AVL Trees

Definition: A **perfectly balanced** binary tree is a binary tree such that:

i. The heights of the left and right subtrees of the root are equal.

ii. The left and right subtrees of the root are perfectly balanced binary trees.

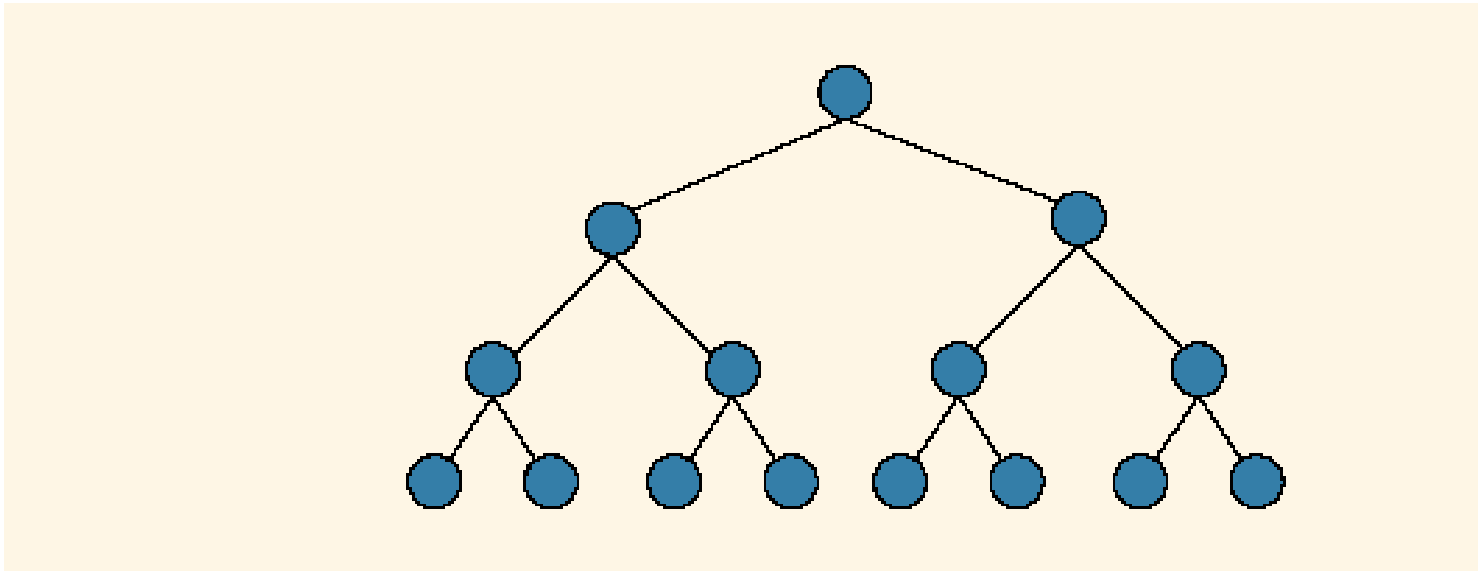


FIGURE AVL-1 Perfectly balanced binary tree

- Let *T* be a binary tree and *x* be a node in *T.*
- Then the height of the left subtree of *x* is the same as the height of the right subtree of *x*.
- It can be proved that if *T* is a perfectly balanced binary tree of height *h*, then the number of nodes in *T* is 2^h-1.
- If the number of items in the data set is not equal to 2^h-1, for some nonnegative integer *h*, then we cannot construct a perfectly balanced binary tree.

Definition: An **AVL tree** (or **height-balanced tree**) is a binary search tree such that:

i. The height of the left and right subtrees of the root differ by at most 1.
ii. The left and right subtrees of the root are AVL trees.

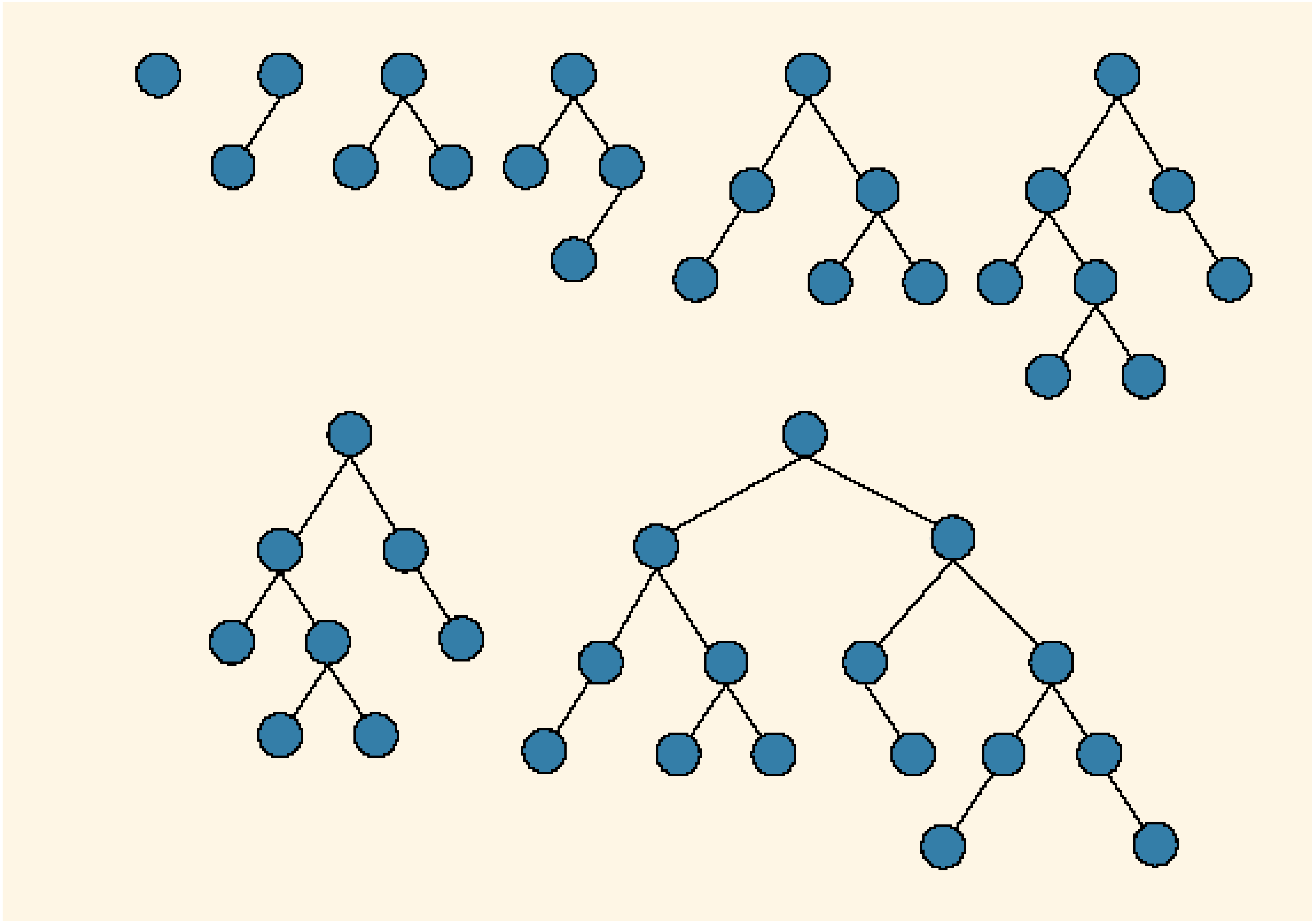


FIGURE AVL-2 AVL trees

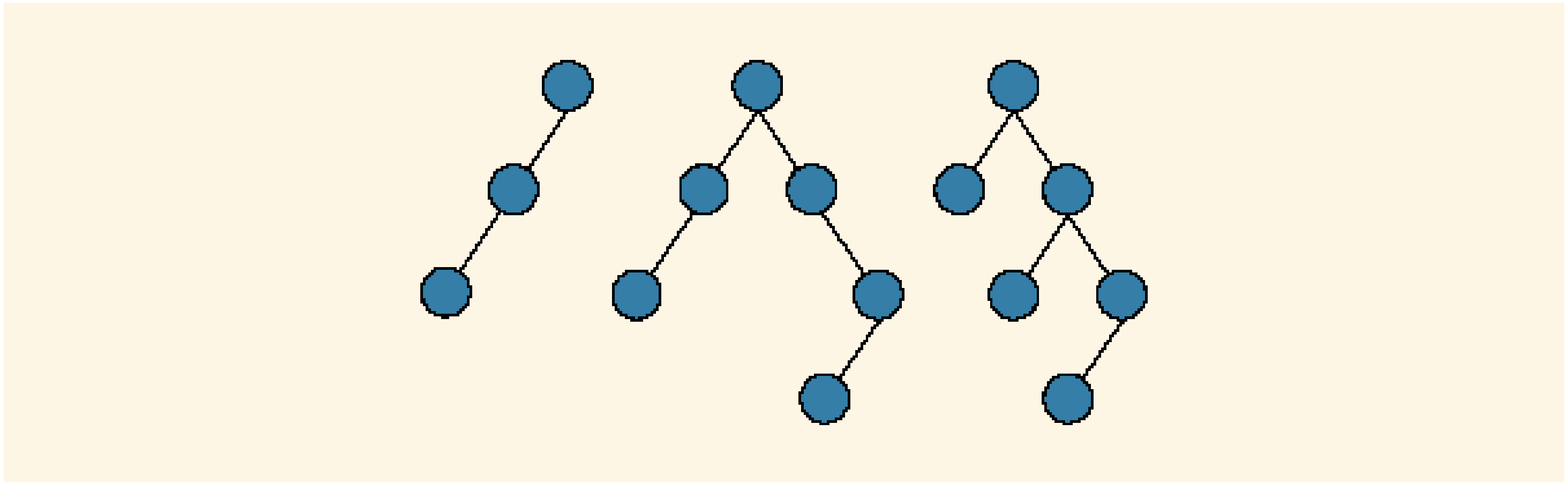


FIGURE AVL-3 Non-AVL trees

Theorem: Let T be an AVL tree and x be a node in T. Then

$$|x_r - x_l| \leq 1,$$

where $|x_r - x_l|$ denotes the absolute value of $x_r - x_l$.

Let x be a node in the AVL tree T.

1. If $x_l > x_r$, we say that x is **left high**. In this case, $x_l = x_r + 1$.
2. If $x_l = x_r$, we say that x is **equal high**.
3. If $x_r > x_l$, we say that x is **right high**. In this case, $x_r = x_l + 1$.

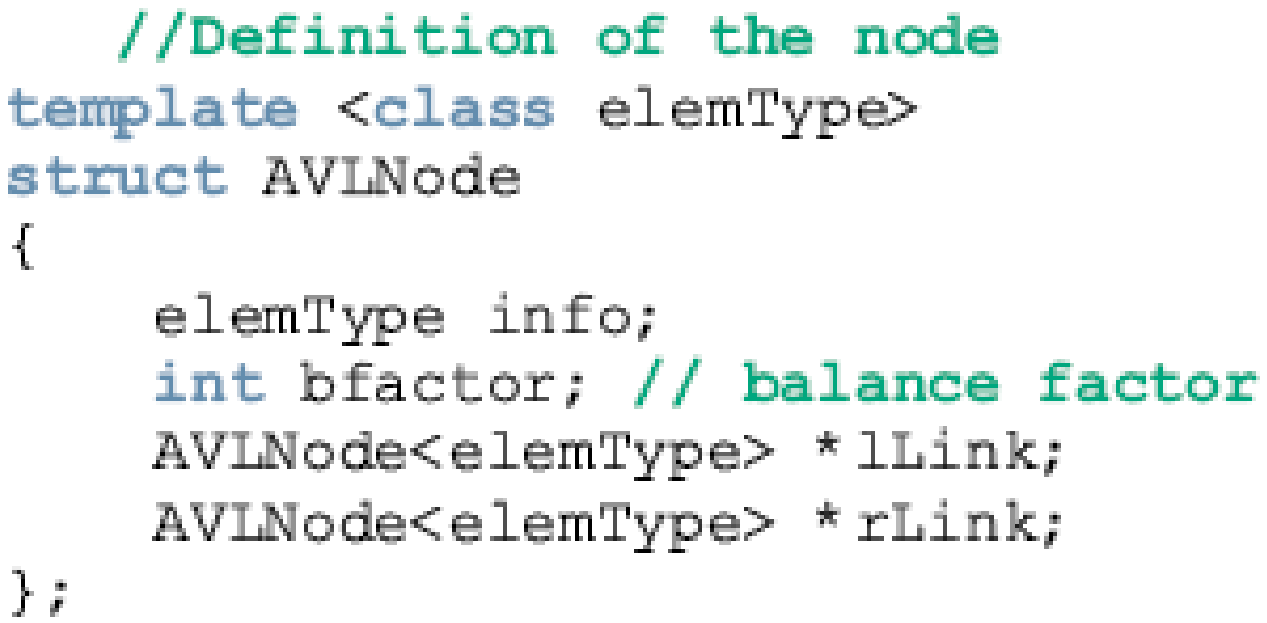
Definition: The **balance factor** of x, written $bf(x)$, is defined by $bf(x) = x_r - x_l$.

