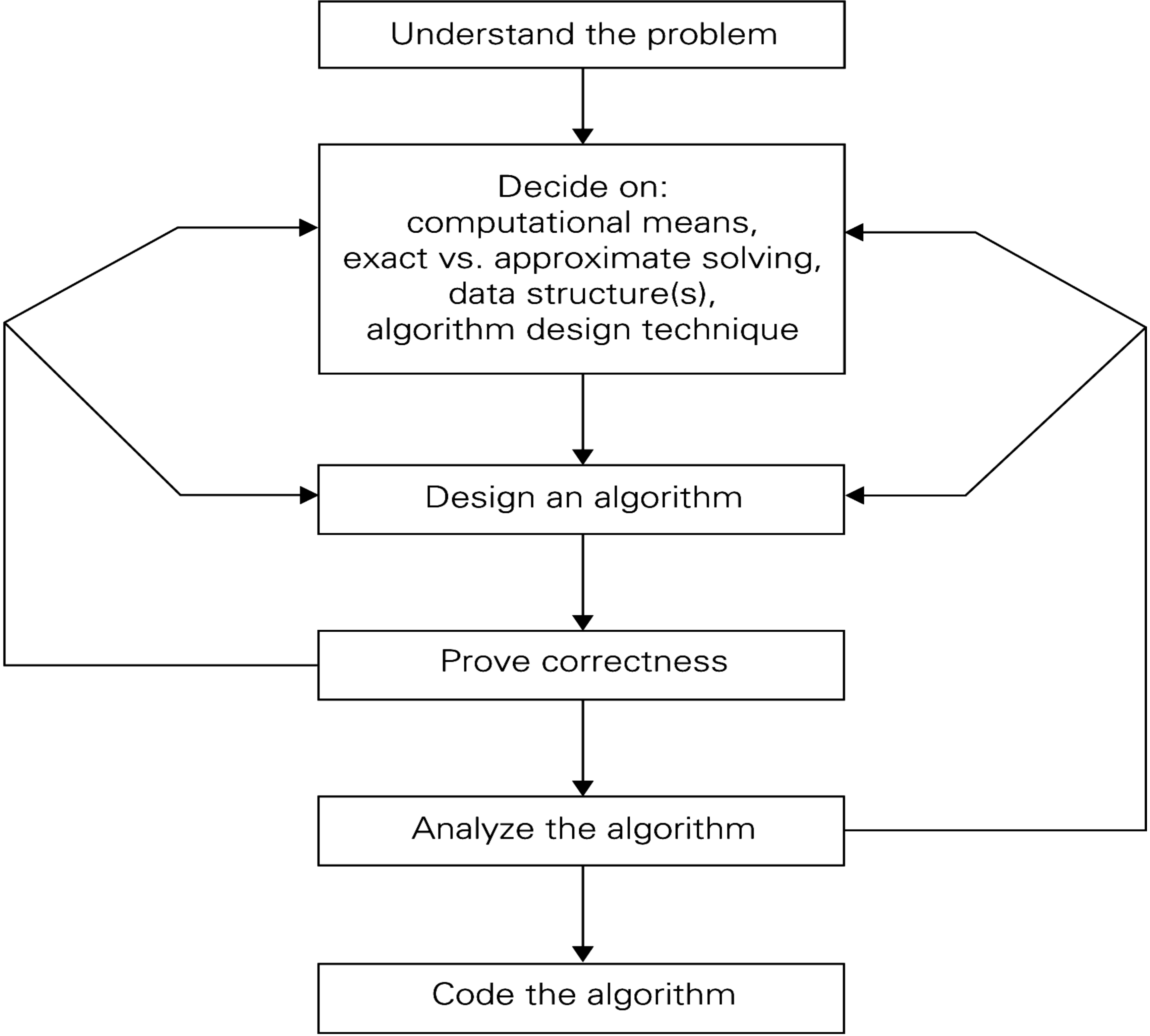
JICSCI803
Algorithms and Data Structures
March to June 2020

Highlights of Lecture 05

1. Binary Trees
2. Binary Search Trees
3. AVL (Adelson-Velski and Landis) trees

Q: For what new data structures are designed?

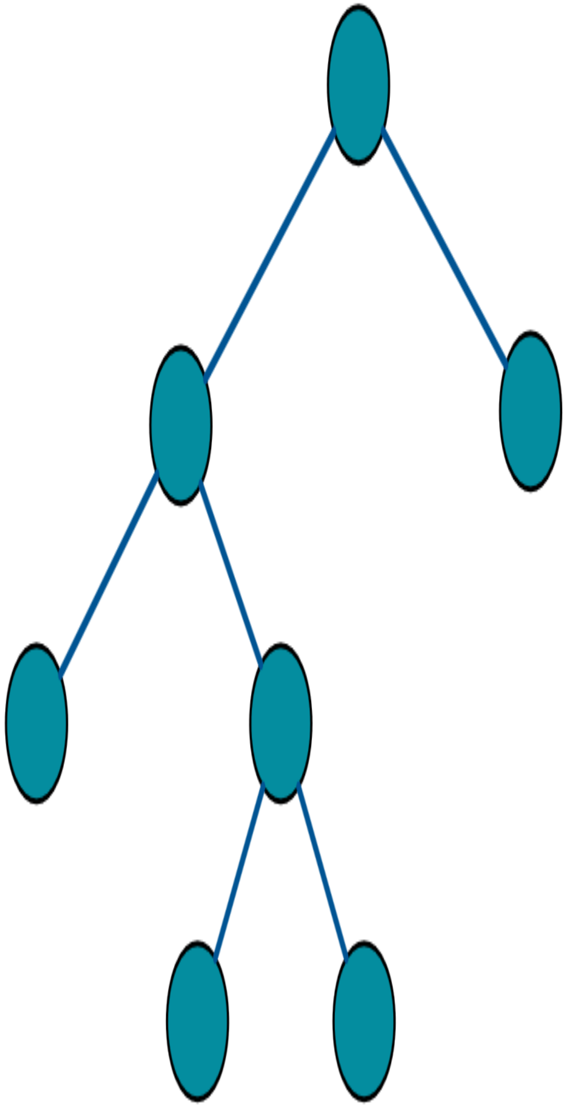
Algorithm Design and Analysis Process

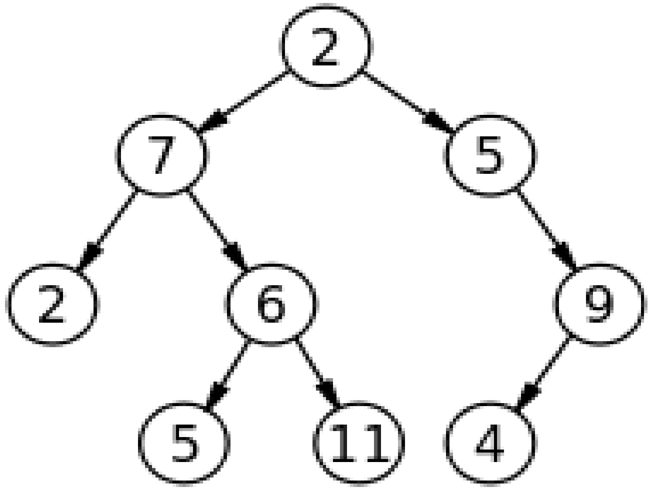


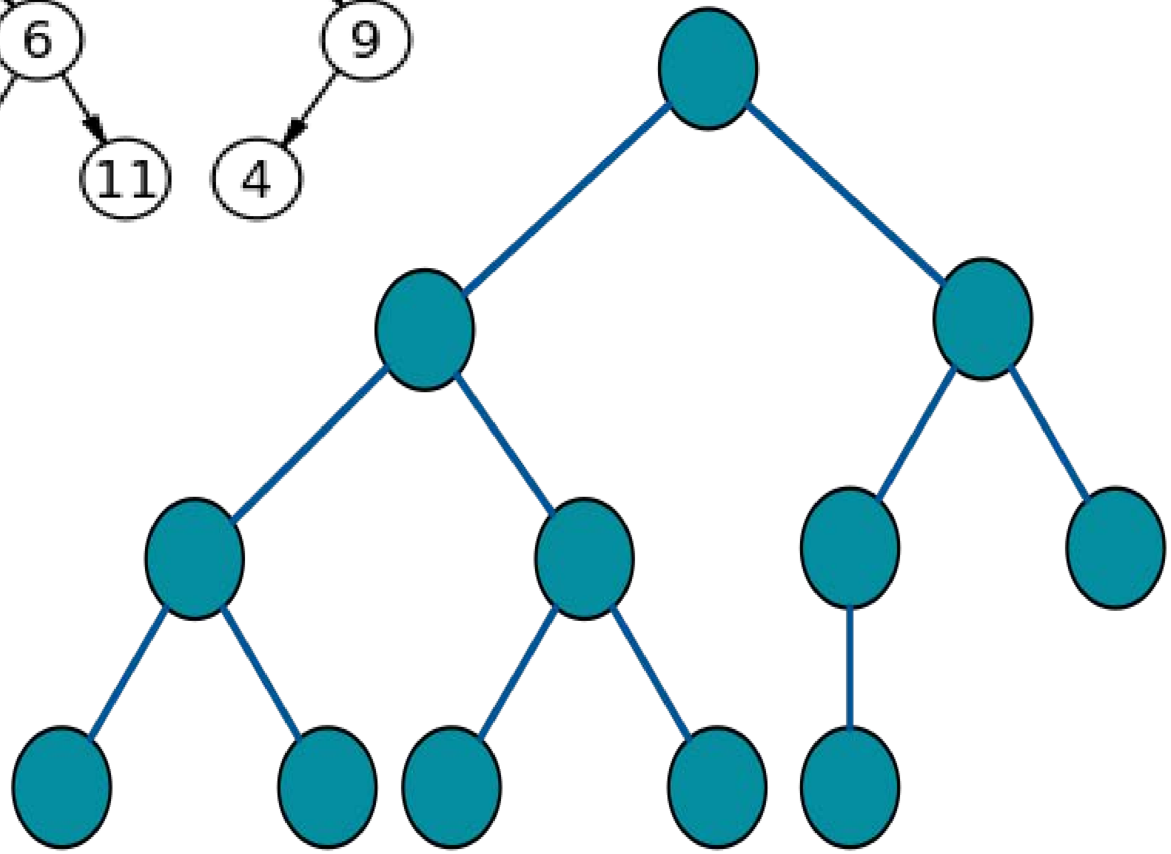
Algorithm Analysis Framework

Measuring an input's size
Measuring running time
Orders of growth (of the algorithm's efficiency function)
Worst-base, best-case and average-case efficiency

Binary Tree



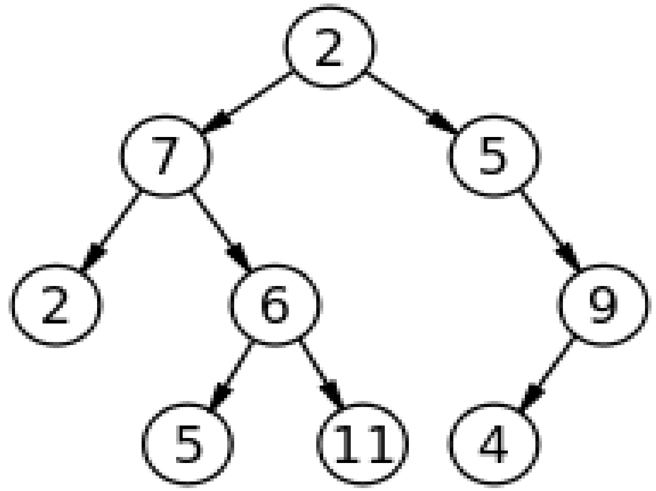




Binary Tree

In computer science, a binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.

