제10장 Efficient Binary Search Tree

경북대학교 임경식 교수

10.1 Optimal binary search tree

we make no additions to or deletions from the set. Only searches are performed. Although the tree is a full binary tree, it may not be the optimal binary search tree to use when the probabilities with which different elements are searched are different. It is reasonable to use the level number of a node as its cost.

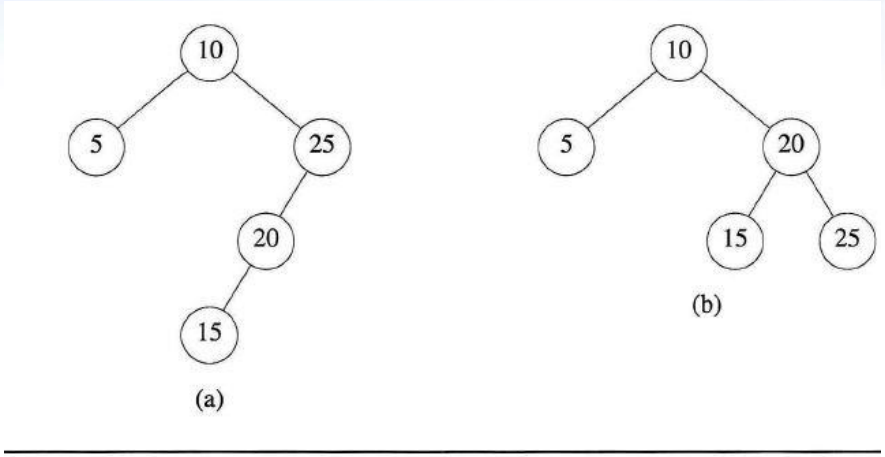


Figure 10.2: Two binary search trees

Assuming that each key is searched for with equal probability, the average number of comparisons for a successful search is 2.4. For the second binary search tree this amount is 2.2. Thus, the second tree has a better average behavior, too.

Suppose that each of 5, 10, 15, 20 and 25 is searched for with probablility 0.3, 0.3, 0.05, 0.05 and 0.3, respectively. The average number of comparisons for a successful search in the trees of Figure 10.2 (a) and (b) is 1.85 and 2.05, respectively. Now, the first tree has better average behavior than the second tree!

10.1 Optimal binary search tree

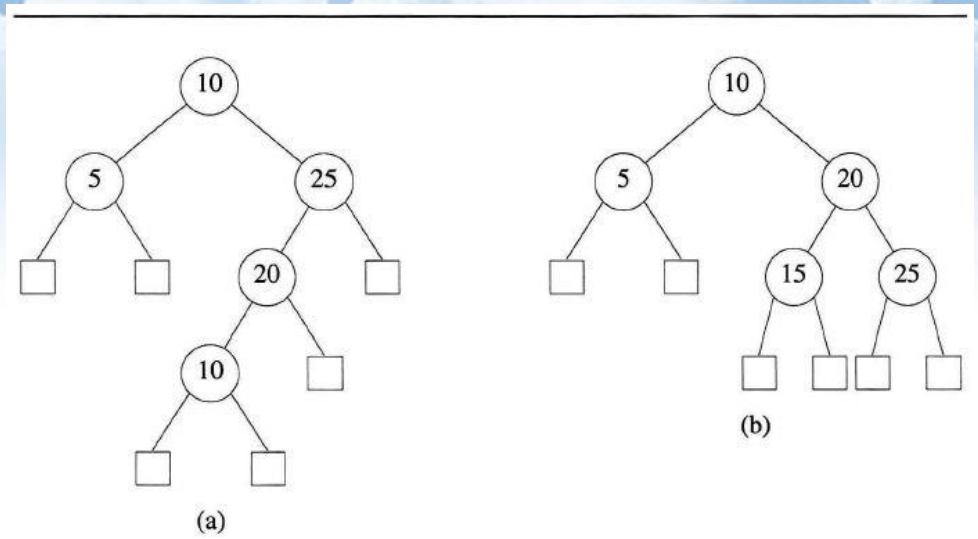


Figure 10.3: Extended binary trees corresponding to search trees of Figure 10.2

In evaluating binary search trees, it is useful to add a special "square" node at every null link. Remember that every binary tree with *n* nodes has *n* + 1 null links and therefore will have *n* + 1 square nodes. We shall call these nodes *external* nodes because they are not part of the original tree. The remaining nodes will be called *internal* nodes. Since all such searches are unsuccessful searches, external nodes will also be referred to as *failure nodes*. A binary tree with external nodes added is an *extended binary tree*.

10.1 Optimal binary search tree

We define the *external path length* of a binary tree to be the sum over all external nodes of the lengths of the paths from the root to those nodes. Analogously, the *internal path length* is the sum over all internal nodes of the lengths of the paths from the root to those nodes. The internal path length, *I,* for the tree of Figure 10.3(a) is I=0+1+1+2+3=7
Its external path length, *E,* is *E* = 2+ 2 + 4 + 4 + 3 + 2 = 17. The internal and external path lengths of a binary tree with *n* internal nodes are related by the formula *E = I + 2n.* Hence, binary trees with the maximum *E* also have maximum *I.*

Let us now return to our original problem of representing a static element set as a binary search tree. Let $a_1, a_2, \cdots, a_n$ with $a_1 < a_2 < \cdots < a_n$ be the element keys. Suppose that the probability of searching for each a_i is p_i. The total cost of any binary search tree for this set of keys is

$$\sum_{1 \le i \le n} p_i \cdot \text{level}(a_i)$$

when only successful searches are made. Since unsuccessful searches (i.e., searches for keys not in the table) will also be made, we should include the cost of these searches in our cost measure, too. Unsuccessful searches terminate with algorithm *iterSearch* (Program 5.17) returning a 0 pointer. Every node with an empty subtree defines a point at which such a termination can take place. Let us replace every empty subtree by a failure node. The keys not in the binary search tree may be partitioned into $n + 1$ classes E_i, $0 \le i \le n$. E_0 contains all keys X such that $X < a_1$. E_i contains all keys X such that

