Algorithmique des arbres

Arbres binaires - Arbres binaires de recherche

L2 Mathématique et Informatique

Outline

Rappels : Définitions

Définition : **Arbres**

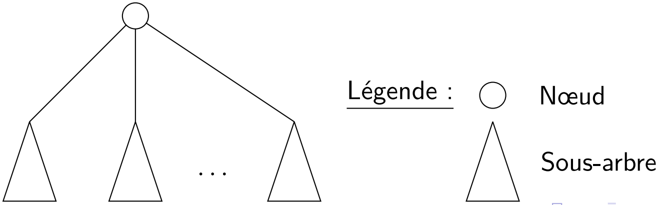
Un *arbre* est défini par :

- un ensemble N de *nœuds*
- un nœud particulier *r* : la *racine*
- P une relation binaire “est parent de”

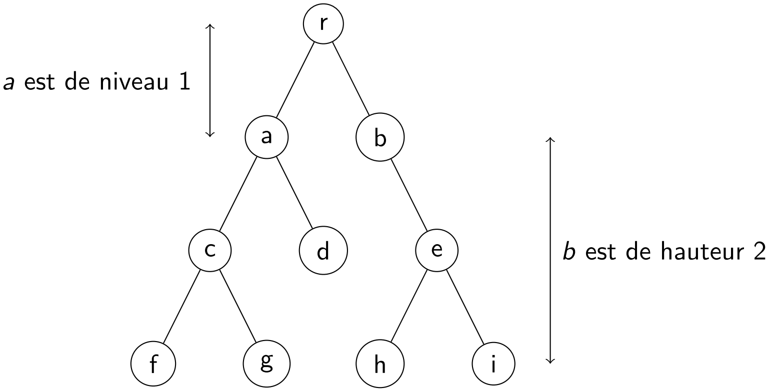
Définition récursive d'un arbre

Un arbre est :

- soit vide (noté Λ)
- soit une racine parent des racines d'arbres disjoints.



Rappels : Définitions



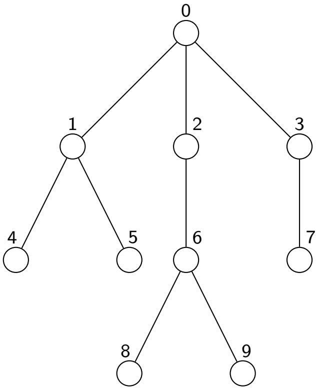
- La racine d'un arbre n'a pas de parent.
- Tout nœud d'un arbre, différent de sa racine, a exactement un parent.

Rappels : Implantation par table des pères

Rappel : L'indice i de la table des pères contient le numéro du père de i

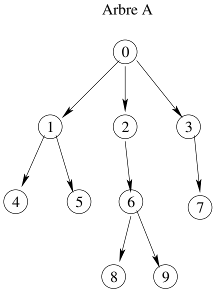
Table des pères :

$-1($ ou $0)$	0	0	0	1	1	2	3	6	6

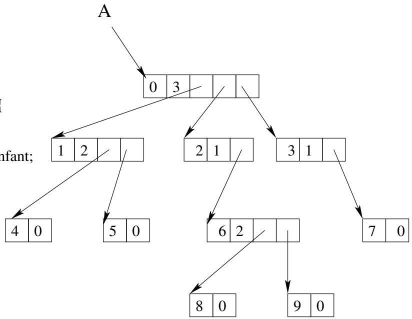


Rappels : Implantation par pointeurs 1

- liens du nœud parent vers chaque enfant



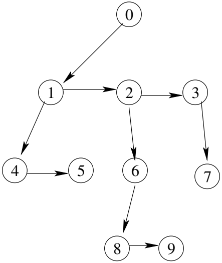

```
typedef struct noeud {
        Element valeur;
        int nbenfant
        struct noeud **enfant;
} Noeud,*Arbre
```



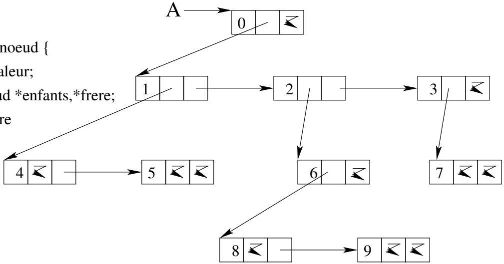
<u>**Remarque :**</u> On a identifié un arbre à l'adresse de sa racine.

