

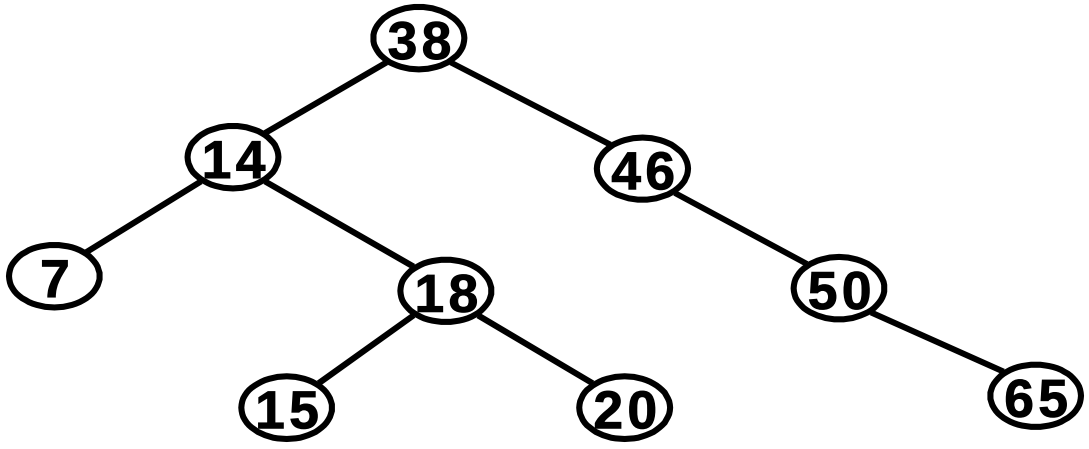
Chapitre 6

Arbres de Recherche



Définition ABR

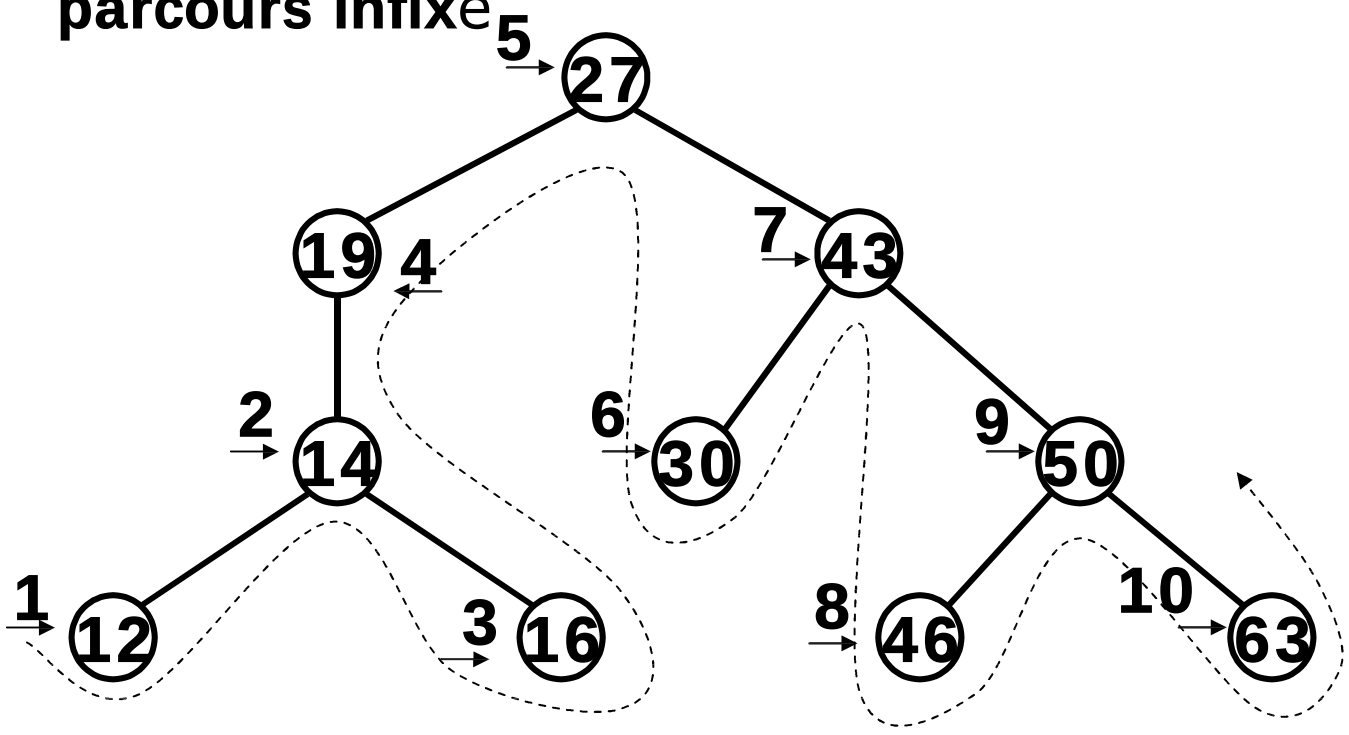
- **Un arbre binaire de recherche (ABR) est un arbre binaire construit sur des éléments qui possèdent une clé d'identification et tel que pour tout nœud sa clé est :**
  - **supérieure à toutes celles de ses descendants de gauche et**
  - **inférieure à toutes celles de ses descendants de droite.**
- **Illustration**



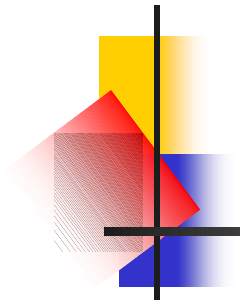
- **Un ABR ne peut donc avoir de valeurs répétées**

Conséquence ...

- **Les propriétés énoncées sur les arbres binaires de recherche permettent de déduire que la liste infixe (GRD) d'un tel arbre correspond à la liste ordonnée par clé croissante**
- **Tous les arbres de recherche contenant les mêmes éléments donnent le même parcours infixé**



- **Parcours infixe : 12, 14, 16, 19, 27, 30, 43, 46, 50, 63.**



Implémentation

- On va utiliser la fonction **elementComparer** qui retourne 0, < 0 ou > 0
- La définition d'un ABR est identique à celle d'un AB
- Avec les ABR on peut insérer et supprimer des éléments contrairement au AB classiques ce qui explique que le TDA ARBINR contient les deux primitives **inserer** et **supprimer**

Représentation chaînée des AB




```
/*****************************************************
 *  Fichier : ARBINR.H
 *  Contenu : Déclaration de la strucure de donnée choisie du TDA
     ARBIN.
*****************************************************/
#ifndef _ARBINR_H
#define _ARBINR_H
#include "ELTPRIM.H"

/* déclaration de structure structNoeud */
typedef struct structNoeud {
  ELEMENT info;              /* information stockée dans le noeud
    */
  struct structNoeud * fg;  /* adresse de fils gauche */
  struct structNoeud * fd;  /* adresse de fils droit */
}  structNoeud, * NOEUD;

/* déclaration de structure laStruct */
typedef struct {
  NOEUD refRacine;  /* racine de l'arbre bianire */
} laStruct, * ARBINR;
#endif
```

