Chap. 5: Implémentation des structures de données

- Implémenter: définir la structure de données concrète et écrire les algorithmes qui réalisent (implémentent) la spécification d'une structures de données abstraite telle que nous l'avons vue au chapitre 4.
- Exigences:
  - fidèle à la spécification
  - efficace en terme de complexité d'algorithme
- Pour implémenter, nous avons à disposition:
  - les types élémentaires (entiers, caractères, ...)
  - les références aux objets (pointeurs)
  - les tableaux et les enregistrements
- on retrouve fréquement certaines structures de données dans l'implémentations des types abstraits: listes chaînées, arbres de recherches et tables de hachage. Avec les types prédéfinis dans le langage Oberon, elles forment les briques de base des implémentations.
- Il y a toujours plusieurs implémentations possibles pour une même structure de données abstraite

Implémentation des types collection ordonnée

Rappel: il s'agit de la pile, queue et séquence

Implémentation avec:

- tableaux
- listes chaînées

Discussion:

- avec tableaux, facile à implémenter mais
  - taille fixée à la compilation
  - insertion/suppression au milieu de la structure coûteuse (séquence), aux deux extrémités astucieuses (queue, séquence)
- avec listes chaînées:
  - taille variable
  - insertion/suppression au milieu de la structure OK
  - insertion/suppression aux deux extrémités OK (à condition d'avoir une référence directe au début ET à la fin de liste)
  - mais accès au i-ème élément demande le parcours séquentiel de la liste depuis le début

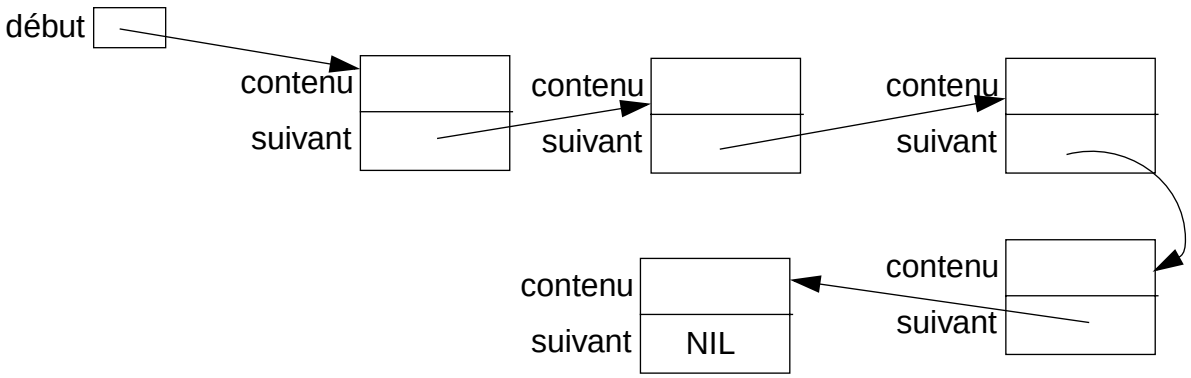
Remarque:

Nous ne verrons pas l'implémentation avec les tableaux

Liste chaînée

Une liste chaînée est composée d'un ensemble de noeuds qui contiennent chacun un élément et le lien vers le noeud suivant.

Illustration:



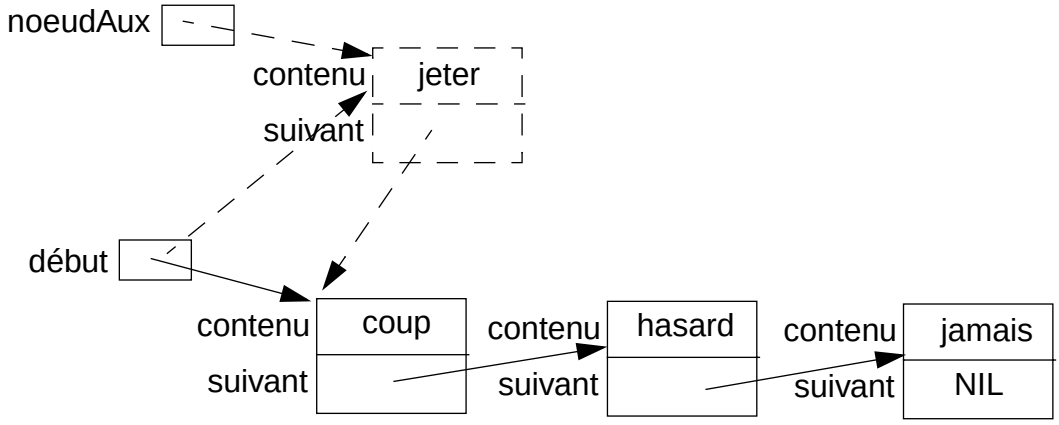
Déclaration Oberon:

```
TYPE    Elément = ARRAY 30 OF CHAR;
                          (* par exemple*)

        Noeud = POINTER TO RECORD
                    contenu: Elément
                    suivant: Noeud;
                END;

VAR début, noeudAux: Noeud;
```

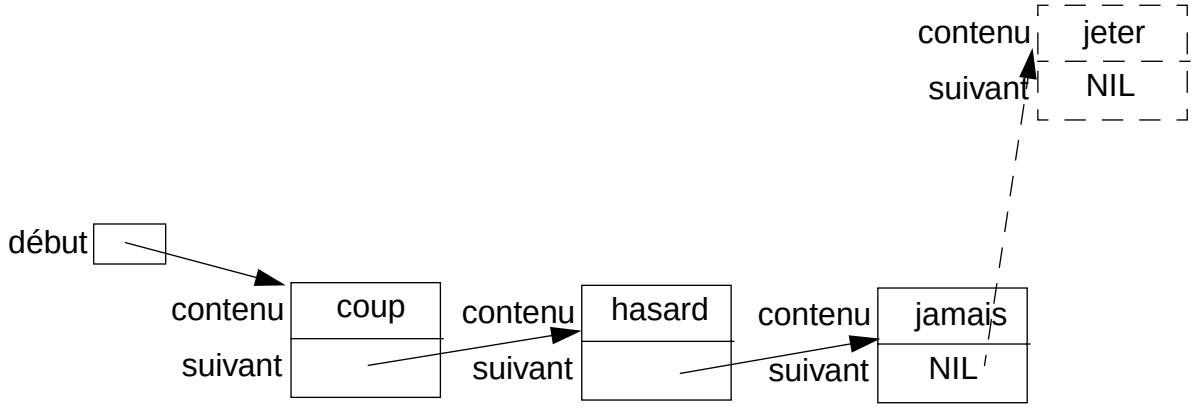
L’opération d’insertion au début



Code Oberon:

```
NEW(noeudAux);
noeudAux.contenu:=”jeter”;
noeudAux.suivant:=début;
début:=noeudAux;
```

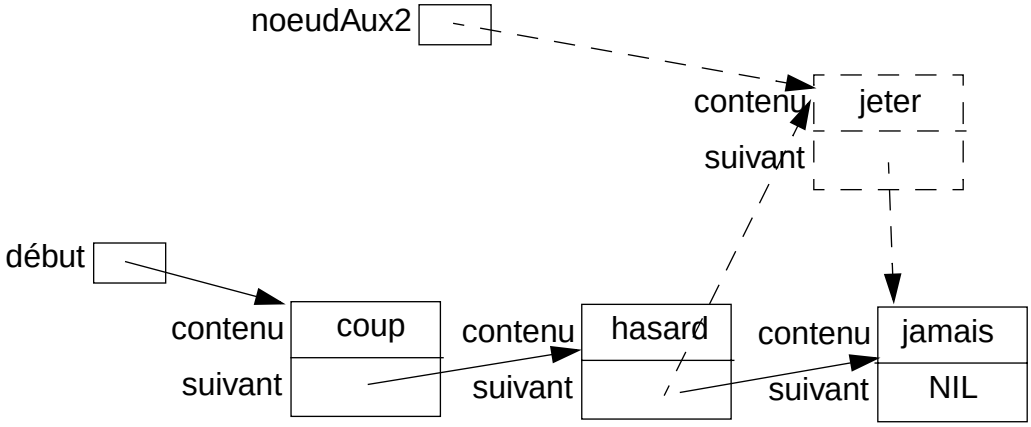
L’opération d’insertion à la fin



Code Oberon:

```
noeudAux:=début;
WHILE noeudAux.suivant # NIL DO
      noeudAux:=noeudAux.suivant;
END;
NEW(noeudAux.suivant)
noeudAux.suivant.contenu:=”jeter”;
noeudAux.suivant.suivant:=NIL; (* facultatif *)
```

L’opération d’insertion au milieu de la liste (insérer en i-ème position)



Code Oberon:

```
i:=3; (* si insertion en 3ème position de la liste *)
noeudAux:=début;
j:=1;
WHILE (noeudAux # NIL) & (j < i-1) DO
      noeudAux:=noeudAux.suivant;
      INC(j);
END;
IF j = i-1THEN
      NEW(noeudAux2);
      noeudAux2.contenu:=”jeter”;
      noeudAux2.suivant:=noeudAux.suivant;
      noeudAux.suivant:=noeudAux2;
END;
```

Implémentation de la liste

```
MODULE MListe;

TYPE Elément* = ARRAY 30 OF CHAR; (* par exemple *)
        Noeud = POINTER TO RECORD
                                contenu: Elément;
                                suivant: Noeud;
                        END;

       Liste* = EXTENSIBLE RECORD
                                début: Noeud;
                        END;

PROCEDURE (VAR liste: Liste)InitListe*, NEW;
BEGIN
       liste.début:=NIL;
END InitListe;

PROCEDURE (VAR liste: Liste)InsérerDébut*(e: Elément), NEW;
VAR noeudAux: Noeud;
BEGIN
       NEW(noeudAux);
       noeudAux.contenu:=e;
       noeudAux.suivant:=liste.début;
       liste.début:=noeudAux;
END InsérerDébut;

PROCEDURE (VAR liste: Liste)SupprimerDébut*, NEW;
BEGIN
       IF liste.début # NIL THEN liste.début := liste.début.suivant; END;
END SupprimerDébut;
```

