数据结构

主讲：项若曦　助教：申智铭、黄毅
rxxiang@blcu.edu.cn
主楼南329

回顾

- **特殊矩阵的压缩存储**
- **稀疏矩阵的存储及转置**
- **树**
  - 树的定义和基本术语
  - 二叉树定义、ADT、性质（五个性质）

本节内容

- 二叉树的遍历（先序中序后续递归遍历、非递归遍历、层次遍历）
- 基于遍历的算法设计

- 遍历二叉树

遍历二叉树

按一定的搜索路径对树中的每一结点访问且仅访问一次

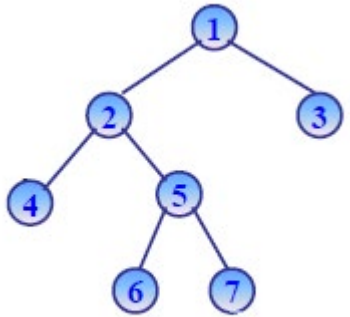
- 遍历时的搜索路线(约定：先左后右，D-根,L-左子树,R-右子树)

深度优先

- 先(根)序遍历：1 2 4 5 6 7 3 (DLR)
- 中(根)序遍历：4 2 6 5 7 1 3 (LDR)
- 后(根)序遍历：4 6 7 5 2 3 1 (LRD)

广度优先

- 层次遍历 ： 1 2 3 4 5 6 7

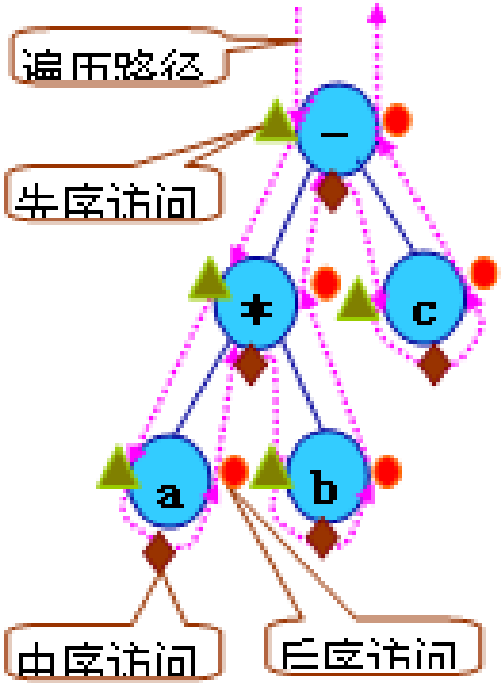


遍历二叉树

- 先/中/后序遍历的区别

  如右图，三者经过的搜索路线是相同的；只是访问结点的时机不同。每一结点在整个搜索路线中会经过3次：

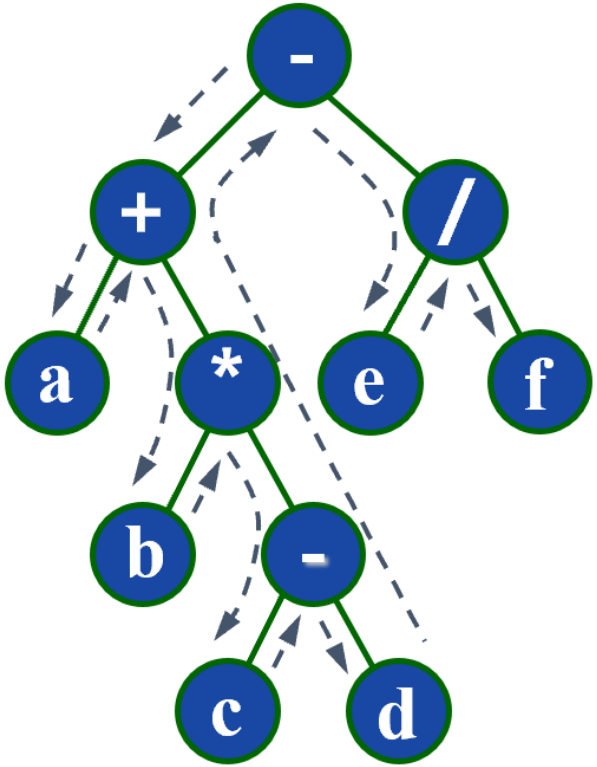
  - 第一次进入到该结点此时访问该结点，称为**先序遍历**
  - 由左子树回溯到该结点此时访问该结点，称为**中序遍历**
  - 由右子树回溯到该结点此时访问该结点，称为**后序遍历**



