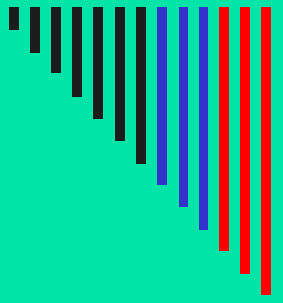


Chapter 8

AVL Search Trees

Objectives

Upon completion you will be able to:

- **Explain the differences between a BST and an AVL tree**
- **Insert, delete, and process nodes in an AVL tree**
- **Understand the operation of the AVL tree ADT**
- **Write application programs using an AVL tree**

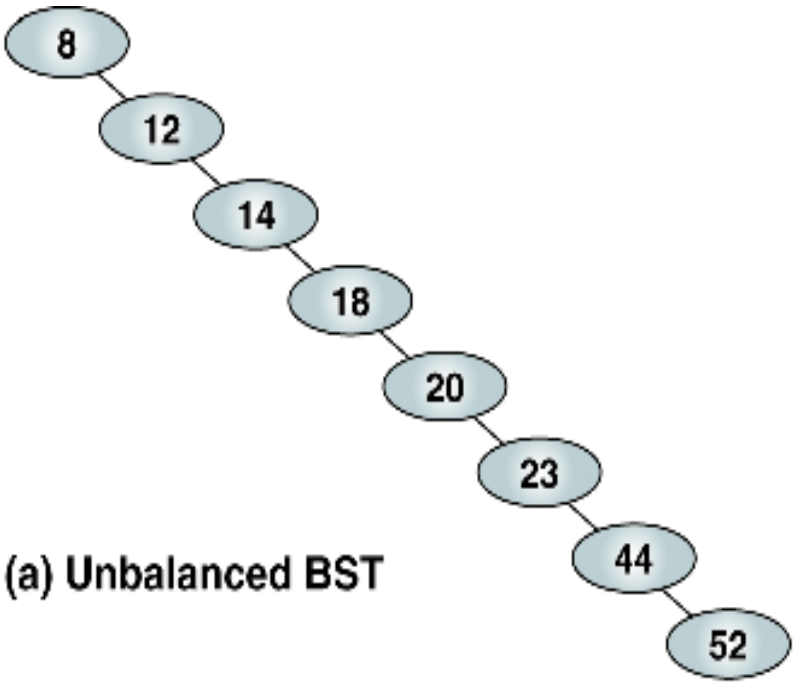
8-1 AVL Tree Basic Concepts

An AVL (Adelson-Velskii and Landis) tree is a BST that is guaranteed to always be balanced. We begin with a discussion of its basic structural differences and then discuss the four different balancing situations that can occur when data are inserted or deleted.

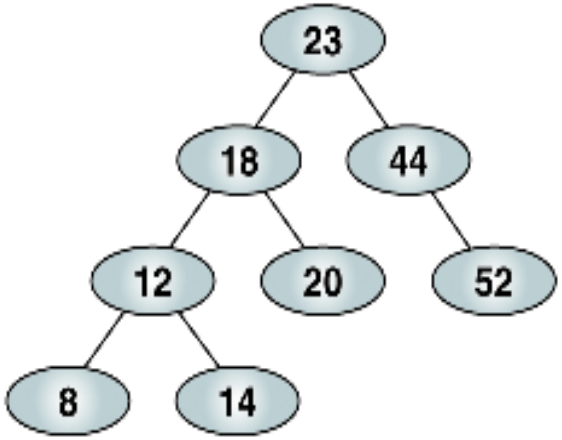
AVL tree = BST + balancing

- **AVL Tree Balance Factor**
- **Balancing Trees**

Two Binary Trees



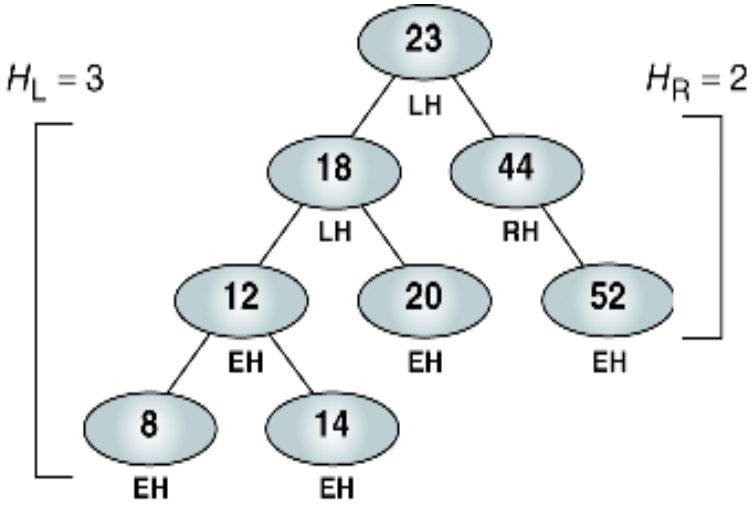
(a) Unbalanced BST



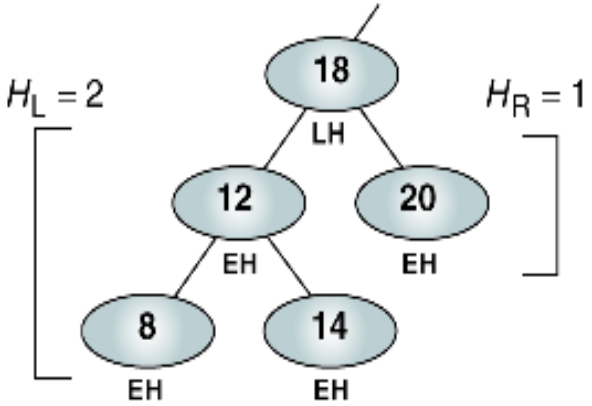
(b) AVL tree

AVL Tree

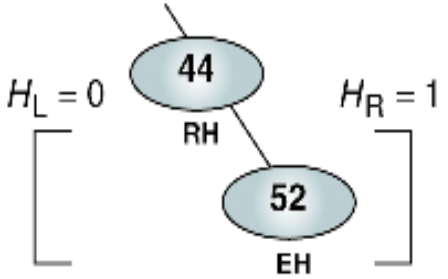
LH: left high (+1)
EH: even high (0)
RH: right high (-1)