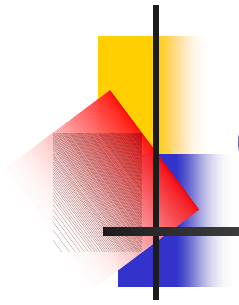
Primitives du TDA ABR

```
/*****************************************************************
 *  Fichier : ARBRPRIM.H
 *  Contenu : Déclaration des primitives du TDA ARBINR
/*****************************************************************/
#ifndef _ARBRPRIM_H
#define _ARBRPRIM_H

#include <stdio.h>
#include <stdlib.h>
#include "ARBRSDD.H"

ARBINR arbinRCreer(void);
void arbinRDetruire(ARBINR);
int arbinRVide(ARBINR);
int arbinRSature(ARBINR);
int arbinRTaille(ARBINR);
int inserer(ARBINR, ELEMENT);
int supprimer(ARBINR, ELEMENT);
int appartient(ARBINR, ELEMENT);
 void arbinRAfficher(ARBINR , int );

#endif
```



Opération de recherche

- **Les ABR permettent de rechercher et de trouver un nœud de façon simple et rapide par rapport à d'autres SD telle que les listes**

- ***Principe :*** **Si la valeur du nœud recherché est inférieure à la valeur du nœud courant alors chercher dans le SAG, sinon chercher dans le SAD**

Implémentation: Appartient

```
int appartient(ARBINR a, ELEMENT e) {

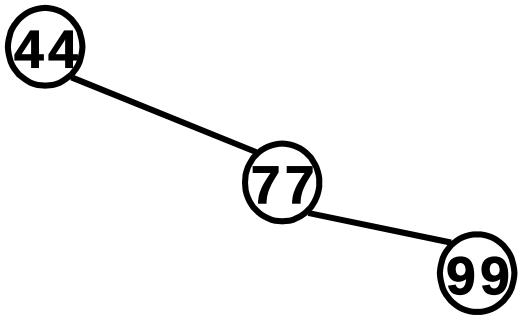
int test=0;
NOEUD p;

if (!arbinRVide(a))
{
p=a->refRacine;

while ((p !=NULL) && (! test))
        {
        if ((elementComparer(e,p->info))<0)
                p=p->fg;
        else
                {
                if ((elementComparer(e,p->info))>0)
                        p=p->fd;
                else
                        test=1;
                }
        }
}
return test;
}
```

Complexité temporelle de la recherche (1)

- **La complexité temporelle de la recherche dépend de deux facteurs:**
- **La taille et la forme de l'arbre (complet – dégénéré etc)**
- **L'emplacement de l'élément à rechercher (au début, au milieu, à la fin)**
- **La Complexité au pire des cas correspond au cas d'un arbre dégénéré où on doit cherche le dernier élément: O(n), exemple chercher 99 dans l'arbre suivant**



Complexité temporelle de la recherche (1)

- **Il est également intéressant de s'intéresser à la complexité au pire avec un arbre complet car le cas dégénéré est trop extrême et rare**
- **Dans ce cas on va effectuer une seule comparaison par niveau donc h comparaisons où h est la hauteur de l'arbre -1**
- **Donc on doit calculer l'hauteur d'un arbre complet**

Théorème: pour tout arbre complet le nombre de feuilles $l = 2^h$ et le nombre de nœuds internes $m = 2^h - 1$

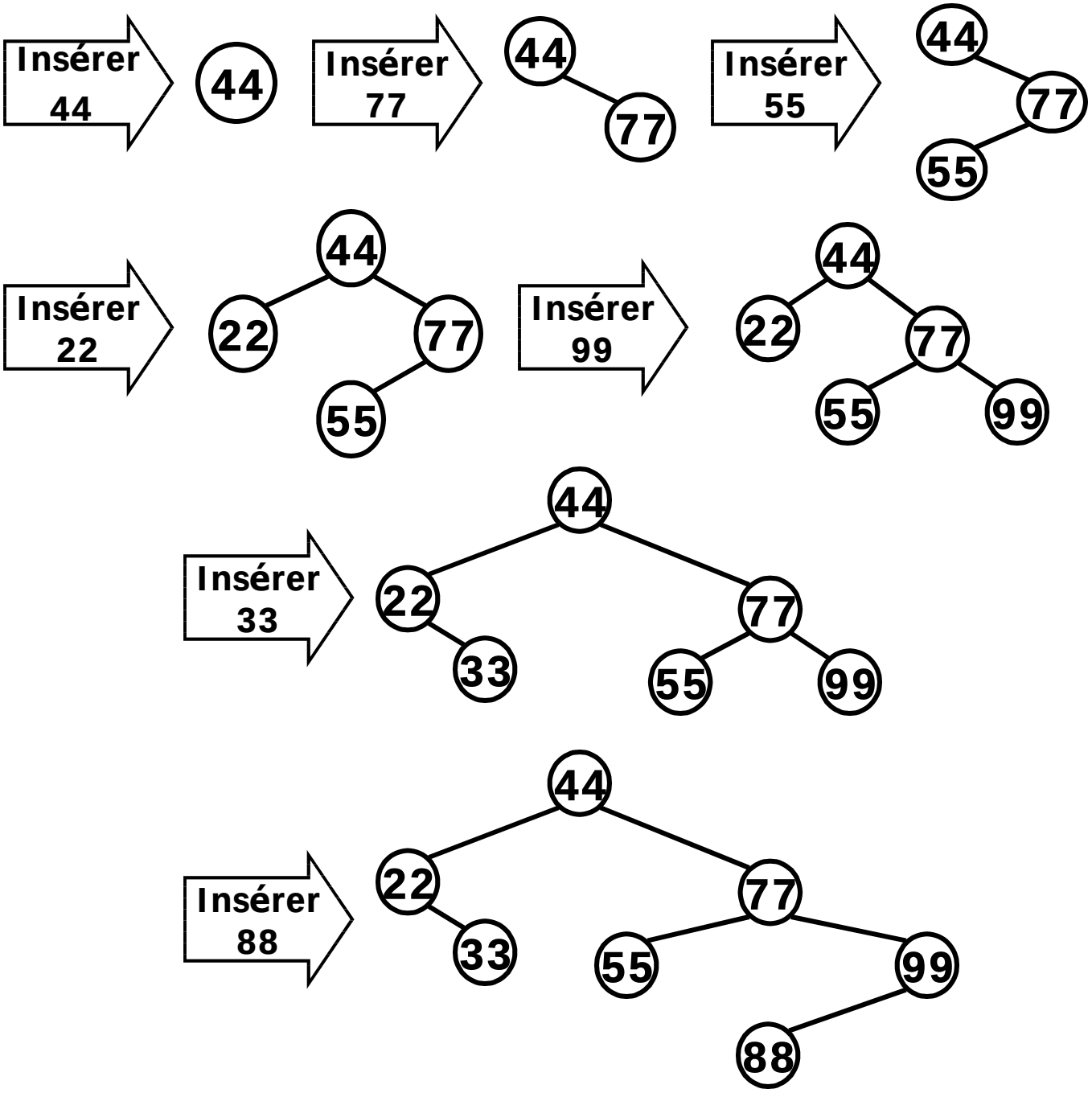
è le nombre de nœuds $n = l + m = 2^{(h+1)} - 1$

è $h = \text{Log}_2 (n) - 1$

- **Donc la complexité dans ce cas est $O(\text{Log}_2 n)$**

Insertion dans ABR

- **Exemple : 44, 77, 55, 22, 99, 33, 88**



Insertion dans un ARB version itérative

```
int inserer(ARBINRa, ELEMENT e){

int succee=1;

NOEUD n, p, q;
if (arbinRSature(a))
        { // 1
                printf ("\nArbre saturé");
                succee=0;}
else    {
                if  (appartient(a,e))
                {
                        printf ("\nElement existant");
                        succee=0;
                }
                else
                {  // 2
                        n=noeudCreer (e, NULL, NULL);
                        if (arbinRVide(a))
                                a->refRacine=n;
```

