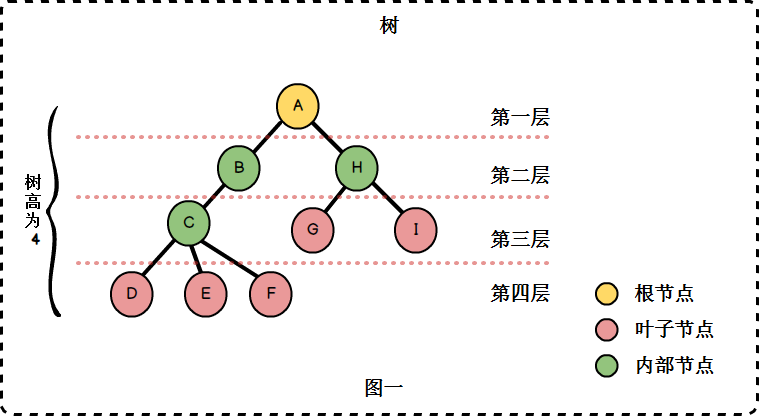
**1. 树的简介**

**1.1 树的特征**

树是一种数据结构，它是n(n>=0)个节点的有限集。n=0时称为空树。n>0时，有限集的元素构成一个具有层次感的数据结构。  


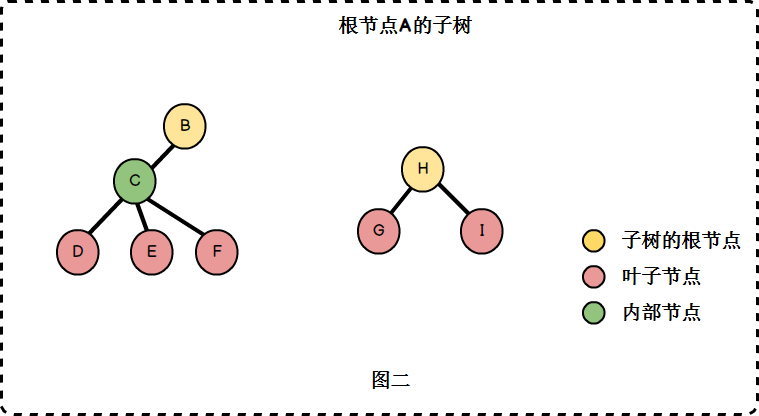
区别于线性表一对一的元素关系，树中的节点是一对多的关系。树具有以下特点：

1. n>0时，根节点是唯一的，不可能存在多个根节点。
2. 每个节点有零个至多个子节点；除了根节点外，每个节点有且仅有一个父节点。根节点没有父节点。

**1.2 树的相关概念**

树有许多相关的术语与概念，在学习树的结构之前，我们要熟悉这些概念。

1. **子树**：除了根节点外，每个子节点都可以分为多个不相交的子树。（图二）
2. **孩子与双亲**：若一个结点有子树，那么该结点称为子树根的"双亲"，子树的根是该结点的"孩子"。在图一中，B、H是A的孩子，A是B、H的双亲。
3. **兄弟**：具有相同双亲的节点互为兄弟，例如B与H互为兄弟。
4. **节点的度**：一个节点拥有子树的数目。例如A的度为2，B的度为1，C的度为3.
5. **叶子**：没有子树，也即是度为0的节点。
6. **分支节点**：除了叶子节点之外的节点，也即是度不为0的节点。
7. **内部节点**：除了根节点之外的分支节点。
8. **层次**：根节点为第一层，其余节点的层次等于其双亲节点的层次加1.
9. **树的高度**：也称为树的深度，树中节点的最大层次。
10. **有序树**：树中节点各子树之间的次序是重要的，不可以随意交换位置。
11. **无序树**：树种节点各子树之间的次序是不重要的。可以随意交换位置。
12. **森林**：0或多棵互不相交的树的集合。例如图二中的两棵树为森林。



