

Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon

`dominique.poulalhon@irif.fr`

Université de Paris

L2 Informatique & DL Bio-Info, Jap-Info, Math-Info

Année universitaire 2021-2022

Rappel : partiel vendredi 18 mars, 16h30-18h

- amphi 1A : de A à D
- amphi 8C : de E à MOHUN
- amphi 13E : de MOKEDDES à Z

Programme : cours n° 1 à 7

2^e évaluation moodle a priori *mardi 19 avril après-midi*
(pas de CM de LC4 et TD d'EA4 du groupe INFO2 déplaçable)

Arbres Binaires de Recherche

I. Motivations – Rappels

QUELLE STRUCTURE DE DONNÉES POUR REPRÉSENTER UN ENSEMBLE ?

Solution 1 : une liste de ses éléments

QUELLE STRUCTURE DE DONNÉES POUR REPRÉSENTER UN ENSEMBLE ?

Solution 1 : une liste de ses éléments

Problème : la complexité *en espace* peut être sensiblement plus importante que nécessaire s'il y a des éléments répétés

QUELLE STRUCTURE DE DONNÉES POUR REPRÉSENTER UN ENSEMBLE ?

Solution 1 : une liste de ses éléments

Problème : la complexité *en espace* peut être sensiblement plus importante que nécessaire s'il y a des éléments répétés

c'est aussi un problème pour la complexité *en temps* : si un ensemble de n éléments est représenté par une liste (*avec répétitions*) de longueur ℓ , la complexité *en temps* de n'importe quel algo (parcours, recherche dichotomique...) dépend de ℓ et non de n

QUELLE STRUCTURE DE DONNÉES POUR REPRÉSENTER UN ENSEMBLE ?

Solution 1 : une liste de ses éléments

Problème : la complexité *en espace* peut être sensiblement plus importante que nécessaire s'il y a des éléments répétés

c'est aussi un problème pour la complexité *en temps* : si un ensemble de n éléments est représenté par une liste (*avec répétitions*) de longueur ℓ , la complexité *en temps* de n'importe quel algo (parcours, recherche dichotomique...) dépend de ℓ et non de n

donc : il est préférable d'interdire les doublons

QUELLE STRUCTURE DE DONNÉES POUR REPRÉSENTER UN ENSEMBLE ?

Solution 1 (bis) : une liste de ses éléments, sans doublon

<i>(dans le pire cas)</i>	tableau		liste chaînée	
	non trié	trié	non triée	triée
recherche d'un élément	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
minimum/maximum		$\Theta(1)$		$\Theta(1)$
sélection du k^e				$\Theta(k)$
union/intersection/...	$\Theta(n^2)$ (*) $\Theta(n \log n)$ (**)	$\Theta(n)$	$\Theta(n^2)$ (*) $\Theta(n \log n)$ (**)	$\Theta(n)$

(*) sans trier (**) en triant

QUELLE STRUCTURE DE DONNÉES POUR REPRÉSENTER UN ENSEMBLE ?

Solution 1 (bis) : une liste de ses éléments, sans doublon

<i>(dans le pire cas)</i>	tableau		liste chaînée	
	non trié	trié	non triée	triée
recherche d'un élément	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
minimum/maximum		$\Theta(1)$		$\Theta(1)$
sélection du k^e				$\Theta(k)$
union/intersection/...	$\Theta(n^2)$ (*) $\Theta(n \log n)$ (**)	$\Theta(n)$	$\Theta(n^2)$ (*) $\Theta(n \log n)$ (**)	$\Theta(n)$

(*) sans trier (**) en triant

Justification rapide, consulter les cours précédents si nécessaire :

- pour la recherche : linéaire *vs* dichotomique
- pour la sélection : sélection rapide (complexité en moyenne, pas au pire) *vs* accès direct ou parcours des k premiers maillons
- pour l'union ou l'intersection : via la fusion de listes triées

QUELLE STRUCTURE DE DONNÉES POUR REPRÉSENTER UN ENSEMBLE ?

Solution 1 (bis) : une liste de ses éléments, sans doublon

(dans le pire cas)	tableau		liste chaînée	
	non trié	trié	non triée	triée
recherche d'un élément	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
minimum/maximum		$\Theta(1)$		$\Theta(1)$
sélection du k ^e				
union/intersection/...	$\Theta(n^2)$ (*) $\Theta(n \log n)$ (**)	$\Theta(n)$	$\Theta(n^2)$ (*) $\Theta(n \log n)$ (**)	$\Theta(n)$

(*) sans trier (**) en triant

⇒ pour ces opérations *statiques*, supériorité nette du tableau trié

QUELLE STRUCTURE DE DONNÉES POUR REPRÉSENTER UN ENSEMBLE ?

Solution 1 (bis) : une liste de ses éléments, sans doublon

Quid des opérations *dynamiques* ?

QUELLE STRUCTURE DE DONNÉES POUR REPRÉSENTER UN ENSEMBLE ?

Solution 1 (bis) : une liste de ses éléments, sans doublon

Quid des opérations *dynamiques* ?

	tableau		liste chaînée	
	non trié	trié	non triée	triée
recherche d'un élément	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
insertion	$+\Theta(1)$	$+\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$
suppression	$+\Theta(n)$	$+\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$

(chaque insertion/suppression nécessite une recherche préalable)

QUELLE STRUCTURE DE DONNÉES POUR REPRÉSENTER UN ENSEMBLE ?

Solution 1 (bis) : une liste de ses éléments, sans doublon

Quid des opérations *dynamiques* ?

	tableau		liste chaînée	
	non trié	trié	non triée	triée
recherche d'un élément	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
insertion	$+\Theta(1)$	$+\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$
suppression	$+\Theta(n)$	$+\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$

(chaque insertion/suppression nécessite une recherche préalable)

⇒ coût linéaire pour chaque forme de liste !

Le tableau est une forme trop rigide pour permettre des évolutions peu coûteuses : tout nouvel élément n'a qu'une seule place possible, et lui faire une place au bon endroit ne peut pas se faire uniquement via des modifications locales.

QUELLE STRUCTURE DE DONNÉES POUR REPRÉSENTER UN ENSEMBLE ?

Moralité : il faudrait trouver une structure ayant à la fois

- la **souplesse** de la liste chaînée,
- le caractère « **ordonné** » des listes triées,
- un moyen d'**accès rapide** à n'importe quel élément

(peut-être pas en $O(1)$, mais pas en $\Theta(n)$)

(et toujours sans doublon)

QUELLE STRUCTURE DE DONNÉES POUR REPRÉSENTER UN ENSEMBLE ?

Moralité : il faudrait trouver une structure ayant à la fois

- la **souplesse** de la liste chaînée,
- le caractère « **ordonné** » des listes triées,
- un moyen d'**accès rapide** à n'importe quel élément

(peut-être pas en $O(1)$, mais pas en $\Theta(n)$)

(et toujours sans doublon)

Solution 2 : un arbre binaire, « trié », sans doublon

QUELLE STRUCTURE DE DONNÉES POUR REPRÉSENTER UN ENSEMBLE ?

Moralité : il faudrait trouver une structure ayant à la fois

- la **souplesse** de la liste chaînée,
- le caractère « **ordonné** » des listes triées,
- un moyen d'**accès rapide** à n'importe quel élément

(peut-être pas en $O(1)$, mais pas en $\Theta(n)$)

(et toujours sans doublon)

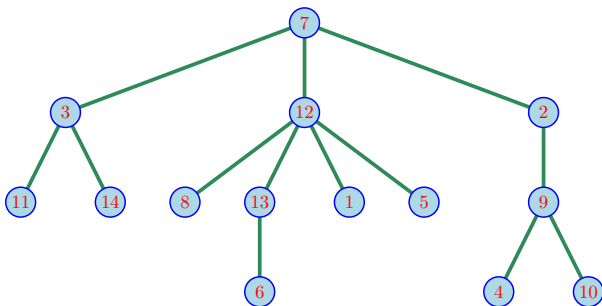
Solution 2 : un arbre binaire, « trié », sans doublon

- multiples formes possibles \implies souplesse pour ajouter ou supprimer un élément en se contentant de modifications locales,
- on veut ranger les éléments – les petits à gauche, et les grands à droite, pour guider la recherche,
- si l'arbre est « à peu près » équilibré, aucun élément ne sera « très loin » de la racine (on peut espérer une hauteur en $\Theta(\log n)$).

