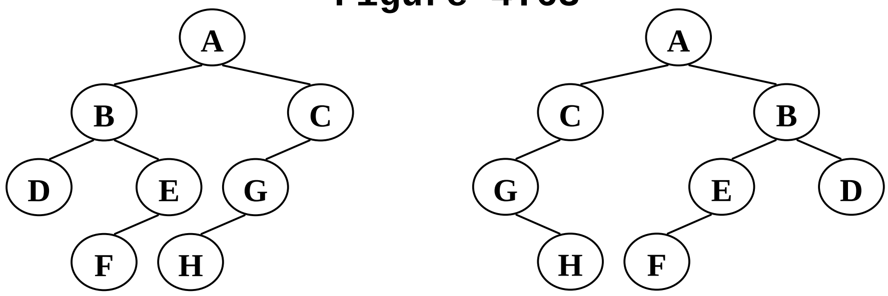
p.144 4.42

Two trees, T_1 and T_2, are *isomorphic* if T_1 can be transformed into T_2 by swapping left and right children of (some of the) nodes in T_1. For instance, the two trees in Figure 4.63 are isomorphic because they are the same if the children of A, B, and G, but not the other nodes, are swapped.

a. Give a polynomial time algorithm to decide if two trees are isomorphic.

*b. What is the running time of your program (there is a linear solution)?

Figure 4.63




```
typedef  struct  TreeNode  *PtrToNode;
typedef  PtrToNode  Tree;

struct  TreeNode  {
    ElementType  Element;
    Tree  Left;
    Tree  Right;
}

#define  True  1;
#define  False  0;

int  Isomorphic ( Tree T1, Tree T2 )
{   /* The algorithm decides if two trees are isomorphic by   */
    /* preorder traversal.   It is assumed that all the elements */
    /* of a tree are distinct. */

    /* Visit this node first */
        if  ( (T1==Null )&& (T2==Null) )  /* both empty */
                return  True;
        if  ( ((T1==Null)&&(T2!=Null)) || ((T1!=Null)&&(T2==Null)) )
                return  False;  /* one of them is empty */
        if  ( T1->Element != T2->Element )
                return  False;  /* roots are different */
    /* Otherwise:  T1->Element == T2->Element  -- this node is okay */
```

```
    /* Visit its children */
        if   ( ( T1->Left == Null )&&( T2->Left == Null ) ) /* both have no left subtree */
                 return   Isomorphic( T1->Right, T2->Right );
        if   ( ((T1->Left!=Null)&&(T2->Left!=Null))&&
              ((T1->Left->Element)==(T2->Left->Element)) )
    /* no need to swap the left and the right subtrees of T1 */
                 return  ( Isomorphic( T1->Left, T2->Left ) &&
                           Isomorphic( T1->Right, T2->Right ) );
        else     /* need to swap the left and the right subtrees of T1 */
                 return ( Isomorphic( T1->Left, T2->Right) &&
                           Isomorphic( T1->Right, T2->Left ) );
}
```

Running time = O(N) where N is the number of nodes in the smaller tree of T1 and T2

p.145   4.45

Since a binary search tree with *N* nodes has *N*+1 NULL pointers, half the space allocated in a binary search tree for pointer information is wasted. Suppose that if a node has a NULL left child, we make its left child point to its inorder predecessor, and if a node has a NULL right child, we make its right child point to its inorder successor. This is known as a *threaded tree* and the extra pointers are called *threads*.

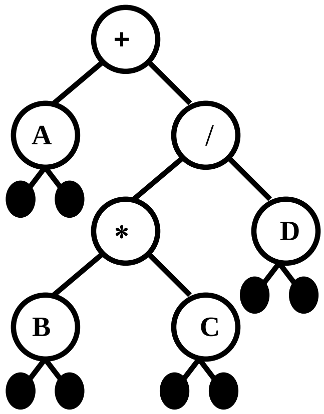
a. How can we distinguish threads from real children pointers?
b. Write routines to perform insertion and deletion into a tree threaded in the manner described above.
c. What is the advantage of using threaded trees?

