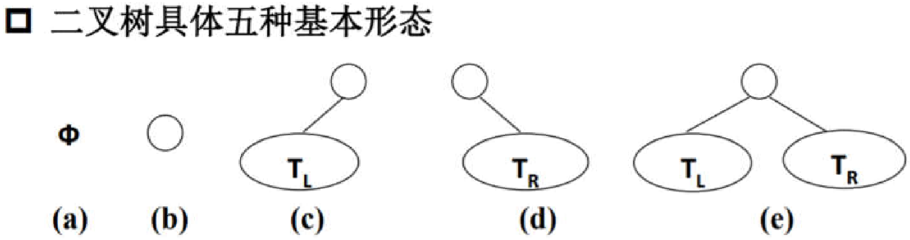
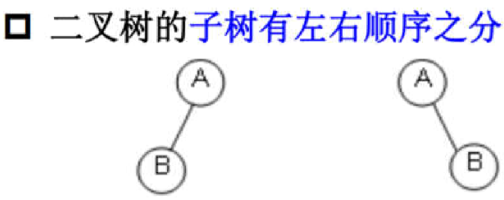
二叉树

2019年6月30日 22:34

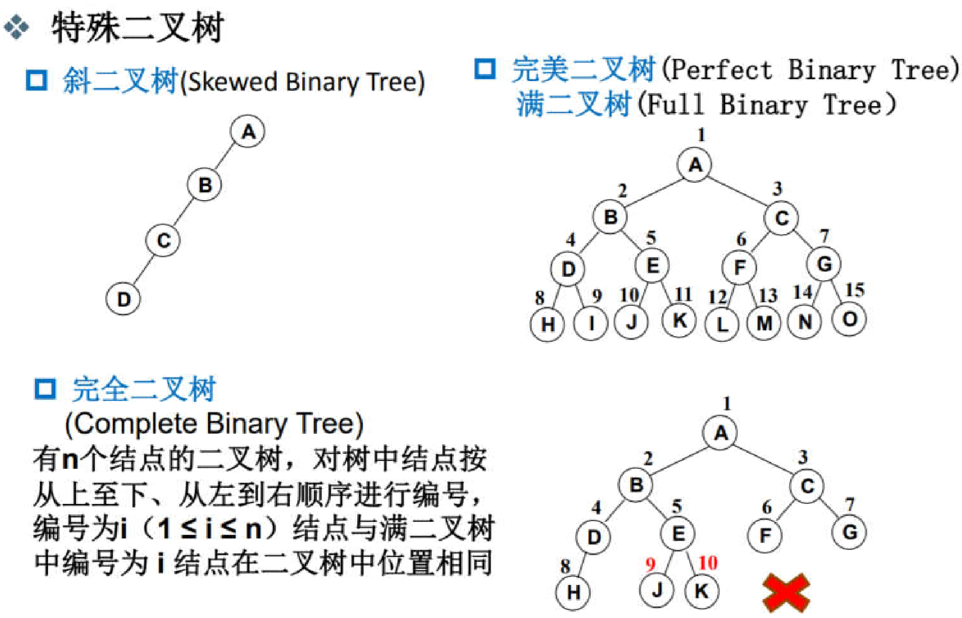
定义：

二叉树T：一个有穷的结点集合。 这个集合可以为空 若不为空，则它是由根结点和称为其左子树TL和右子树TR的 两个不相交的二叉树组成。





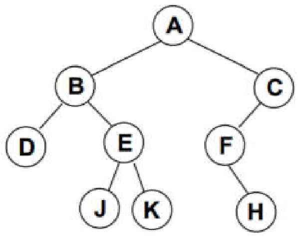
特殊二叉树



二叉树的性质

二叉树几个重要性质

- 一个二叉树第 *i* 层的最大结点数为：2^{i-1}，$i \geq 1$。
- 深度为*k*的二叉树有最大结点总数为： $2^{k}-1$，$k \geq 1$。
- 对任何非空二叉树 T，若n_0表示叶结点的个数、n_2是度为2的非叶结点个数，那么两者满足关系$n_0 = n_2 + 1$。



- $n_0 = 4$，$n_1 = 2$
- $n_2 = 3$；
- $n_0 = n_2 + 1$

二叉树的抽象数据类型定义

类型名称：二叉树
数据对象集：一个有穷的结点集合。
若不为空，则由根结点和其左、右二叉子树组成。

操作集： BT∈ BinTree, Item ∈ ElementType，重要操作有：
1、Boolean IsEmpty(BinTree BT)： 判别BT是否为空；
2、void Traversal(BinTree BT)：遍历，按某顺序访问每个结点；
3、BinTree CreatBinTree( )：创建一个二叉树。