Let x be a node in the AVL tree T. Then:

1. If x is left high, then $bf(x) = -1$.
2. If x is equal high, then $bf(x) = 0$.
3. If x is right high, then $bf(x) = 1$.

Definition: Let *x* be a node in a binary tree. We say that the node *x* **violates the balance criteria** if $|x_r - x_l| > 1$, that is, if the height of the left and right subtrees of *x* differ by more than 1.

The following defines the node of an AVL tree:



```
   //Definition of the node
template <class elemType>
struct AVLNode
{
     elemType info;
     int bfactor; // balance factor
     AVLNode<elemType> *lLink;
     AVLNode<elemType> *rLink;
};
```

Insertion into AVL Trees

- To insert an item in an AVL tree, first we search the tree and find the place where the new item is to be inserted.
- Because an AVL tree is a binary search tree, to find the place for the new item we can search the AVL tree using a search algorithm similar to the one designed for binary search trees.
- If the item to be inserted is already in the tree, then the search ends at a nonempty subtree.
- Because duplicates are not allowed, in this case we can output an appropriate error message.
- Suppose that the item to be inserted is not in the AVL tree. Then the search ends at an empty subtree and we insert the item in that subtree.

- After inserting the new item in the tree, the resulting tree might not be an AVL tree.
- Thus, we must restore the tree’s balance criteria.
- This is accomplished by traveling the same path (back to the root node) that was followed when the new item was inserted in the AVL tree.
- The nodes on this path (back to the root node) are visited and either their balance factors are changed, or we might have to reconstruct part of the tree.

Consider the AVL tree of Figure AVL-4.

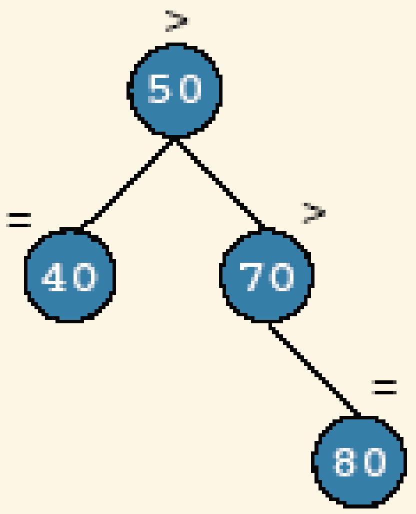


FIGURE AVL-4 AVL tree before inserting 90

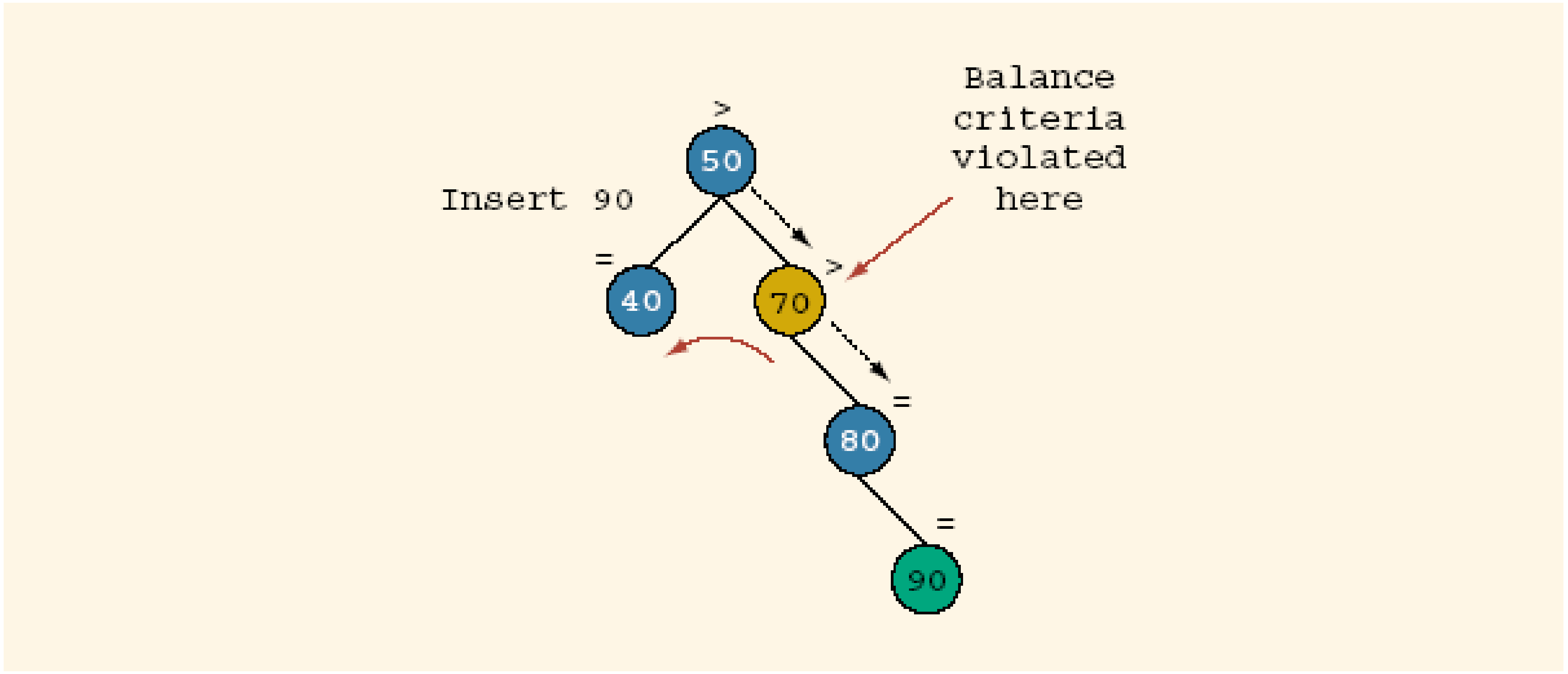


FIGURE AVL-5 Binary search tree of Figure AVL-4 after inserting 90; nodes other than 90 show their balance factors before insertion

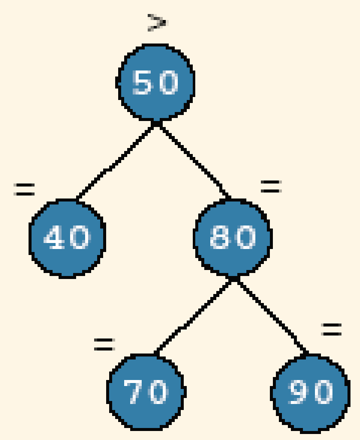


FIGURE AVL-6 AVL tree of Figure AVL-4 after inserting 90 and adjusting the balance factors

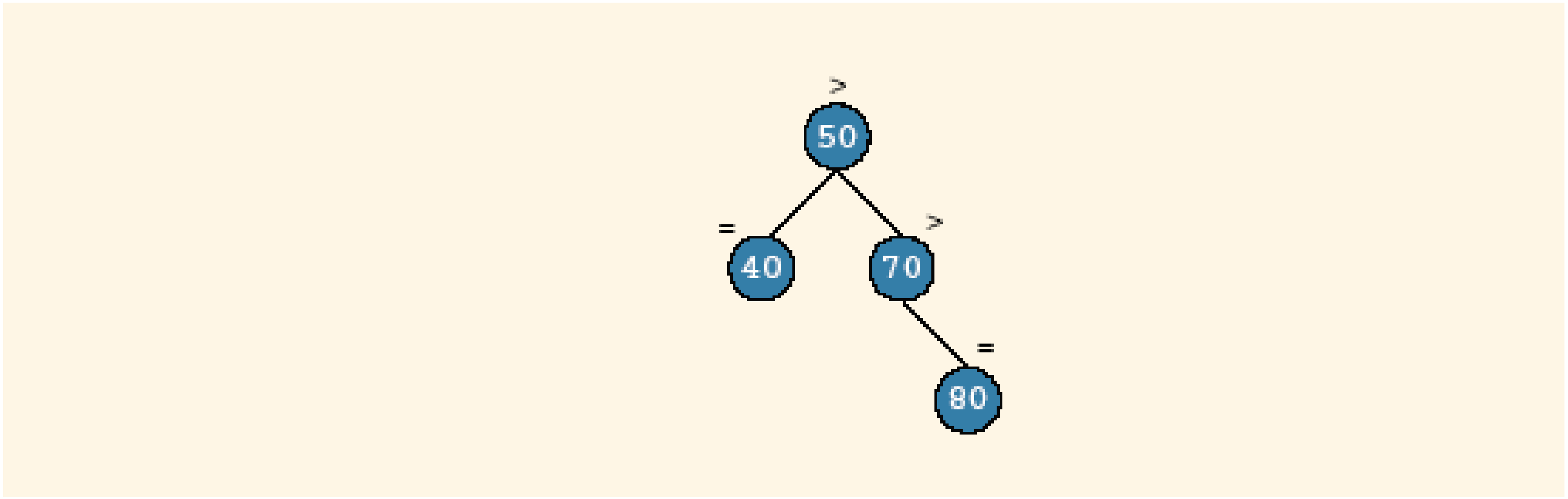


FIGURE AVL-7 AVL tree before inserting 75

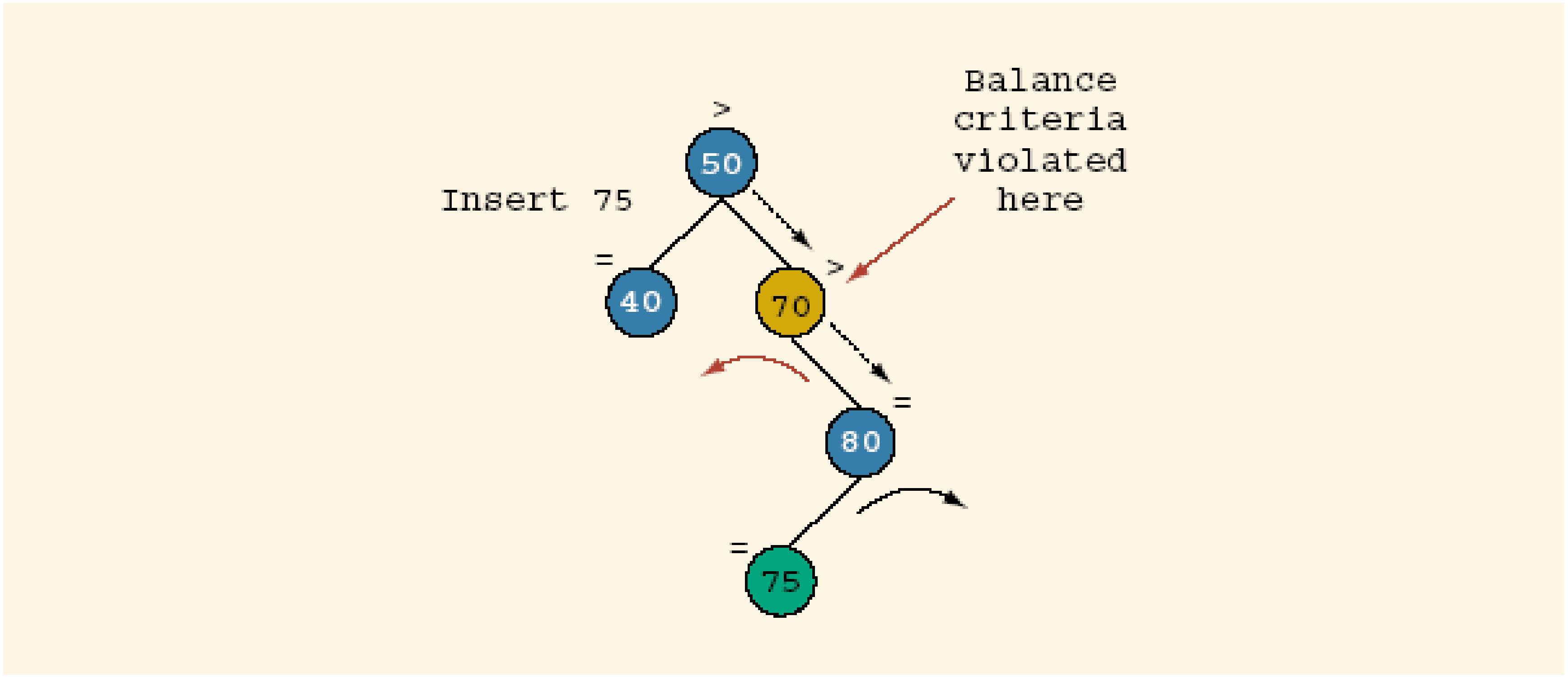


FIGURE AVL-8 Binary search tree of Figure AVL-7 after inserting 75; nodes other than 75 show their balance factors before insertion

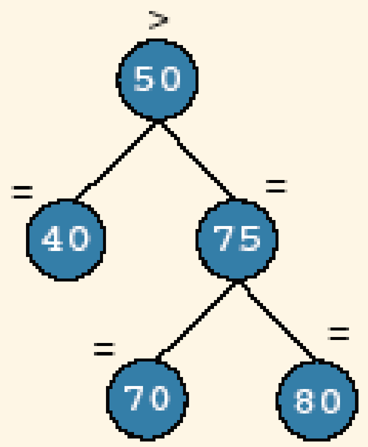


FIGURE AVL-9 AVL tree of Figure AVL-7 after inserting 75 and adjusting the balance factors

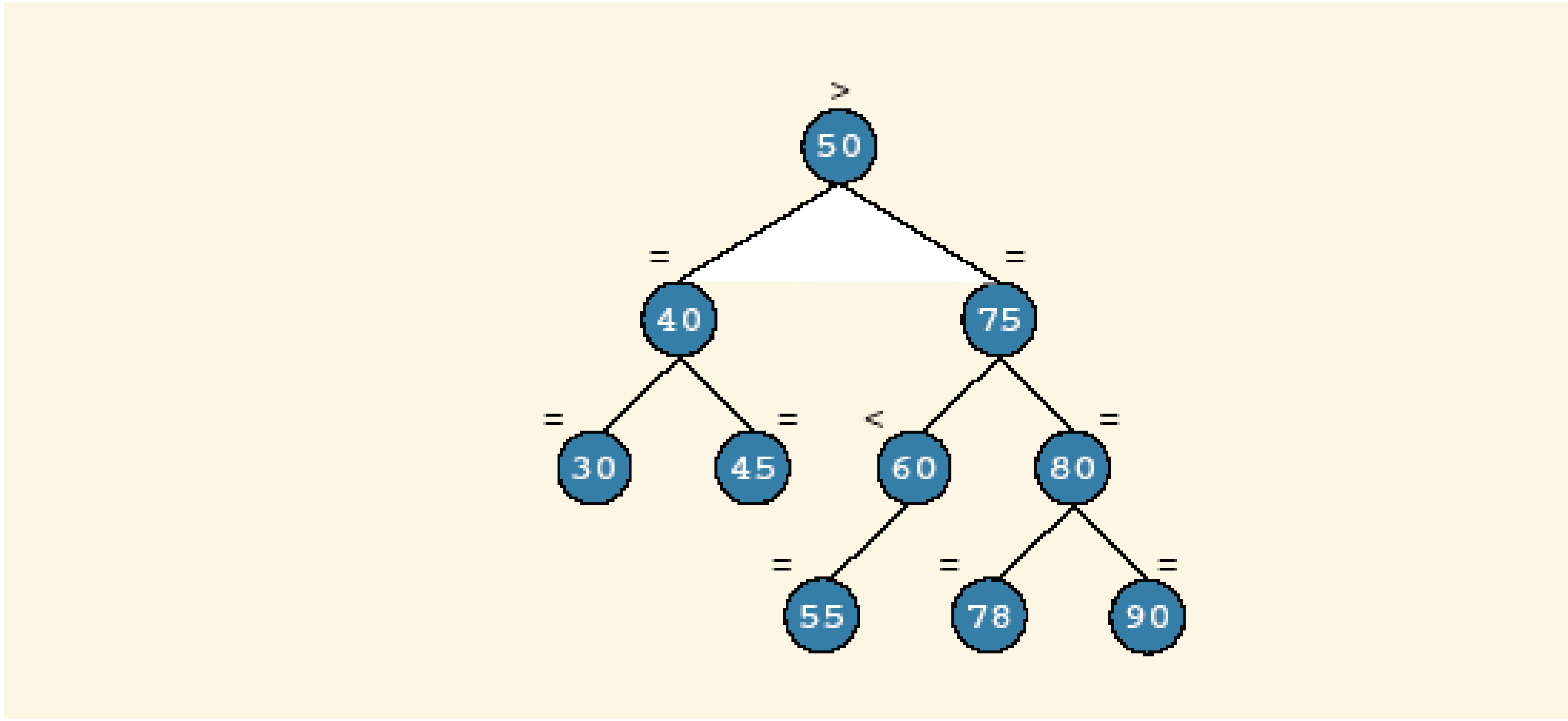


FIGURE AVL-10 AVL tree before inserting 95

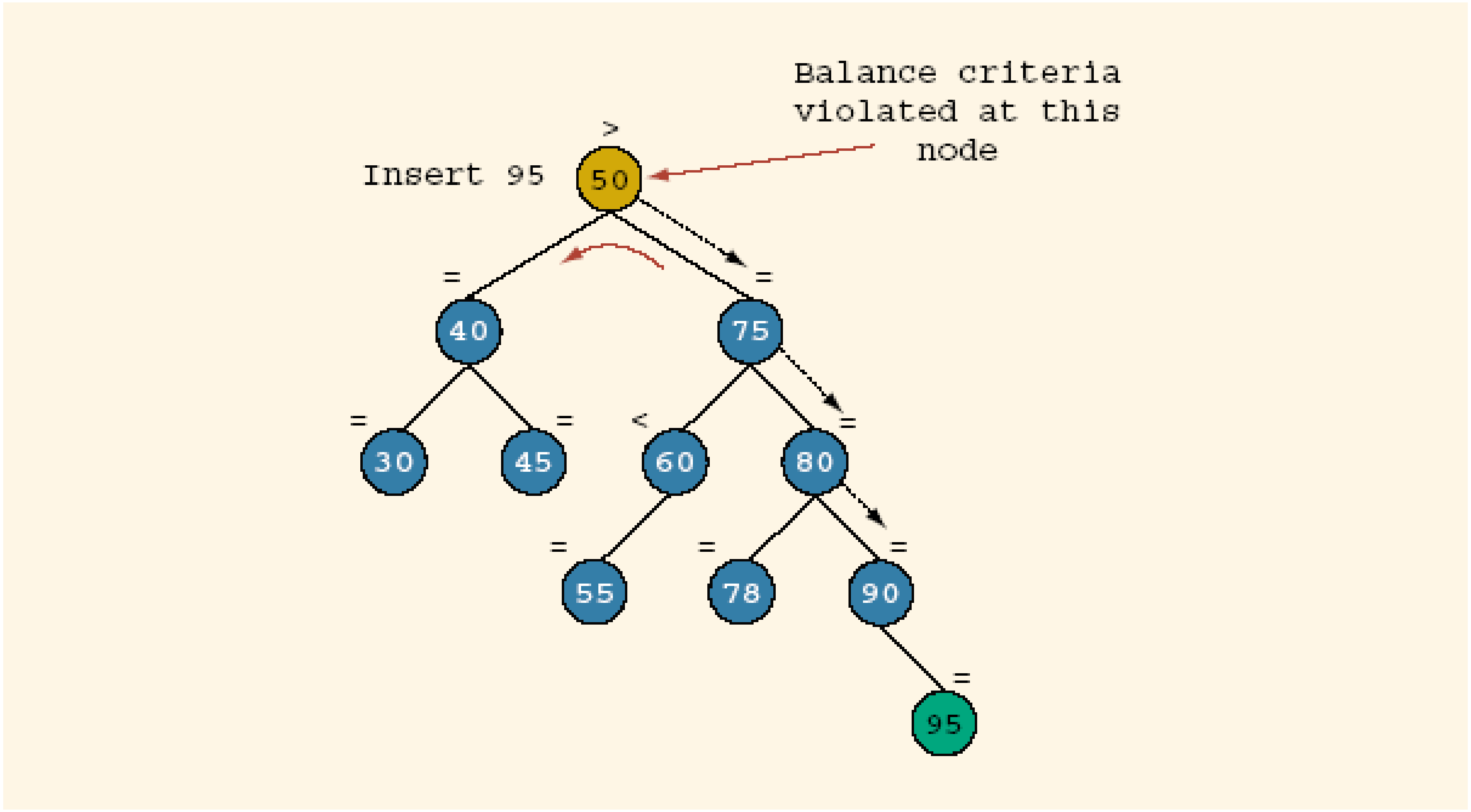


FIGURE AVL-11 Binary search tree of Figure AVL-10 after inserting 95; nodes other than 95 show their balance factors before insertion

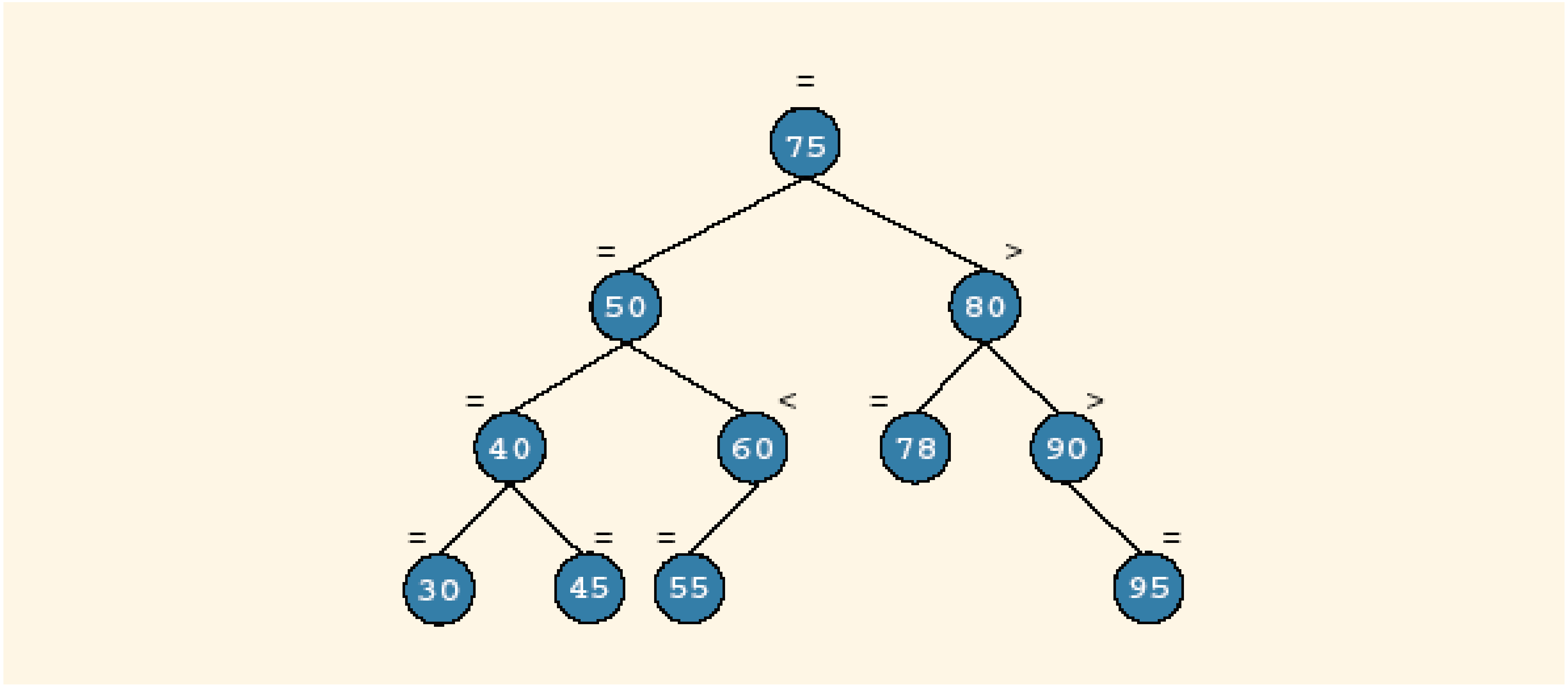


FIGURE AVL-12 AVL tree of Figure AVL-10 after inserting 95 and adjusting the balance factors

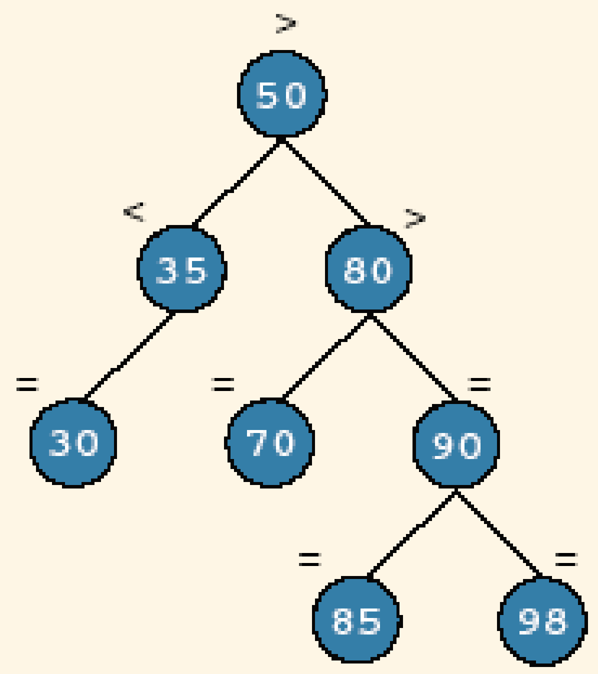


FIGURE AVL-13 AVL tree before inserting 88

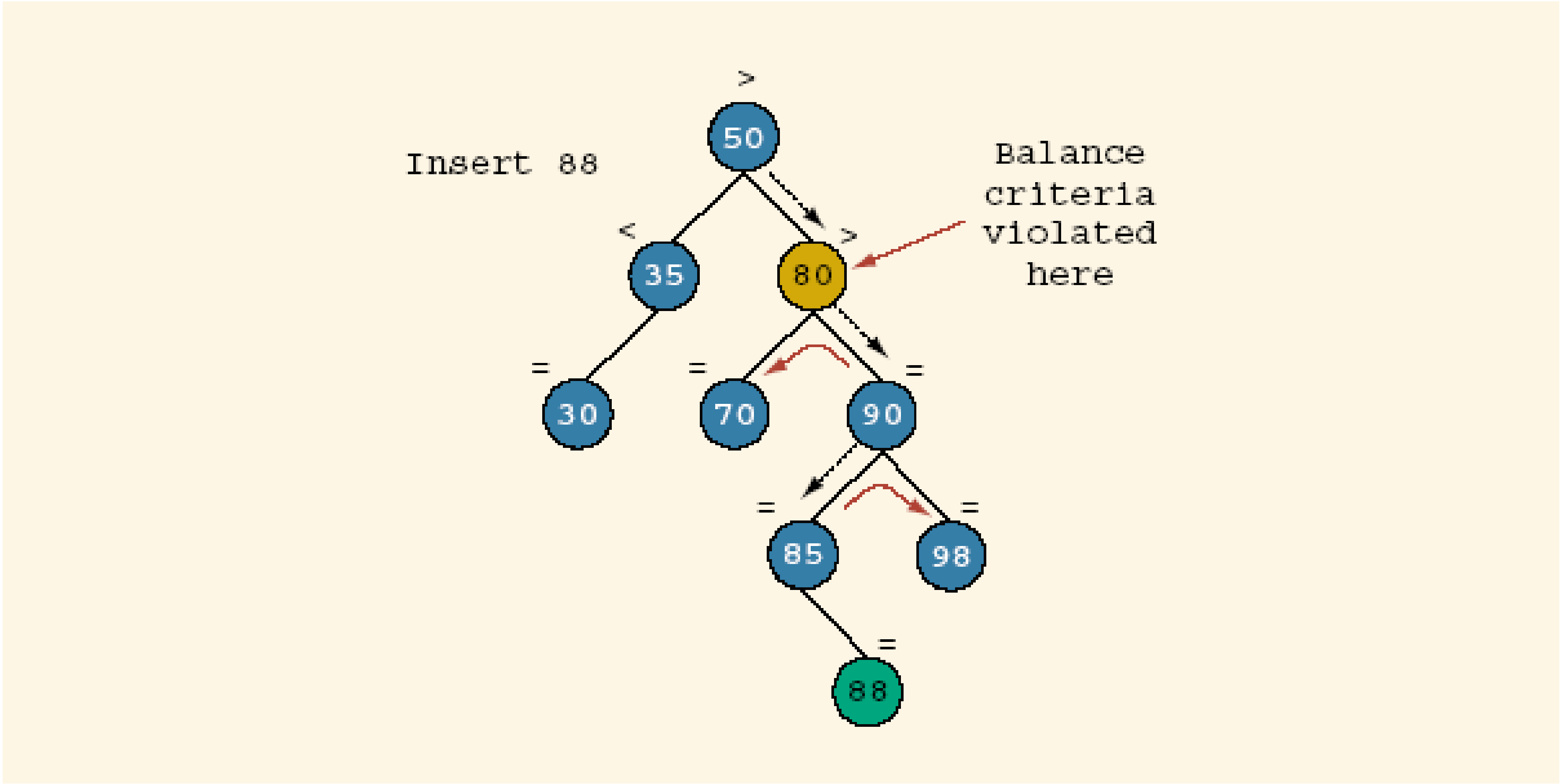


FIGURE AVL-14 Binary search tree of Figure AVL-13 after inserting 88; nodes other than 88 show their balance factors before insertion

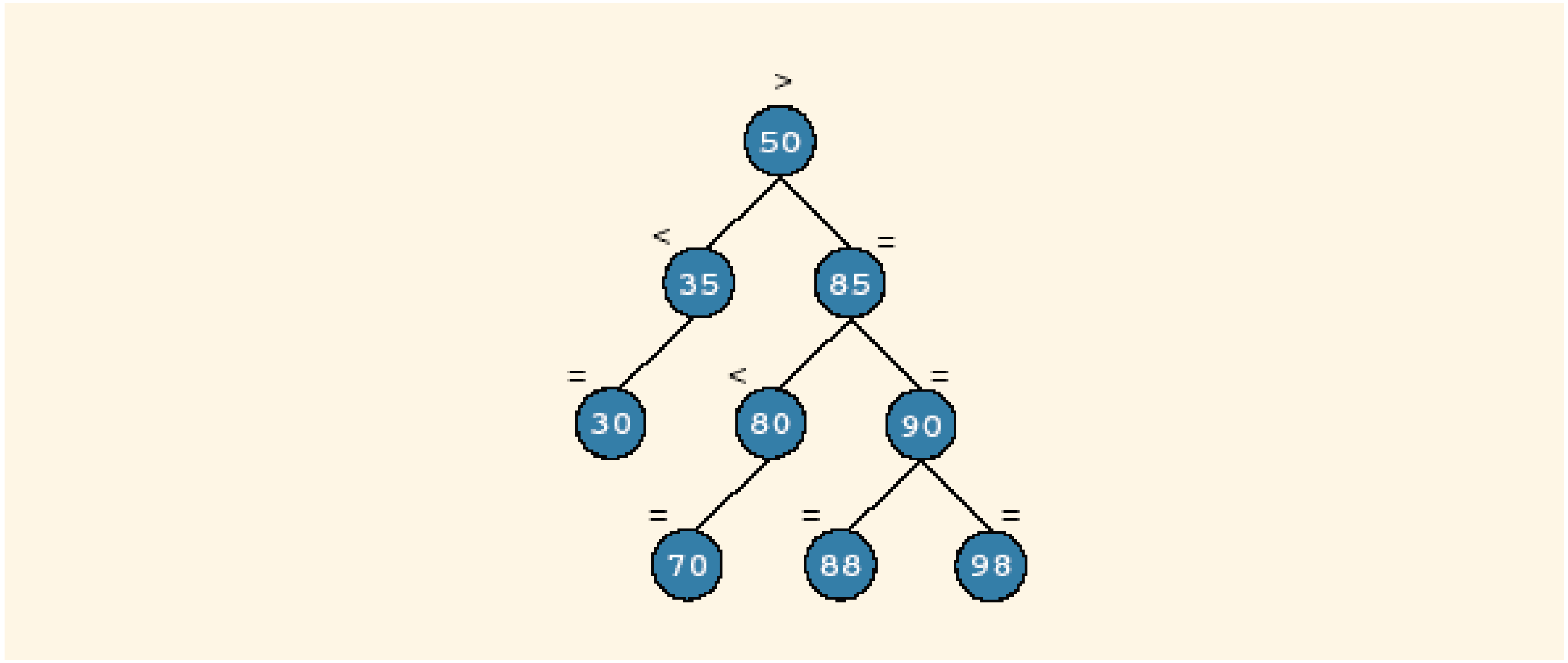


FIGURE AVL-15 AVL tree of Figure AVL-13 after inserting 88 and adjusting the balance factors

AVL Tree Rotations

- The reconstruction procedure is called **rotating** the tree.
- There are two types of rotations, **left rotation** and **right rotation**.
- Suppose that the rotation occurs at node *x*.
- If it is a left rotation, then certain nodes from the right subtree of *x* move to its left subtree; the root of the right subtree of *x* becomes the new root of the reconstructed subtree.
- If it is a right rotation at *x*, certain nodes from the left subtree of *x* move to its right subtree; the root of the left subtree of *x* becomes the new root of the reconstructed subtree.

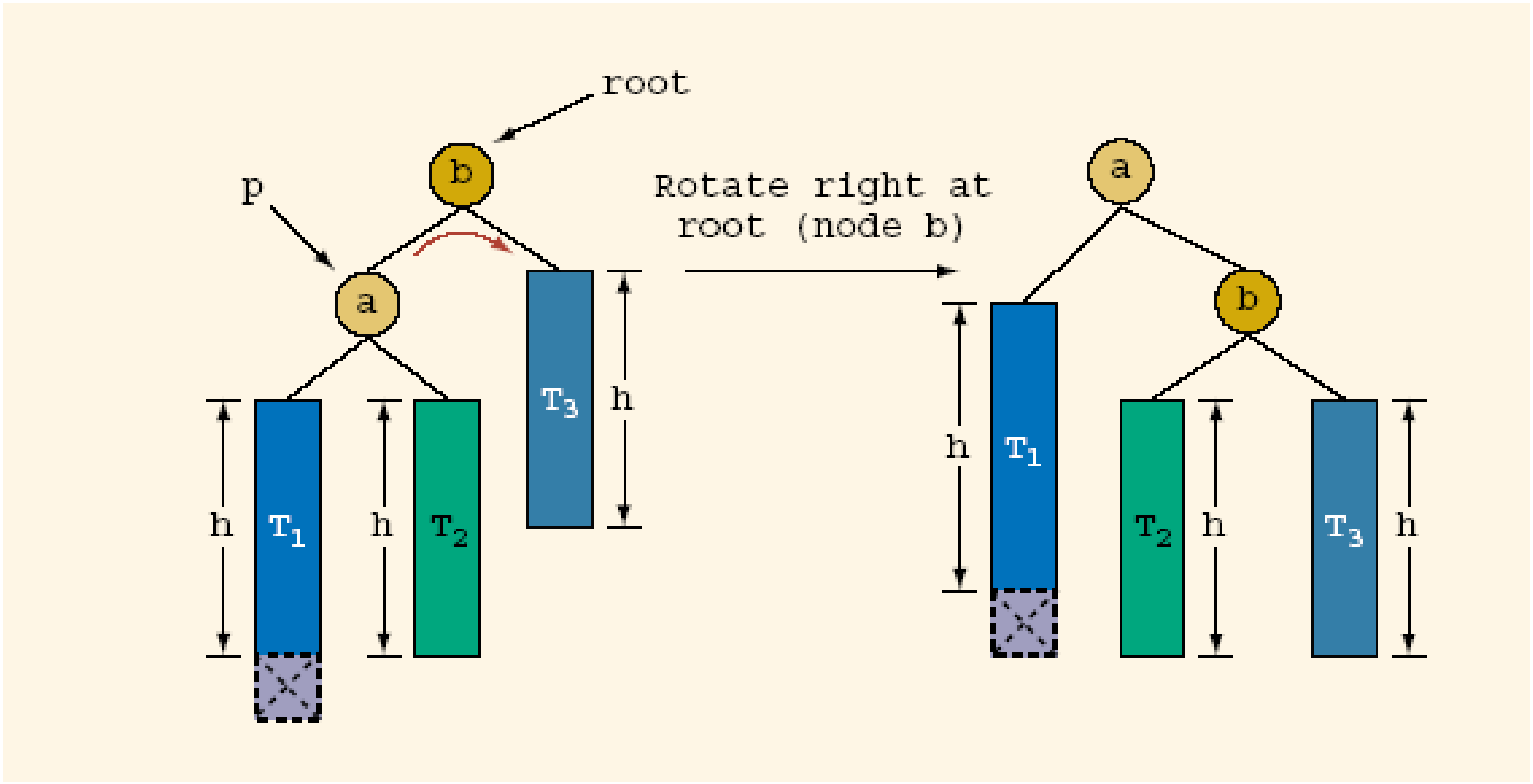


FIGURE AVL-16 Right rotation at *b*

- In Figure AVL-16, subtrees T_1, T_2, and T_3 are of equal height, say *h*. The dotted rectangle shows an item insertion in T_1, causing the height of the subtree T_1 to increase by 1.
- The subtree at node `a` is still an AVL tree, but the balance criteria is violated at the root node. We note the following in this tree.

Because the tree is a binary search tree:

- Every key in T_1 is smaller than the key in node *a*.
- Every key in T_2 is larger than the key in node *a*.
- Every key in T_2 is smaller than the key in node *b*.

Therefore:

1. We make *T*2 (the right subtree of node *a*) the left subtree of node *b*.
2. We make node *b* the right child of node *a*.
3. Node *a* becomes the root node of the reconstructed tree, as shown in Figure AVL-16.

Case 2:

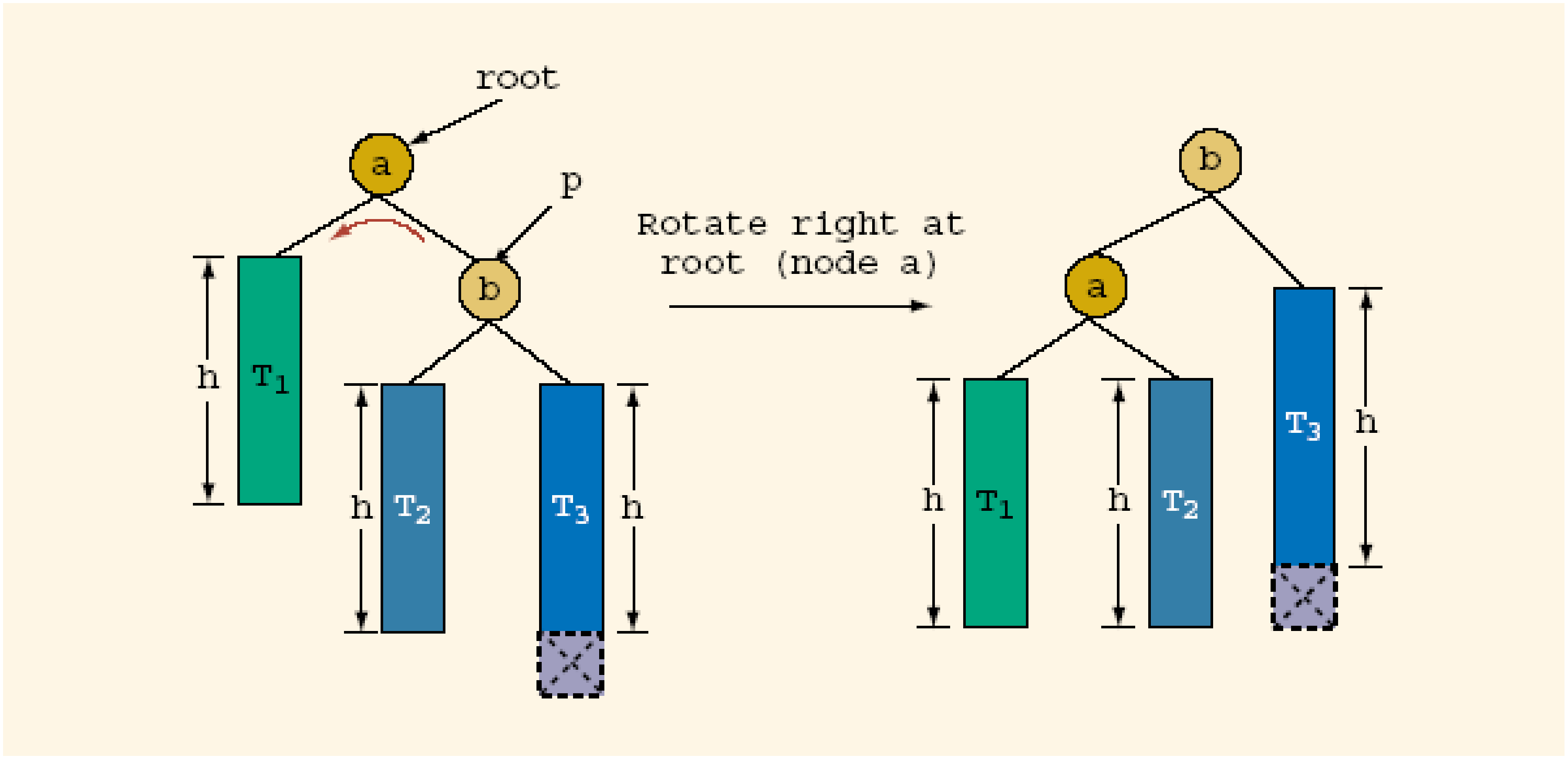


FIGURE AVL-17 Left rotation at *a*

Case 3:

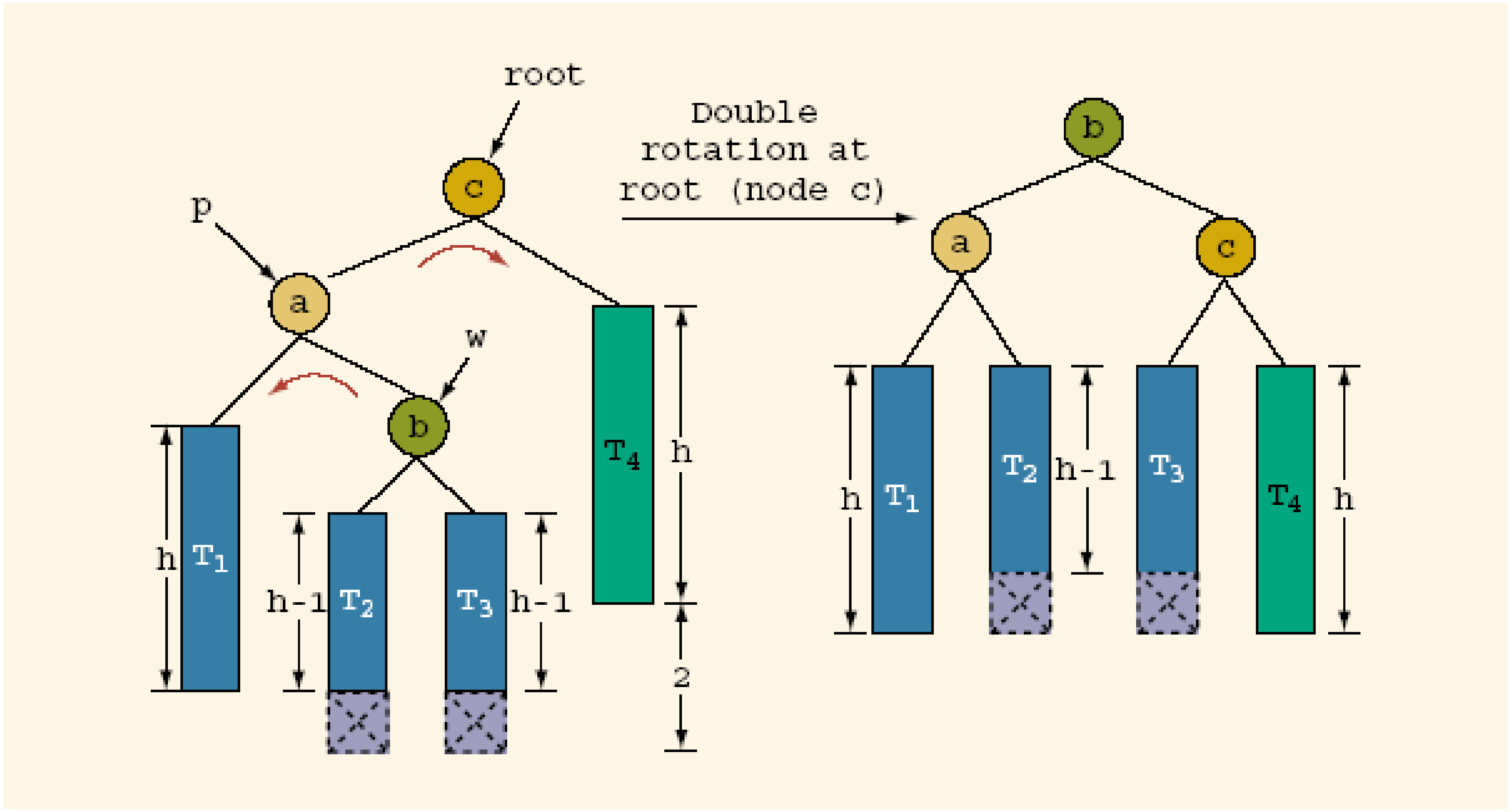


FIGURE AVL-18 Double rotation: first, rotate left at *a*, then rotate right at *c*. New item is inserted in either the subtree T_2 or T_3

- In Figure AVL-18, the dotted rectangle shows that a new item is inserted in the subtree, T_2 or T_3, causing the subtree to grow in height.
- All keys in T_3 are smaller than the key in node *c*.
- All keys in T_3 are larger than the key in node *b*.
- All keys in T_2 are smaller than the key in node *b*.
- All keys in T_2 are larger than the key in node *a*.
- After insertion, the subtrees with root nodes *a* and *b* are still AVL trees.
- The balance criteria is violated at the root node, *c*, of the tree.
- The balance factors of node *c*, $bf(c) = -1$, and node *a*, $bf(a) = 1$, are opposite.
- This is an example of double rotation.
- One rotation is required at node *a* and another rotation is required at node *c*.

- If the balance factor of the node where the tree is to be reconstructed and the balance factor of the higher subtree are opposite, that node requires a double rotation.
- First we rotate the tree at node *a* and then at node *c*.
- Now the tree at node *a* is right high, so we make a left rotation at *a*. Next, because the tree at node *c* is left high, we make a right rotation at *c*.
- Figure AVL-18 shows the resulting tree (which is to the right of the tree after insertion). Figure AVL-19, however, shows both rotations in sequence.

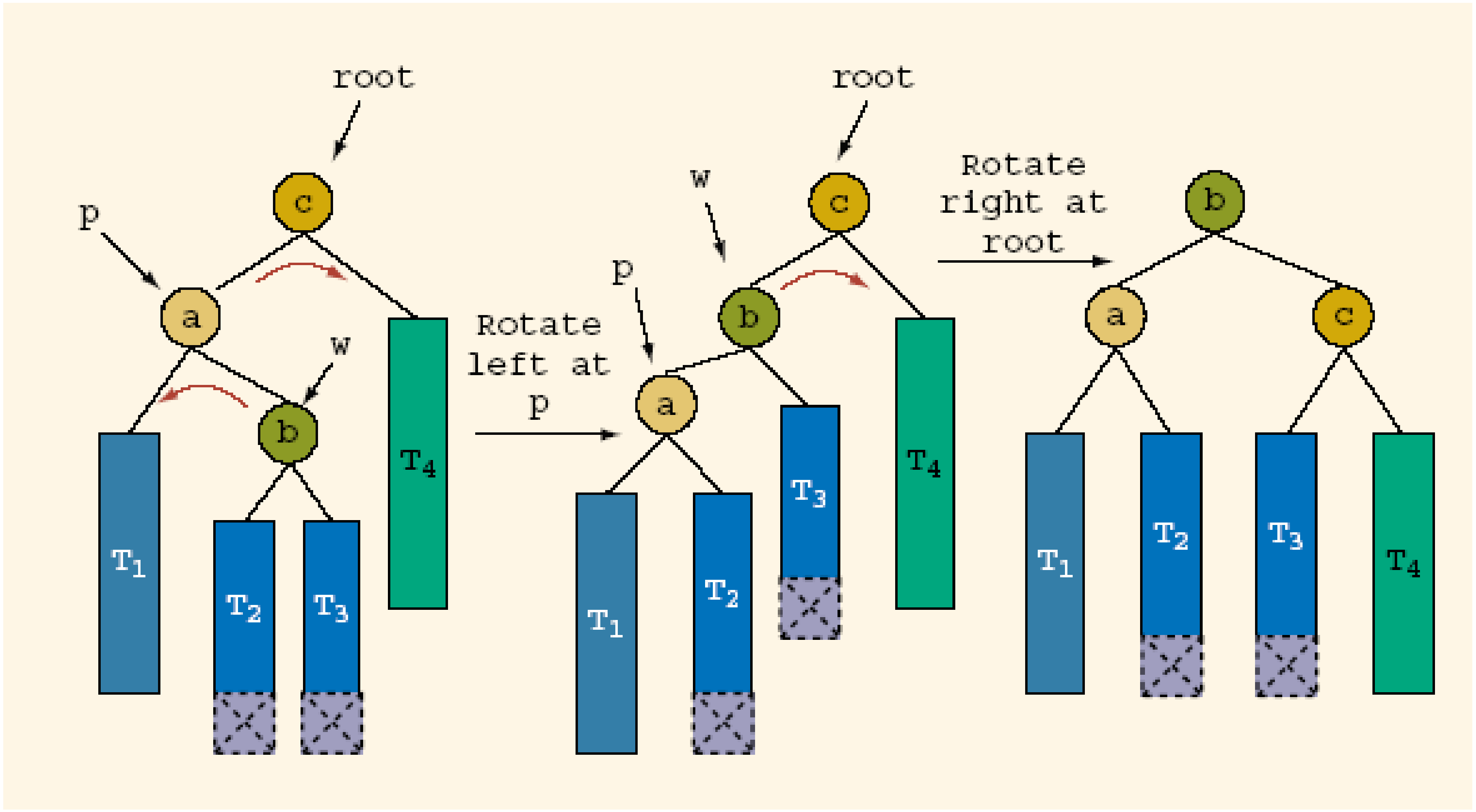


FIGURE AVL-19 Left rotation at *a* followed by a right rotation at *c*

Case 4:

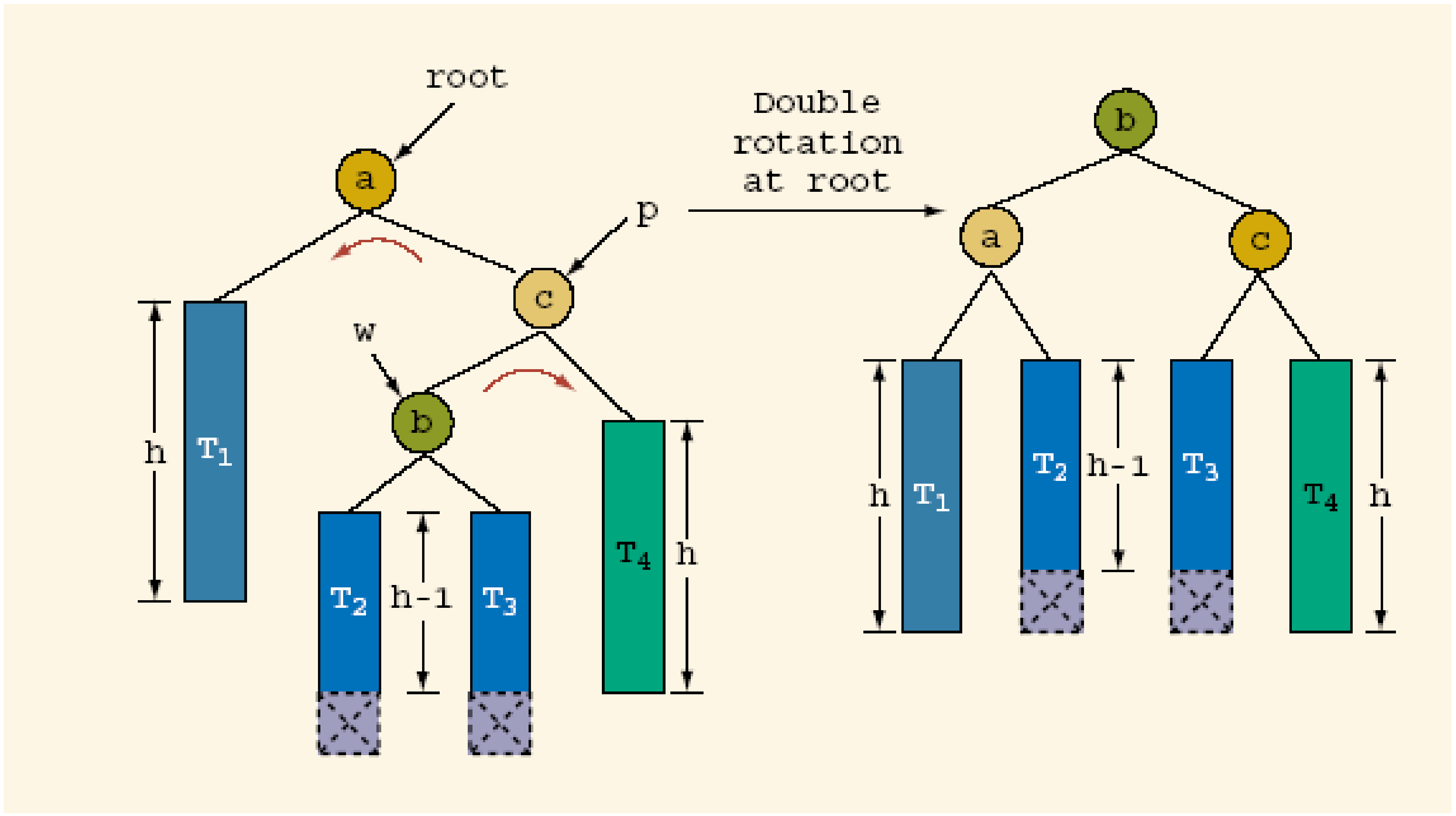


FIGURE AVL-20 Double rotation: first rotate right at *c*, then rotate left at *a*

- Suppose that the tree is to be reconstructed, by rotation, at node *x*. Then the subtree with the root node *x* requires either a single or a double rotation.
  1. Suppose that the balance factor of the node *x* and the balance factor of the root node of the higher subtree of *x* have the same sign, that is, both positive or both negative.
     a. If these balance factors are positive, make a single *left* rotation at *x*.
     b. If these balance factors are negative, make a single *right* rotation at *x*.
  2. Suppose that the balance factor of the node *x* and the balance factor of the higher subtree of *x* are opposite in sign. To be specific, suppose that the balance factor of the node *x* prior to insertion was −1 and suppose that *y* is the root node of the left subtree of *x*. After insertion, the balance factor of the node *y* is 1. That is, after insertion, the right subtree of node *y* grew in height. In this case, we require a *double* rotation at *x*. First we make a left rotation at *y* (because *y* is right high). Then we make a right rotation at *x*.

```
template <class elemType>
void rotateToLeft(AVLNode<elemType>* &root)
{
    AVLNode<elemType> *p;    //pointer to the root of the
                             //right subtree of root
    if (root == NULL)
        cout << "Error in the tree" << endl;
    else if(root->rLink == NULL)
        cout << "Error in the tree:"
             << " No right subtree to rotate." << endl;
    else
    {
        p = root->rLink;
        root->rLink = p->lLink; //the left subtree of p
                               //becomes the right subtree of root
        p->lLink = root;
        root = p; //make p the new root node
    }
}//rotateLeft
```

```
template <class elemType>
void rotateToRight(AVLNode<elemType>* &root)
{
    AVLNode<elemType> *p;  //pointer to the root of
                           //the left subtree of root

    if (root == NULL)
        cout << "Error in the tree" << endl;
    else if(root->lLink == NULL)
        cout << "Error in the tree:"
             << " No left subtree to rotate." << endl;
    else
    {
        p = root->lLink;
        root->lLink = p->rLink; //the right subtree of p
                               //becomes the left subtree of root
        p->rLink = root;
        root = p; //make p the new root node
    }
}//end rotateRight
```

```
template <class elemType>
void balanceFromLeft(AVLNode<elemType>* &root)
{
    AVLNode<elemType> *p;
    AVLNode<elemType> *w;

    p = root->lLink;   //p points to the left subtree of root

    switch (p->bfactor)
    {
    case -1:
        root->bfactor = 0;
        p->bfactor = 0;
        rotateToRight(root);
        break;
    case 0:
        cout << "Error: Cannot balance from the left." << endl;
        break;
    case 1:
        w = p->rLink;
        switch (w->bfactor)  //adjust the balance factors
        {
        case -1:
            root->bfactor = 1;
            p->bfactor = 0;
            break;
        case 0:
            root->bfactor = 0;
            p->bfactor = 0;
            break;
        case 1:
            root->bfactor = 0;
            p->bfactor = -1;
        }//end switch

        w->bfactor = 0;
        rotateToLeft(p);
        root->lLink = p;
        rotateToRight(root);
    }//end switch;
}//end balanceFromLeft
```

```
template <class elemType>
void balanceFromRight(AVLNode<elemType>* &root)
{
    AVLNode<elemType> *p;
    AVLNode<elemType> *w;

    p = root->rLink;   //p points to the left subtree of root

    switch (p->bfactor)
    {
    case -1:
        w = p->lLink;
        switch (w->bfactor) //adjust the balance factors
        {
        case -1:
            root->bfactor = 0;
            p->bfactor = 1;
            break;
        case 0:
            root->bfactor = 0;
            p->bfactor = 0;
            break;
            case 1:
                root->bfactor = -1;
                p->bfactor = 0;
            }//end switch

            w->bfactor = 0;
            rotateToRight(p);
            root->rLink = p;
            rotateToLeft(root);
            break;
        case 0:
            cout << "Error: Cannot balance from the left." << endl;
            break;
        case 1:
            root->bfactor = 0;
            p->bfactor = 0;
            rotateToLeft(root);
        }//end switch;
    }//end balanceFromRight
```

- The following steps describe the function `insertIntoAVL`:
  1. Create a node and assign the item to be inserted to the info field of this node.
  2. Search the tree and find the place for the new node in the tree.
  3. Insert the new node in the tree.
  4. Backtrack the path, which was constructed to find the place for the new node in the tree, to the root node. If necessary, adjust the balance factors of the nodes, or reconstruct the tree at a node on the path.
- The function `insertIntoAVL` also uses a `bool` member variable, `isTaller`, to indicate to the parent whether the subtree grew in height or not.

```
template <class elemType>
void insertIntoAVL(AVLNode<elemType>* &root,
                   AVLNode<elemType> *newNode,
                   bool& isTaller)
{
    if (root == NULL)
    {
        root = newNode;
        isTaller = true;
    }
    else if(root->info == newNode->info)
        cout << "No duplicates are allowed." << endl;
    else if(root->info > newNode->info) //newItem goes in
                                        //the left subtree
    {
        insertIntoAVL(root->lLink, newNode, isTaller);

        if (isTaller)   //after insertion, the subtree grew
                        //in height
            switch (root->bfactor)
            {
            case -1:
                balanceFromLeft(root);
                isTaller = false;
                break;
            case 0:
                root->bfactor = -1;
                isTaller = true;
                break;
            case 1:
                root->bfactor = 0;
                isTaller = false;
            }//end switch
    }//end if
    ...
    else
    {
        insertIntoAVL(root->rLink, newNode, isTaller);

        if (isTaller)        //after insertion, the
                             //subtree grew in height
        switch (root->bfactor)
        {
        case -1:
            root->bfactor = 0;
            isTaller = false;
            break;
        case 0:
            root->bfactor = 1;
            isTaller = true;
            break;
        case 1:
            balanceFromRight(root);
            isTaller = false;
        }//end switch
    }//end else
}//insertIntoAVL
```

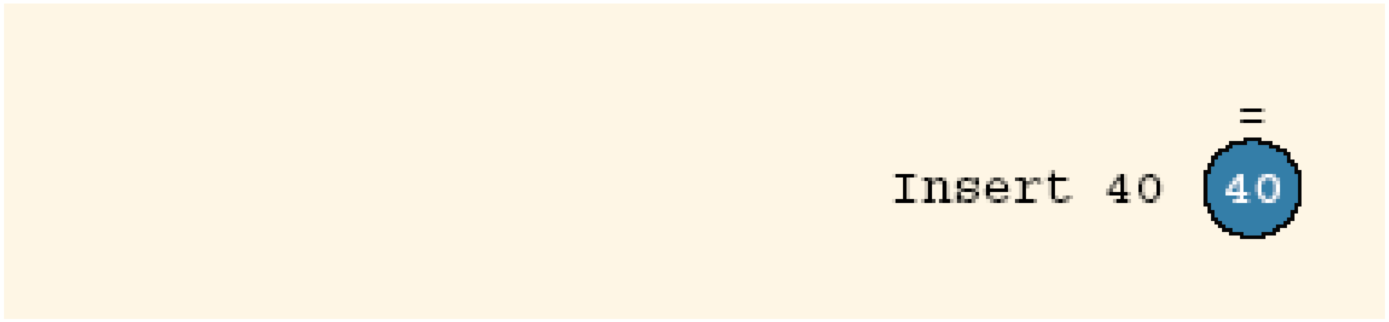


FIGURE AVL-21 AVL tree after inserting 40

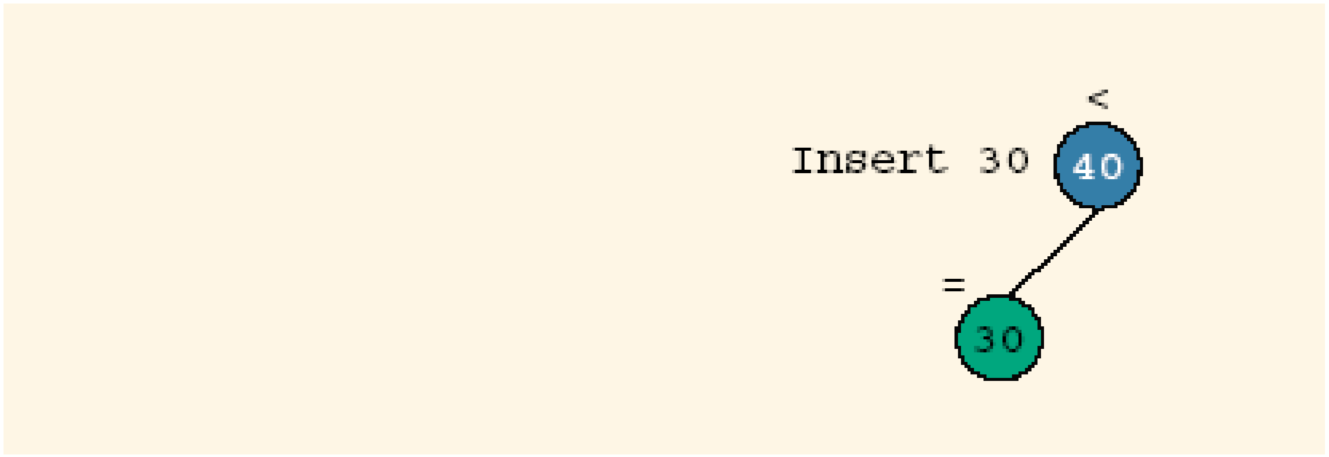


FIGURE AVL-22 AVL tree after inserting 30

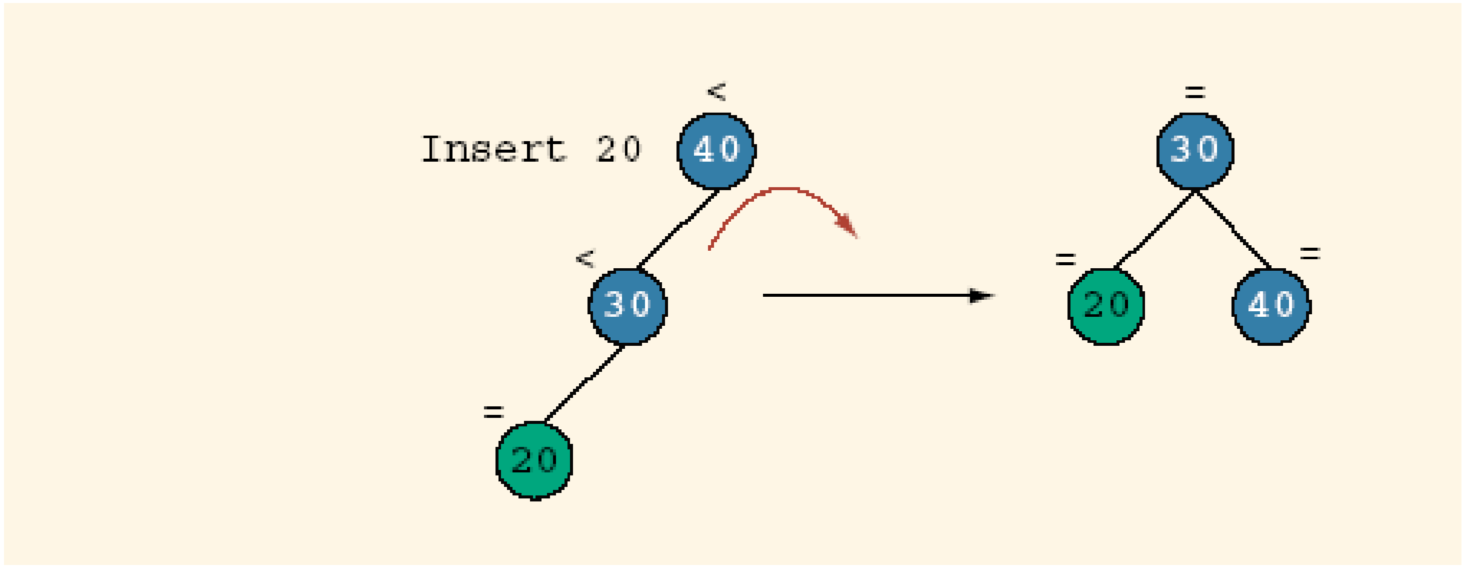


FIGURE AVL-23 AVL tree after inserting 20

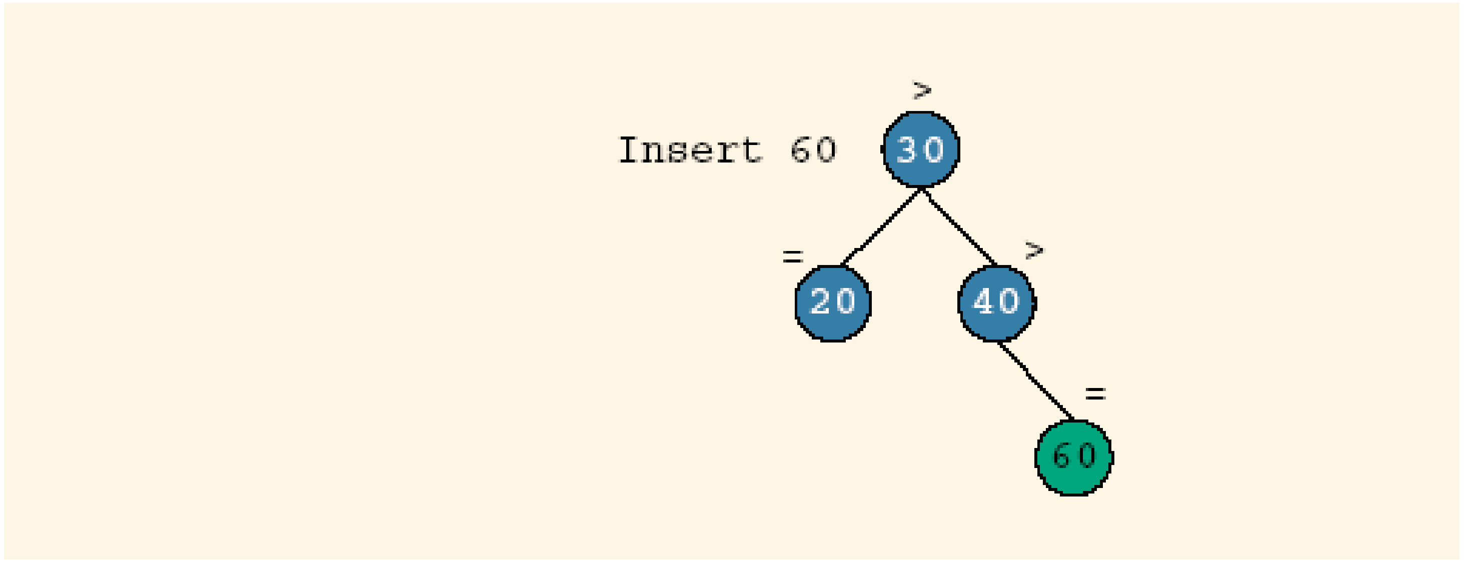


FIGURE AVL-24 AVL tree after inserting 60

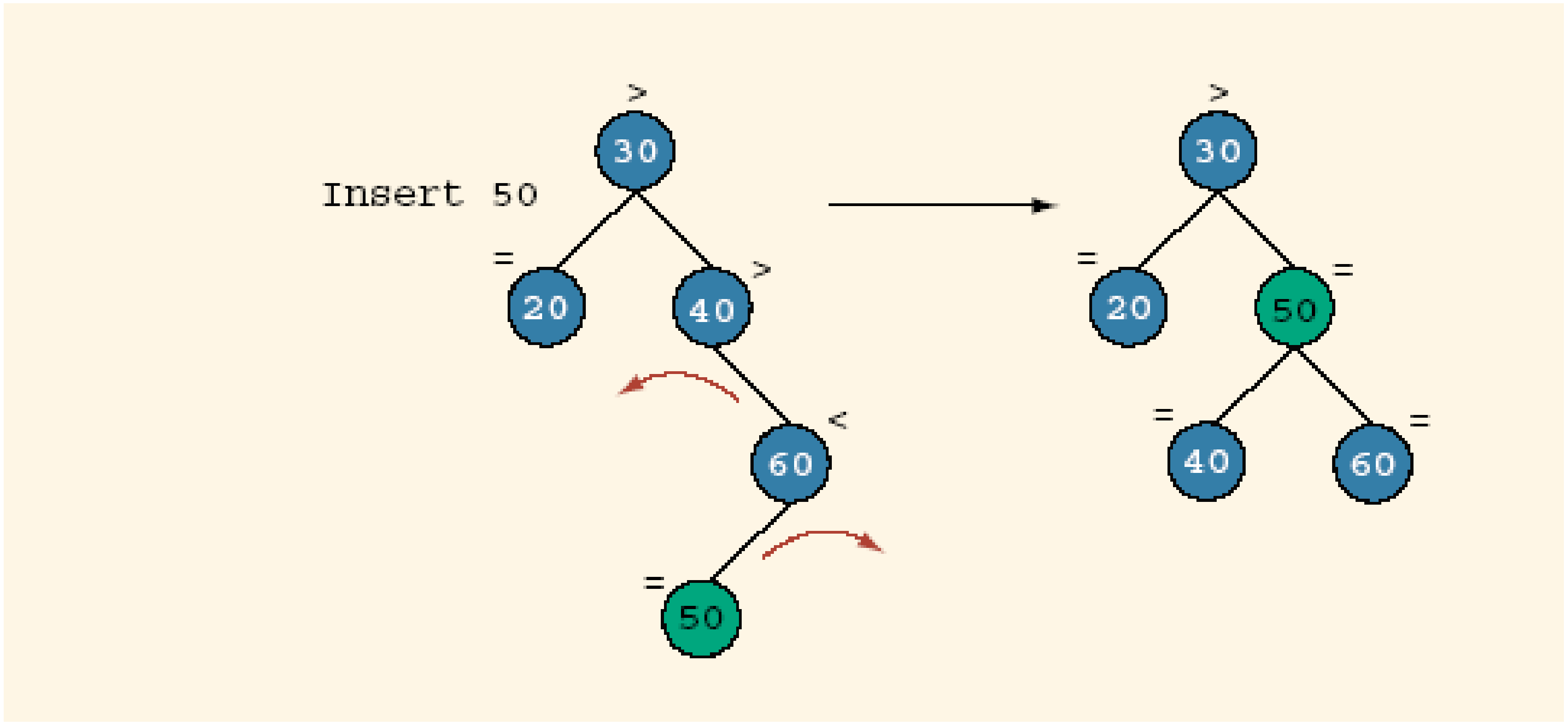


FIGURE AVL-25 AVL tree after inserting 50

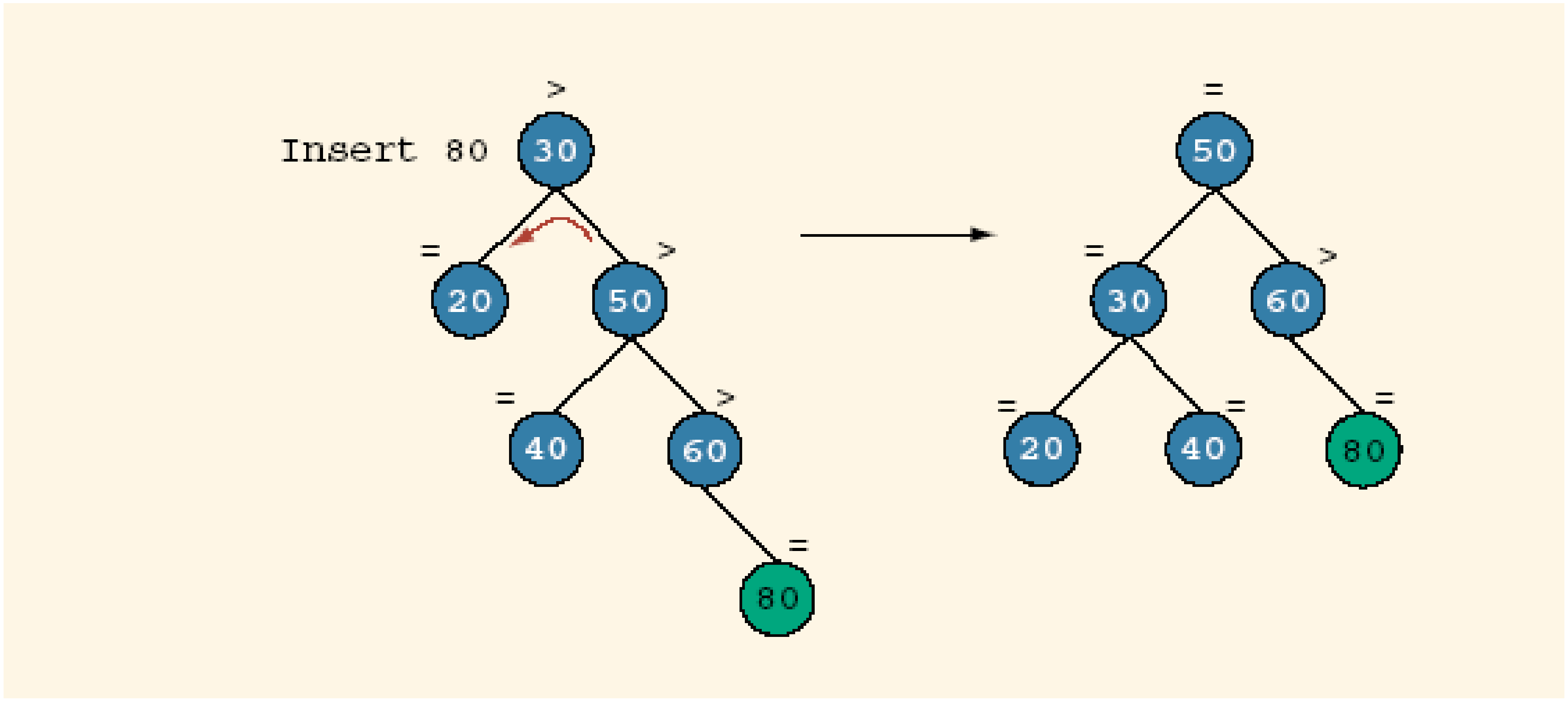


FIGURE AVL-26 AVL tree after inserting 80

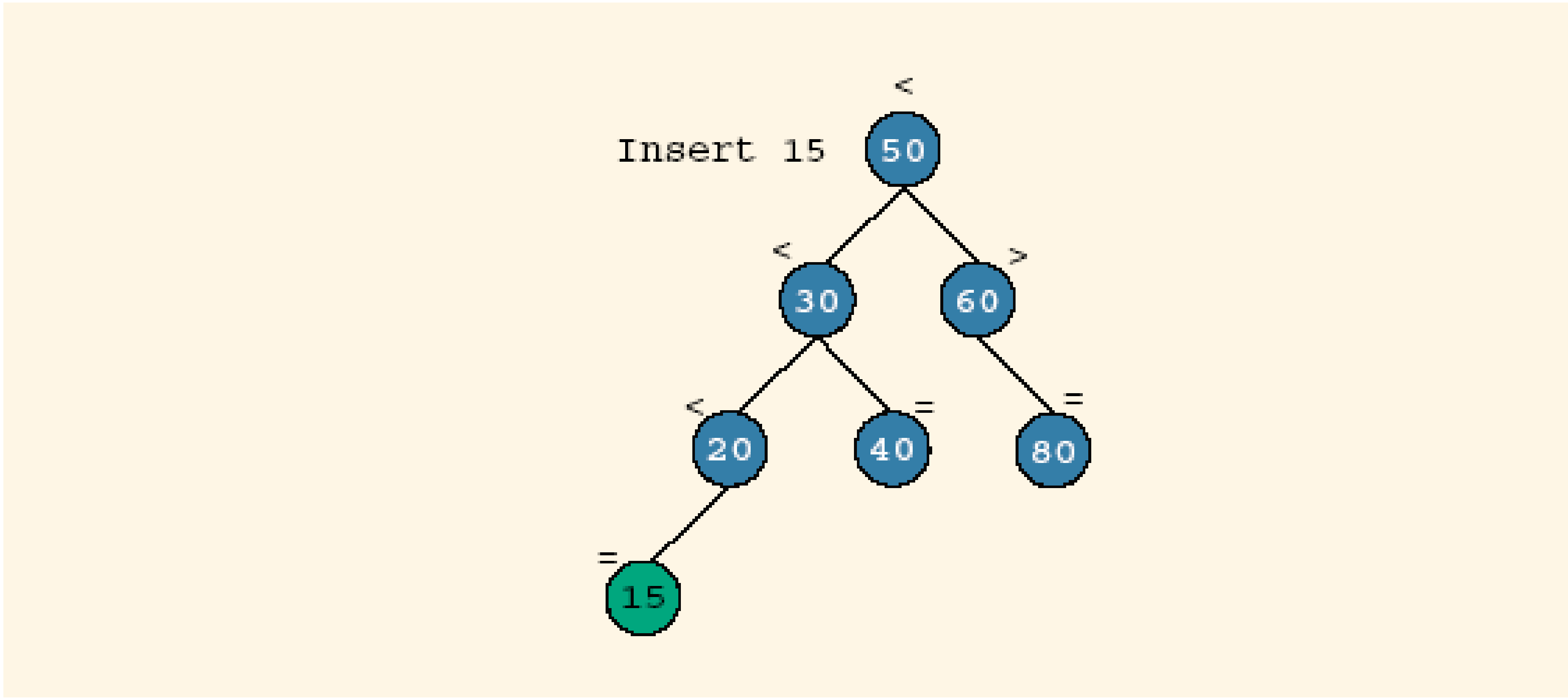


FIGURE AVL-27 AVL tree after inserting 15

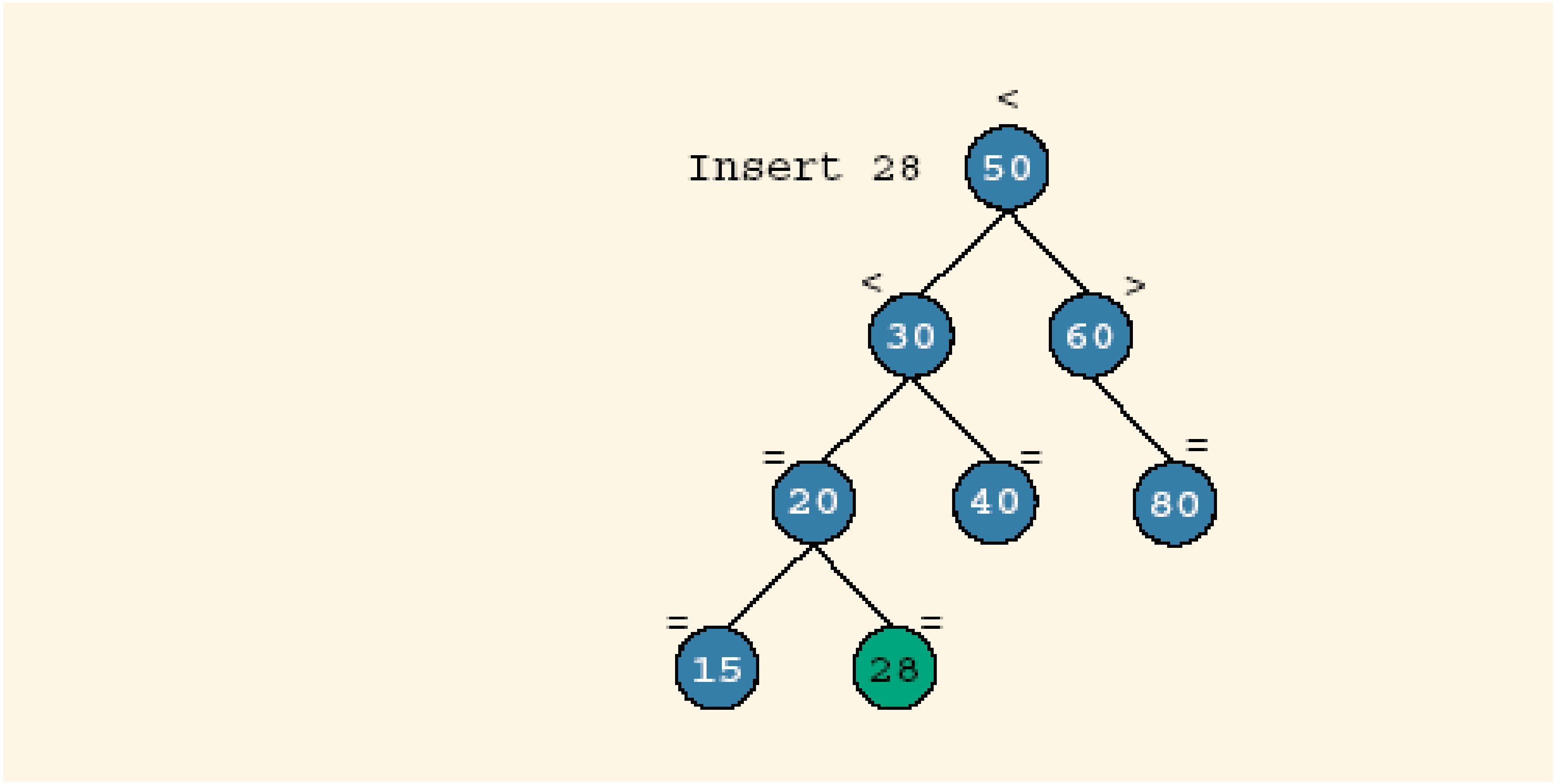


FIGURE AVL-28 AVL tree after inserting 28

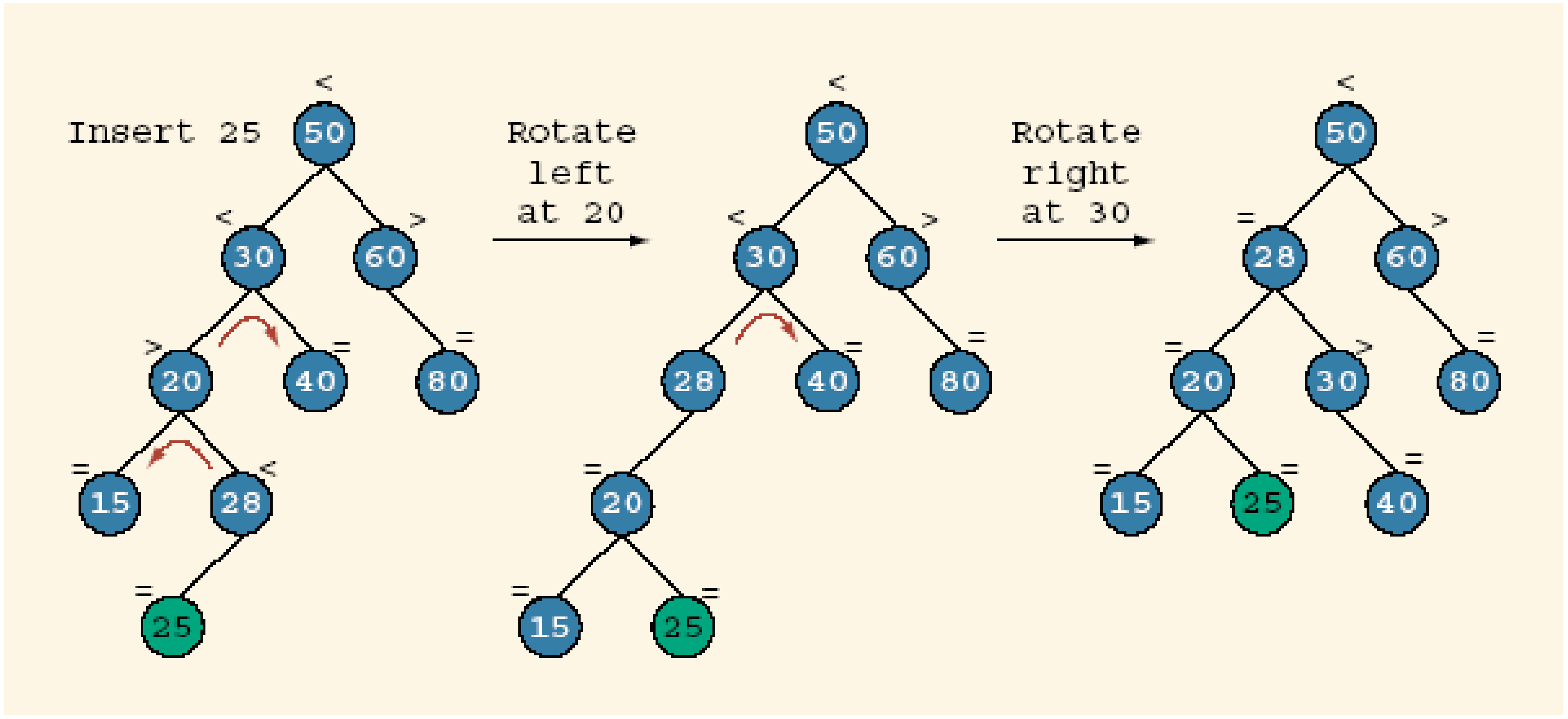


FIGURE AVL-29 AVL tree after inserting 25

```
template <class elemType>
void insert(const elemType &newItem)
{
    bool isTaller = false;
    AVLNode<elemType> *newNode;

    newNode = new AVLNode<elemType>;
    newNode->info = newItem;
    newNode->bfactor = 0;
    newNode->lLink = NULL;
    newNode->rLink = NULL;

    insertIntoAVL(root, newNode, isTaller);
}
```

Deletion from AVL Trees

To delete an item from an AVL tree, first we find the node containing the item to be deleted. The following four cases arise:

Case 1: The node to be deleted is a leaf.

Case 2: The node to be deleted has no right child, that is, its right subtree is empty.

Case 3: The node to be deleted has no left child, that is, its left subtree is empty.

Case 4: The node to be deleted has a left child and a right child.

Analysis: AVL Trees

