二叉树及其应用

目录

[二叉树及其应用 1](#_Toc437362197)

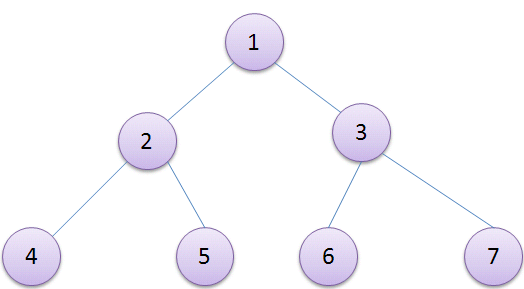
[问题1：已知二叉树的前序序列和中序序列，编写建立二叉树的算法（递归、非递归两个） 2](#_Toc437362198)

[问题2：编写二叉树的四种遍历算法（前序、中序、后序递归、非递归，层序遍历）；要求输出二叉树，二叉树结点个数不少于8个。 4](#_Toc437362199)

[问题3：求任意两点的最近公共祖先。 6](#_Toc437362200)

问题1：已知二叉树的前序序列和中序序列，编写建立二叉树的算法（递归、非递归两个）

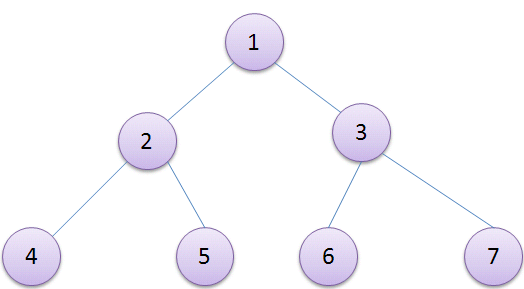
1. 递归算法：

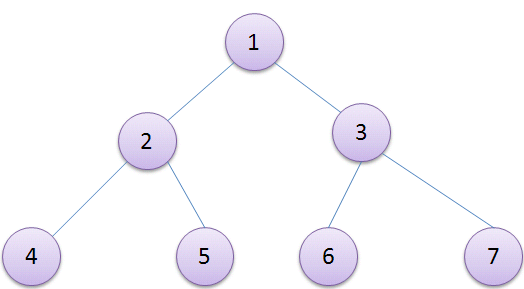


如上图：

先序：1 2 4 5 3 6 7

中序：4 2 5 1 6 3 7

 先序遍历的第一个节点一定是整个树的根节点，而且根的左子树中的节点在中序遍历中一定出现在根的左边，同理，右子树中的节点在中序遍历中一定出现在根的右边。这样就可以把这棵树通过1号节点分成两颗子树，其中红色部分就是左子树的先序遍历和中序遍历。

蓝色为右子树的线序遍历和中序遍历

所以这样一直递归下去就可以构建满足条件的树，且这是唯一满足条件的树。

核心代码：

void build\_by\_DLR\_LDR(char dlr[], char ldr[], int l, int r, Node \*now)

{

now->data = dlr[size];

int p;

for(p = l; p <= r; p++)

{

if(ldr[p] == now->data)

break;

}

if(p > l)

{

now->lson = new Node();

size++;

build\_by\_DLR\_LDR(dlr, ldr, l, p-1, now->lson);

}

if(p < r)

{

now->rson = new Node();

size++;

build\_by\_DLR\_LDR(dlr, ldr, p+1, r, now->rson);

}

}

1. 非递归算法：

分析: 根据先序遍历的性质，在先序遍历中靠后的项肯定是他之前的某一项的孩子，如果他是之前某一项的左孩子，那么他肯定紧跟着它的父亲节点。否则他肯定是之前某个节点的右孩子。而他的父亲节点肯定是第一个中续遍历中在他左边且满足上述条件的节点。这样我们就对于每一个节点确定了他是谁的左孩子或者右孩子，于是就可以建成一颗二叉树了。

核心代码：

void build\_by\_DLR\_LDR2(char dlr[], char ldr[], int len, Node \*root)

{

deque< pair<Node\*, int> > deq;

deq.clear();

Node \*now;

pair<Node\*, int> tmp;

int p;

for(int i = 0; i < len; i++)

{

if(i == 0)

now = root;

else

now = new Node();

now->data = dlr[i];

for(p = 0; p < len; p++)

if(ldr[p] == now->data)

break;

tmp = make\_pair(now, p);

if(i == 0 || deq.empty())

{

deq.push\_front(tmp);

continue;

}

if(deq[0].second > p)

{

deq[0].first->lson = now;

deq.push\_front(tmp);

}

else

{

while(deq.size() > 1 && deq[1].second < p) deq.pop\_front();

deq[0].first->rson = now;

deq.pop\_front();

deq.push\_front(tmp);

}

}

}

问题2：编写二叉树的四种遍历算法（前序、中序、后序递归、非递归，层序遍历）；要求输出二叉树，二叉树结点个数不少于8个。

1. 前序

线序遍历先输出当前结点，再递归遍历左子树，右子树。

核心代码：

void show\_DLR(Node \*now)

{

now->show();

if(now->lson != NULL)

show\_DLR(now->lson);

if(now->rson != NULL)

show\_DLR(now->rson);

}

1. 中序

中序遍历先遍历左子树，在输出当前结点，再遍历右子树。

核心代码：

void show\_LDR(Node \*now)

{

if(now->lson != NULL)

show\_LDR(now->lson);

now->show();

if(now->rson != NULL)

show\_LDR(now->rson);

}

1. 后序

后序遍历先遍历左子树，右子树，最后输出当前结点的信息。

核心代码：

void show\_LRD(Node \*now)

{

if(now->lson != NULL)

show\_LRD(now->lson);

if(now->rson != NULL)

show\_LRD(now->rson);

now->show();

}

非递归：

非递归就需要用一个栈来存储还没有访问的节点，一个节点需要出栈3次，第一次是用来访问左子树，第二次右子树，第三次自己。

核心代码：

void show\_LRD2(Node \*now)

{

deque<pair<Node\*, int> > deq;

deq.clear();

deq.push\_back(make\_pair(now, 0));

pair<Node\* , int> tmp;

while(!deq.empty())

{

tmp = deq.front();

deq.pop\_front();

if(tmp.second <= 0 && tmp.first->lson != NULL)

{

deq.push\_front(make\_pair(tmp.first, 1));

deq.push\_front(make\_pair(tmp.first->lson, 0));

}

else if(tmp.second <= 1 && tmp.first->rson != NULL)

{

deq.push\_front(make\_pair(tmp.first, 2));

deq.push\_front(make\_pair(tmp.first->rson, 0));

}

else

{

tmp.first->show();

}

}

}

1. 层序

用一个队列来记录即将访问的点，由于队列的先进先出的特点，就可以做到一层一层的访问树的节点了。

核心代码：

void show\_layer(Node \*now)

{

deque<Node\*> deq;

deq.clear();

deq.push\_back(now);

while(!deq.empty())

{

now = deq.front();

deq.pop\_front();

now->show();

if(now->lson != NULL)

deq.push\_back(now->lson);

if(now->rson != NULL)

deq.push\_back(now->rson);

}

}

问题3：求任意两点的最近公共祖先。

求LCA，最简单的方法我们可以记录一下root 到指定两个点的路径，然后比较路径最长的相同前缀的最后一个就是LCA

核心代码：

int LCA(char d1, char d2)

{

int find = 1;

NDdeq1.clear();

NDdeq2.clear();

DLR(d1, d2, root, find);

int i, l = min(NDdeq1.size(), NDdeq2.size());

// cout << NDdeq1.size() << " " << NDdeq2.size();

// cout << NDdeq2[0]->data;

for(i = 0; i < l; i++)

{

// cout << NDdeq1[i]->data << " " << NDdeq2[i]->data << endl;

if(NDdeq1[i]->data != NDdeq2[i]->data)

break;

}

return NDdeq1[i-1]->data;

}

void DLR(char d1, char d2, Node\* now, int &find)

{

if(find % 6 == 0) return;

if(find%2 != 0)

{

NDdeq1.push\_back(now);

}

if(find%3 != 0)

{

NDdeq2.push\_back(now);

}

if(now->data == d1)

{

find\*=2;

}

if(now->data == d2)

{

find\*=3;

}

if(now->lson != NULL && find%6 != 0)DLR(d1, d2, now->lson, find);

if(now->rson != NULL && find%6 != 0)DLR(d1, d2, now->rson, find);

if(find%2!=0)

{

NDdeq1.pop\_back();

}

if(find%3!=0)

{

NDdeq2.pop\_back();

}

}

这里我们可以用到一个技巧，设定一个标记find 初值为1，如果找到第一个点那么乘上2，如果找到第二个那么乘上3，这样当find % 6 == 0时就已经找到了两个点，就可以直接寻找答案了。