JavaScript常用基础算法

一、字符串

1.字符串中出现最多次数的字符

```
function findMaxDuplicateChar(str) {
    var cnt = {},  // 用来记录所有的字符的出现频次
    c = "";  // 用来记录最大频次的字符
    for (var i = 0;  i < str.length;  i++) {
        var ci = str[i];
        if (!cnt[ci]) {
            cnt[ci] = 1;
        } else {
            cnt[ci]++;
        }
        if (c == "" || cnt[ci] > cnt[c]) {
            c = ci;
        }
    }
    console.log(cnt)
    return c;
}
```

2.翻转字符串

```
function reverseString(str) {
    return str.split("").reverse().join("");
}
```

3.回文字符串

```
// 判断回文字符串
function palindrome(str) {
    var reg = /[\W\_]/g;
    var str0 = str.toLowerCase().replace(reg, "");
    var str1 = str0.split("").reverse().join("");
    return str0 === str1;
}
```

二、数组

```
// 数组去重
function uniqueArray(arr) {
    var temp = [];
    for (var i = 0; i < arr.length; i++) {
        if (temp.indexOf(arr[i]) == -1) {
            temp.push(arr[i]);
        }
    }
    return temp;
    //or
    return Array.from(new Set(arr));
}
```

三、排序

1.冒泡排序

```
// 冒泡排序
function bubbleSort(arr) {
    for(var i = 1, len = arr.length; i < len - 1; ++i) {
        for(var j = 0; j <= len - i; ++j) {
            if (arr[j] > arr[j + 1]) {
                let temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

2.快速排序

```
// 快速排序
function qSort(arr) {
    // 声明并初始化左边的数组和右边的数组
    var left = [], right = [];
    // 使用数组第一个元素作为基准值
    var base = arr[0];
    // 当数组长度只有1或者为空时，直接返回数组，不需要排序
    if(arr.length <= 1) return arr;
    // 进行遍历
    for(var i = 1, len = arr.length; i < len; i++) {
        if(arr[i] <= base) {
        // 如果小于基准值，push到左边的数组
            left.push(arr[i]);
        } else {
        // 如果大于基准值，push到右边的数组
            right.push(arr[i]);
        }
```

```
    }
    // 递归并且合并数组元素
    return [...qSort(left), ...[base], ...qSort(right)];    //return
qSort(left).concat([base], qSort(right));
}
```

3.插入排序

```
// 插入排序 过程就像你拿到一副扑克牌然后对它排序一样
function insertionSort(arr) {
    var n = arr.length;
    // 我们认为arr[0]已经被排序，所以i从1开始
    for (var i = 1; i < n; i++) {
        // 取出下一个新元素，在已排序的元素序列中从后向前扫描来与该新元素比较大小
        for (var j = i - 1; j >= 0; j--) {
            if (arr[i] >= arr[j]) { // 若要从大到小排序，则将该行改为if (arr[i] <=
arr[j])即可
                // 如果新元素arr[i] 大于等于 已排序的元素序列的arr[j]，
                // 则将arr[i]插入到arr[j]的下一位置，保持序列从小到大的顺序
                arr.splice(j + 1, 0, arr.splice(i, 1)[0]);
                // 由于序列是从小到大并从后向前扫描的，所以不必再比较下标小于j的值比arr[j]小
的值，退出循环
                break;
            } else if (j === 0) {
                // arr[j]比已排序序列的元素都要小，将它插入到序列最前面
                arr.splice(j, 0, arr.splice(i, 1)[0]);
            }
        }
    }
    return arr;
}
```

当目标是升序排序，最好情况是序列本来已经是升序排序，那么只需比较n-1次，时间复杂度O(n)。最坏情况是序列本来是降序排序，那么需比较n(n-1)/2次，时间复杂度O(n2)。所以平均来说，插入排序的时间复杂度是O(n2)。显然，次方级别的时间复杂度代表着插入排序不适合数据特别多的情况，一般来说插入排序适合小数据量的排序。

在这段代码中，我们可以看到，这段代码实现了通过pivot区分左右部分，然后递归的在左右部分继续取pivot排序，实现了快速排序的文本描述，也就是说该的算法实现本质是没有问题的。

虽然这种实现方式非常的易于理解。不过该实现也是有可以改进的空间，在这种实现中，我们发现在函数内定义了left/right两个数组存放临时数据。随着递归的次数增多，会定义并存放越来越多的临时数据，需要Ω(n)的额外储存空间。

四、查找