Insertion dans un ARB version itérative

```
else
{ // 3
p=a->refRacine;
while (p!=NULL)
            {
            q=p;
            if ((elementComparer(e,p->info))<0)
                        p=p->fg;
            else      /* c'est évident que c'est >0 car e
             n'existe pas donc elementComparer
             ne peut pas retourner 0 */
                        p=p->fd;
            }
if ((elementComparer(e,q->info)) > 0)
            q->fd= n;
else
            q->fg= n;
} // 3
} // 2
} // 1
return succee;
}
```

Insertion dans un ARB version récursive (1)

```
int inserer(ARBINR a, ELEMENT e)
{
int succee=1;

if (arbinRSature(a))
            {
            printf ("\nArbre saturé");
            succee=0;}
else
            {
            if  (appartient(a,e))
                        {
                        printf ("\nElement existant");
                        succee=0;
                        }
            else
            a->refRacine = insererNoeud(a->refRacine,
e);

            }
return succee;
}
```

Insertion dans un ARB version récursive (2)

```
NOEUD inserer Noeud(NOEUD n, ELEMENT e)
{

if (n==NULL)
        n=noeudCreer (e, NULL, NULL);
else
        {
        if ((elementComparer(e,n->info))<0)
                n->fg =inserer Noeud(n->fg, e);
        else    /* c'est évident que c'est >0 car e
                 n'existe pas donc elementComparer
                 ne peut pas retourner 0*/
                n->fd =inserer Noeud(n->fd, e);
        }
return n;
}
```

Exemple

```
/*****************************************
*   Fichier : main.c
*   Contenu : Test du TDA ARBINR
*****************************************/
#include <stdio.h>
#include "ELTPRIM.H"
#include "ARBRPRIM.H"

int main()
{
ELEMENT elt;
ARBINR a = arbinRCreer();

elt = elementCreer();
elt= 44;
inserer(a, elt);

elt = elementCreer();
elt=77;
inserer(a, elt);

elt = elementCreer();
elt=55;
inserer(a, elt);

elt = elementCreer();
elt=22;
inserer(a, elt);

elt = elementCreer();
elt=99;
inserer(a, elt);

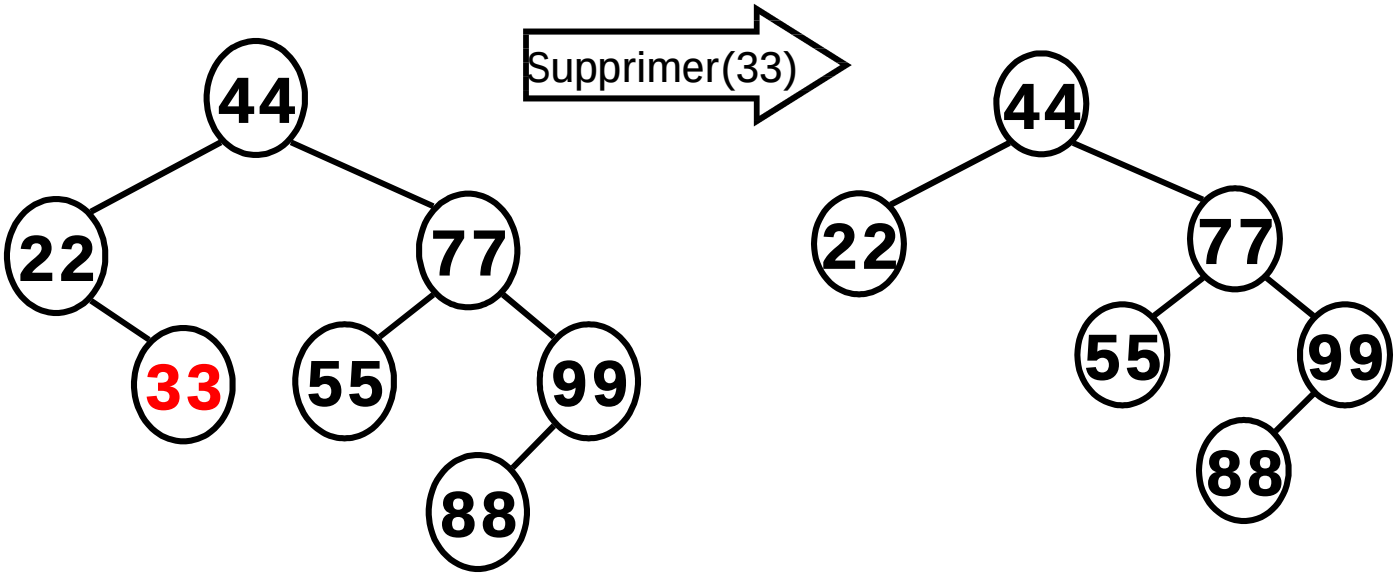
elt = elementCreer();
elt=33;
inserer(a, elt);

elt = elementCreer();
elt=88;
inserer(a, elt);

arbinRAfficher(a, 1);
}
```

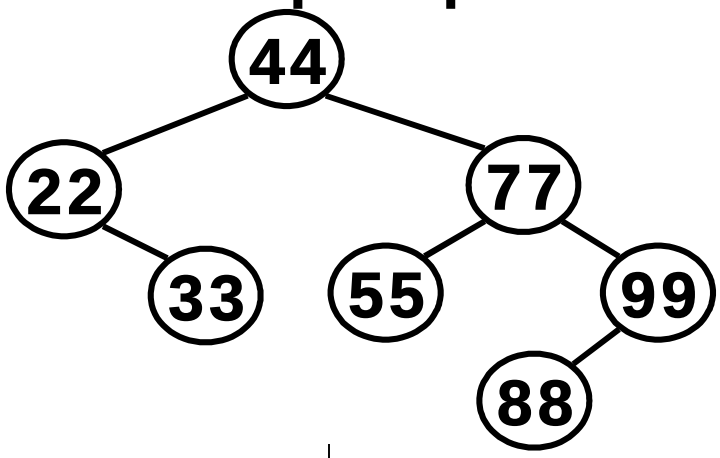
Supprimer un élément dans un ABR (1)

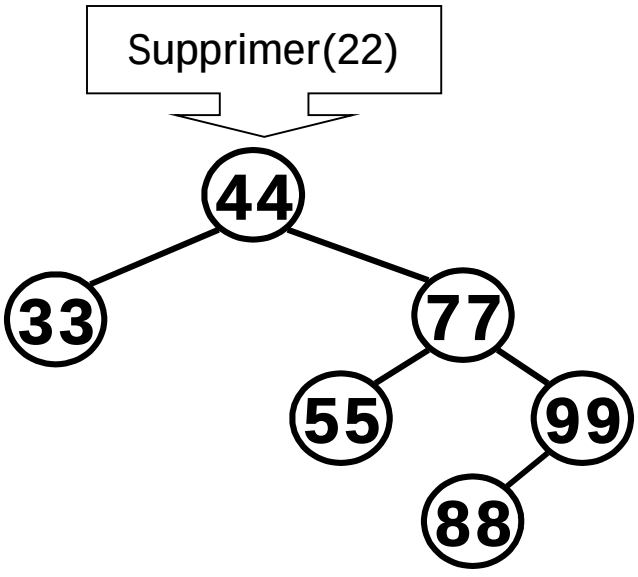
- **Si le nœud dont la valeur de clé est spécifiée par l ′utilisateur n′est pas trouvé dans l′arbre alors échec sinon le supprimer selon les cas suivants:**
- **Soit A un ABR, N le nœud à supprimer et P(N) le père de N**
  - **Premier cas : N a 0 fils (N est une feuille). On supprime N et on remplace son adresse dans son père par NULL**