Rappels : Implantation par pointeurs 2

- lien du nœud parent vers liste des enfants

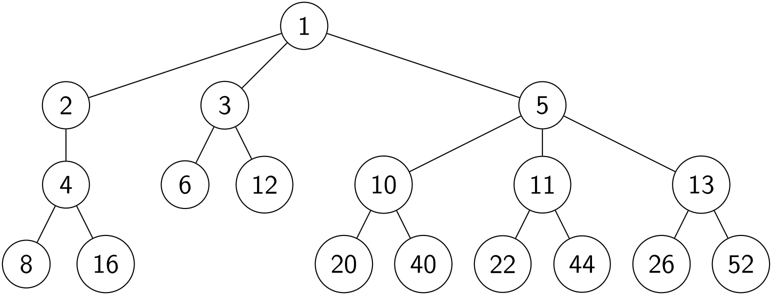



```
typedef struct noeud {
    Element valeur;
    struct noeud *enfants,*frere;
} Noeud,*Arbre
```



Remarque : On a identifié un arbre à l'adresse de sa racine.

Rappels : Parcours d'un arbre



- Parcours largeur :
  ⇝ 1 2 3 5 4 6 12 10 11 13 8 16 20 40 22 44 26 52
- Parcours profondeur préfixe :
  ⇝ 1 2 4 8 16 3 6 12 5 10 20 40 11 22 44 13 26 52
- Parcours profondeur infixe :
  ⇝ 8 4 16 2 1 6 3 12 20 10 40 5 22 11 44 26 13 52
- Parcours profondeur suffixe :
  ⇝ 8 16 4 2 6 12 3 20 40 10 22 44 11 26 52 13 5 1

Outline

Outline

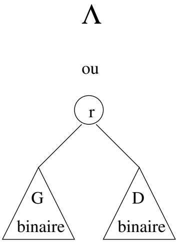
Définitions et exemples

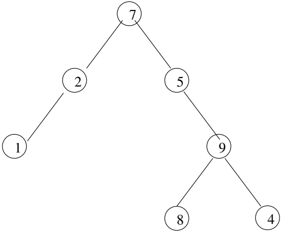
Définition : Arbres binaires

Un arbre a est dit *binaire* lorsque :

$$a = \begin{cases} \Lambda & \text{(arbre vide)} \\ (r, g, d) & \text{où } g \text{ et } d \text{ sont des arbres binaires} \end{cases}$$

Remarque : *Un arbre est binaire lorsque tout nœud de l'arbre possède deux sous-arbres (qui peuvent être vides)*





Définitions et exemples

Définition : Arbres binaires

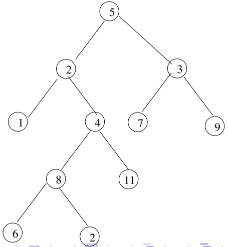
Un arbre a est dit *binaire* lorsque :

$$a = \begin{cases} \Lambda & \text{(arbre vide)} \\ (r, g, d) & \text{où } g \text{ et } d \text{ sont des arbres binaires} \end{cases}$$

Remarque : *Un arbre est binaire lorsque tout nœud de l'arbre possède deux sous-arbres (qui peuvent être vides)*

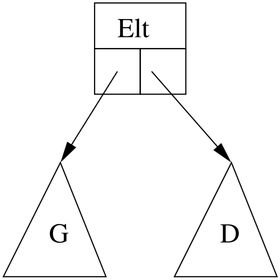
Définition : Arbres strictement binaires

Un arbre binaire est dit *strictement binaire* lorsque tout nœud interne possède deux sous-arbres non vides.