```
// 二分查找
function binary_search(arr, l, r, v) {
    if (l > r) {
        return -1;
    }
    var m = parseInt((l + r) / 2);
```

```
    if (arr[m] == v) {
        return m;
    } else if (arr[m] < v) {
        return binary_search(arr, m+1, r, v);
    } else {
        return binary_search(arr, l, m-1, v);
    }
}
```

将二分查找运用到之前的插入排序中，形成二分插入排序，据说可以提高效率。但我测试的时候也许是数据量太少，并没有发现太明显的差距。。大家可以自己试验一下~（譬如在函数调用开始和结束使用console.time('插入排序耗时')和console.timeEnd('插入排序耗时')）

五、树的搜索/遍历

1.深度优先搜索

```
// 深搜 非递归实现
function dfs(node) {
    var nodeList = [];
    if (node) {
        var stack = [];
        stack.push(node);
        while(stack.length != 0) {
            var item = stack.pop();
            nodeList.push(item);
            var children = item.children;
            for (var i = children.length-1; i >= 0; i--) {
                stack.push(children[i]);
            }
        }
    }
    return nodeList;
}

// 深搜 递归实现
function dfs(node, nodeList) {
    if (node) {
        nodeList.push(node);
        var children = node.children;
        for (var i = 0; i < children.length; i++) {
            dfs(children[i], nodeList);
        }
    }
    return nodeList;
}
```

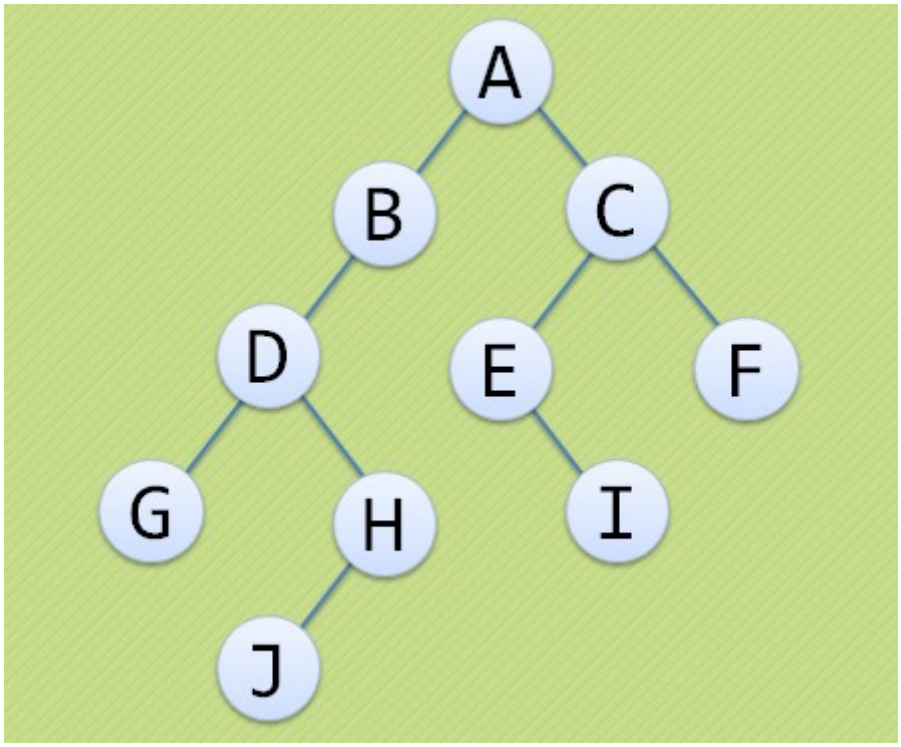
2.广度优先搜索

```
// 广搜 非递归实现
function bfs(node) {
    var nodeList = [];
    if (node != null) {
        var queue = [];
        queue.unshift(node);
        while (queue.length != 0) {
            var item = queue.shift();
            nodeList.push(item);
            var children = item.children;
            for (var i = 0; i < children.length; i++){
                queue.push(children[i]);
            }
        }
    }
    return nodeList;
}

