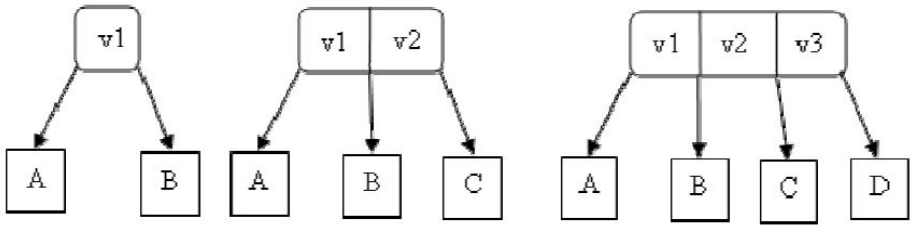
TD 8

Exemple particulier de b-arbre de recherche : les arbres 2-3-4

Les arbres 2-3-4 sont des b-arbres[1] d'ordre 2 tels que:

1) tous les nœuds ont 2, 3 ou 4 fils selon le schéma suivant:



Les sous-arbres A, B, C et D ont les propriétés suivantes :

- tous les nœuds du sous arbre A ont une valeur inférieure à v1.
- tous les nœuds du sous arbre B ont une valeur supérieure ou égale à v1 et inférieure à v2, pour les 3 et 4-nœuds.
- tous les nœuds du sous arbre C ont une valeur supérieure ou égale à v2 et inférieure à v3, pour les 4-nœuds.
- tous les nœuds du sous arbre D ont une valeur supérieure ou égale à v3.

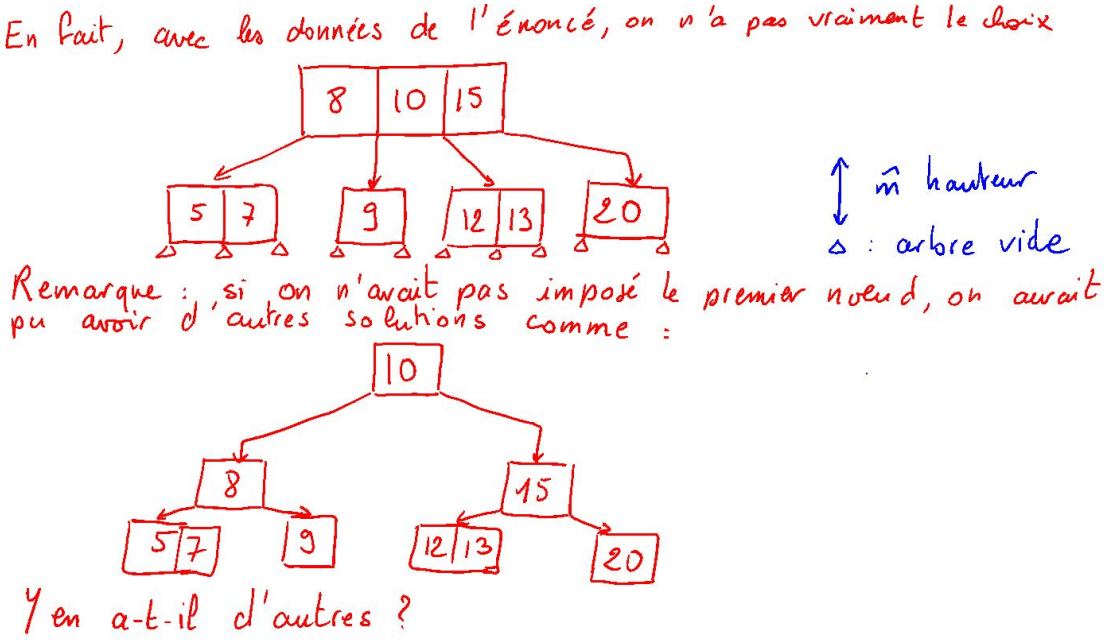
2) tous les nœuds qui sont des feuilles sont à la même profondeur.

Questions

a. Dessiner un arbre 2-3-4 contenant la séquence 10, 5, 8, 7, 15, 20, 12, 9, 13 tel que la racine est un 4-noeud contenant 8, 10 et 15.

Rép. avec le schéma ci-dessous.

