Arbres-B (B-trees)
(Arbres 2-3-4 (2-3-4 trees))
Correction

1 Préliminaires

Solution 1.1 **(Arbres : de recherche, B, B+, 2-3-4. . .)**

1. *Arbre (général) de recherche (m-way search tree)* :

   - Chaque nœud (dit k-nœud) contient $k-1$ clés et k fils s'il est interne.
   - Les clés sont en ordre strictement croissant dans un même nœud.
   - Pour chaque clé x, son fils gauche contient les clés strictement inférieures à x, son fils droit les clés strictement supérieures à x.

   Contrairement à la définition d'un arbre général, un arbre de recherche pourra être vide.

2. (a) *Arbre B*[1] *(B-tree)* :

   Définition du Cormen :
   Un B-arbre est un arbre général de recherche possédant les propriétés suivantes :

   - Toutes les feuilles ont la même profondeur.
   - Il existe un majorant et un minorant pour le nombre de clés pouvant être contenues par un nœud. Ces bornes peuvent être exprimées en fonction d'un entier fixé $t \geqslant 2$ appelé le **degré minimal du B-arbre** (ou l'ordre).
     - Tout nœud autre que la racine doit contenir au moins $t-1$ clés. Tout nœud interne autre que la racine possède au moins t fils. Si l'arbre n'est pas vide, la racine doit posséder au moins une clé.
     - Tout nœud peut contenir au plus $2t-1$ clés. Un nœud interne peut donc posséder au plus $2t$ fils. Un nœud est **complet** s'il contient exactement $2t-1$ clés.

   C'est à dire : les nœuds sont des k-nœuds, avec $t \leq k \leq 2 * t$ sauf pour la racine, pour laquelle $2 \leq k \leq 2 * t$.

   (b) L'*arbre B+ (B+ tree)* est une variante de l'arbre B : toutes les données (les valeurs) sont stockées exclusivement dans des feuilles (les nœuds internes ne contenant que les "intervalles" de clés), et celles-ci sont reliées entre elles (une liste chaînée associative).

3. *Arbre 2-3-4 (2-3-4 tree) :*

   C'est un B-arbre de degré minimal (ordre) 2.

1. On utilisera plus souvent B-arbre.

Solution 1.2 **(Implémentation des B-arbres)**

2. *Type de données représentant les B-arbres* :

```
constantes
    t    = /* minimal degree */
types
     /* t_element */
    t_Btree    = ↑ t_node_Btree
    t_vect_cles  = (2*t-1) t_element
    t_vect_fils  = (2*t) t_Btree
    t_nodeBtree = enregistrement
        entier     nbcles
        t_vect_cles  cles
        t_vect_fils  fils
    fin enregistrement t_node_Btree
```

Remarque : les pointeurs vers les k premiers fils sont à NUL pour les "k-feuilles".

Solution 1.3 **(Minimum et maximum)**

2. **Spécifications :**
   La fonction `min_Btree (B)` retourne la clé minimum du B-arbre non vide B.

```
algorithme fonction min_Btree : t_element
    parametres locaux
        t_Btree    B

debut
    tant que B↑.fils[1] <> NUL faire
        B ← B↑.fils[1]
    fin tant que
    retourne B↑.cles[1]
fin algorithme fonction min_Btree
```

Spécifications :
La fonction `max_Btree (B)` retourne la clé maximum du B-arbre non vide B.

```
algorithme fonction max_Btree : t_element
    parametres locaux
        t_Btree    B

debut
    tant que B↑.fils[1] <> NUL faire
        B ← B↑.fils[B↑.nbcles+1]
    fin tant que
    retourne B↑.cles[B↑.nbcles]
fin algorithme fonction max_Btree
```

Solution 1.4 **(Recherche d'un élément – *contrôle nov. 12*)**

Spécifications :

