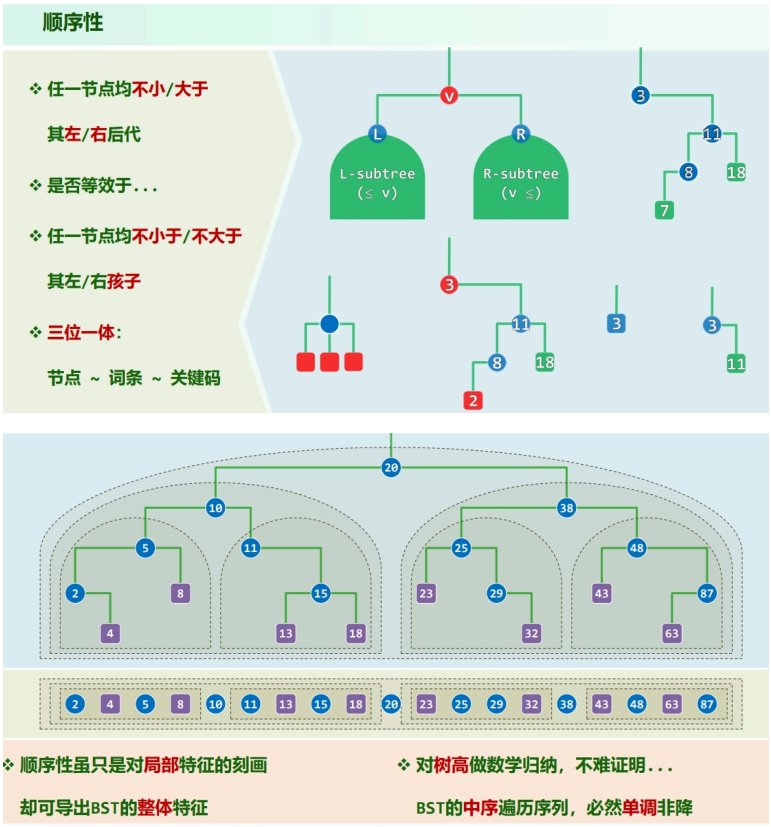
第9章 二叉搜索树（Binary Search Tree）

本章是二叉树的补充，也就是加入了一定性质的树，这里介绍一个后面所有树的公共祖先，二叉搜索树(BST);
后面在BST的基础上加了平衡的概念，有3种常见的BST的变体：AVL树，伸展树(Splay)，红黑树(Red-Black);
不过上面3个变体的插入删除操作需要维持树的平衡性+顺序性，比较麻烦，leetcode高频题中貌似也没看到，这里就不讲了。

In [1]:
```
#include <iostream>
using namespace std;
```

一、基本概念

简单来说就是满足顺序性的二叉树就叫二叉搜索树，下面这张图展示了什么叫顺序性，需要注意的是，
中间节点是大于等于左子树的所有节点的值，而不是仅仅大于左孩子的值。(我们后面讲的都是严格大于小于)



二、接口定义

就三个接口，查找，插入，删除
因为是后面各种变体树的公共祖先，这三个接口是虚方法，但是我们这里就不搞成虚方法了;
另外protected里面的是内部接口，就是实现上面search，insert和remove接口时，
某些中间步骤写成一个函数，这个函数作为中间步骤，放在protected里面，对外不可见。

```
template <typename T> class BST : public BinTree<T> {
public: //以virtual修饰，以便派生类重写
   virtual BinNodePosi<T> & search( const T & ); //查找
   virtual BinNodePosi<T> insert( const T & ); //插入
   virtual bool remove( const T & ); //删除
protected:
   /* ..... */
};
```