Implémentation de la liste (suite)

```
PROCEDURE (VAR liste: Liste)InsérerFin*(e: Elément), NEW;
VAR noeudAux: Noeud;
BEGIN
        noeudAux:=liste.début;
        WHILE noeudAux.suivant # NIL DO
                noeudAux:=noeudAux.suivant;
        END;
        NEW(noeudAux.suivant);
        noeudAux.suivant.contenu:=e;
        noeudAux.suivant.suivant:=NIL; (* facultatif *)
END InsérerFin;

PROCEDURE (VAR liste: Liste)InsérerEnI*(e: Elément; i: INTEGER),
NEW;
VAR noeudAux, noeudAux2: Noeud;
        j: INTEGER;
BEGIN
        noeudAux:=liste.début;
        j:=1;
        WHILE (noeudAux # NIL) &( j < i-1) DO
                noeudAux:=noeudAux.suivant;
                INC(j);
        END;
        IF j = i-1THEN
                NEW(noeudAux2);
                noeudAux2.contenu:=e;
                noeudAux2.suivant:=noeudAux.suivant;
                noeudAux.suivant:=noeudAux2;
        END;
END InsérerEnI;

(* ... etc ... *)

END MListe.
```

Interface de la liste en Obéron

```
DEFINITION MListe;

    TYPE
        Elément = ARRAY 30 OF CHAR;

        Liste = EXTENSIBLE RECORD
            (IN liste: Liste) Début (): Elément, NEW;
            (IN liste: Liste) Fin (): Elément, NEW;
            (VAR liste: Liste) InitListe, NEW;
            (VAR liste: Liste) InsérerDébut (e: Elément), NEW;
            (VAR liste: Liste) InsérerEnI (e: Elément; i: INTEGER),
                                                        NEW;
            (VAR liste: Liste) InsérerFin (e: Elément), NEW;
            (IN liste: Liste) ItérerDepuisDébut (Traiter:
                                PROCEDURE (e: Elément)), NEW;
            (IN liste: Liste) ItérerDepuisFin (Traiter:
                                PROCEDURE (e: Elément)), NEW;
            (IN liste: Liste) Longueur (): INTEGER, NEW,
                                                    EXTENSIBLE;
            (VAR liste: Liste) SupprimerDébut, NEW;
            (VAR liste: Liste) SupprimerI ( i: INTEGER), NEW;
            (VAR liste: Liste) SupprimerFin, NEW;
            (IN liste: Liste) Vide (): BOOLEAN, NEW, EXTENSIBLE
        END;

END MListe.
```

Implémentation de la pile

Implémentation triviale avec la liste !

```
MODULE MPile;

IMPORT MListe;

TYPE Pile* = RECORD (MListe.Liste) END;
        Elément* = MListe.Elément;

(* constructeurs / modifieurs *)

PROCEDURE (VAR pile: Pile)InitPile*, NEW;
BEGIN pile.InitListe; END InitPile;

PROCEDURE (VAR pile: Pile)Empiler*(e: Elément), NEW;
BEGIN pile.InsérerDébut(e); END Empiler;

PROCEDURE (VAR pile: Pile)Dépiler*, NEW;
BEGIN pile.SupprimerDébut; END Dépiler;

(* sélecteurs *)

PROCEDURE (IN pile: Pile)Sommet*():Elément, NEW;
BEGIN RETURN pile.Début; END Sommet;

PROCEDURE (IN pile: Pile)Profondeur* (): INTEGER, NEW;
BEGIN RETURN pile.Longueur; END Profondeur;

PROCEDURE (IN pile: Pile)EstVide* (): BOOLEAN, NEW;
BEGIN RETURN pile.Vide(); END EstVide;

END MPile.
```

Interface de la pile en Oberon

```
DEFINITION MPile;

    IMPORT MListe;

    TYPE
        Elément = MListe.Elément;

        Pile = RECORD (MListe.Liste)
            (VAR pile: Pile) Dépiler, NEW;
            (VAR pile: Pile) Empiler (e: MListe.Elément), NEW;
            (IN p: Pile) EstVide(): BOOLEAN, NEW;
            (VAR pile: Pile) InitPile, NEW;
            (IN p:Pile) Profondeur(): INTEGER, NEW;
            (IN p:Pile) Sommet(): Elément, NEW;
        END;

END MPile.
```

Implémentation des types collection non-ordonnée

Rappel: ensemble, multi-ensemble, fonction

On peut les implémenter avec une liste chaînée mais:

- complexité en temps pour l'insertion: constante si insertion en tête de liste
- la complexité en temps pour la recherche d'un élément: *O*(n)

Si les éléments du type collection possède une relation d'ordre, l'implémentation avec un arbre de recherche est plus efficace:

- complexité insertion, recherche, suppression: *O*(log n)

Pour rappel:

- si *n*=100'000. Nombre moyen de comparaisons:

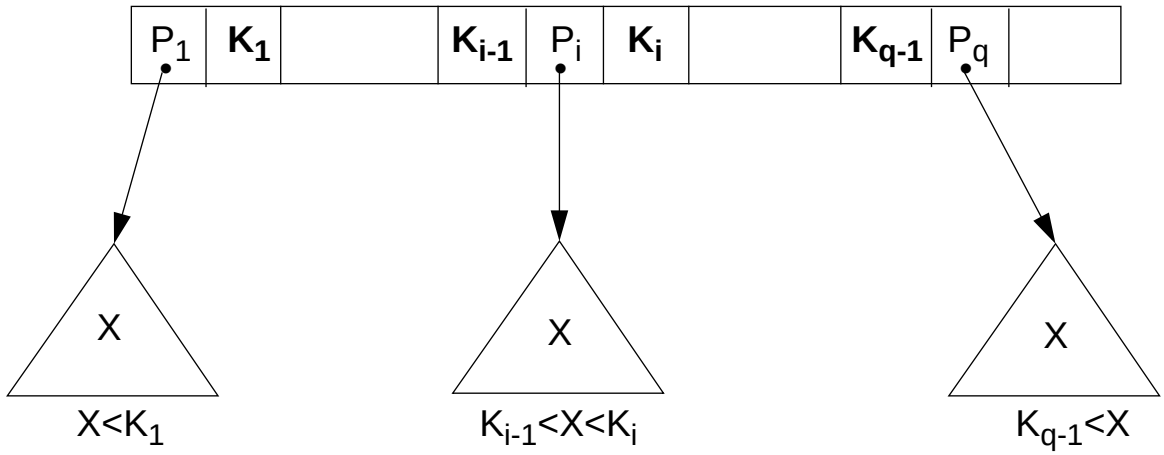
  -> recherche linéaire *O*(n): 50'000
  -> arbre binaire *O*(log n): 17

Arbres de recherche (rappel)

Arbre spécial pour guider la recherche d'un enregistrement en fonction d'une clé et d'une valeur de clé.

Arbre de recherche d'ordre p:

- Chaque noeud contient q-1 valeurs de clé et q pointeurs dans l'ordre $<P_1,K_1,P_2,K_2,...,P_{q-1},K_{q-1},P_q>$ avec $q \leq p$ et $K_1 < K_2 < ... < K_{q-1}$
  ($P_1,P_2,...,P_{q-1},P_q$ sont les pointeurs vers les noeuds descendant)



Un noeud peut donc avoir au maximum p descendants

Hauteur pour n valeurs: $\log_p(n)$ –> complexité de recherche $O(\log_p(n))$

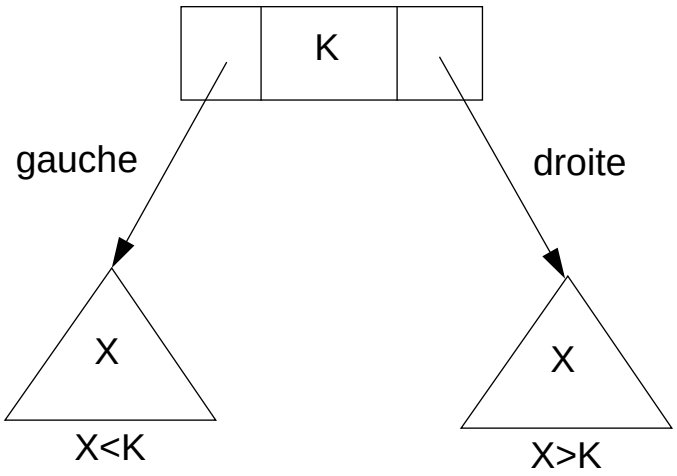
Remarque:

Nous supposons que la clé est unique. Cette restriction peut être levée, mais il faut changer légèrement les formules.

Arbre de recherche binaire (rappel)

Chaque noeud contient:

- une clé (K)
- un pointeur gauche et un pointeur droit t.q.



Opération de l’arbre de recherche (binaire)

Rappel:

Constructeurs d’arbre de recherche

opération	paramètre	résultat	description
new		A: Arbre	crée une nouvel arbre A
initArbre			initialise l’arbre à vide
insérer	c: clé		insérer un élément de clé c dans l’arbre
supprimer	c: clé		supprime l’élément dont la clé vaut c
copier		Arbre	faire une copie de l’arbre

Sélecteurs d’arbre de recherche

opération	paramètre	résultat	description
rechercher	c: clé	booléen	retourne vrai si l’élément dont la clé vaut c appartient à l’arbre
estVide		booléen	vrai si l’arbre est vide
égal	B: Arbre		vrai si l’arbre a le même état que l’arbre B

Implémentation: esquisse des principaux algorithmes

- Recherche d'un élément (appartenance de l'élément à l'arbre):

  La recherche dans un arbre binaire de recherche procède par dichotomie : si l'élément cherché est inférieur à l'élément racine on cherche dans le sous-arbre de gauche, s'il est supérieur on cherche dans celui de droite et s'il est égal on a trouvé.

