CSC 311

Lectures on
Data Structures

by

Dr. Marek A. Suchenek ©

Computer Science
CSUDH

Copyrighted material

CSC 311

Lecture 11
Trees

Definitions, Applications, Implementations, Analysis

8

Trees and Graphs

game trees

search trees

priority queues and heaps

binary trees

representing priority queues using heaps

binary tree traversals

binary search trees

AVL trees

2–3 trees

tries

Huffman codes

graphs are more general than trees

graph representations

flow graphs

graph searching algorithms

topological ordering

8.2 Trees—Basic Concepts and Terminology

LEARNING OBJECTIVES

1. To learn how to refer to various parts of trees.
2. To learn about some relationships that are always true in trees.

roots, children, and descendants

leaves

ancestors and parents

Introduction

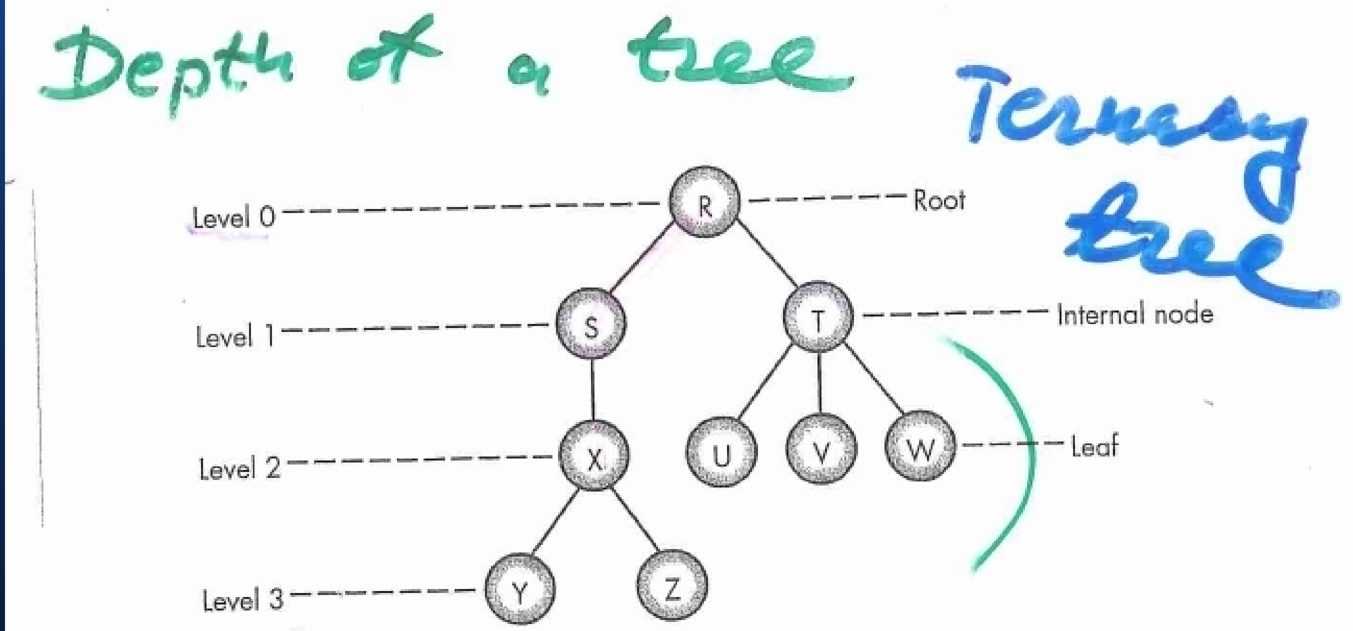


Figure 8.1 Basic Tree Anatomy

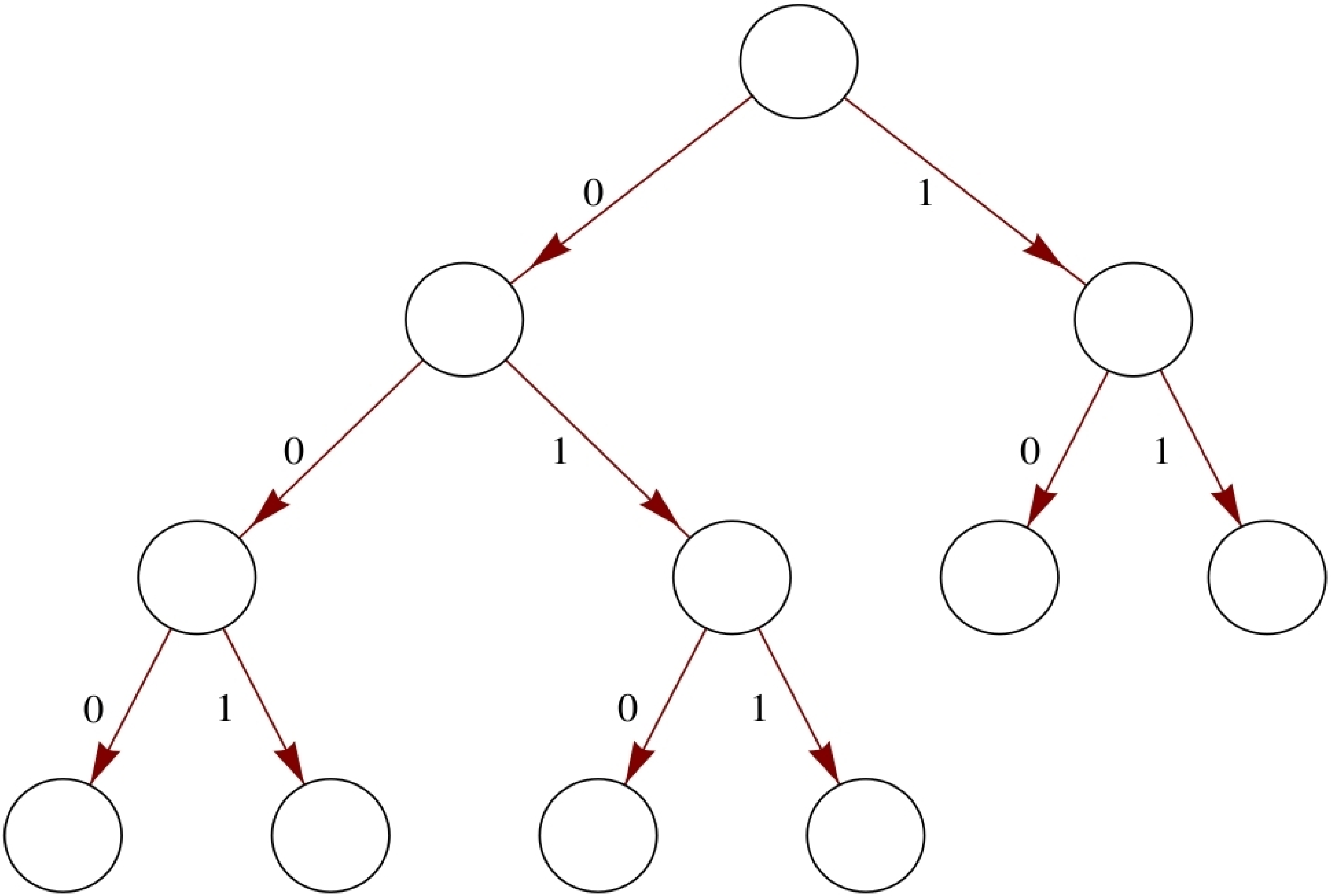
8.2 EXERCISES 1, 2, p 248

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 begining subsequence.

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

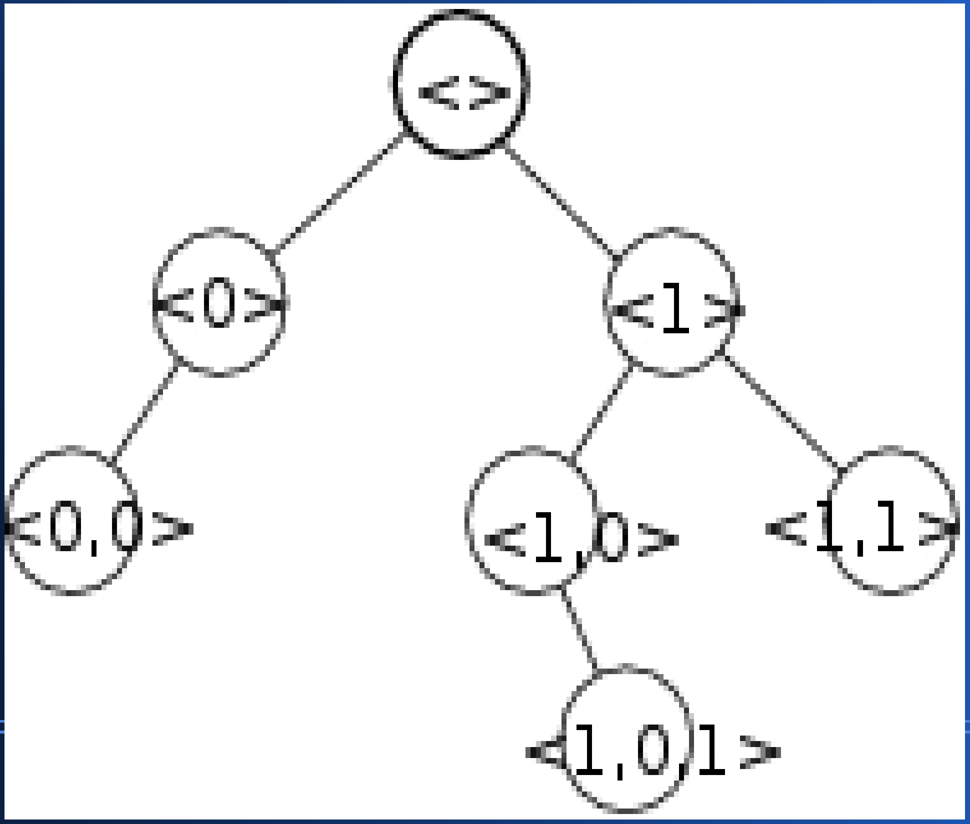
Definitions of tree

Example:

{<>, <0>, <1>, <0,0>,
<1,0>,<1,1>,<1,0,1>}

Definitions of tree

{<>, <0>, <1>, <0,0>, <1,0>,<1,1>,<1,0,1>}



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

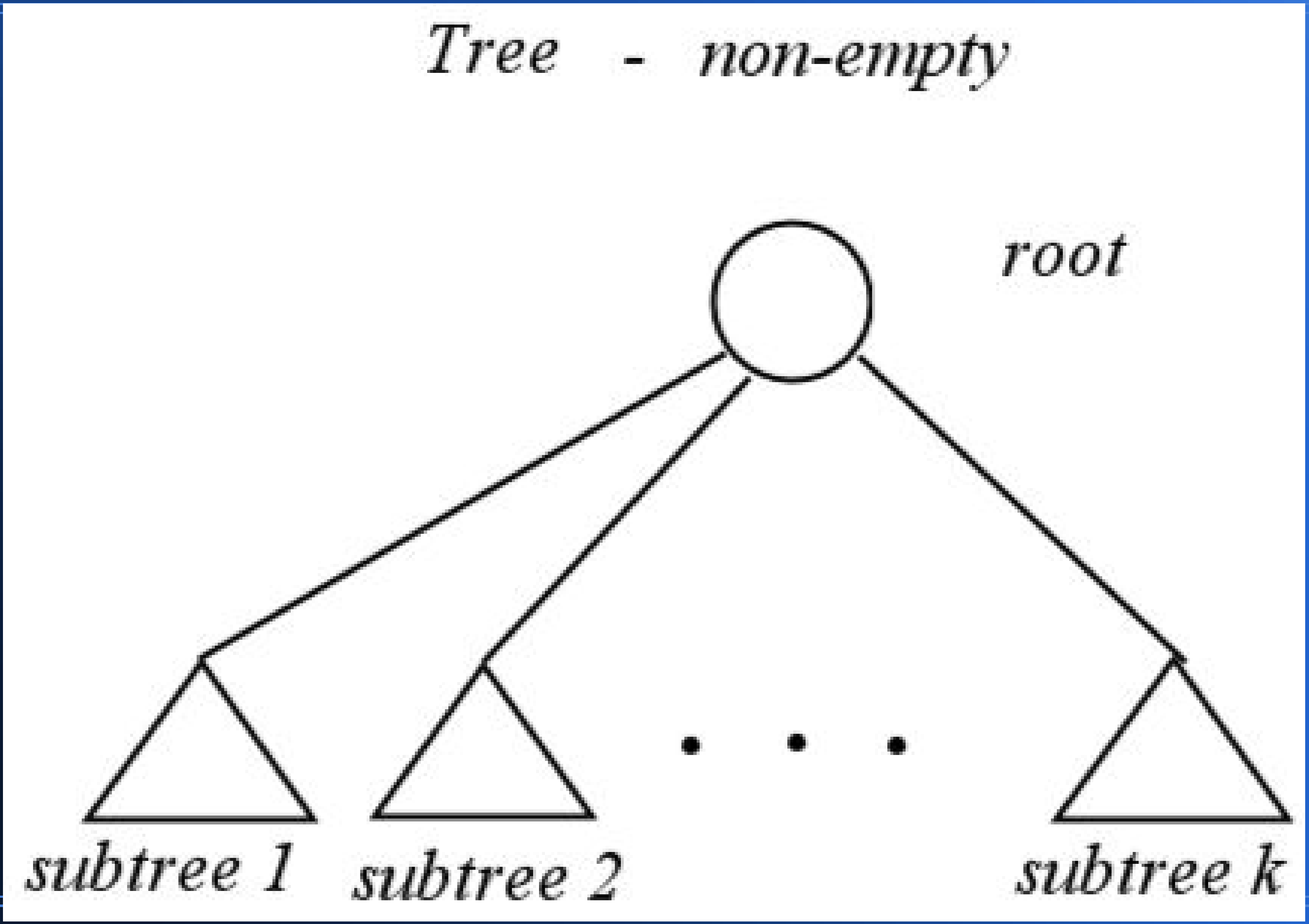
Definitions of tree

(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.)

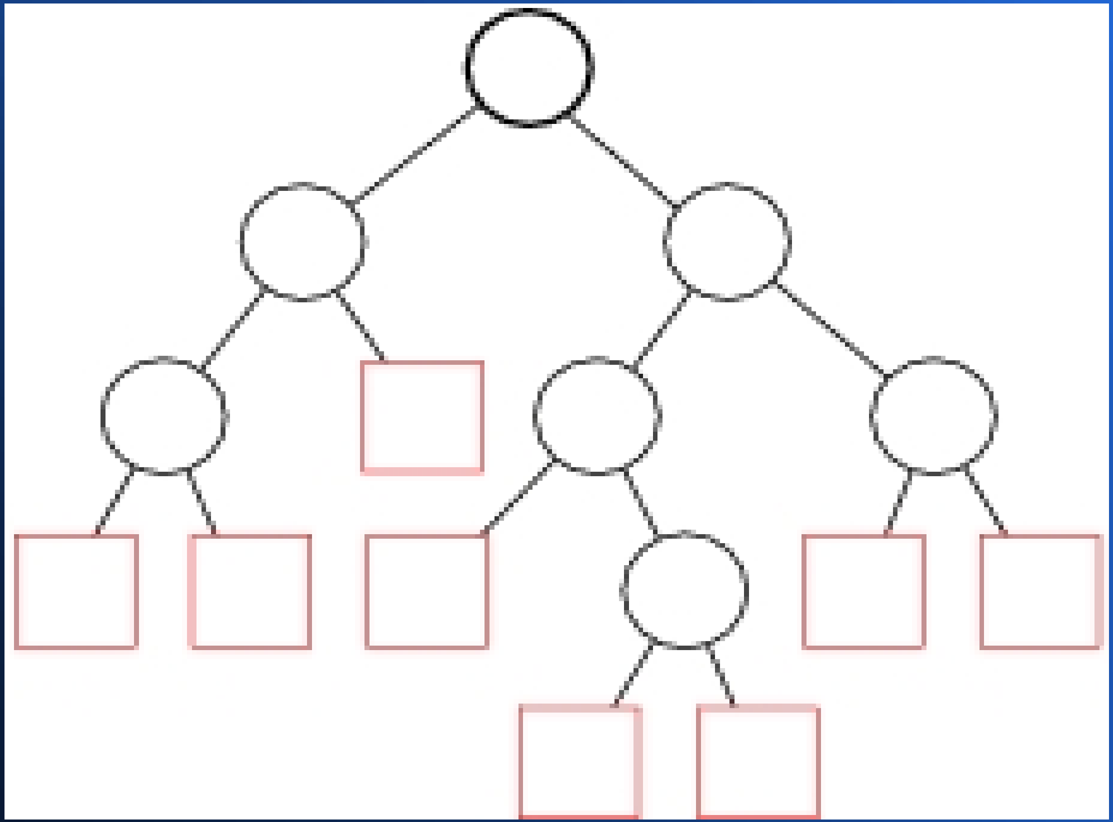
Definitions of tree

(iii) Nothing else is a finite tree.

