Arbres binaires de recherche

Marc Lorenzi

8 juillet 2019

Entrée [1]:
```
import matplotlib.pyplot as plt
import random
```

Entrée [2]:
```
plt.rcParams['figure.figsize'] = (20, 6)
```

Ce notebook reprend dans les sections 1 et 2 un certain nombre de notions déjà vues dans le notebook sur les arbres, qu'il est conseillé d'avoir lu auparavant. Nous allons aborder le sujet des **arbres équilibrés**, et étudier une implémentation de l'arbre binaire de recherche appelée **arbre AVL**.

1. Arbres binaires

1.1 Représentation des arbres en Python

Un arbre binaire est soit vide, soit non vide. S'il n'est pas vide il possède une **racine** et deux **fils**, un fils gauche et un fils droit, qui sont eux-mêmes des arbres binaires.

Nous représenterons les arbres en Python comme suit :

- L'arbre vide est représenté par `None` .
- Si t est un arbre non vide, de racine x, de fils gauche u et de fils droit v, nous représenterons t par le triplet (x, u, v).

Les fonctions ci-dessous résument tout. Elles permettent de récupérer les "parties" d'un arbre, de fabriquer un arbre à partir de ses parties, etc. Elles s'exécutent en temps $O(1)$.

Entrée [3]:
```
def vide(): return None
def racine(t): return t[0]
def fg(t): return t[1]
def fd(t): return t[2]
def arbre(x, u, v): return (x, u, v)
```

Entrée [4]:
```
def est_vide(t): return t == None
```

Notation : Nous utiliserons les notations suivantes :

- L'arbre vide est noté $\varnothing$.
- Si t est un arbre non vide de racine x, de fils gauche u et de fils droit v, nous noterons $t = u \oplus^x v$.

Nous utiliserons de temps en temps pour nos explications l'exemple ci-dessous.

Entrée [5]:
```
exemple = (2, (1, (0, None, None), None), (5, (4, (3, None, None), Nc
          (12, (9, (8, (6, None, (7, None, None)), None), (10, None,
          (13, None, (14, None, (18, (17, (15, None, (16, None, None)
```

Entrée [6]:
```
print(exemple)
```

```
(2, (1, (0, None, None), None), (5, (4, (3, None, None), None), (12,
(9, (8, (6, None, (7, None, None)), None), (10, None, (11, None, Non
e))), (13, None, (14, None, (18, (17, (15, None, (16, None, None)),
None), (19, None, None)))))))
```

1.2 Hauteur, nombre de noeuds

On définit inductivement la hauteur d'un arbre comme suit :

Définition :

- $h(\varnothing) = 0$.
- $h(u \oplus^x v) = 1 + \max(h(u), h(v))$

Entrée [7]:
```
def hauteur(t):
    if est_vide(t): return 0
    else:
        u, v = fg(t), fd(t)
        return 1 + max(hauteur(u), hauteur(v))
```

Entrée [8]:
```
hauteur(exemple)
```

Out[8]: 9

On définit inductivement le nombre de noeuds d'un arbre comme suit :

Définition :

- $|\varnothing| = 0$.
- $|u \oplus^x v| = 1 + |u| + |v|$.

Remarquez la notation $|t|$ pour le nombre de noeuds de l'arbre t. La quantité $|t|$ est une mesure assez représentative de la **taille** de l'arbre, c'est à dire de la mémoire qu'il occupe dans notre machine.

Entrée [9]:
```
def nombre_noeuds(t):
    if est_vide(t): return 0
    else:
        u, v = fg(t), fd(t)
        return 1 + nombre_noeuds(u) + nombre_noeuds(v)
```

Entrée [10]:
```
nombre_noeuds(exemple)
```

Out[10]: 20

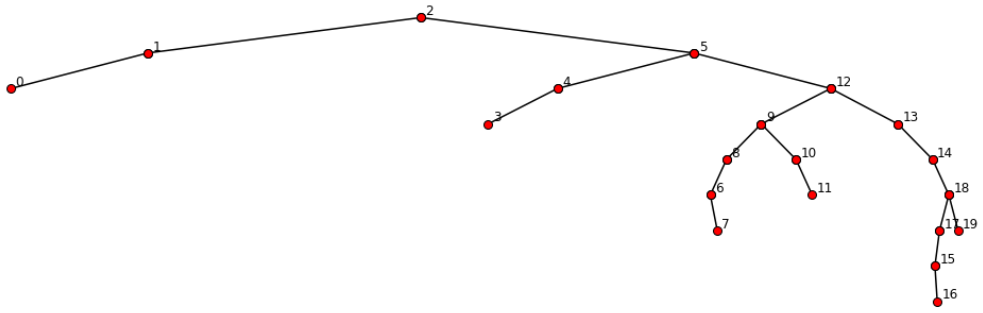
1.3 Dessiner un arbre

Je n'entre pas dans les détails des fonctions ci-dessous. La fonction `dessiner` permet d'avoir une représentation graphique d'un arbre.

Entrée [11]:
```
def dessiner_aux(t, rect, dy, labels, avl=False):
    if est_vide(t): return
    x1, x2, y1, y2 = rect
    xm = (x1 + x2) // 2
    x, t1, t2 = t
    dessiner_aux(t1, (x1, xm, y1, y2 - dy), dy, labels, avl)
    dessiner_aux(t2, (xm, x2, y1, y2 - dy), dy, labels, avl)
    if labels:
        if avl: plt.text(xm + 2, y2, str(x[0]), fontsize=12, horizor
        else: plt.text(xm + 2, y2, str(x), fontsize=12, horizontalal
    if not est_vide(t1):
        a, b = ((xm, (x1 + xm) // 2), (y2, y2 - dy))
        plt.plot(a, b, 'k', marker='o', markerfacecolor='r', markers
    if not est_vide(t2):
        c, d = ((xm, (x2 + xm) // 2), (y2, y2 - dy))
        plt.plot(c, d, 'k', marker='o', markerfacecolor='r', markers
```

Entrée [12]:
```
def dessiner(t, labels=True, avl=False):
    d = 512
    pad = 20
    dy = (d - 2 * pad) / (hauteur(t))
    dessiner_aux(t, (pad, d - pad, pad, d - pad), dy, labels, avl)
    plt.axis([0, d, 0, d])
    plt.axis('off')
```

Entrée [13]:
```
dessiner(exemple)
```



2. Arbres binaires de recherche

2.1 C'est quoi ?

Supposons que les noeuds de nos arbres appartiennent à un ensemble totalement ordonné. Nous allons nous intéresser à des arbres binaires dans lesquels il est entre-autres facile de retrouver un noeud. On les appelle les **arbres binaires de recherche** (en abrégé : ABR).

Définition : Soit t un arbre binaire.

- Si t est vide, c'est un ABR.
- Si $t = u \bigoplus^x v$, t est un ABR lorsque
  - u est un ABR
  - v est un ABR
  - Les noeuds de u sont strictement inférieurs à x
  - Les noeuds de v sont supérieurs ou égaux à x

2.2 Rechercher un objet dans un ABR

Il est facile de rechercher un objet dans un ABR. Si l'objet est plus petit que la racine, on recherche dans le fils gauche. Sinon ... je vous laisse deviner.

Entrée [14]:
```
def rechercher(x, t):
    if est_vide(t): return False
    else:
        y, u, v = racine(t), fg(t), fd(t)
        if x < y: return rechercher(x, u)
        elif x > y: return rechercher(x, v)
        else: return True
```

Entrée [15]:
```
rechercher(18, exemple)
```

Out[15]: True

Entrée [16]:
```
rechercher(27, exemple)
```

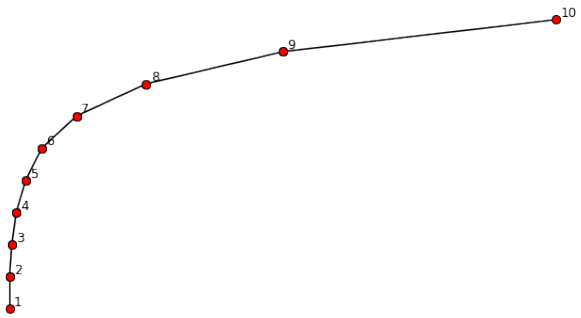
Out[16]:
```
False
```

On voit facilement que le nombre de comparaisons à effectuer pour savoir si, oui ou non, un objet est dans l'arbre, est un $O(h)$ où h est la hauteur de l'arbre. Si l'arbre est "équilibré" (en un sens à préciser, et c'est le but de ce notebook) on peut montrer que sa hauteur est logarithmique en le nombre de noeuds. La recherche est donc très efficace. Même pour des arbres ayant des millions de noeuds, ce sera presque instantané.

