二叉树遍历算法

首先，个人认为，二叉树是很能体会递归算法思想的，因为二叉树的结构是leftTree->root<-rightTree，对于每个非叶子节点，该规律都适用，因此关于二叉树的很多算法也都能用递归思想搞定。递归的优点在于代码简洁，但效率却是问题。其次，对于各种顺序的遍历，又有着相应的非递归算法，非递归算法的代码量相对大一点，但更容易掌控，而且效率更优。

先看节点结构：

```
struct Bitree{
  int val;
  Bitree *left, *right;
  Bitree(int x):val(x), left(nullptr), right(nullptr){
  };
};
```

1. 中序遍历

- 递归算法

  显然，中序遍历的顺序为leftTree， root， rightTree，显然先遍历左子树，然后是根，最后右子树。中序遍历的递归算法自然也就出来了。

```
void inOrderTraverse1(Bitree *root){
  if(root){
    inOrderTraverse1(root->left);
    visit(root);
    inOrderTraverse1(root->right);
  }
}
```

- 非递归算法1.

  非递归算法的思想也比较简单，按从左到右进行访问。先指针往左探寻到底，然后观察最后一个非空节点是否有右节点，若有，将该右节点作为新的探寻起点，再进行下一轮的探寻。显然，“一探到底”的思路需要使用stack来帮助缓存之前的节点。

```
void inOrderTraverse2(Bitree *root){
  stack<Bitree *> S;
  S.push(root);
  Bitree *p = root;
  while(!S.empty()){
    p = S.top();
    while(p){
      S.push(p->left);
      p = p->left;
    }
    S.pop();//pop out the nullptr
    if(!S.empty()){
      p = S.top();
      visit(p);
      S.pop();
      S.push(p->right);//push its right child into the stack
    }
  }
}
```

- 非递归算法2

```
void inOrderTraverse3(Bitree *root){
  stack<Bitree *> S;
  Bitree *p = root;
  while(p || !S.empty()){
    if(p){
      S.push(p);
      p = p->left;
    }else{
```


< 2018年4月 >

日	一	二	三	四	五	六
25	26	27	28	29	30	31
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	1	2	3	4	5

导航

博客园
首页
新随笔
联系
订阅XML
管理

统计

随笔 - 5
文章 - 0
评论 - 0
引用 - 0

公告

昵称：5iCoding
园龄：1年1个月
粉丝：1
关注：1
+加关注

搜索

找找看
谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

我的标签

二叉树(4)
二进制(1)
经典算法(1)
排序(1)
数组(1)
递归(1)

随笔档案

2017年4月 (5)

阅读排行榜

1. 二叉树遍历算法(1996)
2. 二叉树的宽度和深度(1762)
3. 二叉树的遍历——Morris(175)
4. 皇后问题(113)
5. 二叉树的生成(68)

```
 9       p = S.top();
10       visit(p);
11       S.pop();
12       p = p->right;
13     }
14   }
15 }
```

个人认为，虽然两种非递归算法的思路完全一样，但非递归算法2比非递归算法1代码要更为简洁，更值得推荐。

2. 前序遍历

- 递归算法

  前序遍历的顺序为root，leftTree，rightTree，直接上代码

```
1 void preOrderTraverse1(Bitree *root){
2   if(root){
3     visit(root);
4     preOrderTraverse1(root->left);
5     preOrderTraverse1(root->right);
6   }
7 }
```

- 非递归算法1

  对于前序遍历的非递归算法，和中序遍历的非递归算法非常相似，不过是在进栈时就访问该节点，而不是之后再访问。由于代码相似，先给出一种

```
 1 void preOrderTraverse2(Bitree *root){
 2   stack<Bitree *> S;
 3   Bitree *p = root;
 4   while(p || !S.empty()){
 5     if(p){
 6       visit(p);
 7       S.push(p);
 8       p = p->left;
 9     }else{
10       p = S.top();
11       S.pop();
12       p = p->right;
13     }
14   }
15 }
```