- Insertion d'un nouvel élément:

  On procède comme pour la recherche mais au cas où l'élément n'est pas trouvé on crée un nouveau noeud pour le stocker.

Implémentation: Suppression d'un élément

- L'opération de suppression est un peu plus problématique. En particulier si l'élément à supprimer ne se trouve pas dans une feuille de l'arbre. Dans ce dernier cas il faut restructurer l'arbre. L'aglorithme procède de la manière suivante:

(1) Chercher le noeud qui contient l'élément à supprimer

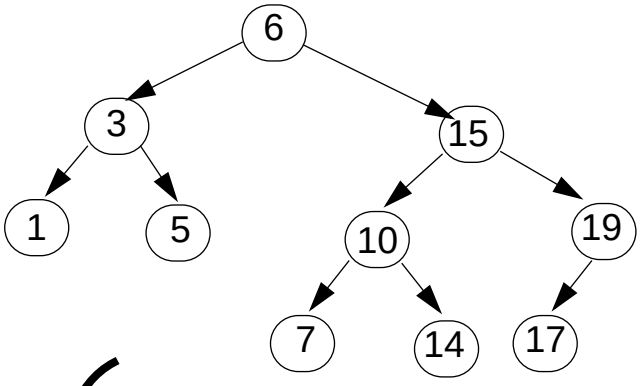
(2) • Si ce noeud est une feuille, on l'enlève de l'arbre.

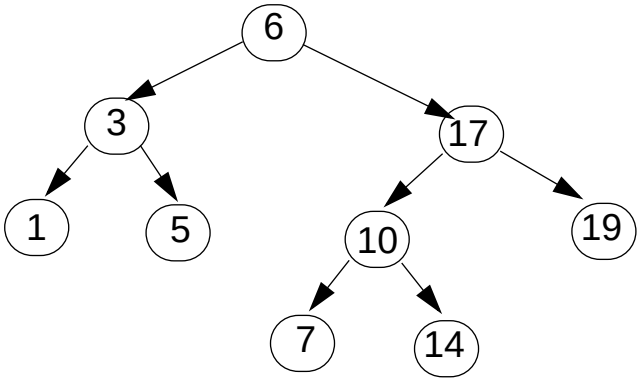
• Si ce noeud n'a qu'un seul descendant direct, on le remplace par ce descendant.

• Si le noeud a deux descendants directs, il faut remplacer le noeud a supprimer par le noeud qui contient l'élément le plus proche. Deux possibiltés: (i) remplacer par le noeud qui se trouve le plus à droite dans le sous-arbre gauche ou (ii) remplacer par le noeud qui se trouve le plus à gauche du sous-arbre droit du noeud à supprimer.

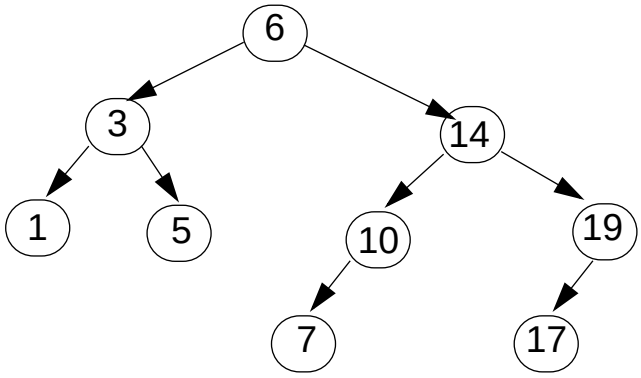
Illustration de la suppression

Exemple: suppression de “15”





ou bien:



Implémentation de l’arbre de recherche binaire en Oberon

```
MODULE ArbreRechBin;

CONST EnOrdre*=1;
      PreOrdre*=2;
      PostOrdre*=3;

TYPE TypeCle* = INTEGER; (* ou n'importe quel autre type *)
     Noeud = POINTER TO RECORD
                 cle : TypeCle;
                 gauche, droite : Noeud;
               END;

     Arbre* = RECORD
                racine : Noeud;
              END;

(* constructeurs *)
PROCEDURE (VAR arbre: Arbre)Inserer*(el : TypeCle),NEW;

  PROCEDURE InsererNoeud(VAR noeud: Noeud);
  BEGIN
     IF noeud = NIL THEN
        NEW(noeud);
        noeud.cle:=el; noeud.gauche:=NIL;
        noeud.droite:=NIL;
     ELSIF el < noeud.cle THEN
        InsererNoeud(noeud.gauche);
     ELSE
        InsererNoeud(noeud.droite)
     END;
  END InsererNoeud;
BEGIN
    InsererNoeud(arbre.racine);
END Inserer;
```

Implémentation arbre de recherche (suite)

```
PROCEDURE (VAR arbre: Arbre)Supprimer*(el: TypeCle;
        VAR fait: BOOLEAN),NEW;

  PROCEDURE SupprimerNoeud(VAR noeud: Noeud);
    PROCEDURE Remplacer(VAR nsa: Noeud);
    (* Remplacer par le noeud le plus à gauche du sous-
       arbre droit *)
    BEGIN
        IF nsa.gauche # NIL THEN Remplacer(nsa.gauche);
        ELSE noeud.cle:=nsa.cle; nsa:=nsa.droite;
        END;
    END Remplacer;
  BEGIN
   IF noeud = NIL THEN fait:=FALSE (* l'élément n'a pas
                          été trouvé dans l'arbre *)
   ELSIF el < noeud.cle THEN SupprimerNoeud(noeud.gauche);
   ELSIF el > noeud.cle THEN SupprimerNoeud(noeud.droite);
   ELSE (* el = noeud.cle -> c'est le noeud qu'il faut
         supprimer *)
     IF noeud.gauche = NIL THEN noeud:=noeud.droite;
     ELSIF noeud.droite = NIL THEN noeud:=noeud.gauche;
     ELSE Remplacer(noeud.droite);
     END;
     fait:=TRUE;
   END;
  END SupprimerNoeud;

BEGIN
   SupprimerNoeud(arbre.racine)
END Supprimer;

PROCEDURE (VAR arbre: Arbre)InitArbre*(),NEW;
BEGIN
    arbre.racine:=NIL;
END InitArbre;
```

Implémentation arbre de recherche (suite)

```
(* sélecteurs *)
PROCEDURE(IN arbre:Arbre)Rechercher*(el:TypeCle):BOOLEAN,NEW;

  PROCEDURE RechercherNoeud(VAR noeud: Noeud):BOOLEAN;
  BEGIN
     IF noeud = NIL THEN RETURN FALSE;
     ELSIF el = noeud.cle THEN
        RETURN TRUE;
     ELSIF el < noeud.cle THEN
        RETURN RechercherNoeud(noeud.gauche);
     ELSE
        RETURN RechercherNoeud(noeud.droite);
     END;
 END RechercherNoeud;

BEGIN
    RETURN RechercherNoeud(arbre.racine);
END Rechercher;

PROCEDURE (IN arbre: Arbre)EstVide*(): BOOLEAN, NEW;
BEGIN
        RETURN arbre.racine = NIL;
END EstVide;

(* Itérateurs *)
PROCEDURE (IN arbre: Arbre)Iterer*(ordre: INTEGER;
          Traiter : PROCEDURE(el: TypeCle)),NEW;

  PROCEDURE ParcourirEnOrdre(noeud: Noeud);
  BEGIN
      IF noeud # NIL THEN
        ParcourirEnOrdre(noeud.gauche);
        Traiter(noeud.cle);
        ParcourirEnOrdre(noeud.droite);END;
   END ParcourirEnOrdre;
```

Implémentation arbre de recherche (suite)

```
PROCEDURE ParcourirPreOrdre(noeud: Noeud);
  BEGIN
     IF noeud # NIL THEN
       Traiter(noeud.cle);
       ParcourirPreOrdre(noeud.gauche);
       ParcourirPreOrdre(noeud.droite);
     END;
  END ParcourirPreOrdre;

  PROCEDURE ParcourirPostOrdre(noeud: Noeud);
  BEGIN
     IF noeud # NIL THEN
       ParcourirPostOrdre(noeud.gauche);
       ParcourirPostOrdre(noeud.droite);
       Traiter(noeud.cle);
     END;
  END ParcourirPostOrdre;

BEGIN
   CASE ordre OF
       EnOrdre   : ParcourirEnOrdre(arbre.racine);
     | PreOrdre  : ParcourirPreOrdre(arbre.racine);
     | PostOrdre : ParcourirPostOrdre(arbre.racine);
     ELSE (* erreur *);
   END;
END Iterer;

END ArbreRechBin.
```