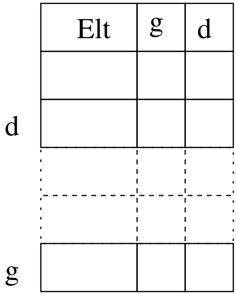
Implantations

- **Par pointeurs :**




```
typedef struct noeud {
  Elt valeur;
  struct noeud *fg,*fd;
} Noeud, *Arbre;
```

- **Par indices :**




```
typedef struct{
  Elt valeur;
  int g,d;
} Noeud;
Noeud memoire[MAXNOEUD];
typedef Noeud * Arbre;
```

Outline

Nombre de nœuds d'un arbre binaire

Propriété :

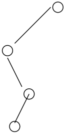
Soit a un arbre binaire quelconque de hauteur h ayant n nœuds.
Alors :

$$h + 1 \leq n \leq 2^{h+1} - 1 \ .$$

$$\log_2(n+1) - 1 \leq h \leq n - 1 \ .$$

Nombre minimum de nœuds :
arbre filiforme

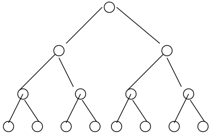
- $\rightsquigarrow$ un seul nœud par niveau.
- $\rightsquigarrow$ 1 nœud au niveau i.



$$n = \sum_{i=0}^{h} 1 = h + 1$$

Nombre maximum de nœuds :
arbre binaire complet

- $\rightsquigarrow$ tous les niveaux sont remplis.
- $\rightsquigarrow$ 2^i nœuds au niveau i.



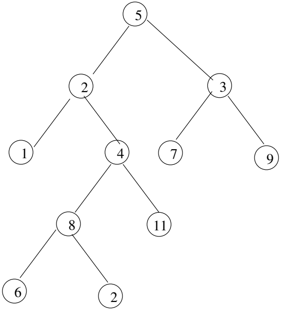
$$n = \sum_{i=0}^{h} 2^i$$

Nombre de nœuds internes d'un arbre strictement binaire 1 / 2

Propriété :

Soit a un arbre strictement binaire non vide.
Le nombre de feuilles de a est égal à son nombre de nœuds internes, augmenté de un :

$$\sharp\text{feuille} = \sharp\text{nœud interne} + 1$$



Nombre de nœuds internes d'un arbre strictement binaire 2 / 2

Démonstration : Par récurrence sur le nb de nœuds internes de $\mathtt{a}$.

- Si l'arbre $\mathtt{a}$ contient 0 nœud interne, $\mathtt{a}$ est réduit à la racine :

  1 feuille et 0 nœud interne.

- Si l'arbre strictement binaire $\mathtt{a}$ contient plus d'un nœud interne :

  il existe un nœud interne $\mathtt{x}$ dont les enfants $\mathtt{g}$ et $\mathtt{d}$ sont des feuilles.

  L'arbre $\mathtt{b}$ obtenu en remplaçant (x, g, d) par une feuille est un arbre strictement binaire, avec :

  $$\begin{cases} \sharp\text{feuille de } \mathtt{b} = \sharp\text{feuille de } \mathtt{a} - 2 + 1 \\ \sharp\text{nœud interne de } \mathtt{b} = \sharp\text{nœud interne de } \mathtt{a} - 1 \end{cases}$$

  L'hypothèse de récurrence s'applique pour l'arbre $\mathtt{b}$:

  $$\sharp\text{feuille de } \mathtt{b} = \sharp\text{nœud interne de } \mathtt{b} + 1$$

  La formule est donc maintenant vraie pourl'arbre $\mathtt{a}$.

Outline

Outline

Arbres binaires de recherche : Définition

Définition

Soit E un ensemble ordonné par une relation d'ordre $<$.
Soit aussi A un arbre binaire ayant des nœuds étiquettés par des éléments de E.

A est appelé un *arbre binaire de recherche* (ABR en abrégé) lorsque :

- A est vide

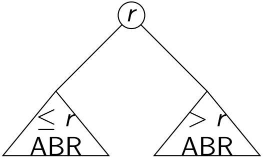
ou

- $A = (r, G, D)$ avec G et D des arbres binaires de recherche tels que :

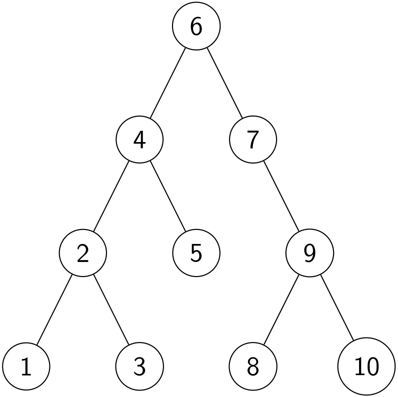
$$\text{Elements}(G) \leq \text{Element}(r) < \text{Elements}(D)$$