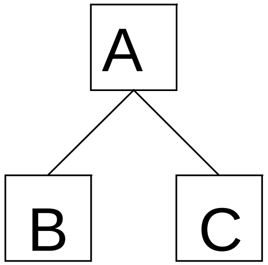
Tree Traversal



Tree Traversal: In computer science, tree traversal (also known as tree search) is a form of graph traversal and refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once.

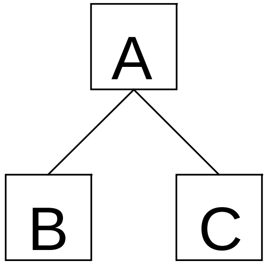
Binary Tree Traversal (three algorithms)

Preorder Traversal
function pretrav(tree)
print tree^.contents
pretrav (tree^.left)
pretrav (tree^.right)



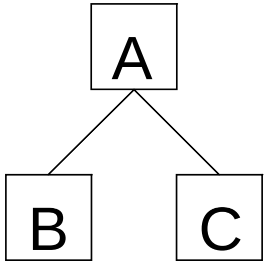
A, B, C

Inorder Traversal
function intrav(tree)
intrav (tree^.left)
print tree^.contents
intrav (tree^.right)



B, A, C

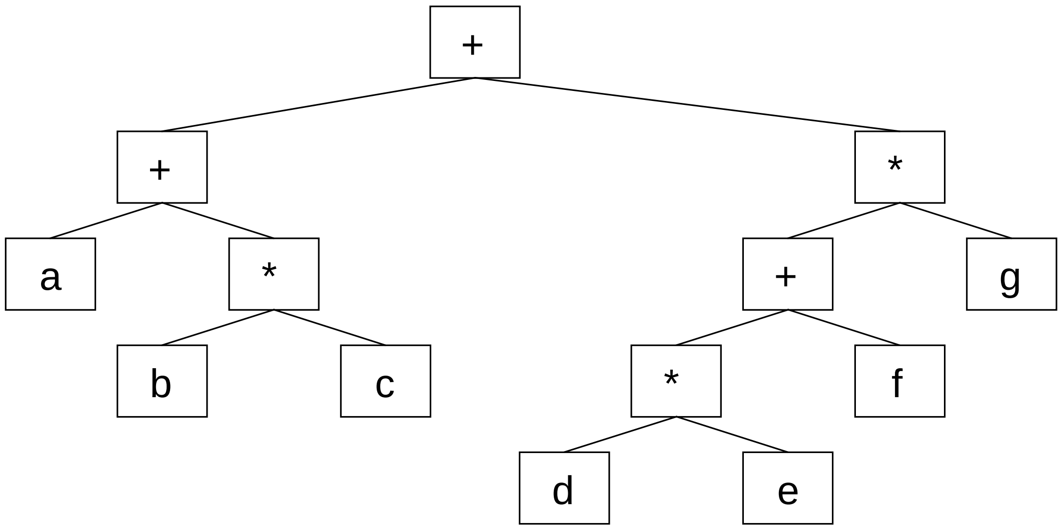
Postorder Traversal
function posttrav(tree)
posttrav (tree^.left)
posttrav (tree^.right)
print tree^.contents



B, C, A

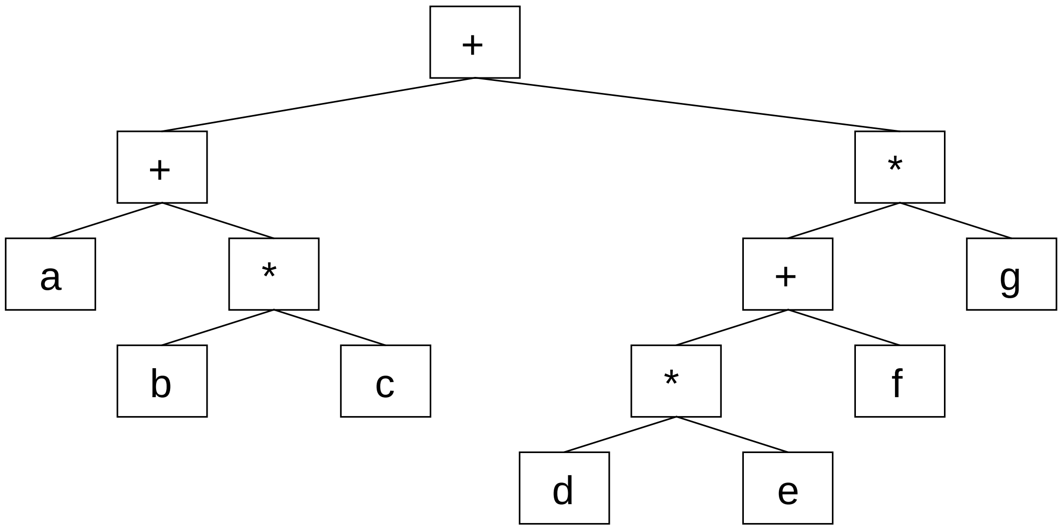
Binary Tree Inorder Traversal

– An Example: Expression Trees



Binary Tree Inorder Traversal

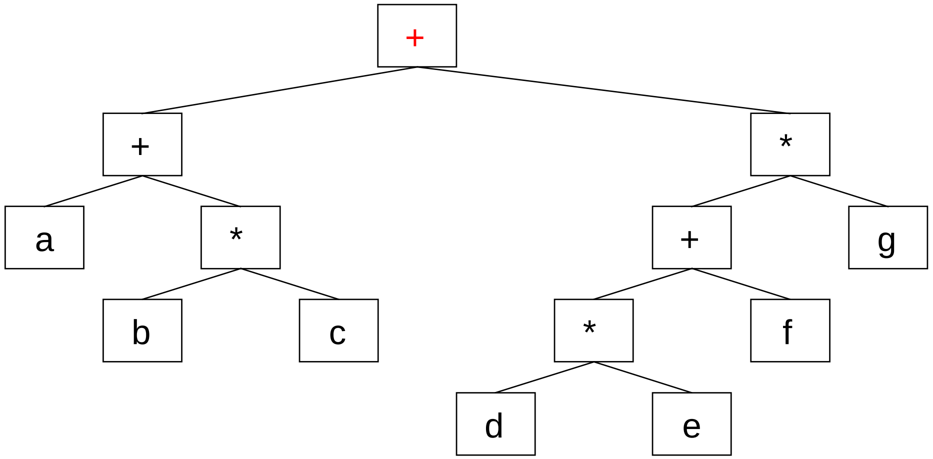
– An Example: Expression Trees



((a + (b * c)) + (((d * e) +f) *g))