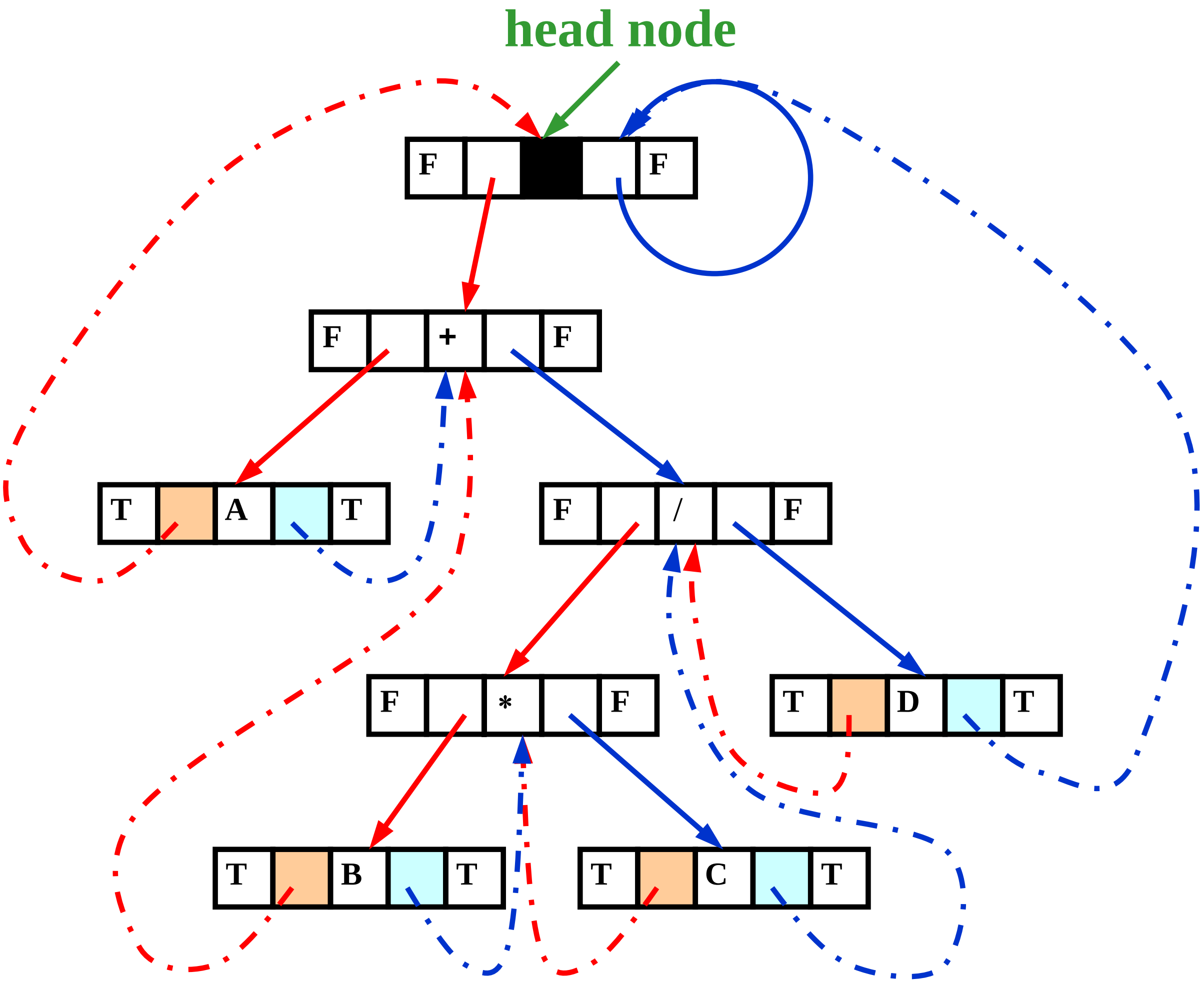
Answer:

a. To distinguish threads from real children pointers, we can add two additional fields to the node structure: Ltag and Rtag.

Ltag=0: Left points to the left child;
Ltag=1: Left points to the inorder predecessor;
Rtag=0: Right points to the right child;
Rtag=1: Right points to the inorder successor.

A general threaded tree





b. The routines:

```
typedef struct TreeNode *PtrToNode;
typedef PtrToNode Tree;
struct TreeNode {
    ElementType Element;
    Tree Left;
    Tree Right;
    short int Ltag;
    short int Rtag;
}

Tree Search( Tree T, Element Key )
{   /* Search a nonempty binary search tree for the Key     */
    /* If the node for the key is present, return NULL.     */
    /* Otherwise, return a parent to the position to insert */
        Tree Parent = T ; /* start from the head node */
        short int Tag = 0; /* Tag of parent – Tag of the head is always 0 */
        T = Parent->Left; /* T is the root */
        while ( !Tag ) { /*while T is indeed a child of Parent */
            if ( Key == T->Element ) /* found Key */
                return Null ;
            Parent = T; /* else move down */
            if ( Key < Parent->Element ) {
                Tag = Parent ->Ltag; T = Parent ->Left;
            }
            else {
                Tag= Parent ->Rtag; T = Parent ->Right;
            }
        } /* end while-loop */
        return Parent;
}

void Insert (Tree T, Element Key)
{   /* If the Key is in the tree, do nothing; otherwise add a new node */
        Tree Ptr, Temp;
        if ( T->Left == T ) { /* tree is empty */
            Ptr = Tree malloc (sizeof(TreeNode));
            if ( Ptr == Null ) FatalError("The memory is full ");
            T->Left = Ptr;
            Ptr->Element = Key;
            Ptr->Ltag = 1; Ptr->Rtag = 1;
            Ptr->Left = T ; Ptr->Right = T ;
            return ; /* create a one node tree and return */
        }
```