Ici :

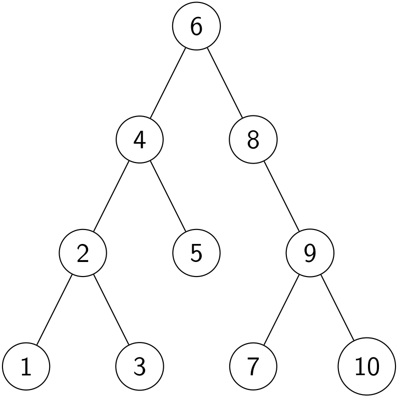
Elements(A) $=$ {etiquettes présentes dans l'arbre A }
Element(nœud) $=$ etiquette du nœud



Exemple et contre-exemple 1



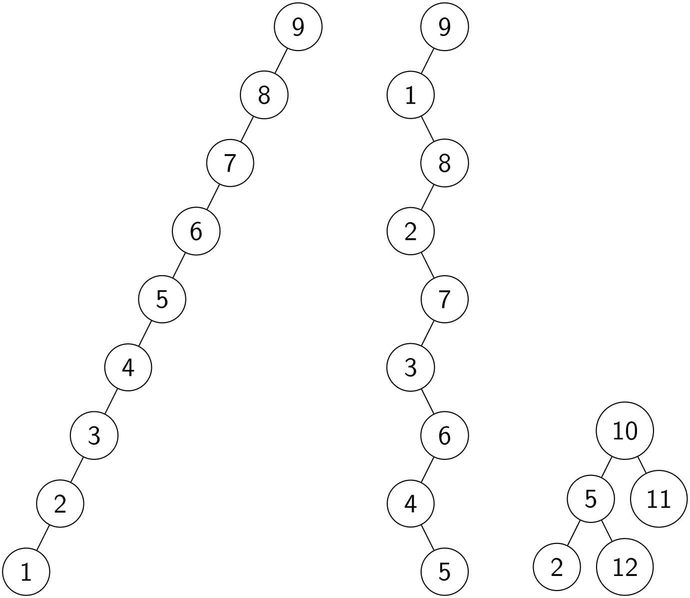
⇝ ABR



⇝ n'est pas un ABR

7 est dans le sous arbre de droite de 8

Exemple et contre-exemple 2



Caractérisations 1

Propriétés :

Soit A est un arbre binaire.
Alors, A est un arbre binaire de recherche si, et seulement si : pour tout nœud p de A, on a :

$$\text{Element}(\text{Gauche}(\text{p})) < \text{Element}(\text{p}) < \text{Element}(\text{Droite}(\text{p}))$$

où Gauche(p) et Droite(p) désigne respectivement le sous-arbre gauche et droit issus du nœud p.

Preuve :

Caractérisations 2

Propriétés :

Soit A est un arbre binaire.
A est un arbre binaire de recherche si, et seulement si le parcours en profondeur, avec affichage en ordre infixe, fournit la liste croissante des éléments de A.

Preuve :

Outline

Opérations sur les ABR

- `creer_ABR_vide();`
- `rechercher(A, x);`
- `ajout(A, x);`
- `supprimer(A, x);`
- `extraire_max(A);`
- `extraire_min(A);`

$\rightsquigarrow$ Compléxité en $\mathcal{O}\big(Hauteur(A)\big)$.

Compléxité et compléxité en moyenne

Rappels : Si a un arbre binaire quelconque de hauteur h ayant n nœuds :

$$\log_2(n+1) - 1 \leq h \leq n - 1 \ .$$

Dans le cas le plus défavorable :

- pour un arbre binaire complet à n nœuds : $\mathcal{O}\big(\ln(n)\big)$.
- pour un arbre réduit à une chaîne linéaire de n nœuds : $\mathcal{O}\big(n\big)$

Mais...

Propriété :

La hauteur attendue d'un arbre binaire de recherche construit aléatoirement est $\mathcal{O}\big(\ln(n)\big)$.

Donc, ces opérations seront en moyenne en $\mathcal{O}\big(\ln(n)\big)$.

