课堂练习补充：

**二叉查找树查找的算法：**

非递归：

template <class Type>

bool BinarySearchTree<Type>::find(const Type &x)

{ node \*p; p=root;

While (p)

{ if (x==p->data) return true;

If (x<p->data) p=p->left;

else p=p->right;

}

return flase;

}

递归：

template <class Type>

bool BinarySearchTree<Type>::find(node \*t, const Type &x)

{ if (!t) return false;

if (x==t->data) return true;

if (x<t->data)

return find(t->left, x);

else

return find(t->right, x);

}

**二叉查找树插入的算法：**

非递归：

template <class Type>

Void BinarySearchTree<Type>::insert(const Type &x)

{ node \*p, \*parent; int flag;

p=root; parent = NULL;

While (p)

{ if (x==p->data) return; //待插入元素已经在查找树中了

If (x<p->data)

{ parent = p; flag = 1; p=p->left;}

else

{ parent = p; flag = 2; p=p->right;}

}

if (!parent) { root = new node(x); reurn;}

if (flag==1)

parent->left = new node(x);

else

parent->right = new node(x);

}

template <class Type>

Void BinarySearchTree<Type>::insert(const Type &x)

{ node \*p;

If (!root) { root = new node(x); return; } //根原本为空

p=root;

While (true)

{ if (x==p->data) return; //待插入元素已经在查找树中了

If (x<p->data)

{ if (!p->left) { p->left = new node(x); return; }

else p=p->left;

}

else

{ if (!p->right) { p->right = new node(x); return; }

else p=p->right;

}

}

}

引用参数例子

void main()

{ node<int> \*p = new node(1);

f(p->left);

g(p->right);

if (p->left) cout<<”left child is linked!”<<endl;

if (p->right) cout<<”right child is linked!”<<endl;

}

void f(node \*t) { t=new node(10); }

void f(node \*&t) { t=new node(20);}

屏幕上会显示：

right child is linked!

递归：

template <class Type>

Void BinarySearchTree<Type>::insert(node \* **&**t; const Type &x)

{

if (!t) { t = new node(x); return; }

if (x==t->data) return;

if (x<t->data)

insert(t->left, x);

else

insert(t->right, x);

}

二叉查找树删除算法：

非递归：

1. 假设待删除结点为叶子结点

template <class Type>

void BinarySearchTree<Type>::remove1( const Type & x)

{

node \*p, \*parent; int flag;

p = root; parent = NULL;

while (p)

{ if (x==p->data) break;

if (x<p->data) { parent = p; flag =1; p=p->left;}

else { parent = p; flag =2; p=p->right; }

}

if (!p) return;

if (!parent) { delete root; root = NULL; return;}

if (flag==1)

{ parent->left = NULL; delete p;}

else

{ parent->right = NULL; delete p;}

}

非递归：

1. 假设待删除结点为只有一个孩子的结点//**其实可以是叶子结点**）

template <class Type>

void BinarySearchTree<Type>::remove2( const Type & x )

{

node \*p, \*parent; int flag;

p = root; parent = NULL;

while (p)

{ if (x==p->data) break;

if (x<p->data) { parent = p; flag =1; p=p->left;}

else { parent = p; flag =2; p=p->right; }

}

if (!p) return;

if (!parent) { root = (root->left)? root->left: root->right; delete p; return; }

if (flag==1)

{ parent->left = (p->left)? p->left: p->right delete p;}

else

{ parent->right = (p->left)? p->left: p->right delete p;}

}

非递归：

1. 假设待删除结点为有两个孩子的结点

template <class Type>

void BinarySearchTree<Type>::remove3( const Type & x )

{

node \*p, \*parent, \*r; int flag;

p = root; parent = NULL;

while (p)

{ if (x==p->data) break;

if (x<p->data) p=p->left;

else p=p->right;

}

if (!p) return;

r = p;

parent = p; flag = 1; p=p->left; //在左子树上找替身结点即最大结点或者说最右侧结点

while (p->right) { parent = p; flag = 2; p=p->right; }

r->data = p->data; //将替身结点的元素值拷贝到待删除结点r上，任务改为删替身结点

//替身结点必无右子

if (flag==1)

{ parent->left = p->left; delete p; }

else

{ parent->right = p->left; delete p;}

}

非递归：

合并以上1、2、3三种情况

template <class Type>

void BinarySearchTree<Type>::remove( const Type & x )

{

node \*p, \*parent; int flag;

p = root; parent = NULL;

while (p)

{ if (x==p->data) break;

if (x<p->data) { parent = p; flag =1; p=p->left;}

else { parent = p; flag =2; p=p->right; }

}

if (!p) return;

if ( (!p->left)||(!p->right)) //待删除结点为叶子或者只有一个孩子的结点

{

if (!parent) { root = (root->left)? root->left: root->right; delete p; return; }

if (flag==1)

{ parent->left = (p->left)? p->left: p->right delete p;}

else

{ parent->right = (p->left)? p->left: p->right delete p;}

}

if ( (p->left)&&( p->right))//待删除结点有两个孩子

{

r = p;

parent = p; flag = 1; p=p->left; //在左子树上找替身结点即最大结点

while (p->right) { parent = p; flag = 2; p=p->right; }

r->data = p->data; //将替身结点的元素值拷贝到待删除结点r上，

//任务改为删替身结点, 替身结点必无右子

if (flag==1)

{ parent->left = p->left; delete p; }

else

{ parent->right = p->left; delete p;}

}

}

递归：

1. 待删结点为叶子

template <class Type>

void BinarySearchTree<Type>::remove1( node \* &t, const Type & x )

{

if (!t) return;

if (x==t->data) { delete t; t=NULL; return;} //t作为赋值语句左值和右值意义不一样

if (x<t->data) remove(t->left, x);

else remove(t->right, x);

}

递归：

1. 待删结点有一个孩子或者只有一个孩子的结点

template <class Type>

void BinarySearchTree<Type>::remove2( node \* &t, const Type & x )

{

node \*p;

if (!t) return;

if (x==t->data) { p=t; t=(t->left)?t->left: t->right; delete p; return;}

if (x<t->data) remove(t->left, x);

else remove(t->right, x);

}

递归：

1. 待删结点有两个孩子

template <class Type>

void BinarySearchTree<Type>::remove3( node \* &t, const Type & x )

{

node \*p;

if (!t) return;

if (x==t->data)

{ //t的左子树上找替身结点即最大结点

p=t->left;

while (p->right) p=p->right;

t->data = p->data;

**remove2**(t->left, x); //问题转换为情况1、2，注意起始地将t->left给引用参数

}

if (x<t->data) remove(t->left, x);

else remove(t->right, x);

}

递归：

合并以上1、2、3三种情况

template <class Type>

void BinarySearchTree<Type>::remove ( node \* &t, const Type & x )

{

node \*p;

if (!t) return;

if (x==t->data)

{ if ((!t->left)||(!t->right))// 待删结点有一个孩子或者只有一个孩子的结点

{ p=t; t=(t->left)?t->left: t->right; delete p; return;}

if ((t->left)&&(t->right))// 待删结点有两个孩子

{ //t的左子树上找替身结点即最大结点

p=t->left;

while (p->right) p=p->right;

t->data = p->data;

remove(t->left, x); //问题转换为情况1、2，注意起始地将t->left给引用参数

}

}

}