[1] Les arbres B ou arbres de Bayer ou B-arbre (B pour balanced par exemple) sont des arbres équilibrés avec des noeuds pouvant contenir 2m-1 valeurs et 2m fils. Voir B-tree sur Wikipédia par exemple.



b. On se donne la spécification suivante. Ici, on indique que le profile des opérations.

```
spéc ARB234(ORDT)
sorte Arb234
opérations
Λ : → Arb234 /* arbre vide */
<_,(_),_> : Arb234 S Arb234 → Arb234      /* enracinement sur un 1-noeud */
<_,(_),_,(_),_> : Arb234 S Arb234 S Arb234 → Arb234
                                          /* enracinement sur un 2-noeud */
<_,(_),_,(_),_,(_),_> : Arb234 S Arb234 S Arb234 S Arb234 → Arb234
                                          /* sur un 3-noeud */
vide : Arb234 → Bool                      /* test de vacuité */
2-fils, 3-fils, 4-fils : Arb234 → Bool    /* test de 2, 3 ou 4 sous-arbres */
f1, f2, f3, f4 : Arb234 → Arb234          /* les sous-arbres */
r1, r2, r3 : Arb234 → S                   /* les étiquettes à la racine */
```

Quelles sont les pré-conditions pour ces opérations?
Ajout : donnez les axiomes (élémentaires), pour ces opérations.

c. Proposer une structure de données d'arbre 2-3-4. Implanter en C toutes les opérations précédentes (de la question b).

d. Proposer une opération permettant de rechercher un élément dans un arbre 2-3-4.

e. Proposer une opération permettant de calculer la hauteur d'un arbre 2-3-4.

f. Proposer une procédure C permettant d'afficher les étiquettes d'un arbre 2-3-4 par ordre croissant.

Rép. On donne les préconditions dans le fichier header suivant. Les noms ont été légèrement changés d'une part parce que le langage C n'offre pas la possibilité de définir de nouveaux