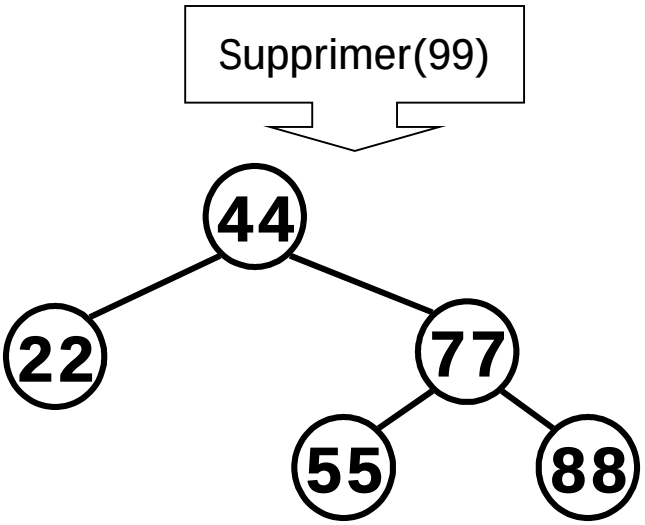
Supprimer un élément dans un ABR (2)

- **<u>Deuxième cas</u> : N a un seul fils (FD ou FG). On supprime N et on remplace son adresse dans son père par l'adresse de son fils.**





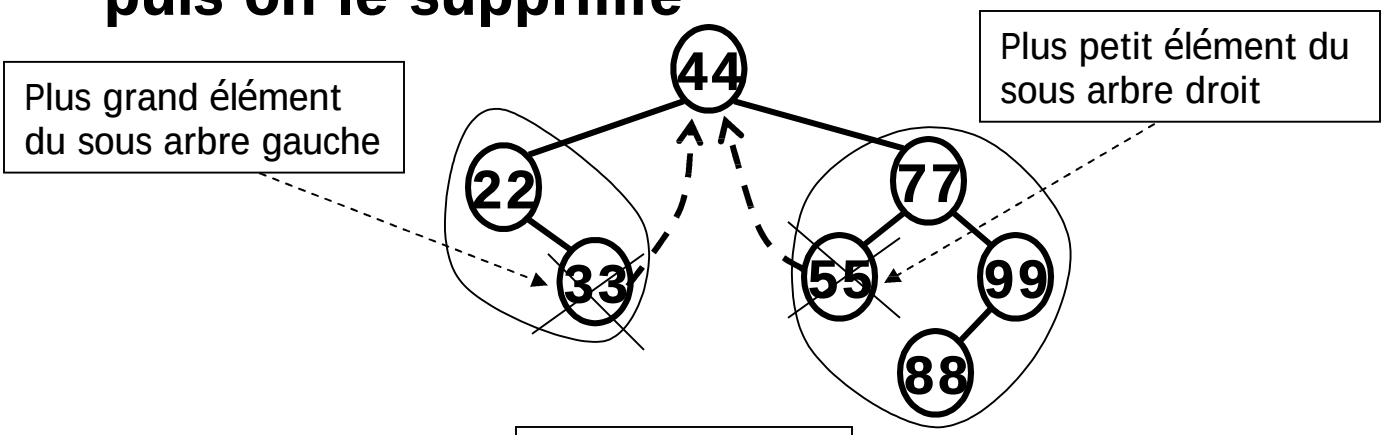
Le fils droit du nœud 22 devient le fils droit du P(22)



Le fils gauche du nœud 99 devient le fils gauche du P(99)

Supprimer un élément dans un ABR (3)

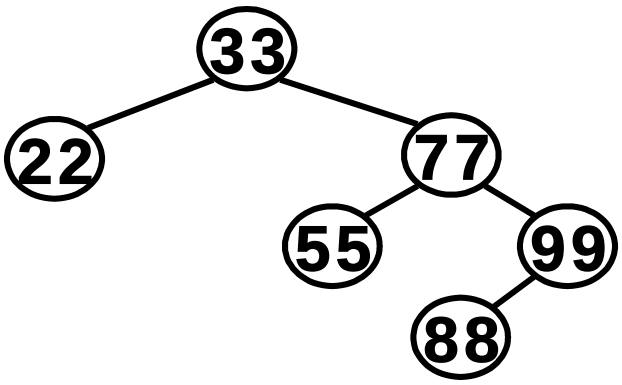
- **Troisième cas : N a deux fils (FD et FG). On supprime N et on le remplace par : (1) l'élément maximal de son sous arbre gauche puis on le supprime ou (2) l'élément minimal de son sous arbre droit puis on le supprime**



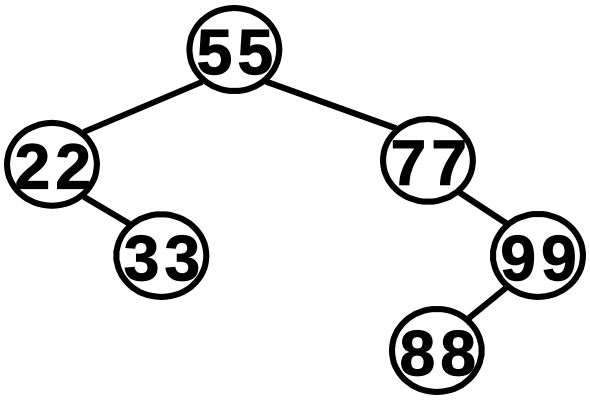
Supprimer(44)

Méthode 1 ou Méthode 2

Le nœud 44 est remplacé par le nœud 33 et on le supprime



Le nœud 44 est remplacé par le nœud 55 et on le supprime



Supprimer un élément dans un ABR (1) - récursive

```
int supprimer (ARBINR a, ELEMENT e){

int succee=1;
if (arbinRVide(a))
        { // 1
        printf ("\nArbre vide");
        succee=0;}
else
        {
        if  (!appartient(a,e))
                {
                printf ("\nElement non existant");
                succee=0;
                }
        else
    a->refRacine = supprimerNoeud(a->refRacine,e);
        }
return succee;
}
```

Supprimer un élément dans un ABR (2) - récursive

```
NOEUD supprimerNoeud(NOEUD n, ELEMENT e) {
NOEUD z;
if (elementComparer(e,n->info) < 0)
        n->fg = supprimerNoeud(n->fg,e);
        else
            if (elementComparer(e,n->info) > 0)
            n->fd = supprimerNoeud(n->fd,e);
            else  // égalité
                  if ( (n->fg != NULL) && (n->fd != NULL))
                      {
                      z =rechercheMin(n->fd) ;
                      elementCopier(&n->info, z->info);
                      n->fd = supprimeMin(n->fd);
                      noeudDetruire(z);
                      }
                  else
                      {
                      if (n->fd != NULL)
                            n=  n->fd;
                      else
                            n=n->fg;
                      }
return n;}
```

3ème Cas 2ème méthode

1er et 2ème cas

Supprimer un élément dans un ABR (3) - récursive

```
NOEUD rechercheMin(NOEUD n)
   {NOEUD m =n;
          while (m->fg != NULL)
                    m = m->fg;
    return m;
    }
```

