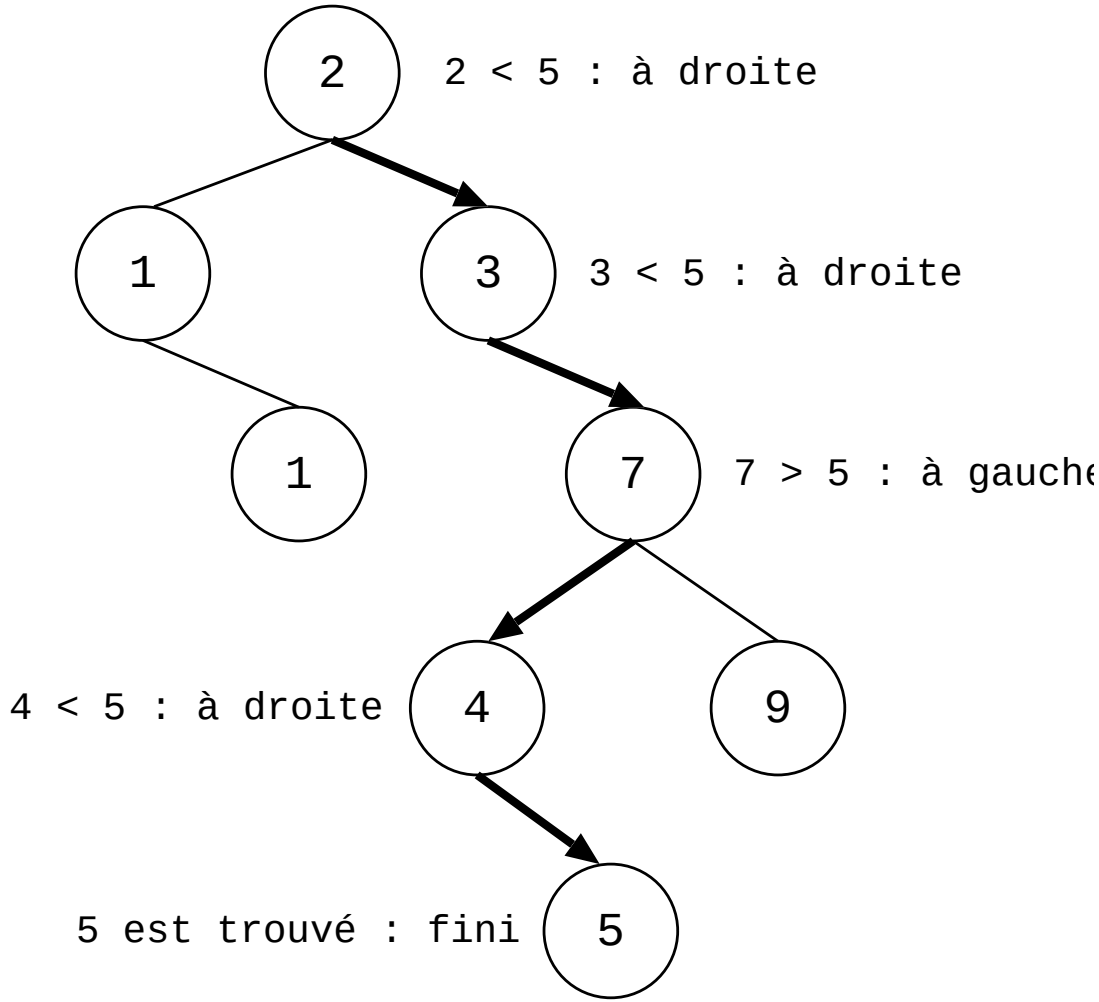
Si x est égale à y on s'arrête

ABR : recherche d'un élément

Ce processus est appliqué récursivement à la racine du sous-arbre considéré à chaque étape

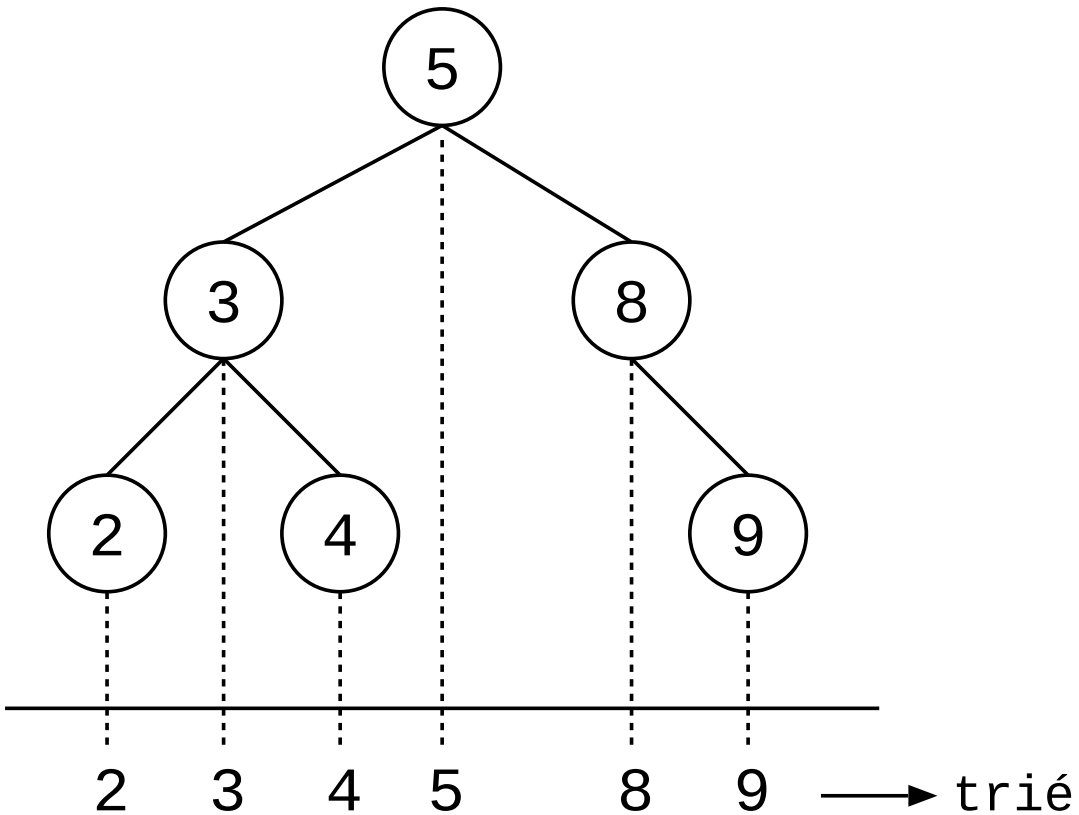
La recherche s'arrête soit parce qu'un nœud ayant une information égale à x a été trouvé, soit parce qu'une feuille a été atteinte sans trouver x