常用的遍历方法有：
- void PreOrderTraversal(BinTree BT)：先序----根、左子树、右子树；
- void InOrderTraversal(BinTree BT)： 中序---左子树、根、右子树；
- void PostOrderTraversal(BinTree BT)：后序---左子树、右子树、根
- void LevelOrderTraversal(BinTree BT)：层次遍历，从上到下、从左到右

存储结构

1. 顺序存储结构

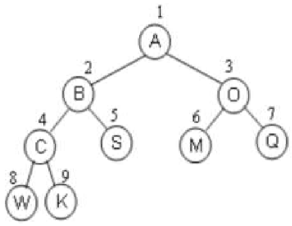
完全二叉树：按从上至下、从左到右顺序存储 n个结点的完全二叉树的结点父子关系：

- 非根结点（序号 i > 1）的父结点的序号是 ⌊i / 2⌋；
- 结点（序号为 i ）的左孩子结点的序号是 2i， （若2 i <= n，否则没有左孩子）；
- 结点（序号为 i ）的右孩子结点的序号是 2i+1， （若2 i +1<= n，否则没有右孩子）；

1. 顺序存储结构

完全二叉树：按从上至下、从左到右顺序存储

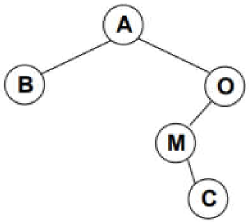
n个结点的完全二叉树的结点父子关系：

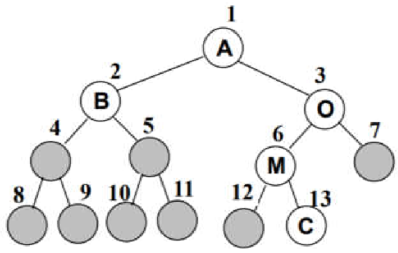


- 非根结点（序号 i > 1）的父结点的序号是 $\lfloor i/2 \rfloor$;
- 结点（序号为 i ）的左孩子结点的序号是 2i，（若2 i <= n，否则没有左孩子）；
- 结点（序号为 i ）的右孩子结点的序号是 2i+1，（若2 i +1<= n，否则没有右孩子）；

结点	A	B	O	C	S	M	Q	W	K
序号	1	2	3	4	5	6	7	8	9

· 一般二叉树也可以采用这种结构，但会造成空间浪费......





(a)一般二叉树　　(b) 对应的完全二叉树

结点	A	B	O	∧	∧	M	∧	∧	∧	∧	∧	∧	C
序号	1	2	3	4	5	6	7	8	9	10	11	12	13

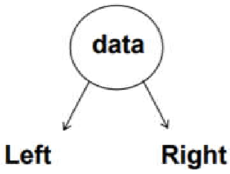
造成空间浪费！

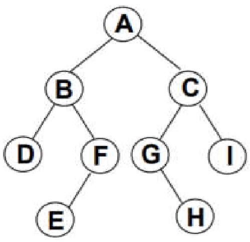
2.链表存储

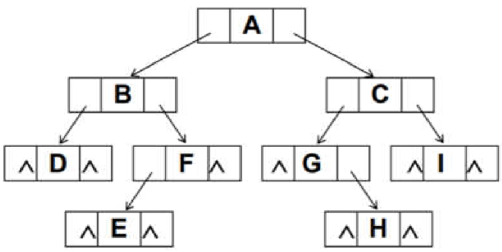
2. 链表存储

```
typedef struct TreeNode *BinTree;
typedef BinTree Position;
struct TreeNode{
        ElementType Data;
        BinTree Left;
        BinTree Right;
}
```

Left	Data	Right







二叉树的遍历

（1）先序遍历

遍历过程为：①访问根结点；②先序遍历其左子树；③先序遍历其右子树。

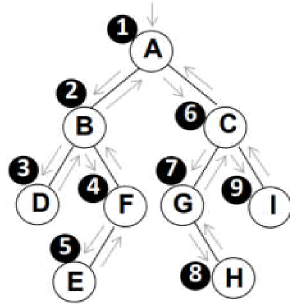
（1）先序遍历

遍历过程为：
① 访问根结点；
② 先序遍历其左子树；
③ 先序遍历其右子树。

A （B D F E） （C G H I）

先序遍历=> A B D F E C G H I

```
void PreOrderTraversal( BinTree BT )
{
    if( BT ) {
        printf("%d", BT->Data);
        PreOrderTraversal( BT->Left );
        PreOrderTraversal( BT->Right );
    }
}
```



