Binary Search Trees

• Data in each node
  – Larger than the data in its left child
  – Smaller than the data in its right child

**FIGURE 11-6** Arbitrary binary tree

**FIGURE 11-7** Binary search tree
Binary Search Trees (cont’d.)

- **class bSearchTreeType**
  - Illustrates basic operations to implement a binary search tree
  - See code on page 618
- **Function** search
- **Function** insert
- **Function** delete
Binary Search Tree: Analysis

- Worst case
  - $T$: linear
  - Successful case
    - Algorithm makes $(n + 1) / 2$ key comparisons (average)
    - Unsuccessful case: makes $n$ comparisons

*FIGURE 11-10* Linear binary trees
Binary Search Tree: Analysis (cont’d.)

• Average-case behavior
  – Successful case
    • Search would end at a node
    • \( n \) items exist, providing \( n! \) possible orderings of the keys
  – Number of comparisons required to determine whether \( x \) is in \( T \)
    • One more than the number of comparisons required to insert \( x \) in \( T \)
  – Number of comparisons required to insert \( x \) in \( T \)
    • Same as number of comparisons made in the unsuccessful search reflecting that \( x \) is not in \( T \)
Binary Search Tree: Analysis (cont’d.)

\[ S(n) = 1 + \frac{U(0) + U(1) + \ldots + U(n-1)}{n} \]  
\[ \text{Equation 11-1} \]

It is also known that

\[ S(n) = \left(1 + \frac{1}{n}\right) U(n) - 3 \]  
\[ \text{Equation 11-2} \]

Solving Equations (11-1) and (11-2), it can be shown that \( U(n) \approx 2.77\log_2 n \) and \( S(n) \approx 1.39\log_2 n \).
Binary Search Tree: Analysis (cont’d.)

• Theorem: let $T$ be a binary search tree with $n$ nodes, where $n > 0$
  – The average number of nodes visited in a search of $T$ is approximately $1.39 \log_2 n = O(\log_2 n)$
  – The number of key comparisons is approximately $2.77 \log_2 n = O(\log_2 n)$
AVL (Height-Balanced) Trees

• AVL tree (height-balanced tree)
  - Resulting binary search is nearly balanced

• Perfectly balanced binary tree
  - Heights of left and right subtrees of the root: equal
  - Left and right subtrees of the root are perfectly balanced binary trees

FIGURE 11-12 Perfectly balanced binary tree
AVL (Height-Balanced) Trees (cont’d.)

• An AVL tree (or height-balanced tree) is a binary search tree such that
  – The heights of the left and right subtrees of the root differ by at most one
  – The left and right subtrees of the root are AVL trees

FIGURE 11-13 AVL and non-AVL trees
Proposition: Let $T$ be an AVL tree and $x$ be a node in $T$. Then $|x_h - x_l| \leq 1$, where $|x_h - x_l|$ denotes the absolute value of $x_h - x_l$.

Let $x$ be a node in the AVL tree $T$.

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

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

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

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

Definition: Let $x$ be a node in a binary tree. We say that the node $x$ **violates the balance criteria** if $|x_h - x_l| > 1$, that is, the heights of the left and right subtrees of $x$ differ by more than 1.
Binary Search Trees (cont’d.)

• A binary search tree, $T$, is either empty or the following is true:
  – $T$ has a special node called the root node
  – $T$ has two sets of nodes, $L_T$ and $R_T$, called the left subtree and right subtree of $T$, respectively
  – The key in the root node is larger than every key in the left subtree and smaller than every key in the right subtree
  – $L_T$ and $R_T$ are binary search trees
AVL (Height-Balanced) Trees (cont’d.)

- Definition of a node in the AVL tree

```cpp
template<class elemType>
struct AVLNNode
{
    elemType info;
    int bfactor;  //balance factor
    AVLNode<elemType> *llink;
    AVLNode<elemType> *rlink;
};
```
AVL (Height-Balanced) Trees (cont’d.)

- AVL binary search tree search algorithm
  - Same as for a binary search tree
  - Other operations on AVL trees
    - Implemented exactly the same way as binary trees
    - Item insertion and deletion operations on AVL trees
      - Somewhat different from binary search trees operations
Insertion

• First search the tree and find the place where the new item is to be inserted
  – Can search using algorithm similar to search algorithm designed for binary search trees
  – If the item is already in tree
    • Search ends at a nonempty subtree
    • Duplicates are not allowed
  – If item is not in AVL tree
    • Search ends at an empty subtree; insert the item there
• After inserting new item in the tree
  – Resulting tree might not be an AVL tree
Insertion (cont’d.)

**FIGURE 11-14** AVL tree before and after inserting 90

**FIGURE 11-15** AVL tree before and after inserting 75
Insertion (cont’d.)

FIGURE 11-16 AVL tree before and after inserting 95

FIGURE 11-17 AVL tree before and after inserting 88
AVL Tree Rotations

• Rotating tree: reconstruction procedure
• Left rotation and right rotation
• Suppose that the rotation occurs at node x
  – Left rotation: 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
  – 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 11-18 Right rotation at $b$

FIGURE 11-19 Left rotation at $a$
FIGURE 11-20 Double rotation: First rotate left at $a$ and then right at $c$

FIGURE 11-21 Left rotation at $a$ followed by a right rotation at $c$
AVL Tree Rotations (cont’d.)

FIGURE 11-22 Double rotation: First rotate right at c, then rotate left at a
template <class elemT>
void rotateToLeft(AVLNode<elemT> * &root) {
    AVLNode<elemT> *p;  // pointer to the root of
                        // the right subtree of root
    if (root == NULL)  
        cerr << "Error in the tree" << endl;
    else if (root->rlink == NULL)  
        cerr << "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 elemT>
void rotateToRight(AVLNode<elemT> * &root) {
    AVLNode<elemT> *p;  // pointer to the root of
                        // the left subtree of root
    if (root == NULL)  
        cerr << "Error in the tree" << endl;
    else if (root->llink == NULL)  
        cerr << "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 elemT>
void balanceFromLeft(AVLNode<elemT>* &root)
{
    AVLNode<elemT> *p;
    AVLNode<elemT> *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:
        cerr << "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 elemT>
void balanceFromRight(AVLNode<elemT>* &root)
{
    AVLNode<elemT> *p;
    AVLNode<elemT> *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:
        cerr << "Error: Cannot balance from the left." << endl;
        break;

    case 1:
        root->bfactor = 0;
        p->bfactor = 0;
        rotateToLeft(root);
    } // end switch
} // end balanceFromRight
FIGURE 11-23 Item insertion into an initially empty AVL tree