ABR : recherche d'un élément



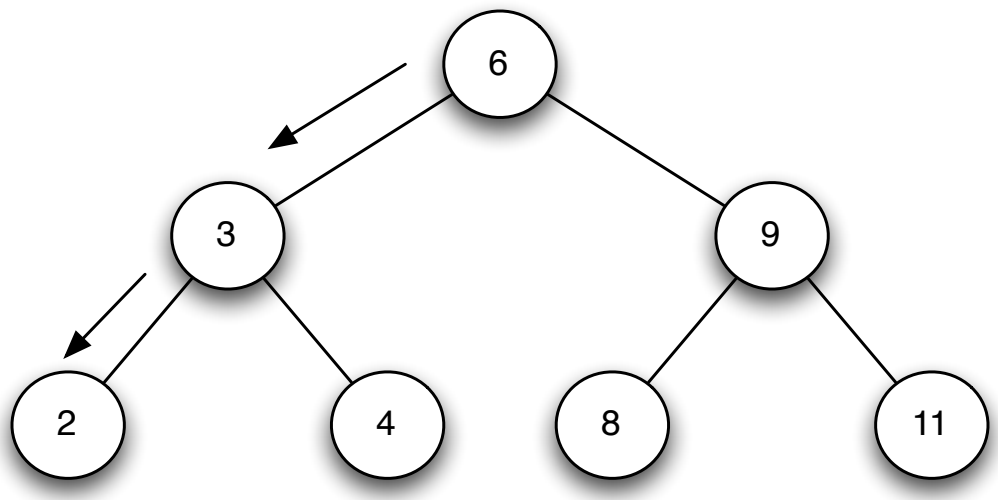
ABR : affichage des éléments

Pour afficher les éléments de l'arbre de manière triée, il suffit de faire un parcours infixe de l'arbre binaire de recherche



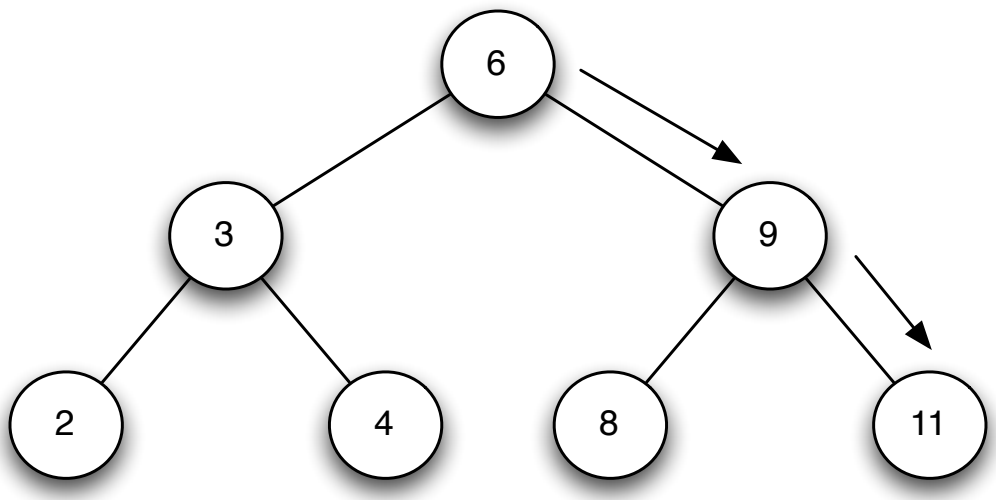
ABR : premier élément

Pour accéder au premier élément d'un arbre binaire de recherche (le plus petit élément) on part de la racine et on descend le plus à gauche possible (en parcourant les fils gauches)



ABR : dernier élément

Pour accéder au dernier élément d'un arbre binaire de recherche (le plus grand élément) on part de la racine et on descend le plus à droite possible (en parcourant les fils droits)

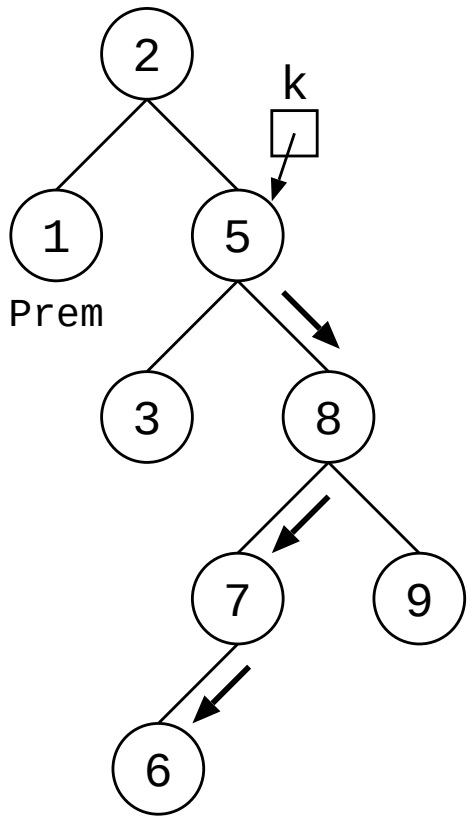


ABR : élément suivant

Pour accéder à l'élément suivant d'un élément courant dans un arbre binaire de recherche, on considère les deux situations suivantes

ABR : élément suivant

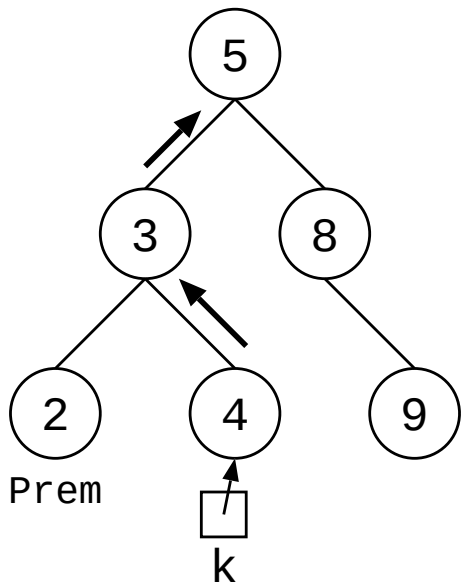
1. Si l'élément courant a un fils droit, alors en partant de ce fils droit, on descend le plus à gauche possible (en parcourant les fils gauches)



Suivant de 5

ABR : élément suivant

2. Si l'élément courant n'a pas de fils droit, alors on remonte dans l'arbre jusqu'au moment où on remonte en suivant une arête « fils gauche »