La fonction `search_pos (t_element` x`, t_Btree` B`)` cherche la valeur x dans le nœud racine du B-arbre B non vide. Elle retourne la position de x dans le nœud s'il est présent, sa position "virtuelle" dans le cas contraire.

```
algorithme fonction search_pos : entier
    parametres locaux
        t_element    x
        t_Btree       B

    variables
        entier     g, d, m
debut
        g ← 1
        d ← B↑.nbcles
        m ← (g + d) div 2
        tant que g < = d faire
            si x = B↑.cles[m] alors
                retourne m
            fin si
            si x < B↑.cles[m] alors
               d ← m-1
            sinon
               g ← m+1
            fin si
            m ← (g + d) div 2
        fin tant que
        retourne g
fin algorithme fonction search_pos
```

Spécifications :

La fonction `search_Btree (t_element` x`, t_Btree` B`)` retourne un pointeur vers le nœud contenant la valeur x dans le B-arbre B ou la valeur NUL si x n'est pas présent dans l'arbre.

```
algorithme fonction search_Btree : t_Btree
    parametres locaux
        t_element    x
        t_Btree       B

    variables
        entier     i
debut
    si B = NUL alors
        retourne NUL
    sinon
        i ← search_pos (x, B)
        si B↑.cles[i] = x alors
            retourne B
        sinon
            retourne search_Btree (x, B↑.fils[i])
        fin si
    fin si
fin algorithme fonction search_Btree
```

***Solution 1.5* (Intervalle – *contrôle nov. 12*)**

Spécifications :

La procédure `range` (B, inf, sup) affiche (en ordre croissant) l'ensemble des clés se trouvant dans le B-arbre B comprises dans l'intervalle $[inf; sup]$. Les clés seront séparées par des espaces.

```
algorithme procedure range
   parametres locaux
      t_Btree     B
      t_element   inf, sup

   variables
      entier    i
debut
   si B <> NUL alors
      i ← 1
      tant que (i <= B↑.nbcles) et (B↑.cles[i] < inf) faire
         i ← i + 1
      fin tant que
      si B↑.fils[1] = NUL alors        /* optimization */
         tant que (i <= B↑.nbcles) et (sup >= B↑.cles[i]) faire
            ecrire (B↑.cles[i], " ")
            i ← i + 1
         fin tant que
      sinon
         range (B↑.fils[i], inf, sup)
         tant que (i <= B↑.nbcles) et (sup >= B↑.cles[i]) faire
            ecrire (B↑.cles[i], " ")
            range (B↑.fils[i+1], inf, sup)
            i ← i + 1
         fin tant que
      fin si
   fin si
fin algorithme procedure range
```

2 Insertions – Suppressions

Solution 2.1 **(Inserer un nouvel élément : la méthode classique)**

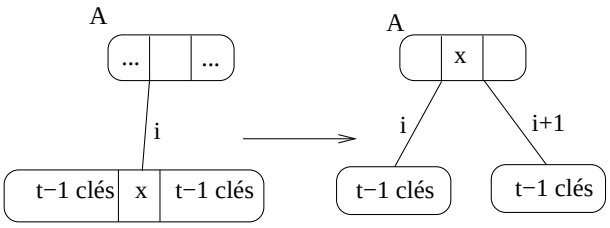

Figure 1 – Éclatement

1. (d) **Spécifications :**
   La procédure `eclate (`B`, `i`)` éclate le fils n°i de l'arbre B (de type `t_Btree`).
   – L'arbre B existe (est non vide) et sa racine n'est pas un $2t$-nœud.
   – Le fils i de B existe et sa racine est un $2t$-nœud.

```
algorithme procedure eclate
    parametres locaux
        t_Btree    B
        entier     i

    variables
        t_Btree   L, R
        entier    j
debut
    L ← B↑.fils[i]    /* left child */
    allouer (R)       /* new right child */

    pour j ← 1 jusqu'a t-1 faire          /* copy keys and children from L to R */
        R↑.fils[j] ← L↑.fils[t+j]
        R↑.cles[j] ← L↑.cles[t+j]
    fin pour
    R↑.fils[t] ← L↑.fils[2*t]
    L↑.nbcles ← t-1
    R↑.nbcles ← t-1

    pour j ← B↑.nbcles+1 decroissant jusqu'a i+1 faire  /* insertion of new key and child in root */
        B↑.fils[j+1] ← B↑.fils[j]
        B↑.cles[j] ← B↑.cles[j-1]
    fin pour
    B↑.cles[i] ← L↑.cles[t]
    B↑.fils[i+1] ← R
    B↑.nbcles ← B↑.nbcles+1
fin algorithme procedure eclate
```

