4

INTRODUCTION TO TREES

Unit Structure:

4.1 TREES

A *tree* can be defined in several ways. One way to define a tree is recursively. A tree is a collection of nodes. The collection can be empty, which is sometimes denoted as A. Otherwise, a tree consists of a distinguished node *r*, called the *root*, and zero or more (sub)trees T_1, T_2, . . . , T_k, each of whose roots are connected by a directed *edge* to *r*.

E-next THE NEXT LEVEL OF EDUCATION

The root of each subtree is said to be a *child* of *r*, and *r* is the *parent* of each subtree root. Figure 4.1 shows a typical tree using the recursive definition.

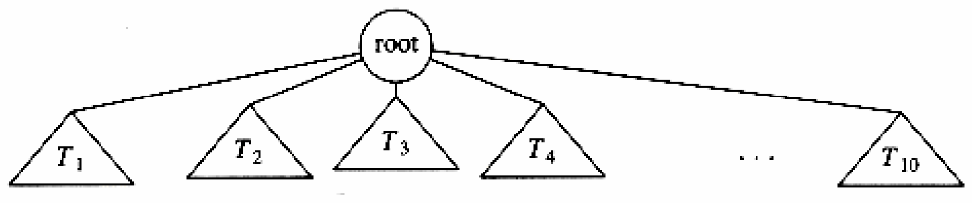


Figure 4.1 Generic tree

From the recursive definition, we find that a tree is a collection of *n* nodes, one of which is the root, and *n* - 1 edges. Having *n* - 1 edges follows from the fact that each edge connects some node to its parent, and every node except the root has one parent (see Fig. 4.2).

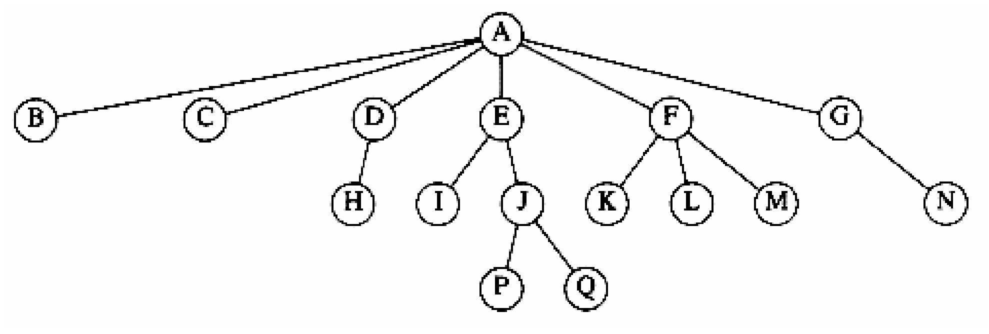


Figure 4.2 A tree

In the tree of Figure 4.2, the root is *A*. Node *F* has *A* as a parent and *K*, *L*, and *M* as children. Each node may have an arbitrary number of children, possibly zero. Nodes with no children are known as *leaves*; the leaves in the tree above are *B*, *C*, *H*, *I*, *P*, *Q*, *K*, *L*, *M*, and *N*. Nodes with the same parent are *siblings*; thus *K*, L, and *M* are all siblings. *Grandparent* and *grandchild* relations can be defined in a similar manner.

A *path* from node n_1 to n_k is defined as a sequence of nodes n_1, n_2, . . . , n_k such that n_i is the parent of n_{i+1} for $1 < i < k$. The *length* of this path is the number of edges on the path, namely *k* -1. There is a path of length zero from every node to itself. Notice that in a tree there is exactly one path from the root to each node.

E-next THE NEXT LEVEL OF EDUCATION

For any node n_i, the *depth* of n_i is the length of the unique path from the root to n_i. Thus, the root is at depth 0. The *height* of n_i is the longest path from n_i to a leaf. Thus all leaves are at height 0. The height of a tree is equal to the height of the root. For the tree in Figure 4.2, *E* is at depth 1 and height 2; *F* is at depth 1 and height 1; the height of the tree is 3. The depth of a tree is equal to the depth of the deepest leaf; this is always equal to the height of the tree.

If there is a path from n_1 to n_2, then n_1 is an *ancestor* of n_2 and n_2 is a *descendant* of n_1.

4.1.1. Implementation of Trees

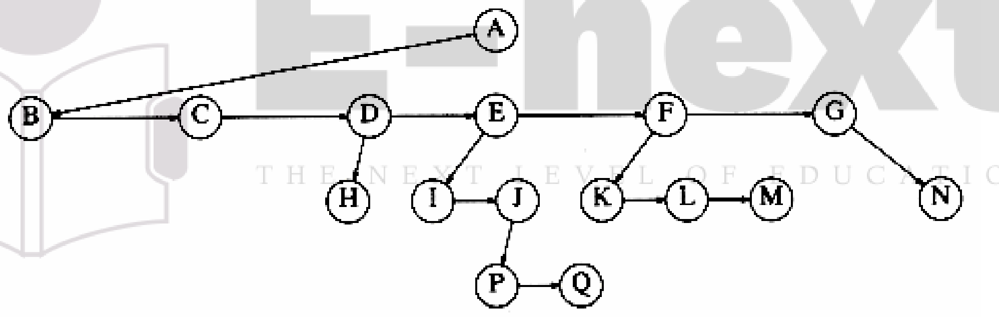
One way to implement a tree would be to have in each node, besides its data, a pointer to each child of the node. However, since the number of children per node can vary so greatly and is not known in advance, it might be infeasible to make the children direct links in the data structure, because there would be too much wasted space. The solution is simple: Keep the children of each node in a linked list of tree nodes. The declaration in Figure 4.3 is typical.

```
typedef struct tree_node *tree_ptr;
struct tree_node
{
        element_type element;
        tree_ptr first_child;
        tree_ptr next_sibling;
};
```

Figure 4.3 Node declarations for trees

Figure 4.4 shows how a tree might be represented in this implementation. Arrows that point downward are *first_child* pointers. Arrows that go left to right are *next_sibling* pointers. Null pointers are not drawn, because there are too many.

In the tree of Figure 4.4, node *E* has both a pointer to a sibling (*F*) and a pointer to a child (*I*), while some nodes have neither.

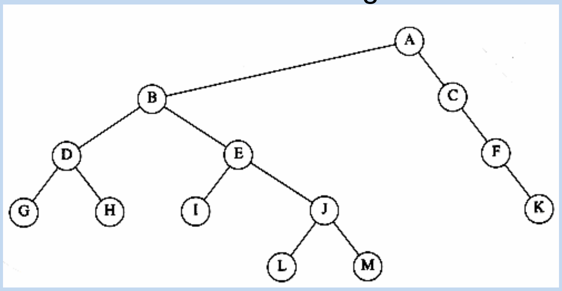


E-next
THE NEXT LEVEL OF EDUCATION

Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2

Check your progress

1. For the tree in the following:



   a. Which node is the root?
   b. Which nodes are leaves?

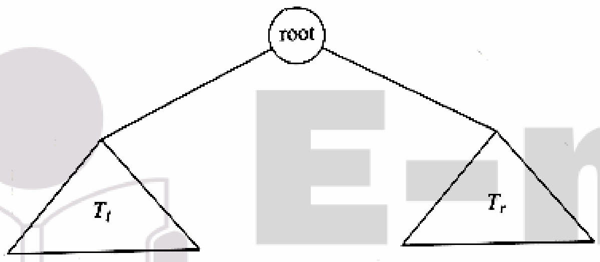
2. For each node in the tree of Figure above:

   a. *Name the parent node.*
   b. *List the children.*
   c. *List the siblings.*
   d. *Compute the depth.*
   e. *Compute the height*.

4.2 BINARY TREES:

A binary tree is a tree in which no node can have more than two children.

Figure 4.5 shows that a binary tree consists of a root and two subtrees, T_l and T_r, both of which could possibly be empty



E-next THE NEXT LEVEL OF EDUCATION

Figure 4.5 Generic binary tree

A property of a binary tree that is sometimes important is that the depth of an average binary tree is considerably smaller than *n*. Unfortunately, the depth can be as large as *n* -1, as the example in Figure 4.6 shows.

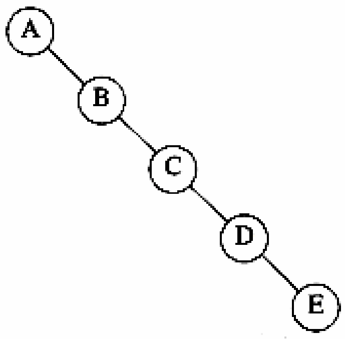


Figure 4.6 Worst-case binary tree

4.2.1 Implementation

Because a binary tree has at most two children, we can keep direct pointers to them. The declaration of tree nodes is similar in structure to that for doubly linked lists, in that a node is a structure consisting of the *key* information plus two pointers (*left* and *right*) to other nodes.

```
typedef struct tree_node *tree_ptr;
struct tree_node
{
        element_type element;
        tree_ptr left;
        tree_ptr right;
};
typedef tree_ptr TREE;
```

Figure 4.7 Binary tree node declarations

Many of the rules that apply to linked lists will apply to trees as well. In particular, when an insertion is performed, a node will have to be created by a call to *malloc*. Nodes can be freed after deletion by calling *free*.