Suivant de 4

ABR : élément précédent

La recherche d'un élément précédant un élément courant suit la même logique mais en parcourant les fils droit de l'éventuel fils gauche de l'élément de départ ou, si cet élément n'a pas de fils gauche, en remontant dans l'arbre jusqu'au moment où on remonte en suivant une arête « fils droit »

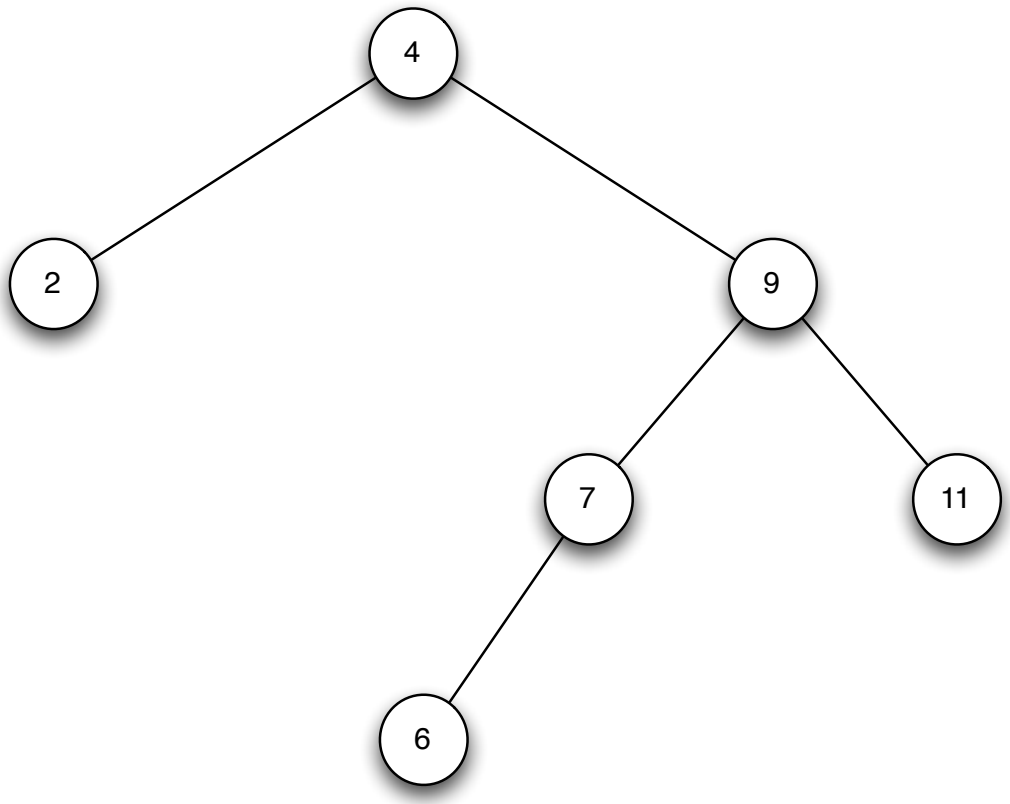
ABR : ajout d'un élément

L'ajout d'un nouvel élément dans un arbre binaire de recherche nécessite de parcourir l'arbre pour trouver l'endroit adéquat où réaliser l'insertion

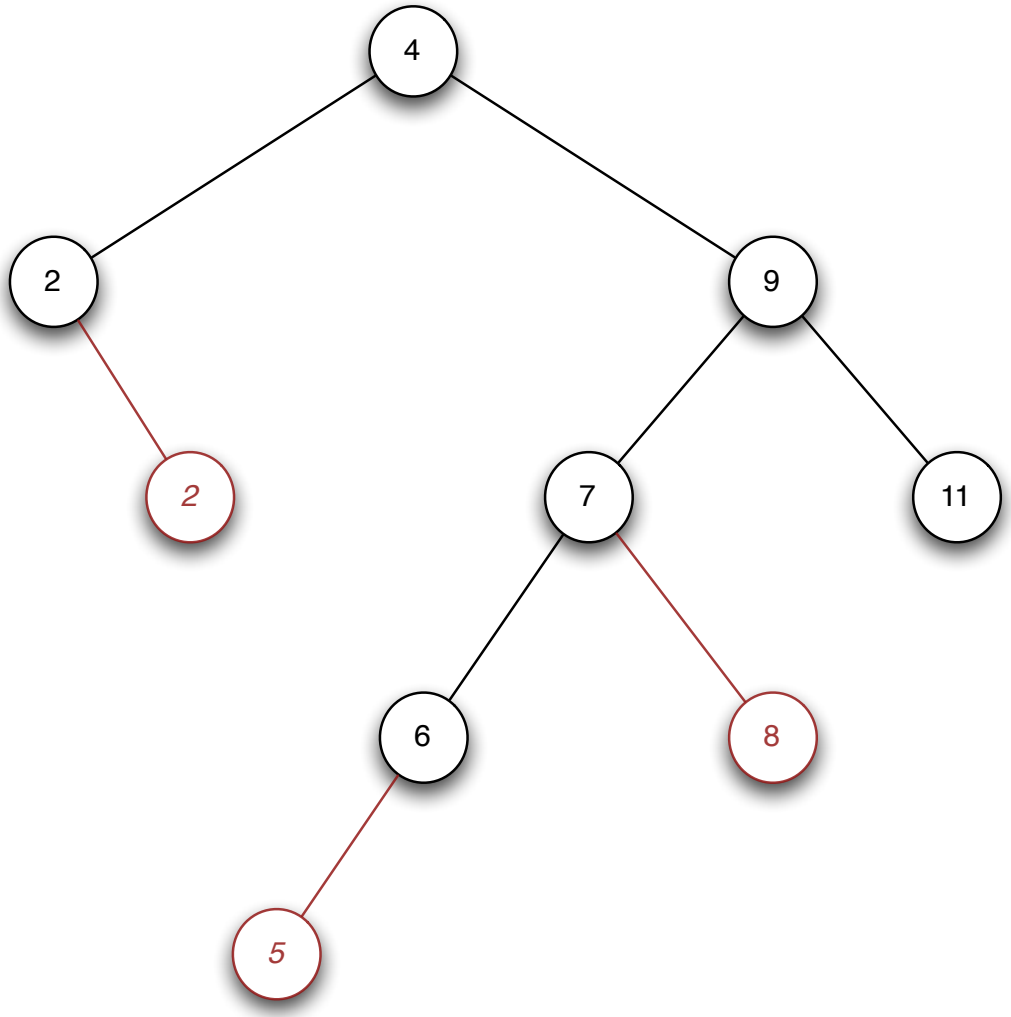
Afin de ne pas modifier la structure courante d'un arbre binaire de recherche, un nouvel élément est toujours inséré en tant que feuille de l'arbre