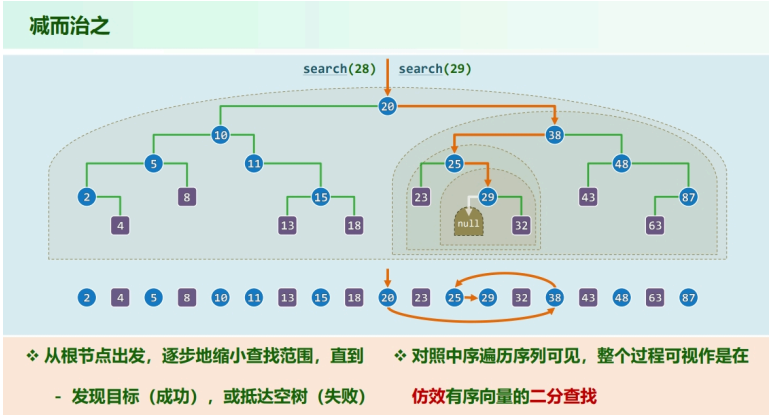
In [2]:
```
#include "bintree.h"
```

In [3]:
```
template <class T>
class BST: public BinTree<T>{
public:
    BinNodePointer(T) hot;
    using BinTree<T>::root;

    BinNodePointer(T) search( const T & );
    BinNodePointer(T) insert( const T & );
    bool remove( const T & );
protected:
    BinNodePointer(T) searchIn(BinNodePointer(T),T,BinNodePointer(T) &hot);
    BinNodePointer(T) removeAt(BinNodePointer(T));

};
```