Consider all the possible AVL trees of height *h*. Let T_h be an AVL tree of height *h* such that T_h has the fewest number of nodes. Let T_{hl} denote the left subtree of T_h and T_{hr} denote the right subtree of T_h. Then

$$|T_h| = |T_{hl}| + |T_{hr}| + 1$$

where $|T_h|$ denotes the number of nodes in T_h.

Analysis: AVL Trees

Because T_h is an AVL tree of height h such that T_h has the fewest number of nodes, it follows that one of the subtrees of T_h is of height $h - 1$ and the other is of height $h - 2$. To be specific, suppose that T_{hl} is of height $h - 1$ and T_{hr} is of height $h - 2$. From the definition of T_h, it follows that T_{hl} is an AVL tree of height $h - 1$ such that T_{hl} has the fewest number of nodes among all AVL trees of height $h - 1$. Similarly, T_{hr} is an AVL tree of height $h - 2$ that has the fewest number of nodes among all AVL trees of height $h - 2$. Thus, T_{hl} is of the form T_{h-1} and T_{hr} is of the form T_{h-2}. Hence,

$$|T_h| = |T_{h-1}| + |T_{h-2}| + 1$$

Clearly:

$$\begin{aligned} |T_0| &= 1 \\ |T_1| &= 2 \end{aligned}$$

Let $F_{h+2} = |T_h| + 1$. Then:

$$\begin{aligned} F_{h+2} &= F_{h+1} + F_h \\ F_2 &= 2 \\ F_3 &= 3. \end{aligned}$$

This is called a **Fibonacci sequence**. The solution to F_h is given by:

$$F_h \approx \frac{\phi^h}{\sqrt{5}}, \quad \text{where} \quad \phi = \frac{1+\sqrt{5}}{2}.$$

Hence:

$$|T_h| \approx \frac{\phi^{h+2}}{\sqrt{5}} = \frac{1}{\sqrt{5}} \left[\frac{1+\sqrt{5}}{2}\right]^{h+2}$$

From this it can be concluded that:

$$h \approx (1.44) \log_2 |T_h|$$