Arbres binaires de recherche (I)

- Version sans équilibrage -

Irena.Rusu@univ-nantes.fr

LINA, bureau 123, 02.51.12.58.16

Temps d'exécution

opérations sur ensemble

implémentation	Ens_vide	Ajouter Enlever	Élément	Min
table	cst	$O(1)$*	$O(n)$	$O(n)$
table triée	cst	$O(n)$	$O(\log n)$	$O(1)$
liste chaînée	cst	$O(1)$*	$O(n)$	$O(n)$
arbre équilibré	cst	$O(\log n)$	$O(\log n)$	$O(\log n)$
arbre	cst	$O(\log n)$	$O(\log n)$	$O(\log n)$
table de hachage	$O(B)$	cst	cst	$O(B)$

en moyenne

n nombre d 'éléments
$B > n$ taille de la table de hachage
**sans le test d 'appartenance*

Sommaire

- Arbres binaires de recherche (ou ABR)
  - Recherche d’un élément
  - Ajout d’un élément
  - Suppression d’un élément
  - Tri d’une liste

Sommaire

- Arbres binaires de recherche (ou ABR)
  - Recherche d’un élément
  - Ajout d’un élément
  - Suppression d’un élément
  - Tri d’une liste

Arbre binaire de recherche

A arbre binaire

noeuds étiquetés par des éléments

A est un arbre binaire de recherche (ou abr ou ABR)

Déf 1 ssi le parcours symétrique donne une liste croissante des éléments

Déf 2 ssi pour tout sous-arbre *B* de *A* et pour tous les éléments *g* de *G(B)* et *d* de *D(B)* la condition suivante est vérifiée