En revanche, si l'arbre n'est pas équilibré on peut avoir un temps de recherche en $O(n)$ où n est le nombre de noeuds de l'arbre. Dans ce cas on n'est pas contents du tout. Il existe des algorithmes permettant d'équilibrer les arbres, nous verrons cela un peu plus loin. Ci-dessous, un arbre où la fonction de recherche a un mauvais comportement.

Entrée [17]:
```
def arbre_pas_equilibre_du_tout(n):
    if n == 0: return vide()
    else:
        t = arbre_pas_equilibre_du_tout(n - 1)
        return arbre(n, t, vide())
```

Entrée [18]:
```
dessiner(arbre_pas_equilibre_du_tout(10))
```



2.3 Insérer un objet dans un ABR

Insérer n'est pas plus difficile que rechercher :

Entrée [19]:
```
def inserer(x, t):
    if est_vide(t): return (x, vide(), vide())
    else:
        y, u, v = racine(t), fg(t), fd(t)
        if x <= y: return (y, inserer(x, u), v)
        else: return (y, u, inserer(x, v))
```

La complexité de l'insertion en termes de comparaisons est la même que celle de la recherche : $O(h)$ où h est la hauteur de l'arbre.

2.4 ABR aléatoire

Nous voici prêts à créer de façon automatique des arbres un peu plus gros que notre petit `exemple` . Voici tout d'abord une fonction `random_list` qui prend en paramètre un entier n et renvoie une permutation des entiers de 0 à $n-1$.

Entrée [20]:
```
def random_list(n):
    s = list(range(n))
    random.shuffle(s)
    return s
```

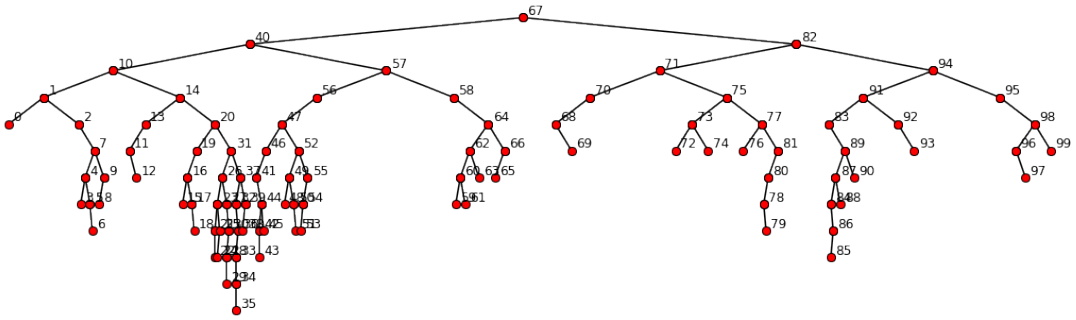
Entrée [21]:
```
random_list(10)
```

Out[21]:
```
[0, 3, 4, 9, 6, 1, 2, 8, 7, 5]
```

Pour créer un ABR "aléatoire", on crée une liste aléatoire et on insère ses éléments dans un ABR initialement vide.

Entrée [22]:
```
def random_abr(n):
    s = random_list(n)
    t = None
    for x in s:
        t = inserer(x, t)
    return t
```

Entrée [23]:
```
t = random_abr(100)
dessiner(t)
```



Nous pouvons maintenant re-tester avec des arbres un peu plus conséquents les fonctions que nous avons déjà écrites.

Entrée [24]:
```
hauteur(random_abr(10000))
```

Out[24]:
```
32
```

Entrée [25]:
```
nombre_noeuds(random_abr(10000))
```

Out[25]:
```
10000
```

Le résultat renvoyé par la fonction précédente est assez rassurant.

2.5 Minimum, maximum, d'un ABR

Le maximum d'un ABR est facile à trouver. On part de la racine et on va à droite, à droite, ... Bref c'est au fond à droite :).

Entrée [26]:
```
def maximum(t):
    if est_vide(t): raise Exception('Arbre Vide')
    else:
        x, v = racine(t), fd(t)
        if est_vide(v): return x
        else: return maximum(v)
```

Entrée [27]:
```
maximum(random_abr(1000))
```

Out[27]: 999

Entrée [28]:
```
def minimum(t):
    if est_vide(t): raise Exception('Arbre Vide')
    else:
        x, u = racine(t), fg(t)
        if est_vide(u): return x
        else: return minimum(u)
```

Entrée [29]:
```
minimum(random_abr(1000))
```

Out[29]: 0

2.6 Un arbre est-il un ABR ?

La fonction ci-dessous renvoie `True` si son paramètre est un ABR, et `False` sinon. Remarquez que pour savoir si tous les noeuds de t_1 sont inférieurs à x, il suffit de regarder si c'est le cas pour le maximum de t_1.

Entrée [30]:
```
def est_ABR(t):
    if est_vide(t): return True
    else:
        x, u, v = racine(t), fg(t), fd(t)
        b = est_ABR(u) and est_ABR(v)
        if est_vide(u): b1 = True
        else: b1 = maximum(u) < x
        if est_vide(v): b2 = True
        else: b2 = minimum(v) >= x
        return b and b1 and b2
```

Entrée [31]:
```
est_ABR(random_abr(1000))
```

Out[31]: `True`

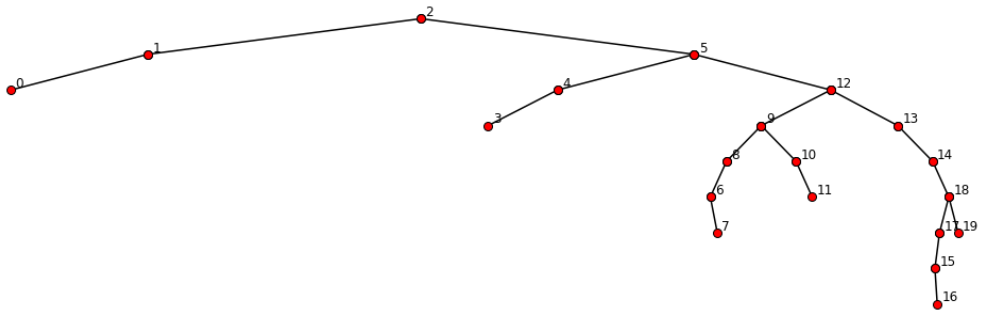
Remarque : Le fait que nous obtenions `True` à la question "Un ABR aléatoire est-il un ABR ?" nous incite à penser que les fonctions `minimum`, `maximum`, `inserer`, etc. sont correctes. Ou alors que tout est faux et que nous avons énormément de chance :-).

2.7 Supprimer un noeud dans un ABR

On fait dans ce paragraphe l'hypothèse que nos ABR ont tous leurs noeuds distincts. Soit t un ABR. Soit x un noeud de t qui a un fils droit non vide. Alors x n'est pas le maximum de t (cf plus haut) et possède donc un **successeur** dans l'arbre, c'est à dire un noeud s tel que $x < s$ mais pour tout noeud y de l'arbre, $y \leq x$ ou $s \leq y$. Dit autrement, il n'y a aucun noeud de l'arbre strictement compris entre x et s. Soit m le minimum du fils droit de x : montrons que m est le successeur de x.

Pour cela, soit y un noeud de l'arbre différent de x et appelons z le plus proche ancêtre commun de x et y. Pour que cela soit bien clair, reprenons notre exemple.

Entrée [32]:
```
dessiner(exemple)
```



Par exemple, le plus proche ancêtre commun de 10 et 15 est 12. Un certain nombre de cas se présentent :

Cas 1 : $z = x$ et y est dans le fils droit de x. Alors $m \leq y$ par définition de m, minimum du fils droit.

Cas 2 : $z = x$ et y est dans le fils gauche de x. Alors $y \leq x$ puisque t est un ABR.

Cas 3 : x est dans le fils gauche de z et y est dans le fils droit de z. Alors $m \leq y$ par définition de m.

Cas 4 : y est dans le fils gauche de z et x est dans le fils droit de z. Alors $y \leq z$ et $z < x$ puisque t est un ABR. Donc $y < x$.

Le successeur de x est donc bien m.

