CHAPTER 4

Data Structures and Algorithm Analysis in Java 3rd Edition by Mark Allen Weiss

TREES

- A tree is a collection of nodes, consisting of a node r called root and zero or more nonempty subtrees T_1, T_2,…. T_k, each of whose roots are connected by a direct edge from r.

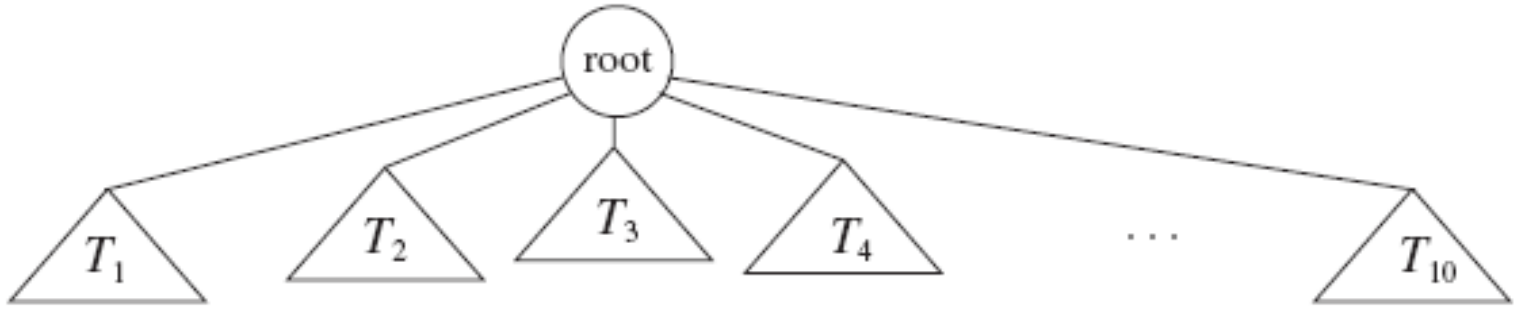


Figure 4.1 Generic tree

Example

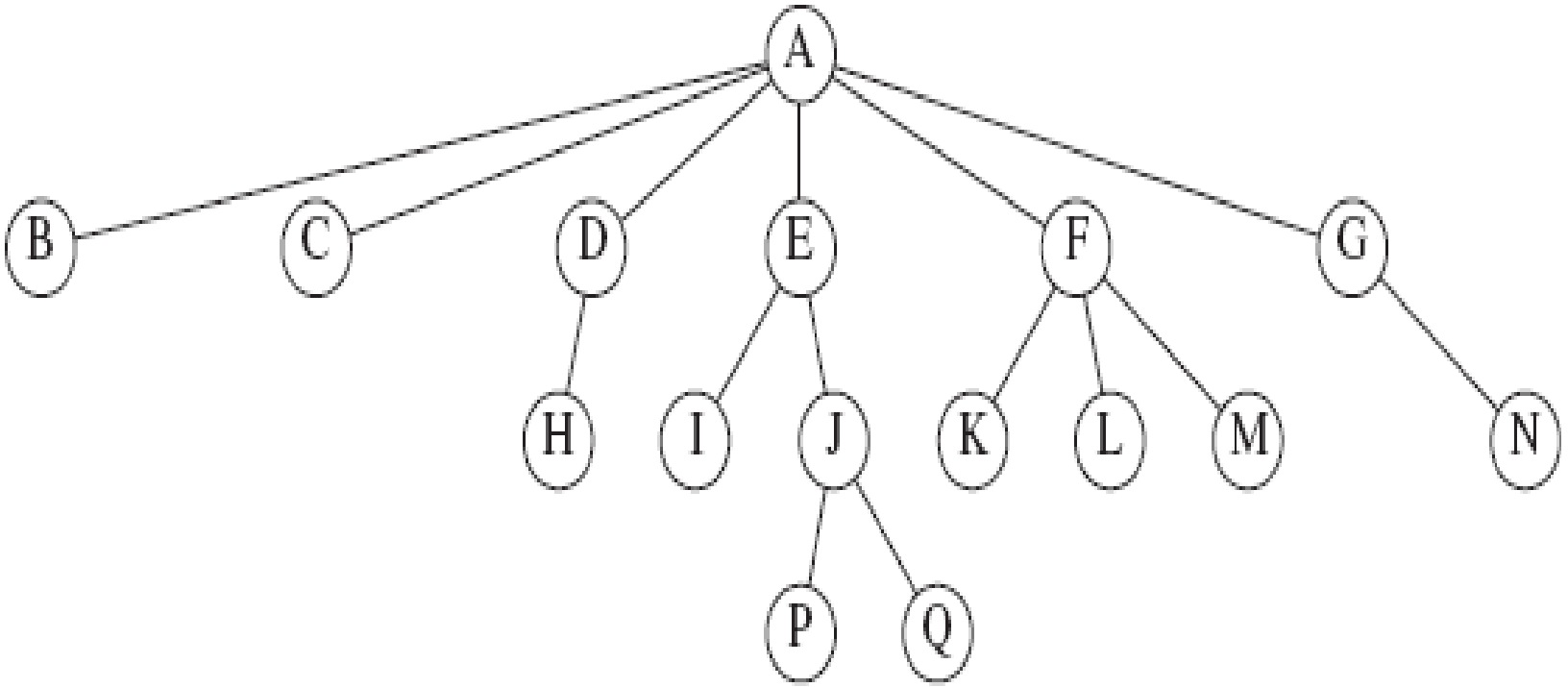


Figure 4.2 A tree

TERMS

- Path : from node n_1 to n_k is defined as sequence of nodes n_1, n_2,… n_k, such that n_i, is the parent of n_{i+1}, for 1 <= i <= k. The length of the path is k-1
- Depth of n_i: is the length of path from root to n_i.
- Height of n_i: is the length of longest path from n_i to a leaf
- Depth of root is 0 and height of all leaves are 0

IMPLEMENTATION OF TREES

- Number of children per node can vary greatly, we linked list of tree nodes.

```
class TreeNode
{
    Object   element;
    TreeNode firstChild;
    TreeNode nextSibling;
}
```

Figure 4.3 Node declarations for trees

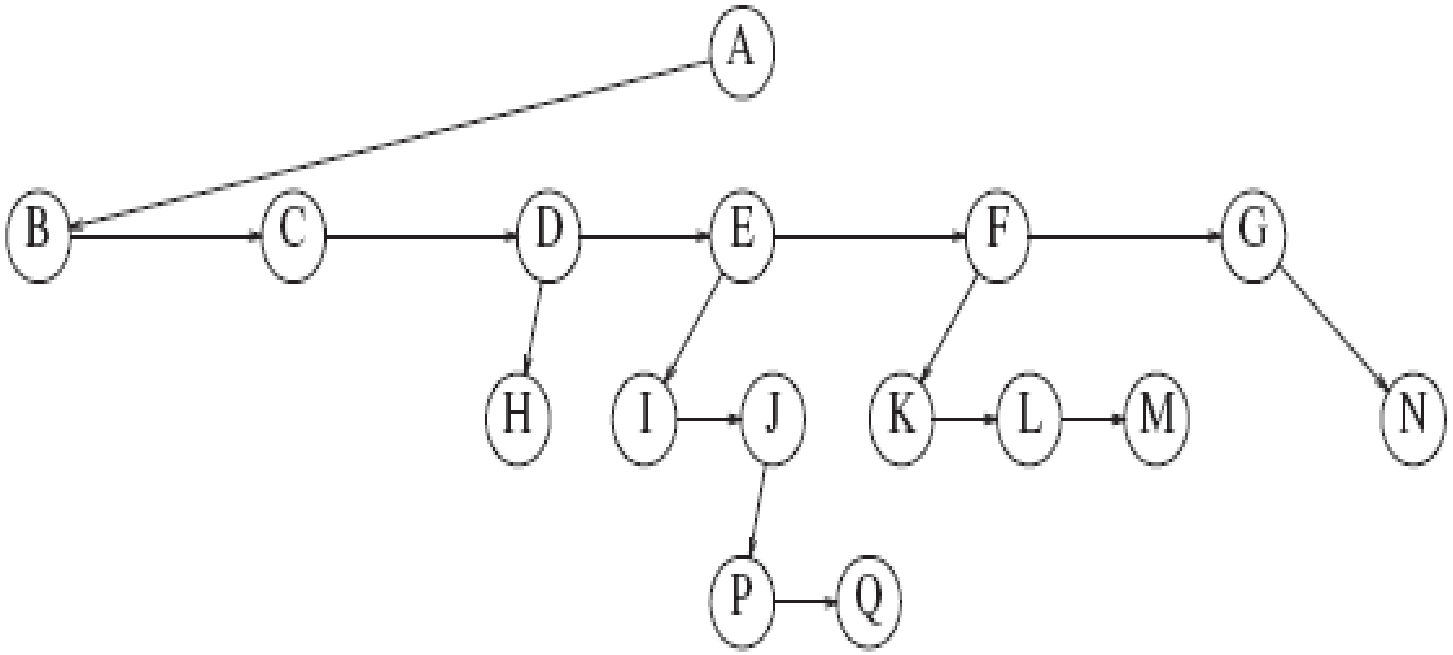


Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2

TREE TRAVERSALS WITH AN APPLICATION

- Directory structure in operating systems

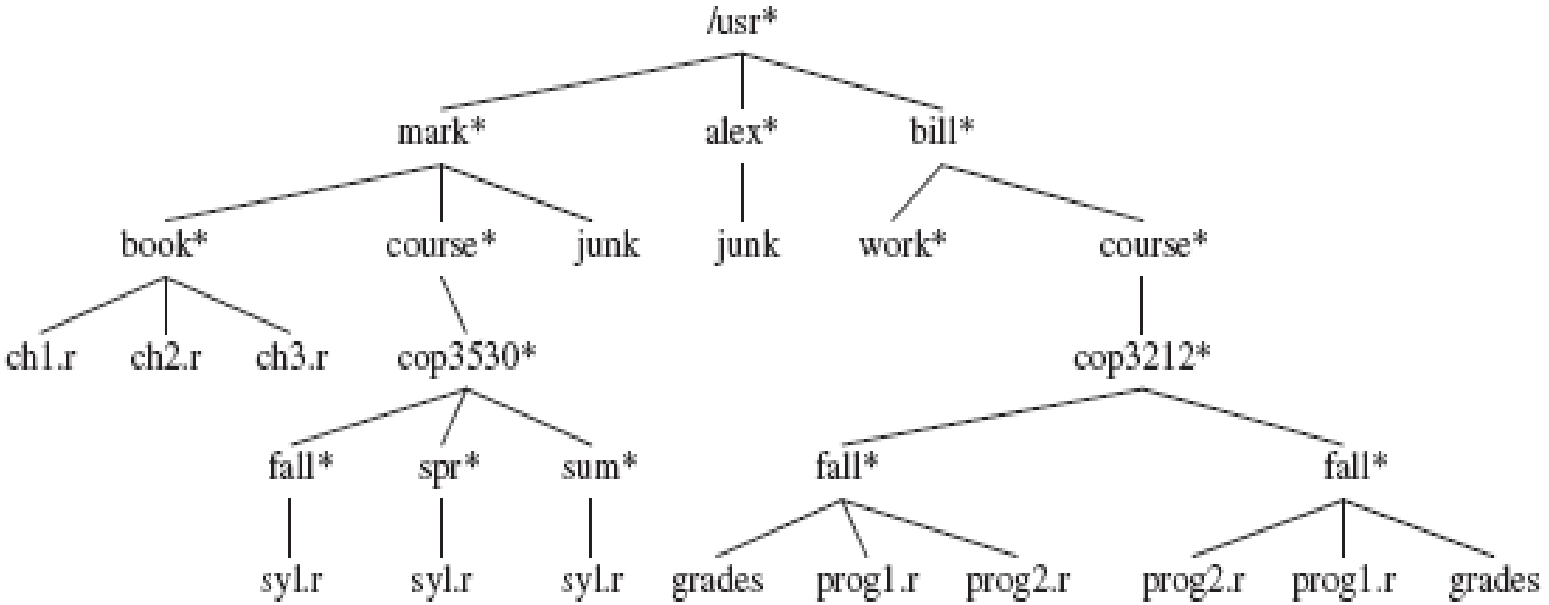


Figure 4.5 UNIX directory

List of Directories

```
1           private void listAll( int depth )
            {
2               printName( depth ); // Print the name of the object
3               if( isDirectory( ) )
4                   for each file c in this directory (for each child)
                        c.listAll( depth + 1 );
            }

            public void listAll( )
            {
                listAll( 0 );
            }
```

Figure 4.6 Pseudocode to list a directory in a hierarchical file system

PREORDER LISTING

```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall
                    syl.r
                spr
                    syl.r
                sum
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall
                    grades
                    prog1.r
                    prog2.r
                fall
                    prog2.r
                    prog1.r
                    grades
```

Figure 4.7 The (preorder) directory listing

POSTORDER TRAVERSAL

- The work at the node is performed after (post) its children are evaluated.
- Example: to calculate number of disk blocks

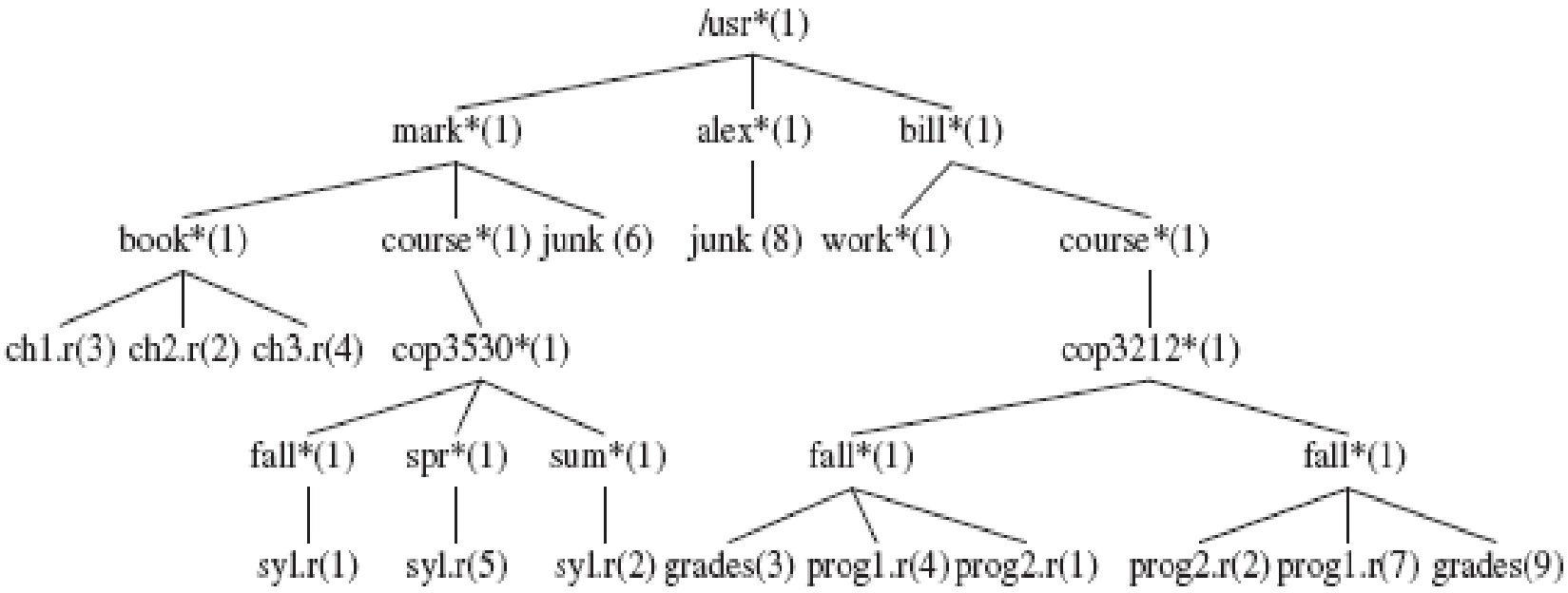


Figure 4.8 UNIX directory with file sizes obtained via postorder traversal

POSTORDER TRAVERSAL FOR CALCULATING THE SIZE

```
        public int size( )
        {
1           int totalSize = sizeOfThisFile( );

2           if( isDirectory( ) )
3               for each file c in this directory (for each child)
4                   totalSize += c.size( );

5           return totalSize;
        }
```

Figure 4.9 Pseudocode to calculate the size of a directory

```
            ch1.r            3
            ch2.r            2
            ch3.r            4
        book                10
                    syl.r    1
                fall         2
                    syl.r    5
                spr          6
                    syl.r    2
                sum          3
            cop3530          12
        course               13
        junk                  6
    mark                     30
        junk                  8
    alex                      9
        work                  1
                    grades   3
                    prog1.r  4
                    prog2.r  1
                fall         9
                    prog2.r  2
                    prog1.r  7
                    grades   9
                fall         19
            cop3212          29
        course               30
    bill                     32
/usr                         72
```

Figure 4.10 Trace of the `size` function

Binary Tress

- A binary tree is a tree in which no node can have more than two children.

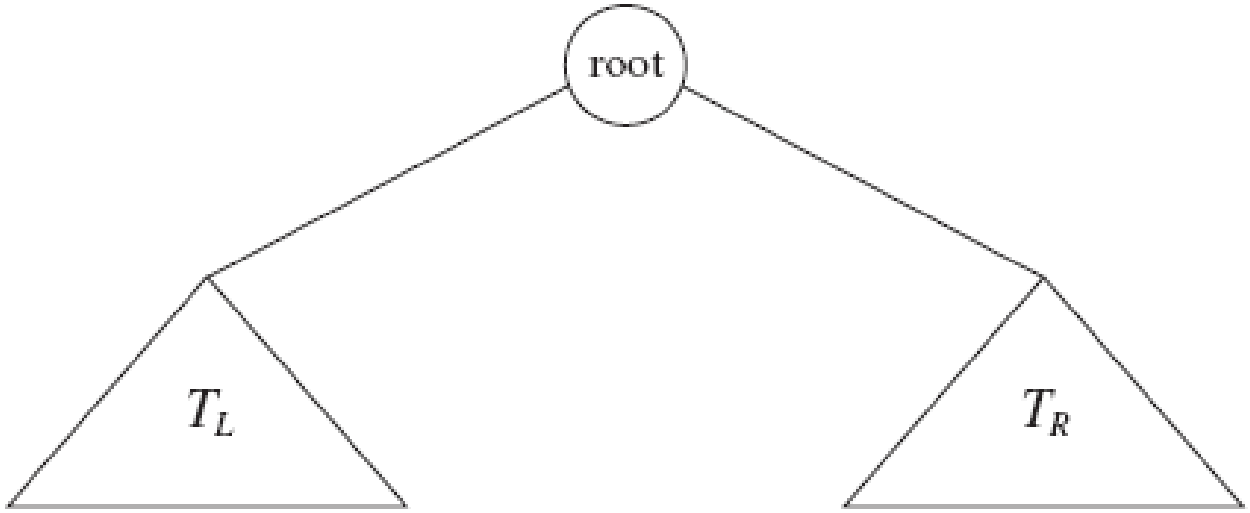


Figure 4.11 Generic binary tree

PROPERTIES

- Depth of an average binary tree is considerable smaller than N. O($\sqrt{N}$).
- For special cases like binary search trees, the average value of depth is O(logN)
- Unfortunately it can be as large as N-1

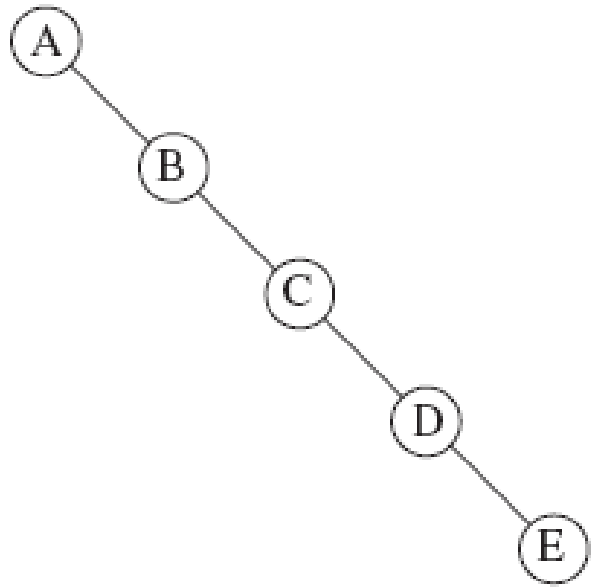


Figure 4.12 Worst-case binary tree

IMPLEMENTATION

```
class BinaryNode
{
        // Friendly data; accessible by other package routines
    Object     element;      // The data in the node
    BinaryNode left;         // Left child
    BinaryNode right;        // Right child
}
```

Figure 4.13 Binary tree node class

An Example: Expression Trees

- The leaves of an expression tree are operands, such as constants or variable names and the other nodes contain operators.

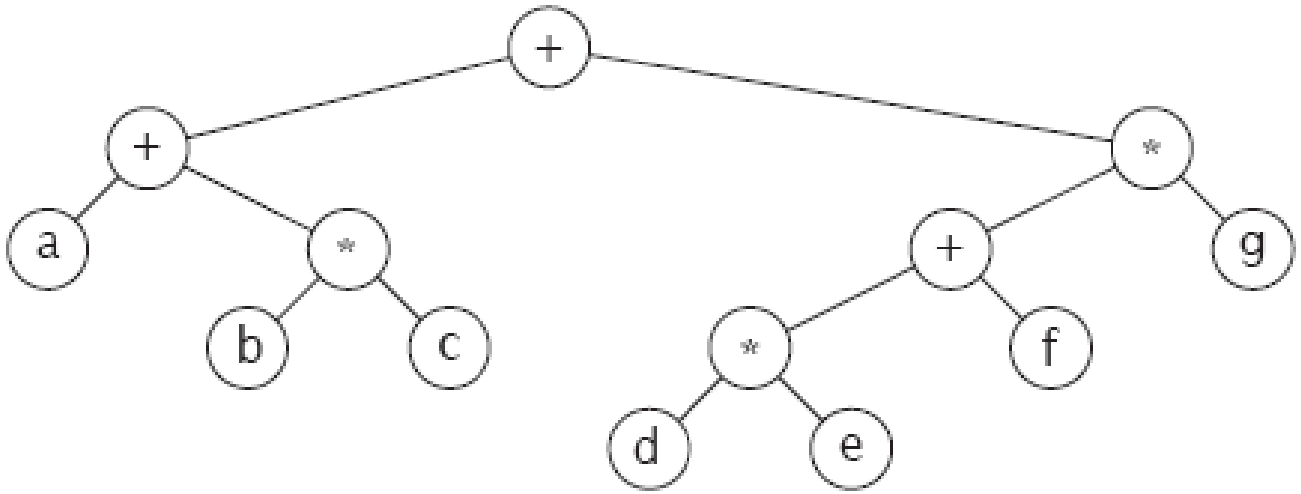


Figure 4.14 Expression tree for `(a + b * c) + ((d * e + f ) * g)`

TRAVERSALS

- INORDER : (left, node, right)
  - Example: ((a + (b * c)) + (((d * e) + f) * g))

- POSTORDER : left right node
  - Example: a b c * + d e * f + g * +

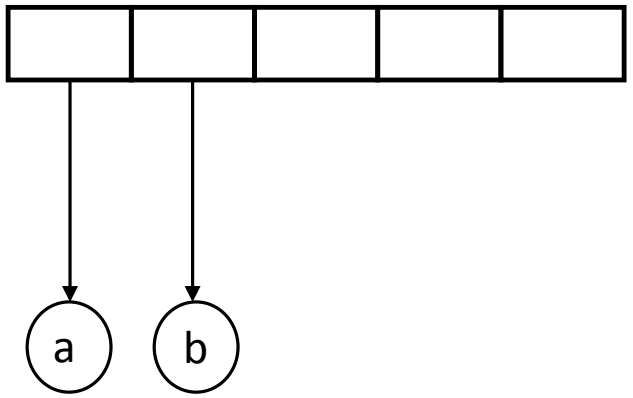
- PREORDER : node left right
  - Example : + + a * b c * + * d e f g

CONSTRUCTING AN EXPRESSION TREE

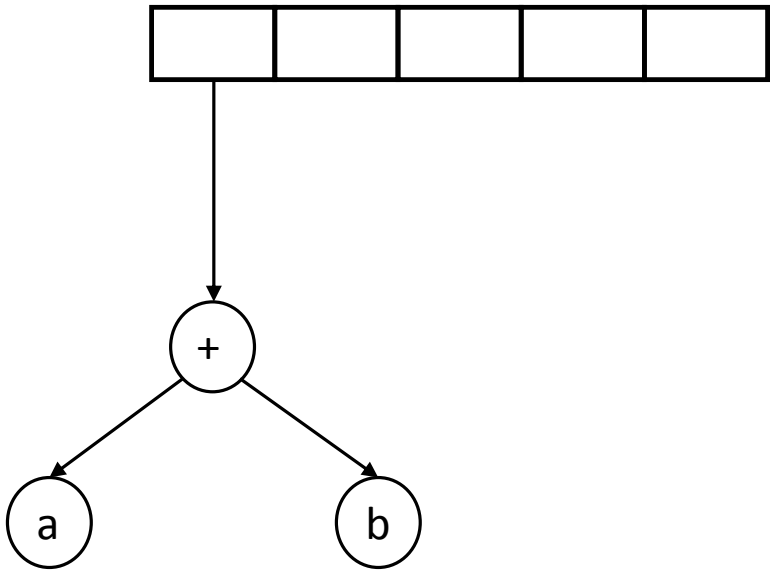
- An algorithm to convert an postfix expression into an expression tree.
- Read one symbol at a time.
  - If the symbol is an operand push into the stack.
  - If the symbol is an operator we pop two trees T_1 and T_2 from the stack to form a new tree whose root is the operator and left and right children are T_2 and T_1 respectively.
  - The new tree is then pushed on to the stack.

Example : a b + c d e + * *

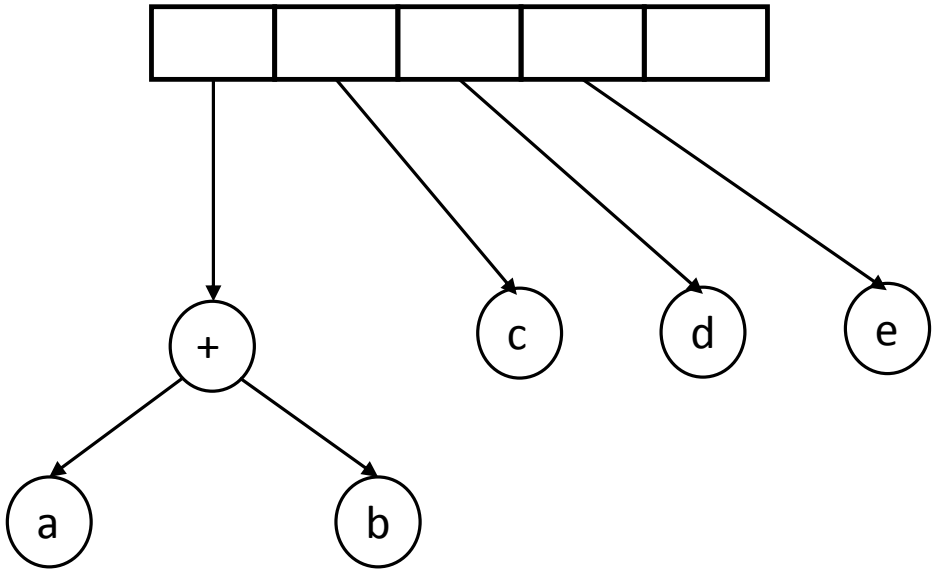
The first two symbols are operands, so we create one-node trees and push them onto the stack



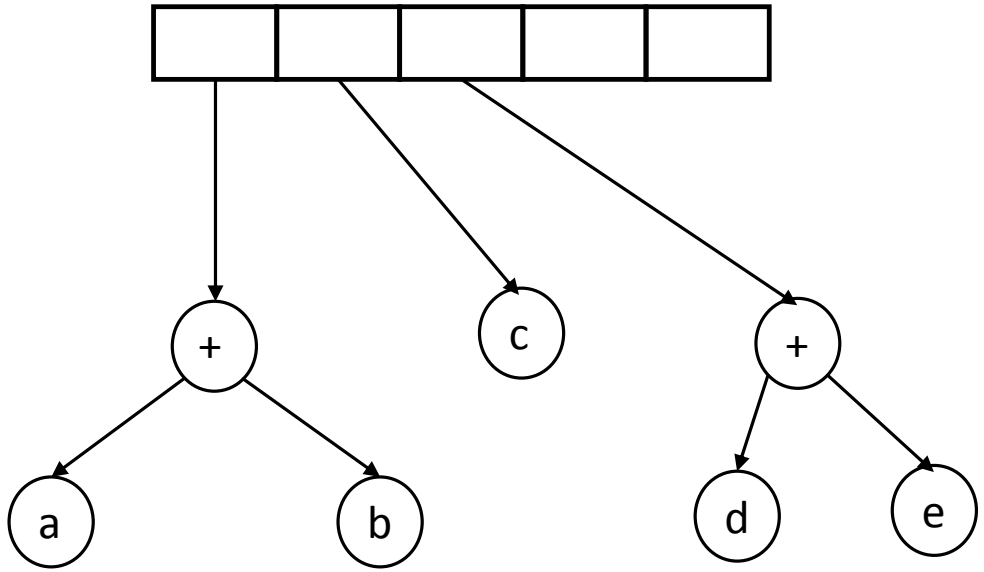
A ‘+’ is read, so two trees are popped
a new tree is formed and pushed
onto the stack



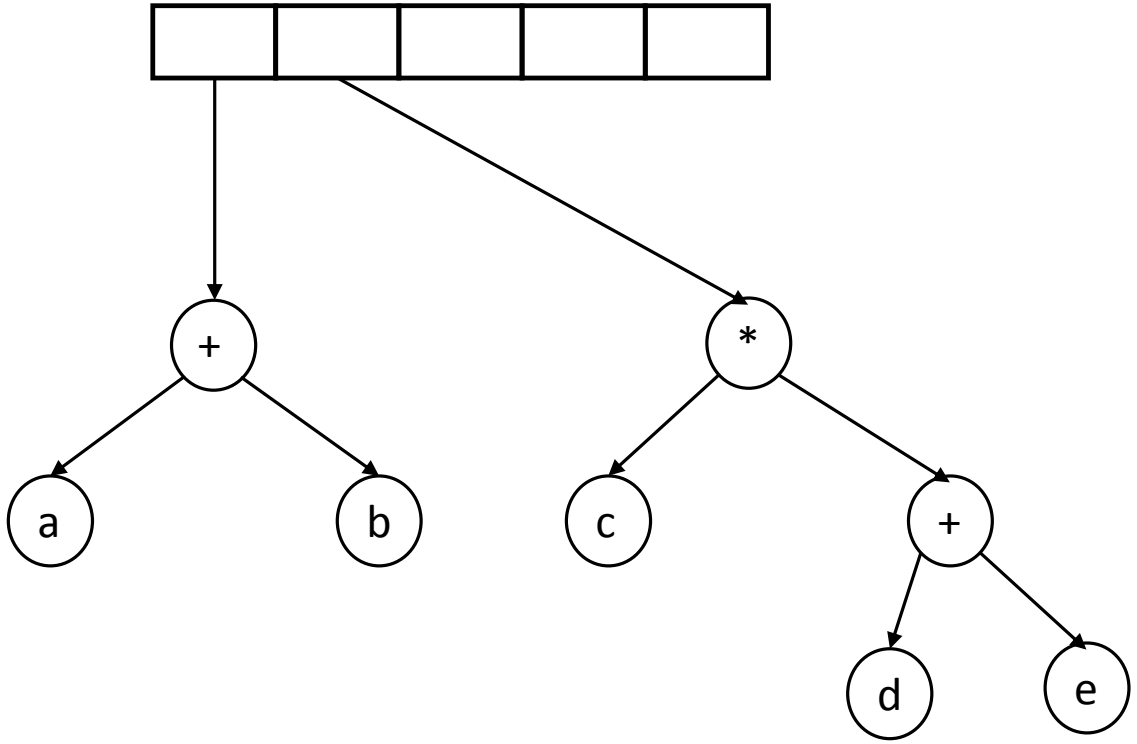
c d e are read and corresponding one node tree is created and pushed onto the stack



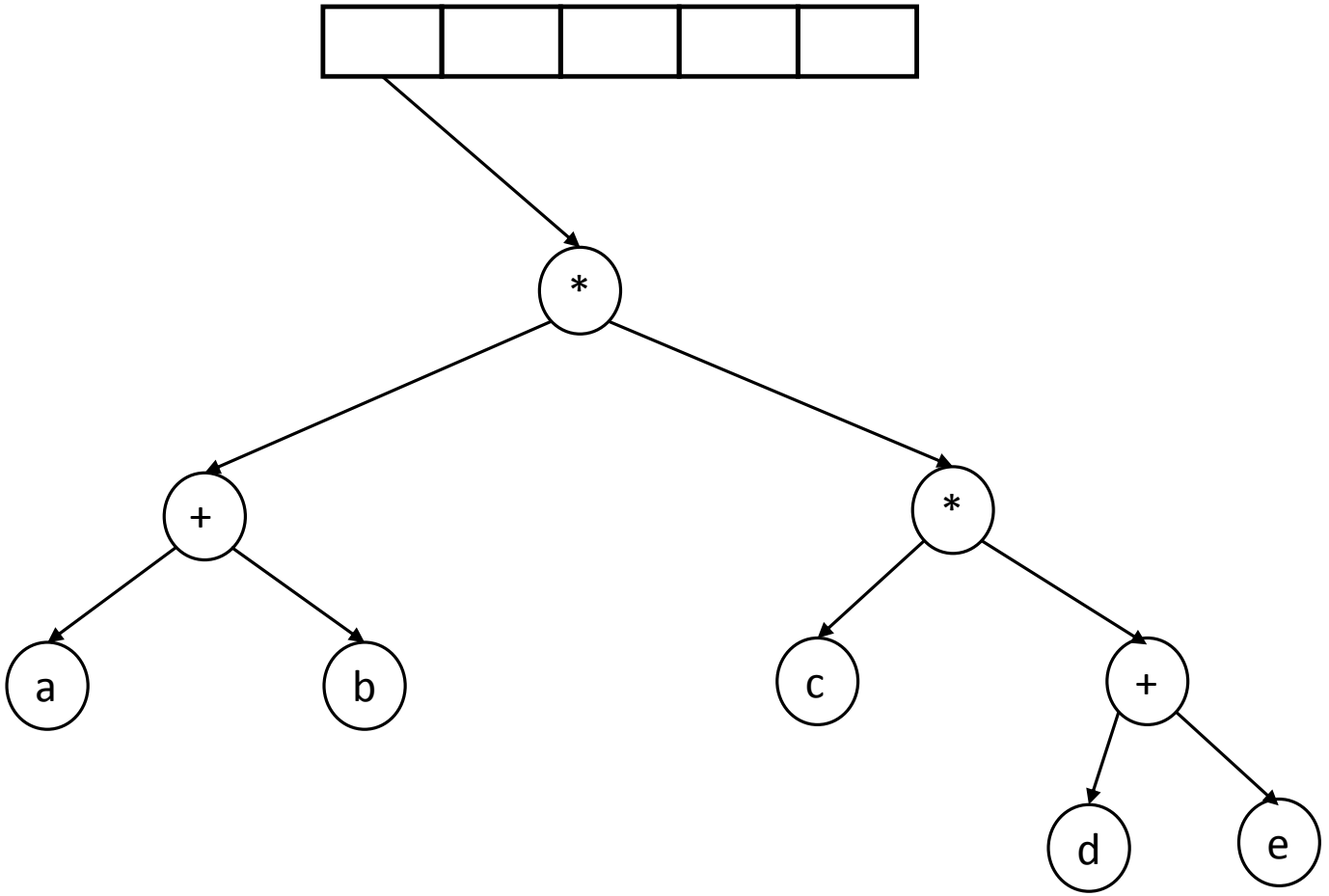
‘+’ is read so two trees are merged



‘*’ is read so two trees are merged



Finally, the last symbol is read, two trees are merged and the final tree is left on the stack.