遍历二叉树

- 遍历算法的递归实现
  - 二叉树的递归定义性质，决定了它的很多算法都可用递归实现，遍历算法就是其中之一。
  - 对于二叉树的遍历，可以不去具体考虑各子问题(左子树、根、右子树)的遍历结果是什么，而去考虑如何由各子问题的求解结果构成原问题(二叉树)的遍历结果——**递归规律的确定**。必须注意的是，当二叉树小到一定程度，即空树时，应直接给出解答——**递归结束条件及处理**。

遍历二叉树

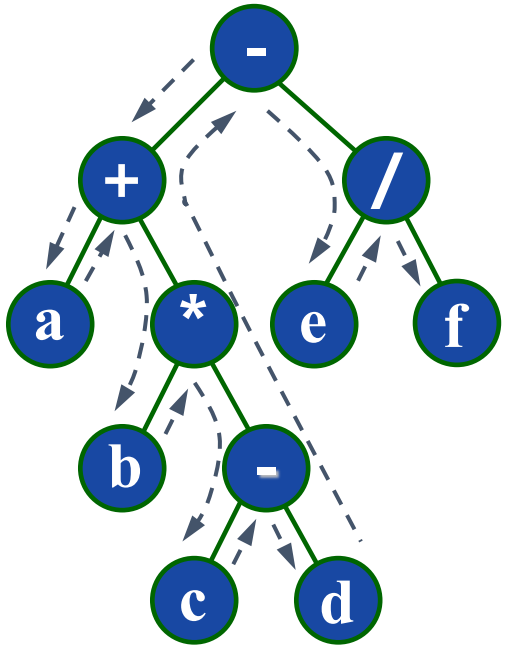


遍历二叉树

先序遍历 (Preorder Traversal)

- 先序遍历二叉树算法的框架是：
  - 若二叉树为空，则空操作；
  - 否则
    - 访问根结点 (V)；
    - 先序遍历左子树 (L)；
    - 先序遍历右子树 (R)。

- 遍历结果 - + a * b - c d / e f



遍历二叉树

二叉树递归的先序遍历算法

```
1. void PreOrder(BiTree T) {//BiTree等同于BiNode*
2.         if (T) {                  //等价于if(T!=NULL)
3.                   cout << T->data;
4.                   PreOrder(T->lchild);
5.                   PreOrder(T->rchild);
6.         }
7. }
```

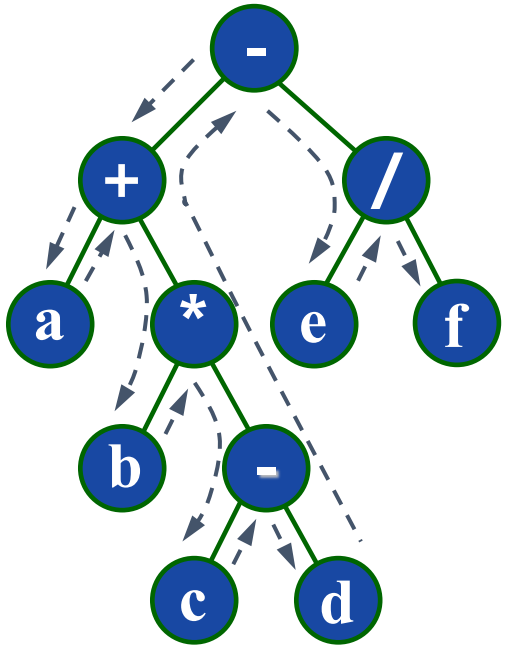
访问结点操作cout，在实际使用时可用相应的操作来替换。

遍历二叉树

中序遍历 (Inorder Traversal)

