ADVANCED DATA STRUCTURES AND ALGORITHMS

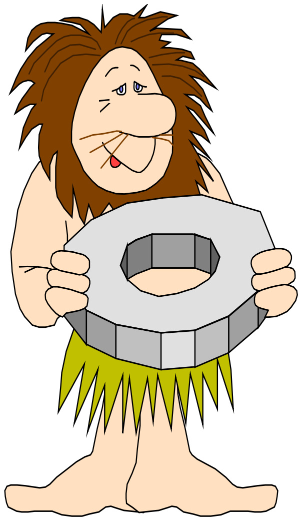
Associate Professor Dr. Raed Ibraheem Hamed

University of Human Development, College of Science and Technology
Computer Science Department

2015 – 2016

What this Lecture is about:

- Expression Trees
- Height and Depth
- Balanced Binary Trees
- 2-3 Tree
- 2-3 Tree: *Definition*
- 2-3 Tree: *Example*
- 2-3 Tree: *Efficiency*



Binary arithmetic expressions

A binary arithmetic expression is made up of numbers joined by binary operations $*$, $+$, $/$, $-$

$(6\times(3+4))$ by using parentheses we can indicate the order in which the operations are to be done. In this example the parentheses indicate that the addition is to be done before the multiplication.

$(6(3+4))$ This is another way of writing $(6\times(3+4))$.

$(6\times(3+4)) = (6\times7)=42$

Expression Trees

Arithmetic expressions are representable by labeled trees, and it is often quite helpful to visualize expressions as trees.

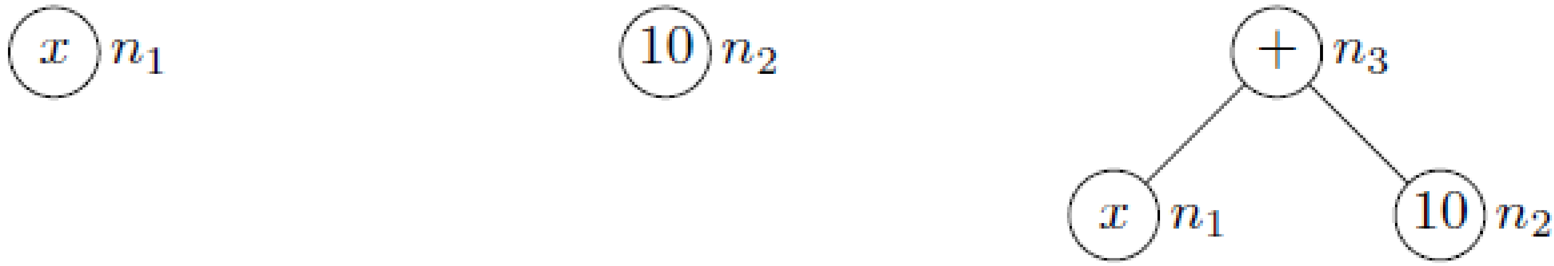
$i)$	x	$iv)$	$(-(x+10))$
$ii)$	10	$v)$	y
$iii)$	$(x+10)$	$vi)$	$\Big(y \times (-(x+10))\Big)$

Expression Trees

$i)\quad x$

$ii)\quad 10$

$iii)\quad (x + 10)$

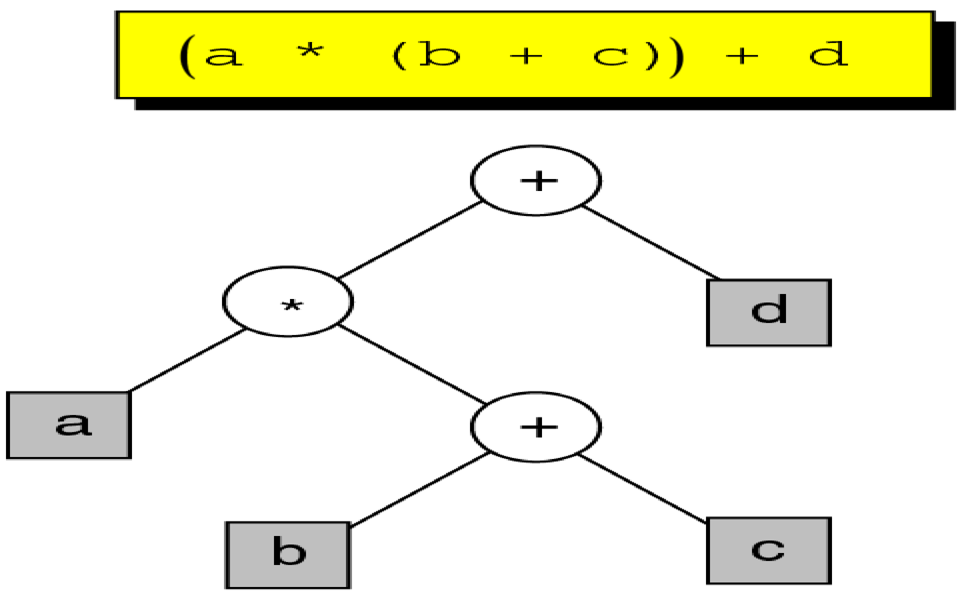


(a) For x. (b) For 10. (c) For $(x + 10)$.

Expression Trees

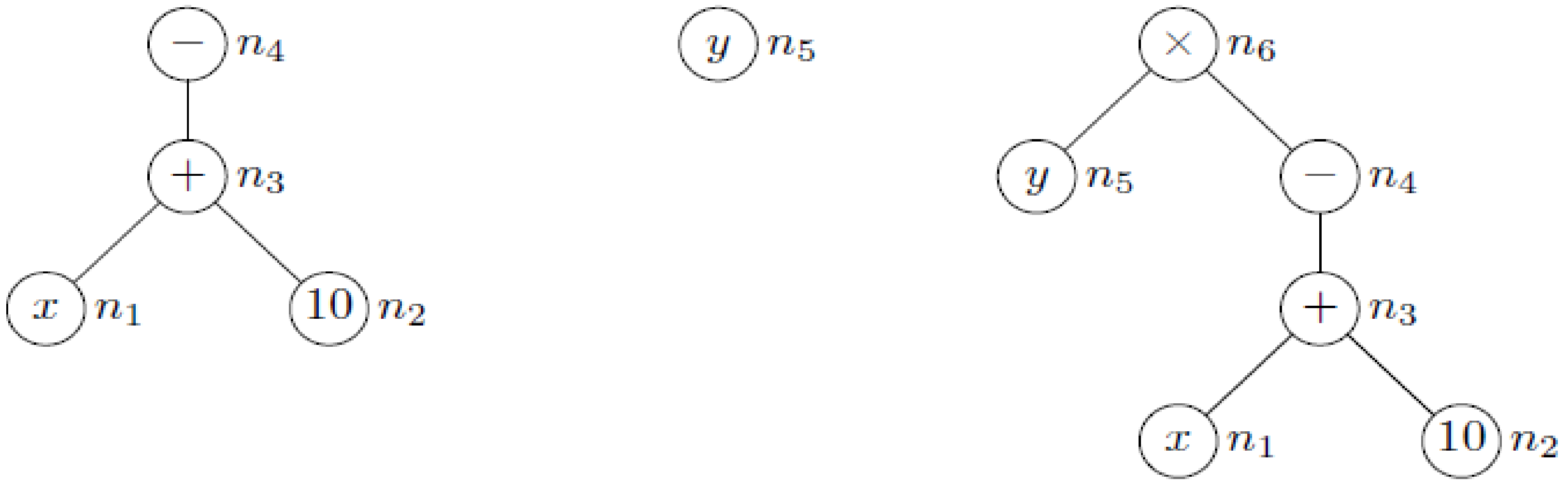
An expression tree is a binary tree with these properties:

1. **Each leaf is an operand.**
2. **The root and internal nodes are operators.**
3. **Subtrees are subexpressions with the root being an operator.**



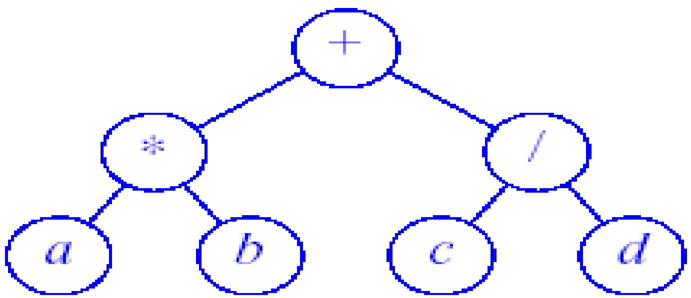
Expression Trees

$$
\begin{array}{rl}
iv) & \big(-(x+10)\big) \\
v) & y \\
vi) & \Big(y \times \big(-(x+10)\big)\Big)
\end{array}
$$

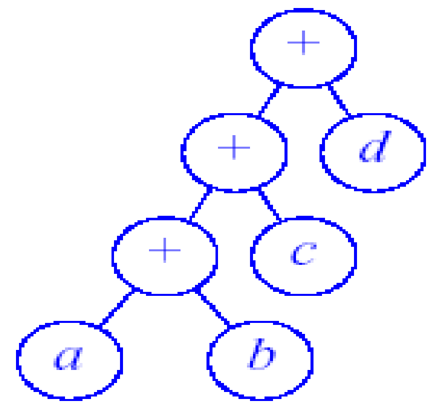


(d) For $(-(x+10))$. (e) For y. (f) For $(y \times (-(x+10)))$.

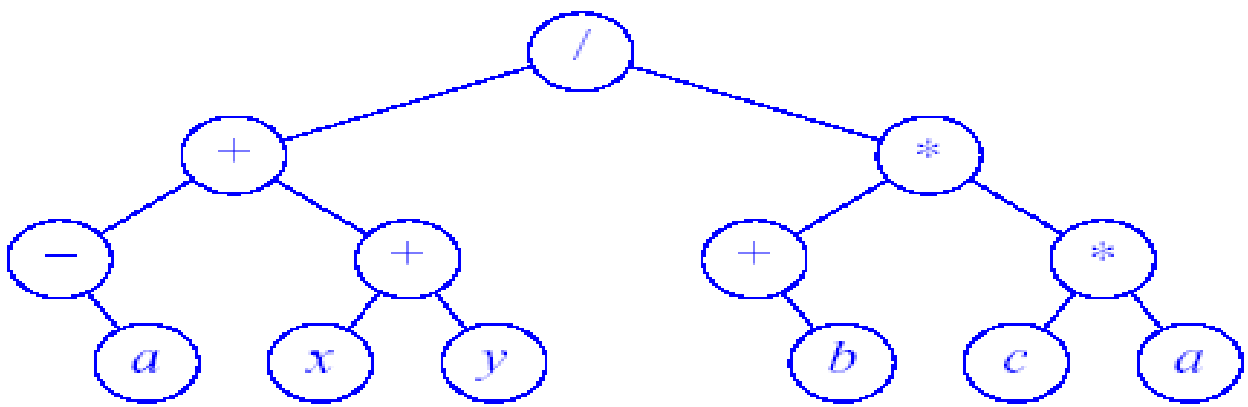
Expression Trees



(a) $(a * b) + (c / d)$



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

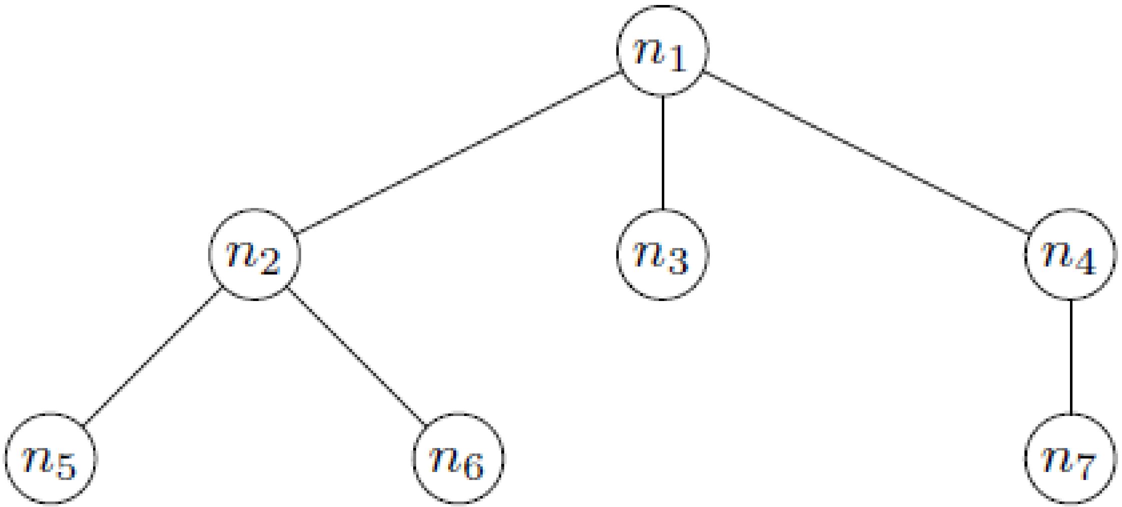


(c) $((-a) + (x + y)) / ((+b) * (c * a))$

Height and Depth

In a tree, the **height** of a node *n* is the length of a longest path from *n* to a *leaf*.

The height of the tree is the height of the root. The depth, or level, of a node *n* is the length of the path from the *root* to *n*.

