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

- **二叉树的递归遍历**
  - 先序、中序、后序遍历
- **基于递归的遍历算法的实现**
  - 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
- **非递归遍历**
  - 先序、中序、后序非递归遍历
  - 层次遍历

如何使用已有的栈和队列？V1一个.cpp

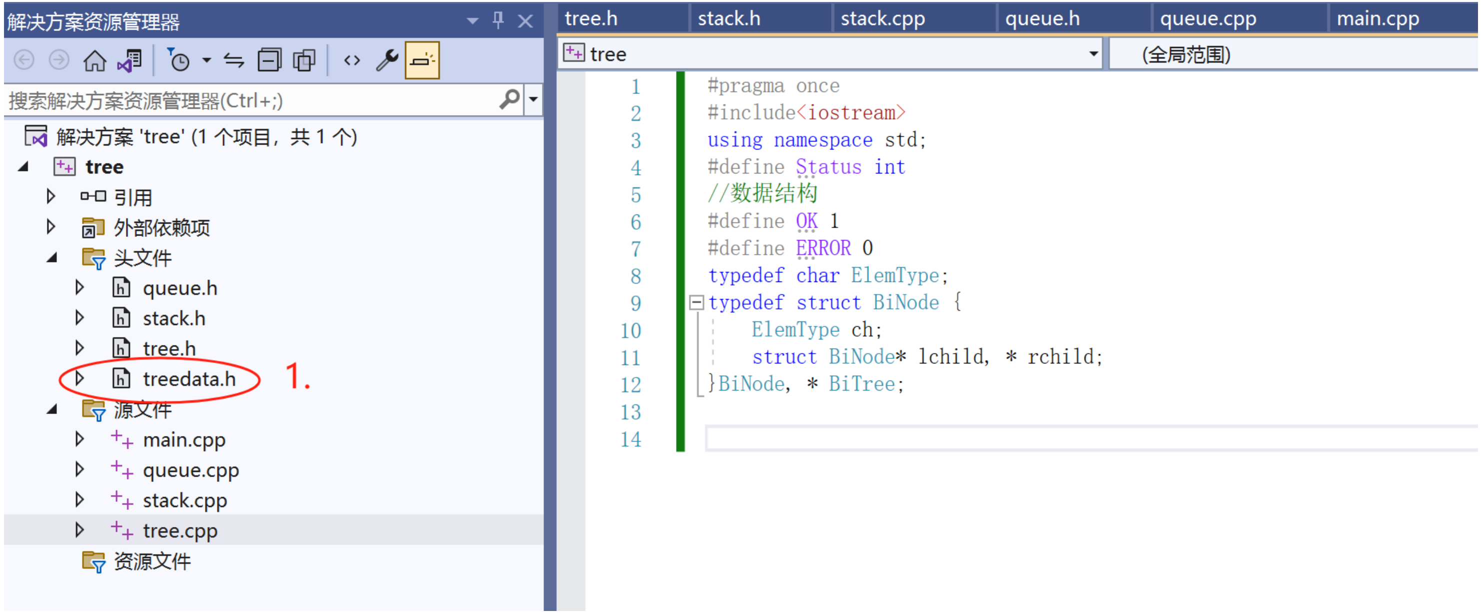
树.cpp

杂项文件 Node

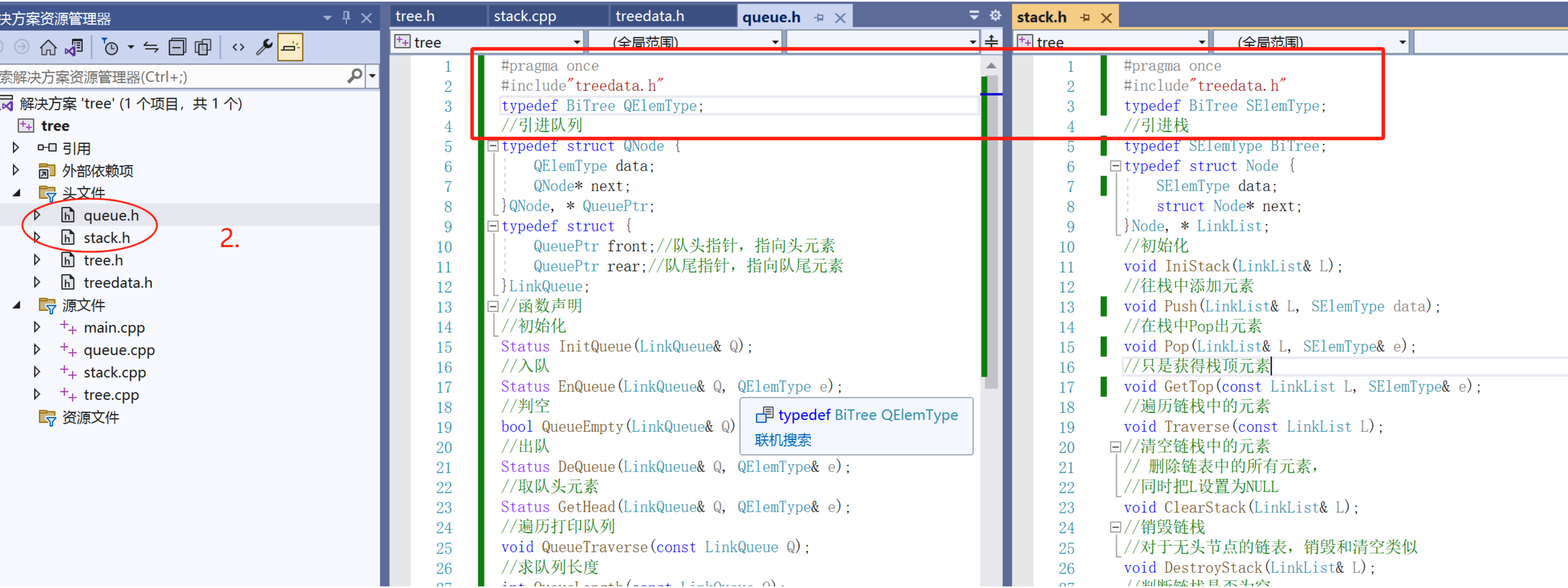
```
#include<iostream>
using namespace std;
#define Status int
//数据结构
#define OK 1
#define ERROR 0
typedef char ElemType;
typedef struct BiNode {
    ElemType ch;
    struct BiNode *lchild, *rchild;
}BiNode, *BiTree;
//引进栈
#define ElemType1 BiTree
#define QElemType BiTree
typedef struct Node {
    ElemType1 data;
    struct Node* next;
}Node, * LinkList;
//引进队列
typedef struct QNode {
    QElemType data;
    QNode* next;
}QNode, * QueuePtr;
typedef struct {
    QueuePtr front;//队头指针，指向头元素
    QueuePtr rear;//队尾指针，指向队尾元素
}LinkQueue;
//函数声明
//初始化
Status InitQueue(LinkQueue& Q);
//入队
Status EnQueue(LinkQueue& Q, QElemType e);
```