- 中序遍历二叉树算法的框架是：
  - 若二叉树为空，则空操作；
  - 否则
    - 中序遍历左子树 (L)；
    - 访问根结点 (V)；
    - 中序遍历右子树 (R)。

- 遍历结果 a + b * c - d - e / f



二叉树递归的中序遍历算法

```
1.  void InOrder(BiTree T) {
2.          if (T != NULL) {
3.                  InOrder(T->lchild);
4.                  cout << T->data;
5.                  InOrder(T->rchild);
6.          }
7.  }
```

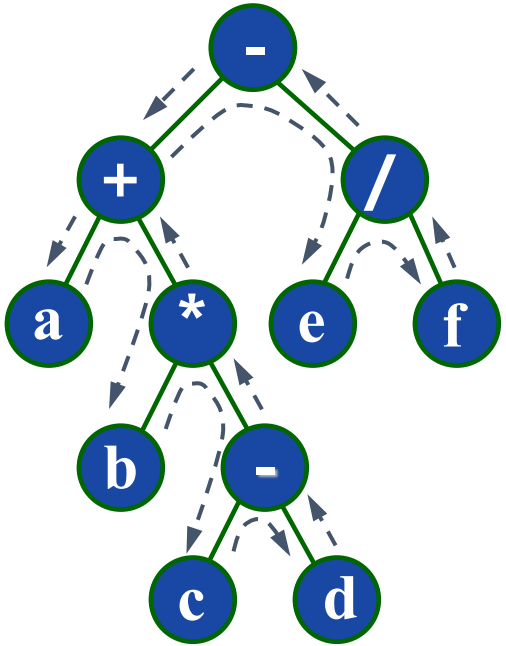
与先序遍历算法相比，访问结点操作cout放在两个子树递归语句之间。

遍历二叉树

后序遍历 (Postorder Traversal)

- 后序遍历二叉树算法的框架是：
  - 若二叉树为空，则空操作；
  - 否则
    - 后序遍历左子树 (L)；
    - 后序遍历右子树 (R)；
    - 访问根结点 (V)。

- 遍历结果a b c d - * + e f / -



二叉树递归的后序遍历算法

```
1.  void PostOrder(BiTree T) {
2.          if (T != NULL) {
3.                  PostOrder(T->lchild);
4.                  PostOrder(T->rchild);
5.                  cout << T->data;
6.          }
7.  }
```

与中序遍历算法相比，cout操作放在两个子树递归后序遍历的最后面。

本节内容

- **二叉树**
  - 基于遍历的算法的应用

基于先/中/后序遍历算法的应用

- 基于先/中/后序遍历的算法应用
  - 基于先序遍历的二叉树(二叉链)的创建
  - 统计二叉树中叶子结点的数目
  - 释放二叉树的所有结点空间
  - 删除并释放二叉树中以元素值为x的结点作为根的各子树
  - 求位于二叉树先序序列中第**k**个位置的结点的值

分析问题本身的特征，选择合适的遍历次序！应用递归编写算法，简洁！
注意：递归结束条件，用递归调用的结果

二叉树-链式存储结构

二叉树的链式存储结构

- 二叉链的类型定义(动态链表)

```
typedef struct BiTNode{
    ElemType data;
    struct BiTNode *lchild, *rchild; // 左右孩子指针
}BiTNode, *BiTree;
```

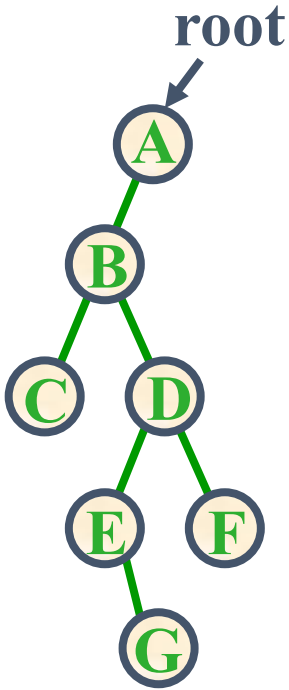
- 说明：若有n个结点，则共有2n个链域；其中n-1不为空，指向孩子；另外n+1个为空链域

