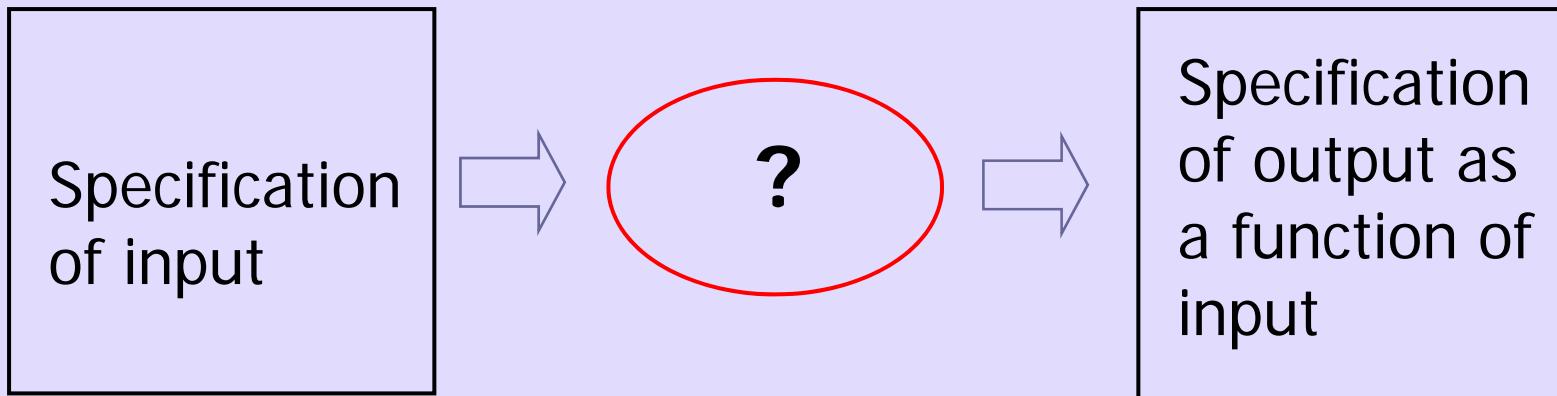


Data Structures and Algorithms

- Algorithm: Outline, the essence of a computational procedure, step-by-step instructions
- Program: an implementation of an algorithm in some programming language
- Data structure: **Organization** of data needed to solve the problem

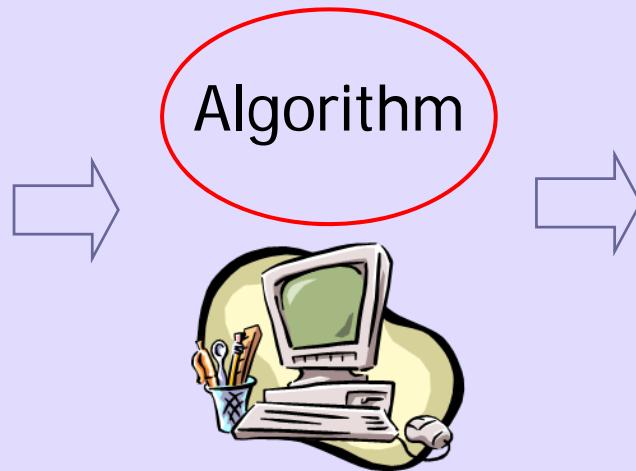
Algorithmic problem



- Infinite number of input *instances* satisfying the specification. For eg: A sorted, non-decreasing sequence of natural numbers of non-zero, finite length:
 - 1, 20, 908, 909, 100000, 100000000.
 - 3.

Algorithmic Solution

Input instance,
adhering to
the
specification



Output
related to
the input as
required

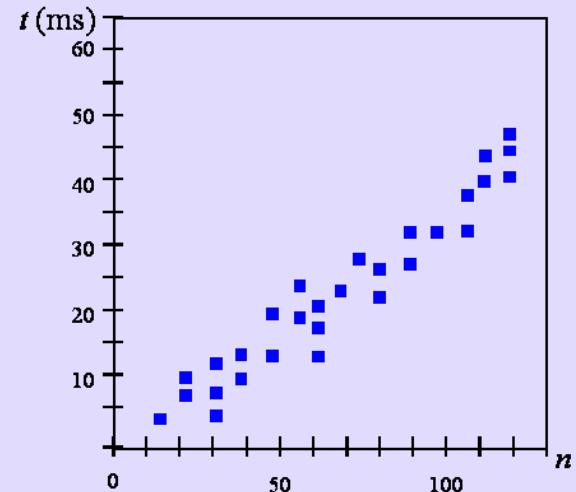
- Algorithm describes actions on the input instance
- Infinitely many correct algorithms for the same algorithmic problem

What is a Good Algorithm?

- Efficient:
 - Running time
 - Space used
- Efficiency as a function of input size:
 - The number of bits in an input number
 - Number of data elements (numbers, points)

Measuring the Running Time

How should we measure the running time of an **algorithm**?



Experimental Study

- Write a **program** that implements the algorithm
- Run the program with data sets of varying size and composition.
- Use a method like **System.currentTimeMillis()** to get an accurate measure of the actual running time.

Limitations of Experimental Studies

- It is necessary to **implement** and test the algorithm in order to determine its running time.
- Experiments can be done only on a **limited set of inputs**, and may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same **hardware and software environments** should be used.

Beyond Experimental Studies

We will develop a general methodology for analyzing running time of algorithms. This approach

- Uses a high-level description of the algorithm instead of testing one of its implementations.
- Takes into account all possible inputs.
- Allows one to evaluate the efficiency of any algorithm in a way that is independent of the hardware and software environment.

Pseudo-Code

- A mixture of natural language and high-level programming concepts that describes the main ideas behind a generic implementation of a data structure or algorithm.
- Eg: **Algorithm arrayMax(A, n):**

Input: An array A storing n integers.

Output: The maximum element in A.

currentMax \leftarrow A[0]

for i \leftarrow 1 **to** n-1 **do**

if currentMax < A[i] **then** currentMax \leftarrow A[i]

return currentMax

Pseudo-Code

It is more structured than usual prose but less formal than a programming language

- Expressions:
 - use standard mathematical symbols to describe numeric and boolean expressions
 - use \leftarrow for assignment (“=” in Java)
 - use = for the equality relationship (“==” in Java)
- Method Declarations:
 - **Algorithm** name(param1, param2)

Pseudo Code

- Programming Constructs:
 - decision structures: **if ... then ... [else ...]**
 - while-loops: **while ... do**
 - repeat-loops: **repeat ... until ...**
 - for-loop: **for ... do**
 - array indexing: **A[i], A[i,j]**
- Methods:
 - calls: object method(args)
 - returns: **return value**

Analysis of Algorithms

- **Primitive Operation:** Low-level operation independent of programming language.
Can be identified in pseudo-code. For eg:
 - Data movement (assign)
 - Control (branch, subroutine call, return)
 - arithmetic and logical operations (e.g. addition, comparison)
- By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm.

Example: Sorting

INPUT

sequence of numbers

$a_1, a_2, a_3, \dots, a_n$

2 5 4 10 7



OUTPUT

a permutation of the sequence of numbers

$b_1, b_2, b_3, \dots, b_n$

2 4 5 7 10

Correctness (requirements for the output)

For any given input the algorithm halts with the output:

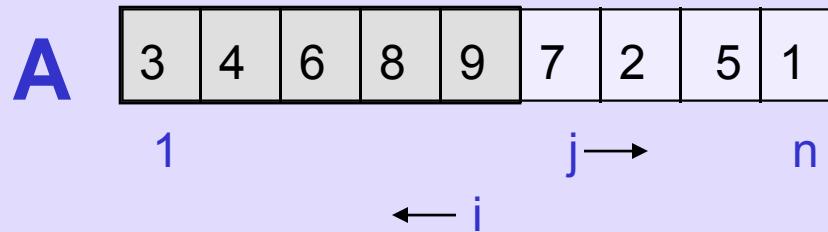
- $b_1 < b_2 < b_3 < \dots < b_n$
- $b_1, b_2, b_3, \dots, b_n$ is a permutation of $a_1, a_2, a_3, \dots, a_n$

Running time

Depends on

- number of elements (n)
- how (partially) sorted they are
- algorithm

Insertion Sort



Strategy

- Start “empty handed”
- Insert a card in the right position of the already sorted hand
- Continue until all cards are inserted/sorted

INPUT: $A[1..n]$ – an array of integers
OUTPUT: a permutation of A such that $A[1] \leq A[2] \leq \dots \leq A[n]$

```
for j←2 to n do
    key ← A[j]
    Insert A[j] into the sorted sequence
    A[1..j-1]
    i←j-1
    while i>0 and A[i]>key
        do A[i+1]←A[i]
            i--
    A[i+1]←key
```

Analysis of Insertion Sort

	cost	times
for $j \leftarrow 2$ to n do	c_1	n
key $\leftarrow A[j]$	c_2	$n-1$
Insert $A[j]$ into the sorted sequence $A[1..j-1]$	0	$n-1$
$i \leftarrow j-1$	c_3	$n-1$
while $i > 0$ and $A[i] > key$	c_4	$\sum_{j=2}^n t_j$
do $A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^n (t_j - 1)$
$i--$	c_6	$\sum_{j=2}^n (t_j - 1)$
$A[i+1] \leftarrow key$	c_7	$n-1$

$$\begin{aligned} \text{Total time} = & n(c_1 + c_2 + c_3 + c_7) + \sum_{j=2}^n t_j (c_4 + c_5 + c_6) \\ & - (c_2 + c_3 + c_5 + c_6 + c_7) \end{aligned}$$

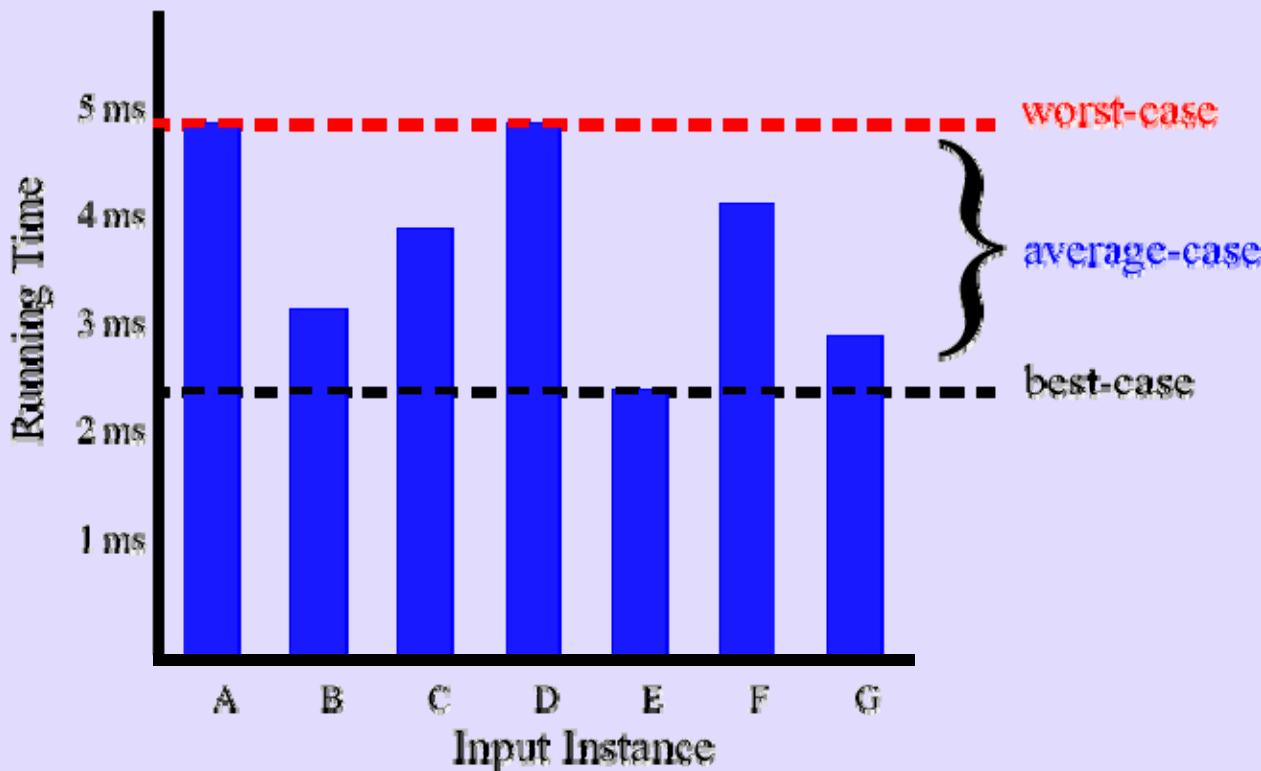
Best/Worst/Average Case

$$\begin{aligned}\text{Total time} = & n(c_1 + c_2 + c_3 + c_7) + \sum_{j=2}^n t_j (c_4 + c_5 + c_6) \\ & - (c_2 + c_3 + c_5 + c_6 + c_7)\end{aligned}$$

- **Best case:** elements already sorted; $t_j=1$, running time = $f(n)$, i.e., *linear* time.
- **Worst case:** elements are sorted in inverse order; $t_j=j$, running time = $f(n^2)$, i.e., *quadratic* time
- **Average case:** $t_j=j/2$, running time = $f(n^2)$, i.e., *quadratic* time

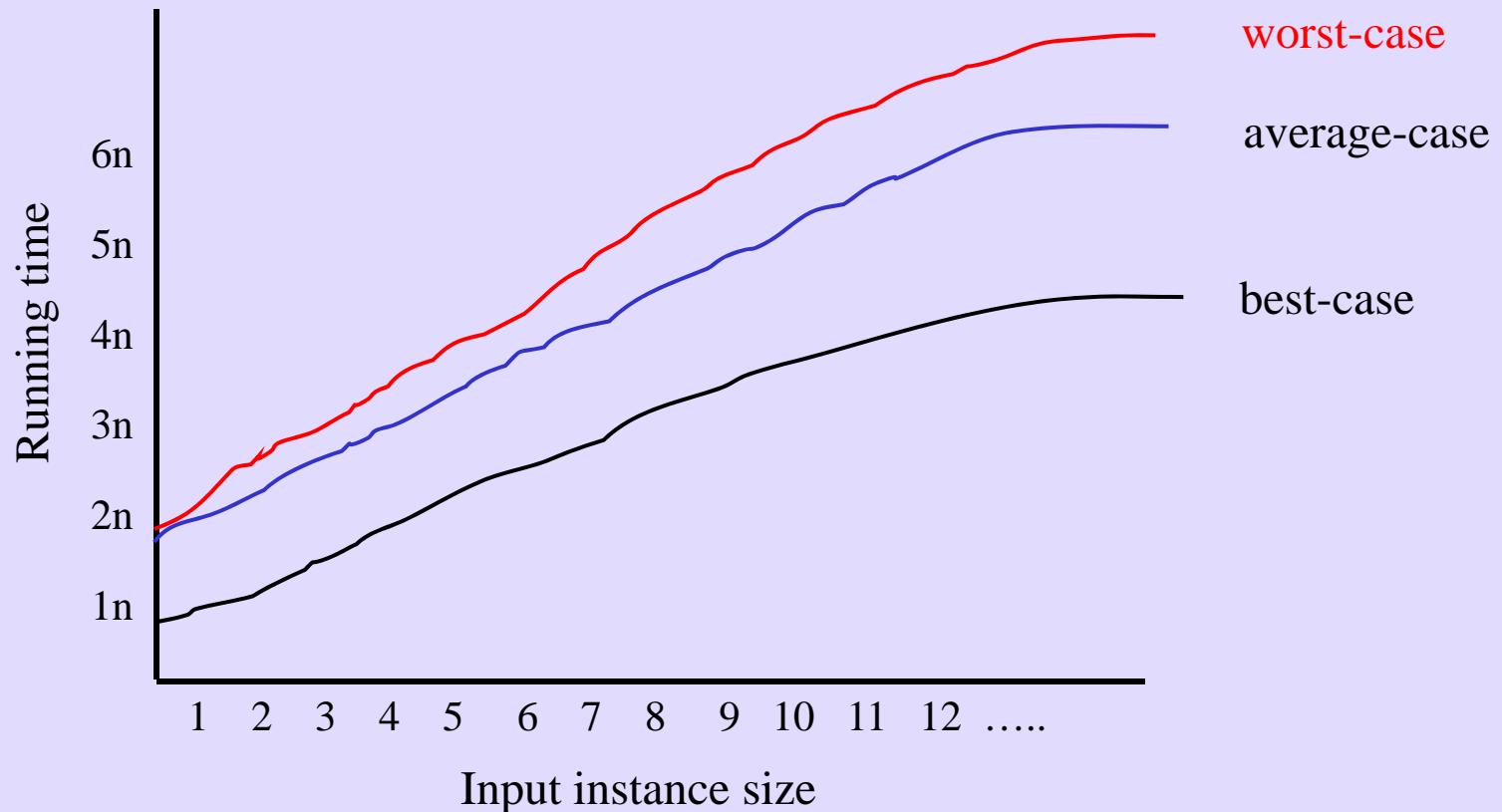
Best/Worst/Average Case (2)

- For a specific size of input n , investigate running times for different input instances:



Best/Worst/Average Case (3)

For inputs of all sizes:



Best/Worst/Average Case (4)

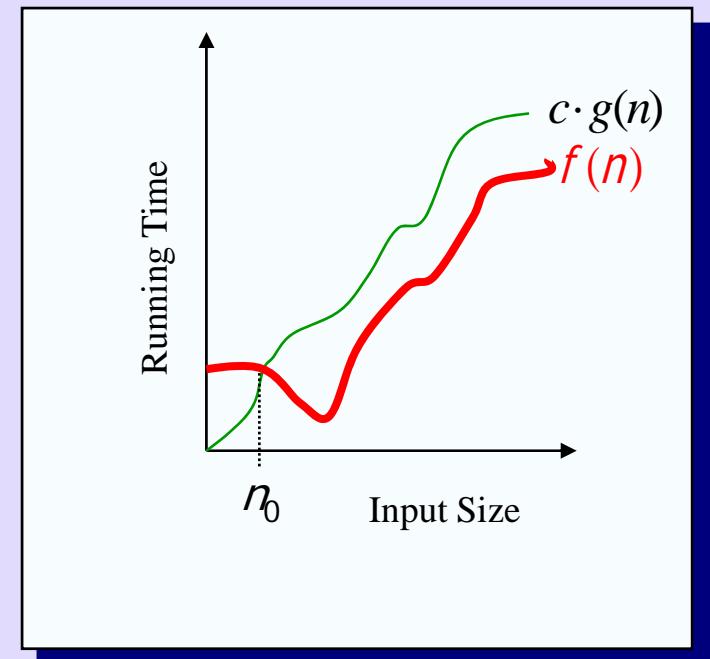
- **Worst case** is usually used: It is an upper-bound and in certain application domains (e.g., air traffic control, surgery) knowing the **worst-case** time complexity is of crucial importance
- For some algorithms **worst case** occurs fairly often
- **Average case** is often as bad as the **worst case**
- Finding **average case** can be very difficult

Asymptotic Analysis

- Goal: to simplify analysis of running time by getting rid of "details", which may be affected by specific implementation and hardware
 - like "rounding": $1,000,001 \approx 1,000,000$
 - $3n^2 \approx n^2$
- Capturing the essence: how the running time of an algorithm increases with the size of the input *in the limit.*
 - Asymptotically more efficient algorithms are best for all but small inputs

Asymptotic Notation

- The “big-Oh” O -Notation
 - asymptotic upper bound
 - $f(n)$ is $O(g(n))$, if there exists constants c and n_0 , s.t. $f(n) \leq c g(n)$ for $n \geq n_0$
 - $f(n)$ and $g(n)$ are functions over non-negative integers
- Used for *worst-case* analysis

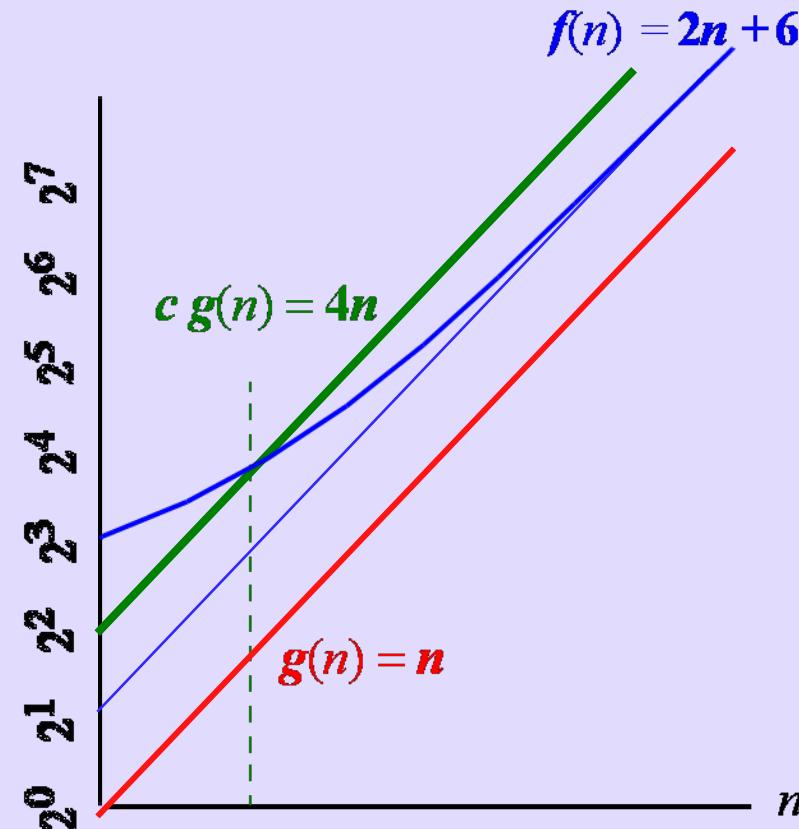


Example

For functions $f(n)$ and $g(n)$ there are positive constants c and n_0 such that: $f(n) \leq c g(n)$ for $n \geq n_0$

conclusion:

$2n+6$ is $O(n)$.



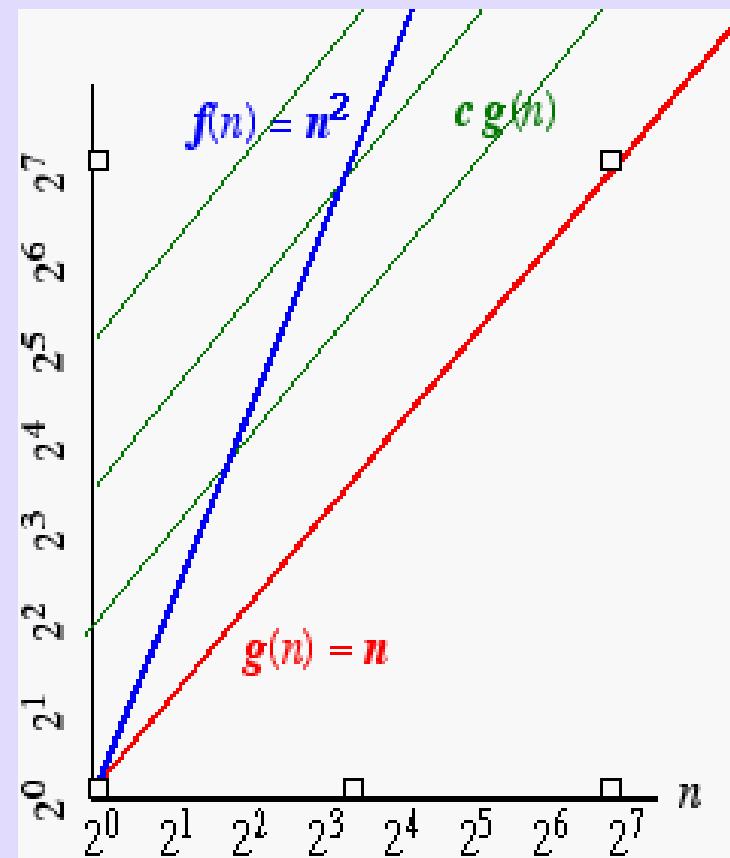
Another Example

On the other hand...

n^2 is not $O(n)$ because there is no c and n_0 such that:

$$n^2 \leq cn \text{ for } n \geq n_0$$

The graph to the right illustrates that no matter how large a c is chosen there is an n big enough that $n^2 > cn$).



Asymptotic Notation

- Simple Rule: Drop lower order terms and constant factors.
 - $50 n \log n$ is $O(n \log n)$
 - $7n - 3$ is $O(n)$
 - $8n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$
- Note: Even though $(50 n \log n)$ is $O(n^5)$, it is expected that such an approximation be of as small an order as possible

Asymptotic Analysis of Running Time

- Use O -notation to express number of primitive operations executed as function of input size.
- Comparing asymptotic running times
 - an algorithm that runs in $O(n)$ time is better than one that runs in $O(n^2)$ time
 - similarly, $O(\log n)$ is better than $O(n)$
 - hierarchy of functions: $\log n < n < n^2 < n^3 < 2^n$
- **Caution!** Beware of very large constant factors.
An algorithm running in time $1,000,000 n$ is still $O(n)$ but might be less efficient than one running in time $2n^2$, which is $O(n^2)$

Example of Asymptotic Analysis

Algorithm prefixAverages1(X):

Input: An n -element array X of numbers.

Output: An n -element array A of numbers such that
 $A[i]$ is the average of elements $X[0], \dots, X[i]$.

for $i \leftarrow 0$ **to** $n-1$ **do**

$a \leftarrow 0$

for $j \leftarrow 0$ **to** i **do**

$a \leftarrow a + X[j] \leftarrow 1$

$A[i] \leftarrow a/(i+1)$ step

return array A

i iterations
with
 $i=0,1,2,\dots,n-1$

n iterations

Analysis: running time is $O(n^2)$

A Better Algorithm

Algorithm prefixAverages2(X):

Input: An n -element array X of numbers.

Output: An n -element array A of numbers such that $A[i]$ is the average of elements $X[0], \dots, X[i]$.

$s \leftarrow 0$

for $i \leftarrow 0$ **to** n **do**

$s \leftarrow s + X[i]$

$A[i] \leftarrow s/(i+1)$

return array A

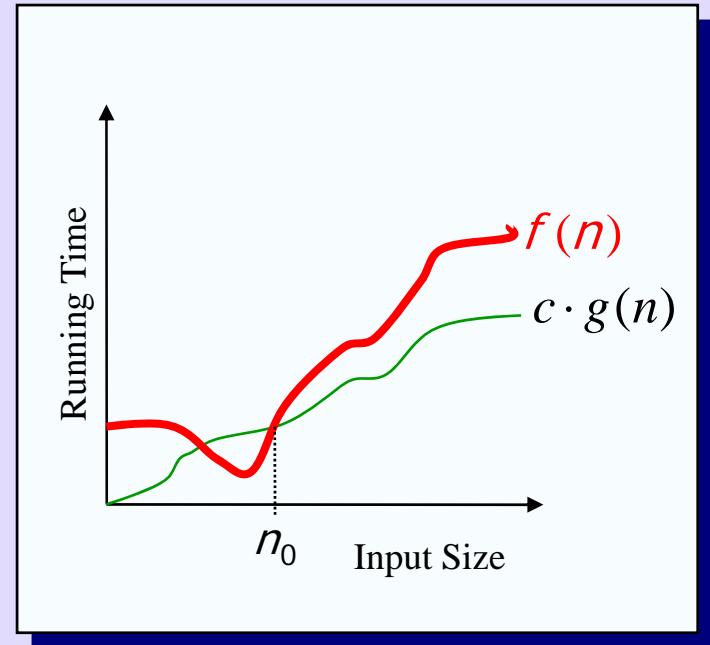
Analysis: Running time is $O(n)$

Asymptotic Notation (*terminology*)

- Special classes of algorithms:
 - Logarithmic: $O(\log n)$
 - Linear: $O(n)$
 - Quadratic: $O(n^2)$
 - Polynomial: $O(n^k)$, $k \geq 1$
 - Exponential: $O(a^n)$, $a > 1$
- “Relatives” of the Big-Oh
 - $\Omega(f(n))$: Big Omega -asymptotic *lower bound*
 - $\Theta(f(n))$: Big Theta -asymptotic *tight bound*

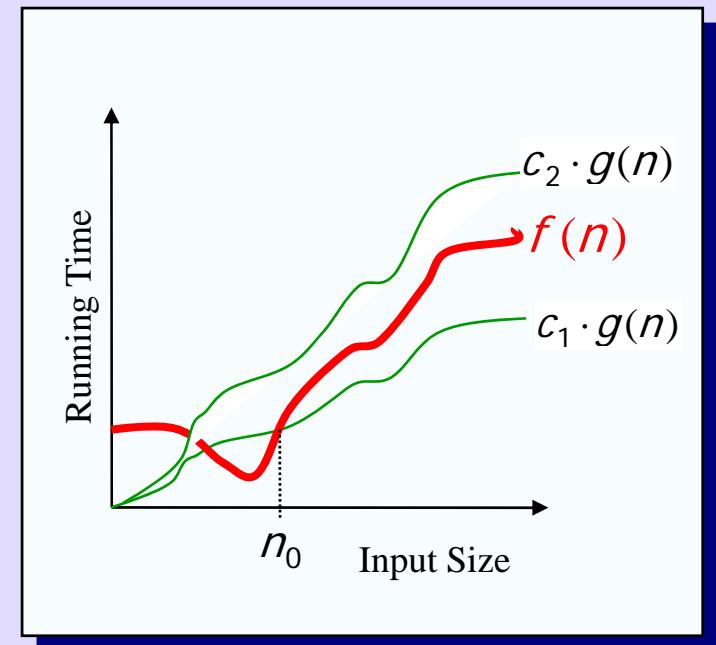
Asymptotic Notation

- The “big-Omega” Ω –Notation
 - asymptotic lower bound
 - $f(n)$ is $\Omega(g(n))$ if there exists constants c and n_0 , s.t.
 $c \cdot g(n) \leq f(n)$ for $n \geq n_0$
- Used to describe *best-case* running times or lower bounds for algorithmic problems
 - E.g., lower-bound for searching in an unsorted array is $\Omega(n)$.



Asymptotic Notation

- The “big-Theta” Θ -Notation
 - asymptotically tight bound
 - $f(n) = \Theta(g(n))$ if there exists constants c_1 , c_2 , and n_0 , s.t.
 $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n \geq n_0$
 - $f(n)$ is $\Theta(g(n))$ if and only if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$
 - $O(f(n))$ is often misused instead of $\Theta(f(n))$



Asymptotic Notation

Two more asymptotic notations

- "Little-Oh" notation $f(n)$ is $o(g(n))$
non-tight analogue of Big-Oh
 - For every c , there should exist n_0 , s.t. $f(n) \leq c g(n)$ for $n \geq n_0$
 - Used for **comparisons** of running times.
If $f(n)=o(g(n))$, it is said that $g(n)$ *dominates* $f(n)$.
- "Little-omega" notation $f(n)$ is $\omega(g(n))$
non-tight analogue of Big-Omega

Asymptotic Notation

- Analogy with real numbers

- $f(n) = O(g(n)) \quad \approx \quad f \leq g$

- $f(n) = \Omega(g(n)) \quad \approx \quad f \geq g$

- $f(n) = \Theta(g(n)) \quad \approx \quad f = g$

- $f(n) = o(g(n)) \quad \approx \quad f < g$

- $f(n) = \omega(g(n)) \quad \approx \quad f > g$

- Abuse of notation: $f(n) = O(g(n))$ actually means $f(n) \in O(g(n))$

Comparison of Running Times

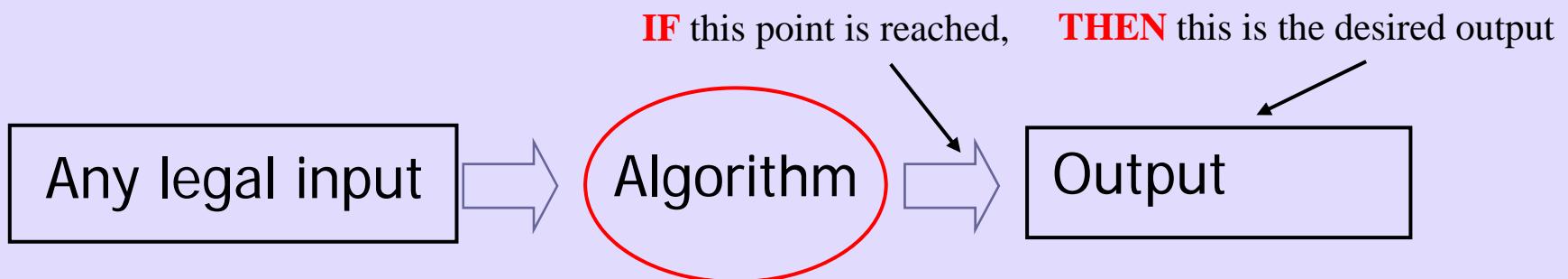
Running Time	Maximum problem size (n)		
	1 second	1 minute	1 hour
$400n$	2500	150000	9000000
$20n \log n$	4096	166666	7826087
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

Correctness of Algorithms

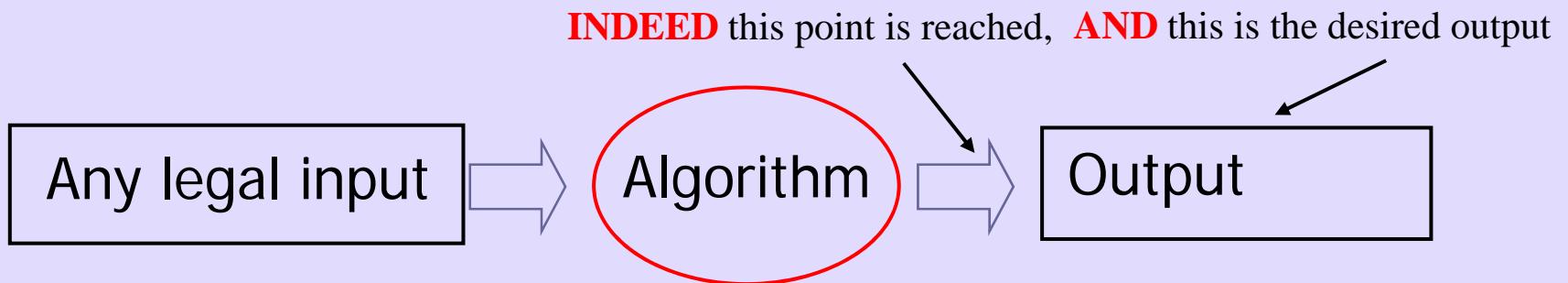
- The algorithm is *correct* if for any legal input it terminates and produces the desired output.
- Automatic proof of correctness is not possible
- But there are practical techniques and rigorous formalisms that help to reason about the correctness of algorithms

Partial and Total Correctness

□ Partial correctness



□ Total correctness



Assertions

- To prove correctness we associate a number of **assertions** (statements about the state of the execution) with specific checkpoints in the algorithm.
 - E.g., $A[1], \dots, A[k]$ form an increasing sequence
- **Preconditions** – assertions that must be valid *before* the execution of an algorithm or a subroutine
- **Postconditions** – assertions that must be valid *after* the execution of an algorithm or a subroutine

Loop Invariants

- **Invariants** – assertions that are valid any time they are reached (many times during the execution of an algorithm, e.g., in loops)
- We must show three things about loop invariants:
 - **Initialization** – it is true prior to the first iteration
 - **Maintenance** – if it is true before an iteration, it remains true before the next iteration
 - **Termination** – when loop terminates the invariant gives a useful property to show the correctness of the algorithm

Example of Loop Invariants (1)

- **Invariant:** *at the start of each **for** loop, $A[1\dots j-1]$ consists of elements originally in $A[1\dots j-1]$ but in sorted order*

```
for j ← 2 to length(A)
    do key ← A[j]
        i ← j-1
        while i>0 and A[i]>key
            do A[i+1] ← A[i]
                i--
        A[i+1] ← key
```

Example of Loop Invariants (2)

- **Invariant:** at the start of each **for** loop, $A[1\dots j-1]$ consists of elements originally in $A[1\dots j-1]$ but in sorted order

```
for j ← 2 to length(A)
    do key ← A[j]
        i ← j-1
        while i>0 and A[i]>key
            do A[i+1]← A[i]
                i--
        A[i+1] ← key
```

- **Initialization:** $j = 2$, the invariant trivially holds because $A[1]$ is a sorted array ☺

Example of Loop Invariants (3)

- **Invariant:** at the start of each **for** loop, $A[1\dots j-1]$ consists of elements originally in $A[1\dots j-1]$ but in sorted order

```
for j ← 2 to length(A)
    do key ← A[j]
        i ← j-1
        while i>0 and A[i]>key
            do A[i+1] ← A[i]
                i--
        A[i+1] ← key
```

- **Maintenance:** the inner **while** loop moves elements $A[j-1], A[j-2], \dots, A[j-k]$ one position right without changing their order. Then the former $A[j]$ element is inserted into k -th position so that $A[k-1] \leq A[k] \leq A[k+1]$.

$A[1\dots j-1]$ sorted + $A[j]$ \rightarrow $A[1\dots j]$ sorted

Example of Loop Invariants (4)

- **Invariant:** *at the start of each **for** loop, $A[1\dots j-1]$ consists of elements originally in $A[1\dots j-1]$ but in sorted order*

```
for j ← 2 to length(A)
    do key ← A[j]
        i ← j-1
        while i>0 and A[i]>key
            do A[i+1] ← A[i]
                i--
        A[i+1] ← key
```

- **Termination:** the loop terminates, when $j=n+1$. Then the invariant states: " $A[1\dots n]$ consists of elements originally in $A[1\dots n]$ but in sorted order" ☺

Math You Need to Review

□ Properties of logarithms:

$$\log_b(xy) = \log_b x + \log_b y$$

$$\log_b(x/y) = \log_b x - \log_b y$$

$$\log_b x^a = a \log_b x$$

$$\log_b a = \log_x a / \log_x b$$

□ Properties of exponentials:

$$a^{(b+c)} = a^b a^c ; a^{bc} = (a^b)^c$$

$$a^b / a^c = a^{(b-c)} ; b = a^{\log_a b}$$

□ Floor: $\lfloor x \rfloor$ = the largest integer $\leq x$

□ Ceiling: $\lceil x \rceil$ = the smallest integer $\geq x$

Math Review

□ Geometric progression

□ given an integer n_0 and a real number $0 < a \neq 1$

$$\sum_{i=0}^n a^i = 1 + a + a^2 + \dots + a^n = \frac{1 - a^{n+1}}{1 - a}$$

□ geometric progressions exhibit exponential growth

□ Arithmetic progression

$$\sum_{i=0}^n i = 1 + 2 + 3 + \dots + n = \frac{n^2 + n}{2}$$

Summations

- The running time of insertion sort is determined by a nested loop

```
for j←2 to length(A)
    key←A[j]
    i←j-1
    while i>0 and A[i]>key
        A[i+1]←A[i]
        i←i-1
    A[i+1]←key
```

- Nested loops correspond to summations

$$\sum_{j=2}^n (j-1) = O(n^2)$$

Proof by Induction

- We want to show that property P is true for all integers $n \geq n_0$
- **Basis:** prove that P is true for n_0
- **Inductive step:** prove that if P is true for all k such that $n_0 \leq k \leq n - 1$ then P is also true for n
- Example $S(n) = \sum_{i=0}^n i = \frac{n(n+1)}{2}$ for $n \geq 1$
- Basis $S(1) = \sum_{i=0}^1 i = \frac{1(1+1)}{2}$

Proof by Induction (2)

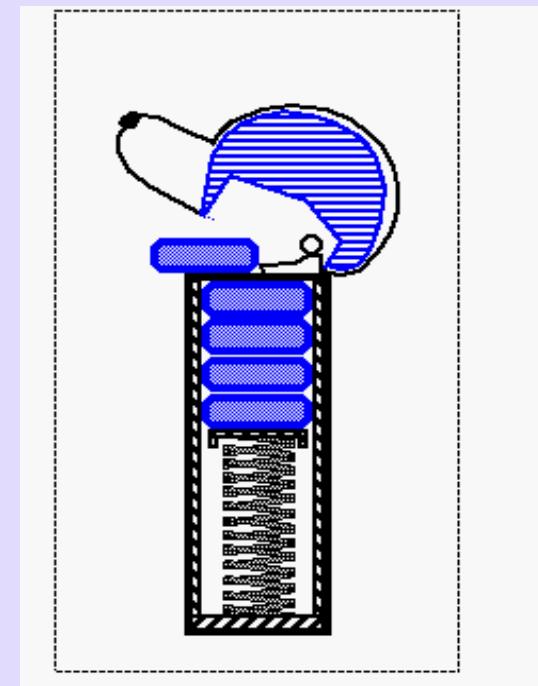
□ Inductive Step

$$S(k) = \sum_{i=0}^k i = \frac{k(k+1)}{2} \text{ for } 1 \leq k \leq n-1$$

$$\begin{aligned} S(n) &= \sum_{i=0}^n i = \sum_{i=0}^{n-1} i + n = S(n-1) + n = \\ &= (n-1) \frac{(n-1+1)}{2} + n = \frac{(n^2 - n + 2n)}{2} = \\ &= \frac{n(n+1)}{2} \end{aligned}$$

Stacks

- Abstract Data Types (ADTs)
- Stacks
- Interfaces and exceptions
- Java implementation of a stack
- Application to the analysis of a time series
- Growable stacks
- Stacks in the Java virtual machine



Abstract Data Types (ADTs)

- *ADT* is a mathematically specified entity that defines a set of its *instances*, with:
 - a specific *interface* – a collection of signatures of operations that can be invoked on an instance,
 - a set of *axioms (preconditions and postconditions)* that define the semantics of the operations (i.e., what the operations do to instances of the ADT, but not how)

Abstract Data Types (ADTs)

Types of operations:

- Constructors
- Access functions
- Manipulation procedures

Abstract Data Types

- Why do we need to talk about ADTs in a DS course?
 - They serve as *specifications* of *requirements* for the building blocks of solutions to algorithmic problems
 - Provides a language to talk on a higher level of abstraction
 - ADTs encapsulate *data structures* and algorithms that *implement* them
 - Separate the issues of *correctness* and *efficiency*

Example - Dynamic Sets

- We will deal with ADTs, instances of which are sets of some type of elements.
 - Operations are provided that change the set
- We call such class of ADTs *dynamic sets*

Dynamic Sets (2)

■ An example dynamic set ADT

- Methods:

- **New():ADT**
- **Insert(S:ADT, v:element):ADT**
- **Delete(S:ADT, v:element):ADT**
- **IsIn(S:ADT, v:element):boolean**

- **Insert** and **Delete** – *manipulation* operations
- **IsIn** – *Access method* method

Dynamic Sets (3)

■ Axioms that define the methods:

- **IsIn(New(), v) = false**
- **IsIn(Insert(S, v), v) = true**
- **IsIn(Insert(S, u), v) = IsIn(S, v), if v ≠ u**
- **IsIn>Delete(S, v), v) = false**
- **IsIn>Delete(S, u), v) = IsIn(S, v), if v ≠ u**

Other Examples

- Simple ADTs:

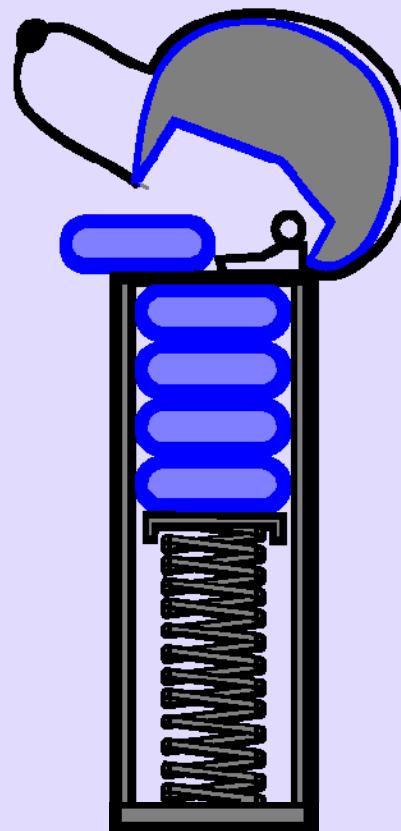
- *Queue*
 - *Deque*
 - *Stack*

Stacks

- A **stack** is a container of objects that are inserted and removed according to the last-in-first-out (**LIFO**) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as “**pushing**” onto the stack. “**Popping**” off the stack is synonymous with removing an item.

Stacks (2)

- A PEZ ® dispenser as an analogy:



Stacks(3)

- A stack is an ADT that supports four main methods:
 - **new():ADT** – Creates a new stack
 - **push(S:ADT, o:element):ADT** - Inserts object *o* onto top of stack *S*
 - **pop(S:ADT):ADT** - Removes the top object of stack *S*; if the stack is empty an error occurs
 - **top(S:ADT):element** – Returns the top object of the stack, without removing it; if the stack is empty an error occurs

Stacks(4)

- The following support methods could also be defined:
 - **size(S:ADT):integer** - Returns the number of objects in stack S
 - **isEmpty(S:ADT): boolean** - Indicates if stack S is empty
- Axioms
 - **Pop(Push(S, v)) = S**
 - **Top(Push(S, v)) = v**

Java Stuff

- Given the stack ADT, we need to code the ADT in order to use it in the programs. We need two constructs: **interfaces** and **exceptions**.
- An **interface** is a way to declare what a class is to do. It does not mention how to do it.
 - For an **interface**, we just write down the method names and the **parameters**. When specifying **parameters**, what really matters is their **types**.
 - Later, when we write a **class** for that **interface**, we actually code the content of the methods.
 - Separating **interface** and **implementation** is a useful programming technique.

A Stack Interface in Java

The stack data structure is a “built-in” class of Java’s `java.util` package. But we define our own stack interface:

```
public interface Stack {  
  
    // accessor methods  
    public int size();  
    public boolean isEmpty();  
    public Object top()    throws StackEmptyException;  
  
    // update methods  
    public void push (Object element);  
    public Object pop()   throws StackEmptyException;  
}
```

Exceptions

- Exceptions are yet another programming construct, useful for handling errors. When we find an error (or an ***exceptional*** case), we just ***throw*** an exception.
- As soon as the exception is thrown, the flow of control exits from the current method and goes to where that method was called from.
- **What is the point of using exceptions?** We can delegate upwards the responsibility of handling an error. Delegating upwards means letting the code who called the current code deal with the problem.

More Exceptions

```
public void eatPizza() throws  
StomachAcheException  
{...  
if (ateTooMuch)  
throw new  
StomachAcheException("Ouch");  
...}  
  
private void simulateMeeting()  
{...  
try  
{TA.eatPizza();}  
catch (StomachAcheException e)  
{system.out.println("somebody has a stomach ache");}  
...}
```

So when
`StomachAcheException`
is thrown, we exit from
method `eatPizza()` and
go to `TA.eatpizza()`.

Even More Exceptions

- The try block and the **catch** block means that we are listening for exceptions that are specified in the catch parameter.
- Because **catch** is listening for **StomachAcheException**, the flow of control will now go to the catch block. And **System.out.println** will get executed.
- Note that a **catch** block can contain anything. It does not have to do only **System.out.println**. We can handle the caught error in any way you like; we can even **throw** them again.

Exceptions (finally)

- Note that if somewhere in your method, you throw an exception, you need to add a **throws** clause next to your method name.
- If you never catch an exception, it will propagate upwards and upwards along the chain of method calls until the user sees it.
- What exactly are exceptions in Java? Classes.

```
public class StomachAcheException extends  
RuntimeException {  
    public StomachAcheException (String err)  
        {super(err);}  
}
```

Array-Based Stack in Java

- Create a stack using an array by specifying a maximum size N for our stack.
- The stack consists of an N -element array S and an integer variable t , the index of the top element in array S .



- Array indices start at 0, so we initialize t to -1

Array-Based Stack in Java (2)

```
public class ArrayStack implements Stack {  
    // Implementation of the Stack interface using an array  
  
    public static final int CAPACITY = 1024;  
        // default capacity of stack  
    private int N;  
        // maximum capacity of the stack  
    private Object S[ ];  
        // S holds the elements of the stack  
    private int t = -1;  
        // the top element of the stack  
    public ArrayStack( ) // Initialize the stack with default capacity  
    { this(CAPACITY) }  
    public ArrayStack(int cap)  
        // Initialize the stack with given capacity  
    {N = cap; S = new Object[N]}
```

Array-Based Stack in Java (3)

```
public int size( ) //Return the current stack size  
{return (t + 1)}  
  
public boolean isEmpty( ) //Return true iff the stack is empty  
{return (t < 0)}  
  
public void push(Object obj) throws StackFullException{  
    //Push a new element on the stack  
if (size() == N)  
    throw new StackFullException("Stack overflow.")  
S[++t] = obj;}
```

Array-Based Stack in Java (4)

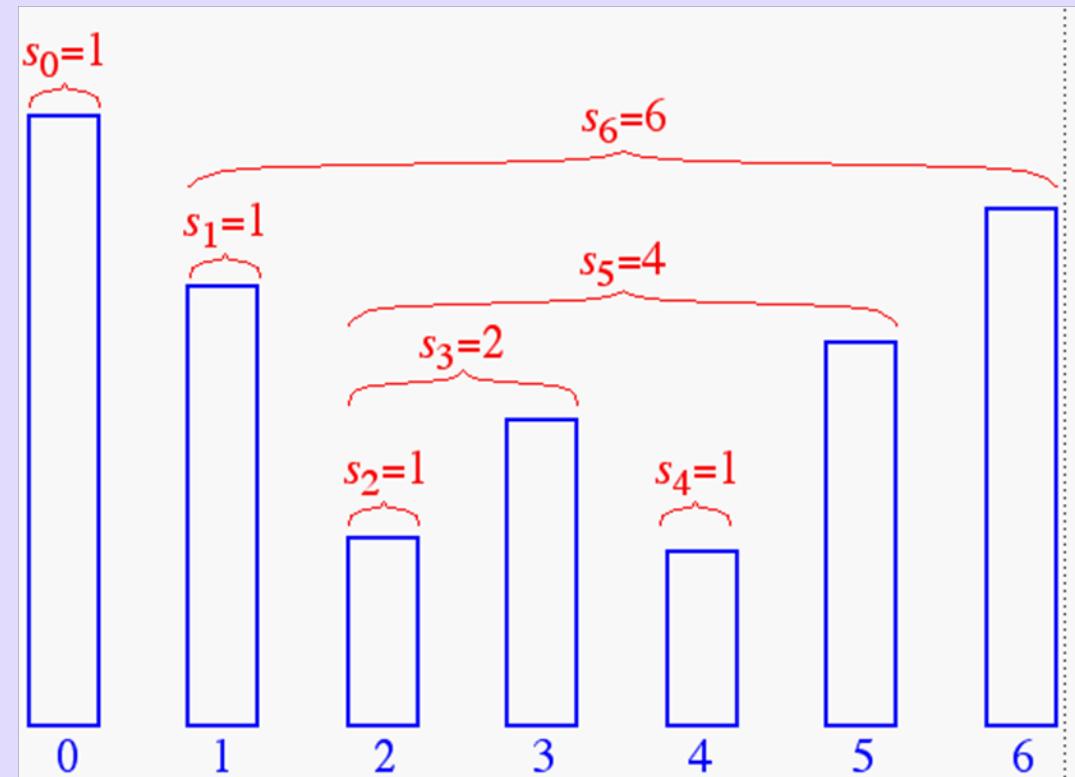
```
public Object top( ) throws StackEmptyException {  
    // Return the top stack element  
    if (isEmpty( ))  
        throw new StackEmptyException("Stack is empty.");  
    return S[t]}  
  
public Object pop() throws StackEmptyException {  
    // Pop off the stack element  
    Object elem;  
    if (isEmpty( ))  
        throw new StackEmptyException("Stack is Empty.");  
    elem = S[t];  
    S[t--] = null;      // Dereference S[top] and decrement top  
    return elem}
```

Array-Based Stack in Java (5)

- The array implementation is simple and efficient (methods performed in $O(1)$).
- There is an upper bound, N , on the size of the stack. The arbitrary value N may be too small for a given application, or a waste of memory.
- **StackEmptyException** is required by the interface.
- **StackFullException** is particular to this implementation.

Application: Time Series

- The **span s_i** of a stock's price on a certain day i is the maximum number of consecutive days (up to the current day) the price of the stock has been less than or equal to its price on day i



An Inefficient Algorithm

Algorithm `computeSpans1(P)`:

Input: an n -element array P of numbers such that $P[i]$ is the price of the stock on day i

Output: an n -element array S of numbers such that $S[i]$ is the span of the stock on day i

for $i \leftarrow 0$ to $n - 1$ do

$k \leftarrow 0$; $\text{done} \leftarrow \text{false}$

 repeat

 if $P[i - k] \leq P[i]$ then $k \leftarrow k + 1$

 else $\text{done} \leftarrow \text{true}$

 until ($k = i$) or done

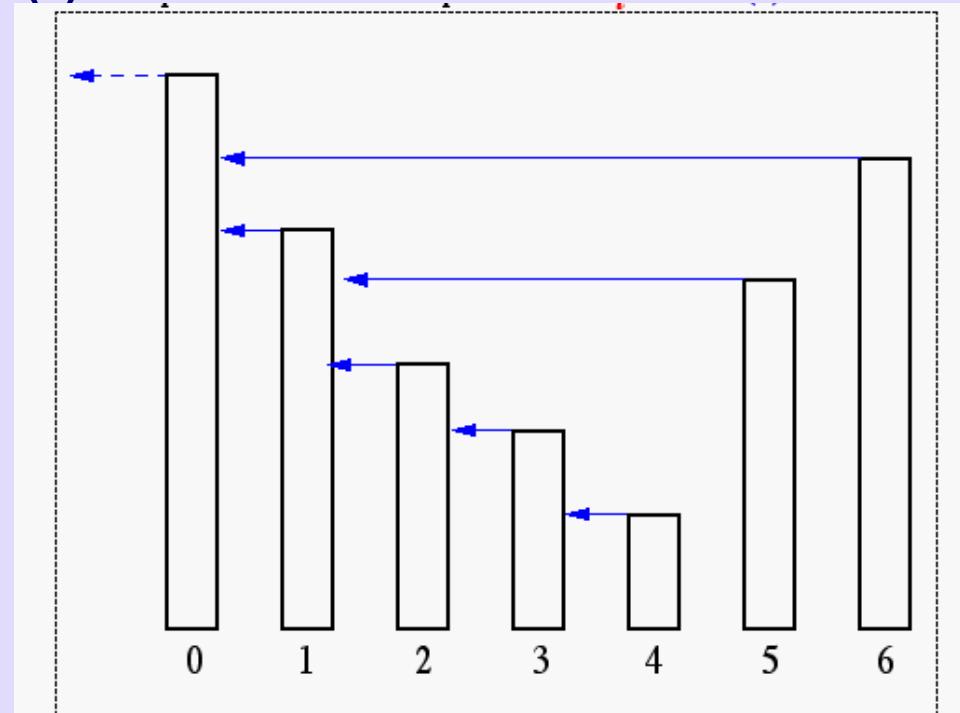
$S[i] \leftarrow k$

return S

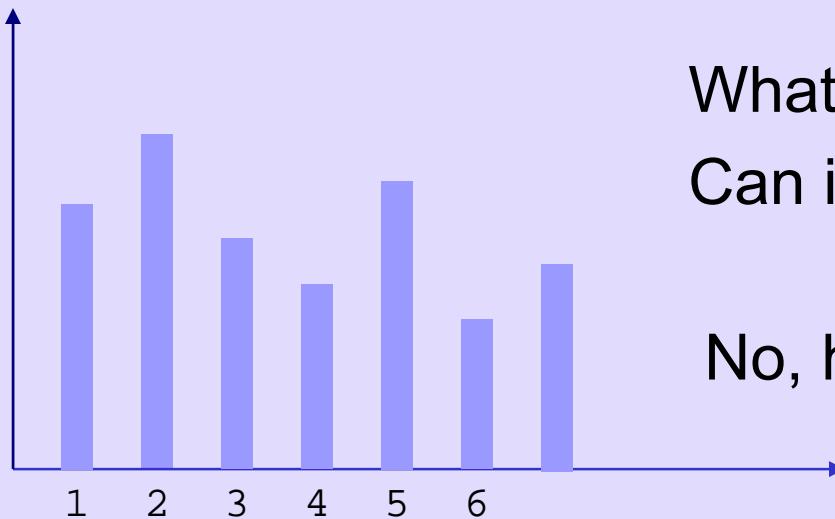
The running time of this algorithm is $O(n^2)$. Why?

A Stack Can Help

- s_i can be easily computed if we know the closest day preceding i , on which the price on day i . If such a day exists, let's call it $h(i)$, otherwise, we conventionally define $h(i) = -1$
- In the figure, $h(3)=2$, $h(5)=1$ and $h(6)=0$.
- The span is now computed as $s_i = i - h(i)$



What to do with the Stack?



What are possible values of $h(7)$?
Can it be 1 or 3 or 4?
No, $h(7)$ can only be 2 or 5 or 6.

- We store indices 2,5,6 in the stack.
- To determine $h(7)$ we compare the price on day 7 with prices on day 6, day 5, day 2 in that order.
- The first price larger than the price on day 7 gives $h(7)$
- The stack should be updated to reflect the price of day 7
- It should now contain 2,5,7

An Efficient Algorithm

Algorithm `computeSpans2(P)`:

Let D be an empty stack

`for i ← 0 to n - 1 do`

`k ← 0; done ← false`

`while not (D.isEmpty() or done) do`

`if P[i] ≥ P[D.top()] then D.pop()`

`else done ← true`

`if D.isEmpty() then h ← -1`

`else h ← D.top()`

`S[i] ← i - h`

`D.push(i)`

`return S`

A Growable Array-Based Stack

- Instead of giving up with a `StackFullException`, we can replace the array `S` with a larger one and continue processing push operations.

Algorithm `push(o)`

`if size() = N then A ← new array of length f(N)`

`for i ← 0 to N – 1`

`A[i] ← S[i]`

`S ← A; t ← t + 1`

`S[t] ← o`

- How large should the new array be?
 - tight strategy** (add a constant): $f(N) = N + c$
 - growth strategy** (double up): $f(N) = 2N$

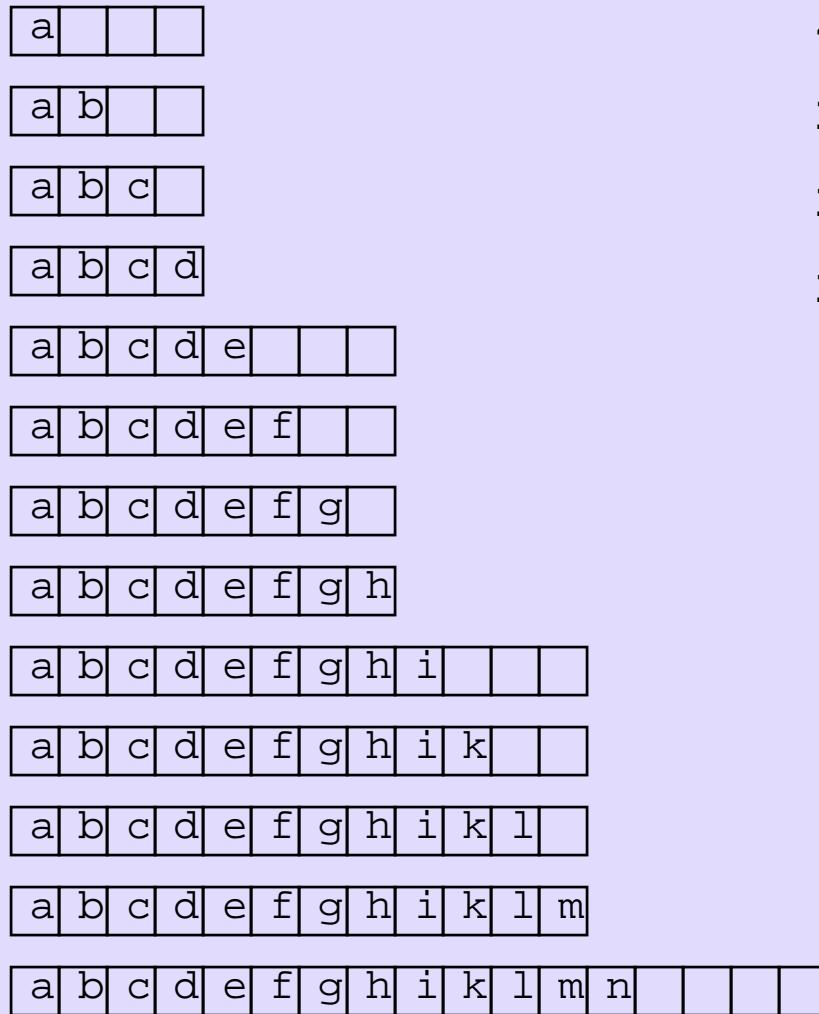
Tight vs. Growth Strategies: *a comparison*

To compare the two strategies, we use the following cost model:

- A *regular push operation*: adds one element and cost one unit.
- A *special push operation*: create an array of size $f(N)$, copy N elements, and add one element. Costs $f(N)+N+1$ units

Tight Strategy ($c=4$)

start with an array of size 0. cost of a special push is $2N + 5$



$4+1$

1

1

1

$8+4+1$

1

1

1

$12+8+1$

1

1

1

Phase 1

Phase 2

Phase 3

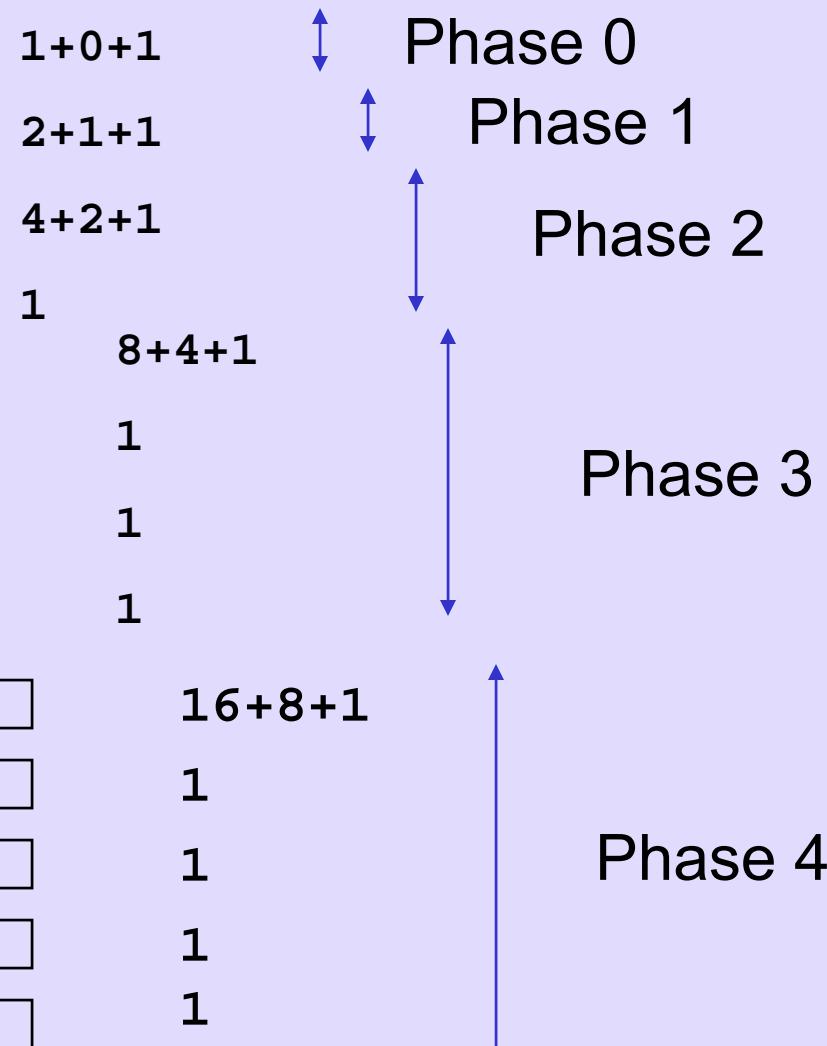
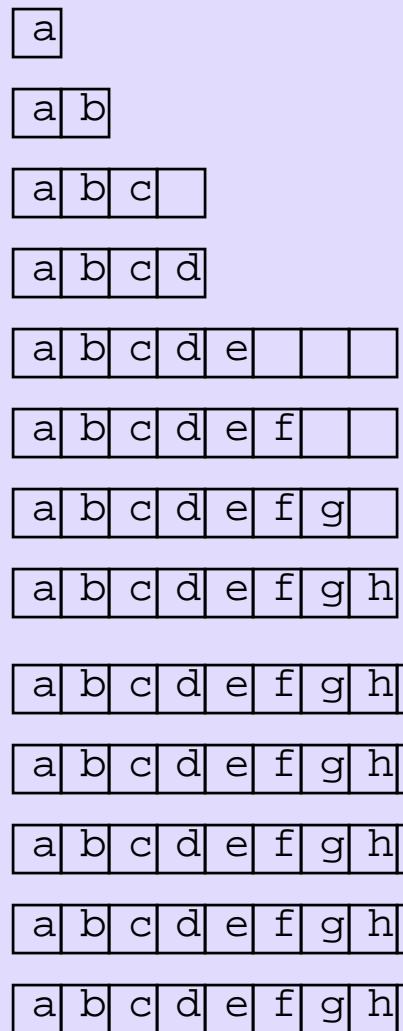
$16+12+1$

Performance of the Tight Strategy

- In phase i the array has size $c \times i$
- Total cost of phase i is
 - $c \times i$ is the cost of creating the array
 - $c \times (i-1)$ is the cost of copying elements into new array
 - c is the cost of the c pushes.
- Hence, cost of phase i is $2ci$
- In each phase we do c pushes. Hence for n pushes we need n/c phases. Total cost of these n/c phases is
$$= 2c (1 + 2 + 3 + \dots + n/c) \approx O(n^2/c)$$

Growth Strategy

start with an array of size 0. cost of a special push is $3N + 1$



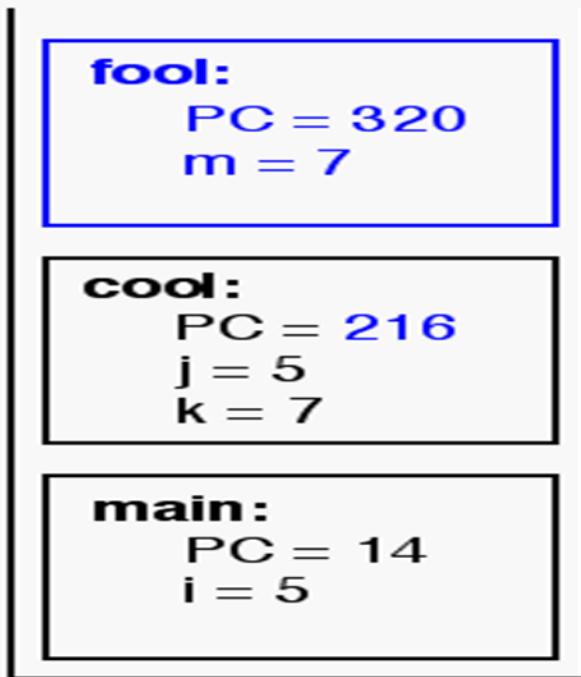
Performance of the Growth Strategy

- In phase i the array has size 2^i
- Total cost of phase i is
 - 2^i is the cost of creating the array
 - 2^{i-1} is the cost of copying elements into new array
 - 2^{i-1} is the cost of the 2^{i-1} pushes done in this phase
- Hence, cost of phase i is 2^{i+1} .
- If we do n pushes, we will have $\log n$ phases.
- Total cost of n pushes
$$= 2 + 4 + 8 + \dots + 2^{\log n + 1} = 4n - 1$$
- The growth strategy wins!

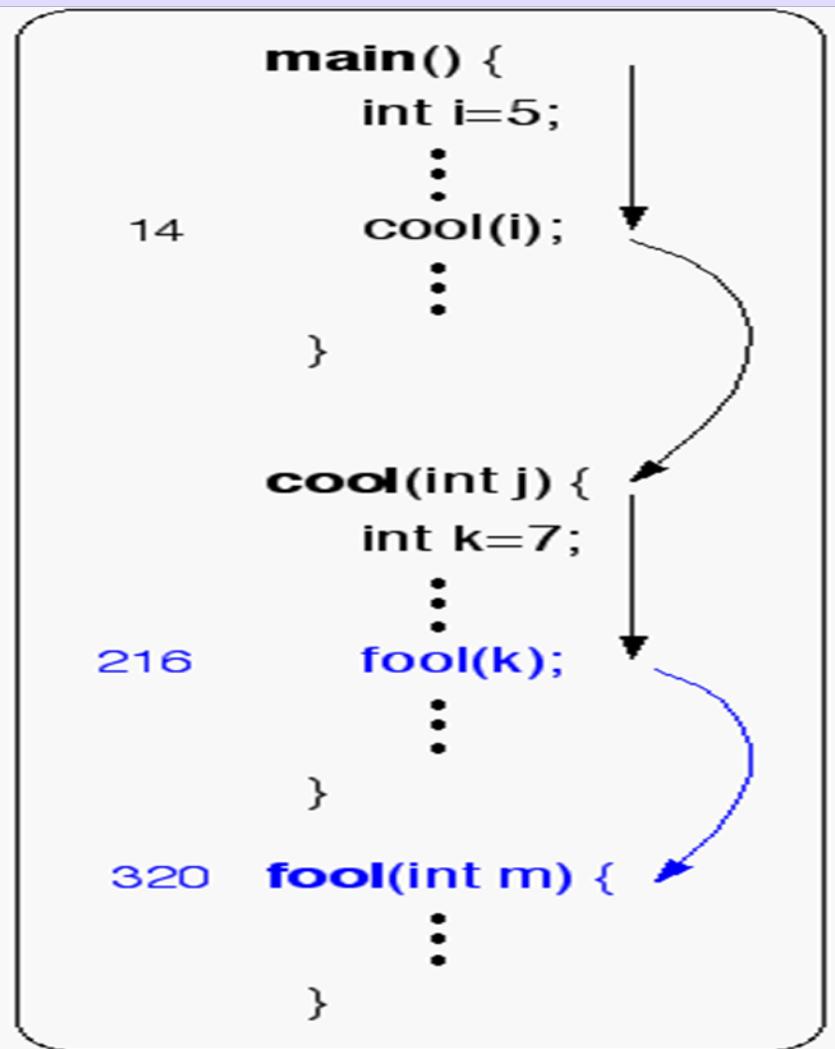
Stacks in the *Java Virtual Machine*

- Each process running in a Java program has its own Java Method Stack.
- Each time a method is called, it is pushed onto the stack.
- The choice of a stack for this operation allows Java to do several useful things:
 - Perform recursive method calls
 - Print stack traces to locate an error

Java Method Stack



Java Stack

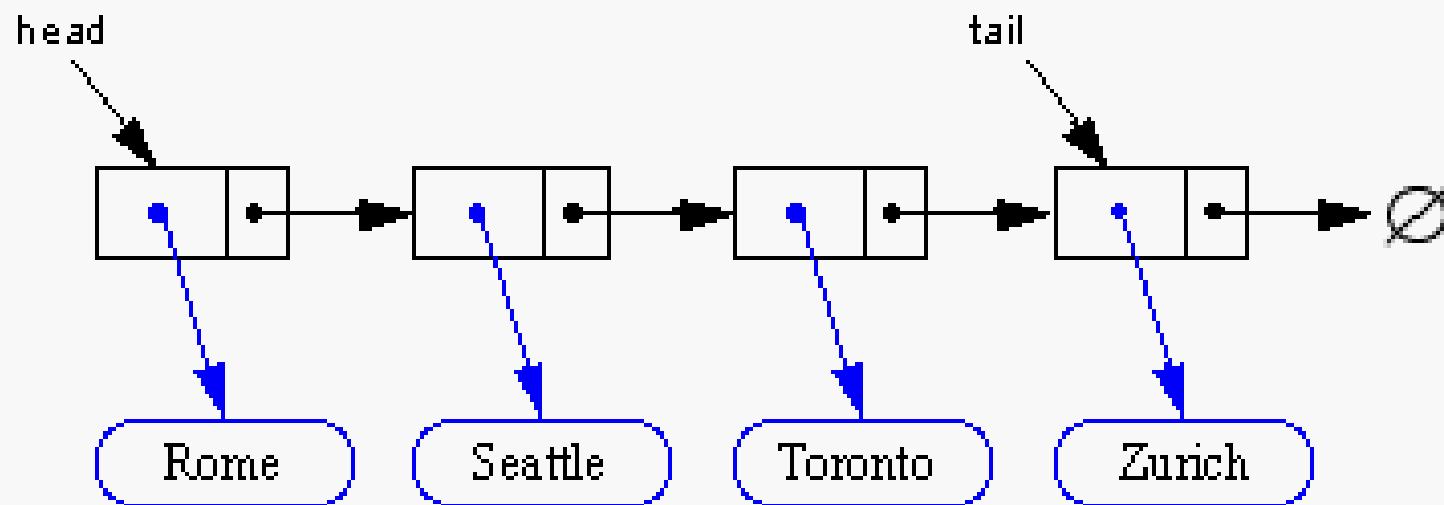


Queues and Linked Lists

Queues

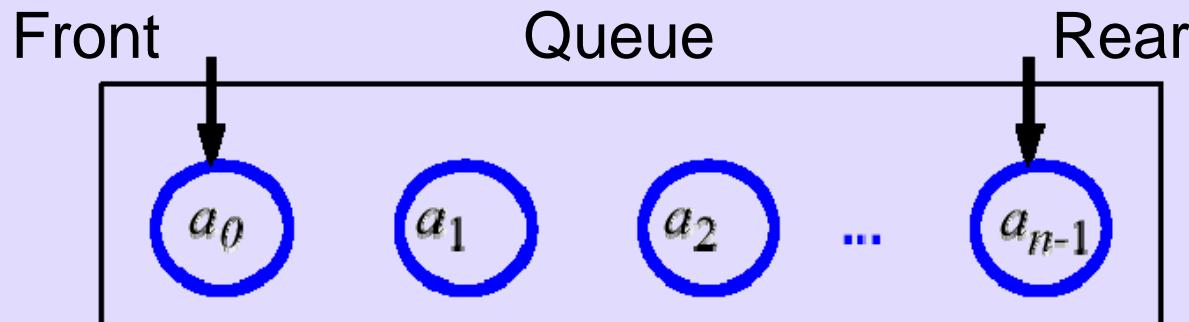
Linked Lists

Double-Ended Queues



Queues

- A queue differs from a stack in that its insertion and removal routines follows the **first-in-first-out** (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the **rear** (enqueued) and removed from the **front** (dequeued)



Queues (2)

- The **queue** supports three fundamental methods:
 - **New():ADT** – Creates an empty queue
 - **Enqueue(S:ADT, o:element):ADT** - Inserts object o at the rear of the queue
 - **Dequeue(S:ADT):ADT** - Removes the object from the front of the queue; an error occurs if the queue is empty
 - **Front(S:ADT):element** - Returns, but does not remove, the front element; an error occurs if the queue is empty

Queues (3)

- These support methods should also be defined:
 - **Size(S:ADT):integer**
 - **IsEmpty(S:ADT):boolean**
- Axioms:
 - **Front(Enqueue(New(), v)) = v**
 - **Dequeue(Enqueue(New(), v)) = New()**
 - **Front(Enqueue(Enqueue(Q, w), v)) = Front(Enqueue(Q, w))**
 - **Dequeue(Enqueue(Enqueue(Q, w), v)) = Enqueue(Dequeue(Enqueue(Q, w)), v)**

An Array Implementation

- Create a queue using an array in a circular fashion
- A maximum size N is specified.
- The queue consists of an N -element array Q and two integer variables:
 - f , index of the front element (head – for dequeue)
 - r , index of the element after the rear one (tail – for enqueue)



An Array Implementation (2)

- “wrapped around” configuration



- what does $f=r$ mean?

An Array Implementation (3)

■ Pseudo code

Algorithm **size()**

return $(N-f+r) \bmod N$

Algorithm **isEmpty()**

return $(f=r)$

Algorithm **front()**

if isEmpty() **then**

return Queueemptyexception

return $Q[f]$

Algorithm **dequeue()**

if isEmpty() **then**

return Queueemptyexception

$Q[f] \leftarrow \text{null}$

$f \leftarrow (f+1) \bmod N$

Algorithm **enqueue(o)**

if size = $N - 1$ **then**

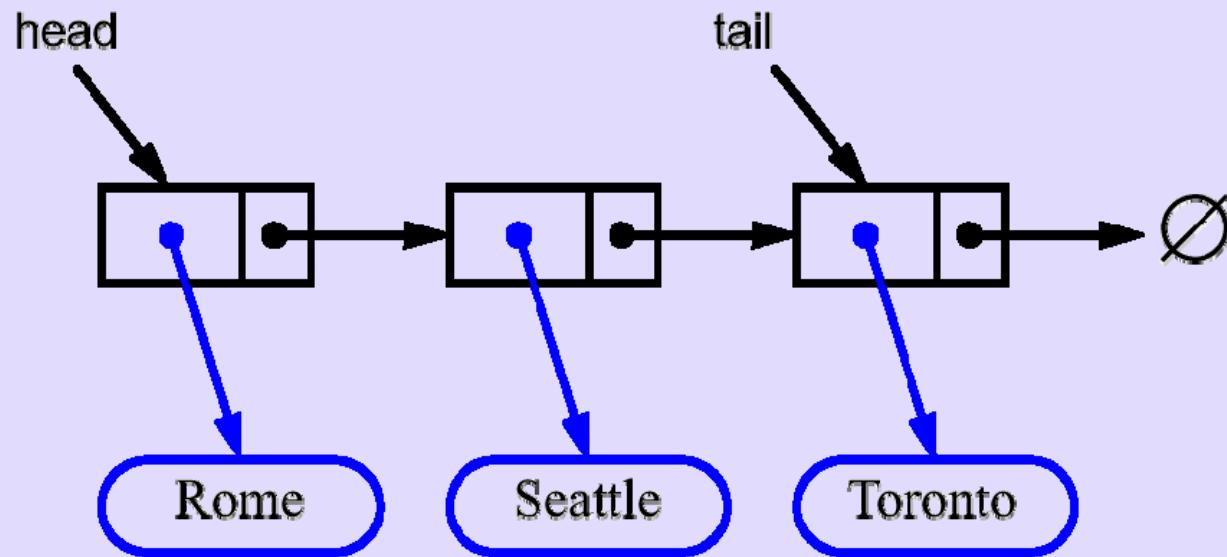
return Queuefullexception

$Q[r] \leftarrow o$

$r \leftarrow (r+1) \bmod N$

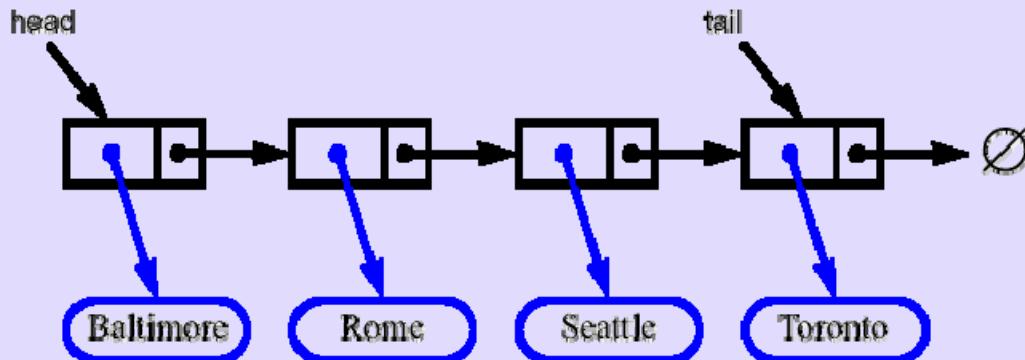
Implementing Queue with Linked List

- Nodes (*data, pointer*) connected in a chain by links

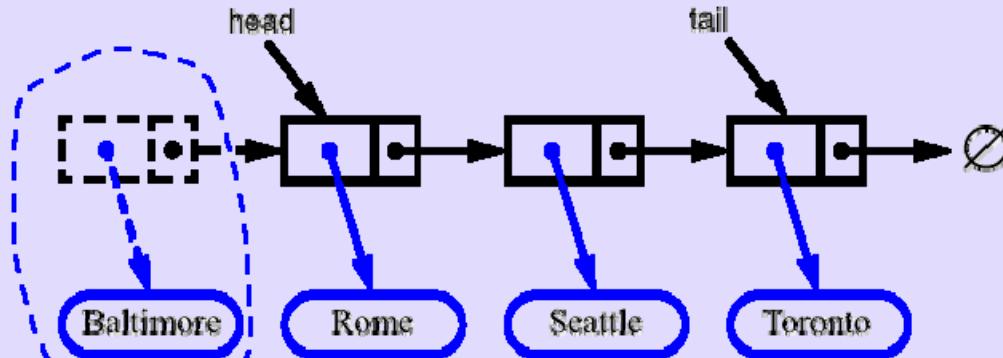


- The head of the list is the front of the queue, the tail of the list is the rear of the queue. ***Why not the opposite?***

Linked List Implementation



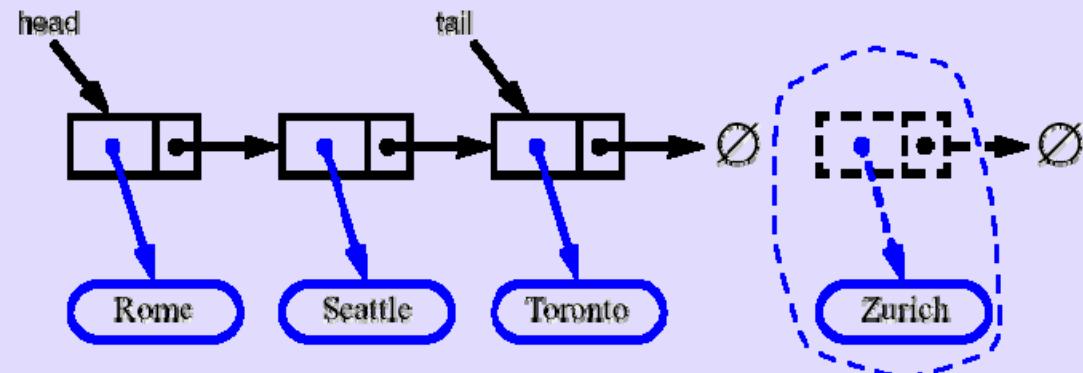
- Dequeue - advance head reference



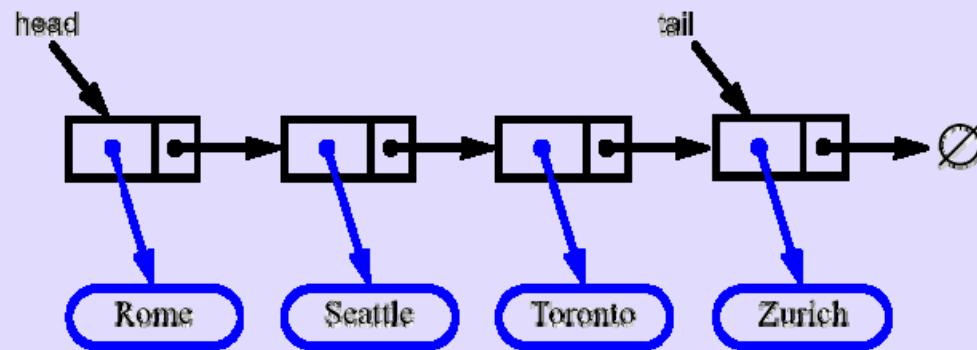
- Inserting at the head is just as easy

Linked List Implementation (2)

- Enqueue - create a new node at the tail



- chain it and move the tail reference



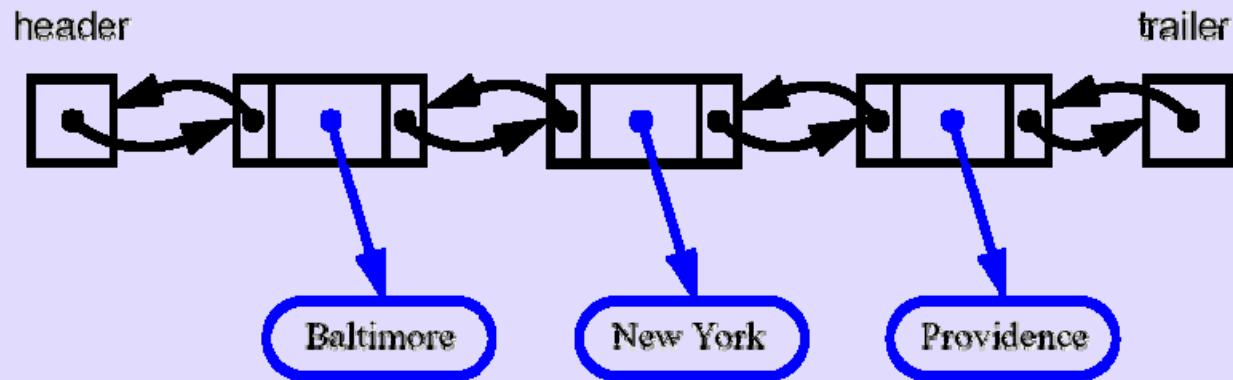
- How about removing at the tail?

Double-Ended Queue

- A **double-ended queue**, or **deque**, supports insertion and deletion from the front and back
- The deque supports six fundamental methods
 - **InsertFirst(S:ADT, o:element):ADT** - Inserts e at the beginning of deque
 - **InsertLast(S:ADT, o:element):ADT** - Inserts e at end of deque
 - **RemoveFirst(S:ADT):ADT** – Removes the first element
 - **RemoveLast(S:ADT):ADT** – Removes the last element
 - **First(S:ADT):element** and **Last(S:ADT):element** – Returns the first and the last elements

Doubly Linked Lists

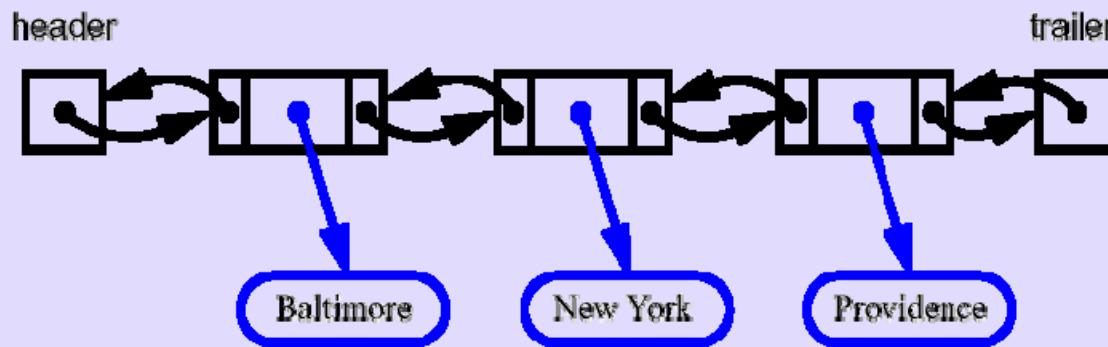
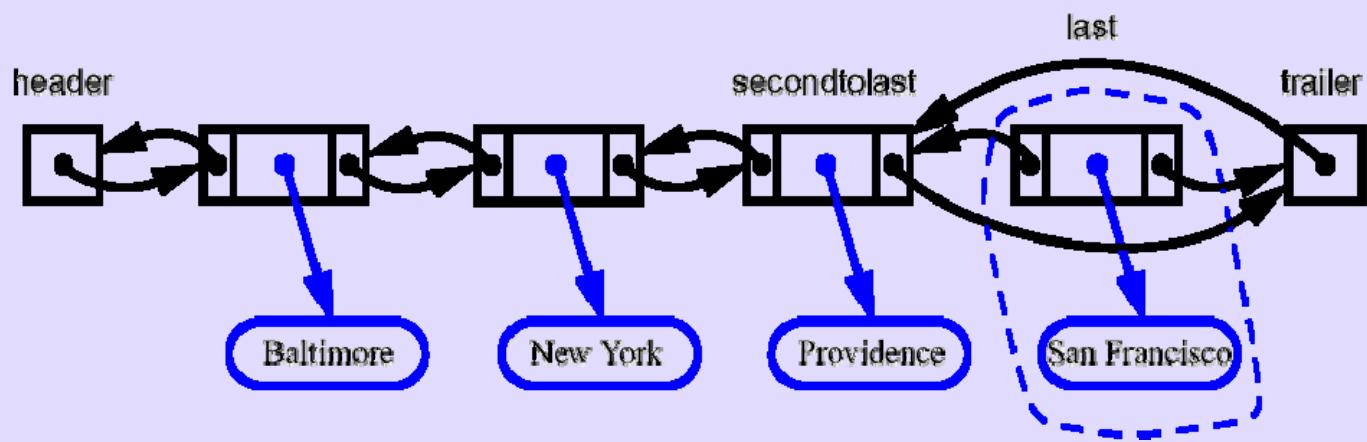
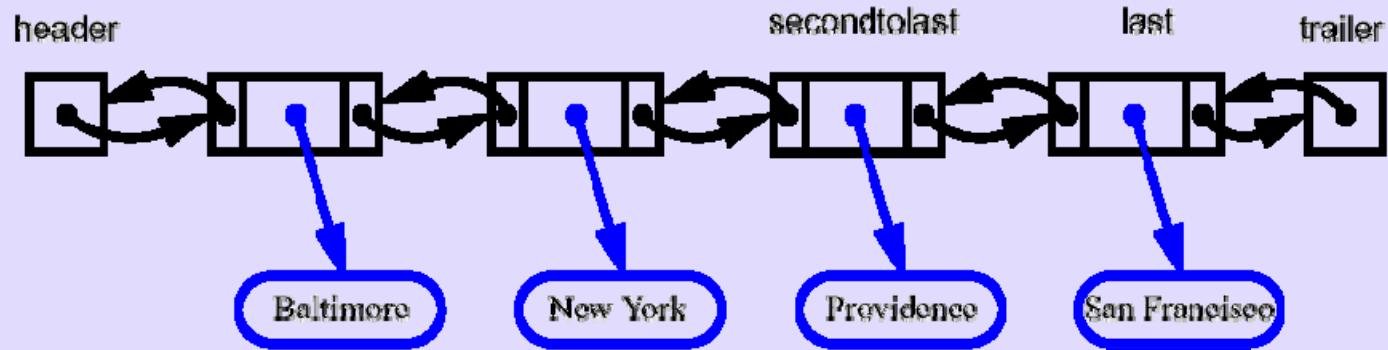
- Deletions at the tail of a singly linked list cannot be done in constant time
- To implement a deque, we use a **doubly linked list**



- A node of a doubly linked list has a **next** and a **prev** link
- Then, all the methods of a deque have a constant (that is, $O(1)$) running time.

Doubly Linked Lists (2)

- When implementing a doubly linked lists, we add two special nodes to the ends of the lists: the header and trailer nodes
 - The header node goes before the first list element. It has a valid next link but a null prev link.
 - The trailer node goes after the last element. It has a valid prev reference but a null next reference.
- The header and trailer nodes are **sentinel** or “dummy” nodes because they do not store elements



Stacks with Deques

- Implementing ADTs using implementations of other ADTs as building blocks

Stack Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
top()	last()
push(o)	insertLast(o)
pop()	removeLast()

Queues with Deques

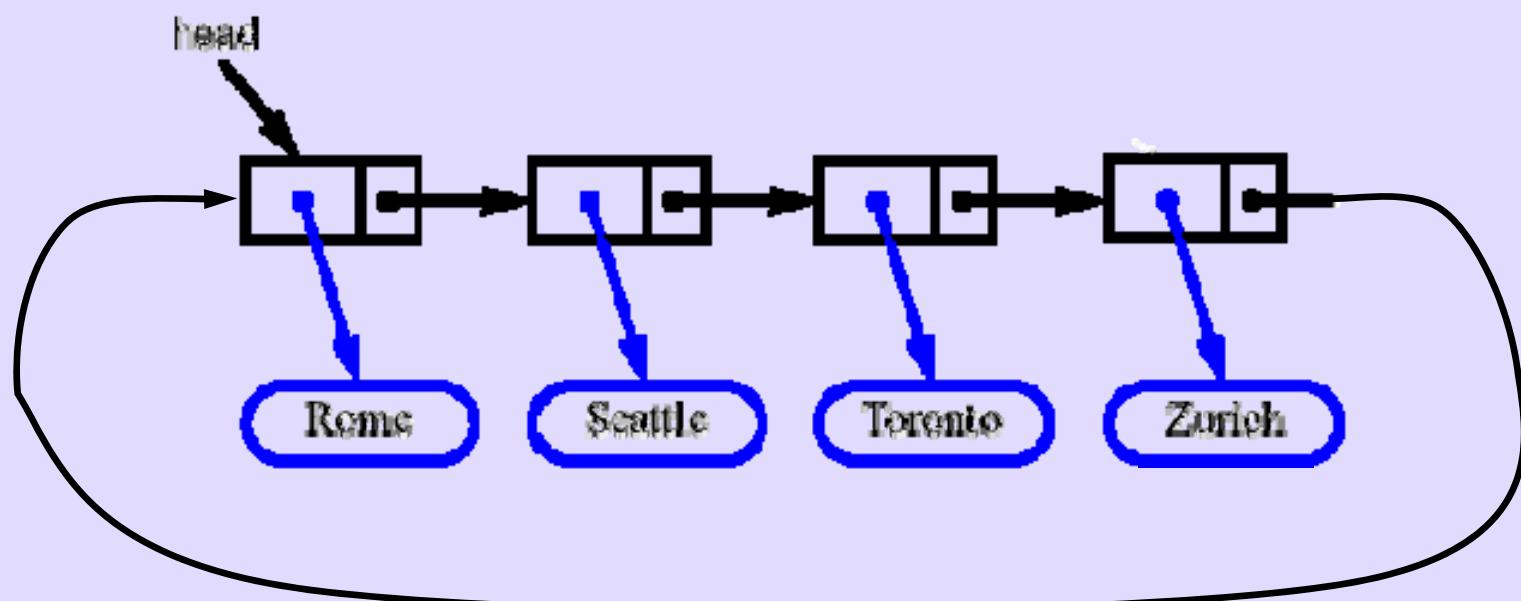
Queue Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue(o)	insertLast(o)
dequeue()	removeFirst()

The Adaptor Pattern

- Using a deque to implement a stack or queue is an example of the **adaptor pattern**. Adaptor patterns implement a class by using methods of another class
- In general, adaptor classes specialize general classes
- Two such applications
 - Specialize a general class by changing some methods eg implementing a stack with a deque.
 - Specialize the types of objects used by a general class eg defining an **IntegerArrayList** class that adapts **ArrayList** to only store integers.

Circular Lists

- No end and no beginning of the list, only one pointer as an **entry point**
- *Circular doubly linked list with a sentinel* is an elegant implementation of a stack or a queue



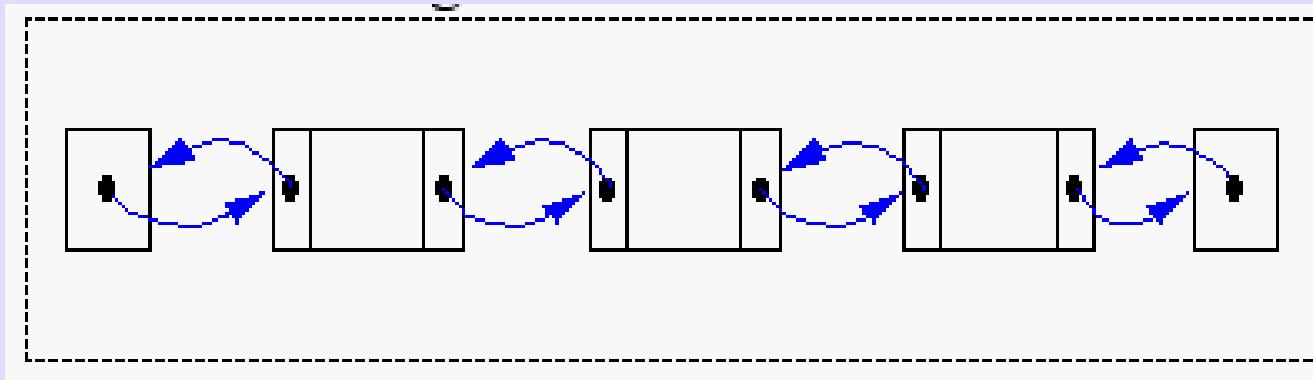






Sequences

- Vectors
- Positions
- Lists
- General Sequences



The Vector ADT

A sequence S (with n elements) that supports the following methods:

- `elemAtRank(r)`: Return the element of S with rank r; **an error occurs if $r < 0$ or $r > n - 1$**
- `replaceAtRank(r,e)`: Replace the element at rank r with e and return the old element; **an error condition occurs if $r < 0$ or $r > n - 1$**
- `insertAtRank(r,e)`: Insert a new element into S which will have rank r; **an error occurs if $r < 0$ or $r > n$**
- `removeAtRank(r)`: Remove from S the element at rank r; **an error occurs if $r < 0$ or $r > n - 1$**

Array-Based Implementation

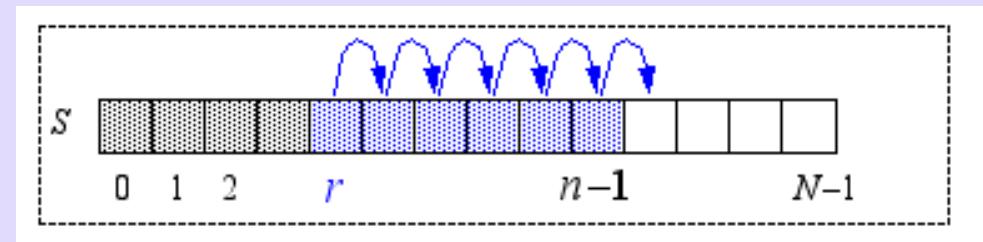
Algorithm `insertAtRank(r,e)`:

for $i = n - 1$ downto r do

$S[i+1] \leftarrow S[i]$

$S[r] \leftarrow e$

$n \leftarrow n + 1$



Algorithm `removeAtRank(r)`:

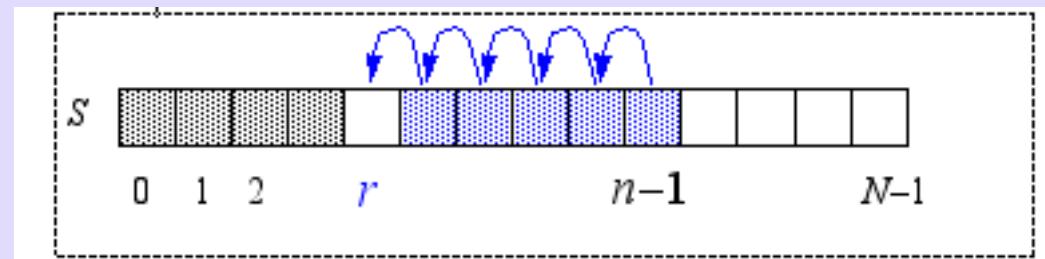
$e \leftarrow S[r]$

 for $i = r$ to $n - 2$ do

$S[i] \leftarrow S[i + 1]$

$n \leftarrow n - 1$

 return



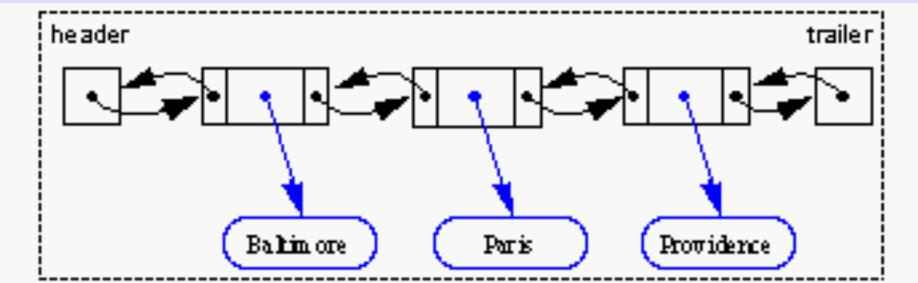
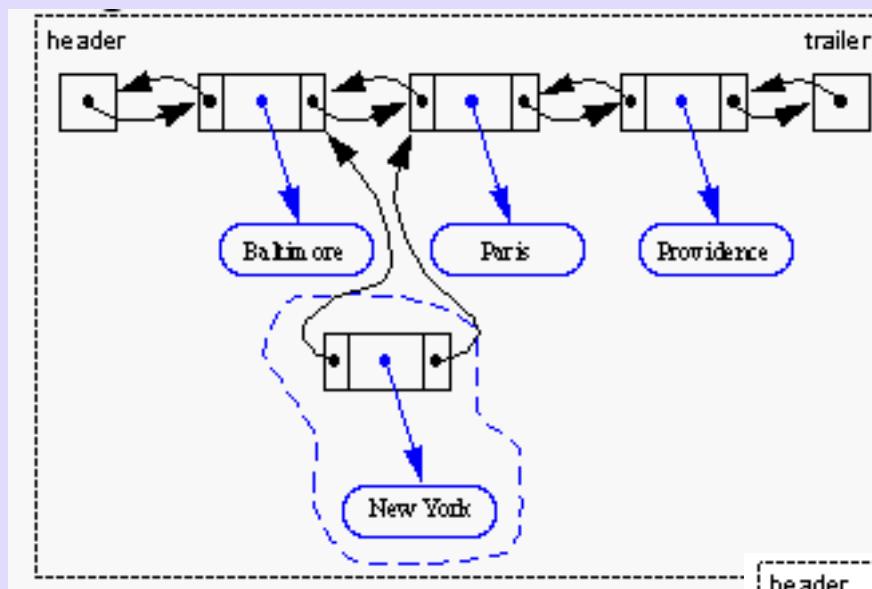
Array-Based Implementation (contd.)

Time complexity of the various methods:

Method	Time
size	$O(1)$
isEmpty	$O(1)$
elemAtRank	$O(1)$
replaceAtRank	$O(1)$
insertAtRank	$O(n)$
removeAtRank	$O(n)$

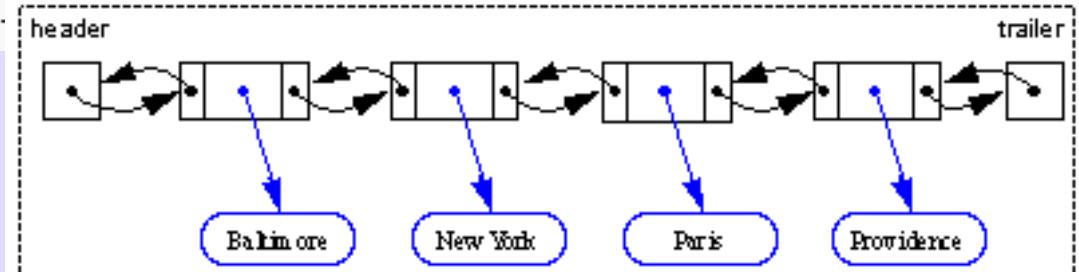
Implém. with a Doubly Linked List

the list before insertion



creating a new node for insertion:

the list after insertion

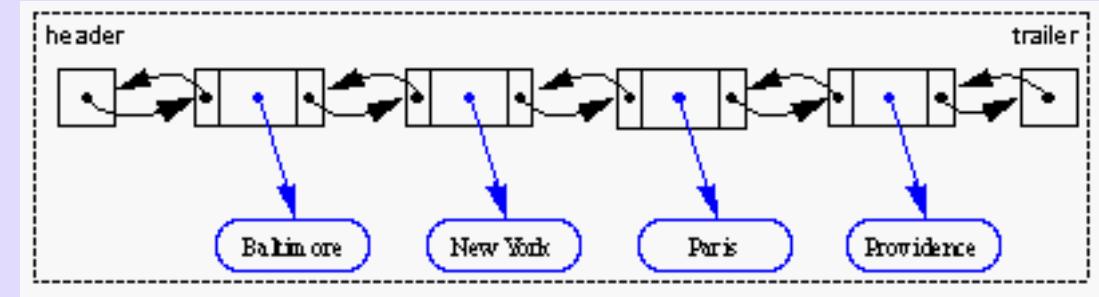


Java Implementation

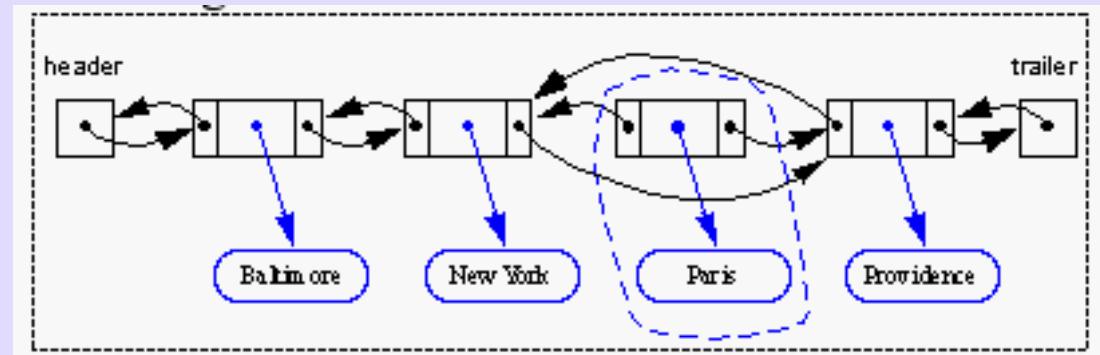
```
public void insertAtRank (int rank, Object element)
    throws BoundaryViolationException {
if (rank < 0 || rank > size())
    throw new BoundaryViolationException("invalid rank");
DLNode next = nodeAtRank(rank);
                    // the new node will be right before this
DLNode prev = next.getPrev();
                    // the new node will be right after this
DLNode node = new DLNode(element, prev, next);
                    // new node knows about its next & prev.
                    // Now we tell next & prev about the new node.
next.setPrev(node);
prev.setNext(node);
size++;
}
```

Implementation with a Doubly Linked List

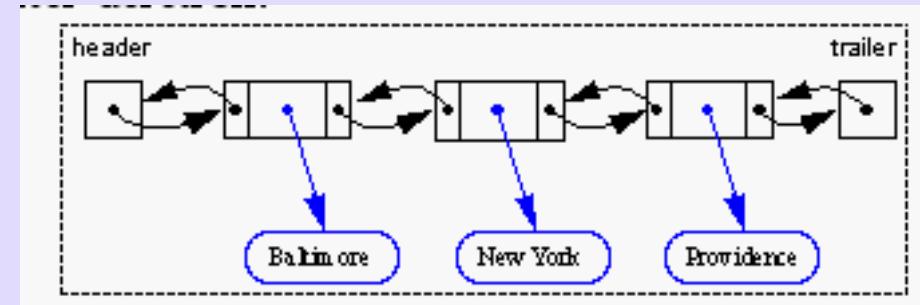
the list before deletion



deleting a node



after deletion



Java Implementation

```
public Object removeAtRank (int rank)
    throws BoundaryViolationException {
if (rank < 0 || rank > size()-1)
    throw new BoundaryViolationException("Invalid rank.");
DLNode node = nodeAtRank(rank); // node to be removed
DLNode next = node.getNext();           // node before it
DLNode prev = node.getPrev();          // node after it
prev.setNext(next);
next.setPrev(prev);
size--;
return node.getElement();
                                // returns the element of the deleted node
}
```

Java Implementation (contd.)

```
private DLNode nodeAtRank (int rank) {  
    //auxiliary method to find node of element with given rank  
    DLNode node;  
    if (rank <= size()/2) {                                //scan forward from head  
        node = header.getNext();  
        for (int i=0; i < rank; i++)  
            node = node.getNext();  
    } else {                                                 //scan backward from the tail  
        node = trailer.getPrev();  
        for (int i=0; i < size()-rank-1 ; i++)  
            node = node.getPrev();  
    }  
    return node;  
}
```

Nodes

- Linked lists support the efficient execution of *node-based operations*:
- `removeAtNode(Node v)` and `insertAfterNode(Node v, Object e)`, would be $O(1)$.
- However, node-based operations are not meaningful in an array-based implementation because there are no nodes in an array.
- Nodes are implementation-specific.
- **Dilemma:**
 - If we do not define node based operations, we are not taking full advantage of doubly-linked lists.
 - If we do define them, we violate the generality of ADTs.

From Nodes to Positions

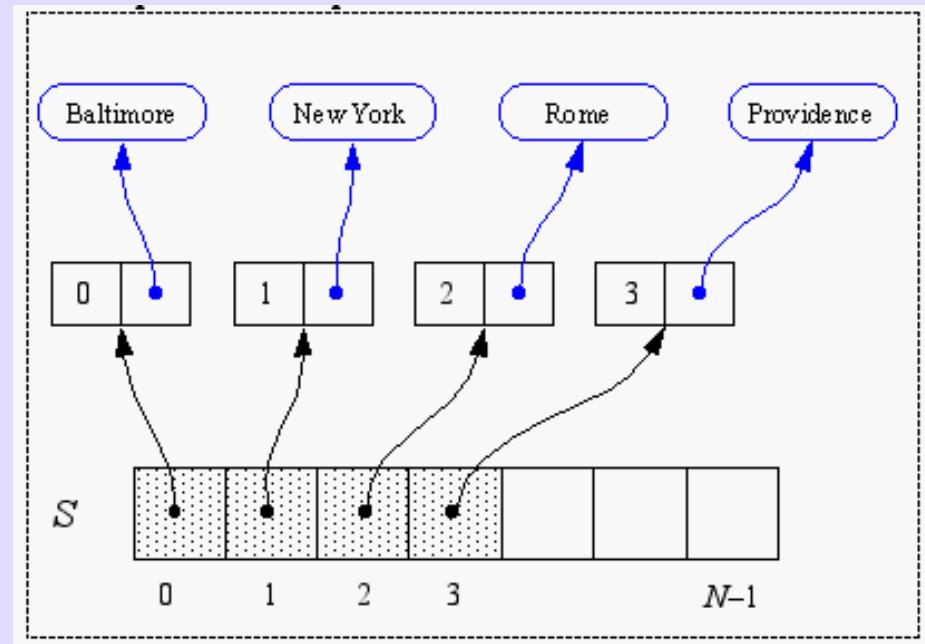
- We introduce the ***Position*** ADT
- Intuitive notion of “place” of an element.
- Positions have only one method:
element(): Returns the element at this position
- Positions are defined relative to other positions (before/after relation)
- Positions are not tied to an element or rank

The List ADT

- ADT with position-based methods
- generic methods: `size()`, `isEmpty()`
- query methods: `isFirst(p)`, `isLast(p)`
- accessor methods: `first()`, `last()`, `before(p)`, `after(p)`
- update methods
 - `swapElements(p,q)`, `replaceElement(p,e)`
 - `insertFirst(e)`, `insertLast(e)`
 - `insertBefore(p,e)`, `insertAfter(p,e)`
 - `remove(p)`
- each method takes $O(1)$ time if implemented with a doubly linked list

The Sequence ADT

- Combines the Vector and List ADT (multiple inheritance)
- Adds methods that bridge between ranks and positions
 - `atRank(r)` returns a position
 - `rankOf(p)` returns an integer rank
- An array-based implementation needs to use objects to represent the positions



Comparison of Sequence Implementations

Operations	Array	List
size, isEmpty	$O(1)$	$O(1)$
atRank, rankOf, elemAtRank	$O(1)$	$O(n)$
first, last	$O(1)$	$O(1)$
before, after	$O(1)$	$O(1)$
replaceElement, swapElements	$O(1)$	$O(1)$
replaceAtRank	$O(1)$	$O(n)$
insertAtRank, removeAtRank	$O(n)$	$O(n)$
insertFirst, insertLast	$O(1)$	$O(1)$
insertAfter, insertBefore	$O(n)$	$O(1)$
remove	$O(n)$	$O(1)$

Iterators

- Abstraction of the process of scanning through a collection of elements
- Encapsulates the notions of “place” and “next”
- Extends the position ADT
- Generic and specialized iterators
- *ObjectIterator*: `hasNext()`, `nextObject()`, `object()`
- *PositionIterator*: `nextPosition()`
- Useful methods that return iterators: `elements()`, `positions()`

Dictionaries

- the dictionary ADT
- binary search
- Hashing

Dictionaries

- Dictionaries store elements so that they can be located quickly using **keys**
- A dictionary may hold bank accounts
 - each account is an object that is identified by an account number
 - each account stores a wealth of additional information
 - including the current balance,
 - the name and address of the account holder, and
 - the history of deposits and withdrawals performed
 - an application wishing to operate on an account would have to provide the account number as a search **key**

The Dictionary ADT

- A dictionary is an abstract model of a database
 - A dictionary stores key-element pairs
 - The main operation supported by a dictionary is searching by key
- simple container methods: `size()`, `isEmpty()`, `elements()`
- query methods: `findElem(k)`, `findAllElem(k)`
- update methods: `insertItem(k,e)`, `removeElem(k)`, `removeAllElem(k)`
- special element: `NIL`, returned by an unsuccessful search

The Dictionary ADT

- Supporting order (methods *min*, *max*, *successor*, *predecessor*) is not required, thus it is enough that **keys are comparable for equality**

The Dictionary ADT

- Different data structures to realize dictionaries
 - arrays, linked lists (inefficient)
 - **Hash table** (used in Java...)
 - Binary trees
 - Red/Black trees
 - AVL trees
 - B-trees
- In Java:
 - `java.util.Dictionary` – abstract class
 - `java.util.Map` – interface

Searching

INPUT

- sequence of numbers (database)
- a single number (query)

OUTPUT

- index of the found number or *NIL*

$a_1, a_2, a_3, \dots, a_n; q$

2 5 4 10 7; 5

2 5 4 10 7; 9

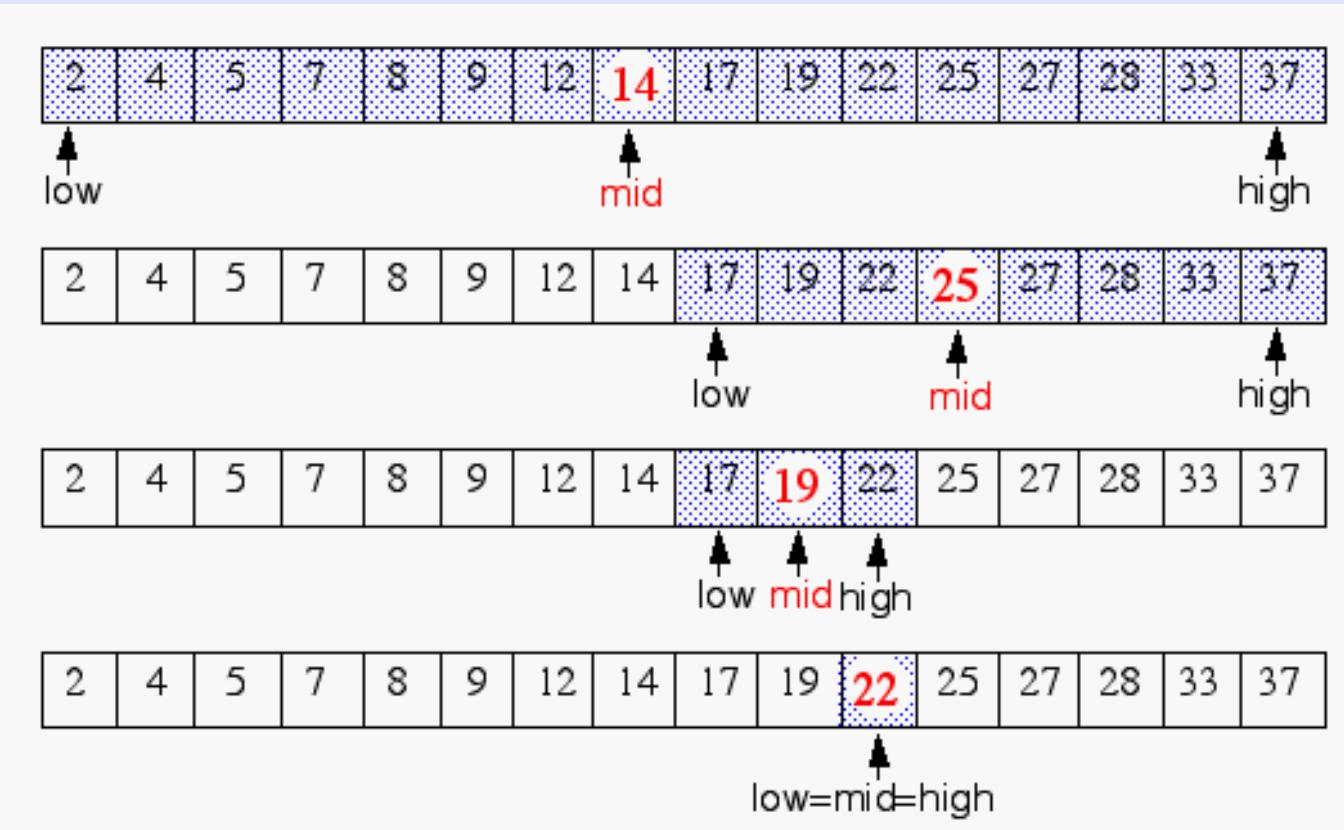
j

2

NIL

Binary Search

- Idea: *Divide and conquer*, a key design technique
- narrow down the search range in stages
- `findElement(22)`



A recursive procedure

Algorithm **BinarySearch**(A, k, low, high)

if $\text{low} > \text{high}$ then return **Nil**

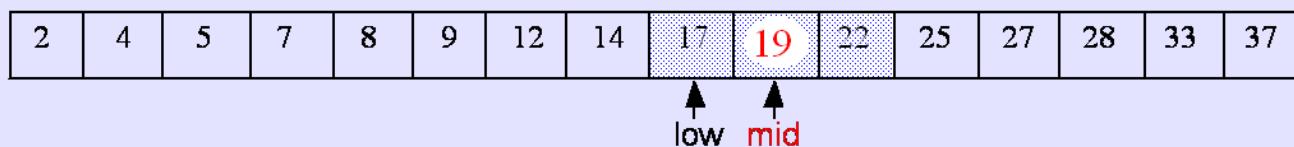
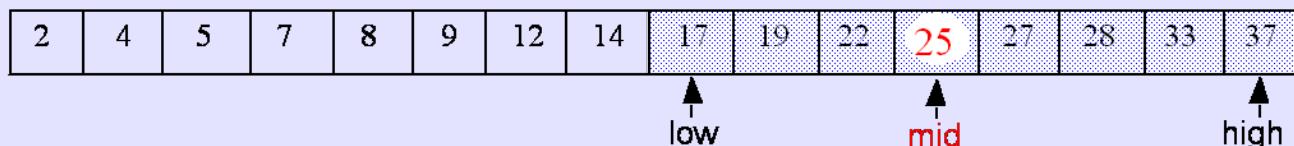
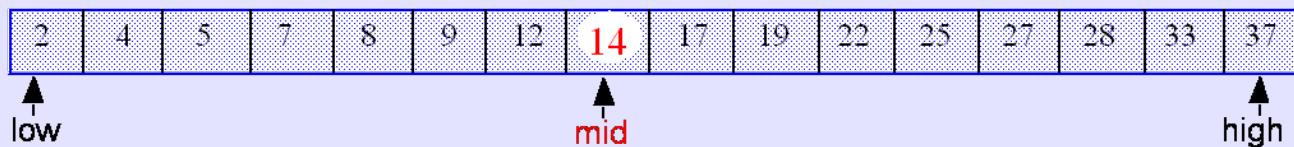
else $\text{mid} \leftarrow (\text{low}+\text{high}) / 2$

 if $k = A[\text{mid}]$ then return mid

 elseif $k < A[\text{mid}]$ then

 return **BinarySearch**(A, k, low, mid-1)

 else return **BinarySearch**(A, k, mid+1, high)



An iterative procedure

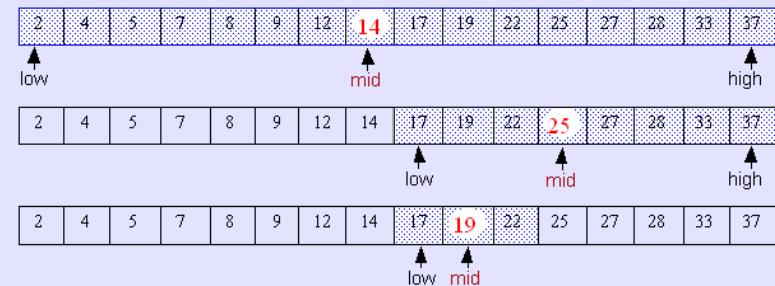
INPUT: A[1..n] – a sorted (non-decreasing) array of integers, key – an integer.

OUTPUT: an index j such that $A[j] = k$.
 NIL , if $\forall j (1 \leq j \leq n): A[j] \neq k$

```
low ← 1
high ← n
do
```

```
    mid ← (low+high)/2
    if A[mid] = k then return mid
    else if A[mid] > k then high ← mid-1
                           else low ← mid+1
```

```
while low <= high
return NIL
```



Running Time of Binary Search

- The range of candidate items to be searched is ***halved after each comparison***

comparison	search range
0	n
1	$n/2$
2	$n/4$
...	...
2^i	$n/2^i$
$\log_2 n$	1

- In the array-based implementation, access by rank takes $O(1)$ time, thus **binary search runs in $O(\log n)$ time**

Searching in an unsorted array

INPUT: $A[1..n]$ – an array of integers, q – an integer.

OUTPUT: an index j such that $A[j] = q$. NIL , if $\forall j (1 \leq j \leq n)$:
 $A[j] \neq q$

```
j ← 1
while  $j \leq n$  and  $A[j] \neq q$ 
  do  $j++$ 
if  $j \leq n$  then return  $j$ 
  else return  $\text{NIL}$ 
```

- Worst-case running time: $O(n)$, average-case: $O(n)$
- We can't do better. This is a *lower bound* for the problem of searching in an arbitrary sequence.

The Problem

T&T is a large phone company, and they want to provide caller ID capability:

- given a phone number, return the caller's name
- phone numbers range from 0 to $r = 10^8 - 1$
- There are n phone numbers, $n \ll r$.
- want to do this as efficiently as possible

Using an unordered sequence

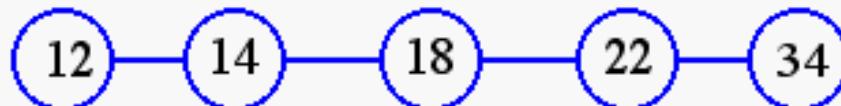
- *unordered sequence*



- searching and removing takes $O(n)$ time
- inserting takes $O(1)$ time
- applications to log files (frequent insertions, rare searches and removals)

Using an ordered sequence

- *array-based ordered sequence* (assumes keys can be ordered)



- searching takes $O(\log n)$ time (binary search)
- inserting and removing takes $O(n)$ time
- application to look-up tables (frequent searches, rare insertions and removals)

Other Suboptimal ways

direct addressing: an array indexed by key:

- takes $O(1)$ time,
- $O(r)$ space where r is the range of numbers (10^8)
- huge amount of wasted space

(null)	(null)	Ankur	(null)	(null)
0000-0000	0000-0000	9635-8904	0000-0000	0000-0000

Another Solution

- Can do better, with a **Hash table** -- $O(1)$ expected time, $O(n+m)$ space, where m is table size
- Like an array, but come up with a function to map the large range into one which we can manage
 - e.g., take the original key, modulo the (relatively small) size of the array, and use that as an index
 - Insert (9635-8904, Ankur) into a hashed array with, say, five slots. $96358904 \bmod 5 = 4$

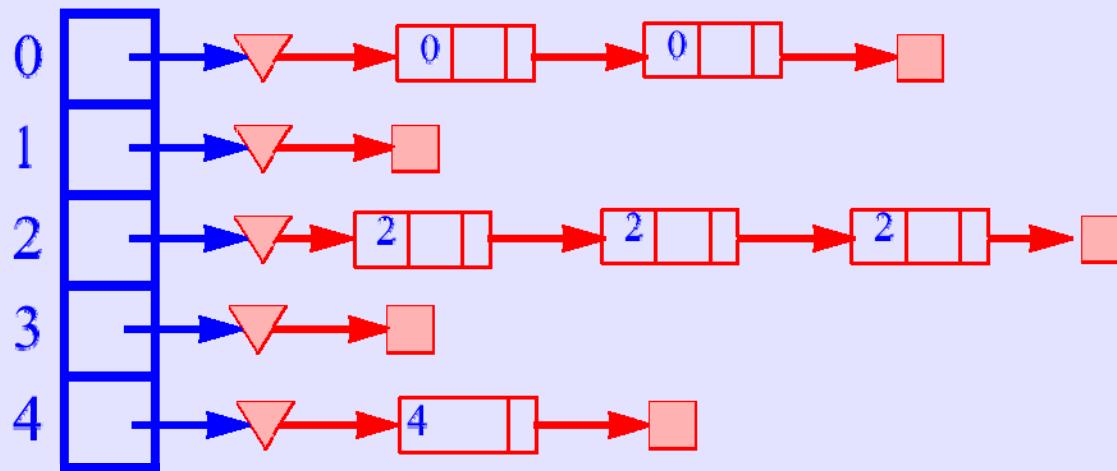
(null)	(null)	(null)	(null)	Ankur
0	1	2	3	4

An Example

- Let keys be entry no's of students in CSL201. eg. 2004CS10110.
- There are 100 students in the class. We create a hash table of size, say 100.
- Hash function is, say, last two digits.
- Then 2004CS10110 goes to location 10.
- Where does 2004CS50310 go?
- Also to location 10. **We have a collision!!**

Collision Resolution

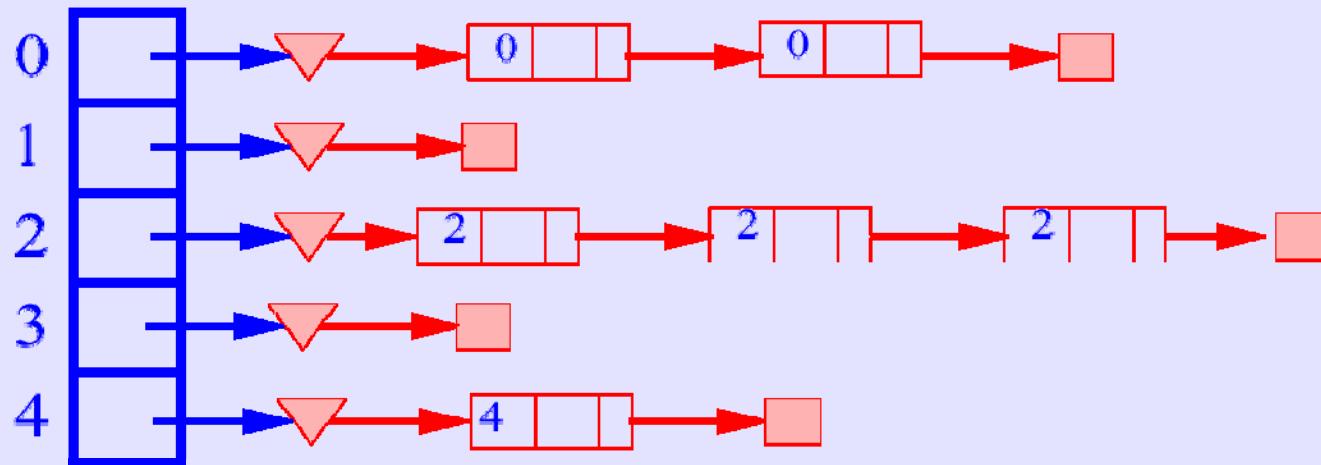
- How to deal with two keys which hash to the same spot in the array?
- Use **chaining**
 - Set up an array of links (a **table**), indexed by the keys, to **lists** of items with the same key



- Most efficient (time-wise) collision resolution scheme

Collision resolution (2)

- To find/insert/delete an element
 - using h , look up its position in table T
 - Search/insert/delete the element in the linked list of the hashed slot



Analysis of Hashing

- An element with key k is stored in slot $h(k)$ (instead of slot k without hashing)
- The hash function h maps the universe U of keys into the slots of hash table $T[0\dots m-1]$
$$h: U \rightarrow \{0, 1, \dots, m-1\}$$
- Assume time to compute $h(k)$ is $\Theta(1)$

Analysis of Hashing(2)

- A good hash function is one which distributes keys evenly amongst the slots.
- An ideal hash function would pick a slot, uniformly at random and hash the key to it.
- However, this is not a hash function since we would not know which slot to look into when searching for a key.
- For our analysis we will use this simple uniform hash function
- Given hash table T with m slots holding n elements, the **load factor** is defined as $\alpha = n/m$

Analysis of Hashing(3)

Unsuccessful search

- element is not in the linked list
- *Simple uniform* hashing yields an average list length $\alpha = n/m$
- expected number of elements to be examined α
- search time $O(1+\alpha)$ (includes computing the hash value)

Analysis of Hashing (4)

Successful search

- assume that a new element is inserted at the end of the linked list
- upon insertion of the i -th element, the expected length of the list is $(i-1)/m$
- in case of a successful search, the expected number of elements examined is 1 more than the number of elements examined when the sought-for element was inserted!

Analysis of Hashing (5)

- The expected number of elements examined is thus

$$\begin{aligned}\frac{1}{n} \sum_{i=1}^n \left(1 + \frac{i-1}{m} \right) &= 1 + \frac{1}{nm} \sum_{i=1}^n (i-1) \\ &= 1 + \frac{1}{nm} \cdot \frac{(n-1)n}{2} \\ &= 1 + \frac{n-1}{2m} \\ &= 1 + \frac{n}{2m} - \frac{1}{2m} \\ &= 1 + \frac{\alpha}{2} - \frac{1}{2m}\end{aligned}$$

- Considering the time for computing the hash function, we obtain

$$\Theta(2 + \alpha/2 - 1/2m) = \Theta(1 + \alpha)$$

Analysis of Hashing (6)

Assuming the number of hash table slots is proportional to the number of elements in the table

- $n = O(m)$
- $\alpha = n/m = O(m)/m = O(1)$
- searching takes constant time on average
- insertion takes $O(1)$ worst-case time
- deletion takes $O(1)$ worst-case time when the lists are doubly-linked

Hashing

- Hash functions
 - Hash-code maps
 - Compression maps
- Open addressing
 - Linear Probing
 - Double hashing

Hash Functions

- Need to choose a good hash function
 - quick to compute
 - distributes keys uniformly throughout the table
 - good hash functions are very rare – *birthday paradox*
- How to deal with hashing non-integer keys:
 - find some way of turning keys into integers
 - eg. remove hyphen in 9635-8904 to get 96358904!
 - for a string, add up ASCII values of the characters of your string (e.g., `java.lang.String.hashCode()`)
 - then use standard hash function on the integers

From Keys to Indices

- The mapping of keys to indices of a hash table is called a ***hash function***
- A hash function is usually the composition of two maps, a ***hash code map*** and a ***compression map***.
 - An essential requirement of the hash function is to map equal keys to equal indices
 - A “good” hash function minimizes the probability of collisions
 - ***hash code map***: key → integer
 - ***compression map***: integer → $[0, N - 1]$

Popular Hash-Code Maps

- **Integer cast:** for numeric types with 32 bits or less, we can reinterpret the bits of the number as an int
- **Component sum:** for numeric types with more than 32 bits (e.g., long and double), we can add the 32-bit components.
- Why is the component-sum hash code bad for strings?

Hash-Code Maps (2)

- **Polynomial accumulation:** for strings of a natural language, combine the character values (ASCII or Unicode) $a_0 a_1 \dots a_{n-1}$ by viewing them as the coefficients of a polynomial:

$$a_0 + a_1 x + \dots + x^{n-1} a_{n-1}$$

- The polynomial is computed with Horner's rule, ignoring overflows, at a fixed value x :

$$a_0 + x(a_1 + x(a_2 + \dots + x(a_{n-2} + x a_{n-1}) \dots))$$

- The choice $x = 33, 37, 39$, or 41 gives at most 6 collisions on a vocabulary of 50,000 English words

Compression Maps

- Use the remainder
 - $h(k) = k \bmod m$, k is the key, m the size of the table
- Need to choose m
- $m = b^e$ (**bad**)
 - if m is a power of 2, $h(k)$ gives the **e** least significant bits of k
 - all keys with the same ending go to the same place
- m prime (**good**)
 - helps ensure uniform distribution
 - primes not too close to exact powers of 2

Compression Maps (2)

- Example
 - hash table for $n = 2000$ character strings
 - we don't mind examining 3 elements
 - $m = 701$
 - a prime near $2000/3$
 - but not near any power of 2

Compression Maps (3)

- Use
 - $h(k) = \lfloor m (k A \bmod 1) \rfloor$
 - k is the key, m the size of the table, and A is a constant
 $0 < A < 1$
- The steps involved
 - map $0 \dots k_{max}$ into $0 \dots k_{max} A$
 - take the fractional part ($\bmod 1$)
 - map it into $0 \dots m-1$

Compression Maps (4)

Choice of m and A

- value of m is not critical, typically use $m = 2^p$
- optimal choice of A depends on the characteristics of the data
- Knuth says use $A = \frac{\sqrt{5}-1}{2}$ (conjugate of the *golden ratio*) – *Fibonacci hashing*

Compression Maps (5)

Multiply, Add, and Divide (MAD):

$$h(k) = |ak + b| \bmod N$$

- eliminates patterns provided a is not a multiple of N
- same formula used in linear congruential (pseudo) random number generators

Universal Hashing

- For any choice of hash function, there exists a bad set of identifiers
- A malicious adversary could choose keys to be hashed such that all go into the same slot (bucket)
- Average retrieval time is $\Theta(n)$
- Solution
 - a random hash function
 - choose hash function independently of keys!
 - create a **set of hash functions H** , from which h can be randomly selected

Universal Hashing (2)

- A collection H of hash functions is *universal* if for any randomly chosen f from H (and two keys k and l),

$$\Pr\{f(k) = f(l)\} \leq 1/m$$

More on Collisions

- A key is mapped to an already occupied table location
 - what to do?!?
- Use a collision handling technique
- We've seen *Chaining*
- Can also use *Open Addressing*
 - Probing
 - Double Hashing

Open Addressing

- All elements are stored in the hash table (can fill up!), i.e., $n \leq m$
- Each table entry contains either an element or null
- When searching for an element, systematically probe table slots

Open Addressing (2)

- Modify hash function to take the probe number i as the second parameter

$$h : U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$$

- Hash function, h , determines the sequence of slots examined for a given key
- Probe sequence for a given key k given by $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$ - a permutation of $\langle 0, 1, \dots, m-1 \rangle$

Linear Probing

- If the current location is used, try the next table location

```
LinearProbingInsert(k)
```

```
    if (table is full) error
```

```
    probe = h(k)
```

```
    while (table[probe] occupied)
```

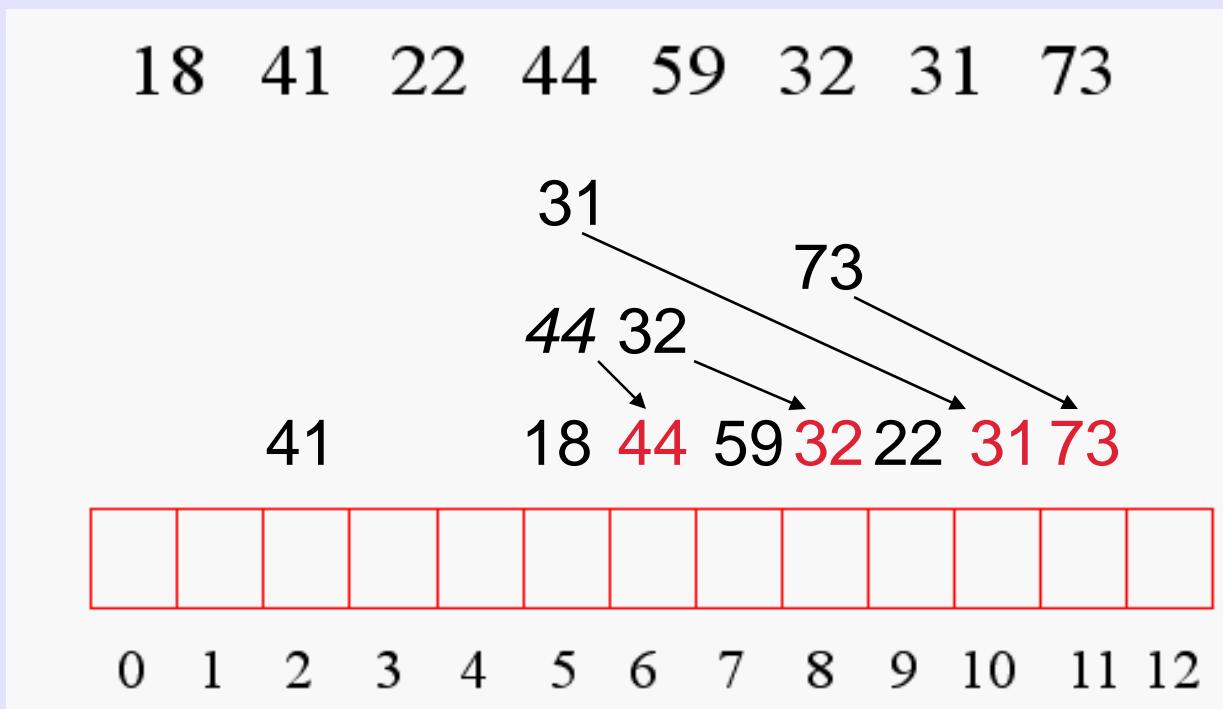
```
        probe = (probe+1) mod m
```

```
    table[probe] = k
```

- Uses less memory than chaining as one does not have to store all those links
- Slower than chaining since one might have to walk along the table for a long time

Linear Probing Example

- $h(k) = k \bmod 13$
- insert keys: 18 41 22 44 59 32 31 73

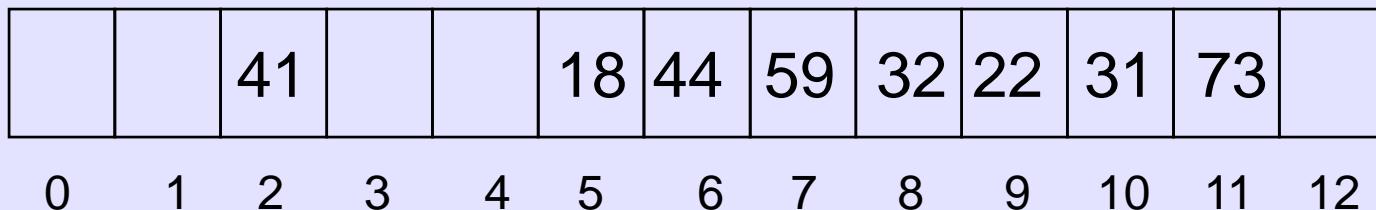


Lookup in linear probing



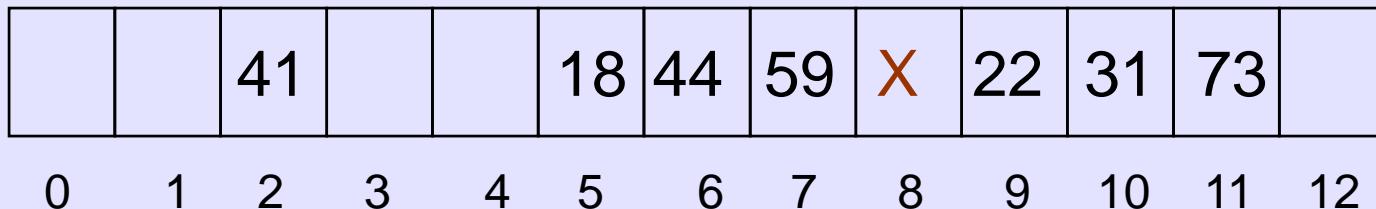
- To search for a key k we go to $(k \bmod 13)$ and continue looking at successive locations till we find k or encounter an empty location.
- Successful search: To search for 31 we go to $(31 \bmod 13) = 5$ and continue onto 6,7,8... till we find 31 at location 10
- Unsuccessful search: To search for 33 we go to $(33 \bmod 13) = 7$ and continue till we encounter an empty location (12)

Deletion in Linear Probing



- To delete key 32 we first search for 32.
- 32 is found in location 8. Suppose we set this location to null.
- Now if we search for 31 we will encounter a null location before seeing 31.
- Lookup procedure would declare that 31 is not present.

Deletion (2)



- Instead of setting location 8 to null place a tombstone (a marker) there.
- When lookup encounters a tombstone it ignores it and continues with next location.
- If Insert comes across a tombstone it puts the element at that location and removes the tombstone.
- Too many tombstones degrades lookup performance.
- Rehash if there are too many tombstones.

Double Hashing

Uses two hash functions, h_1 , h_2

- $h_1(k)$ is the position in the table where we first check for key k
- $h_2(k)$ determines the offset we use when searching for k
- In linear probing $h_2(k)$ is always 1.

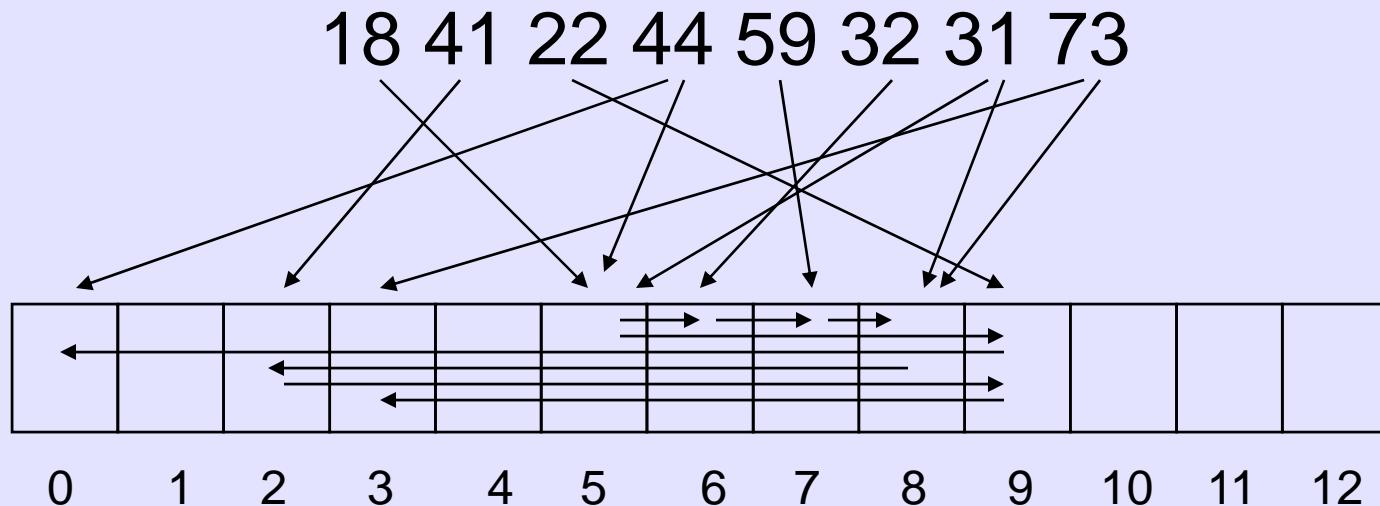
```
DoubleHashingInsert(k)
if (table is full) error
probe = h1(k); offset = h2(k)
while (table[probe] occupied)
    probe = (probe+offset) mod m
table[probe] = k
```

Double Hashing(2)

- If m is prime, we will eventually examine every position in the table
- Many of the same (dis)advantages as linear probing
- Distributes keys more uniformly than linear probing

Double Hashing Example

- $h1(k) = k \bmod 13$
- $h2(k) = 8 - (k \bmod 8)$
- insert keys: 18 41 22 44 59 32 31 73



Analysis of Double Hashing

- The load factor α is less than 1.
- We assume that every probe looks at a random location in the table.
- $1 - \alpha$ fraction of the table is empty.
- Expected number of probes required to find an empty location (unsuccessful search) is $1/(1 - \alpha)$

Analysis (2)

- Average no of probes for a successful search = average no of probes required to insert all the elements.
- To insert an element we need to find an empty location.

inserting	Avg no of probes	Total no of probes
First $m/2$	≤ 2	m
Next $m/4$	≤ 4	m
Next $m/8$	≤ 8	m

Analysis(3)

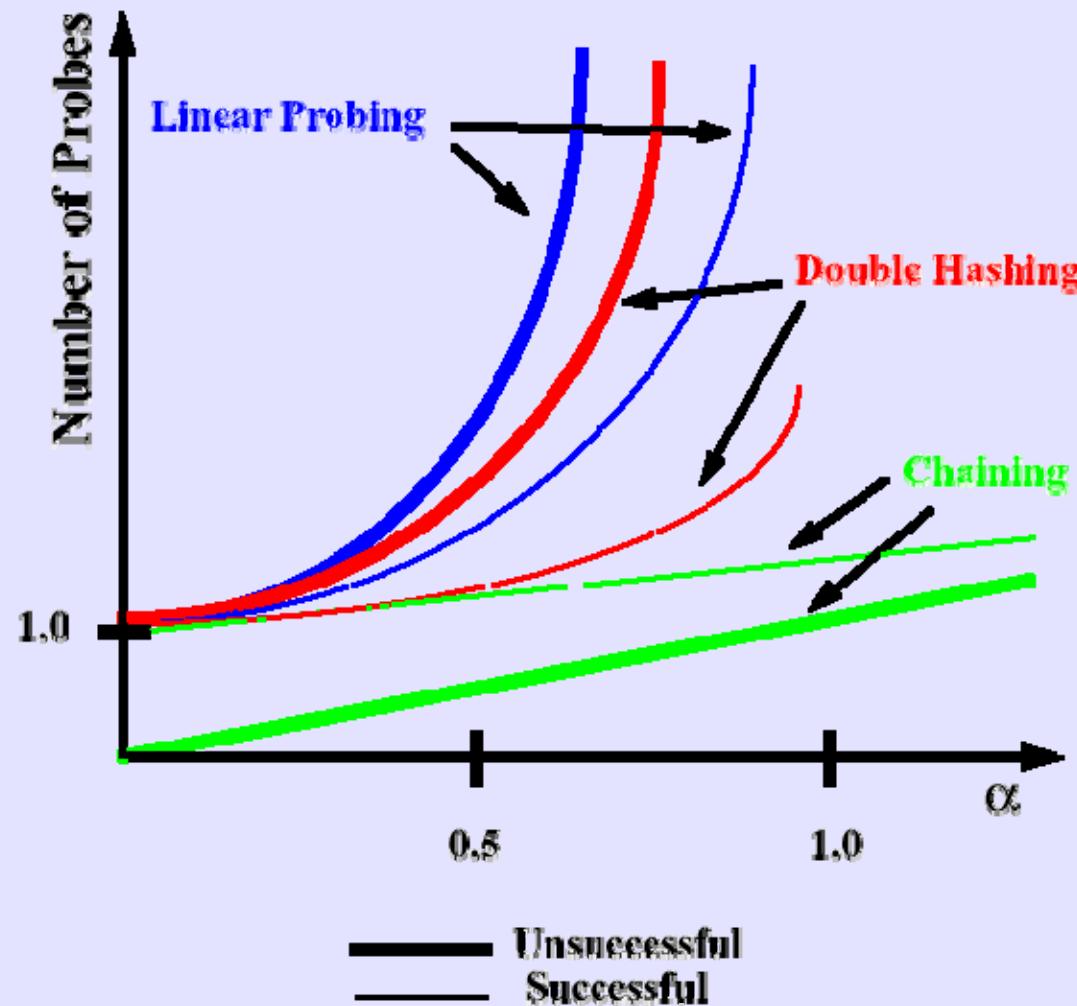
- No of probes required to insert $m/2+m/4+m/8+\dots+m/2^i$ elements = number of probes required to leave 2^{-i} fraction of the table empty = $m \times i$.
- No of probes required to leave $1-\alpha$ fraction of the table empty = $-m \log (1-\alpha)$
- Average no. of probes required to insert n elements is $-(m/n) \log (1-\alpha) = -(1/\alpha) \log (1-\alpha)$

Expected Number of Probes

- Load factor $\alpha < 1$ for probing
- Analysis of probing uses *uniform hashing* assumption – any permutation is equally likely
 - What about linear probing and double hashing?