Si on insère un nœud ayant une information déjà présente dans l'arbre, on insère ce nouvel élément dans le sous-arbre droit du nœud qui est déjà présent dans l'arbre et qui a la même information

ABR : ajout d'un élément



ABR : ajout d'un élément



ABR : suppression d'un élément

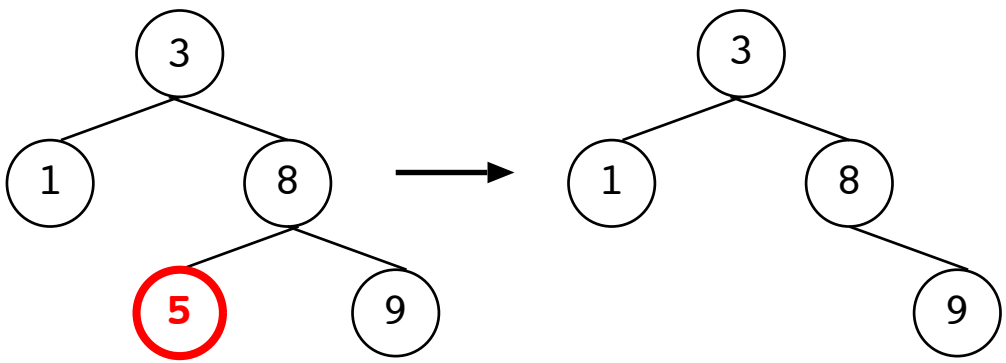
La suppression d'un élément dans un arbre binaire de recherche nécessite de considérer trois configurations possibles

Tout d'abord, il est nécessaire de chercher l'élément x à supprimer dans l'arbre

Une fois trouvé, on distingue les situations (1) où x n'a pas d'enfants, (2) où x n'a pas de fils droit et (3) où x a un fils droit