基于先/中/后序遍历算法的应用

- **例1 基于先序遍历的二叉树(二叉链)的创建 P.126 算法5.3，图在P121页**

【本例特征】 ABCΦΦDEΦGΦΦFΦΦΦ

如何基于二叉树的先序、中序、后序遍历的递归算法进行问题求解?



基于先/中/后序遍历算法的应用

例1 基于先序遍历的二叉树(二叉链)的创建 P.126 算法5.3

【本例特征】 ABCΦΦDEΦGΦΦFΦΦΦ

如何基于二叉树的先序、中序、后序遍历的递归算法进行问题求解?

【思路】

	先序遍历PreOrderTraverse	创建CreateBiTree
输入	二叉链表示的二叉树的头指针T	带虚结点的先序序列ch
输入的表现方式	参数	由输入设备输入cin>>ch
输出	对结点的访问结果	二叉链表示的二叉树的头指针T
输出的表现方式	由输出设备输出	变参
空树（递归结束）的条件及处理 (直接求解)	T == NULL	ch == END_DATA（表示虚结点的值）
	空	T = NULL
根结点的访问 (子问题直接求解)	Cout<< T->data;	T = new BiTNode; T->data = ch
左子树 (使用递归调用的解)	PreOrderTraverse(T->lchild)	CreateBiTree(T->lchild)
右子树 (使用递归调用的解)	PreOrderTraverse(T->rchild)	CreateBiTree(T->rchild)

例1 基于先序遍历的二叉树(二叉链)的创建 P.126 算法5.3

```
1.  Status CreateBiTree(BiTree& T) {
2.          //按先序次序输入二叉树中结点的值(一个字符)
3.          //空格字符表示空树,构造二叉链表表示的二叉树T
4.          cin>>ch;                              // 如果用空格，输入请用cin.getline()
5.          if (ch ==   'Φ' ) T = NULL; // Φ是空格，或者其他特殊字符。
6.          else {
7.                      if (!(T = new BiTNode))
8.                                  exit(OVERFLOW);
9.                      T->data = ch;
10.                     CreateBiTree(T->lchild);
11.                     CreateBiTree(T->rchild);
12.         }
13.         return OK;
14.} // CreateBiTree
```

例2 统计二叉树中叶子结点的数目

【本例特征】

如何通过全局变量、变参、返回值三种渠道返回处理结果?

【思路】

在遍历二叉树时，对一些特殊的结点(无左右孩子)进行计数。可以修改遍历算法的**结点访问操作**为**对特殊结点的判定和计数过程**，需要注意的是计数器的处理方式。可以有以下几种计数处理:

·用遍历函数的返回值传出求得的叶子结点的数目;

·为遍历函数增加一个引用参数，用来传出指定二叉树的叶子结点数目。

·引入全局的计数器，初始为0;

此处，遍历次序的选择对本算法没有太大影响

例2 统计二叉树中叶子结点的数目

【算法1 全局的计数器】

```
1.  // n为叶子结点的计数器
2.  int    n = 0;
3.  void  leaf(BiTree T) {    // 利用二叉树的先序遍历
4.          if (T != NULL) {
5.                  // 访问结点->叶子的判定和计数
6.                  if (T->lchild == NULL && T->rchild == NULL)
7.                          n++;
8.                  leaf(T->lchild);
9.                  leaf(T->rchild);
10.         }
11.}// 调用结束，即可由n获得二叉树T的叶子结点数目，需注意下次调用前须n=0;
```

例2 统计二叉树中叶子结点的数目

【算法2 以函数返回值返回】

```
1.  // 函数值为T的叶子结点数
2.  int    leaf(BiTree T) {    // 利用二叉树的中序遍历, n为局部变量
3.         n = 0;
4.         if (T != NULL) {
5.                 n = leaf(T->lchild);
6.                 // 访问结点->叶子结点的判定和计数
7.                 if (T->lchild == NULL && T->rchild == NULL)
8.                         n++;
9.                 n += leaf(T->rchild);
10.        }
11.        return n;
12.}
```

例2 统计二叉树中叶子结点的数目

【算法3 通过引用参数返回】

```
1.  // 引用参数n等同于全局变量，方法二
2.  // 把T所指向的二叉树中的叶子结点数累加到n
3.  // 注意：在调用leaf(T,n)之前要先执行“n = 0;”
4.  n = 0;
5.  void leaf(BiTree T, int& n) {     // 利用二叉树的后序遍历
6.          if (T != NULL) {
7.                  leaf(T->lchild, n);
8.                  leaf(T->rchild, n);
9.                  // 访问结点->叶子结点的判定和计数
10.                 if (T->lchild == NULL && T->rchild == NULL)n++;
11.         }
12.}
```

例3 释放二叉树的所有结点空间

【思路】

·二叉树为空时，不必释放；

·若T不为空，则先释放其左右子树的所有结点的空间，再释放根结点的空间——后序。

若在释放子树的空间前，先释放根结点的空间，则需要将子树的根结点的指针暂存到其他变量；否则，无法找到子树。

【算法】

```
1. void deleteBiTree(BiTree& T) { // 此处T应为引用参数
2.         if (T != NULL) {
3.                 deleteBiTree(T->lchild);
4.                 deleteBiTree(T->rchild);
5.                 // 访问结点->释放结点的空间
6.                 delete T;
7.                 T = NULL;
8.         }
9. }
```

例4 删除并释放二叉树中以元素值为x的结点作为根的各子树

【本例特征】
如何选择二叉树的先序、中序、后序遍历来解决问题，它们对问题求解有何影响？

【思路】
整个过程分为两个方面：
·遍历中查找元素值为x的结点
·查到该结点时，调用例3的算法释放子树空间。
需要考虑的问题是：
·如何将全部的结点找到并释放？
·外层查找采用的遍历次序对本算法有何影响？
从以下3个算法中可以看出，利用先序遍历是最合适的；中序和后序，存在一定的多余操作。

- **例4 删除并释放二叉树中以元素值为x的结点作为根的各子树**

【算法1】

```
1.  void  deleteXTree(BiTree& T, ElemType x) {// 基于先序的查找
2.      if (T != NULL) {
3.      // 访问结点->判断是否为指定结点->释放树空间
4.          if (T->data == x) deleteBiTree(T);
5.          else {// 此处else不能省略
6.                      deleteXTree(T->lchild, x);
7.                      deleteXTree(T->rchild, x);
8.                      }
9.          }
10.}
```

例4 删除并释放二叉树中以元素值为x的结点作为根的各子树

【算法2】

```
1.  void deleteXTree(BiTree& T, ElemType x) {       // 基于中序的查找
2.          if (T != NULL) {
3.                  deleteXTree(T->lchild, x);
4.                  // 若T->data== x，则此步骤多余
5.                  // 访问结点->判断是否为指定结点->释放树空间
6.                  if (T->data == x) deleteBiTree(T);
7.                  else deleteXTree(T->rchild, x);
8.          }
9.  }
```

- **例4 删除并释放二叉树中以元素值为x的结点作为根的各子树**

【算法3】

```
1.  void deleteXTree(BiTree& T, ElemType x) {// 基于后序的查找
2.          if (T != NULL) {
3.                  deleteXTree(T->lchild, x);      // 若T->data== x，则此步骤多余
4.                  deleteXTree(T->rchild, x);      // 若T->data== x，则此步骤多余
5.                  // 访问结点->判断是否为指定结点->释放树空间
6.                  if (T->data == x) deleteBiTree(T);
7.          }
8.  }
```

例5 求位于二叉树先序序列中第k个位置的结点的值

【本例特征】

多个返回结果如何处理?

【思路】

·待查找的结点的存在性：当二叉树为空树，或者k非法时，待查找的结点是不存在的

➔ 函数应返回待查找的结点是否**存在**的状态指示：TRUE-存在，FALSE-不存在

·当待查找的结点存在时，需进一步返回该结点的值

问题1：该算法需要返回多个值，如何处理?

答：一种做法是用返回值返回存在性，用变参返回值。

问题2：该算法可以基于二叉树的先序遍历的递归算法来构造，如何知道当前访问的结点是先序序列中的第几个结点?

答：引入计数器，对于该计数器可以采用全局变量来存储，也可以通过变参来处理。

例5 求位于二叉树先序序列中第k个位置的结点的值

```
1.  Status PreorderKnode(BiTree T, int k, ElemType& e, int& count) {
2.        // 输入：T为二叉链表示的二叉树，k为待查找的结点在先序序列中的位序\
3.        // 输出：返回值—TRUE：待查找的结点存在；FALSE：待查找的结点不存在
4.        // e—当待查找的结点存在时，该结点的值通过e带回
5.        // 中间变量：count—记录当前已经访问过的结点个数
6.    if (T == NULL) return FALSE;
7.    count++;   // 访问结点，对已访问的结点进行计数
8.    if (count == k) {//判断该结点是否是待查找的结点
9.       e = T->data;
10.      return TRUE;// 查到，则设置e，并返回TRUE
11.   }
12.   else if (count > k)  return FALSE; // 计数器count已经超出k(当k<0时)，则直接返回FALSE
13.   else {
14.      if (PreorderKnode(T->lchild, k, e, count) == FALSE) // 在左子树中查找
15.         return PreorderKnode(T->rchild, k, e, count);  // 若未找到，则在右子树中查找
16.      return TRUE;
17.   }
18. }
```

【调用示意】

```
int c=0;
...
if ( PreorderKnode(T, k, e, c)==TRUE ) ...
```

其他问题：

- **二叉树**
  - 二叉树的计数
  - 求二叉树的高度
  - 复制一棵二叉树P127 算法5.4
  - 判断两个二叉树是否相同
  - 交换一颗二叉树（的每个结点的左右子树）
  - ……

- 非递归遍历

二叉树-链式存储结构

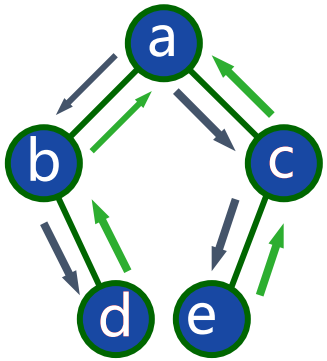
二叉树的链式存储结构

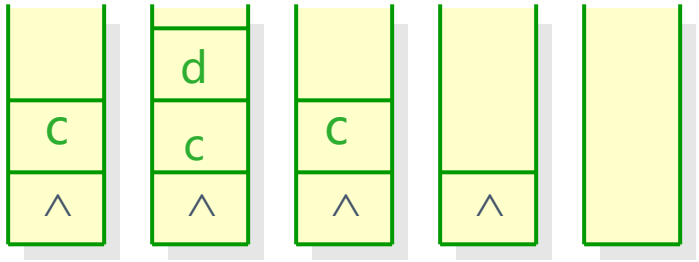
- 二叉链的类型定义(动态链表)

```
typedef struct BiTNode{
    ElemType data;
    struct BiTNode *lchild, *rchild; // 左右孩子指针
}BiTNode, *BiTree;
```

- 说明：若有n个结点，则共有2n个链域；其中n-1不为空，指向孩子；另外n+1个为空链域

利用栈的先序遍历的非递归算法





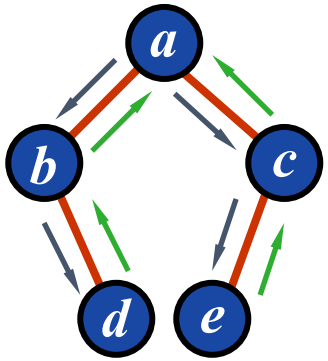
访问 a	访问 b	退栈 d	退栈 c	访问 e
进栈 c	进栈 d	访问 d	访问 c	左进 空
左进 b	左进 空	左进 空	左进 e	退栈 ^

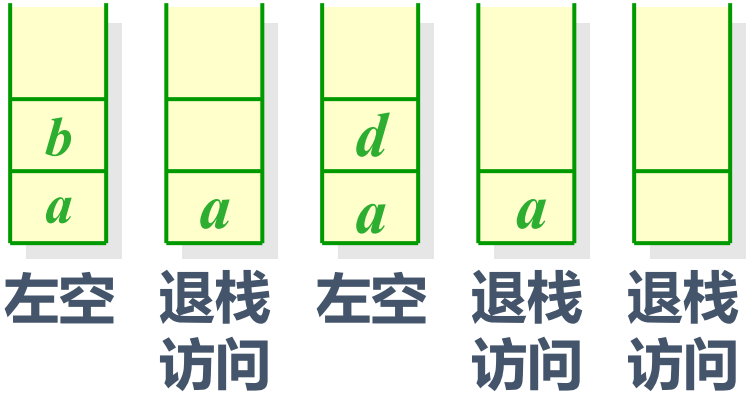
结束

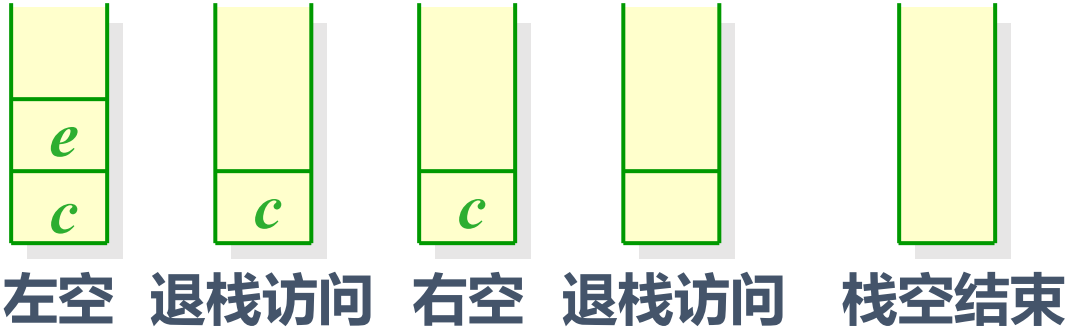
利用栈的先序遍历的非递归算法


```
1.  void PreOrder(BiTree T) {
2.      stack S;   InitStack(S);                    //递归工作栈
3.      BinTreeNode* p = T;  Push(S, NULL);
4.      while (p != NULL) {
5.          cout<<p->data;
6.          if (p->rchild != NULL)
7.              Push(S, p->rchild);
8.          if (p->lchild != NULL)
9.              p = p->lchild;                      //进左子树
10.         else Pop(S, p);                         //左子树空, 进右子树
11.     }
12.}
```

利用栈的中序遍历的非递归算法







利用栈的中序遍历的非递归算法


```
1.  void InOrder(BiTree T) {
2.      stack S;   InitStack(S);          //递归工作栈
3.      BiTree p = T;                     //初始化
4.      do {
5.          while (p != NULL) {           //子树非空找中序第一个
6.              Push(S, p);   p = p->lchild;
7.          }                             //边找边进栈
8.          if (!StackEmpty(S)) {         //栈非空
9.              Pop(S, p);                //子树中序第一个退栈
10.             cout<<p->data;                    //访问之
11.             p = p->rchild;            //向右子树走
12.         }
13.     } while (p != NULL || !StackEmpty(S));
14.}
```

利用栈的后序遍历的非递归算法

后序遍历时使用的栈的结点定义

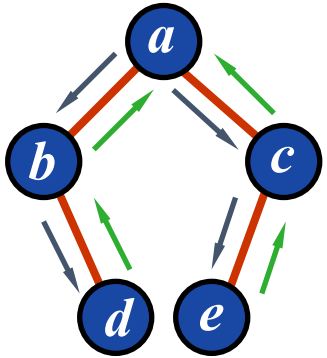
```
typedef  struct {
        BiTree ptr;                    //结点指针
        enum tag{ L, R };              //该结点退栈标记
} StackNode;
```

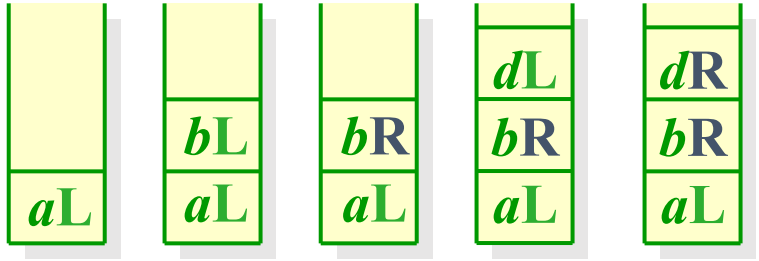
根结点的

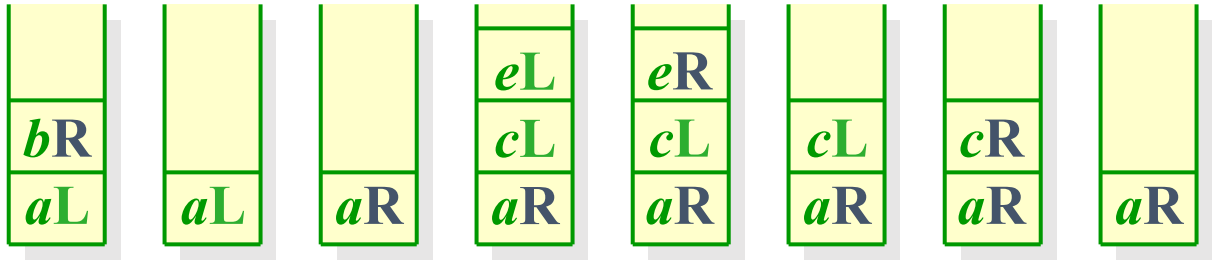
tag = L，表示从左子树退出, 访问右子树。

tag = R，表示从右子树退出, 访问根。

利用栈的后序遍历的非递归算法







利用栈的后序遍历的非递归算法


```
1.  void PostOrder(BiTree T) {
2.      stack S;   InitStack(S);   StackNode w;
3.      BiTree p = T;
4.      do {
5.          while (p != NULL) {          //向左子树走
6.              w.ptr = p;   w.tag = L;   Push(S, w);
7.              p = p->lchild;
8.          }
9.          int succ = 1;                //继续循环标记
```

利用栈的后序遍历的非递归算法

```
10. while (succ && !StackEmpty(S)) {
11.             Pop(S, w);  p = w.ptr;
12.             switch (w.tag) {       //判断栈顶tag标记
13.                     case L :  w.tag = R;   Push(S, w);
14.                                             succ = 0;
15.                                             p = p->rchild;   break;
16.                     case R :  cout<<p->data;
17.             }
18.         }
19.     } while ( !StackEmpty(S) );
20.}
```

- 层次遍历

层次遍历


```
1.  void LevelOrder(BiTree T) {                    //伪码
2.      Queue Q; InitQueue(Q); BiTree p = T;
3.      if (p) {
4.          EnQueue(Q, p);                         //根结点入队
5.          while (!IsEmpty(Q)) {
6.              DeQueue(Q, p); cout<<p->data;
7.              if (p->lchild)
8.                  EnQueue(Q, p->lchild);         // 左子入队
9.              if (p->rchild)
10.                 EnQueue(Q, p->rchild);         //右子入Q
11.         }
12.     }
13.}
```

作业预告

- 二叉树的创建及遍历