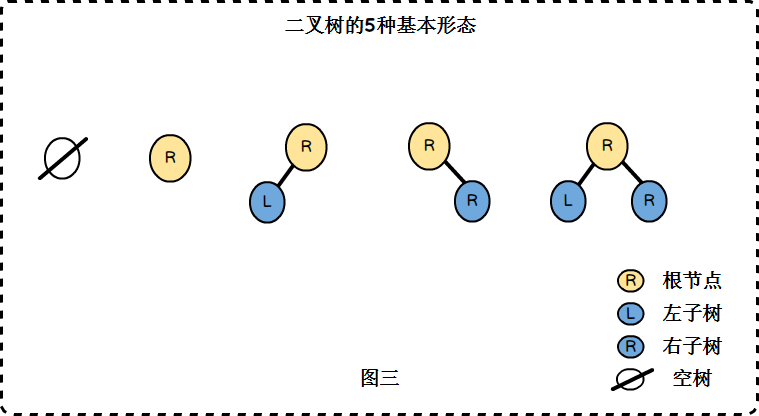
**2. 二叉树简介**

**2.1 二叉树的定义**

二叉树或者为空集，或者由一个根节点和两棵互不相交的、分别称为左子树和右子树的二叉树组成。从定义可以看出一棵二叉树：

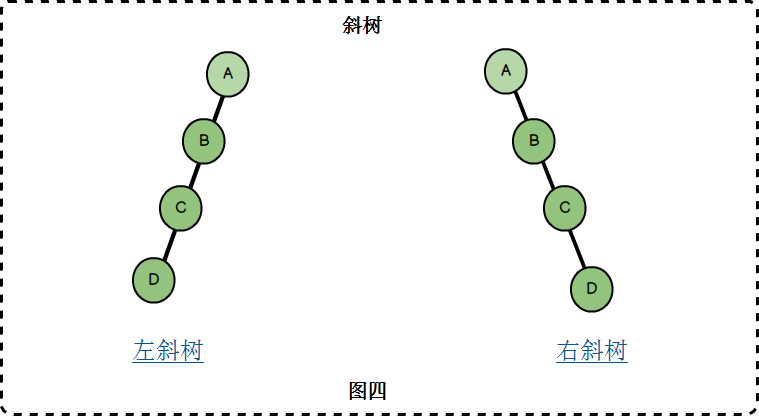
1. 二叉树是**有序树**，区分左子树与右子树，**不可以随意交换子树位置。**
2. **一个节点的子树数量取值范围为0，1，2。0代表该节点是叶子节点，1代表该节点只有左子树或只有右子树，2代表该节点有左右子树。**

根据定义，一棵二叉树有5中基本形态：



**2.2 斜树、满二叉树、完全二叉树、二叉查找树**

**斜树**

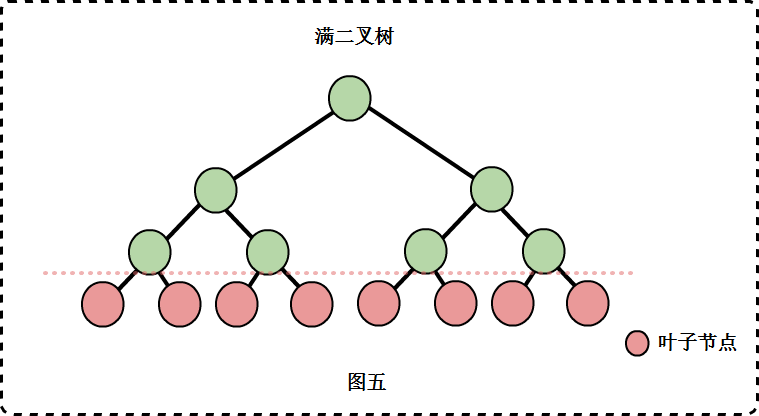
所有节点都只有左子树的二叉树叫做左斜树，所有节点都只有右子树的二叉树叫做右斜树。左斜树和右子树统称为斜树。  
斜树已经退化成线性结构，二叉树在查找上表现出来优异性能在斜树得不到体现。  