Binary Tree Traversal

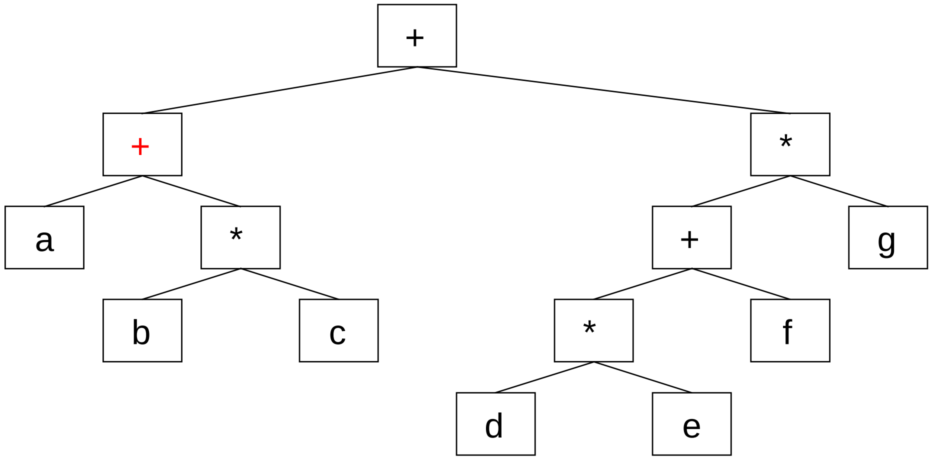
– An Example: Expression Trees



+

Binary Tree Traversal

– An Example : Expression Trees

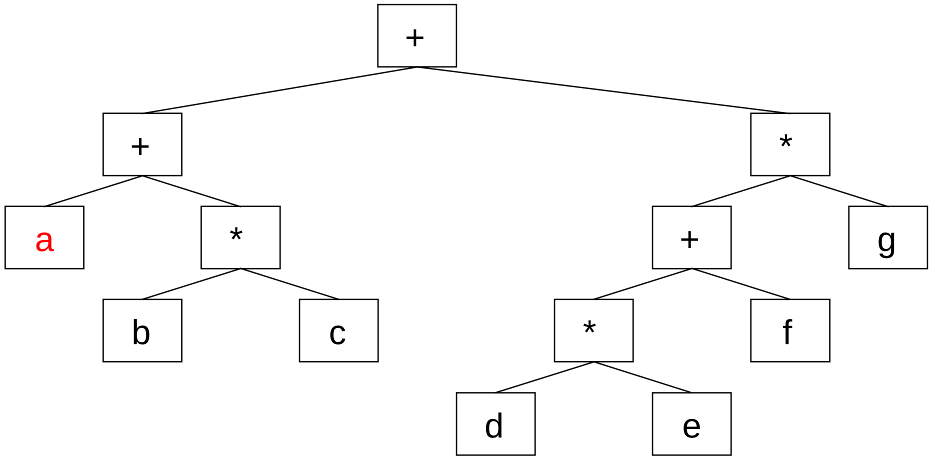


Preorder
inorder
Postorder

+ +

Binary Tree Traversal

– An Example: Expression Trees

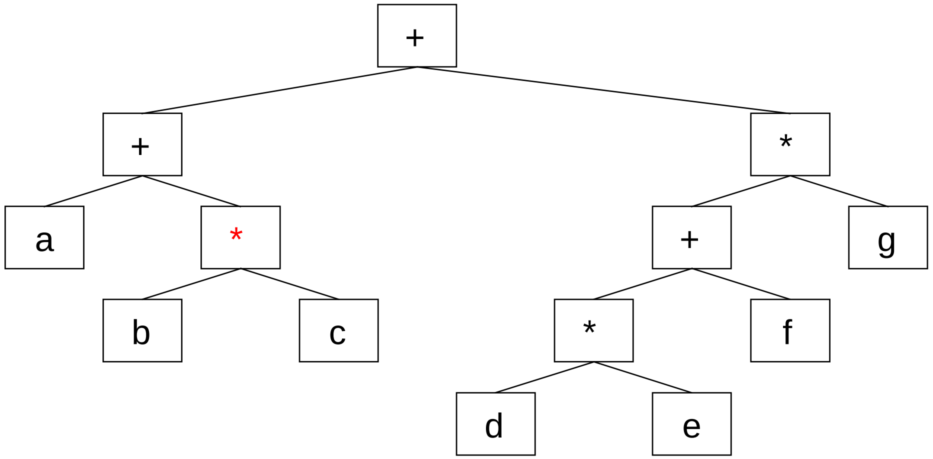


Preorder
inorder
Postorder

+ + a

Binary Tree Traversal

– An Example: Expression Trees

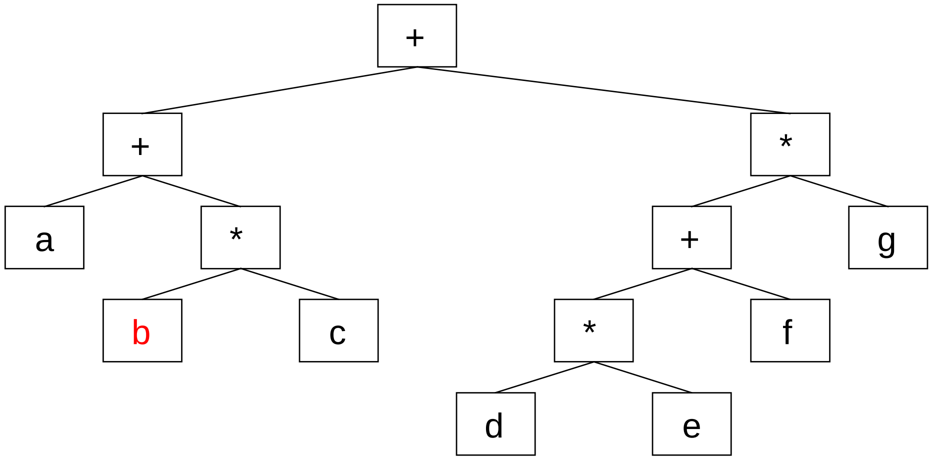


Preorder
inorder
Postorder

+ + a *

Binary Tree Traversal

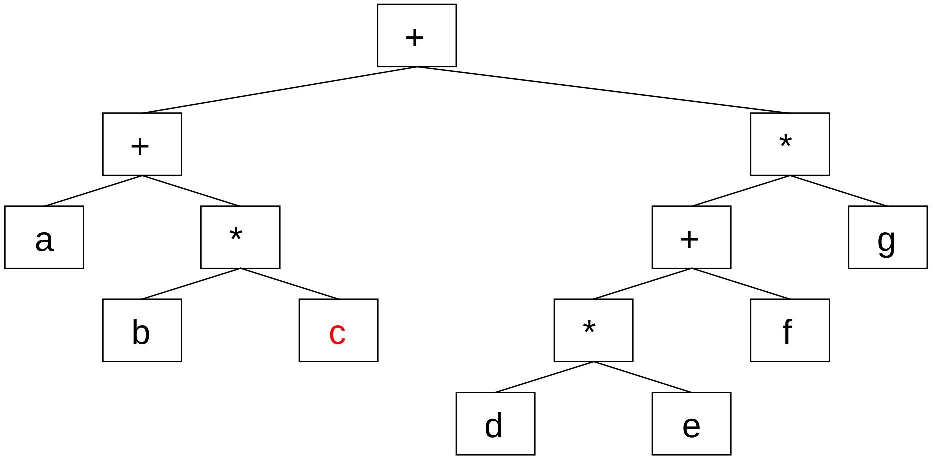
– An Example: Expression Trees



+ + a * b

Binary Tree Traversal

– An Example: Expression Trees

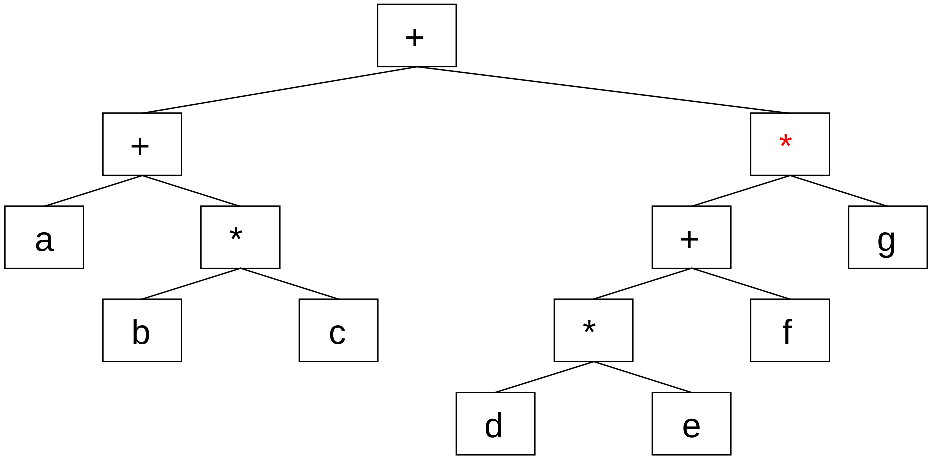


Preorder
inorder
Postorder

+ + a * b c

Binary Tree Traversal

– An Example: Expression Trees

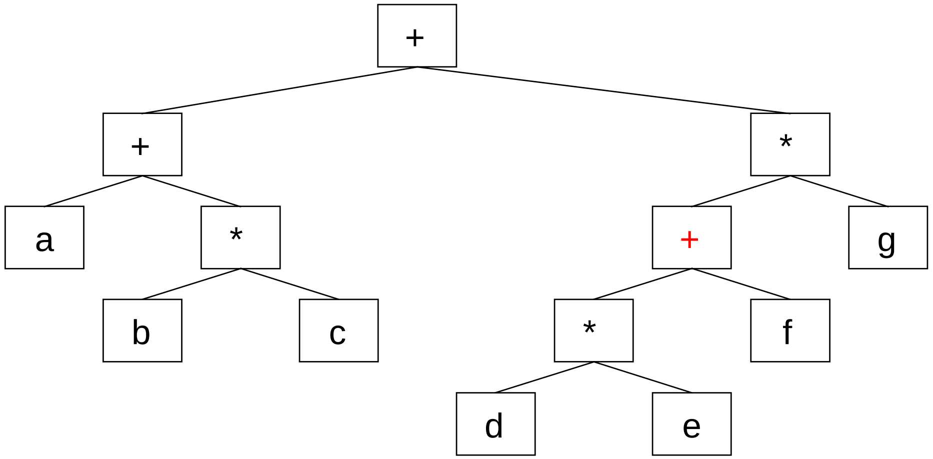


Preorder
inorder
Postorder

+ + a * b c *

Binary Tree Traversal

– An Example: Expression Trees

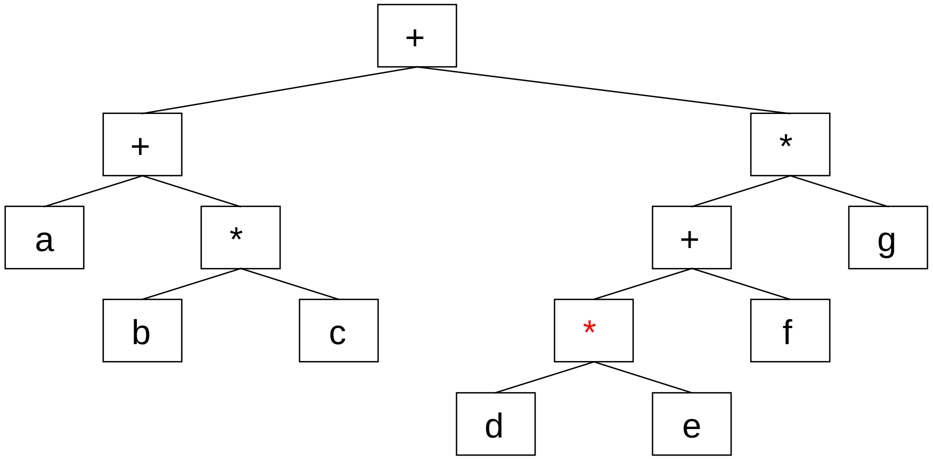


Preorder
inorder
Postorder

+ + a * b c * +

Binary Tree Traversal

– An Example: Expression Trees

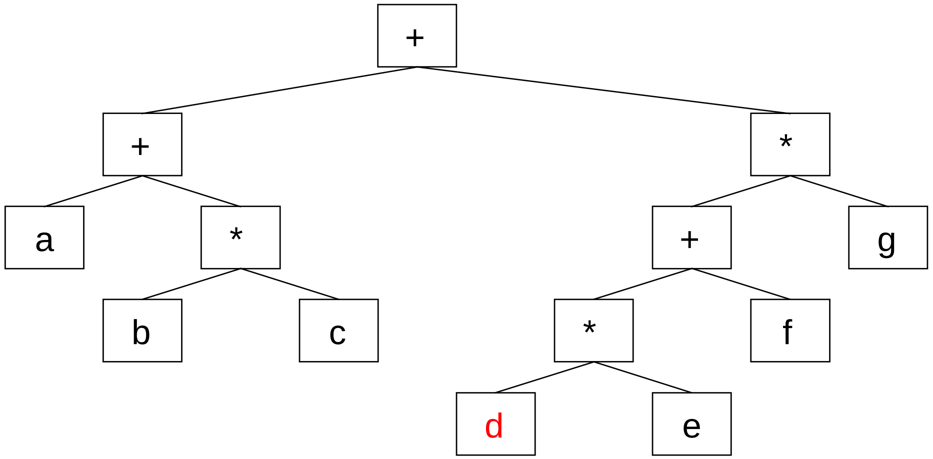


Preorder
inorder
Postorder

+ + a * b c * + *

Binary Tree Traversal

– An Example: Expression Trees

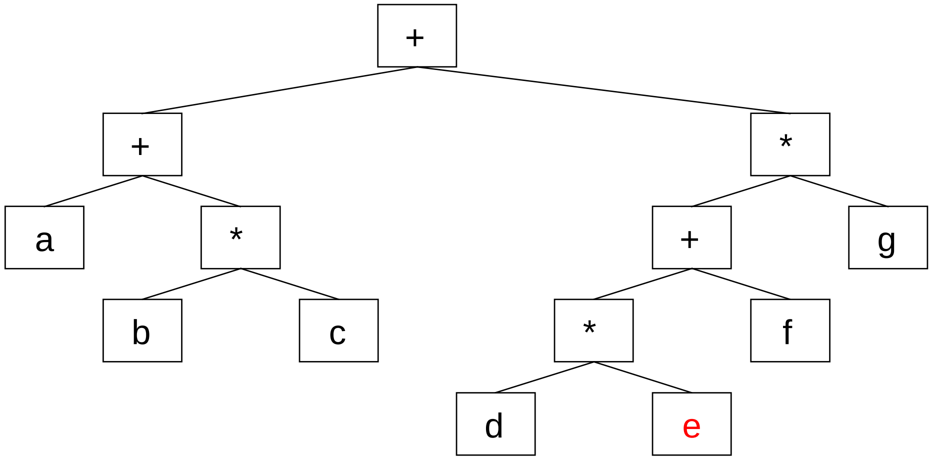