La fonction de suppression s'en déduit. On désire supprimer x dans l'ABR t. Que peut-il arriver ? Si x n'est pas la racine de t, on supprime x dans le fils gauche ou le fils droit de t. Sinon :

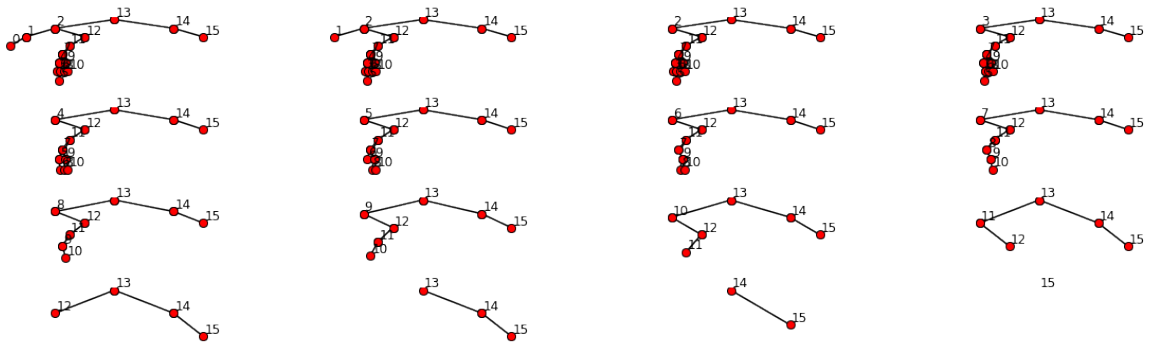
- si x n'a pas de fil droit, c'est facile, renvoyer le fils gauche de t.
- Si x a un fils droit, soit m le minimum de ce fils droit. Remplacer x par m et supprimer récursivement m du fils droit de x.

On vérifie facilement que l'on conserve la structure d'ABR parce que aucun noeud de t n'est compris entre x et m.

Entrée [33]:
```
def supprimer(x, t):
    if est_vide(t): return None
    else:
        y, u, v = racine(t), fg(t), fd(t)
        if x < y: return (y, supprimer(x, u), v)
        elif x > y: return (y, u, supprimer(x, v))
        elif est_vide(v): return u
        else:
            m = minimum(v)
            return (m, u, supprimer(m, v))
```

Testons. Partant d'un ABR aléatoire à 16 noeuds, on supprime successivement les noeuds de valeurs 0 à 14. À la fin il ne doit en rester qu'un :-). Le tableau de graphique se lit de gauche à droite, puis du haut vers le bas.

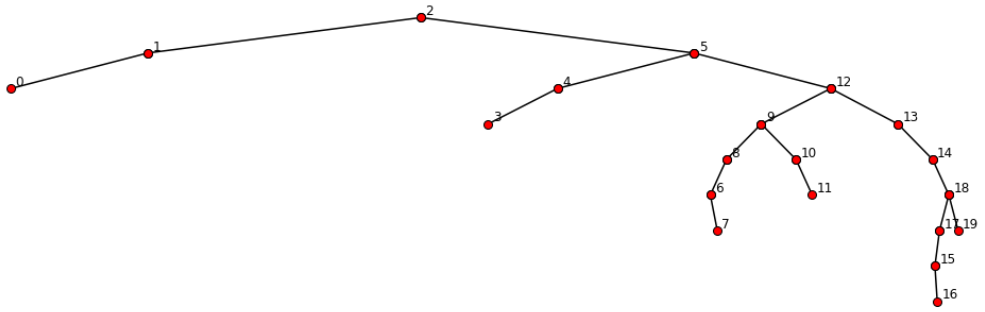
Entrée [34]:
```
t = random_abr(16)
ts = [t]
for k in range(15):
    t = supprimer(k, t)
    ts.append(t)
for k in range(1, 17):
    plt.subplot(4, 4, k)
    dessiner(ts[k - 1])
```



Remarque : Soit t un ABR. Soit x un noeud de t qui n'a pas de fils droit et qui n'est pas le maximum de t. Où est le successeur de x ? Eh bien le successeur de x est le plus proche ancêtre de x dont la racine du fils gauche est aussi un ancêtre de x. Démonstration laissée au lecteur.

Sur notre exemple :

Entrée [35]:
```
dessiner(exemple)
```



Les ancêtres de 11 (au sens large) sont : 11, 10, 9, 12, 5 et 2. Le successeur de 11 est 12 parce que la racine du fils gauche de 12 est 9, qui est encore un ancêtre de 11. Et 12 est le plus proche ancêtre de 11 à vérifier cette propriété.

Dit autrement, on part de 11, on monte à gauche, puis à gauche, etc. Au premier virage à droite on est arrivés. Bref, c'est la première à droite en montant.

3. Arbres équilibrés

Venons-en au fait.

3.1 Comparaison entre hauteur et nombre de noeuds

Proposition : Soit t un arbre binaire. On a

$$h(t) \leq |t| \leq 2^{h(t)} - 1$$

Démonstration : Faisons une récurrence sur la hauteur h de l'arbre t.

- Si $h = 0$, alors $t = \varnothing$ et $|t| = 0$. La double inégalité à montrer est alors une double égalité.
- Soit $h \in \mathbb{N}$. Supposons la propriété vraie pour tous les arbres de hauteur inférieure ou égale à h. Soit $t = u \bigoplus^x v$ un arbre de hauteur $h + 1$. On a, par exemple, $h(u) \leq h(v) = h$. De plus,

$$|t| = |u| + |v| + 1$$

Pour la première inégalité, on a

$$h(t) = h(v) + 1 \leq |v| + 1 \leq |u| + |v| + 1 = |t|$$

Pour la deuxième inégalité,

$$|t| = |u| + |v| + 1 \leq (2^{h(u)} - 1) + (2^{h(v)} - 1) + 1 \leq 2(2^h - 1) + 1 = 2^{h+1} - 1$$

De la proposition précédente on déduit, en passant aux logarithmes en base 2, que

$$h(t) \geq \lg(|t| + 1)$$

Nous avons vu dans la partie 1 que les opérations d'insertion et de suppression dans un ABR t s'effectuaient en $O(h(t))$. Ce temps est donc au moins logarithmique en le nombre de noeuds de t ... dit autrement nous venons d'obtenir exactement le contraire de ce que nous aurions voulu, un temps **au plus** logarithmique en $|t|$!

Remarquons que le majorant $h(t) \leq |t|$ est terrible. Il semble suggérer que pour certains ABR, l'insertion d'un élément se fait en temps **linéaire** en le nombre de noeuds de t. Si c'est le cas, les ABR ne servent à rien. Autant se servir de listes pour stocker nos données.

Alors, que faire ?

- Tout d'abord, trouver une condition simple sur l'arbre t pour que sa hauteur soit logarithmique en son nombre de noeuds.
- Puis imaginer une amélioration des algorithmes que nous avons vus sur les ABR pour que cette condition soit préservée par insertion ou suppression d'un noeud.

3.2 C'est quoi un arbre équilibré ?

Définition : Nous donnons une définition inductive de la notion d'arbre équilibré.

- $\varnothing$ est équilibré.
- L'arbre $u \bigoplus^x v$ est équilibré lorsque
  - u est équilibré
  - v est équilibré
  - $|h(u) - h(v)| \leq 1$

Entrée [36]:
```
def est_equilibre(t):
    if est_vide(t): return True
    else:
        u, v = fg(t), fd(t)
        return est_equilibre(u) and est_equilibre(v) and abs(hauteur
```

Entrée [37]:
```
est_equilibre(exemple)
```

Out[37]:
```
False
```

On s'en serait doutés :-).

Proposition : Il existe un réel $c > 1$ tel que, pour tout arbre équilibré t, on ait

$$|t| \geq c^{h(t)} - 1$$

et donc

$$h(t) \leq \log_c(1 + |t|)$$

Démonstration : Une récurrence sur la hauteur s'impose. Pour l'arbre vide, n'importe quel réel $c > 1$ fait l'affaire. Soit $h \in \mathbb{N}$. Supposons trouvé un réel $c > 1$ convenant pour tous les arbres équilibrés de hauteur inférieure ou égale à h. Soit $t = u \oplus^x v$ un arbre équilibré de hauteur $h + 1$. On a, par exemple,

$$h - 1 \leq h(u) \leq h(v) = h$$

De là,

$$|t| = |u| + |v| + 1 \geq (c^{h-1} - 1) + (c^h - 1) + 1 = c^{h+1}\left(\frac{1}{c^2} + \frac{1}{c}\right) - 1$$

Existe-t-il un réel $c > 1$ tel que $\frac{1}{c^2} + \frac{1}{c} = 1$? Ce serait le nombre idéal pour nous. Un tel réel vérifierait

$$c^2 = c + 1$$

Mais oui, un tel réel existe ! Il s'agit du nombre d'or, $c = \frac{1}{2}(1 + \sqrt{5})$. Notre proposition est donc montrée en prenant pour c le nombre d'or. Nous le noterons dorénavant ϕ.

En résumé : On a pour tout arbre t équilibré,

$$\log_2(1 + |t|) \leq h(t) \leq \log_\phi(1 + |t|)$$

La hauteur d'un arbre équilibré est donc logarithmique en son nombre de noeuds.