三、接口实现

查找

由于顺序性，因此和根节点比较后，如果比根节点大，就从右子树中递归查找，否则从左子树递归查找。

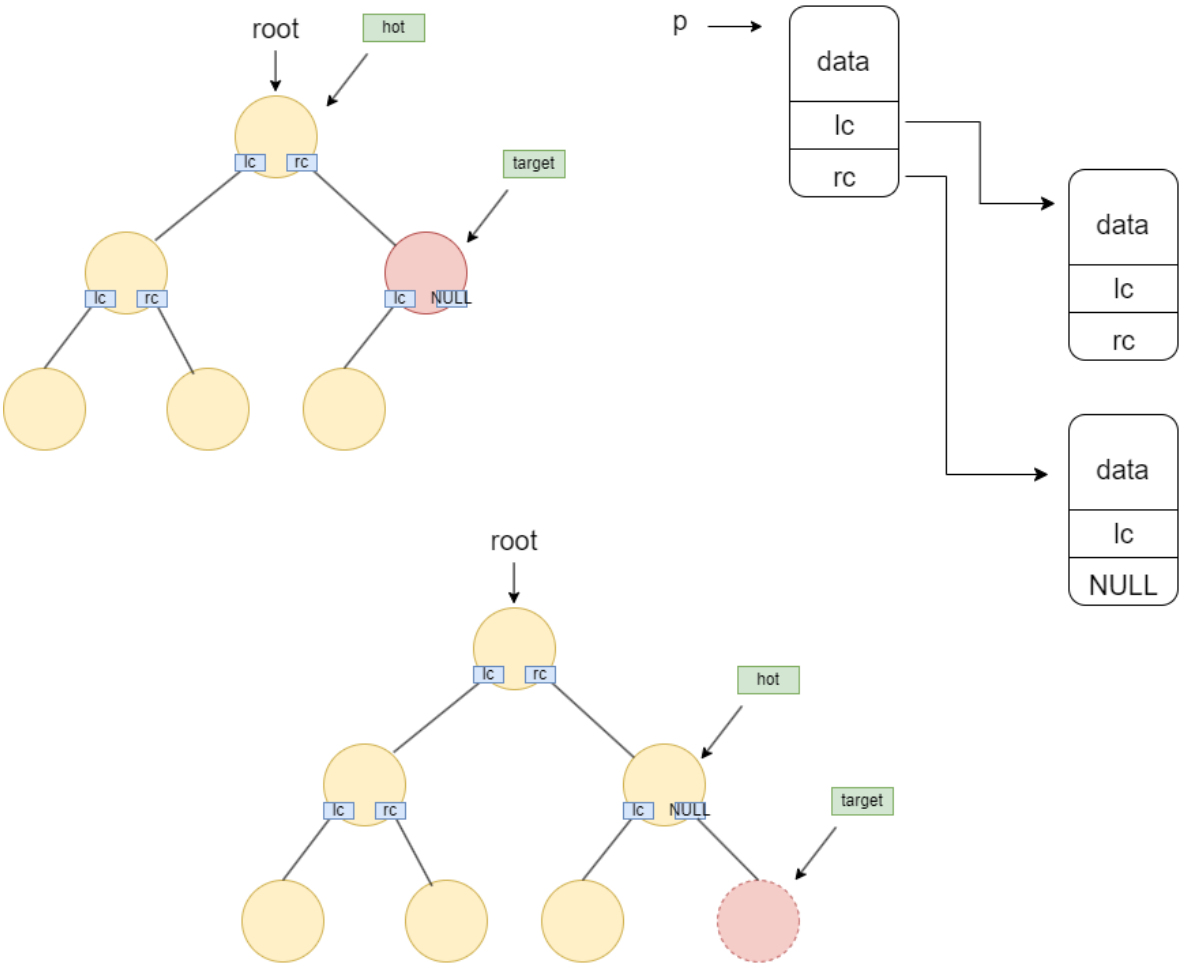


In [4]:
```
template <class T>
BinNodePointer(T) BST<T>::searchIn(BinNodePointer(T) p, T e, BinNodePointer(T)& hot){
    if (!p || p -> data == e) return p;
    hot = p;
    if ( e > p -> data){
        p = p -> rc;
        return searchIn(p,e,hot);
    }
    if ( e < p -> data) {
        p = p -> rc;
        return searchIn(p,e,hot);
    }
}
```

In [5]:
```
template <class T>
BinNodePointer(T) BST<T>::search(const T & e){
    hot = nullptr;
    return searchIn(root,e,hot);
}
```

查找命中后的内存状态

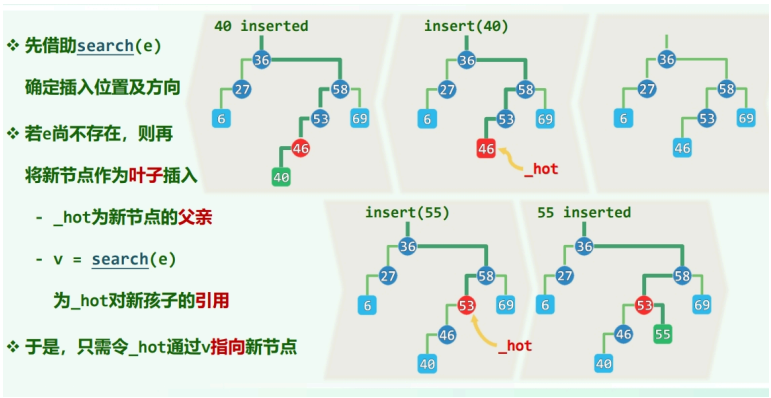
首先原二叉树的节点都是用lc和rc找到的，所以修改原二叉树，其实就是修改这些lc和rc，也就是下图节点中的小方块；
而上面查找的返回值是节点指针的一个拷贝，是重新创建了一个节点指针，虽然它和节点中的小方块存储的是相同的地址，但是它不是二叉树的一部分，但是它指向的节点确实是二叉树的节点没错，不过修改的时候需要当心。
比如对于插入操作，是下面target指向一个空节点(用虚圆表示)，想要在这个位置插入新建的节点，通过target是没办法的，如果令target=new节点，只是让target指向了新节点，原来的树的那个位置还是没有，所以只能通过hot->rc=new节点才能让最后hot节点里面那个rc指针指向新创建的节点。
再比如删除操作，看上面那个图，如果要删除target，还是只能让hot->rc指向待删除节点的左下角那个节点，没法直接通过target删除。



