Epita:Algo:Cours:Info-Sup:Arbres de recherche

De EPITACoursAlgo.

Dans les méthodes de base, nous avons vu que l'on pouvait obtenir des temps de recherche logarithmique sur des structures séquentielles en représentation contiguë. L'idéal serait de pouvoir obtenir les mêmes performances sur des structures dynamiques. Cela n'est pas possible avec des structures séquentielles, en revanche cela l'est avec les structures arborescentes, donc...

Sommaire

les arbres binaires de recherche (ABR)

N'importe quelle collection de ***n*** éléments dont les clés appartiennent à un ensemble ordonné peut être stockée et classée à l'intérieur d'un arbre binaire de ***n*** noeuds. Dans ce cas, les noeuds contiennent les éléments et les liens père-fils permettent de gérer l'ordre entre ces éléments.

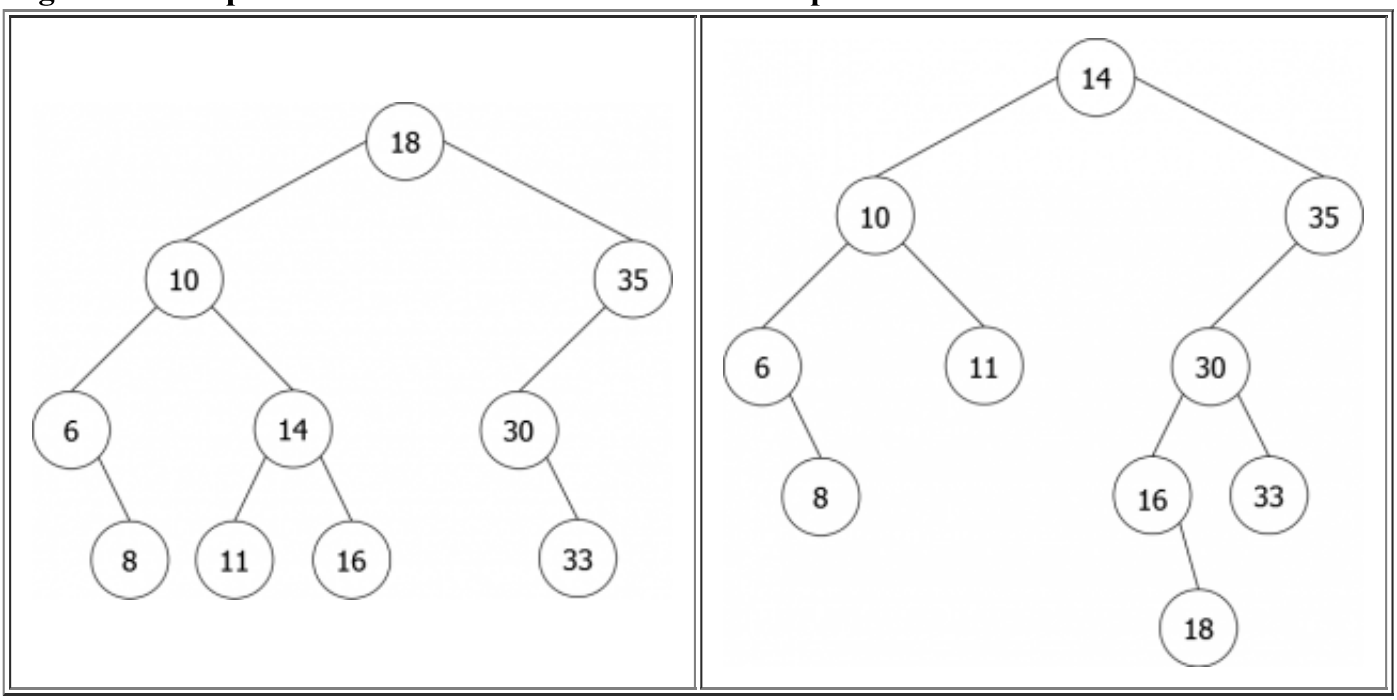
Un ***ABR*** est un arbre binaire étiqueté tel qu'en tout noeud ***v*** de l'arbre:

- les éléments du sous-arbre gauche de l'arbre de racine ***v*** sont inférieurs ou égaux à celui contenu dans ***v***,

- les éléments du sous-arbre droit de l'arbre de racine ***v*** sont strictement supérieurs à celui contenu dans ***v***.

Il peut y avoir plusieurs ABRs représentant un même ensemble de données, comme le montre l'exemple suivant (cf. figure 1), pour l'ensemble d'entiers ***E = {6,8,10,11,14,16,18,30,33}***.

Figure 1. Exemple de deux arbres binaires de recherche possibles d'un même ensemble d'entiers.



Recherche dans un ABR

L'opération de recherche d'un élément ***x*** dans un ABR ***B*** définie abstraitement par :

```
opérations

   rechercheABR : element x arbrebinaire ⟶ booléen
```

répond au principe suivant :

```
- si B est un arbre vide, la recherche est négative
- si x est égal à l'élément de la racine de B, la recherche est positive
- si x est inférieur à l'élément de la racine de B, la recherche se poursuit sur le sous-arbre g
- si x est supérieur à l'élément de la racine de B, la recherche se poursuit sur le sous-arbre d
```

L'algorithme correspondant à ce principe serait le suivant :

```
algorithme fonction rechercherABR : booléen
Paramètres locaux
   élément  x
   arbrebinaire  B
Début
   si B = arbrevide alors
      retourne(Faux)                          /* recherche négative : échec */
```

```
   sinon
      si x=contenu(racine(B) alors
         retourne(Vrai)                    /* Recherche positive : succès */
      sinon
         si x<contenu(racine(B) alors
            retourne(chercherABR(x,g(B)))  /* poursuite dans sous-arbre gauche */
         sinon
            retourne(chercherABR(x,d(B)))  /* poursuite dans sous-arbre droit */
         fin si
      fin si
   fin si
fin algorithme fonction rechercherABR
```

Ajout dans un ABR

L'opération d'ajout d'un élément ***x*** dans un ABR ***B*** peut se faire de deux manières, *en feuille* ou *en racine de l'arbre* **B**.

Ajout en feuille

L'opération d'ajout d'un élément ***x*** en feuille d'un ABR ***B*** définie abstraitement par :

```
opérations

   ajouterfeuilleABR : element x arbrebinaire ⟶ arbrebinaire
```

répond au principe suivant :

```
- déterminer la place d'adjonction
- réaliser l'adjonction
```

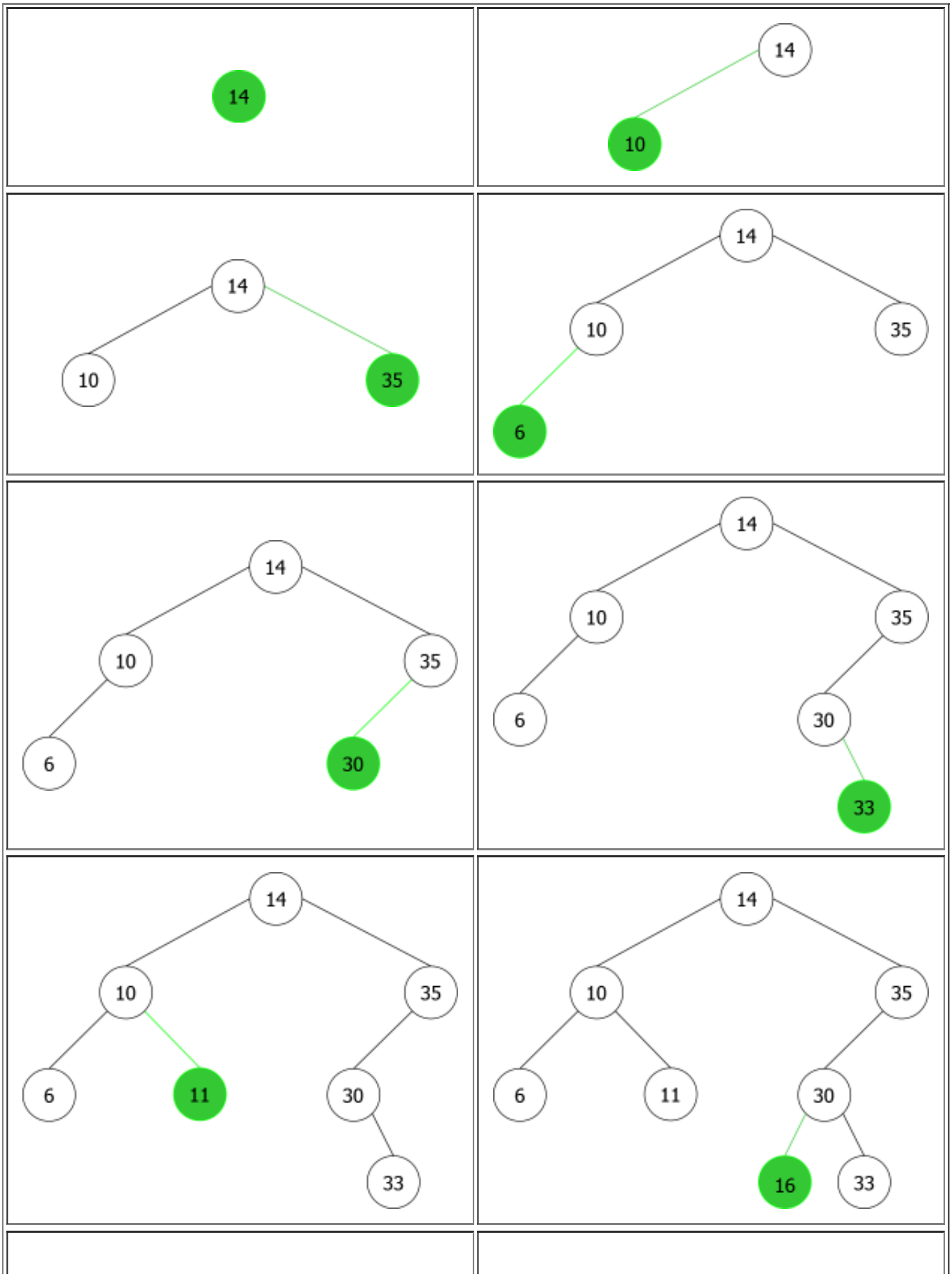
La première partie se fait sur le même principe que la recherche, le dernier appel récursif se terminant sur un arbre vide. Il ne reste plus ensuite qu'à créer un nouveau noeud (une nouvelle feuille) contenant l'élément ***x*** à ajouter. Ce qui pourrait donner l'algorithme suivant :

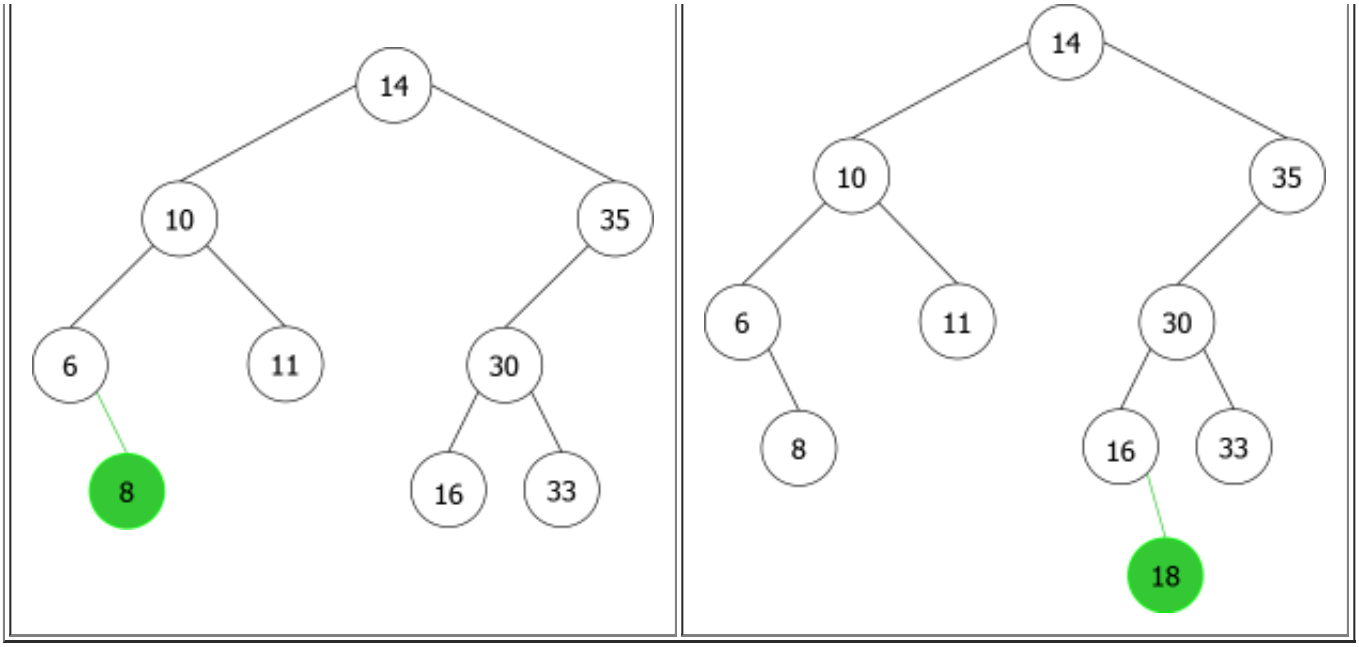
```
algorithme fonction ajouterfeuilleABR : arbrebinaire
Paramètres locaux
   élément  x
   arbrebinaire  B
Variables
   noeud  r
Début
   si B = arbrevide alors
      contenu(r) ⟵ x
      retourne(<r,arbrevide,arbrevide>)  /* retour de l'arbre réduit au noeud créé */
   sinon
      si x <= contenu(racine(B) alors
         retourne(<racine(B),ajouterfeuilleABR(x,g(B)),d(B)>)   /* retour de l'arbre avec ajout
      sinon
         retourne(<racine(B),g(B),ajouterfeuilleABR(x,d(B))>)   /* retour de l'arbre avec ajout
      fin si
   fin si
fin algorithme fonction ajouterfeuilleABR
```

Construction d'un ABR par ajouts successifs en feuille

En appliquant l'algorithme précédent pour les entiers suivants : ***14, 10, 35, 6, 30, 33, 11, 16, 8, 18*** nous aurions la séquence de construction suivante d'ABRs:

Figure 2. Construction d'un ABR par ajouts successifs des valeurs *14, 10, 35, 6, 30, 33, 11, 16, 8 et 18*.

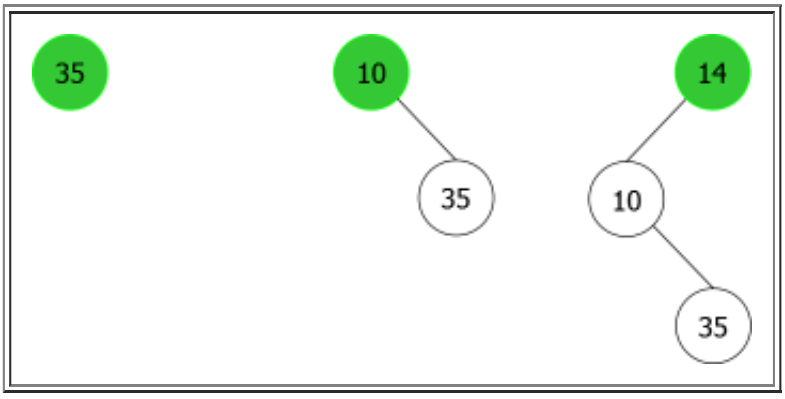




Ajout en racine

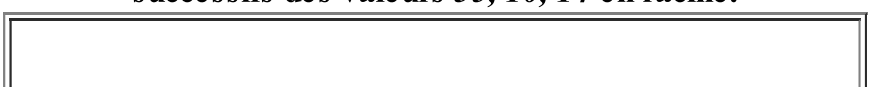
Le problème est que l'on ne peut pas se contenter simplement d'ajouter le nouveau noeud racine contenant l'élément ***x*** en se basant sur l'élément de l'actuelle racine. Ce qui positionnerait cette dernière comme fils gauche ou droit de la nouvelle racine. En effet, cela ne permettrait pas de respecter à coup sûr la relation d'ordre, comme le montre la séquence d'ajouts en racine des éléments ***{35, 10, 14}*** (cf. Figure 3).

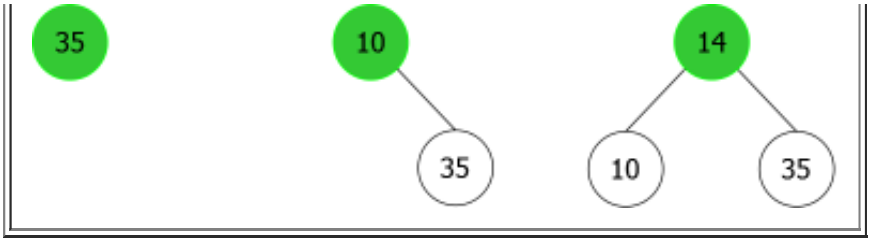
Figure 3. Mauvaise construction d'un ABR par ajouts successifs des valeurs *35, 10, 14* en racine.



On voit bien sur cet exemple qu'au troisième ajout en racine (celui de ***14***), le ***35*** se retrouve dans le sous-arbre gauche ce qui n'est pas possible. En effet, ***35*** est bien supérieur à ***10***, mais aussi à ***14***, il devrait donc se situer dans le sous-arbre droit de l'arbre de racine ***14***, ce que montre la figure 4.

Figure 4. Construction correcte d'un ABR par ajouts successifs des valeurs *35, 10, 14* en racine.



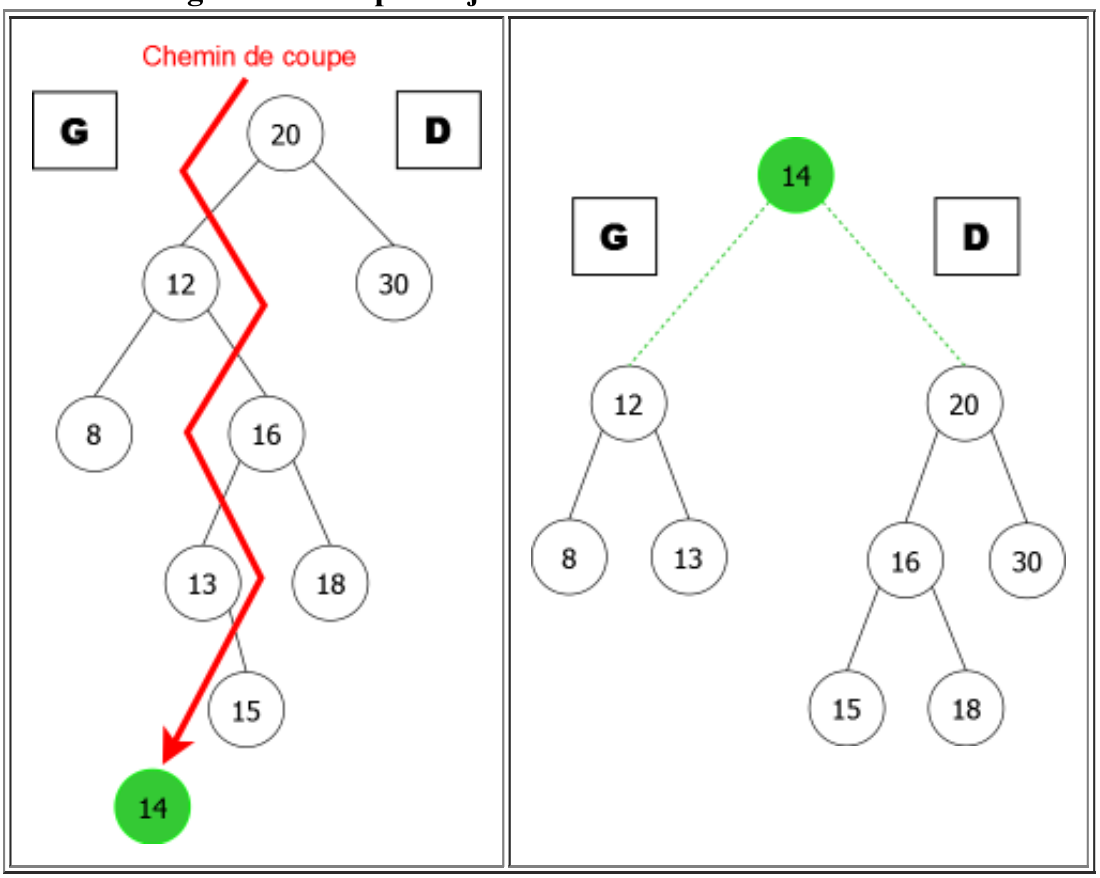


Pour régler ce problème, il faudrait répartir dans les sous-arbres gauche (appelé ***G***) et droit (appelé ***D***) du nouvel arbre créé par l'ajout en racine de l'élément ***x***, les éléments existants de ***B*** selon qu'ils sont inférieurs ou supérieurs à ***x***.

On va alors suivre le chemin (appelé ***chemin de coupe***) allant de la racine de l'arbre ***B*** jusqu'à la nouvelle feuille accueillant ***x*** si l'ajout se faisait en feuille. A chaque noeud rencontré, on va regarder si l'élément de celui-ci est inférieur(resp. supérieur) à ***x***. S'il est inférieur (resp. supérieur), il ira dans ***G***(resp. ***D***) accompagné implicitement de son propre sous-arbre gauche(resp. droit) cont enant des éléments qui lui sont plus petits(resp. grands) donc implicitement inférieurs(resp. supérieurs) à ***x***.

Il ne reste plus ensuite qu'à accrocher ***G*** et ***D*** à la nouvelle racine ***x***, comme le montre l'exemple de la figure 5.

Figure 5. Exemple d'ajout en racine de *14* sur un ABR..



L'opération d'ajout d'un élément ***x*** en racine d'un ABR ***B*** définie abstraitement par :

```
opérations

   ajouterracineABR : element x arbrebinaire ⟶ arbrebinaire
```

va nous permettre d'ajouter l'élément ***x*** dans une nouvelle racine de l'arbre ***B*** et non pas dans une nouvelle feuille. L'utilité de cet ajout est de pouvoir, tout en respectant la relation d'ordre, insérer un nouvel élément n'importe où dans un ABR (pas seulement à la racine). Elle répond au principe suivant :

```
- Créer le nouveau noeud contenant x
- couper B en deux arbres G et D selon le chemin de coupe
- accrocher G et D au noeud contenant x
```

Nous allons donc avoir besoin d'une procédure réalisant la coupe de l'ABR ***B*** en deux sous-arbres ***G*** et ***D*** selon la valeur de ***x***. Ce qui donne l'algorithme suivant :

```
algorithme procedure couper
Paramètres locaux
   élément  x
Paramètres globaux
   arbrebinaire  B, G, D
Début
   si B = arbrevide alors   /* fin de récursion et fermeture des liens G et D */
      G ⟵ arbrevide
      D ⟵ arbrevide
   sinon
      si x < contenu(racine(B)) alors
         D ⟵ B
         couper(x, g(B), G, g(D))  /* récursion sur le sous-arbre gauche et mise en attente du f
      sinon
         G ⟵ B
         couper(x, d(B), d(G), D)  /* récursion sur le sous-arbre droit et mise en attente du fi
      fin si
   fin si
fin algorithme procedure couper
```

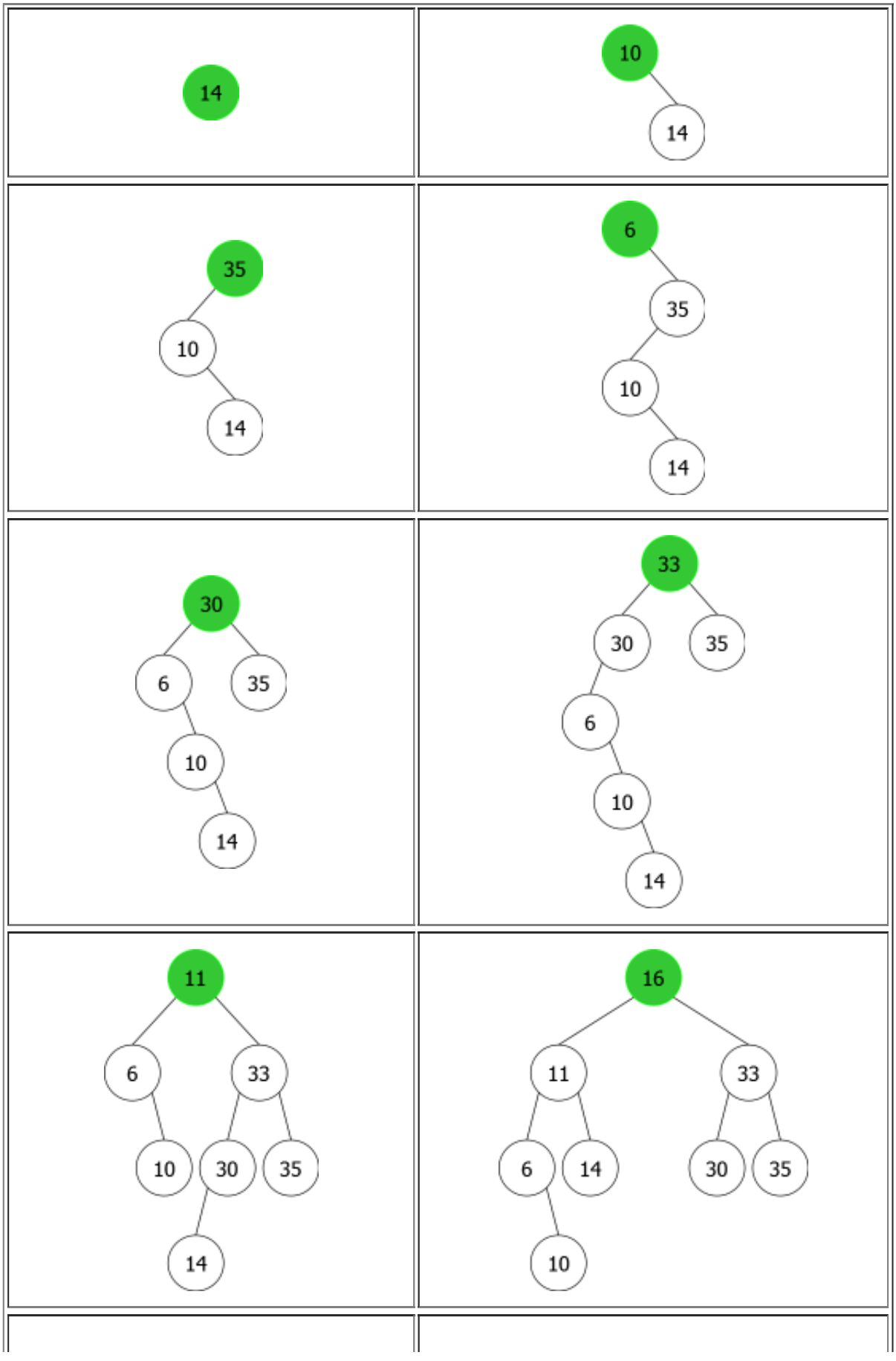
Il est à noter que cette version de l'algorithme d'ajout en racine (procédure **couper** comprise), ne construit pas les arbres ***G*** et ***D*** pour les accrocher ensuite, mais elle les construit au fur et à mesure qu'elle parcourt le chemin de coupe. Dans ce cas, l'appel de la procédure **couper** se fait directement sur les fils gauche et droit du noeud créé comme nouvelle racine. La procédure d'appel **ajouterracineABR** pourrait alors être la suivante:

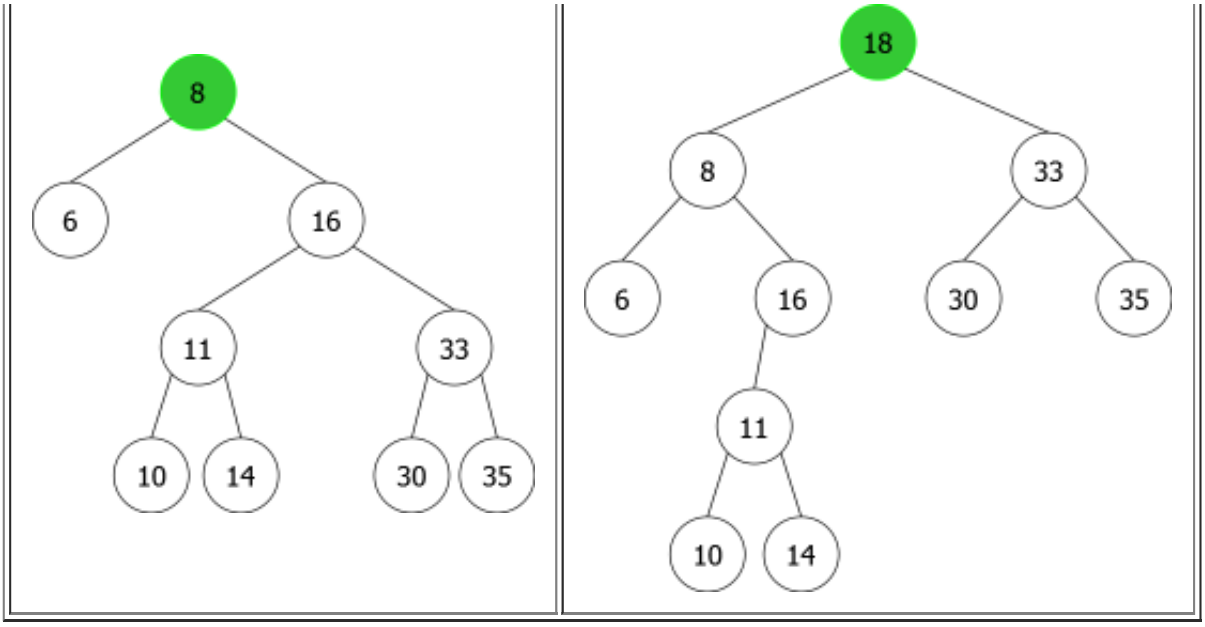
```
algorithme procedure ajouterracineABR
Paramètres locaux
   élément  x
Paramètres globaux
   arbrebinaire  B
Variables
   Arbrebinaire  R
Début
   contenu(racine(R)) ⟵ x
   couper(x, B, g(R), d(R))   /* appel de couper avec les fils de R pour G et D */
   B ⟵ R
fin algorithme procedure ajouterracineABR
```

Construction d'un ABR par ajouts successifs en racine

En appliquant l'algorithme précédent pour les entiers suivants : ***14, 10, 35, 6, 30, 33, 11, 16, 8, 18*** nous aurions la séquence de construction suivante d'ABRs:

Figure 6. Construction d'un ABR par ajouts successifs des valeurs *14, 10, 35, 6, 30, 33, 11, 16, 8 et 18*.





Suppression dans un ABR

L'opération de suppression d'un élément ***x*** dans un ABR ***B*** définie abstraitement par :

```
opérations

   supprimerABR : element x arbrebinaire ⟶ arbrebinaire
```

répond au principe suivant :

```
- déterminer la place de l'élément à supprimer
- réaliser la suppression
- réorganiser éventuellement l'arbre
```

La première partie se fait sur le même principe que la recherche. Il ne reste plus ensuite qu'à déterminer à quel type de noeud (contenant l'élément ***x*** à supprimer) nous avons affaire. En effet, celui-ci peut être de trois types différents avec les implication suivantes :

1. Le noeud contenant ***x*** est une feuille comme pour la suppression de ***14*** sur figure 7(a). Dans ce cas, le noeud est simplement détruit et l'arbre obtenu est celui de la figure 7(b).
2. Le noeud contenant ***x*** est un point simple, comme pour la suppression de ***38*** (point simple à droite) sur figure 8(a). Dans ce cas, le noeud est simplement détruit et remplacé par son fils (droit dans le cas présent). L'arbre obtenu est celui de la figure 8(b) ou le noeud ***40*** est venu remplacer le ***38***.
3. Le noeud contenant ***x*** est un point double, comme pour la suppression de ***18*** (racine de l'arbre) sur figure 9(a). Dans ce cas, deux possibilités s'offrent à nous pour ne pas avoir à reconstruire complètement l'arbre:
   - Remplacer l'élément du noeud par son immédiat inférieur (bout du bord droit du sous-arbre gauche), le noeud ***16*** sur cet exemple.
   - Remplacer l'élément du noeud par son immédiat supérieur (bout du bord gauche du sous-arbre droit), le noeud ***30*** sur cet exemple.

L'intérêt de cette méthode est double

- une fois le remplacement fait, il n'y a plus qu'à détruire le noeud remplaçant (***16*** ou ***30*** sur cet exemple) qui ne peut-être qu'une feuille ou un point simple
- la relation d'ordre est conservée aisément dans la mesure où le plus grand des plus petits est toujours inférieur au plus petit des plus grands et réciproquement.

Sur l'exemple de la figure 9, nous avons choisi de le remplacer par son immédiat inférieur (le plus grand des plus petits). L'arbre obtenu est celui de la figure 9(b) ou le noeud ***16*** est venu remplacer le ***18*** et a été détruit et remplacé simplement par son fils gauche ***11*** comme dans le cas 2 (point simple).

Figure 7. Suppression du noeud *14* (feuille).