2.3 Arbres de Fibonacci.

Quand le nombre d'or apparaît, Fibonacci n'est jamais bien loin ... Dans les arbres que nous allons examiner la valeur des noeuds ne nous intéresse pas. Mettons par exemple les noeuds à 0.

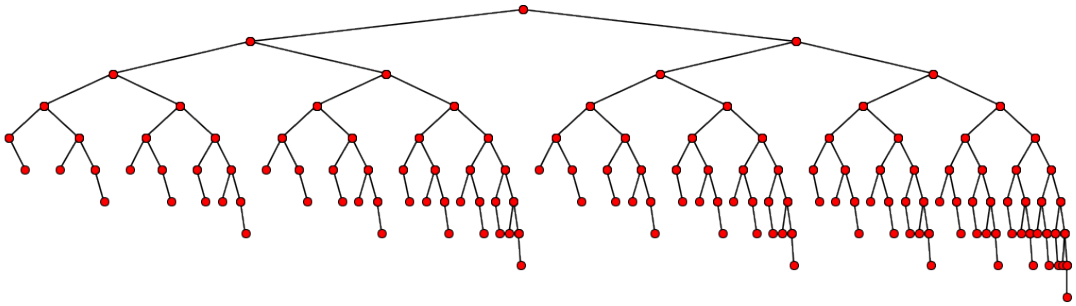
Définition : La suite de Fibonacci est la suite d'arbres binaires définie par

- F_0 est l'arbre vide.
- F_1 a une racine et pas de fils.
- Pour tout $n \in \mathbb{N}$, $F_{n+2} = F_n \oplus^0 F_{n+1}$

où la valeur de la racine, sans importance, n'est pas précisée.

Entrée [38]:
```
def fibonacci(n):
    f = vide()
    g = arbre(0, vide(), vide())
    for k in range(n):
        f, g = g, arbre(0, f, g)
    return f
```

Entrée [39]:
```
dessiner(fibonacci(10), labels=False)
```



Quelle est la hauteur d'un arbre de Fibonacci ?

Entrée [40]:
```
[hauteur(fibonacci(k)) for k in range(10)]
```

Out[40]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Proposition : Pour tout $n \in \mathbb{N}$, $h(F_n) = n$.

Démonstration : Récurrence facile à deux termes sur n, laissée au lecteur.

Combien de noeuds un arbre de Fibonacci possède-t-il ?

Entrée [41]:
```
[nombre_noeuds(fibonacci(k)) for k in range(10)]
```

Out[41]:
```
[0, 1, 2, 4, 7, 12, 20, 33, 54, 88]
```

Définissons la suite $(f_n)_{n\geq 0}$ des **nombres de Fibonacci** par

- $f_0 = 0$
- $f_1 = 1$
- Pour tout $n \in \mathbb{N}$, $f_{n+2} = f_n + f_{n+1}$.

Entrée [42]:
```
def nb_fibo(n):
    a = 0
    b = 1
    for k in range(n):
        a, b = b, a + b
    return a
```

Entrée [43]:
```
print([nb_fibo(k) for k in range(10)])
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Entrée [44]:
```
print([nb_fibo(k + 2) - 1 for k in range(10)])
```

```
[0, 1, 2, 4, 7, 12, 20, 33, 54, 88]
```

Proposition : On a pour tout $n \in \mathbb{N}$,

$$|F_n| = f_{n+2} - 1$$

Démonstration : On fait une récurrence à deux termes sur n.

- Pour $n = 0, 1$ c'est évident.
- Soit $n \in \mathbb{N}$. Supposons la propriété vraie pour n et $n+1$. On a alors

$$|F_{n+2}| = |F_n \oplus F_{n+1}| = |F_n| + |F_{n+1}| + 1 = f_{n+2} - 1 + f_{n+3} - 1 + 1 = f_{n+4} - 1$$

Remarquons maintenant que

$$f_n = \frac{\phi^n - \hat{\phi}^n}{\phi - \hat{\phi}}$$

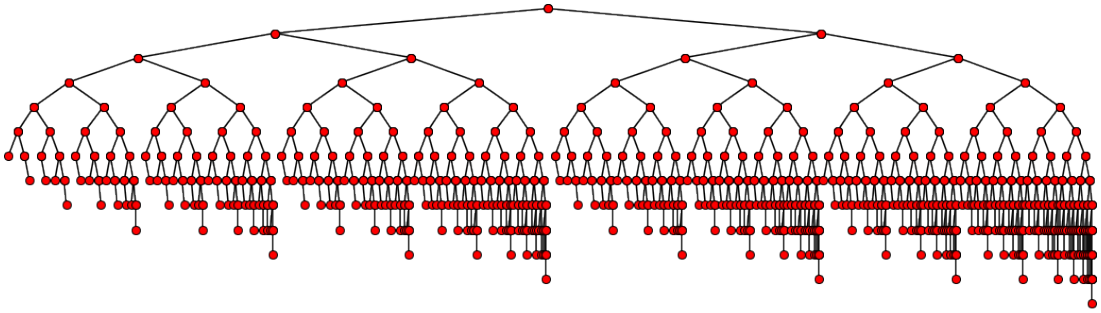
où ϕ, rappelons-le, est le nombre d'or et $\hat{\phi} = \frac{1}{2}(1 - \sqrt{5})$. On a clairement $f_n \sim \frac{1}{\phi - \hat{\phi}} \phi^n$.

De là,

$$\log_c(|F_n| + 1) = \log_\phi f_{n+2} \sim n + 2 \sim n = h(F_n)$$

Ceci montre que l'inégalité que nous avons montrée dans la section précédente est **asymptotiquement optimale** : les arbres de Fibonacci sont, d'une certaine manière, les plus déséquilibrés des arbres équilibrés :-).

Entrée [45]:
```
dessiner(fibonacci(13), labels=False)
```



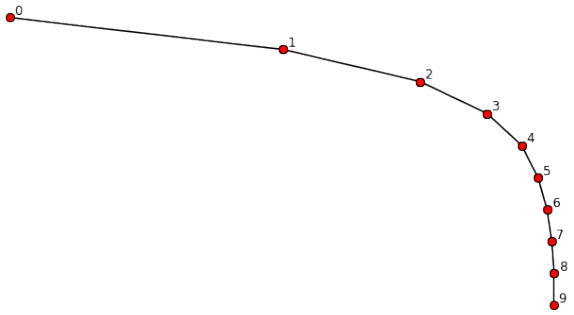
3. Arbres AVL

Les arbres AVL tirent leur nom des initiales de leurs inventeurs (découvreurs ?), Georgii Adelson-Velsky and Evgenii Landis.

3.1 C'est quoi ?

Le problème des ABR tels que nous les avons définis est que des insertions ou des suppressions peuvent déséquilibrer l'arbre. Or, le coût de ces opérations est fortement lié au fait que l'arbre soit équilibré. Petit exemple :

Entrée [46]:
```
t = vide()
for k in range(10): t = inserer(k, t)
dessiner(t)
```



On voit facilement que le coût de n insertions d'entiers successifs dans un arbre initialement vide est $O(n^2)$, ce qui n'est pas acceptable. Alors, impasse ? Pas forcément, il s'agit de réfléchir à la façon de maintenir les arbres équilibrés lors d'une insertion ou d'une suppression.

Il existe différentes approches du problème : arbres 2-3-4, arbres rouges et noirs, arbres AVL, ... Nous allons nous concentrer sur les arbres AVL.

Dans un arbre AVL, on stocke à la racine le couple formé de la "vraie" valeur de la racine et de la hauteur de l'arbre. Le calcul de la hauteur d'un arbre AVL se fait donc en $O(1)$ puisqu'il suffit de savoir lire :-).

Entrée [47]:
```
def racine_avl(t):
    if est_vide(t):
        raise Exception('Arbre vide')
    else: return t[0][0]