THE SEARCH TREE ADT

- The property that makes an binary try into a binary search tree is that for every node X,
  - the value of all the items in its left subtree are smaller than X
  - the value of all the items in its right subtree are larger than X

Figure 4.15 Two binary trees (only the left tree is a search tree)

IMPLEMENTATION

```
1       private static class BinaryNode<AnyType>
2       {
3           // Constructors
4           BinaryNode( AnyType theElement )
5             { this( theElement, null, null ); }
6
7           BinaryNode( AnyType theElement, BinaryNode<AnyType> lt, BinaryNode<AnyType> rt )
8             { element  = theElement; left = lt; right = rt; }
9
10          AnyType element;            // The data in the node
11          BinaryNode<AnyType> left;   // Left child
12          BinaryNode<AnyType> right;  // Right child
13      }
```

Figure 4.16 The BinaryNode class

```
1    public class BinarySearchTree<AnyType extends Comparable<? super AnyType>>
2    {
3        private static class BinaryNode<AnyType>
4          { /* Figure 4.16 */ }
5
6        private BinaryNode<AnyType> root;
7
8        public BinarySearchTree( )
9          { root = null; }
10
11       public void makeEmpty( )
12         { root = null; }
13       public boolean isEmpty( )
14         { return root == null; }
15
16       public boolean contains( AnyType x )
17         { return contains( x, root ); }
18       public AnyType findMin( )
19         { if( isEmpty( ) ) throw new UnderflowException( );
20           return findMin( root ).element;
21         }
22       public AnyType findMax( )
23         { if( isEmpty( ) ) throw new UnderflowException( );
24           return findMax( root ).element;
25         }
26       public void insert( AnyType x )
27         { root = insert( x, root ); }
28       public void remove( AnyType x )
29         { root = remove( x, root ); }
30       public void printTree( )
31         { /* Figure 4.56 */ }
32
33       private boolean contains( AnyType x, BinaryNode<AnyType> t )
34         { /* Figure 4.18 */ }
35       private BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
36         { /* Figure 4.20 */ }
37       private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
38         { /* Figure 4.20 */ }
39
40       private BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )
41         { /* Figure 4.22 */ }
42       private BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
43         { /* Figure 4.25 */ }
44       private void printTree( BinaryNode<AnyType> t )
45         { /* Figure 4.56 */ }
46   }
```

Figure 4.17 Binary search tree class skeleton

METHOD : *contains*

```
1        /**
2         * Internal method to find an item in a subtree.
3         * @param x is item to search for.
4         * @param t the node that roots the subtree.
5         * @return true if the item is found; false otherwise.
6         */
7        private boolean contains( AnyType x, BinaryNode<AnyType> t )
8        {
9            if( t == null )
10               return false;
11
12           int compareResult = x.compareTo( t.element );
13
14           if( compareResult < 0 )
15               return contains( x, t.left );
16           else if( compareResult > 0 )
17               return contains( x, t.right );
18           else
19               return true;    // Match
20       }
```

Figure 4.18 contains operation for binary search trees

METHODS : *findMin* and *findMax*

- To perform a findMin, start at the root and go left as long as there is a left child. The stopping point is the smallest element.
- *findMin* is done recursively and *findMax* is done using a while loop.

```
1       /**
2        * Internal method to find the smallest item in a subtree.
3        * @param t the node that roots the subtree.
4        * @return node containing the smallest item.
5        */
6       private BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
7       {
8           if( t == null )
9               return null;
10          else if( t.left == null )
11              return t;
12          return findMin( t.left );
13      }
14
15      /**
16       * Internal method to find the largest item in a subtree.
17       * @param t the node that roots the subtree.
18       * @return node containing the largest item.
19       */
20      private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
21      {
22          if( t != null )
23              while( t.right != null )
24                  t = t.right;
25
26          return t;
27      }
```

Figure 4.20 Recursive implementation of `findMin` and nonrecursive implementation of `findMax` for binary search trees

METHOD : *insert*

- To insert X into the tree T, proceed down the tree as you would with a contains. If X is found do nothing.

Figure 4.21 Binary search trees before and after inserting 5

METHOD : *insert*

```
1       /**
2        * Internal method to insert into a subtree.
3        * @param x the item to insert.
4        * @param t the node that roots the subtree.
5        * @return the new root of the subtree.
6        */
7       private BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )
8       {
9           if( t == null )
10              return new BinaryNode<>( x, null, null );
11
12          int compareResult = x.compareTo( t.element );
13
14          if( compareResult < 0 )
15              t.left = insert( x, t.left );
16          else if( compareResult > 0 )
17              t.right = insert( x, t.right );
18          else
19              ;  // Duplicate; do nothing
20          return t;
21      }
```

Figure 4.22 Insertion into a binary search tree

- Deletion is hard.
  - if it is a leaf it can be deleted immediately.
  - if it has one child, the node can be deleted after its parent adjusts a link to bypass the node.
  - if it has two children, replace the node with smallest data on the right subtree and recursively delete that node.
- Lazy deletion is a popular strategy
  - When an element is to deleted, it is left in the tree and merely marked as being deleted.

METHOD : *remove* with one child

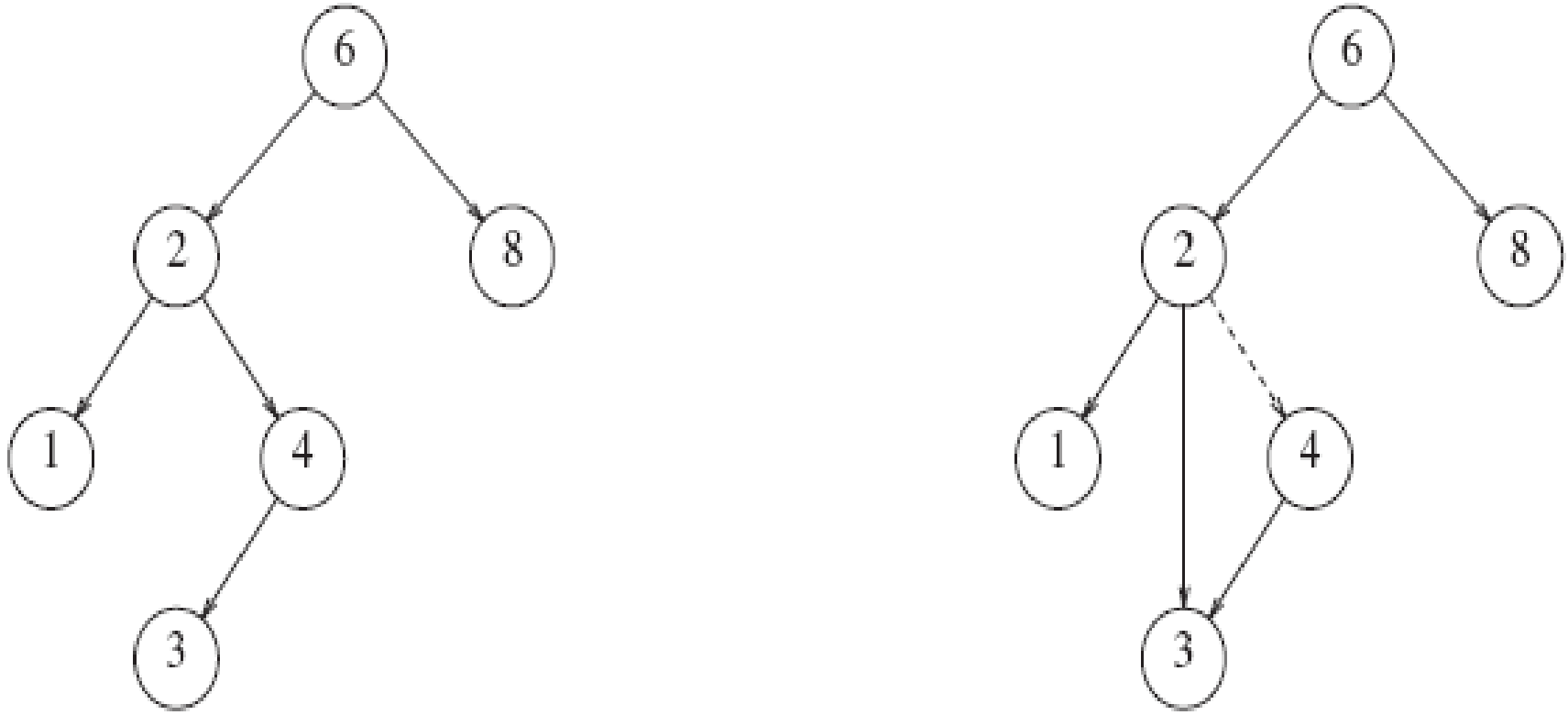


Figure 4.23 Deletion of a node (4) with one child, before and after

METHOD : *remove* with two children

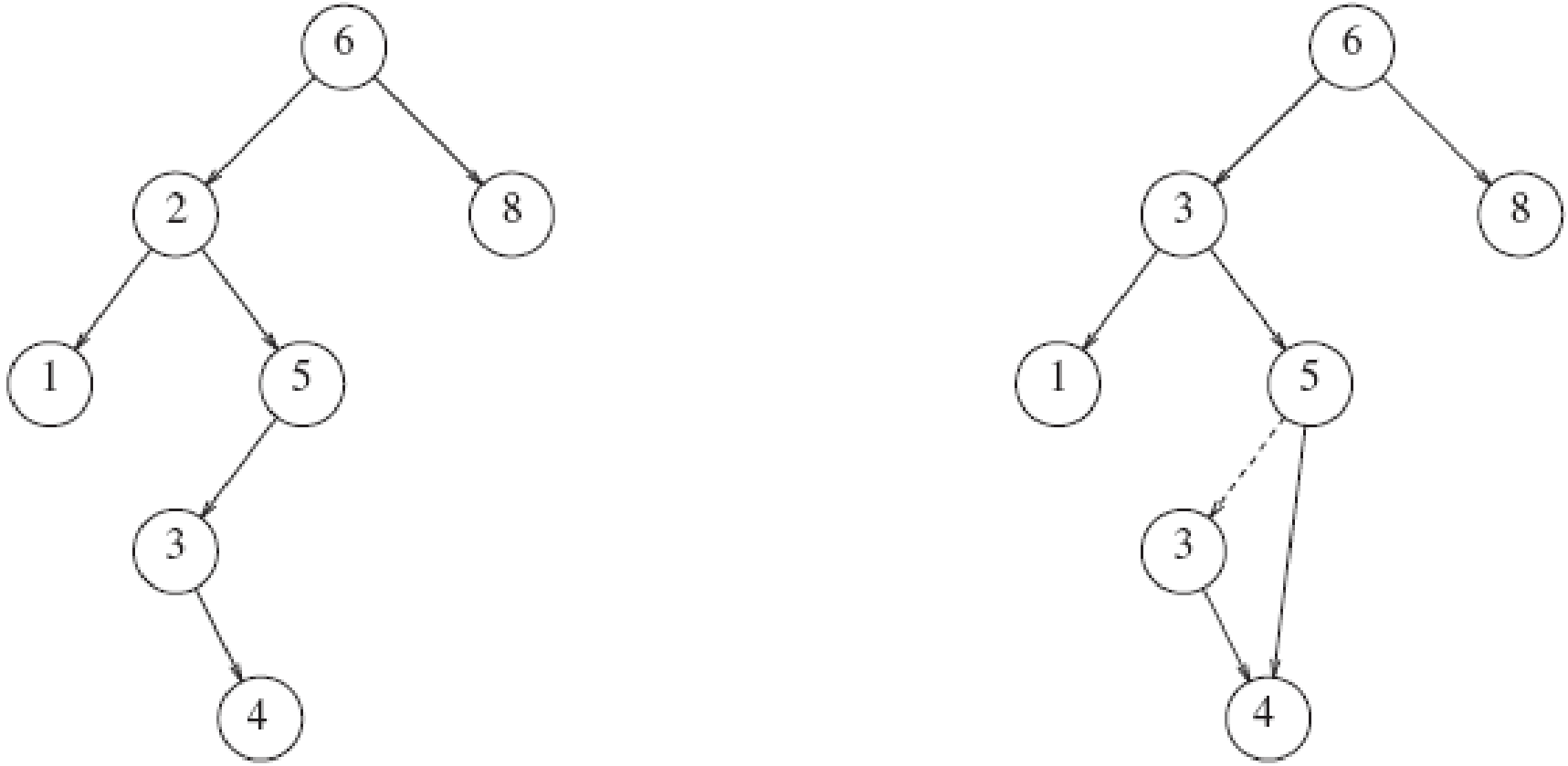


Figure 4.24 Deletion of a node (2) with two children, before and after

```
1         /**
2          * Internal method to remove from a subtree.
3          * @param x the item to remove.
4          * @param t the node that roots the subtree.
5          * @return the new root of the subtree.
6          */
7         private BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
8         {
9             if( t == null )
10                return t;   // Item not found; do nothing
11
12            int compareResult = x.compareTo( t.element );
13
14            if( compareResult < 0 )
15                t.left = remove( x, t.left );
16            else if( compareResult > 0 )
17                t.right = remove( x, t.right );
18            else if( t.left != null && t.right != null ) // Two children
19            {
20                t.element = findMin( t.right ).element;
21                t.right = remove( t.element, t.right );
22            }
23            else
24                t = ( t.left != null ) ? t.left : t.right;
25            return t;
26        }
```

Figure 4.25 Deletion routine for binary search trees

Average – Case Analysis

- The average running time of all the operations discussed in the previous section is O(logN) . This is not entirely true.
- Because deletion tend make left subtree deeper than right

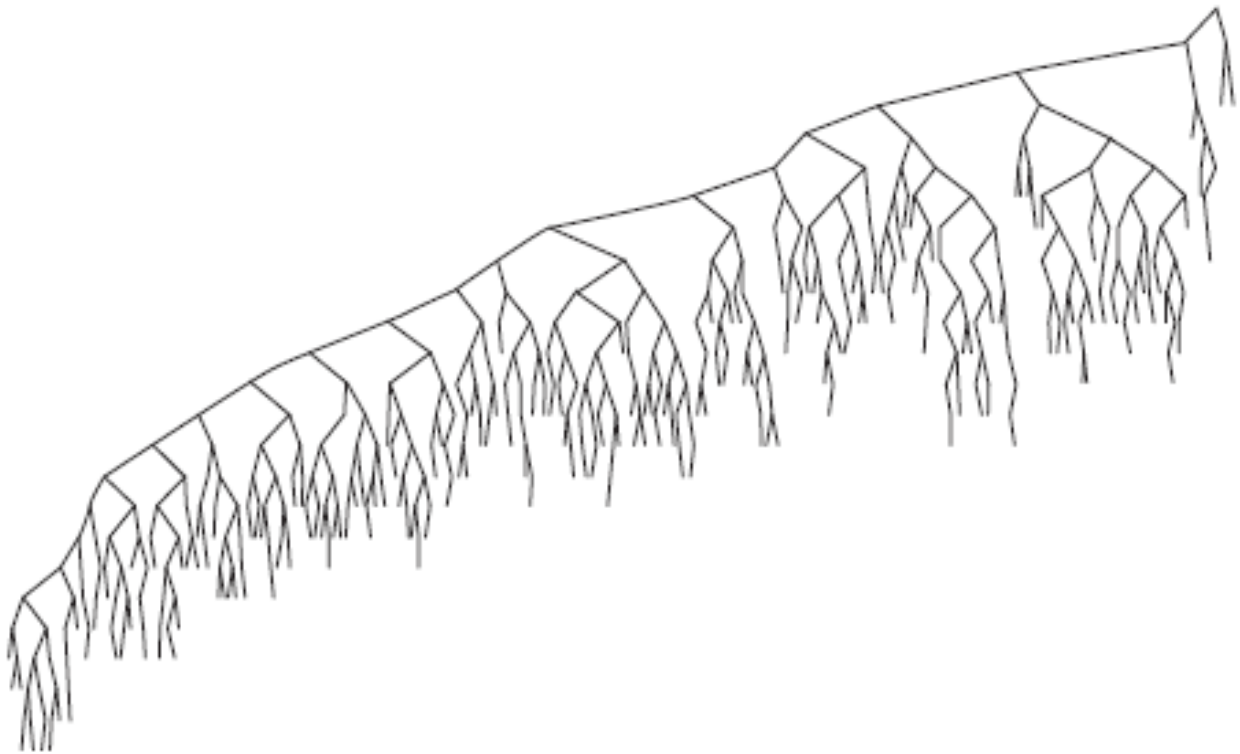


Figure 4.27 Binary search tree after $\Theta(N^2)$ `insert`/`remove` pairs

AVL TREES

- AVL tree is a binary search tree with balance condition which ensures that the depth of the tree is O(logN).

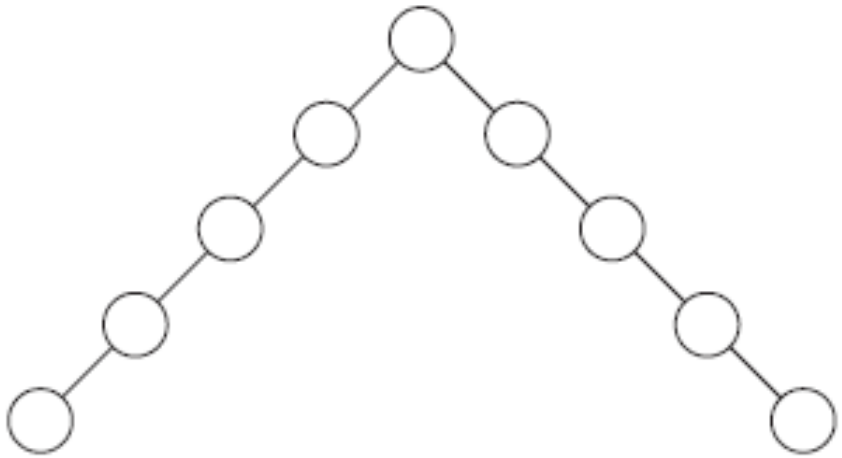


Figure 4.28 A bad binary tree. Requiring balance at the root is not enough

Balance Condition

- An AVL tree is identical to binary search tree, expect that for every node in the tree the height of the left and right subtree can differ by at most 1.

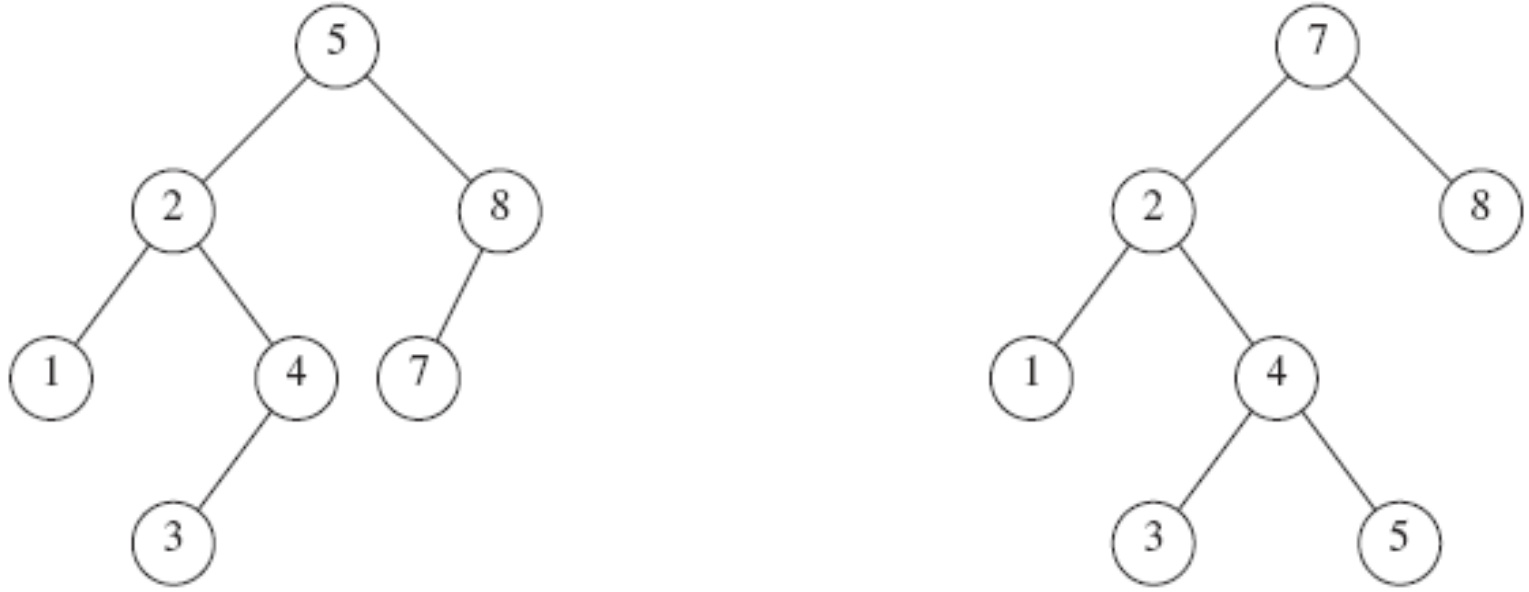


Figure 4.29 Two binary search trees. Only the left tree is AVL

Properties

- Minimum number of nodes for height h is

  S(h) = S(h-1) + S(h-2) +1

  for h = 0 S(h) = 1

  for h =1 S(h) = 2

  Thus all the tree operations can be performed in O(log N) time, expect insert.

Example

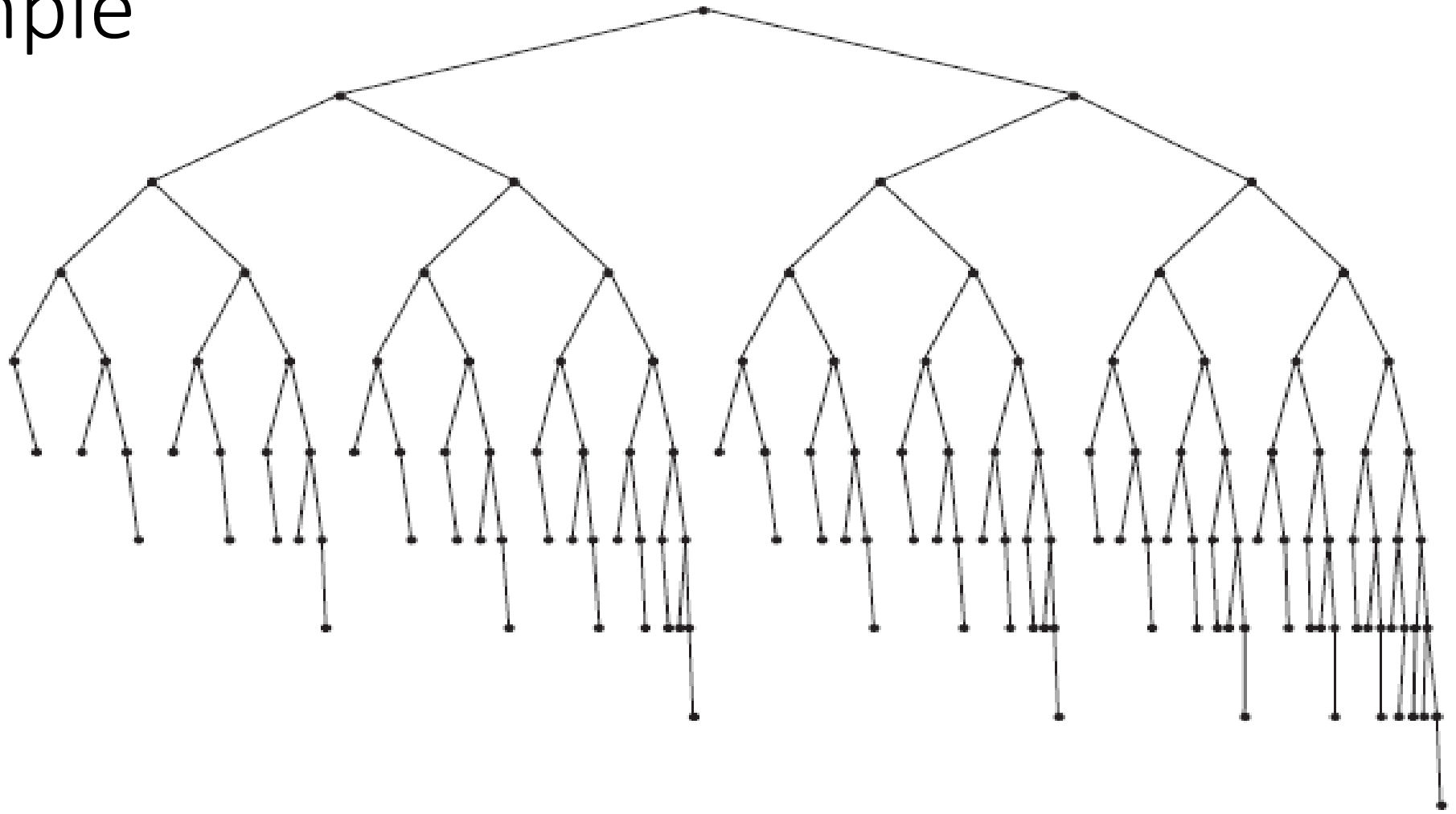


Figure 4.30 Smallest AVL tree of height 9

ROTATION

- After insertion balance has to be restored. Let us call the node that must be rebalanced α. Violation occurs in the following four cases.

1. An insertion into the left subtree of the left child of α.
2. An insertion into the right subtree of the left child of α.
3. An insertion into the left subtree of the right child of α.
4. An insertion into the right subtree of the right child of α.