// 广搜 递归实现
var i=0;  // 自增标识符
function bfs(node, nodeList) {
    if (node) {
        nodeList.push(node);
        if (nodeList.length > 1) {
            bfs(node.nextElementSibling, nodeList);  // 搜索当前元素的下一个兄弟元素
        }
        node = nodeList[i++];
        bfs(node.firstElementChild, nodeList); // 该层元素节点遍历完了，去找下一层的节点遍历
    }
    return nodeList;
}
```

六、二叉树

二叉树（Binary Tree）是n（n>=0）个结点的有限集合，该集合或者为空集（空二叉树），或者由一个根结点和两棵互不相交的、分别称为根结点的左子树和右子树的二叉树组成。



1. 二叉树的特点

每个结点最多有两棵子树，所以二叉树中不存在度大于2的结点。二叉树中每一个节点都是一个对象，每一个数据节点都有三个指针，分别是指向父母、左孩子和右孩子的指针。每一个节点都是通过指针相互连接的。相连指针的关系都是父子关系。

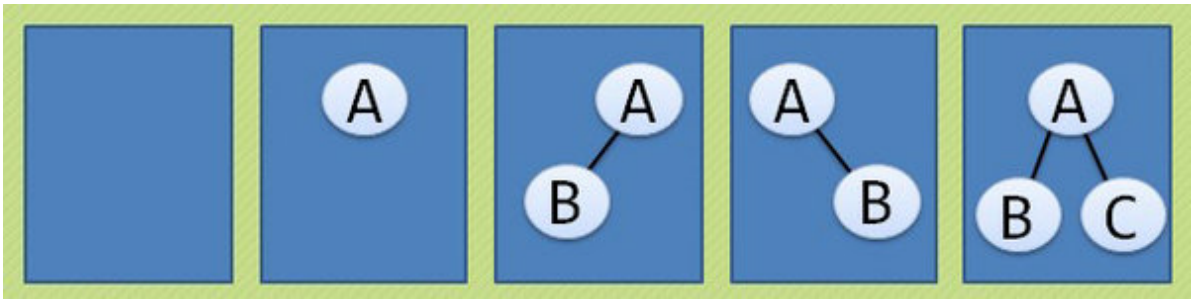
2. 二叉树节点的定义

二叉树节点定义如下：

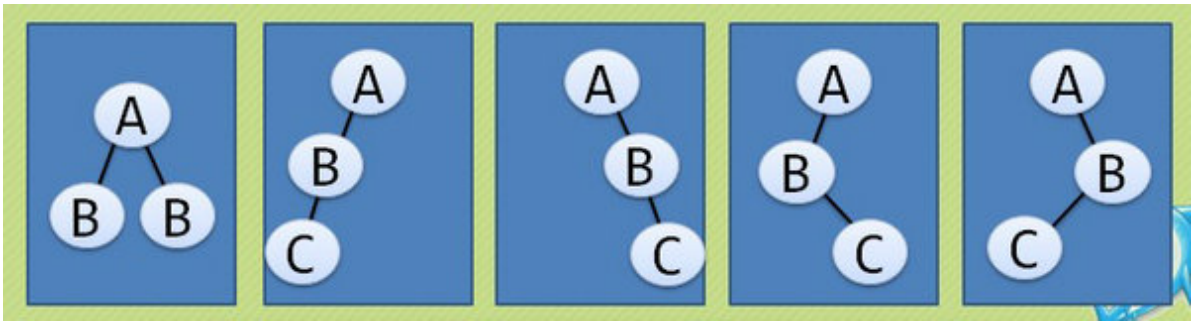
```
struct BinaryTreeNode
{
    int m_nValue;
    BinaryTreeNode* m_pLeft;
    BinaryTreeNode* m_pRight;
};
```

3. 二叉树的五种基本形态