def hauteur_avl(t):
    if est_vide(t): return 0
    else: return t[0][1]
```

Il est alors facile de fabriquer en temps $O(1)$ un arbre AVL de racine x et de fils u et v :

Entrée [48]:
```
def arbre_avl(x, u, v):
    h = 1 + max(hauteur_avl(u), hauteur_avl(v))
    return ((x, h), u, v)
```

Il faut également réécrire les fonctions calculant le minimum et le maximum d'un arbre AVL.

Entrée [49]:
```
def maximum_avl(t):
    if est_vide(t): raise Exception('Arbre Vide')
    else:
        x, v = racine_avl(t), fd(t)
        if est_vide(v): return x
        else: return maximum_avl(v)
```

Entrée [50]:
```
def minimum_avl(t):
    if est_vide(t): raise Exception('Arbre Vide')
    else:
        x, u = racine_avl(t), fg(t)
        if est_vide(u): return x
        else: return minimum_avl(u)
```

3.2 Rotations !

Définissons deux fonctions de l'ensemble des arbres binaires sur lui-même, la rotation droite ρ_d et la rotation gauche ρ_g. Pour tous arbres u, v, w :

- $\rho_d((u \oplus^x v) \oplus^y w) = u \oplus^x (v \oplus^y w)$
- $\rho_g(u \oplus^x (v \oplus^y w)) = (u \oplus^x v) \oplus^y w$

L'arbre $\rho_d(t)$ est défini pour tout arbre t non vide dont le fils gauche est lui-même non vide. Propriété analogue pour ρ_g.

Exercice essentiel : Dessinez un arbre et sa rotation droite. Pourquoi appelle-t-on cela une rotation ?

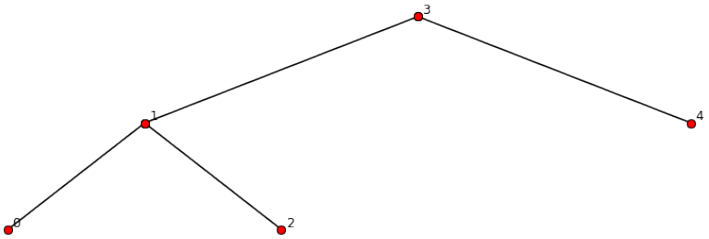
Entrée [51]:
```
def rotation_droite(t):
    x = racine_avl(t)
    t1, w = fg(t), fd(t)
    u, v = fg(t1), fd(t1)
    y = racine_avl(t1)
    return arbre_avl(y, u, arbre_avl(x, v, w))
```

Entrée [52]:
```
def rotation_gauche(t):
    x = racine_avl(t)
    u, t2 = fg(t), fd(t)
    v, w = fg(t2), fd(t2)
    y = racine_avl(t2)
    return arbre_avl(y, arbre_avl(x, u, v), w)
```

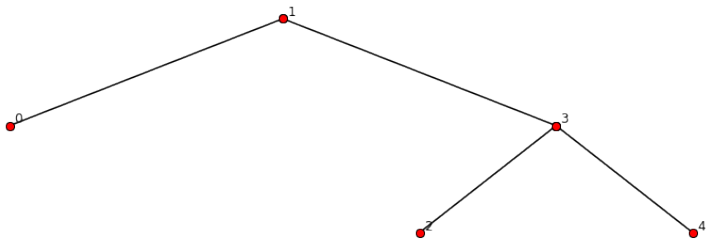
Voici un petit exemple.

Entrée [53]:
```
t = arbre_avl(3,
              arbre_avl(1,
                        arbre_avl(0, vide(), vide()),
                        arbre_avl(2, vide(), vide())),
              arbre_avl(4, vide(), vide()))
```

Entrée [54]:
```
dessiner(t, avl=True)
```

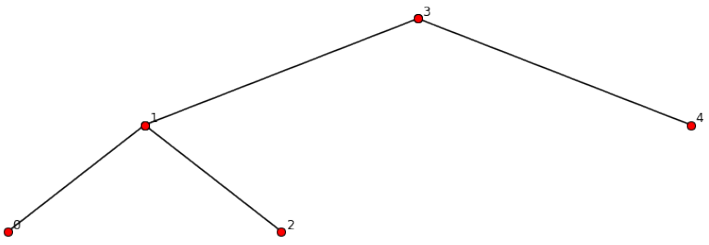


Entrée [55]:
```
t1 = rotation_droite(t)
dessiner(t1, avl=True)
```



Bien entendu, une rotation gauche de t_1 fait revenir à l'arbre de départ.

Entrée [56]:
```
dessiner(rotation_gauche(t1), avl=True)
```



Propriété : Les opérations de rotation conservent la structure d'ABR.

Démonstration : En exercice. Il suffit d'utiliser la définition d'ABR.

Appelons **facteur d'équilibre** d'un arbre le nombre suivant.

- $\epsilon(\varnothing) = 0$.
- $\epsilon(u \oplus^x v) = h(u) - h(v)$.

Le facteur d'équilibre d'un arbre équilibré est donc -1, 0 ou 1.

Entrée [57]:
```
def facteq(t):
    if est_vide(t): return 0
    else:
        return hauteur_avl(fg(t)) - hauteur_avl(fd(t))
```

Nous allons maintenant voir comment, à partir d'un arbre **légérement** déséquilibré, on peut rétablir l'équilibre par des rotations.

Soit $t = (u \oplus^x v) \oplus^y w$ un arbre. Supposons que $u \oplus^x v$ et w sont équilibrés, mais que t est légérement déséquilibré, c'est à dire que son facteur d'équilibre vaut -2 ou 2. Comment rééquilibrer t ?

Discutons le cas où $\epsilon(t) = 2$. L'autre cas est identique, en échangeant rotations droites et rotations gauches. Soit h la hauteur de t. On a donc

$$h(u \oplus^x v) = h - 1$$

et

$$h(w) = h - 3$$

Trois cas se présentent, selon la valeur du facteur d'équilibre de $u \oplus^x v$.

- Cas 1, $\epsilon(u \oplus^x v) = 1$. On a donc $h(u) = h - 2$ et $h(v) = h - 3$.

Faisons une rotation droite de t. Voici dans un tableau les hauteurs et facteurs d'équilibre qui nous intéressent.

	u	v	w	$v \oplus^y w$	$u \oplus^x (v \oplus^y w)$
h	$h-2$	$h-3$	$h-3$	$h-2$	$h-1$
ϵ				0	0

- Cas 2, $\epsilon(u \oplus^x v) = 0$. On a donc $h(u) = h - 2$ et $h(v) = h - 2$.

Ici encore, faisons une rotation droite de t.

	u	v	w	$v \oplus^y w$	$u \oplus^x (v \oplus^y w)$
h	$h-2$	$h-2$	$h-3$	$h-1$	h
ϵ				1	-1

- Cas 3, $\epsilon(u \oplus^x v) = -1$. On a donc $h(u) = h - 3$ et $h(v) = h - 2$.

Là, nous allons faire deux rotations : une rotation gauche de $u \oplus^x v$, suivie d'une rotation droite du "nouveau" t. Qu'est ce que cela donne ? Posons $v = v_1 \oplus^z v_2$. Nous avons

$$\rho_d(\rho_g(u \oplus^x (v_1 \oplus^z v_2)) \oplus^y w) = \rho_d((u \oplus^x v_1) \oplus^z v_2) \oplus^y w)$$
$$= (u \oplus^x v_1) \oplus^z (v_2 \oplus^y w)$$

Comme $v_1 \oplus^z v_2$ est équilibré et de hauteur $h-2$, nous avons 3 possibilités pour les hauteurs de v_1 et v_2 . Résumons tout cela dans un tableau.

	u	v_1	v_2	w	$u \oplus^x v_1$	$v_2 \oplus^y w$	$(u \oplus^x v_1) \oplus^z (v_2 \oplus^y w)$
h_1	$h-3$	$h-3$	$h-3$	$h-3$	$h-2$	$h-2$	$h-1$
ϵ_1					0	0	0
h_2	$h-3$	$h-2$	$h-3$	$h-3$	$h-2$	$h-2$	$h-1$
ϵ_2					1	0	0
h_3	$h-3$	$h-3$	$h-2$	$h-3$	$h-2$	$h-2$	$h-1$
ϵ_3					0	-1	0

Dans tous les cas, on constate que les opérations de rotation effectuées donnent bien des arbres équilibrés. Remarquez également que la hauteur de l'arbre obtenu est la même que celle de l'arbre de départ, voire un de moins.

Voici donc la fonction `reequilibrer` qui prend en paramètres un objet y et deux arbres équilibrés u et v, et qui effectue un recollement équilibré.

Entrée [58]:
```
def reequilibrer(y, u, v):
    e = hauteur_avl(u) - hauteur_avl(v)
    if abs(e) <= 1: return arbre_avl(y, u, v)
    elif e == 2:
        e1 = facteq(u)
        if e1 == -1: u = rotation_gauche(u)
        return rotation_droite(arbre_avl(y, u, v))
    elif e == -2:
        e2 = facteq(v)
        if e2 == 1: v = rotation_droite(v)
        return rotation_gauche(arbre_avl(y, u, v))
    else: raise Exception('Cas impossible')
```

Remarque : Les fonctions de rotation et de rééquilibrage s'exécutent toutes en temps $O(1)$.

3.3 Insertion dans un arbre AVL

Voici la fonction d'insertion dans un AVL.

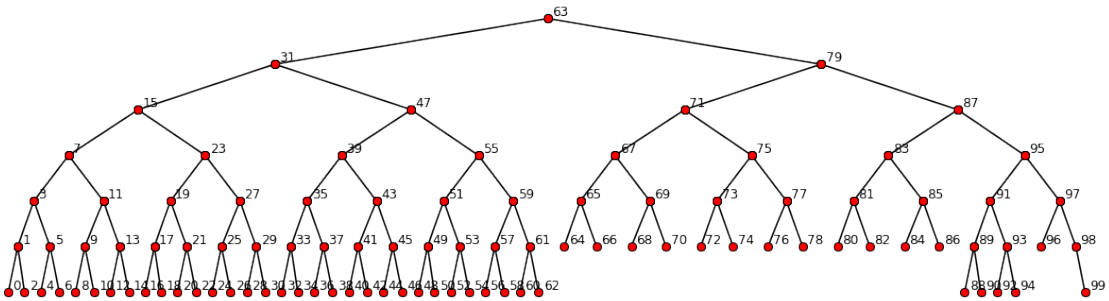
Entrée [59]:
```
def inserer_avl(x, t):
    if est_vide(t):
        return arbre_avl(x, vide(), vide())
    else:
        y = racine_avl(t)
        if x < y:
            return reequilibrer(y, inserer_avl(x, fg(t)), fd(t))
        else:
            return reequilibrer(y, fg(t), inserer_avl(x, fd(t)))
```

Proposition : L'insertion dans un AVL équilibré t renvoie un AVL équilibré t' tel que $h(t) \leq h(t') \leq h(t) + 1$.

Démonstration : Gardons la démonstration pour tout à l'heure. Nous avons hâte de tester notre nouvelle fonction !

Rappelez-vous la catastrophe qui était survenue lors de l'insertion de n entiers successifs dans un ABR. Maintenant, plus de souci !

Entrée [60]:
```
t = vide()
for k in range(100):
    t = inserer_avl(k, t)
dessiner(t, True, avl=True)
```



Entrée [61]:
```
def est_ABR_avl(t):
    if est_vide(t): return True
    else:
        x, u, v = racine_avl(t), fg(t), fd(t)
        b = est_ABR_avl(u) and est_ABR_avl(v)
        if est_vide(u): b1 = True
        else: b1 = maximum_avl(u) < x
        if est_vide(v): b2 = True
        else: b2 = minimum_avl(v) >= x
        return b and b1 and b2
```

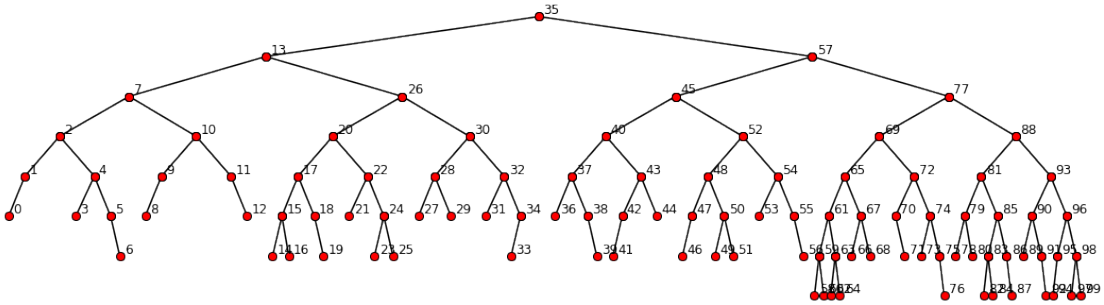
Entrée [62]:
```
print(est_ABR_avl(t))
print(est_equilibre(t))
```