补充几句：
书上用的引用作为查找返回值，就是把原树中的小方块的别名，但是c++引用一旦确定了，就无法改变，后面的插入删除同样需要注意一些问题，这里就不采用这个方案了。

插入

这里插入后，返回root节点，主要参考leetcode里面的写法



In [6]:
```
template <class T>
BinNodePointer(T) BST<T>::insert(const T & e){
    BinNodePointer(T) target = search(e);
    if (target) return root;                // 如果直接找到了，不用插了，直接返回
    else {                                  // 如果是空指针，让其指向一个新创建的节点
        if (hot->data > target) hot->lc = new BinNode<T>(e);
        if (hot->data < target) hot->rc = new BinNode<T>(e);
        return root;
    }
}
```

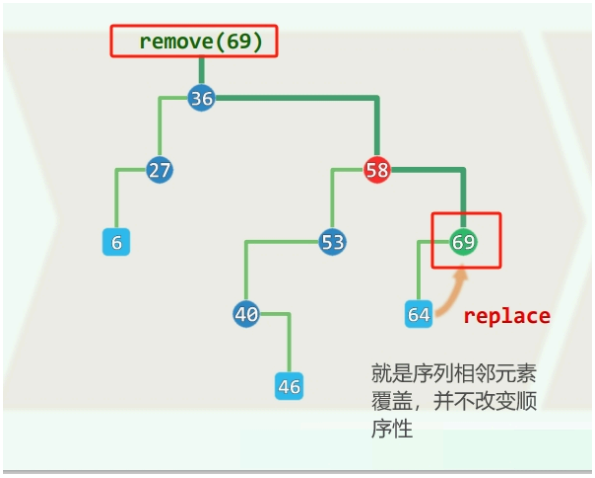
leetcode中的二叉搜索树插入节点

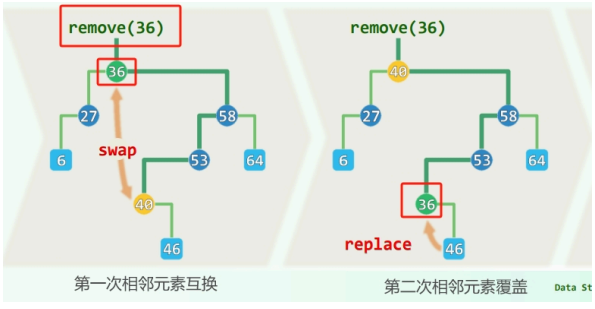
注意不是官方的代码，就是自己写的通过的算法

```
class Solution {
public:
    TreeNode* hot;
    TreeNode* searchIn(TreeNode* p, const int & val,TreeNode*& hot){
        if (!p || p->val == val)  return p;
        hot = p;
        if (p -> val < val) return searchIn(p -> right, val, hot);
        if (p -> val > val)  return searchIn(p -> left, val, hot);
        return p;
    }

    TreeNode* insertIntoBST(TreeNode* root, int val) {
        if (!root) return new TreeNode(val);
        TreeNode* x = searchIn(root, val,hot);
        if (!x) {
            if (hot->val > val) hot->left=new TreeNode(val);
            if (hot->val < val) hot->right=new TreeNode(val);
        }
        return root;
    }
};
```

删除

如果待删除的节点不是根节点，分3种情况：
1.没有左孩子，直接用右孩子顶替要删除的节点（从中序遍历序列来看，右侧大于e的所有值整体左移一格）；
2.没有右孩子，直接用左孩子顶替（从中序遍历序列来看，e左边整体右移一格）；
3.两个孩子都有，将e与它的直接后继（中序遍历意义下）交换（从序列来看，相当于相邻元素交换位置）,然后变成了情形1，用情形1删除即可。
实际上上面还可以将e与它的直接前继交换，然后变成情形2进行处理。