Grammaire 1/2

- **Lorsqu'il n'y a pas besoin de modifier la racine**

```
...  operation(Arbre a, ...){
    if (a == NULL) / Si l'arbre est vide */
         ...
    /* traitement récursif sur le fils gauche */
    ...  operation(a->fg, ...) ... ;
    /* traitement récursif sur le fils droit */
    ...  operation(a->fd, ...) ... ;
    /* fin de l'operation */
    ...
}
```

Rappel : Un arbre *a* est un pointeur sur une structure `Noeud`.

Donc, on accède aux sous-arbre gauche et droit par :

`a->fg` et `a->fd`

- **Lorsqu'il y a besoin de modifier la racine :**

```
... operation(Arbre *a, ...){
    if (*a == NULL) /* Si l'arbre sous-jacent est vide */
         ...
    /* traitement récursif sur le fils gauche */
    ...  operation(&((*a)->fg), ...) ... ;
    /* traitement récursif sur le fils droit */
    ...  operation(&((*a)->fd), ...) ... ;
    /* fin de l'operation */
    ...
}
```

Rappel : Un arbre *a* est un pointeur sur une structure `Noeud`.

- `*a` est un `Arbre` ;
- on accède aux sous-arbre gauche et droit par :
  `(*a)->fg` et `(*a)->fd`
- on prends l'adresse de ces sous-arbres pour les appels récursifs :
  `&((*a)->fg)` et `&((*a)->fd)`.

Outline

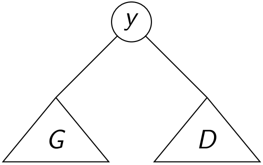
Opération rechercher 1 / 3

Description :

`rechercher(A,x)` vaut vrai si, et seulement si, x est l'étiquette d'un nœud de A.

- Pour un arbre binaire A vide, $\texttt{rechercher}(A, x) = \texttt{faux}$.
- Pour un arbre binaire de recherche $A = (y, G, D)$ non vide :

$$\texttt{rechercher}(A, x) = \begin{cases} \text{vrai, si } x = y \\ = \texttt{rechercher}(G, x) \text{ si } x \leq y \\ = \texttt{rechercher}(D, x) \text{ si } x > y \end{cases}$$



$\rightsquigarrow$ Recherche d'un élément en $\mathcal{O}(\text{Hauteur(A)})$

Opération rechercher 2 / 3

```
typedef struct noeud{
    Elt valeur;
    struct noeud *fg,*fd;
} Noeud, *Arbre;


/* On renvoie l'adresse du Noeud contenant x, ou NULL */
Noeud * rechercher_rec(Arbre A, Elt x){
    if (A == NULL)
        return NULL;
    if (A->valeur == x)
        return A;
    if (x <A->valeur)
        return rechercher_rec(A->fg, x);
    return  rechercher_rec(A->fd, x);
}
```

Opération rechercher 3 / 3

```
typedef struct noeud{
    Elt valeur;
    struct noeud *fg,*fd;
} Noeud, *Arbre;
```

```
/* On renvoie l'adresse du Noeud contenant x, ou NULL */
Noeud * rechercher_iter(Arbre A, Elt x){
    /* on descend le long d'une branche de l'arbre */
    while (A != NULL && A->valeur != x)
        if (x < A->valeur)
            A = A->fg;
        else
            A = A->fd;
    return A;
}
```

Outline

Ajout d'un nouvel élément 1 / 2

Description :

On descend le long de la branche qui doit contenir l'élément.
S'il n'est pas présent, on crée une nouvelle feuille.

- Pour un arbre binaire vide, $A + \{x\} = (x, \Lambda, \Lambda)$
- Pour un arbre binaire non vide $A = (r, G, D)$:

$$A + \{x\} = \begin{cases} A & \text{si } x = \text{Element}(r) \\ (r, G + \{x\}, D) & \text{si } x < \text{Element}(r) \\ (r, G, D + \{x\}) & \text{si } x > \text{Element}(r) \end{cases}$$

$\rightsquigarrow$ Ajout d'un élément en $\mathcal{O}(\text{Hauteur(A)})$

Ajout d'un nouvel élément 2 / 2

```
Noeud * ajout(Arbre *A, Element x){
    /* Renvoie l'adresse du noeud contenant x */
    if (*A == NULL) {
        *A = alloue_noeud(x);
        return *A;
    }
    if (x == (*A)->valeur)
        return *A; /* x est a la racine */
    if (x < (*A)->valeur)
        return ajout(&((*A)->fg), x);
    else /* x > (*A)->valeur */
        return ajout(&((*A)->fd), x);
}
```