We could draw the binary trees using the rectangular boxes that are customary for linked lists, but trees are generally drawn as circles connected by lines, because they are actually graphs. We also do not explicitly draw *NULL* pointers when referring to trees, because every binary tree with *n* nodes would require *n* + 1 *NULL* pointers.

Binary trees have many important uses not associated with searching. One of the principal uses of binary trees is in the area of compiler design.

4.2.2 Types of binary trees:

- A **rooted binary tree** is a rooted tree in which every node has at most two children.
- A **full binary tree** (sometimes **proper binary tree** or **2-tree** or **strictly binary tree**) is a tree in which every node other than the leaves has two children.
- A **perfect binary tree** is a full binary tree in which all leaves are at the same depth or same level. (This is ambiguously also called a **complete binary tree**.)
- A **complete binary tree** is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

- An **infinite complete binary** tree is a tree with N_0 levels, where for each level d the number of existing nodes at level d is equal to 2d. The cardinal number of the set of all nodes is N_0. The cardinal number of the set of all paths is 2^{N_0}.
- A **balanced binary tree** is where the depth of all the leaves differs by at most 1. Balanced trees have a predictable depth (how many nodes are traversed from the root to a leaf, root counting as node 0 and subsequent as 1, 2, ..., depth). This depth is equal to the integer part of $log_2(n)$ where n is the number of nodes on the balanced tree. Example 1: balanced tree with 1 node, $log_2(1)$ = 0 (depth = 0). Example 2: balanced tree with 3 nodes, $log_2(3)$ = 1.59 (depth=1). Example 3: balanced tree with 5 nodes, $log_2(5)$ = 2.32 (depth of tree is 2 nodes).
- A **rooted complete binary tree** can be identified with a free magma.
- A **degenerate tree** is a tree where for each parent node, there is only one associated child node. This means that in a performance measurement, the tree will behave like a linked list data structure.

A rooted tree has a top node as root.

4.2.3 Properties of binary trees

E-next THE NEXT LEVEL OF EDUCATION

- The number of nodes n in a perfect binary tree can be found using this formula: $n = 2^{h + 1} - 1$ where h is the height of the tree.
- The number of nodes n in a complete binary tree is minimum: $n = 2^h$ and maximum: $n = 2^{h + 1} - 1$ where h is the height of the tree.
- The number of nodes n in a perfect binary tree can also be found using this formula: $n = 2L - 1$ where L is the number of leaf nodes in the tree.
- The number of leaf nodes n in a perfect binary tree can be found using this formula: $n = 2^h$ where h is the height of the tree.
- The number of NULL links in a Complete Binary Tree of n-node is (n+1).
- The number of leaf node in a Complete Binary Tree of n-node is *UpperBound* (n / 2).
- For any non-empty binary tree with n_0 leaf nodes and n_2 nodes of degree 2, $n_0 = n_2 + 1$.

4.2.4 Tree Traversals

In computer science, **tree-traversal** refers to the process of visiting (examining and/or updating) each node in a tree data structure, exactly once, in a systematic way. Such traversals are classified by the order in which the nodes are visited. The following algorithms are described for a binary tree, but they may be generalized to other trees as well.

Compared to linear data structures like linked lists and one dimensional arrays, which have only one logical means of traversal, tree structures can be traversed in many different ways. Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps (in no particular order) are: performing an action on the current node (referred to as "visiting" the node), traversing to the left child node, and traversing to the right child node. Thus the process is most easily described through recursion.

To traverse a non-empty binary tree in **preorder**, perform the following operations recursively at each node, starting with the root node:

1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.

(This is also called Depth-first traversal.)

E-next THE NEXT LEVEL OF EDUCATION

To traverse a non-empty binary tree in **inorder**, perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree.

(This is also called **Symmetric traversal**.)

To traverse a non-empty binary tree in **postorder**, perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root.

Finally, trees can also be traversed in **level-order**, where we visit every node on a level before going to a lower level. This is also called Breadth-first traversal.

Example

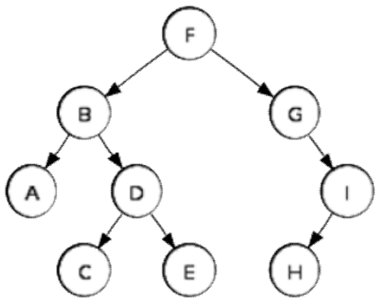


Figure 4.8 Binary tree

In this binary tree fig. 4.8

- Preorder traversal sequence: F, B, A, D, C, E, G, I, H (root, left, right)
- Inorder traversal sequence: A, B, C, D, E, F, G, H, I (left, root, right)
- Postorder traversal sequence: A, C, E, D, B, H, I, G, F (left, right, root)
- Level-order traversal sequence: F, B, G, A, D, I, C, E, H

E-next THE NEXT LEVEL OF EDUCATION

4.2.5 Sample implementations

```
preorder(node)
  print node.value
  if node.left ≠ null then preorder(node.left)
  if node.right ≠ null then preorder(node.right)
```

Figure 4.9 Sample preorder implementation

```
inorder(node)
  if node.left  ≠ null then inorder(node.left)
  print node.value
  if node.right ≠ null then inorder(node.right)
```

Figure 4.10 Sample inorder implementation

```
postorder(node)
  if node.left  ≠ null then postorder(node.left)
  if node.right ≠ null then postorder(node.right)
  print node.value
```

Figure 4.11 Sample inorder implementation

All sample implementations will require call stack space proportional to the height of the tree. In a poorly balanced tree, this can be quite considerable.

4.3 EXPRESSION TREES:

Figure 4.12 shows an example of an *expression tree*. The leaves of an expression tree are *operands*, such as constants or variable names, and the other nodes contain *operators*. This particular tree happens to be binary, because all of the operations are binary, and although this is the simplest case, it is possible for nodes to have more than two children. It is also possible for a node to have only one child, as is the case with the *unary minus* operator. We can evaluate an expression tree, *T*, by applying the operator at the root to the values obtained by recursively evaluating the left and right subtrees. In our example, the left subtree evaluates to *a* + (*b* * *c*) and the right subtree evaluates to ((*d* **e*) + *f*)**g.* The entire tree therefore represents (*a* + (*b***c*)) + (((*d* * *e*) + *f*)* *g*).

We can produce an (overly parenthesized) infix expression by recursively producing a parenthesized left expression, then printing out the operator at the root, and finally recursively producing a parenthesized right expression. This general strattegy (left, node, right) is known as an *inorder* traversal; it is easy to remember because of the type of expression it produces.

E-next THE NEXT LEVEL OF EDUCATION

An alternate traversal strategy is to recursively print out the left subtree, the right subtree, and then the operator. If we apply this strategy to our tree above, the output is a b c * + d e * f + g * +, which is easily seen to be the postfix representation. This traversal strategy is generally known as a *postorder* traversal.

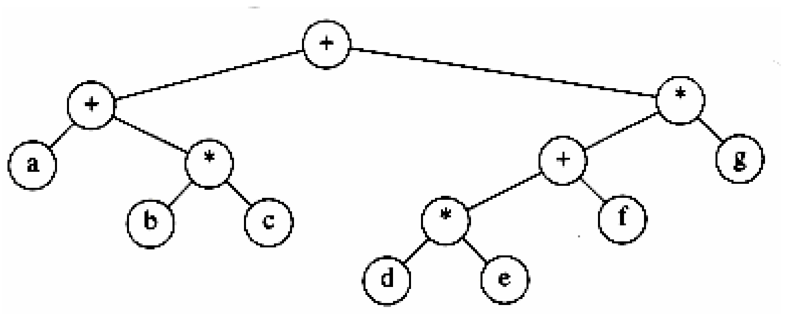


Figure 4.12 Expression tree for (a + b * c) + ((d * e + f) * g)

A third traversal strategy is to print out the operator first and then recursively print out the left and right subtrees. The resulting expression, + + a * b c * + * d e f g, is the less useful *prefix* notation and the traversal strategy is a *preorder* traversal.

4.3.1 Constructing an Expression Tree

We now give an algorithm to convert a postfix expression into an expression tree. Since we already have an algorithm to convert infix to postfix, we can generate expression trees from the two common types of input. We read our expression one symbol at a time. If the symbol is an operand, we create a one-node tree and push a pointer to it onto a stack. If the symbol is an operator, we pop pointers to two trees T_1 and T_2 from the stack (T_1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T_2 and T_1 respectively. A pointer to this new tree is then pushed onto the stack.