10.1 Optimal binary search tree

$a_i < X < a_{i+1}$, $1 \le i < n$, and E_n contains all keys X, $X > a_n$. It is easy to see that for all keys in a particular class E_i, the search terminates at the same failure node, and it terminates at different failure nodes for keys in different classes. The failure nodes may be numbered 0 to n, with i being the failure node for class E_i, $0 \le i \le n$. If q_i is the probability that the key being sought is in E_i, then the cost of the failure nodes is

$$\sum_{0 \le i \le n} q_i \cdot (\text{level(failure node } i) - 1)$$

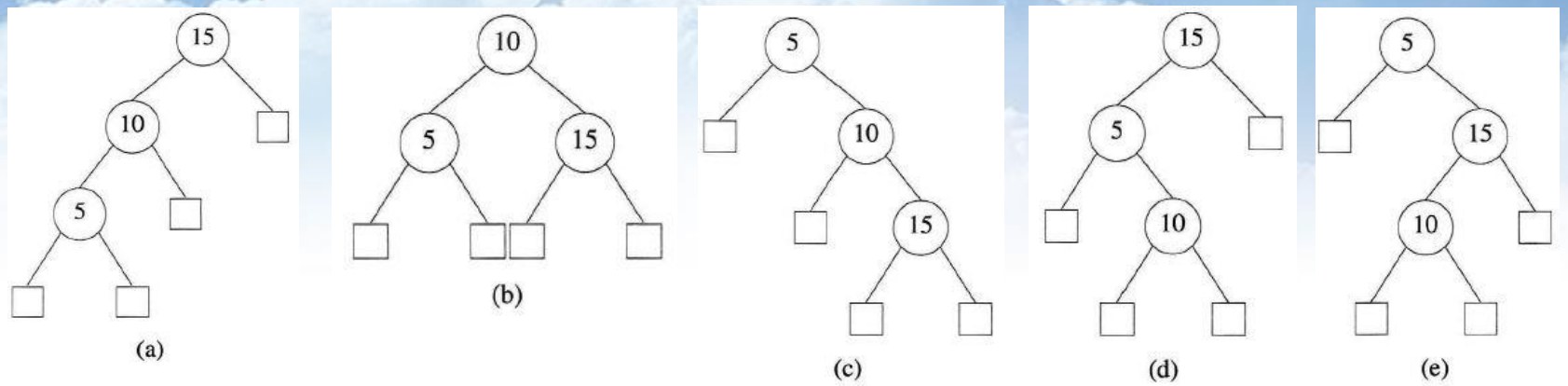
Therefore, the total cost of a binary search tree is

$$\sum_{1 \le i \le n} p_i \cdot \text{level}(a_i) + \sum_{0 \le i \le n} q_i \cdot (\text{level (failure node } i) - 1) \qquad (10.1)$$

An *optimal binary search tree* for $a_1, \cdots, a_n$ is one that minimizes Eq. (10.1) over all possible binary search trees for this set of keys. Note that since all searches must terminate either successfully or unsuccessfully, we have

$$\sum_{1 \le i \le n} p_i + \sum_{0 \le i \le n} q_i = 1$$

10.1 Optimal binary search tree



Example 10.2: Figure 10.4 shows the possible binary search trees for the key set (a_1,a_2,a_3)= (5, 10, 15). With equal probabilities, $p_i = q_j = 1/7$ for all i and j, we have

cost (tree a) = 15/7; cost (tree b) = 13/7
cost (tree c) = 15/7; cost (tree d) = 15/7
cost (tree e) = 15/7

As expected, tree b is optimal. With $p_1 = 0.5$, $p_2 = 0.1$, $p_3 = 0.05$, $q_0 = 0.15$, $q_1 = 0.1$, $q_2 = 0.05$, and $q_3 = 0.05$ we have

cost (tree a) = 2.65; cost (tree b) = 1.9
cost (tree c) = 1.5; cost (tree d) = 2.05
cost (tree e) = 1.6

Tree c is optimal with this assignment of p's and q's. □

10.1 Optimal binary search tree

How does one determine the optimal binary search tree? We could proceed as in Example 10.2 and explicitly generate all possible binary search trees, then compute the cost of each tree, and determine the tree with minimum cost. Since the cost of an n-node binary search tree can be determined in $O(n)$ time, the complexity of the optimal binary search tree algorithm is $O(n\,N(n))$, where $N(n)$ is the number of distinct binary search trees with n keys. From Section 5.11 we know that $N(n) = O(4^n/n^{3/2})$. Hence, this brute-force algorithm is impractical for large n. We can find a fairly efficient algorithm by making some observations about the properties of optimal binary search trees.

Let $a_1 < a_2 < \cdots < a_n$ be the n keys to be represented in a binary search tree. Let T_{ij} denote an optimal binary search tree for $a_{i+1}, \cdots, a_j$, $i < j$. By convention T_{ii} is an empty tree for $0 \le i \le n$, and T_{ij} is not defined for $i > j$. Let c_{ij} be the cost of the search tree T_{ij}. By definition c_{ii} will be 0. Let r_{ij} be the root of T_{ij}, and let

$$w_{ij} = q_i + \sum_{k=i+1}^{j} (q_k + p_k)$$

be the weight of T_{ij}. By definition $r_{ii} = 0$, and $w_{ii} = q_i$, $0 \le i \le n$. Therefore, T_{0n} is an optimal binary search tree for $a_1, \cdots, a_n$. Its cost is c_{0n}, its weight is w_{0n}, and its root is r_{0n}.

