www.Linuxidc.com

Chapter 4: Trees

4.1 (a) A.

(b) G, H, I, L, M, and K.

4.2 For node B:

(a) A.

(b) D and E.

(c) C.

(d) 1.

(e) 3.

4.3 4.

4.4 There are N nodes. Each node has two pointers, so there are $2N$ pointers. Each node but the root has one incoming pointer from its parent, which accounts for $N-1$ pointers. The rest are *NULL*.

4.5 Proof is by induction. The theorem is trivially true for $H = 0$. Assume true for $H = 1, 2, ..., k$. A tree of height $k+1$ can have two subtrees of height at most k. These can have at most $2^{k+1}-1$ nodes each by the induction hypothesis. These $2^{k+2}-2$ nodes plus the root prove the theorem for height $k+1$ and hence for all heights.

4.6 This can be shown by induction. Alternatively, let N = number of nodes, F = number of full nodes, L = number of leaves, and H = number of half nodes (nodes with one child). Clearly, $N = F + H + L$. Further, $2F + H = N - 1$ (see Exercise 4.4). Subtracting yields $L - F = 1$.

4.7 This can be shown by induction. In a tree with no nodes, the sum is zero, and in a one-node tree, the root is a leaf at depth zero, so the claim is true. Suppose the theorem is true for all trees with at most k nodes. Consider any tree with $k+1$ nodes. Such a tree consists of an i node left subtree and a $k - i$ node right subtree. By the inductive hypothesis, the sum for the left subtree leaves is at most one with respect to the left tree root. Because all leaves are one deeper with respect to the original tree than with respect to the subtree, the sum is at most ½ with respect to the root. Similar logic implies that the sum for leaves in the right subtree is at most ½, proving the theorem. The equality is true if and only if there are no nodes with one child. If there is a node with one child, the equality cannot be true because adding the second child would increase the sum to higher than 1. If no nodes have one child, then we can find and remove two sibling leaves, creating a new tree. It is easy to see that this new tree has the same sum as the old. Applying this step repeatedly, we arrive at a single node, whose sum is 1. Thus the original tree had sum 1.

4.8 (a) - * * a b + c d e.

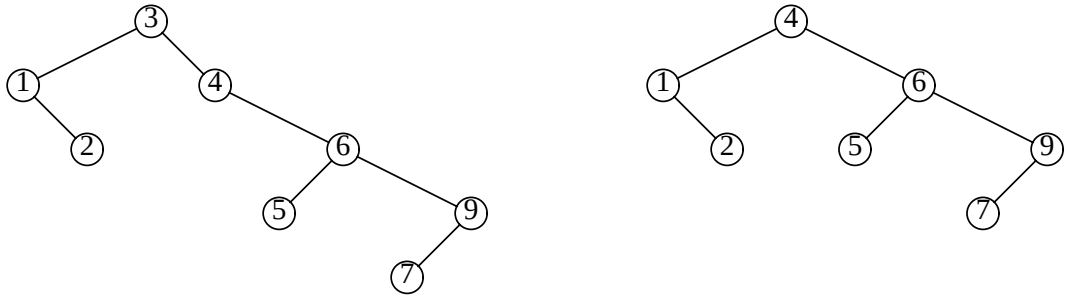
(b) ((a * b) * (c + d)) - e.

(c) a b * c d + * e -.

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

4.9



4.11 This problem is not much different from the linked list cursor implementation. We maintain an array of records consisting of an element field, and two integers, left and right. The free list can be maintained by linking through the left field. It is easy to write the *CursorNew* and *CursorDispose* routines, and substitute them for *malloc* and *free*.

4.12 (a) Keep a bit array B. If i is in the tree, then $B[i]$ is true; otherwise, it is false. Repeatedly generate random integers until an unused one is found. If there are N elements already in the tree, then $M - N$ are not, and the probability of finding one of these is $(M - N)/M$. Thus the expected number of trials is $M/(M-N) = \alpha/(\alpha - 1)$.