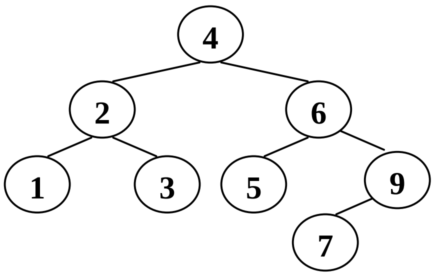
```
        Temp = Search( T,   Key );
        if ( Temp )   {   /*key is not in the tree */
            Ptr = Tree malloc (sizeof(TreeNode));
            if ( Ptr==Null )     FatalError("The memory is full ");
            Ptr->Element=Key;
            Ptr->Ltag=1;
            Ptr->Rtag=1;
            if   (Key<Temp->Element)
            {    /*Ptr is inserted as a left child of Temp*/
                Ptr->Left = Temp->Left;
                Ptr->Right = Temp;
                Temp->Left = Ptr;
                Temp->Ltag = 0;
            }    /*note: the predecessor of Temp has a right child */
            else
            {   /*Ptr is inserted as a right child of Temp*/
                Ptr->Left = Temp;
                Ptr->Right = Temp->Right;
                Temp->Right = Ptr;
                Temp->Rtag = 0;
            }     /*note: the successor of Temp has a left child*/
        }   /* end if ( Temp ) - insertion */
}

Tree Delete(Tree T, Element Key )
{      Tree   TmpCell;
       if ( T->Left == T )     return   Error( "Element not found" ); /* tree is empty */
       if ( T->Right == T )   T->Left = Delete( T->Left, Key ); /* if T is the head node */
       else   if ( Key < T->Element )   /* Go left */
                 T->Left = Delete( T->Left, Key );
              else   if ( Key > T->Element )   /* Go right */
                        T->Right = Delete( T->Right, Key );
                     else   /* Found element to be deleted */
                         if ( !T->Ltag && !T->Rtag ) {   /* Two children */
                             /* Replace with smallest in right subtree */
                             TmpCell = FindMin( T->Right );
                             T->Element = TmpCell->Element;
                             T->Right = Delete( T->Right, T->Element );   } /* End if */
                         else {   /* One or zero child */
                             TmpCell = T;
                             if ( T->Ltag && !T->Rtag ) {
                                 /* if T has a left thread and a right child */
                                 T = FindMin( T->Right );
                                 T->Left = TmpCell->Left; /* reset threads */
                                 T = TempCell->Right;
                             }
```

```
                else   if ( !T->Ltag && T->Rtag ) {
                         /* if T has a left child and a right thread */
                         T = FindMax( T->Left );
                         T->Right = TmpCell->Right; /* reset threads */
                         T = TmpCell->Left;
                     }
                     else   { /* if T is a leaf node */
                         if ( T->Right->Left == T ) { /* T is a left child */
                              T->Right->Left = T->left;
                              if ( T->Right->Left != T->Right )
                                   /* T's parent is not the head node */
                                   T->Right->Ltag = 1;
                              T = T->Left;
                         }
                         else { /* T is a right child */
                              T->Left->Right = T->Right;
                              T->Left->Rtag = 1;
                              T = T->Right;
                         }
                     } /*End if T is a leaf node */
                free( TmpCell );   }   /* End else 1 or 0 child */
     return   T;
}
```

c. By using threads, no stack is needed for preorder, inorder and postorder traversals.

p.141 4.16

Show the result of inserting 2, 1, 4, 5, 9, 3, 6, 7 into an initially empty AVL tree.



