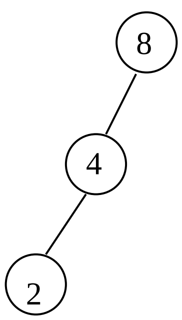
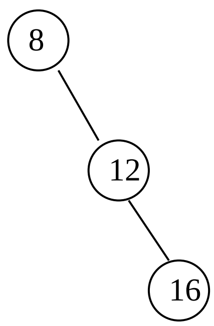
1 AVL Trees Continued

To refresh:

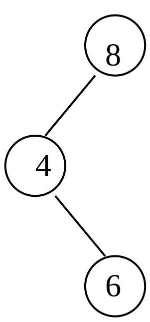
CASE 1



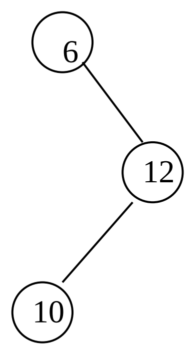
CASE 2

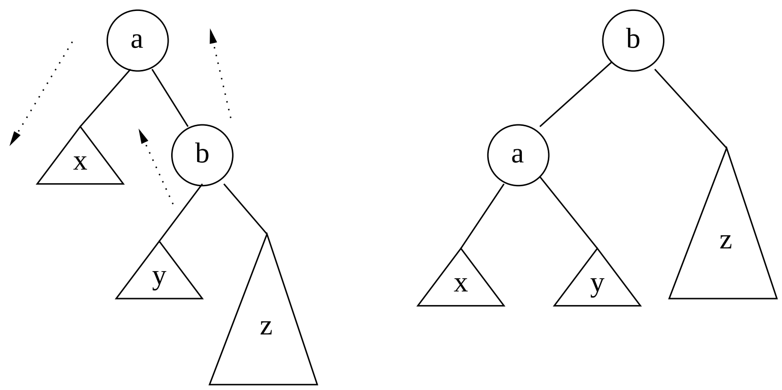


CASE 3

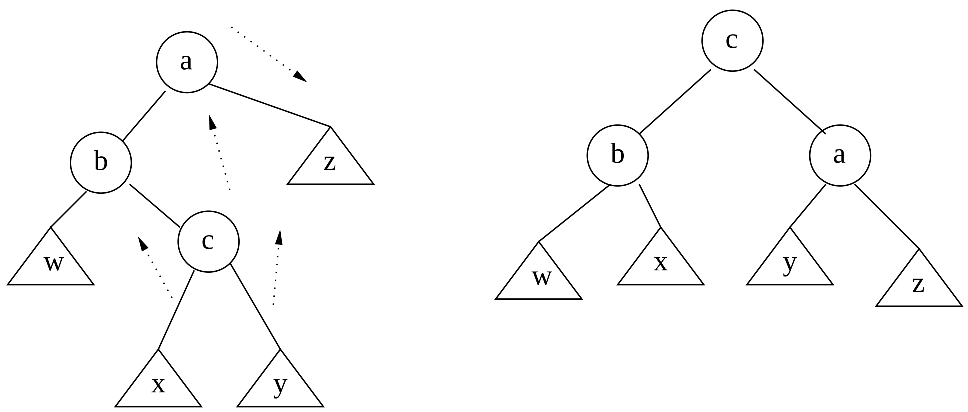


CASE 4





SINGLE ROTATION



DOUBLE ROTATION

File: AvlTree.h

Purpose: Class declaration and member functions for AVL trees. It is built from the code provided in the text. There are three main differences:

- It is not a derived class of Binary_Search_Tree, so the confusing use of pointer casting in the text is avoided.
- It implements the "Remove" operation. (The author had said to just use "lazy deletion".)
- The tree balancing code has been reorganized. It is all in two functions, "Balance_Left" and "Balance_Right". Along with Insert and Remove, these are the functions that should be studied most carefully.
- Finally, note the use of the compiler directive #ifdef DEMO to control whether or not output occurs from the balancing code.

```
//  Template for nodes in the AVL tree.
//  It includes the height of a node as
//  well as friend functions for computing
//  the height and obtaining the height.
//  Additionally, it contains friend functions
//  to do the balancing operations.

template <class Etype>
class Avl_Node {
protected:
  Etype Element;
  Avl_Node<Etype> *Left;
  Avl_Node<Etype> *Right;
  int Height;

  Avl_Node( Etype E = 0,
            Avl_Node<Etype> *L = 0,
            Avl_Node<Etype> *R = 0,
            int H = 0 )
  : Element(E), Left(L), Right(R), Height(H) { }

  friend int Node_Ht( Avl_Node<Etype> *P );
  friend void Calculate_Height( Avl_Node<Etype> *P );
  friend void Balance_Left( Avl_Node<Etype> * & T );
  friend void Balance_Right( Avl_Node<Etype> * & T );

  friend class AvlTree<Etype>;
};
```

- In the AVL tree node we use an incomplete class and friend declaration to grant the AVL tree class access to Avl_Node.
- **Etype** is the Element type (the data type stored in the node).
- ***Left** and ***Right** are pointers to the left and right children of the node.
- **Height** is the height of the tree (how strange?).
- There is one **constructor** which as a default sets the right and left pointers to null, set the data to zero, and sets the height to zero.
- **Node_Ht** returns the height of a given node. (note that the height must be calculated before-hand)
- **Calculate_Height** actually calculates the height.
- **Balance_Left** and **Balance_Right** are the two functions that maintain the balance condition of the AVL tree. These are the functions that we will study the most today (I hope).

```
//  The actual class for AVL trees.
//  Many of these functions are repeated
//  from class Binary_Search_Tree.
//  The main differences are in
//  functions Insert and Remove.

template <class Etype>
class AvlTree
{
protected:
  void Make_Empty ( Avl_Node<Etype> * & T );
  Avl_Node<Etype> * Copy( const Avl_Node<Etype> *T );
  void Insert( const Etype & X, Avl_Node<Etype> * & T );
  void Remove( const Etype & X, Avl_Node<Etype> * & T );
  void Print_Tree( Avl_Node<Etype> *  T, int depth ) const;
  Avl_Node<Etype> *
    Find( const Etype & X, Avl_Node<Etype> * T ) const;
  Avl_Node<Etype> *
    Find_Min( Avl_Node<Etype> * T ) const;

  Avl_Node<Etype> *Root;
  Avl_Node<Etype> *Last_Find;
```

- **Make_Empty:** Given a starting node, this function recursively empties all the nodes "below" the starting node. It uses the delete function to reclaim the memory. This is used by both the destructor and the assignment operator.
- **Copy:** Given a starting node, this function recursively copies all the nodes "below" the starting node. This function is used by the assignment operator.
- **Insert:** Given a starting node, this function inserts a node into the AVL tree. Maintains the binary search tree condition and the AVL balance condition. The function takes data (Element Type), creates a node that stores the data, and inserts the node in the proper place.
- **Remove:** Given a starting node, this function removes a node from the AVL tree. The function takes target data, tries to find the target data in the tree, and if a node is found that matches the target data then the node is removed. Maintains the binary search tree condition and the AVL balance condition.

- **Print_Tree:** Given a starting node and initial depth, this function will print every node "below" the starting node. The depth is used to properly indent the printed results. This function is used by the public Print_Tree function.
- **Find:** Finds a target value "below" an initial starting node. Returns a constant reference to an item that matches the target value. Used by the public Find function.
- **Find_Min:** Finds the minimum value "below" an initial starting node. Used in the Remove function.
- **Root:** The Root of the AVL tree
- **Last_Find:** A pointer to the last item that was found. This member variable is used by the public Find function.

```
public:
  AvlTree( ) :  Root( 0 ), Last_Find( 0 ) { }
  AvlTree( AvlTree<Etype> & Value )
    { Warn ( "Tree passed by value" ); operator=( Value ); }

  // Destructor.
  ~AvlTree( )
    { Make_Empty( Root ); Root = Last_Find = 0; }

  // Assignment operator.
  const AvlTree<Etype> & operator = ( const AvlTree<Etype> & Value );

  // Member functions.
  void Make_Empty( )
    { Make_Empty ( Root ); Root = Last_Find = 0; }
  void Print_Tree( ) const
    { Print_Tree( Root, 0 ); }
  int Find( const Etype & X )
    { return int( Last_Find  = Find( X, Root ) ); }
  void Insert( const Etype & X )
    { Insert( X, Root ); }
  void Remove( const Etype & X )
    { Remove( X, Root ); }
  int Height( )
    { return (Root == 0) ? -1 : Root->Height; }
};
```

- **Constructors:** As you would expect, there is a default constructor that sets Root and Last_Find to null. There is also a copy constructor that spits out a warning and then calls the assignment operator.
- **Destructor:** As you would expect, the destructor calls Make_Empty. Which recursively empties the nodes starting at the Root. Root and Last_Find are set to null.
- **Assignment operator:** The assignment operator follows the usual procedure, first calling Make_Empty to reclaim any memory, and then making a copy of the Right-Hand-Side.
- **Make_Empty:** Empties the tree by calling the private Make_Empty and then sets Root and Last_Find to null.
- **Print_Tree:** Calls the private Print_Tree on the Root with a starting depth of 0.
- **Find:** Calls the private Find with the target value and the Root as arguments.
- **Insert:** Calls the private Insert with the target value and the Root as arguments.
- **Remove:** Calls the private Remove with the target value and the Root as arguments.
- **Height:** Returns the height of the tree (go figure?)