Exemple d’utilisation de l’arbre de recherche binaire par un module client

L’exemple consiste à lire une suite de nombres entiers, à les stocker dans l’arbre de recherche, et à les imprimer en utilisant l’itérateur (i) avec un parcours de l’arbre “en ordre” (ce qui aura pour effet d’afficher les nombres triés), (ii) avec un parcours en “pré-ordre” et (iii) avec un parcours en “post-ordre”.

```
MODULE TestArbreRechBin;

IMPORT IO, StdLog, ArbreRechBin;

VAR arbre: ArbreRechBin.Arbre;

PROCEDURE Afficher(elem: ArbreRechBin.TypeCle);
BEGIN
   StdLog.Int(elem,4);
END Afficher;

PROCEDURE Test*;
VAR elem : ArbreRechBin.TypeCle;
BEGIN
  arbre.Vider;
  IO.OpenIn("nombres.txt");
  LOOP
    IO.ReadInt(elem);
    IF IO.Eot() THEN EXIT END;
    StdLog.Int(elem,4);
    arbre.Inserer(elem);
  END;
  StdLog.Ln;
  StdLog.String("Arbre affiché 'en ordre': "); StdLog.Ln;
  arbre.Iterer(ArbreRechBin.EnOrdre, Afficher); StdLog.Ln;
```

```
   StdLog.String("Arbre affiché en 'pré-ordre': "); StdLog.Ln;
   arbre.Iterer(ArbreRechBin.PreOrdre, Afficher); StdLog.Ln;
   StdLog.String("Arbre affiché en 'post-ordre': "); StdLog.Ln;
   arbre.Iterer(ArbreRechBin.PostOrdre, Afficher); StdLog.Ln;
END Test;

END TestArbreRechBin.Test
```

Avec les nombres13 34 5 67 78 55 23 11 999 6 7 8, l’exécution du programme produit:

```
Fichier ouvert en lecture
  13  34   5  67  78  55  23  11 999   6   7   8
Arbre affiché 'en ordre':
   5   6   7   8  11  13  23  34  55  67  78 999
Arbre affiché en 'pré-ordre':
  13   5  11   6   7   8  34  23  67  55  78 999
Arbre affiché en 'post-ordre':
   8   7   6  11   5  23  55 999  78  67  34  13
```

Implémentation d'un arbre de recherche générique

- Une structure de données est dite générique si elle peut être utilisée pour différents types d'élément
- L'idée est que les algorithmes d'insertion ou de suppression d'un objet dans une structure de donnée, p.e. un arbre, sont indépendants du fait que l'arbre contienne des nombres, des chaînes de caractères ou encore des objets plus complexes.
- Intérêt: écrire une seule implémentation de l'arbre qui pourra par la suite être utilisé pour stocker des nombres, des chaînes caractères etc
- On aimerait pouvoir écrire qqch du genre

  TYPE Arbre* = ... (* structure *) ... **OF T**

  puis, par exemple, l'utiliser dans le module client

  TYPE String = POINTER TO ARRAY OF CHAR
  VAR a : ArbreRechBin.Arbre **OF String**

- Cette construction existe en Ada et Eiffel !
- ... mais pas en Oberon, ... ni en Java

-> nous allons simuler la généricité en utilisant l'extension des classes

Simulation de la généricité en utilsant les classes

- Principe: écrire une première classe (fournisseur) qui implémente les algorithmes pour gérer un arbre d'objets généraux (abstraits); écrire une deuxième classe (client) qui importe la première et remplacer ces objets abstraits par des objets concrets, nombres, chaînes de caractères, etc.
- Comment procéder ? (quels changements par rapport à un arbre de recherche non générique)

  1) Enlever du type *Noeud* le champ contenant l'élément à stocker (à ce stade on ne connaît pas encore son type) et ne garder que les champs *gauche* et *droite* destinés à faire les liens entre les noeuds de l'arbre. Le champ de l'élément à stocker sera défini par extension du type "Noeud" dans le module client.

  2) Qualifier le type *Noeud* avec *ABSTRACT* pour permettre son extension dans le module client et pour indiquer qu'il ne peut pas être instancié (puisqu'il ne contient pas l'élément à stocker). Il doit bien entendu aussi être exporté. On obtient la déclaration:

```
TYPE Noeud* =  POINTER TO ABSTRACT RECORD
                     gauche, droite : Noeud;
               END;
```

Simulation de la généricité (suite)

3) **Point crucial**: pour écrire les algorithmes de l'arbre de recherche nous avons besoin de comparer les éléments de l'arbre entre eux. Mais, à ce stade, nous ne connaissons pas leur type -> nous devons faire l'hypothèse qu'il est possible de les comparer. En Oberon, nous allons exprimer cette hypothèse avec la déclaration de trois méthodes, resp. pour <, > et =

```
PROCEDURE (noeud1: Noeud)Lt*(noeud2: Noeud): BOOLEAN,
                                        NEW, ABSTRACT;

PROCEDURE (noeud1: Noeud)Gt*(noeud2: Noeud): BOOLEAN,
                                        NEW, ABSTRACT;

PROCEDURE (noeud1: Noeud)Eq*(noeud2: Noeud): BOOLEAN,
                                        NEW, ABSTRACT;
```

• Seule la *signature* (nom et paramètres) de la méthode de comparaison de deux éléments est définie; il n'y a pas de corps de méthode. Il sera décrit (*implémenté*) dans le module client en fonction des objets que l'on va stocker dans l'abre. Dans ce cas, la méthode est marquée *abstraite* (ABSTRACT). Mais ce sera suffisant pouvoir exprimer des comparaisons, p.e.

```
... ELSIF el.Lt(noeud) THEN ...
```

4) La création des noeuds (NEW(noeud)) ne se fait plus dans la méthode "Inserer" mais dans le module client. Le noeud ainsi créé est passé comme paramètre à "Inserer".

Implémentation de l’arbre de recherche générique

```
MODULE ArbreRechBinGen;

CONST EnOrdre*=1;
      PreOrdre*=2;
      PostOrdre*=3;

TYPE
    Noeud* = POINTER TO ABSTRACT RECORD
                        gauche, droite : Noeud;
                END;

    Arbre* = RECORD
                        racine : Noeud;
                END;

PROCEDURE (noeud1: Noeud)Lt*(noeud2: Noeud): BOOLEAN, NEW,
ABSTRACT;

PROCEDURE (noeud1: Noeud)Gt*(noeud2: Noeud): BOOLEAN, NEW,
ABSTRACT;

PROCEDURE (noeud1: Noeud)Eq*(noeud2: Noeud): BOOLEAN, NEW,
ABSTRACT;

(* constructeurs / modifieurs *)
PROCEDURE (VAR arbre: Arbre)Inserer*(el : Noeud),NEW;

  PROCEDURE InsererNoeud(VAR noeud: Noeud);
  BEGIN
     IF noeud = NIL THEN
                noeud:=el;
                noeud.gauche:=NIL;
                noeud.droite:=NIL;
     ELSIF el.Lt(noeud) THEN
        InsererNoeud(noeud.gauche);
```

```
      ELSE
        InsererNoeud(noeud.droite)
      END;
  END InsererNoeud;

BEGIN
        ASSERT(el # NIL);
        InsererNoeud(arbre.racine);
END Inserer;

PROCEDURE (VAR arbre: Arbre)Supprimer*(el: Noeud;
         VAR fait: BOOLEAN),NEW;

   PROCEDURE SupprimerNoeud(VAR noeud: Noeud);
     PROCEDURE Remplacer(VAR nsa: Noeud);
     (* Remplacer par le noeud le plus à gauche du sous-
        arbre droit *)
     BEGIN
          IF nsa.gauche # NIL THEN Remplacer(nsa.gauche);
          ELSE nsa:=nsa.droite;
          END;
     END Remplacer;
   BEGIN
    IF noeud = NIL THEN fait:=FALSE (* l'élément n'a pas
                          été trouvé dans l'arbre *)
    ELSIF el.Lt(noeud) THEN SupprimerNoeud(noeud.gauche);
    ELSIF el.Gt( noeud) THEN SupprimerNoeud(noeud.droite);
    ELSE (* el = noeud -> c'est le noeud qu'il faut supprimer *)
      IF noeud.gauche = NIL THEN noeud:=noeud.droite;
      ELSIF noeud.droite = NIL THEN noeud:=noeud.gauche;
      ELSE Remplacer(noeud.droite);
      END;
      fait:=TRUE;
    END;
   END SupprimerNoeud;

BEGIN
    SupprimerNoeud(arbre.racine)
```

```
END Supprimer;

PROCEDURE (VAR arbre: Arbre)InitArbre*(),NEW;
BEGIN
    arbre.racine:=NIL;
END InitArbre;

(* sélecteurs *)
PROCEDURE (IN arbre:Arbre)Rechercher*(el:Noeud):BOOLEAN,NEW;

  PROCEDURE RechercherNoeud(noeud: Noeud):BOOLEAN;
  BEGIN
     IF noeud = NIL THEN RETURN FALSE;
     ELSIF el.Eq(noeud) THEN
        RETURN TRUE;
     ELSIF el.Lt(noeud) THEN
        RETURN RechercherNoeud(noeud.gauche);
     ELSE
        RETURN RechercherNoeud(noeud.droite);
     END;
 END RechercherNoeud;

BEGIN
   RETURN RechercherNoeud(arbre.racine);
END Rechercher;

PROCEDURE  (IN arbre:Arbre)EstVide*():BOOLEAN,NEW;
BEGIN
        RETURN arbre.racine = NIL;
END EstVide;

(* Itérateurs *)
PROCEDURE (VAR arbre: Arbre)Iterer*(ordre: SHORTINT;
          Traiter : PROCEDURE(el: Noeud)),NEW;

  PROCEDURE ParcourirEnOrdre(noeud: Noeud);
  BEGIN
     IF noeud # NIL THEN
```

```
        ParcourirEnOrdre(noeud.gauche);
        Traiter(noeud);
        ParcourirEnOrdre(noeud.droite);
      END;
   END ParcourirEnOrdre;

  PROCEDURE ParcourirPreOrdre(noeud: Noeud);
  BEGIN
     IF noeud # NIL THEN
        Traiter(noeud);
        ParcourirPreOrdre(noeud.gauche);
        ParcourirPreOrdre(noeud.droite);
     END;
  END ParcourirPreOrdre;

  PROCEDURE ParcourirPostOrdre(noeud: Noeud);
  BEGIN
     IF noeud # NIL THEN
        ParcourirPostOrdre(noeud.gauche);
        ParcourirPostOrdre(noeud.droite);
        Traiter(noeud);
     END;
  END ParcourirPostOrdre;

BEGIN
   CASE ordre OF
       EnOrdre   : ParcourirEnOrdre(arbre.racine);
     | PreOrdre  : ParcourirPreOrdre(arbre.racine);
     | PostOrdre : ParcourirPostOrdre(arbre.racine);
     ELSE (* erreur *);
   END;
END Iterer;

END ArbreRechBinGen.
```

Arbre de recherche générique “côté client”

Comment utiliser une structure de données générique ?