V2 .h+.cpp

北京语言大学 BEIJING LANGUAGE AND CULTURE UNIVERSITY

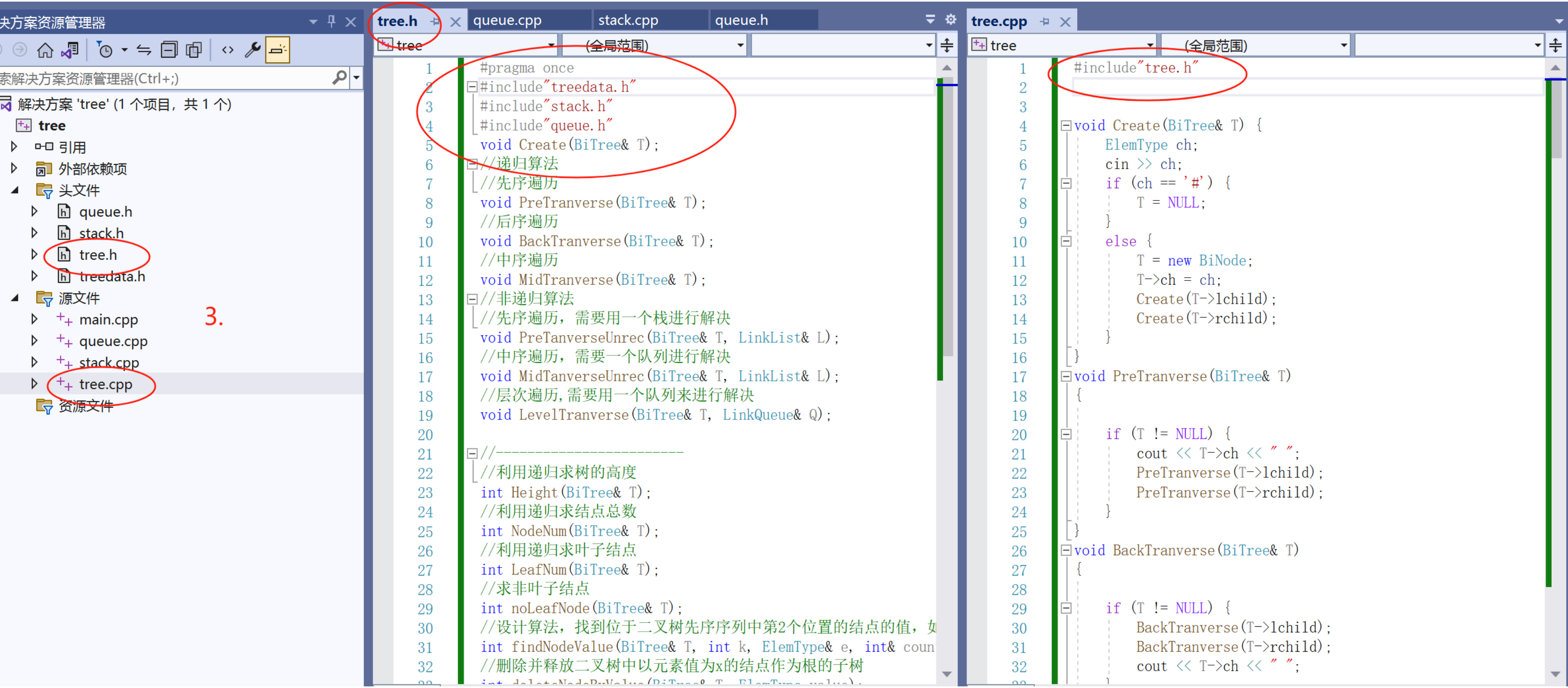


使用原有的栈和队列

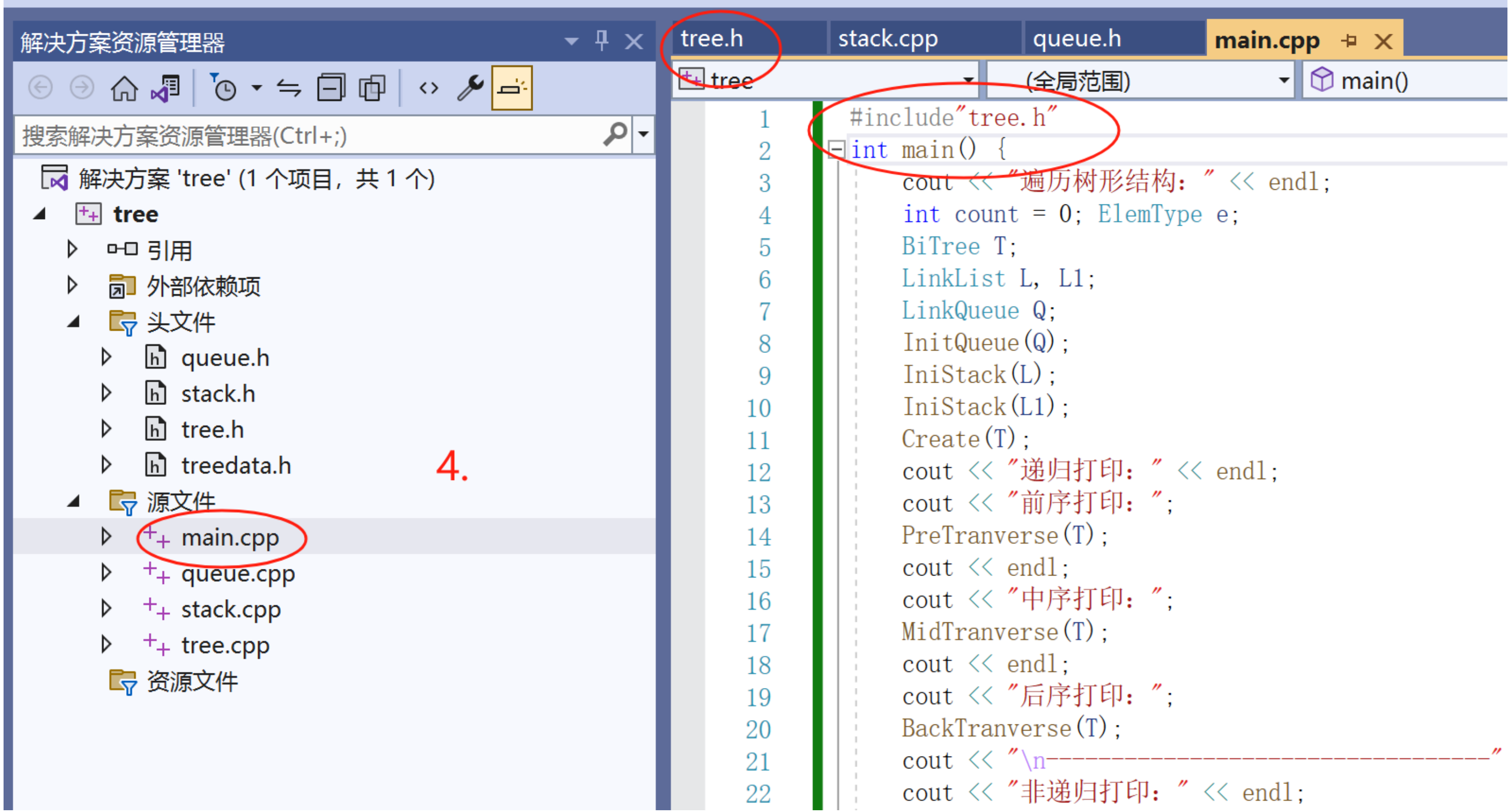


决方案资源管理器
索解决方案资源管理器(Ctrl+;)
解决方案 'tree' (1 个项目，共 1 个)
tree
引用
外部依赖项
头文件
queue.h
stack.h
tree.h
treedata.h
源文件
main.cpp
queue.cpp
stack.cpp
tree.cpp
资源文件
2.
tree.h
stack.cpp
treedata.h
queue.h
tree
(全局范围)
#pragma once
#include"treedata.h"
typedef BiTree QElemType;
//引进队列
typedef struct QNode {
QElemType data;
QNode* next;
}QNode, * QueuePtr;
typedef struct {
QueuePtr front;//队头指针，指向头元素
QueuePtr rear;//队尾指针，指向队尾元素
}LinkQueue;
//函数声明
//初始化
Status InitQueue(LinkQueue& Q);
//入队
Status EnQueue(LinkQueue& Q, QElemType e);
//判空
bool QueueEmpty(LinkQueue& Q)
//出队
Status DeQueue(LinkQueue& Q, QElemType& e);
//取队头元素
Status GetHead(LinkQueue& Q, QElemType& e);
//遍历打印队列
void QueueTraverse(const LinkQueue Q);
//求队列长度
typedef BiTree QElemType
联机搜索
stack.h
tree
(全局范围)
#pragma once
#include"treedata.h"
typedef BiTree SElemType;
//引进栈
typedef SElemType BiTree;
typedef struct Node {
SElemType data;
struct Node* next;
}Node, * LinkList;
//初始化
void IniStack(LinkList& L);
//往栈中添加元素
void Push(LinkList& L, SElemType data);
//在栈中Pop出元素
void Pop(LinkList& L, SElemType& e);
//只是获得栈顶元素
void GetTop(const LinkList L, SElemType& e);
//遍历链栈中的元素
void Traverse(const LinkList L);
//清空链栈中的元素
// 删除链表中的所有元素，
//同时把L设置为NULL
void ClearStack(LinkList& L);
//销毁链栈
//对于无头节点的链表，销毁和清空类似
void DestroyStack(LinkList& L);

使用原有的栈和队列


解决方案资源管理器
搜索解决方案资源管理器(Ctrl+;)
解决方案 'tree' (1 个项目，共 1 个)
tree
引用
外部依赖项
头文件
queue.h
stack.h
tree.h
treedata.h
源文件
main.cpp
queue.cpp
stack.cpp
tree.cpp
资源文件
3.
tree.h
queue.cpp
stack.cpp
queue.h
tree
(全局范围)
#pragma once
#include"treedata.h"
#include"stack.h"
#include"queue.h"
void Create(BiTree& T);
//递归算法
//先序遍历
void PreTranverse(BiTree& T);
//后序遍历
void BackTranverse(BiTree& T);
//中序遍历
void MidTranverse(BiTree& T);
//非递归算法
//先序遍历，需要用一个栈进行解决
void PreTanverseUnrec(BiTree& T, LinkList& L);
//中序遍历，需要一个队列进行解决
void MidTanverseUnrec(BiTree& T, LinkList& L);
//层次遍历,需要用一个队列来进行解决
void LevelTranverse(BiTree& T, LinkQueue& Q);
//-----------------------
//利用递归求树的高度
int Height(BiTree& T);
//利用递归求结点总数
int NodeNum(BiTree& T);
//利用递归求叶子结点
int LeafNum(BiTree& T);
//求非叶子结点
int noLeafNode(BiTree& T);
//设计算法，找到位于二叉树先序序列中第2个位置的结点的值，如
int findNodeValue(BiTree& T, int k, ElemType& e, int& coun
//删除并释放二叉树中以元素值为x的结点作为根的子树
tree.cpp
tree
(全局范围)
#include"tree.h"
void Create(BiTree& T) {
ElemType ch;
cin >> ch;
if (ch == '#') {
T = NULL;
}
else {
T = new BiNode;
T->ch = ch;
Create(T->lchild);
Create(T->rchild);
}
}
void PreTranverse(BiTree& T)
{
if (T != NULL) {
cout << T->ch << " ";
PreTranverse(T->lchild);
PreTranverse(T->rchild);
}
}
void BackTranverse(BiTree& T)
{
if (T != NULL) {
BackTranverse(T->lchild);
BackTranverse(T->rchild);
cout << T->ch << " ";

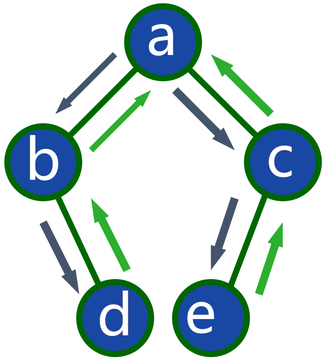
Main函数

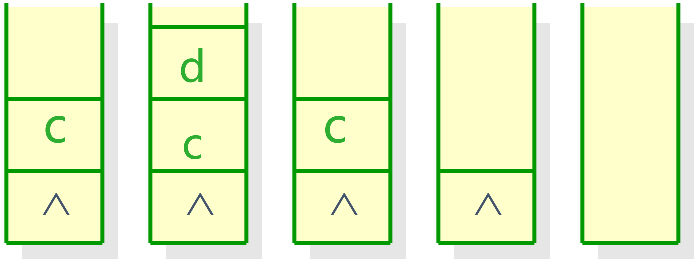

解决方案资源管理器
搜索解决方案资源管理器(Ctrl+;)
解决方案 'tree' (1 个项目，共 1 个)
tree
引用
外部依赖项
头文件
queue.h
stack.h
tree.h
treedata.h
源文件
main.cpp
queue.cpp
stack.cpp
tree.cpp
资源文件
4.
tree.h
stack.cpp
queue.h
main.cpp
tree
(全局范围)
main()
#include"tree.h"
int main() {
cout << "遍历树形结构：" << endl;
int count = 0; ElemType e;
BiTree T;
LinkList L, L1;
LinkQueue Q;
InitQueue(Q);
IniStack(L);
IniStack(L1);
Create(T);
cout << "递归打印：" << endl;
cout << "前序打印：";
PreTranverse(T);
cout << endl;
cout << "中序打印：";
MidTranverse(T);
cout << endl;
cout << "后序打印：";
BackTranverse(T);
cout << "\n--------------------------------"
cout << "非递归打印：" << endl;