```
空二叉树
只有一个根结点
根结点只有左子树
根结点只有右子树
根结点既有左子树又有右子树
```



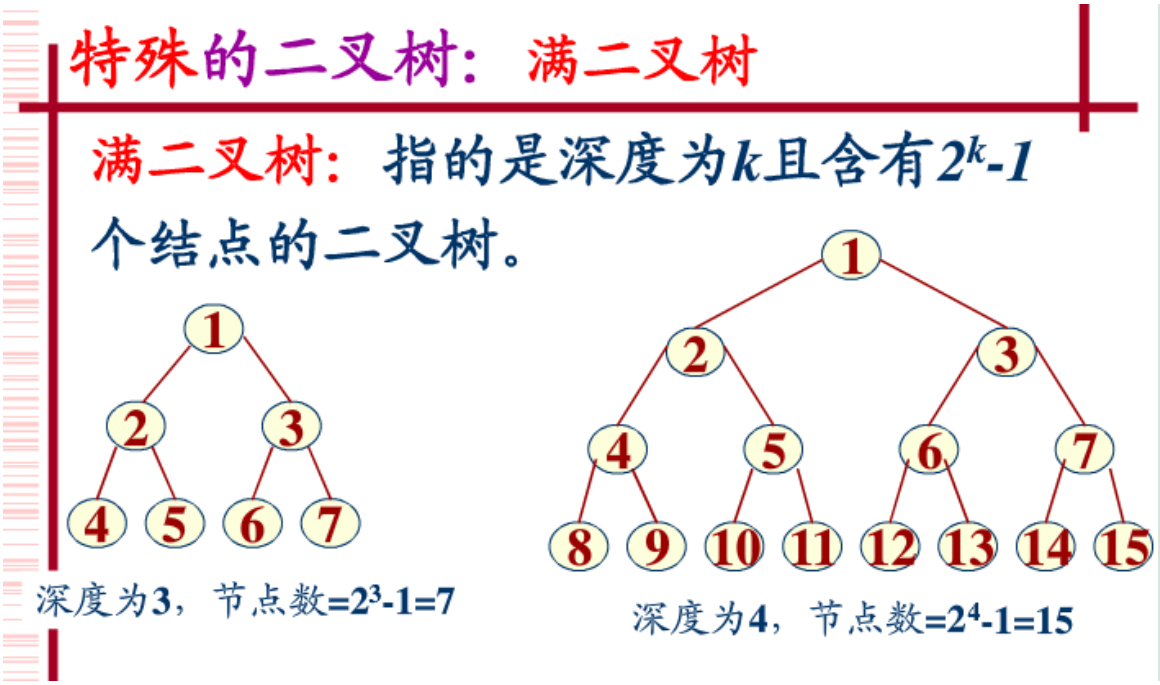
拥有三个结点的普通树只有两种情况：两层或者三层。但由于二叉树要区分左右，所以就会演变成如下的五种形态：



4. 特殊二叉树

满二叉树

在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶子都在同一层上，这样的二叉树称为满二叉树。如下图所示：



完全二叉树

完全二叉树是指最后一层左边是满的，右边可能满也可能不满，然后其余层都是满的。一个深度为k，节点个数为 2^k - 1 的二叉树为满二叉树（完全二叉树）。就是一棵树，深度为k，并且没有空位。

完全二叉树的特点有：

叶子结点只能出现在最下两层。

最下层的叶子一定集中在左部连续位置。

倒数第二层，若有叶子结点，一定都在右部连续位置。

如果结点度为1，则该结点只有左孩子。

同样结点树的二叉树，完全二叉树的深度最小。

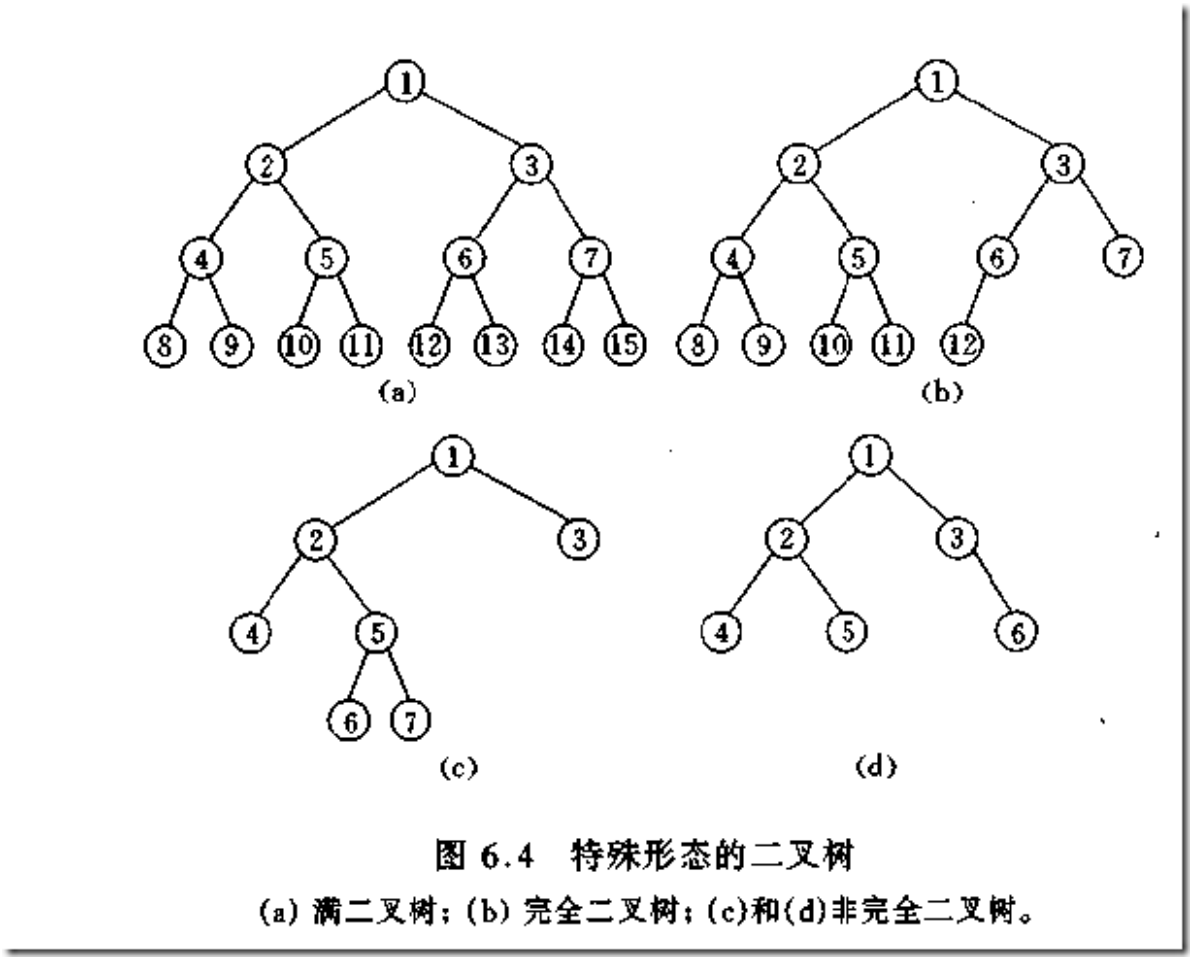


图 6.4 特殊形态的二叉树

(a) 满二叉树；(b) 完全二叉树；(c)和(d)非完全二叉树。

注意：满二叉树一定是完全二叉树，但完全二叉树不一定是满二叉树。

算法如下：