10.1 Optimal binary search tree

If T_{ij} is an optimal binary search tree for $a_{i+1}, \cdots, a_j$, and $r_{ij} = k$, then k satisfies the inequality $i < k \leq j$. T_{ij} has two subtrees L and R. L is the left subtree and contains the keys $a_{i+1}, \cdots, a_{k-1}$, and R is the right subtree and contains the keys $a_{k+1}, \cdots, a_j$ (Figure 10.5). The cost c_{ij} of T_{ij} is

$$c_{ij} = p_k + \text{cost}(L) + \text{cost}(R) + \text{weight}(L) + \text{weight}(R) \tag{10.2}$$

where weight (L) = weight $(T_{i,k-1}) = w_{i,k-1}$, and weight (R) = weight $(T_{kj}) = w_{kj}$.

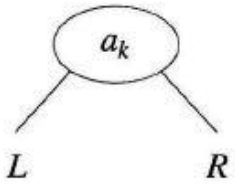


Figure 10.5: An optimal binary search tree T_{ij}

$$\begin{aligned} c_{ij} &= p_k + c_{i,k-1} + c_{kj} + w_{i,k-1} + w_{kj} \\ &= w_{ij} + c_{i,k-1} + c_{kj} \end{aligned} \tag{10.3}$$

$T_{ii} = \phi$ and $c_{ii} = 0$ ⇒ T_{0n} and c_{0n}

10.1 Optimal binary search tree

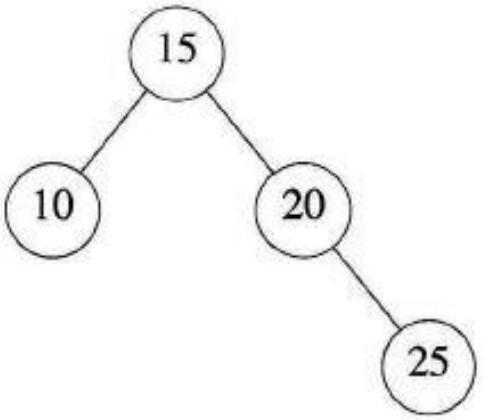
Example 10.3: Let $n = 4$ and $(a_1, a_2, a_3, a_4) = (10, 15, 20, 25)$. Let $(p_1, p_2, p_3, p_4) = (3, 3, 1, 1)$ and $(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$. The p's and q's have been multiplied by 16 for convenience. Initially, $w_{ii} = q_i$, $c_{ii} = 0$, and $r_{ii} = 0$, $0 \le i \le 4$. Using Eqs. (10.3) and (10.4), we get

$$
\begin{aligned}
w_{01} &= p_1 + w_{00} + w_{11} = p_1 + q_1 + w_{00} = 8 \\
c_{01} &= w_{01} + \min\{c_{00} + c_{11}\} = 8 \\
r_{01} &= 1 \\
w_{12} &= p_2 + w_{11} + w_{22} = p_2 + q_2 + w_{11} = 7 \\
c_{12} &= w_{12} + \min\{c_{11} + c_{22}\} = 7 \\
r_{12} &= 2 \\
w_{23} &= p_3 + w_{22} + w_{33} = p_3 + q_3 + w_{22} = 3 \\
c_{23} &= w_{23} + \min\{c_{22} + c_{33}\} = 3 \\
r_{23} &= 3 \\
w_{34} &= p_4 + w_{33} + w_{44} = p_4 + q_4 + w_{33} = 3 \\
c_{34} &= w_{34} + \min\{c_{33} + c_{44}\} = 3 \\
r_{34} &= 4
\end{aligned}
$$

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

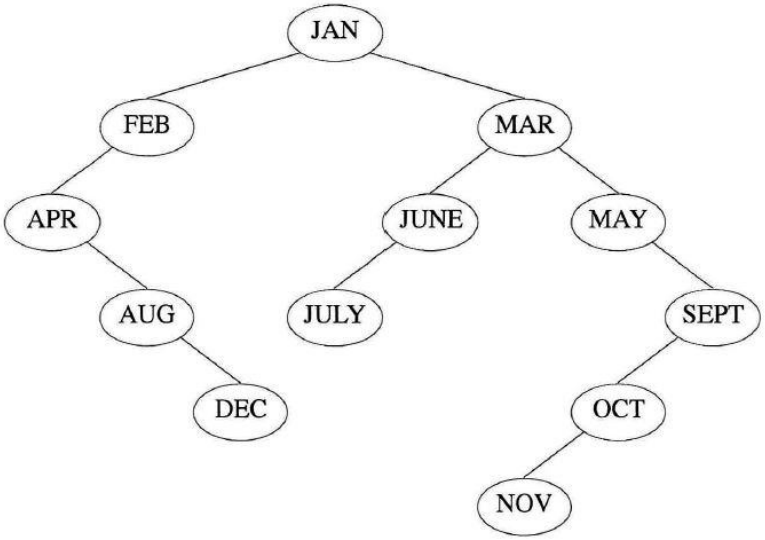
10.1 Optimal binary search tree

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				



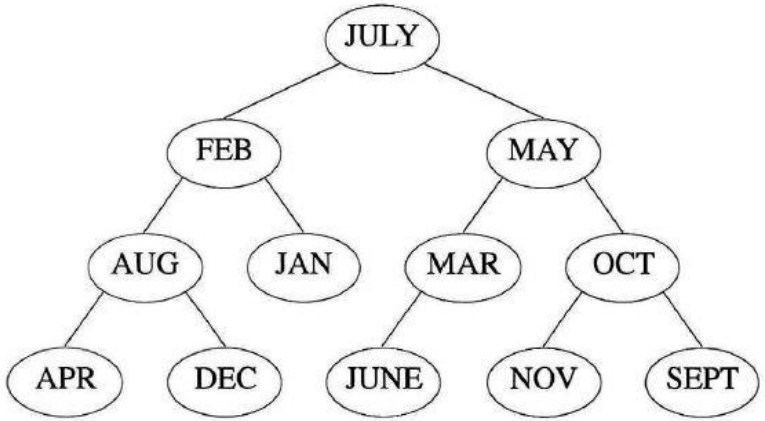
10.2 AVL trees

The binary search tree obtained by entering the months JANUARY to DECEMBER in that order into an initially empty binary search tree by using function *insert* (Program 5.21).