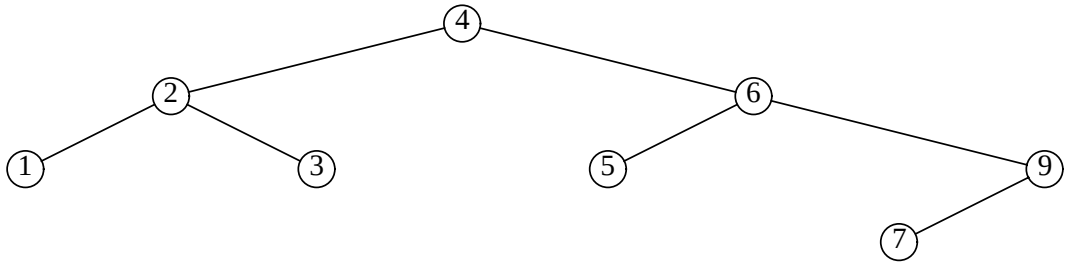
(b) To find an element that is in the tree, repeatedly generate random integers until an already-used integer is found. The probability of finding one is N/M, so the expected number of trials is $M/N = \alpha$.

(c) The total cost for one insert and one delete is $\alpha/(\alpha - 1) + \alpha = 1 + \alpha + 1/(\alpha - 1)$. Setting $\alpha = 2$ minimizes this cost.

4.15 (a) $N(0) = 1$, $N(1) = 2$, $N(H) = N(H-1) + N(H-2) + 1$.

(b) The heights are one less than the Fibonacci numbers.

4.16



4.17 It is easy to verify by hand that the claim is true for $1 \le k \le 3$. Suppose it is true for $k = 1, 2, 3, \ldots H$. Then after the first $2^H - 1$ insertions, 2^{H-1} is at the root, and the right subtree is a balanced tree containing $2^{H-1} + 1$ through $2^H - 1$. Each of the next 2^{H-1} insertions, namely, 2^H through $2^H + 2^{H-1} - 1$, insert a new maximum and get placed in the right

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

subtree, eventually forming a perfectly balanced right subtree of height $H-1$. This follows by the induction hypothesis because the right subtree may be viewed as being formed from the successive insertion of $2^{H-1}+1$ through $2^H+2^{H-1}-1$. The next insertion forces an imbalance at the root, and thus a single rotation. It is easy to check that this brings 2^H to the root and creates a perfectly balanced left subtree of height $H-1$. The new key is attached to a perfectly balanced right subtree of height $H-2$ as the last node in the right path. Thus the right subtree is exactly as if the nodes 2^H+1 through 2^H+2^{H-1} were inserted in order. By the inductive hypothesis, the subsequent successive insertion of $2^H+2^{H-1}+1$ through $2^{H+1}-1$ will create a perfectly balanced right subtree of height $H-1$. Thus after the last insertion, both the left and the right subtrees are perfectly balanced, and of the same height, so the entire tree of $2^{H+1}-1$ nodes is perfectly balanced (and has height H).

4.18 The two remaining functions are mirror images of the text procedures. Just switch *Right* and *Left* everywhere.

4.20 After applying the standard binary search tree deletion algorithm, nodes on the deletion path need to have their balance changed, and rotations may need to be performed. Unlike insertion, more than one node may need rotation.

4.21 (a) $O(\log \log N)$.

(b) The minimum AVL tree of height 255 (a huge tree).

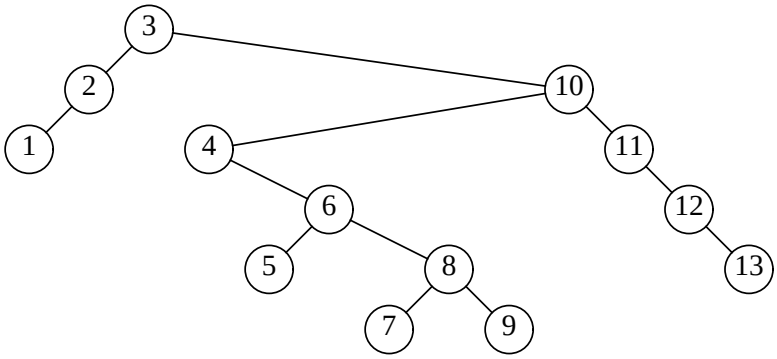
4.22

```
Position
DoubleRotateWithLeft( Position K3 )
{
        Position K1, K2;

        K1 = K3->Left;
        K2 = K1->Right;

        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 K3;
}
```