**注意：为了只关注重点，我们所有的节点都采用统一浅绿色着色，若有特殊节点将在图中备注**

**满二叉树**

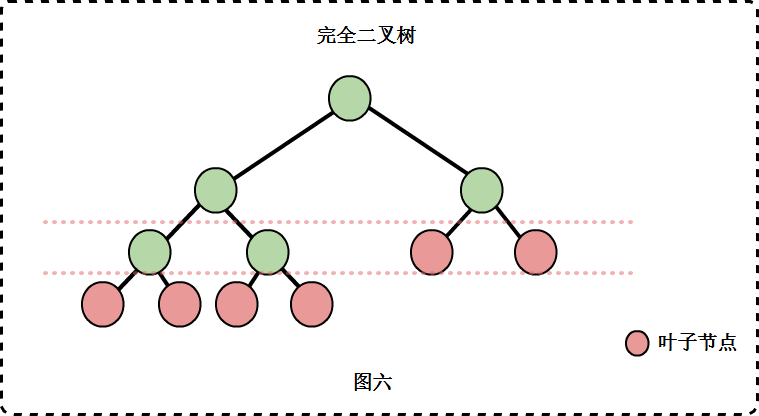
满二叉树要满足两个条件：

1. **所有的节点都同时具有左子树和右子树。**
2. **所有的叶子节点都在同一层上。**

在同样深度的二叉树中，满二叉树的节点数目是最多的，叶子数也是最多的。  


**完全二叉树**

在一棵二叉树中，只有最下两层的度可以小于2，并且最下一层的叶子节点集中出现在靠左的若干位置上。  
或者这样定义：对一棵具有n个节点的二叉树按层序从左到右编序，二叉树树某个节点的编序与同样位置的满二叉树节点的编序相同如果所有节点都满足这个条件，则二叉树为完全二叉树。  
**从定义可以看出： 满二叉树一定是完全二叉树；完全二叉树不一定是满二叉树。**

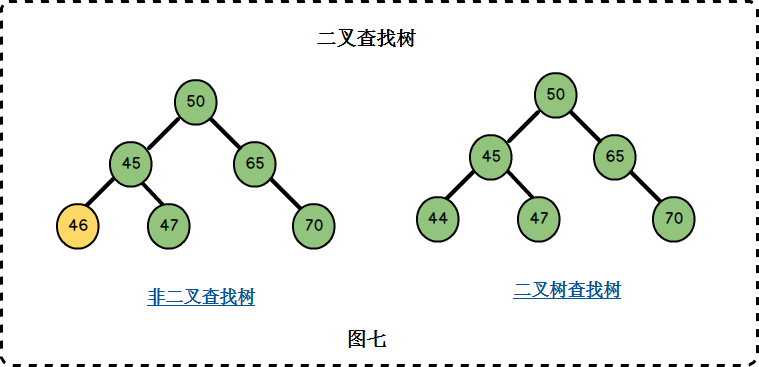


**二叉查找树**

二叉排序树也称为二叉搜索树或二叉排序树。二叉排序树的节点包含键值key。二叉排序树或者是一棵空树，否则要求：

1. 若它的左子树不为空，那么左子树上所有节点的key都小于根节点的key
2. 若它的右子树不为空，那么右子树上所有节点的key都大于根节点的key
3. 它的左右子树也分别为二叉排序树

根据定义，二叉查找树中没有重复key的节点。



**在实际的应用中，二叉排序树的应用比较多，我们后面要讲的AVL树本身也是一种二叉排序树。**

**2.3 二叉树的性质**

**性质一：在二叉树的第i层上至多有2^(i-1)个节点(i>=1)**

证明：利用数学归纳法进行证明

1. 当i==1时，第1层节点数目为2^(i-1) = 2^(1-1) = 2^0 = 1。显然成立，此时二叉树只有根节点。
2. 假设i>1时，第i层的节点数目为2^(i-1)。  
   根据假设，只需证明第i+1层节点数为2^i 即可。  
   由于二叉树每个节点最多有两个孩子，故第(i+1)层上的节点数最多是第i层的两倍。  
   即：第i+1层上节点数最多为: 2\* 2^(i-1) = 2 ^ i  
   故假设成立，命题得证。

**性质二：深度为k的二叉树至多有2^k-1个节点**

证明：二叉树节点数最多时，每层的节点树都必须最多。  
根据性质一，深度为k的二叉树的节点数最多为: 2^0 + 2^1 +....+2^(k-1) = 2 ^ k -1

**性质三：对任何一棵二叉树T，如果终端节点数为n0，度为2的节点数为n2 ，那么 n0 = n2 +1**

证明：二叉树节点度数最大为2，则 ： n = n0 + n1 + n2 (等式一)  
从孩子个数角度出发： 度为0的节点没有孩子， 度为1的节点没有1个孩子，度为2的节点有2个孩子，孩子总数为 n0*0 + n1*1 +n2 *2 = n1+2*n2；树的所有节点中，只有根不是任何节点的孩 子，因此有 n -1 = n1 + 2\* n2 ,即 n = n1 + 2\* n2 + 1. （等式二）  
由等式一等式而可以推出 n0 = n2 +1