1) Ajouter (par extension) au type *Noeud* générique le champ nécessaire à stocker l’élément. Par exemple, si l’on veut stocker des mots

```
TYPE
  String = POINTER TO ARRAY OF CHAR;
  NoeudMot = POINTER TO RECORD (ArbreRechBinGen.Noeud)
     mot: String;
  END;
```

2) (re)définir les méthodes abstraites de comparaison Lt, Gt et Eq en définissant leur corps. **Attention**: le paramètre (*noeud2*) doit être rigoureusement du même type que celui de la méthode abstraite, càd de type *Noeud* et non *NoeudMot*. Utiliser la garde de type *noeud2(NoeudMot)* pour accéder au champ *mot* .

```
PROCEDURE (noeud1: NoeudMot)Lt(noeud2:
        ArbreRechBinGen.Noeud): BOOLEAN;
BEGIN RETURN noeud1.mot$ < noeud2(NoeudMot).mot$;
END Lt;

PROCEDURE (noeud1: NoeudMot)Gt(noeud2:
        ArbreRechBinGen.Noeud): BOOLEAN;
BEGIN RETURN noeud1.mot$ > noeud2(NoeudMot).mot$;
END Gt;

PROCEDURE (noeud1: NoeudMot)Eq(noeud2:
        ArbreRechBinGen.Noeud): BOOLEAN;
BEGIN RETURN noeud1.mot$ = noeud2(NoeudMot).mot$;
END Eq;
```

Utilisation d’un arbre générique (suite)

3) Les mots seront insérés dans l’arbre (par le module client) de la manière suivante

```
VAR  arbre: ArbreRechBinGen.Arbre;
     el: NoeudMot;
...
NEW(el); el.InitNoeudMot(str);
arbre.Inserer(el);
```

Exemple d’utilisation

L’exemple d’utilisation suivant est semblable à celui de l’utilisation de l’arbre de recherche non générique (voir p 5-22) mais les éléments stockés dans l’arbre sont des mots (chaînes de caractères)

```
MODULE ArbreRechBinGenTest;

IMPORT IO, StdLog, ArbreRechBinGen;

TYPE String = POINTER TO ARRAY OF CHAR;

     NoeudMot = POINTER TO RECORD (ArbreRechBinGen.Noeud)
          mot: String;
     END;

VAR arbre: ArbreRechBinGen.Arbre;

PROCEDURE (noeud1: NoeudMot)Lt(noeud2:
ArbreRechBinGen.Noeud): BOOLEAN;
BEGIN
     RETURN noeud1.mot$ < noeud2(NoeudMot).mot$;
END Lt;
```

```
PROCEDURE (noeud1: NoeudMot)Gt(noeud2:
ArbreRechBinGen.Noeud): BOOLEAN;
BEGIN
        RETURN noeud1.mot$ > noeud2(NoeudMot).mot$;
END Gt;

PROCEDURE (noeud1: NoeudMot)Eq(noeud2:
ArbreRechBinGen.Noeud): BOOLEAN;
BEGIN
        RETURN noeud1.mot$ = noeud2(NoeudMot).mot$;
END Eq;

PROCEDURE (noeud: NoeudMot)InitNoeudMot(str: ARRAY OF CHAR),
NEW;
BEGIN
        NEW(noeud.mot,LEN(str));
        noeud.mot^:=str$;
END InitNoeudMot;

PROCEDURE Afficher(elem:ArbreRechBinGen.Noeud );
BEGIN
   StdLog.String(elem(NoeudMot).mot$); StdLog.String("  ");
END Afficher;

PROCEDURE Test*;
VAR str: ARRAY 80 OF CHAR;
        el: NoeudMot;
BEGIN
        IO.OpenRead(FALSE);
        arbre.InitArbre;
        LOOP
                IO.ReadString(str);
                IF IO.Eot() THEN EXIT END;
                StdLog.String(str$ + "  ");
                NEW(el); el.InitNoeudMot(str);
                arbre.Inserer(el);
        END;
```

```
        StdLog.Ln;
        StdLog.String("Arbre affiché 'en ordre': "); StdLog.Ln;
        arbre.Iterer(ArbreRechBinGen.EnOrdre, Afficher); StdLog.Ln;
        StdLog.String("Arbre affiché en 'pré-ordre': "); StdLog.Ln;
        arbre.Iterer(ArbreRechBinGen.PreOrdre, Afficher); StdLog.Ln;
        StdLog.String("Arbre affiché en 'post-ordre': "); StdLog.Ln;
        arbre.Iterer(ArbreRechBinGen.PostOrdre, Afficher); StdLog.Ln;
END Test;

END ArbreRechBinGenTest.
```

Si le fichier d'entrée contient la phrase "buvons un coup ma serpette est perdue mais le manche m'est revenu", l'exécution du programme produit:

```
Fichier ouvert en lecture
buvons un coup ma serpette est perdue mais le manche m'est revenu
Arbre affiché 'en ordre':
buvons coup est le m'est ma mais manche perdue revenu serpette un
Arbre affiché en 'pré-ordre':
buvons un coup ma est le m'est serpette perdue mais manche revenu
Arbre affiché en 'post-ordre':
m'est le est manche mais revenu perdue serpette ma coup un bouvons
```

Structures de données génériques: bilan

- Avantages de l'arbre de recherche binaire générique
  - La structure "Arbre" peut stocker n'importe quel objet dont le type est dérivé du type "Noeud"
  - "Arbre" peut être réutilisé sans qu'il soit nécessaire de recompiler le module où il est défini
  - Il est même possible de construire des arbres de recherche hétérogènes, mais à condition que les éléments possèdent une relation d'ordre entre eux.
- En résumé:

  (1) Souvent, le module générique encapsule les algorithmes qui opèrent sur des structures de données dynamiques. De telles structures de données sont cachées et représentent des types abstraits.

  (2) Le module générique contient également la méthode d'initialisation de la structure de données générique (p.e. InitArbre)

  (3) Le module client ajoute des champs spécifiques à l'application au type "noeud" du module générique.

  (4) C'est au module client que revient la tâche de générer les instances des données puis de les ajouter à la structure de données en invoquant ses méthodes.

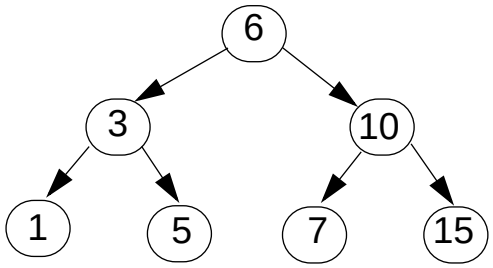
  (5) Lorsque des données sont retournées par la structure, le recours à une garde de type est requise.

Problème de l'équilibre dans les structures d'arbre de recherche

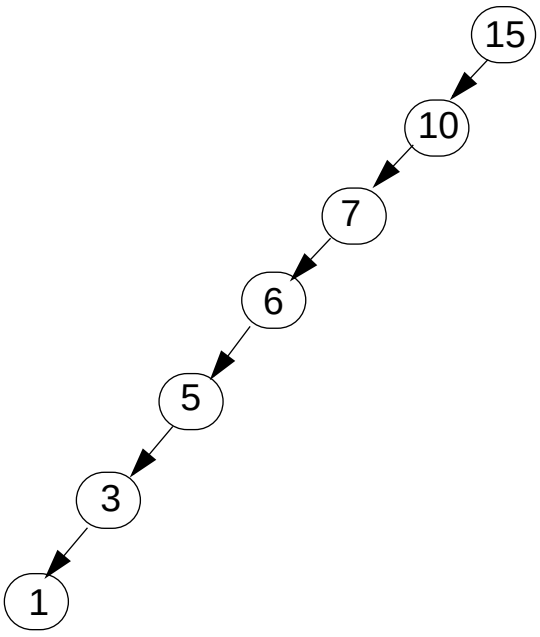
La forme de l'arbre dépend de l'ordre dans lequel on introduit les données.

Ex. avec arbre de recherche d'ordre 2 (arbre binaire):

- suite de nombres 6,3,1,10,7,5,15 –> arbre équilibré



suite de nombres 15,10,7,6,5,3,1–> arbre déséquilibré



Problème de l'équilibre (suite)

Conséquences

Complexité de recherche:

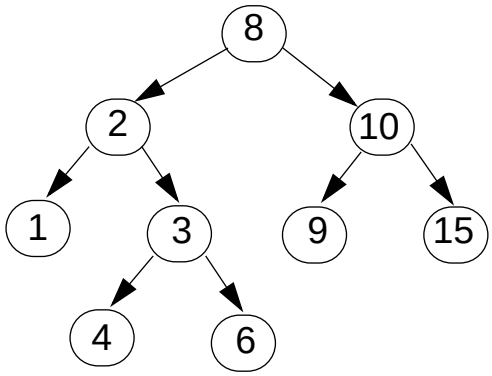
- $O(\log_2(n))$ si arbre équilibré (ex: max. 3 comparaisons)
- $O(n)$ si arbre déséquilibré (ex: max. 7 comparaisons); en fait, on se retrouve dans les cas de la liste chaînée.

Solutions:

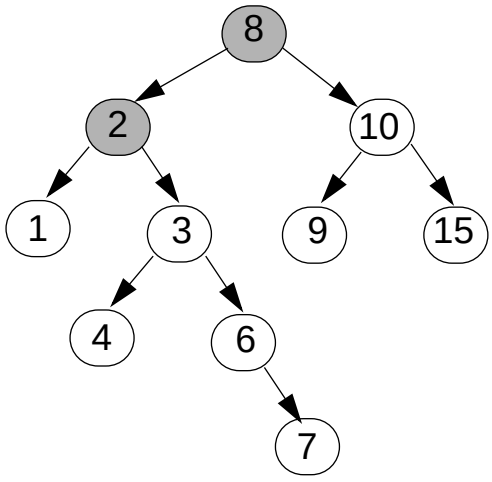
- algorithmes de rééquilibrage (p.e. arbres AVL)
- B-arbres

Arbres AVL

- Inventeurs: Adelson - Velskii et Landis (AVL)
- Définition: “un arbre AVL est un arbre de recherche binaire avec la propriété d’équilibre suivante: pour chaque noeud, la hauteur du sous-arbre gauche et la hauteur du sous-arbre droit ne peuvent différer qu’au maximum de 1.”
- Rééquilibrage après chaque insertion ou suppression
- Exemple d’arbre AVL:



- Contre exemple(en gris les noeuds non équilibrés):



Rééquilibrage

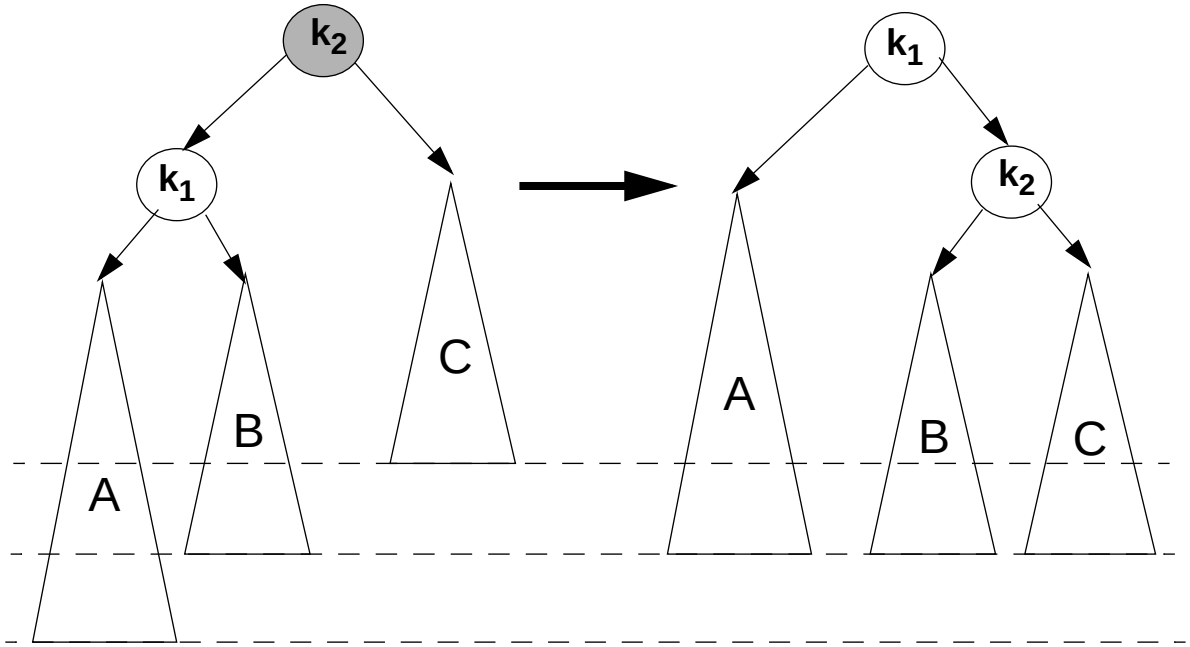
Le rééquilibrage est nécessaire lorsqu'à la suite d'une insertion (ou suppression) un noeud X n'est plus équlibré, càd différence des hauteurs = 2

4 cas à examiner:

- (1) insertion dans le sous-arbre gauche de l'enfant gauche du noeud X
- (2) insertion dans le sous-arbre droit de l'enfant gauche du noeud X
- (3) insertion dans le sous-arbre gauche de l'enfant droit du noeud X
- (4) insertion dans le sous-arbre droit de l'enfant droit du noeud X

Remarque: les cas 1 et 4 sont symétriques; les cas 2 et 3 sont symétriques

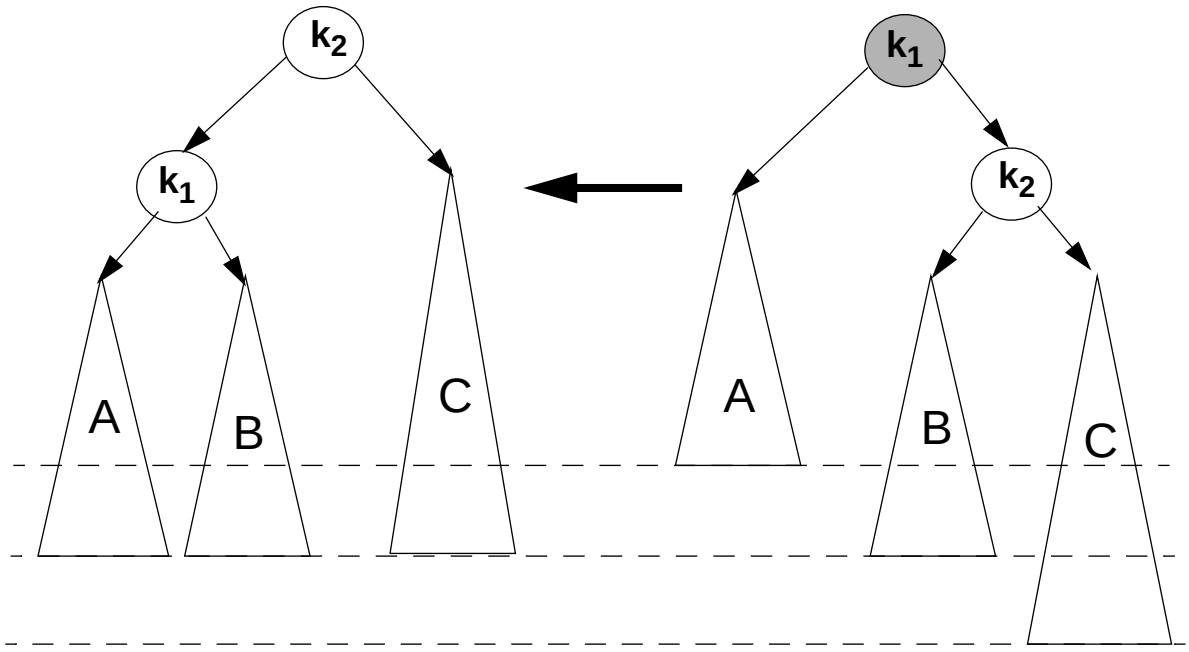
Rotation simple pour réparer le cas n°1:



Esquisse du code Oberon:

```
k2.gauche:=k1.droite;
k1.droite:=k2;
```

Rotation simple pour réparer le cas n°4 :



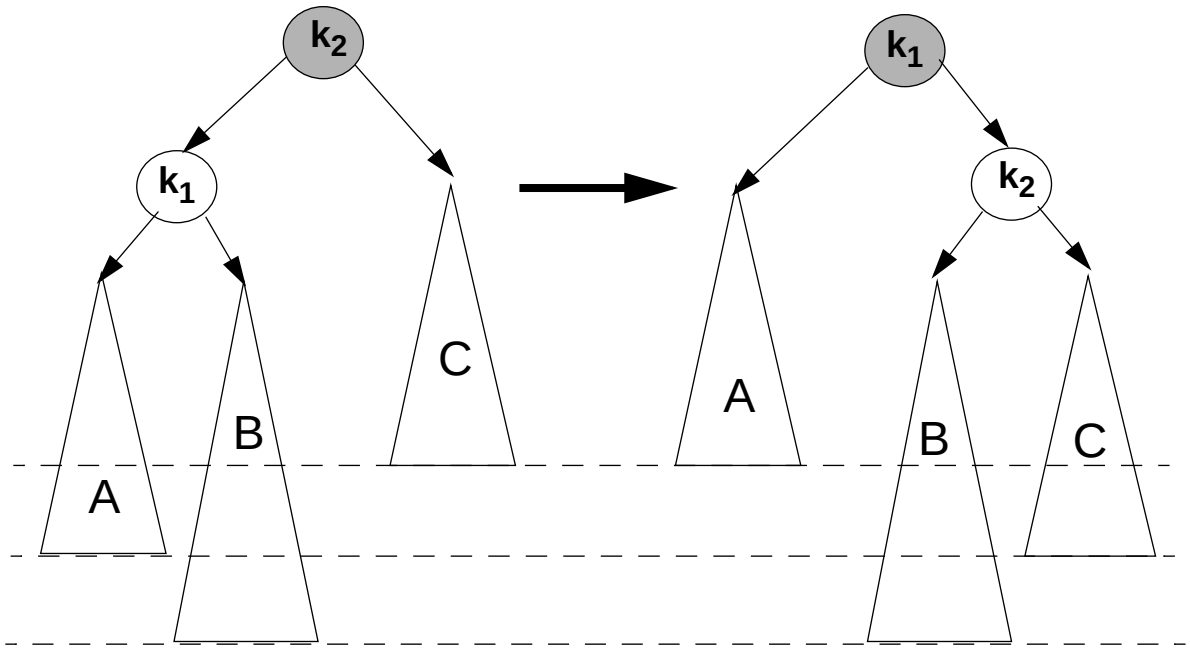
Comme annocé, le traitement du cas n°4 est le symétrique du cas n°1:

Esquisse du code Oberon:

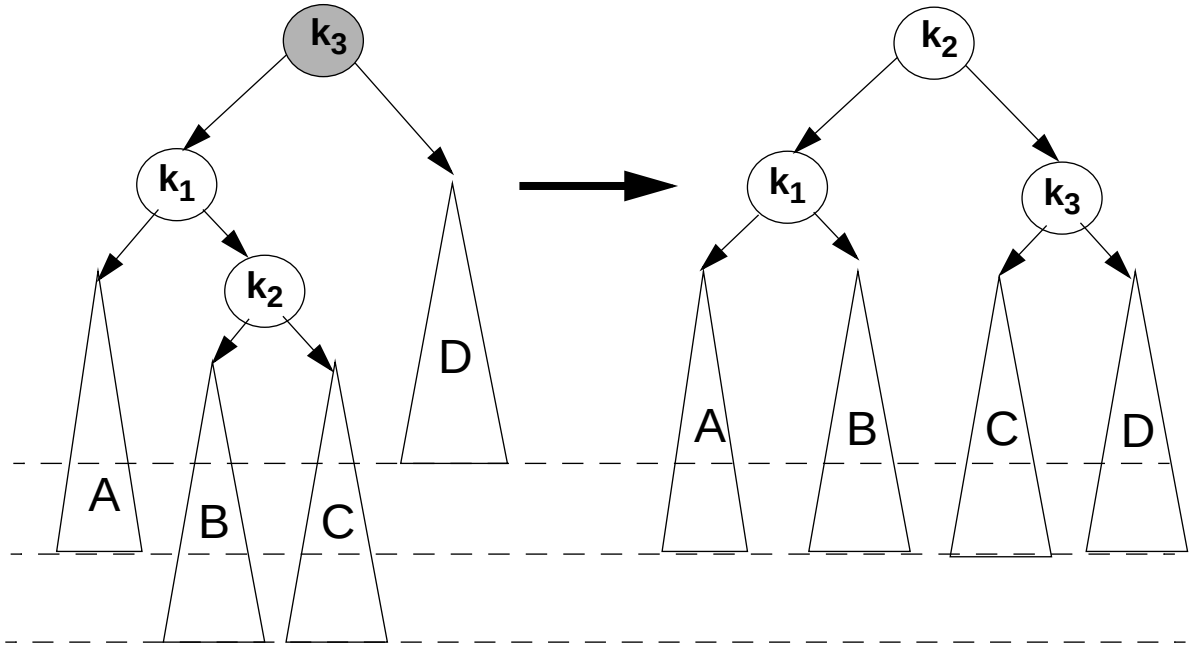
```
k1.droite:=k2.gauche;
k2.gauche:=k1;
```

Réparation pour le cas 2:

La rotation simple ne marche pas pour le cas 2 !



Rotation double pour réparer le cas n°2



En fait, il s'agit de deux rotations simples:

- rotation entre le fils de k_3 (k_1) et le petit fils de k_3 (k_2)
- rotation entre $k_3$et son nouveau fils k_2

Esquisse du code Oberon:

```
(* 1ère rotation : *)
      k1.droite:=k2.gauche;
      k2.gauche:=k1;
(2ème rotation : *)
      k3.gauche:=k2.droite;
      k2.droite:=k3;
```

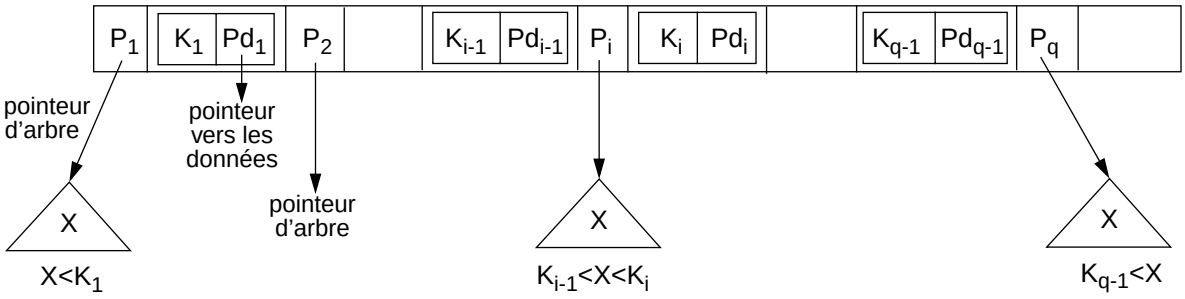
Les B-arbres

Arbre de recherche avec propriétés supplémentaires:

- équilibré, bien rempli(noeuds au moins à moitié pleins)

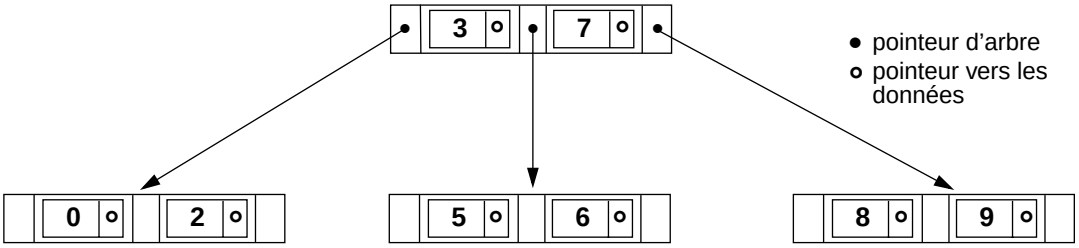
B-arbre d'ordre p

- chaque noeud interne a la forme: $<P_1,<K_1,Pd_1>P_2,<K_2,Pd_2>,...,P_{q-1},<K_{q-1}Pd_{q-1}>,P_q>$ avec $q \leq p$ ($Pd_1,Pd_2,...,Pd_{q-1}$ sont des pointeurs vers des enregistrement de données)



- La racine a au moins 2 descendants - sauf si c'est une feuille
- Chaque noeud intérieur a au max. p descendants et au min. [p/2] descendants
- Toutes les feuilles apparaissent au même niveau

Exemple: un B-arbre d'ordre 3



Les B-arbres: performances

Complexité de la recherche: O($\log_p(n)$)

Remarque:

On réserve un bloc disque pour le stockage de chaque noeud.

Calcul du nombre de points d'entrée. Exemple:

Hypothèse:

- longueur de la clé: 9 octets
- taille bloc disque: 512 octets
- taille pointeur vers un bloc: 6 octets

–> ordre du B-arbre: $(p * 6) + ((p-1) * (6+9)) \leq 512$
–> $p = 25$

Hypothèse:

- les noeuds sont remplis à 69% (chiffre obtenu par simulation) –> remplissage moyen 17 ptr / noeuds

–> nombre d'entrées en fonction du niveau de l'arbre:

racine:	1 noeud	16 entrées	17 pointeurs
niveau 1:	17 noeuds	272 entrées	289 pointeurs
niveau 2:	289 noeuds	4624 entrées	4913 pointeurs
niveau 3:	4913 noeuds	78608 entrées	83521 pointeurs

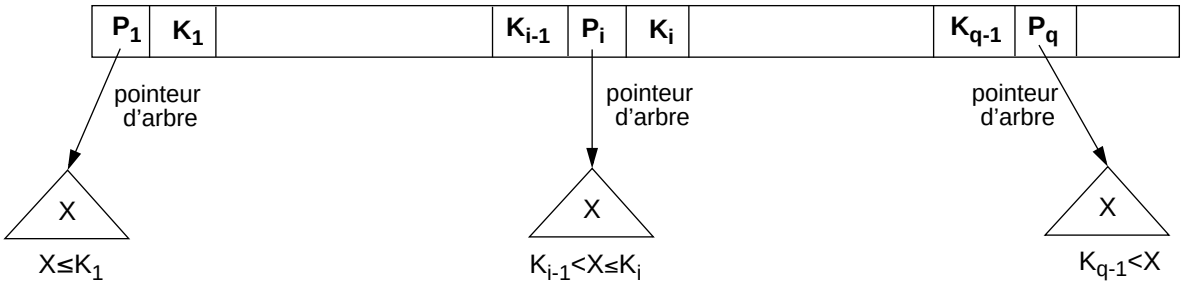
Les B^+-arbres

La plupart des implantations des index de fichiers utilisent une variation du B-arbre appelée B^+-arbre.

Particularités:

- les pointeurs vers les enregistrements de données apparaissent uniquement au niveau des feuilles
- les feuilles sont reliées entre-elles par des pointeurs

Noeud interne:



Noeud feuille :

Les B^+-arbres: performances

Toutes les valeurs de clé apparaissent dans les noeuds feuille de l'arbre

–> parcours séquentiel dans l'ordre des clés très rapide;

–> certaines valeurs de clés sont stockées plusieurs fois;

–> à la suppression il faut également ajuster les blocs supérieurs.