```
True
True
```

Et voici une fonction `random_avl` ...

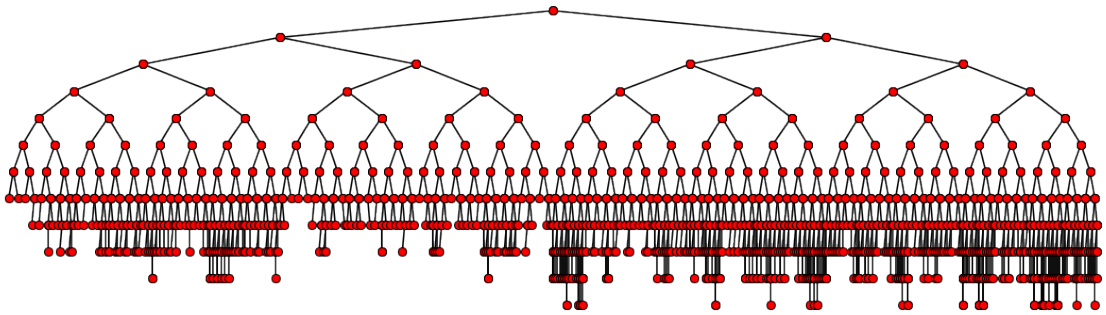
Entrée [63]:
```
def random_avl(n):
    s = list(range(n))
    random.shuffle(s)
    t = vide()
    for x in s: t = inserer_avl(x, t)
    return t
```

Entrée [64]:
```
t = random_avl(100)
dessiner(t, avl=True)
```



Voici un "gros" arbre AVL.

Entrée [65]:
```
t = random_avl(1000)
dessiner(t, False)
```



Entrée [66]:
```
print(est_ABR_avl(t))
print(est_equilibre(t))
```

```
True
True
```

Si vous vous inquiétez du temps mis pour afficher un arbre AVL aléatoire de 1000 noeuds, sachez que ce n'est pas la faute de `inserer` mais celle de `dessiner` . La preuve, voici le calcul quasi-instantané d'un arbre AVL aléatoire de 10^4 noeuds. Je vous déconseille de le dessiner.

Entrée [67]:
```
t = random_avl(10000)
print(nombre_noeuds(t))
print(est_ABR_avl(t))
print(est_equilibre(t))
```