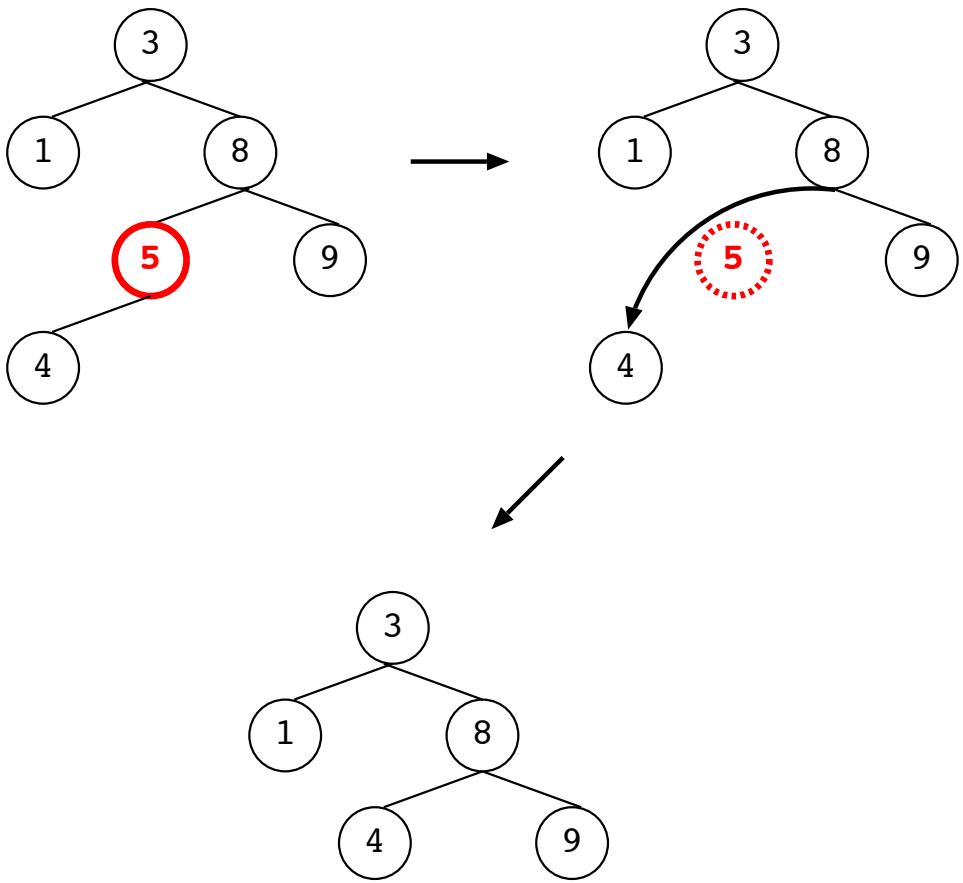
ABR : suppression d'un élément

Si l'élément à supprimer n'a pas d'enfants, on peut le supprimer directement



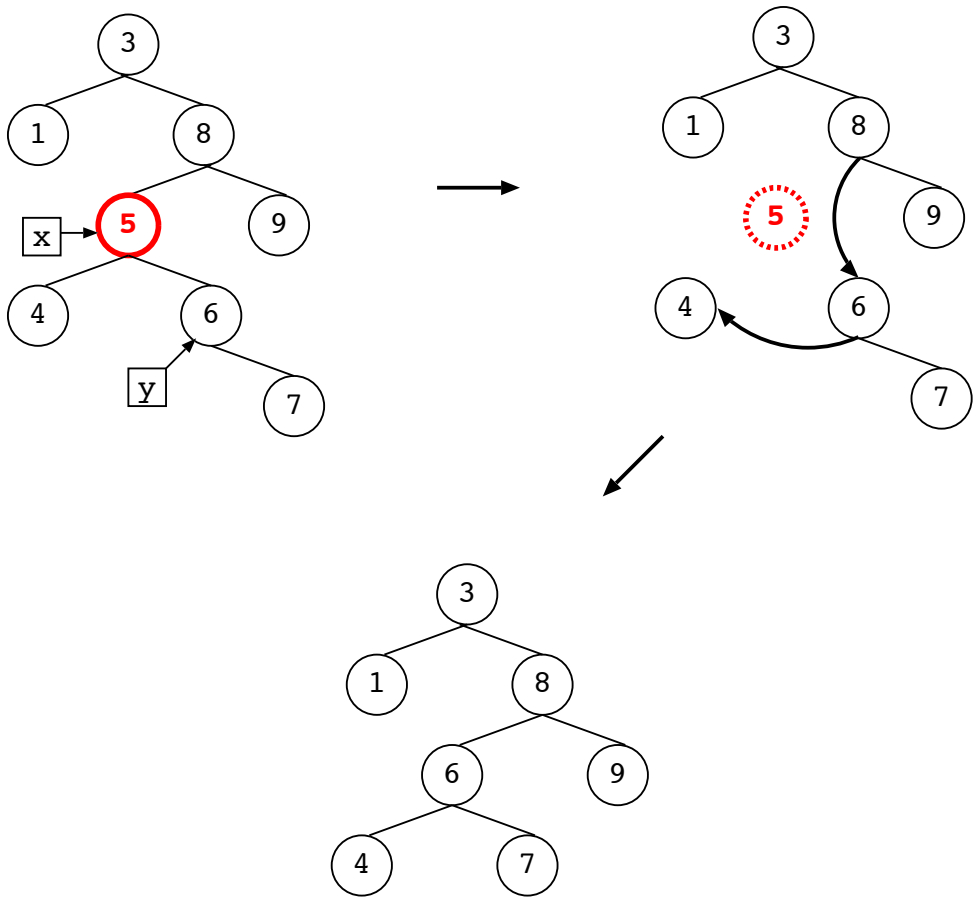
ABR : suppression d'un élément

Si x n'a pas de fils droit on relie l'élément père de x au fils gauche de x et on supprime x



ABR : suppression d'un élément

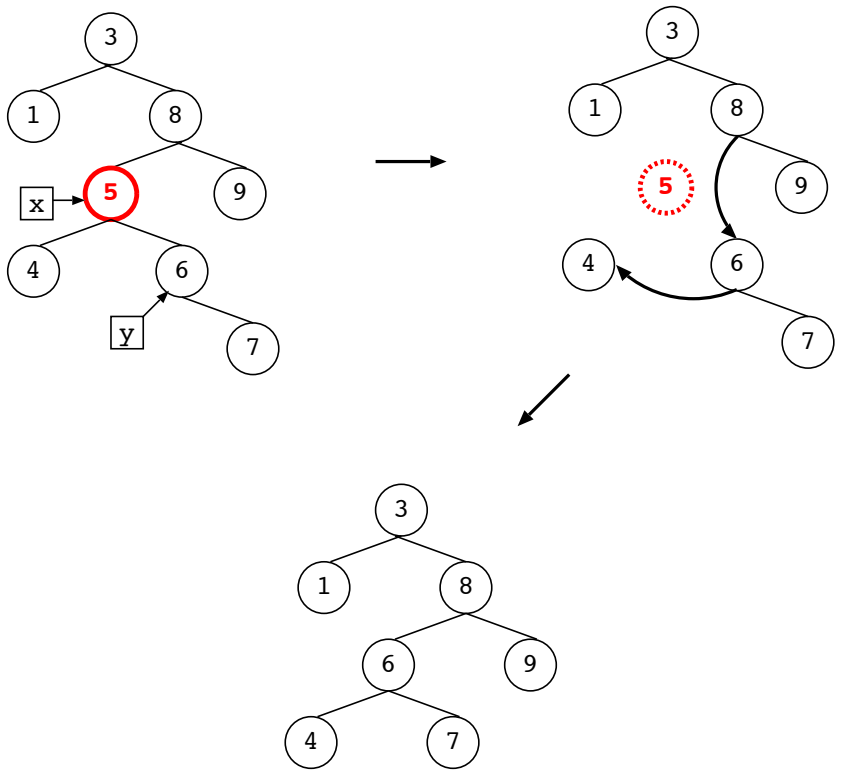
Si x a un fils droit, nous pouvons le remplacer par son « suivant »



ABR : suppression d'un élément

Si l'élément x à supprimer a un fils droit qui n'a pas de fils gauche, alors le « suivant » y de x est le fils droit de x

Pour supprimer x, il faudra assigner le fils gauche de x au fils gauche de y, puis relier l'élément père de x à y