Preorder
inorder
Postorder

+ + a * b c * + * d

Binary Tree Traversal

– An Example: Expression Trees

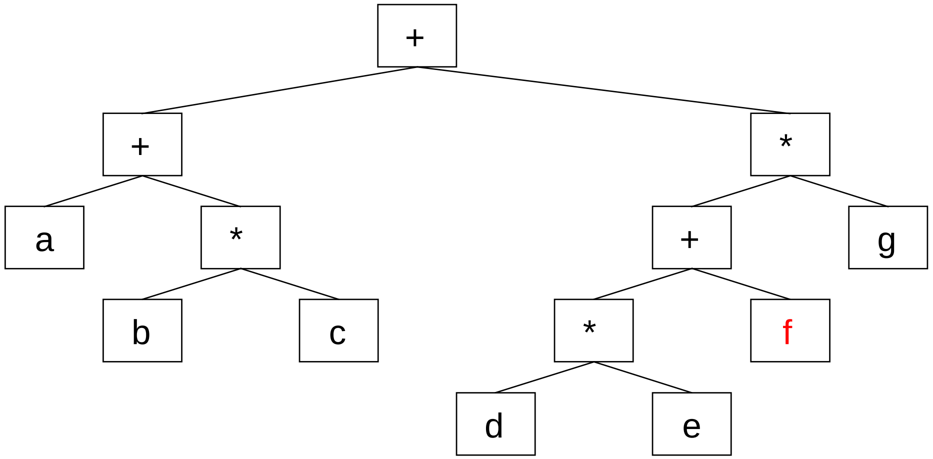


Preorder
inorder
Postorder

+ + a * b c * + * d e

Binary Tree Traversal

– An Example: Expression Trees

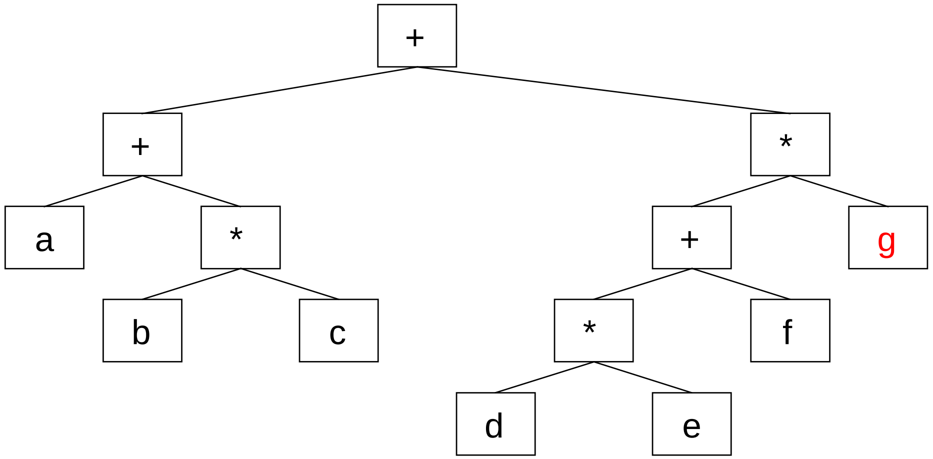


Preorder
inorder
Postorder

\+ + a * b c * + * d e f

Binary Tree Traversal

– An Example: Expression Trees

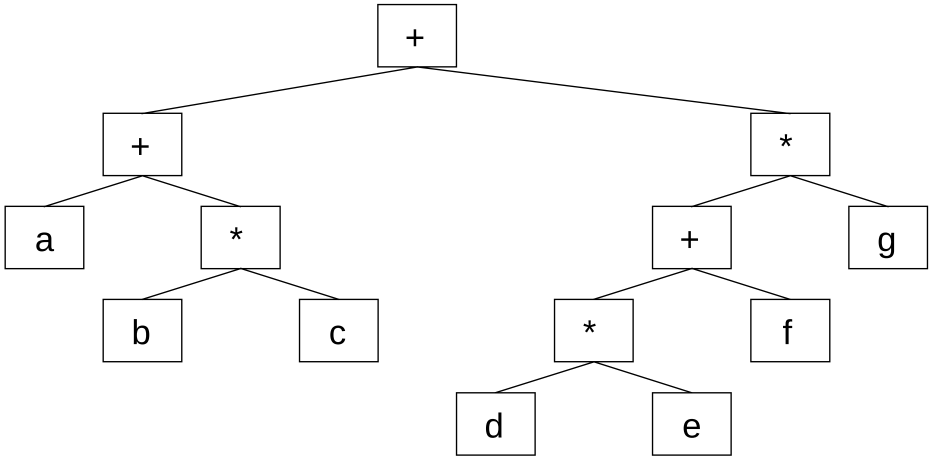


Preorder
inorder
Postorder

+ + a * b c * + * d e f g

Binary Tree Traversal

– An Example: Expression Trees

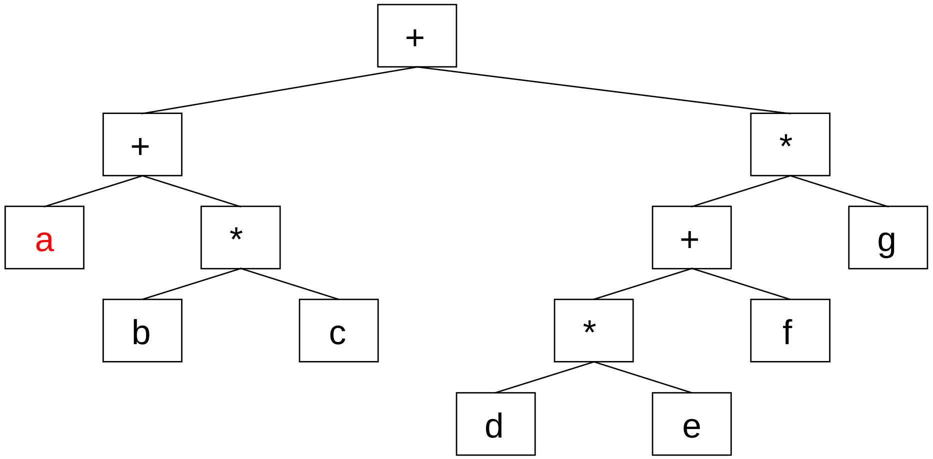


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)

Binary Tree Traversal

– An Example: Expression Trees



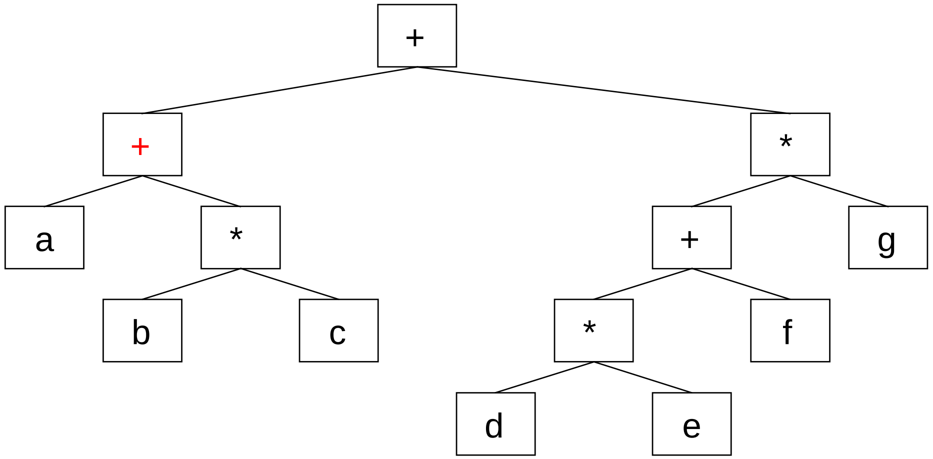
Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)

a

Binary Tree Traversal

– An Example: Expression Trees



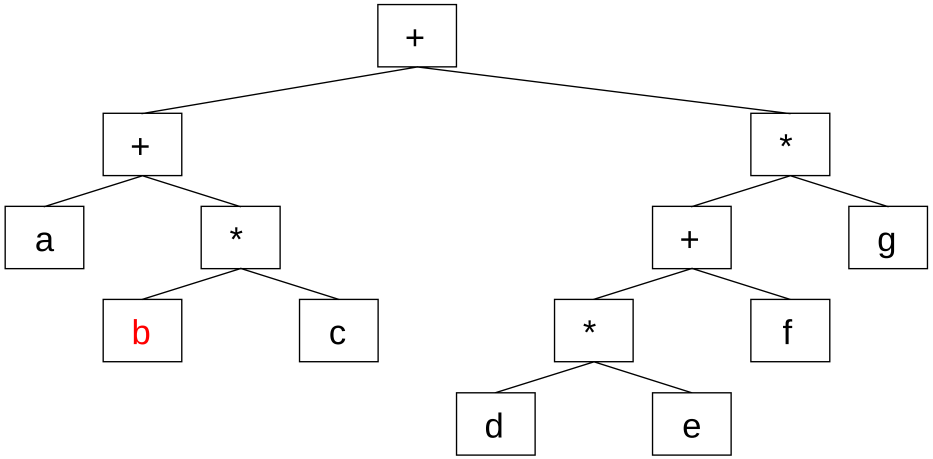
Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)

a +

Binary Tree Traversal

– An Example: Expression Trees



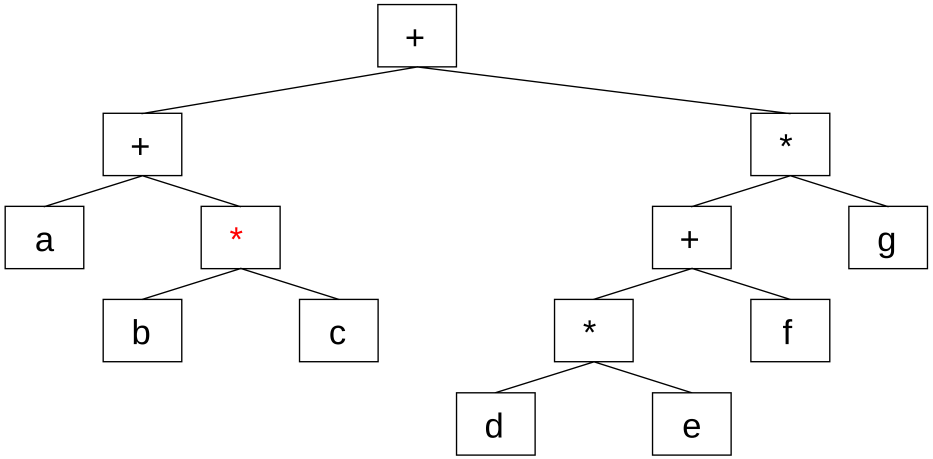
Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)

a + b

Binary Tree Traversal

– An Example: Expression Trees

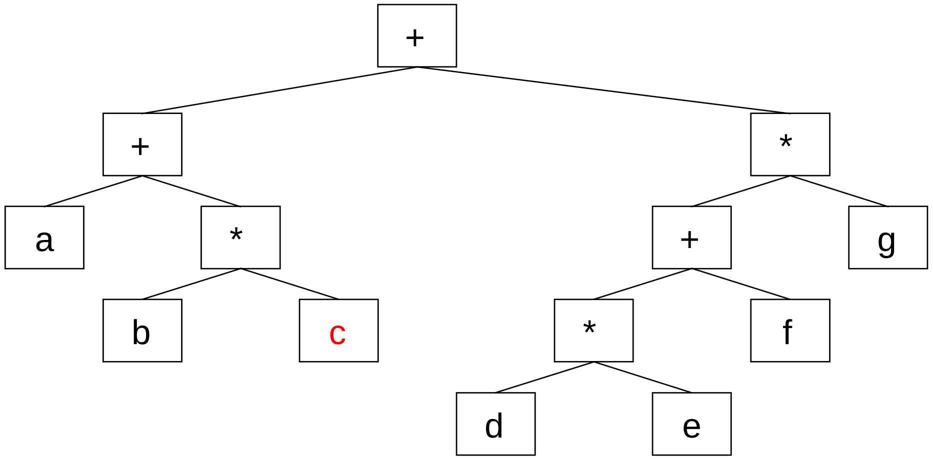


+ + a * b c * + * d e f g (preorder)