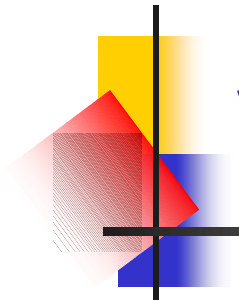
le plus petit élément du SAD se trouve nécessairement à gauche

```
NOEUD supprimeMin(NOEUD n)
   {if (n->fg != NULL)
          {
          n->fg= supprimeMin (n->fg);
          return n;
          }
    else
          return n->fd;
    }
```

On re-recherche le min et on le supprime

Le min n'a pas de FG

Mais il peut avoir un fils droit et dans ce cas ce fils doit le remplacer



Supprimer un élément dans un ABR – itérative

- **il existe une solution de la suppression avec un parcours itératif mais qui utilise SupprimeMin qui est récursive**

Supprimer un élément dans un ABR – itérative (1)

```
int supprimer (ARBINR a, ELEMENT e){

int succee=1;
NOEUD z, p, q;
int test=0;
int supprimeRacine=0;

if (arbinRVide(a))
        {
        printf ("\nArbre vide");
        succee=0;
        }
else
        {       // 1
        p=a->refRacine;
        if (elementComparer(e,p->info) == 0)
                {
                supprimeRacine =1;
                test=1;
                }
```

Supprimer un élément dans un ABR – itérative (1)

```
int supprimer (ARBINR a, ELEMENT e){

int succee=1;
NOEUD z, p, q;
int test=0;
int supprimeRacine=0;

if (arbinRVide(a))
        {
        printf ("\nArbre vide");
        succee=0;
        }
else
        {       // 1
        p=a->refRacine;
        if (elementComparer(e,p->info) == 0)
                {
                supprimeRacine =1;
                test=1;
                }
```

Supprimer un élément dans un ABR – itérative (2)

```
while ((p !=NULL) && (!test))
                    { // 2
                    if (elementComparer(e,p->info) < 0)
                              {q=p;
                               p=p->fg;}
                    else
                    if (elementComparer(e,p->info) > 0)
                              {q=p;
                               p=p->fd;}
                    else
                              test=1;
                    }   // 2
// à la sortie p est l'élément à supprimer et q est son père
if (test) // on a trouvé l'élément
          { // 3
          if ((p->fd!=NULL) && (p->fg!=NULL))
                    {
                    z =rechercheMin(p->fd) ;
                    elementCopier(&p->info, z->info);
                    p->fd = supprimeMin(p->fd);
                    noeudDetruire(z);
                    }
```

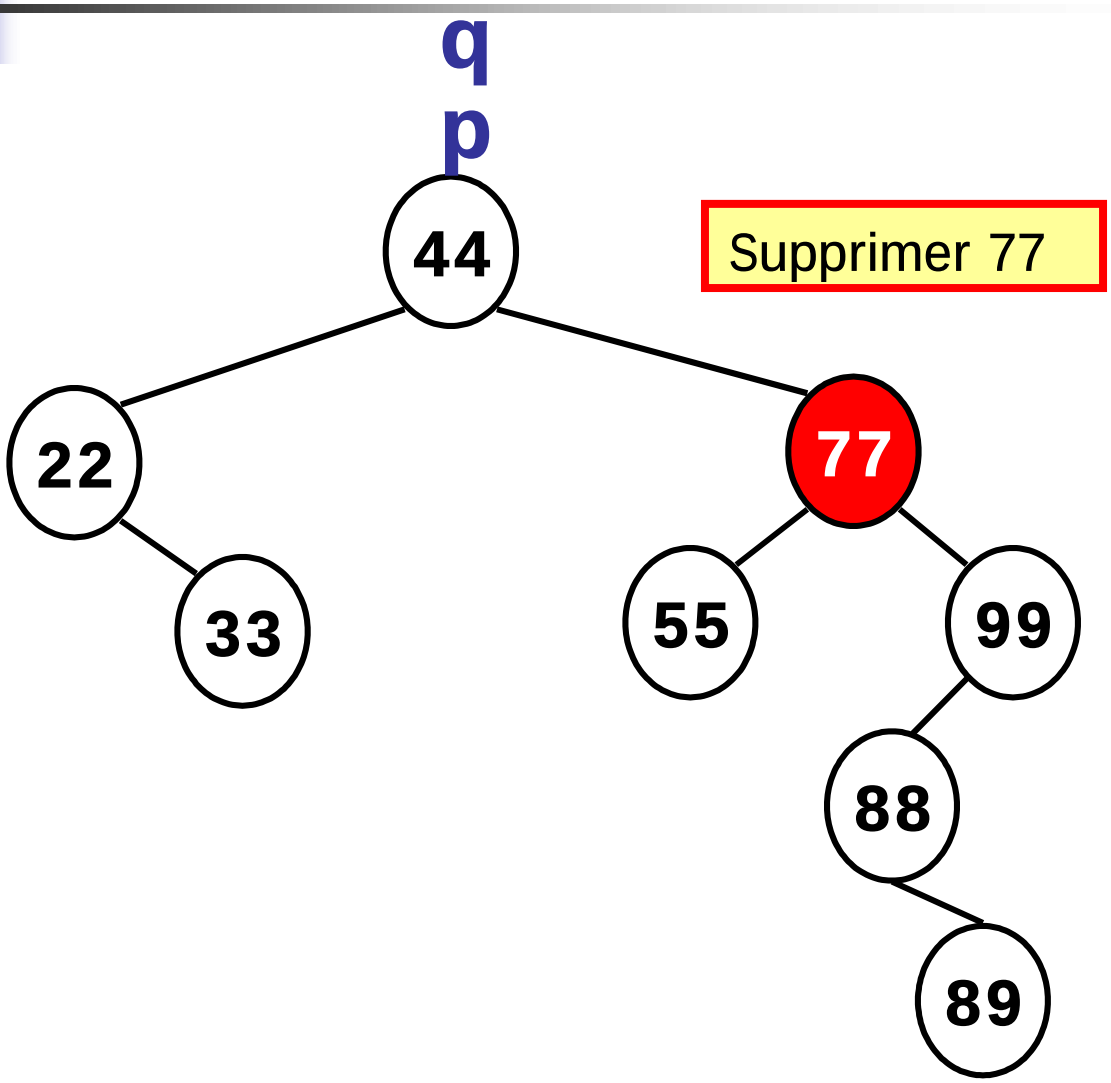
3ème Cas 2ème méthode

Supprimer un élément dans un ABR – itérative (3)