```
10000
True
True
```

Vous n'avez pas oublié qu'il nous reste une démonstration à faire ? Rappelons la propriété.

Propriété : L'insertion dans un AVL équilibré t renvoie un AVL équilibré t' tel que $h(t) \leq h(t') \leq h(t) + 1$.

Démonstration : On fait une récurrence forte sur la hauteur de t.

- C'est évident si t est vide.
- Soit $h \in \mathbb{N}$. Supposons la propriété vraie pour tous les arbres AVL équilibrés de hauteur inférieure ou égale à h. Soit $t = u \oplus^y v$ un arbre AVL équilibré de hauteur $h + 1$. Pour fixer les idées, $h(u) = h$ et $h(v) = h$ ou $h - 1$. Insérons un objet x dans t. Deux cas se présentent.
- Cas 1, $x < y$. Par l'hypothèse de récurrence, l'insertion de x dans u produit un arbre AVL équilibré u' de hauteur h ou $h + 1$. L'arbre $t' = u' \oplus^y v$ a donc un facteur d'équilibre de 1, ou 2. Son rééquilibrage renvoie donc un arbre AVL équilibré. Le lecteur est invité à vérifier que le rééquilibrage produit un arbre de hauteur $h + 1$ ou $h + 2$. Il faut pour cela considérer plusieurs cas, selon le facteur d'équilibre de u'.
- Cas 2, $x >= y$. Par l'hypothèse de récurrence, l'insertion de x dans v produit un arbre AVL équilibré v' de hauteur $h - 1$ ou h. L'arbre $t' = u \oplus^y v'$ a donc un facteur d'équilibre de 0 ou 1. Il est donc équilibré et son rééquilibrage le renvoie tel quel. De plus, la hauteur de t' est $h + 1$.

La fonction d'insertion crée donc des arbres AVL équilibrés qui, comme nous l'avons déjà vu plus haut, ont aussi la propriété d'ABR.

3.4 Complexité de l'insertion

Propriété : L'insertion d'un objet dans un arbre AVL équilibré t a une complexité en $O(\log |t|)$.

Démonstration : En effet, nous avons vu que l'insertion a une complexité en $O(h(t))$, et comme l'arbre est équilibré, $h(t) = O(\log |t|)$.

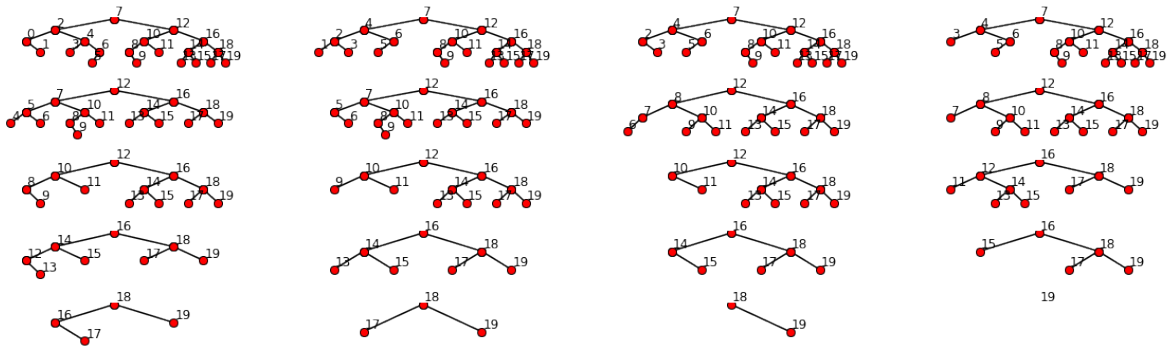
3.5 Suppression dans un arbre AVL

Je ne détaille pas. Cette fonction est quasiment analogue à la fonction de suppression dans les ABR. Il faut juste penser à rééquilibrer a bon escient. Cette fonction a, comme la fonction d'insertion, une complexité en $O(\log |t|)$.

Entrée [68]:
```
def supprimer_avl(x, t):
    if est_vide(t): return vide()
    else:
        y, u, v = racine_avl(t), fg(t), fd(t)
        if x < y: return reequilibrer(y, supprimer_avl(x, u), v)
        elif x > y: return reequilibrer(y, u, supprimer_avl(x, v))
        elif est_vide(v): return u
        else:
            m = minimum_avl(v)
            return reequilibrer(m, u, supprimer_avl(m, v))
```

Comme pour les ABR, voici une illustration de la fonction de suppression. Partant d'un arbre AVL aléatoire dont les noeuds sont les entiers de 0 à 19, on supprime successivement les noeuds de valeurs 0 à 18. À la fin il ne doit en rester qu'un :-).

Entrée [69]:
```
t = random_avl(20)
ts = [t]
for k in range(19):
    t = supprimer_avl(k, t)
    ts.append(t)
for k in range(1, 21):
    plt.subplot(5, 4, k)
    dessiner(ts[k - 1], avl=True)
```



4. Retour sur les ABR aléatoires

4.1 Nombre d'ABR à n noeuds

Soit E un ensemble totalement ordonné de cardinal $n \in \mathbb{E}$. Combien y a-t-il d'ABR dont les noeuds sont distincts et sont exactement les éléments de E ? Clairement, cela ne dépend que de n et pas de l'ensemble particulier E. Notons donc C_n le nombre recherché.

- Si $n = 0$, c'est facile, il n'y a qu'un ABR qui convient, c'est $\emptyset$. On a donc $C_0 = 1$.
- Supposons maintenant $n \geq 1$. Notons $E = \{x_0, \dots, x_{n-1}\}$ où $x_0 < \dots < x_{n-1}$. Pour $0 \leq k \leq n-1$, combien y a-t-il d'ABR dont la racine est x_k ? Pour un tel ABR, les noeuds de son fils gauche sont $x_0, \dots, x_{k-1}$, au nombre de k. Il y a donc C_k fils gauches possibles. De même, il y a C_{n-1-k} fils droits possibles. Ainsi, il y a $C_k C_{n-1-k}$ ABR dont la racine est x_k. En conclusion,

$$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$$

Cette relation de récurrence a l'air terrible mais on peut la résoudre. Posons

$$f(z) = \sum_{n=0}^{\infty} C_n z^n$$

Pour ne pas rallonger un notebook déjà terriblement long, j'admettrai ici que cette série entière a un rayon de convergence strictement positif (il vaut en fait $\frac{1}{4}$). On a

$$f(z)^2 = \sum_{n=0}^{\infty} a_n z^n$$

où $a_n = \sum_{k=0}^{n} C_k C_{n-k} = C_{n+1}$. Ainsi,

$$zf(z)^2 = \sum_{n=0}^{\infty} C_{n+1} z^{n+1} = f(z) - 1$$

ou encore

$$zf(z)^2 - f(z) + 1 = 0$$

$f(z)$ est donc racine d'une équation du second degré. Le discriminant de cette équation est $1 - 4z$. Continuant formellement les calculs, il vient pour z non nul

$$f(z) = \frac{1 \pm \sqrt{1 - 4z}}{2z}$$

Des considérations de continuité pour la fonction f montrent que le signe $\pm$ est le même pour tout z. Toujours pour des raisons de continuité, la fonction f est bornée au voisinage de 0. Le numérateur de la fraction tend donc vers 0 lorsque z tend vers 0. Ainsi,

$$f(z) = \frac{1 - \sqrt{1 - 4z}}{2z}$$

Développons $\sqrt{1 - 4z}$ en série entière.

$$\sqrt{1 - 4z} = \sum_{n=0}^{\infty} (-4)^n z^n \binom{\frac{1}{2}}{n}$$

où $\binom{\frac{1}{2}}{0} = 1$, et

$$\binom{\frac{1}{2}}{n} = \frac{\frac{1}{2}\left(\frac{1}{2} - 1\right) \dots \left(\frac{1}{2} - n + 1\right)}{n!}$$

si $n \geq 1$. Le lecteur est invité à vérifier que si $n \geq 1$,

$$\binom{\frac{1}{2}}{n} = \frac{(-1)^{n-1}}{2^{2n-1}} \frac{1}{n} \binom{2n-2}{n-1}$$

On en déduit après quelques simplifications que

$$f(z) = \sum_{n=0}^{\infty} \frac{1}{n+1} \binom{2n}{n} z^n$$

Par unicité de l'écriture d'une série entière on en déduit que pour tout $n \in \mathbb{N}$,

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

Les entiers C_n sont appelés les **nombres de Catalan**.

Exercice : En utilisant la formule de Stirling, montrer que

$$C_n \sim \frac{4^n}{n\sqrt{\pi n}}$$

Entrée [70]:
```
def catalan(n):
    p = 1
    for k in range(n):
        p *= (2 * n - k)
    for k in range(n):
        p = p // (k + 1)
    return p // (n + 1)
```

Entrée [71]:
```
print([catalan(k) for k in range(15)])
```