a + b *

Binary Tree Traversal

– An Example: Expression Trees

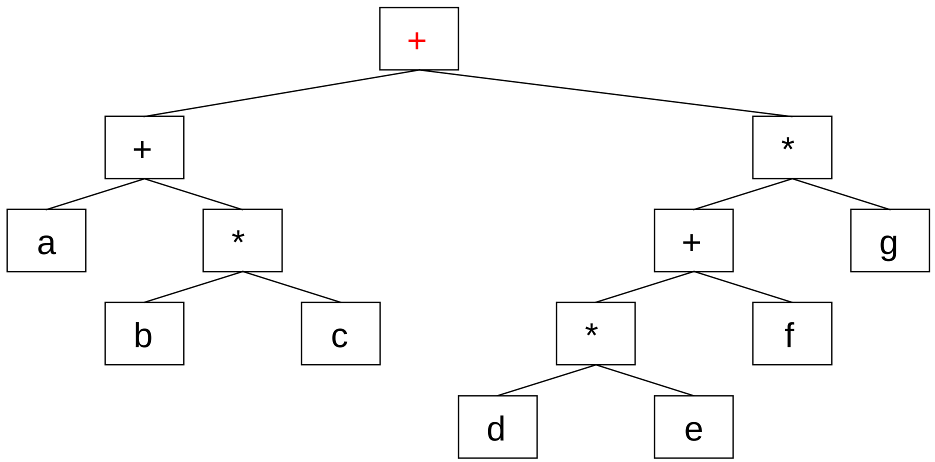


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c

Binary Tree Traversal

– An Example: Expression Trees



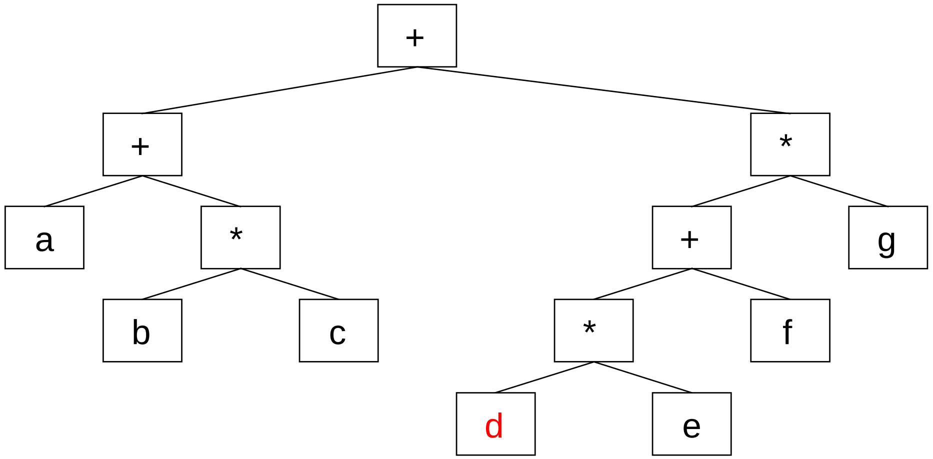
Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)

a + b * c +

Binary Tree Traversal

– An Example: Expression Trees

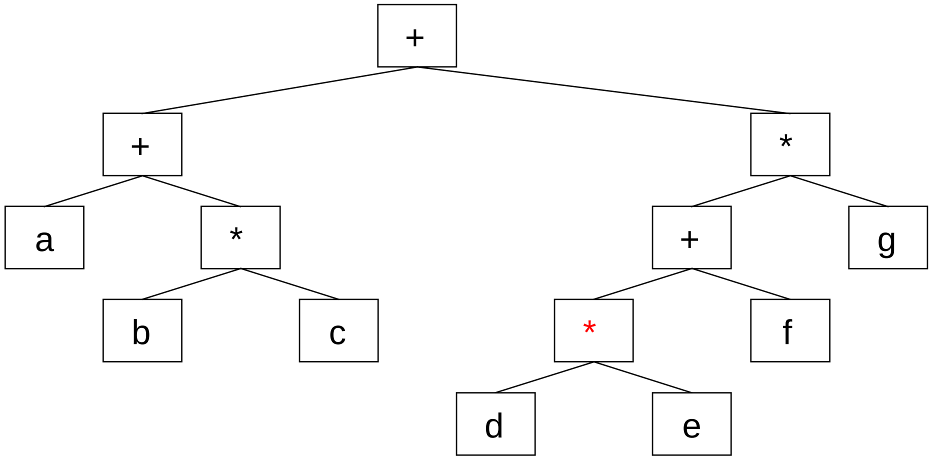


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c + d

Binary Tree Traversal

– An Example: Expression Trees



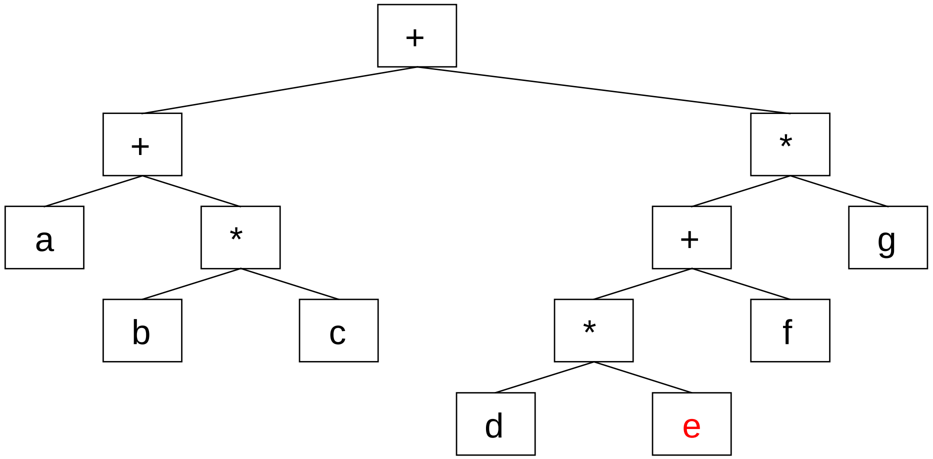
Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)

a + b * c + d *

Binary Tree Traversal

– An Example: Expression Trees



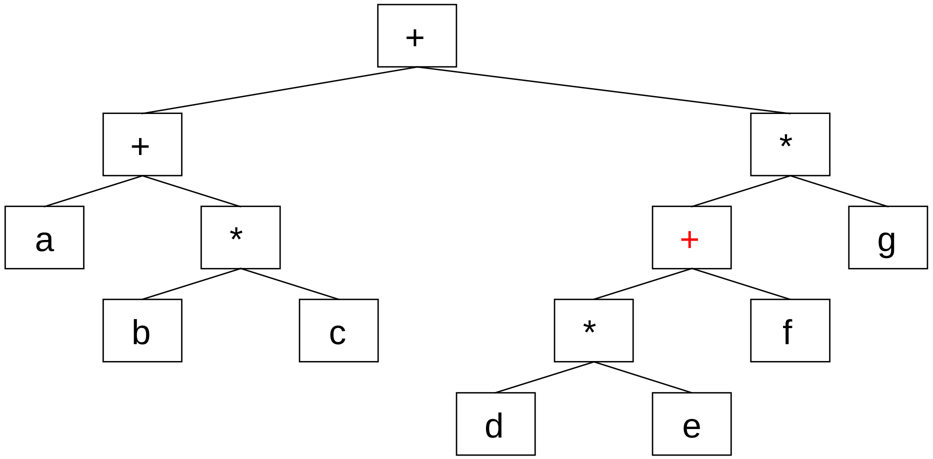
Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)

a + b * c + d * e

Binary Tree Traversal

– An Example: Expression Trees



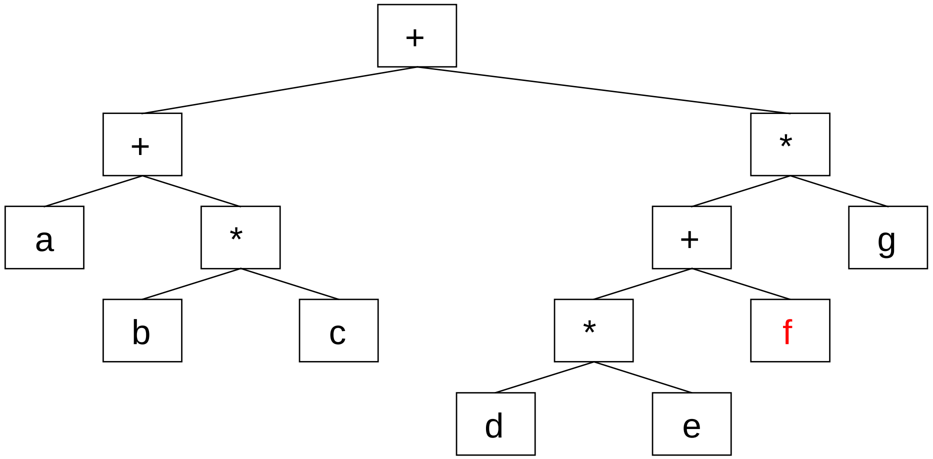
Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)

a + b * c + d * e +

Binary Tree Traversal

– An Example: Expression Trees

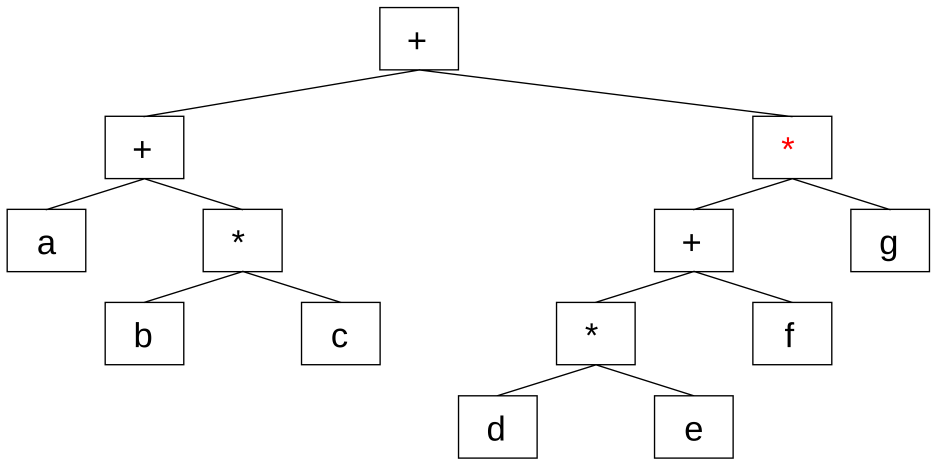


+ + a * b c * + * d e f g (preorder)