Arbres Binaires de Recherche

II. Généralités sur les arbres

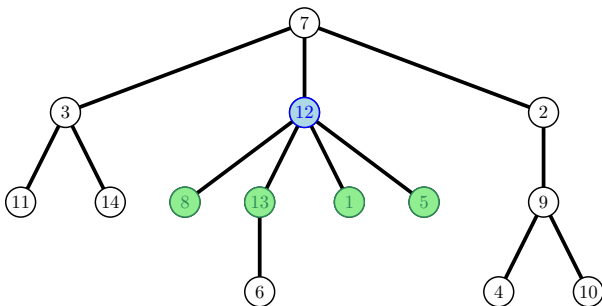
RAPPELS DE TERMINOLOGIE



sommets contenant des étiquettes reliés par des arêtes

Ici, 14 sommets, reliés par 13 arêtes, et étiquetés de 1 à 14

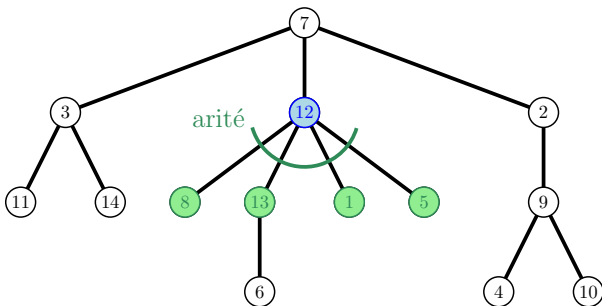
RAPPELS DE TERMINOLOGIE



hiérarchie entre les sommets : père, fils

Le sommet d'étiquette 12 a 4 fils (d'étiquettes 8, 13, 1 et 5).

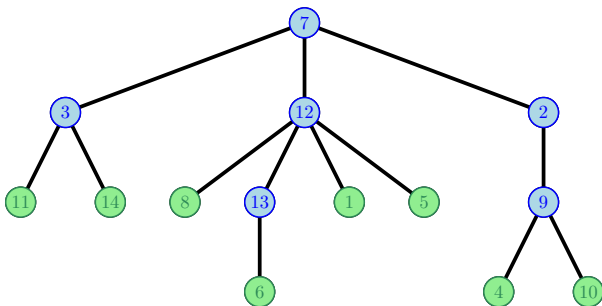
RAPPELS DE TERMINOLOGIE



hiérarchie entre les sommets : père, fils

Le sommet d'étiquette 12 a 4 fils (d'étiquettes 8, 13, 1 et 5).
On dit qu'il est d'arité 4, ou que c'est un nœud quaternaire.

RAPPELS DE TERMINOLOGIE

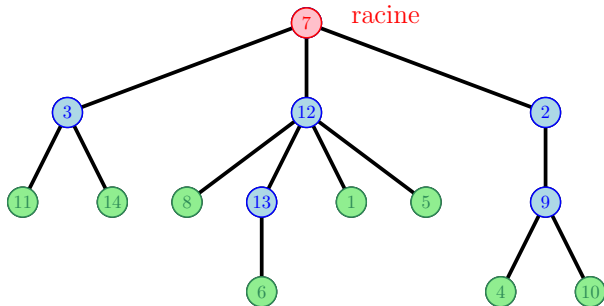


sommet = **nœud** ou **feuille**

Les sommets d'arité 0 sont appelés *feuilles* – ici il y en a 8.

Les autres sont les *nœuds* – ici, 6.

RAPPELS DE TERMINOLOGIE



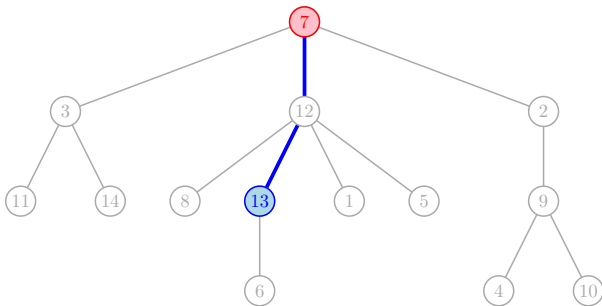
sommet = **nœud** ou **feuille**

Les sommets d'arité 0 sont appelés *feuilles* – ici il y en a 8.

Les autres sont les *nœuds* – ici, 6.

Le seul sommet sans père est la racine – c'est en général un nœud.

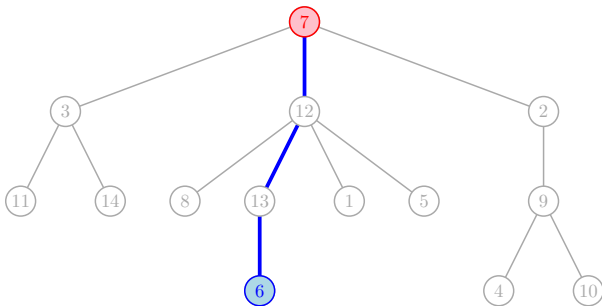
RAPPELS DE TERMINOLOGIE



profondeur d'un sommet = distance à la racine

Le sommet d'étiquette 13 est à profondeur 2.

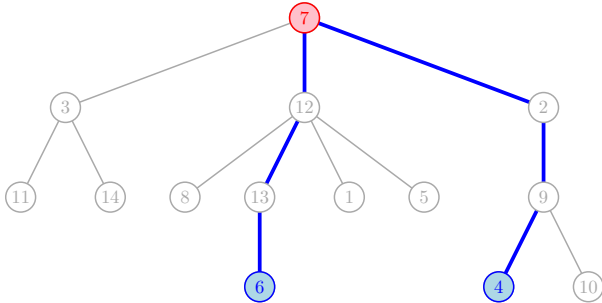
RAPPELS DE TERMINOLOGIE



hauteur de l'arbre = profondeur maximale

La hauteur de cet arbre est 3.

RAPPELS DE TERMINOLOGIE



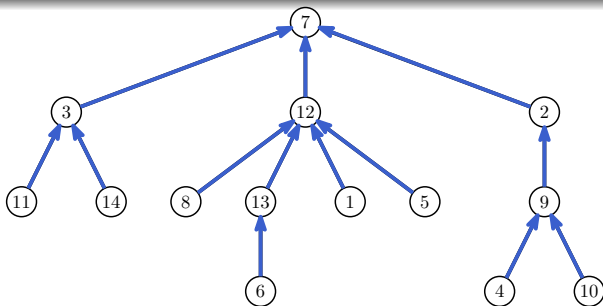
hauteur de l'arbre = profondeur maximale

La hauteur de cet arbre est 3.