Definitions of tree



Definitions of tree

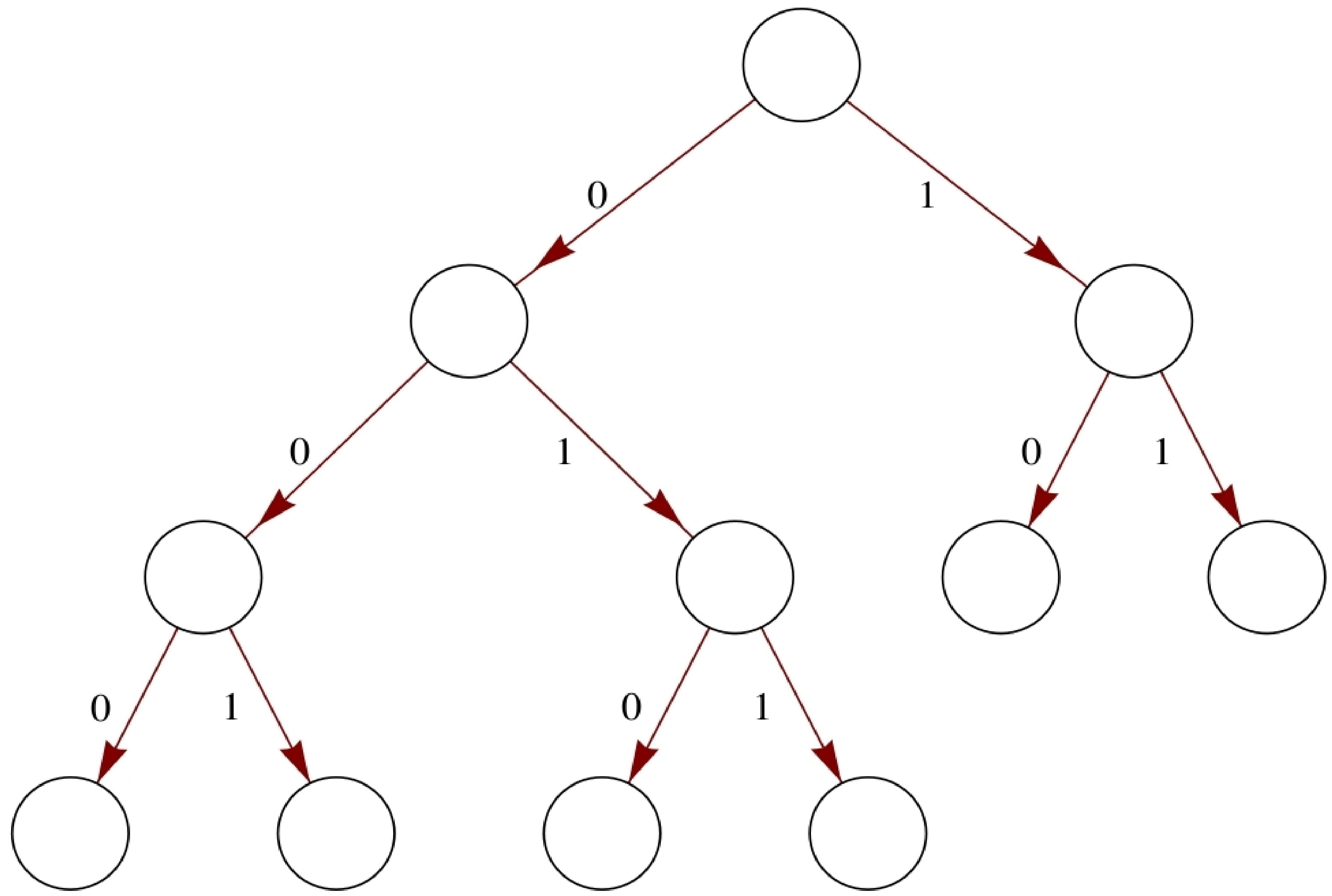


Boxes represent empty subtrees

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 begining subsequence.

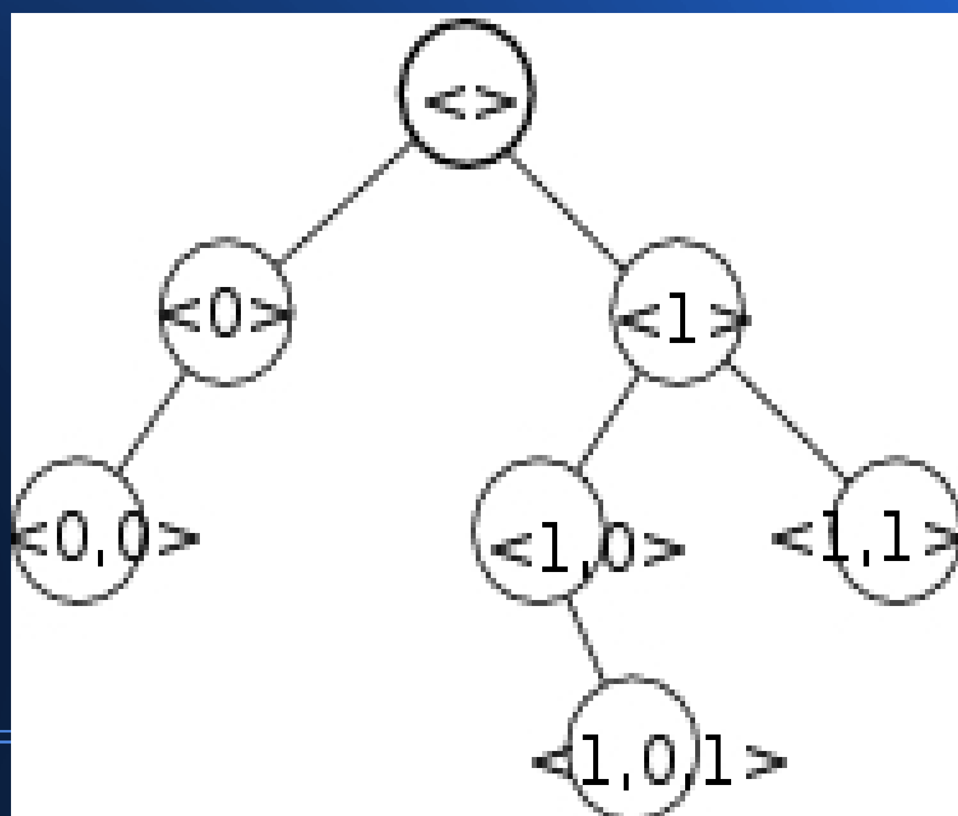
Definitions of binary tree

Example:

{<>, <0>, <1>, <0,0>, <1,0>,<1,1>,<1,0,1>}

Definitions of binary tree

{<>, <0>, <1>, <0,0>, <1,0>,<1,1>,<1,0,1>}



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

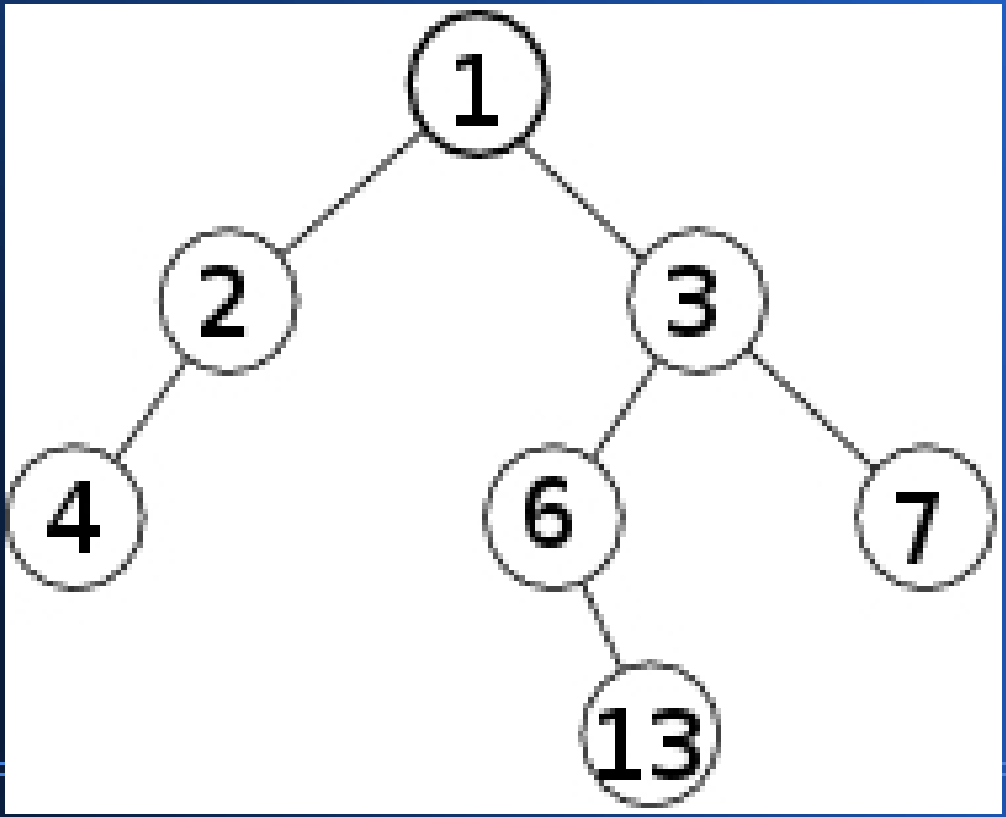
{1, 2, 3, 4, 6, 7,13}

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

Is: {<1>, <1,0>, <1,1>, <1,0,0>, <1,1,0>,<1,1,1>,<1,1,0,1>}

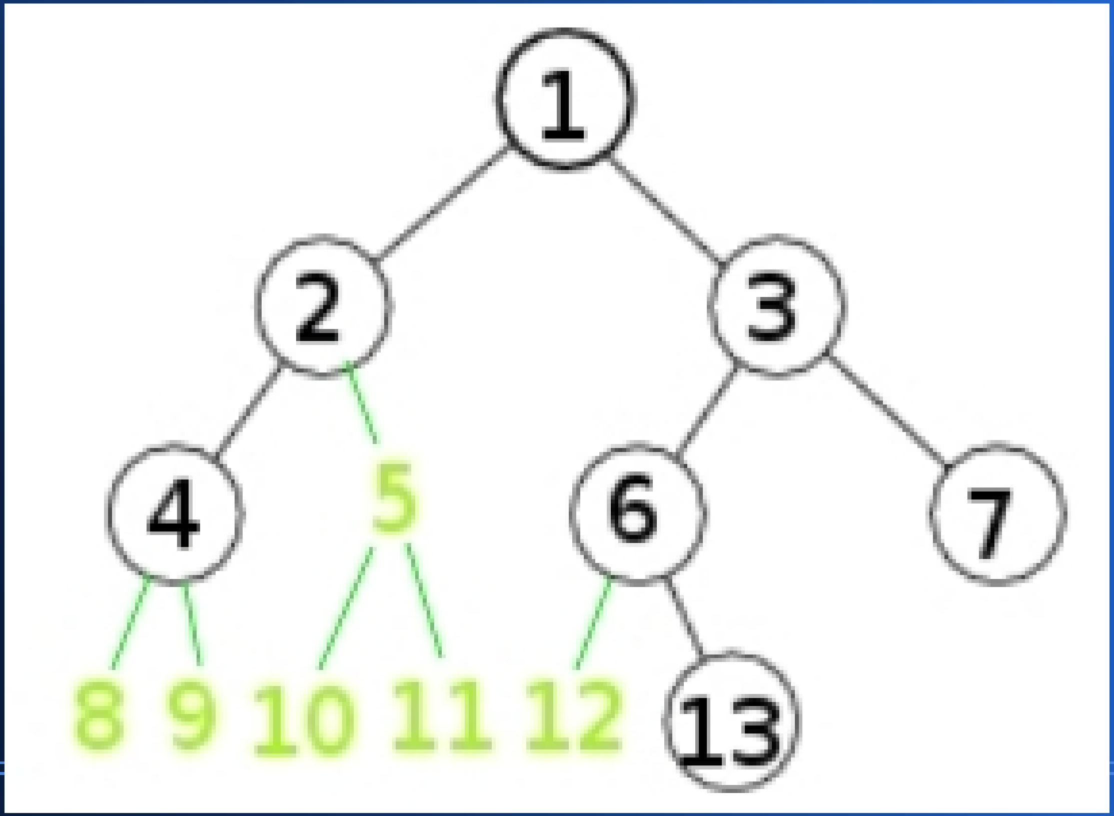
Definitions of binary tree

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



Definitions of binary tree

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



Def. of complete binary tree

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

Def. of complete binary tree

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

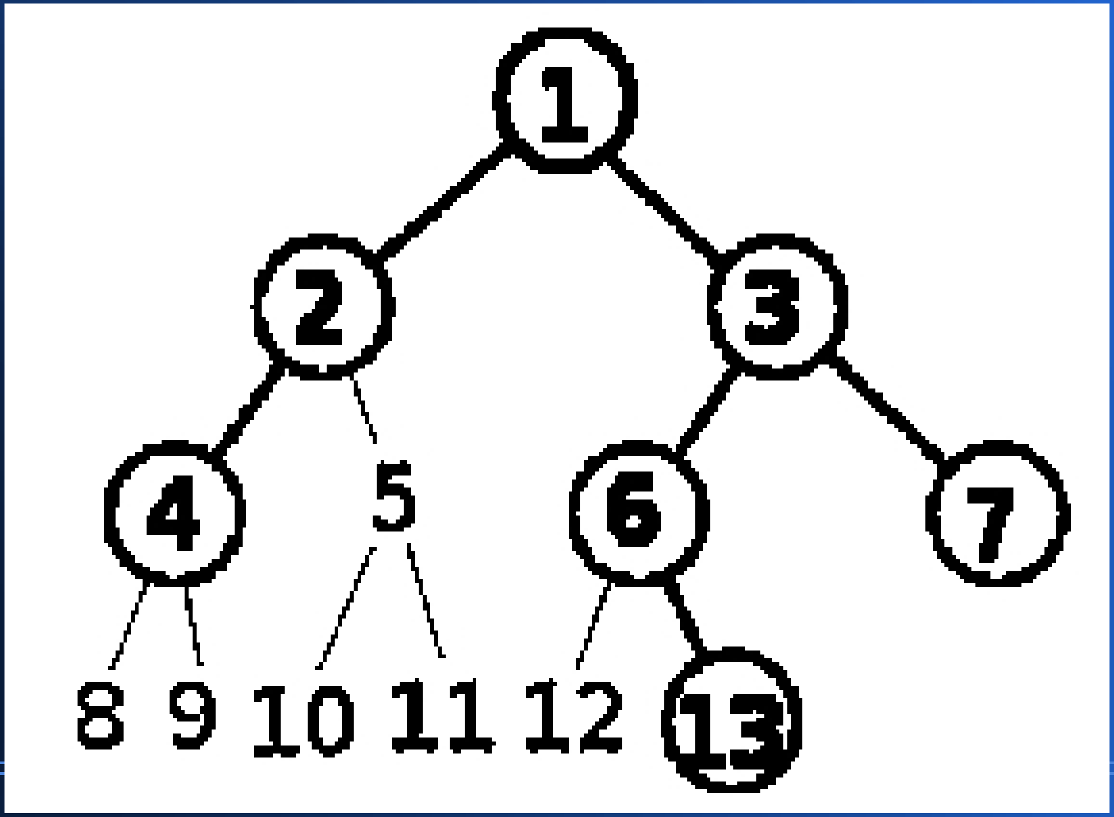
Of course, it closed under operation of positive integer division by 2.