```
bool is_complete(tree *root)
{
    queue q;
    tree *ptr;
    // 进行广度优先遍历（层次遍历），并把NULL节点也放入队列
    q.push(root);
    while ((ptr = q.pop()) != NULL)
    {
        q.push(ptr->left);
        q.push(ptr->right);
    }

    // 判断是否还有未被访问到的节点
    while (!q.is_empty())
    {
        ptr = q.pop();
```

```
        // 有未访问到的的非NULL节点，则树存在空洞，为非完全二叉树
        if (NULL != ptr)
        {
            return false;
        }
    }

    return true;
}
```

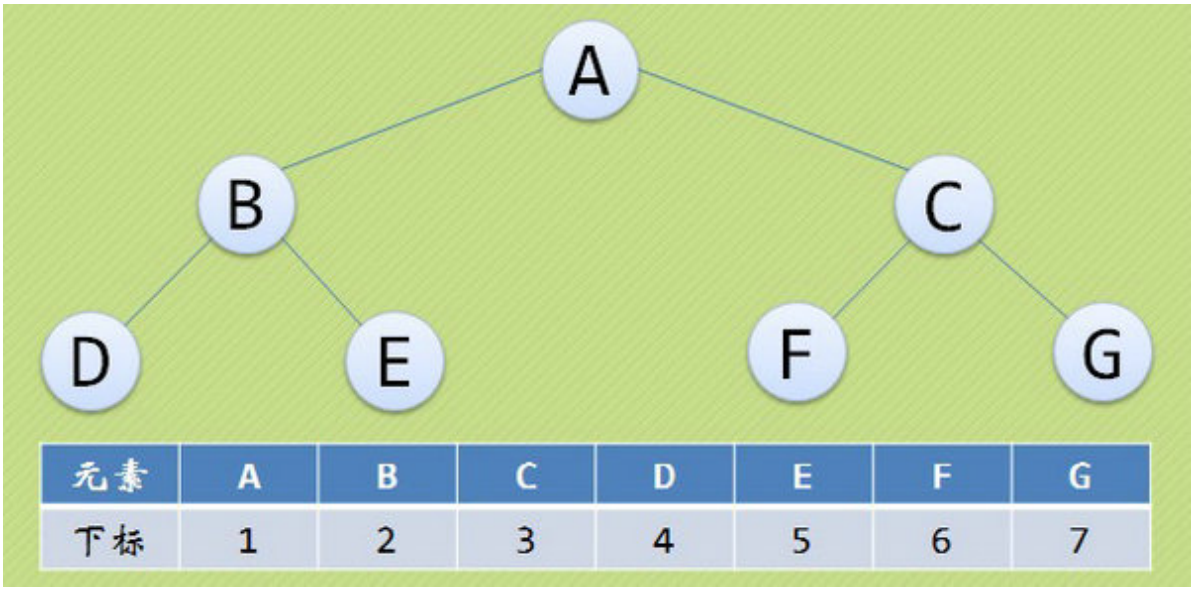
5. 二叉树的性质

二叉树的性质一：在二叉树的第i层上至多有2^(i-1)个结点(i>=1)

二叉树的性质二：深度为k的二叉树至多有2^k-1个结点(k>=1)

6. 二叉树的顺序存储结构