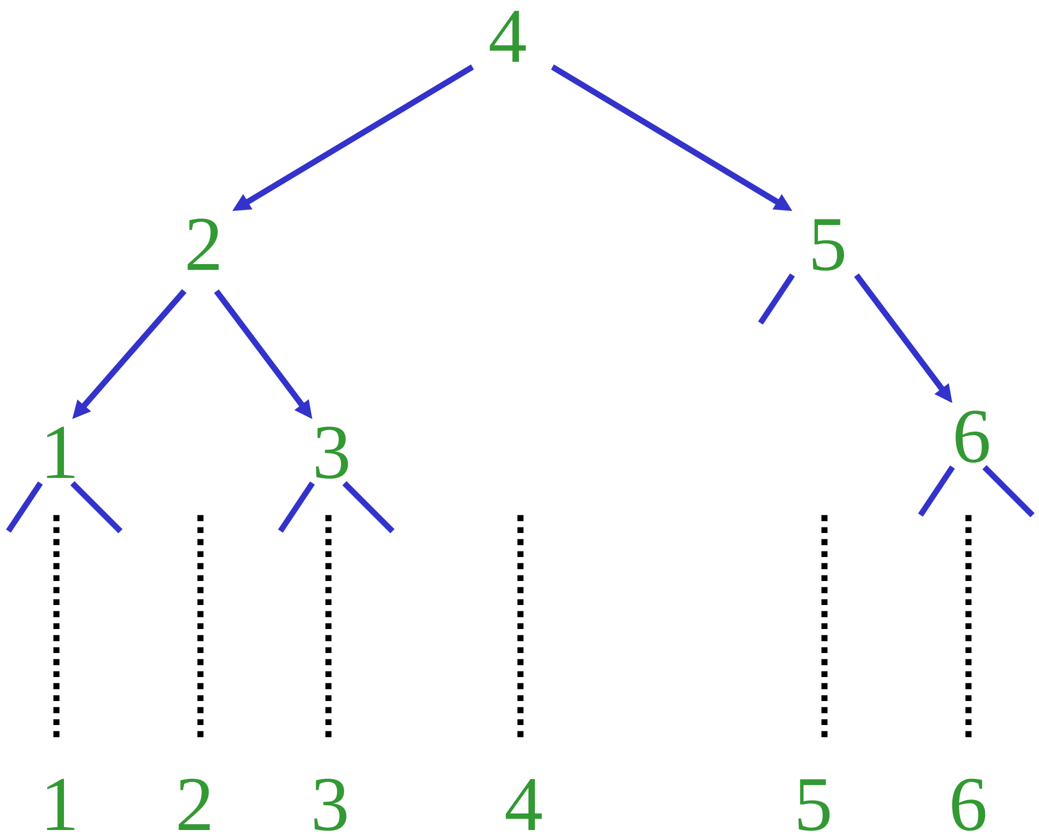
$$g \leq \mathrm{Elt}(B) < \mathrm{d}$$

Déf 3 ssi $A = \text{Arbre\_vide}$ ou

$A = (r, G, D)$ avec

- *G, D* arbres binaires de recherche et
- g ≤ *r* < d pour tout élément g de G et tout élément d de D

Exemple



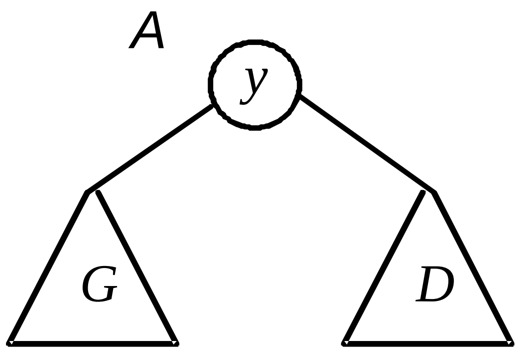
Sommaire

- Arbres binaires de recherche (ou ABR)
  - Recherche d’un élément
  - Ajout d’un élément
  - Suppression d’un élément
  - Tri d’une liste

Recherche d'un élément (1)

Elément (A, x) = vrai ssi x est étiquette d'un noeud de A (abr)

Place(A,x) pointe sur le nœud où x est placé, ou retourne NULL.



Élément (A, x) =

faux	si A vide
vrai	si $x = y$
Elément (G(A), x)	si $x < y$
Elément (D(A), x)	si $x > y$

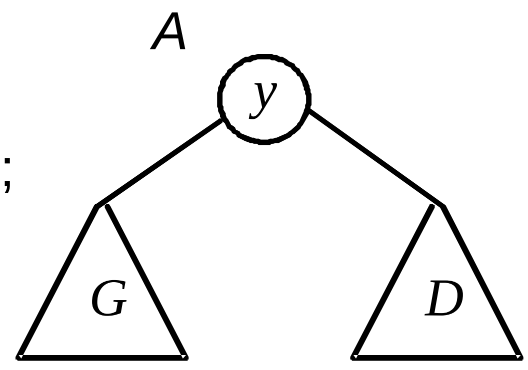
Calcul en O(Hauteur(A))

Recherche d'un élément (2)

Elément (A, *x*) = vrai ssi *x* est étiquette d'un noeud de A (abr)

Place(A,x) pointe sur le nœud où x est placé, ou retourne NULL

abr : le type ABR.



fonction Element (A abr, *x* élément) : booleen ;

début

si *A=vide* **alors** retour (faux)

si *x=Elt(A)* **alors** retour (vrai)

si *(x<y)* **alors** retour (Element(G(A),x))

sinon retour(Element(D(A),x)

fin

Complexité de Element(.,.)

fonction Element (A abr, *x* élément) : booleen ;

début

  si *A=vide* **alors** retour (faux)

  si *x=Elt(A)* **alors** retour (vrai)