回顾

- **遍历的非递归算法**
  - 先序的非递归遍历

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

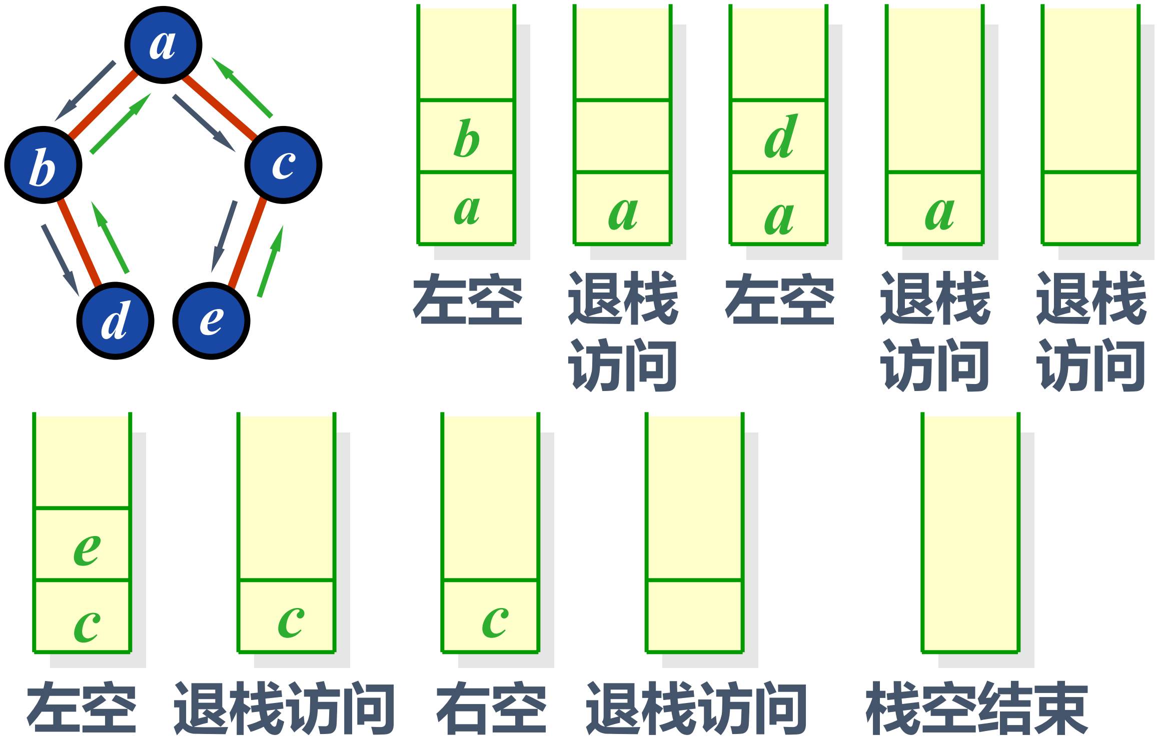
利用栈的先序遍历的非递归算法

```
void PreOrder(BiTree T) {
    stack S;   InitStack(S);          //递归工作栈
    BiTree p = T;  Push (S, NULL);
    while (p != NULL) {
        visit(p->data);        //cout...
        if (p->rchild != NULL)
            Push(S, p->rchild);
        if (p->lchild != NULL)
            p = p->lchild;            //进左子树
        else Pop(S, p);           //左子树空, 进右子树
    }
}
```

回顾

- **遍历的非递归算法**
  - 中序的非递归遍历

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

```
void InOrder(BiTree T) {
    stack S;   InitStack(S);                    //递归工作栈
    BiTree p = T;                               //初始化
    do {
        while (p != NULL)                       //子树非空找中序第一个
            { Push(S, p);   p = p->lchild; }    //边找边进栈
        if (!StackEmpty(S)) {                   //栈非空
            Pop(S, p);                          //子树中序第一个退栈
            visit(p->data);                     //访问之
            p = p->rchild;                      //向右子树走
        }
    } while ( p != NULL || !StackEmpty(S) );
}
```

回顾

- **遍历的非递归算法**
  - 后序的非递归遍历

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

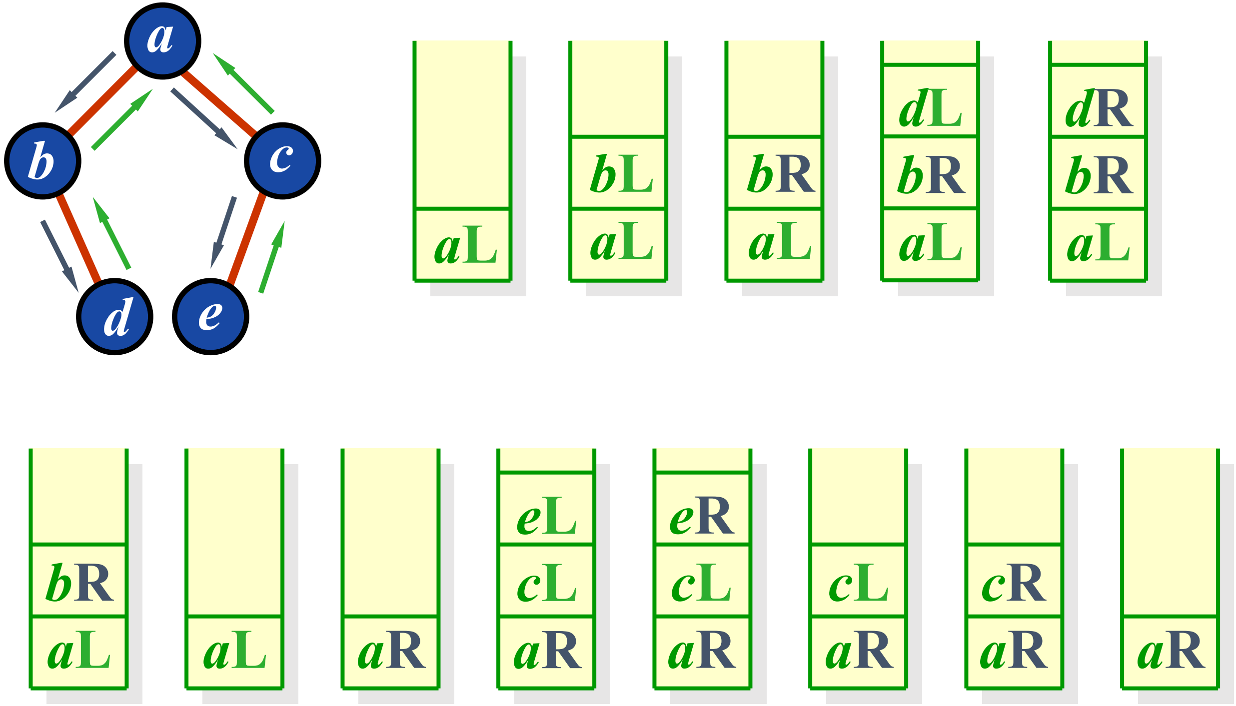
```
typedef  struct {
    BiTree ptr;      //结点指针
    enum tag{ L, R };      //该结点退栈标记
} StackNode;
```

ptr	tag{L,R}

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

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

利用栈的后序遍历的非递归算法

```
    while (succ && !StackEmpty(S)) {
            Pop(S, w);  p = w.ptr;
            switch (w.tag) {        //判断栈顶tag标记
                case L :  w.tag = R;   Push(S, w);
                        succ = 0;
                  p = p->rchild;   break;
                case R :  visit(p->data);
            }
        }
    } while ( !StackEmpty(S) );
}
```

回顾

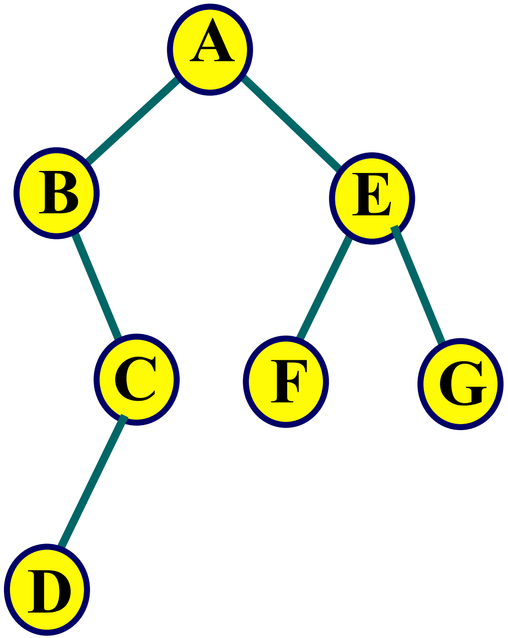
- 层次遍历算法

层次遍历

```
void LevelOrder( BiTree T){//伪码
    Queue Q;InitQueue(Q) ; BiTree p=T ;
    if (p){
        EnQueue(Q,p);              //根结点入队
        while (!IsEmpty(Q)){
            DeQueue(Q,p); visit( p->data );
            if (p->lchild)
                EnQueue(Q,p->lchild);     // 左子入队
            if (p->rchild)
                EnQueue(Q,p->rchild);    //右子入Q
        }
    }
}
```

回顾

- **如何创建一棵二叉树?**
  - 方法一：补虚结点的先序序列（也叫增广的先序序列，上次课五个算法中的第一个）
  - ——推广：先序=》中序、后序



先序：ABCDEFG
中序：BDCAFEG
后序：DCBFGEA

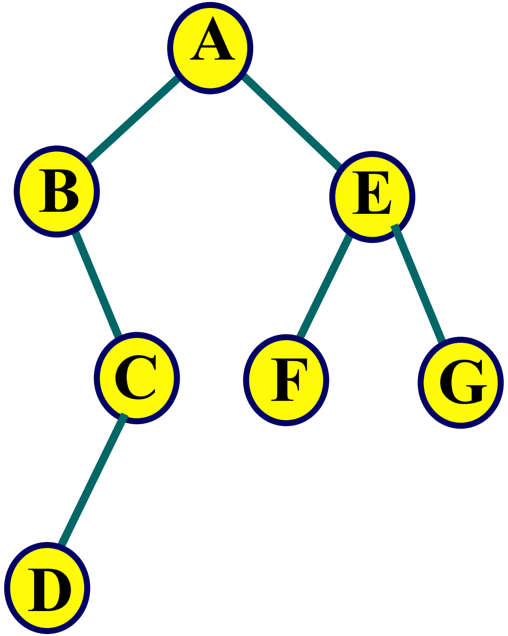
先序+中序、中序+后序可以唯一确定一棵二叉树；
先序+后序不能唯一确定一棵二叉树。

本节内容

- 创建二叉树之一：补虚结点的先序序列
- 创建二叉树之二：双序列递归生成二叉树

双序列递归生成二叉树

给定



则其遍历序列：

先序：ABCDEFG
中序：BDCAFEG
后序：DCBFGEA

先序是先访问根，再访问左子树，最后访问右子树（知道当前根，不知其子树结构、大小）

中序是先访问左子树，再访问根，最后访问右子树（不知根位置信息）

双序列递归生成二叉树

问题---给定先序中序遍历序列，递归确定二叉树
先序：ABCDEFG　　中序：BDCAFEG

CreatPreIn的递归框架：
1、先序当前字符为根，生成节点；(原子问题)
2、定位左子树，CreatPreIn；（子问题1）----在先序中
3、定位右子树，CreatPreIn；（子问题2）

```
void CreatPreIn(BiTree &T,char *pre,char *in,int n){
    int k=0;    char *p=in;            //n为(子)树节点个数
    if(pre&&n)    {
        T = new BiTNode;
        T->data = pre[0]; T->lchild = NULL;
        T->rchild = NULL;
        while(*(p++)!=pre[0])k++;//得到左子树节点个数
        CreatPreIn(T->lchild, pre+1,in,k);
        CreatPreIn(T->rchild,pre+k+1,in+k+1, n-k-1);
    }
}
```