二叉树的顺序存储结构就是用一维数组存储二叉树中的各个结点，并且结点的存储位置能体现结点之间的逻辑关系。



7. 二叉链表

既然顺序存储方式的适用性不强，那么我们就要考虑链式存储结构啦。二叉树的存储按照国际惯例来说一般也是采用链式存储结构的。

二叉树每个结点最多有两个孩子，所以为它设计一个数据域和两个指针域是比较自然的想法，我们称这样的链表叫做二叉链表。

8. 二叉树的遍历

二叉树的遍历(traversing binary tree)是指从根结点出发，按照某种次序依次访问二叉树中所有结点，使得每个结点被访问一次且仅被访问一次。

二叉树的遍历有三种方式，如下：

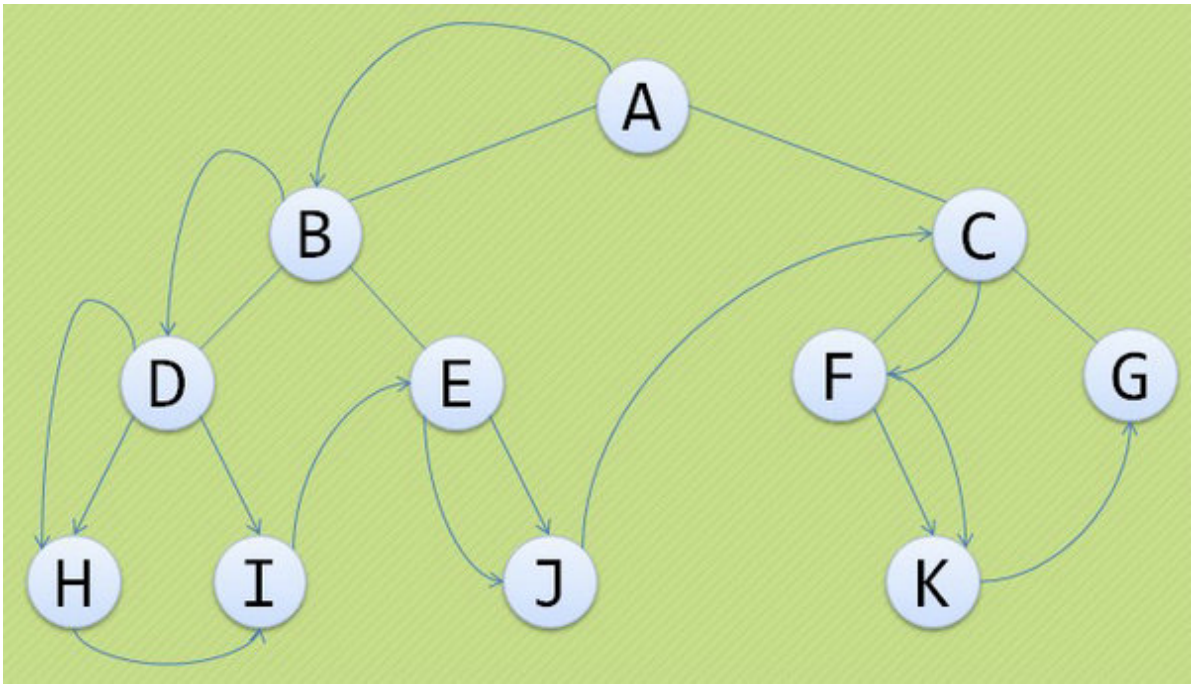
（1）前序遍历（DLR），首先访问根结点，然后遍历左子树，最后遍历右子树。简记根-左-右。

（2）中序遍历（LDR），首先遍历左子树，然后访问根结点，最后遍历右子树。简记左-根-右。

（3）后序遍历（LRD），首先遍历左子树，然后遍历右子树，最后访问根结点。简记左-右-根。

1）前序遍历：

若二叉树为空，则空操作返回，否则先访问根结点，然后前序遍历左子树，再前序遍历右子树。

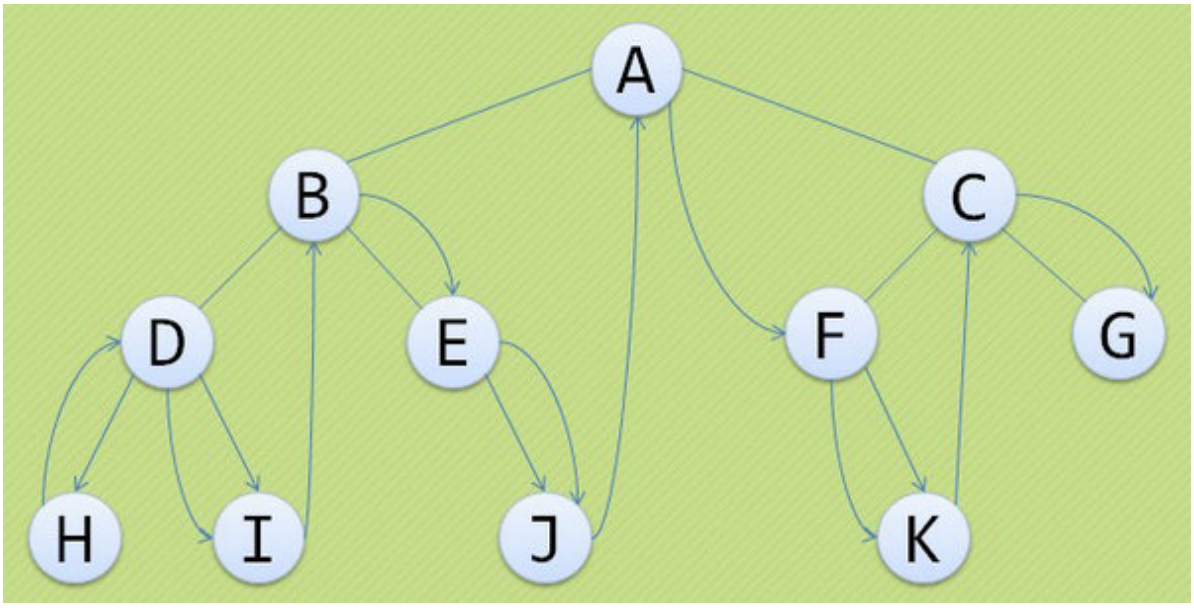


遍历的顺序为：`A B D H I E J C F K G`

