树

2019年4月13日

15:48

**树**

树可以用几种方式定义。定义树的一种自然的方式是递归的方式。一棵树是一些节点的集合。这个集合可以是空集；若不是空集，则树由称作根的节点r以及0个或多个非空的(子)树T1,T2,···，Tk组成，这些子树中每一棵的根都被来自根r的一条有向的边所连结。每一棵子树的根叫作根r的儿子，而r是每一棵子树的根的父亲。

每一个节点可以有任意多个儿子，也可能是0个儿子。没有儿子的节点称为树叶(leaf);具有相同父亲的节点为兄弟(siblings)。在一棵树中从根到每个节点恰好存在一条路径。对任意节点ni，ni的深度为从根到ni的唯一的路径长。ni的高是从ni到一片树叶的最长路径的长。

**树的实现**

实现树的一种方法可以实在每一个节点除数据外还要由一些链，使得该节点有一个链指向它。通常将每个节点的所有儿子都放在树节点的链表中。

**二叉树**

二叉树是一棵树，其中每个节点都不能有多于两个的儿子。二叉树的一个性质是一棵平均二叉树的深度要比节点个数小的多。分析表明，其平均深度为O(√N)。

**二叉树的实现**

因为一个二叉树最多有两个节点，所以我们可以保存直接链接到它们的链。在声名中，节点就是由element(元素)的信息加上两个到其他节点的引用组成的结构

class BinaryNode

{

Object element;//当前节点的数据

BinaryNode left;

BinaryNode right;

}

二叉查找树

对于树中的每个节点x，它的左子树中所有项的值小于x中的项，而它的右子树中所有项的值大于x中的项。

public class BinarySearchTree<AnyType extends Comparable<? super AnyType>>

{

private static class BinaryNode<AnyType>

{

BinaryNode( AnyType theElement )

{ this( theElement, null, null ); }

BinaryNode( AnyType theElement, BinaryNode<AnyType> lt, BinaryNode<AnyType> rt )

{ element = theElement; left = lt; right = rt; }

AnyType element;

BinaryNode<AnyType> left;

BinaryNode<AnyType> right;

}

private BinaryNode<AnyType> root;

public BinarySearchTree( )

{ root = null; }

public void makeEmpty( )

{ root = null; }

public boolean isEmpty( )

{ return root == null; }

public boolean contains( AnyType x )

{ return contains( x, root ); }

public AnyType findMin( )

{

if( isEmpty( ) )

throw new UnderflowException( );

return findMin( root ).element;

}

public AnyType findMax( )

{

if( isEmpty( ) )

throw new UnderflowException( );

return findMax( root ).element;

}

public void insert( AnyType x )

{ root = insert( x, root ); }

public void remove( AnyType x )

{ root = remove( x, root ); }

public void printTree( )

{

if( isEmpty( ) )

System.out.println( "Empty tree" );

Else

printTree( root );

}

private boolean contains( AnyType x, BinaryNode<AnyType> t )

{

if( t == null )

return false;

int compareResult = x.compareTo( t.element );

if( compareResult < 0 )

return contains( x, t.left );

else if( compareResult > 0 )

return contains( x, right );

else

return true;

}

private BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )

{

if( t == null )

return null;

else if( t.left == null )

return t;

return findMin( t.left );

}

private BinaryNode<AnyType> findMax( BianryNode<AnyType> t )

{

if( t != null )

while( t.right != null )

t = t.right;

return t;

}

private BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )

{

if( t == null )

return new BinaryNode<>( x, null, null );

int compareResult = x.compareTo( t.element );

if( compareResult < 0 )

t.left = insert( x, t.left );

else if( compareResult > 0 )

t.right = insert( x, t.right);

esle ;

return t;

}

private BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )

{

if( t == null )

return t;

int cmopareResult = x.compareTo( t.element );

if( compareResult < 0 )

t.left = remove( x, t.left );

else if( compareResult > 0 )

t.right = remove( x, t.right );

else if( t.left != null && t.right != null )

{

t.element = findMin( t.right ).element;

t.right = remove( t.element, t.right);

}

else

t = ( t.left != null ) ? t.left : t.right;

return t;

}

private void printTree( BinaryNode<AnyType> t )