Les sommets à profondeur 3 sont donc des feuilles
(et il peut très bien y en avoir plusieurs)

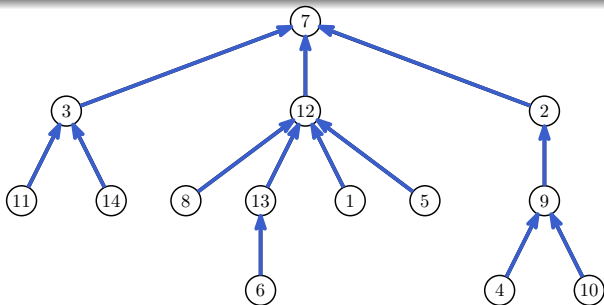
REPRÉSENTATION DES ARBRES – SOLUTION I

en gardant pour chaque sommet la **référence du père** (dans le sommet, ou regroupées dans un tableau)



REPRÉSENTATION DES ARBRES – SOLUTION I

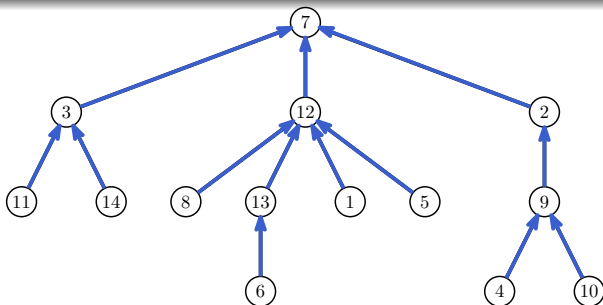
en gardant pour chaque sommet la **référence du père** (dans le sommet, ou regroupées dans un tableau)



avantage : représentation extrêmement compacte

REPRÉSENTATION DES ARBRES – SOLUTION I

en gardant pour chaque sommet la **référence du père** (dans le sommet, ou regroupées dans un tableau)



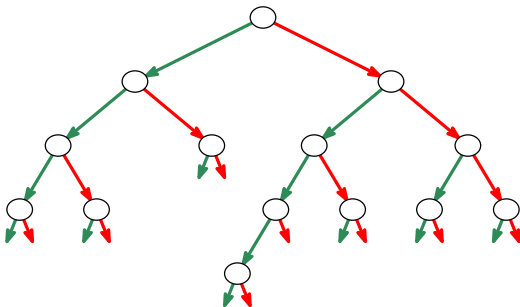
avantage : représentation extrêmement compacte

(gros) inconvénient : l'arbre ne peut être parcouru (efficacement) que de bas en haut... ce qui n'est pas ce qu'on souhaite faire en général

⇒ essentiellement utilisée pour représenter une partition en sous-ensembles (structure *union-find*, cf cours d'algo de L3)

REPRÉSENTATION DES ARBRES – SOLUTION II

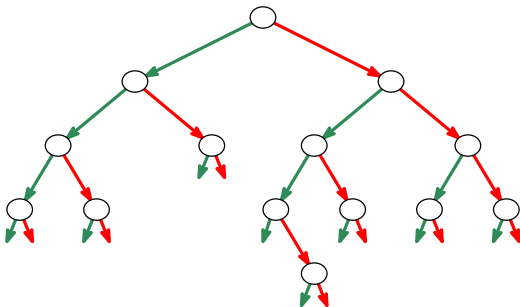
cas des arbres d'arité fixe, en particulier les arbres binaires : pour chaque nœud, stocker une référence de chaque fils



arbre binaire : au plus 2 fils par nœud, avec distinction entre fils **gauche** et fils **droit**. *(échanger les deux fils donne un arbre différent)*

REPRÉSENTATION DES ARBRES – SOLUTION II

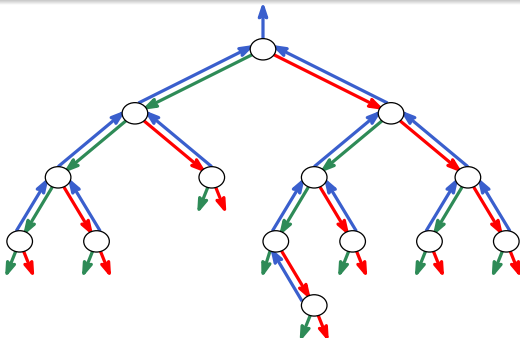
cas des arbres d'arité fixe, en particulier les arbres binaires : pour chaque nœud, stocker une référence de chaque fils



arbre binaire : au plus 2 fils par nœud, avec distinction entre fils **gauche** et fils **droit**. *(échanger les deux fils donne un arbre différent)*

REPRÉSENTATION DES ARBRES – SOLUTION II

cas des arbres d'arité fixe, en particulier les arbres binaires : pour chaque nœud, stocker une référence de chaque fils (et éventuellement du père)

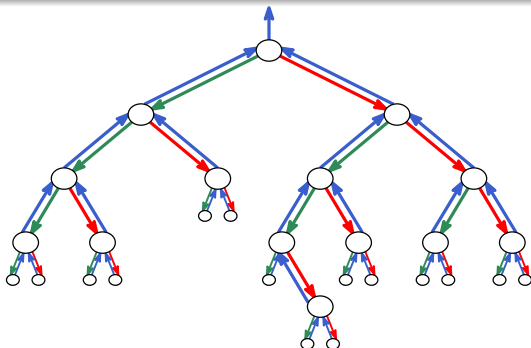


arbre binaire : au plus 2 fils par nœud, avec distinction entre fils **gauche** et fils **droit**.
(échanger les deux fils donne un arbre différent)

ajouter un 3^e pointeur vers le père facilite les manipulations, notamment les modifications (comme le double chaînage d'une liste)

REPRÉSENTATION DES ARBRES – SOLUTION II

cas des arbres d'arité fixe, en particulier les arbres binaires : pour chaque nœud, stocker une référence de chaque fils (et éventuellement du père)



compléter un arbre binaire = rajouter une feuille vide à la place de chaque pointeur vers un sommet absent

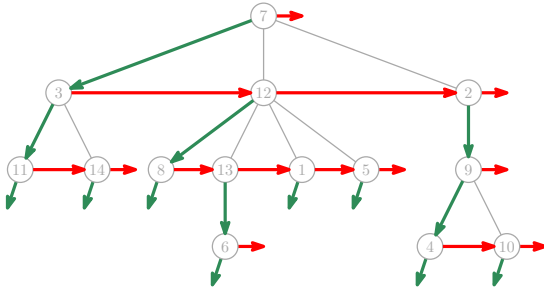
(plus homogène, peut faciliter la programmation des modifications)

arbres binaires *complets* : exactement 2 fils par nœud

REPRÉSENTATION DES ARBRES – SOLUTION III

(Pour la culture générale, mais pas indispensable pour le cours sur les ABR)

cas général : références du **fil** **ainé** et du **frère cadet**



le **fil aîné** sert de **référence** à la liste **chaînée** des fils.

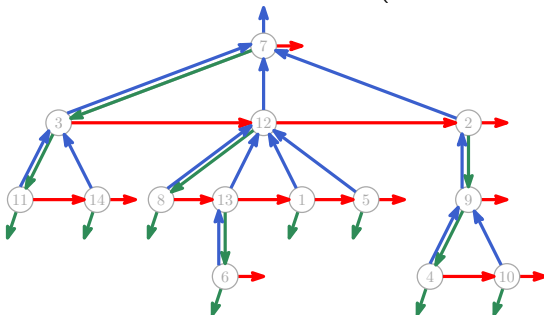
i.e. : on représente les arbres généraux par des arbres binaires (dont la racine n'a pas de fils droit) : le **fil aîné** d'un sommet devient son **fil gauche**, et son **frère cadet** devient son **fils droit**.

REPRÉSENTATION DES ARBRES – SOLUTION III

(Pour la culture générale, mais pas indispensable pour le cours sur les ABR)

cas général : références du fils aîné et du frère cadet

(et éventuellement du père)

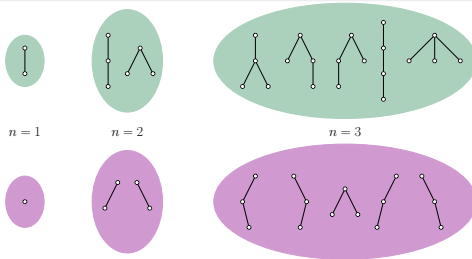


le fils aîné sert de référence à la liste chaînée des fils.

i.e. : on représente les arbres généraux par des arbres binaires (dont la racine n'a pas de fils droit) : le fils aîné d'un sommet devient son fils gauche, et son frère cadet devient son fils droit.