As an example, suppose the input is a b + c d e + * *
The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack.*

*For convenience, we will have the stack grow from left to right in the diagrams.

E-next
THE NEXT LEVEL OF EDUCATION

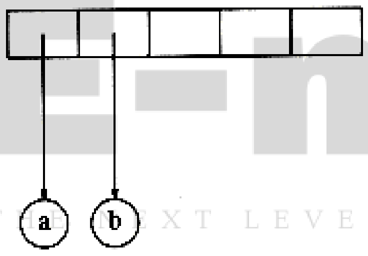


Figure 4.13 Pushing pointers to a and b to stack

Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.

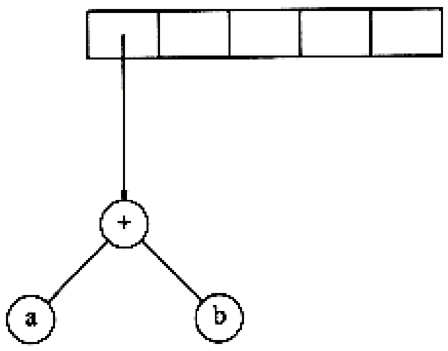


Figure 4.14 Pushing pointers to + to stack

Next, *c*, *d*, and *e* are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.

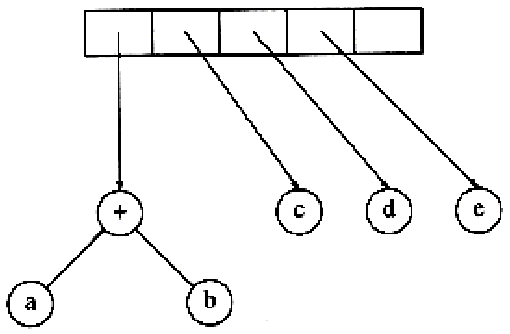
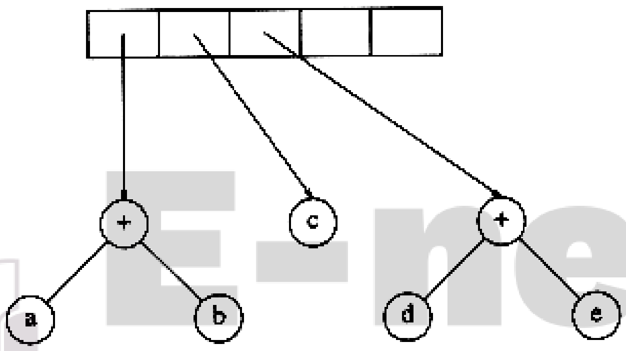


Figure 4.15 Pushing pointers to c, d and e to stack

Now a '+' is read, so two trees are merged.



E-next
THE NEXT LEVEL OF EDUCATION

Figure 4.16 Merging two trees

Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.

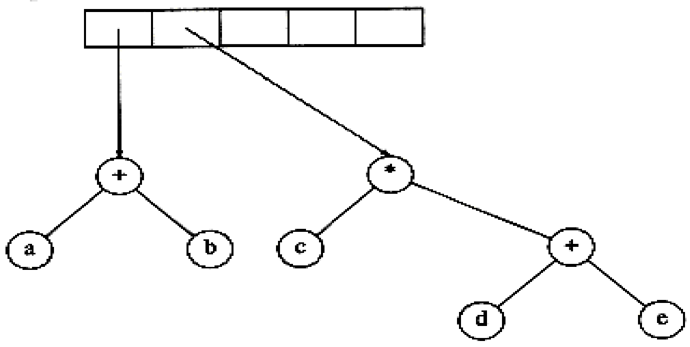


Figure 4.17 Popping two tree pointers and forming a new tree with a '*' as root

Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.

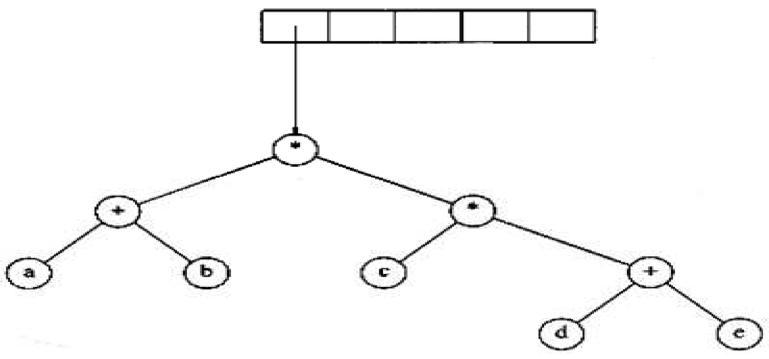
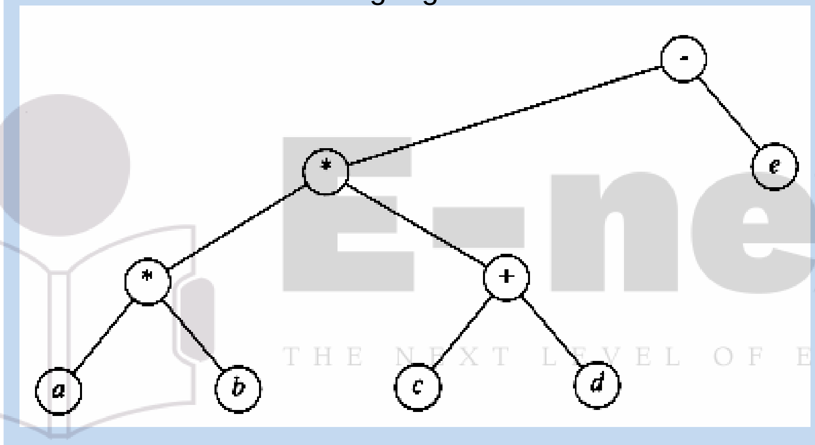


Figure 4.18 Final tree

Check your progress:

1. Give the prefix, infix, and postfix expressions corresponding to the tree in the following Figure:



E-next
THE NEXT LEVEL OF EDUCATION

2. Convert the infix expression a + b * c + (d * e + f) * g into postfix.

4.4. AVL TREES:

An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a *balance* condition. The balance condition must be easy to maintain, and it ensures that the depth of the tree is *O*(log *n*). The simplest idea is to require that the left and right subtrees have the same height. As Figure 4.19 shows, this idea does not force the tree to be shallow.

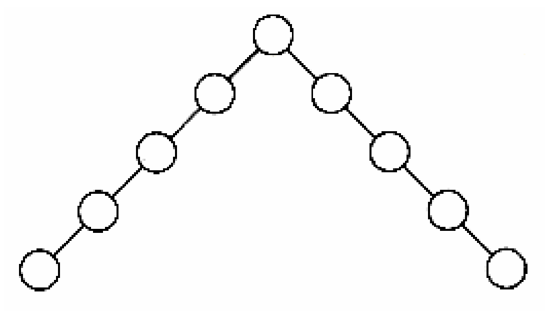


Figure 4.19 A bad binary tree. Requiring balance at the root is not enough.

Another balance condition would insist that every node must have left and right subtrees of the same height. If the height of an empty subtree is defined to be -1 (as is usual), then only perfectly balanced trees of 2^k - 1 nodes would satisfy this criterion. Thus, although this guarantees trees of small depth, the balance condition is too rigid to be useful and needs to be relaxed.

An AVL tree is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1. (The height of an empty tree is defined to be -1.) In Figure 4.20 the tree on the left is an AVL tree, but the tree on the right is not. Height information is kept for each node (in the node structure). It is easy to show that the height of an AVL tree is at most roughly 1.44 log(n + 2) - .328, but in practice it is about log(n + 1) + 0.25 (although the latter claim has not been proven). As an example, the AVL tree of height 9 with the fewest nodes (143) is shown in Figure 4.21. This tree has as a left subtree an AVL tree of height 7 of minimum size. The right subtree is an AVL tree of height 8 of minimum size. This tells us that the minimum number of nodes, $N(h)$, in an AVL tree of height h is given by $N(h) = N(h -1) + N(h - 2) + 1$. For $h = 0$, $N(h) = 1$. For $h = 1$, $N(h) = 2$. The function $N(h)$ is closely related to the Fibonacci numbers, from which the bound claimed above on the height of an AVL tree follows.

Thus, all the tree operations can be performed in $O(\log n)$ time, except possibly insertion (we will assume lazy deletion). When we do an insertion, we need to update all the balancing information for the nodes on the path back to the root, but the reason that insertion is potentially difficult is that inserting a node could violate the AVL tree property. (For instance, inserting 6½ into the AVL tree in Figure 4.20 would destroy the balance condition at the node with key 8.) If this is the case, then the property has to be restored before the insertion step is considered over. It turns out that this can always be done with a simple modification to the tree, known as a rotation.