**性质四： 具有n个节点的完全二叉树的高度为至少为log2(n+1)**

证明：高度为h的二叉树最多有2{h}–1个结点。反之，对于包含n个节点的二叉树的高度至少为log2(n+1)。

**性质五：如果对一棵有n个节点的完全二叉树的节点按层序编号（从第一层开始到最下一层，每一层从左到右编号），对任一节点i有：**

1. 如果i=1 ，则节点为根节点，没有双亲。
2. 如果2 \* i > n ，则节点i没有左孩子 ；否则其左孩子节点为2\*i . （n为节点总数）
3. 如果2 \* i+1>n ，则节点i没有右孩子；否则其右孩子节点为2\*1+1

**3. 二叉查找树**

二叉查找树的定义我们已经知道。要维护二叉查找树的特性，比较复杂的是删除节点操作，我们将进行重点的解析。不过我们先来看看二叉查找树的节点结构定义与类定义。

**3.1 节点结构**

//二叉查找树的节点结构

template <typename T>

struct BSNode

{

BSNode(T t)

: value(t), lchild(nullptr), rchild(nullptr){}

BSNode() = default;

T value;

BSNode<T>\* lchild;

BSNode<T>\* rchild;

BSNode<T>\* parent;

};

1. value:节点的值，也即是上文的key，类型由模板参数决定
2. lchild :指向节点的左孩子
3. rchild:指向节点的右孩子
4. parent: 指向节点的双亲

**3.2 二叉查找树的抽象数据结构**

//二叉查找树类

template <typename T>

class BSTree

{

public:

BSTree();

~BSTree();

void preOrder(); //前序遍历二叉树

void inOrder(); //中序遍历二叉树

void postOrder(); //后序遍历二叉树

void layerOrder(); //层次遍历二叉树

BSNode<T>\* search\_recursion(T key); //递归地进行查找

BSNode<T>\* search\_Iterator(T key); //迭代地进行查找

T search\_minimun(); //查找最小元素

T search\_maximum(); //查找最大元素

BSNode<T>\* successor (BSNode<T>\* x); //查找指定节点的后继节点

BSNode<T>\* predecessor(BSNode<T>\* x); //查找指定节点的前驱节点

void insert(T key); //插入指定值节点

void remove(T key); //删除指定值节点

void destory(); //销毁二叉树

void print(); //打印二叉树

private:

BSNode<T>\* root; //根节点

private:

BSNode<T>\* search(BSNode<T>\* & p, T key);

void remove(BSNode<T>\* p, T key);

void preOrder(BSNode<T>\* p);

void inOrder(BSNode<T>\* p);

void postOrder(BSNode<T>\* p);

T search\_minimun(BSNode<T>\* p);

T search\_maximum(BSNode<T>\* p);

void destory(BSNode<T>\* &p);

};

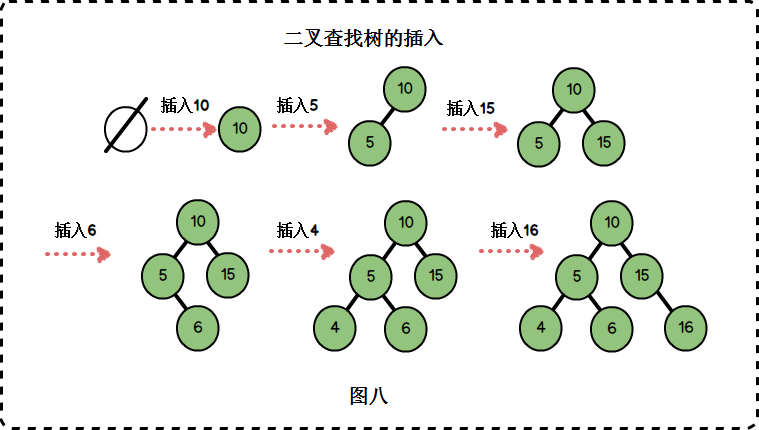
这里我们定义了二叉排序树的类型BSTree。它包含了：

1. BSTree的根节点root，这是唯一的数据成员
2. 操作的外部接口与内部实现接口。例如 preOrder()为提供给用户使用的接口，接口声明为public；而preOrder(AVLTreeNode\* pnode)是类内部为了递归操作所使用的接口，接口声明为private。

提供的其他接口都有相应的备注说明。

**3.3 插入新节点**

假设我们要为数组 a[] = {10 ， 5 ， 15 ， 6 ， 4 ， 16 }构建一个二叉排序树，我们按顺序逐个插入元素。



插入过程是这样的：

1. 如果是空树，则创建一个新节点，新节点作为根，因此以元素10构建的节点为该二叉查找树的根。
2. 插入5，5比10小，与10的左孩子节点进行比较，10的左孩子节点为空，进行插入。
3. 插入15，15比10大，与10的右孩子节点进行比较，10的右孩子节点为空，进行插入。
4. 插入6，6比10小，与10的左孩子节点5比较；6比5大，与5的右孩子节点进行比较，5的右孩子为空，进行插入。
5. 插入4，4比10小，与10的左孩子节点5比较；4比5小，与5的左孩子节点进行比较，5的左孩子为空，进行插入。
6. 插入16，16比10大，与10的右孩子节点15比较；16比15大，与15的右孩子节点进行比较，15的右孩子为空，进行插入。

从这个过程我们可以总结出插入新元素的步骤：

1. 寻找元素合适的插入位置：**新元素与当前结点进行比较，若值大于当前结点，则从右子树进行寻找；否则从左子树进行寻找**.
2. **找到插入位置之后，以元素的值构建新节点，插入二叉排序树中**

该过程的实现代码：

/\*插入函数\*/

template <typename T>

void BSTree<T>::insert(T key)

{

BSNode<T>\* pparent = nullptr;

BSNode<T>\* pnode = root;

while (pnode != nullptr) //寻找合适的插入位置

{

pparent = pnode;

if (key > pnode->value)

pnode = pnode->rchild;

else if (key < pnode->value)

pnode = pnode->lchild;

else

break;

}

pnode = new BSNode<T>(key); //以元素的值构建新节点

if (pparent == nullptr) //如果是空树

{

root = pnode; //则新节点为根

}

else

{

if (key > pparent->value)

{

pparent->rchild = pnode;//否则新节点为其父节点的左孩

}

else

pparent->lchild = pnode; //或右孩

}

pnode->parent = pparent; //指明新节点的父节点

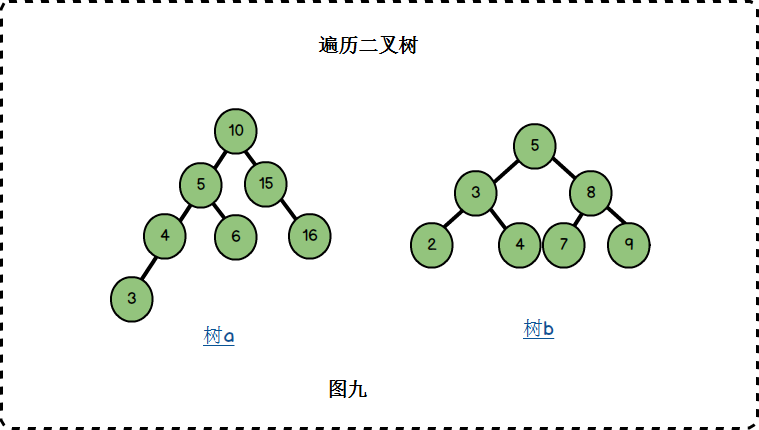
};

将构建出来的新节点插入二叉排序树时，需要修改链接指针的指向。

**3.2 遍历平衡二叉树**

遍历平衡二叉树，就是以某种方式逐个“访问”二叉树的每一个节点。“访问”是指对节点的进行某种操作，例如输出节点的值。  
平衡二叉树是有序树，严格区分左子树与右子树，如果规定左子树先于右子树的次序，我们有三种方式遍历二叉树：

1. 前序遍历
2. 中序遍历
3. 后序遍历

我们以如图的两棵二叉排序树进行遍历的算法演示。  


**前序遍历**

若二叉树为空，则空操作返回，否则先访问根节点，然后前序遍历左子树，再前序遍历右子树。（简记为：VLR）

/\*前序遍历算法\*/

template <typename T>

void BSTree<T>::preOrder()

{

preOrder(root);

};

template <typename T>

void BSTree<T>::preOrder(BSNode<T> \*p)

{

if (p != nullptr)

{

cout << p->value << endl;

preOrder(p->lchild);

preOrder(p->rchild);

}

};

前序遍历树a：10 5 4 3 6 15 16  
前序遍历树b：5 3 2 4 8 7 9

**中序遍历**

若二叉树为空，则空操作返回，否则从根节点开始，中序遍历根节点的左子树，然后访问根节点，最后中序遍历右子树。（简记为：LVR）

/\*中序遍历算法\*/

template <typename T>

void BSTree<T>::inOrder()

{

inOrder(root);

};

template<typename T>

void BSTree<T>::inOrder(BSNode<T>\* p)

{

if (p != nullptr)

{

inOrder(p->lchild);

cout << p->value << endl;

inOrder(p->rchild);

}

};

前序遍历树a：3 4 5 6 10 15 16  
前序遍历树b：2 3 4 5 7 8 9

二叉排序树的中序遍历刚好输出一个非递减的有序序列。

**后序遍历**

若树为空，则返回空操作，否则从左到右先叶子后节点的方式遍历访问左右子树，左右子树都访问结束，才访问根节点。（简称LRV）

/\*后序遍历算法\*/

template <typename T>

void BSTree<T>::postOrder()

{

postOrder(root);

};

template <typename T>

void BSTree<T>::postOrder(BSNode<T>\* p)

{

if (p != nullptr)

{

postOrder(p->lchild);

postOrder(p->rchild);

cout << p->value<<endl;

}

};

后序遍历树a：3 4 6 5 16 15 10  
后序遍历树b：2 4 3 7 9 8 5

**3.2 前驱与后继**

对于一棵二叉排序树，中序遍历时刚好可以输出一个非递减的序列。例如前序遍历图九树a：3 4 5 6 10 15 16，则可称：

* 4是5 前驱节点，6是5的后继节点
* 6是10的前驱节点，15是10的后继节点

**一个节点的前驱节点有3种情况：**

1. **它有左子树，则左子树根节点为其前驱节点**
2. **它没有左子树，且它本身为右子树，则其父节点为其前驱节点**
3. **它没有左子树，且它本身为左子树，则它的前驱节点为“第一个拥有右子树的父节点”**

/\*寻找其前驱节点\*/

template <typename T>

BSNode<T>\* BSTree<T>::predecessor(BSNode<T>\* pnode)

{

if (pnode->lchild != nullptr)

{

pnode = pnode->lchild;

while (pnode->rchild != nullptr)

{

pnode = pnode->rchild;

}

return pnode;

}

BSNode<T>\* pparent = pnode->parent;

while (pparent != nullptr && pparent->lchild == pnode)//如果进入循环，则是第三种情况；否则为第二种情况

{

pnode = pparent;

pparent = pparent->parent;

}

return pparent;

};

**同样的，一个节点的后继节点也有三种情况：**

1. **它有右子树；则其后继节点为其右子树的最左节点**
2. **它没有右子树，但它本身是一个左孩子，则后继节点为它的双亲**
3. **它没有右子树，但它本身是一个右孩子，则其后继节点为“具有左孩子的最近父节点”**

/\*寻找其后继节点\*/

template <typename T>

BSNode<T>\* BSTree<T>::successor(BSNode<T>\* pnode)

{

if (pnode->rchild != nullptr)

{

pnode = pnode->rchild;

while (pnode->lchild != nullptr)

{

pnode = pnode->lchild;

}

return pnode;

}

BSNode<T>\* pparent = pnode->parent;

while (pparent!=nullptr&& pparent->rchild == pnode)

{

pnode = pparent;

pparent = pparent->parent;

}

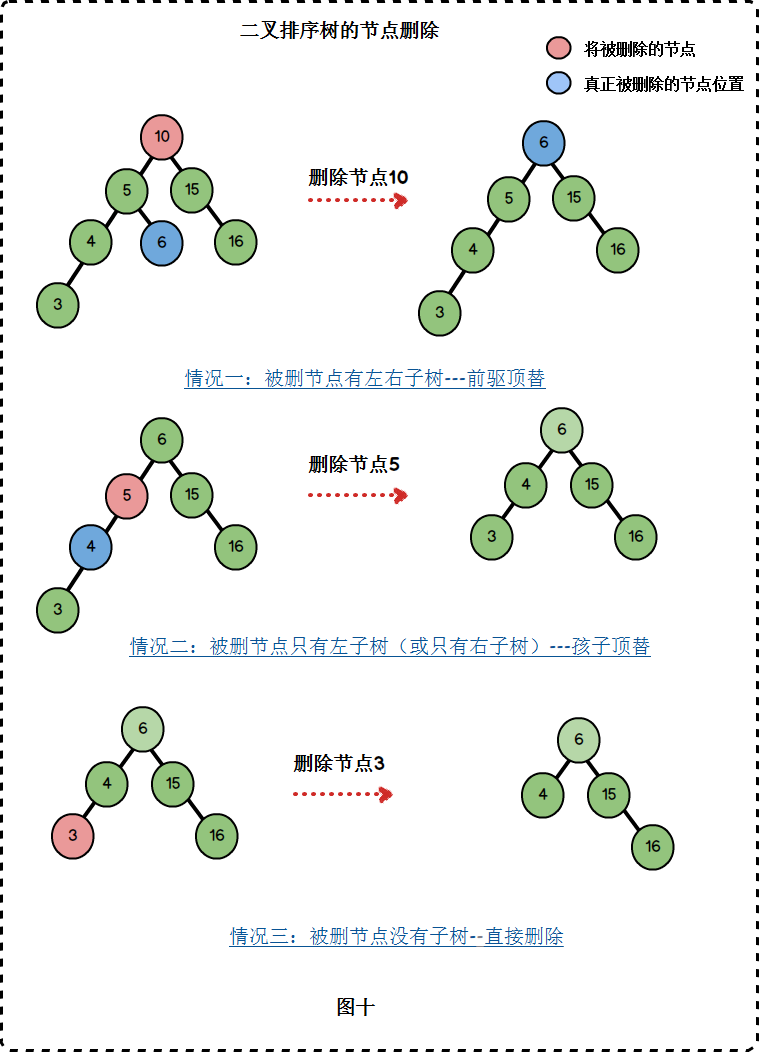
return pparent;

};

**3.3 删除节点**

删除二叉排序树的某个节点有三种情况：

1. 被删除节点同时有左子树与右子树。
2. 被删除节点只有左子树或只有右子树。
3. 被删除节点没有子树。

对于第一种情况，我们的处理方式是将前驱节点的值保存在当前结点，继而删除前驱节点。  
对于第二种情况，我们直接用子树替换被删节点。  
对于第三种情况，我们可以直接删除节点。  


删除节点的代码：

/\*删除指定节点\*/

template <typename T>

void BSTree<T>::remove(T key)

{

remove(root, key);

};

/\*删除指定节点\*/

/\*内部使用函数\*/

template <typename T>

void BSTree<T>::remove(BSNode<T>\* pnode, T key)

{

if (pnode != nullptr)

{

if (pnode->value == key)

{

BSNode<T>\* pdel=nullptr;

if (pnode->lchild == nullptr || pnode->rchild == nullptr)

pdel = pnode; //情况二、三：被删节点只有左子树或右子树，或没有孩子

else

pdel = predecessor(pnode); //情况一：被删节点同时有左右子树，则删除该节点的前驱

//此时，被删节点只有一个孩子（或没有孩子）.保存该孩子指针

BSNode<T>\* pchild=nullptr;

if (pdel->lchild != nullptr)

pchild = pdel->lchild;

else

pchild = pdel->rchild;

//让孩子指向被删除节点的父节点

if (pchild != nullptr)

pchild->parent = pdel->parent;

//如果要删除的节点是头节点，注意更改root的值

if (pdel->parent == nullptr)

root = pchild;

//如果要删除的节点不是头节点，要注意更改它的双亲节点指向新的孩子节点

else if (pdel->parent->lchild==pdel)

{

pdel->parent->lchild = pchild;

}

else

{

pdel->parent->rchild = pchild;

}

if (pnode->value != pdel->value)

pnode->value = pdel->value;

delete pdel;

}

//进行递归删除

else if (key > pnode->value)

{

remove(pnode->rchild, key);

}

else remove(pnode->lchild, key);

}

};

**3.4 查找元素**

我们可以递归或非递归地进行元素的查找。元素的查找过程与元素的插入过程一致，也是在不断地与当前结点进行比较，若值比当前节点的值大，则在右子树进行查找，若值比当前节点的值小，则在左子树进行查找，可以看到这是一个很适合递归操作的过程。

而由于二叉排序树这种左小右大的节点特征，也很容易进行非递归查找。

/\*查找指定元素的节点（非递归）\*/

template <typename T>

BSNode<T>\* BSTree<T>::search\_Iterator(T key)

{

BSNode<T> \* pnode = root;

while (pnode != nullptr)

{

if (key == pnode->value) //找到

return pnode;

if (key > pnode->value) //关键字比节点值大，在节点右子树查找

pnode = pnode->rchild;

else

pnode = pnode->lchild; //关键字比节点值小，在节点左子树查找

}

return nullptr;

};

/\*查找指定元素的节点（递归）\*/

template <typename T>

BSNode<T>\* BSTree<T>::search\_recursion(T key)

{

return search(root, key);

};

/\*private:search()\*/

/\*递归查找的类内部实现\*/

template <typename T>

BSNode<T>\* BSTree<T>::search(BSNode<T>\* & pnode, T key)

{

if (pnode == nullptr)

return nullptr;

if (pnode->value == key)

return pnode;

//cout << "-->" << pnode->value << endl; //可以输出查找路径

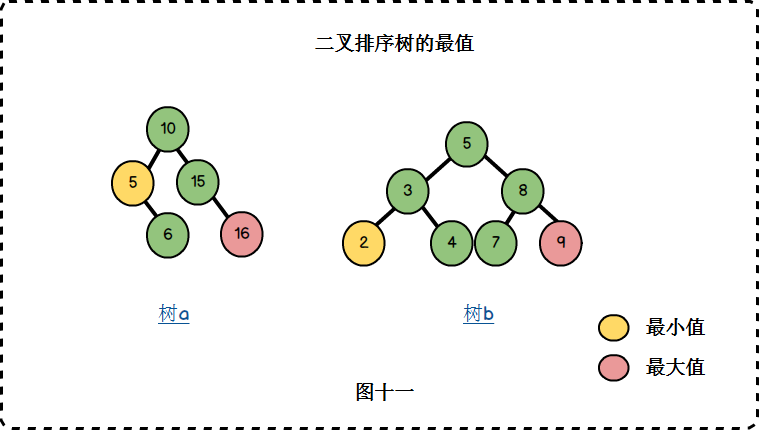
if (key > pnode->value)

return search(pnode->rchild, key);

return search(pnode->lchild, key);

};

**3.5 查找最值元素**

二叉排序树的最小值位于其最左节点上；最大值位于其最右节点上：  


/\*寻找最小元素\*/

template <typename T>

T BSTree<T>::search\_minimun()

{

return search\_minimun(root);

};

template <typename T>

T BSTree<T>::search\_minimun(BSNode<T>\* p)

{

if (p->lchild != nullptr)

return search\_minimun(p->lchild);

return p->value;

};

/\*寻找最大元素\*/

template <typename T>

T BSTree<T>::search\_maximum()

{

return search\_maximum(root);

};

template <typename T>

T BSTree<T>::search\_maximum(BSNode<T>\*p)

{

if (p->rchild != nullptr)

return search\_maximum(p->rchild);

return p->value;

};

**3.6 销毁二叉树**

使用后序遍历递归销毁二叉树

/\*销毁二叉树\*/

template<typename T>

void BSTree<T>::destory()

{

destory(root);

};

template <typename T>

void BSTree<T>::destory(BSNode<T>\* &p)

{

if (p != nullptr)

{

if (p->lchild != nullptr)

destory(p->lchild);

if (p->rchild != nullptr)

destory(p->rchild);

delete p;

p = nullptr;

}

};

**3.7测试代码**

int main()

{

BSTree<int> t;

t.insert(62);

t.insert(58);

t.insert(47);

t.insert(51);

t.insert(35);

t.insert(37);

t.insert(88);

t.insert(73);

t.insert(99);

t.insert(93);

t.insert(95);

cout << endl << "中序遍历：" << endl;

t.inOrder();

cout << "最大元素:" << t.search\_maximum() << endl;

cout << "最小元素:" << t.search\_minimun() << endl;

cout << "删除元素99" << endl;

t.remove(99);

cout << "最大元素:" << t.search\_maximum() << endl;

t.destory();

getchar();

return 0;

}

运行结果：

|  |
| --- |
| 中序遍历：  35  37  47  51  58  62  73  88  93  95  99  最大元素:99  最小元素:35  删除元素99  最大元素:95 |

**4. 二叉查找树完整代码**

在github上存放了二叉排序树的vs项目工程。这是二叉排序树的源代码：  
<https://github.com/huanzheWu/Data-Structure/blob/master/BSTree/BSTree/BSTree.h>