Théorème

Les arbres à $n + 1$ sommets sont en bijection avec les arbres binaires à n sommets, et donc avec les arbres binaires complets à n nœuds et $n + 1$ feuilles



Théorème (admis)

Le nombre d'arbres binaires complets à n nœuds et $n + 1$ feuilles est

$$\frac{1}{n+1} \binom{2n}{n} \in \Theta \left(\frac{4^n}{n\sqrt{n}} \right)$$

REPRÉSENTATION DES ARBRES

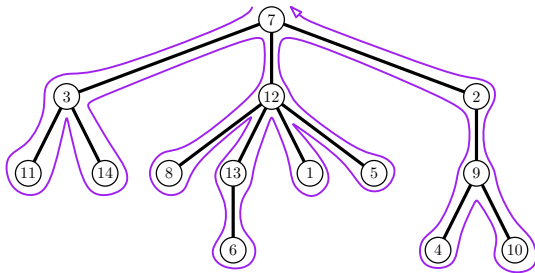
À partir de maintenant, on suppose qu'on dispose des fonctions suivantes, dont le code dépend de la représentation choisie :

- `pere(noeud)`
- `liste_des_fils(noeud)`
- `arite(noeud)`
- `etiquette(noeud)`
- (pour les arbres binaires) `gauche(noeud)` et `droite(noeud)`

Que peut-on faire avec un arbre? Par exemple, peut-on parcourir tous les éléments qu'il contient raisonnablement efficacement, comme une liste?

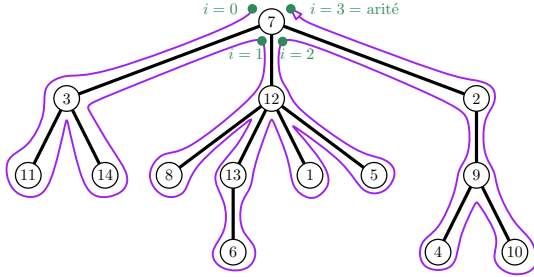
PARCOURS EN PROFONDEUR GÉNÉRIQUE

Une solution : effectuer un **parcours en profondeur** d'un arbre, c'est-à-dire en faire le tour complet, en partant de la racine et en le tenant fermement de la main gauche :



PARCOURS EN PROFONDEUR GÉNÉRIQUE

Une solution : effectuer un **parcours en profondeur** d'un arbre, c'est-à-dire en faire le tour complet, en partant de la racine et en le tenant fermement de la main gauche :



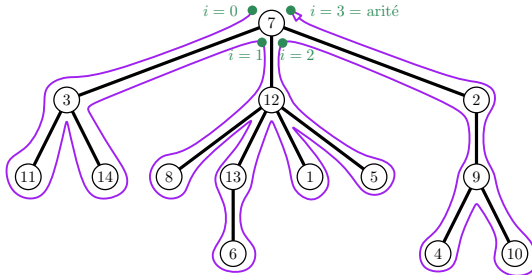
ce tour passe plusieurs fois en chaque sommet – une de plus que son arité

chaque passage en un sommet est l'occasion d'effectuer un traitement : un affichage, un stockage, un décompte... ou rien du tout.

Un tel parcours se décrit particulièrement simplement de façon récursive.

PARCOURS EN PROFONDEUR GÉNÉRIQUE

```
def parcours_generique(racine) :  
    for i, noeud in enumerate(liste_des_fils(racine)) :  
        traitement(i, racine)  
        # = traitement après avoir visité i sous-arbres  
        parcours_generique(noeud)  
    traitement(arite(racine), racine)
```



PARCOURS EN PROFONDEUR GÉNÉRIQUE

```
def parcours_generique(racine) :  
    for i, noeud in enumerate(liste_des_fils(racine)) :  
        traitement(i, racine)  
        # = traitement après avoir visité i sous-arbres  
        parcours_generique(noeud)  
    traitement(arite(racine), racine)
```

Théorème

`parcours_generique(racine)` visite tous les sommets de l'arbre enraciné en `racine`, en temps $\Theta(n)$ si chaque traitement est en $\Theta(1)$

PARCOURS EN PROFONDEUR GÉNÉRIQUE

```
def parcours_generique(racine) :  
    for i, noeud in enumerate(liste_des_fils(racine)) :  
        traitement(i, racine)  
        # = traitement après avoir visité i sous-arbres  
        parcours_generique(noeud)  
    traitement(arite(racine), racine)
```

Théorème

`parcours_generique(racine)` visite tous les sommets de l'arbre enraciné en `racine`, en temps $\Theta(n)$ si chaque traitement est en $\Theta(1)$

- si `i = 0` : on parle de *pré-traitement*
- si `i = arite(racine)` : on parle de *post-traitement*

PARCOURS EN PROFONDEUR SPÉCIFIQUES

Trois cas particuliers (et particulièrement importants) :

Parcours préfixe : `traitement(i, racine)` vide sauf si `i = 0`

```
def parcours_prefixe(racine) :  
    pre_traitement(racine)  
    for noeud in liste_des_fils(racine) : parcours_prefixe(noeud)
```

Parcours postfixe : `traitement(i, racine)` vide sauf si `i = arite(racine)`

```
def parcours_postfixe(racine) :  
    for noeud in liste_des_fils(racine) : parcours_postfixe(noeud)  
    post_traitement(racine)
```

Parcours infixé : cas binaire avec seulement un traitement intermédiaire

```
def parcours_infixe(racine) :  
    if gauche(racine) != None : parcours_infixe(gauche(racine))  
    traitement(racine)  
    if (droite(racine) != None) : parcours_infixe(droite(racine))
```

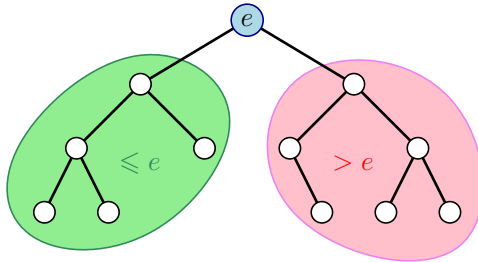
Arbres Binaires de Recherche

III. Définition et manipulations simples

QUE PEUT BIEN SIGNIFIER « TRIER » UN ARBRE ?

Un arbre binaire de recherche (ABR) est un arbre binaire, étiqueté, tel que l'étiquette de *chaque* sommet est comprise entre

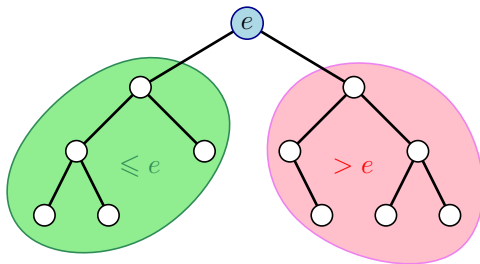
- *toutes les étiquettes du sous-arbre gauche (plus petites)* et
- *toutes les étiquettes du sous-arbre droit (plus grandes)*



QUE PEUT BIEN SIGNIFIER « TRIER » UN ARBRE ?