- 非递归算法2

  该算法采用了和前序遍历相同的思想，即root节点先进栈，root节点出栈时，将其右节点先进栈，然后是左节点进栈。这样，利用栈先进后出的性质，访问顺序自然变为了root，左子树，右子树。

```
 1 void preOrderTraverse3(Bitree *root){
 2   if(!root){
 3     return;
 4   }
 5   stack<Bitree *> S;
 6   Bitree *p = root;
 7   S.push(root);
 8   while(!S.empty()){
 9     p = S.top();
10     visit(p);
11     S.pop();
12     if(p->right){
13       S.push(p->right);
14     }
15     if(p->left){
16       S.push(p->left);
17     }
18   }
19 }
```

3. 后续遍历

- 递归算法

  后续遍历的顺序是leftTree，rightTree和root，因此递归算法也自然出来了

```
void postOrderTraversel(Bitree *root){
  if(root){
    postOrderTraversel(root->left);
    postOrderTraversel(root->right);
    visit(root);
  }
}
```

- 非递归算法1

  和之前中序和前序算法不同，后续遍历的root节点要最后才能被访问，因此，我们若想访问某节点，那么我们需要知道该节点的右节点是否已经被访问过。只有该节点的右节点为null，或者已被访问过，那么该节点才能被访问；否则需要先将右节点访问完。为了判断该节点的右节点是否已经被访问过，需另外设一个记录指针last来指示已经访问过的节点，如果之前访问过的节点last恰为该节点的右节点，说明其右子树已经访问完，应该访问该节点。

```
void postOrderTraverse2(Bitree *root){
  Bitree *last = nullptr;
  Bitree *p = root;
  stack<Bitree *> S;
  while(p || !S.empty()){
    while(p){
      S.push(p);
      p = p->left;
    }
    p = S.top();
    if(p->right && p->right != last){
      p = p->right;
    }else{
      visit(p);
      S.pop();
      last = p;
      p = nullptr;//p needs to be updated to null for next loop
    }
  }
}
```

tip 1：后续遍历中，root节点最后才能被访问到，因此，栈能记录每一个节点的路径，包括叶子节点。这一点性质可用于求解和树的路径有关的问题。

- 非递归算法2

  和前序遍历的非递归算法2一样，这里也给出后续遍历对应的非递归算法2，思路也是类似。由于后序遍历中，根节点要最后才能被访问到，不像前序遍历中刚访问到便可以输出。但在实际查找过程中，我们又只能先从根节点开始查找，才能接着查找左子树和右子树，由此可以再利用栈先进后出的特性来存储根节点。

```
void postOrderTraverse3(Bitree *root){
  if(!root){
    return;
  }
  Bitree *p = root;
  stack<Bitree *> S;
  stack<Bitree *> postOrder;
  S.push(p);
  while(!S.empty()){
    p = S.top();
    postOrder.push(p);
    S.pop();
    if(p->left){
      S.push(p->left);//first IN, later OUT
    }
    if(p->right){
      S.push(p->right);//later IN, first OUT
    }
  }
  while(!postOrder.empty()){
    p = postOrder.top();
    visit(p);
    postOrder.pop();
  }
}
```

4. 层序遍历

层序遍历的思想和之前三种遍历方式不同，需要借助queue来对节点进行缓存，先进队列的节点需要先离开。这和图的BFS思想一样，毕竟树本质上也是一种特殊的图。

```
void levelOrderTraverse(Bitree *root){
  if(!root){
    return;
  }
  queue<Bitree *> Q;
  Bitree *p = nullptr;
  Q.push(root);
  while(!Q.empty()){
    p = Q.front();
    Q.pop();
    visit(p);
    if(p->left){
      Q.push(p->left);
    }
    if(p->right){
      Q.push(p->right);
    }
  }
}
```

Tip 2: 层序遍历思想简单，利用queue来一层一层输出。因此，可用于求解数的宽度和高度。

标签: 二叉树

5iCoding
关注 - 1
粉丝 - 1
+加关注

» 下一篇: 二叉树的生成

posted on 2017-04-05 21:05 5iCoding 阅读(1995) 评论(0) 编辑 收藏

· 写给自学者的入门指南
· 和程序员谈恋爱
· 学会学习
· 优秀技术人的管理陷阱
· 作为一个程序员，数学对你到底有多重要
» 更多知识库文章...