3. **Spécifications :**
   La fonction `insert_rec (x, B)` insère la clé x dans l'arbre B de type `t_Btree`, sauf si celle-ci est déjà présente. L'arbre B n'est pas vide, et sa racine n'est pas un nœud complet (pas un $2t$-nœud).

```
algorithme fonction insert_rec : booleen
   parametres locaux
      t_element    x
      t_Btree      B

   variables
      entier     i, j, m
debut
   i ← search_pos (x, B)                    /* search where to insert x */

   si (i <= B↑.nbcles) et (B↑.cles[i] = x) alors     /* x ∈ node */
      retourne faux
   sinon
      si B↑.fils[1] = NUL alors              /* insertion */
         pour j ← B↑.nbcles decroissant jusqu'a i faire
            B↑.cles[j+1] ← B↑.cles[j]
         fin pour
         B↑.cles[i] ← x
         B↑.nbcles ← B↑.nbcles + 1
         B↑.fils[B↑.nbcles+1] ← NUL
         retourne vrai
      sinon
         si B↑.fils[i]↑.nbcles = 2*t-1 alors
            si B↑.fils[i]↑.cles[t] = x alors
               retourne faux
            fin si
            eclate (B, i)
            si x > B↑.cles[i] alors
               i ← i + 1
            fin si
         fin si
         retourne insert_rec (x, B↑.fils[i])
      fin si
   fin si
fin algorithme fonction insert_rec
```

Spécifications :
La fonction `insertion_Btree (t_element` x`, t_Btree` B`)` insère la clé x dans le B-arbre B sauf si elle est déjà présente. Elle retourne un booléen indiquant si l'insertion a eu lieu.

```
algorithme fonction insertion_Btree : booleen
    parametres locaux
        t_element    x
    parametres globaux
        t_Btree      B

    variables
        entier       i
        t_Btree     R
debut
    si B = NUL alors
        allouer (B)
        B↑.nbcles ← 1
        B↑.cles[1] ← x
        B↑.fils[1] ← NUL
        B↑.fils[2] ← NUL
        retourne vrai
    sinon
        si B↑.nbcles = 2*t-1 alors
            allouer (R)          /* new root */
            R↑.nbcles ← 0
            R↑.fils[1] ← B
            B ← R
            eclate (B, 1)
        fin si
        retourne insert_rec (x, B)
    fin si
fin algorithme fonction insertion_Btree
```

Solution 2.2 **(Suppression d'un élément : à la descente)**

2. **Rotations**

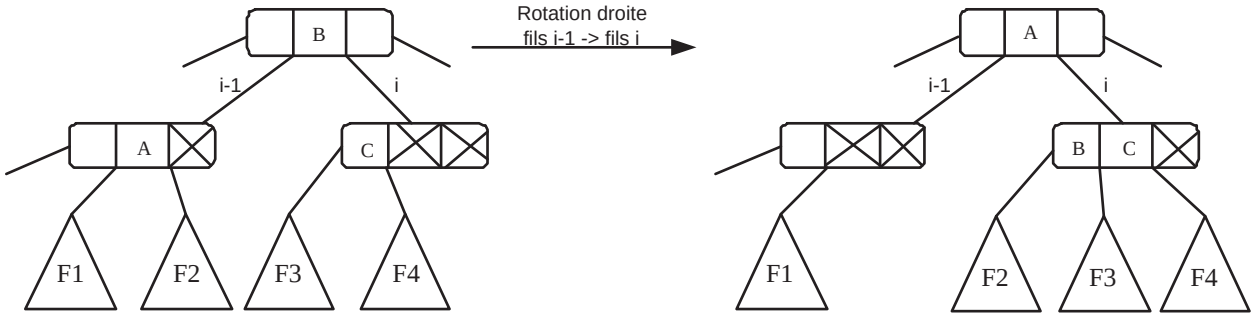

FIGURE 2 – Rotation droite.