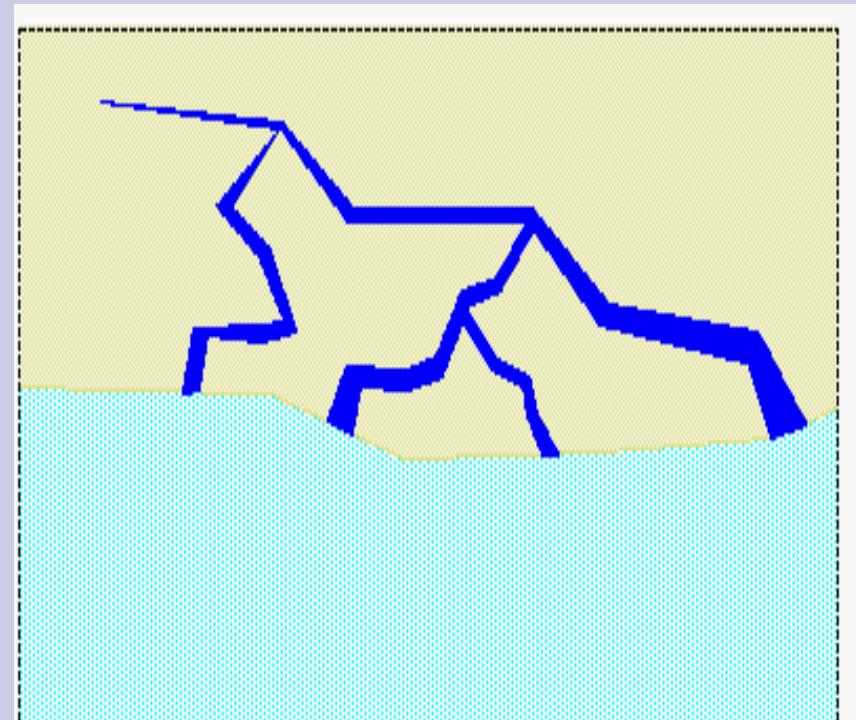
	unsuccessful	successful
chaining	$O(1 + \alpha)$	$O(1 + \alpha)$
probing	$O\left(\frac{1}{1 - \alpha}\right)$	$O\left(\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}\right)$

Expected Number of Probes (2)



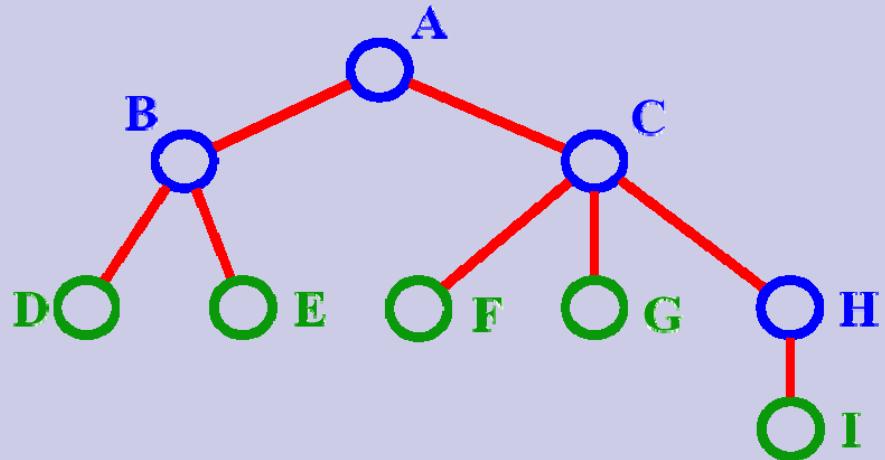
Trees

- trees
- binary trees
- data structures for trees



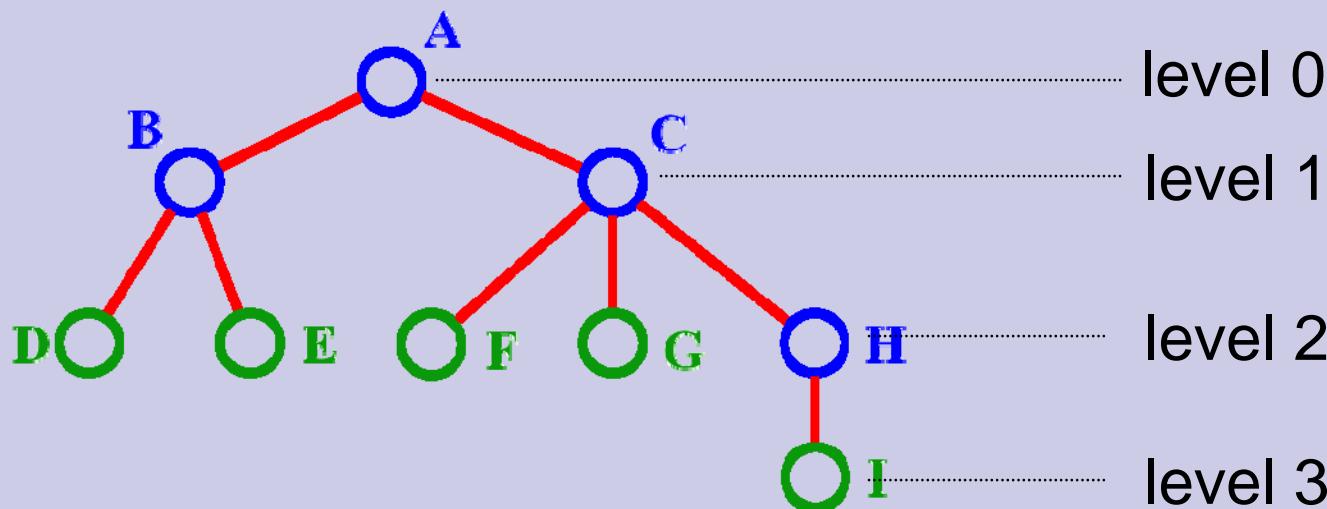
Trees: Definitions

- *A* is the *root* node.
- *B* is *parent* of D & E.
- *A* is *ancestor* of D & E.
- D and E are *descendants* of *A*.
- *C* is the *sibling* of B
- *D* and *E* are the *children* of B.
- *D, E, F, G, I* are *leaves*.



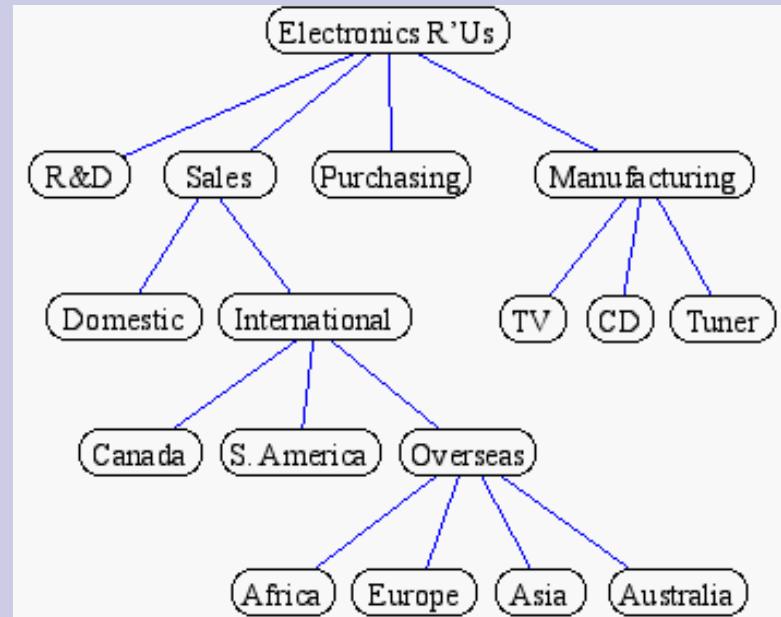
Trees: Definitions (2)

- A, B, C, H are *internal nodes*
- The *depth (level)* of E is 2
- The *height* of the tree is 3
- The *degree* of node B is 2

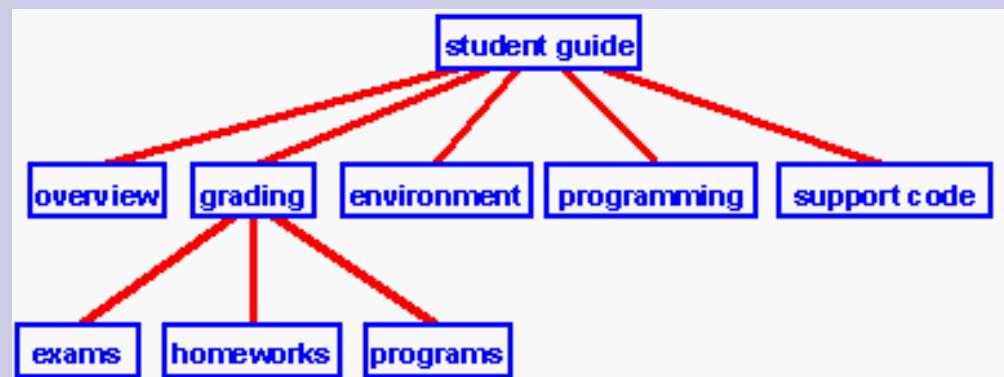


Trees

A tree represents a hierarchy, for e.g. the organization structure of a corporation

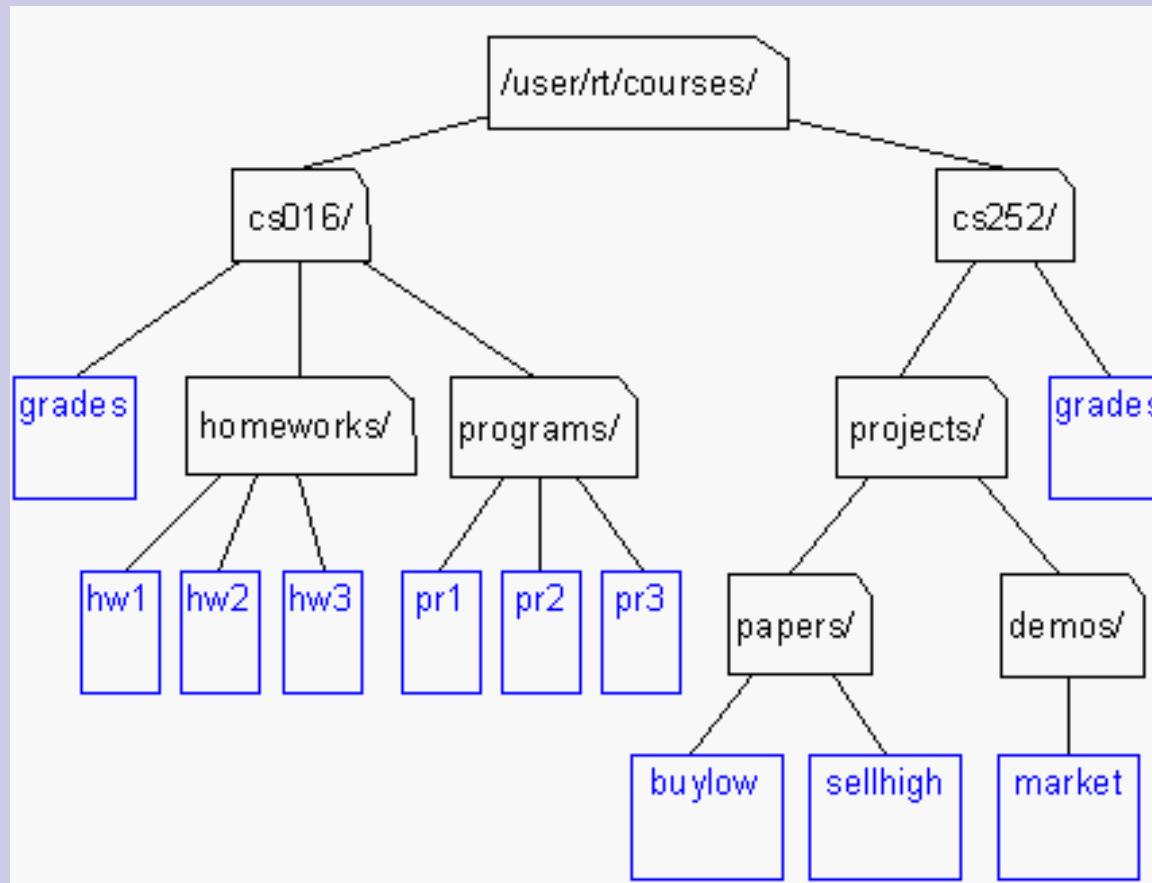


Or table of contents of a book



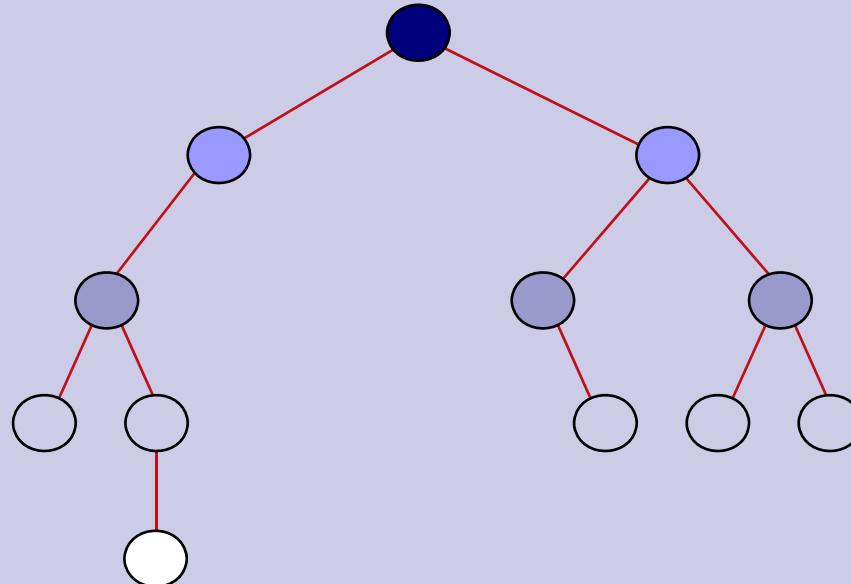
Another Example

Unix or DOS/Windows file system



Binary Tree

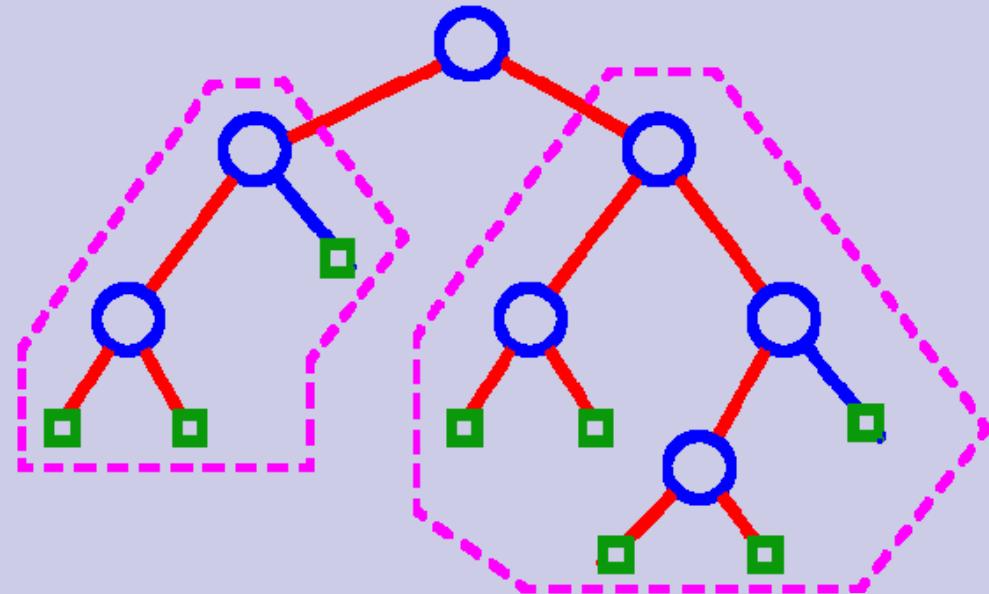
- An **ordered tree** is one in which the children of each node are ordered.
- **Binary tree:** ordered tree with all nodes having at most 2 children.



Recursive definition of binary tree

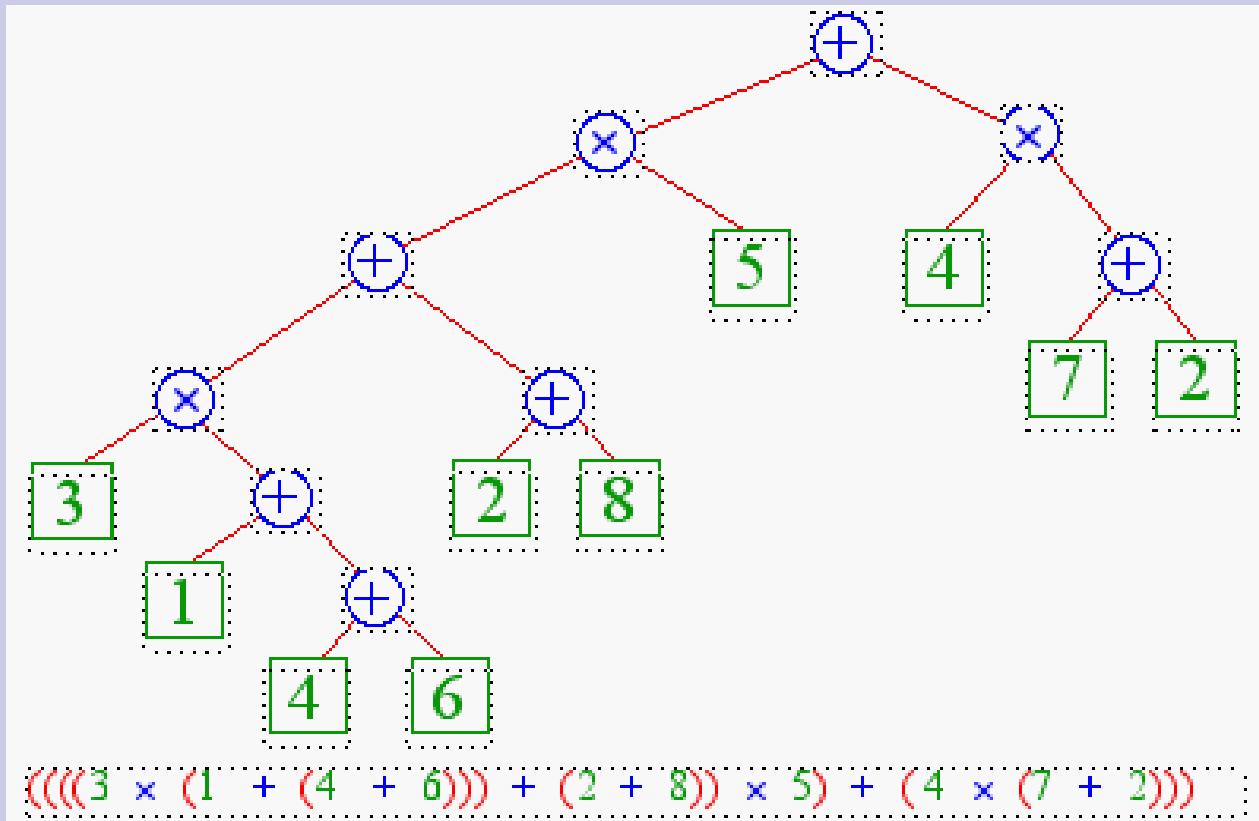
A binary tree is either a

- leaf or
- An internal node (the root) and one/two binary trees (left subtree and/or right subtree).



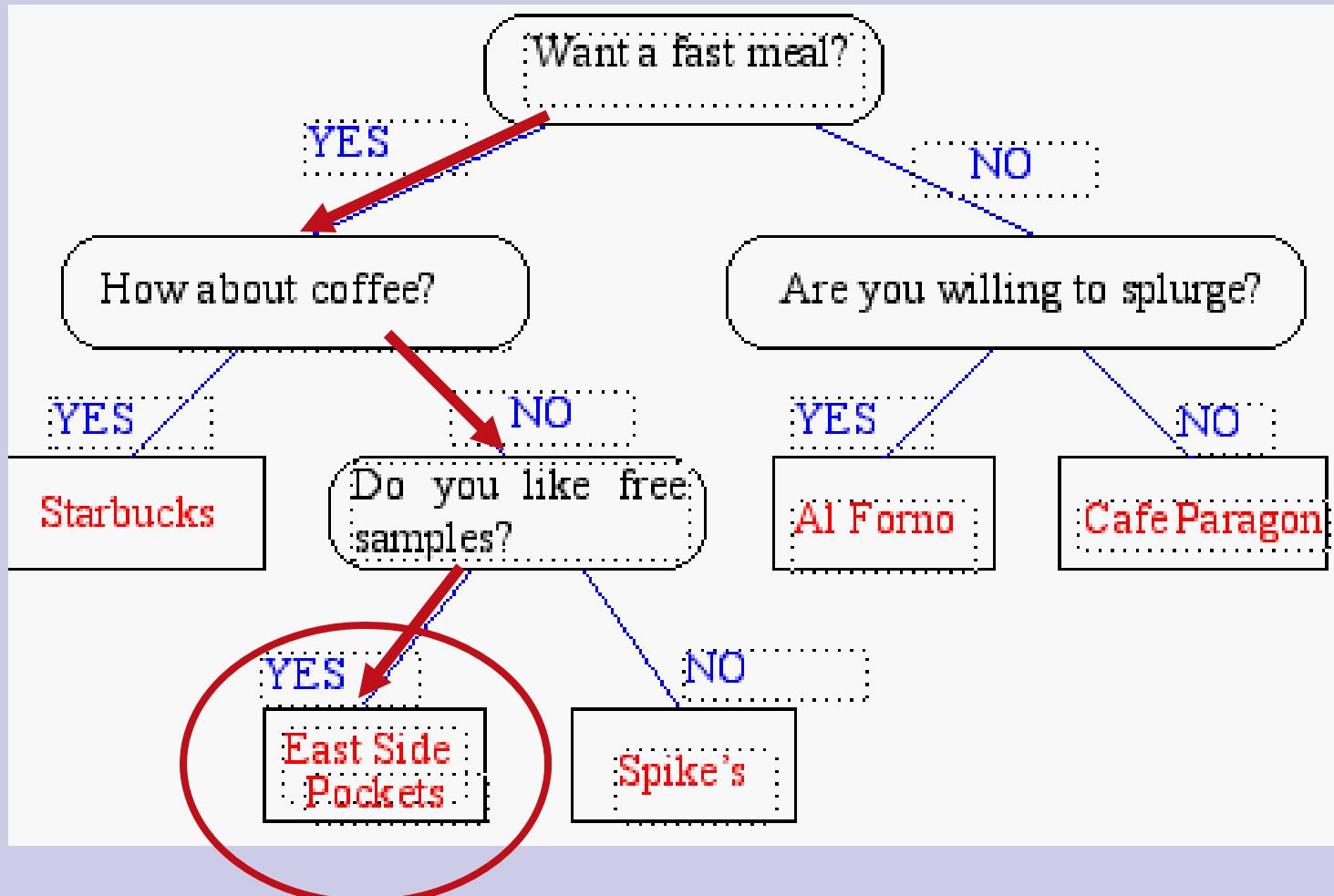
Examples of Binary Trees

arithmetic expressions



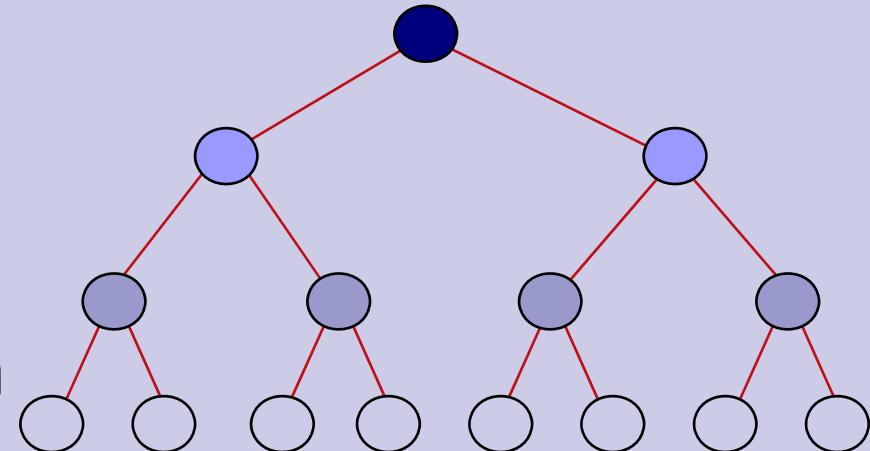
Examples of Binary Trees

decision trees



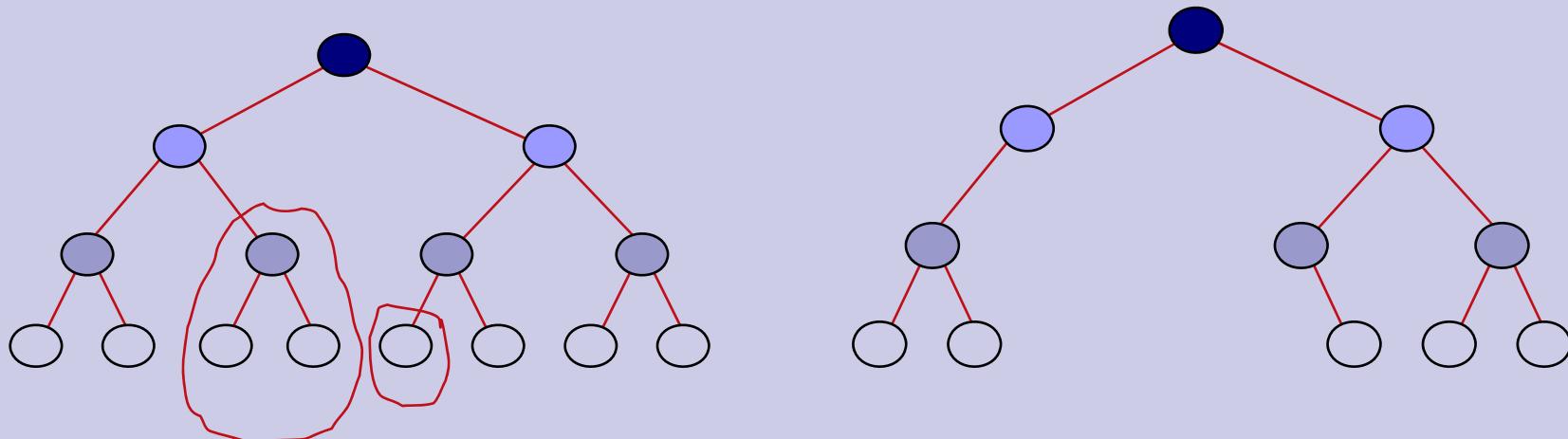
Complete Binary tree

- level i has 2^i nodes
- In a tree of height h
 - leaves are at level h
 - No. of leaves is 2^h
 - No. of internal nodes = $1+2+2^2+\dots+2^{h-1} = 2^h - 1$
 - **No of internal nodes = no of leaves - 1**
 - Total no. of nodes is $2^{h+1}-1 = n$
- In a tree of n nodes
 - No of leaves is $(n+1)/2$
 - Height = \log_2 (no of leaves)



Binary Tree

- A Binary tree can be obtained from an appropriate complete binary tree by pruning

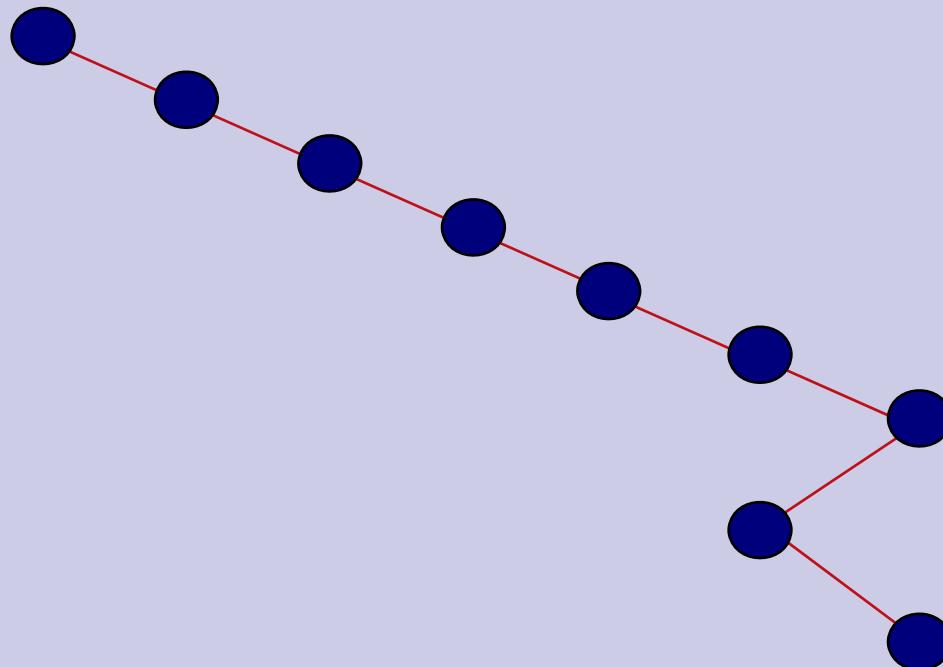


Minimum height of a binary tree

- A binary tree of height h has
 - At most 2^i nodes at level i
 - At most $1+2+2^2+\dots+2^h = 2^{h+1}-1$ nodes
- If the tree has n nodes then
 - $n \leq 2^{h+1}-1$
 - Hence $h \geq \log_2(n+1)/2$

Maximum height of a binary tree

- A binary tree on n nodes has height at most $n-1$
- This is obtained when every node (except the leaf) has exactly one child



No of leaves in a binary tree

- no of leaves $\leq 1 +$ no of internal nodes.
- Proof: by induction on no of internal nodes
 - Tree with 1 node has a leaf but no internal node.
 - Assume stmt is true for tree with $k-1$ internal nodes.
 - A tree with k internal nodes has k_1 internal nodes in left subtree and $(k-k_1-1)$ internal nodes in right subtree.
 - No of leaves $\leq (k_1+1)+(k-k_1) = k+1$

leaves in a binary tree (2)

For a binary tree on n nodes

- No of leaves + no of internal nodes = n
- No of leaves \leq no of internal nodes + 1
- Hence, no of leaves $\leq (n+1)/2$
- Minimum no of leaves is 1

ADTs for Trees

- generic container methods: `size()`, `isEmpty()`, `elements()`
- positional container methods: `positions()`, `swapElements(p,q)`, `replaceElement(p,e)`
- query methods: `isRoot(p)`, `isInternal(p)`, `isExternal(p)`
- accessor methods: `root()`, `parent(p)`, `children(p)`
- update methods: application specific

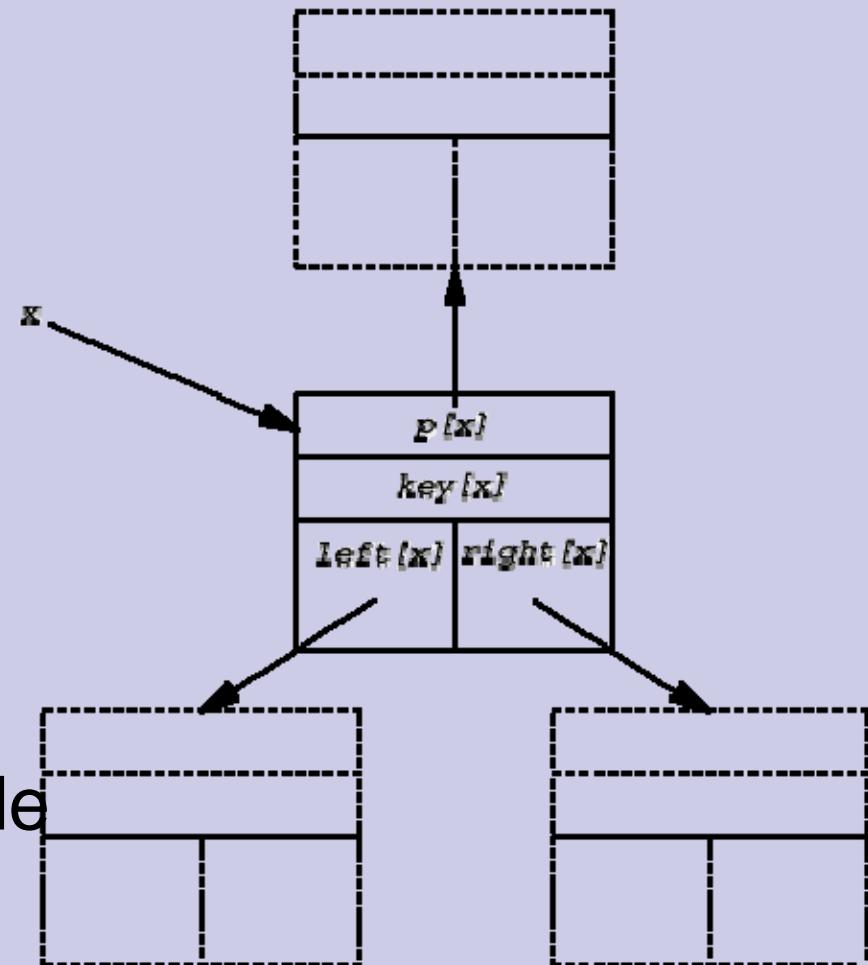
ADTs for Binary Trees

- accessor methods: `leftChild(p)`,
`rightChild(p)`, `sibling(p)`
- update methods:
 - `expandExternal(p)`, `removeAboveExternal(p)`
 - other application specific methods

The Node Structure

Each node in the tree contains

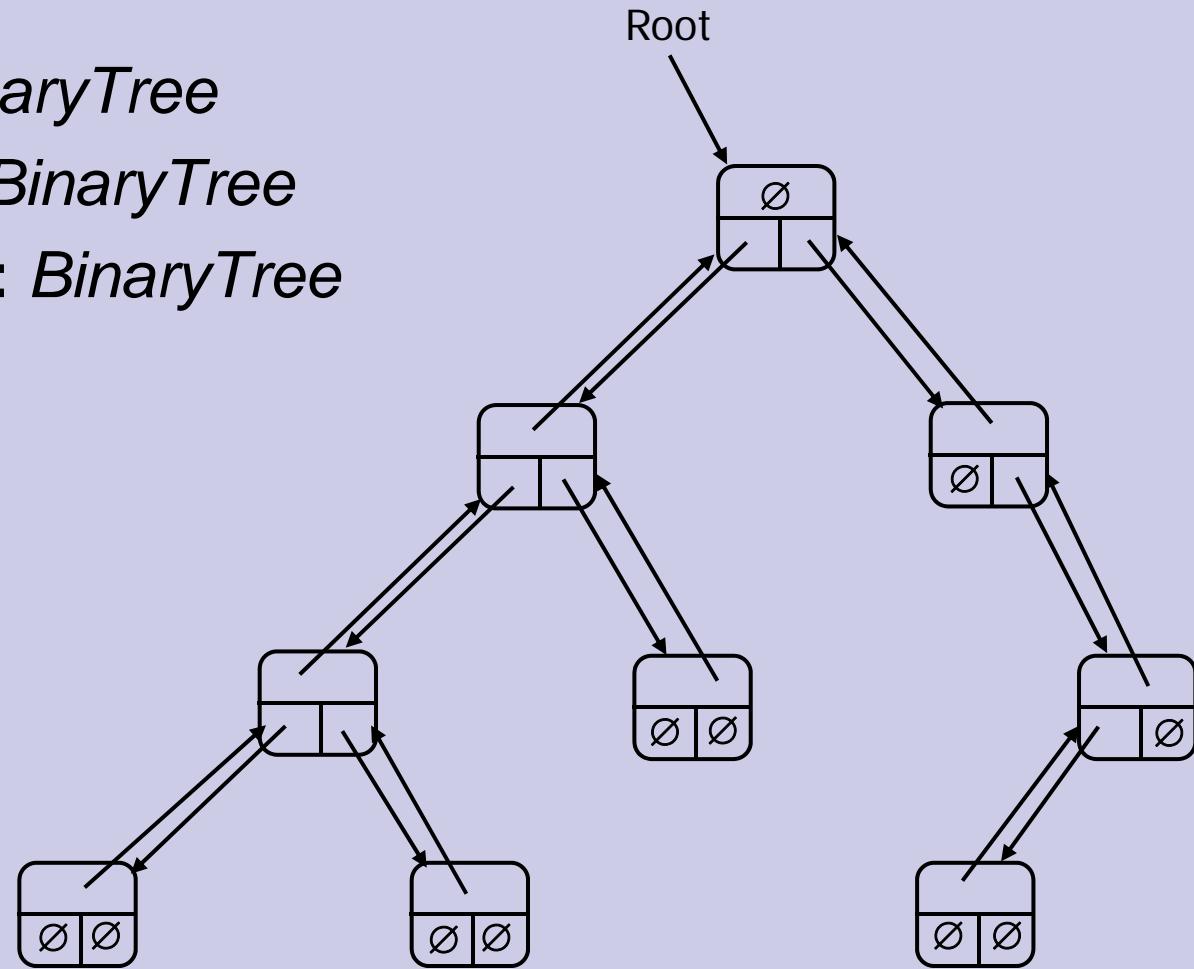
- $\text{key}[x]$ – key
- $\text{left}[x]$ – pointer to left child
- $\text{right}[x]$ – pt. to right child
- $p[x]$ – pt. to parent node



Representing Rooted Trees

BinaryTree:

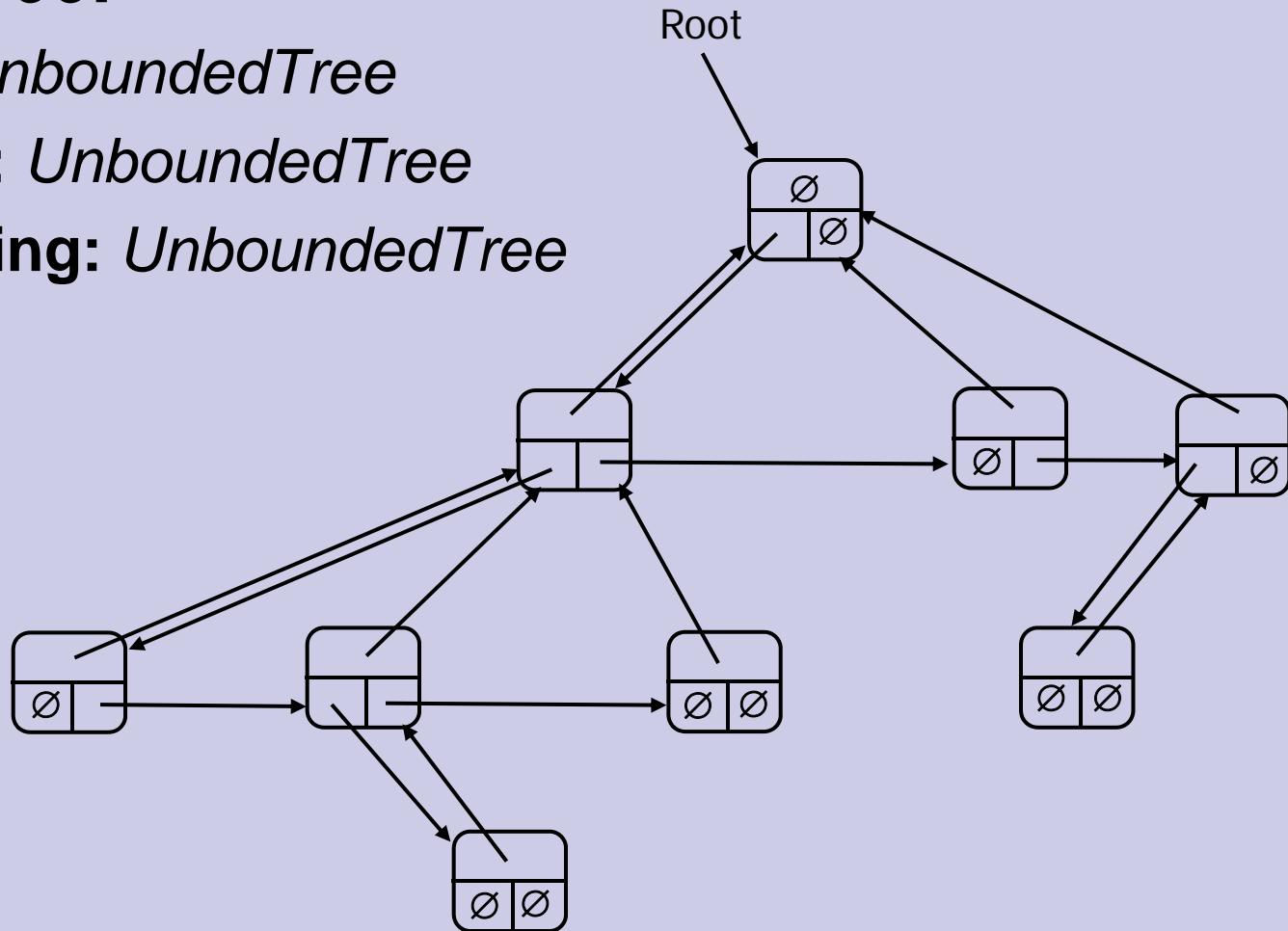
- **Parent**: *BinaryTree*
- **LeftChild**: *BinaryTree*
- **RightChild**: *BinaryTree*



Unbounded Branching

UnboundedTree:

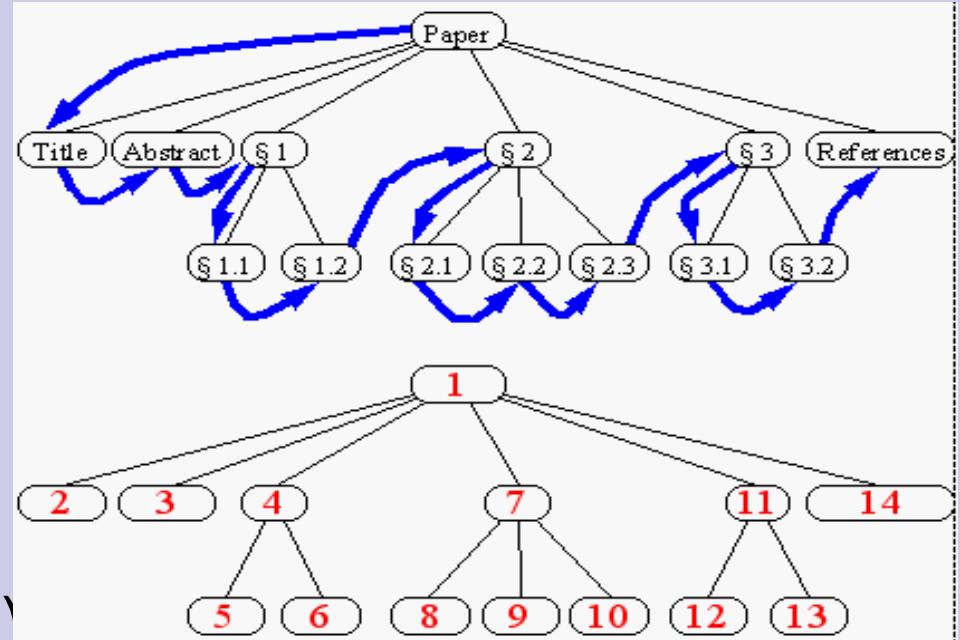
- **Parent**: *UnboundedTree*
- **LeftChild**: *UnboundedTree*
- **RightSibling**: *UnboundedTree*



Tree Walks/Traversals

- A tree walk or traversal is a way of visiting all the nodes in a tree in a specified order.
- A preorder tree walk processes each node before processing its children
- A postorder tree walk processes each node after processing its children

Traversing Trees (preorder)



- preorder traversal

Algorithm preOrder(v)

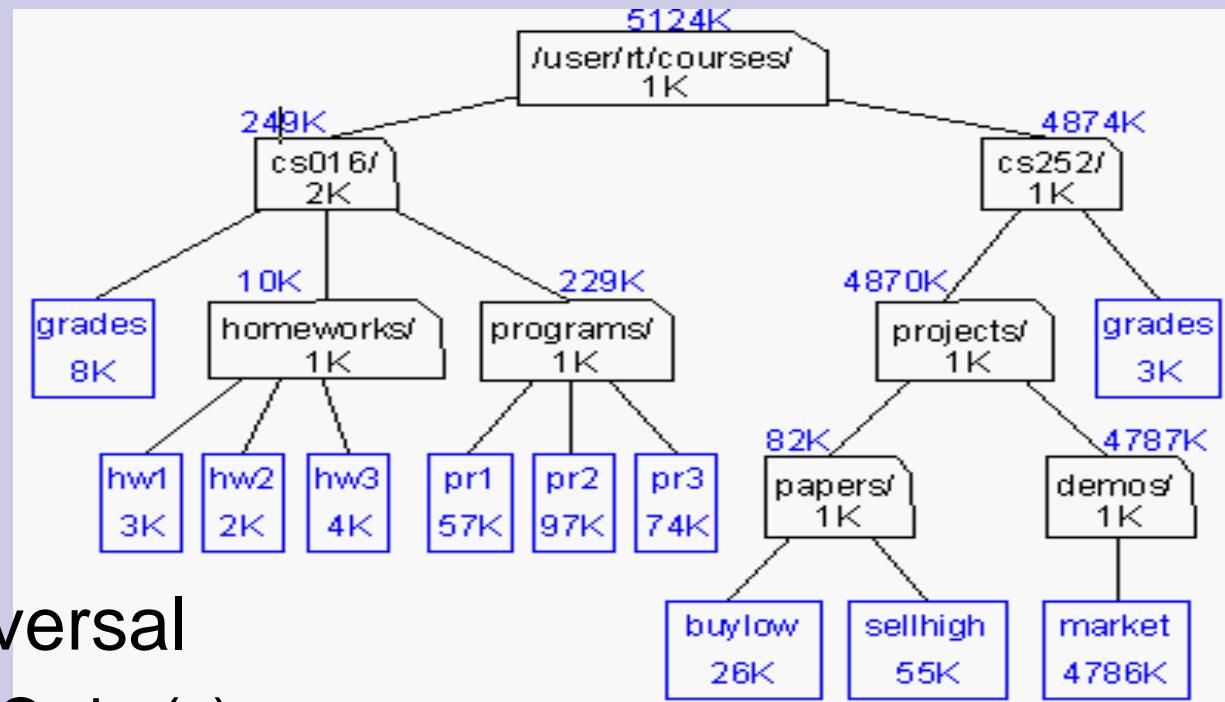
“visit” node v

for each child w of v do

recursively perform preOrder(w)

- reading a document from beginning to end

Traversing Trees (postorder)



- postorder traversal

Algorithm postOrder(v)

for each child w of v do

recursively perform postOrder(w)

“visit” node v

- du (disk usage) command in Unix

Traversals of Binary Trees

preorder(v)

```
if (v == null) then return  
else visit(v)  
    preorder(v.leftchild())  
    preorder(v.rightchild())
```

postorder(v)

```
if (v == null) then return  
else postorder(v.leftchild())  
    postorder(v.rightchild())  
    visit(v)
```

Examples of pre and postorder

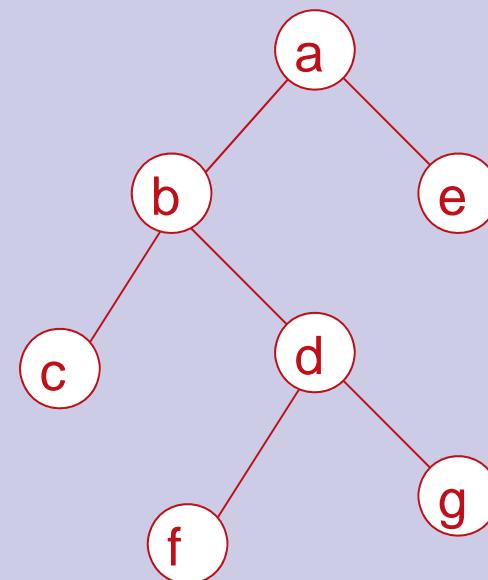
- We assume that we are only printing the data in a node when we visit it.

Preorder

a b c d f g e

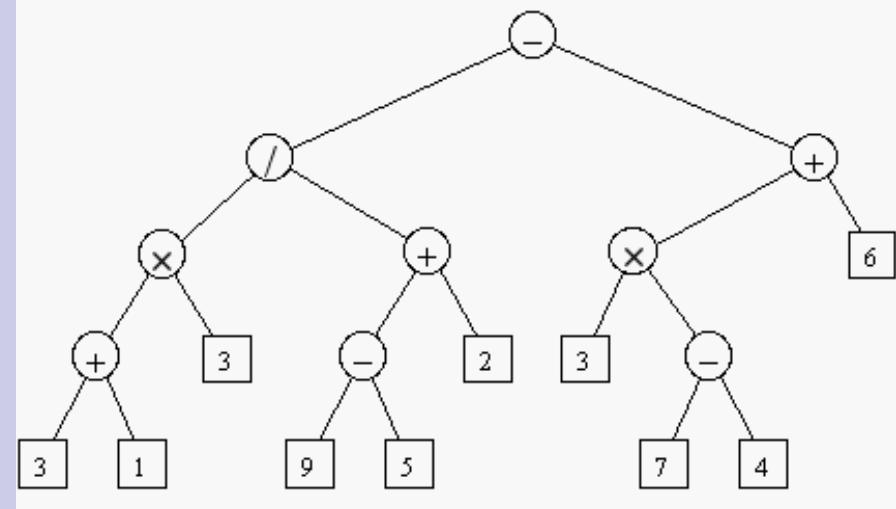
Postorder

c f g d b e a



Evaluating Arithmetic Expressions

- specialization of a postorder traversal



Algorithm evaluate(v)

if v is a leaf

return the variable stored at v

else

let o be the operator stored at v

$x \rightarrow \text{evaluate}(v.\text{leftChild}())$

$y \rightarrow \text{evaluate}(v.\text{rightChild}())$

return $x \circ y$

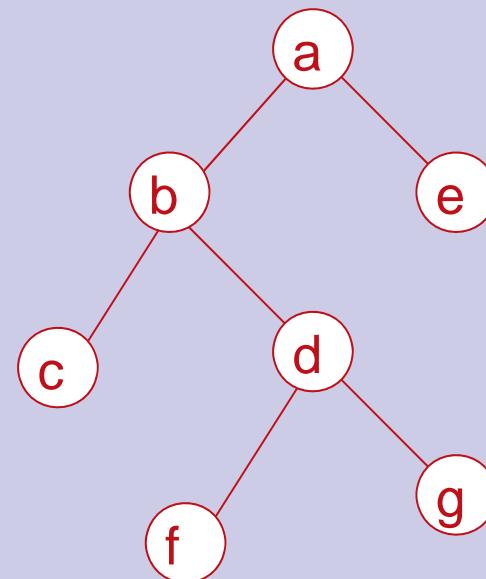
Traversing Trees

- Besides preorder and postorder, a third possibility arises when v is visited between the visit to the left ad right subtree.
- **Algorithm** inOrder(v)
 - if (v == null) then return
 - else inOrder(v.leftChild())
 - visit(v)
 - inOrder(v.rightChild())

Inorder Example

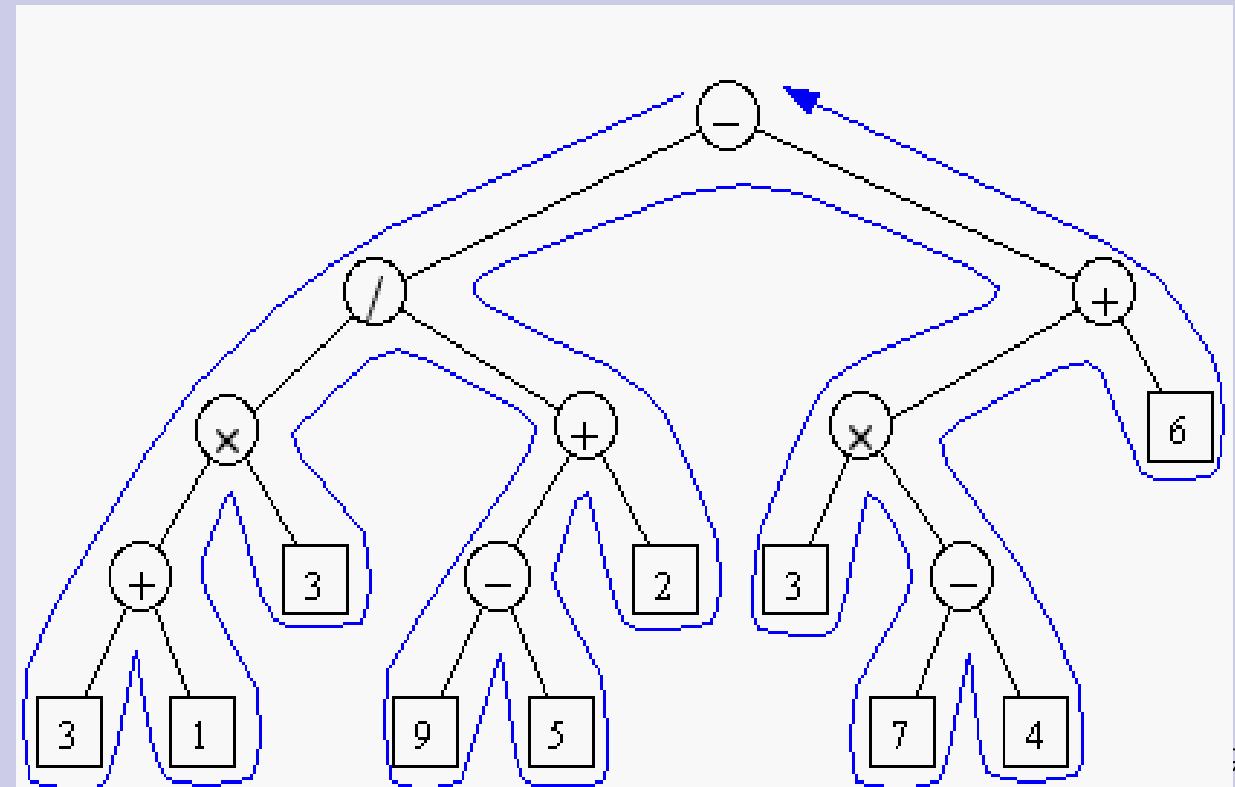
- Inorder

c b f d g a e



Euler Tour Traversal

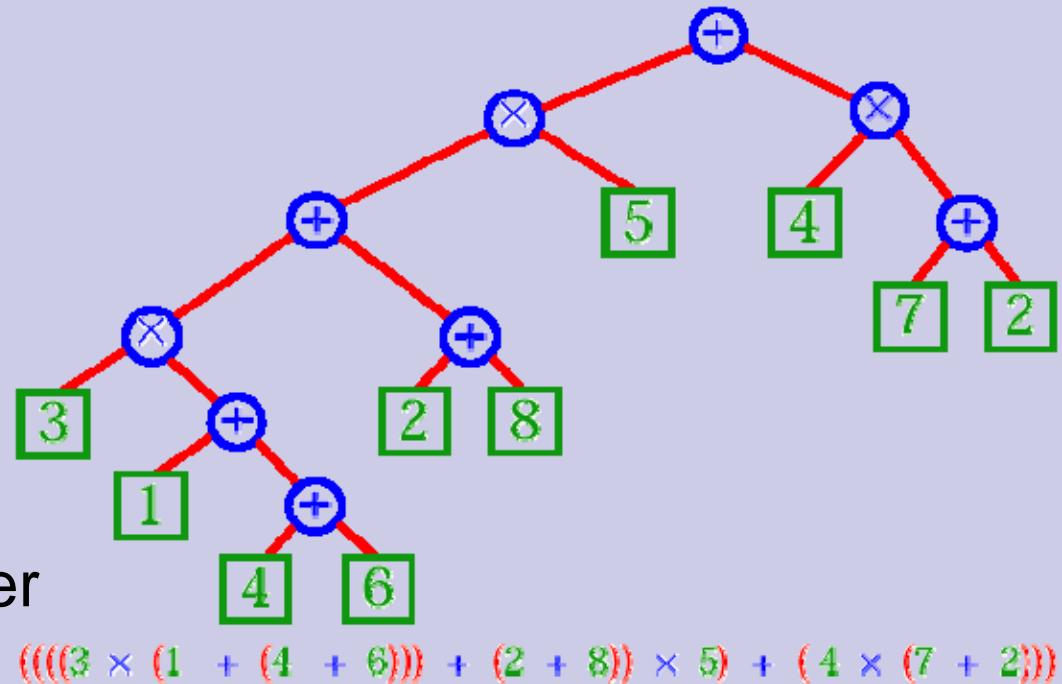
- generic traversal of a binary tree
- the preorder, inorder, and postorder traversals are special cases of the Euler tour traversal
- “walk around” the tree and visit each node three times:
 - on the left
 - from below
 - on the right



Printing an arithmetic expression

- Printing an arithmetic expression - so called Euler's walk:

- Print "(" before traversing the left subtree, traverse it
- Print the value of a node
- Traverse the right subtree, print ")" after traversing it



Template Method Pattern

- generic computation mechanism that can be specialized by redefining certain steps
- implemented by means of an abstract Java class with methods that can be redefined by its subclasses

```
public abstract class BinaryTreeTraversal {  
  
    protected BinaryTree tree;  
  
    ...  
  
    protected Object traverseNode(Position p) {  
        TraversalResult r = initResult();  
        if (tree.isExternal(p)) {  
            external(p, r);  
        } else {  
            left(p, r);  
            r.leftResult = traverseNode(tree.leftChild(p));  
            below(p, r);  
            r.rightResult = traverseNode(tree.rightChild(p));  
            right(p, r);  
        }  
        return result(r);  
    }  
}
```

Specializing Generic Binary Tree Traversal

- printing an arithmetic expression

```
public class PrintExpressionTraversal extends  
BinaryTreeTraversal {  
    ...  
    protected void external(Position p, TraversalResult r)  
    { System.out.print(p.element()); }  
    protected void left(Position p, TraversalResult r)  
    { System.out.print("("); }  
    protected void below(Position p, TraversalResult r)  
    { System.out.print(p.element()); }  
    protected void right(Position p, TraversalResult r)  
    { System.out.print(")"); }  
}
```

Building tree from pre- and in- order

- Given the preorder and inorder traversals of a binary tree we can uniquely determine the tree.

Preorder

a b c d f g e

b c d f g

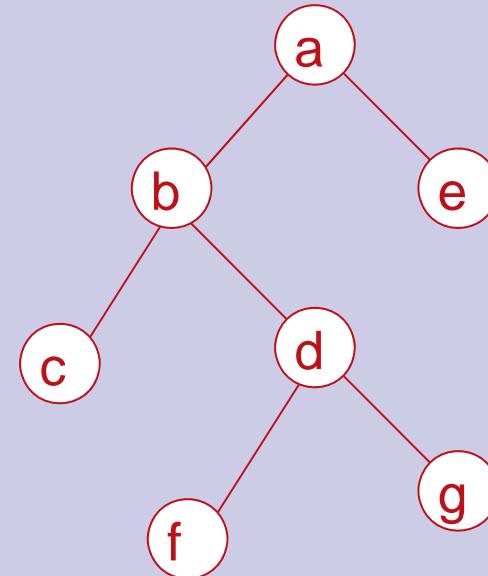
d f g

Inorder

c b f d g a e

c b f d g

f d g



Building tree from post and inorder

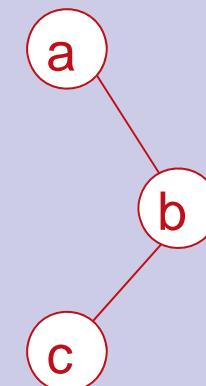
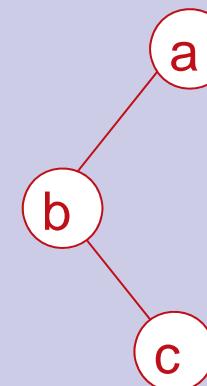
- In place of preorder we can use postorder.
- The last node visited in the postorder traversal is the root of the binary tree.
- This can then be used to split in the inorder traversal to identify the left and right subtrees.
- Procedure is similar to the one for obtaining tree from preorder and inorder traversals.

Insufficiency of pre & postorder

- Given the pre and postoder traversal of a binary tree we cannot uniquely identify the tree.
- This is because there can be two trees with the same pre and postorder traversals.

Preorder: a b c

Postorder: c b a



A special case

- If each internal node of the binary tree has at least two children then the tree can be determined from the pre and post order traversals.

Preorder

a	b	c	d	f	g	e
---	---	---	---	---	---	---

postorder

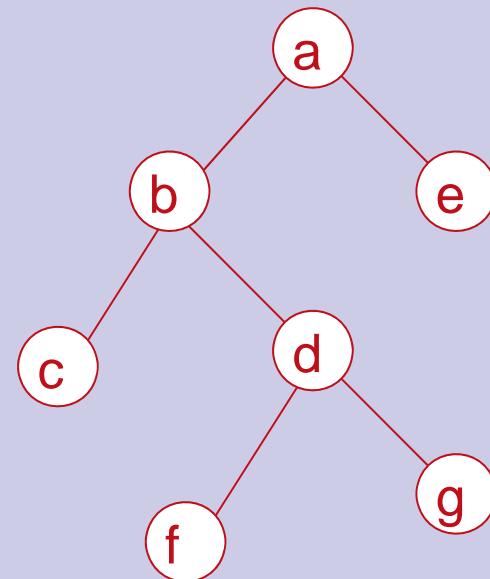
c	f	g	d	b	e	a
---	---	---	---	---	---	---

b	c	d	f	g
---	---	---	---	---

c	f	g	d	b
---	---	---	---	---

d	f	g
---	---	---

f	g	d
---	---	---



Ordered Dictionaries

- In addition to dictionary functionality, we want to support following operations:
 - **Min()**
 - **Max()**
 - **Predecessor(S, k)**
 - **Successor(S, k)**
- For this we require that there should be a total order on the keys.

A List-Based Implementation

- Unordered list

- searching takes $O(n)$ time
 - inserting takes $O(1)$ time



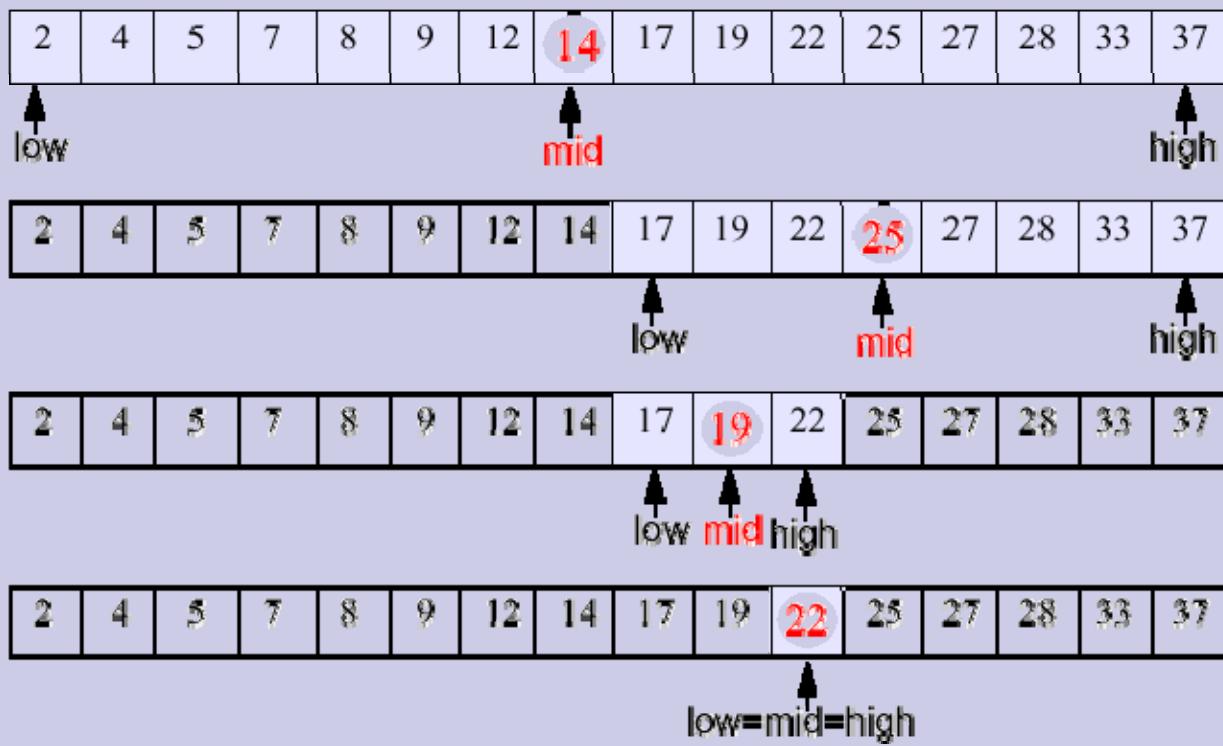
- Ordered list

- searching takes $O(n)$ time
 - inserting takes $O(n)$ time
 - Using array would definitely improve search time.



Binary Search

- ☐ Narrow down the search range in stages
 - ☐ `findElement(22)`

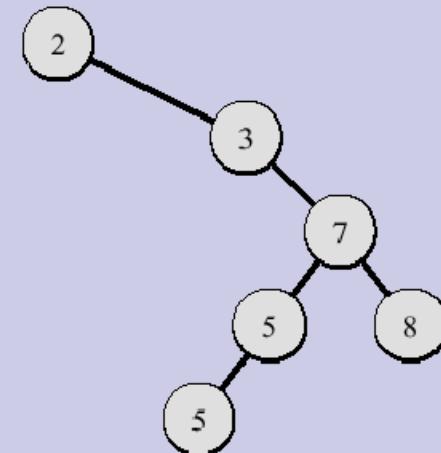
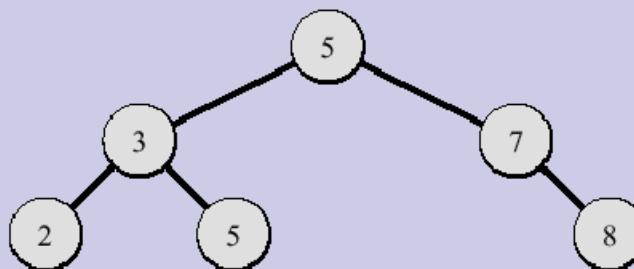


Running Time

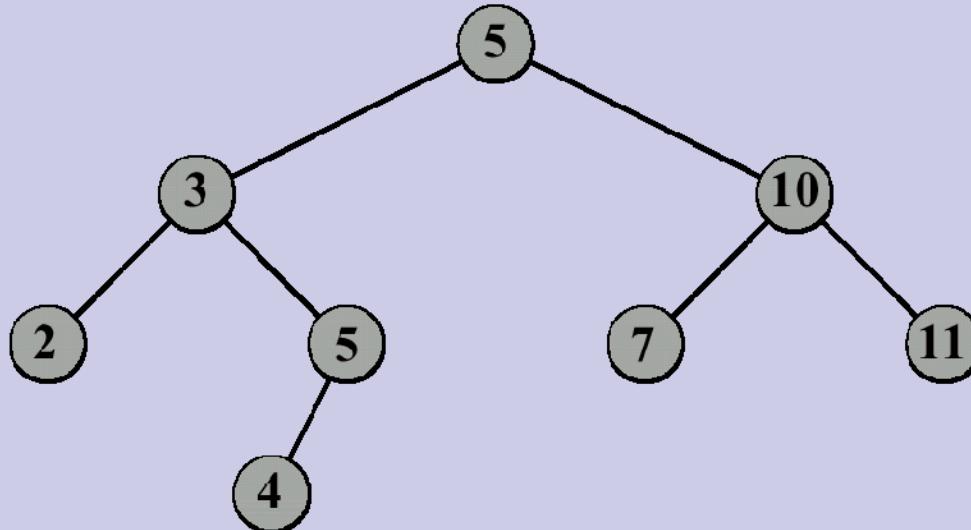
- The range of candidate items to be searched is halved after comparing the key with the middle element
- Binary search runs in $O(\lg n)$ time (remember recurrence...)
- What about insertion and deletion?

Binary Search Trees

- A binary search tree is a binary tree T such that
 - each internal node stores an item (k,e) of a dictionary
 - keys stored at nodes in the **left subtree** of v are **less than or equal** to k
 - keys stored at nodes in the **right subtree** of v are **greater than or equal** to k
- Example sequence 2,3,5,5,7,8



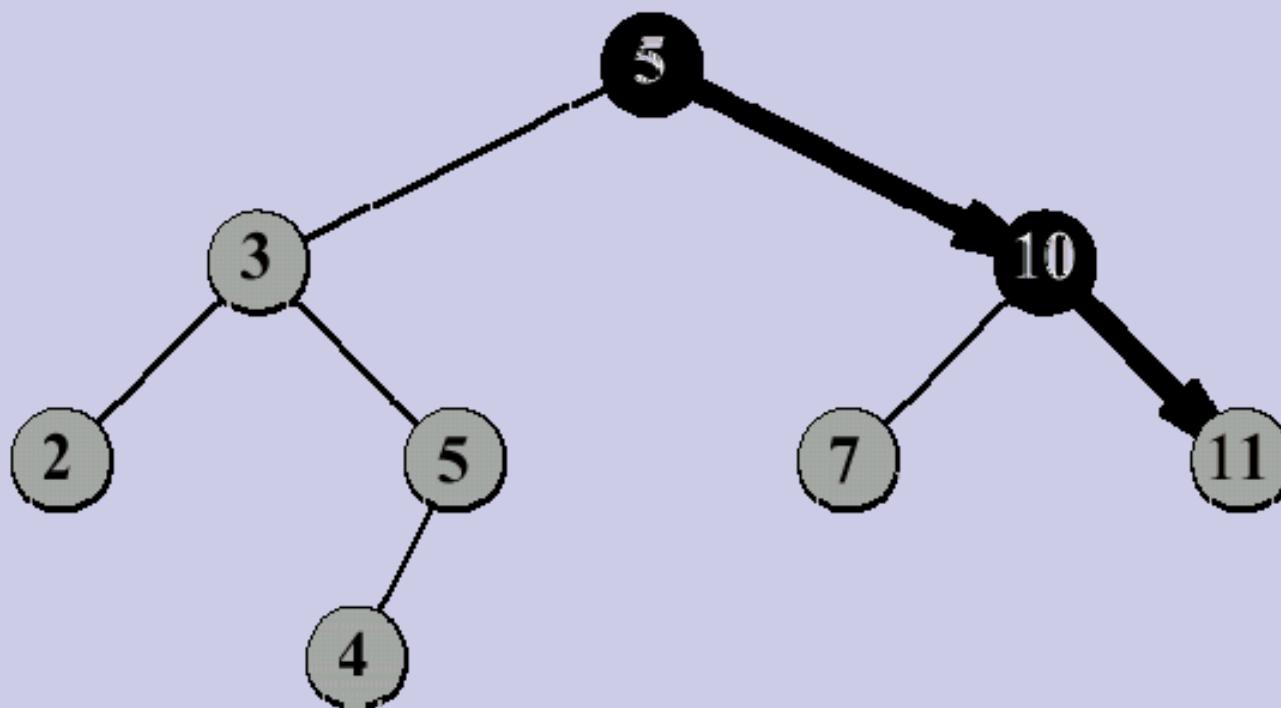
Searching a BST



- To find an element with key k in a tree T
 - compare k with $\text{key}[\text{root}[T]]$
 - if $k < \text{key}[\text{root}[T]]$, search for k in $\text{left}[\text{root}[T]]$
 - otherwise, search for k in $\text{right}[\text{root}[T]]$

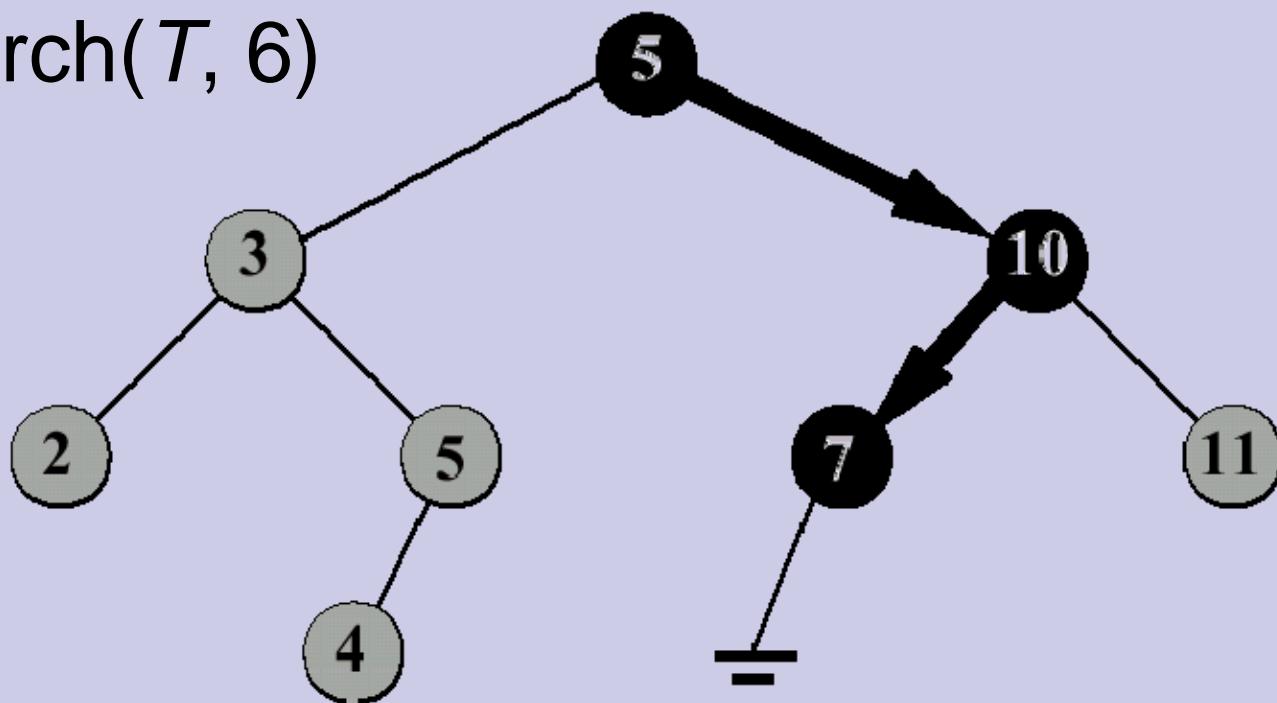
Search Examples

- Search($T, 11$)



Search Examples (2)

□ Search(T , 6)



Pseudocode for BST Search

□ Recursive version

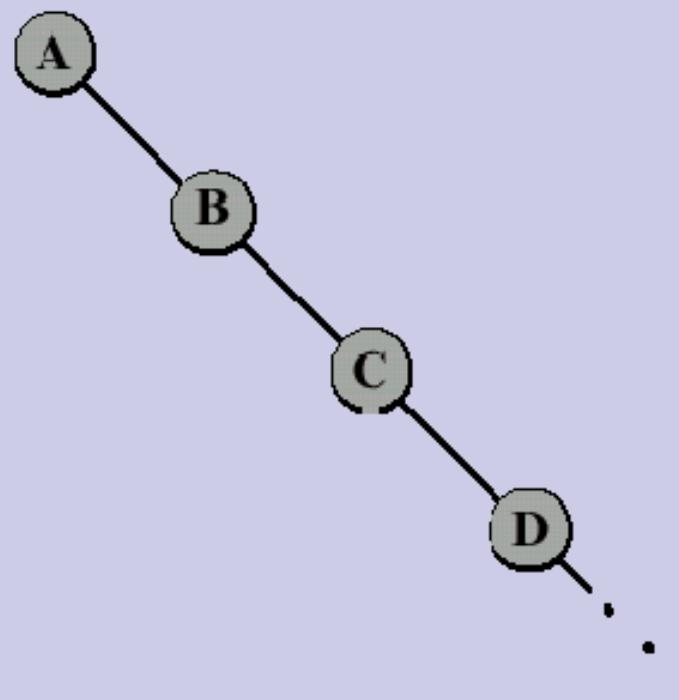
```
Search(T, k)
01 x ← root[T]
02 if x = NIL then return NIL
03 if k = key[x] then return x
04 if k < key[x]
05   then return Search(left[x], k)
06   else return Search(right[x], k)
```

□ Iterative version

```
Search(T, k)
01 x ← root[T]
02 while x ≠ NIL and k ≠ key[x] do
03   if k < key[x]
04     x ← left[x]
05   else x ← right[x]
06 return x
```

Analysis of Search

- Running time on tree of height h is $O(h)$
- After the insertion of n keys, the worst-case running time of searching is $O(n)$



BST Minimum (Maximum)

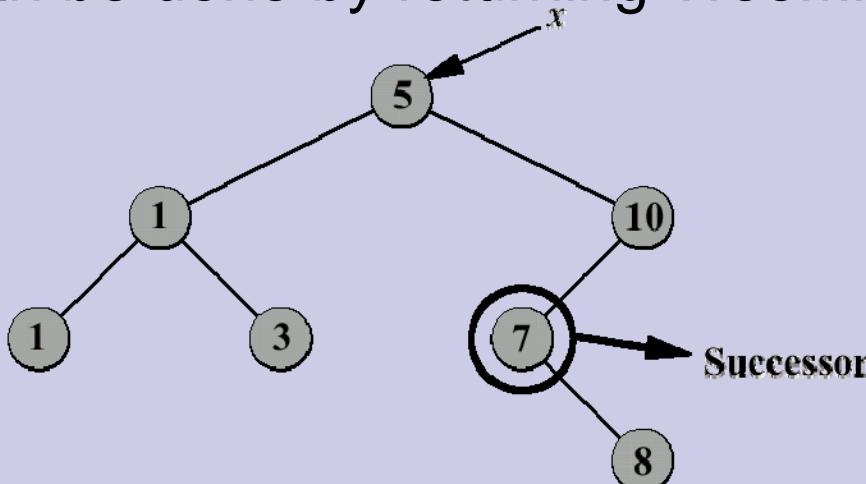
- Find the minimum key in a tree rooted at x

```
TreeMinimum(x)
01 while left[x] ≠ NIL
02   do x ← left[x]
03 return x
```

- Running time $O(h)$, i.e., it is proportional to the height of the tree

Successor

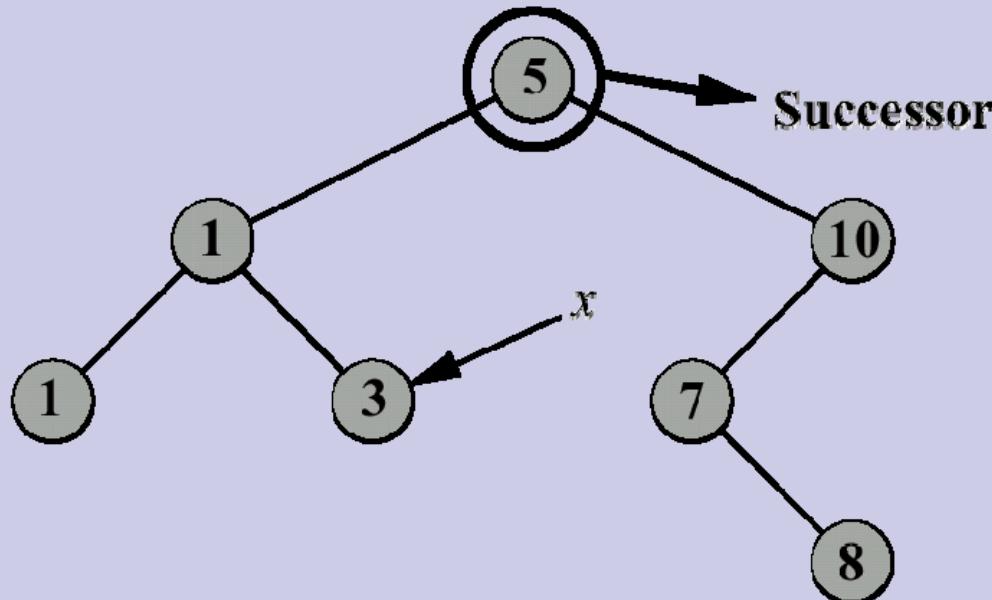
- Given x , find the node with the smallest key greater than $\text{key}[x]$
- We can distinguish two cases, depending on the right subtree of x
- Case 1
 - right subtree of x is nonempty
 - successor is leftmost node in the right subtree (Why?)
 - this can be done by returning $\text{TreeMinimum}(\text{right}[x])$



Successor (2)

□ Case 2

- the right subtree of x is empty
- successor is the lowest ancestor of x whose left child is also an ancestor of x (Why?)



Successor Pseudocode

```
TreeSuccessor(x)
01 if right[x] ≠ NIL
02     then return TreeMinimum(right[x])
03 y ← p[x]
04 while y ≠ NIL and x = right[y]
05     x ← y
06     y ← p[y]
03 return y
```

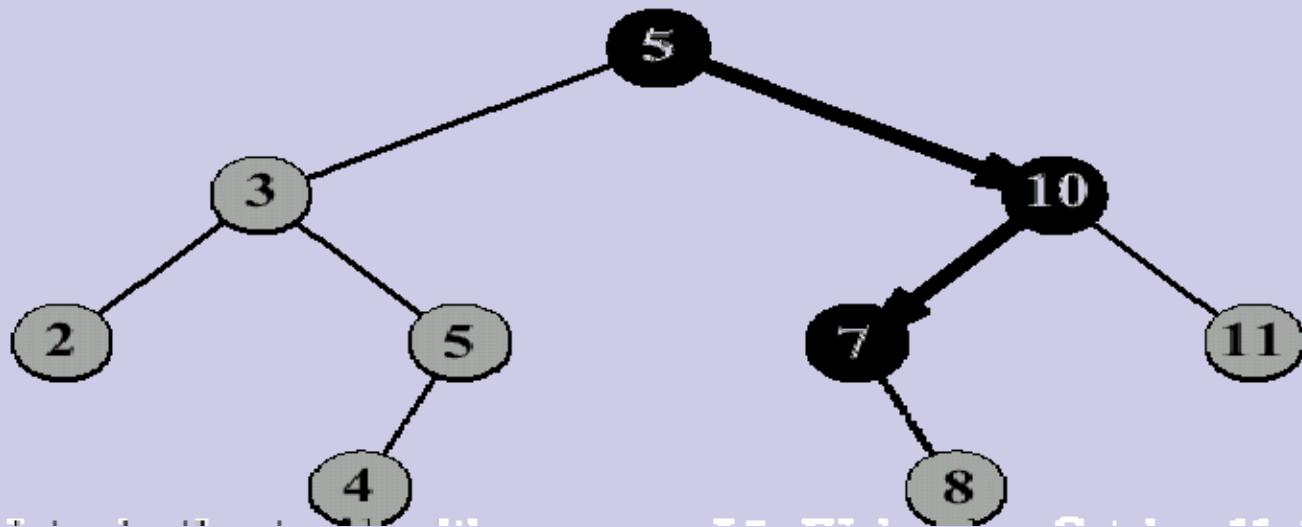
- For a tree of height h , the running time is $O(h)$

BST Insertion

- The basic idea is similar to searching
 - take an element z (whose left and right children are NIL) and insert it into T
 - find place in T where z belongs (as if searching for z),
 - and add z there
- The running time on a tree of height h is $O(h)$, i.e., it is proportional to the height of the tree

BST Insertion Example

□ Insert 8

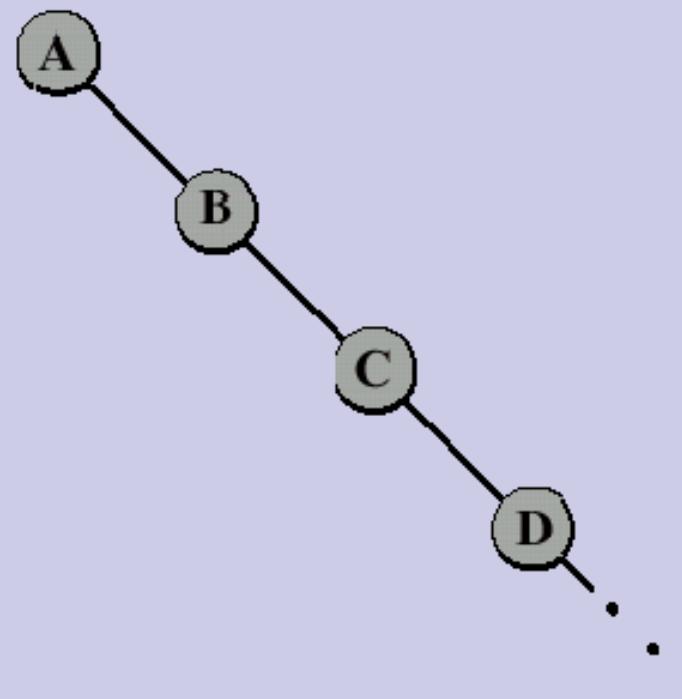


BST Insertion Pseudo Code

```
TreeInsert(T, z)
01 y ← NIL
02 x ← root[T]
03 while x ≠ NIL
04     y ← x
05     if key[z] < key[x]
06         then x ← left[x]
07         else x ← right[x]
08 p[z] ← y
09 if y = NIL
10    then root[T] ← z
11    else if key[z] < key[y]
12        then left[y] ← z
13        else right[y] ← z
```

BST Insertion: Worst Case

- In what sequence should insertions be made to produce a BST of height n ?

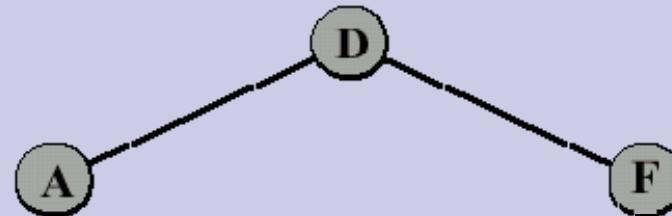
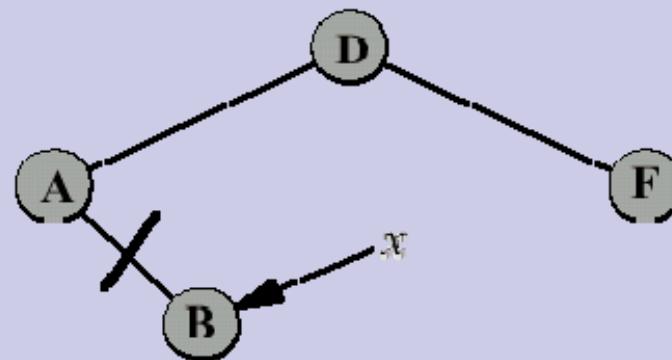


Deletion

- Delete node x from a tree T
- We can distinguish three cases
 - x has no children
 - x has one child
 - x has two children

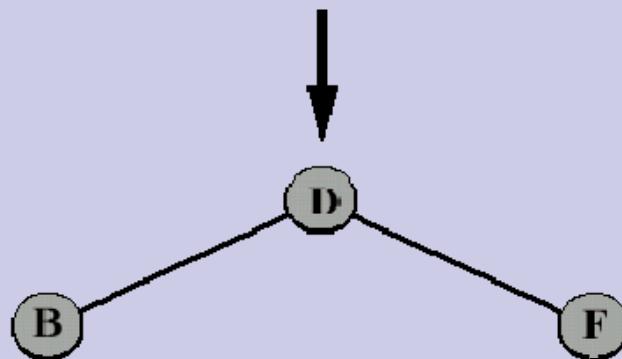
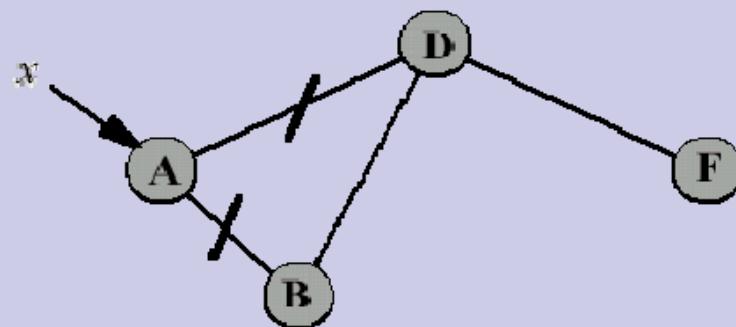
Deletion Case 1

- If x has no children – just remove x



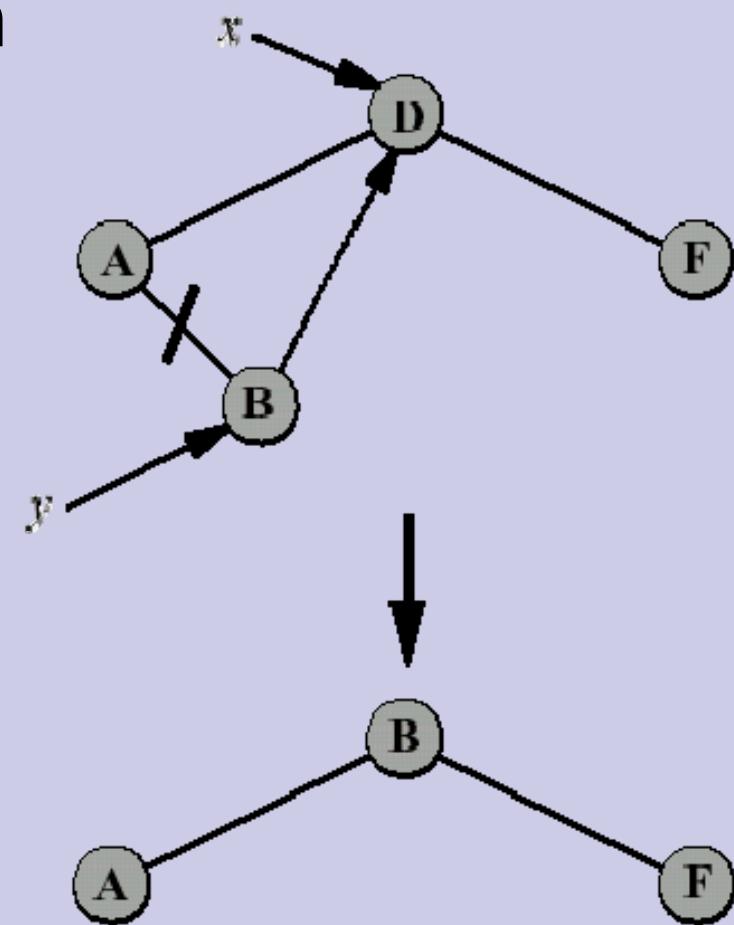
Deletion Case 2

- If x has exactly one child, then to delete x , simply make $p[x]$ point to that child



Deletion Case 3

- If x has two children, then to delete it we have to
 - find its successor (or predecessor) y
 - remove y (note that y has at most one child – why?)
 - replace x with y

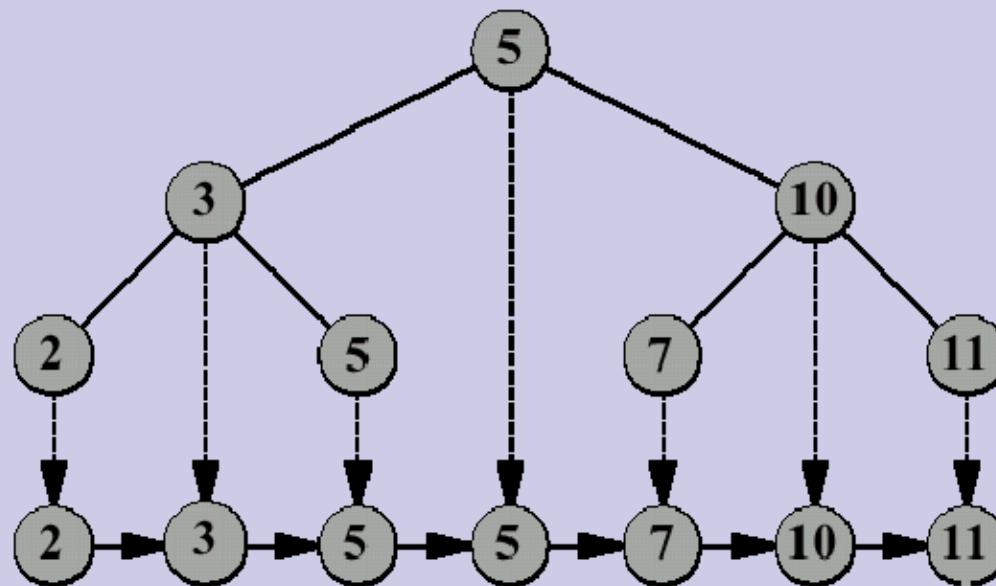


Delete Pseudocode

```
TreeDelete(T, z)
01 if left[z] = NIL or right[z] = NIL
02   then y ← z
03   else y ← TreeSuccessor(z)
04 if left[y] ≠ NIL
05   then x ← left[y]
06   else x ← right[y]
07 if x ≠ NIL
08   then p[x] ← p[y]
09 if p[y] = NIL
10   then root[T] ← x
11   else if y = left[p[y]]
12     then left[p[y]] ← x
13     else right[p[y]] ← x
14 if y ≠ z
15   then key[z] ← key[y] //copy all fileds of y
16 return y
```

In order traversal of a BST

- ITW can be thought of as a projection of the BST nodes onto a one dimensional interval



BST Sorting

- Use TreeInsert and InorderTreeWalk to sort a list of n elements, A

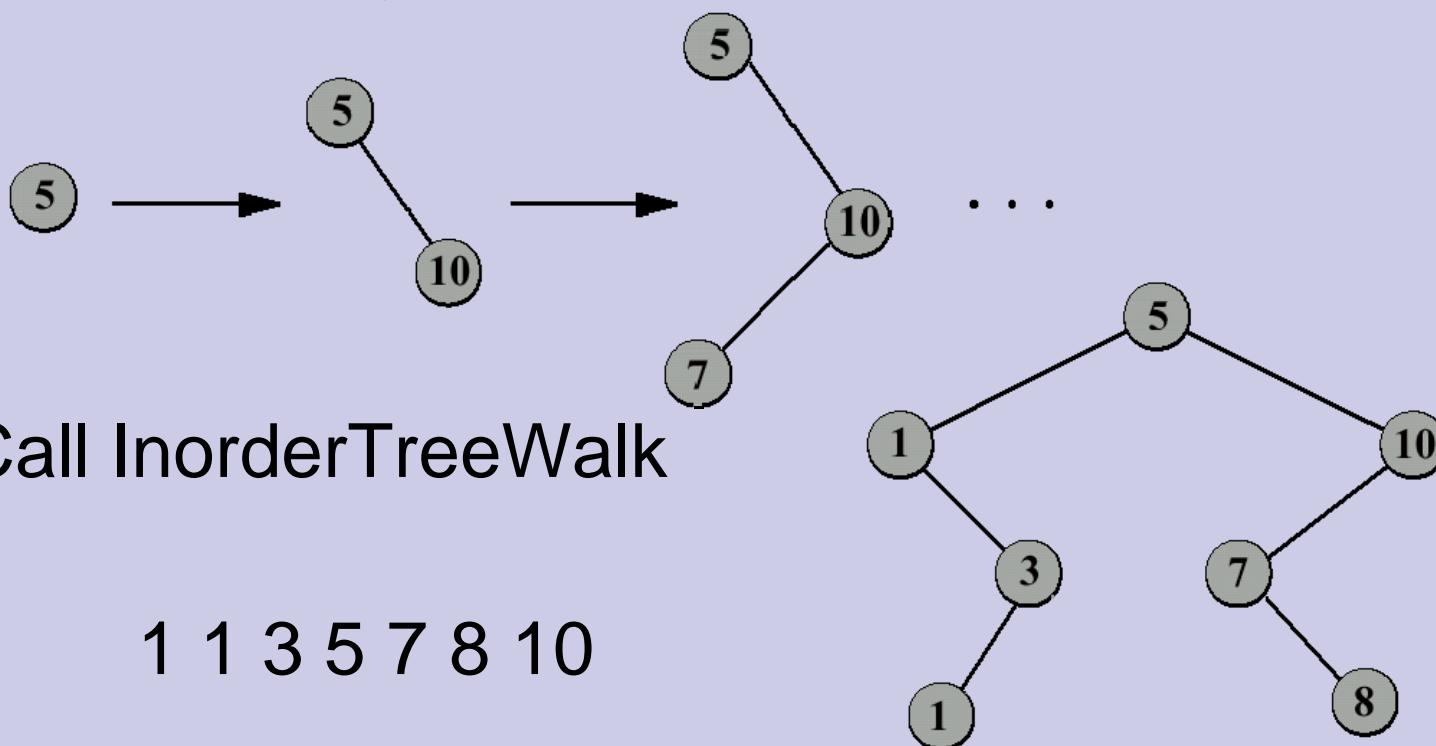
```
TreeSort(A)
01 root[T] ← NIL
02 for i ← 1 to n
03   TreeInsert(T,A[i])
04 InorderTreeWalk(root[T])
```

BST Sorting (2)

- Sort the following numbers

5 10 7 1 3 1 8

- Build a binary search tree



In what order should we insert?

- We want to sort numbers $\{1, 2, \dots, n\}$
- Total time taken to insert these numbers equals the sum of the level numbers of the nodes.
- Thus if numbers were inserted in ascending order we would get a tree of height $n-1$ in which there is one node at each level.
- So total time for insertion in this case is $1+2+3+\dots+n-1 = O(n^2)$.

Inserting a random permutation

- Suppose we take a random permutation of the keys and inserted them in this order.
- The total time required for insertion is now a random variable.
- We want to compute the expected value of this r.v.
- Recall that the expected value of a r.v. is the average value it takes over a large number of trials.

Expected insertion time for a random permutation

- We will compute the average time taken to insert keys in the order specified by the $n!$ permutations.
- In other words, for each of the $n!$ permutations we will compute the time taken to insert keys in that order and then compute the average.
- Let $T(n)$ denote this quantity.

Inserting a random permutation (2)

- Of the $n!$ permutations, there are $(n-1)!$ permutations in which the first element is i .
- The tree formed in these instances has i as the root. The left subtree has keys $1..(i-1)$ and the right subtree has keys $(i+1)..n$
- Consider the ordering of keys $1..(i-1)$ in the $(n-1)!$ permutations. All $(i-1)!$ permutations appear and each occurs $(n-1)!/(i-1)!$ times.

Inserting a random permutation(3)

- Recall that if we only had keys $1..(i-1)$ then average time taken to insert them is $T(i-1)$.
- The average is taken over all permutations of $1..(i-1)$.
- hence total time to insert all $(i-1)!$ permutations is $(i-1)!T(i-1)$.

Inserting a random permutation(4)

- When inserting keys $1..(i-1)$ into the left subtree, each key has to be compared with the root.
- This leads to an additional unit cost for each key.
- So total time to insert all $(i-1)!$ permutations is $(i-1)!(T(i-1)+(i-1))$.
- Since each permutation appears $(n-1)!/(i-1)!$ times, total time to insert keys $1..(i-1)$ is $(n-1)!(T(i-1)+(i-1))$

Inserting a random permutation(5)

- Time to insert keys $1..(i-1)$ is
 $(n-1)!(T(i-1)+(i-1))$
- Similarly, time to insert keys $(i+1)..n$ is
 $(n-1)!(T(n-i)+(n-i))$
- Total time to insert all n keys in permutations where the first key is i is
 $(n-1)! (T(i-1)+T(n-i)+ n-1)$
- Total time to insert all n keys in all $n!$ permutations is
 $(n-1)! \sum_{i=1}^n (T(i-1)+T(n-i)+ n-1).$

Building the recurrence

Average time to insert n keys is

$$\begin{aligned} T(n) &= \frac{1}{n} \sum_{i=1}^n (T(i-1) + T(n-i) + n-1) \\ &= \frac{2}{n} \sum_{i=0}^{n-1} T(i) + n - 1 \end{aligned}$$

Note that $T(0)=0$

- We are expressing the value of function $T()$ at point n in terms of the value of $T()$ at points $0..n-1$. This is called a **recurrence relation**.

Solving the recurrence

Since

$$T(n-1) = \frac{2}{n-1} \sum_{i=0}^{n-2} T(i) + n - 2$$

we have

$$\frac{2}{n} \sum_{i=0}^{n-2} T(i) = \frac{n-1}{n} (T(n-1) - n + 2)$$

Substituting in the expression for $T(n)$ we get

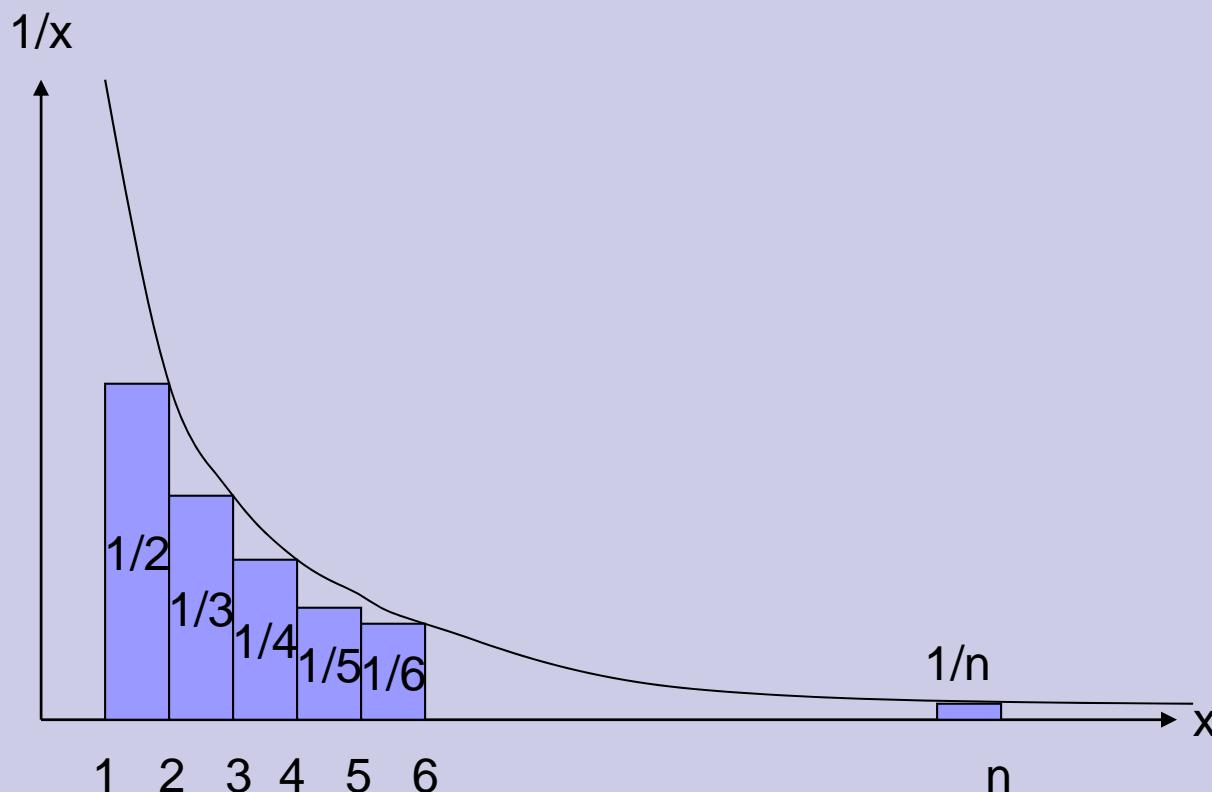
$$\begin{aligned} T(n) &= \frac{n-1}{n} (T(n-1) - n + 2) + \frac{2}{n} T(n-1) + n - 1 \\ &= \frac{n+1}{n} T(n-1) + \frac{2(n-1)}{n} \end{aligned}$$

Solving the recurrence(2)

$$\begin{aligned} T(n) &= \frac{n+1}{n}T(n-1) + 2(n-1)/n \\ &\leq (1 + 1/n)T(n-1) + 2 \\ &\leq \frac{n+1}{n} \left(\frac{n}{n-1}T(n-2) + 2 \right) + 2 \\ &= \frac{n+1}{n-1}T(n-2) + \frac{2(n+1)}{n} + 2 \\ &\leq \frac{n+1}{n-2}T(n-3) + 2(n+1) \left(\frac{1}{n} + \frac{1}{n-1} \right) + 2 \\ &\leq \frac{n+1}{n-3}T(n-4) + 2(n+1) \left(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} \right) + 2 \\ &\leq (n+1)T(0) + 2(n+1) \left(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{2} \right) + 2 \\ &= 2(n+1) \left(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{2} \right) + 2 \end{aligned}$$

Summing the harmonic series

- What is the sum $\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$?



It is at most the area under the curve $f(x)=1/x$ between limits 1 and n

The expected time for insertion

$$\begin{aligned} T(n) &\leq 2(n+1) \int_1^n \frac{dx}{x} + 2 \\ &= 2(n+1) \ln n + 2 \\ &= O(n \log n) \end{aligned}$$

Thus the expected time for inserting a randomly chosen permutation of n keys is $O(n \log n)$

Minimum time to insert n keys

- The time required to insert n keys is minimum when the resulting tree has the smallest possible height.
- A binary tree on n nodes has height at least $\log_2 n$
- To insert the $n/2$ nodes at level $\log_2 n$ we require at least $(n/2)\log_2 n$ time.
- Hence inserting n keys requires $\Omega(n\log_2 n)$ time.

Summary of Running times

- To insert n keys into a binary search tree which is initially empty requires
- $O(n^2)$ time in the worst case.
- $O(n \log n)$ time in the best case.
- $O(n \log n)$ time in the average case; the average is taken over the $n!$ different orders in which the n keys could be inserted.

Quick Sort

- Characteristics
 - sorts almost in "place," i.e., does not require an additional array
 - very practical, average sort performance $O(n \log n)$ (with small constant factors), but worst case $O(n^2)$

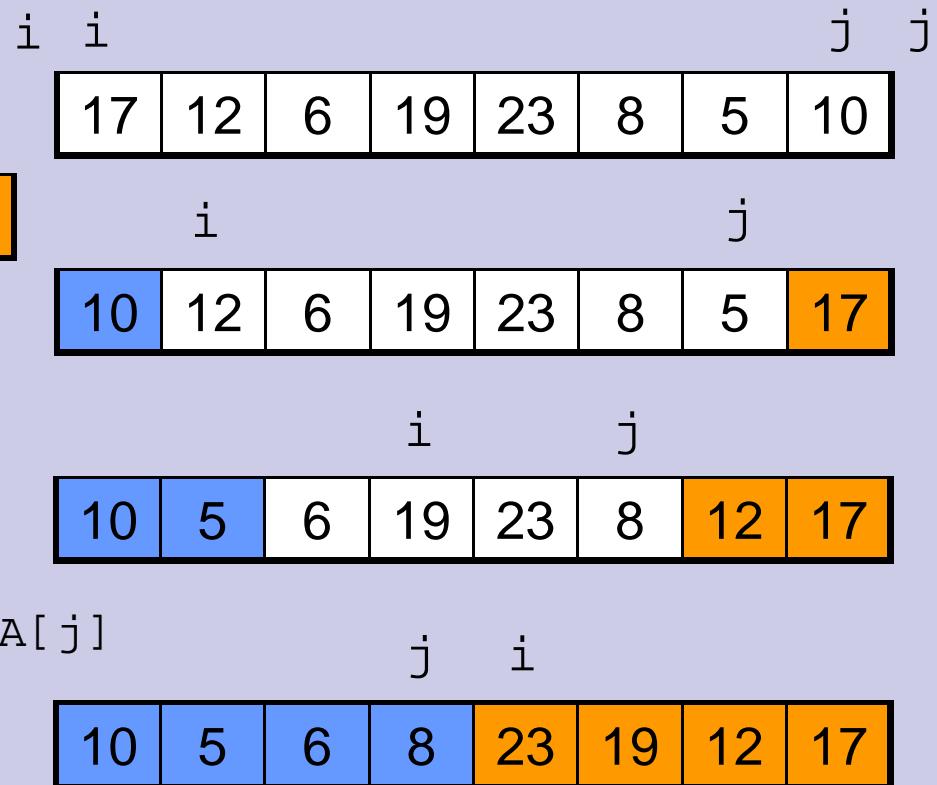
Quick Sort – the Principle

- To understand quick-sort, let's look at a high-level description of the algorithm
- A divide-and-conquer algorithm
 - **Divide**: partition array into 2 subarrays such that elements in the lower part \leq elements in the higher part
 - **Conquer**: recursively sort the 2 subarrays
 - **Combine**: trivial since sorting is done in place

Partitioning

□ Linear time partitioning procedure

```
Partition(A,p,r)
01 x←A[r]
02 i←p-1
03 j←r+1
04 while TRUE
05   repeat j←j-1
06     until A[j] ≤x
07   repeat i←i+1
08     until A[i] ≥x
09   if i < j
10     then exchange A[i]↔A[j]
11   else return j
```



Quick Sort Algorithm

- Initial call **Quicksort(A, 1, length[A])**

```
Quicksort(A,p,r)
```

```
01  if p < r  
02      then q ← Partition(A, p, r)  
03          Quicksort(A, p, q)  
04          Quicksort(A, q+1, r)
```

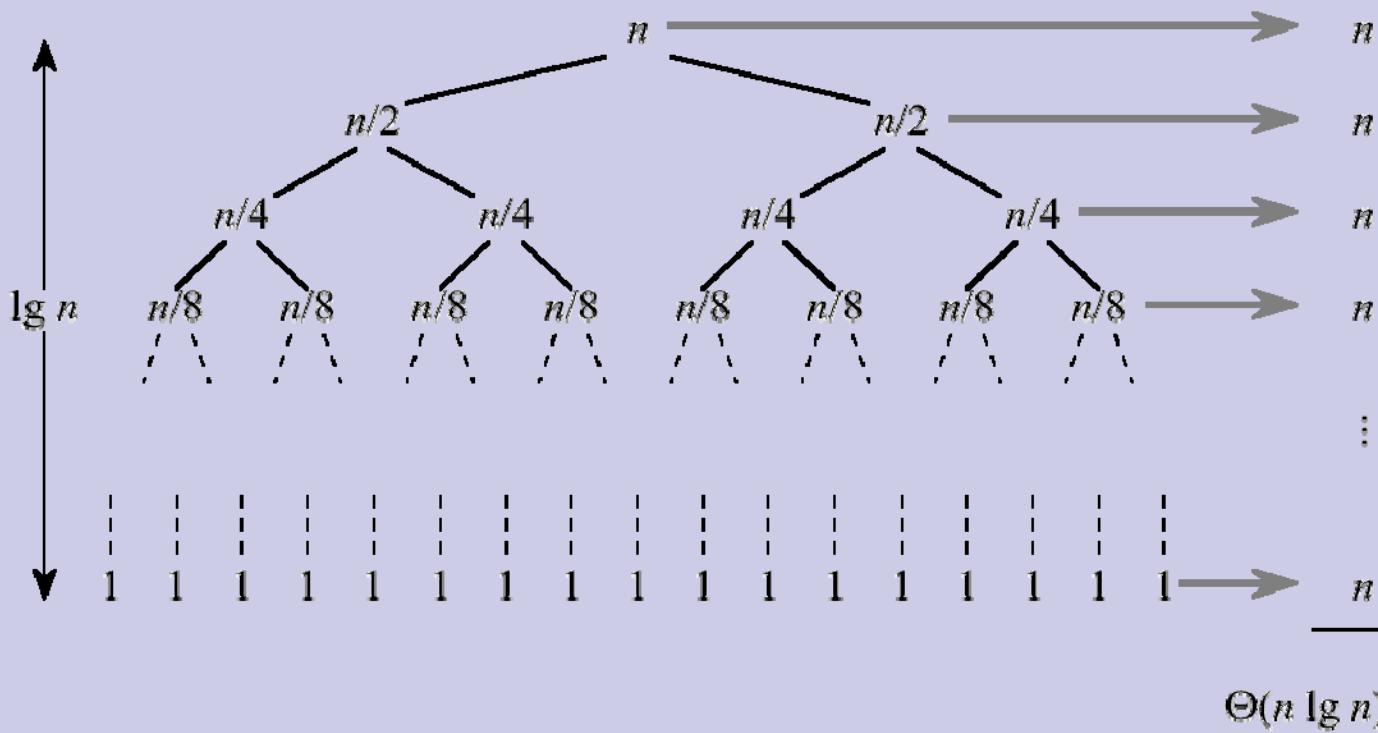
Analysis of Quicksort

- Assume that all input elements are distinct
- The running time depends on the distribution of splits

Best Case

- If we are lucky, Partition splits the array evenly

$$T(n) = 2T(n/2) + \Theta(n)$$



Worst Case

- What is the worst case?
- One side of the partition has only one element

$$T(n) = T(1) + T(n - 1) + \Theta(n)$$

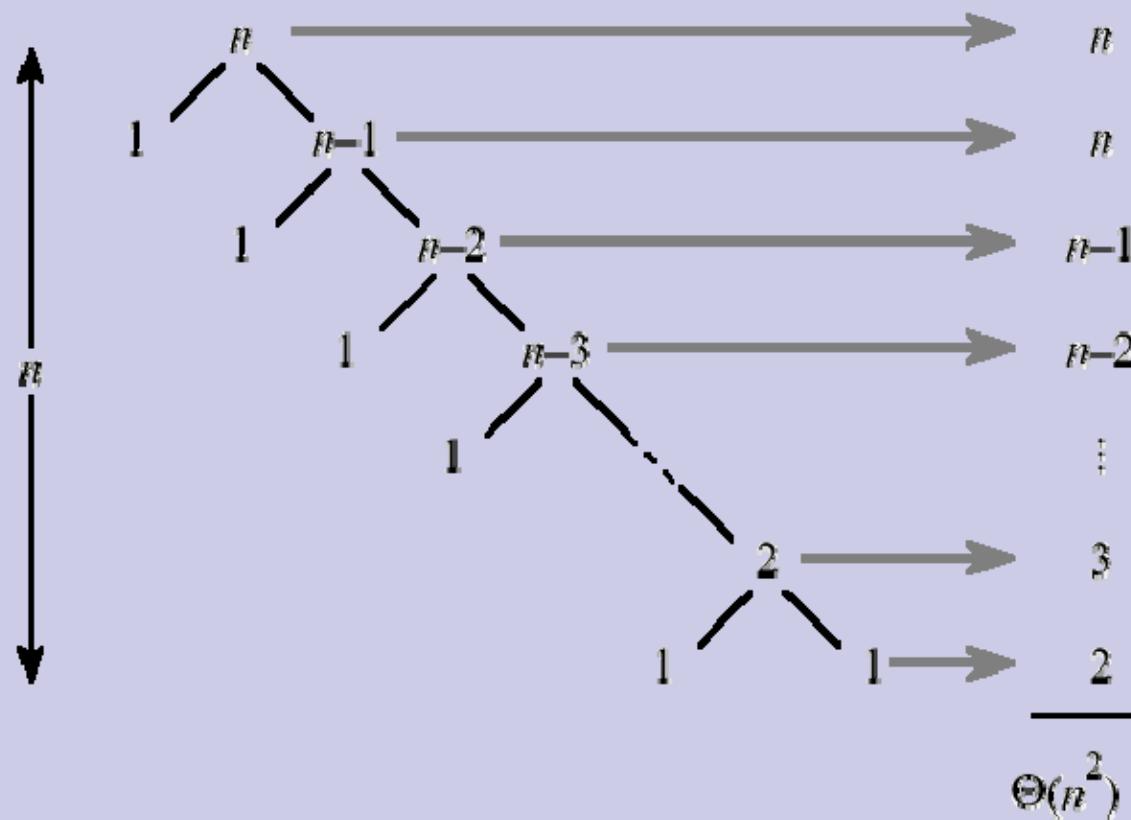
$$= T(n - 1) + \Theta(n)$$

$$= \sum_{k=1}^n \Theta(k)$$

$$= \Theta\left(\sum_{k=1}^n k\right)$$

$$= \Theta(n^2)$$

Worst Case (2)



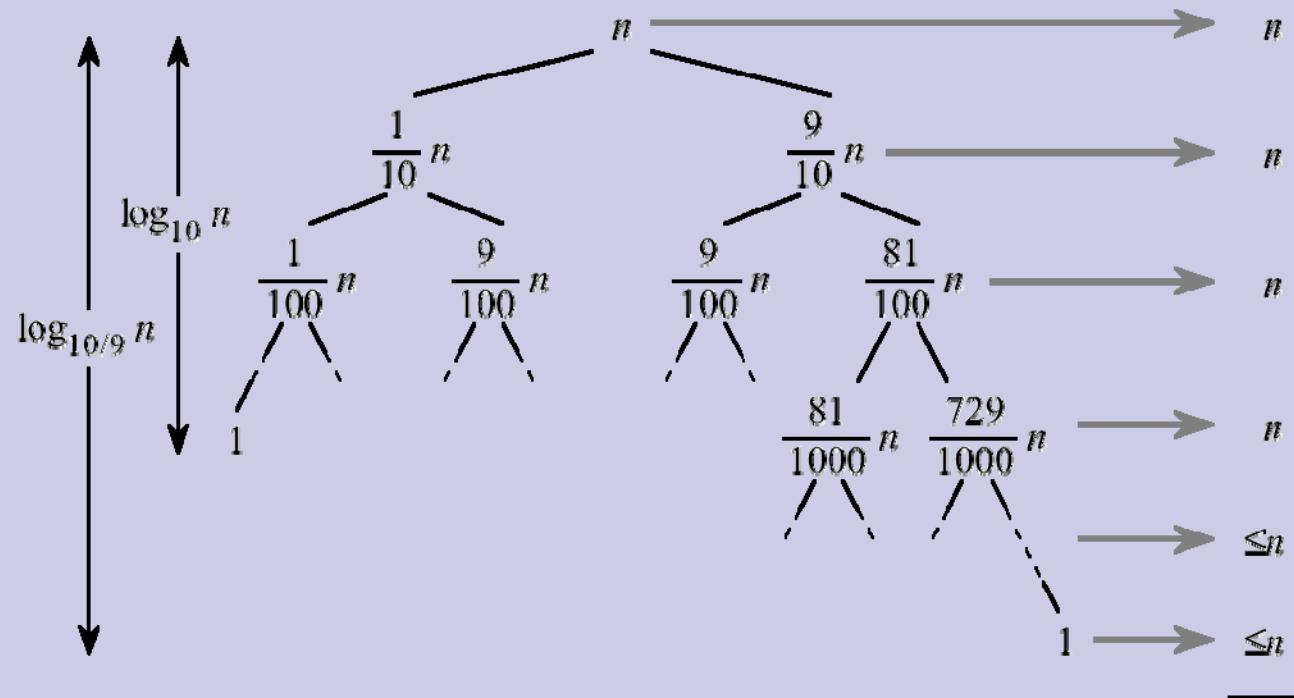
Worst Case (3)

- When does the worst case appear?
 - input is sorted
 - input reverse sorted
- Same recurrence for the worst case of insertion sort
- However, sorted input yields the best case for insertion sort!

Analysis of Quicksort

- Suppose the split is 1/10 : 9/10

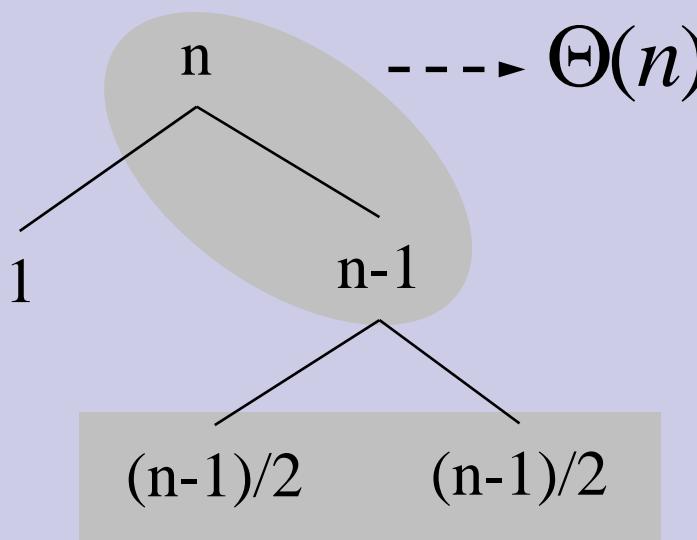
$$T(n) = T(n/10) + T(9n/10) + \Theta(n) = \Theta(n \log n)!$$



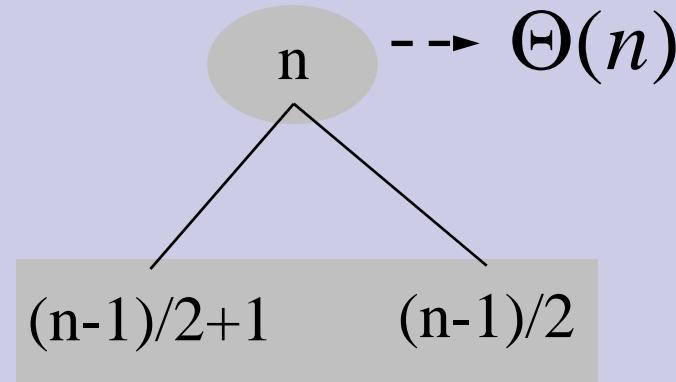
$\Theta(n \lg n)$

An Average Case Scenario

- Suppose, we alternate lucky and unlucky cases to get an average behavior



$$\begin{aligned}L(n) &= 2U(n/2) + \Theta(n) \quad \text{lucky} \\U(n) &= L(n-1) + \Theta(n) \quad \text{unlucky} \\&\text{we consequently get} \\L(n) &= 2(L(n/2 - 1) + \Theta(n/2)) + \Theta(n) \\&= 2L(n/2 - 1) + \Theta(n) \\&= \Theta(n \log n)\end{aligned}$$



An Average Case Scenario (2)

- How can we make sure that we are usually lucky?
 - Partition around the "middle" ($n/2$ th) element?
 - Partition around a random element (works well in practice)
- Randomized algorithm
 - running time is independent of the input ordering
 - no specific input triggers worst-case behavior
 - the worst-case is only determined by the output of the random-number generator

Randomized Quicksort

- Assume all elements are distinct
- Partition around a random element
- Consequently, all splits ($1:n-1$, $2:n-2$, ..., $n-1:1$) are equally likely with probability $1/n$

- Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity

Randomized Quicksort (2)

Randomized-Partition(A,p,r)

```
01 i ← Random(p,r)
02 exchange A[r] ↔ A[i]
03 return Partition(A,p,r)
```

Randomized-Quicksort(A,p,r)

```
01 if p < r then
02     q ← Randomized-Partition(A,p,r)
03     Randomized-Quicksort(A,p,q)
04     Randomized-Quicksort(A,q+1,r)
```

Randomized Quicksort Analysis

- Let $T(n)$ be the expected number of comparisons needed to quicksort n numbers.
- Since each split occurs with probability $1/n$, $T(n)$ has value $T(i-1)+T(n-i)+n-1$ with probability $1/n$.
- Hence,

$$\begin{aligned} T(n) &= \frac{1}{n} \sum_{j=1}^n (T(j-1) + T(n-j) + n-1) \\ &= \frac{2}{n} \sum_{j=0}^{n-1} T(j) + n - 1 \end{aligned}$$

Randomized Quicksort Analysis(2)

- We have seen this recurrence before.
- It is the recurrence for the expected number of comparisons required to insert a randomly chosen permutation of n elements.
- We proved that $T(n) = O(n \log_2 n)$.
- Hence expected number of comparisons required by randomized quicksort is $O(n \log_2 n)$

Randomized Quicksort running times

- Worst case running time of quicksort is $O(n^2)$
- Best case running time of quicksort is $O(n \log_2 n)$
- Expected running time of quicksort is $O(n \log_2 n)$

What does expected running time mean?

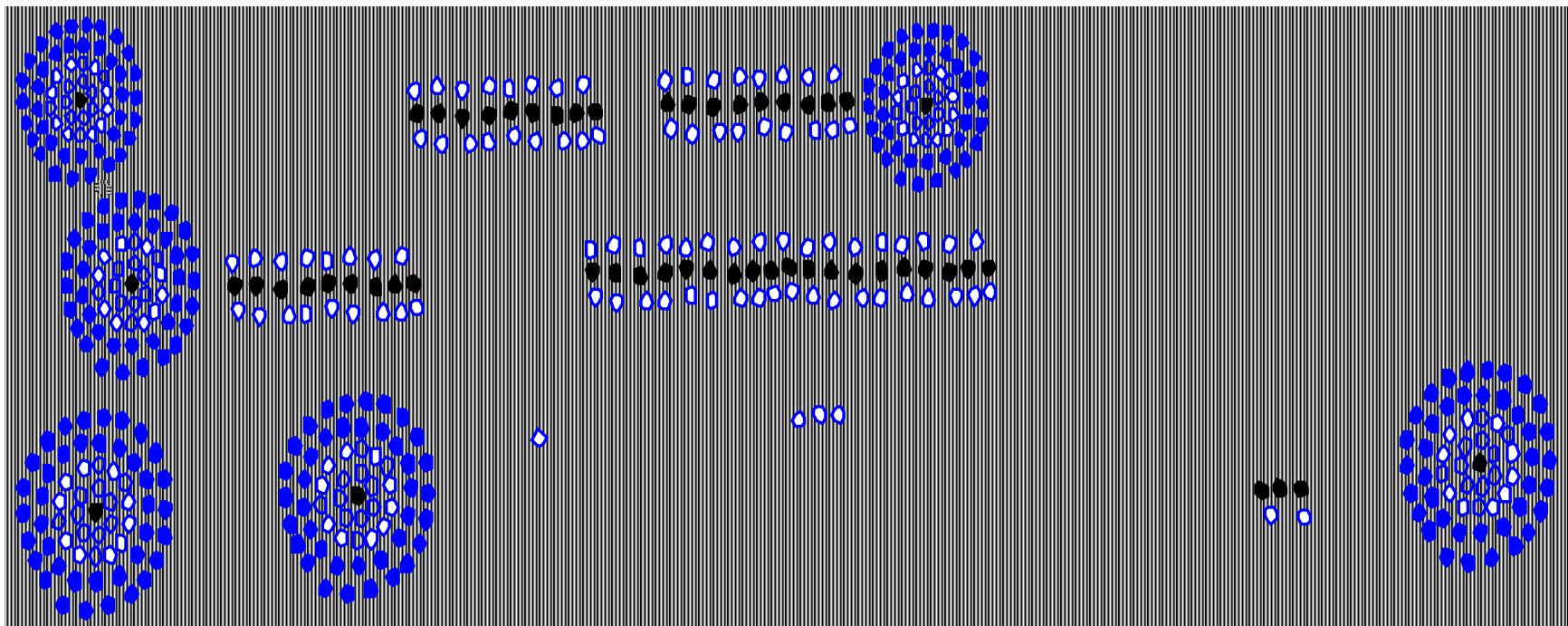
- The running time of quicksort does not depend on the input. It depends on the random numbers provided by the generator.
- Thus for the same input the program might take 3sec today and 5sec tomorrow.
- The average time taken over many different runs of the program would give us the expected time.
- Same as saying that we are taking average over all possible random number sequences provided by the generator.

Analysis of insertion in BST

- When creating a binary search tree on n elements the running time does depend on the order of the elements.
- Our algorithm for insertion did not employ random bits.
- Given a specific input order the algorithm takes the same time each day.
- However, the time taken is different for different input orders.
- The average time taken over all possible input orders is $O(n \log_2 n)$.

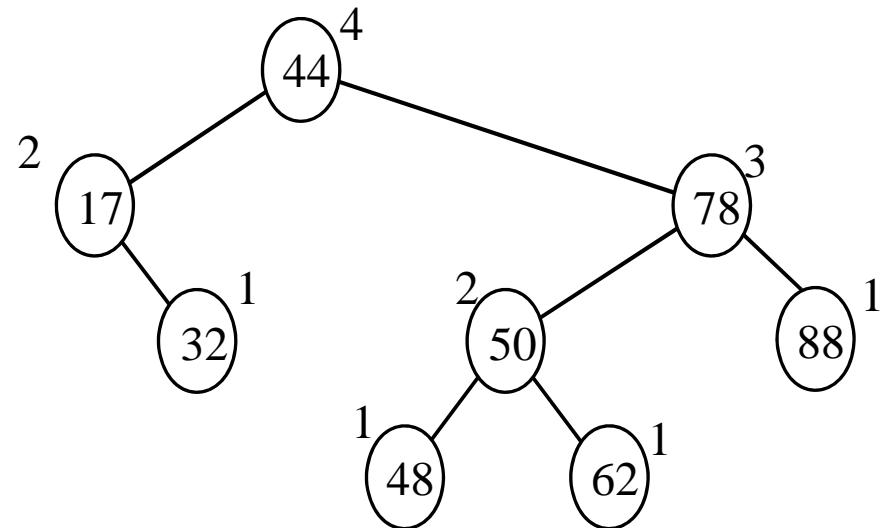
AVL Trees

□ AVL Trees



AVL Tree

- AVL trees are balanced.
- An AVL Tree is a ***binary search tree*** such that for every internal node v of T , the ***heights of the children of v can differ by at most 1***.



An example of an AVL tree where the heights are shown next to the nodes:

Height of an AVL Tree

- **Proposition:** The *height* of an AVL tree T storing n keys is $O(\log n)$.
- **Justification:** The easiest way to approach this problem is to find $n(h)$: the *minimum number of nodes* in an AVL tree of height h.
- We see that $n(1) = 1$ and $n(2) = 2$
- For $h \geq 3$, an AVL tree of height h contains the root node, one AVL subtree of height $h-1$ and the other AVL subtree of height $h-1$ or $h-2$.
- i.e. $n(h) = 1 + n(h-1) + n(h-2)$

Height of an AVL Tree (2)

- Knowing $n(h-1) \geq n(h-2)$, we get

$$n(h) = n(h-1) + n(h-2) + 1 > 2n(h-2)$$

$$n(h) > 2n(h-2)$$

$$> 4n(h-4)$$

$$> 8n(h-6)$$

...

$$> 2^i n(h-2i)$$

- When $i = h/2 - 1$ we get: $n(h) > 2^{h/2-1} n(2) = 2^{h/2}$
- Taking logarithms: $h < 2\log n(h)$
- Thus the height of an AVL tree is $O(\log n)$

A sharper bound

- We will show how to obtain a sharper bound on the height of an AVL tree.
- We prove using induction that the minimum number of nodes in an AVL tree of height h , $n(h) \geq c^h$, where c is some number > 1 .
- Base case: $h=1$. Now $n(h) \geq c > 1$.
- Suppose claim is true for all $h < k$
- We have to show that $n(k) \geq c^k$

Sharper bound (2)

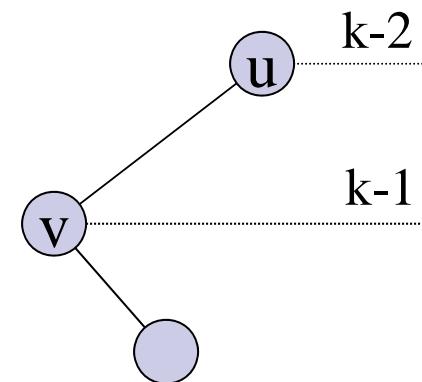
- $n(k) = n(k-1) + n(k-2) + 1$
 $\geq c^{k-1} + c^{k-2}$ (by induction hypothesis)
- We will be able to show that $n(k) \geq c^k$ if we can show that $c^{k-1} + c^{k-2} \geq c^k$.
- So c should be such that $c^2 - c - 1 \leq 0$.
- The quadratic equation $c^2 - c - 1 = 0$ has roots $\frac{1-\sqrt{5}}{2}$ and $\frac{1+\sqrt{5}}{2}$.
- Hence we can take c as $\frac{1+\sqrt{5}}{2}$ which is roughly 1.63
- Hence AVL tree on n nodes has height atmost $\log_{1.63} n$

Structure of an AVL tree

- Consider an AVL tree on n nodes.
- Consider a leaf which is closest to the root.
- Suppose this leaf is at level k .
- We will show that the height of the tree is at most $2k-1$.

Structure of an AVL tree (2)

- Claim: Since closest leaf is at level k all nodes at levels $1..k-2$ have 2 children.
- Proof is by contradiction
- Suppose node u at level $k-2$ has only 1 child, v .
- v is at level $k-1$ and so cannot be a leaf.
- Hence subtree rooted at v has height at least 2.
- Height-balanced property is violated at u



Structure of an AVL tree (3)

- By previous claim, all levels 1 to $k-1$ are full.
- Hence tree has at least 2^{k-1} nodes.
- Since height of tree is at most $2k-1$ it has at most 2^{2k-1} nodes.
- Thus $2^{k-1} \leq n \leq 2^{2k-1}$
- Substituting h for $2k-1$ we get

$$2^{(h-1)/2} \leq n \leq 2^h$$

Summary of AVL tree structure

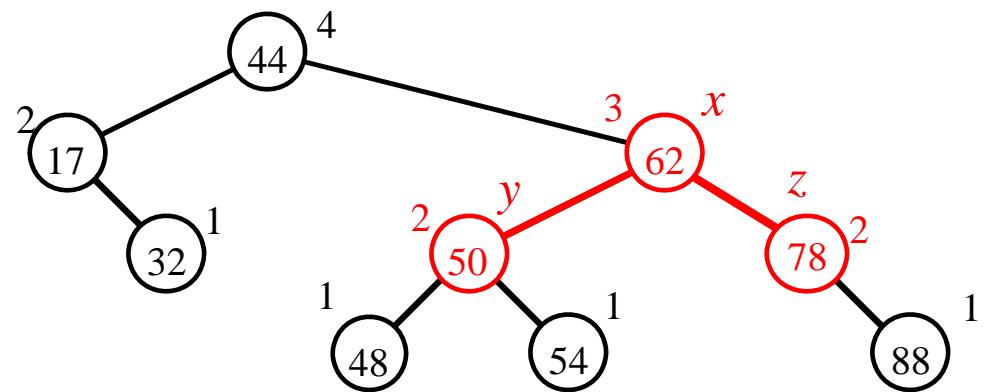
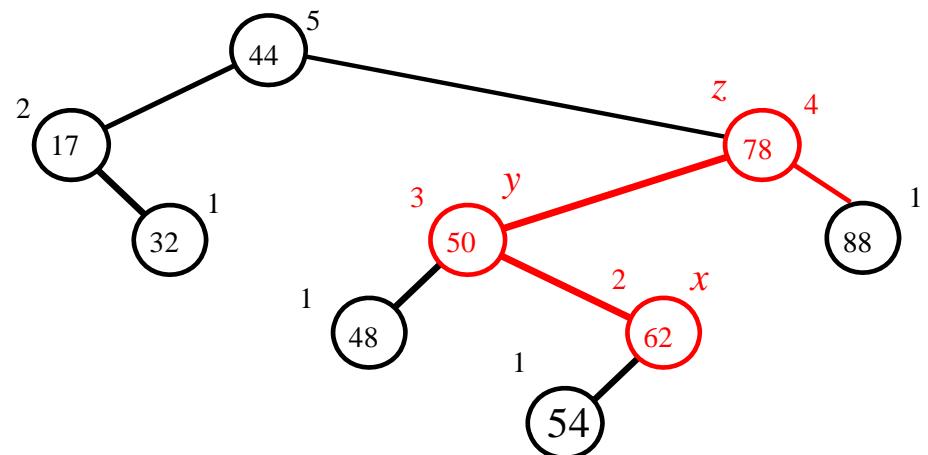
- In an AVL tree of height h , the leaf closest to the root is at level at least $(h+1)/2$.
- On the first $(h-1)/2$ levels the AVL tree is a complete binary tree.
- After $(h-1)/2$ levels the AVL tree may start “thinning out”.
- Number of nodes in the AVL tree is at least $2^{(h-1)/2}$ and at most 2^h

Insertion

- A binary tree T is called **height-balanced** if for every node v , height of v 's children differ by atmost one.
- Inserting a node into an AVL tree changes the heights of some of the nodes in T .
- If insertion causes T to become **unbalanced**, we travel up the tree from the newly created node until we find the first node x such that its grandparent z is unbalanced node.
- Let y be the parent of node x .

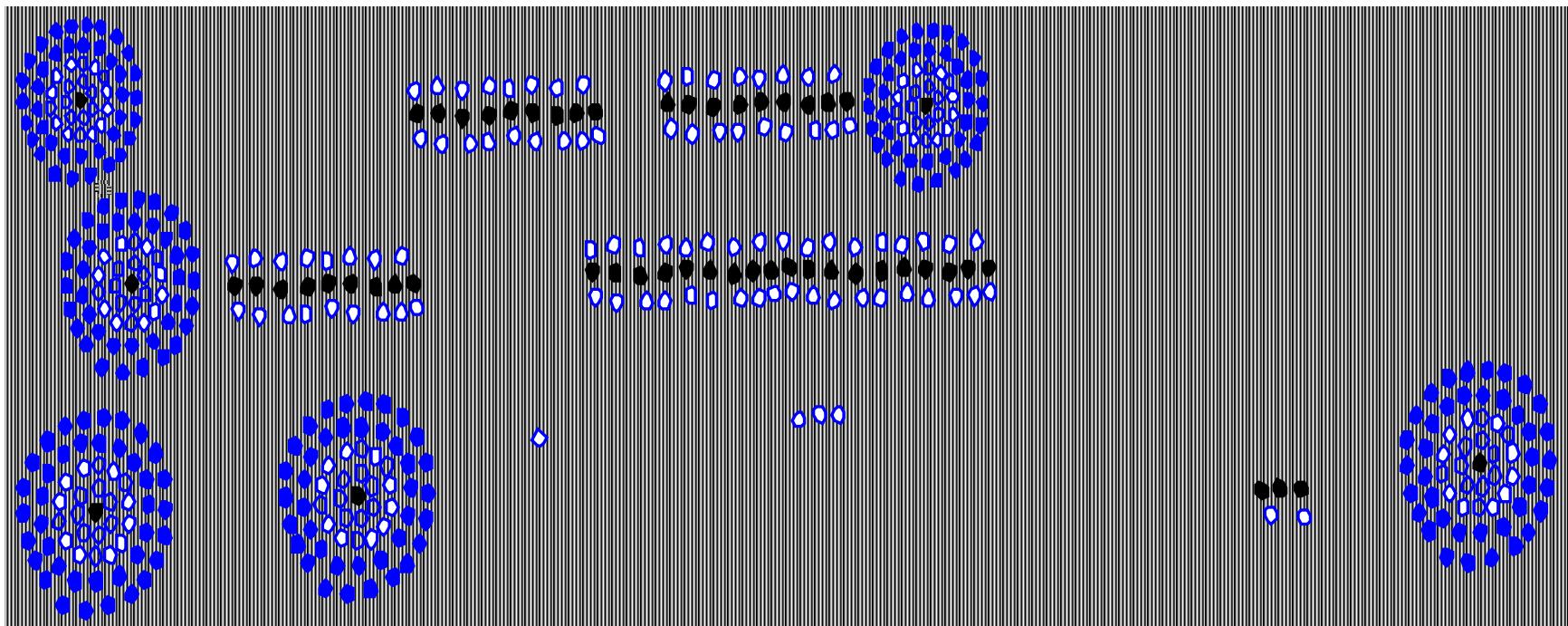
Insertion (2)

To rebalance the subtree rooted at z , we must perform a *rotation*.



AVL Trees

□ AVL Trees

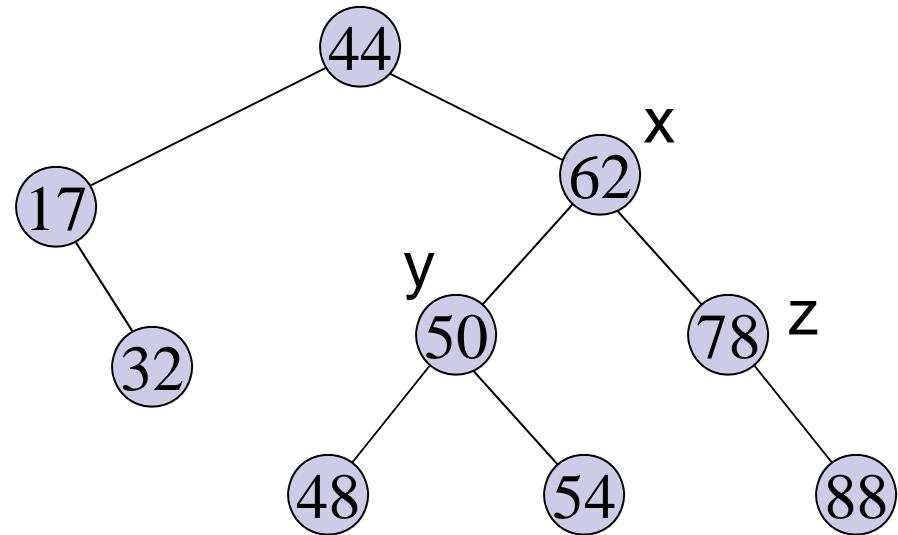
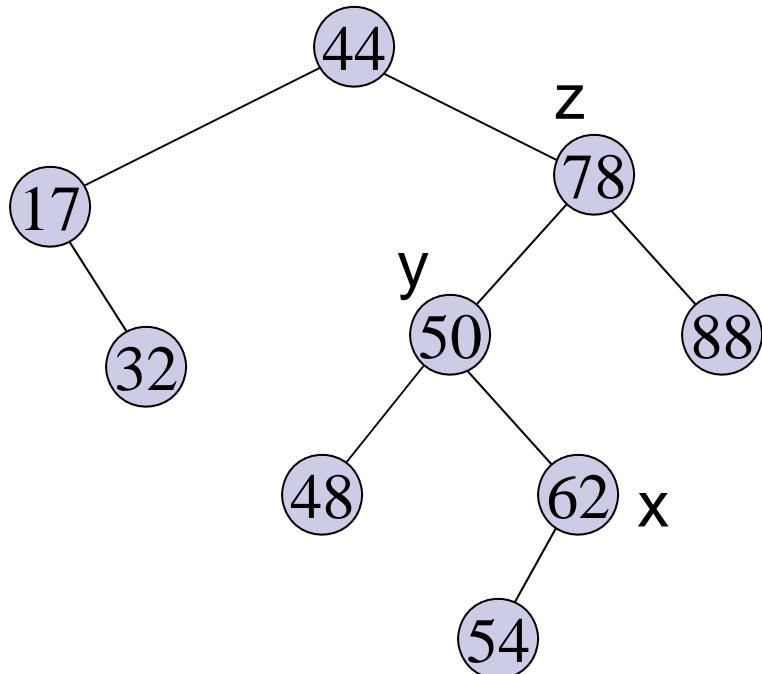


Insertion

- Inserting a node, v , into an AVL tree changes the heights of some of the nodes in T .
- The only nodes whose heights can increase are the ancestors of node v .
- If insertion causes T to become **unbalanced**, then some ancestor of v would have a height-imbalance.
- We travel up the tree from v until we find the first node x such that its grandparent z is unbalanced.
- Let y be the parent of node x .