a + b * c + d * e + f

Binary Tree Traversal

– An Example: Expression Trees

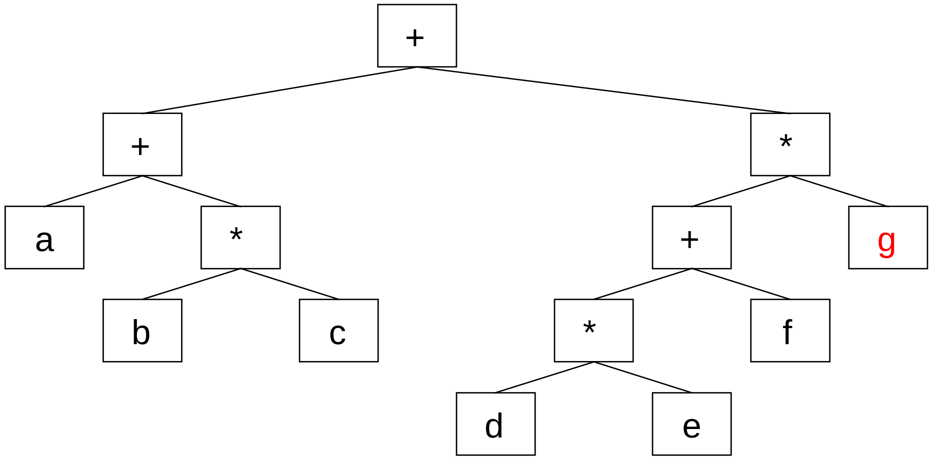


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c + d * e + f *

Binary Tree Traversal

– An Example: Expression Trees

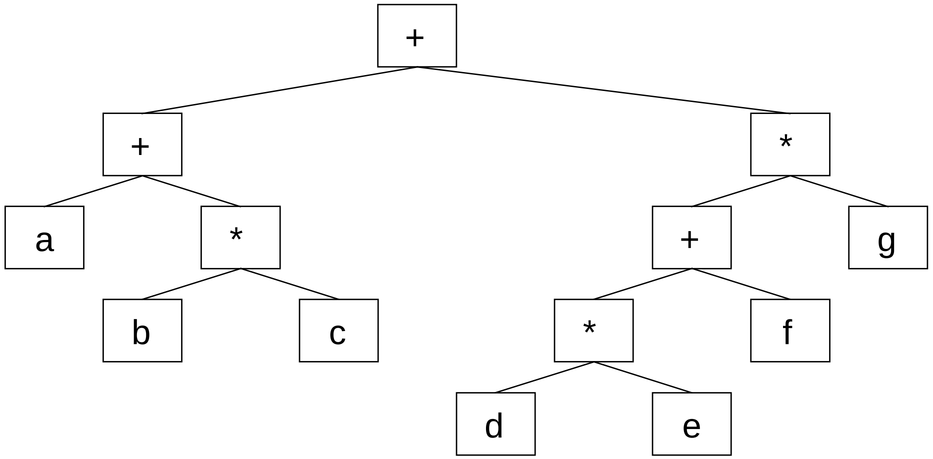


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c + d * e + f * g

Binary Tree Traversal

– An Example: Expression Trees

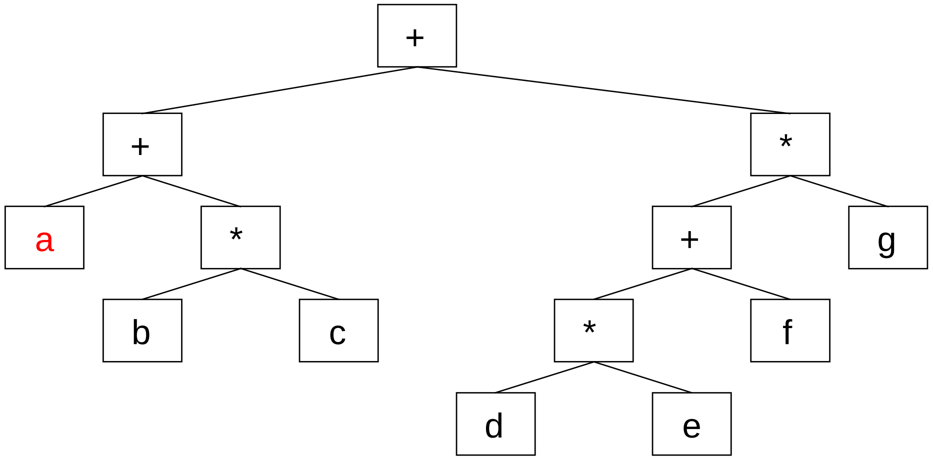


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c + d * e + f * g (inorder)

Binary Tree Traversal

– An Example: Expression Trees

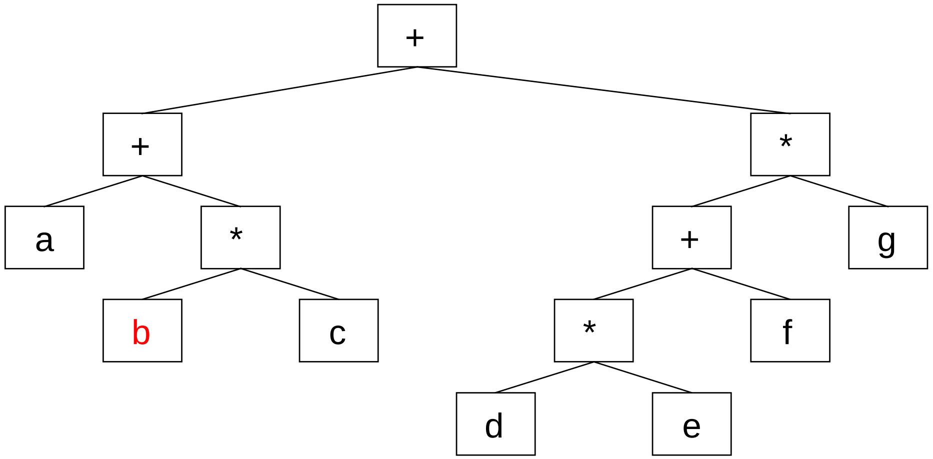


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c + d * e + f * g (inorder)
a

Binary Tree Traversal

– An Example: Expression Trees

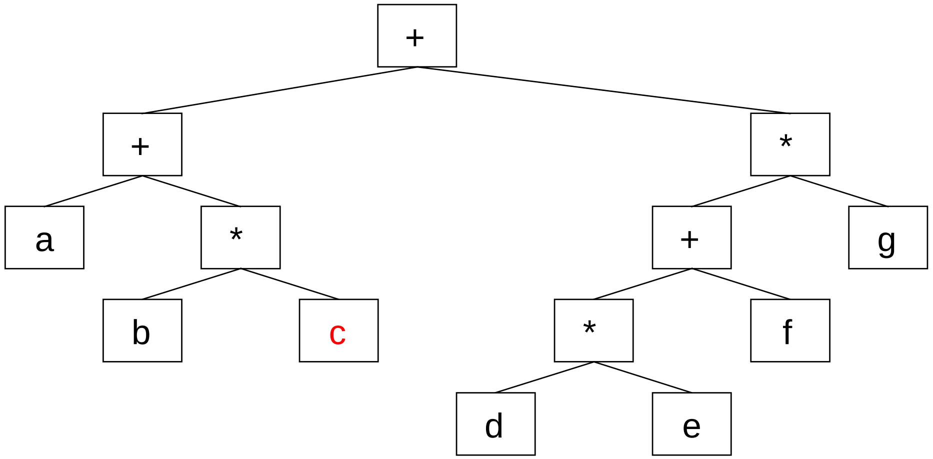


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c + d * e + f * g (inorder)
a b

Binary Tree Traversal

– An Example: Expression Trees

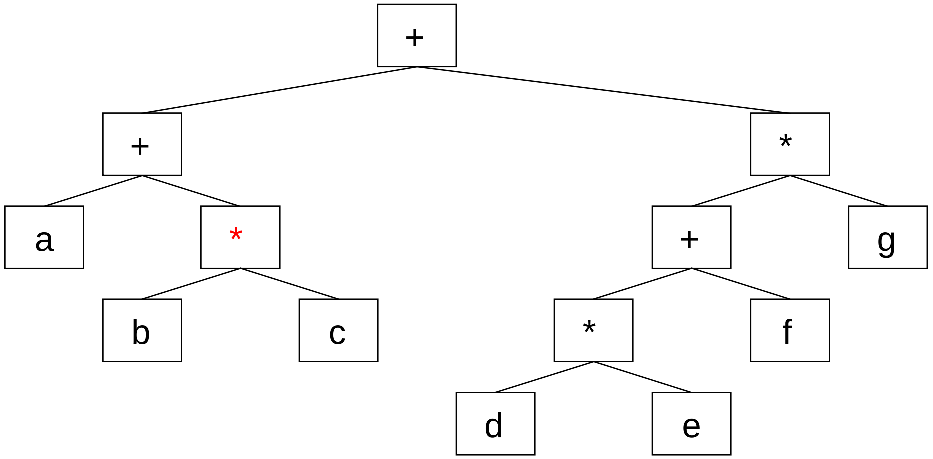


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c + d * e + f * g (inorder)
a b c

Binary Tree Traversal

– An Example: Expression Trees

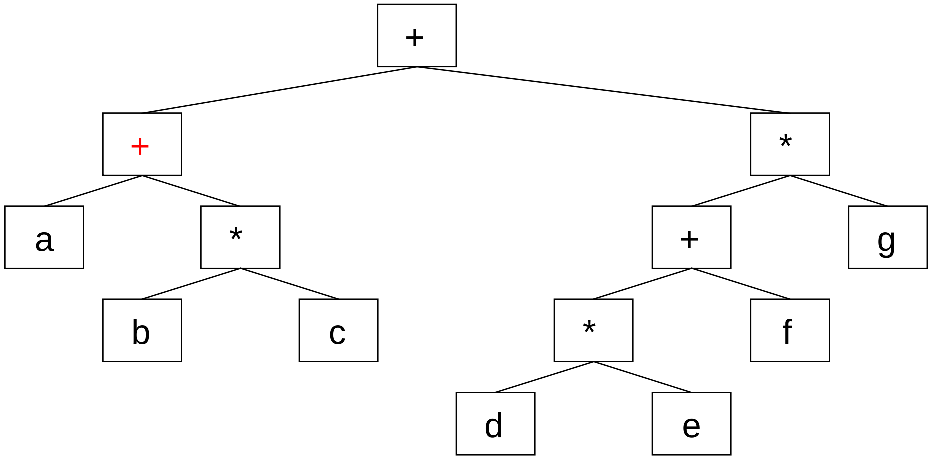


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c + d * e + f * g (inorder)
a b c *

Binary Tree Traversal

– An Example: Expression Trees

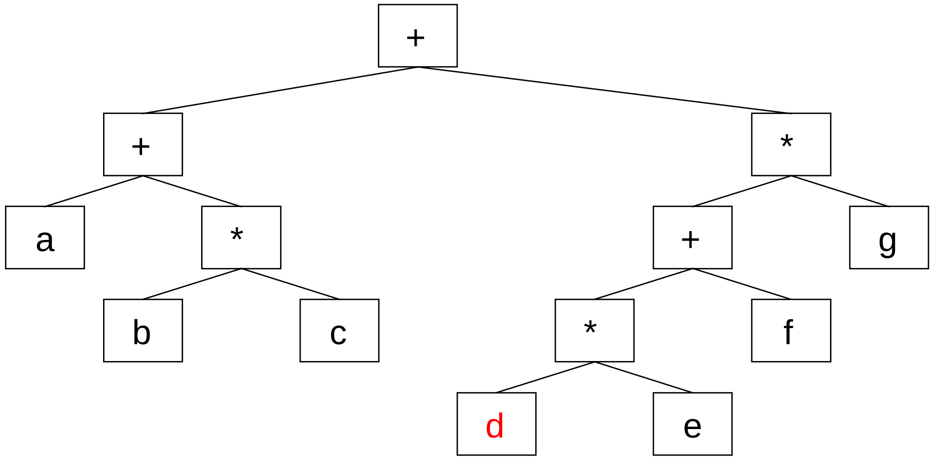


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c + d * e + f * g (inorder)
a b c * +