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

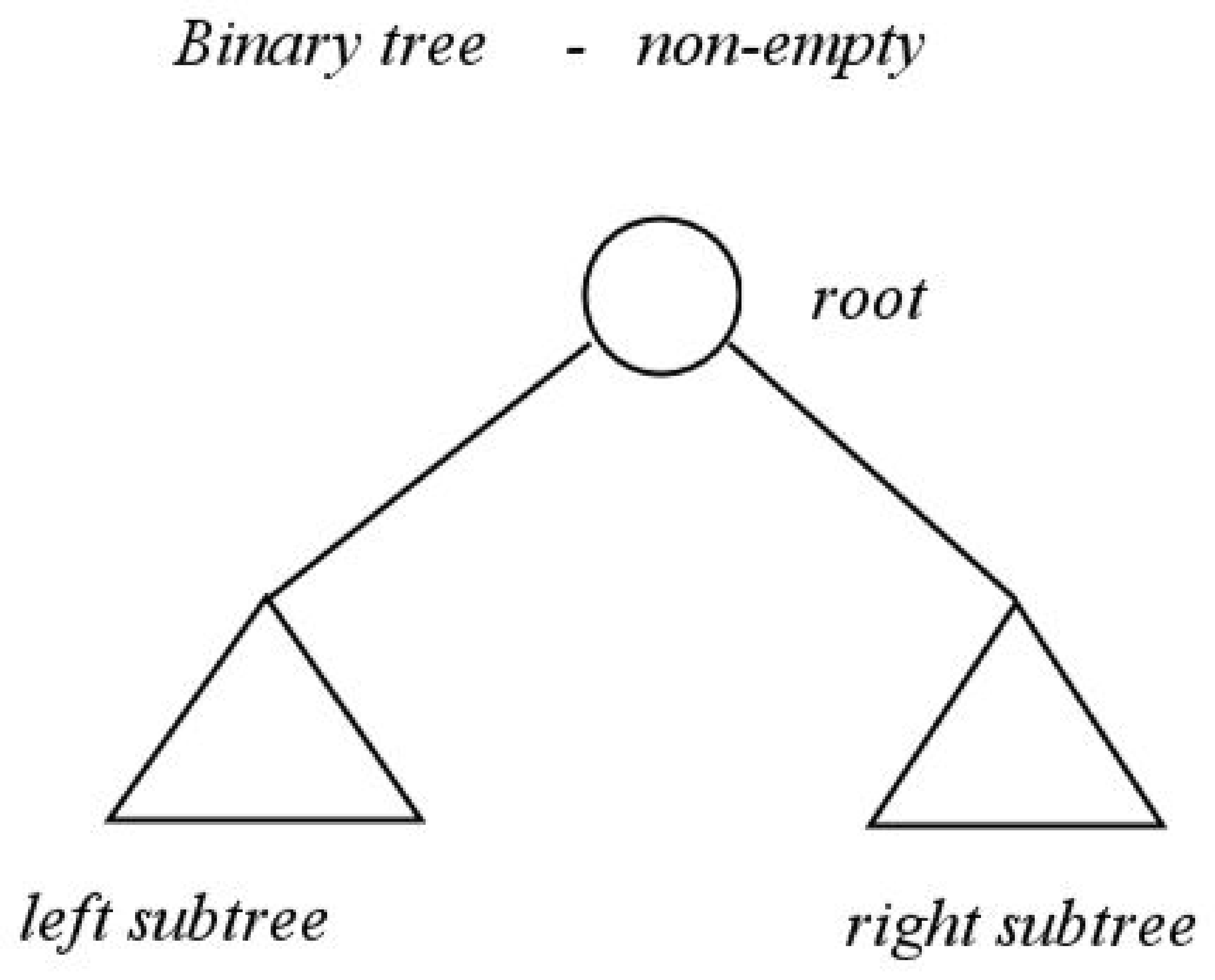
Definitions of binary tree

(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.)

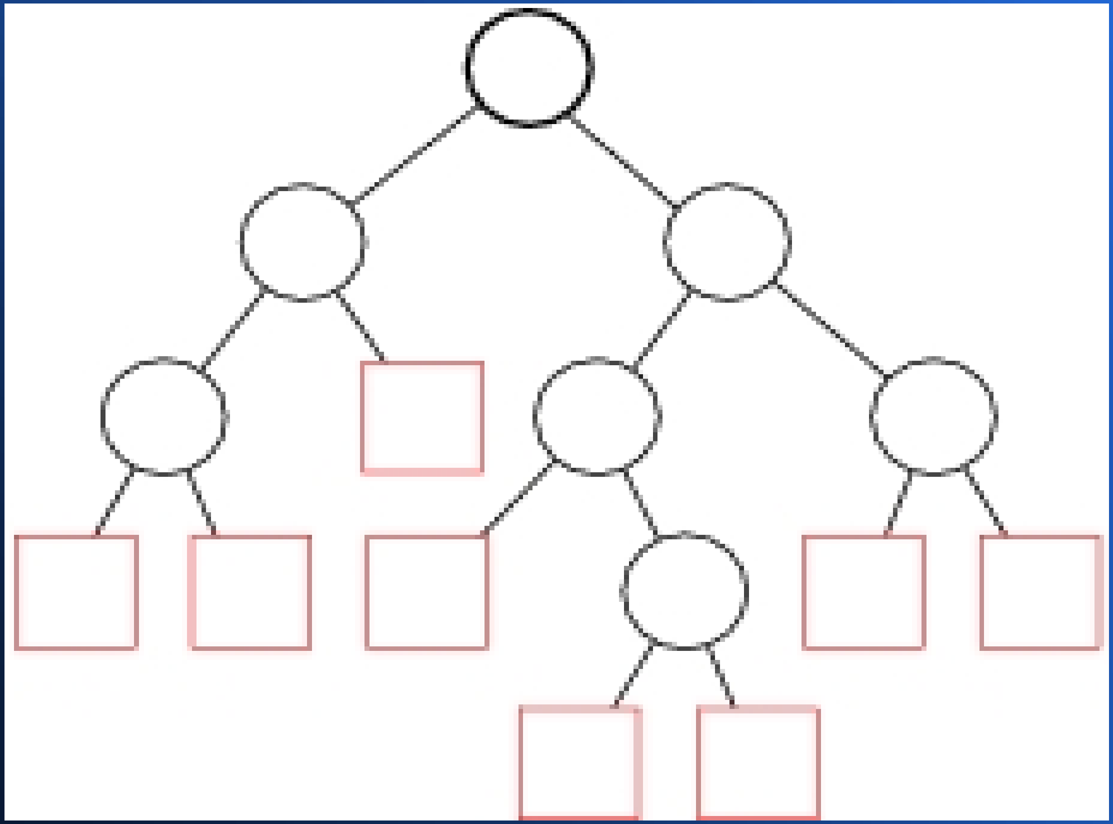
Definitions of binary tree

(iii) Nothing else is a finite tree.

Definitions of binary tree



Definitions of binary tree



Boxes represent empty subtrees

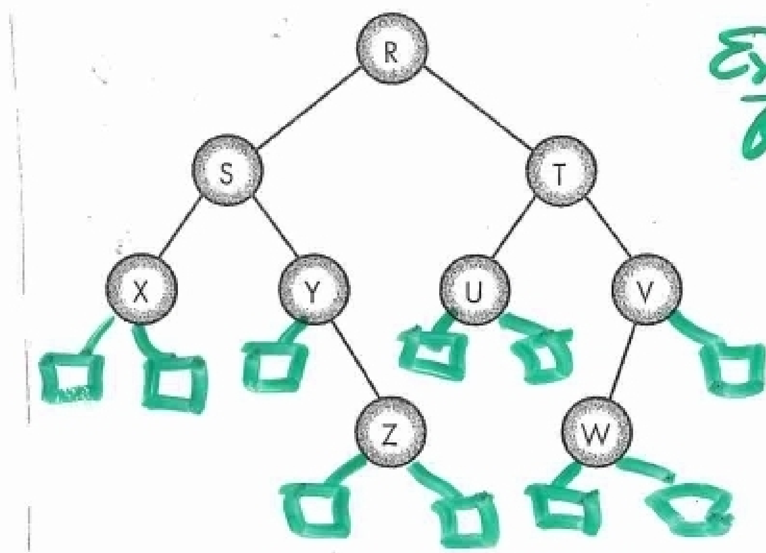
8.3 Binary Trees

LEARNING OBJECTIVES

1. To become familiar with the definition of binary trees.
2. To learn the definition of extended and complete binary trees.
3. To prepare for the discussion of binary tree representations and binary tree operations in the remainder of the chapter.

finite

A **binary tree** is either the *empty tree* or a node that has *left* and *right* subtrees that are binary trees.



Extended binary tree

2-tree

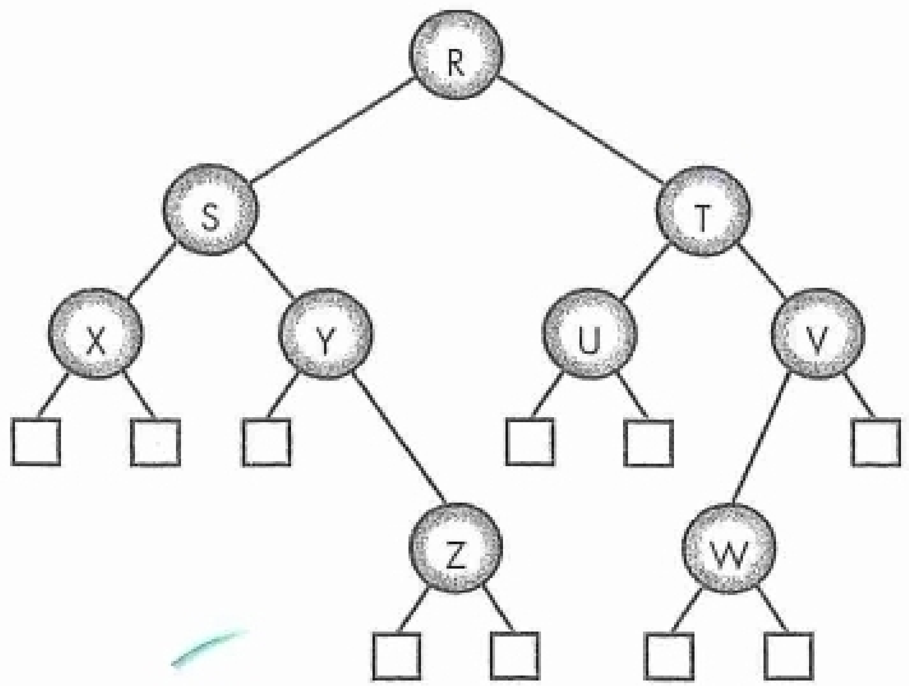


Figure 8.3 An Extended Binary Tree

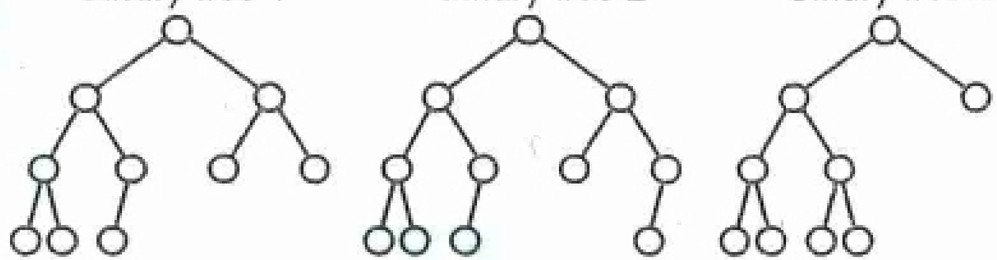


Figure 8.4 Complete and Incomplete Binary Trees

8.3 EXERCISES

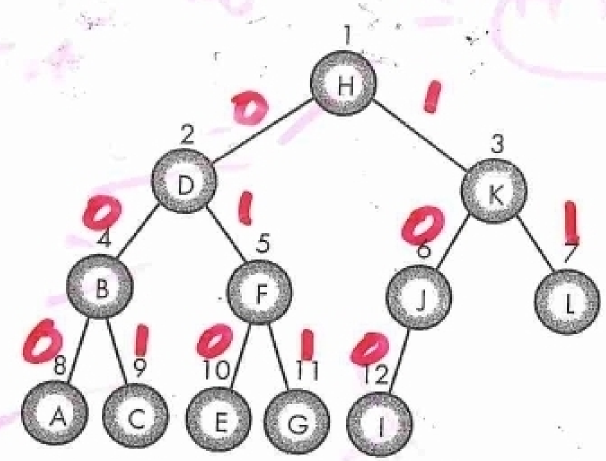
1, 2 p. 250

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:

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

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[2 i]	2 i ≤ n
The right child of A[i]	A[2 i + 1]	2 i + 1 ≤ 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	2 i > n