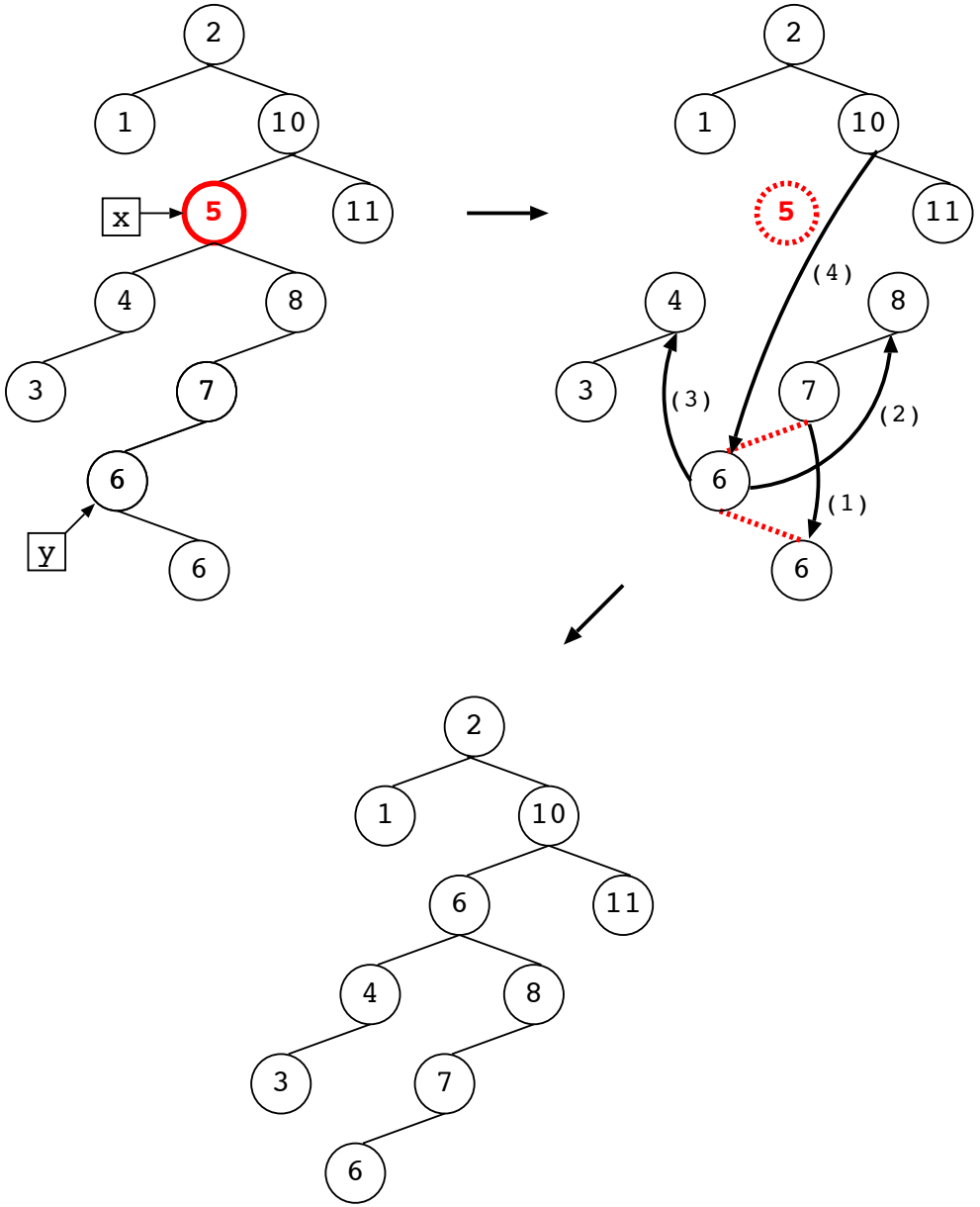


ABR : suppression d'un élément

Si l'élément x à supprimer a un fils droit qui a un fils gauche, alors soit y le « suivant » de x (y est donc le dernier fils gauche de la séquence de fils gauches du fils droit de x)

Pour supprimer x, il faudra (1) assigner le fils droit de y au fils gauche du père de y, (2) assigner le fils droit de x au fils droit de y, (3) assigner le fils gauche de x au fils gauche de y, (4) puis relier l'élément père de x à y

ABR : suppression d'un élément



ABR : suppression d'un élément

1. On cherche l'élément X à supprimer
2.
   - si X a un fils gauche mais n'a pas de fils droit :
     - soit Y le fils gauche de X
     - le père de Y = le père de X
   - Si X a un fils droit :
     - si le fils droit de X n'a pas de fils gauche : (1) soit Y le fils droit de X ; (2) le fils gauche de Y = le fils gauche de X ; (3) le père de l'éventuel fils gauche de X = Y ; (4) le père de Y = le père X ;
     - si le fils droit de X a une séquence de fils gauches : (1) soit Y le dernier fils gauche (ce Y doit remplacer X) ; (2) le fils gauche du père de Y = le fils droit de Y ; (3) le père de l'éventuel fils droit de Y = le père de Y ; (4) le fils droit de Y = le fils droit de X ; (5) le père du fils droit de X = Y ; (6) le fils gauche de Y = le fils gauche de X ; (7) le père de l'éventuel fils gauche de X = Y ; (8) le père de Y = le père de X ;
3. On supprime X
4. Le fils gauche ou droit du père de l'élément effacé = Y

ABR : complexité

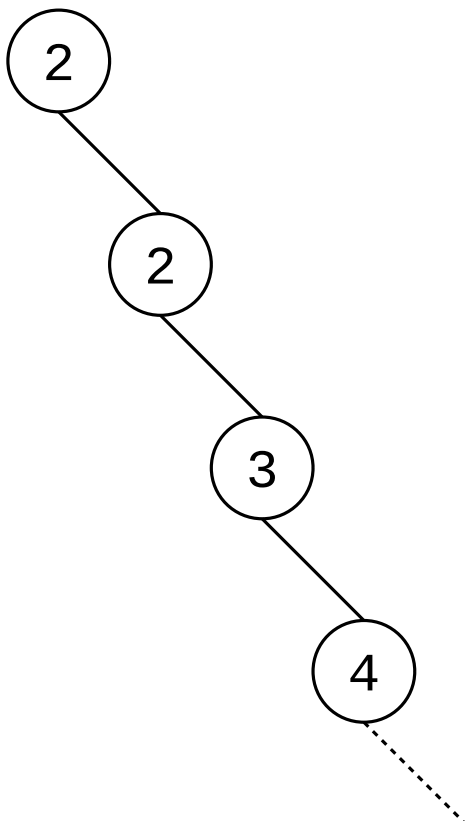
Idéalement, la complexité des différentes opérations à réaliser sur l'arbre binaire de recherche pourrait s'exprimer en $O(\log(n))$ puisqu'il s'agit de parcourir, vers le haut ou le bas, au plus une profondeur de l'arbre

En pratique, ce n'est pas le cas si l'arbre n'est pas équilibré

Une situation « dégénérée » peut ainsi apparaître et mener à des complexités linéaires

ABR : pire cas

La figure ci-dessous représente un arbre induisant une telle dégradation de la complexité des algorithmes de manipulation de l'arbre :



Rebalancement d'un ABR

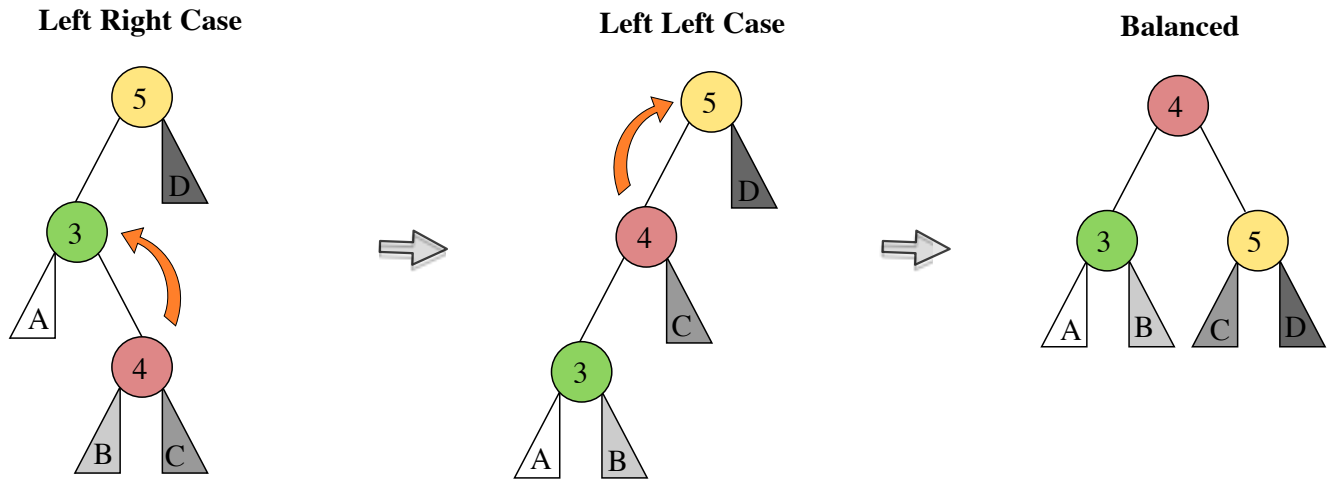
Afin d'éviter la dégradation des performances des différents algorithmes manipulant des arbres binaires de recherche, il peut être nécessaire de *rebalancer* l'arbre afin d'équilibrer pour tout nœud la profondeur de son sous-arbre gauche et de son sous-arbre droit