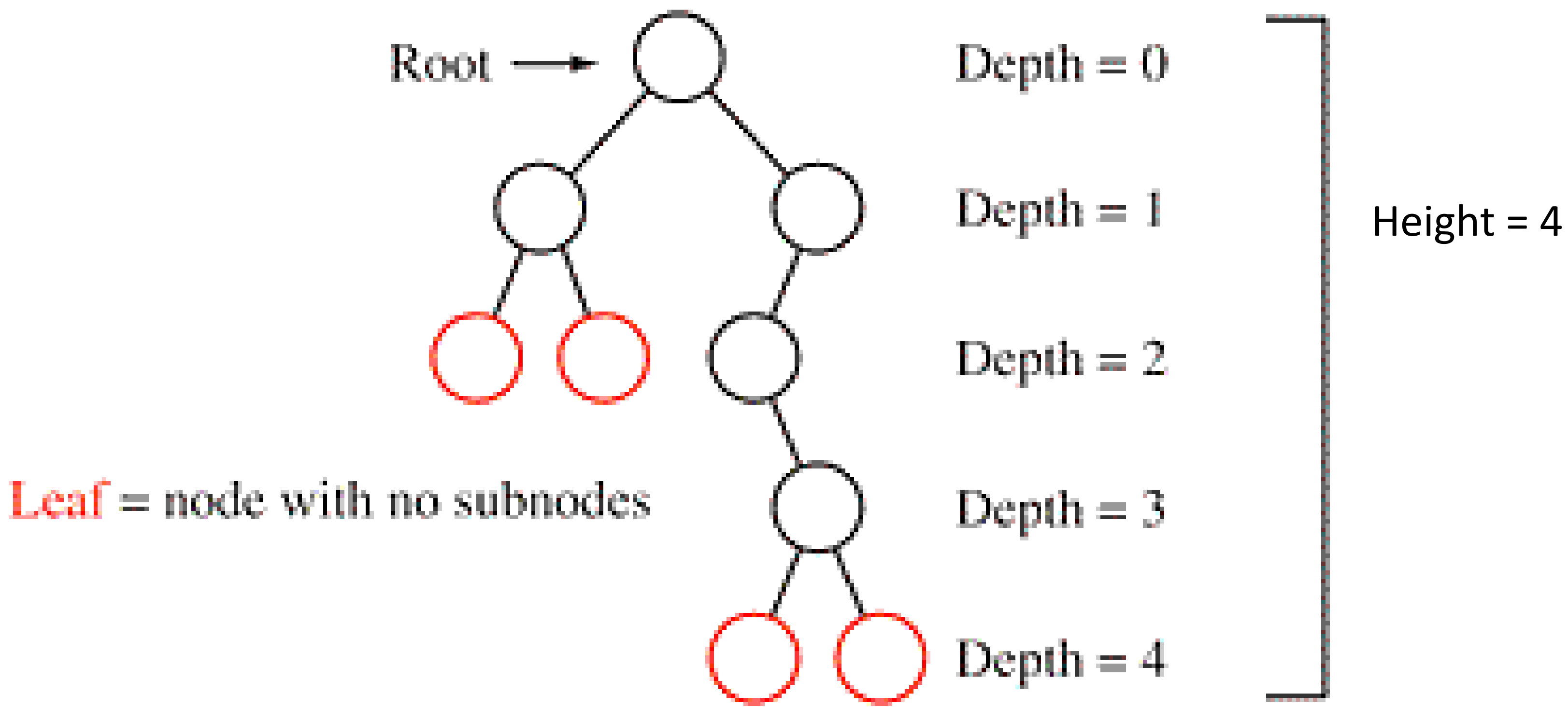


Tree with seven nodes.

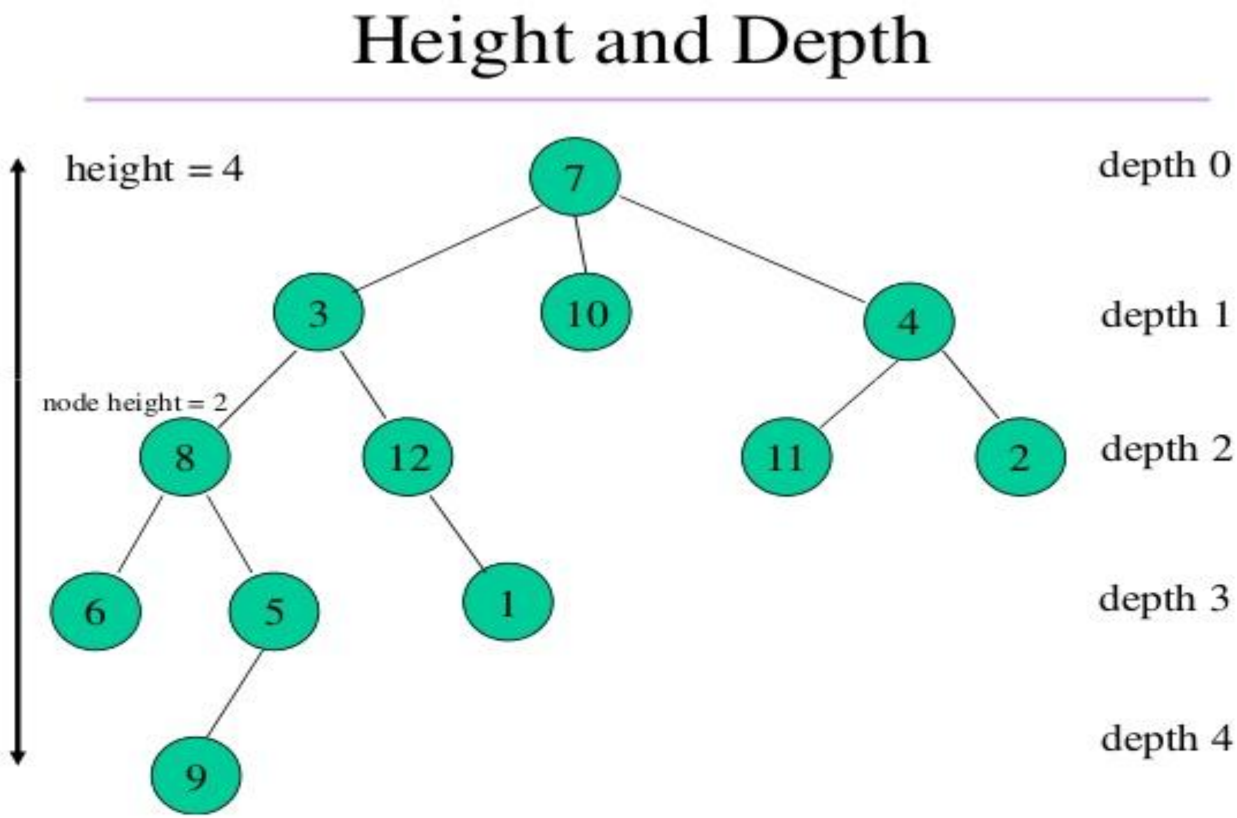
Height and Depth

- In this figure , node *n*1 has height 2, *n*2 has height 1, and leaf *n*3 has height 0.
- In fact, any leaf has height 0.
- The tree in this figure has height **2**.
- The depth of *n*1 is 0, the depth of *n*2 is 1, and the depth of ***n*5** is **2**.

Height and Depth

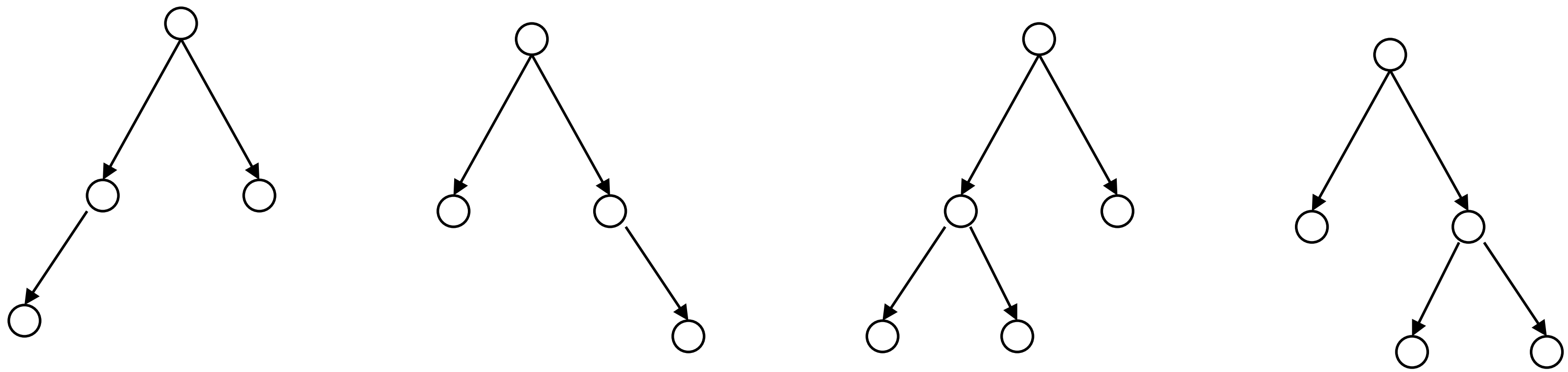


Height and Depth



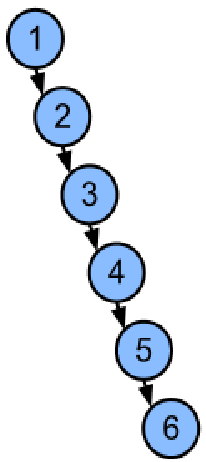
Balanced Binary Trees

- Recall that a *binary tree* is ***balanced*** if the difference in **height** between any **node's left and right subtree is ≤ 1**.