n > 0

8.4 EXERCISES

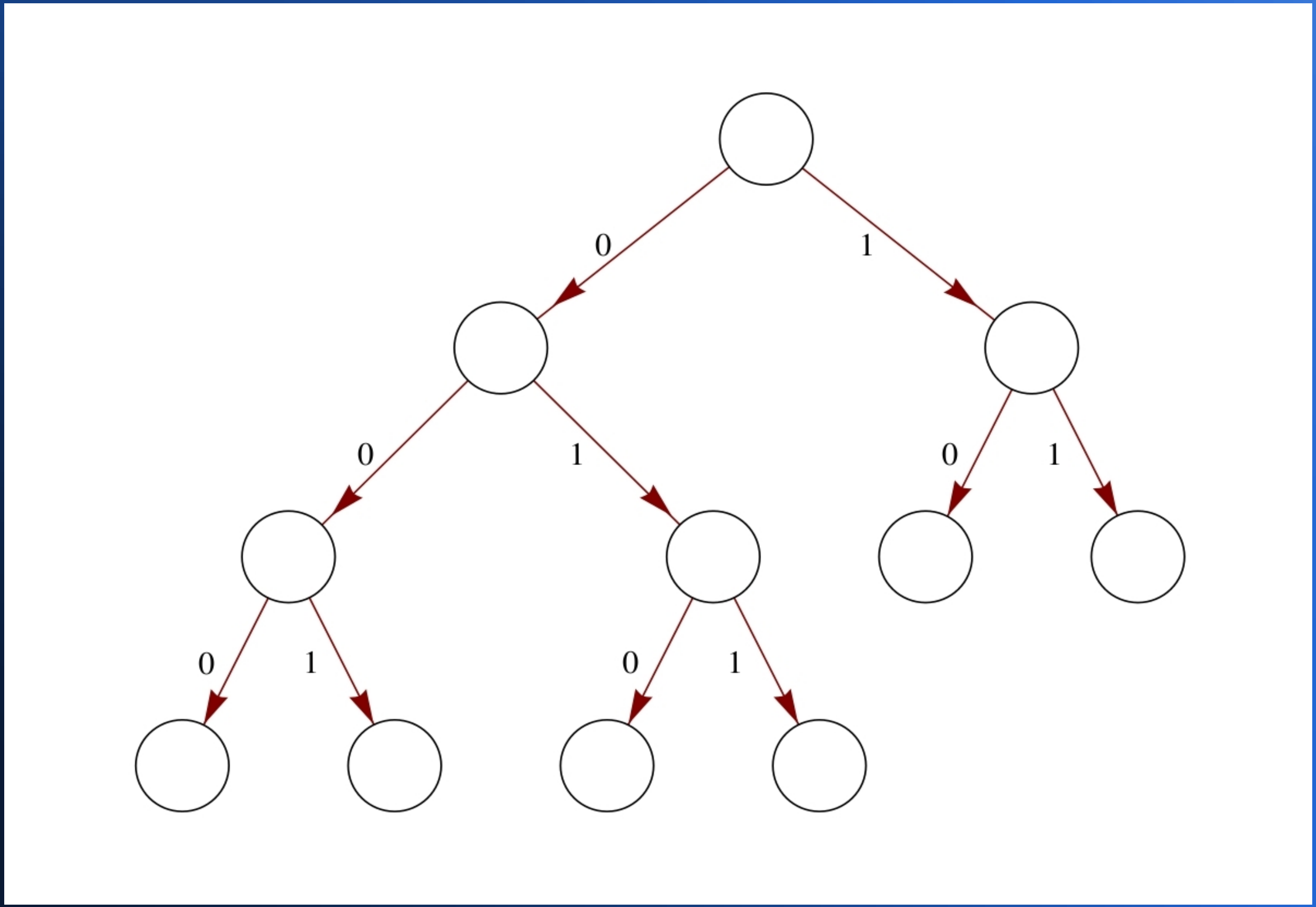
1, 2, 3 p 252

8.5 An Application—Heaps and Priority Queues

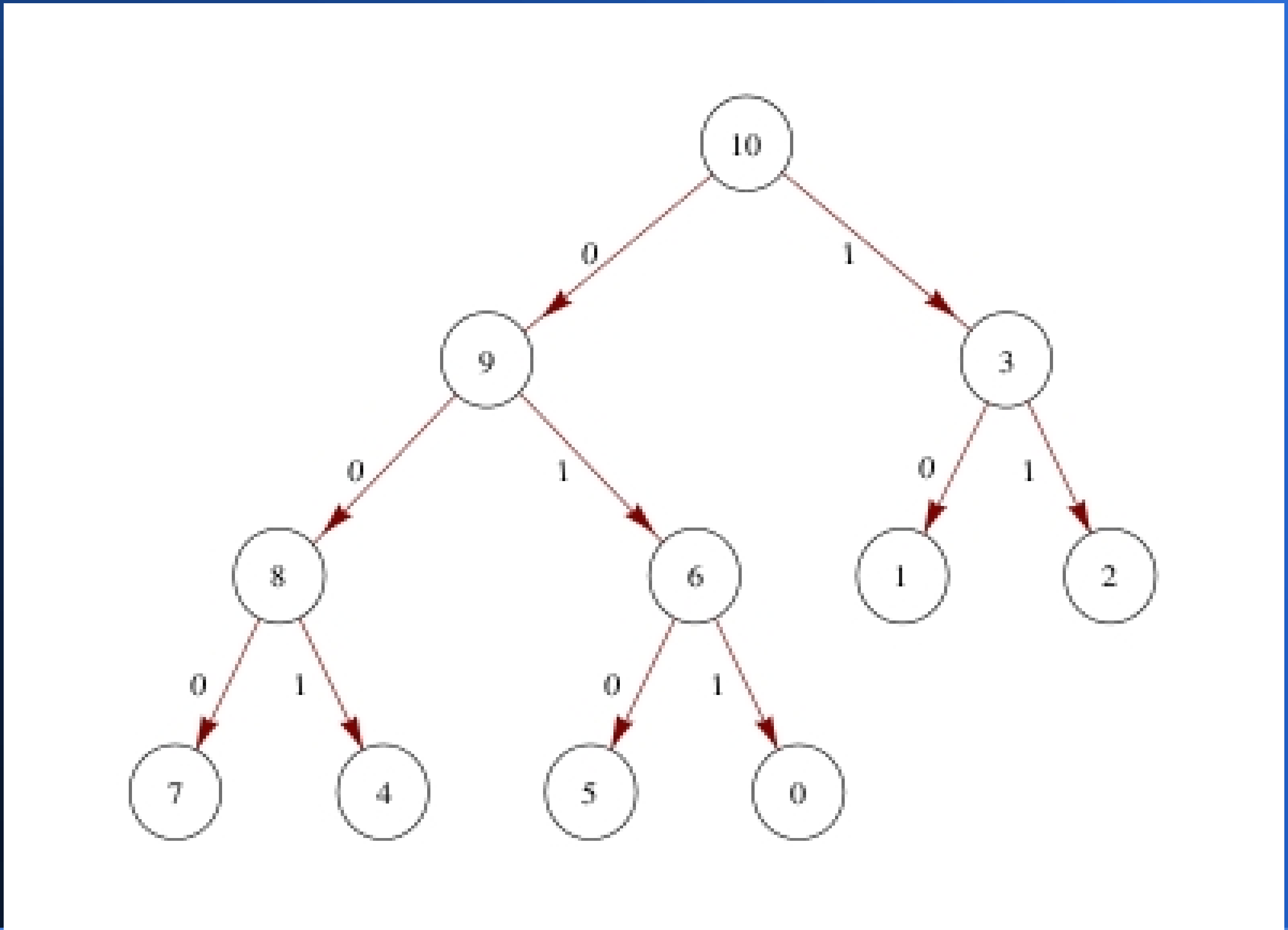
LEARNING OBJECTIVES

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.

Level-by-level complete 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.

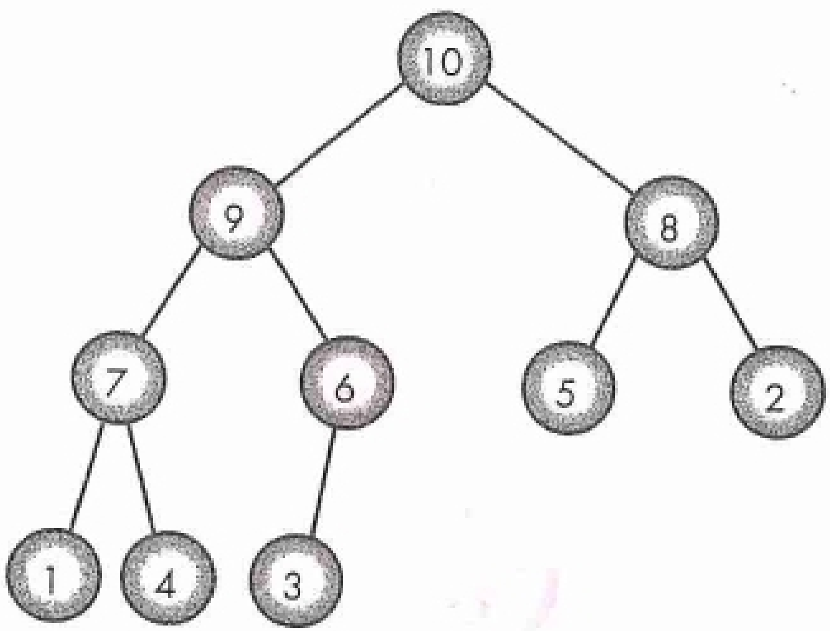


Figure 8.8 An Example of a Heap

Priority Queue (heap)

```
/*
 *   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

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

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)

```
public class PriorityQueue {

    //Uses heap as implementation


    private int count; //actual number of elements
    private int capacity; //the size of the array - 1
    private int capacityIncrement;
    private int[] itemArray;

```

Priority Queue (heap)

```
    /** Creates a new instance of PriorityQueue */
    public PriorityQueue() {
        count=0;
        capacity=10;
        capacityIncrement=2;
        itemArray=new int[capacity + 1]; //itemArr
    }
```

Priority Queue (heap)

```java
    public void insert(int newItem)
    {
        if(count==capacity) //no more space, "resize"
        {
            capacity*=capacityIncrement;
            int[] tempArray = new int[capacity + 1];
            for (int i = 1; i <= count; i++)
            {
                tempArray[i] = itemArray[i];
                 cnt2.incr();
            }
            itemArray = tempArray;
```

Priority Queue (heap)

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

Priority Queue (heap)

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

```

Priority Queue (heap)

```
public int remove()
{
 if (count==0) return -9999;
 //here count != 0
    int maxItem = itemArray[1]; //the root
    int demotee = count;
    count--;
    int i = 1;
    boolean demoted = true;
```

Priority Queue (heap)

```
        while ((2*i <= count) && demoted)
        {
            int j = 2*i; // first child
            if ((j < count) && (cnt3.incr() & (itemArray[j] < itemArray[j + 1]))) j++;
            if (cnt3.incr() & (itemArray[j] > itemArray[demotee])) //demote patch
            {
                itemArray[i] = itemArray[j]; //promote its larger child
                cnt.incr();
                i = j; //demote patch's index

            }
```

Priority Queue (heap)

```
            }
            else
            {
                demoted = false;
            }
        }
        //i is the place for the patch
        itemArray[i] = itemArray[demotee];
        cnt.incr();
        itemArray[demotee] = 0; // for demonstration purpose only
```

Priority Queue (heap)

```
        if ((count < capacity / capacityIncrement) &&  (10 <= capacity / capacityIncre
        {
          capacity/=capacityIncrement;
            int[] tempArray = new int[capacity+1];  //because itemArray[0] is not use
            for (i = 1; i <= count; i++)
            {
                tempArray[i] = itemArray[i];
                cnt2.incr();
            }
            itemArray = tempArray;
        }
        return maxItem;
    }
```

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( ); O(1) // let PQ be initially empty
    for (i = 0; i < n; i++) PQ.insert(A[i]);     f(i)  // put A's items into PQ
    for (i = n–1; i >=0; i––) A[i] = PQ.remove( );  g(i+1) // remove PQ's items
                                             // and put them in A
}
```

Program 5.2 A Priority Queue Sorting Method

worst case

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

$$\le \boxed{nf(n)}$$

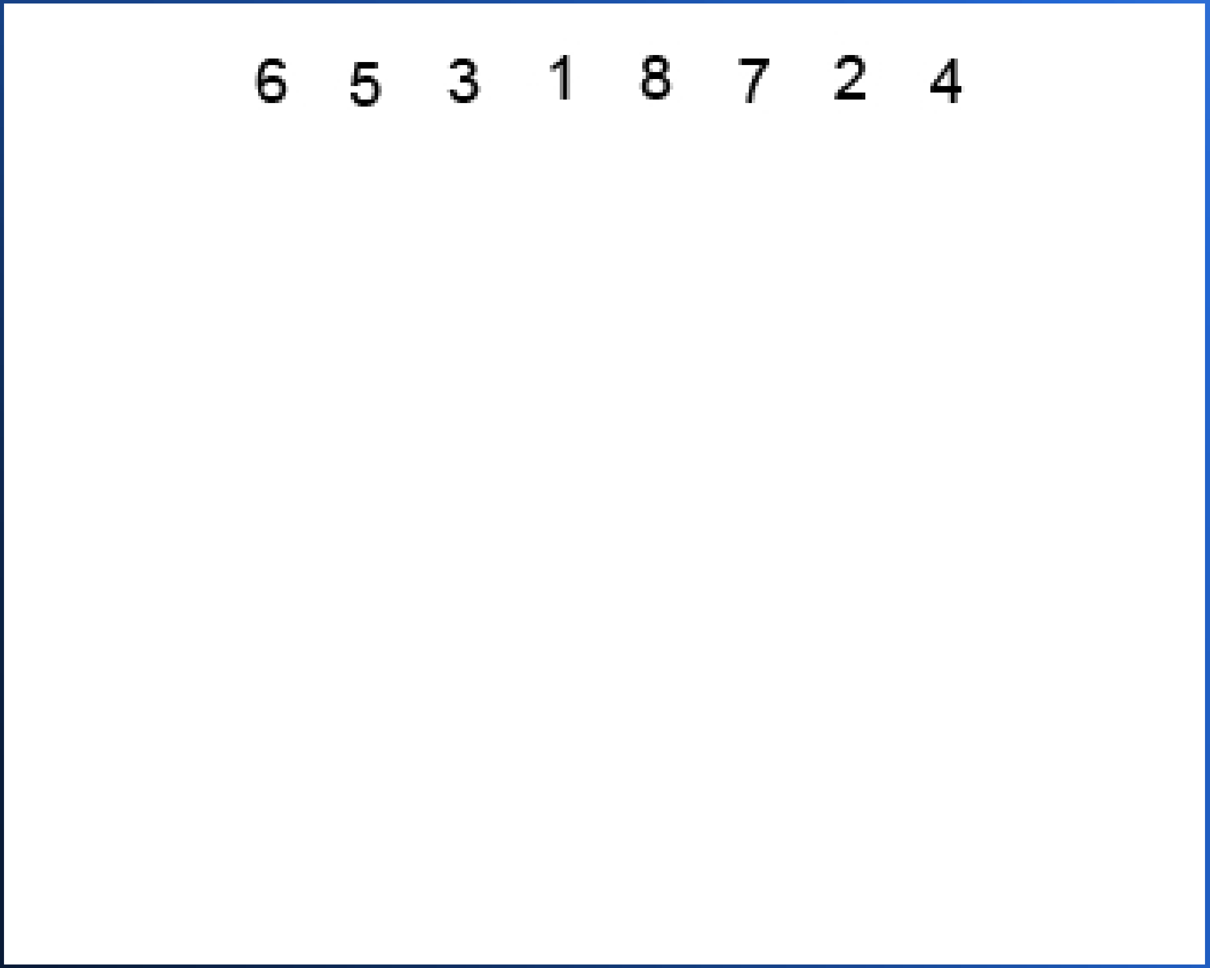
$$\sum_{i=0}^{n-1} g(i+1) \le \sum_{i=0}^{n-1} g(n) = \boxed{ng(n)}$$

$$\text{Total} \le O(1 + n \cdot f(n) + n \cdot g(n)) = O(1 + n(f(n) + g(n)))$$

$$= O(n(f(n) + g(n))) =$$

$$= \boxed{O(n \cdot \max(f(n), g(n)))}$$

Priority Queue (heap)



http://upload.wikimedia.org/wikipedia/commons/4/4d/Heapsort-example.gif

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

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

Exceptfrom "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( );  O(1) // let PQ be initially empty
    for (i = 0; i < n; i++) PQ.insert(A[i]);   f(i)   // put A's items into PQ
    for (i = n–1; i >=0; i––) A[i] = PQ.remove( );  g(i+1)  // remove PQ's items
                                           // and put them in A
}
```

Program 5.2 A Priority Queue Sorting Method

worst case

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

$$\le n f(n)$$

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

$$\text{Total} \le O(1 + n \cdot f(n) + n \cdot g(n)) = O(1 + n(f(n) + g(n)))$$

$$= O(n(f(n) + g(n))) =$$

$$= O(n \cdot \max(f(n), g(n)))$$

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

    S.push(T);                              // push the pointer T onto the empty stack S

    while ( !S.empty( ) ) {

        N = (TreeNode)S.pop( );             // pop top pointer of S into N

        if (N != null) {
            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 */

    if (T != null) {                                   // if T == null, do nothing

        if ( traversalOrder == PRE_ORDER )  {

            visit(T);
            traverse(T.llink, PRE_ORDER);
            traverse(T.rlink, PRE_ORDER);

        } else if ( traversalOrder == IN_ORDER ) {

            traverse(T.llink, IN_ORDER);
            visit(T);
            traverse(T.rlink, IN_ORDER);

        } else if ( traversalOrder == POST_ORDER ) {

            traverse(T.llink, POST_ORDER);
            traverse(T.rlink, POST_ORDER);
            visit(T);
        }
    }
}
```

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

      Q.insert(T);                     // insert the pointer T into queue Q

      while ( ! Q.empty( ) ) {

            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
                  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.

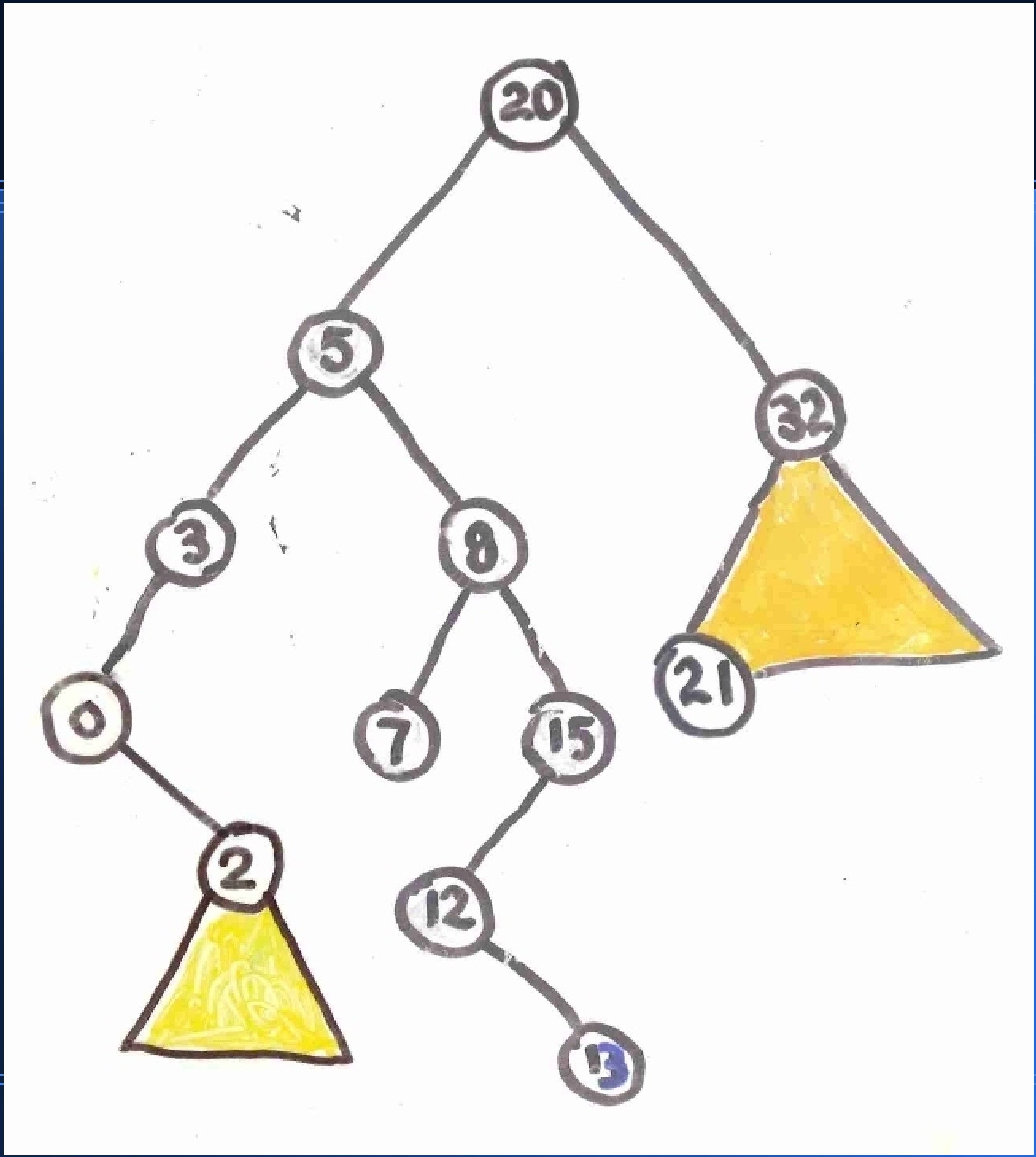
A binary tree T is called a binary search tree if, and only if, in-order traversal with listing of T lists the nodes of T in an increasing order.

BS Tree

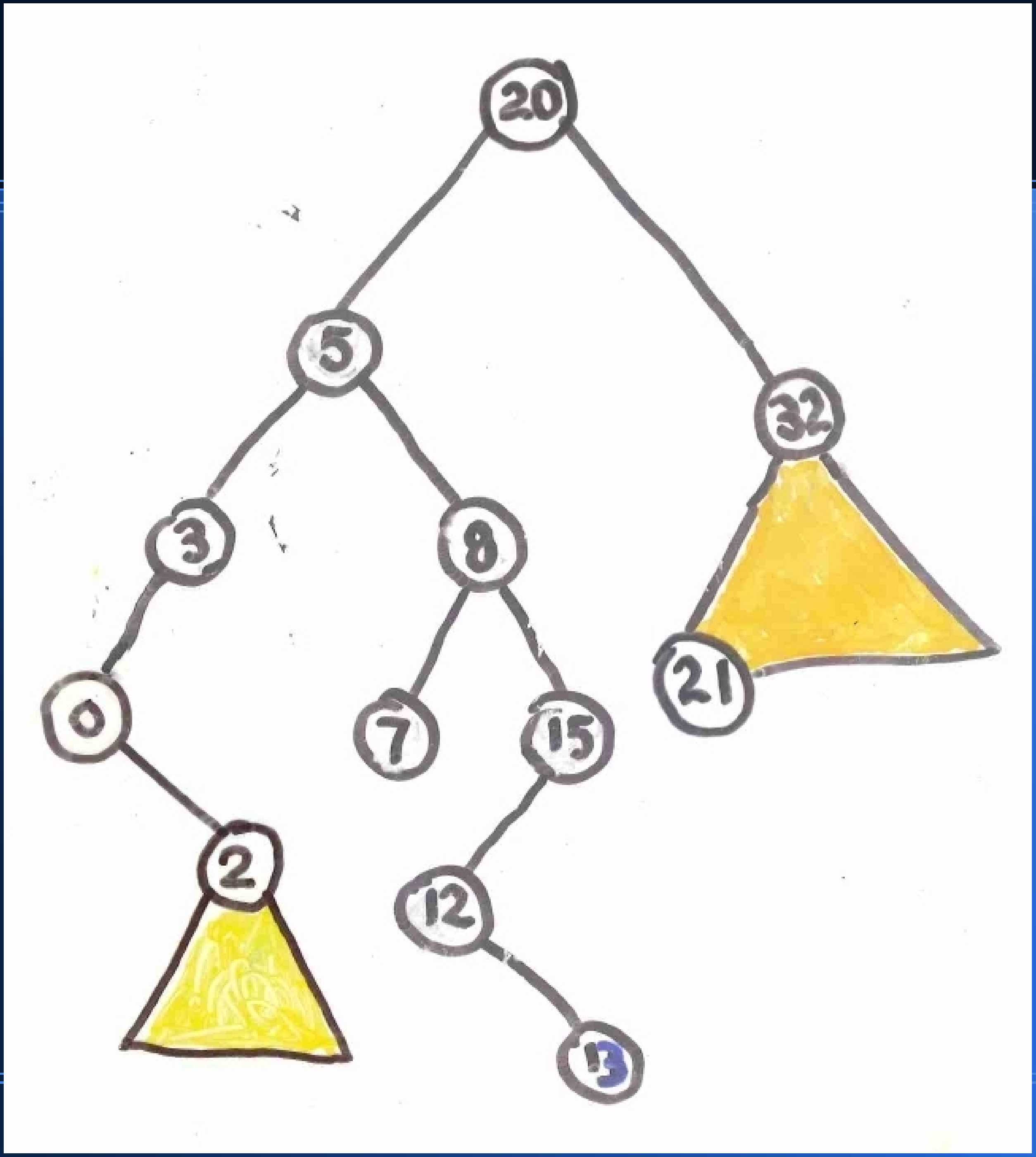
Exercise

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

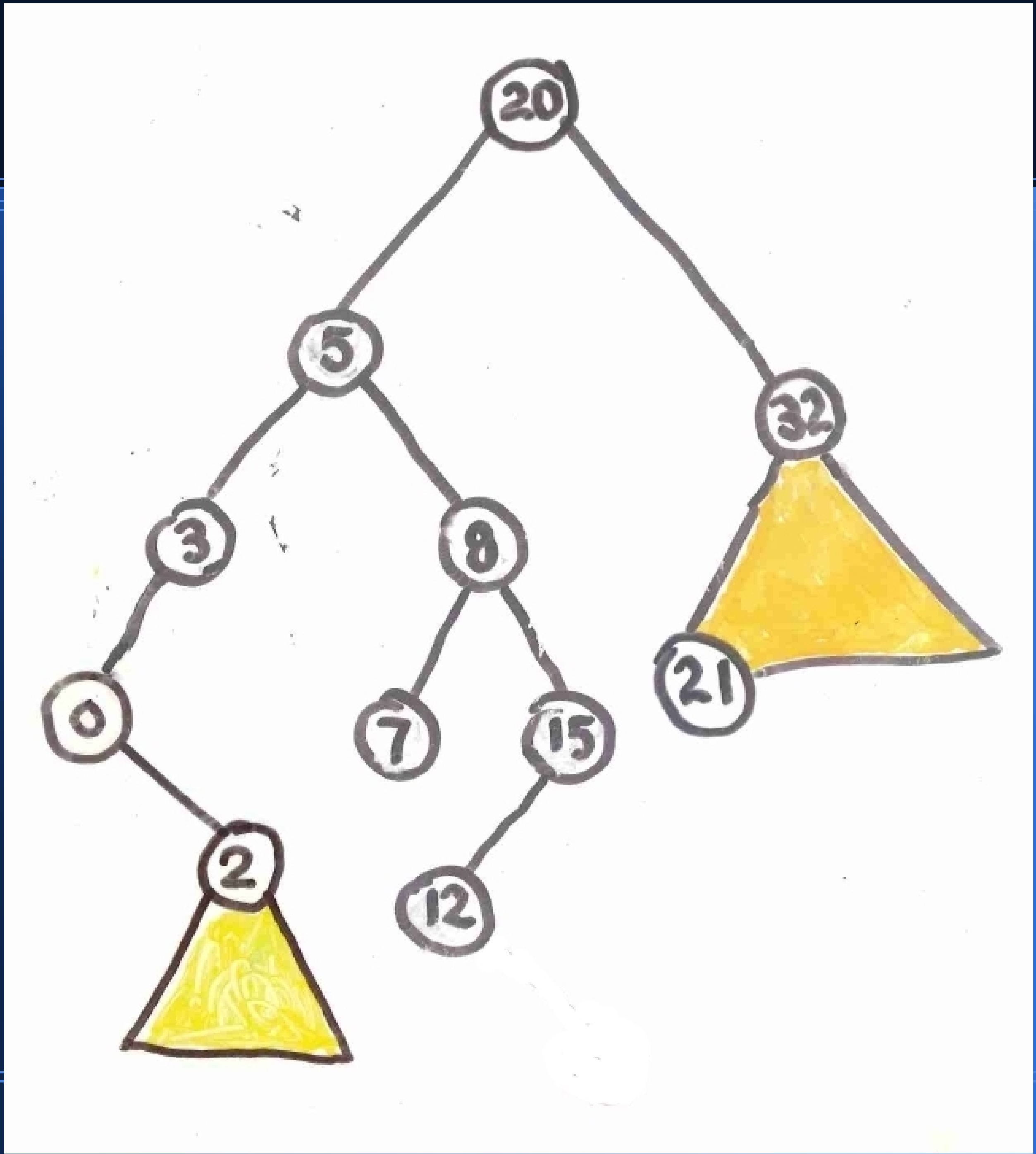
Delete



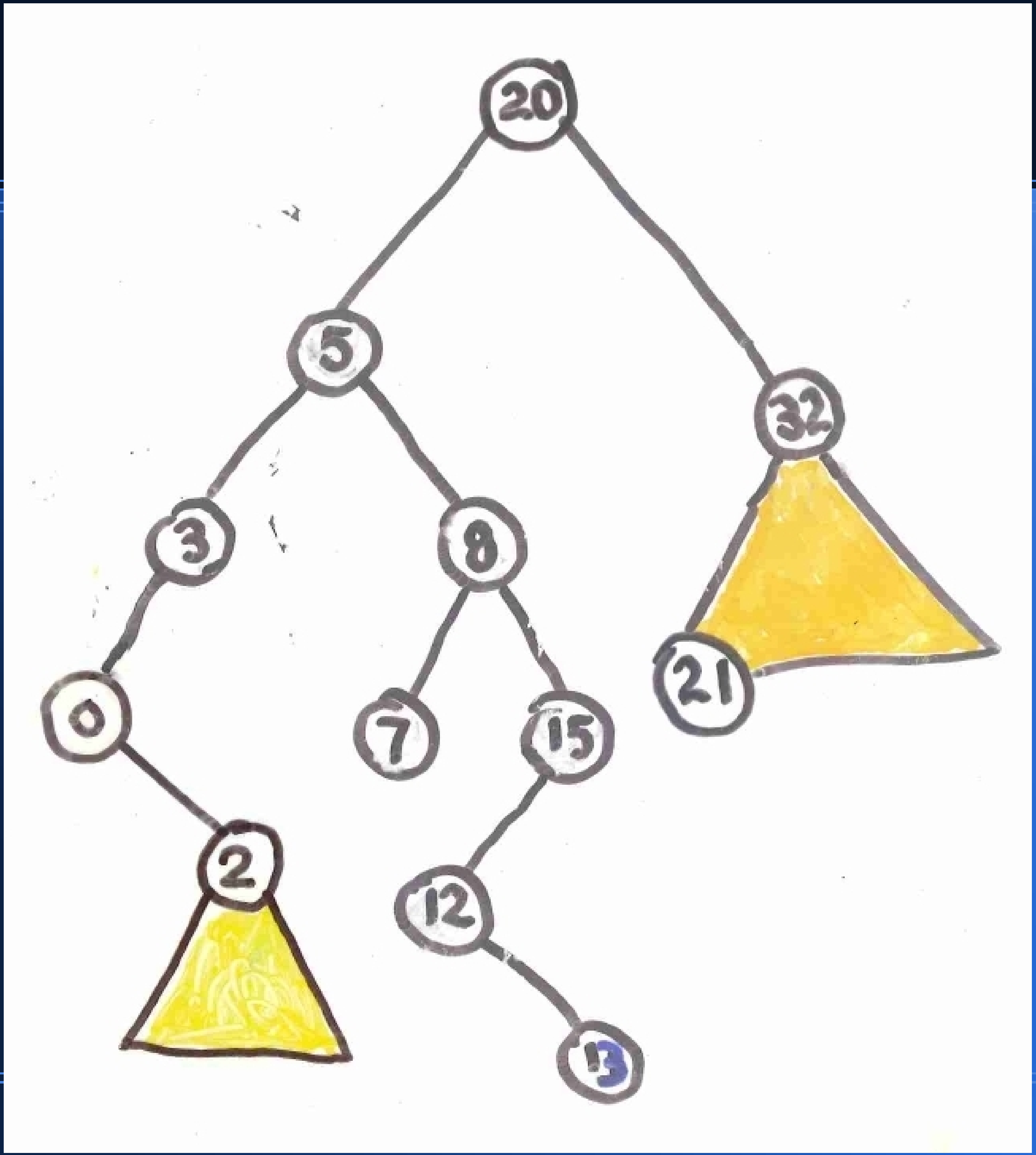
Delete(13)



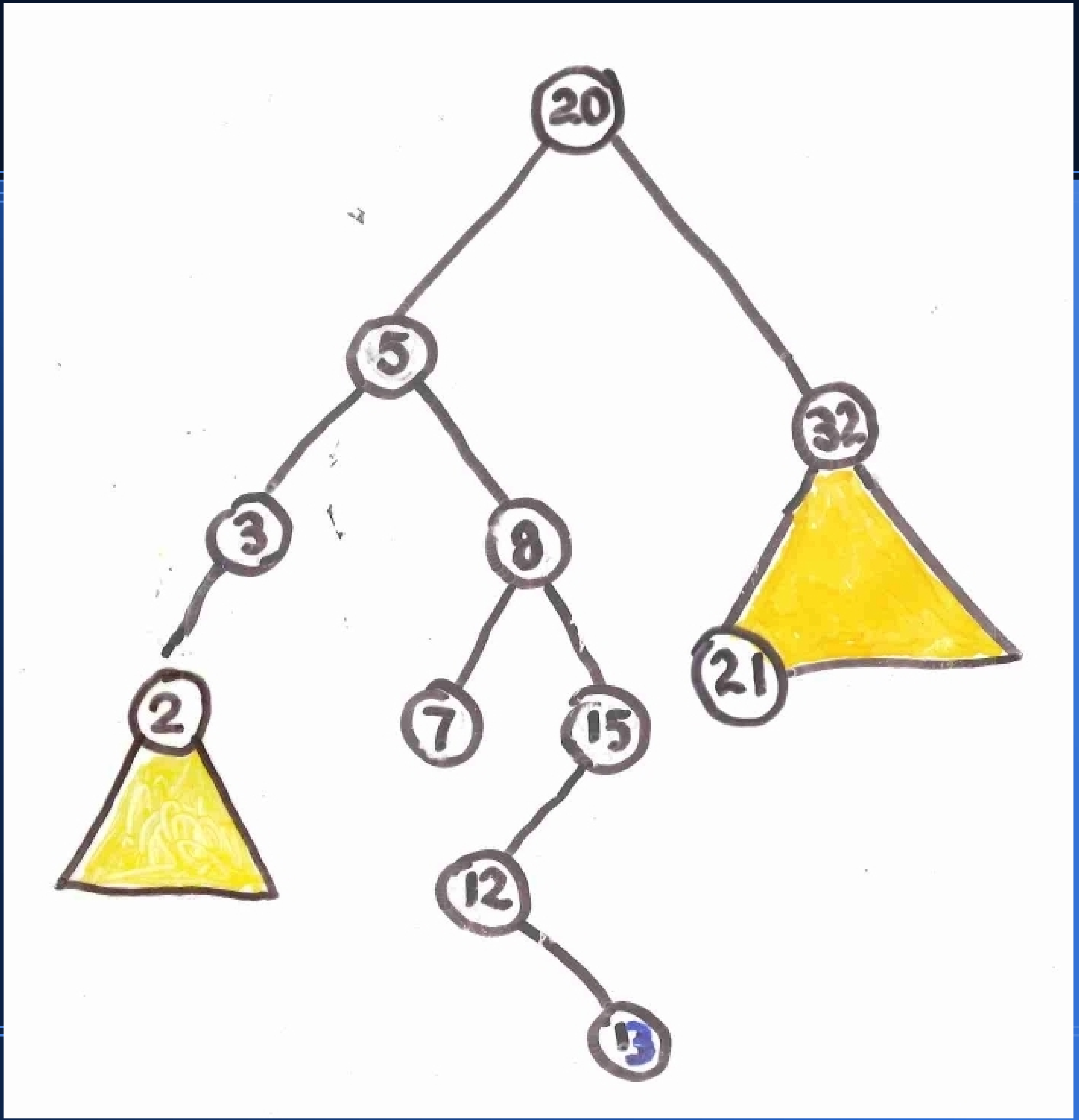
Delete(13)



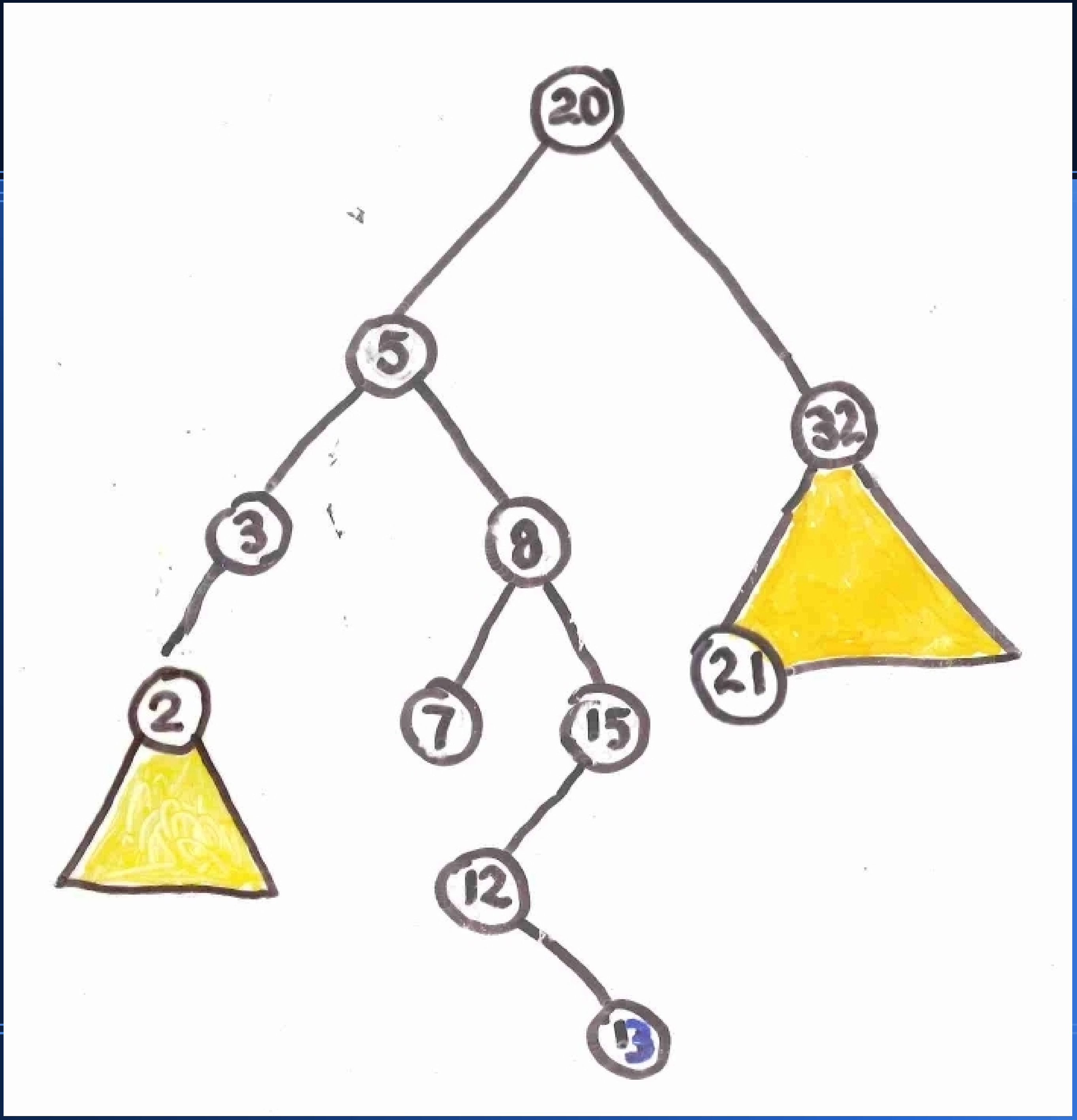
Delete(0)



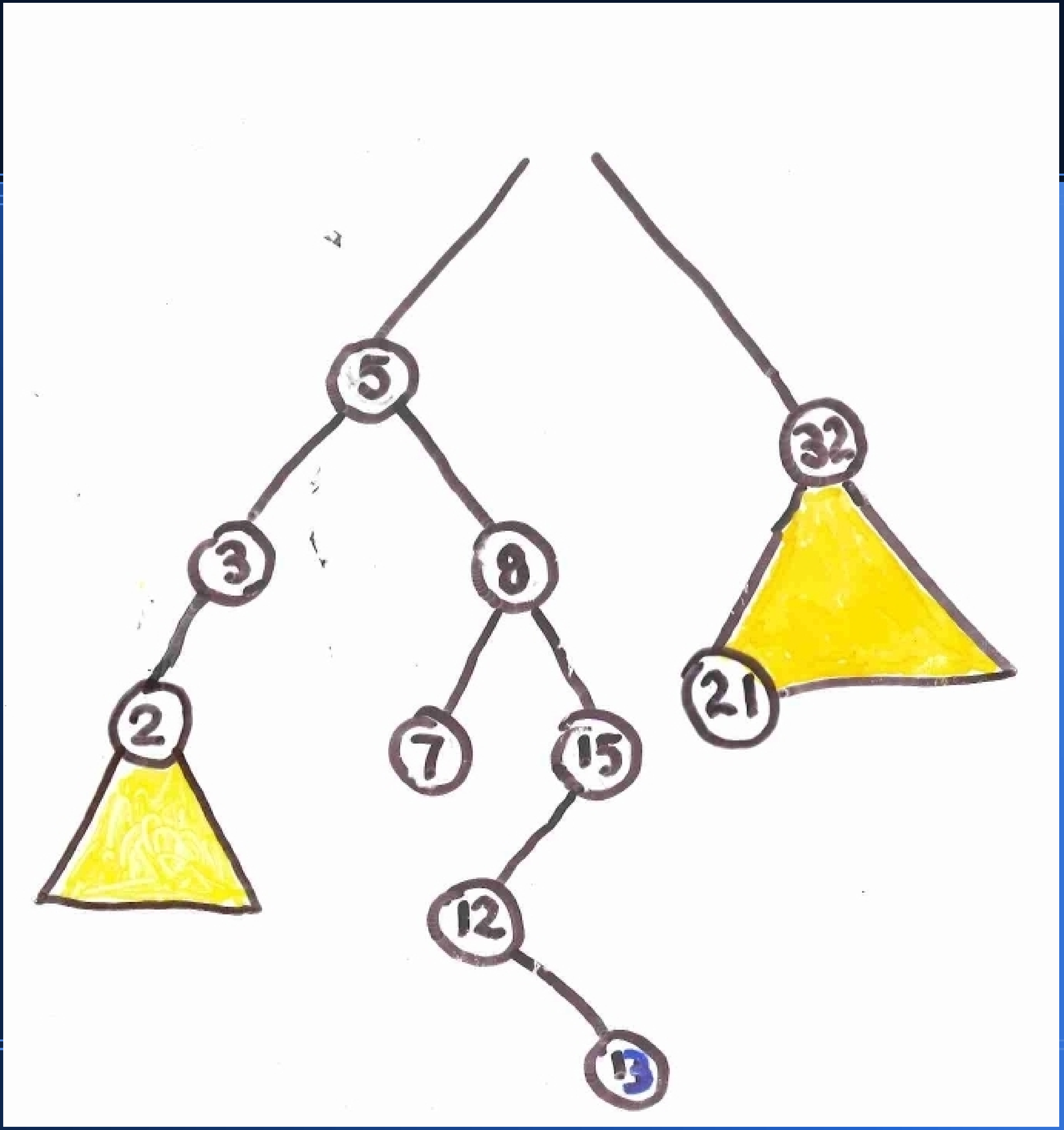
Delete(0)



Delete(20)

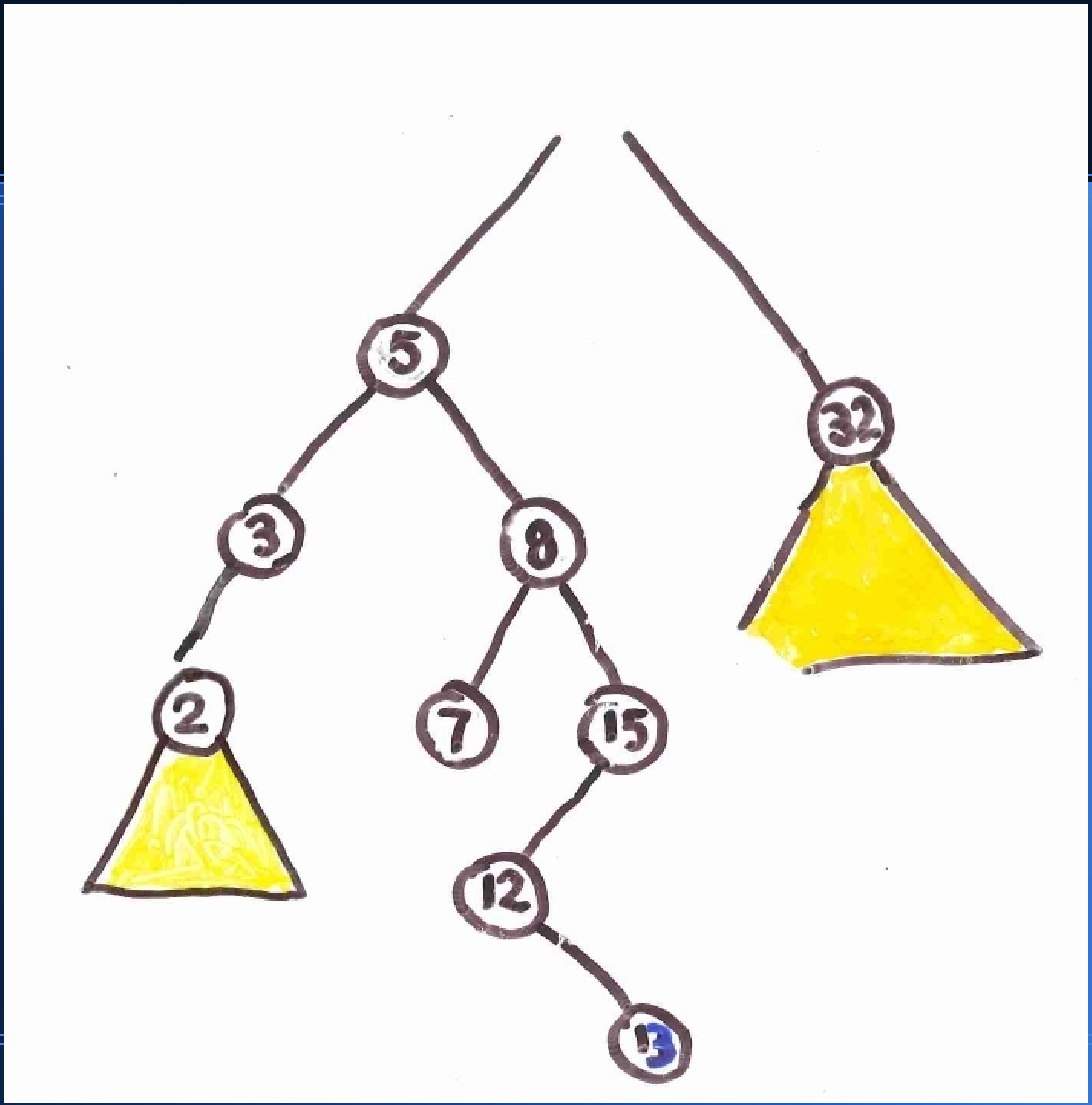


Delete(20)

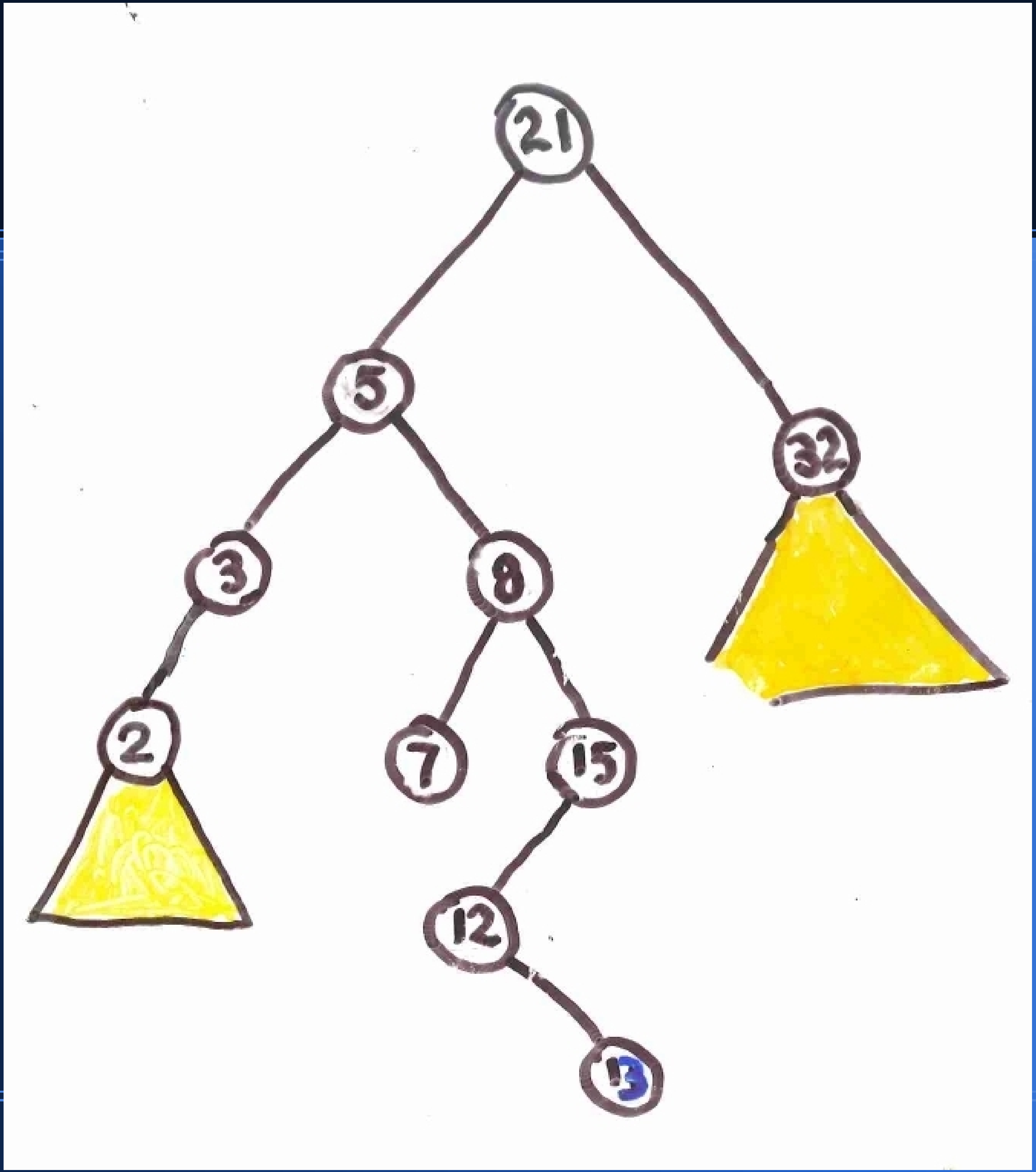


Delete(20)

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.

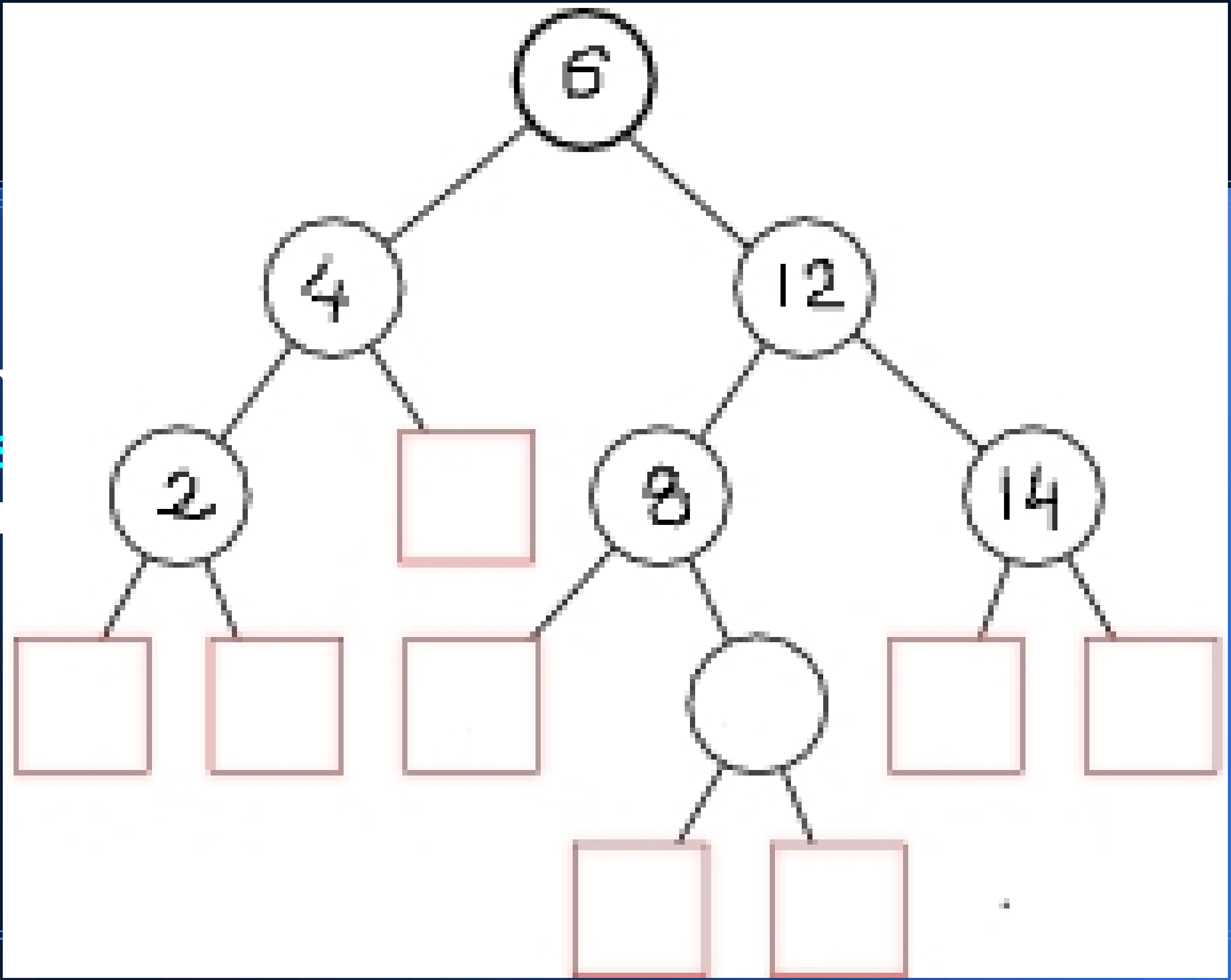
BS Tree

FACT 2

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

level(x)

of the newly inserted node with value x.

Th
s
bi


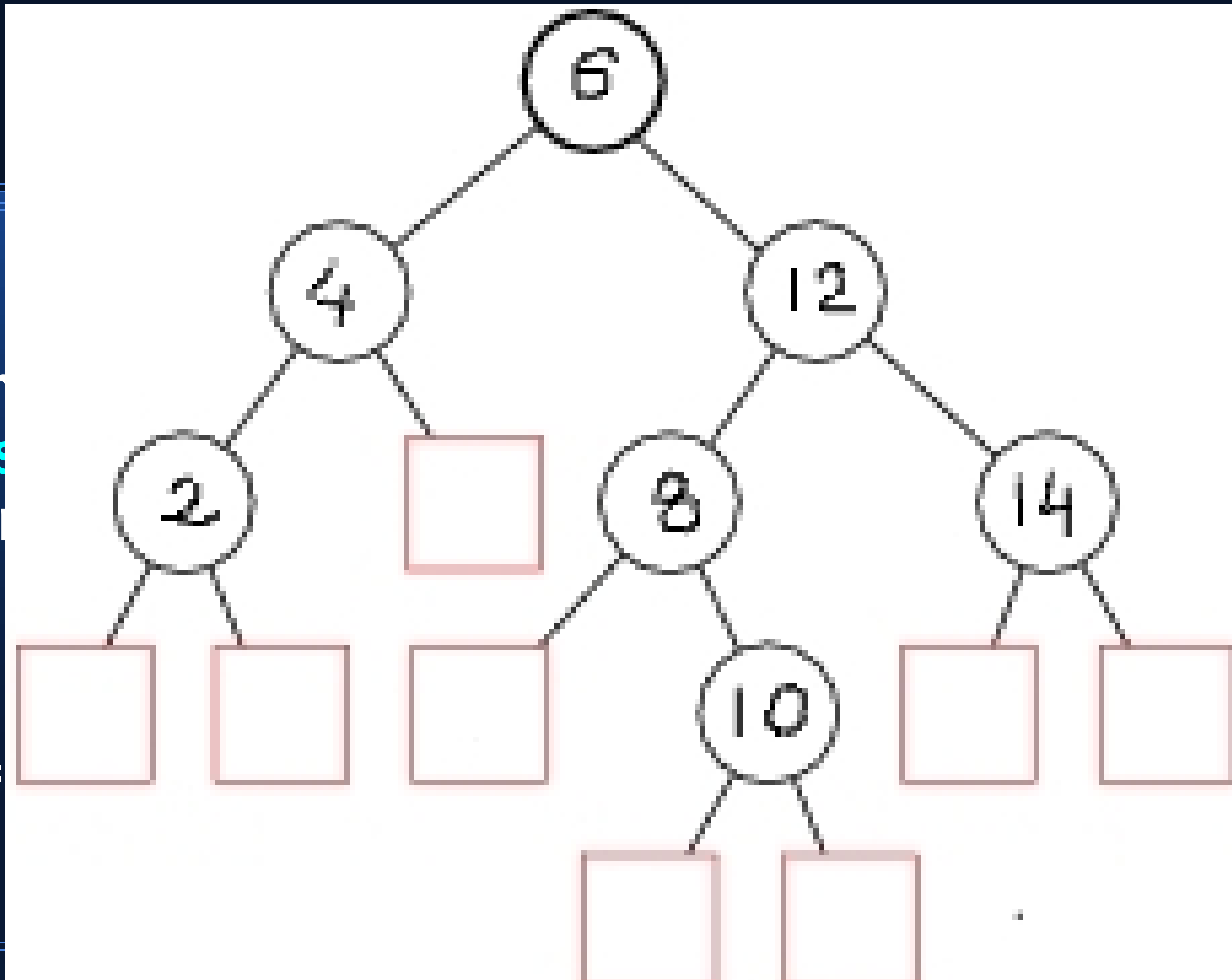
of

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:

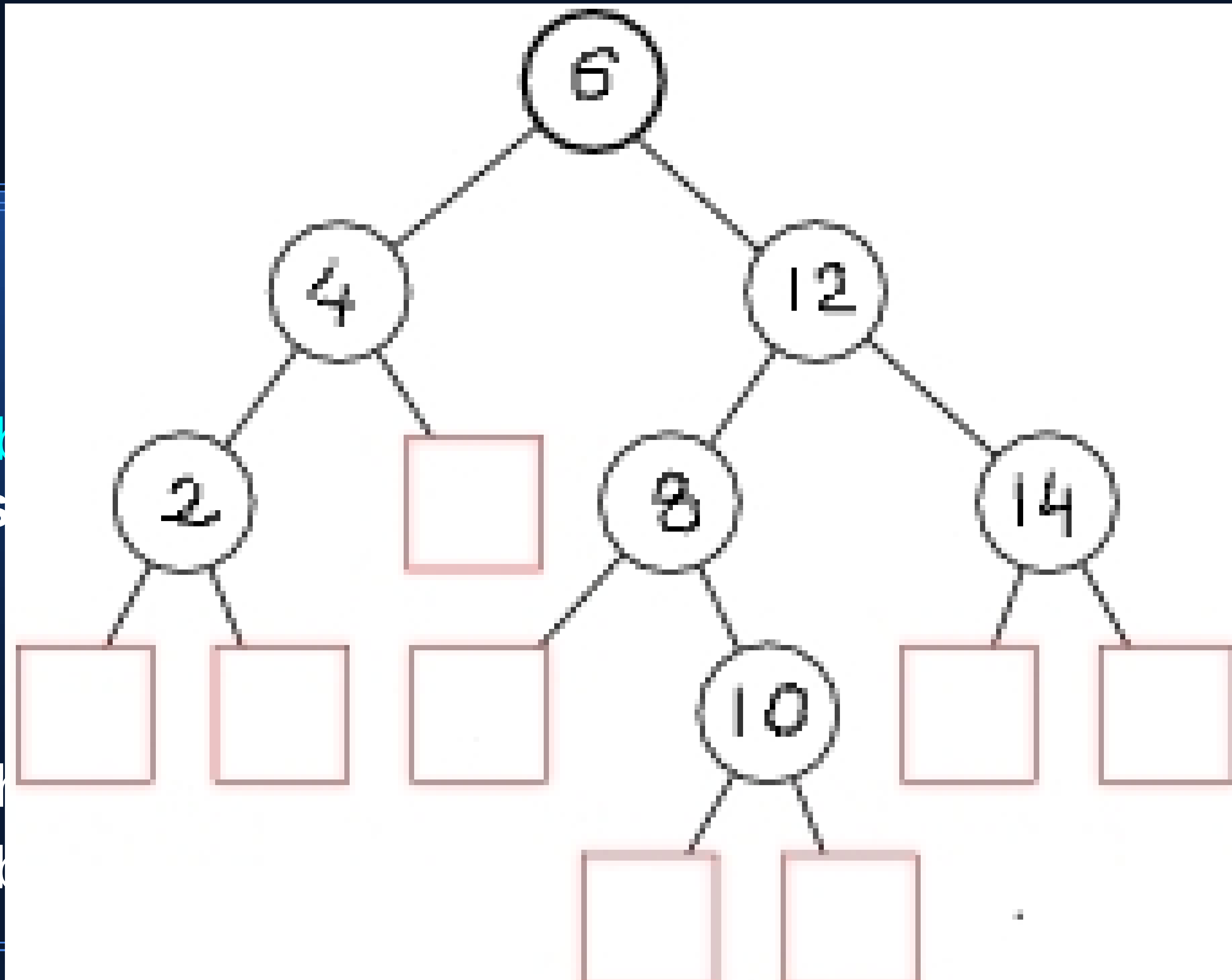
$$level(x_1) + level(x_2) + ... + level(x_n),$$

where $level(x_i)$ is the level number to which x_i belongs.

In e

b

as



wh

b

BS Tree

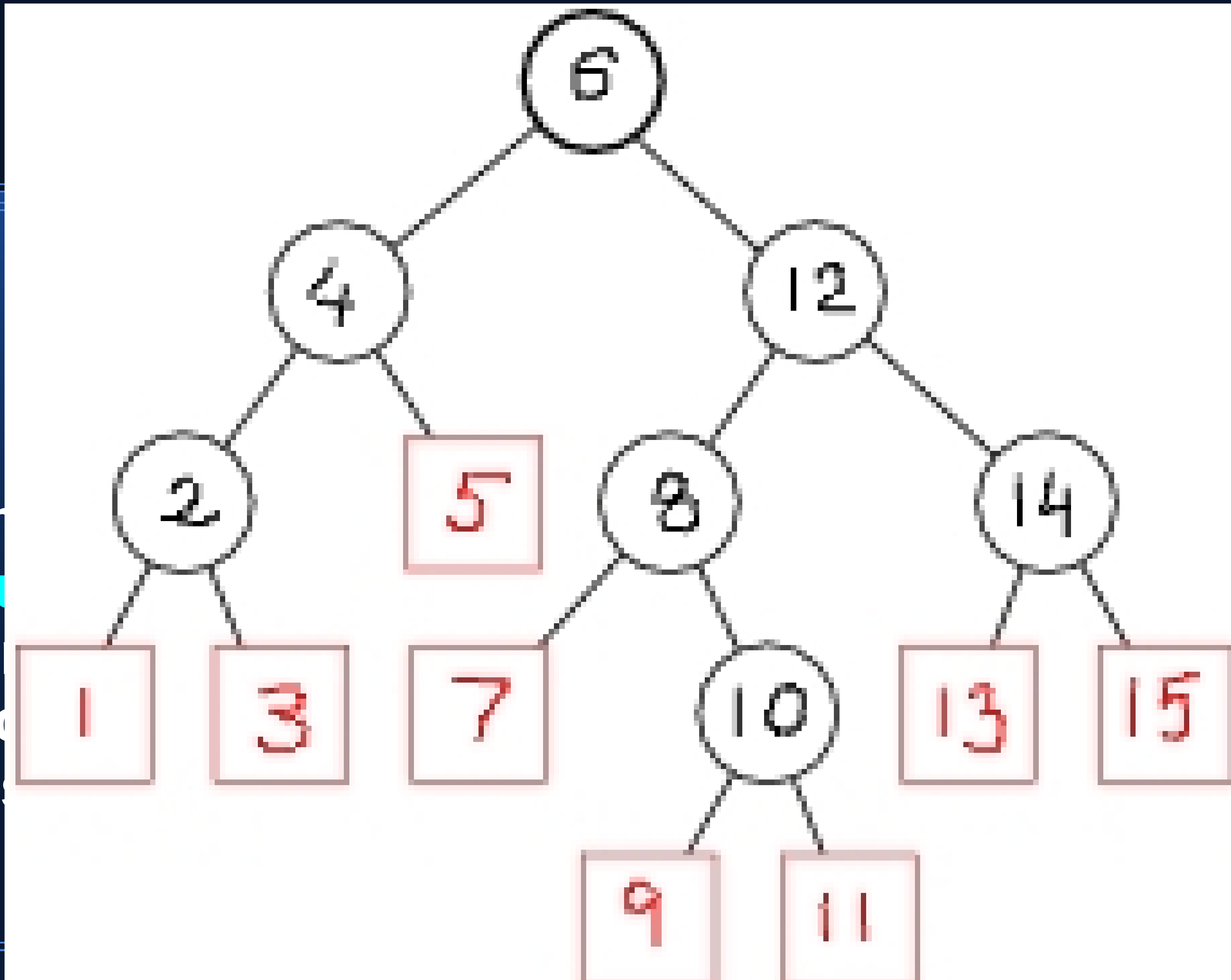
FACT 3

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

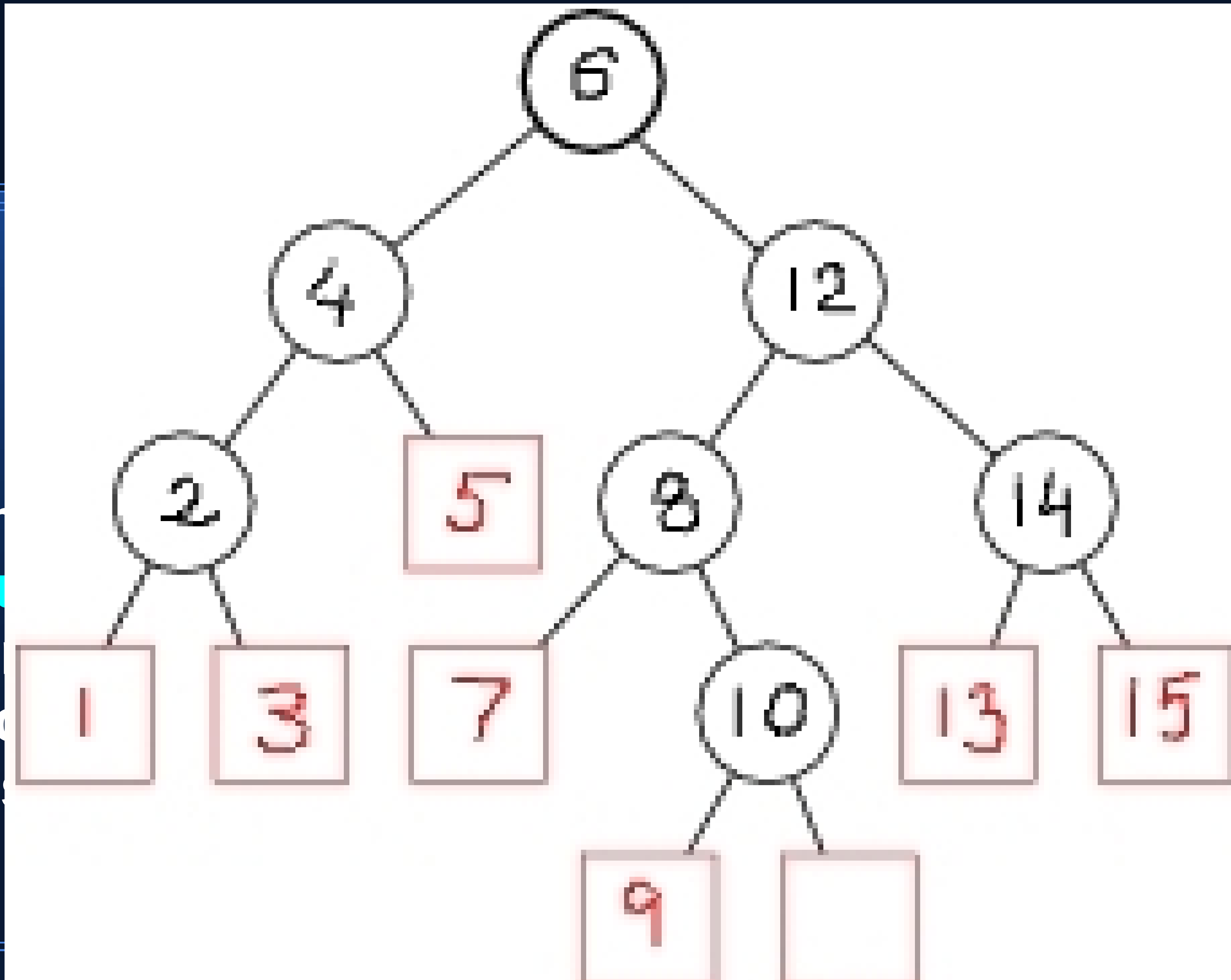
in



Th

bi

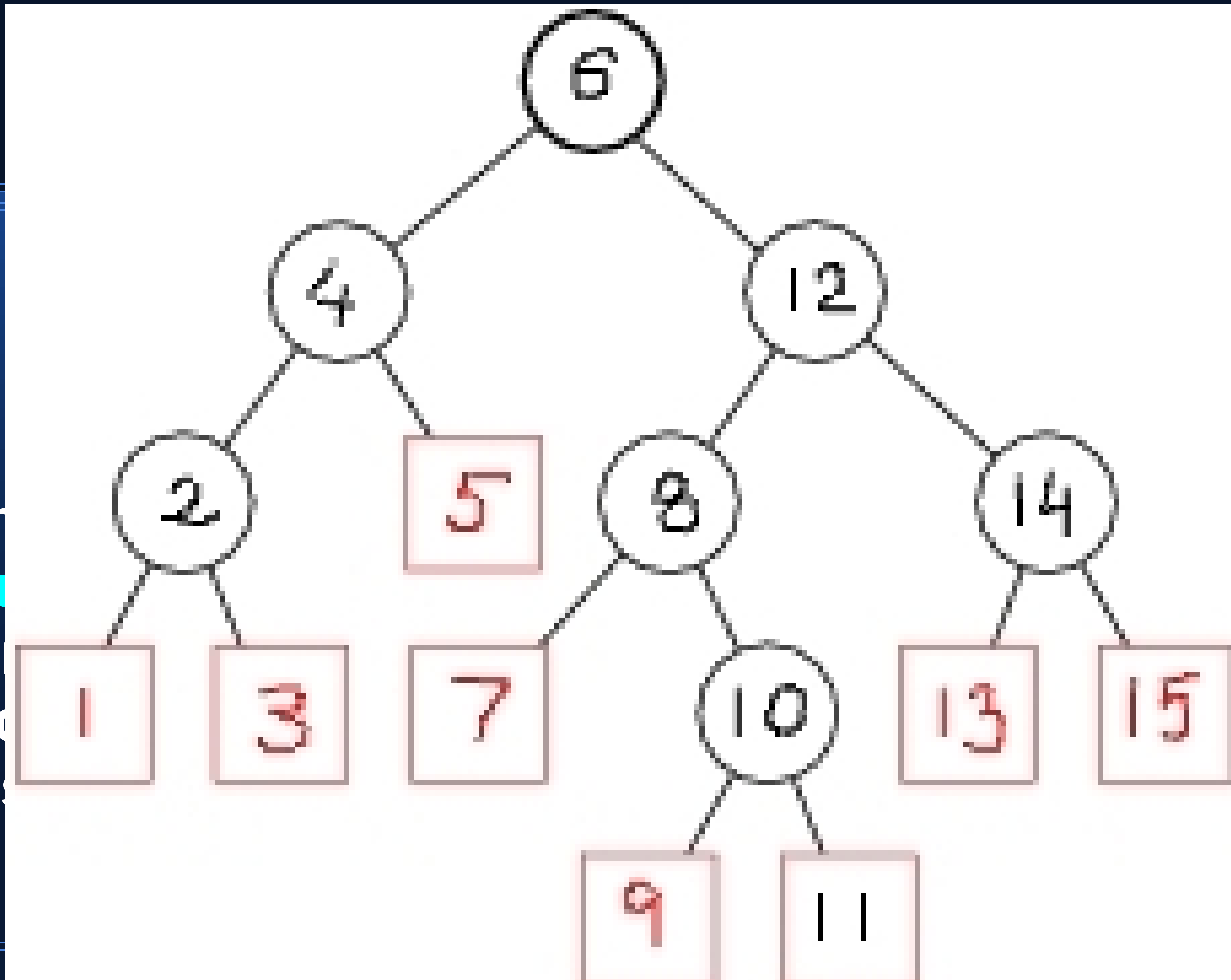
in



Th

bi

ins



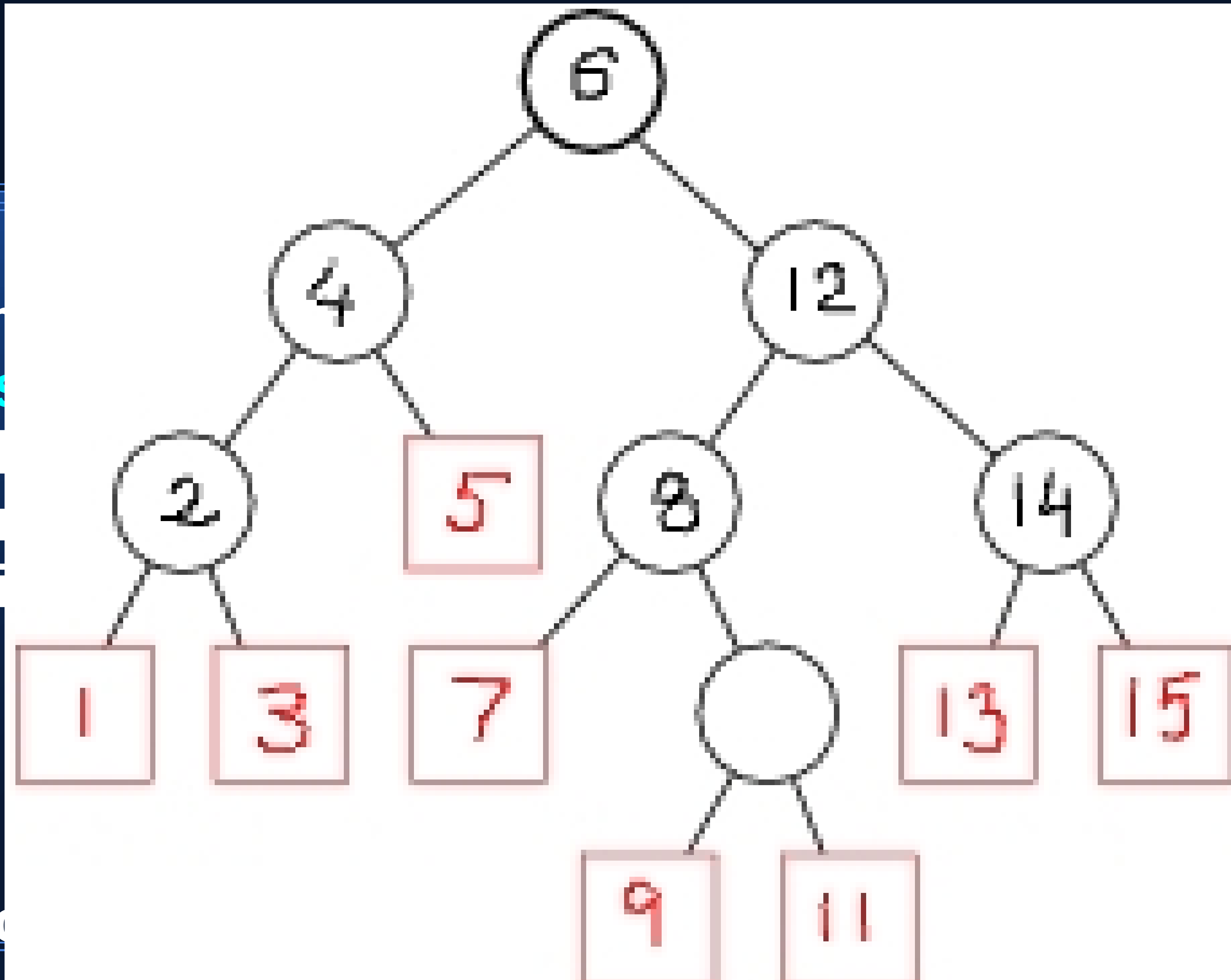
BS Tree

FACT 4

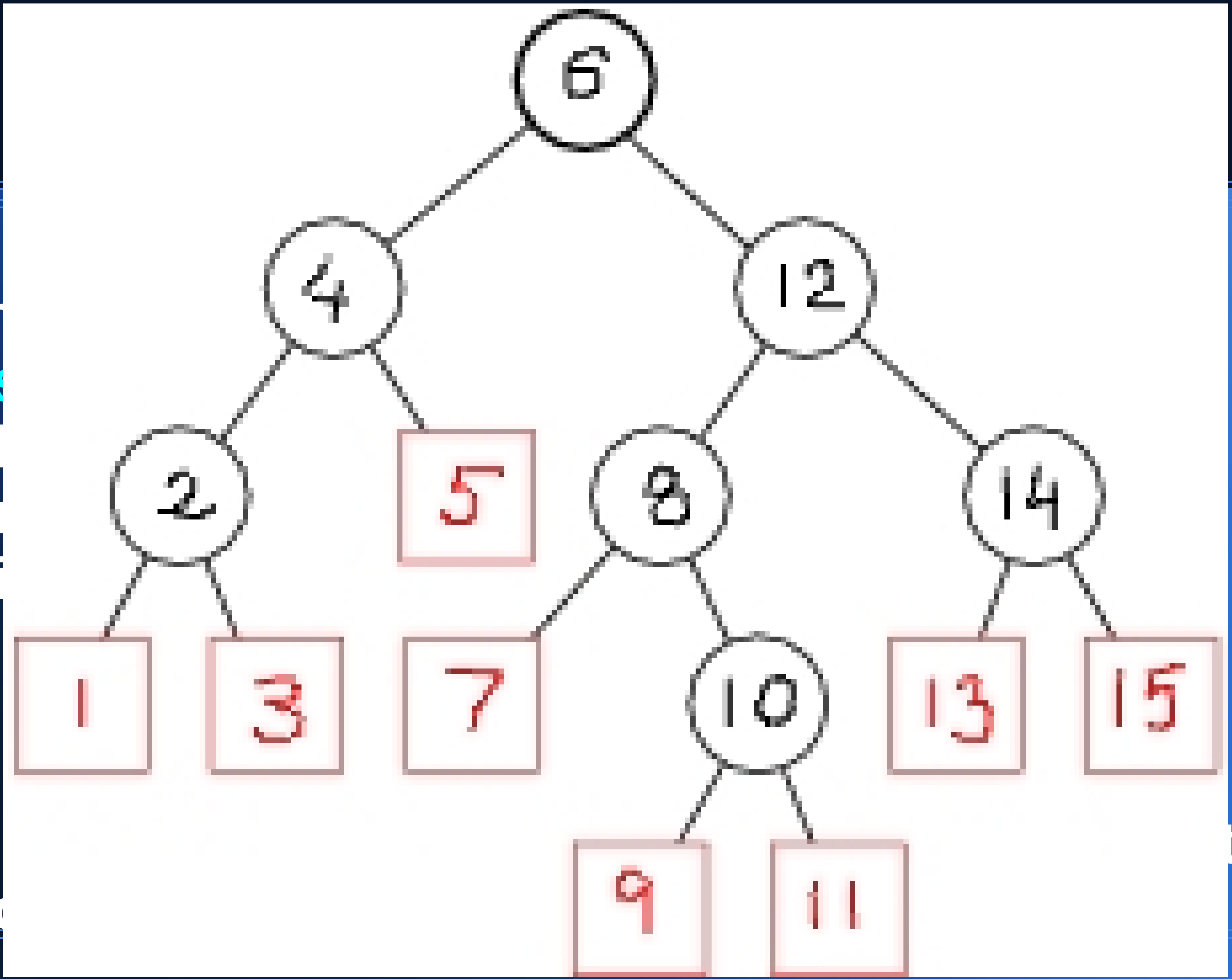
The number of comparisons while **successfully searching** for value x in a binary search tree is equal to 1 plus the number of comparisons made while inserting x into T,

$$1 + level(x)$$

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


6
4
12
2
5
8
14
1
3
7
13
15
9
11

Th



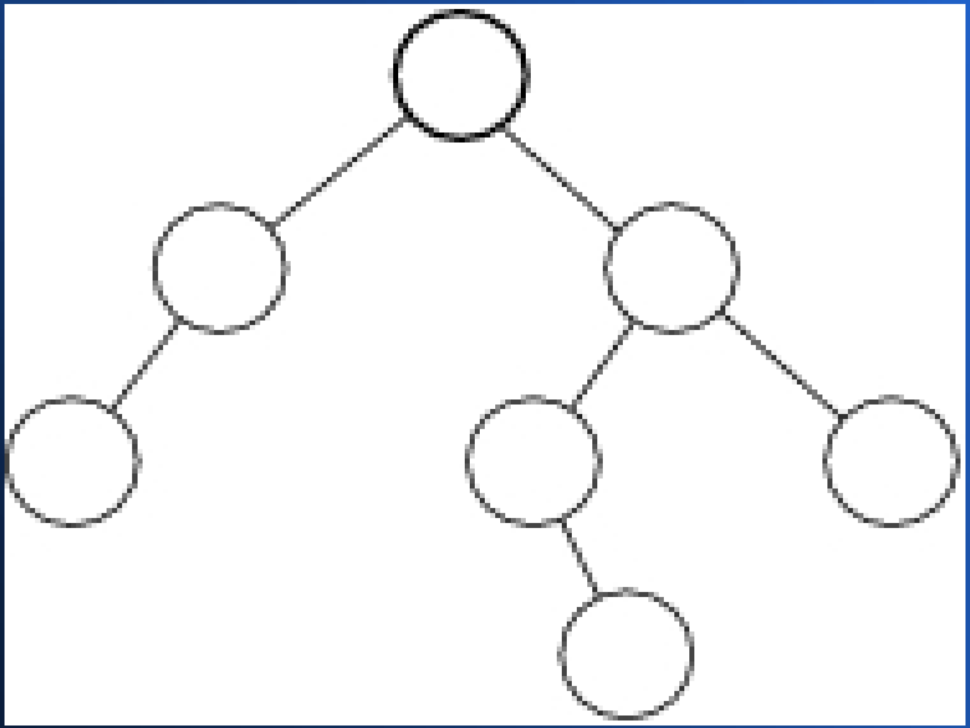
th t

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

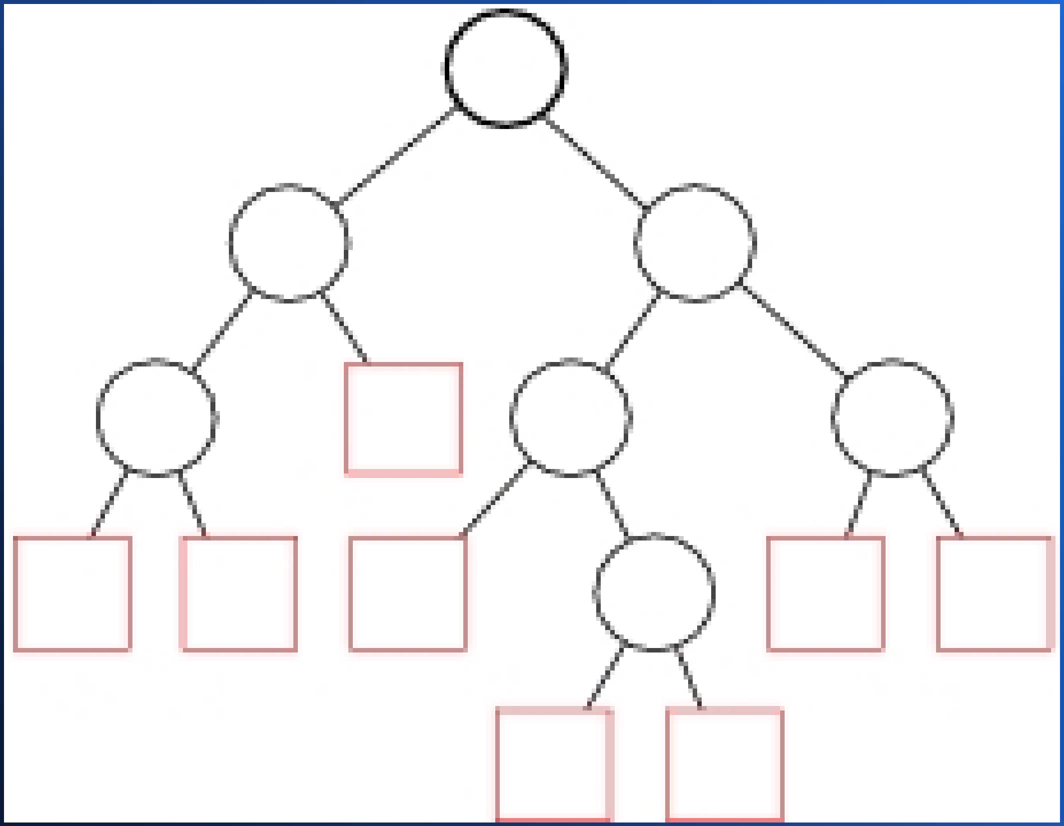


Performance of BS Trees

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 ... ees

E, I: External path length and internal path length

$$E = I + 2n$$

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

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

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 ... ees

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

Avg. case

$I \approx 1.4\, n \lg n - 2.8\, n$

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

$C \approx 1.4 \lg n - 1.8$

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

Worst case

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

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

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

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

Perf… ees

-1- Notes to Standish p. 278

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

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

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

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

$$E_n = I_n + 2n =$$

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

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

Perf … ees

- 7 - Notes to Standish p 278

So,

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

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

Also,

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

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

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

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

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

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

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

Best case $\lfloor \lg n \rfloor - 1 \lesssim C_n \lesssim \lfloor \lg n \rfloor$

Perf… …ees

−3− Notes to Standish p278

$$C'_n = \frac{E_n}{n+1}$$

$$\lfloor \lg n \rfloor + \frac{2}{n+1} \le C'_n <$$

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

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

best case $\lfloor \lg n \rfloor \lesssim C'_n \lesssim \lfloor \lg n \rfloor + 1$

To be continued ...

in Lecture Notes ...