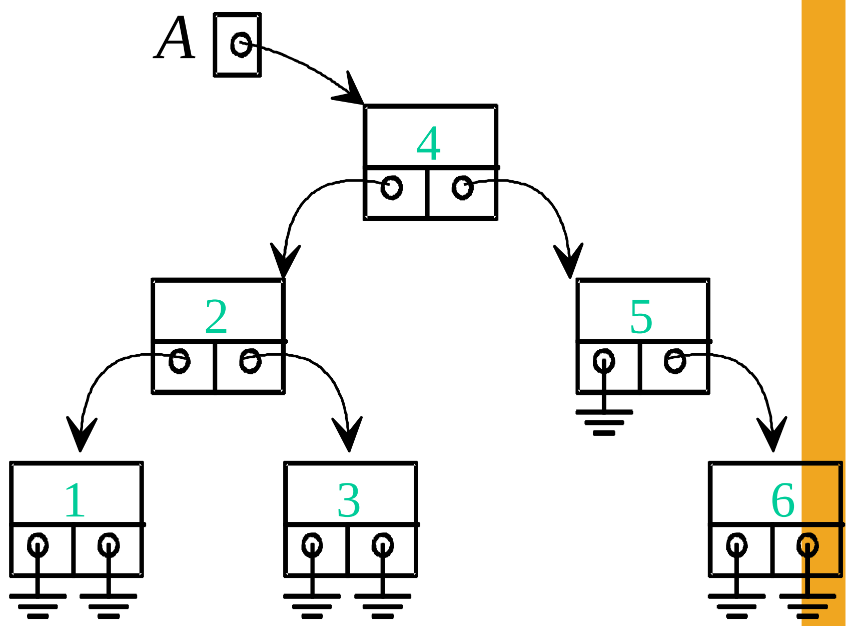
  si *(x<y)* **alors** retour (Element(G(A),x))

      sinon retour(Element(D(A),x)

fin

NbOper(A)=

- 2, si A= vide
- 3, si x est à la racine
- 3 + NbOper(G(A)), OU
- 3 +NbOper(D(A)) sinon



Complexité de Element(.,.)

NbOper(A)=

2, si A= vide

3, si x est à la racine

3 + NbOper(G(A)), OU

3 +NbOper(D(A)) sinon

$\}$ ≤ 3 + max (NbOper(G(A)), NbOper(G(B))

La **descente** d'un niveau – passage de A à G(A) ou D(A)) - **coûte 3**.

L'**arrêt** coûte **2** ou 3.

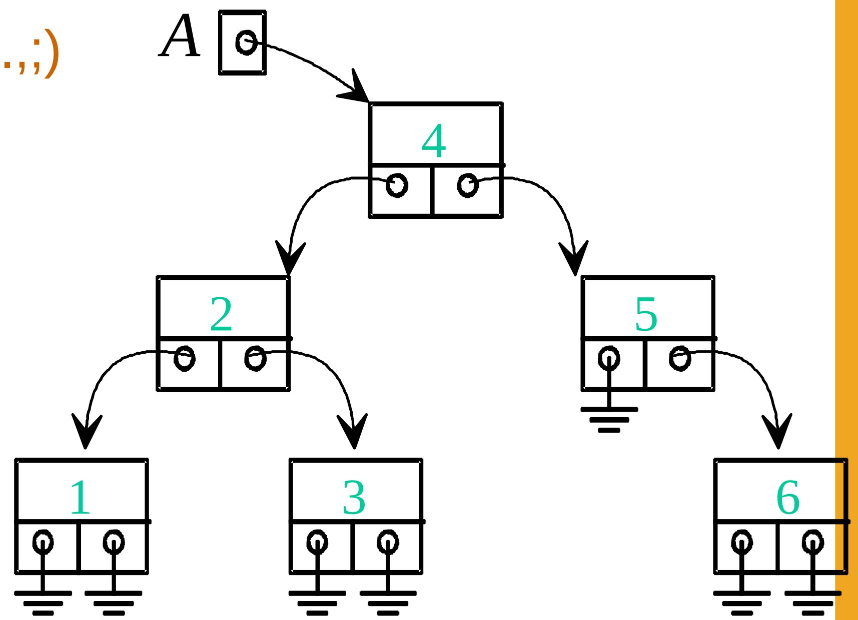
Au pire, on descend jusqu'au tout dernier niveau, + 1 tentative

→ NbOper(A) ≤ 3 x (h(A)+1) +2 (si tentative, on finit sur A=vide)