The average number of comparisons is (1 for JANUARY+ 2 each for FEBRUARY and MARCH+ 3 each for APRIL, JUNE and MAY+ · · · + 6 for NOVEMBER)/12 = 42/12 = 3.5.

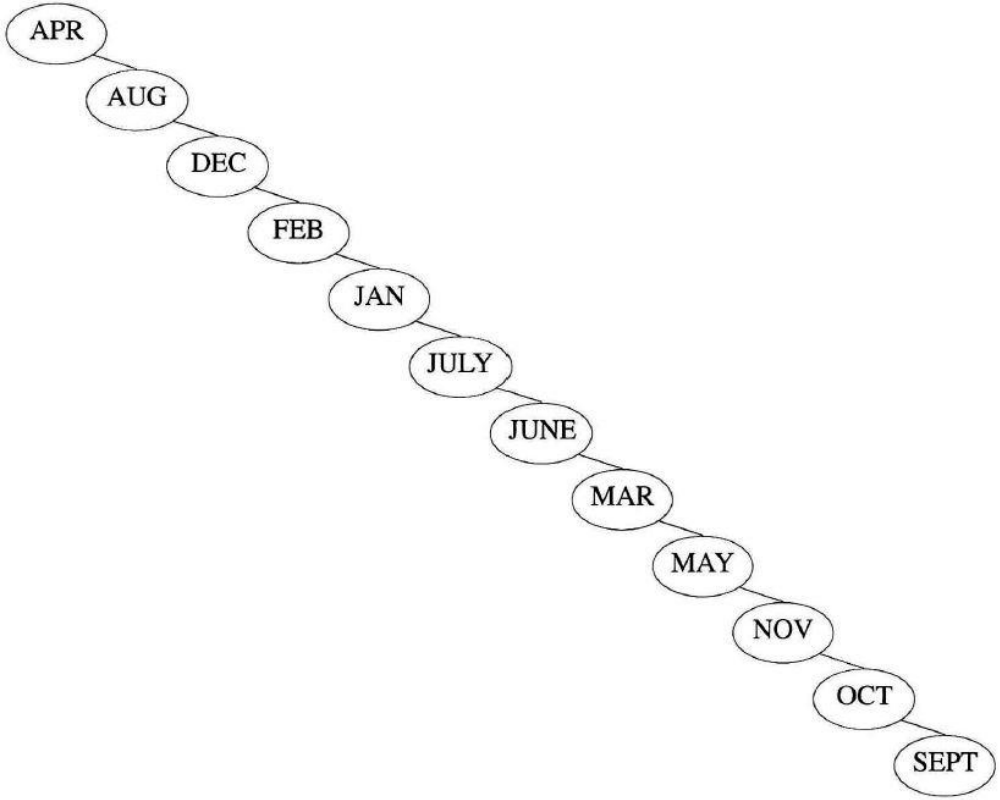
If the months are entered in the order JULY, FEBRUARY, MAY, AUGUST, DECEMBER, MARCH, OCTOBER, APRIL, JANUARY, JUNE, SEPTEMBER, NOVEMBER



The maximum number of key comparisons needed to find any key is now 4, and the average is 37/12 = 3 .1.

10.2 AVL trees

If the months are entered in lexicographic order, instead, the tree degenerates to a chain. Thus, in the worst case, searching a binary search tree corresponds to sequential searching in a sorted linear list.



10.2 AVL trees

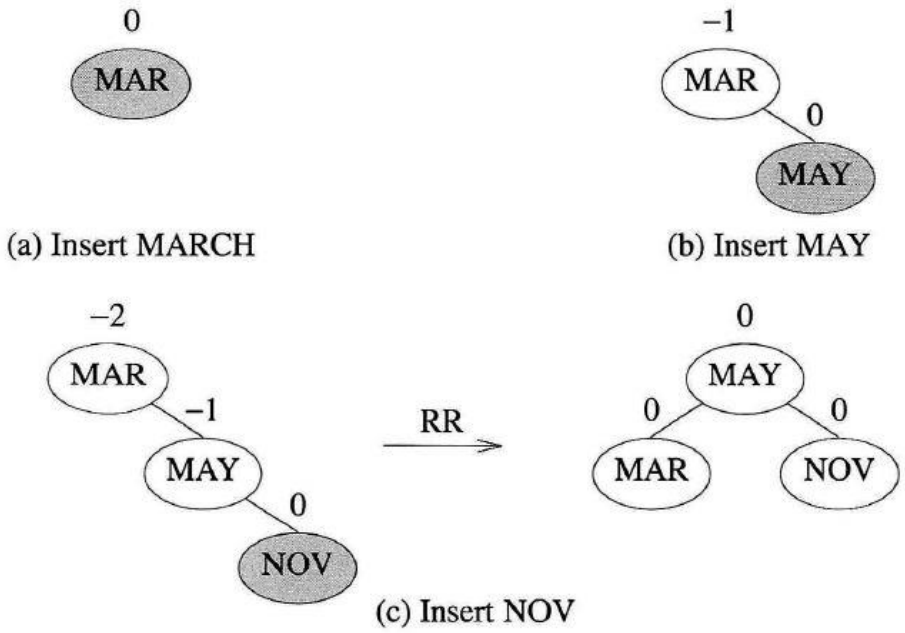
From our earlier study of binary trees, we know that both the average and maximum search time will be minimized if the binary search tree is maintained as a complete binary tree at all times. However, since we are dealing with a dynamic situation, it is difficult to achieve this ideal without making the time required to insert a key very high. This is so because in some cases it would be necessary to restructure the whole tree to accommodate the new entry and at the same time have a complete binary search tree. It is, however, possible to keep the tree balanced to ensure both an average and worst-case search time of $O(\log n)$ for a tree with n nodes. In this section, we study one method of growing balanced binary trees. These balanced trees will have satisfactory search, insertion and deletion time properties. Other ways to maintain balanced search trees are studied in later sections.