```
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742
900, 2674440]
```

4.2 Critique de la fonction `random_abr`

Rappelez-vous la fonction `random_abr` : on insère dans un ABR initialement vide une permutation aléatoire des entiers de 0 à $n-1$. Combien y a-t-il de telles permutations ? Il y en a $n!$. Mais $C_n = o(n!)$, beaucoup de permutations différentes fournissent le même ABR "aléatoire".

Regardez la fonction `inserer` . En partant d'un arbre initialement vide et en y insérant des objets, c'est le premier objet inséré qui est la racine de l'arbre final.

Question 1 : Quelle est la probabilité que la fonction `random_abr` renvoie un ABR dont la racine est 0 ?

Réponse : C'est la probabilité qu'une permutation s des entiers $0, \dots, n-1$ vérifie $s[0] = 0$, c'est à dire $\frac{1}{n}$.

Question 2 : On met la probabilité uniforme sur l'ensemble des ABR dont les noeuds sont $0, \dots, n-1$. Quelle est la probabilité qu'un ABR ait pour racine 0 ?

Réponse : Les ABR qui nous intéressent sont ceux qui ont un fils gauche vide et un fils droit qui est un ABR de taille $n-1$ dont les noeuds sont $1, 2, \dots, n-1$. Il y en a C_{n-1}. La probabilité qu'un ABR soit de ce type est donc

$$\frac{C_{n-1}}{C_n}$$

Par un calcul direct à partir de l'expression des nombres de Catalan, on voit que cette probabilité est

$$\frac{n+1}{2(2n-1)} \sim \frac{1}{4}$$

Cela ne correspond pas du tout à la probabilité qui concerne la fonction `random_abr` ! La fonction `random_abr` ne renvoie donc pas un ABR aléatoire.

4.3 Quelle est la racine d'un ABR aléatoire ?

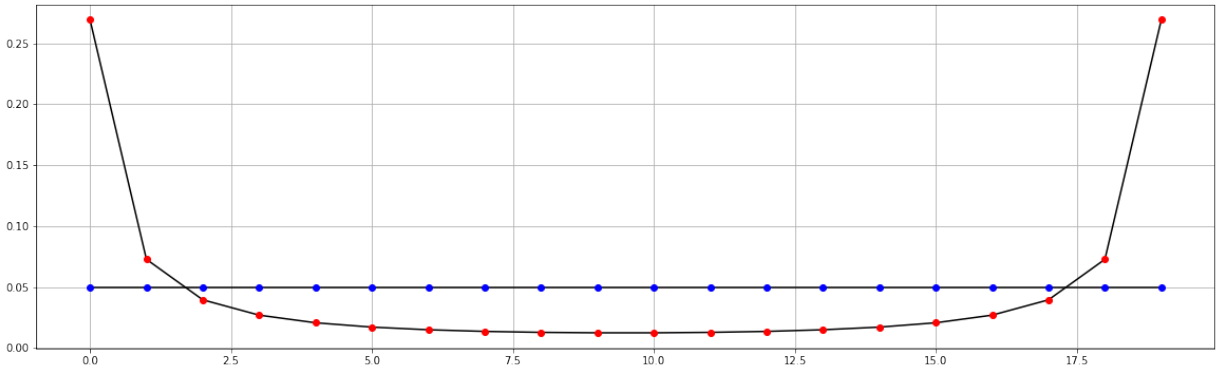
Histoire de bien voir que notre fonction `random_abr` est très mauvaise, quelle est la probabilité que la racine d'un ABR aléatoire ait une certaine valeur ? Quelle est la racine la plus probable ? La moins probable ?

Supposons que l'ensemble des noeuds est l'ensemble des entiers entre 0 et $n-1$. Soit $0 \leq k \leq n-1$. Quelle est la probabilité qu'un ABR ait k pour racine ? Il y a C_k fils gauches possibles et C_{n-1-k} fils droits possibles. La probabilité cherchée est donc

$$p_k = \frac{C_k C_{n-1-k}}{C_n}$$

Pas question de rechercher le minimum et le maximum de cette quantité. Contentons-nous d'une simulation numérique. En bleu, la probabilité (constante) qu'un arbre **renvoyé par `random_abr`** ait k pour racine. En rouge, la probabilité réelle.

Entrée [72]:
```
n = 20
cs = [catalan(k) for k in range(n + 1)]
s = [cs[k] * cs[n - 1 - k] / cs[n] for k in range(n)]
s1 = n * [1 / n]
plt.plot(s, 'k')
plt.plot(s, 'or')
plt.plot(s1, 'k')
plt.plot(s1, 'ob')
plt.grid()
```



La probabiité maximale survient pour $k = 0$ et $k = n - 1$. Il y a plus d'une chance sur 2 que la racine d'un ABR aléatoire n'ait qu'un fils ! La probabilité minimale a l'air de survenir pour k environ égal à $\frac{n}{2}$ ... et, définitivement, la fonction `random_abr` ne renvoie pas un ABR aléatoire.

4.4 Mais alors, comment fabriquer un ABR aléatoire ?

Commençons par le résultat suivant :

Proposition Soit t un arbre binaire à n noeuds. Soit E un ensemble totalement ordonné ayant n éléments. Il existe un et un seul étiquetage des noeuds de t par les éléments de E qui fasse de t un ABR.

Démonstration : On fait une récurrence sur n.

- Si $n = 0$ c'est évident vu qu'il n'y a rien à étiqueter.
- Soit $n \geq 0$. Supposons la propriété vraie pour tous les entiers inférieurs ou égaux à n. Soit t un arbre binaire à $n+1$ noeuds. Soit $E = \{x_0, \dots, x_n\}$ un ensemble totalement ordonné, où $x_0 < \dots < x_n$. Soit k le nombre de noeuds du fils gauche de t, $0 \leq k \leq n$. La racine de t doit être supérieure à k éléments de E et inférieure à $n-k$ éléments de E. La racine de t est donc x_k.

Il faut ensuite

- Étiqueter le fils gauche de t avec $x_0, \dots, x_{k-1}$. Par l'hypothèse de récurrence il y a une et une seule façon de faire cela.
- Étiqueter le fils droit de t avec $x_{k+1}, \dots, x_n$. Par l'hypothèse de récurrence il y a une et une seule façon de faire cela.

On a donc bien un et un seul étiquetage qui fasse de t un ABR.

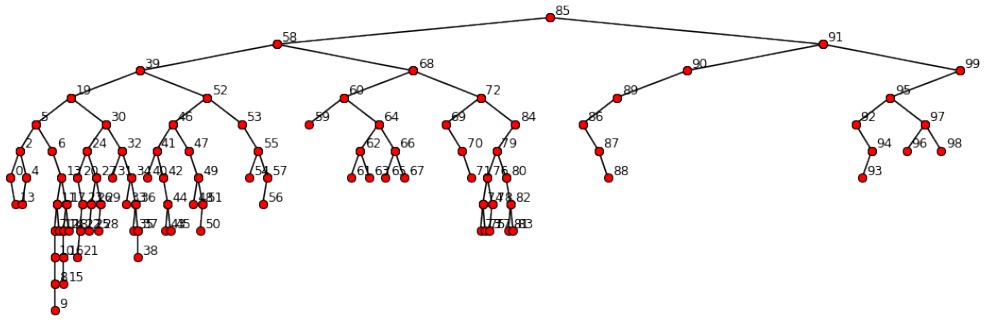
Voici la fonction d'étiquetage. Elle suppose que l'ensemble E est donné par la liste de ses éléments **dans l'ordre croissant**.

Entrée [73]:
```
def etiqueter(t, E):
    if est_vide(t): return vide()
    else:
        u, v = fg(t), fd(t)
        k = nombre_noeuds(u)
        u1 = etiqueter(u, [e for e in E if e < E[k]])
        v1 = etiqueter(v, [e for e in E if e > E[k]])
        return arbre(E[k], u1, v1)
```

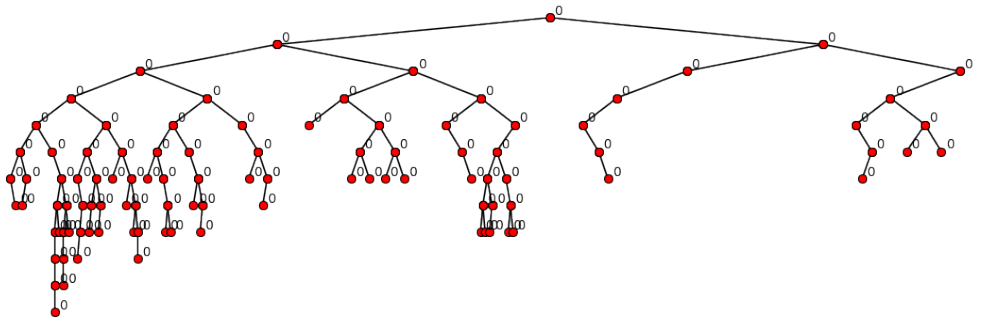
Comment tester cette fonction ? Idée, fabriquons un ABR "aléatoire" avec `random_abr` . Puis effaçons ses noeuds. Enfin appelons `etiqueter` . On devrait retrouver l'ABR de départ !

Entrée [74]:
```
def effacer(t, z):
    if est_vide(t): return vide()
    else:
        u, v = fg(t), fd(t)
        u1 = effacer(u, z)
        v1 = effacer(v, z)
        return arbre(z, u1, v1)
```

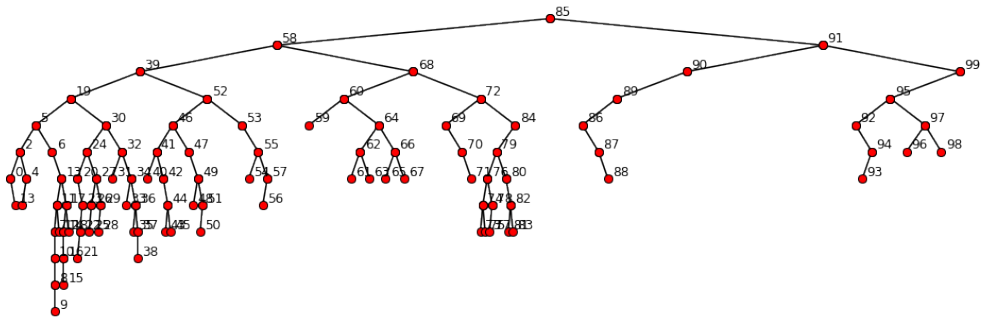
Entrée [75]:
```
t = random_abr(100)
dessiner(t)
```



Entrée [76]:
```
t1 = effacer(t, 0)
dessiner(t1)
```



Entrée [77]:
```
t2 = etiqueter(t1, list(range(100)))
dessiner(t2)
```



Entrée [78]:
```
print(t == t2)
```

```
True
```

Conséquence : Pour fabriquer un ABR aléatoire dont les noeuds sont les éléments de E, de cardinal n :

1. On fabrique un arbre aléatoire (pas forcément "de recherche") ayant n noeuds.
2. On étiquette cet arbre avec les éléments de E, il y a une seule façon de faire cela.

Reste à savoir comment fabriquer un arbre binaire aléatoire. Ceci est un sujet à part entière. Si vous voulez vraiment savoir, consultez le notebook intitulé "arbres binaires aléatoires" ...