NbOper(A)=O(h(A)), où h(A) est la hauteur (ou profondeur) de A

Version itérative

pour Place (.,;)



fonction Place (A abr, x élément) : abr ;

début

  $B \leftarrow A$;

  tant que $B \neq$ nil **et** $x \neq$ Elt(B) **faire**

    si $x <$ Elt (B) **alors** $B \leftarrow G(B)$;

              sinon $B \leftarrow D(B)$;

  retour (B) ;

fin

Complexité de Place (.,.)

fonction Place (A abr, x élément) : abr ;

début
    $B \leftarrow A$;
    tant que $B \neq$ nil **et** $x \neq$ Elt(B) **faire**
        si $x <$ Elt (B) **alors** $B \leftarrow G(B)$;
        sinon $B \leftarrow D(B)$;
    retour (B) ;
fin

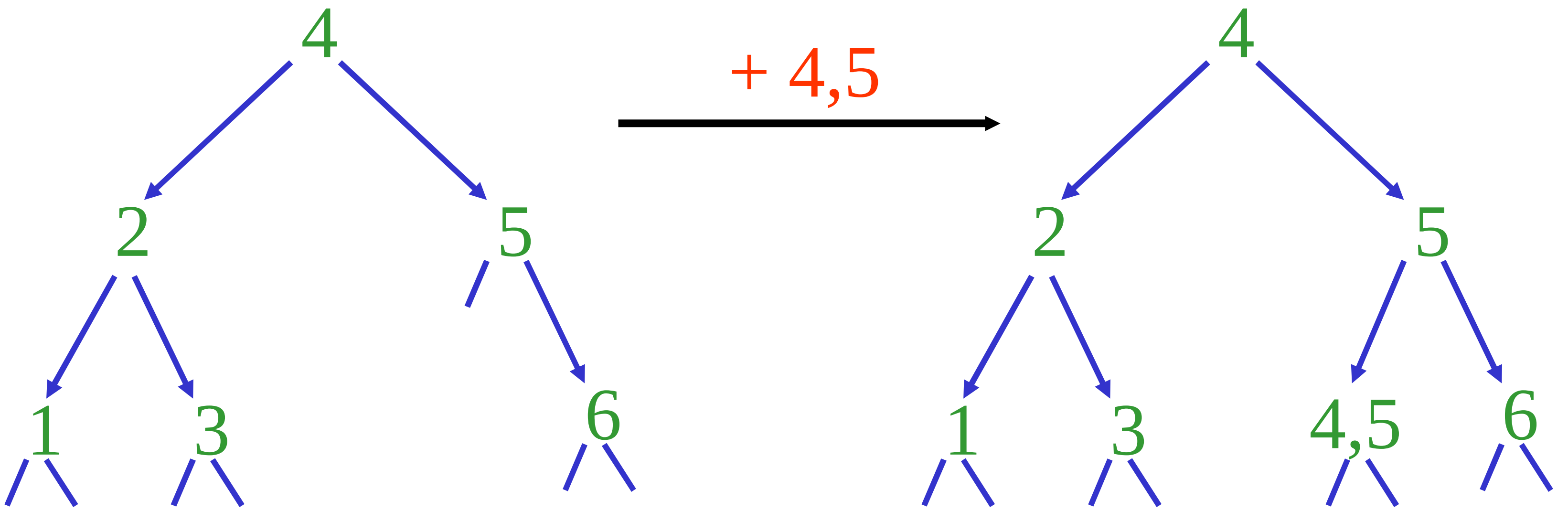
NbOper(A)≤1+2*h(A)*2+1 =O(h(A))

Remarque. Ne pas en déduire qu'Element(.,.) est plus rapide que Place à cause des constantes … la récursivité cache quelques opérations..

Sommaire

- Arbres binaires de recherche (ou ABR)
  - Recherche d'un élément
  - Ajout d'un élément
  - Suppression d'un élément
  - Tri d'une liste

Ajout d'un élément*



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

*** s'il n'est pas déjà dans l'arbre**

Version récursive

fonction Ajouter (A abr, *x* élément) : abr ;
début

si Vide (A) **alors**

retour (Cons(x, vide, vide)) **;**

sinon si $x <$ Elt (A) **alors**

retour (Cons(Elt(A), Ajouter(G(A), *x*), D(A)))

sinon si $x >$ Elt (A) **alors**

retour (Cons(Elt(A), G(A), Ajouter (D(A), *x*)))

sinon retour (A)

fin

Complexité de Ajouter(.,.)

fonction Ajouter (A abr, *x* élément) : abr ;
début

si Vide (A) **alors**

retour (Cons(x, vide, vide)) **;**

sinon si $x <$ Elt (A) **alors**

retour (Cons(Elt(A), Ajouter(G(A), *x*), D(A)))

sinon si $x >$ Elt (A) **alors**

retour (Cons(Elt(A), G(A), Ajouter (D(A), *x*)))

sinon **retour** (A)

fin

NbOper(A)=

O(1) +

O(1)+ NbOper(G(A)) +

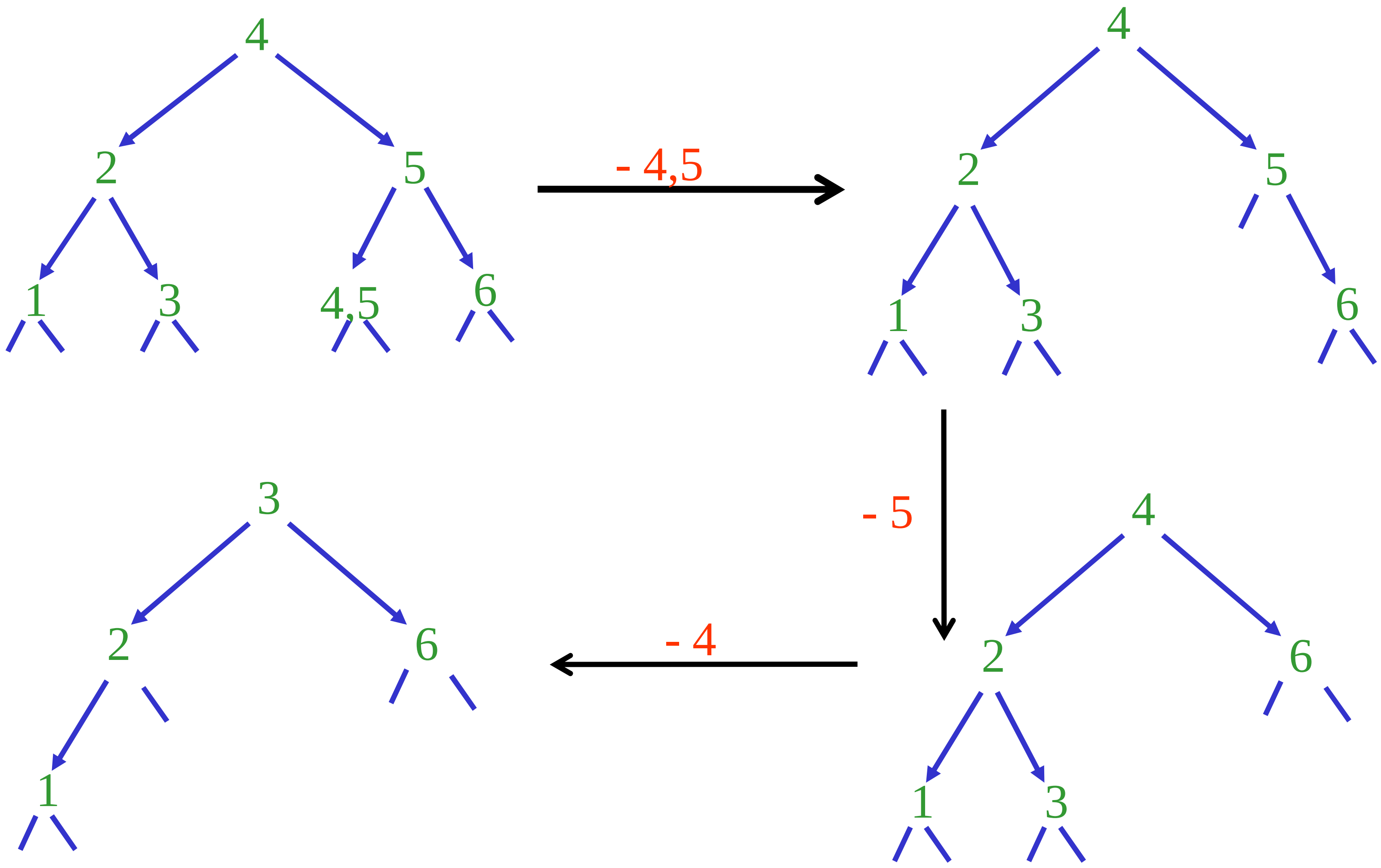
O(1)+NbOper(D(A))