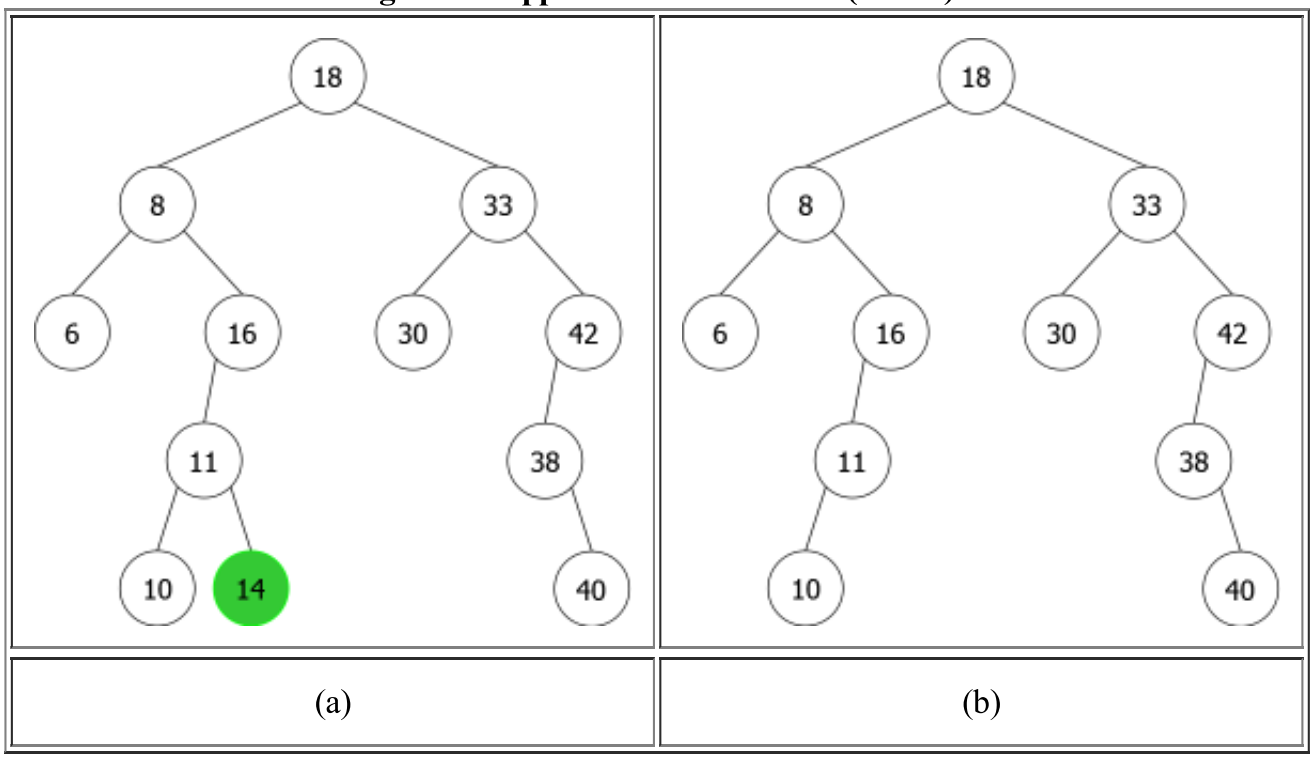
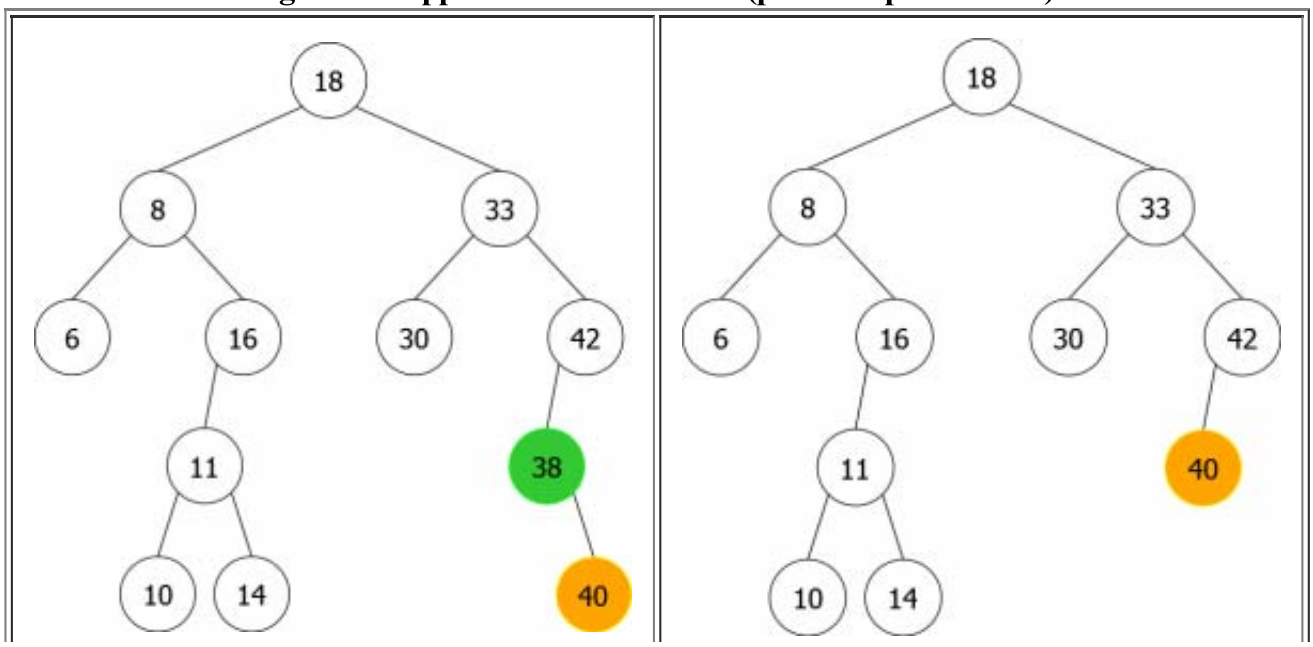
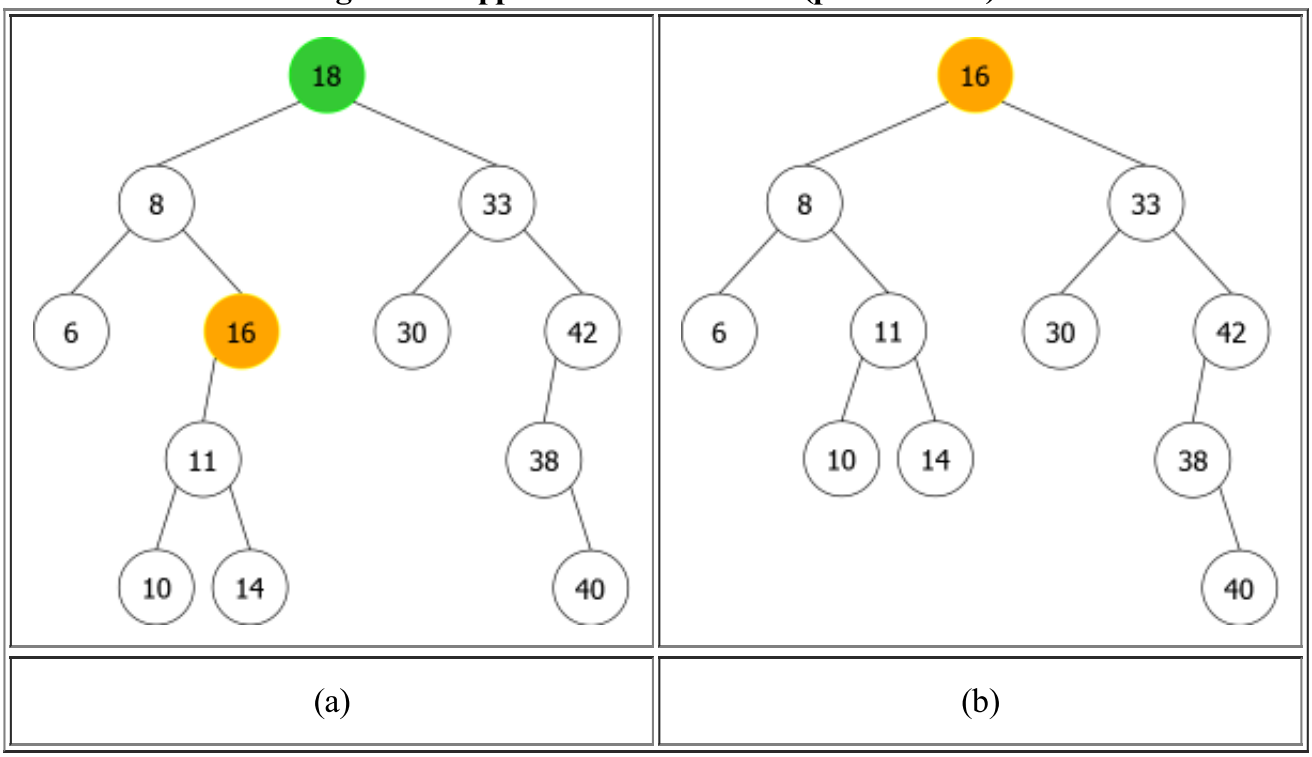


Figure 8. Suppression du noeud *38* (point simple à droite).



(a)	(b)

Figure 9. Suppression du noeud *18* (point double).



(a)	(b)

Remarque: Ces deux solutions sont équivalentes s'il n'y a pas de redondance de valeurs dans l'ABR.