In 1962, Adelson-Velskii and Landis introduced a binary tree structure that is balanced with respect to the heights of subtrees. As a result of the balanced nature of this type of tree, dynamic retrievals can be performed in $O(\log n)$ time if the tree has n nodes in it. At the same time, a new key can be entered or deleted from such a tree in time $O(\log n)$. The resulting tree remains height-balanced. This tree structure is called an AVL tree. As with binary trees, it is natural to define AVL trees recursively.

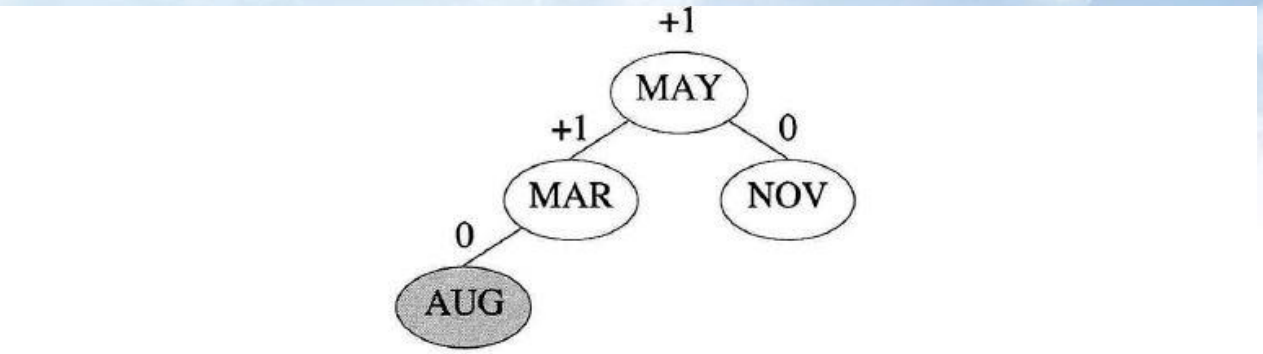
10.2 AVL trees

Definition: An empty tree is height-balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees respectively, then T is *height-balanced* iff (1) T_L and T_R are height-balanced and (2) $| h_L - h_R | \leq 1$ where h_L and h_R are the heights of T_L and T_R, respectively. □

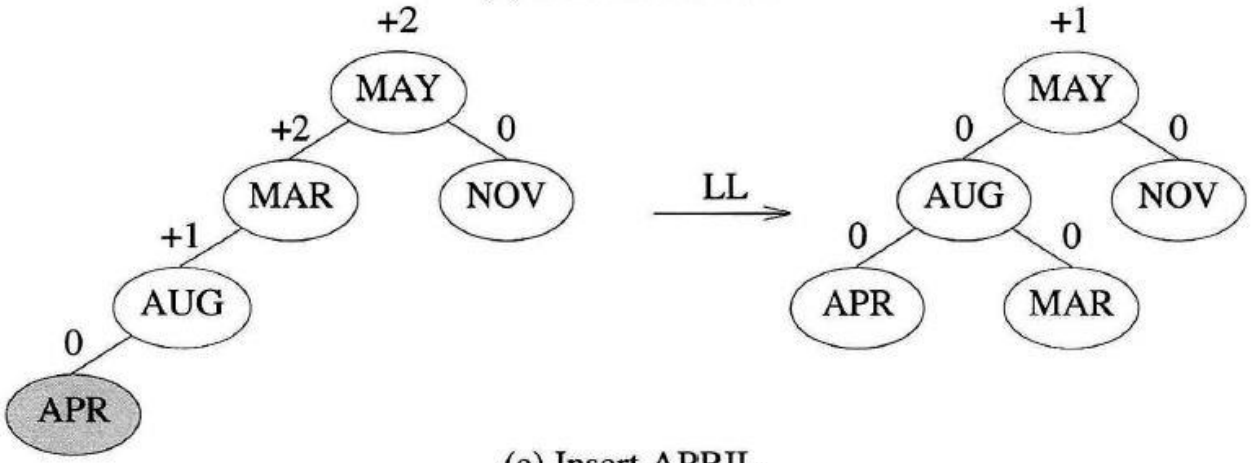
Let us assume that the insertions are made in the following order: MARCH, MAY, NOVEMBER, AUGUST, APRIL, JANUARY, DECEMBER, JULY, FEBRUARY, JUNE, OCTOBER, SEPTEMBER.