NbOper(A)=O(1)+O(h(A))=O(h(A))

Sommaire

- Arbres binaires de recherche (ou ABR)
  - Recherche d’un élément
  - Ajout d’un élément
  - Suppression d’un élément
  - Tri d’une liste

Suppression d’un élément



Fonction Enlever

$A = (r, G, D)$

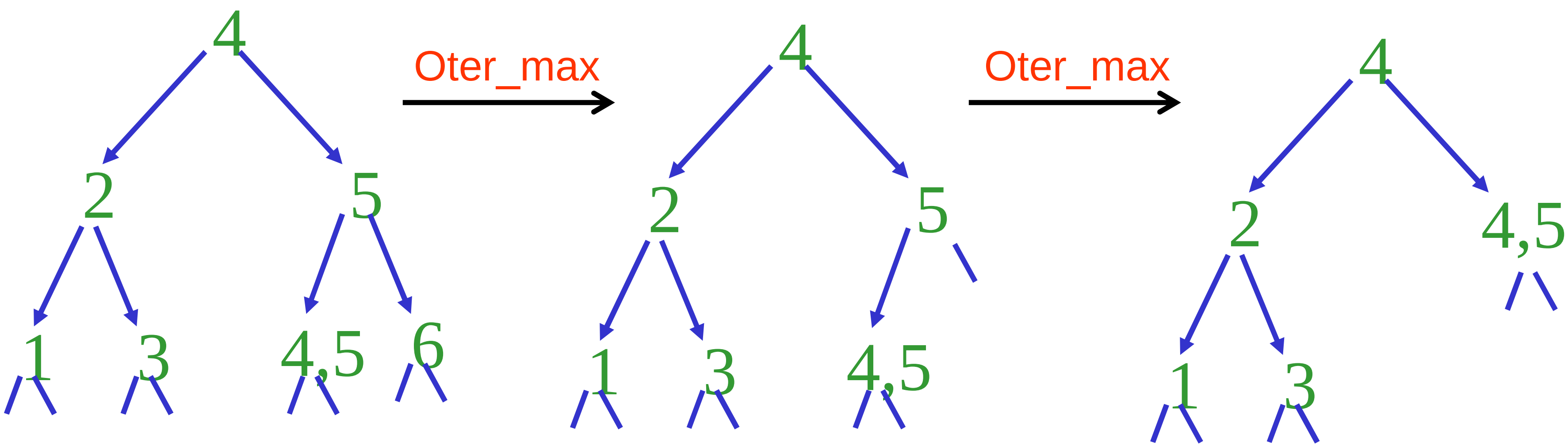
$$A - \{x\} = \begin{cases} (r, G - \{x\}, D) & \text{si } x < \text{Elt}(r) \\ (r, G, D - \{x\}) & \text{si } x > \text{Elt}(r) \\ D & \text{si } x = \text{Elt}(r) \text{ et } G \text{ vide} \\ G & \text{si } x = \text{Elt}(r) \text{ et } D \text{ vide} \\ (r, G - \{\text{MAX}(G)\}, D) & \text{sinon} \\ \text{avec Elt}(r) = \text{MAX}(G) & \end{cases}$$

Note. On peut utiliser MIN(D) à la place de MAX(G) de manière similaire.

