Les arbres en C

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

Les arbres d'arité quelconque

Les arbres en C

C vs OCaml

Nous allons maintenant voir comment implémenter des structures d'**arbres** en **C**.

Cela ne sera pas aussi simple qu'en **OCaml**, où il suffit de définir un nouveau **type récursif**.

Nous allons ici gérer une telle structure à la main, avec des **pointeurs**, comme on l'a fait pour les **listes chaînées**.

Une autre différence sera qu'on manipulera des structures **mutables**, et non **persistantes**.

Les arbres en C

Structures mutables ou persistantes

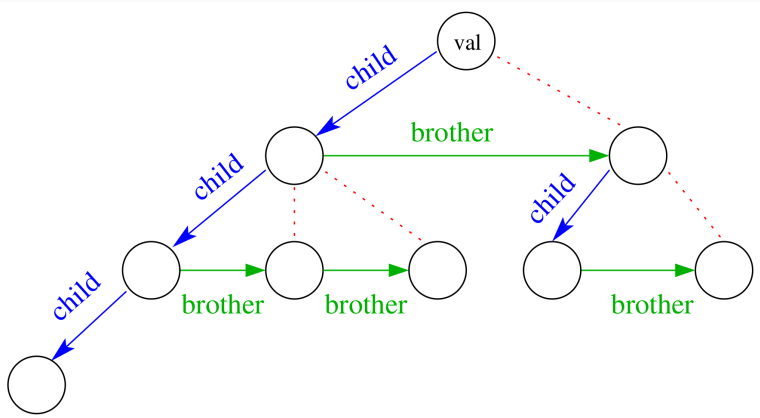
- En **OCaml**, nous manipulons en général des structures **persistantes** : nos fonctions ne modifient pas les objets existants par effets de bord, mais renvoient de **nouveaux** objets (construits potentiellement avec des parties de l'objet initial).
- En **C**, nous manipulons en général des structures **mutables** : nos fonctions modifient directement l'objet pris en entrée par **effets de bord**, sans en faire de copie.

Implémentation des arbres n-aires par pointeurs

Left-Child Right-Sibling

Dans cette implémentation :

- chaque nœud a un **pointeur** sur son **premier fils** ;
- tous les **fils** d'un nœud sont stockés dans une **liste chaînée** (**liste des frères**).




```c
typedef int value_t; // type des valeurs

typedef struct node_t node_t;
struct node_t {
  value_t data;
  node_t *child;
  node_t *sibling;
};
```

Parcours en profondeur

Rappel

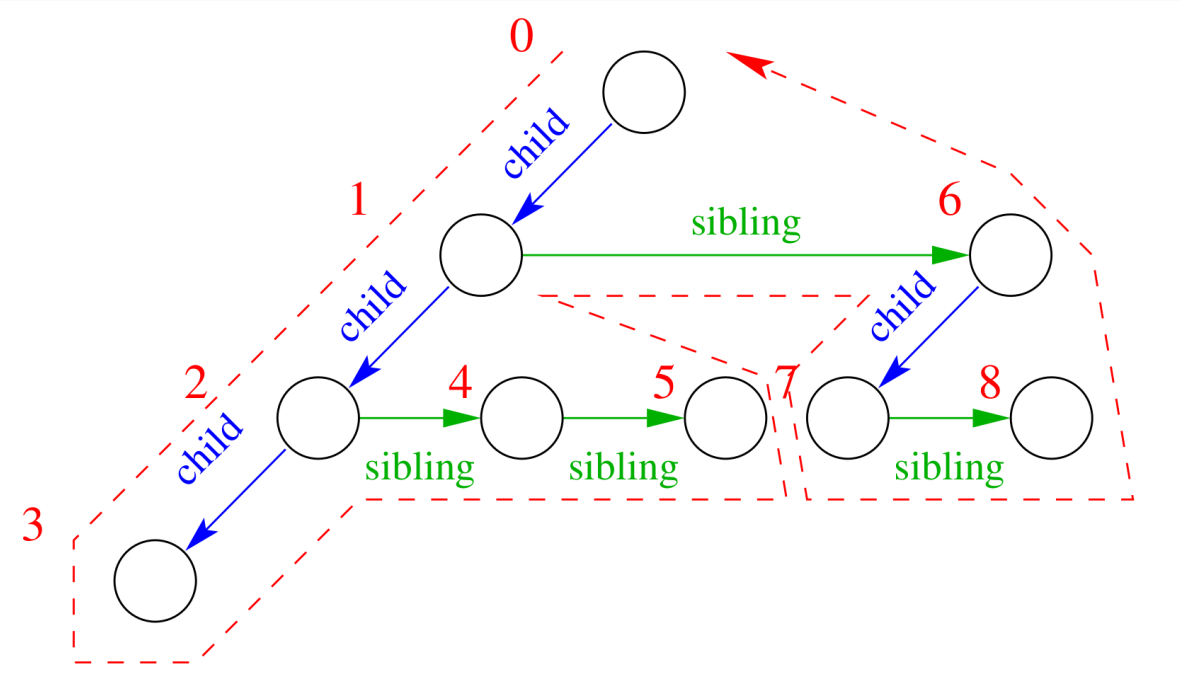
Le **parcours en profondeur** (“**depth first search**” en anglais) consiste à traiter les nœuds de l'arbre dans l'ordre suivant :

- on part de la **racine** ;
- on **descend** d'abord dans le **premier fils** ;
- une fois que tous les **descendants** d'un fils ont été traités (récursivement), on passe au **prochain fils** ;
- une fois que tous les fils ont été traités, on **remonte**.