p.141 4.22

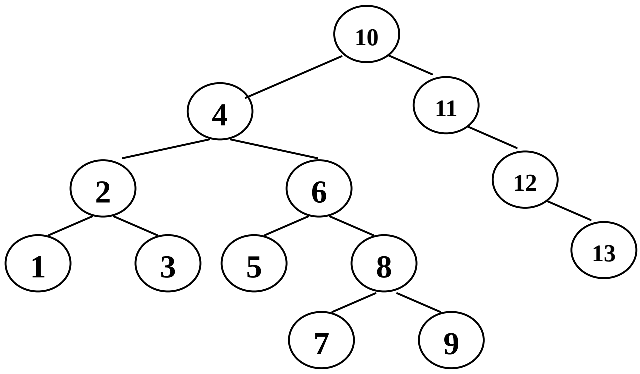
Write the functions to perform the double rotation without the inefficiency of doing two single rotations.

```
#ifndef   _AvlTree_H
#define   _AvlTree_H
struct   AvlNode;
typedef   struct   AvlNode   *Position;
typedef   struct   AvlNode   *AvlTree;
/* function declarations are omitted */
#endif   /* _AvlTree_H */
struct   AvlNode   {
     ElementType   Element;
     AvlTree   Left, Right;
     int   Height;
}

static Position   DoubleRotateWithLeft( Position   K3 )
{    /* Do the left—right double rotation.   K3 is the trouble finder. */
      Position   K1, K2;
      K1=K3->Left;       /* mark parent */
      K2=K1->Right;     /* mark trouble maker */
      K1->Right=K2->Left;
      K3->Left=K2->Right;
      K2->Left=K1;
      K2->Right=K3;     /* finish setting links */
      K1->Height=Max( Height(K1->Left), Height(K1->Right) ) + 1;
      K3->Height=Max( Height(K3->Left), Height(K3->Right) ) + 1;
      K2->Height=Max( K1->Height, K3->Height ) + 1; /* finish setting heights */
      return   K2;    /* K2 is the new root */
}

static Position   DoubleRotateWithRight( Position   K1 )
{    /* Do the right--left double rotation.   K1 is the trouble finder. */
      Position   K2, K3; /* Similar to the above function */
      K3=K1->Right;
      K2=K3->Left;
      K1->Right=K2->Left;
      K3->Left=K2->Right;
      K2->Left=K1;
      K2->Right=K3;
      K1->Height=Max( Height(K1->Left), Height(K1->Right) ) + 1;
      K3->Height=Max( Height(K3->Left), Height(K3->Right) ) + 1;
      K2->Height=Max( K1->Height, K3->Height ) + 1;
      return   K2;
}
```

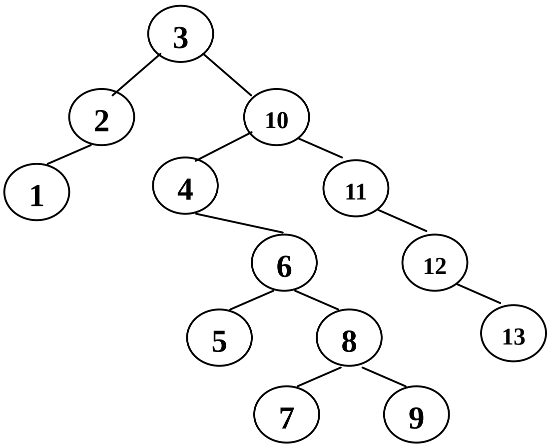
p.141 4.23

Show the result of accessing the keys 3, 9, 1, 5 in order in the splay tree in Figure 4.61.