先序：ABCDEFG　　中序：BDCAFEG

双序列递归生成二叉树

其中/后序遍历序列： 中序：BDCAFEG后序：DCBFGEA

后序:知道当前根，不知其子树结构、大小

中序:不知根位置信息

PostIn的递归框架：

1、后续当前字符为根，生成；(原子问题)

2、定位左子树，PostIn；（子问题1）---在后序中

3、定位右子树，PostIn；（子问题2）

```
void CreatInPost(BiTree& T,  char *in, char *post,int n){
    int k=0;    char *p=in;
    if(post&&n)    {
        T = new BiTNode;
        T->data = post[n-1];        T->lchild = NULL;
        T->rchild = NULL;
        while(*(p++)!=post[n-1])k++;//得到左子树长度
        CreatInPost(T->lchild, in, post, k);
        CreatInPost(T->rchild, in+k+1, post+k, n-k-1);
    }
}
```

中序：BDCAFEG　　后序：DCBFGEA

双序列递归生成二叉树

前/后序遍历序列： 先序：ABCDEFG　　后序：DCBFGEA

先序是先访问根，再访问左子树，最后访问右子树（知道当前根，不知其子树结构、大小）

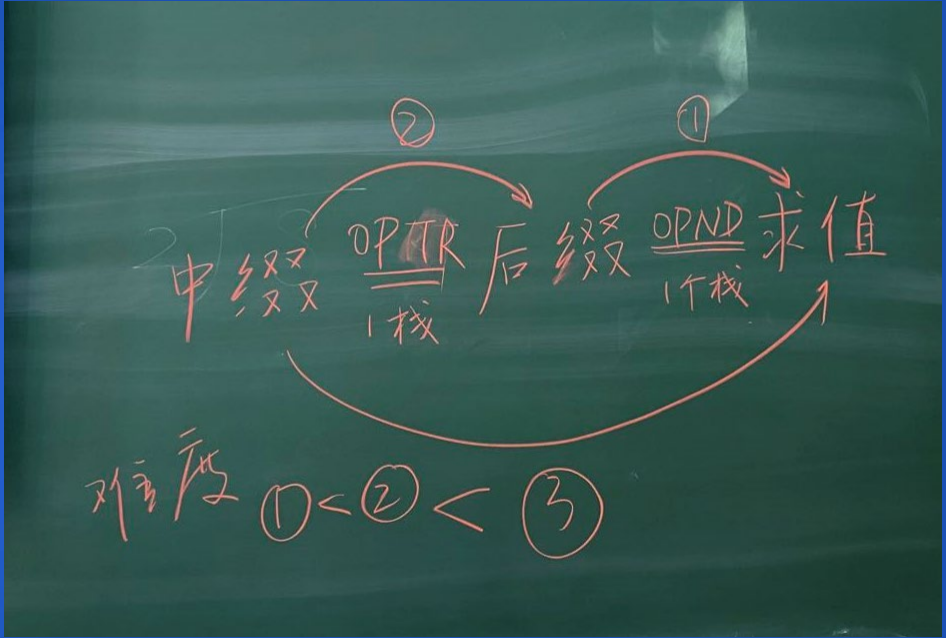
后序是先访问左子树，再访问右子树，最后根（知道当前根，不知其子树结构、大小）

结论：仅知道先/后序遍历序列（不包含镜像），无法唯一确定二叉树

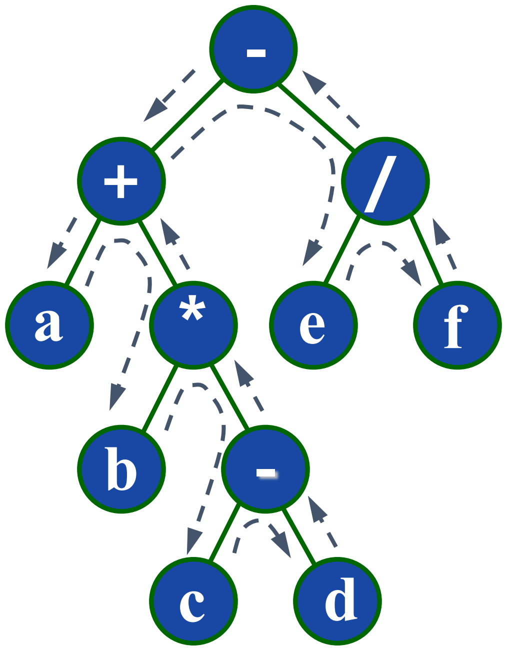
本节内容

- 创建二叉树之一：补虚结点的先序序列（增广序列）
- 创建二叉树之二：双序列递归生成二叉树
- 创建二叉树之三：表达式树

回顾



表达式树



1.先序序列： -+a*b-cd/ef;
2.中序序列： a+b*c-d-e/f a+b*(c-d)-e/f
3.后序序列： abcd-*+ef/-
4.层次序列： -+/a*efb-cd

表达式树的创建---【算法步骤】P143 算法5.12（以中缀表达式求值为例）

① 初始化OPTR栈和EXPT栈，将表达式起始符“#”压入OPTR栈。

② 扫描表达式，读入第一个字符ch，如果表达式没有扫描完毕至“#”或OPTR的栈顶元素不为“#”时，则循环执行以下操作：

- 若ch不是运算符，则以ch为根创建一棵只有根结点的二叉树，且将该树根结点压入EXPT栈，读入下一字符ch；
- 若ch是运算符，则根据OPTR的栈顶元素和ch的优先级比较结果，做不同的处理：
  - 若是小于，则ch压入OPTR栈，读入下一字符ch；
  - 若是大于，则弹出OPTR栈顶的运算符，从EXPT栈弹出两个表达式子树的根结点，以该运算符为根结点，以EXPT栈中弹出的第二个子树作为左子树，以EXPT栈中弹出的第一个子树作为右子树，创建一棵新二叉树，并将该树根结点压入EXPT栈；
  - 若是等于，则OPTR的栈顶元素是“(”且ch是“)”，这时弹出OPTR栈顶的“(”，相当于括号匹配成功，然后读入下一字符ch。

表达式树的求值---【算法步骤】P144 算法5.13

① 设变量lvalue和rvalue分别用以记录表达式树中左子树和右子树的值，初始均为0。

② 如果当前结点为叶子（结点为操作数），则返回该结点的数值，否则（结点为运算符）执行以下操作：

- 递归计算左子树的值记为lvalue；
- 递归计算右子树的值记为rvalue；
- 根据当前结点运算符的类型，将lvalue和rvalue进行相应运算并返回。

本节内容

- 线索化

二叉树的链式存储结构

- 二叉链的类型定义(动态链表)

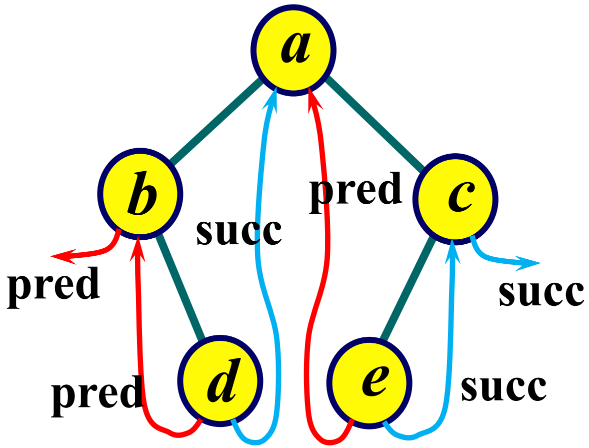
```
typedef struct BiTNode{
    ElemType data;
    struct BiTNode *lchild, *rchild; // 左右孩子指针
}BiTNode, *BiTree;
```

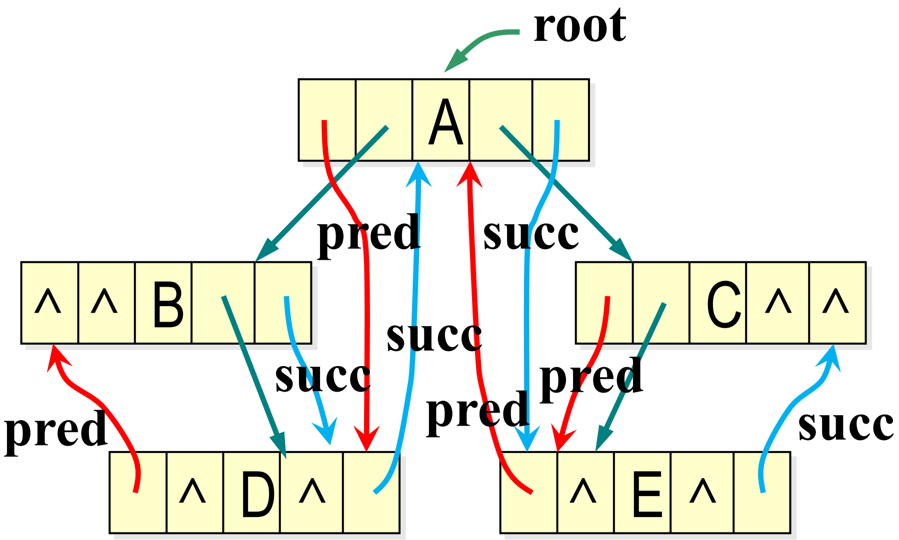
- 说明：若有n个结点，则共有2n个链域；其中n-1不为空，指向孩子；另外　　个为空链域