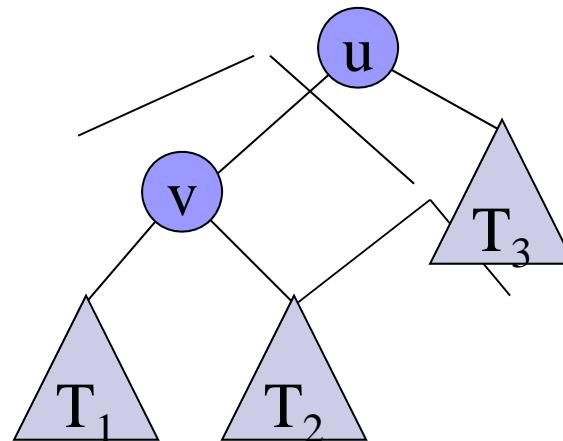
Insertion (2)

To rebalance the subtree rooted at z , we must perform a *rotation*.



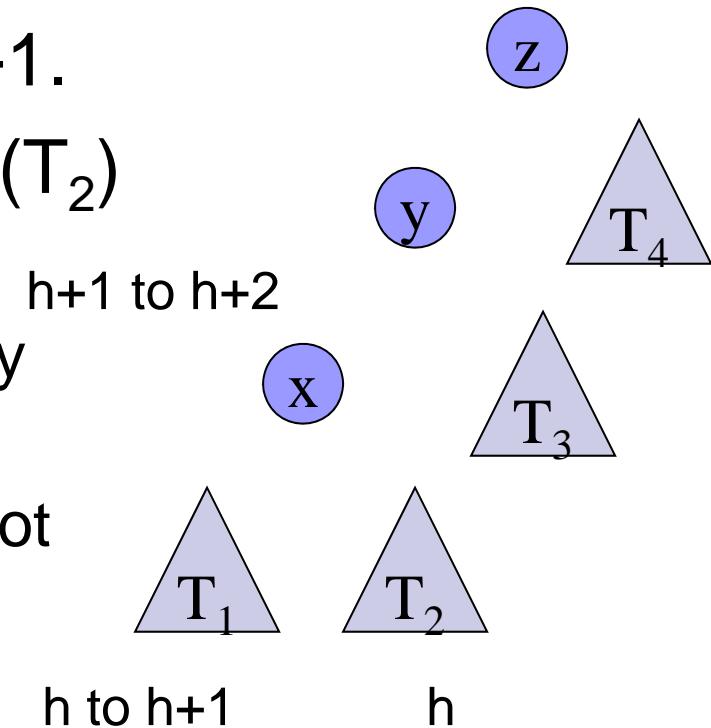
Rotations

- Rotation is a way of locally reorganizing a BST.
- Let u, v be two nodes such that $u = \text{parent}(v)$
- $\text{Keys}(T_1) < \text{key}(v) < \text{keys}(T_2) < \text{key}(u) < \text{keys}(T_3)$

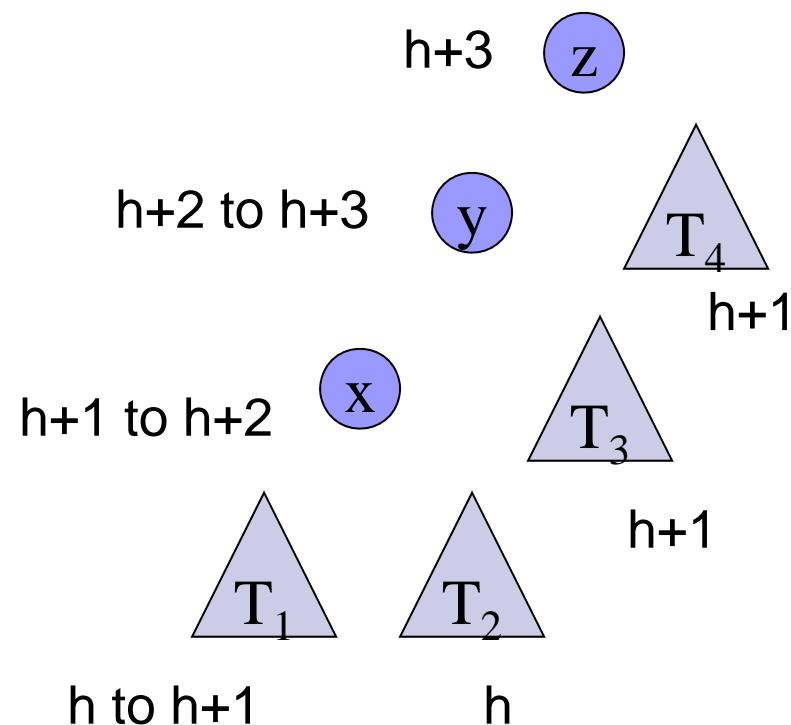


Insertion

- Insertion happens in subtree T_1 .
- $ht(T_1)$ increases from h to $h+1$.
- Since x remains balanced $ht(T_2)$ is h or $h+1$ or $h+2$.
 - If $ht(T_2)=h+2$ then x is originally unbalanced
 - If $ht(T_2)=h+1$ then $ht(x)$ does not increase.
 - Hence $ht(T_2)=h$.
- So $ht(x)$ increases from $h+1$ to $h+2$.

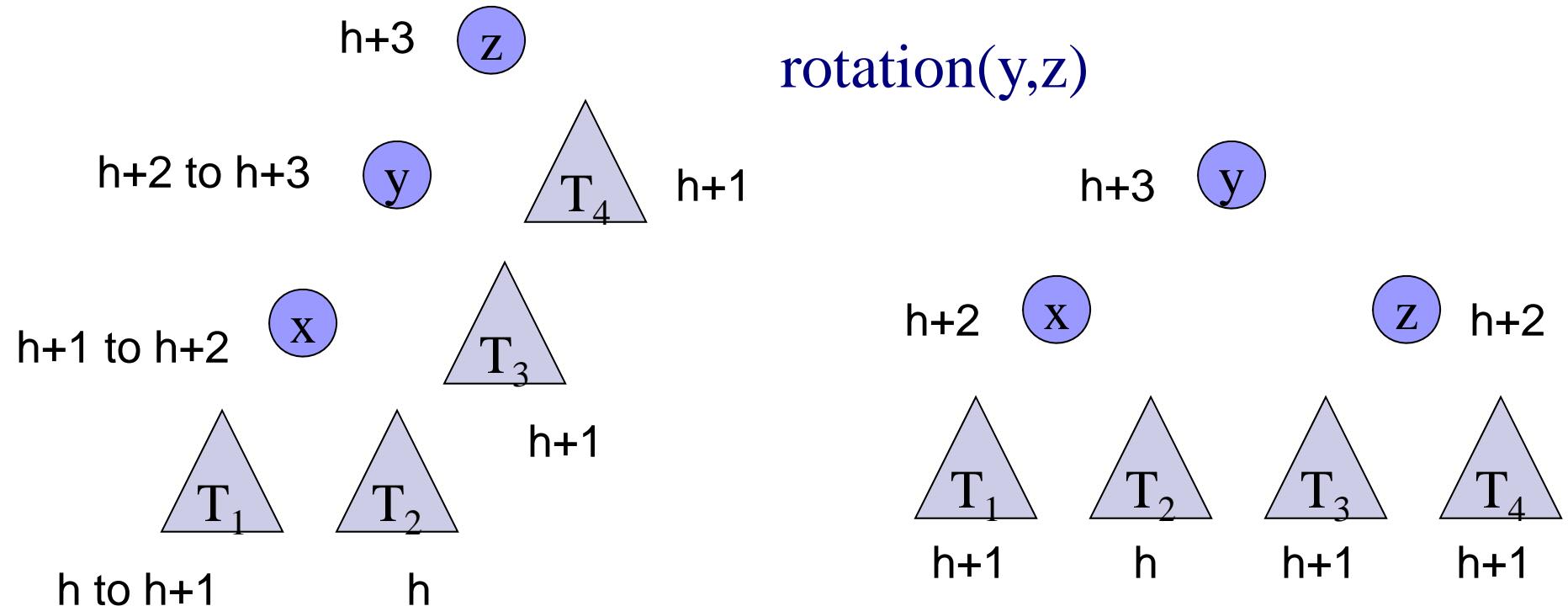


Insertion(2)



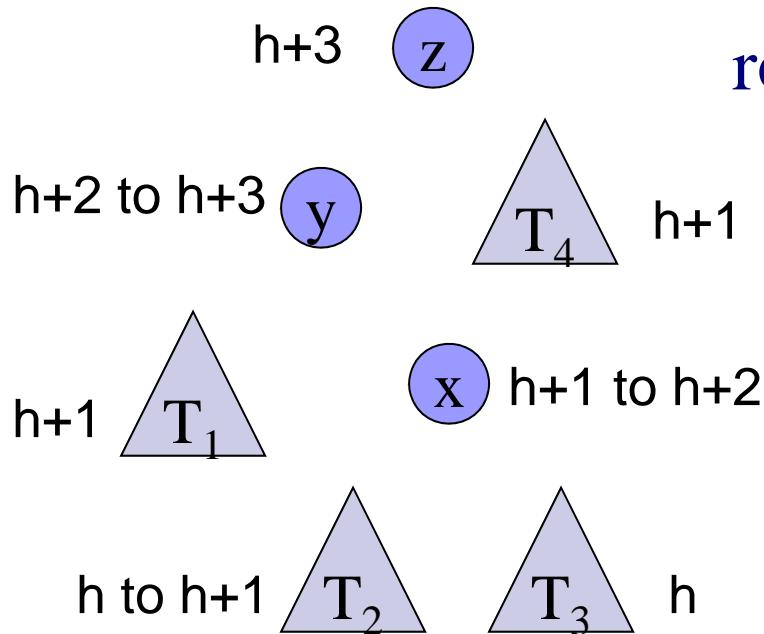
- Since y remains balanced, $\text{ht}(T_3)$ is $h+1$ or $h+2$ or $h+3$.
 - If $\text{ht}(T_3)=h+3$ then y is originally unbalanced.
 - If $\text{ht}(T_3)=h+2$ then $\text{ht}(y)$ does not increase.
 - So $\text{ht}(T_3)=h+1$.
- So $\text{ht}(y)$ inc. from $h+2$ to $h+3$.
- Since z was balanced $\text{ht}(T_4)$ is $h+1$ or $h+2$ or $h+3$.
- z is now unbalanced and so $\text{ht}(T_4)=h+1$.

Single rotation

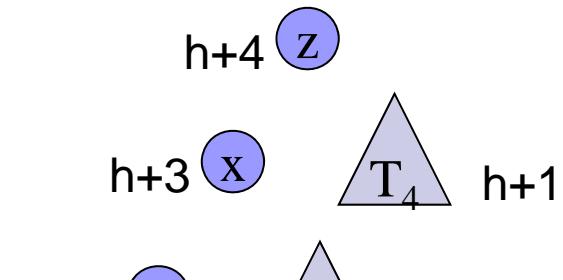


The height of the subtree remains the same after rotation. Hence no further rotations required

Double rotation

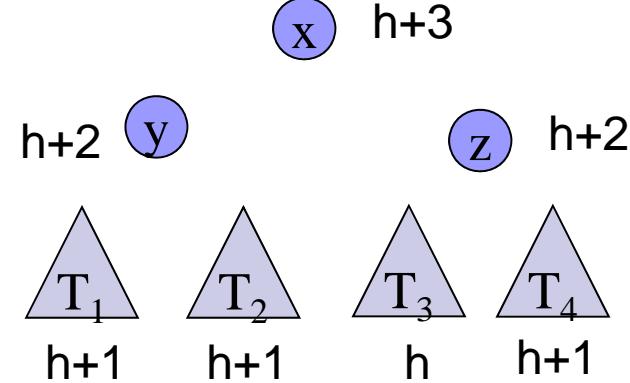


rotation(x,y)



rotation(x,z)

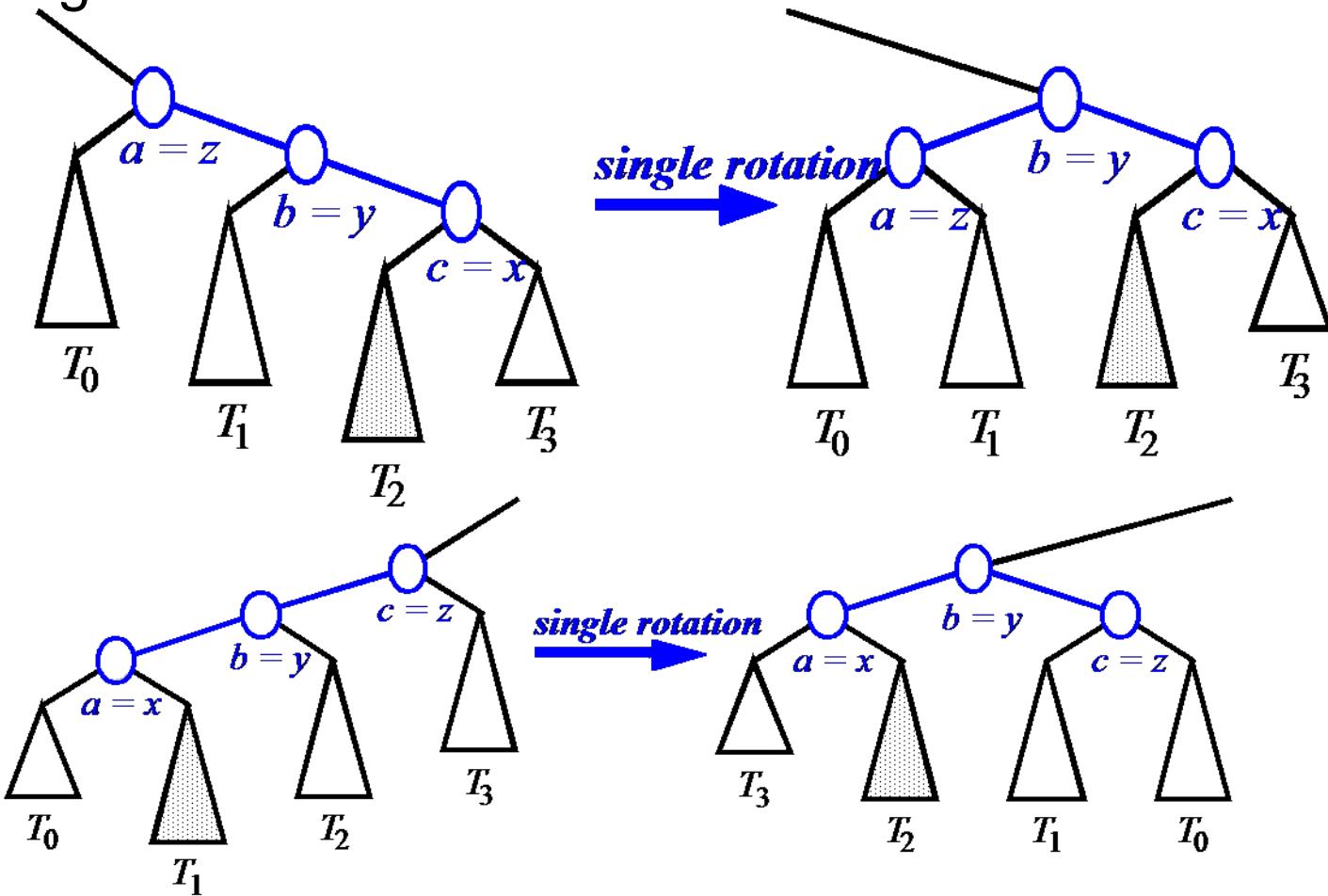
Final tree has same height as original tree. Hence we need not go further up the tree.



Restructuring

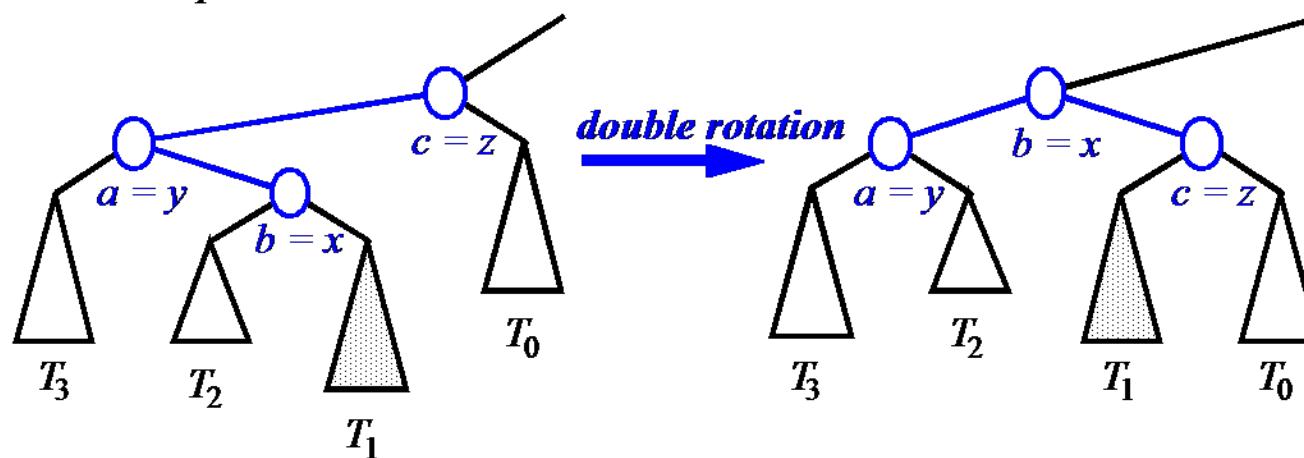
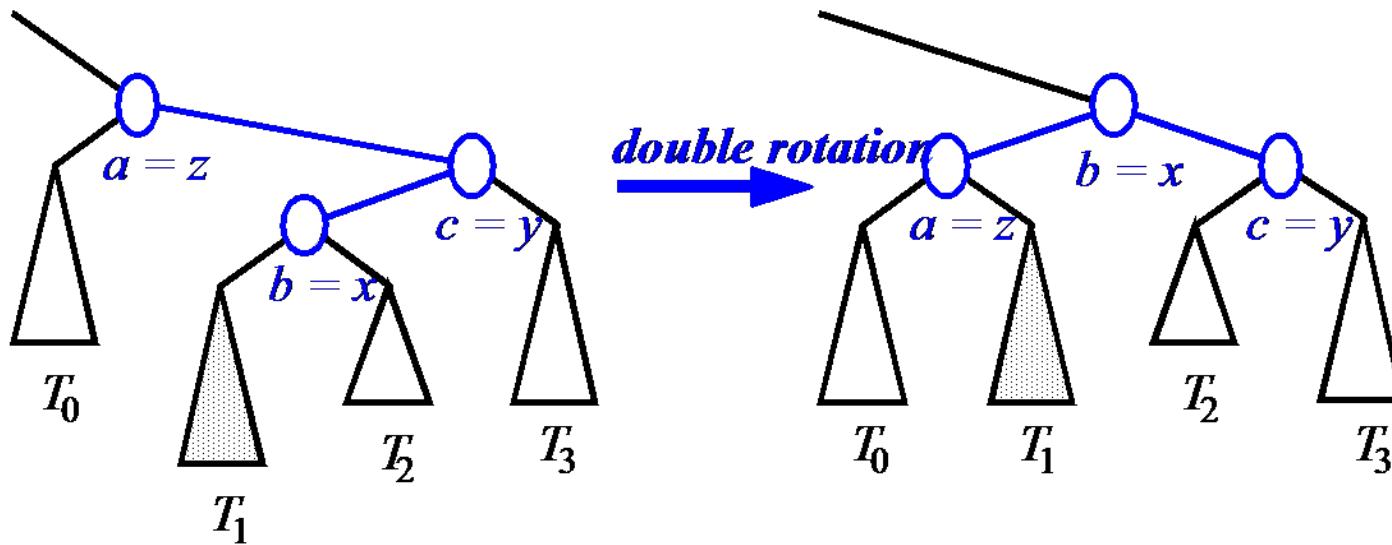
- The four ways to rotate nodes in an AVL tree, graphically represented

-Single Rotations:



Restructuring (contd.)

- double rotations:



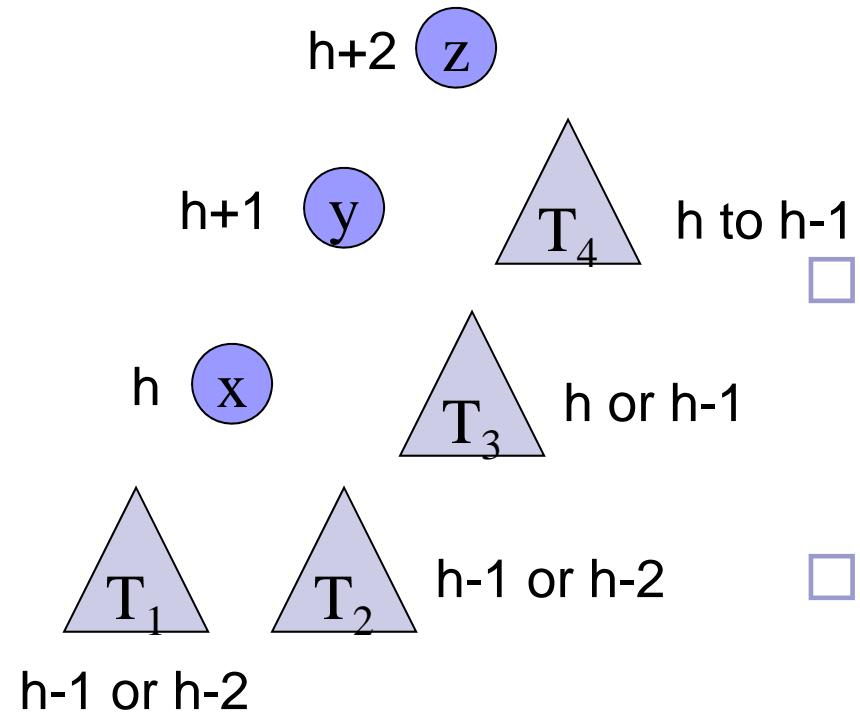
Deletion

- When deleting a node in a BST, we either delete a leaf or a node with only one child.
- In an AVL tree if a node has only one child then that child is a leaf.
- Hence in an AVL tree we either delete a leaf or the parent of a leaf.
- Hence deletion can be assumed to be at a leaf.

Deletion(2)

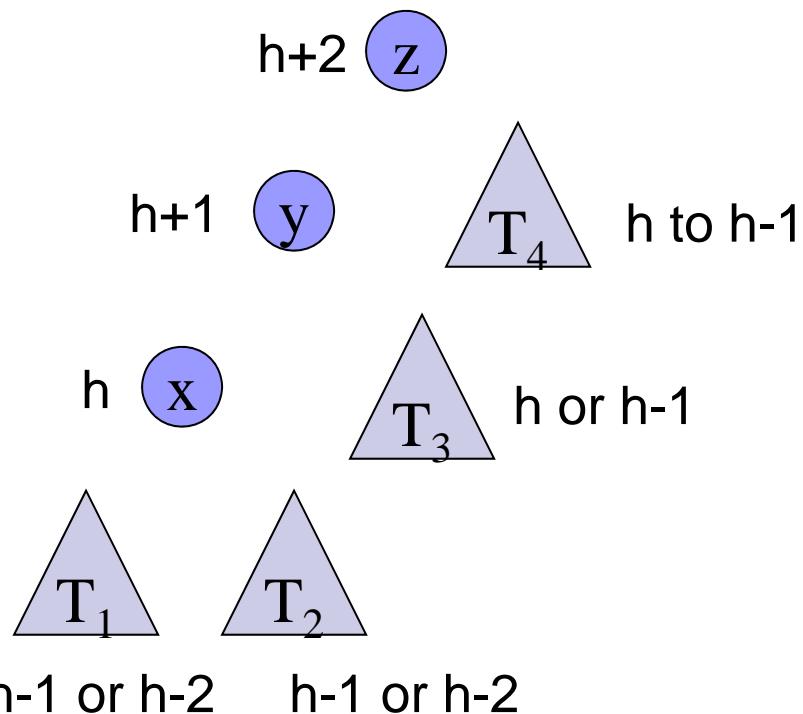
- Let w be the node deleted.
- Let **z** be the **first unbalanced** node encountered while travelling up the tree from w. Also, let y be the child of z with larger height, and let x be the child of y with larger height.
- We perform rotations to restore balance at the subtree rooted at z.
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached

Deletion(3)



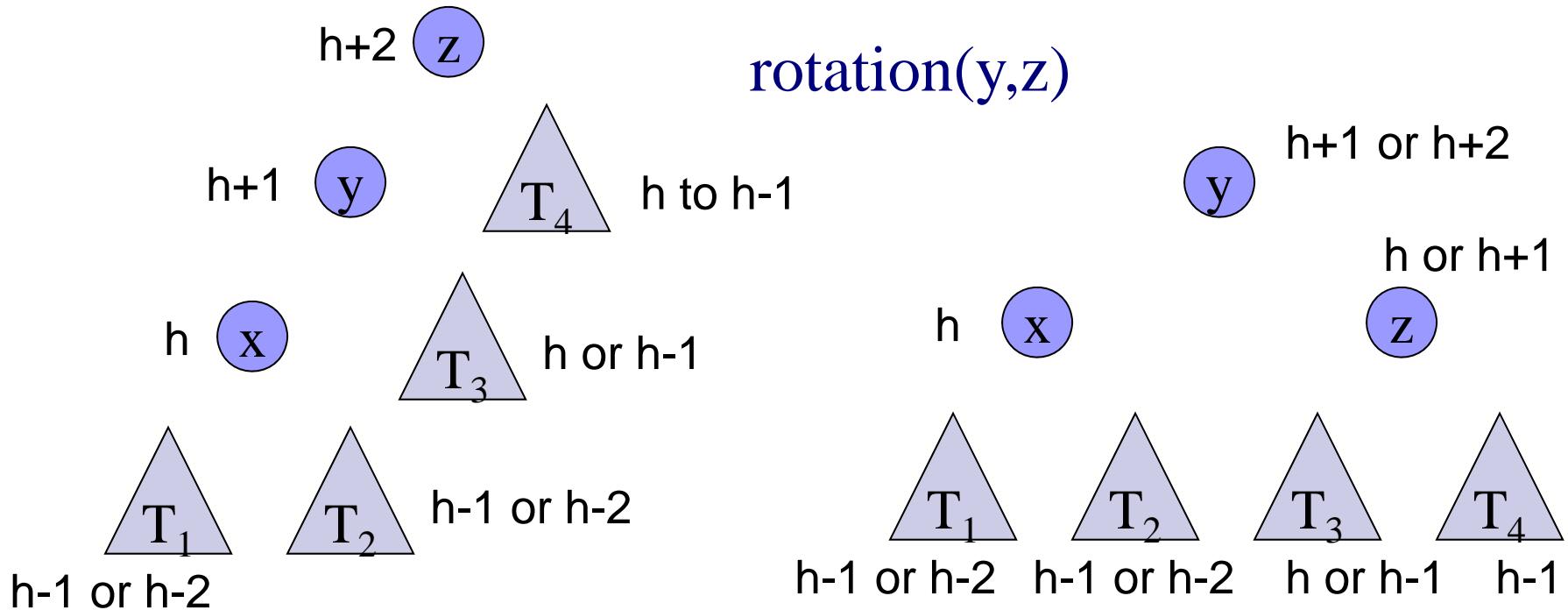
- Suppose deletion happens in subtree T_4 and its ht. reduces from h to $h-1$.
- Since z was balanced but is now unbalanced, $ht(y) = h+1$.
- x has larger ht. than T_3 and so $ht(x)=h$.
- Since y is balanced $ht(T_3)= h$ or $h-1$

Deletion(4)



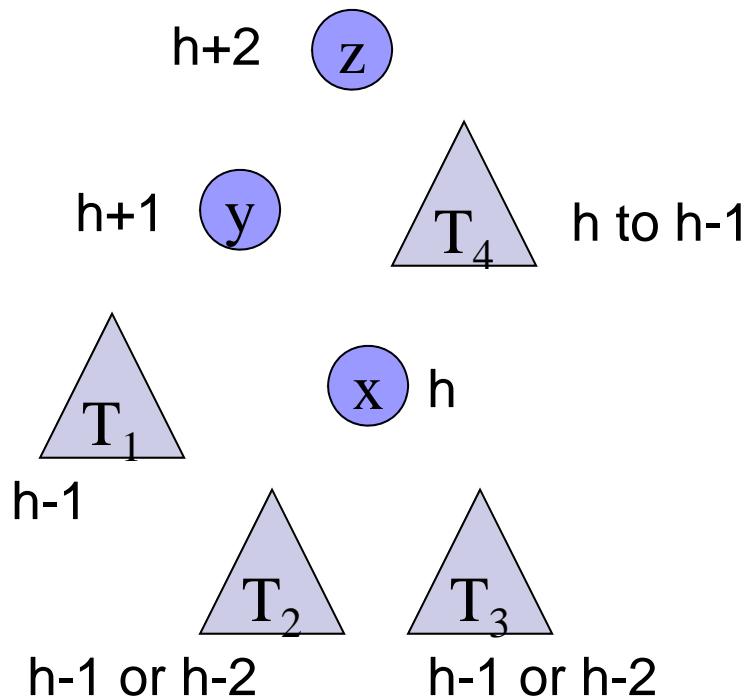
- Since $\text{ht}(x)=h$, and x is balanced $\text{ht}(T_1), \text{ht}(T_2)$ is $h-1$ or $h-2$.
- However, both T_1 and T_2 cannot have ht. $h-2$

Single rotation (deletion)



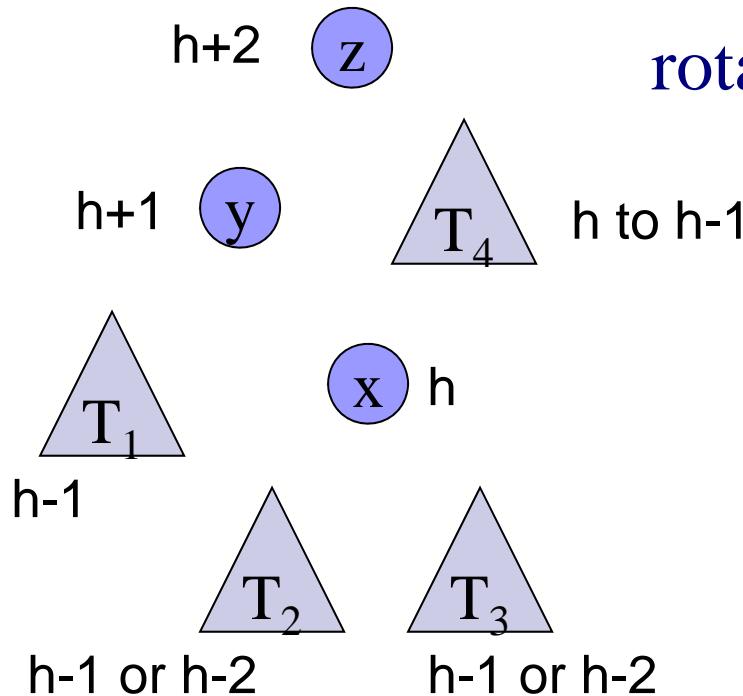
After rotation height of subtree might be 1 less than original height. In that case we continue up the tree

Deletion: another case

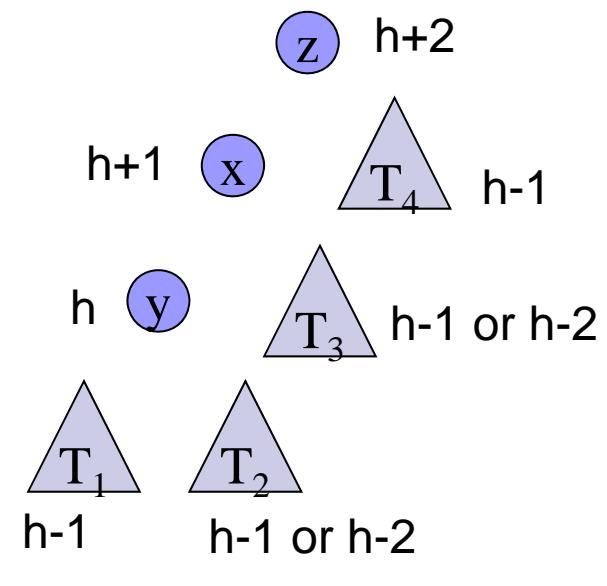


- As before we can claim that $\text{ht}(y)=h+1$ and $\text{ht}(x)=h$.
- Since y is balanced $\text{ht}(T_1)$ is h or $h-1$.
- If $\text{ht}(T_1)$ is h then we would have picked x as the root of T_1 .
- So $\text{ht}(T_1)=h-1$

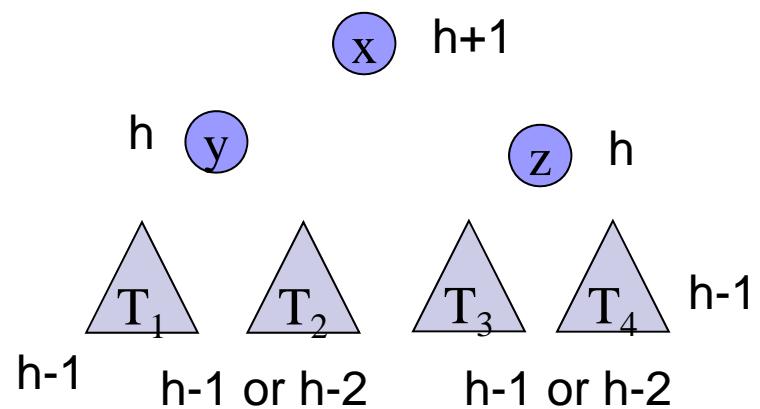
Double rotation



$\text{rotation}(x,y)$



$\text{rotation}(x,z)$



Final tree has height less than original tree. Hence we need to continue up the tree

Running time of insertion & deletion

□ Insertion

- We perform rotation only once but might have to go $O(\log n)$ levels to find the unbalanced node.
- So time for insertion is $O(\log n)$

□ Deletion

- We need $O(\log n)$ time to delete a node.
- Rebalancing also requires $O(\log n)$ time.
- More than one rotation may have to be performed.

(2,4) Trees

- What are they?
 - They are search Trees (but not binary search trees)
 - They are also known as 2-4, 2-3-4 trees

Multi-way Search Trees

- Each internal node of a multi-way search tree T:
 - has at least two children
 - stores a collection of items of the form (k, x) , where k is a key and x is an element
 - contains $d - 1$ items, where d is the number of children
 - Has pointers to d children
- Children of each internal node are “between” items
- all keys in the subtree rooted at the child fall between keys of those items.

Multi-way Searching

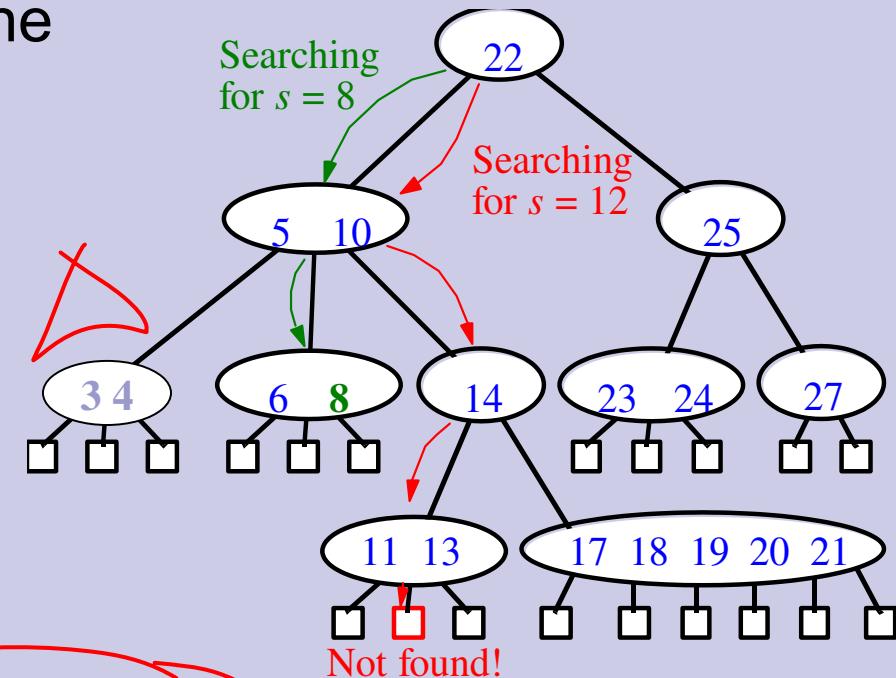
- Similar to binary searching

- If search key $s < k_1$, search the leftmost child
- If $s > k_{d-1}$, search the rightmost child

- That's it in a binary tree; what about if $d > 2$?

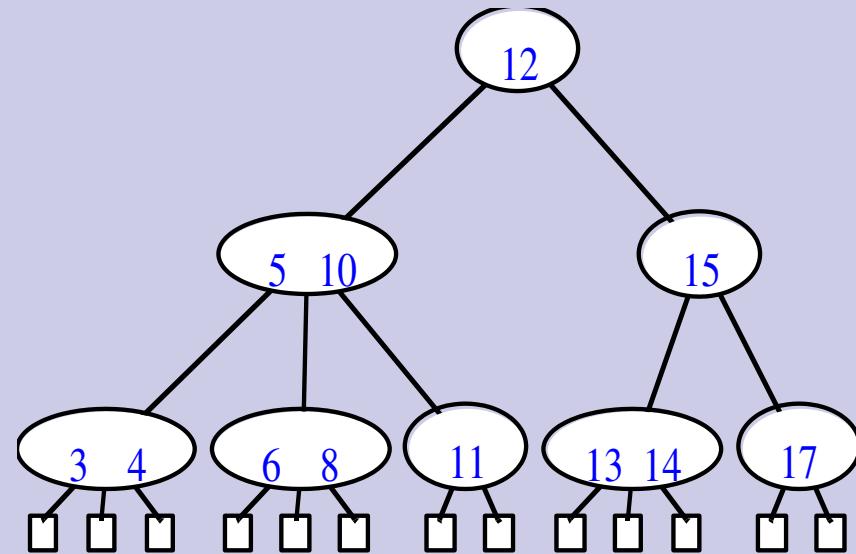
- Find two keys k_{i-1} and k_i between which s falls, and search the child v_i .

- What would an in-order traversal look like?



(2,4) Trees

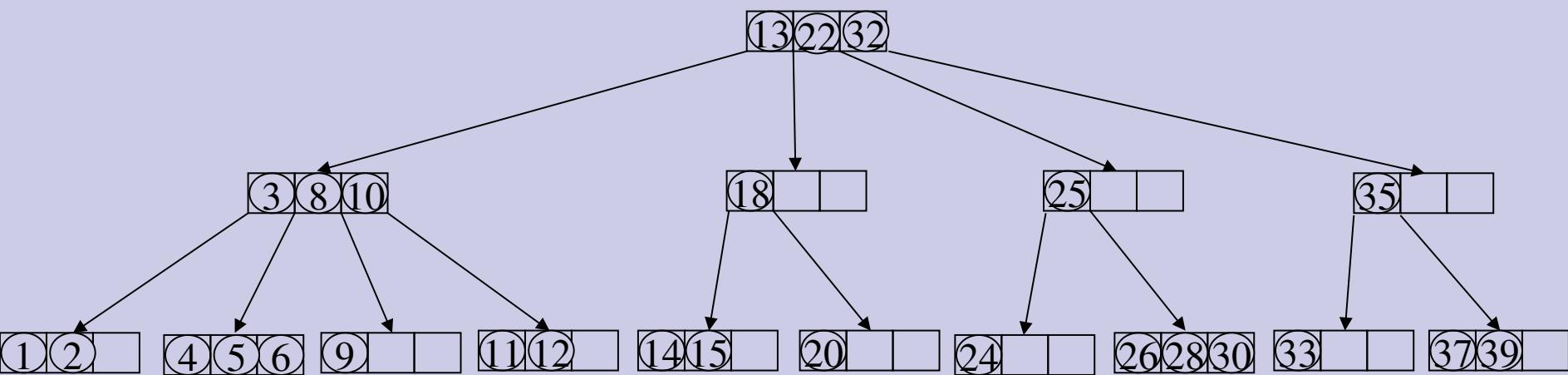
- Properties:
 - At most 4 children
 - All leaf nodes are at the same level.
 - Height h of (2,4) tree is at least $\log_4 n$ and atmost $\log_2 n$
- How is the last fact useful in searching?



Insertion

21 23 40 29 7

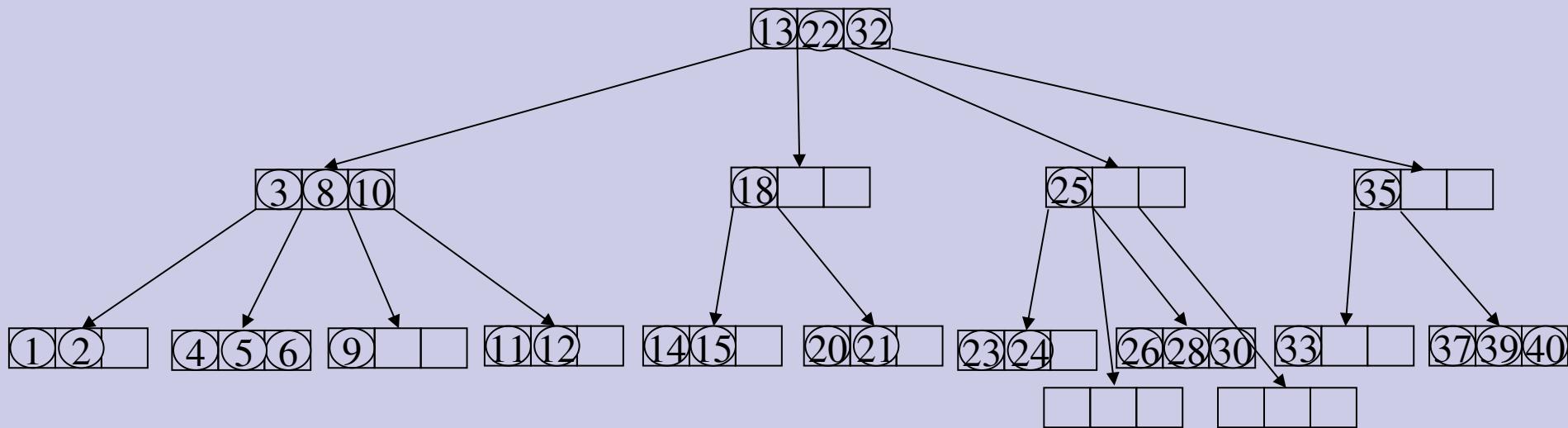
- No problem if the node has empty space



Insertion(2)

②9 ⑦

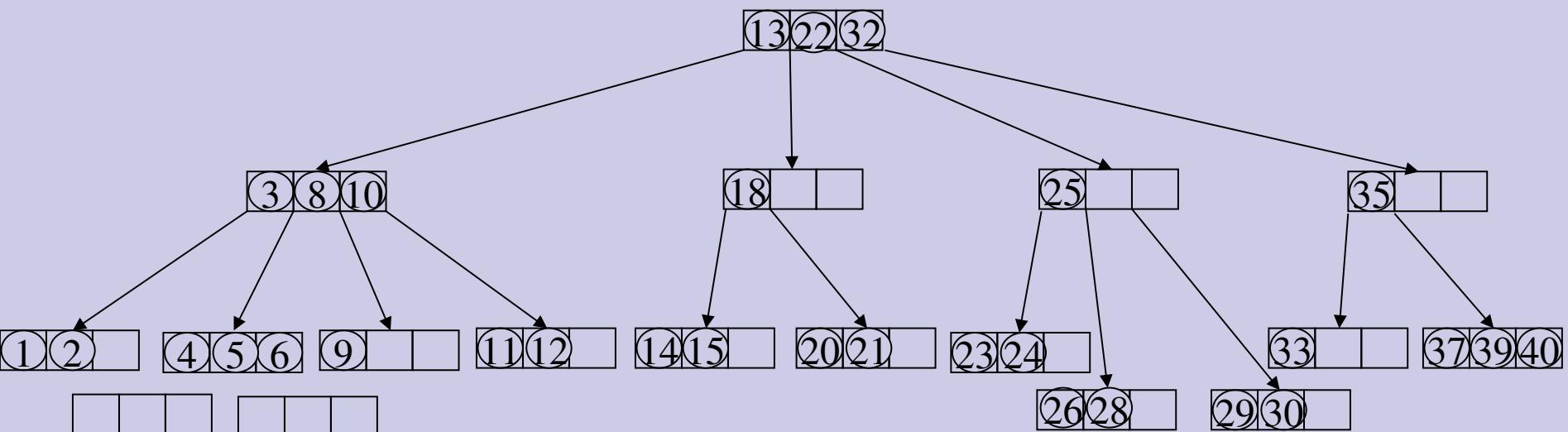
- Nodes get split if there is insufficient space.



Insertion(3)

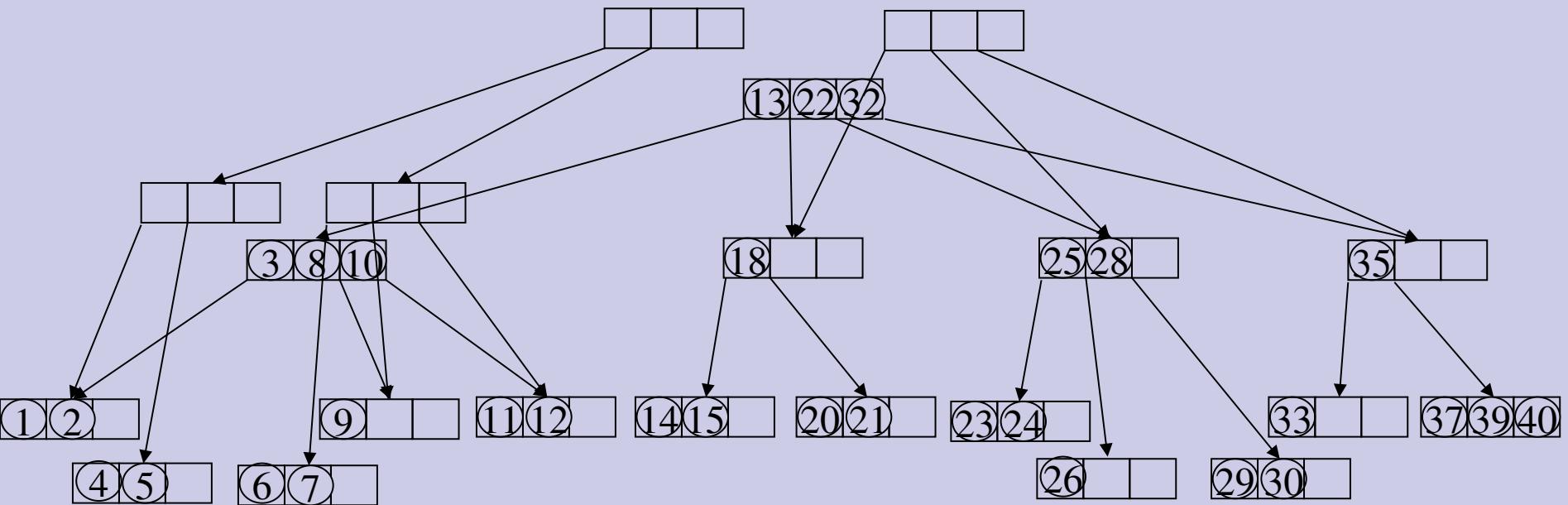
⑦

- One key is promoted to parent and inserted in there



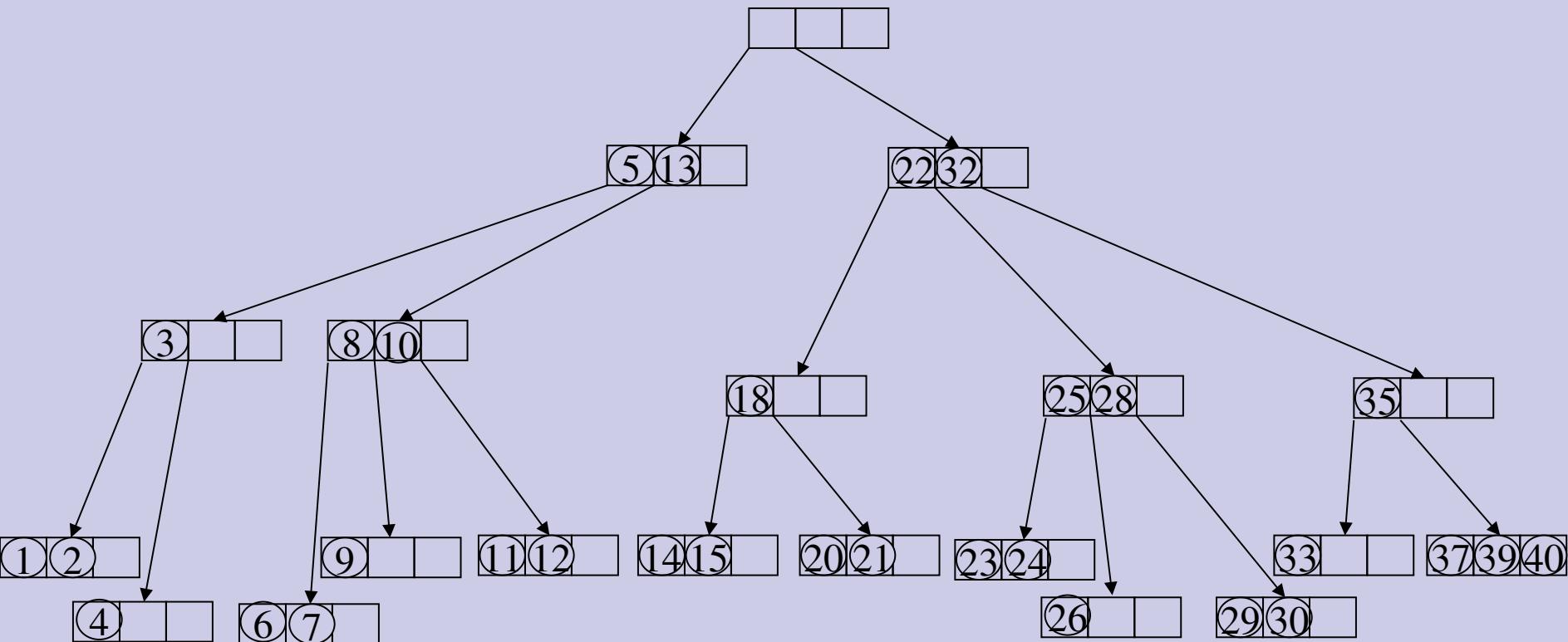
Insertion(4)

- If parent node does not have sufficient space then it is split.
- In this manner splits can cascade.



Insertion(5)

- Eventually we may have to create a new root.
- This increases the height of the tree

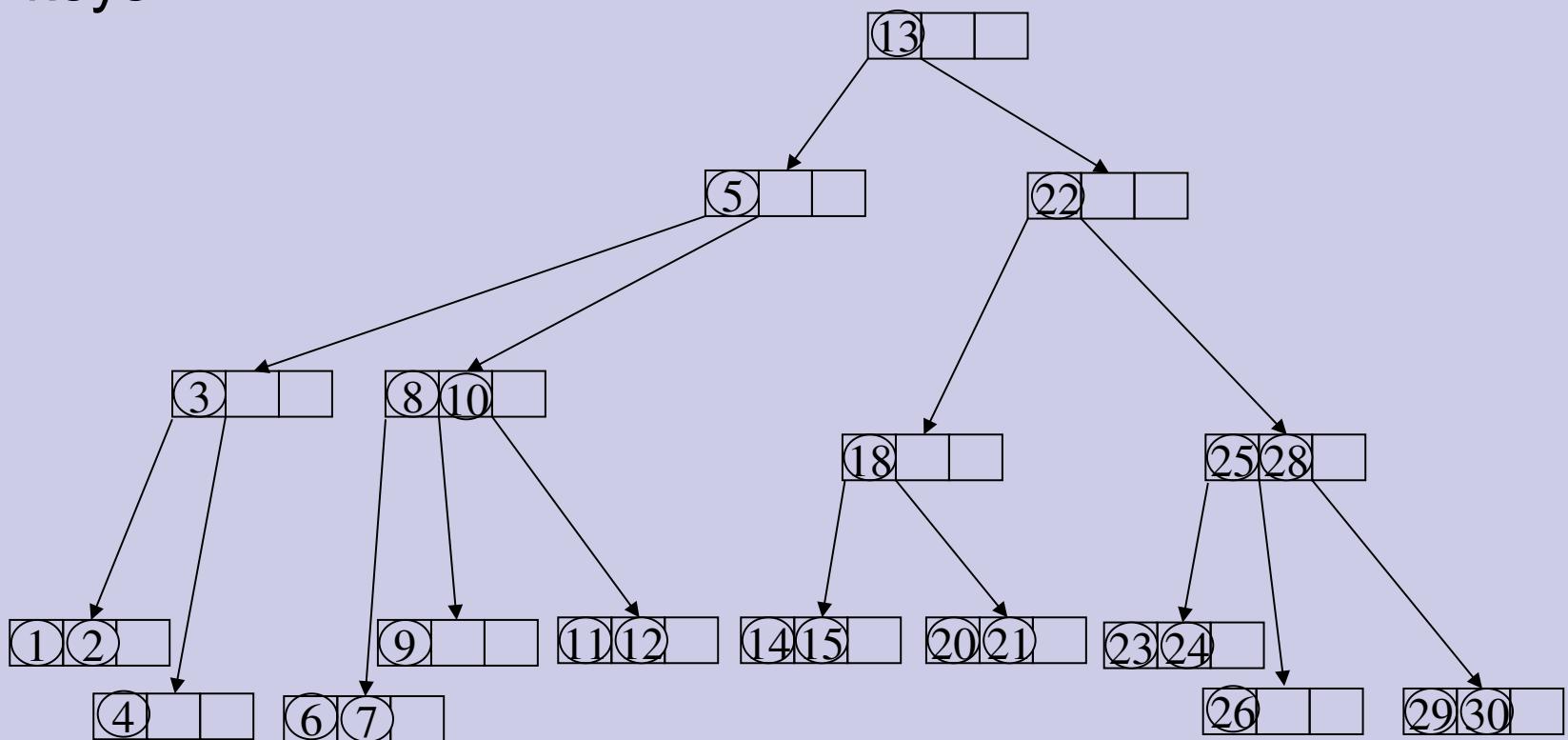


Time for Search and Insertion

- A search visits $O(\log N)$ nodes
- An insertion requires $O(\log N)$ node splits
- Each node split takes constant time
- Hence, operations Search and Insert each take time $O(\log N)$

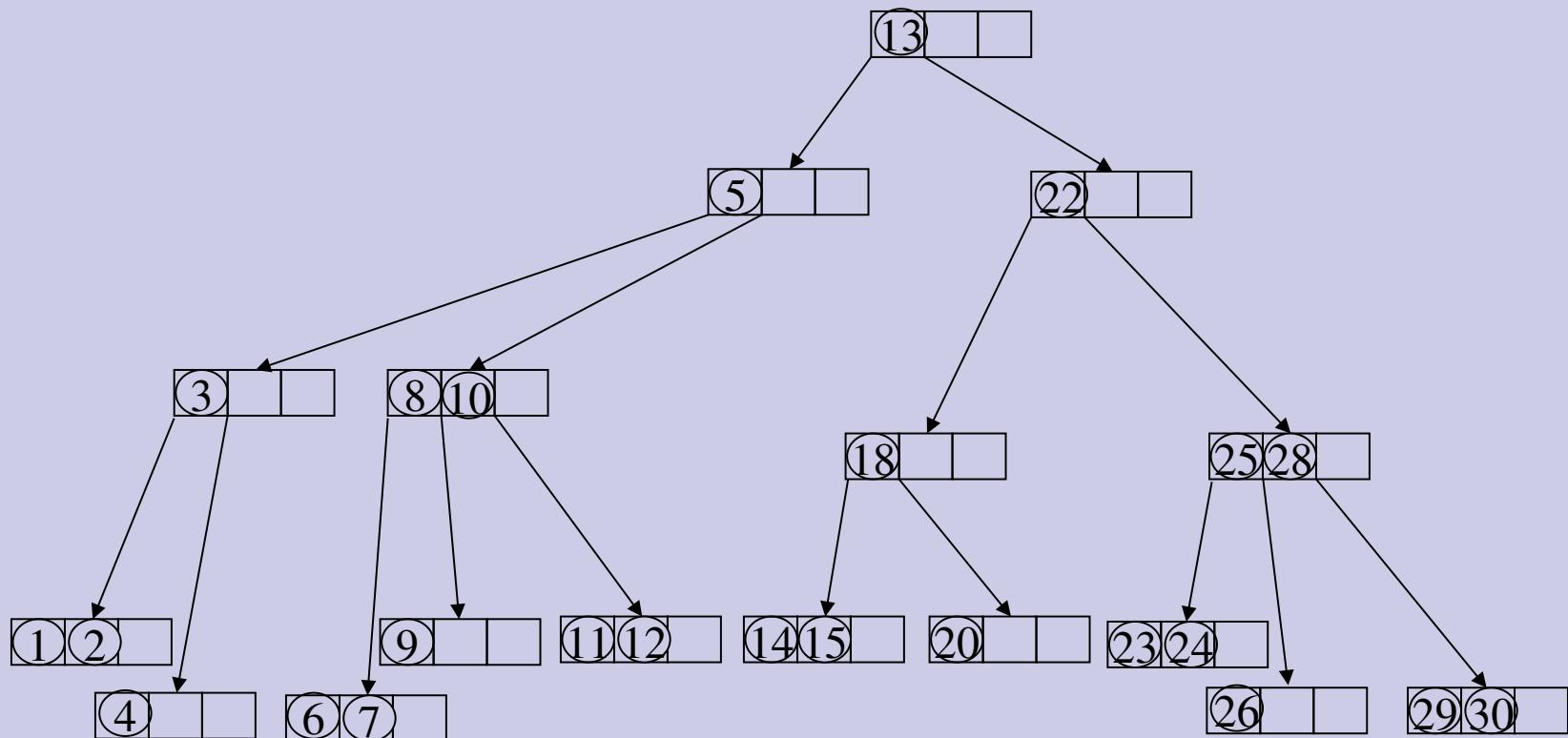
Deletion

- Delete 21.
- No problem if key to be deleted is in a leaf with at least 2 keys



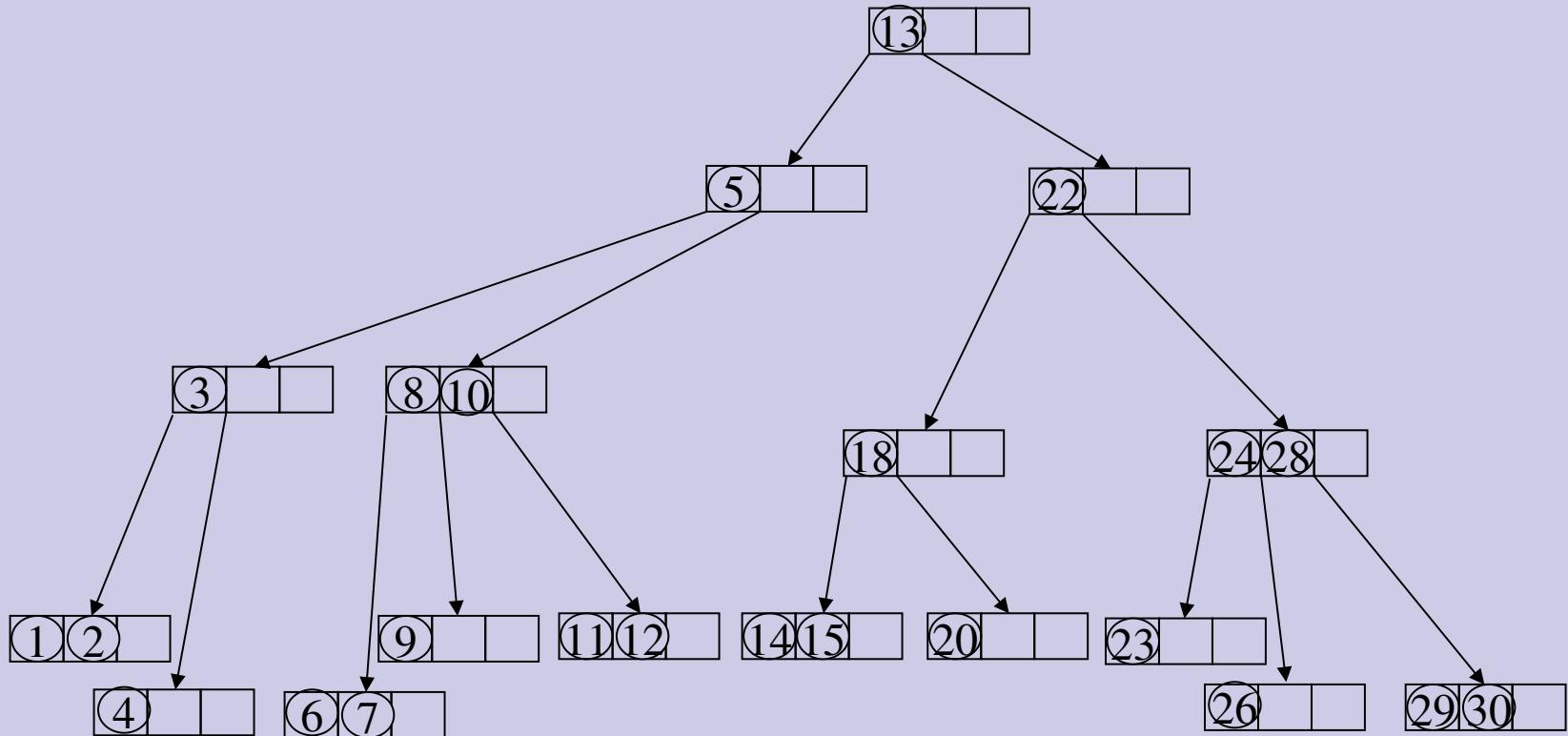
Deletion(2)

- If key to be deleted is in an internal node then we swap it with its predecessor (which is in a leaf) and then delete it.
- Delete 25



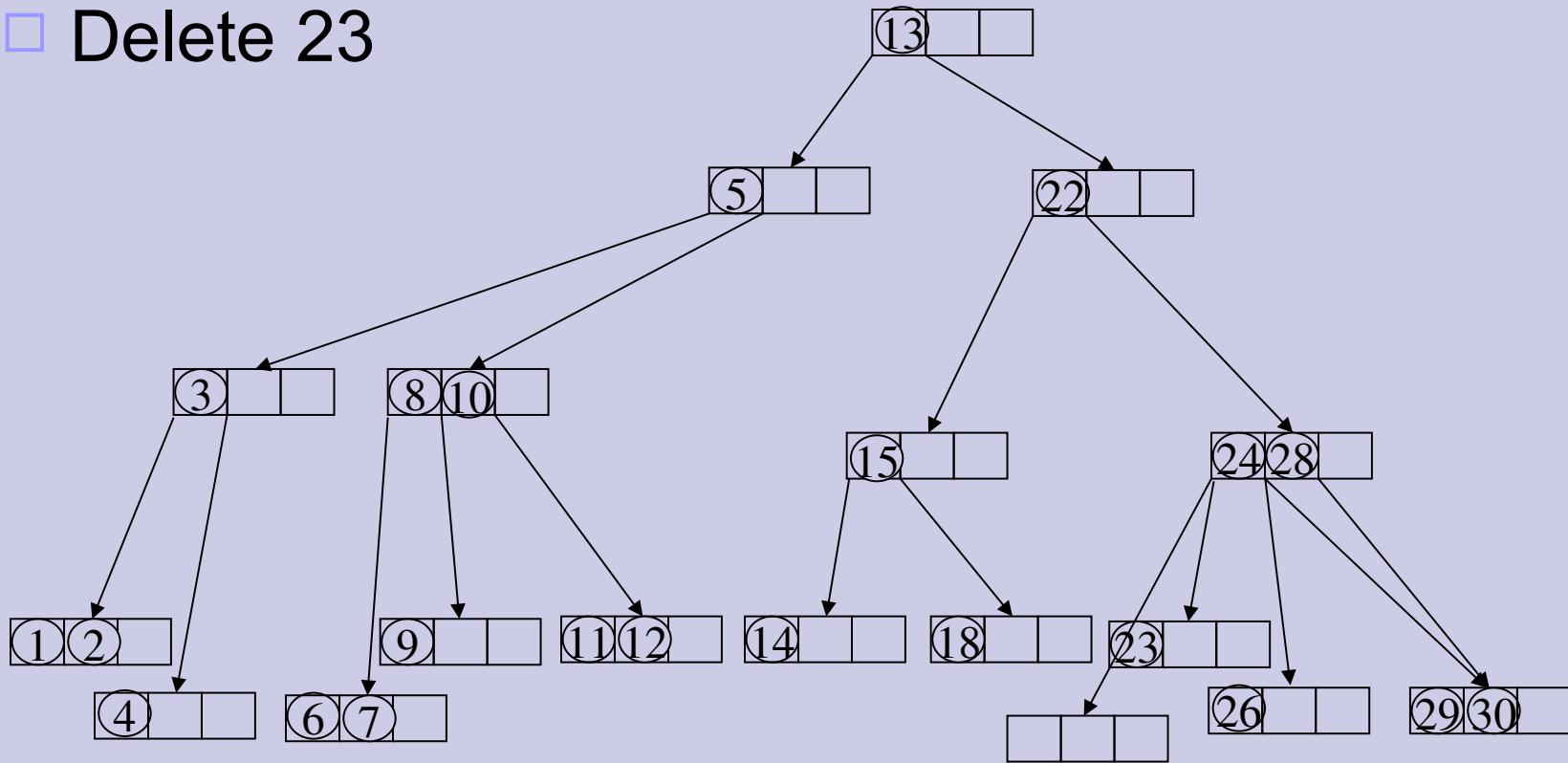
Deletion(3)

- If after deleting a key a node becomes empty then we borrow a key from its sibling.
- Delete 20



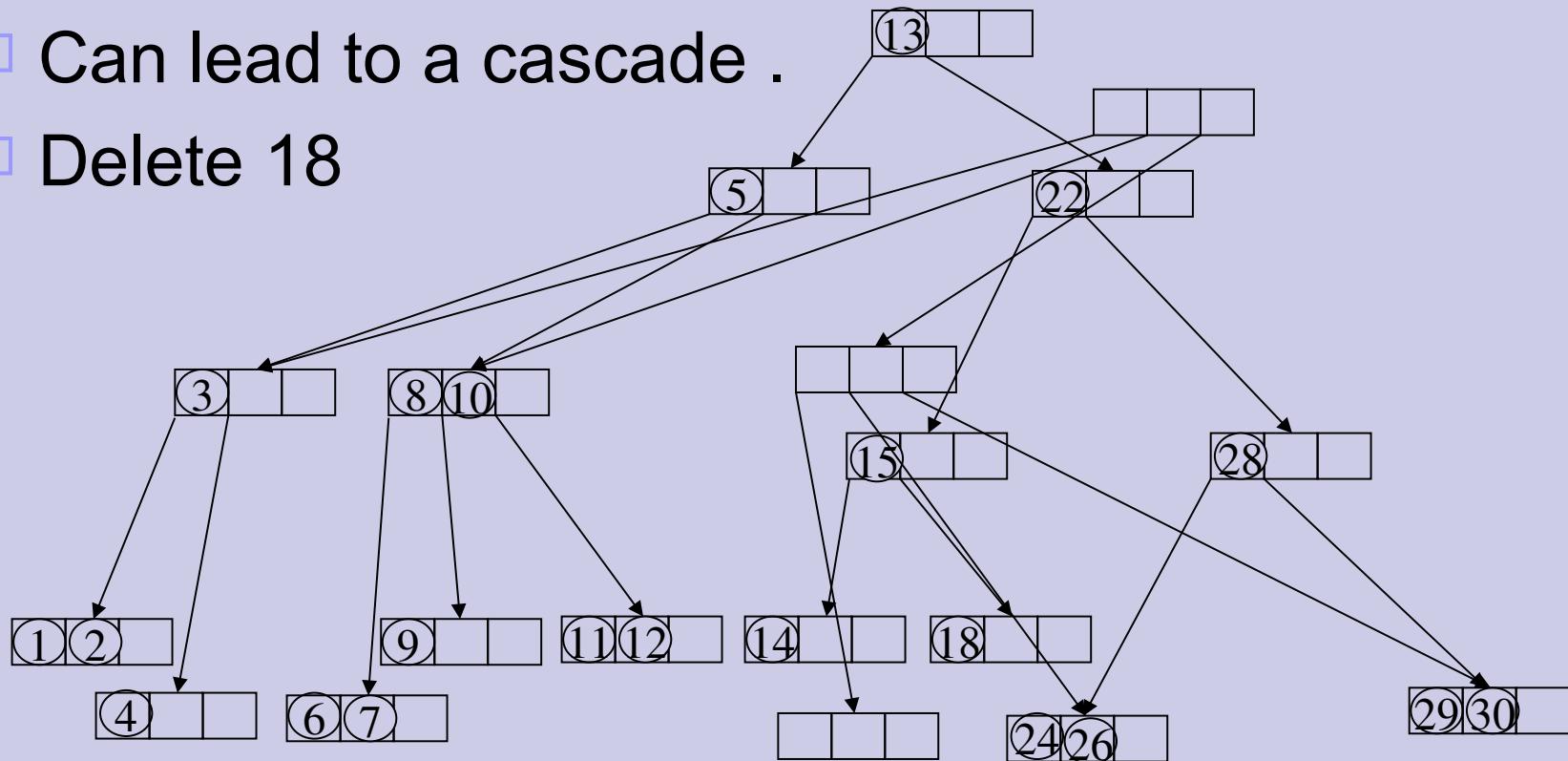
Deletion(4)

- If sibling has only one key then we merge with it.
- The key in the parent node separating these two siblings moves down into the merged node.
- Delete 23



Delete(5)

- Moving a key down from the parent corresponds to deletion in the parent node.
- The procedure is the same as for a leaf node.
- Can lead to a cascade .
- Delete 18



(2,4) Conclusion

- The height of a (2,4) tree is $O(\log n)$.
- Split, transfer, and merge each take $O(1)$.
- Search, insertion and deletion each take $O(\log n)$.
- Why are we doing this?
 - (2,4) trees are fun! Why else would we do it?
 - Well, there's another reason, too.
 - They're pretty fundamental to the idea of Red-Black trees as well.

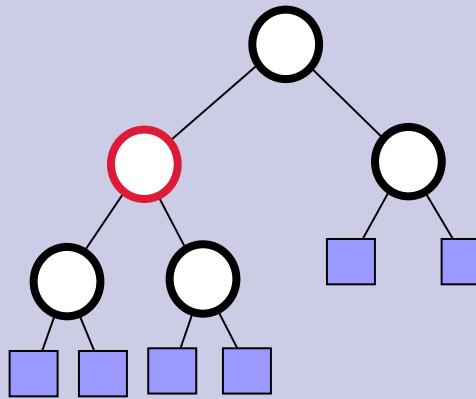
Red-Black Trees

- What are they?
- Their relation to 2-4 trees
- Deletion in red-black trees

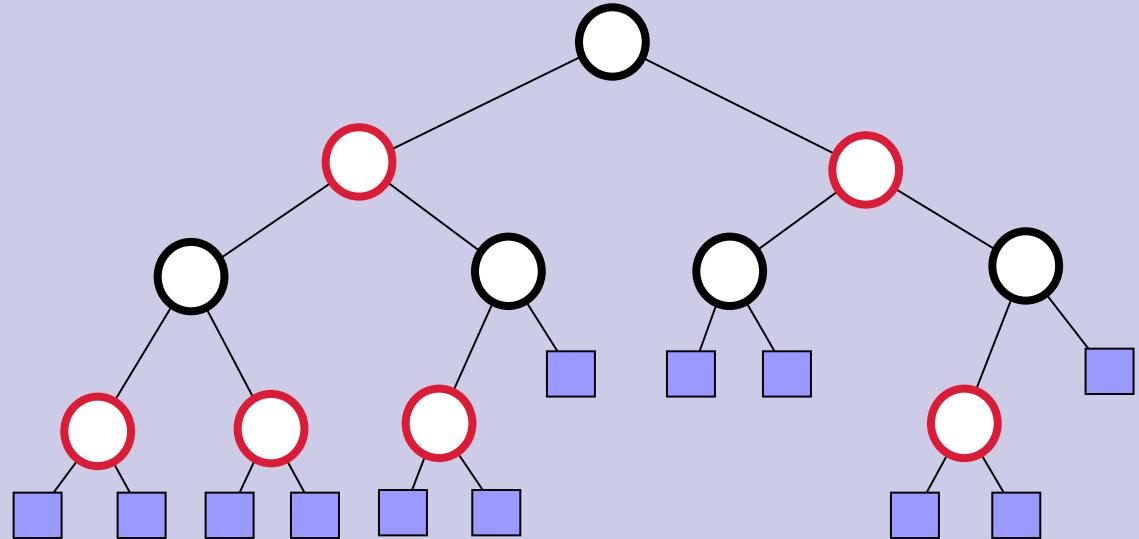
Red-Black Trees

- A red black tree is a binary search tree in which each node is colored red or black.
- The root is colored black.
- A red node can have only black children.
- If a node of the BST does not have a left and/or right child then we add an external node.
- External nodes are not colored.
- The black depth of an external node is defined as the number of black ancestors it has.
- In a red-black tree every external node has the same black depth.

Examples of red-black trees

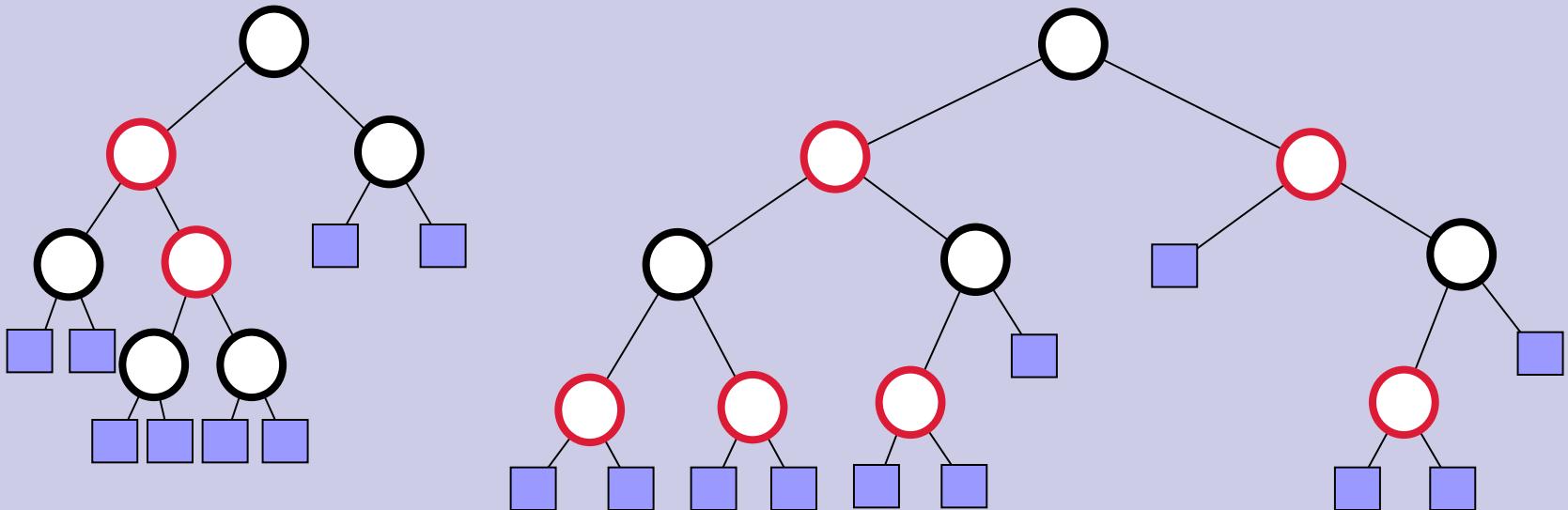


Black height of
tree is 2



Black height of
tree is 2

Trees which are not red-black



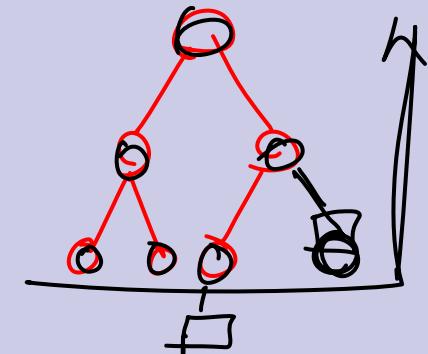
Double red

Black height not uniform

Height of a red-black tree

- Let h be the black height of a red-black tree on n nodes.
- n is smallest when all nodes are black. In this case tree is a complete binary tree of height h and $n=2^h - 1$.
- n is largest when alternate levels of tree are red. Then height of tree is $2h$ and $n=2^{2h}-1$.
- Hence, $\log_4 n < h < 1 + \log_2 n$

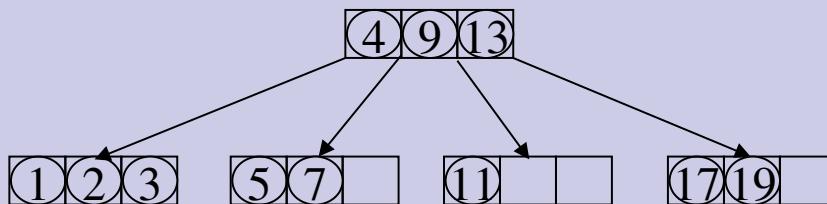
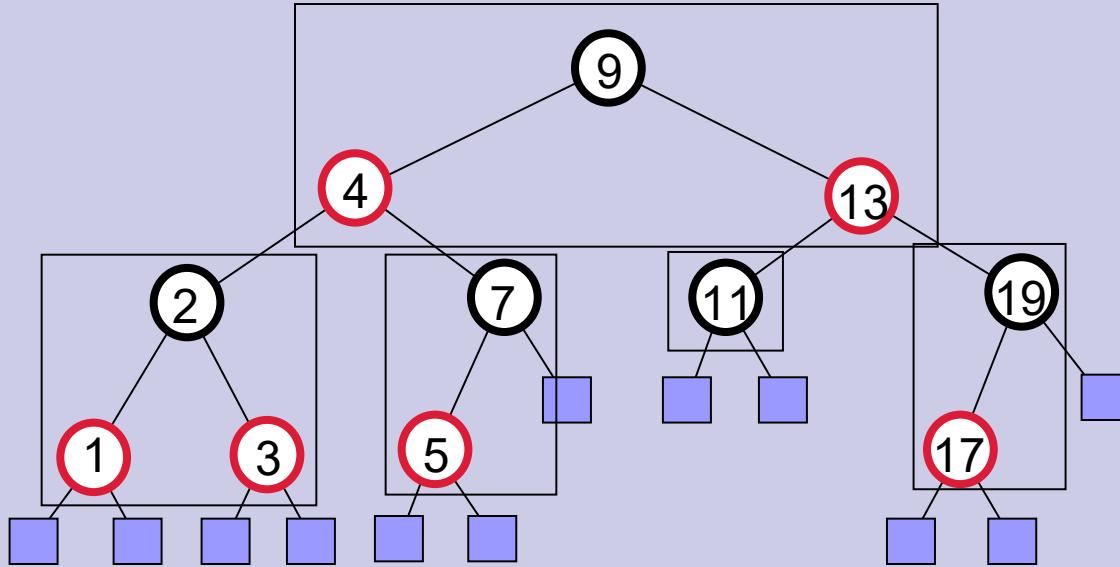
$$2^{h-1} < n < 2^{2h} - 1$$



Red-black trees to 2-4 trees

- Any red-black tree can be converted into a 2-4 tree.
- Take a black node and its red children (at most 2) and combine them into one node of a 2-4 tree.
- Each node so formed has at least 1 and at most 3 keys.
- Since black height of all external nodes is same, in the resulting 2-4 tree all leaves are at same level.

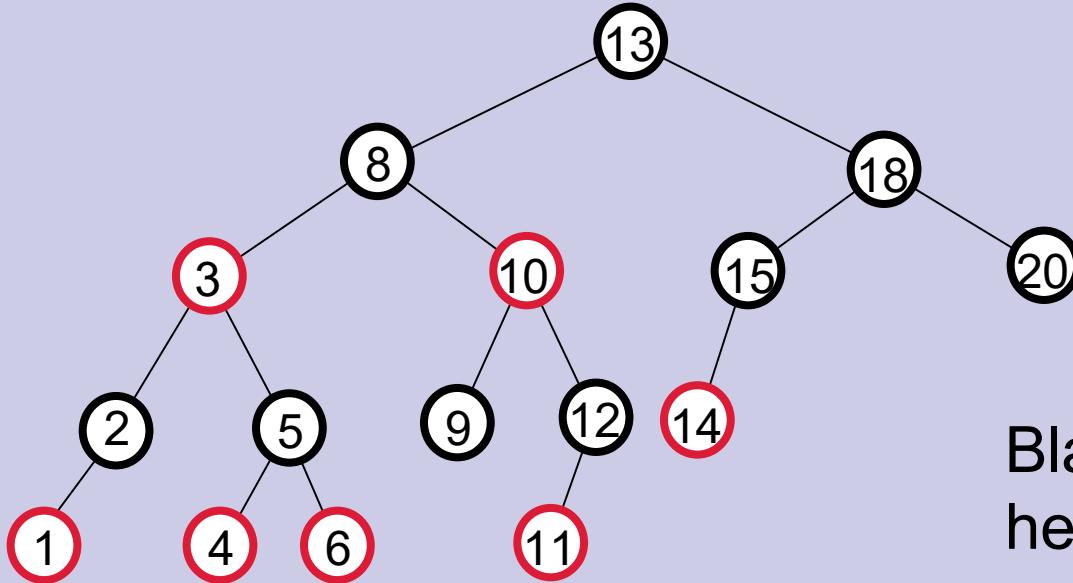
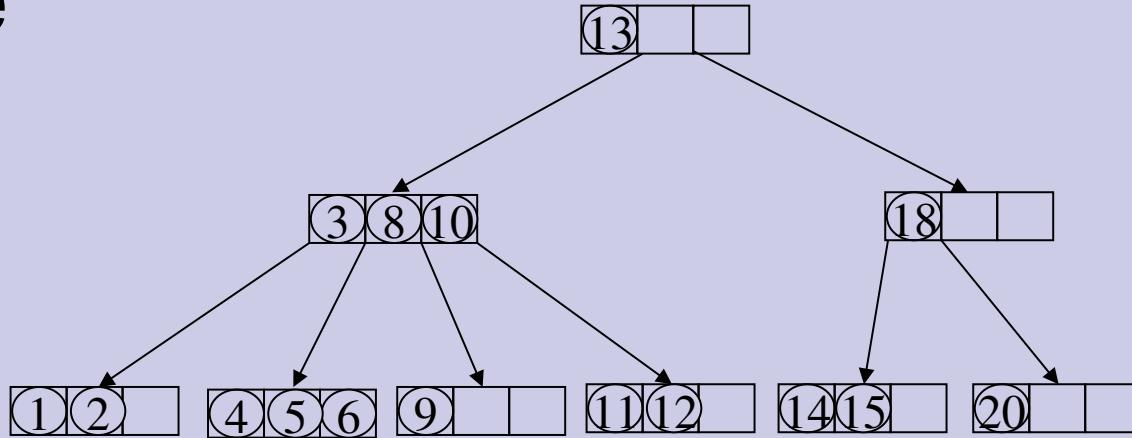
Example



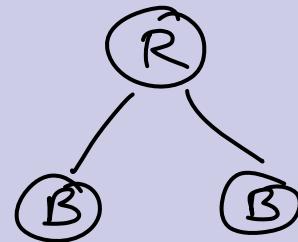
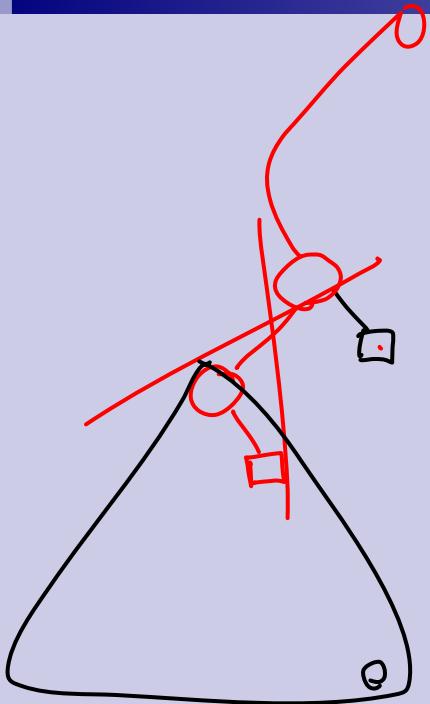
2-4 trees to red-black trees

- Any 2-4 tree can be converted into a red-black tree.
- We replace a node of the 2-4 tree with one black node and 0/1/2 red nodes which are children of the black node.
- The height of the 2-4 tree is the black height of the red-black tree created.
- Every red node has a black child.

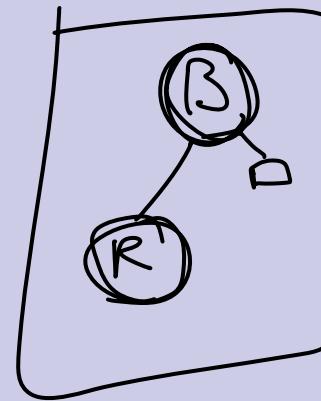
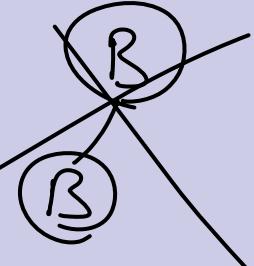
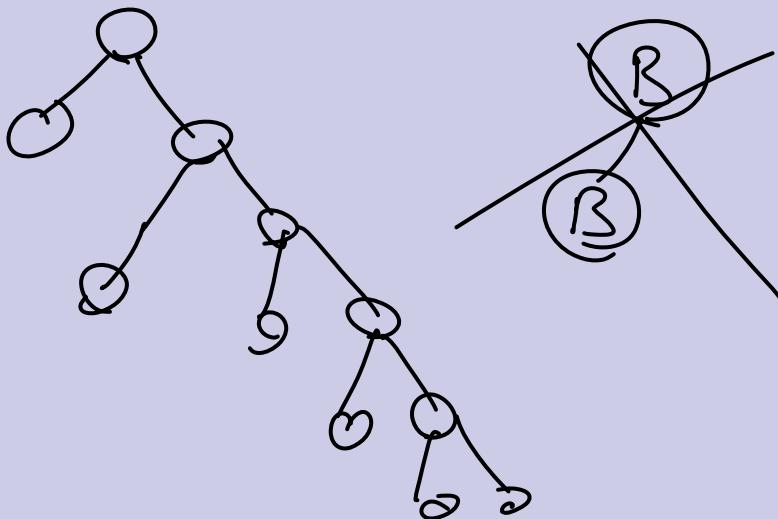
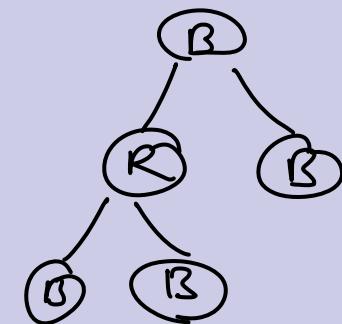
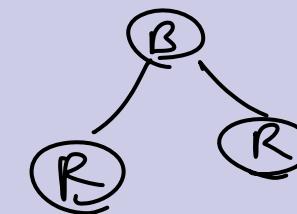
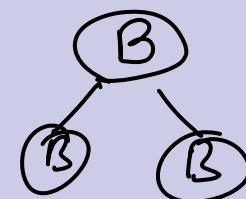
Example

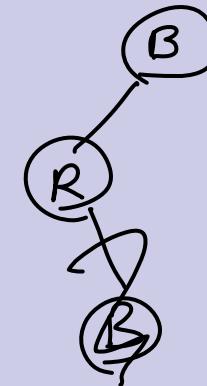
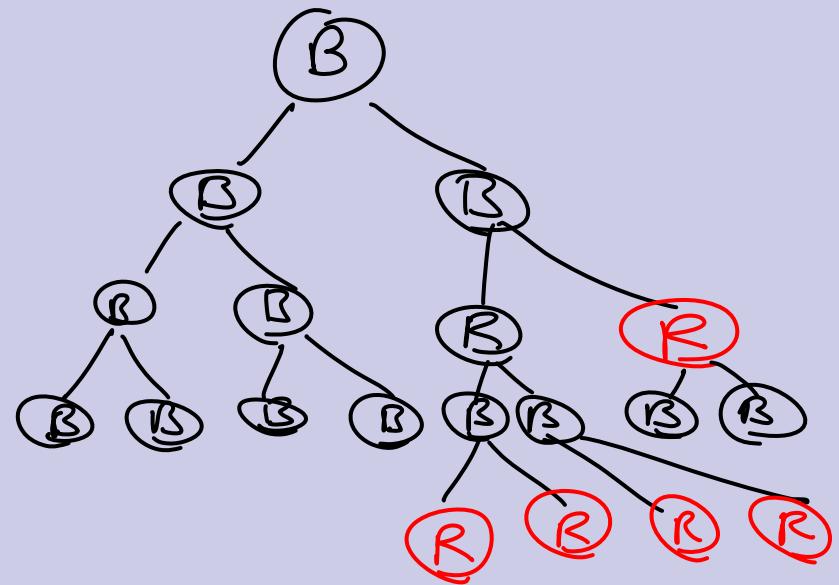
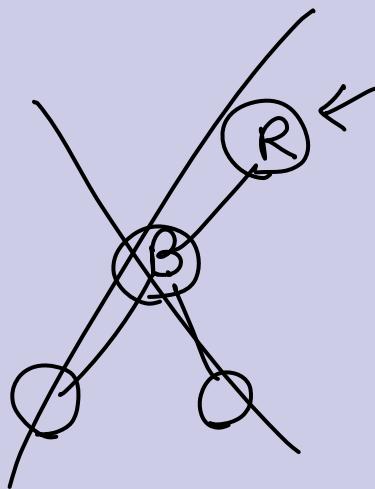
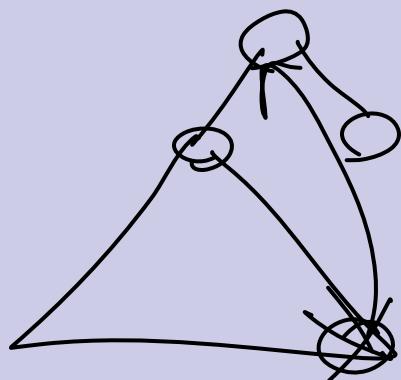


Black height is same as
height of 2-4 tree



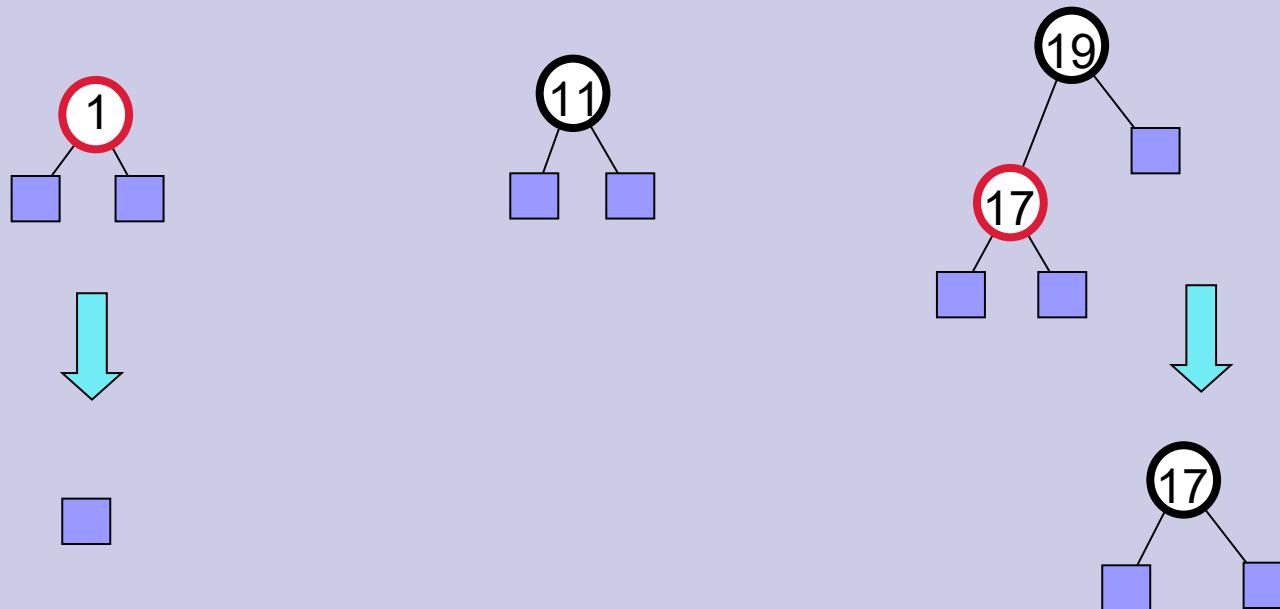
(R)





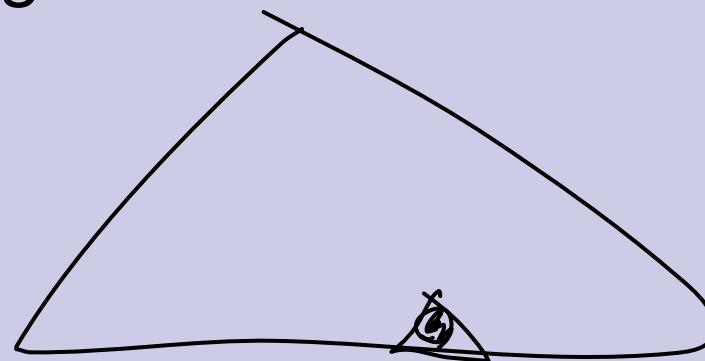
Deletion: preliminaries

- To delete a node we proceed as in a BST.
- Thus the node which is deleted is the parent of an external node.
- Hence it is either a leaf or the parent of a leaf.

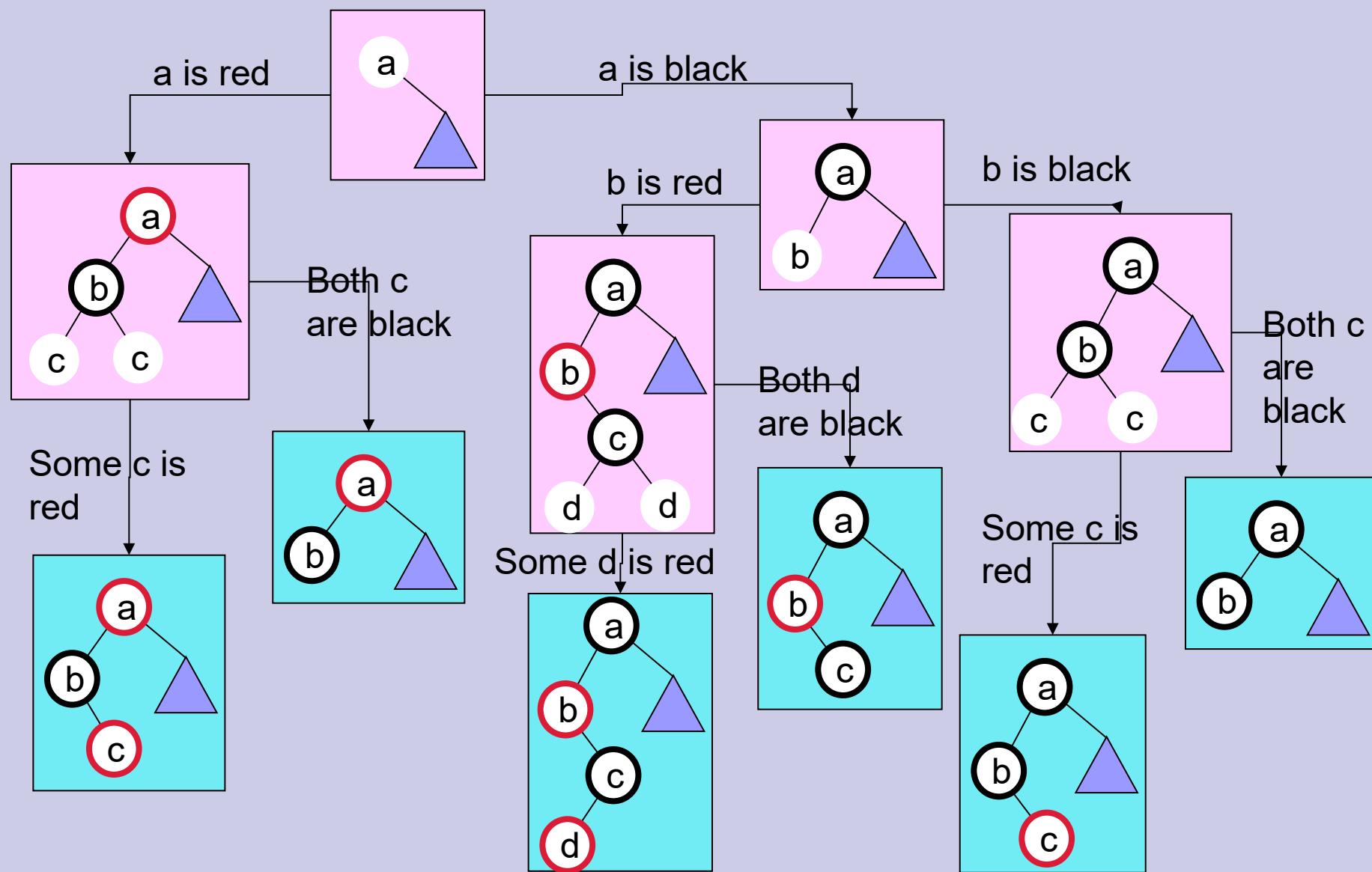


Deletion: preliminaries(2)

- Hence we can assume that the node deleted is a black leaf.
- Removing this reduces black depth of an external node by 1.
- Hence, in a general step, we consider how to reorganize the tree when the black height of some subtree goes down from h to $h-1$.

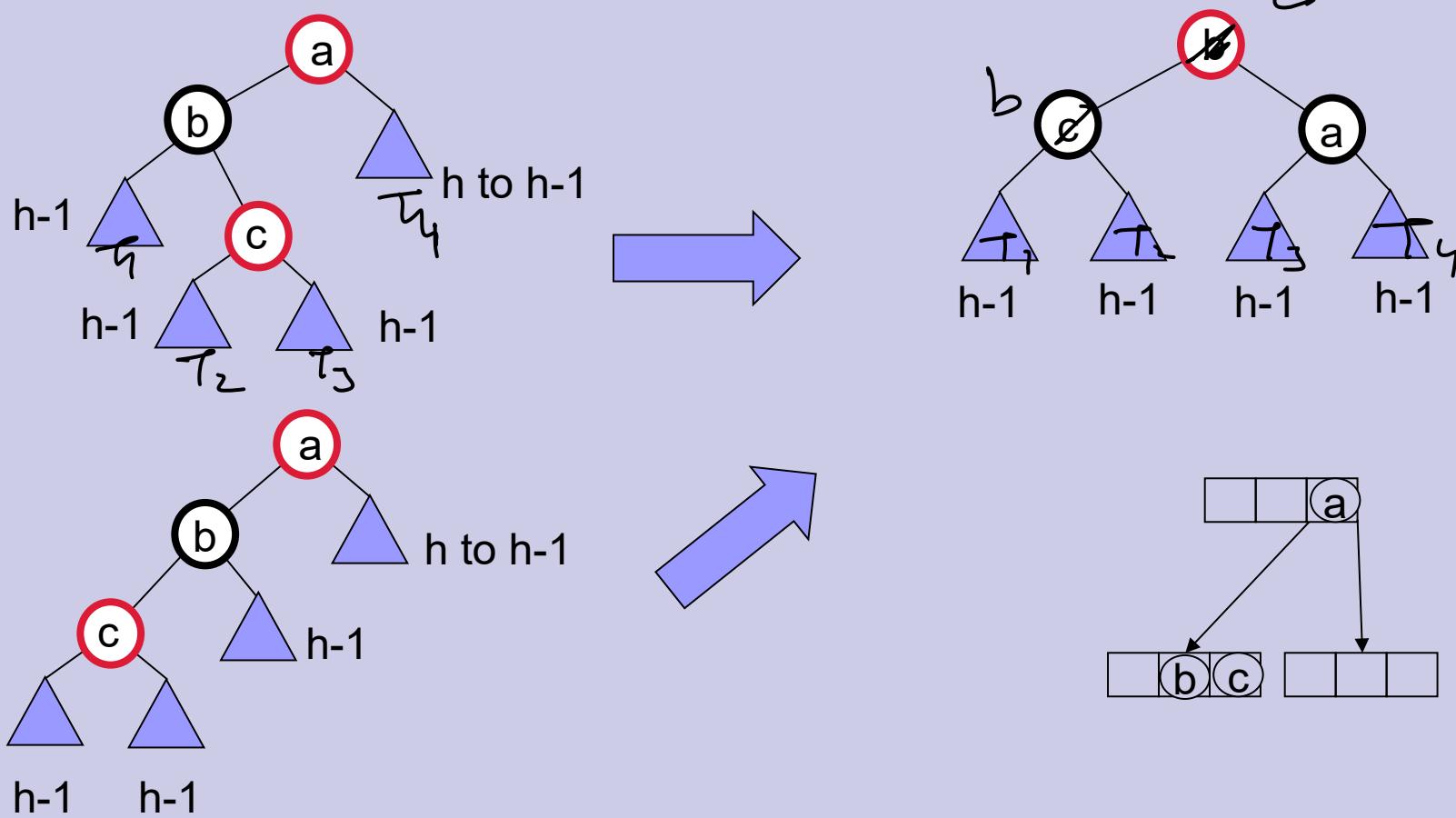


Deletion: the cases



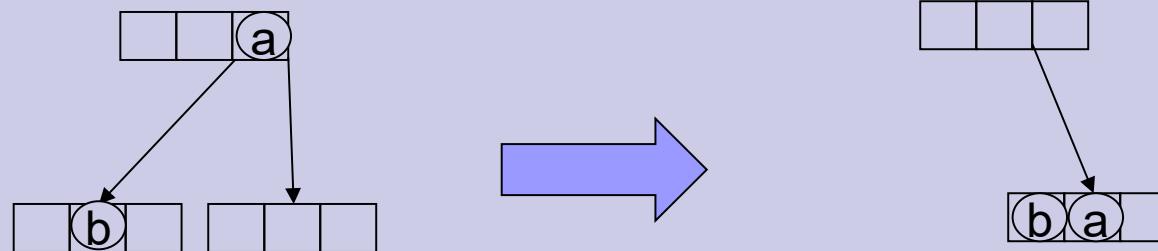
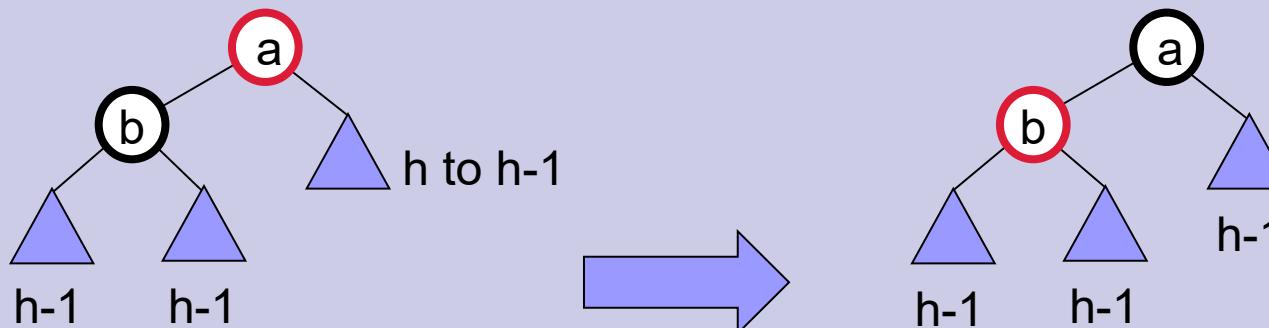
Deletion: case1.1

- If parent is a red node (a).
- Then it has a child (b) which must be black
- If b has a red child (c)



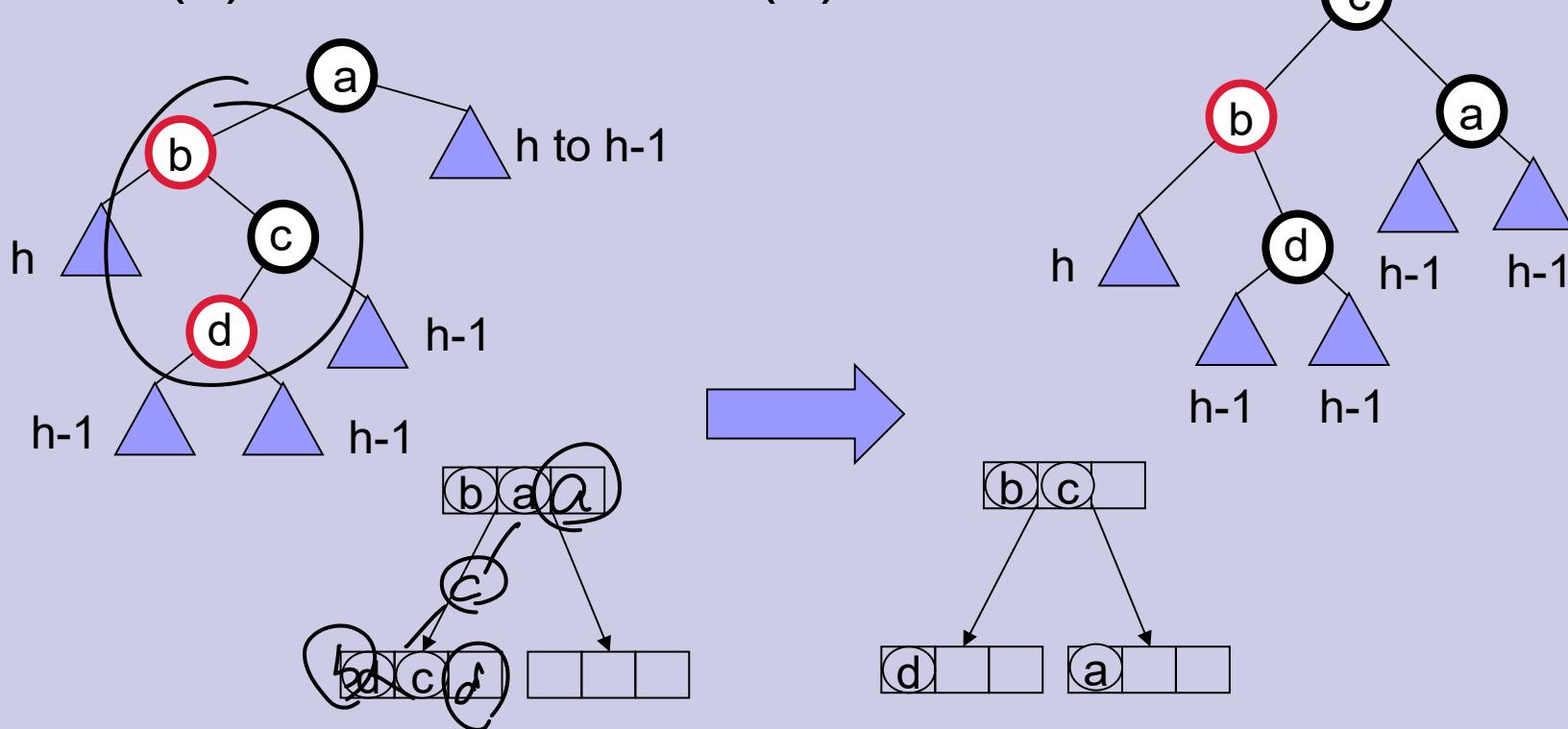
Deletion: case 1.2

- If parent is a red node (a).
- Then it has a child (b) which must be black
- If b has no red child



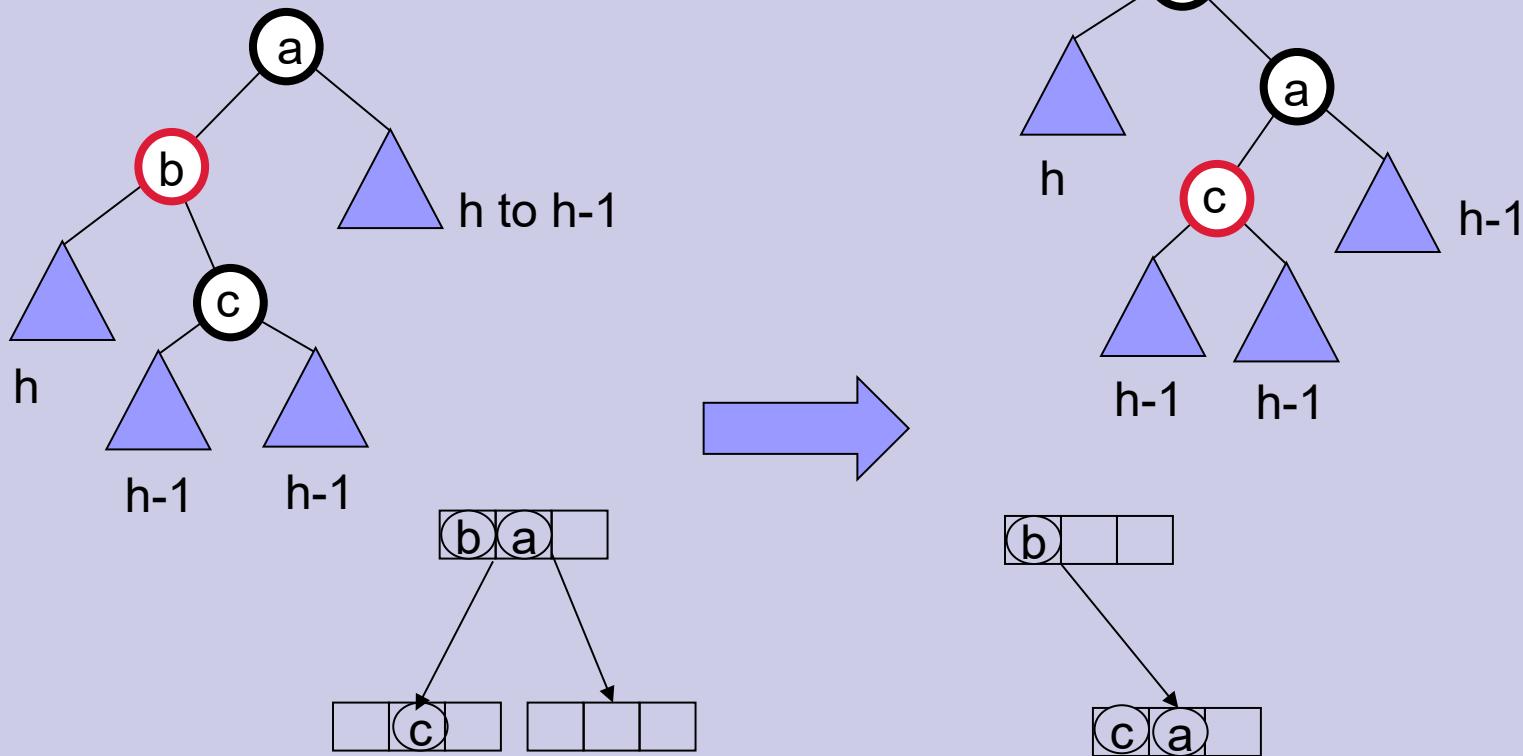
Deletion: case 2.1.1

- If parent is a black node (a).
- If a has a red child (b)
- Consider right child of b (c) which must be black.
- If (c) has a red child (d).



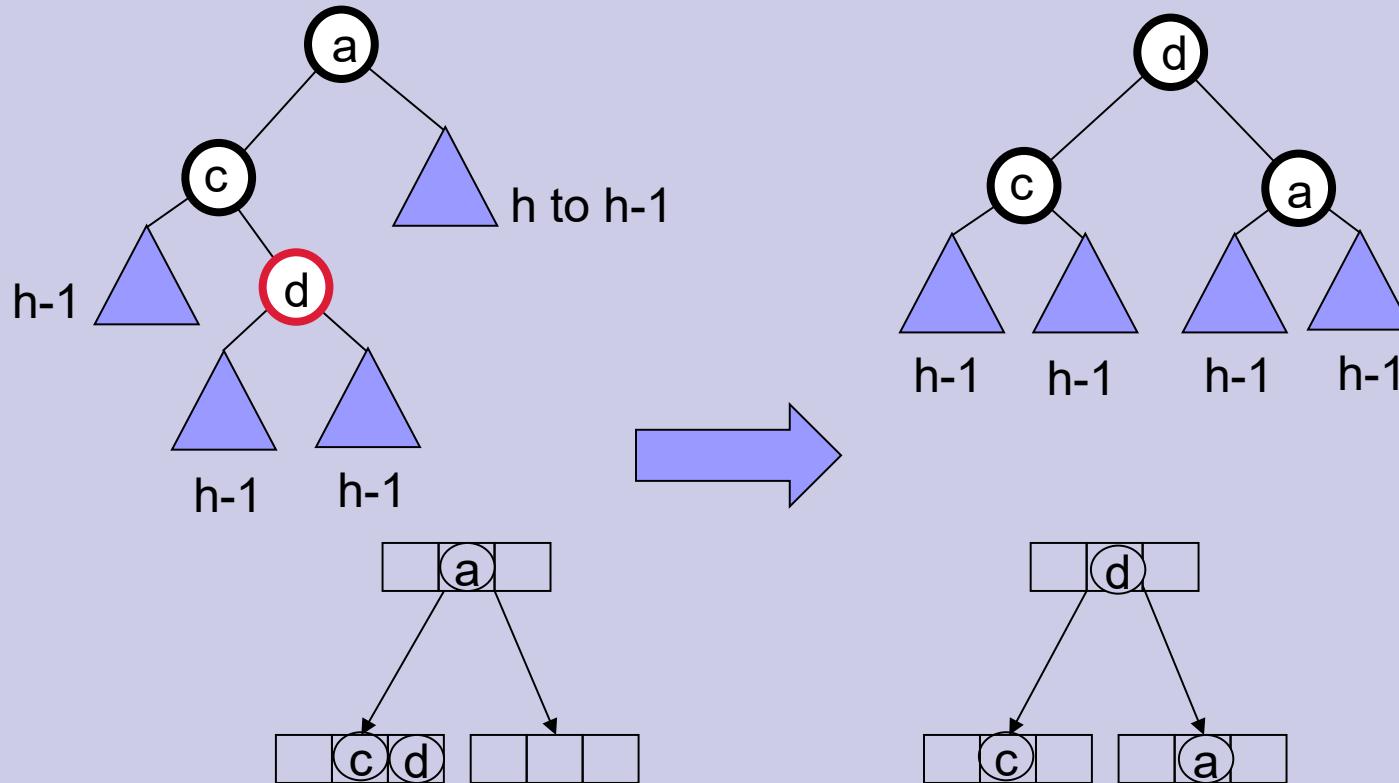
Deletion: case 2.1.2

- If parent is a black node (a).
- If a has a red child (b)
- Consider right child of b (c) which must be black.
- If (c) has no red child.



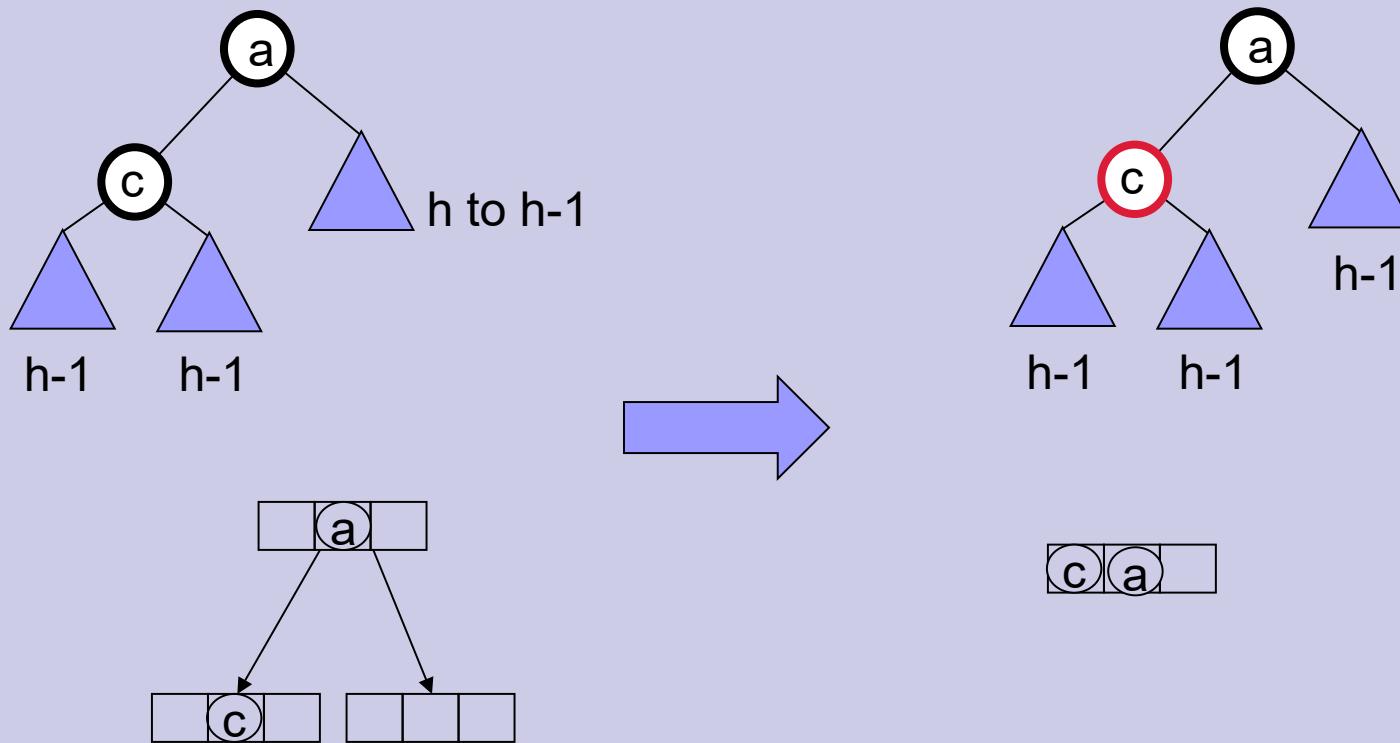
Deletion: case 2.2.1

- If parent is a black node (a).
- If the child of node a (c) is black.
- If (c) has a red child (d)



Deletion: case 2.2.2

- If parent is a black node (a).
- If the child of node a (c) is black.
- If (c) has no red child.



Deletion: Summary

- In all cases, except 2.2.2, deletion can be completed by a simple rotation/recoloring
- In case 2.2.2, the height of the subtree reduces and so we need to proceed up the tree.
- But in case 2.2.2 we only recolor nodes.
- Thus, if we proceed up the tree then we only need to recolor. Eventually we would do a rotation.

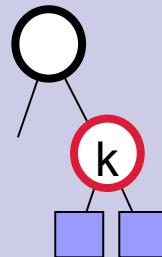
- Insertion in red-black trees
- a-b trees
 - What are they?
 - Insertion and deletion in a-b trees

Insertion in red-black trees

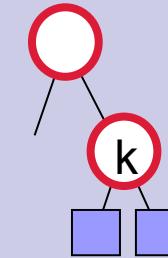
- Let k be the key being inserted
- As in the case of a BST we first search for k ; this gives us the place where we have to insert k .
- We create a new node with key k and insert it at this place.
- The new node is colored red.

Insertion(2)

- Since inserted node is colored red, the black height of the tree remains unchanged.
- However, if the parent of inserted node is also red then we have a double red problem.



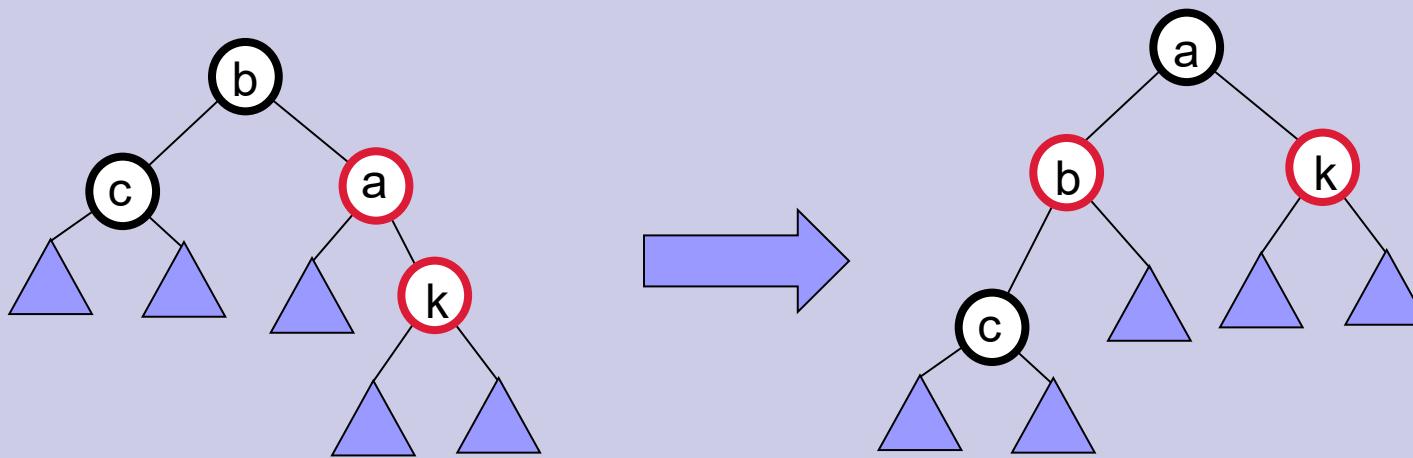
No problem



Double red problem

Insertion: case 1

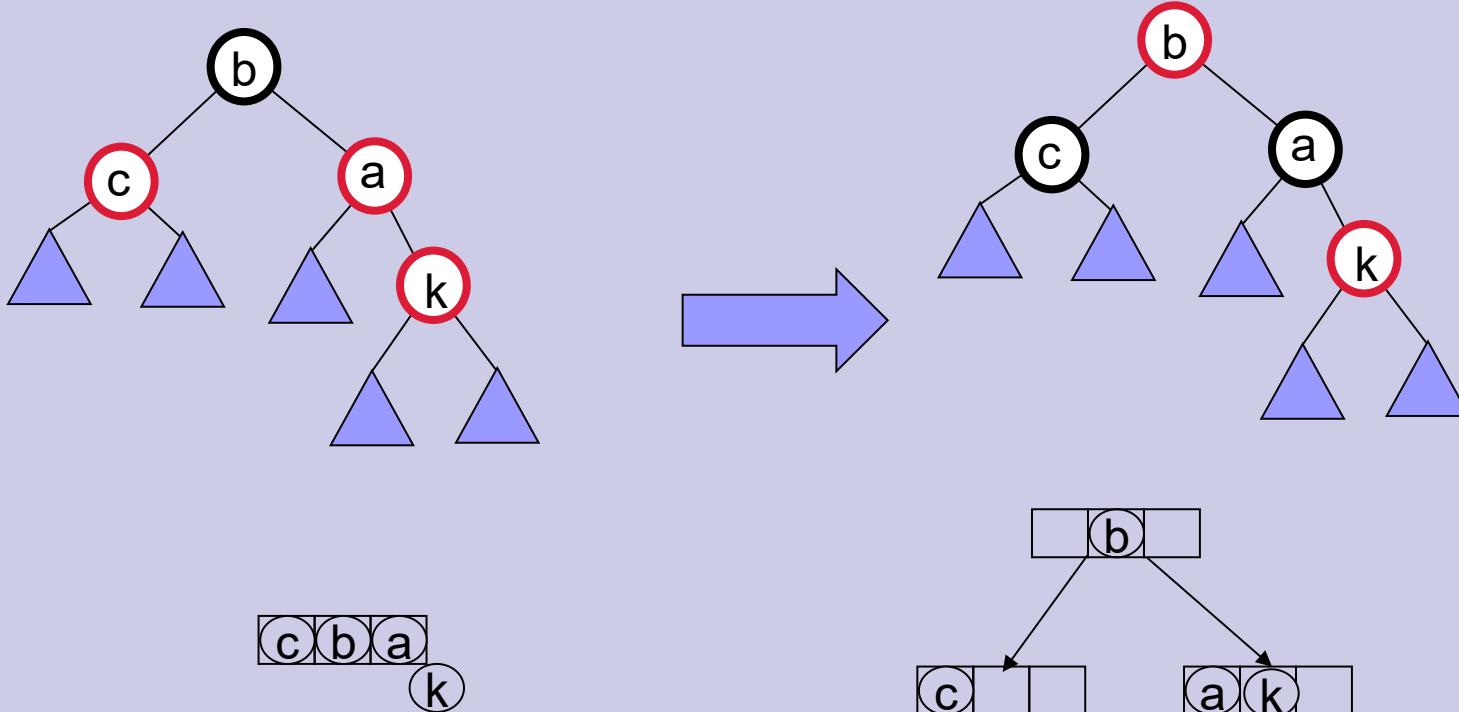
- Parent of inserted node (a) is red.
- Parent of (a) must be black (b)
- The other child of (b) is black (c).



The 2-4 tree node contains {b,a,k} and is malformed.
The rotation corrects the defect.

Insertion: Case 2

- Parent of inserted node (a) is red.
- Parent of (a) must be black (b)
- The other child of (b) is also red (c).



Insertion: case 2 (contd)

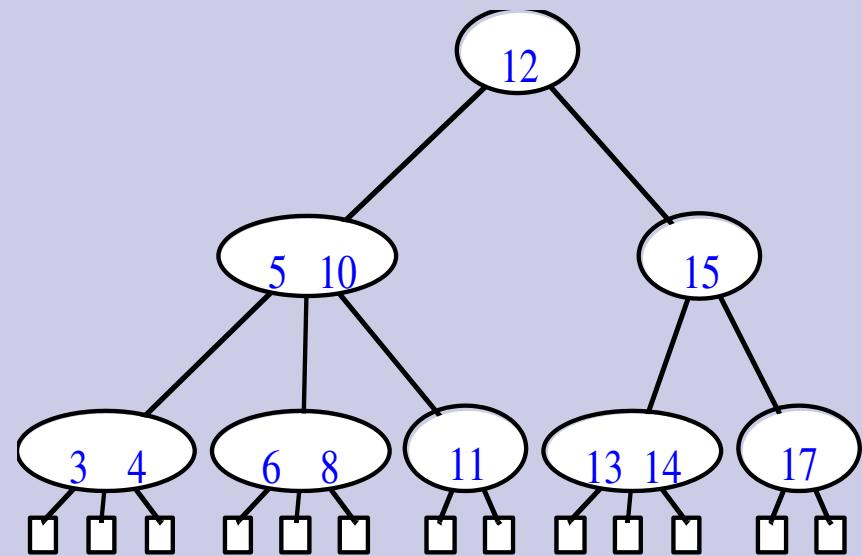
- The parent of b could also be red. In that case, the double red problem moves up a level.
- We repeat this process at the next level.
- Eventually, we might color the root red.
- In this case we recolor the root black. This increases the black depth of every external node by 1.
- In the 2-4 tree this corresponds to splitting the root.

Insertion and Deletion: Summary

- In both insertion and deletion we need to make at most one rotation.
- We might have to move up the tree but in doing so we only recolor nodes.
- Time taken is $O(\log n)$

(a,b) Trees

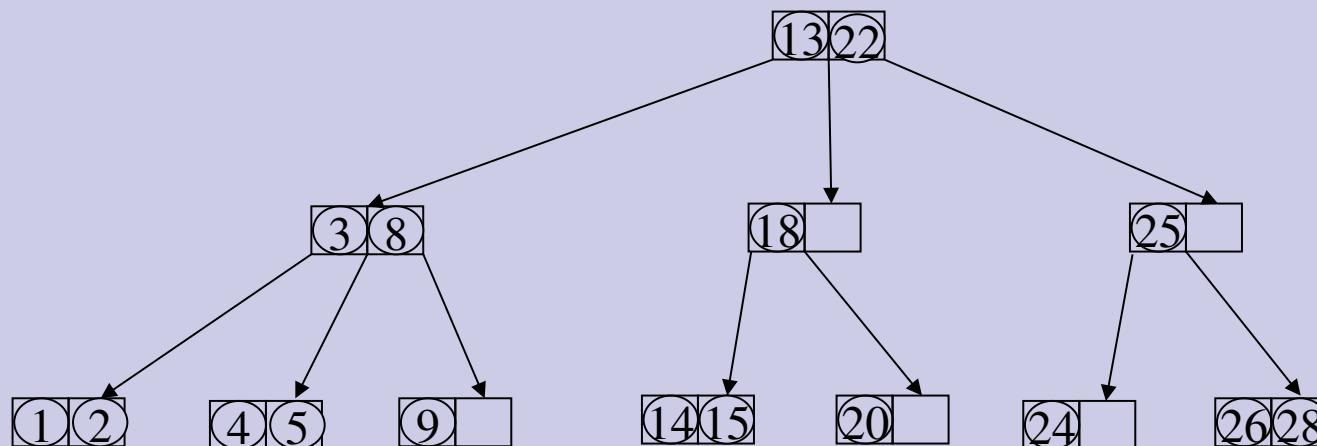
- A multiway search tree.
- Each node has at least a and at most b children.
- Root can have less than a children but it has at least 2 children.
- All leaf nodes are at the same level.
- Height h of (a,b) tree is at least $\log_b n$ and at most $\log_a n$.



Insertion

②1 ②3 ②9 ⑦

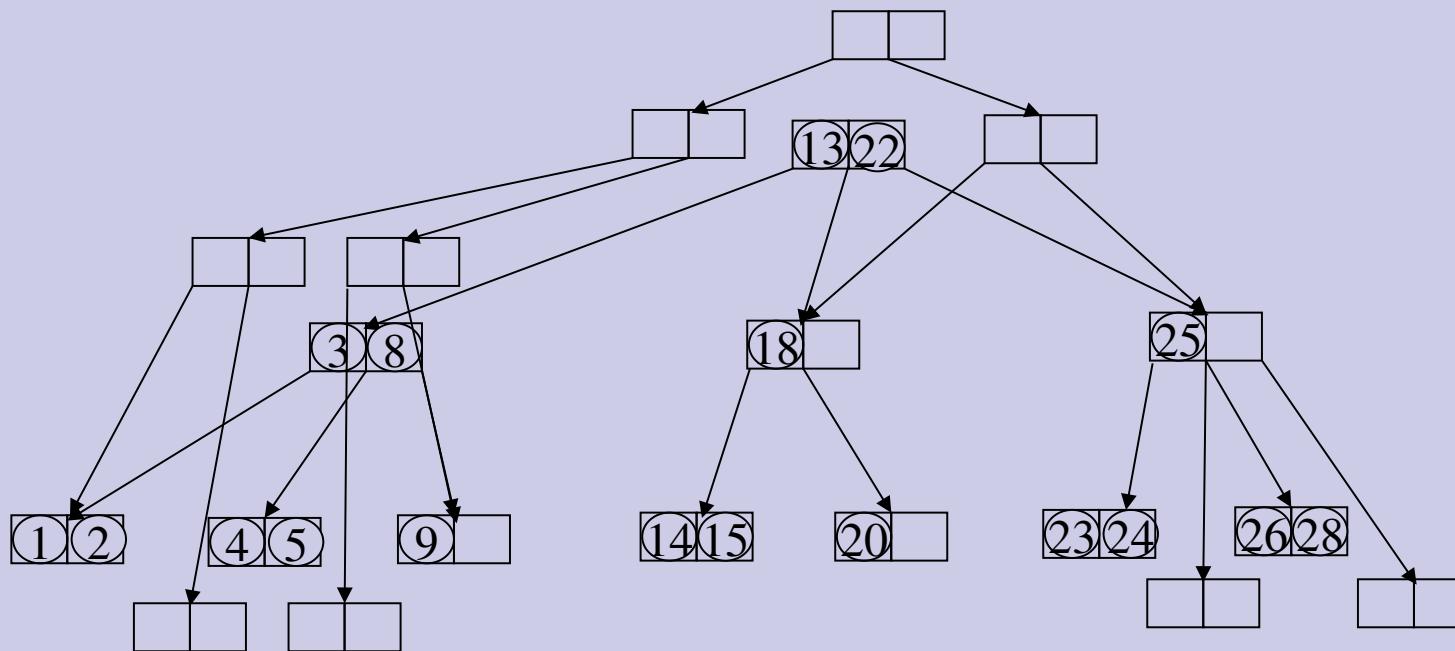
- No problem if the node has empty space



Insertion(2)

㉙ ⑦

- Nodes get split if there is insufficient space.
- The median key is promoted to the parent node and inserted there

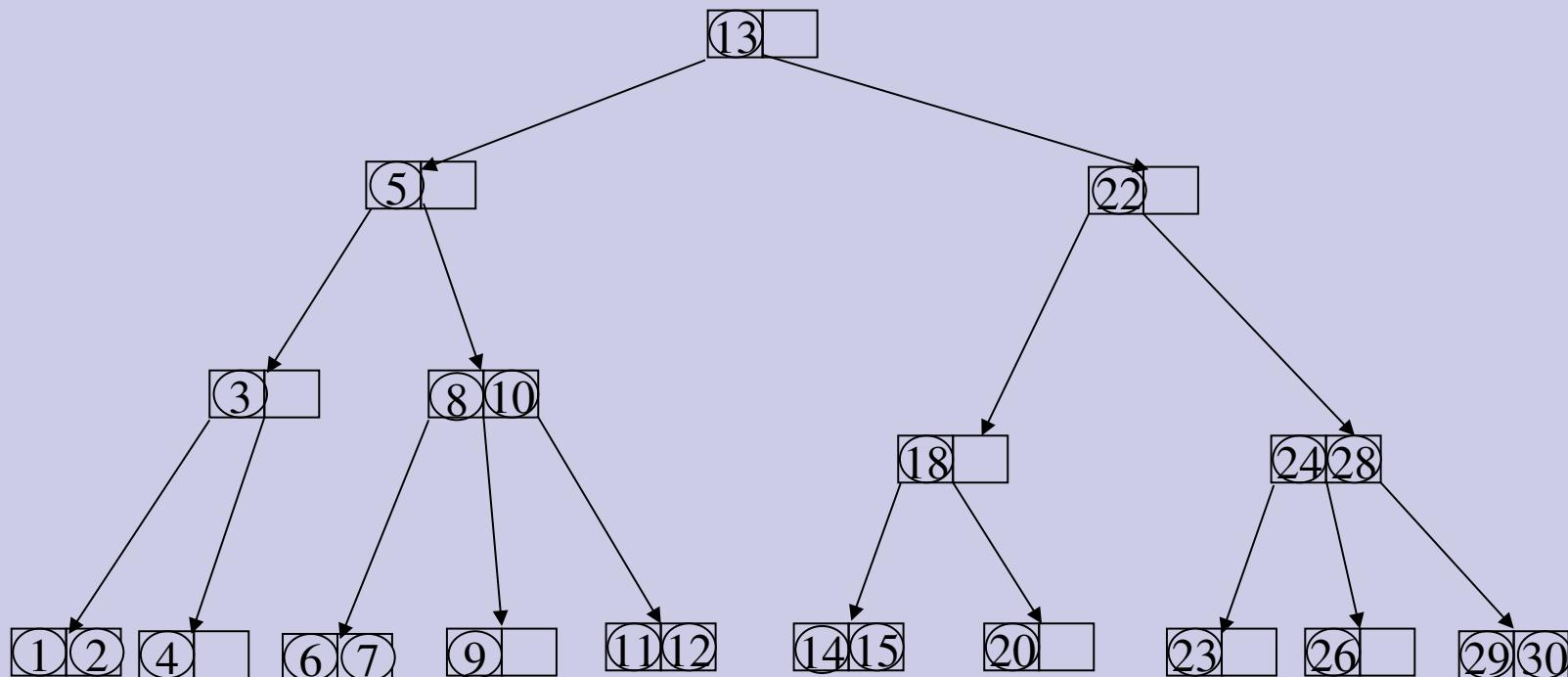


Insertion(3)

- A node is split when it has exactly b keys.
- One of these is promoted to the parent and the remaining are split between two nodes.
- Thus one node gets $\lceil \frac{b-1}{2} \rceil$ and the other $\lfloor \frac{b-1}{2} \rfloor$ keys.
- This implies that $a-1 \geq \lfloor \frac{b-1}{2} \rfloor$

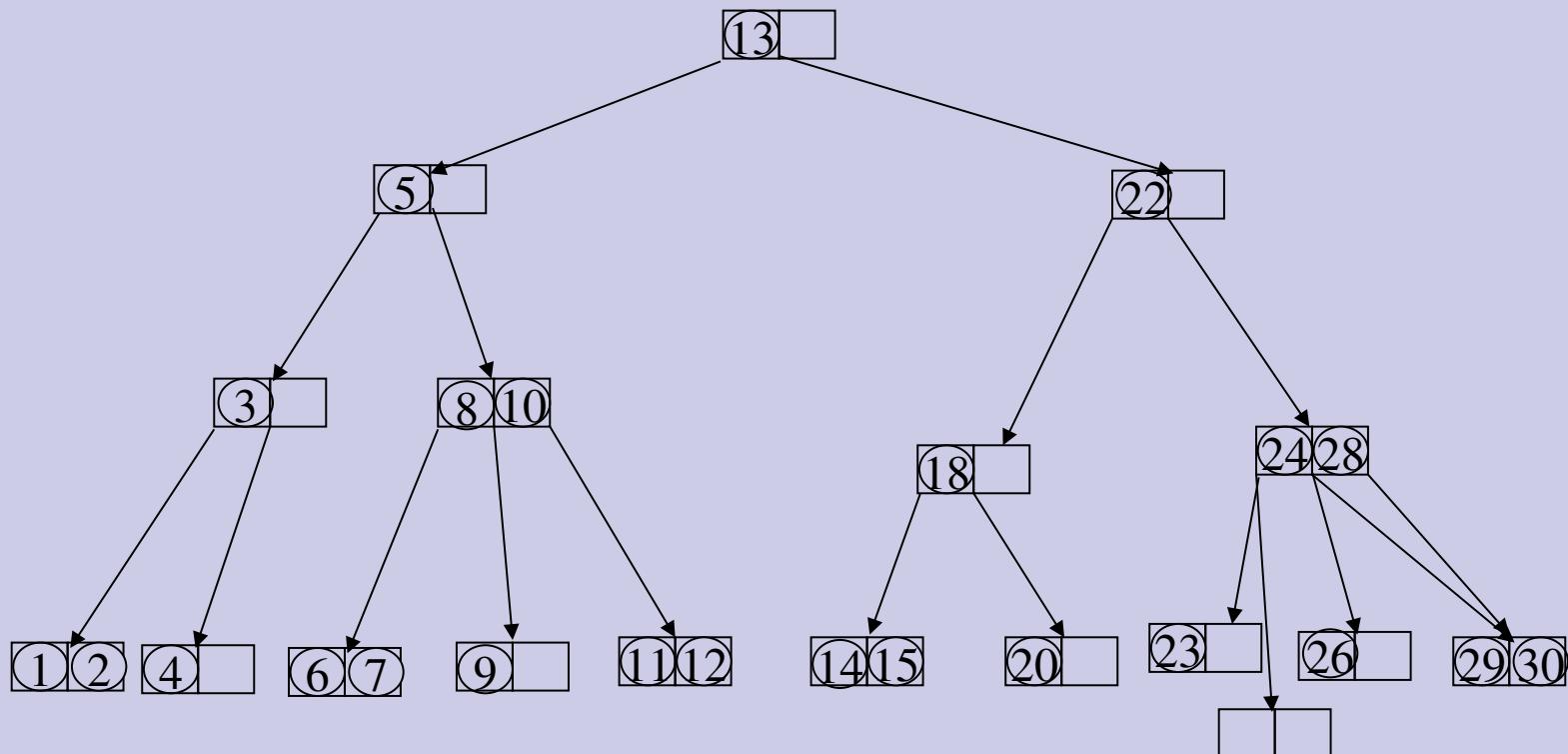
Deletion

- If after deleting a key a node becomes empty then we borrow a key from its sibling.
- Delete 20



Deletion(2)

- If sibling has only one key then we merge with it.
- The key in the parent node separating these two siblings moves down into the merged node.
- Delete 23



Deletion(3)

- In an (a,b) tree we will merge a node with its sibling if the node has $a-2$ keys and its sibling has $a-1$ keys.
- Thus the merged node has $2(a-1)$ keys.
- This implies that $2(a-1) \leq b-1$ which is equivalent to $a-1 \leq \lfloor \frac{b-1}{2} \rfloor$.
- Earlier too we argued that $a-1 \leq \lfloor \frac{b-1}{2} \rfloor$
- This implies $b \geq 2a-1$
- For $a=2$, $b \geq 3$

Conclusion

- The height of a (a,b) tree is $O(\log n)$.
- $b \geq 2a-1$.
- For insertion and deletion we take time proportional to the height.

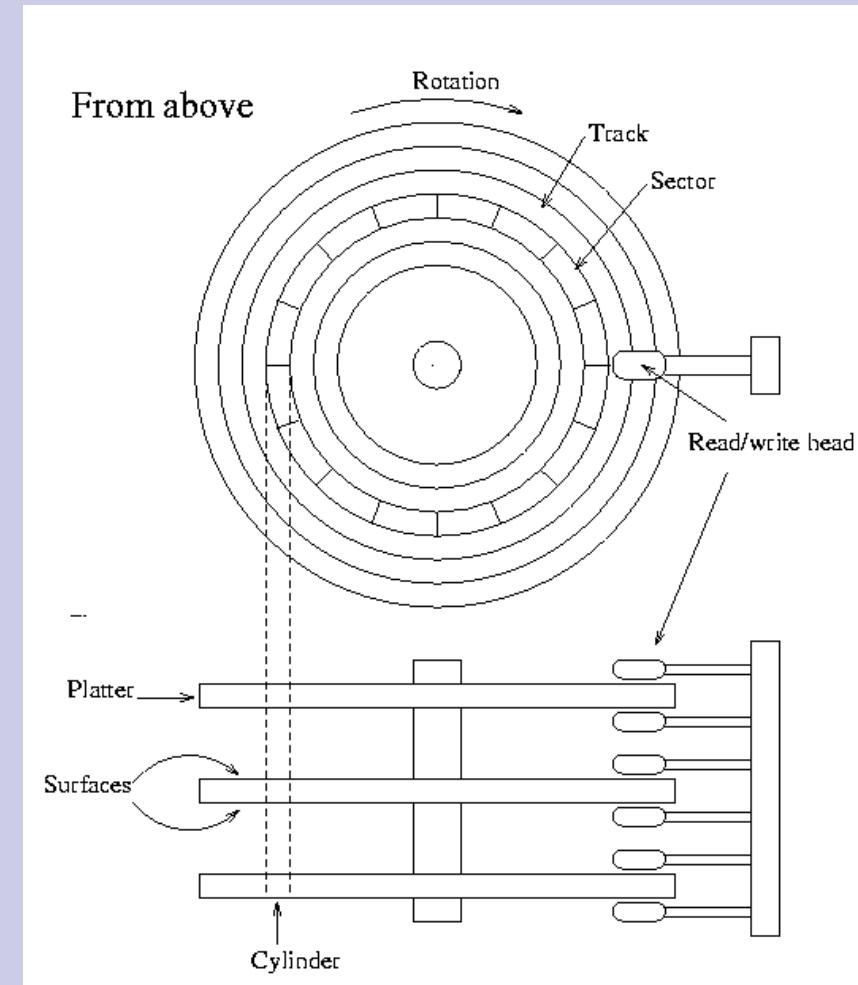
Disk Based Data Structures

- So far search trees were limited to main memory structures
 - Assumption: the dataset organized in a search tree fits in main memory (including the tree overhead)
- Counter-example: transaction data of a bank > 1 GB per day
 - use secondary storage media (punch cards, hard disks, magnetic tapes, etc.)
- Consequence: make a search tree structure secondary-storage-enabled



Hard Disks

- Large amounts of storage, but slow access!
- Identifying a page takes a long time (seek time plus rotational delay – 5-10ms), reading it is fast
 - It pays off to read or write data in **pages** (or blocks) of 2-16 Kb in size.



Algorithm analysis

- The running time of disk-based algorithms is measured in terms of
 - computing time (CPU)
 - number of disk accesses
 - sequential reads
 - random reads
- Regular main-memory algorithms that work one data element at a time can not be “ported” to secondary storage in a straight-forward way

Principles

- Pointers in data structures are no longer addresses in main memory but locations in files
- If x is a pointer to an object
 - if x is in main memory $\text{key}[x]$ refers to it
 - otherwise $\text{DiskRead}(x)$ reads the object from disk into main memory ($\text{DiskWrite}(x)$ – writes it back to disk)

Principles (2)

□ A typical working pattern

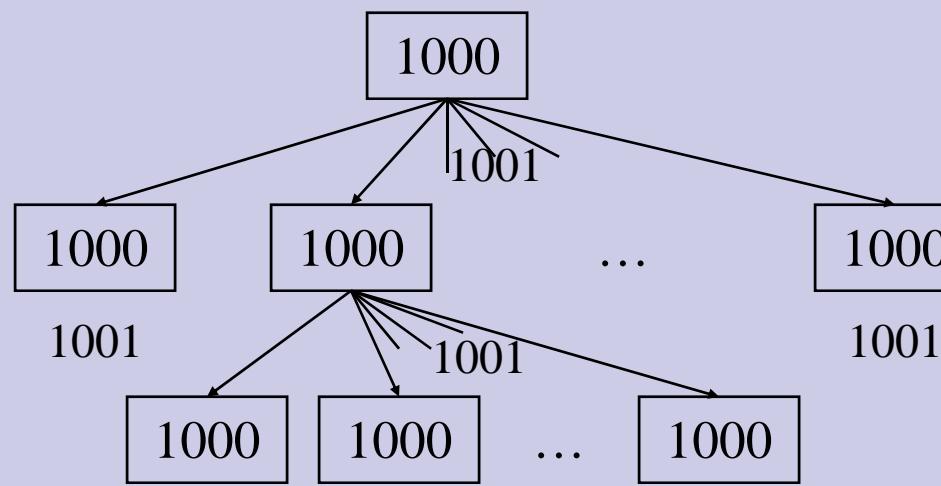
```
01 ...
02 x ← a pointer to some object
03 DiskRead(x)
04 operations that access and/or modify x
05 DiskWrite(x) //omitted if nothing changed
06 other operations, only access no modify
07 ...
```

□ Operations:

- DiskRead(x:pointer_to_a_node)
- DiskWrite(x:pointer_to_a_node)
- AllocateNode():pointer_to_a_node

Binary-trees vs. B-trees

- Size of B-tree nodes is determined by the page size. One page – one node.
- A B-tree of height 2 may contain > 1 billion keys!
- Heights of Binary-tree and B-tree are logarithmic
 - B-tree: logarithm of base, e.g., 1000
 - Binary-tree: logarithm of base 2



1 node
1000 keys

1001 nodes,
1,001,000 keys

1,002,001 nodes,
1,002,001,000 keys

B-tree Definitions

- Node x has fields
 - $n[x]$: the number of keys of that the node
 - $\text{key}_1[x] \leq \dots \leq \text{key}_{n[x]}[x]$: the keys in ascending order
 - $\text{leaf}[x]$: true if leaf node, false if internal node
 - if internal node, then $c_1[x], \dots, c_{n[x]+1}[x]$: pointers to children
- Keys separate the ranges of keys in the subtrees. If k_i is an arbitrary key in the subtree $c_i[x]$ then $k_i \leq \text{key}_i[x] \leq k_{i+1}$

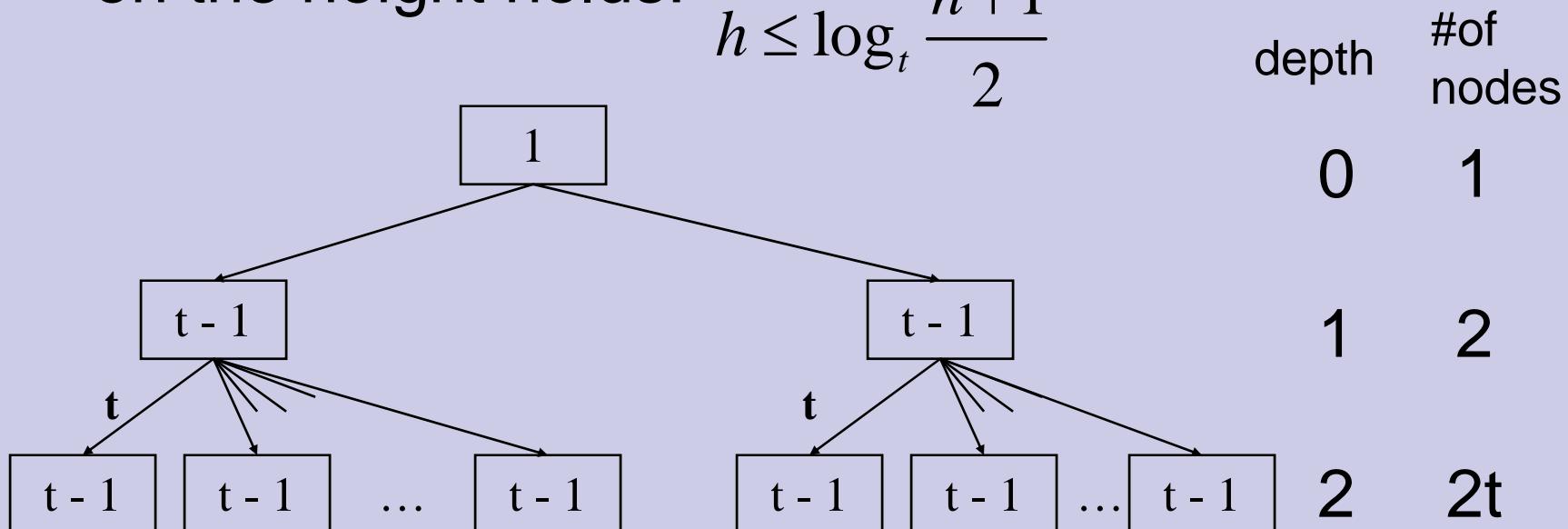
B-tree Definitions (2)

- Every leaf has the same depth
- In a B-tree of a **degree** t all nodes except the root node have between t and $2t$ children (i.e., between $t-1$ and $2t-1$ keys).
- The root node has between 0 and $2t$ children (i.e., between 0 and $2t-1$ keys)

Height of a B-tree

- B-tree T of height h , containing $n \geq 1$ keys and minimum degree $t \geq 2$, the following restriction on the height holds:

$$h \leq \log_t \frac{n+1}{2}$$



$$n \geq 1 + (t-1) \sum_{i=1}^h 2t^{i-1} = 2t^h - 1$$

B-tree Operations

- An implementation needs to support the following B-tree operations
 - **Searching** (simple)
 - **Creating** an empty tree (trivial)
 - **Insertion** (complex)
 - **Deletion** (complex)

Searching

- Straightforward generalization of a binary tree search

```
BTreeSearch(x,k)
01 i ← 1
02 while i ≤ n[x] and k > keyi[x]
03     i ← i+1
04 if i ≤ n[x] and k = keyi[x] then
05     return(x,i)
06 if leaf[x] then
07     return NIL
08 else DiskRead(ci[x])
09         return BTtreeSearch(ci[x],k)
```

Creating an Empty Tree

- Empty B-tree = create a root & write it to disk!

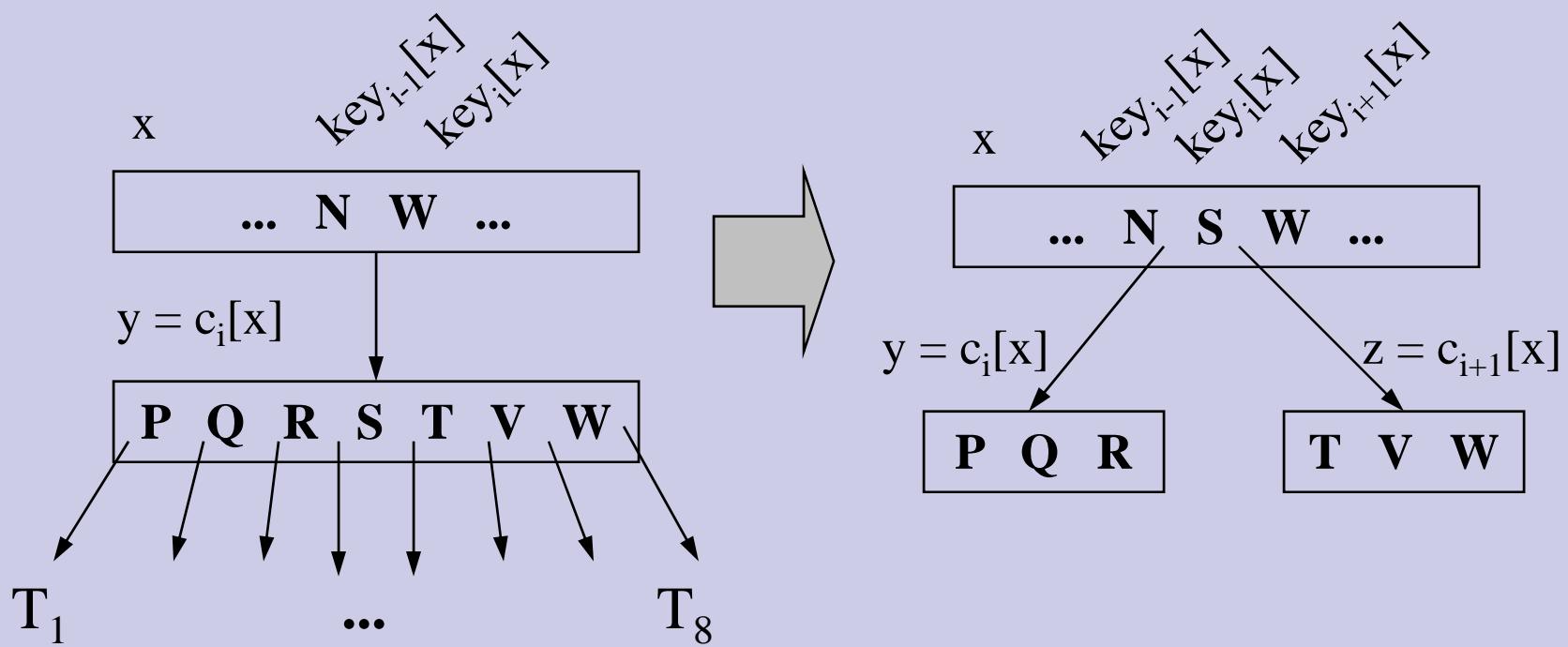
```
BTreeCreate(T)
01 x ← AllocateNode();
02 leaf[x] ← TRUE;
03 n[x] ← 0;
04 DiskWrite(x);
05 root[T] ← x
```

Splitting Nodes

- Nodes fill up and reach their maximum capacity $2t - 1$
- Before we can insert a new key, we have to “make room,” i.e., split nodes

Splitting Nodes (2)

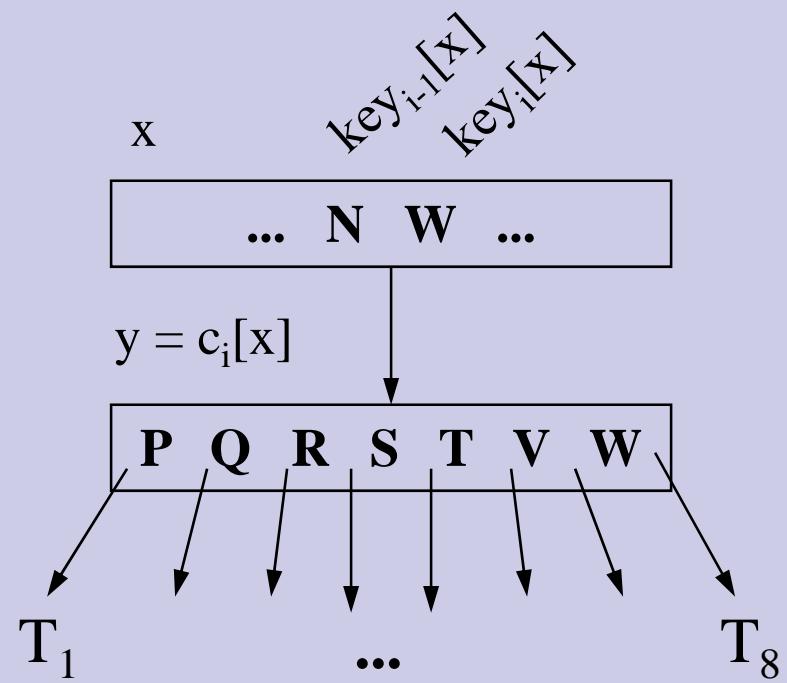
- Result: one key of x moves up to parent + 2 nodes with $t-1$ keys



Splitting Nodes (2)

```
BTreeSplitChild(x,i,y)
z ← AllocateNode()
leaf[z] ← leaf[y]
n[z] ← t-1
for j ← 1 to t-1
    keyj[z] ← keyj+t[y]
if not leaf[y] then
    for j ← 1 to t
        cj[z] ← cj+t[y]
n[y] ← t-1
for j ← n[x]+1 downto i+1
    cj+1[x] ← cj[x]
ci+1[x] ← z
for j ← n[x] downto i
    keyj+1[x] ← keyj[x]
keyi[x] ← keyt[y]
n[x] ← n[x]+1
DiskWrite(y)
DiskWrite(z)
DiskWrite(x)
```

x: parent node
y: node to be split and child of x
i: index in x
z: new node



Split: Running Time

- A local operation that does not traverse the tree
- $\Theta(t)$ CPU-time, since two loops run t times
- 3 I/Os

Inserting Keys

- Done recursively, by starting from the root and recursively traversing down the tree to the leaf level
- Before descending to a lower level in the tree, make sure that the node contains $< 2t - 1$ keys:
 - so that if we split a node in a lower level we will have space to include a new key

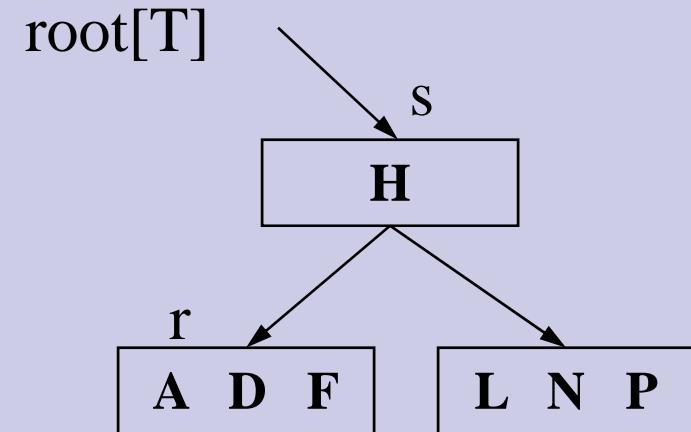
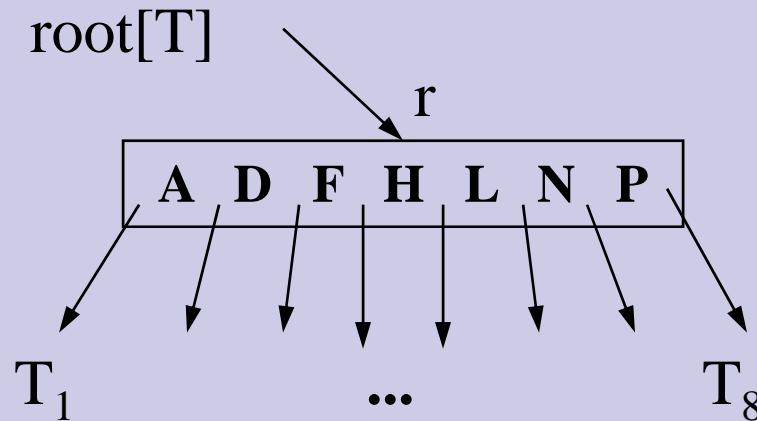
Inserting Keys (2)

- Special case: root is full (BtreeInsert)

```
BTreeInsert(T)
    r ← root[T]
    if n[r] = 2t - 1 then
        s ← AllocateNode()
        root[T] ← s
        leaf[s] ← FALSE
        n[s] ← 0
        c1[s] ← r
        BTreesplitChild(s,1,r)
        BTreearrangeNonFull(s,k)
    else
        BTreearrangeNonFull(r,k)
```

Splitting the Root

- Splitting the root requires the creation of a new root



- The tree grows at the top instead of the bottom

Inserting Keys

- BtreeNonFull tries to insert a key k into a node x , which is **assumed to be non-full** when the procedure is called
- BTreelnsert and the recursion in BTreelnsertNonFull guarantee that this assumption is true!

Inserting Keys: Pseudo Code

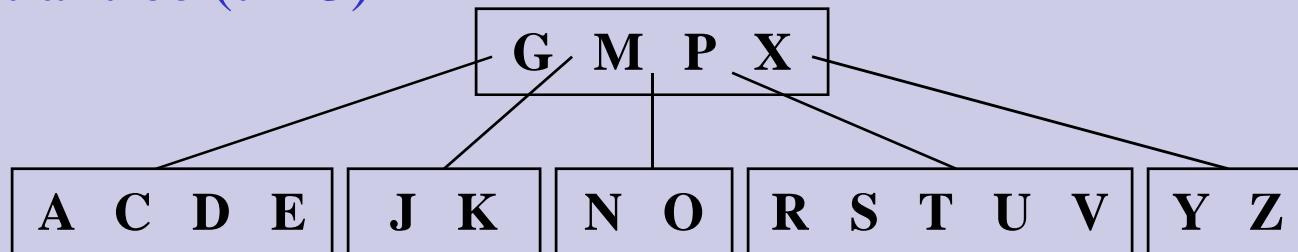
```
BTreeInsertNonFull(x, k)
```

```
01 i ← n[x]
02 if leaf[x] then
03     while i ≥ 1 and k < keyi[x]
04         keyi+1[x] ← keyi[x]
05         i ← i - 1                                leaf insertion
06     keyi+1[x] ← k
07     n[x] ← n[x] + 1
08     DiskWrite(x)

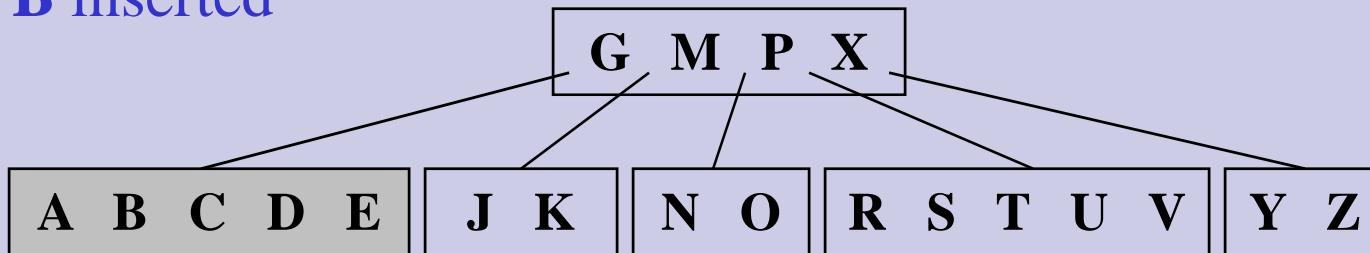
09 else while i ≥ 1 and k < keyi[x]
10     i ← i - 1
11     i ← i + 1                                internal node:
12     DiskRead ci[x]                          traversing tree
13     if n[ci[x]] = 2t - 1 then
14         BTreeSplitChild(x, i, ci[x])
15         if k > keyi[x] then
16             i ← i + 1
17         BTreeInsertNonFull(ci[x], k)
```

Insertion: Example

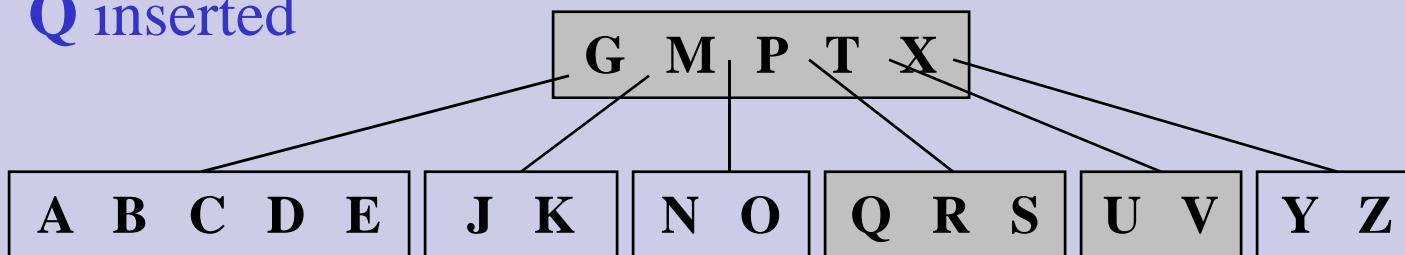
initial tree ($t = 3$)



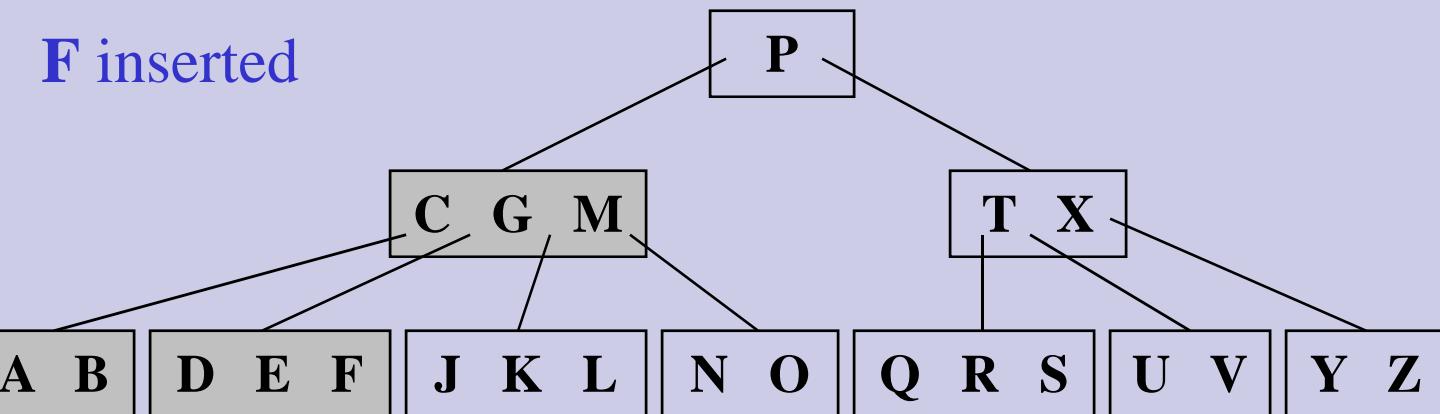
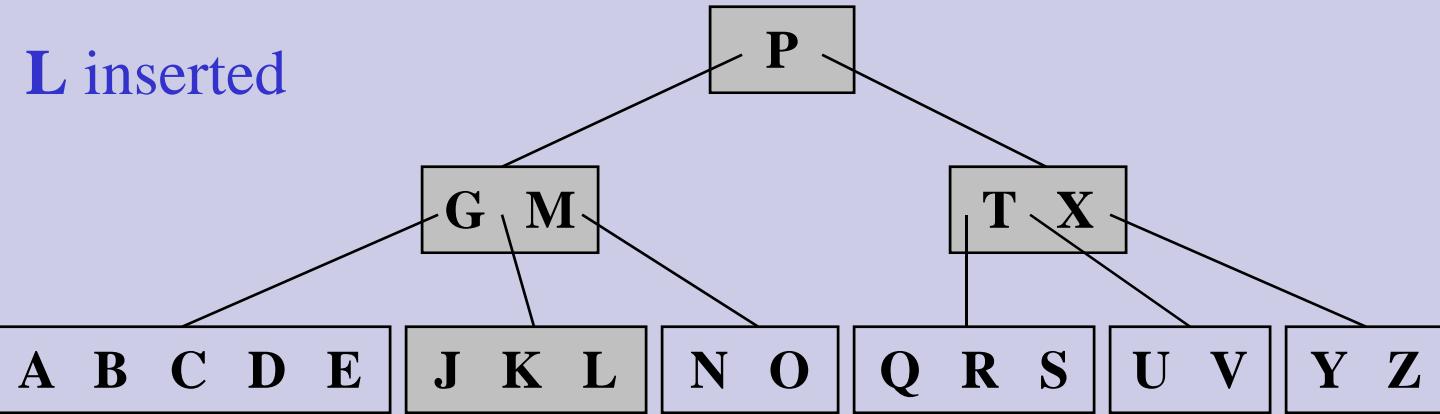
B inserted



Q inserted



Insertion: Example (2)



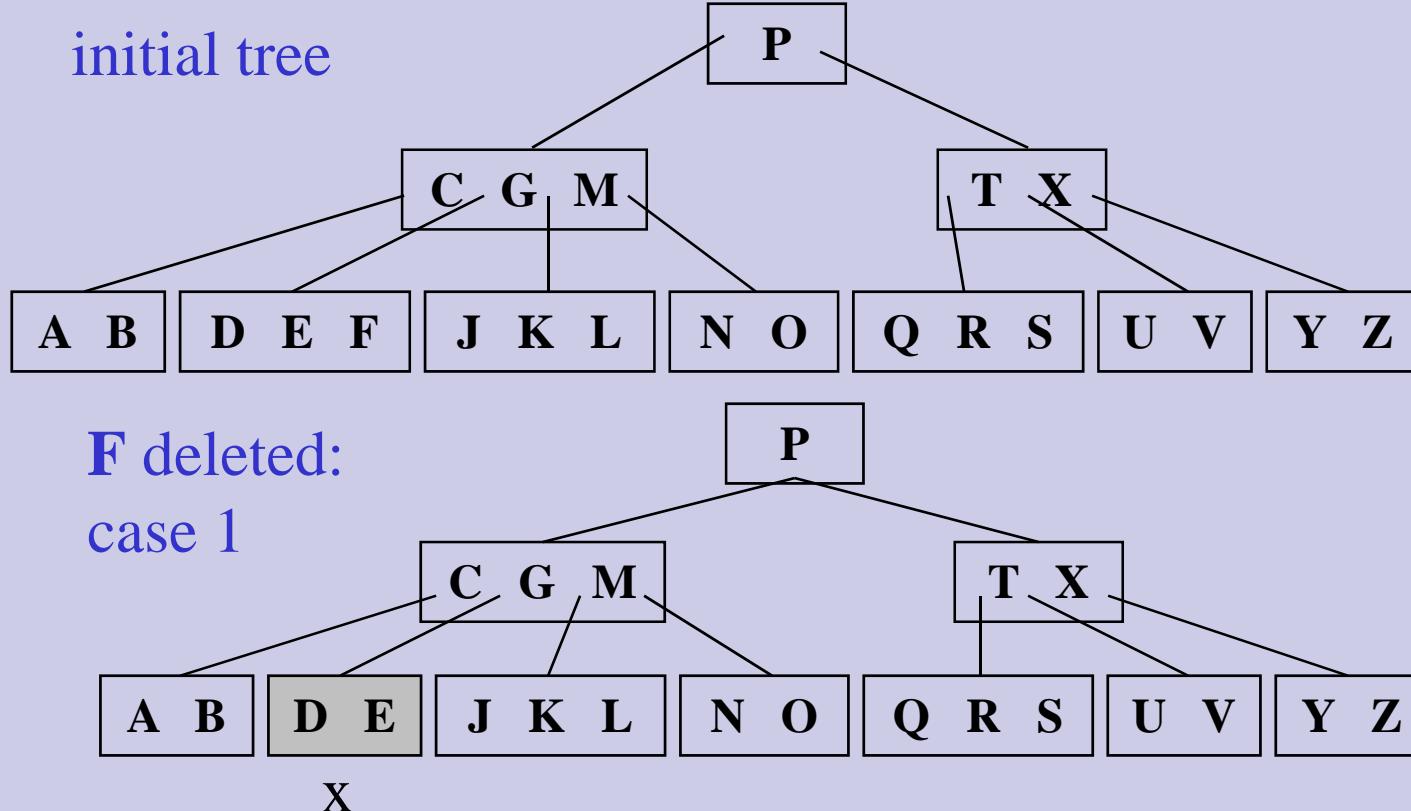
Insertion: Running Time

- Disk I/O: $O(h)$, since only $O(1)$ disk accesses are performed during recursive calls of BTreelInsertNonFull
- CPU: $O(th) = O(t \log_t n)$
- At any given time there are $O(1)$ number of disk pages in main memory

Deleting Keys

- Done recursively, by starting from the root and recursively traversing down the tree to the leaf level
- Before descending to a lower level in the tree, make sure that the node contains $\geq t$ keys (cf. insertion $< 2t - 1$ keys)
- BtreeDelete distinguishes three different stages/scenarios for deletion
 - Case 1: key k found in leaf node
 - Case 2: key k found in internal node
 - Case 3: key k suspected in lower level node

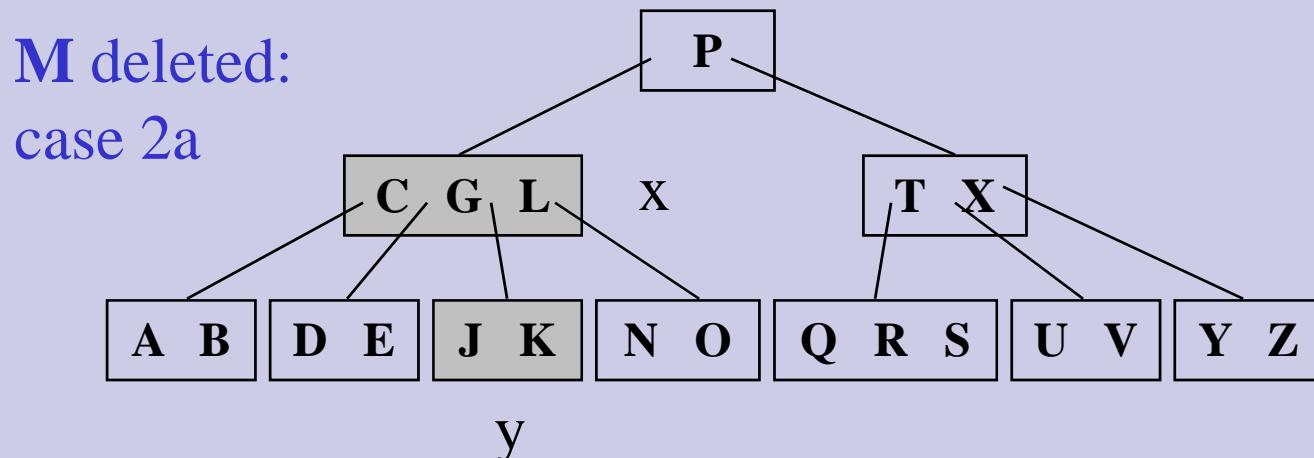
Deleting Keys (2)



- Case 1: If the key k is in node x , and x is a leaf, delete k from x

Deleting Keys (3)

- Case 2: If the key k is in node x , and x is not a leaf, delete k from x
 - a) If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the sub-tree rooted at y . Recursively delete k' , and replace k with k' in x .
 - b) Symmetrically for successor node z



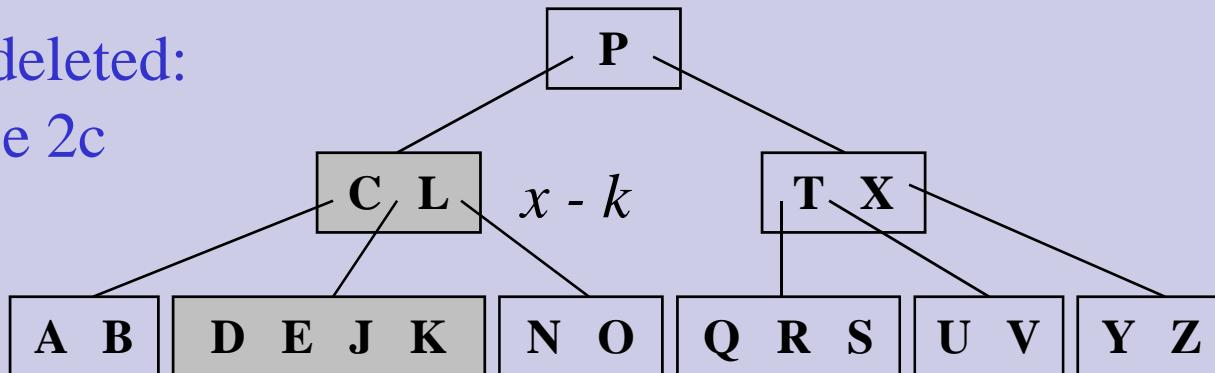




Deleting Keys (4)

- If both y and z have only $t - 1$ keys, **merge** k with the contents of z into y , so that x loses both k and the pointers to z , and y now contains $2t - 1$ keys. Free z and recursively delete k from y .

G deleted:
case 2c

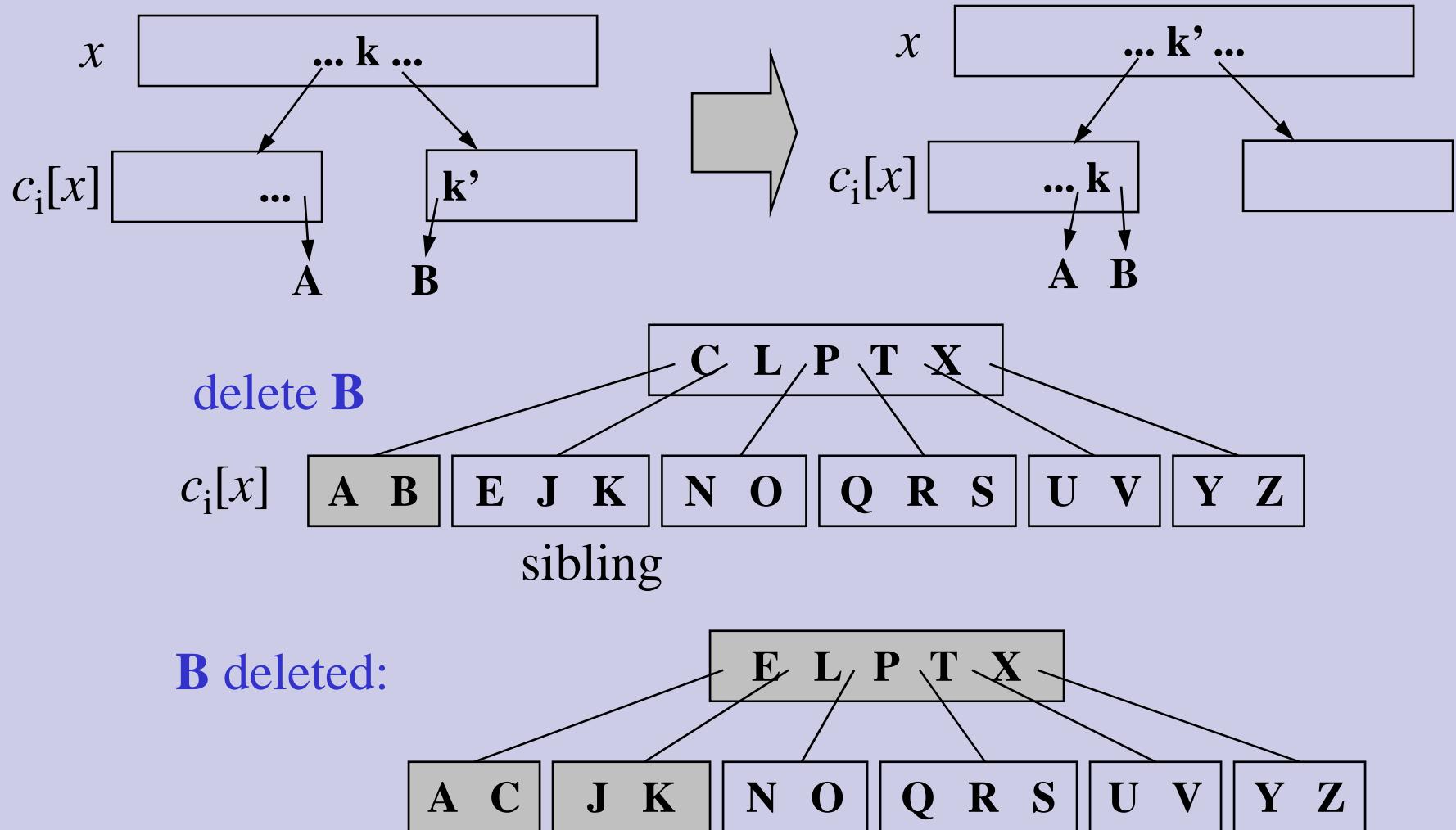


$$y = y + k + z - k$$

Deleting Keys - Distribution

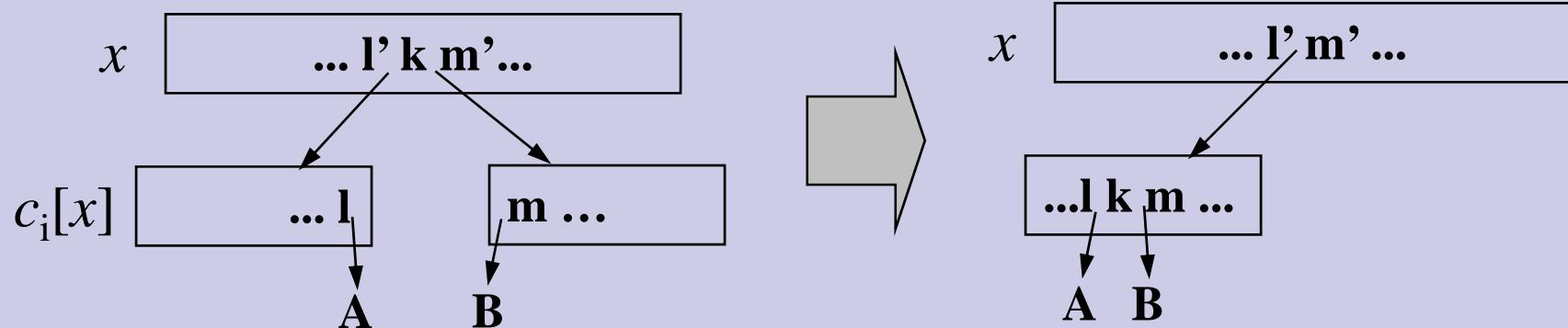
- Descending down the tree: if k not found in current node x , find the sub-tree $c_i[x]$ that has to contain k .
- If $c_i[x]$ has only $t - 1$ keys take action to ensure that we descent to a node of size at least t .
- We can encounter two cases.
 - If $c_i[x]$ has only $t - 1$ keys, but a sibling with at least t keys, give $c_i[x]$ an extra key by moving a key from x to $c_i[x]$, moving a key from $c_i[x]$'s immediate left and right sibling up into x , and moving the appropriate child from the sibling into $c_i[x]$ - ***distribution***

Deleting Keys – Distribution(2)

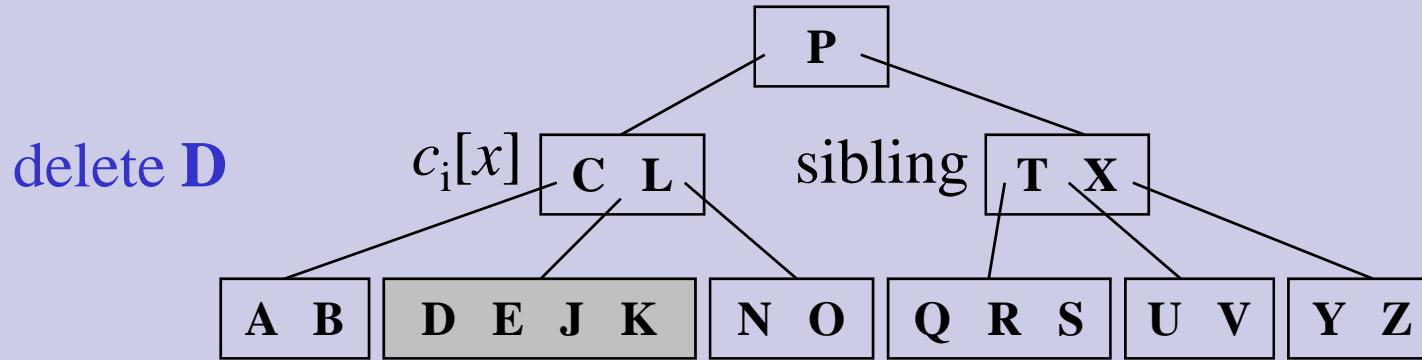


Deleting Keys - Merging

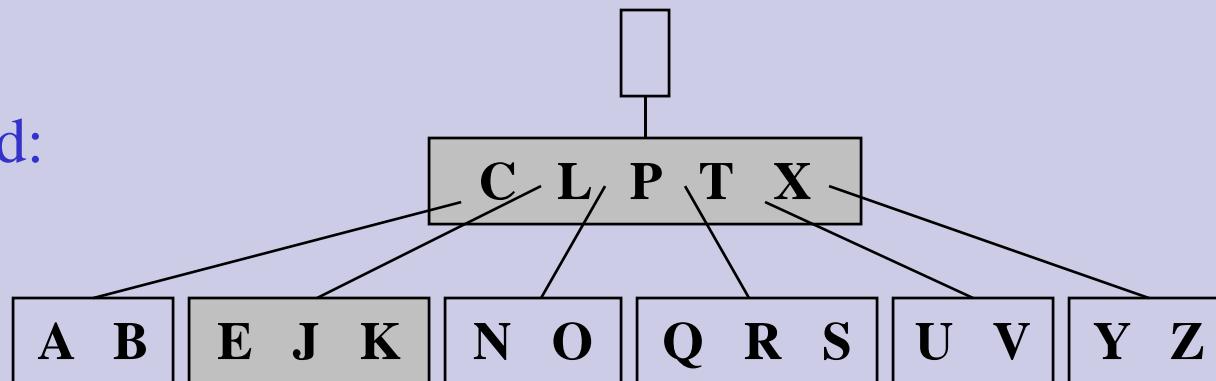
- If $c_i[x]$ and both of $c_i[x]$'s siblings have $t - 1$ keys, **merge** c_i with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node



Deleting Keys – Merging (2)



D deleted:



tree shrinks in height

Deletion: Running Time

- Most of the keys are in the leaf, thus deletion most often occurs there!
- In this case deletion happens in one downward pass to the leaf level of the tree
- Deletion from an internal node might require “backing up” (case 2)
- Disk I/O: $O(h)$, since only $O(1)$ disk operations are produced during recursive calls
- CPU: $O(th) = O(t \log_t n)$

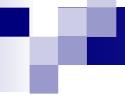
Two-pass Operations

- Simpler, practical versions of algorithms use two passes (down and up the tree):
 - *Down* – Find the node where deletion or insertion should occur
 - *Up* – If needed, split, merge, or distribute; propagate splits, merges, or distributes up the tree
- To avoid reading the same nodes twice, use a buffer of nodes

Case Study: Searching for Patterns

Problem: find all occurrences of pattern P of length m inside the text T of length n .

⇒ *Exact matching problem*



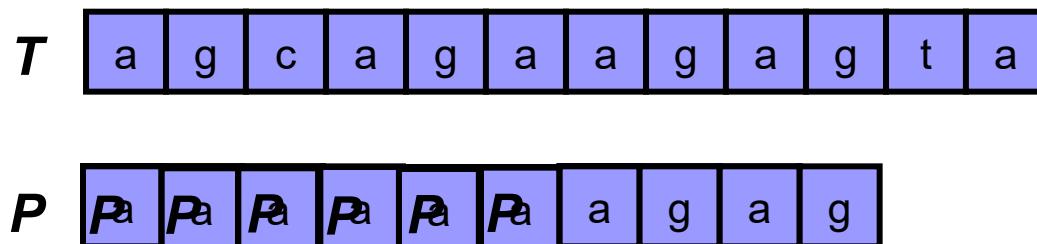
String Matching - Applications

- Text editing
- Term rewriting
- Lexical analysis
- Information retrieval
- And, bioinformatics

Exact matching problem

Given a string P (pattern) and a longer string T (text).
Aim is to find all occurrences of pattern P in text T .

The naive method:



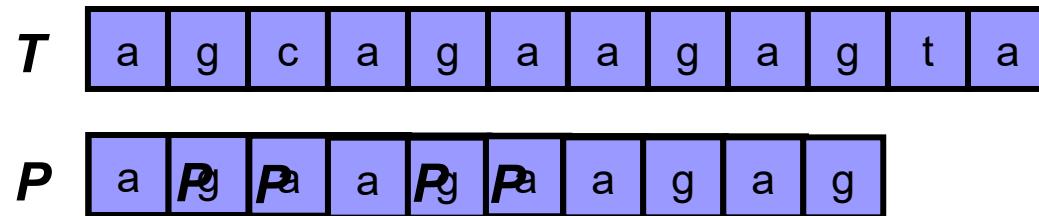
If m is the length of P , and n is the length of T , then

Time complexity = $O(m.n)$,

Space complexity = $O(m + n)$

Can we be more clever ?

- When a mismatch is detected, say at position k in the pattern string, we have already successfully matched $k-1$ characters.
- We try to take advantage of this to decide where to restart matching



The Knuth-Morris-Pratt Algorithm

Observation: when a mismatch occurs, we may not need to restart the comparison all the way back (from the next input position).

What to do:

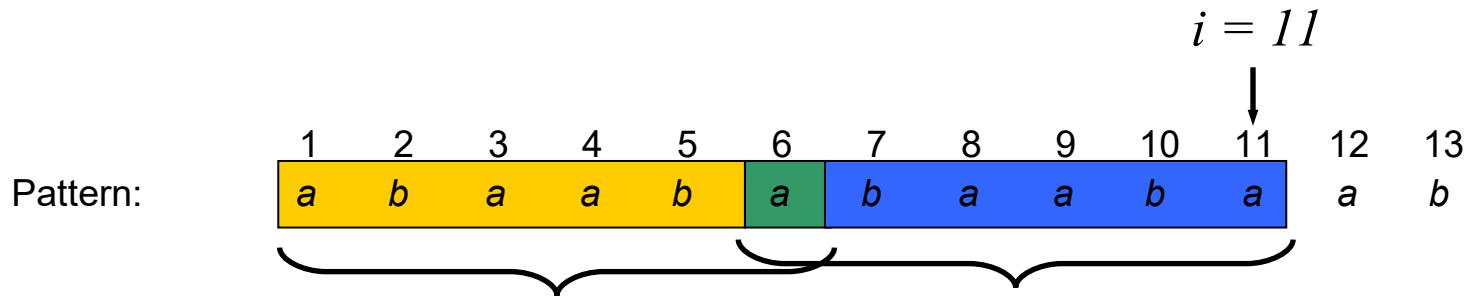
Constructing an array h , that determines how many characters to shift the pattern to the right in case of a mismatch during the pattern-matching process.

KMP (2)

The **key idea** is that if we have successfully matched the prefix $P[1 \dots i-1]$ of the pattern with the substring $T[j-i+1, \dots, j-1]$ of the input string and $P(i) \neq T(j)$, then **we do not need to reprocess any of the suffix $T[j-i+1, \dots, j-1]$** since we know this portion of the text string is the **prefix** of the pattern that we have just matched.

The failure function h

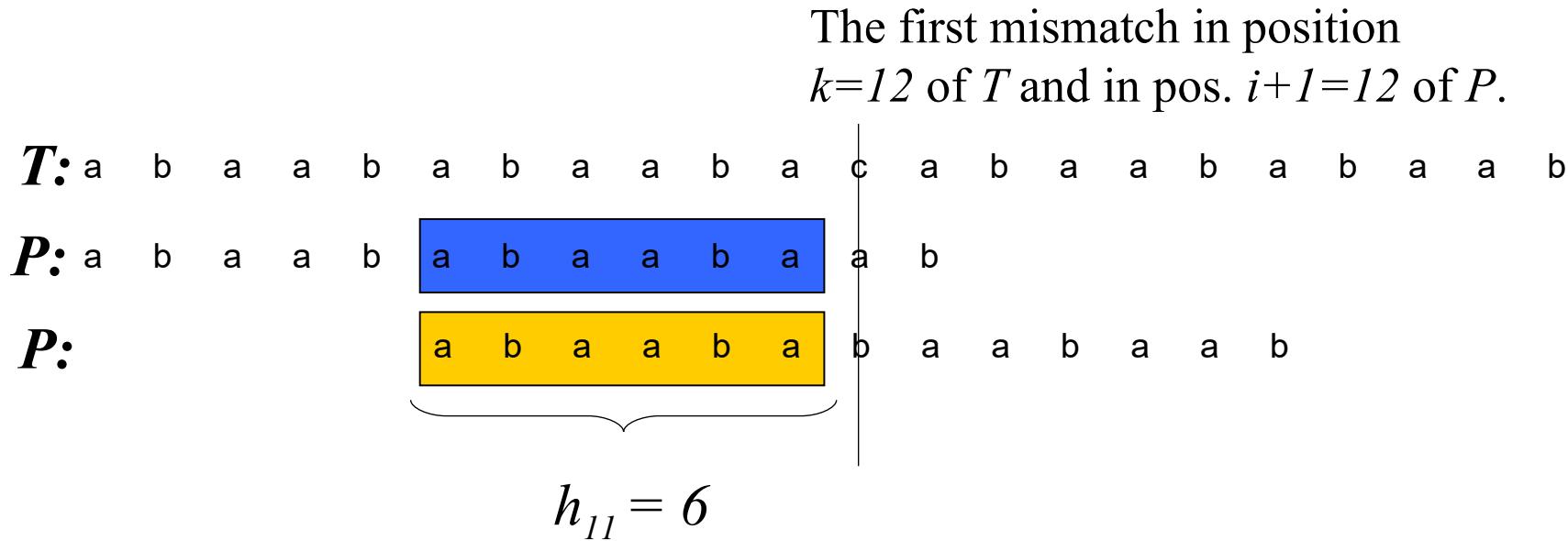
For each position i in pattern P , define h_i to be the length of the longest proper suffix of $P[1, \dots, i]$ that matches a prefix of P .



Hence, $h(11) = 6$.

If there is no proper suffix of $P[1, \dots, i]$ with the property mentioned above, then $h(i) = 0$

The KMP shift rule



Shift P to the right so that $P[1, \dots, h(i)]$ aligns with the suffix
 $T[k - h(i), \dots, k - 1]$.

They must match because of the definition of h .

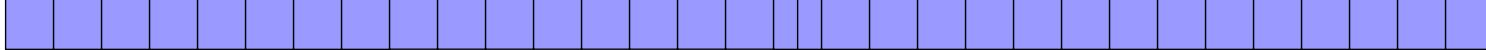
In other words, shift P exactly $i - h(i)$ places to the right.

If there is no mismatch, then shift P by $m - h(m)$ places to the right.

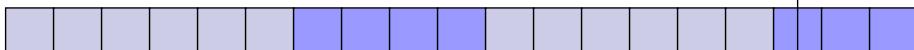
The KMP algorithm finds all occurrences of P in T .

Suppose not, ...

$T:$



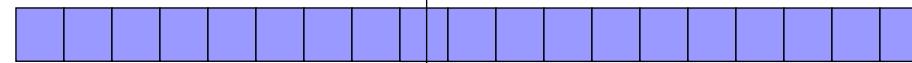
First mismatch in position
 k of T and in pos. $i+1$ of P .



P before shift



P after shift

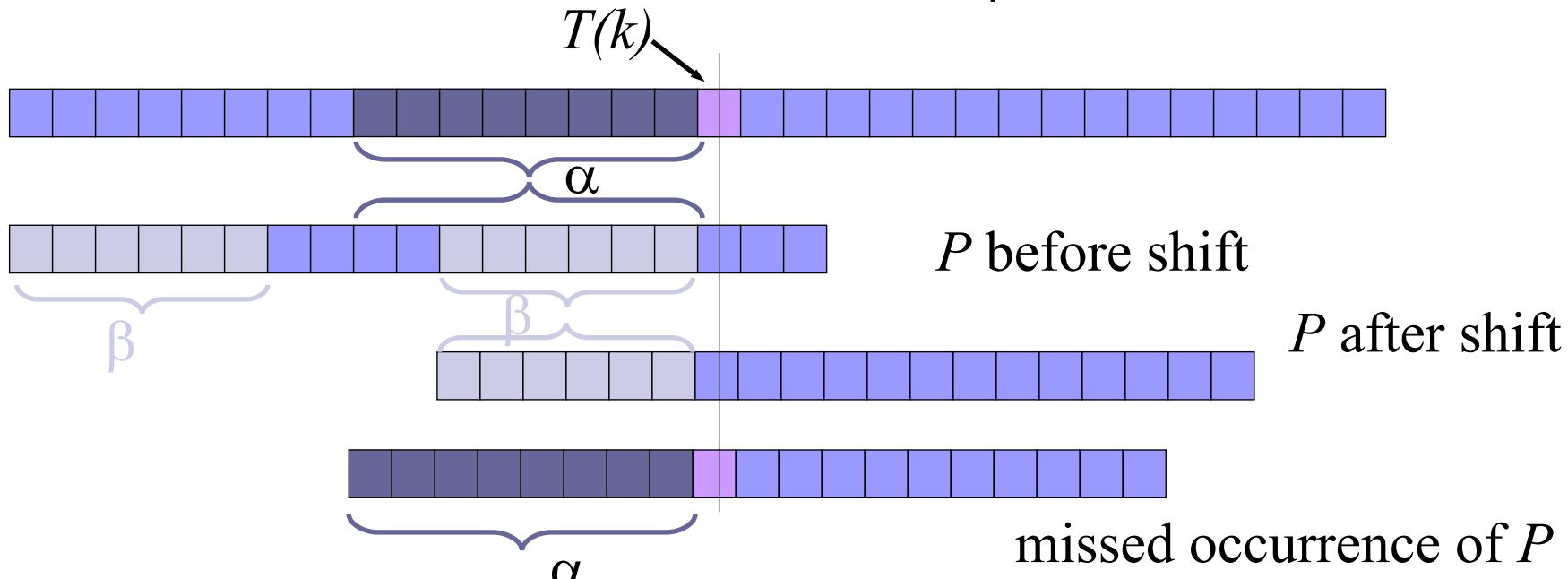


missed occurrence of P

Correctness of KMP.

First mismatch in position
 k of T and in pos. $i+1$ of P .

$T:$



$$|\alpha| > |\beta| = h(i)$$

It is a contradiction.

An Example

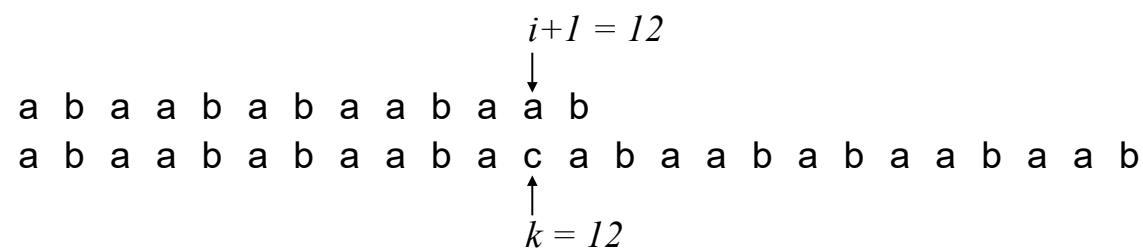
Given:

	1	2	3	4	5	6	7	8	9	10	11	12	13
Pattern:	a	b	a	a	b	a	b	a	a	b	a	a	b
Array h :	0	0	1	1	2	3	2	3	4	5	6	4	5

Input string:

abaababaabacabaababaabaab.

Scenario 1:

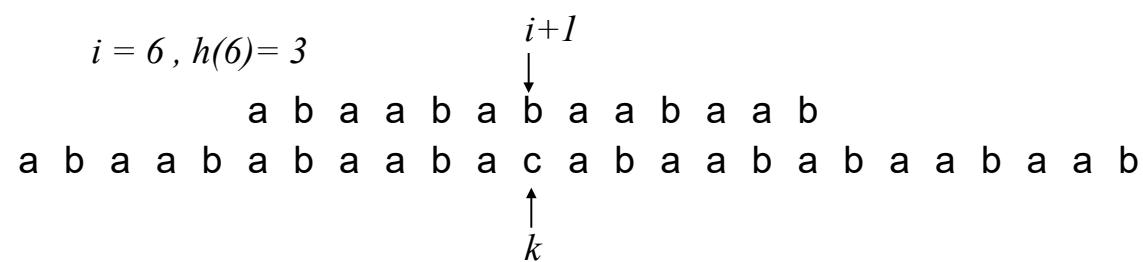


What is $h(i) = h(11) = ?$

$$h(11) = 6 \Rightarrow i - h(i) = 11 - 6 = 5$$

Scenario 2:

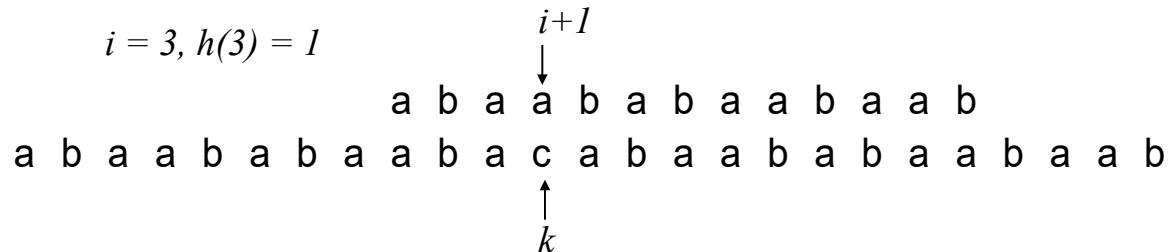
$$i = 6, h(6) = 3$$



An Example

Scenario 3:

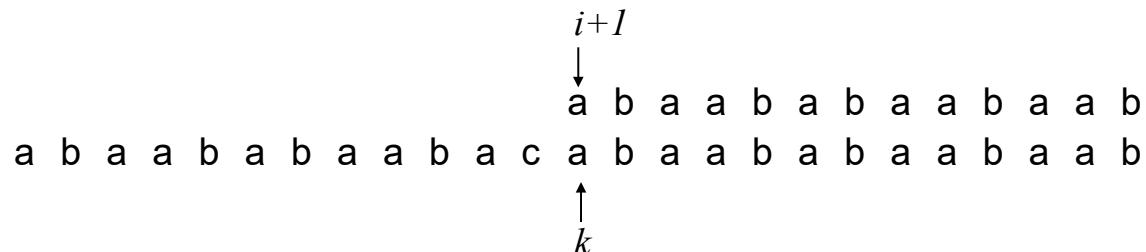
$$i = 3, h(3) = 1$$



Subsequently

$$i = 2, 1, 0$$

Finally, a match is found:

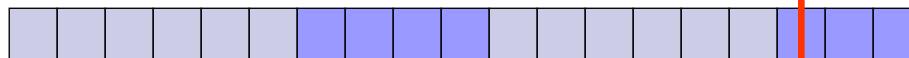
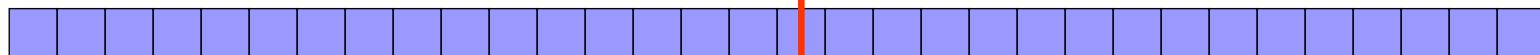


Complexity of KMP

In the KMP algorithm, the number of character comparisons is at most $2n$.

After shift, no character will be compared before **this** position no backtrack

$T:$



P before shift



P after shift

In any shift at most one comparison involves a character of T that was previously compared.

Hence $\#comparisons \leq \#shifts + |T| \leq 2|T| = 2n$.

Computing the failure function

- We can compute $h(i+1)$ if we know $h(1)..h(i)$
- To do this we run the KMP algorithm where the text is the pattern with the first character replaced with a \$.
- Suppose we have successfully matched a prefix of the pattern with a suffix of $T[1..i]$; the length of this match is $h(i)$.
- If the next character of the pattern and text match then $h(i+1)=h(i)+1$.
- If not, then in the KMP algorithm we would shift the pattern; the length of the new match is $h(h(i))$.
- If the next character of the pattern and text match then $h(i+1)=h(h(i))+1$, else we continue to shift the pattern.
- Since the no. of comparisons required by KMP is length of pattern+text, time taken for computing the failure function is $O(n)$.

Computing h: an example

Given:

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

Failure function h :

0 0 1 1 2 3 2 3 4 5 6 4 5

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

Text: \$ b a a b a b a a b a a b

Pattern: a b a a b a **b** h(11)=6

Pattern a b a **a** **b** h(6)=3

$h(12) = 4 = h(6) + 1 = h(h(11)) + 1$

$h(13) = 5 = h(12) + 1$

KMP - Analysis

- The KMP algorithm never needs to backtrack on the text string.

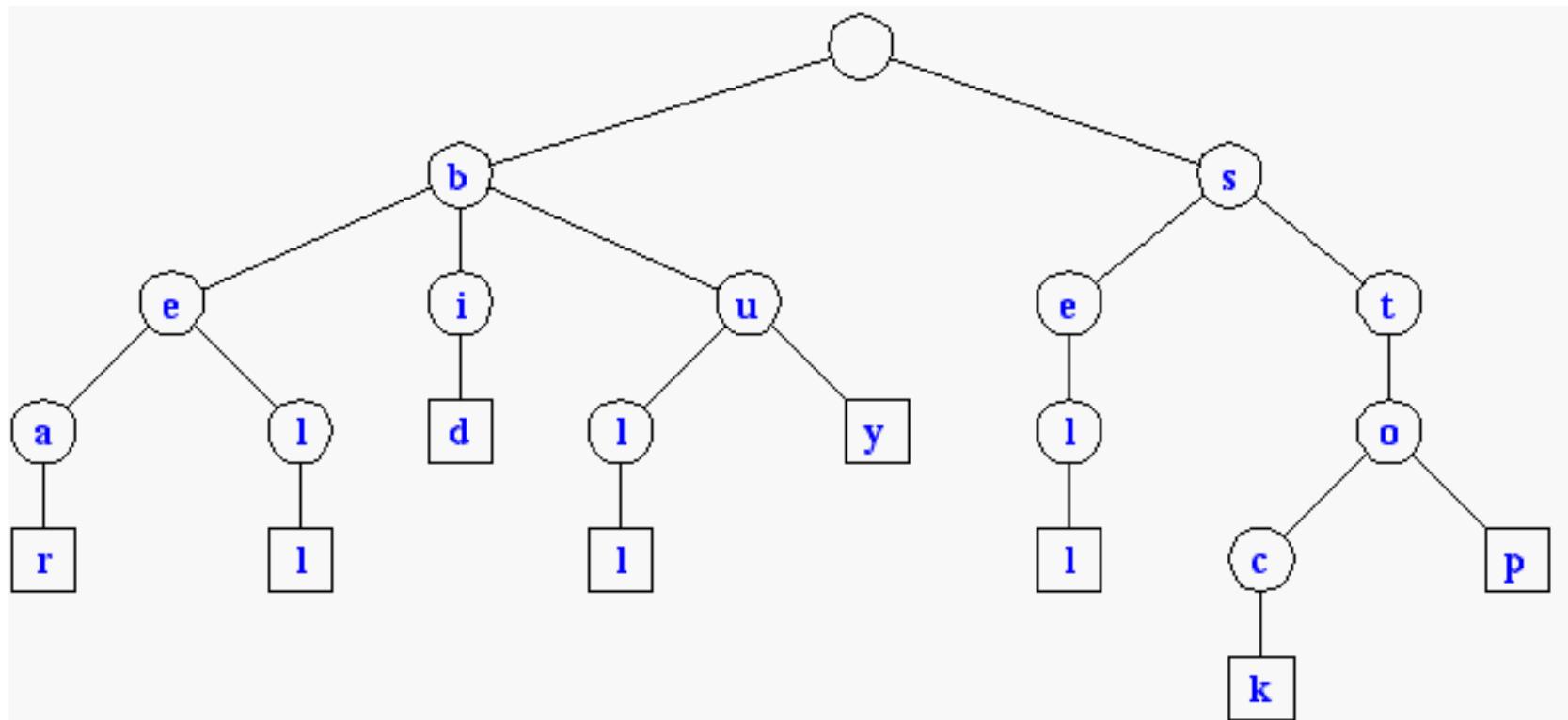
preprocessing *searching*

Time complexity = $O(m + n)$

Space complexity = $O(m + n)$,
where $m = |P|$ and $n = |T|$.

Tries

- Standard Tries
- Compressed Tries
- Suffix Tries

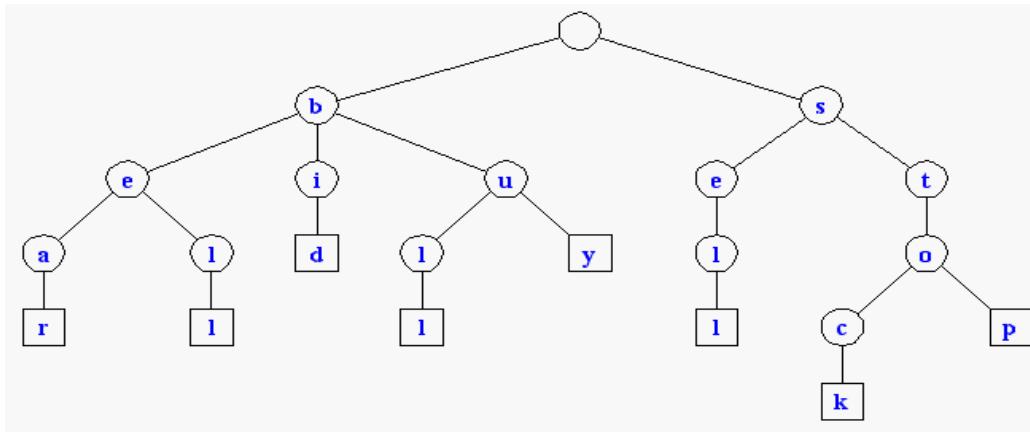


Text Processing

- We have seen that preprocessing the pattern speeds up pattern matching queries
- After preprocessing the pattern in time proportional to the pattern length, the KMP algorithm searches an arbitrary English text in time **proportional to the text length**
- If the text is large, immutable and searched for often (e.g., works by Shakespeare), we may want to preprocess the text instead of the pattern in order to perform pattern matching queries in time **proportional to the pattern length.**

Standard Tries

- The **standard trie** for a set of strings S is an ordered tree such that:
 - each node but the root is labeled with a character
 - the children of a node are alphabetically ordered
 - the paths from the external nodes to the root yield the strings of S
- Eg. S = { bear, bell, bid, bull, buy, sell, stock, stop }



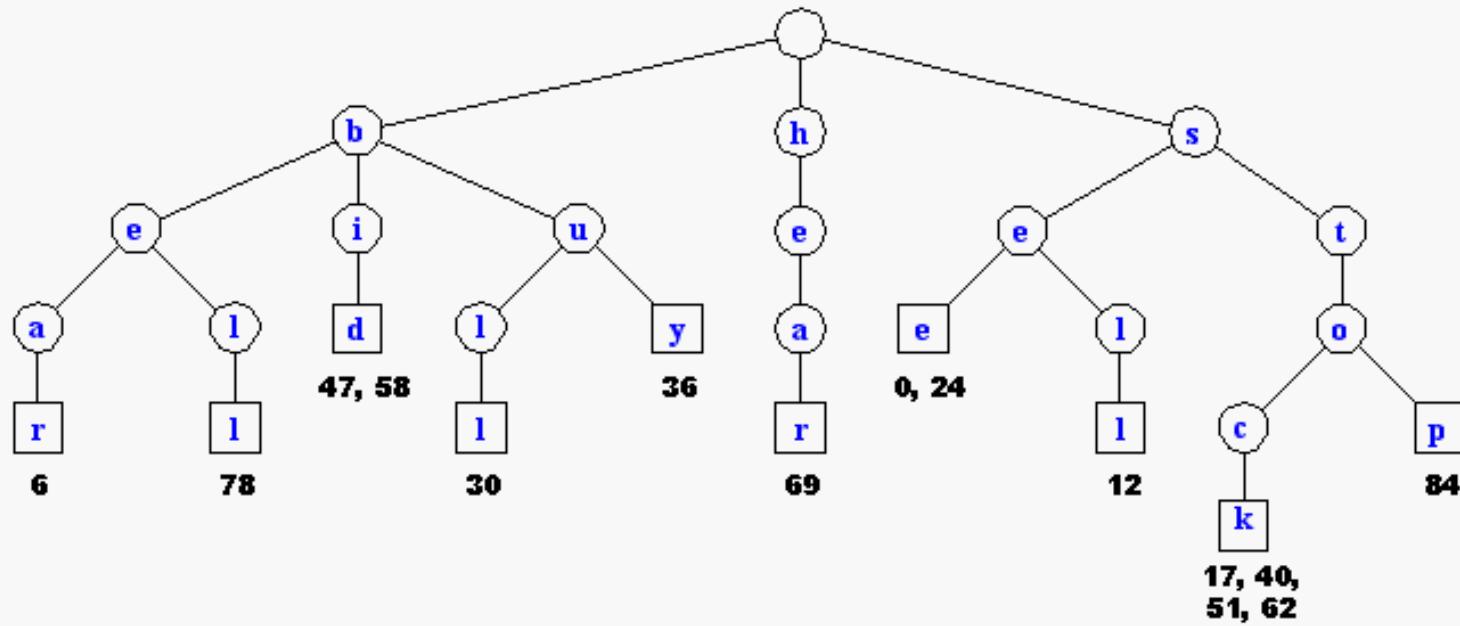
Running time for operations

- A standard trie uses $O(W)$ space.
- Operations (find, insert, remove) take time $O(dm)$ each, where:
 - W = total size of the strings in S ,
 - m = size of the string involved in the operation
 - d = alphabet size,

Applications of Tries

- A standard trie supports the following operations on a preprocessed text in time $O(m)$, where $m = |X|$
 - **word matching**: find the first occurrence of word X in the text
 - **prefix matching**: find the first occurrence of the longest prefix of word X in the text
- Each operation is performed by tracing a path in the trie starting at the root

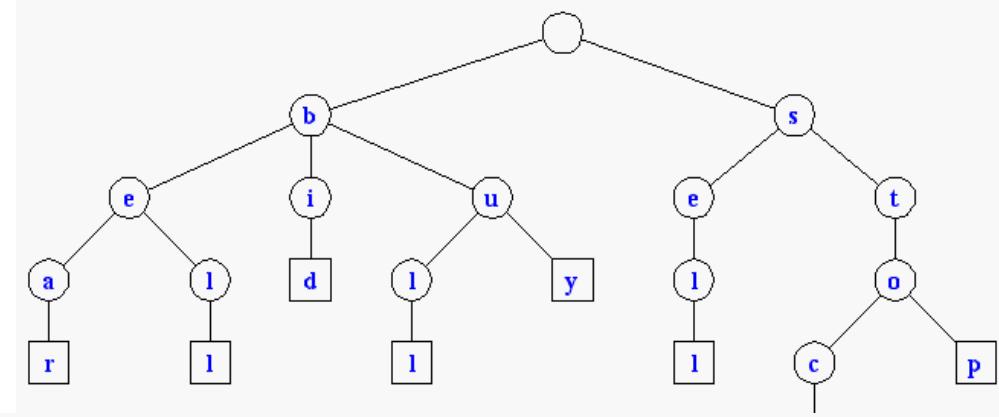
s	e	e	a	b	e	a	r	?	s	e	l	l	s	t	o	c	k	!					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
s	e	e	a	b	u	l	l	?	b	u	y		s	t	o	c	k	!					
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	
b	i	d		s	t	o	c	k	!	b	i	d		s	t	o	c	k	!				
47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68		
h	e	a	r		t	h	e	b	e	l	l	?		s	t	o	p	!					
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88				



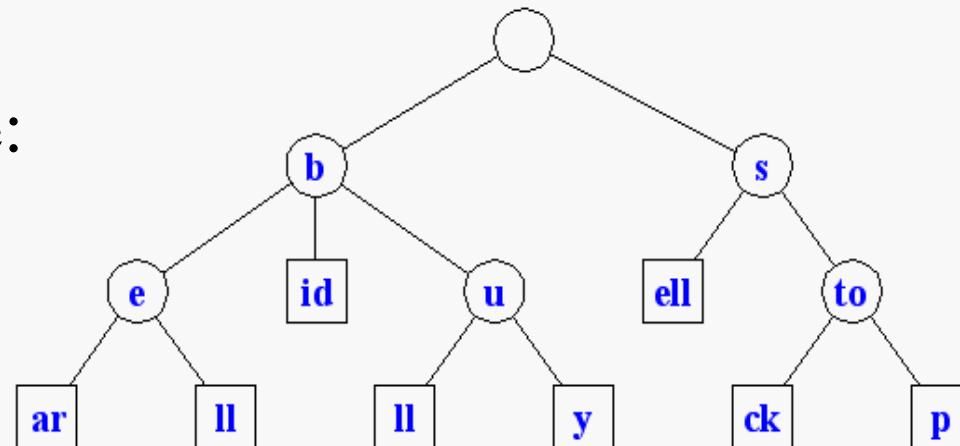
Compressed Tries

- Trie with nodes of degree at least 2
- Obtained from standard trie by compressing chains of redundant nodes.

Standard Trie:



Compressed Trie:



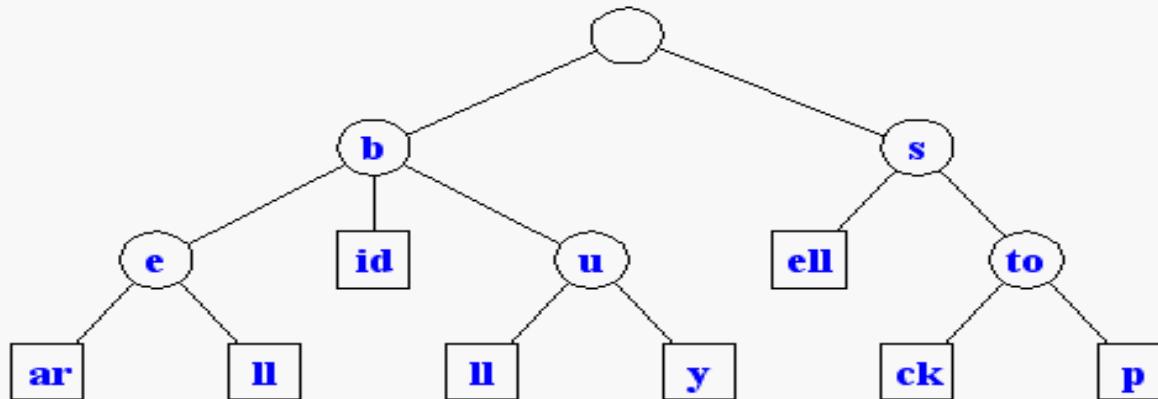
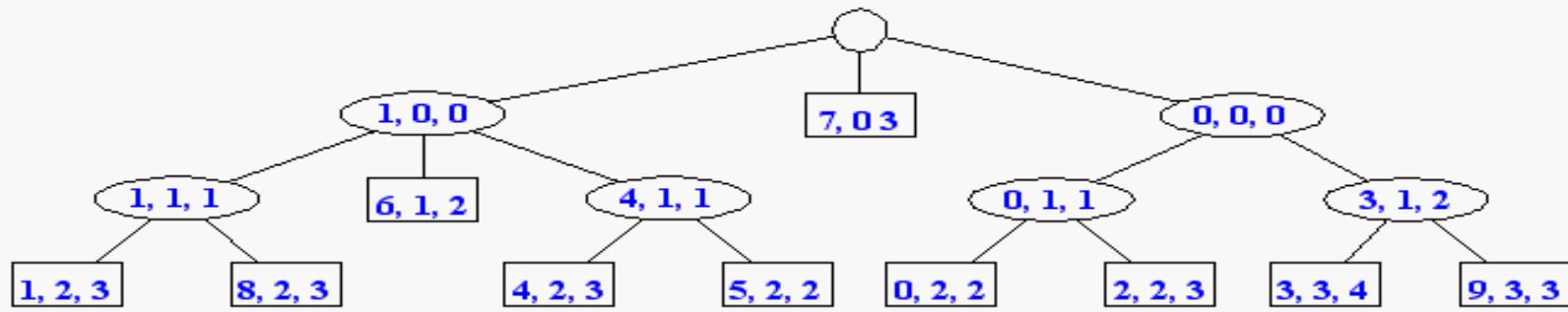
Why Compressed Tries ?

- A tree in which every node has at least 2 children has at most $L-1$ internal nodes, where L is the number of leaves.
- The number of nodes in a compressed trie is $O(s)$, where $s = |S|$.
- The label in each node can be stored by using $O(1)$ space **index ranges** at the nodes

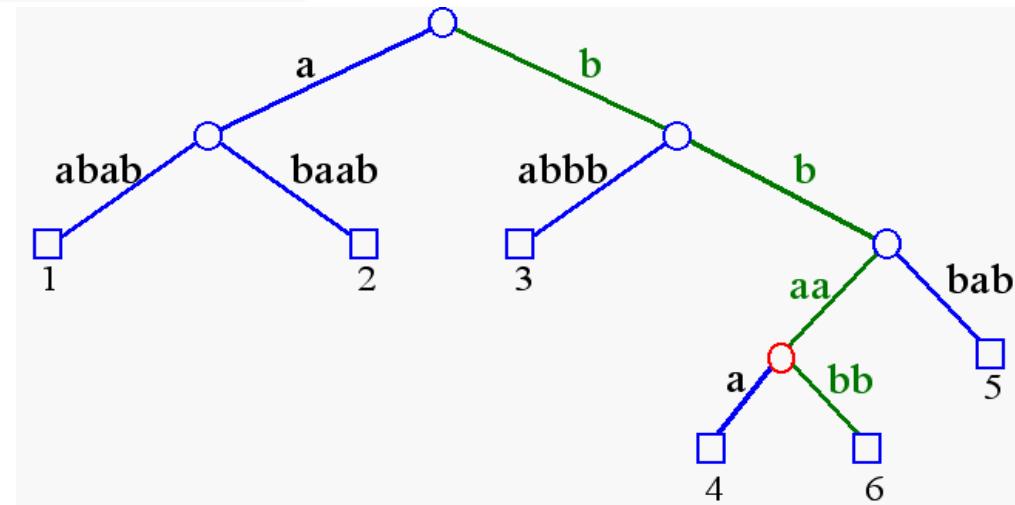
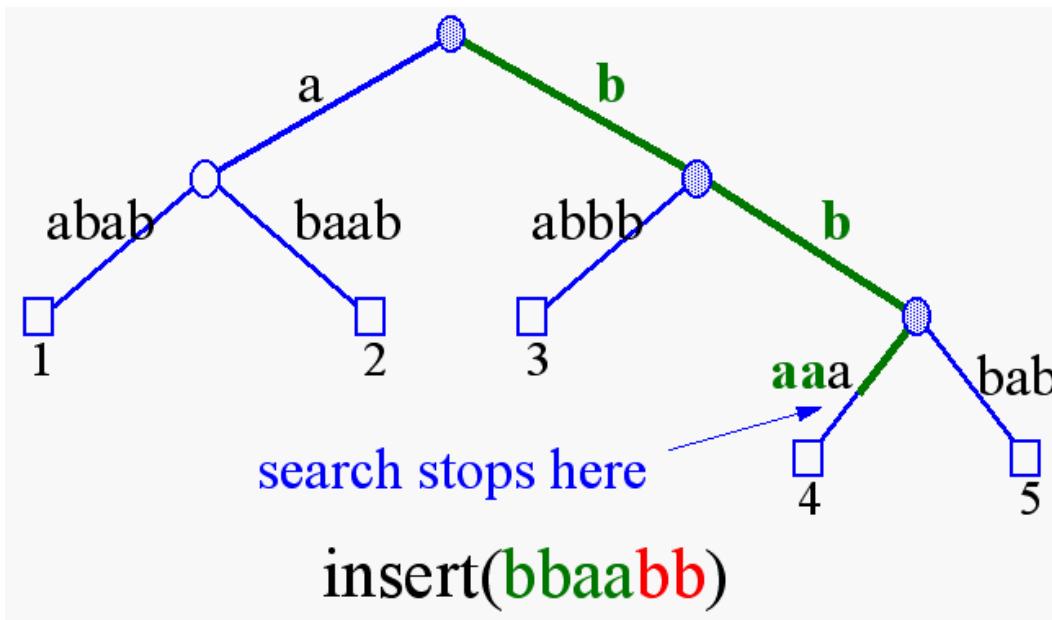
	0	1	2	3	4
$S[0]$ =	s	e	e		
$S[1]$ =	b	e	a	r	
$S[2]$ =	s	e	l	l	
$S[3]$ =	s	t	o	c	k

	0	1	2	3
$S[4]$ =	b	u	l	l
$S[5]$ =	b	u	y	
$S[6]$ =	b	i	d	

	0	1	2	3
$S[7]$ =	h	e	a	r
$S[8]$ =	b	e	l	l
$S[9]$ =	s	t	o	p



Insertion/Deletion in Compressed Tries



Tries and Web Search Engines

- The **index of a search engine** (collection of all searchable words) is stored into a compressed trie
- Each leaf of the trie is associated with a word and has a list of pages (URLs) containing that word, called **occurrence list**
- The trie is kept in internal memory
- The occurrence lists are kept in external memory and are ranked by relevance
- Boolean queries for sets of words (e.g., Java and coffee) correspond to set operations (e.g., intersection) on the occurrence lists
- Additional **information retrieval** techniques are used, such as
 - stopword elimination (e.g., ignore “the” “a” “is”)
 - stemming (e.g., identify “add” “adding” “added”)
 - link analysis (recognize authoritative pages)

Tries and Internet Routers

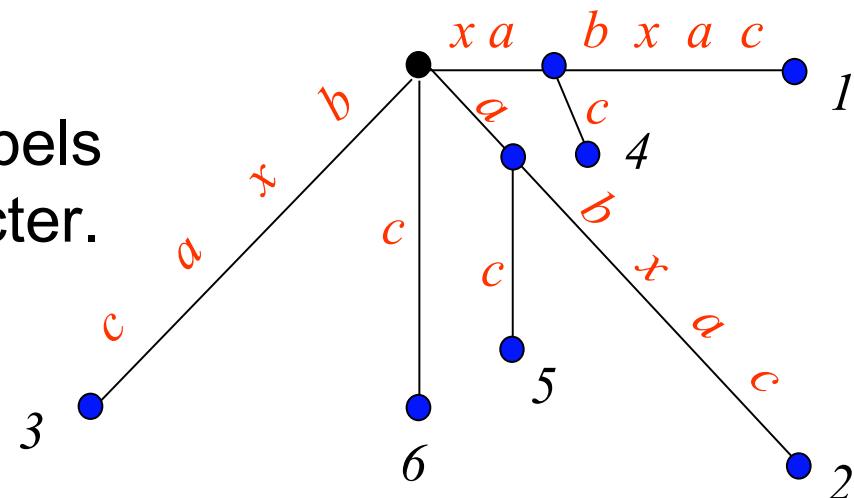
- Computers on the internet (hosts) are identified by a unique 32-bit IP (**internet protocol**) address, usually written in “dotted-quad-decimal” notation
- E.g., www.google.com is 64.233.189.104
- Use nslookup on Unix to find out IP addresses
- An organization uses a subset of IP addresses with the same prefix, e.g., IITD uses 10.*.*.*
- Data is sent to a host by fragmenting it into packets. Each packet carries the IP address of its destination.
- A router forwards packets to its neighbors using **IP prefix matching** rules.
- Routers use tries on the alphabet 0,1 to do prefix matching.

Back to Pattern Matching

- Instead of preprocessing the pattern P , preprocess the text T !
- Use a tree structure where all suffixes of the text are represented;
- Search for the pattern by looking for substrings of the text;
- You can easily test whether P is a substring of T because any substring of T is the prefix of some suffix of T .

Suffix Tree

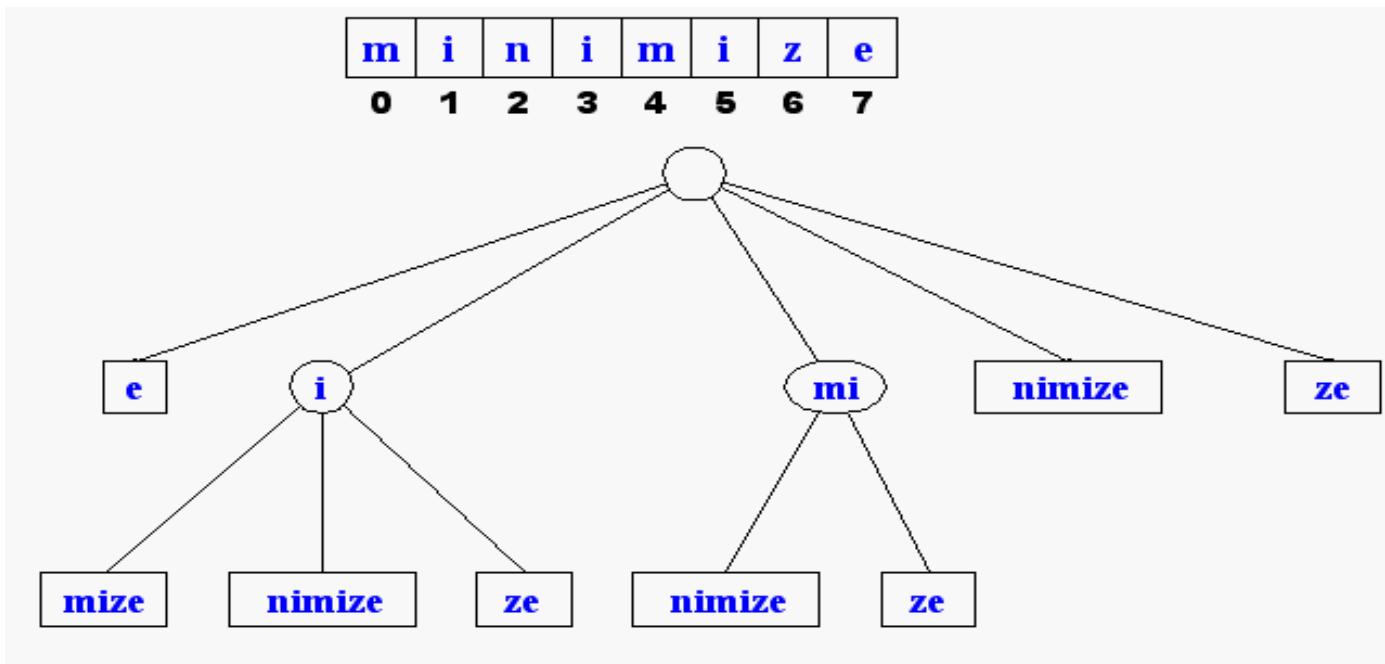
- A suffix tree T for string S is a rooted directed tree whose edges are labeled with nonempty substrings of S .
- Each leaf corresponds to a suffix of S in the sense that the concatenation of the edge-labels on the unique path from the root to the leaf spells out this suffix.
- Each internal node, other than the root, has at least two children.
- No two out-edges of a node can have edge-labels with the same first character.



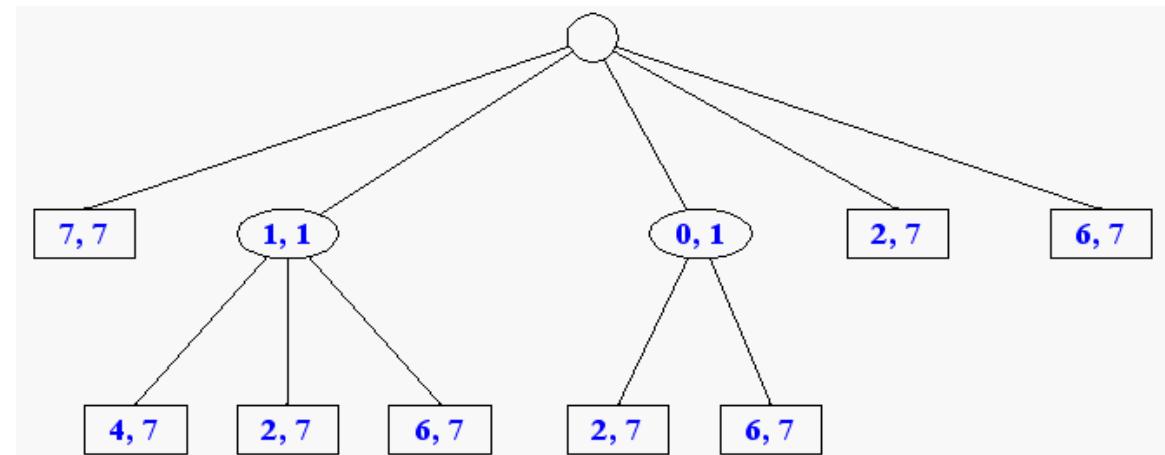
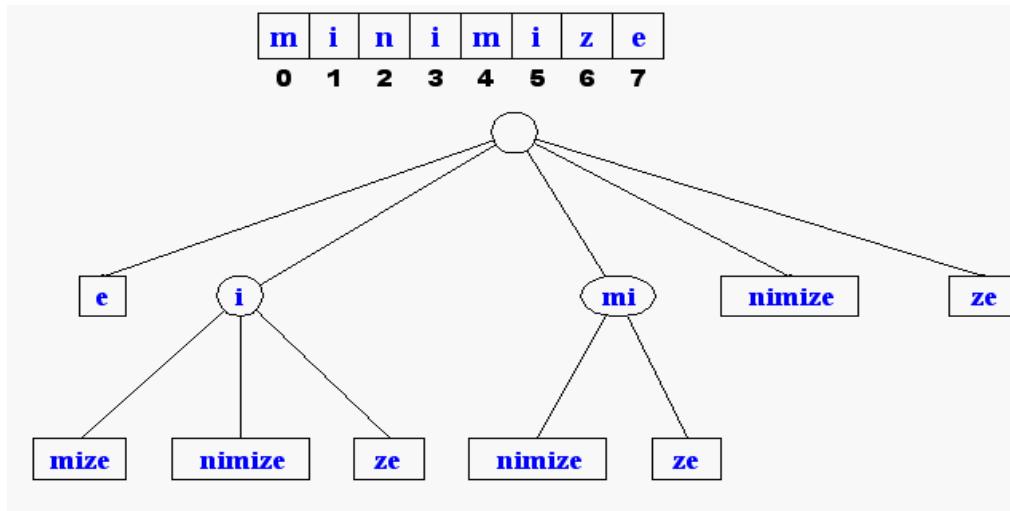
Suffix tree for string $\textcolor{red}{xabxac}$.

Suffix Tree (2)

- A **suffix tree** is essentially a compressed trie for all the suffixes of a text
- The **suffix tree** for a text X of size n from an alphabet of size d stores all the n suffixes of X in $O(n)$ space.

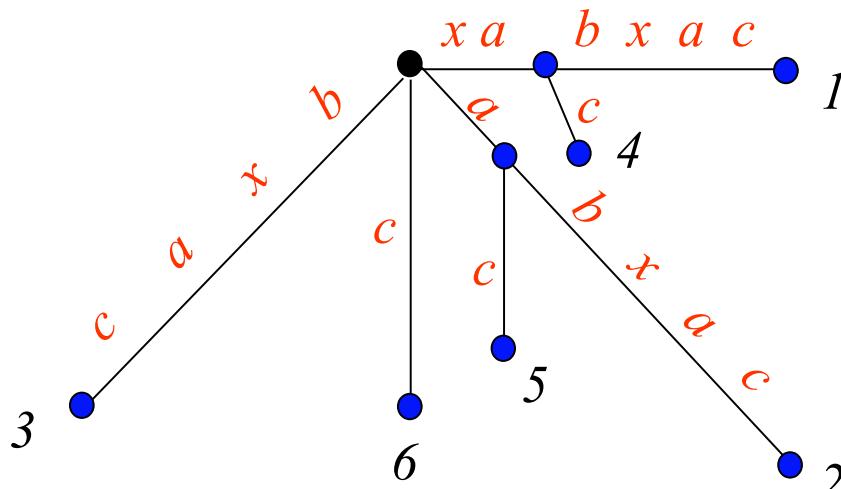


Compact Representation



Note on Suffix Tree

If two suffixes have a same prefix, then their corresponding paths are the same at their beginning, and the concatenation of the edge labels of the mutual part is the prefix.



Suffix tree for string **xabxac**.

For example, suffix *xabxac* and suffix *xac* have the same prefix, *xa*.

Note on Suffix Tree

- Not all strings guaranteed to have corresponding suffix trees
- For example:

consider *xabxa*: it does not have a suffix tree, because here suffix *xa* is also a prefix of another suffix, *xabxa*.
(The path spelling out *xa* would not end at a leaf.)
- How to fix the problem: add \$ - a special “termination” character to the alphabet.

Building a Suffix Tree

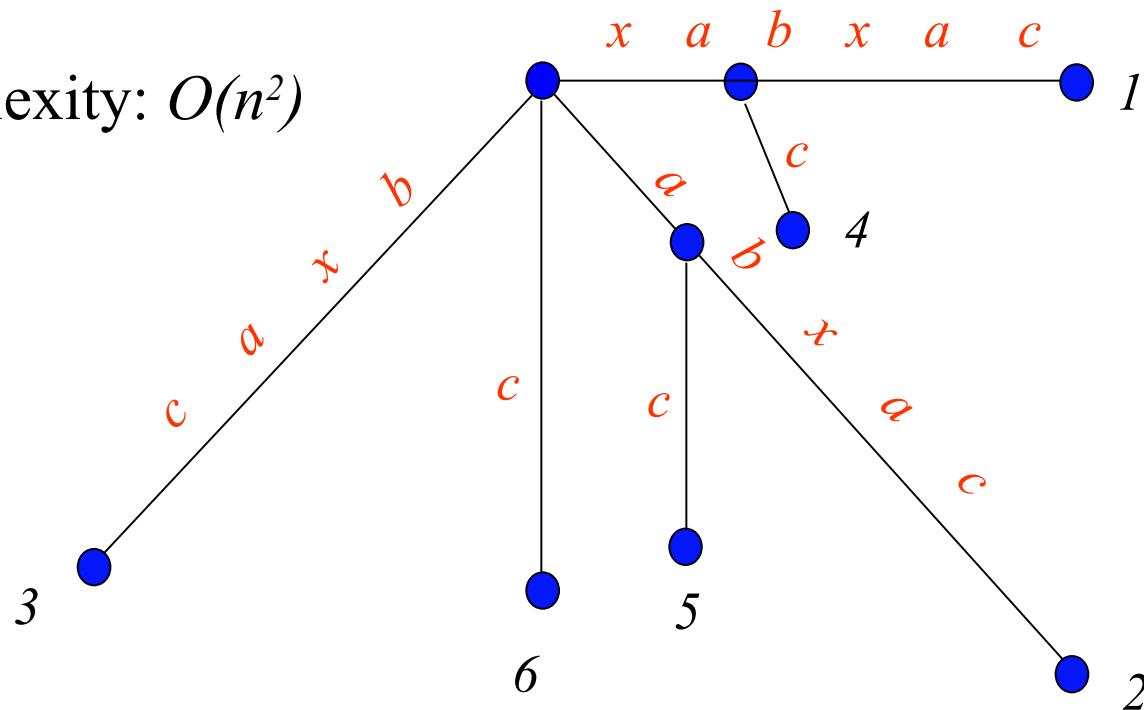
The method starts from a single edge for suffix $S[1..n]$, then it successively enters the edges for suffix $S[i..n]$ into the growing tree, for i increasing from 2 to n .

The general step of the algorithm

- Enter the edge for suffix $S[i..n]$ into the tree as follows.
- Starting at the root find the longest path from the root whose label matches a prefix of $S[i..n]$. At some point, no further matches are possible.
 - If this point is at a node, then denote this node by w .
 - If it is in the middle of an edge, then insert a new node, called w , at this point.
- Create a new edge running from w to a new leaf labeled $S[i..n]$.

Example

Time complexity: $O(n^2)$



Building suffix tree for string **xabxac**.

Suffix Trees in Pattern Matching

- Given a string P (pattern) and a longer string T (text). Our aim is to find all occurrences of pattern P in text T .
- The idea of the algorithm is that every occurrence of P in T is a prefix of a suffix of T ,
- Thus, an occurrence of P in T can be obtained as the concatenation of the labels of edges along the path beginning at the root.

Suffix Trees in Pattern Matching

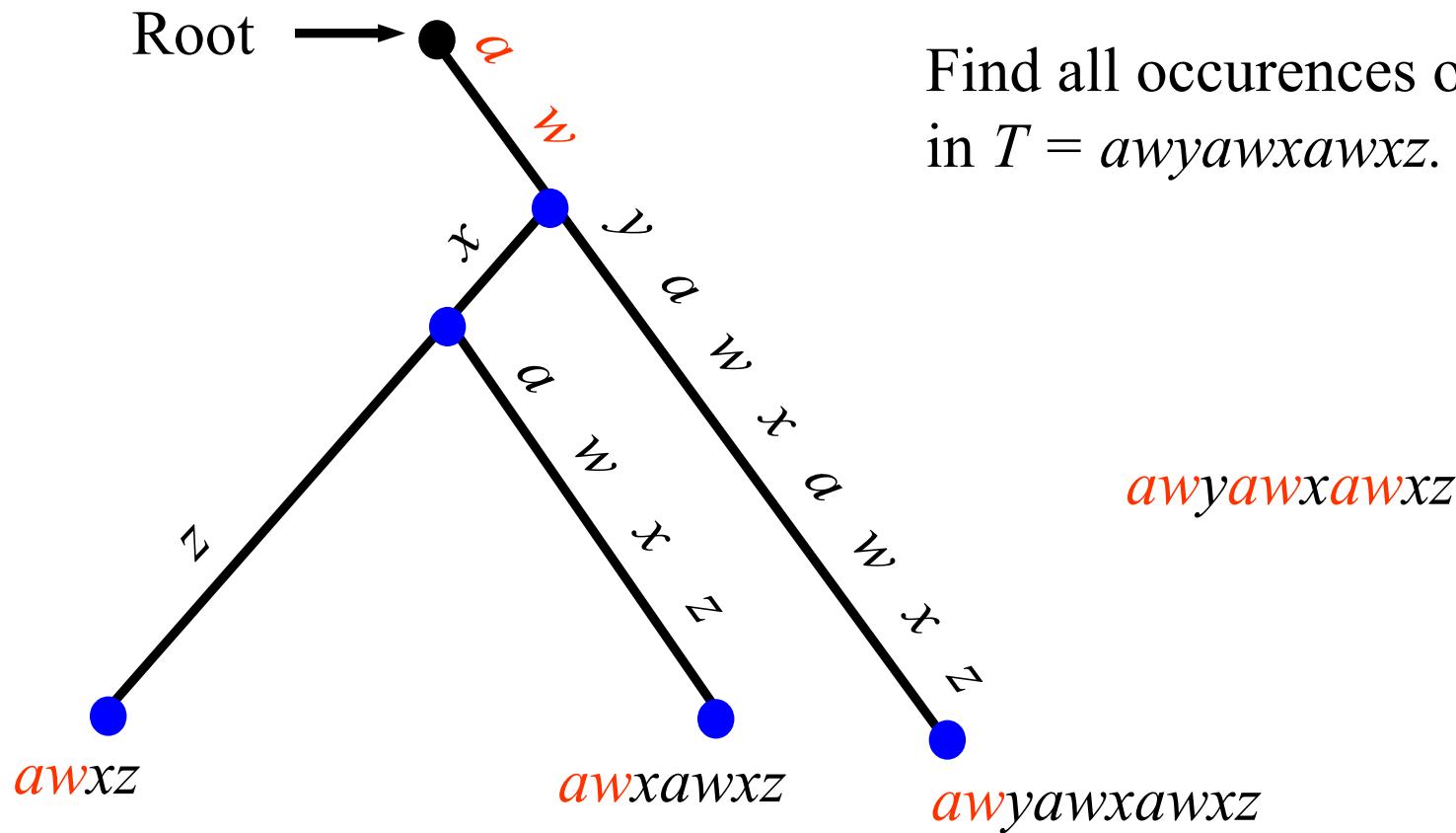
- Build a suffix tree \mathcal{T} for text T .
- Match the characters of P along the **unique** path in \mathcal{T} beginning at the root until
 - P is exhausted or
 - no more matches are possible

In case 2, P does not appear anywhere in T .

In case 1, P is a prefix of a suffix obtained by extending the path until we reach a leaf.

Each extension gives a suffix a prefix of which is P , thus, each extension provides an occurrence of P in T .

Example



Constructing a Suffix Trees

- A suffix tree can be constructed in linear time

[Weiner73, McCreight76, Ukkonen95]

Complexity of Pattern matching

Time complexity

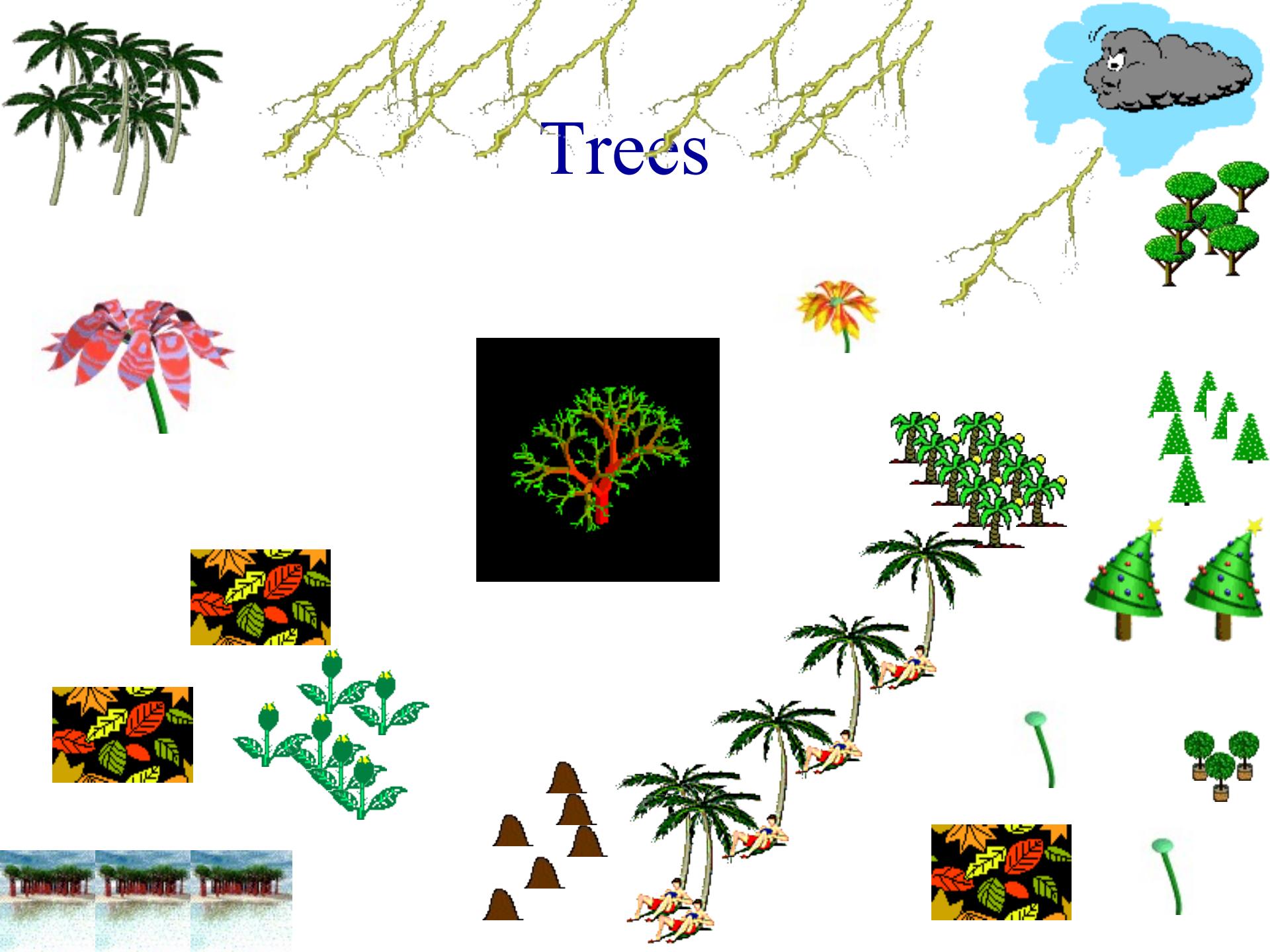
Preprocessing : $O(|T|)$

Searching : $O(|P| + k)$,

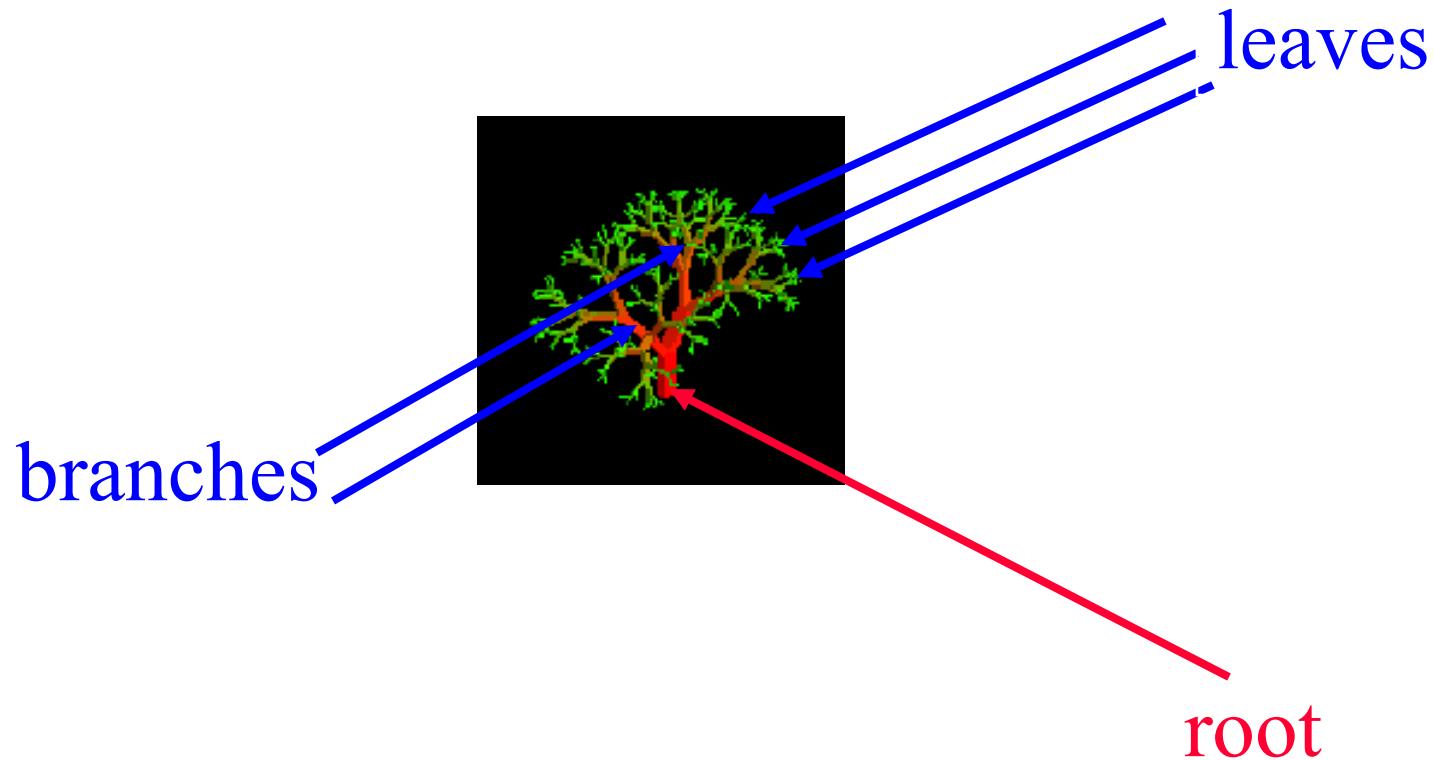
where k is # occurrences of P in T

Space complexity

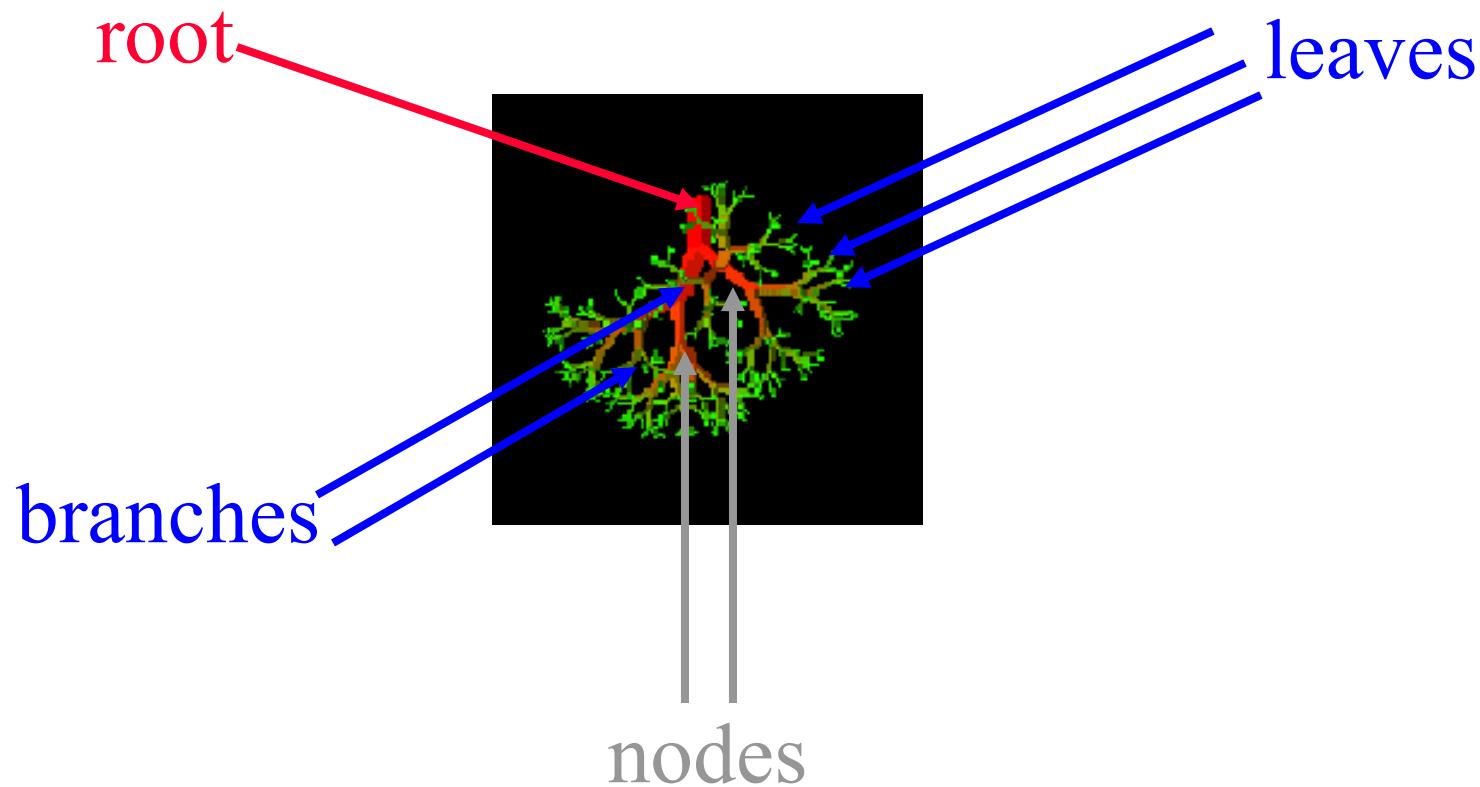
$O(|P| + |T|)$



Nature Lover's View Of A Tree



Computer Scientist's View





Linear Lists And Trees



- Linear lists are useful for serially ordered data.
 - $(e_0, e_1, e_2, \dots, e_{n-1})$
 - Days of week.
 - Months in a year.
 - Students in this class.
- Trees are useful for hierarchically ordered data.
 - Employees of a corporation.
 - President, vice presidents, managers, and so on.
 - Java's classes.
 - Object is at the top of the hierarchy.
 - Subclasses of Object are next, and so on.

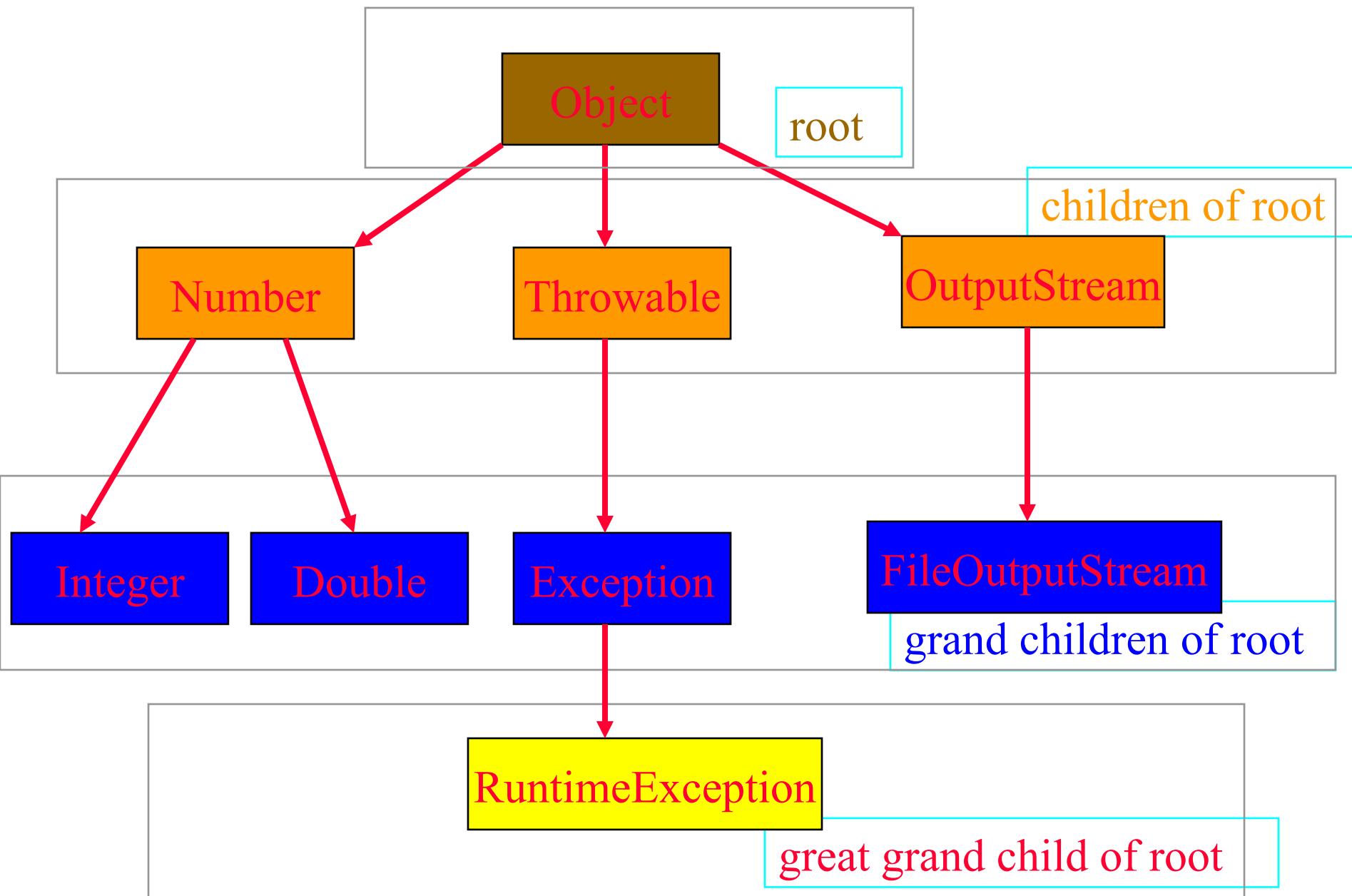


Hierarchical Data And Trees



- The element at the top of the hierarchy is the **root**.
- Elements next in the hierarchy are the **children** of the root.
- Elements next in the hierarchy are the **grandchildren** of the root, and so on.
- Elements that have no children are **leaves**.

Java's Classes (Part Of Figure 1.1)





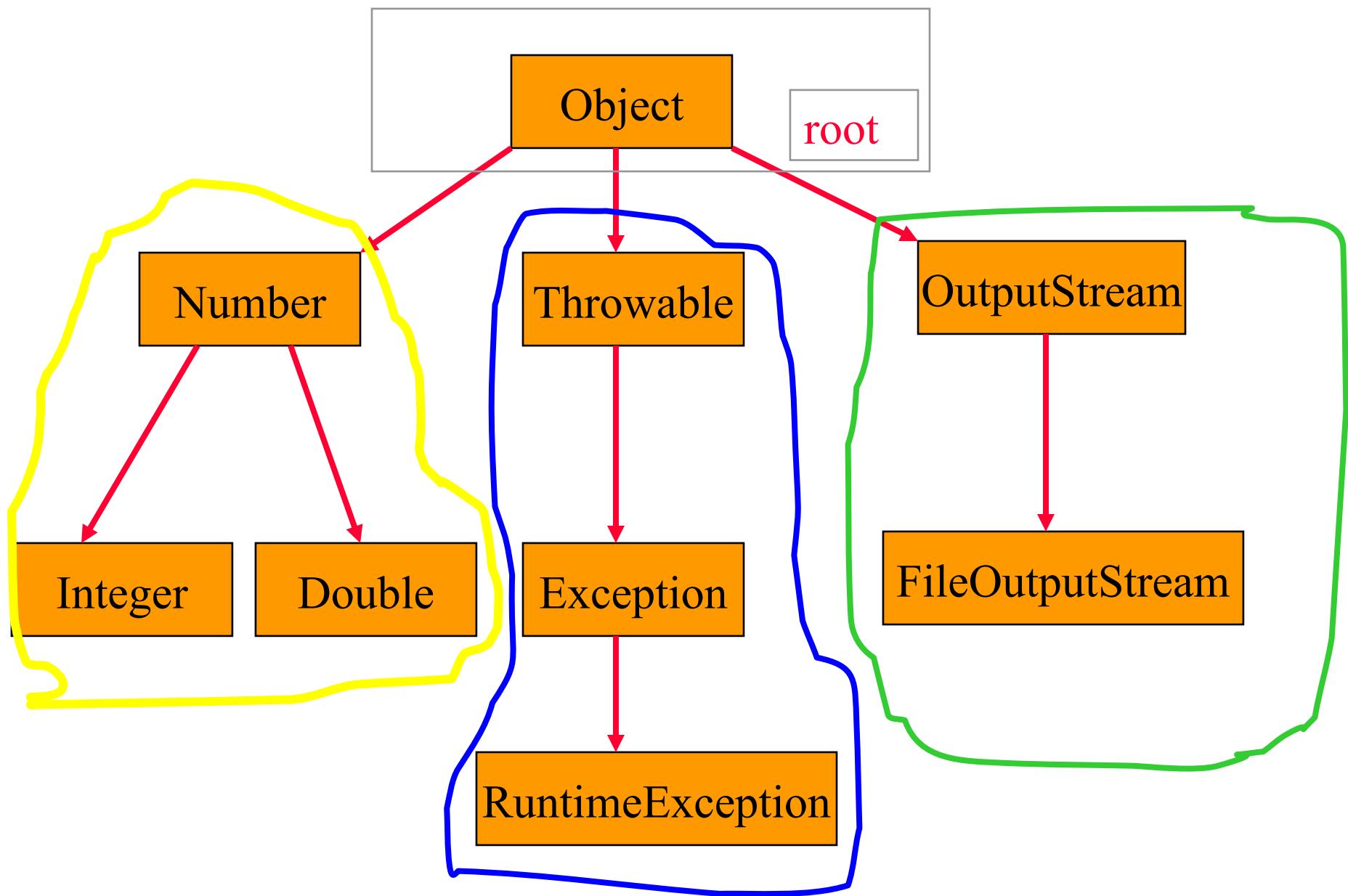
Definition



- A tree t is a finite nonempty set of elements.
- One of these elements is called the root.
- The remaining elements, if any, are partitioned into trees, which are called the subtrees of t .

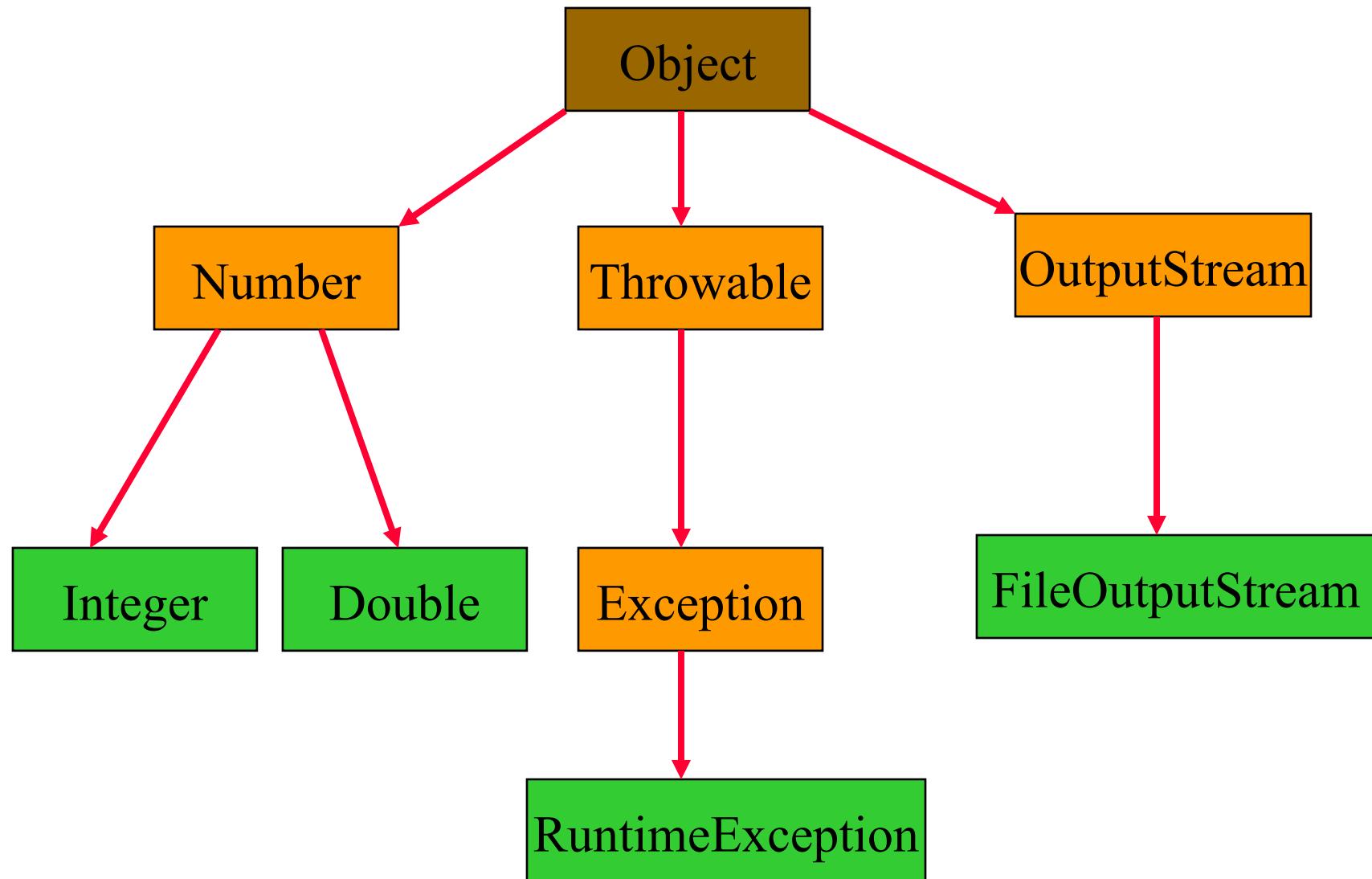


Subtrees

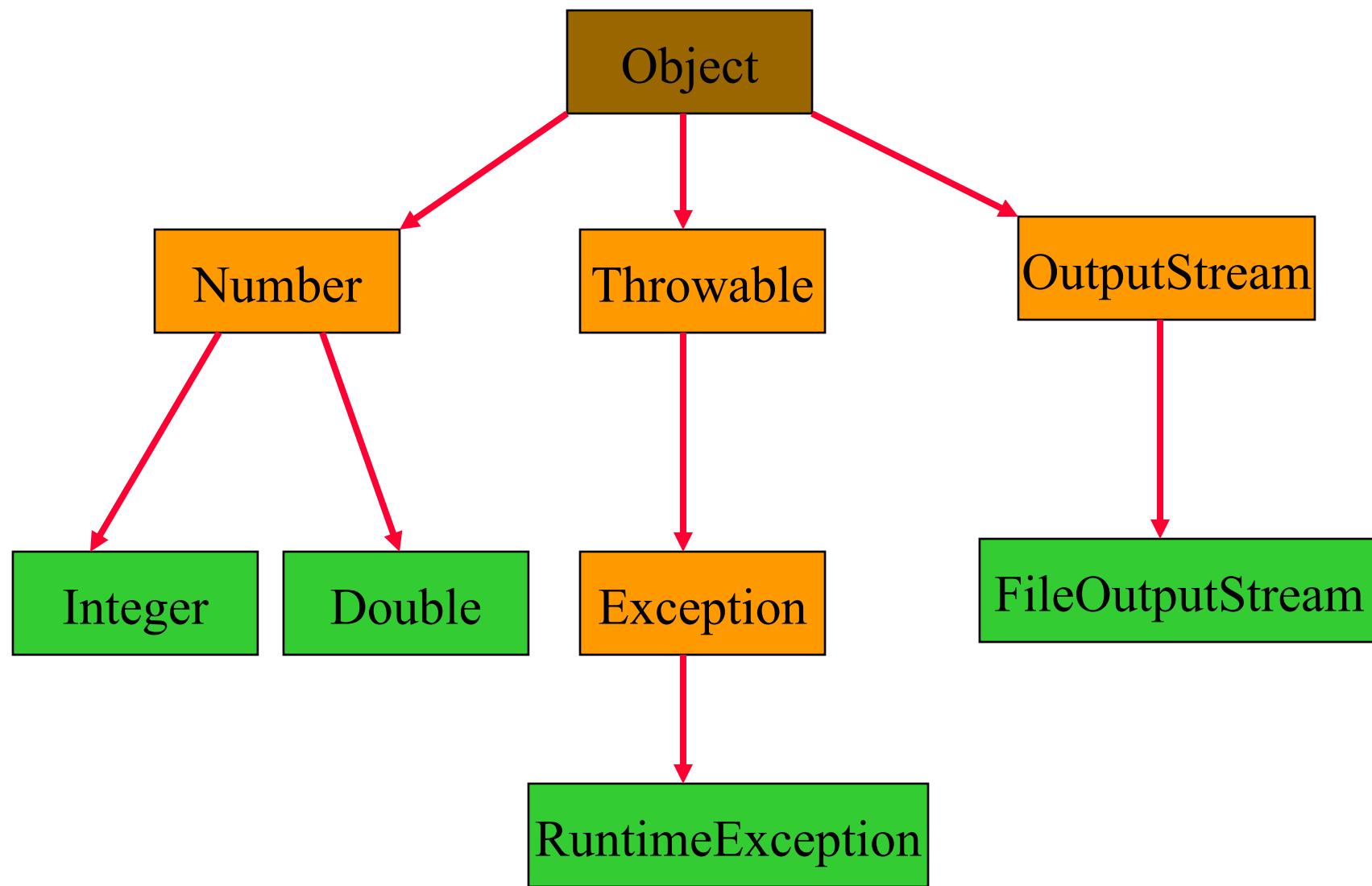




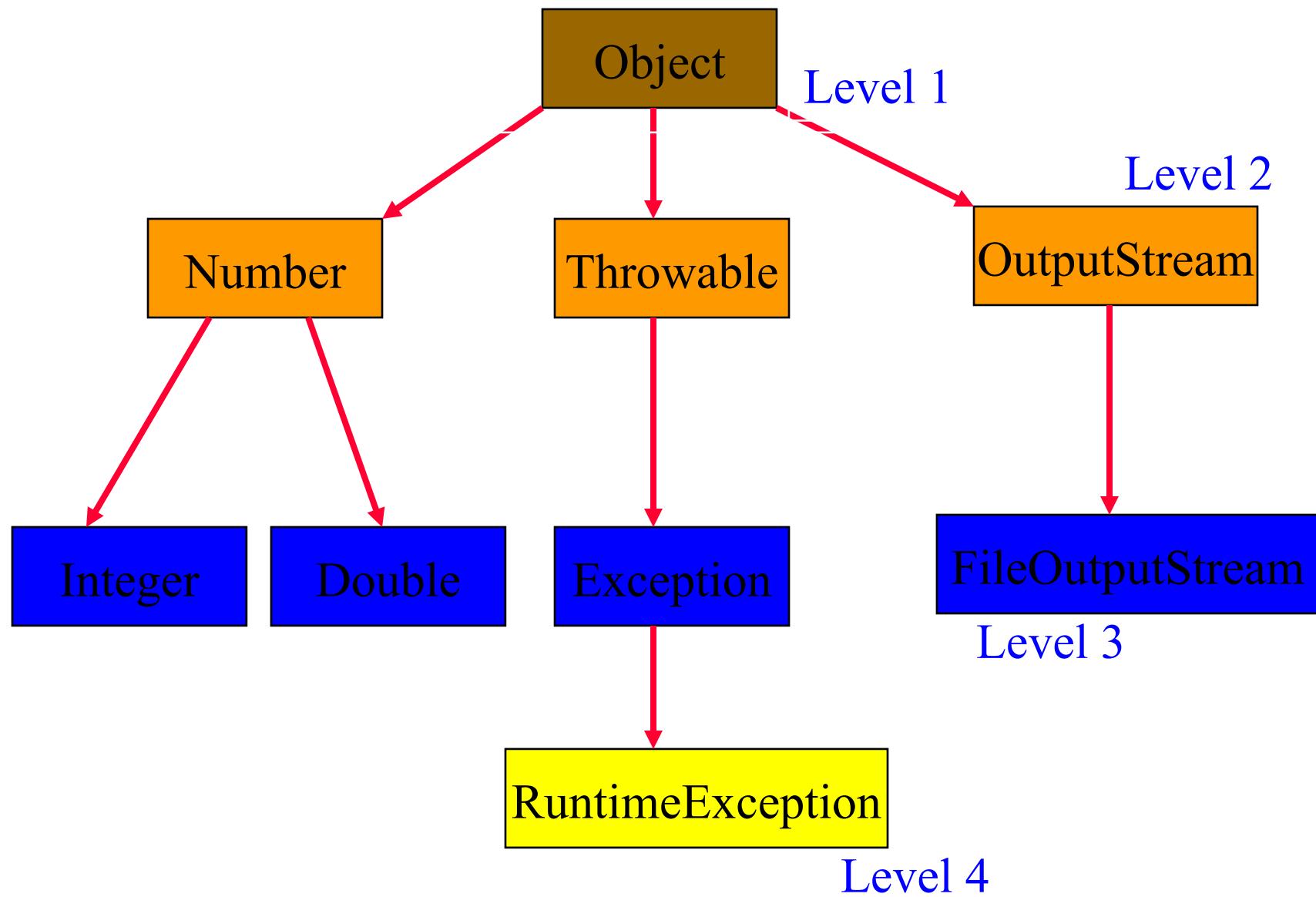
Leaves



Parent, Grandparent, Siblings, Ancestors, Descendants



Levels



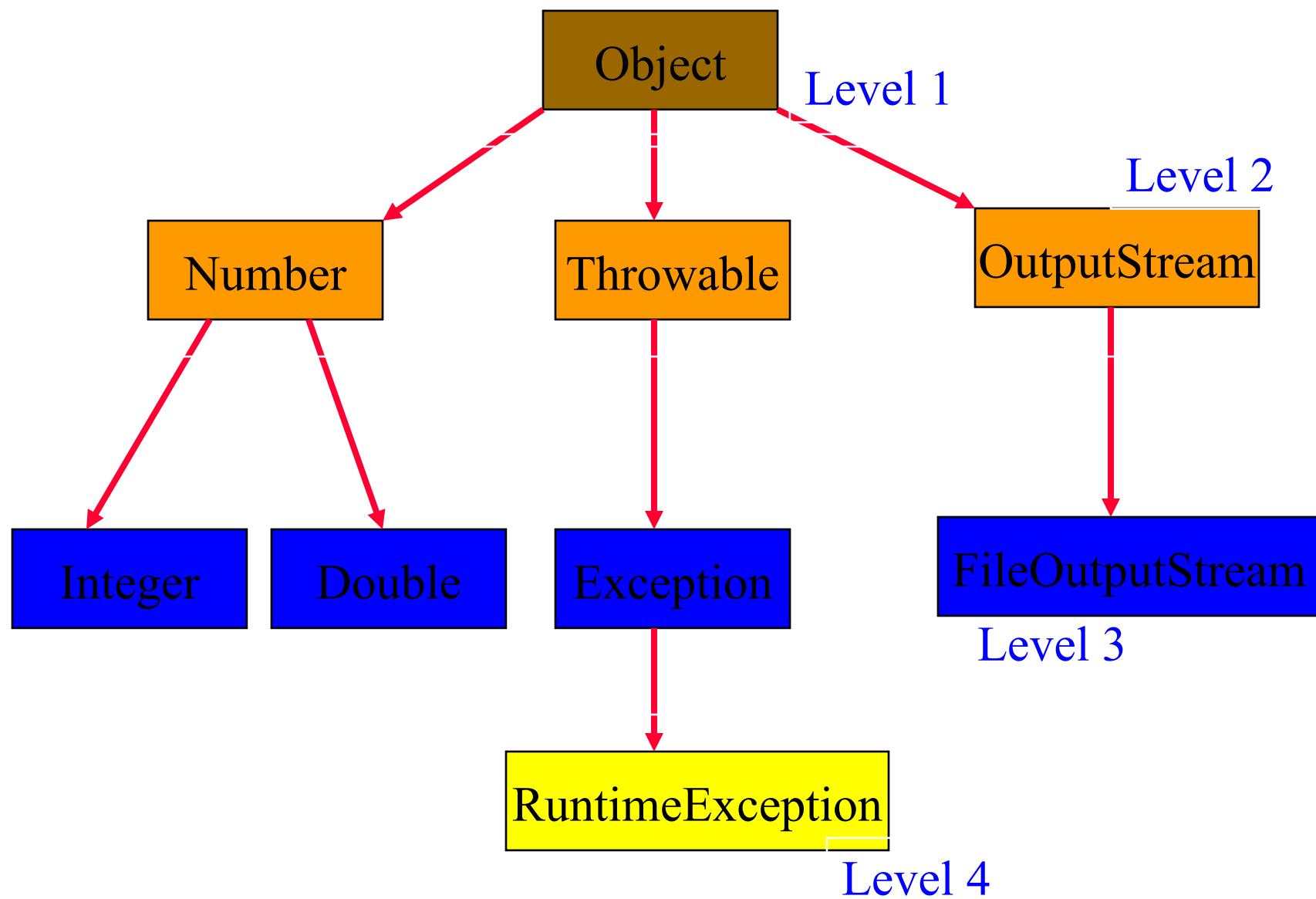


Caution

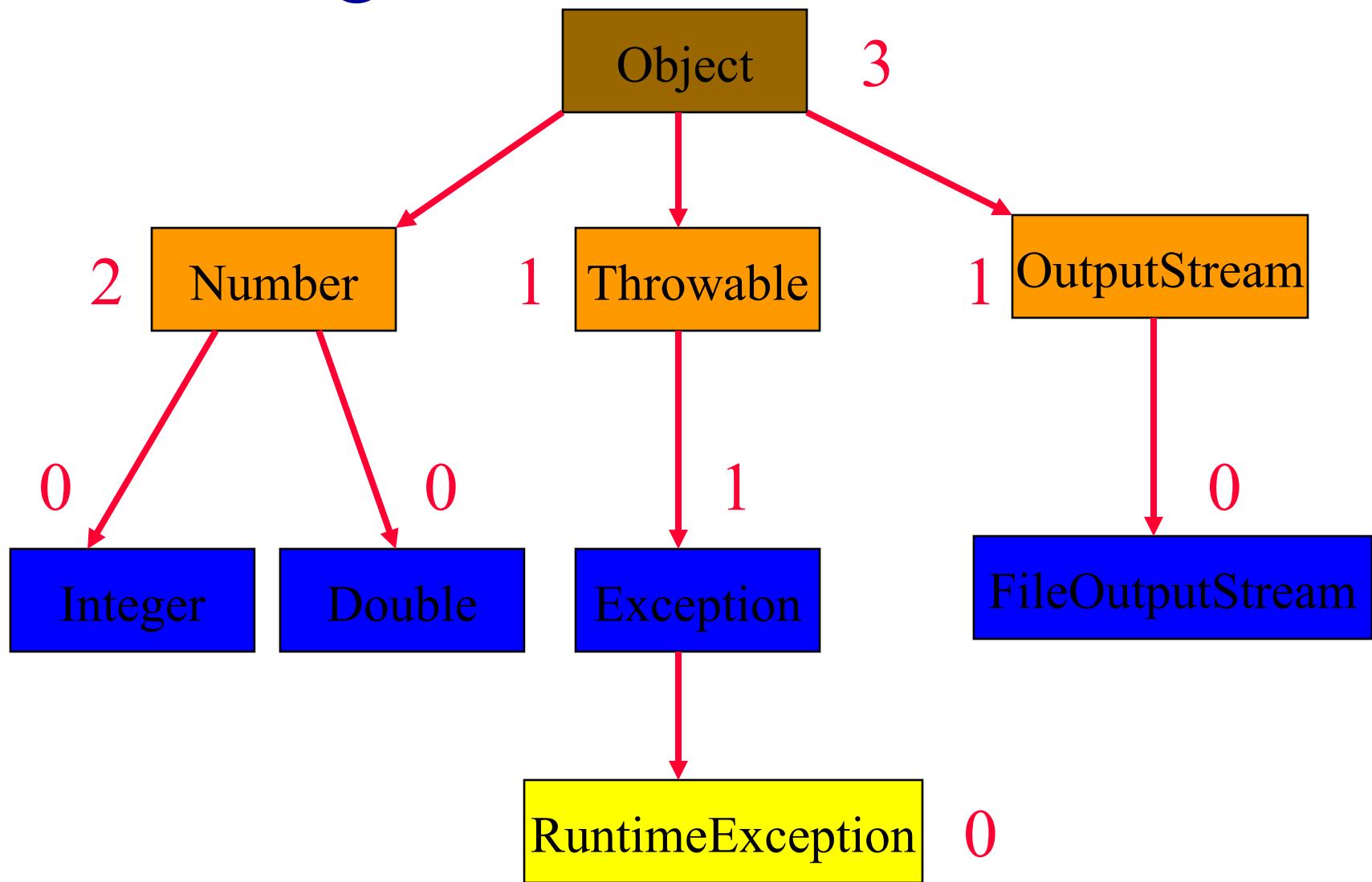


- Some texts start level numbers at 0 rather than at 1.
- Root is at level 0.
- Its children are at level 1.
- The grand children of the root are at level 2.
- And so on.
- We shall number levels with the root at level 1.

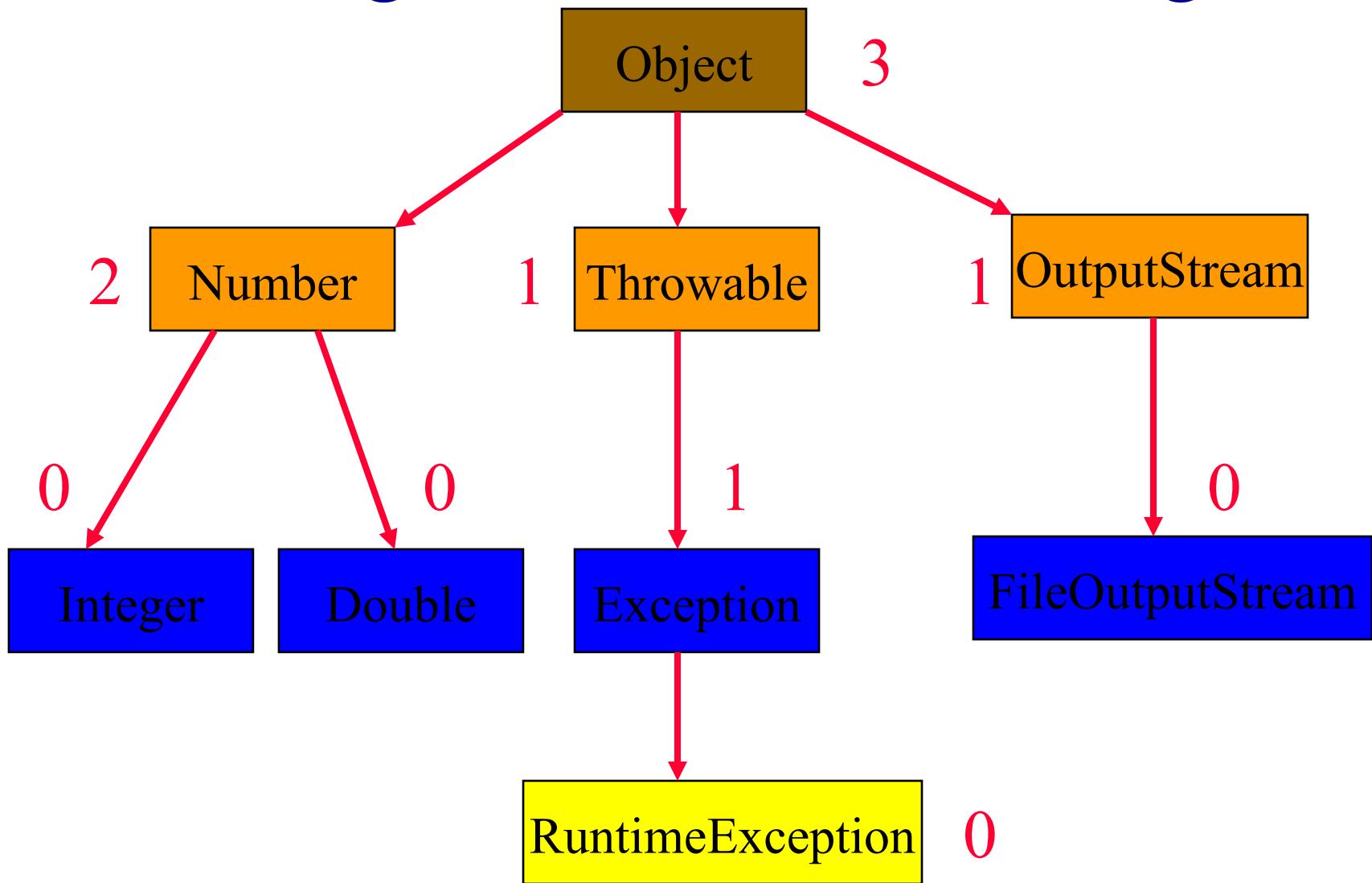
height = depth = number of levels



Node Degree = Number Of Children



Tree Degree = Max Node Degree



Degree of tree = 3.

Binary Tree

- Finite (possibly empty) collection of elements.
- A **nonempty** binary tree has a **root** element.
- The remaining elements (if any) are partitioned into **two** binary trees.
- These are called the **left** and **right** subtrees of the binary tree.

Differences Between A Tree & A Binary Tree

- No node in a binary tree may have a degree more than 2, whereas there is no limit on the degree of a node in a tree.
- A binary tree may be empty; a tree cannot be empty.

Differences Between A Tree & A Binary Tree

- The subtrees of a binary tree are ordered; those of a tree are not ordered.



- Are different when viewed as binary trees.
- Are the same when viewed as trees.

Arithmetic Expressions

- $(a + b) * (c + d) + e - f/g*h + 3.25$
- Expressions comprise three kinds of entities.
 - Operators (+, -, /, *).
 - Operands (a, b, c, d, e, f, g, h, 3.25, (a + b), (c + d), etc.).
 - Delimiters ((,)).

Operator Degree

- Number of operands that the operator requires.
- Binary operator requires two operands.
 - $a + b$
 - c / d
 - $e - f$
- Unary operator requires one operand.
 - $+ g$
 - $- h$

Infix Form

- Normal way to write an expression.
- Binary operators come **in** between their left and right operands.
 - $a * b$
 - $a + b * c$
 - $a * b / c$
 - $(a + b) * (c + d) + e - f/g*h + 3.25$

Operator Priorities

- How do you figure out the operands of an operator?
 - $a + b * c$
 - $a * b + c / d$
- This is done by assigning operator priorities.
 - $\text{priority}(*) = \text{priority}(/) > \text{priority}(+) = \text{priority}(-)$
- When an operand lies between two operators, the operand associates with the operator that has higher priority.

Tie Breaker

- When an operand lies between two operators that have the same priority, the operand associates with the operator on the left.
 - $a + b - c$
 - $a * b / c / d$

Delimiters

- Subexpression within delimiters is treated as a single operand, independent from the remainder of the expression.
 - $(a + b) * (c - d) / (e - f)$

Infix Expression Is Hard To Parse

- Need operator priorities, tie breaker, and delimiters.
- This makes computer evaluation more difficult than is necessary.
- Postfix and prefix expression forms do not rely on operator priorities, a tie breaker, or delimiters.
- So it is easier for a computer to evaluate expressions that are in these forms.

Postfix Form

- The postfix form of a variable or constant is the same as its infix form.
 - a, b, 3.25
- The relative order of operands is the same in infix and postfix forms.
- Operators come immediately **after** the postfix form of their operands.
 - Infix = a + b
 - Postfix = ab+

Postfix Examples

- Infix = $a + b * c$
 - Postfix = a b c * +
- Infix = $a * b + c$
 - Postfix = a b * c +
- Infix = $(a + b) * (c - d) / (e + f)$
 - Postfix = a b + c d - * e f + /

Unary Operators

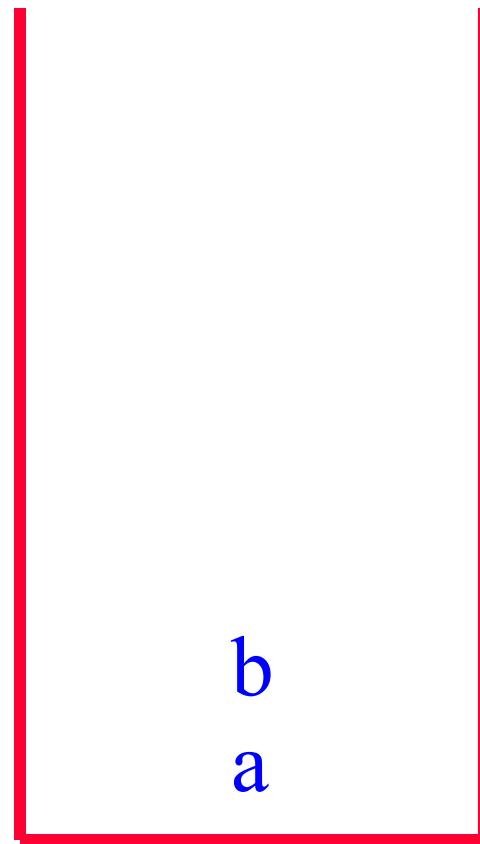
- Replace with new symbols.
 - $+ a \Rightarrow a @$
 - $+ a + b \Rightarrow a @ b +$
 - $- a \Rightarrow a ?$
 - $- a - b \Rightarrow a ? b -$

Postfix Evaluation

- Scan postfix expression from left to right pushing operands on to a stack.
- When an operator is encountered, pop as many operands as this operator needs; evaluate the operator; push the result on to the stack.
- This works because, in postfix, operators come immediately after their operands.

Postfix Evaluation

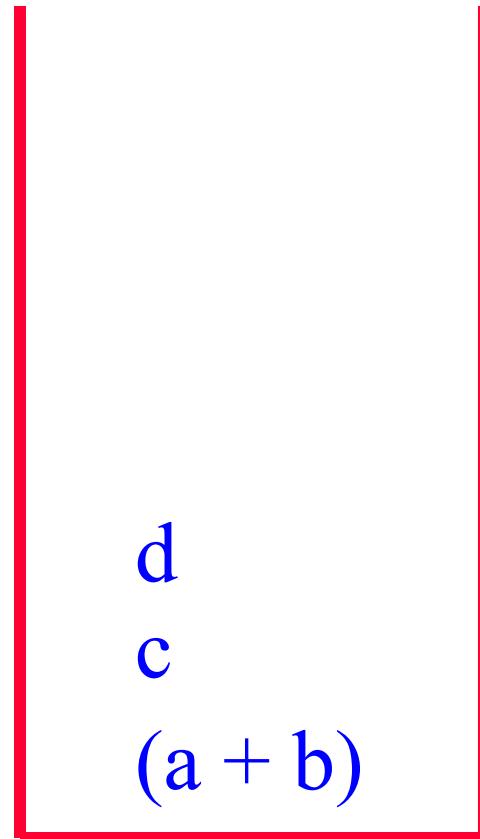
- $(a + b) * (c - d) / (e + f)$
- $a\ b + c\ d - * e\ f + /$
- $a\ b + c\ d - * e\ f + /$
- $a\ b + c\ d - * e\ f + /$
- $a\ b + c\ d - * e\ f + /$



stack

Postfix Evaluation

- $(a + b) * (c - d) / (e + f)$
- $a b + c d - * e f + /$
- $a b + c d - * e f + /$
- $a b + c d - * e f + /$
- $a b + c d - * e f + /$
- $a b + c d - * e f + /$
- $a b + c d - * e f + /$
- $a b + c d - * e f + /$



stack

Postfix Evaluation

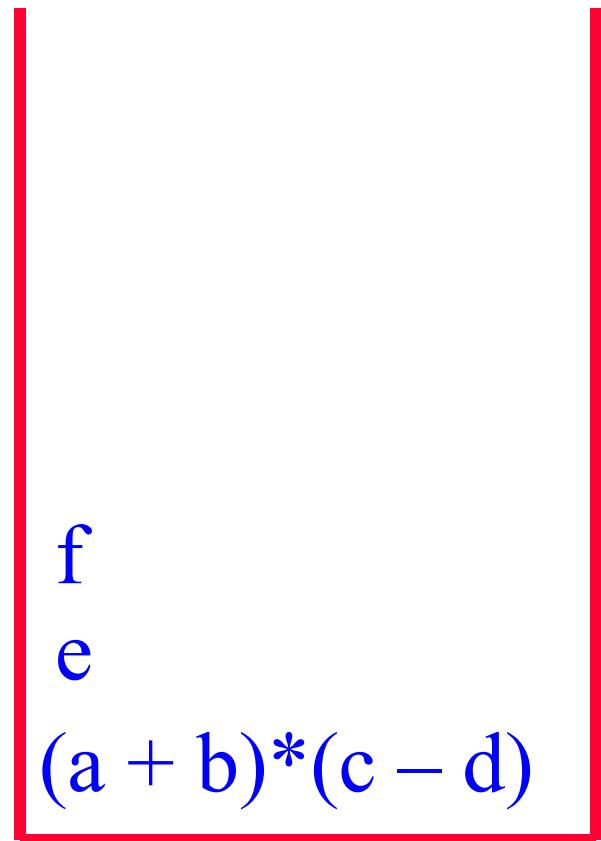
- $(a + b) * (c - d) / (e + f)$
- $a\ b\ +\ c\ d\ -\ *\ e\ f\ +\ /$
- $a\ b\ +\ c\ d\ -\ *\ e\ f\ +\ /$

$(c - d)$
 $(a + b)$

stack

Postfix Evaluation

- $(a + b) * (c - d) / (e + f)$
- $a b + c d - * e f + /$
- $a b + c d - * e f + /$
- $a b + c d - * e f + /$
- $a b + c d - * e f + /$
- $a b + c d - * e f + /$



stack

Postfix Evaluation

- $(a + b) * (c - d) / (e + f)$
- $a b + c d - * e f + /$
- $a b + c d - * e f + /$
- $a b + c d - * e f + /$
- $a b + c d - * e f + /$
- $a b + c d - * e f + /$
- $a b + c d - * e f + /$

$(e + f)$
 $(a + b)* (c - d)$

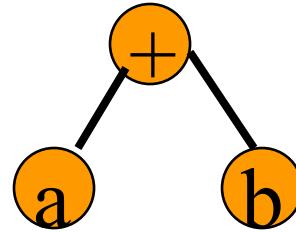
stack

Prefix Form

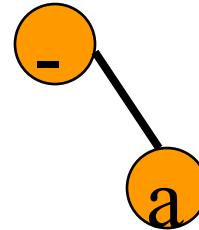
- The prefix form of a variable or constant is the same as its infix form.
 - $a, b, 3.25$
- The relative order of operands is the same in infix and prefix forms.
- Operators come immediately **before** the prefix form of their operands.
 - Infix = $a + b$
 - Postfix = $ab+$
 - Prefix = $+ab$

Binary Tree Form

- $a + b$

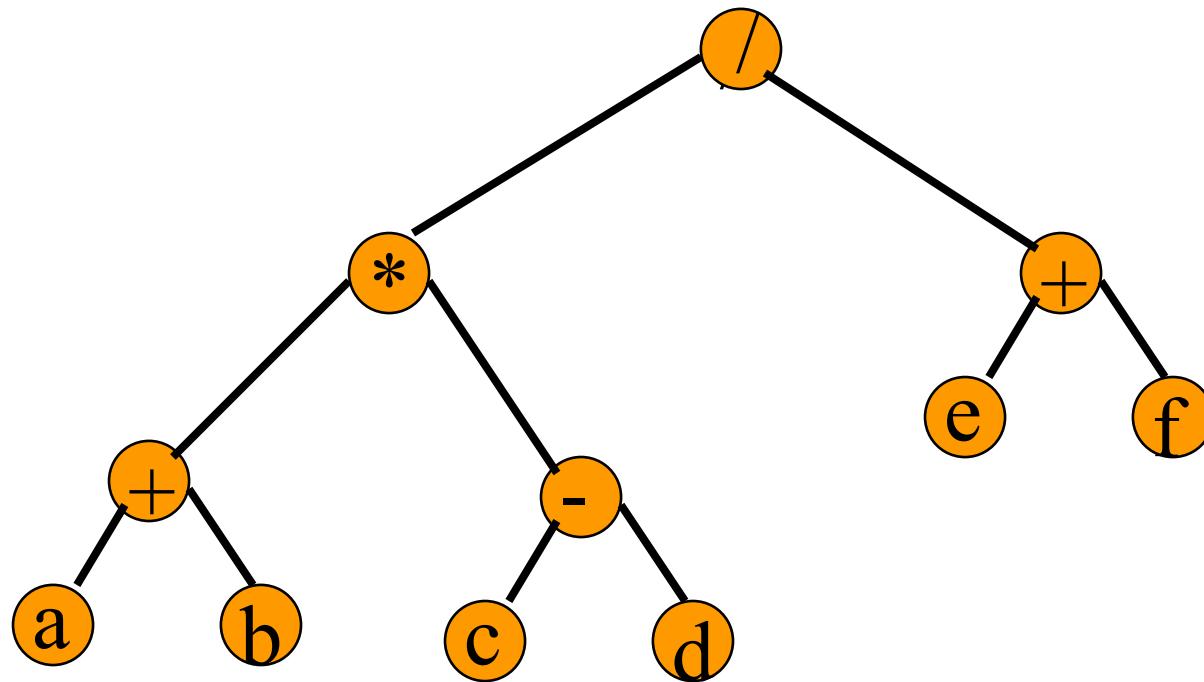


- $- a$



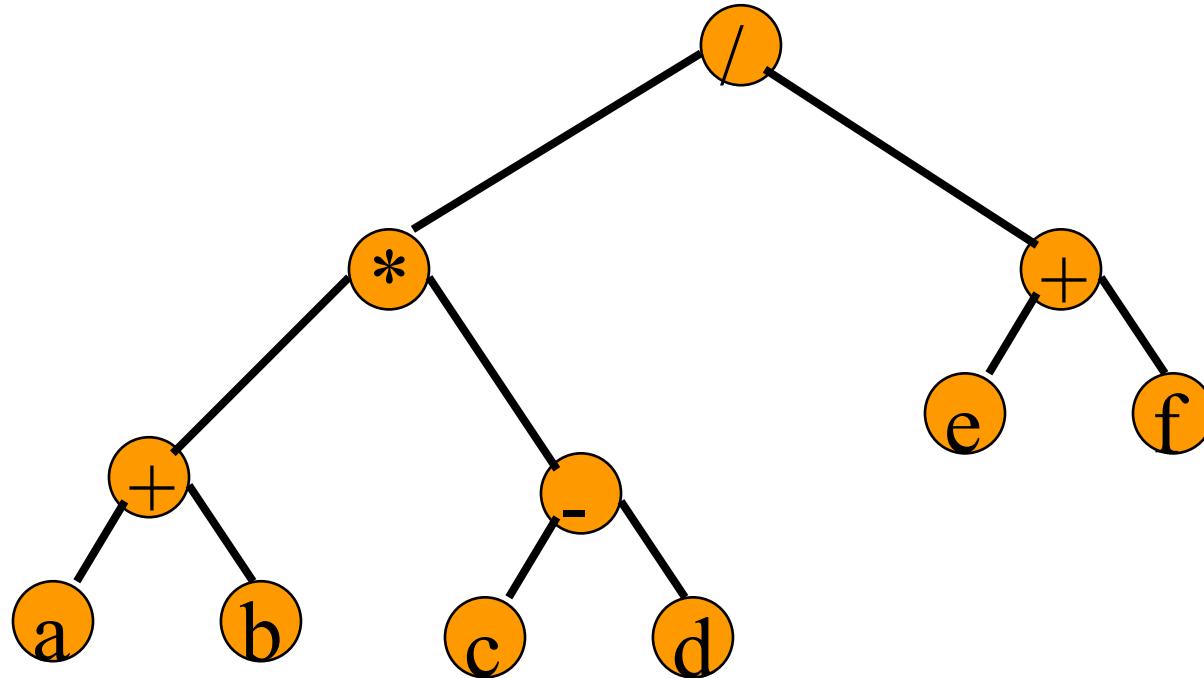
Binary Tree Form

- $(a + b) * (c - d) / (e + f)$

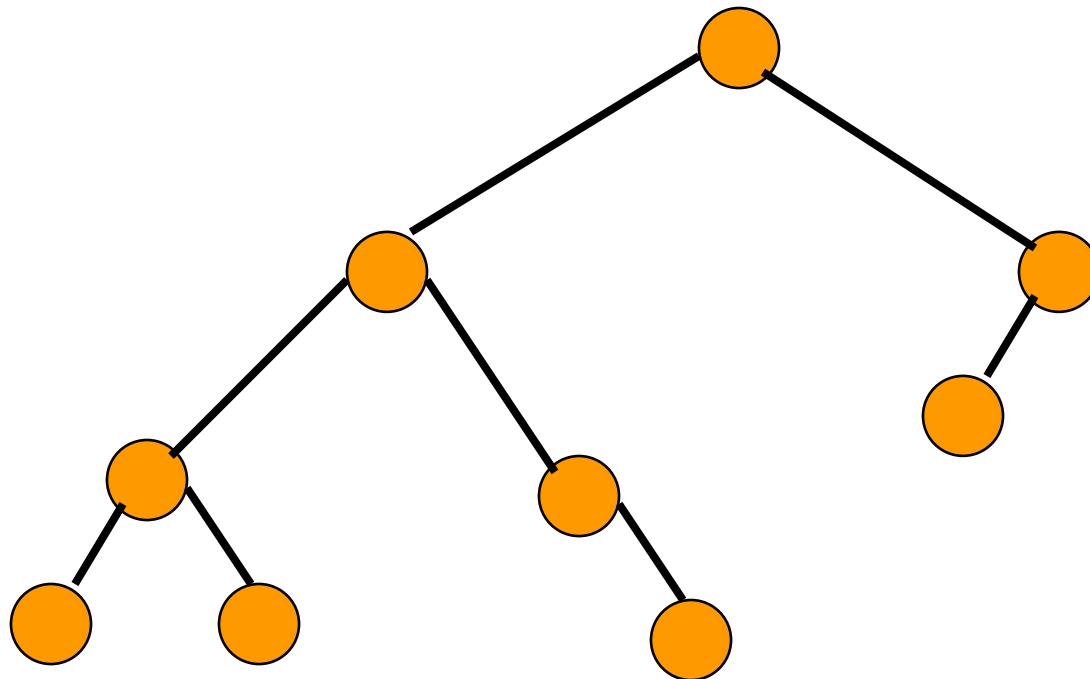
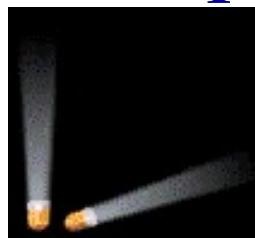
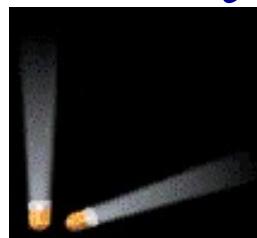


Merits Of Binary Tree Form

- Left and right operands are easy to visualize.
- Code optimization algorithms work with the binary tree form of an expression.
- Simple recursive evaluation of expression.

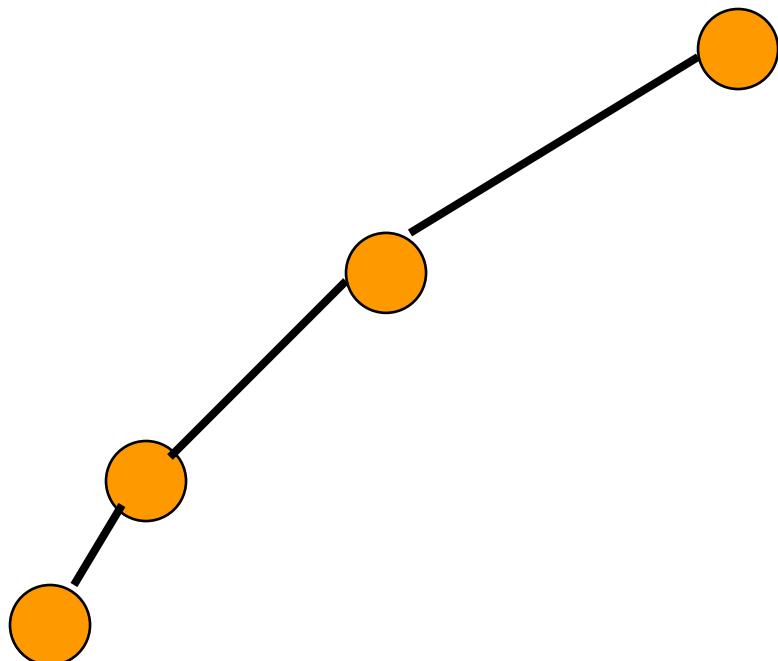


Binary Tree Properties & Representation



Minimum Number Of Nodes

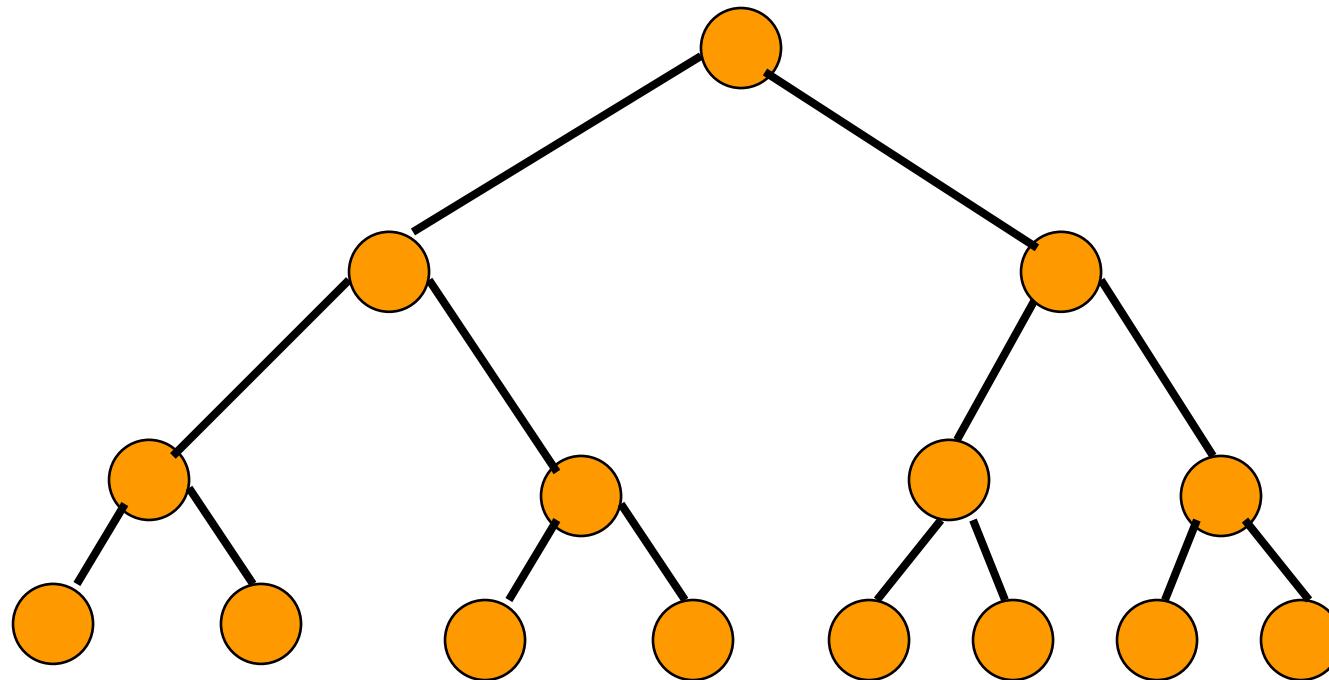
- Minimum number of nodes in a binary tree whose height is h .
- At least one node at each of first h levels.



minimum number of
nodes is h

Maximum Number Of Nodes

- All possible nodes at first h levels are present.



Maximum number of nodes

$$= 1 + 2 + 4 + 8 + \dots + 2^{h-1}$$

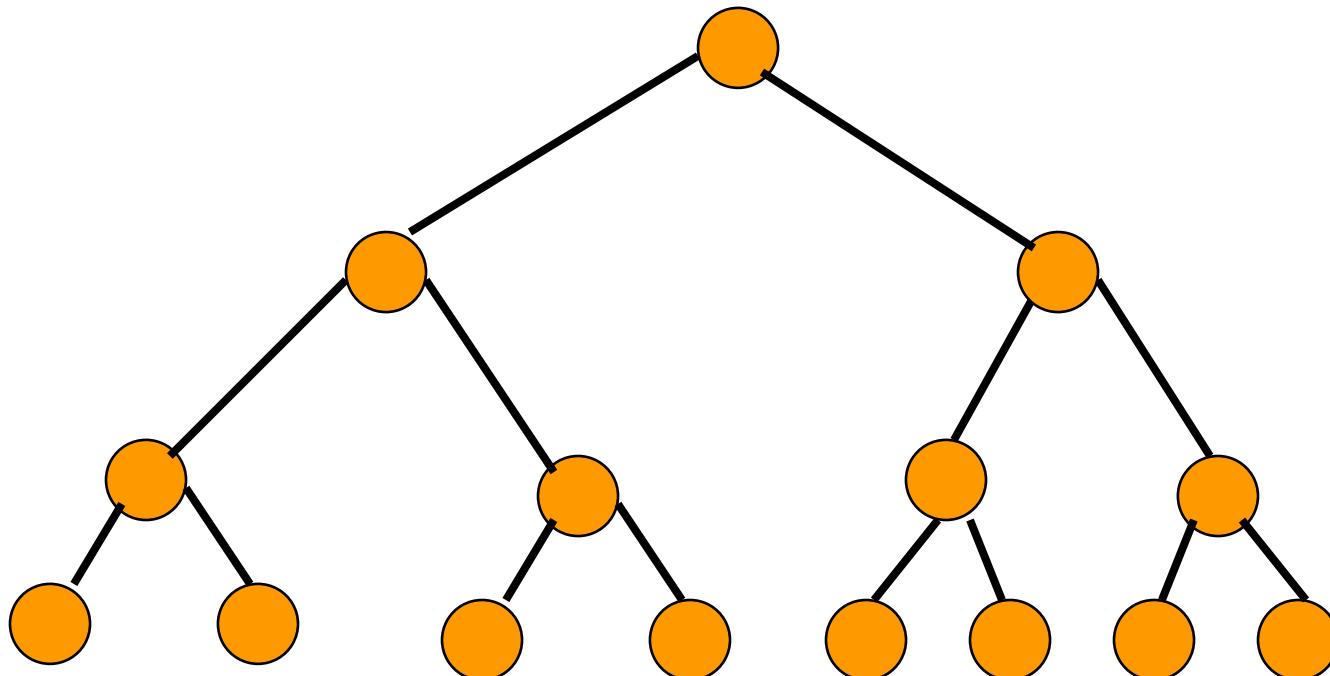
$$= 2^h - 1$$

Number Of Nodes & Height

- Let n be the number of nodes in a binary tree whose height is h .
- $h \leq n \leq 2^h - 1$
- $\log_2(n+1) \leq h \leq n$

Full Binary Tree

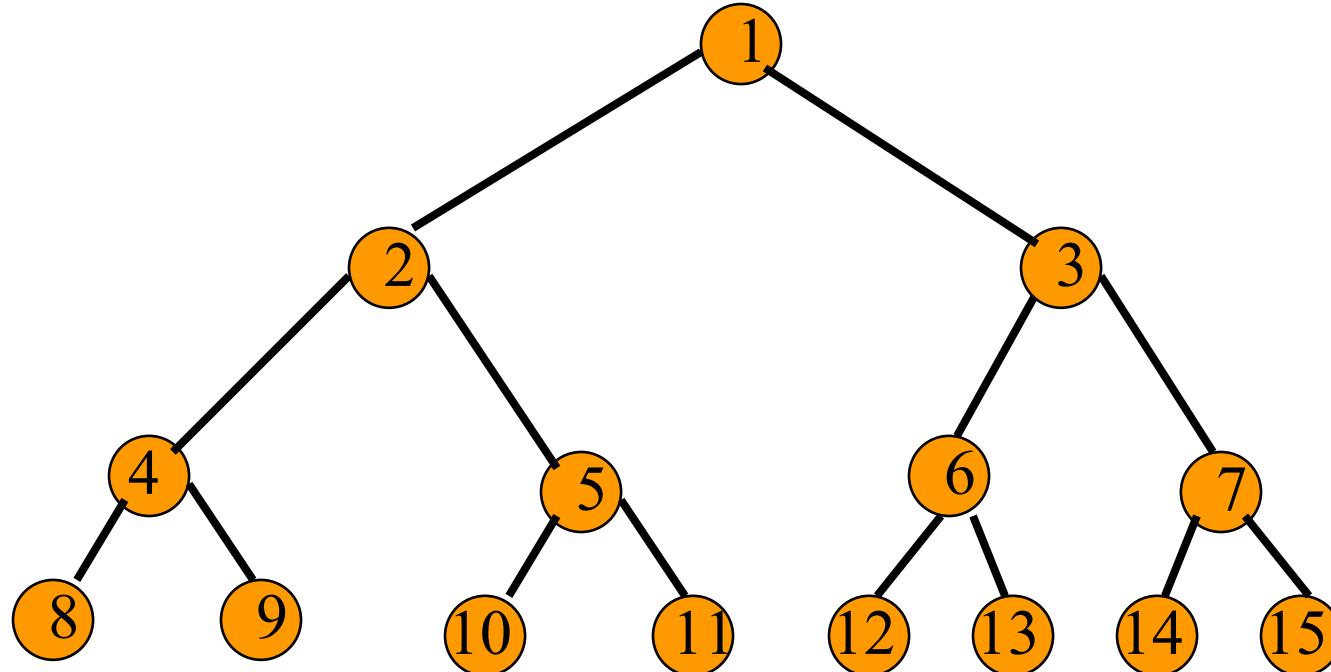
- A full binary tree of a given height h has $2^h - 1$ nodes.



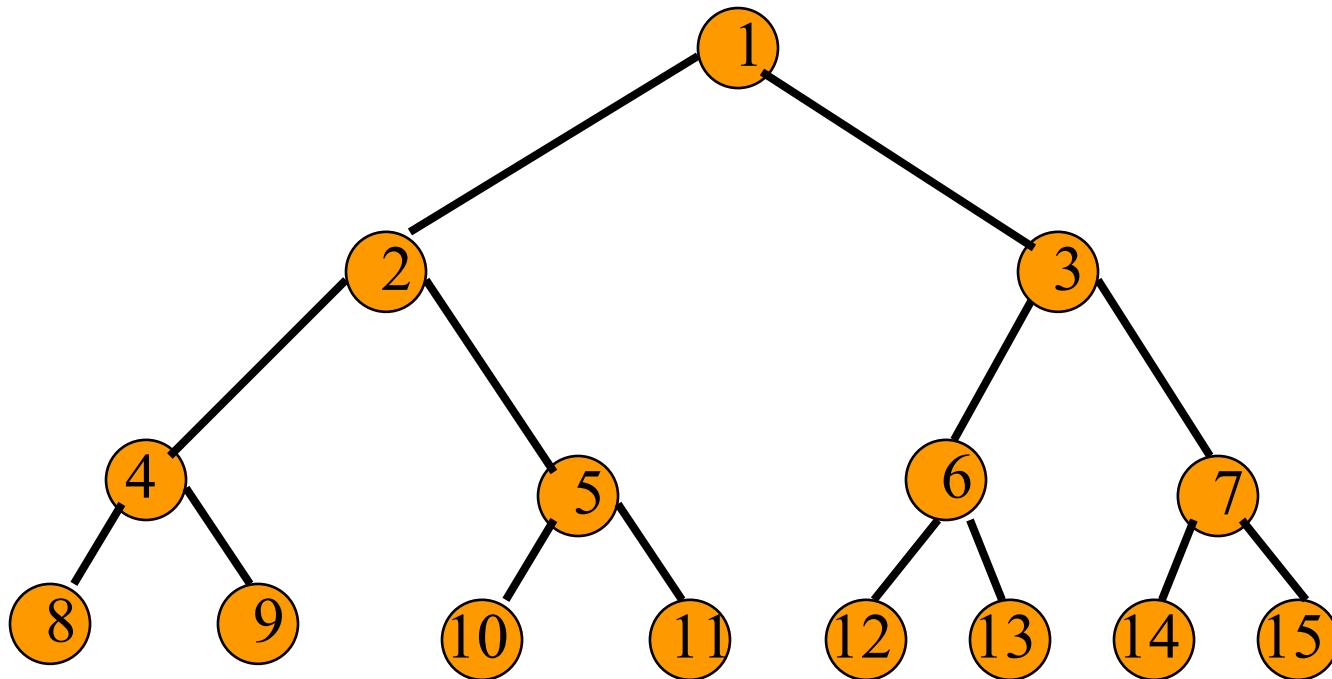
Height 4 full binary tree.

Numbering Nodes In A Full Binary Tree

- Number the nodes 1 through $2^h - 1$.
- Number by levels from top to bottom.
- Within a level number from left to right.

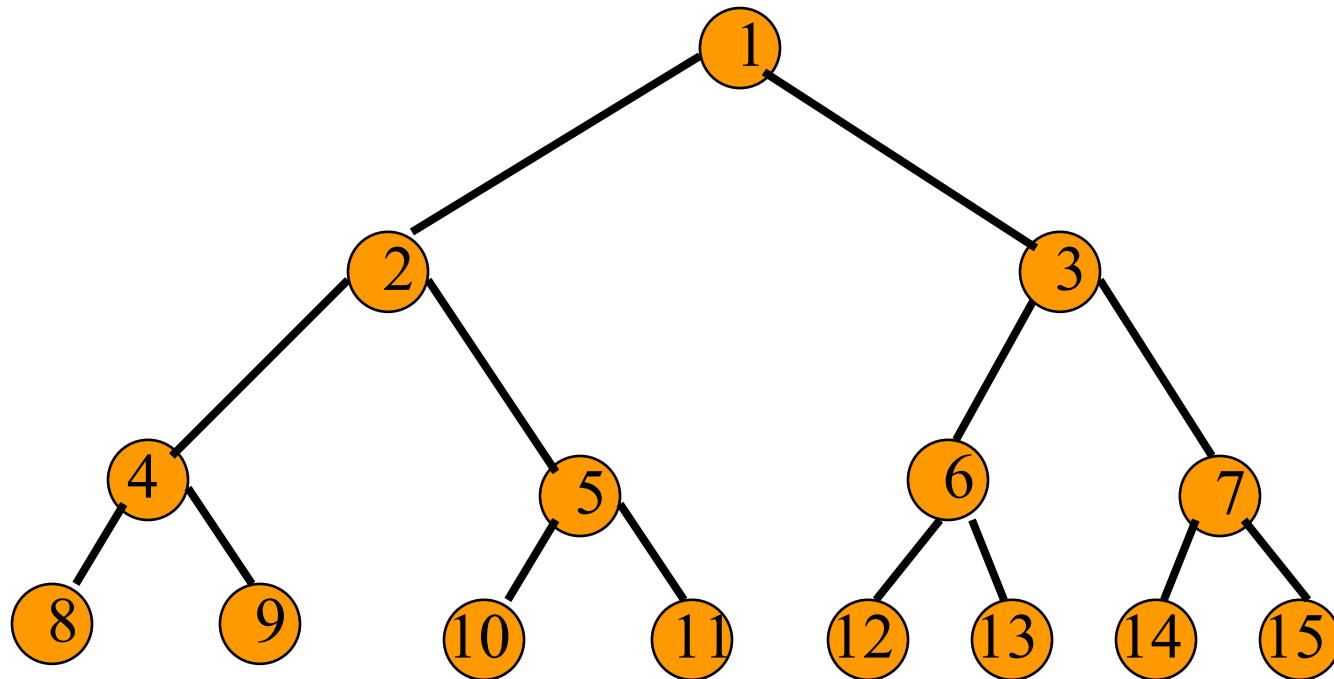


Node Number Properties



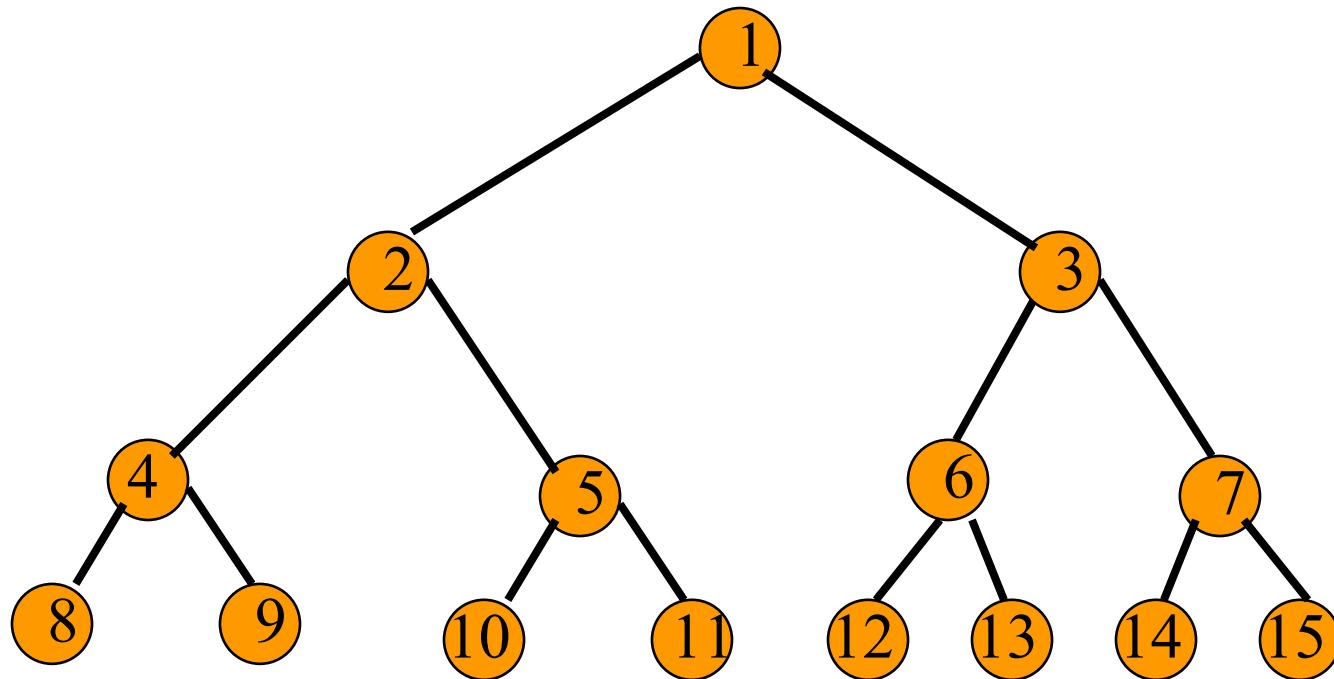
- Parent of node i is node $i / 2$, unless $i = 1$.
- Node 1 is the root and has no parent.

Node Number Properties



- Left child of node i is node $2i$, unless $2i > n$, where n is the number of nodes.
- If $2i > n$, node i has no left child.

Node Number Properties

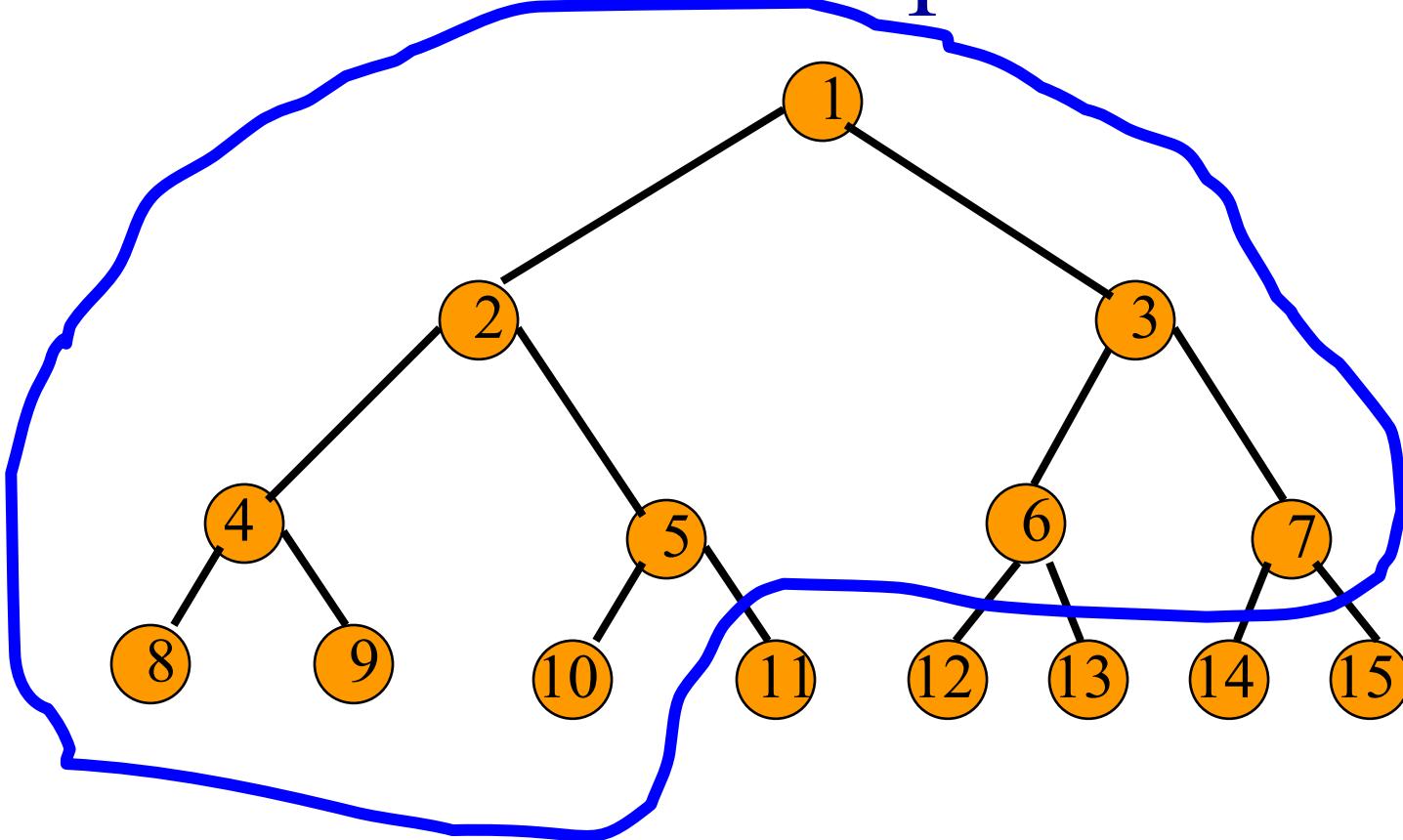


- Right child of node i is node $2i+1$, unless $2i+1 > n$, where n is the number of nodes.
- If $2i+1 > n$, node i has no right child.

Complete Binary Tree With n Nodes

- Start with a full binary tree that has at least n nodes.
- Number the nodes as described earlier.
- The binary tree defined by the nodes numbered 1 through n is the unique n node complete binary tree.

Example



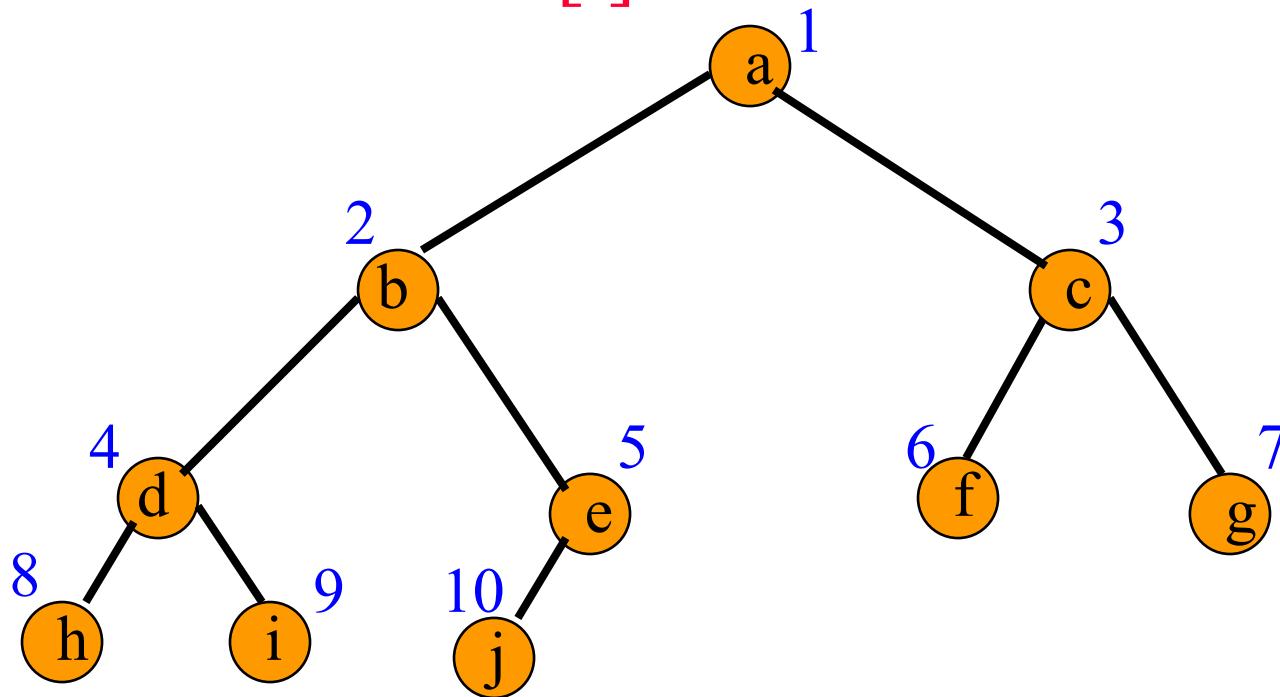
- Complete binary tree with 10 nodes.

Binary Tree Representation

- Array representation.
- Linked representation.

Array Representation

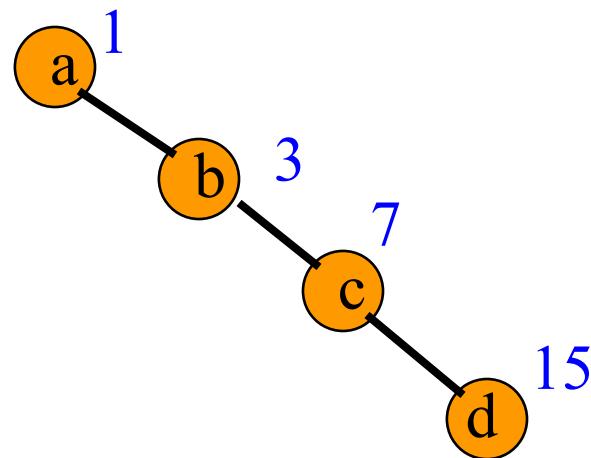
- Number the nodes using the numbering scheme for a full binary tree. The node that is numbered i is stored in `tree[i]`.



`tree[]`

	a	b	c	d	e	f	g	h	i	j
0					5				10	

Right-Skewed Binary Tree



tree[]

	a	-	b	-	-	-	c	-	-	-	-	-	-	-	d
0			5				10								15

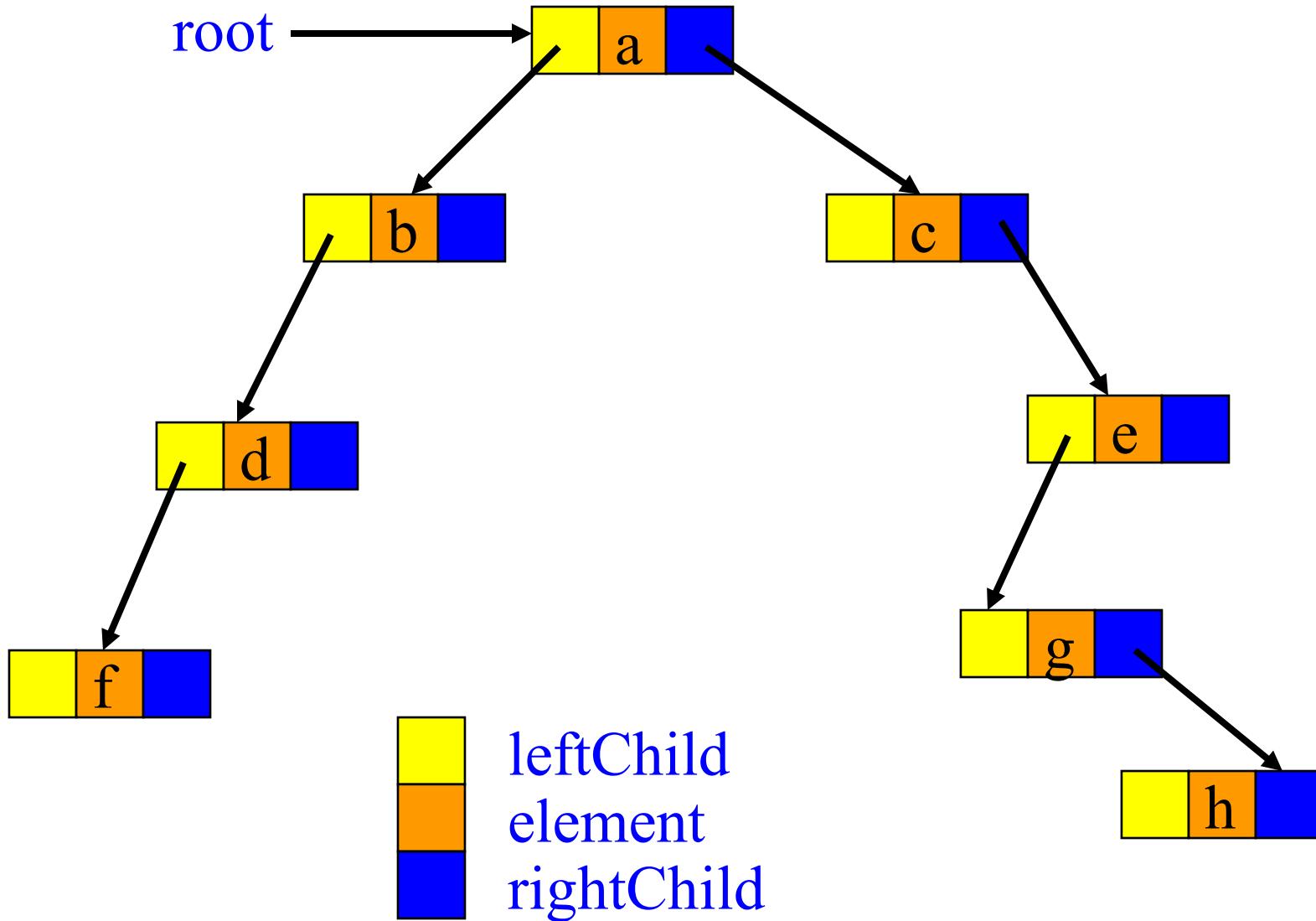
Linked Representation

- Each binary tree node is represented as an object whose data type is **BinaryTreeNode**.
- The space required by an **n** node binary tree is **n * (space required by one node)**.

The Class BinaryTreeNode

```
package dataStructures;  
public class BinaryTreeNode  
{  
    Object element;  
    BinaryTreeNode leftChild; // left subtree  
    BinaryTreeNode rightChild; // right subtree  
    // constructors and any other methods  
    // come here  
}
```

Linked Representation Example



Some Binary Tree Operations

- Determine the height.
- Determine the number of nodes.
- Make a clone.
- Determine if two binary trees are clones.
- Display the binary tree.
- Evaluate the arithmetic expression represented by a binary tree.
- Obtain the infix form of an expression.
- Obtain the prefix form of an expression.
- Obtain the postfix form of an expression.

Binary Tree Traversal

- Many binary tree operations are done by performing a **traversal** of the binary tree.
- In a traversal, each element of the binary tree is **visited** exactly once.
- During the **visit** of an element, all action (make a clone, display, evaluate the operator, etc.) with respect to this element is taken.

Binary Tree Traversal Methods

- Preorder
- Inorder
- Postorder
- Level order

Binary Tree Traversal Methods



- In a traversal of a binary tree, each element of the binary tree is **visited** exactly once.
- During the **visit** of an element, all action (make a clone, display, evaluate the operator, etc.) with respect to this element is taken.

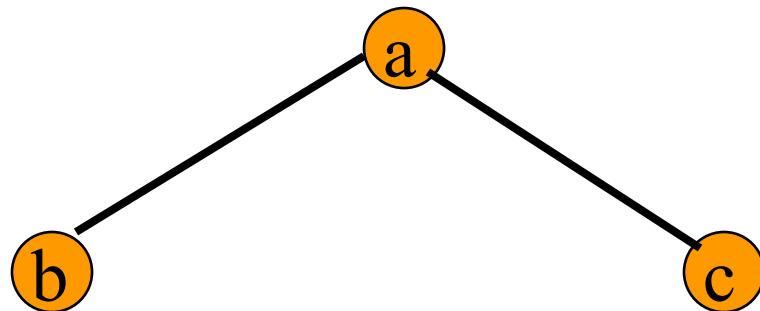
Binary Tree Traversal Methods

- Preorder
- Inorder
- Postorder
- Level order

Preorder Traversal

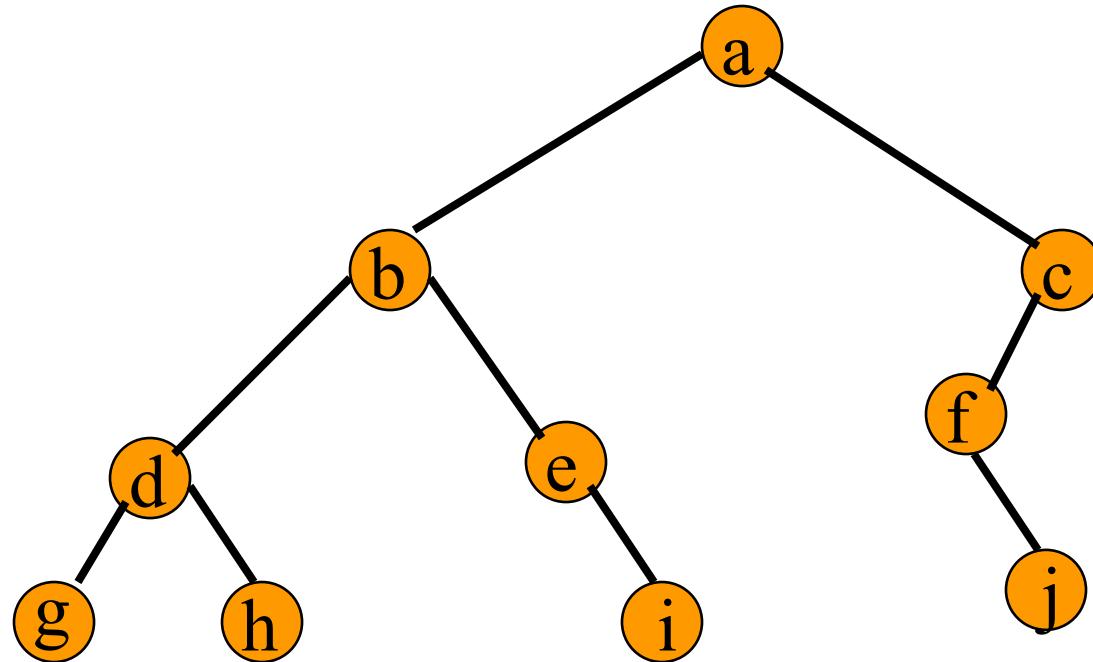
```
public static void preOrder(BinaryTreeNode t)
{
    if (t != null)
    {
        visit(t);
        preOrder(t.leftChild);
        preOrder(t.rightChild);
    }
}
```

Preorder Example (visit = print)



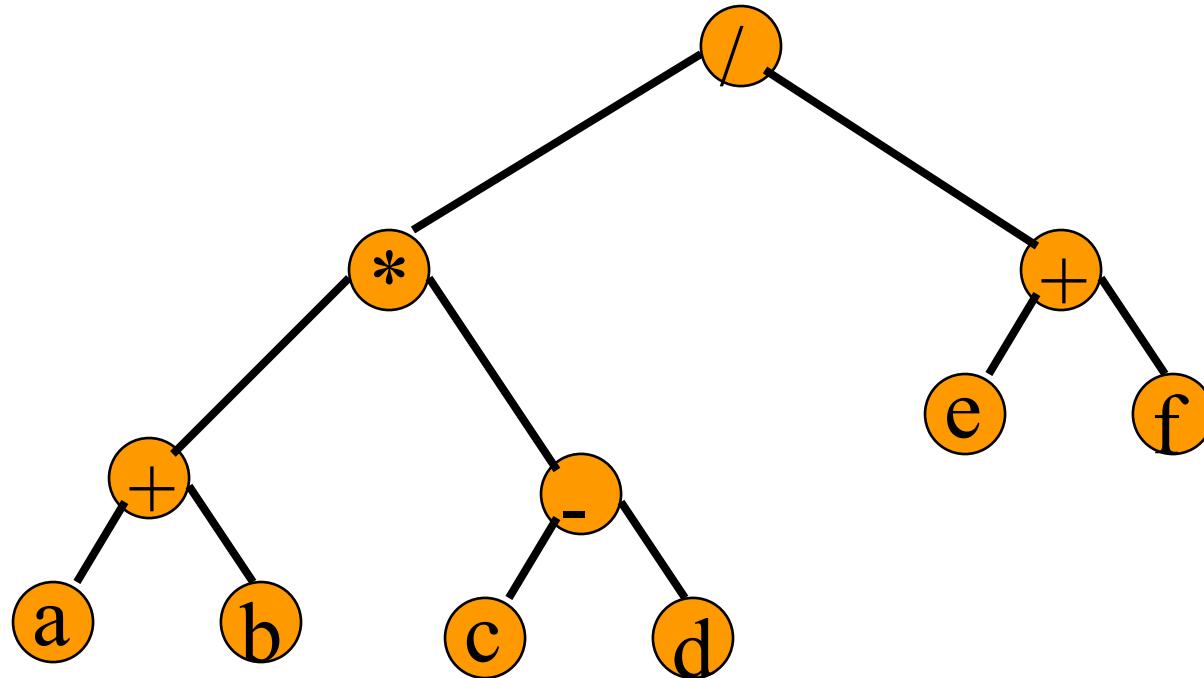
a b c

Preorder Example (visit = print)



a b d g h e i c f j

Preorder Of Expression Tree



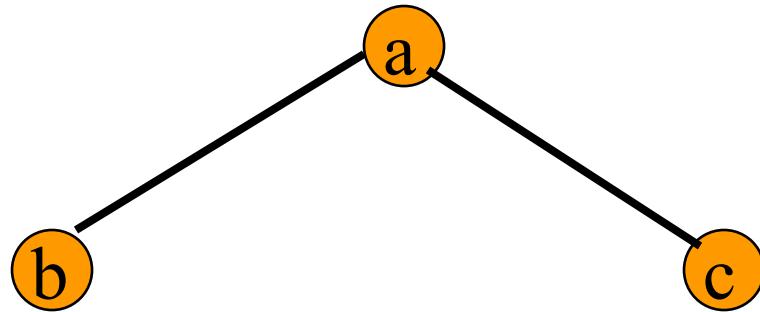
/ * + a b - c d + e f

Gives prefix form of expression!

Inorder Traversal

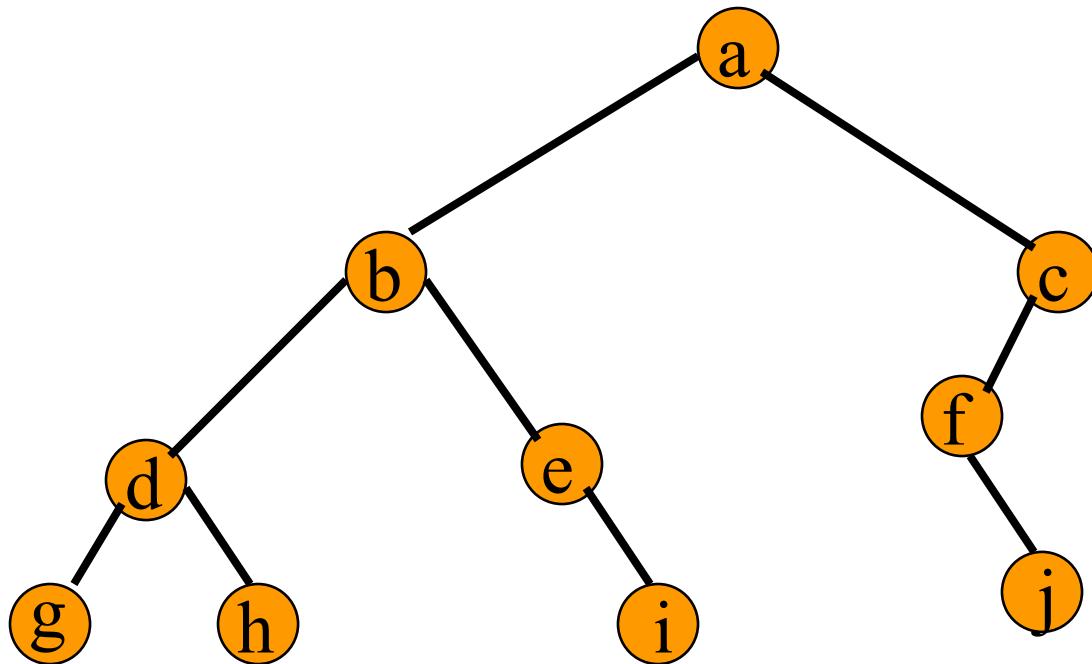
```
public static void inOrder(BinaryTreeNode t)
{
    if (t != null)
    {
        inOrder(t.leftChild);
        visit(t);
        inOrder(t.rightChild);
    }
}
```

Inorder Example (visit = print)



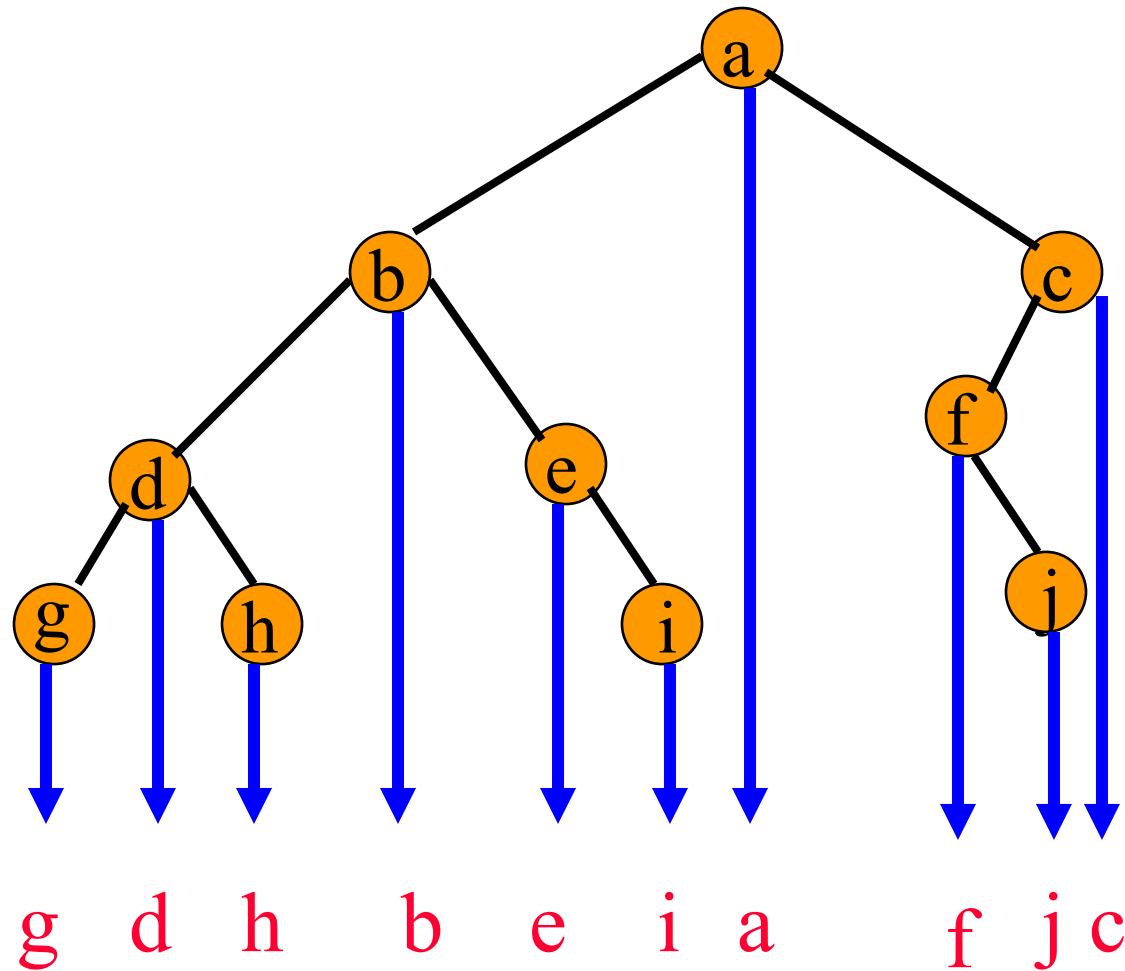
b a c

Inorder Example (visit = print)

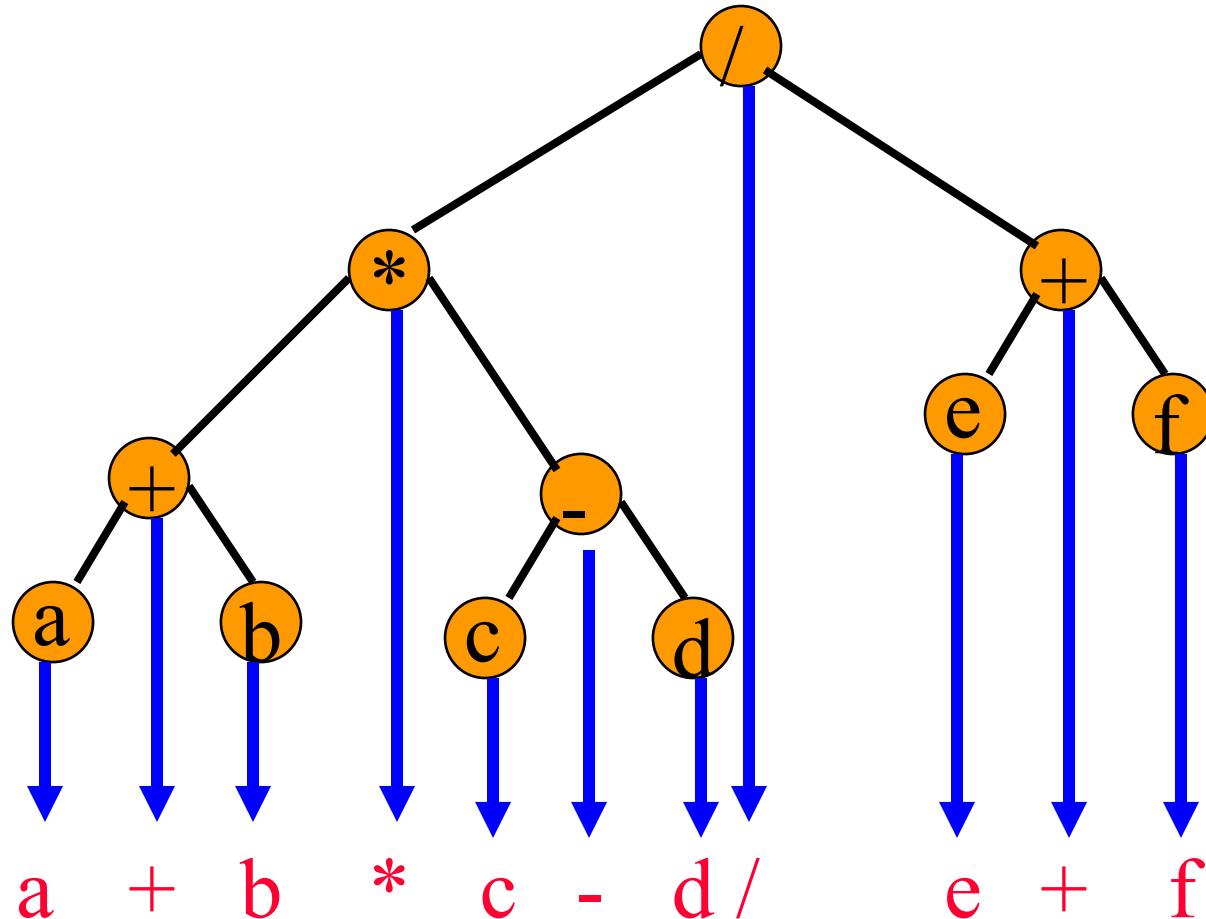


g d h b e i a f j c

Inorder By Projection (Squishing)



Inorder Of Expression Tree

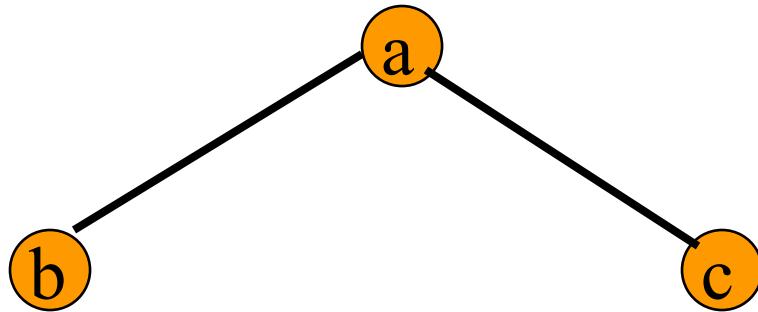


Gives infix form of expression (sans parentheses)!

Postorder Traversal

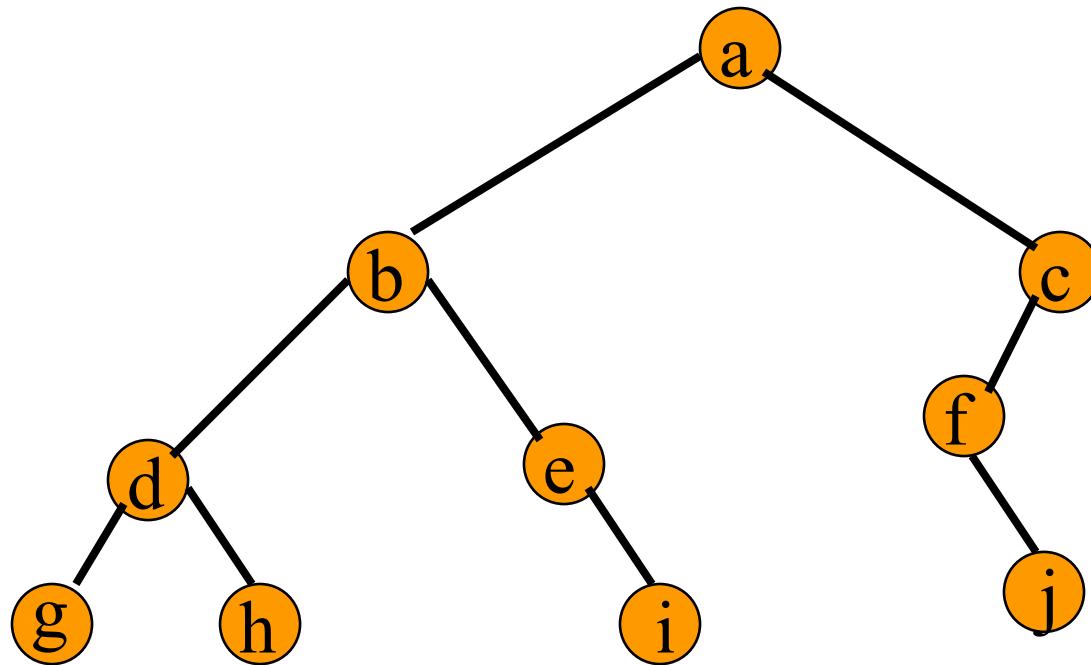
```
public static void postOrder(BinaryTreeNode t)
{
    if (t != null)
    {
        postOrder(t.leftChild);
        postOrder(t.rightChild);
        visit(t);
    }
}
```

Postorder Example (visit = print)



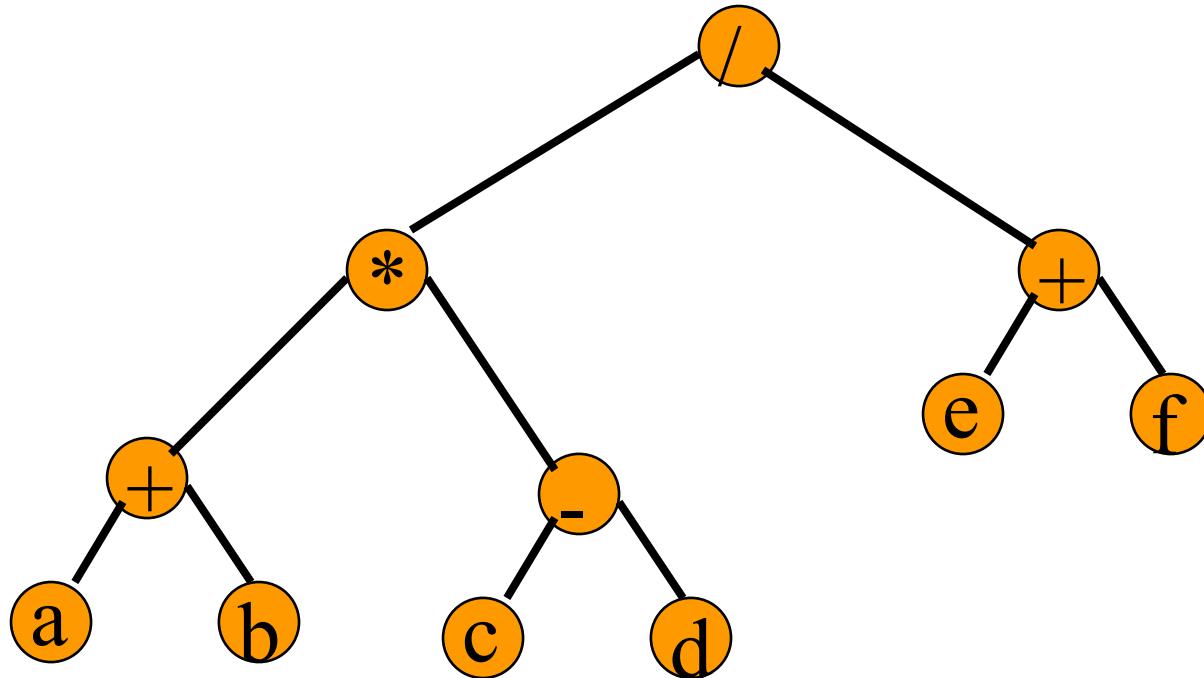
b c a

Postorder Example (visit = print)



g h d i e b j f c a

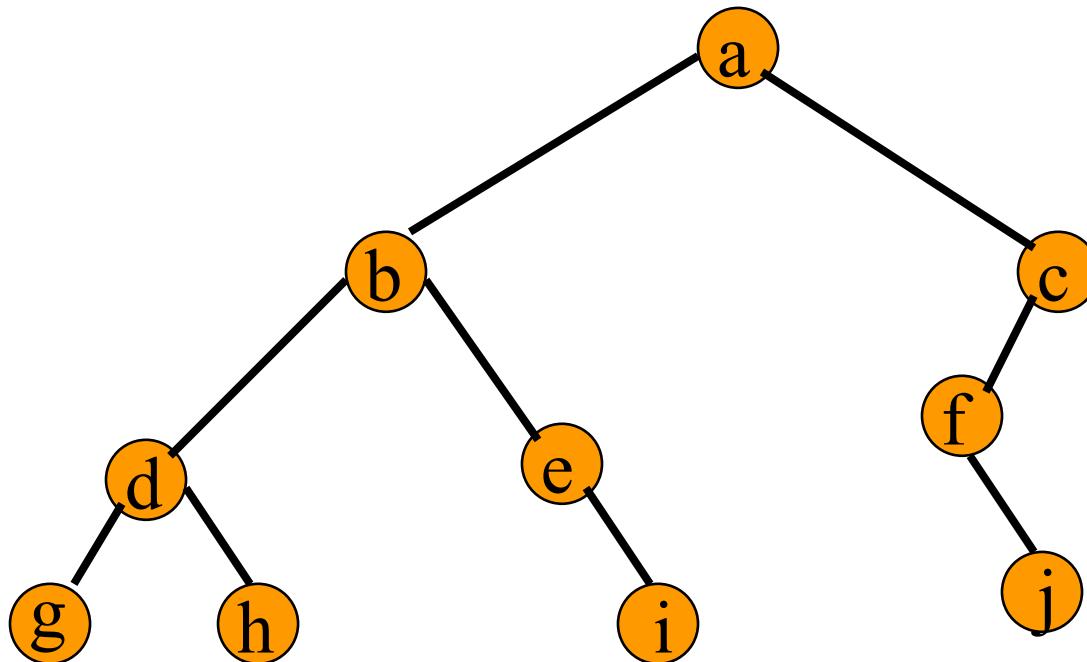
Postorder Of Expression Tree



a b + c d - * e f + /

Gives postfix form of expression!

Traversal Applications



- Make a clone.
- Determine height.
- Determine number of nodes.

Level Order

Let t be the tree root.

while ($t \neq \text{null}$)

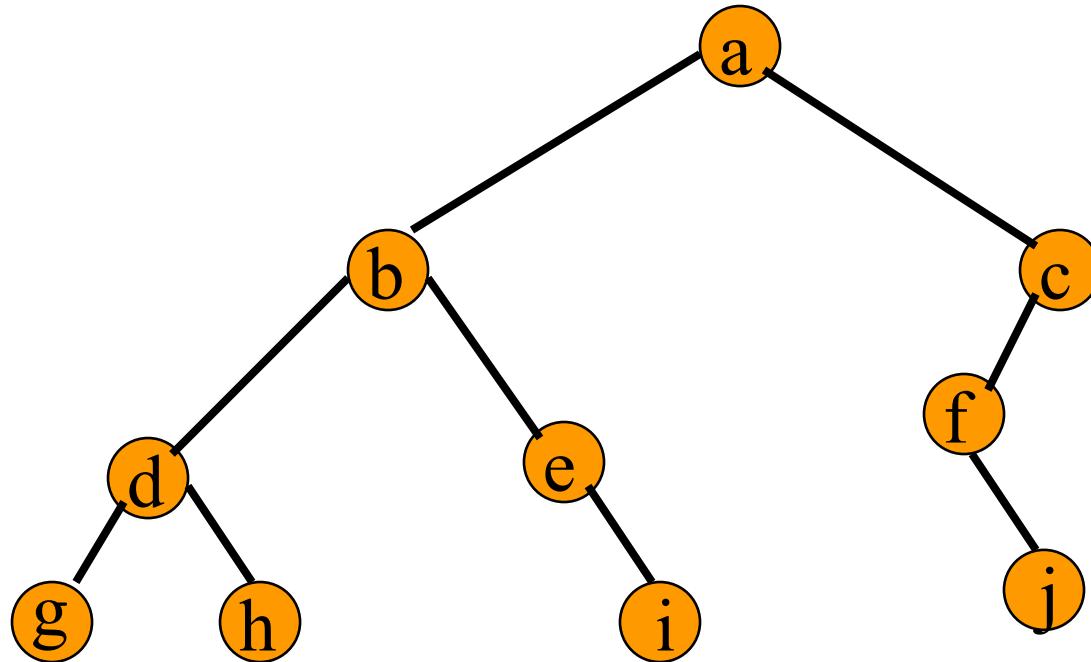
{

visit t and put its children on a FIFO queue;
remove a node from the FIFO queue and call
it t ;

// remove returns null when queue is empty

}

Level-Order Example (visit = print)



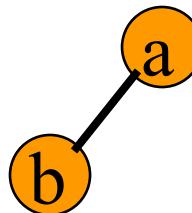
a b c d e f g h i j

Binary Tree Construction

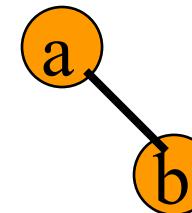
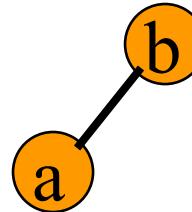
- Suppose that the elements in a binary tree are distinct.
- Can you construct the binary tree from which a given traversal sequence came?
- When a traversal sequence has more than one element, the binary tree is not uniquely defined.
- Therefore, the tree from which the sequence was obtained cannot be reconstructed uniquely.

Some Examples

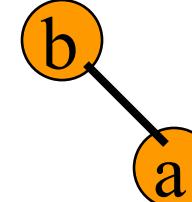
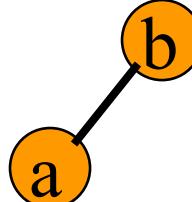
preorder
= ab



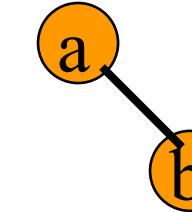
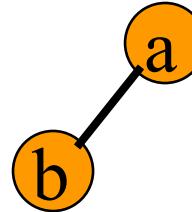
inorder
= ab



postorder
= ab



level order
= ab



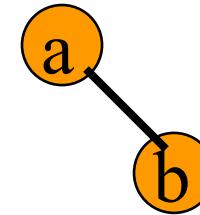
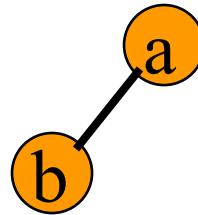
Binary Tree Construction

- Can you construct the binary tree, given two traversal sequences?
- Depends on which two sequences are given.

Preorder And Postorder

preorder = ab

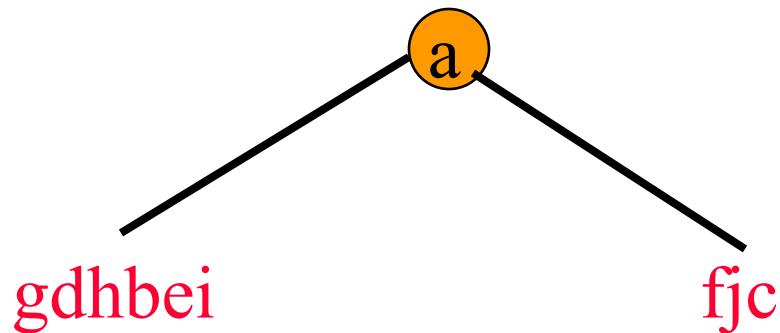
postorder = ba



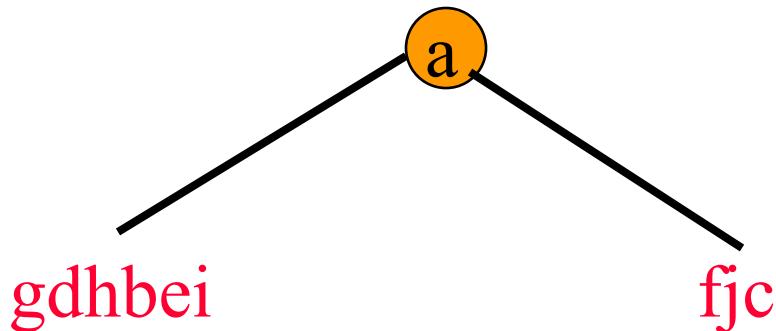
- Preorder and postorder do not uniquely define a binary tree.
- Nor do preorder and level order (same example).
- Nor do postorder and level order (same example).

Inorder And Preorder

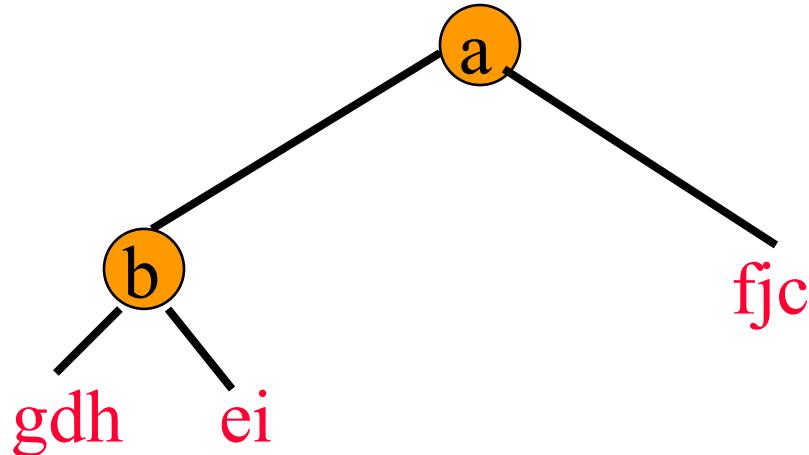
- inorder = g d h b e i a f j c
- preorder = a b d g h e i c f j
- Scan the preorder left to right using the inorder to separate left and right subtrees.
- a is the root of the tree; gdhbei are in the left subtree; fjc are in the right subtree.



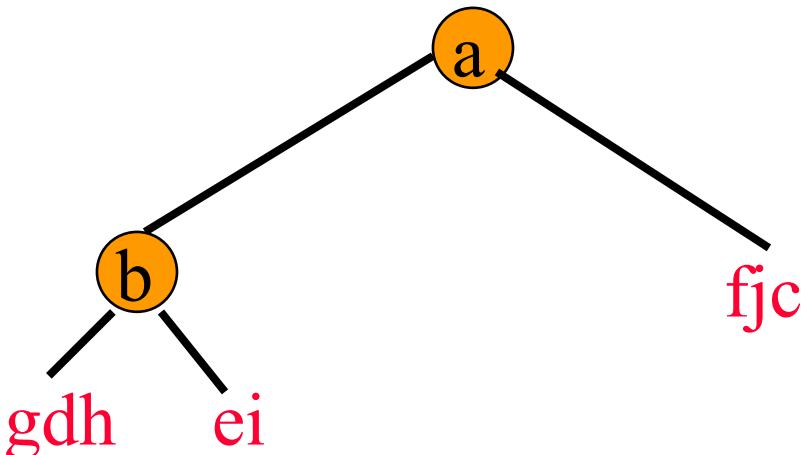
Inorder And Preorder



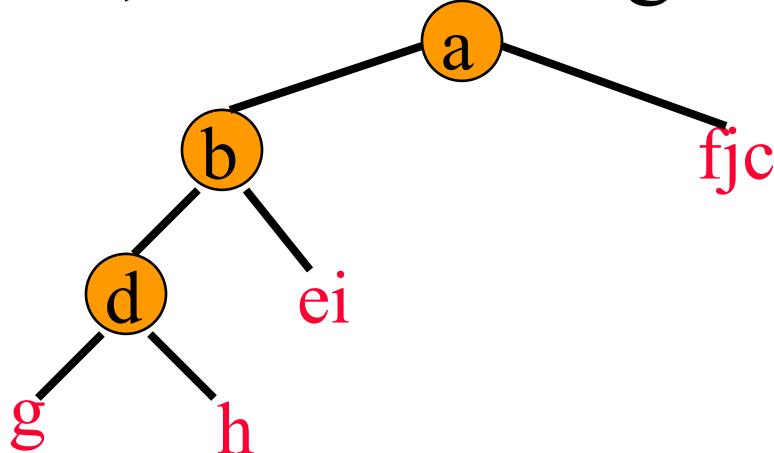
- preorder = a b d g h e i c f j
- b is the next root; gdh are in the left subtree; ei are in the right subtree.



Inorder And Preorder



- preorder = a b d g h e i c f j
- d is the next root; g is in the left subtree; h is in the right subtree.



Inorder And Postorder

- Scan postorder from right to left using inorder to separate left and right subtrees.
- $\text{inorder} = \text{g d h b e i a f j c}$
- $\text{postorder} = \text{g h d i e b j f c a}$
- Tree root is **a**; **gdhbei** are in left subtree; **fjc** are in right subtree.

Inorder And Level Order

- Scan level order from left to right using inorder to separate left and right subtrees.
- inorder = g d h b e i a f j c
- level order = a b c d e f g h i j
- Tree root is a; gdhbei are in left subtree; fjc are in right subtree.

Priority Queues



Two kinds of priority queues:

- Min priority queue.
- Max priority queue.

Min Priority Queue

- Collection of elements.
- Each element has a priority or key.
- Supports following operations:
 - isEmpty
 - size
 - add/put an element into the priority queue
 - get element with min priority
 - remove element with min priority

Max Priority Queue

- Collection of elements.
- Each element has a priority or key.
- Supports following operations:
 - isEmpty
 - size
 - add/put an element into the priority queue
 - get element with **max** priority
 - remove element with **max** priority

Complexity Of Operations

Two good implementations are heaps
and leftist trees.

isEmpty, size, and get => O(1) time

put and remove => O(log n) time
where **n** is the size of the priority
queue

Applications

Sorting

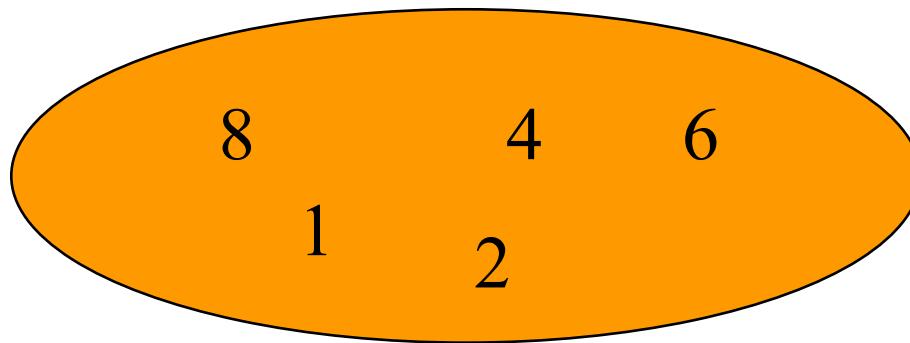
- use element key as priority
- put elements to be sorted into a priority queue
- extract elements in priority order
 - if a min priority queue is used, elements are extracted in ascending order of priority (or key)
 - if a max priority queue is used, elements are extracted in descending order of priority (or key)

Sorting Example

Sort five elements whose keys are 6, 8, 2, 4, 1
using a max priority queue.

- Put the five elements into a max priority queue.
- Do five remove max operations placing removed elements into the sorted array from right to left.

After Putting Into Max Priority Queue

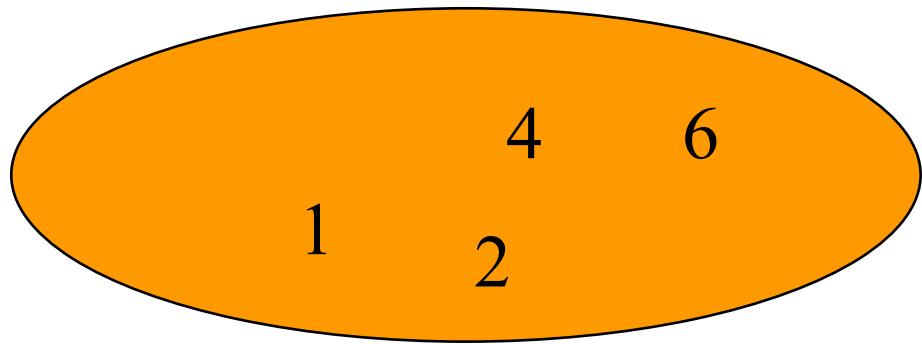


Max Priority
Queue

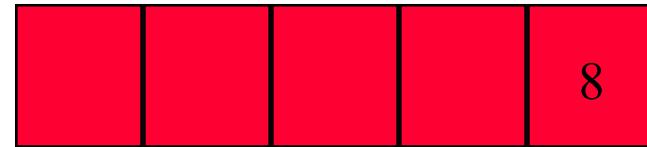


Sorted Array

After First Remove Max Operation

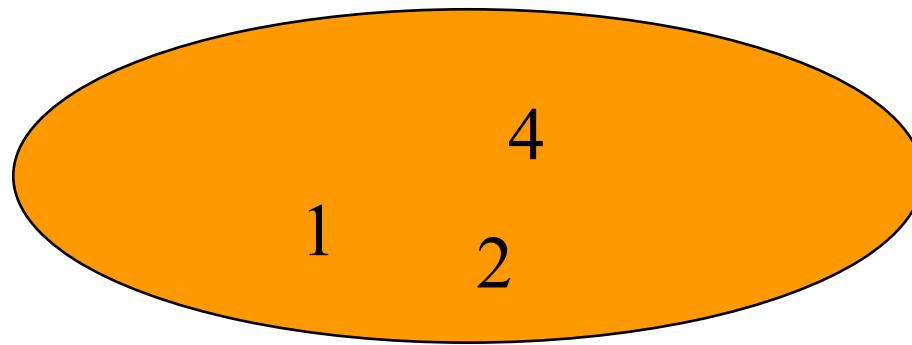


Max Priority
Queue

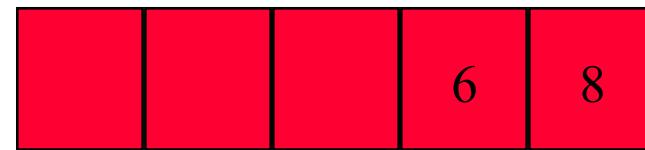


Sorted Array

After Second Remove Max Operation

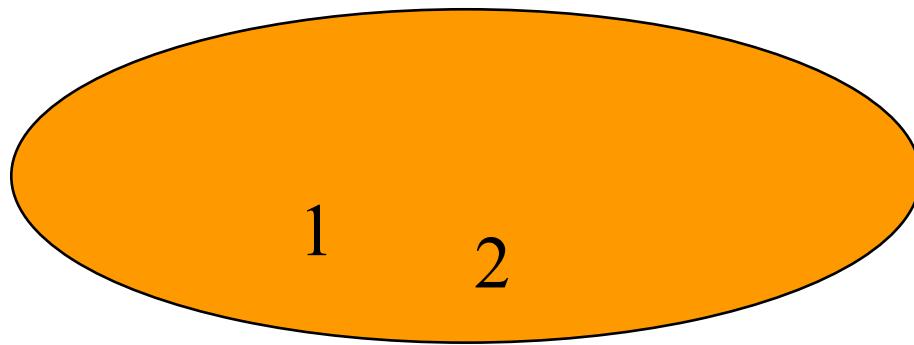


Max Priority
Queue

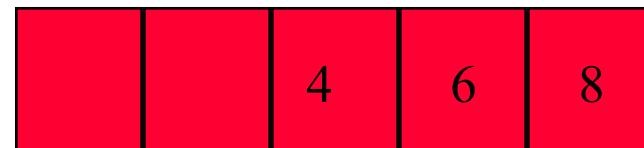


Sorted Array

After Third Remove Max Operation

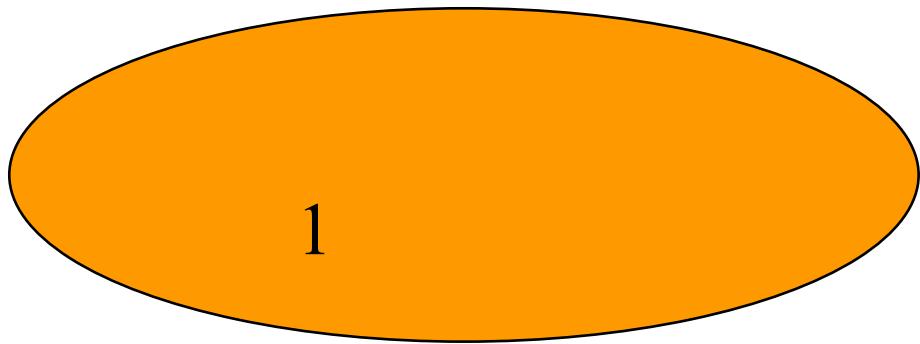


Max Priority
Queue

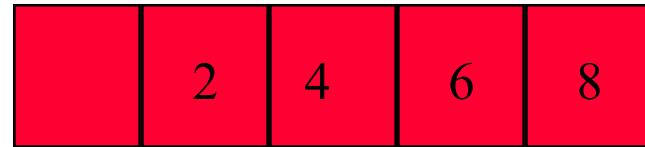


Sorted Array

After Fourth Remove Max Operation

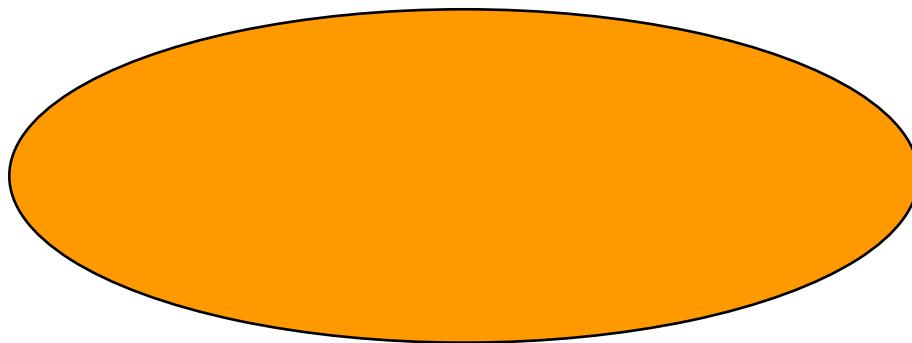


Max Priority
Queue



Sorted Array

After Fifth Remove Max Operation



Max Priority
Queue



Sorted Array

Complexity Of Sorting

Sort n elements.

- n put operations $\Rightarrow O(n \log n)$ time.
- n remove max operations $\Rightarrow O(n \log n)$ time.
- total time is $O(n \log n)$.
- compare with $O(n^2)$ for sort methods of Chapter 2.

Heap Sort

Uses a max priority queue that is implemented as a heap.

Initial put operations are replaced by a heap initialization step that takes $O(n)$ time.

Machine Scheduling

- m identical machines (drill press, cutter, sander, etc.)
- n jobs/tasks to be performed
- assign jobs to machines so that the time at which the last job completes is minimum

Machine Scheduling Example

3 machines and 7 jobs

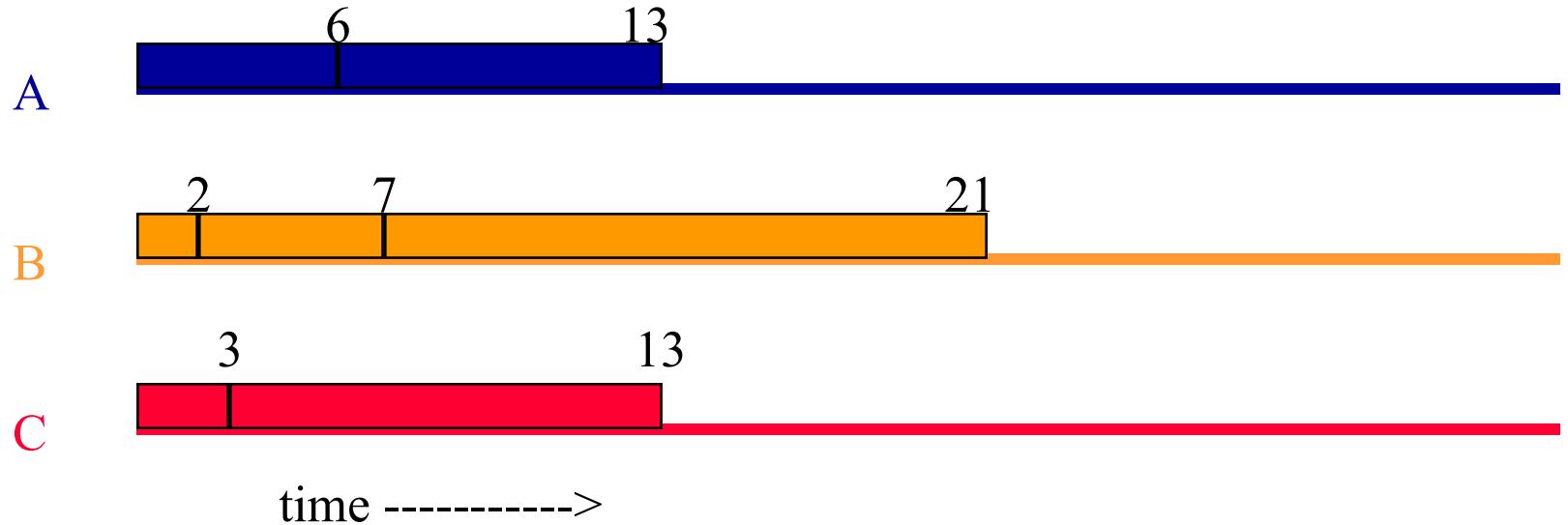
job times are [6, 2, 3, 5, 10, 7, 14]

possible schedule



time ----->

Machine Scheduling Example



Finish time = 21

Objective: Find schedules with minimum finish time.

LPT Schedules

Longest Processing Time first.

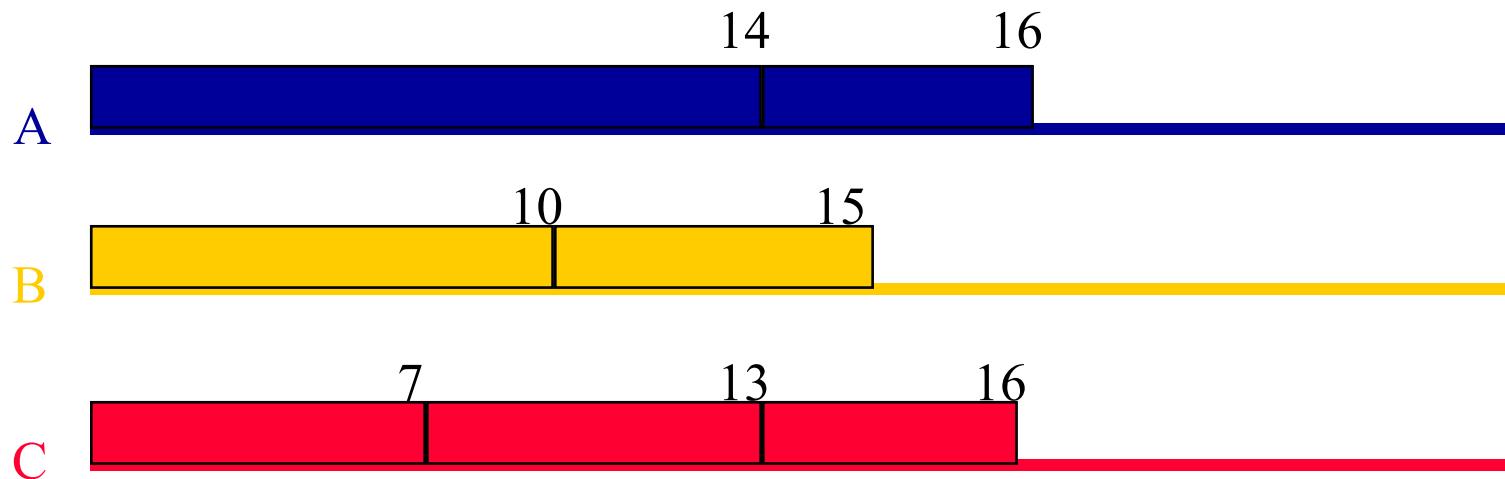
Jobs are scheduled in the order

14, 10, 7, 6, 5, 3, 2

Each job is scheduled on the machine
on which it finishes earliest.

LPT Schedule

[14, 10, 7, 6, 5, 3, 2]



Finish time is 16!

LPT Schedule

- LPT rule does not guarantee minimum finish time schedules.
- $(\text{LPT Finish Time})/(\text{Minimum Finish Time}) \leq 4/3 - 1/(3m)$ where **m** is number of machines.
- Usually LPT finish time is much closer to minimum finish time.
- Minimum finish time scheduling is NP-hard.

NP-hard Problems

- Infamous class of problems for which no one has developed a polynomial time algorithm.
- That is, no algorithm whose complexity is $O(n^k)$ for any constant k is known for any NP-hard problem.
- The class includes thousands of real-world problems.
- Highly unlikely that any NP-hard problem can be solved by a polynomial time algorithm.

NP-hard Problems

- Since even polynomial time algorithms with degree $k > 3$ (say) are not practical for large n , we must change our expectations of the algorithm that is used.
- Usually develop fast heuristics for NP-hard problems.
 - Algorithm that gives a solution close to best.
 - Runs in acceptable amount of time.
- LPT rule is good heuristic for minimum finish time scheduling.

Complexity Of LPT Scheduling

- Sort jobs into decreasing order of task time.
 - $O(n \log n)$ time (n is number of jobs)
- Schedule jobs in this order.
 - assign job to machine that becomes available first
 - must find minimum of m (m is number of machines) finish times
 - takes $O(m)$ time using simple strategy
 - so need $O(mn)$ time to schedule all n jobs.

Using A Min Priority Queue

- Min priority queue has the finish times of the **m** machines.
- Initial finish times are all **0**.
- To schedule a job remove machine with minimum finish time from the priority queue.
- Update the finish time of the selected machine and put the machine back into the priority queue.

Using A Min Priority Queue

- m put operations to initialize priority queue
- 1 remove min and 1 put to schedule each job
- each put and remove min operation takes $O(\log m)$ time
- time to schedule is $O(n \log m)$
- overall time is
$$O(n \log n + n \log m) = O(n \log (mn))$$

Huffman Codes

Useful in lossless compression.

May be used in conjunction with LZW method.

Read from text.

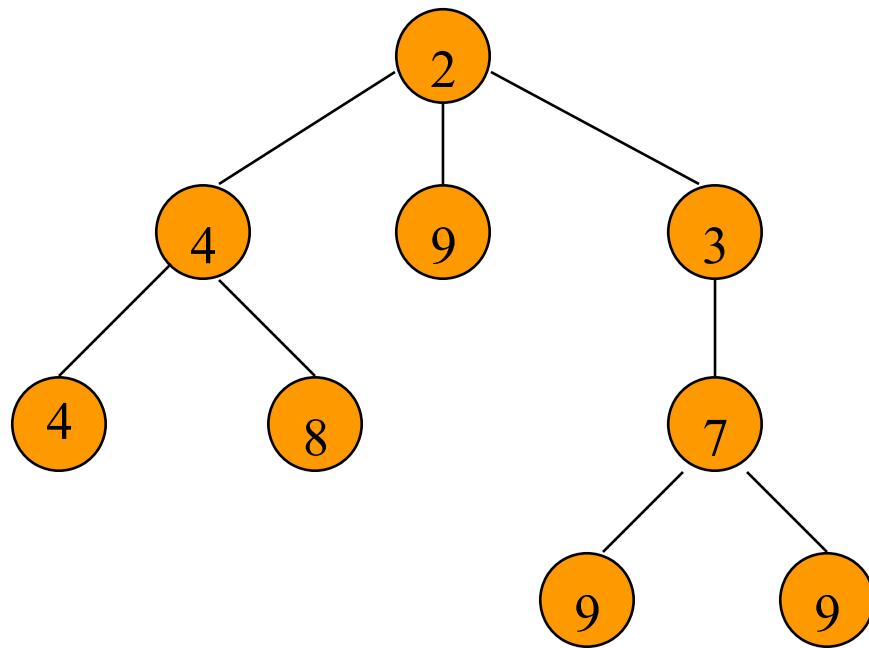
Min Tree Definition

Each tree node has a value.

Value in any node is the minimum value in the subtree for which that node is the root.

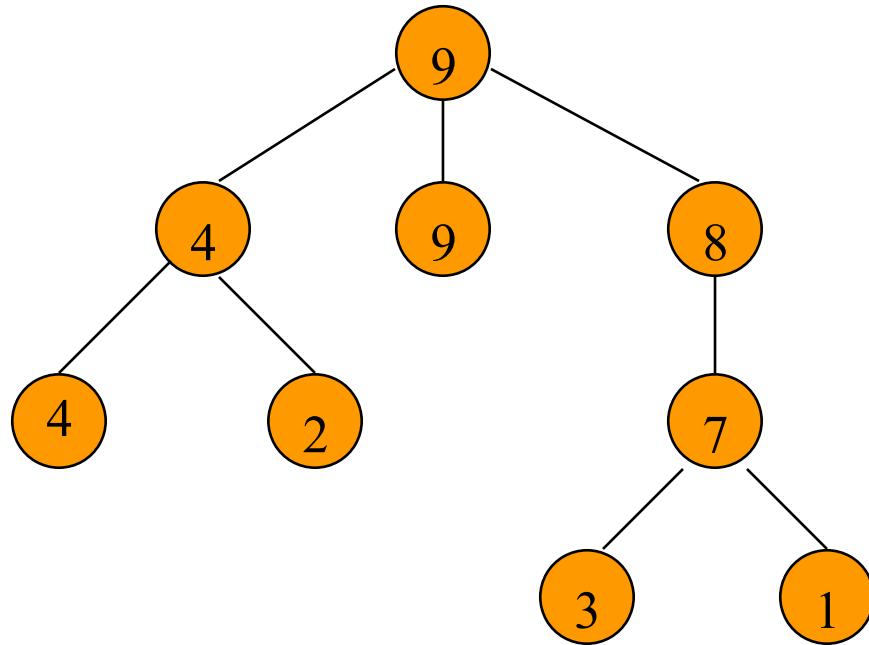
Equivalently, no descendent has a smaller value.

Min Tree Example



Root has minimum element.

Max Tree Example

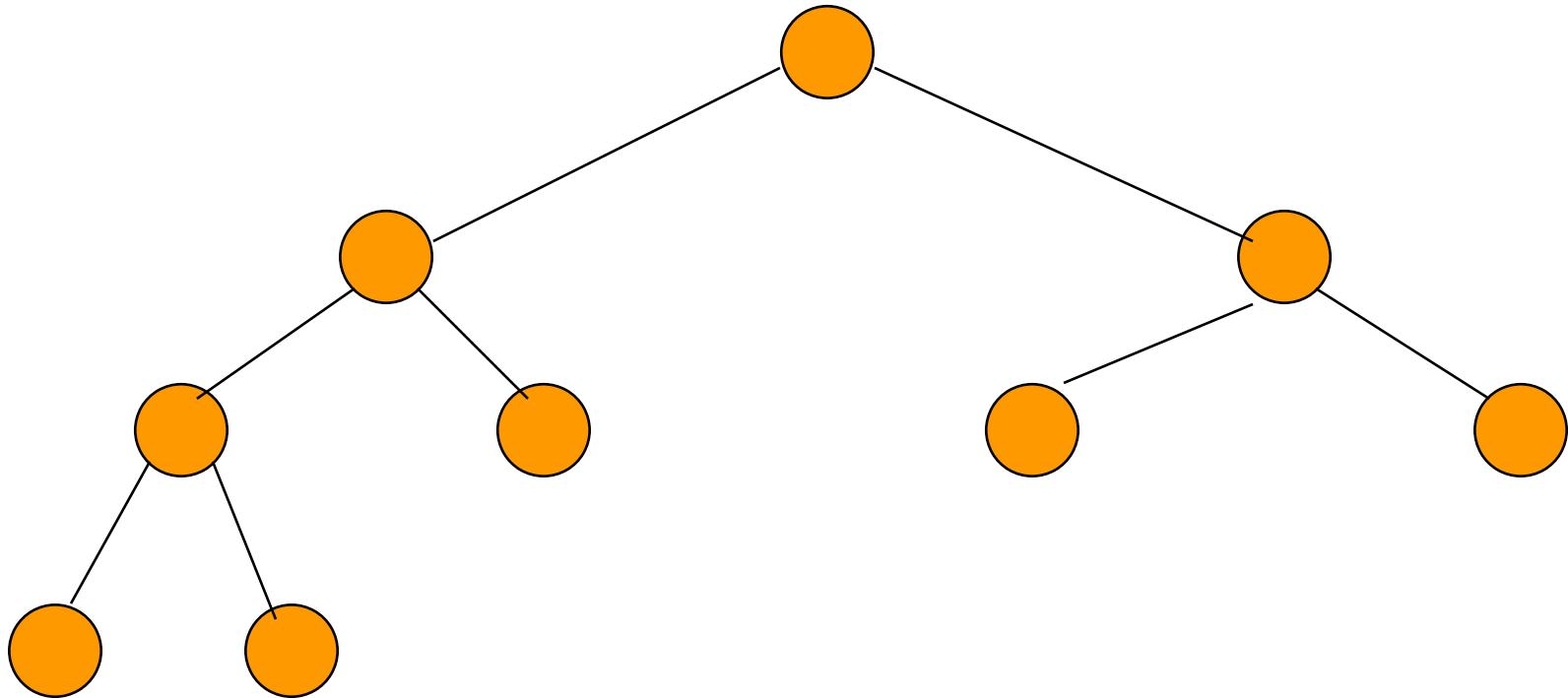


Root has maximum element.

Min Heap Definition

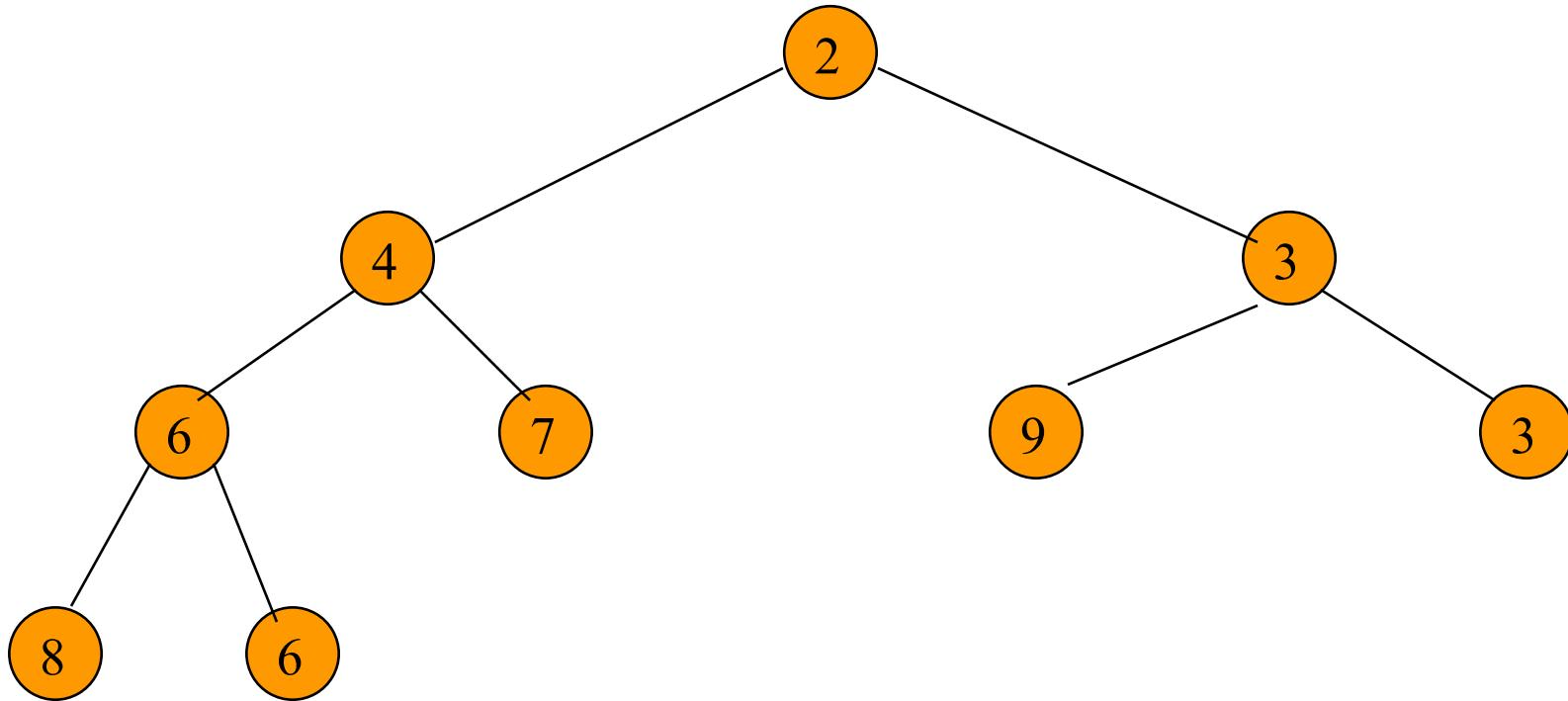
- complete binary tree
- min tree

Min Heap With 9 Nodes



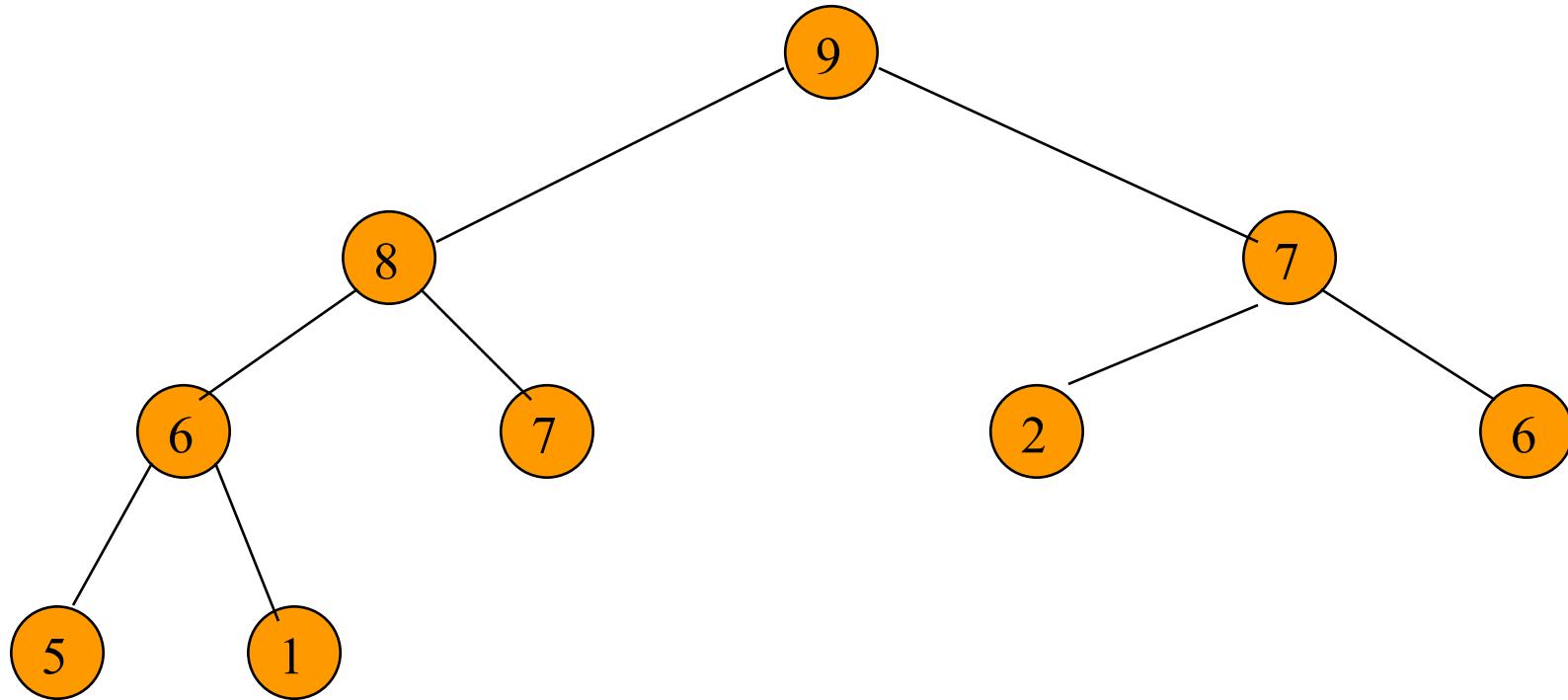
Complete binary tree with 9 nodes.

Min Heap With 9 Nodes



Complete binary tree with 9 nodes
that is also a min tree.

Max Heap With 9 Nodes

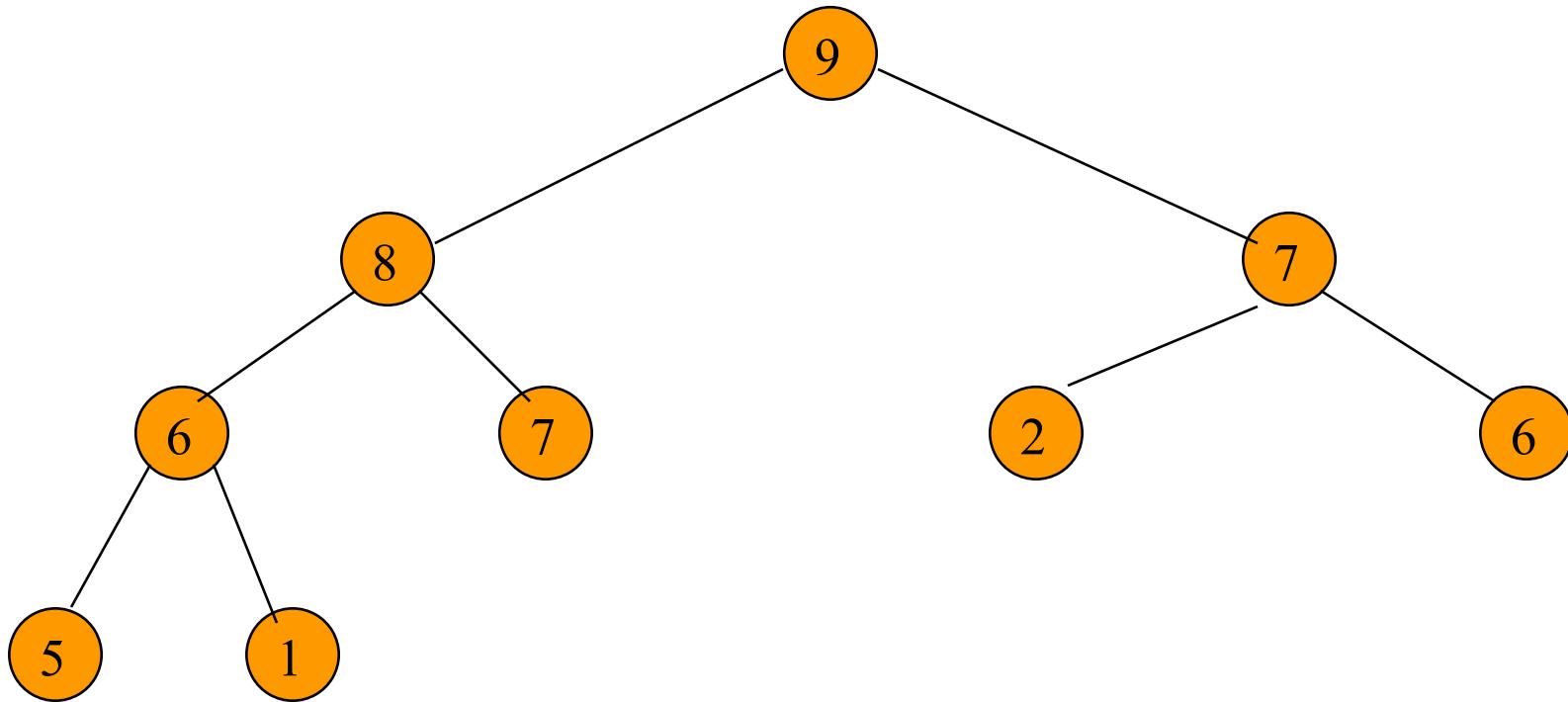


Complete binary tree with 9 nodes
that is also a max tree.

Heap Height

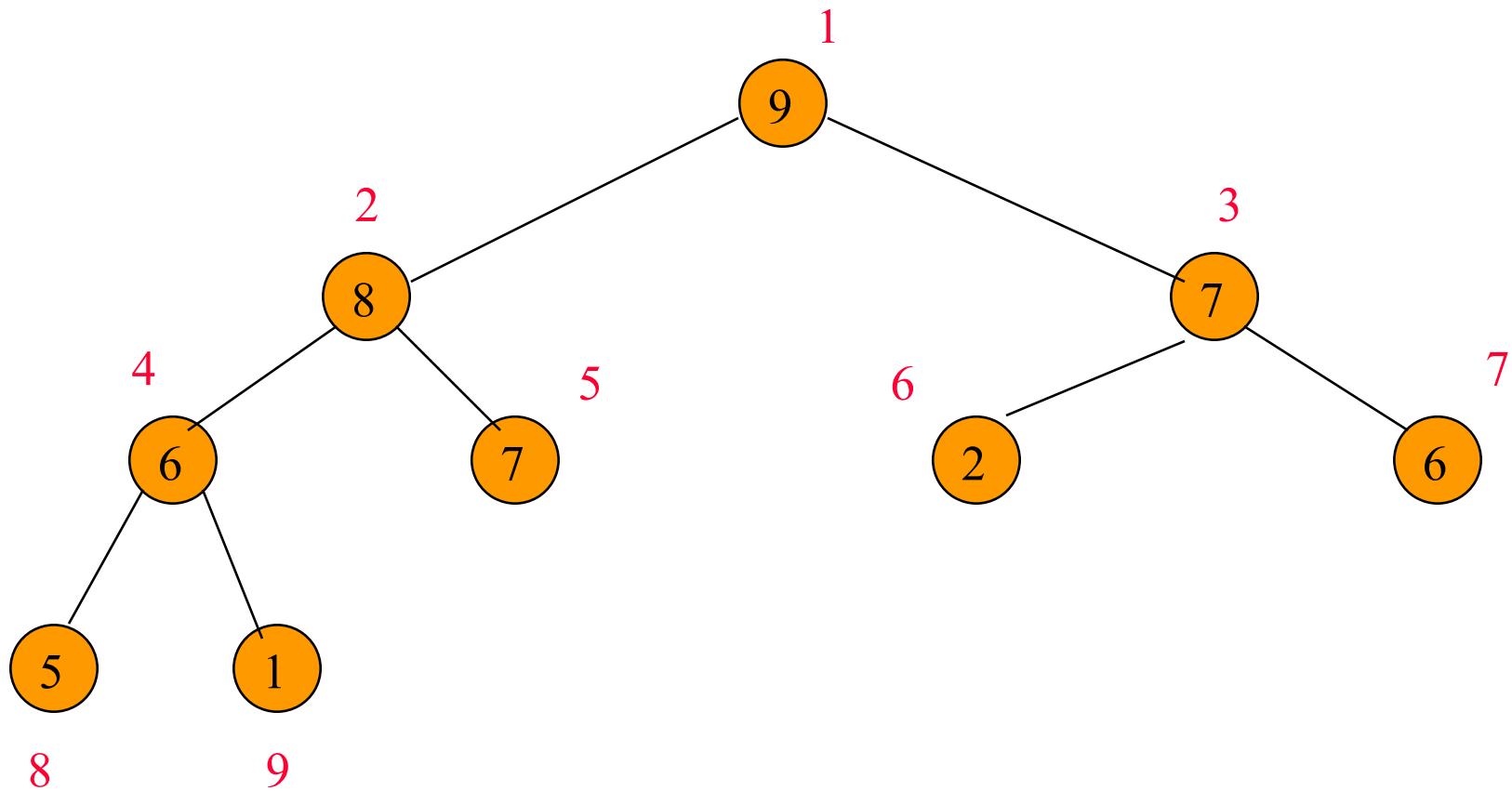
Since a heap is a complete binary tree, the height of an n node heap is $\log_2(n+1)$.

A Heap Is Efficiently Represented As An Array

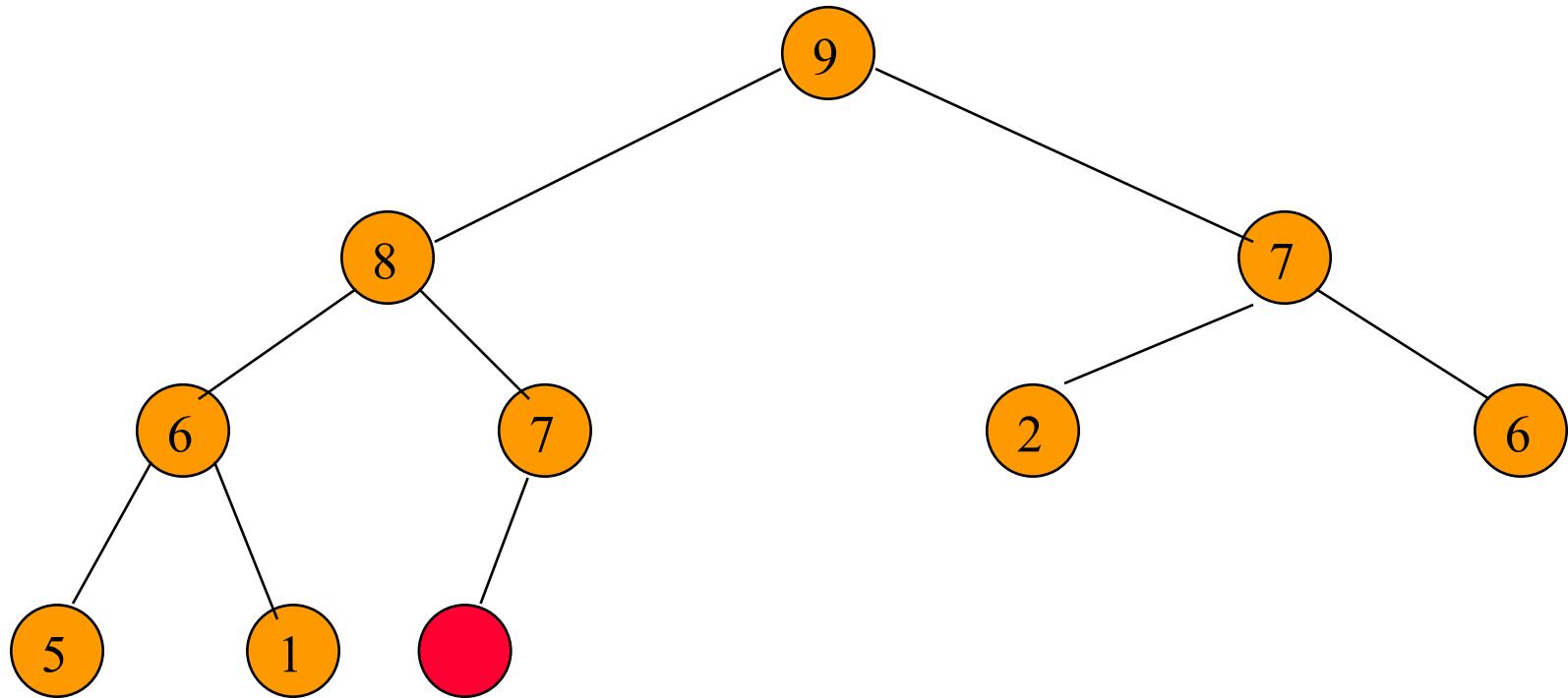


0 1 2 3 4 5 6 7 8 9 10

Moving Up And Down A Heap

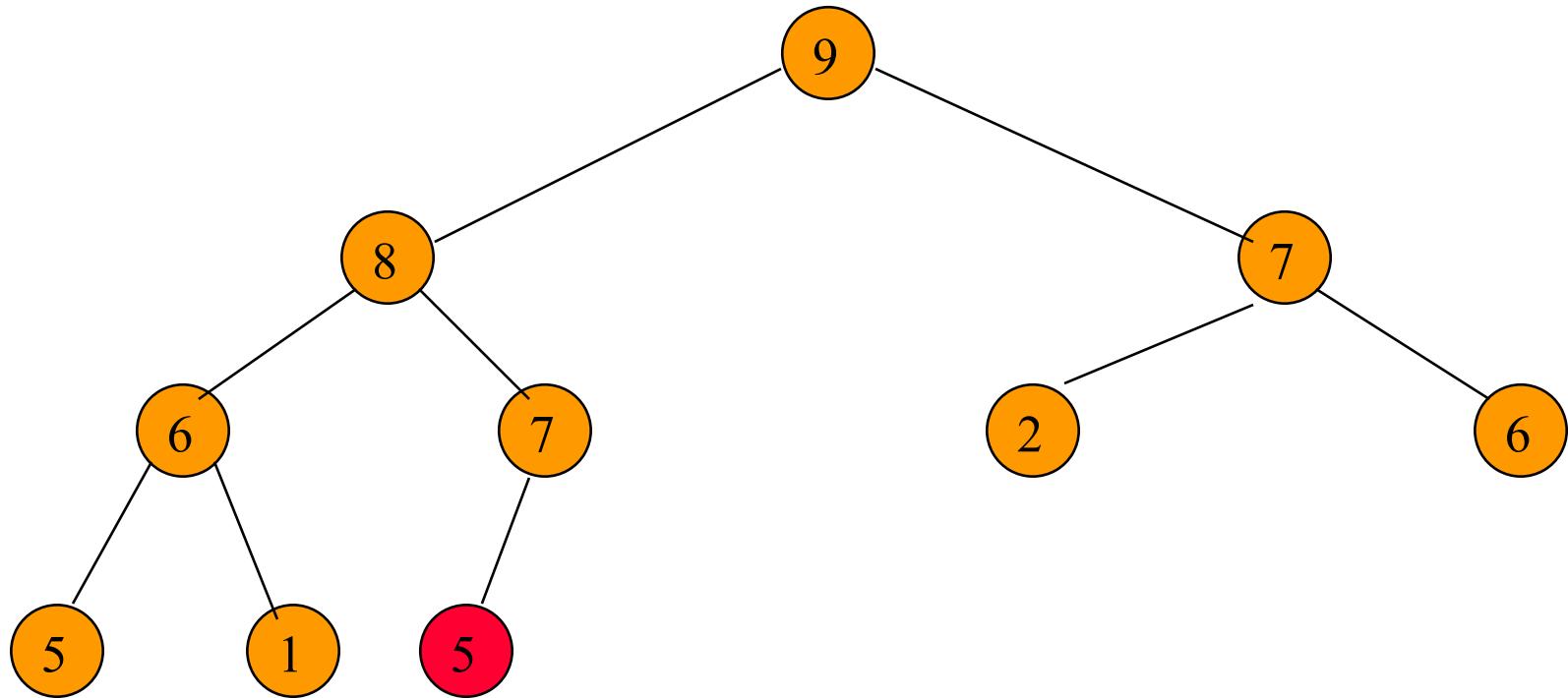


Putting An Element Into A Max Heap



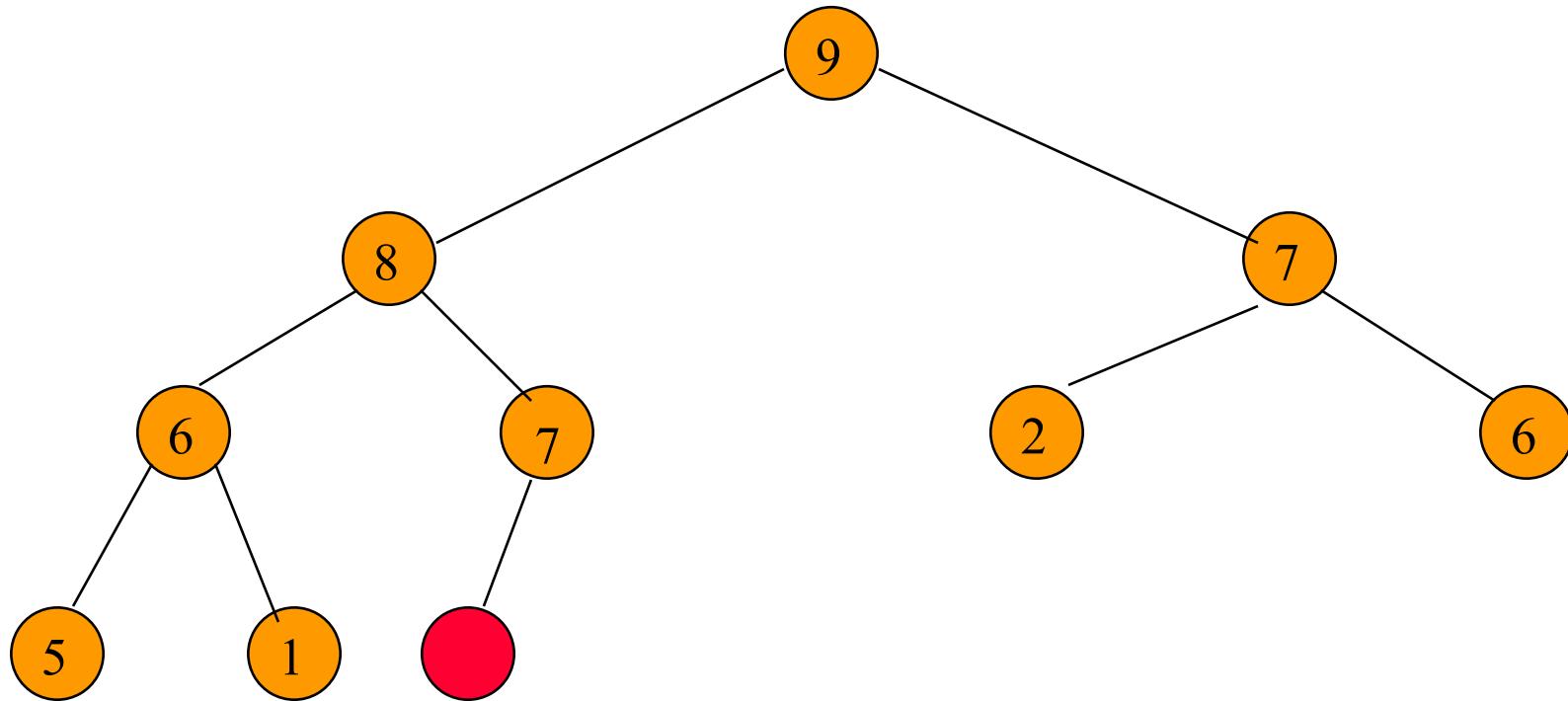
Complete binary tree with 10 nodes.

Putting An Element Into A Max Heap



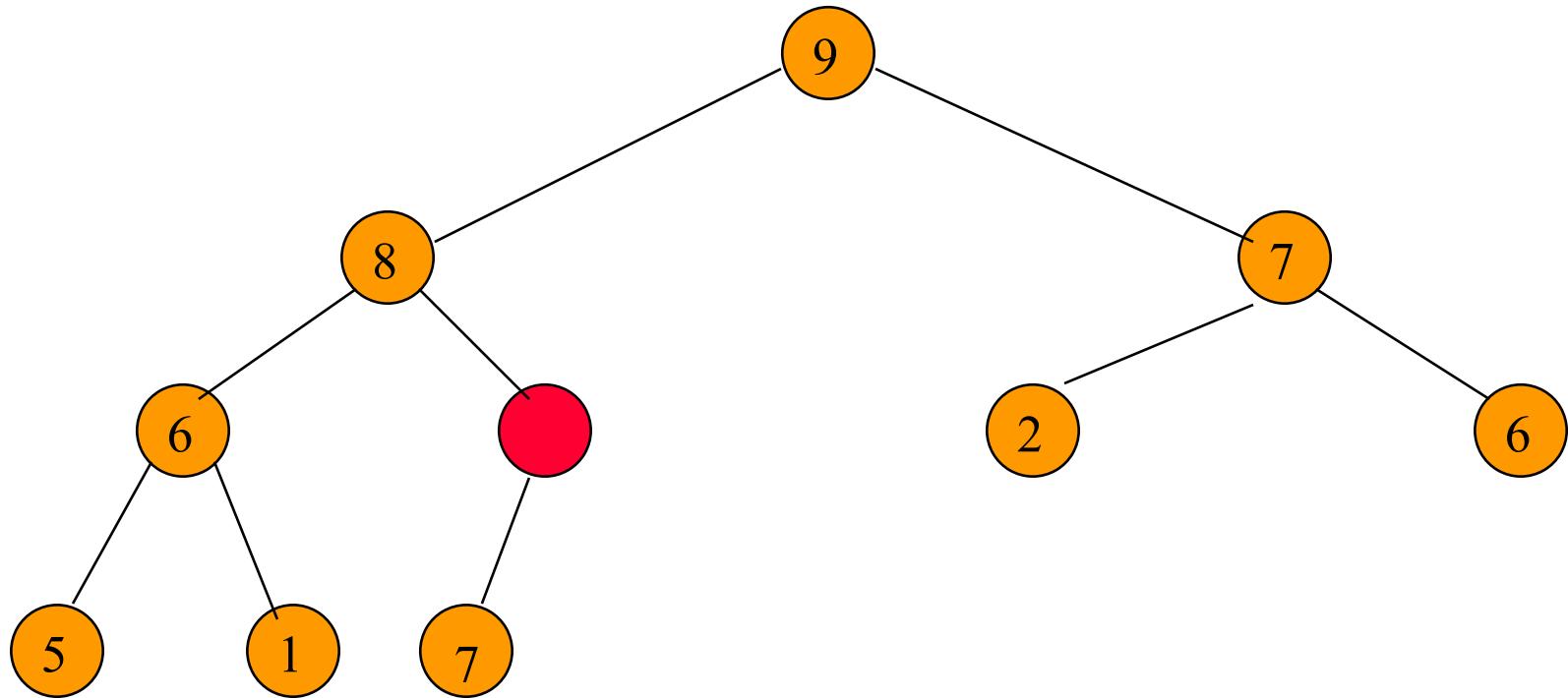
New element is 5.

Putting An Element Into A Max Heap



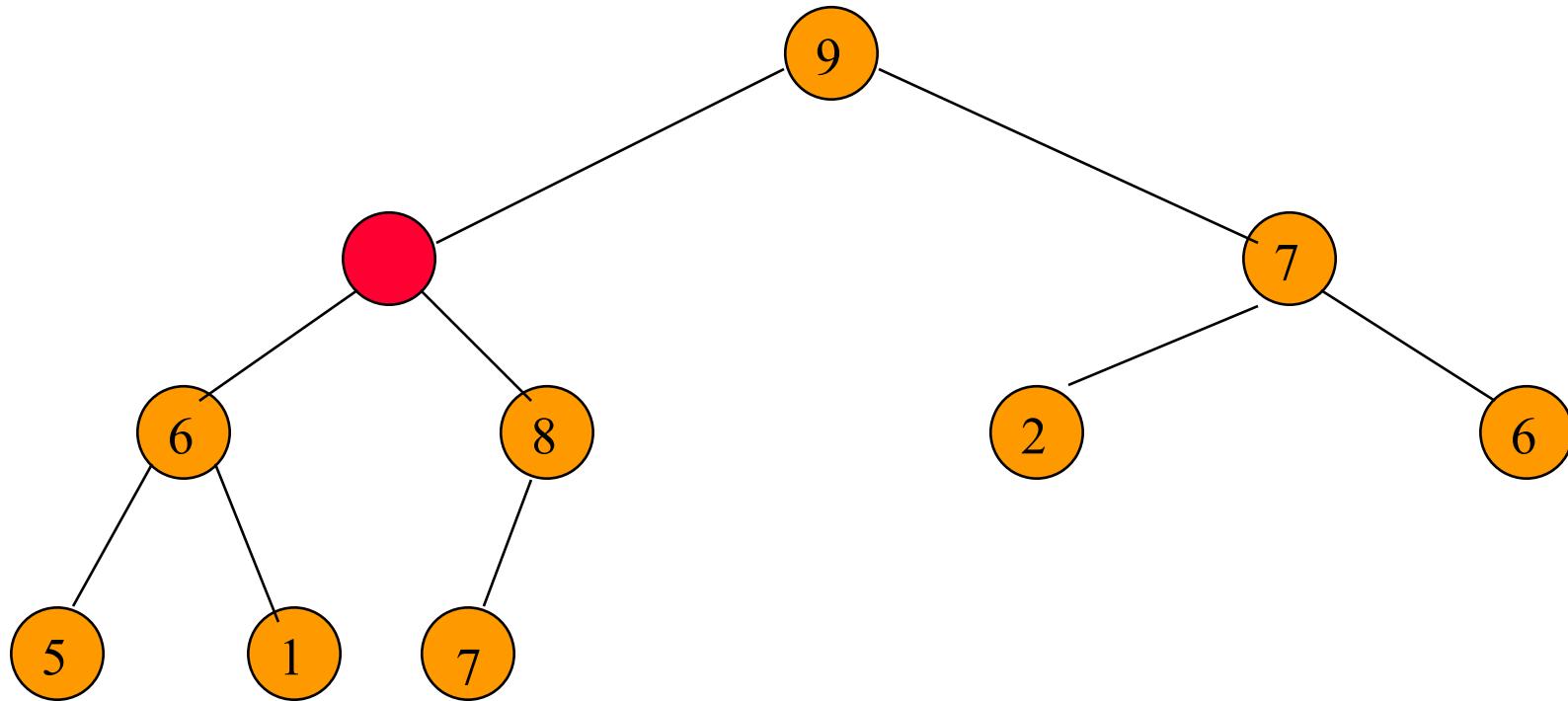
New element is 20.

Putting An Element Into A Max Heap



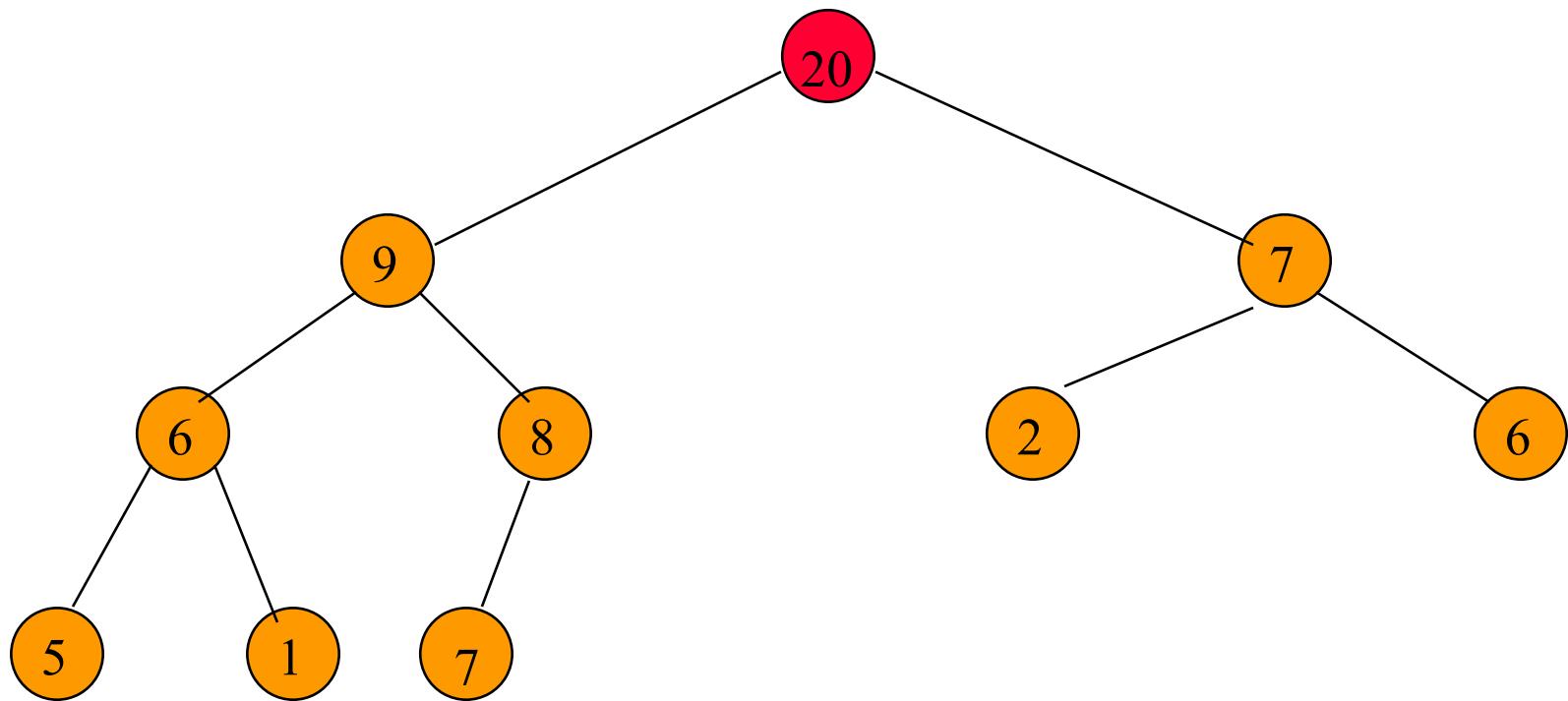
New element is 20.

Putting An Element Into A Max Heap



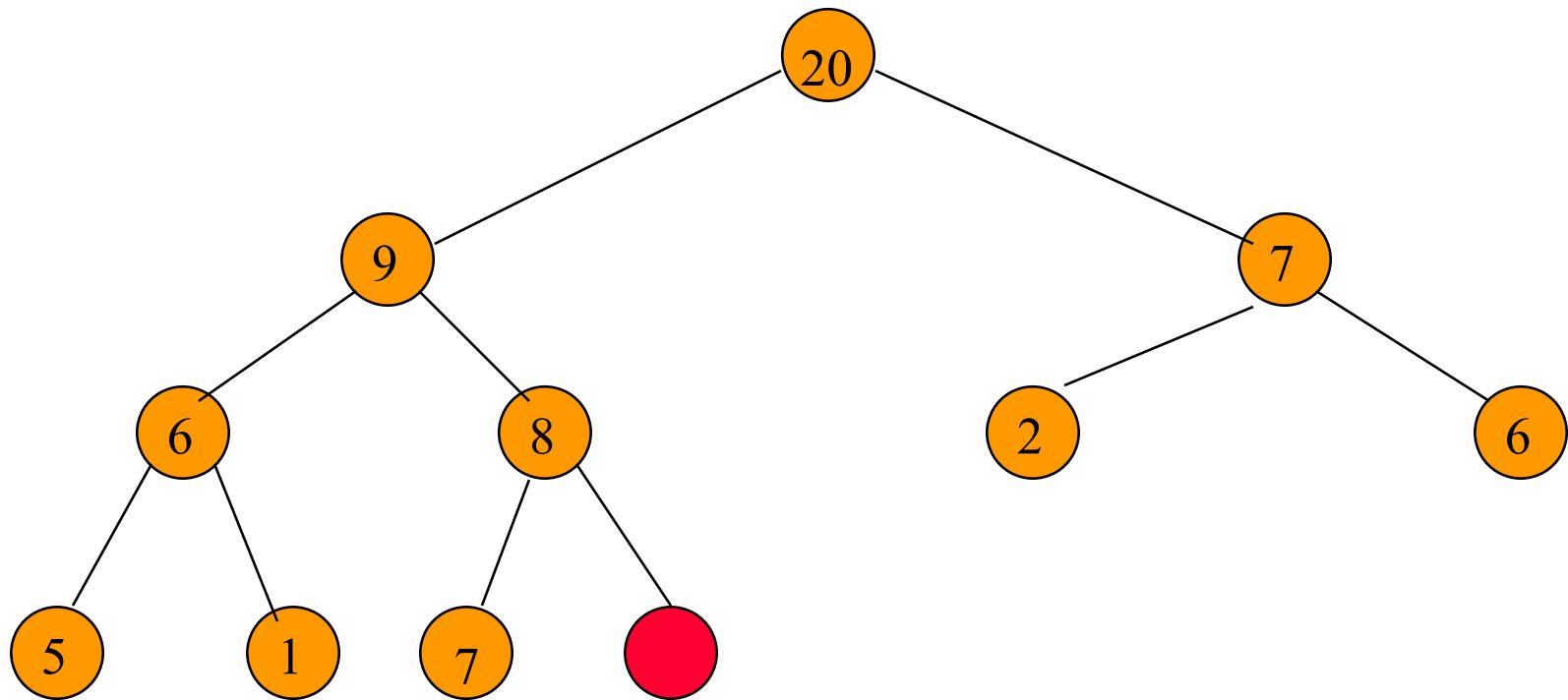
New element is 20.

Putting An Element Into A Max Heap



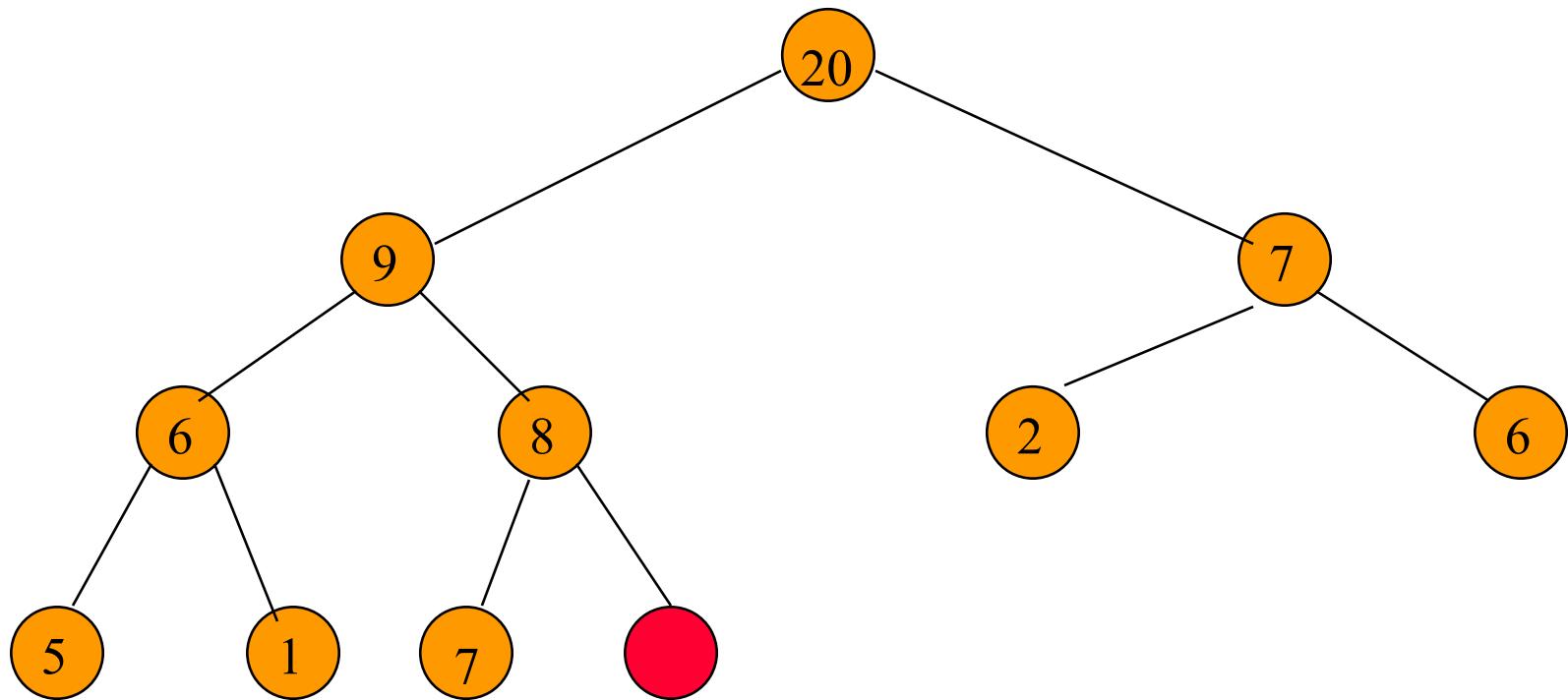
New element is 20.

Putting An Element Into A Max Heap



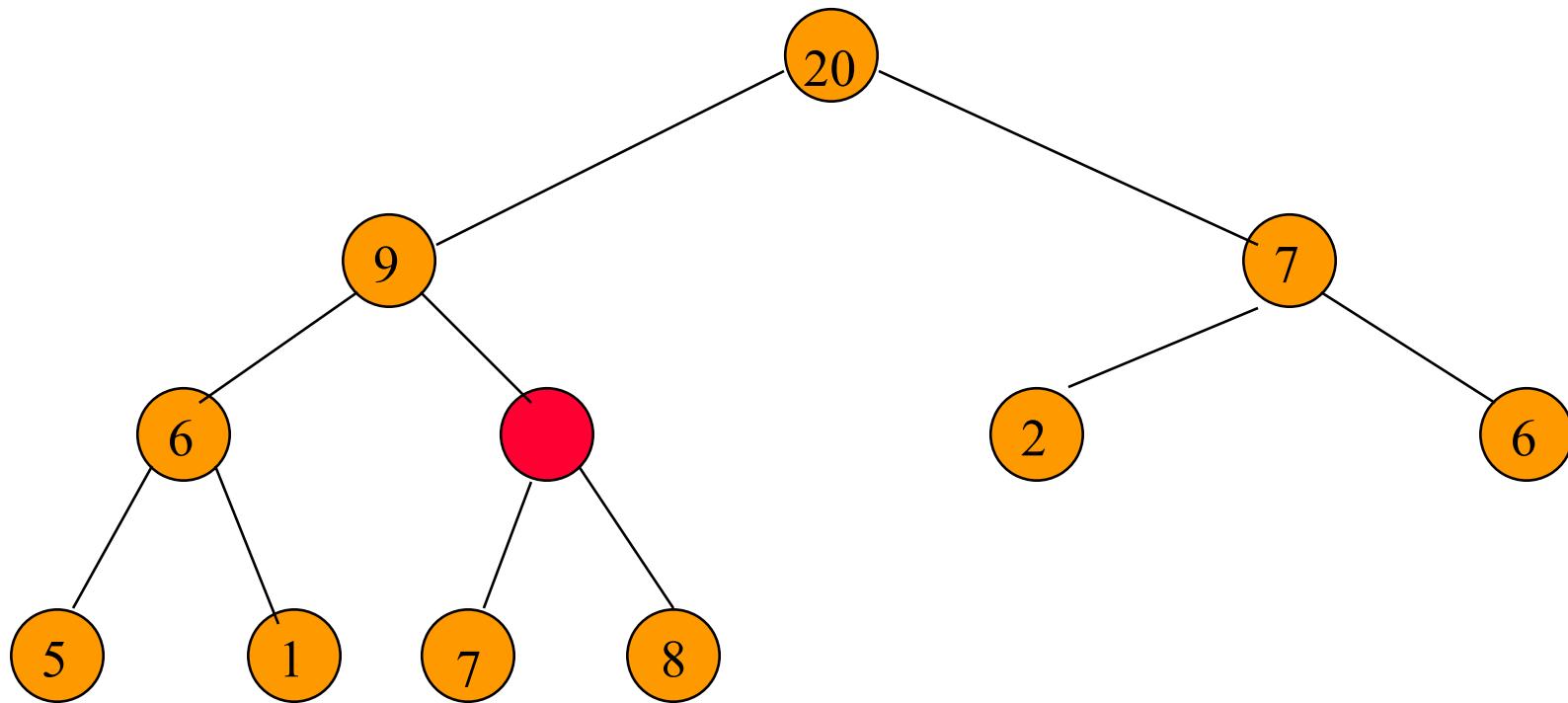
Complete binary tree with 11 nodes.

Putting An Element Into A Max Heap



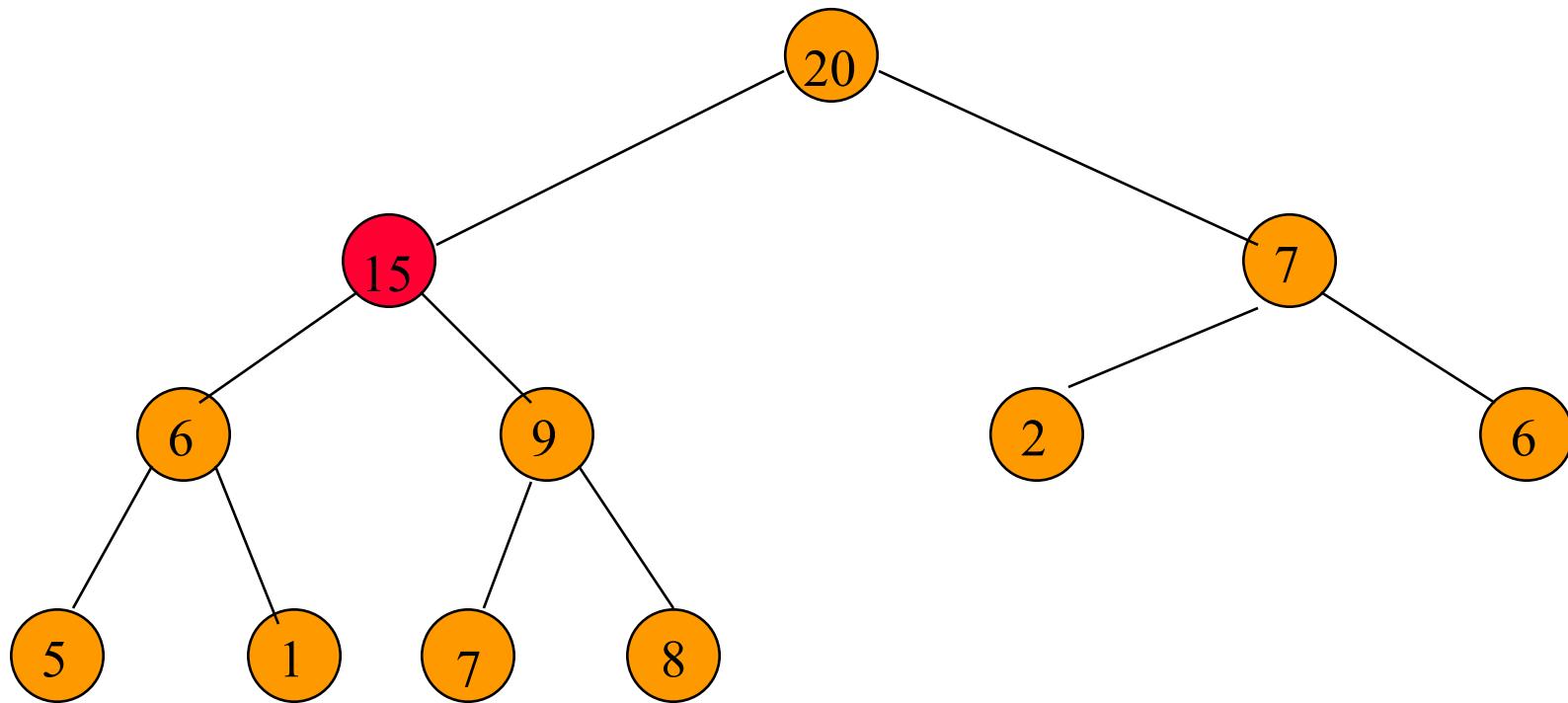
New element is 15.

Putting An Element Into A Max Heap



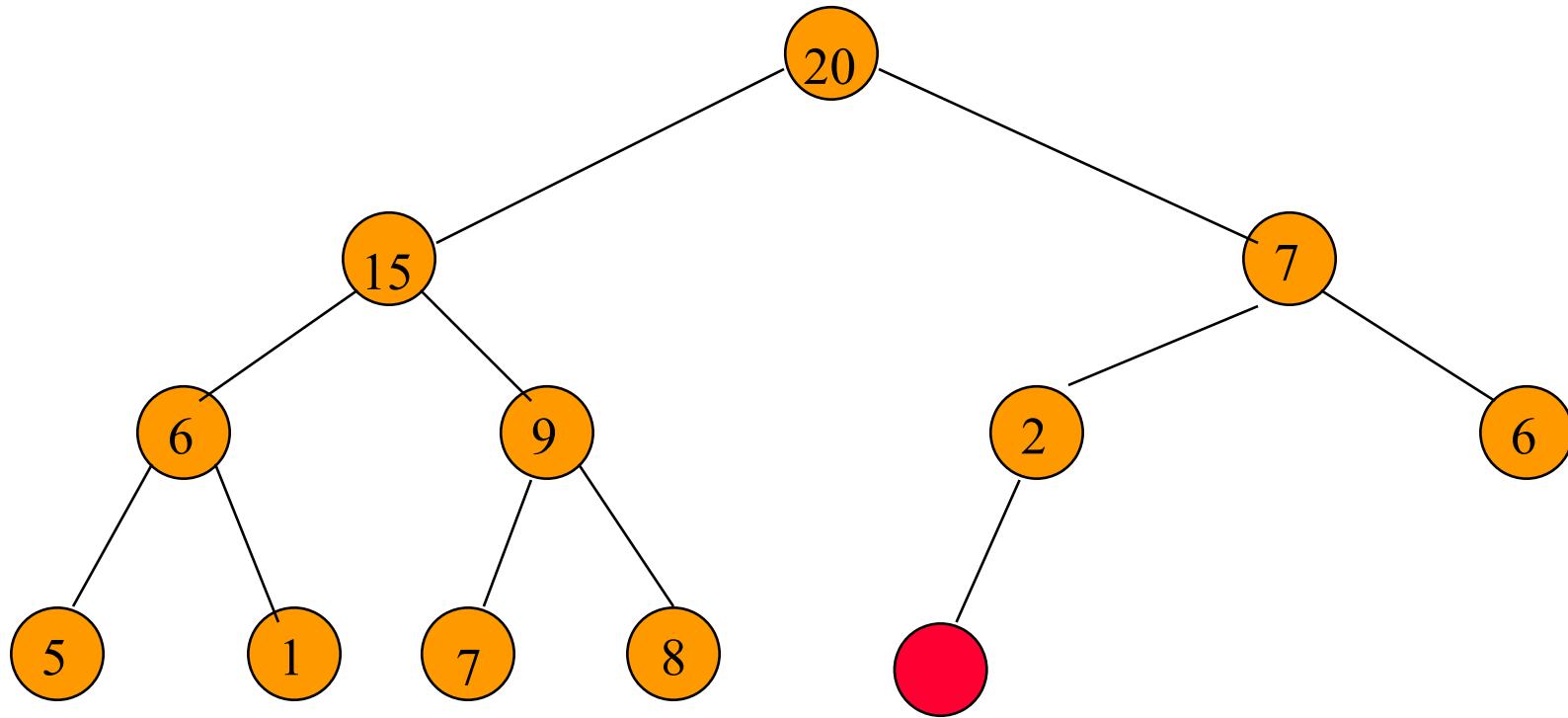
New element is 15.

Putting An Element Into A Max Heap



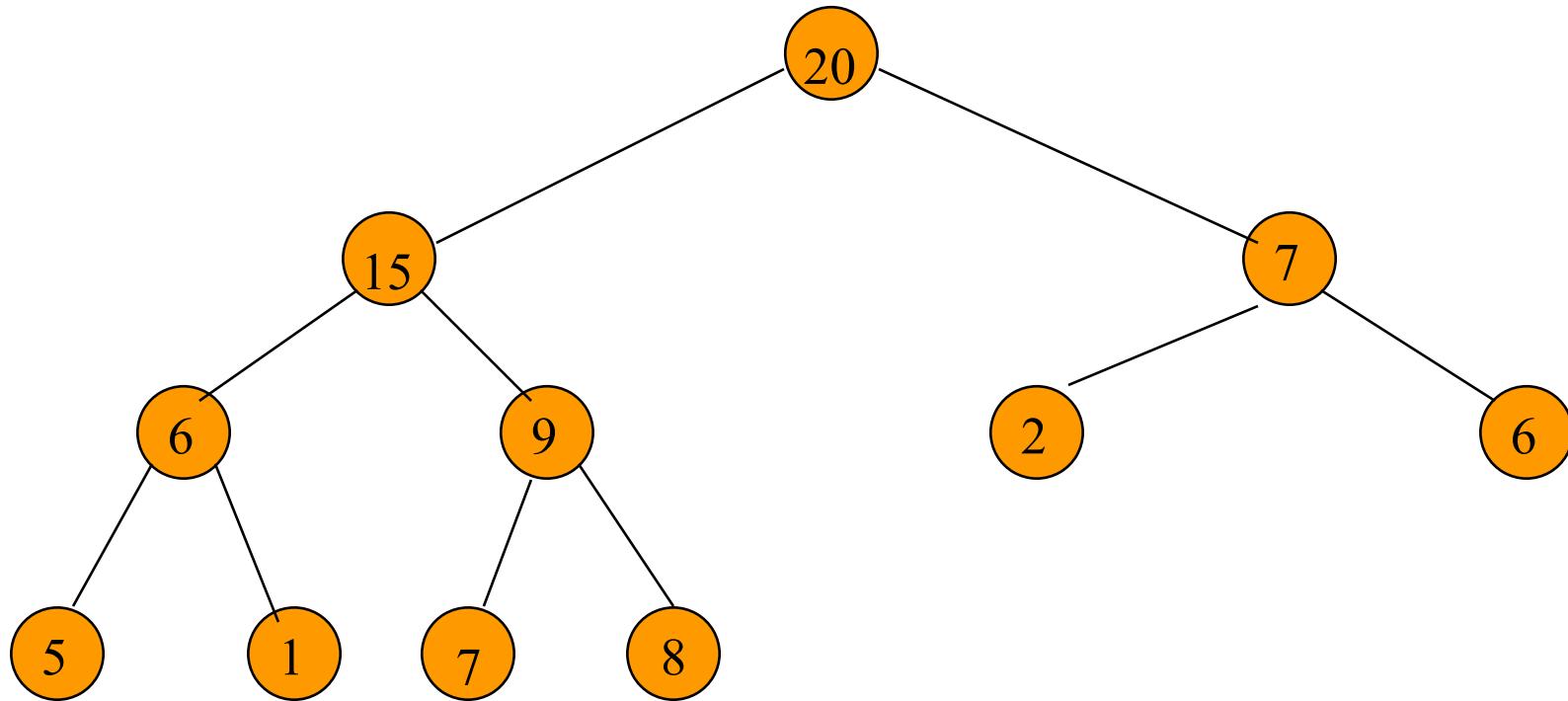
New element is 15.

Complexity Of Put



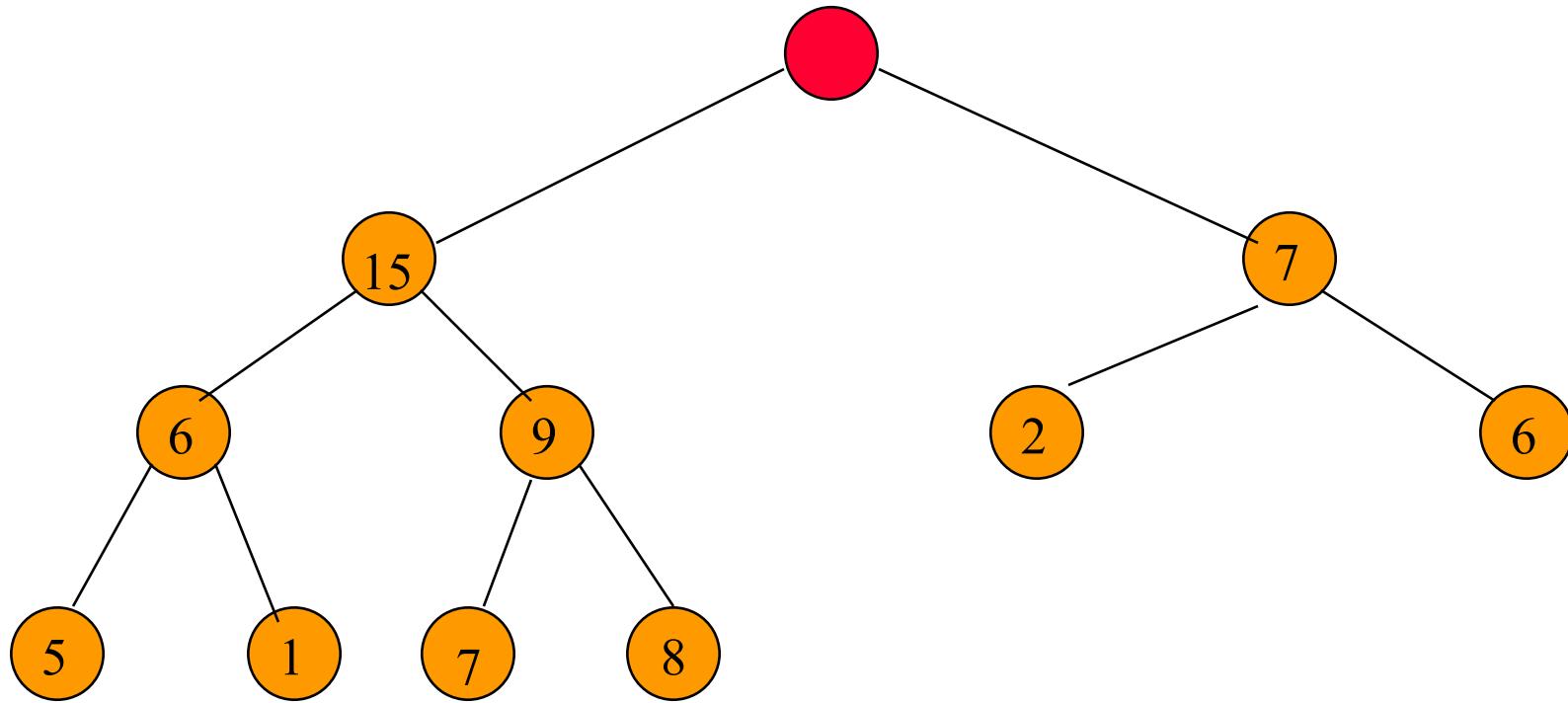
Complexity is $O(\log n)$, where n is
heap size.

Removing The Max Element



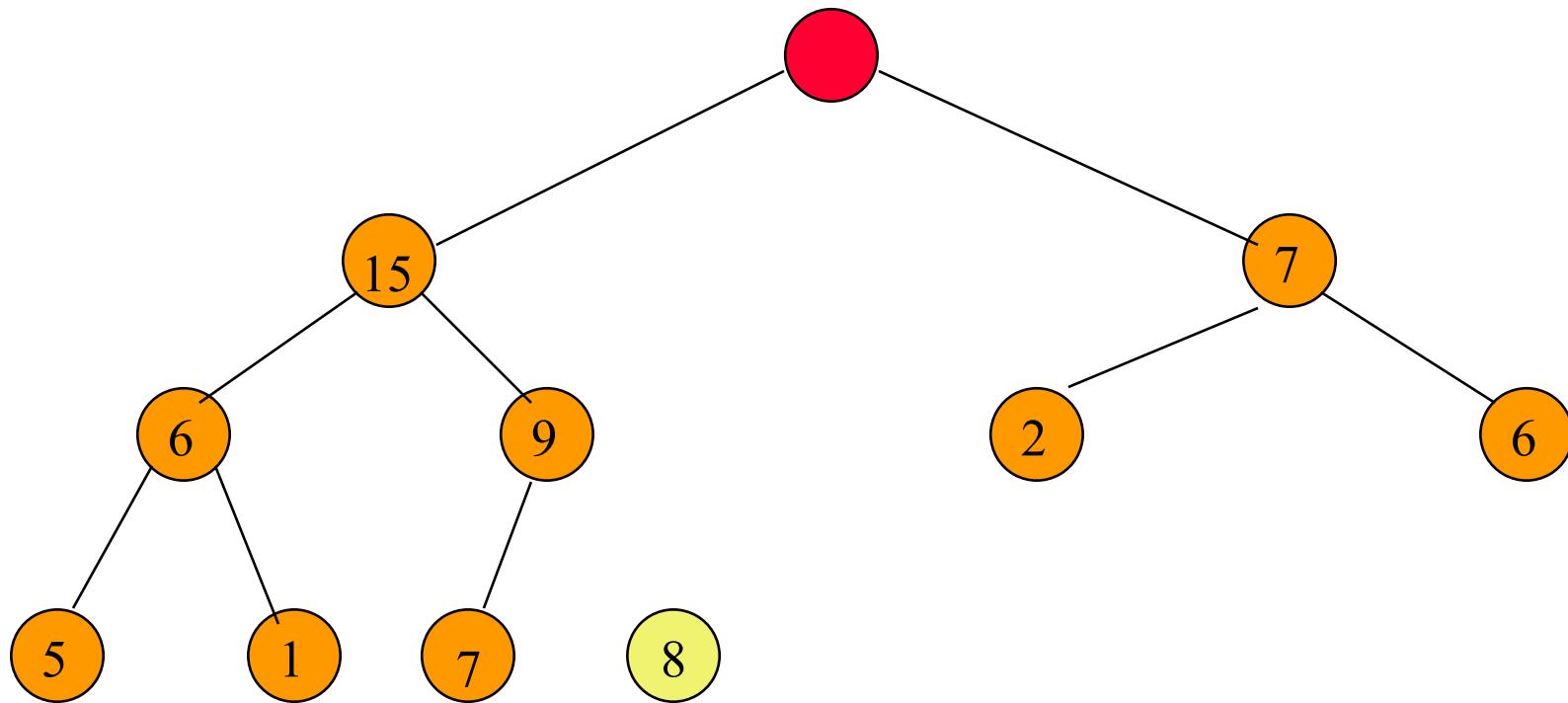
Max element is in the root.

Removing The Max Element



After max element is removed.

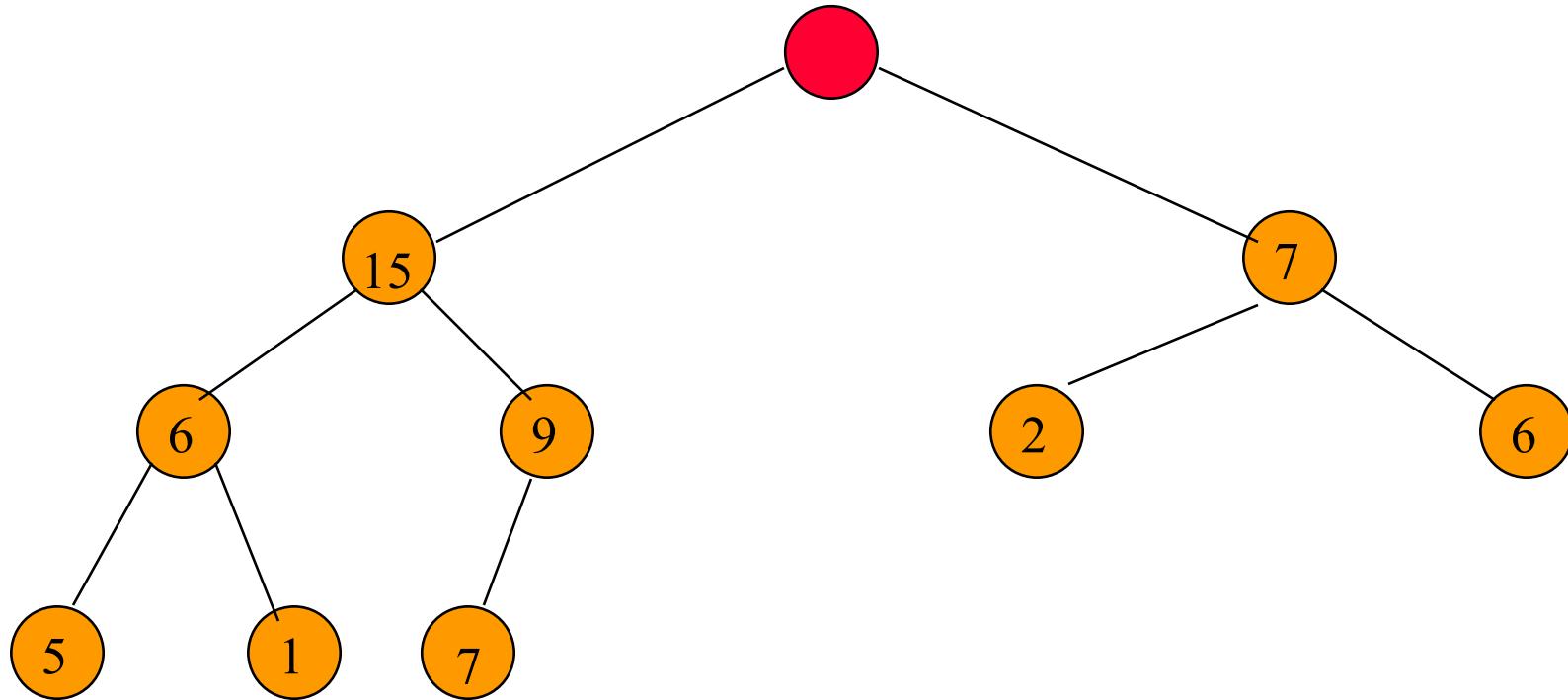
Removing The Max Element



Heap with 10 nodes.

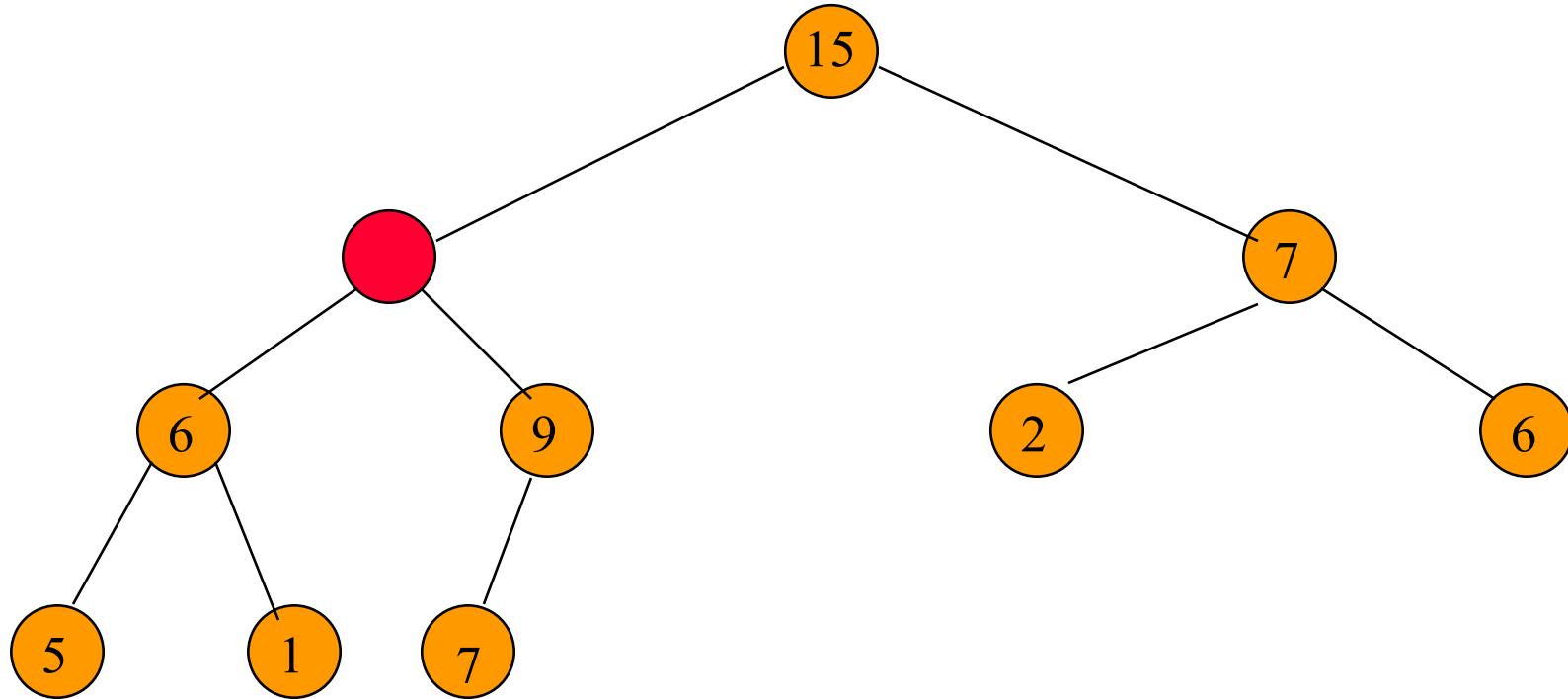
Reinsert 8 into the heap.

Removing The Max Element



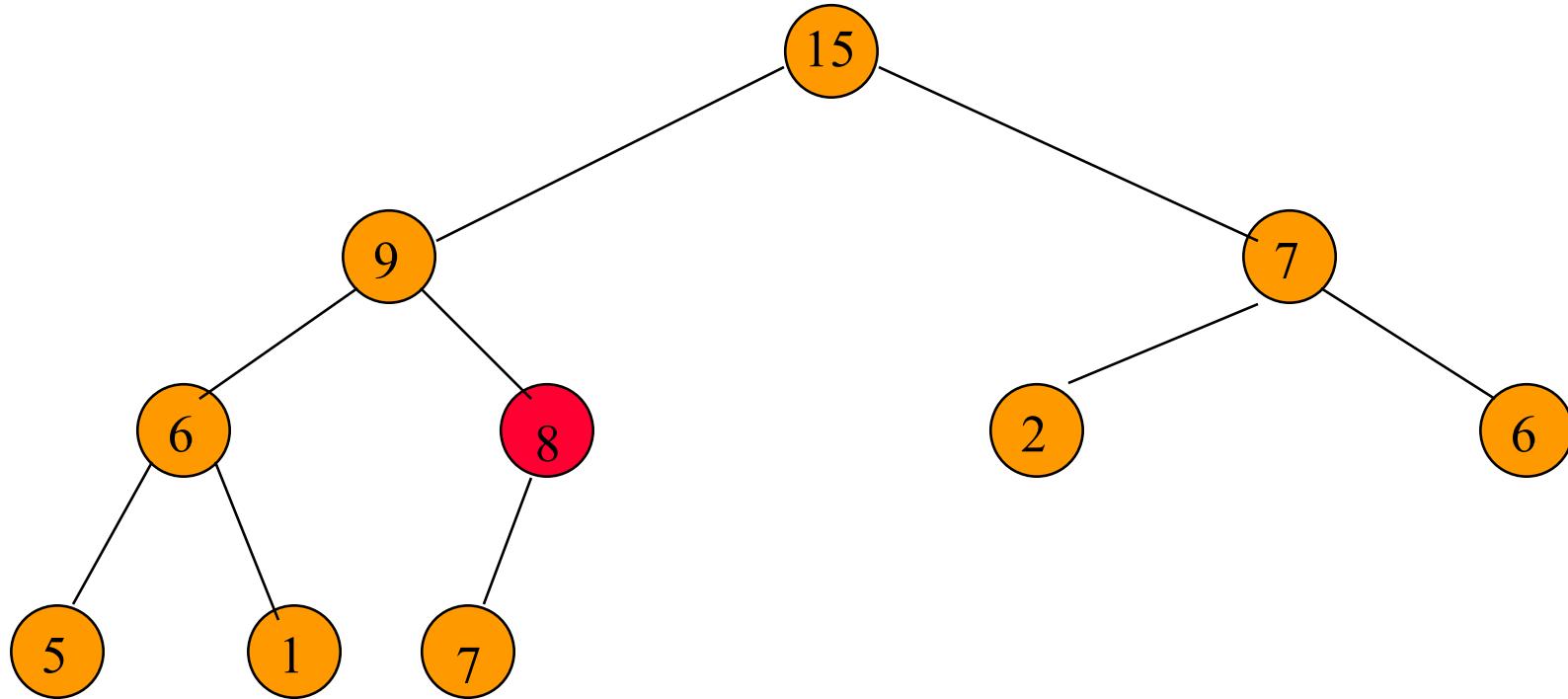
Reinsert 8 into the heap.

Removing The Max Element



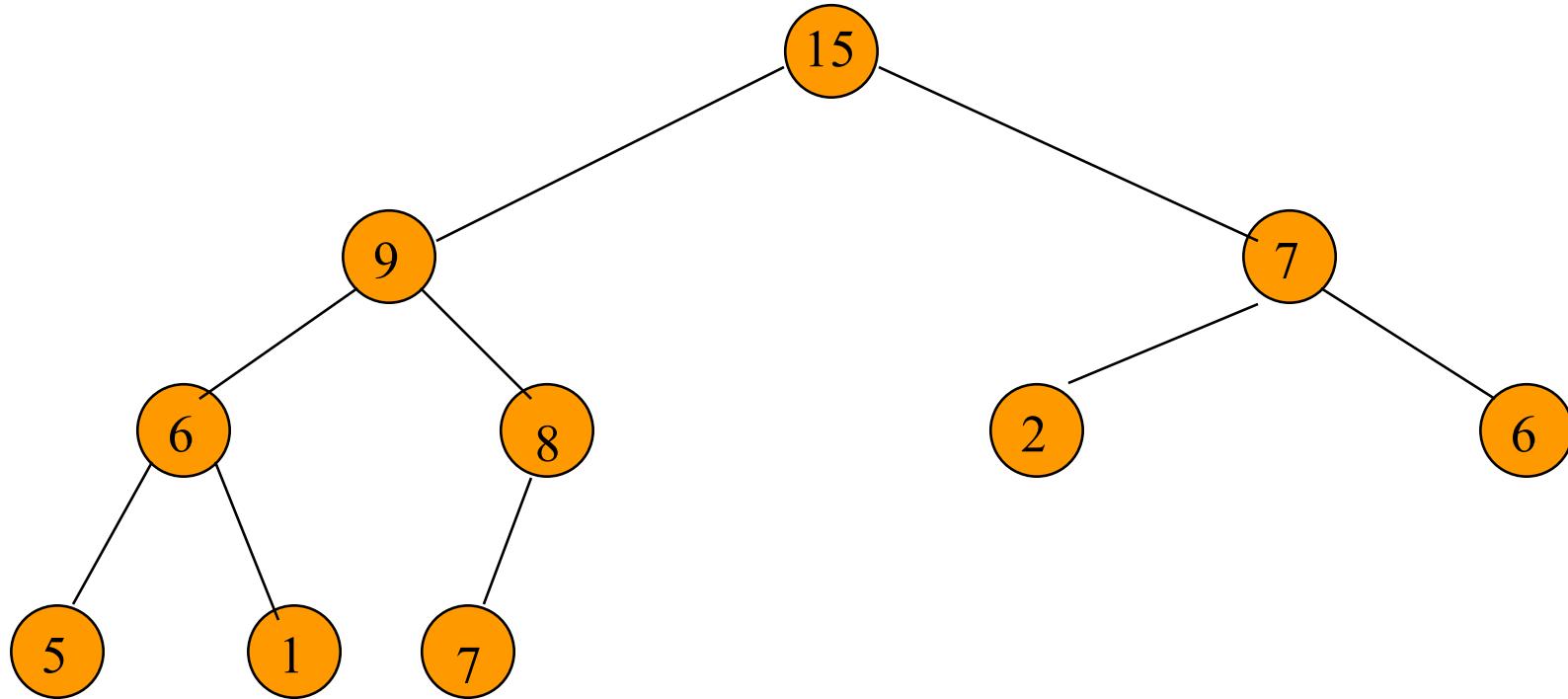
Reinsert 8 into the heap.

Removing The Max Element



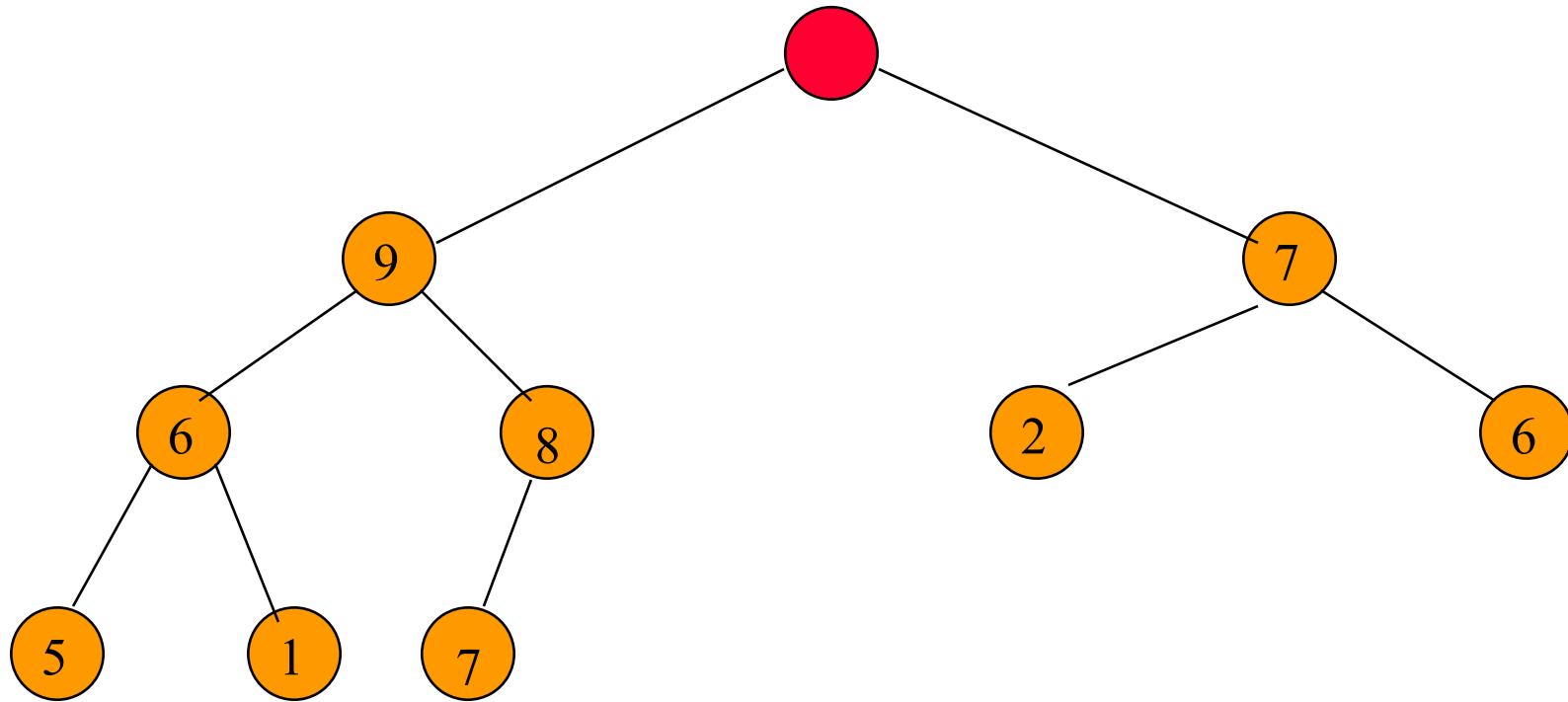
Reinsert 8 into the heap.

Removing The Max Element



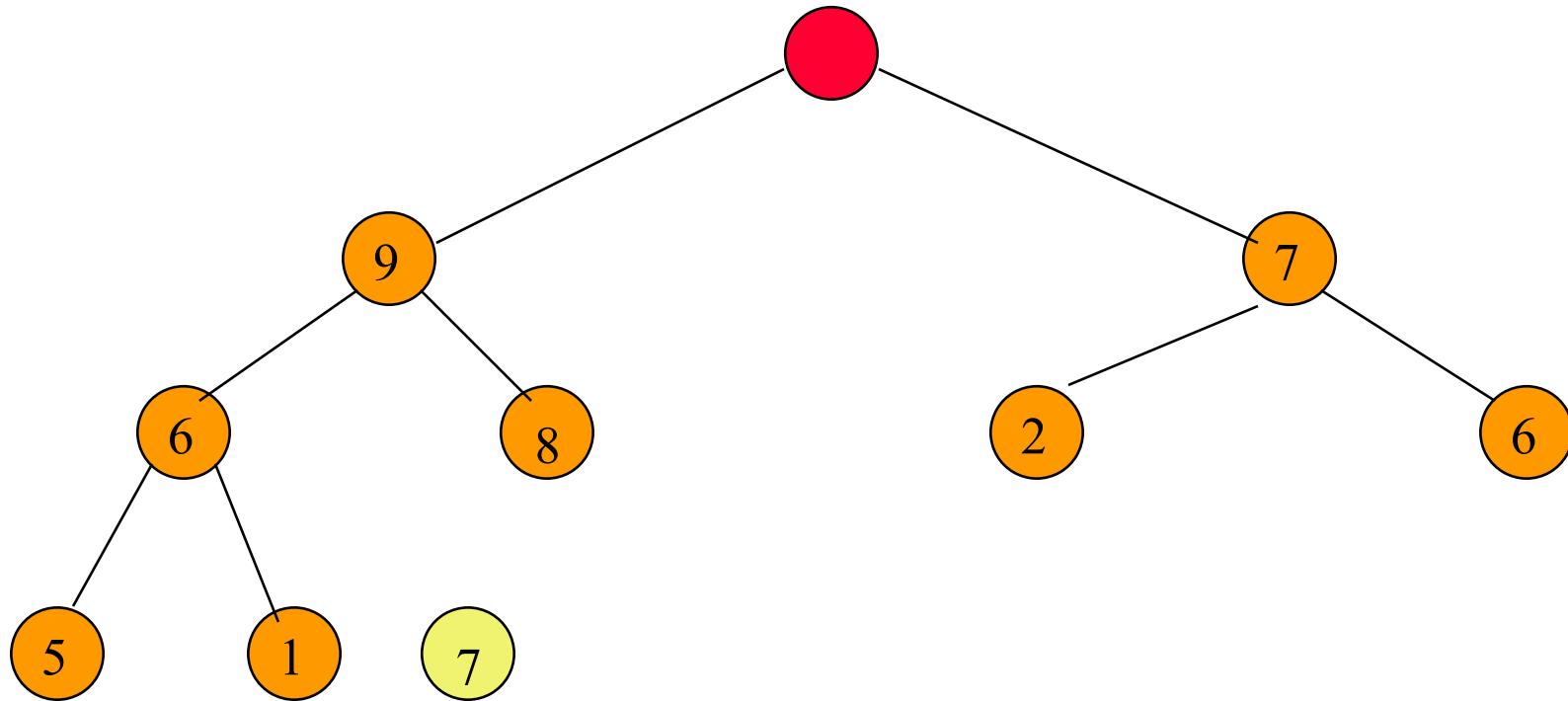
Max element is 15.

Removing The Max Element



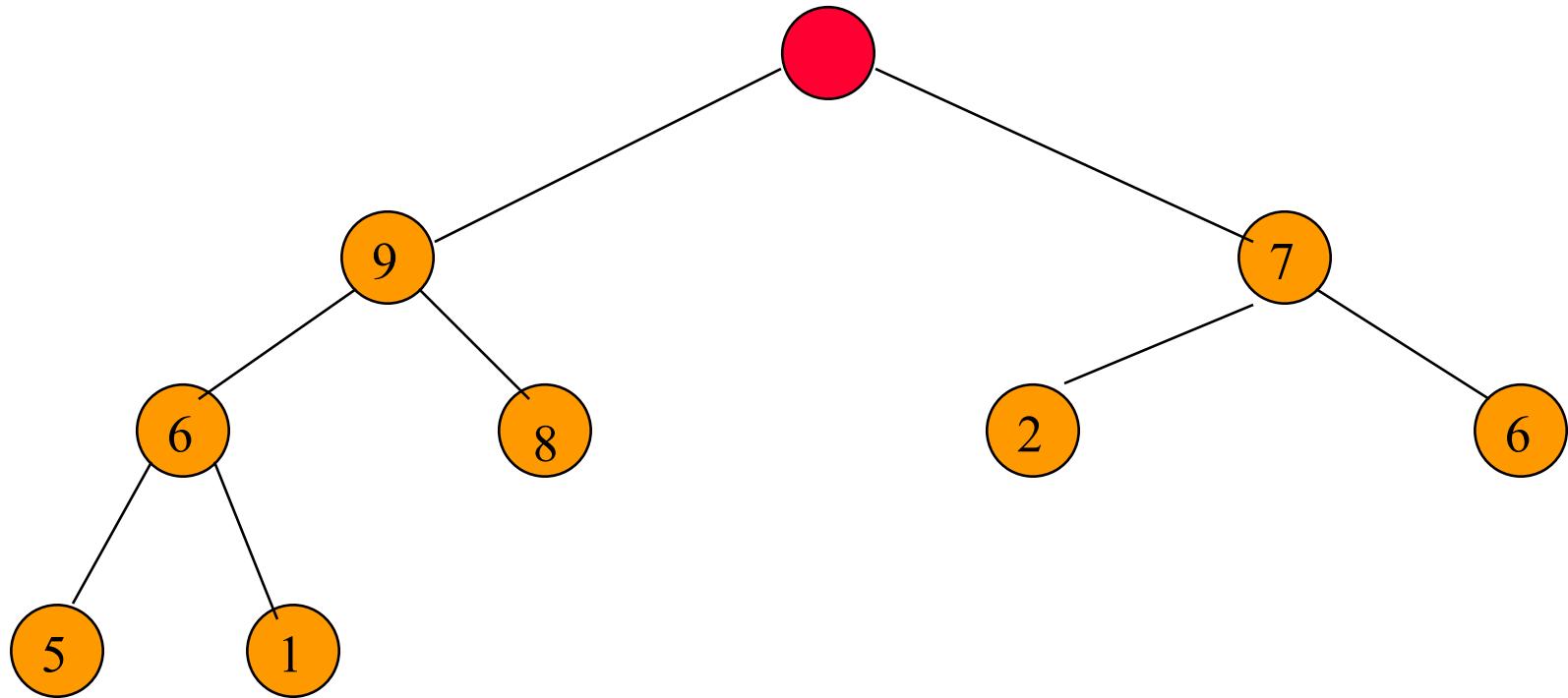
After max element is removed.

Removing The Max Element



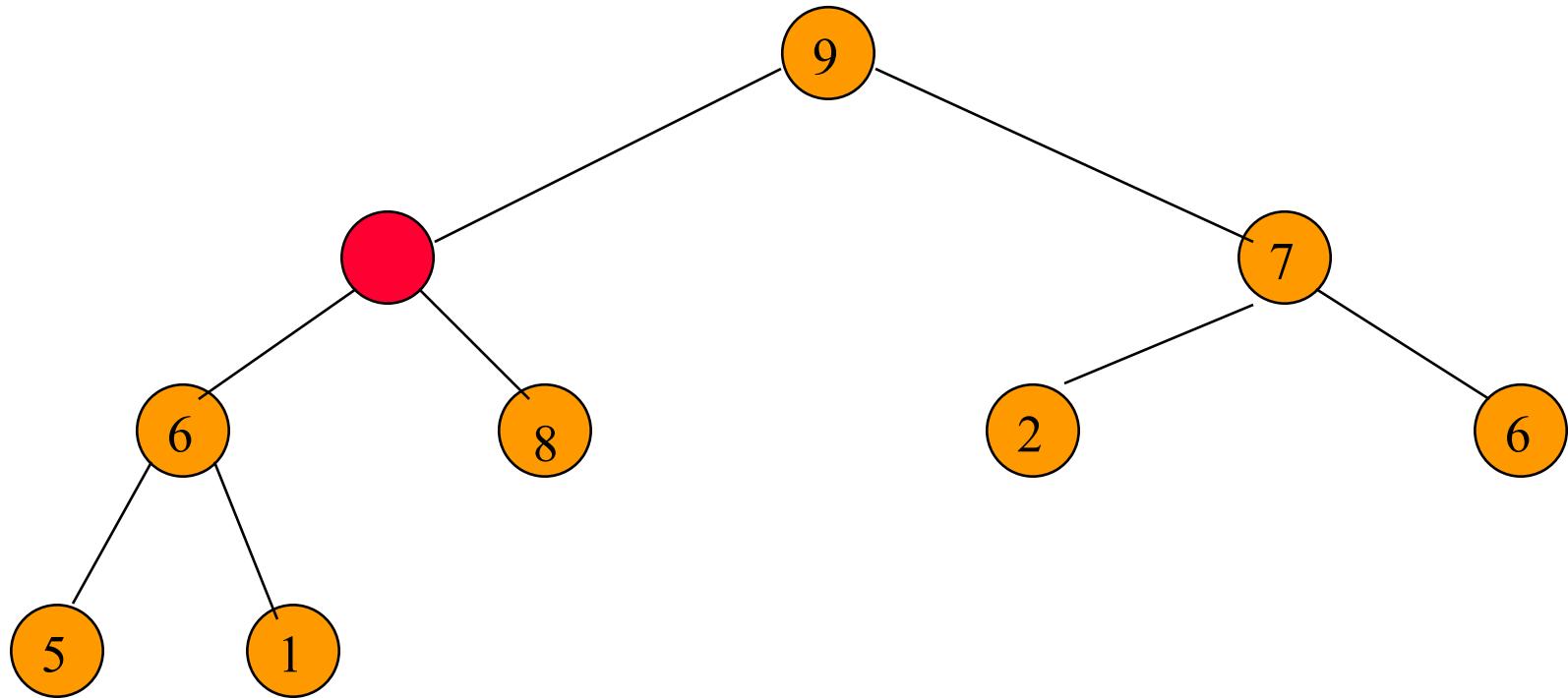
Heap with 9 nodes.

Removing The Max Element



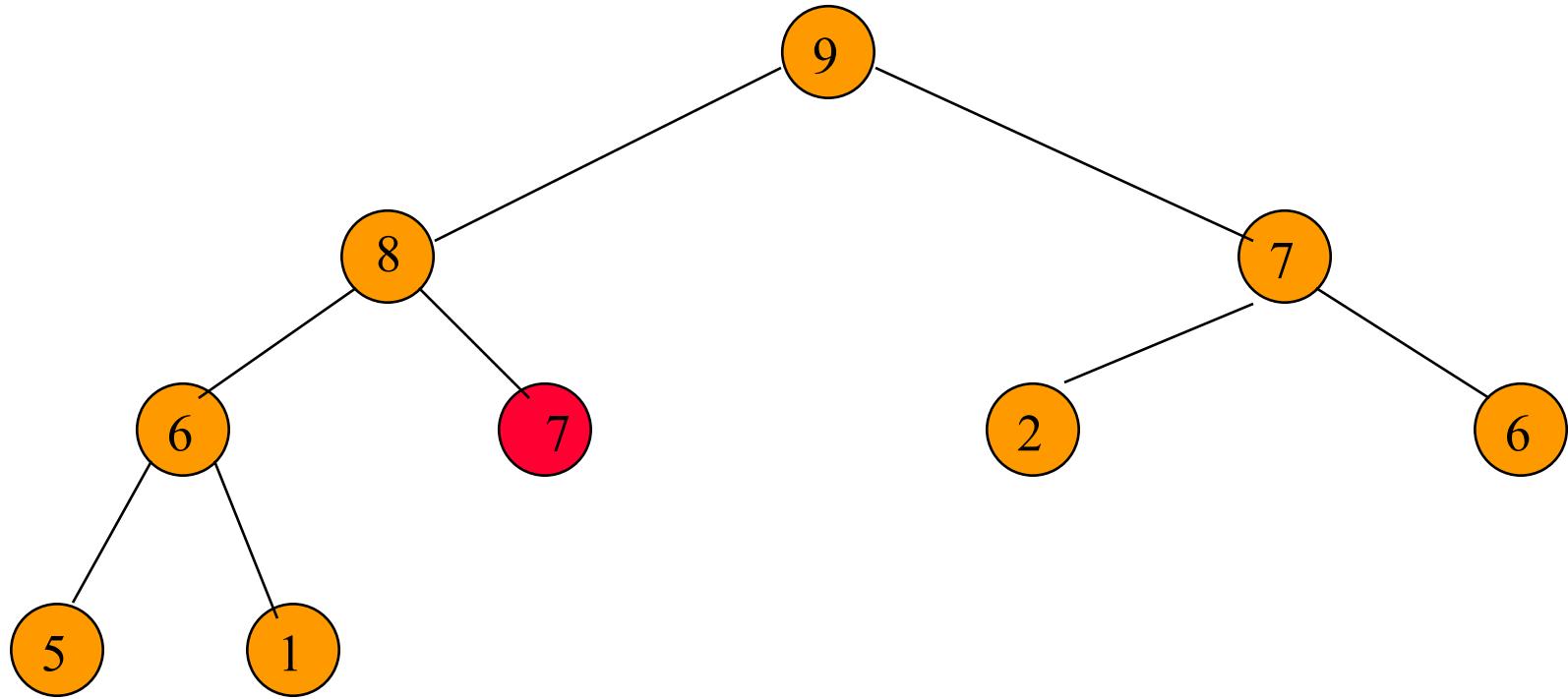
Reinsert 7.

Removing The Max Element



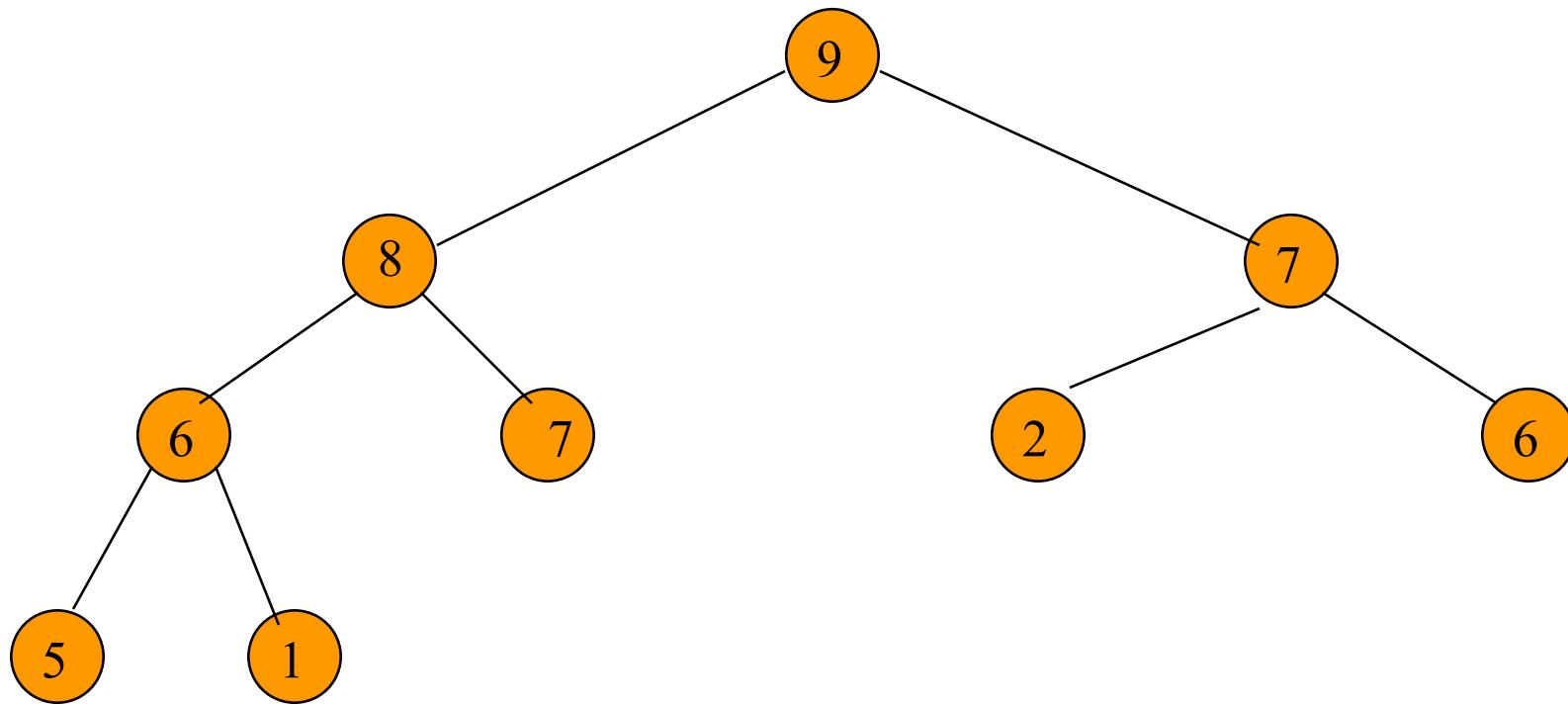
Reinsert 7.

Removing The Max Element



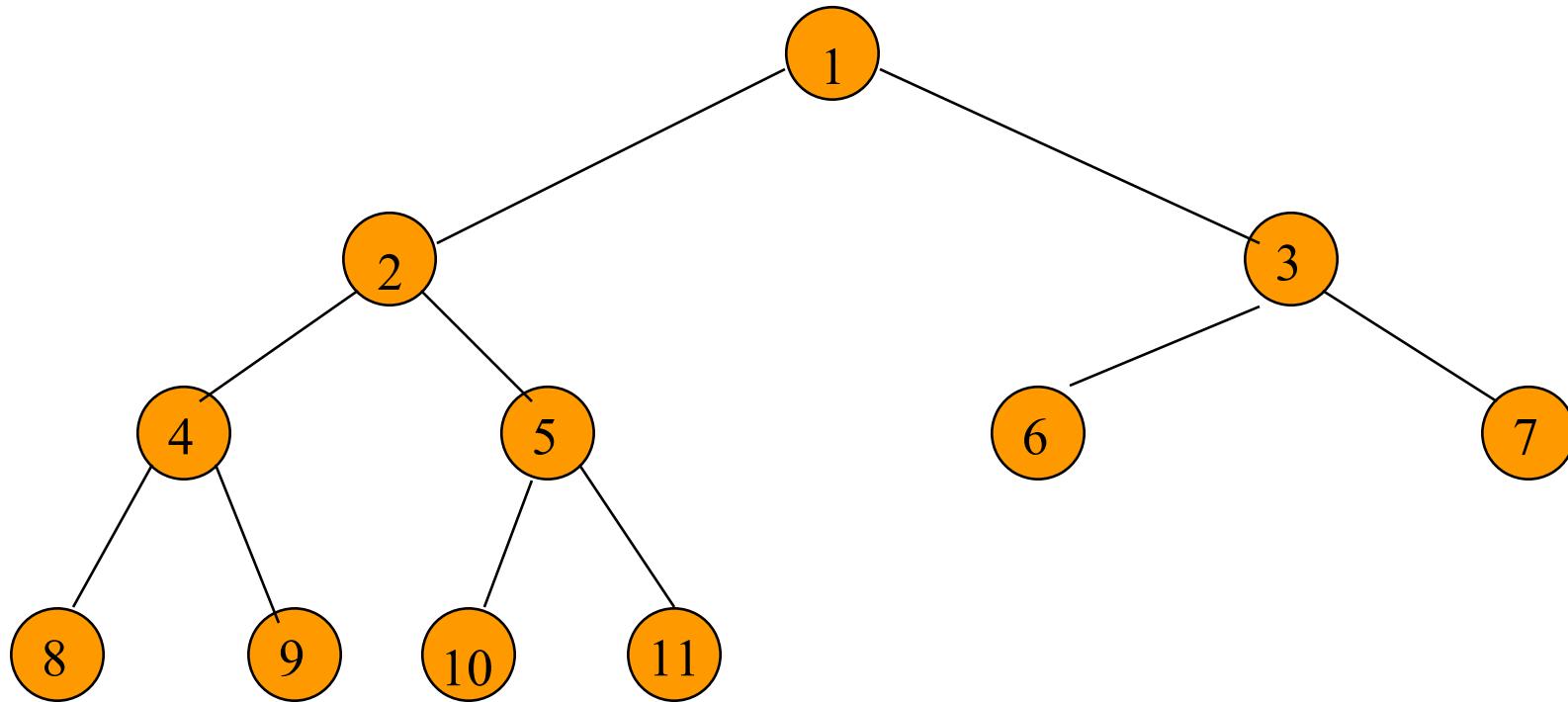
Reinsert 7.

Complexity Of Remove Max Element



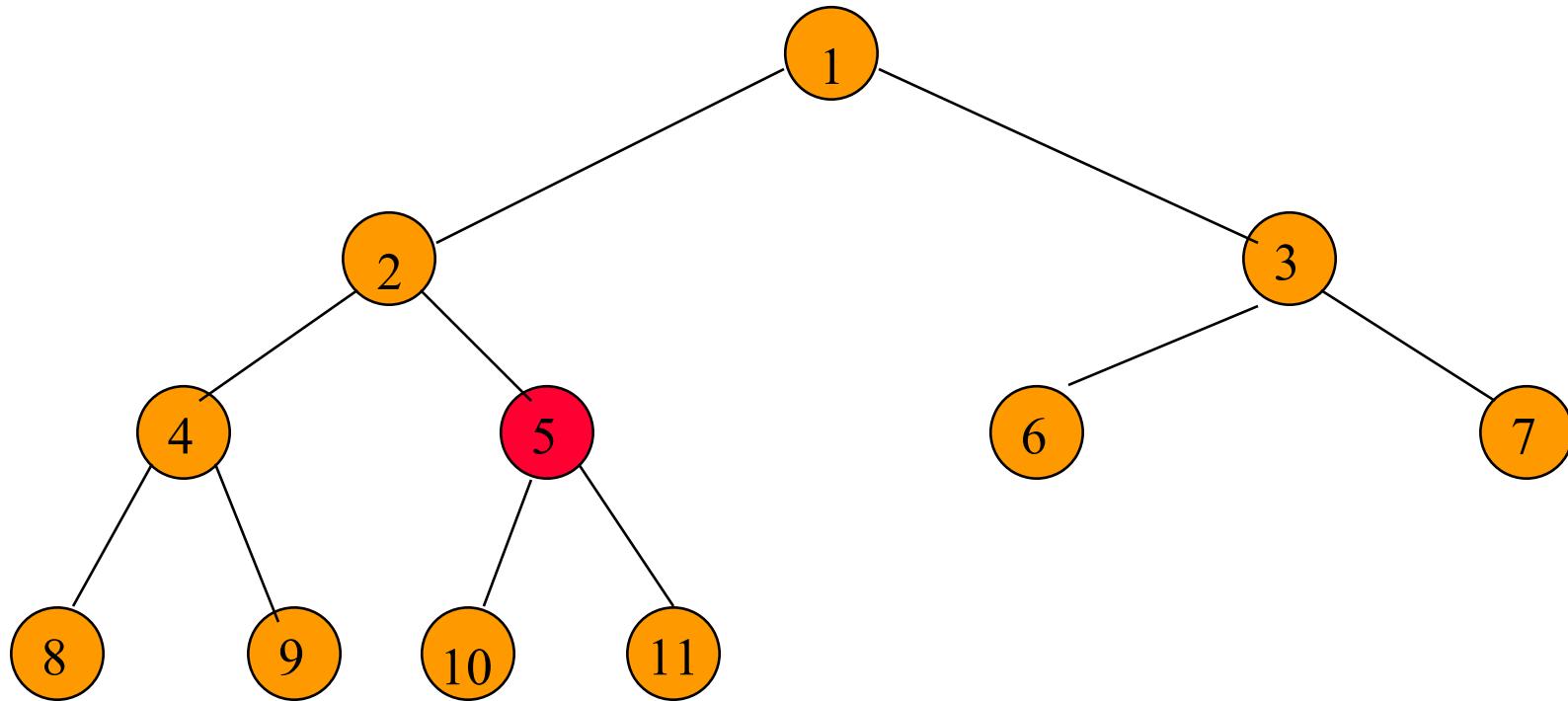
Complexity is $O(\log n)$.

Initializing A Max Heap



input array = [-, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

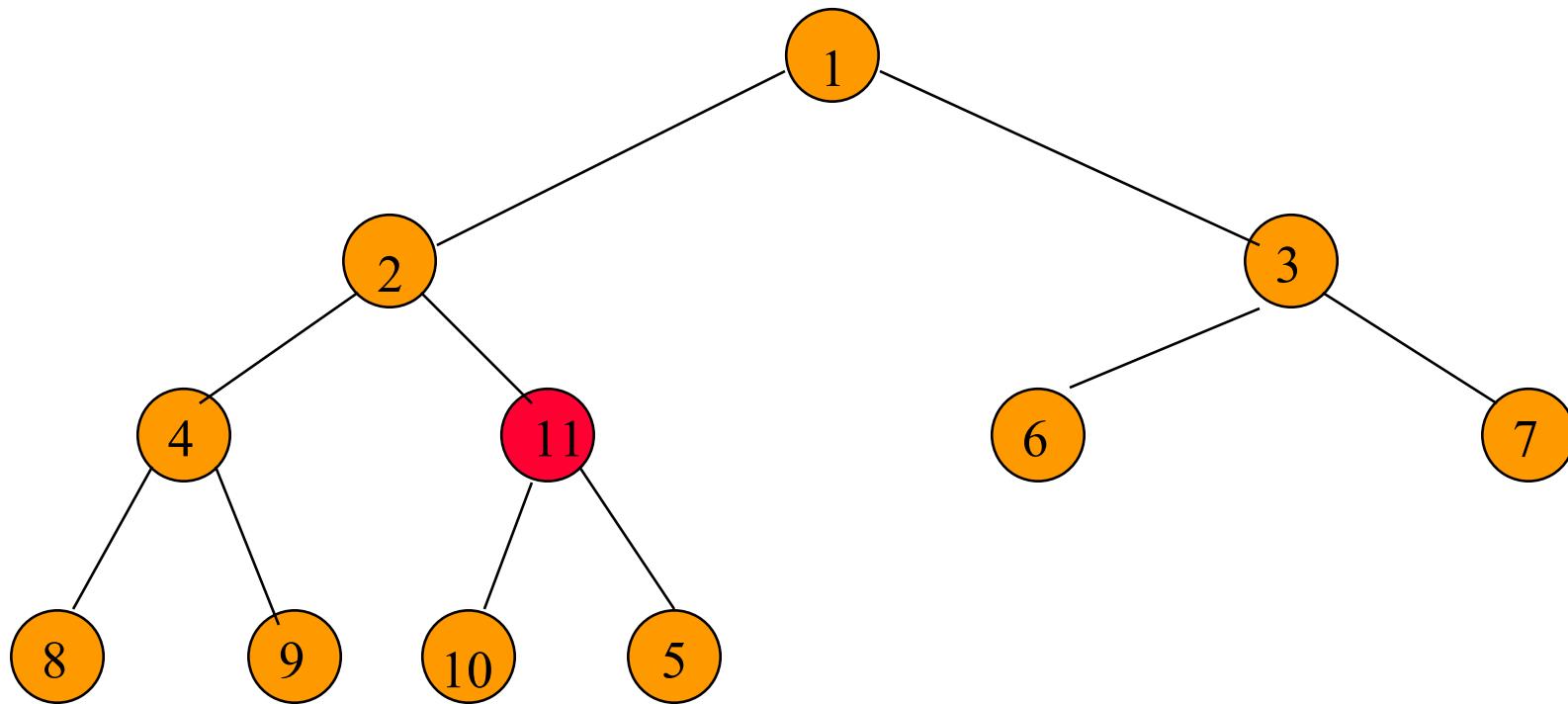
Initializing A Max Heap



Start at rightmost array position that has a child.

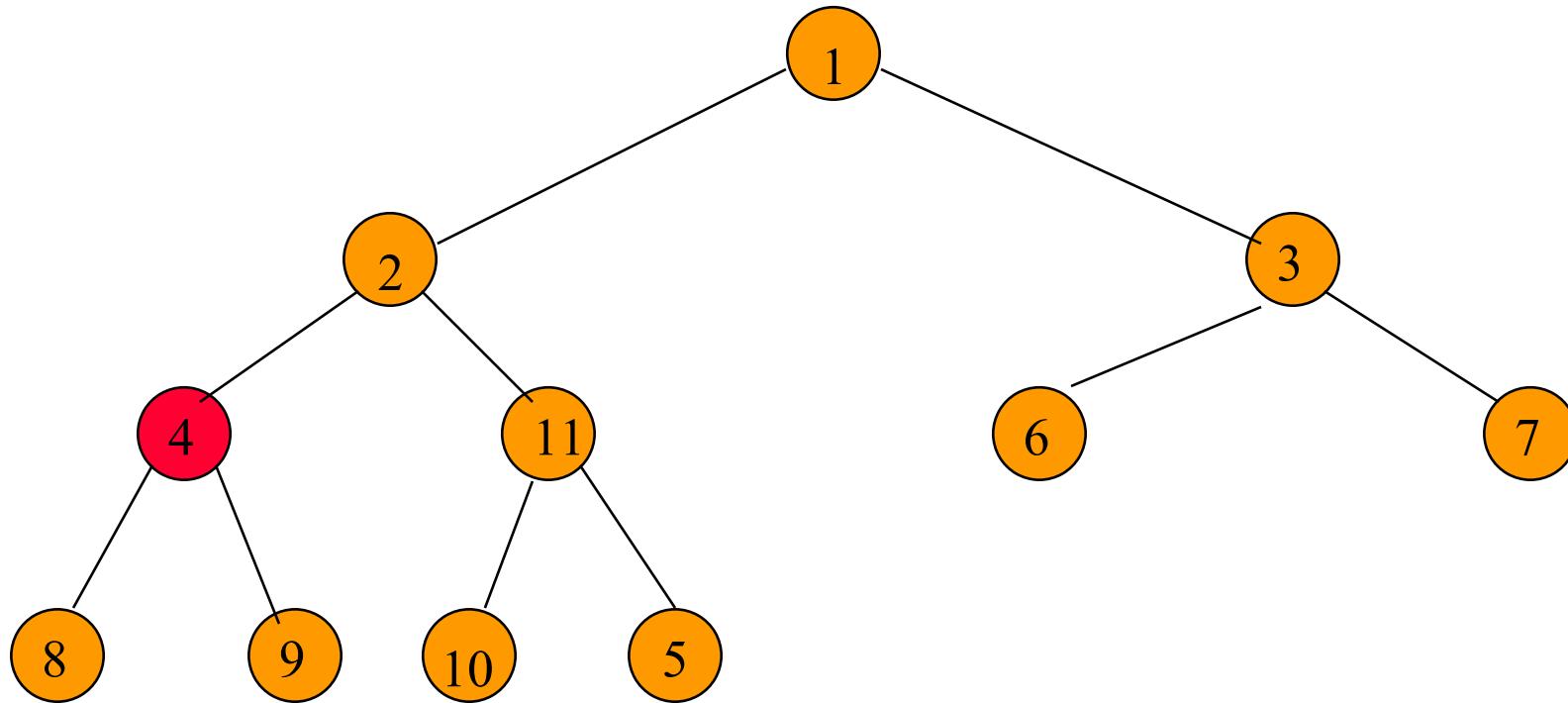
Index is $n/2$.

Initializing A Max Heap

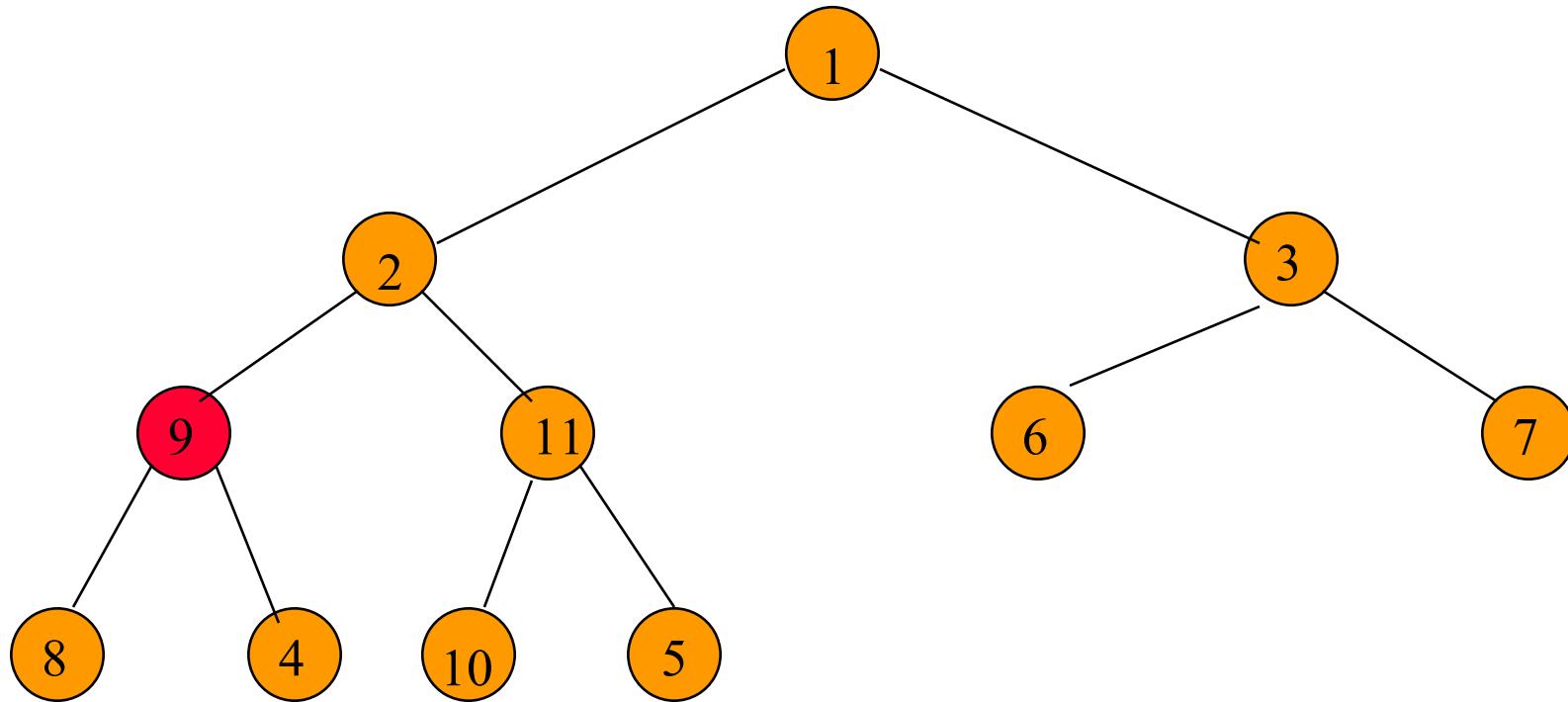


Move to next lower array position.

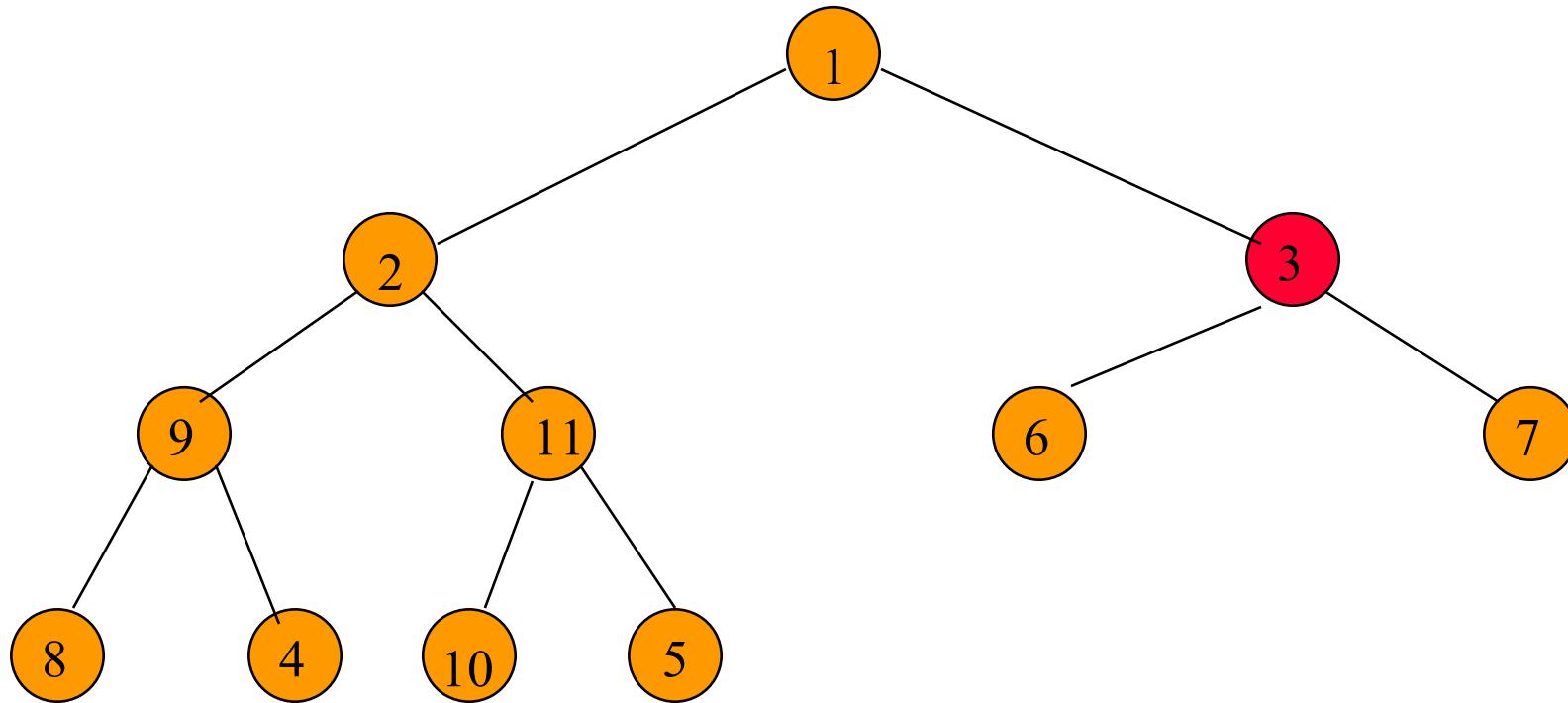
Initializing A Max Heap



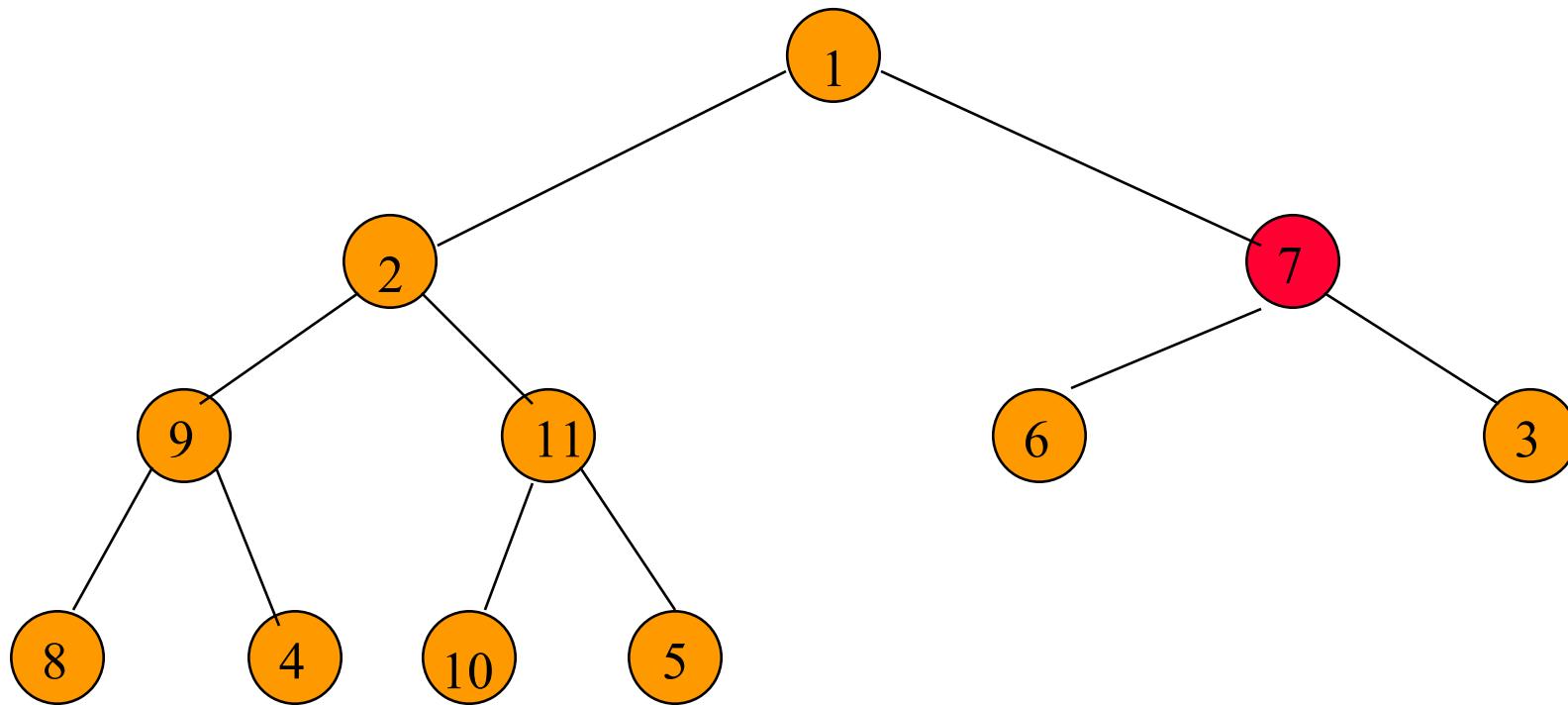
Initializing A Max Heap



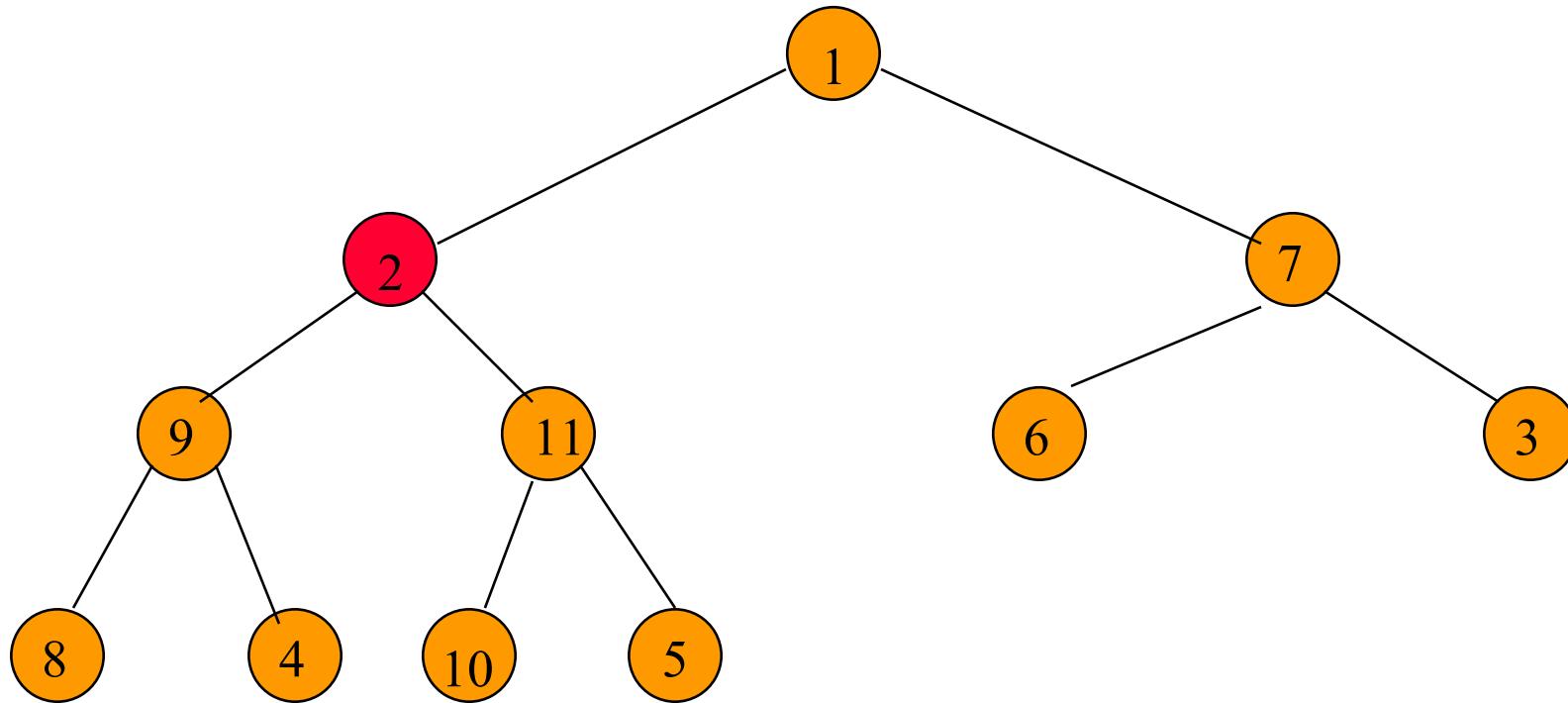
Initializing A Max Heap



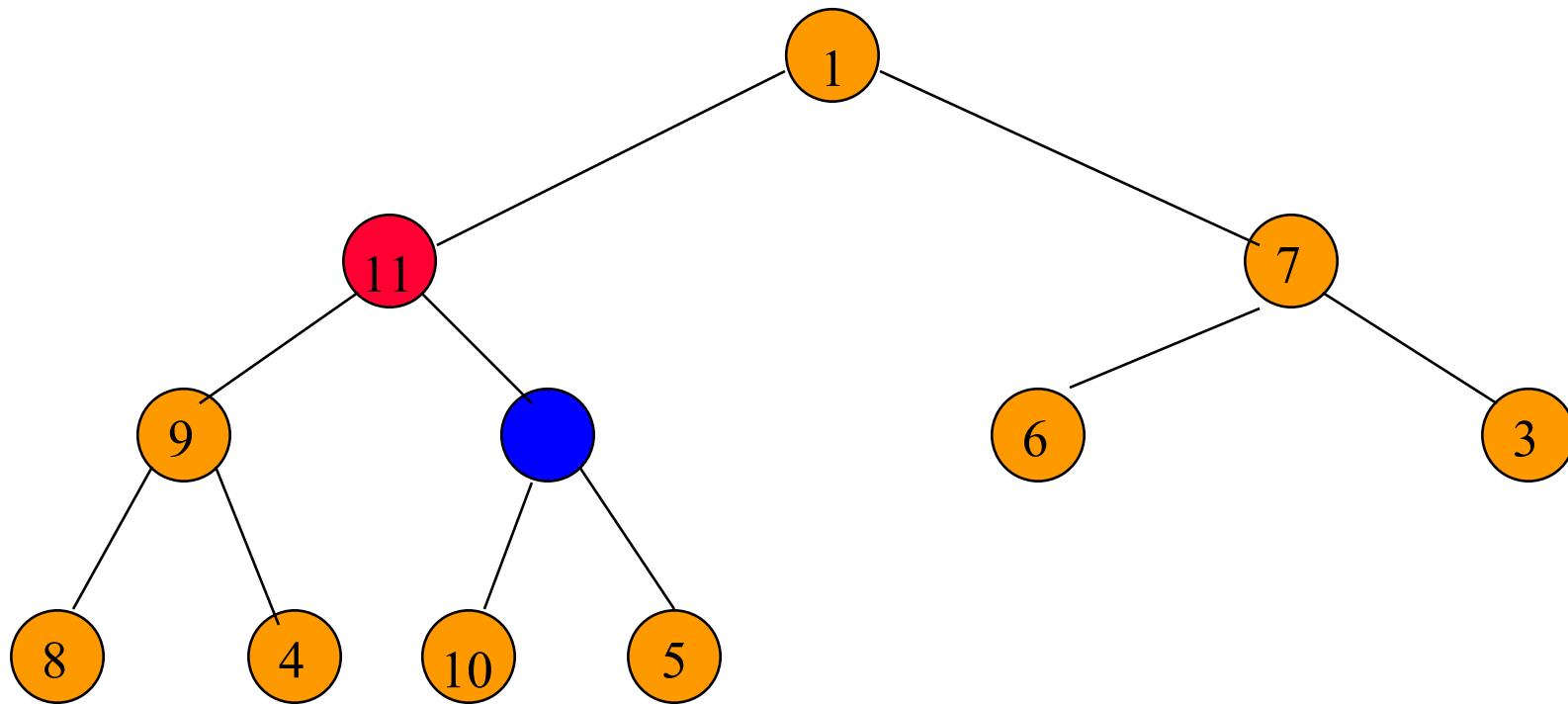
Initializing A Max Heap



Initializing A Max Heap

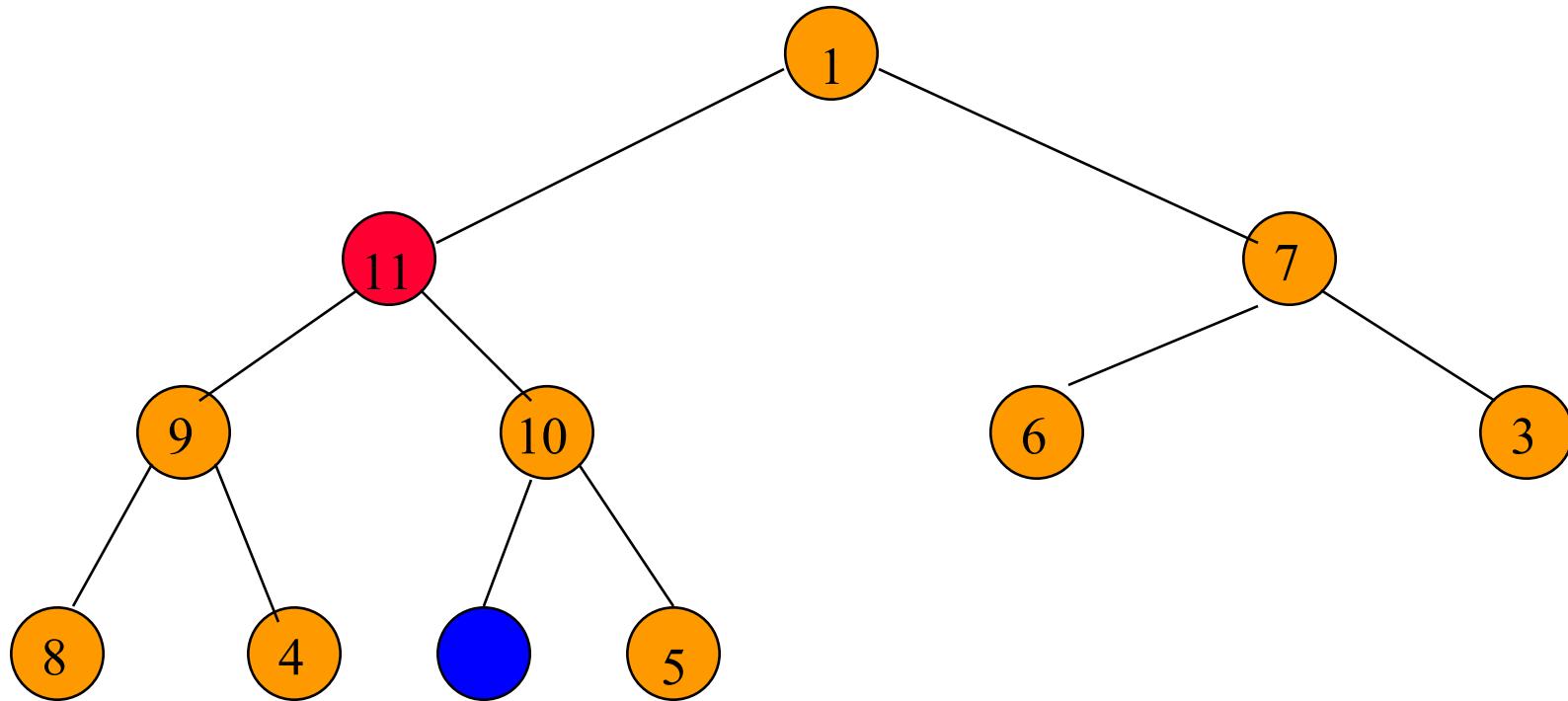


Initializing A Max Heap



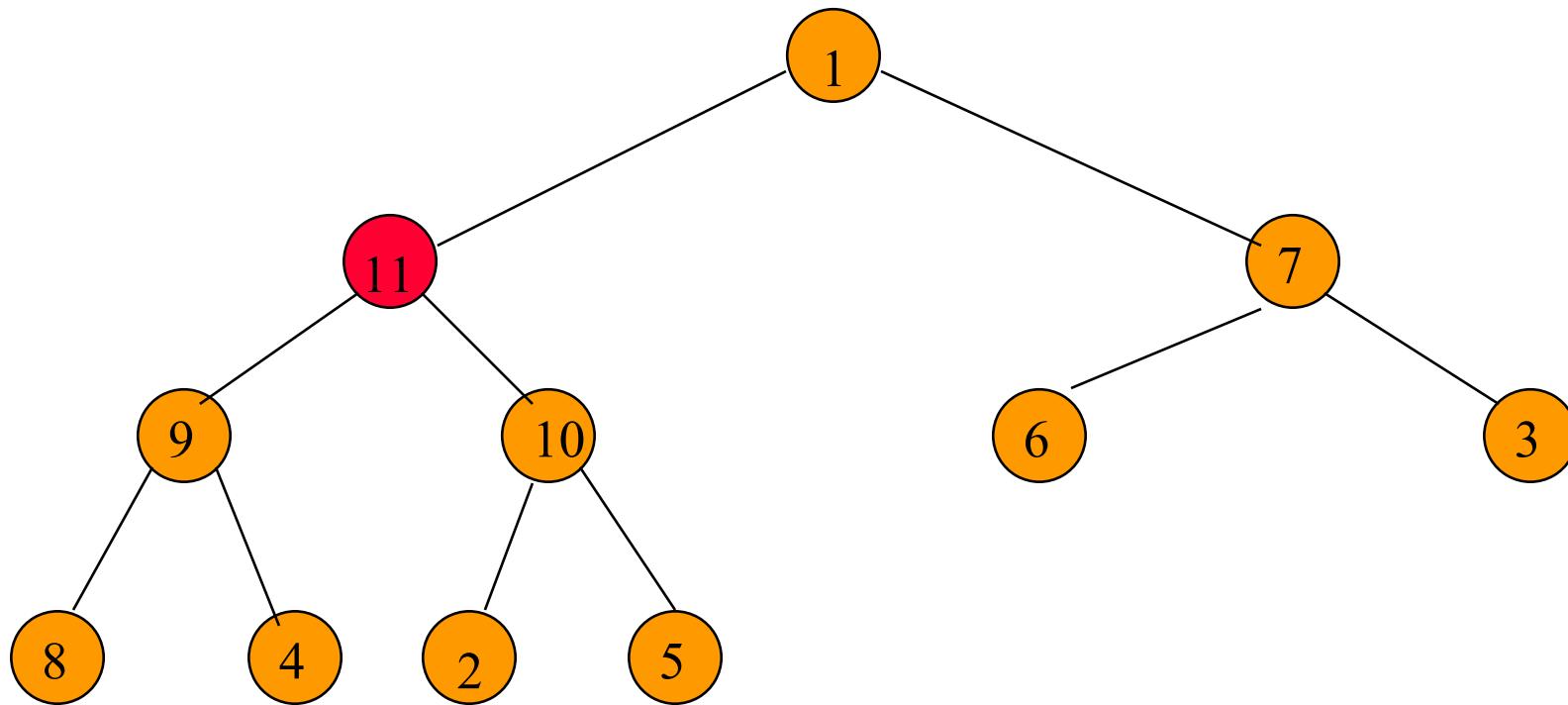
Find a home for 2.

Initializing A Max Heap



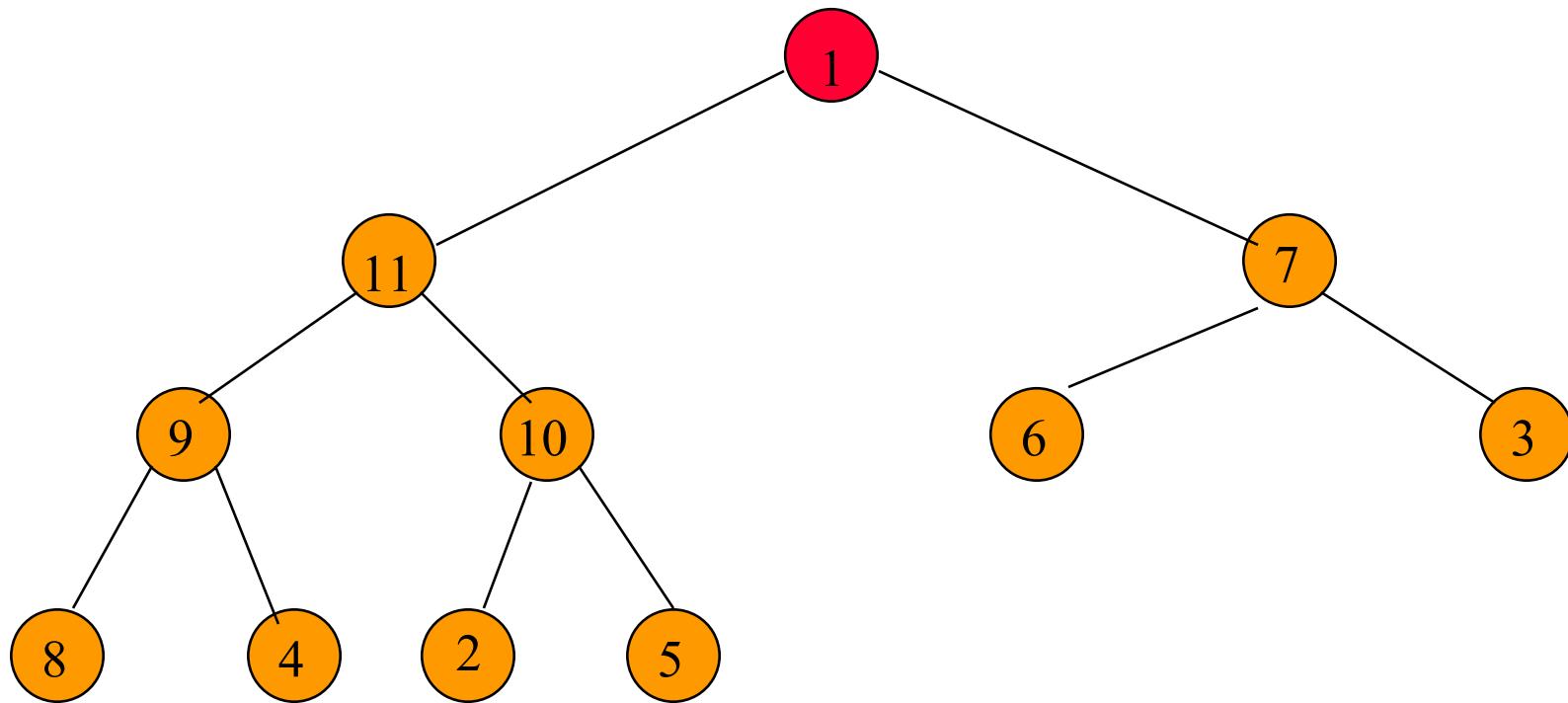
Find a home for 2.

Initializing A Max Heap



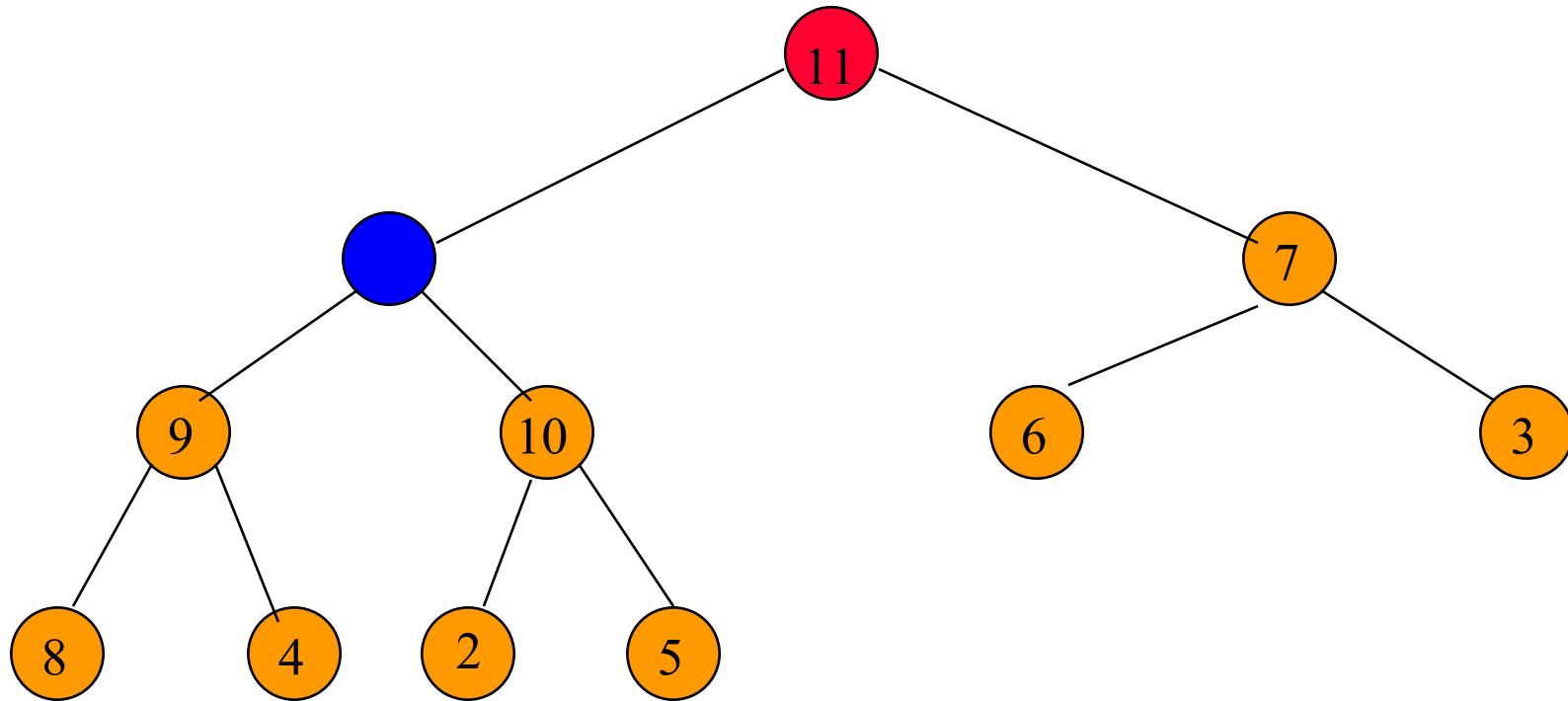
Done, move to next lower array position.

Initializing A Max Heap



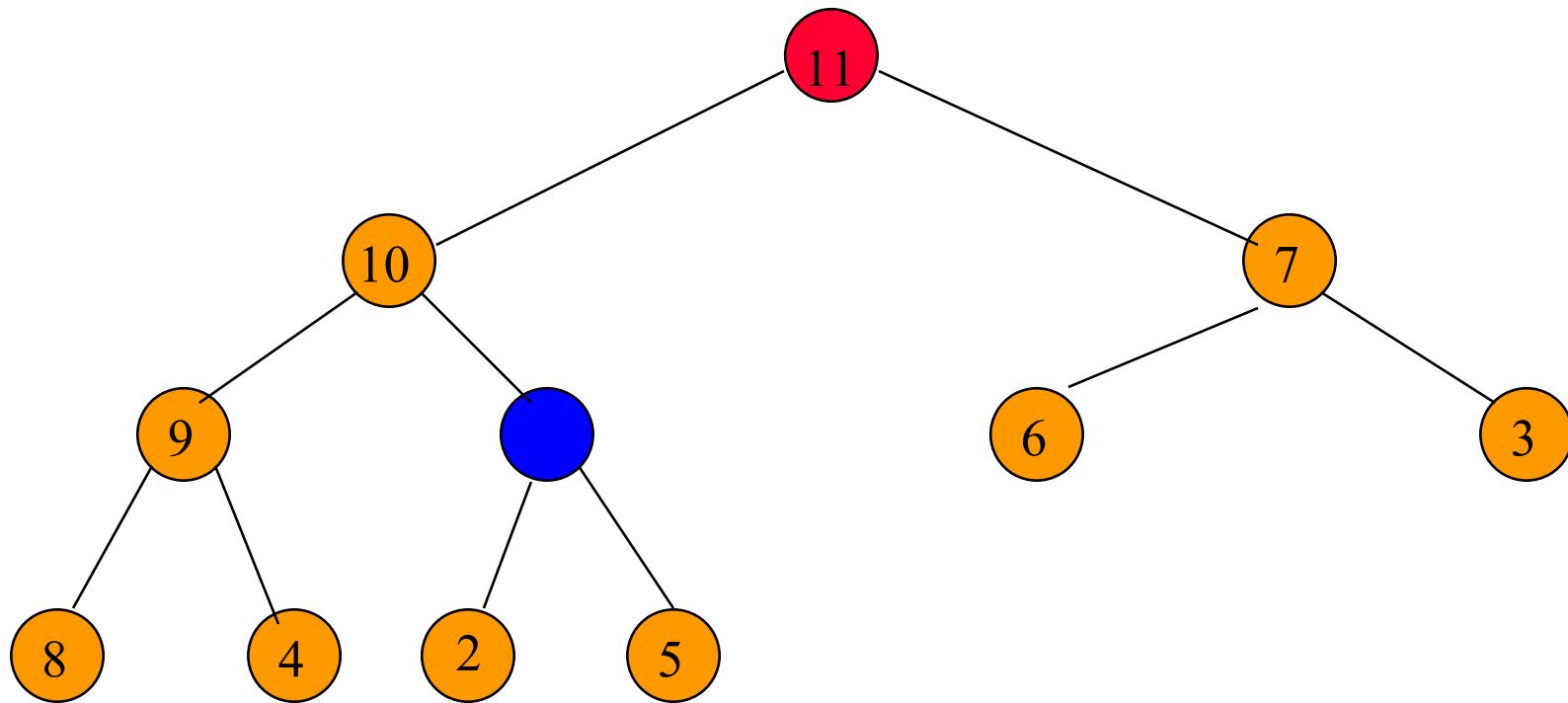
Find home for 1.

Initializing A Max Heap



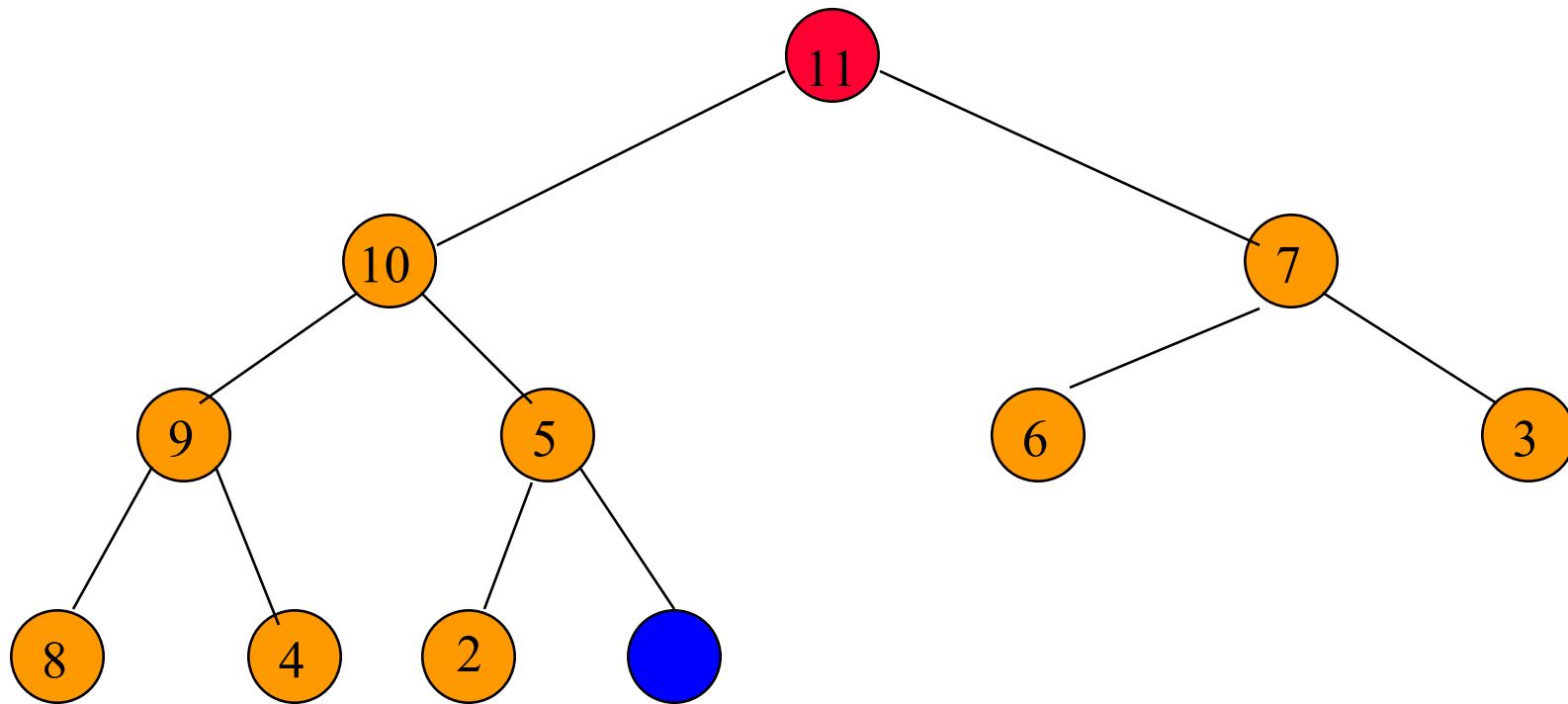
Find home for 1.

Initializing A Max Heap



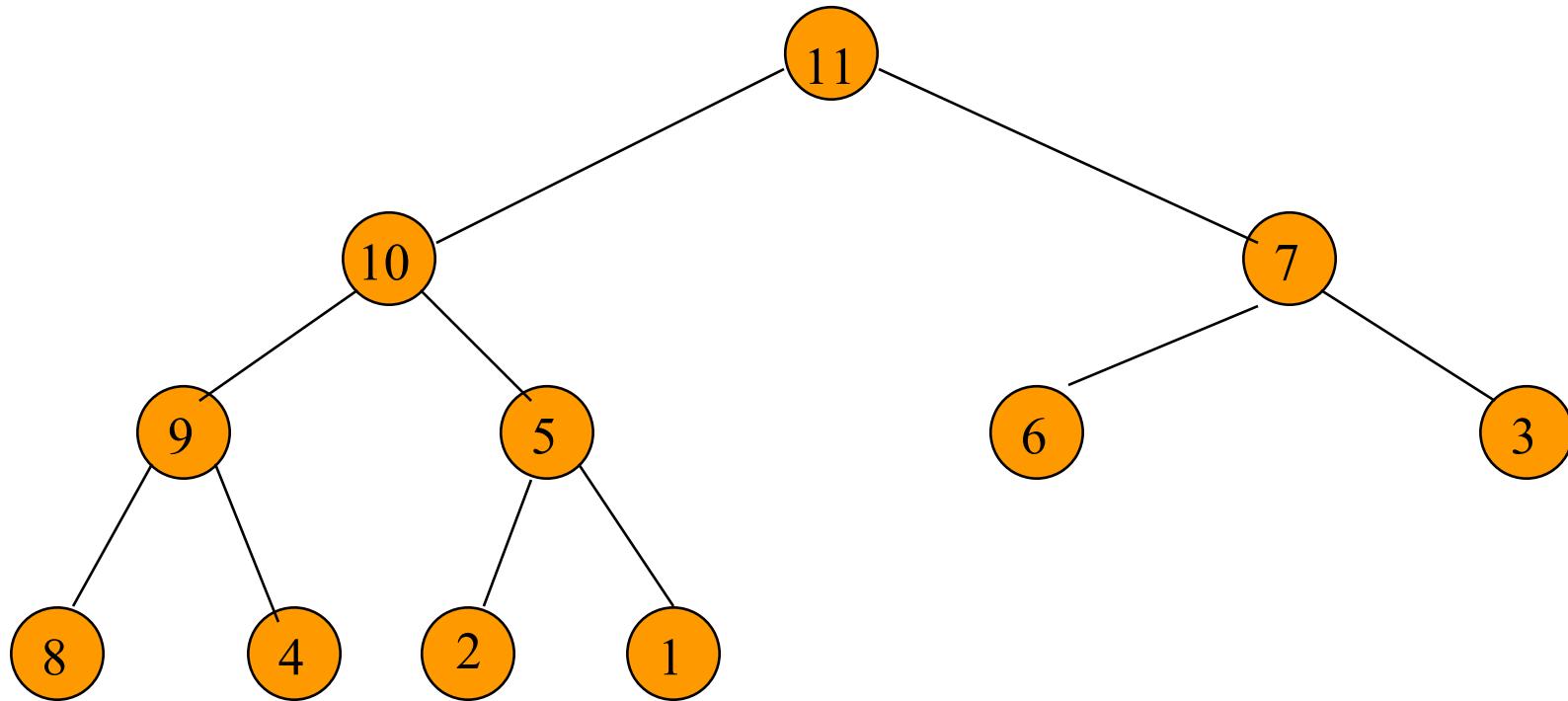
Find home for 1.

Initializing A Max Heap



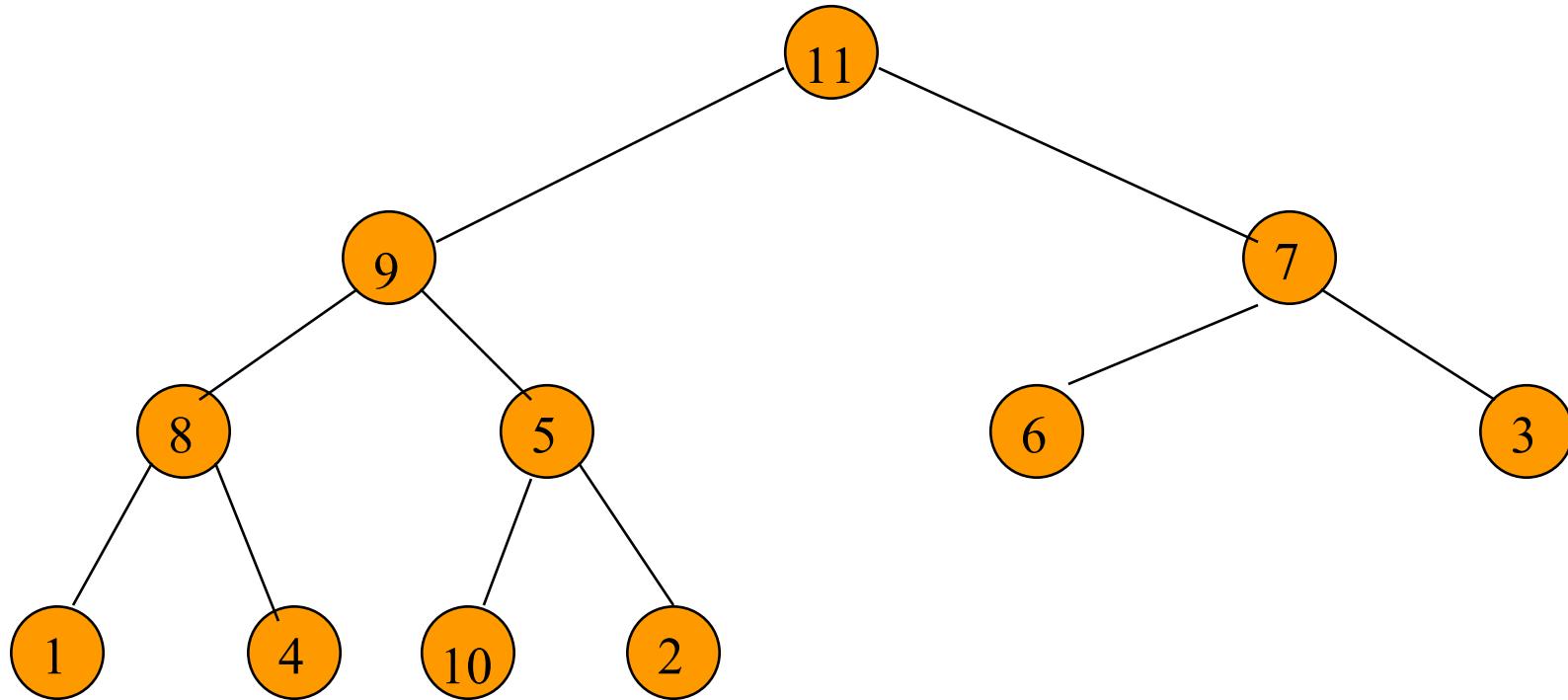
Find home for 1.

Initializing A Max Heap



Done.

Time Complexity



Height of heap = h .

Number of subtrees with root at level j is $\leq 2^{j-1}$.

Time for each subtree is $O(h-j+1)$.



Complexity

Time for level j subtrees is $\leq 2^{j-1}(h-j+1) = t(j)$.

Total time is $t(1) + t(2) + \dots + t(h-1) = O(n)$.

Leftist Trees

Linked binary tree.

Can do everything a heap can do and in the same asymptotic complexity.

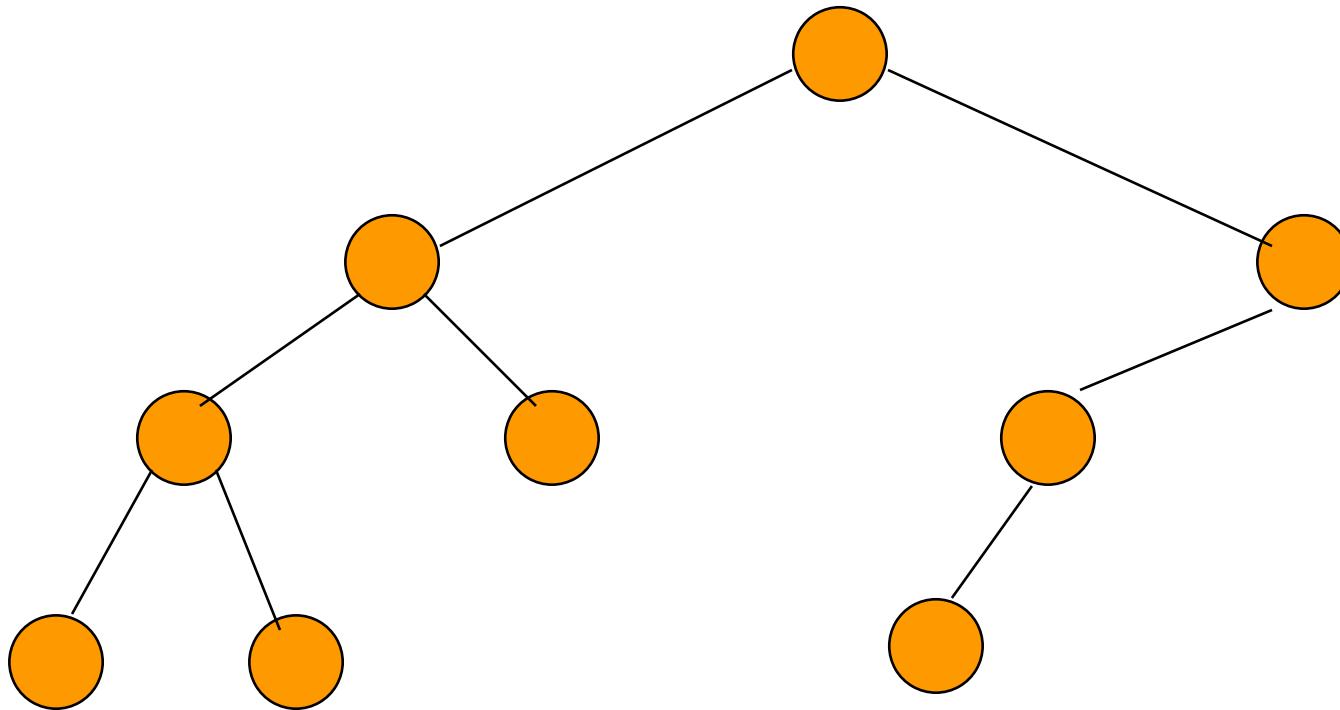
Can meld two leftist tree priority queues in $O(\log n)$ time.

Extended Binary Trees

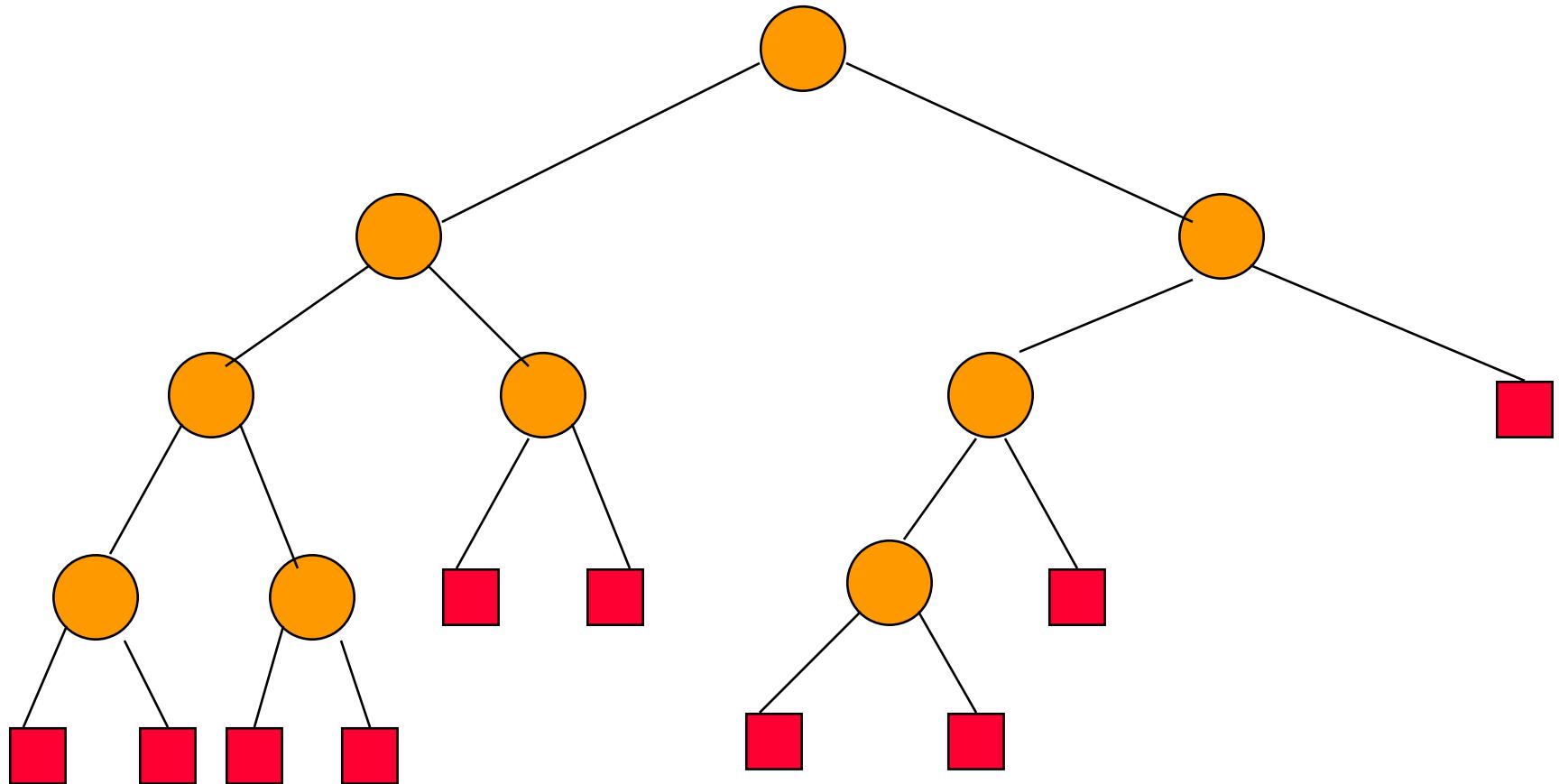
Start with any binary tree and add an external node wherever there is an empty subtree.

Result is an **extended** binary tree.

A Binary Tree



An Extended Binary Tree

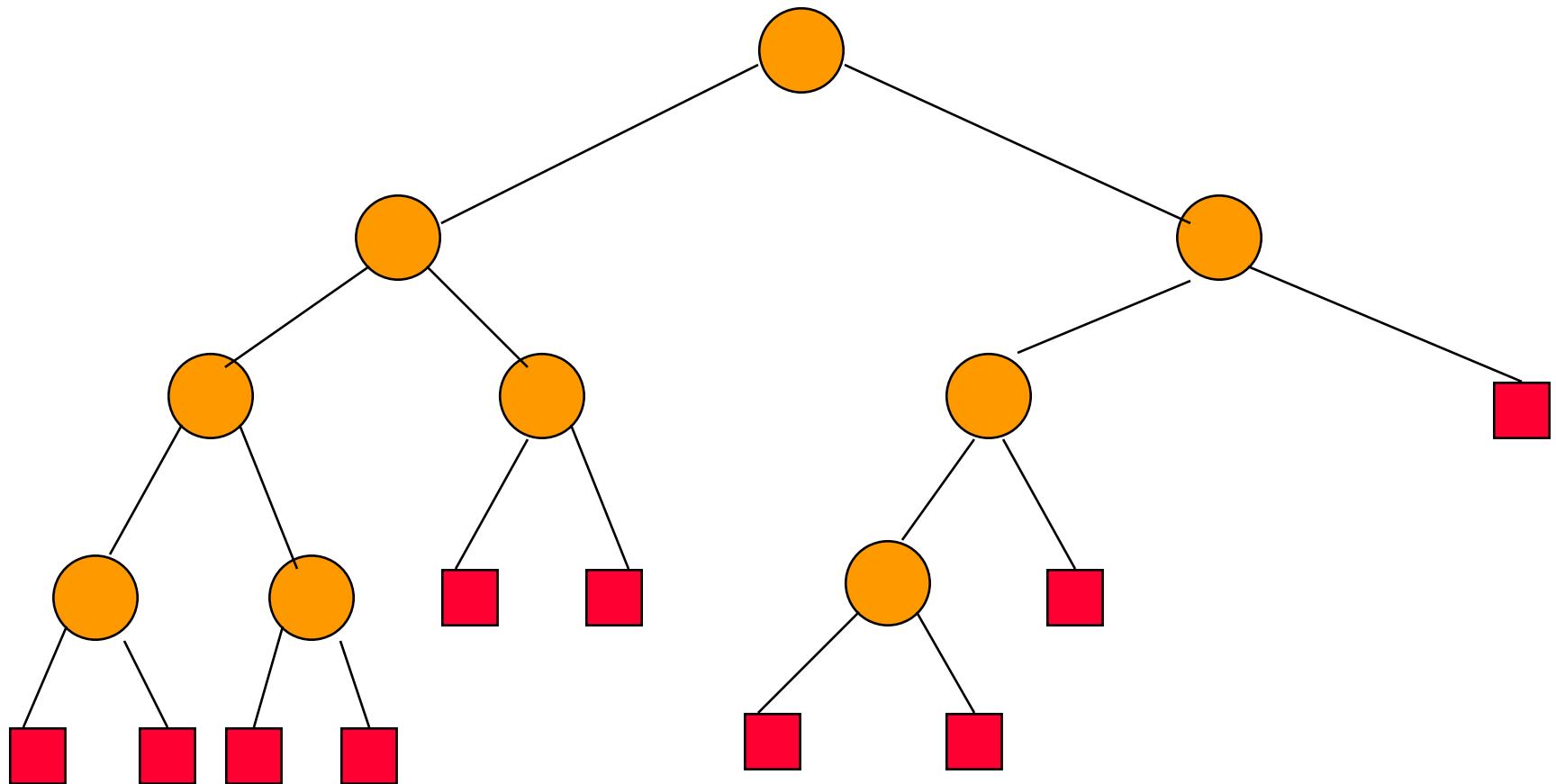


number of external nodes is $n+1$

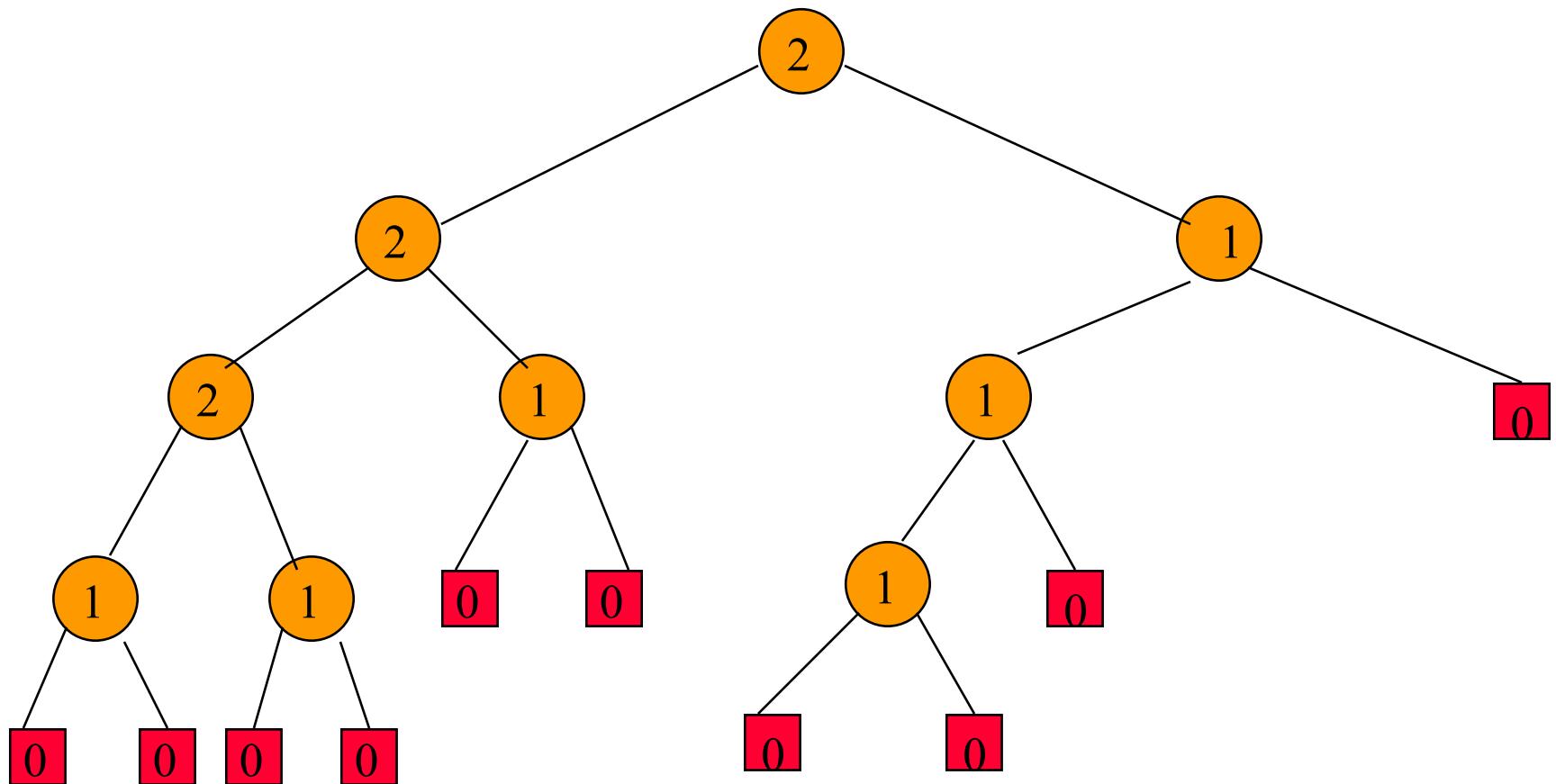
The Function $s()$

For any node x in an extended binary tree,
let $s(x)$ be the length of a shortest path
from x to an external node in the subtree
rooted at x .

$s()$ Values Example



$s()$ Values Example



Properties Of $s()$

If x is an external node, then $s(x) = 0$.

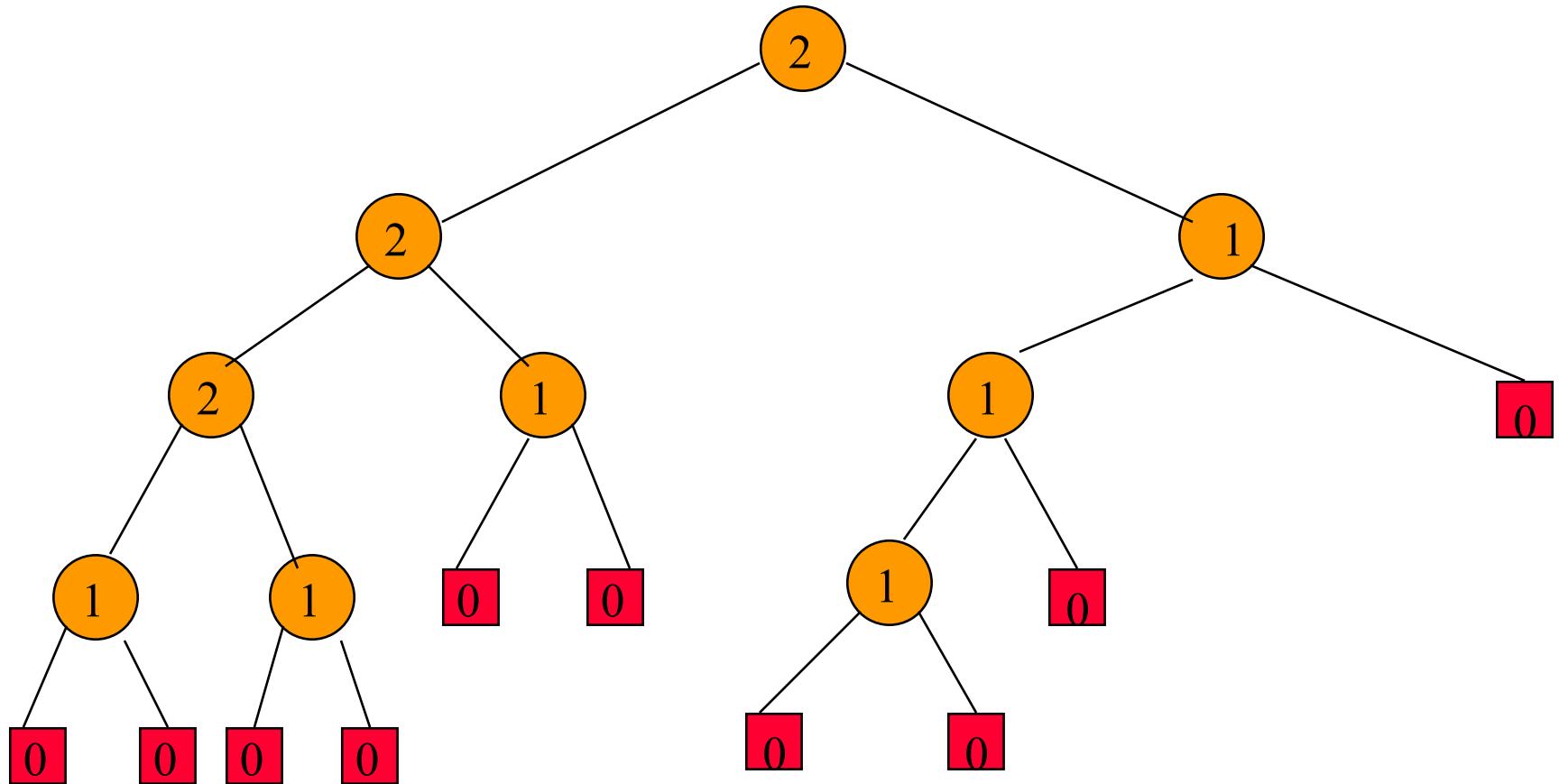
Otherwise,

$$s(x) = \min \{ s(\text{leftChild}(x)), \\ s(\text{rightChild}(x)) \} + 1$$

Height Biased Leftist Trees

A binary tree is a (height biased) leftist tree iff for every internal node x ,
 $s(\text{leftChild}(x)) \geq s(\text{rightChild}(x))$

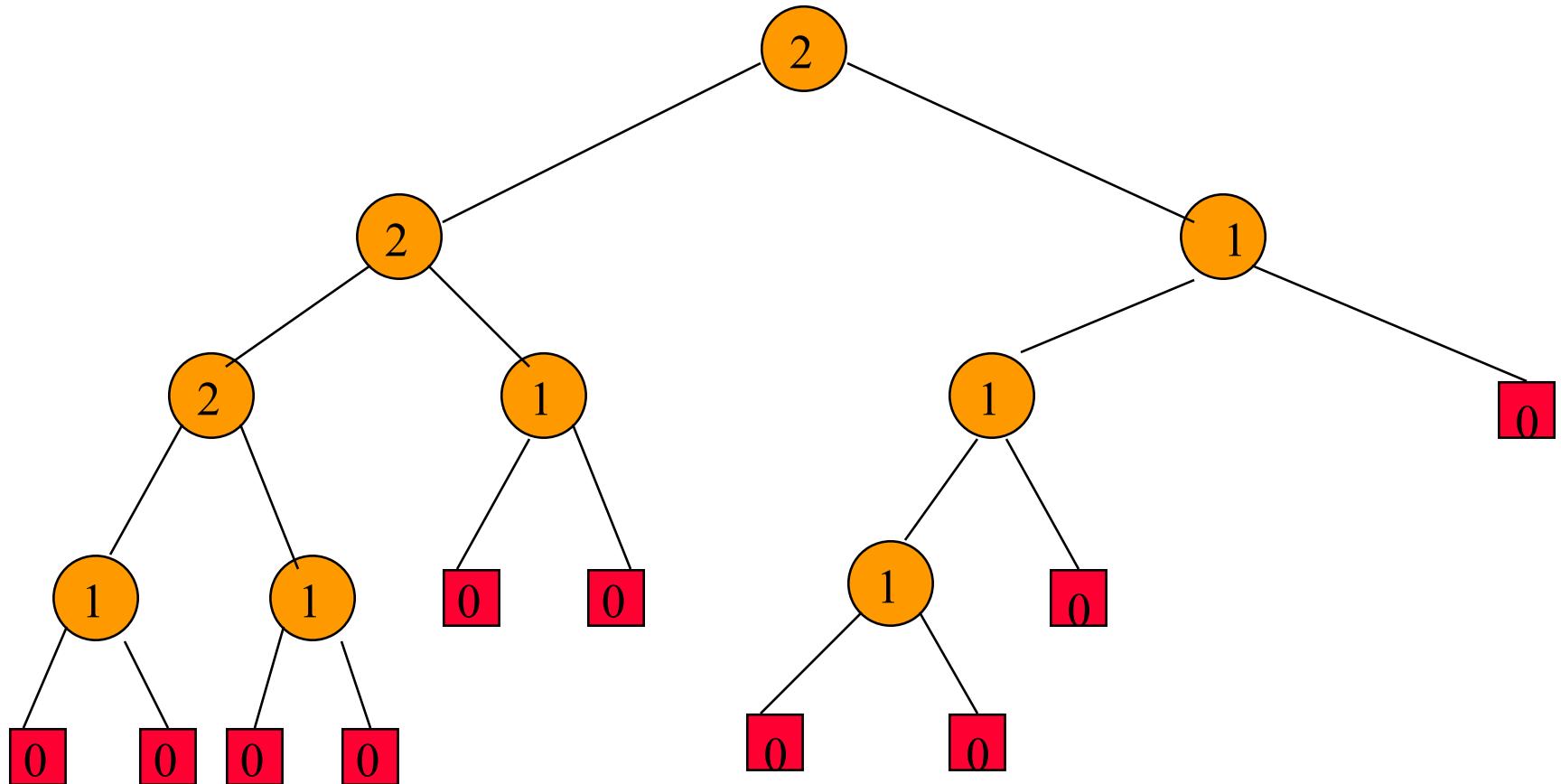
A Leftist Tree



Leftist Trees--Property 1

In a leftist tree, the rightmost path is a shortest root to external node path and the length of this path is $s(\text{root})$.

A Leftist Tree



Length of rightmost path is **2**.

Leftist Trees—Property 2

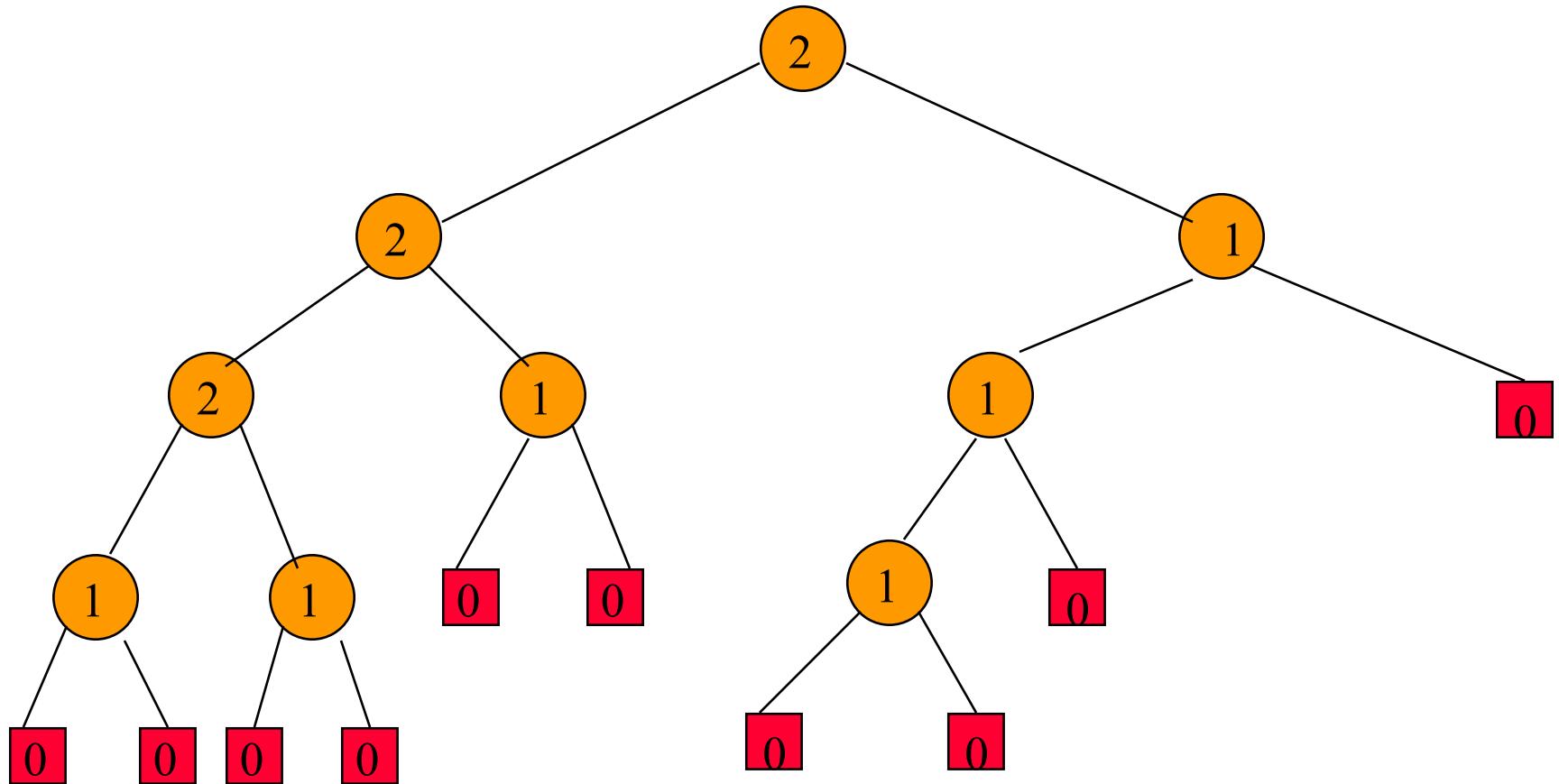
The number of internal nodes is at least

$$2^{s(\text{root})} - 1$$

Because levels 1 through $s(\text{root})$ have no external nodes.

$$\text{So, } s(\text{root}) \leq \log(n+1)$$

A Leftist Tree



Levels 1 and 2 have no external nodes.

Leftist Trees—Property 3

Length of rightmost path is $O(\log n)$,
where n is the number of nodes in a
leftist tree.

Follows from Properties 1 and 2.

Leftist Trees As Priority Queues

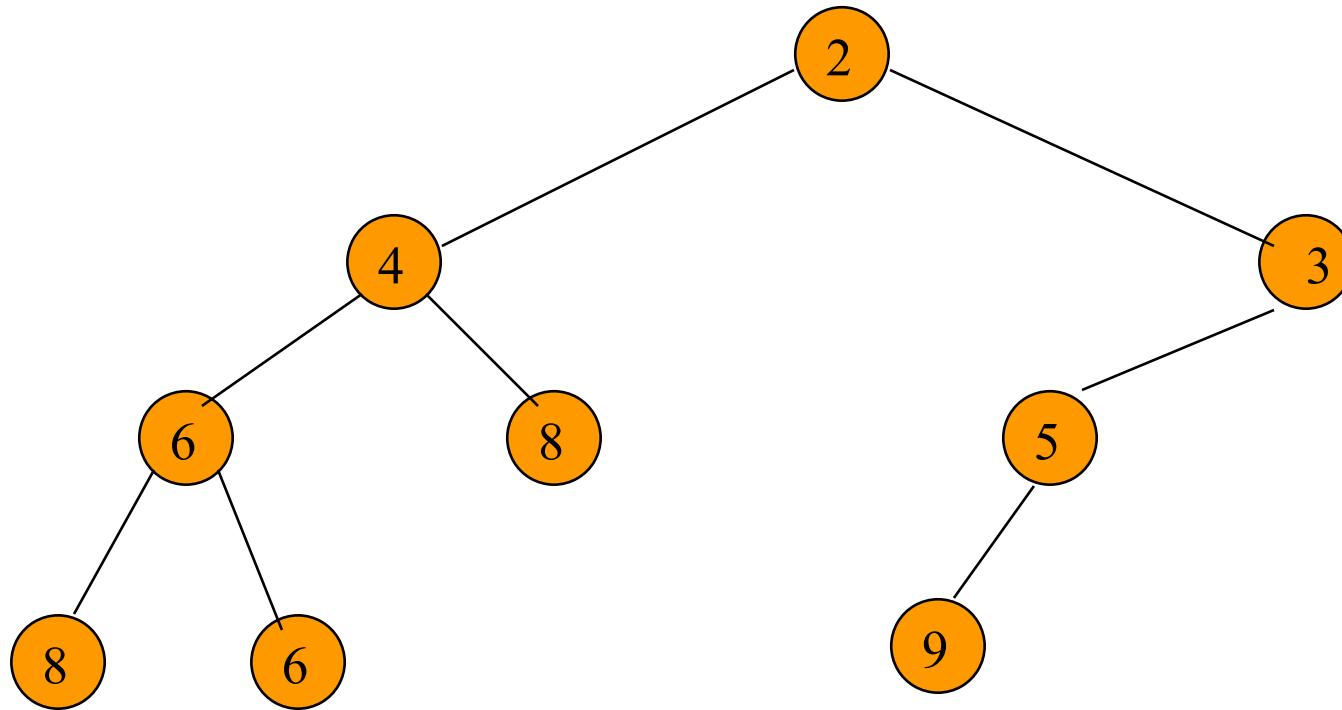
Min leftist tree ... leftist tree that is a min tree.

Used as a min priority queue.

Max leftist tree ... leftist tree that is a max tree.

Used as a max priority queue.

A Min Leftist Tree



Some Min Leftist Tree Operations

put()

remove()

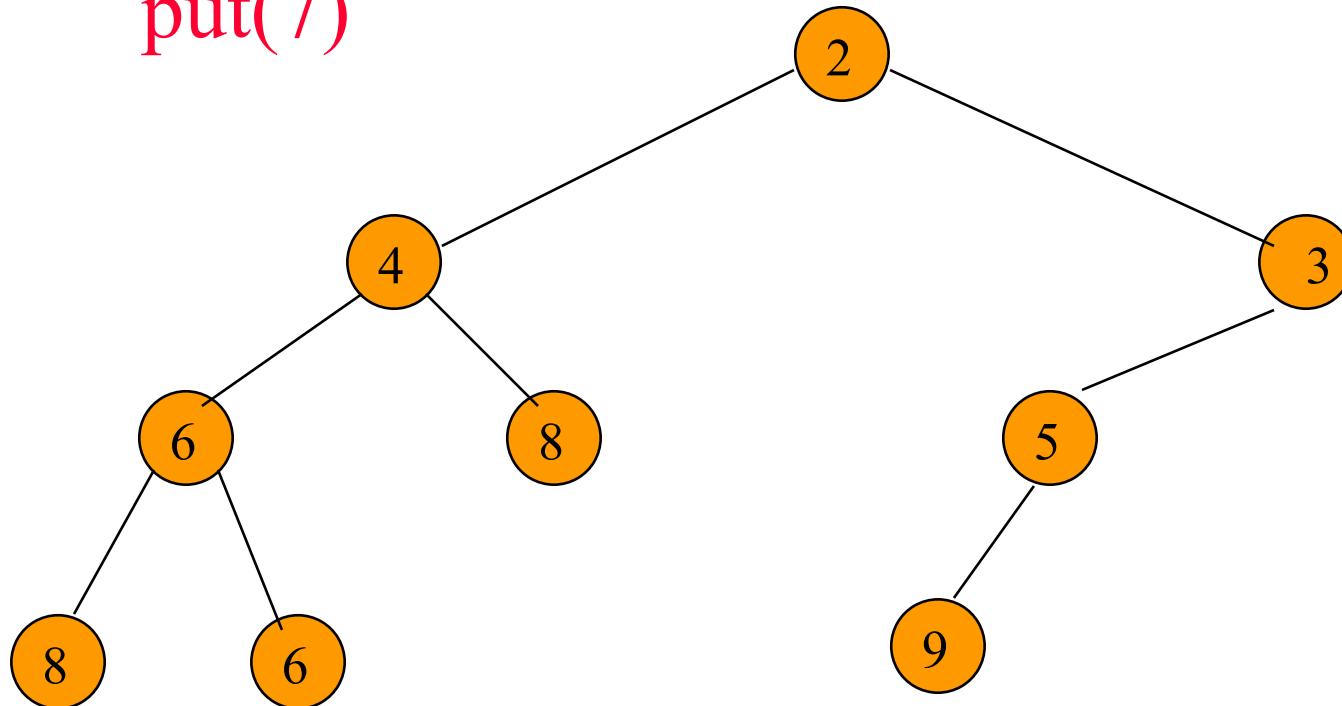
meld()

initialize()

put() and remove() use meld().

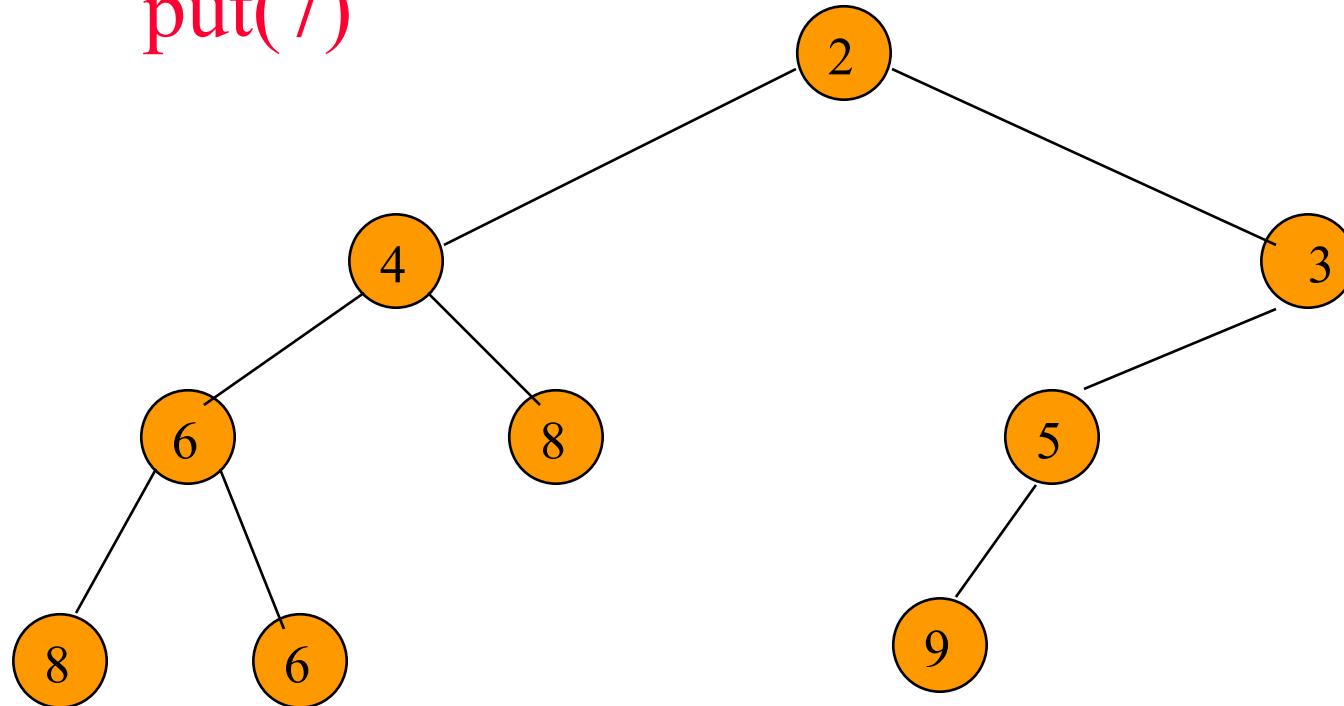
Put Operation

put(7)



Put Operation

put(7)

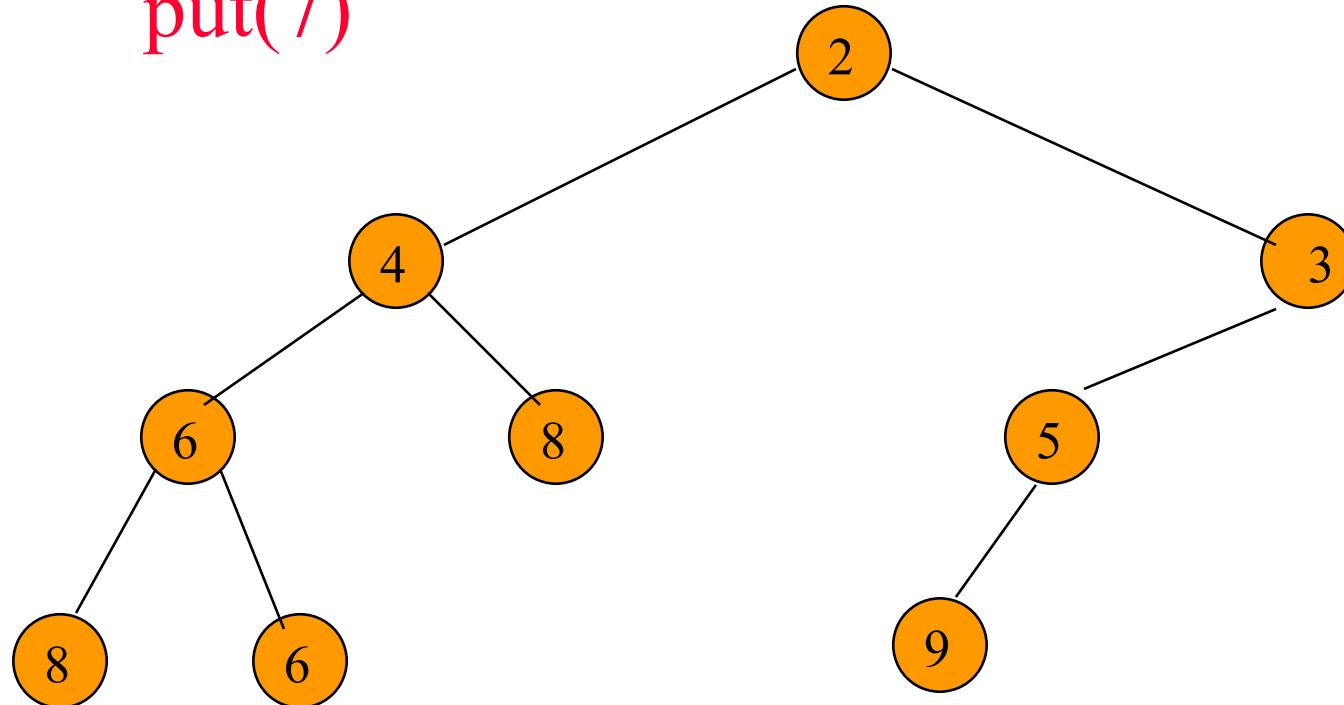


Create a single node min leftist tree.



Put Operation

put(7)

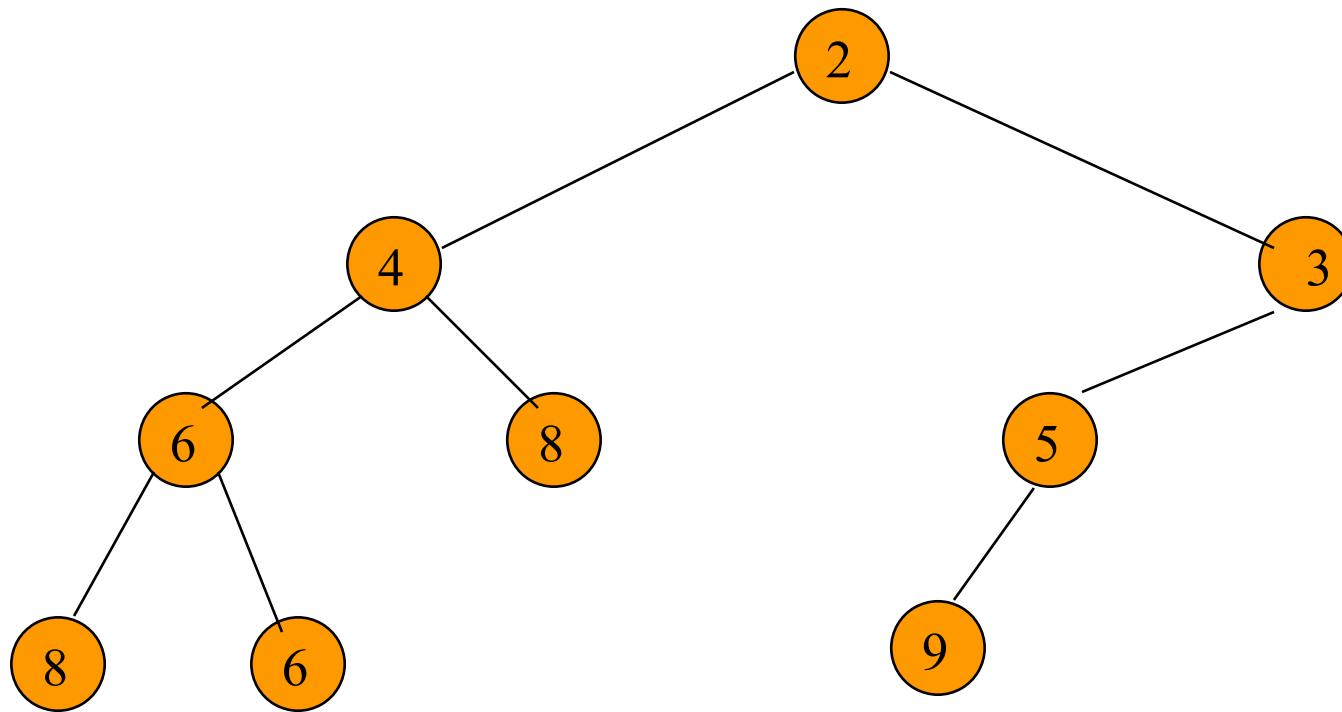


Create a single node min leftist tree.

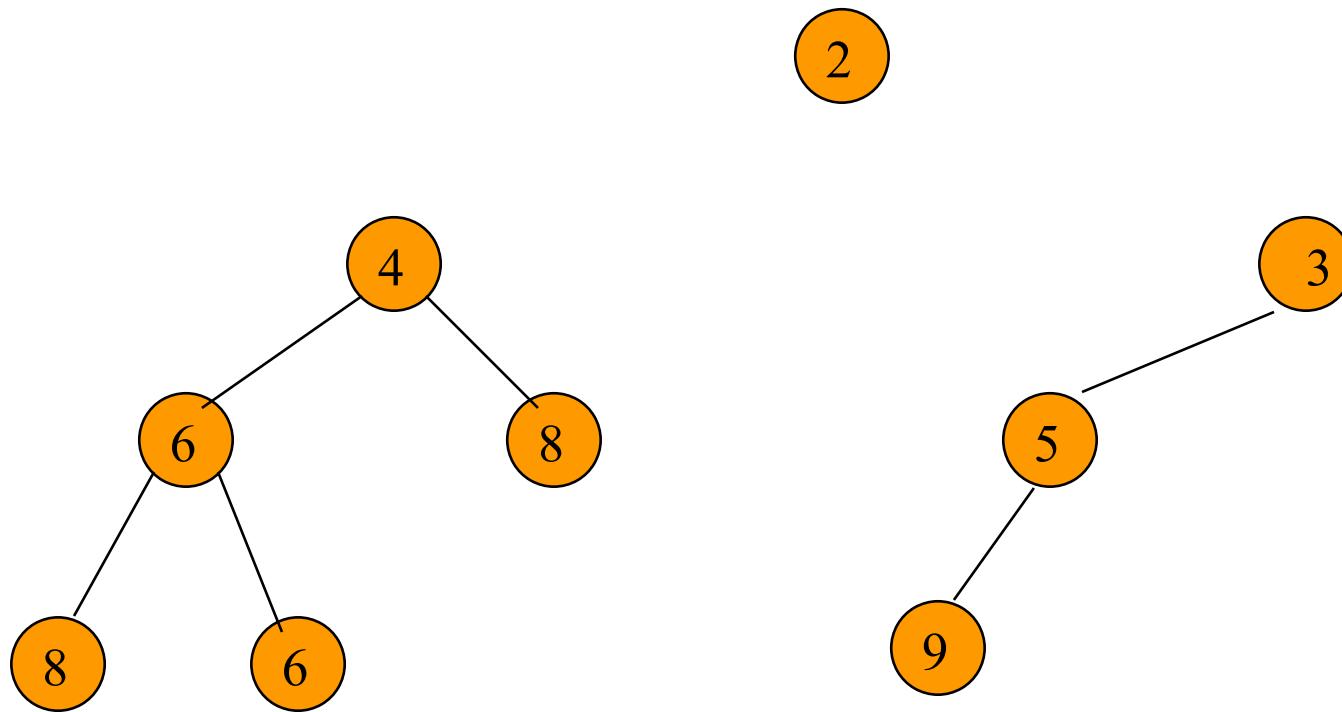


Meld the two min leftist trees.

Remove Min

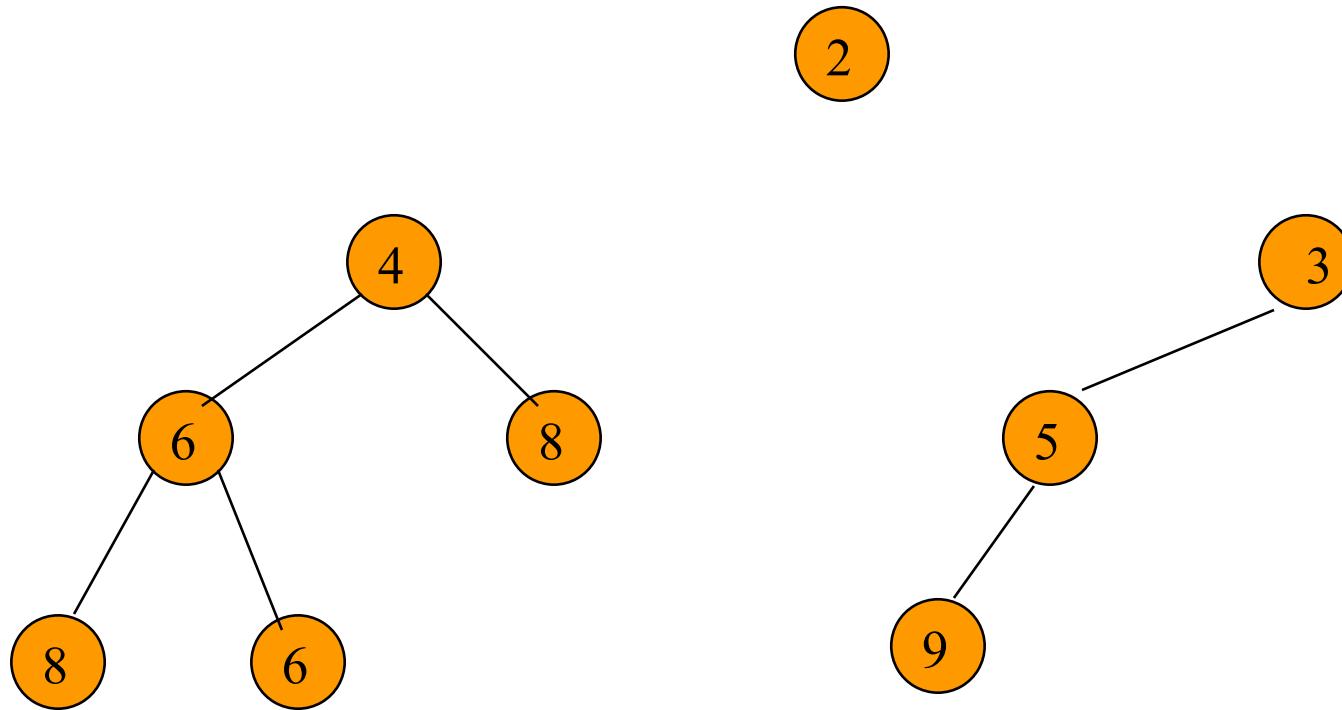


Remove Min



Remove the root.

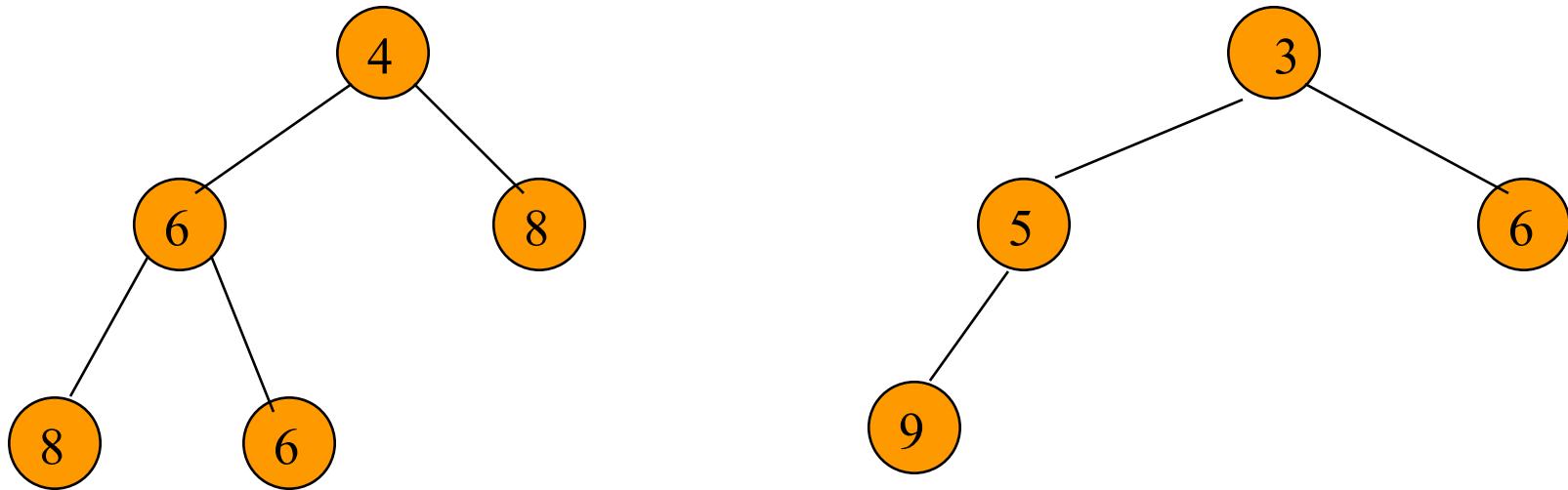
Remove Min



Remove the root.

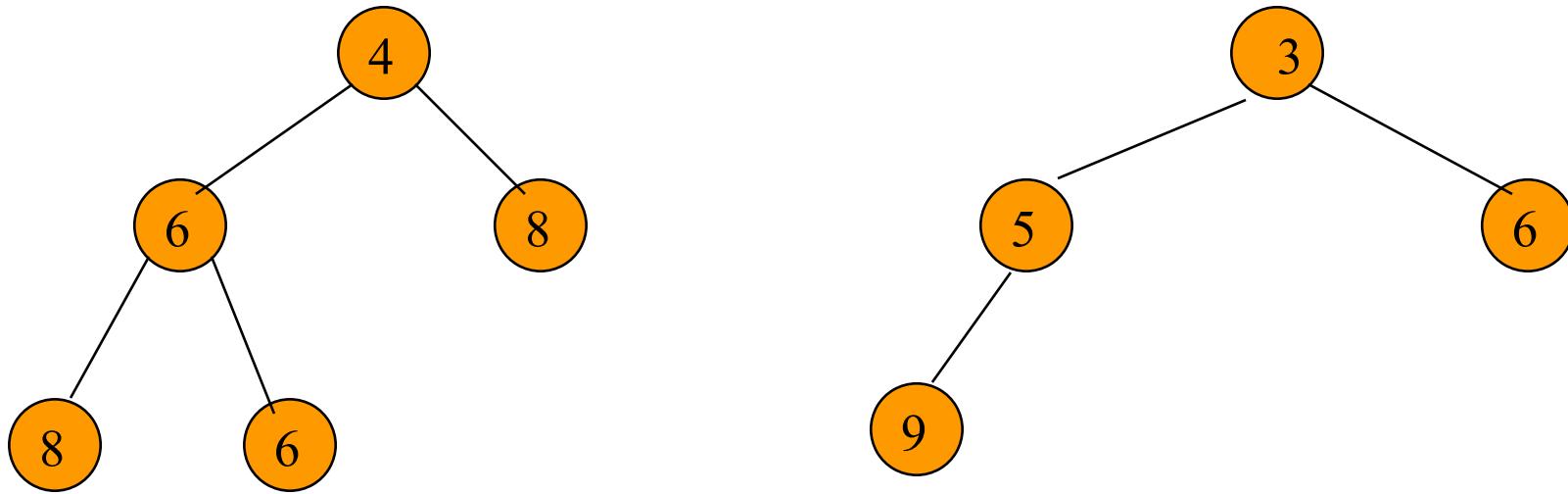
Meld the two subtrees.

Meld Two Min Leftist Trees



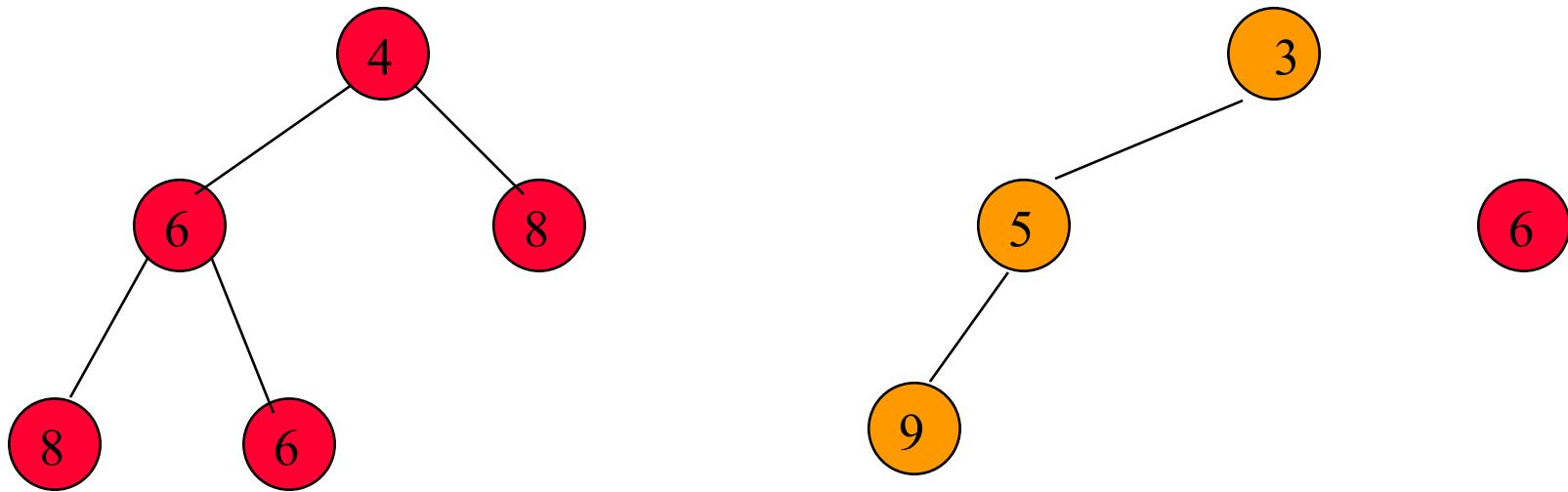
Traverse only the rightmost paths so as to get logarithmic performance.

Meld Two Min Leftist Trees



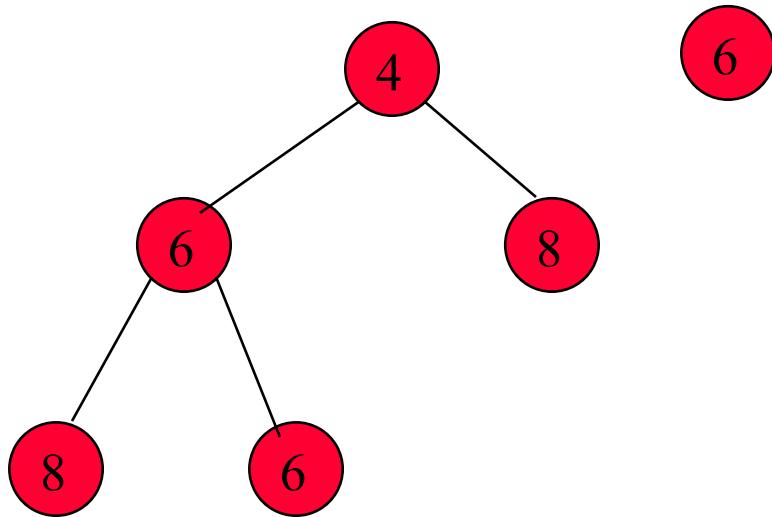
Meld right subtree of tree with smaller root and all of other tree.

Meld Two Min Leftist Trees



Meld right subtree of tree with smaller root and all of other tree.

Meld Two Min Leftist Trees



Meld right subtree of tree with smaller root and all of other tree.

Meld Two Min Leftist Trees

8

6

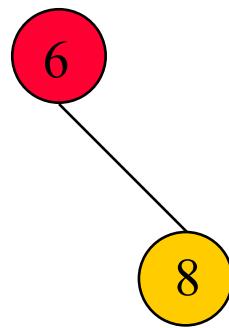
Meld right subtree of tree with smaller root and all of other tree.

Right subtree of 6 is empty. So, result of melding right subtree of tree with smaller root and other tree is the other tree.

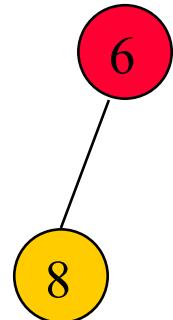
Meld Two Min Leftist Trees



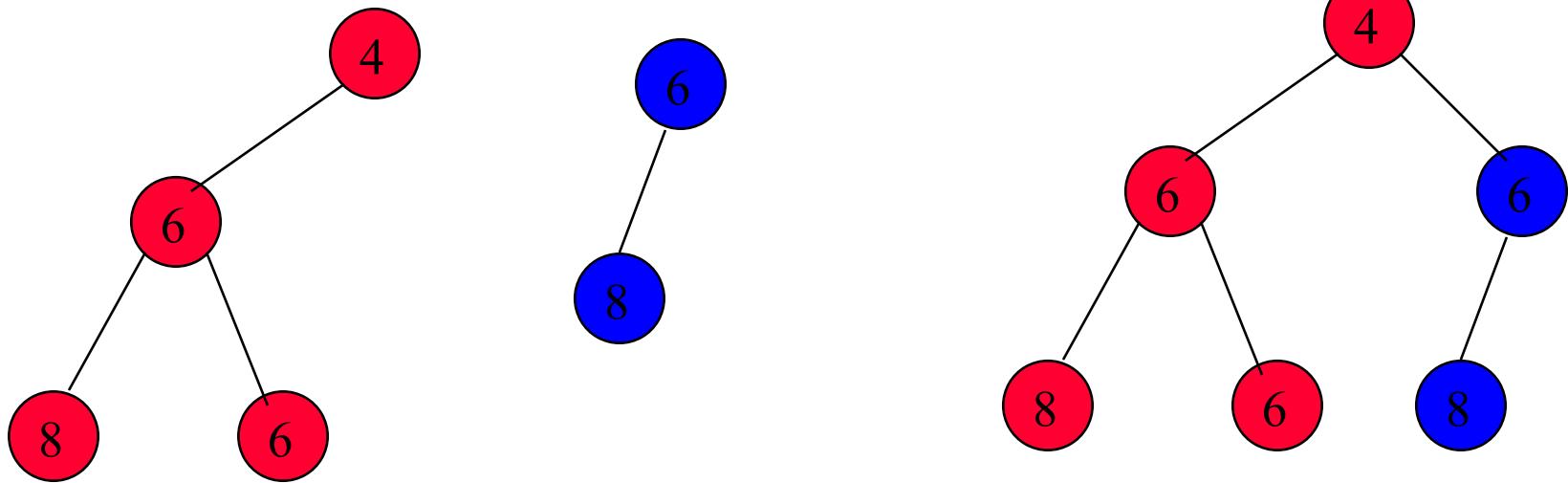
Make melded subtree right subtree of smaller root.



Swap left and right subtree if $s(\text{left}) < s(\text{right})$.



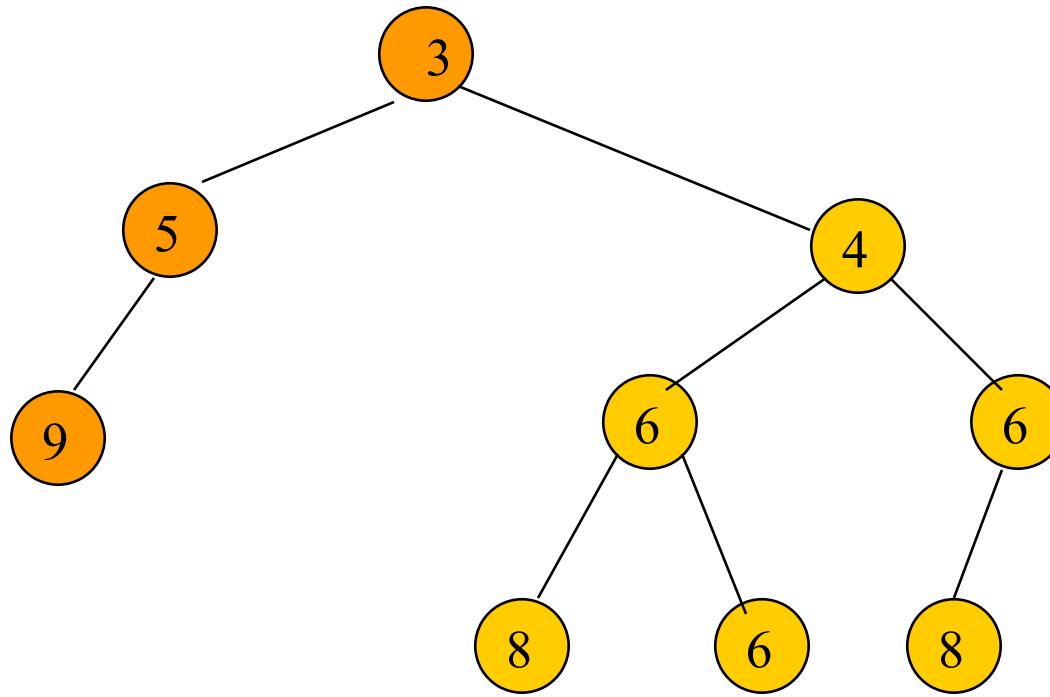
Meld Two Min Leftist Trees



Make melded subtree right subtree of smaller root.

Swap left and right subtree if $s(\text{left}) < s(\text{right})$.

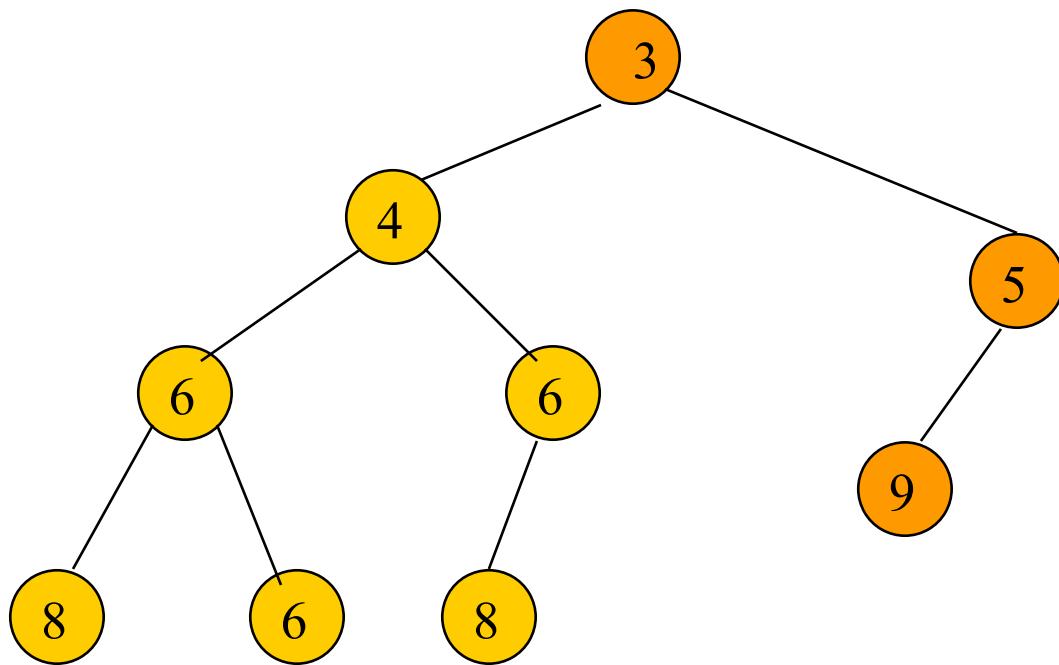
Meld Two Min Leftist Trees



Make melded subtree right subtree of smaller root.

Swap left and right subtree if $s(\text{left}) < s(\text{right})$.

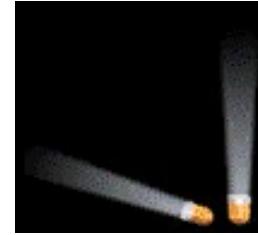
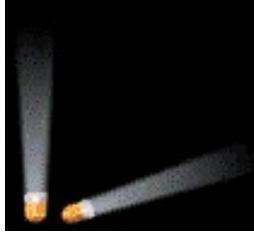
Meld Two Min Leftist Trees



Initializing In $O(n)$ Time

- create n single node min leftist trees and place them in a FIFO queue
- repeatedly remove two min leftist trees from the FIFO queue, meld them, and put the resulting min leftist tree into the FIFO queue
- the process terminates when only 1 min leftist tree remains in the FIFO queue
- analysis is the same as for heap initialization

Tournament Trees



Winner trees.

Loser Trees.

Winner Trees

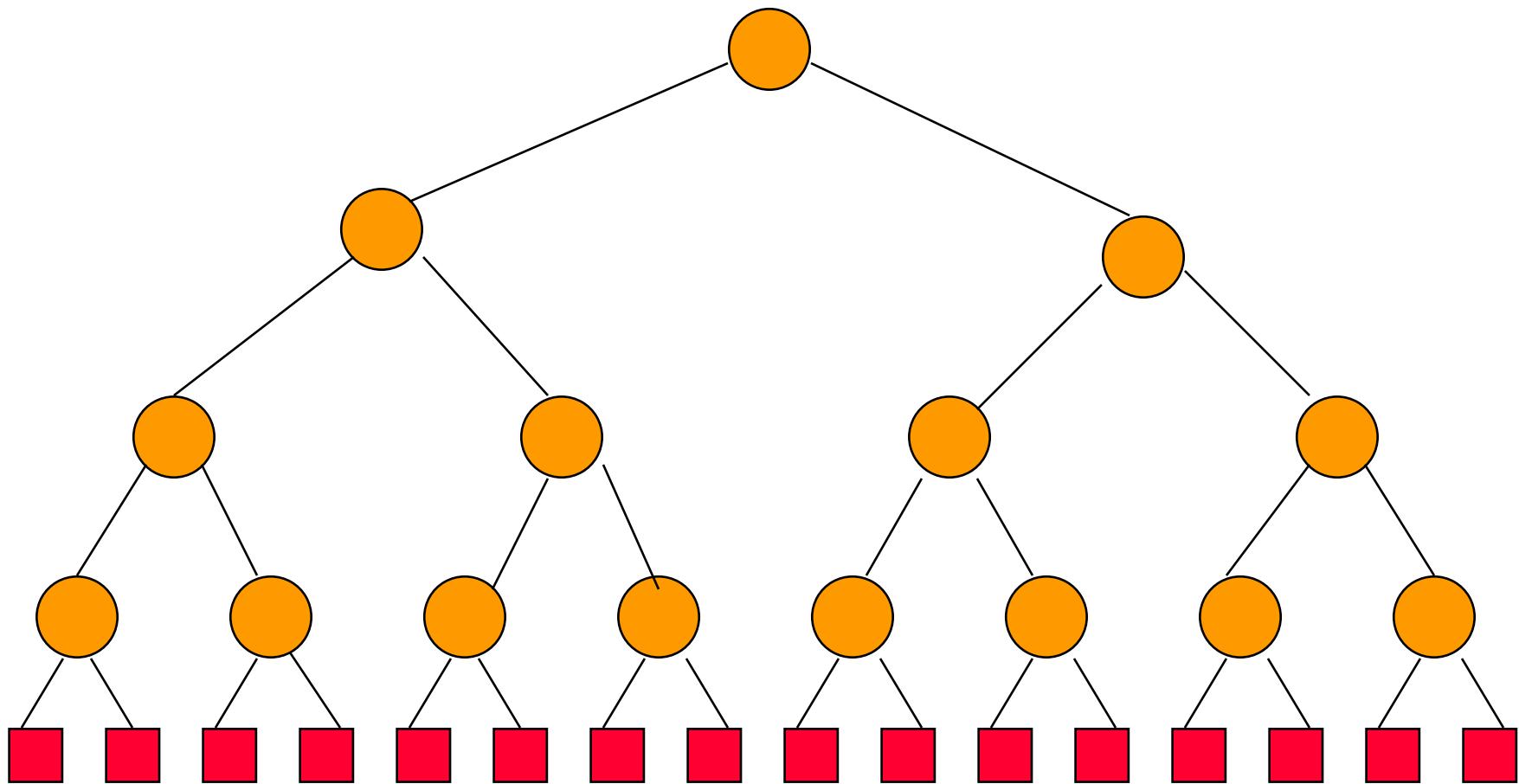
Complete binary tree with n external nodes and $n - 1$ internal nodes.

External nodes represent tournament players.

Each internal node represents a match played between its two children; the winner of the match is stored at the internal node.

Root has overall winner.

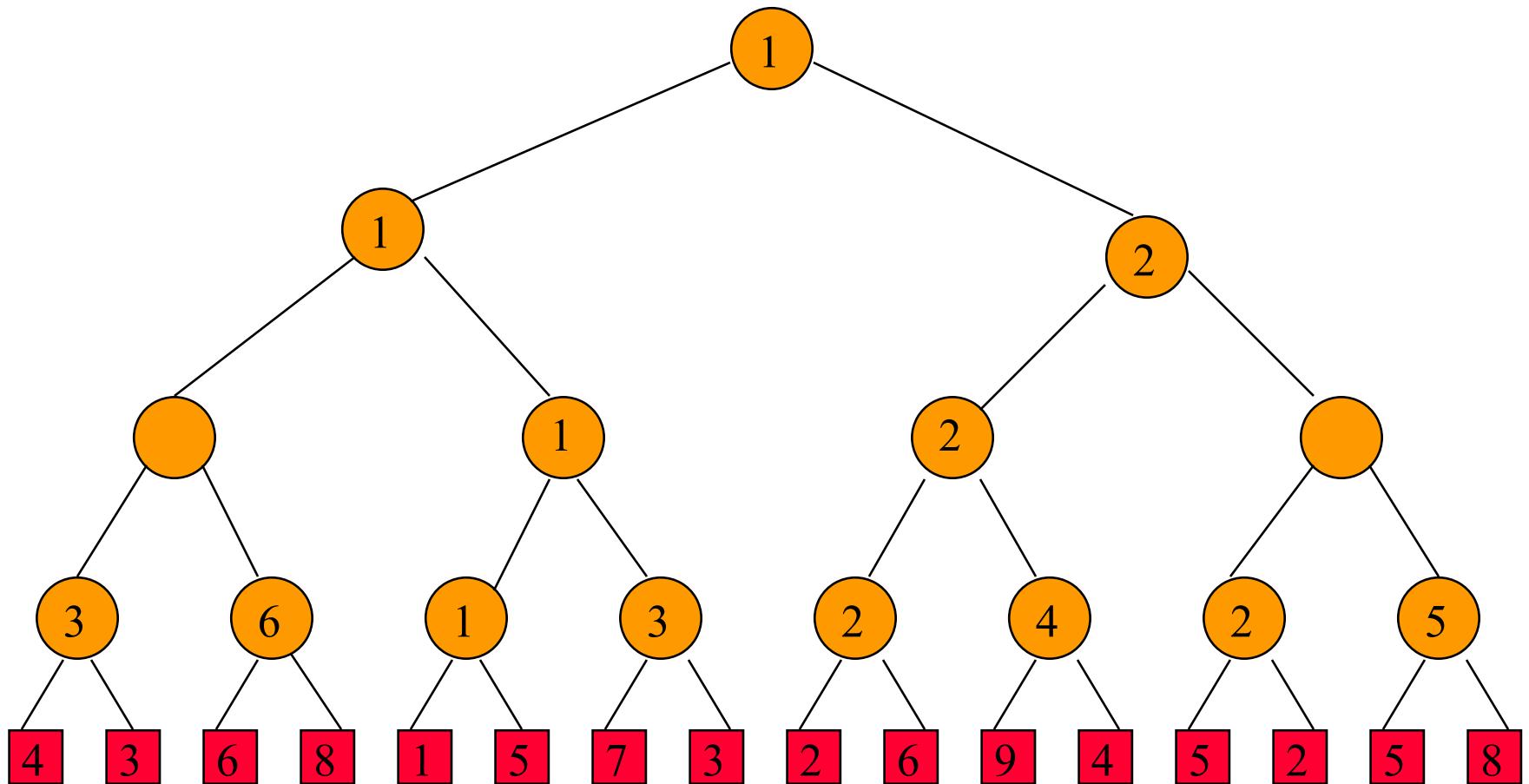
Winner Tree For 16 Players



■ player

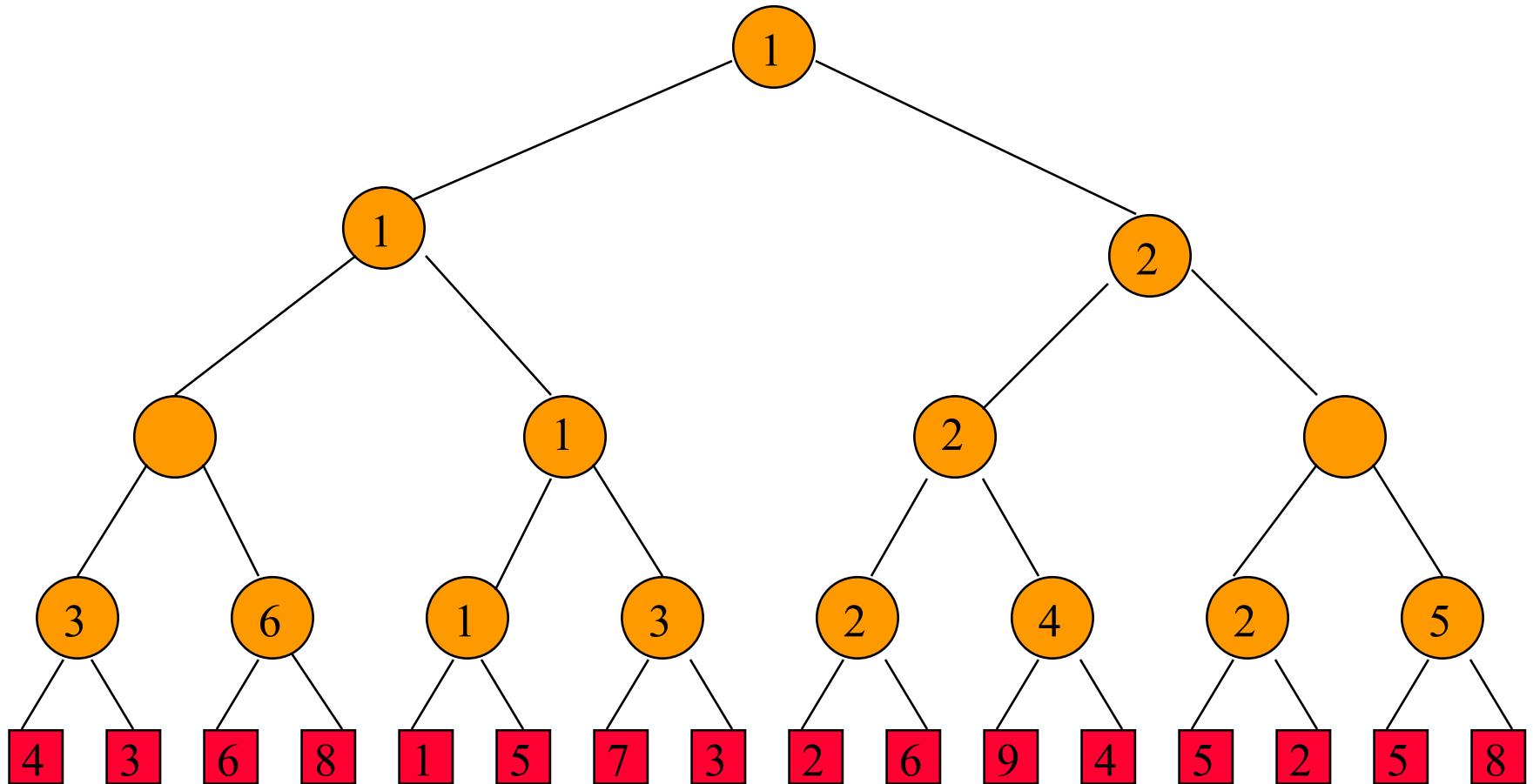
● match node

Winner Tree For 16 Players



Smaller element wins => min winner tree.

Winner Tree For 16 Players



height is $\log_2 n$ (excludes player level)

Complexity Of Initialize

- $O(1)$ time to play match at each match node.
- $n - 1$ match nodes.
- $O(n)$ time to initialize n player winner tree.

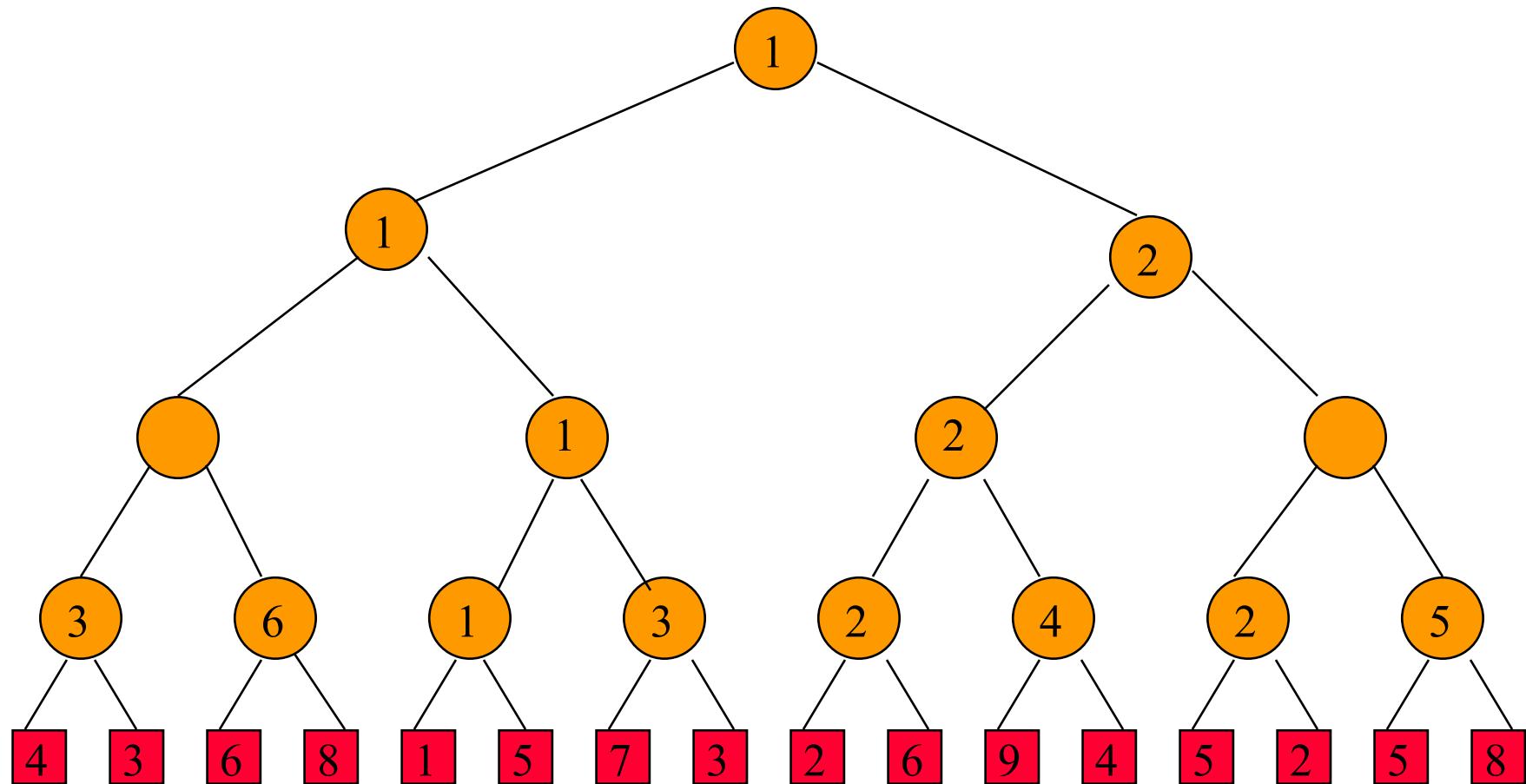
Applications

Sorting.

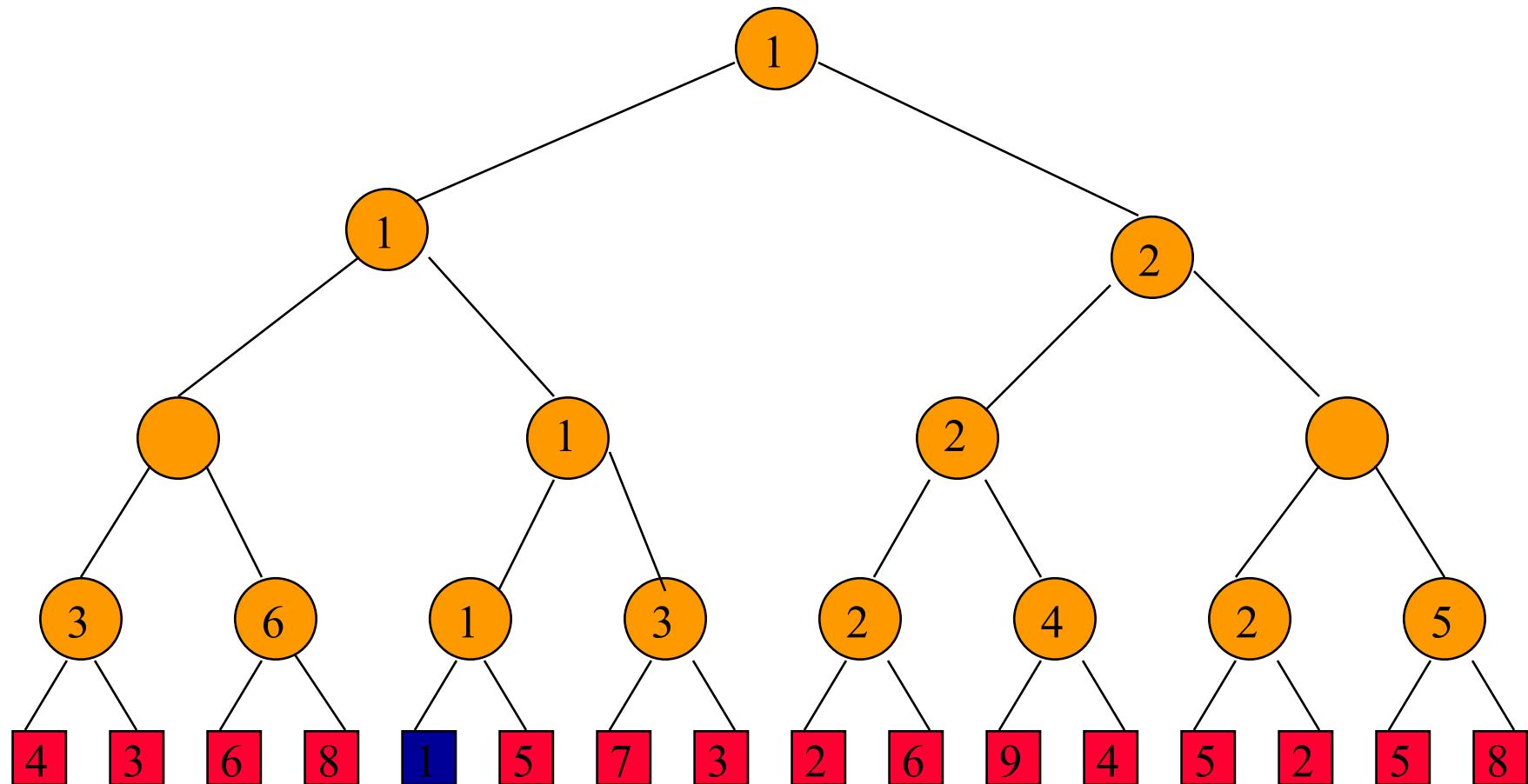
Put elements to be sorted into a winner tree.

Repeatedly extract the winner and replace by a large value.

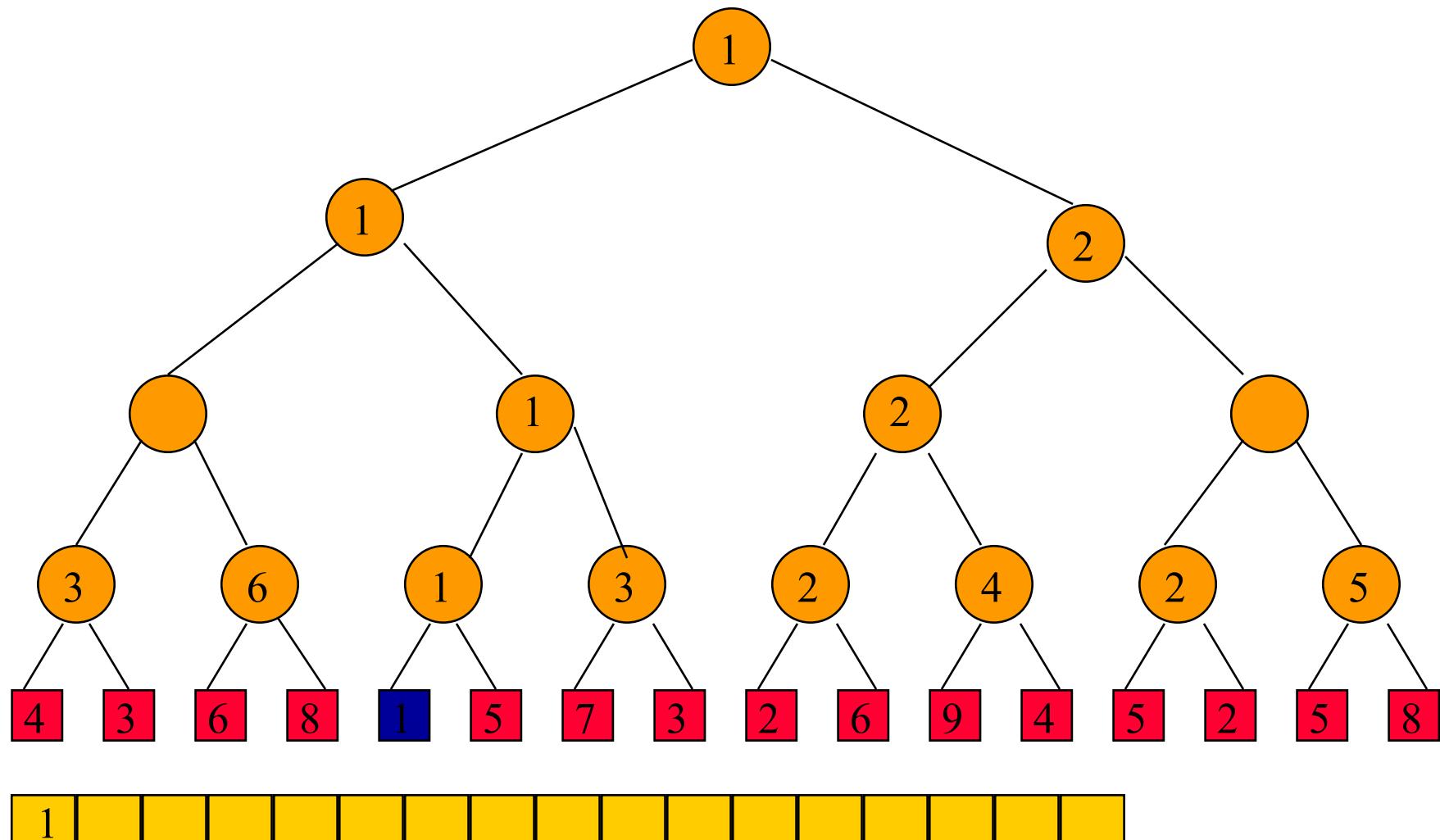
Sort 16 Numbers



Sort 16 Numbers

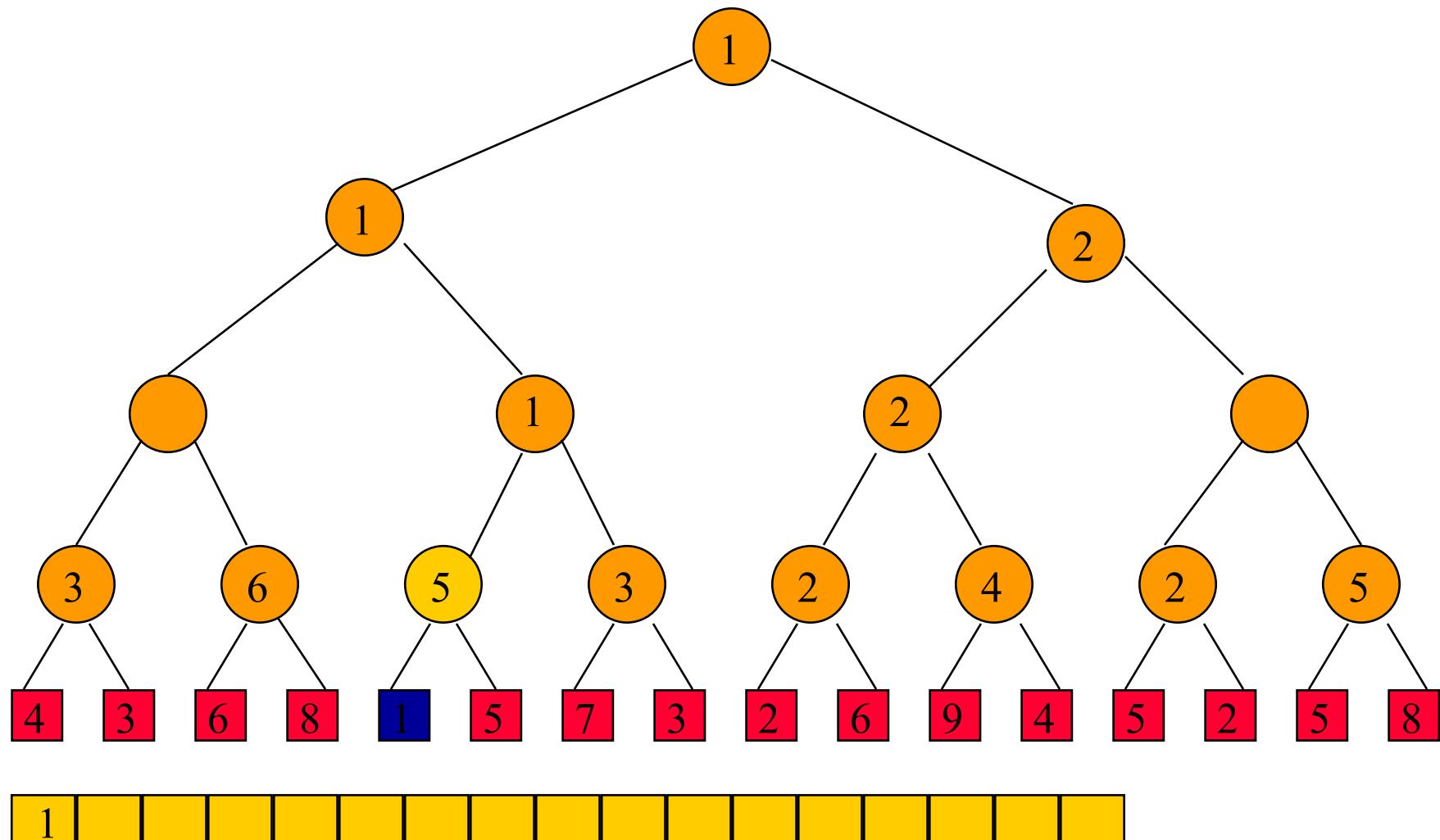


Sort 16 Numbers



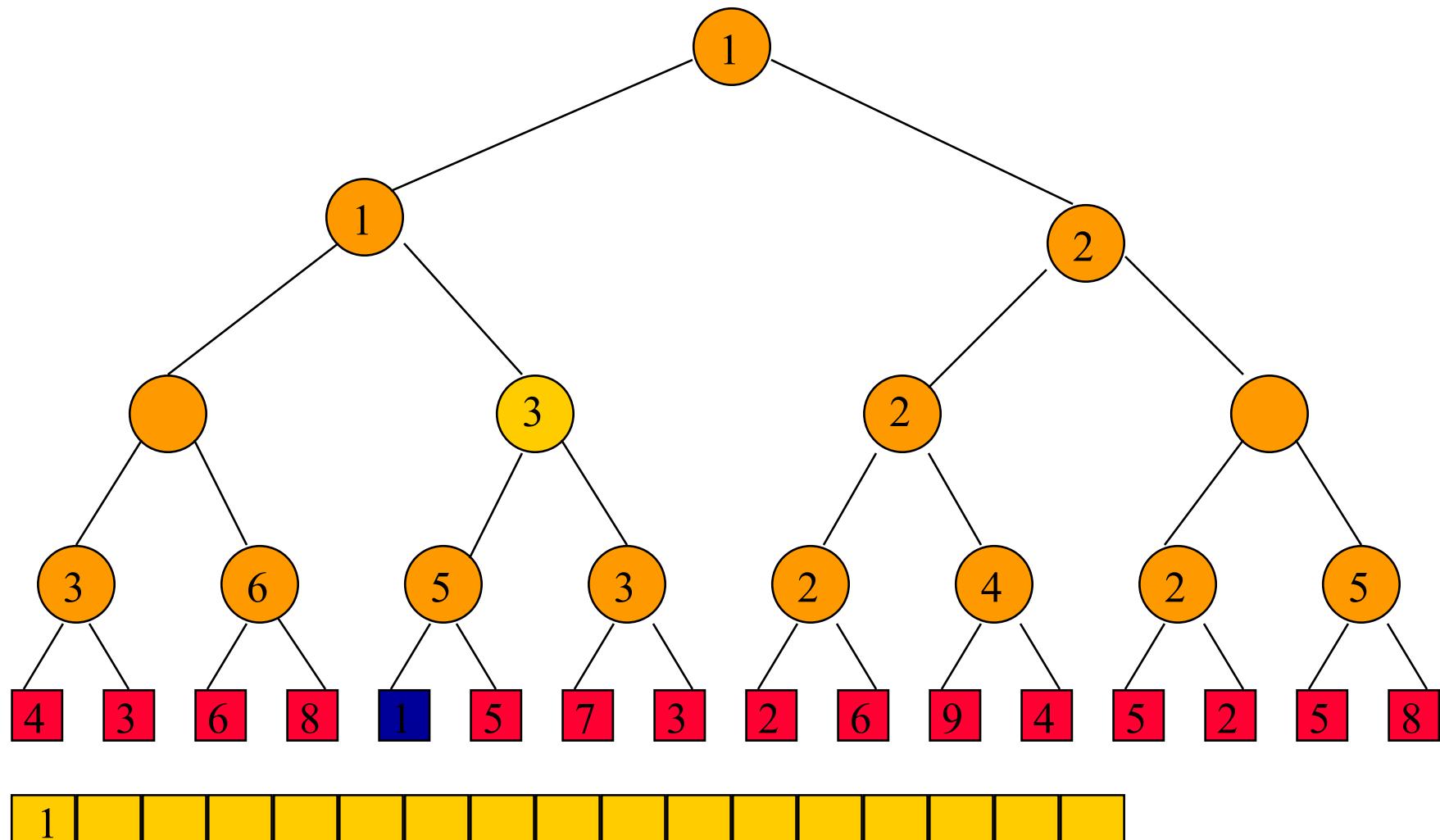
Sorted array.

Sort 16 Numbers



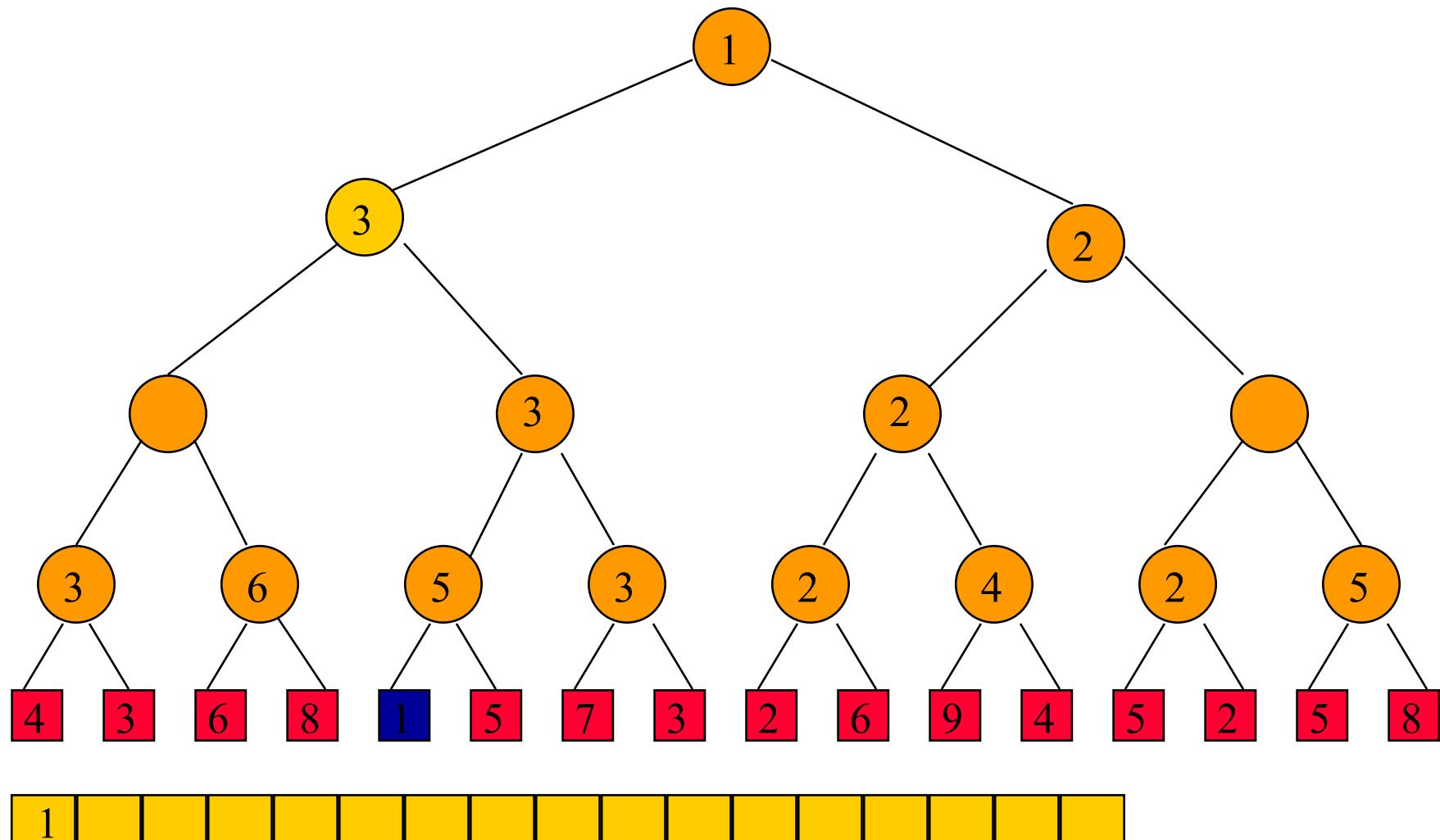
Sorted array.

Sort 16 Numbers



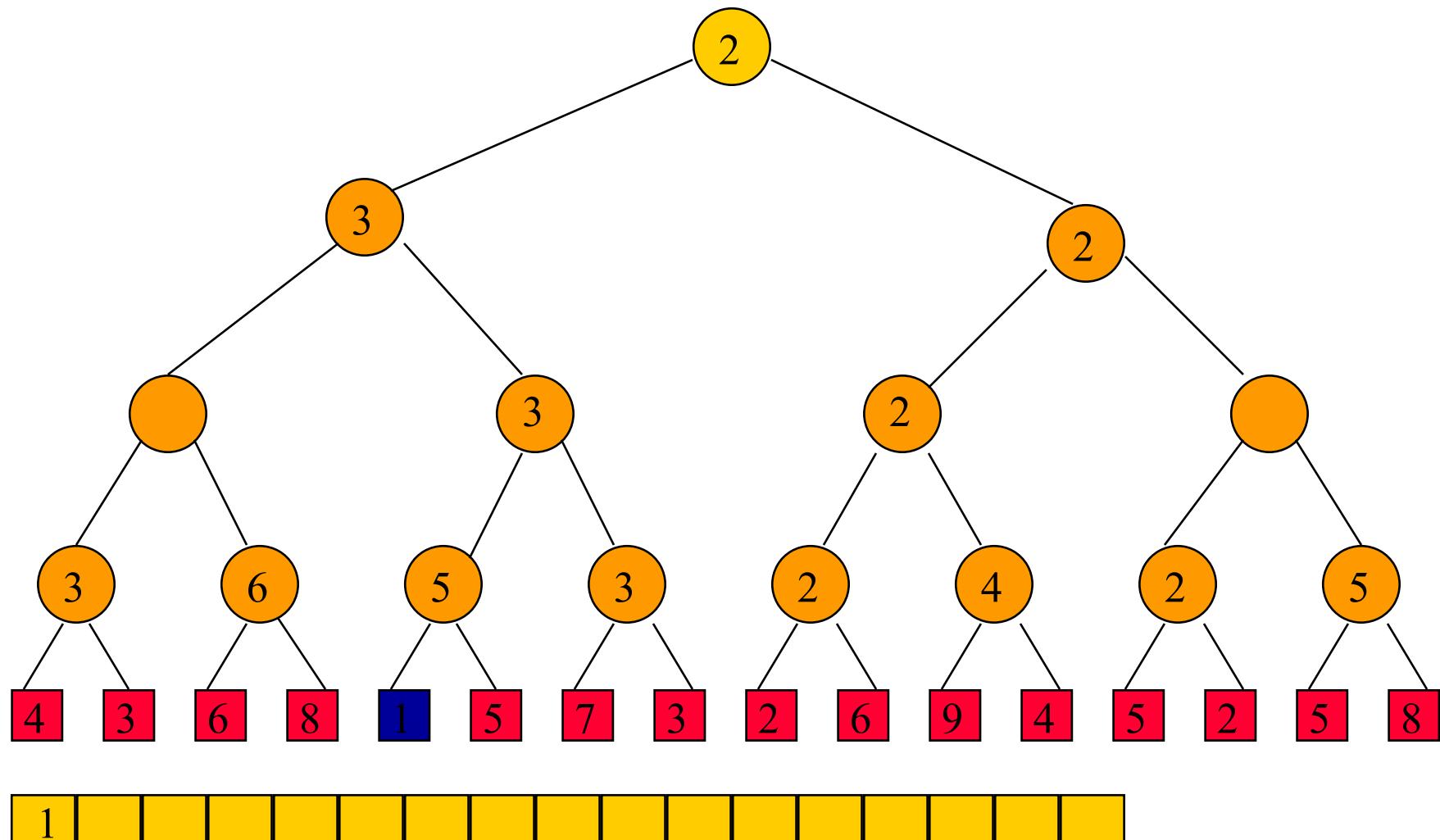
Sorted array.

Sort 16 Numbers



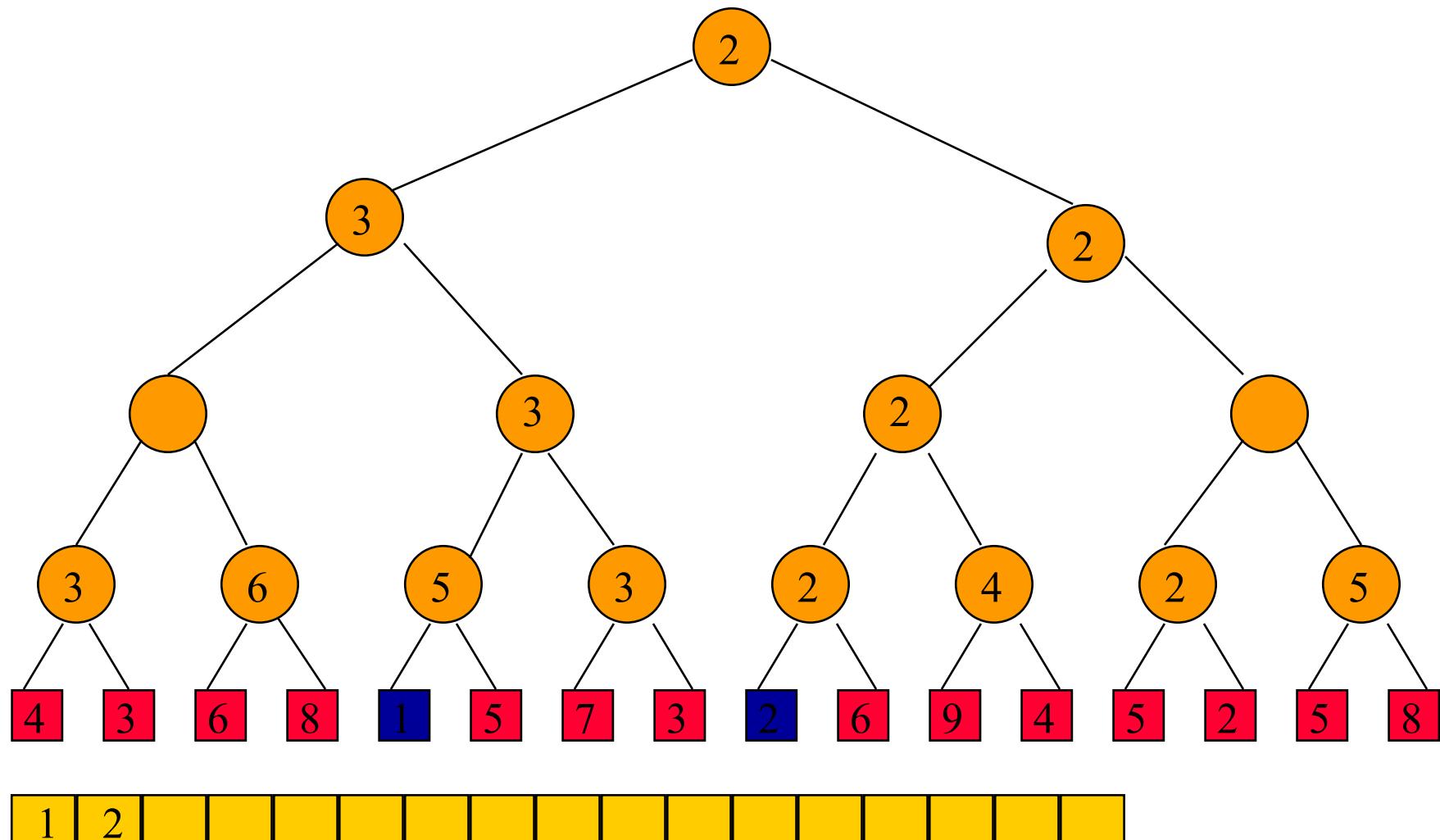
Sorted array.

Sort 16 Numbers



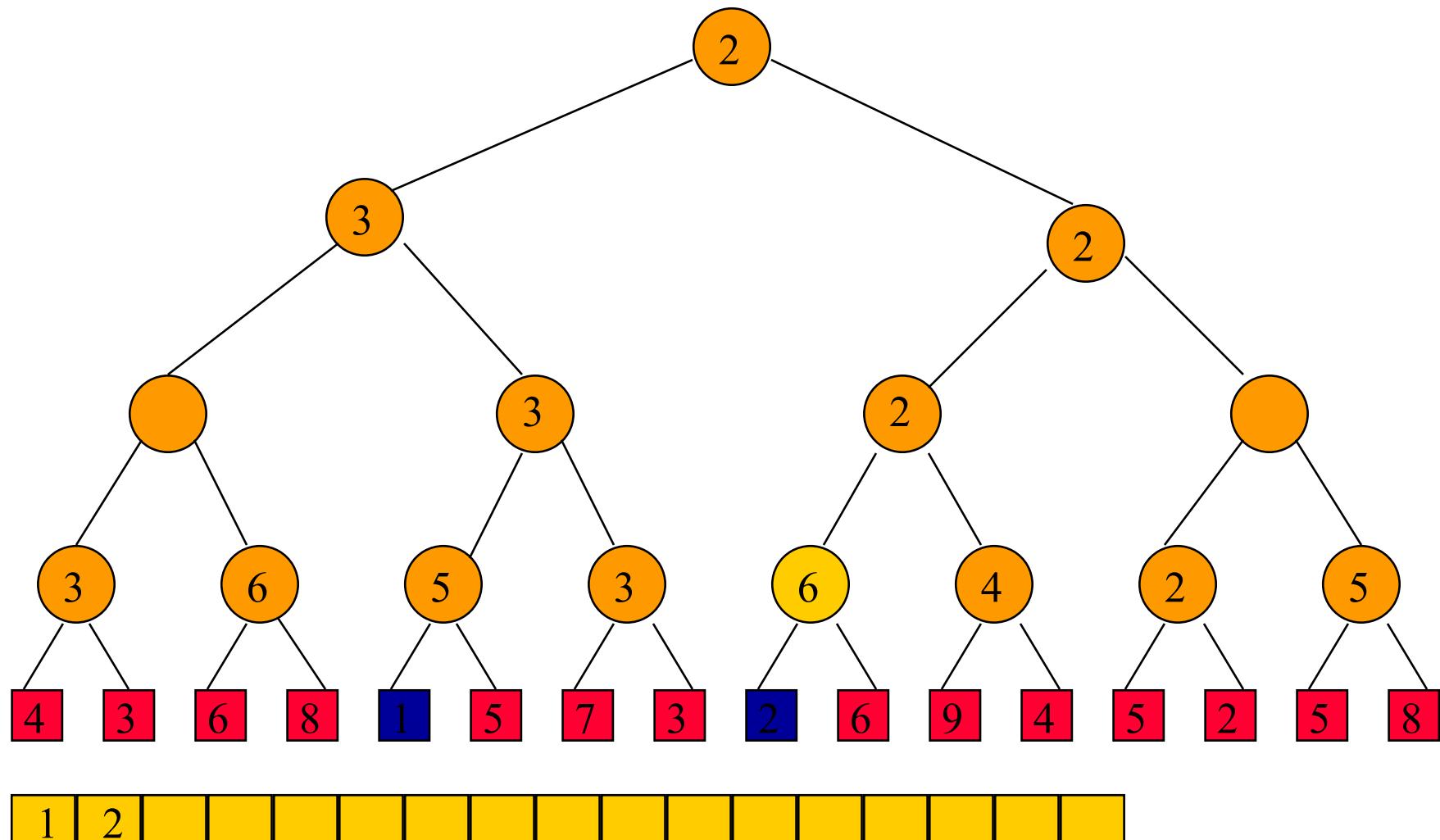
Sorted array.

Sort 16 Numbers



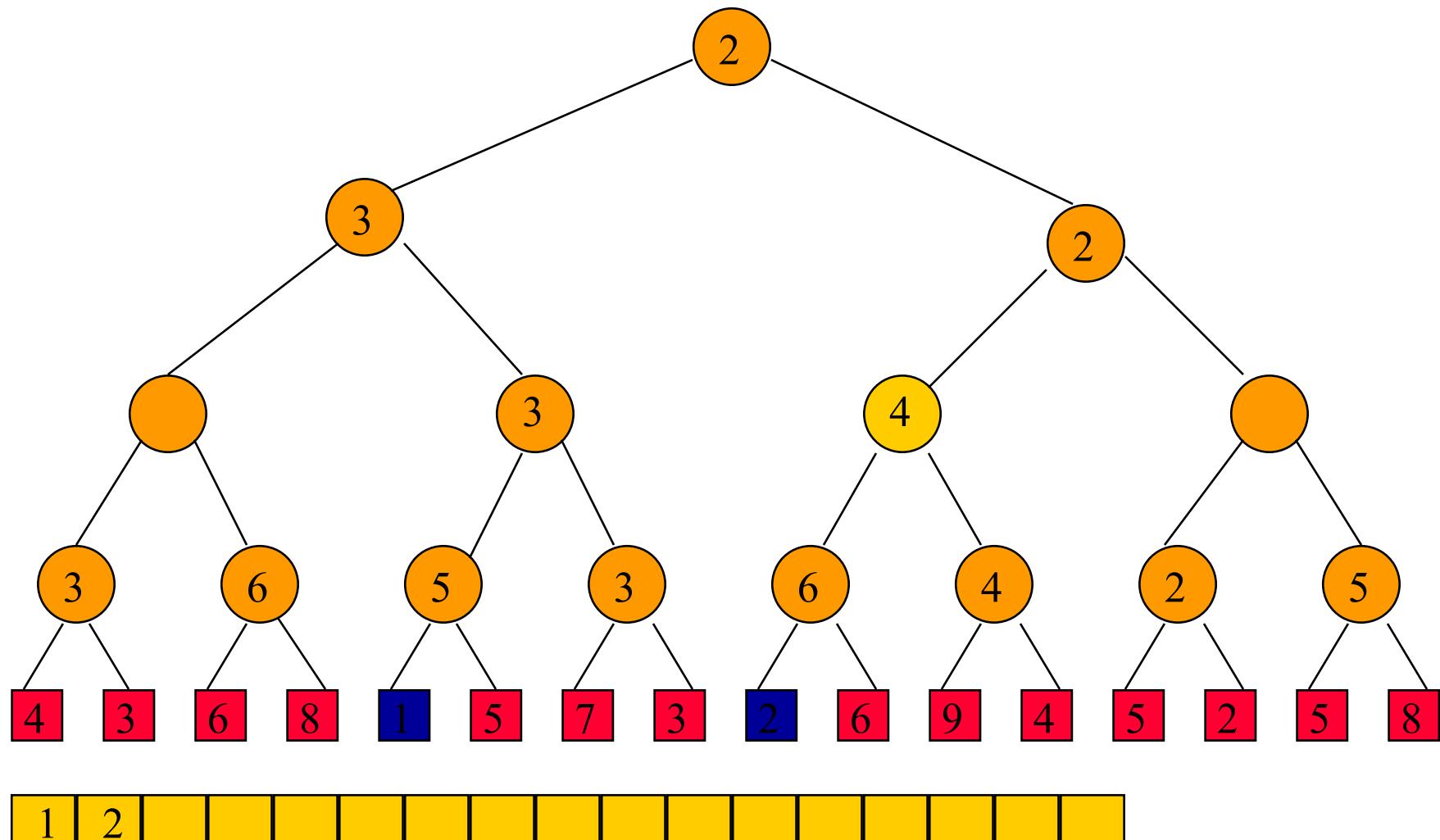
Sorted array.

Sort 16 Numbers



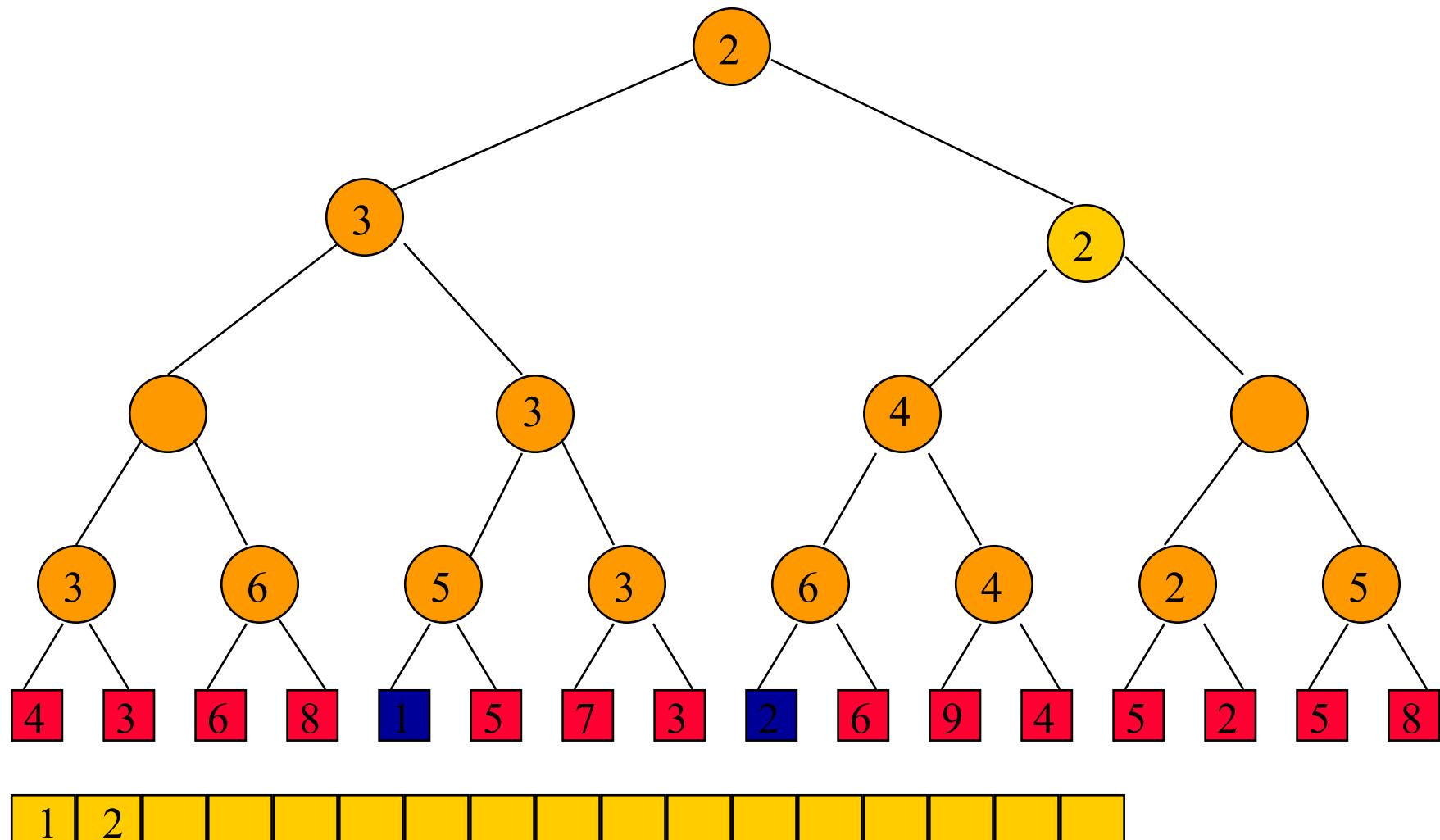
Sorted array.

Sort 16 Numbers



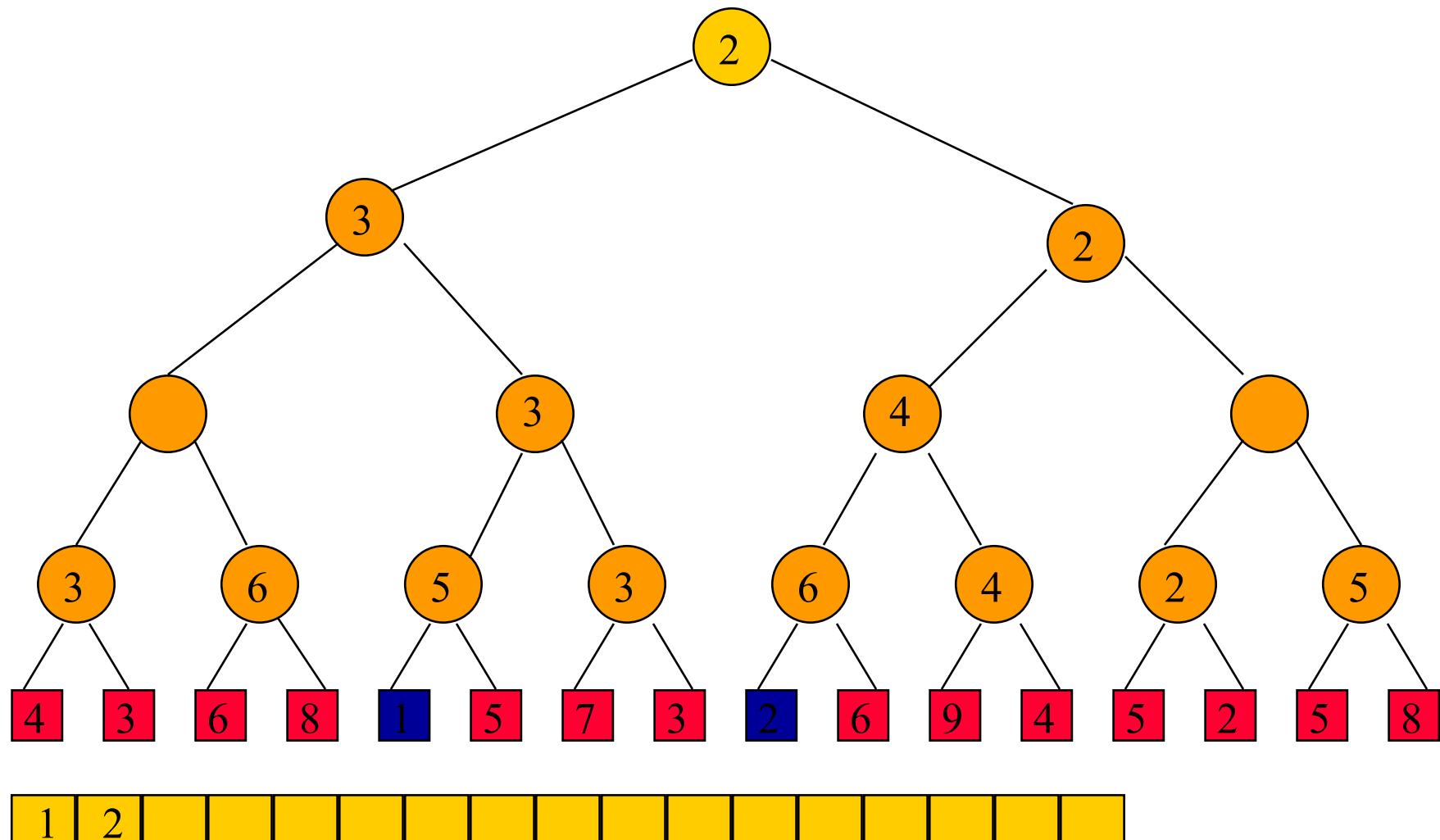
Sorted array.

Sort 16 Numbers



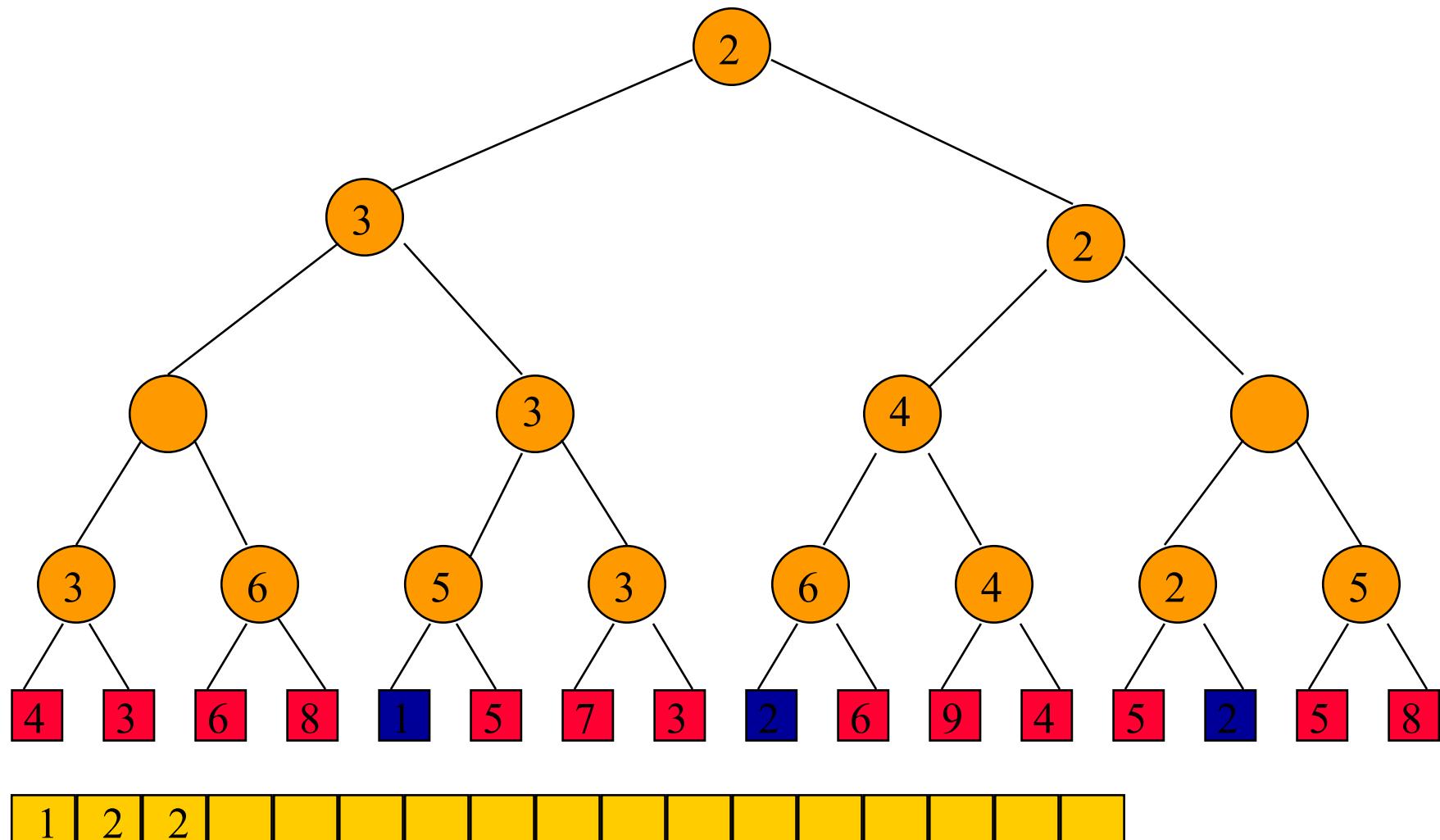
Sorted array.

Sort 16 Numbers



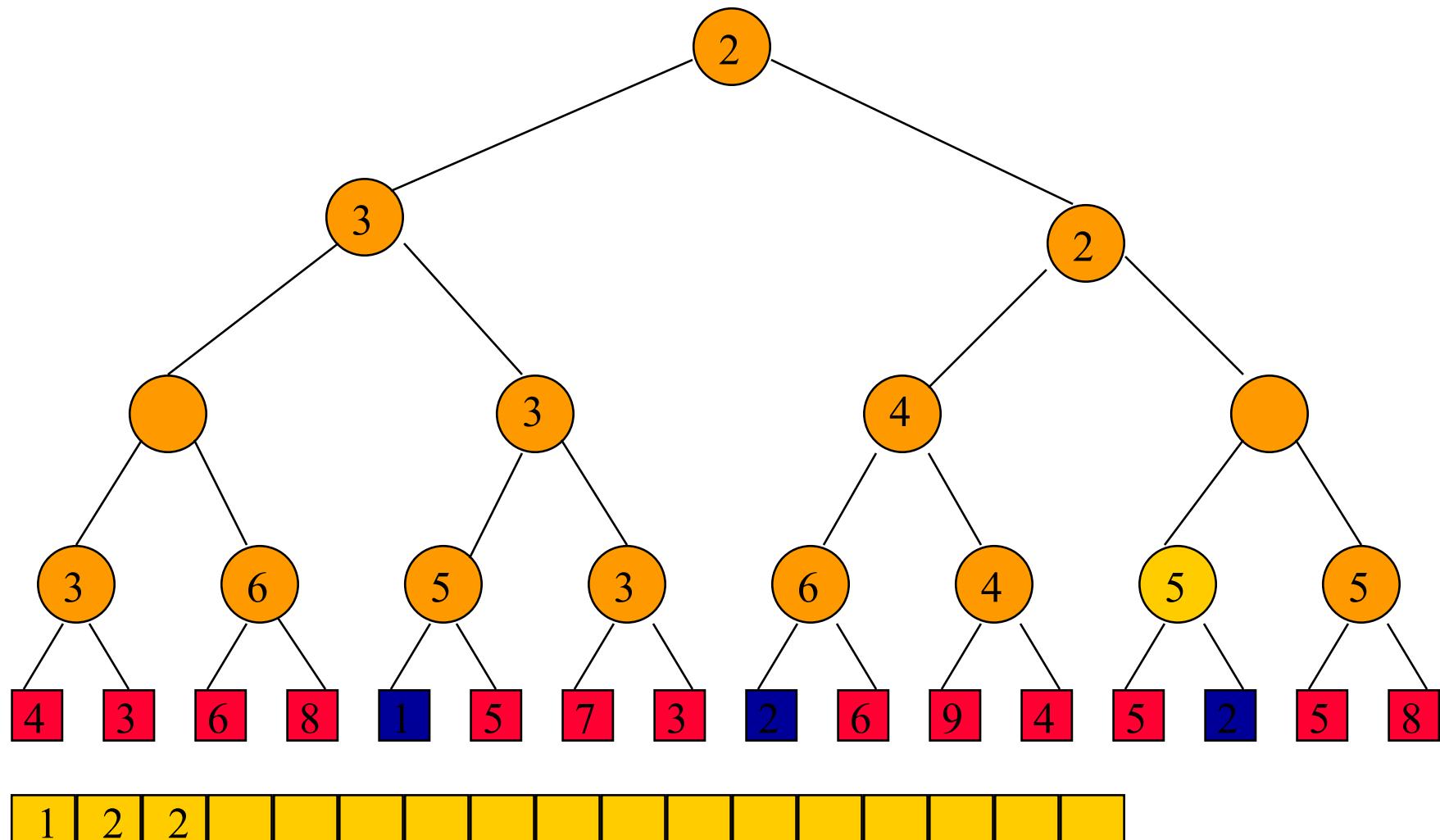
Sorted array.

Sort 16 Numbers



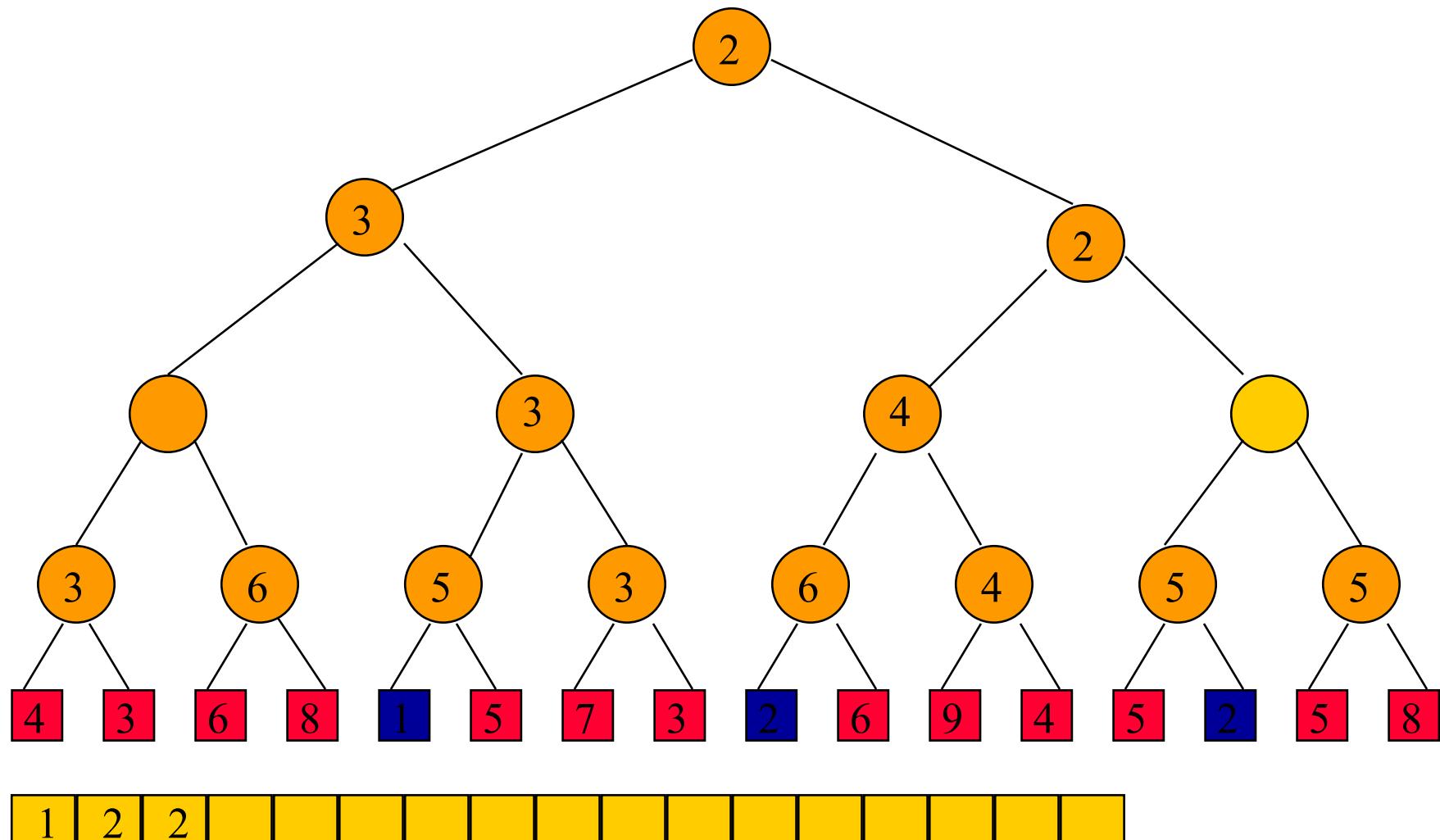
Sorted array.

Sort 16 Numbers



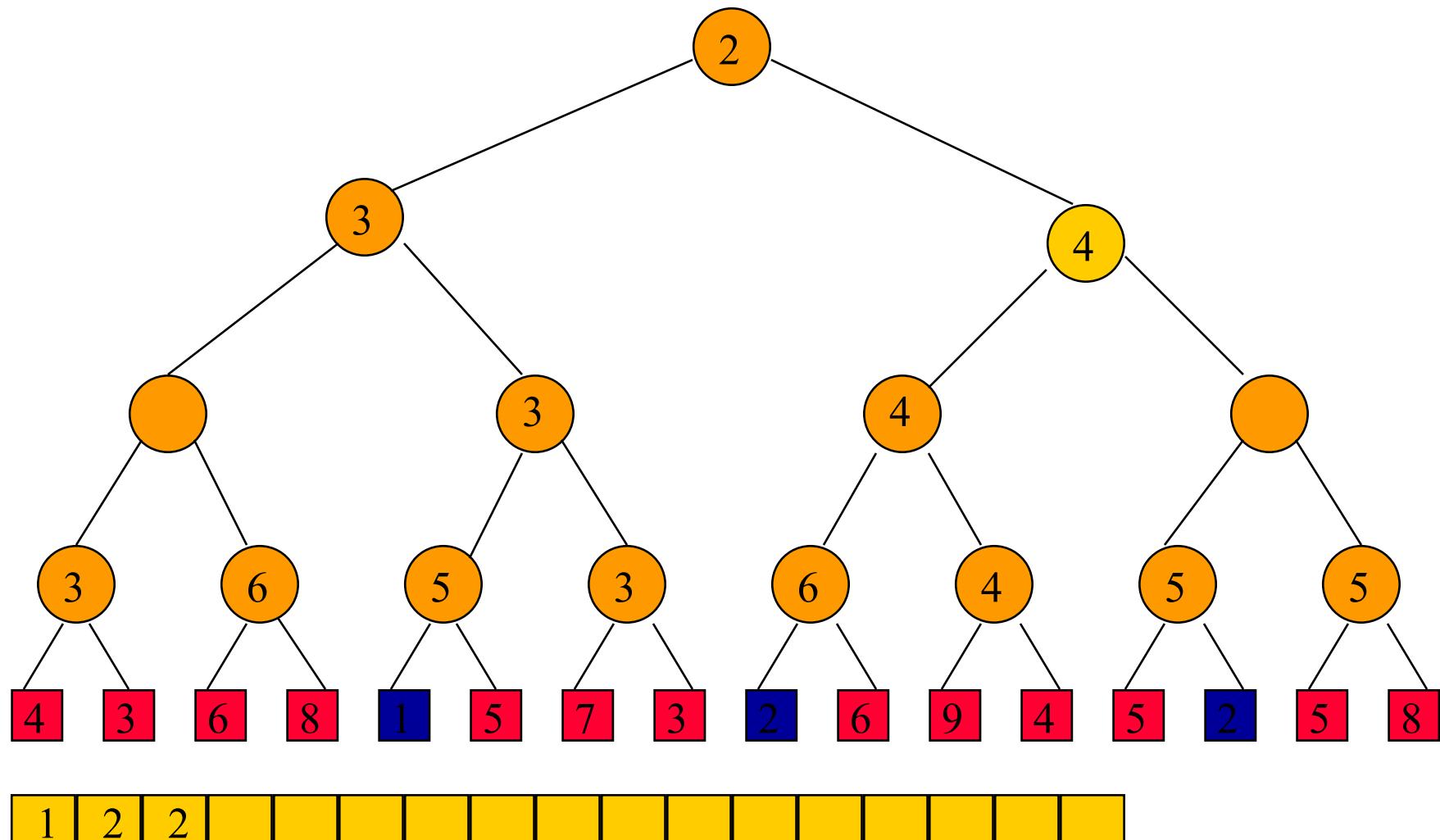
Sorted array.

Sort 16 Numbers



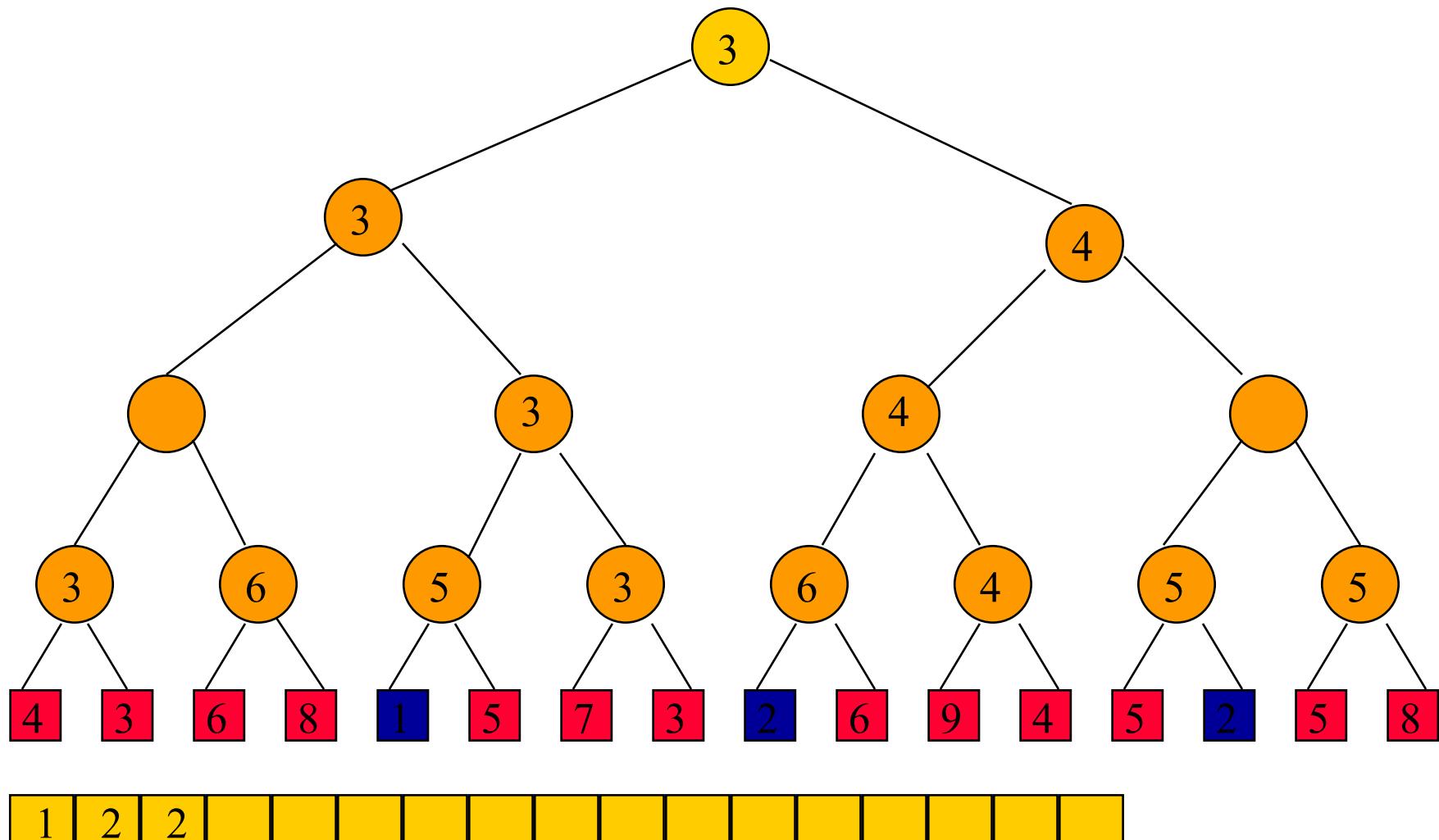
Sorted array.

Sort 16 Numbers



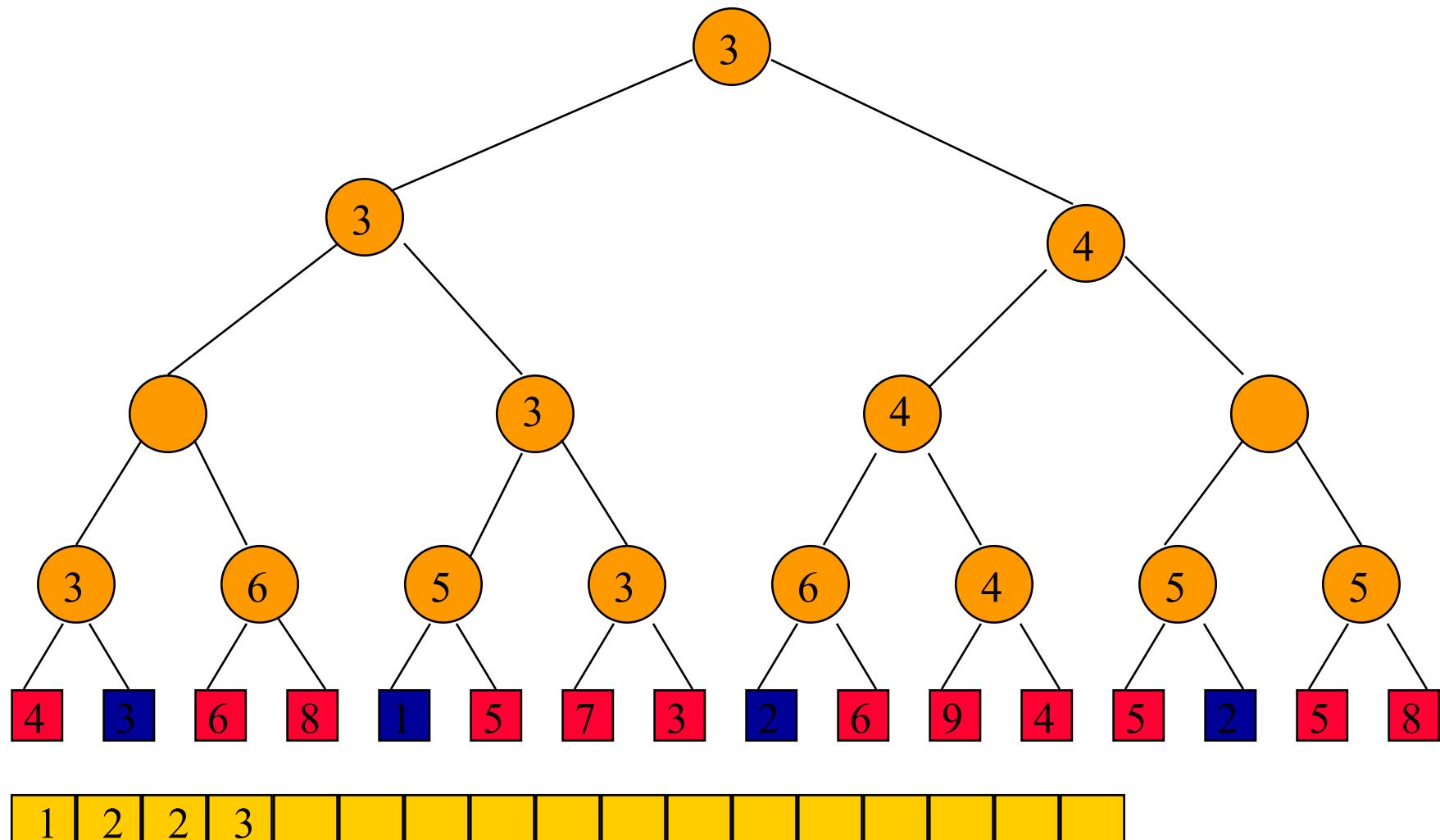
Sorted array.

Sort 16 Numbers



Sorted array.

Sort 16 Numbers



Sorted array.



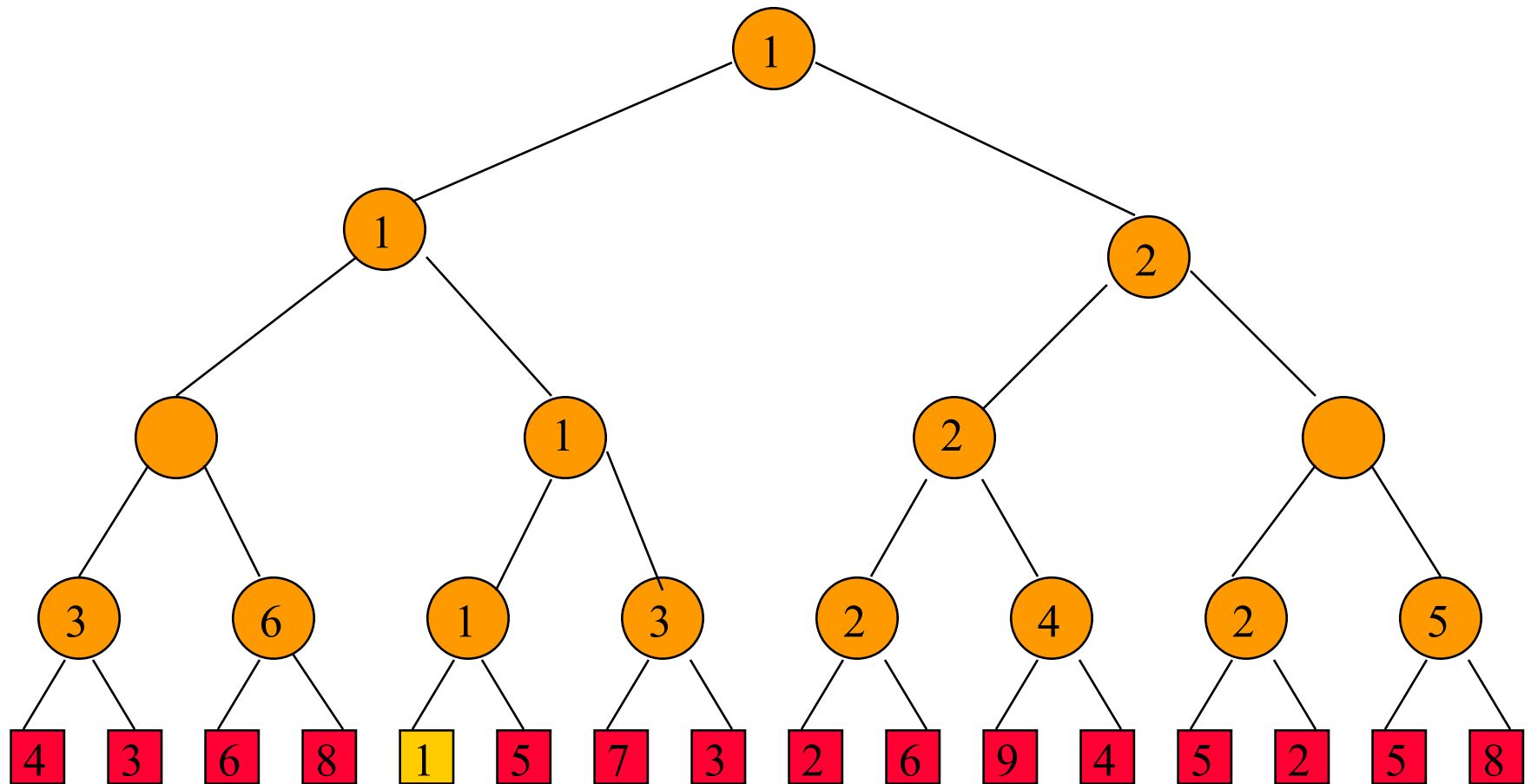
Time To Sort

- Initialize winner tree.
 - $O(n)$ time
- Remove winner and replay.
 - $O(\log n)$ time
- Remove winner and replay n times.
 - $O(n \log n)$ time
- Total sort time is $O(n \log n)$.
- Actually $\Theta(n \log n)$.

Winner Tree Operations

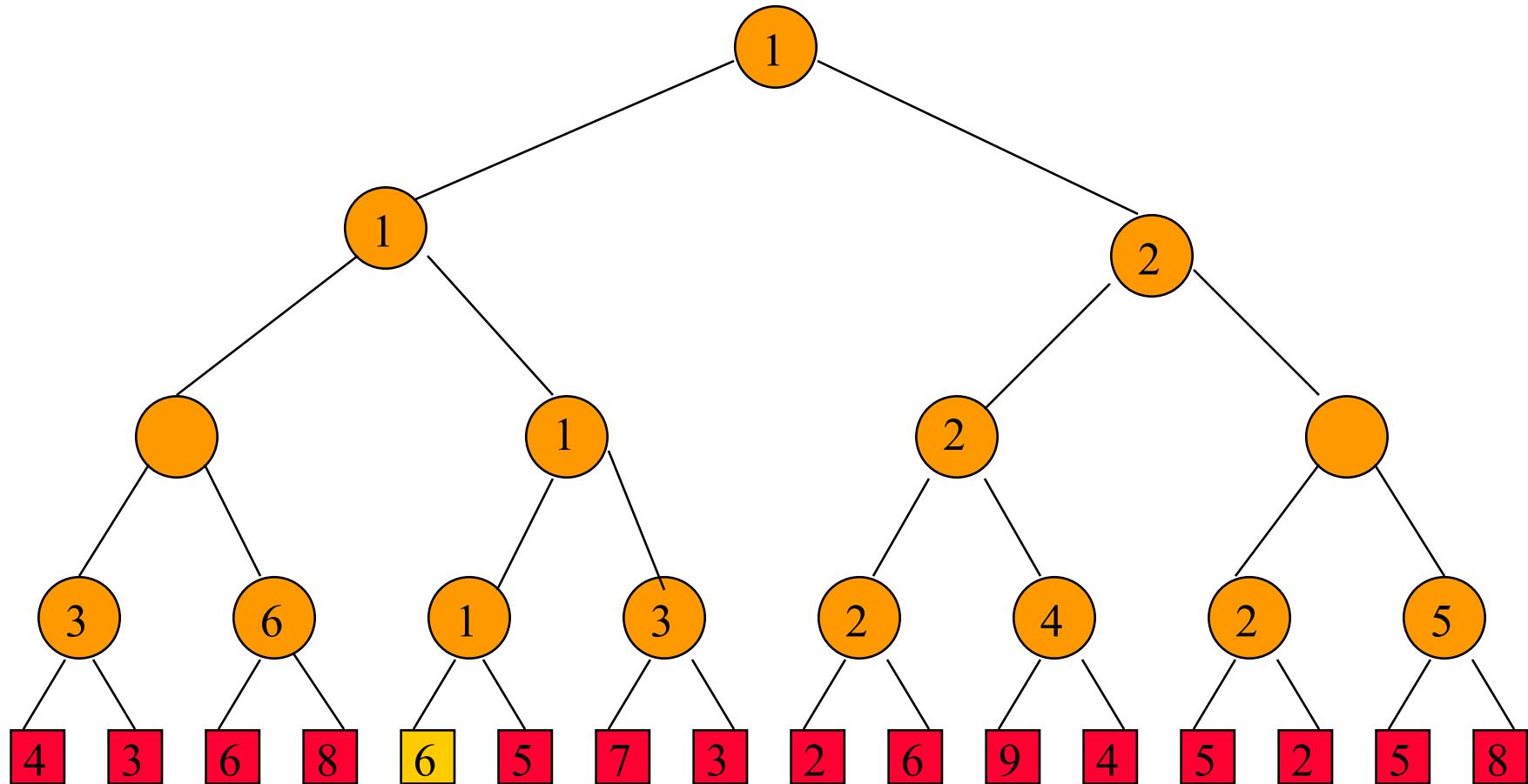
- Initialize
 - $O(n)$ time
- Get winner
 - $O(1)$ time
- Remove/replace winner and replay
 - $O(\log n)$ time
 - more precisely $\Theta(\log n)$

Replace Winner And Replay



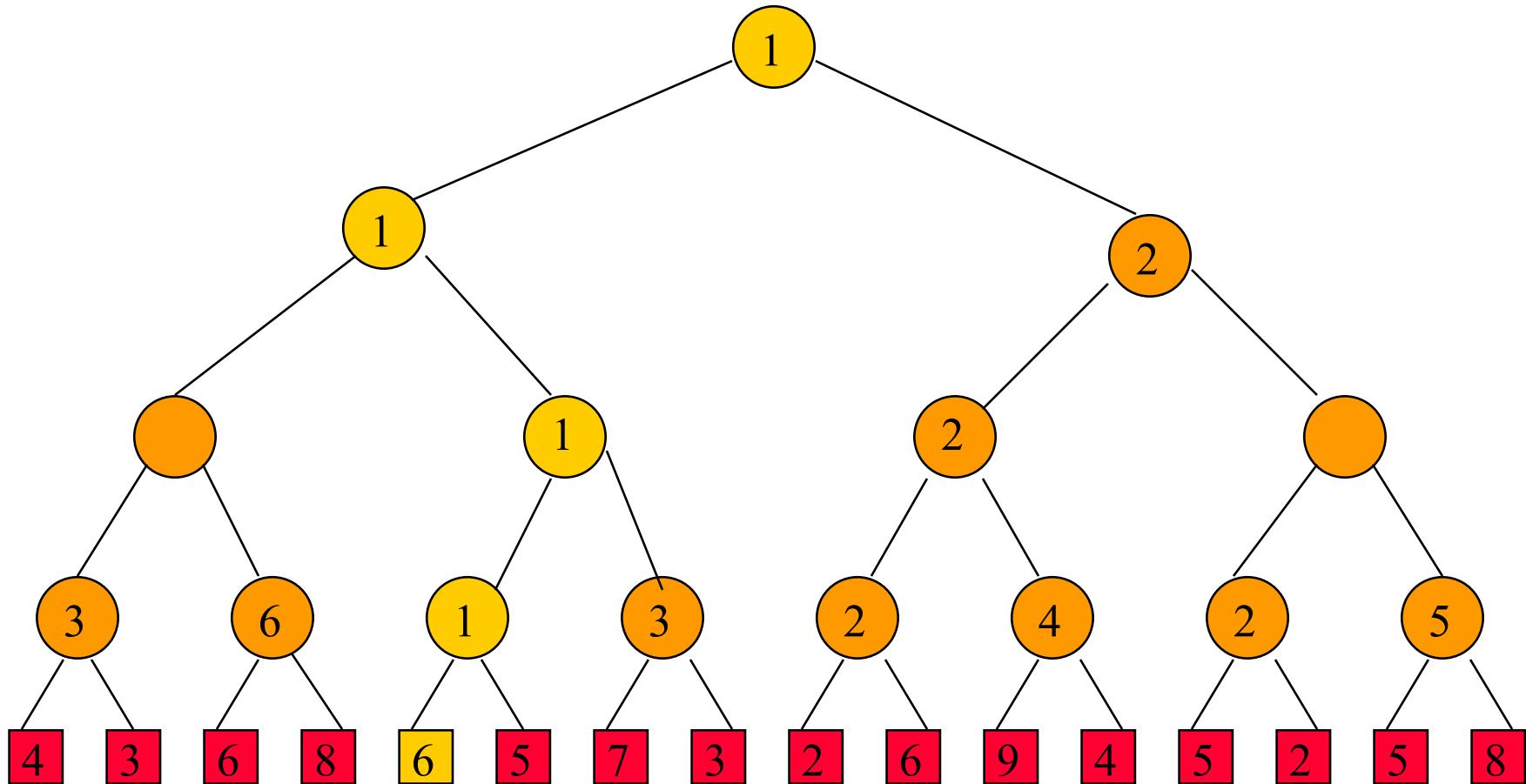
Replace winner with 6.

Replace Winner And Replay



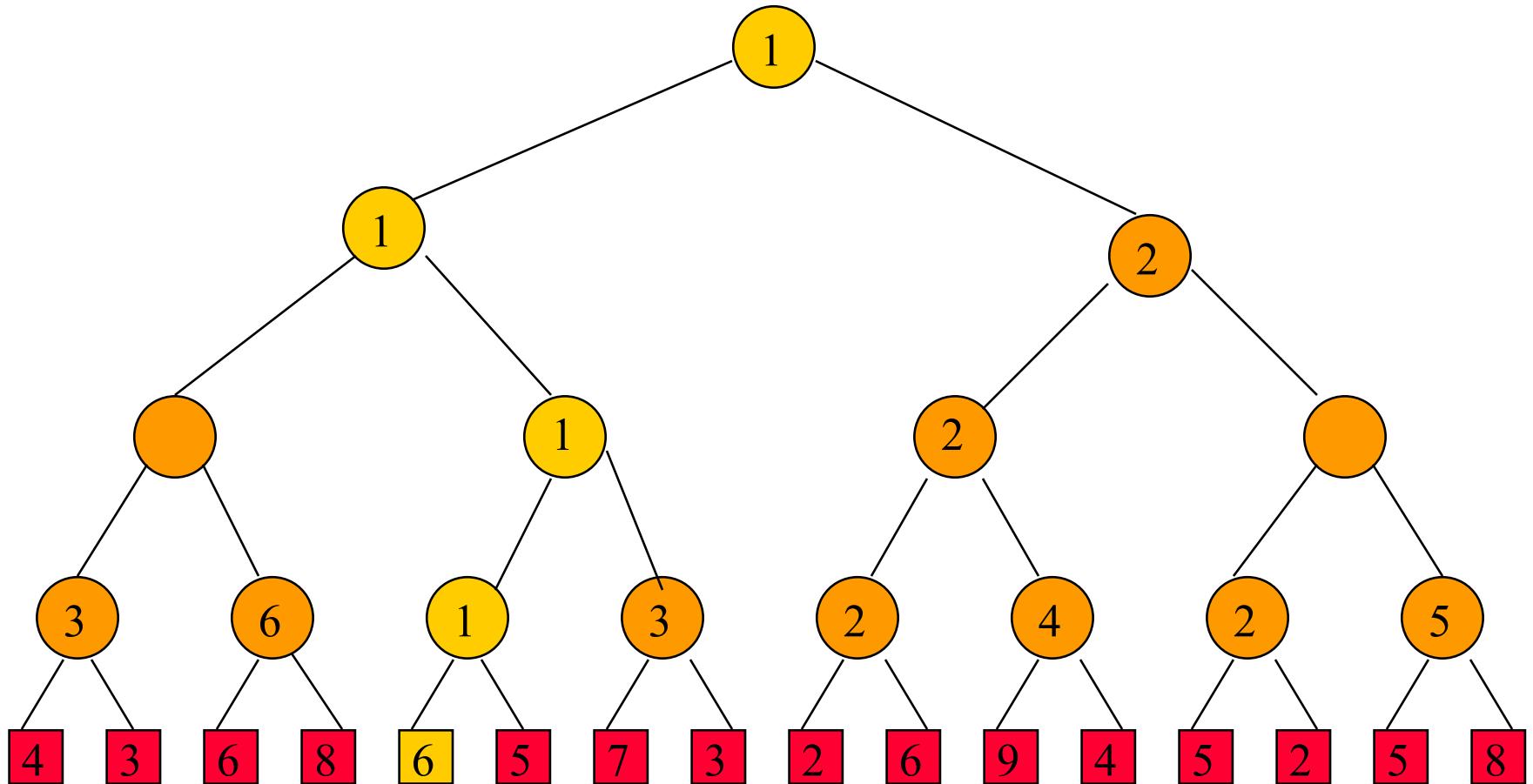
Replay matches on path to root.

Replace Winner And Replay



Replay matches on path to root.

Replace Winner And Replay

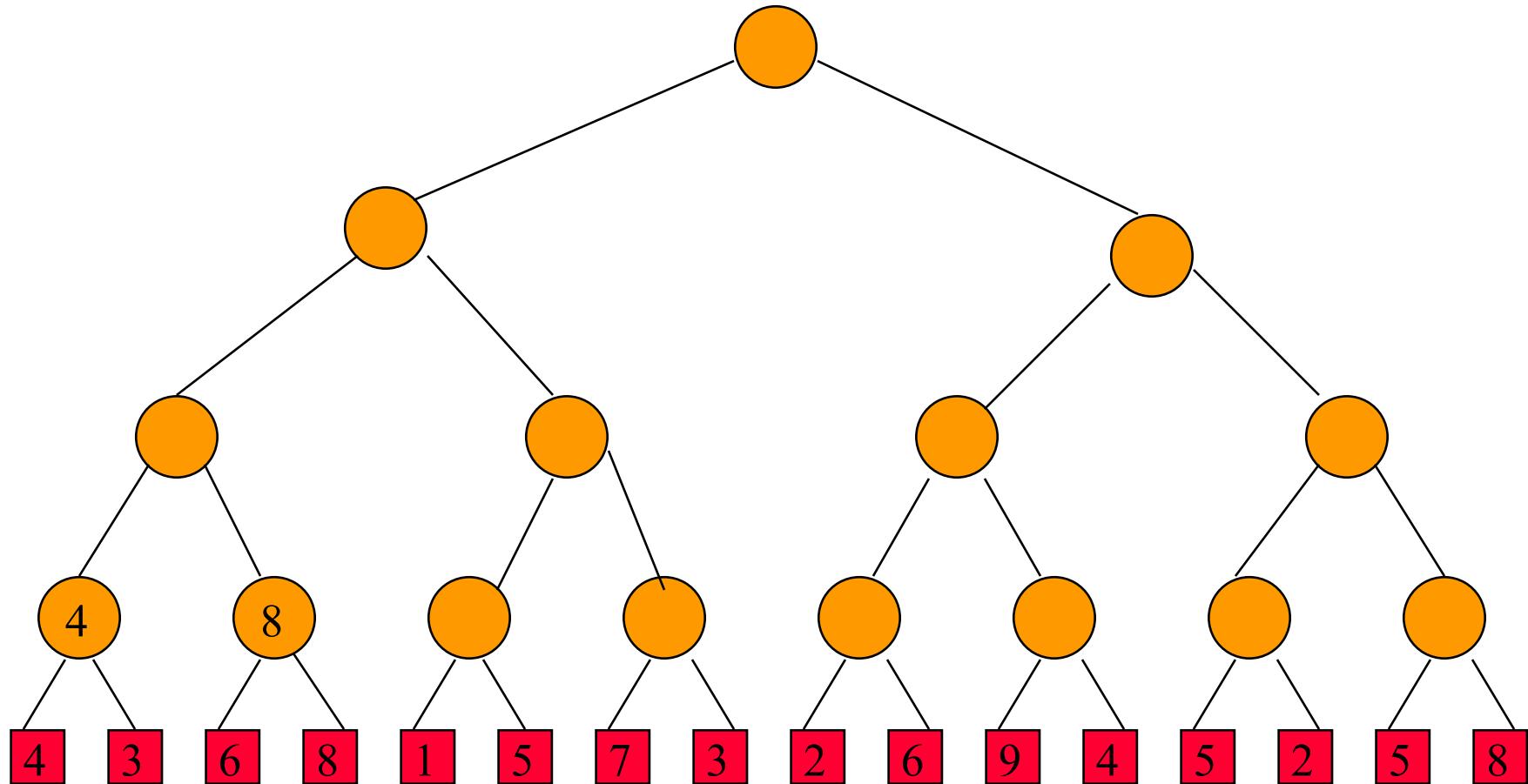


Opponent is player who lost last match played at this node.

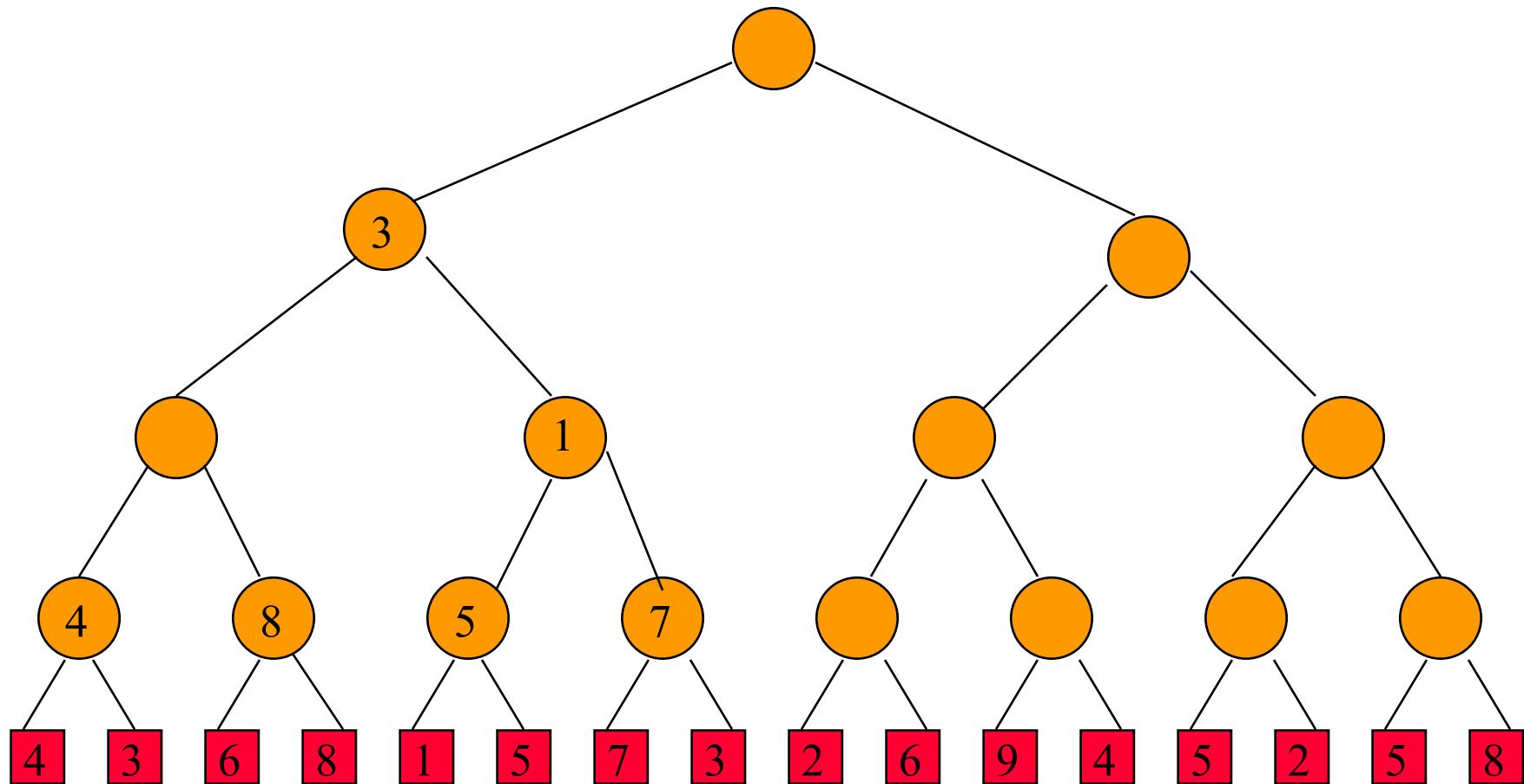
Loser Tree

Each match node stores the match loser rather than the match winner.

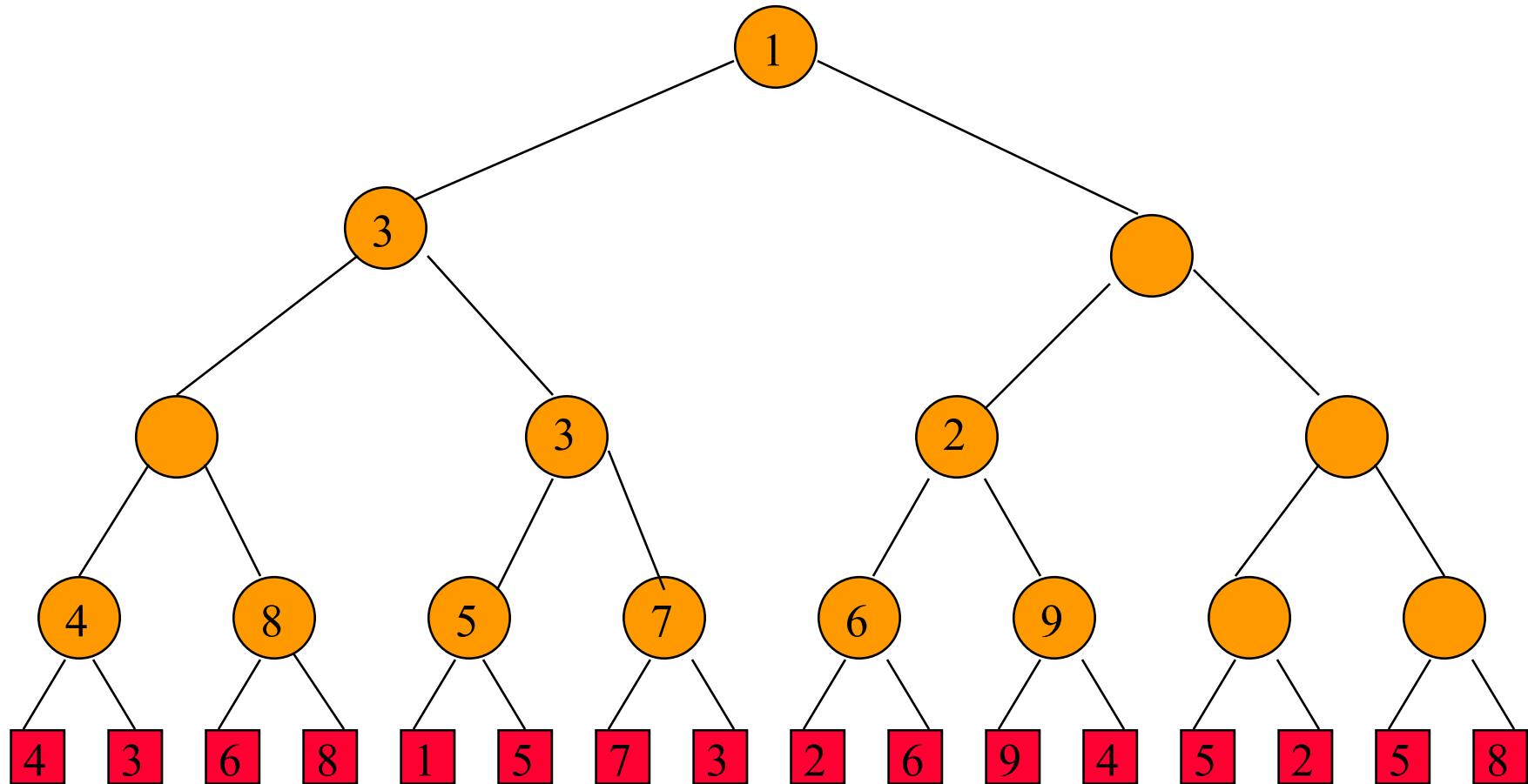
Min Loser Tree For 16 Players



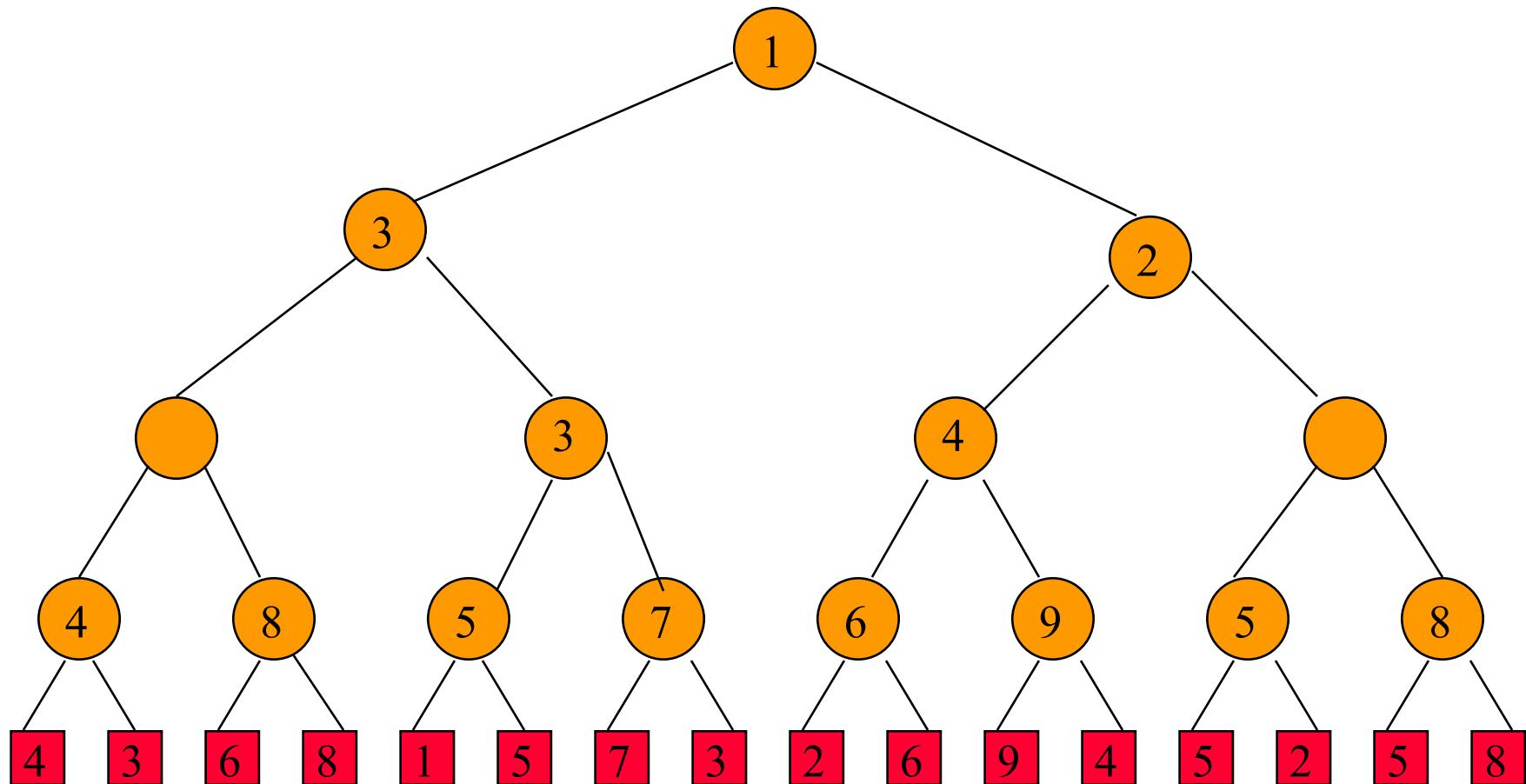
Min Loser Tree For 16 Players



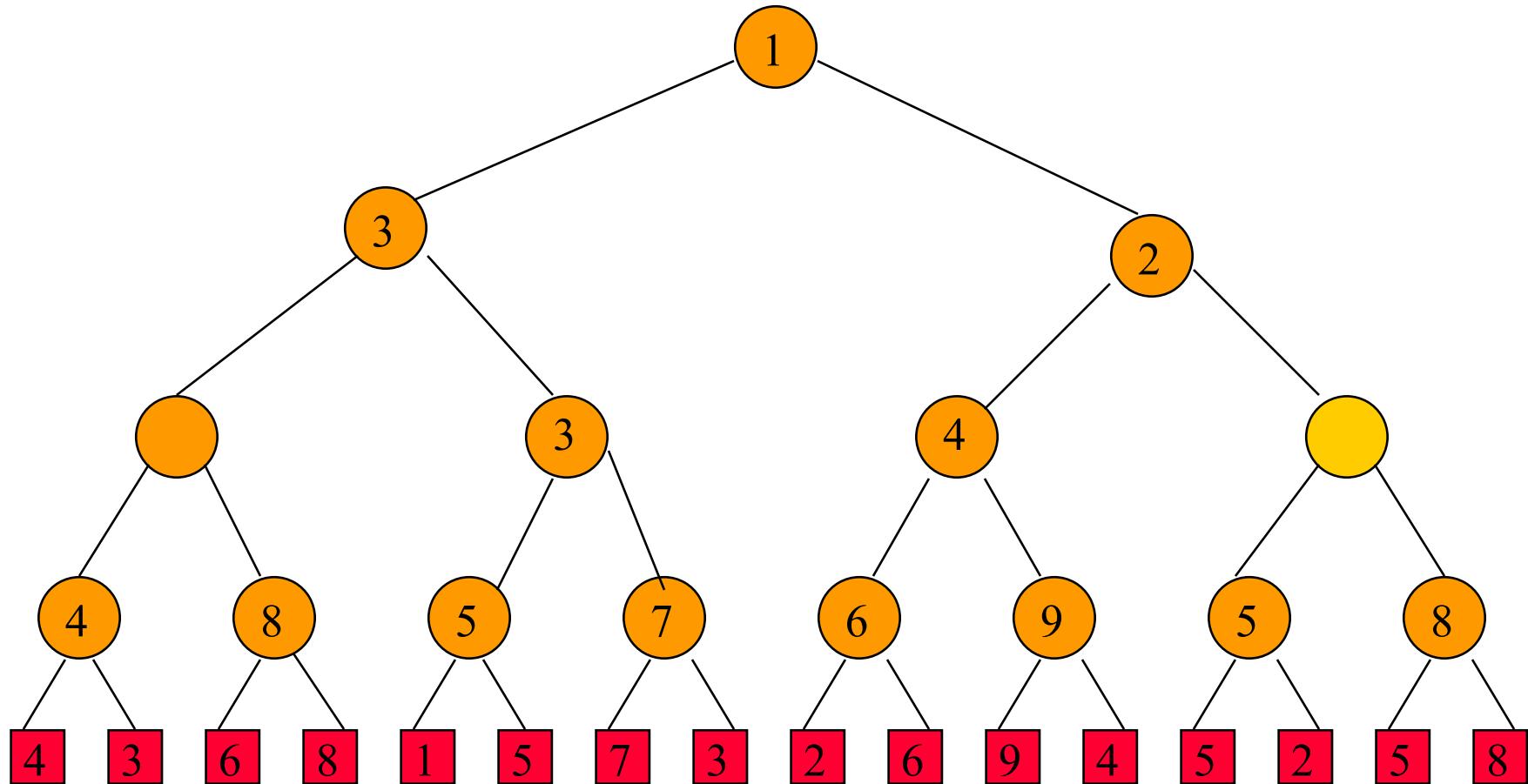
Min Loser Tree For 16 Players



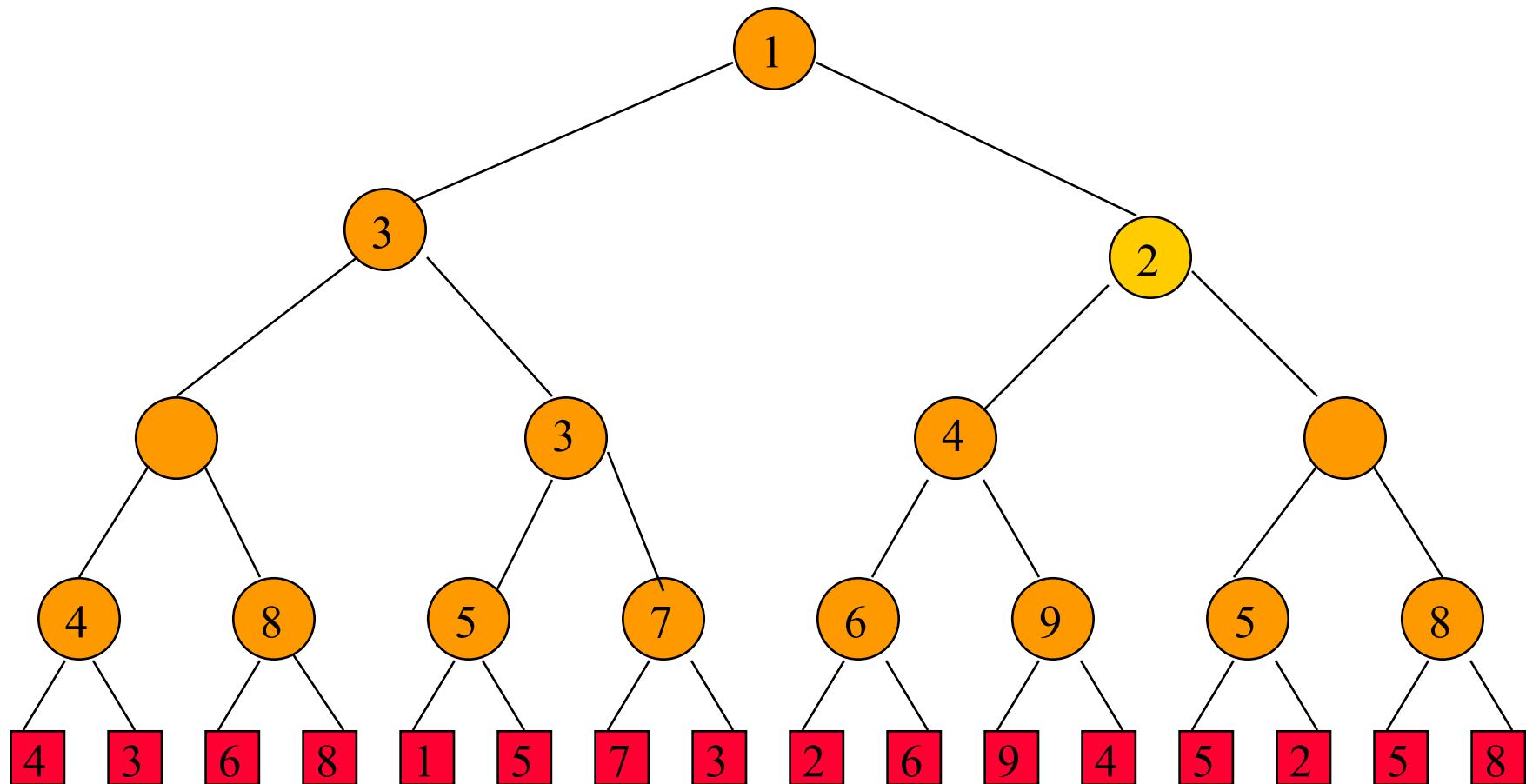
Min Loser Tree For 16 Players



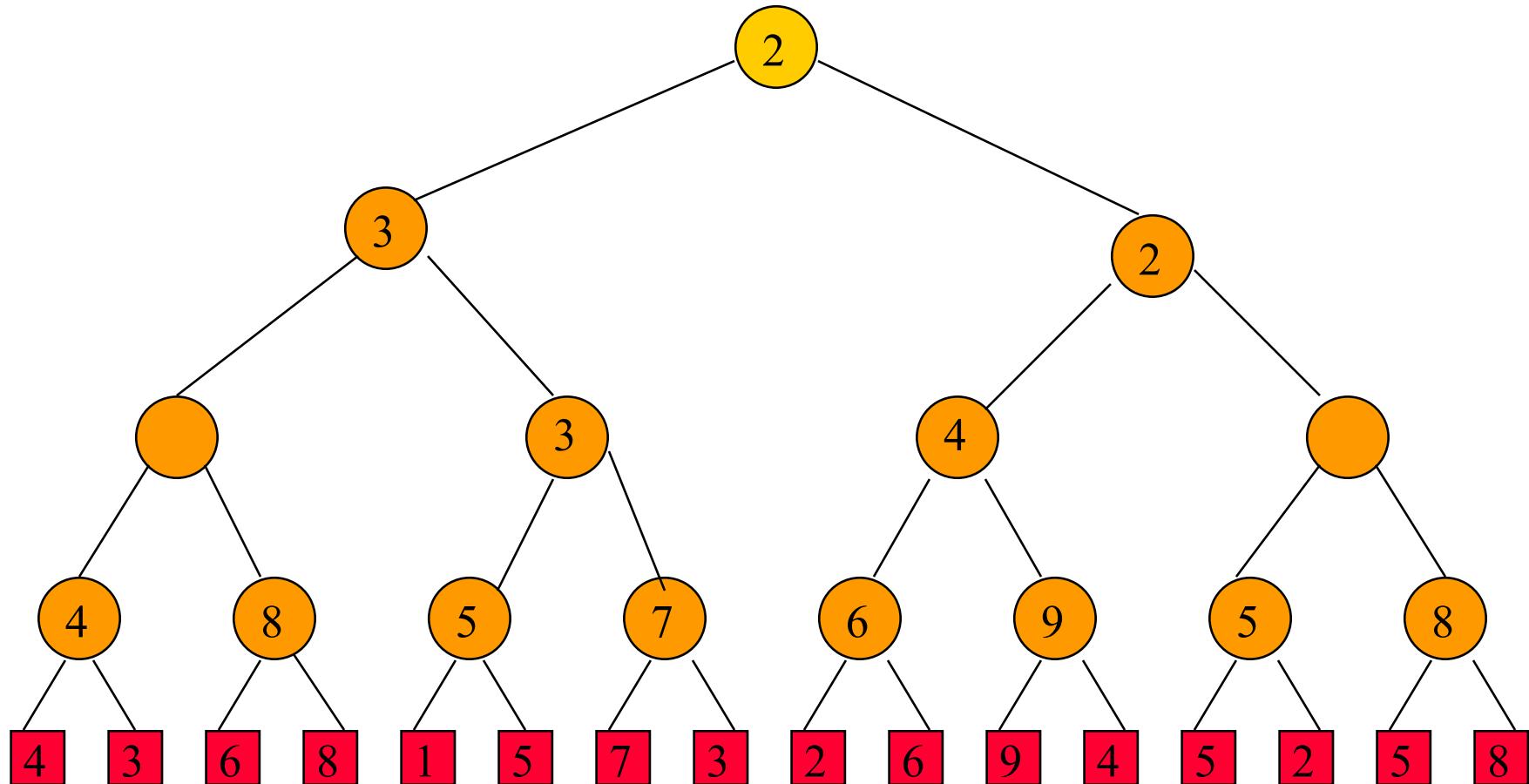
Min Loser Tree For 16 Players

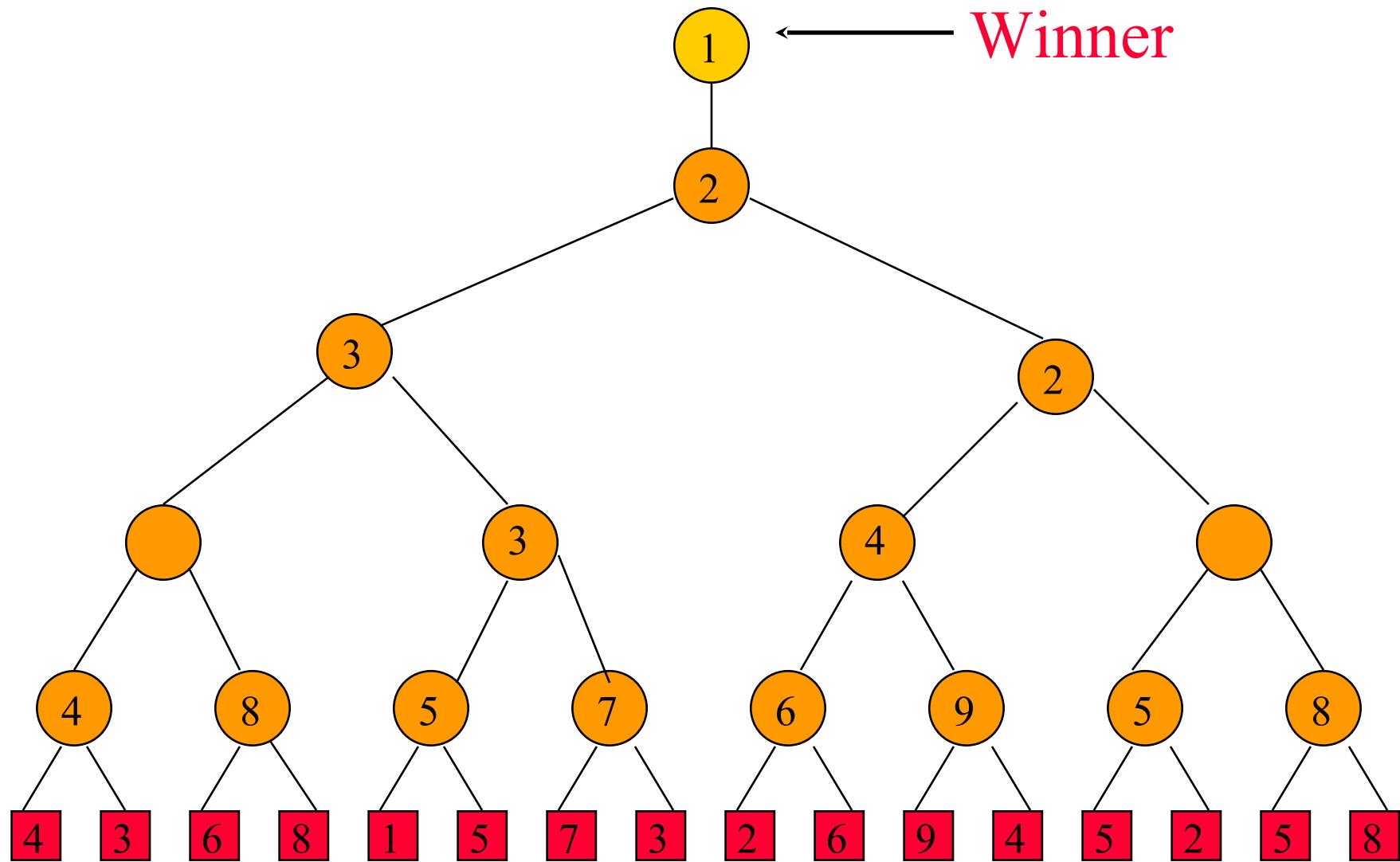


Min Loser Tree For 16 Players



Min Loser Tree For 16 Players



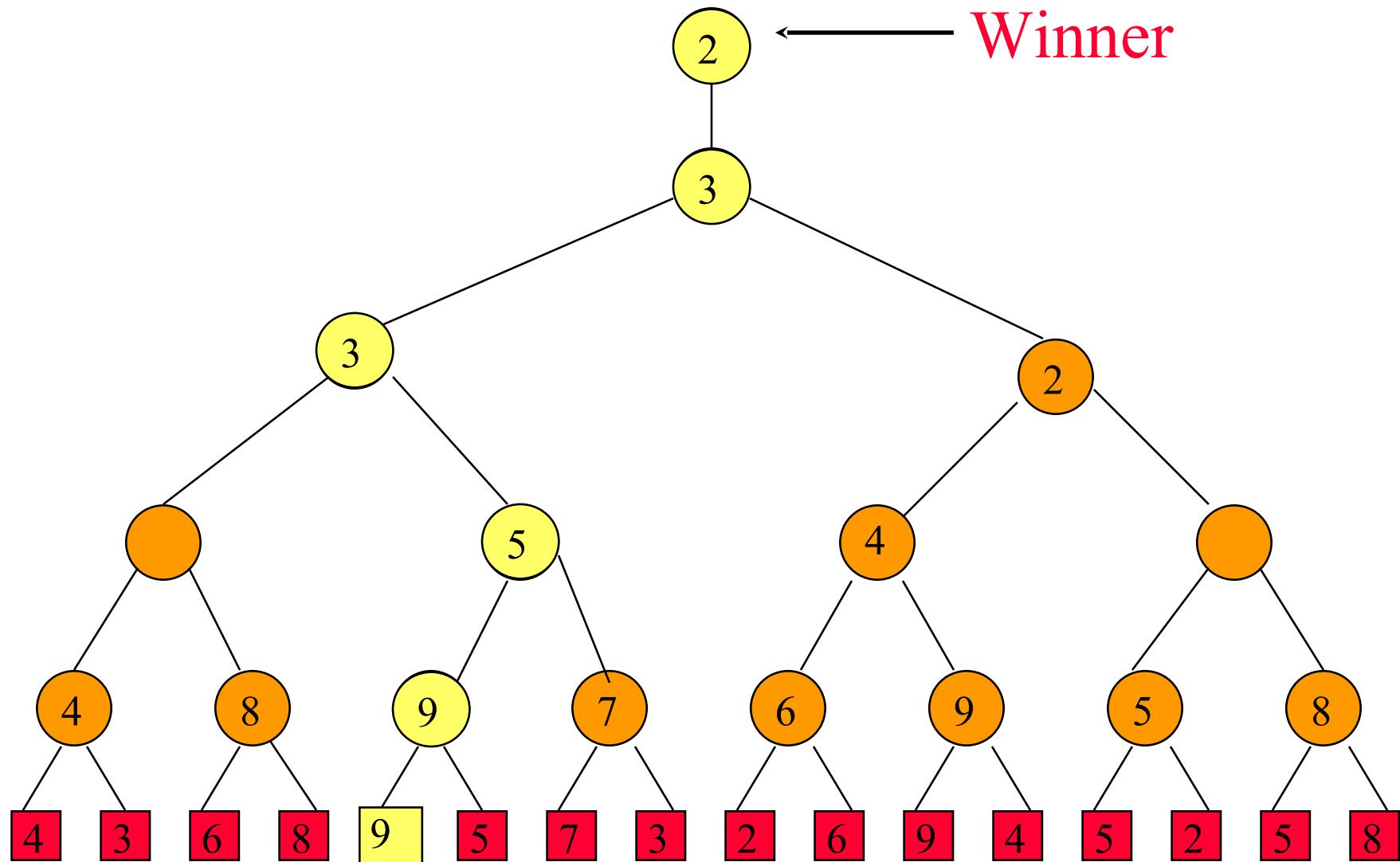


Complexity Of Loser Tree

Initialize



- One match at each match node.
- One store of a left child winner.
- Total time is $O(n)$.
- More precisely $\Theta(n)$.



Replace winner with 9 and replay matches.

Complexity Of Replay



- One match at each level that has a match node.
- $O(\log n)$
- More precisely $\Theta(\log n)$.

More Tournament Tree Applications

- k -way merging of runs during an external merge sort
- Truck loading

Truck Loading



- n packages to be loaded into trucks
- each package has a weight
- each truck has a capacity of c tons
- minimize number of trucks

Truck Loading

$n = 5$ packages

weights [2, 5, 6, 3, 4]

truck capacity $c = 10$

Load packages from left to right. If a package doesn't fit into current truck, start loading a new truck.

Truck Loading

$n = 5$ packages

weights [2, 5, 6, 3, 4]

truck capacity $c = 10$

truck1 = [2, 5]

truck2 = [6, 3]

truck3 = [4]

uses 3 trucks when 2 trucks suffice

Truck Loading

$n = 5$ packages

weights [2, 5, 6, 3, 4]

truck capacity $c = 10$

truck1 = [2, 5, 3]

truck2 = [6, 4]

Bin Packing

- n items to be packed into bins
- each item has a size
- each bin has a capacity of c
- minimize number of bins

Bin Packing

Truck loading is same as bin packing.

Truck is a bin that is to be packed (loaded).

Package is an item/element.

Bin packing to minimize number of bins is NP-hard.

Several fast heuristics have been proposed.

Bin Packing Heuristics

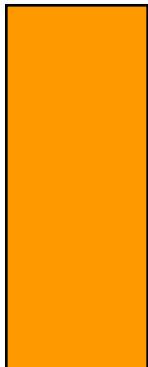
- First Fit.
 - Bins are arranged in left to right order.
 - Items are packed one at a time in given order.
 - Current item is packed into leftmost bin into which it fits.
 - If there is no bin into which current item fits, start a new bin.

First Fit

$n = 4$

weights = [4, 7, 3, 6]

capacity = 10



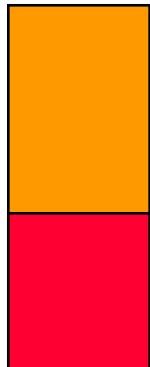
Pack red item into first bin.

First Fit

$n = 4$

weights = [4, 7, 3, 6]

capacity = 10



Pack blue item next.

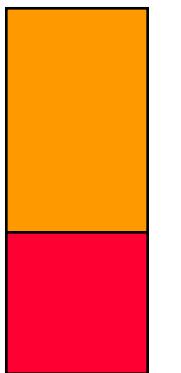
Doesn't fit, so start a new bin.

First Fit

$n = 4$

weights = [4, 7, 3, 6]

capacity = 10

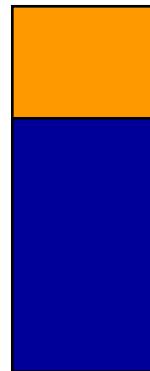
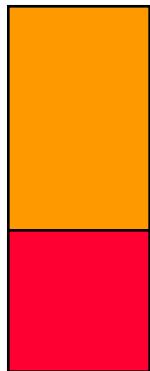


First Fit

$n = 4$

weights = [4, 7, 3, 6]

capacity = 10



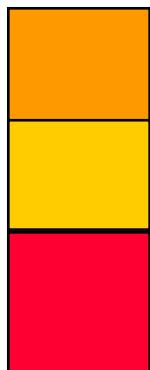
Pack yellow item into first bin.

First Fit

$n = 4$

weights = [4, 7, 3, 6]

capacity = 10



Pack green item.

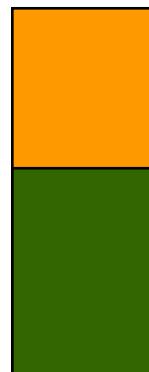
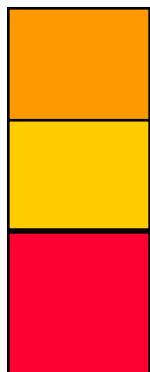
Need a new bin.

First Fit

$n = 4$

weights = [4, 7, 3, 6]

capacity = 10



Not optimal.

2 bins suffice.

Bin Packing Heuristics

- First Fit Decreasing.
 - Items are sorted into decreasing order.
 - Then first fit is applied.

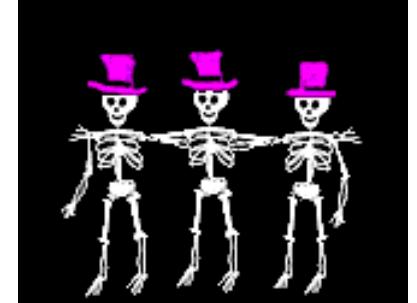
Bin Packing Heuristics

- Best Fit.
 - Items are packed one at a time in given order.
 - To determine the bin for an item, first determine set **S** of bins into which the item fits.
 - If **S** is empty, then start a new bin and put item into this new bin.
 - Otherwise, pack into bin of **S** that has least available capacity.

Bin Packing Heuristics

- Best Fit Decreasing.
 - Items are sorted into decreasing order.
 - Then best fit is applied.

Performance



- For first fit and best fit:

$$\text{Heuristic Bins} \leq (17/10)(\text{Minimum Bins}) + 2$$

- For first fit decreasing and best fit decreasing:

$$\text{Heuristic Bins} \leq (11/9)(\text{Minimum Bins}) + 4$$

Complexity Of First Fit



Use a max tournament tree in which the players are n bins and the value of a player is the available capacity in the bin.

$O(n \log n)$, where n is the number of items.



Binary Search Trees



- Dictionary Operations:
 - `get(key)`
 - `put(key, value)`
 - `remove(key)`
- Additional operations:
 - `ascend()`
 - `get(index)` (indexed binary search tree)
 - `remove(index)` (indexed binary search tree)

Complexity Of Dictionary Operations

get(), put() and remove()

Data Structure	Worst Case	Expected
Hash Table	$O(n)$	$O(1)$
Binary Search Tree	$O(n)$	$O(\log n)$
Balanced Binary Search Tree	$O(\log n)$	$O(\log n)$

n is number of elements in dictionary

Complexity Of Other Operations

ascend(), get(index), remove(index)

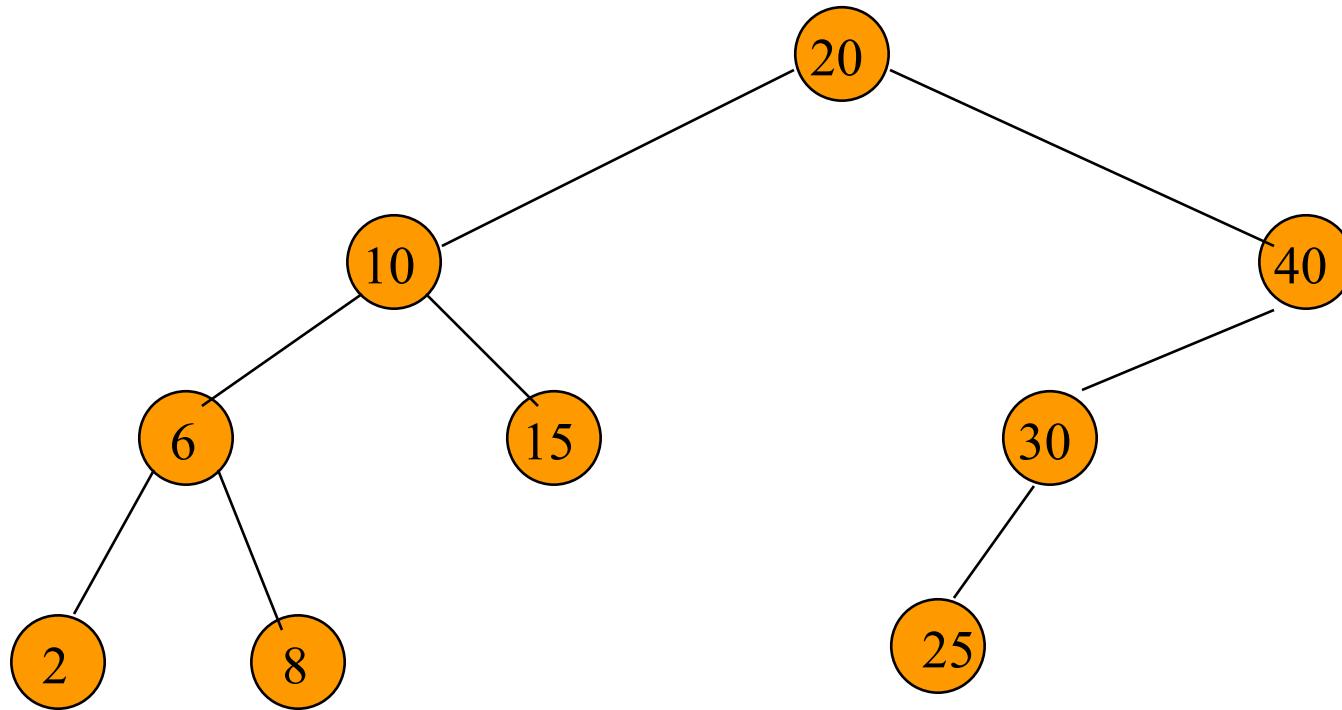
Data Structure	ascend	get and remove
Hash Table	$O(D + n \log n)$	$O(D + n \log n)$
Indexed BST	$O(n)$	$O(n)$
Indexed Balanced BST	$O(n)$	$O(\log n)$

D is number of buckets

Definition Of Binary Search Tree

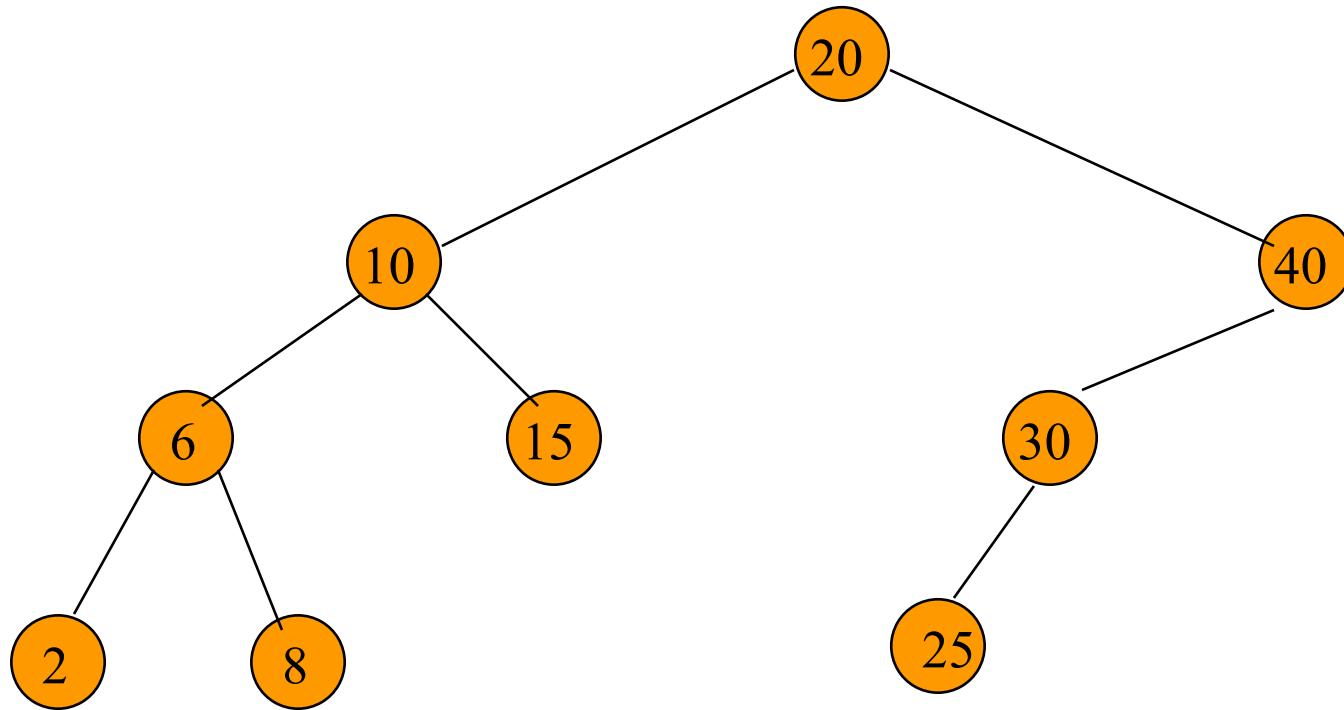
- A binary tree.
- Each node has a (key, value) pair.
- For every node x , all keys in the left subtree of x are smaller than that in x .
- For every node x , all keys in the right subtree of x are greater than that in x .

Example Binary Search Tree



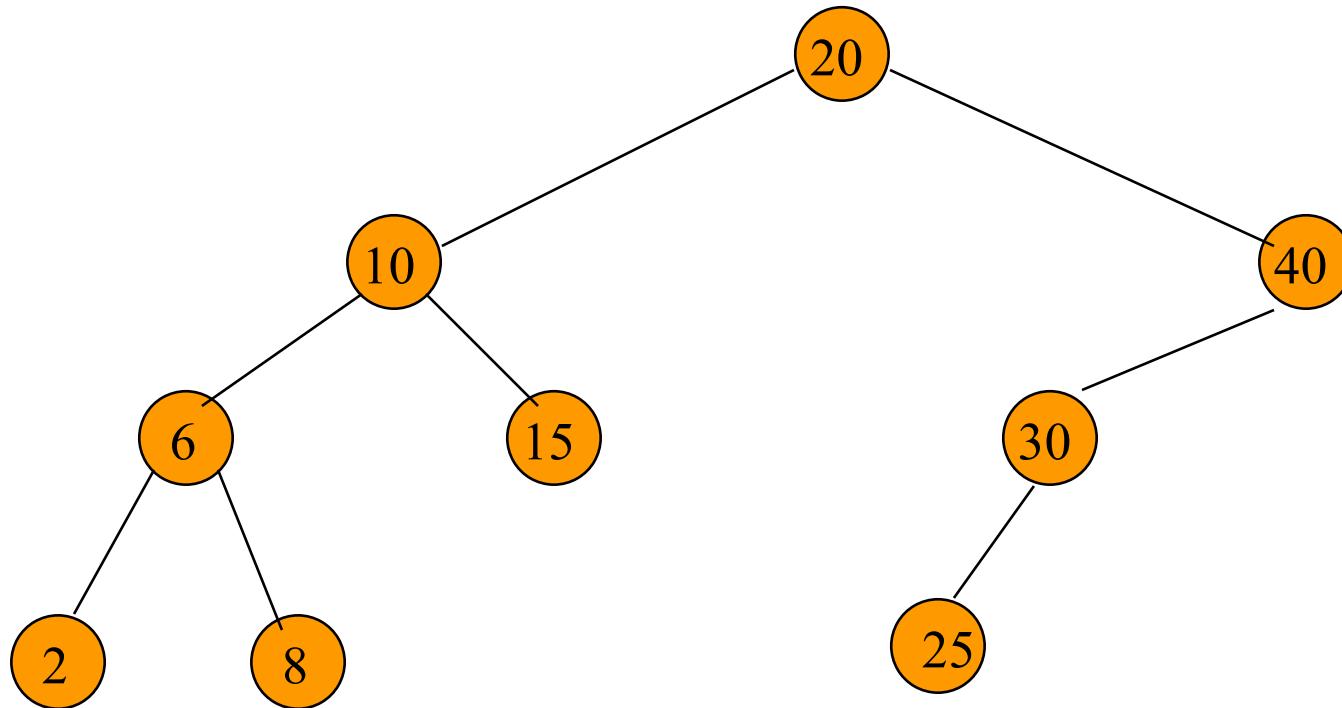
Only keys are shown.

The Operation `ascend()`



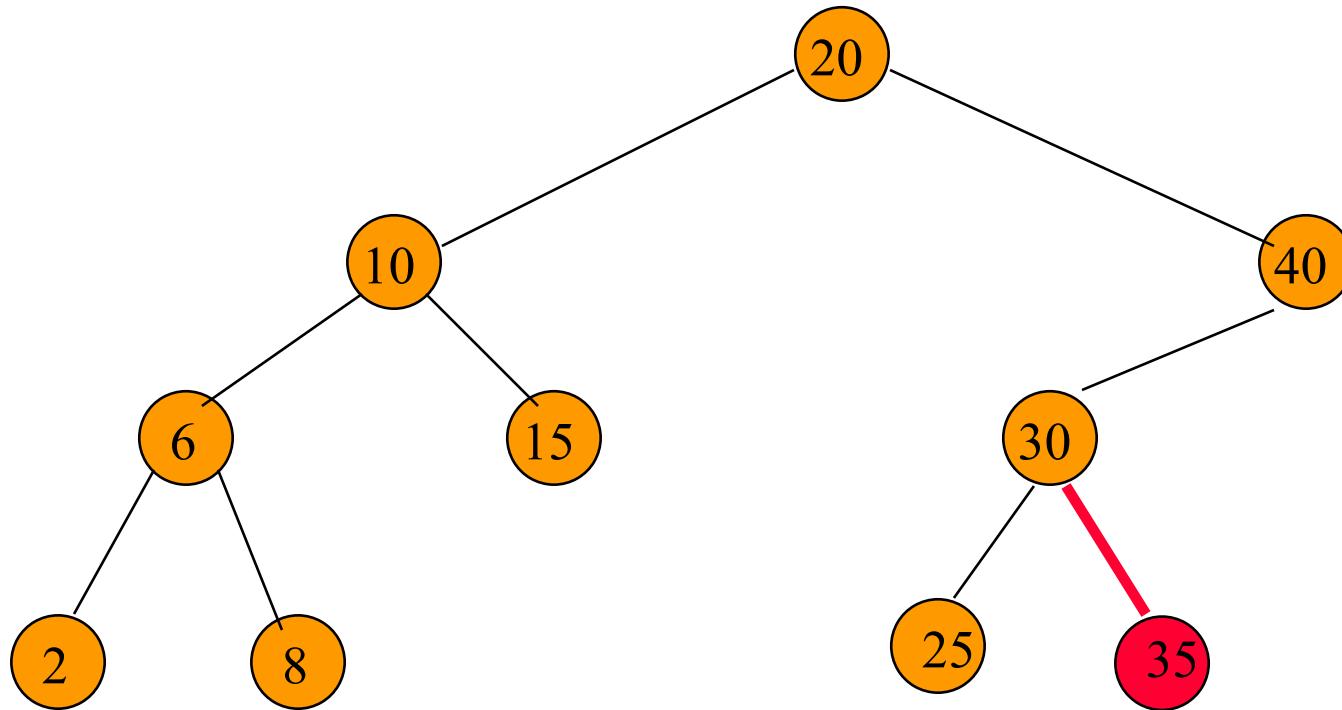
Do an inorder traversal. $O(n)$ time.

The Operation get()



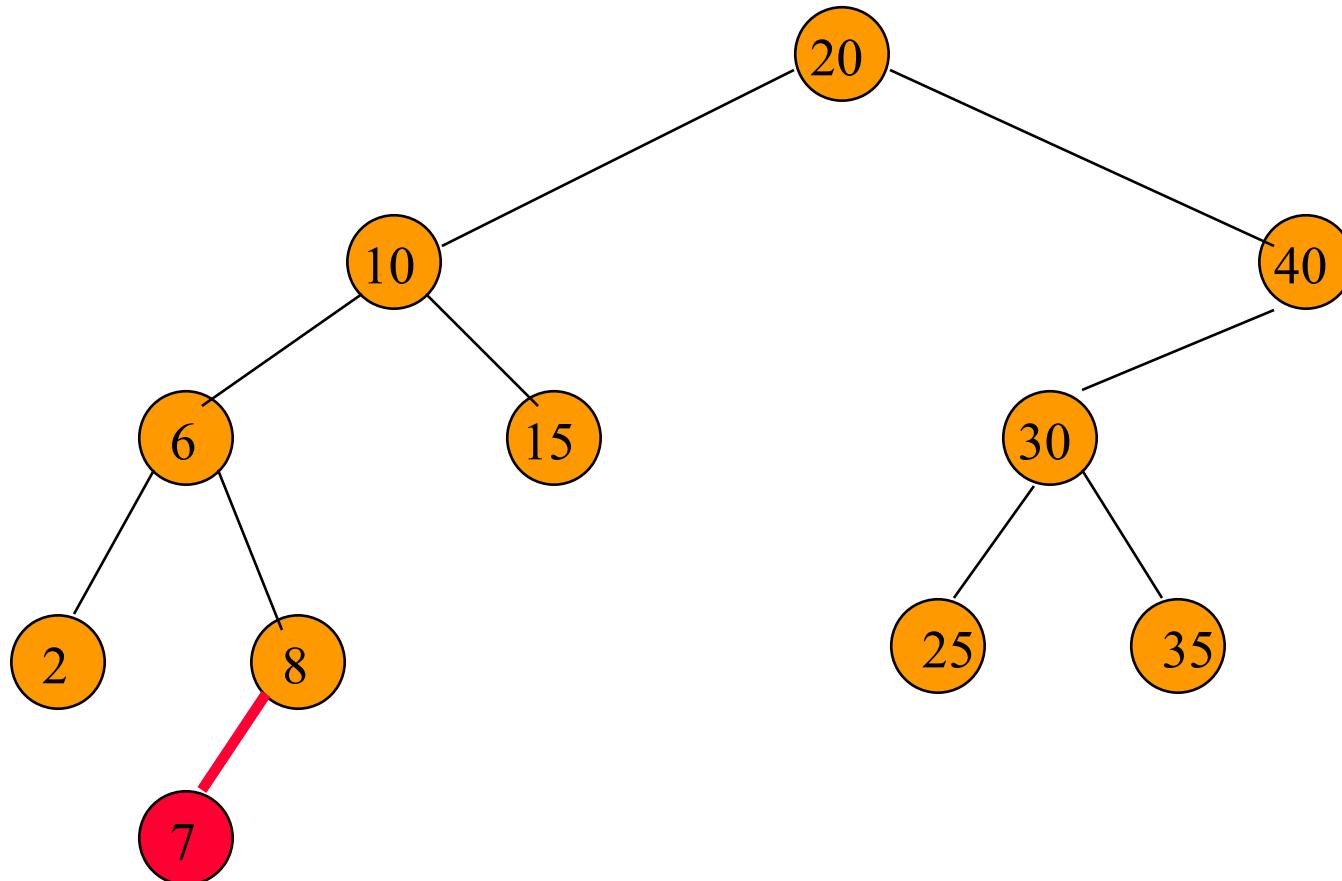
Complexity is $O(\text{height}) = O(n)$, where n is number of nodes/elements.

The Operation put()



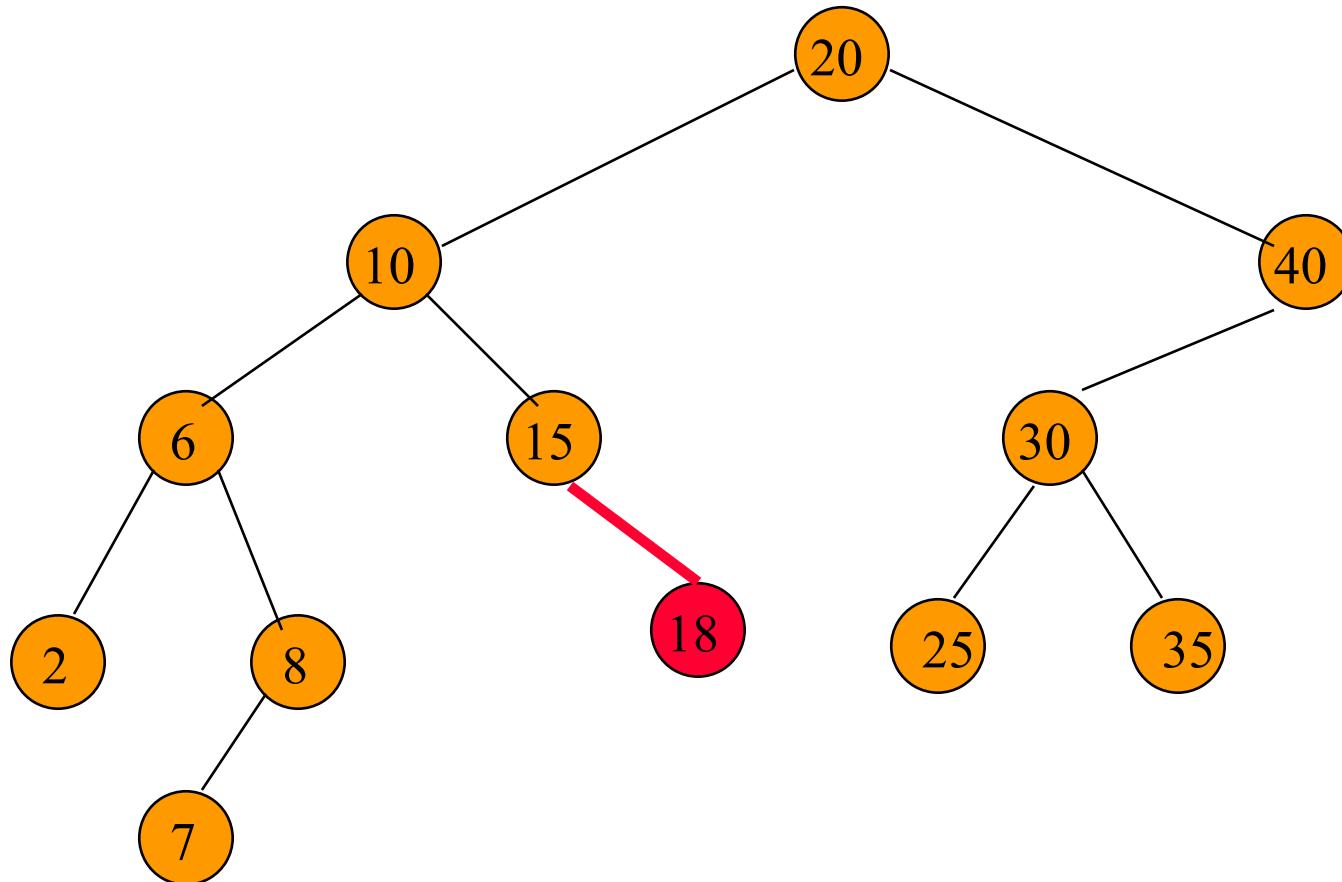
Put a pair whose key is 35.

The Operation put()



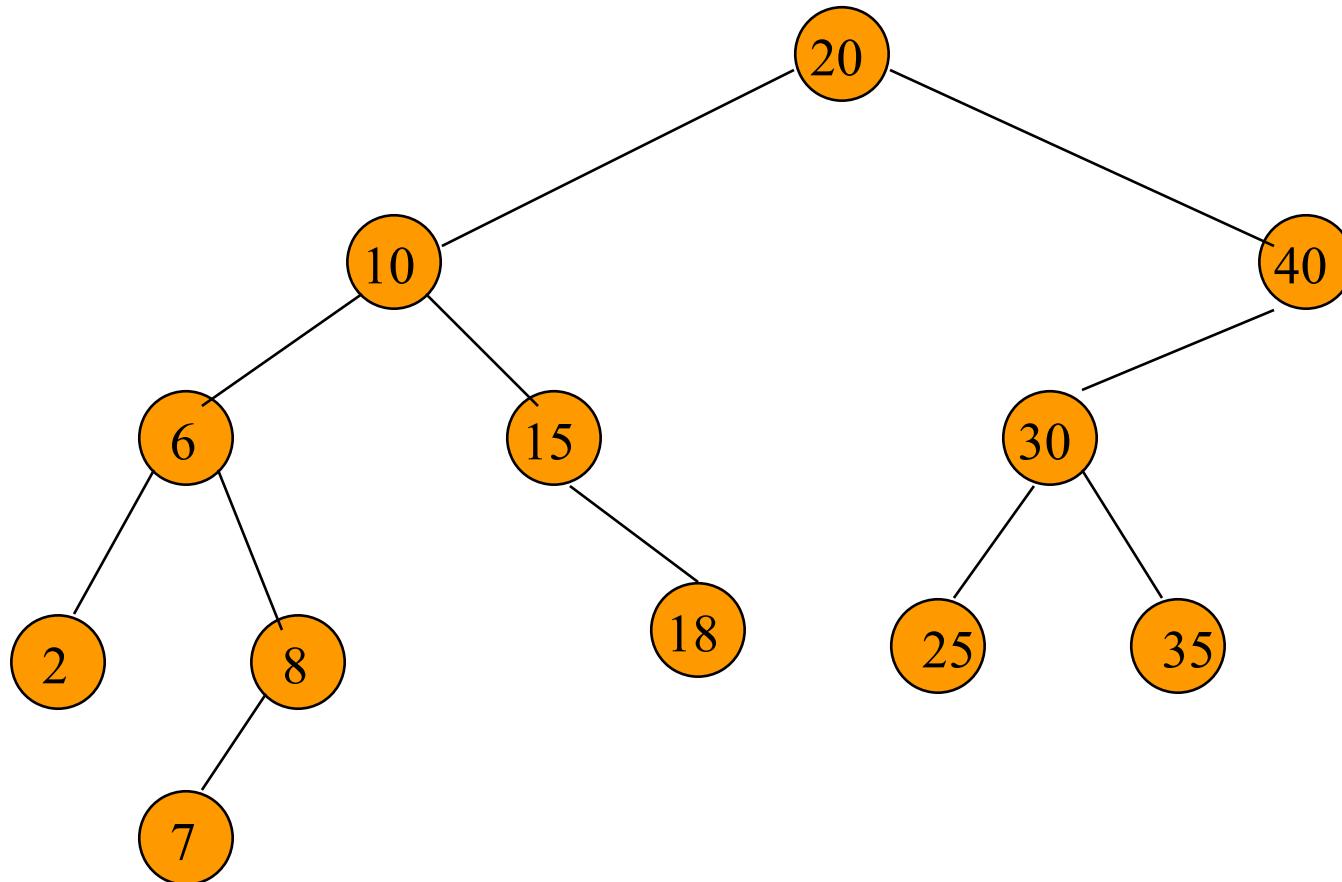
Put a pair whose key is 7.

The Operation put()



Put a pair whose key is 18.

The Operation put()



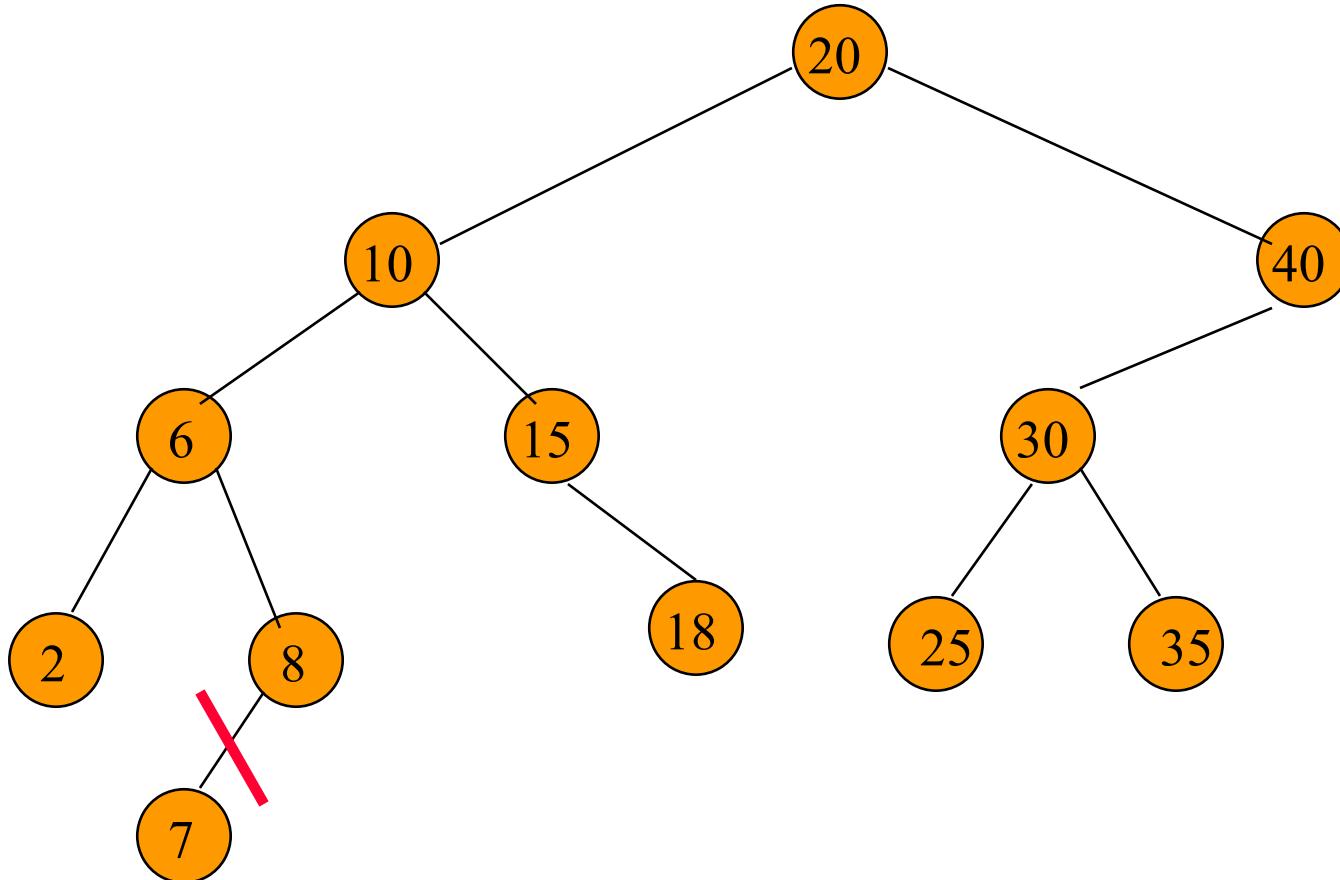
Complexity of `put()` is $O(\text{height})$.

The Operation remove()

Three cases:

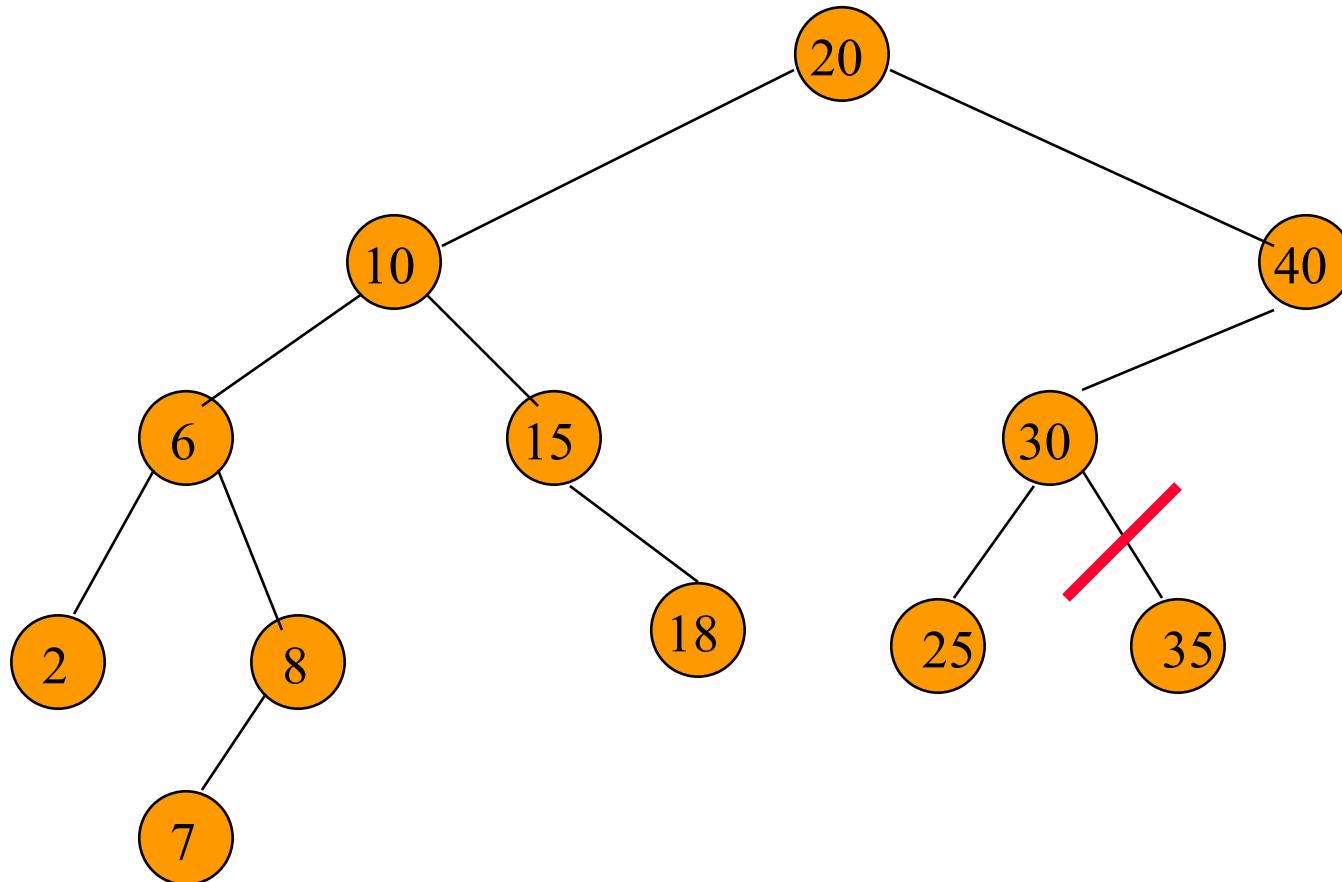
- Element is in a leaf.
- Element is in a degree 1 node.
- Element is in a degree 2 node.

Remove From A Leaf



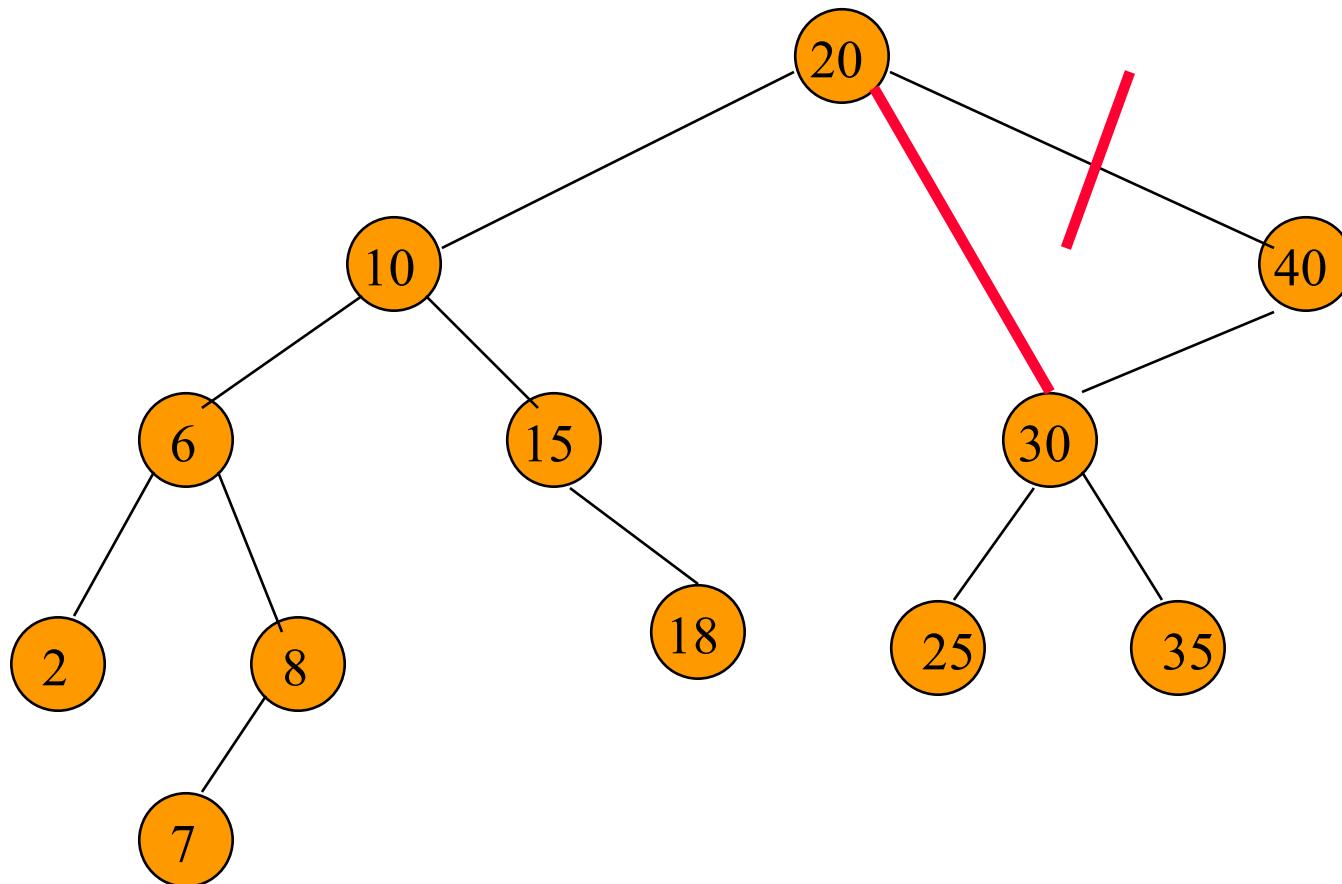
Remove a leaf element. key = 7

Remove From A Leaf (contd.)



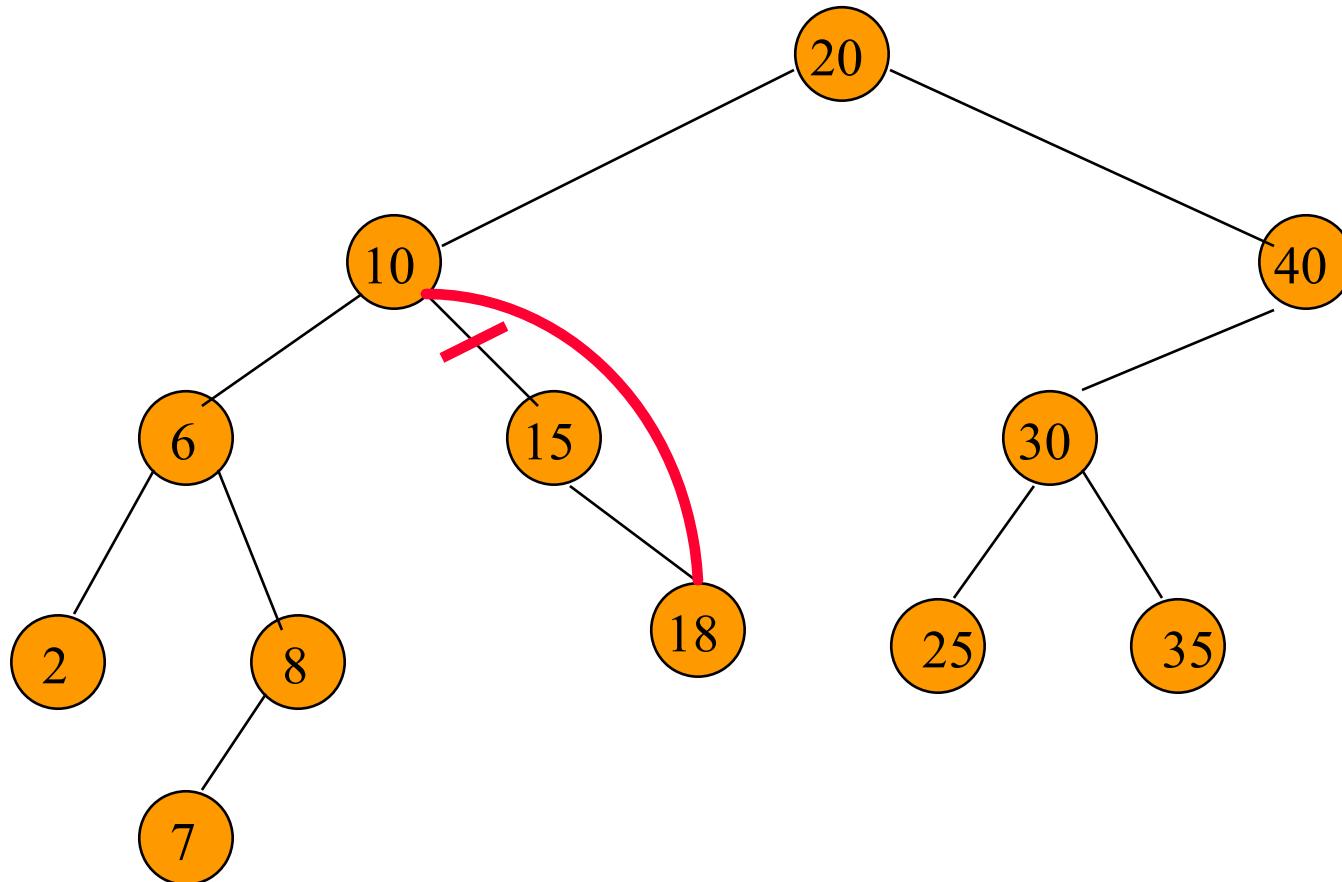
Remove a leaf element. key = 35

Remove From A Degree 1 Node



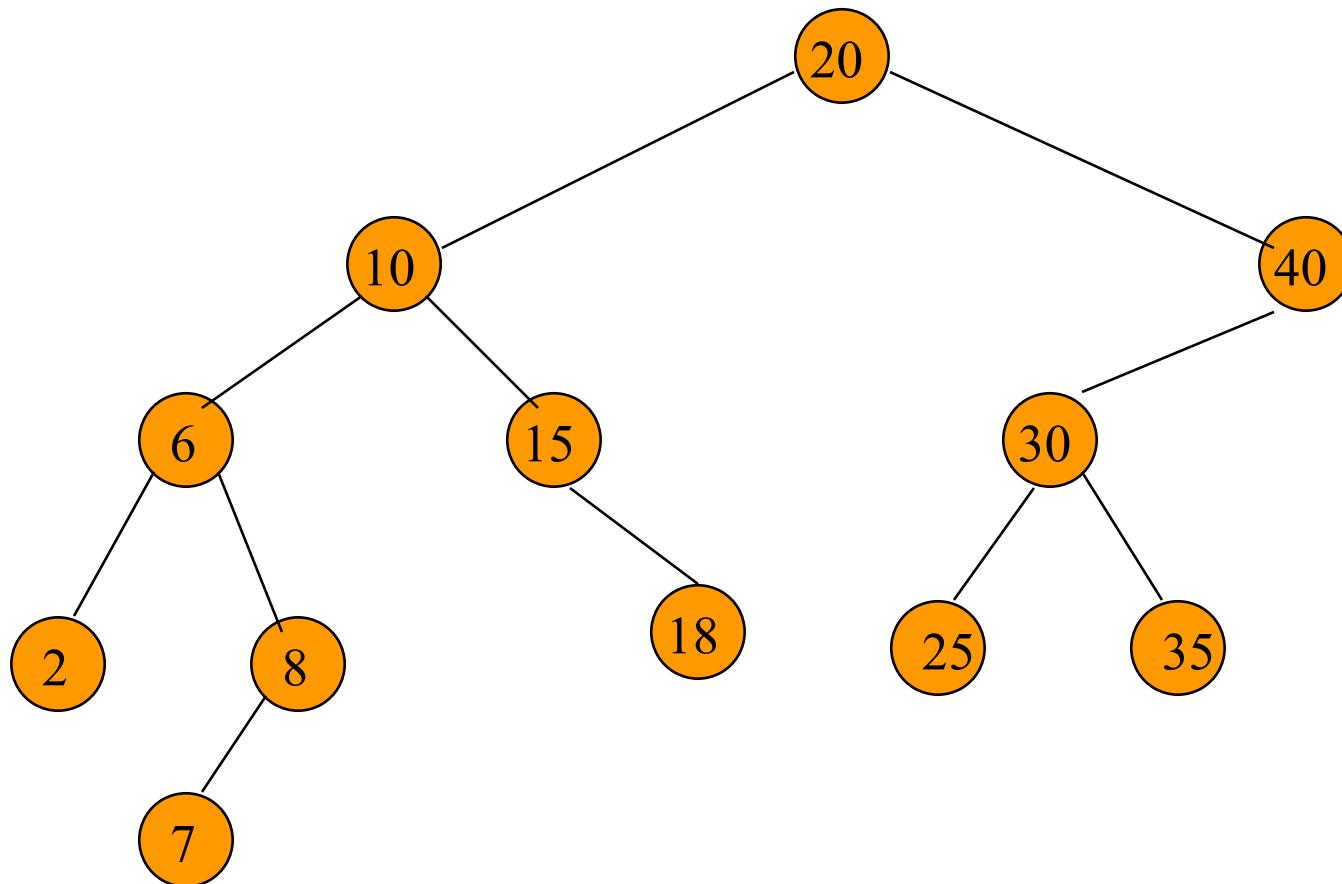
Remove from a degree 1 node. key = 40

Remove From A Degree 1 Node (contd.)



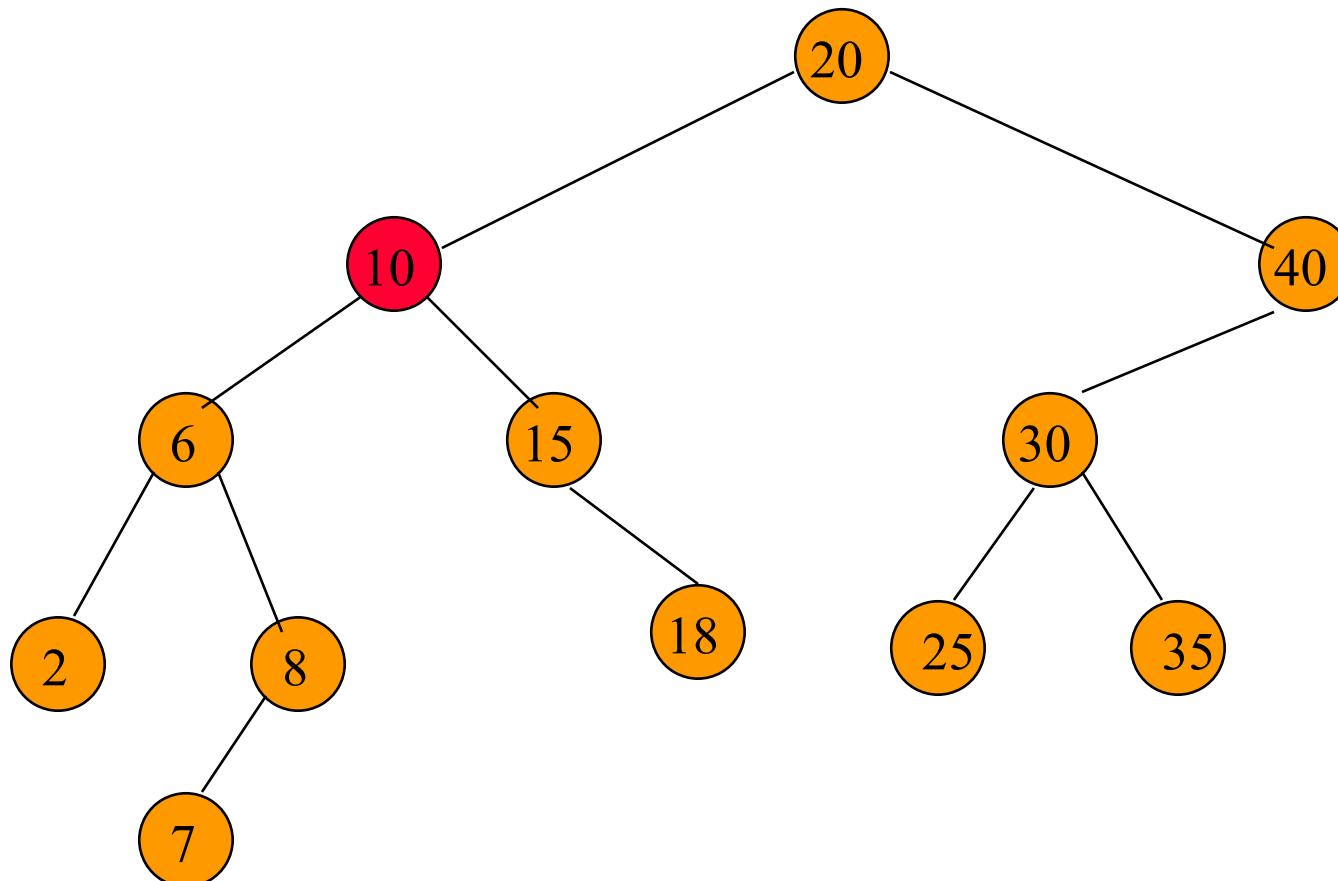
Remove from a degree 1 node. key = 15

Remove From A Degree 2 Node



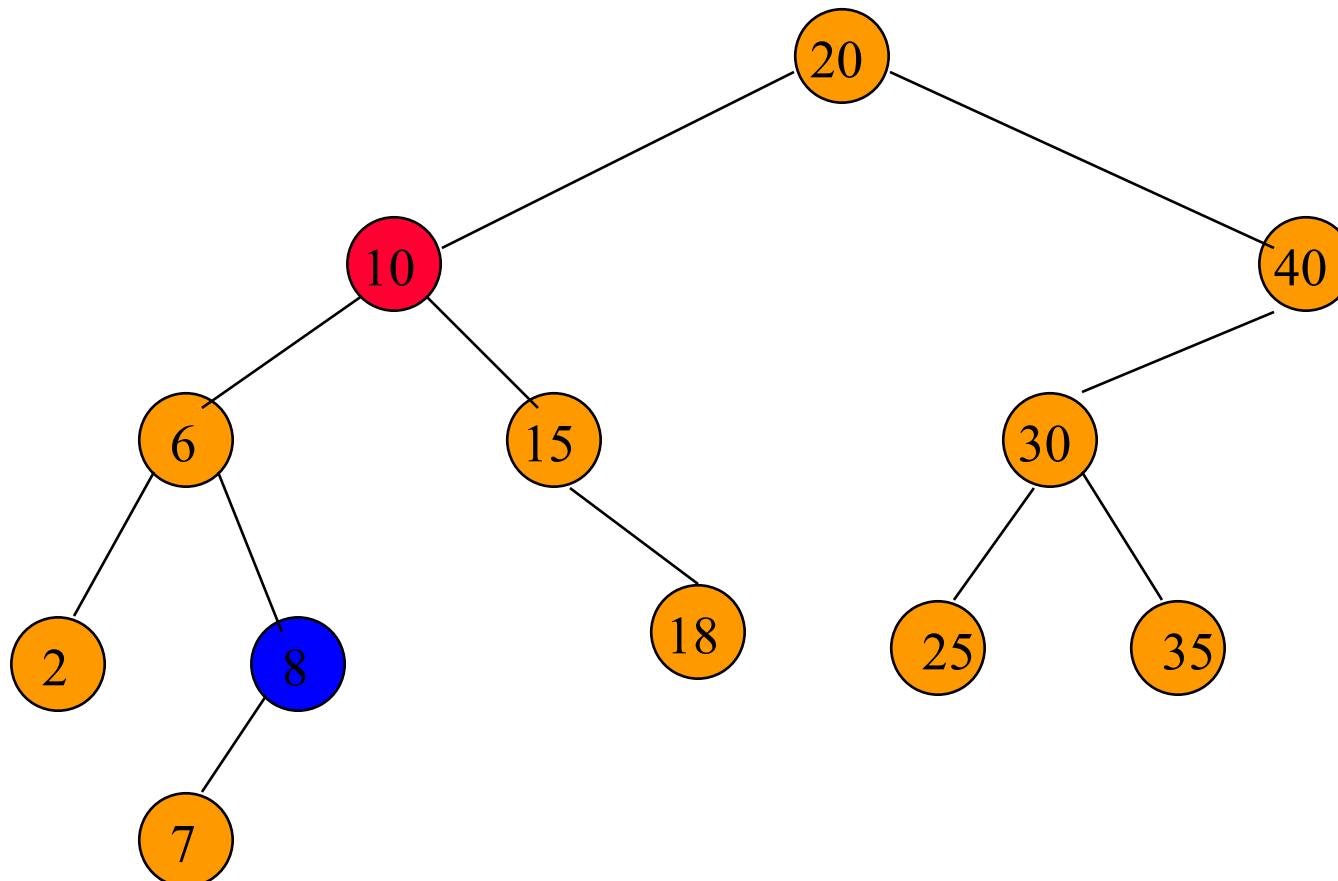
Remove from a degree 2 node. key = 10

Remove From A Degree 2 Node



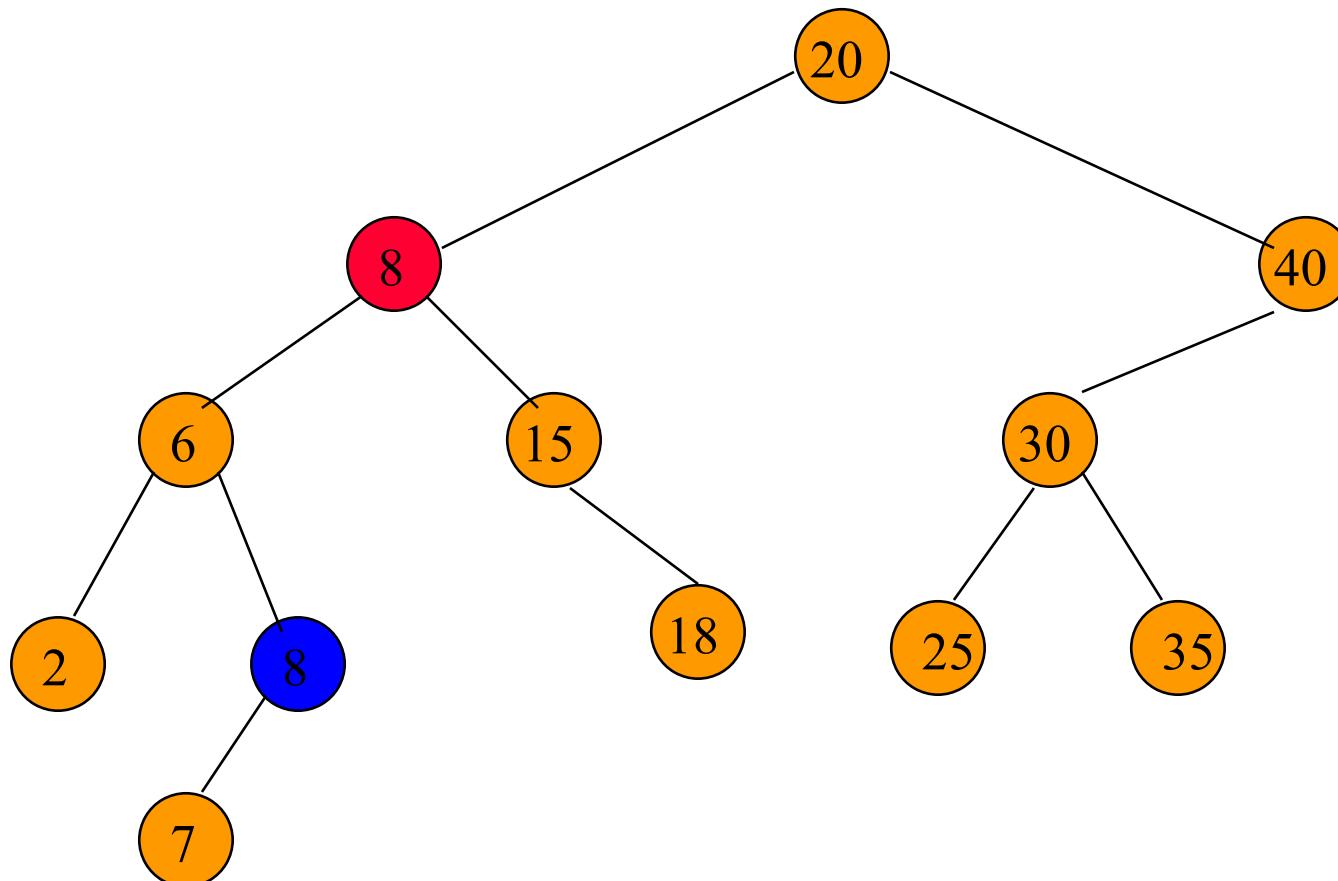
Replace with largest key in left subtree (or smallest in right subtree).

Remove From A Degree 2 Node



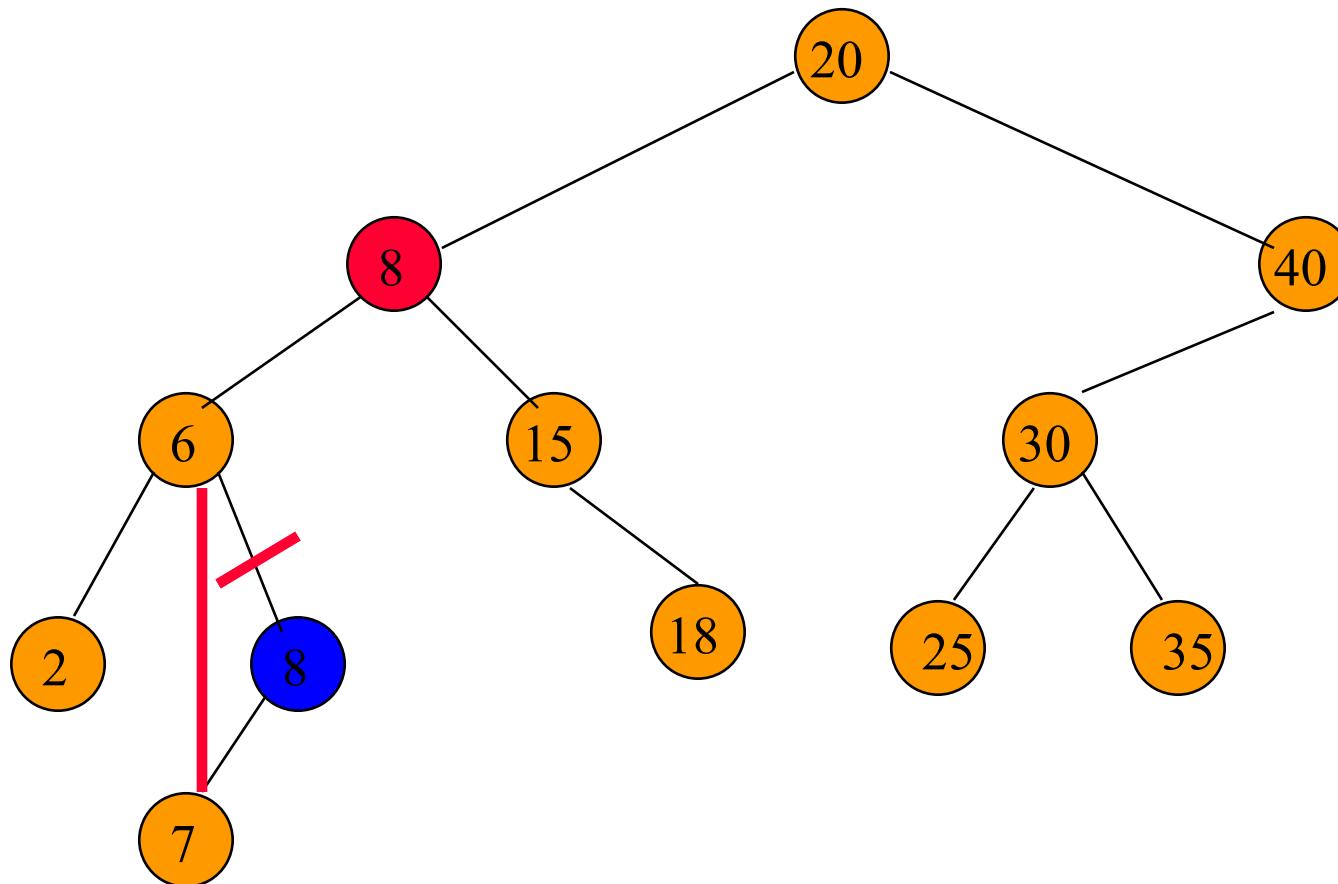
Replace with largest key in left subtree (or
smallest in right subtree).

Remove From A Degree 2 Node



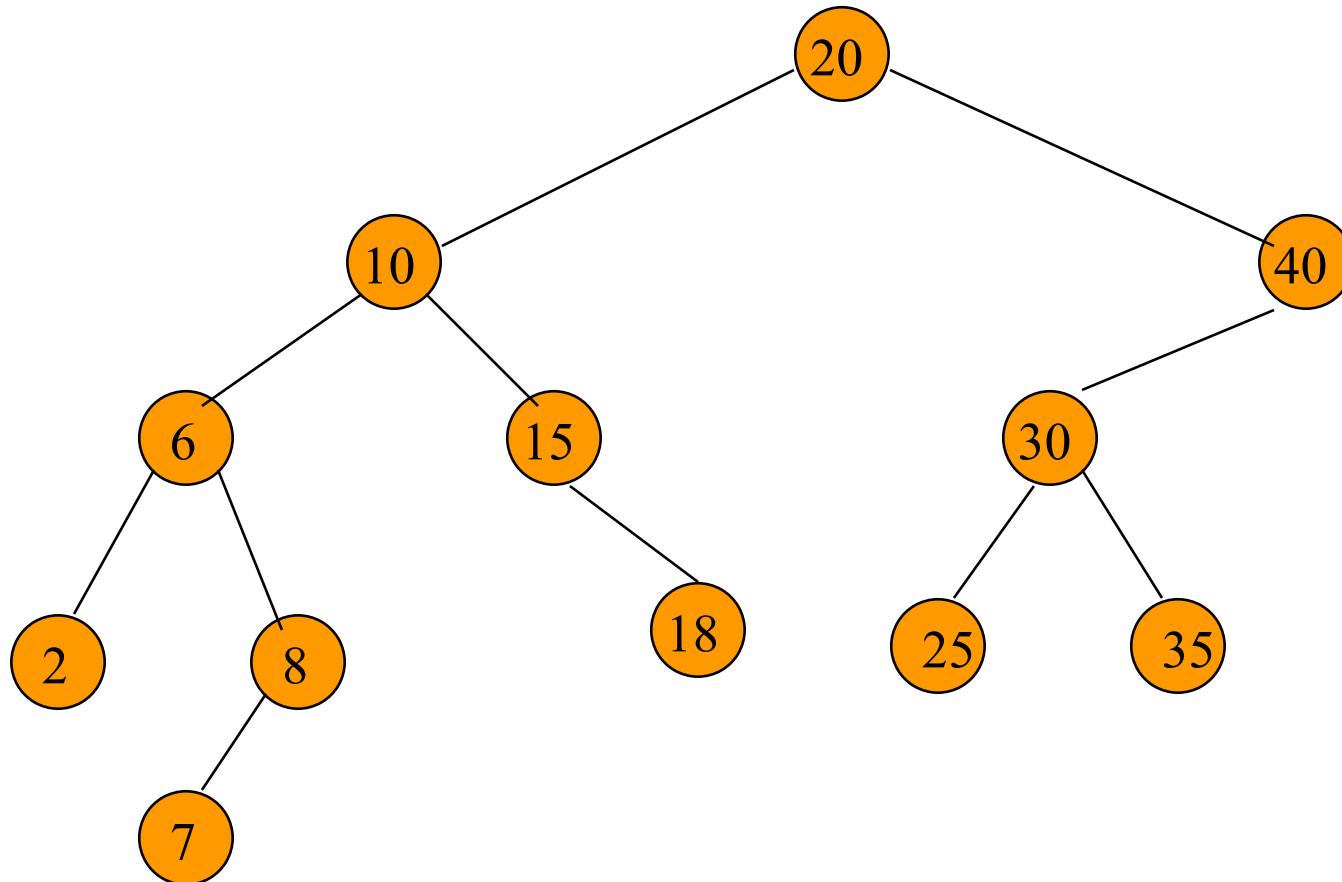
Replace with largest key in left subtree (or
smallest in right subtree).

Remove From A Degree 2 Node



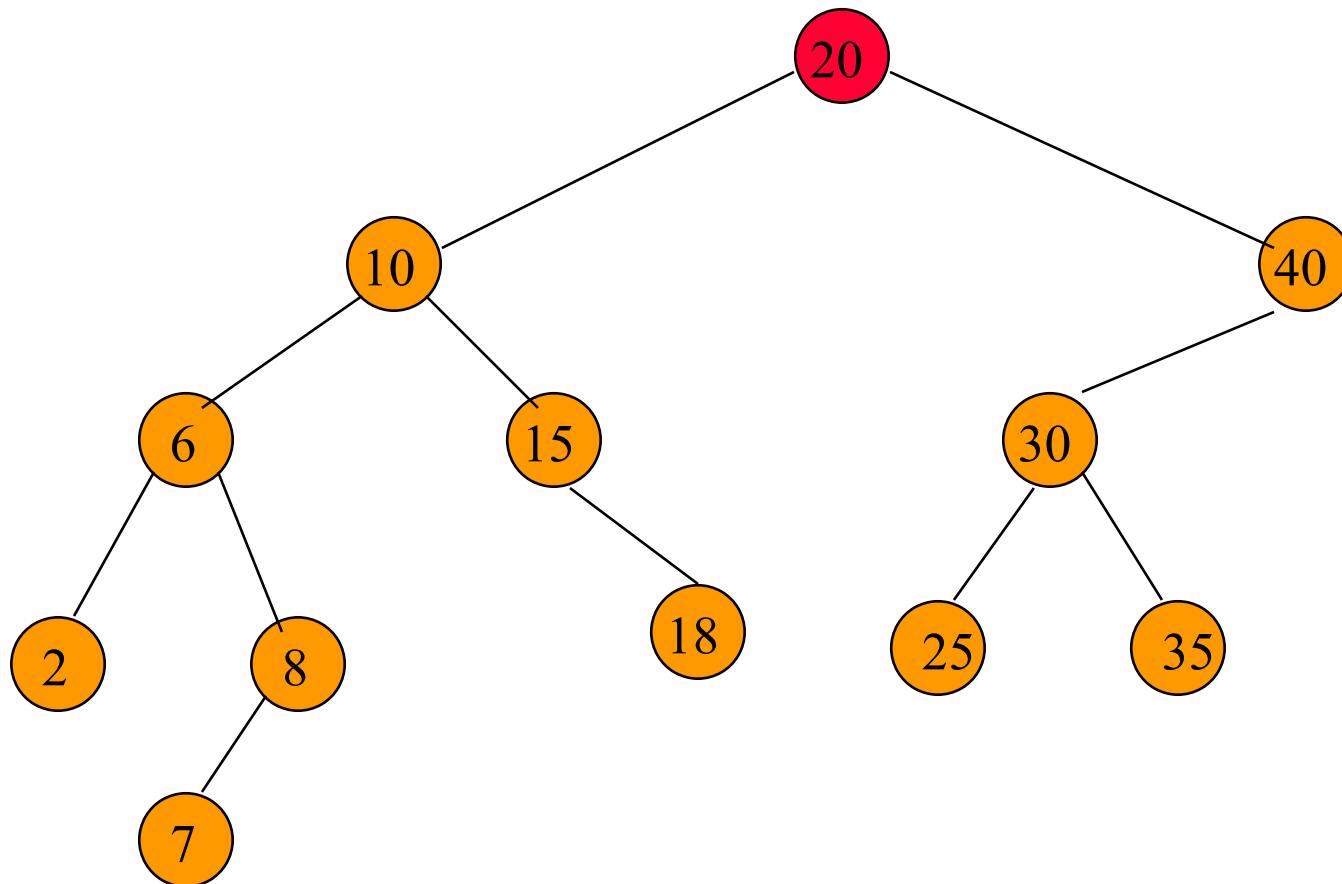
Largest key must be in a leaf or degree 1 node.

Another Remove From A Degree 2 Node



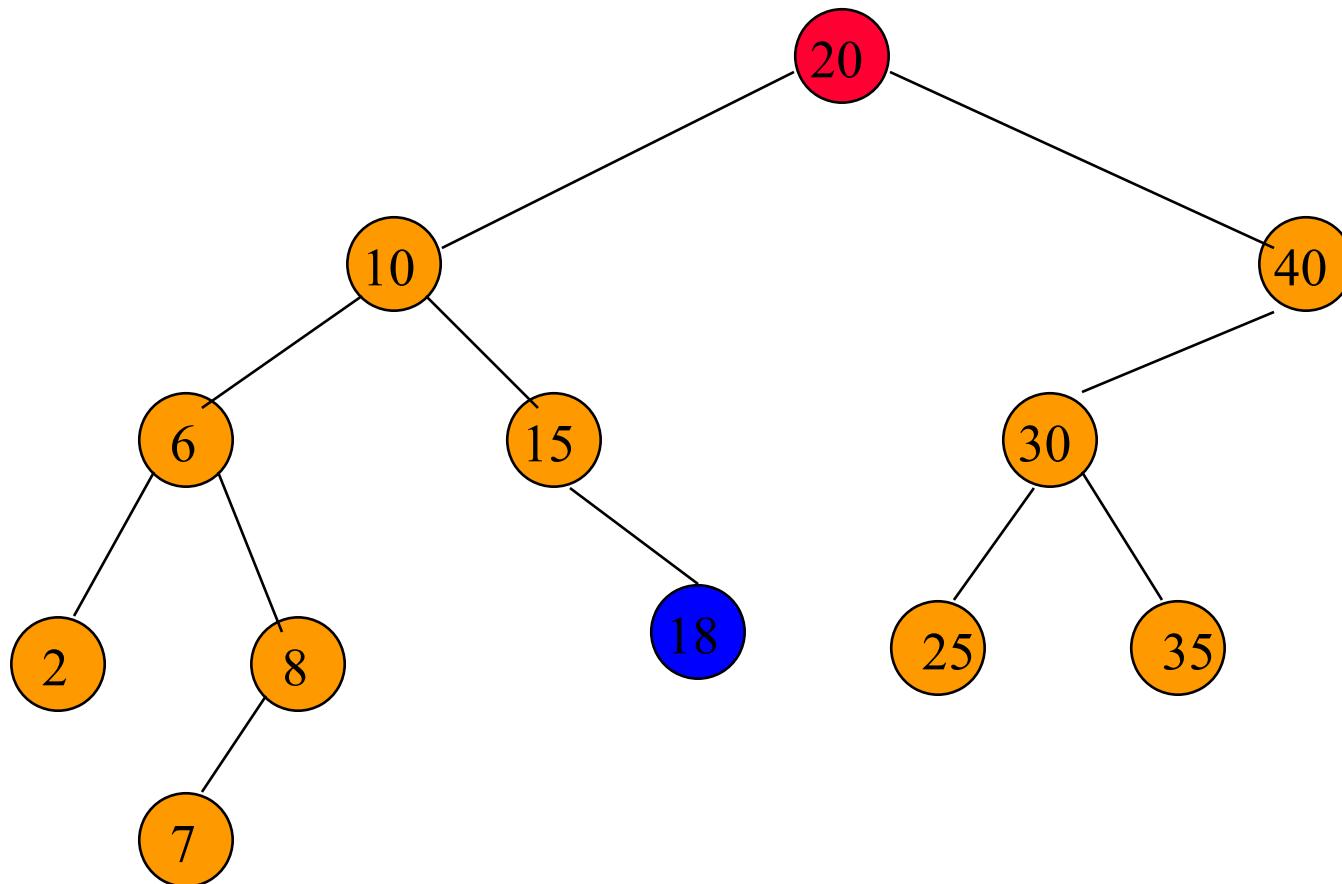
Remove from a degree 2 node. key = 20

Remove From A Degree 2 Node



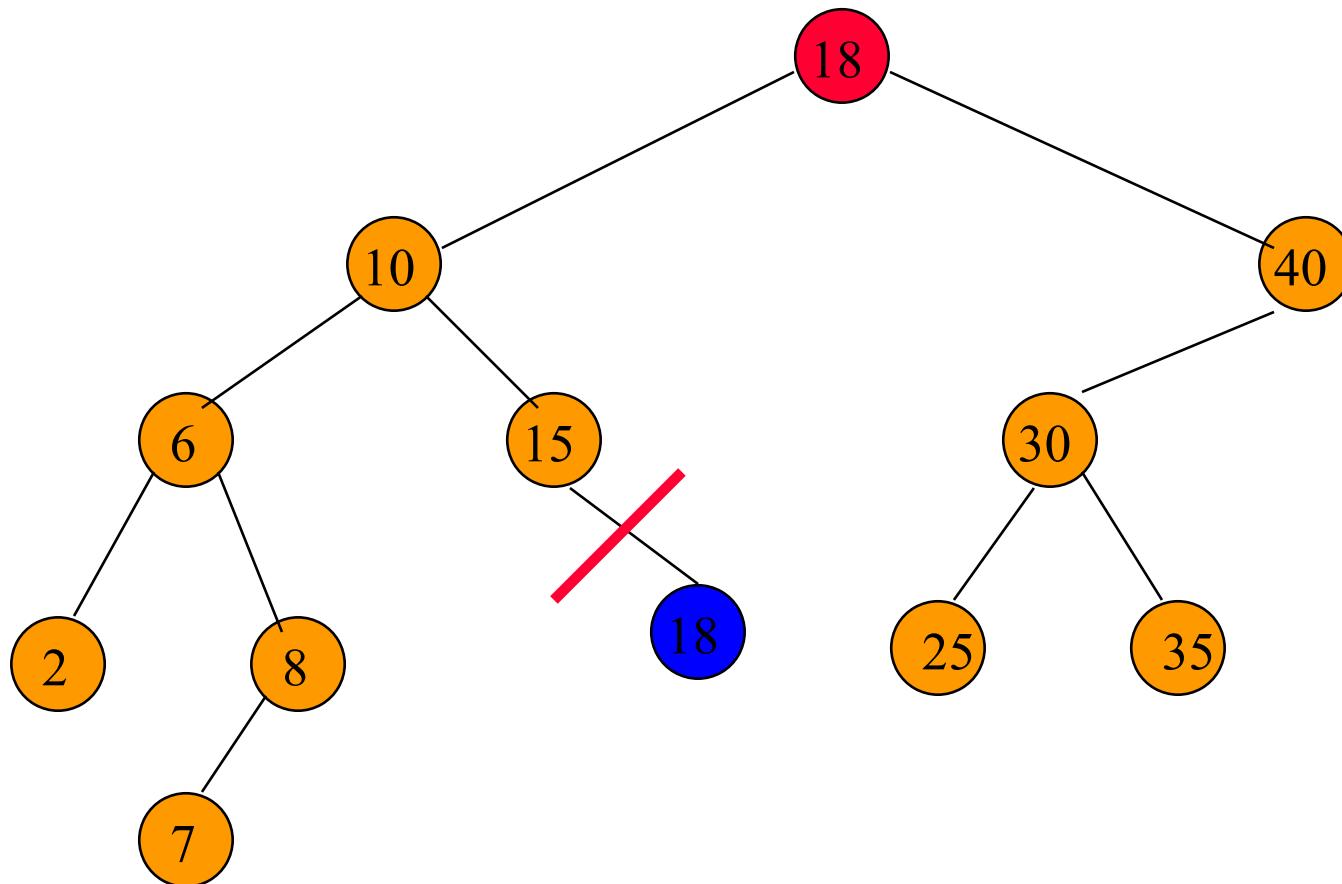
Replace with largest in left subtree.

Remove From A Degree 2 Node



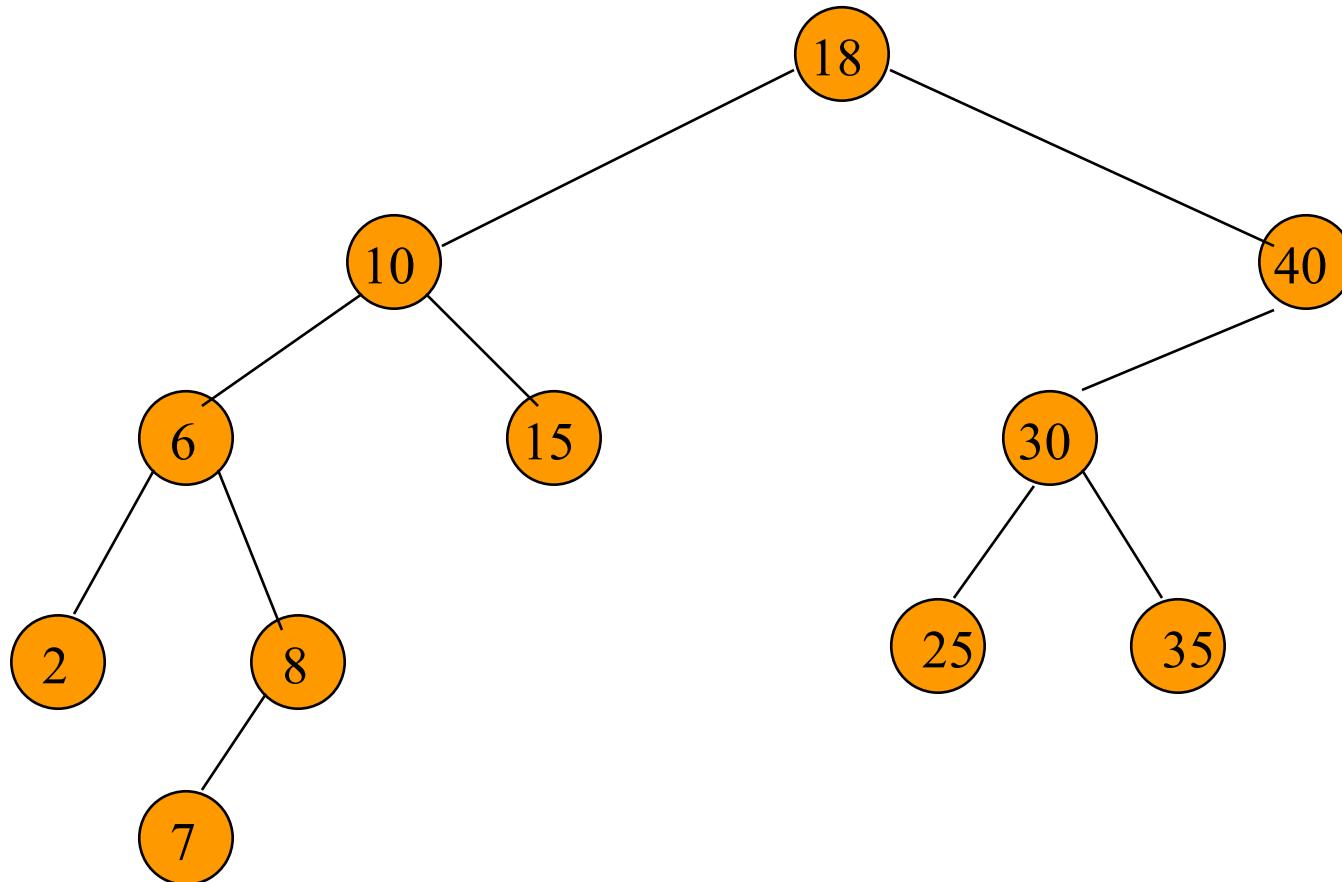
Replace with largest in left subtree.

Remove From A Degree 2 Node



Replace with largest in left subtree.

Remove From A Degree 2 Node

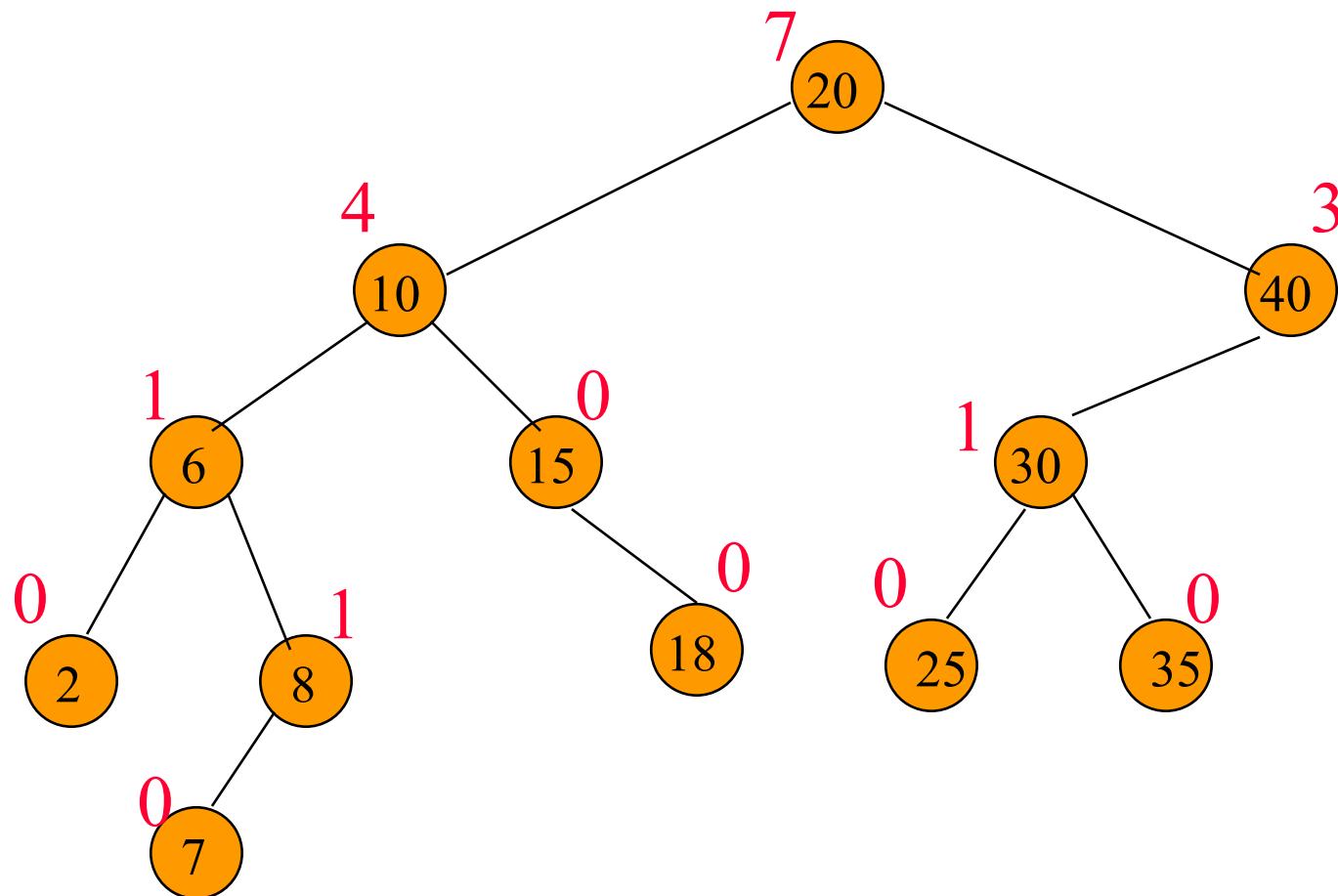


Complexity is $O(\text{height})$.

Indexed Binary Search Tree

- Binary search tree.
- Each node has an additional field.
 - **leftSize** = number of nodes in its left subtree

Example Indexed Binary Search Tree



leftSize values are in red

leftSize And Rank

Rank of an element is its position in inorder
(inorder = ascending key order).

[2,6,7,8,10,15,18,20,25,30,35,40]

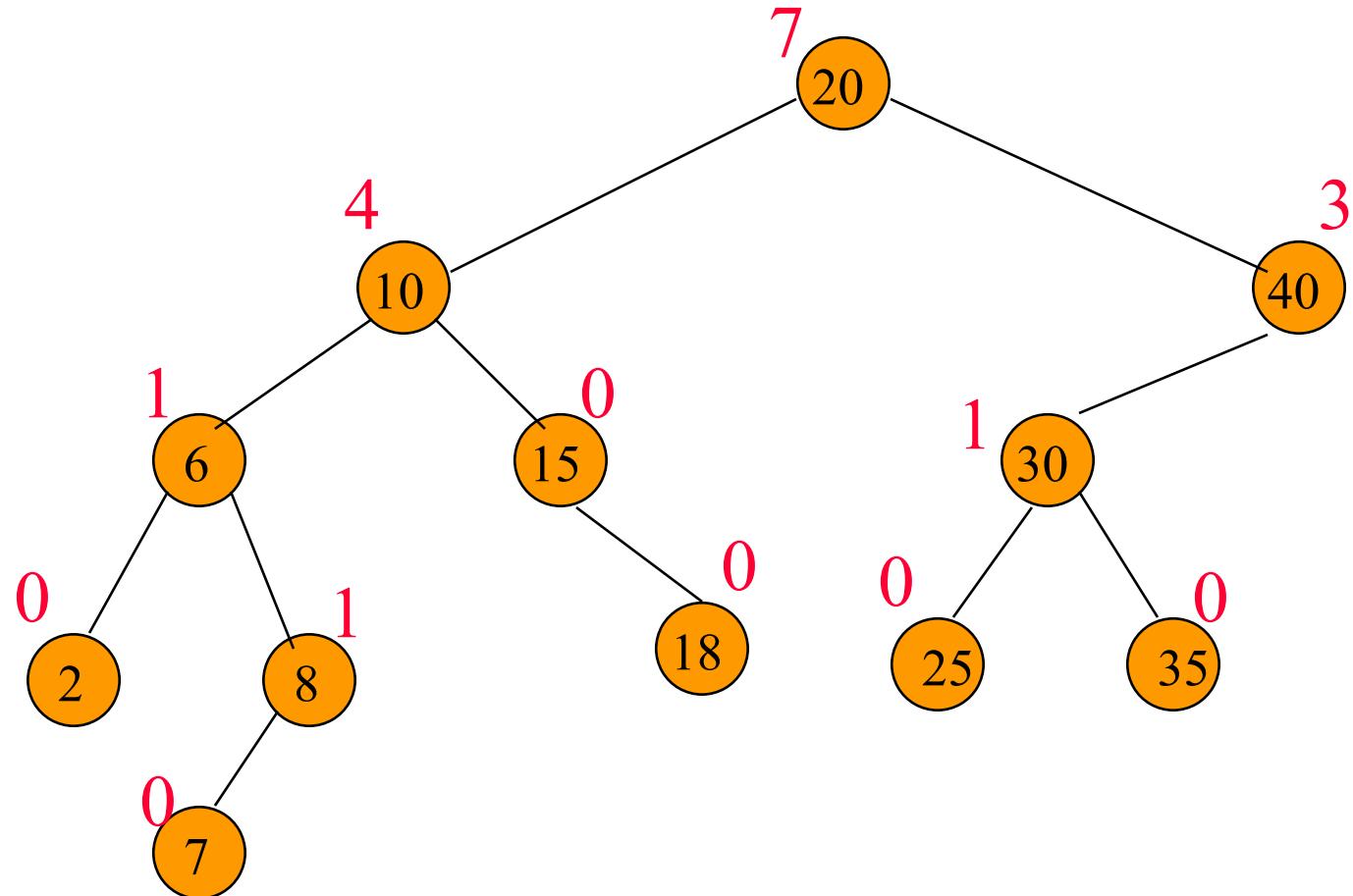
rank(2) = 0

rank(15) = 5

rank(20) = 7

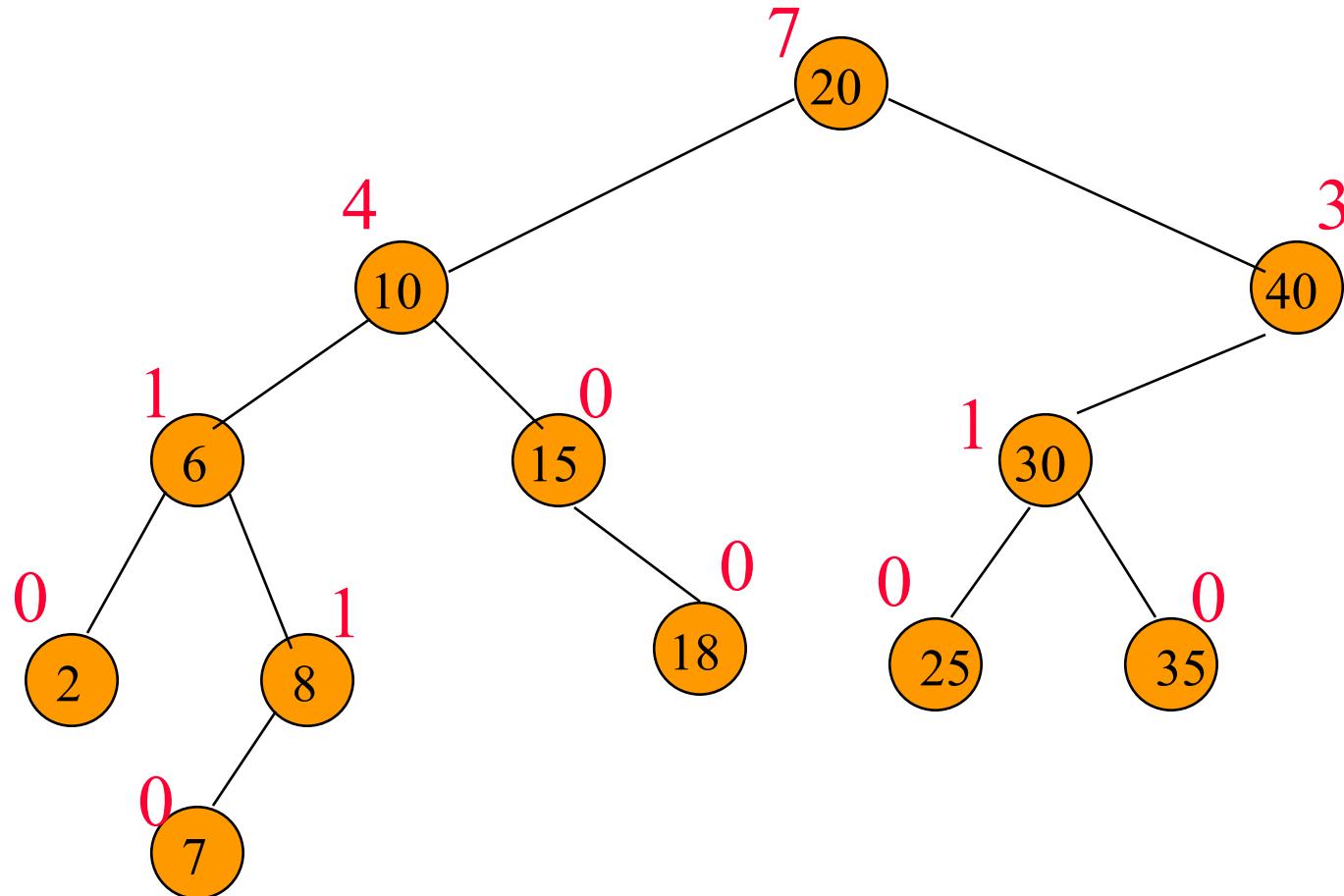
leftSize(x) = rank(x) with respect to elements in subtree rooted at x

leftSize And Rank



sorted list = [2,6,7,8,10,15,18,20,25,30,35,40]

get(index) And remove(index)



sorted list = [2,6,7,8,10,15,18,20,25,30,35,40]

get(index) And remove(index)

- if $\text{index} = \text{x.leftSize}$ desired element is x.element
- if $\text{index} < \text{x.leftSize}$ desired element is index 'th element in left subtree of x
- if $\text{index} > \text{x.leftSize}$ desired element is $(\text{index} - \text{x.leftSize}-1)$ 'th element in right subtree of x

Applications

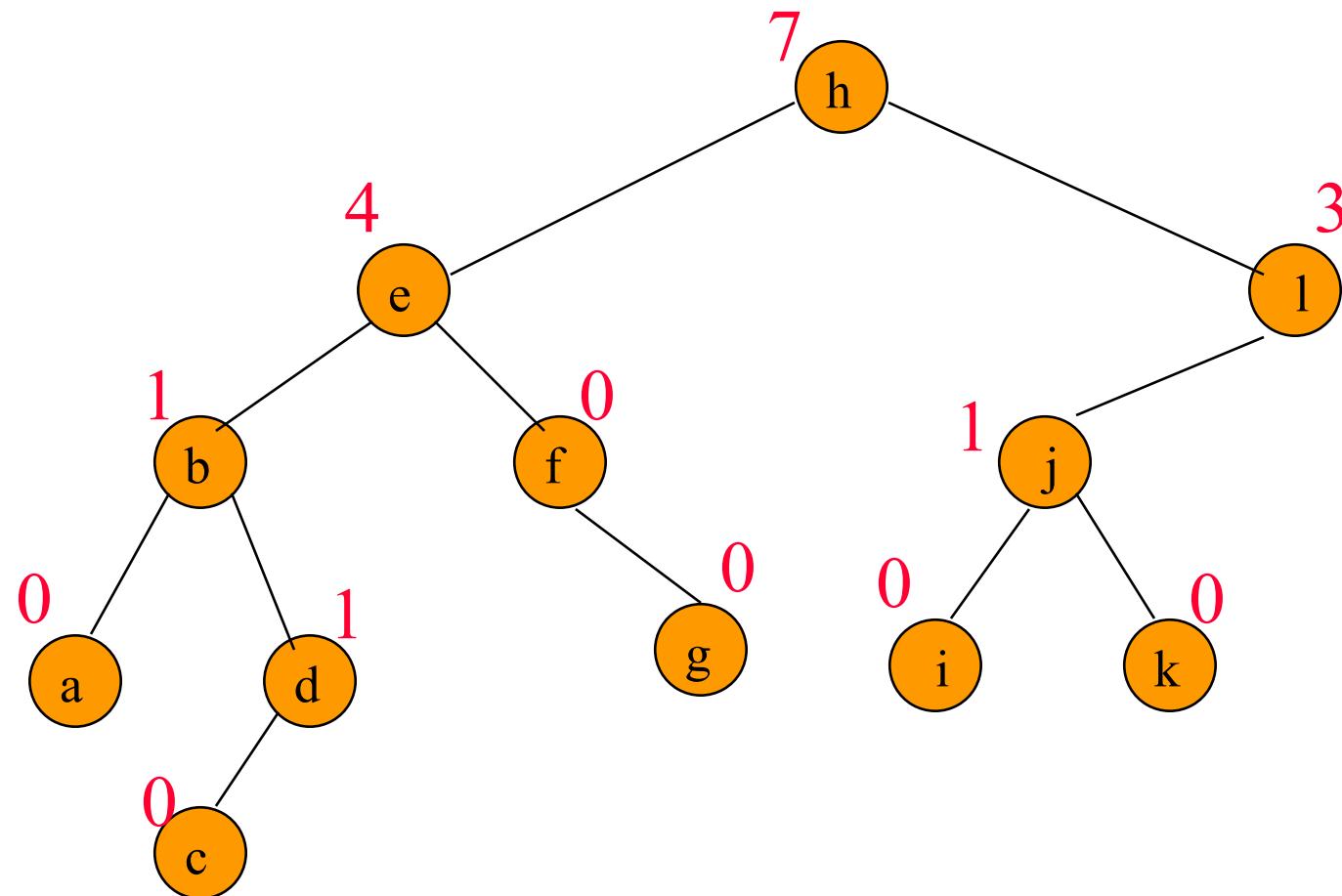
(Complexities Are For Balanced Trees)

Best-fit bin packing in $O(n \log n)$ time.

Representing a linear list so that `get(index)`,
`add(index, element)`, and `remove(index)`
run in $O(\log(\text{list size}))$ time (uses an
indexed binary tree, not indexed binary
search tree).

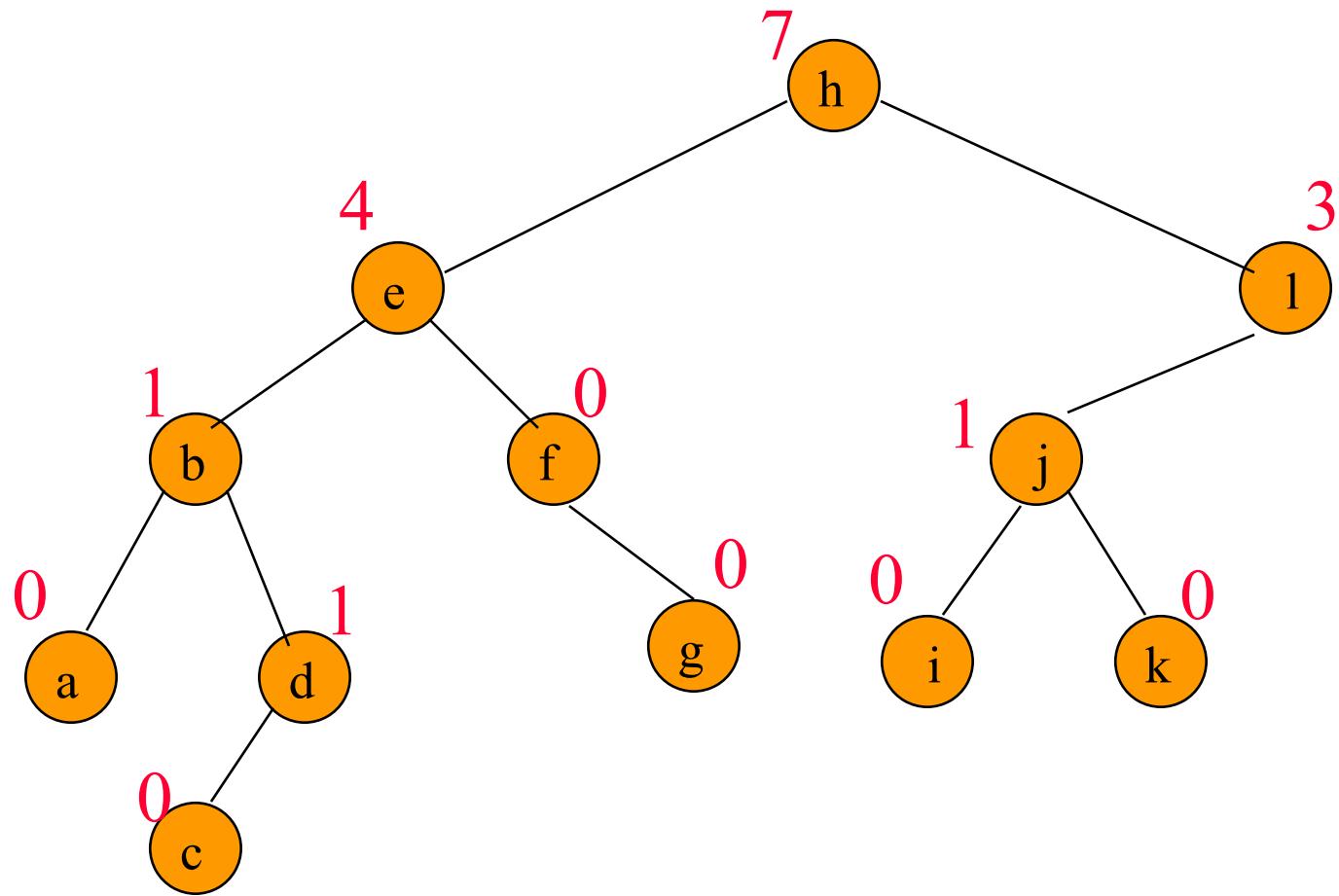
Can't use hash tables for either of these
applications.

Linear List As Indexed Binary Tree



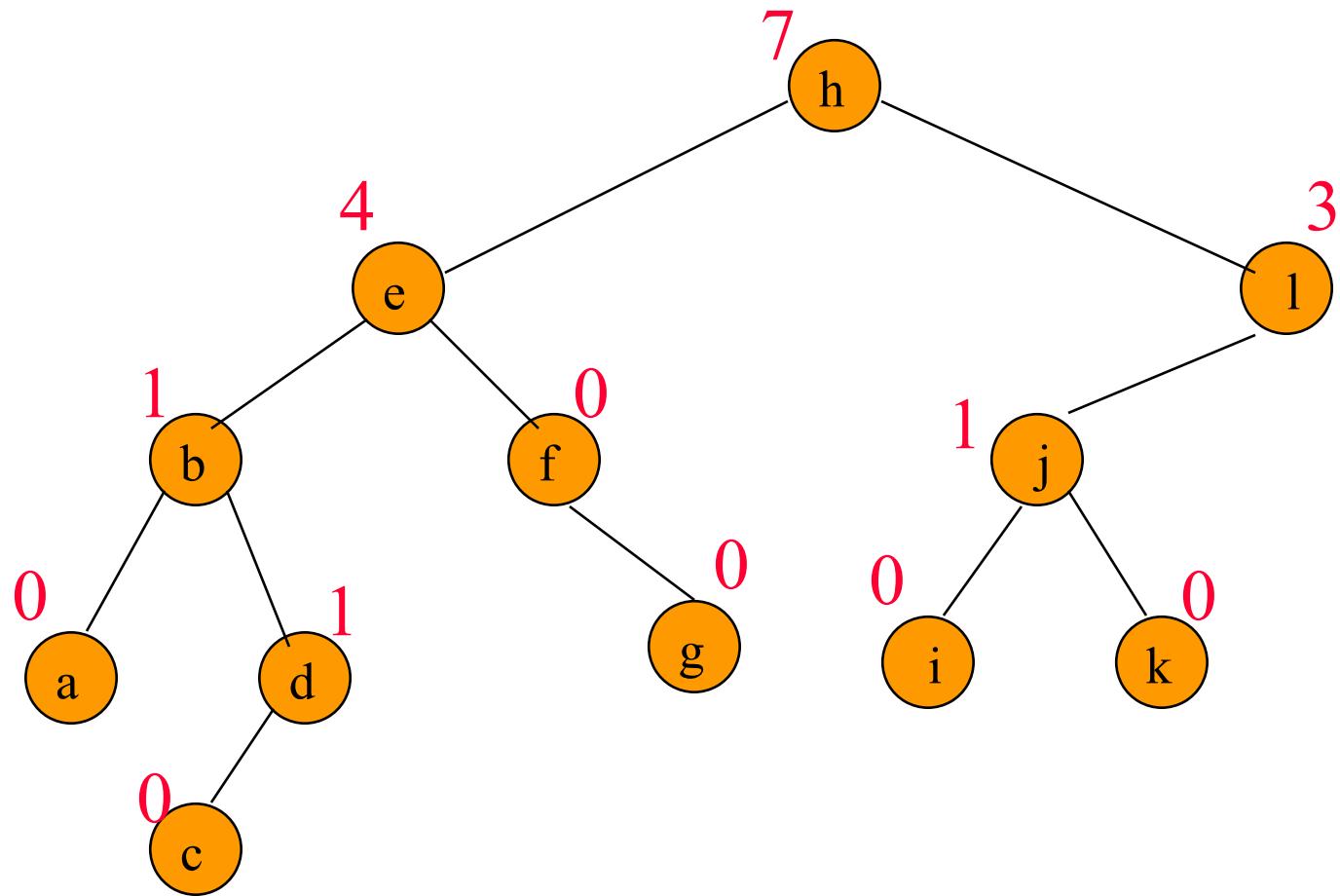
list = [a,b,c,d,e,f,g,h,i,j,k,l]

add(5,'m')



list = [a,b,c,d,e,f,g,h,i,j,k,l]

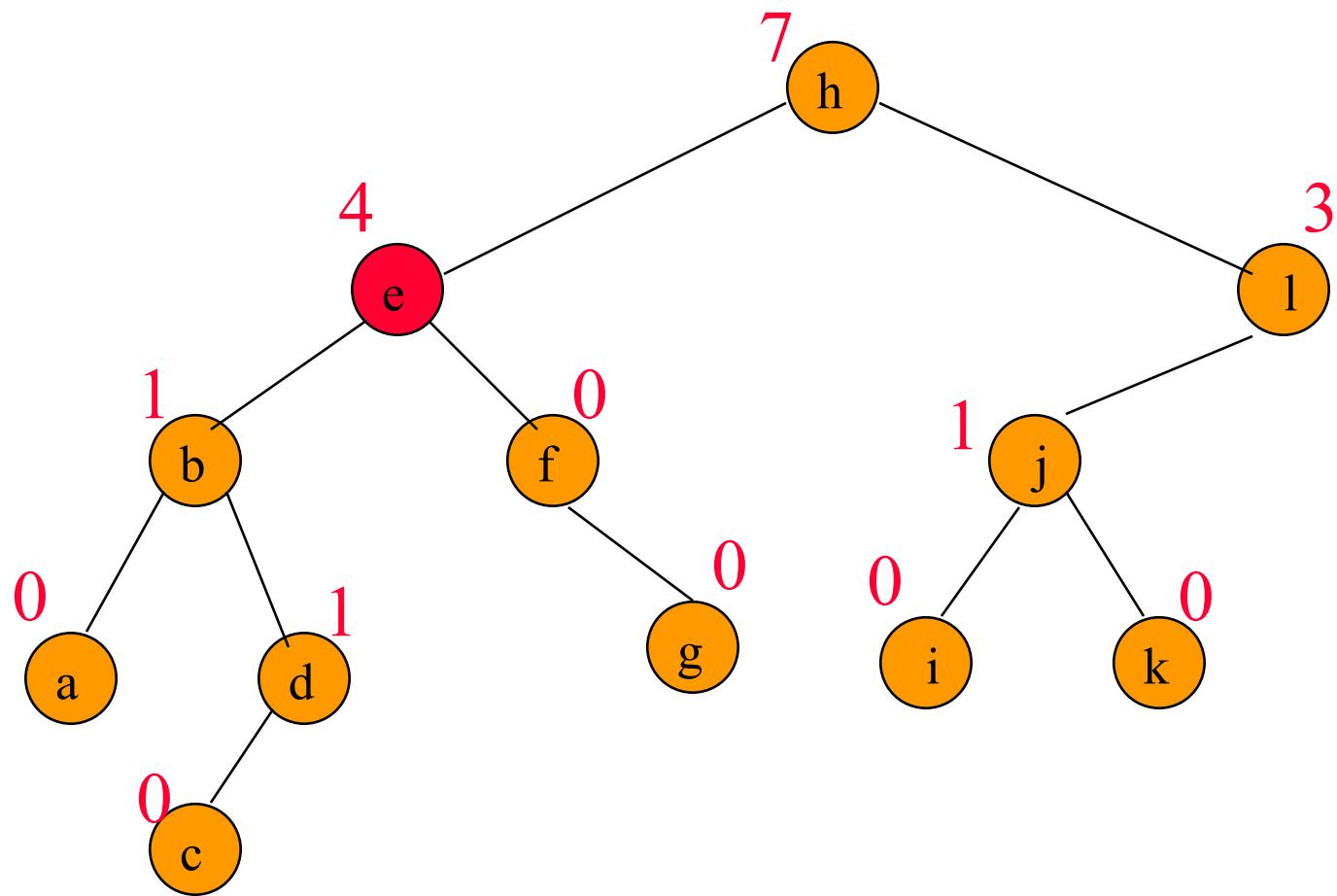
add(5,'m')



list = [a,b,c,d,e, m,f,g,h,i,j,k,l]

find node with element 4 (e)

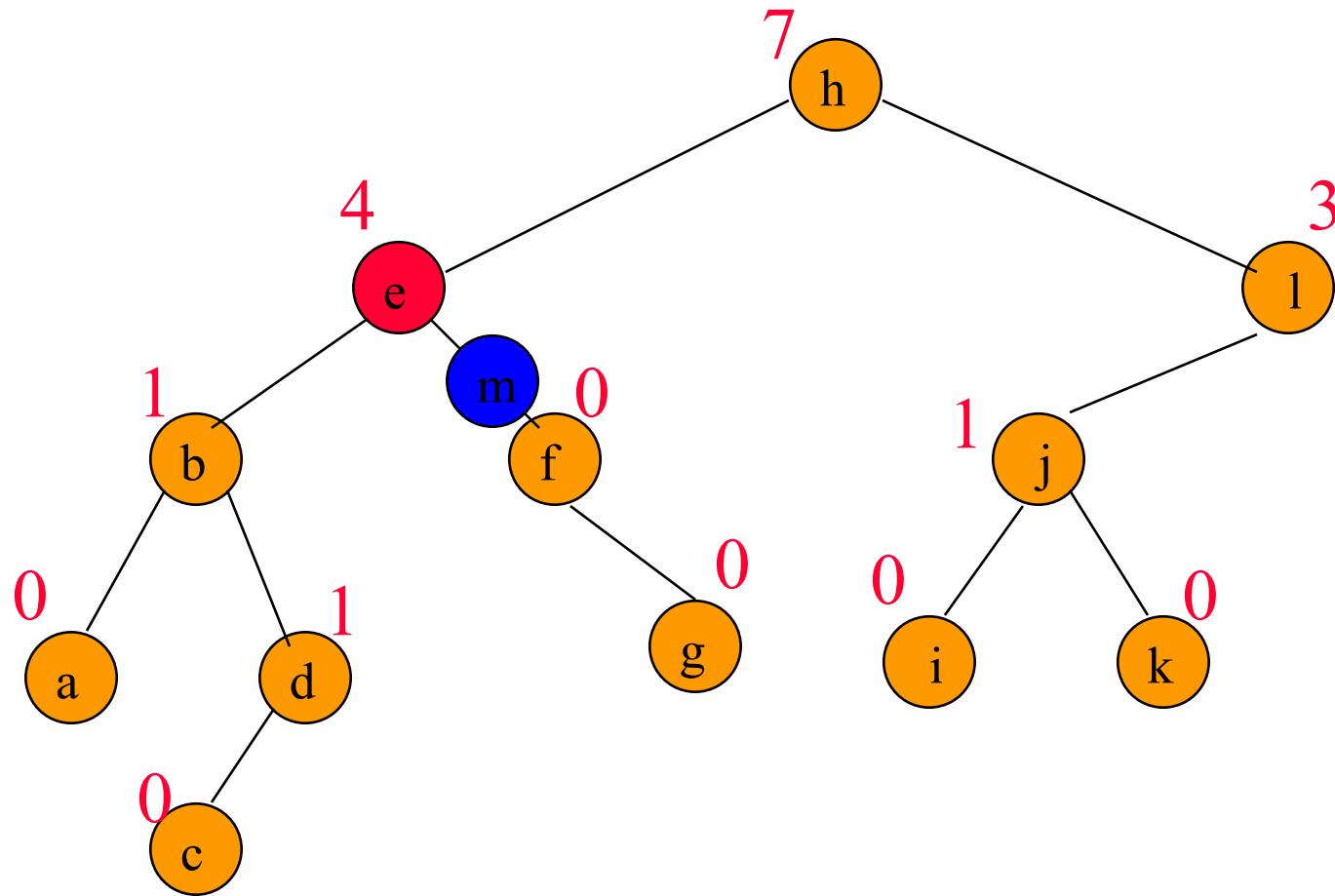
add(5,'m')



list = [a,b,c,d,e, m,f,g,h,i,j,k,l]

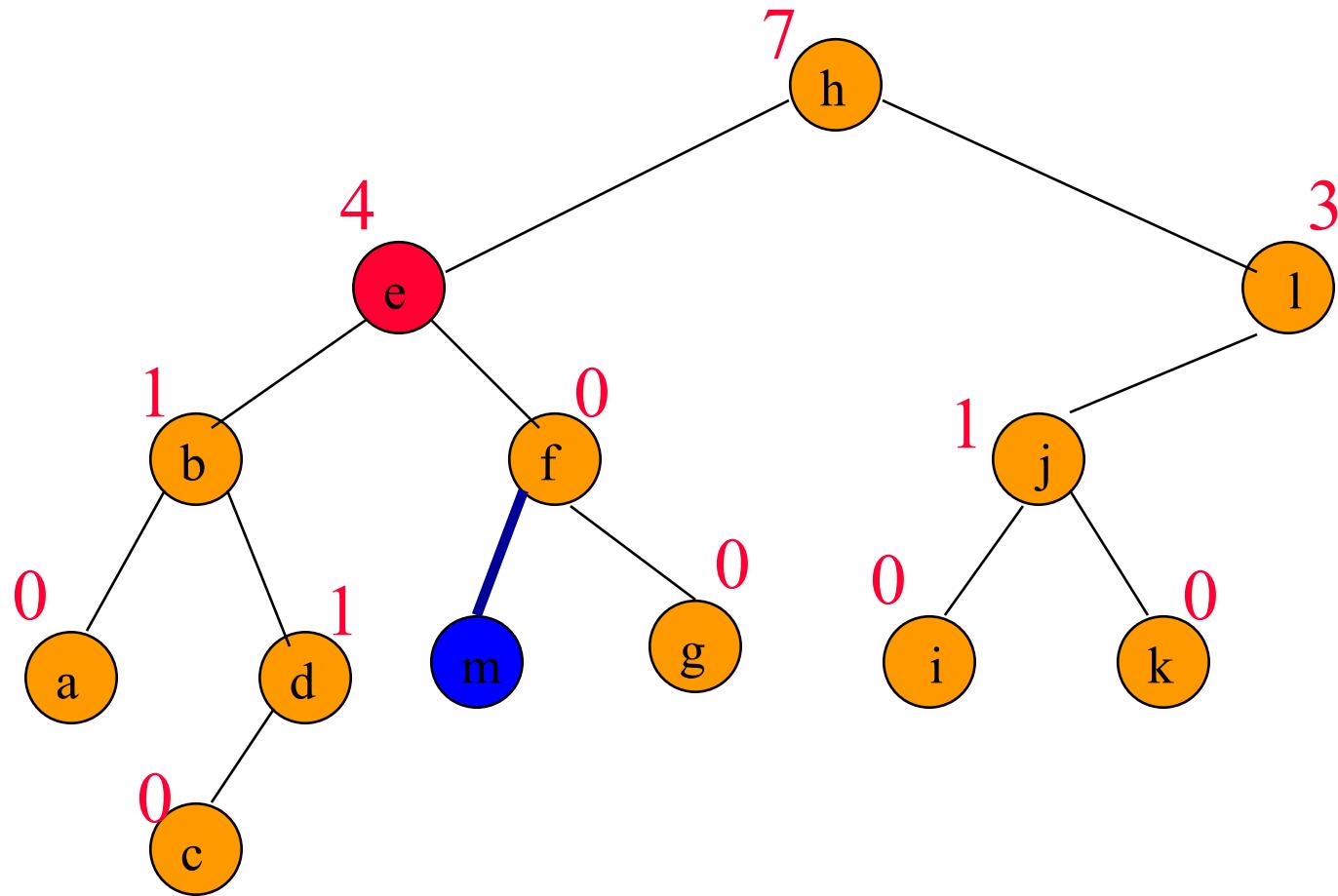
find node with element 4 (e)

add(5,'m')



add m as right child of e; former right subtree of e becomes right subtree of

$\text{add}(5, \text{'m'})$



add m as leftmost node in right subtree
of e

add(5,'m')

- Other possibilities exist.
- Must update some **leftSize** values on path from root to new node.
- Complexity is **O(height)**.