Binary Tree Traversal

– An Example: Expression Trees

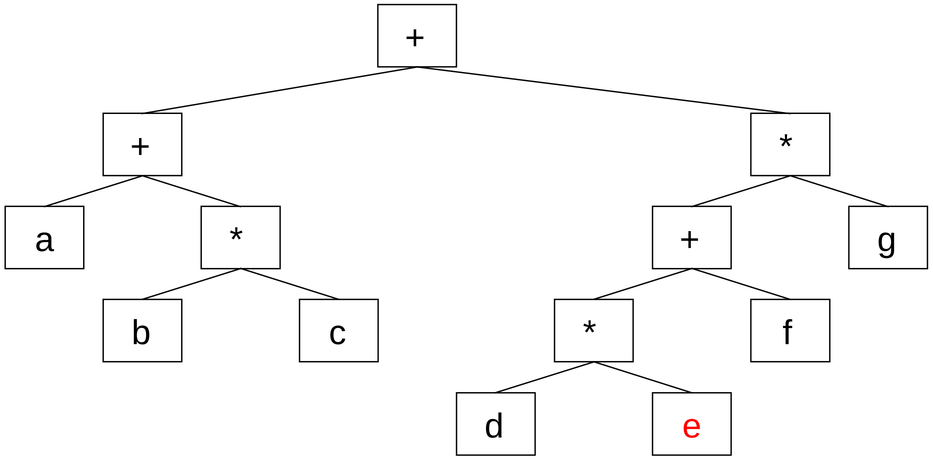


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c + d * e + f * g (inorder)
a b c * + d

Binary Tree Traversal

– An Example: Expression Trees

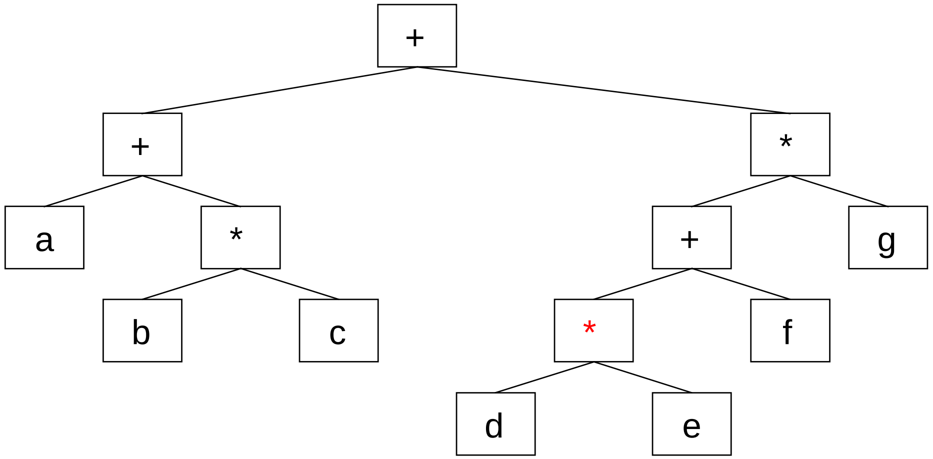


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c + d * e + f * g (inorder)
a b c * + d e

Binary Tree Traversal

– An Example: Expression Trees

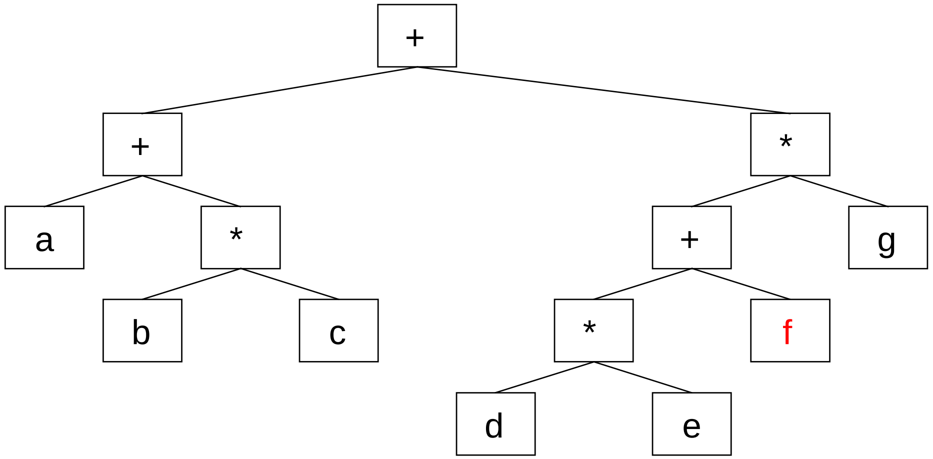


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c + d * e + f * g (inorder)
a b c * + d e *

Binary Tree Traversal

– An Example: Expression Trees

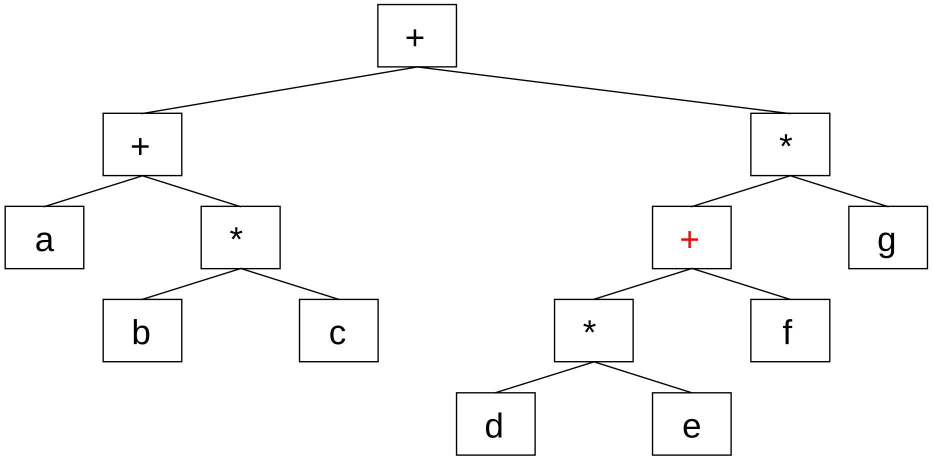


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c + d * e + f * g (inorder)
a b c * + d e * f

Binary Tree Traversal

– An Example: Expression Trees

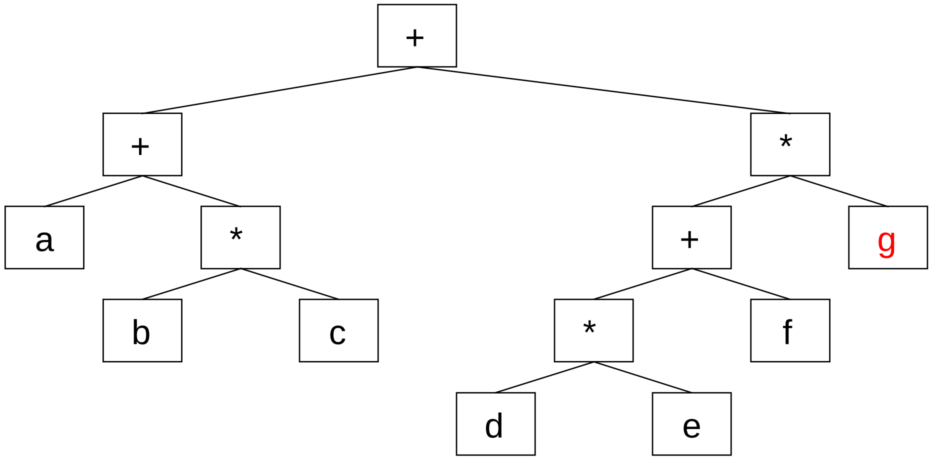


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c + d * e + f * g (inorder)
a b c * + d e * f +

Binary Tree Traversal

– An Example: Expression Trees

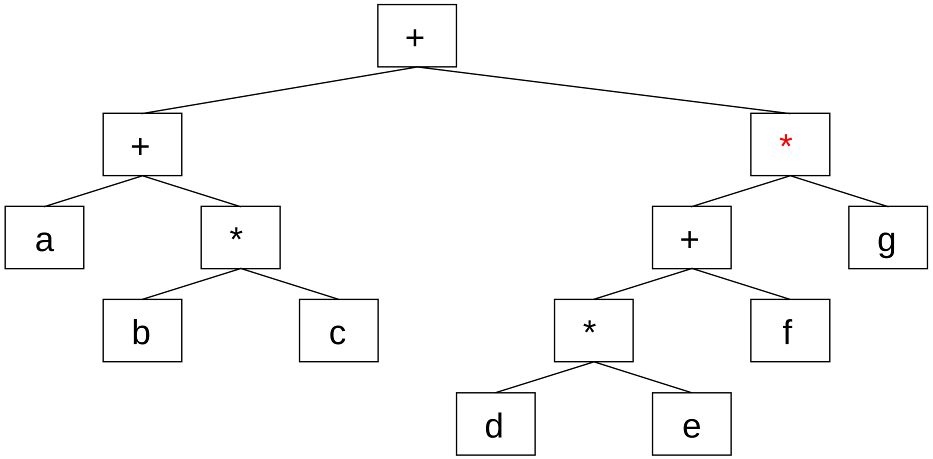


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c + d * e + f * g (inorder)
a b c * + d e * f + g

Binary Tree Traversal

– An Example: Expression Trees

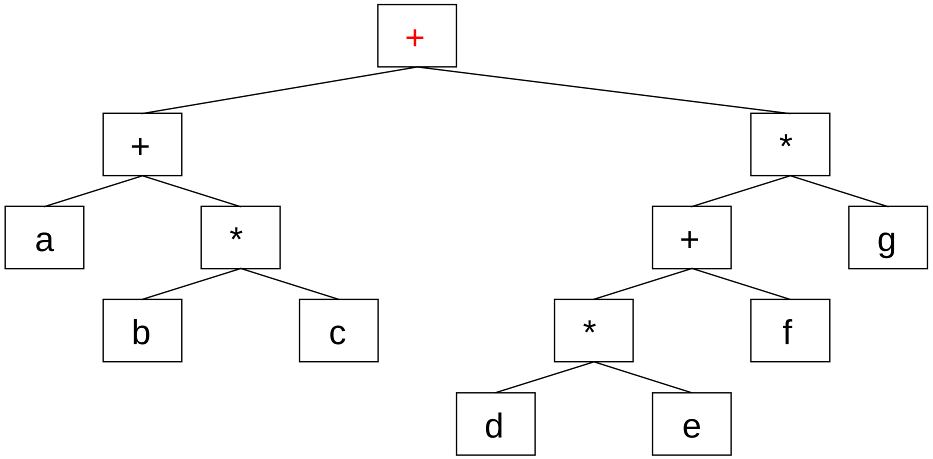


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c + d * e + f * g (inorder)
a b c * + d e * f + g *