Version récursive

```
fonction  Enlever (A abr, x élément) : abr ;
début
   si Vide (A) alors
       retour (A)
  sinon si  x < Elt (A) alors
             retour (Cons(Elt(A), Enlever (G(A), x), D(A)))
           sinon si  x >  Elt (A)  alors
                    retour (Cons(Elt(A), G(A), Enlever (D(A),x)) )
                  sinon si  Vide (G(A)) alors
                         retour (D(A))
                        sinon si  Vide (D(A)) alors
                                retour  (G(A))
                              sinon
                                retour (Cons(MAX(G(A)), Oter_max (G(A)), D(A)))
fin
```

Oter_MAX



fonction Oter_max (*A* abr) : abr ;
début

si vide (D (*A*)) **alors**

retour (G (*A*))

sinon

retour (Elt(*A*), G (*A*), Oter_max (D (*A*))) ;

fin

MAX(A) calculé de manière similaire …

Complexité : O(h(A))

Version récursive

fonction Enlever (A abr, x élément) : abr ;
début
si Vide (A) **alors**
retour (A)
sinon si $x <$ Elt (A) **alors**
retour (Cons(Elt(A), Enlever (G(A), x), D(A)))
sinon si $x >$ Elt (A) **alors**
retour (Cons(Elt(A), G(A), Enlever (D(A),x)))
sinon si Vide (G(A)) **alors**
retour (D(A))
sinon si Vide (D(A)) **alors**
retour (G(A))
sinon
retour (Cons(MAX(G(A)), Oter_max (G(A)), D(A)))
fin

Complexité : O(profondeur(x)) pour la recherche de x

O(h(A_x)) pour la recherche du MAX

Total :O(h(A))

Sommaire

- Arbres binaires de recherche (ou ABR)
  - Recherche d’un élément
  - Ajout d’un élément
  - Suppression d’un élément
  - Tri d’une liste

Trier

fonction TRI (*L* liste) : liste ;

début $A \leftarrow$ abr_vide ;

pour *x* $\leftarrow$ premier au dernier élément de *L* **faire**

$A \leftarrow$ Ajouter (A, *x*) ;

retour (Parcours_symétrique (A)) ;

fin.

Temps :

au pire O (|L| ^{2}) avec un ABR, car h(A)=O(|L|)

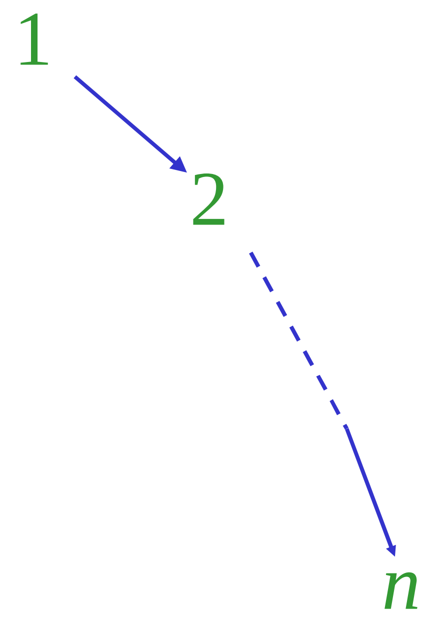
au pire O (|*L*| log |*L*|) avec ABR équilibré (ou AVL),

car h(A)=O(log|L|)

Temps maximal avec un ABR

Insertions successives de 1, 2, ..., *n*

dans l'arbre vide



Nombre de comparaisons d'éléments :

$$0 + 1 + \ldots + n - 1 = \frac{n(n-1)}{2}$$

Temps maximal avec un ABR équilibré(ou AVL)

- Insertions successives en O(h(A)) chacune
- h(A) est maintenue à une valeur en O(log n), où n est le nombre d'éléments dans l'arbre
- On évite l' « allongement » de l'arbre en faisant basculer des sous-arbres entiers de gauche vers la droite ou inversement.