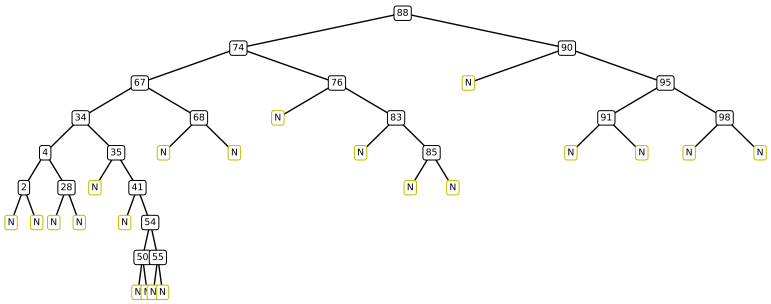
Arbre binaire de recherche

1. Introduction

On s'intéresse à un moyen de mémoriser des nombres entiers (tous différents) par ordre croissant, de manière à rendre la recherche d'un de ces nombres la plus efficace possible. Un arbre binaire de recherche, dont la figure suivante montre un exemple, est une structure de données permettant de le faire :



Chaque nœud de cet arbre contient une clé, qui est un nombre entier dont la valeur est affichée sur la représentation ci-dessus. Par exemple, la clé du nœud racine de l'arbre a la valeur 88. Dans un arbre binaire, chaque nœud a soit deux nœuds descendants (gauche et droite), soit est un nœud nul (noté N sur la figure). Les feuilles de cet arbre sont donc des nœuds nuls. Un arbre binaire de recherche possède la propriété suivante : pour tout nœud dont la clé a la valeur x, les nœuds du sous-arbre de gauche ont des clés strictement inférieures à x et ceux du sous-arbre de droite ont des clés strictement supérieures à x.

La recherche d'un nombre entier y se fait en partant de la racine. Ce nombre est comparé à la clé x du nœud puis on se déplace vers le nœud enfant de gauche si $y < x$ ou le nœud enfant de droite si $y > x$. Si on rencontre un nœud dont la clé est y, la recherche est terminée. Si on rencontre un nœud nul, l'entier recherché n'est pas dans l'arbre. Par exemple, lorsqu'on recherche la valeur 91 dans l'arbre ci-dessus, on parcourt l'arbre en suivant les nœuds de clés 88, 90, 95 et 91. Si on recherche la valeur 73, on parcourt les nœuds de clés 88, 74, 67, 68 puis on atteint le nœud nul enfant de droite de ce dernier.

Si l'arbre est bien équilibré, sa hauteur est en moyenne de l'ordre de $ln(N)$. La recherche d'un nombre entier, ou l'insertion d'un nouveau nombre, se fait donc en un temps $\Theta(ln(N))$.

Le minimum d'un arbre (ou d'un sous-arbre) est le nœud de clé minimale que contient cet arbre. Dans l'exemple ci-dessus, le minimum de l'arbre complet est le nœud de clé 2, celui du sous-arbre descendant du nœud de clé 90 est lui-même.

On appelle *successeur* d'un nœud celui qui possède la plus petite clé supérieure à la clé de ce nœud. Par exemple, le successeur du nœud de clé 4 est celui de clé 28. Si l'enfant de droite d'un nœud n'est pas nul, son successeur est le nœud minimal du sous-arbre de droite. Si l'enfant de droite d'un nœud est nul, le successeur du nœud est le premier ancêtre dont l'enfant de gauche est aussi un ancêtre de ce nœud. Par exemple, le successeur du nœud de clé 68 est celui de clé 74. Il s'obtient en remontant l'arbre jusqu'à rencontrer un nœud qui est l'enfant de gauche de son parent, et ce parent est le nœud recherché.

2. Implémentation de l'arbre

En Python, les arbres binaires peuvent être aisément implémentés au moyen de *dictionnaires*. Un dictionnaire est une structure de données qui associe à une clé (par exemple une chaîne de caractère) un objet. Pour retrouver un objet dans le dictionnaire, il faut connaître sa clé (ne pas confondre avec la clé d'un nœud de l'arbre de recherche). Voici par exemple comment créer un dictionnaire comportant 3 paires (clé,objet) :

```
D = {'a':0,'x':0.55,'c':'bonjour'}
```

La clé `a` (chaîne de caractères) est associée à l'objet `0` (entier). La clé `'x'` est associée à l'objet `0.55` (flottant). Pour accéder à un objet par sa clé, on utilise la syntaxe suivante :

```
D['a']
>>> 0
```

Les dictionnaires sont des objets mutables, c'est-à-dire qu'on peut modifier un dictionnaire sans changer son emplacement en mémoire, par exemple :

```
D['a']=1
```

a pour effet de changer l'objet associé à la clé `'a'` sans changer le dictionnaire lui-même. On peut donc considérer que le nom `D` fait référence a un dictionnaire en mémoire.