Rebalancement d'un ABR

Il est possible de faire en sorte que l'arbre binaire de recherche ne « dégénère » pas, on parle alors d'arbres binaires de recherche automatiquement équilibrés

Un exemple d'arbres binaires de recherche automatiquement équilibrés sont les arbres AVL de Georgy Adelson-Velskii et Evgeny Landis : l'arbre binaire de recherche est maintenu équilibré au moyen de *rotations*

Rebalancement d'un ABR



. CyHawk, Wikipedia page on AVL tree

ABR : implémentation

Pour simplifier l'implémentation de notre arbre binaire de recherche, nous allons l'écrire récursivement

Nous allons ainsi pouvoir nous inspirer de notre implémentation récursive d'arbres binaires

Nous allons ainsi aussi pouvoir écrire récursivement certaines primitives de notre arbre binaire de recherche

ABR : suppression récursive

Une des primitives qui subit une importante mutation lorsqu'on l'envisage récursivement est la primitive de suppression d'un nœud

La version récursive de la suppression d'un nœud, si elle est plus claire que la version itérative détaillée précédemment, peut être néanmoins moins efficace (certaines mêmes opérations peuvent être réalisées plusieurs fois au cours des différents appels récursifs)

De même, l'arbre binaire de recherche résultant de la suppression peut être, avec la version récursive, moins bien balancé qu'après exécution de la version itérative

ABR : suppression récursive

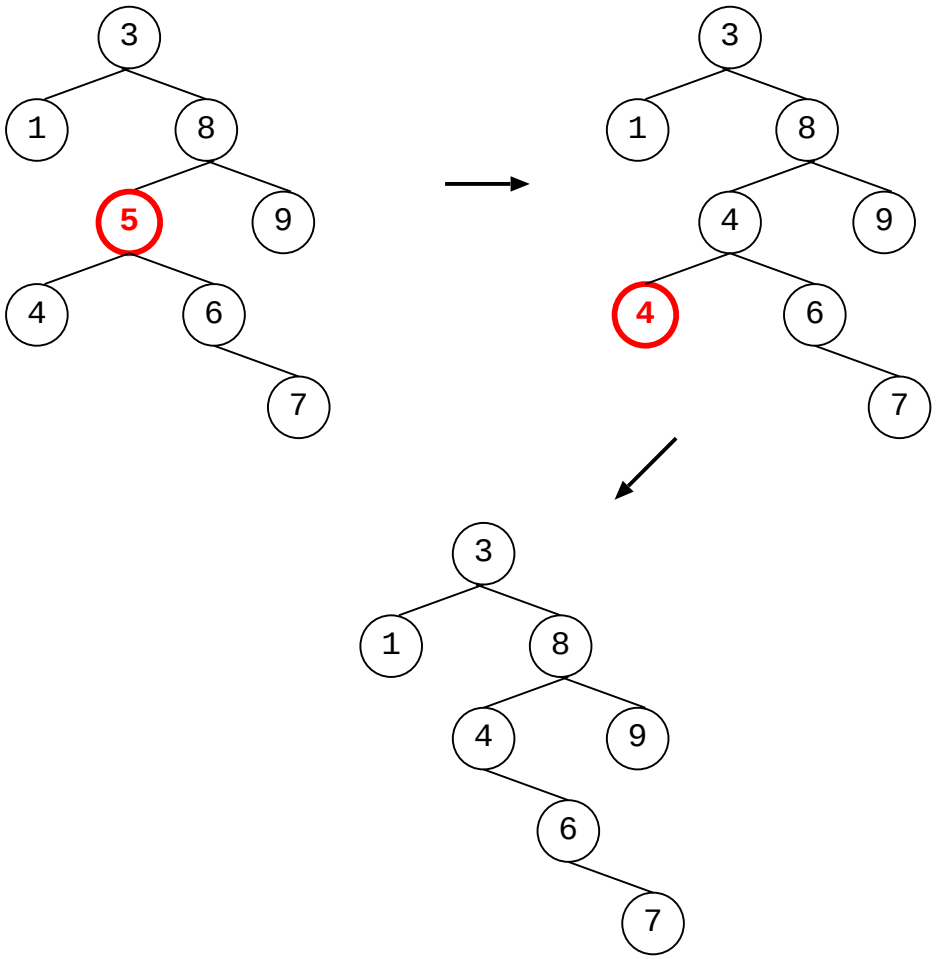
L'algorithme suppose recevoir un pointeur vers l'élément `X` à supprimer

1. Si le noeud X a un fils gauche
   - soit Y l'élément précédant X
   - l'information de Y est recopiée dans X
   - l'élément Y est supprimé (appel récursif)
2. Si le noeud X n'a pas de fils gauche, mais a un fils droit
   - soit Y l'élément suivant X
   - l'information de Y est recopiée dans X
   - l'élément Y est supprimé (appel récursif)
3. Si le noeud X n'a ni fils gauche, ni fils droit
   - l'élément X est retiré de l'arbre (pas d'appel récursif)