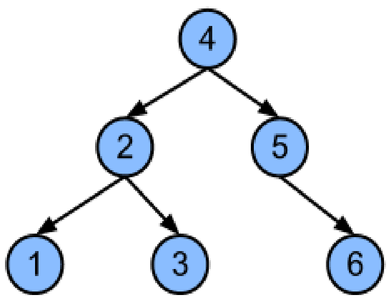


Balanced Binary Trees

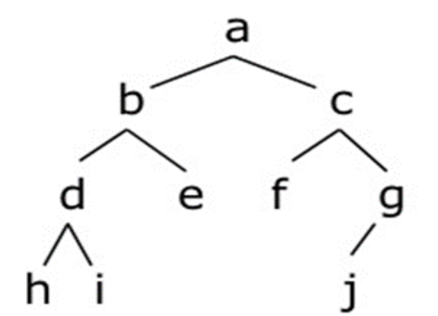
Non-balanced



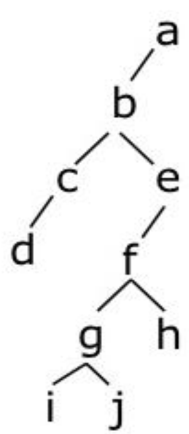
Balanced



Example 1



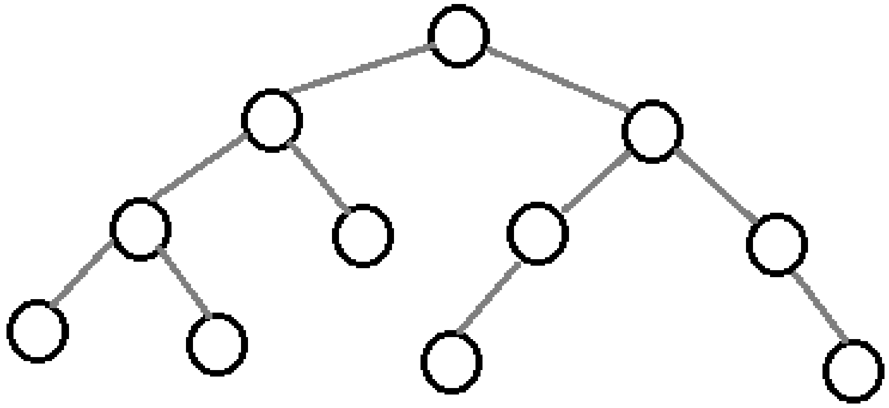
A balanced binary tree



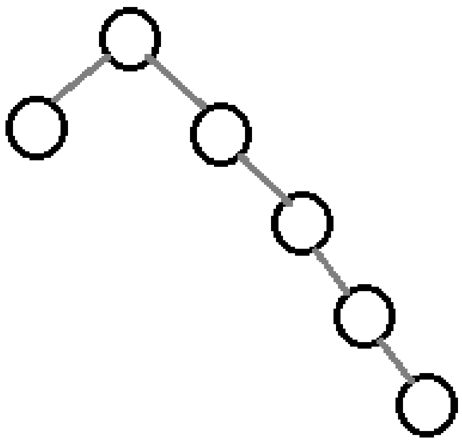
An unbalanced binary tree

Example 2

Balanced Binary Trees



Balanced Binary Tree



Unbalanced Binary Tree

2-3 Tree

- Intuitively, a ***2-3 tree*** is a tree in which each parent node has either **2 or 3 children**, and all leaves are at the **same level**.

- According to Donald Knuth, *2-3 trees* were invented by John E. Hopcroft.

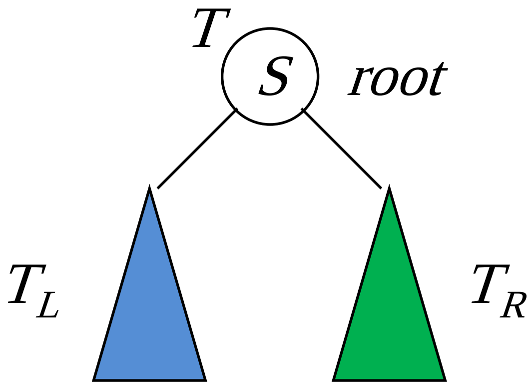
2-3 Tree: *Definition*

Formally, T is a 2-3 tree of height h if

a) T is ***empty*** (h = 0); OR

b) T consists of a *root* and ***2 subtrees***, T_L , T_R :

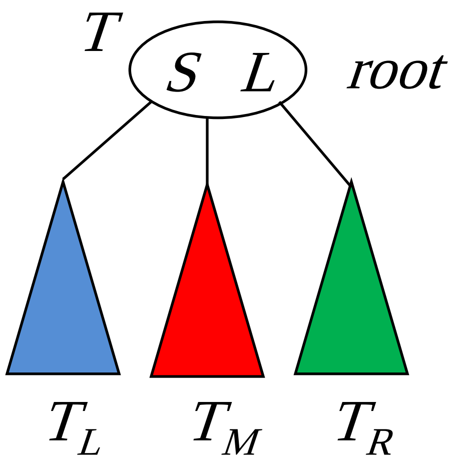
- T_L and T_R are both 2-3 trees, each of height h – 1,
- the *root* contains **one data item** with search key S,
- S > each search key in T_L ,
- S < each search key in T_R ; OR …



2-3 Tree: *Definition*

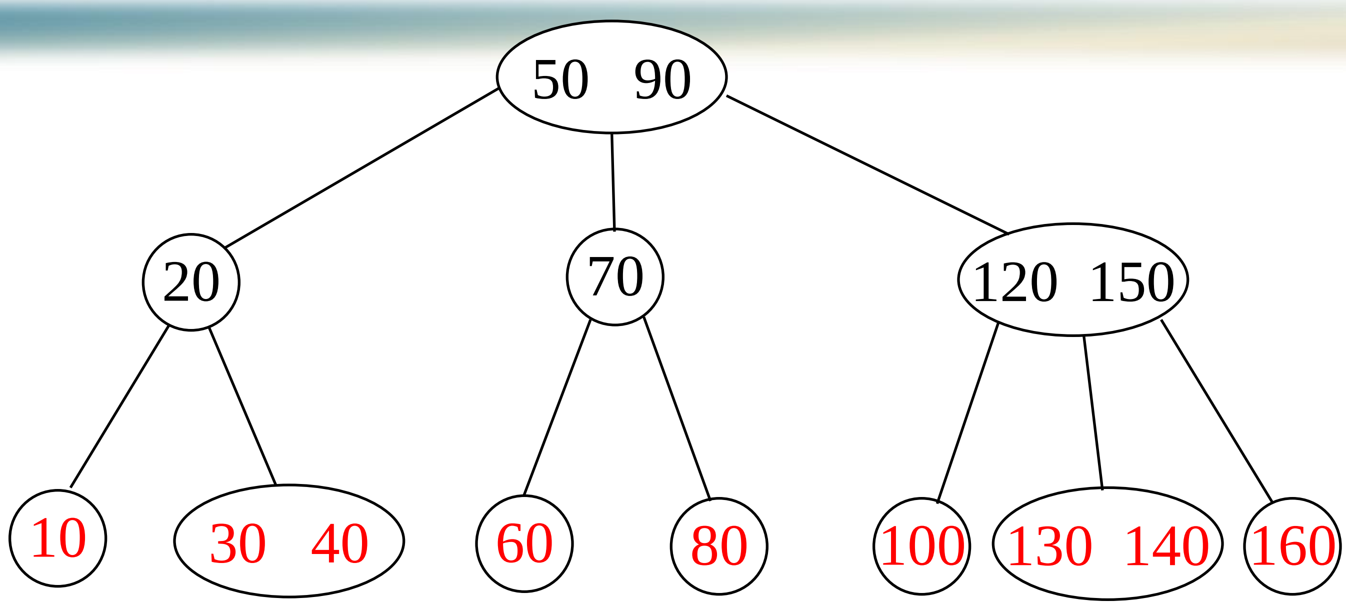
c) *T* consists of a *root* and ***3 subtrees***, T_L, T_M, T_R :

- T_L, T_M, T_R are all 2-3 trees, each of height $h - 1$,
- the *root* contains ***two*** data items with search keys *S* and *L*,
- each search key in T_L < S < each search key in T_M ,
- each search key in T_M < L < each search key in T_R .



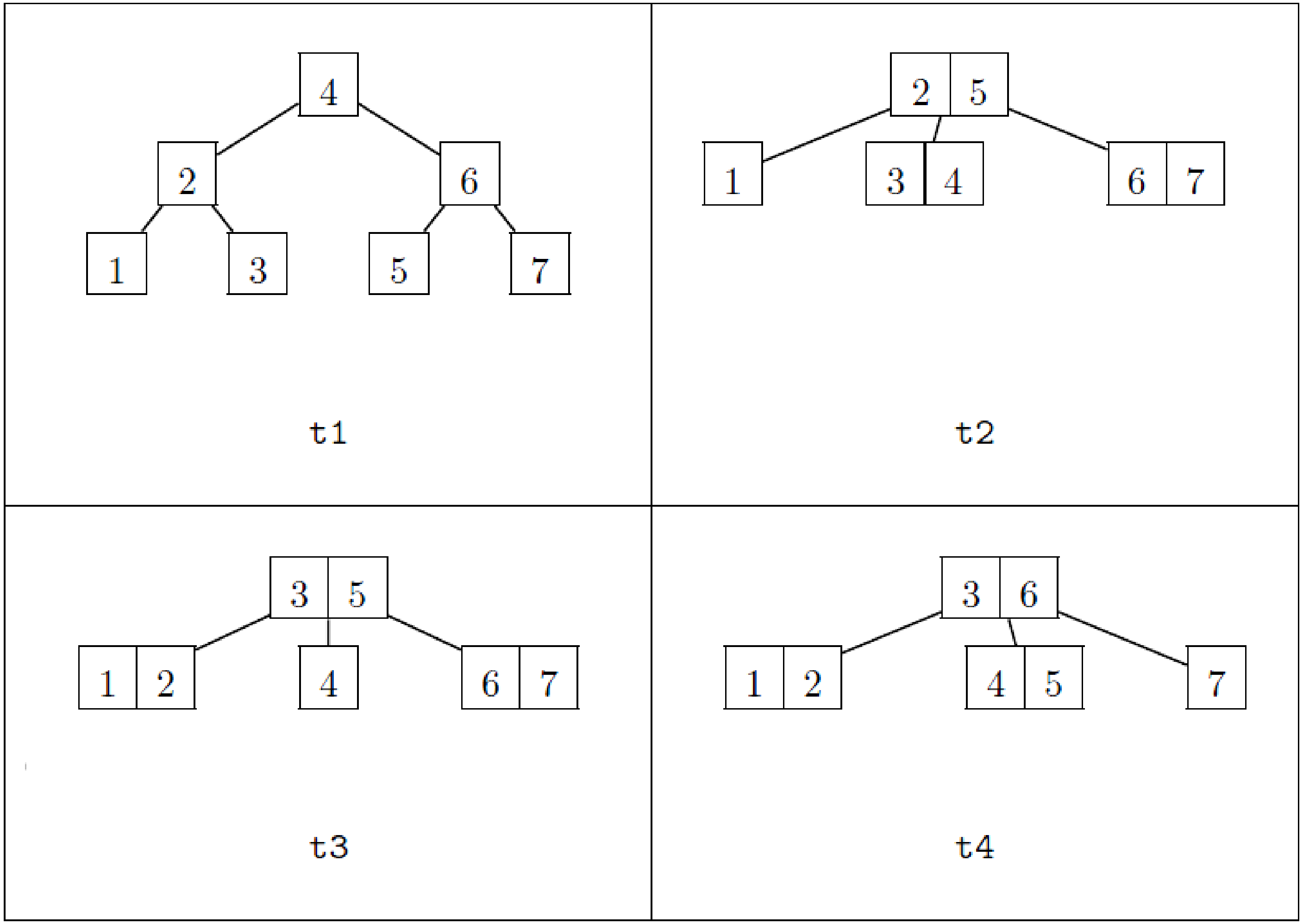
S is called the left value of the 3-node; **L** is called the right value of the 3-node; T_L is its left subtree; T_M is its middle subtree; and T_R is its right subtree.

2-3 Tree: *Example*



- Each non-leaf has 2 or 3 children,
- All leaves are at the same level, and
- Each node contains either one or two key values

2-3 Tree: Examples



2-3 Tree: *Efficiency*

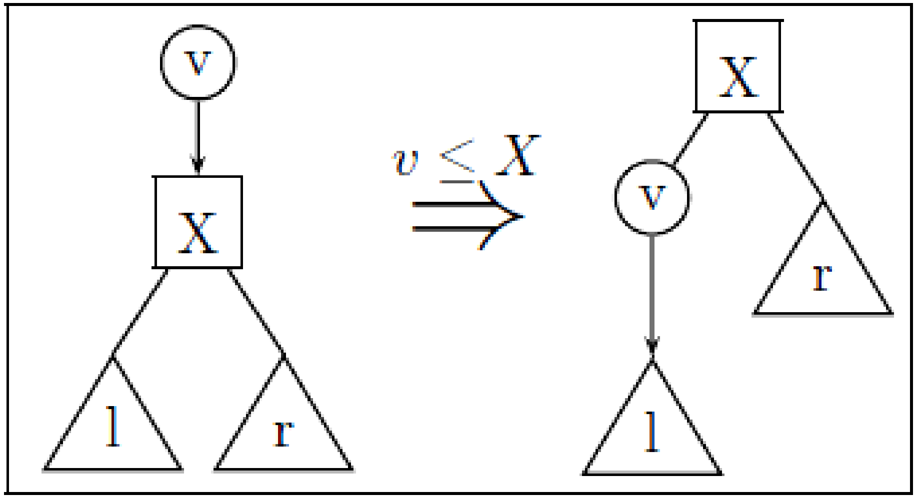
Insertion can be defined so that the 2-3 tree *remains balanced* and its other properties are maintained.

2-3 Tree Insertion: *Basic Idea*

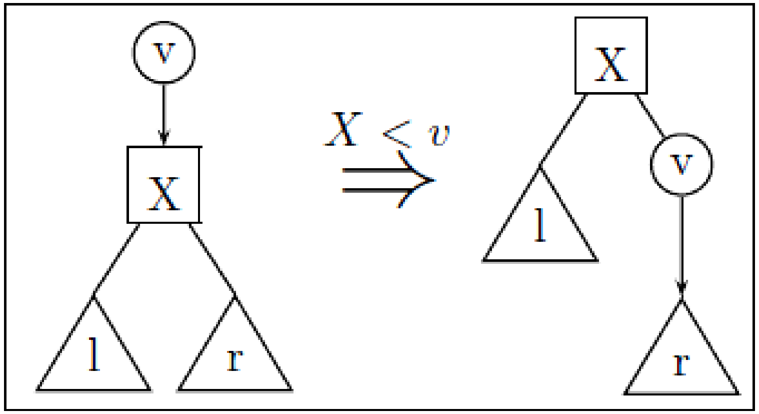
1) Find the leaf where a new item, X, should be inserted and insert it.
2) If the leaf now contains 2 items, you are done.
3) If the leaf now contains 3 items, X, Y, Z, then
   - replace the leaf by two new nodes, n1 and n2, with the smallest of **X**, Y, **Z** going into **n1**, the largest going into **n2**, and the middle value going into the leaf's parent node, p ;
   - make n1 and n2 children of parent **p**.
4) If parent **p** now contains 2 items (and has 3 children) you are done.
5) If parent **p** now contains 3 items (and has 4 children) proceed as in step 3, except that
   - **p**'s two leftmost children are attached to n1, and
   - **p**'s two rightmost children are attached to n2.
6) Repeat steps 3 - 5, until arriving at a parent node containing 2 items.

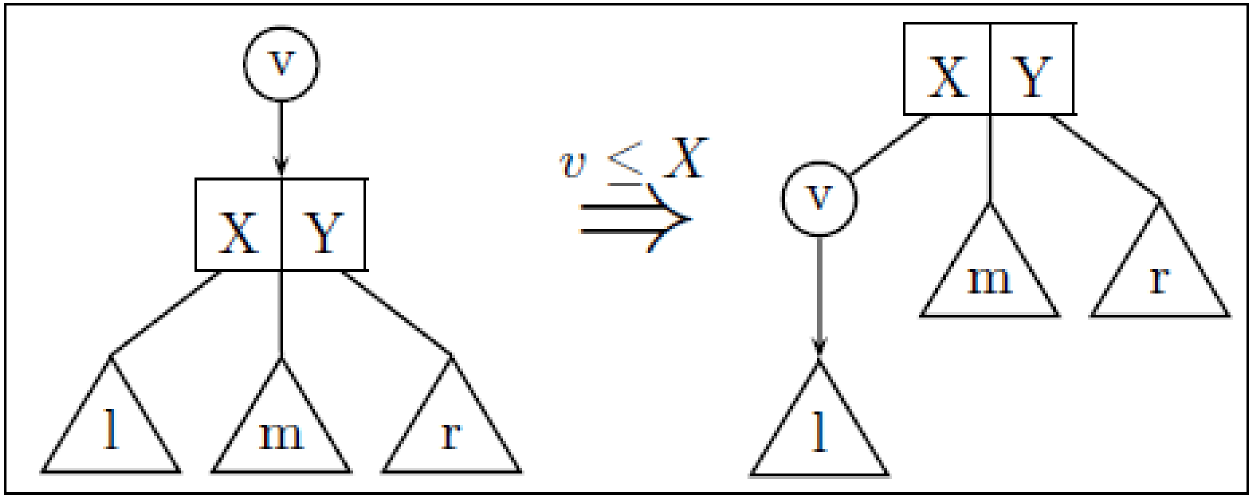
2-3 Tree Insertion:

In the following rules, the result of inserting an element **v** into a 2-3 tree is depicted as a circled **v** with an arrow pointing down toward the tree in which it is to be inserted. **X** and **Y** are variables that stand for any elements, while triangles labeled **l**, **m**, and **r** stand for whole subtrees.

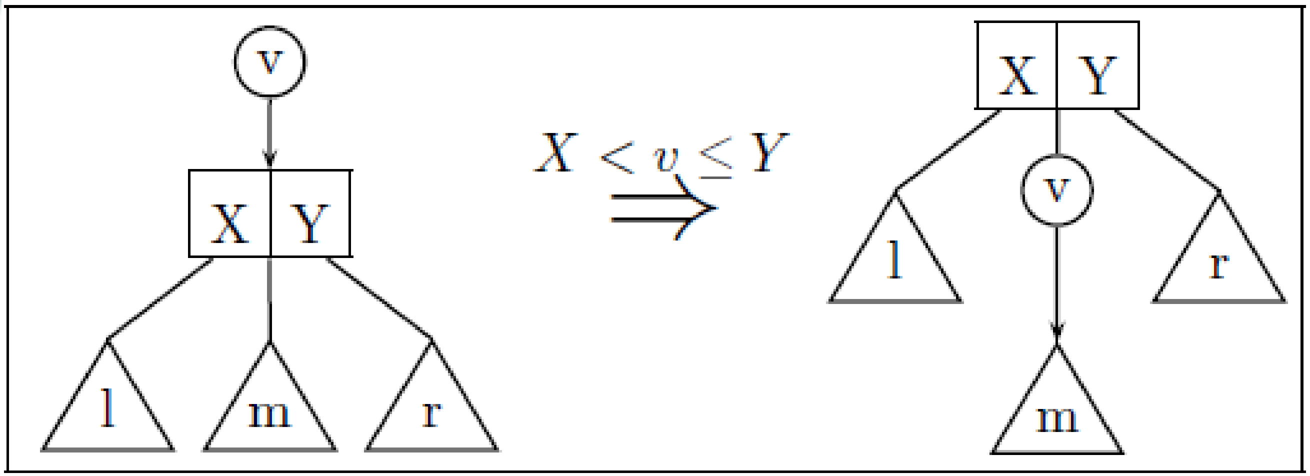


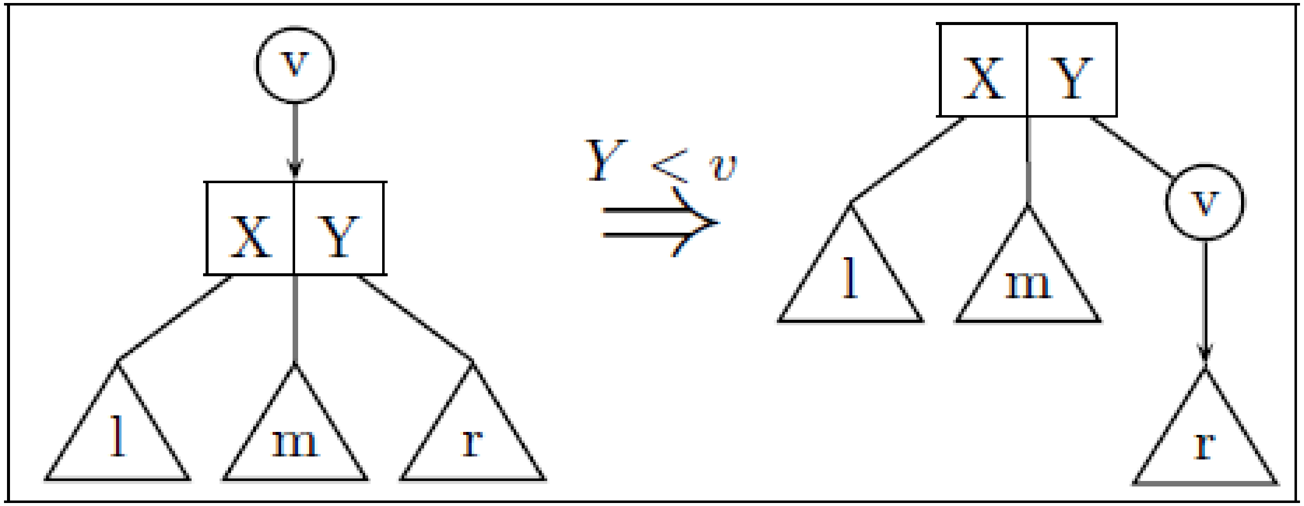
2-3 Tree Insertion:



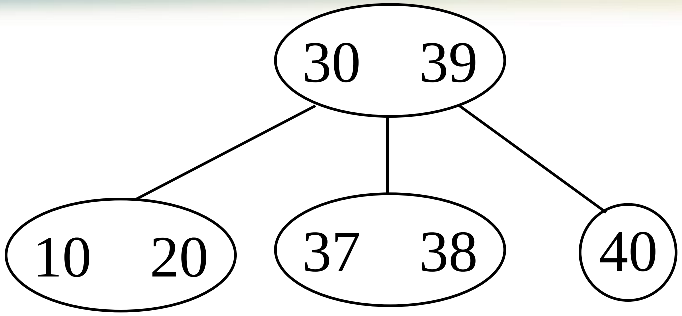


2-3 Tree Insertion:

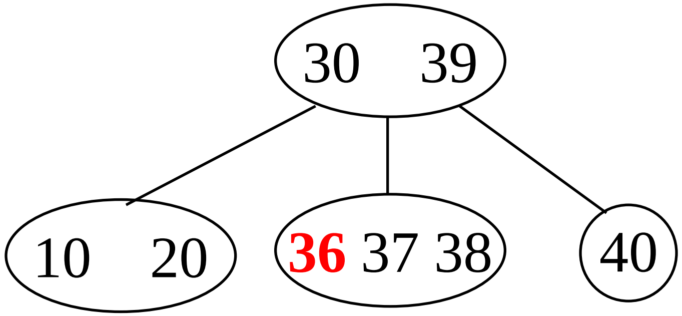




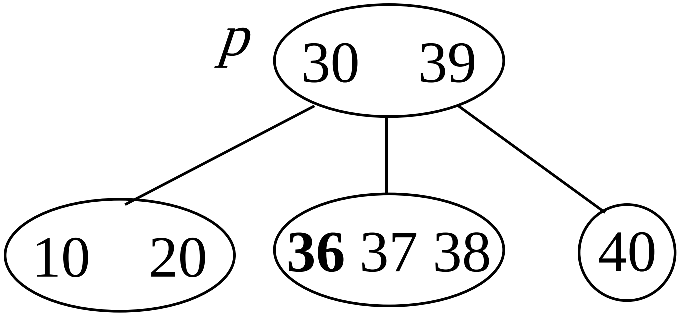
2-3 Tree Insertion: *Example*

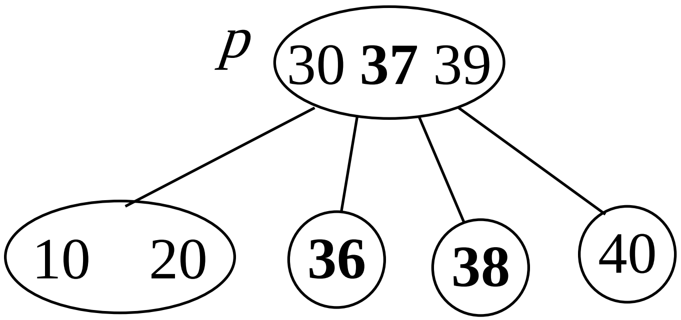


Insert 36



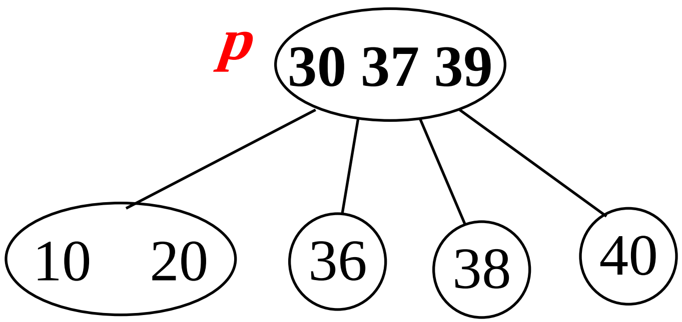
2-3 Tree Insertion: *Example*

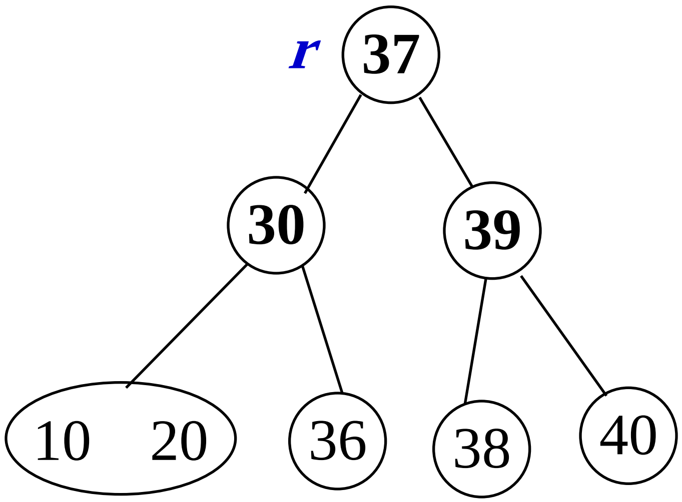




- Since the leaf with 36 in it now contains 3 items, replace the leaf by two new nodes containing 36 (the smallest) and 38 (the largest).
- Move 37 (the middle value) up to its parent, **p**.
- Make nodes containing 36 and 38 children of parent, **p**.

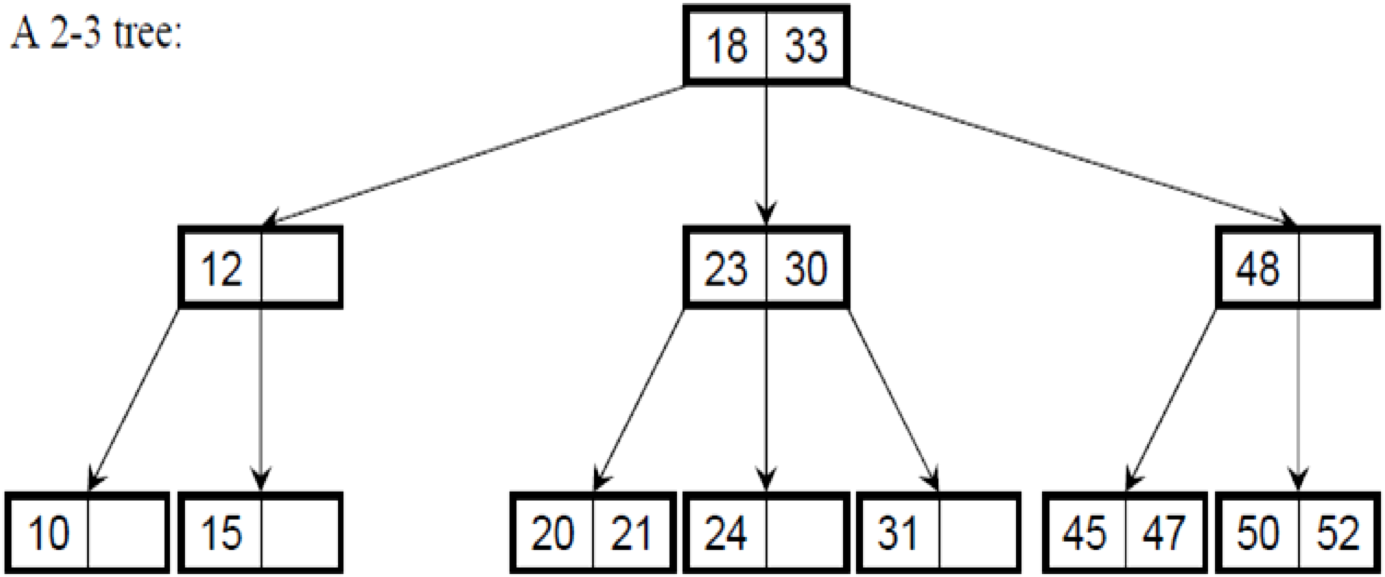
2-3 Tree Insertion: *Example*





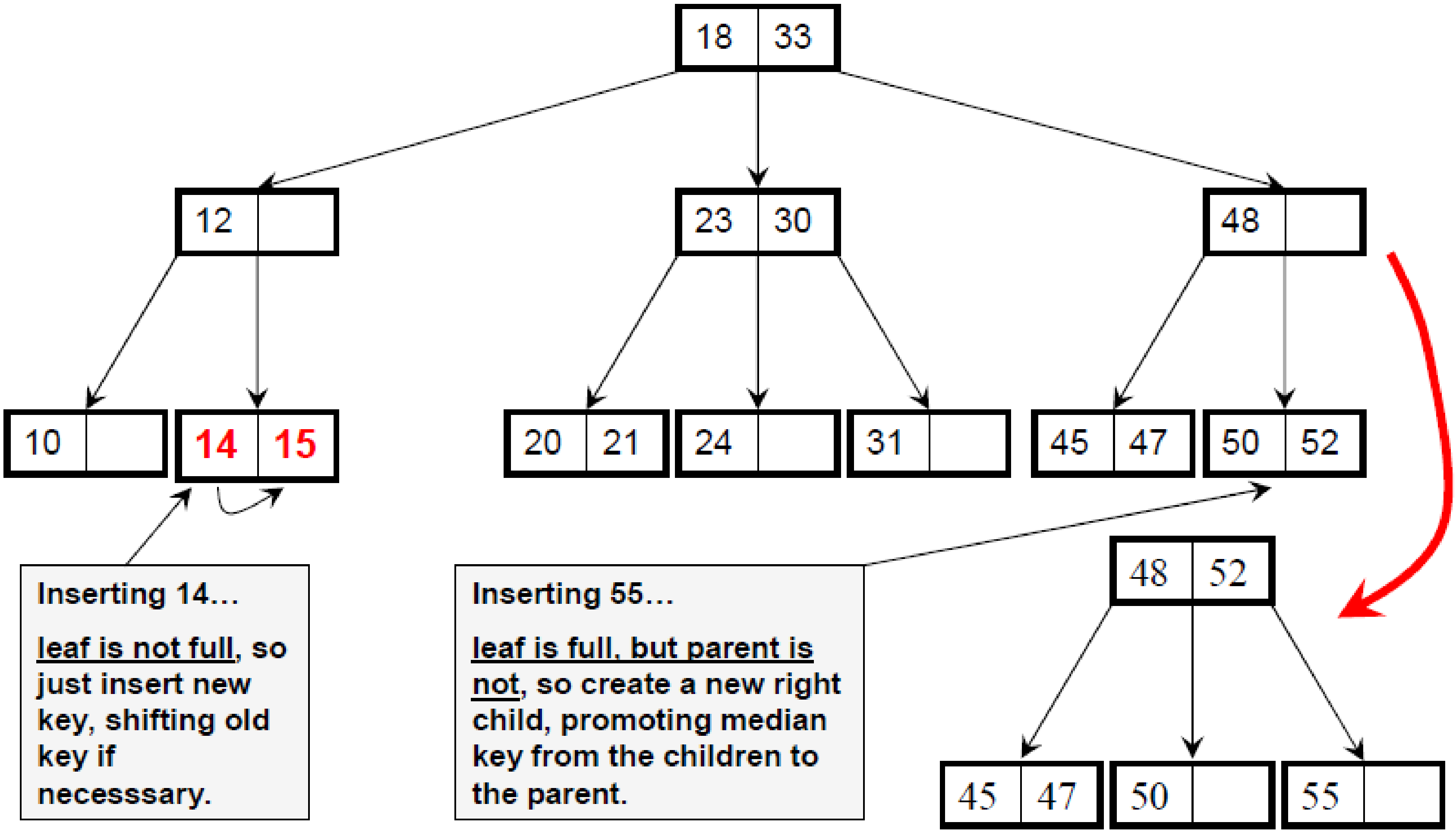
- Since node **p** now contains 3 items, replace **p** by two new nodes containing 30 (the smallest) and 39 (the largest).
- Since **p** has no parent, create a new node, **r**, and move 37 (the middle value) into it.
- Make nodes containing 30 and 39 children of **r**.
- Finally, **p**'s leftmost children are attached to the node containing 30; **p**'s rightmost children are attached to the node containing 39.

2-3 Tree Properties



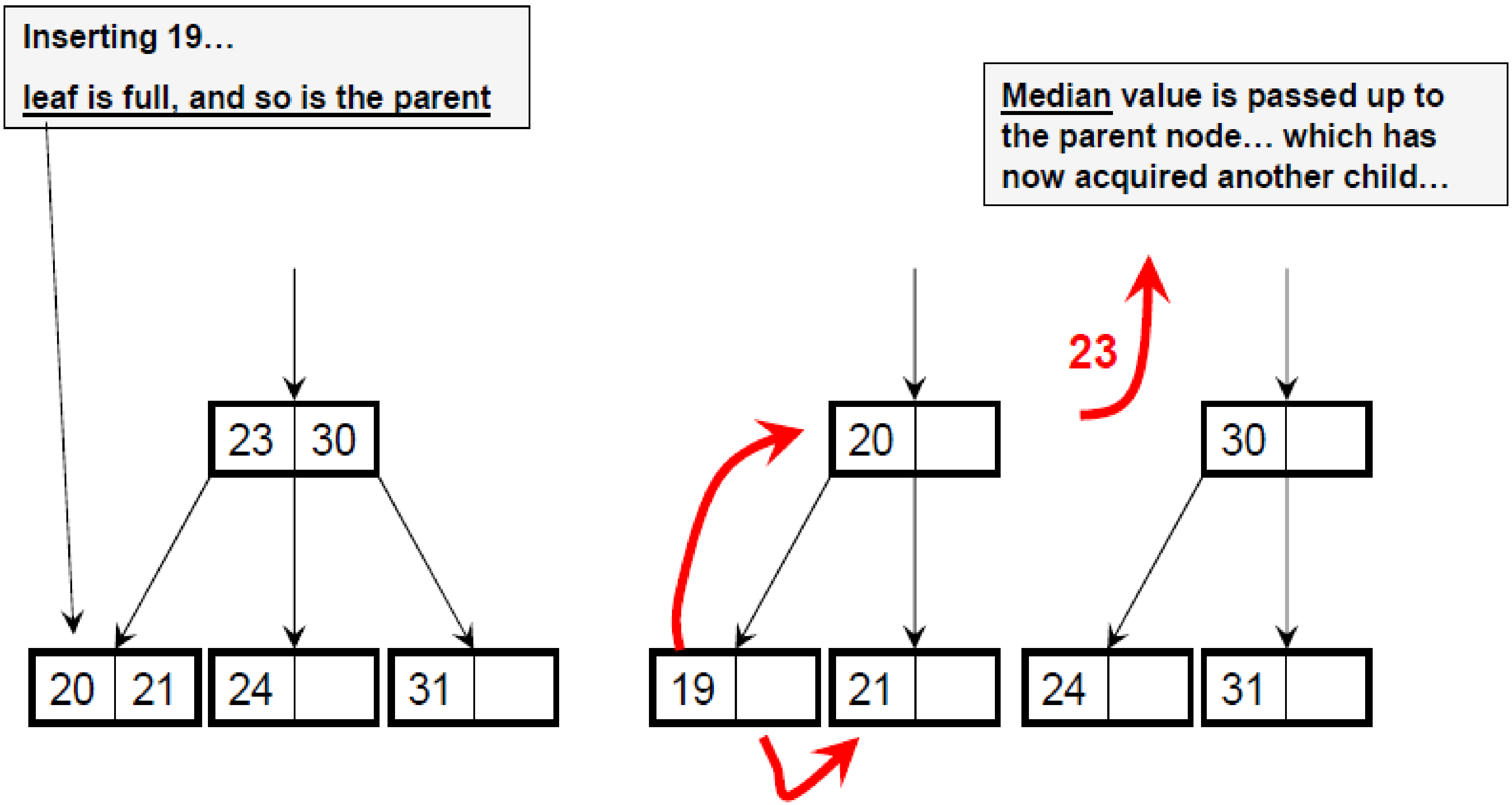
- Each node can store up to two key values and up to three pointers.

2-3 Tree Properties



2-3 Tree Splitting

If a subtree is sufficiently full, insertion may cause the parent to split:



Thank you

???