线索（Thread）——v1：增加前驱指针和后继指针

pred	lchild	data	rchild	succ

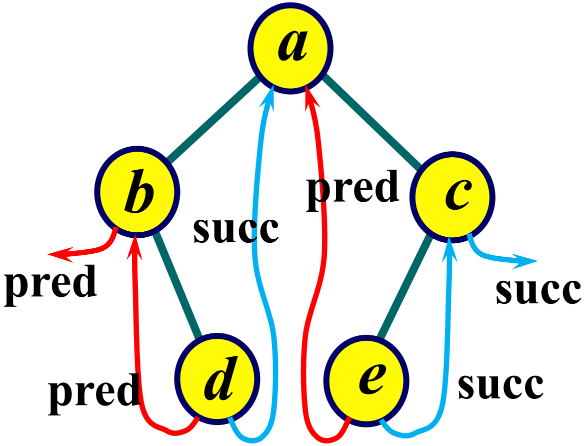
➢增加前驱Pred指针和后继Succ指针的二叉树

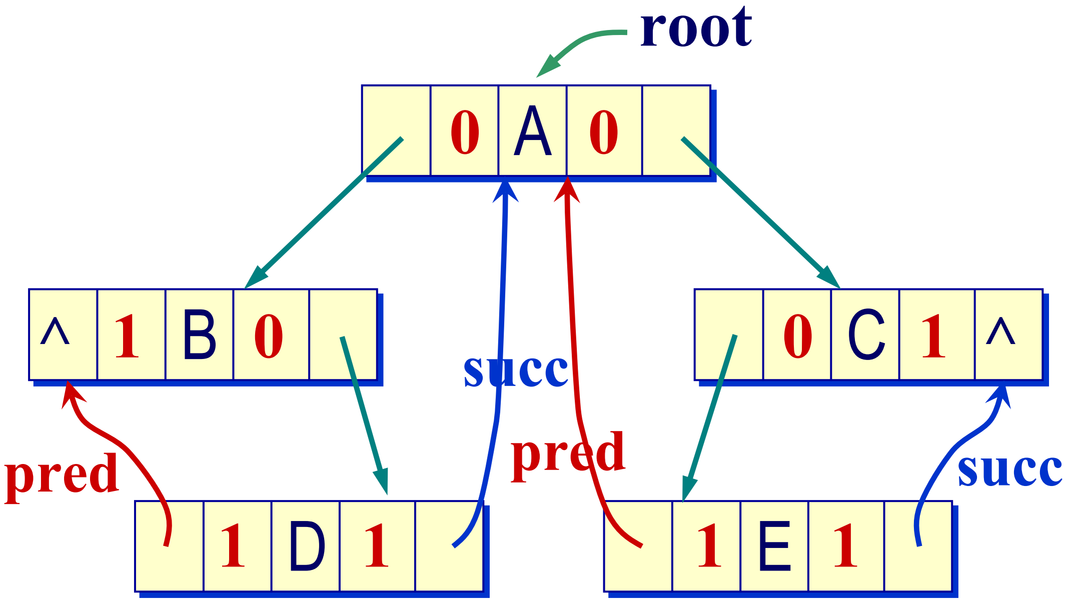




线索（Thread）

leftChild	ltag	data	rtag	rightChild

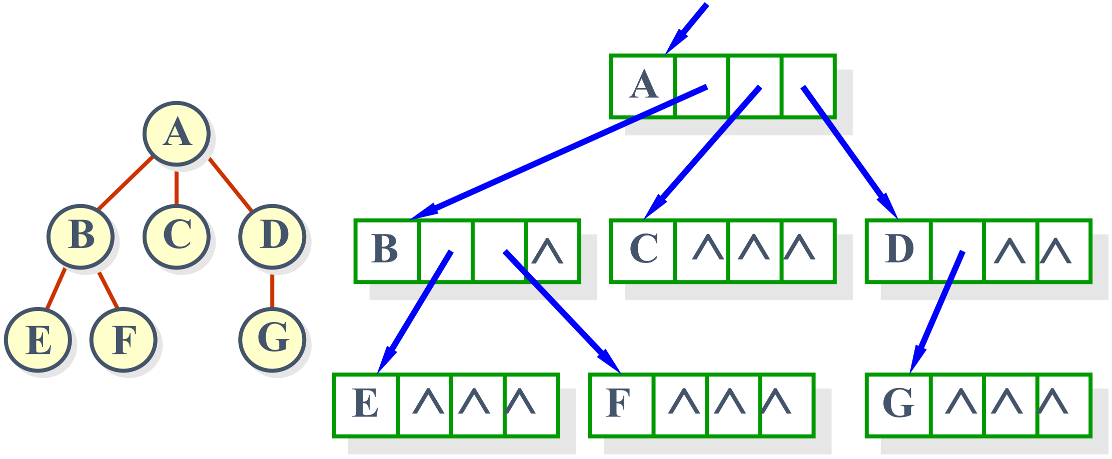




本节内容

- 树、森林的存储

- 树的存储表示之**孩子指针表示**



等数量的链域

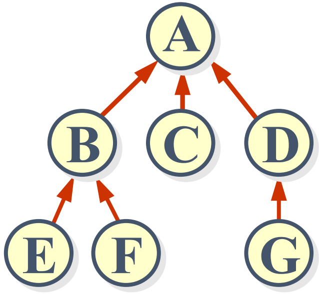
空链域$2n+1$个

data	child_1	child_2	child_3	■ ■ ■ ■	child_d

问题：可能产生很多空闲指针，造成存储浪费。

一棵普通的树如何存储……

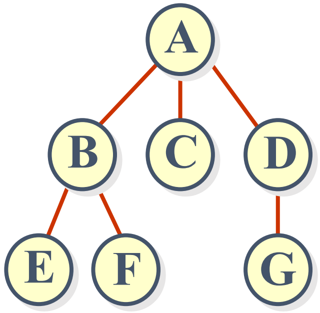
- 树的存储表示之**双亲表示**

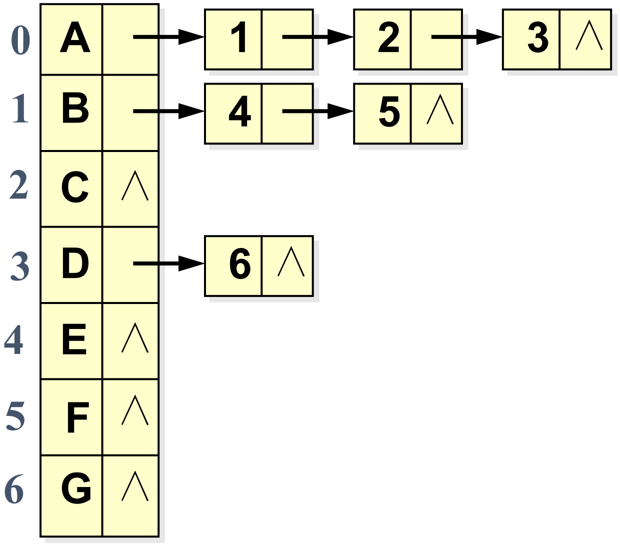


	0	1	2	3	4	5	6
data	A	B	C	D	E	F	G
parent	-1	0	0	0	1	1	3

并查集应用也用这中表达方式

- 树的存储表示之**链表表示**



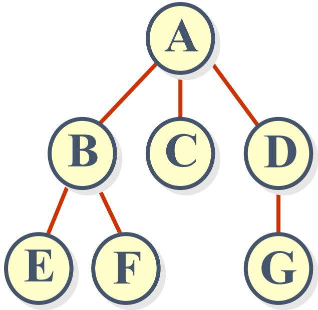


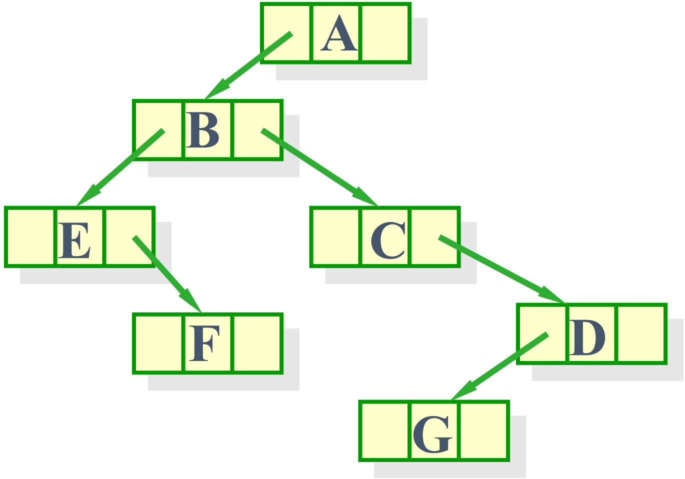
图的邻接矩阵也用这样的存储结构

一棵普通的树如何存储……

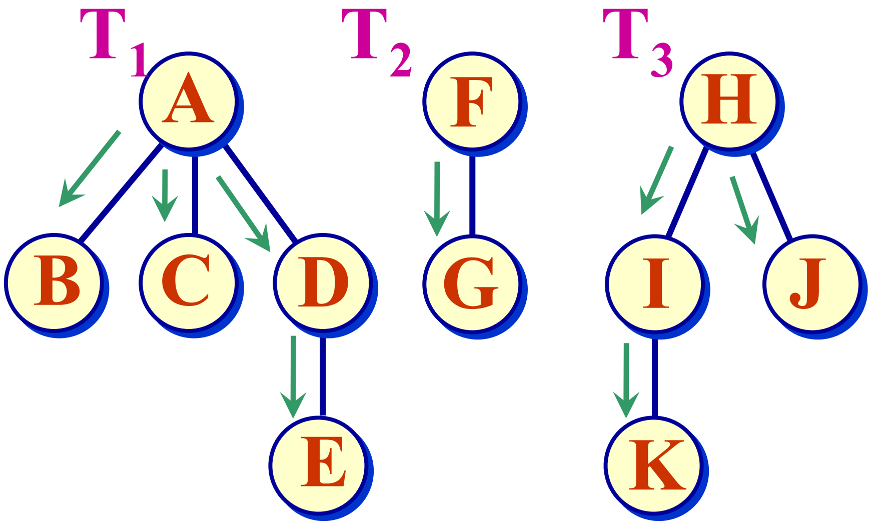
➢ 树的存储表示之**左孩子 - 右兄弟表示**

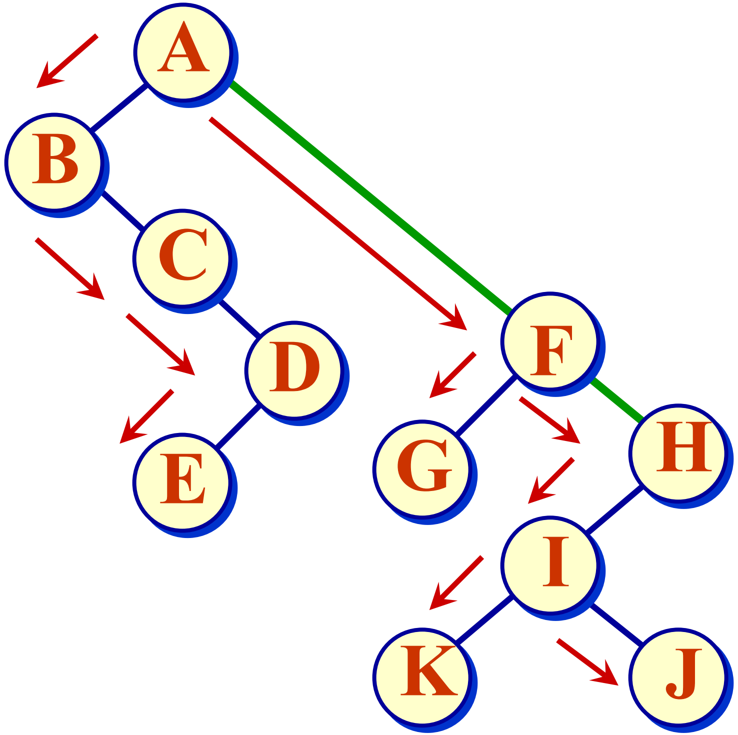
data	firstChild	nextSibling





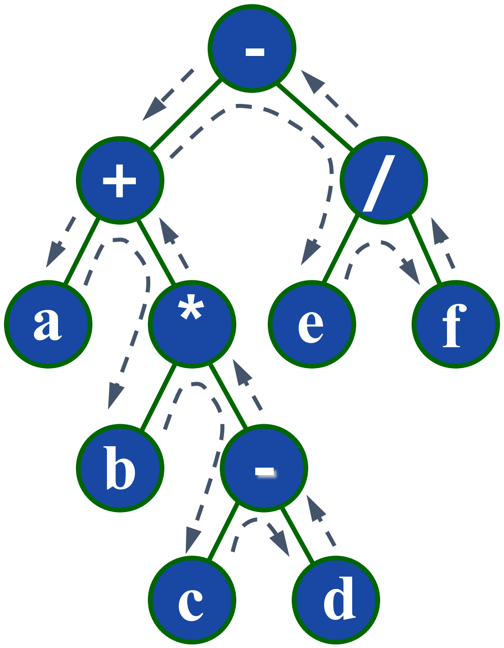
森林

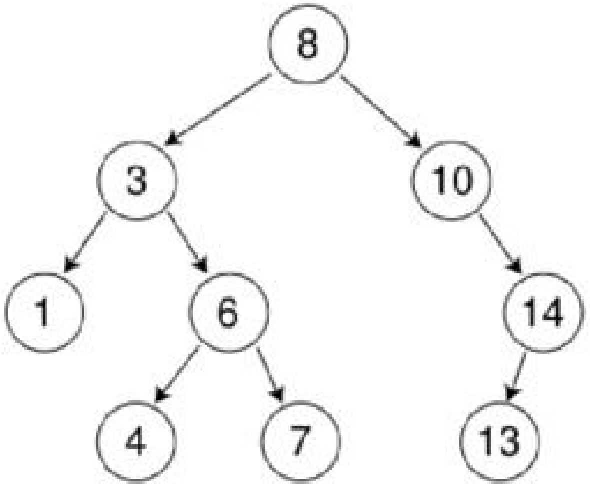




本节内容

- **二叉树的应用**
  - 二叉排序树
  - Huffman树
  - 平衡二叉树（*）
  - 堆（排序）
  - 并查集（*）


-
+
/
a
*
e
f
b
-
c
d


8
3
10
1
6
14
4
7
13

二叉排序树

- 概念、存储
- 操作：查找、插入、创建、删除
- 性能分析

二叉排序树 (Binary Search Tree)

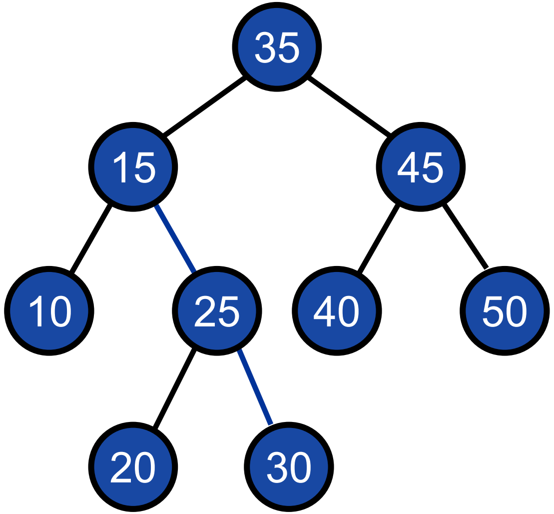
北京语言大学 BEIJING LANGUAGE AND CULTURE UNIVERSITY

定义

- 二叉排序树或者是一棵空树，或者是具有下列性质的二叉树：
  - 每个结点都有一个作为查找依据的关键字(key)，所有结点的关键字互不相同。
  - 左子树（如果非空）上所有结点的关键字都小于根结点的关键字。
  - 右子树（如果非空）上所有结点的关键字都大于根结点的关键字。
  - 左子树和右子树也是二叉排序树。

二叉排序树

- 结点左子树上所有关键字小于结点关键字；
- 结点右子树上所有关键字大于结点关键字；
- 如果对一棵二叉排序树进行中序遍历，可以按从小到大的顺序将各结点关键字排列起来。



注意，国外教材统称为二叉搜索树。

二叉排序树的结构定义

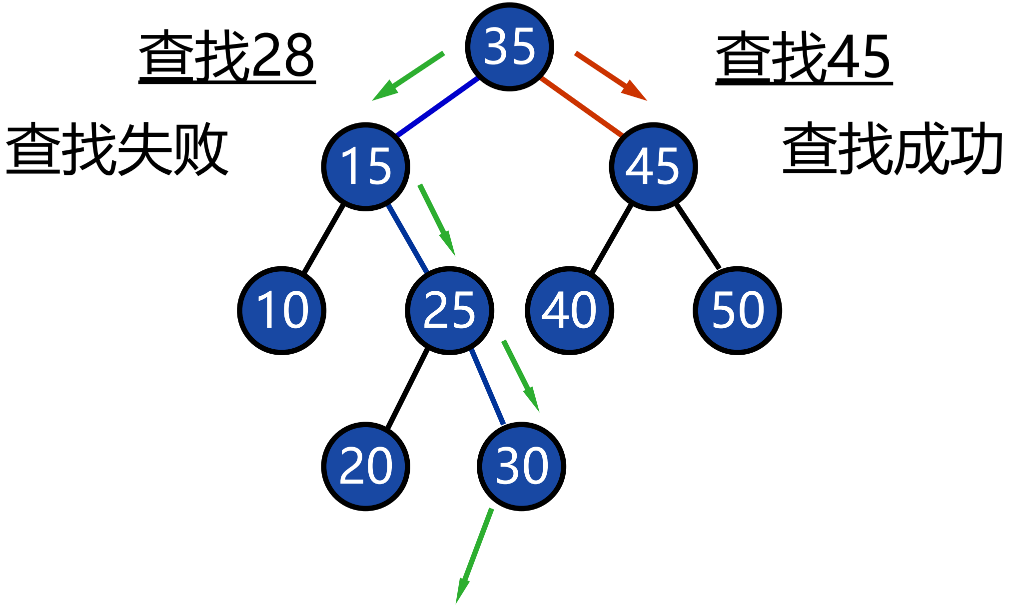
```
typedef char ElemType;          //树结点数据类型
typedef struct node {           //二叉排序树结点
    ElemType data;
    struct node *lchild, *rchild;
} BstNode, *BST;                //二叉排序树定义
```

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

- 二叉排序树是二叉树的特殊情形，它继承了二叉树的结构，增加了自己的特性，对数据的存放增加了约束。

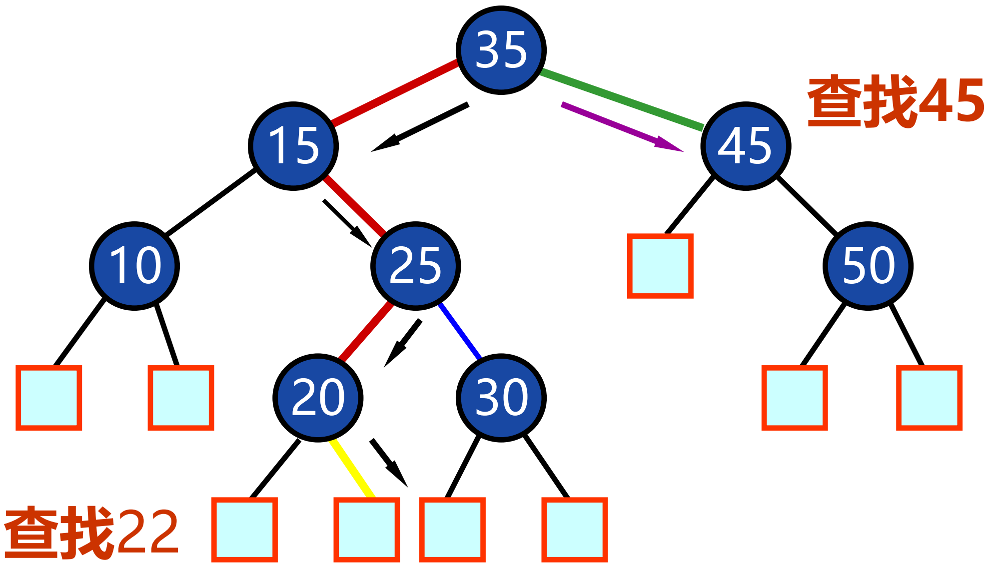
二叉排序树上的查找

- 在二叉排序树上进行查找，是一个从根结点开始，沿某一个分支逐层向下进行比较判等的过程。它可以是一个递归的过程。



- 假设想要在二叉排序树中查找关键字为x的元素，查找过程从根结点开始。
- 如果根指针为NULL，则查找不成功；否则用给定值 x 与根结点的关键字进行比较：
  - 如果给定值等于根结点的关键字值，则查找成功。
  - 如果给定值小于根结点的关键字值，则继续递归查找根结点的左子树；
  - 否则。递归查找根结点的右子树。
- 查找成功时检测指针停留在树中某个结点。

- 可用判定树描述查找过程。内结点是树中原有结点，外结点是失败结点，代表树中没有的数据。
- 查找不成功时检测指针停留在某个失败结点。



二叉排序树的查找算法(递归)

- 参见详见P199，算法7.4！

```
BiTree SearchBST(BiTree T, ElemType x)
void SearchBST(BiTree T, ElemType x, BiTree &p, BiTree &pr, );
```

观察递归有什么特点?