```
//先序遍历
function preOrder(node){
    if(!node == null){
        putstr(node.show()+ " ");
        preOrder(node.left);
        preOrder(node.right);
    }
}
```

2）中序遍历：

若树为空，则空操作返回，否则从根结点开始（注意并不是先访问根结点），中序遍历根结点的左子树，然后是访问根结点，最后中序遍历右子树。

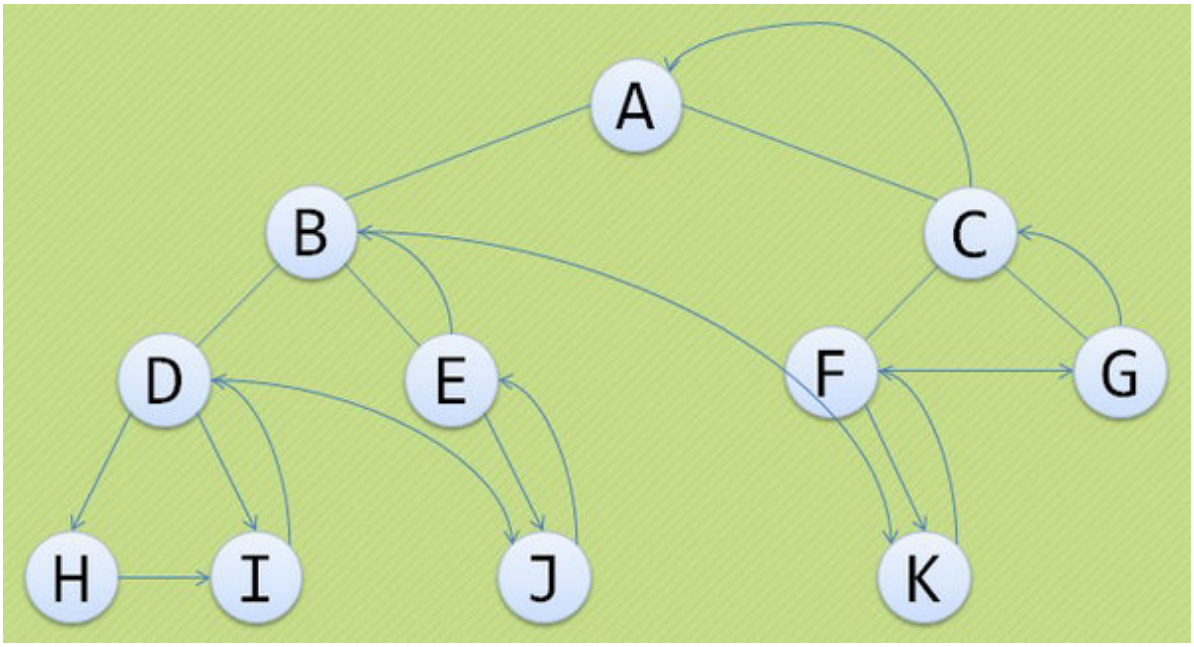


遍历的顺序为：`H D I B E J A F K C G`