Un arbre binaire de recherche (ABR) est un arbre binaire, étiqueté, tel que l'étiquette de *chaque* sommet est comprise entre

- *toutes les étiquettes du sous-arbre gauche (plus petites)* et
- *toutes les étiquettes du sous-arbre droit (plus grandes)*

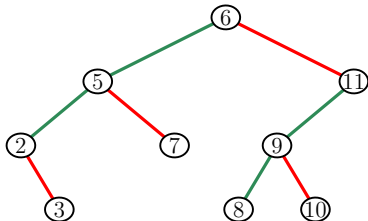
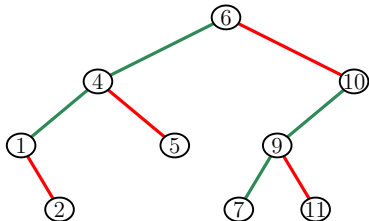
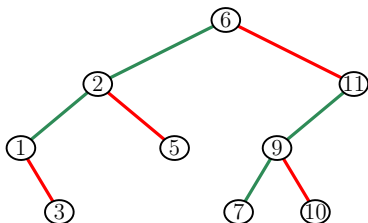
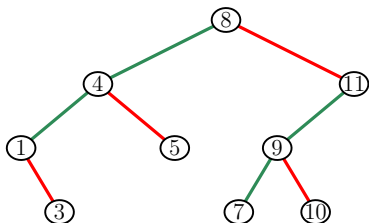


Attention !!!

localement, *cela entraîne* que chaque nœud a une étiquette comprise entre celle de son fils gauche et celle de son fils droit *mais ceci ne suffit pas*

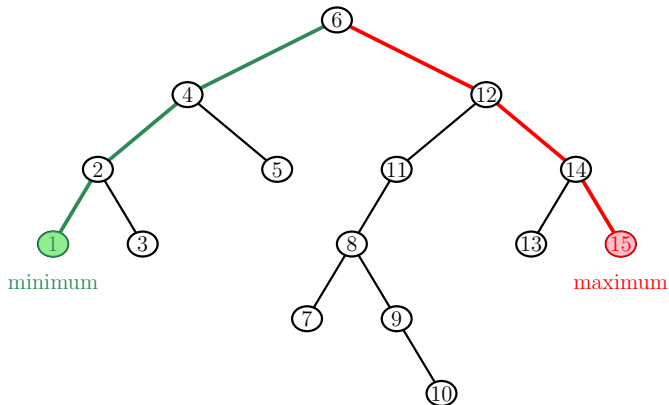
ABR OR NOT ?

Vérifier qu'il n'y a aucun ABR parmi les exemples ci-dessous.



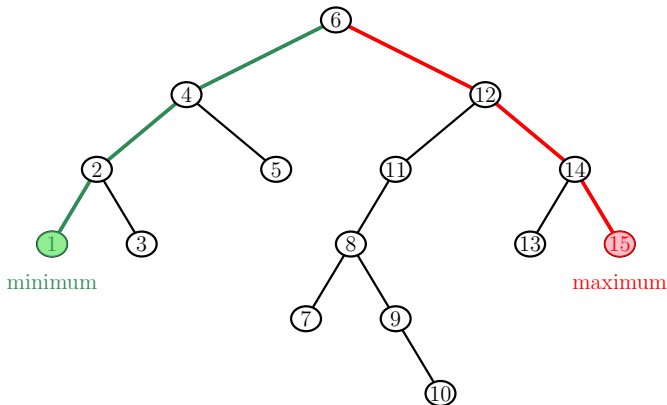
MINIMUM ET MAXIMUM D'UN ABR

La règle définissant les ABR rend certaines recherches faciles...



MINIMUM ET MAXIMUM D'UN ABR

La règle définissant les ABR rend certaines recherches faciles...



Plus généralement, la « disposition spatiale » des éléments suit leur ordre : plus un élément est à gauche, plus il est petit.

ORDRE DANS UN ABR

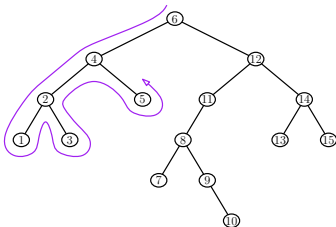
Propriété : dans un ABR, chaque sous-arbre contient un **intervalle** de la liste triée des clés

ORDRE DANS UN ABR

Propriété : dans un ABR, chaque sous-arbre contient un **intervalle** de la liste triée des clés

Un ABR est donc « presque » une liste triée :

```
def liste_triee(noeud) :  
    res = []  
    if noeud != None :  
        res = liste_triee(gauche(noeud))  
        res += [ etiquette(noeud) ]  
        res += liste_triee(droit(noeud))  
    return res
```

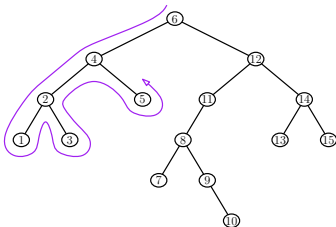


ORDRE DANS UN ABR

Propriété : dans un ABR, chaque sous-arbre contient un **intervalle** de la liste triée des clés

Un ABR est donc « presque » une liste triée :

```
def liste_triee(noeud) :  
    res = []  
    if noeud != None :  
        res = liste_triee(gauche(noeud))  
        res += [ etiquette(noeud) ]  
        res += liste_triee(droit(noeud))  
    return res
```



Théorème

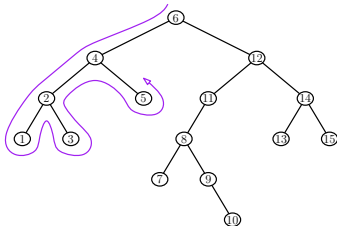
le *parcours infixe* d'un ABR à n nœuds produit la *liste triée* de ses éléments en temps $\Theta(n)$.

ORDRE DANS UN ABR

Propriété : dans un ABR, chaque sous-arbre contient un **intervalle** de la liste triée des clés

Un ABR est donc « presque » une liste triée :

```
def liste_triee(noeud) :  
    res = []  
    if noeud != None :  
        res = liste_triee(gauche(noeud))  
        res += [ etiquette(noeud) ]  
        res += liste_triee(droit(noeud))  
    return res
```



Théorème

le **parcours infixe** d'un ABR à n nœuds produit la **liste triée** de ses éléments en temps $\Theta(n)$.

Corollaire

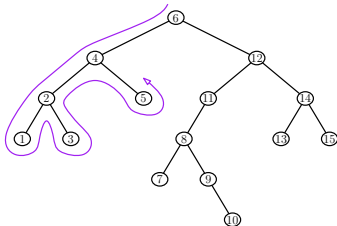
pour déterminer si un arbre est un ABR, il suffit de vérifier si son parcours infixe fournit bien une liste triée

ORDRE DANS UN ABR

Propriété : dans un ABR, chaque sous-arbre contient un **intervalle** de la liste triée des clés

Un ABR est donc « presque » une liste triée :

```
def liste_triee(noeud) :  
    res = []  
    if noeud != None :  
        res = liste_triee(gauche(noeud))  
        res += [ etiquette(noeud) ]  
        res += liste_triee(droit(noeud))  
    return res
```



Théorème

le **parcours infixe** d'un ABR à n nœuds produit la **liste triée** de ses éléments en temps $\Theta(n)$.

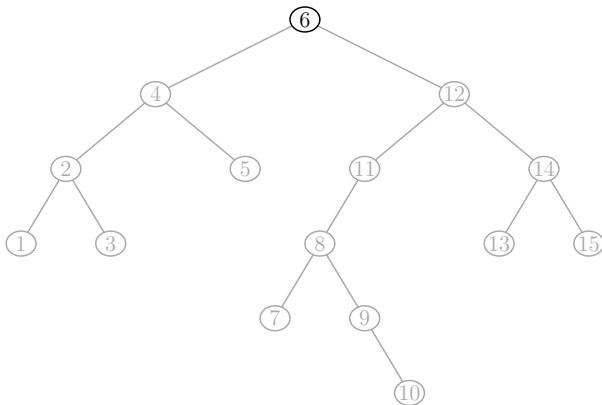
Corollaire

pour déterminer si un arbre est un ABR, il suffit de vérifier si son **parcours infixe** fournit bien une **liste triée** (qu'il n'est même pas nécessaire de construire)

RECHERCHE DANS UN ABR

à la manière de la recherche dichotomique, en comparant l'élément cherché avec l'élément à la racine au lieu de l'élément médian du tableau