opérateurs infixes. Et comme nous l'avons souvent fait, nous avons systématiquement préfixé par `ba_` toutes les opérations relatives aux b-arbres. Notez aussi que `ba_vide()` désigne l'arbre vide et le test de vacuité est réalisé par la fonction `ba_estvide()`.
Enfin, on utilise un type énuméré (et la fonction `ba_ntype()` qui renvoie une valeur de ce type), ce qui est plus facilement généralisable que les fonctions de type proposées par la spécification que vous devriez quand même implanter (exercice imposé par l'énoncé et non corrigé ici).

```c
// fichier arbre234.h
#ifndef __ARBRE234_H__
#define __ARBRE234_H__
#include<stdbool.h>

typedef unsigned Nat;
typedef int S;
typedef enum { type2fils=1, type3fils, type4fils} node_type;

typedef struct node {
    node_type type;          // type du noeud 2, 3 ou 4
    struct node *fils1;      // 1er fils
    S val1;                  // 1ère racine
    struct node *fils2;      // etc.
    S val2;
    struct node *fils3;
    S val3;
    struct node *fils4;
                    } strNode, *Arb234;


// préc : aucune
// post : retourne l'arbre vide
// modif ; aucune
Arb234 ba_vide();

// description : enracinement avec une racine à une valeur
// préc ba_enrac1(f1, r1, f2) = ba_hauteur(f1) == ba_hauteur(f2)
//                           le max de f1 est  inférieur à r1 qui est inférieur
//                           au min de de f2
// post : retourne un arbre 234 avec une racine à 1 valeur : r1
// et deux ss-arbres f1 et f2
// modif : allocation d'une structure strNode

Arb234 ba_enrac1(Arb234 f1, S r1, Arb234 f2);
```

```
// description : enracinement avec une racine à deux valeurs
// préc ba_enrac1(f1, r1, f2, r2, f3) = ba_hauteur(f1) == ba_hauteur(f2)
//                                          == ba_hauteur(f3)
// post : retourne un arbre 234 avec une racine à 2 valeurs : r1 et r2
//                                et trois ss-arbres f1, f2 et f3
//               les valeurs rangées dans f1 sont inférieures à r1
//               celles de f2 sont comprises entre r1 et r2
//               celles de f3 supérieures à r2
// modif : allocation d'une structure strNode
Arb234 ba_enrac2(Arb234 f1, S r1, Arb234 f2, S r2, Arb234 f3);

// description : enracinement avec une racine à trois valeurs
// préc ba_enrac1(f1, r1, f2, r2, f3, r3, f4) =
//       ba_hauteur(f1) == ba_hauteur(f2) == ba_hauteur(f3) == ba_hauteur(f4)
//               les valeurs rangées dans f1 sont inférieures à r1
//               celles de f2 sont comprises entre r1 et r2
//               celles de f3 comprises entre r2 et r3
//               celles de f4 supérieures à r3
// post : retourne un arbre 234 avec une racine à 3 valeur : r1, r2 et r3
//                                  et quatre ss-arbres f1, f2, f3 et f4
// modif : allocation d'une structure strNode
Arb234 ba_enrac3(Arb234 f1, S r1, Arb234 f2,S r2, Arb234 f3, S r3, Arb234 f4);

// description : test de vacuité
// préc ba_estvide(b) = vrai
// post : true si b est l'arbre vide, false sinon
// modif : aucune
bool ba_estvide(Arb234 b);


// préc : aucune
// post : retourne le le type du noeud type2fils ou type3fils ou type4fils
// modif ; aucune
node_type ba_ntype(Arb234 b);


// préc ba_f1(b) = true
// post : retourne le premier fils
// modif ; aucune
Arb234 ba_f1(Arb234 b);

// préc ba_f2(b) = true
// post :retourne le
// modif ; aucune
Arb234 ba_f2(Arb234 b);

// préc ba_f3(b) = ba_ntype(b) != type2fils
// post :retourne le deuxième fils
// modif ; aucune
Arb234 ba_f3(Arb234 b);
```

```
// préc ba_f4(b) = ba_ntype = type4fils
// post :retourne le troisième fils
// modif ; aucune
Arb234 ba_f4(Arb234 b);

// préc ba_hauteur(b) = b est un arbre 234 : toutes les branches ont la même
longueur
// post : retourne la hauteur de b
// modif : aucune
Nat ba_hauteur(Arb234 b);

// affichage sans fioriture pour pouvoir faire des test
void ba_print(Arb234 b);
void ba_print_aux(Arb234 b, Nat prof);

// affichage par ordre croissant des valeurs stockées dans l'arbre
// préc : aucune
// modif : affichage
void ba_print_val(Arb234 b);
// exercice : faites un affichage dans l'ordre décroissant


////////////////////////////
// recherche

// préc aucune
// post : un arbre dont la racine contient x, NULL si x n'apparaît pas dans b
// modif : aucune
Arb234 ba_app(S x, Arb234 b);
#endif
```