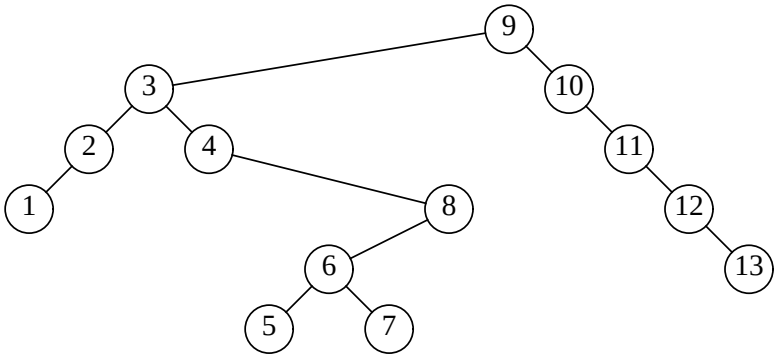

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

4.23 After accessing 3,



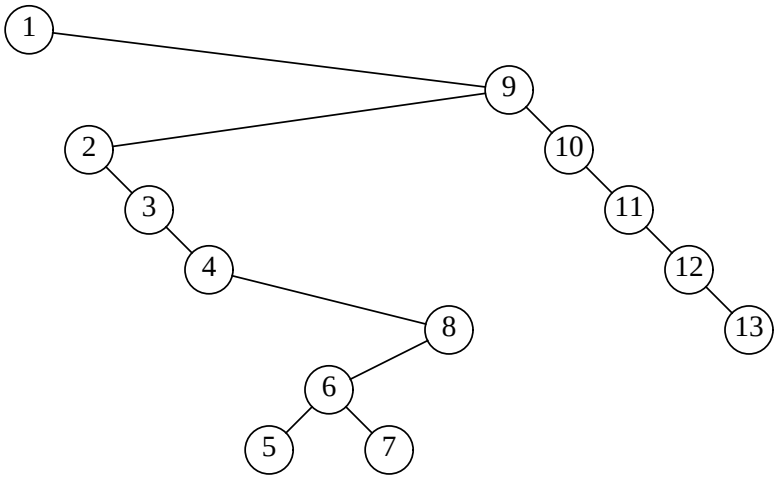
After accessing 9,



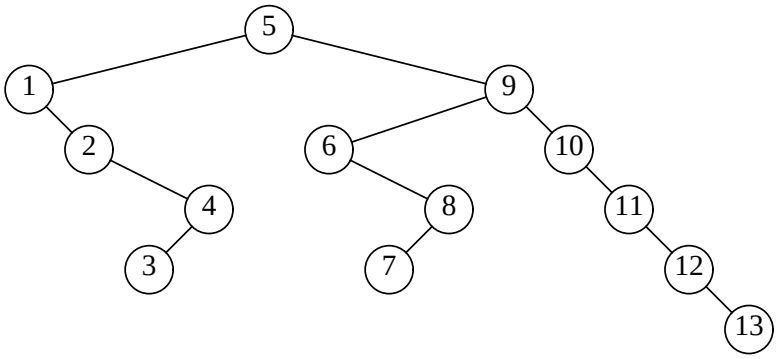
Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

After accessing 1,



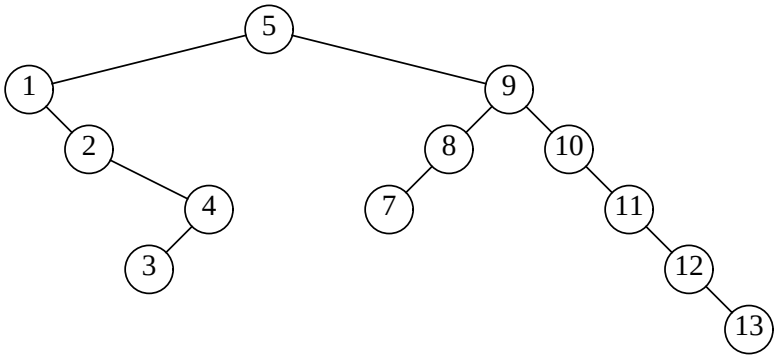
After accessing 5,



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

4.24



4.25 (a) 523776.

(b) 262166, 133114, 68216, 36836, 21181, 13873.

(c) After *Find* (9).

4.26 (a) An easy proof by induction.