Outline

Extraction du Max 1 / 3

Description :

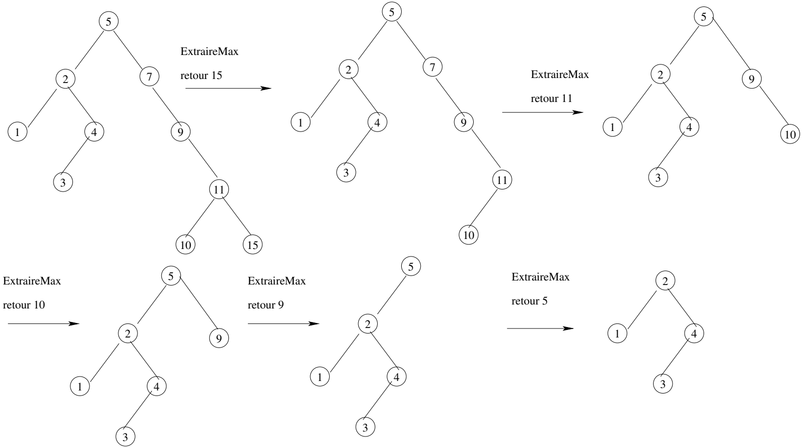
Extraire l'élément maximum d'un arbre binaire de recherche, c'est extraire son élément le plus à droite.

- Pour un arbre binaire vide, $A - \{max\} = A$
- Pour un arbre binaire non vide $A = (r, G, D)$:

$$A - \{max\} = \begin{cases} G & \text{si } D \text{ est vide} \\ (r, G, D - \{max\}) & \text{si } D \text{ est non vide} \end{cases}$$

$\rightsquigarrow$ Extraction du maximum en $\mathcal{O}(\text{Hauteur(A)})$

Extraction du Max 2 / 3



Extraction du Max 3 / 3

Warning ! La racine de l'arbre peut changer...

```
Noeud * extrait_max(Arbre *A){
    /* Renvoie l'adresse du noeud contenant l'etiquette
    maximale */
    Noeud * tmp;
    if (*A == NULL)
        return *A;
    if ((*A)->fd == NULL) {
        tmp = *A;
        *A = (*A)->fg;
        return tmp;
    }
    return extrait_max(&((*A)->fd));
}
```

Outline

Suppression d'un élément 1 / 6

Description :

Suppression d'un élément x dans un arbre binaire de recherche :

- S'il n'existe pas de nœud contenant x, il n'y a rien à faire.
- Sinon, se placer sur le sous arbre de racine x :
  - 0 fils : suppression simple
  - 1 fils : remplacement par ce fils
  - 2 fils : remplacement par le plus grand élément du fils gauche ou le plus petit du fils droit

Suppression d'un élément 2 / 6

- Pour un arbre binaire vide, $A - \{x\} = A$

- Pour un arbre binaire non vide $A = (r, G, D)$, avec $Element(r) \neq x$:

$$A - \{x\} = \begin{cases} (r, G - \{x\}, D) & \text{si} \quad x < \text{Element}(r) \\ (r, G, D - \{x\}) & \text{si} \quad x > \text{Element}(r) \end{cases}$$

- Pour un arbre binaire non vide $A = (x, G, D)$:
  - si x n'a pas d'enfant $A - \{x\} = \Lambda$
  - si x a un seul enfant :

$$A - \{x\} = \begin{cases} D & \text{si} \quad A = (r, \Lambda, D) \\ G & \text{si} \quad A = (r, G, \Lambda) \end{cases}$$

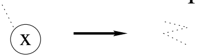
  - si x a deux enfants :

$$A - \{x\} = \begin{array}{l} (Min(D), G, D - Min(D)) \\ ou \\ (Max(G), G - Max(G), D) \end{array}$$

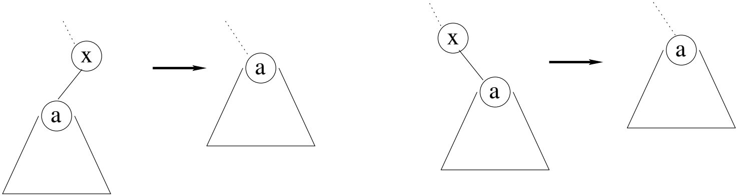
$\rightsquigarrow$ Extraction du maximum en $\mathcal{O}(\text{Hauteur}(A))$

