Python实现二叉树

2016年10月16日 14:16:26 阅读数：3890

二叉树是数据结构中非常重要的一种数据结构，在用Python搭建决策树模型时，发现需要先实现多叉树，于是回过头来，看了遍二叉树，有了如下的成果。

我先构建了这样一个节点型数据，他有这样几个属性和功能：

（1）属性：名称，数据，左子节点，右子节点，父节点，子节点个数（度）；

（2）方法：添加子节点和删除子节点，并且其子节点个数随之变化，子节点的父节点变成该节点。

我需要使我的二叉树有这样功能：

（1）属性：深度、根节点、所有可插节点字典{名称：数据}、所有节点字典{名称：数据}

（2）方法：

- 利用现有的一个节点或一个有多层子节点的节点来生成树结构；
- 随时计算树的深度；
- 利用树的某一个节点生成子树；
- 在树的某一个可插入节点上插入或删除子树；
- 打印树结构

下面上代码

[python]

```python
#-*-coding:utf-8-*-
'''''
created by zwg in 2016-10-7
'''
import copy
class node:
    def __init__(self,name,data):
        self.data=data
        self.name=name
        self.Rchild=None
        self.Lchild=None
        self.child_number=0
```

```
        self.parent=None
    def add_Rchild(self,node):
        if self.Rchild is not None:
            self.Rchild=node
        else:
            self.Rchild=node
            self.child_number+=1
        node.set_parent(self)
    def drop_Rchild(self):
        self.Rchild=None
        self.child_number-=1
    def set_parent(self,node):
        self.parent=node
    def add_Lchild(self,node):
        if self.Lchild is not None:
            self.Lchild=node
        else:
            self.Lchild=node
            self.child_number+=1
        node.set_parent(self)
    def drop_Lchild(self):
        self.Lchild=None
        self.child_number-=1
                                                载
class tree:
    def __init__(self,node):
        '''''
        初始化
        可使用一个无子节点的作为树的根
        也可使用一个本身就包含了多层树结构的节点来构建树
        每个节点包含名称和数据两个主要属性，以及两个节点
        all_node为所用节点
        enable_node为可插节点
        '''
        self.parent=node
        self.depth=1
        self.all_node={node.name:node}
        self.enable_node={node.name:node}
        c1=node.Rchild
        c2=node.Lchild
```

```
        C=[c1,c2]
        B=[i for i in C if i is not None]
        if len(B)==2:
            del self.enable_node[node.name]
        while len(B)!=0:
            self.depth+=1
            C=copy.copy(B)
            for i in B:
                C.remove(i)
                self.all_node[i.name]=i
                if i.child_number!=2:
                    self.enable_node[i.name]=i
                if i.Rchild is not None:
                    C.append(i.Rchild)
                if i.Lchild is not None:
                    C.append(i.Lchild)
            B=copy.copy(C)
    def get_depth(self):
        '''''
        计算树的深度
        '''
        depth=1
        node=self.parent
        c1=node.Rchild
        c2=node.Lchild
        C=[c1,c2]
        B=[i for i in C if i is not None]
        while len(B)!=0:
            depth+=1
            C=copy.copy(B)
            for i in B:
                C.remove(i)
                if i.Rchild is not None:
                    C.append(i.Rchild)
                if i.Lchild is not None:
                    C.append(i.Lchild)
            B=copy.copy(C)
        return depth
    def show(self):
        '''''
```

```
93.         打印树结构
94.         '''
95.         a=[copy.deepcopy(self.parent)]
96.         n=copy.deepcopy(self.depth)
97.         m=copy.copy(n)
98.         print self.parent.name.center(2**n*m)
99.         while n>=1:
100.            b=[]
101.            n-=1
102.            for i in a:
103.                if i is not None:
104.                    c1=i.Lchild
105.                    b.append(c1)
106.                    if c1 is not None:
107.                        print c1.name.center(2**n*m),
108.                    else:
109.                        print ''.center(2**n*m),
110.                    c2=i.Rchild
111.                    b.append(c2)
112.                    if c2 is not None:
113.                        print c2.name.center(2**n*m),
114.                    else:
115.                        print ''.center(2**n*m),
116.                else:
117.                    print ''.center(2**n*m),
118.                    print ''.center(2**n*m),
119.            a=copy.deepcopy(b)
120.            print '\n'
121.        #del a,n,b
122.    def generate_childtree(self,child_name):
123.        '''''
124.        选取child_name这个节点生成子树，
125.        子树的根节点为child_node
126.        '''
127.        child_node=self.all_node[child_name]
128.        child_tree=tree(child_node)
129.        return child_tree
130.    def add_child_tree(self,parent_name,child_tree,RL='right'):
131.        '''''
132.        增加子树
```

```
        子树可以是单节点树，也可以是多层节点的树
        '''
        L1=child_tree.all_node
        L2=child_tree.enable_node
        L4=child_tree.parent
        parent_node=self.all_node[parent_name]
        if RL=='right':
            parent_node.add_Rchild(L4)
        if RL=='left':
            parent_node.add_Lchild(L4)
        for i in L1.keys():
            self.all_node[i]=L1[i]
        for i in L2.keys():
            self.enable_node[i]=L2[i]
        if parent_node.child_number==2:
            self.enable_node.pop(parent_node)
        self.depth=self.get_depth()
    def drop_child_tree(self,child_name):
        '''''
        删除子树，child_name为删除的所在节点名
        该节点及其后面所有子节点一并删除
        '''
        child_node=self.all_node[child_name]
        child_tree=tree(child_node)
        L1=child_tree.all_node
        L2=child_tree.enable_node
        parent_node=child_node.parent
        if parent_node.Rchild==child_node:
            parent_node.drop_Rchild()
        else:
            parent_node.drop_Lchild()
        for i in L1.keys():
            self.all_node.pop(L1[i].name)
        for i in L2:
            self.enable_node.pop(L1[i].name)
        if not self.enable_node.has_key(parent_node.name):
            self.enable_node[parent_node.name]=parent_node
        self.depth=self.get_depth()

```

```
173.
174.
175.
176. if __name__=='__main__':
177.     a=node('a',1)
178.     a1=node('a1',2)
179.     a2=node('a2',2)
180.     a11=node('a11',3)
181.     a12=node('a12',3)
182.     a21=node('a21',3)
183.     a111=node('a111',4)
184.     a112=node('a112',4)
185.     a211=node('a211',4)
186.     a212=node('a212',4)
187.     a11.add_Lchild(a111)
188.     a11.add_Rchild(a112)
189.     a21.add_Lchild(a211)
190.     a21.add_Rchild(a212)
191.     a.add_Lchild(a1)
192.     a.add_Rchild(a2)
193.     a1.add_Rchild(a11)
194.     a1.add_Lchild(a12)
195.     a2.add_Rchild(a21)
196.     '''''
197.     验证node的属性及方法是否正确
198.     print a.Lchild.name
199.     print a.Rchild.name
200.     print a.child_number
201.     print a.parent
202.     print a1.Rchild.name
203.     print a.Rchild.Rchild.name
204.     '''
205.
206.
207.     #生成node关于a的树，a有4层
208.     T=tree(a)
209.     print T.depth
210.     print T.all_node.keys()
211.     print T.enable_node.keys()
212.     #T.show()#打印树
```

```
213.
214.     #生成node关于b的树，a有两层
215.     b=node('b',5);b1=node('b1',6);b2=node('b2',6)
216.     b.add_Lchild(b1);b.add_Rchild(b2)
217.     b_tree=tree(b)
218.     #b_tree.show()#打印树
219.
220.     #生成树T的子树，以a1为节点，a1的深度为3
221.     t1=T.generate_childtree('a1')
222.     print t1.depth
223.     print t1.all_node.keys()
224.     print t1.enable_node.keys()
225.     #t1.show()#打印树
226.
227.
228.     #增加子树，在a111后面加上子树b_tree，这时树的高度为6
229.     T.add_child_tree('a111',b_tree,'left')
230.     print T.depth
231.     print T.enable_node.keys()
232.     print T.all_node.keys()
233.     #T.show()#打印树
234.
235.     #删除以节点b开始的子树，还原为原来的样子
236.     T.drop_child_tree('b')
237.     print T.depth
238.     print T.enable_node.keys()
239.     print T.all_node.keys()
240.     #T.show()#打印树
```

结果：

```
4
['a', 'a21', 'a112', 'a11', 'a12', 'a211', 'a212', 'a1', 'a2', 'a111']
['a111', 'a112', 'a12', 'a212', 'a211', 'a2']
3
['a1', 'a111', 'a112', 'a11', 'a12']
['a111', 'a112', 'a12']
6
['a111', 'a112', 'a12', 'a212', 'a211', 'b1', 'b2', 'a2']
['a', 'a21', 'a112', 'a11', 'a12', 'a211', 'a212', 'a1', 'a2', 'b1', 'b2', 'b', 'a111']
4
```

```
['a111', 'a112', 'a12', 'a212', 'a211', 'a2']
['a', 'a21', 'a112', 'a11', 'a12', 'a211', 'a212', 'a1', 'a2', 'a111']
```