SINGLE ROTATION

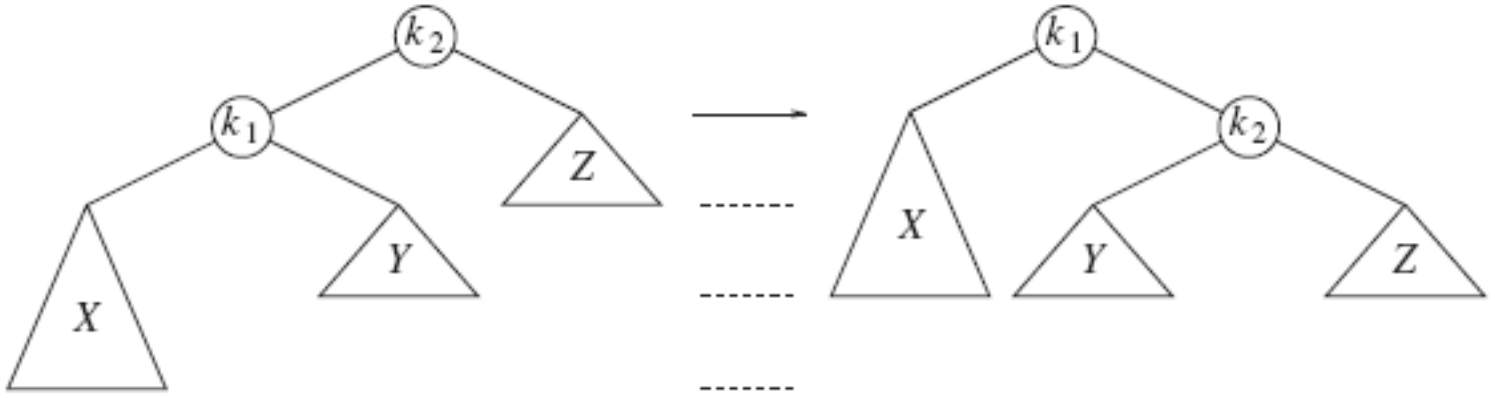


Figure 4.31 Single rotation to fix case 1

Left subtree of the left child scenario. After inserting 6, the tree loses its balance at node 8 (height of right tree is 0 and left tree is 2). Single rotation fixes the problem

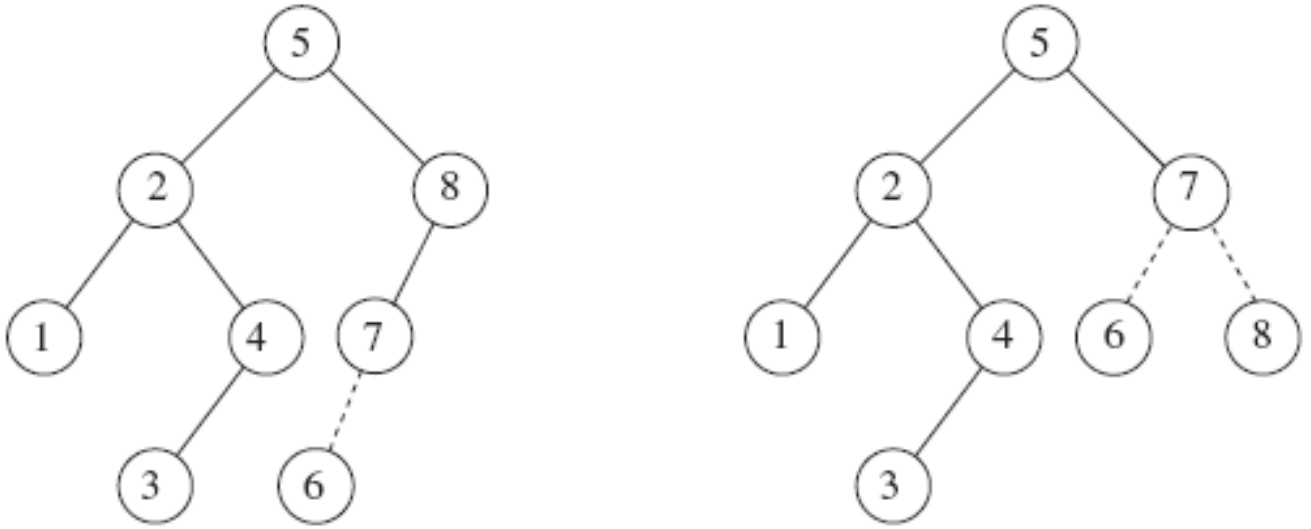


Figure 4.32 AVL property destroyed by insertion of 6, then fixed by a single rotation

Right Subtree Of The Right Child Scenario

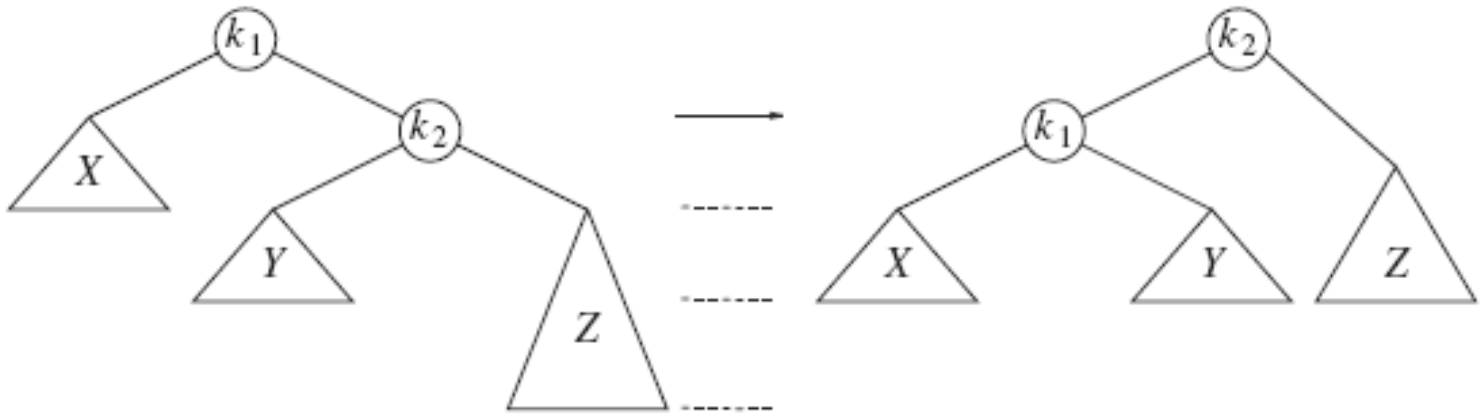
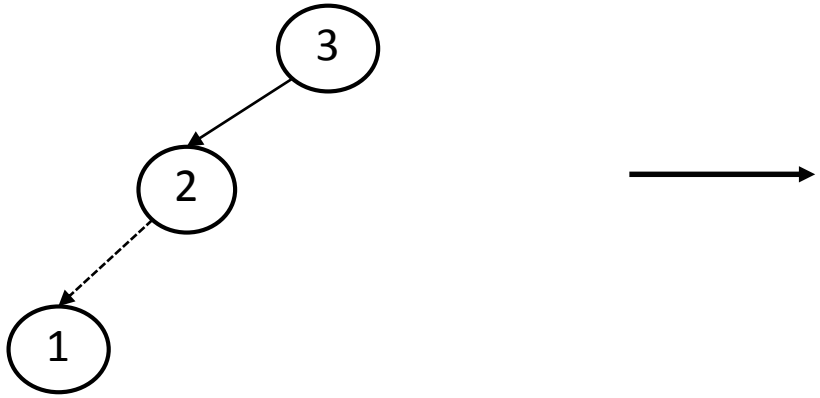
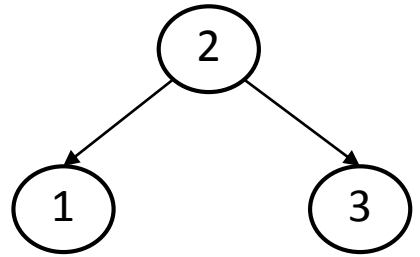
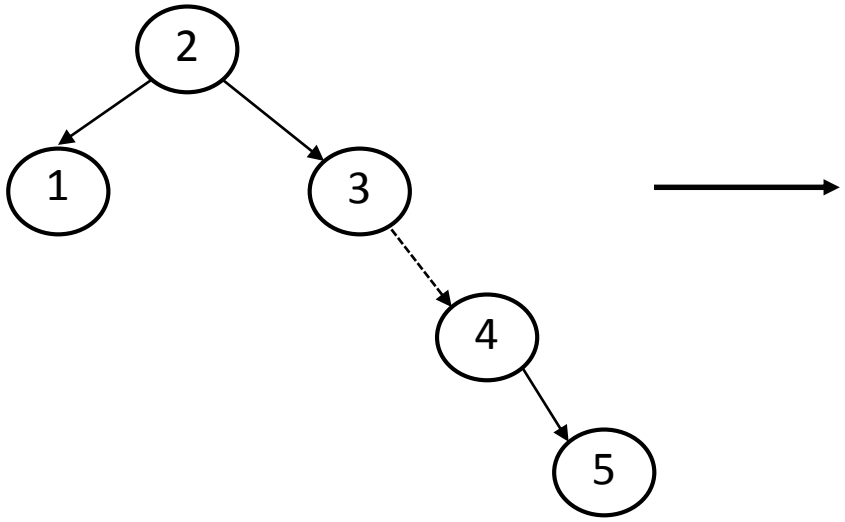


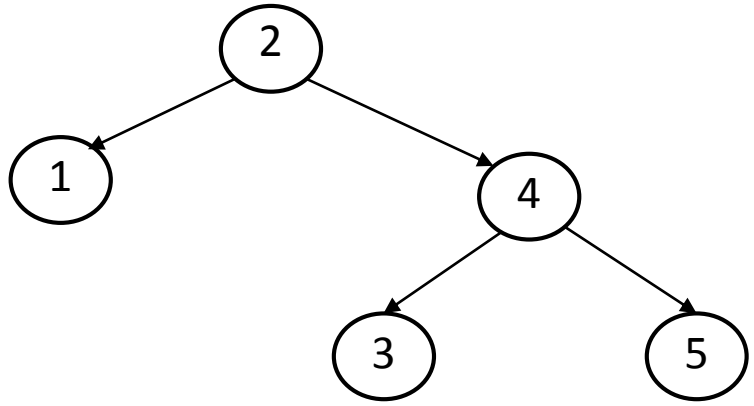
Figure 4.33 Single rotation fixes case 4

Examples: start with an empty AVL tree and insert items 3,2,1, 4 and 5. Try adding 6 and 7









DOUBLE ROTATION

- Single rotation does not work for cases 2 and 3.

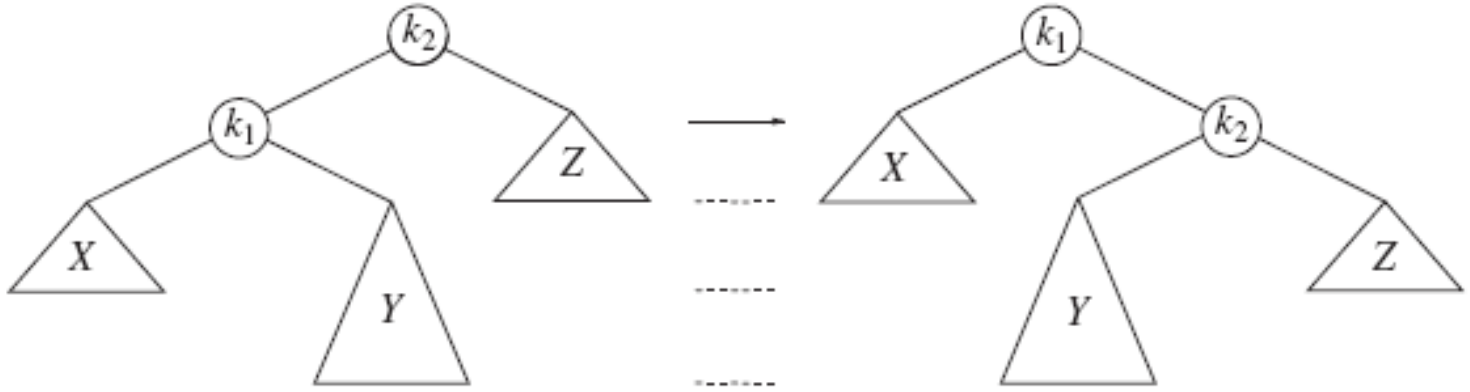


Figure 4.34 Single rotation fails to fix case 2

The fact that subtree Y has had an item inserted into it grantees that it is not is nonempty.

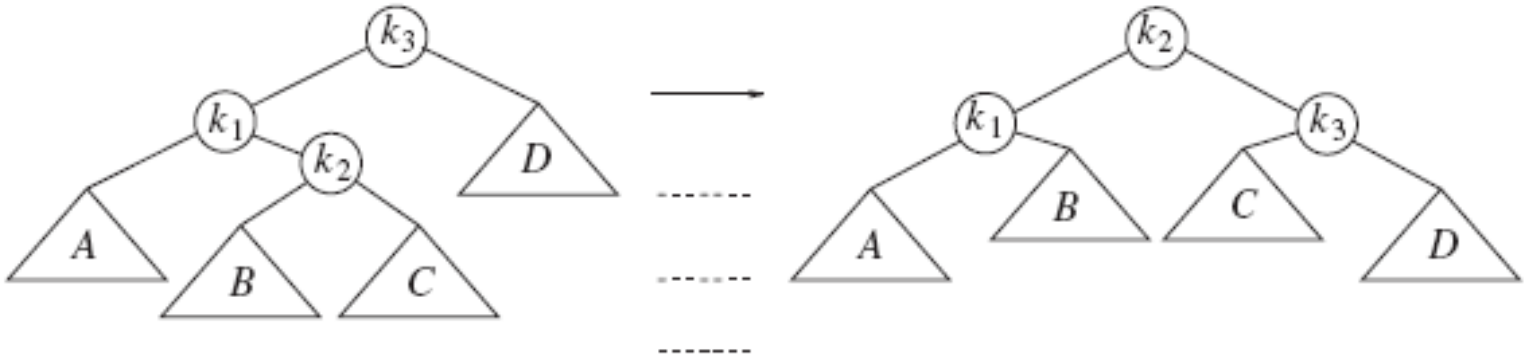


Figure 4.35 Left–right double rotation to fix case 2

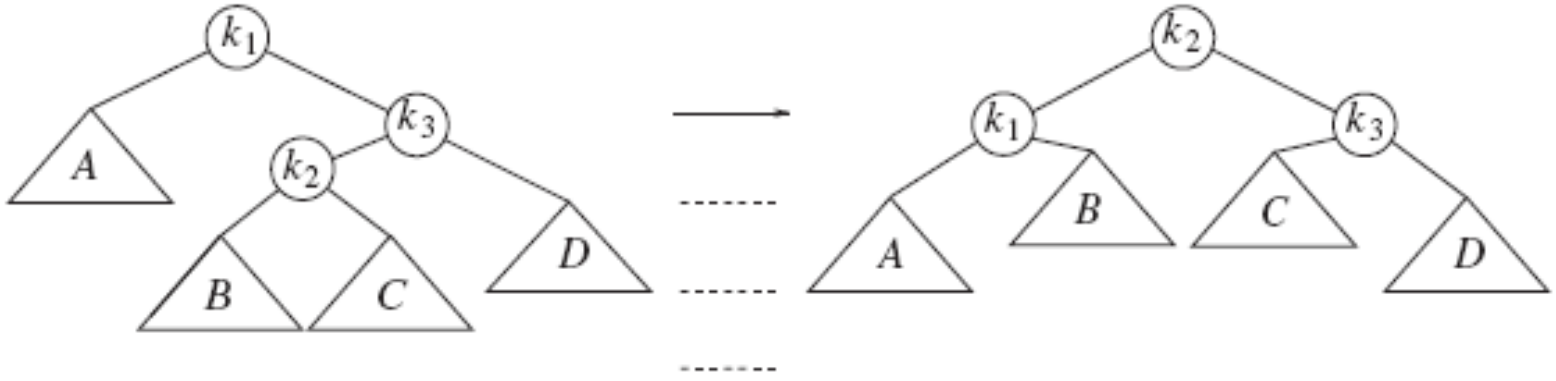
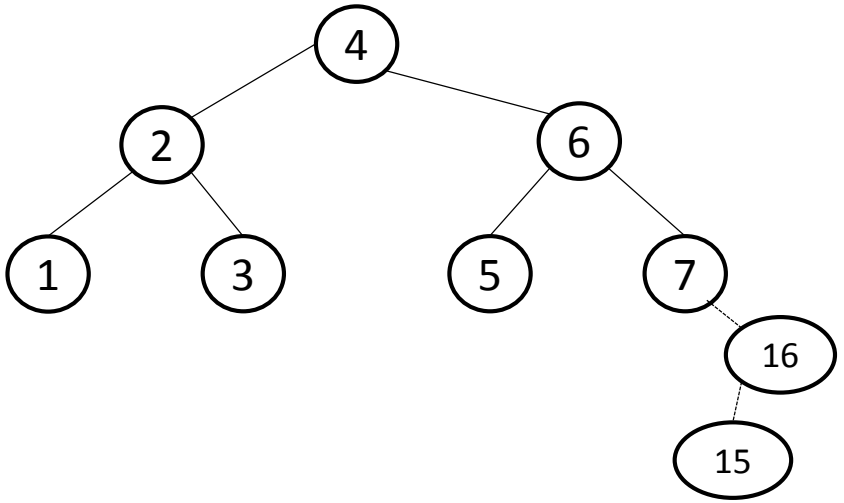
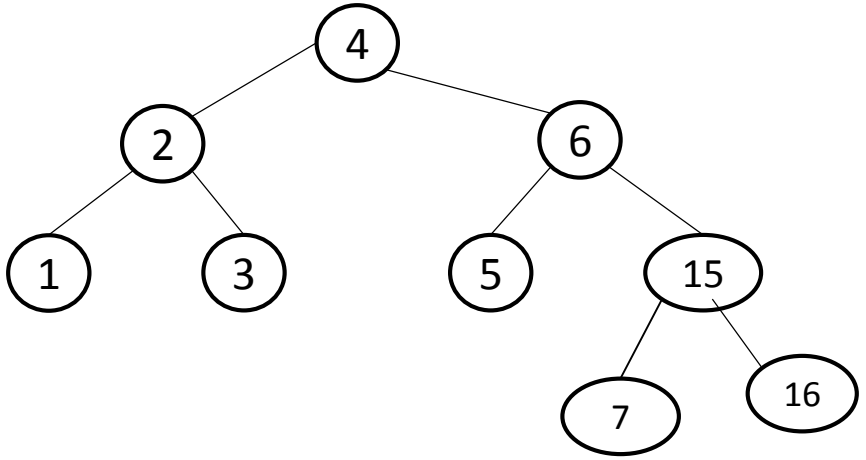
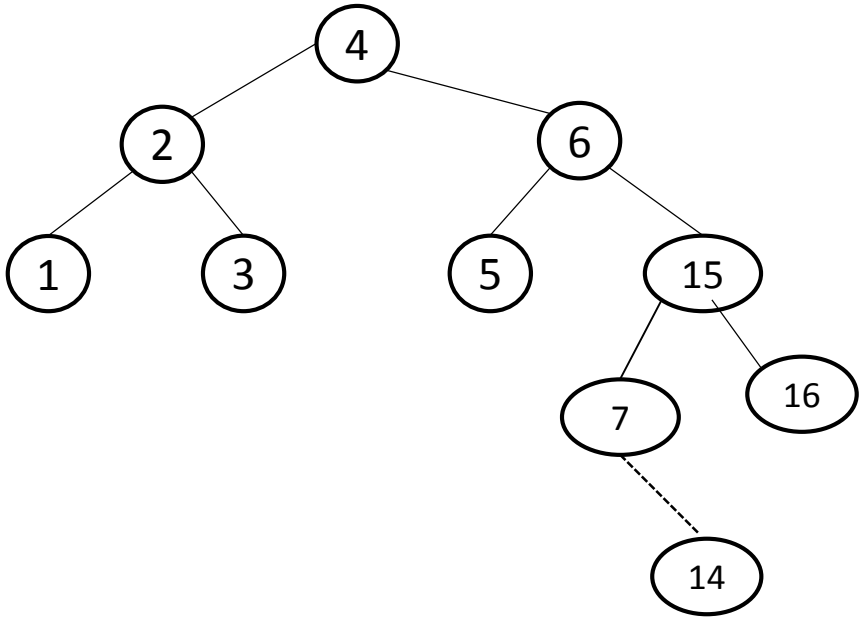


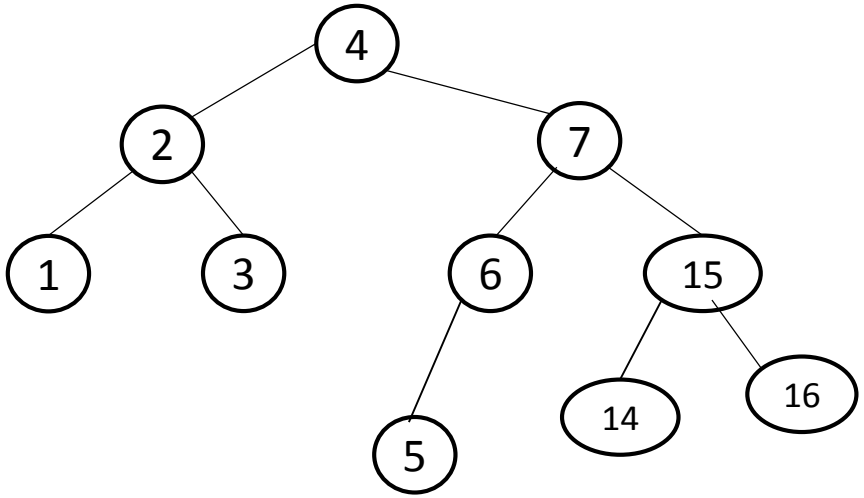
Figure 4.36 Right–left double rotation to fix case 3

Example : insert 16 ,15 and 14

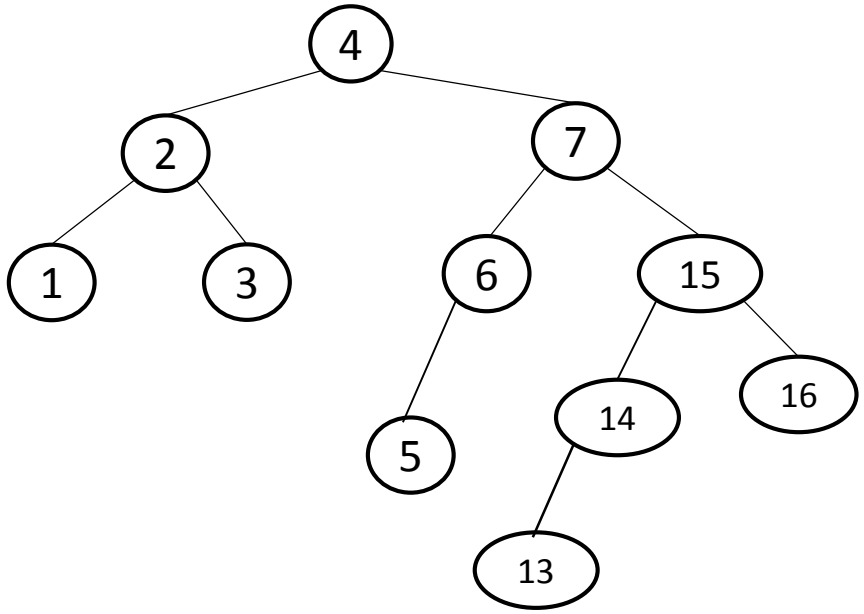


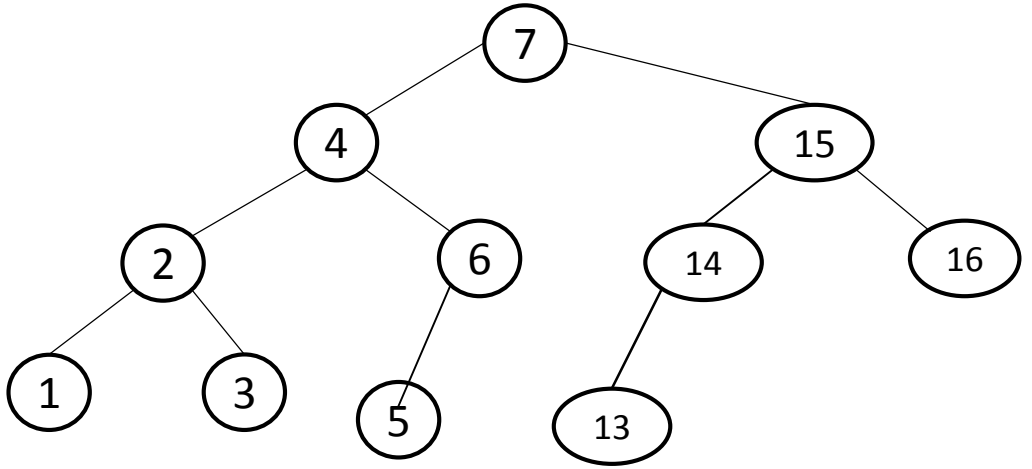




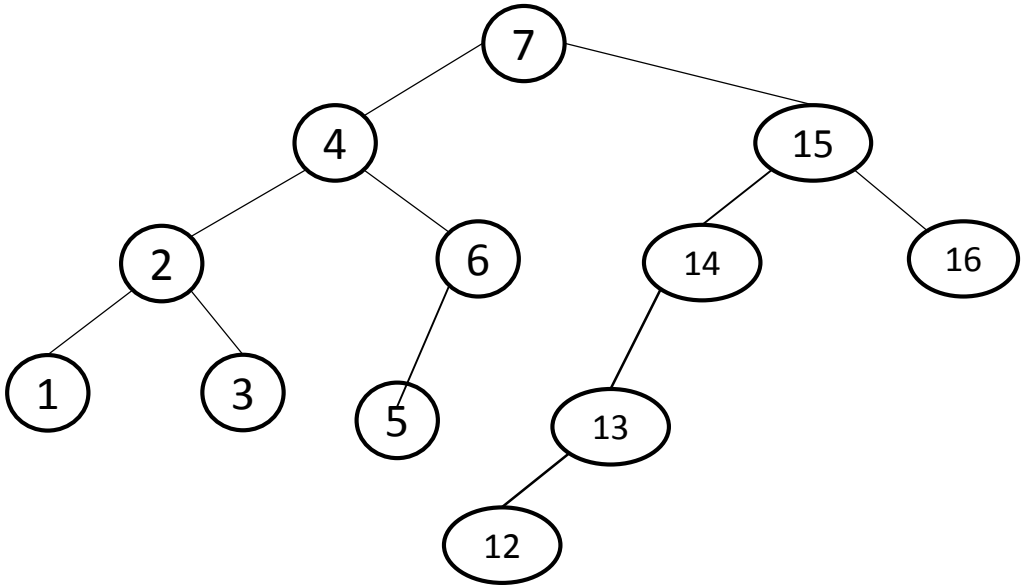


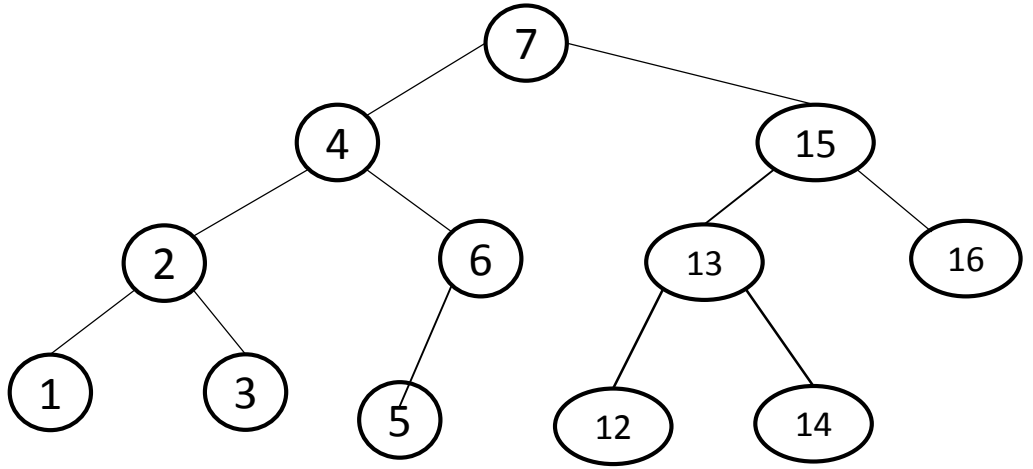
Example : insert 13





Example : insert 12





AVL TREES IMPLEMENTATION

- To insert a new node X, we recursively insert X into appropriate subtree T_{LR}.
  - If the height of T_{LR} does not change, insertion complete.
  - If the height changes, single or double rotation is done depending on X

- Height information is stored at every node to avoid repetitive calculation of balance factors

AVL NODE

```
1       private static class AvlNode<AnyType>
2       {
3           // Constructors
4           AvlNode( AnyType theElement )
5             { this( theElement, null, null ); }
6
7           AvlNode( AnyType theElement, AvlNode<AnyType> lt, AvlNode<AnyType> rt )
8             { element = theElement; left = lt; right = rt; height = 0; }
9
10          AnyType           element;      // The data in the node
11          AvlNode<AnyType>  left;         // Left child
12          AvlNode<AnyType>  right;        // Right child
13          int               height;       // Height
14      }
```

Figure 4.37 Node declaration for AVL trees

METHOD : *height*

```
1       /**
2        * Return the height of node t, or -1, if null.
3        */
4       private int height( AvlNode<AnyType> t )
5       {
6           return t == null ? -1 : t.height;
7       }
```

Figure 4.38 Method to compute height of an AVL node

```
1          /**
2           * Internal method to insert into a subtree.
3           * @param x the item to insert.
4           * @param t the node that roots the subtree.
5           * @return the new root of the subtree.
6           */
7          private AvlNode<AnyType> insert( AnyType x, AvlNode<AnyType> t )
8          {
9              if( t == null )
10                 return new AvlNode<>( x, null, null );
11
12             int compareResult = x.compareTo( t.element );
13
14             if( compareResult < 0 )
15                 t.left = insert( x, t.left );
16             else if( compareResult > 0 )
17                 t.right = insert( x, t.right );
18             else
19                 ;  // Duplicate; do nothing
20             return balance( t );
21         }
22
23         private static final int ALLOWED_IMBALANCE = 1;
24
25         // Assume t is either balanced or within one of being balanced
26         private AvlNode<AnyType> balance( AvlNode<AnyType> t )
27         {
28             if( t == null )
29                 return t;
30
31             if( height( t.left ) - height( t.right ) > ALLOWED_IMBALANCE )
32                 if( height( t.left.left ) >= height( t.left.right ) )
33                     t = rotateWithLeftChild( t );
34                 else
35                     t = doubleWithLeftChild( t );
36             else
37             if( height( t.right ) - height( t.left ) > ALLOWED_IMBALANCE )
38                 if( height( t.right.right ) >= height( t.right.left ) )
39                     t = rotateWithRightChild( t );
40                 else
41                     t = doubleWithRightChild( t );
42
43             t.height = Math.max( height( t.left ), height( t.right ) ) + 1;
44             return t;
45         }
```

Figure 4.39 Insertion into an AVL tree

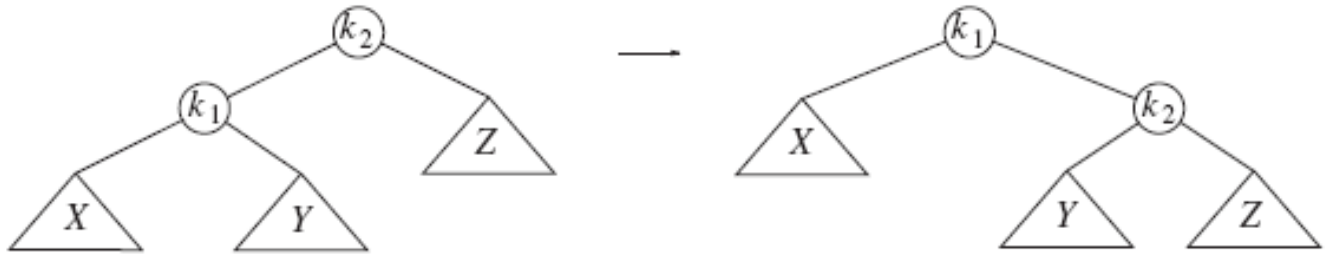


Figure 4.40 Single rotation

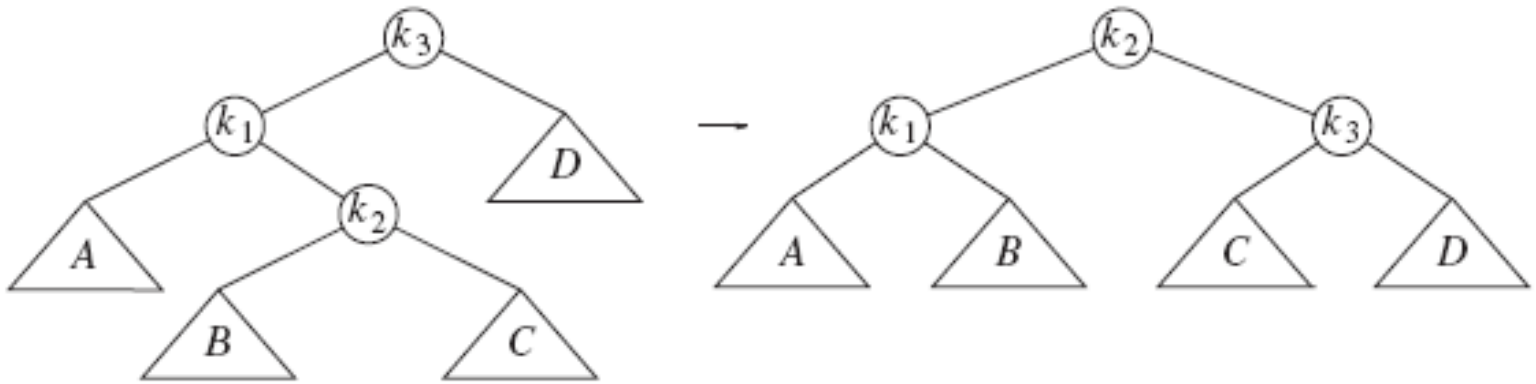


Figure 4.42 Double rotation

SINGLE ROTATION

```
1       /**
2        * Rotate binary tree node with left child.
3        * For AVL trees, this is a single rotation for case 1.
4        * Update heights, then return new root.
5        */
6       private AvlNode<AnyType> rotateWithLeftChild( AvlNode<AnyType> k2 )
7       {
8           AvlNode<AnyType> k1 = k2.left;
9           k2.left = k1.right;
10          k1.right = k2;
11          k2.height = Math.max( height( k2.left ), height( k2.right ) ) + 1;
12          k1.height = Math.max( height( k1.left ), k2.height ) + 1;
13          return k1;
14      }
```

Figure 4.41 Routine to perform single rotation

DOUBLE ROTATION

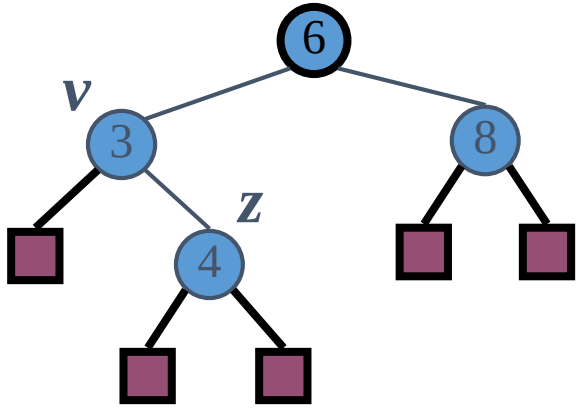
```
1       /**
2        * Double rotate binary tree node: first left child
3        * with its right child; then node k3 with new left child.
4        * For AVL trees, this is a double rotation for case 2.
5        * Update heights, then return new root.
6        */
7       private AvlNode<AnyType> doubleWithLeftChild( AvlNode<AnyType> k3 )
8       {
9           k3.left = rotateWithRightChild( k3.left );
10          return rotateWithLeftChild( k3 );
11      }
```

Figure 4.43 Routine to perform double rotation

```
1         /**
2          * Internal method to remove from a subtree.
3          * @param x the item to remove.
4          * @param t the node that roots the subtree.
5          * @return the new root of the subtree.
6          */
7         private AvlNode<AnyType> remove( AnyType x, AvlNode<AnyType> t )
8         {
9             if( t == null )
10                return t;    // Item not found; do nothing
11
12            int compareResult = x.compareTo( t.element );
13
14            if( compareResult < 0 )
15                t.left = remove( x, t.left );
16            else if( compareResult > 0 )
17                t.right = remove( x, t.right );
18            else if( t.left != null && t.right != null ) // Two children
19            {
20                t.element = findMin( t.right ).element;
21                t.right = remove( t.element, t.right );
22            }
23            else
24                t = ( t.left != null ) ? t.left : t.right;
25            return balance( t );
26        }
```