Suppression d'un élément 3 / 6

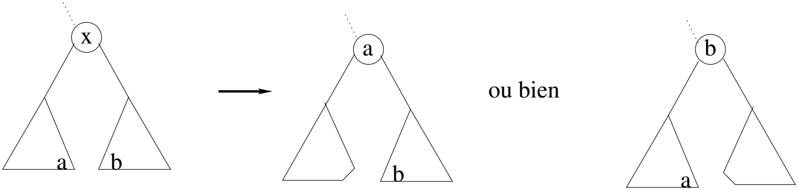
Feuille



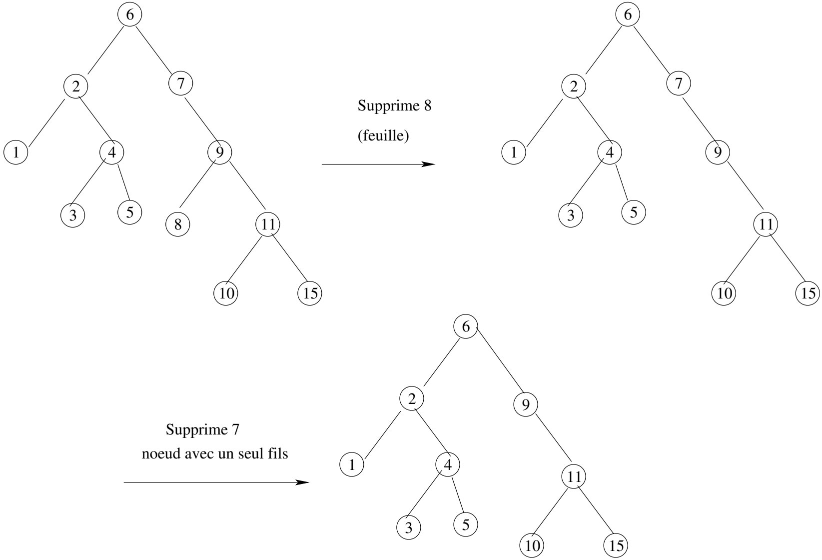
un fils



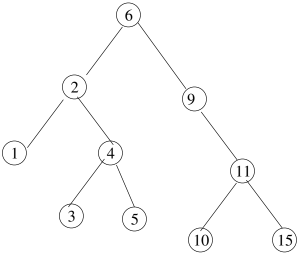
deux fils



Suppression d'un élément 4 / 6

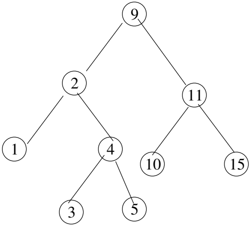


Suppression d'un élément 5 / 6

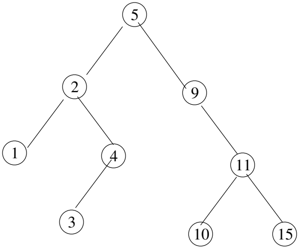


Supprime 6
noeud avec deux fils

Plus petit du sous arbre droit



Plus grand du sous arbre gauche



Suppression d'un élément 6 / 7 – 6

```
Noeud * suppression(Arbre * A, int x) {
    Noeud * tmp, * max;
    if (*A == NULL)
        return *A;
    if ((*A)->valeur > x)
        return suppression(&((*A)->fg), x);
    if ((*A)->valeur < x)
        return suppression(&((*A)->fd), x);
    tmp = *A; /* Désormais, la racine vaut x */
    if ((*A)->fg == NULL && (*A)->fd == NULL) /* Pas d'enfants *
        *A = NULL;
    else if ((*A)->fg == NULL) /* 1 enfant à droite */
             *A = (*A)->fd;
          else if ((*A)->fd == NULL) /* 1 enfant à gauche */
                  *A = (*A)->fg;
               else {  /* 2 enfants => on remonte le max */
                   max = extrait_max(&((*A)->fg));
                   (*A)->valeur = max->valeur;
               }
   return tmp;
}
```