二叉排序树的查找算法(递归)

```
BSTree SearchBST(BSTree T, ElemType key) {
//课本p199算法7.4 数据结构稍微和本节使用不一样。
    if ((!T) || key == T->data.key) return T;
    else if (key < T->data.key)  return SearchBST(T->lchild, key);    //在左子树中继续查找
    else return SearchBST(T->rchild, key);                    //在右子树中继续查找
} //注意：查找成功返回什么？查找不成功返回什么？
```

二叉排序树的查找算法(递归)


```
void Find(BST t, ElemType x, BST& p, BST &pr) {
    //在二叉排序树 t 中查找关键字等于 x 的结点,
    //成功时 p 返回找到结点地址, pr 是其双亲结点.
    //不成功时 p 为空, pr 返回最后走到的结点地址.
    if (t == NULL) { p = NULL; }
    else if (t->data == x) p = t;
    else if (t->data > x) {
        pr = t;
        Find(t->lchild, x, p, pr);
    }
    else {
        pr = t;
        Find(t->rchild, x, p, pr);
    }
}
```

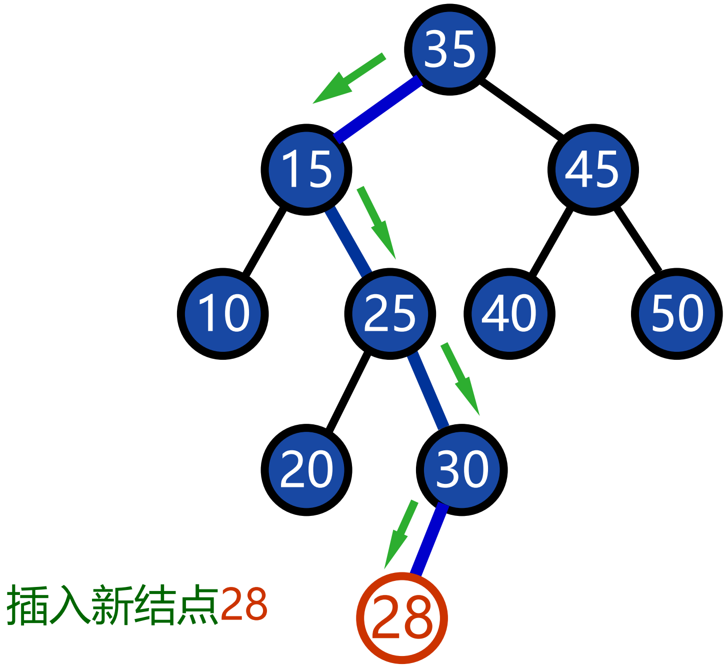
二叉排序树的查找算法

```
void Find(BST t, ElemType x, BST& p, BST& pr) {
    //在二叉排序树 t 中查找关键字等于 x 的结点，
    //成功时 p 返回找到结点地址, pr 是其双亲结点.
    //不成功时 p 为空, pr 返回最后走到的结点地址.
    if (t != NULL) {
        p = t;  pr = NULL;          //从根查找
        while (p != NULL && p->data != x) {
            pr = p;
            if (p->data < x) p = p->rchild;
            else p = p->lchild;
        }
    }
}
```

查找的关键字比较次数最多不超过树的高度。

二叉排序树的插入

- 每次结点的插入，都要从根结点出发查找插入位置，然后把新结点作为叶结点插入。
- 为了向二叉排序树中插入一个新元素，必须先检查这个元素是否在树中已经存在。
- 为此，在插入之前先使用查找算法在树中检查要插入元素有还是没有。
  - 查找成功：树中已有这个元素,不再插入。
  - 查找不成功：树中原来没有关键字等于给定值的结点，把新元素加到查找操作停止的地方。



二叉排序树的插入（非递归）

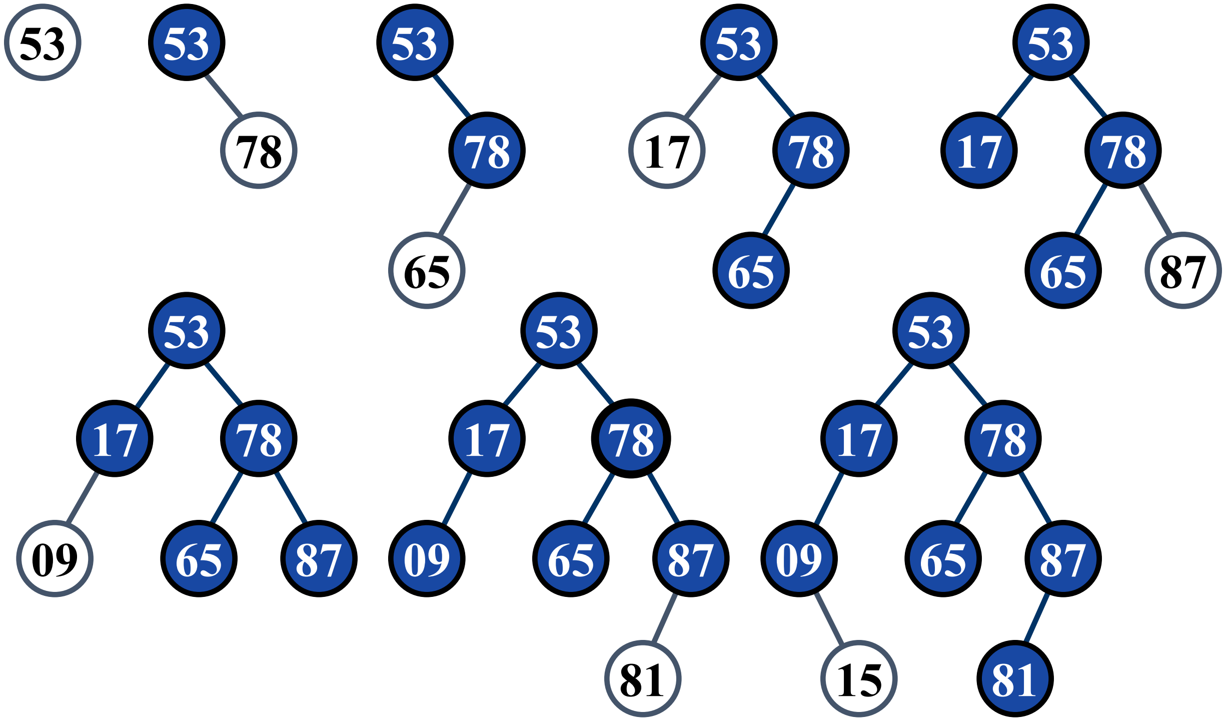

```
void Insert(BST& t, ElemType x) {
    //将新元素 x 插到以 *t 为根的二叉排序树中
    BstNode* pt, * prt=NULL, * q;
    Find(t, x, pt, prt);                    //查找结点插入位置
    if (pt == NULL) {                       //查找失败时可插入
        q = new BstNode;  q->data = x;      //创建结点
        q->lchild = q->rchild = NULL;
        if (prt == NULL) t = q;             //空树
        else if (x < prt->data) prt->lchild = q;
        else prt->rchild = q;
    }
}
```

二叉排序树的插入（递归，P201算法7.5）

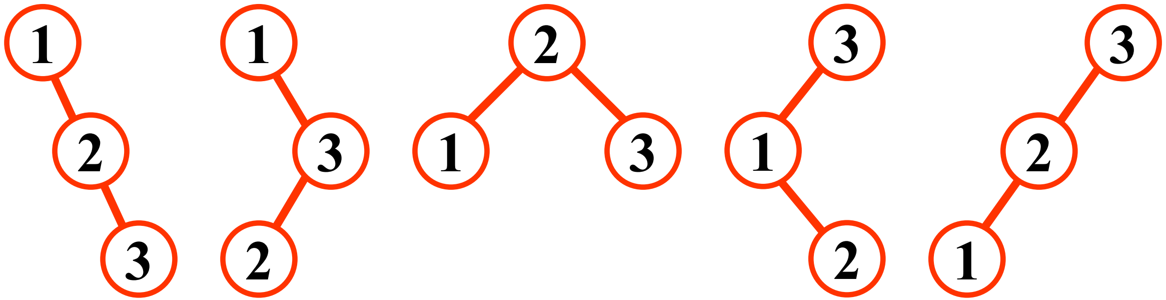
```
void InsertBST(BSTree& T, ElemType e) {
//当二叉排序树 T中不存在关键字等于e.key的数据元素时， 则插入该元素
    if (!T) {                                //找到插入位置， 递归结束
        S = new BSTNode;                     //生成新结点*S
        S->data = e;                         //新结点*S的数据域置为e
        S->lchild = S->rchild = NULL;        //新结点*S作为叶子结点
        T = S;                               //把新结点*S链接到已找到的插入位置
    }
    else if (e.key < T->data)
        InsertBST(T->lchild, e);             //将*S插入左子树
    else if (e.key > T->data)
        InsertBST(T->rchild, e);             //将*S插入右子树
}
```

输入数据，建立二叉排序树的过程

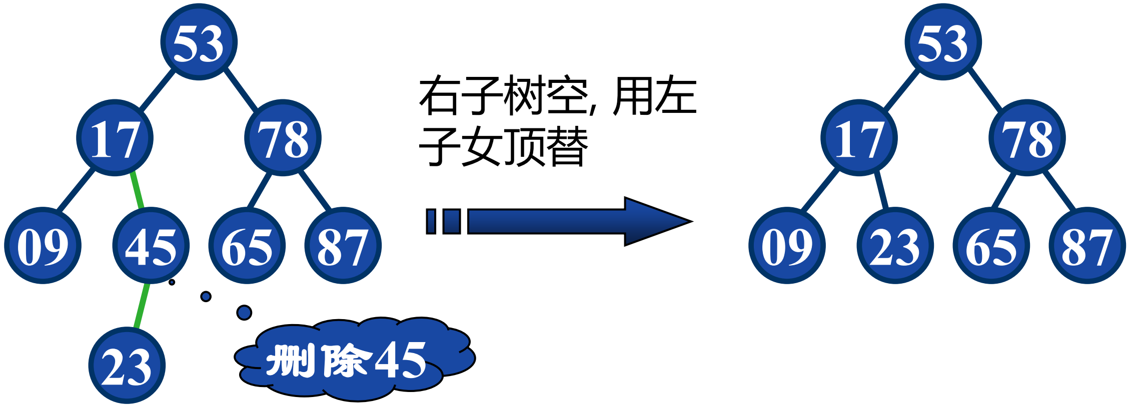
- 输入数据 { 53, 78, 65, 17, 87, 09, 81, 15 }