Voici maintenant le code C de toutes ces fonctions (avec à la fin, un micro programme de test, préférez un fichier à part quand vous faites un projet : c'est beaucoup plus souple). Quelques fonctions ne sont pas codées dans le fragment de code ci-dessous. Complétez-le.

```
// fichier arbre234.c

#include<stdlib.h>
#include<stdio.h>
#include<assert.h>
#include<stdbool.h>
#include "arbre234.h"
/* Rappel
typedef enum { type2fils=1, type3fils, type4fils} node_type;

typedef struct node {
   node_type type;
   struct node *fils1;
   S val1;
   struct node *fils2;
   S val2;
   struct node *fils3;
```

```
   S val3;
   struct node *fils4;
                  } strNode, *Arb234;
*/
// préc : aucune
// post : retourne l'arbre vide
// modif : aucune
Arb234 ba_vide()
{
   return (arbre234) NULL;
}

// description : enracinement avec une racine à une valeur
// préc ba_enrac1(f1, r1, f2) = ba_hauteur(f1) == ba_hauteur(f2)
// modif : allocation d'une structure strNode
Arb234 ba_enrac1(Arb234 f1, S r1, Arb234 f2)
{
   Arb234 res = (Arb234)malloc(sizeof(strNode));
   res->type = type2fils;
   res->val1 = r1;
   res->fils1 = f1;
   res->fils2 = f2;
   return res;
}

// description : enracinement avec une racine à deux valeurs
// préc ba_enrac1(f1, r1, f2, r2, f3) =
// ba_hauteur(f1) == ba_hauteur(f2) == ba_hauteur(f3)
// modif : allocation d'une structure strNode
Arb234 ba_enrac2(Arb234 f1, S r1, Arb234 f2, S r2, Arb234 f3)
{
   Arb234 res = (Arb234)malloc(sizeof(strNode));
   res->type = type3fils;
   res->val1 = r1;
   res->val2 = r2;
   res->fils1 = f1;
   res->fils2 = f2;
   res->fils3 = f3;
   return res;
}

// description : enracinement avec une racine à trois valeurs
// préc ba_enrac1(f1, r1, f2, r2, f3, r3, f4) =
//   ba_hauteur(f1) == ba_hauteur(f2) == ba_hauteur(f3) == ba_hauteur(f4)
// modif : allocation d'une structure strNode
Arb234 ba_enrac3(Arb234 f1, S r1, Arb234 f2,S r2, Arb234 f3, S r3, Arb234 f4)
{
   Arb234 res = (arbre234)malloc(sizeof(strNode));
   res->type = type4fils;
   res->val1 = r1;
   res->val2 = r2;
   res->val3 = r3;
   res->fils1 = f1;
   res->fils2 = f2;
   res->fils3 = f3;
   res->fils4 = f4;
   return res;
}

// préc : aucune
// post : retourne le le type du noeud type2fils ou type3fils ou type4fils
// modif ; aucune
node_type ba_ntype(Arb234 b)
{
   return b->type;
}
```

```
// préc ba_f1(b) = true
// post : retourne le premier fils
// modif ; aucune
Arb234 ba_f1(Arb234 b)
{
   return b->fils1;
}

// préc ba_f2(b) = true
// post :retourne le
// modif ; aucune
Arb234 ba_f2(Arb234 b)
{
   return b->fils2;
}


// préc ba_f3(b) = ba_ntype(b) != type2fils
// post :retourne le deuxième fils
// modif ; aucune
Arb234 ba_f3(Arb234 b)
{
   return b->fils3;
}

// préc ba_f4(b) = ba_ntype = type4fils
// post :retourne le troisième fils
// modif ; aucune
Arb234 ba_f4(Arb234 b)
{
   return b->fils4;
}

void ba_print(Arb234 b)
{
   ba_print_aux(b,0);
}
void ba_print_aux(Arb234 b, Nat prof)
{
   if(!b) {
             for(int i=0; i<prof; i++) printf("   ");
             printf("   <|\n"); return;
           }
   ba_print_aux(b->fils1, prof+2);
   for(int i=0; i<prof; i++) printf("   ");
   printf("|: %3d\n", b->val1);
   ba_print_aux(b->fils2, prof+2);
   if(b->type > type2fils)
   {
       for(int i=0; i<prof; i++) printf("   ");
       printf("|: %3d\n", b->val2);
       ba_print_aux(b->fils3, prof+2);
       if(b->type > type3fils)
       {
           for(int i=0; i<prof; i++) printf("   ");
           printf("|: %3d\n", b->val3);
           ba_print_aux(b->fils4, prof+2);
       }
   }

}

// préc ba_hauteur(b) = b est un arbre 234 : toutes les branches
//                         ont la même longueur
```

```c
// post : retourne la hauteur de b
// modif : aucune
Nat ba_hauteur(Arb234 b)
{
   int i=-1;
   for(;b;b=b->fils1)i++;
   return i;
}

// préc aucune
// post : un arbre dont la racine contient x, NULL si x n'apparaît pas dans b
// modif : aucune
Arb234 ba_app(S x, Arb234 b)
{
   if(!b) return ba_vide();
   switch (b->type)
   {
   case type4fils :
       if(x > b->val3) return ba_app(x, b->fils4);
       if(x == b->val3) return b;
   case type3fils :
       if(x > b-> val2) return ba_app(x,b->fils3);
       if(x == b->val2) return b;
   default :
       if(x > b-> val1) return ba_app(x,b->fils2);
       if(x == b->val1) return b;
       return ba_app(x, b->fils1);
   }
}


void ba_print_val(Arb234 b)
{
   if(!b) return;
   ba_print_val(b->fils1);
   printf(" %3d,",b->val1);
   ba_print_val(b->fils2);
   if(b->type > type2fils)
   {
       printf(" %3d,",b->val2);
       ba_print_val(b->fils3);
       if(b->type == type4fils)
       {
           printf(" %3d,",b->val3);
           ba_print_val(b->fils4);
       }
   }
}

// test
int main()
{
   Arb234 f1 = ba_enrac2(ba_vide(),5,ba_vide(),7,ba_vide()),
          f2 = ba_enrac1(ba_vide(),9,ba_vide()),
          f3 = ba_enrac2(ba_vide(),12,ba_vide(),13,ba_vide()),
          f4 = ba_enrac1(ba_vide(),20,ba_vide()),
          b = ba_enrac3(f1,8,f2,10,f3,15,f4);

   ba_print(b);

   printf("hauteur : %d\n", ba_hauteur(b));

   int x; scanf("%d",&x);
   printf(" %d est dans b : %s\n", x, ((bool)ba_app(x, b)) ? "oui" : "non");

   ba_print_val(b);
   putchar('\n');
```

```
}
```