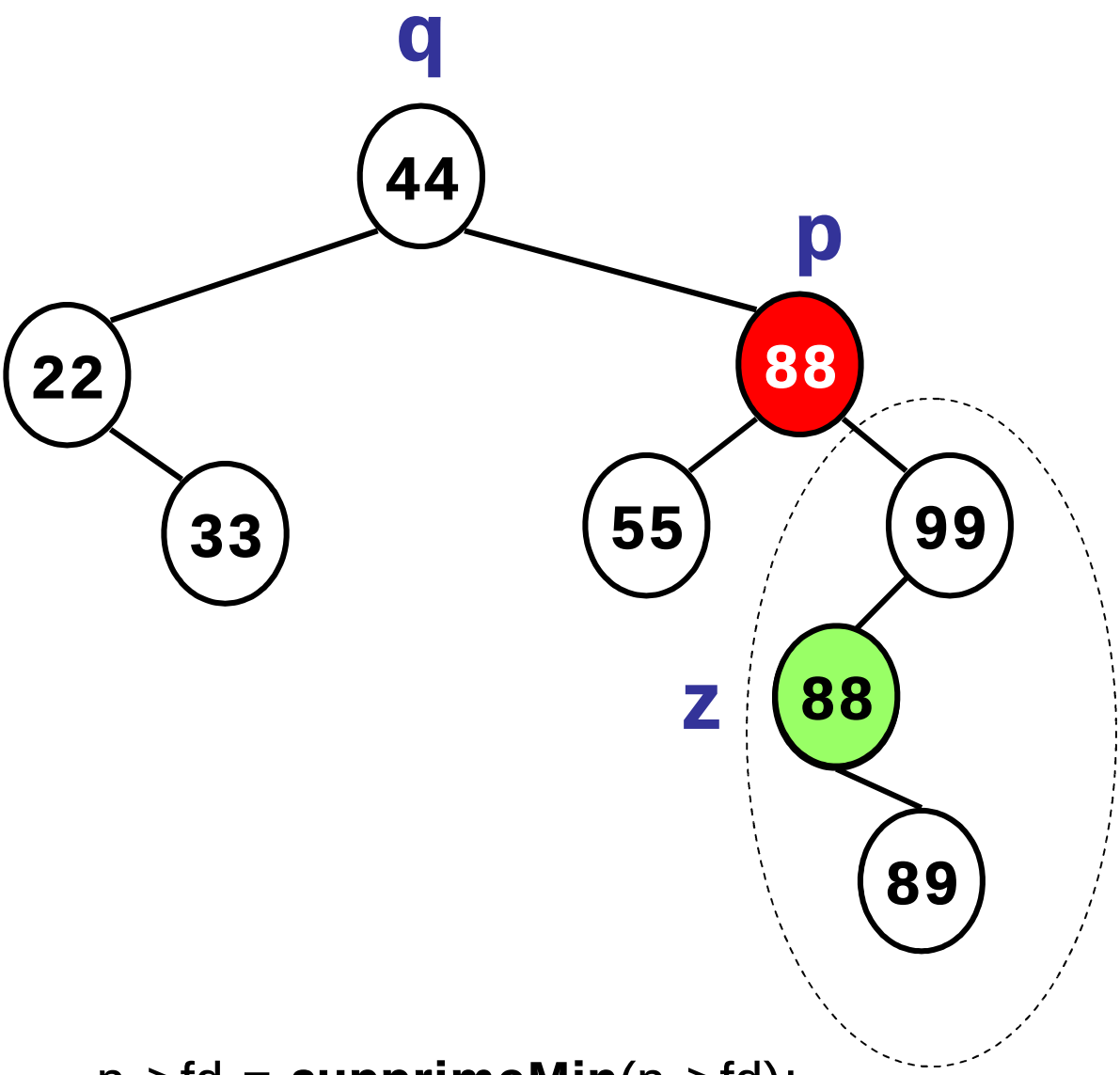
1er et 2ème cas

```
else
        if (p->fd!=NULL)
                {if (supprimeRacine)
                        a->refRacine=p->fd;
                else
                 { if (q->fd==p) // l'élém à supp :FD
                        q->fd= p->fd;
                   else  // l'élément à supp est un FG
                        q->fg= p->fd;
                 }}
        else // p a un fils gauche ou bien p est une feuille
                {if (supprimeRacine)
                        a->refRacine=p->fg;
                else
                {       if (q->fd==p) // l'élém à supp: FD
                                q->fd= p->fg;
                        else // l'élément à supp est un FG
                                q->fg= p->fg;
                }} } // 3
else // on n'a pas trouvé l'élément
        {printf ("\nElement existant");
        succee=0;
        }}
return succee;}
```

Exemple 1

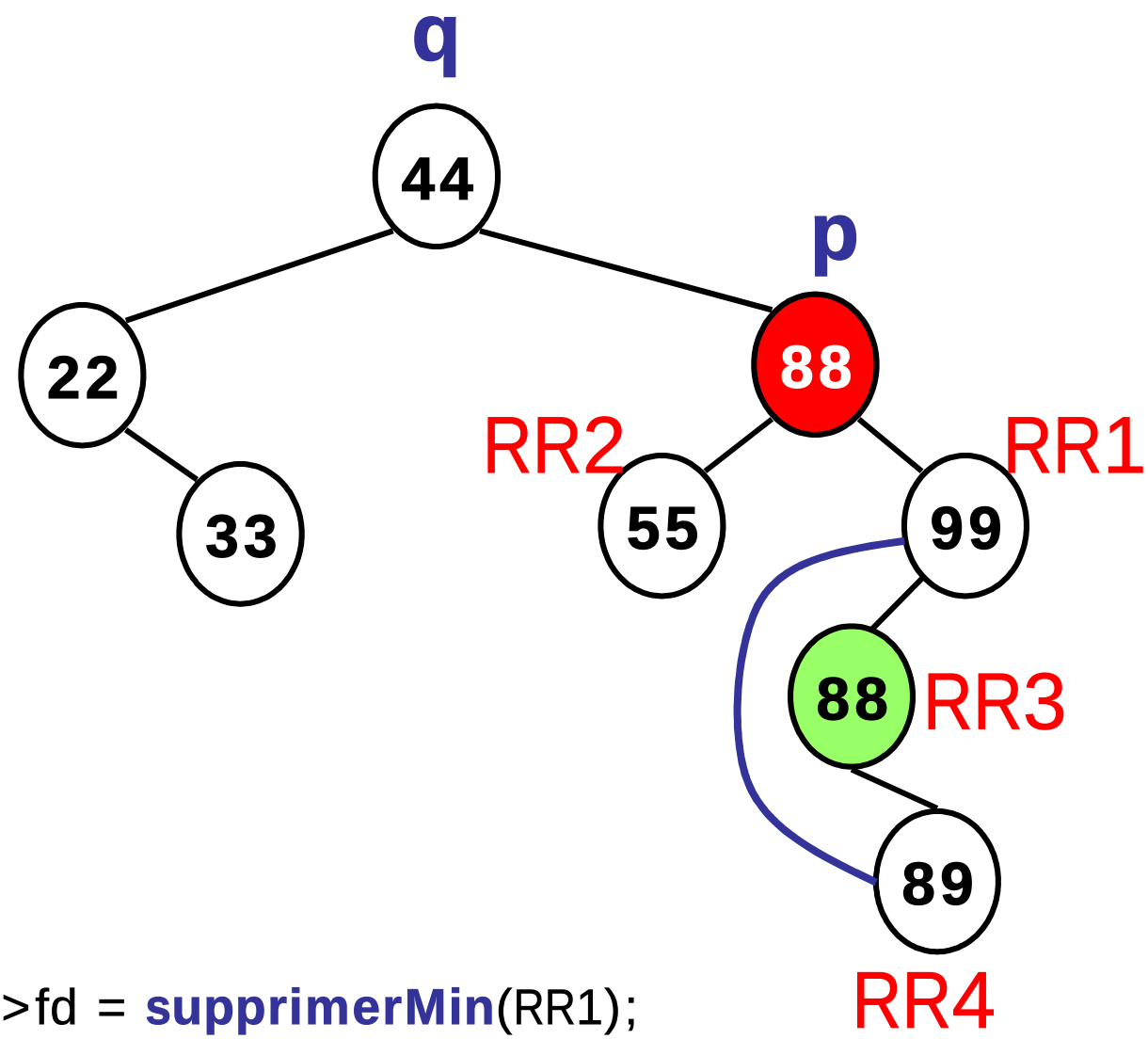


Exemple 1 (suite)