E-next THE NEXT LEVEL OF EDUCATION

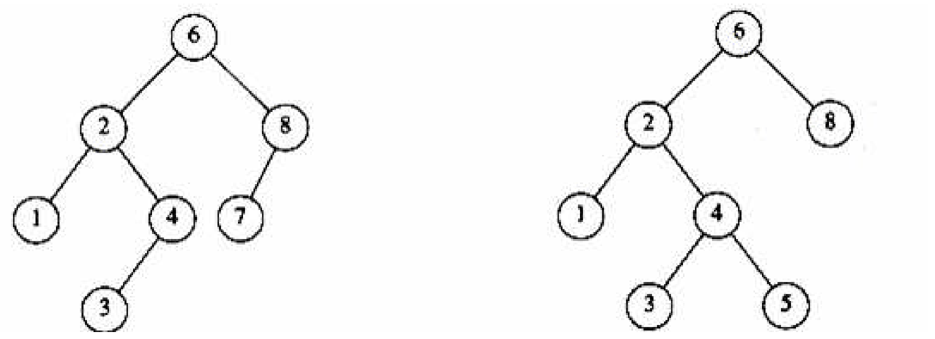


Figure 4.20 Two binary search trees. Only the left tree is AVL.

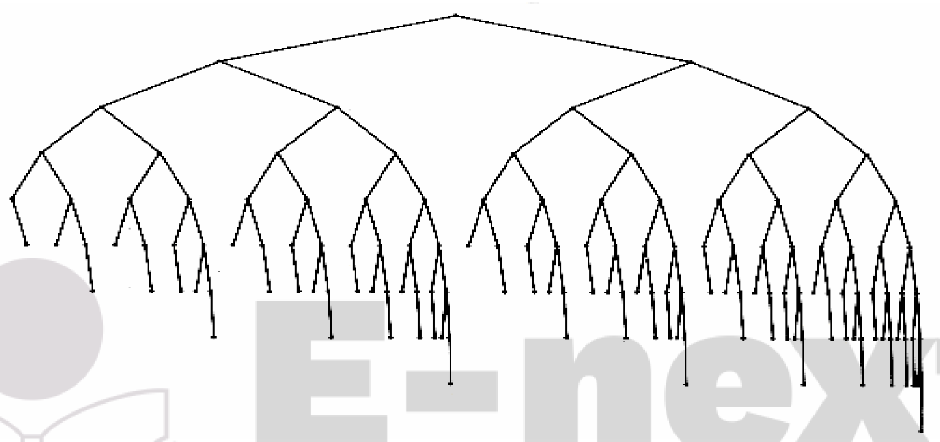


Figure 4.21 Smallest AVL tree of height 9

E-next THE NEXT LEVEL OF EDUCATION

4.4.1 Operations of AVL trees

The basic operations of an AVL tree generally involve carrying out the same actions as would be carried out on an unbalanced binary search tree, but modifications are preceded or followed by one or more operations called tree rotations, which help to restore the height balance of the subtrees.

4.4.1.1 Insertion

Pictorial description of how rotations cause rebalancing tree, and then retracing one's steps toward the root updating the balance factor of the nodes.

After inserting a node, it is necessary to check each of the node's ancestors for consistency with the rules of AVL. For each node checked, if the balance factor remains -1, 0, or 1 then no rotations are necessary. However, if the balance factor becomes 2 or -2 then the subtree rooted at this node is unbalanced. If insertions are performed serially, after each insertion, at most two tree rotations are needed to restore the entire tree to the rules of AVL.

There are four cases which need to be considered, of which two are symmetric to the other two. Let P be the root of the unbalanced subtree. Let R be the right child of P. Let L be the left child of P.

Right-Right case and **Right-Left case**: If the balance factor of P is -2, then the right subtree outweighs the left subtree of the given node, and the balance factor of the right child (R) must be checked. If the balance factor of R is -1, a **left rotation** is needed with P as the root. If the balance factor of R is +1, a **double left rotation** is needed. The first rotation is a right rotation with R as the root. The second is a left rotation with P as the root.

Left-Left case and **Left-Right case**: If the balance factor of P is +2, then the left subtree outweighs the right subtree of the given node, and the balance factor of the left child (L) must then checked. If the balance factor of L is +1, a **right rotation** is needed with P as the root. If the balance factor of L is -1, a **double right rotation** is needed. The first rotation is a left rotation with L as the root. The second is a right rotation with P as the root.

E-next
THE NEXT LEVEL OF EDUCATION

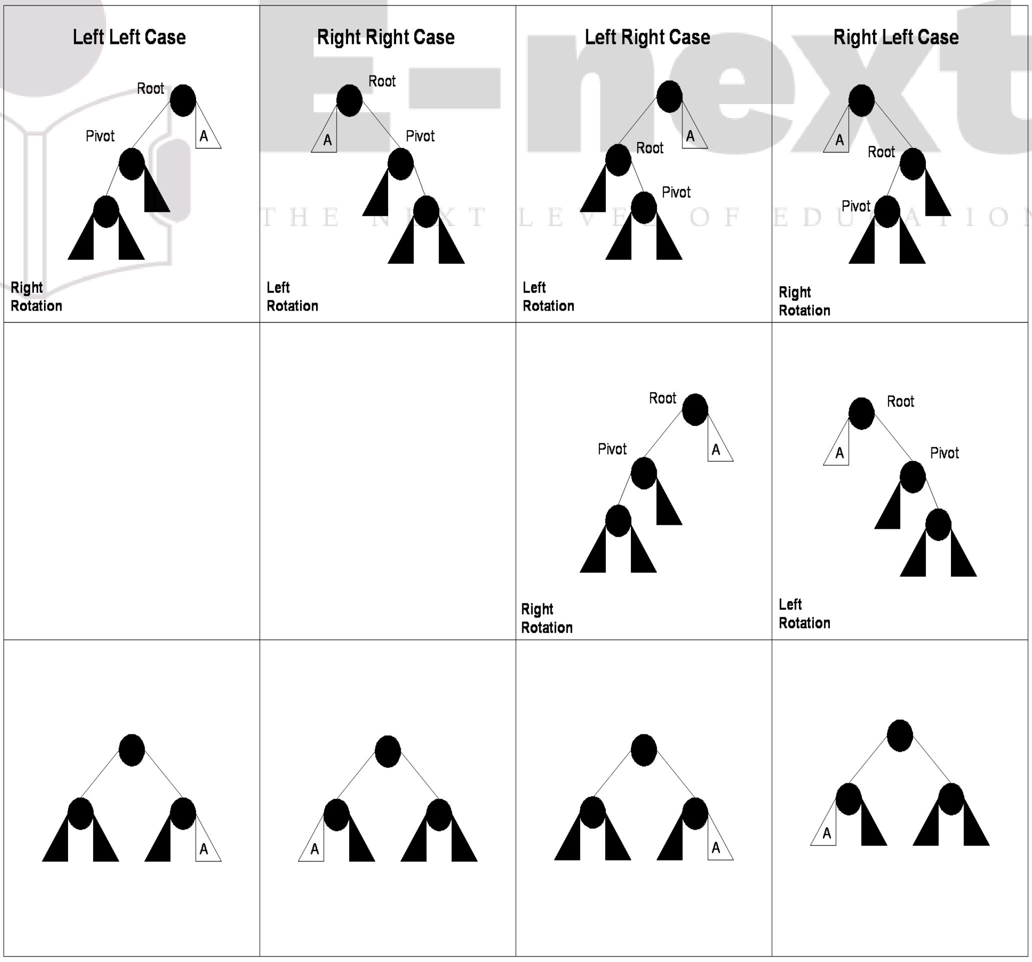


Figure 4.22 Insertion

4.4.1.2 Rotation Algorithms for putting an out-of-balance AVL-tree back in balance.

Note: After the insertion of each node the tree should be checked for balance. The only nodes that need checked are the ones along the insertion path of the newly inserted node. Once the tree is found to be out-of-balance then re-balance it using the appropriate algorithm. If the out-of-balance is detected as soon as it happens and the proper algorithm is used then the tree will be back in balance after one rotation.

Step 1: Set up the pointers:

A - points to the node that is out of balance. If more than one node is out of balance then select the one that is furthest from the root. If there is a tie then you missed a previous out-of-balance.

B – points to the child of A in the direction of the out-of-balance

C – points to the child of B in the direction of the out-=of-balance

F – points to the parent of A. This is the only pointer of these 4 that is allowed to be NULL.

E-next
THE NEXT LEVEL OF EDUCATION

Step 2: Determine the appropriate algorithm:

The first letter of the algorithm represents the direction from A to B (either **R**ight or **L**eft). The second letter represents the direction from B to C (either **R**ight or **L**eft).