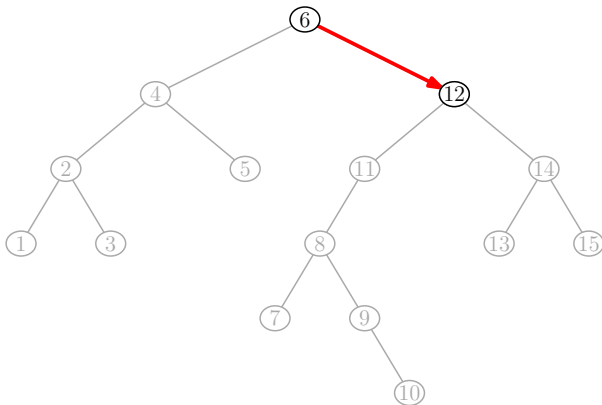
Exemple – recherche de 9 :



RECHERCHE DANS UN ABR

à la manière de la recherche dichotomique, en comparant l'élément cherché avec l'élément à la racine au lieu de l'élément médian du tableau

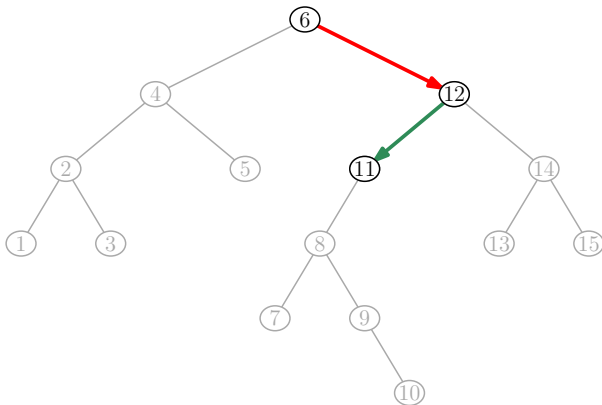
Exemple – recherche de 9 :



RECHERCHE DANS UN ABR

à la manière de la recherche dichotomique, en comparant l'élément cherché avec l'élément à la racine au lieu de l'élément médian du tableau

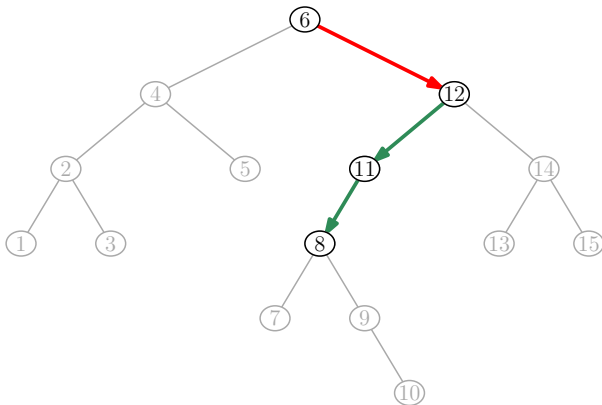
Exemple – recherche de 9 :



RECHERCHE DANS UN ABR

à la manière de la recherche dichotomique, en comparant l'élément cherché avec l'élément à la racine au lieu de l'élément médian du tableau

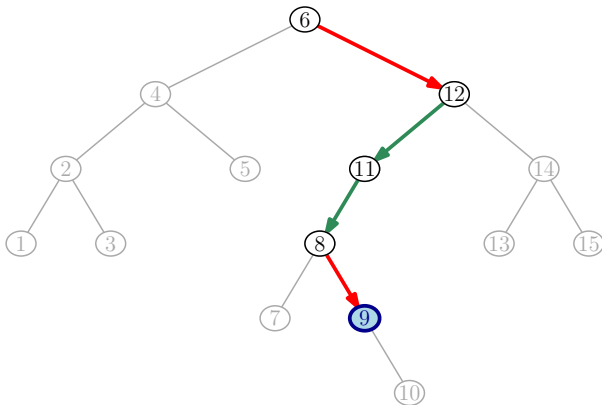
Exemple – recherche de 9 :



RECHERCHE DANS UN ABR

à la manière de la recherche dichotomique, en comparant l'élément cherché avec l'élément à la racine au lieu de l'élément médian du tableau

Exemple – recherche de 9 :



RECHERCHE DANS UN ABR

Cet algorithme s'écrit très simplement de manière récursive :

```
def recherche(noeud, x) :    # version récursive
    if noeud == None : return None
    elif etiquette(noeud) == x : return noeud
    elif etiquette(noeud) > x :
        return recherche(gauche(noeud), x)
    else :
        return recherche(droit(noeud), x)
```

RECHERCHE DANS UN ABR

Cet algorithme s'écrit très simplement de manière récursive :

```
def recherche(noeud, x) :    # version récursive
    if noeud == None : return None
    elif etiquette(noeud) == x : return noeud
    elif etiquette(noeud) > x :
        return recherche(gauche(noeud), x)
    else :
        return recherche(droit(noeud), x)
```

(facile à dérécuriver, puisqu'il y a au plus un appel récursif, terminal)

RECHERCHE DANS UN ABR

Cet algorithme s'écrit très simplement de manière récursive :

```
def recherche(noeud, x) :    # version récursive
    if noeud == None : return None
    elif etiquette(noeud) == x : return noeud
    elif etiquette(noeud) > x :
        return recherche(gauche(noeud), x)
    else :
        return recherche(droit(noeud), x)
```

(facile à dérécuriver, puisqu'il y a au plus un appel récursif, terminal)

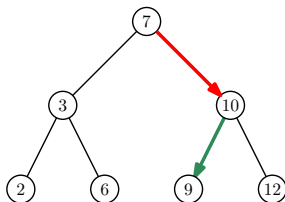
Théorème

$recherche(r, x)$ effectue la recherche d'un élément x dans l'ABR de racine r en temps $\Theta(h)$ au pire, où h est la hauteur de l'ABR.

CAS EXTRÊMES

La recherche (et plus généralement tous les algorithmes sur les ABR) se comportent radicalement différemment selon la **forme** de l'arbre, qui détermine le rapport entre sa **taille** et sa **hauteur**.

Cas sympathique : ABR « parfait »

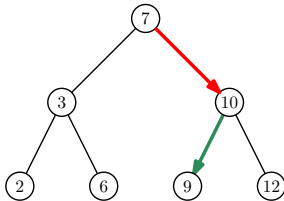


Hauteur $\log n$, recherche aussi efficace que la recherche dichotomique

CAS EXTRÊMES

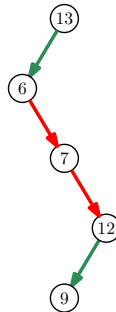
La recherche (et plus généralement tous les algorithmes sur les ABR) se comportent radicalement différemment selon la **forme** de l'arbre, qui détermine le rapport entre sa **taille** et sa **hauteur**.

Cas sympathique : ABR « parfait »



Hauteur $\log n$, recherche aussi efficace que la recherche dichotomique

Cas désagréable : ABR « filiforme »



Hauteur n , recherche aussi inefficace que dans une liste chaînée

CAS PARTICULIERS : MINIMUM/MAXIMUM

```
def minimum(noeud) : # version récursive  
    if gauche(noeud) == None : return noeud  
    return minimum(gauche(noeud))
```

```
def minimum(noeud) : # version itérative  
    while gauche(noeud) != None :  
        noeud = gauche(noeud)  
    return noeud
```

Théorème

minimum(r) détermine le plus petit élément dans l'ABR de racine r en temps $\Theta(h)$ au pire, où h est la hauteur de l'ABR.

SUCCESSEUR D'UN ÉLÉMENT

Un problème un peu plus compliqué :

successeur(n)

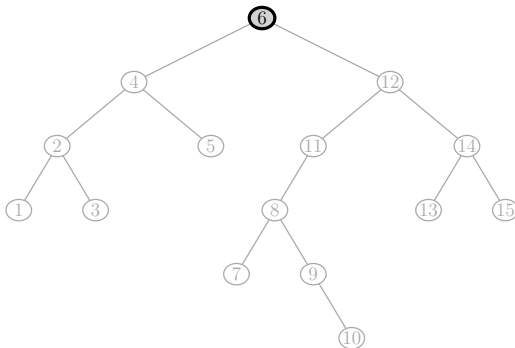
étant donné un nœud n d'un ABR, d'étiquette e , déterminer le nœud de l'arbre ayant la plus petite étiquette supérieure à e .