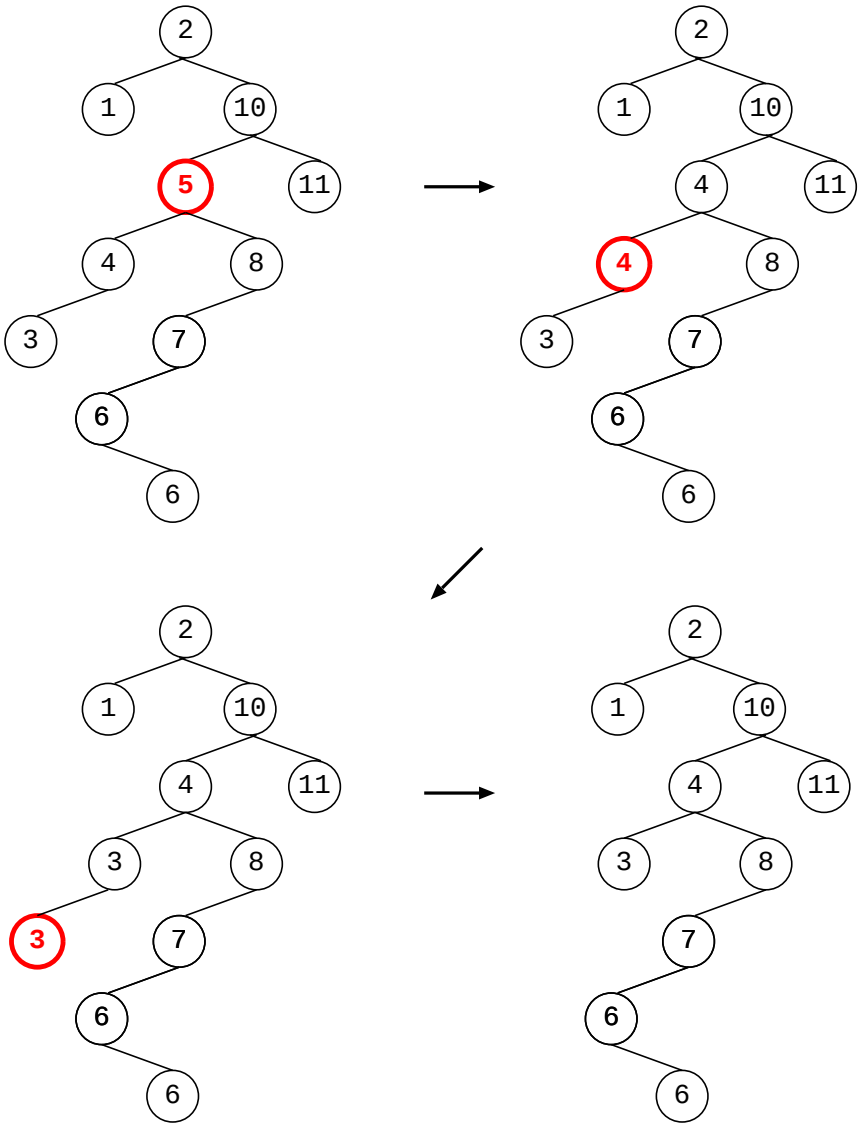
ABR : suppression récursive

Exemple : si l'élément à supprimer, possède un fils gauche : on copie dans l'élément à supprimer la valeur de son précédent et on supprime (récursivement) ce précédent

ABR : suppression récursive (exemple 1)



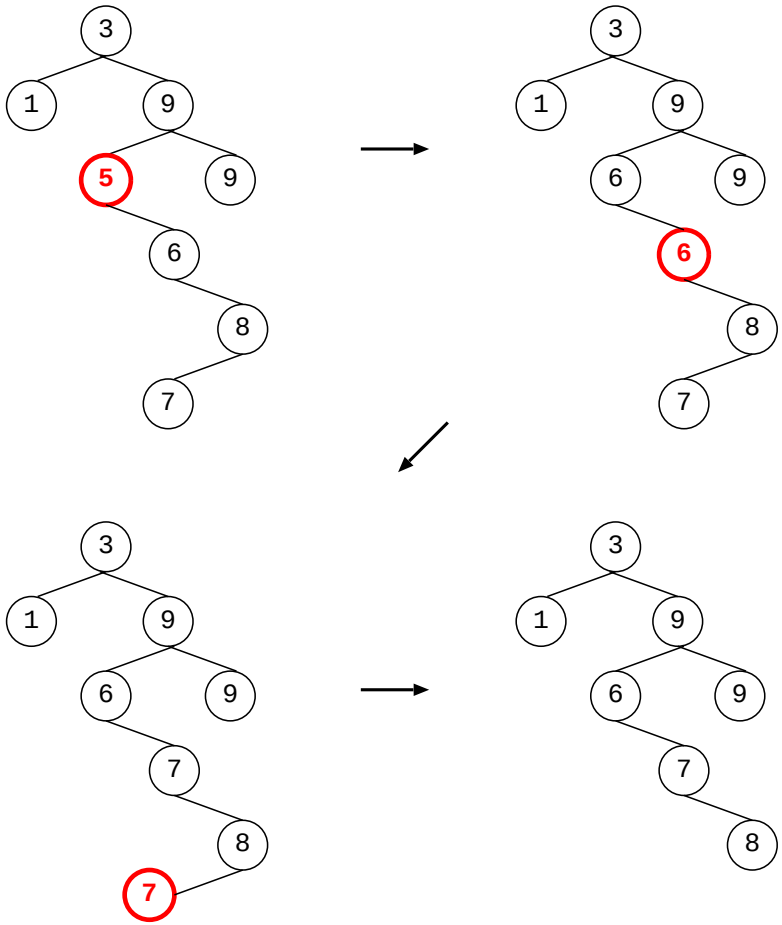
ABR : suppression récursive (exemple 2)



ABR : suppression récursive

Exemple : si l'élément à supprimer ne possède pas de fils gauche mais possède un fils droit : on copie dans l'élément à supprimer la valeur de son suivant et on supprime (récursivement) ce suivant

ABR : suppression récursive (exemple 3)



Arbre binaire de recherche

```
class BinarySearchTree:
    def __init__(self,item):
        self.info = item
        self.left = None
        self.right = None
        self.father = None

    def getRootVal(self):
        return self.info

    def setRootVal(self,item):
        self.info = item

    def getData(self):
        return self.info
```

Arbre binaire de recherche

```
    def getGlobalRoot(self):
        res = self
        if res != None:
            while res.father != None:
                res = res.father
        return res
```

Arbre binaire de recherche

```
    def getNext(self):
        if self.right != None:
            res = self.right
            while res.left != None:
                res = res.left
        else:
            fils = self
            res = self.father
            while res != None and res.right == fils:
                fils = res
                res = res.father
        return res
```

Arbre binaire de recherche

```
    def getPrevious(self):
        if self.left != None:
            res = self.left
            while res.right != None:
                res = res.right
        else:
            fils = self
            res = self.father
            while res != None and res.left == fils:
                fils = res
                res = res.father
        return res
```

Arbre binaire de recherche

```
    def getFirst(self):
        res = self.getGlobalRoot()
        while res.left != None:
            res = res.left
        return res
```

Arbre binaire de recherche

```
    def getLast(self):
        res = self.getGlobalRoot()
        while res.right != None:
            res = res.right
        return res
```

Arbre binaire de recherche

```
    def insert(self,item):
        res = self.getGlobalRoot()
        while res != None:
            p = res
            if item < p.info:
                res = res.left
            else:
                res = res.right
        if item < p.info:
            p.left = BinarySearchTree(item)
            p.left.father = p
        else:
            p.right = BinarySearchTree(item)
            p.right.father = p
```

Arbre binaire de recherche

```
    def find(self,item):
        res = self.getGlobalRoot()
        while res != None and res.info != item:
            if item < res.info:
                res = res.left
            else:
                res = res.right
        return res
```

Arbre binaire de recherche

```
    def remove(self):
        n = self
        if self.left != None:
            q = self.getPrevious()
            n.info = q.info
            q.remove()
        elif self.right != None:
            q = self.getNext()
            n.info = q.info
            q.remove()
        else:
            if n.father != None:
                if n.father.left == n:
                    n.father.left = None
                else:
                    n.father.right = None
```