Parcours en profondeur

```c
void dfs_prefix(node_t *n) {
  if (n != NULL) {
    /* Traitement du noeud en premier */
    traitement(n->data);
    dfs_prefix(n->child); // descente
    dfs_prefix(n->sibling); // prochain frère
  }
}
```

```c
void dfs_postfix(node_t *n) {
  if (n != NULL) {
    dfs_postfix(n->child); // descente
    dfs_postfix(n->sibling); // prochain frère
    /* Traitement du noeud en dernier */
    traitement(n->data);
  }
}
```



Parcours en largeur

Rappel

Le **parcours en largeur** (“**breadth first search**” en anglais) consiste à traiter les nœuds de l'arbre dans l'ordre suivant :

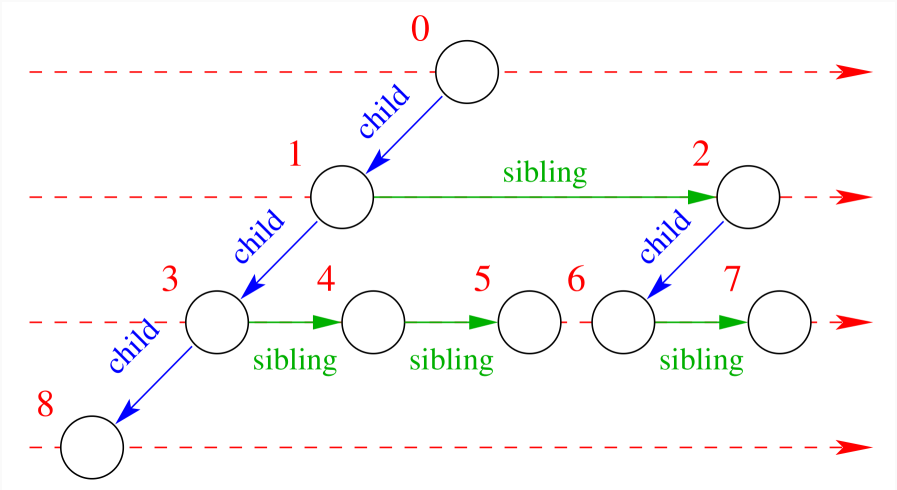
- on part de la **racine** (nœud de profondeur 0) ;
- une fois qu'on a traité tous les nœuds de profondeur k, on traite tous les nœuds de profondeur $k+1$.

Pour cela, on peut utiliser une **file** (cf. chapitre 8) :

- on **extrait** un nœud de la file ;
- on le **traite** ;
- on **insère** tous ses fils.

Parcours en largeur

```
1   // on suppose qu'on a codé une file de *node_t
2   // comme les files d'int du chapitre 8
3
4   void bfs(node_t *n) {
5     file *f = creer_file();
6     enfiler(f,n);
7     while (!file_est_vide(f)) {
8       node_t *s = defiler(f);
9       while (s != NULL) {
10        traitement(s);
11        enfiler(f,s->child);
12        s = s->sibling;
13      }
14    }
15    detruire_file(f);
16  }
```



Les arbres binaires

Arbres binaires en C

```c
typedef int value_t; // type des valeurs

typedef struct node_t node_t;
struct node_t {
  value_t data;
  node_t *left;
  node_t *right;
};
```

Arbres binaires

Pour implémenter les **arbres binaires**, chaque nœud possède :

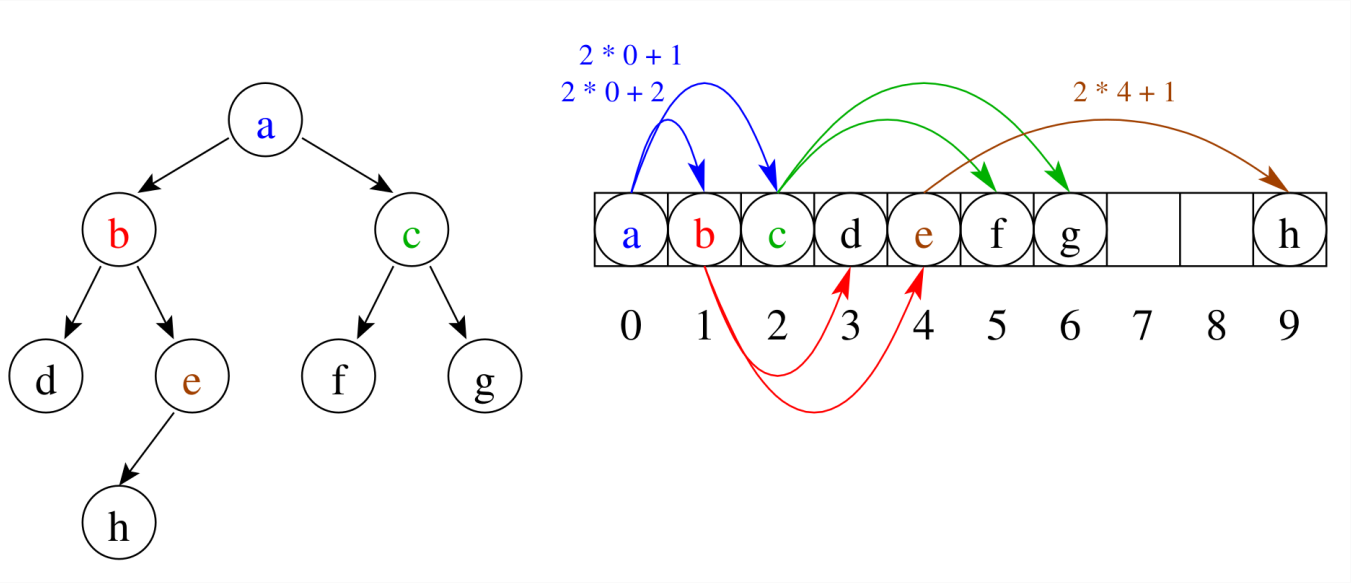
- un champ stockant l'**étiquette** ;
- un **pointeur** vers son **fils gauche** (éventuellement `NULL`) ;
- un **pointeur** vers son **fils droit** (éventuellement `NULL`).

Arbres binaires en C

Remarque

On peut également utiliser un **tableau**, comme on l'a fait pour les **tas** en **OCaml** dans le chapitre précédent.

Dans ce cas, il n'y a pas besoin de pointeur, et le code vu en OCaml s'adapte très facilement en C.



Parcours en profondeur

```c
void prefix(node_t *n) {
  if (n != NULL) {
    traitement(n->data);
    prefix(n->left);
    prefix(n->right);
  }
}
```

```c
void infix(node_t *n) {
  if (n != NULL) {
    infix(n->left);
    traitement(n->data);
    infix(n->right);
  }
}
```

```c
void postfix(node_t *n) {
  if (n != NULL) {
    postfix(n->left);
    postfix(n->right);
    traitement(n->data);
  }
}
```

Ordres

On présente ici les 3 **parcours en profondeur** sur les **arbres binaires** : **préfixe**, **infixe**, et **postfixe**.

Parcours en largeur

```
1  // on suppose qu'on a codé une file de *node_t
2  // comme les files d'int du chapitre 8
3
4  void bfs(node_t *n) {
5    file *f = creer_file();
6    enfiler(f,n);
7    while (!file_est_vide(f)) {
8      node_t *current = defiler(f);
9      if (current != NULL) {
10       traitement(current->data);
11       enfiler(f,s->left);
12       enfiler(f,s->right);
13     }
14   }
15   detruire_file(f);
16 }
```

Parcours en largeur

Pour le **parcours en largeur**, on utilise une **file** comme dans la partie précédente.

Les arbres bicolores

Arbres binaires de recherche

Rappel

Un **arbre binaire de recherche** (ABR) est un arbre binaire éventuellement vide dont les nœuds sont des couples (clé, valeur), tel que :

- deux nœuds n'ont **jamais** la **même clé** ;
- les clés sont à valeur dans un ensemble **totalement ordonné** ;
- pour tout nœud de l'arbre, les clés du **sous-arbre gauche** sont **strictement inférieures** à celle du nœud, et les clés du **sous-arbre droit** sont **strictement supérieures** à celle du nœud.

ABR équilibrés

On a remarqué dans le chapitre précédent que les ABR ne sont pas forcément **équilibrés** : si un ABR à n nœuds est de hauteur h, on peut avoir $h = O(n)$.

On a vu une première méthode pour garantir $h = O(\log n)$: les **arbres AVL**.

On en voit maintenant une autre : les **arbres bicolores** (aussi appelés **arbres rouge-noir**).

Arbres bicolores

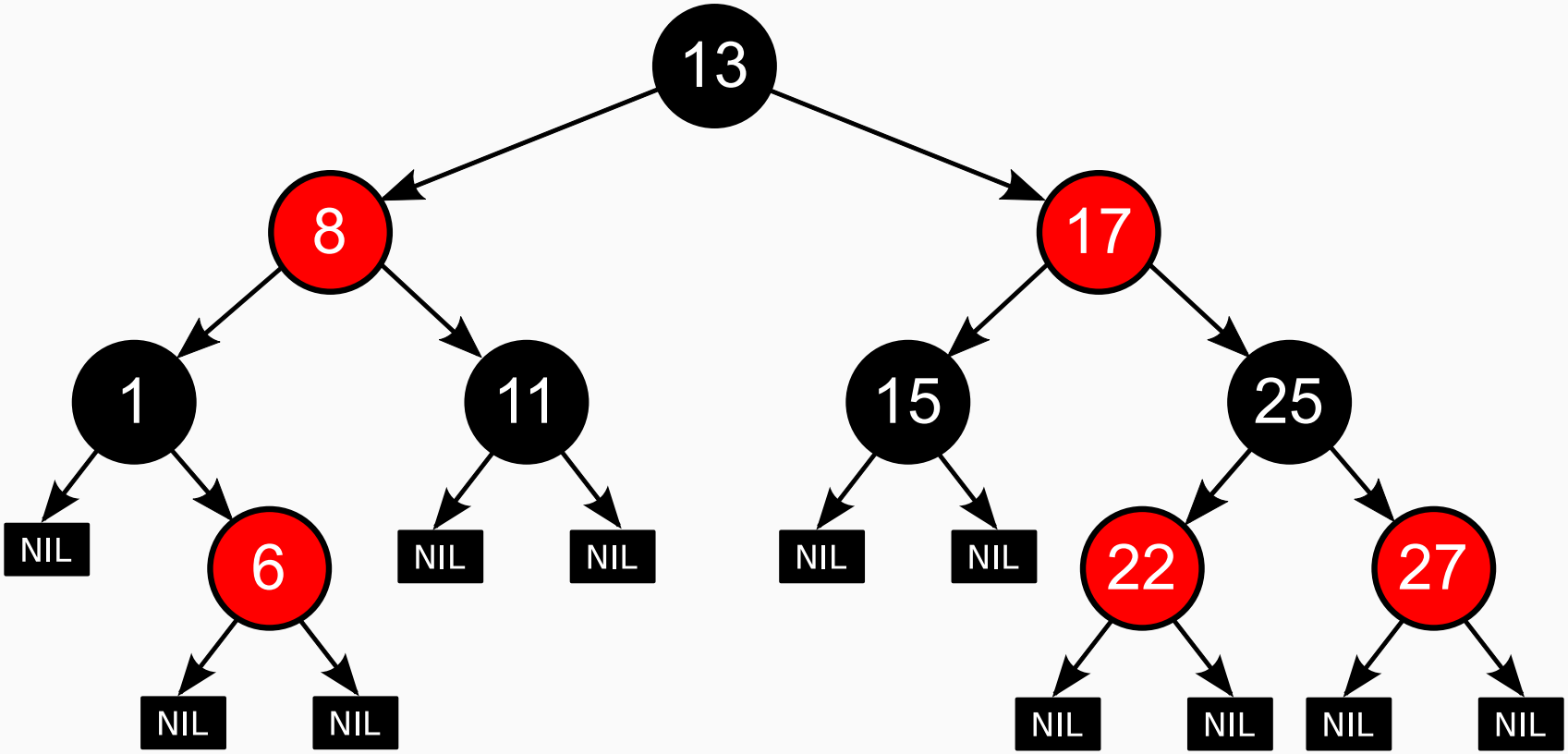
Définition (Arbres bicolores)

Un **arbre bicolore** est un **ABR** dans lequel chaque nœud possède un attribut supplémentaire : sa **couleur** (**rouge** ou **noire**).

Il doit vérifier les propriétés suivantes :

1. un nœud est soit **rouge**, soit **noir** ;
2. la racine est noire ;
3. les enfants d'un nœud rouge sont noirs ;
4. tout nœud possède deux enfants (éventuellement des feuilles `NULL`) ; les feuilles `NULL` sont noires ;
5. le chemin de la racine à n'importe quelle feuille (`NULL`) contient le même nombre de nœuds noirs, appelé **hauteur noire** de l'arbre.

Arbres bicolores



Arbres bicolores

Propriété

Soit h la hauteur et h_n la hauteur noire d'un **arbre bicolore**.

Alors : $h_n \leq h \leq 2 * h_n$.

Preuve

La première inégalité est atteinte si l'arbre ne possède aucun nœud rouge.

La seconde inégalité est atteinte si toutes les branches alternent toujours entre un nœud noir et un nœud rouge.

Arbres bicolores

Propriété

Soit h la hauteur d'un **arbre bicolore** à n nœuds.

Alors : $h = O(\log n)$.

Preuve

Soit h_n la hauteur noire de l'arbre, et n_n son nombre de nœuds noirs. On a $\frac{n}{2} \leq n_n \leq n$ (à cause de l'alternance rouge/noir).

De plus, $h_n = \lfloor \log n_n \rfloor$ car, si on enlève tous les nœuds rouges, on obtient un arbre binaire complet à n_n nœuds et de hauteur h_n. Donc :

$$\lfloor \log(\frac{n}{2}) \rfloor \leq \lfloor \log n_n \rfloor = h_n \leq h \leq 2 * h_n = 2 * \lfloor \log n_n \rfloor \leq 2 * \lfloor \log(\frac{n}{2}) \rfloor$$

Donc $h = O(\log n)$.

Arbres bicolores

Implémentation des arbres bicolores

On va maintenant voir une **implémentation** des **arbres bicolores** en C.

Tout comme pour les **arbres AVL**, on implémente les opérations des **ABR**.

L'insertion ou la suppression d'un nœud va peut être "casser" le caractère **bicolore** de l'arbre, qu'il faudra alors **réparer**.

Complexité

Comme pour les **arbres AVL**, les **complexités** de toutes les opérations se feront en $O(h) = O(\log n)$.

Arbres bicolores

```c
#define NOIR 0
#define ROUGE 1
#define FEUILLE NULL
typedef value_t int;

typedef struct noeud noeud;
struct noeud {
  noeud *gauche; //Pointeur vers fils gauche
  noeud *droit; //Pointeur vers fils droit
  noeud *parent; //Pointeur vers père
  int couleur; // ROUGE ou NOIR
  value_t cle;
};
```

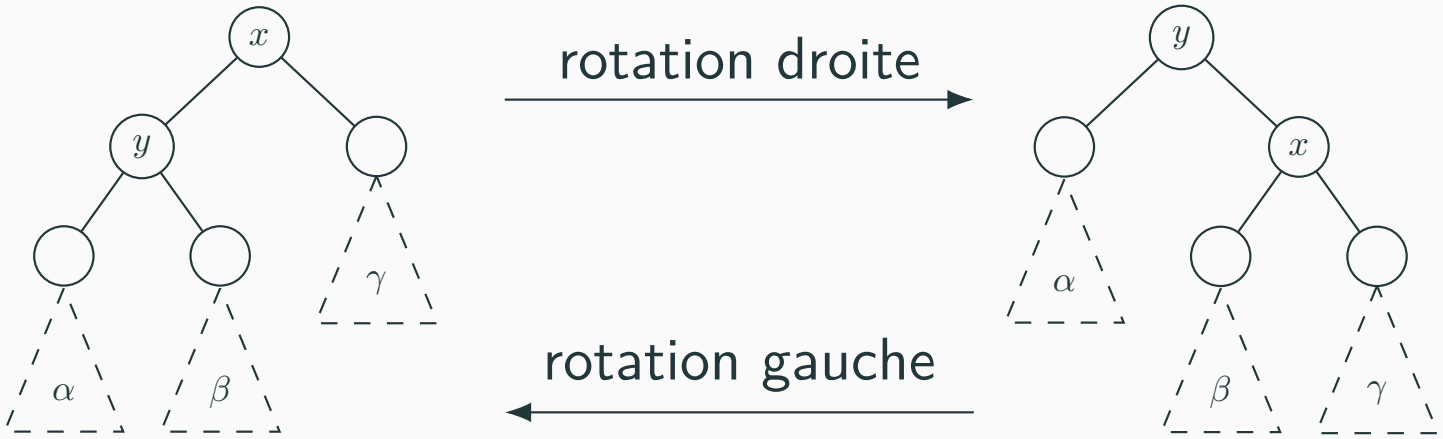
Arbres bicolores

On définit d'abord la structure d'**arbre bicolore**, et on implémente des fonctions permettant de **naviguer** dans l'arbre.

Arbres bicolores

```
1  noeud* parent(noeud* n) {
2    return n->parent;
3  }
4
5  noeud* grandparent(noeud* n) {
6    noeud* p = parent(n);
7    if (p == NULL)
8      return NULL; // Un noeud sans parent n'a pas de grand-parent
9    return parent(p);
10 }
11
12 noeud* frere(noeud* n) {
13   noeud* p = parent(n);
14   if (p == NULL)
15     return NULL; // Un noeud sans parent n'a pas de frere
16   if (n == p->gauche)
17     return p->droit;
18   else
19     return p->gauche;
20 }
21
22 noeud* oncle(noeud* enfant) {
23   noeud* p = parent(enfant);
24   noeud* g = grandparent(enfant);
25   if (g == NULL)
26     return NULL; // Pas de grand-parent, donc pas d'oncle
27   return frere(p);
28 }
```

Rotations d'arbres binaires



Rotations

Ensuite, on implémente les **rotations** d'arbres binaires qu'on a déjà vu sur pour les **arbres AVL**.

Rotations d'arbres binaires

```c
void rotation_droite(noeud *x) {
  noeud* y = x->gauche;
  // Transfert du fils du "milieu"
  x->gauche = y->droit;
  if (y->droit != FEUILLE) {
    y->droit->parent = x;
  }
  y->parent = x->parent;
  // Si x est la racine, y devient la racine
  if (x->parent == NULL) {
    x = y;
  }
  // Sinon, on remplace x par y
  else if (x == x->parent->droit) {
    x->parent->droit = y;
  }
  else {
    x->parent->gauche = y;
  }
  // On attache x à droite de y
  y->droit = x;
  x->parent = y;
}
```

```c
void rotation_gauche(noeud *x) {
  noeud* y = x->droit;
  // Transfert du fils du "milieu"
  x->droit = y->gauche;
  if (y->gauche != FEUILLE) {
    y->gauche->parent = x;
  }
  y->parent = x->parent;
  // Si x est la racine, y devient la racine
  if (x->parent == NULL) {
    x = y;
  }
  // Sinon, on remplace x par y
  else if (x == x->parent->gauche) {
    x->parent->gauche = y;
  }
  else {
    x->parent->droit = y;
  }
  // On attache x à gauche de y
  y->gauche = x;
  x->parent = y;
}
```

Recherche dans l'arbre

```
1   noeud *recherche(noeud *n, value_t x) {
2     noeud *current = n;
3     while (current != NULL && current->cle != x) {
4       if (current < x) {
5         current = current->droit;
6       }
7       else {
8         curent = current->gauche;
9       }
10    }
11    return current;
12  }
```

Recherche

La **recherche** d'un élément dans un **arbre bicolore** se fait comme d'habitude pour les **ABR**, on descend à gauche ou à droite selon la valeur rencontrée, jusqu'à tomber sur l'élément recherché ou arriver au bout de la branche suivie.

Recherche dans l'arbre

```
1   noeud *recherche(noeud *n, value_t x) {
2     noeud *current = n;
3     while (current != NULL && current->cle != x) {
4       if (current < x) {
5         current = current->droit;
6       }
7       else {
8         curent = current->gauche;
9       }
10    }
11    return current;
12  }
```

Complexité

Les **couleurs** des nœuds de l'arbre n'interviennent pas directement, mais la longueur de la branche (et donc la **complexité**) est garantie d'être en $O(\log n)$, où n est le nombre de nœuds dans l'arbre.

Insertion dans un arbre bicolore

Insertion

L'**insertion** commence comme pour un **ABR** classique : en partant de la racine, on descend de la même manière que pour la **recherche** jusqu'à arriver au bout de la branche suivie, et on insère le nouveau nœud.

Le nouveau nœud est initialement colorié en **rouge**.

Il faut ensuite vérifier si l'arbre est toujours un **arbre bicolore**, et dans le cas contraire, effectuer des **changements de couleurs** et des **rotations** pour retrouver un **arbre bicolore**.

Puisque la **racine** risque de changer (avec les rotations), on renvoie un **pointeur** vers la **nouvelle racine** de l'arbre.

Insertion dans un arbre bicolore

```
1   noeud *insertion(noeud *racine, noeud *n) {
2     // Insertion d'un nouveau noeud dans l'arbre
3     insertion_recursif(racine, n);
4
5     // Réparation de l'arbre au cas où les propriétés rouge-noir seraient violées
6     insertion_repare_arbre(n);
7
8     // Recherche de la nouvelle racine à renvoyer
9     racine = n;
10    while (parent(racine) != NULL) {
11      racine = parent(racine);
12    }
13    return racine;
14  }
```

Insertion

- `insertion_recursif` insère un nœud comme dans un **ABR**.
- `insertion_repare_arbre` rend l'arbre "**bicolore**".
- La boucle **`while`** trouve la nouvelle **racine**.

Insertion dans un arbre bicolore

```
1   void insertion_recursif(noeud *racine, noeud *n) {
2     // Descente récursive dans l'arbre jusqu'à atteindre une feuille
3     if (racine != NULL && n->cle < racine->cle) {
4       if (racine->gauche != FEUILLE) {
5         insertion_recursif(racine->gauche, n);
6         return;
7       }
8       else {
9         racine->gauche = n;
10      }
11    }
12    else if (racine != NULL) {
13      if (racine->droit != FEUILLE) {
14        insertion_recursif(racine->droit, n);
15        return;
16      }
17      else {
18        racine->droit = n;
19      }
20    }
21
22    // Insertion du nouveau noeud n
23    n->parent = racine;
24    n->gauche = FEUILLE; // NIL
25    n->droit = FEUILLE; // NIL
26    n->couleur = ROUGE;
27  }
```

Insertion dans un arbre bicolore

```c
void insertion_repare_arbre(noeud *n) {
  if (parent(n) == NULL)
     insertion_cas1(n);
  else if (parent(n)->couleur == NOIR)
     insertion_cas2(n);
  else if (oncle(n)->couleur == ROUGE)
     insertion_cas3(n);
  else
     insertion_cas4(n);
}
```

```c
void insertion_cas1(noeud *n) {
  if (parent(n) == NULL) {
    n->couleur = NOIR;
  }
}
```

```c
void insertion_cas2(noeud *n) {
  return; /* Ne rien faire */
}
```

Réparation

Il y a plusieurs cas possibles pour rétablir les propriétés de l'**arbre bicolore**.

1. Si le nœud n'a pas de parent : c'est la racine de l'arbre. Dans ce cas, il faut colorier le nœud en noir pour respecter la propriété 2.
2. Si le **parent** du nœud inséré est **noir**, il n'y a rien à faire.

Insertion dans un arbre bicolore

Réparation

3. Le parent (P) du nouveau nœud est **rouge** ; alors la propriété 3 est invalide.

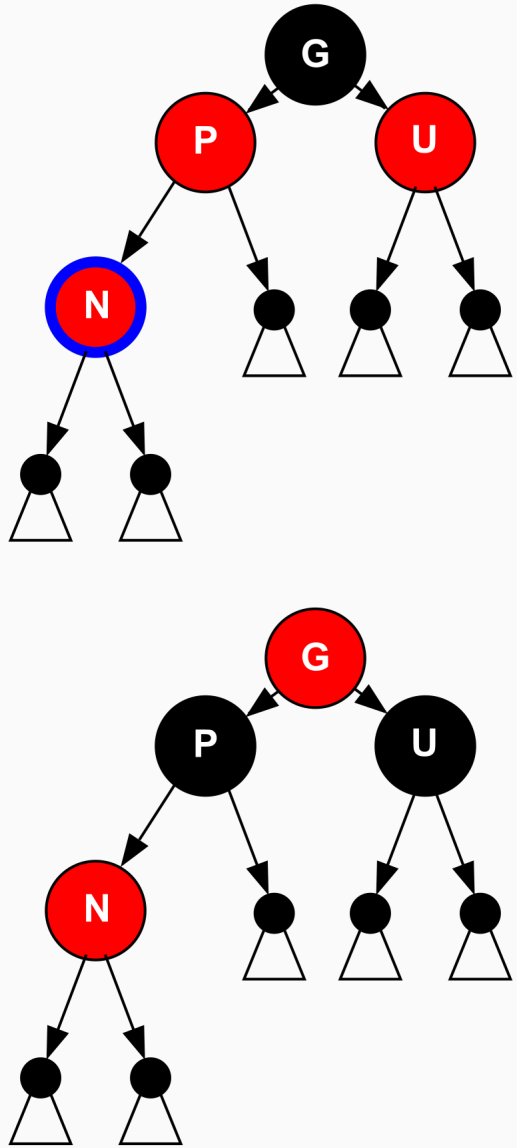
   L'action à effectuer dépend de la couleur de l'oncle (U) du nœud (N).

   ↪ Si (U) est également **rouge**, alors on colorie (P) et (U) en **noir**, et le grand-parent (G) (qui était nécessairement **noir**) est colorié en **rouge**.

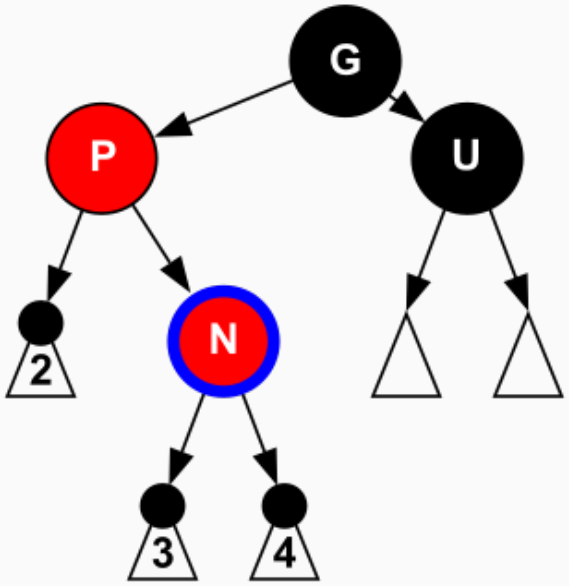
   Il faut alors continuer la réparation en partant de (G) (car son changement de couleur a pu violer d'autres propriétés).

Insertion dans un arbre bicolore

```
1  void insertion_cas3(noeud *n) {
2    parent(n)->couleur = NOIR;
3    oncle(n)->couleur = NOIR;
4
5    noeud *g = grandparent(n);
6    g->couleur = ROUGE;
7    insertion_repare_arbre(g);
8  }
```



Insertion dans un arbre bicolore

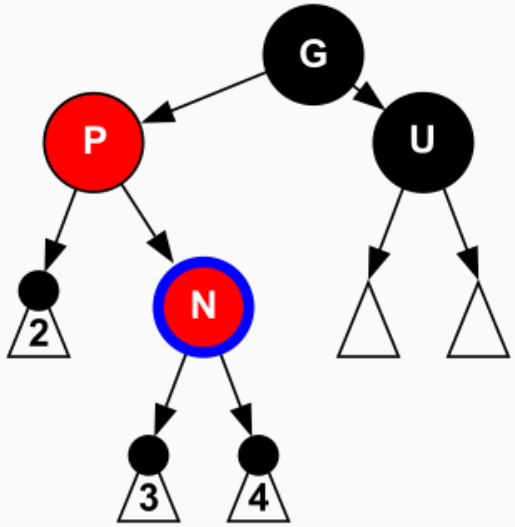


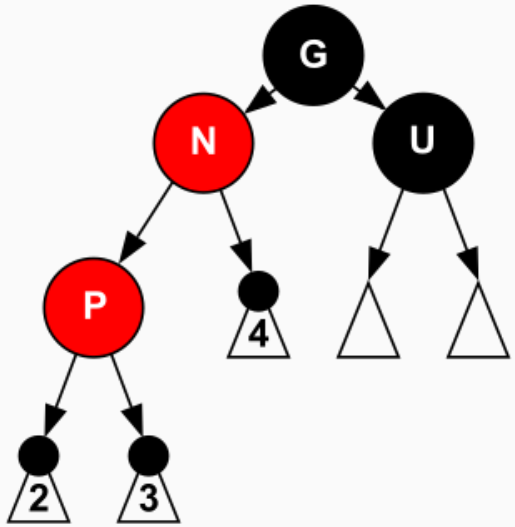
Réparation

4. Si l'oncle est **noir**, et s'il faut faire un "**zig-zag**" pour remonter au grand-parent, on effectue d'abord une **rotation** au niveau du parent (P) pour enlever ce "zig-zag", et on continue la réparation avec (P), où l'on est maintenant dans le cas "**sans zig-zag**".

Insertion dans un arbre bicolore

```
1   void insertion_cas4(noeud *n) {
2     noeud *p = parent(n);
3     noeud *g = grandparent(n);
4
5     if (n == g->gauche->droit) {
6       rotation_gauche(p);
7       n = n->gauche;
8     }
9     else if (n == g->droit->gauche) {
10      rotation_droit(p);
11      n = n->droit;
12    }
13
14    insertion_cas5(n);
15  }
```





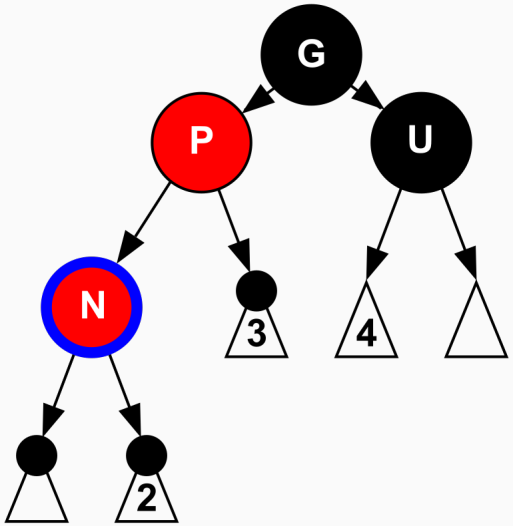
Insertion dans un arbre bicolore

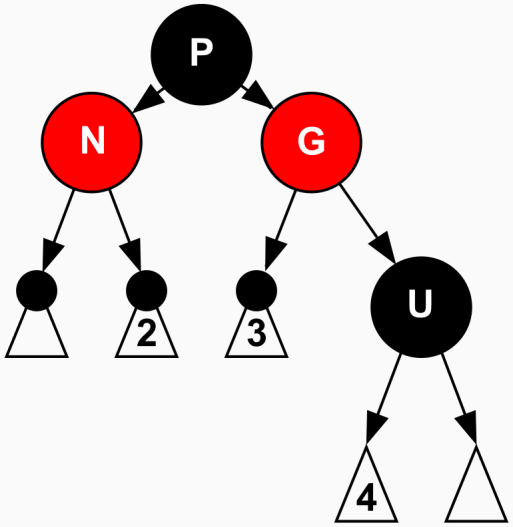
Réparation

5. Si l'oncle (U) est **noir**, et qu'il n'y a pas (ou plus) de "**zig-zag**" pour remonter au grand-parent (G) du nœud considéré, alors on effectue une rotation au niveau de (G) pour qu'il descende du côté de (U), puis on colorie (G) en **rouge** et (U) en **noir**.

Insertion dans un arbre bicolore

```
1   void insertion_cas5(node *n) {
2     noeud *p = parent(n);
3     noeud *g = grandparent(n);
4
5     if (n == p->gauche) {
6       rotation_droit(g);
7     }
8     else {
9       rotation_gauche(g);
10    }
11
12    p->couleur = NOIR;
13    g->couleur = ROUGE;
14  }
```





Insertion dans un arbre bicolore

Correction

Le seul cas où la **réparation** ne se **termine** pas immédiatement est le cas 3, dans lequel on change le grand-parent de noir à rouge, ce qui oblige à effectuer une nouvelle vérification en partant du grand-parent.

Cependant, il est aisé de vérifier que la fonction se **termine** toujours.

Puisque le nœud à vérifier est toujours **strictement plus haut** que le précédent, on finira inévitablement par se retrouver dans l'un des cas non récursifs (dans le pire des cas, on remontera jusqu'à atteindre la racine de l'arbre, c'est-à-dire le cas 1).

Insertion dans un arbre bicolore

Complexité

Il y aura donc au plus deux rotations, et un nombre de changements de couleurs inférieur à la moitié de la hauteur de l'arbre, c'est-à-dire en $O(\log n)$.

En pratique la probabilité de tomber plusieurs fois de suite sur le cas 3 est exponentiellement décroissante ; en moyenne le coût de la correction des propriétés est donc presque **constant**.

Suppression dans un arbre bicolore

Suppression

On commence la suppression comme dans un **ABR** classique :

- on cherche le nœud à supprimera (S), puis on cherche le plus grand élément de son sous-arbre gauche (M) ;
- on remplace l'étiquette de (S) par celle de (M) (sans changer la couleur de (S) ;
- on souhaite alors supprimer (M) de l'arbre.

Suppression dans un arbre bicolore

Suppression

Puisque (M) est le plus grand élément du sous-arbre gauche, son **fils droit** est une feuille (`NULL`).

Notons (C) sont **fils gauche** (qui est peut-être une feuille `NULL`).

(M) va être remplacé par (C).

La réparation dépend alors des **couleurs** de (M) et de (C).

Suppression dans un arbre bicolore

Suppression

On discute ici seulement quelques cas :

- Si (M) est **rouge**, alors (C) est forcément **noir**, et il n'y a pas de problème : la hauteur noire de change pas.
- si (M) est **noir** et (C) est **rouge**, alors il suffit de colorier (C) en **rouge**.
- Le cas le plus compliqué se produit si (M) et (C) sont tous les deux **noirs**.