Step 3: Follow the algorithm:

```
RR:
A.Right = B.Left
B.Left = A
If F = NULL
    B is new Root of tree
Else
    If F.Right = A
        F.Right = B
    Else
        F.Left = B
    End If
End If
```

```
LL:
A.Left = B.Right
B.Right = A
If F = NULL
    B is new Root of tree
Else
    If F.Right = A
        F.Right = B
    Else
        F.Left = B
    End If
End If
```

```
RL:                                 LR:
B.Left = C.Right                    A.Left = C.Right
A.Right = C.Left                    B.Right = C.Left
C.Right = B                         C.Left = B
C.Left = A                          C.Right = A
If F = NULL                         If F = NULL
    C is new Root of tree               C is new Root of tree
Else                                Else
    If F.Right = A                      If F.Right = A
        F.Right = C                         F.Right = C
    Else                                Else
        F.Left = C                          F.Left = C
    End If                              End If
End If                              End If
```

Figure 4.23 Rotation Algorithms

4.4.1.3 Deletion

If the node is a leaf, remove it. If the node is not a leaf, replace it with either the largest in its left subtree (inorder predecessor) or the smallest in its right subtree (inorder successor), and remove that node. The node that was found as replacement has at most one subtree. After deletion, retrace the path back up the tree (parent of the replacement) to the root, adjusting the balance factors as needed.

E-next THE NEXT LEVEL OF EDUCATION

As with all binary trees, a node's in-order successor is the left-most child of its right subtree, and a node's in-order predecessor is the right-most child of its left subtree. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.

In addition to the balancing described above for insertions, if the balance factor for the tree is 2 and that of the left subtree is 0, a right rotation must be performed on P. The mirror of this case is also necessary.

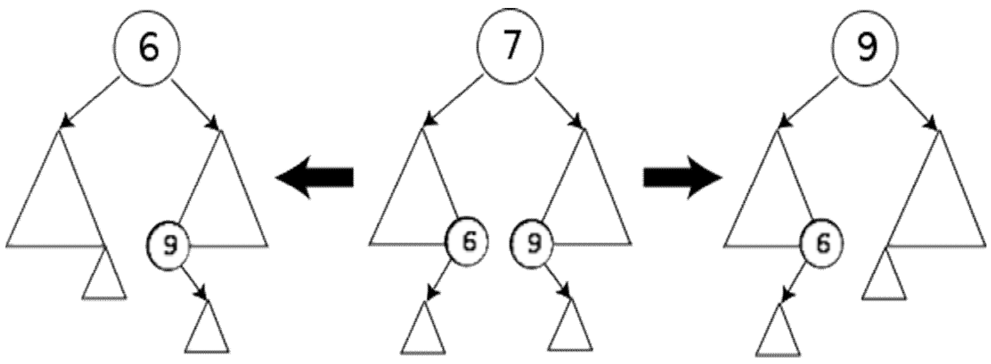


Figure 4.24 Deletion

The retracing can stop if the balance factor becomes -1 or 1 indicating that the height of that subtree has remained unchanged. If the balance factor becomes 0 then the height of the subtree has decreased by one and the retracing needs to continue. If the balance factor becomes -2 or 2 then the subtree is unbalanced and needs to be rotated to fix it. If the rotation leaves the subtree's balance factor at 0 then the retracing towards the root must continue since the height of this subtree has decreased by one. This is in contrast to an insertion where a rotation resulting in a balance factor of 0 indicated that the subtree's height has remained unchanged.

The time required is O(log *n*) for lookup, plus a maximum of O(log *n*) rotations on the way back to the root, so the operation can be completed in O(log *n*) time.

4.4.1.4 Lookup

Lookup in an AVL tree is performed exactly as in an unbalanced binary search tree. Because of the height-balancing of the tree, a lookup takes O(log *n*) time. No special provisions need to be taken, and the tree's structure is not modified by lookups. (This is in contrast to splay tree lookups, which do modify their tree's structure.)

E-next THE NEXT LEVEL OF EDUCATION

If each node additionally records the size of its subtree (including itself and its descendants), then the nodes can be retrieved by index in O(log *n*) time as well.

Once a node has been found in a balanced tree, the *next* or *previous* node can be obtained in amortized constant time. (In a few cases, about 2*log(n) links will need to be traversed. In most cases, only a single link needs to be traversed. On average, about two links need to be traversed.)

Check your progress:

1. Give a precise expression for the minimum number of nodes in an AVL tree of height *h*.
2. What is the minimum number of nodes in an AVL tree of height 15?
3. Show the result of inserting 2, 1, 4, 5, 9, 3, 6, 7 into an initially empty AVL tree.
4. Keys 1, 2, . . . , 2^k -1 are inserted in order into an initially empty AVL tree. Prove that the resulting tree is perfectly balanced.
5. Write the procedures to implement AVL single and double rotations.

6. Write a nonrecursive function to insert into an AVL tree.
7. How can you implement (nonlazy) deletion in AVL trees?
8. How many bits are required per node to store the height of a node in an n-node AVL tree?
9. What is the smallest AVL tree that overflows an 8-bit height counter?

4.5 SPLAY TREES:

A **splay tree** is a self-balancing binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log(n)) amortized time. For many non-uniform sequences of operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown. The splay tree was invented by Daniel Sleator and Robert Tarjan.

All normal operations on a binary search tree are combined with one basic operation, called *splaying*. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top. Alternatively, a top-down algorithm can combine the search and the tree reorganization into a single phase.

4.5.1 Advantages and disadvantages

Good performance for a splay tree depends on the fact that it is self-balancing, and indeed self optimizing, in that frequently accessed nodes will move nearer to the root where they can be accessed more quickly. This is an advantage for nearly all practical applications, and is particularly useful for implementing caches and garbage collection algorithms.

Advantages include:

- Simple implementation -- simpler than other self-balancing binary search trees, such as red-black trees or AVL trees.
- Comparable performance -- average-case performance is as efficient as other trees.
- Small memory footprint -- splay trees do not need to store any bookkeeping data.

- Possible to create a persistent data structure version of splay trees -- which allows access to both the previous and new versions after an update. This can be useful in functional programming, and requires amortized O(log *n*) space per update.
- Work well with nodes containing identical keys -- contrary to other types of self balancing trees. Even with identical keys, performance remains amortized O(log *n*). All tree operations preserve the order of the identical nodes within the tree, which is a property similar to stable sorting algorithms. A carefully designed find operation can return the left most or right most node of a given key.

Disadvantages

- There could exist trees which perform "slightly" faster (a log(log(N))[?] factor) for a given distribution of input queries.
- Poor performance on uniform access (with workaround) -- a splay tree's performance will be considerably (although not asymptotically) worse than a somewhat balanced simple binary search tree for uniform access.

One worst case issue with the basic splay tree algorithm is that of sequentially accessing all the elements of the tree in the sorted order. This leaves the tree completely unbalanced (this takes n accesses, each an O(log *n*) operation). Reaccessing the first item triggers an operation that takes O(n) operations to rebalance the tree before returning the first item. This is a significant delay for that final operation, although the amortized performance over the entire sequence is actually O(log *n*). However, recent research shows that randomly rebalancing the tree can avoid this unbalancing effect and give similar performance to the other self-balancing algorithms.[

E-next THE NEXT LEVEL OF EDUCATION

4.5.2 Operations

4.5.2.1 Splaying

When a node *x* is accessed, a splay operation is performed on *x* to move it to the root. To perform a splay operation we carry out a sequence of *splay steps*, each of which moves *x* closer to the root. By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root and the tree remains roughly balanced, so that we achieve the desired amortized time bounds.

Each particular step depends on three factors:

- Whether *x* is the left or right child of its parent node, *p*,
- Whether *p* is the root or not, and if not
- Whether *p* is the left or right child of *its* parent, *g* (the *grandparent* of x).

The three types of splay steps are:

Zig Step: This step is done when *p* is the root. The tree is rotated on the edge between *x* and *p*. Zig steps exist to deal with the parity issue and will be done only as the last step in a splay operation and only when *x* has odd depth at the beginning of the operation.

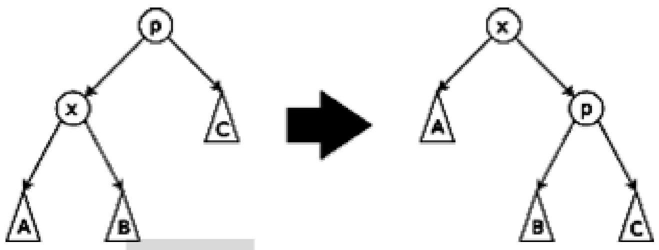


Figure 4.25 Zig Step

E-next THE NEXT LEVEL OF EDUCATION

Zig-zig Step: This step is done when *p* is not the root and *x* and *p* are either both right children or are both left children. The picture below shows the case where *x* and *p* are both left children. The tree is rotated on the edge joining *p* with *its* parent *g*, then rotated on the edge joining *x* with *p*. Note that zig-zig steps are the only things that differentiate splay trees from the *rotate to root* method introduced by Allen and Munro prior to the introduction of splay trees.