{

if( t != null )

{

printTree( t.left );

System.out.println( t.element );

printTree( t.right );

}

}

}

**AVL树**

AVL树是带有平衡条件的二叉查找树。这个平衡条件必须要容易保持，而且它保证树的深度须是O(log N)。

一棵AVL树是其每个节点的左子树和右子树的高度最多差1的二叉查找树。在高度为h的AVL树中，最少节点数S(h) = S(h-1) + S(h-2) + 1。

出去可能的插入外，所有的树操作都可以以时间O(logN)执行。当进行插入操作时我们需要更新通向根节点路径上的那些节点的所有平衡信息，而插入一个节点可能破坏AVL树的特性。如果发生这种情况，那么就要在考虑这一步插入完成之前恢复平衡的性质。事实上，这总可以通过对树进行简单的修正来做到，我们称其为旋转。

/\*\*

Avl树类

\*/

public static class AvlTree<AnyType extends Comparable<? super AnyType> >

{

/\*\*

Avl节点类，含有单值实例函数和带有节点的实例函数

\*/

public static class AvlNode<AnyType>

{

AvlNode( AnyType theElement )

{ this( theElement, null, null ); }

AvlNode( AnyType theElement, AvlNode<AnyType> lt, AvlNode<AnyType> rt )

{ element = theEle { element = theElement; left = lt; right = rt; height = 0; }

AnyType element;

AvlNode<AnyType> left;

AvlNode<AnyType> right;

int height;

}

/\*\*

计算节点的高度函数

\*/

private int height( AvlNode<AnyType> t )

{ return t == null ? -1 : t.height; }

/\*\*

插入函数

\*/

private AvlNode<AnyType> insert( AnyType x, AvlNode<AnyType> t )

{

if( t == null )//如果节点为空，则插入动作变为实例化一个新节点，插入函数结束

return new AvlNode<>( x, null, null );

//比较要插入的值与当前节点的值，计算要插入的位置

int compareResult = x.compareTo( t.element );

if( compareResult < 0 )

t.left = insert( x, t.left );

else if( compareResult > 0 )

t.right = insert( x, t.right );

else

;

return balance( t );

}

private static finall int ALLOWED\_IMBALANCE = 1;//平衡允许的高度差

/\*\*

平衡函数

\*/

private AvlNode<AnyType> balance( AvlNode<AnyType> t )

{

if( t == null )//空节点无需平衡

return t;

if( height( t.left ) - height( t.right ) > ALLOWED\_IMBALANCE )//判读左儿子和右儿子谁高，选择选择方向

if( height( t.left.left ) >= height( t.left.right ) )//判断左子树和右子树谁高，选择旋转方式

t = rotateWithLeftChild( t );

else

t = doubleWithLeftChild( t );

else

if( height( t.right ) - height( t.left ) > ALLOWED\_IMBALANCE )

if( height( t.right.right ) >= height( t.right.left )

t = rotateWithRightChild( t );

else

t = doubleWithRightChild( t );

t.height = Math.max( height( t.left ), height( t.right ) ) + 1;//重新计算节点的高度

return t;

}

priavate AvlNode<AnyType> rotateWithLeftChild( AvlNode<AnyType> k2 )

{

AvlNode<AnyType> k1 = k2.left;

k2.left = k1.right;

k1.right = k2;

k2.height = Math.max( height( k2.left ), height( k2.right ) ) + 1;

k1.height = Math.max( height( k1.left ), k2.height ) + 1;

return k1;

}

private AvlNode<AnyType> doubleWithLeftChild( AvlNode<AnyType> k3 )

{

k3.left = rotateWithRightChild( k3.left );

return rotateWithLeftChild( k3 );

}

private AvlNode<AnyType> remove( AnyType x, AvlNode<AnyType> t )

{

if( t == null )

return t;

int compareResult = x.compareTo( t.element );

if( compareResult < 0 )

t.left = remove( x, t.left );

else if( compareResult > 0 )

t.right = remove( x, t.right );

else if( t.left != null && t.right != null )

{

t.element = findMin( t.right ).element;

t.right = remove( t.element, t.right );

}

else

t = ( t.left != null ) ? t.left : t.right;

return balance( t );

}

}