```
//使用递归方式实现中序遍历
function inOrder(node){
    if(!(node == null)){
        inOrder(node.left);//先访问左子树
        putstr(node.show()+ " ");//再访问根节点
        inOrder(node.right);//最后访问右子树
    }
}
```

3）后序遍历：

若树为空，则空操作返回，否则从左到右先叶子后结点的方式遍历访问左右子树，最后访问根结点。



遍历的顺序为：`H I D J E B K F G C A`

```
//后序遍历
function postOrder(node){
    if(!node == null){
        postOrder(node.left);
        postOrder(node.right);
        putStr(node.show()+ " ");
    }
}
```

9. 实现二叉查找树

二叉查找树（BST）由节点组成，所以我们定义一个`Node`节点对象如下：

```
function Node(data,left,right){
    this.data = data;
    this.left = left;//保存left节点链接
    this.right = right;
    this.show = show;
}


function show(){
    return this.data;//显示保存在节点中的数据
}
```

10. 查找最大和最小值

查找BST上的最小值和最大值非常简单，因为较小的值总是在左子节点上，在BST上查找最小值，只需遍历左子树，直到找到最后一个节点

1）查找最小值

```
function getMin(){
    var current = this.root;
    while(!(current.left == null)){
        current = current.left;
    }
    return current.data;
}
```

该方法沿着BST的左子树挨个遍历，直到遍历到BST最左的节点，该节点被定义为：

```
current.left = null;
```

这时，当前节点上保存的值就是最小值

2）查找最大值

在BST上查找最大值只需要遍历右子树，直到找到最后一个节点，该节点上保存的值就是最大值。

```
function getMax(){
    var current = this.root;
    while(!(current.right == null)){
        current = current.right;
    }
    return current.data;
}
```