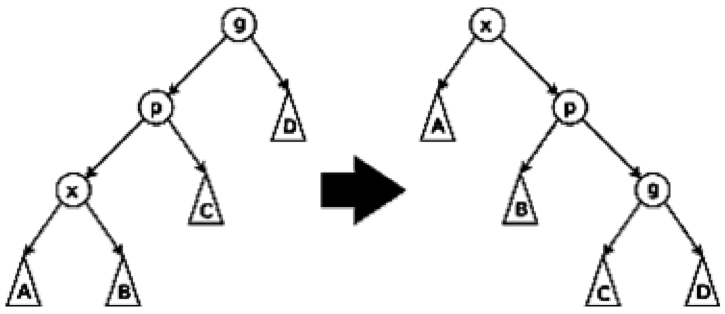


Figure 4.26 Zig-zig Step

Zig-zag Step: This step is done when *p* is not the root and *x* is a right child and *p* is a left child or vice versa. The tree is rotated on the edge between *x* and *p*, then rotated on the edge between *x* and its new parent *g*.

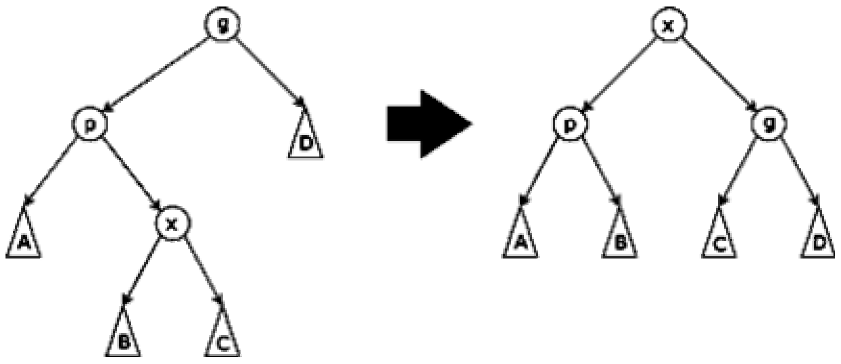


Figure 4.27 Zig-Zag Step

4.5.2.2 Insertion

The process of inserting a node x into a splay tree is *different* from inserting a node into a binary tree. The reason is that after insertion we want x to be the new root of the splay tree.

First, we search x in the splay tree. If x does not already exist, then we will not find it, but its parent node y. Second, we perform a splay operation on y which will move y to the root of the splay tree. Third, we insert the new node x as root in an appropriate way. In this way either y is left or right child of the new root x.

E-next THE NEXT LEVEL OF EDUCATION

4.5.2.3 Deletion

The deletion operation for splay trees is somewhat different than for binary or AVL trees. To delete an arbitrary node *x* from a splay tree, the following steps can be followed:

1. Splay node *x*, sending it to the root of the tree.

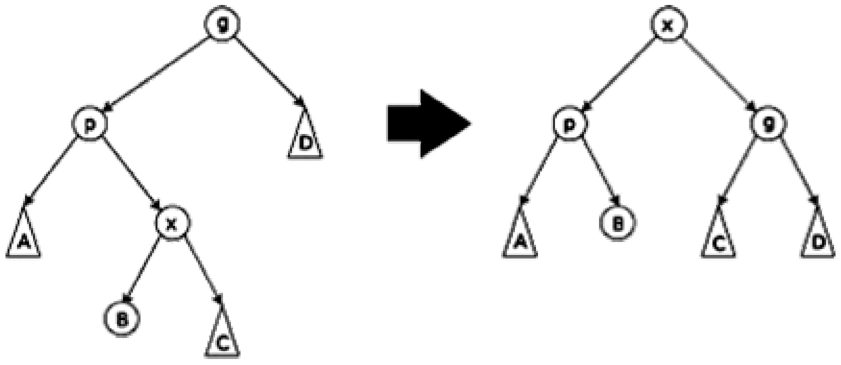


Figure 4.28 Deletion Step 1

2. Perform a left tree rotation on the first left child of *x* until the first left child of *x* has no right children.

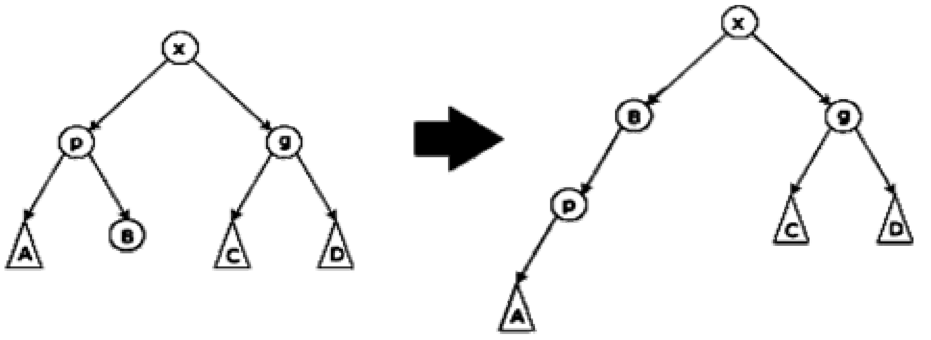
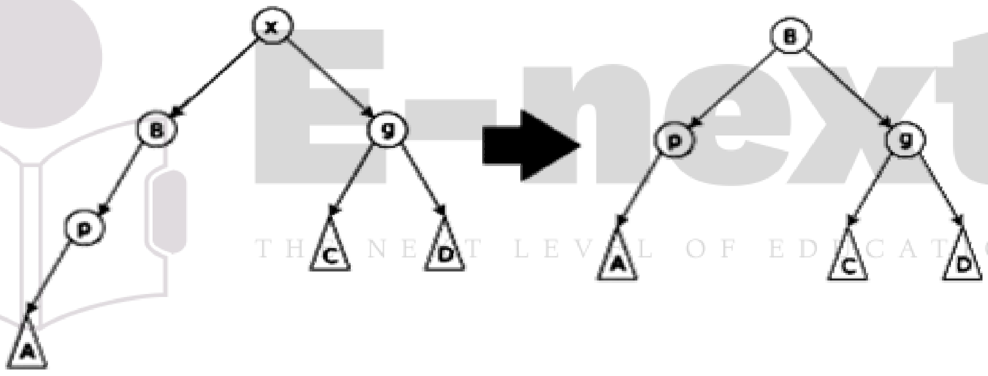


Figure 4.29 Deletion Step 2

3. Delete *x* from the tree & replace the root with the first left child of *x*.

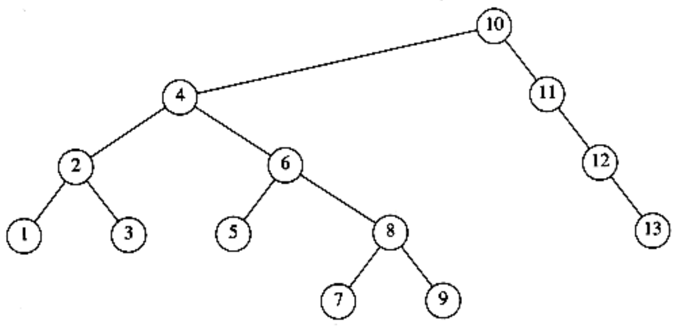


E-next
THE NEXT LEVEL OF EDUCATION

Figure 4.30 Deletion Step 3

Check your progress:

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



2. Nodes 1 through *n* = 1024 form a splay tree of left children.
    a. What is the internal path length of the tree (exactly)?
    b. Calculate the internal path length after each of find(1), find(2), find(3), find(4), find(5), find(6).
    c. If the sequence of successive finds is continued, when is the internal path length minimized?
3. Show that if all nodes in a splay tree are accessed in sequential order, the resulting tree consists of a chain of left children.
4. Write a program to perform random operations on splay trees. Count the total number of rotations performed over the sequence. How does the running time compare to AVL trees and unbalanced binary search trees?

4.6 THE SEARCH TREE ADT-BINARY SEARCH TREES

An important application of binary trees is their use in searching. Let us assume that each node in the tree is assigned a key value. In our examples, we will assume for simplicity that these are integers, although arbitrarily complex keys are allowed. We will also assume that all the keys are distinct.