（2）中序遍历

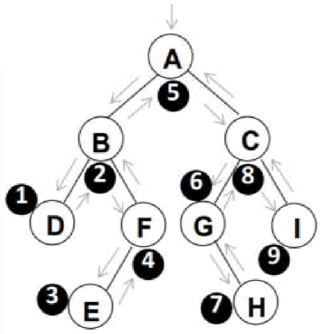
遍历过程为：①中序遍历其左子树；②访问根结点；③中序遍历其右子树。

（D B E F） A （G H C I）

遍历过程为：
① 中序遍历其左子树；
② 访问根结点；
③ 中序遍历其右子树。

中序遍历=> D B E F A G H C I

```
void InOrderTraversal( BinTree BT )
{
    if( BT ) {
        InOrderTraversal( BT->Left );
        printf("%d", BT->Data);
        InOrderTraversal( BT->Right );
    }
}
```



（3）后序遍历

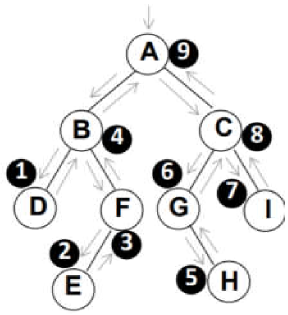
遍历过程为：①后序遍历其左子树；②后序遍历其右子树；③访问根结点。

遍历过程为：
① 后序遍历其左子树；
② 后序遍历其右子树；
③ 访问根结点。

（D E F B） （H G I C） A

后序遍历=> D E F B H G I C A

```
void PostOrderTraversal( BinTree BT )
{
    if( BT ) {
        PostOrderTraversal( BT->Left );
        PostOrderTraversal( BT->Right);
        printf("%d", BT->Data);
    }
}
```



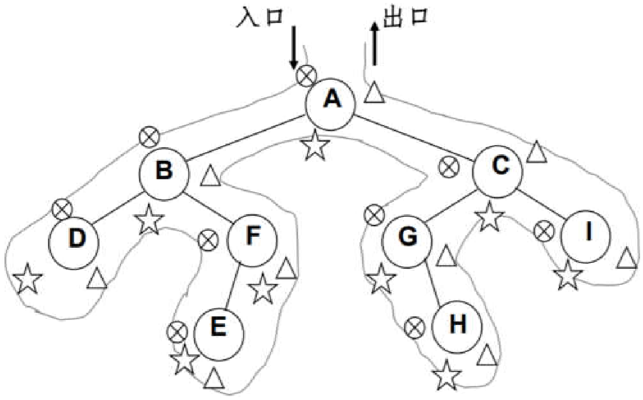
· 先序、中序和后序遍历过程：遍历过程中经过结点的路线一 样，只是访问各结点的时机不同。

· 图中在从入口到出口的曲线上用 · 、 · 和 · 三种符号分别标 记出了先序、中序和后序访问各

结点的时刻

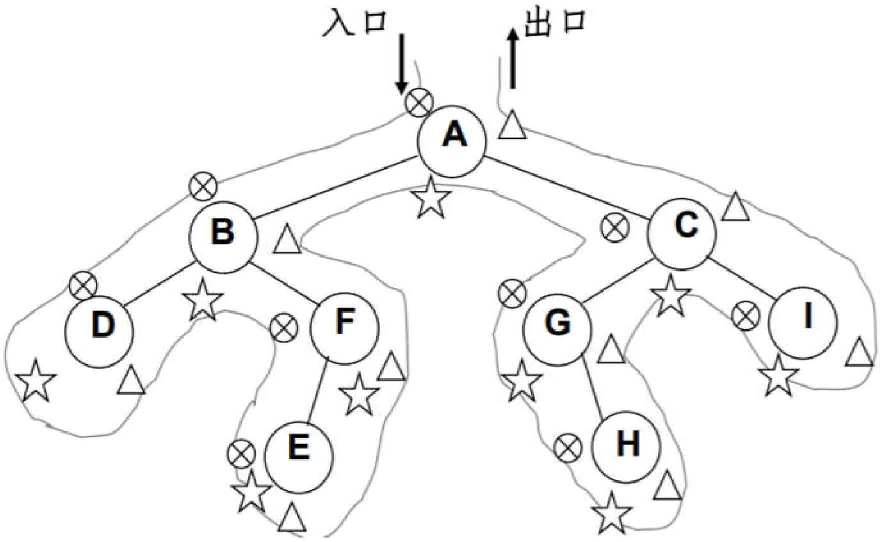
❖ 先序、中序和后序遍历过程：遍历过程中经过结点的路线一样，只是访问各结点的时机不同。