Member Function
Assignment Operator:

```
template <class Etype>
const AvlTree<Etype> &
AvlTree<Etype>::
operator = ( const AvlTree<Etype> & Value )
{
  if( this != &Value ) {
    Make_Empty( Root );
    Root = Copy( Value.Root );
    Last_Find = Root;
  }
  return *this;
}
```

Member Function
Copy Function:

```
template <class Etype>
Avl_Node<Etype> *
AvlTree<Etype>::
Copy( const Avl_Node<Etype> *T )
{
  if( T != 0 )
    return
      new Avl_Node<Etype> ( T->Element,
                            Copy( T->Left ),
                            Copy( T->Right ),
                            T->Height );
  else
    return 0;
}
```

Member Function
Make_Empty Function:

```
template <class Etype>
void
AvlTree<Etype>::
Make_Empty( Avl_Node<Etype> * & T )
{
  if( T != 0 ) {
    Make_Empty( T->Left );
    Make_Empty( T->Right );
    delete T;
    T = 0;
  }
}
```

Member Function
Find Function:

```
template <class Etype>
Avl_Node<Etype>*
AvlTree<Etype>::
Find( const Etype & X, Avl_Node<Etype> * T ) const
{
  while ( T != 0 ) {
    if ( X < T->Element )
      T = T->Left;
    else if ( X > T->Element )
      T = T->Right;
    else
      return T;
  }
  return 0;

}
```

Member Function
Print_Tree Function:

```
template <class Etype>
void
AvlTree<Etype>::
Print_Tree( Avl_Node<Etype> *T, int depth ) const
{
   if( T != 0 ) {
      Print_Tree( T->Right, depth+1 );
      for ( int i=0; i<depth; i++ ) std::cout << "   ";
          std::cout << T->Element << std::endl ;
      Print_Tree( T->Left, depth+1 );
   }
}
```

Member Function
Find_Min Function:

```
template <class Etype>
Avl_Node<Etype>*
AvlTree<Etype>::
Find_Min( Avl_Node<Etype> *T ) const
{
  while ( T != 0 ) {
    if ( T->Left == 0 ) break;
    T = T->Left;
  }
  return T;   // returns 0 if the tree is empty
}
```

Friend Function
Node_Ht Function:

```
template <class Etype>
int
Node_Ht( Avl_Node<Etype> *P )
{
        if ( P )
                return P->Height;
        else  {
                // null pointers are height -1
                return -1;
        }
}
```

Friend Function
Calculate_Height Function:

```
template <class Etype>
void
Calculate_Height( Avl_Node<Etype> *P )
{
        P->Height = 1 + Max( Node_Ht( P->Left ),
                             Node_Ht( P->Right ) );
}
```

Friend Function
Balance_Left Function:

```
//  For a given node T, check whether the left
//  subtree has gotten too large relative to the
//  right subtree.  This could occur if a node
//  has been added to the left subtree or deleted
//  from the right.  If the left subtree has gotten
//  too big, then either do a single or a double
//  rotation.  If the left subtree has not gotten
//  too big, then just calculate the height of T.
template <class Etype>
void
Balance_Left( Avl_Node<Etype> * & T )
{
  // T's left child
  Avl_Node<Etype> * k1;  // T's left child
  // the right child of T's left child
  Avl_Node<Etype> * k2;
```

Friend Function
Balance_Left Function:

```
if (Node_Ht(T->Left)-Node_Ht(T->Right) == 2 ) {
  if (Node_Ht(T->Left->Right)>Node_Ht(T->Left->Left)) {
      //  Double rotation from the left.
#ifdef DEMO
      std::cout << "Double rotation from left.\n";
#endif
      k1 = T->Left;
      k2 = T->Left->Right;
      k1->Right = k2->Left;
      T->Left = k2->Right;
      k2->Left = k1;
      k2->Right = T;
      k1->Height = 1 + Max( Node_Ht( k1->Left ),
                            Node_Ht( k1->Right ) );
      T->Height = 1 +  Max( Node_Ht( T->Left ),
                            Node_Ht( T->Right ) );
      k2->Height = 1 + Max( Node_Ht( k2->Left ),
                            Node_Ht( k2->Right ) );
      T = k2;
  }
```

Friend Function
Balance_Left Function:

```
    else {
      //  Single rotation from the left.
#ifdef DEMO
      std::cout << "Single rotation from the left.\n";
#endif
      k1 = T->Left;
      T->Left = k1->Right;
      k1->Right = T;
      T->Height = 1 +  Max( Node_Ht( T->Left ),
                            Node_Ht( T->Right ) );
      k1->Height = 1 + Max( Node_Ht( k1->Left ),
                            Node_Ht( k1->Right ) );
      T = k1;
    }
  }
  else {
    Calculate_Height( T );
  }
}
```

Friend Function
Balance_Right Function:

```
//  For a given node T, check whether the right
//  subtree has gotten too large relative to the
//  left subtree.  This could occur if a node
//  has been added to the right subtree or deleted
//  from the left.  If the right subtree has gotten
//  too big, then either do a single or a double
//  rotation.  If the right subtree has not gotten
//  too big, then just calculate the height of T.
template <class Etype>
void
Balance_Right( Avl_Node<Etype> * & T )
{
  Avl_Node<Etype> * k1;
  Avl_Node<Etype> * k2;
```

Friend Function
Balance_Right Function:

```
if (Node_Ht(T->Right)-Node_Ht(T->Left) == 2) {
  if (Node_Ht(T->Right->Left)>Node_Ht(T->Right->Right)) {
#ifdef DEMO
      std::cout << "Double rotation from right.\n";
#endif
      k1 = T->Right;
      k2 = T->Right->Left;
      k1->Left = k2->Right;
      T->Right = k2->Left;
      k2->Right = k1;
      k2->Left = T;
      k1->Height = 1 + Max( Node_Ht( k1->Left ),
                            Node_Ht( k1->Right ) );
      T->Height = 1 + Max( Node_Ht( T->Left ),
                           Node_Ht( T->Right ) );
      k2->Height = 1 + Max( Node_Ht( k2->Left ),
                            Node_Ht( k2->Right ) );
      T = k2;
    }
```

Friend Function
Balance_Right Function:

```
    else {
#ifdef DEMO
      std::cout << "Single rotation from the right.\n";
#endif
      k1 = T->Right;
      T->Right = k1->Left;
      k1->Left = T;
      T->Height = 1 + Max( Node_Ht( T->Left ),
                           Node_Ht( T->Right ) );
      k1->Height = 1 + Max( Node_Ht( k1->Left ),
                            Node_Ht( k1->Right ) );
      T = k1;
    }
  }
  else {
    Calculate_Height( T );
  }
}
```

Member Function
Insert Function:

```
//  Insert a value into an AVL tree.
//  This function is much simplified
//  by the use of the Balance_Left
//  and Balance_Right functions.
template <class Etype>
void AvlTree<Etype>::
Insert( const Etype & X, Avl_Node<Etype> * & T ) {
  if( T == 0 ) {    // Create a one node tree.
    T = new Avl_Node<Etype>( X );
    if( T == 0 )
      Error( "Out of space" );
    return;
  }
  if( X < T->Element ) {
    Insert( X, T->Left );
    Balance_Left( T );
  }
  else if( X > T->Element ) {
    Insert( X, T->Right );
    Balance_Right( T );
  }
  // Else X is in the tree already, so we'll do nothing.
}
```

Member Function
Remove Function:

```
//  Remove a value from an AVL tree.
//  This function is much simplified
//  by the use of the Balance_Left
//  and Balance_Right functions.
template <class Etype> void
AvlTree<Etype>::
Remove( const Etype & X, Avl_Node<Etype> * & T ) {
  Avl_Node<Etype> * Tmp;

  if ( T == 0 ) {
    Error( "Element not found in \"Remove\" " );
  }
  else if ( X < T->Element ) {
    // Look in left subtree
    Remove( X, T->Left );
    // Right subtree may now be too tall
    Balance_Right( T );
  }
  else if ( X > T->Element ) {
    // Look in left subtree
    Remove( X, T->Right );
    // Left subtree may now be too tal
    Balance_Left( T );
  }
```

Member Function
Remove Function:

```
  // Element is found.
  // Remove now depends on whether the node
  // containing X has 0, 1, or 2 children
  //
  else if ( T->Left != 0 && T->Right != 0 ) {
    Tmp = Find_Min( T->Right );
    T->Element = Tmp->Element;
    Remove( T->Element, T->Right );
    // Left subtree may now be too tal
    Balance_Left( T );
  }
  else {
    Tmp = T;
    // No height recalculation needed here.
    if ( T->Left == 0 )
      T = T->Right;
    else
      T = T->Left;
    delete Tmp;
  }
}
```