(b) **Spécifications :**
La procédure `rd_gen(`B`, `i`)` effectue une rotation du fils $i-1$ vers le fils i (voir figure 2). Conditions : l'arbre B existe, son fils i existe et sa racine n'est pas un $2t$-nœud, le fils $i-1$ existe et sa racine n'est pas un t-nœud.

```
algorithme procedure rd_gen
    parametres locaux
        t_Btree    B
        entier      i

    variables
        t_Btree   L, R
        entier     j
debut
    L ← B↑.fils[i-1]
    R ← B↑.fils[i]
                                /* shift R keys and children */
    pour j ← R↑.nbcles jusqu'a 1 decroissant faire
        R↑.fils[j+2] ← R↑.fils[j+1]
        R↑.cles[j+1] ← R↑.cles[j]
    fin pour
    R↑.fils[2] ← R↑.fils[1]
    R↑.cles[1] ← B↑.cles[i-1]
    R↑.nbcles ← R↑.nbcles + 1
                                /* move lat key and child of L */
    B↑.cles[i-1] ← L↑.cles[L↑.nbcles]
    R↑.fils[1] ← L↑.fils[L↑.nbcles+1]
    L↑.nbcles ← L↑.nbcles-1
fin algorithme procedure rd_gen
```

Spécifications :
La procédure `rg_gen (B, i)` effectue une rotation du fils $i+1$ vers le fils i de B.
Conditions : L'arbre B existe, son fils i existe et sa racine n'est pas un $2t$-nœud, le fils $i+1$ existe et sa racine n'est pas un t-nœud.

```
algorithme procedure rg_gen
   parametres locaux
      t_Btree    B
      entier     i

   variables
      entier     j
      t_Btree    L, R      /* to simplify */
debut
    L ← B↑.fils[i]
    R ← B↑.fils[i+1]

    L↑.nbcles ← L↑.nbcles + 1
    L↑.cles[L↑.nbcles] ← B↑.cles[i]
    L↑.fils[L↑.nbcles+1] ← R↑.fils[1]

    B↑.cles[i] ← R↑.cles[1]
                                          /* R keys shifts */
    pour j ← 1 jusqu'a R↑.nbcles-1 faire
        R↑.cles[j] ← R↑.cles[j+1]
    fin pour
    si R↑.fils[1] <> NUL alors        /* R children shifts */
        pour j ← 1 jusqu'a R↑.nbcles faire
            R↑.fils[j]← R↑.fils[j+1]
        fin pour
    fin si
    R↑.nbcles ← R↑.nbcles-1
fin algorithme procedure rg_gen
```

3. **Fusion**

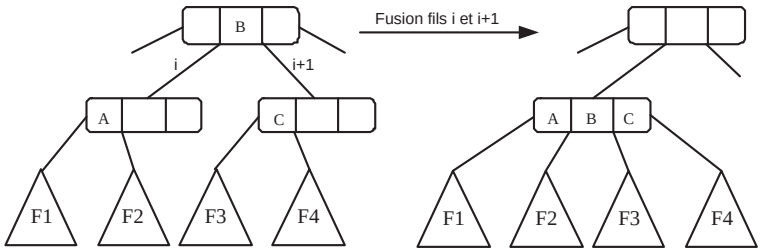

FIGURE 3 – Fusion dans un arbre 2-3-4

Pour les B-arbres inverser le schéma de l'éclatement (figure 1) !