Enext
THE NEXT LEVEL OF EDUCATION

The property that makes a binary tree into a binary search tree is that for every node, *X*, in the tree, the values of all the keys in the left subtree are smaller than the key value in *X*, and the values of all the keys in the right subtree are larger than the key value in *X*. Notice that this implies that all the elements in the tree can be ordered in some consistent manner. In Figure 4.31, the tree on the left is a binary search tree, but the tree on the right is not. The tree on the right has a node with key 7 in the left subtree of a node with key 6 (which happens to be the root).

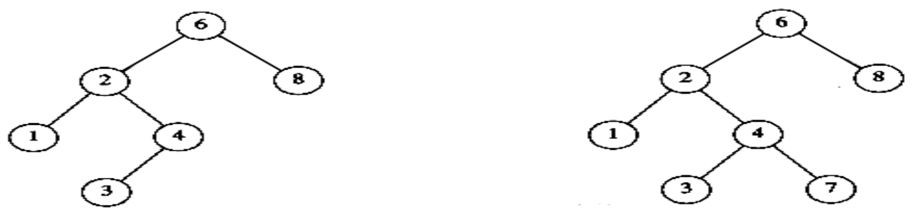


Figure 4.31 Two binary trees (only the left tree is a search tree)

We now give brief descriptions of the operations that are usually performed on binary search trees. Note that because of the recursive definition of trees, it is common to write these routines recursively. Because the average depth of a binary search tree is *O*(log *n*), we generally do not need to worry about running out of stack space. We repeat our type definition in Figure 4.32. Since all the elements can be ordered, we will assume that the operators <, >, and = can be applied to them, even if this might be syntactically erroneous for some types.

```
typedef struct tree_node *tree_ptr;
struct tree_node
{
        element_type element;
        tree_ptr left;
        tree_ptr right;
};
typedef tree_ptr SEARCH_TREE;
```

Figure 4.32 Binary search tree declarations

E-next
THE NEXT LEVEL OF EDUCATION

4.6.1. Make_null

This operation is mainly for initialization. Some programmers prefer to initialize the first element as a one-node tree, but our implementation follows the recursive definition of trees more closely. It is also a simple routine, as evidenced by Figure 4.33

```
SEARCH_TREE
make_null ( void )
{
        return NULL;
}
```

Figure 4.33 Routine to make an empty tree

4.6.2. Find

This operation generally requires returning a pointer to the node in tree *T* that has key *x*, or *NULL* if there is no such node. The structure of the tree makes this simple. If *T* is *, then we can just return* . Otherwise, if the key stored at *T* is *x*, we can return *T*. Otherwise, we make a recursive call on a subtree of *T*, either left or right, depending on the relationship of *x* to the key stored in *T*. The code in Figure 4.34 is an implementation of this strategy.

```
tree_ptr
find( element_type x, SEARCH_TREE T )
{
        if( T == NULL )
                return NULL;
        if( x < T->element )
                return( find( x, T->left ) );
        else
                if( x > T->element )
                    return( find( x, T->right ) );
                else
                    return T;
}
```

Figure 4.34 Find operation for binary search trees

E-next THE NEXT LEVEL OF EDUCATION

Notice the order of the tests. It is crucial that the test for an empty tree be performed first, since otherwise the indirections would be on a *NULL* pointer. The remaining tests are arranged with the least likely case last. Also note that both recursive calls are actually tail recursions and can be easily removed with an assignment and a *goto*. The use of tail recursion is justifiable here because the simplicity of algorithmic expression compensates for the decrease in speed, and the amount of stack space used is expected to be only $O(\log n)$.

4.6.3. Find_min and find_max

These routines return the position of the smallest and largest elements in the tree, respectively. Although returning the exact values of these elements might seem more reasonable, this would be inconsistent with the *find* operation. It is important that similar-looking operations do similar things. To perform a *find_min*, start at the root and go left as long as there is a left child. The stopping point is the smallest element. The *find_max* routine is the same, except that branching is to the right child.

This is so easy that many programmers do not bother using recursion. We will code the routines both ways by doing *find_min* recursively and *find_max* nonrecursively (see Figs. 4.35 and 4.36).

Notice how we carefully handle the degenerate case of an empty tree. Although this is always important to do, it is especially crucial in recursive programs. Also notice that it is safe to change *T* in *find_max*, since we are only working with a copy. Always be extremely careful, however, because a statement such as *T -> right* : =*T -> right -> right* will make changes in most languages.

```
tree_ptr
find_min( SEARCH_TREE T )
{
        if( T == NULL )
                return NULL;
        else
                if( T->left == NULL )
                        return( T );
                else
                        return( find_min ( T->left ) );
}
```

Figure 4.35 Recursive implementation of find_min for binary search trees

```
tree_ptr
find_max( SEARCH_TREE T )
{
        if( T != NULL )
                while( T->right != NULL )
                        T = T->right;
        return T;
}
```

E-next
THE NEXT LEVEL OF EDUCATION

Figure 4.36 Nonrecursive implementation of find_max for binary search trees

4.6.4. Insert

The insertion routine is conceptually simple. To insert *x* into tree *T*, proceed down the tree as you would with a *find*. If *x* is found, do nothing (or "update" something). Otherwise, insert *x* at the last spot on the path traversed. Figure 4.37 shows what happens. To insert 5, we traverse the tree as though a *find* were occurring. At the node with key 4, we need to go right, but there is no subtree, so 5 is not in the tree, and this is the correct spot.

Duplicates can be handled by keeping an extra field in the node record indicating the frequency of occurrence. This adds some extra space to the entire tree, but is better than putting duplicates in the tree (which tends to make the tree very deep). Of course this strategy does not work if the key is only part of a larger record. If that is the case, then we can keep all of the records that have the same key in an auxiliary data structure, such as a list or another search tree.

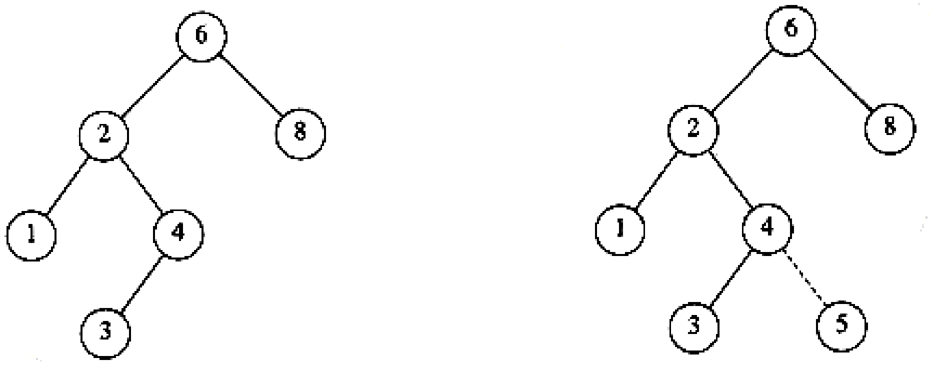


Figure 4.37 Binary search trees before and after inserting 5

Figure 4.38 shows the code for the insertion routine. Since *T* points to the root of the tree, and the root changes on the first insertion, *insert* is written as a function that returns a pointer to the root of the new tree. Lines 8 and 10 recursively insert and attach *x* into the appropriate subtree.

E-next
THE NEXT LEVEL OF EDUCATION

```
tree_ptr
insert( element_type x, SEARCH_TREE T )
{
/*1*/       if( T == NULL )
{  /* Create and return a one-node tree */
/*2*/            T = (SEARCH_TREE) malloc ( sizeof (struct tree_node)
);
/*3*/            if( T == NULL )
/*4*/                fatal_error("Out of space!!!");
else
{
/*5*/                T->element = x;
/*6*/                T->left = T->right = NULL;
}
}
else
/*7*/       if( x < T->element )
/*8*/            T->left = insert( x, T->left );
else
/*9*/       if( x > T->element )
/*10*/           T->right = insert( x, T->right );
/* else x is in the tree already. We'll do nothing */
/*11*/      return T; /* Don't forget this line!! */
}
```

Figure 4.38 Insertion into a binary search tree

4.6.5. Delete

As is common with many data structures, the hardest operation is deletion. Once we have found the node to be deleted, we need to consider several possibilities.

If the node is a leaf, it can be deleted immediately. If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node (we will draw the pointer directions explicitly for clarity). See Figure 4.39. Notice that the deleted node is now unreferenced and can be disposed of only if a pointer to it has been saved.

The complicated case deals with a node with two children. The general strategy is to replace the key of this node with the smallest key of the right subtree (which is easily found) and recursively delete that node (which is now empty). Because the smallest node in the right subtree cannot have a left child, the second *delete* is an easy one. Figure 4.40 shows an initial tree and the result of a deletion. The node to be deleted is the left child of the root; the key value is 2. It is replaced with the smallest key in its right subtree (3), and then that node is deleted as before.