SUCCESEUR D'UN ÉLÉMENT

successeur(n)

étant donné un nœud n d'un ABR, d'étiquette e , déterminer le nœud de l'arbre ayant la plus petite étiquette supérieure à e .

Cas n° 1 : si le nœud a un fils droit

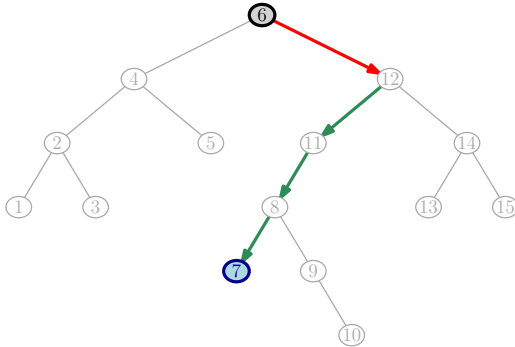


SUCCESEUR D'UN ÉLÉMENT

successeur(n)

étant donné un nœud n d'un ABR, d'étiquette e , déterminer le nœud de l'arbre ayant la plus petite étiquette supérieure à e .

Cas n° 1 : si le nœud a un fils droit



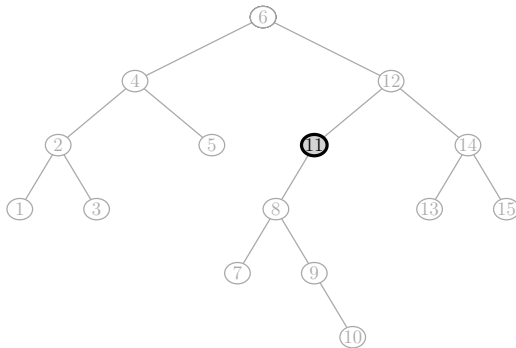
⇒ le successeur est le minimum du sous-arbre droit

SUCCESEUR D'UN ÉLÉMENT

successeur(n)

étant donné un nœud n d'un ABR, d'étiquette e , déterminer le nœud de l'arbre ayant la plus petite étiquette supérieure à e .

Cas n° 2 : si le nœud n n'a pas de fils droit

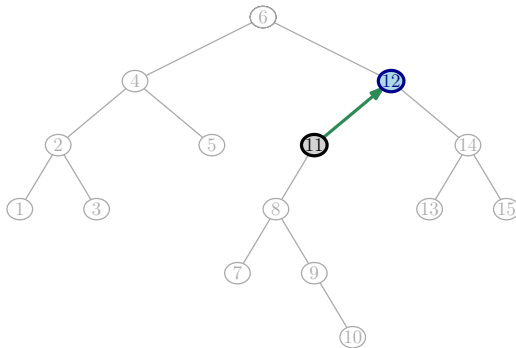


SUCCESEUR D'UN ÉLÉMENT

successeur(n)

étant donné un nœud n d'un ABR, d'étiquette e , déterminer le nœud de l'arbre ayant la plus petite étiquette supérieure à e .

Cas n° 2 : si le nœud n n'a pas de fils droit

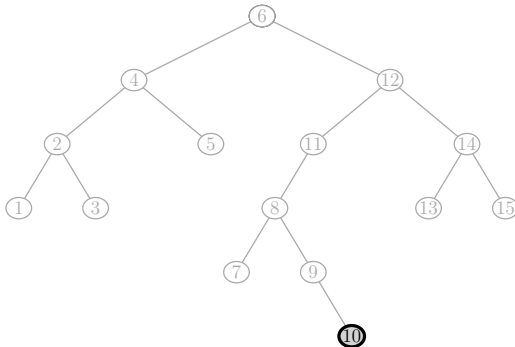


SUCCESSEUR D'UN ÉLÉMENT

successeur(n)

étant donné un nœud n d'un ABR, d'étiquette e , déterminer le nœud de l'arbre ayant la plus petite étiquette supérieure à e .

Cas n° 2 : si le nœud n n'a pas de fils droit

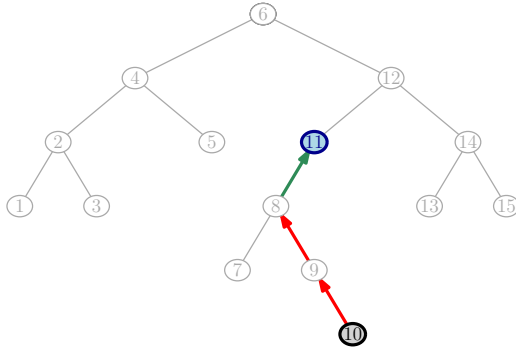


SUCCESSEUR D'UN ÉLÉMENT

successeur(n)

étant donné un nœud n d'un ABR, d'étiquette e , déterminer le nœud de l'arbre ayant la plus petite étiquette supérieure à e .

Cas n° 2 : si le nœud n n'a pas de fils droit



⇒ le successeur est le premier ancêtre supérieur à l'élément – donc le premier vers lequel on remonte depuis la gauche

SUCCESSEUR D'UN ÉLÉMENT

Ce qui donne :

```
def successeur(noeud) :  
    if droit(noeud) != None :  
        return minimum(droit(noeud))  
    while pere(noeud) != None and est_fils_droit(noeud) :  
        noeud = pere(noeud)  
    # soit pere(noeud) == None : noeud est la racine, le noeud  
    # initial était le maximum, et n'a pas de successeur  
    # soit noeud est un fils gauche : le successeur est son père  
    return pere(noeud)
```

Théorème

successeur(noeud) détermine le successeur d'un noeud d'un ABR en temps $\Theta(h)$ au pire, où h est la hauteur de l'ABR.

Arbres Binaires de Recherche

IV. Opérations de modification

INSERTION DANS UN ABR

stratégies imaginables pour ajouter un élément x à un ABR :

- ① *insertion à la racine*, comme une insertion en tête de liste chaînée
- ② *insertion aux feuilles*, comme une insertion en queue de liste chaînée

INSERTION DANS UN ABR

stratégies imaginables pour ajouter un élément x à un ABR :

- 1 *insertion à la racine*, comme une insertion en tête de liste chaînée
- 2 *insertion aux feuilles*, comme une insertion en queue de liste chaînée

insertion à la racine : idée plus naturelle, car la racine est le « point d'entrée » de l'arbre ; il faudrait :

- créer une nouvelle racine contenant x ,
- partitionner les éléments de l'ABR selon le « pivot » x
- reconstruire deux sous-ABR avec, respectivement, les petits et les grands éléments

non seulement c'est compliqué, mais en plus, il faudrait le faire *efficacement* – c'est-à-dire sans parcourir tout l'arbre

INSERTION DANS UN ABR

stratégies imaginables pour ajouter un élément x à un ABR :

- 1 ~~*insertion à la racine*~~, comme une insertion en tête de liste chaînée
- 2 *insertion aux feuilles*, comme une insertion en queue de liste chaînée

insertion à la racine : idée plus naturelle, car la racine est le « point d'entrée » de l'arbre ; il faudrait :

- créer une nouvelle racine contenant x ,
- partitionner les éléments de l'ABR selon le « pivot » x
- reconstruire deux sous-ABR avec, respectivement, les petits et les grands éléments

non seulement c'est compliqué, mais en plus, il faudrait le faire *efficacement* – c'est-à-dire sans parcourir tout l'arbre

c'est peine perdue... *moralité* : essayons autrement !