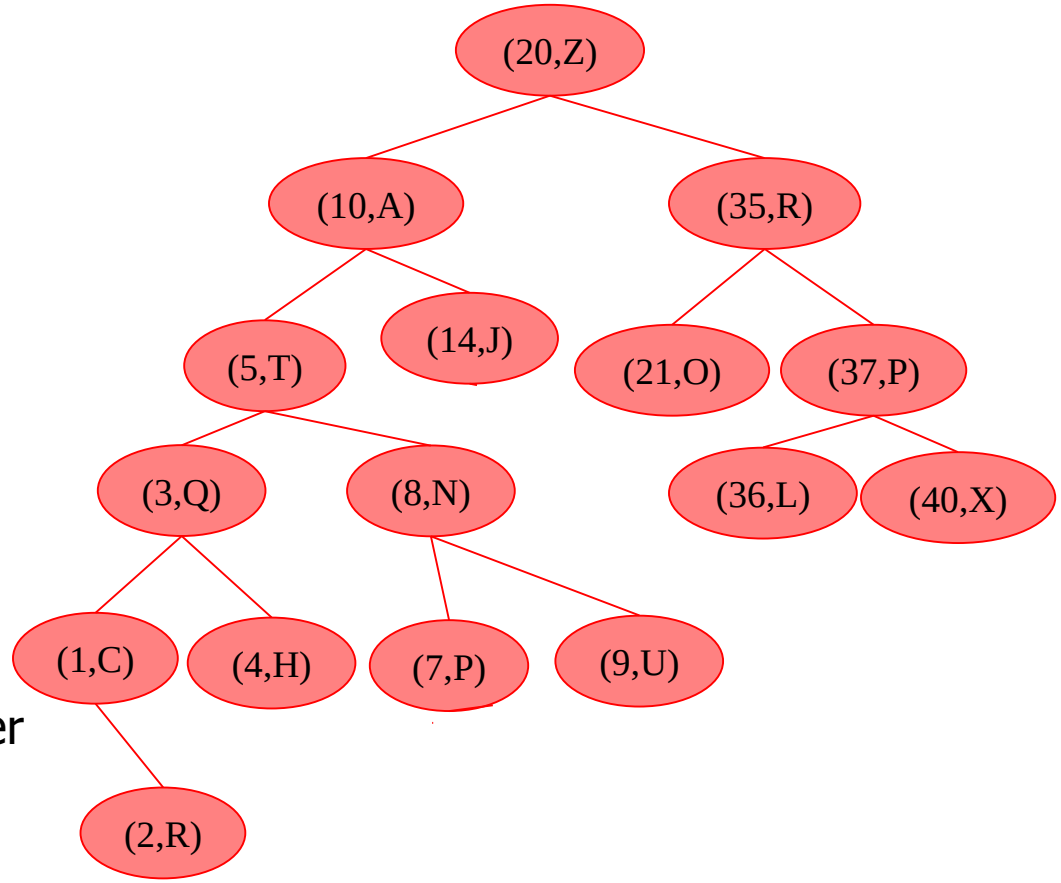
Figure 4.44 Deletion in an AVL tree

Splay Trees



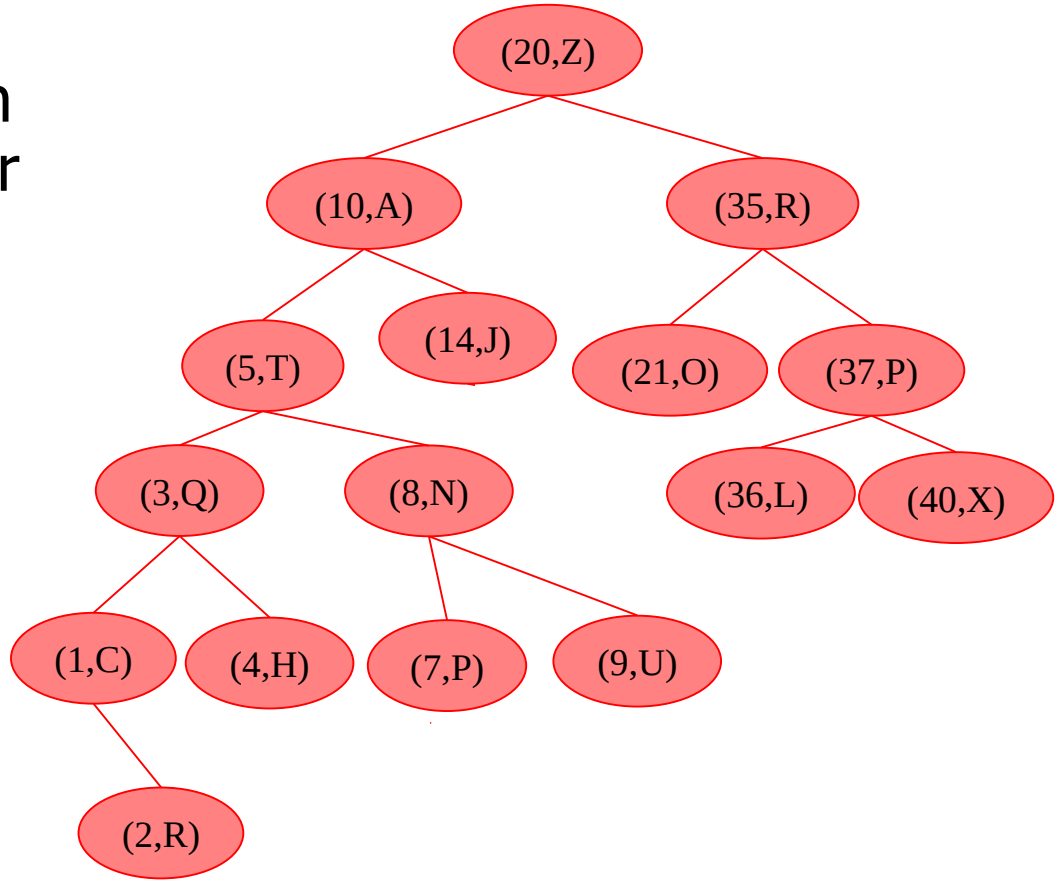
Splay Trees are Binary Search Trees

- BST Rules:
  - entries stored only at internal nodes
  - keys stored at nodes in the left subtree of *v* are less than or equal to the key stored at *v*
  - keys stored at nodes in the right subtree of *v* are greater than or equal to the key stored at *v*
- An inorder traversal will return the keys in order



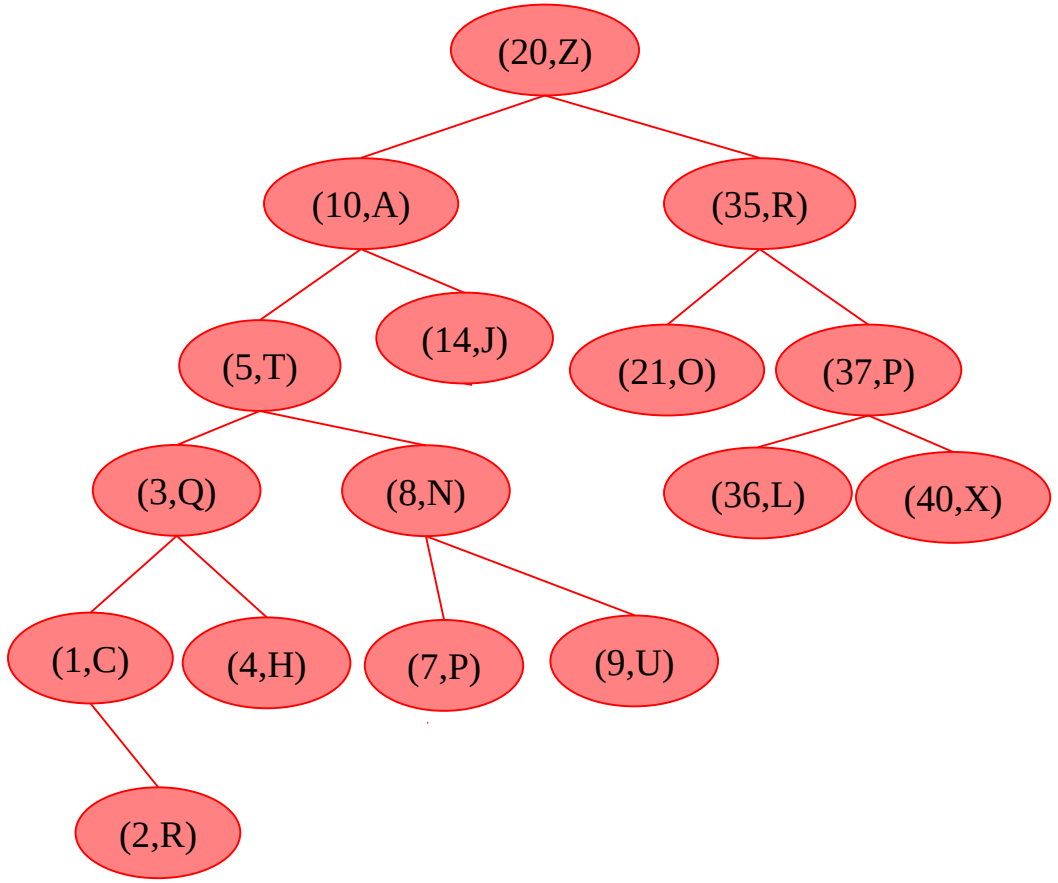
Searching in a Splay Tree: Starts the Same as in a BST

- Search proceeds down the tree to find item or an external node.
- Example: Search for item with key 11.



Example Searching in a BST, continued

- search for key 8, ends at an internal node.

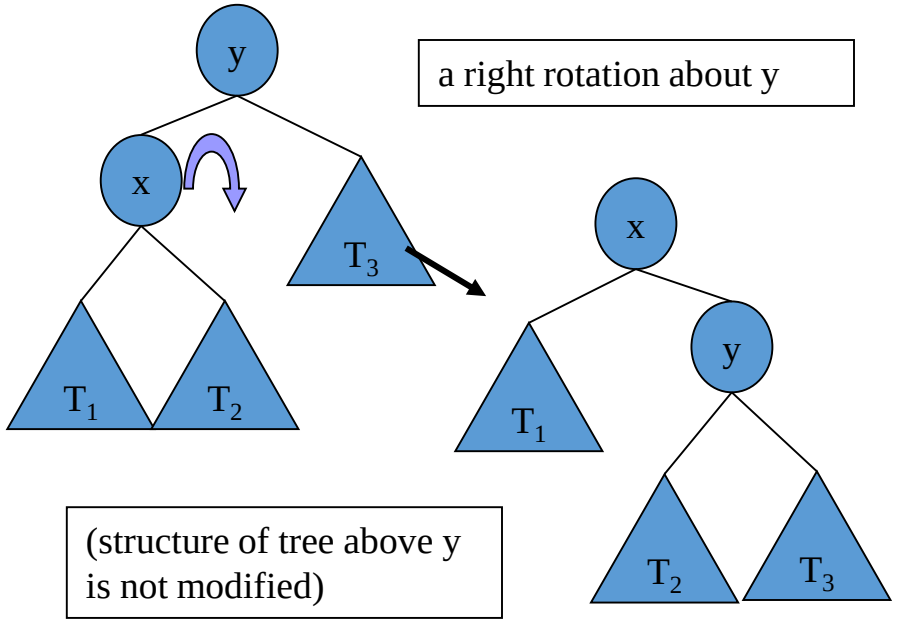


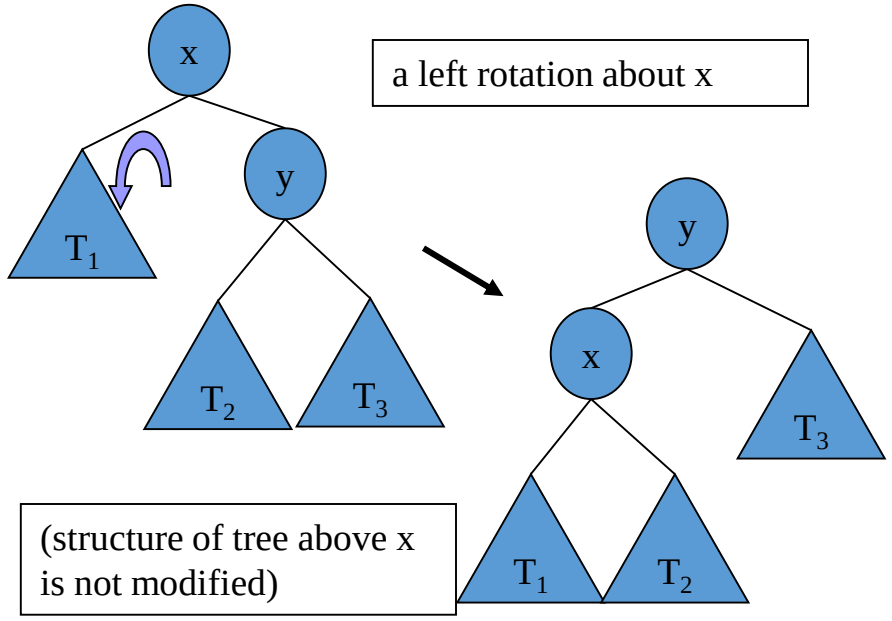
Splay Trees do Rotations after Every Operation (Even Search)

- new operation: ***splay***
  - splaying moves a node to the root using rotations

- right rotation
  - makes the left child *x* of a node *y* into *y*'s parent; *y* becomes the right child of *x*

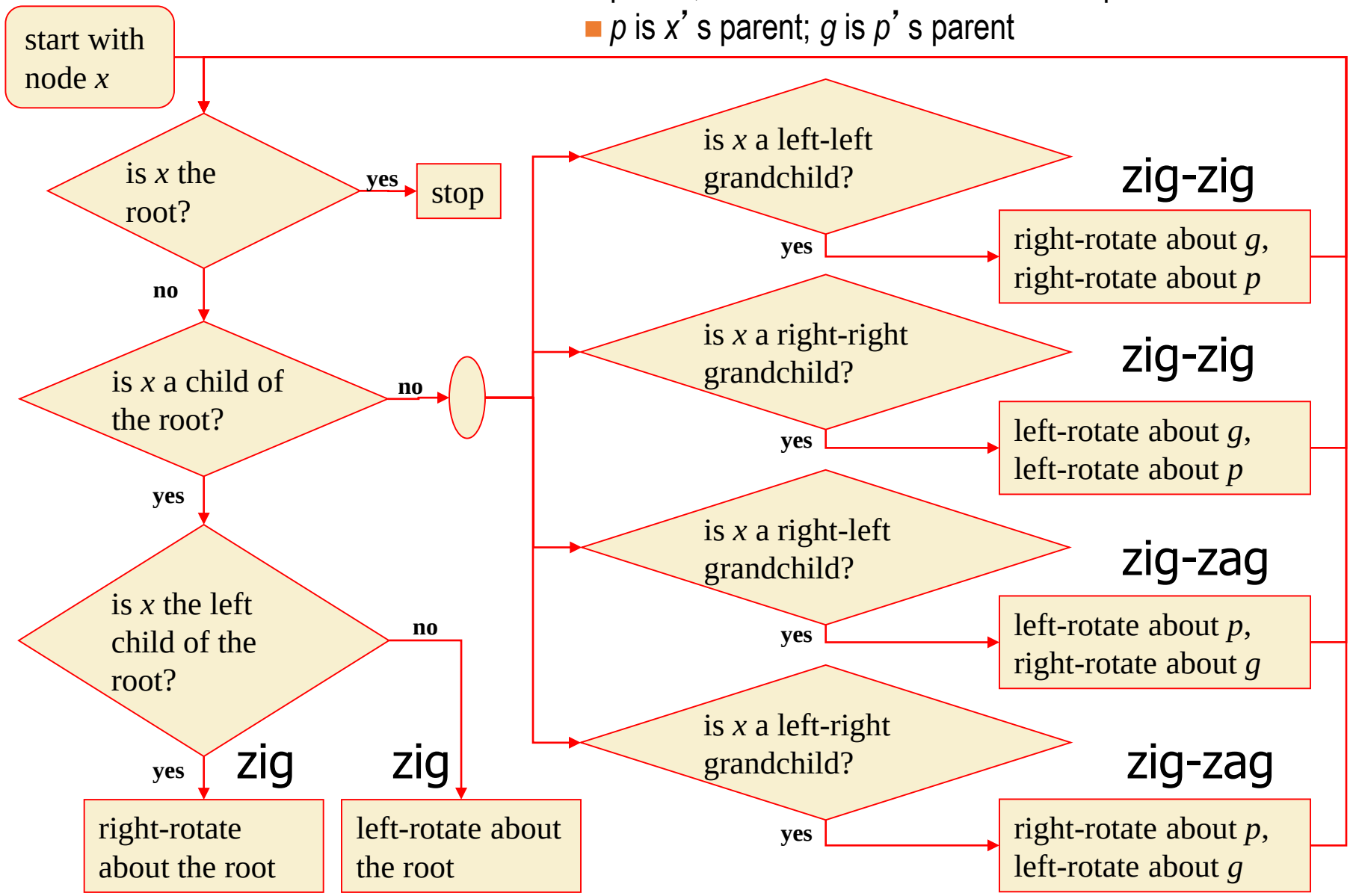
- left rotation
  - makes the right child *y* of a node *x* into *x*'s parent; *x* becomes the left child of *y*



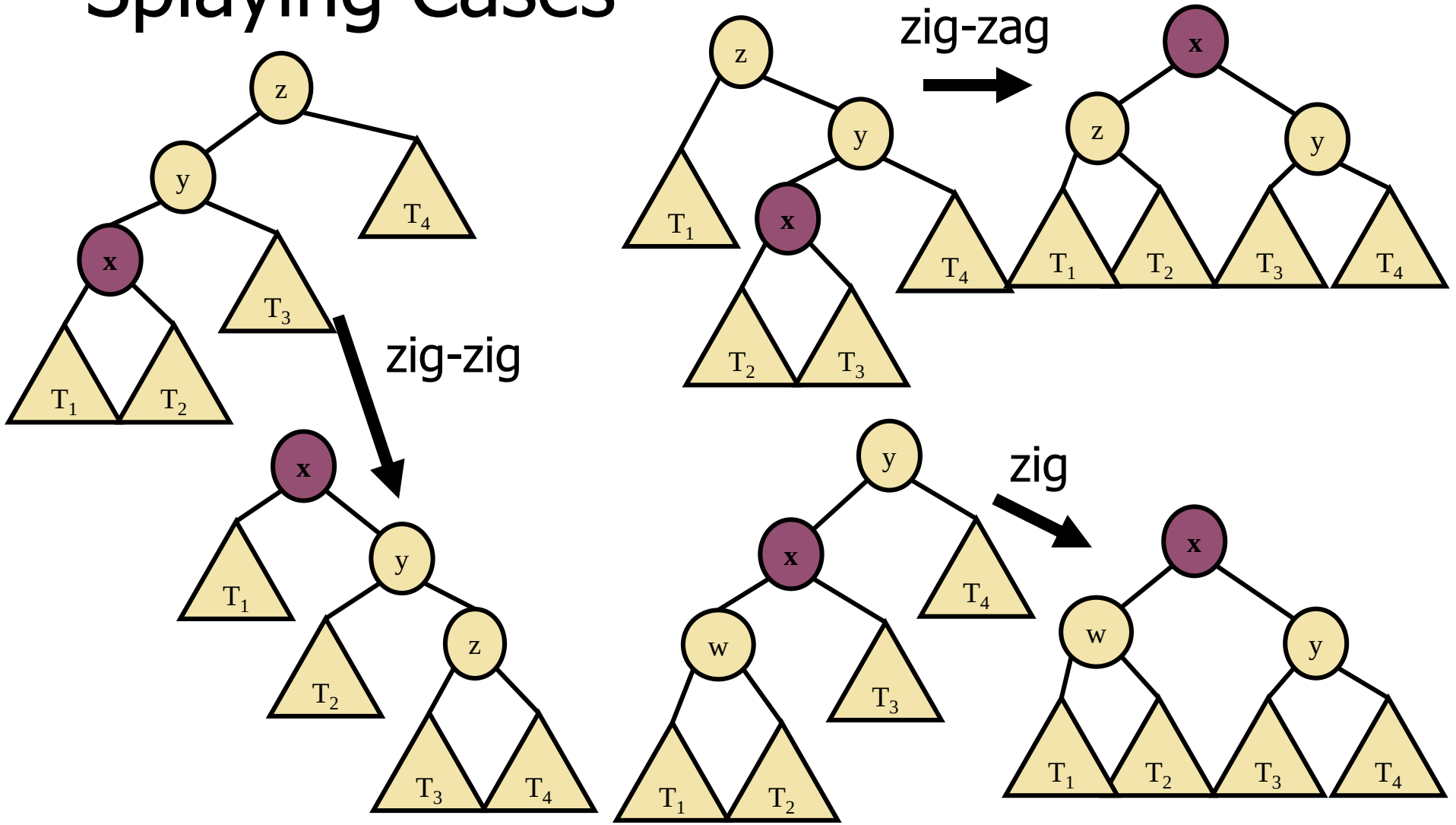


Splaying:

- "*x* is a left-left grandchild" means *x* is a left child of its parent, which is itself a left child of its parent
- *p* is *x*'s parent; *g* is *p*'s parent

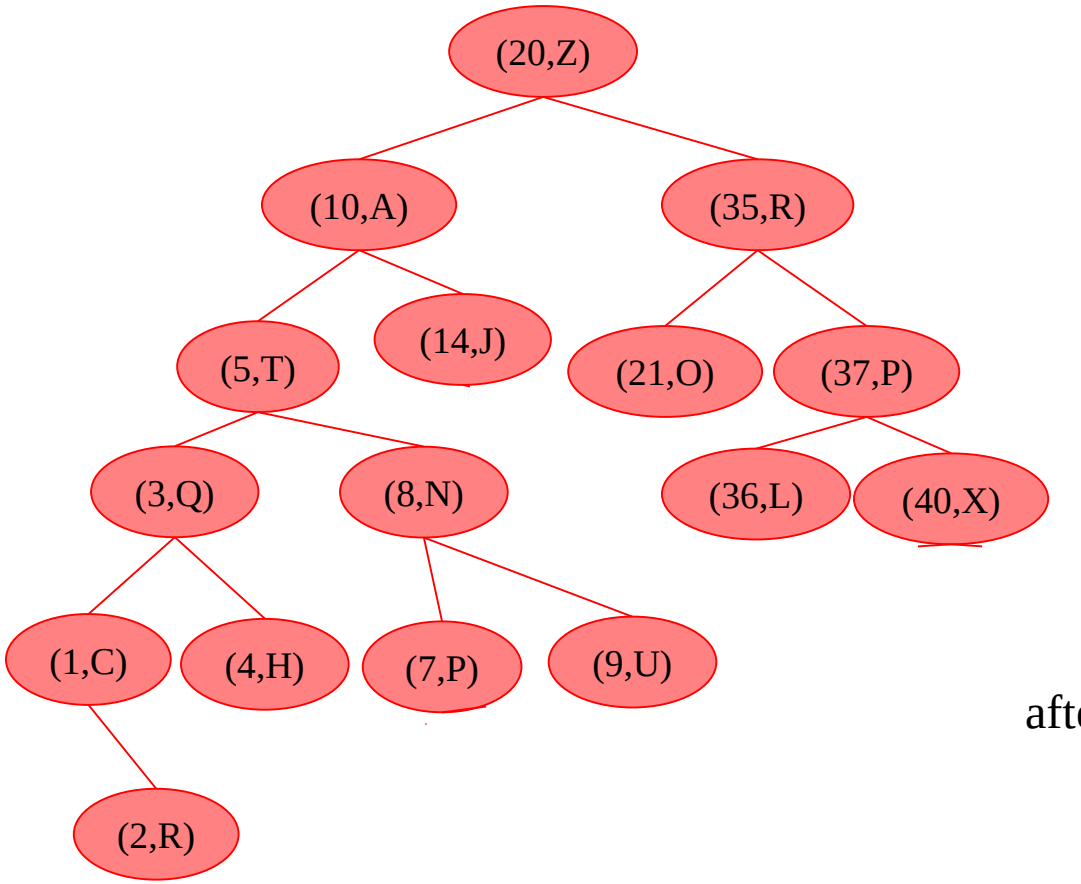


Visualizing the Splaying Cases

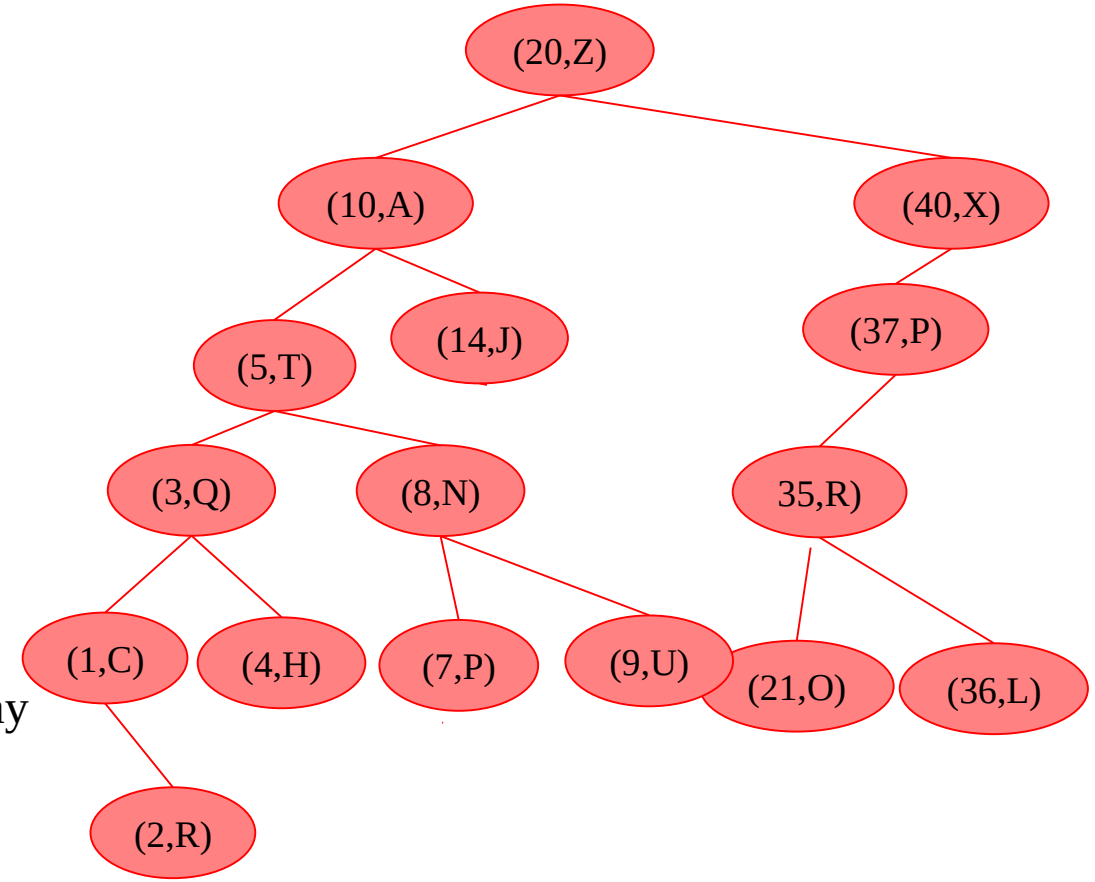


Example Result of Splaying

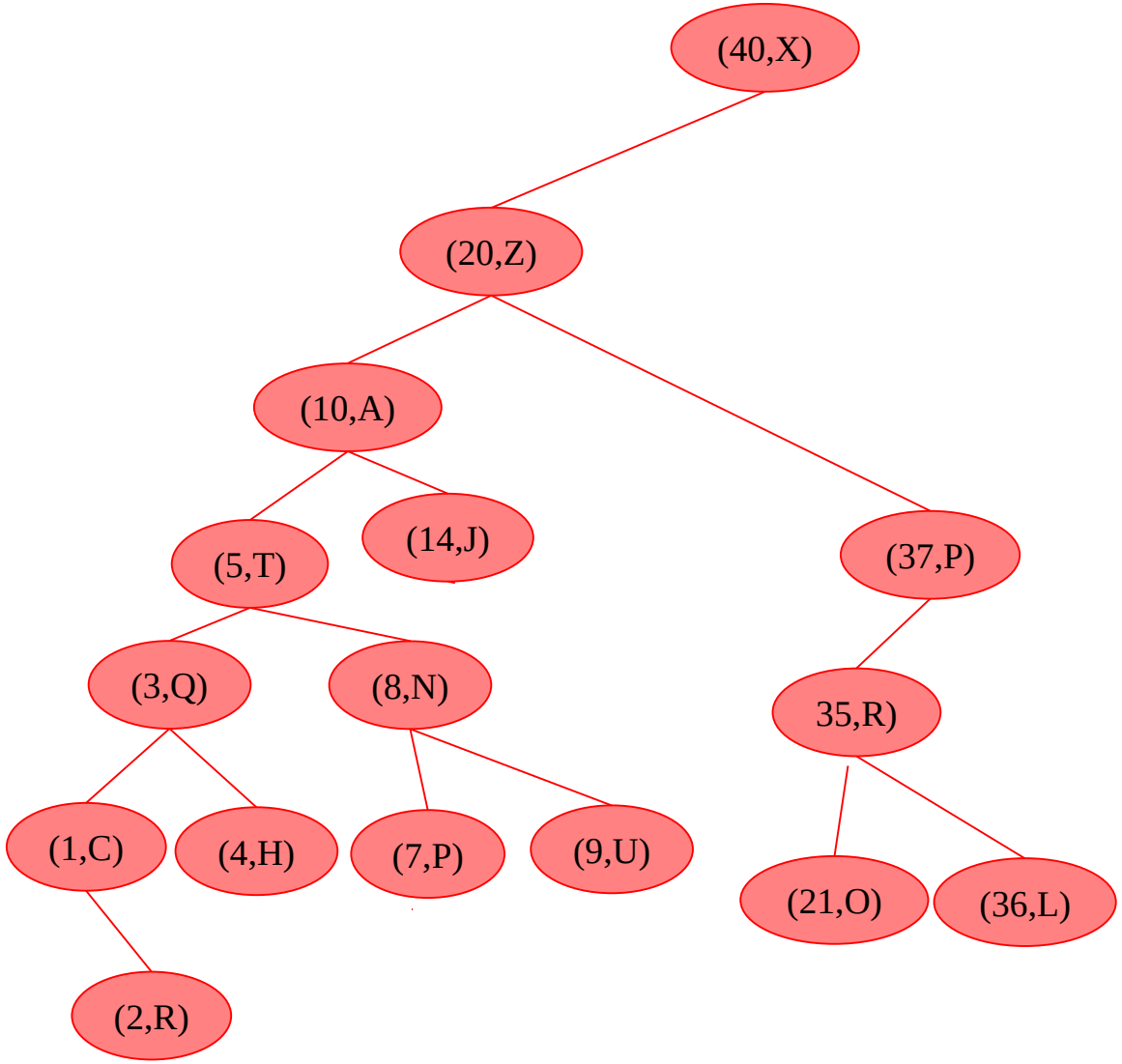
- e.g. splay (40,X)



after first splay



Example Result of Splaying



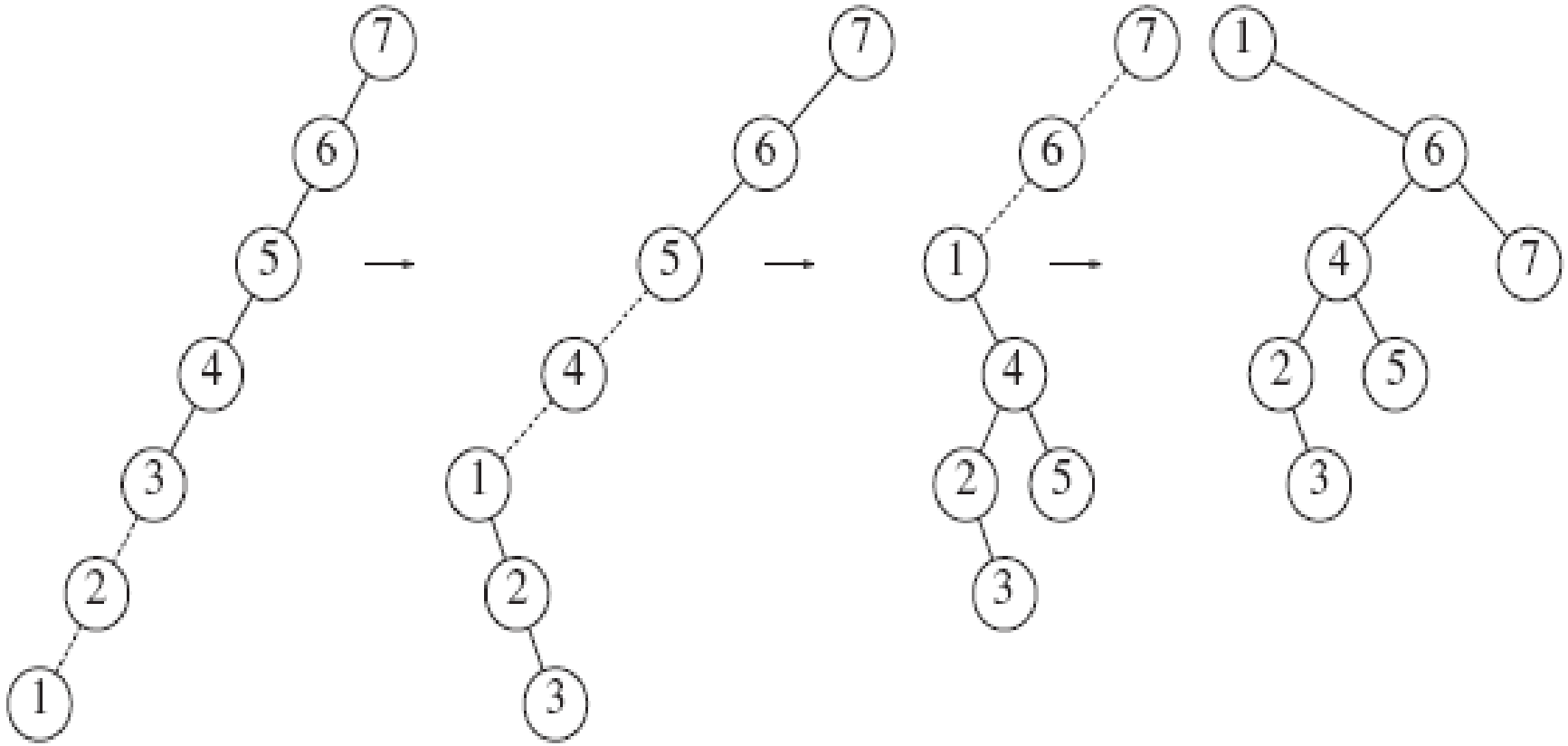


Figure 4.47 Result of splaying at node 1

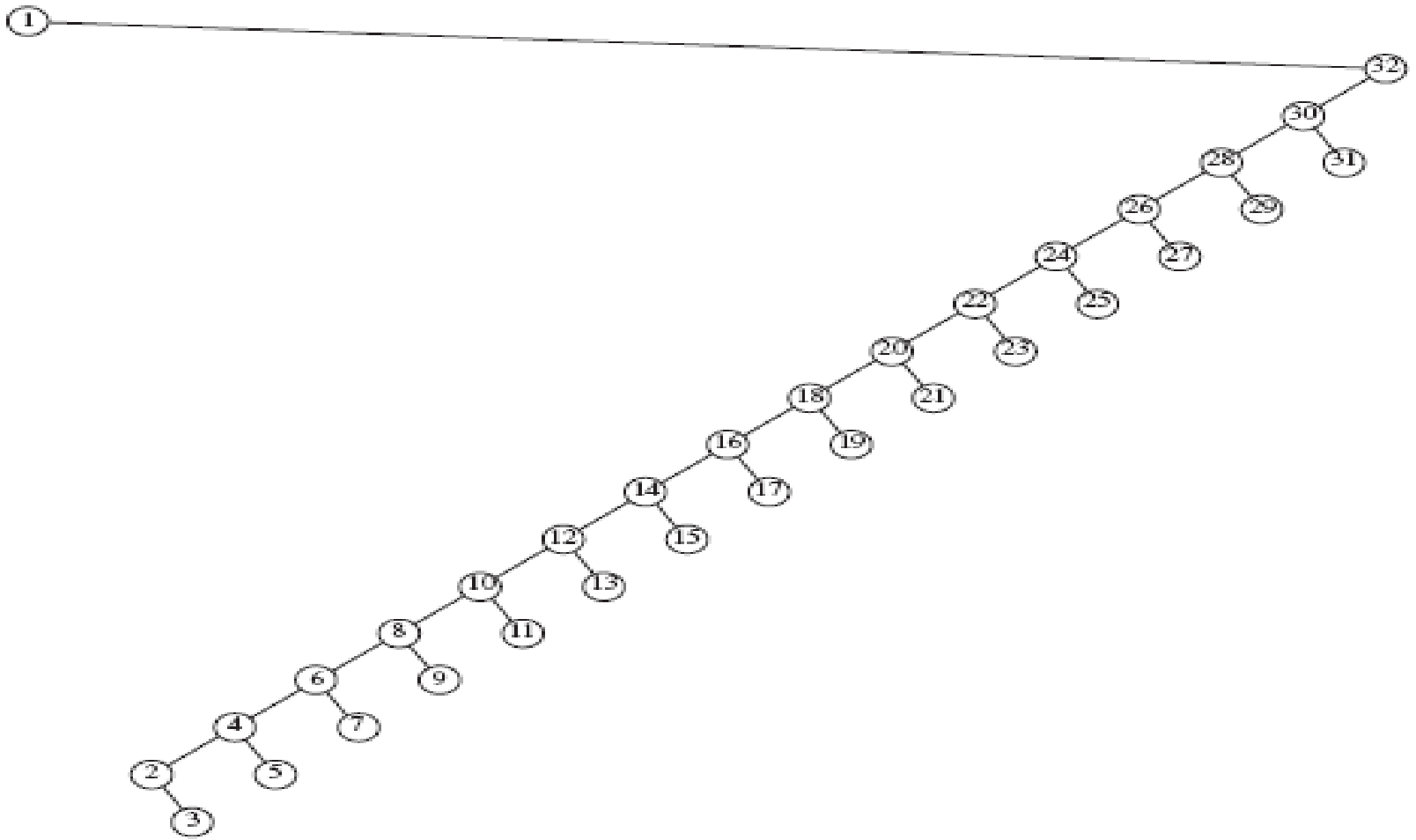


Figure 4.48 Result of splaying at node 1 a tree of all left children

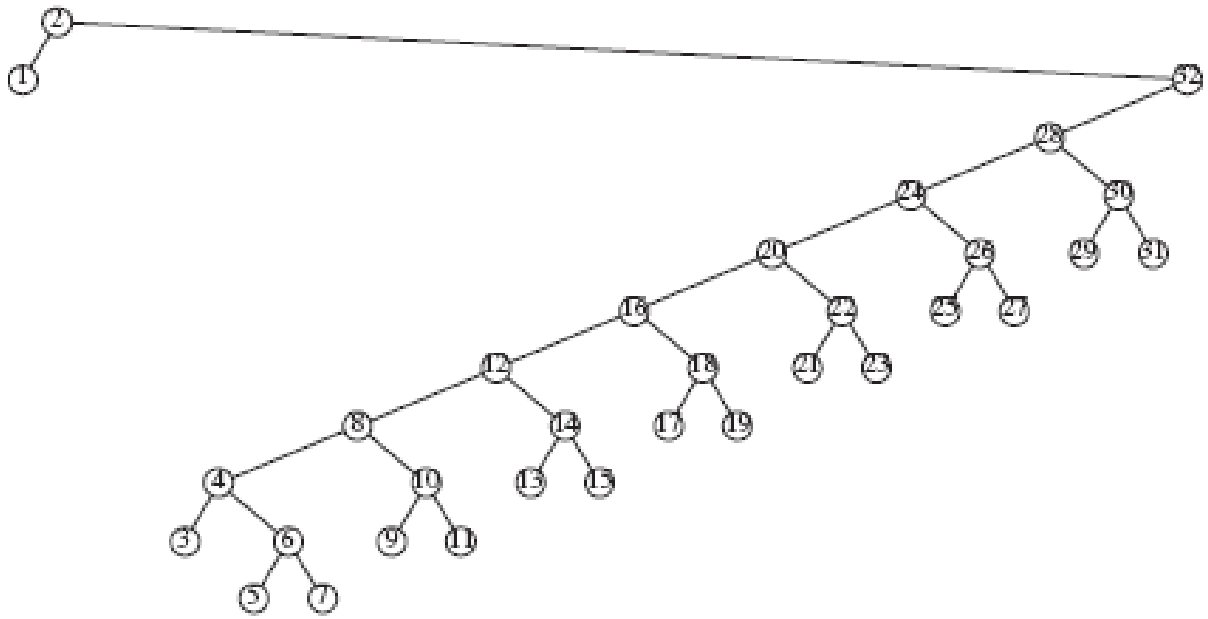


Figure 4.49 Result of splaying the previous tree at node 2

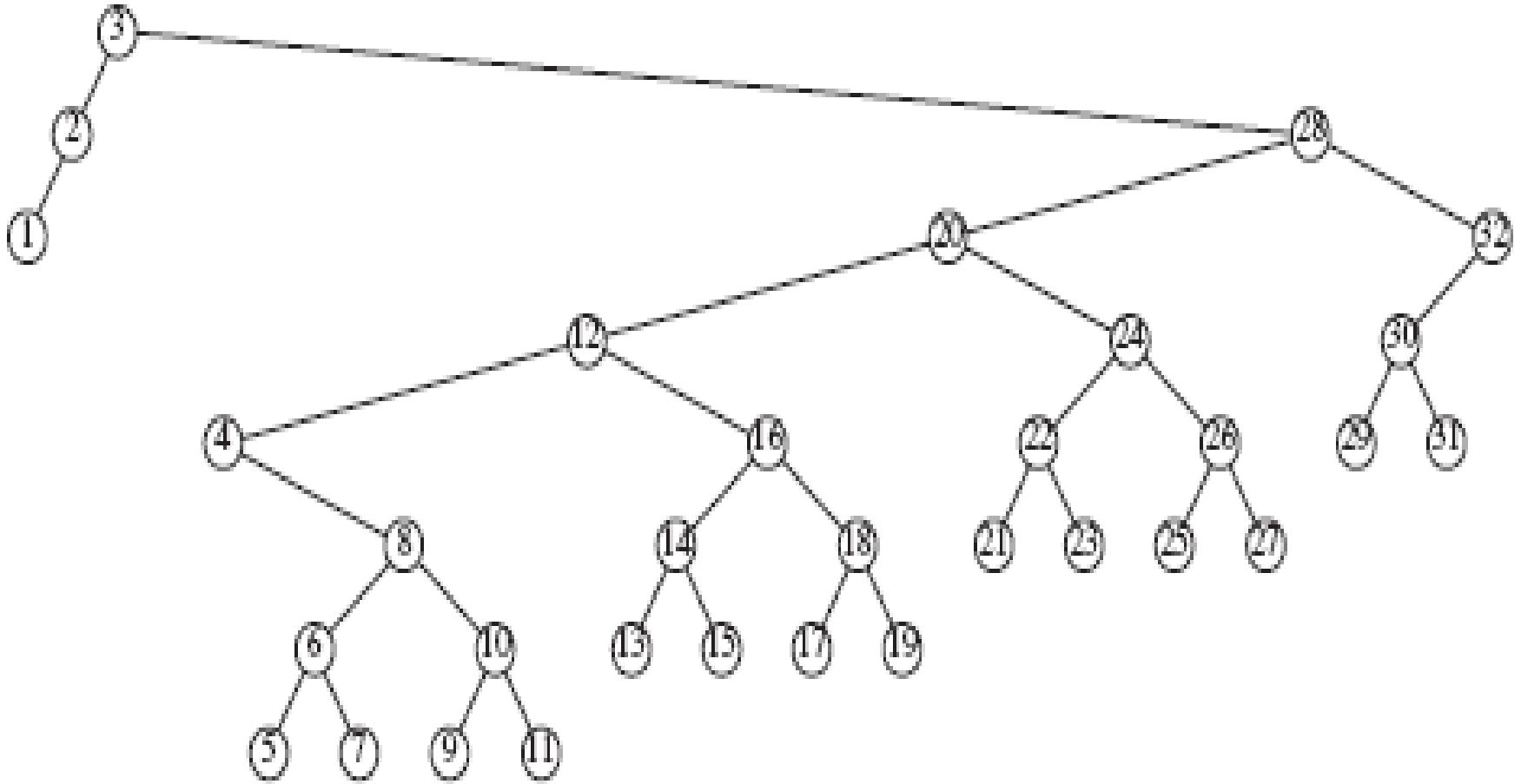


Figure 4.50 Result of splaying the previous tree at node 3

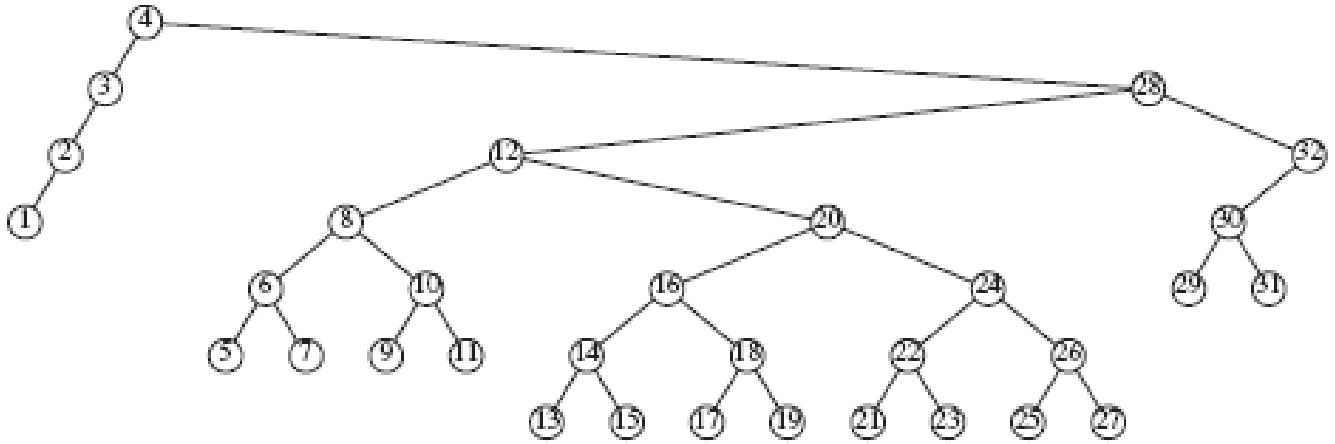


Figure 4.51 Result of splaying the previous tree at node 4

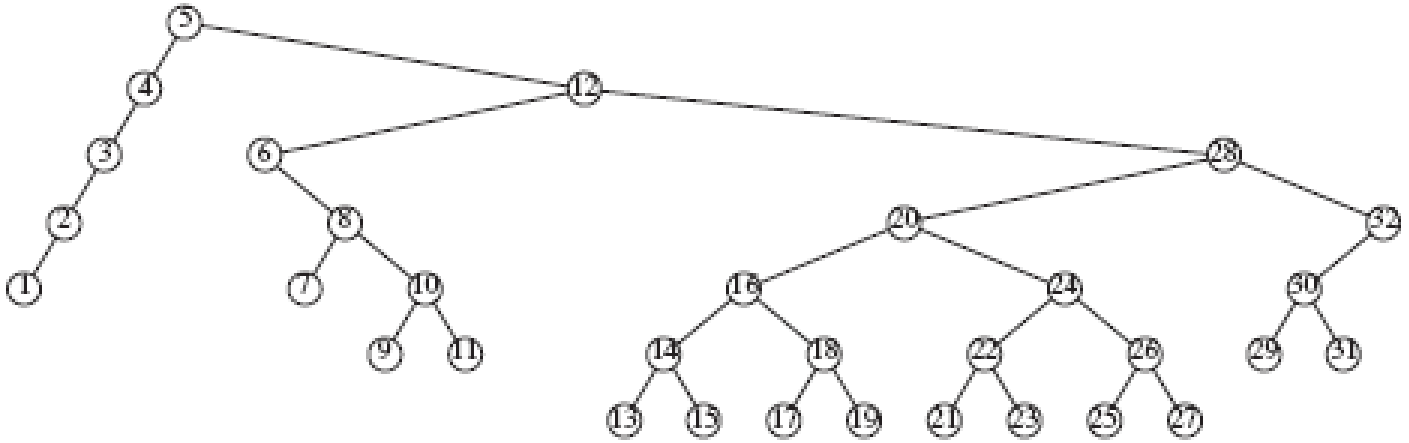


Figure 4.52 Result of splaying the previous tree at node 5

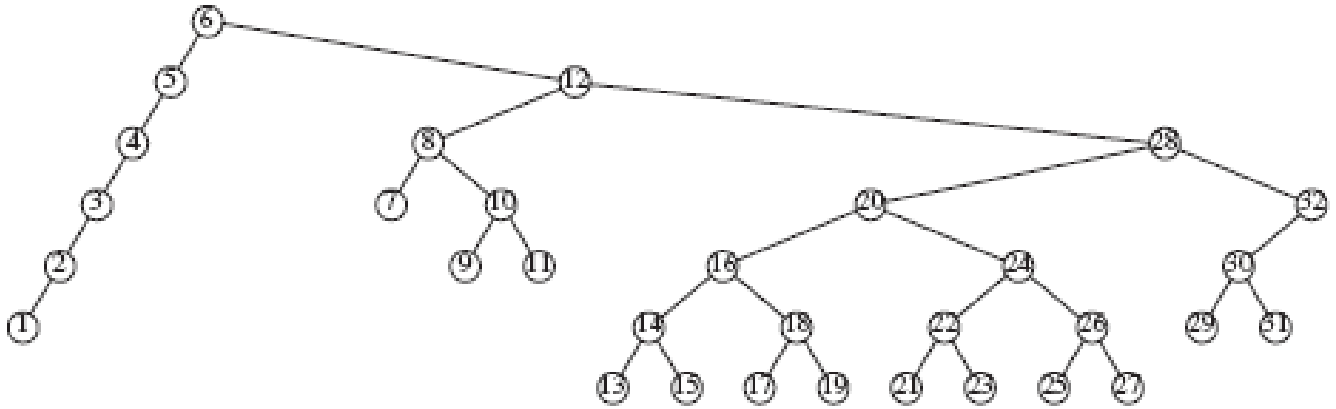


Figure 4.53 Result of splaying the previous tree at node 6

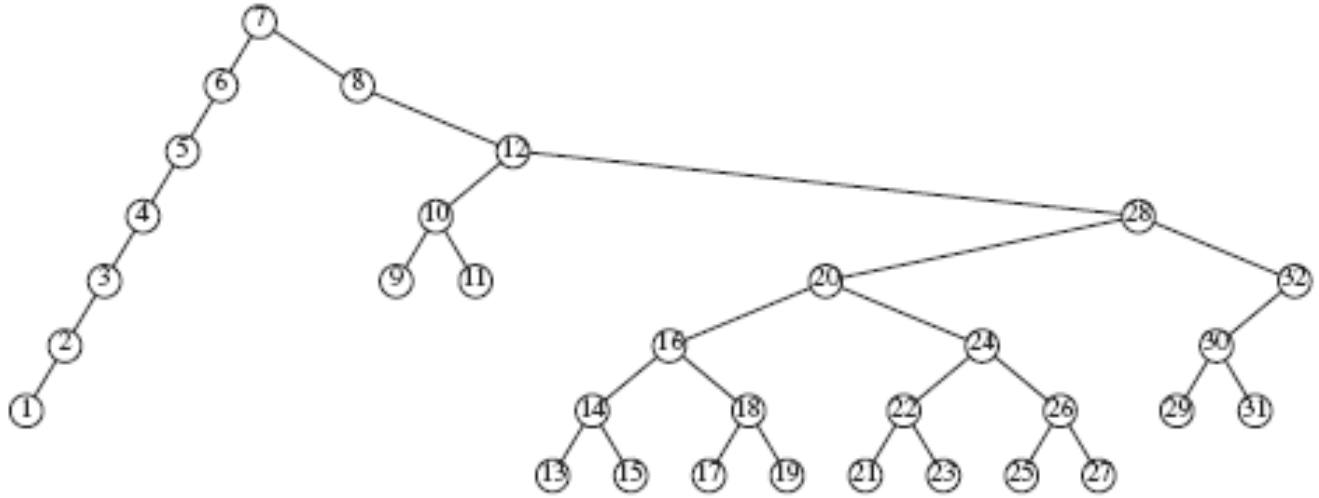


Figure 4.54 Result of splaying the previous tree at node 7

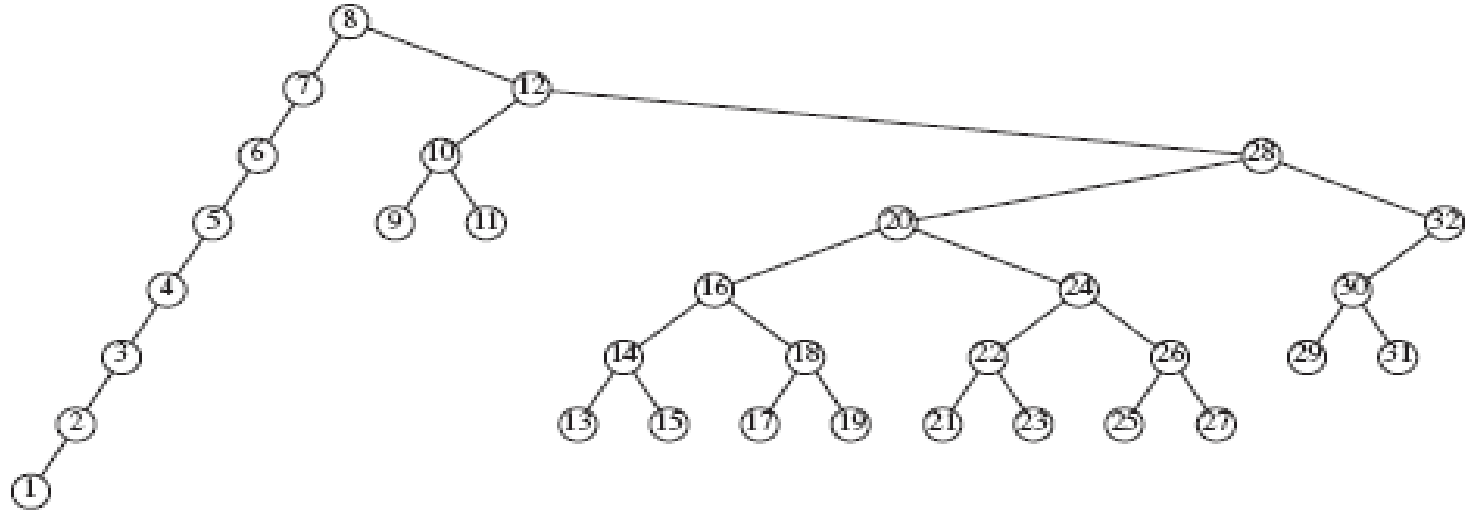


Figure 4.55 Result of splaying the previous tree at node 8

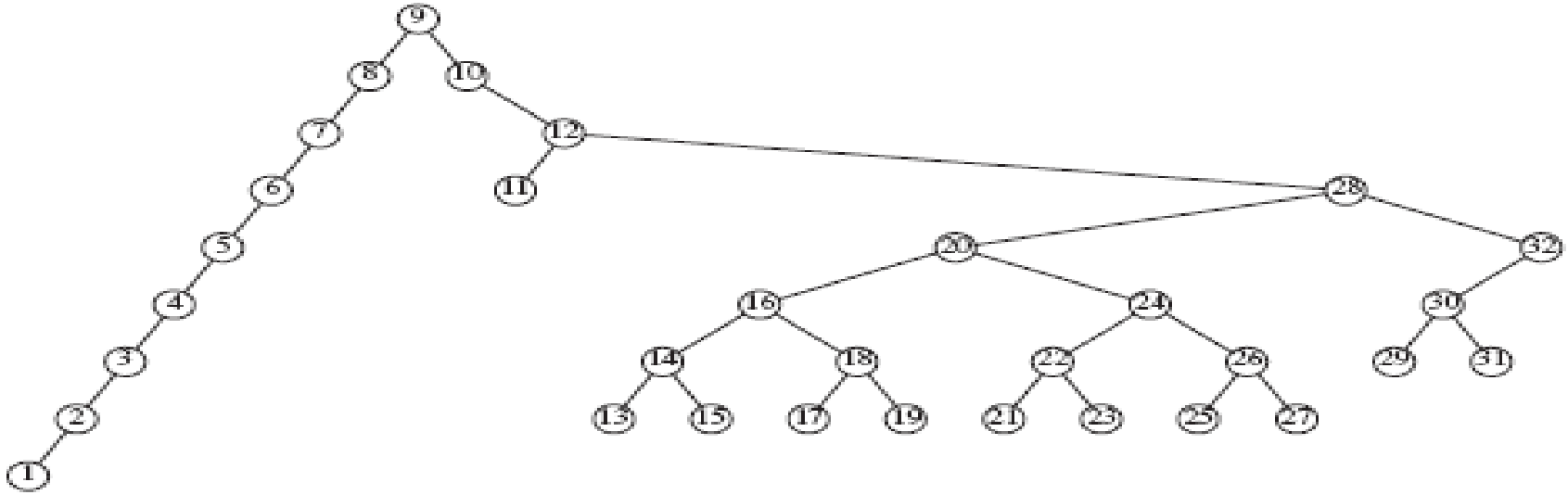
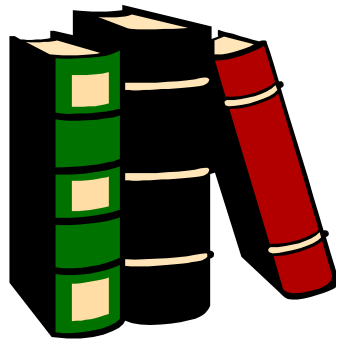


Figure 4.56 Result of splaying the previous tree at node 9

Splay Tree Definition



- a splay tree is a binary search tree where a node is splayed after it is accessed (for a search or update)
  - deepest internal node accessed is splayed
  - splaying costs O(h), where h is height of the tree – which is still O(n) worst-case
    - O(h) rotations, each of which is O(1)

Splay Trees & Ordered Dictionaries

- which nodes are splayed after each operation?

method	**splay node**
Search for k	if key found, use that node if key not found, use parent of ending external node
Insert (k,v)	use the new node containing the entry inserted
Remove item with key k	use the parent of the internal node that was actually removed from the tree (the parent of the node that the removed item was swapped with)

PERFORMANCE OF SPLAY TREES

- The analysis of splay tree is difficult because it must take into account the ever changing structure of the tree.
- Amortized cost of any splay operation is O(log n)
- In fact, the analysis goes through for any reasonable definition of rank(x)
- Splay trees can actually adapt to perform searches on frequently-requested items much faster than O(log n) in some cases

Tree Traversal

- In order traversal of any binary search tree will result in list of all items in sorted order.
- This is done in O(N)
- Post Order traversal is required in calculating the height of the tree. Running time O(N)
- Preorder Traversal like printing each node with depth also has O(N) running time.

In order Traversal

```
1         /**
2          * Print the tree contents in sorted order.
3          */
4         public void printTree( )
5         {
6             if( isEmpty( ) )
7                 System.out.println( "Empty tree" );
8             else
9                 printTree( root );
10        }
11
12        /**
13         * Internal method to print a subtree in sorted order.
14         * @param t the node that roots the subtree.
15         */
16        private void printTree( BinaryNode<AnyType> t )
17        {
18            if( t != null )
19            {
20                printTree( t.left );
21                System.out.println( t.element );
22                printTree( t.right );
23            }
24        }
```

Figure 4.57 Routine to print a binary search tree in order

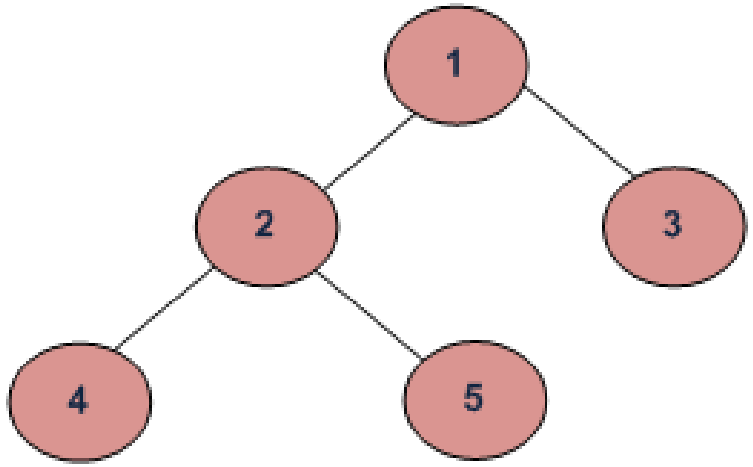
Post Order Traversal

```
1       /**
2        * Internal method to compute height of a subtree.
3        * @param t the node that roots the subtree.
4        */
5       private int height( BinaryNode<AnyType> t )
6       {
7           if( t == null )
8               return -1;
9           else
10              return 1 + Math.max( height( t.left ), height( t.right ) );
11      }
```

Figure 4.58 Routine to compute the height of a tree using a postorder traversal

LEVEL ORDER TRAVERSAL

- All nodes at depth d are processed before any node at depth d + 1.
- Recursion cannot be used.



Level order traversal of the tree is 1 2 3 4 5

B-TREES

- Search trees are expensive when they are stored on disk, because disk access are incredible expensive.
- Disk access in binary trees and AVL trees are proportional to their height.
- We introduce a new tree that has more branching and less height
- Binary tree with 31 nodes will have five levels. 5-ary tree of 31 nodes will have 3 levels.

height of binary tree = $\log_2 N$

height of M-ary tree = $\log_M N$

B-TREES

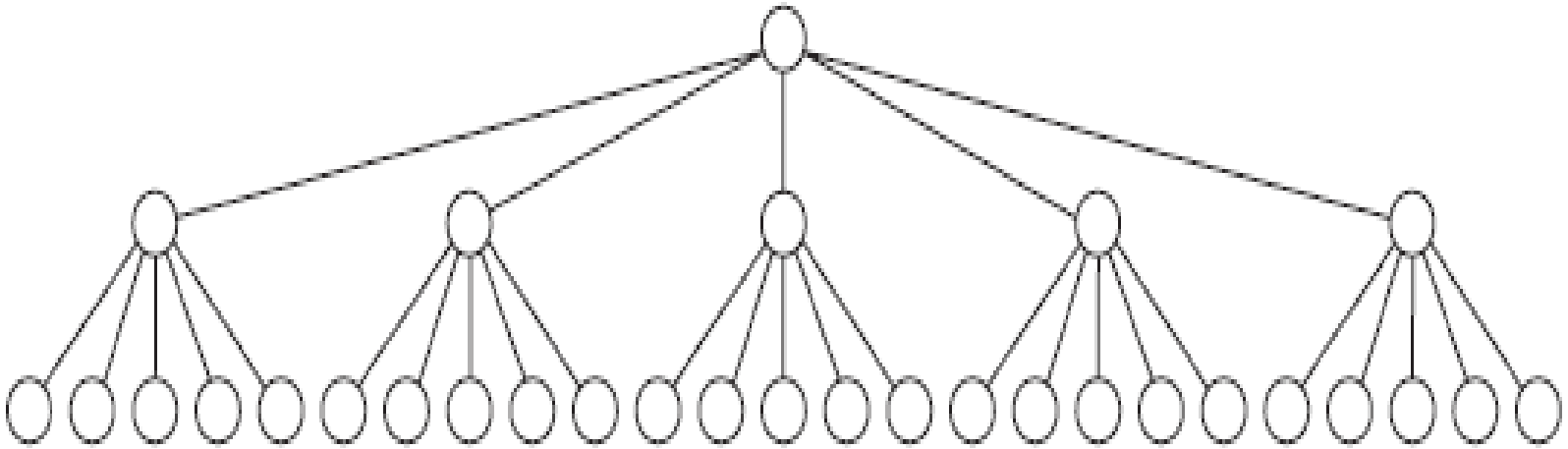


Figure 4.59 5-ary tree of 31 nodes has only three levels

PROPERTIES

1. The data items are stores at leaves
2. The non-leaf nodes store up to M-1 keys to guide the searching; key i represents the smallest key in subtree i+1
3. The root is either a leaf or has between two and M-children.
4. All non-leaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children.
5. All leaves are at the same depth and have between $\lceil L/2 \rceil$ and L data items, for some L.

Each node represents a disk block, we choose M and L on the basis of the size of the items that are being stores.

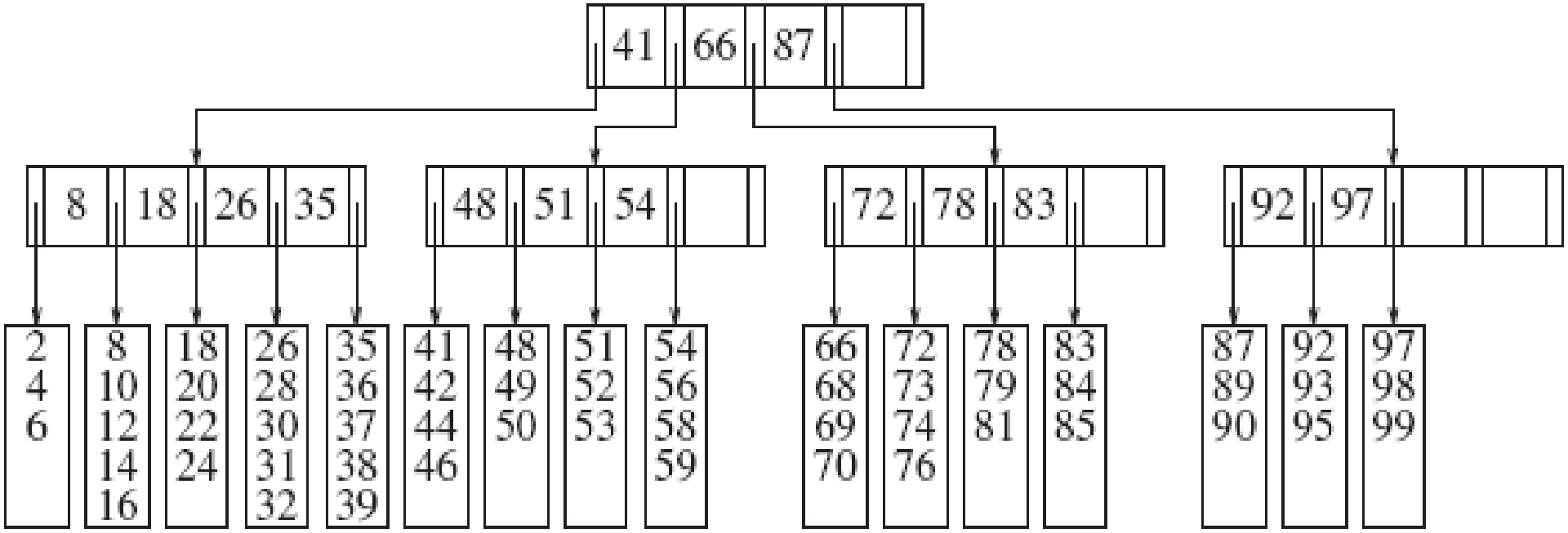


Figure 4.60 B-tree of order 5

Inserting 57 into B-tree does not change the tree.