(b) **Spécifications :**
La procédure `fusion(`B`, `i`)` fusionne les fils i et $i+1$ de l'arbre B (voir figure 3).
Conditions : l'arbre B existe et sa racine n'est pas un t-nœud, ses fils i et $i+1$ existent et leurs racines sont des t-nœuds.

```
algorithme procedure fusion
    parametres locaux
        t_Btree  B
        entier    i

    variables
        entier    j
        t_Btree  L, R
debut
    L ← B↑.fils[i]
    R ← B↑.fils[i+1]

    L↑.cles[t] ← B↑.cles[i]
    pour j ← 1 jusqu'a t-1 faire
       L↑.fils[t+j] ← R↑.fils[j]
       L↑.cles[t+j] ← R↑.cles[j]
    fin pour
    L↑.fils[2*t] ← R↑.fils[t]
    L↑.nbcles ← 2*t
    liberer (R)

    pour j ← i jusqu'a B↑.nbcles-1 faire
        B↑.cles[j] ← B↑.cles[j+1]
        B↑.fils[j+1] ← B↑.fils[j+2]
    fin pour
    B↑.nbcles ← B↑.nbcles -1
fin algorithme procedure fusion
```

5. **Spécifications :**
   La procédure `delete_rec (x, B)` supprime la clé x du B-arbre B. L'arbre B n'est pas vide.

```
algorithme procedure delete_rec
    parametres locaux
        t_element  x
        t_Btree       B

    variables
        entier   i
debut
    i ← search_pos (x, B)                                   /* search x position */

    si B↑.fils[1] <> NUL alors                                          /* internal node */
        si (i <= B↑.nbcles) et (B↑.cles[i] = x) alors      /* x found */
            si B↑.fils[i]↑.nbcles > B↑.fils[i+1]↑.nbcles alors
                B↑.cles[i] ← max_Btree (B↑.fils[i])
                delete_rec (B↑.cles[i], B↑.fils[i])
            sinon
                si B↑.fils[i+1]↑.nbcles > t-1 alors
                    B↑.cles[i] ← min_Btree (B↑.fils[i+1])
                    delete_rec (B↑.cles[i], B↑.fils[i+1])
                sinon
                    fusion(B, i)
                    delete_rec (x, B↑.fils[i])
                fin si
            fin si
        sinon                                          /* x ∉ B */
            si B↑.fils[i]↑.nbcles = t-1 alors
                si (i > 1) et (B↑.fils[i-1]↑.nbcles > t-1) alors
                    rd_gen (B, i)
                sinon
                    si (i <= B↑.nbcles) et (B↑.fils[i+1]↑.nbcles > t-1) alors
                        rg_gen (B, i)
                    sinon
                        si i > 1 alors
                            i ← i-1
                        fin si
                        fusion (B,i)
                    fin si
                fin si
            fin si
            delete_rec (x, B↑.fils[i])
        fin si
    sinon                                                               /* leaf */
        si (i <= B↑.nbcles) et (B↑.cles[i] = x) alors
            pour j ← i jusqu'a B↑.nbcles-1 faire
                B↑.cles[j] ← B↑.cles[j+1]
            fin pour
            B↑.nbcles ← B↑.nbcles-1
        fin si                                          /* x ∉ B */
    fin si
fin algorithme procedure delete_rec
```

La procédure d'appel : elle se contente de lancer la procédure de suppression et de remplacer la racine par son fils unique en retour si celle-ci est devenue vide (la procédure récursive a effectué une fusionsur la racine qui était un 2-nœud).

Spécifications :

La procédure `delete_Btree (`x`,` B`)` supprime la clé x du B-arbre B.

```
algorithme procedure delete_Btree
    parametres locaux
        t_element  x
    parametres globaux
        t_Btree       B

    variables
        t_Btree    temp
debut
    si B <> NUL alors
        delete_rec (x, B)
        si B↑.nbcles = 0 alors
            temp ← B
            B ← B↑.fils[1]
            liberer (temp)
        fin si
    fin si
fin algorithme procedure delete_Btree
```