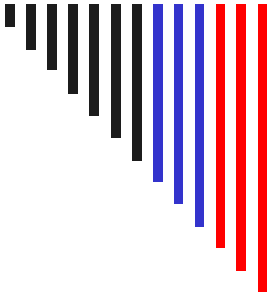
BLM212 Veri Yapıları

Binary Search Trees

2021-2022 Güz Dönemi



Binary Search Trees

Hedefler

- **İkili arama ağacını oluşturmak ve gerçekleştirmek.**
- **İkili arama ağacı ADT'nin çalışmasını anlamak.**
- **İkili arama ağacı ADT'sini kullanarak uygulamalar yazmak.**
- **BST kullanarak liste tasarlamak ve gerçekleştirmek**

BST Tanımı

İkili arama ağacı (**Binary Search Tree**), aşağıdaki özelliklere sahip bir ikili ağaçtır:

- Sol alt ağaçtaki tüm elemanlar kökten daha küçüktür.
- Sağ alt ağaçtaki tüm elemanlar kökten büyük veya köke eşittir.
- Her alt ağacın kendisi ikili bir arama ağacıdır.

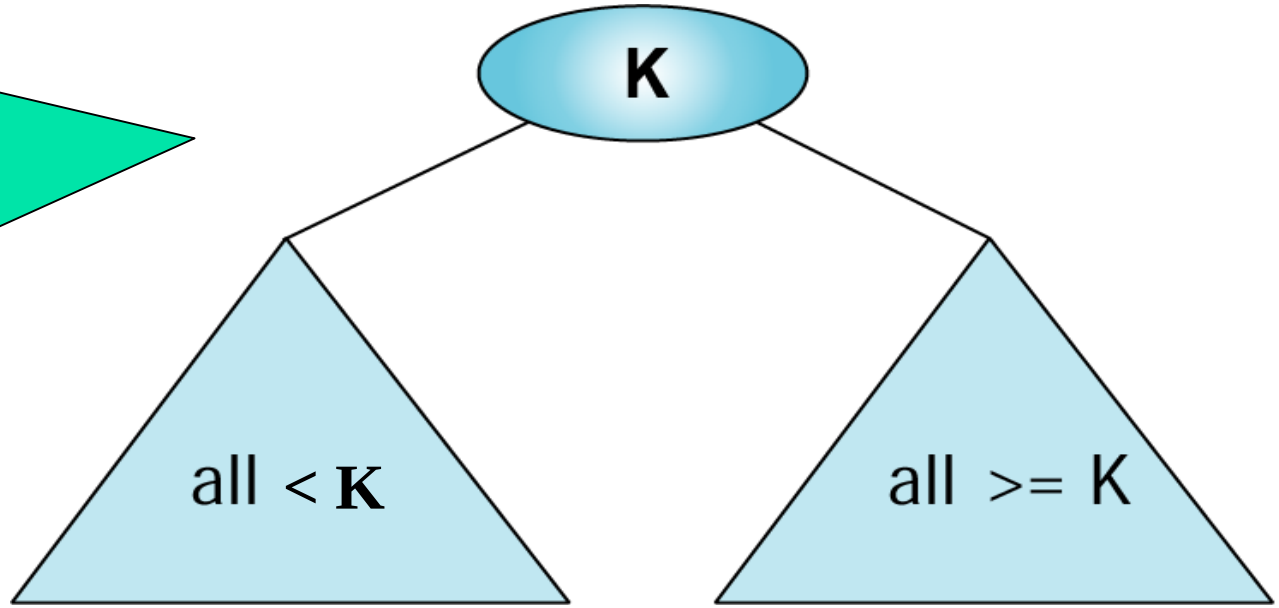
Temel Kavramlar

İkili arama ağaçları bir listede arama yapmak için ve aynı zamanda listeye veri eklemek/silmek için mükemmel bir yapı sağlar.

BST Tanımı

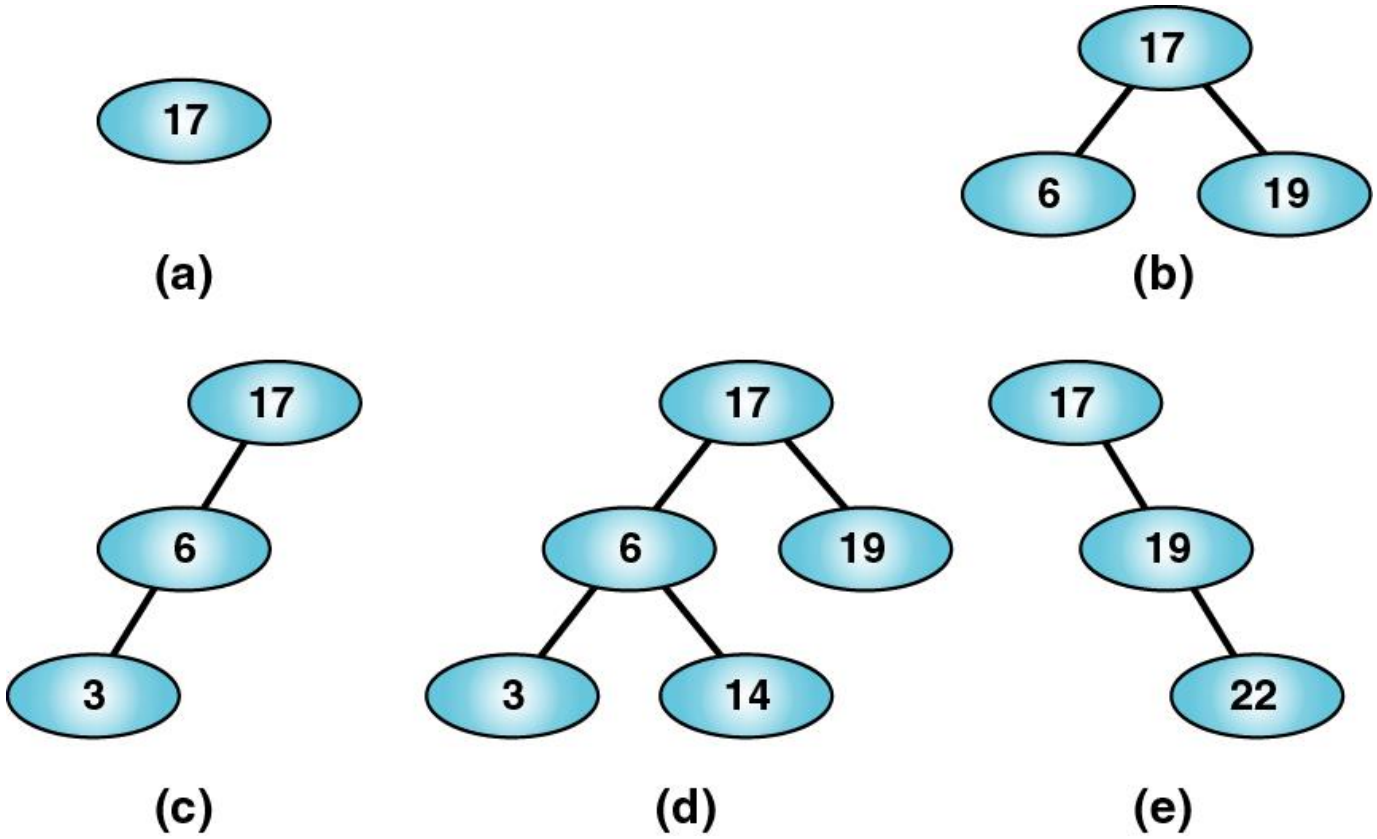
Genel olarak, her bir düğüm tarafından temsil edilen bilgi, tek bir veri elemanı yerine bir **kayıttır**. İkili arama ağacı tanımı bir kayda uygulandığında, sıralama özellikleri kaydın anahtarına atıfta bulunur.

İkili arama ağacında, **sol alt ağaç, kökünden daha küçük** anahtar değerler içerir ve **sağ alt ağaç, kökünden daha büyük veya ona eşit** anahtar değerler içerir.



Binary Search Tree

Binary Search Trees



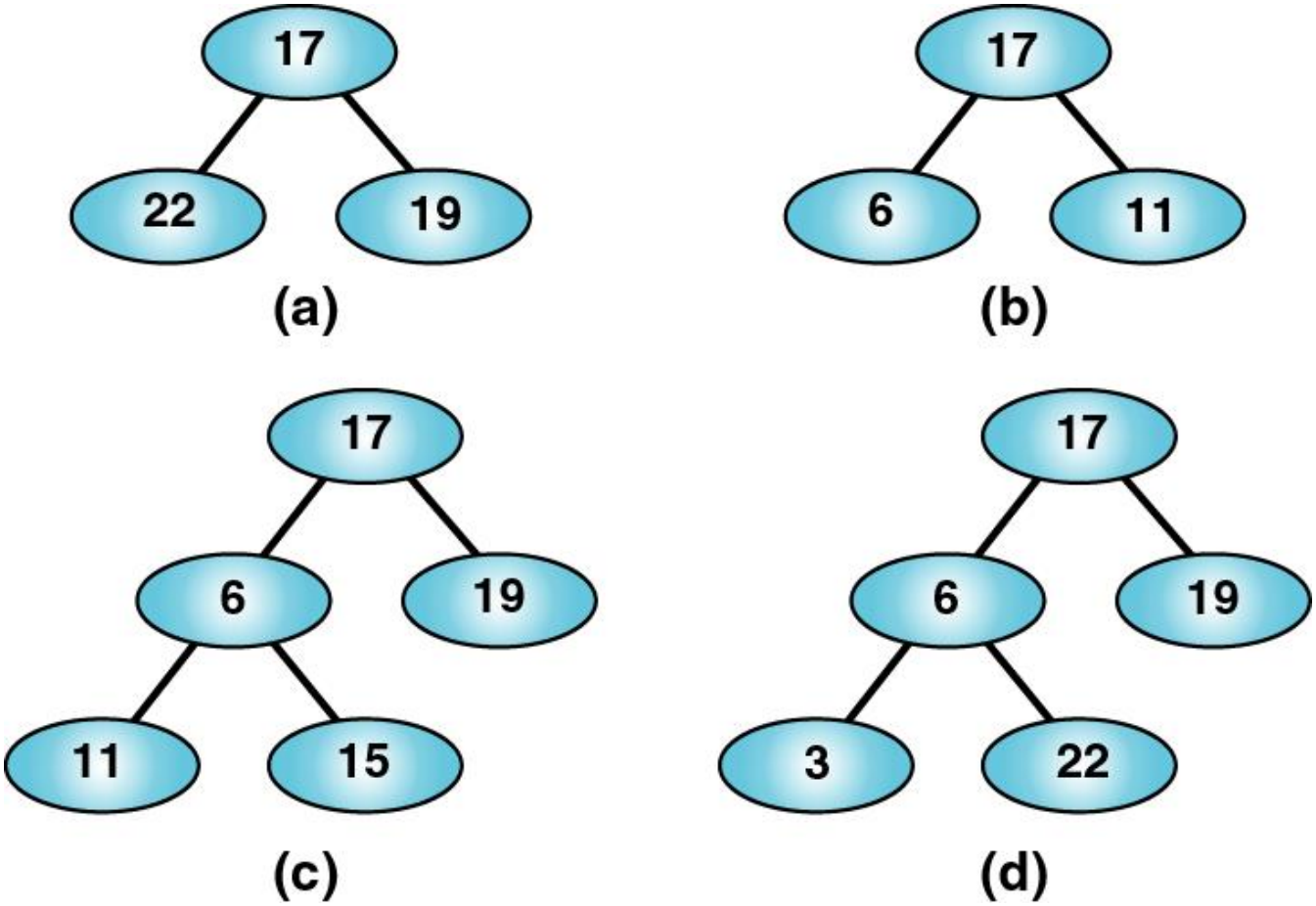
Valid Binary Search Trees

(a), (b) - complete (eksiksiz**) and balanced (**dengeli**) trees;**

(d) – nearly complete and balanced tree;

(c), (e) – neither complete nor balanced trees

Binary Search Trees



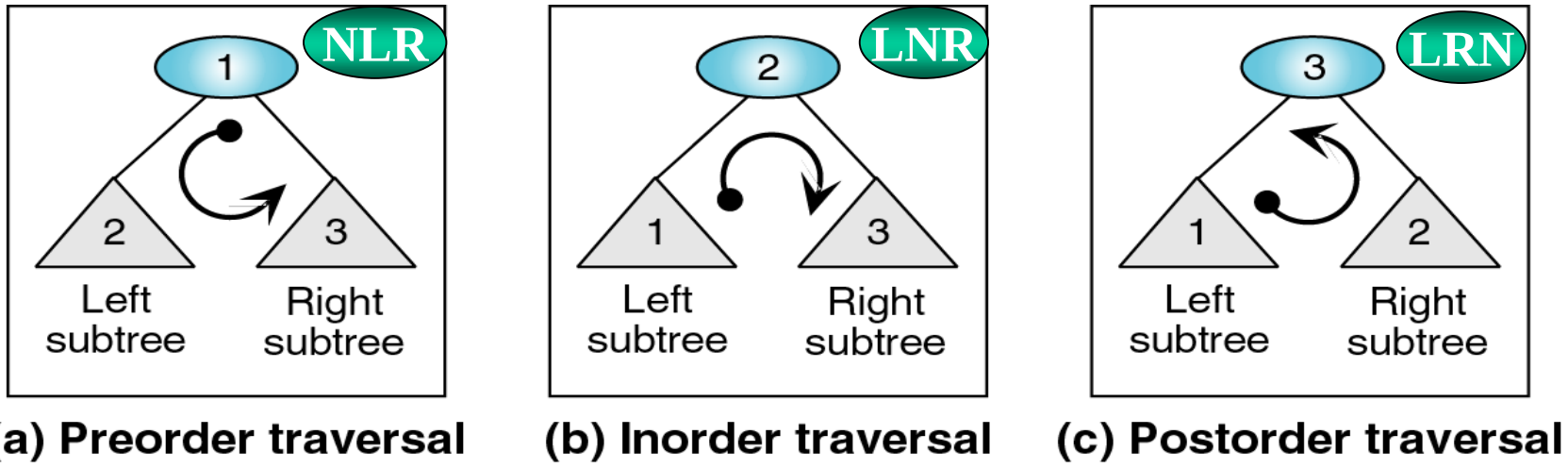
Invalid Binary Search Trees

BST Operations

Dört temel BST işleminden bahsedilir: gezinme, arama, ekleme ve silme;

- **Traversals**
- **Searches**
- **Insertion**
- **Deletion**

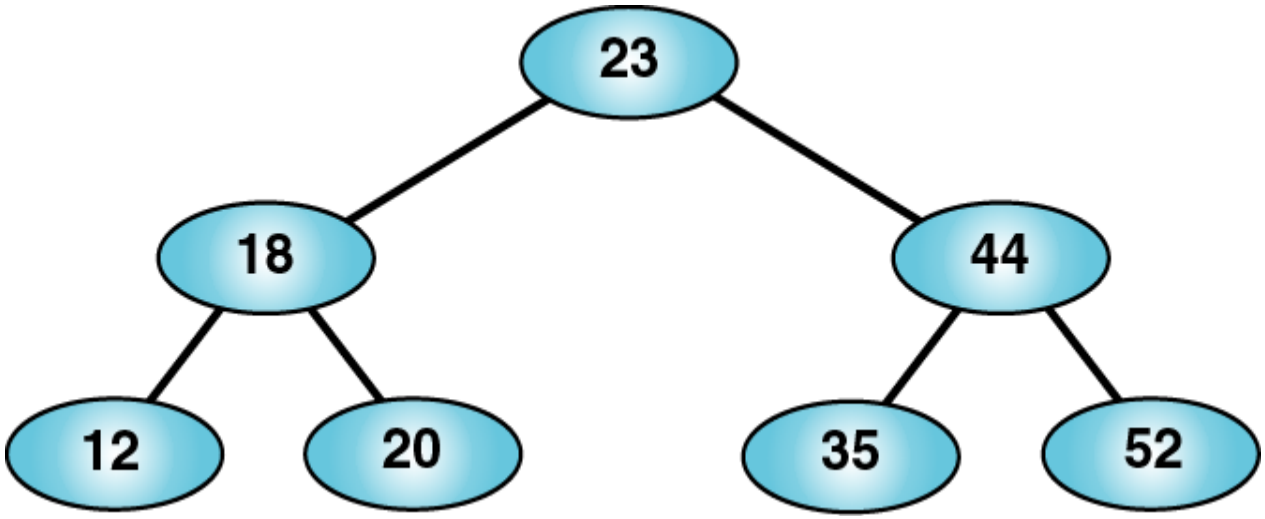
Traversal



(a) Preorder traversal **(b) Inorder traversal** **(c) Postorder traversal**

Binary Tree Traversal

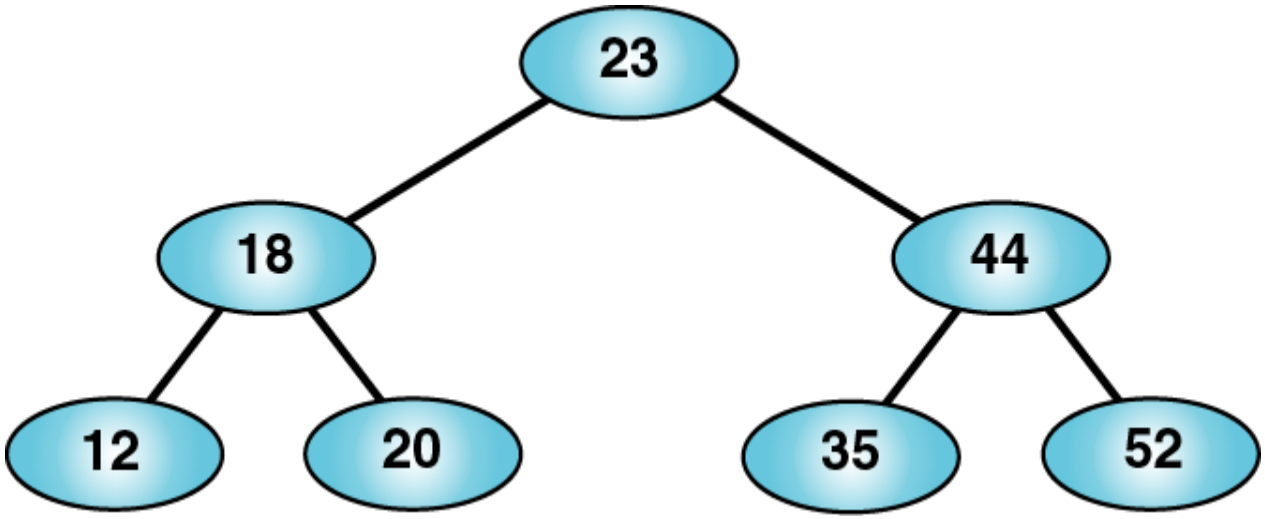
Binary Search Trees



Example of a Binary Search Tree

Preorder traversal?
Postorder traversal?
Inorder traversal?

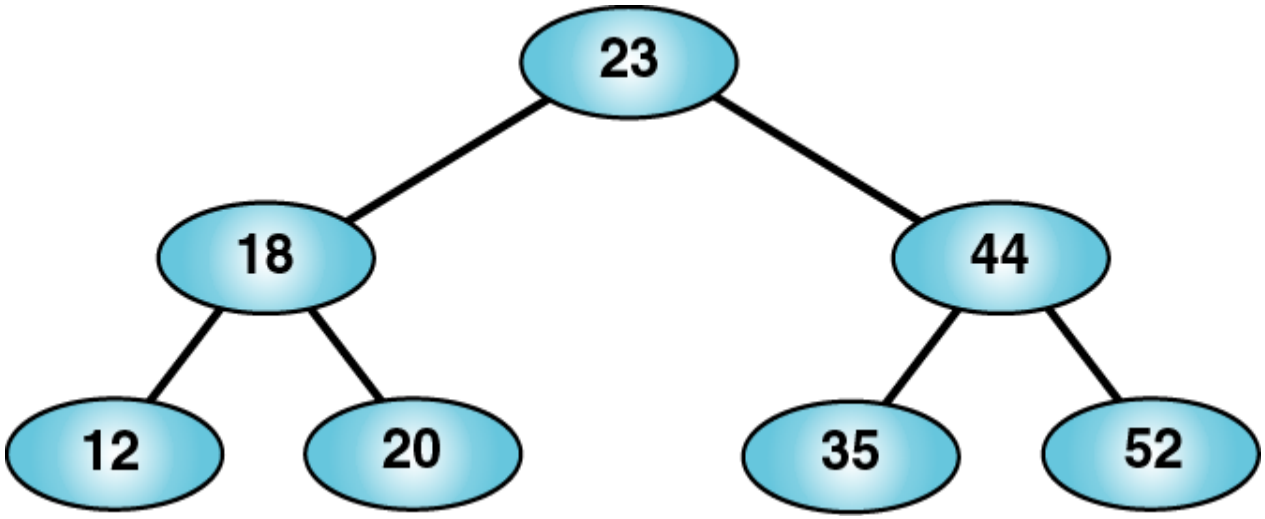
Preorder Traversal



Example of a Binary Search Tree

23 18 12 20 44 35 52

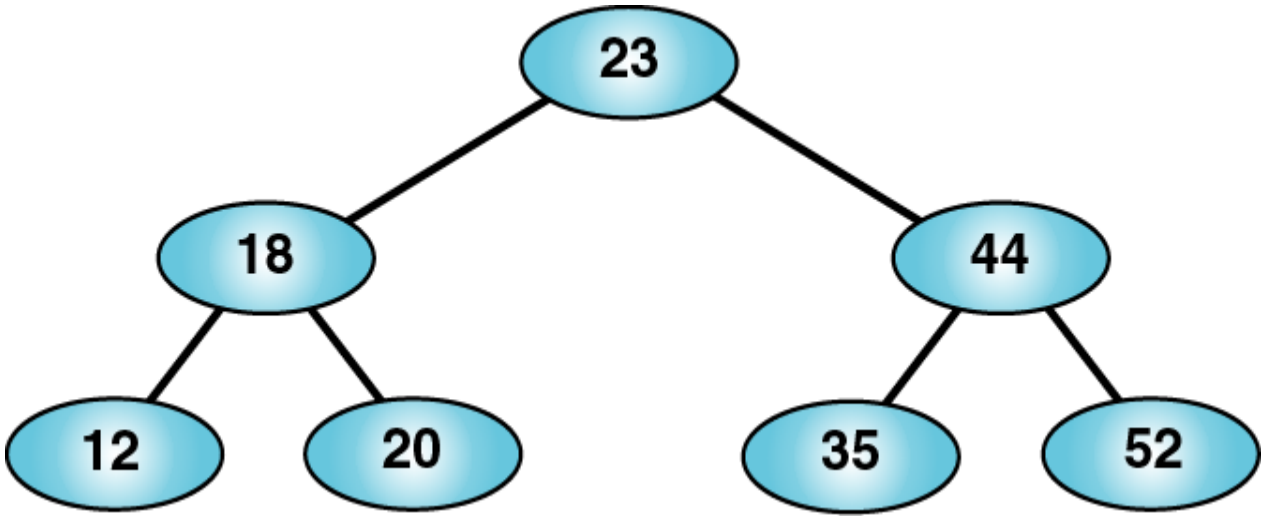
Postorder Traversal



Example of a Binary Search Tree

12 20 18 35 52 44 23

Inorder Traversal

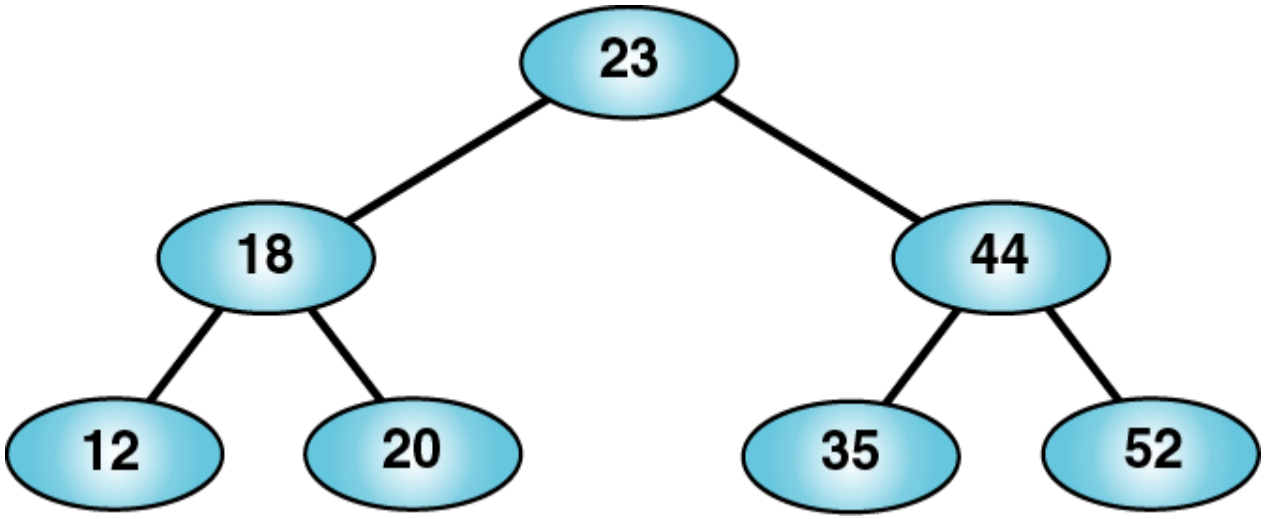


Example of a Binary Search Tree

12 18 20 23 35 44 52

İkili arama ağacını inorder gezinme/dolaşma sıralı bir liste oluşturur.

Right-Node-Left Traversal



Example of a Binary Search Tree

52 44 35 23 20 18 12

İkili arama ağacını «Right-Node-Left » gezinme azalan sıralı liste üretir.

Three BST search algorithms:

- Find the smallest node
- Find the largest node
- Find a requested node

Binary Search Trees

Find the smallest value

algorithm **findsmallestBST**(val root <pointer>)

This algorithm finds the smallest node in a BST.

PRE root is a pointer to a non-empty BST.

Return address of smallest node

1. if (root->left null)
    1. return (root)
2. return **findsmallestBST**(root->left)

end findsmallestBST

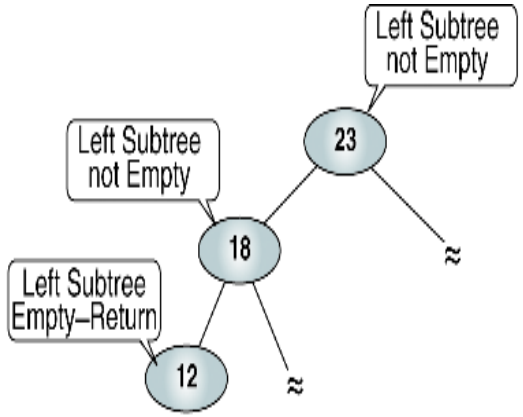
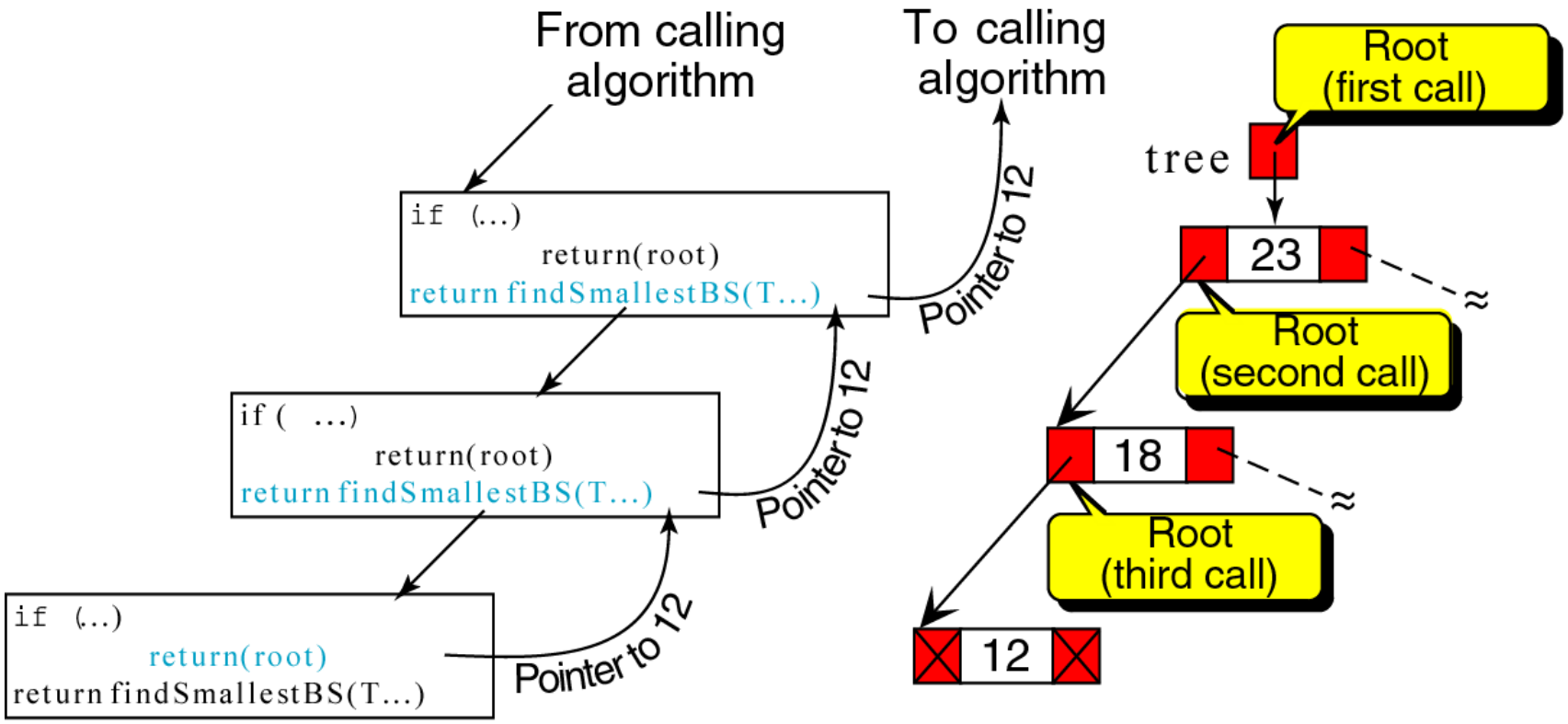


FIGURE 7-5 Find Smallest Node in a BST

Binary Search Trees



Find the smallest node in BST.

Figure 8-5

Binary Search Trees

Find the largest value

algorithm **findlargestBST**(val root <pointer>)

This algorithm finds the larges node in a BST.

PRE root is a pointer to a non-empty BST.

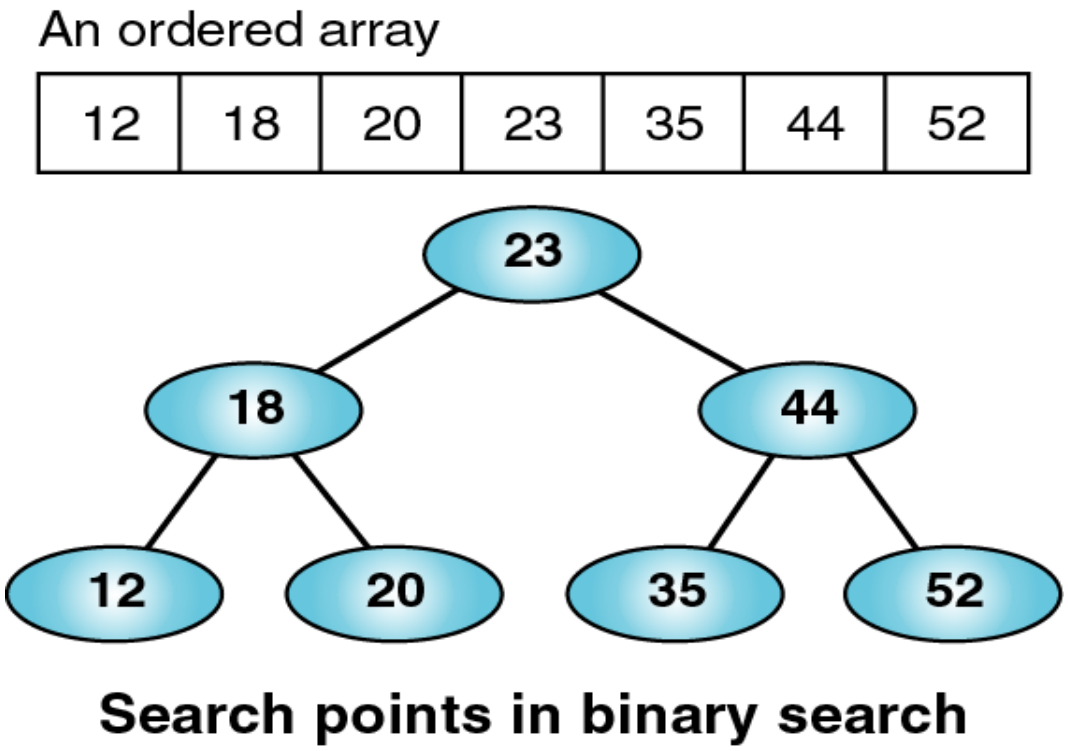
Return address of largest node

1. if (root->right null)
    1. return (root)
2. return **findlargestBST**(root->right)

end **findlargestBST**

Search a BST

Ağaçta belirli bir düğümü arama/bulma!



Binary Search Trees

Find the specific value

algorithm **searchBST**(val root <pointer>, val targetKey <key>)

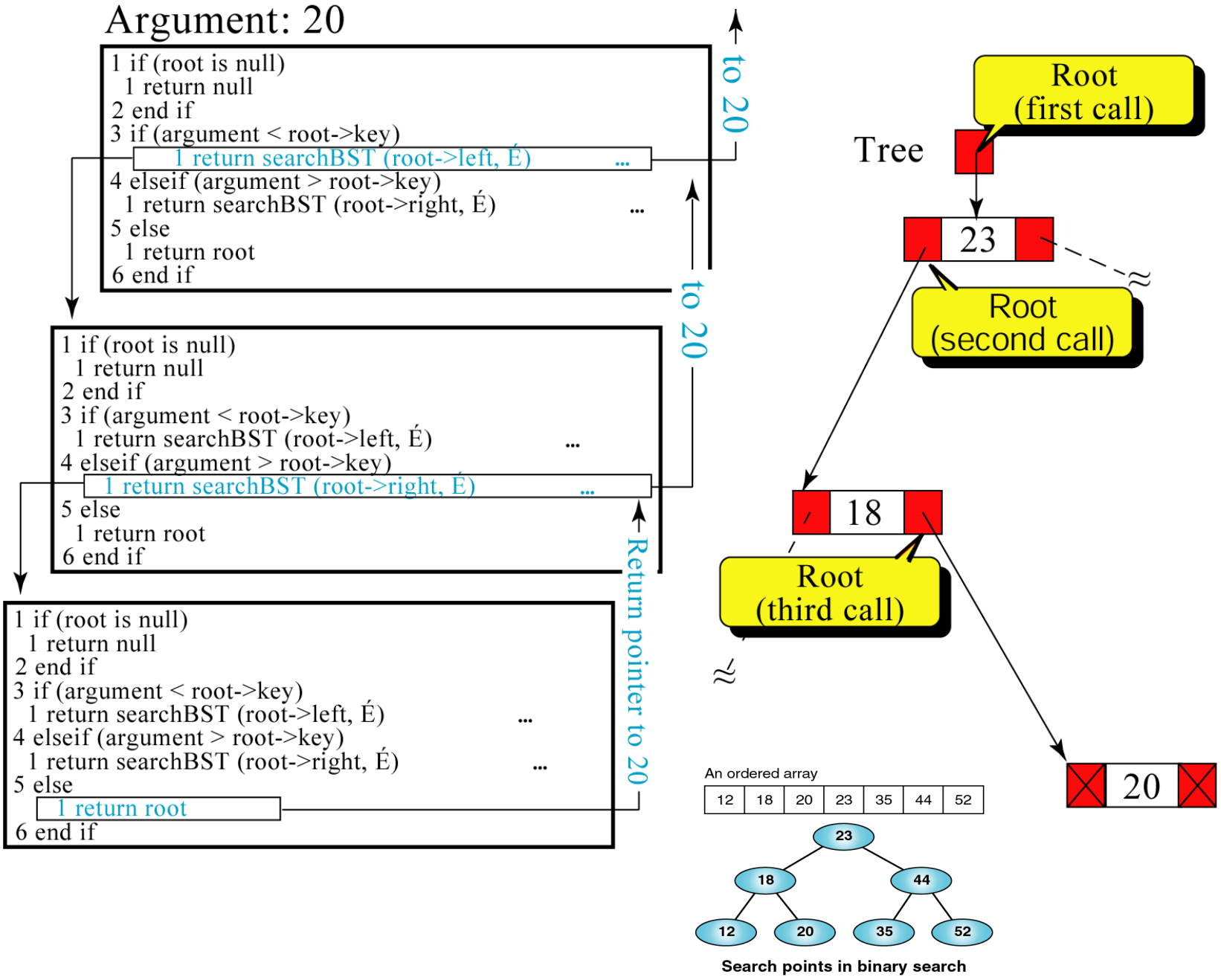
Search a binary search tree for a given value.

PRE root is the root to a binary tree or subtree, targetKey is the key value requested

Return the node address if the value is found, null if the node is not in the tree.

```
1 If (root is null)
    1 return null
2 If (targetKey < root->key)
    1 return searchBST(root->left, targetKey)
3 else If  (targetKey > root->key)
    1 return searchBST(root->right, targetKey)
4 else
    1 return root
end searchBST
```

20 numaralı düğümü arıyoruz…



BST Insertion

- Veri eklemek için tek yapmamız gereken boş bir alt ağaca (**empty subtree**) kadar dalları takip etmek ve sonra yeni düğümü eklemek.
- Başka bir deyişle, tüm eklemeler bir **yaprakta** veya **yaprak benzeri bir düğümde (leaf-like node)** gerçekleşir.
  - **leaf-like node:** sadece bir tane boş alt ağacı olan bir düğüm

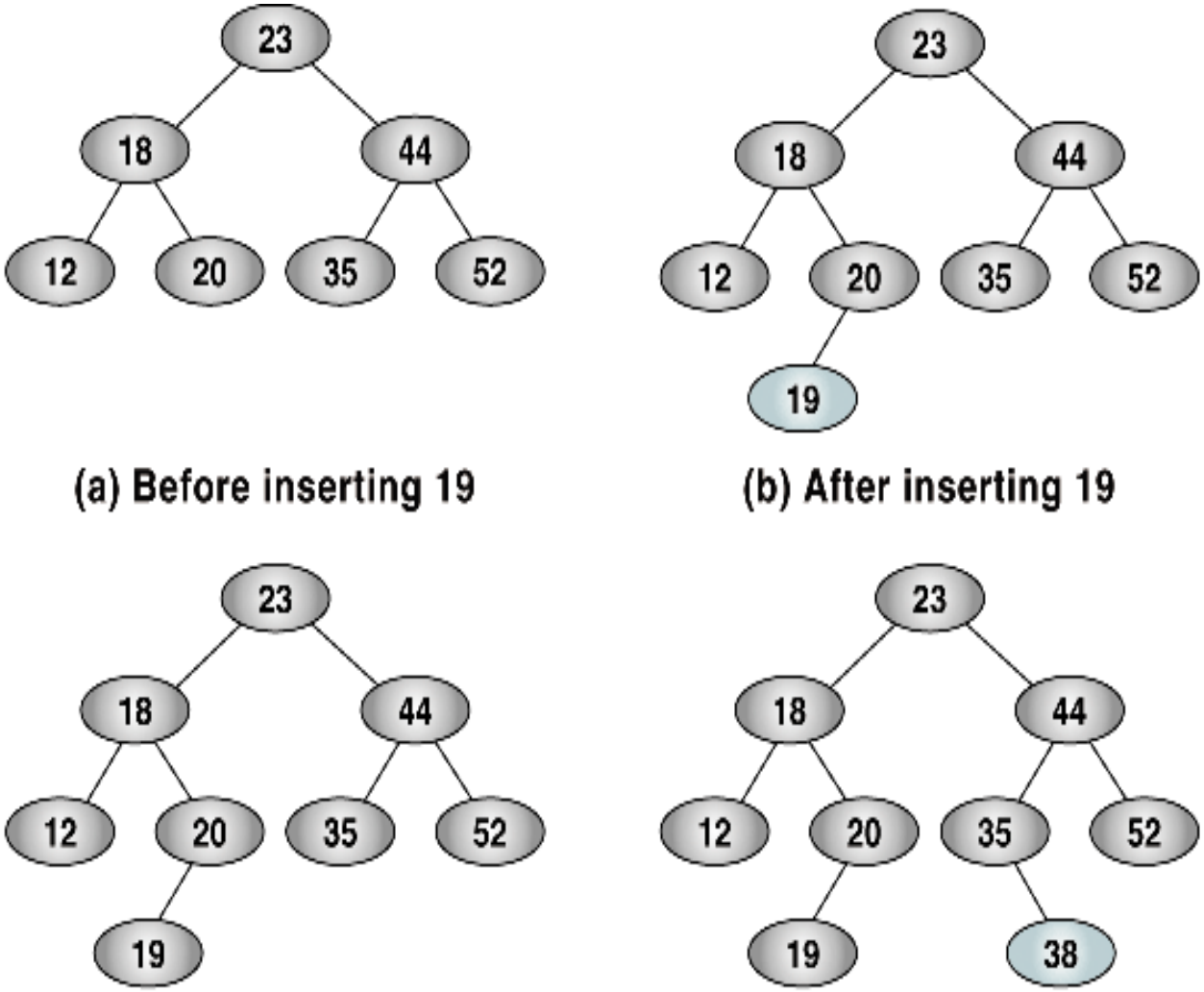
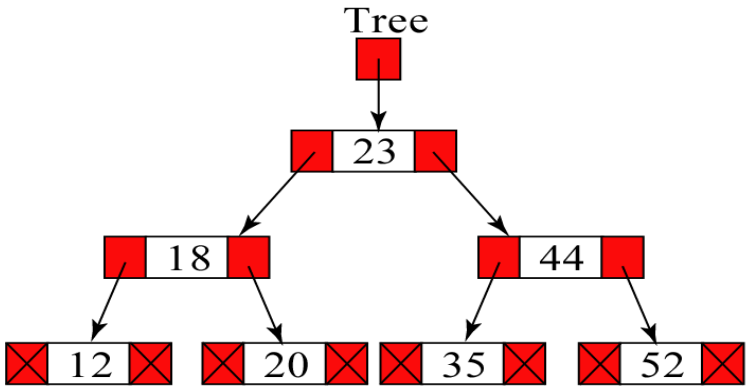
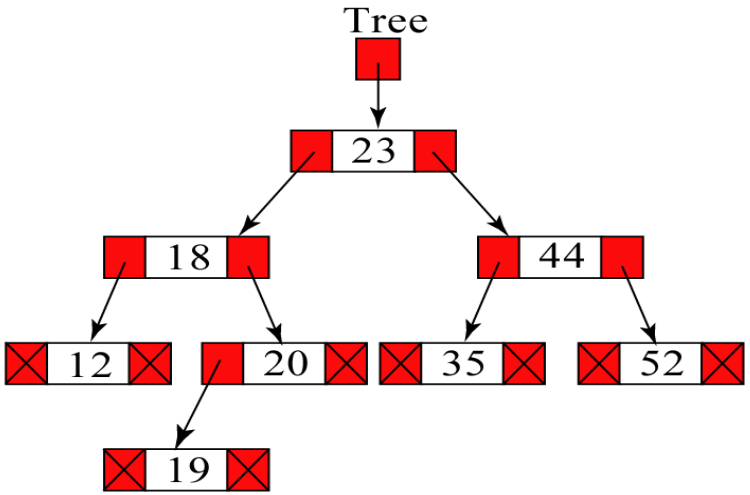


FIGURE 7-8 BST Insertion

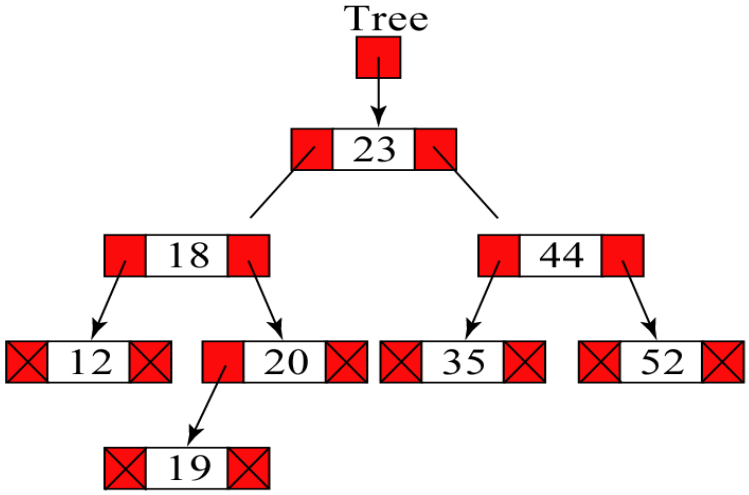
Binary Search Tree – Insert Node



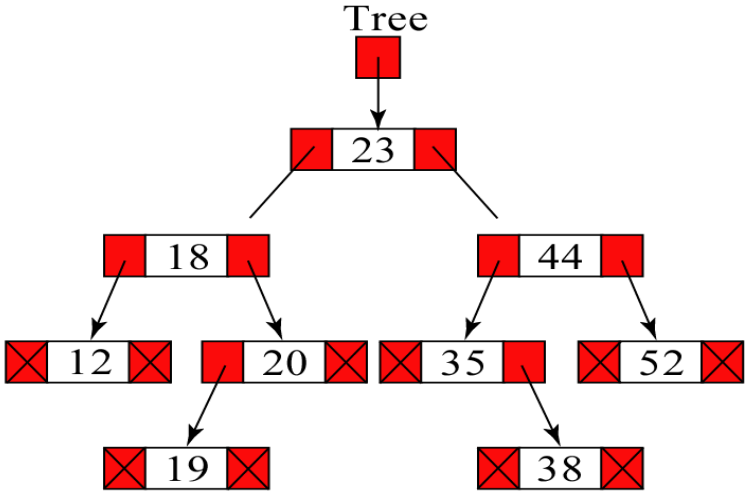
(a) Before inserting 19



(b) After inserting 19



(c) Before inserting 38



(d) After inserting 38

ALGORITHM 7-4 Add Node to BST

```
Algorithm addBST (root, newNode)
Insert node containing new data into BST using recursion.
   Pre    root is address of current node in a BST
          newNode is address of node containing data
   Post   newNode inserted into the tree
   Return address of potential new tree root
1  if (empty tree)
   1   set root to newNode
   2   return newNode
2  end if

   Locate null subtree for insertion
3  if (newNode < root)
   1   return addBST (left subtree, newNode)
4  else
   1   return addBST (right subtree, newNode)
5  end if
end addBST
```

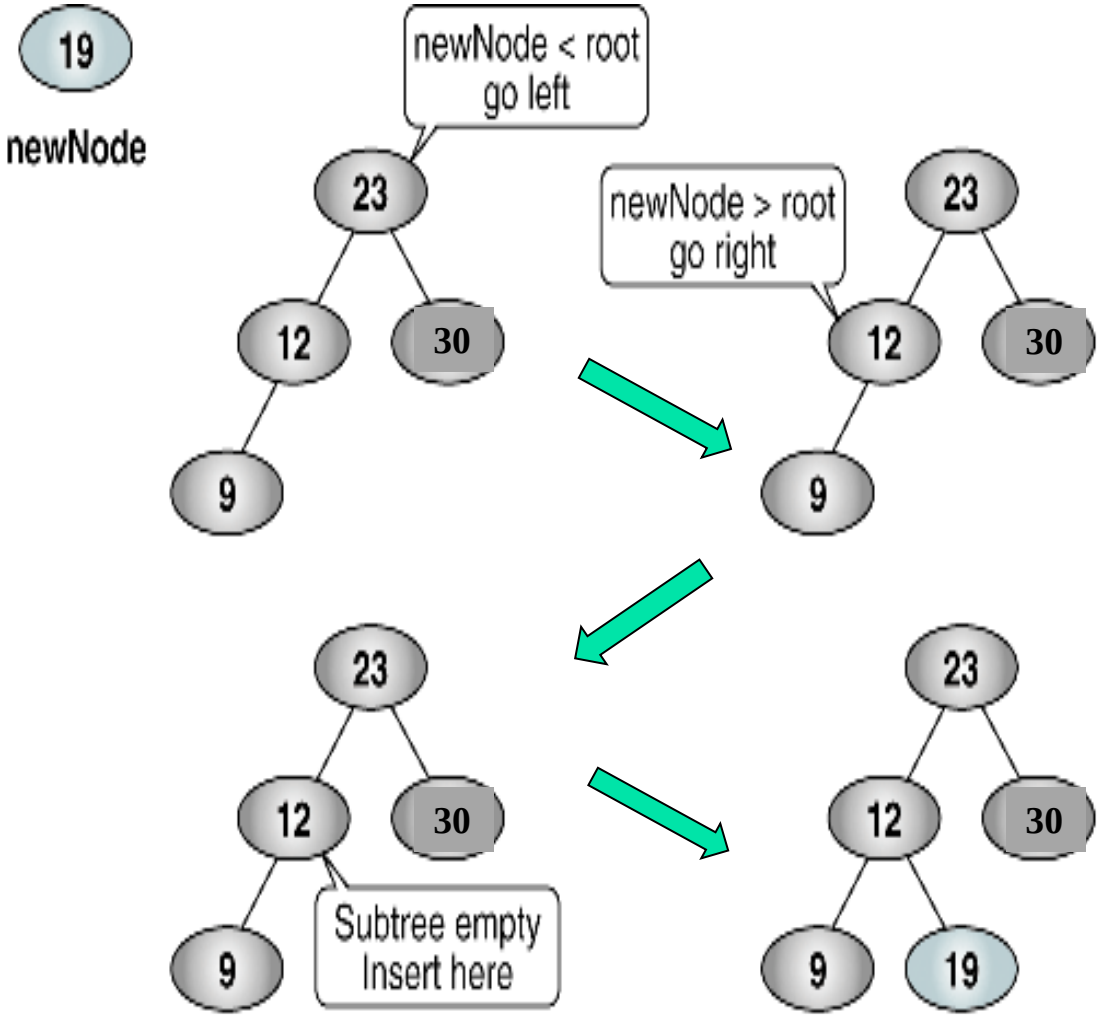


FIGURE 7-9 Trace of Recursive BST Insert

Binary Search Tree – Insert Node

Iterative Algorithm

algorithm **insertBST**(ref root <pointer>, val new <pointer>)

Insert node containing new node into BST using iteration

PRE **root** is address of the first node in a BST, **new** is address of node containing data to be inserted.

POST new node inserted into tree.

```
1 If (root is null)
      1 root = new
2 else
      1 pwalk = root
      2 loop (pwalk not null)
            1 parent =pwalk
            2 If (new->key  <  pwalk->key)
                  1 pwalk=pwalk->left
            3 else pwalk=pwalk->right
      Location for new node found
      3 If (new->key < parent->key)
            1 parent->left=new
      4 else parent->right=new
3 return
end insertBST
```

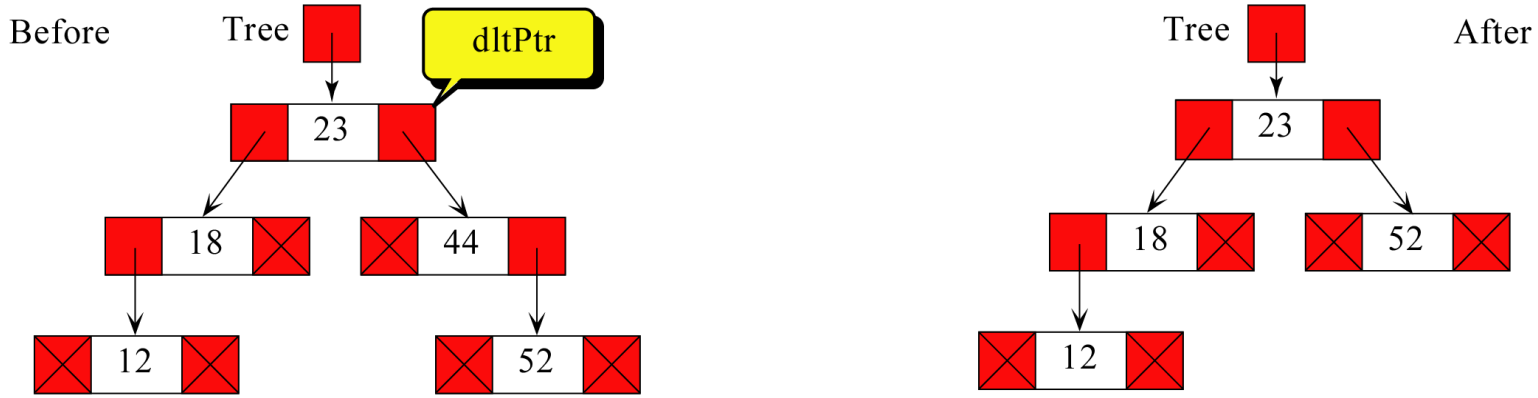
Deletion

- Bir düğümü silerken aşağıdaki muhtemel durumlar vardır:
  - Silinecek düğümün çocuğu yoktur (**no child**). Bu durumda tek yapmamız gereken düğümü silmek.
  - Silinecek düğümün **sadece bir sağ alt ağacı** vardır. Düğümü siler ve sağ alt ağacı, silinmiş düğümün ebeveynine tutttururuz/bağlarız.
  - Silinecek düğüm **sadece sol alt ağaca** sahiptir. Düğümü siler ve sol alt ağacı, silinmiş düğümün ebeveynine tutttururuz/bağlarız.
  - Silinecek düğümün **iki alt ağacı** vardır. Bir düğümü ağacın ortasından silmek mümkündür, ancak sonuç çok dengesiz ağaçlar oluşturma eğilimindedir. Çözüm, sol alt ağacın en büyük elemanını veya sağ alt ağacın en küçük elemanı köke taşımak.

Binary Search Tree – Delete Node

Silinecek düğümün çocuğu yoktur (no child). → Bu durumda tek yapmamız gereken düğümü silmek.

(a) Delete node (44) that has no children

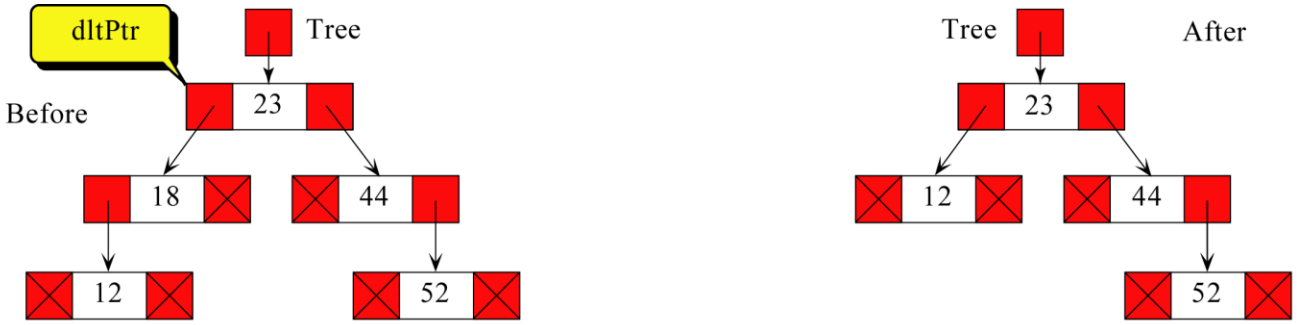


(b) Delete node (44) with no left child

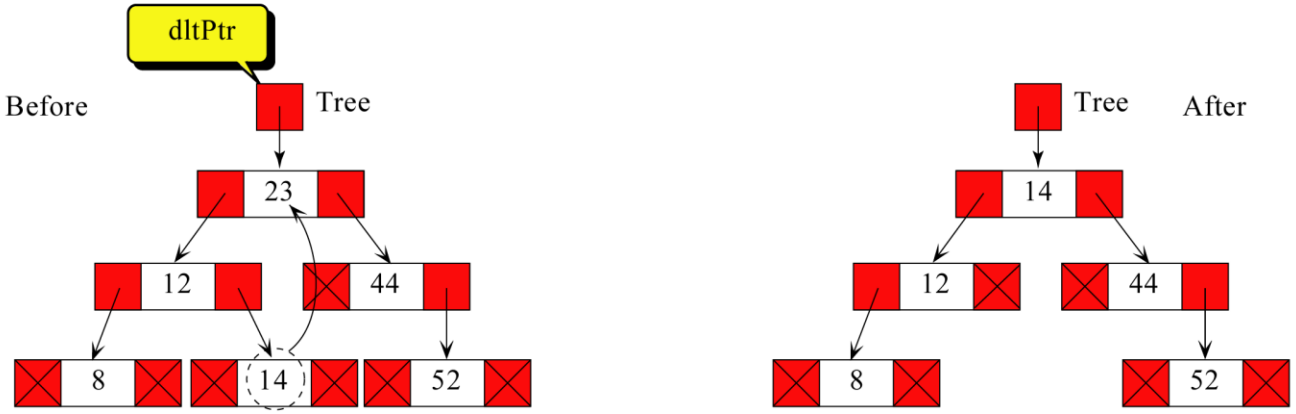
Silinecek düğümün sadece bir sağ alt ağacı vardır. → Düğümü siler ve sağ alt ağacı, silinmiş düğümün ebeveynine tuttururuz/bağlarız.

Binary Search Tree – Delete Node

Silinecek düğüm sadece sol alt ağaca sahiptir. → Düğümü siler ve sol alt ağacı, silinmiş düğümün ebeveynine tuttururuz/bağlarız.



(c) Delete node (18) with no right child



(d) Delete node 23 with two children

Silinecek düğümün iki alt ağacı vardır. → Bir düğümü ağacın ortasından silmek mümkündür, ancak sonuç çok dengesiz ağaçlar oluşturma eğilimindedir. Çözüm sol alt ağacın en büyük elemanını veya sağ alt ağacın en küçük elemanı köke taşımak.

Deletion from the middle of a tree

- Düğümü silmek yerine, silinen verilerin yerini alacak verileri bularak mevcut yapıyı mümkün olduğunca korumaya çalışırız. Bu iki yoldan biriyle yapılabilir.
  - Silinen düğümün sol alt ağacındaki en büyük düğümü bulabilir ve onun verisi silinen düğümün verisinin yerini alır.
  - Silinen düğümün sağ alt ağacındaki en küçük düğümü bulabilir ve onun verisi silinen düğümün verisinin yerini alır.

Bu işlemlerden herhangi biri, ikili arama ağacının bütünlüğünü korur.

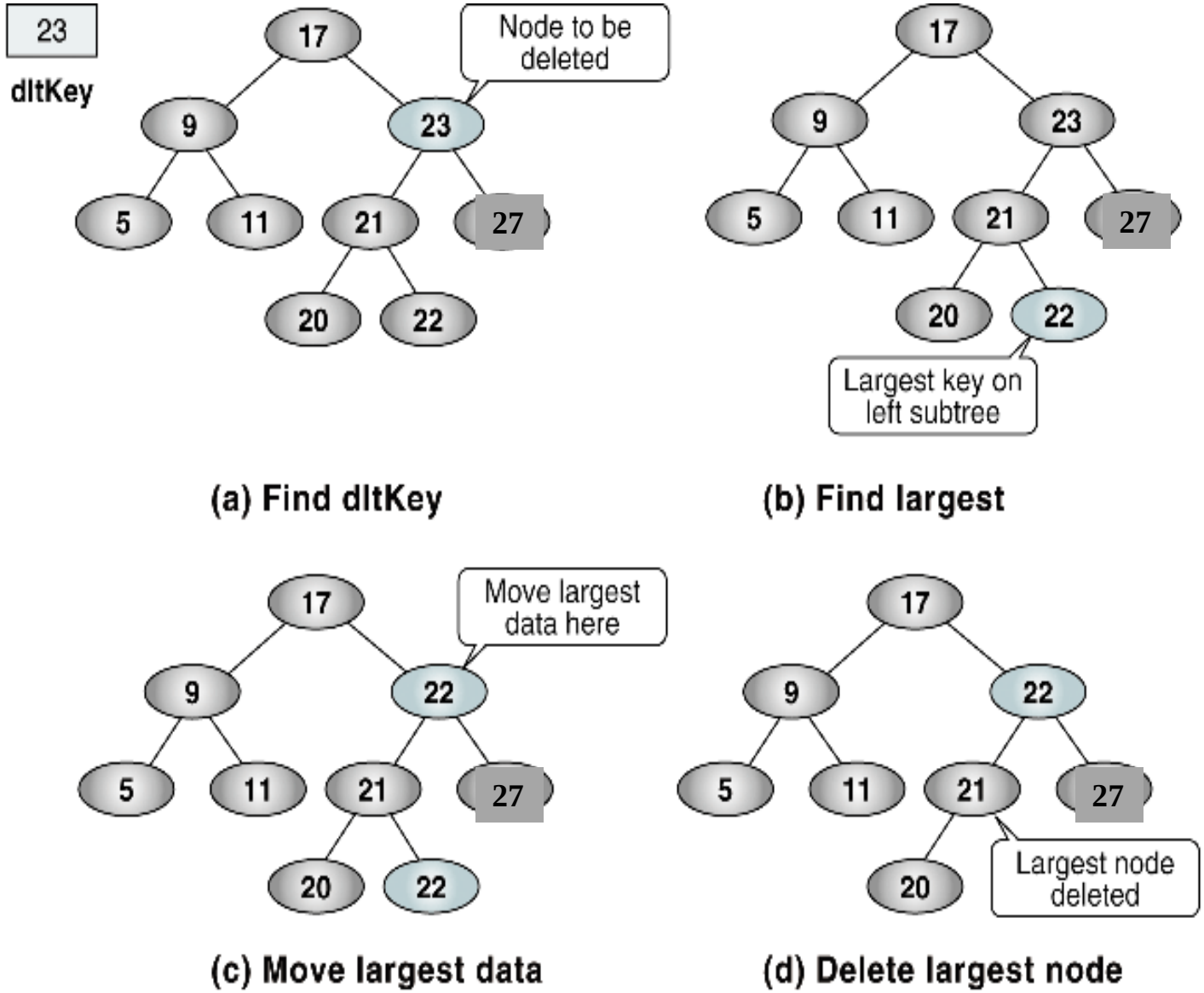


FIGURE 7-10 Delete BST Test Cases

Binary Search Tree – Delete Node

algorithm **deleteBST**(ref root <pointer>, val dltkey <key>)

This algorithm deletes a node from BST.

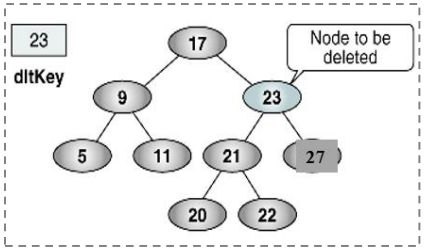
Pre root is pointer to tree containing data to be deleted,

dltkey is key of node to be deleted.

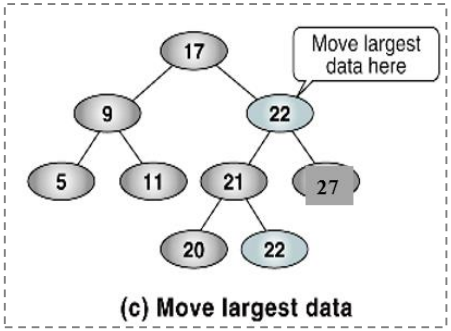
Post node deleted & memory rcycled, if dltkey not found, root unchanged.

Return true if node deleted, false if not found.

Binary Search Tree – Delete Node




```
1 If (root null)
    1 return false
2 If (dltkey < root->key)   return deleteBST(root->left, dltkey)
3 else If (dltkey > root->key) return deleteBST(root->right, dltkey)
4 else /*(Delete node found --- test for leaf node)*/
    1 If (root ->left null) /*make right subtree the root*/
          1 dltPtr=root,    root =root->right,   recycle(dltPtr),    return true
    2 else If (root->right null) /*make left subtree the root*/
          1 dltPtr=root,    root =root->left,     recycle(dltPtr),    return true
    3 else /*Node to be deleted is not a leaf, find largest node on left subtree*/
          1 dltPtr = root->left
          2 loop (dltPtr->right not null)
                1 dltprt=dltprt->right
          3 root->data =dltPtr->data
          4 return deleteBST(root->left, dltPtr->data.key)
end deleteBST
```



Binary Search Tree ADT

- Data Structure
- Algorithms

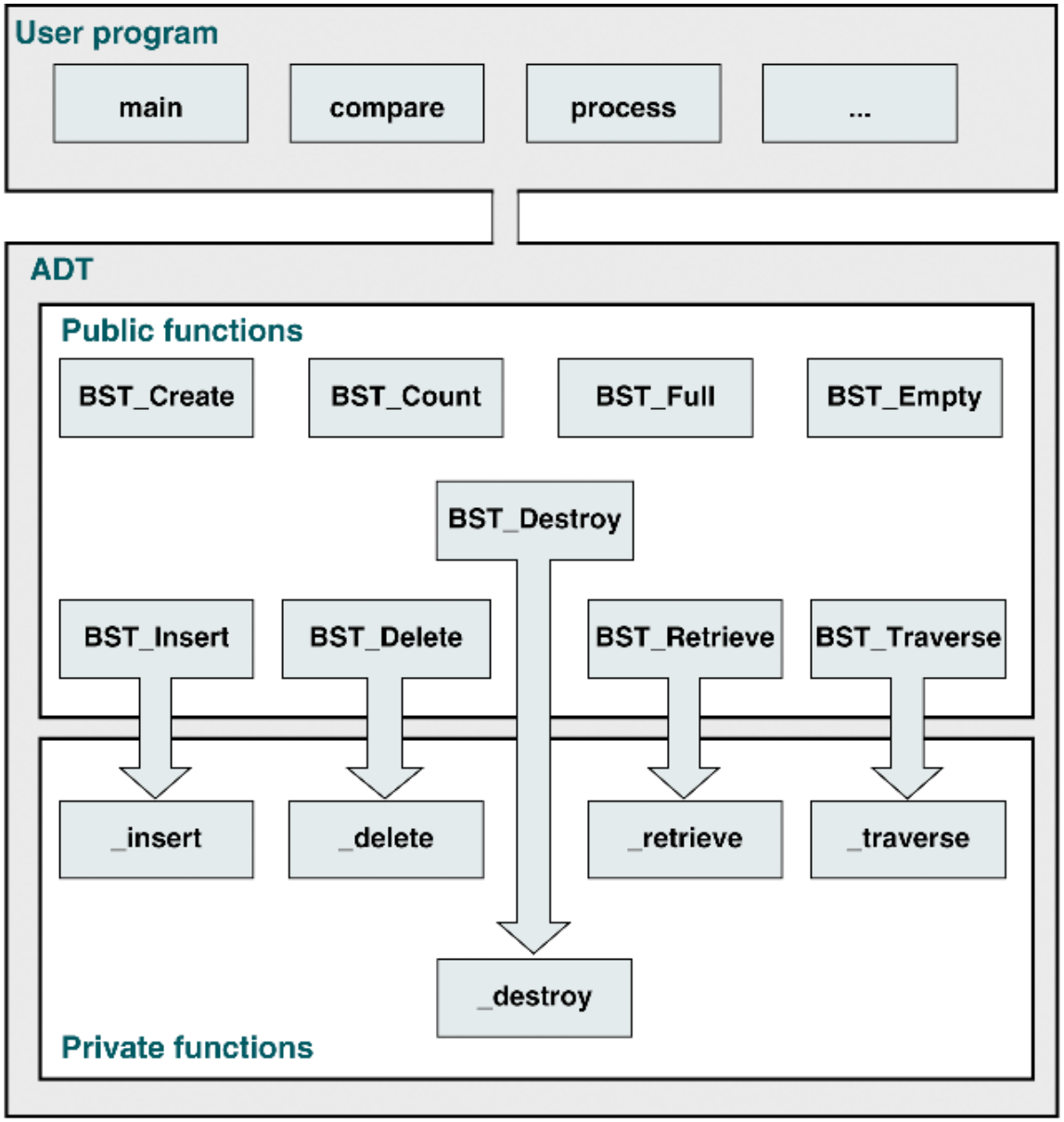
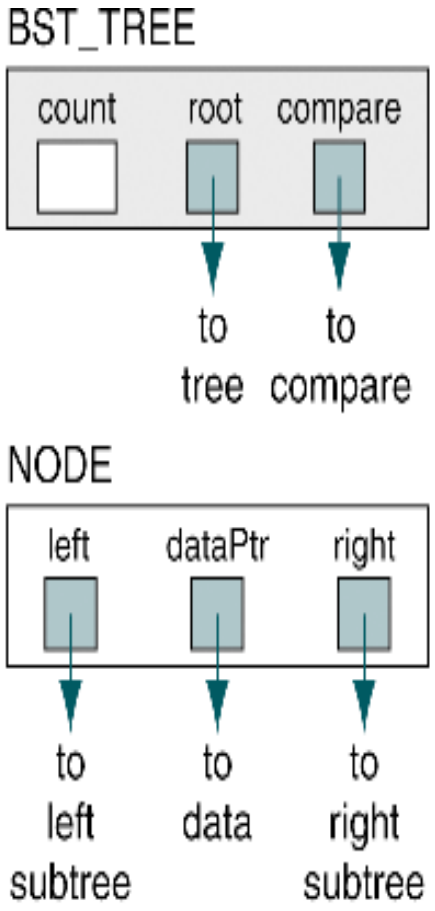


FIGURE 7-11 BST ADT Design




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

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

FIGURE 7-12 BST Tree Data Structure

PROGRAM 7-1 BST Declarations

```
/* Header file for binary search tree (BST). Contains
   structural definitions and prototypes for BST.
      Written by:
      Date:
*/

#include <stdbool.h>

// Structure Declarations
typedef struct node
   {
    void*        dataPtr;
    struct node* left;
    struct node* right;
   } NODE;

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

// Prototype Declarations
   BST_TREE* BST_Create
            (int (*compare) (void* argu1, void* argu2));
   BST_TREE* BST_Destroy (BST_TREE* tree);

   bool  BST_Insert   (BST_TREE* tree, void* dataPtr);
   bool  BST_Delete   (BST_TREE* tree, void* dltKey);
   void* BST_Retrieve (BST_TREE* tree, void* keyPtr);
   void  BST_Traverse (BST_TREE* tree,
                       void (*process)(void* dataPtr));

   bool BST_Empty (BST_TREE* tree);
   bool BST_Full  (BST_TREE* tree);
   int  BST_Count (BST_TREE* tree);

   static NODE* _insert
                  (BST TREE* tree, NODE* root,
```

. . .

Create a BST

PROGRAM 7-2 Create BST Application Interface

```
1  /* ================ BST_Create ================
2     Allocates dynamic memory for an BST 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  BST_TREE* BST_Create
10          (int  (*compare) (void* argu1, void* argu2))
11 {
12 // Local Definitions
13    BST_TREE* tree;
14
15 // Statements
16    tree = (BST_TREE*) malloc (sizeof (BST_TREE));
17    if (tree)
18       {
19        tree->root    = NULL;
20        tree->count   = 0;
21        tree->compare = compare;
22       } // if
23
24    return tree;
25 }  // BST_Create
```

Insert a BST

PROGRAM 7-3 Insert BST Application Interface

```
/* ================ BST_Insert ==================
   This function inserts new data into the tree.
      Pre    tree is pointer to BST tree structure
      Post   data inserted or memory overflow
      Return Success (true) or Overflow (false)
*/
bool BST_Insert (BST_TREE* tree, void* dataPtr)
{
// Local Definitions
   NODE* newPtr;

// Statements
   newPtr = (NODE*)malloc(sizeof(NODE));
   if (!newPtr)
      return false;

   newPtr->right   = NULL;
   newPtr->left    = NULL;
   newPtr->dataPtr = dataPtr;

   if (tree->count == 0)
       tree->root  =  newPtr;
   else
       _insert(tree, tree->root, newPtr);

   (tree->count)++;
   return true;
}  // BST_Insert
```

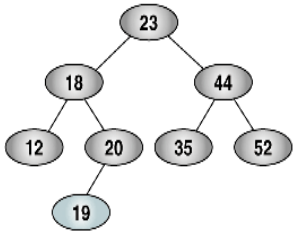
Internal Insert Function

PROGRAM 7-4 Internal Insert Function

```
1  /* ==================== _insert ====================
2     This function uses recursion to insert the new data
3     into a leaf node in the BST tree.
4        Pre    Application has called BST_Insert, which
5               passes root and data pointer
6        Post   Data have been inserted
7        Return pointer to [potentially] new root
8  */
9  NODE* _insert (BST_TREE* tree, NODE* root, NODE* newPtr)
10 {
11 // Statements
12    if (!root)
13       // if NULL tree
14       return newPtr;
15
16    // Locate null subtree for insertion
17    if (tree->compare(newPtr->dataPtr,
18                      root->dataPtr) < 0)
19       {
20        root->left = _insert(tree, root->left, newPtr);
21        return root;
22       } // new < node
23    else
24       // new data >= root data
25       {
26        root->right = _insert(tree, root->right, newPtr);
27        return root;
28         } // else new data >= root data
29    return root;
30 } // _insert
```

Bu özyinelemeli bir fonksiyon olduğu için bir temel durumu olmalıdır. Temel durumu neresidir?

Temel durum, bir yaprak bulup (14. satırdaki işlemle) newPtr'yi döndürdüğümüzde ortaya çıkar. Bu noktada ağaçtan geri dönmeye başlarız.



(b) After inserting 19

Delete a BST

PROGRAM 7-5 Delete BST Application Interface

```
/* ================== BST_Delete ==================
   This function deletes a node from the tree and
   rebalances it if necessary.
      Pre    tree initialized--null tree is OK
             dltKey is pointer to data structure
                    containing key to be deleted
      Post   node deleted and its space recycled
             -or- An error code is returned
      Return Success (true) or Not found (false)
*/
bool BST_Delete (BST_TREE* tree, void* dltKey)
{
// Local Definitions
   bool  success;
   NODE* newRoot;

// Statements
   newRoot = _delete (tree, tree->root, dltKey,
                      &success);
   if (success)
      {
       tree->root = newRoot;
       (tree->count)--;
       if (tree->count == 0)
           // Tree now empty
           tree->root = NULL;
      } // if
   return success;
}  // BST_Delete
```

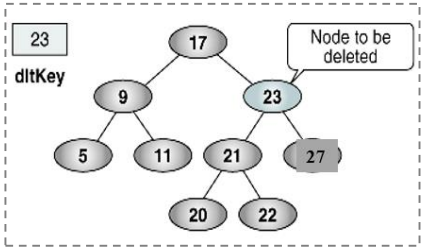
PROGRAM 7-6 Internal Delete Function

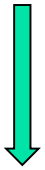
Internal Delete Function

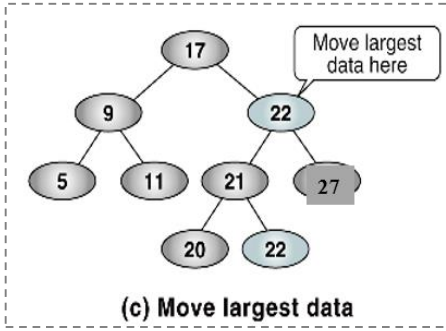
```
/* ==================== _delete ====================
   Deletes node from the tree and rebalances
   tree if necessary.
      Pre    tree initialized--null tree is OK
             dataPtr contains key of node to be deleted
      Post   node is deleted and its space recycled
             -or- if key not found, tree is unchanged
            success is true if deleted; false if not
     Return pointer to root
*/
NODE*  _delete (BST_TREE* tree,    NODE* root,
                void*     dataPtr, bool* success)
{
// Local Definitions
   NODE* dltPtr;
   NODE* exchPtr;
   NODE* newRoot;
   void* holdPtr;

// Statements
   if (!root)
      {
       *success = false;
       return NULL;
      } // if

   if (tree->compare(dataPtr, root->dataPtr) < 0)
        root->left  = _delete (tree,    root->left,
                               dataPtr, success);
   else if (tree->compare(dataPtr, root->dataPtr) > 0)
        root->right = _delete (tree,    root->right,
                               dataPtr, success);
   else
       // Delete node found--test for leaf node
       {
```

Internal Delete Function

```
33      else
34          // Delete node found--test for leaf node
35          {
36           dltPtr = root;
37           if (!root->left)
38               // No left subtree
39               {
40                free (root->dataPtr);       // data memory
41                newRoot = root->right;
42                free (dltPtr);              // BST Node
43                *success = true;
44                return newRoot;             // base case
45               } // if true
46           else
47               if (!root->right)
48                   // Only left subtree
49                   {
50                    newRoot = root->left;
51                    free (dltPtr);
52                    *success = true;
53                    return newRoot;         // base case
54                  } // if
55               else
56                   // Delete Node has two subtrees
57                   {
58                    exchPtr = root->left;
59                    // Find largest node on left subtree
60                    while (exchPtr->right)
61                        exchPtr = exchPtr->right;
62
63                    // Exchange Data
64                    holdPtr          = root->dataPtr;
65                    root->dataPtr    = exchPtr->dataPtr;
66                    exchPtr->dataPtr = holdPtr;
67                    root->left       =
68                       _delete (tree,   root->left,
69                                exchPtr->dataPtr, success);
70                   } // else
71          } // node found
72      return root;
73  }  // _delete
```







(c) Move largest data

Retrieve a BST

Alma (Retrieve) fonksiyonu, istenen düğüm bulunana kadar ağacın sol-sağ yapısını takip eder. Bulunduğunda, verinin adresi çağıran fonksiyona döndürülür. Veri bulunmazsa, boş bir işaretçi döndürülür.

PROGRAM 7-7 Retrieve BST Application Interface

```
/* ================== BST_Retrieve ==================
   Retrieve node searches tree for the node containing
   the requested key and returns pointer to its data.
      Pre     Tree has been created (may be null)
              data is pointer to data structure
                   containing key to be located
      Post    Tree searched and data pointer returned
      Return  Address of matching node returned
              If not found, NULL returned
*/
void* BST_Retrieve  (BST_TREE* tree, void* keyPtr)
{
// Statements
   if (tree->root)
       return _retrieve (tree, keyPtr, tree->root);
   else
       return NULL;
}  // BST_Retrieve
```

Internal Retrieve Function

PROGRAM 7-8 Internal Retrieve Function

```
1  /* ==================== _retrieve ====================
2     Searches tree for node containing requested key
3     and returns its data to the calling function.
4        Pre     _retrieve passes tree, dataPtr, root
5                dataPtr is pointer to data structure
6                   containing key to be located
7        Post    tree searched; data pointer returned
8        Return  Address of data in matching node
9                If not found, NULL returned
10 */
11 void* _retrieve (BST_TREE* tree,
12                   void* dataPtr, NODE* root)
13 {
14 // Statements
15    if (root)
16          {
17           if (tree->compare(dataPtr, root->dataPtr) < 0)
18              return _retrieve(tree, dataPtr, root->left);
19          else if (tree->compare(dataPtr,
20                   root->dataPtr) > 0)
21             return _retrieve(tree, dataPtr, root->right);
22           else
23               // Found equal key
24               return root->dataPtr;
25          }  // if root
26    else
27          // Data not in tree
28          return NULL;
29 }  // _retrieve
```

Traverse a BST

- Bu gezinme (**Travese**) fonksiyonu, **inorder** gezinmeyi kullanır ve düğümü işlemek için «**process**» isimli uygulamaya bağlı bir fonksiyonu çağırır.
- Gezinme işi standarttır fakat uygulama tarafındaki düğüm işleme fonksiyonu standart değildir.
  - **Bu nedenle uygulama, gezinme fonksiyonunu her çağırdığında ayrıca veriyi işleyecek fonksiyonun adresini de bu fonksiyona iletmelidir.**
- İşleme fonksiyonu sadece bir parametre kullanır: **işlenecek düğümün adresi**

PROGRAM 7-9 Traverse BST Application Interface

```
1  /* ================== BST_Traverse ==================
2     Process tree using inorder traversal.
3        Pre   Tree has been created (may be null)
4              process "visits" nodes during traversal
5        Post  Nodes processed in LNR (inorder) sequence
6  */
7  void BST_Traverse (BST_TREE* tree,
8                     void (*process) (void* dataPtr))
9  {
```

```
10 // Statements
11    _traverse (tree->root, process);
12    return;
13 }  // end BST_Traverse
```

Internal Traverse Function

PROGRAM 7-10 Internal Traverse Function

```
1  /* ================== _traverse ==================
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 _traverse (NODE* root,
8                  void (*process) (void* dataPtr))
9  {
10 // Statements
11 if  (root)
12     {
13      _traverse (root->left, process);
14      process   (root->dataPtr);
15      _traverse (root->right, process);
16     } // if
17 return;
18 }  // _traverse
```

```
229 /* ================== processStu ====================
230    Print one student's data.
231        Pre  stu is a pointer to a student
232        Post data printed and line advanced
233 */
234 void processStu (void* stuPtr)
235 {
236 // Local Definitions
237    STUDENT aStu;
238
239 // Statements
240    aStu = *(STUDENT*)stuPtr;
241    printf("%04d  %-40s %4.1f\n",
242           aStu.id, aStu.name, aStu.gpa);
243    return;
244 }  // processStu
```

Destroy a BST

Silinmesi gereken tüm verileri ve düğümleri bulmak için ağacı gezinmemiz gerektiğinden, fiziksel silmeleri yapmak için özyinelemeli bir fonksiyon çağırırız.

PROGRAM 7-14 Destroy BST Application Interface

```
1  /* ============== BST_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  BST_TREE* BST_Destroy (BST_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 }  // BST_Destroy
```

Internal Destroy Function

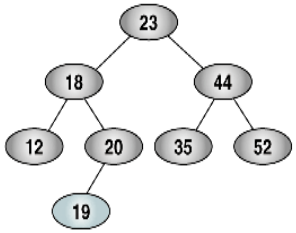
Özyinelemeli silme fonksiyonundaki mantık biraz daha karmaşıktır. Büyük soru şudur: Verileri ve düğümü ne zaman sileceğiz? Her birini yalnızca bir kez ve doğru zamanda yaptığımızdan emin olmalıyız.

Sol alt ağaçtan döndükçe veriler silinir. Bu, 16. deyimde yapılır. Ancak, düğümü silmeye henüz hazır değiliz, çünkü doğru alt ağacı işlemedik.

Silmek için sağ alt ağaç geçişinden geri dönene kadar beklemeliyiz (bkz. Deyim 18).

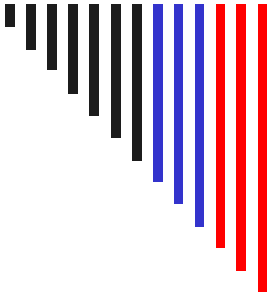
Internal Destroy Function

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



Lab Uygulaması

Kitaptaki PROGRAM 7-16 BST Integer Application örnek uygulamasında BST ADT'nin kullanım biçimi incelenecek ve uygulama çalıştırılacak.



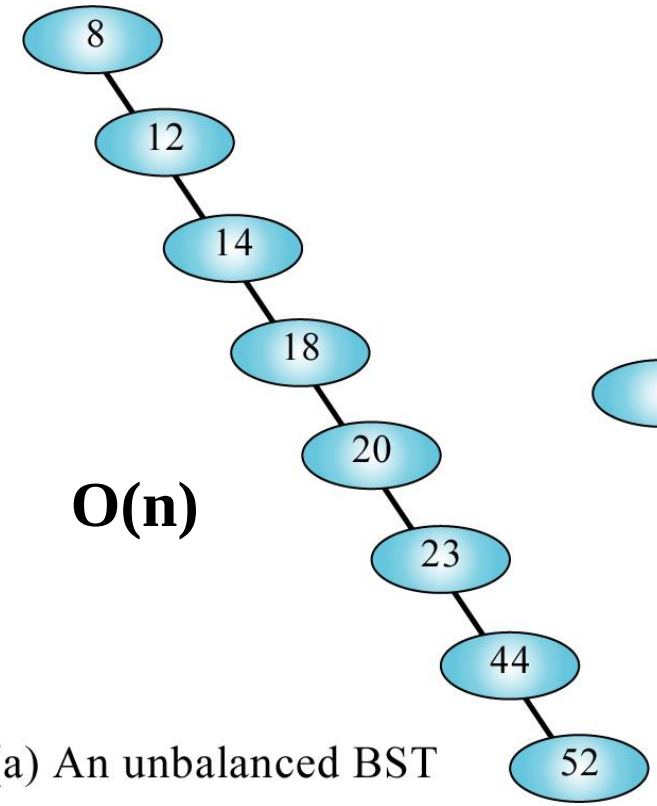
AVL Search Trees

AVL Trees

1962 yılında, iki Rus matematikçi, G.M. Adelson-Velskil ve E. M. Landis dengeli bir ikili ağaç yapısı oluşturdu ve bu yapı onlardan sonra AVL ağaçları adıyla anıldı.

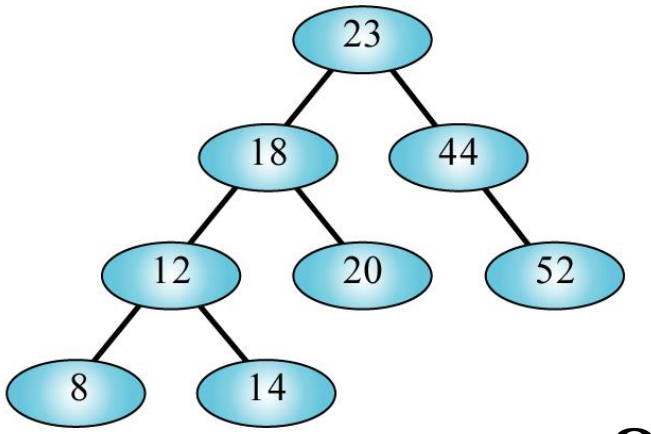
AVL ağacı, alt ağaçlarının yükseklik farkının en fazla 1 olduğu bir arama ağacıdır.

- Dolayısıyla dengeli ikili ağaçtır (**balanced binary tree**)



(a) An unbalanced BST

O(n)



(b) An AVL tree

$O(\log_2 n)$

Max. search effort

$$|H_L - H_R| <= 1$$

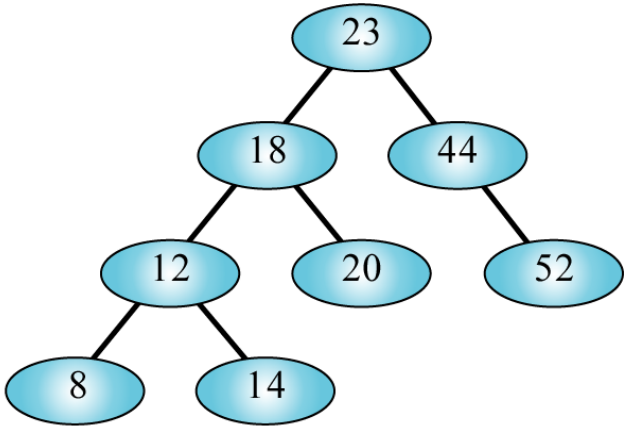
Height balanced trees

Bu isimle de anılırlar

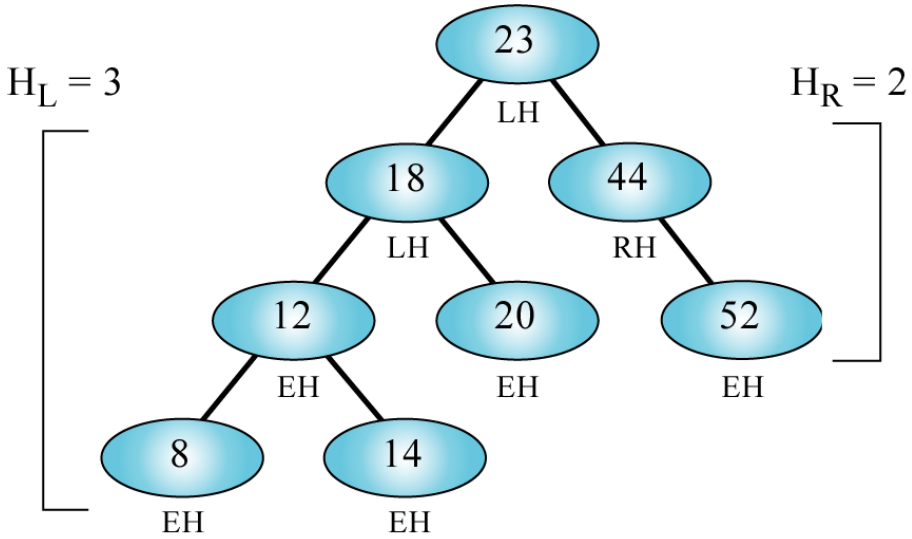
AVL Trees

Bir **AVL ağacı**, boş ya da aşağıda gösterildiği gibi yükseklikleri 1'den büyük olmayan iki AVL alt ağacı olan T_L ve T_R'den oluşan ikili bir ağaçtır.

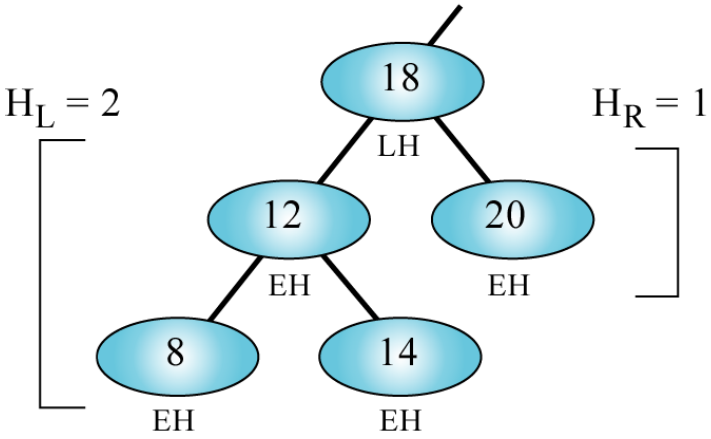
$$|H_L - H_R| <= 1$$



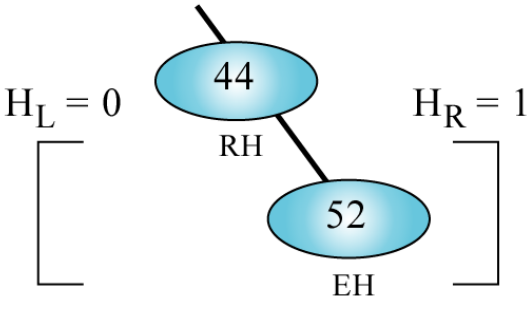
(b) An AVL tree



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



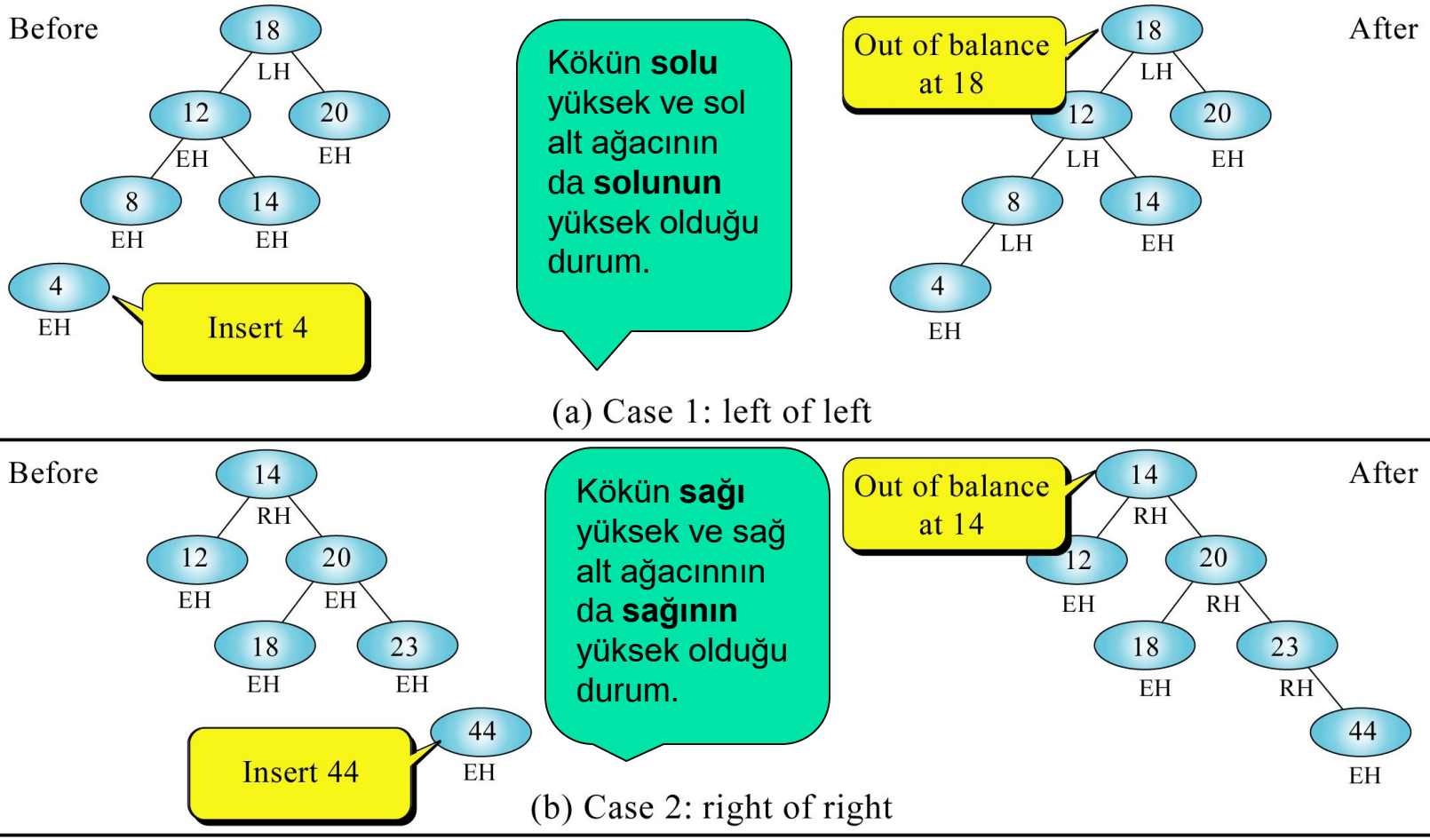
(b) Subtree 18 appears balanced:
$H_L - H_R = 1$

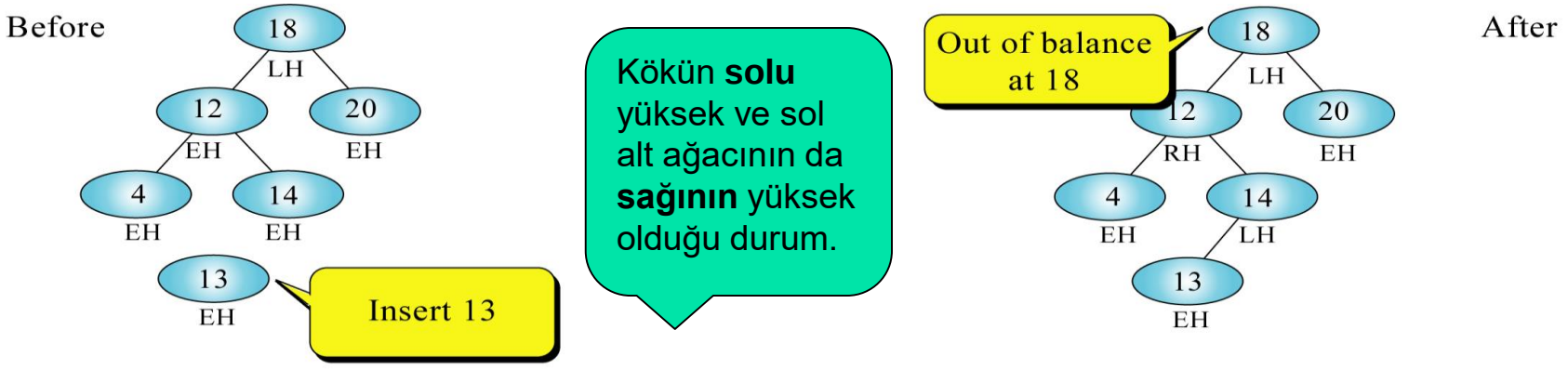


(c) Subtree 44 is balanced:
$| H_L - H_R | = 1$

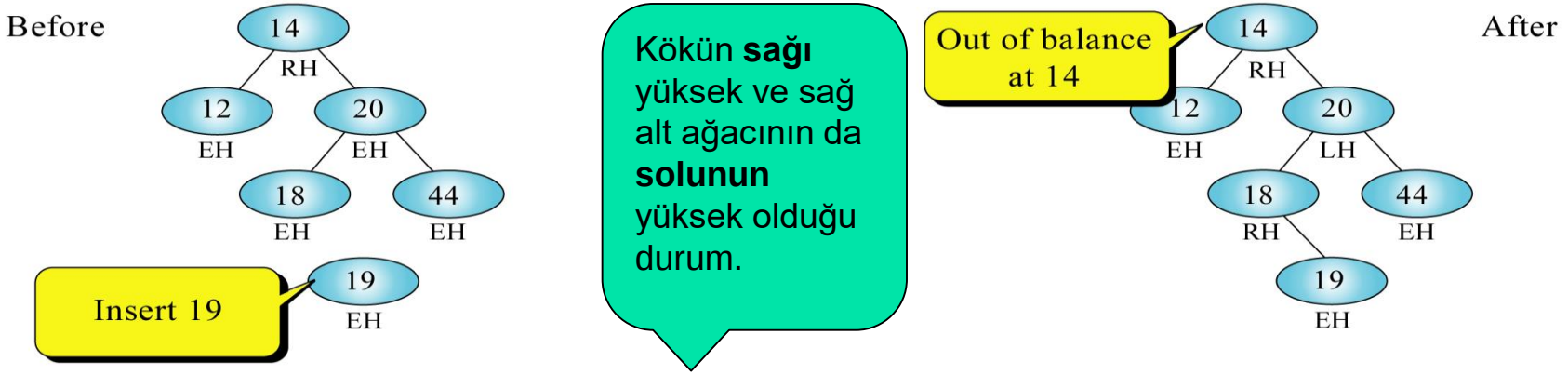
AVL Trees – Balancing

Ağaca bir düğüm eklediğimizde veya bir düğümü ağaçtan sildiğimizde, ortaya çıkan ağaç dengesiz olabilir ve onu yeniden dengelemeliyiz.





(c) Case 3: right of left



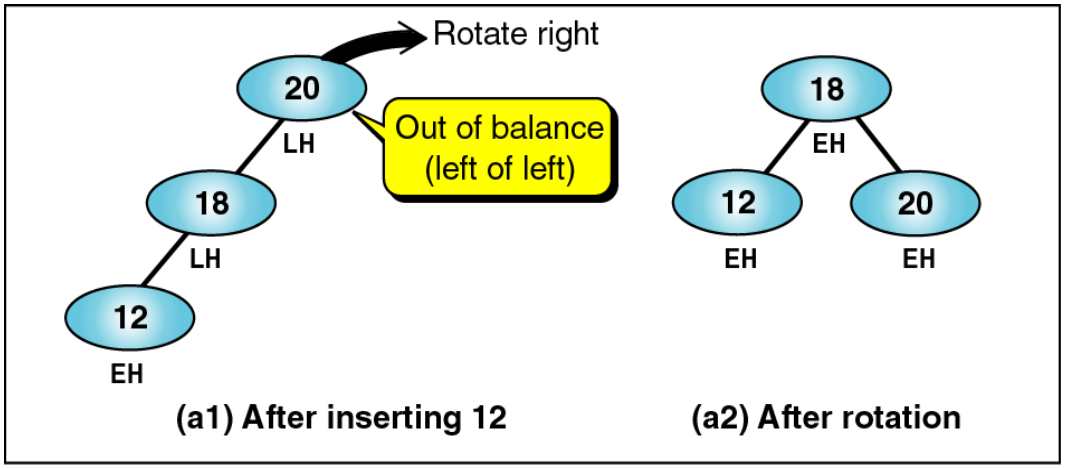
(d) Case 4: left of right

Tüm dengesiz ağaçlar bu dört durumdan birine uyar:

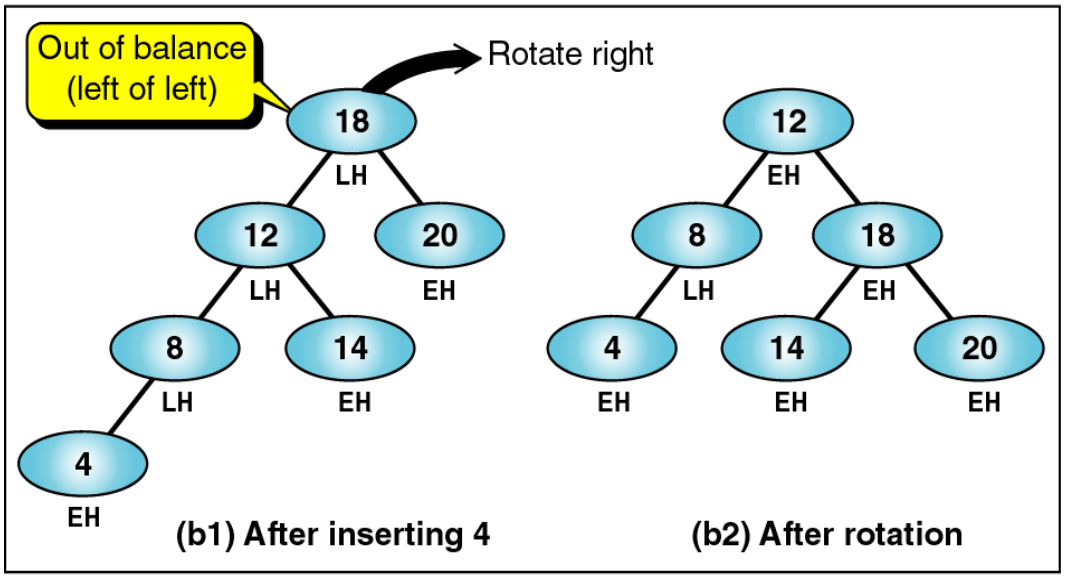
1. **Left of left**
2. **Right of right**
3. **Right of left**
4. **Left of right**

Case 1: Left of left

Dengede olmayan düğümü (out-of-balance node) sağa döndürerek solu yüksek ağacı dengeye getirmeliyiz.



(a) Simple right rotation

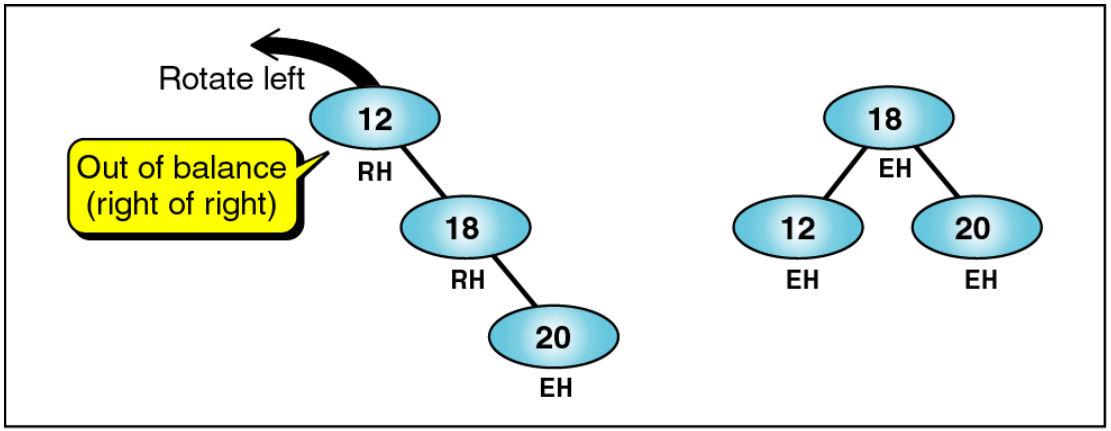


(b) Complex right rotation

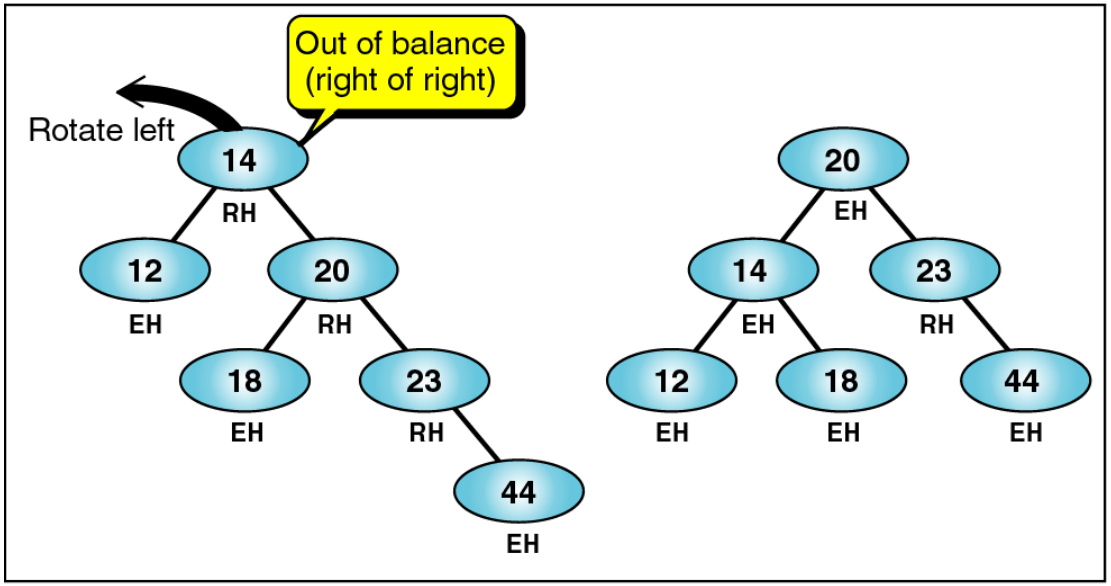
Eski kök düğüm(18), sol alt ağacını (12) rotasyon esnasında kaybeder. (Yeni kök düğüm 12 olur.) **Dolayısıyla 18'in boşta kalan sol koluna 14 tutturulur.**

Case 2: Right of right

Dengede olmayan düğümü (out-of-balance node) sola döndürerek sağı yüksek ağacı dengeye getirmeliyiz.



(a) Simple left rotation



(b) Complex left rotation

Figure 8-16

Dengesizliği düzeltmek için, kökü sola döndürüyoruz, sağ alt ağacı, 20, yeni kök yapıyoruz. Bu süreçte, 20'nin sol alt ağacı, **eski kökün sağ alt ağacı olarak bağlanır** ve bu da arama ağacının düzenini korur.

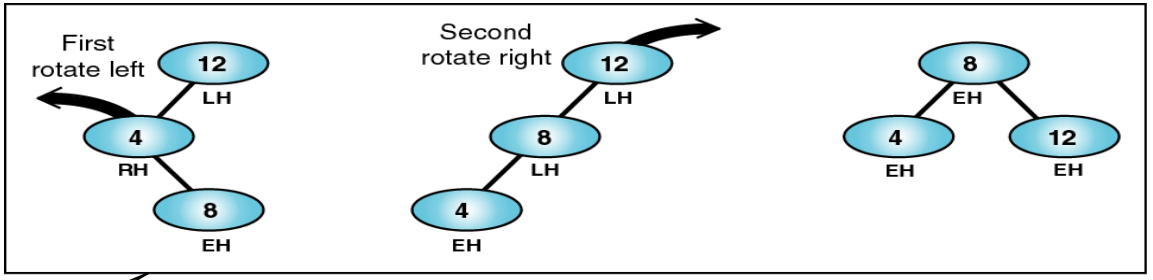
Case 3: Right of Left

1. **Rotate left**
2. **Rotate right**

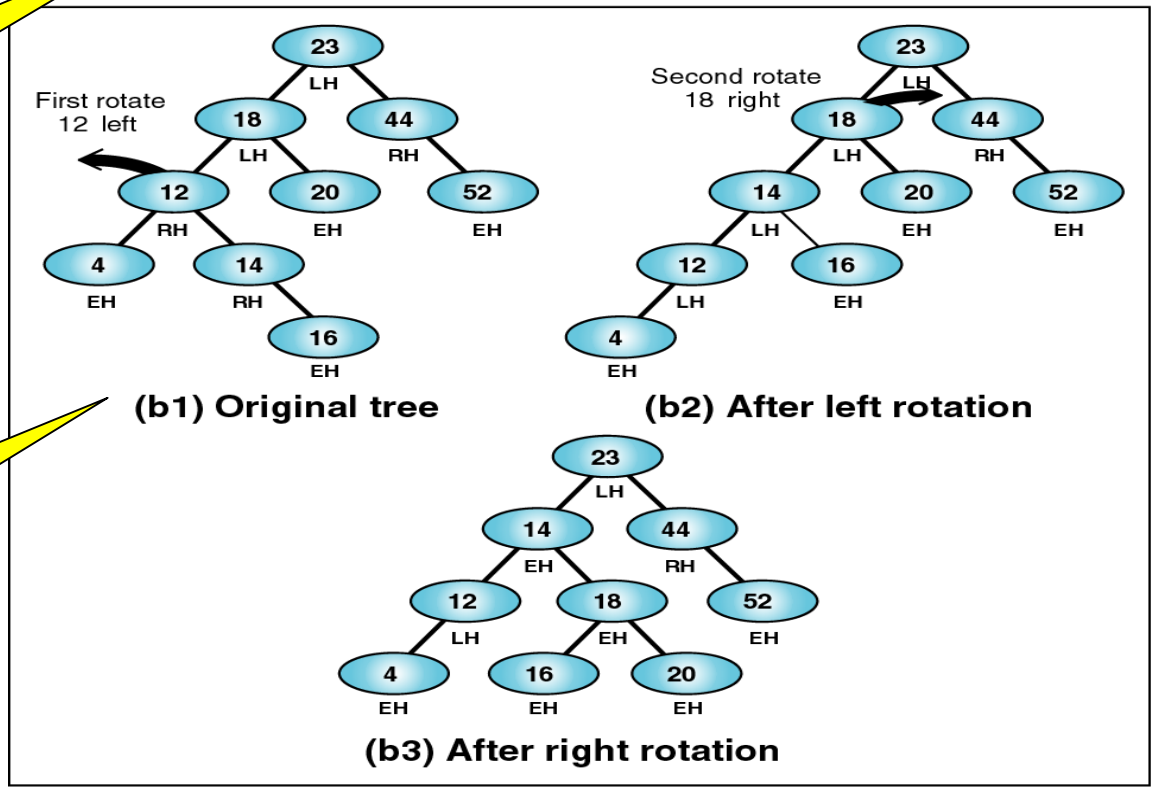
Ağacı dengeye getirmek için iki düğümü döndürmemiz gerekiyor, birini sola diğerini sağa.

Kökün solu yüksek ve sol alt ağacın da sağının yüksek bırakıldığı dengede olmayan bir ağaç görülüyor. (a **right of left** tree)

Bir düğümün (18) sol yüksek bırakıldığı ve sol alt ağacının (12) sağ yüksek olduğu bir dengede olmayan ağacı görüyoruz.



(a) Simple double rotation right



(b1) Original tree

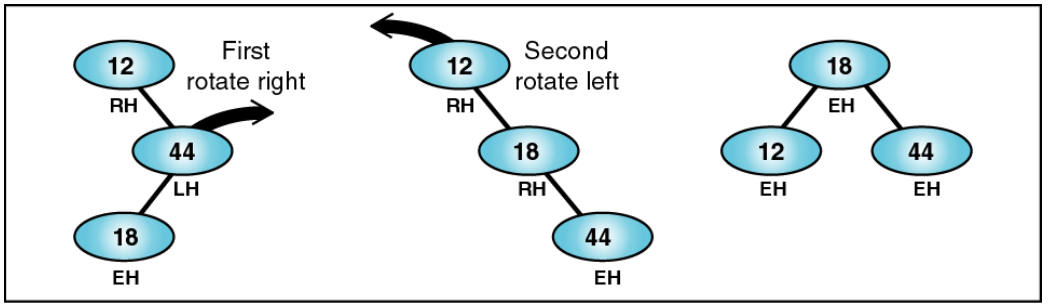
(b2) After left rotation

(b3) After right rotation

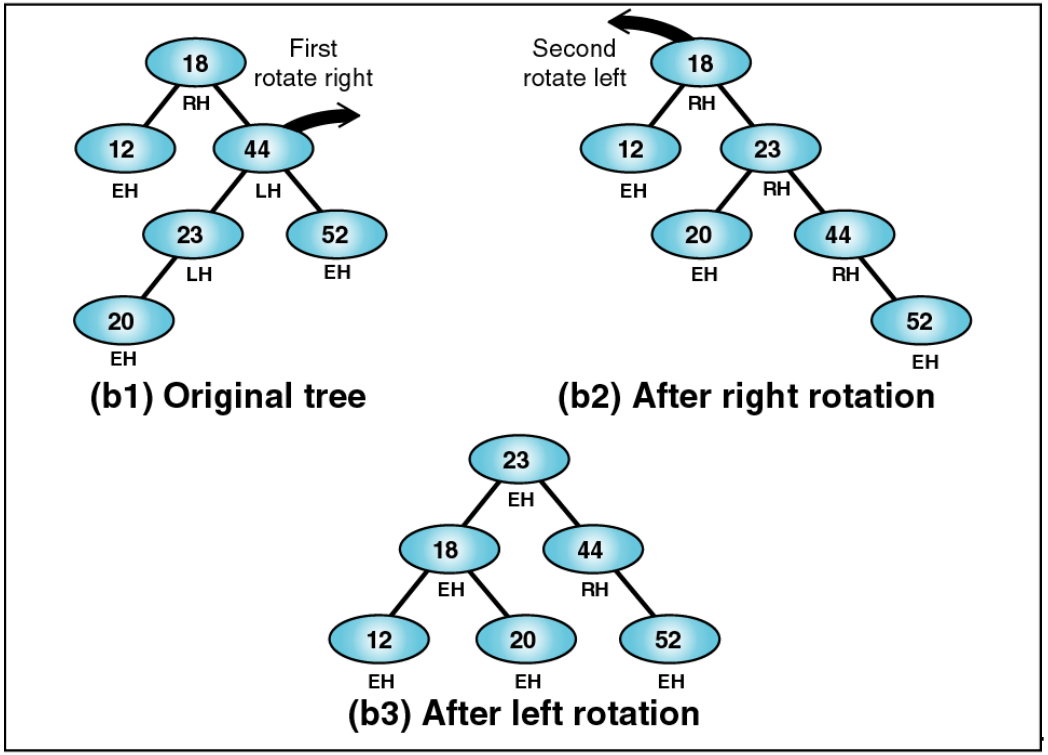
(b) Complex double rotation right

Case 4: Left of Right

1. **Rotate right**
2. **Rotate left**



(a) Simple double rotation right



(b1) Original tree

(b2) After right rotation

(b3) After left rotation

(b) Complex double rotation right

Denge Faktörlerini ayarlama (Adjusting Balance Factors)

Bir ekleme veya silme işleminden sonra, ağacı dengelediğimiz için denge faktörlerini ayarlamamız gerekir.
Denge faktörlerinde yapılan ayarlamalar her durum için ayrı ayrı analiz edilmekle birlikte, genel bir örüntü vardır:

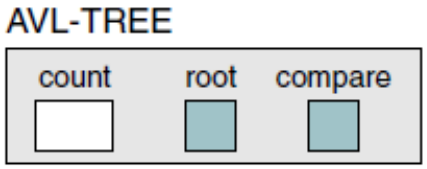
- Eğer eklemeden önce kök dengeliyse, eklemeden sonra eklemenin yapıldığı tarafı daha yüksek hale gelir.

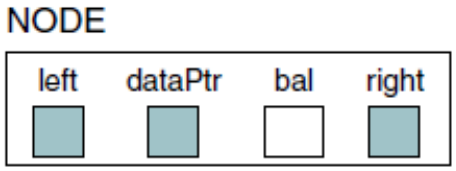
- Eğer ekleme, dengede olmayan ağacın kısa olan alt ağacı tarafına yapılmışsa, kök artık dengeye gelmiştir.

- Eğer ekleme, dengede olmayan ağacın yüksek tarafına yapıldıysa, kök döndürülmelidir.

AVL Tree Data Structure

Diğer ADT yapılarında olduğu gibi AVL ağacı ADT'yi de «**head structure**» ve «**node structure**» ile uyguluyoruz.






```
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;
```

Denge faktörü

+1 →LH
0 →EH
-1 →RH

AVL Head Structure

AVL ağaç baş yapısı, AVL_TREE,

- bir sayı,
- bir kök işaretçisi ve
- listeyi aramak için gereken karşılaştırma fonksiyonunun adresini içerir.

Uygulama programının ağacı görüş alanı sadece head structure'a işaret eden bir pointer'dır.

AVL Node Structure

Node

key	<key type>
data	<data type>
leftsubtree	<pointer to Node>
rightsubtree	<pointer to Node>
bal	<LH, EH, RH>

End Node

```
#define LH +1 // Left High
#define EH 0 // Even High
#define RH -1 // Right High
```

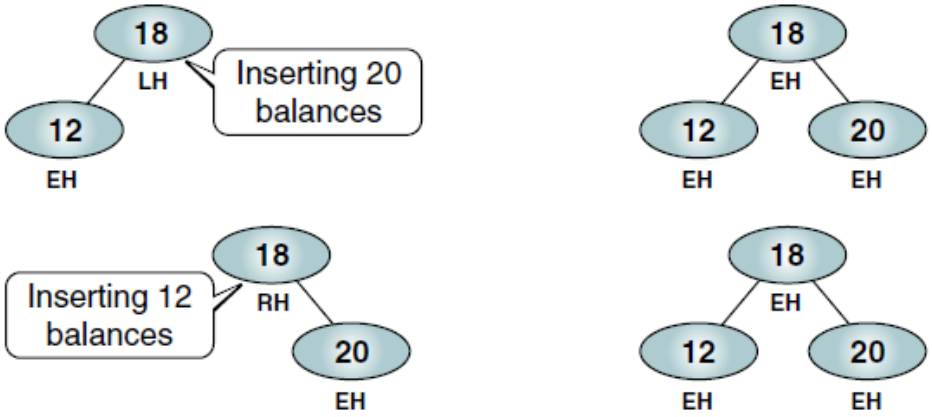
AVL Insert

- Arama (search) ve alma/getirme (retrieval) algoritmaları, herhangi bir ikili ağacınkiyle aynıdır.
- Doğal olarak **Inorder** gezinme kullanıyoruz çünkü AVL ağaçları, arama ağaçlarıdır.
- Bununla birlikte, ekleme ve silme algoritmaları yeniden yazılmalıdır.
- İkili arama ağacı olarak, sol veya sağ alt ağaçta uygun **yaprak** düğümünü bulmalıyız, sonra bulunan bu üst/ebeveyn düğüme yeni düğümü bağlayıp geriye doğru gitmeliyiz.
- Geriye doğru yukarıya çıktıkça her bir düğümün dengesini kontrol ederiz. **Dengesiz bir düğüm bulursak onu dengeler ve ağaçta yukarı doğru devam edilir.**
  - **AVL ağaç ile ikili arama ağacının faklılaştığı nokta burasıdır.**

AVL Insert

- Bütün eklemeler dengesiz bir durum yaratmaz.

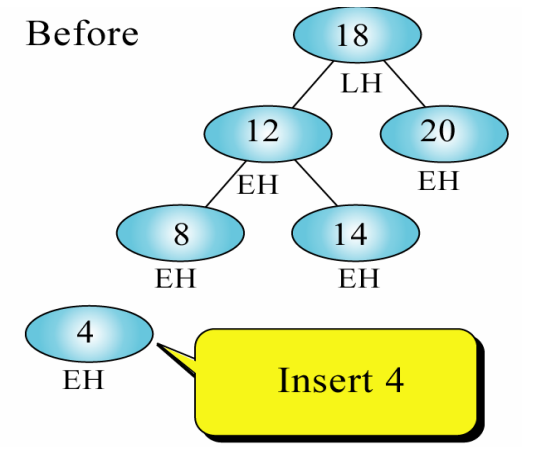
Sol yüksek düğümün sağ koluna bir düğüm eklediğimizde, **otomatik dengeleme** meydana gelir ve düğüm şimdi dengededir (**even high**).

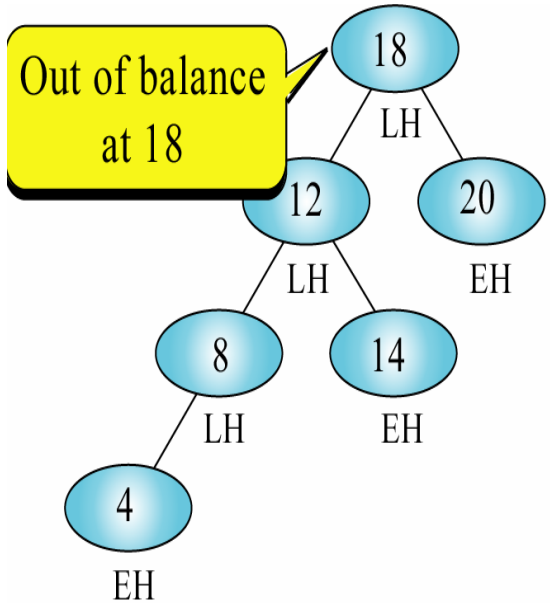


Automatic AVL Tree Balancing

Tersine, sağ yüksek düğümün sol koluna bir düğüm eklediğimizde, düğüm otomatik olarak dengelenir.

AVL Insert Algorithm





AVL Tree Insert

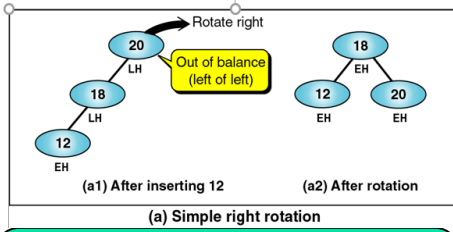
```
Algorithm AVLInsert (root, newData)
Using recursion, insert a node into an AVL tree.
   Pre    root is pointer to first node in AVL tree/subtree
          newData is pointer to new node to be inserted
   Post   new node has been inserted
   Return root returned recursively up the tree
 1 if (subtree 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 Left Balance Algorithm

Bir sol alt ağacı dengeleme mantığı ve bir sağ alt ağacı dengeleme mantığı birbirinin aynası (mirror) olduğundan, yalnızca sol alt ağacı dengeleme mantığı verilmiştir.

AVL Tree Left Balance

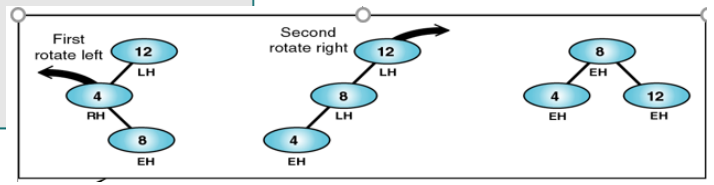
```
Algorithm leftBalance (root)
This algorithm is entered when the root is left high (the
left subtree is higher than the right subtree).
   Pre    root is a pointer to the root of the [sub]tree
   Post   root has been updated (if necessary)
 1 if (left subtree high)
   1  rotateRight (root)
 2 else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



(a) Simple right rotation

Case 1: Left of left, single rotation required.

Case 2: Right of left. Double rotation required



AVL Rotate Algorithms

algorithm **rotateRight**(ref root <tree pointer>)

This algorithm exchanges pointers to rotate the tree right.

PRE root points to tree to be rotated.

POST Node rotated and root updated.

1 tempPtr = root->left

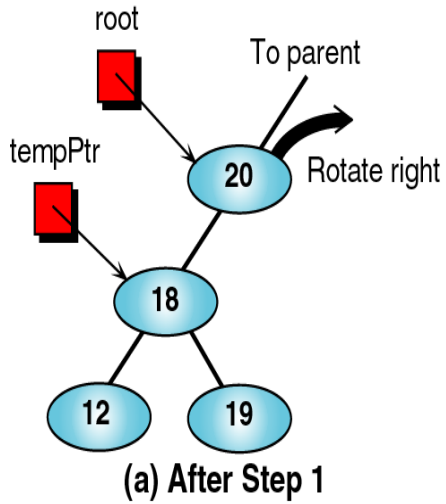
2 root->left = tempPtr->right

3 tempPtr->right = root

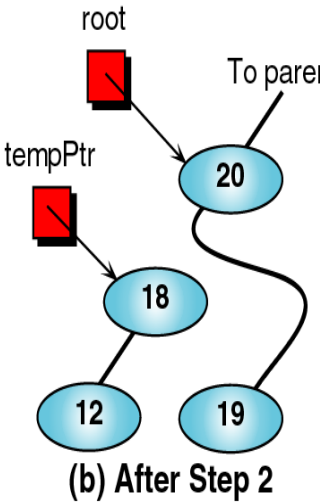
4 root = tempPtr

5 return

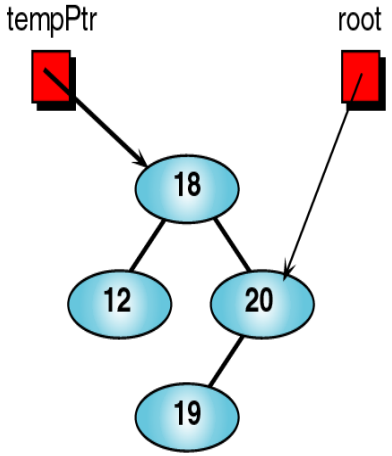
end **rotateRight**



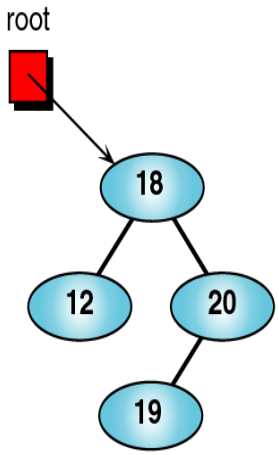
(a) After Step 1



(b) After Step 2



(c) After Step 3



(d) After Step 4

Ödev: AVL **RotateLeft** Algorithm

algorithm **rotateLeft**(ref root <tree pointer>)

This algorithm exchanges pointers to rotate the tree left.

PRE root points to tree to be rotated.

POST Node rotated and root updated**.**

……..

……

……

……

……

……..

…….

end **rotateLeft**

AVL Rotate Algorithms

ALGORITHM 8-3 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

- İkili arama ağacındaki gibi, tüm silme işlemlerinin bir **yaprak düğümünde** yapılması gerekir.
- Silme mantığı, ikili ağaçtaki silme mantığının aynısını izler
  - İlave olarak ağacı dengeleme işleminin de içerir
- Ekleme mantığında olduğu gibi, dengeleme ağaçta yukarıya çıktıkça gerçekleşir.
  - Delete Right Balance
  - Delete Left Balance

ALGORITHM 8-4 AVL Tree Delete

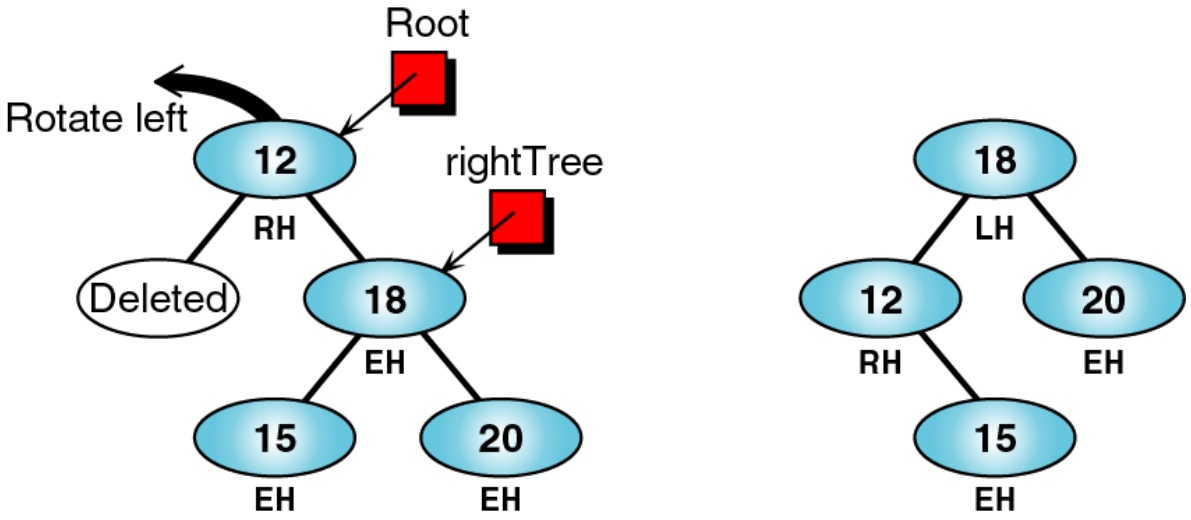
```
...
...
   Return pointer to root of [potential] new subtree
1  if Return (empty subtree)
      Not found
   1  set success to false
   2  return null
2  end if
3  if (dltKey < root)
   1  set left-subtree to AVLDelete(left subtree, dltKey,
                                    success)
   2  if (tree 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 Right Balance

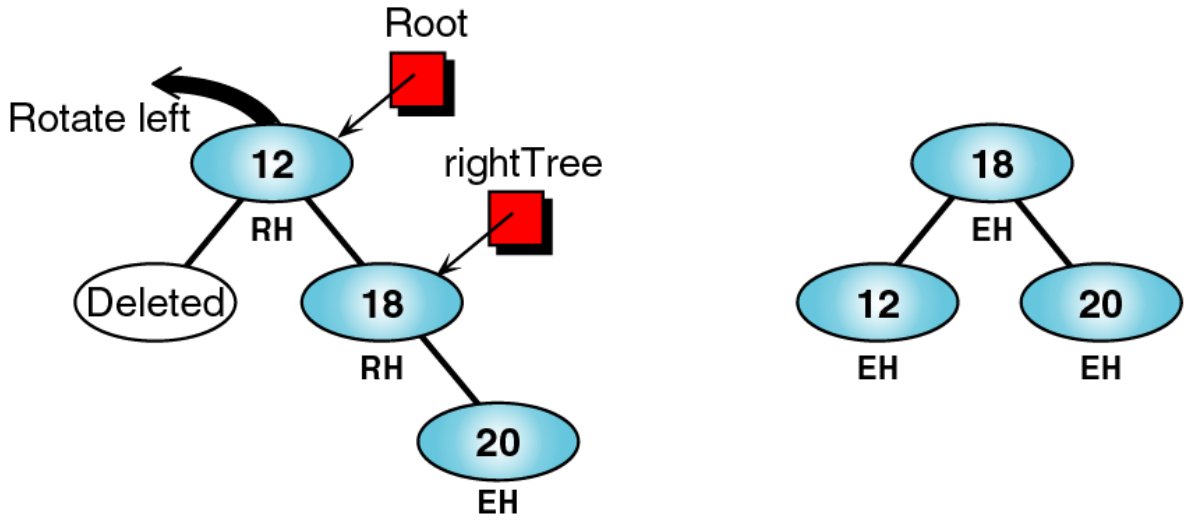
ALGORITHM 8-5 AVL Tree Delete Right Balance

```
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 Delete Balancing



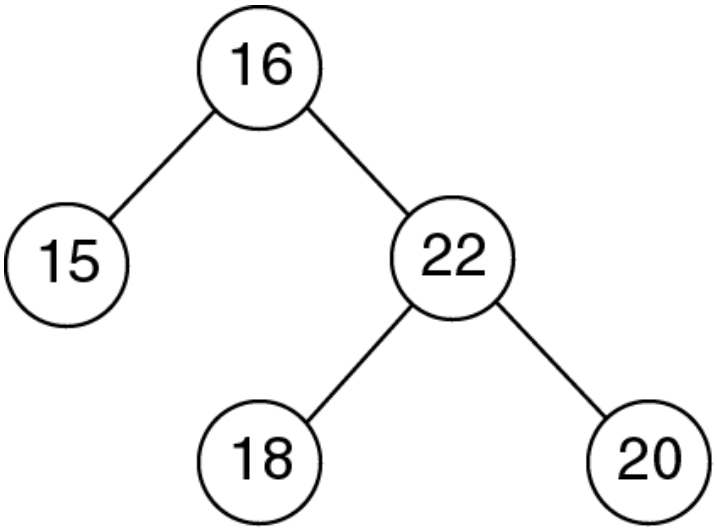
(a) Right subtree is even balanced



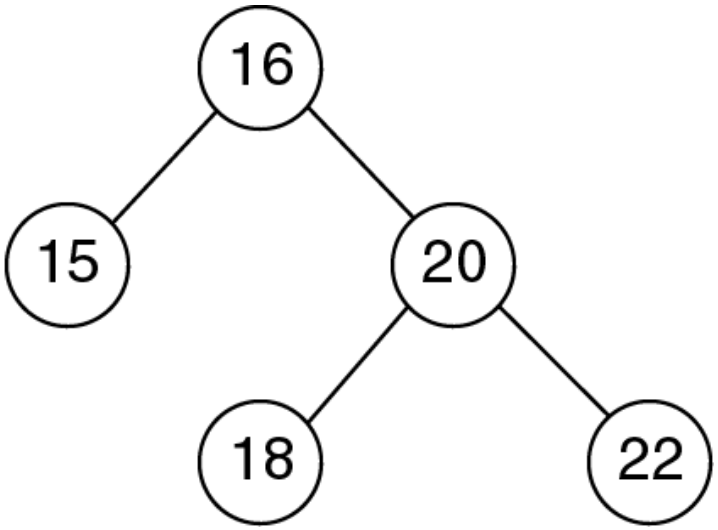
(b) Right subtree is not even balanced

Alıştırmalar

Aşağıdaki şekildeki ağaçlardan hangisi geçerli bir ikili arama ağacıdır ve hangisi değildir?



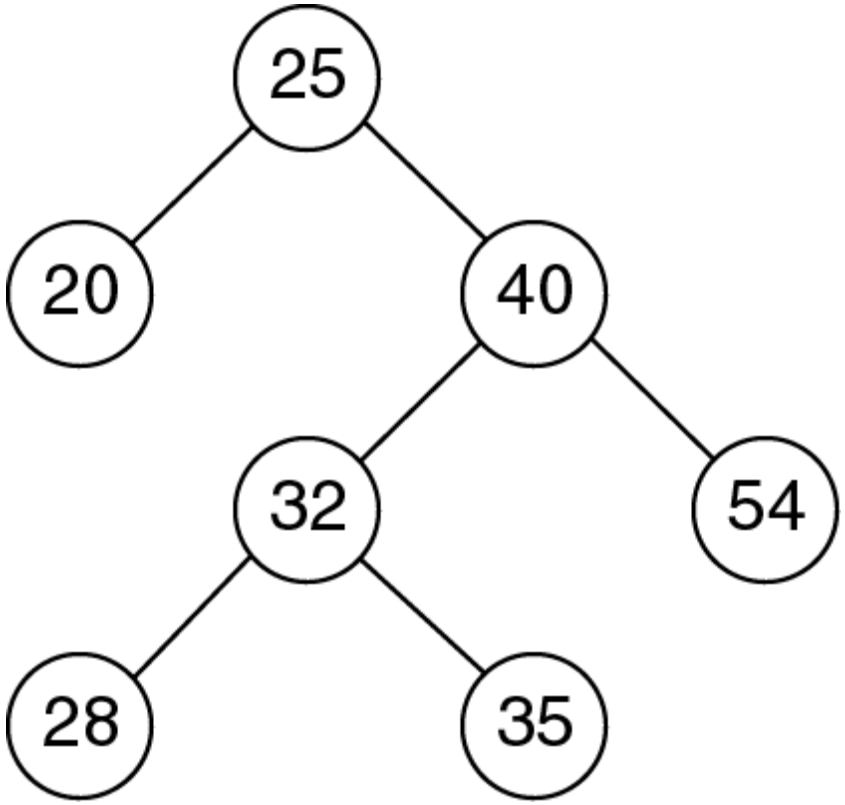
Geçersiz (invalid)



Geçerli (valid)

Alıştırmalar

Inorder gezinme tekniğiyle aşağıdaki ikili arama ağacını dolaşın.

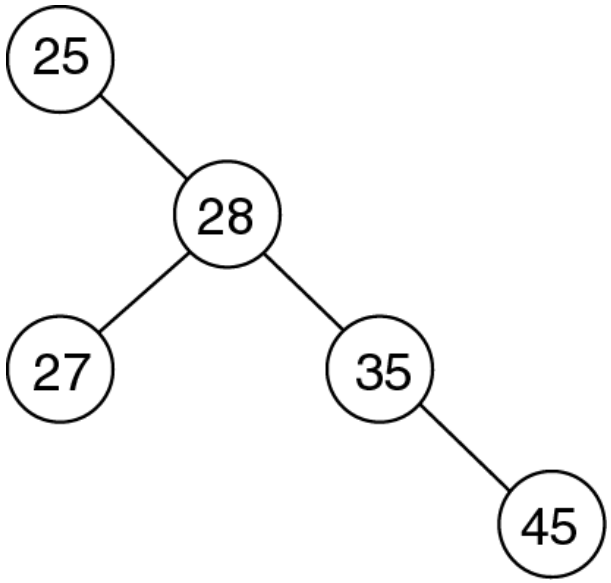


20 25 28 32 35 40 54

İkili arama ağacında **inorder** gezinme neyi verir?

Alıştırmalar

Aşağıdaki ikili arama ağacı boş bir ağaçtan başlayarak ve klavyeden veri girerek oluşturulmuştur. Veriler hangi sırayla girilmiştir? Birden fazla olası dizi varsa, alternatifleri belirleyin.



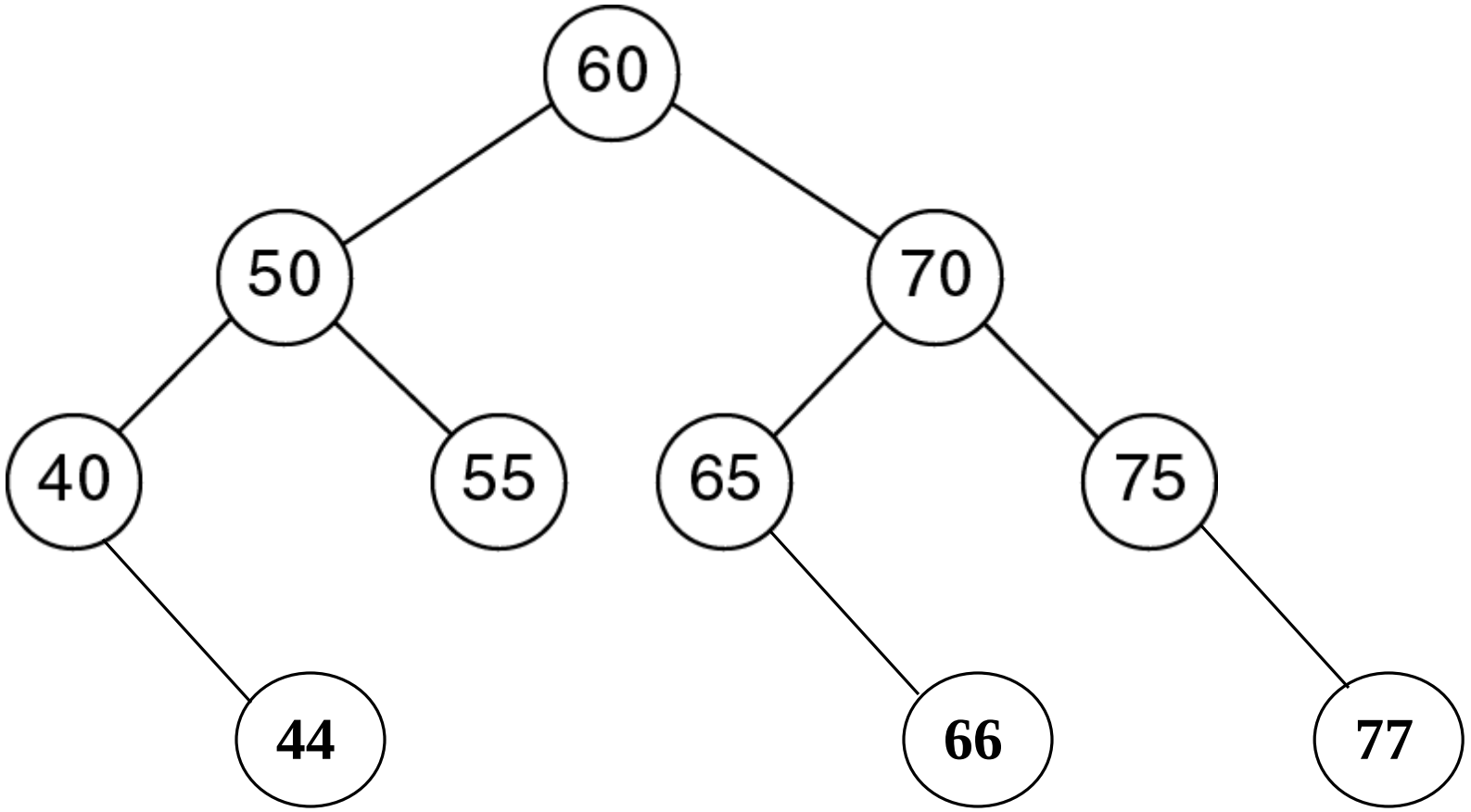
25 28 27 35 45

25 28 35 27 45

25 28 35 45 27

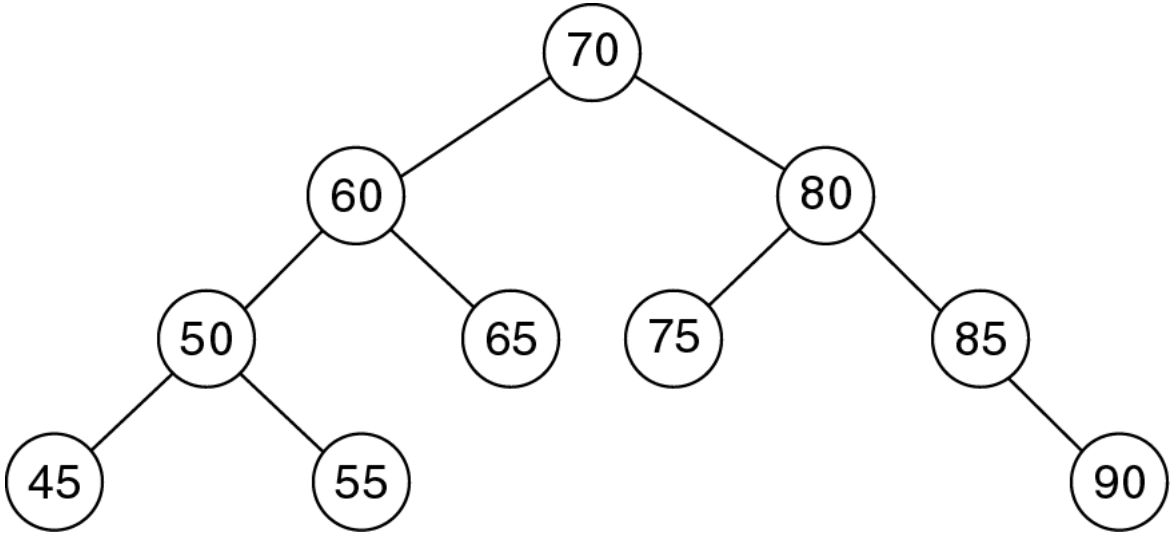
Alıştırmalar

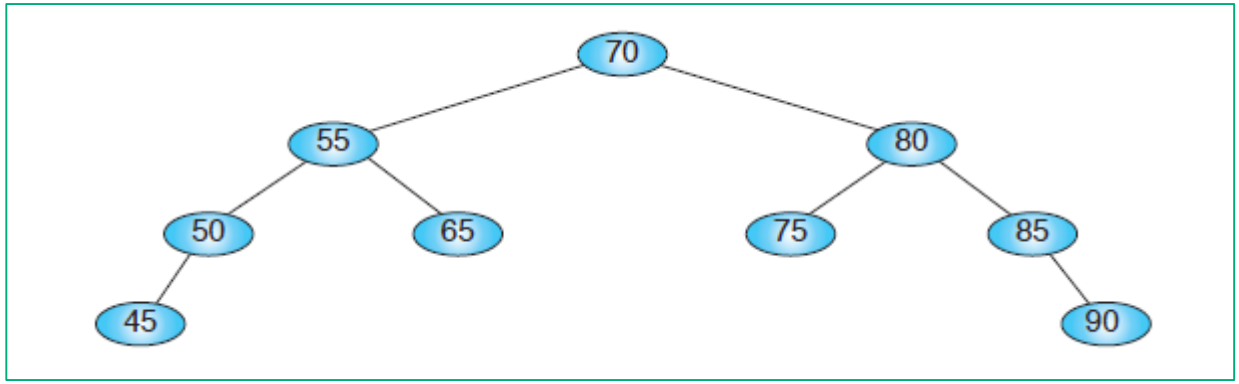
44, 66 ve 77'yi aşağıdaki ikili arama ağacına ekleyiniz.



Alıştırmalar

Aşağıdaki ikili arama ağacından 60'ı içeren düğüm siliniz.

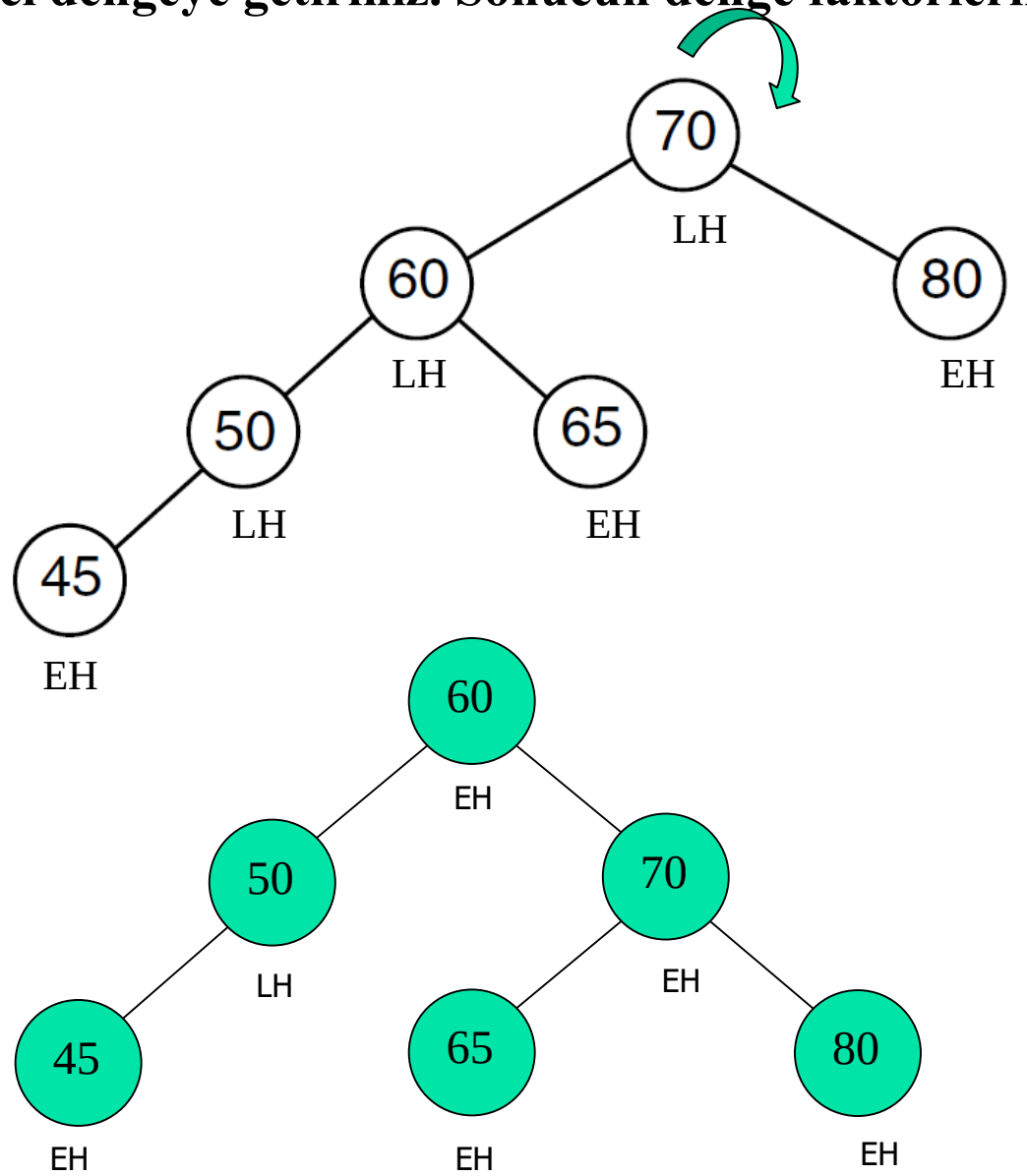




Alıştırmalar

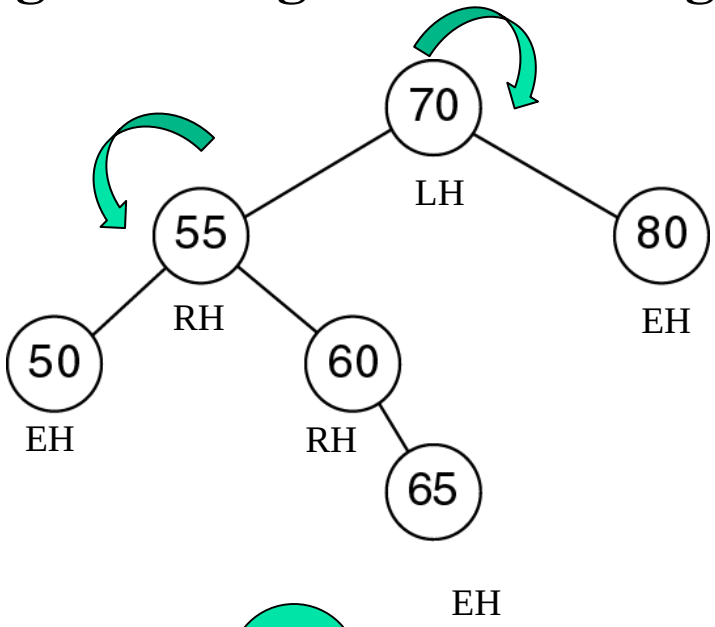
Aşağıdaki AVL ağacı dengeye getiriniz. Sonucun denge faktörlerini gösterin.

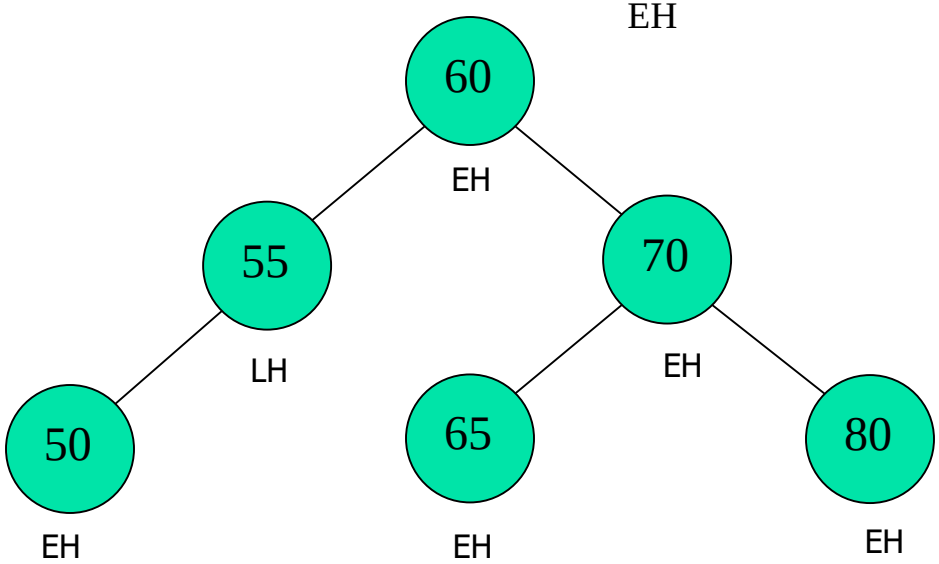
Left of Left case



Alıştırmalar

Aşağıdaki AVL ağacı dengeye getiriniz. Sonuçta ortaya çıkan ağacın denge faktörlerini gösteriniz.



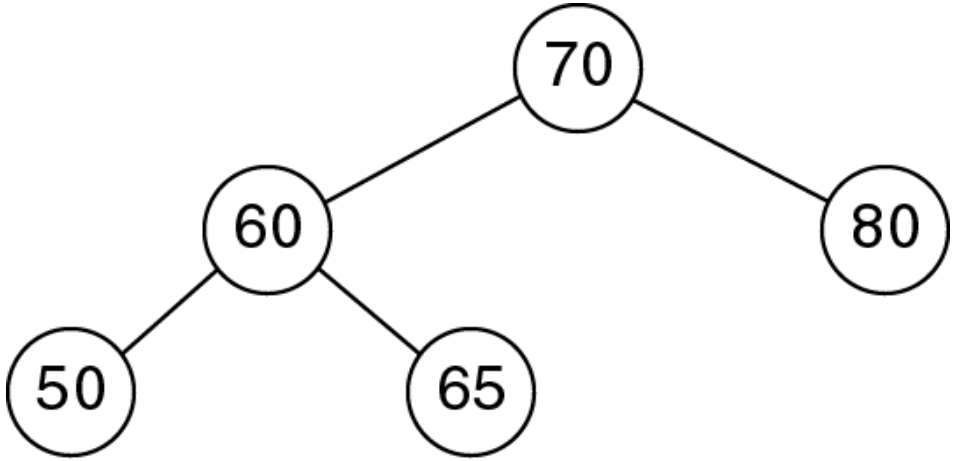


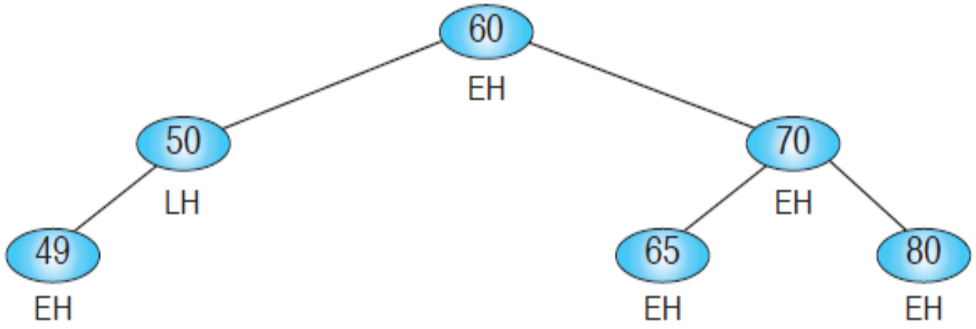
Right of Left case

1. 55' içreren düğüm sola söndürülür.
2. 70'i çeren düğüm (yani eski kök düğüm) sağa döndürülür.
3. Eski kökün boşta kalan sol koluna boşta duran 65'i içreren düğüm tutturulur.

Alıştırmalar

Aşağıda verilen AVL ağaca 49'u ekleyin. Sonuç yine bir AVL ağaç olmalıdır. Ortaya çıkan ağaçtaki denge faktörlerini gösteriniz.

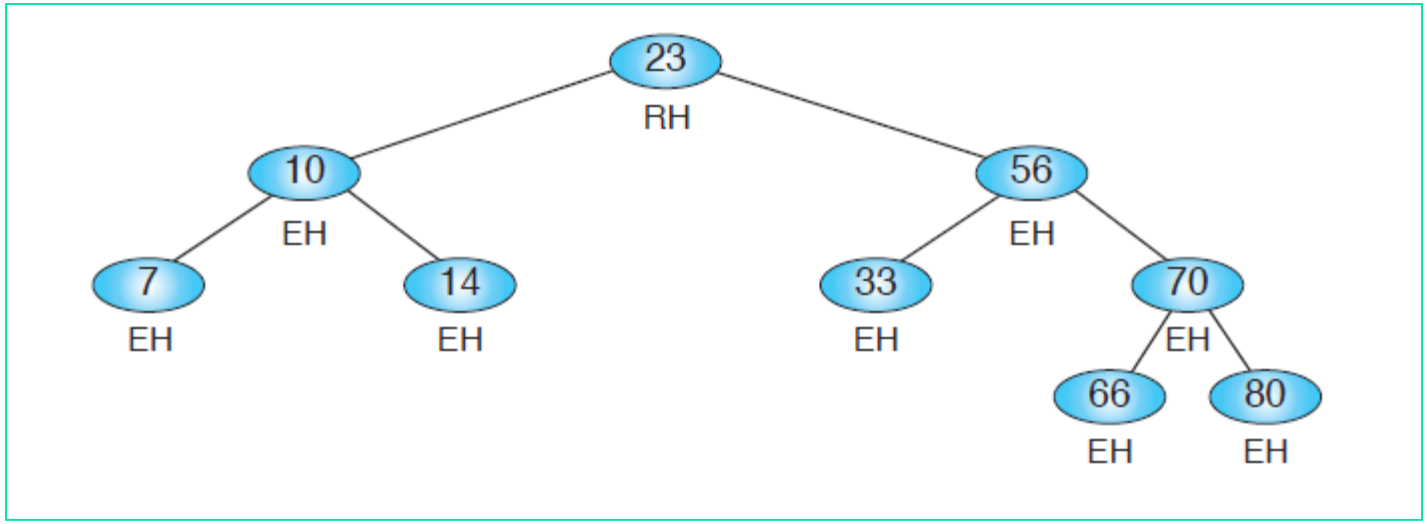




Alıştırmalar

Sıralı olarak girilen aşağıdaki verileri kullanarak bir AVL ağacı oluşturun. Nihai ağaçtaki denge faktörlerini gösterin.

7	10	14	23	33	56	66	70	80



Alıştırmalar

Aşağıdaki AVL ağaçtan değeri 70 olan düğümü silin.