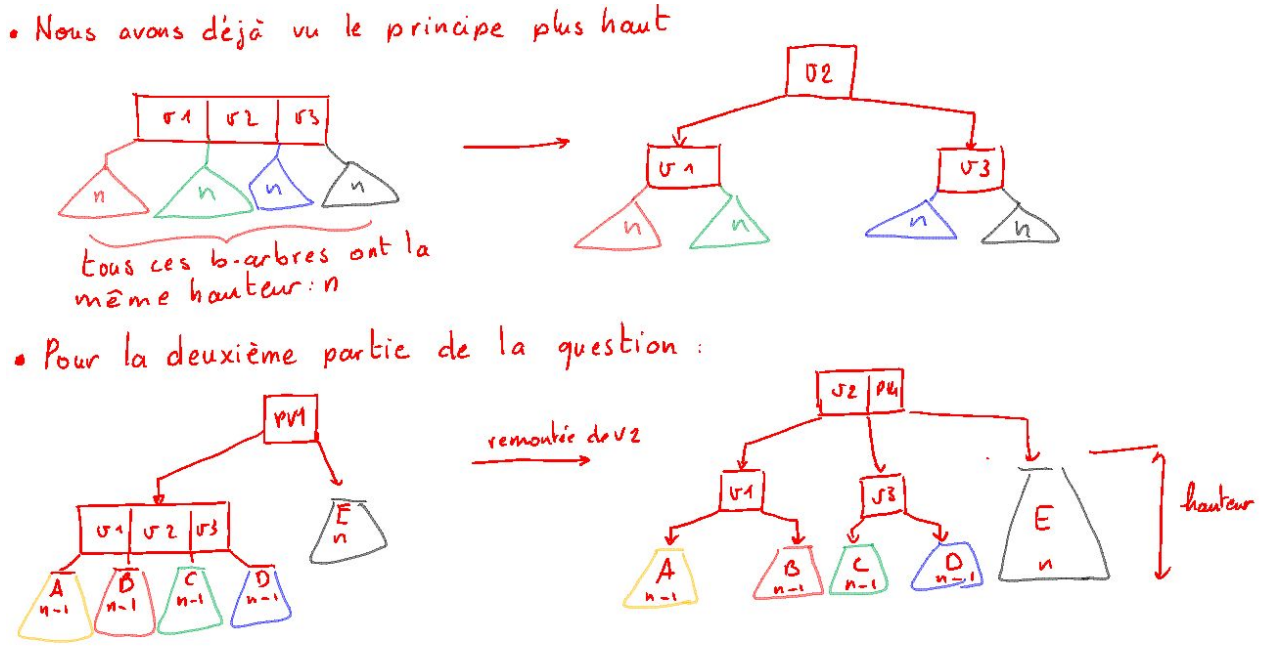
---

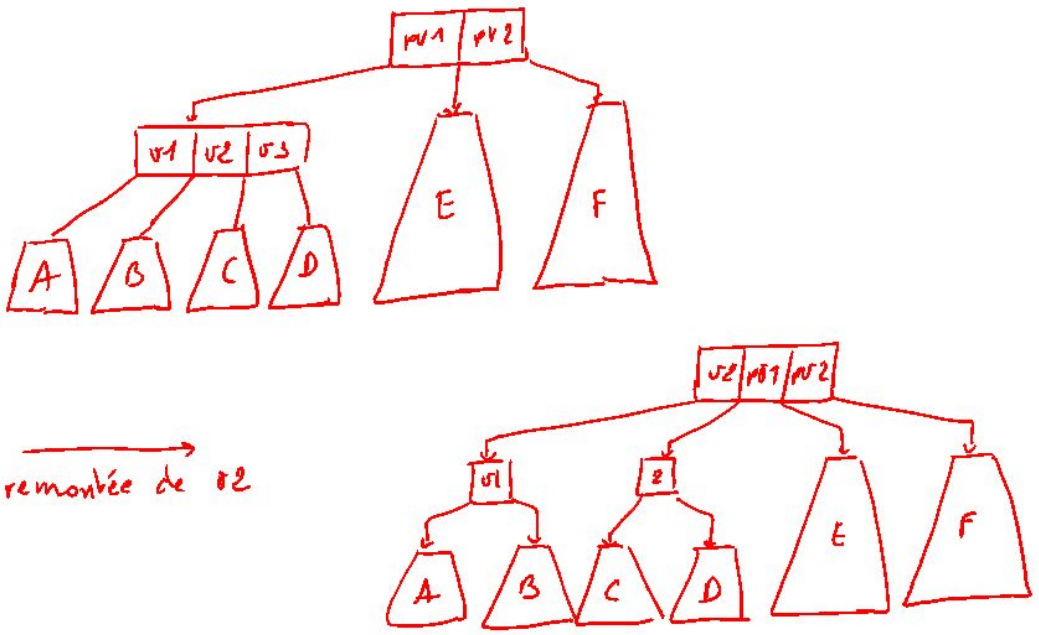
g. Opération d'insertion en feuille

g.1 Comment éclater un 4-noeud ? Faire un dessin d'un 4-noeud et montrer comment il peut être remplacé par trois 2-noeud en augmentant la hauteur de l'arbre.

g.2 Comment éclater les 4-noeuds dans les deux cas suivants sans augmenter la hauteur de l'arbre ?

Rép. avec les schémas suivants





g.3 Insérer une nouvelle valeur en feuille.
L'insertion en feuille ne pose aucun problème lorsque qu'on insère dans un 2- ou 3-noeud: il suffit de transformer le 2-noeud en 3-noeud et le 3-noeud en 4-noeud. Par contre, un problème se pose lorsque l'on est en présence d'un 4-noeud: on ne peut plus ajouter. On se propose alors, quand on descend dans l'arbre vers un 4-nœud, d'éclater celui-ci avant d'y parvenir en utilisant les techniques du e.1 et e.2. Ainsi, on est certain de ne jamais ajouter une nouvelle valeur dans un 4-nœud.
Si le 4-nœud n'est pas la racine, il est le fils d'un 2-nœud ou d'un 3-nœud. L'insertion conserve alors la propriété «toutes les feuilles sont à la même profondeur» : le seul moment où la hauteur de l'arbre augmente, est lors de l'éclatement de la racine. Cet éclatement augmente les profondeurs de toutes les feuilles de 1.

On y va progressivement. Les fonctions déclarées avec "static" ne sont visible que dans le fichier où elles sont définies (en principe, on ne les déclare pas dans un fichier header, leur usage est purement local). Deux petites fonctions de test classique.

```
////////////////////////////////////////////////////////////////////////////////
// insertion aux feuilles
// avec éclatement à la descente de tout noeud de type type4tils
////////////////////////////////////////////////////////////////////////////////
// test de foliacitude
// en principe tester fils 1 suffit si on a bien un arbre234
bool ba_estvide(arbre234 b)
{
   return b == (arbre234) NULL;
}

bool ba_estfeuille(arbre234 b)
{
   if(!b) return false;
   else return ba_estvide(b->fils1);
   /* //code plus défensif
   bool res = true;
   switch(b->type)
   {
```

```
        type4fils : res = ba_estvide(b->fils4);
        type3fils : res = res && ba_estvide(b->fils3);
        type2fils : res = res && ba_estvide(b->fils2) && ba_estvide(b->fils1);
    }
    return res;
    */
}
```

Ensuite, 5 fonctions auxiliaires pour coder toutes les variantes des petites opérations de "remonté de v2" expliquées graphiquement plus haut. Elles s'appellent ba_rem<x><l> avec x = 1 ou 2 suivant que le noeud père soit de type 2 fils ou 3 fils et l = d (droit), g (gauche) ou m (milieu) (on ne considère pas le cas où le père à 4 fils car il faut alors considérer le grand père … jusqu'à la racine. La stratégie d'éclatement indiquée permet d'éviter ce cas. Cela donne 5 cas et 5 fonctions qui ne sont pas difficiles à comprendre.

```
// remontée de la valeur centrale
// dans le cas où la racine a 1 élément (type type2fils --> type3fils)
// et le noeud à éclater est la ss-arbre gauche
static arbre234 ba_rem1g(arbre234 b)
{
   arbre234 f1 = b->fils1;

   // déplacement de la valeur à la racine et de son fils gauche:
   b->type = type3fils;
   b->fils3 = b->fils2;
   b->val2 = b->val1;
   // puis remonté de v2 et de ses nouveaux fils
   b->val1 = f1->val2;
   b->fils2 = ba_enrac1(f1->fils3,f1->val3, f1->fils4);
   f1->type = type2fils;
   return b;
}

// remontée de la valeur centrale
// dans le cas où la racine a 1 élément (type type2fils --> type3fils)
// et le noeud à éclater est la ss-arbre droit
static arbre234 ba_rem1d(arbre234 b)
{
   arbre234 f2 = b->fils2;


   b->type=type3fils;
   // on ne change pas le lien du f1, ni celui de f2
   f2->type = type2fils;   // on n'efface rien attention
   b->val2  = f2->val2;
   b->fils3 = ba_enrac1(f2->fils3,f2->val3, f2->fils4);


   return b;
}

// remontée de la valeur centrale dans le cas où
// la recine a 3 fils : 3 cas à considérer
// remontée depuis le fils gauche
static arbre234 ba_rem2g(arbre234 b)
{
   arbre234 f1 = b->fils1;

   // décalage des valeur et des fils pour insérer v2 en premier
   b->type = type4fils;
   b->val3 = b->val2;
   b->val2 = b->val1;
   b->val1 = f1->val2;
   b->fils4 = b->fils3;
```

```
   b->fils3 = b->fils2;
   // placement des nouveaux fils (comme dans ba_rem1g)
   b->fils2 = ba_enrac1(f1->fils3,f1->val3, f1->fils4);
   f1->type = type2fils;

   return b;
}

// remontée de la valeur centrale dans le cas où
// la recine a 3 fils : 3 cas à considérer
// remontée depuis le fils du milieu
static arbre234 ba_rem2m(arbre234 b)
{
   arbre234 f2 = b->fils2;
   // on de décale que la deuxième valeur
   b->type = type4fils;
   b->val3 = b->val2;
   b->fils4 = b->fils3;
   // on insére v2 et ses deux nouveux fils
   b->val2 = f2->val2;
   b->fils3 = ba_enrac1(f2->fils3, f2->val3, f2->fils4);
   // le point sur fils2 ne change pas ! seulement le type
   f2->type = type2fils;
   return b;
}

// remontée de la valeur centrale dans le cas où
// la recine a 3 fils : 3 cas à considérer
// remontée depuis le fils du droit
static arbre234 ba_rem2d(arbre234 b)
{
   arbre234 f3 = b->fils3;

   // pas de décalage : v2 va à droite dans le nouveu noeud
   b->type = type4fils;
   b->val3 = f3->val2;
   b-> fils4 = ba_enrac1(f3->fils3, f3->val3, f3->fils4);
   // le point sur fils3 ne change pas ! seulement le type
   f3 ->type = type2fils;

   return b;
}
```

On considère ensuite l'insertion dans une feuille : on se placerra toujours dans le cas où il y a au moins une place de libre dans la feuille.

```
// prec f est une feuille et son type est < type4fils
static arbre234 ba_insfeuille(S x, arbre234 f)
{
   if(f->type==type3fils)
   {
       f->type=type4fils;
       f->fils4 = ba_vide();
       if(x > f->val2) f->val3 = x;
       else if(x > f->val1) {f->val3=f->val2; f->val2 = x;}
       else { f->val3 = f->val2; f->val2 = f->val1; f->val1 = x;}
   }
   else // type 2 fils
   {
       f->type = type3fils;
       f->fils3 = ba_vide();
       if(x > f-> val1) f->val2=x;
       else {f->val2 = f->val1; f->val1 = x;}
   }
   return f;

}
```

La fonction suivante est strictement utilitaire : elle permet de descendre dans un noeud de type type4fils sans l'éclater (c'est utile, dans un cas particulier vu pus bas).

```
// fonction d'insertion sur 1 pas sans faire éclater de noeud
// cette fonction est nécessaire pour éviter de faire remonter
// les conséquences d'éclatements de noeuds de type 4fils
static arbre234 ba_ins_aux(S x, arbre234 b)
{
   switch (b->type)
   {
   case type4fils: if(x > b->val3) {ba_ins(x,b->fils4); return b;}
   case type3fils: if(x > b-> val2) {ba_ins(x, b-> fils3); return b;}
   default : if(x > b->val1) ba_ins(x,b->fils2); else ba_ins(x,b-> fils1);
             return b;
   }
}
```

Et voici, maintenant la fonction d'insertion proprement dite. C'est une fonction récusive ici, mais on peut facilement la dérécursiver.

```
arbre234 ba_ins(S x, arbre234 b)
{
   if(!b) return ba_enrac1(ba_vide(),x,ba_vide()); // cas de l'arbre vide
   if(b->type==type4fils) // ne doit se produire qu'à la racine !
   {
       arbre234 f2 = ba_enrac1(b->fils3,b->val3,b->fils4);
       b->type = type2fils;
       arbre234 res = ba_enrac1(b, b->val2, f2);
       return ba_ins(x, res);
   }
   // la racine ne peut plus maintenant être de type 4fils
   // mais ça peut être une feuille (au début)
   if(ba_estfeuille(b)) ba_insfeuille(x,b);
   else
       // maintenant, l'insertion ne peut plus se faire à la racine
       // à la suite b ne peut pas être une feuille
       //            b ne peut pas être de type 4fils
       // on va travailler un coup en avance
       switch (b->type)
       {
       case type3fils : // la remontée de v2 fait apparaître un noeud de
                        //type 4fils à la racine "locale"
                       // on ne peut donc pas faire un appel récursif
                        //à ba_ins() qui soit déséquilibreait
                       // l'arbre, soit nécessiterait l'utilisation d'une
                       // pile pour remonter dans l'arbre
                       // on tulise donc ba_ins-aux() vue plus haut.
                      if(x > b->val2)
                           if(b->fils3->type == type4fils)
                               {b= ba_rem2d(b); b = ba_ins_aux(x, b);}
                           else ba_ins(x,b->fils3);
                       else if(x>b->val1)
                           if(b->fils2->type == type4fils)
                                {b=ba_rem2m(b); b = ba_ins_aux(x, b);}
                           else ba_ins(x,b->fils2);
                       else
                           if(b->fils1->type == type4fils)
                                 {b = ba_rem2g(b); b = ba_ins_aux(x, b);}
                           else ba_ins(x, b->fils1);
                       break;
       case type2fils : if(x > b->val1)
                           if(b->fils2->type == type4fils)
                                 {b = ba_rem1d(b); b = ba_ins(x,b);}
```

```
                        else ba_ins(x,b->fils2);
                    else
                        if(b->fils1->type == type4fils)
                              {b = ba_rem1g(b); b = ba_ins(x,b);}
                        else ba_ins(x,b->fils1);
    }
    return b;

}
```

Et voici une fonction de test qui affiche les insertions successives jusqu'à avoir l'arbre affiché plus bas.

```
// test
int main()
{


    arbre234 t = ba_vide();
    int tab[16] = {10,5,8,7,15,20,12,9,13, 11, 14, 50, 51, 52,53,54};
    for(int i=0;i<16;i++)
    {
        printf("insertion de %d : \n",tab[i]);
        t = ba_ins(tab[i],t);
        ba_print(t);
        printf("----------------------\n");
    }



}
```

---

insertion de 54 :

```
            <|
        |:  5
            <|
        |:  7
            <|
    |:  8
            <|
        |:  9
            <|
|:  10
            <|
        |:  11
            <|
    |:  12
            <|
        |:  13
            <|
```

```
          |:  14
                <|
|:  15
                <|
          |:  20
                <|
     |:  50
                <|
          |:  51
                <|
     |:  52
                <|
          |:  53
                <|
          |:  54
                <|
----------------------
```