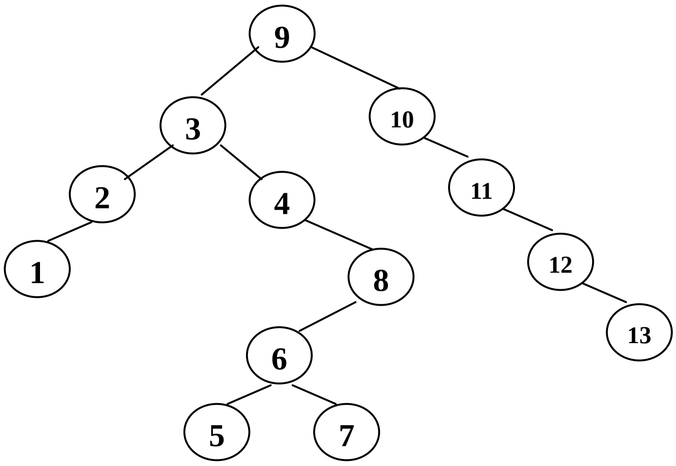
Figure 4.61



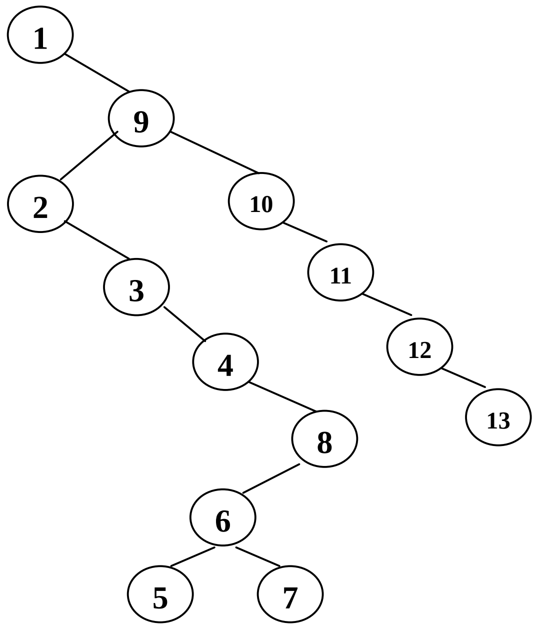
Result for 3:



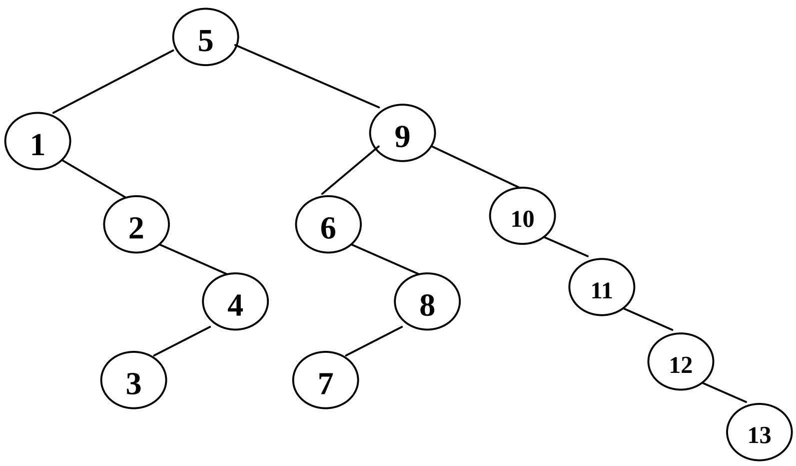
Result for 9:



Result for 1:



Result for 5:

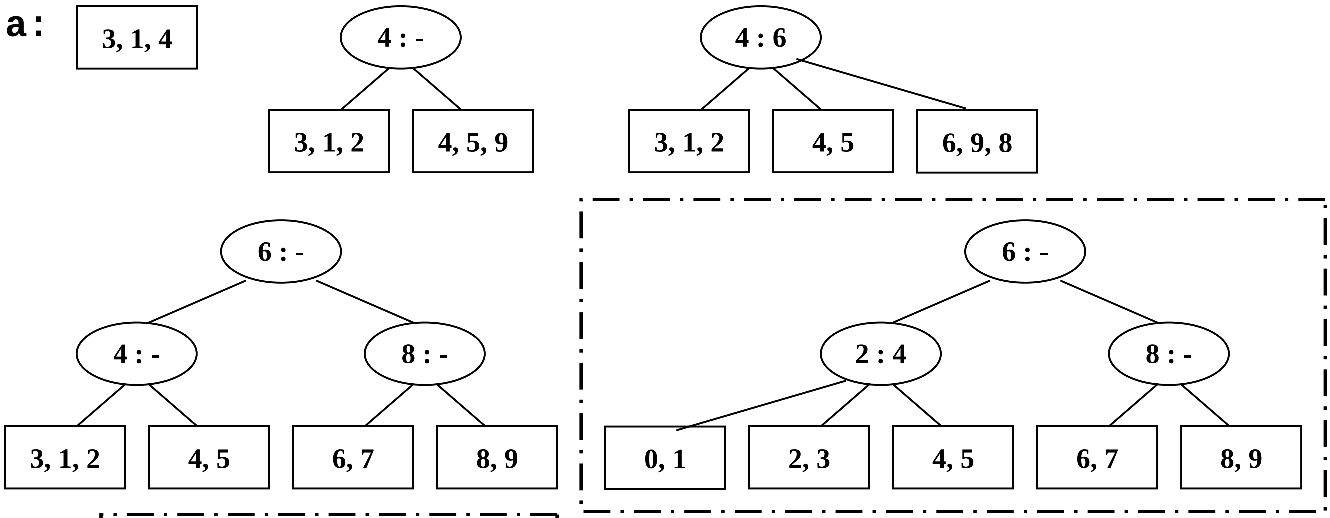


p.143 4.36

a. Show the result of inserting the following keys into an initially empty 2-3 tree: 3, 1, 4, 5, 9, 2, 6, 8, 7, 0.

b. Show the result of deleting 0 and then 9 from the 2-3 tree created in part (a).

a:



b: