

CSC 501/401

Lectures on Analysis of Algorithms

by

Dr. Marek A. Suchenek ©

Computer Science
CSUDH

Copyrighted material

All rights reserved

Copyright by Dr. Marek A. Suchenek ©
and
Addison-Wesley ©

CSC 501/401

A brief review of **Trees**

(Excerpts from CSC 311 Data Structures slides)

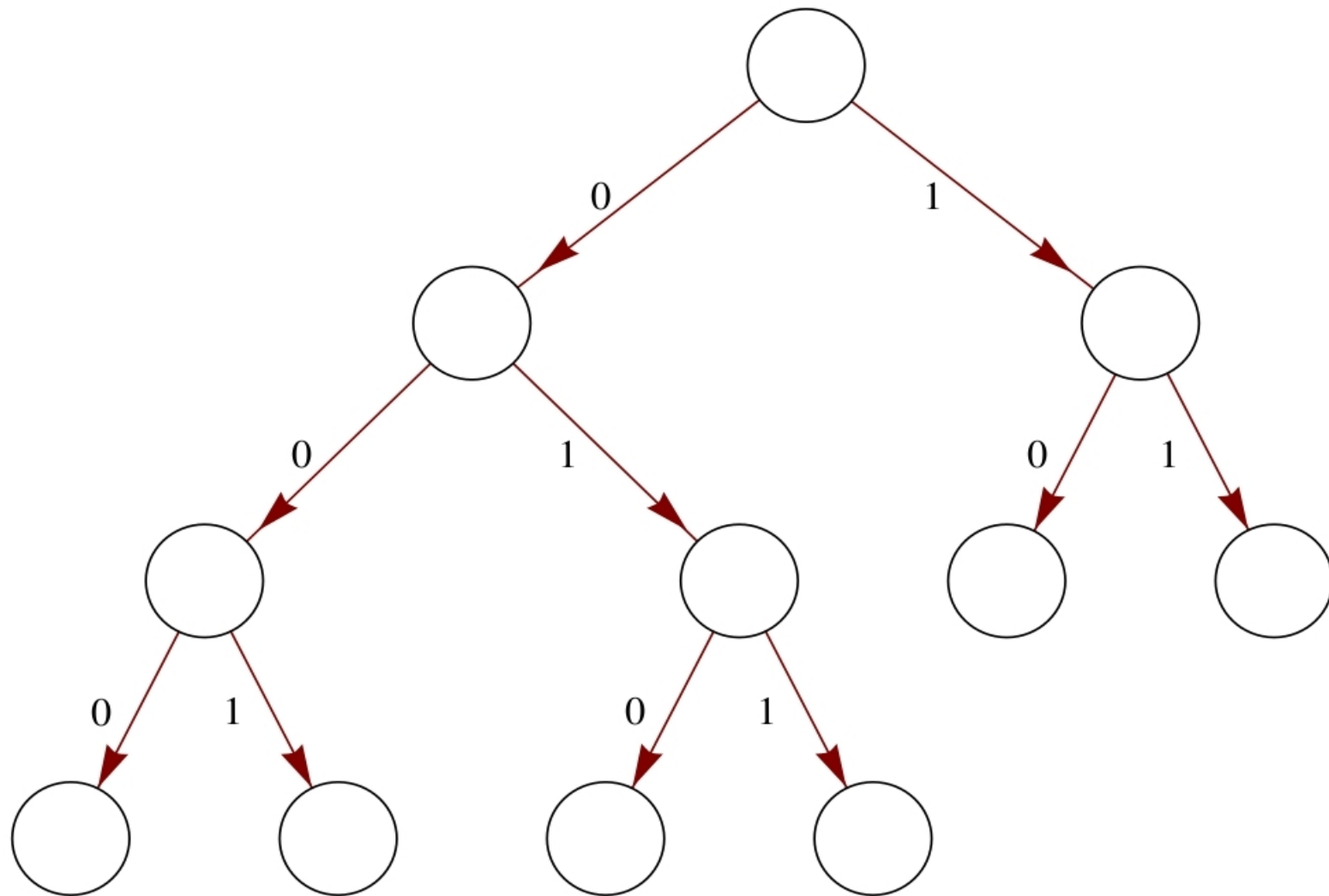
**Definitions, Applications, Implementations,
Analysis**

Definitions of tree

1. A tree is an acyclic and connected graph.

If it's non-empty then one of its nodes is designated as the root.

Definitions of tree



Definitions of tree

2. A tree is a set of sequences closed under operation of taking a beginning subsequence.

(This is sometimes referred to as a "tree of paths".)

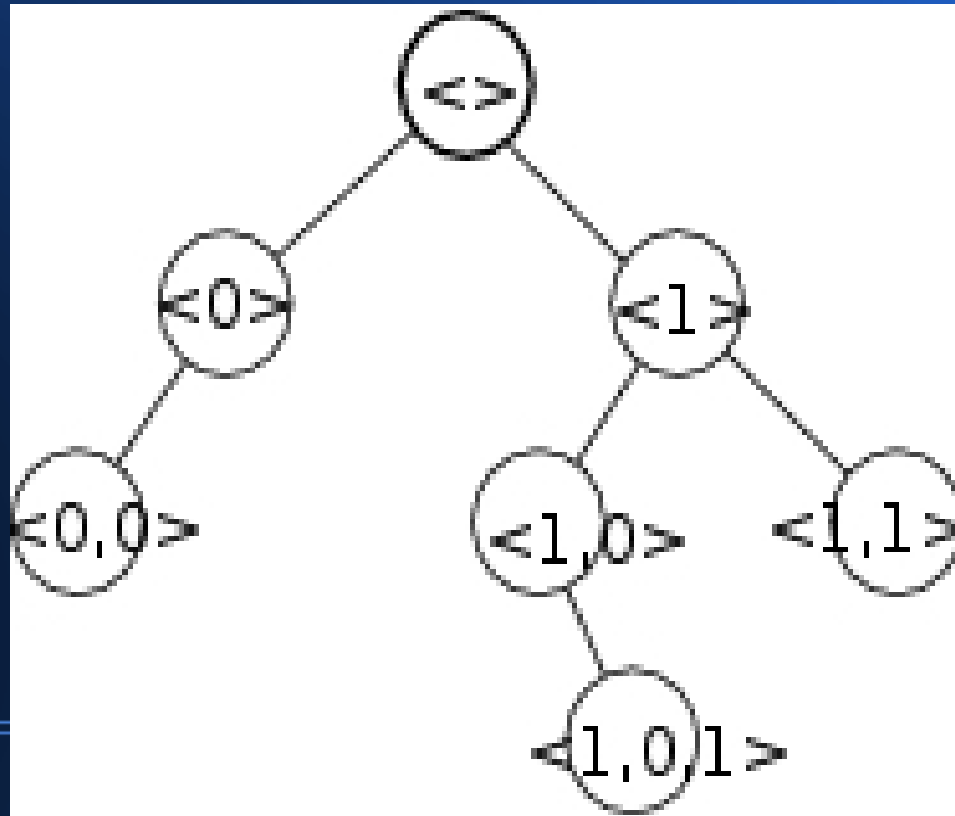
Definitions of tree

Example:

$$\{ \langle \rangle, \langle 0 \rangle, \langle 1 \rangle, \langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle, \langle 1, 0, 1 \rangle \}$$

Definitions of tree

$\{ \langle \rangle, \langle 0 \rangle, \langle 1 \rangle, \langle 0,0 \rangle, \langle 1,0 \rangle, \langle 1,1 \rangle, \langle 1,0,1 \rangle \}$



Definitions of tree

3. A **finite** tree is any of the following:

Definitions of tree

3. A **finite** tree is any of the following:

(i) the empty set

Definitions of tree

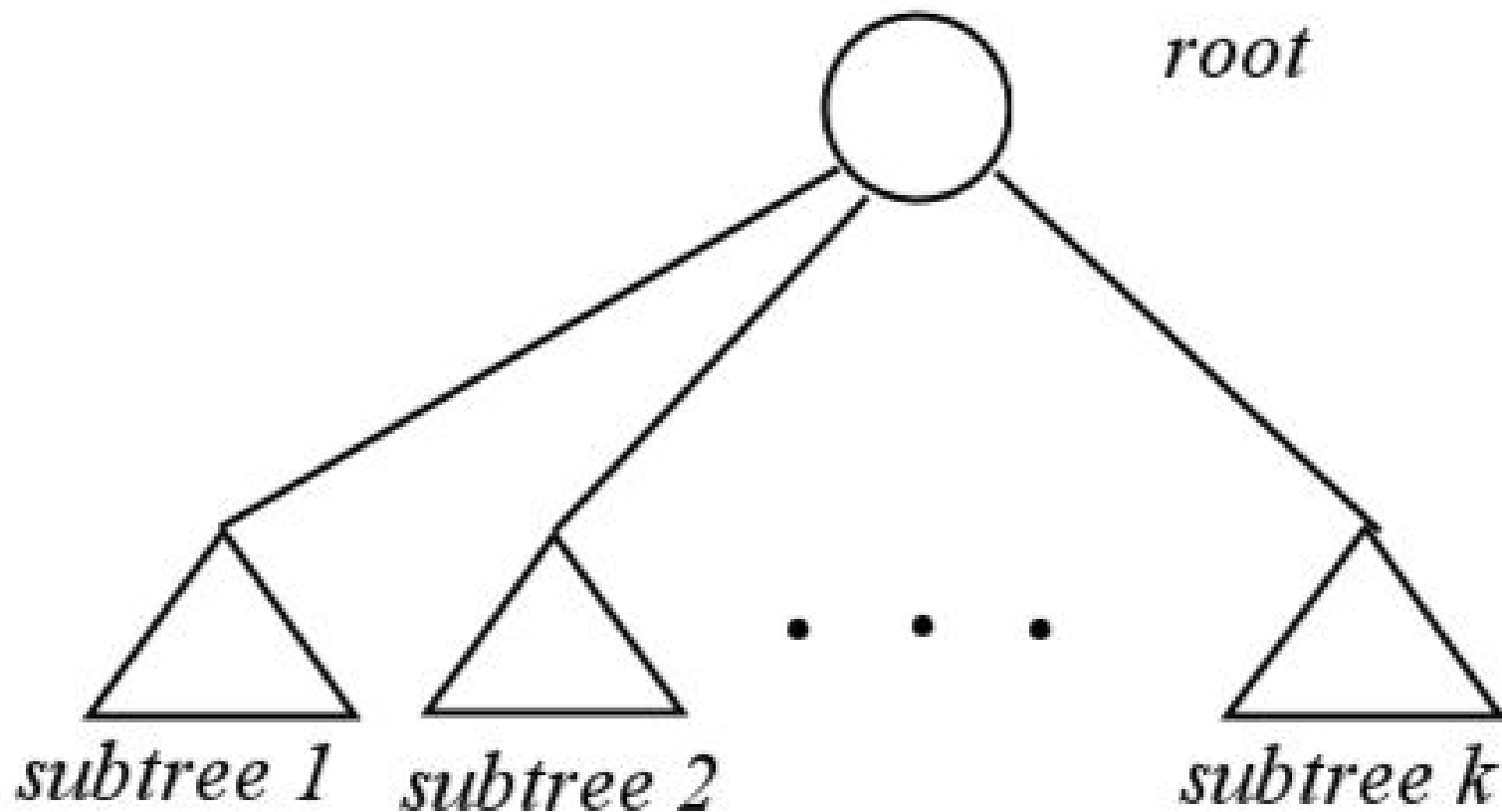
(ii) the root (a node) with some number of **finite** subtrees attached to it. (If the attached subtree is non-empty then the attachment has a form of an edge that goes from the root of the tree to the root of the subtree in question.)

Definitions of tree

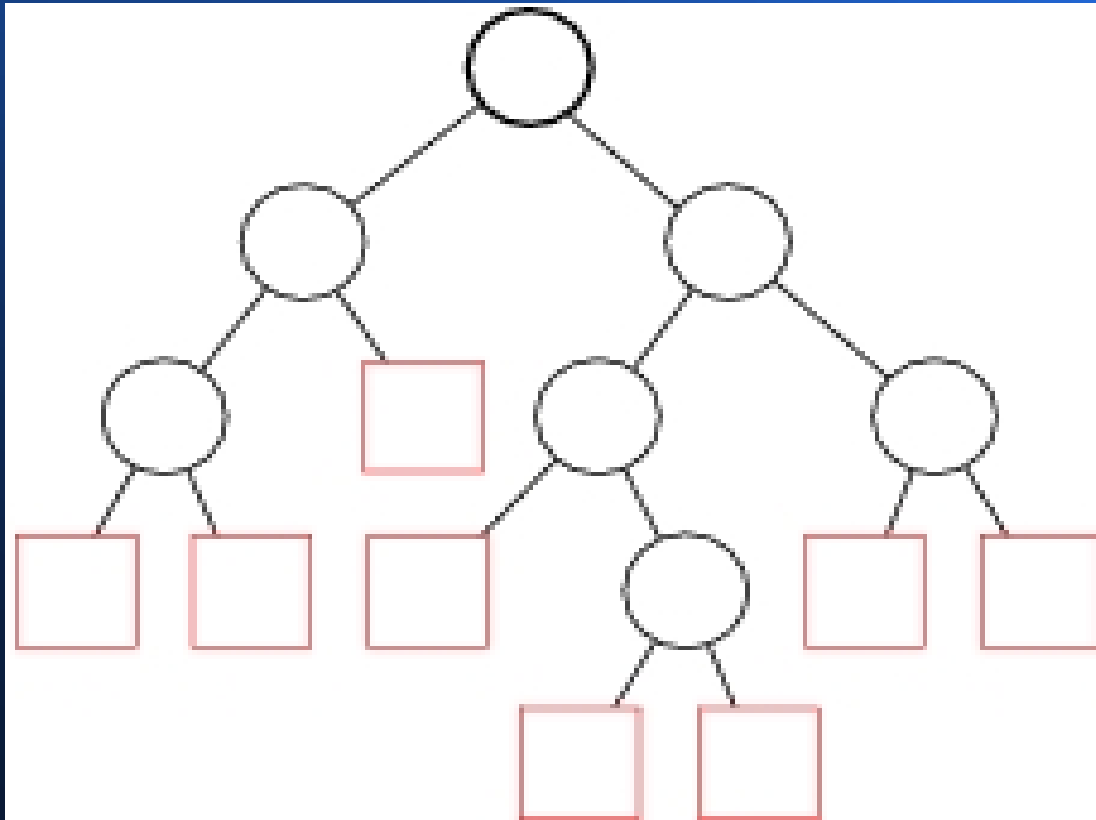
(iii) Nothing else is a finite tree.

Definitions of tree

Tree - non-empty



Definitions of tree

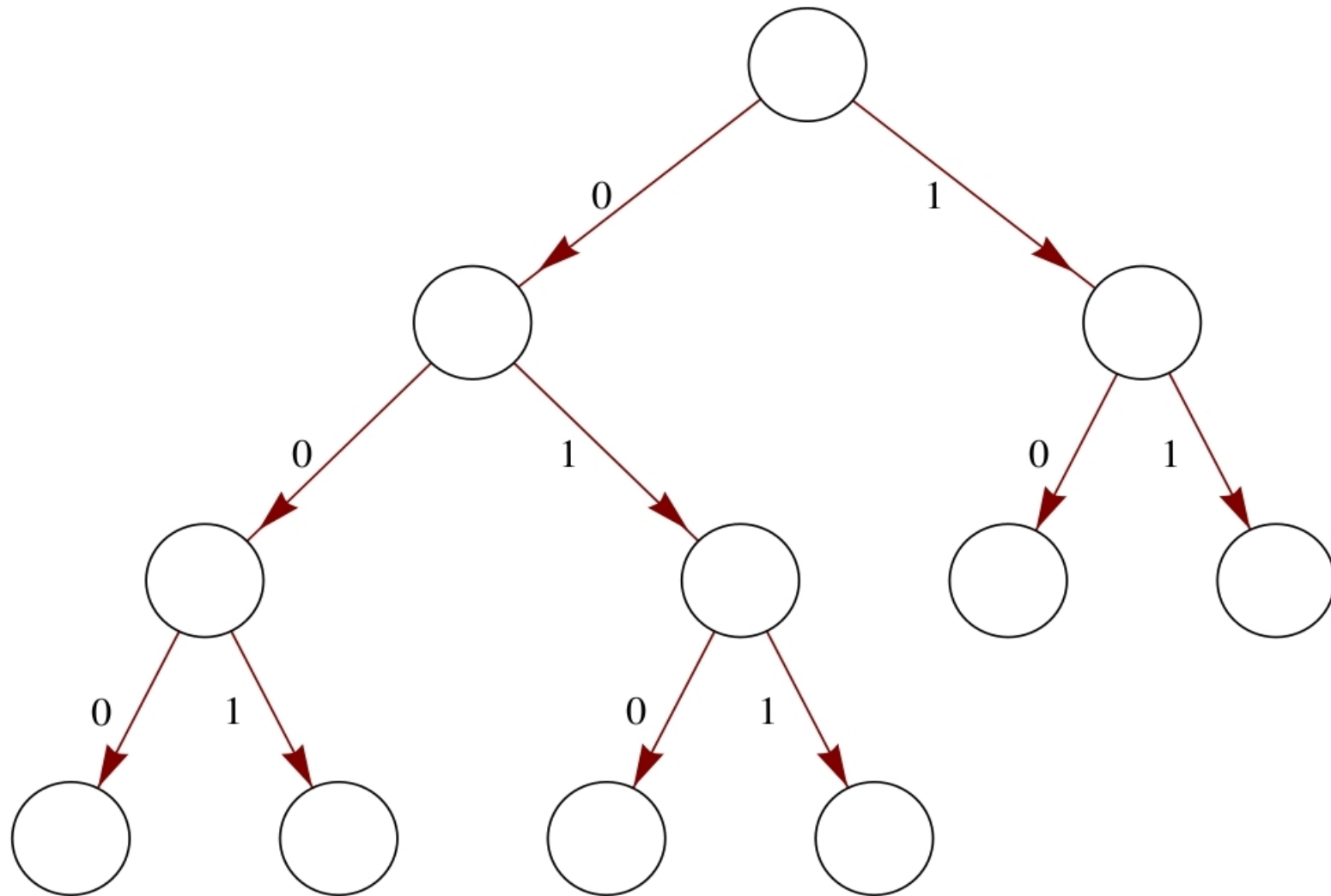


Boxes represent empty subtrees

Definitions of **binary** tree

A tree whose root has at most two edges incident on it and any node other than the root has at most three edges incident on it.

Definitions of **binary** tree



Definitions of **binary** tree

2. A **binary** tree is a set of **binary** sequences closed under operation of taking a beginning subsequence.

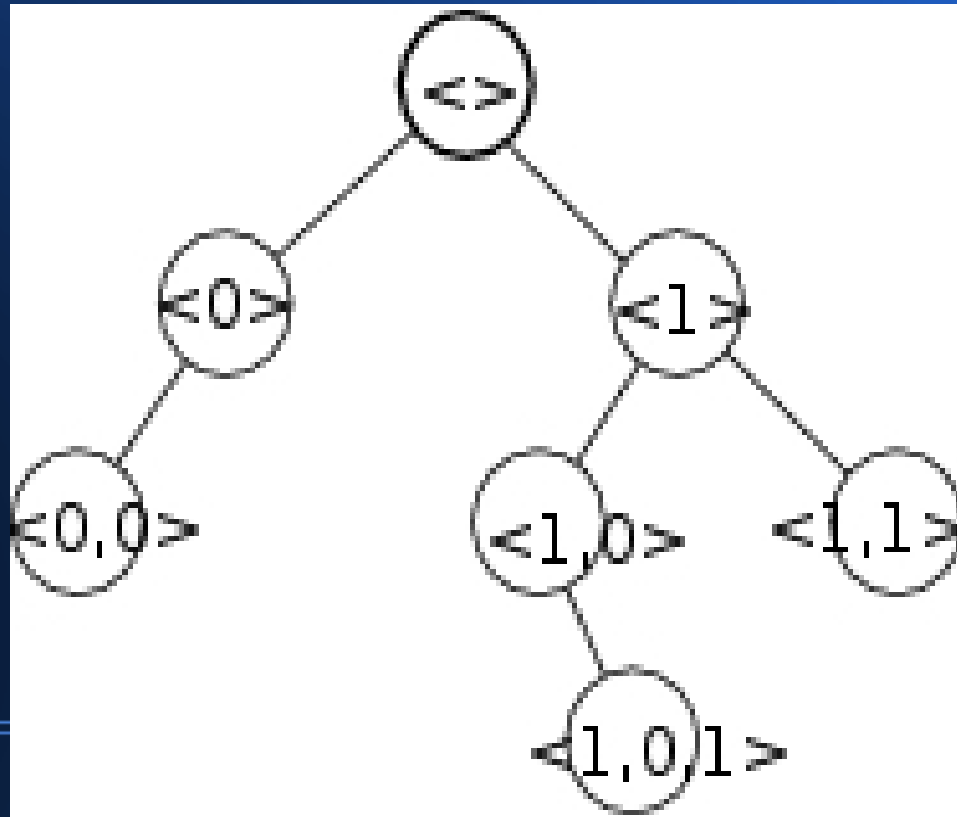
Definitions of **binary** tree

Example:

$$\{ \langle \rangle, \langle 0 \rangle, \langle 1 \rangle, \langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle, \langle 1, 0, 1 \rangle \}$$

Definitions of **binary** tree

$\{ \langle \rangle, \langle 0 \rangle, \langle 1 \rangle, \langle 0,0 \rangle, \langle 1,0 \rangle, \langle 1,1 \rangle, \langle 1,0,1 \rangle \}$



Definitions of **binary** tree

2a. A **binary** tree is a set of positive integers closed under operation of positive integer division by **2**.

Definitions of **binary** tree

$\{1, 2, 3, 4, 6, 7, 13\}$

Definitions of **binary** tree

$\{1, 2, 3, 4, 6, 7, 13\}$

Was: $\{<>, <0>, <1>, <0,0>, <1,0>, <1,1>, <1,0,1>\}$

Definitions of **binary** tree

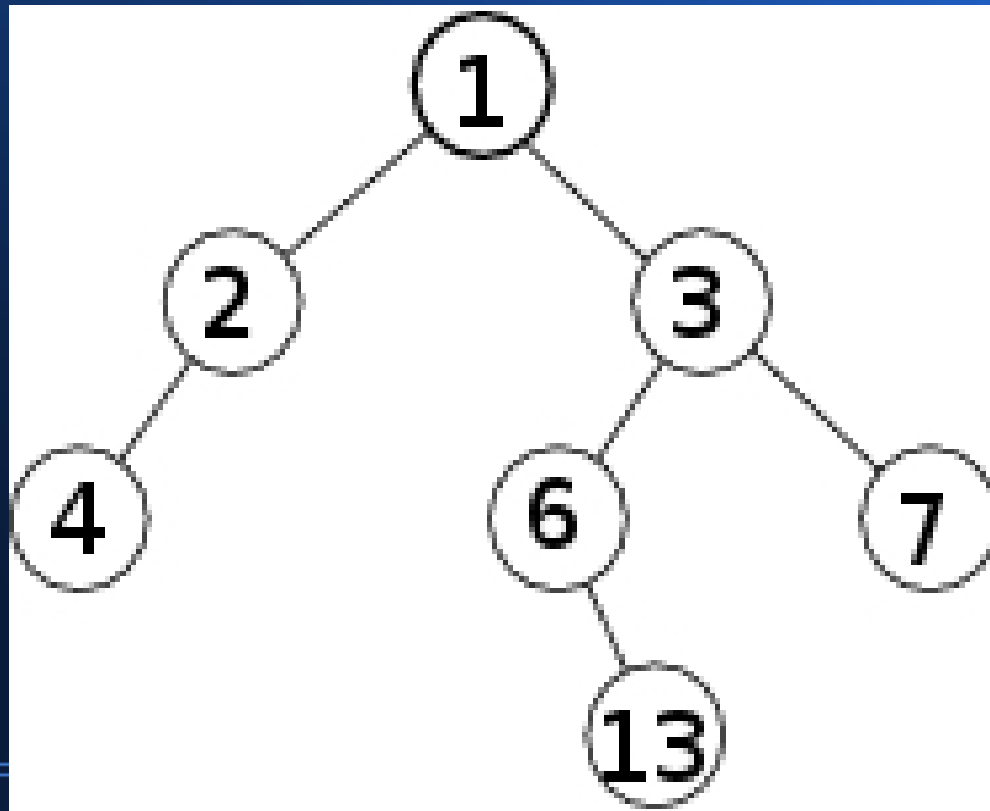
$\{1, 2, 3, 4, 6, 7, 13\}$

Was: $\{<>, <0>, <1>, <0,0>, <1,0>, <1,1>, <1,0,1>\}$

Is: $\{<\mathbf{1}>, <\mathbf{1},0>, <\mathbf{1},1>, <\mathbf{1},0,0>, <\mathbf{1},1,0>, <\mathbf{1},1,1>, <\mathbf{1},1,0,1>\}$

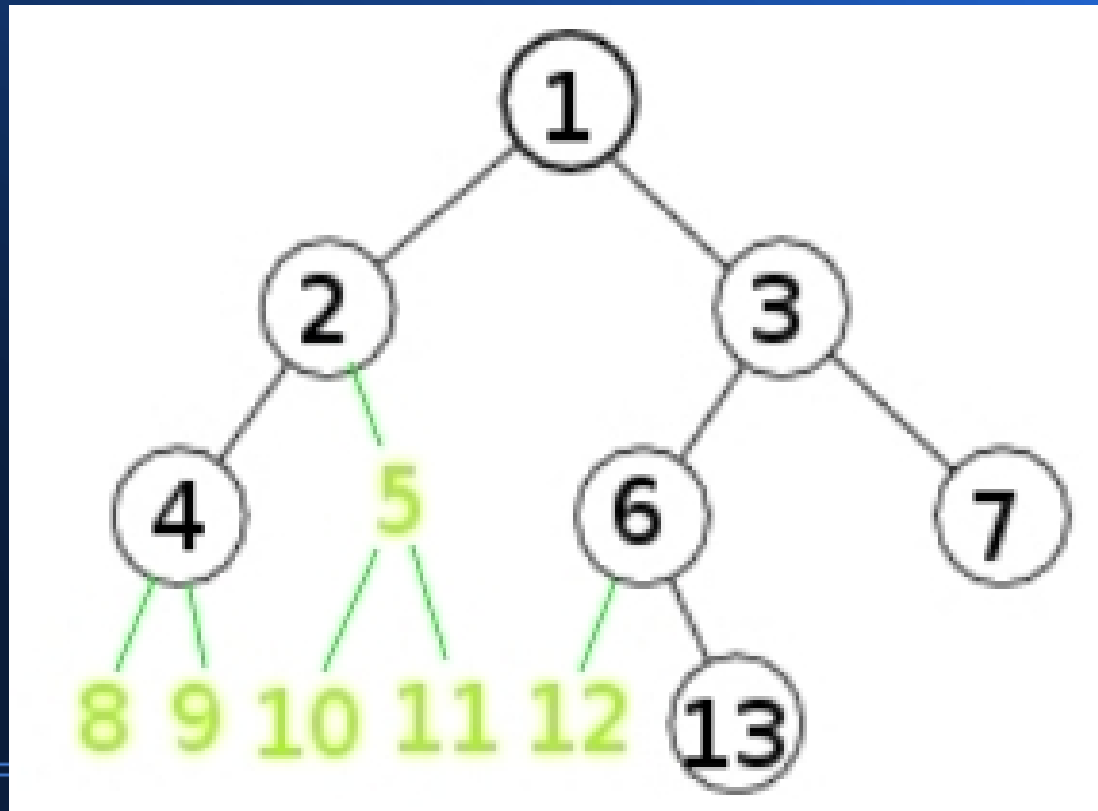
Definitions of **binary** tree

$\{1, 2, 3, 4, 6, 7, 13\}$



Definitions of **binary** tree

$\{1, 2, 3, 4, 6, 7, 13\}$



Def. of **convex** **binary** tree

2b. A **convex** **binary** tree is a set of first n positive integers.

Def. of **complete binary** tree

2b. A **convex binary** tree is a set of first n positive integers.

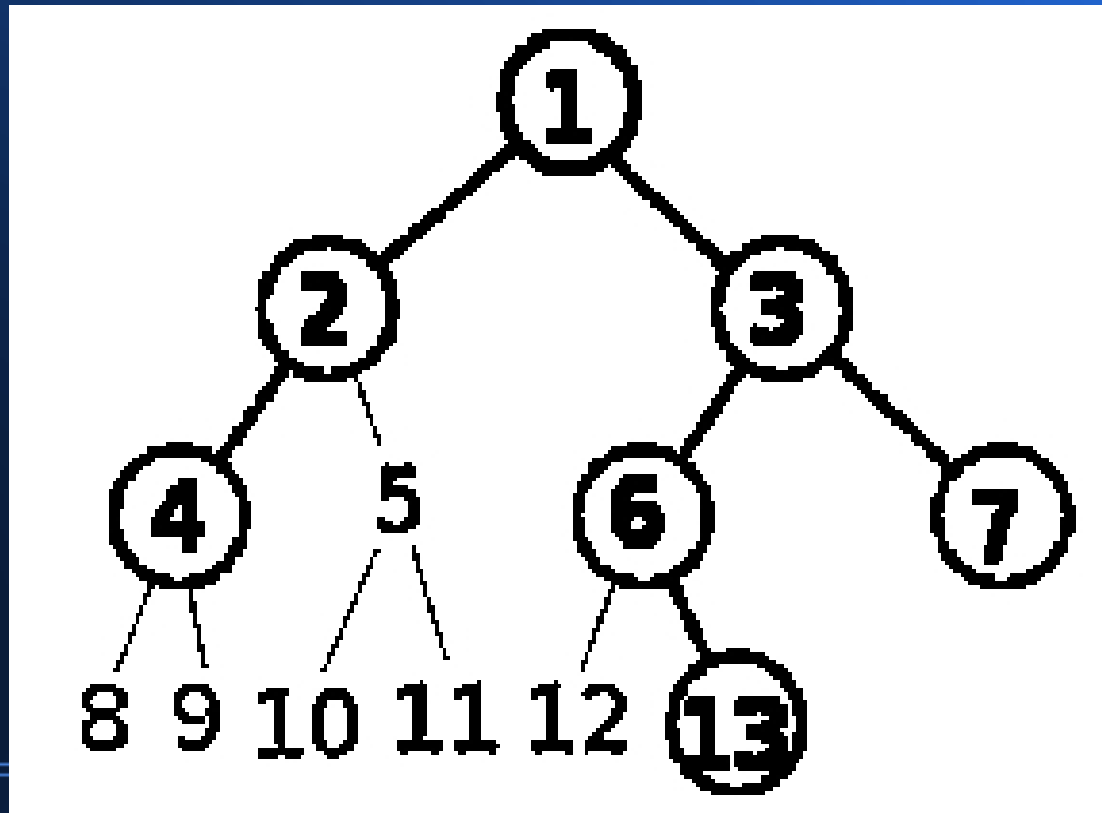
Of course, it is closed under operation of positive integer division by **2**.

Def. of **complete** **binary** tree

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13}

Def. of **complete** **binary** tree

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13}



Definitions of **binary** tree

3. A **finite binary** tree is any of the following:

(i) the empty set

Definitions of **binary** tree

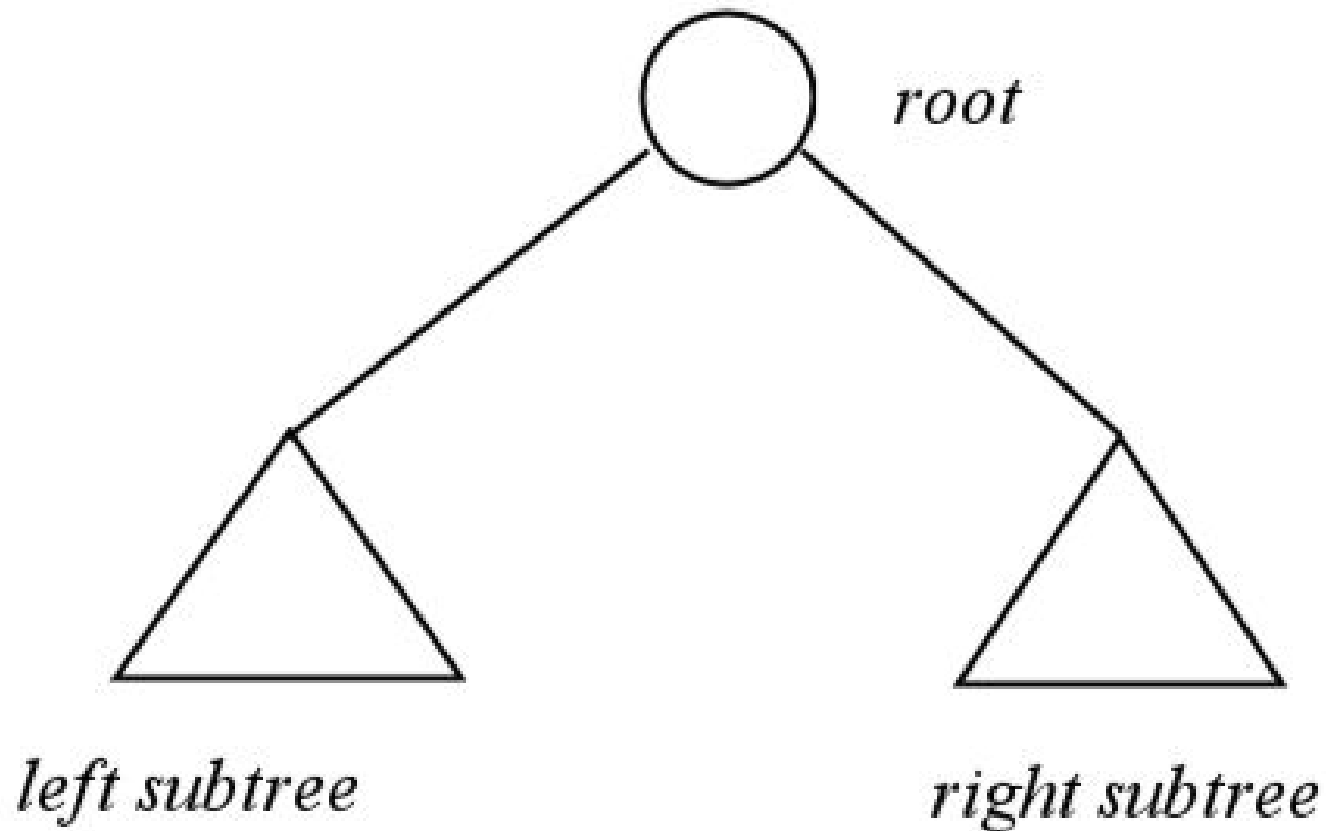
(ii) the root (a node) with **two** **finite** subtrees attached to it. (If the attached subtree is non-empty then the attachment has a form of an edge that goes from the root of the tree to the root of the subtree in question.)

Definitions of **binary** tree

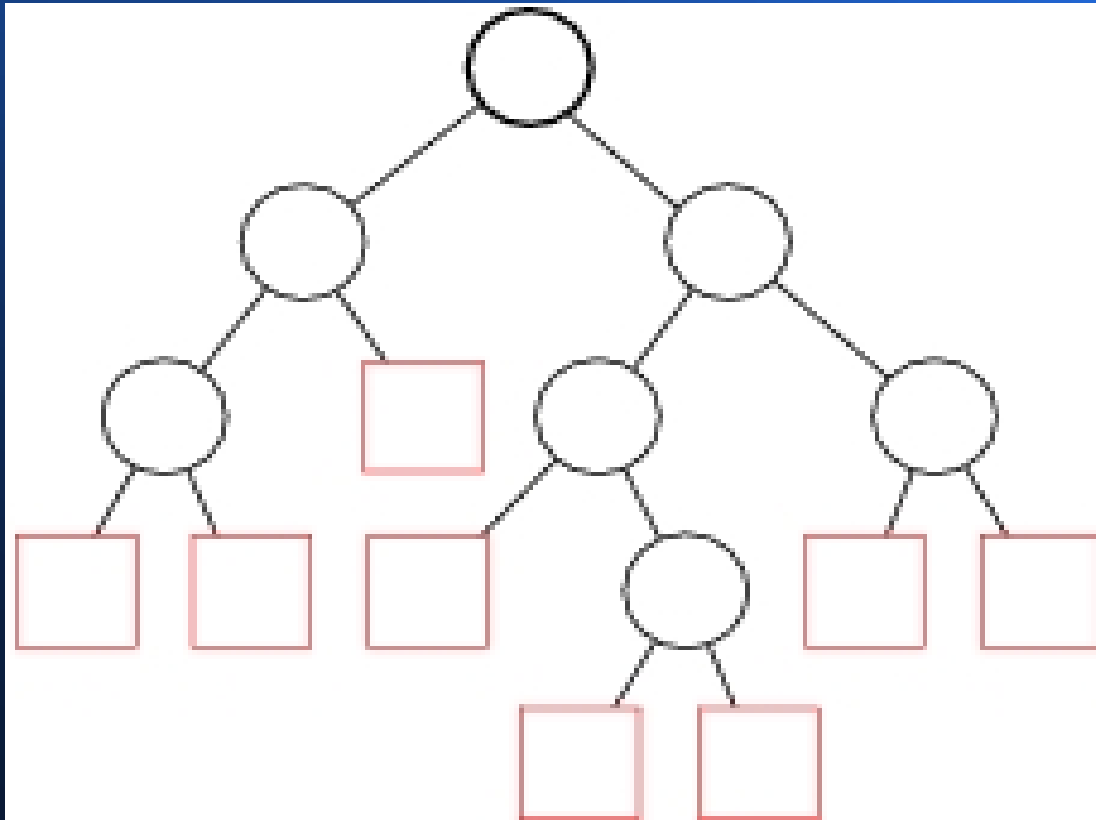
(iii) Nothing else is a finite tree.

Definitions of **binary** tree

Binary tree - non-empty



Definitions of **binary** tree



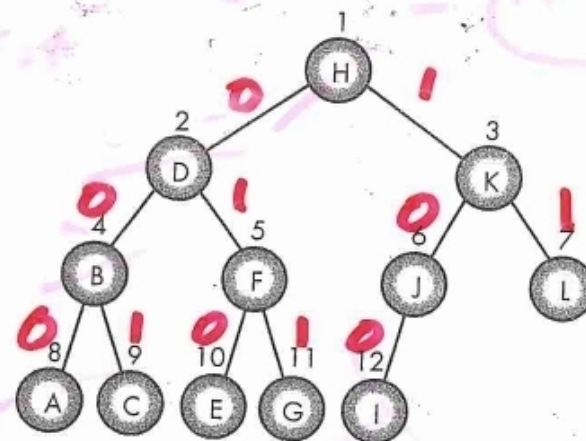
Boxes represent empty subtrees

8.4 A Sequential Binary Tree Representation

LEARNING OBJECTIVES

1. To learn about one of the important sequential representations of complete binary trees.
2. To learn how to find the parents and children of nodes in this sequential representation.
3. To learn the conditions for a node being a root, a leaf, and an internal node in this representation.

numbering nodes
level-by-level



A:

	H	D	K	B	F	J	L	A	C	E	G	I
0	1	2	3	4	5	6	7	8	9	10	11	12

Figure 8.6 Sequential Representation of a Complete Binary Tree (with A[0] Empty)

To Find:	Use:	Provided:
The left child of $A[i]$	$A[2i]$	$2i \leq n$
The right child of $A[i]$	$A[2i + 1]$	$2i + 1 \leq n$
The parent of $A[i]$	$A[i / 2]$	$i > 1$
The root	$A[1]$	A is nonempty
Whether $A[i]$ is a leaf	true	$2i > n$

$n > 0$

8.4 EXERCISES

1, 2, 3 p 252

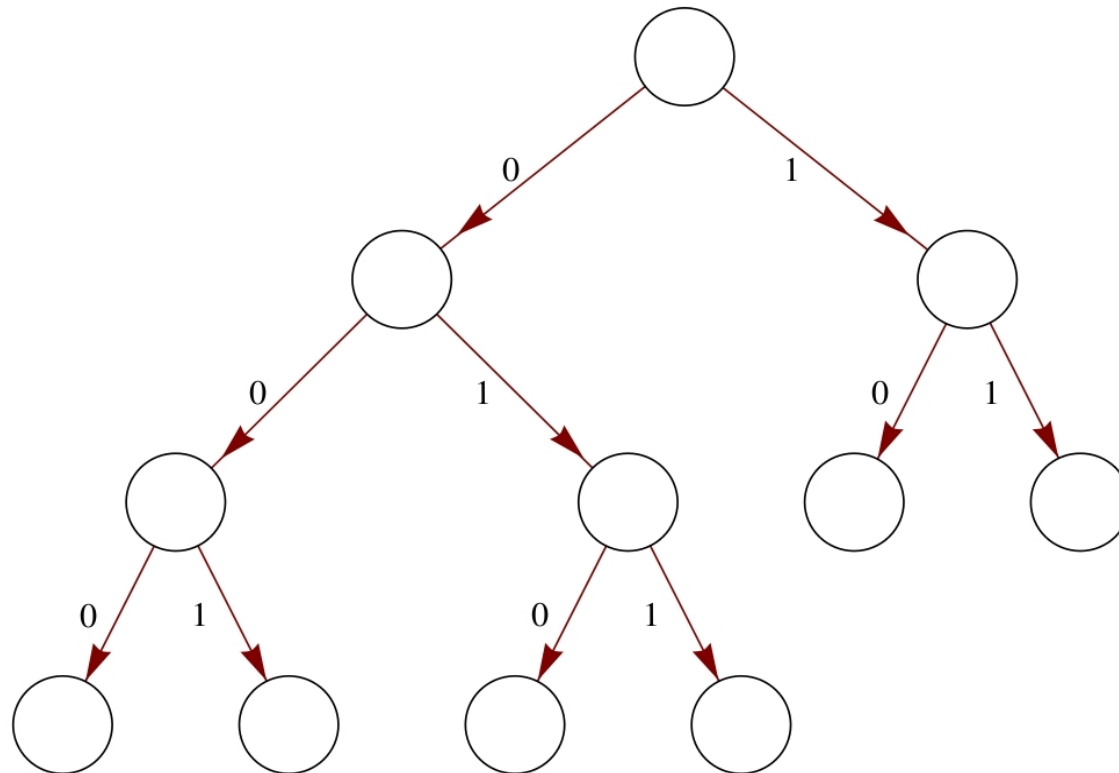
8.5

An Application—Heaps and Priority Queues

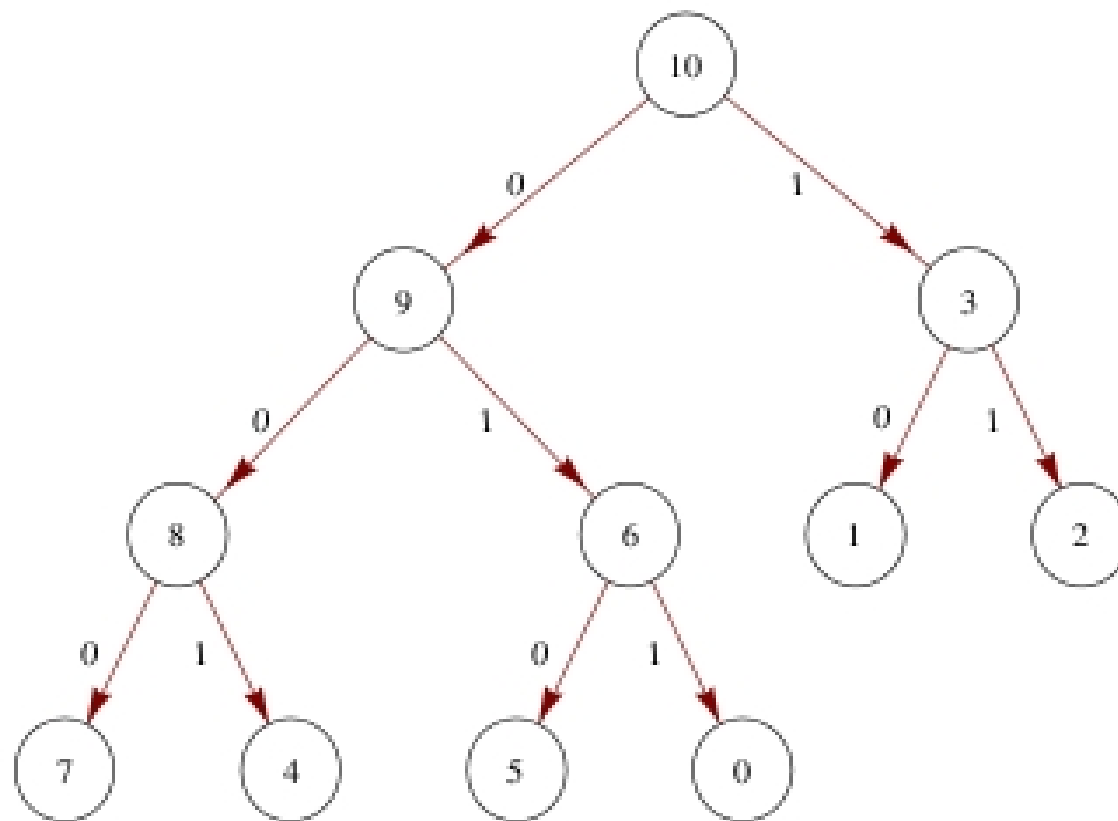
LEARNING OBJECTIVES

1. To learn how to represent a heap using a contiguous sequential representation.
2. To learn how heaps can serve as efficient representations for priority queues.
3. To discover some important mathematical properties of heaps that will be used later.

Convex tree



Heap



Definition of heap

A **heap** is a complete binary tree with values stored in its nodes such that no child has a value greater than the value of its parent.

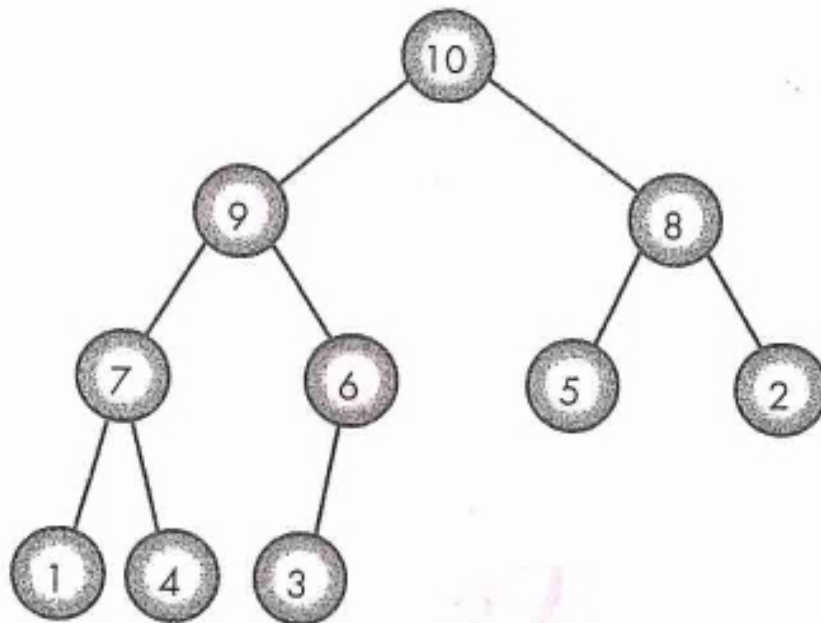


Figure 8.8 An Example of a Heap

Priority Queue (heap)

```
5  |  /*  
   |  *   The public interface for the PriorityQueue class contains  
   |  *   the following method calls. Here, let PQ be a variable having  
   |  *   a PriorityQueue object as its value, let X be a variable that  
   |  *   contains a priority queue item, and let n be an integer variable.  
   |  */  
   |  
   |  PQ = new PriorityQueue();           // creates an initially empty priority queue PQ  
10 |  
   |  n = PQ.size();                      // returns the number of items in PQ and  
   |                                     // stores it in the integer variable n  
   |  
   |  PQ.insert(X);                      // puts X into PQ  
15 |  
   |  X = PQ.remove( );                  // removes the highest priority item from PQ and  
   |                                     // assigns it to be the value of the variable X
```

Program 5.1 Informal Interface for a PriorityQueue Class

Priority Queue (heap)

```
17 public class PriorityQueue {  
18  
19     //Uses heap as implementation  
20  
21  
22     private int count; //actual number of elements  
23     private int capacity; //the size of the array - 1  
24     private int capacityIncrement;  
25     private int[] itemArray;  
26
```

Priority Queue (heap)

```
27 ☐ /** Creates a new instance of PriorityQueue */
28 ☐ public PriorityQueue() {
29     count=0;
30     capacity=10;
31     capacityIncrement=2;
32     itemArray=new int[capacity + 1]; //itemArra
33 }
34
```

Priority Queue (heap)

```
35     public void insert(int newItem)
36     {
37         if(count==capacity) //no more space, "resize"
38         {
39             capacity*=capacityIncrement;
40             int[] tempArray = new int[capacity + 1];
41             for (int i = 1; i <= count; i++)
42             {
43                 tempArray[i] = itemArray[i];
44                 cnt2.incr();
45             }
46             itemArray = tempArray;
```

Priority Queue (heap)

```
48     }
49     //try insert at the end
50     count++; //1st element sits at index 1, 2nd element at index 2, etc,..
51     //the newly inserted element may be too large to be a leaf
52     int i = count; //initial "logical" position of newItem
53     //we keep it in itemArray[0] to save time on swapping
54     while ((i > 1) && (cnt3.incr() & (newItem > itemArray[i/2])))
55     {
56         itemArray[i] = itemArray[i/2]; //demote the parent
57         cnt.incr();
58         i = i/2; //promote the new one i /= 2;
59     }
```

Priority Queue (heap)

```
60         //here i is the index for the newly inserted element
61         itemArray[i] = newItem;
62         cnt.incr();
63     }
64
```

Priority Queue (heap)

```
65     public int remove()  
66     {  
67         if (count==0) return -9999;  
68         //here count != 0  
69         int maxItem = itemArray[1]; //the root  
70         int demotee = count;  
71         count--;  
72         int i = 1;  
73         boolean demoted = true;
```

Priority Queue (heap)

```
74     while ((2*i <= count) && demoted)
75     {
76         int j = 2*i; // first child
77         if ((j < count) && (cnt3.incr() & (itemArray[j] < itemArray[j + 1]))) j++;
78         if (cnt3.incr() & (itemArray[j] > itemArray[demotee])) //demote patch
79         {
80             itemArray[i] = itemArray[j]; //promote its larger child
81             cnt.incr();
82             i = j; //demote patch's index
83         }
84     }
```

Priority Queue (heap)

```
84         }
85         else
86         {
87             demoted = false;
88         }
89     }
90     //i is the place for the patch
91     itemArray[i] = itemArray[demotee];
92     cnt.incr();
93     itemArray[demotee] = 0; // for demonstration purpose only
94
```

Priority Queue (heap)

```
95     if ((count < capacity / capacityIncrement) && (10 <= capacity / capacityIncrement))
96     {
97         capacity/=capacityIncrement;
98         int[] tempArray = new int[capacity+1]; //because itemArray[0] is not used
99         for (i = 1; i <= count; i++)
100         {
101             tempArray[i] = itemArray[i];
102             cnt2.incr();
103         }
104         itemArray = tempArray;
105     }
106     return maxItem;
107 }
```

Excerpt from "Data Structures in Java" by Standish

how to sort using
priority queues

Non-decreasing
functions

```
void priorityQueueSort(ComparisonKey[ ] A) {  
    int i;  
    // let i be an integer array index variable  
    int n = A.length;  
    // let n be the length of the array A to be sorted  
    PriorityQueue PQ = new PriorityQueue( );  
    // let PQ be initially empty  
    for (i = 0; i < n; i++) PQ.insert(A[i]);  
    // put A's items into PQ  
    for (i = n-1; i >= 0; i--) A[i] = PQ.remove( );  
    // remove PQ's items  
    // and put them in A  
}
```

Program 5.2 A Priority Queue Sorting Method

worst case

$$\sum_{i=0}^{n-1} f(i) \leq \sum_{i=0}^{n-1} f(n-1) = n \cdot f(n-1) \leq n f(n)$$
$$\sum_{i=0}^{n-1} g(i+1) \leq \sum_{i=0}^{n-1} g(n) = n g(n)$$

$$\begin{aligned} \text{Total} &\leq O(1 + n \cdot f(n) + n \cdot g(n)) = O(1 + n(f(n) + g(n))) \\ &= O(n(f(n) + g(n))) = \\ &= O(n \cdot \max(f(n), g(n))) \end{aligned}$$

Priority Queue (heap)

6 5 3 1 8 7 2 4

Performance of heap

Insert to a heap with n nodes:

Worst-case

$$T(n) \in \Theta(\log n)$$

Performance of heap

Delete from a heap with n nodes:

Worst-case

$$T(n) \in \Theta(\log n)$$

Excerpt from "Data Structures in Java" by Standish

how to sort using
priority queues

Non-decreasing
functions

```
void priorityQueueSort(ComparisonKey[ ] A) {  
    int i;  
    // let i be an integer array index variable  
    int n = A.length;  
    // let n be the length of the array A to be sorted  
    PriorityQueue PQ = new PriorityQueue( );  
    // let PQ be initially empty  
    for (i = 0; i < n; i++) PQ.insert(A[i]);  
    // put A's items into PQ  
    for (i = n-1; i >= 0; i--) A[i] = PQ.remove( );  
    // remove PQ's items  
    // and put them in A  
}
```

Program 5.2 A Priority Queue Sorting Method

worst case

$$\sum_{i=0}^{n-1} f(i) \leq \sum_{i=0}^{n-1} f(n-1) = n \cdot f(n-1) \leq n f(n)$$
$$\sum_{i=0}^{n-1} g(i+1) \leq \sum_{i=0}^{n-1} g(n) = n g(n)$$

$$\begin{aligned} \text{Total} &\leq O(1 + n \cdot f(n) + n \cdot g(n)) = O(1 + n(f(n) + g(n))) \\ &= O(n(f(n) + g(n))) = \\ &= O(n \cdot \max(f(n), g(n))) \end{aligned}$$

Performance of heap

PriorityQueueSort:

Worst-case

$$T(n) \in \Theta(n \log n)$$

Tree Traversal

Pre-order

In-order

Post-order

Level-by-level

Tree Traversal

Animation

<http://www.cosc.canterbury.ac.nz/mukundan/dsal/BTree.html>

Tree Traversal

```
void preOrderTraversal(TreeNode T) {  
    Stack S = new Stack( );           // let S be an initially empty stack  
    TreeNode N;                       // N points to nodes during traversal  
5   S.push(T);                        // push the pointer T onto the empty stack S  
    while ( !S.empty( ) ) {  
10      N = (TreeNode)S.pop( );        // pop top pointer of S into N  
      if (N != null) {  
15        System.out.print(N.info);    // print N's info field  
        S.push(N.rlink);               // push the right pointer onto S  
        S.push(N.llink);              // push the left pointer onto S  
      }  
    }  
}
```

Program 8.27 PreOrder Traversal of an Expression Tree Using a Stack

Tree Traversal

```
void traverse(TreeNode T, int traversalOrder) {  
    /* to visit T's nodes in the order specified by the */  
    /* traversalOrder parameter */  
5   if (T != null) {                                     // if T == null, do nothing  
        if ( traversalOrder == PRE_ORDER ) {  
10            visit(T);  
            traverse(T.llink, PRE_ORDER);  
            traverse(T.rlink, PRE_ORDER);  
        } else if ( traversalOrder == IN_ORDER ) {  
15            traverse(T.llink, IN_ORDER);  
            visit(T);  
            traverse(T.rlink, IN_ORDER);  
        } else if ( traversalOrder == POST_ORDER ) {  
20            traverse(T.llink, POST_ORDER);  
            traverse(T.rlink, POST_ORDER);  
            visit(T);  
25        }  
    }  
}
```

Program 8.26 Generalized Recursive Traversal Method

Tree Traversal

```
void levelOrderTraversal(TreeNode T) {  
    Queue Q = new Queue( );           // let Q be an initially empty queue  
    TreeNode N;                       // N points to nodes during traversal  
    5    Q.insert(T);                   // insert the pointer T into queue Q  
    while ( ! Q.empty( ) ) {  
        10    N = (TreeNode) Q.remove( );           // remove first pointer of Q  
                                                    // and put it into N  
        if (N != null ) {  
            System.out.print(N.info);               // print N's info field  
            Q.insert(N.llink)                       // insert left pointer on rear of Q  
            15    Q.insert(N.rlink)                 // insert right pointer on rear of Q  
        }  
    }  
}
```

Program 8.28 LevelOrder Binary Tree Traversal Using Queues

BS Tree

Definition of binary search tree.

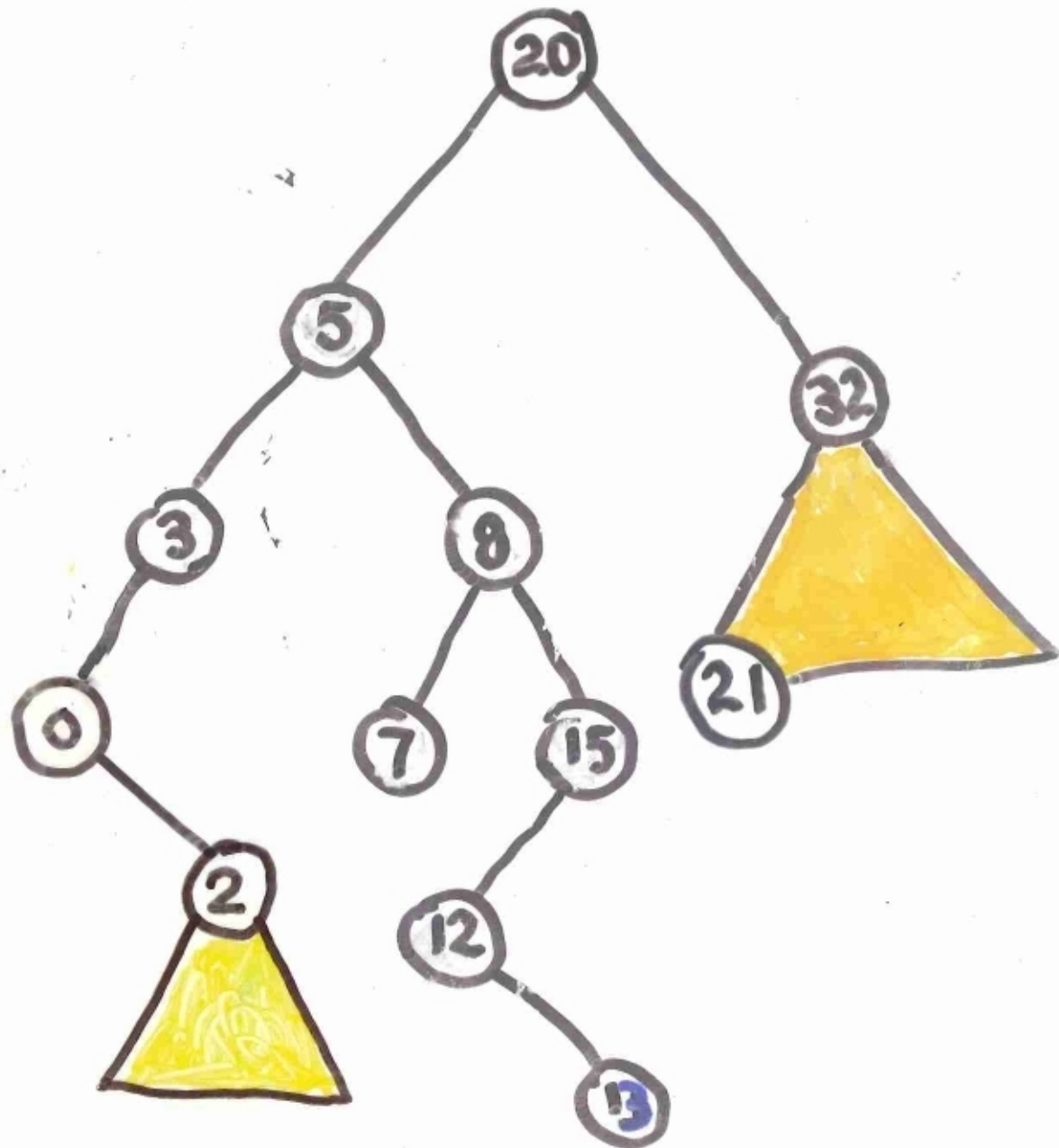
A binary tree T is called a binary search tree if, and only if, in-order traversal with listing of T lists the nodes of T in an increasing order.

BS Tree

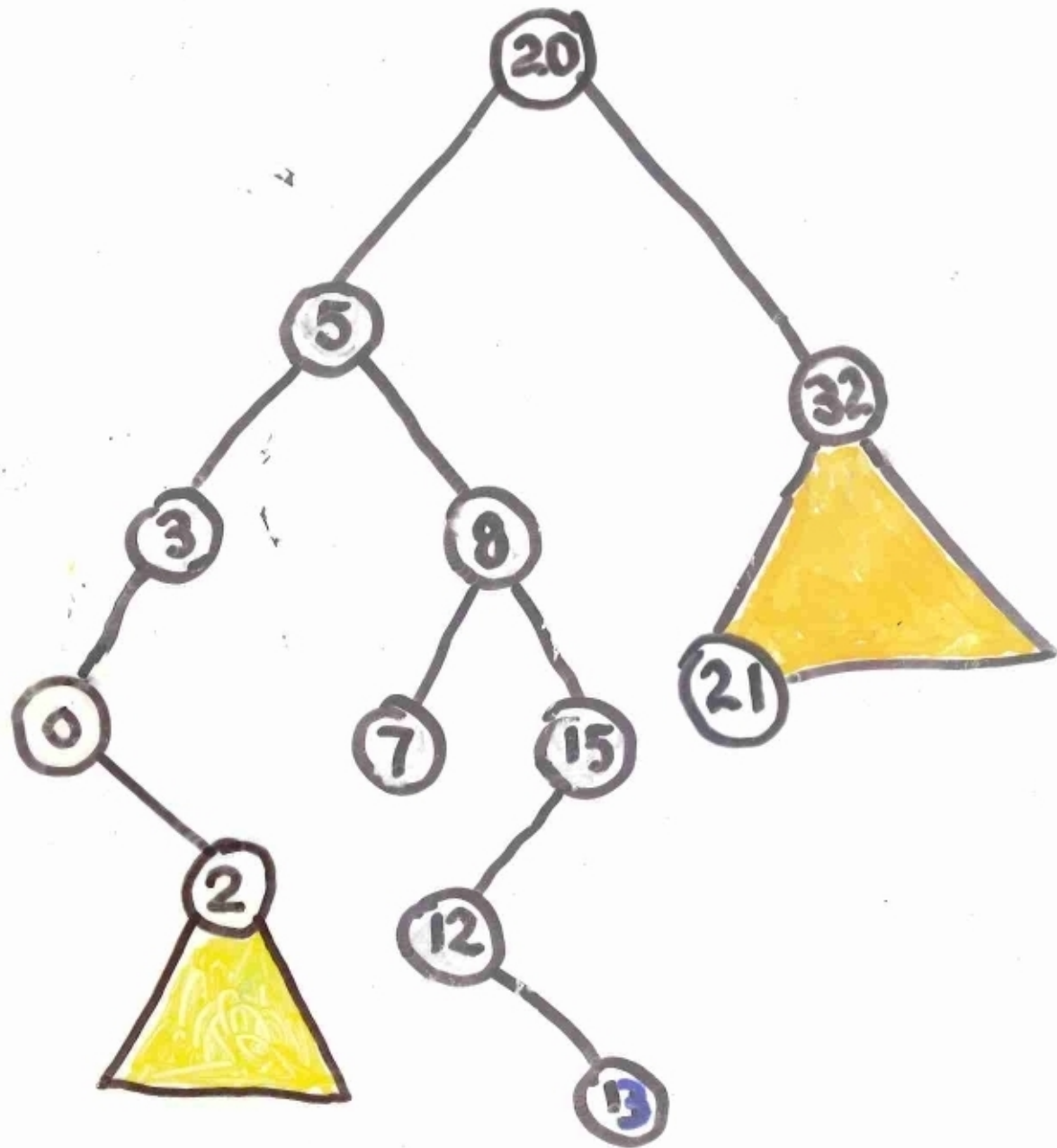
Exercise

<http://nova.umuc.edu/~jarc/idsv/lesson4.html>

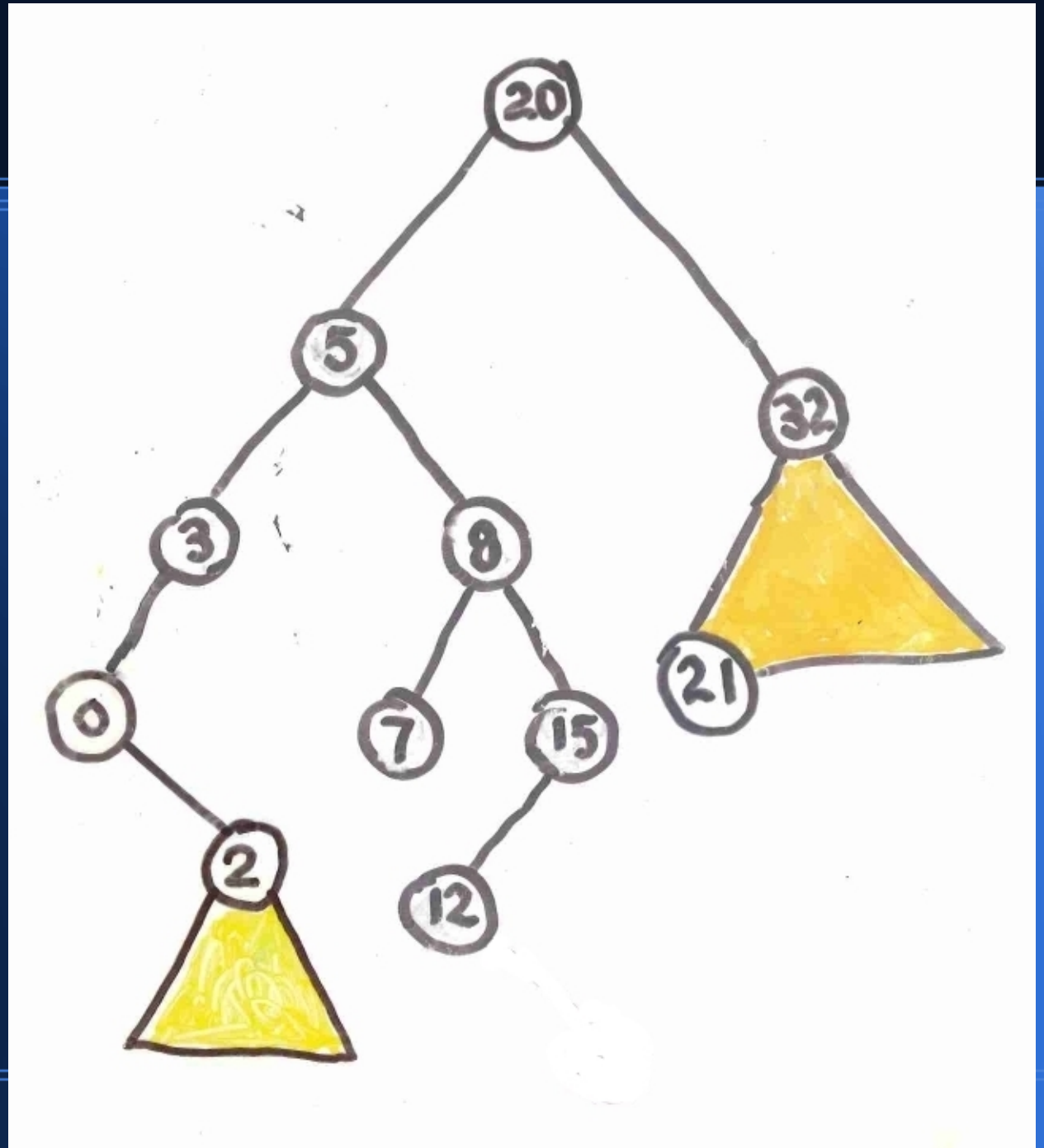
Delete



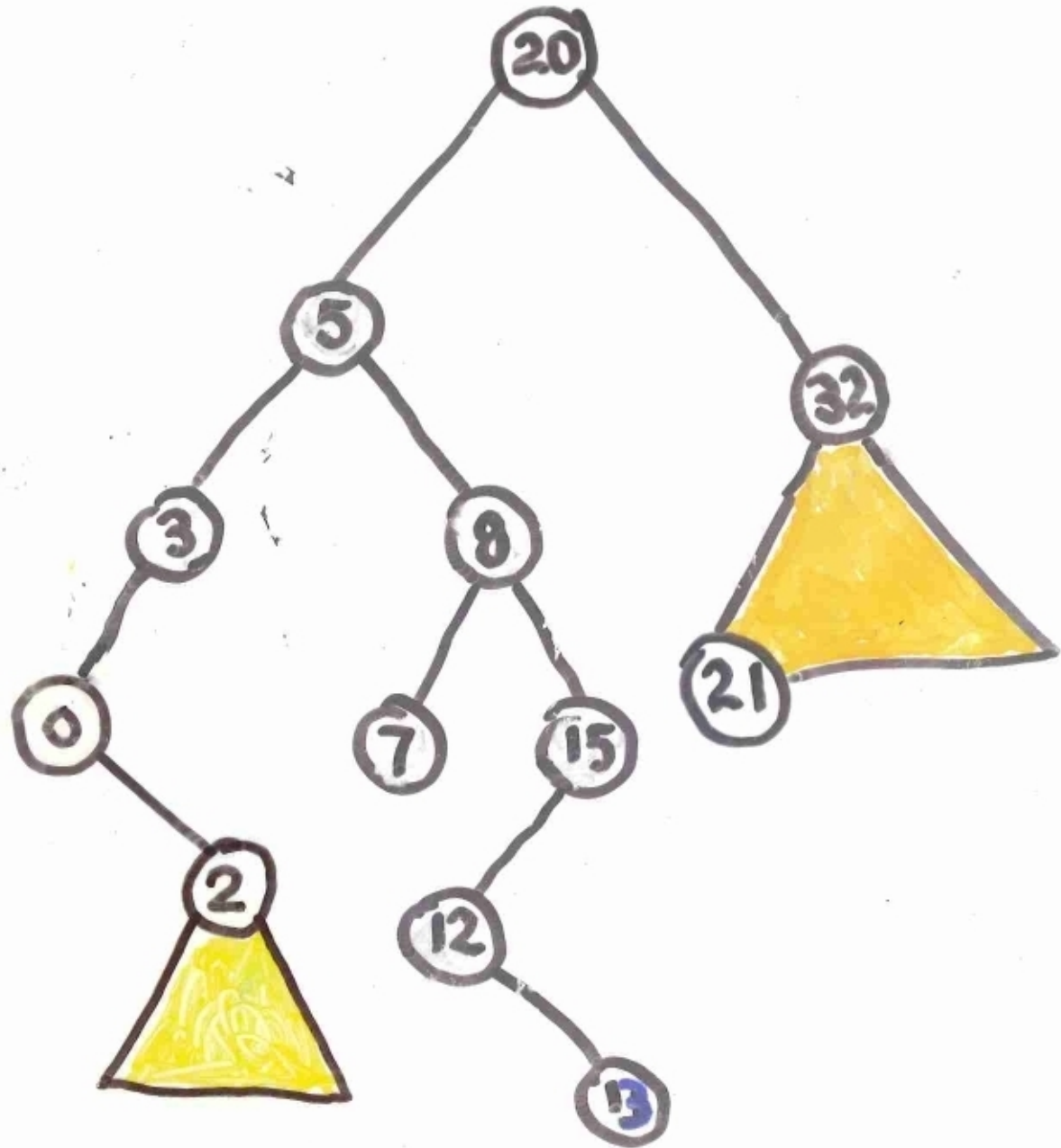
Delete(13)



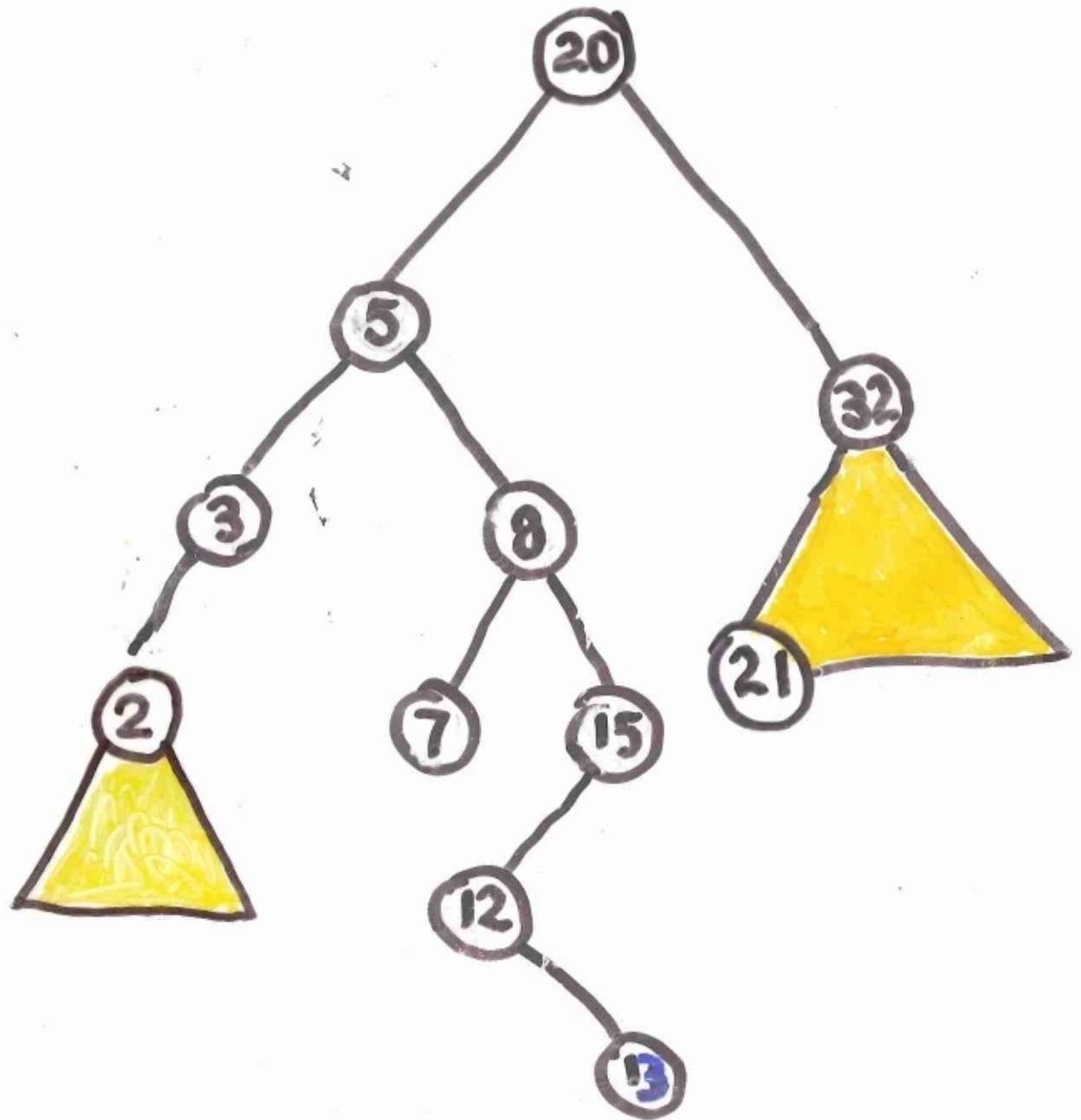
Delete(13)



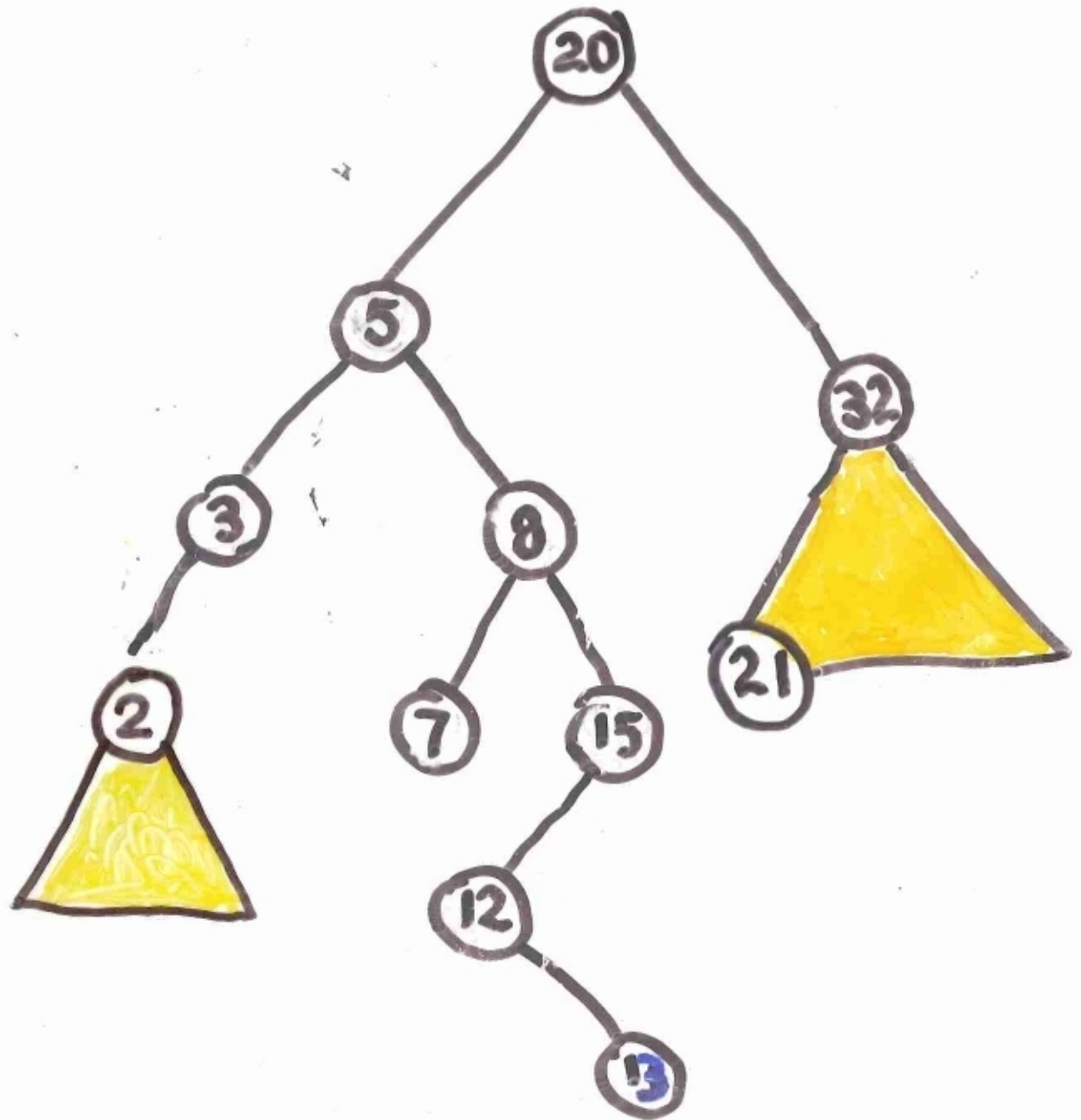
Delete(0)



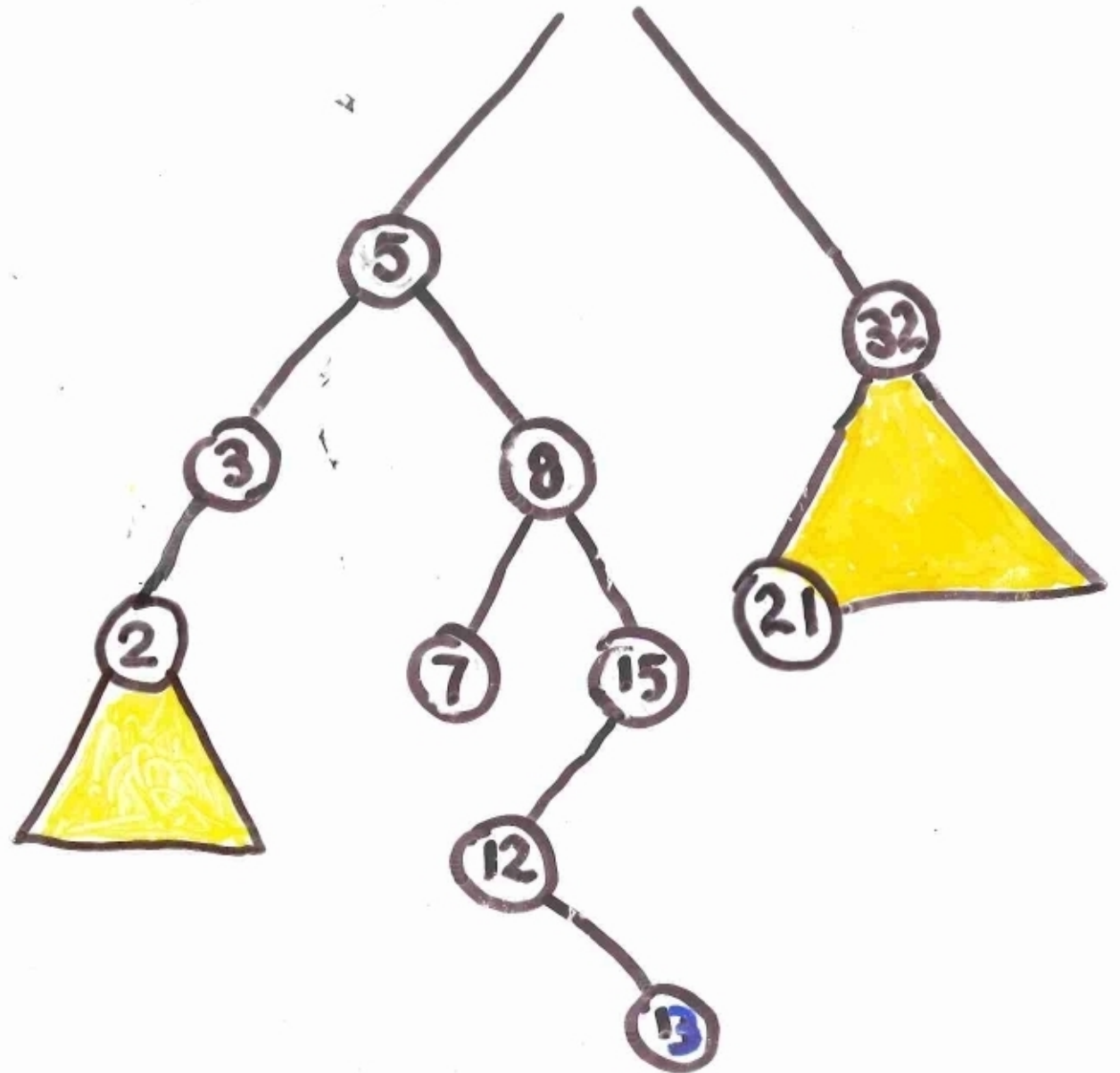
Delete(0)



Delete(20)

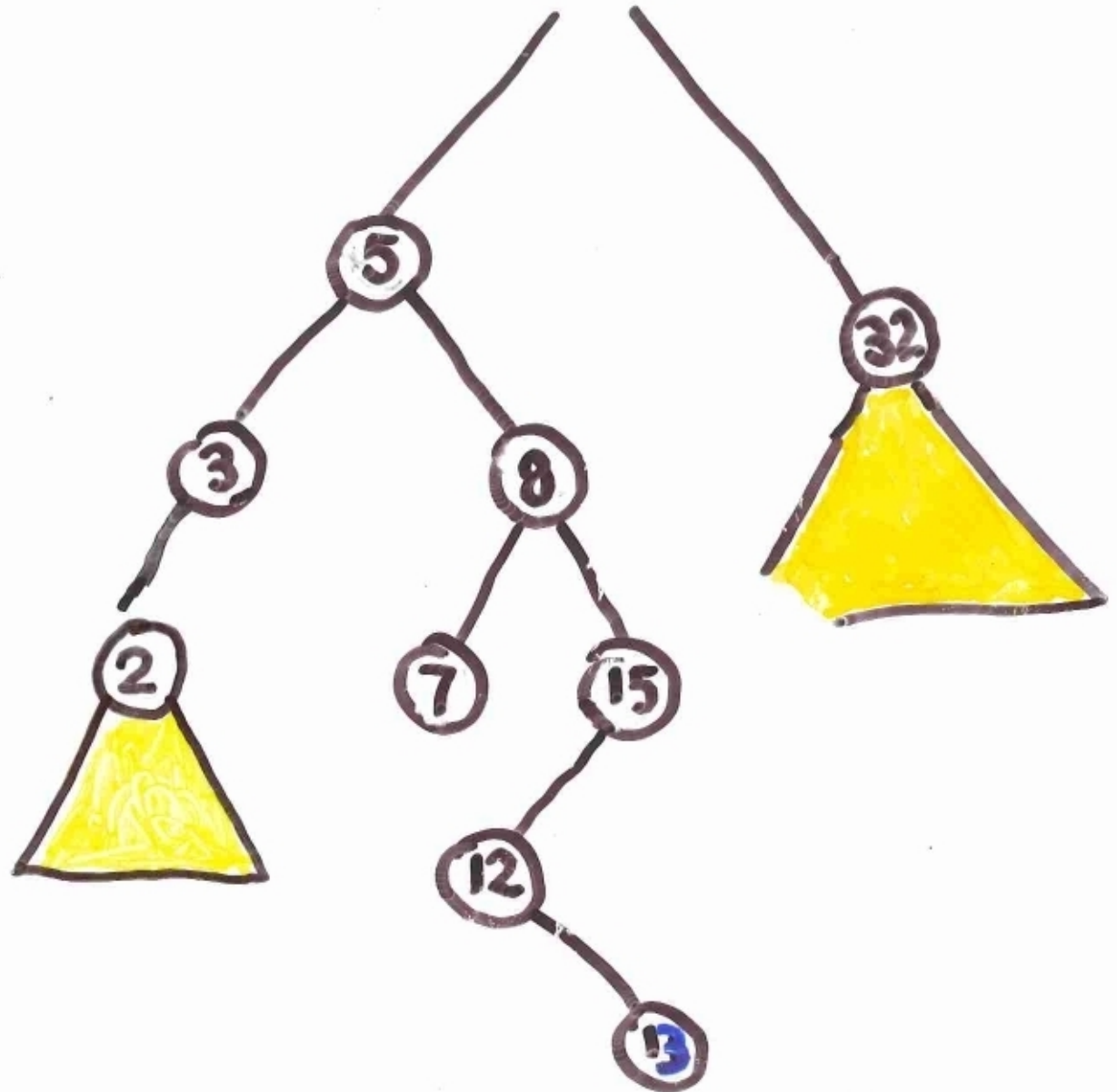


Delete(20)

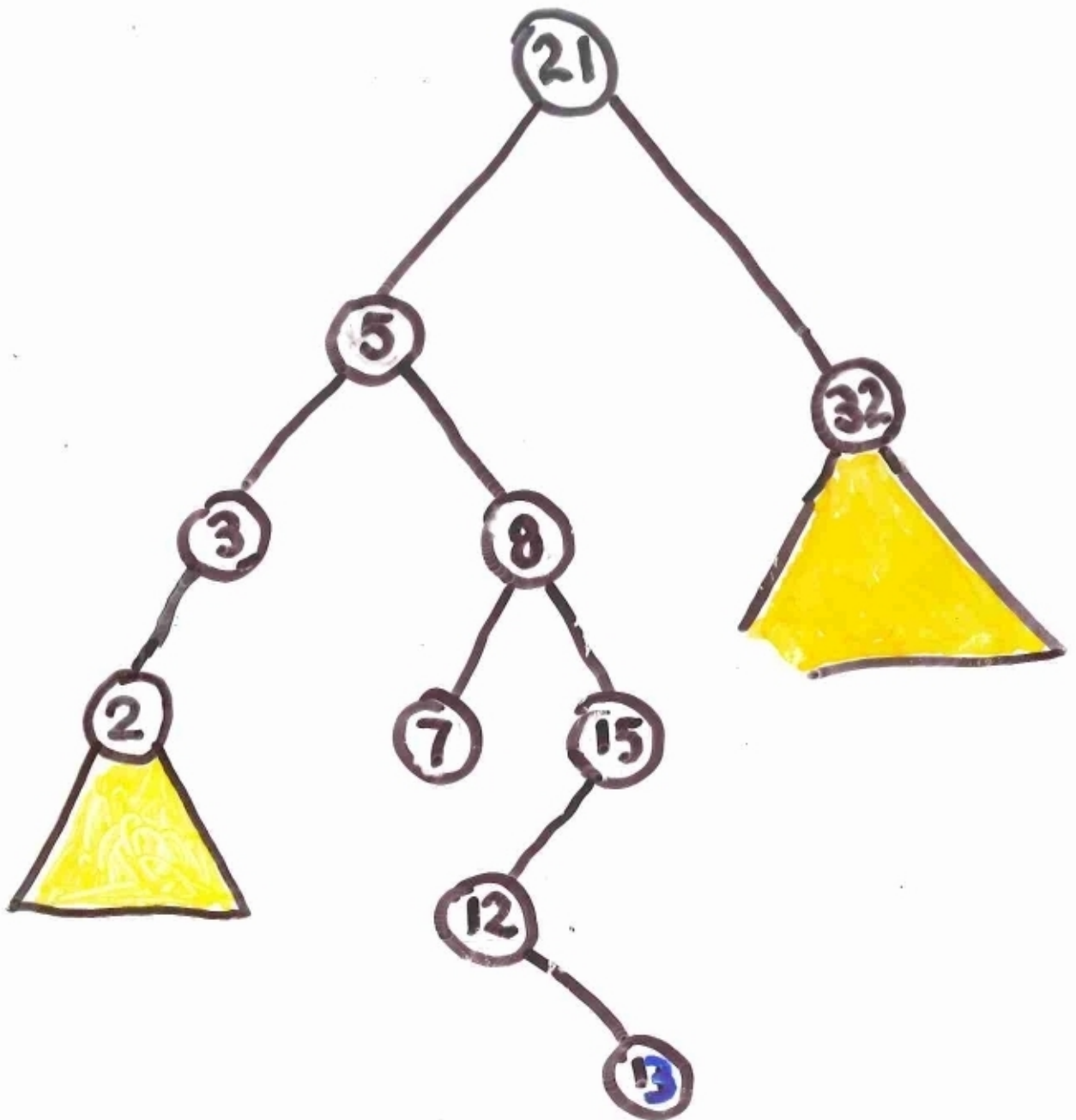


Delete(20)

21



Delete(20)



BS Tree

FACT 1

T is a binary search tree if, and only if,

- T is empty, or**
- The left and the right subtree of T are both binary search trees, and no node in the left subtree is larger than the root of T, and no node in the right subtree is smaller than the root of T.**

BS Tree

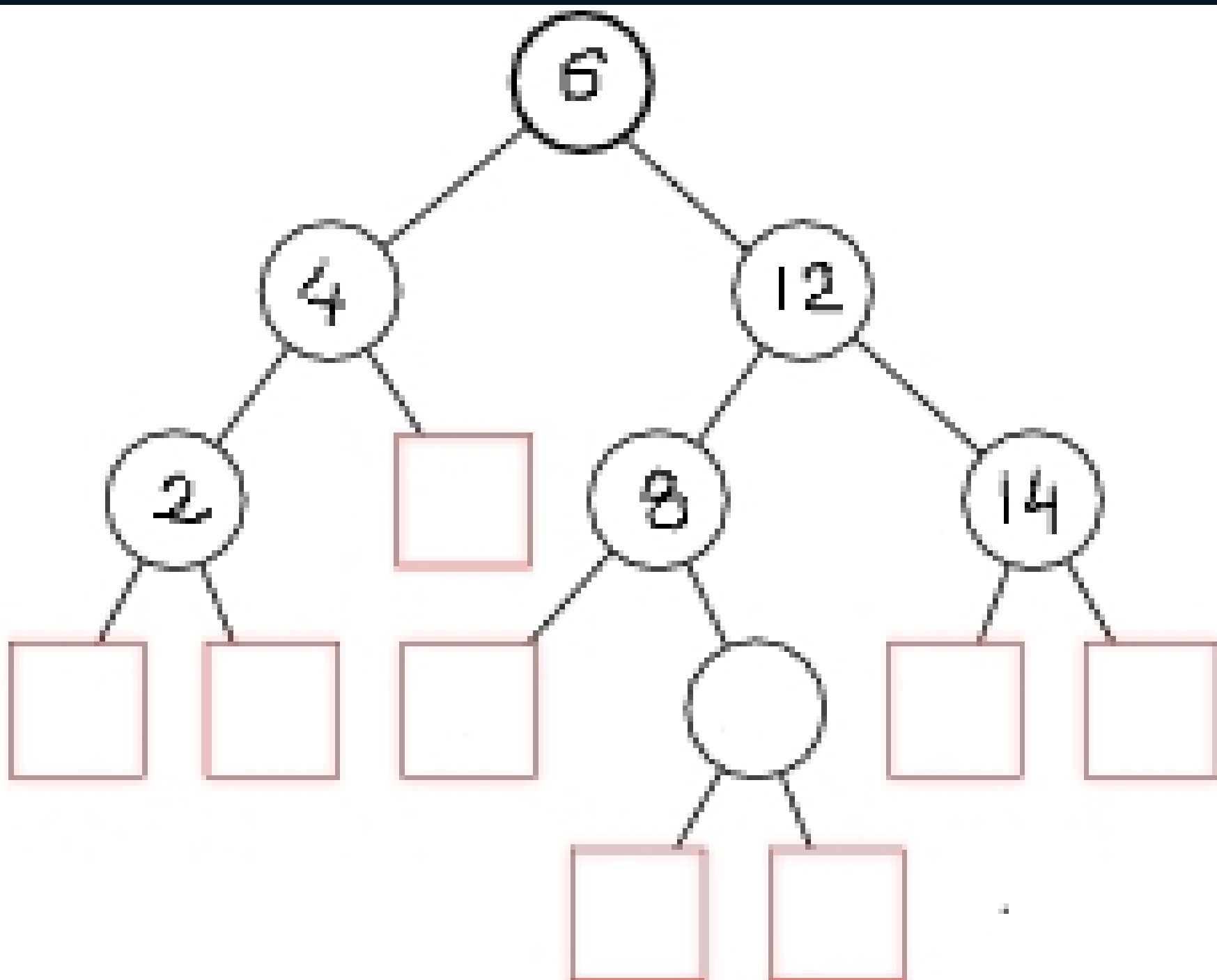
FACT 2

The number of comparisons while **successfully inserting** a new value x into a binary search tree is equal to the level

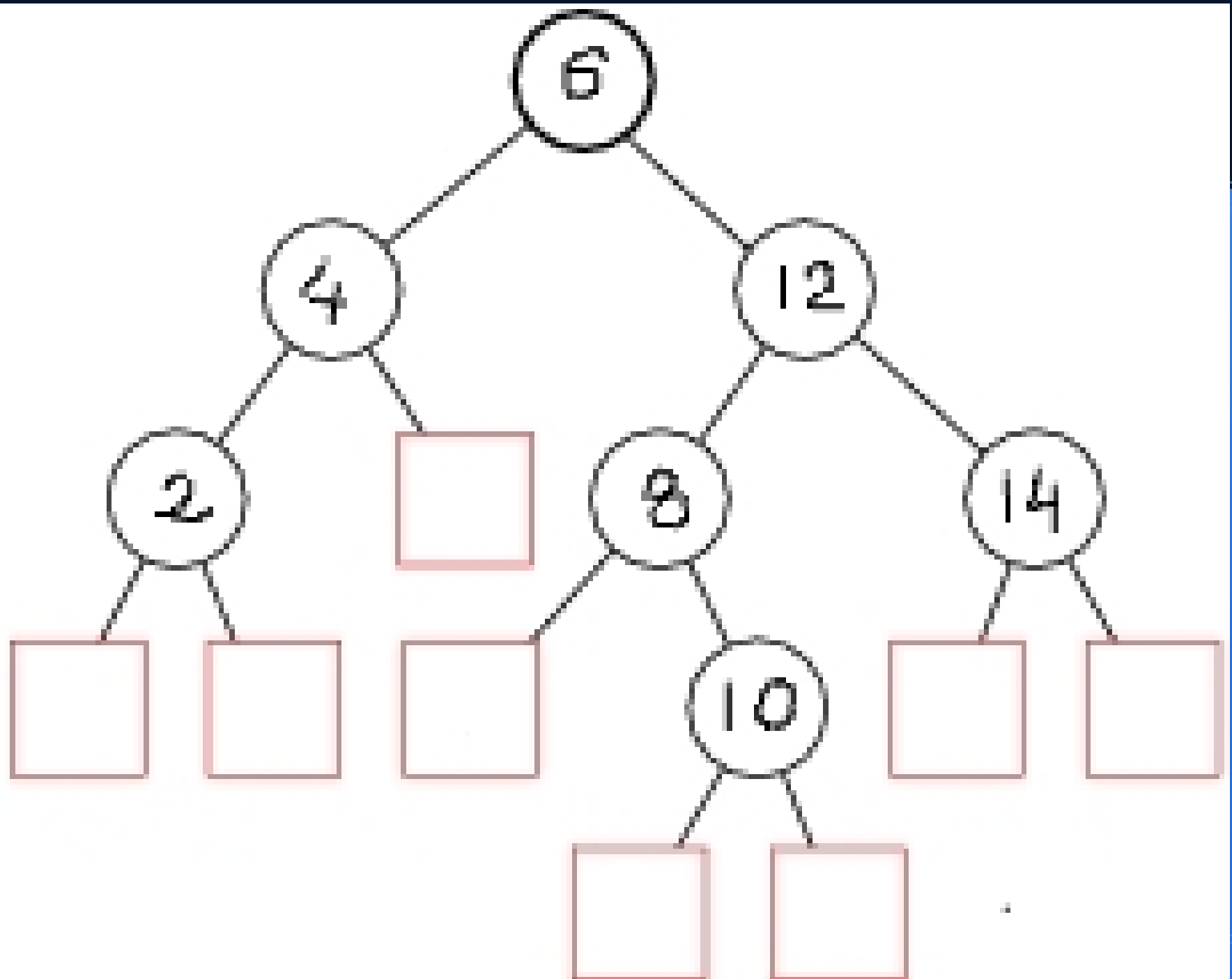
$\text{level}(x)$

of the newly inserted node with value x .

The
s
bin
of



Th
s
bi
of



BS Tree

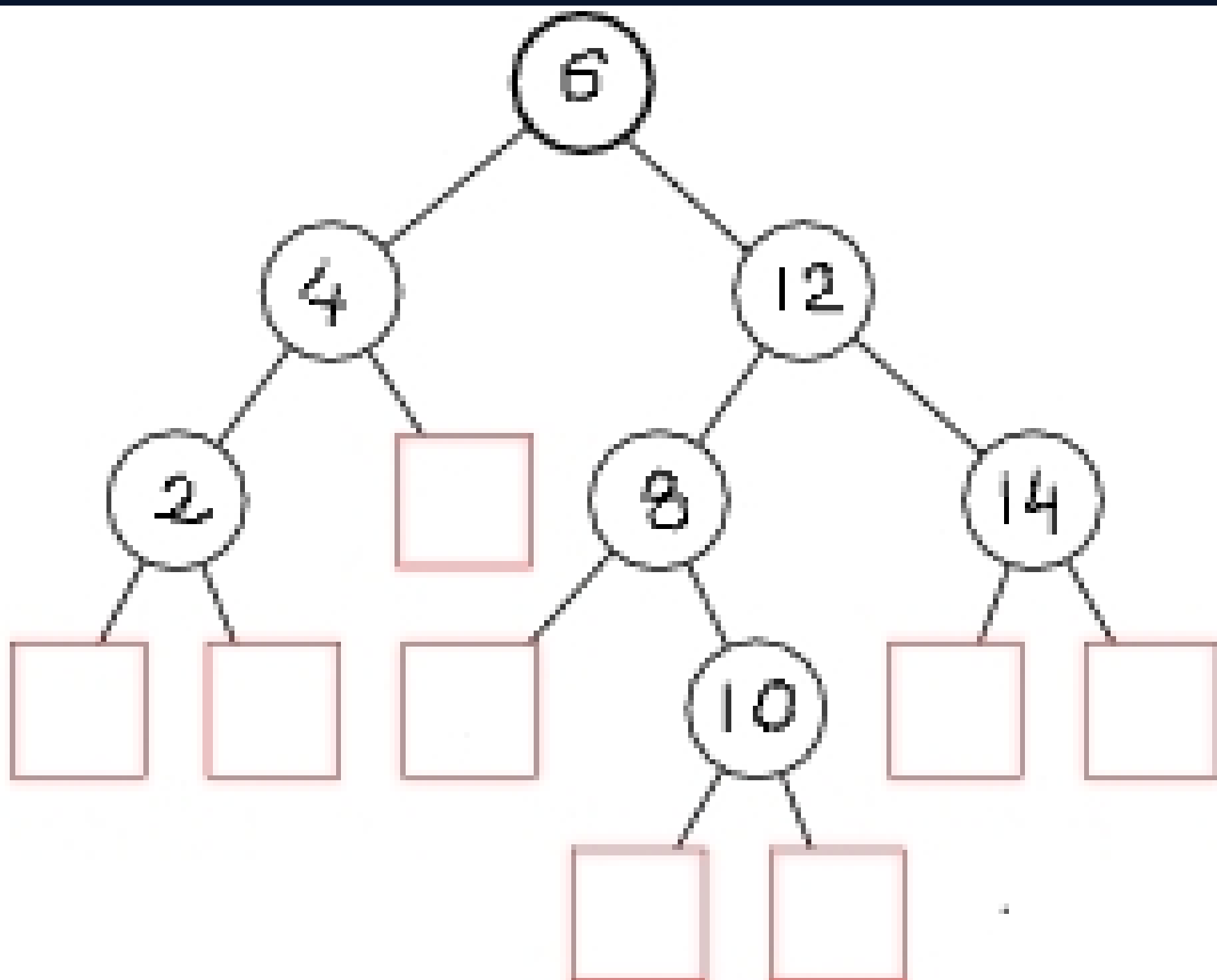
In particular, the number of comparisons while **building a binary search tree T** as a sequence of consecutive insertions is:

$$\text{level}(x_1) + \text{level}(x_2) + \dots + \text{level}(x_n),$$

where $\text{level}(x_i)$ is the level number to which x_i belongs.

In
v
as

wh
k

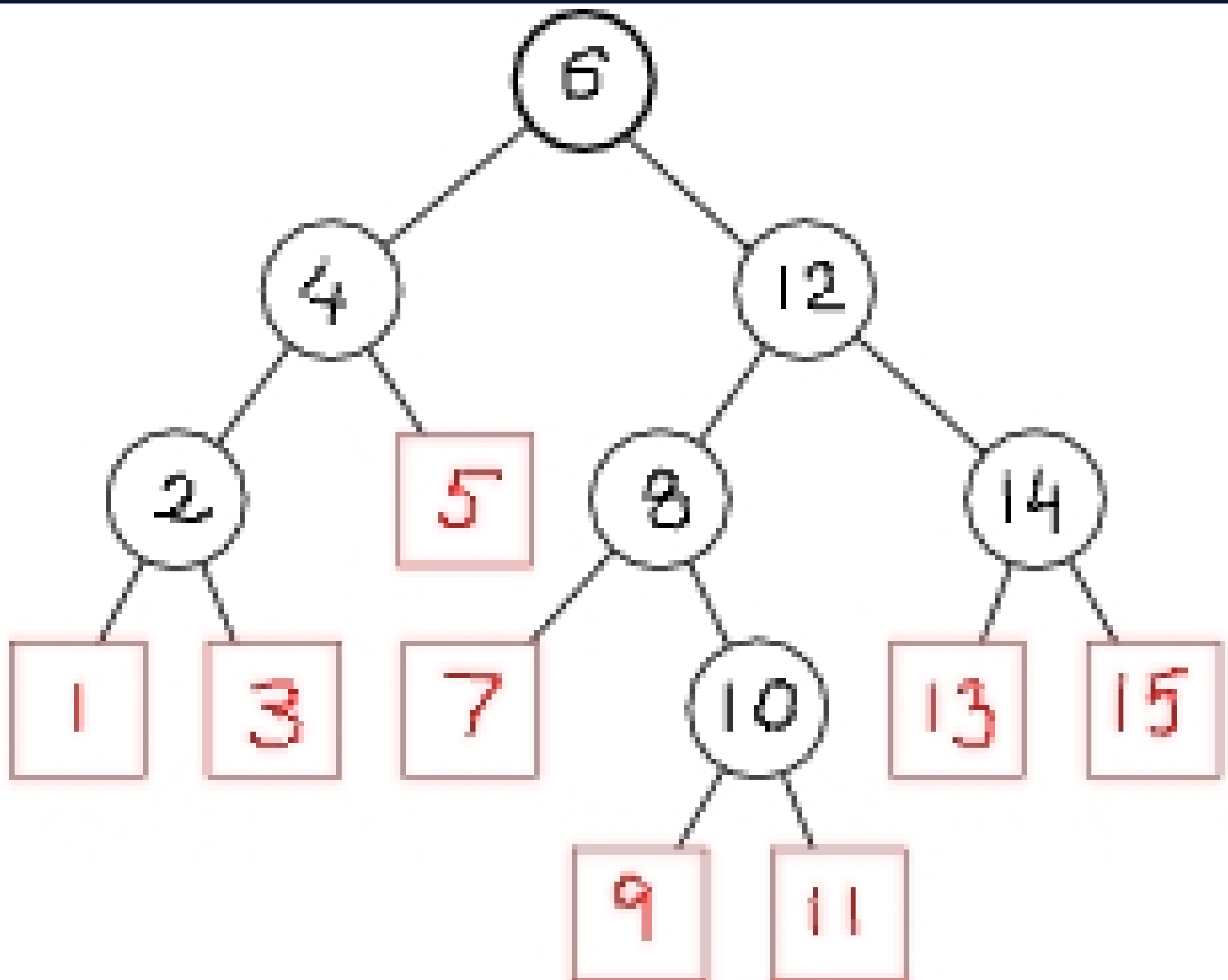


BS Tree

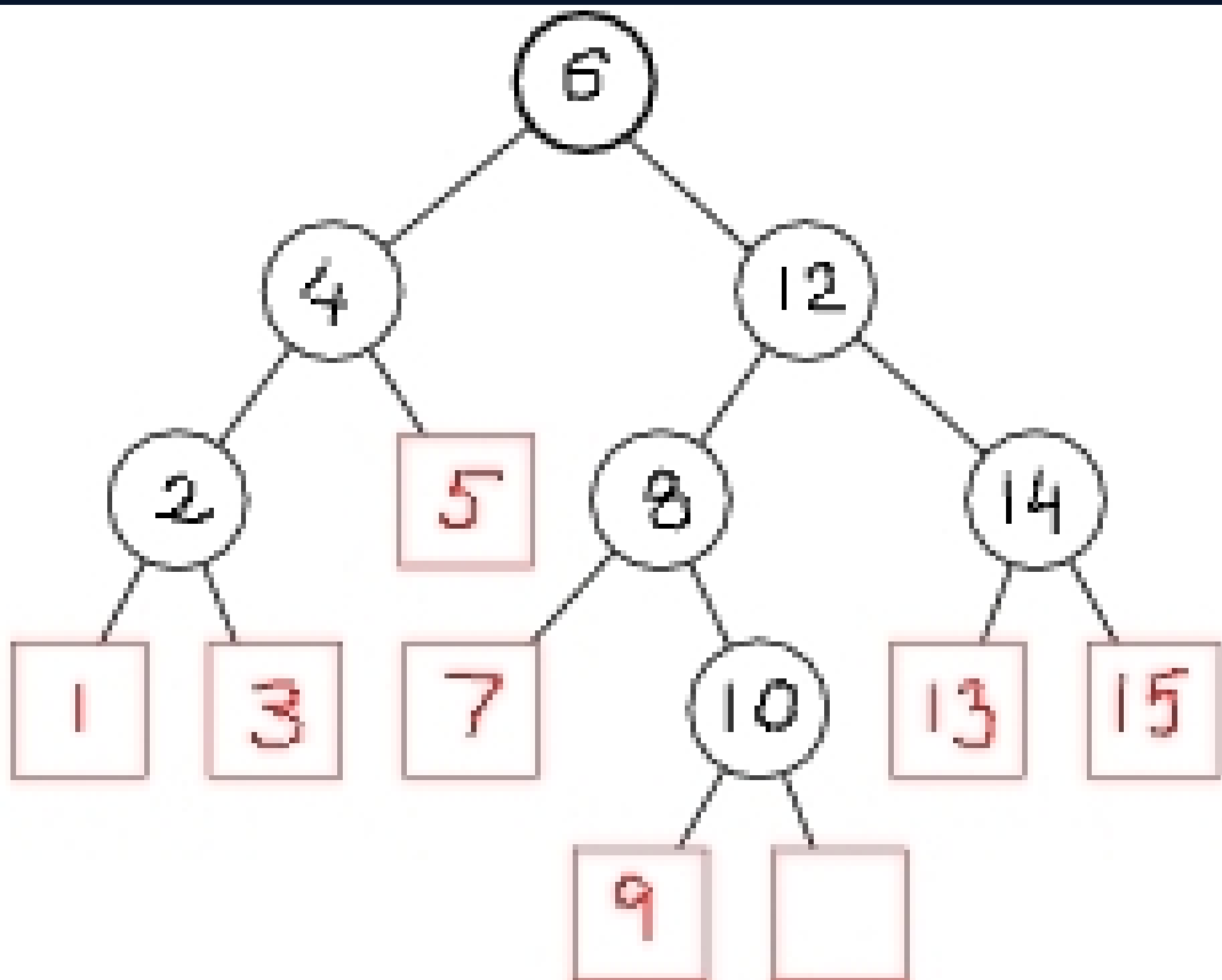
FACT 3

The number of comparisons while **unsuccessfully searching** for a value x in a binary search tree T is equal to the number of comparisons while successfully inserting x into T .

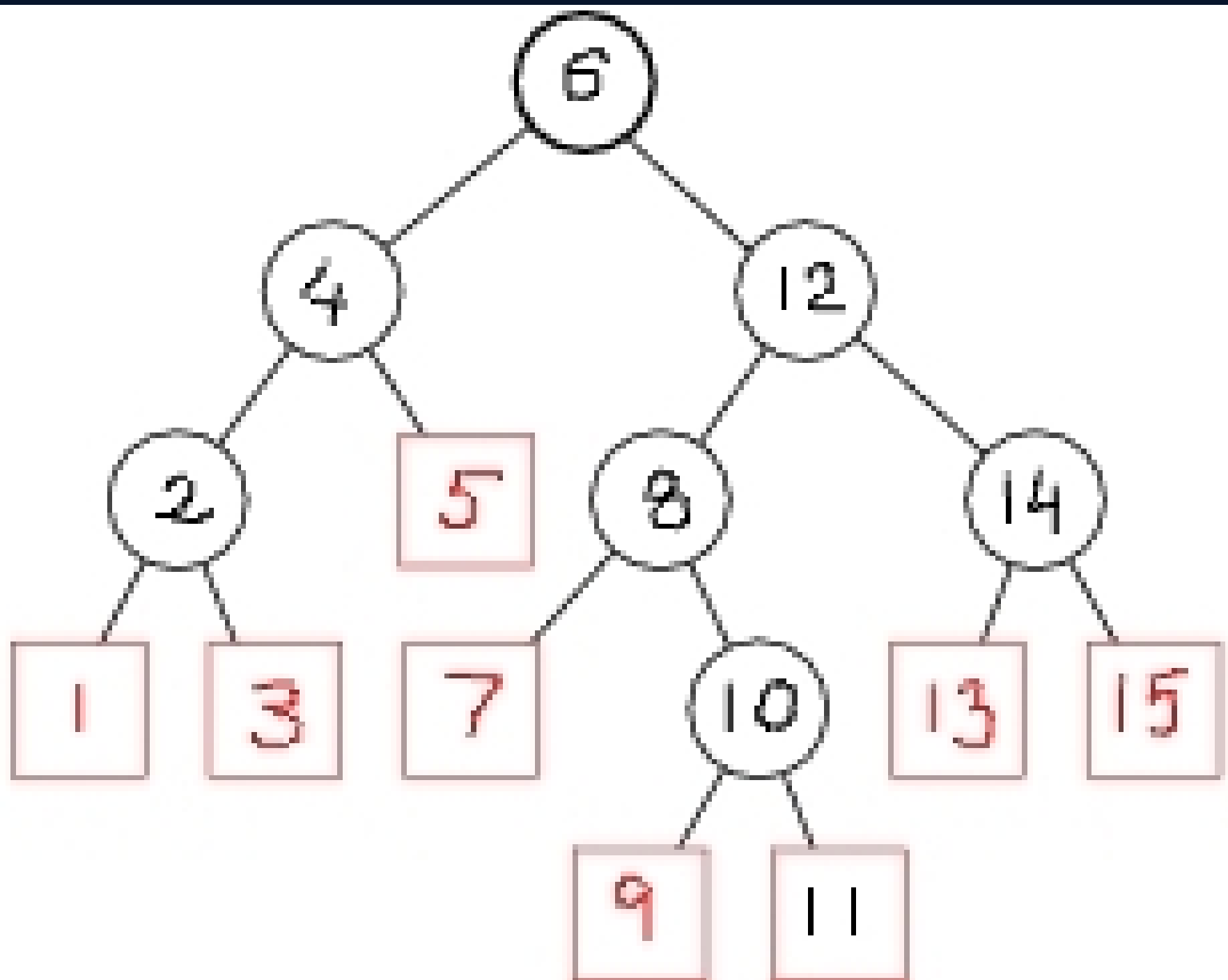
Th
bi
c
in



Th
bi
c
in



Th
bi
c
in



BS Tree

FACT 4

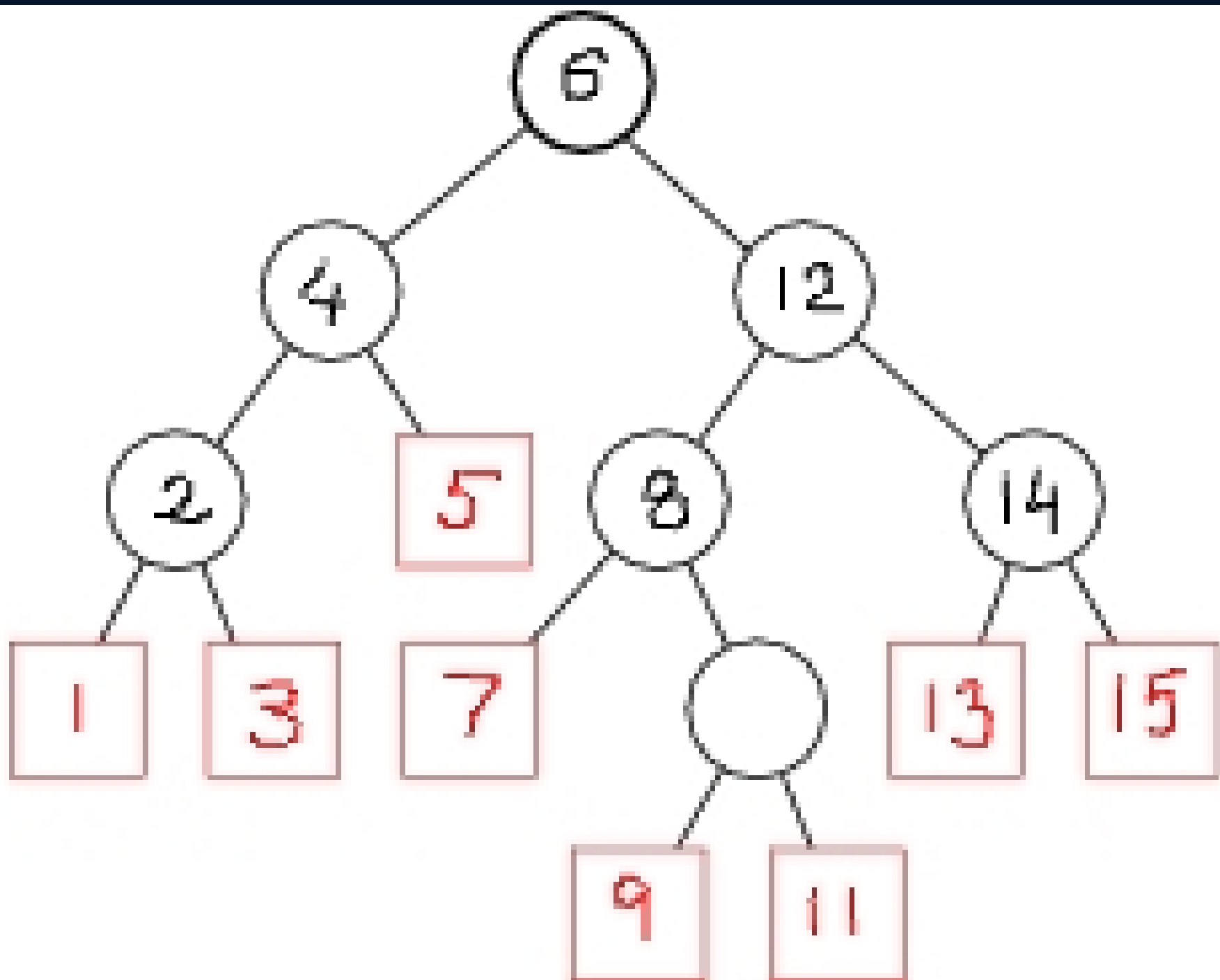
The number of comparisons while **successfully searching** for value x in a binary search tree is equal to 1 plus the number of comparisons made while inserting x into T ,

$$1 + \text{level}(x)$$

that is, one plus the level of the first node that contains that value.

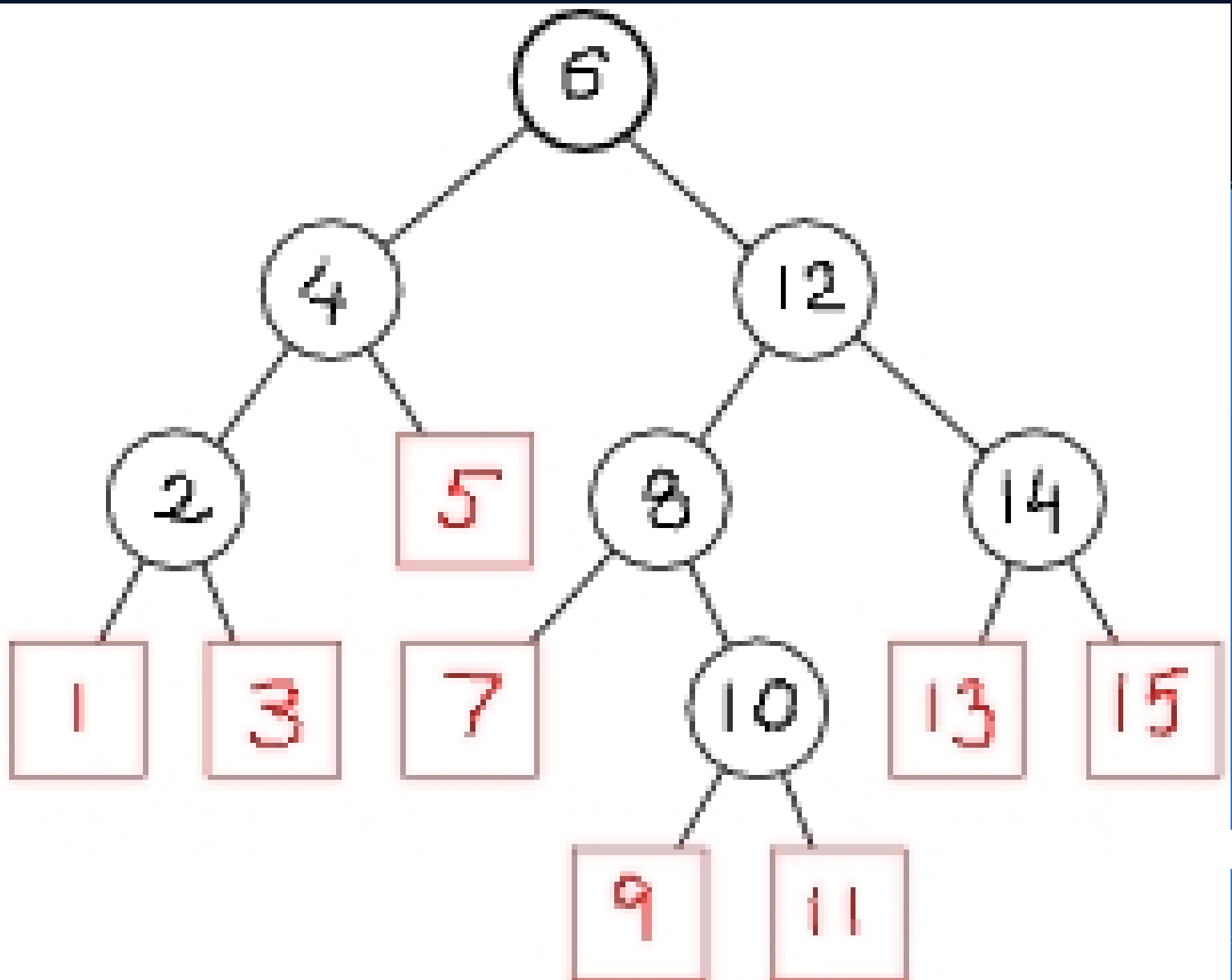
Th
c

th
c



Th
c

th
c

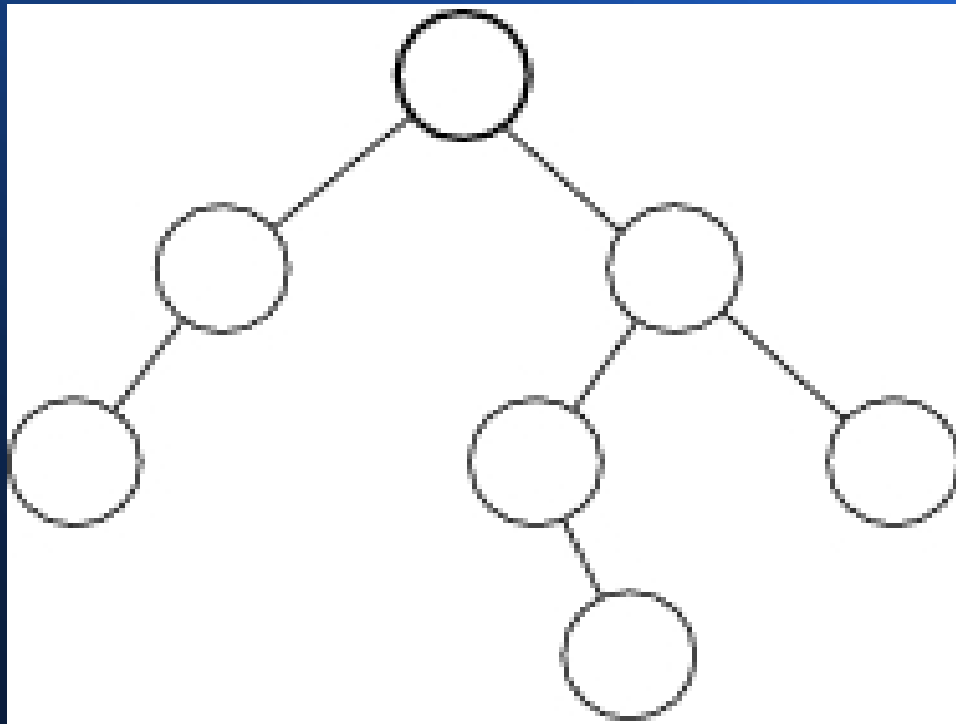


Performance of BS Trees

Definition of internal path

Internal path length I_T in a tree T is the sum of lengths of all paths from the root of T to non-leaves of T .

Performance of BS Trees

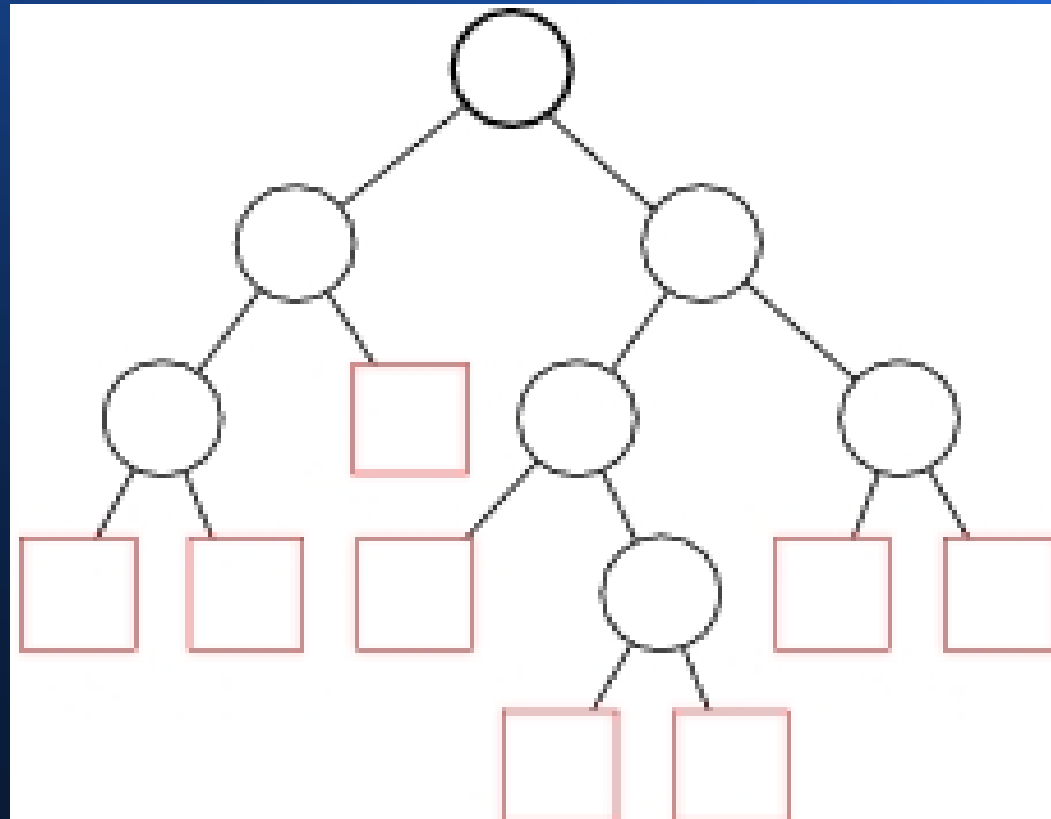


Performance of BS Trees

Definition of the external path

The external path length E_T in tree T is the sum of lengths of all paths from the root to the leaves of T .

Performance of BS Trees



Perf

es

2012

E
I

External path length and
internal path length

$$E = I + 2n$$

Search

I - the total number of
comparisons needed
to build a binary search
tree whose internal
path length is I.

Search

- the total number of
comparisons to run the
binary search sort
via such a tree
(same as for Quicksort)

Dr.

Copyright

Avg. number of comparisons
to search successfully
for a key in a B.S.
tree whose internal
path length is I:

$$C = \frac{I + n}{n} = \frac{I}{n} + 1$$

Perf

es

Dr. Manish A. Sachdev 202

© Copyright

Avg. number of comparisons
to search unsuccessfully
for a key in a B.S.
tree whose external
path length is E

$$c' = \frac{E}{n+1} = \frac{I+2n}{n+1} \approx \frac{I+2n}{n} =$$
$$= \frac{I}{n} + 2 = c+1$$

Best case

$$I \approx n \lg n - 2n$$

$$E = I + 2n \approx n \lg n$$

$$c = \frac{I}{n} + 1 \approx \lg n - 1$$

$$c' \approx c + 1 = \lg n$$

Per

ees

Copyright © Dr. Murely A. Sudhakar 2012

Avg. case

$$T \approx 1.4 n \lg n - 2.8 n$$

$$E \approx 1.4 n \lg n - 0.8 n$$

$$C \approx 1.4 \lg n - 1.8$$

$$C' \approx 1.4 \lg n - 0.8$$

Worst case

$$T = \frac{n(n-1)}{2}$$

$$E = \frac{n(n+3)}{2}$$

$$C = \frac{n+1}{2}$$

$$C' = \frac{n+3}{2}$$

- 1 -

Notes to Standish p. 278

© Copyright by Dr. Muel A. Enchens 2012

Minimum internal path length I_n of any binary tree with n nodes is:

$$I_n = \sum_{i=1}^n \lfloor \lg i \rfloor =$$

$$= (n+1) \lfloor \lg n \rfloor - \underbrace{2^{\lfloor \lg n \rfloor + 1}}_{n < y \leq 2n} + 2$$

Minimum external path length E_n of any binary tree with n nodes is:

$$E_n = I_n + 2n =$$

$$= (n+1) \lfloor \lg n \rfloor - 2^{\lfloor \lg n \rfloor + 1} + 2 + 2n =$$

$$= (n+1) \lfloor \lg n \rfloor + 2 \underbrace{(n - 2^{\lfloor \lg n \rfloor})}_{0 \leq x < \frac{n}{2}} + 2$$

Perf

ees

Notes to Standish p 218

2012

So,

So,

Also,

Also,

Also,

Also,

Also,

Also,

$$(n+1)\lfloor \lg n \rfloor + 2 \leq E_n <$$

$$< (n+1)\lfloor \lg n \rfloor + n + 2$$

$$(n+1)\lfloor \lg n \rfloor - 2n + 2 \leq I_n <$$

$$< (n+1)\lfloor \lg n \rfloor - n + 2$$

$$C_n = \frac{I_n + n}{n} = \frac{I_n}{n} + 1$$

$$\frac{n+1}{n}\lfloor \lg n \rfloor - 2 + \frac{2}{n} + 1 \leq C_n <$$

$$< \frac{n+1}{n}\lfloor \lg n \rfloor - 1 + \frac{2}{n} + 1$$

$$\lfloor \lg n \rfloor + \frac{1}{n}(\lfloor \lg n \rfloor + 2) - 1 \leq C_n <$$

$$< \lfloor \lg n \rfloor + \frac{1}{n}(\lfloor \lg n \rfloor + 2)$$

$$\text{Best case } \lfloor \lg n \rfloor - 1 \leq C_n \leq \lfloor \lg n \rfloor$$

Perf

ees

- 3 -

Notes to Standish p 278

Dr. Harold A. Standish
2012

$$C_n' = \frac{E_n}{n+1}$$

$$\lfloor \lg n \rfloor + \frac{2}{n+1} \leq C_n' <$$

$$< \lfloor \lg n \rfloor + \frac{n}{n+1} + \frac{2}{n+1} =$$

$$= \lfloor \lg n \rfloor + \frac{(n+1)+1}{n+1} = \lfloor \lg n \rfloor + 1 +$$

$$+ \frac{1}{n+1}$$

best case

$$\lfloor \lg n \rfloor \leq C_n' \leq \lfloor \lg n \rfloor + 1$$

⑥