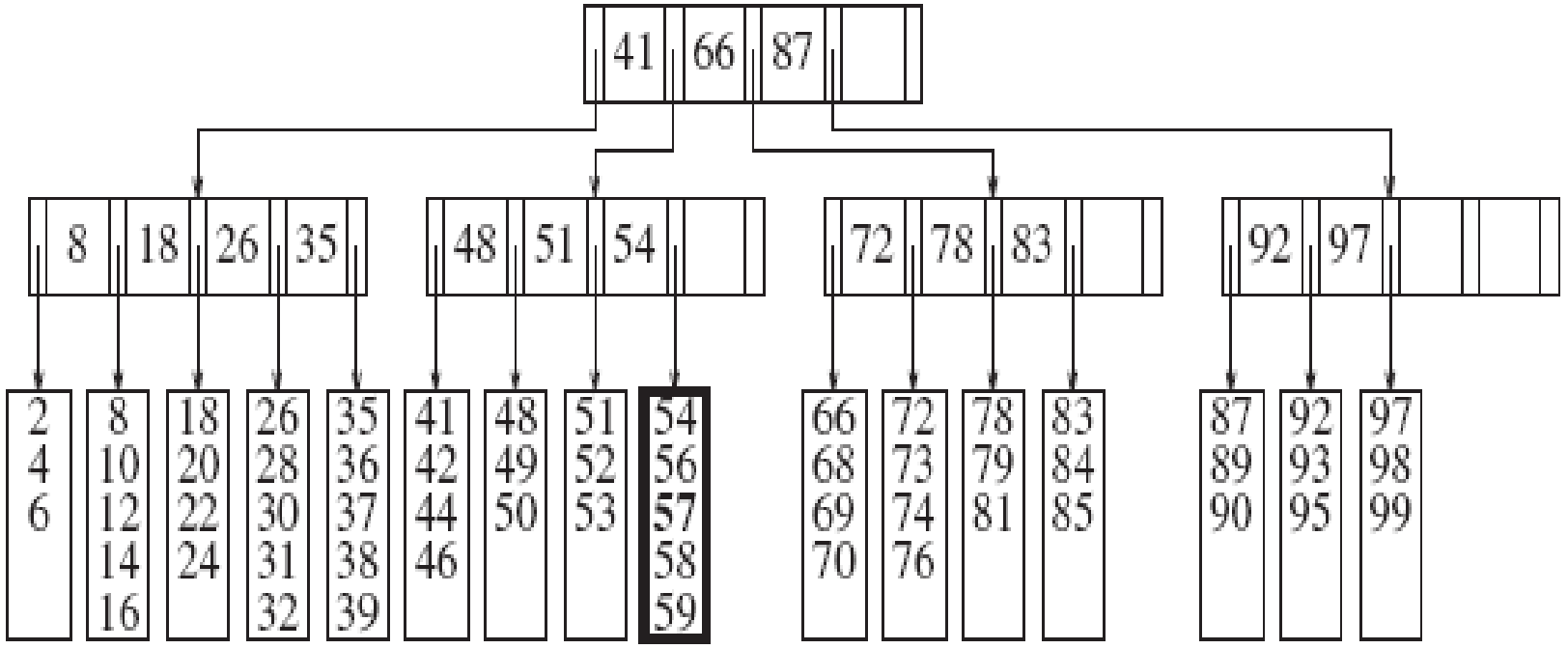


Figure 4.61 B-tree after insertion of 57 into the tree in Figure 4.60

Inserting 55 into the B-Tree violates the rules since the leaf where 55 wants to go is already full. Split the leaf into two leaves and update the parent

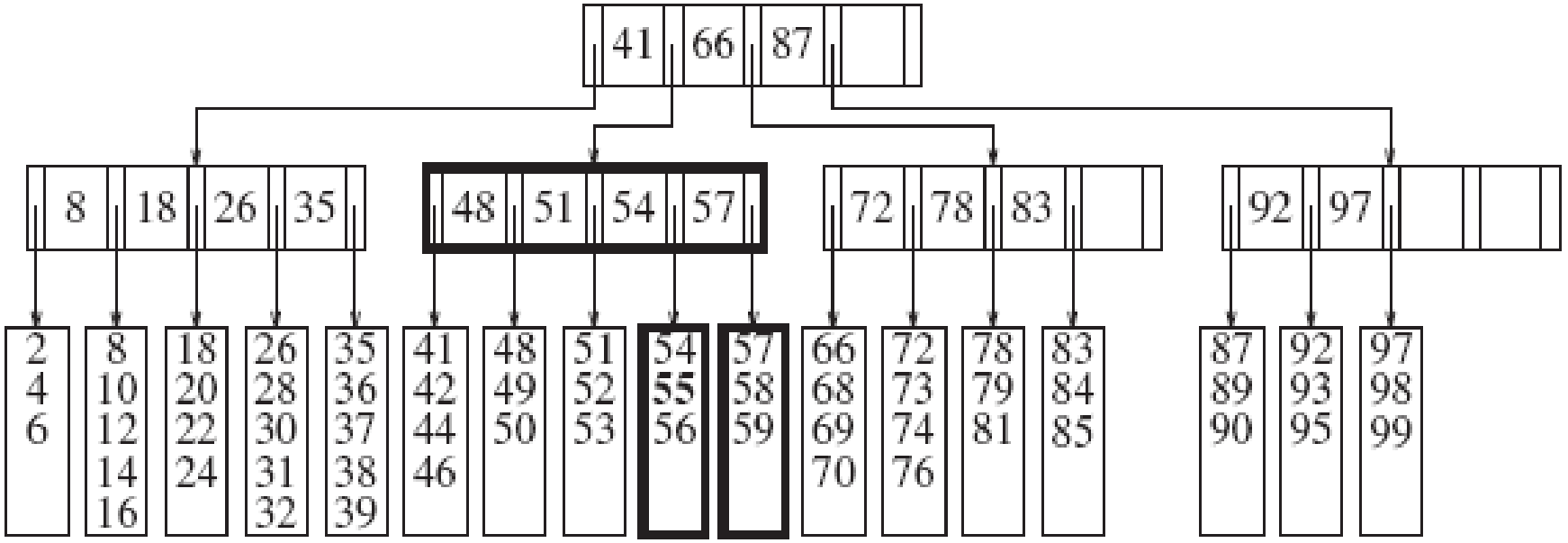


Figure 4.62 Insertion of 55 into the B-tree in Figure 4.61 causes a split into two leaves

Inserting 40 into the B-Tree creates more problems since the parent is also full.
If the parent is full split the parent

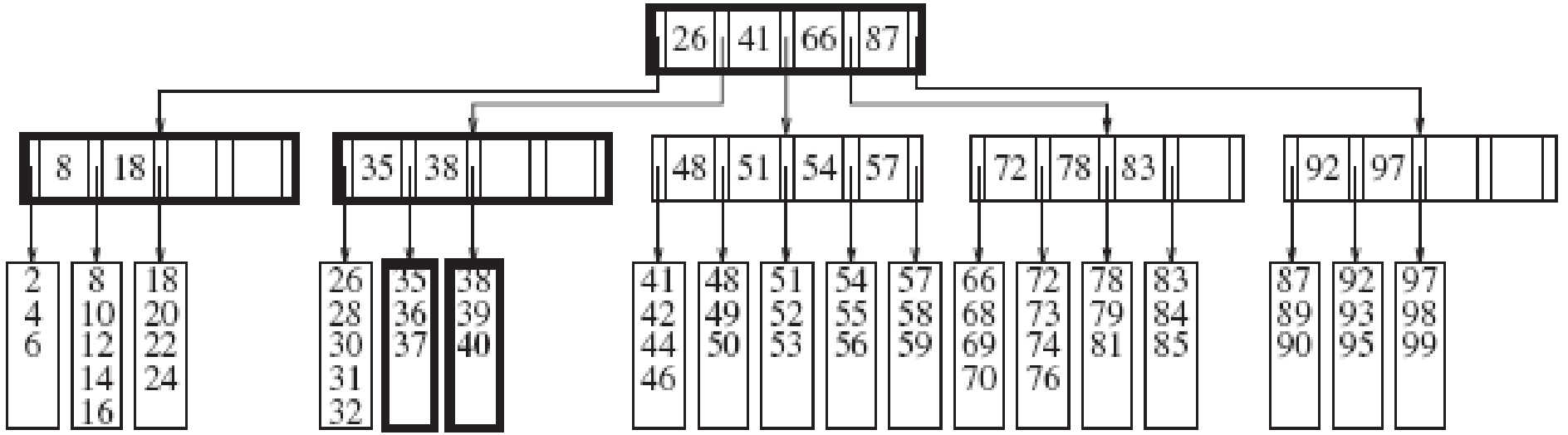


Figure 4.63 Insertion of 40 into the B-tree in Figure 4.62 causes a split into two leaves and then a split of the parent node

There are other ways to handle overflowing of children. One technique is to pit a child up for adoption should a neighbor have room. For example inserting 29 into the B-Tree., moving 32 to the next leaf creates space

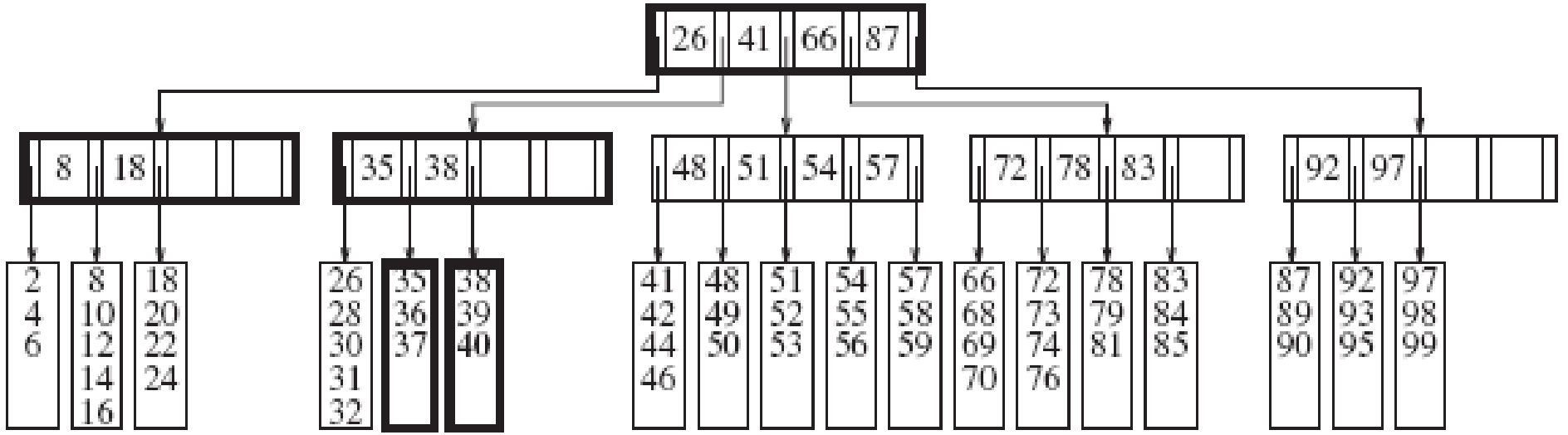


Figure 4.63 Insertion of 40 into the B-tree in Figure 4.62 causes a split into two leaves and then a split of the parent node

Deletion is performed by finding the item and removing it. The problem is that if that leaf that item was in had minimum number of data items. This is rectified by adopting a neighbor. If the neighbor is also at minimum occupancy combine them to form one node. This means the parent has lost a child. if this causes the parent to fall below its minimum, then follow the same strategy all the way up to the root.

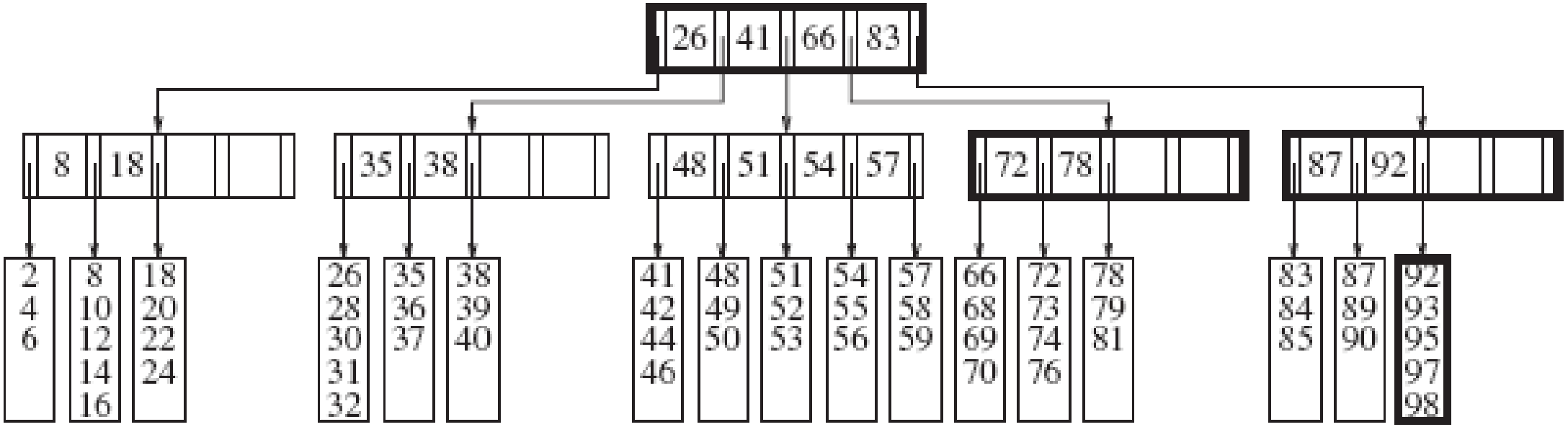


Figure 4.64 B-tree after the deletion of 99 from the B-tree in Figure 4.63

Sets and Maps in Standard Library

The Set interface extends the Collection interface. It does not introduce new methods or constants, but it stipulates that an instance of Set contains no duplicate elements. The concrete classes that implement Set must ensure that no duplicate elements can be added to the set. That is no two elements e1 and e2 can be in the set such that e1.equals(e2) is true.

The SortedSet Interface and the TreeSet Class

SortedSet is a sub-interface of Set, which guarantees that the elements in the set are sorted. TreeSet is a concrete class that implements the SortedSet interface. You can use an iterator to traverse the elements in the sorted order. The elements can be sorted in two ways.

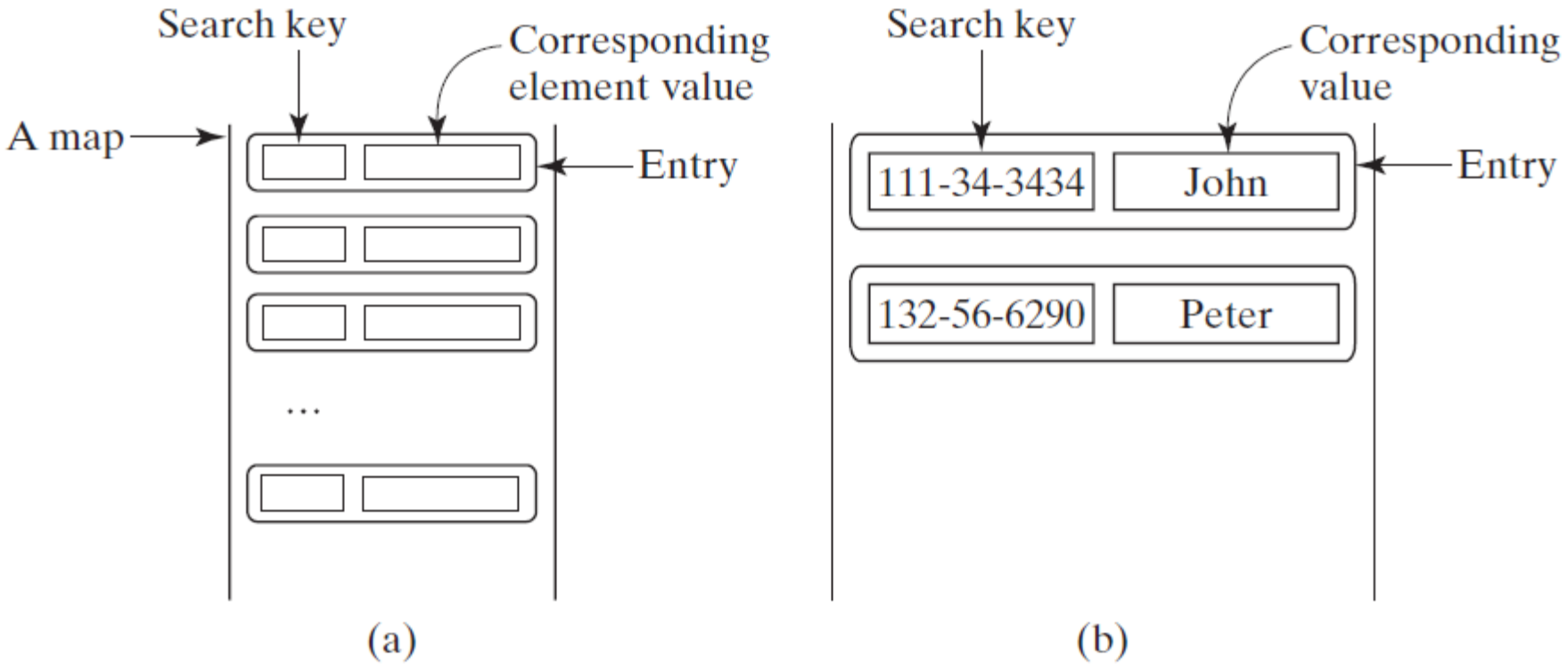
The SortedSet Interface and the TreeSet Class, cont.

One way is to use the Comparable interface.

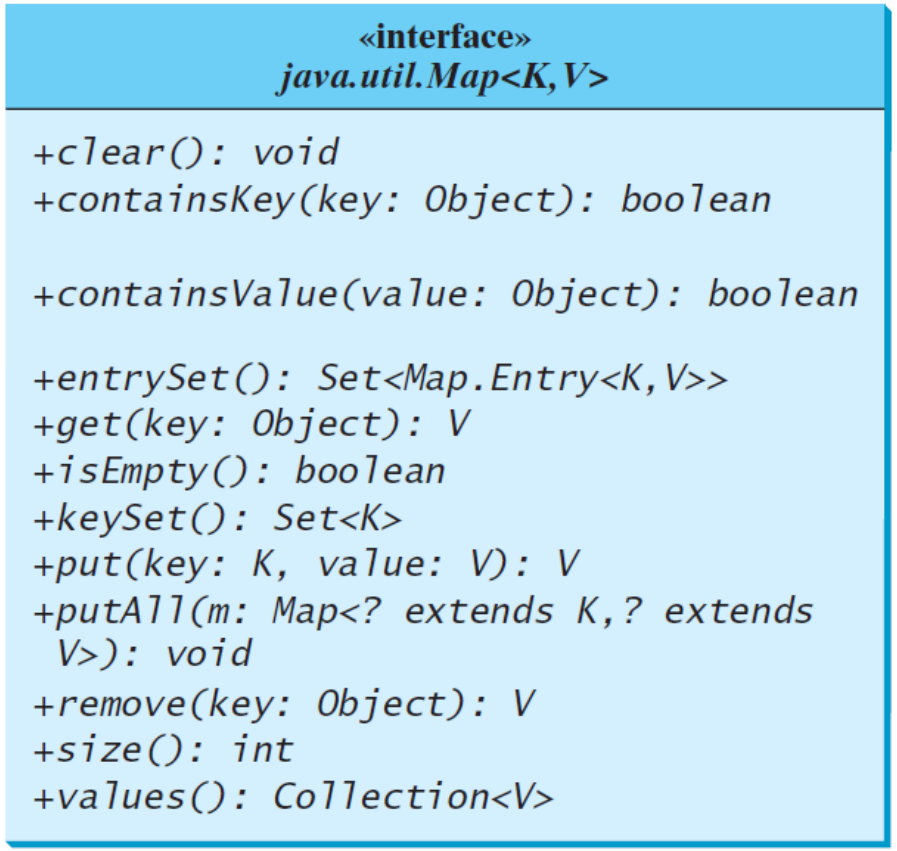
The other way is to specify a comparator for the elements in the set if the class for the elements does not implement the Comparable interface, or you don't want to use the compareTo method in the class that implements the Comparable interface. This approach is referred to as *order by comparator*.

The Map Interface

The Map interface maps keys to the elements. The keys are like indexes. In List, the indexes are integer. In Map, the keys can be any objects.



The Map Interface UML Diagram



Removes all entries from this map.

Returns true if this map contains an entry for the specified key.

Returns true if this map maps one or more keys to the specified value.

Returns a set consisting of the entries in this map.

Returns the value for the specified key in this map.

Returns true if this map contains no entries.

Returns a set consisting of the keys in this map.

Puts an entry into this map.

Adds all the entries from `m` to this map.

Removes the entries for the specified key.

Returns the number of entries in this map.

Returns a collection consisting of the values in this map.

```
1       public static void printHighChangeables( Map<String,List<String>> adjWords,
2                                                int minWords )
3       {
4           for( Map.Entry<String,List<String>> entry : adjWords.entrySet( ) )
5           {
6               List<String> words = entry.getValue( );
7
8               if( words.size( ) >= minWords )
9               {
10                  System.out.print( entry.getKey( ) + " (" );
11                  System.out.print( words.size( ) + "):" );
12                  for( String w : words )
13                      System.out.print( " " + w );
14                  System.out.println( );
15              }
16          }
17      }
```

Figure 4.65 Given a map containing words as keys and a list of words that differ in only one character as values, output words that have `minWords` or more words obtainable by a one-character substitution

```
1       // Returns true if word1 and word2 are the same length
2       // and differ in only one character.
3       private static boolean oneCharOff( String word1, String word2 )
4       {
5           if( word1.length( ) != word2.length( ) )
6               return false;
7
8           int diffs = 0;
9
10          for( int i = 0; i < word1.length( ); i++ )
11              if( word1.charAt( i ) != word2.charAt( i ) )
12                  if( ++diffs > 1 )
13                      return false;
14
15          return diffs == 1;
16      }
```

Figure 4.66 Routine to check if two words differ in only one character

```
1         // Computes a map in which the keys are words and values are Lists of words
2         // that differ in only one character from the corresponding key.
3         // Uses a quadratic algorithm (with appropriate Map).
4         public static Map<String,List<String>>
5         computeAdjacentWords( List<String> theWords )
6         {
7             Map<String,List<String>> adjWords = new TreeMap<>( );
8
9             String [ ] words = new String[ theWords.size( ) ];
10
11            theWords.toArray( words );
12            for( int i = 0; i < words.length; i++ )
13                for( int j = i + 1; j < words.length; j++ )
14                    if( oneCharOff( words[ i ], words[ j ] ) )
15                    {
16                        update( adjWords, words[ i ], words[ j ] );
17                        update( adjWords, words[ j ], words[ i ] );
18                    }
19
20            return adjWords;
21        }
22
23        private static <KeyType> void update( Map<KeyType,List<String>> m,
24                                              KeyType key, String value )
25        {
26            List<String> lst = m.get( key );
27            if( lst == null )
28            {
29                lst = new ArrayList<>( );
30                m.put( key, lst );
31            }
32
33            lst.add( value );
34        }
```

Figure 4.67 Function to compute a map containing words as keys and a list of words that differ in only one character as values. This version runs in 75 seconds on an 89,000-word dictionary

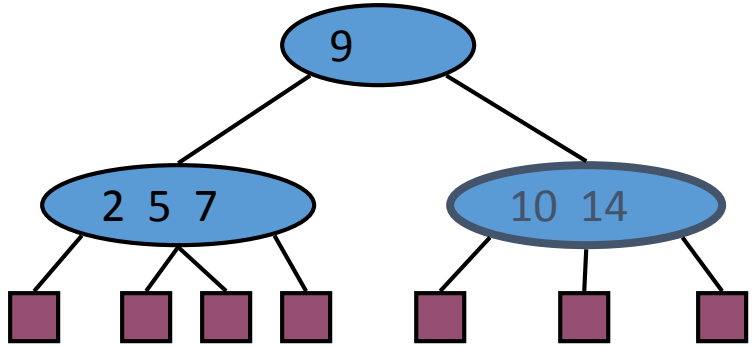
```
1         // Computes a map in which the keys are words and values are Lists of words
2         // that differ in only one character from the corresponding key.
3         // Uses a quadratic algorithm (with appropriate Map), but speeds things by
4         // maintaining an additional map that groups words by their length.
5         public static Map<String,List<String>>
6         computeAdjacentWords( List<String> theWords )
7         {
8             Map<String,List<String>> adjWords = new TreeMap<>( );
9             Map<Integer,List<String>> wordsByLength = new TreeMap<>( );
10
11              // Group the words by their length
12            for( String w : theWords )
13                update( wordsByLength, w.length( ), w );
14
15              // Work on each group separately
16            for( List<String> groupsWords : wordsByLength.values( ) )
17            {
18                String [ ] words = new String[ groupsWords.size( ) ];
19
20                groupsWords.toArray( words );
21                for( int i = 0; i < words.length; i++ )
22                    for( int j = i + 1; j < words.length; j++ )
23                        if( oneCharOff( words[ i ], words[ j ] ) )
24                        {
25                            update( adjWords, words[ i ], words[ j ] );
26                            update( adjWords, words[ j ], words[ i ] );
27                        }
28            }
29
30            return adjWords;
31        }
```

Figure 4.68 Function to compute a map containing words as keys and a list of words that differ in only one character as values. Splits words into groups by word length. This version runs in 16 seconds on an 89,000-word dictionary

```
1         // Computes a map in which the keys are words and values are Lists of words
2         // that differ in only one character from the corresponding key.
3         // Uses an efficient algorithm that is O(N log N) with a TreeMap.
4         public static Map<String,List<String>>
5         computeAdjacentWords( List<String> words )
6         {
7             Map<String,List<String>> adjWords = new TreeMap<>( );
8             Map<Integer,List<String>> wordsByLength = new TreeMap<>( );
9
10              // Group the words by their length
11            for( String w : words )
12                update( wordsByLength, w.length( ), w );
13
14              // Work on each group separately
15            for( Map.Entry<Integer,List<String>> entry : wordsByLength.entrySet( ) )
16            {
17                List<String> groupsWords = entry.getValue( );
18                int groupNum = entry.getKey( );
19
20                // Work on each position in each group
21                for( int i = 0; i < groupNum; i++ )
22                {
23                    // Remove one character in specified position, computing
24                    // representative.  Words with same representative are
25                    // adjacent, so first populate a map ...
26                    Map<String,List<String>> repToWord = new TreeMap<>( );
27
28                    for( String str : groupsWords )
29                    {
30                        String rep = str.substring( 0, i ) + str.substring( i + 1 );
31                        update( repToWord, rep, str );
32                    }
33
34                    // and then look for map values with more than one string
35                    for( List<String> wordClique : repToWord.values( ) )
36                        if( wordClique.size( ) >= 2 )
37                            for( String s1 : wordClique )
38                                for( String s2 : wordClique )
39                                    if( s1 != s2 )
40                                        update( adjWords, s1, s2 );
41                }
42            }
43
44            return adjWords;
45        }
```

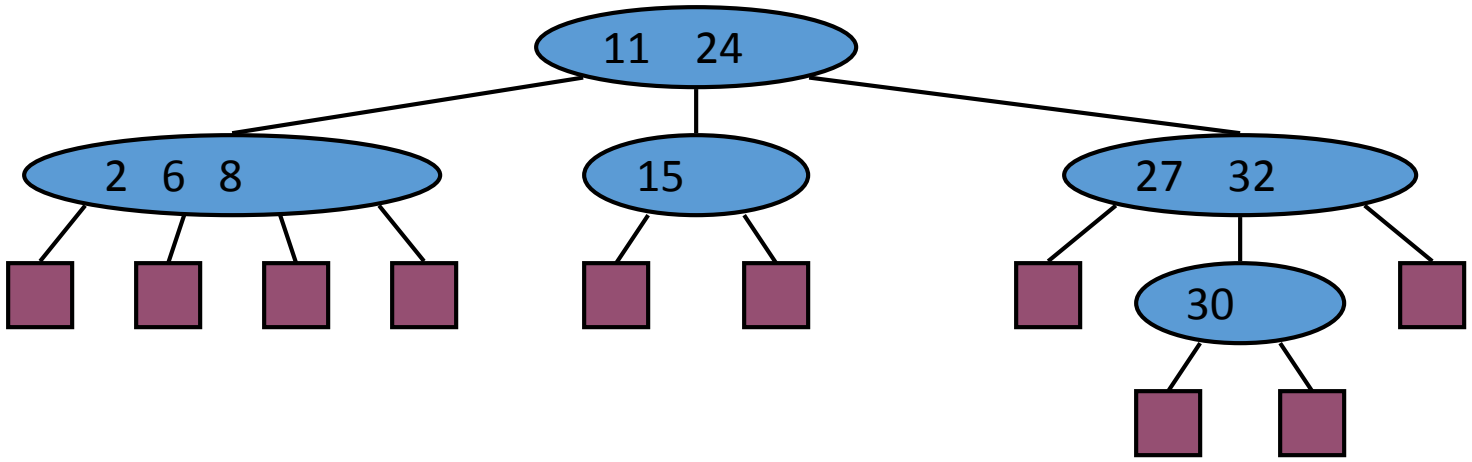
Figure 4.69 Function to compute a map containing words as keys and a list of words that differ in only one character as values. Runs in 1 second on an 89,000-word dictionary

(2,4) Trees



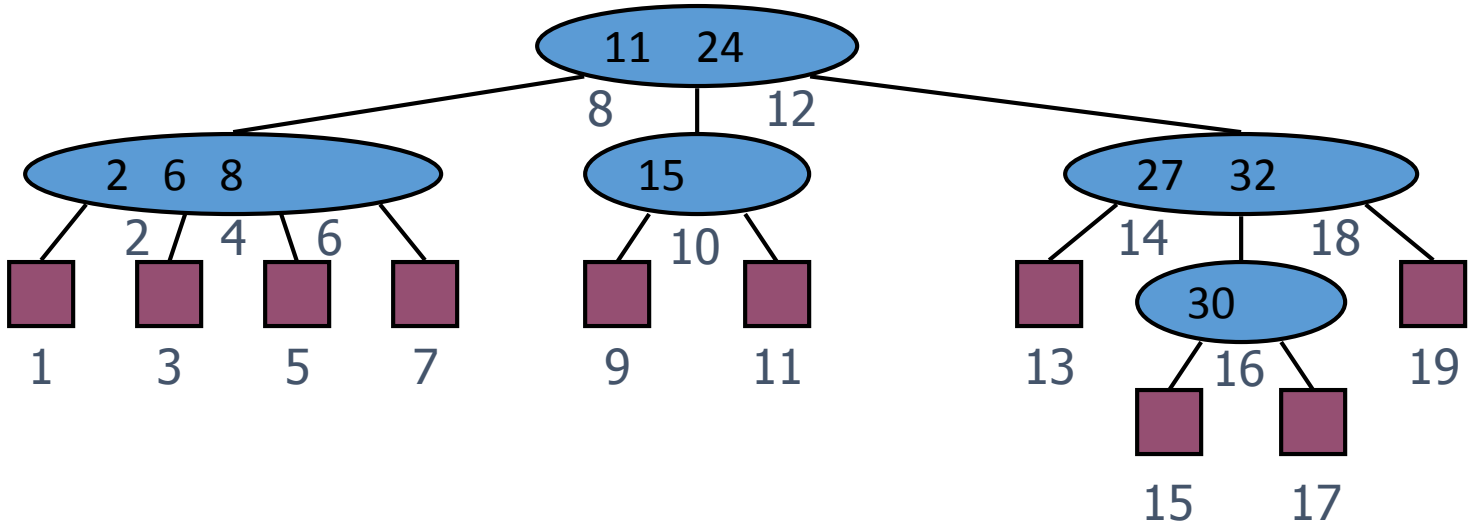
Multi-Way Search Tree

- A multi-way search tree is an ordered tree such that
  - Each internal node has at least two children and stores $d-1$ key-element items (k_i, o_i), where d is the number of children
  - For a node with children $v_1 v_2 \ldots v_d$ storing keys $k_1 k_2 \ldots k_{d-1}$
    - keys in the subtree of v_1 are less than k_1
    - keys in the subtree of v_i are between k_{i-1} and k_i $(i = 2, \ldots, d-1)$
    - keys in the subtree of v_d are greater than k_{d-1}
  - The leaves store no items and serve as placeholders



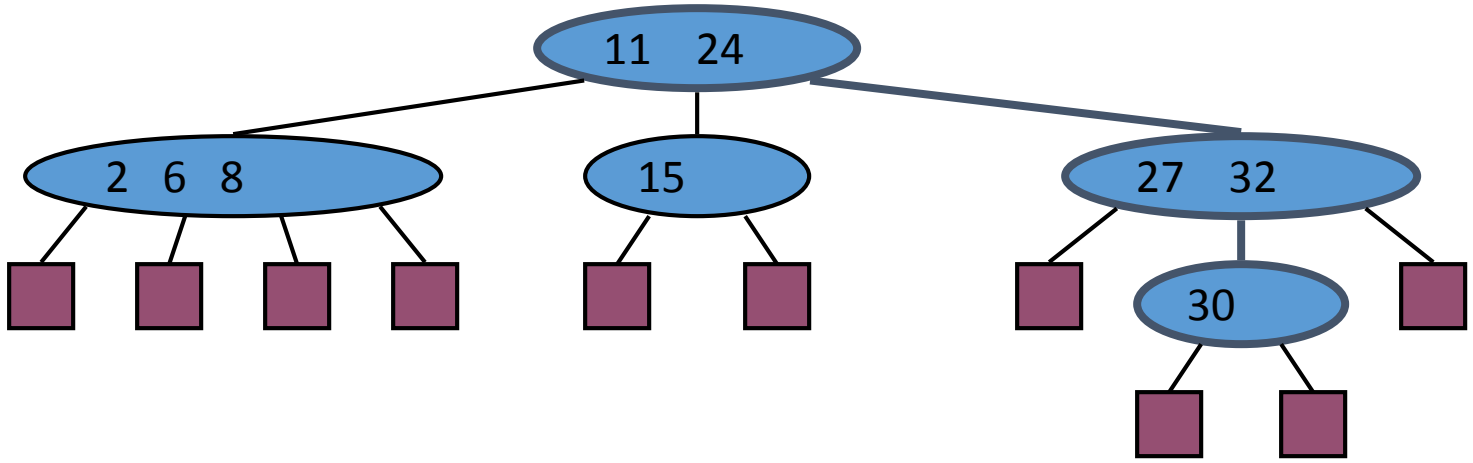
Multi-Way Inorder Traversal

- We can extend the notion of inorder traversal from binary trees to multi-way search trees
- Namely, we visit item (k_i, o_i) of node v between the recursive traversals of the subtrees of v rooted at children v_i and v_{i+1}
- An inorder traversal of a multi-way search tree visits the keys in increasing order



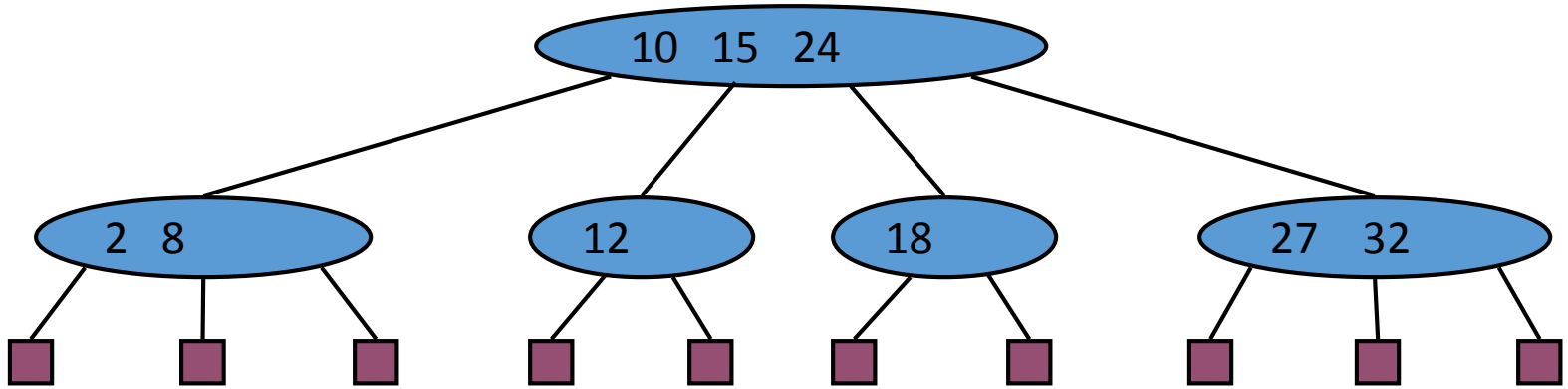
Multi-Way Searching

- Similar to search in a binary search tree
- A each internal node with children $v_1\, v_2 \ldots v_d$ and keys $k_1\, k_2 \ldots k_{d-1}$
  - $k = k_i$ $(i = 1, \ldots, d-1)$: the search terminates successfully
  - $k < k_1$: we continue the search in child v_1
  - $k_{i-1} < k < k_i$ $(i = 2, \ldots, d-1)$: we continue the search in child v_i
  - $k > k_{d-1}$: we continue the search in child v_d
- Reaching an external node terminates the search unsuccessfully
- Example: search for 30



(2,4) Trees

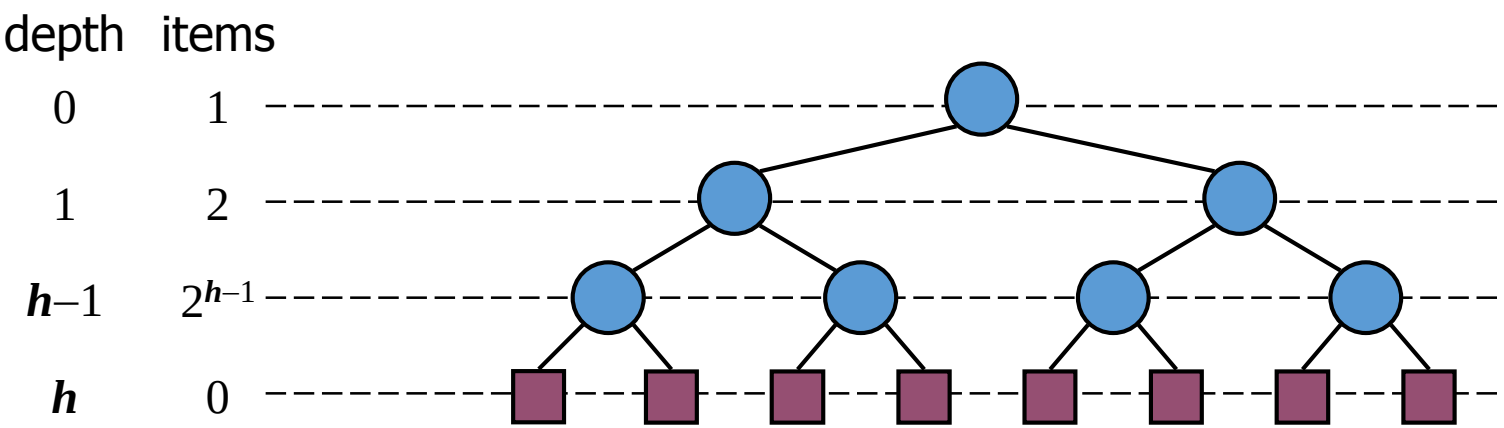
- A (2,4) tree (also called 2-4 tree or 2-3-4 tree) is a multi-way search with the following properties
  - Node-Size Property: every internal node has at most four children
  - Depth Property: all the external nodes have the same depth
- Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node



Height of a (2,4) Tree

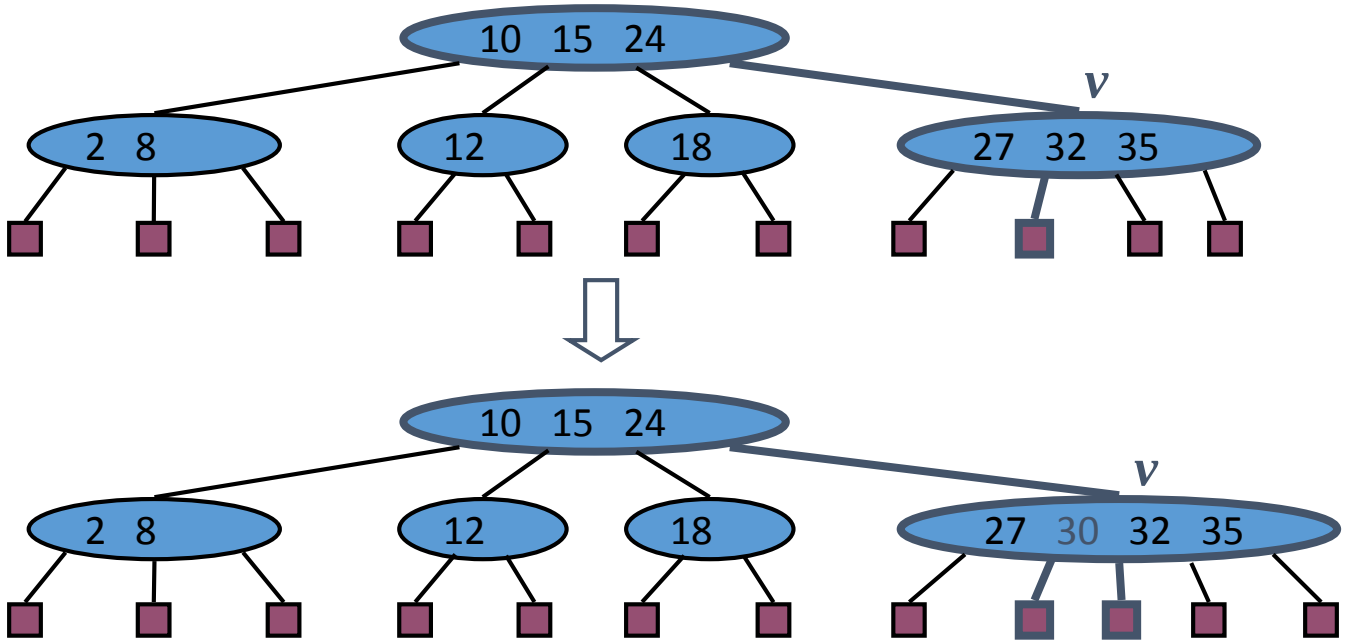
- Theorem: A (2,4) tree storing n items has height $O(\log n)$

  Proof:
  - Let h be the height of a (2,4) tree with n items
  - Since there are at least 2^i items at depth $i = 0, \ldots, h-1$ and no items at depth h, we have
    $$n \geq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$$
  - Thus, $h \leq \log(n+1)$
- Searching in a (2,4) tree with n items takes $O(\log n)$ time



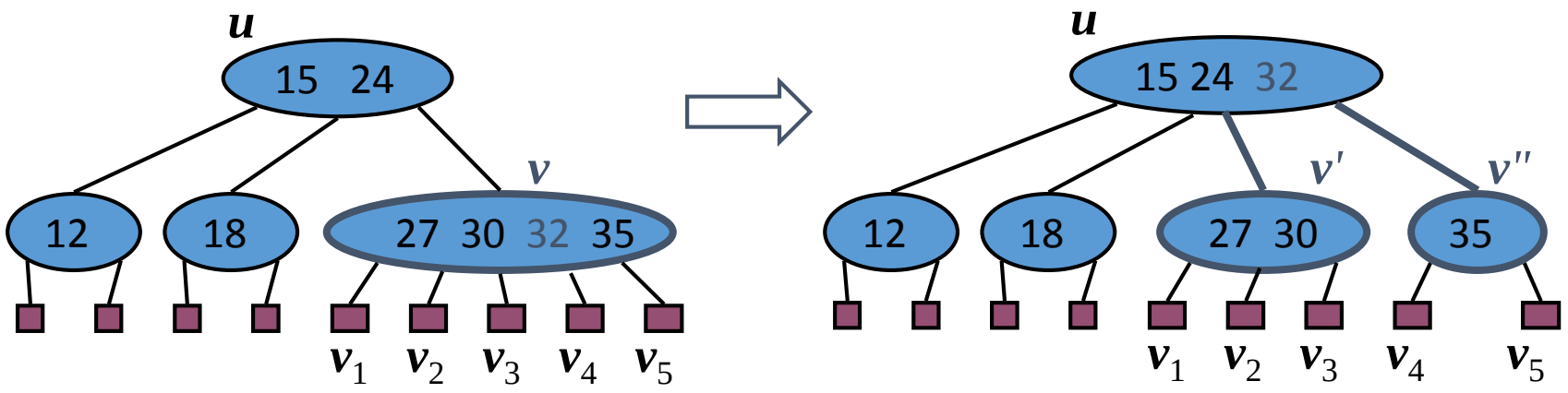
Insertion

- We insert a new item (k, o) at the parent v of the leaf reached by searching for k
  - We preserve the depth property but
  - We may cause an overflow (i.e., node v may become a 5-node)
- Example: inserting key 30 causes an overflow



Overflow and Split

- We handle an overflow at a 5-node v with a split operation:
  - let $v_1 \ldots v_5$ be the children of v and $k_1 \ldots k_4$ be the keys of v
  - node v is replaced nodes v' and v''
    - v' is a 3-node with keys k_1 k_2 and children v_1 v_2 v_3
    - v'' is a 2-node with key k_4 and children v_4 v_5
  - key k_3 is inserted into the parent u of v (a new root may be created)
- The overflow may propagate to the parent node u



Analysis of Insertion

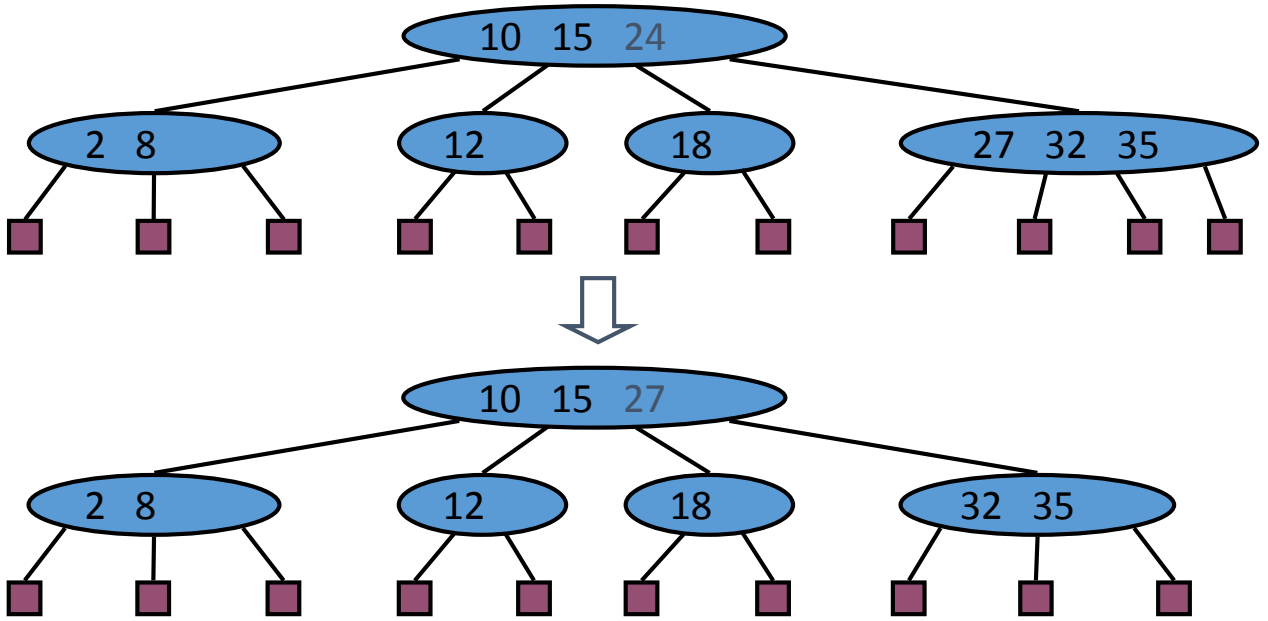
Algorithm ***put*(*k*, *o*)**

1. We search for key k to locate the insertion node v
2. We add the new entry (k, o) at node v
3. **while** ***overflow*(*v*)**
   if ***isRoot*(*v*)**
   create a new empty root above v
   $v \leftarrow$ ***split*(*v*)**

- Let T be a (2,4) tree with n items
  - Tree T has $O(\log n)$ height
  - Step 1 takes $O(\log n)$ time because we visit $O(\log n)$ nodes
  - Step 2 takes $O(1)$ time
  - Step 3 takes $O(\log n)$ time because each split takes $O(1)$ time and we perform $O(\log n)$ splits
- Thus, an insertion in a (2,4) tree takes $O(\log n)$ time

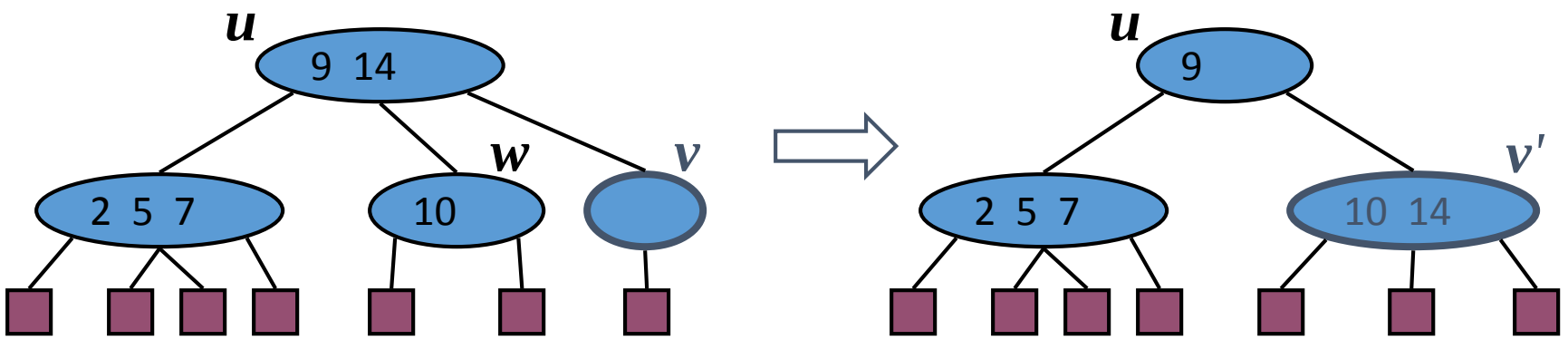
Deletion

- We reduce deletion of an entry to the case where the item is at the node with leaf children
- Otherwise, we replace the entry with its inorder successor (or, equivalently, with its inorder predecessor) and delete the latter entry
- Example: to delete key 24, we replace it with 27 (inorder successor)



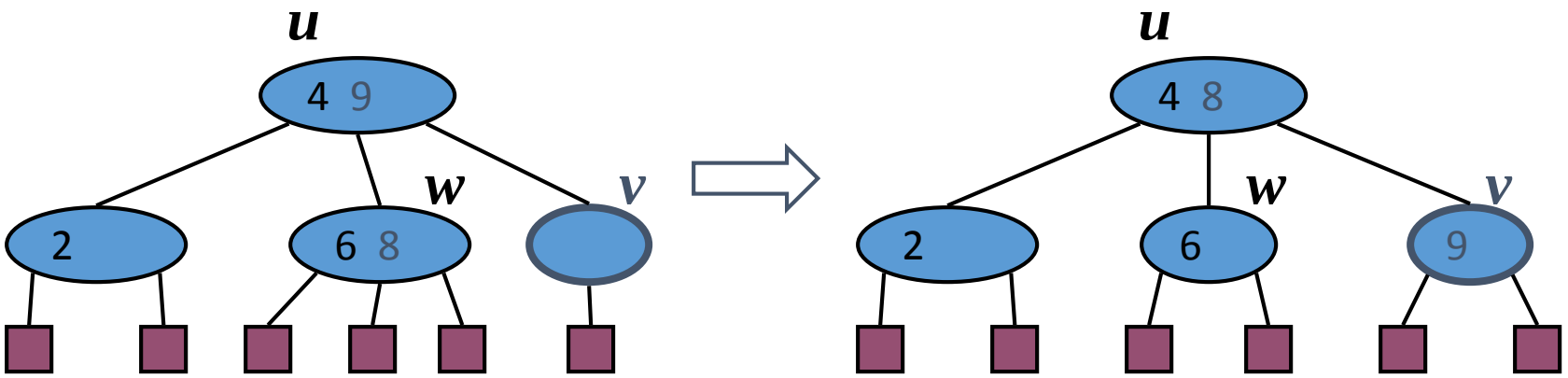
Underflow and Fusion

- Deleting an entry from a node v may cause an underflow, where node v becomes a 1-node with one child and no keys
- To handle an underflow at node v with parent u, we consider two cases
- Case 1: the adjacent siblings of v are 2-nodes
  - Fusion operation: we merge v with an adjacent sibling w and move an entry from u to the merged node v'
  - After a fusion, the underflow may propagate to the parent u



Underflow and Transfer

- To handle an underflow at node v with parent u, we consider two cases
- Case 2: an adjacent sibling w of v is a 3-node or a 4-node
  - Transfer operation:
    1. we move a child of w to v
    2. we move an item from u to v
    3. we move an item from w to u
  - After a transfer, no underflow occurs



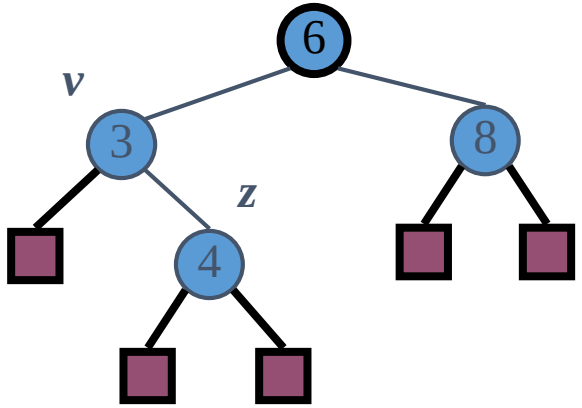
Analysis of Deletion

- Let T be a (2,4) tree with n items
  - Tree T has $O(\log n)$ height
- In a deletion operation
  - We visit $O(\log n)$ nodes to locate the node from which to delete the entry
  - We handle an underflow with a series of $O(\log n)$ fusions, followed by at most one transfer
  - Each fusion and transfer takes $O(1)$ time
- Thus, deleting an item from a (2,4) tree takes $O(\log n)$ time

Comparison of Map Implementations

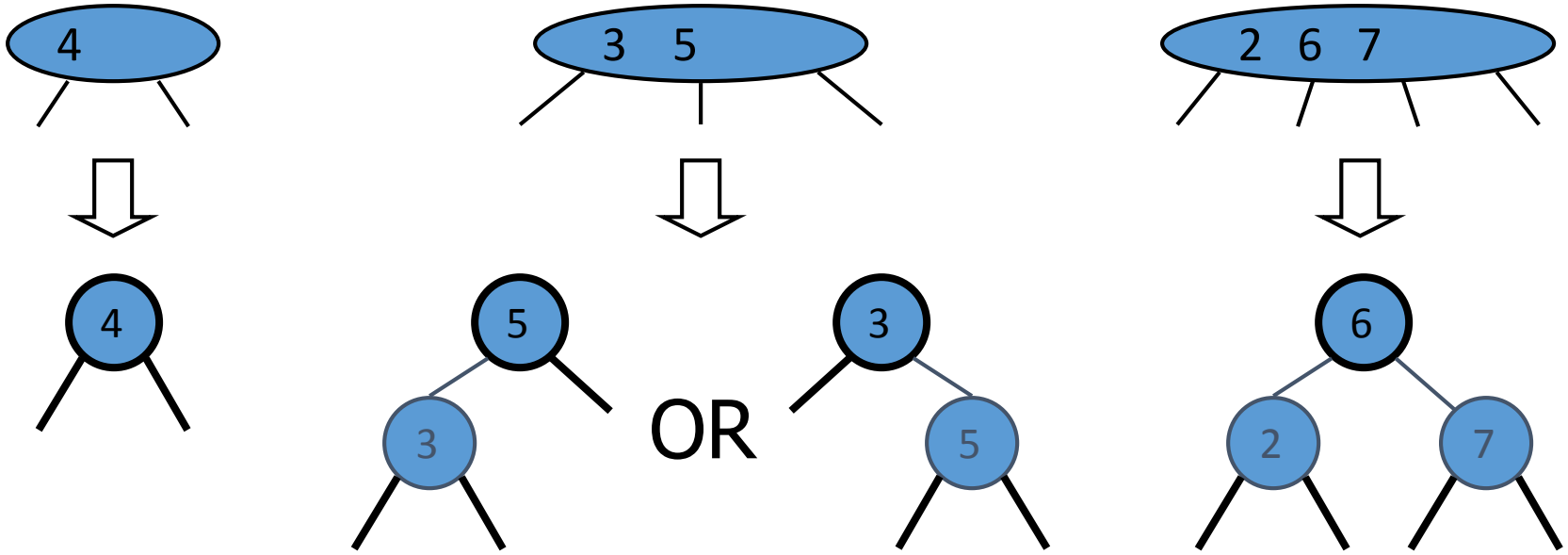
	Search	Insert	Delete	Notes
Hash Table	1 expected	1 expected	1 expected	○ no ordered map methods ○ simple to implement
Skip List	$\log n$ high prob.	$\log n$ high prob.	$\log n$ high prob.	○ randomized insertion ○ simple to implement
AVL and (2,4) Tree	$\log n$ worst-case	$\log n$ worst-case	$\log n$ worst-case	○ complex to implement

Red-Black Trees



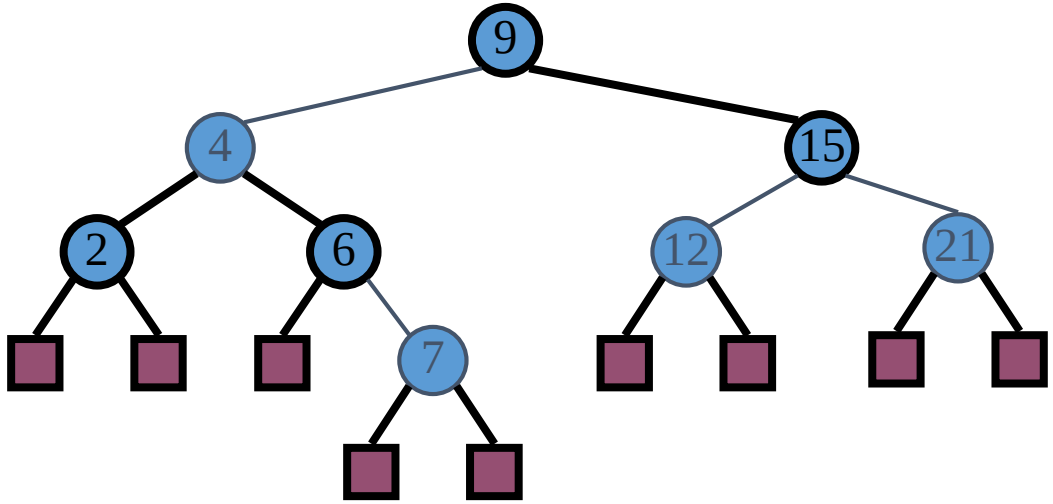
From (2,4) to Red-Black Trees

- A red-black tree is a representation of a (2,4) tree by means of a binary tree whose nodes are colored red or **black**
- In comparison with its associated (2,4) tree, a red-black tree has
  - same logarithmic time performance
  - simpler implementation with a single node type



Red-Black Trees

- A red-black tree can also be defined as a binary search tree that satisfies the following properties:
  - Root Property: the root is black
  - External Property: every leaf is black
  - Internal Property: the children of a red node are black
  - Depth Property: all the leaves have the same black depth



Height of a Red-Black Tree

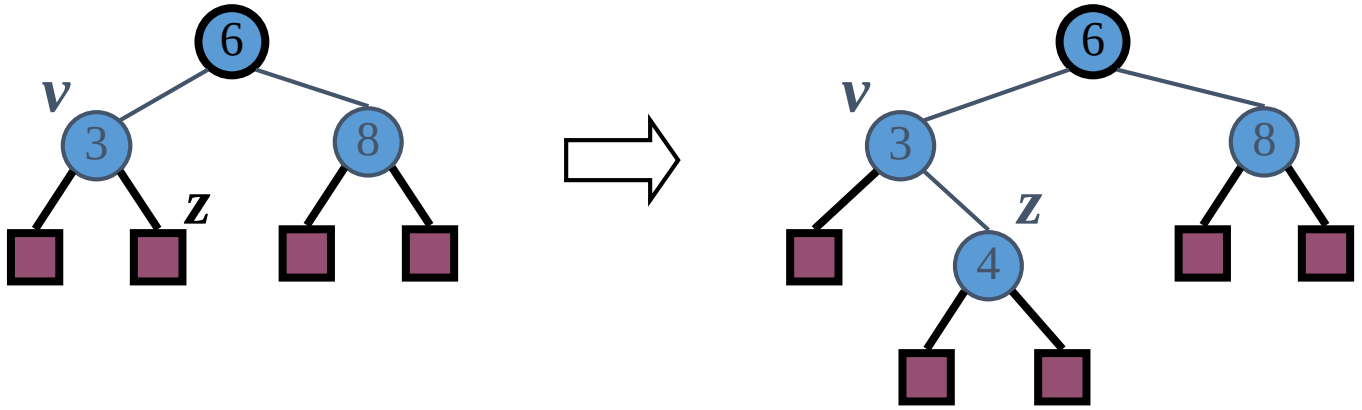
- Theorem: A red-black tree storing n items has height $O(\log n)$

  Proof:

  - The height of a red-black tree is at most twice the height of its associated (2,4) tree, which is $O(\log n)$
- The search algorithm for a binary search tree is the same as that for a binary search tree
- By the above theorem, searching in a red-black tree takes $O(\log n)$ time

Insertion

- To insert (k, o), we execute the insertion algorithm for binary search trees and color red the newly inserted node z unless it is the root
  - We preserve the root, external, and depth properties
  - If the parent v of z is black, we also preserve the internal property and we are done
  - Else (v is red) we have a double red (i.e., a violation of the internal property), which requires a reorganization of the tree
- Example where the insertion of 4 causes a double red:

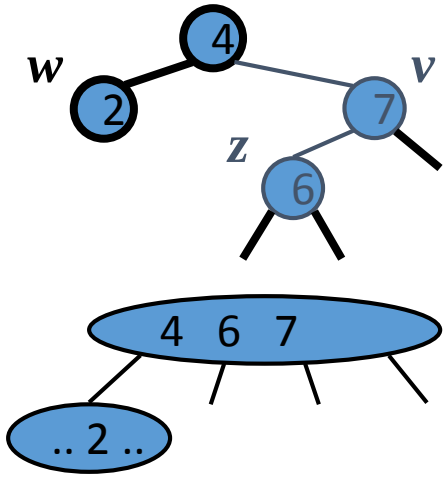


Remedying a Double Red

- Consider a double red with child z and parent v, and let w be the sibling of v

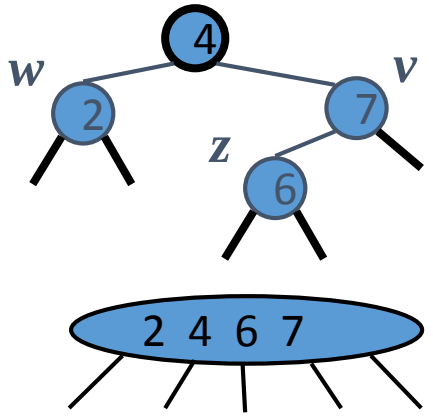
Case 1: w is black

- The double red is an incorrect replacement of a 4-node
- Restructuring: we change the 4-node replacement