Un nœud de l'arbre de recherche binaire sera représenté par un dictionnaire de la forme suivante :

```
noeud = {'cle':x,'gauche':None,'droite':None,parent:'None'}
```

La chaîne (de caractères) `'cle'` permet d'accéder à la clé du nœud. La chaîne `'gauche'` permet d'accéder au nœud enfant de gauche, qui est l'objet `None` s'il s'agit d'un nœud nul. La chaîne `'droite'` permet d'accéder au nœud enfant de droite et la chaîne `parent` au nœud parent. Le nœud racine de l'arbre a l'objet `None` pour parent mais la racine est un nœud comme un autre.

Supposons qu'on ait créé un dictionnaire nommé `noeud_gauche` contenant le nœud à placer à gauche. On pourra écrire :

```
noeud['gauche'] = noeud_gauche
```

Cette opération a pour effet d'associer au nom `gauche` le dictionnaire `noeud_gauche`. En raison du caractère mutable des dictionnaires, il sera possible de modifier le dictionnaire `noeud_gauche` après qu'il ait été placé dans le nœud `noeud`. Cette propriété est bien sûr nécessaire pour implémenter un arbre car il faut pouvoir modifier un nœud quelconque de l'arbre sans avoir à le reconstruire entièrement.

3. Représentation graphique de l'arbre

La manière la plus simple de parcourir entièrement un arbre est d'utiliser la récursivité. Par exemple, la fonction suivante permet d'obtenir une représentation graphique de l'arbre dans une fenêtre graphique de matplotlib :

```
import matplotlib.pyplot as plt

def affichage_arbre(noeud,x,y,dx,dy):
    if noeud==None:
        text = plt.text(x,y,'N',ha='center',va='center')
        text.set_bbox(dict(boxstyle="round",facecolor='w',edgecolor='y',alpha=1.0))
    else:
        text = plt.text(x,y,str(noeud['cle']),ha='center',va='center')
        text.set_bbox(dict(boxstyle="round",facecolor='w',edgecolor='k',alpha=1.0))
        plt.plot([x,x-dx],[y,y-dy],'k')
        affichage_arbre(noeud['gauche'],x-dx,y-dy,dx*0.6,dy)
        plt.plot([x,x+dx],[y,y-dy],'k')
        affichage_arbre(noeud['droite'],x+dx,y-dy,dx*0.6,dy)
```

Les arguments à transmettre à cette fonction sont `noeud`, le nœud à afficher, `x,y` les coordonnées de ce nœud et `dx,dy` le décalage (en valeur absolu) à appliquer pour parvenir à un nœud enfant. À chaque appel récursif, on réduit la valeur du décalage horizontal, ce qui limite le risque de chevauchement. Il est néanmoins difficile de représenter un arbre très grand sans chevauchement des branches.

4. Travaux pratiques

On se propose de construire un arbre binaire de recherche en insérant des nombres entiers dans un ordre aléatoire. On doit commencer par écrire une fonction qui permet d'insérer un nouveau nœud dans un arbre.

[1] Écrire une fonction `inserer_valeur(arbre,x)` qui insère le nombre entier `x` dans l'arbre défini par le dictionnaire `arbre`. On conviendra qu'un arbre transmis en argument à cette fonction doit comporter au minimum un nœud racine avec une clé associée.

[2] Au moyen de la fonction `random.randint`, générer une liste de N nombres entiers (par ex. N=20), tous différents, compris entre 0 et 100. Ces nombres devront être rangés dans un ordre aléatoire.

[3] Générer un arbre binaire de recherche à partir de la liste précédente.

[4] Afficher l'arbre avec la fonction `affichage_arbre` donnnée ci-dessus.

[5] Écrire une fonction `parcours_arbre(noeud)` qui parcourt récursivement l'arbre afin d'afficher les clés par valeur croissante.

[6] Écrire une fonction `recherche_arbre(noeud,x)` qui recherche la valeur `x` dans un arbre. Cette fonction doit renvoyer `'oui'` si la valeur figure dans l'arbre, `'non'` dans le cas contraire. Tester cette fonction sur toutes les valeurs de 0 à 100.

[7] Modifier la fonction précédente afin qu'elle renvoie, en plus du résultat de la recherche, le nombre de nœuds parcourus pour parvenir au nombre cherché.

[8] Écrire une fonction `minimum_arbre(arbre)` qui renvoie le nœud de l'arbre dont la clé est minimale.

[9] Écrire une fonction `successeur_arbre(noeud)` qui renvoie le successeur d'un nœud. Utiliser cette fonction pour obtenir le successeur du nœud minimal.

[10] Écrire une fonction qui affiche la clé du successeur d'un nœud dont la clé est donnée.

5. Solution

L'insertion d'une valeur dans un arbre peut être réalisée par récursivité :

```
def inserer_valeur(arbre,val):
    if val < arbre['cle']:
        if arbre['gauche']==None:
            arbre['gauche'] =  {'cle':val,'gauche':None,'droite':None,'parent':arbre}
        else:
            inserer_valeur(arbre['gauche'],val)
    elif val > arbre['cle']:
        if arbre['droite']==None:
            arbre['droite'] = {'cle':val,'gauche':None,'droite':None,'parent':arbre}
        else:
            inserer_valeur(arbre['droite'],val)
```

Remarque : le paramètre `arbre` de cette fonction désigne en fait un sous-arbre quelconque de l'arbre en cours de construction. C'est seulement lors de l'appel depuis l'extérieur qu'il s'agit de l'arbre complet.

Pour générer N nombres différents, on les place dans une liste. Chaque nouveau tirage aléatoire est répété tant que le nombre tiré se trouve déjà dans la liste.

```
P=100
N=20
liste = [random.randint(0,P-1)]
for i in range(1,N):
    x = random.randint(0,P-1)
    while x in liste:
        x = random.randint(0,P-1)
    liste.append(x)
```

On doit tout d'abord créer un arbre ne comportant que la racine avec pour clé le premier élément de la liste, puis appeler la fonction `inserer_valeur` pour insérer dans l'arbre les autres éléments de la liste :

```
arbre = {'cle':liste[0],'gauche':None,'droite':None,'parent':None}
for i in range(1,N):
    inserer_valeur(arbre,liste[i])
```

Le parcours de l'arbre se fait récursivement. Pour chaque nœud, on doit parcourir le sous-arbre à gauche, afficer la clé, puis parcourir le sous-arbre à droite :

```
def parcours_arbre(noeud):
    if noeud!=None:
        parcours_arbre(noeud['gauche'])
        print(noeud['cle'])
        parcours_arbre(noeud['droite'])
```

Voici la fonction qui effectue la recherche récursive d'une clé :

```
def recherche_arbre(noeud,x,k):
    if noeud==None:
        return ('non',k)
    elif noeud['cle']==x:
        return ('oui',k)
    else:
        if x < noeud['cle']:
            return recherche_arbre(noeud['gauche'],x,k+1)
        else:
            return recherche_arbre(noeud['droite'],x,k+1)
```

Cette fonction renvoie 'oui' ou 'non', si la valeur est présente dans l'arbre ou pas, et le nombre de nœuds parcourus. Il faut remarquer que chaque appel récursif doit être associé à un `return` qui permet de renvoyer à la fonction appelante le résutat de la recherche. Dans le cas présent, la commande `return` est placée juste avant l'appel récursif car aucune action n'est nécessaire lorsque cet appel est accompli.

Voici la fonction qui renvoie le minimum d'un arbre. Elle consiste à se déplacer (itérativement) en suivant l'enfant de gauche jusqu'à rencontrer un nœud vide.

```
def minimum_arbre(noeud):
    while noeud['gauche']!=None:
        noeud = noeud['gauche']
    return noeud
```

Voici la fonction qui renvoie le successeur d'un nœud :

```
def successeur_arbre(noeud):
    if noeud['droite']!=None:
        return minimum_arbre(noeud['droite'])
    else:
        parent = noeud['parent']
        while parent!=None and noeud == parent['droite']:
            noeud = parent
            parent = parent['parent']
        return parent
```