INSERTION DANS UN ABR

stratégies imaginables pour ajouter un élément x à un ABR :

- 1 ~~*insertion à la racine*~~, comme une insertion en tête de liste chaînée
- 2 *insertion aux feuilles*, comme une insertion en queue de liste chaînée

insertion aux feuilles : créer une feuille à un emplacement libre, c'est-à-dire l'attacher à un nœud non complet

INSERTION DANS UN ABR

stratégies imaginables pour ajouter un élément x à un ABR :

- ① ~~*insertion à la racine*~~, comme une insertion en tête de liste chaînée
- ② *insertion aux feuilles*, comme une insertion en queue de liste chaînée

insertion aux feuilles : créer une feuille à un emplacement libre, c'est-à-dire l'attacher à un nœud non complet

contrairement aux listes chaînées, **non-unicité des emplacements libres**

- point négatif, il faut trouver la bonne feuille,
- point positif, cela donne de la souplesse : insérer une feuille revient, en terme de liste, à insérer un élément à n'importe quelle position, **sans effectuer de réorganisation majeure de la structure**

INSERTION DANS UN ABR

stratégies imaginables pour ajouter un élément x à un ABR :

- 1 ~~*insertion à la racine*~~, comme une insertion en tête de liste chaînée
- 2 *insertion aux feuilles*, comme une insertion en queue de liste chaînée

insertion aux feuilles : créer une feuille à un emplacement libre, c'est-à-dire l'attacher à un nœud non complet

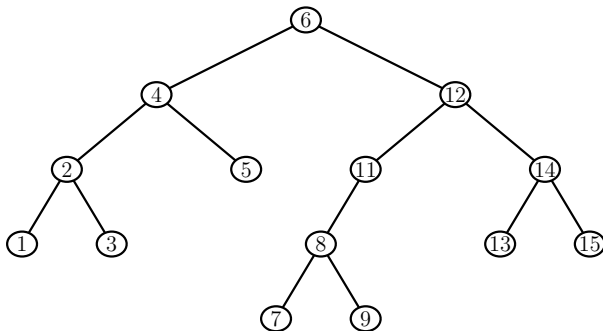
contrairement aux listes chaînées, *non-unicité des emplacements libres*

- point négatif, il faut trouver la bonne feuille,
- point positif, cela donne de la souplesse : insérer une feuille revient, en terme de liste, à insérer un élément à n'importe quelle position, *sans effectuer de réorganisation majeure de la structure*

autre point positif : cela permet d'éviter facilement les doublons

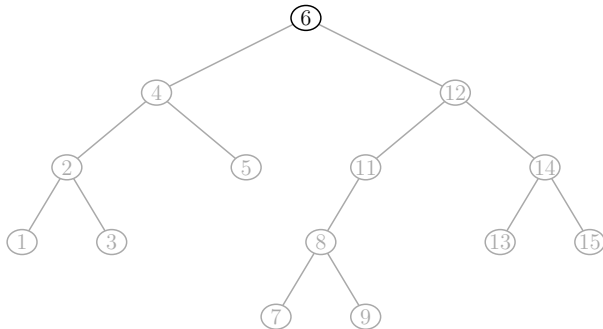
INSERTION DANS UN ABR

Exemple – insertion de 10 :



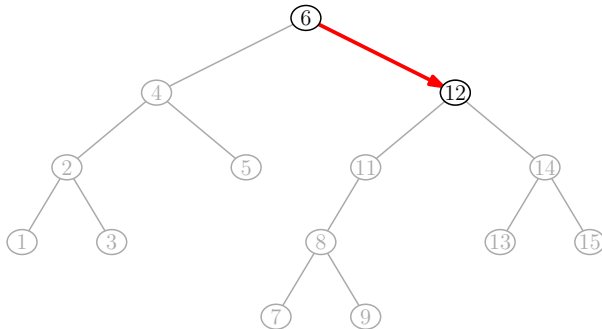
INSERTION DANS UN ABR

Exemple – insertion de 10 :



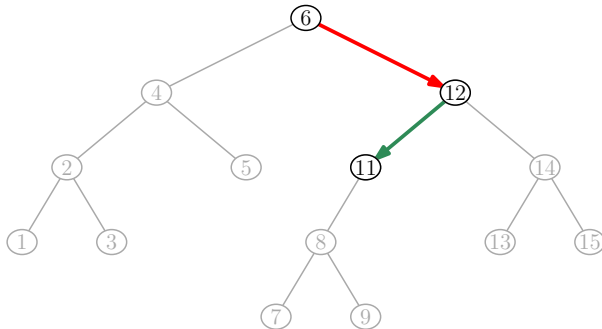
INSERTION DANS UN ABR

Exemple – insertion de 10 :



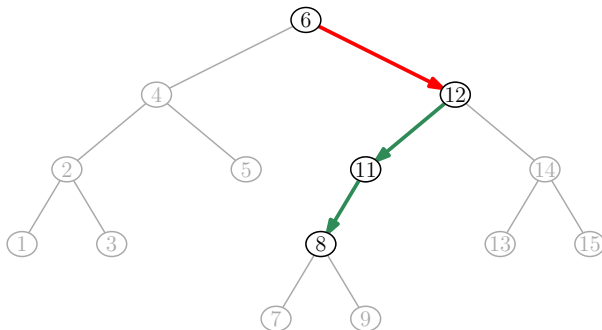
INSERTION DANS UN ABR

Exemple – insertion de 10 :



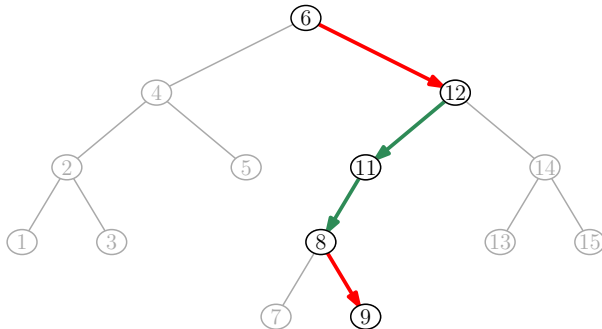
INSERTION DANS UN ABR

Exemple – insertion de 10 :



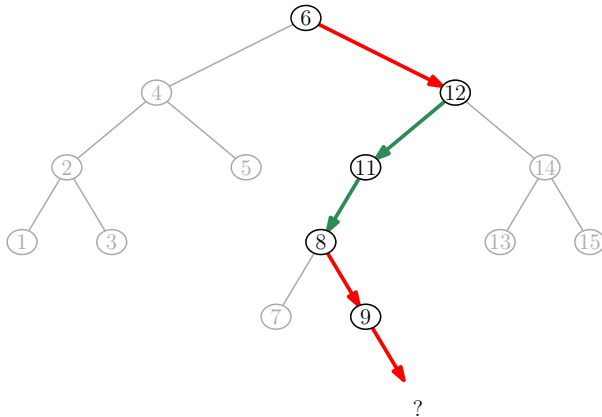
INSERTION DANS UN ABR

Exemple – insertion de 10 :



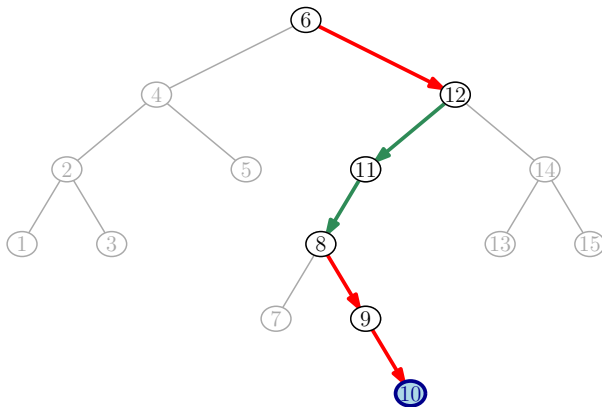
INSERTION DANS UN ABR

Exemple – insertion de 10 :



INSERTION DANS UN ABR

Exemple – insertion de 10 :



INSERTION DANS UN ABR

l'algorithme d'insertion n'est finalement qu'une petite modification de l'algorithme de recherche, donc :

Théorème

L'insertion d'un nouvel élément dans un ABR de hauteur h peut se faire en temps $\Theta(h)$ au pire.