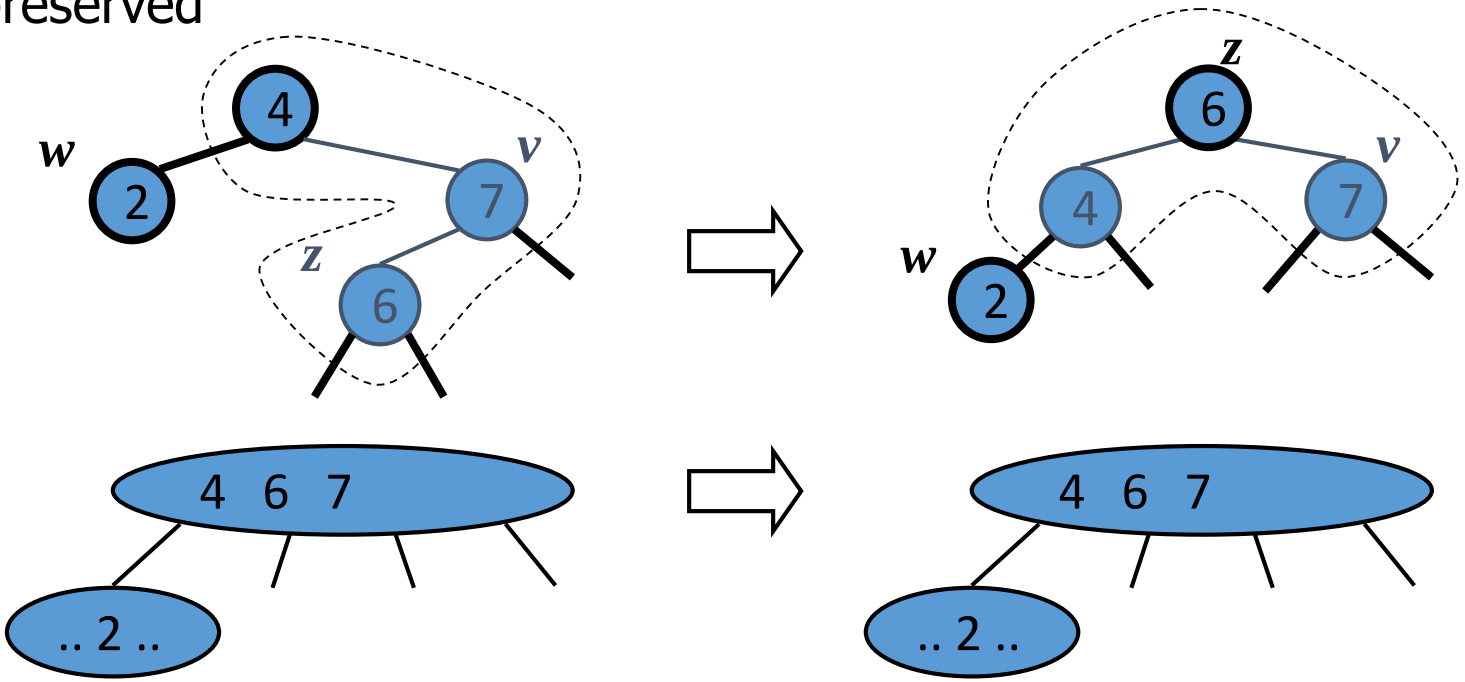
Case 2: w is red

- The double red corresponds to an overflow
- Recoloring: we perform the equivalent of a split



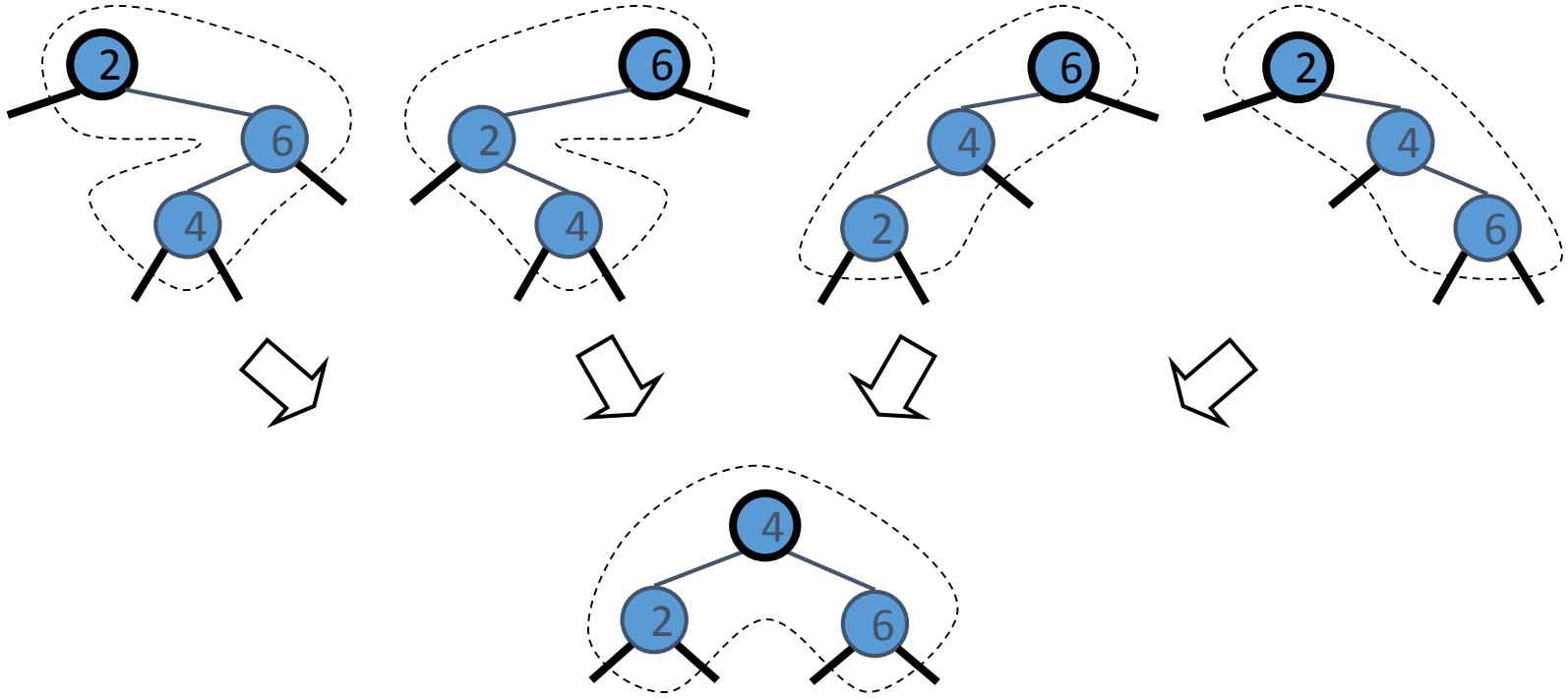
Restructuring

- A restructuring remedies a child-parent double red when the parent red node has a black sibling
- It is equivalent to restoring the correct replacement of a 4-node
- The internal property is restored and the other properties are preserved



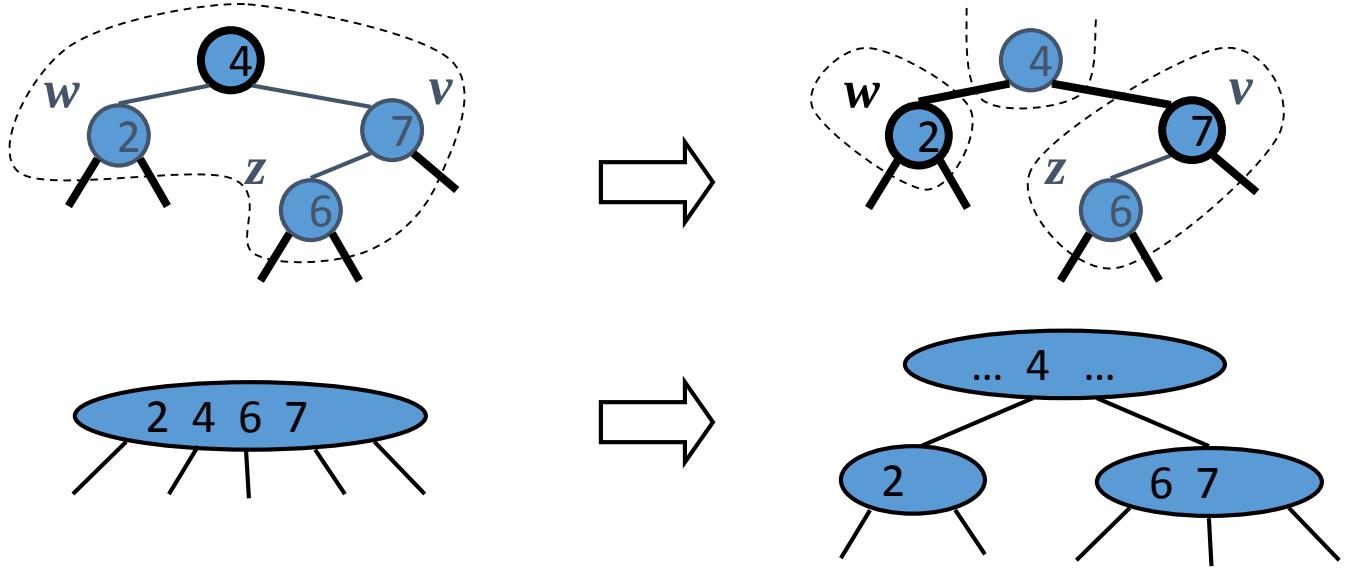
Restructuring (cont.)

- There are four restructuring configurations depending on whether the double red nodes are left or right children



Recoloring

- A recoloring remedies a child-parent double red when the parent red node has a red sibling
- The parent v and its sibling w become black and the grandparent u becomes red, unless it is the root
- It is equivalent to performing a split on a 5-node
- The double red violation may propagate to the grandparent u



Analysis of Insertion

Algorithm *insert*(k, o)

1. We search for key k to locate the insertion node z

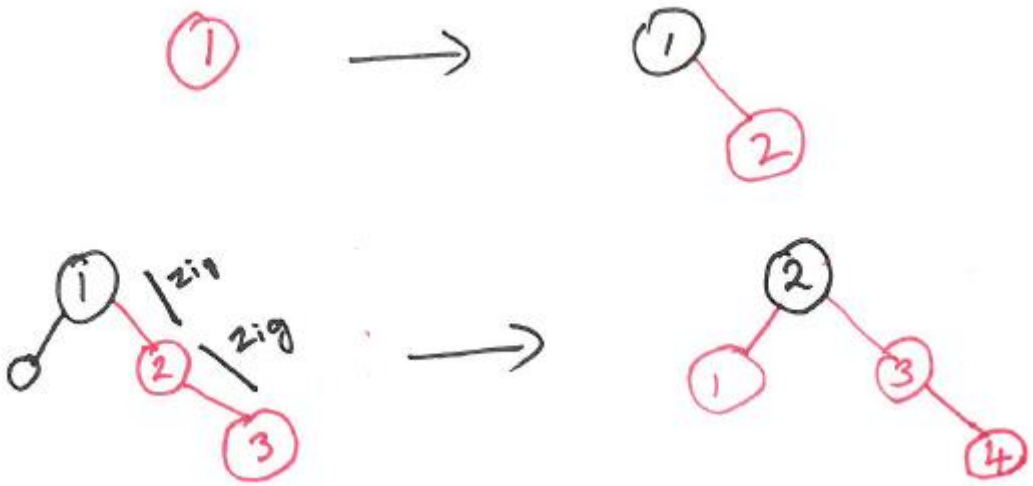
2. We add the new entry (k, o) at node z and color z red

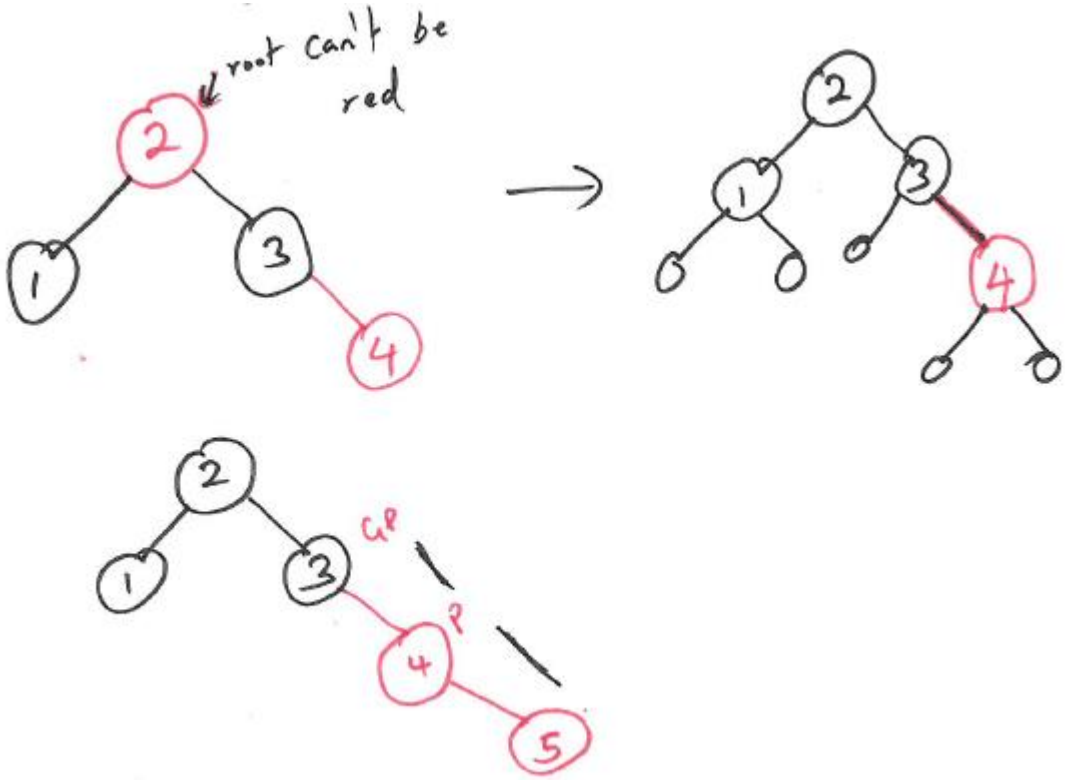
3. **while** *doubleRed*(z)
   if *isBlack*(*sibling*(*parent*(z)))
      $z \leftarrow$ *restructure*(z)
      return
   else { *sibling*(*parent*(z)) is red }
      $z \leftarrow$ *recolor*(z)

- Recall that a red-black tree has $O(\log n)$ height
- Step 1 takes $O(\log n)$ time because we visit $O(\log n)$ nodes
- Step 2 takes $O(1)$ time
- Step 3 takes $O(\log n)$ time because we perform
  - $O(\log n)$ recolorings, each taking $O(1)$ time, and
  - at most one restructuring taking $O(1)$ time
- Thus, an insertion in a red-black tree takes $O(\log n)$ time

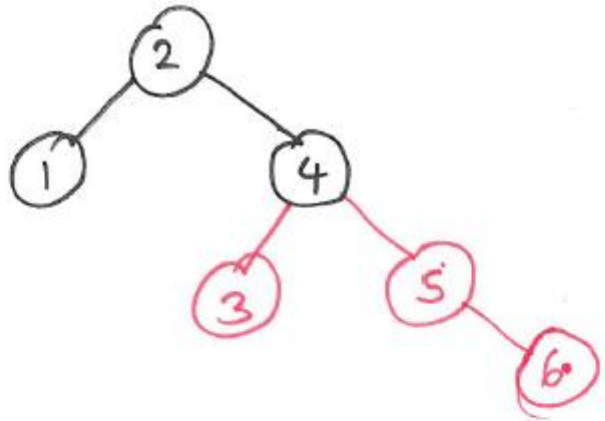
RBT – Insertion Example:

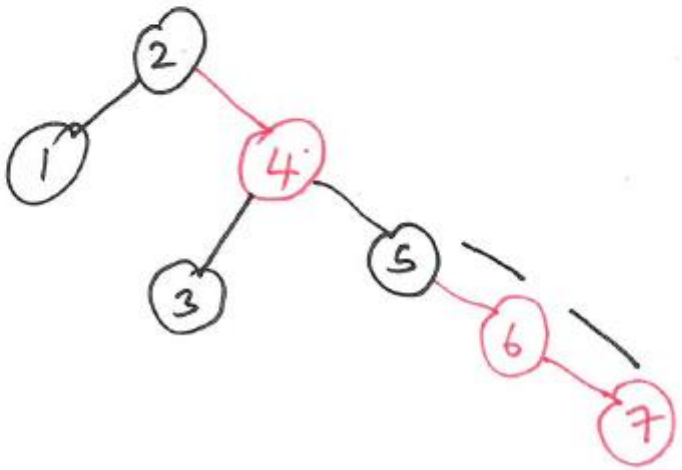
- Insert 1, 2, 3, 4, 5, 6, 7 and 8

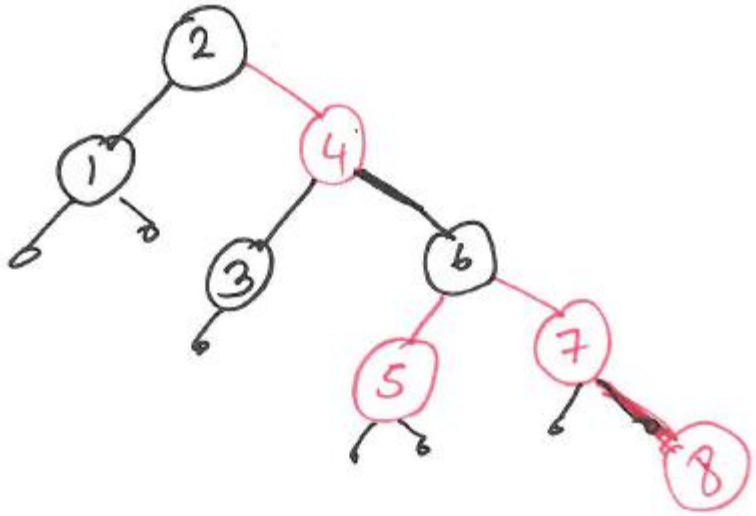




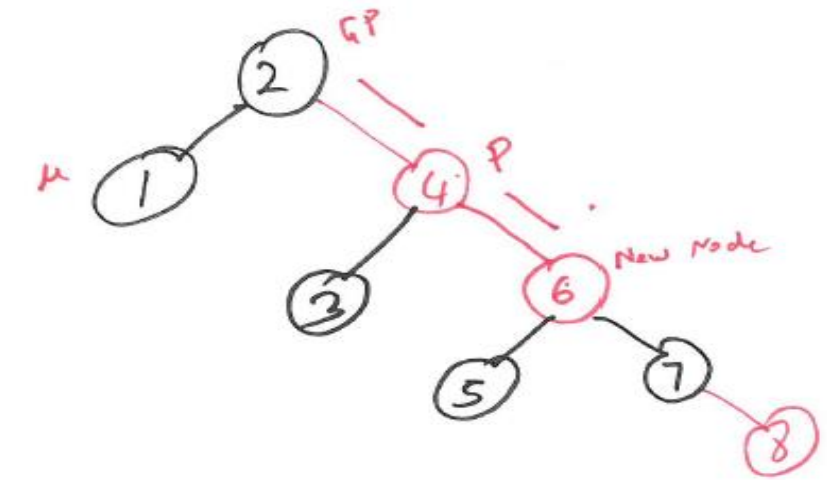
RBT – Insertion Example:

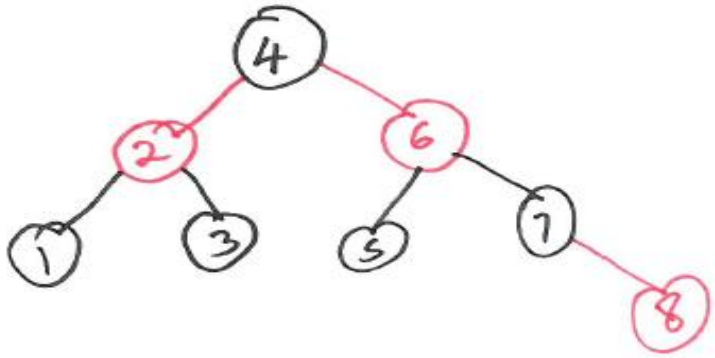






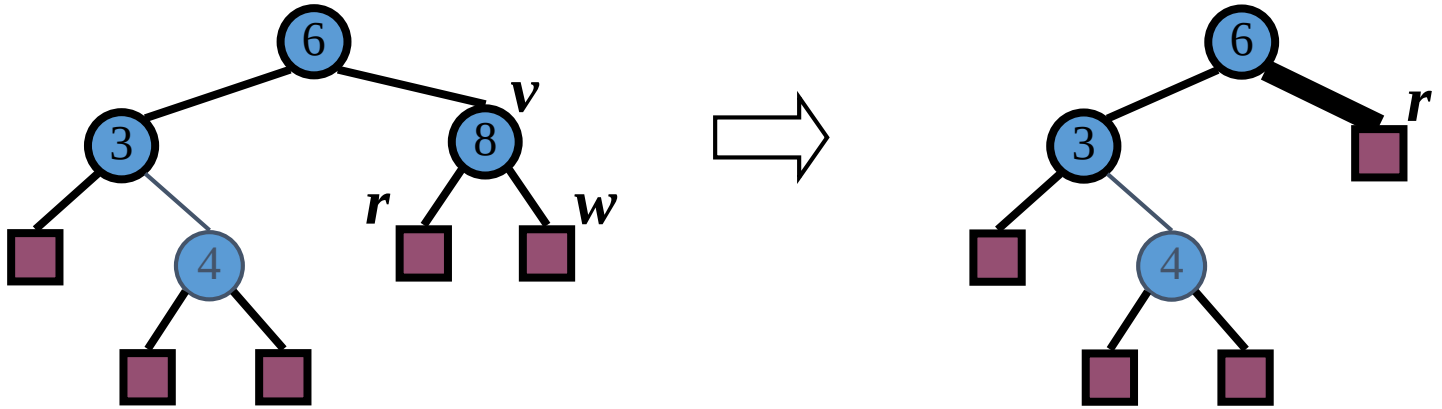
RBT – Insertion Example:





Deletion

- To perform operation remove(k), we first execute the deletion algorithm for binary search trees
- Let v be the internal node removed, w the external node removed, and r the sibling of w
  - If either v or r was red, we color r black and we are done
  - Else (v and r were both black) we color r ***double black***, which is a violation of the internal property requiring a reorganization of the tree
- Example where the deletion of 8 causes a double black:



Remedying a Double Black

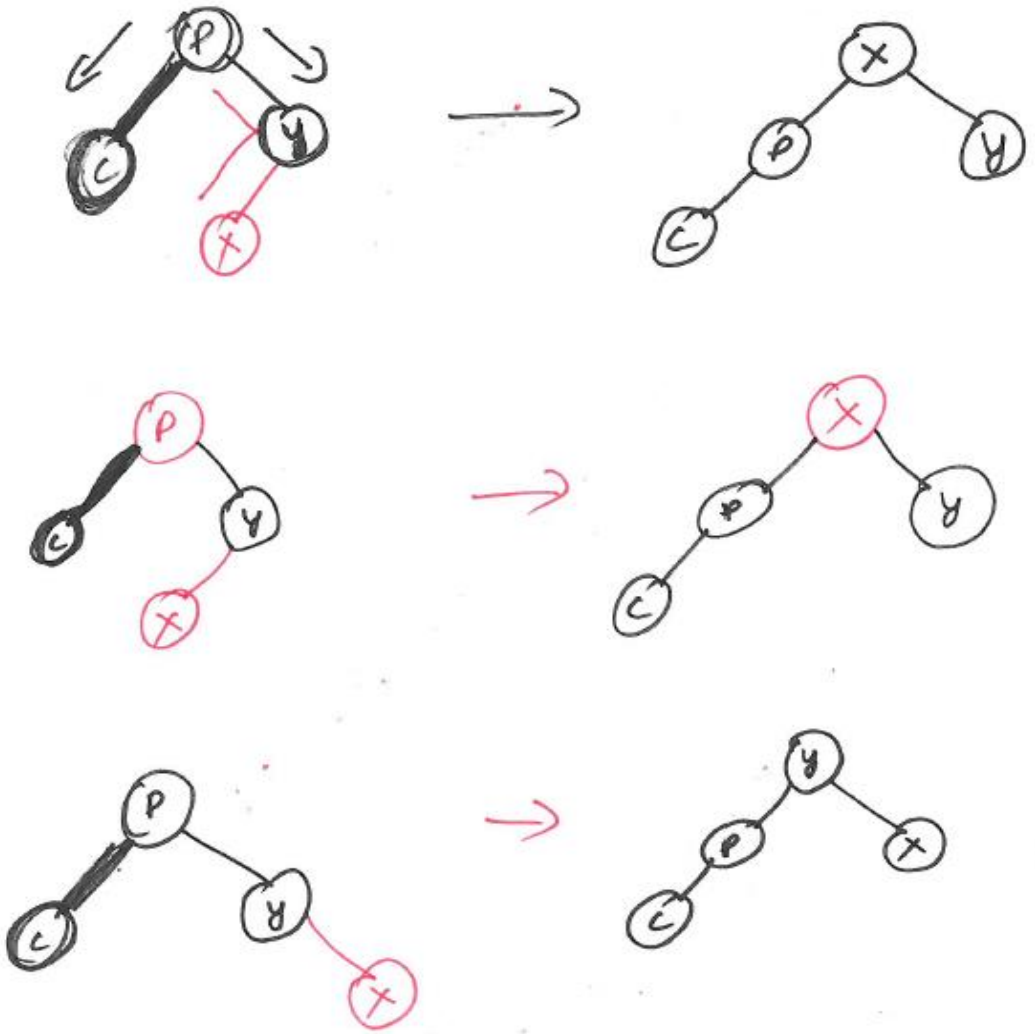
- The algorithm for remedying a double black node w with sibling y considers three cases

  Case 1: y is black and has a red child
  - We perform a restructuring, equivalent to a transfer , and we are done

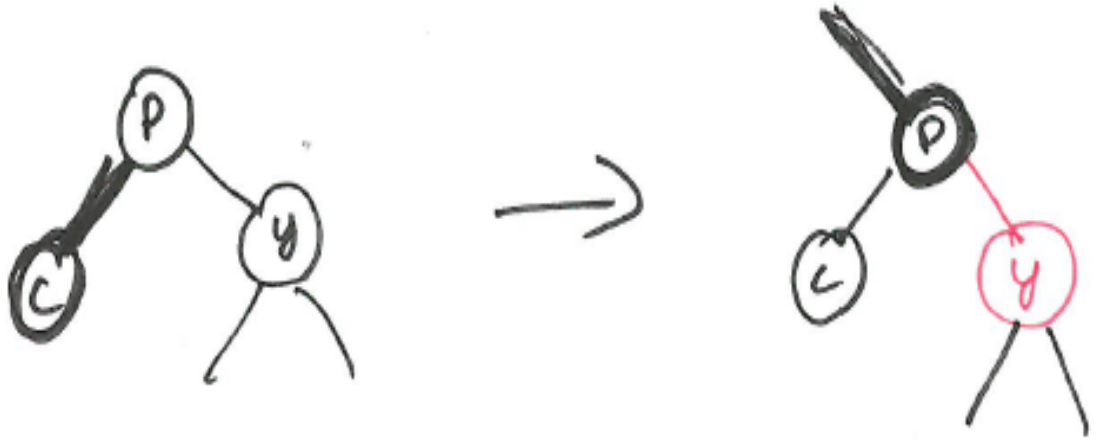
  Case 2: y is black and its children are both black
  - We perform a recoloring, equivalent to a fusion, which may propagate up the double black violation

  Case 3: y is red
  - We perform an adjustment, equivalent to choosing a different representation of a 3-node, after which either Case 1 or Case 2 applies
- Deletion in a red-black tree takes $O(\log n)$ time

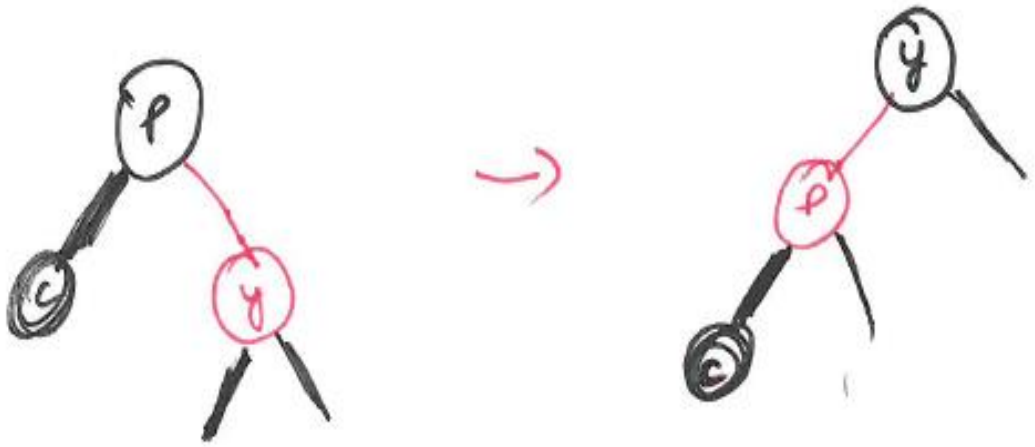
Case 1: y is black and has a red child



Case 2: y is black and its children are both black



Case 3: y is red



Red-Black Tree Reorganization

Insertion	remedy double red	
Red-black tree action	**(2,4) tree action**	**result**
restructuring	change of 4-node representation	double red removed
recoloring	split	double red removed or propagated up

Deletion	remedy double black	
Red-black tree action	**(2,4) tree action**	**result**
restructuring	transfer	double black removed
recoloring	fusion	double black removed or propagated up
adjustment	change of 3-node representation	restructuring or recoloring follows

Java Implementation

```
1  /** An implementation of a sorted map using a red-black tree. */
2  public class RBTreeMap<K,V> extends TreeMap<K,V> {
3    /** Constructs an empty map using the natural ordering of keys. */
4    public RBTreeMap( ) { super( ); }
5    /** Constructs an empty map using the given comparator to order keys. */
6    public RBTreeMap(Comparator<K> comp) { super(comp); }
7    // we use the inherited aux field with convention that 0=black and 1=red
8    // (note that new leaves will be black by default, as aux=0)
9    private boolean isBlack(Position<Entry<K,V>> p) { return tree.getAux(p)==0;}
10   private boolean isRed(Position<Entry<K,V>> p) { return tree.getAux(p)==1; }
11   private void makeBlack(Position<Entry<K,V>> p) { tree.setAux(p, 0); }
12   private void makeRed(Position<Entry<K,V>> p) { tree.setAux(p, 1); }
13   private void setColor(Position<Entry<K,V>> p, boolean toRed) {
14     tree.setAux(p, toRed ? 1 : 0);
15   }
16   /** Overrides the TreeMap rebalancing hook that is called after an insertion. */
17   protected void rebalanceInsert(Position<Entry<K,V>> p) {
18     if (!isRoot(p)) {
19       makeRed(p);                    // the new internal node is initially colored red
20       resolveRed(p);                 // but this may cause a double-red problem
21     }
22   }
```

Java Implementation, 2

```
23   /** Remedies potential double-red violation above red position p. */
24   private void resolveRed(Position<Entry<K,V>> p) {
25     Position<Entry<K,V>> parent,uncle,middle,grand; // used in case analysis
26     parent = parent(p);
27     if (isRed(parent)) {                          // double-red problem exists
28       uncle = sibling(parent);
29       if (isBlack(uncle)) {                       // Case 1: misshapen 4-node
30         middle = restructure(p);                  // do trinode restructuring
31         makeBlack(middle);
32         makeRed(left(middle));
33         makeRed(right(middle));
34       } else {                                    // Case 2: overfull 5-node
35         makeBlack(parent);                        // perform recoloring
36         makeBlack(uncle);
37         grand = parent(parent);
38         if (!isRoot(grand)) {
39           makeRed(grand);                         // grandparent becomes red
40           resolveRed(grand);                      // recur at red grandparent
41         }
42       }
43     }
44   }
```

Java Implementation, 3

```
45    /** Overrides the TreeMap rebalancing hook that is called after a deletion. */
46    protected void rebalanceDelete(Position<Entry<K,V>> p) {
47      if (isRed(p))                                          // deleted parent was black
48        makeBlack(p);                                        // so this restores black depth
49      else if (!isRoot(p)) {
50        Position<Entry<K,V>> sib = sibling(p);
51        if (isInternal(sib) && (isBlack(sib) || isInternal(left(sib))))
52          remedyDoubleBlack(p);                // sib's subtree has nonzero black height
53      }
54    }
55
```

Java Implementation, 4

```
56    /** Remedies a presumed double-black violation at the given (nonroot) position. */
57    private void remedyDoubleBlack(Position<Entry<K,V>> p) {
58      Position<Entry<K,V>> z = parent(p);
59      Position<Entry<K,V>> y = sibling(p);
60      if (isBlack(y)) {
61        if (isRed(left(y)) || isRed(right(y))) {          // Case 1: trinode restructuring
62          Position<Entry<K,V>> x = (isRed(left(y)) ? left(y) : right(y));
63          Position<Entry<K,V>> middle = restructure(x);
64          setColor(middle, isRed(z)); // root of restructured subtree gets z's old color
65          makeBlack(left(middle));
66          makeBlack(right(middle));
67        } else {                                          // Case 2: recoloring
68          makeRed(y);
69          if (isRed(z))
70            makeBlack(z);                                 // problem is resolved
71          else if (!isRoot(z))
72            remedyDoubleBlack(z);                         // propagate the problem
73        }
74      } else {                                            // Case 3: reorient 3-node
75        rotate(y);
76        makeBlack(y);
77        makeRed(z);
78        remedyDoubleBlack(p);                             // restart the process at p
79      }
80    }
81  }
```