Complexité de la recherche: $O(\log_p(n))$ (idem B-arbre)

Calcul du nombre de points d'entrée. Exemple:

(on prend les mêmes hypothèses que pour les B-arbres)

–> ordre du B^+-arbre: $(p * 6) + ((p-1) * 9) \leq 512$
–> $p = 34$

–> remplissage moyen: 23 pointeurs par noeud

–> nombre d'entrées en fonction du niveau de l'arbre:

racine:	1 noeud	22 entrées	23 pointeurs
niveau 1:	23 noeuds	506 entrées	529 pointeurs
niveau 2:	529 noeuds	11638 entrées	12167 pointeurs
niveau 3:	12167 noeuds	267674 entrées	

(remarque: le niveau 3 correspond au niveau feuille)

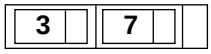
Insertion dans les B^+-arbres

Insérer un enregistrement avec k pour valeur de clé:

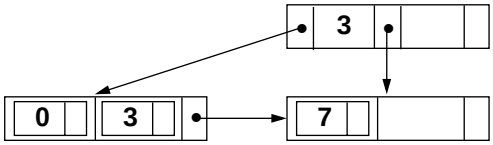
1. En partant de la racine, chercher la feuille f qui devra contenir k
2. insérer k dans f
3. si débordement de f:
   - éclater la feuille f en deux feuilles f' et f" en redistribuant les valeurs de clé de manière $f' \leq k' < f"$ (k' est la valeur médiane de $f' \cup f"$)
   - insérer la valeur médiane de $f' \cup f"$ k' dans le noeud parent
   - si le noeud parent deborde, l'éclater à son tour (les éclatements peuvent éventuellement se propager jusqu'à la racine)

Insertion dans les B$^+$-arbres : exemple

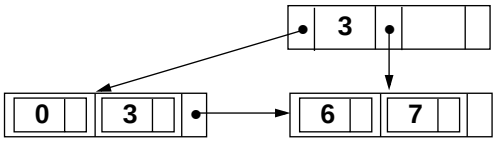
Insertion de 3 et 7



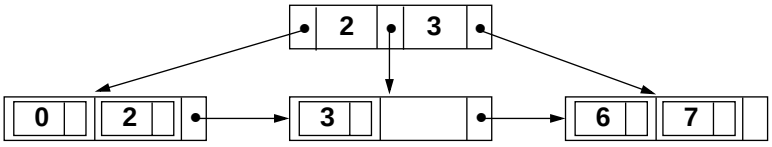
Insertion de 0: débordement, création d'un nouveau niveaux



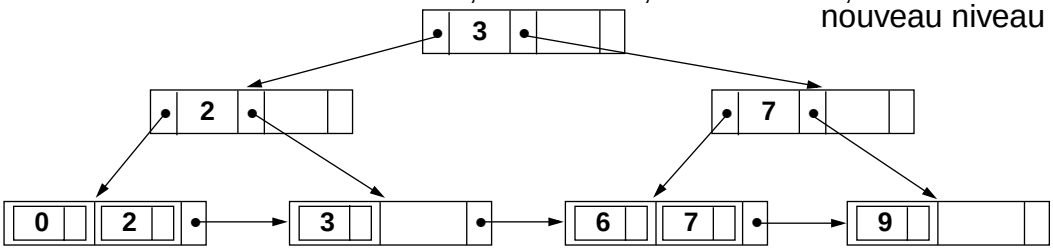
Insertion de 6



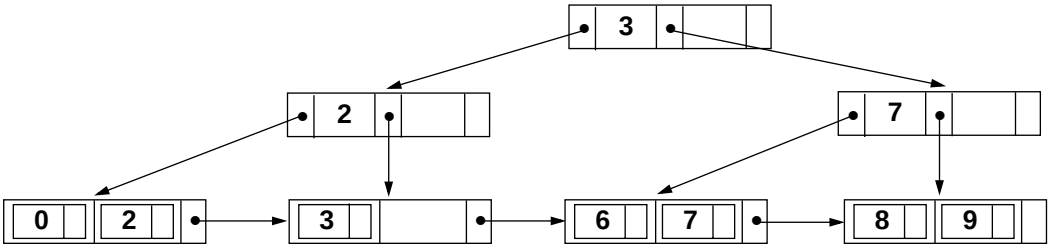
Insertion de 2: débordement, éclatement, redistribution



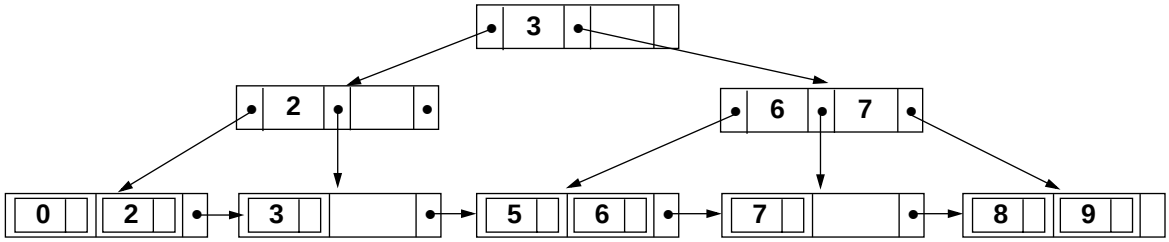
Insertion de 9: débordement, éclatement, redistribution, création d'un nouveau niveau



Insertion de 8



Insertion de 5: débordement, éclatement, redistribution

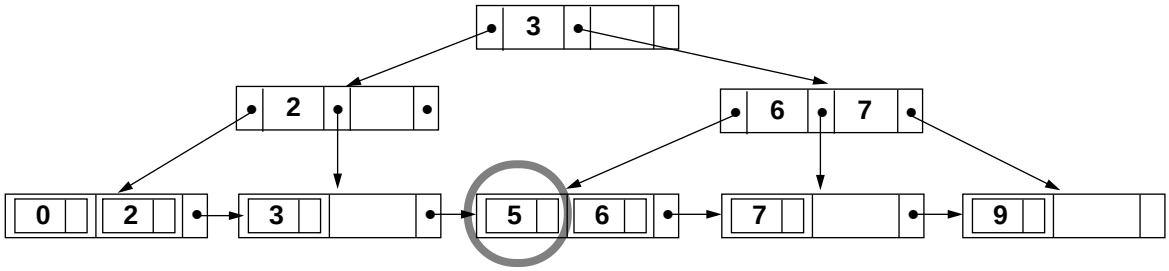


Suppression dans les B^+-arbres

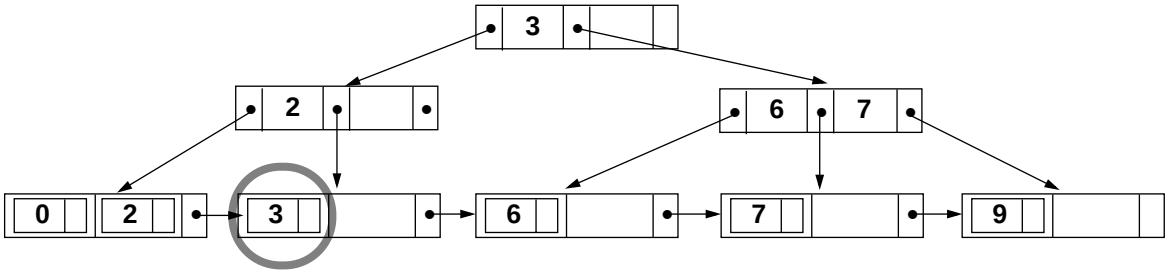
Supprimer un enregistrement avec k pour valeur de clé:

1. En partant de la racine, chercher la feuille f qui contient k
2. supprimer k et le pointeur vers l'enregistrement de données associé
3. supprimer k dans tous les noeuds parents où elle apparaît
4. si sous-occupation de f (i.e. f moins qu'à moitié plein):
   - redistribuer les valeurs de clé sur les feuilles de l'arbre
   - si la redistribution est impossible, fusionner f avec la feuille adjacente
5. si sous-occupation d'un noeud parent :
   - idem feuille
     (les fusions peuvent éventuellement se propager jusqu'à la racine)

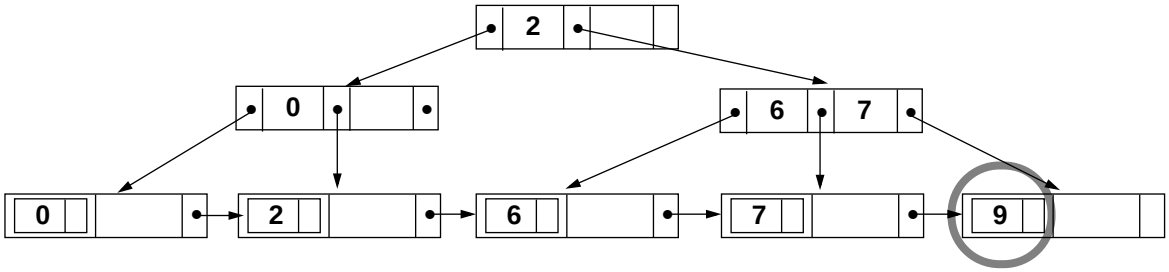
Suppression dans les B$^+$-arbres: exemple



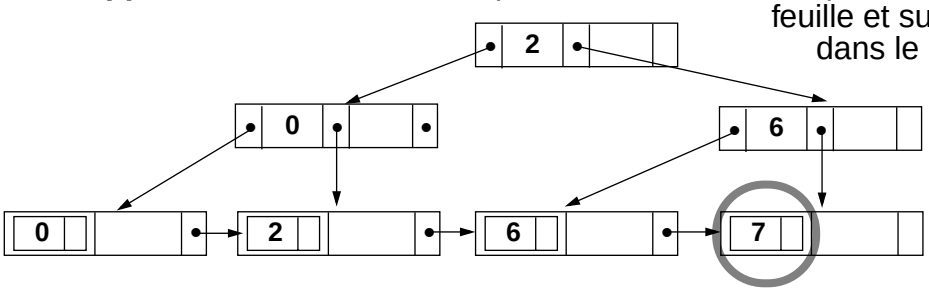
Suppression de 5



Suppression de 3: sous-occupation, redistribution



Suppression de 9: sous-occupation, redistribution impossible, fusion de la feuille et supression de 7 dans le noeud interne



Suppression de 7: sous-occupation, redistribution impossible, fusion de la feuille, fusion de niveaux