In [7]:
```
template <class T>
bool BST<T>::remove(const T & e){
    BinNodePointer(T) x = search(e);
    // 如果找不到要删除的节点，直接返回
    if (!x) return false;

    // 删除根节点，此时hot为nullptr，因此删除逻辑不太一样
    if (x==root){
        return false;  // 这里就不写了，参考下面其实不难写，考虑4种情况，没孩子，有左无右，有右无左，都有
                       // 其中两个孩子都有时，可以直接按照下面普通节点的逻辑删除，因为这种情况，hot会重置到正确的位
    }

    // 删除普通节点
```

```
    if (!haslchild(*x)) {
        if (hot->lc==x) hot->lc = x->rc;
        if (hot->rc==x) hot->rc = x->rc;
        return true;
    }
    if (!hasrchild(*x)) {
        if (hot->lc==x) hot->lc = x->lc;
        if (hot->rc==x) hot->rc = x->lc;
        return true;
    }
    // 情形3，先找直接后继，再交换，然后删除
    BinNodePointer(T) succ = x -> rc;
    hot = x;

    while(succ -> lc) {
        hot = succ;
        succ = succ -> lc;
    }
    swap(x->data,succ->data);
    // 然后执行删除succ，方法和前面完全一样,但是这里succ一定没有左孩子
    x = succ;
    if (hasrchild(*x)) {
        if (hot->lc==x) hot->lc = x->rc;
        if (hot->rc==x) hot->rc = x->rc;
        return true;
    }
    if (!hasrchild(*x)) {
        if (hot->lc==x) hot->lc = nullptr;
        if (hot->rc==x) hot->rc = nullptr;
        return true;
    }

}
```

leetcode中的二叉搜索树删除节点

```
    class Solution {
    public:
        TreeNode* hot;
        TreeNode* searchIn(TreeNode* p, const int & val,TreeNode*& hot){
            if (!p || p->val == val)  return p;
            hot = p;
            if (p -> val < val) return searchIn(p -> right, val, hot);
            if (p -> val > val)  return searchIn(p -> left, val, hot);
            return p;
        }

        TreeNode* deleteNode(TreeNode* root, int key) {
            TreeNode* x = searchIn(root,key,hot);
            // 找不到要删除的节点，直接返回
            if (!x) return root;

            // 待删除节点是root时,hot=nullptr,不能直接借助hot删除
            if (x == root) {
                if (!(x->left)&&!(x->right)) {
                return nullptr;
            }
            if ((x->left)&&!(x->right)) {
                root = x -> left;
                return root;
            }
            if ((x->right)&&!(x->left)) {
                root = x -> right;
                return root;
            }
            // 如果根节点既有左边又有右边，按普通节点的删除逻辑删除
            }

            // 处理其他节点
            if (!(x->left)&&!(x->right)) {
                if (hot->left==x) hot->left=nullptr;
                if (hot->right==x) hot->right=nullptr;
                return root;
            }
            if ((x->left)&&!(x->right)) {
```

```
                if (hot->left==x) hot->left=x->left;
                if (hot->right==x) hot->right=x->left;
                return root;
            }
            if ((x->right)&&!(x->left)) {
                if (hot->left==x) hot->left=x->right;
                if (hot->right==x) hot->right=x->right;
                return root;
            }

            TreeNode* succ = x -> right;
            hot = x;
            while (succ -> left) {
                hot = succ;
                succ = succ -> left;
            }
            swap(x->val,succ->val);

            // 下面就按照同样的步骤删除succ即可，只不过succ一定没有left
            if (succ -> right) {
                if (hot->left==succ) hot->left=succ->right;
                if (hot->right==succ) hot->right=succ->right;
            }
            if (!(succ -> right)){
                if (hot->left==succ) hot->left=nullptr;
                if (hot->right==succ) hot->right=nullptr;
            }
            return root;
        }
    };
```

In [ ]: