Chapter 12

Binary Tree

12.1 Introduction

12.2 Representation of binary tree

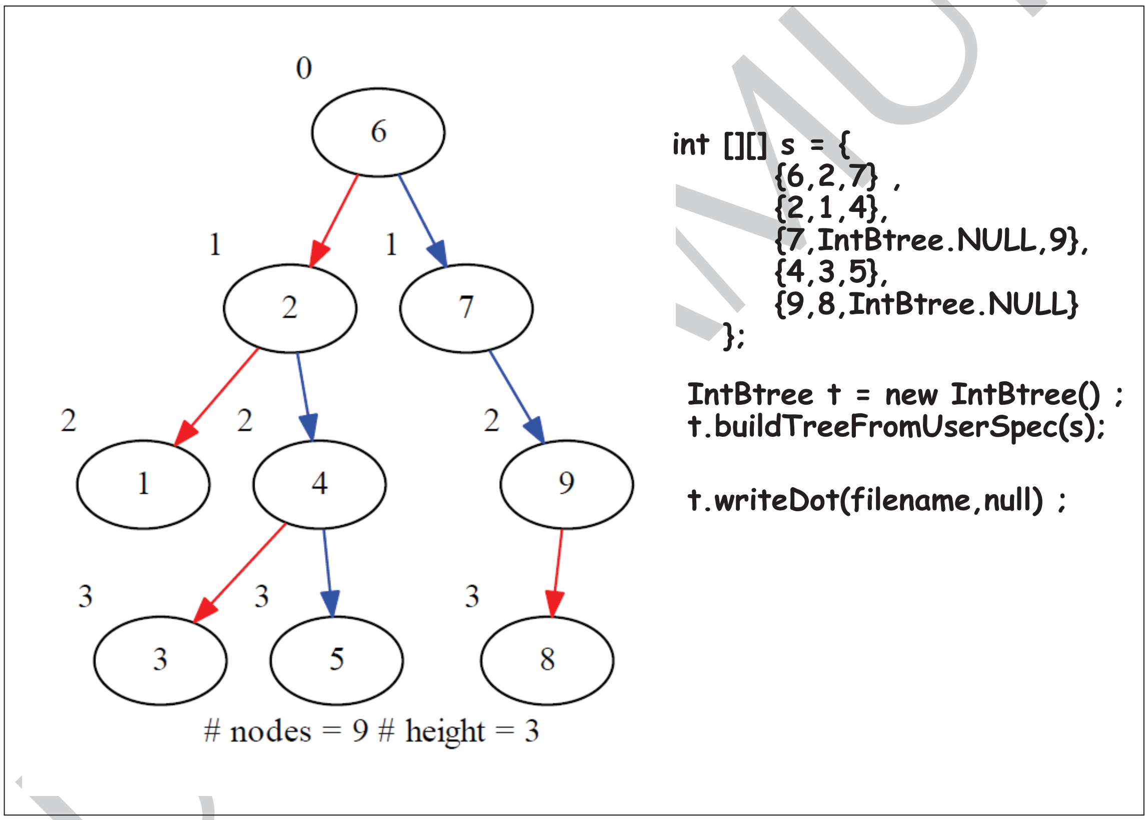
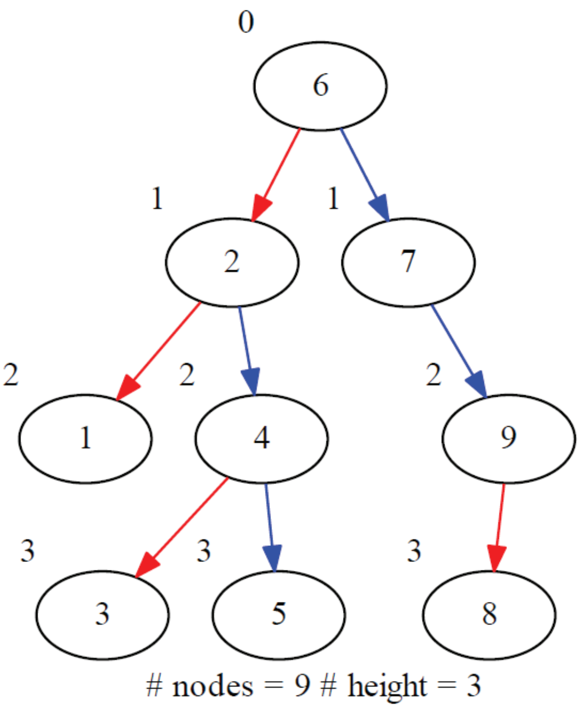


Figure 12.1: Representation of binary tree of **int**




```
int [][] s = {
        {2,1,4},
        {4,3,5},
        {9,8,IntBtree.NULL},
        {7,IntBtree.NULL,9},
        {6,2,7}
    };

    IntBtree b = new IntBtree(t);
    t.buildTreeFromUserSpec(s) ;
```

Figure 12.2: To illustrate **root** is automatically detected

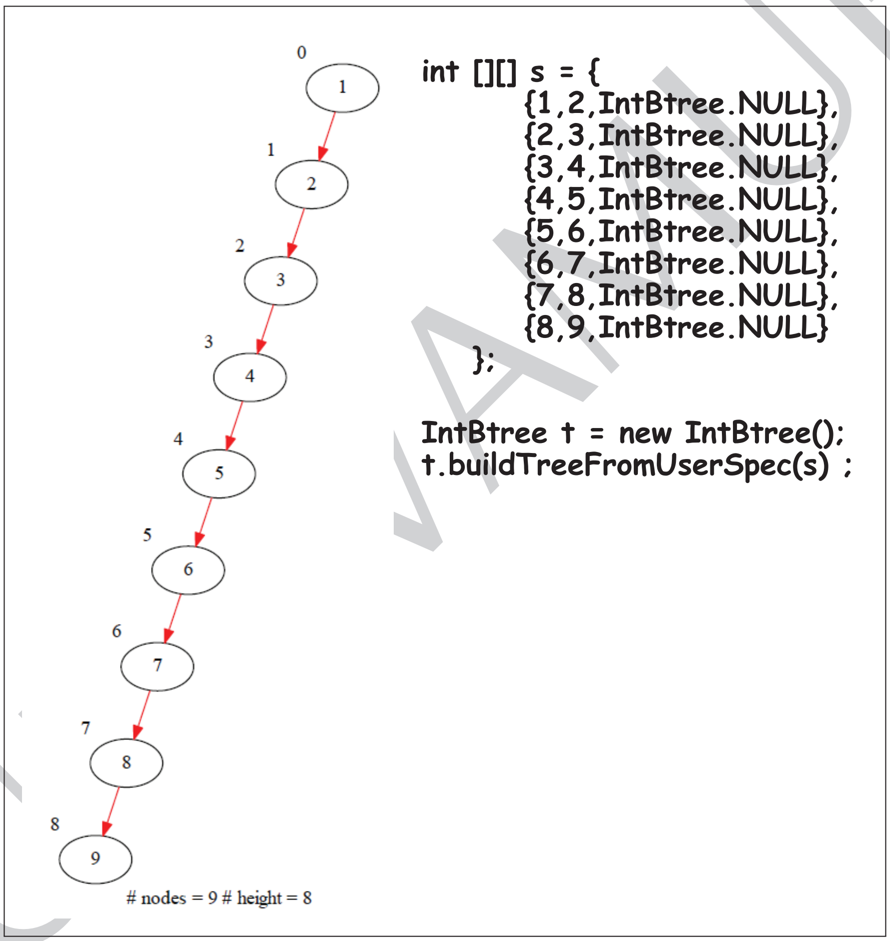


Figure 12.3: To illustrate **linked list** is also a **tree**

12.3 Data structure of a binary tree


```
protected class node {
   protected int d;
   protected int depth;
   protected node father;
   protected node left;
   protected node right;

   private node(int x) {
     d = x;
     depth = 0;
     father = null;
     left = null;
     right = null;
   }
 }
```

```
class IntBtree {
  protected node root;
  private int num;
  private int height ;
  //Largest depth of any node is the height of the tree
  protected static final IntUtil u = new IntUtil();
  public static final int NULL = -99999999;
```

```
public IntBtree() {
   root = null;
   num = 0;
   height = 0 ;
}
```

```
public int height() {
  return this.height ;
}
```

```
public int size() {
  return num;
}
```

```
public boolean isLeaf(node n) {
  return (n.left == null && n.right == null) ? true : false
}
```

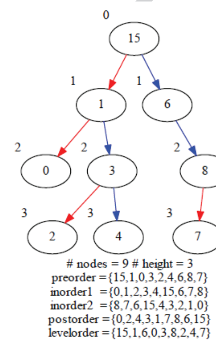
```
private void depth_r(node r) {
    if (r != null) {
      if (r == root) {
        r.depth = 0;
      } else {
        r.depth = r.father.depth + 1;
        if (r.depth > height) {
          height = r.depth ;
        }
      }
      depth_r(r.left);
      depth_r(r.right);
    }
  }
public void computeDepth() {
   depth_r(root);
 }
```

```
protected node isOneKid(node n) {
   if (n.left == null && n.right != null) {
     return n.right;
   }
   if (n.left != null && n.right == null) {
     return n.left;
   }
   return null ;
 }
```

Figure 12.4: Data structure of a binary tree

```
public void writeDot(String fname, String info) {
  if (root != null) {
    try {
      FileWriter o = new FileWriter(fname);
      computeDepth();
      o.write("## Jagadeesh Vasudevamurthy ####\n");
      o.write("## dot -Tpdf " + fname + " -o " + fname + ".pdf\n");
      o.write("digraph g {\n");

      /* make label */
      String label = " label = ";
      label = label + "\" " + " # nodes = " + size() + " # height = " + height() ;
      if (info != null) {
        label = label + " " + info ;
      }
      label = label +  "\"\n";
      o.write(label);
      Queue<node> q = new LinkedList();
      q.add(root);
      int nk = 0; // null kount
      while (q.isEmpty() == false) {
        node n = q.remove();
        if (n.left == null && n.right == null) {
          o.write(" " + n.d + "[xlabel = \"" + n.depth + "\"]");
          continue;
        }
        if (n.left == null) {
          String nulls = " null" + nk++;
          o.write(nulls + " [shape=point  style=invis]\n");
          o.write(" " + n.d + " ->" + nulls + " [color=red  style=invis]\n");
        } else {
          o.write(" " + n.d + " ->" + n.left.d + " [color=red]\n");
          o.write(" " + n.d + "[xlabel = \"" + n.depth + "\"]");
          q.add(n.left);
        }
        if (n.right == null) {
          String nulls = " null" + nk++;
          o.write(nulls + " [shape=point  style=invis]\n");
          o.write(" " + n.d + " ->" + nulls + " [color=blue  style=invis]\n");
        } else {
          o.write(" " + n.d + " ->" + n.right.d + " [color=blue]\n");
          o.write(" " + n.d + "[xlabel = \"" + n.depth + "\"]");
          q.add(n.right);
        }
      }
      o.write("}\n");
      o.close();
      System.out.println("You can see dot file at " + fname);
      System.out.println("Run the following command to get pdf file");
      System.out.println("dot -Tpdf " + fname + " -o " + fname + ".pdf");
    } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }
}
```




```
digraph g {
 label = "  # nodes = 9 # height = 3
 preorder ={15,1,0,3,2,4,6,8,7}
 inorder1  ={0,1,2,3,4,15,6,7,8}
 inorder2  ={8,7,6,15,4,3,2,1,0}
 postorder ={0,2,4,3,1,7,8,6,15}
 levelorder ={15,1,6,0,3,8,2,4,7}"
 15 ->1 [color=red]
 15[xlabel = "0"] 15 ->6 [color=blue]
 15[xlabel = "0"] 1 ->0 [color=red]
 1[xlabel = "1"] 1 ->3 [color=blue]
 1[xlabel = "1"] null0 [shape=point  style=invis]
 6 -> null0 [color=red  style=invis]
 6 ->8 [color=blue]
 6[xlabel = "1"] 0[xlabel = "2"] 3 ->2 [color=red]
 3[xlabel = "2"] 3 ->4 [color=blue]
 3[xlabel = "2"] 8 ->7 [color=red]
 8[xlabel = "2"] null1 [shape=point  style=invis]
 8 -> null1 [color=blue  style=invis]
 2[xlabel = "3"] 4[xlabel = "3"] 7[xlabel = "3"]
}
```

Figure 12.5: Printing binary tree as a **dot** file

12.4 Tree traversal

12.4.1 Preorder tree traversal

12.4.1.1 Preorder tree traversal using recursion

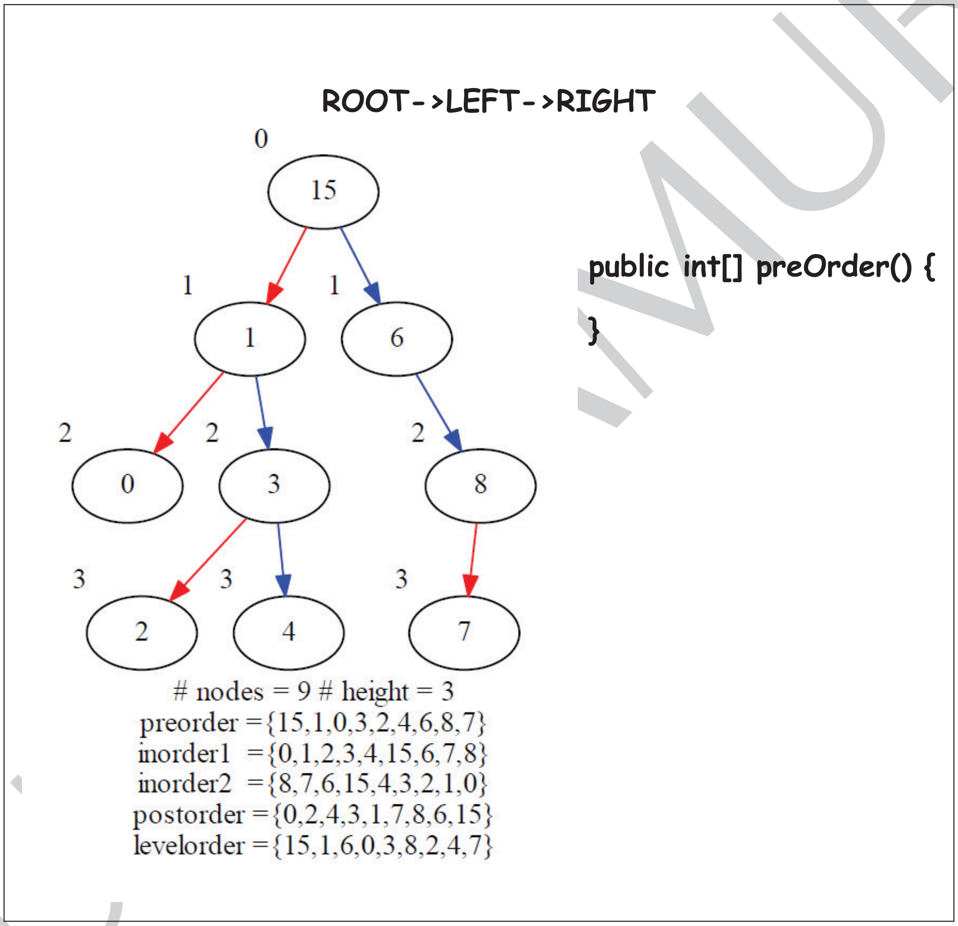
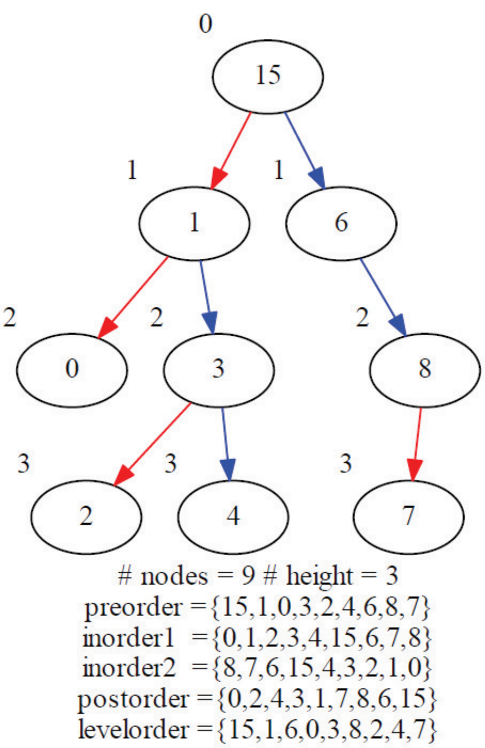


Figure 12.6: Preorder tree traversal

12.4.1.2 Preorder tree traversal without using recursion



ROOT->LEFT->RIGHT

```
Stack s ;

s.push(root);

while (s.empty() == false) {
    node t = s.pop() ;
    print t ;
    if (t.right != null) {
        s.push(t.right);
    }
    if (t.left != null) {
        s.push(t.left);
    }
}
```

Figure 12.7: Preorder tree traversal

12.4.2 Inorder tree traversal

12.4.2.1 Inorder tree traversal using recursion

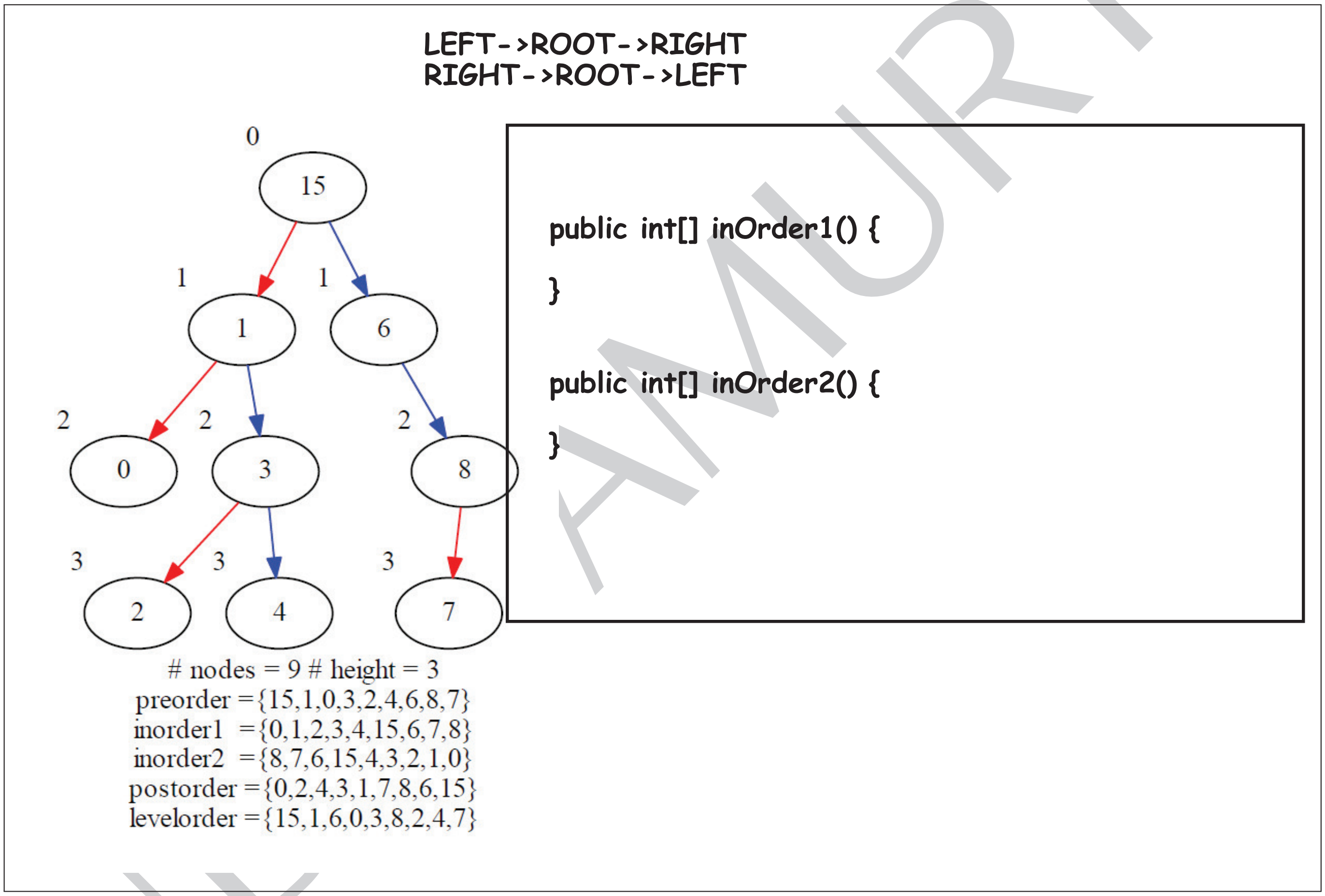
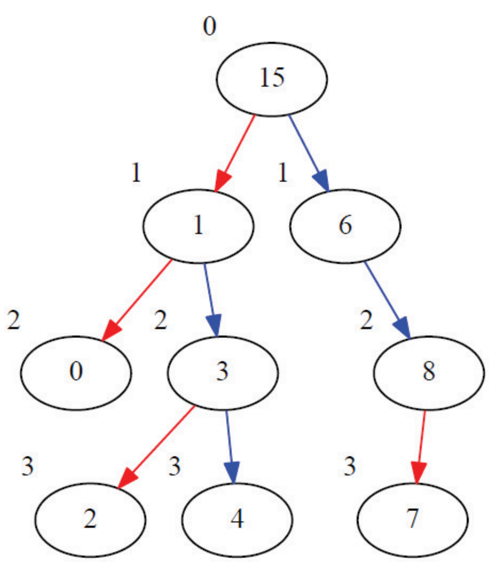


Figure 12.8: Inorder tree traversal

12.4.2.2 Inorder tree traversal without using recursion

LEFT->ROOT->RIGHT
RIGHT->ROOT->LEFT



# nodes = 9 # height = 3
preorder ={15,1,0,3,2,4,6,8,7}
inorder1 ={0,1,2,3,4,15,6,7,8}
inorder2 ={8,7,6,15,4,3,2,1,0}
postorder ={0,2,4,3,1,7,8,6,15}
levelorder ={15,1,6,0,3,8,2,4,7}

Left, Root, Right

```
Stack s ;
node t = root ;
do {
      if (t != null) {
        s.push(t) ;
        t = t.left ;
      }else {
        if (s.empty()) {
            done ;
            return ;
        }else {
            t = s.pop() ;
            print t ;
            t = t.right ;
        }
      }
 }
```

Figure 12.9: Inorder tree traversal

12.4.3 Postorder tree traversal

12.4.3.1 Postorder tree traversal using recursion

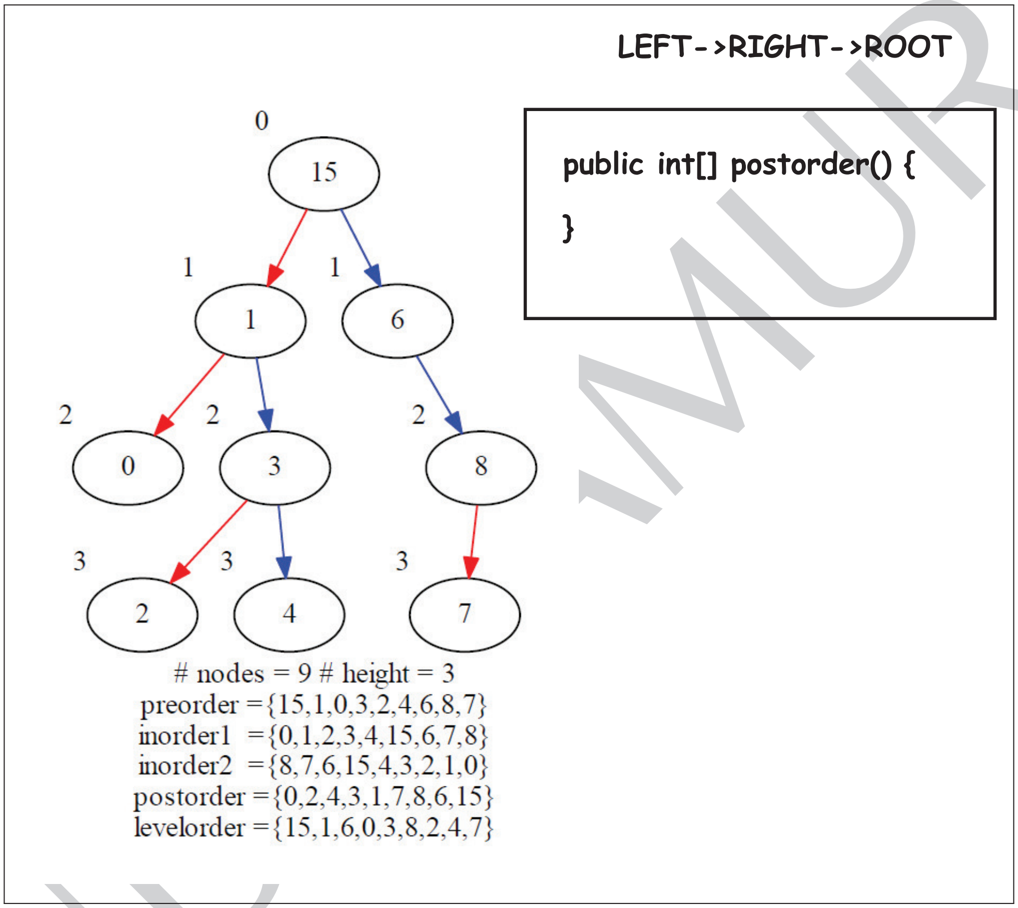
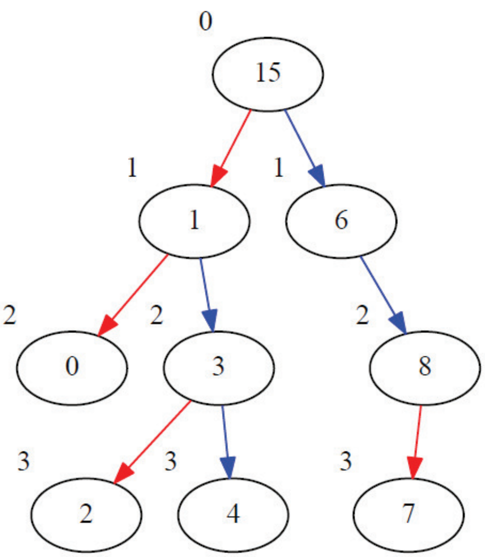


Figure 12.10: Postorder tree traversal

12.4.3.2 Postorder tree traversal without using recursion

LEFT->RIGHT->ROOT



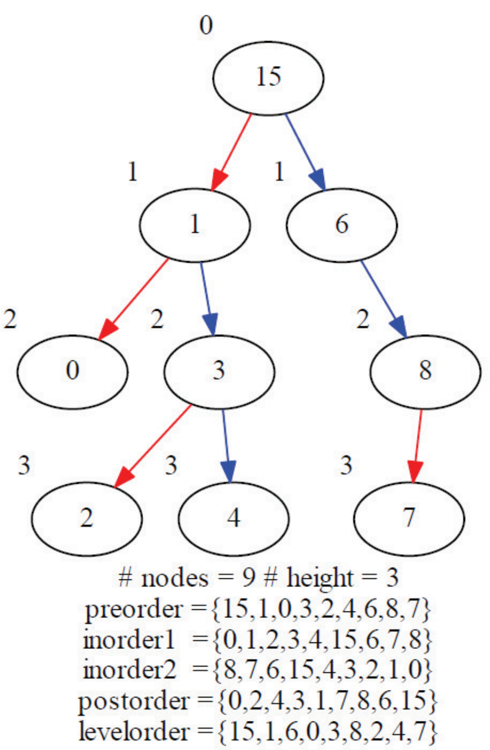
\# nodes = 9 # height = 3
preorder ={15,1,0,3,2,4,6,8,7}
inorder1 ={0,1,2,3,4,15,6,7,8}
inorder2 ={8,7,6,15,4,3,2,1,0}
postorder ={0,2,4,3,1,7,8,6,15}
levelorder ={15,1,6,0,3,8,2,4,7}

```
Stack s1 ;
Stack s2 ;
s1.push(root) ;
while (s1.empty() == false) {
    node t = s1.pop() ;
    s2.push(t) ;
    if (t.left != null) {
      s1.push(t.left) ;
    }
    if (t.right != null) {
      s1.push(t.right) ;
   }
}

while (s2.empty() == false) {
   node t = s2.pop() ;
   print t ;
}
```

Figure 12.11: Postorder tree traversal

12.4.4 Levelorder tree traversal




```
public int[] levelorder() {
}
```

Figure 12.12: Levelorder tree traversal

12.4.5 Tree traversal for a tree that has only left children

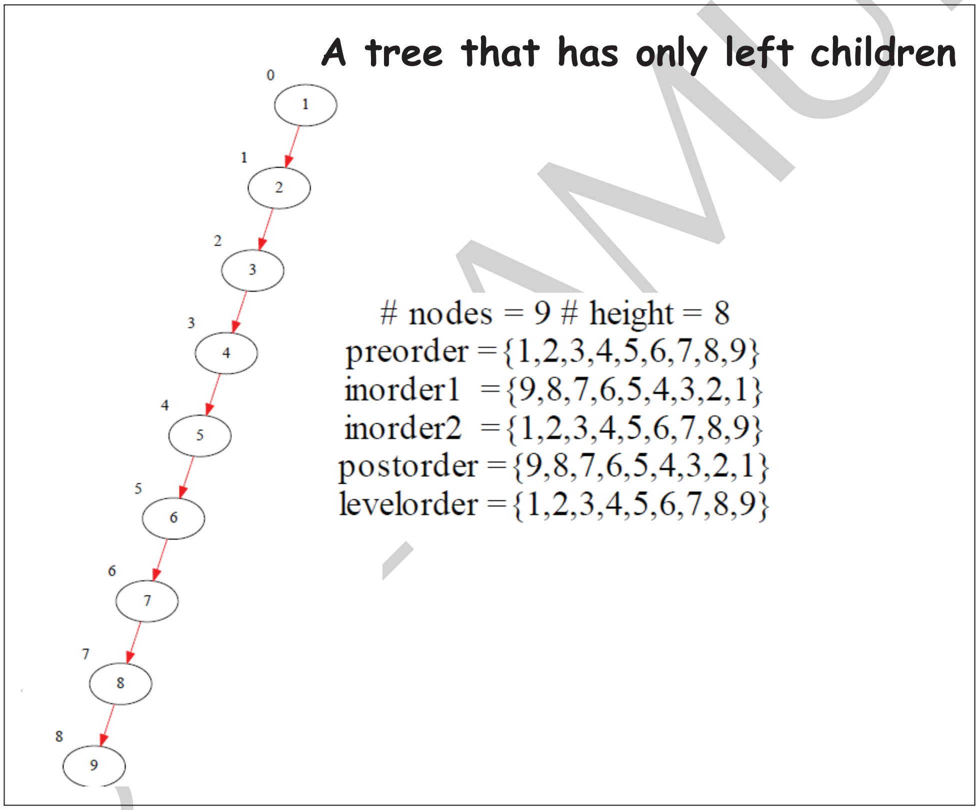


Figure 12.13: Tree traversal for a tree that has only left children

12.5 Application of tree traversal

12.5.1 Preorder tree traversal

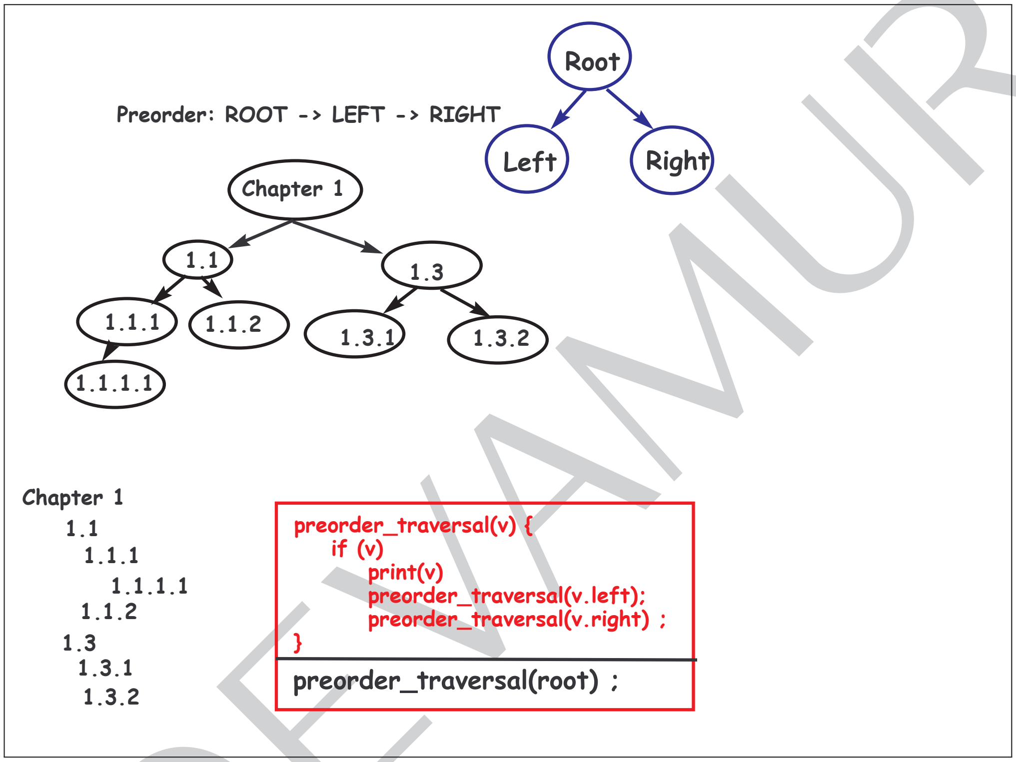


Figure 12.14: Need for preorder tree traversal

12.5.2 Postorder tree traversal

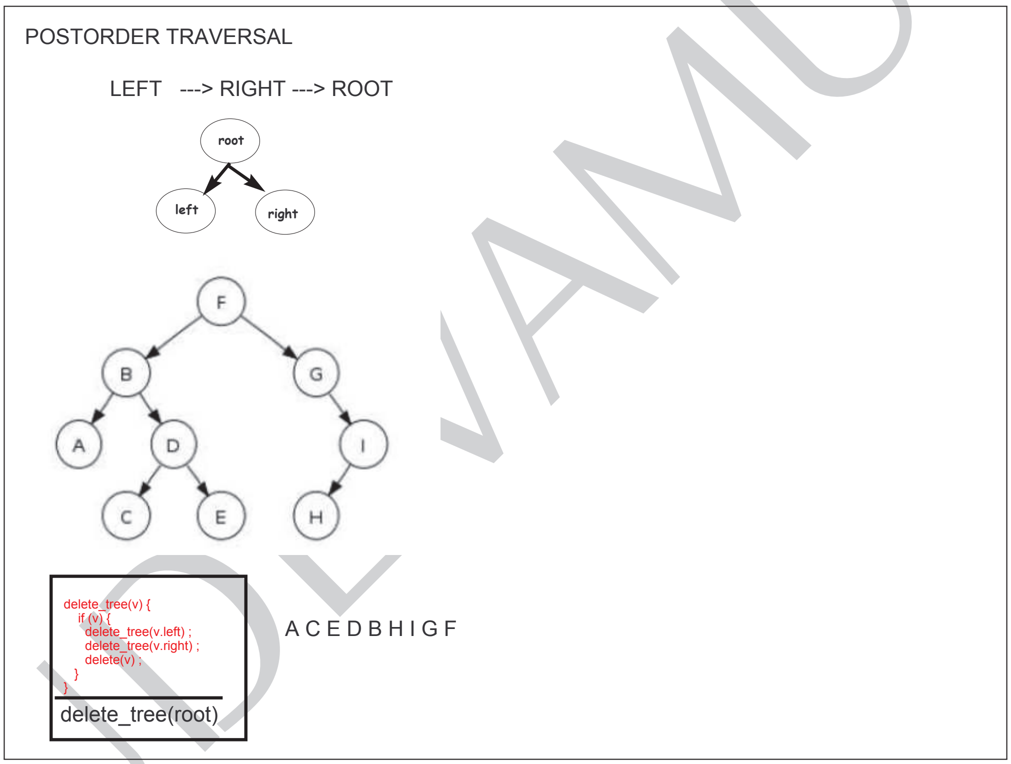


Figure 12.15: Need for postorder tree traversal

12.5.3 Inorder tree traversal

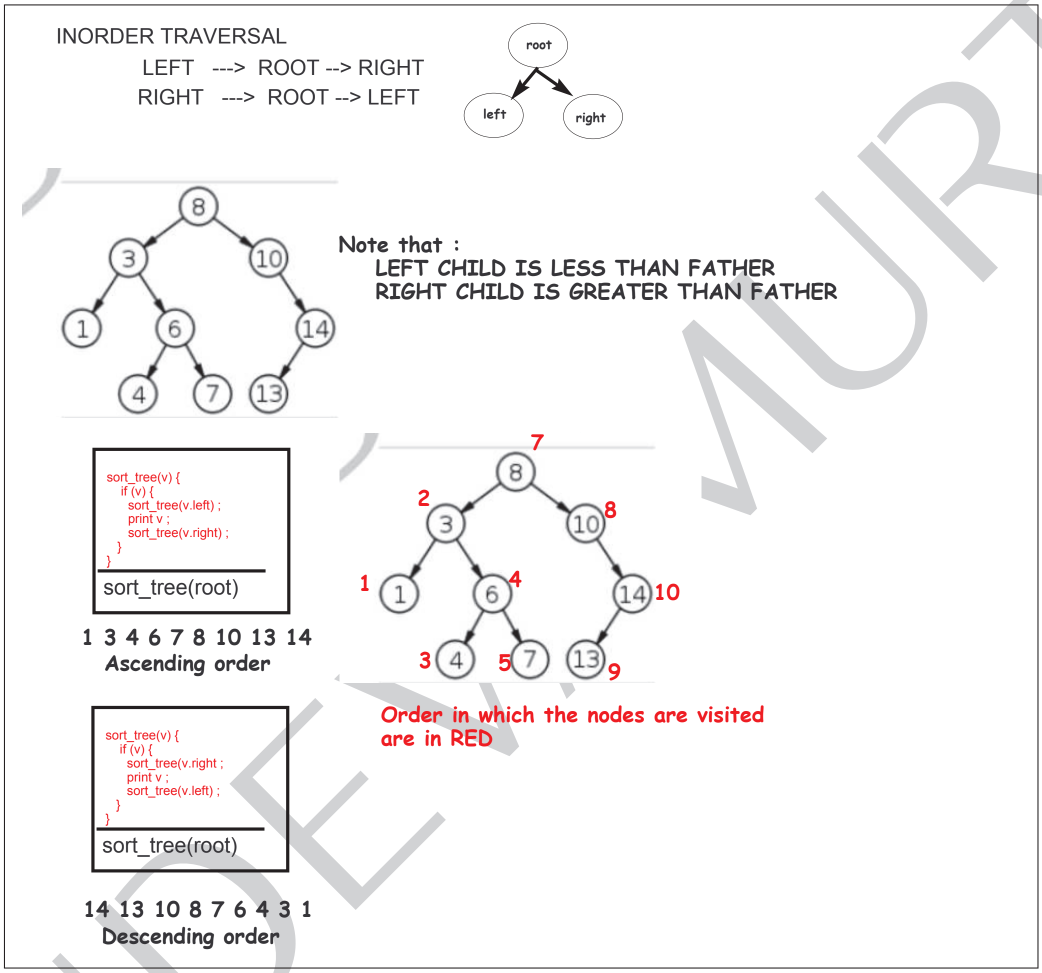


Figure 12.16: Need for inorder tree traversal

12.6 Quiz

1

A perfect binary tree has every leaf on the same level, and every nonleaf node has two children. A perfect binary tree with k leaves contains how many nodes?

(A) k
(B) k^2
(C) 2^k
(D) $\log_2 k$
(E) $2k - 1$

2

The level of a node is the length of the path (or number of edges) from the root to that node. The level of a tree is equal to the level of its deepest leaf. A binary tree has level k. Which represents

1. The maximum possible number of nodes, and
2. The minimum possible number of nodes in the tree?

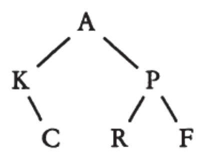
(A)	(1) 2^{k+1}	(2) $2^k + 1$
(B)	(1) 2^{k+1}	(2) k
(C)	(1) $2^{k+1} - 1$	(2) k
(D)	(1) $2^{k+1} - 1$	(2) $k + 1$
(E)	(1) $2^k + 1$	(2) 2^k

Figure 12.17: quiz

3

The binary tree shown is traversed preorder. During the traversal, each element, when accessed, is pushed onto an initially empty stack s of String. What output is produced when the following code is executed?

```
while (!s.isEmpty())
    System.out.print(s.pop());
```

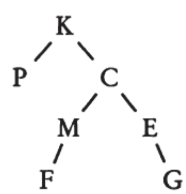


(A) AKCPRF
(B) CKRFPA
(C) FPRACK
(D) APFRKC
(E) FRPCKA

4

The tree shown is traversed postorder and each element is pushed onto a stack s as it is encountered. The following program fragment is then executed:

```
for (int i = 1; i <= 5; i++)
    x = s.pop();
```



What value is contained in x after the segment is executed?
(A) M
(B) G
(C) K
(D) F
(E) P

Figure 12.18: quiz

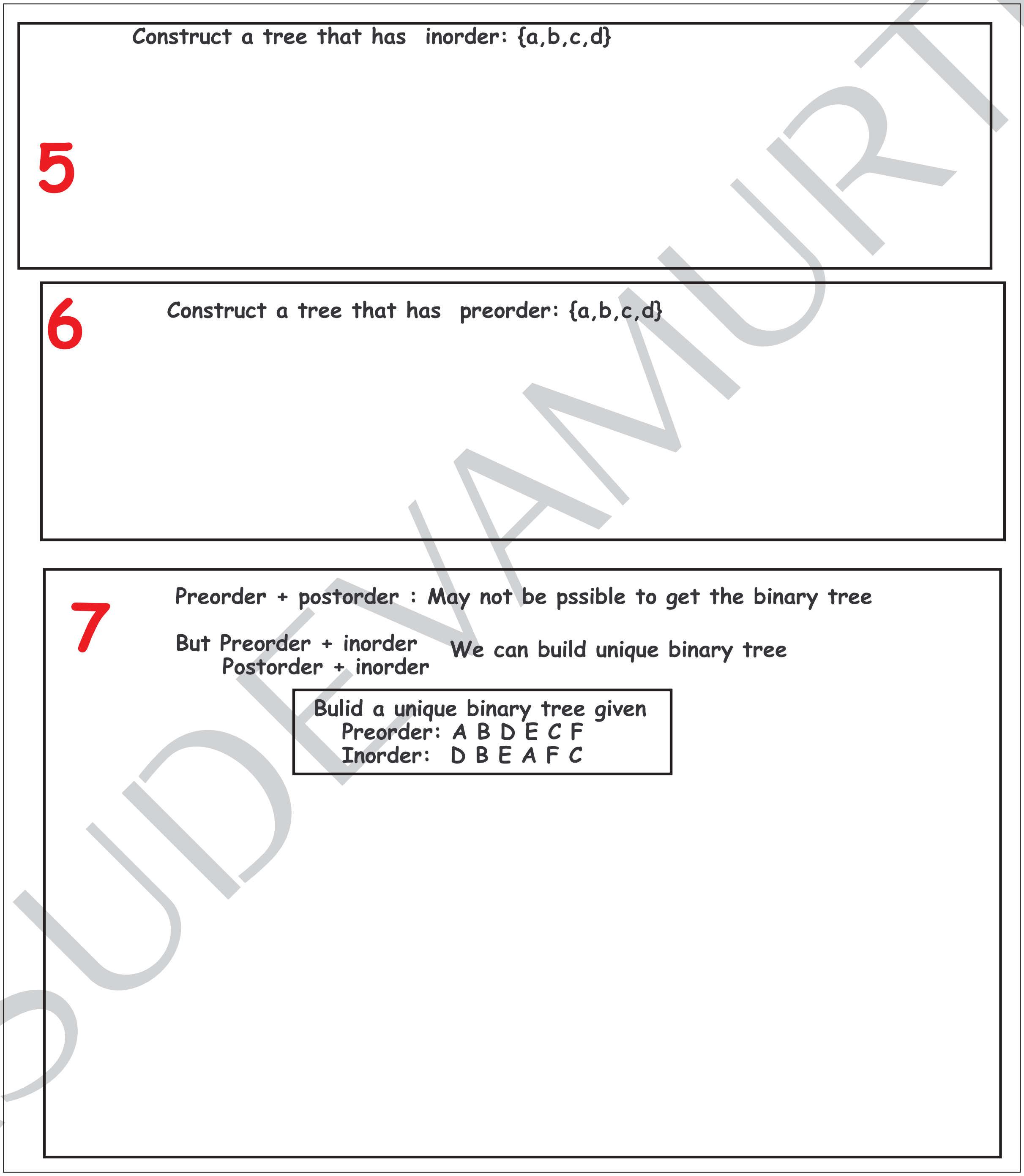


Figure 12.19: quiz

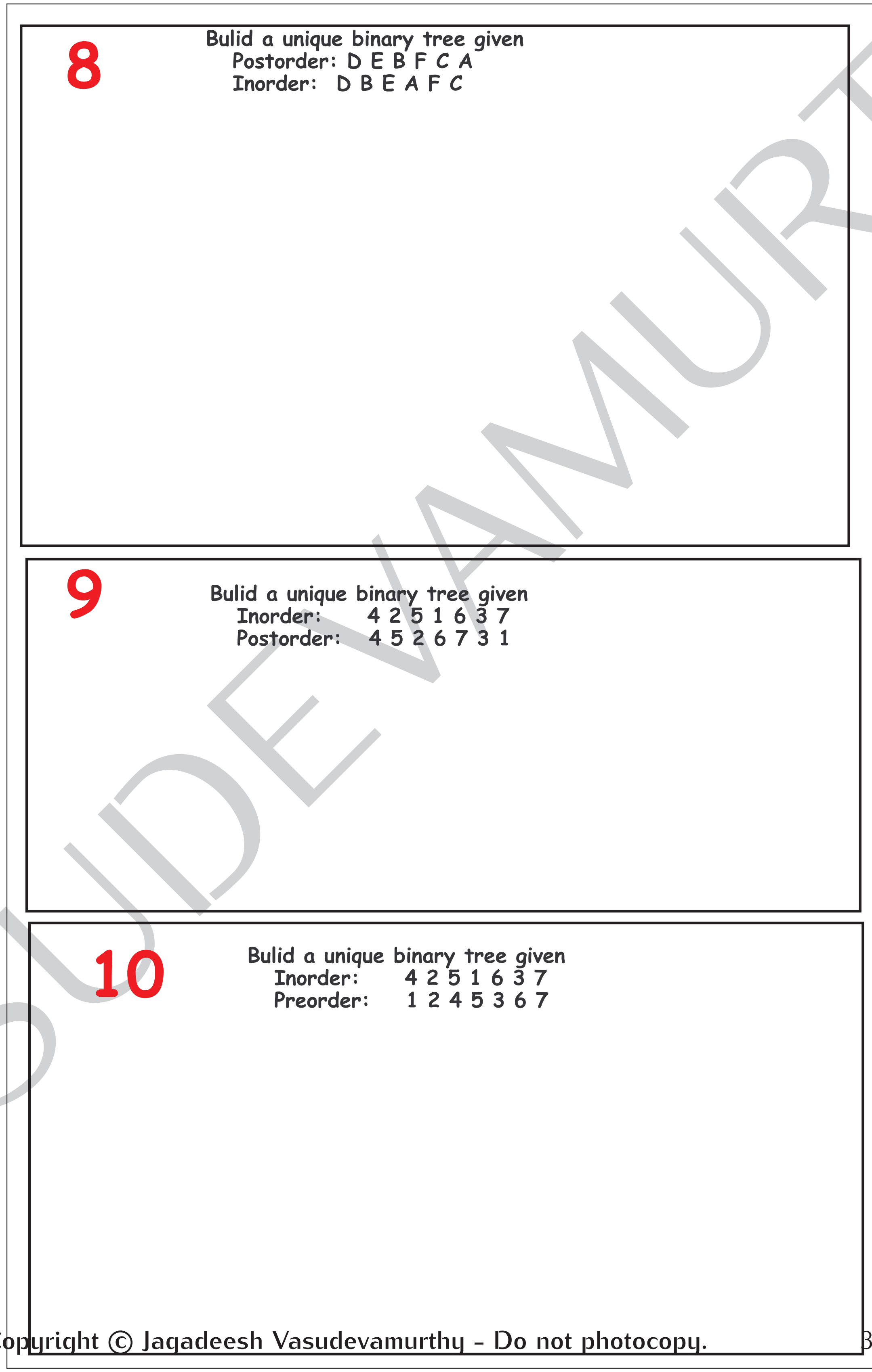


Figure 12.20: quiz

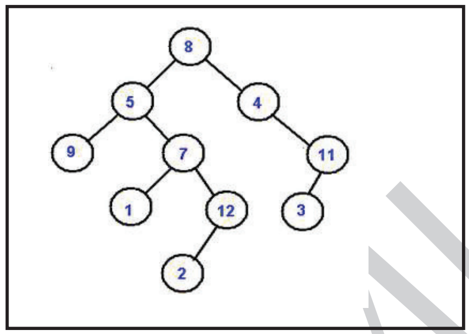
11 How many different possible trees are possible with n nodes? (Amazon)

12 if we store a tree as an array a, with root at a[1], left kid at a[2] and right kid at a[3] what will be the max size of array for n nodes?

Figure 12.21: quiz

12.7 Problem set

Problem 12.7.1. Show the traversals of a tree as explained in figure 12.22.



Show
1. **Preorder**
2. **Inorder1**
3. **Inorder2**
4. **postorder**

using recursion

Show
1. **Preorder**
2. **Inorder1**
3. **Inorder2**
4. **postorder**

without using recursion

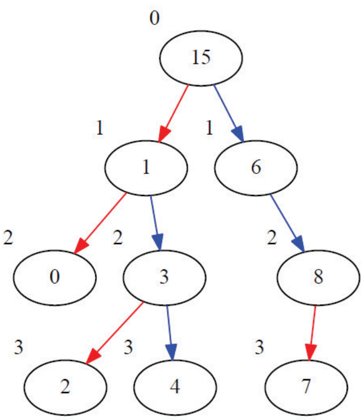
Show
Level order

No code is required. Show by hand writing, with figures, each step of the algorithm.

Figure 12.22: Traversing tree with and without recusion

Problem 12.7.2. Implement all the traversals procedures on a tree as explained in figure 12.23. **email IntBtreeTraversal.java**

Tree Traversal.



nodes = 9 # depth = 3
preorder 15 1 0 3 2 4 6 8 7
inorder1 0 1 2 3 4 15 6 7 8
inorder2 8 7 6 15 4 3 2 1 0
postorder 0 2 4 3 1 7 8 6 15
levelorder 15 1 6 0 3 8 2 4 7

```
//IntBtreeTraversal.java

IntBtreeTraversal(IntBtree t, String s, int[] ans){
   //Cannot change anything here
   tree = t ;
   if (s == "preorder") {
      preOrder(ans) ;
   }else if (s == "inorder1") {
      inOrder1(ans) ;
   }else if (s == "inorder2") {
      inOrder2(ans) ;
   }else if (s == "postorder"){
      postOrder(ans) ;
   }else if (s == "levelorder"){
     levelOrder(ans) ;
   )
}
```

Write all the routines in IntBtreeTraversal.java

```
//IntBtree.java
public int[] preOrder() {
  //Write in file: IntBtreeTraversal.java
  int [] a = new int[num] ; //Answer cannot be bigger than this
  IntBtreeTraversal b = new IntBtreeTraversal(this,"preorder",a);
  return a ;
}
```

Figure 12.23: Traversing a binary tree

Problem 12.7.3. Build various binary trees as explained in figure 12.24, 12.25 and 12.26. **email IntBtreeBuild.java**

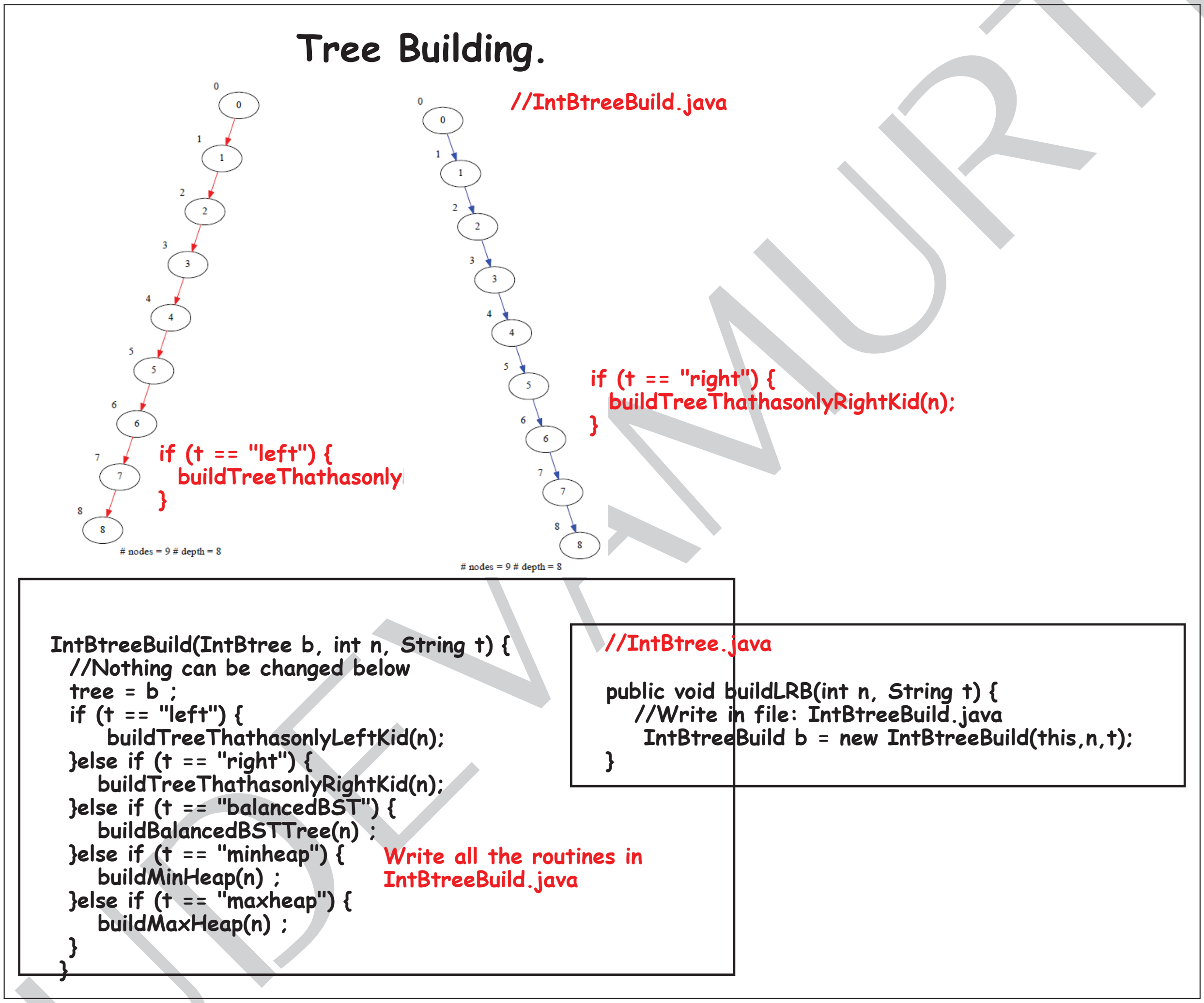


Figure 12.24: Trees that has only left or right kid

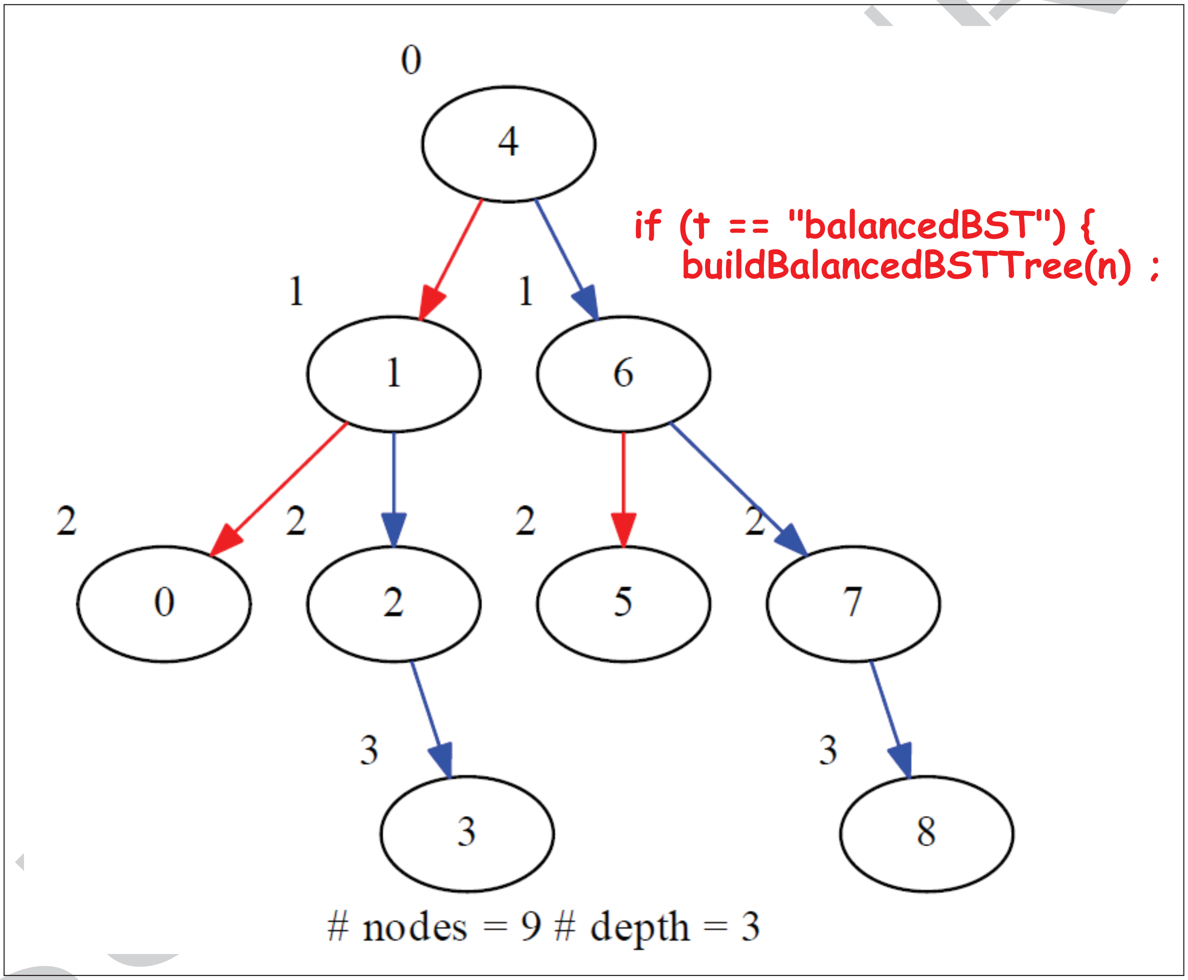


Figure 12.25: Building a binary search tree

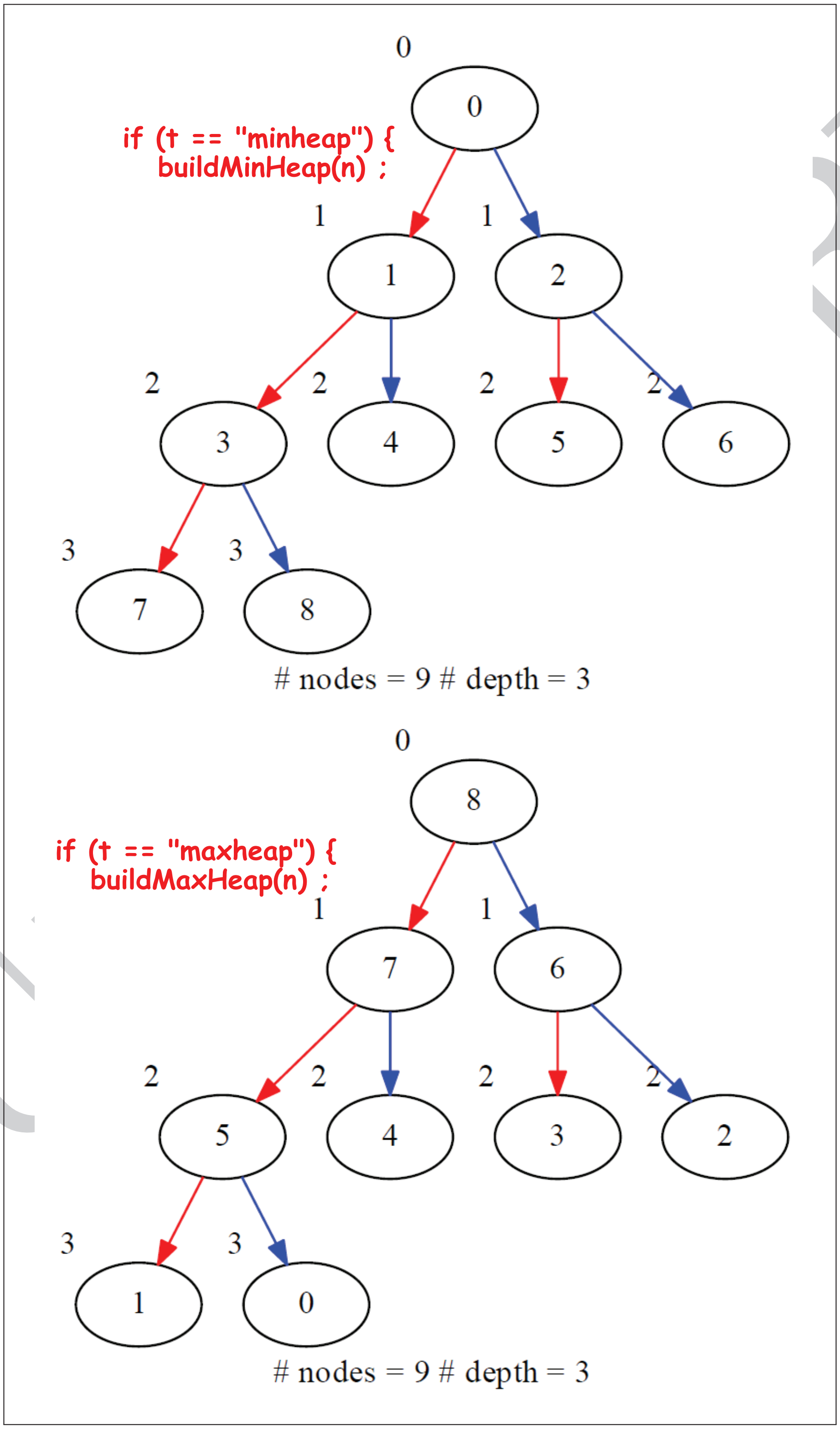


Figure 12.26: Building a minheap or a maxheap tree

Problem 12.7.4. Find all the paths from root to leaves for various binary trees as explained in figure 12.27, 12.28 and 12.29. **email IntBtreeTraversal.java**

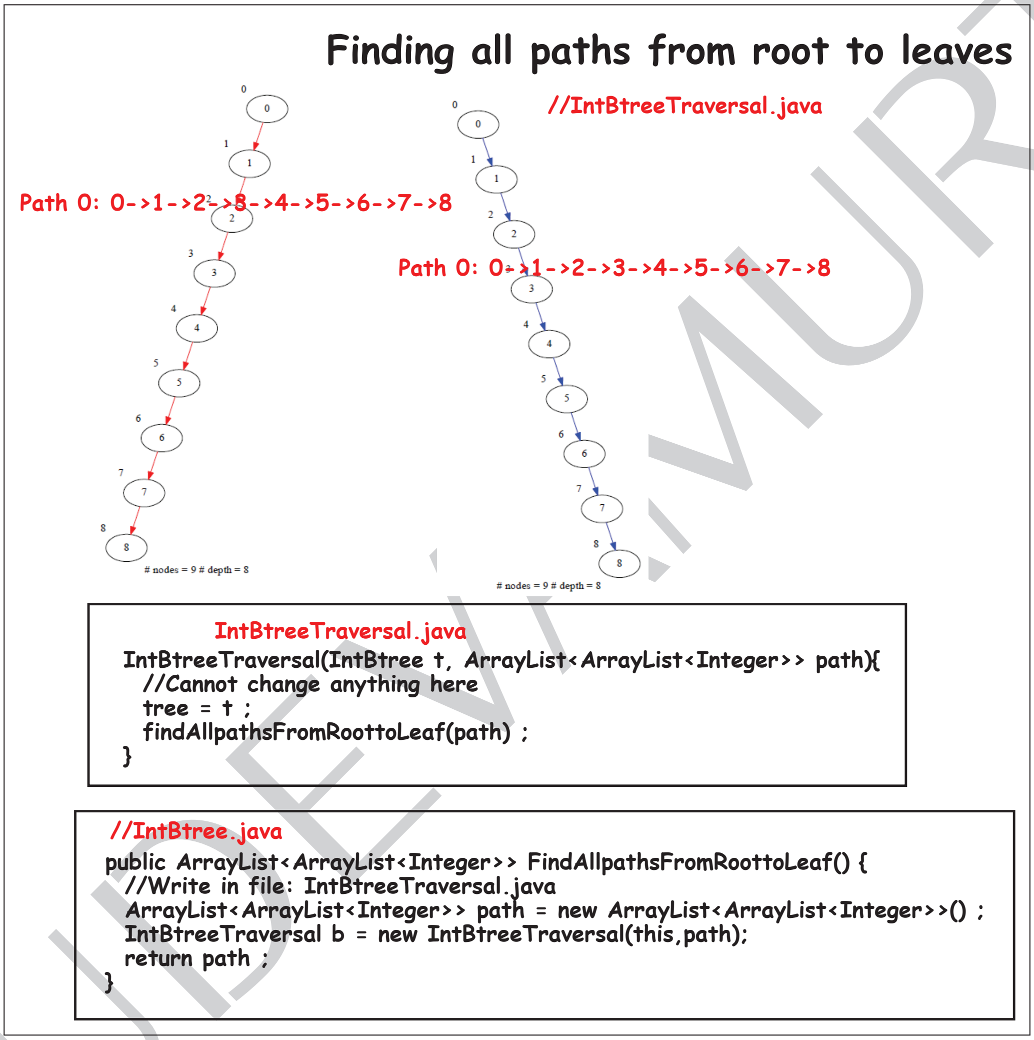


Figure 12.27: All paths from root to leaves of a binary tree that has only left or right kid

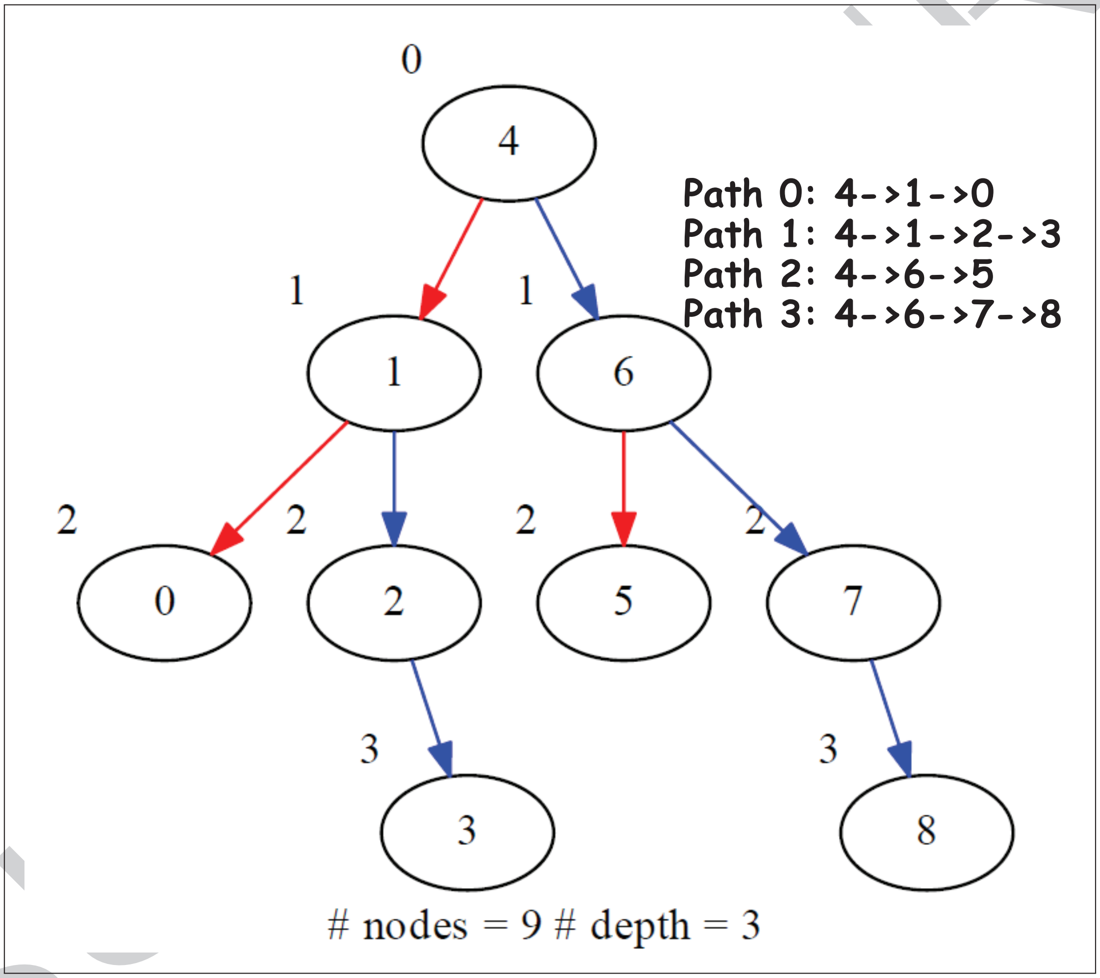


Figure 12.28: All paths from root to leaves of a binary search tree

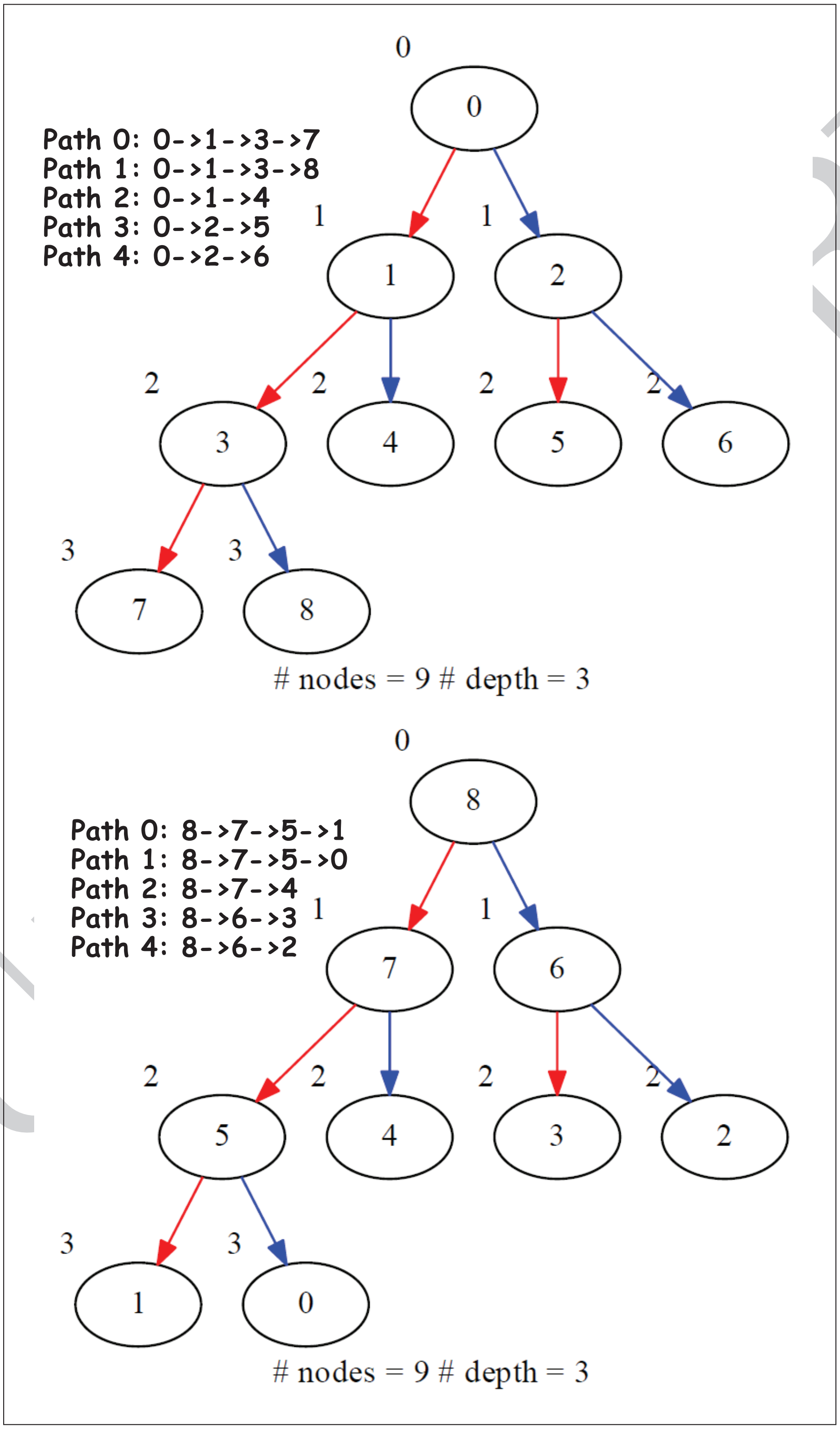


Figure 12.29: All paths from root to leaves of a minheap or maxheap trees

Problem 12.7.5. Traverse a binary tree in zigzag way as explained in figures 12.30, 12.31, 12.32, 12.33 and 12.34. **email IntBtreeTraversal.java**

Microsoft Interview question

```
IntBtreeTraversal(IntBtree t, String s, int[] ans){
    //Cannot change anything here
    tree = t ;
    if (s == "zigzag1") {
      ZigZag1(ans) ;
    }else if (s == "zigzag2") {
      ZigZag2(ans) ;
    }else if (s == "zigzag3") {
      ZigZag3(ans) ;
    }else if (s == "zigzag4"){
      ZigZag4(ans) ;
    }
}
```

Implement in IntBtreeTraversal.java

```
//ZigZag traversal
public int [] ZigZag(int i) {
    //Write in file: IntBtreeTraversal.java
    String [] s = {"zigzag1","zigzag2","zigzag3","zigzag4"};
    int [] t = new int[num] ; //Answer cannot be bigger than this
    IntBtreeTraversal b = new IntBtreeTraversal(this,s[i],t);
    return t ;
}
```

IntBtree.java

```
private void testZigZag() {
    int [] e1 = {15,6,1,0,3,8,7,4,2};
    int [] e2 = {15,1,6,8,3,0,2,4,7};
    int [] e3 = {2,4,7,0,3,8,1,6,15};
    int [] e4 = {7,4,2,8,3,0,6,1,15};
    int [][] e = {e1,e2,e3,e4} ;
    IntBtree t = new IntBtree() ;
    t.buildTreeFromUserSpec(treeExamples(3));
    for (int i = 0; i < 4; ++i) {
       int [] z = t.ZigZag(i);
       printAPath(z) ;
       assertOrder(e[i],z) ;
    }
    System.out.println("All ZigZag Passed. Ready for microsoft intervi
}
```

IntBtreeTest.java

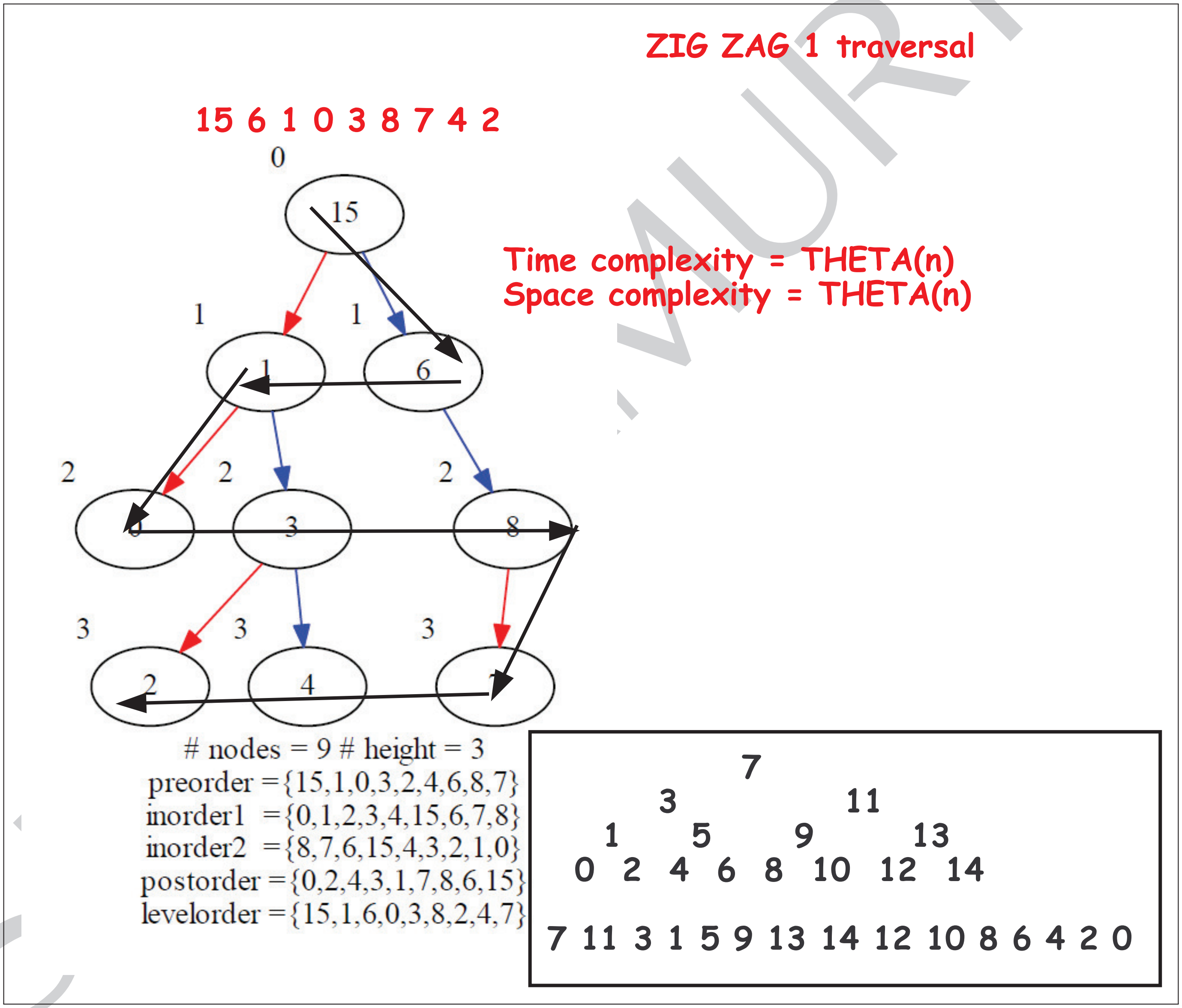


Figure 12.31: Zigzag traversal 1

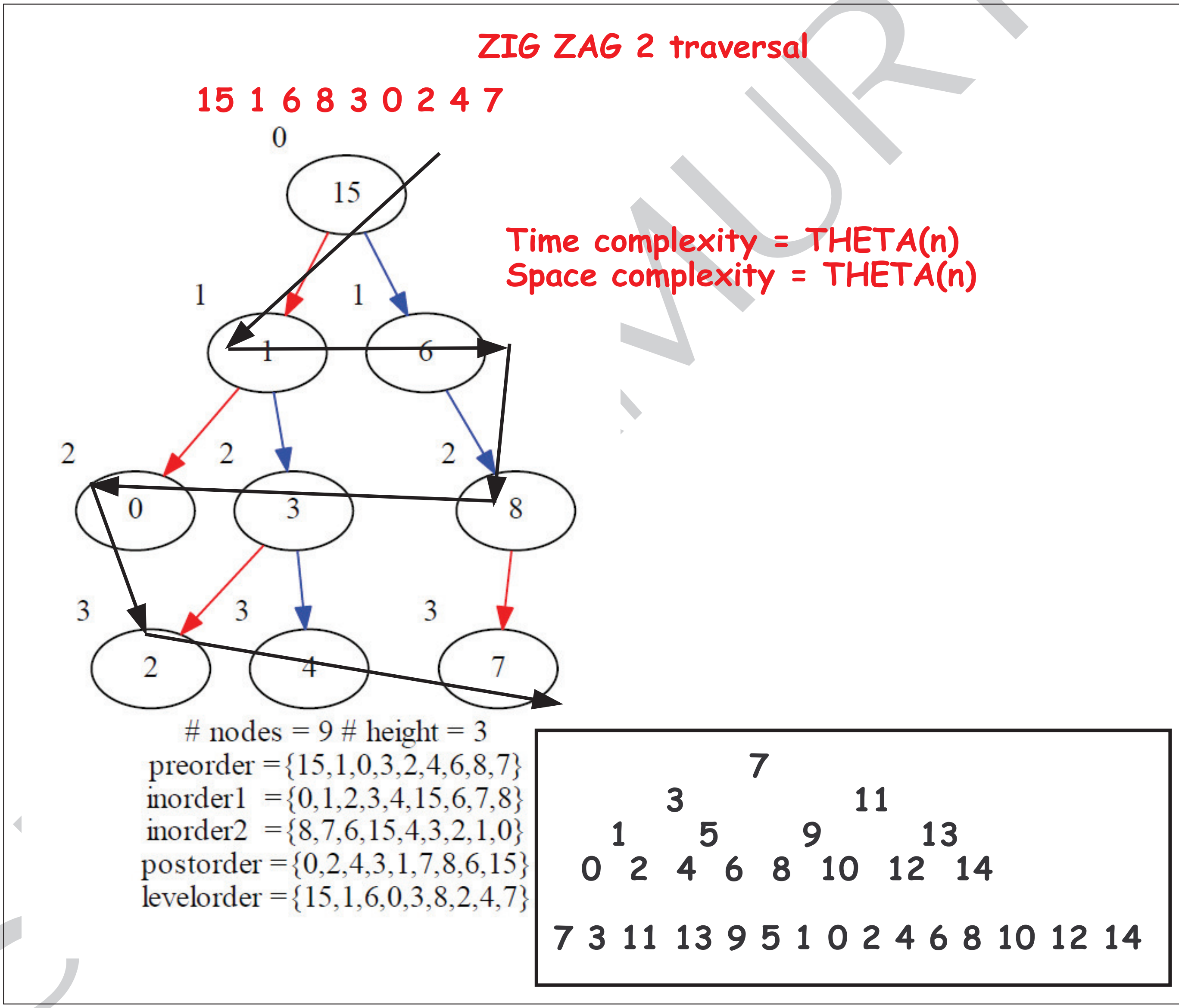


Figure 12.32: Zigzag traversal 2

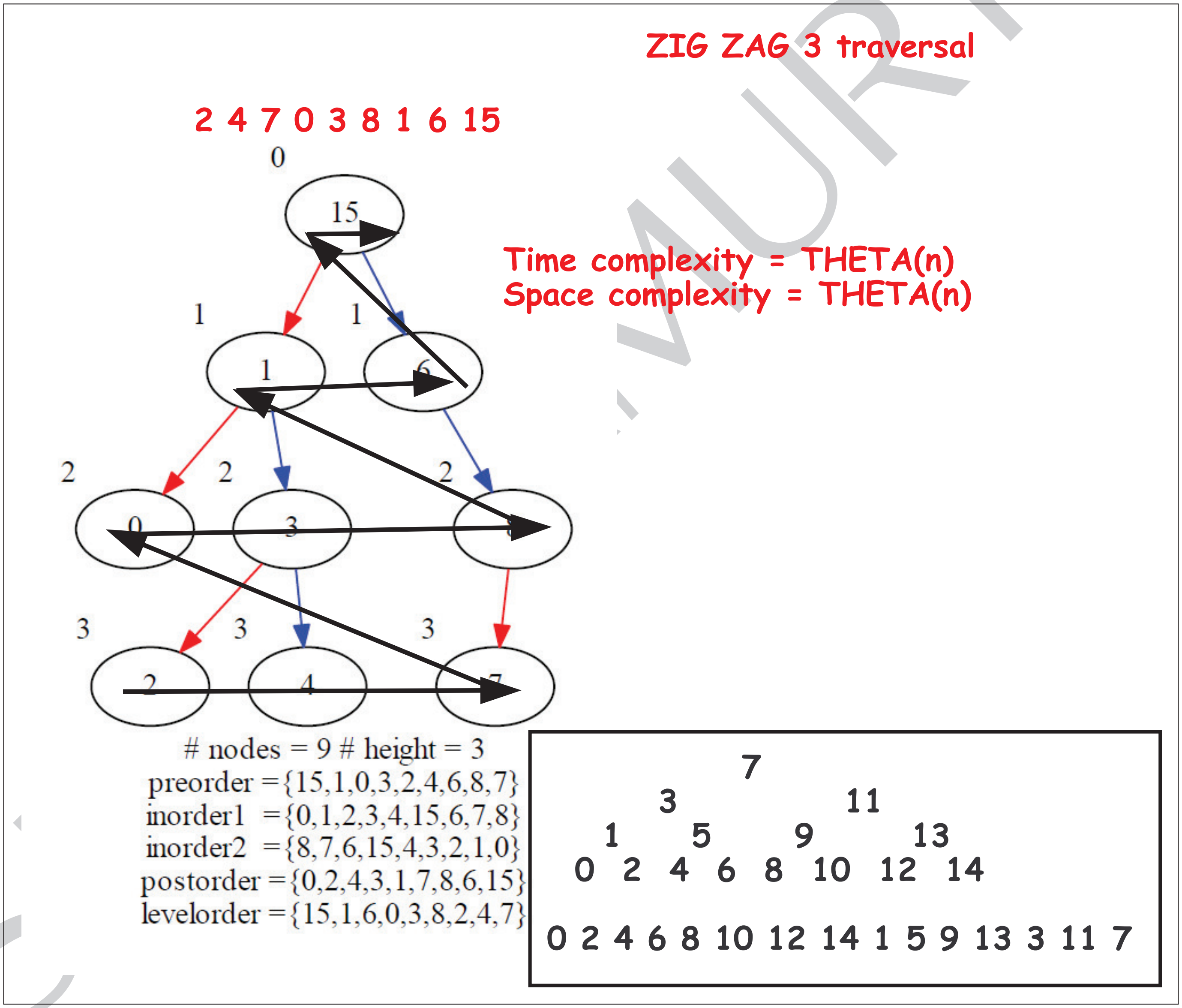


Figure 12.33: Zigzag traversal 3

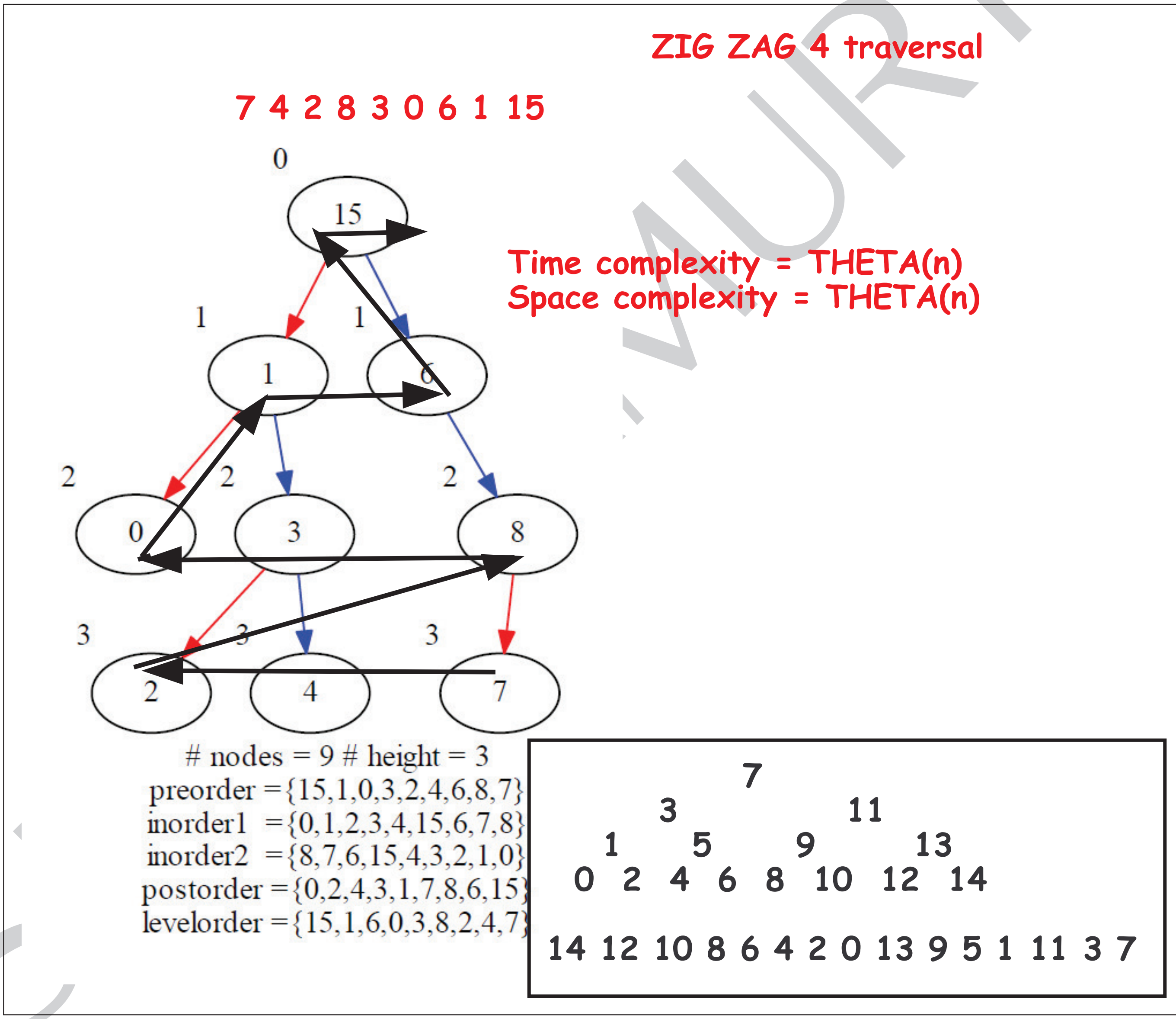


Figure 12.34: Zigzag traversal 4

Chapter 13

Binary Search Tree

13.1 Introduction

13.2 Definition of BST

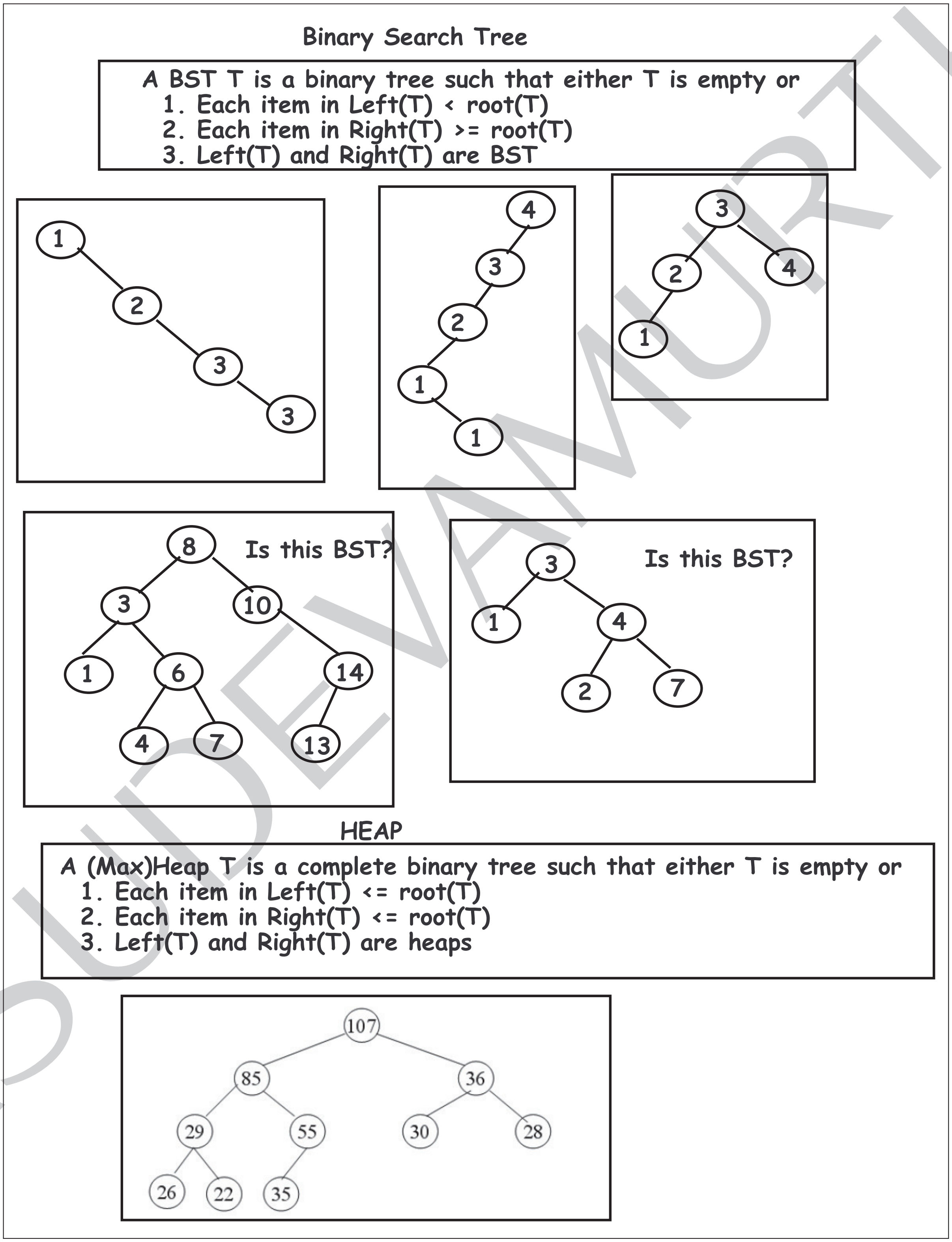


Figure 13.1: Definition **BST**

13.3 Is BST?

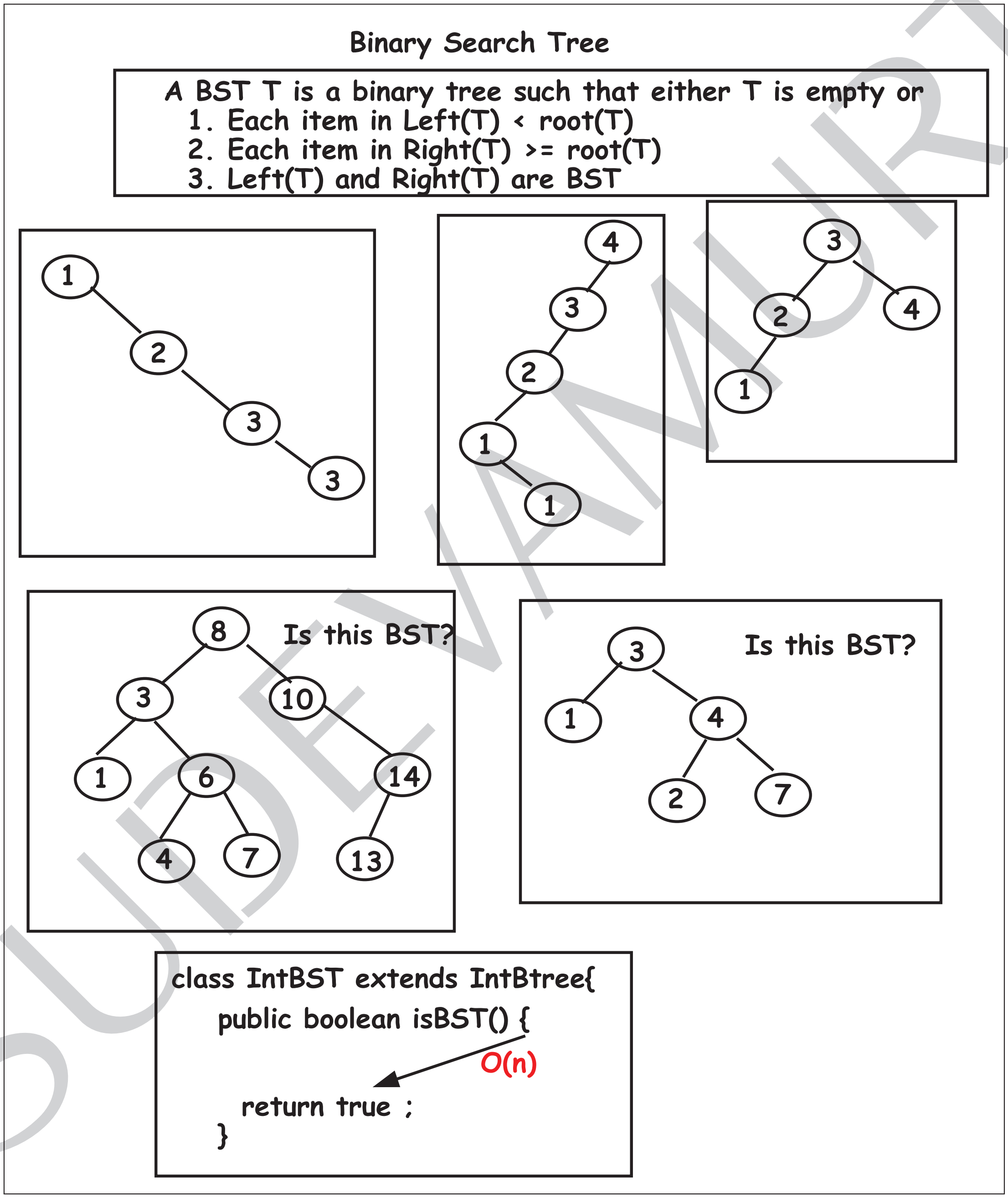


Figure 13.2: Testing a BST

13.4 Building BST

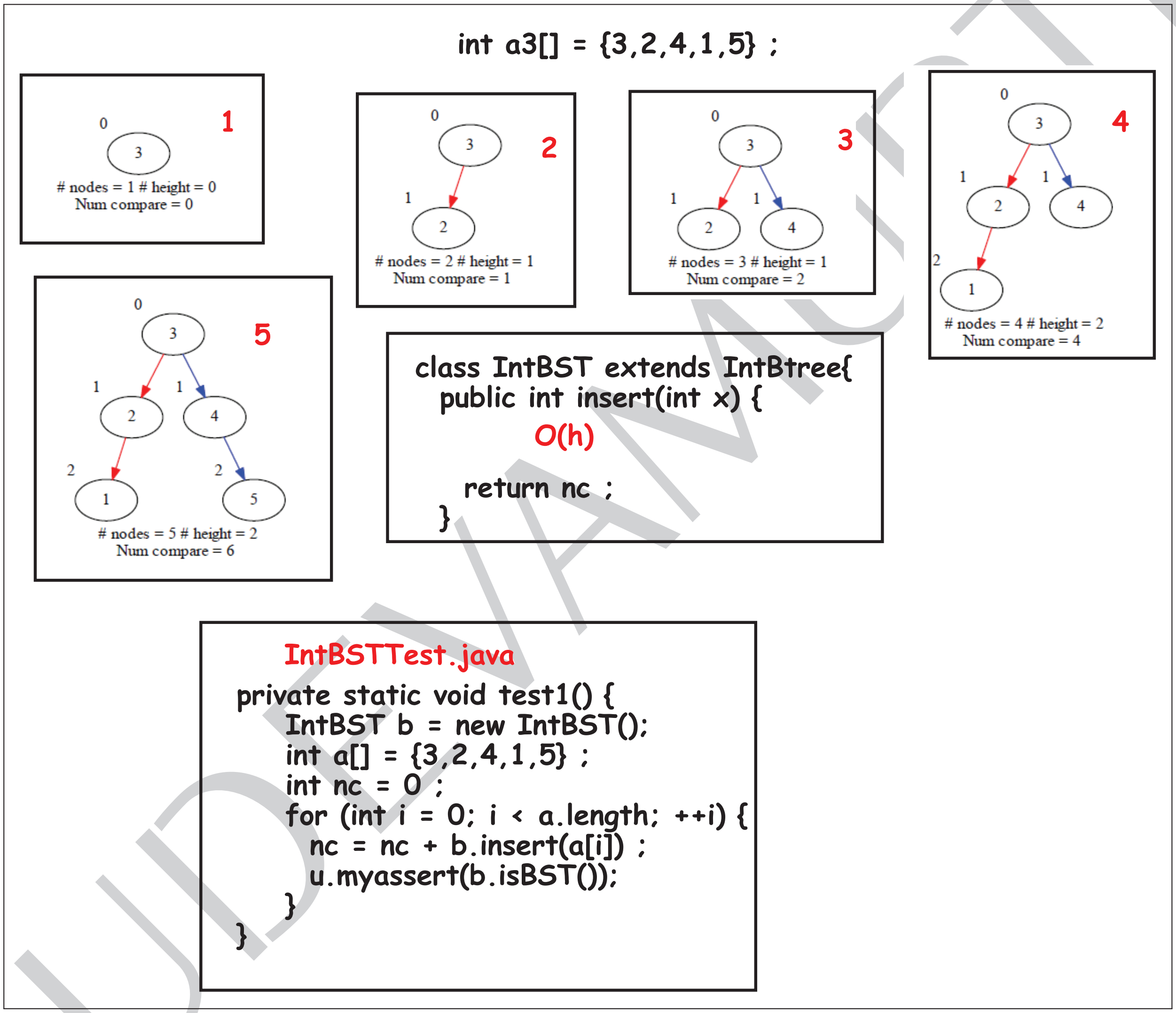
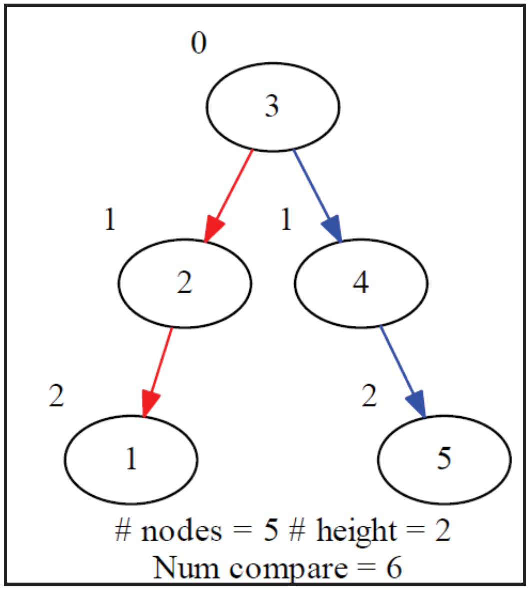


Figure 13.3: Building BST

13.5 Searching an element in BST




```
class IntBST extends IntBtree{
public boolean find(int x, Int nc) {
  node r = root ;
  while (r != null) {
        O(h)
  }
  return false ;
}
```

```
3 can be found with 0 comparsion
2 can be found with 1 comparsion
4 can be found with 1 comparsion
1 can be found with 2 comparsion
5 can be found with 2 comparsion
-7 cannot be found with 3 comparsion
8 cannot be found with 3 comparsion
6 cannot be found with 3 comparsion
```

IntBSTTest.jav

```
int a[] = {3,2,4,1,5} ;
for (int i = 0; i < a.length; ++i) {
      Int ncc = new Int(0) ;
      boolean f = b.find(a[i], ncc);
      if (f) {
        System.out.println(a[i] + " can be
        found with "  + ncc.get() + " comparsion") ;
      }
    }
    int [] z = {-7, 8, 6} ;
    for (int i = 0; i < z.length; ++i) {
      Int ncc = new Int(0) ;
      boolean f = b.find(z[i], ncc);
      if (!f) {
        System.out.println(z[i] + " cannot be
         found with "  + ncc.get() + " comparsion") ;
      }
    }
```

Figure 13.4: Searching an element BST

13.6 Minimum and maximum of a node

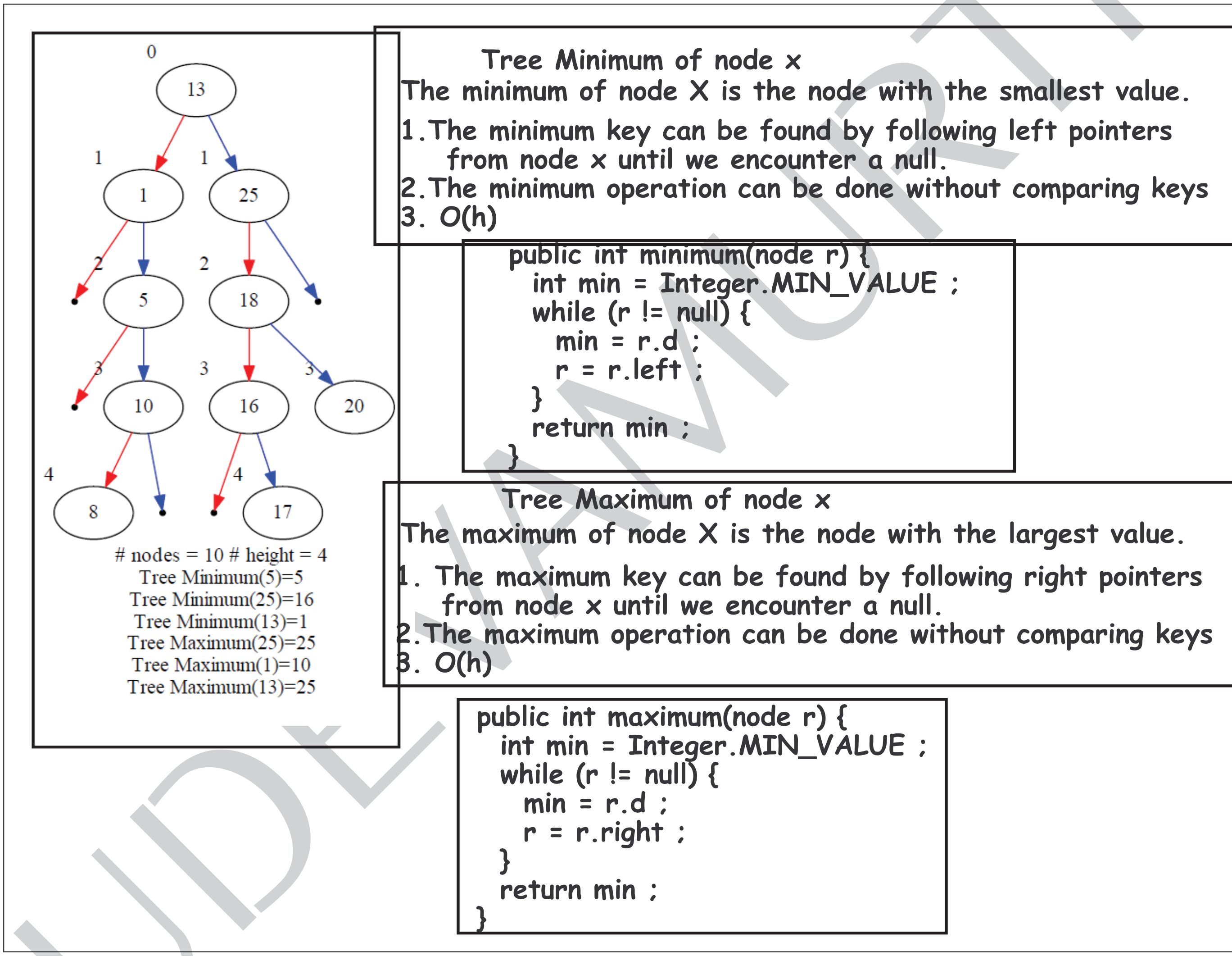
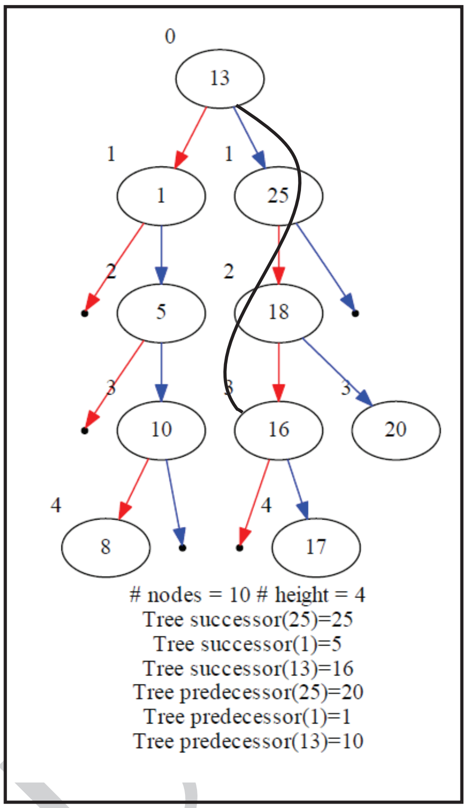


Figure 13.5: Finding minimum and maximum of a node

13.7 Successor of a node



Tree Successor of a node x

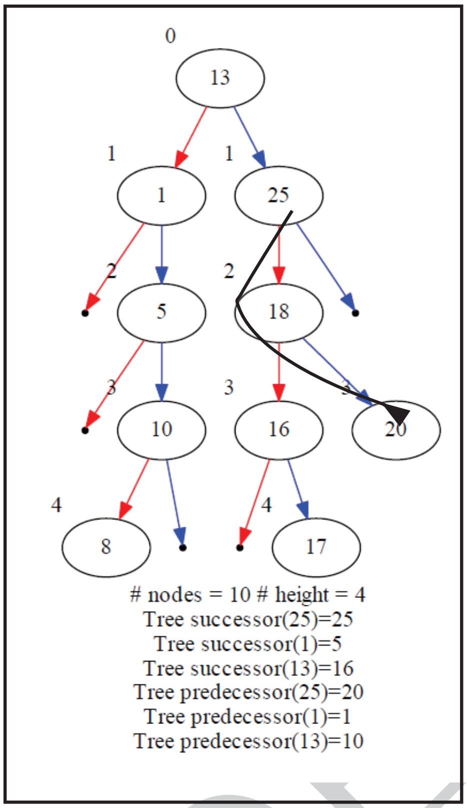
The successor of a node x is the node
with the next possible smallest value

1. The successor key is found by right followed
   by left pointer
2. Successor has NO left child
3. Successor operation is done without comparing key
4. O(h)

```
private node successor(node r) {
  node f = r ;
  //WRITE CODE
  //Tree successor has NO left child
  u.myassert(f.left == null);
  return f ;
}
```

Figure 13.6: Finding successor of a node

13.8 Predecessor of a node



Tree Predecessor of a node x

The predecessor of a node x is the node with the next possible largest value

1. The predecessor key is found by left followed by right pointer
2. Predecessor has NO right child
3. Predecessor operation is done without comparing key
4. O(h)

```
private node predecessor(node r) {
  node f = r ;
  //WRITE CODE
  u.myassert(f.right == null);
  return f ;
}
```

Figure 13.7: Finding predecessor of a node

13.9 Deleting an element from BST

13.9.1 Case 1: Deleting a leaf from BST

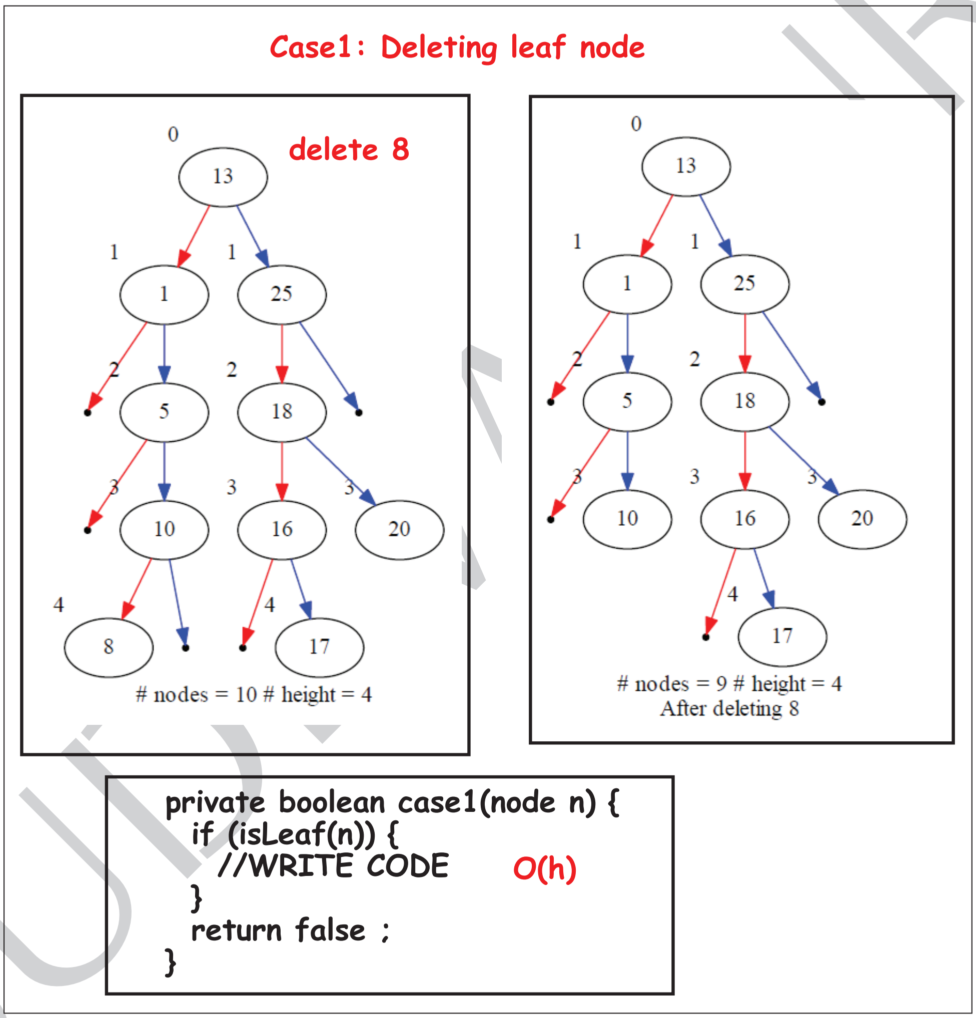


Figure 13.8: Delete a leaf from BST

13.9.2 Case 2: Deleting a node that has one kid from BST

13.9.2.1 Case 2a: Kid is a leaf

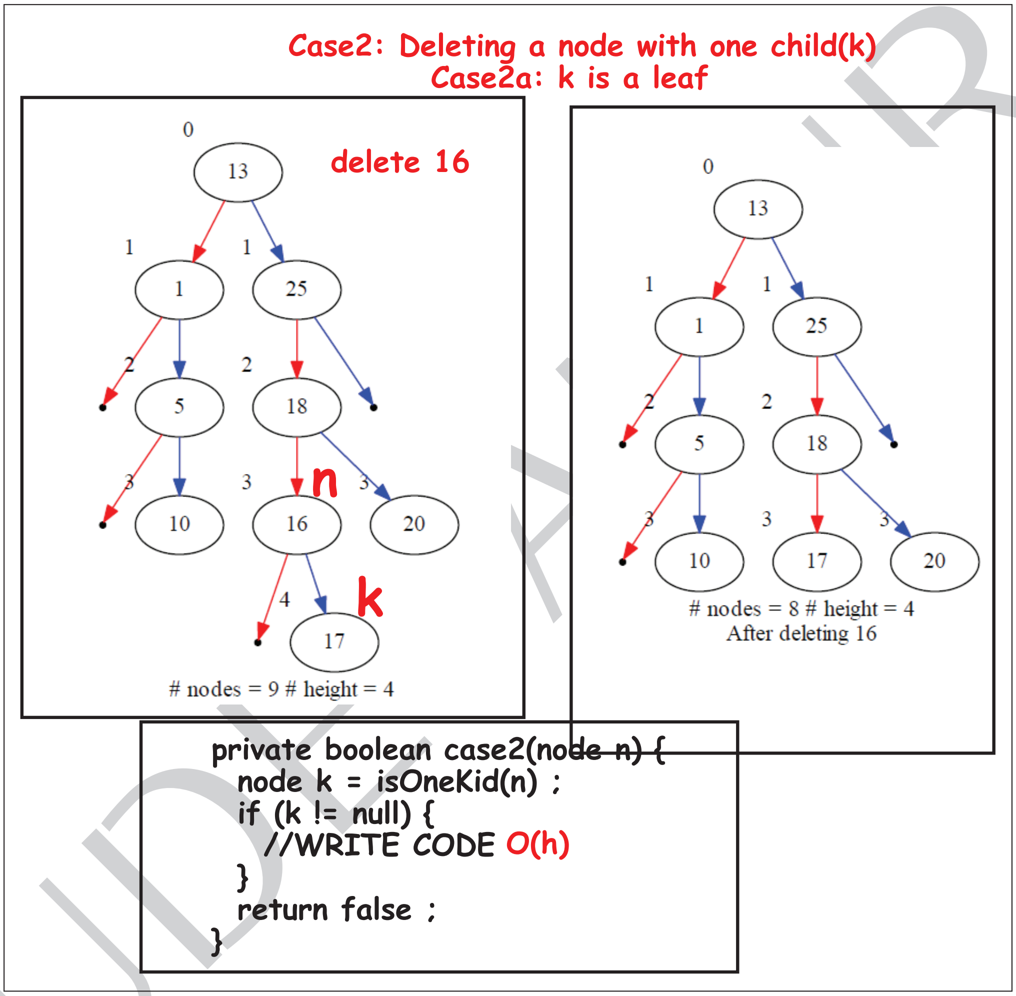


Figure 13.9: Delete a node that has one kid(which is a leaf) from BST

13.9.2.2 Case 2b: Kid is a non leaf

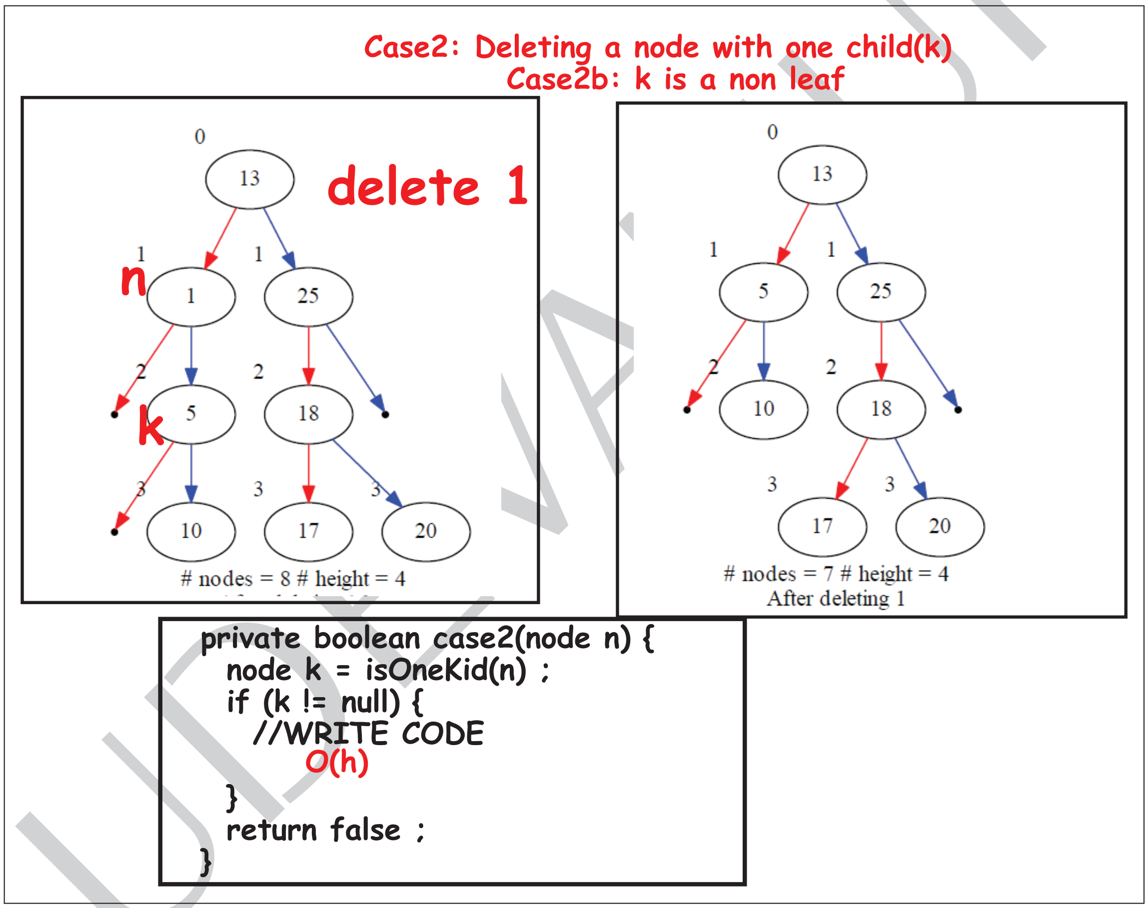


Figure 13.10: Delete a node that has one kid(which is a non leaf) from BST

13.9.3 Case 3: Deleting a node that has two kids from BST

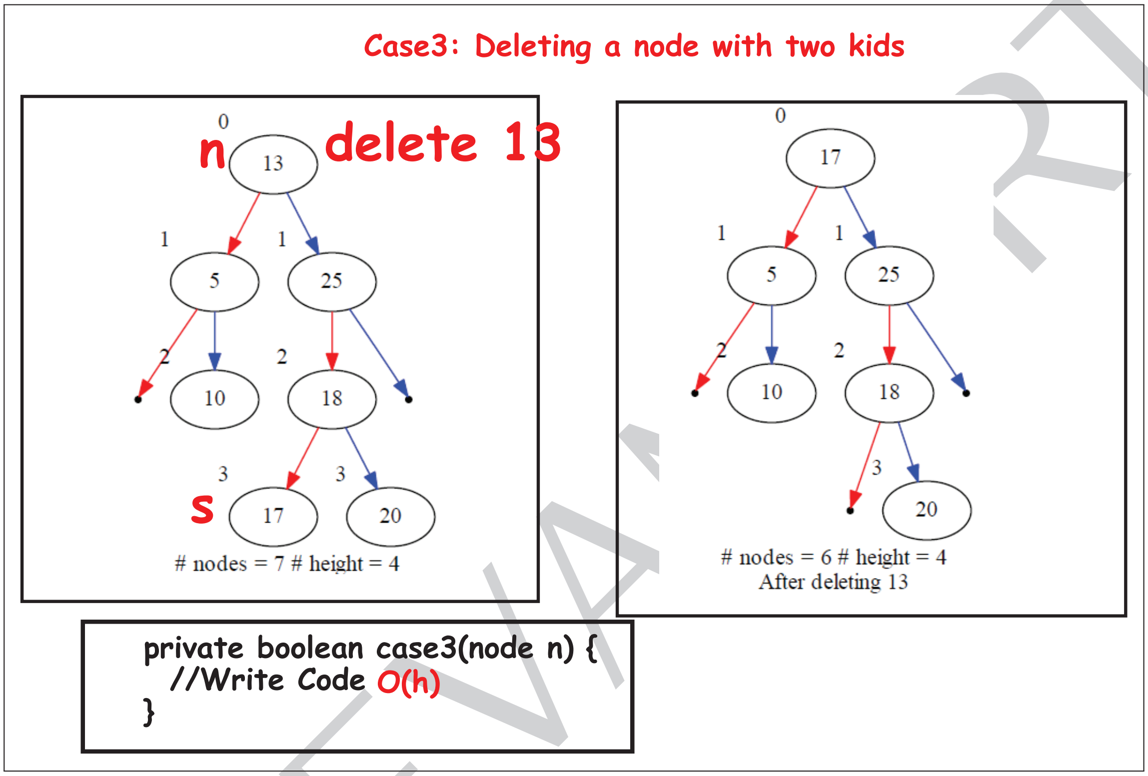


Figure 13.11: Delete a node that has two kids from **BST**

13.9.4 Deletion in action

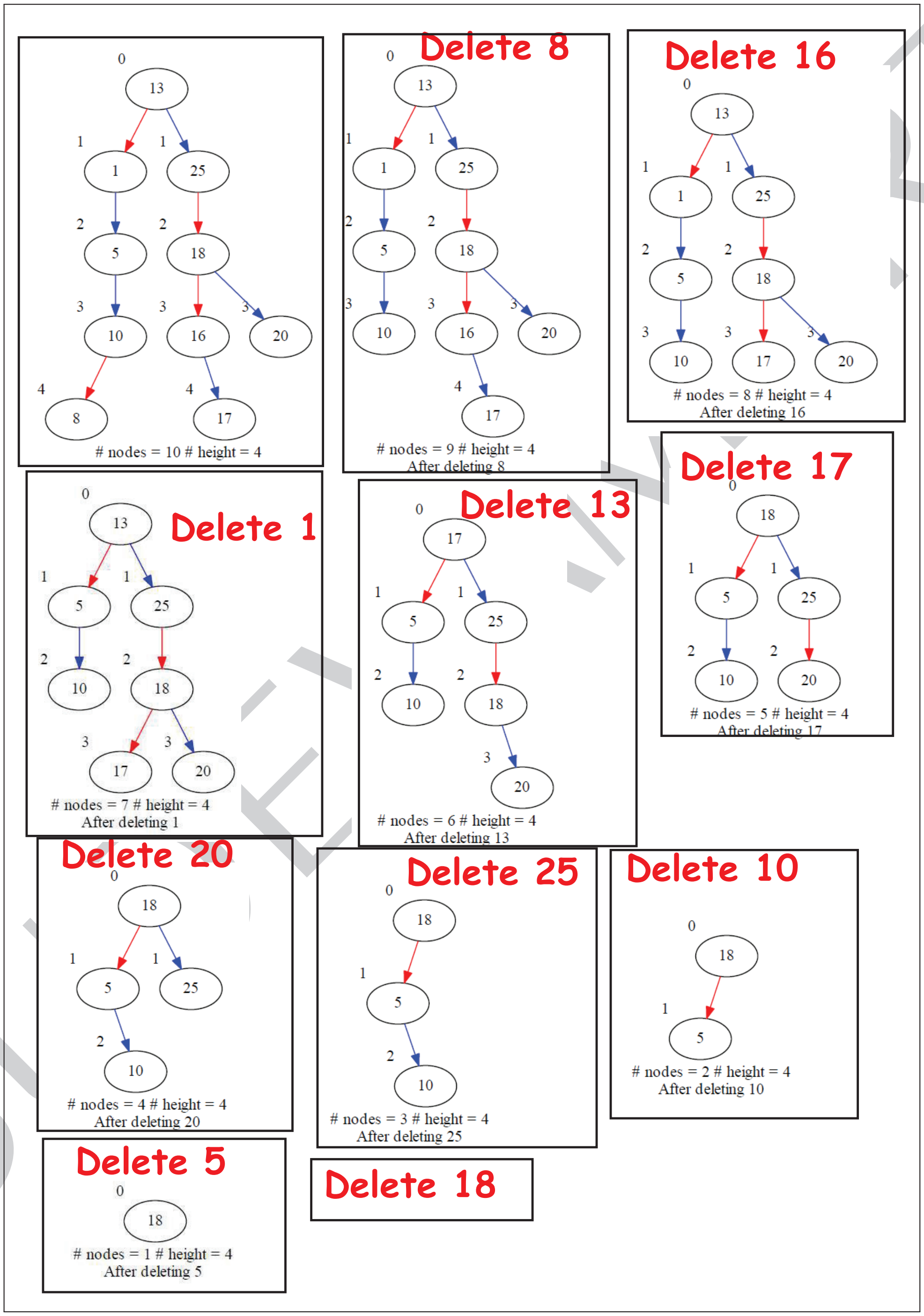


Figure 13.12: Delete a node from **BST**

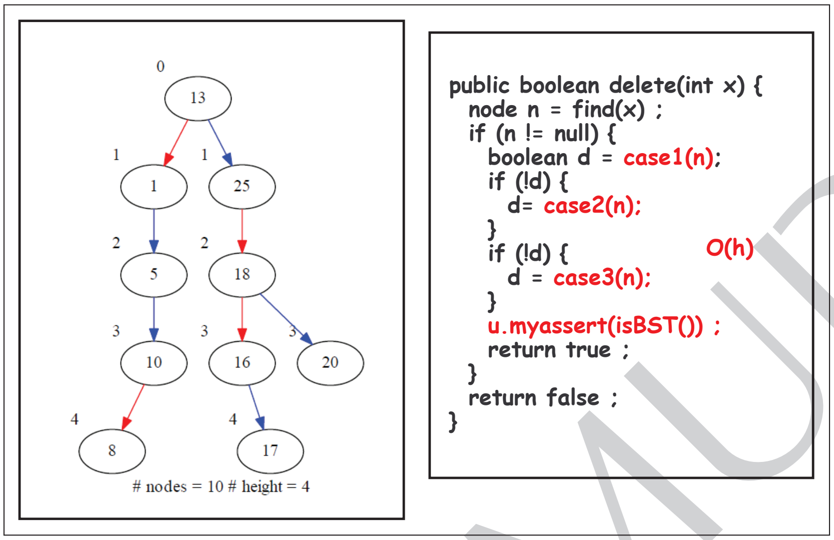


Figure 13.13: Code for delete

13.10 TreeMap

Class TreeMap<K,V>

java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.TreeMap<K,V>

Type Parameters:

`K` - the type of keys maintained by this map

`V` - the type of mapped values

Red-Black Tree implementation
Requires only < function

Insert O(log n)
Find O(log n)
Delete O(log n)

```
public class Complex implements Comparable<Complex> {
  // All private data members
  private final int x; //WE cannot change it. Used for Hash
  private int y;
  private String s;


        /* < operator */
         @Override
        public int compareTo(Complex o) {
              if (this.x < o.x) {
                   return -1;
              }
              if (this.x > o.x) {
                 return 1;
              }
              return 0;
        }
        public int hashCode() {
          //NOT REQUIRED
        }

        public boolean equals(Object obj) {
         // NOT required. Even if if you implement, WILL NOT BE CALLED
              (a == b) = !(a < b) && !(b < a)

        }
   }
```

NOTE FINAL

$\boxed{(a == b) = !(a < b)\ \&\&\ !(b < a)}$

Figure 13.14: Treemap class

```java

/**
 * File Name: Complex.java
 * class acts as collection of real, imaginary and
 * string s = real +j (imaginary)
 *
 * @author Jagadeesh Vasudevamurthy
 * @year 2018
 */
```

You need to implement <

```java

public class Complex implements Comparable<Complex> {
  // All private data members
  private final int x; //WE cannot change it. Used for Hash
  private int y;
```

Note final x

```java
  private String s;
  private static boolean m_display = false;

  public Complex(int x, int y) {
    if (m_display) {
      System.out.println("In complex constructor " + x + " " + y);
    }
    this.x = x;
    this.y = y;
    this.s = null ;
    buildString();
  }

  public Complex(int x) {
    this(x, 0); // This must be in first line
    if (m_display) {
      System.out.println("In complex constructor " + x + " " + "0");
    }
  }

  public Complex() {
    this(0, 0); // This must be in first line
    if (m_display) {
      System.out.println("In complex constructor " + "0" + " " + "0");
    }
  }

  public int getX() {
    return x ;
  }

  public int getY() {
    return y ;
```

```
}

public void setY(int y) {
  this.y = y;
  buildString();
}

@Override
public String toString() {
  return s;
}

//Comment hashCode() and equals() for testing Treemap.java
//
 //A class that overrides equals must also override hashCode
/*
@Override
public int hashCode() {
  final int prime = 31;
  int result = 1;
  result = prime * result + x;
  return result;
}

```

hashcode()
and
equals()
not required. Even if you write will not be called

```
//A class that overrides equals must also override hashCode
@Override
public boolean equals(Object obj) {
  if (this == obj)
    return true;
  if (obj == null)
    return false;
  if (getClass() != obj.getClass())
    return false;
  Complex other = (Complex) obj;
  if (x != other.x)
    return false;
  return true;
}
*/
/* < operator */
@Override
public int compareTo(Complex o) {
```

Must implement <

```
  // TODO Auto-generated method stub
  boolean show = true ;
  if (show) {
    System.out.println("this = " + this) ;
```

```
    System.out.println("o    = " + o) ;
  }
  if (this.x < o.x) {
    if (show) {
      System.out.println("this < o. Return -1") ;
    }
    return -1 ;        < -1
  }
  if (this.x > o.x) {
    if (show) {
      System.out.println("this > o. Return 1") ;
    }
    return 1 ;         > 1
  }
  if (show) {
    System.out.println("this == o. Return 0") ;
  }
  return 0 ;         == 0
}

private String convertIntToString(int x) {
  String s = new String();
  String s1 = new String();
  if (x < 0) {
    s1 = s1 + '-';
    x = -x;
  }
  do {
    s = s + (x % 10);
    x = x / 10;
  } while (x != 0);
  for (int i = s.length() - 1; i >= 0; --i) {
    s1 = s1 + s.charAt(i);
  }
  return s1;
}

private void buildString() {
  s = convertIntToString(x);
  int ty = y ;
  if (y < 0) {
    s = s + "-j";
    ty = -y;
  } else {
    s = s + "+j";
  }
  s = s + convertIntToString(ty);
```

```java
  }

  private static void testBench() {
    Complex c1 = new Complex(2, 3);
    System.out.println("c1 = " + c1) ;
    Complex c2 = new Complex(2, -200);
    System.out.println("c2 = " + c2) ;
    Complex c3 = new Complex(-20, 4);
    System.out.println("c3 = " + c3) ;
    Complex c4 = new Complex(-18, -99);
    System.out.println("c4 = " + c4) ;
    c2.setY(3);
    System.out.println("c2 = " + c2) ;
    if (c1 == c2) {
      System.out.println("c1 == c2");
    } else {
      System.out.println("c1 != c2");
    }
    if (c1.equals(c2)) {
      System.out.println("c1 equals c2");
    } else {
      System.out.println("c1 !equals c2");
    }
    int h1 = c1.hashCode();
    int h2 = c2.hashCode() ;
    System.out.println("c1 hashcode = " + h1) ;
    System.out.println("c2 hashcode = " + h2) ;

    Complex c6 = c1 ;
    if (c6 == c1) {
      System.out.println("c6 == c1");
    } else {
      System.out.println("c6 != c1");
    }
    h1 = c1.hashCode();
    h2 = c6.hashCode() ;
    System.out.println("c1 hashcode = " + h1) ;
    System.out.println("c6 hashcode = " + h2) ;
  }

  public static void main(String[] args) {
    System.out.println("Complex.java starts");
    testBench();
    System.out.println("Complex.java DONE");
  }
}
```

Red-Black tree
O(log n) complexity
Only requires < operator

```java

import java.util.Map.Entry;
import java.util.TreeMap;

/**
 * File Name: TestTreeMap.java
 *
 * To Compile: TestTreeMap.java Complex.java
 *
 * @author Jagadeesh Vasudevamurthy
 * @year 2018
 */

public class TestTreeMap{

  TestTreeMap() {

  }

  private void testContainsKey(TreeMap<Complex,String> h,Complex c) {
    boolean x = h.containsKey(c) ;
    if (x) {
      System.out.println("Hash has " + c) ;
    }else {
      System.out.println("Hash DOES NOT has " + c) ;
    }
  }

  private void testChangeValue(TreeMap<Complex,String> h, Complex c, String news) {
    testContainsKey(h,c);
    h.put(c,news) ;
  }

  private void testChangeValue(TreeMap<String,Complex> h, String s,int n) {
    Complex c = h.get(s);
    if (c != null) {
      c.setY(n) ;
      h.put(s,c) ;
    }
  }

  private void printSC(String t, TreeMap<String,Complex> h){
    System.out.println("============ " + t + " +++++++++++++++") ;
    for (Entry<String,Complex> entry : h.entrySet()) {
      String key = entry.getKey();
      Complex value = entry.getValue();
```

```
      System.out.println("For Key " + key + " value is  " + value);
    }
  }

  private void printCS(String t, TreeMap<Complex,String> h){
    System.out.println("============ " + t + " +++++++++++++++") ;
    for (Entry<Complex, String> entry : h.entrySet()) {
      Complex key = entry.getKey();
      String value = entry.getValue();
      System.out.println("For Key " + key + " value is  " + value);
    }
  }

  /*
   * Key is String.
   * Value is Complex
   */
  private void test_String_Complex() {
    //Key is String. Value is Complex
    System.out.println("============  test_String_Complex +++++++++++++++") ;
    TreeMap<String,Complex> hm = new TreeMap<String,Complex>() ;
    Complex a1_1 = new Complex(1,1) ;
    Complex a2_2 = new Complex(2,2) ;
    hm.put("525_11_1240",a1_1) ;
    hm.put("525_22_1240",a2_2) ;
    printSC("After Insertion ",hm);
    System.out.println("============  Testing changing values +++++++++++++++") ;
    testChangeValue(hm,"525_11_1240",420);
    printSC("After Insertion ",hm);
  }

  /*
   * Key is Complex
   * Value is String
   */

  private void test_Complex_String() {
    System.out.println("============  test_Complex_String +++++++++++++++") ;
    TreeMap<Complex,String> hm = new TreeMap<Complex,String>() ;
    Complex a5_20 = new Complex(5,20) ;
    Complex a5_20z = new Complex(5,20) ;
    Complex a0_0 = new Complex() ;
    Complex a4_0 = new Complex(4) ;
    Complex a10_0 = new Complex(10) ;
    Complex a11_0 = new Complex(11) ;

```

Use compareTo routine provided by String class

Uses compareTo routine written in complex class

```java
    /*
     * COMMENT IN Complex.java
     * public int hashCode()
     * public boolean equals(Object obj)
     * to prove only < is required
     * public int compareTo(Complex o)
     */

    // Insertion

    hm.put(a5_20,"a5_20") ;
    hm.put(a0_0,"a0_0") ;
    hm.put(a4_0,"a4_0");
    hm.put(a4_0,"a4JNEW_0");

    printCS("After Insertion ",hm);

    System.out.println("============  Testing contains +++++++++++++++") ;


    //Find without having equal opera
    testContainsKey(hm,a5_20) ;
    testContainsKey(hm,a5_20z) ;

    //Changing value in Hash without having equal operator
    //You can change Value and not the Key
    //final K key; V value;
    System.out.println("============  Testing changing values ++++++++++++++
+") ;
    testChangeValue(hm,a5_20,"Changeda5_20")
    testChangeValue(hm,a5_20z,"Changeda5_20z
    testChangeValue(hm,a10_0,"a10_0z");
    printCS("After Change ",hm);

    //Removing an element from Has
    System.out.println("==========
    hm.remove(a4_0) ;
    hm.remove(a11_0) ;
    printCS("After Removing a4_0 and a11_0",hm);

    //Remove all entires
    hm.clear();
    printCS("After Removing all entr
  }

  private void testBench() {
    //Key is String. Value is Complex  SSN --> person
```

```
============ After Insertion  ++++++
For Key 0+j0 value is  a0_0
For Key 4+j0 value is  a4JNEW_0
For Key 5+j20 value is  a5_20
```

```
============  Testing contains +++++
Hash has 5+j20
Hash has 5+j20
```

```
==Testing changing values
Hash has 5+j20
Hash has 5+j20
Hash DOES NOT has 10+j0
```

```
===After Change
For Key 0+j0 value is  a0_0
For Key 4+j0 value is  a4JNEW_0
For Key 5+j20 value is  Changeda5_20z
For Key 10+j0 value is  a10_0z
```

```
After Removing a4_0 and a11_0

For Key 0+j0 value is  a0_0
For Key 5+j20 value is  Changeda5_20z
For Key 10+j0 value is  a10_0z
```

```
    test_String_Complex() ;
    //Key is Complex. Value is String  person  -> SSN
    test_Complex_String() ;
  }

  public static void main(String[] args) {
    System.out.println("TestTreeMap.java");
    TestTreeMap t = new TestTreeMap() ;
    t.testBench();
    System.out.println("TestTreeMap.java Done");
  }
}
```

13.11 Quiz

1

A BST is constructed(without balancing) from the array
{50, 15, 62, 5, 20, 58, 91, 3, 8, 37,60,24}

What is the number of nodes
in left subtree and right subtree
{l,r}

2

A binary search tree is used to locate the number 43. Which of the following probe sequences are possible and which are not? Explain.

(a)	61	52	14	17	40	43
(b)	2	3	50	40	60	43
(c)	10	65	31	48	37	43
(d)	81	61	52	14	41	43
(e)	17	77	27	66	18	43

3

Suppose the numbers 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 are inserted in that order into an initially empty binary search tree. The binary search tree uses the usual ordering on natural numbers. What is the in-order traversal sequence of the resultant tree ?

(A) 7 5 1 0 3 2 4 6 8 9

(B) 0 2 4 3 1 6 5 9 8 7

(C) 0 1 2 3 4 5 6 7 8 9

(D) 9 8 6 4 2 3 0 1 5 7

Figure 13.15: quiz

4

The following numbers are inserted into an empty binary search tree in the given order: 10, 1, 3, 5, 15, 12, 16. What is the height of the binary search tree (the height is the maximum distance of a leaf node from the root)?

A 2
B 3
C 4
D 6

5

A BST contains the numbers {1,2,3,4,5,6,7,8}
If you do preorder You get {5,3,1,2,4,6,8,7}
What number we will get if we do postorder?

Figure 13.16: quiz

6

BST has a number between 1 to 100
We are looking for 55
Which of the sequence CANNOT be happen while finding 55?

A)10,75,64,43,60,57,55
B)90,12,68,34,62,45,55
C)9,85,47,68,43,57,55
D)7,9,14,72,56,16,53,55

7

A binary search tree contains the value 1, 2, 3, 4, 5, 6, 7, 8. The tree is traversed in pre-order and the values are printed out. Which of the following sequences is a valid output?

(a) 5 3 1 2 4 7 8 6
(b) 5 3 1 2 6 4 8 7
(c) 5 3 2 4 1 6 7 8
(d) 5 3 1 2 4 7 6 8

Figure 13.17: quiz

13.12 Problem set

Problem 13.12.1. Solve the problem shown in figure 13.18 by hand.

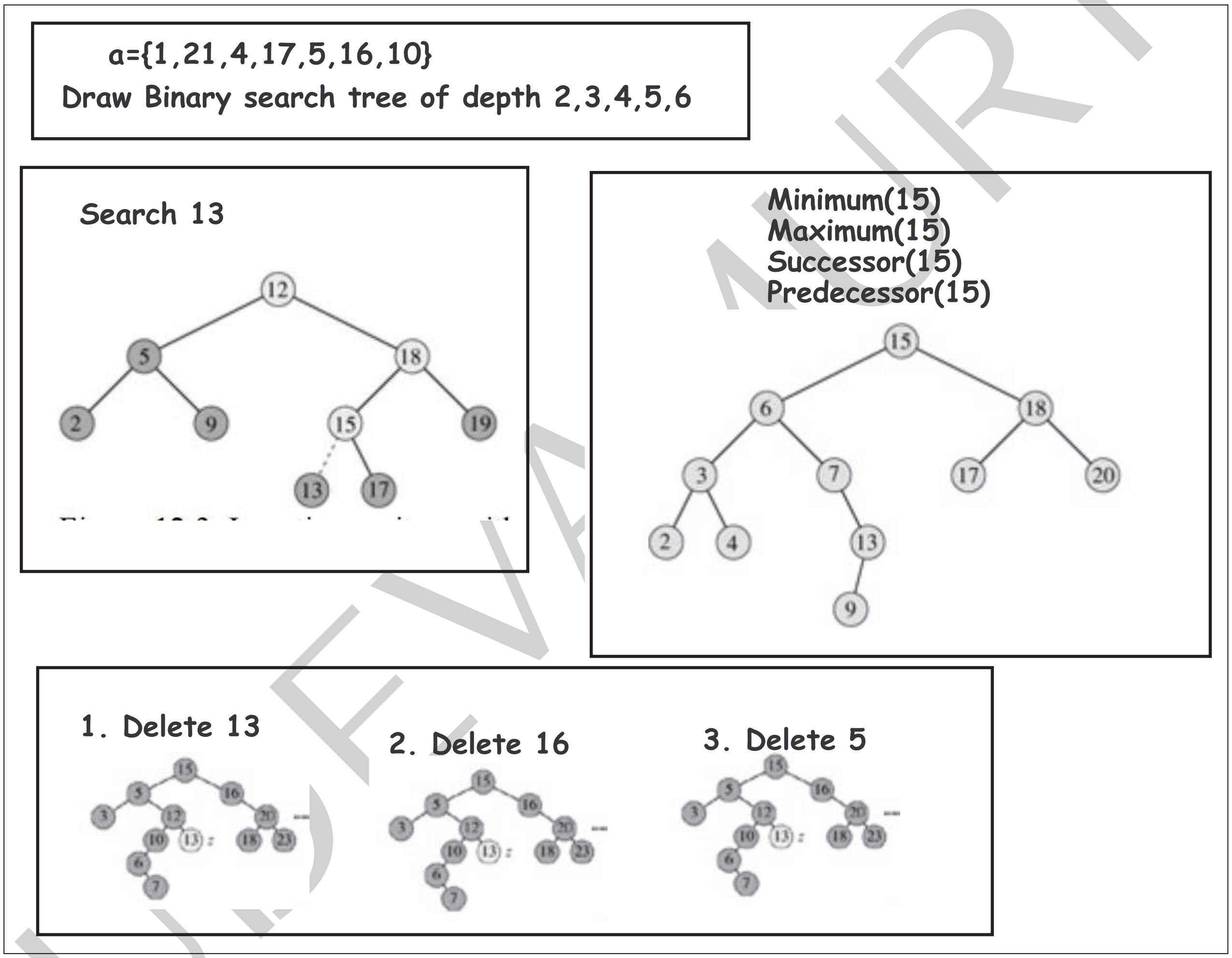


Figure 13.18: Various operations on BST