4.28 (a-c) All these routines take linear time.

```
/* These functions use the type BinaryTree, which is the same */
/* as TreeNode *, in Fig 4.16. */

int
CountNodes( BinaryTree T )
{
        if( T == NULL )
                return 0;
        return 1 + CountNodes(T->Left) + CountNodes(T->Right);
}

int
CountLeaves( BinaryTree T )
{
        if( T == NULL )
                return 0;
        else if( T->Left == NULL && T->Right == NULL )
                return 1;
        return CountLeaves(T->Left) + CountLeaves(T->Right);
}
```

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
/* An alternative method is to use the results of Exercise 4.6. */

int
CountFull( BinaryTree T )
{
        if( T == NULL )
                return 0;
        return ( T->Left != NULL && T->Right != NULL ) +
                        CountFull(T->Left) + CountFull(T->Right);
}
```

4.29 We assume the existence of a function *RandInt(Lower,Upper)*, which generates a uniform random integer in the appropriate closed interval. *MakeRandomTree* returns NULL if N is not positive, or if N is so large that memory is exhausted.

```
SearchTree
MakeRandomTree1( int Lower, int Upper )
{
        SearchTree T;
        int RandomValue;

        T = NULL;
        if( Lower <= Upper )
        {
                T = malloc( sizeof( struct TreeNode ) );
                if( T != NULL )
                {
                        T->Element = RandomValue = RandInt( Lower, Upper );
                        T->Left  = MakeRandomTree1( Lower, RandomValue - 1 );
                        T->Right = MakeRandomTree1( RandomValue + 1, Upper );
                }
                else
                        FatalError( "Out of space!" );
        }
        return T;
}

SearchTree
MakeRandomTree( int N )
{
        return MakeRandomTree1( 1, N );
}
```


Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com


```
/* LastNode is the address containing last value that was assigned to a node */

SearchTree
GenTree( int Height, int *LastNode )
{
        SearchTree T;

        if( Height >= 0 )
        {
                T = malloc( sizeof( *T ) ); /* Error checks omitted; see Exercise 4.29. */
                T->Left = GenTree( Height - 1, LastNode );
                T->Element = ++*LastNode;
                T->Right = GenTree( Height - 2, LastNode );
                return T;
        }
        else
                return NULL;
}

SearchTree
MinAvlTree( int H )
{
        int LastNodeAssigned = 0;
        return GenTree( H, &LastNodeAssigned );
}
```

4.31 There are two obvious ways of solving this problem. One way mimics Exercise 4.29 by replacing *RandInt(Lower,Upper)* with *(Lower+Upper) / 2*. This requires computing $2^{H+1}-1$, which is not that difficult. The other mimics the previous exercise by noting that the heights of the subtrees are both $H-1$. The solution follows:

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

```
/* LastNode is the address containing last value that was assigned to a node. */

SearchTree
GenTree( int Height, int *LastNode )
{
        SearchTree T = NULL;

        if( Height >= 0 )
        {
                T = malloc( sizeof( *T ) );  /* Error checks omitted; see Exercise 4.29. */
                T->Left = GenTree( Height - 1, LastNode );
                T->Element = ++*LastNode;
                T->Right = GenTree( Height - 1, LastNode );
        }
        return T;
}

SearchTree
PerfectTree( int H )
{
        int LastNodeAssigned = 0;
        return GenTree( H, &LastNodeAssigned );
}
```

4.32 This is known as one-dimensional range searching. The time is $O(K)$ to perform the inorder traversal, if a significant number of nodes are found, and also proportional to the depth of the tree, if we get to some leaves (for instance, if no nodes are found). Since the average depth is $O(\log N)$, this gives an $O(K + \log N)$ average bound.