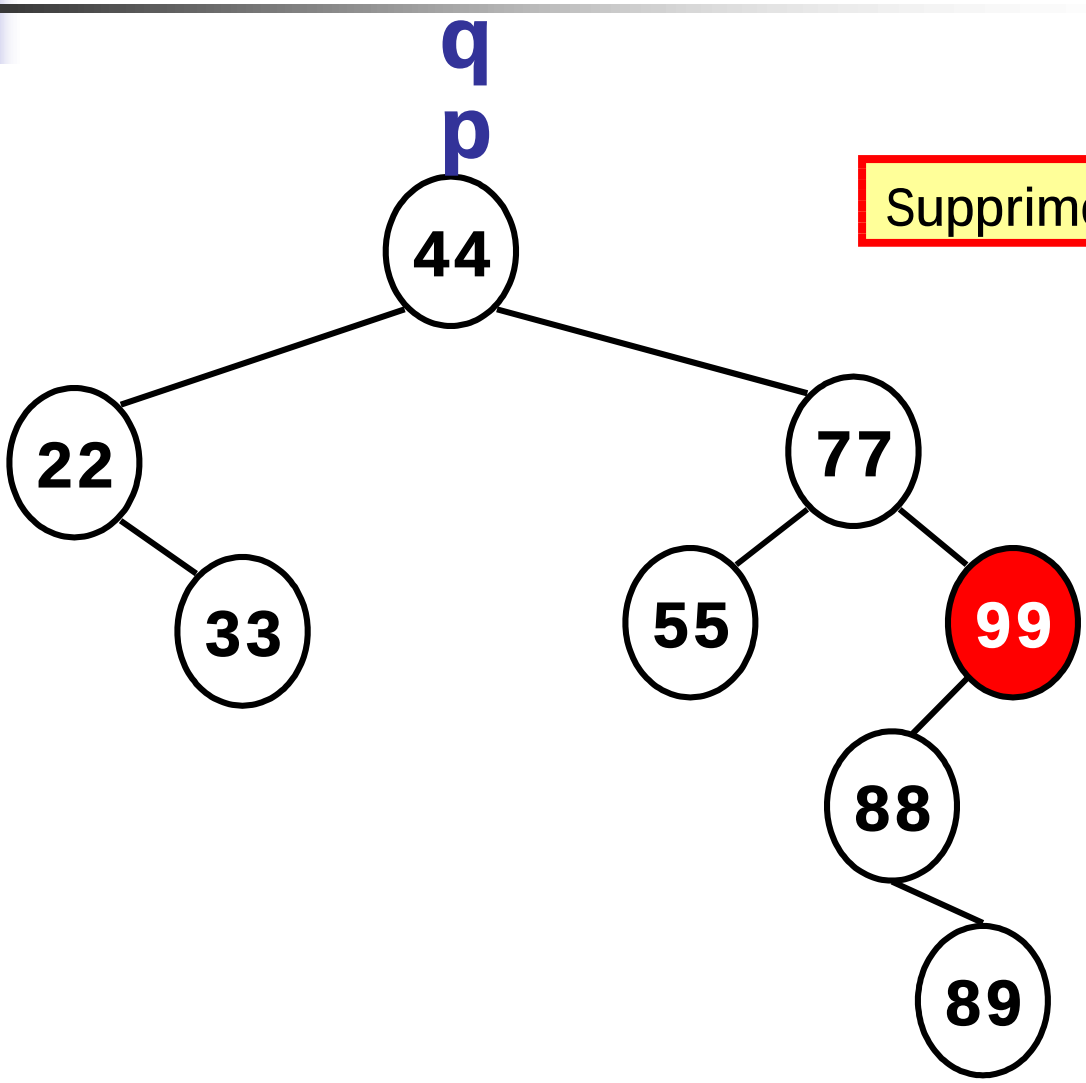
p->fd = **supprimeMin**(p->fd);

```
{if (n->fg != NULL)       {
            n->fg= supprimeMin (n->fg);
            return n;           }
    else
            return n->fd;     }
```



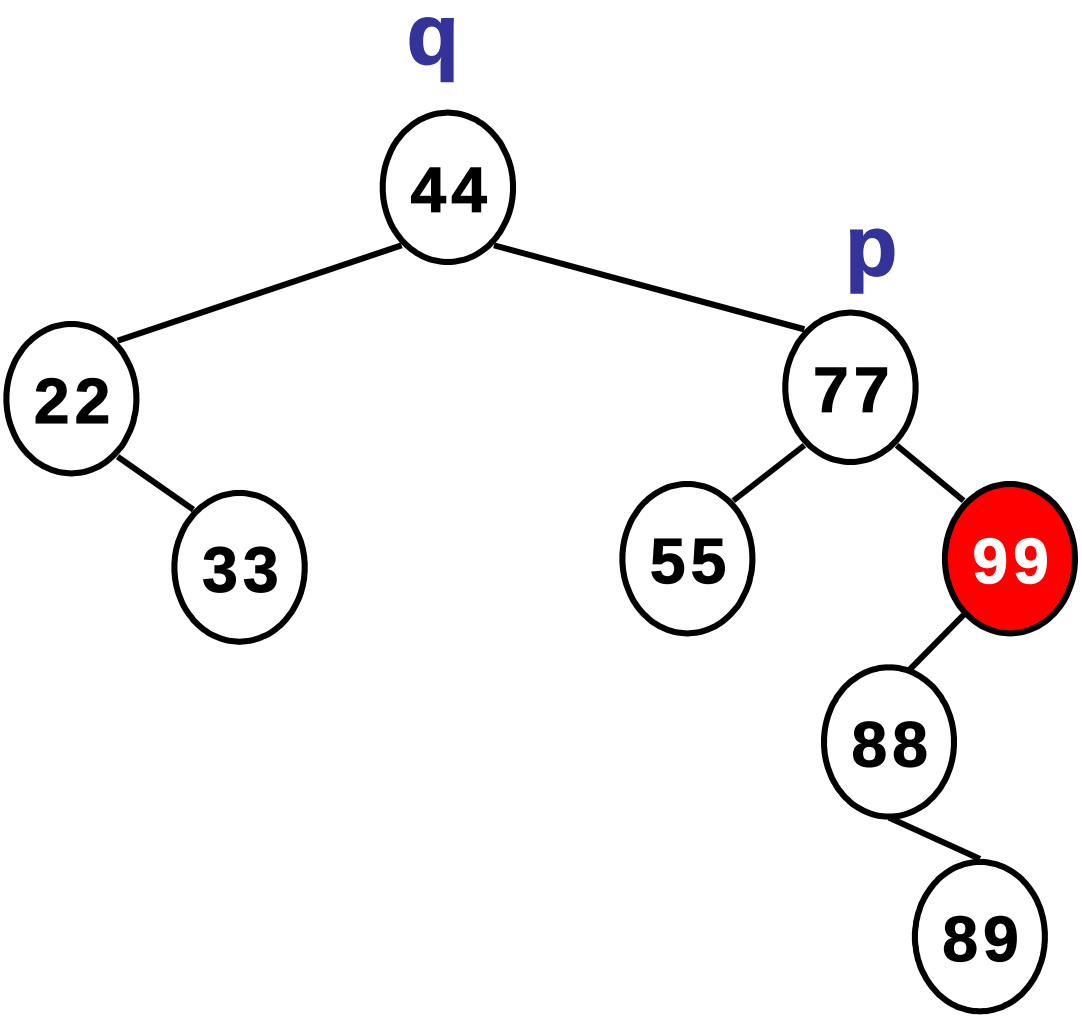

```
p->fd = supprimerMin(RR1);
      RR1->fg != null
      RR1->fg= supprimerMin(RR3);
                RR3->fg == NULL
                return RR4 cad RR1->fg = RR4
      return RR1 cad p->fd =RR1
```

Exemple 2

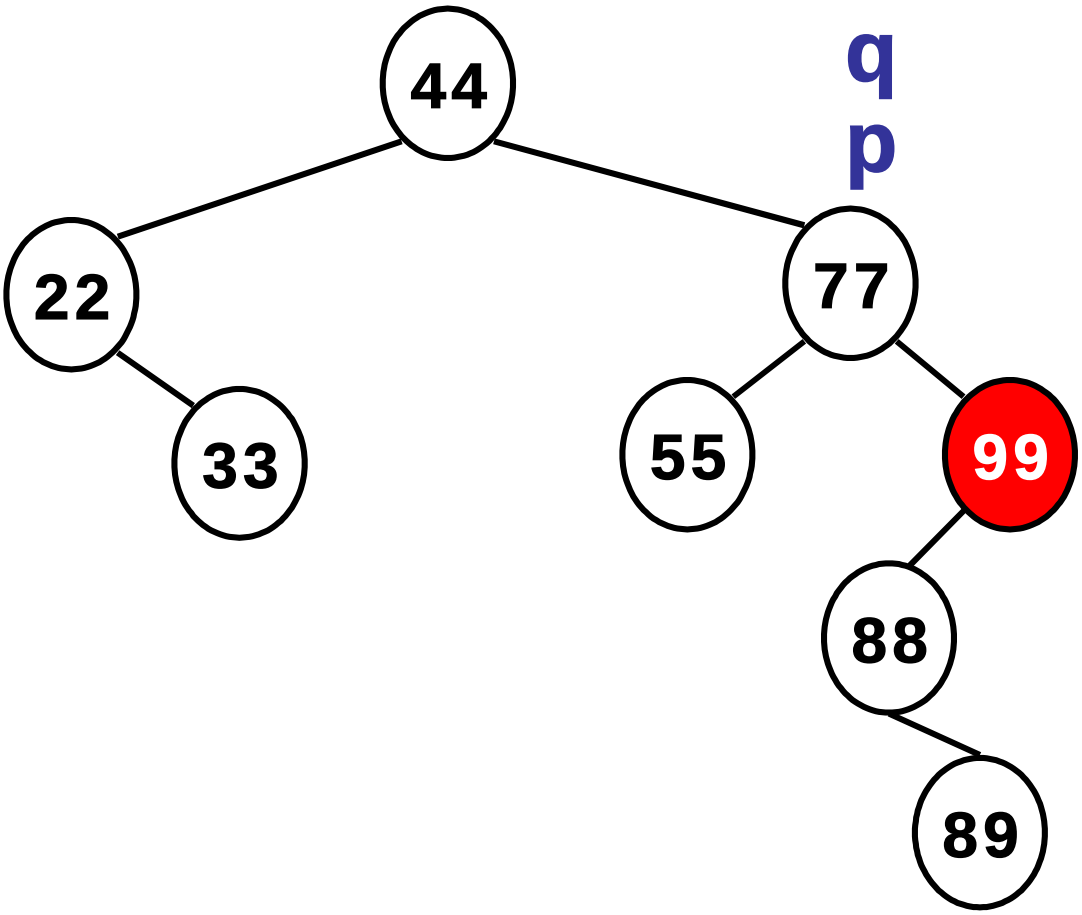


Supprimer 99

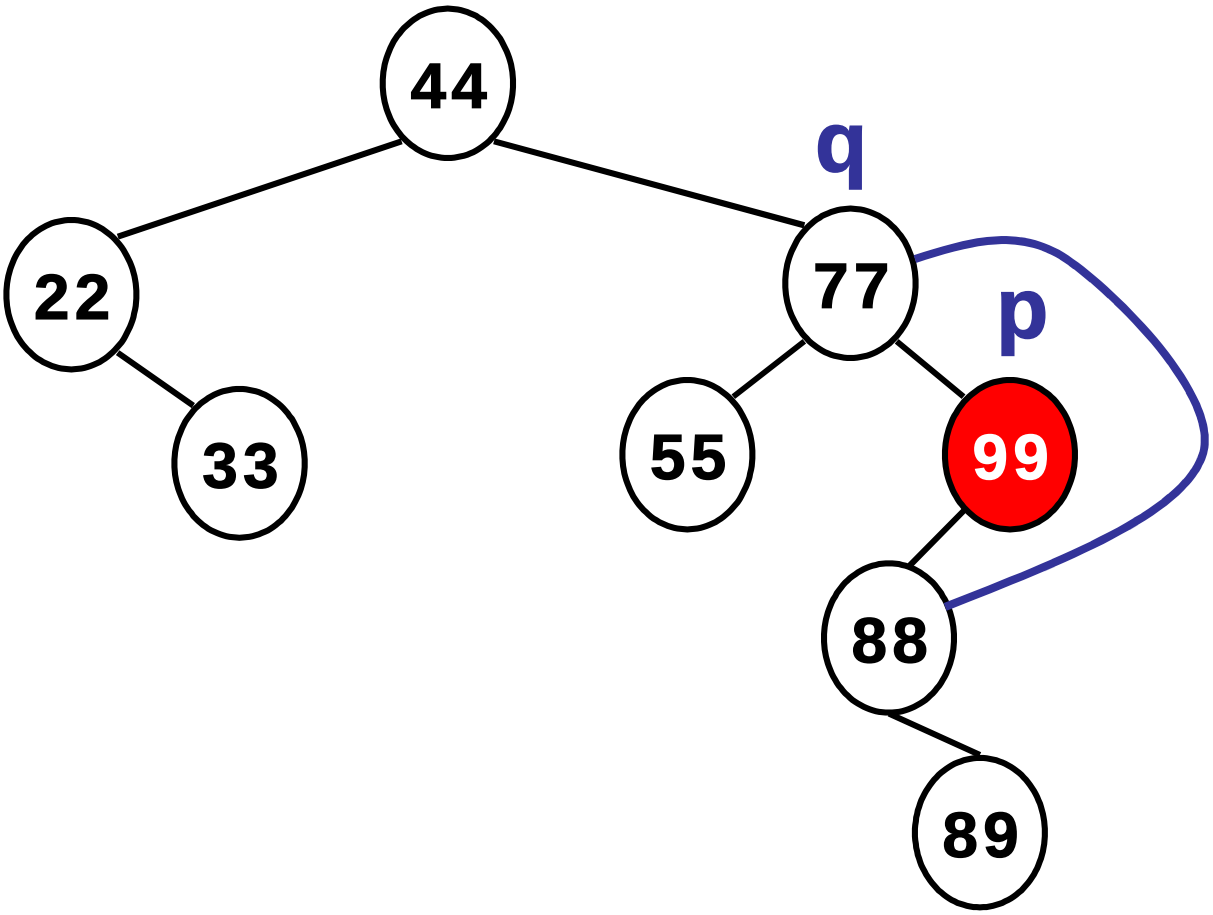
Exemple 2 (suite)



Exemple 2 (suite)



```
else
        {       if (q->fd==p) // l'élém à supp: FD
                        q->fd= p->fg;
                else // l'élément à supp est un FG
                        q->fg= p->fg;
        }
```



q->fd==p (99 est un FD à 77)
q ->fd = p ->fg