❖ 图中在从入口到出口的曲线上用⊗、☆ 和△三种符号分别标记出了先序、中序和后序访问各结点的时刻



二叉树的非递归遍历

中序遍历非递归遍历算法 非递归算法实现的基本思路：使用堆栈



❖中序遍历非递归遍历算法

- 遇到一个结点，就把它压栈，并去遍历它的左子树；
- 当左子树遍历结束后，从栈顶弹出这个结点并访问它；
- 然后按其右指针再去中序遍历该结点的右子树。

```
void InOrderTraversal( BinTree BT )
{  BinTree T=BT;
   Stack S = CreatStack( MaxSize ); /*创建并初始化堆栈S*/
   while( T || !IsEmpty(S) ){
       while(T){    /*一直向左并将沿途结点压入堆栈*/
           Push(S,T);
           T = T->Left;
       }
       if(!IsEmpty(S)){
           T = Pop(S); /*结点弹出堆栈*/
           printf("%5d", T->Data); /*（访问）打印结点*/
           T = T->Right; /*转向右子树*/
       }
   }
}
```

一直向左走，直到左边结束，抛出父节点，转向右子树，再循环向左，抛父节点，转右。。。

先序遍历的非递归遍历算法？

```
void InOrderTraversal( BinTree BT )
{   BinTree T BT;
    Stack S = CreatStack( MaxSize ); /*创建并初始化堆栈S*/
    while( T || !IsEmpty(S) ){
        while(T){    /*一直向左并将沿途结点压入堆栈*/
            Push(S,T);
            T = T->Left;
        }
        if(!IsEmpty(S)){
            T = Pop(S); /*结点弹出堆栈*/
            printf("%5d", T->Data); /*（访问）打印结点*/
            T = T->Right; /*转向右子树*/
        }
    }
}
```

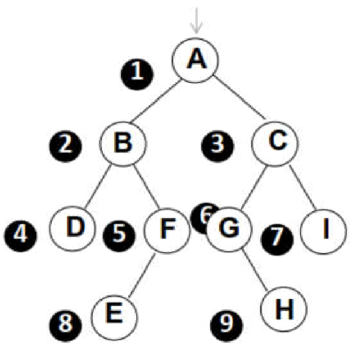
二叉树的层序遍历（递归）

二维结构线性化，利用队列

❖ 队列实现：遍历从根结点开始，首先将根结点入队，然后开始执行循环：结点出队、访问该结点、其左右儿子入队

A B C D F G I E H

层序遍历 => A B C D F G I E H



层序基本过程：先根结点入队，然后：

① 从队列中取出一个元素；
② 访问该元素所指结点；
③ 若该元素所指结点的左、右孩子结点非空，
则将其左、右孩子的指针顺序入队。

```
void LevelOrderTraversal ( BinTree BT )
{   Queue Q;  BinTree T;
    if ( !BT ) return; /* 若是空树则直接返回 */
    Q = CreatQueue( MaxSize ); /*创建并初始化队列Q*/
    AddQ( Q, BT );
    while ( !IsEmptyQ( Q ) ) {
        T = DeleteQ( Q );
        printf("%d\n", T->Data); /*访问取出队列的结点*/
        if ( T->Left )  AddQ( Q, T->Left );
        if ( T->Right )  AddQ( Q, T->Right );
    }
}
```

例 遍历二叉树的应用：输出二叉树中的叶子结点。

□ 在二叉树的遍历算法中增加检测结点的“ 左右子树是否都为空”。

```
void PreOrderPrintLeaves( BinTree BT )
{
if( BT ) {
if ( !BT-Left && !BT->Right )
printf("%d", BT->Data );
PreOrderPrintLeaves ( BT->Left );
PreOrderPrintLeaves ( BT->Right );
}
}
```

两种遍历确定一棵树：必须有中序遍历

先序和中序遍历序列来确定一棵二叉树

〖分析〗

- 根据先序遍历序列第一个结点确定根结点；
- 根据根结点在中序遍历序列中分割出左右两个子序列
- 对左子树和右子树分别递归使用相同的方法继续分解。

二叉搜索树

BST

二叉搜索树：一棵二叉树，可以为空；如果不为空，满足以下性质：

1. 非空左子树的所有键值小于其根结点的键值。
2. 非空右子树的所有键值大于其根结点的键值。
3. 左、右子树都是二叉搜索树。

二叉搜索树的查找操作： **Find**

- 查找从根结点开始，如果树为空，返回**NULL**
- 若搜索树非空，则根结点关键字和**X**进行比较， 并进行不同处理：

① 若**X**小于根结点键值，只需在左子树中继续搜索；

② 如果**X**大于根结点的键值， 在右子树中进行继续搜索；

③ 若两者比较结果是相等，搜索完成，返回指向此结点的指针。

实现:

尾递归

```
Position Find( ElementType X, BinTree BST )
{
if( !BST ) return NULL; /*查找失败*/
if( X > BST->Data )
return Find( X, BST->Right ); /*在右子树中继续查找*/
Else if( X < BST->Data )
return Find( X, BST->Left ); /*在左子树中继续查找*/
else /* X == BST->Data */
return BST; /*查找成功， 返回结点的找到结点的地址*/
}
```

非递归

```
Position IterFind( ElementType X, BinTree BST )
{
while( BST ) {
if( X > BST->Data )
BST = BST->Right; /*向右子树中移动， 继续查找*/
else if( X < BST->Data )
BST = BST->Left; /*向左子树中移动， 继续查找*/
else /* X == BST->Data */
return BST; /*查找成功， 返回结点的找到结点的地址*/
}
return NULL; /*查找失败*/
}
```

查找的效率决定于树的高度

查找最大和最小元素

- 最大元素一定是在树的最右分枝的端结点上
- 最小元素一定是在树的最左分枝的端结点上

```
Position FindMax( BinTree BST )
{
if(BST )
while( BST->Right ) BST = BST->Right;
/*沿右分支继续查找，直到最右叶结点*/
return BST;
}
Position FindMin( BinTree BST )
{
if( !BST ) return NULL; /*空的二叉搜索树，返回NULL*/
else if( !BST->Left )
return BST; /*找到最左叶结点并返回*/
else
return FindMin( BST->Left ); /*沿左分支继续查找*/
}
```

二叉搜索树的插入

〖分析〗 关键是要找到元素应该插入的位置，
可以采用与**Find**类似的方法

```
BinTree Insert( ElementType X, BinTree BST )
{
if( !BST ){
/*若原树为空， 生成并返回一个结点的二叉搜索树*/
BST = malloc(sizeof(struct TreeNode));
BST->Data = X;
BST->Left = BST->Right = NULL;
}else /*开始找要插入元素的位置*/
if( X < BST->Data )
BST->Left = Insert( X, BST->Left);
/*递归插入左子树*/
```

else	if(X > BST->Data) BST->Right = Insert(X, BST->Right);

```
/*递归插入右子树*/
/* else X已经存在， 什么都不做 */
return BST;
}
```

二叉搜索树的删除

- 考虑三种情况：

要删除的是叶结点： 直接删除， 并再修改其父结点指针**---**置为**NULL**

要删除的结点只有一个孩子结点:

将其父结点的指针指向要删除结点的孩子结点

要删除的结点有左、右两棵子树：

用另一结点替代被删除结点： 右子树的最小元素 或者 左子树的最大元素

```
BinTree Delete( ElementType X, BinTree BST )
{ Position Tmp;
if( !BST ) printf("要删除的元素未找到");
else if( X < BST->Data )
BST->Left = Delete( X, BST->Left); /* 左子树递归删除 */
else if( X > BST->Data )
BST->Right = Delete( X, BST->Right); /* 右子树递归删除 */
else /*找到要删除的结点 */
if( BST->Left && BST->Right ) { /*被删除结点有左右两个子结点 */
Tmp = FindMin( BST->Right );
/*在右子树中找最小的元素填充删除结点*/
BST->Data = Tmp->Data;
BST->Right = Delete( BST->Data, BST->Right);
/*在删除结点的右子树中删除最小元素*/
} else { /*被删除结点有一个或无子结点*/
Tmp = BST;
if( !BST->Left ) /* 有右孩子或无子结点*/
BST = BST->Right;
else if( !BST->Right ) /*有左孩子或无子结点*/
BST = BST->Left;
free( Tmp );
}
return BST;
}
```