```
void
PrintRange( ElementType Lower, ElementType Upper, SearchTree T )
{
        if( T != NULL )
        {
                if( Lower <= T->Element )
                        PrintRange( Lower, Upper, T->Left );
                if( Lower <= T->Element && T->Element <= Upper )
                        PrintLine( T->Element );
                if( T->Element <= Upper )
                        PrintRange( Lower, Upper, T->Right );
        }
}
```

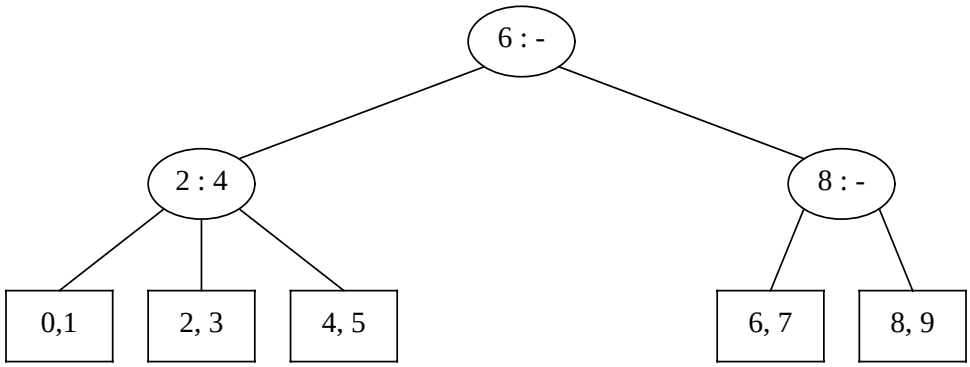

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

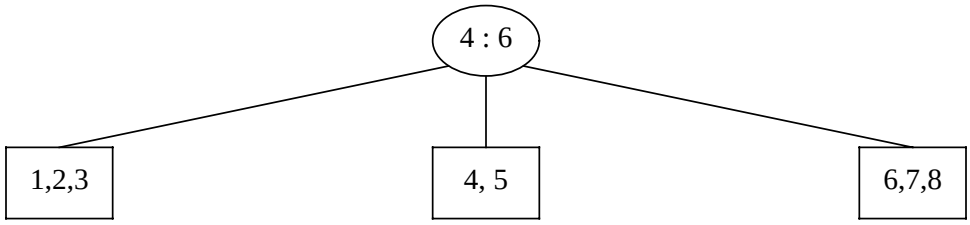
4.33 This exercise and Exercise 4.34 are likely programming assignments, so we do not provide code here.

4.35 Put the root on an empty queue. Then repeatedly *Dequeue* a node and *Enqueue* its left and right children (if any) until the queue is empty. This is $O(N)$ because each queue operation is constant time and there are N *Enqueue* and N *Dequeue* operations.

4.36 (a)



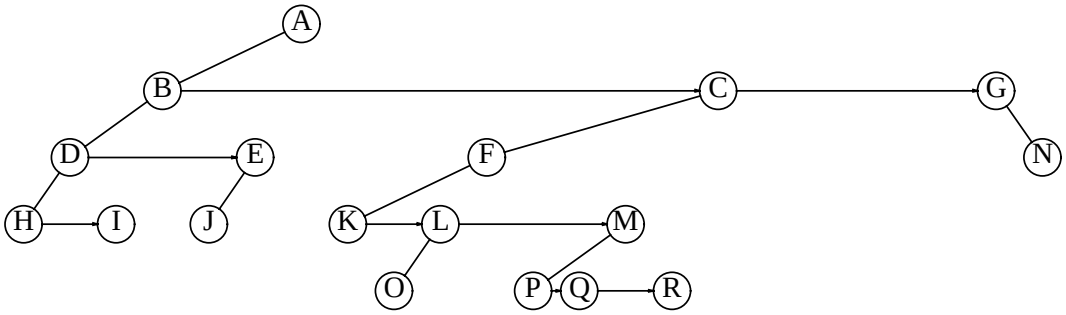
(b)



Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。

www.Linuxidc.com

4.39



4.41 The function shown here is clearly a linear time routine because in the worst case it does a traversal on both $T1$ and $T2$.

```
int
Similar( BinaryTree T1, BinaryTree T2 )
{
        if( T1 == NULL || T2 == NULL )
                return T1 == NULL && T2 == NULL;
        return Similar( T1->Left, T2->Left ) && Similar( T1->Right, T2->Right );
}
```

4.43 The easiest solution is to compute, in linear time, the inorder numbers of the nodes in both trees. If the inorder number of the root of T2 is x, then find x in T1 and rotate it to the root. Recursively apply this strategy to the left and right subtrees of T1 (by looking at the values in the root of T2's left and right subtrees). If d_N is the depth of x, then the running time satisfies $T(N) = T(i) + T(N-i-1) + d_N$, where i is the size of the left subtree. In the worst case, d_N is always $O(N)$, and i is always 0, so the worst-case running time is quadratic. Under the plausible assumption that all values of i are equally likely, then even if d_N is always $O(N)$, the average value of $T(N)$ is $O(N \log N)$. This is a common recurrence that was already formulated in the chapter and is solved in Chapter 7. Under the more reasonable assumption that d_N is typically logarithmic, then the running time is $O(N)$.

4.44 Add a field to each node indicating the size of the tree it roots. This allows computation of its inorder traversal number.

4.45 (a) You need an extra bit for each thread.

(c) You can do tree traversals somewhat easier and without recursion. The disadvantage is that it reeks of old-style hacking.

Linux公社（www.LinuxIDC.com）是专业的Linux系统门户网站，实时发布最新Linux资讯。