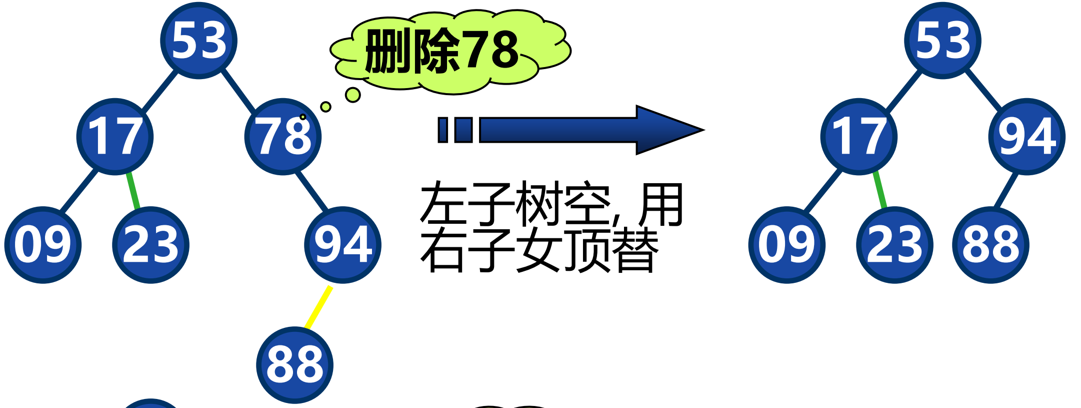


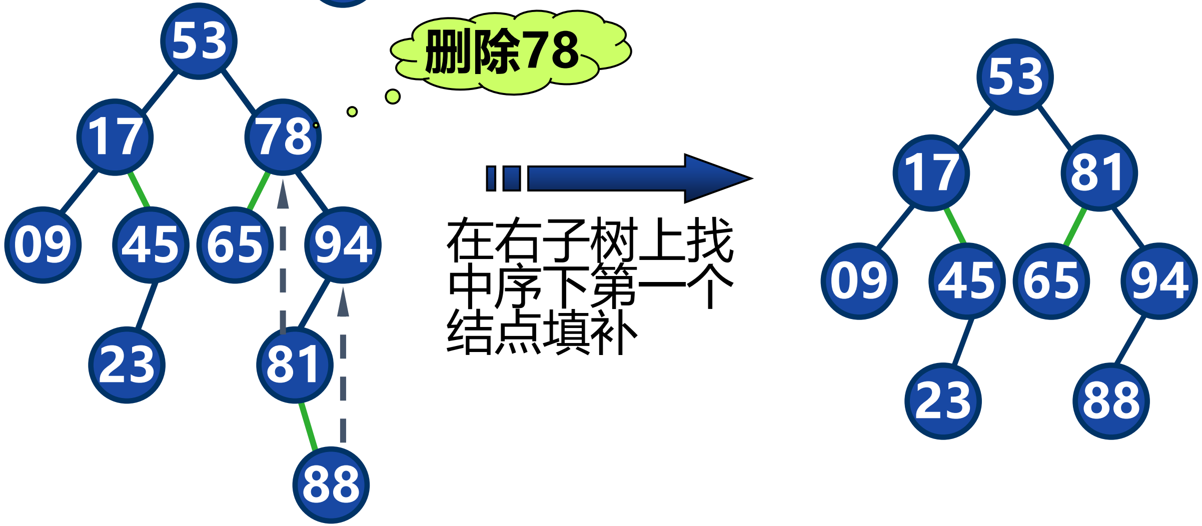
- 同样 3 个数据{ 1, 2, 3 }，输入顺序不同，建立起来的二叉排序树的形态也不同。这直接影响到二叉排序树的查找性能。
- 如果输入序列选得不好，会建立起一棵单支树，使得二叉排序树的高度达到最大。
- {2, 1, 3}
- {1, 2, 3} {1, 3, 2} {2, 3, 1} {3, 1, 2} {3, 2, 1}



二叉排序树的删除




删除78
左子树空, 用右子女顶替


删除78
在右子树上找中序下第一个结点填补

二叉排序树的删除

- **在二叉排序树中删除一个结点时，必须将因删除结点而断开的二叉链表重新链接起来，同时确保二叉排序树的性质不会失去。**
- **为保证在删除后树的查找性能不至于降低，还需要防止重新链接后树的高度增加。**
  - 被删结点的右子树为空，可以拿它的左子女结点顶替它的位置，再释放它。
  - 被删结点的左子树为空，可以拿它的右子女结点顶替它的位置，再释放它。
  - 被删结点的左、右子树都不为空，可以在它的右子树中寻找中序下的第一个结点(所有比被删关键字大的关键字中它的值最小),用它的值填补到被删结点中，再来处理这个结点的删除问题。当然，也可以到它的左子树中寻找中序下的最后一个结点。

二叉排序树性能分析

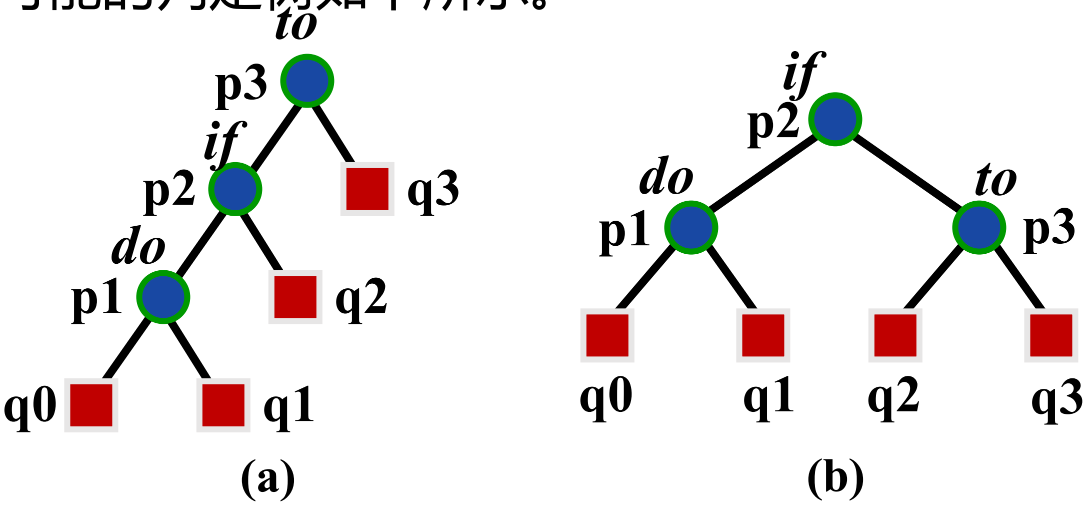
- 对于有 n 个关键字的集合，其关键字有 n! 种不同排列，可构成不同二叉排序树有

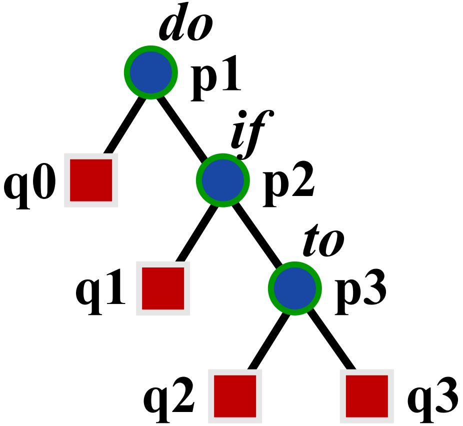
$$\frac{1}{n+1}C_{2n}^{n}$$

(棵)

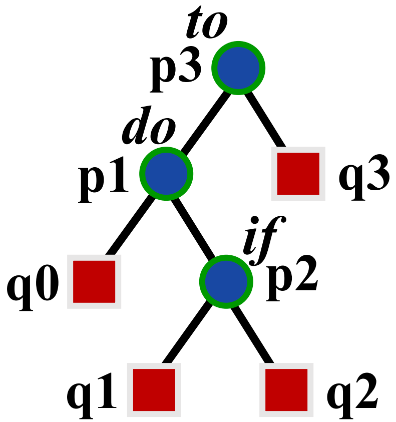
- 用树的查找效率来评价这些二叉排序树。
- 用判定树来描述查找过程，在判定树中，○表示内结点，它包含关键字集合中的某一个关键字；□表示外结点，代表各关键字间隔中的不在关键字集合中的关键字。它们是查找失败时到达的结点，物理上实际不存在。

- 在每两个外部结点间必存在一个内部结点。
- 例，已知关键字集合 { a1, a2, a3 } = { do, if, to }，对应查找概率 p1, p2, p3, 在各查找不成功间隔内查找概率分别为 q0, q1, q2, q3。可能的判定树如下所示。

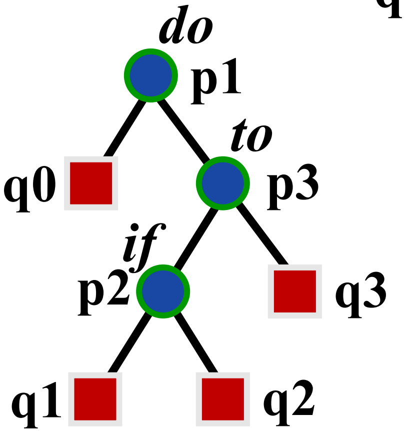



do
p1
if
q0
p2
to
q1
p3
q2
q3

(c)


to
p3
do
p1
q3
if
q0
p2
q1
q2

(d)


do
p1
to
q0
p3
if
p2
q3
q1
q2

(e)

- 一棵判定树的查找成功的平均查找长度ASLsucc 定义为该树所有内结点上的权值 p[i]与查找该结点时所需的关键字比较次数c[i] (= l[i])乘积之和：

$$ASL_{succ} = \sum_{i=1}^{n} p[i] * l[i].$$

- 查找不成功的平均查找长度ASLunsucc为树中所有外结点上权值q[j]与到达该外结点所需关键字比较次数c'[j] (= l'[j]-1)乘积之和：

$$ASL_{unsucc} = \sum_{j=0}^{n} q[j] * (l'[j] - 1).$$

相等查找概率的情形

设树中所有内、外结点的查找概率都相等：

$p[i] = 1/3, \ 1 \le i \le 3$

$q[j] = 1/4, \ 0 \le j \le 3$

图(a): ASLsucc = 1/3*(3 + 2 +1) = 6/3 = 2

ASLunsucc = 1/4*(3 + 3 +2 + 1) = 9/4

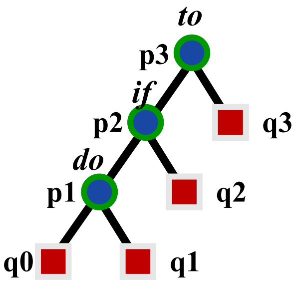
图(b): ASLsucc = 1/3*(2 + 1 + 2) = 5/3

ASLunsucc = 1/4*(2 + 2 + 2 + 2) = 8/4

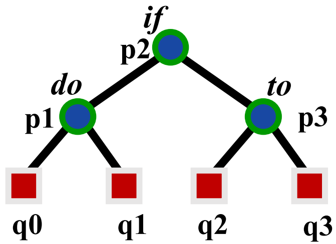
图(c): ASLsucc = 2,   ASLunsucc = 9/4

图(d): ASLsucc = 2,   ASLunsucc = 9/4

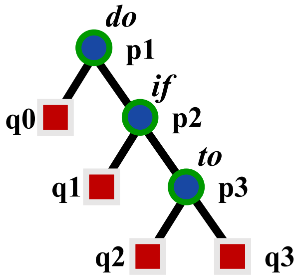
图(e): ASLsucc = 2,   ASLunsucc = 9/4



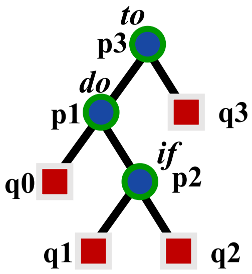
(a)



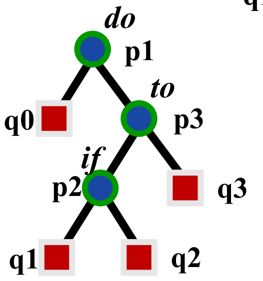
(b)



(c)



(d)



(e)

- 图(b)的情形所得的平均查找长度最小。一般把平均查找长度达到最小的判定树称作最优二叉排序树。
- 在相等查找概率的情形下，最优二叉排序树的上面 h-1（h是高度）层都是满的，只有第 h 层不满。如果结点集中在该层的左边，则它是完全二叉树；如果结点散落在该层各个地方，则有人称之为理想平衡树。