(a) Tree 23 appears balanced: $H_L - H_R = 1$

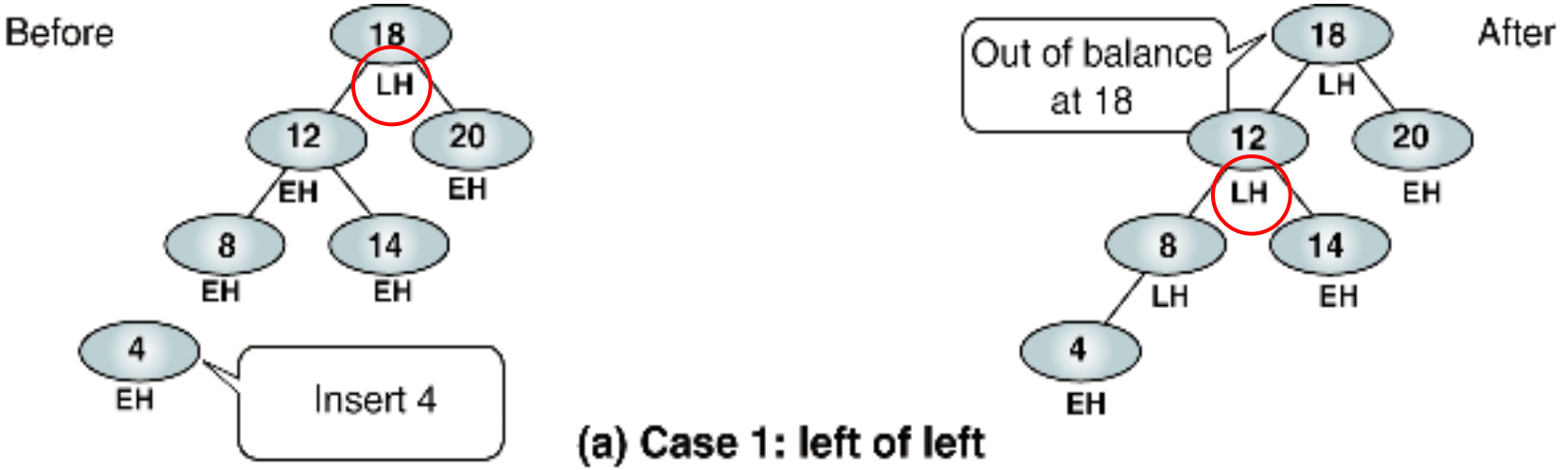


(b) Subtree 18 appears balanced:
$H_L - H_R = 1$

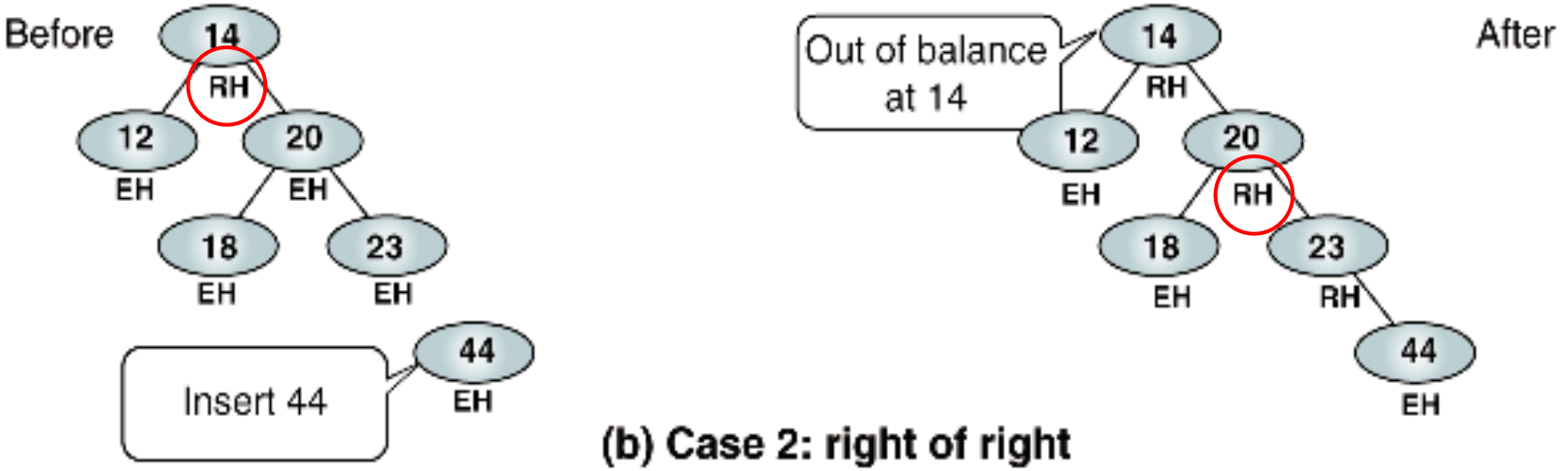


(c) Subtree 44 is balanced:
$| H_L - H_R | = 1$

Out-of-balance AVL Tree

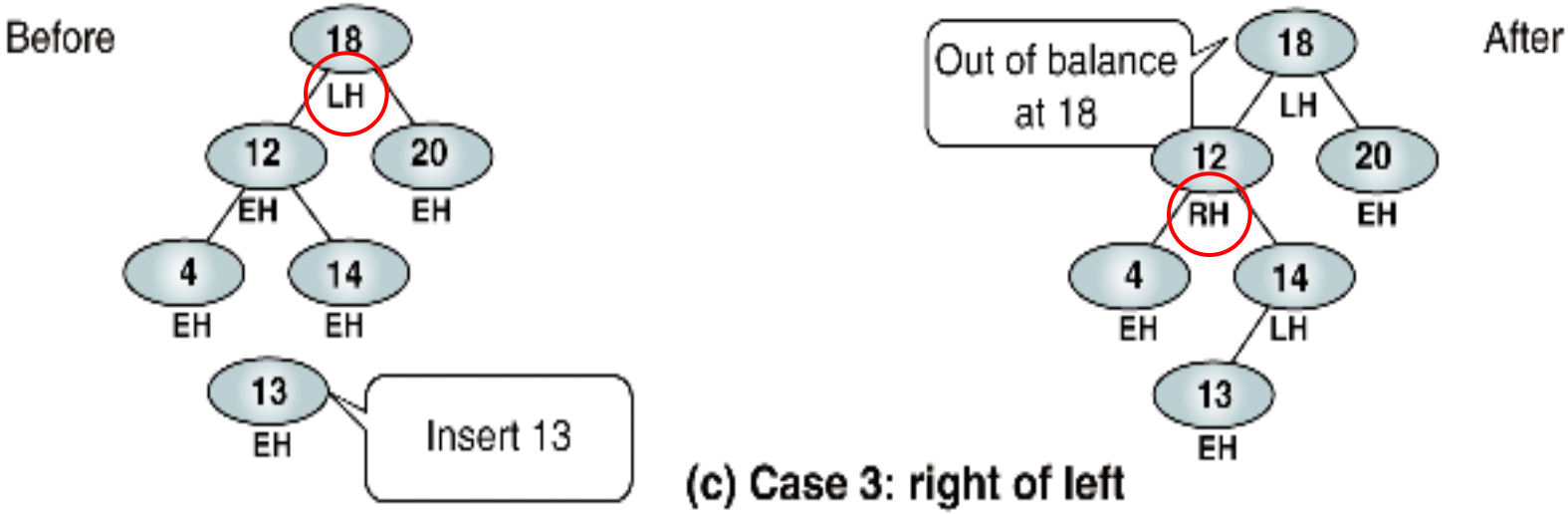


(a) Case 1: left of left

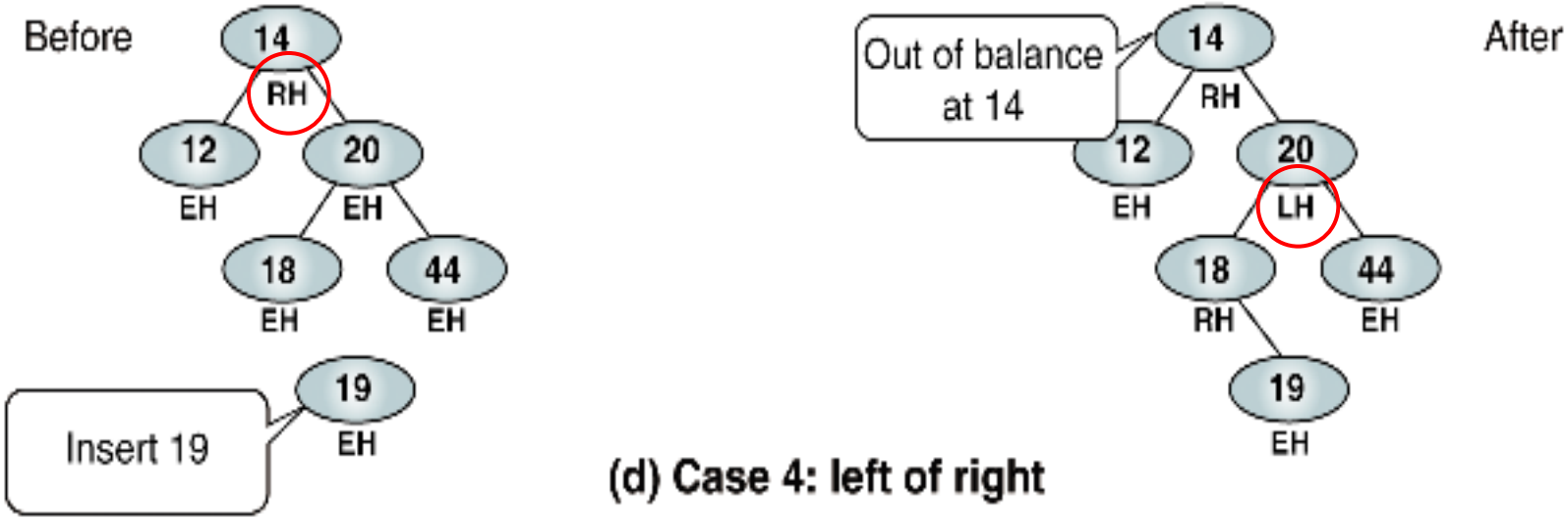


(b) Case 2: right of right

Out-of-balance AVL Tree (cont.)

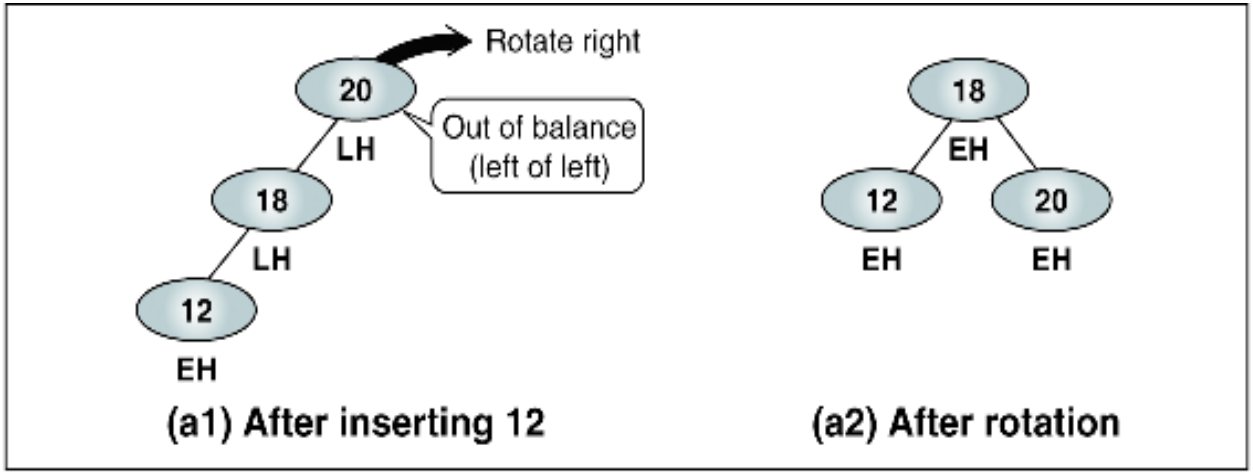


(c) Case 3: right of left

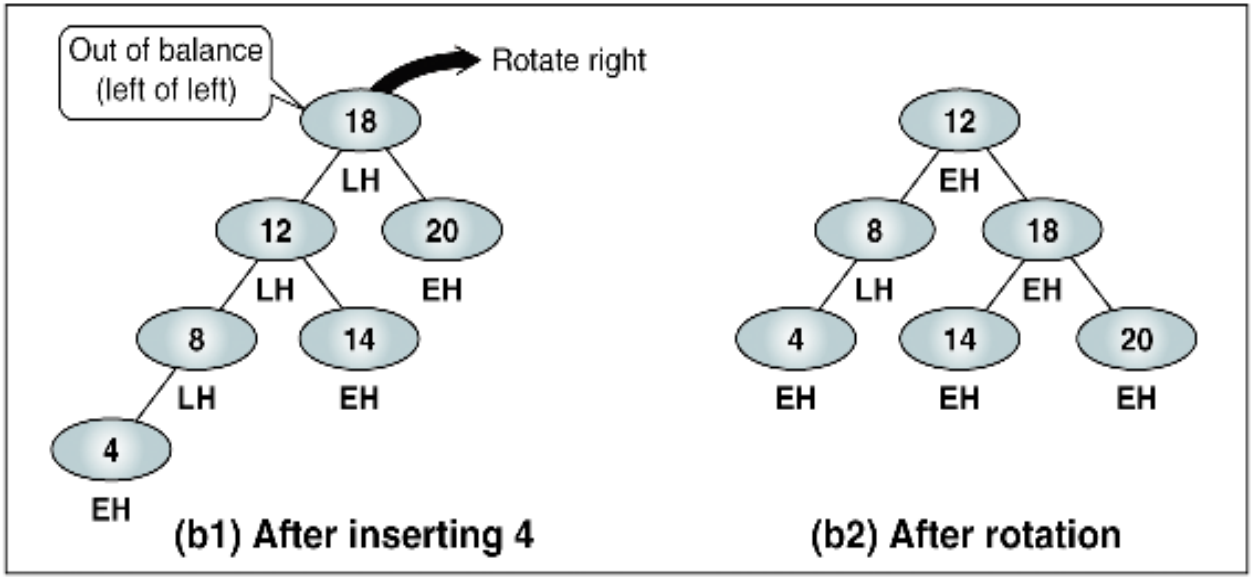


(d) Case 4: left of right

Left of Left – Single Rotation Right

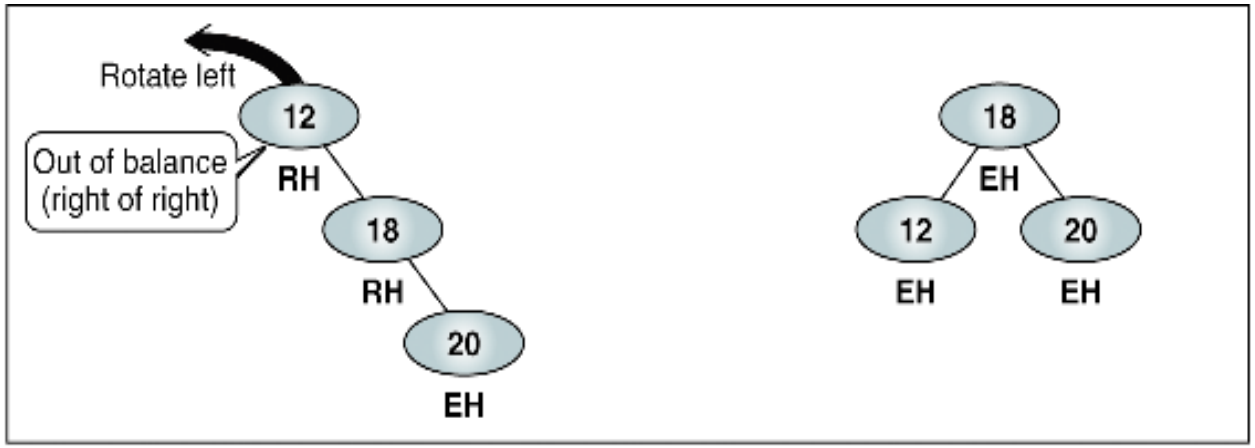


(a) Simple right rotation

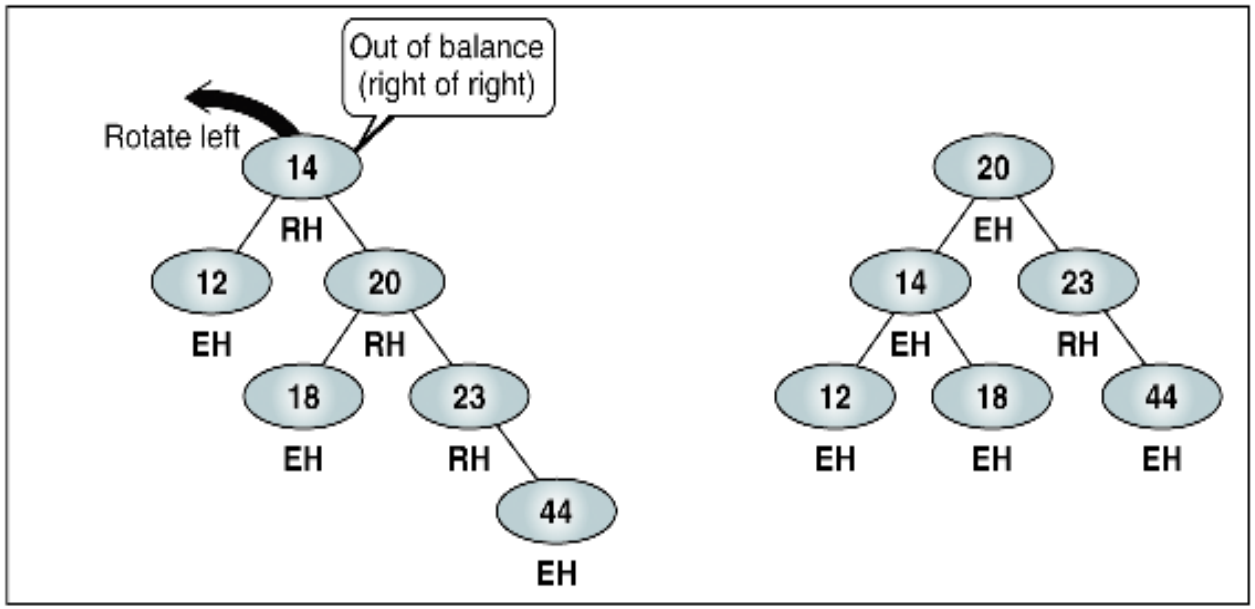


(b) Complex right rotation

Right of Right – Single Rotation Left

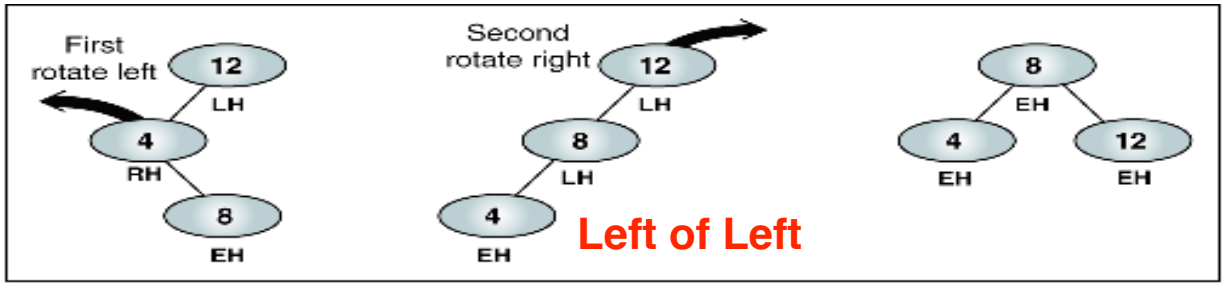


(a) Simple left rotation

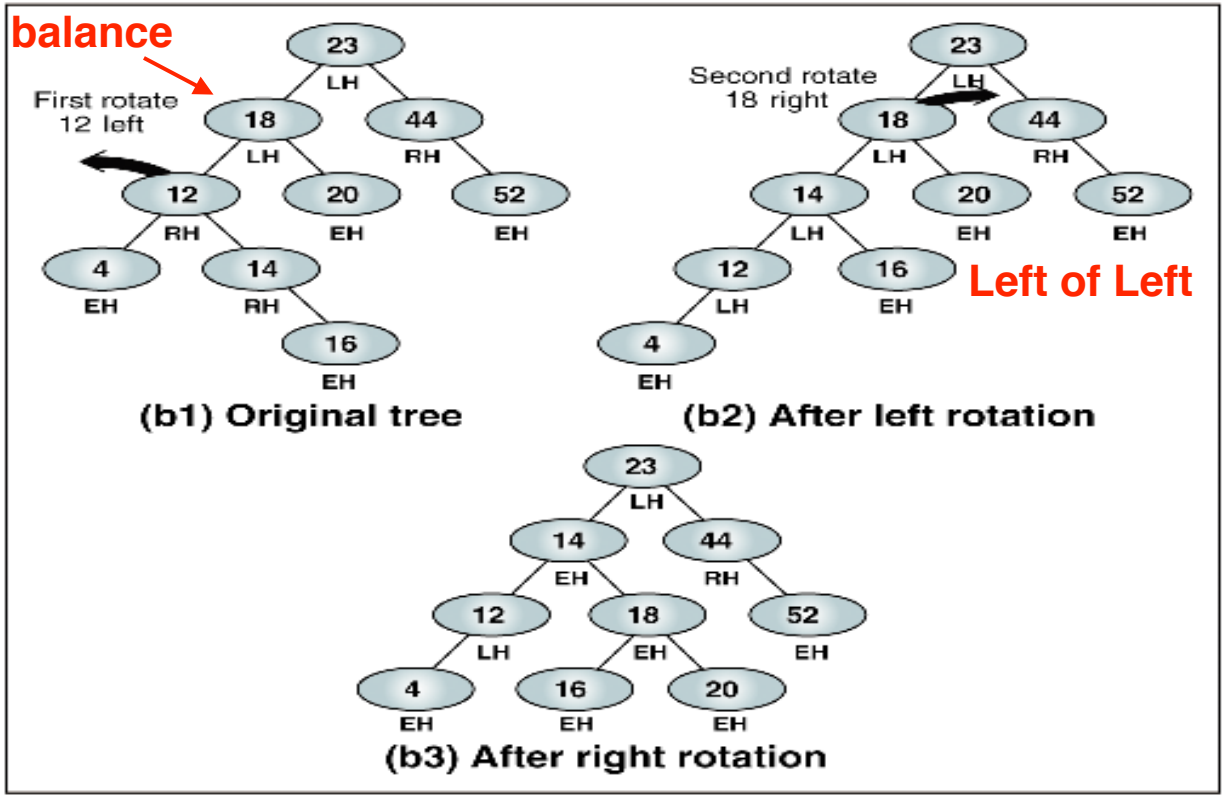


(b) Complex left rotation

Right of Left – Double Rotation Right



(a) Simple double rotation right



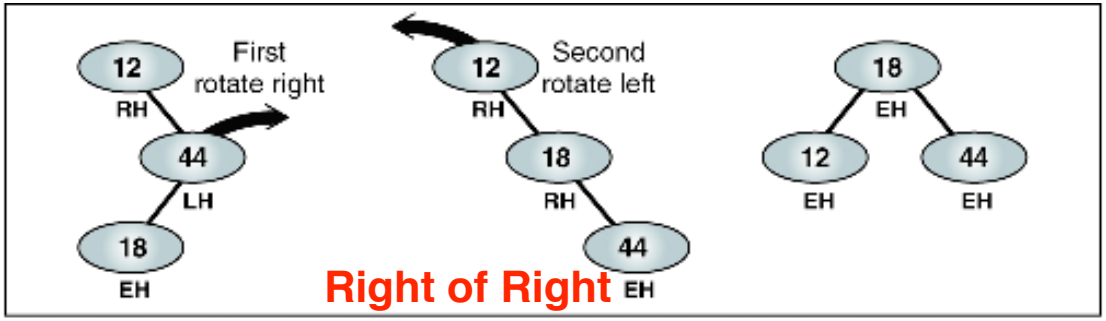
(b1) Original tree

(b2) After left rotation

(b3) After right rotation

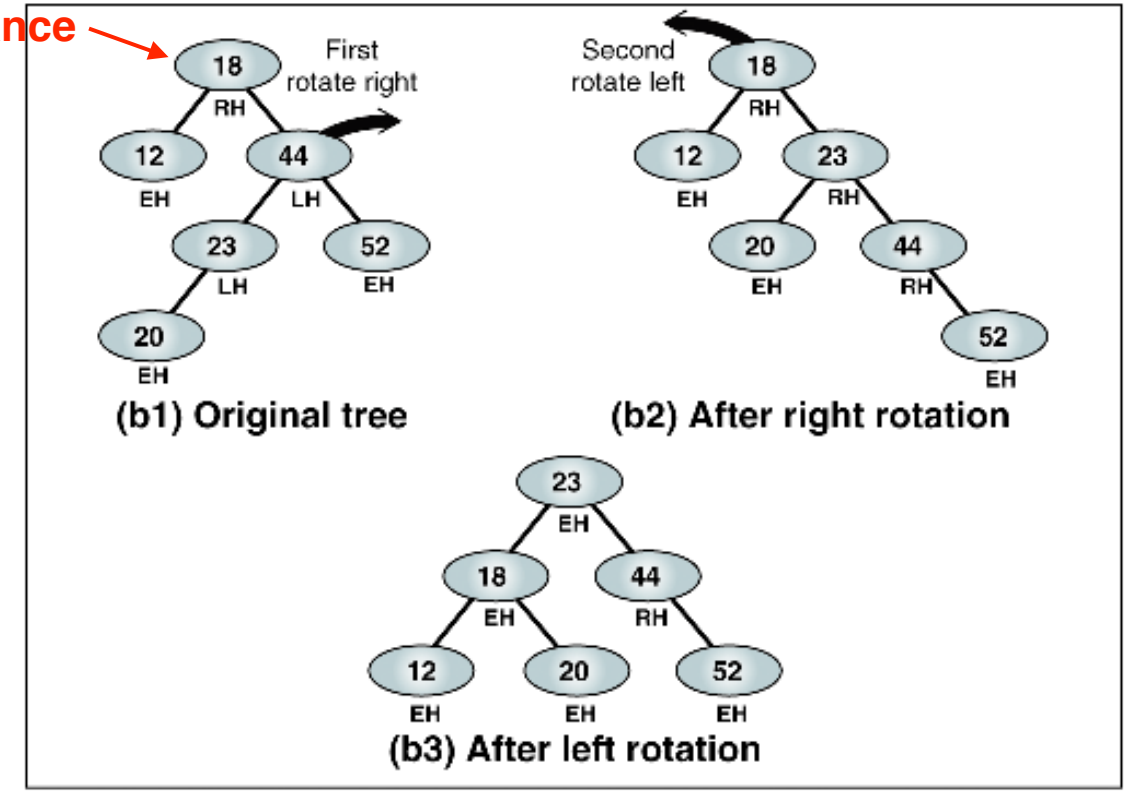
(b) Complex double rotation right

Left of Right – Double Rotation Right



(a) Simple double rotation right

out of balance



(b1) Original tree

(b2) After right rotation

(b3) After left rotation

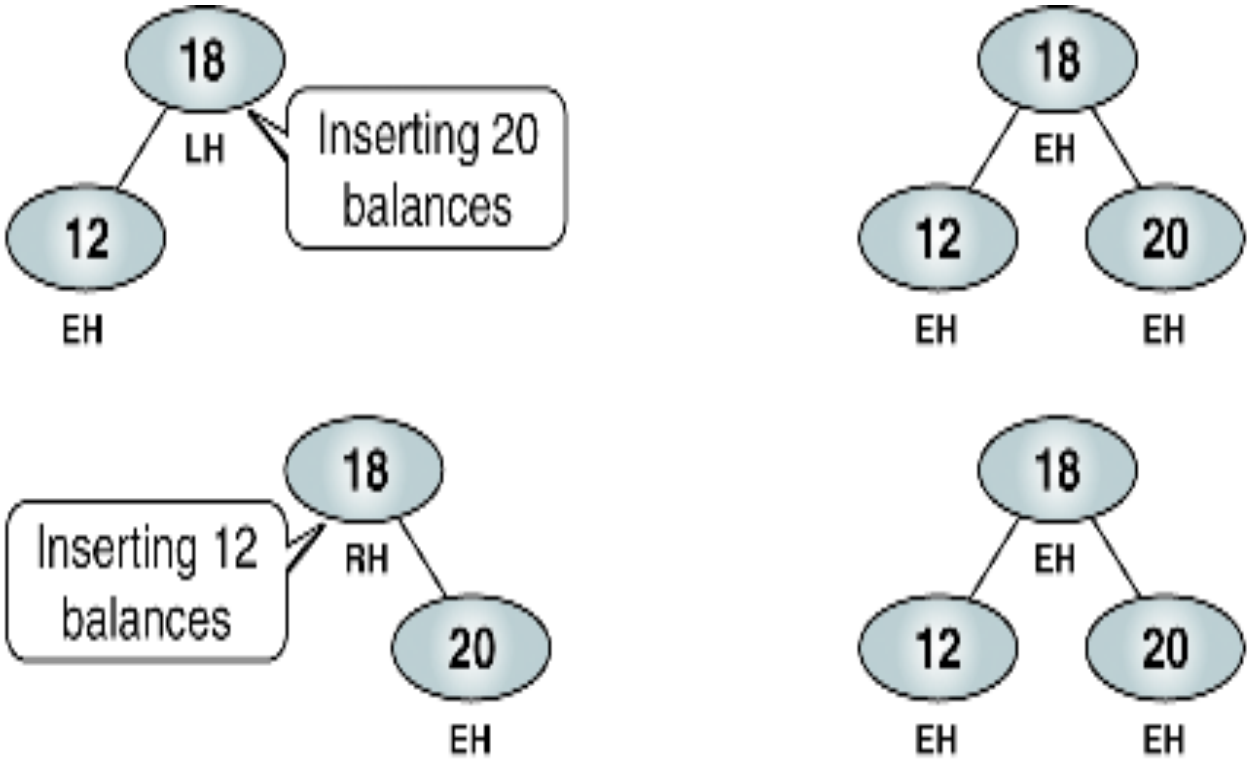
(b) Complex double rotation right

8-2 AVL Tree Implementations

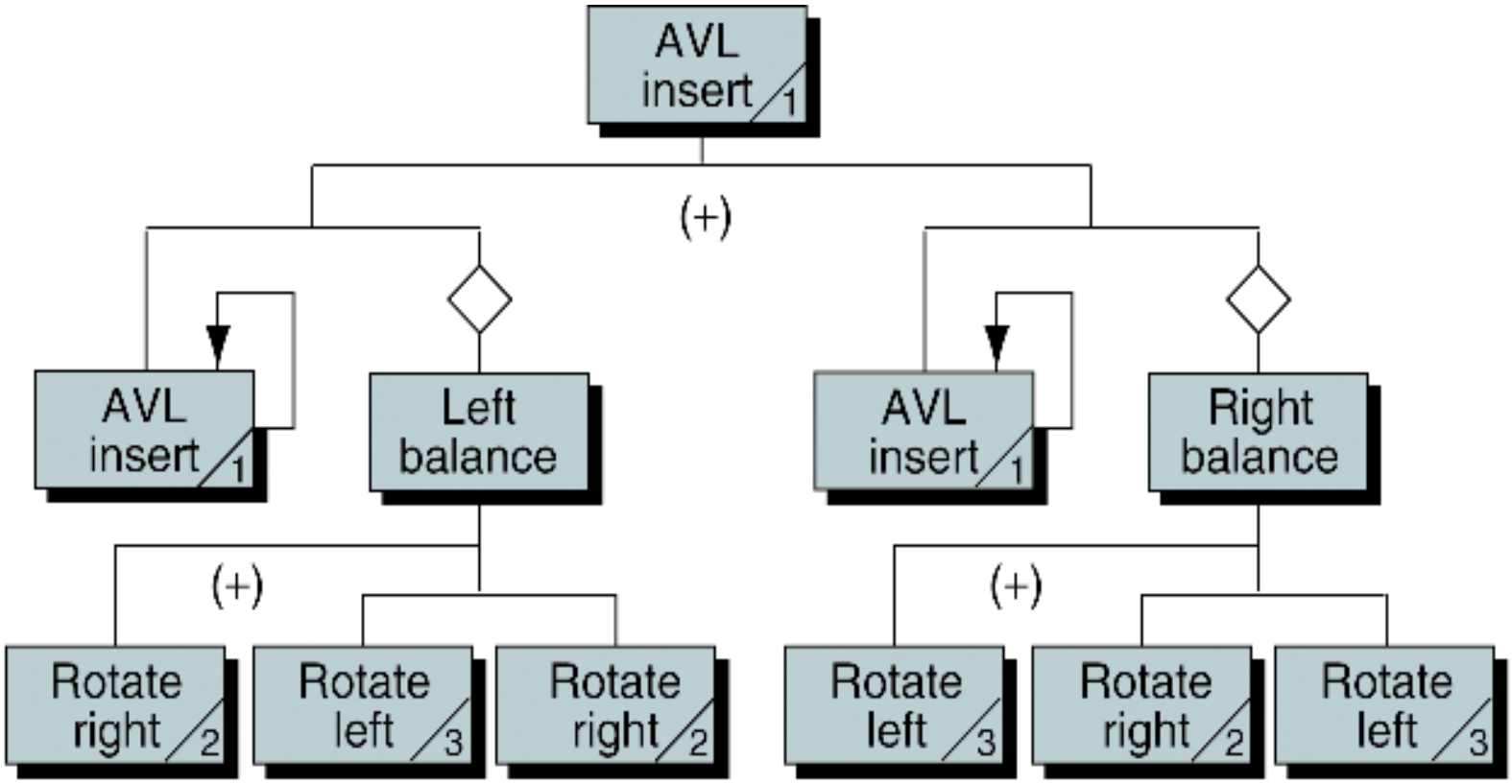
Insertion and deletion in an AVL tree follow the same basic rules for insertion and deletion in a BST. The difference lies in the tree balancing, which occurs as we back out of the tree. In this section we develop the insertion and deletion algorithms for a AVL tree, including algorithms to balance the tree.

- **Insert into AVL Tree**
- **AVL Tree Delete Algorithm**

Automatic AVL Tree Balancing



AVL Tree Insert Design



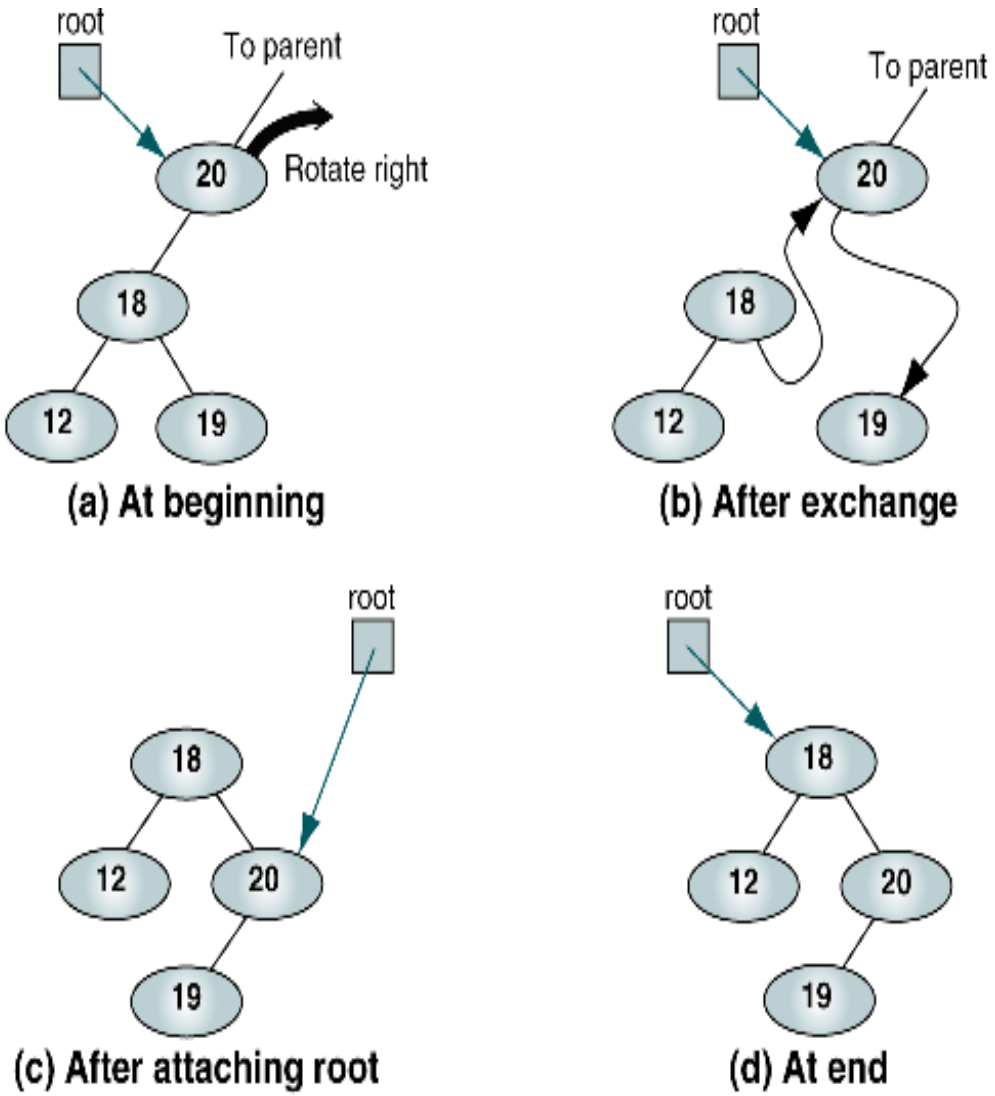
AVL Tree Insert

```
Algorithm AVLInsert (root, newData)
Using recursion, insert a node into an AVL tree.
   Pre    root is pointer to first node in AVL tree/subtree
          newData is pointer to new node to be inserted
   Post   new node has been inserted
   Return root returned recursively up the tree
 1 if (subtree empty)
   Insert at root
   1   insert newData at root
   2   return root
 2 end if
 3 if (newData < root)
   1   AVLInsert (left subtree, newData)
   2   if (left subtree taller)
       1   leftBalance (root)
   3   end if
 4 else
   New data >= root data
   1   AVLInsert (right subtree, newPtr)
   2   if(right subtree taller)
       1   rightBalance (root)
   3   end if
 5 end if
 6 return root
end AVLInsert
```

AVL Tree Left Balance

```
Algorithm leftBalance (root)
This algorithm is entered when the root is left high (the
left subtree is higher than the right subtree).
   Pre    root is a pointer to the root of the [sub]tree
   Post   root has been updated (if necessary)
1  if (left subtree high)
   1   rotateRight (root)
2  else
   1   rotateLeft (left subtree)
   2   rotateRight (root)
3  end if
end leftBalance
```

AVL Tree Rotate Right



Rotate AVL Tree Right and Left

```
Algorithm rotateRight (root)
This algorithm exchanges pointers to rotate the tree right.
   Pre    root points to tree to be rotated
   Post   node rotated and root updated
 1 exchange left subtree with right subtree of left subtree
 2 make left subtree new root
end rotateRight

Algorithm rotateLeft (root)
This algorithm exchanges pointers to rotate the tree left.
   Pre    root points to tree to be rotated
   Post   node rotated and root updated
 1 exchange right subtree with left subtree of right subtree
 2 make right subtree new root
end rotateLeft
```

AVL Tree Delete

```
Algorithm AVLDelete (root, dltKey, success)
This algorithm deletes a node from an AVL tree and
rebalances if necessary.
   Pre    root is a pointer to a [sub]tree
          dltKey is the key of node to be deleted
          success is reference to boolean variable
   Post   node deleted if found, tree unchanged if not
          success set true (key found and deleted)
             or false (key not found)
   Return pointer to root of [potential] new subtree
1  if Return (empty subtree)
      Not found
   1  set success to false
   2  return null
2  end if
3  if (dltKey < root)
   1  set left-subtree to AVLDelete(left subtree, dltKey,
                                    success)
```

AVL Tree Delete (cont.)

```
   2   if (tree shorter)
       1  set root to deleteRightBalance(root)
   3   end if
4  elseif (dltKey > root)
   1   set right subtree to AVLDelete(root->right, dltKey,
                                      success)
   2   if (tree shorter)
       1  set root to deleteLeftBalance (root)
   3   end if
5  else
       Delete node found--test for leaf node
   1   save root
```

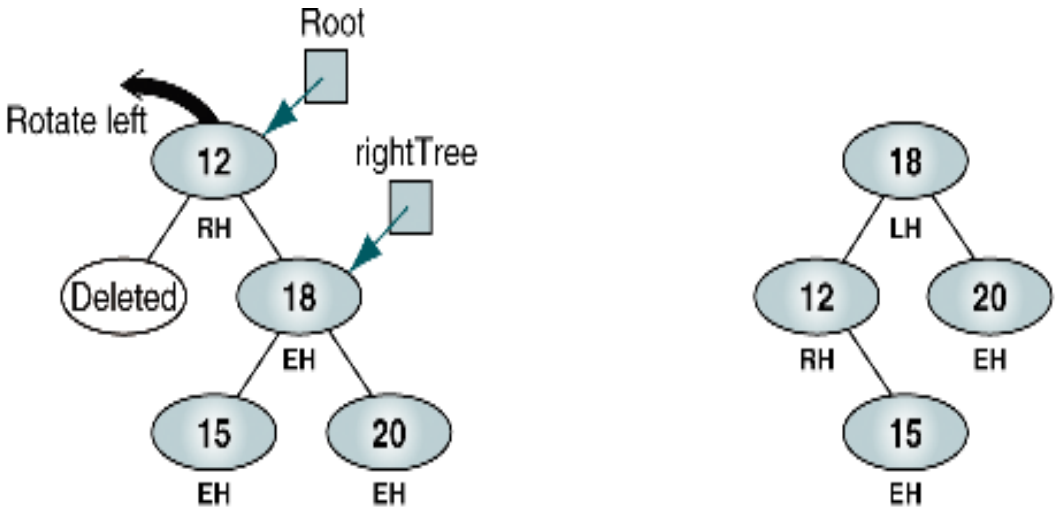
AVL Tree Delete (cont.)

```
   2   if (no right subtree)
       1   set success to true
       2   return left subtree
   3   elseif (no left subtree)
           Have right but no left subtree
       1   set success to true
       2   return right subtree
   4   else
           Deleted node has two subtrees
           Find substitute--largest node on left subtree
       1   find largest node on left subtree
       2   save largest key
       3   copy data in largest to root
       4   set left subtree to AVLDelete(left subtree,
                                         largest key, success)
       5   if (tree shorter)
           1   set root to dltRightBal (root)
       6   end if
   5   end if
 6   end if
 7   return root
end AVLDelete
```

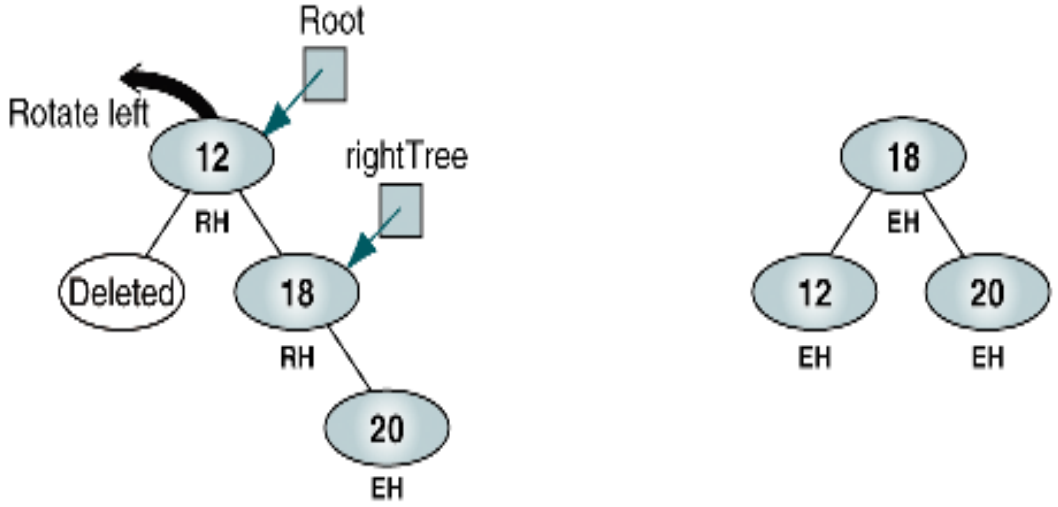
AVL Tree Delete Right Balance

```
Algorithm deleteRightBalance (root)
The [sub]tree is shorter after a deletion on the left branch.
If necessary, balance the tree by rotating.
   Pre    tree is shorter
   Post   balance restored
   Return new root
1  if (tree not balanced)
      No rotation required if tree left or even high
   1  set rightOfRight to right subtree
   2  if (rightOfRight left high)
         Double rotation required
      1  set leftOfRight to left subtree of rightOfRight
         Rotate right then left
      2  right subtree = rotateRight (rightOfRight)
      3  root          = rotateLeft (root)
   3  else
         Single rotation required
      1  set root to rotateLeft (root)
   4  end if
2  end if
3  return root
end deleteRightBalance
```

AVL Tree Delete Balance



(a) Right subtree is even balanced



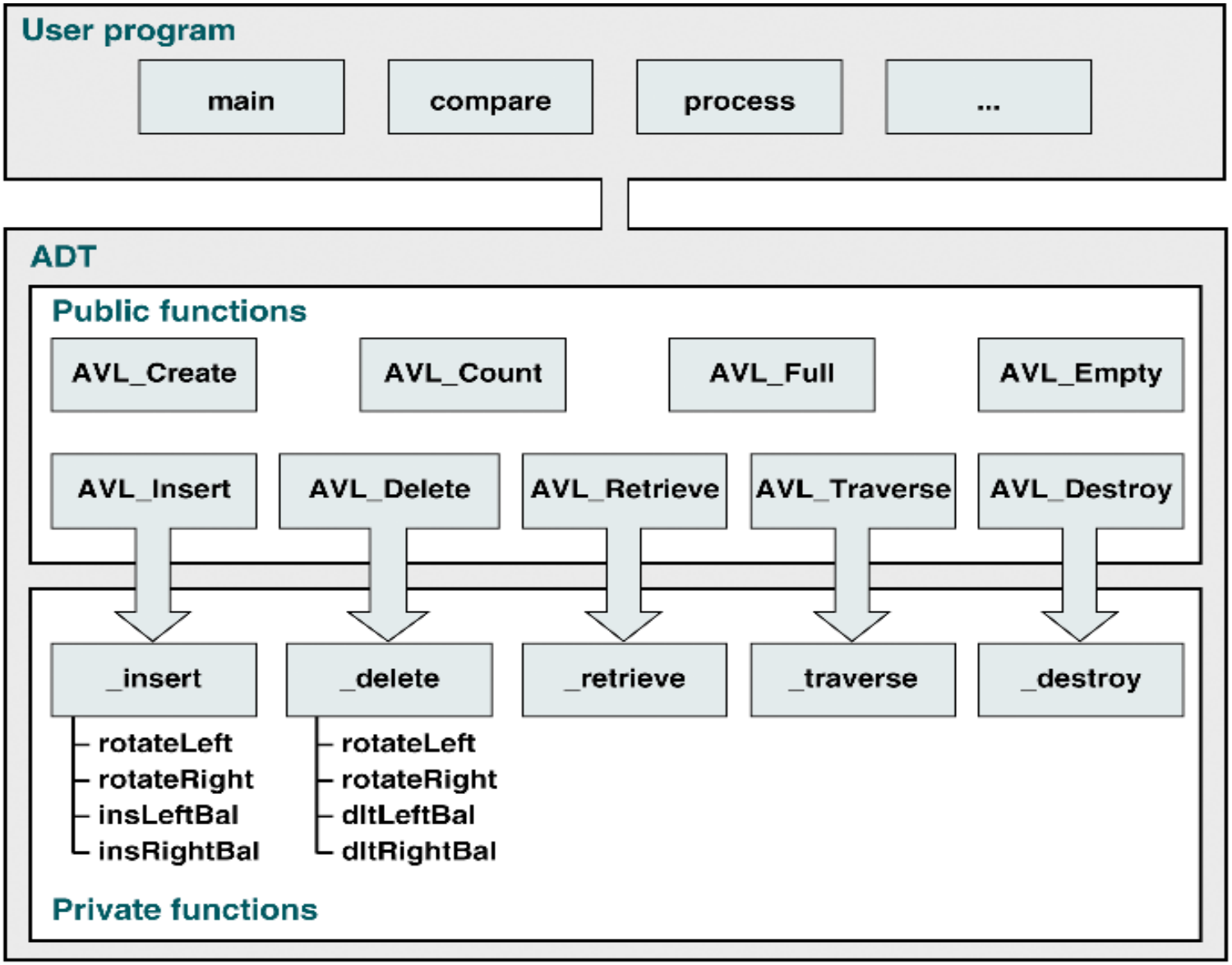
(b) Right subtree is not even balanced

8-3 AVL Tree Abstract Data Type

In this section we discuss and write the C code for the AVL ADT. We begin with a discussion of the data structure and then develop 17 functions for the ADT. Some functions are left for student development.

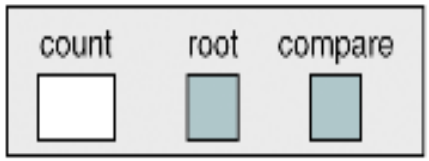
- **Insert into AVL Tree**
- **AVL Tree Delete Algorithm**

AVL Tree Design

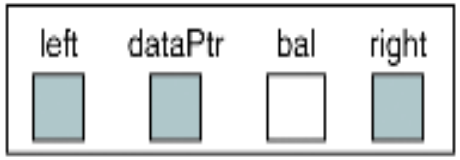


AVL Tree Data Structure

AVL-TREE



NODE




```
typedef struct
{
  int   count;
  NODE* root;
  int   (*compare) (void* argu1, void* argu2);
} AVL_TREE;

typedef struct node
{
  struct node* left;
  void*        dataPtr;
  int          bal;
  struct node* right;
} NODE;
```

AVL Tree Declarations

```
 1  #define LH +1                                  // Left High
 2  #define EH  0                                  // Even High
 3  #define RH -1                                  // Right High
 4
 5  // Structure Declarations
 6  typedef struct node
 7     {
 8      void*         dataPtr;
 9      struct node*  left;
10      int           bal;
11      struct node*  right;
12     } NODE;
13
14  typedef struct
15     {
16      int    count;
17      int   (*compare) (void* argu1, void* argu2);
18      NODE*   root;
19     } AVL_TREE;
```

AVL Tree Declarations (cont.)

```
20
21  // Prototype Declarations
22     AVL_TREE* AVL_Create
23               (int (*compare)(void* argu1, void* argu2));
24     AVL_TREE* AVL_Destroy (AVL_TREE* tree);
25
26     bool  AVL_Insert   (AVL_TREE* tree,
27                         void* dataInPtr);
28     bool  AVL_Delete   (AVL_TREE* tree,  void* dltKey);
29     void* AVL_Retrieve (AVL_TREE* tree,  void* keyPtr);
30     void  AVL_Traverse (AVL_TREE* tree,
31                         void (*process)(void* dataPtr));
32     int  AVL_Count     (AVL_TREE* tree);
33     bool AVL_Empty     (AVL_TREE* tree);
34     bool AVL_Full      (AVL_TREE* tree);
35
```

AVL Tree Declarations (cont.)

```
36    static NODE* _insert
37                   (AVL_TREE* tree, NODE* root,
38                    NODE* newPtr,   bool* taller);
39    static NODE* _delete
40                   (AVL_TREE* tree,
41                    NODE* root,      void* dltKey,
42                    bool* shorter,   bool* success);
43    static void* _retrieve
44                   (AVL_TREE* tree,
45                    void* keyPtr,    NODE* root);
46    static void _traversal
47                   (NODE* root,
48                    void (*process)(void* dataPtr));
49    static void _destroy     (NODE* root);
50
51    static NODE* rotateLeft  (NODE* root);
52    static NODE* rotateRight (NODE* root);
53    static NODE* insLeftBal  (NODE* root, bool* taller);
54    static NODE* insRightBal (NODE* root, bool* taller);
55    static NODE* dltLeftBal  (NODE* root, bool* shorter);
56    static NODE* dltRightBal (NODE* root, bool* shorter);
```

Create AVL Tree Application Interface

```
 1  /* ================= AVL_Create ================
 2     Allocates dynamic memory for an AVL tree head
 3     node and returns its address to caller.
 4        Pre    compare is address of compare function
 5               used when two nodes need to be compared
 6        Post   head allocated or error returned
 7        Return head node pointer; null if overflow
 8  */
 9  AVL_TREE* AVL_Create
10           (int  (*compare) (void* argu1, void* argu2))
11  {
12  // Local Definitions
13     AVL_TREE* tree;
14
15  // Statements
16     tree = (AVL_TREE*) malloc (sizeof (AVL_TREE));
17     if (tree)
18        {
19         tree->root    = NULL;
20         tree->count   = 0;
21         tree->compare = compare;
22        } // if
23
24     return tree;
25  }  // AVL_Create
```

Insert AVL Tree Application Interface

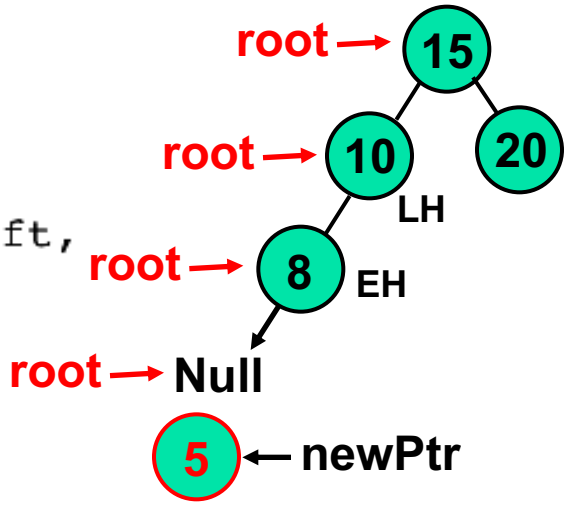
```
 1  /* ================= AVL_Insert ====================
 2     This function inserts new data into the tree.
 3        Pre    tree is pointer to AVL tree structure
 4        Post   data inserted or memory overflow
 5        Return Success (true) or Overflow (false)
 6  */
 7  bool AVL_Insert (AVL_TREE* tree, void* dataInPtr)
 8  {
 9  // Local Definitions
10     NODE* newPtr;
11     bool  forTaller;
12
13  // Statements
14     newPtr = (NODE*)malloc(sizeof(NODE));
15     if (!newPtr)
16        return false;
17
18     newPtr->bal      = EH;
19     newPtr->right    = NULL;
20     newPtr->left     = NULL;
21     newPtr->dataPtr = dataInPtr;
22
23     tree->root = _insert(tree,  tree->root,
24                    newPtr,      &forTaller);
25     (tree->count)++;
26     return true;
27  }  // AVL_Insert
```

Internal Insert Function

```
 1  /* ==================== _insert ====================
 2     This function uses recursion to insert the new data
 3     into a leaf node in the AVL tree.
 4     Pre   Application has called AVL_Insert, which passes
 5           root and data pointers.
 6     Post  Data have been inserted.
 7     Return pointer to [potentially] new root.
 8  */
 9  NODE* _insert (AVL_TREE* tree,   NODE* root,
10                 NODE*     newPtr, bool* taller)
11  {
12  // Statements
13     if (!root)
14        {
15         // Insert at root
16         root     = newPtr;
17         *taller  = true;
18         return   root;
19        } // if NULL tree
```

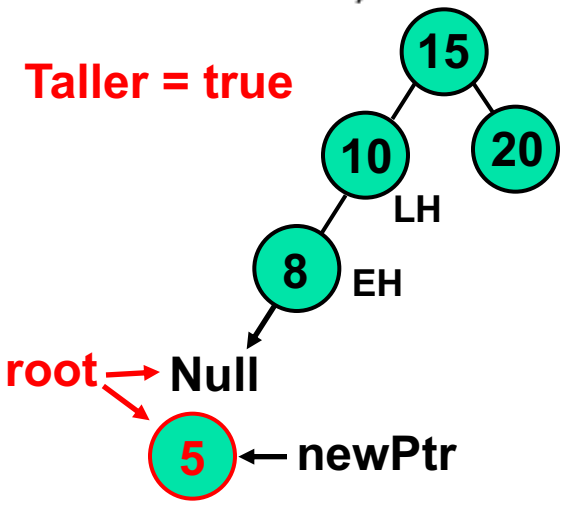
Internal Insert Function (cont.)

```
20
21      if (tree->compare(newPtr->dataPtr,
22                        root->dataPtr) < 0)
23          {
24           // newData < root -- go left
25          root->left = _insert(tree,   root->left,
26                               newPtr, taller);
27           if (*taller)
28              // Left subtree is taller
29              switch (root->bal)
30                  {
31                   case LH:  // Was left high--rotate
32                             root = insLeftBal (root, taller);
33                             break;
34
35                   case EH:  // Was balanced--now LH
36                             root->bal = LH;
37                             break;
38
39                   case RH:  // Was right high--now EH
40                             root->bal = EH;
```



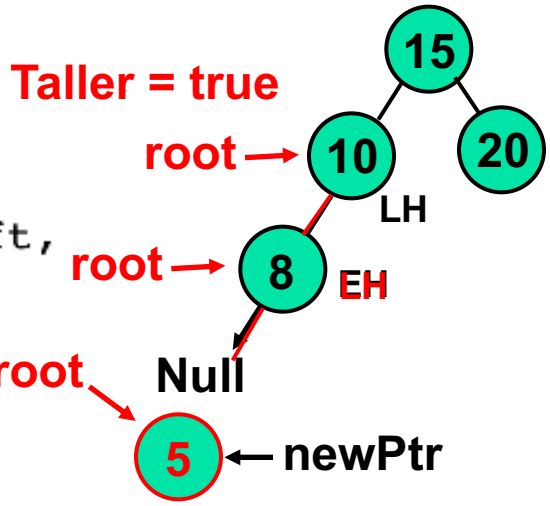
Internal Insert Function

```
 1  /* ====================  _insert  ====================
 2     This function uses recursion to insert the new data
 3     into a leaf node in the AVL tree.
 4     Pre   Application has called AVL_Insert, which passes
 5           root and data pointers.
 6     Post  Data have been inserted.
 7     Return pointer to [potentially] new root.
 8  */
 9  NODE* _insert (AVL_TREE* tree,   NODE* root,
10                 NODE*     newPtr, bool* taller)
11  {
12  // Statements
13     if (!root)
14        {
15         // Insert at root
16         root      = newPtr;
17         *taller   = true;
18         return    root;
19        } // if NULL tree
```



Internal Insert Function (cont.)

```
20
21      if (tree->compare(newPtr->dataPtr,
22                        root->dataPtr) < 0)
23          {
24           // newData < root -- go left
25          root->left = _insert(tree,   root->left,
26                               newPtr, taller);
27          if (*taller)
28              // Left subtree is taller
29              switch (root->bal)
30                  {
31               case LH:  // Was left high--rotate
32                         root = insLeftBal (root, taller);
33                         break;
34
35               case EH:  // Was balanced--now LH
36                         root->bal = LH;
37                         break;
38
39                   case RH:  // Was right high--now EH
40                         root->bal = EH;
```



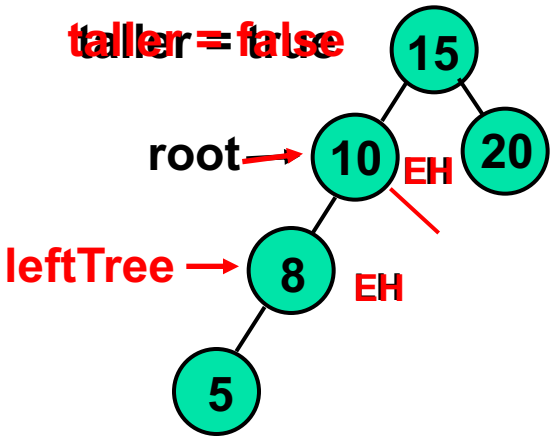
Internal Insert Function (cont.)

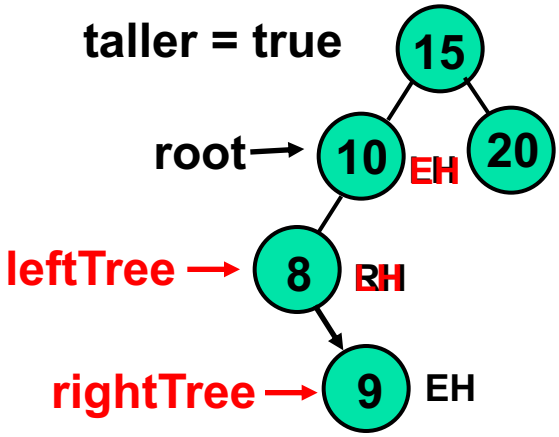
```
41                          *taller   = false;
42                          break;
43                   } // switch
44                return root;
45             } // new < node
46          else
47              // new data >= root data
48              {
49               root->right = _insert (tree,   root->right,
50                                      newPtr, taller);
51               if (*taller)
52                  // Right subtree is taller
53                  switch (root->bal)
54                      {
55                       case LH:  // Was left high--now EH
56                                 root->bal = EH;
57                                 *taller   = false;
58                                 break;
59
60                       case EH:  // Was balanced--now RH
61                                 root->bal = RH;
62                                 break;
63
64                       case RH:  // Was right high--rotate
65                                 root = insRightBal
66                                           (root, taller);
67                                 break;
68                      } // switch
69               return root;
70               } // else new data >= root data
71          return root;
72      } // _insert
```

Internal Insert Left Balance Function

```
 1  /* ================== insLeftBalance ==================
 2     Tree out-of-balance to the left. This function
 3     rotates the tree to the right.
 4        Pre  The tree is left high
 5        Post Balance restored; return potentially new root
 6  */
 7     NODE* insLeftBal (NODE* root, bool* taller)
 8  {
 9  // Local Definitions
10     NODE* rightTree;
11     NODE* leftTree;
12
13  // Statements
14     leftTree = root->left;
15     switch (leftTree->bal)
16         {
17        case LH: // Left High--Rotate Right
18                 root->bal      = EH;
19                 leftTree->bal = EH;
20
21                 // Rotate Right
22                 root      = rotateRight (root);
23                 *taller   = false;
24                 break;
25         case EH: // This is an error
26                 printf ("\n\aError in insLeftBal\n");
27                 exit (100);
28        case RH: // Right High-Requires double
29                 // rotation: first left, then right
30                 rightTree = leftTree->right;
31                 switch (rightTree->bal)
32                     {
33                      case LH: root->bal      = RH;
34                               leftTree->bal = EH;
35                               break;
36                 case EH: root->bal      = EH;
37                               leftTree->bal = LH;
```

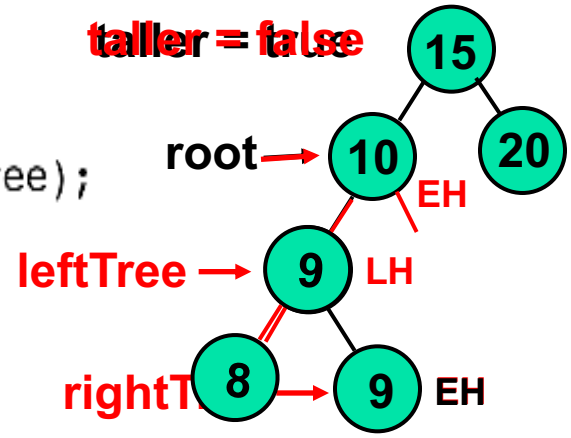
這裡有一個動畫





Internal Insert Left Balance Function (cont.)

```
38                              break;
39                        case RH: root->bal     = EH;
40                                 leftTree->bal = LH;
41                                 break;
42                      } // switch rightTree
43
44                   rightTree->bal = EH;
45                   // Rotate Left
46                   root->left = rotateLeft (leftTree);
47
48                   // Rotate Right
49                   root     = rotateRight (root);
50                   *taller  = false;
51          } // switch
52       return root;
53    } // leftBalance
```

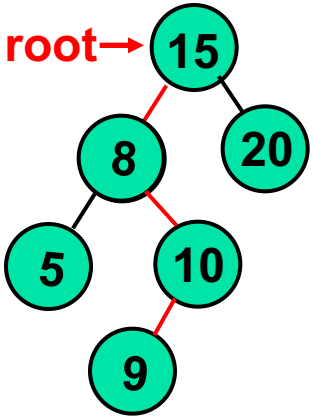


Internal Rotate Left & Right Function

```
1  /* ================== rotateLeft ==================
2     Exchanges pointers to rotate the tree to the left.
3        Pre  root points to tree to be rotated
4        Post Node rotated and new root returned
5  */
6  NODE* rotateLeft (NODE* root)
7  {
8  // Local Definitions
9     NODE* tempPtr;
10
11 // Statements
12    tempPtr        = root->right;
13    root->right    = tempPtr->left;
```

Internal Rotate Left & Right Function (cont.)

```
14      tempPtr->left  = root;
15
16      return tempPtr;
17   }  // rotateLeft
18
19   /* ================== rotateRight ==================
20      Exchange pointers to rotate the tree to the right.
21         Pre  root points to tree to be rotated
22         Post Node rotated and new root returned
23   */
24   NODE* rotateRight (NODE* root)
25   {
26   // Local Definitions
27      NODE* tempPtr;
28
29   // Statements
30      tempPtr          = root->left;
31      root->left       = tempPtr->right;
32      tempPtr->right   = root;
33
34      return tempPtr;
35   }  // rotateRight
```



Delete AVL Tree Application Interface

```
1  /* ==================== AVL_Delete ====================
2     This function deletes a node from the tree and
3     rebalances it if necessary.
4        Pre    tree initialized--null tree is OK
5               dltKey is pointer to key to be deleted
6        Post   node deleted and its space recycled
7               -or- An error code is returned
8        Return Success (true) or Not found (false)
9  */
```

Delete AVL Tree Application Interface (cont.)

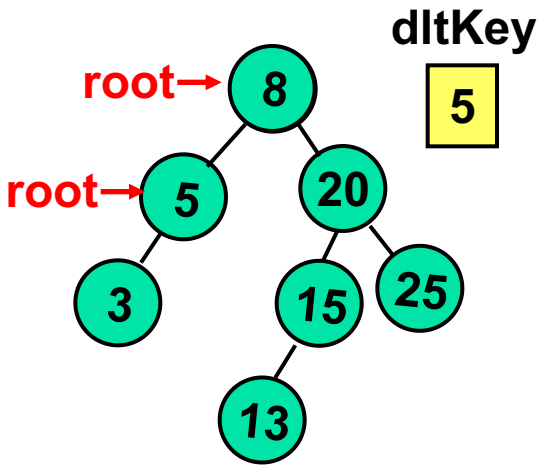
```
10  bool AVL_Delete (AVL_TREE* tree, void* dltKey)
11  {
12  // Local Definitions
13     bool  shorter;
14     bool  success;
15     NODE* newRoot;
16
17  // Statements
18  newRoot = _delete (tree,    tree->root, dltKey,
19                     &shorter, &success);
20  if (success)
21     {
22      tree->root = newRoot;
23      (tree->count)--;
24      return true;
25     } // if
26  else
27     return false;
28  }  // AVL_Delete
```

Internal Delete Function

```
1  /* ==================== _delete ====================
2     Deletes node from the tree and rebalances
3     tree if necessary.
4        Pre    tree initialized--null tree is OK.
5               dltKey contains key of node to be deleted
6               shorter indicates tree is shorter
7        Post   node is deleted and its space recycled
8               -or- if key not found, tree is unchanged
9        Return true if deleted; false if not found
10              pointer to root
11 */
12
13 NODE* _delete (AVL_TREE* tree,   NODE* root,
14                 void*     dltKey, bool* shorter,
15                 bool*     success)
16 {
```

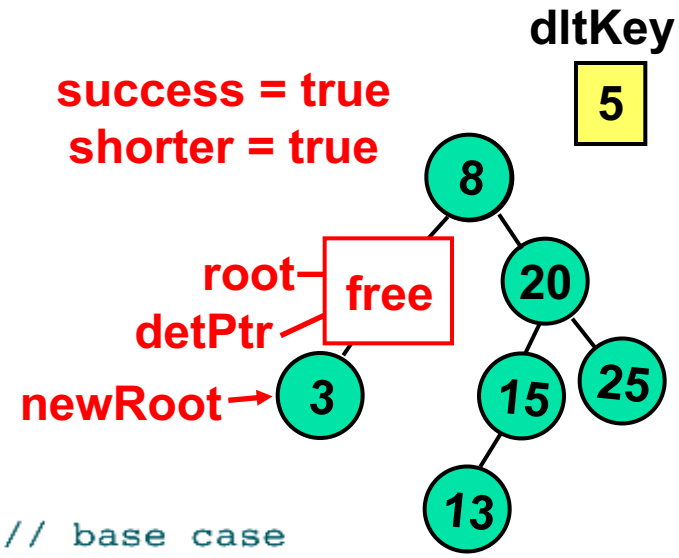
Internal Delete Function (cont.)

```
17  // Local Definitions
18     NODE* dltPtr;
19     NODE* exchPtr;
20     NODE* newRoot;
21
22  // Statements
23     if (!root)      // root = NULL
24        {
25         *shorter = false;
26         *success = false;
27         return NULL;
28        } // if
29
30     if (tree->compare(dltKey, root->dataPtr) < 0)
31         {
32          root->left = _delete (tree,
33                                root->left, dltKey,
34                                shorter,    success);
35          if (*shorter)
36              root    = dltRightBal (root, shorter);
37         } // if less
```



Internal Delete Function (cont.)

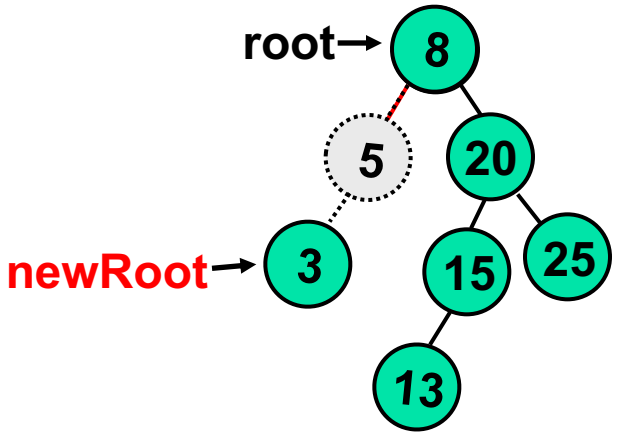
```
38      else if (tree->compare(dltKey, root->dataPtr) > 0)
39           {
40            root->right = _delete (tree,
41                                   root->right, dltKey,
42                                   shorter,     success);
43            if (*shorter)
44                 root = dltLeftBal (root, shorter);
45           } // if greater
46      else
47           // Found equal node
48           {
49            dltPtr = root;
50            if (!root->right)
51                 // Only left subtree
52                 {
53                  newRoot   = root->left;
54                  *success = true;
55                  *shorter = true;
56                  free (dltPtr);
57                  return newRoot;                  // base case
58                 } // if true
59            else
60                 if (!root->left)
61                      // Only right subtree
62                      {
63                       newRoot   = root->right;
```



Internal Delete Function (cont.)

```
17  // Local Definitions
18     NODE* dltPtr;
19     NODE* exchPtr;
20     NODE* newRoot;
21
22  // Statements
23     if (!root)
24        {
25         *shorter = false;
26         *success = false;
27         return NULL;
28        } // if
29
30     if (tree->compare(dltKey, root->dataPtr) < 0)
31         {
32          root->left = _delete (tree,
33                                root->left, dltKey,
34                                shorter,    success);
35          if (*shorter)
36              root    = dltRightBal (root, shorter);
37         } // if less
```

success = true
shorter = true



Internal Delete Function (cont.)

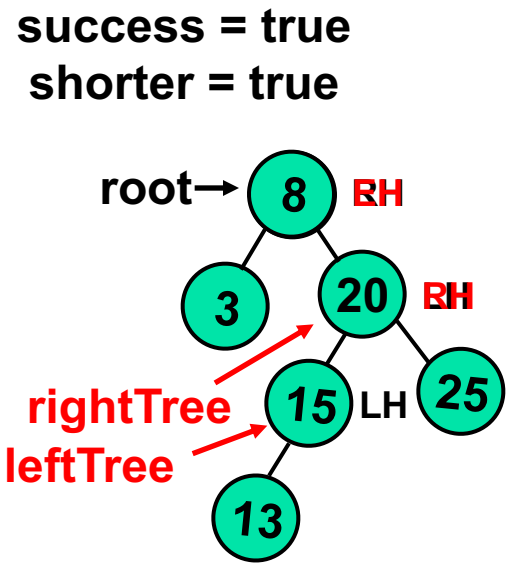
```
64                    *success = true;
65                    *shorter = true;
66                    free (dltPtr);
67                    return newRoot;          // base case
68                 } // if
69              else
70                 // Delete Node has two subtrees
71                 {
72                  exchPtr = root->left;
73                  while (exchPtr->right)
74                      exchPtr = exchPtr->right;
75                  root->dataPtr = exchPtr->dataPtr;
76                  root->left = _delete (tree,
77                        root->left, exchPtr->dataPtr,
78                        shorter,    success);
79                  if (*shorter)
80                     root = dltRightBal (root, shorter);
81                 } // else
82           } // equal node
83     return root;
84  } // _delete
```

Internal Delete Right Balance Function

```
 1  /* ================== dltRightBal ==================
 2     The tree is shorter after a delete on the left. This
 3     function adjusts the balance factors and rotates
 4     the tree to the left if necessary.
 5        Pre     tree shorter
 6        Post    Balance factors reset-balance restored
 7        Returns potentially new root
 8  */
 9  NODE* dltRightBal (NODE* root, bool* shorter)
10  {
11  // Local Definitions
12     NODE* rightTree;
13     NODE* leftTree;
14
15  // Statements
16     switch (root->bal)
17         {
18          case LH:        // Deleted Left--Now balanced
19               root->bal  = EH;
20               break;
```

Internal Delete Right Balance Function (cont.)

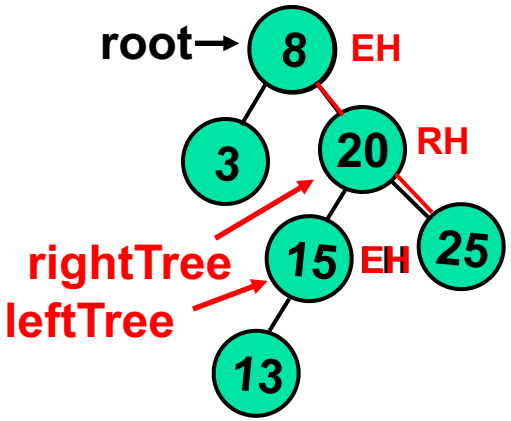
```
21
22  case EH:        // Now Right high
23       root->bal   = RH;
24       *shorter    = false;
25       break;
26
27  case RH:        // Right High - Rotate Left
28       rightTree = root->right;
29       if (rightTree->bal == LH)
30           // Double rotation required
31           {
32            leftTree  = rightTree->left;
33
34            switch (leftTree->bal)
35                {
36                 case LH: rightTree->bal = RH;
37                          root->bal      = EH;
38                          break;
```

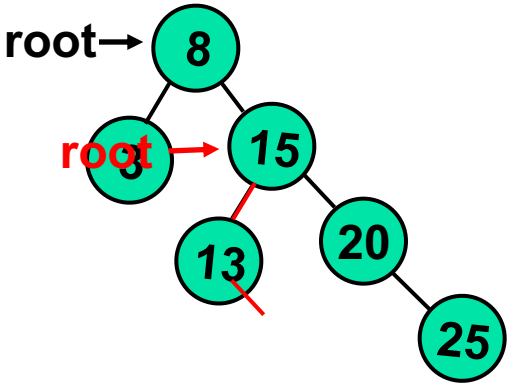


Internal Delete Right Balance Function (cont.)

```
39              case EH: root->bal      = EH;
40                       rightTree->bal = EH;
41                       break;
42              case RH: root->bal      = LH;
43                       rightTree->bal = EH;
44                       break;
45            } // switch
46
47       leftTree->bal = EH;
48
49       // Rotate Right then Left
50       root->right =
51              rotateRight (rightTree);
52       root = rotateLeft  (root);
53     } // if rightTree->bal == LH
54 else
55     {
```

success = true
shorter = true





Internal Delete Right Balance Function (cont.)

```
56                      // Single Rotation Only
57                      switch (rightTree->bal)
58                          {
59                           case LH:
60                           case RH: root->bal      = EH;
61                                    rightTree->bal = EH;
62                                    break;
63                           case EH: root->bal      = RH;
64                                    rightTree->bal = LH;
65                                    *shorter       = false;
66                                    break;
67                          } // switch rightTree->bal
68                      root = rotateLeft (root);
69                     } // else
70          } // switch
71      return root;
72  }  // dltRightBal
```

Retrieve AVL Tree Application interface

```
1  /* ================== AVL_Retrieve ================
2     Retrieve node searches tree for node containing
3     the requested key and returns pointer to its data.
4        Pre     Tree has been created (may be null)
5                data is pointer to data structure
6                     containing key to be located
7        Post    Tree searched and data pointer returned
8        Return  Address of matching node returned
9                If not found, NULL returned
10 */
11 void* AVL_Retrieve  (AVL_TREE* tree, void* keyPtr)
12 {
13 // Statements
14    if (tree->root)
15        return _retrieve (tree, keyPtr, tree->root);
16    else
17        return NULL;
18 }  // AVL_Retrieve
```

Internal Retrieve Function

```
 1  /* ==================== _retrieve ====================
 2     Searches tree for node containing requested key
 3     and returns its data to the calling function.
 4        Pre     AVL_Retrieve passes tree, keyPtr, root
 5                keyPtr is pointer to data structure
 6                   containing key to be located
 7        Post    tree searched; data pointer returned
 8        Return  Address of matching node returned
 9                     if not found, NULL returned
10  */
11  void*  _retrieve (AVL_TREE* tree,
12                    void*     keyPtr, NODE* root)
13  {
14  // Statements
15     if (root)
16         {
```

Internal Retrieve Function (cont.)

```
17          if (tree->compare(keyPtr, root->dataPtr) < 0)
18              return _retrieve(tree, keyPtr, root->left);
19          else if (tree->compare(keyPtr, root->dataPtr) > 0)
20              return _retrieve(tree, keyPtr, root->right);
21          else
22              // Found equal key
23              return root->dataPtr;
24          } // if root
25      else
26          // Data not in tree
27          return NULL;
28  } // _retrieve
```

Traverse AVL Tree Application Interface

```
1   /* ================= AVL_Traverse =================
       Process tree using inorder traversal.
2         Pre   Tree has been created (may be null)
3               process "visits" nodes during traversal
4         Post  Nodes processed in LNR (inorder) sequence
5   */
6   void AVL_Traverse (AVL_TREE* tree,
7                      void (*process) (void* dataPtr))
8   {
9
10  // Statements
11     _traversal (tree->root, process);
12     return;
13  }  // end AVL_Traverse
```

Internal Traverse Function

```
1  /* ================== _traversal ==================
2     Inorder tree traversal. To process a node, we use
3     the function passed when traversal was called.
4        Pre   Tree has been created (may be null)
5        Post  All nodes processed
6  */
7  void _traversal (NODE* root,
8                   void (*process) (void* dataPtr))
9  {
10 // Statements
11 if  (root)
12     {
13      _traversal  (root->left, process);
14      process     (root->dataPtr);
15      _traversal  (root->right, process);
16     } // if
17 return;
18 }  // _traversal
```

Empty AVL Tree Application Interface

```
 1  /* ================== AVL_Empty ==================
 2     Returns true if tree is empty; false if any data.
 3        Pre      Tree has been created. May be null
 4        Returns  True if tree empty, false if any data
 5  */
 6  bool AVL_Empty (AVL_TREE* tree)
 7  {
 8  // Statements
 9     return (tree->count == 0);
10  }  // AVL_Empty
```

Full AVL Tree Application Interface

```
 1  /* ================ AVL_Full ===============
 2     If there is no room for another node, returns true.
 3     Pre      Tree has been created
 4     Returns  True if no room for another insert,
 5              false if room.
 6  */
 7  bool AVL_Full (AVL_TREE* tree)
 8  {
 9  // Local Definitions
10     NODE* newPtr;
11
12  // Statements
13     newPtr = (NODE*)malloc(sizeof (*(tree->root)));
14     if (newPtr)
15        {
16         free (newPtr);
17         return false;
18        } // if
19     else
20          return true;
21  }  // AVL_Full
```

AVL Tree Count Application Interface

```
1  /* ============== AVL_Count =============
2     Returns number of nodes in tree.
3        Pre     Tree has been created
4        Returns tree count
5  */
6  int AVL_Count (AVL_TREE* tree)
7  {
8  // Statements
9     return (tree->count);
10 }  // AVL_Count
```

Destroy AVL Tree Application Interface

```
1  /* =============== AVL_Destroy ==============
2     Deletes all data in tree and recycles memory.
3     The nodes are deleted by calling a recursive
4     function to traverse the tree in inorder sequence.
5        Pre      tree is a pointer to a valid tree
6        Post     All data and head structure deleted
7        Return   null head pointer
8  */
9  AVL_TREE* AVL_Destroy (AVL_TREE* tree)
10 {
11 // Statements
12    if (tree)
13       _destroy (tree->root);
14
15    // All nodes deleted. Free structure
16    free (tree);
17    return NULL;
18 }  // AVL_Destroy
```

Internal Destroy Function

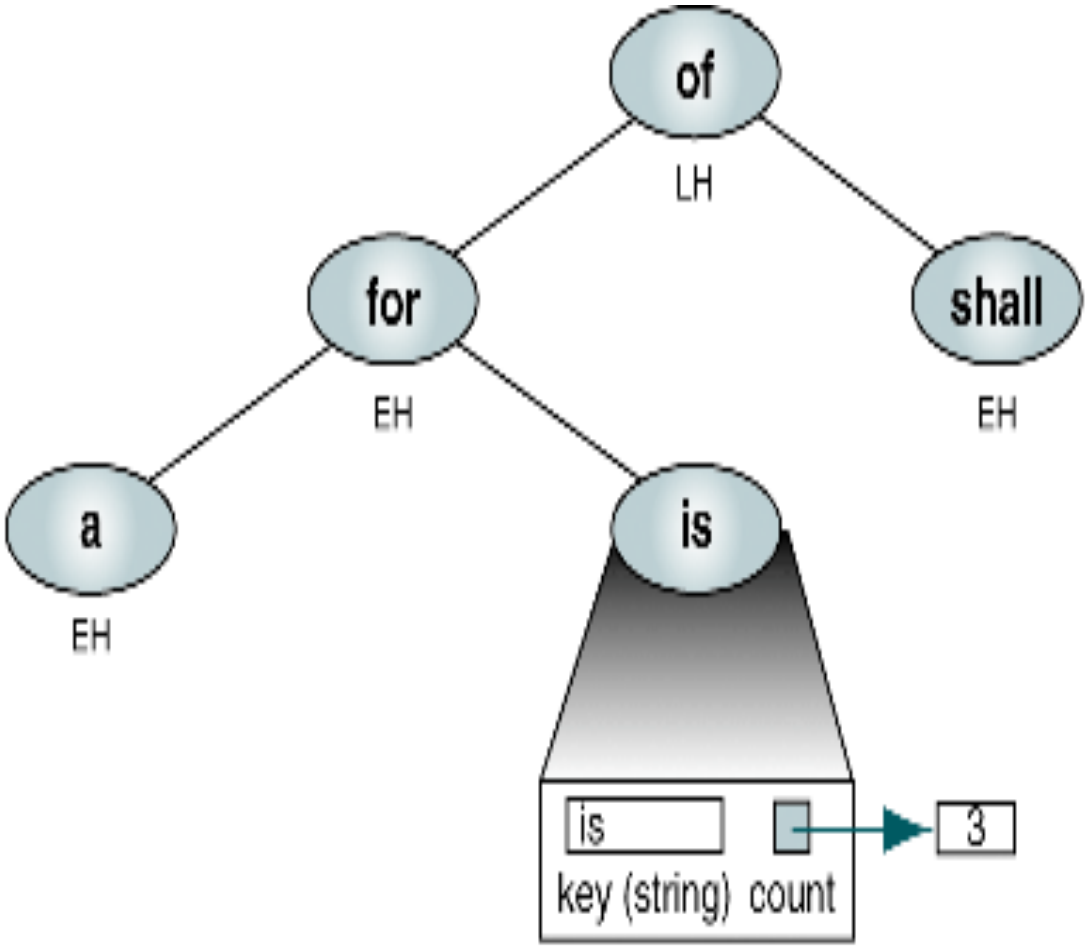
```
1  /* ================== _destroy ================
2     Deletes all data in tree and recycles memory.
3     The nodes are deleted by calling a recursive
4     function to traverse the tree in inorder sequence.
5        Pre      root is pointer to valid tree/subtree
6        Post     All data and head structure deleted
7        Return   null head pointer
8  */
9  void _destroy (NODE* root)
10 {
11 // Statements
12    if (root)
13       {
14        _destroy (root->left);
15        free (root->dataPtr);
16        _destroy (root->right);
17        free (root);
18       } // if
19    return;
20 } // _destroy
```

8-4 Application-Count Words

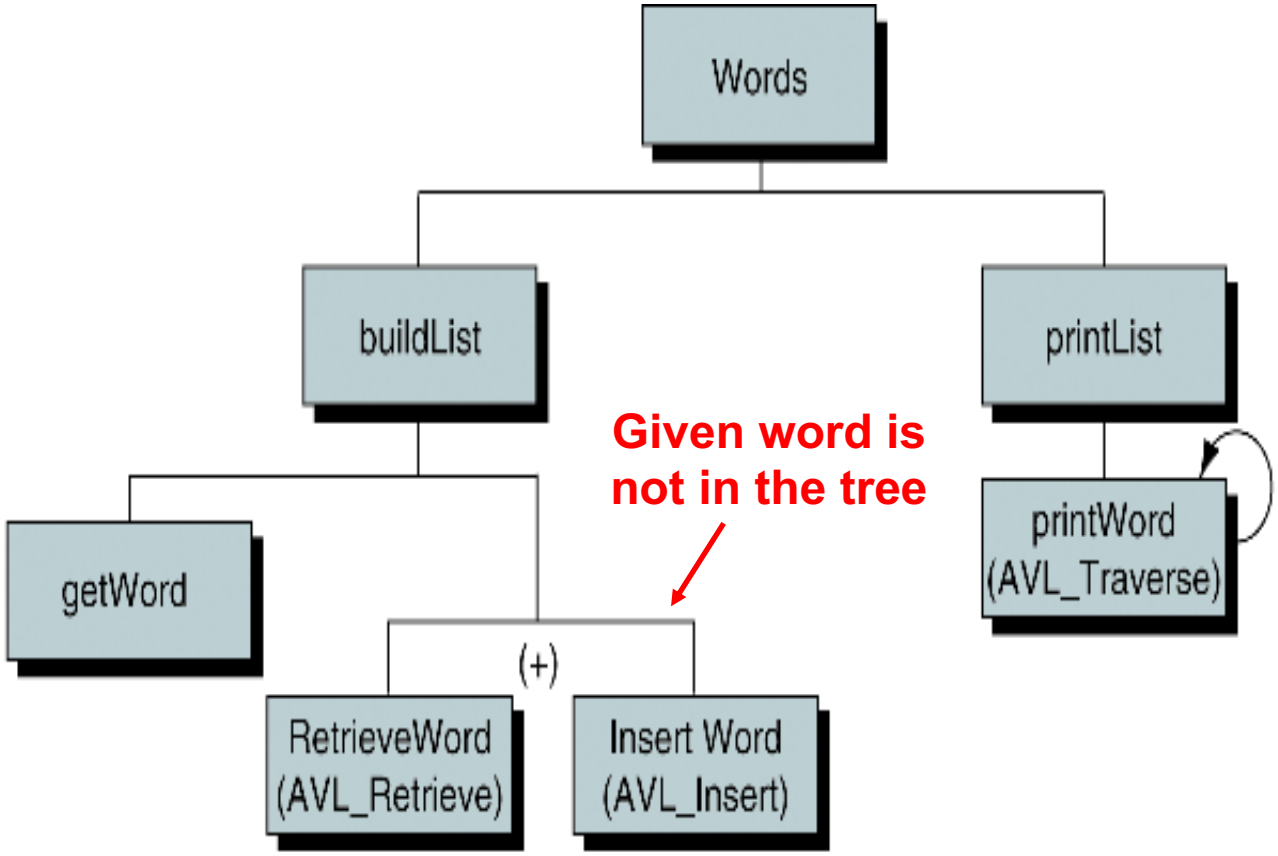
To demonstrate the use of an AVL tree, we write an application to count the words in a file. After discussing the application data structure, we write five functions, including main, to build and print the word list.

- Data Structure
- Program Design

Count Words Data Structure



Count Words Design



Count Words

```
1  /* This program counts the words in a file.
2        Written by:
3        Date:
4  */
5  #include <stdio.h>
6  #include <string.h>
7  #include <ctype.h>
8  #include <stdlib.h>
```

Count Words (cont.)

```
 9  #include <stdbool.h>
10  #include "P8AVLADT.h"
11
12  // Structures
13  typedef struct
14     {
15      char word[51];                          // One word
16      int  count;
17     } DATA;
18
19  // Prototype Declarations
20  void  buildList    (AVL_TREE*       wordList);
21  void  insertWord   (AVL_TREE*       words);
22  void  deleteWord   (AVL_TREE*       words);
23  void  printList    (AVL_TREE*       wordList);
24  void  printWord    (void* aWord);
25  bool  getWord      (DATA* aWord,    FILE* fpWords);
26  int   compareWords (void* arguPtr, void* listPtr);
```

Count Words (cont.)

```
27
28  int main (void)
29  {
30  // Local Definitions
31     AVL_TREE* wordList;
32
33  // Statements
34     printf("Start count words in document\n");
35     wordList = AVL_Create (compareWords);
36
37     buildList (wordList);
38     printList (wordList);
39
40     printf("End count words\n");
41     return 0;
42  }  // main
```

Build List

```
1  /* ================== buildList ==================
2     Reads file and creates AVL tree containing list
3     of all words used in the file with count of the
4     number of times each word is found in the file.
5        Pre   wordList has been created
6        Post  AVL tree (list) built or error returned
7  */
8  void buildList (AVL_TREE* wordList)
9  {
10 // Local Definitions
11    char  fileName[25];
12    FILE* fpWords;
13
14    bool  success;
```

Build List (cont.)

```
15      DATA* aWord;
16      DATA  newWord;
17
18  // Statements
19      printf("Enter name of file to be processed: ");
20      scanf ("%24s", fileName);
21
22      fpWords = fopen (fileName, "r");
23      if (!fpWords)
24          {
25           printf("%-s could not be opened\a\n",fileName);
26           printf("Please verify name and try again.\n");
27           exit (100);
28          } // !fpWords
29
30      while (getWord (&newWord, fpWords))
31          {
32           aWord = AVL_Retrieve(wordList, &(newWord));
33           if (aWord)
34               (aWord->count)++;
35           else
36               {
37                aWord = (DATA*) malloc (sizeof (DATA));
38                if (!aWord)
39                    {
40                     printf("Error 120 in buildList\n");
41                     exit (120);
42                    } // if
43                // Add word to list
44                *aWord         = newWord;
45                aWord->count = 1;
46                success        = AVL_Insert (wordList, aWord);
47                if (!success)
48                    {
```

Build List (cont.)

```
49                  printf("Error 121 in buildList\a\n");
50                  exit (121);
51                } // if overflow test
52            } // else
53        } // while
54
55    printf("End AVL Tree\n");
56    return;
57  } // buildList
```

Get Word

```
 1  /* ================== getWord =================
 2     Reads one word from file.
 3        Pre  nothing
 4        Post word read into reference parameter
 5  */
 6  bool getWord (DATA* aWord, FILE* fpWords)
 7  {
 8  // Local Definitions
 9     char strIn[51];
10     int  ioResult;
11     int  lastChar;
12
13  // Statements
14     ioResult = fscanf(fpWords, "%50s", strIn);
15     if (ioResult != 1)
16         return false;
```

Get Word (cont.)

```
17      // Copy and remove punctuation at end of word.
18      strcpy (aWord->word, strIn);
19      lastChar = strlen(aWord->word) - 1;
20      if (ispunct(aWord->word[lastChar]))
21         aWord->word[lastChar] = '\0';
22      return true;
23  }  // getWord
```

Compare Word Function

```
 1  /* ================== compareWords ================
 2     This function compares two integers identified
 3     by pointers to integers.
 4        Pre    arguPtr and listPtr are pointers to DATA
 5        Return -1: arguPtr value <  listPtr value
 6               -0: arguPtr value == listPtr value
 7               +1: arguPtr value >  listPtr value
 8  */
 9  int compareWords (void* arguPtr, void* listPtr)
10  {
11  // Local Declarations
12     DATA arguValue;
13     DATA listValue;
14
15  // Statements
16     arguValue = *(DATA*)arguPtr;
17     listValue = *(DATA*)listPtr;
18
19     return (strcmp(arguValue.word, listValue.word));
20  }  // compare
```

Print Word List

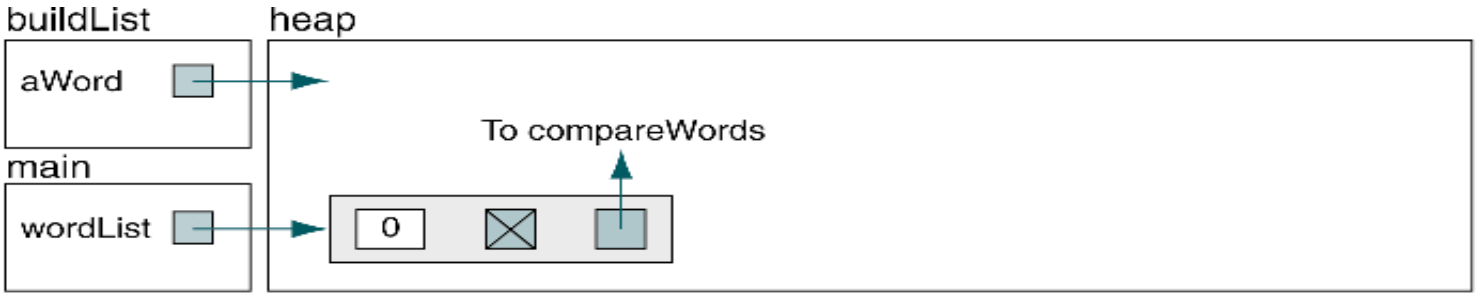
```
 1  /* ================== printList =================
 2     Prints the list with the count for each word.
 3        Pre  list has been built
 4        Post list printed
 5  */
 6  void printList (AVL_TREE* wordList)
 7  {
 8  // Statements
 9     printf("\nWords found in list\n");
10     AVL_Traverse (wordList, printWord);
11     printf("\nEnd of word list\n");
12     return;
13  }  // printList
14
15  /* ================== printWord =================
16     Prints one word from the list with its count.
17        Pre  ADT calls function to print data
18        Post data printed
19  */
20  void printWord (void* aWord)
21  {
22  // Statements
23     printf("%-25s %3d\n",
24           ((DATA*)aWord)->word, ((DATA*)aWord)->count);
25     return;
26  }  // printWord
```

```
Results:
Start count words in document
```

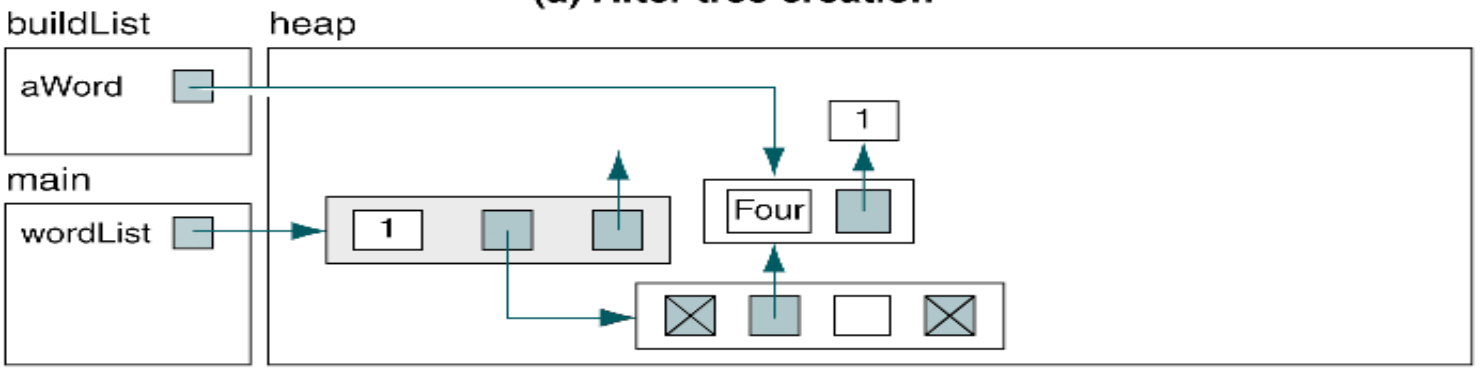
Print Word List (cont.)

```
Enter name of file to be processed: gtsybrg.txt
End AVL Tree

Words found in list
But                             1
Four                            1
God                             1
It                              3
...
years                           1

End of word list
End count words
```

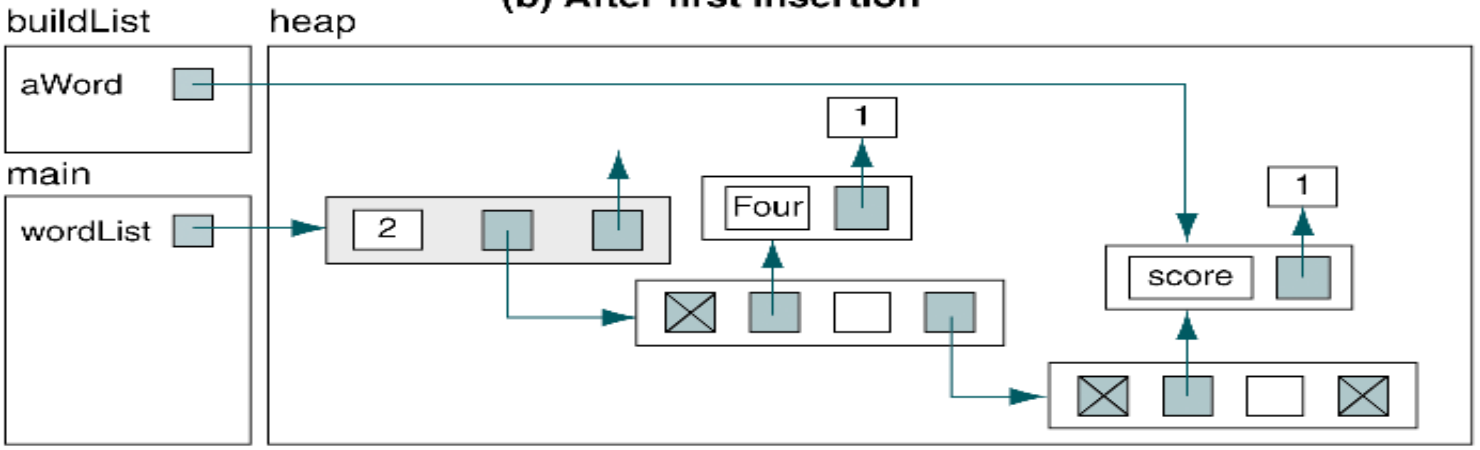
Insertion into AVL Tree



(a) After tree creation



(b) After first insertion



(c) After second insertion