10.2 AVL trees

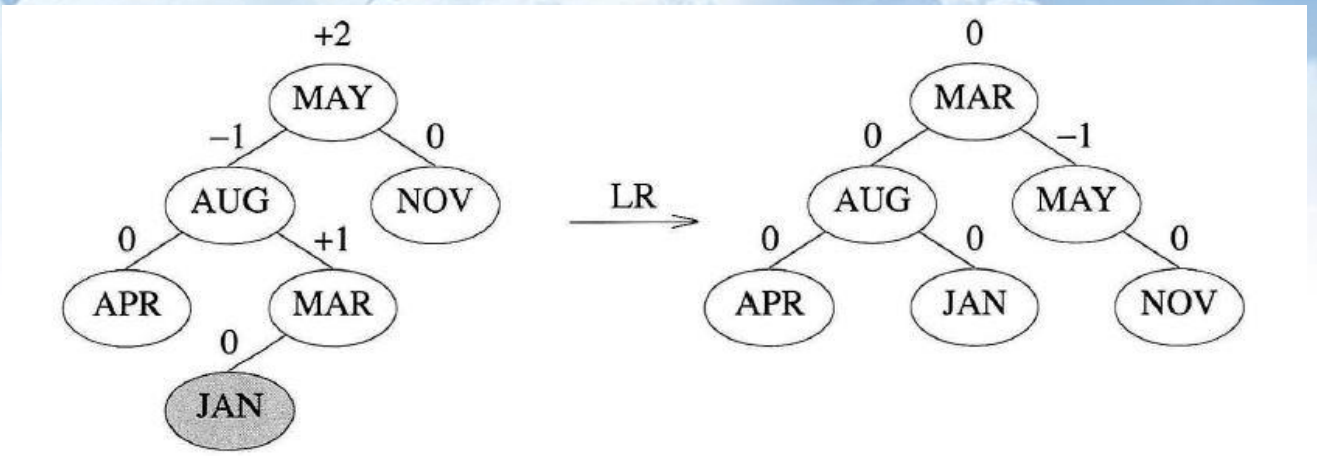


(d) Insert AUGUST

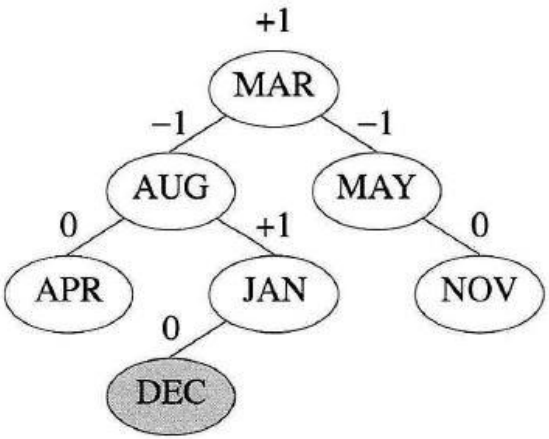


(e) Insert APRIL

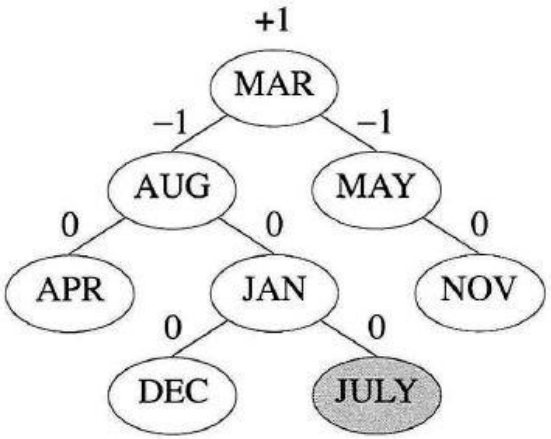
10.2 AVL trees



(f) Insert JANUARY

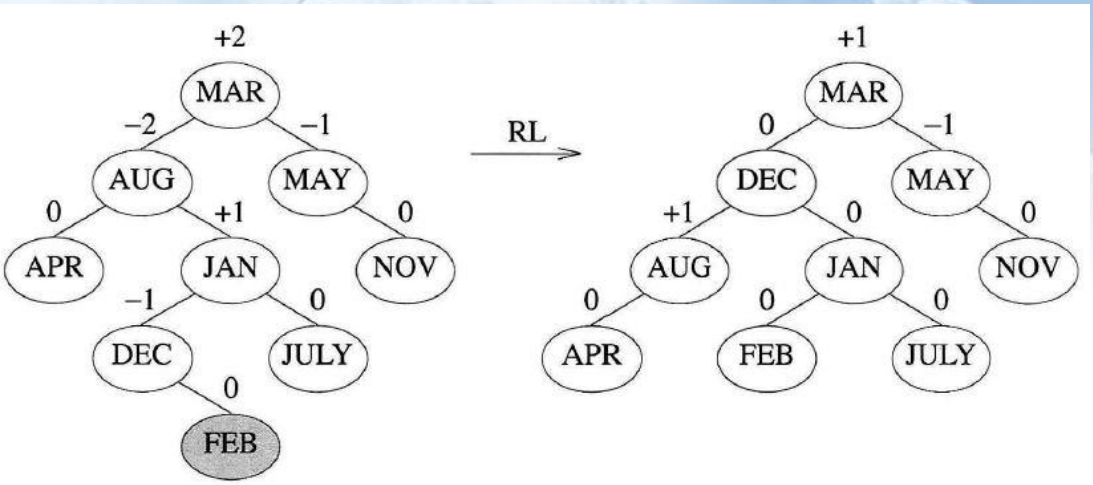


(g) Insert DECEMBER

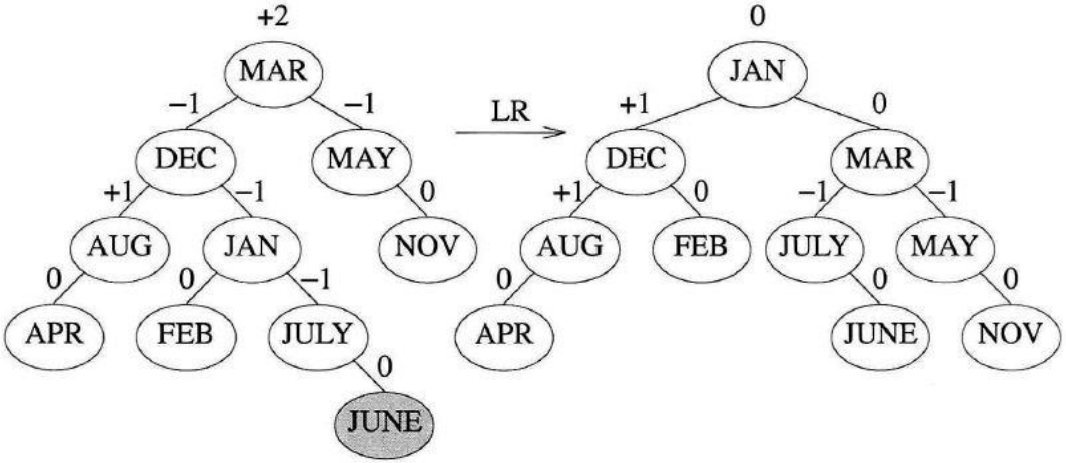


(h) Insert JULY

10.2 AVL trees

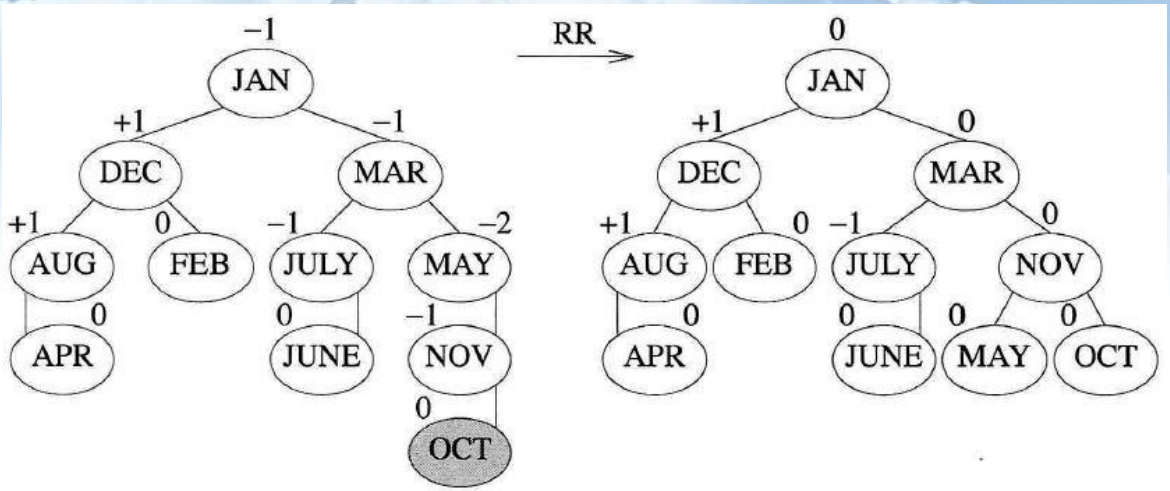


(i) Insert FEBRUARY

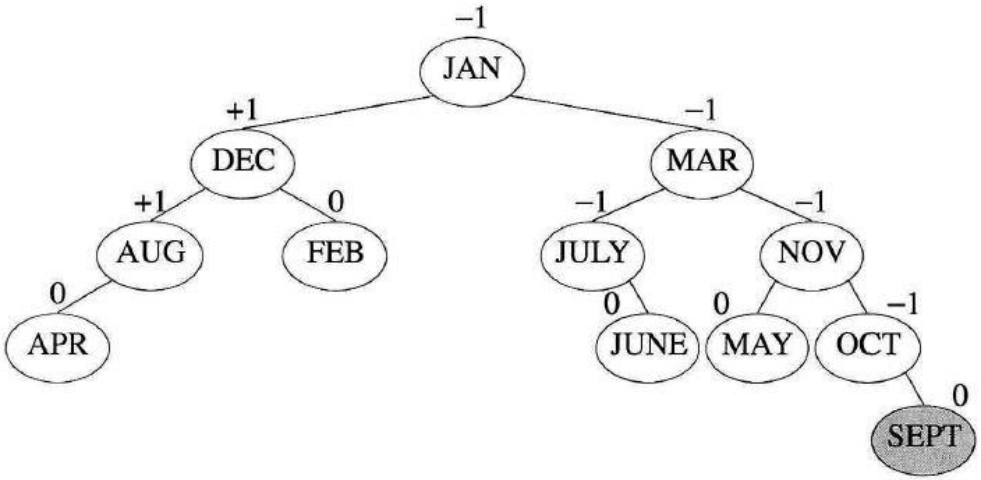


(j) Insert JUNE

10.2 AVL trees



(k) Insert OCTOBER



(l) Insert SEPTEMBER

10.2 AVL trees

In the preceding example we saw that the addition of a node to a balanced binary search tree could unbalance it. The rebalancing was carried out using four different kinds of rotations: LL, RR, LR, and RL (Figure 10.11 (e), (c), (f), and (i), respectively). LL and RR are symmetric, as are LR and RL. These rotations are characterized by the nearest ancestor, A, of the inserted node, Y, whose balance factor becomes ± 2. The following characterization of rotation types is obtained:

LL: new node Y is inserted in the left subtree of the left subtree of A

LR: Y is inserted in the right subtree of the left subtree of A

RR: Y is inserted in the right subtree of the right subtree of A

RL: Y is inserted in the left subtree of the right subtree of A

10.2 AVL trees

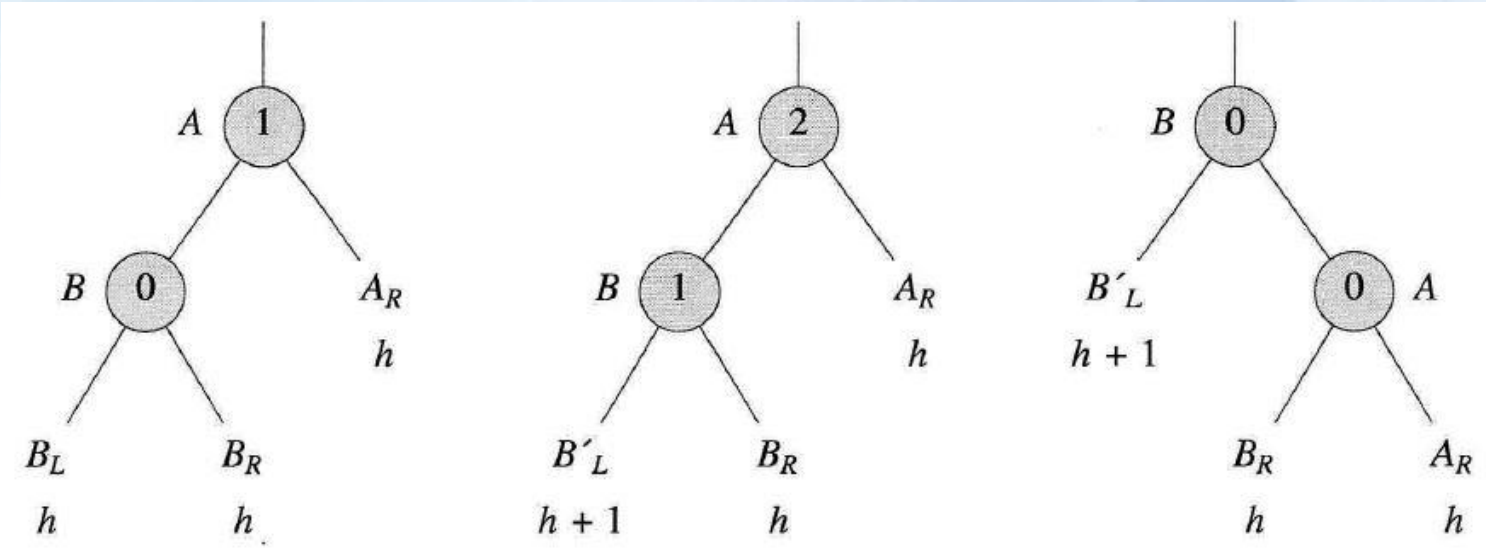
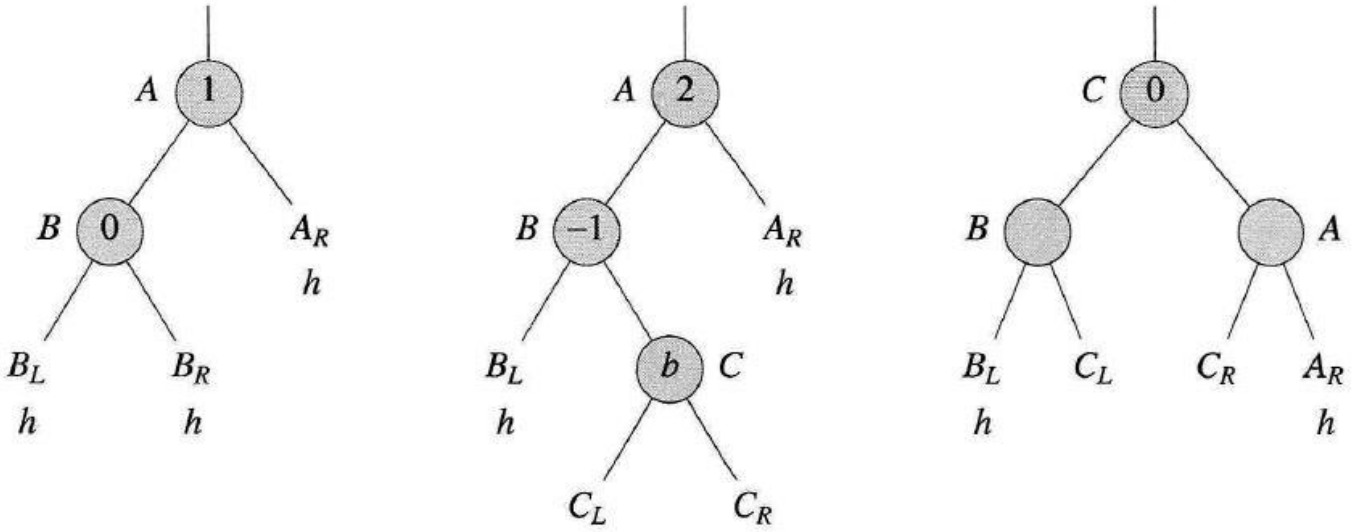


Figure 10.12: An LL rotation

10.2 AVL trees



(a) Before insertion (b) After inserting into B_R (c) After LR rotation

$b=0 \Rightarrow bf(B)=bf(A)=0$ after rotation
$b=1 \Rightarrow bf(B)=0$ and $bf(A)=-1$ after rotation
$b=-1 \Rightarrow bf(B)=1$ and $bf(A)=0$ after rotation

Figure 10.13: An LR rotation

10.2 AVL trees

```
typedef struct {
        int key;
        } element;
typedef struct treeNode *treePointer;
        struct treeNode {
                treePointer leftChild;
                element     data;
                short int   bf;
                treePointer rightChild;
                };
```

Thank You !

노력 없이 이룰 수 있는 것 아무것도 없다.