E-next THE NEXT LEVEL OF EDUCATION

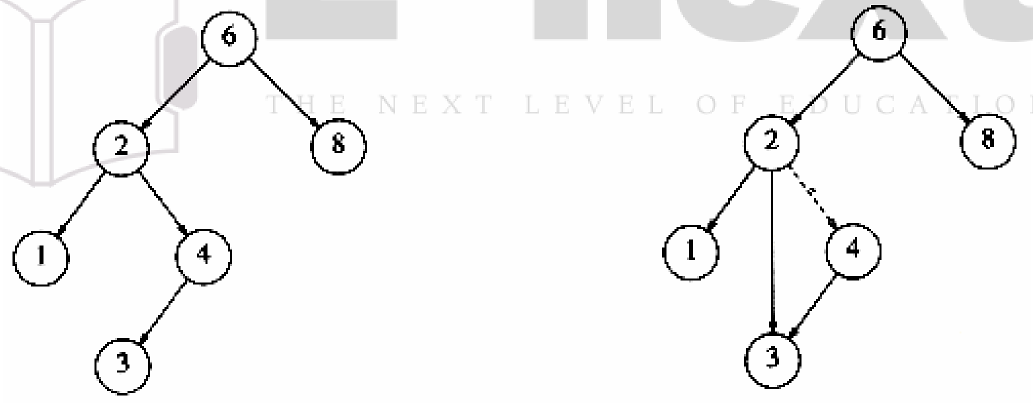


Figure 4.39 Deletion of a node (4) with one child, before and after

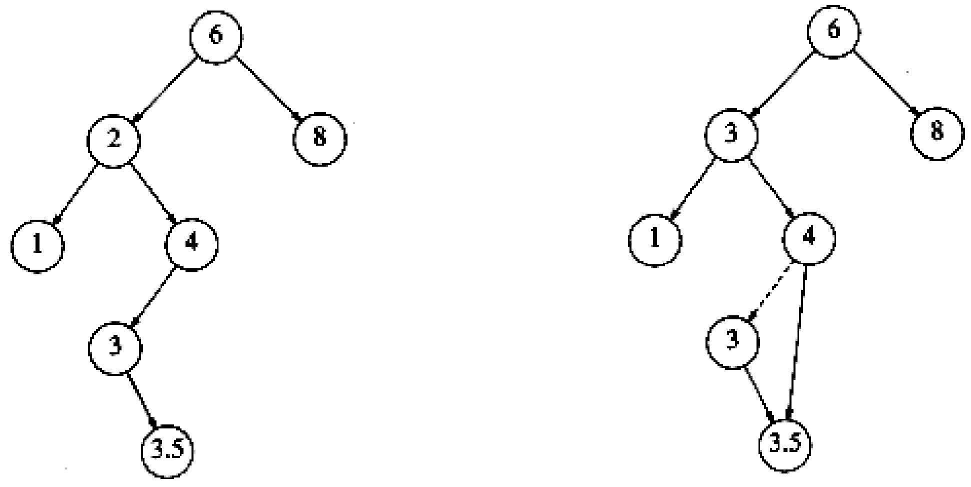


Figure 4.40 Deletion of a node (2) with two children, before and after

The code in Figure 4.41 performs deletion. It is inefficient, because it makes two passes down the tree to find and delete the smallest node in the right subtree when this is appropriate. It is easy to remove this inefficiency, by writing a special *delete_min* function, and we have left it in only for simplicity.

E-next
THE NEXT LEVEL OF EDUCATION

If the number of deletions is expected to be small, then a popular strategy to use is *lazy deletion:* When an element is to be deleted, it is left in the tree and merely *marked* as being deleted. This is especially popular if duplicate keys are present, because then the field that keeps count of the frequency of appearance can be decremented. If the number of real nodes in the tree is the same as the number of "deleted" nodes, then the depth of the tree is only expected to go up by a small constant, so there is a very small time penalty associated with lazy deletion. Also, if a deleted key is reinserted, the overhead of allocating a new cell is avoided.

```
tree_ptr
delete( element_type x, SEARCH_TREE T )
{
        tree_ptr tmp_cell, child;
        if( T == NULL )
                error("Element not found");
        else
                if( x < T->element )  /* Go left */
                        T->left = delete( x, T->left );
                else
```

```
                    if( x > T->element )  /* Go right */
                         T->right = delete( x, T->right );
                    else      /* Found element to be deleted */
                    if( T->left && T->right )  /* Two children */
                    {
                    /* Replace with smallest in right subtree */
                         tmp_cell = find_min( T->right );
                         T->element = tmp_cell->element;
                         T->right = delete( T->element, T->right );
                    }
                    else      /* One child */
                    {
                         tmp_cell = T;
                         if( T->left == NULL )
                         /* Only a right child */
                         child = T->right;
                         if( T->right == NULL )
                         /* Only a left child */
                         child = T->left;
                         free( tmp_cell );
                         return child;
                    }
       return T;
}
```

Figure 4.41 Deletion routine for binary search trees

E-next THE NEXT LEVEL OF EDUCATION

Check your progress:

1. Show the result of inserting 3, 1, 4, 6, 9, 2, 5, 7 into an initially empty binary search tree.
2. Show the result of deleting the root.
3. Write routines to implement the basic binary search tree operations.
4. Binary search trees can be implemented with cursors, using a strategy similar to a cursor linked list implementation. Write the basic binary search tree routines using a cursor implementation.

5. Write a program to evaluate empirically the following strategies for deleting nodes with two children:

   a. Replace with the largest node, *X*, in T_L and recursively delete *X*.

   b. Alternately replace with the largest node in T_L and the smallest node in T_R, and recursively delete appropriate node.

   c. Replace with either the largest node in T_L or the smallest node in T_R (recursively deleting the appropriate node), making the choice randomly. Which strategy seems to give the most balance? Which takes the least CPU time to process the entire sequence?

6. Write a routine to list out the nodes of a binary tree in *level-order*. List the root, then nodes at depth 1, followed by nodes at depth 2, and so on.

Let us Sum up:

We have seen uses of trees in operating systems, compiler design, and searching. Expression trees are a small example of a more general structure known as a *parse tree*, which is a central data structure in compiler design. Parse trees are not binary, but are relatively simple extensions of expression trees (although the algorithms to build them are not quite so simple).

E-next THE NEXT LEVEL OF EDUCATION

Search trees are of great importance in algorithm design. They support almost all the useful operations, and the logarithmic average cost is very small. Nonrecursive implementations of search trees are somewhat faster, but the recursive versions are sleeker, more elegant, and easier to understand and debug. The problem with search trees is that their performance depends heavily on the input being random. If this is not the case, the running time increases significantly, to the point where search trees become expensive linked lists.

We saw several ways to deal with this problem. AVL trees work by insisting that all nodes' left and right subtrees differ in heights by at most one. This ensures that the tree cannot get too deep. The operations that do not change the tree, as insertion does, can all use the standard binary search tree code. Operations that change the tree must restore the tree. This can be somewhat complicated, especially in the case of deletion.

We also examined the splay tree. Nodes in splay trees can get arbitrarily deep, but after every access the tree is adjusted in a somewhat mysterious manner.

In practice, the running time of all the balanced tree schemes is worse (by a constant factor) than the simple binary search tree, but this is generally acceptable in view of the protection being given against easily obtained worst-case input.

A final note: By inserting elements into a search tree and then performing an inorder traversal, we obtain the elements in sorted order. This gives an $O(n \log n)$ algorithm to sort, which is a worst-case bound if any sophisticated search tree is used.

References:

1. G. M. Adelson-Velskii and E. M. Landis, "An Algorithm for the Organization of Information," *Soviet Math. Doklady* 3.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
3. B. Allen and J. I. Munro, "Self Organizing Search Trees," *Journal of the ACM*, 25 (1978).
4. R. A. Baeza-Yates, "Expected Behaviour of B^+- trees under Random Insertions," *Acta Informatica* 26 (1989).
5. R. A. Baeza-Yates, "A Trivial Algorithm Whose Analysis Isn't: A Continuation," *BIT* 29 (1989).
6. R. Bayer and E. M. McGreight, "Organization and Maintenance of Large Ordered Indices," *Acta Informatica* 1 (1972).
7. J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM* 18 (1975).
8. Data structure – A Pseudocode Approach with C – Richard F Gilberg Behrouz A. Forouzan, Thomson
9. Schaum's Outlines Data structure Seymour Lipschutz Tata McGraw Hill 2nd Edition
10. Data structures & Program Design in C Robert Kruse, C. L.Tondo, Bruce Leung Pearson
11. "Data structure using C" AM Tanenbaum, Y Langsam & M J Augustein, Prentice Hall India

E-next

Question Pattern

1. What are trees? Explain the different methods of tree traversals.
2. What are expression trees? Explain with examples.
3. Explain the AVL tree in detail.
4. What are splay trees? Explain in detail.
5. Explain the Binary search trees in detail.

❖❖❖❖