Nous allons avoir besoin de deux opérations supplémentaires pour pouvoir répondre au problème de la suppression d'un point double. Les deux opérations sont définies abstraitement par :

```
opérations

    max : arbrebinaire ⟶ element
   dmax : arbrebinaire ⟶ arbrebinaire
```

Elles répondent aux principes suivants :

```
- max(B) retourne le plus grand élément de l'ABR B
- dmax(B) retourne l'ABR B amputé de son plus grand élément
```

Nous allons les réunir en une seule opération *suppmax*, ce qui pourrait donner l'algorithme suivant :

```
algorithme procédure suppmax
Paramètres globaux
   élément  max
   arbrebinaire  B
Début
   si d(B) = arbrevide alors       /* pas de fils droit, nous sommes sur le plus grand élément de
      max ⟵ contenu(racine(B))
      B ⟵ g(B)
```

```
   sinon
      suppmax(max, d(B))          /* poursuite à droite */
   fin si
fin algorithme procédure suppmax
```

Il ne reste plus alors qu'à écrire l'algorithme de la procédure de suppression d'un élément ***x*** dans un ABR ***B***, ce qui pourrait donner l'algorithme suivant :

```
algorithme procédure supprimerABR
Paramètres locaux
   élément  x
Paramètres globaux
   arbrebinaire  B
Variables
   élément max
Début
   si B <> arbrevide alors
      si x < contenu(racine(B)) alors
         supprimerABR(x, g(B))
      sinon
         si x > contenu(racine(B)) alors
            supprimerABR(x, d(B))
         sinon       /* x = contenu(racine(B)), x est trouvé => destruction. */
            si g(B) = arbrevide alors /* B est un point simple à droite ou une feuille */
               B ← d(B)
            sinon
               si d(B) = arbrevide alors /* B est un point simple à gauche */
                  B ← g(B)
               sinon  /* B est un point double => appel à suppmax */
                  suppmax(max, g(B))
                  contenu(racine(B)) ← max
               fin si
            fin si
         fin si
      fin si
   fin si
fin algorithme procédure supprimerABR
```

Pour cet algorithme, on a décidé arbitrairement de faire la recherche et ensuite de gérer la suppression lorsque l'élément est trouvé. Nous aurions tout aussi bien pu faire l'inverse. Cela ne change en rien la complexité de cet algorithme et dans le deuxième cas, il se rapprocherait plus du schéma de l'algorithme de recherche qui effectivement traite le "*trouvé*" avant le "*essaie encore*".

Analyse du nombre de comparaisons (complexité des précédents algorithmes)

Les algorithmes de recherche, d'adjontion et de suppression procèdent par comparaisons de valeurs (***x*** = *contenu(racine(**B**))*, ***x*** < *contenu(racine(**B**))*, ***x*** > *contenu(racine(**B**))*). La comparaison entre deux éléments est l'opération fondamentale, celle qui va nous permettre de faire l'analyse de ces algorithmes et de déterminer leurs performances. Or le nombre de comparaisons effectuées entre deux valeurs par ces algorithmes peut être lu directement sur l'ABR manipulé, en effet :

- pour la recherche:
  - si elle se termine positivement sur un noeud ***v***, le nombre de comparaisons effectuées est **2*profondeur(v)+1**
  - si elle se termine négativement après le noeud ***v***, le nombre de comparaisons effectuées est **2*profondeur(v)+2**

- pour l'ajout:
  - en feuille: la recherche de position se termine toujours négativement après un noeud ***v***. Mais le nombre de comparaisons est divisé par deux puisque l'on se demande si ***x***<=*contenu(racine(**B**))*. Le nombre de comparaisons effectuées est alors **profondeur(v)+1**
  - en racine: le nombre de comparaisons est le même puisque l'algorithme compare l'élément à insérer à ceux du chemin de recherche de l'ajout en feuille (*chemin de coupe*). Le nombre de comparaisons effectuées est donc là aussi **profondeur(v)+1**

- pour la suppression: Le nombre de comparaisons est identique à celui de l'algorithme de recherche. Cela étant, si l'on tient compte du temps de recherche de l'élément de remplacement (cas de la suppression d'un point double), il faut ajouter la complexité de la procédure *suppmax*.

Conclusion: L'analyse, basée sur le nombre de comparaisons, des algorithmes de recherche, d'adjonction et de suppression d'un élément dans un arbre binaire de recherche se ramène à l'étude de la profondeur des noeuds dans l'ABR. La profondeur moyenne d'un ABR de *n* éléments est comprise entre *n* si l'arbre est dégénéré et log_2^n s'il est complet ou parfait (cf. Arbres binaires particuliers).

Le problème est que les algorithmes d'ajout et de suppression ne garantissent pas l'équilibre de l'ABR et que dans ce cas il ne nous reste plus qu'à miser sur la chance ou à modifier nos algorithmes pour que ceux-ci nous assurent l'équilibrage des arbres de recherche. C'est cette deuxième solution que nous allons retenir et dont traite le chapitre suivant.

Les arbres de recherche équilibrés

Dans le chapitre précédent, nous avons vu que la complexité des opérations de recherche, d'ajout et de suppression pouvait être au pire linéaire. Cela est du au fait qu'un arbre binaire n'est pas nécessairement équilibré, et que dans le pire des cas, il peut être filiforme. Imaginez un ajout aux feuilles avec comme entrées successives la liste {a,b,c,d,e,f}.

Pour conserver, cette complexité logarithmique, nous devons donc équilibrer l'arbre. Dans ce cas, il est clair que la réorganisation de celui-ci doit être la plus rapide possible. Ce qui impliquera un léger déséquilibre de l'arbre. Nous allons étudier deux classes d'arbres équilibrés, les A-V.L. (arbre binaires) et les Arbres 2.3.4 (arbres généraux). Pour chacune de ces classes, les complexités des algorithmes de recherche, d'ajout et de suppression seront logarithmiques.

Rotations

Dans un premier temps, nous allons implémenter des algorithmes de rééquilibrage appelés rotations. Ils effectuent des transformations locales de l'arbre et sont au nombre de quatre : rotation simples (gauche et droite) ; rotation double (gauche-droite et droite-gauche). Des exemples de ces rotations sont montrés en figures 10, 11, 12 et 13.

Figure 10. Rotation droite de l'arbre B.

Figure 11. Rotation gauche de l'arbre B.

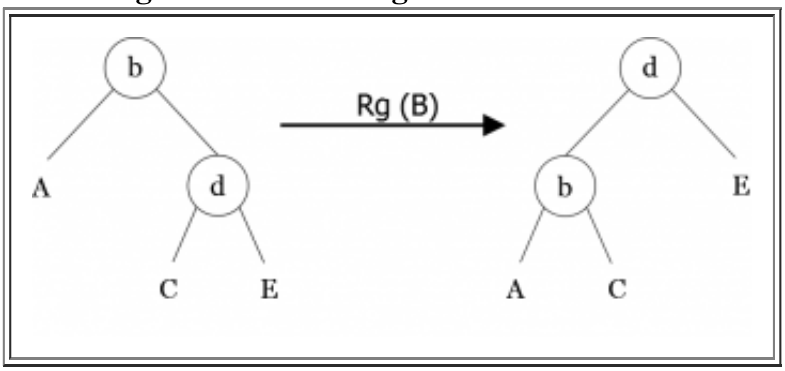


Figure 12. Rotation gauche-droite de l'arbre B.

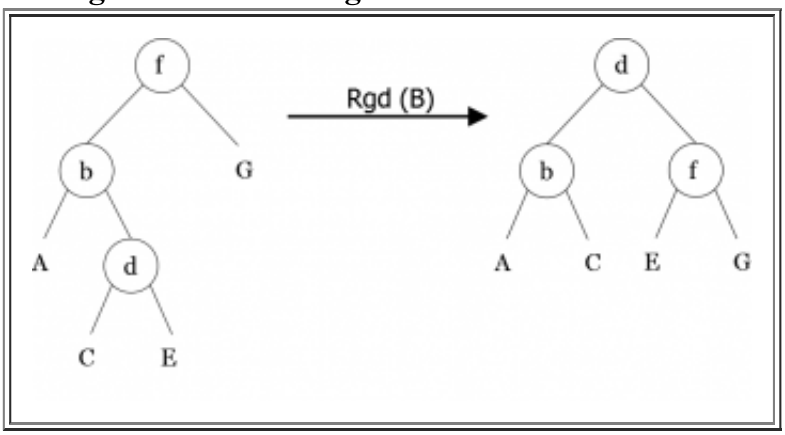
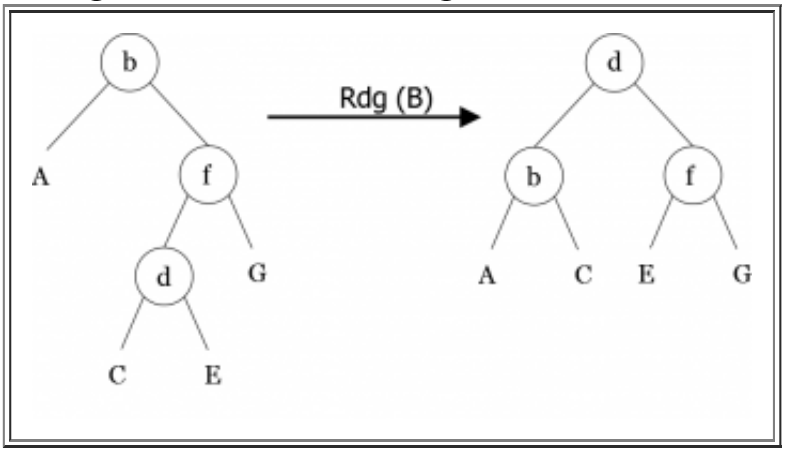


Figure 13. Rotation droite-gauche de l'arbre B.



Spécification formelle

Ces quatre opérations (Rg, Rd, Rgd et Rdg) sont définies de la manière suivante :

```
opérations

   rg  : arbrebinaire ⟶ arbrebinaire
   rd  : arbrebinaire ⟶ arbrebinaire
   rgd : arbrebinaire ⟶ arbrebinaire
   rdg : arbrebinaire ⟶ arbrebinaire

préconditions

   rg(C) est-défini-ssi C ≠ arbrevide & d(C) ≠ arbrevide
   rd(C) est-défini-ssi C ≠ arbrevide & g(C) ≠ arbrevide
   rgd(C) est-défini-ssi C ≠ arbrevide & g(C) ≠ arbrevide & d(g(C)) ≠ arbrevide
   rdg(C) est-défini-ssi C ≠ arbrevide & d(C) ≠ arbrevide & g(d(C)) ≠ arbrevide

axiomes

  rg(< b, A, < d, C, E > >) = < d, < b, A, C >, E >
  rd(< d, < b, A, C >, E >)= < b, A, < d, C, E > >
  rgd(< f, < b, A, < d, C, E > >, G >)= < d, < b, A, C >, < f, E, G > >
  rdg(< b, A, < f, < d, C, E >, G > >)= < d, < b, A, C >, < f, E, G > >

avec

  arbrebinaire A, C, E, G
  noeud        b, d, f
```

Algorithme (Procédure rg)

```
algorithme procédure Rg
Paramètres globaux
   arbrebinaire  B
Variables
   arbrebinaire  Aux
Début
   Aux ⟵ d(B)          /* Affectations des différents liens */
   d(B) ⟵ g(Aux)
   g(Aux) ⟵ B
   B ⟵ Aux
fin algorithme procédure Rg
```

Algorithme (Procédure rd)

```
algorithme procédure Rd
Paramètres globaux
   arbrebinaire  B
Variables
   arbrebinaire  Aux
Début
   Aux ⟵ g(B)          /* Affectations des différents liens */
   g(B) ⟵ d(Aux)
   d(Aux) ⟵ B
   B ⟵ Aux
fin algorithme procédure Rd
```

Algorithme (Procédure rgd)

```
algorithme procédure Rgd
Paramètres globaux
   arbrebinaire  B
Variables
   arbrebinaire  Aux
Début
   Rg(g(B))              /* Appel des rotations simples */
   Rd(B)
fin algorithme procédure Rgd
```

Algorithme (Procédure rdg)

```
algorithme procédure Rdg
Paramètres globaux
   arbrebinaire  B
Variables
   arbrebinaire  Aux
Début
   Rd(d(B))              /* Appel des rotations simples */
   Rg(B)
fin algorithme procédure Rdg
```

Remarques :

- *Comme on peut le constater, les rotations doubles sont la combinaison de deux rotations simples. La rotation gauche-droite (droite-gauche) est composée d'une rotation gauche (droite) sur le sous-arbre gauche (sous-arbre droit) de **B** suivie d'une rotation droite (gauche) sur **B**. Ces deux fonctions sont optimisables en remplaçant les deux appels de fonction par les affectations adéquates. L'intérêt étant d'éviter les deux appels de fonctions et de réduire le nombre d'affectations de **8** à **6**.*

- *Les rotations conservent la propriété d'arbre binaire de recherche (heureusement, hein ?). On constate, en effet, que la* lecture infixe *des arbres **B**,* rd(**B**), rg(**B**), rgd(**B**) *et* rdg(**B**) *est la même.*

Les arbres A-V.L.

Les **arbres A-V.L.** datent des années 60 (*1960, pas du temps des Romains et toussa...*) et sont la première classe d'**arbres H-équilibrés**. Leur nom vient de leurs concepteurs (*Adelson-Velskii, Landis*).

Définitions

- On dit qu'un arbre binaire ***B*** = < ***o, G, D*** > est **H-équilibré** si en tout noeud de l'arbre ***B***, les hauteurs des sous-arbres gauche et droit de ***B*** différent au plus de 1. C'est à dire si *hauteur(**G**) - hauteur(**D**)* $\in$ { ***-1, 0, +1*** }.

- Un **arbre AVL** est un **arbre binaire de recherche H-équilibré**.

Spécification formelle

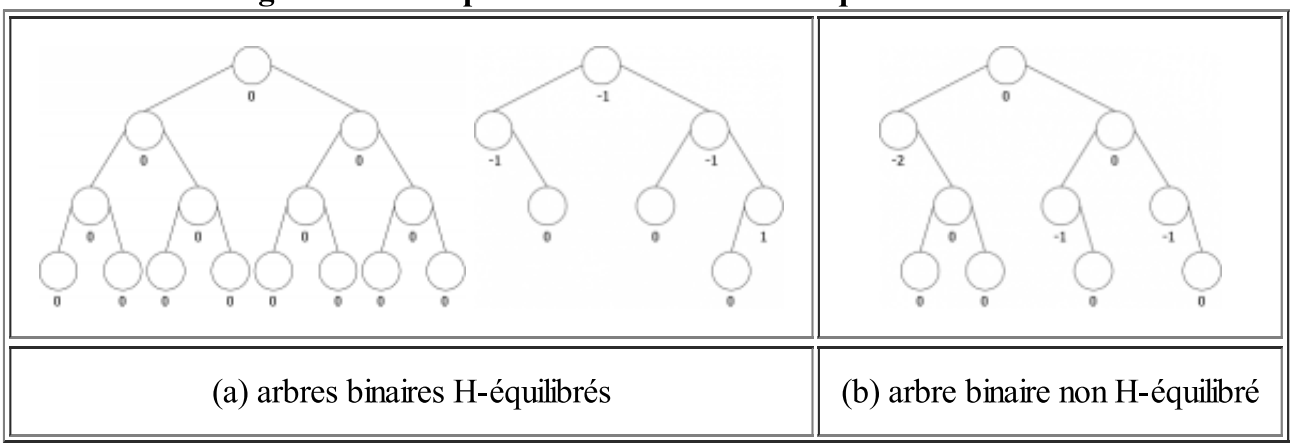
Nous avons alors besoin d'une une opération de *déséquilibre*, définie récursivement par :

```
opérations

  déséquilibre : arbrebinaire ⟶ entier

axiomes

  déséquilibre(arbrevide) = 0
  déséquilibre(< o, G, D >) = hauteur(G) - hauteur(D)

avec

  arbrebinaire B, G, D
  noeud        o
```

Donc, un arbre binaire **B** est H-équilibré si pour tout sous-arbre **C** de **B** on a : Déséquilibre(**C**) $\in$ **{ -1, 0, 1 }**

Nous pouvons voir sur la figure 14.a un exemple d'arbres H-équilibrés, et sur la figure 14.b l'exemple d'un arbre qui ne l'est pas.

Figure 14. Exemples d'arbres binaires H-équilibrés ou non.



(a) arbres binaires H-équilibrés | (b) arbre binaire non H-équilibré

Nous ne détaillerons pas l'algorithme de recherche dans les AVL. Dans la mesure où ceux-ci sont des arbres binaires de recherche, il suffit de reprendre ceux de la recherche dans un ABR. Le nombre de comparaisons est toujours d'ordre logarithmique.

En revanche, l'ajout ou la suppression dans un AVL peut provoquer un déséquilibre qui générera alors une réorganisation locale ou complète de l'arbre. Nous allons donc étudier (enfin on va essayer) les algorithmes d'ajout et de suppression dans les AVL.

Ajout dans un AVL

Le principe de construction d'un AVL est le suivant: on fait l'**ajout** du nouvel élément **aux feuilles**. Puis on rééquilibre l'arbre si cet ajout l'a déséquilibré. Dans ce cas, le déséquilibre maximum que l'on peut rencontrer est ± 2.

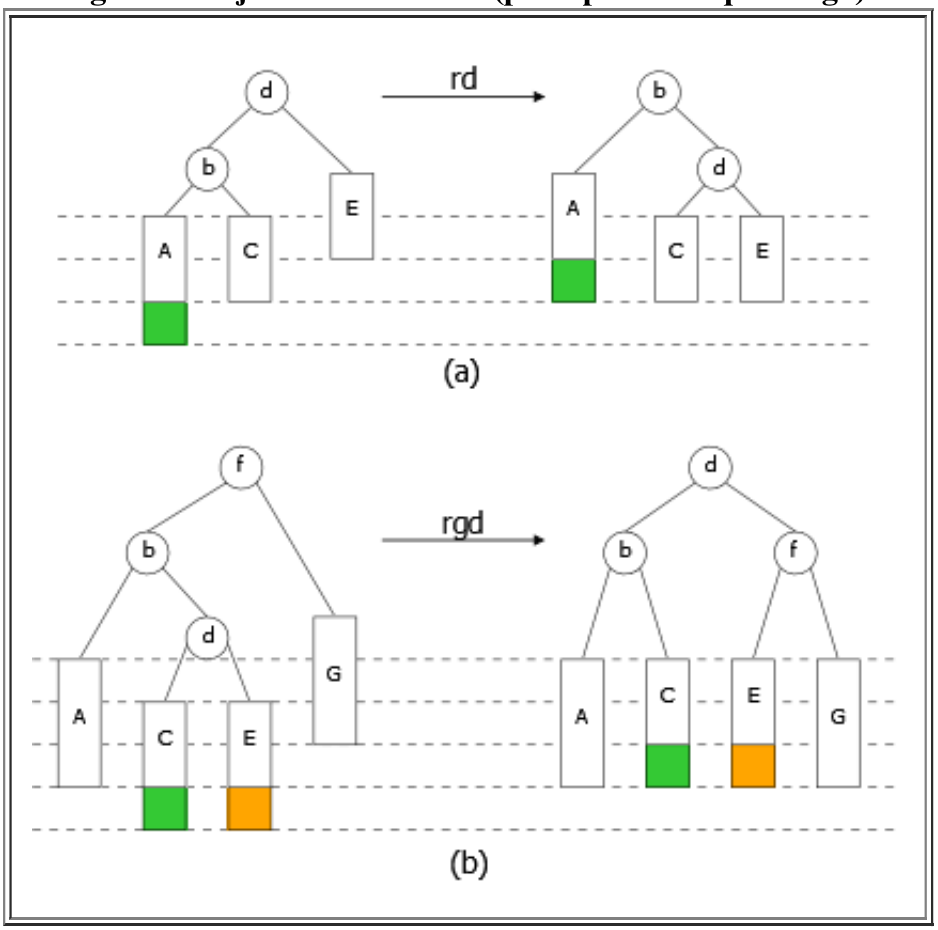
Lorsqu'il y a déséquilibre après l'ajout d'un noeud **x**, il suffit de rééquilibrer l'arbre à partir d'un noeud **y** se trouvant dans le chemin de la racine à la feuille **x** (**y** peut être la racine elle-même). Ce noeud **y** est le dernier noeud rencontré (le plus proche de **x**) pour lequel le déséquilibre est ± 2.

Principe général de rééquilibrage

Soit ***B*** = < ***r, G, D*** > B un AVL , donc n'ayant pas de sous-arbres en déséquilibre ± 2. Supposons que l'on ajoute un élément ***x*** sur une feuille de ***G*** et que la hauteur de ce dernier augmente de 1 et que ***G*** reste un AVL (avant l'ajout dans ***G***, son déséquilibre était 0). Dans ce cas, nous avons les trois possibilités suivantes :

1. Si le déséquilibre de ***B*** valait 0 avant l'ajout, il vaut 1 après. ***B*** reste un AVL et sa hauteur a augmenté de 1.
2. Si le déséquilibre de ***B*** valait –1 avant l'ajout, il vaut 0 après. ***B*** reste un AVL et sa hauteur n'est pas modifiée.
3. Si le déséquilibre de ***B*** valait 1 avant l'ajout, il vaut 2 après. ***B*** n'est plus H-équilibré, il faut le réorganiser (cf. figure 15).

Figure 15. Ajout dans un AVL (principe de rééquilibrage).



Comme on peut le voir sur la Figure 15, le troisième cas présente deux possibilités :

1. Le déséquilibre de ***G*** passe de 0 à 1 (Figure 15(a)) et le rééquilibrage de ***B*** se fait à l'aide d'une rotation simple à droite.
2. Le déséquilibre de ***G*** passe de 0 à –1, il y a inversion du sens de déséquilibre entre ***B*** et ***G***. Dans ce cas, le rééquilibrage s'effectue à l'aide d'une rotation double gauche-droite (Figure 15(b)).

Dans les deux cas, on récupère un arbre ***B*** rééquilibré qui est bien un AVL et dont la hauteur est redevenue celle qu'il avait avant l'ajout dé l'élément ***x*** (magnifique !).

*Remarque: Bien évidemment il existe les trois cas symétriques dans le cas où l'ajout se fasse sur **D**.*

Spécification formelle

Pour l'ajout dans un AVL, nous avons besoin d'une opération ajouter-AVL et de deux opérations gérant le déséquilibre, à savoir *rééquilibrer* et *déséquilibre*. La première rééquilibre l'arbre, la deuxième permet de connaître le déséquilibre d'un arbre. Ces opérations sont définies de la manière suivante :

```
opérations

   Ajouter-avl  : element x arbrebinaire ⟶ arbrebinaire
   rééquilibrer : arbrebinaire ⟶ arbrebinaire
   déséquilibre : arbrebinaire ⟶ entier

préconditions

   rééquilibrer(B) est-défini-ssi déséquilibre(B) ∈ {-2,-1,0,1,2}

axiomes

   Ajouter-avl( x, Arbre-vide ) = x
   x <= r => ajouter-avl( x, < r, G, D >))= rééquilibrer(< r, ajouter-avl( x, G), D >)
   x > r => Ajouter-avl( x, < r, G, D >))= rééquilibrer(< r, G, ajouter-avl( x, D)>)
   déséquilibre(B) = -1 => rééquilibrer(B) = B
   déséquilibre(B) =  0 => rééquilibrer(B) = B
   déséquilibre(B) =  1 => rééquilibrer(B) = B
   déséquilibre(B) =  2 & déséquilibre(G) =  1 => rééquilibrer(B) = rd(B)
   déséquilibre(B) =  2 & déséquilibre(G) = -1 => rééquilibrer(B) = rgd(B)
   déséquilibre(B) = -2 & déséquilibre(D) =  1 => rééquilibrer(B) = rdg(B)
   déséquilibre(B) = -2 & déséquilibre(D) = -1 => rééquilibrer(B) = rg(B)

avec

   arbrebinaire B, G, D
   noeud r
   element x
```

Remarque: On peut noter qu'il y a au plus un rééquilibrage à chaque ajout.

Algorithme

En terme d'implémentation, nous devons mémoriser dans chaque noeud la valeur de déséquilibre de l'arbre dont il est racine. Il est donc nécessaire de rajouter un champ *deseq* prenant les valeurs {-2,-1,0,1,2} dans l'enregistrement de type *t_noeud* défini pour la représentation dynamique des arbres binaires. Ce qui donne :

```
Types

   t_element = ...                 /* Définition du type des éléments */
   t_avl = ↑t_noeud                /* Définition du type t_arbre (pointeur sur t_noeud) */
   t_noeud = enregistrement        /* Définition du type t_noeud */
      entier    deseq
```

```
      t_element elt
      t_avl     fg, fd
   fin enregistrement t_noeud

Variables

   t_avl B
```

*<u>Remarque:</u> Nous allons rester dans un formalisme abstrait, et pour cela nous utiliserons le type Avl qui est un dérivé du type ArbreBinaire auquel nous avons ajouté les opérations propres aux Avl , en détaillant toutefois certaines d'entre elles comme **rééquilibrer**.*

```
algorithme procédure ajouterAVL
paramètres globaux
   AVL  B
paramètres locaux
   element x
variables
   AVL  Y, A, AA, P, PP
Début
      Y ⟵ < 0, x, arbrevide, arbrevide >    /* Création du noeud, deseq = 0 */
   si B = arbrevide alors
      B ⟵ Y
   sinon
      A ⟵ B, AA ⟵ arbrevide                /* AA est le père de A */
      P ⟵ B, PP ⟵ arbrevide                /* PP est le père de P */
   tant que P <> arbrevide faire            /* Recherche de la feuille d'ajout */
      si déséquilibre(P) <> 0 alors         /* Si un noeud a un déséquilibre */
         A ⟵ P                              /* différent de 0, on le mémorise */
         AA ⟵ PP                            /* dans A, et son père dans AA  */
      fin si
      PP ⟵ P
      si x <= contenu(racine(P)) alors
         P ⟵ g(P)
      sinon
         P ⟵ d(P)
      fin si
   fin tant que
```

(Christophe "krisboul" Boullay)

- Dernière modification de cette page le 26 mars 2013 à 17:20.