Binary Tree Traversal

– An Example: Expression Trees

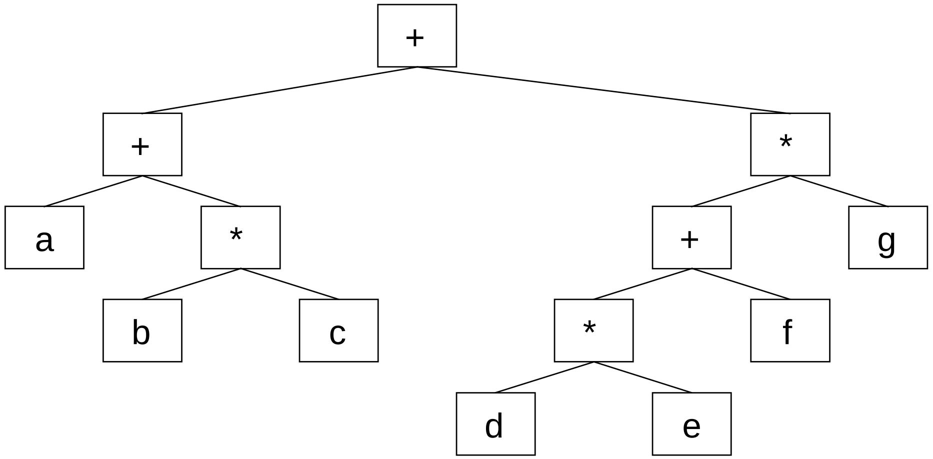


Preorder
inorder
Postorder

+ + a * b c * + * d e f g (preorder)
a + b * c + d * e + f * g (inorder)
a b c * + d e * f + g * +

Binary Tree Traversal

– An Example: Expression Trees



Preorder
inorder
Postorder

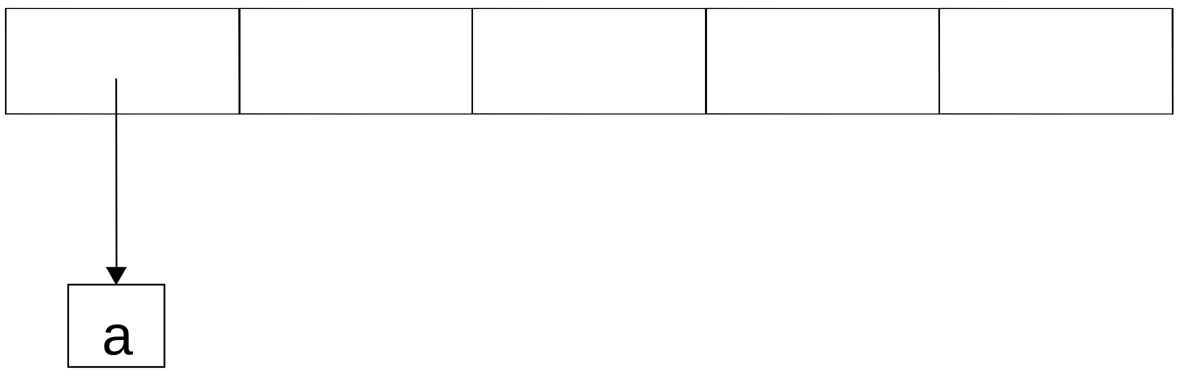
+ + a * b c * + * d e f g (preorder)
a + b * c + d * e + f * g (inorder)
a b c * + d e * f + g * + (postorder)

Binary Tree

- Constructing an Expression Tree from a postfix sequence: post order
- E.g. a b + c d + e * *

Binary Tree

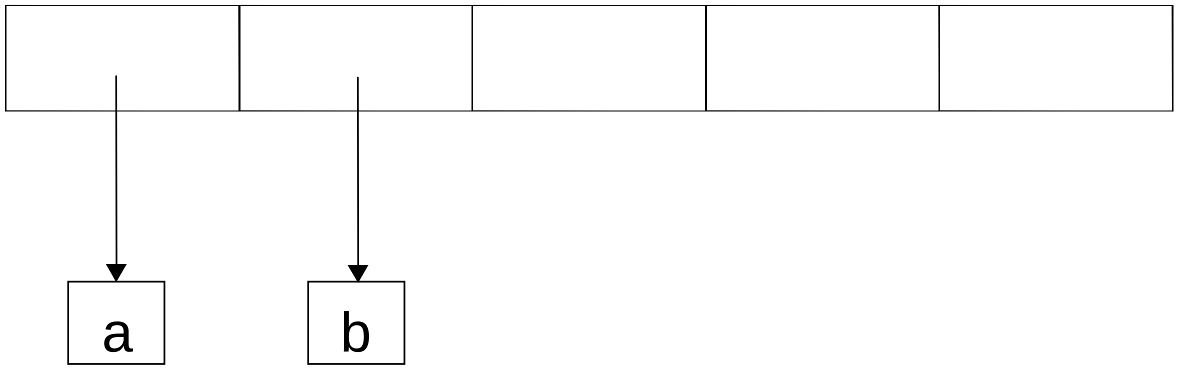
– Constructing an Expression Tree from a postfix sequence
– E.g. a b + c d + e * *



Binary Tree

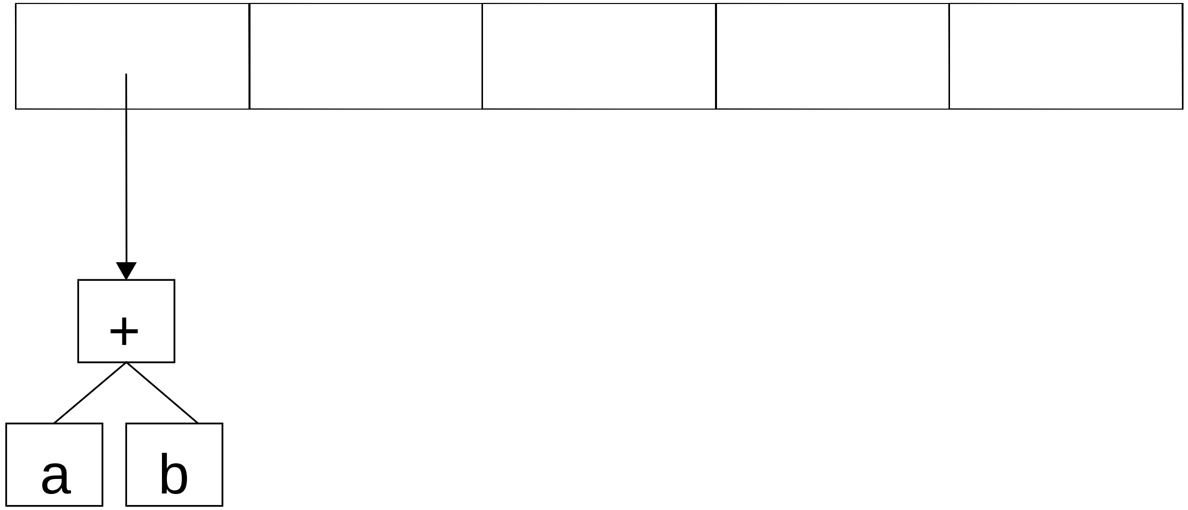
– Constructing an Expression Tree from a postfix sequence

– E.g. a b + c d + e * *



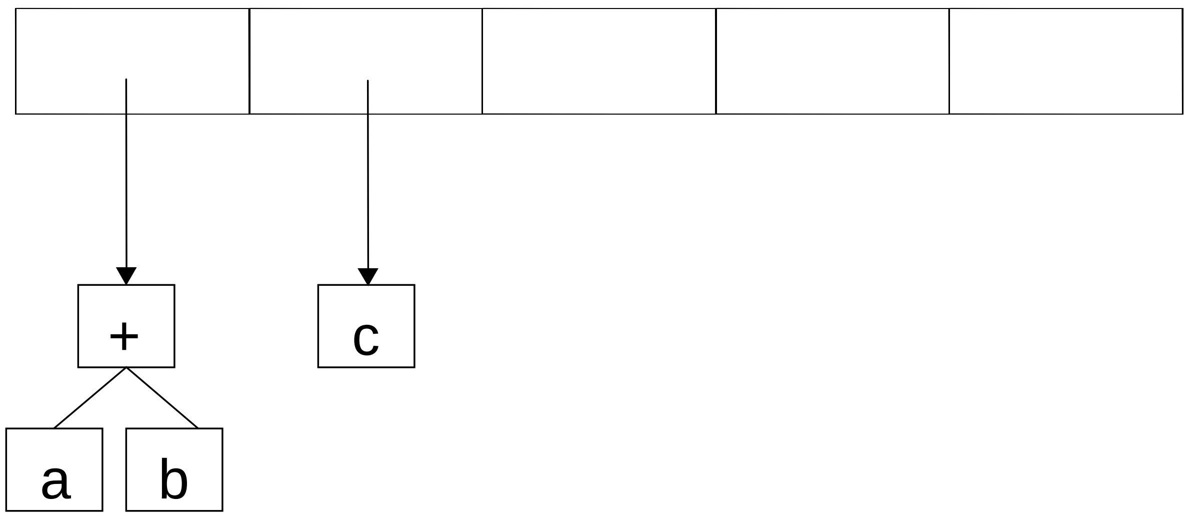
Binary Tree

– Constructing an Expression Tree from a postfix sequence
– E.g. a b + c d + e * *



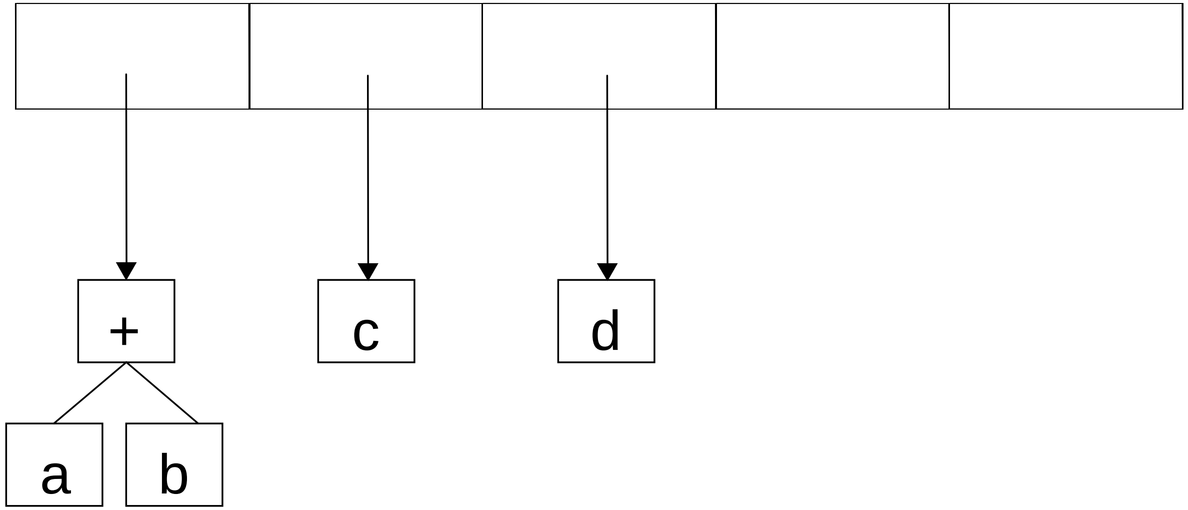
Binary Tree

– Constructing an Expression Tree from a postfix sequence
– E.g. a b + c d + e * *



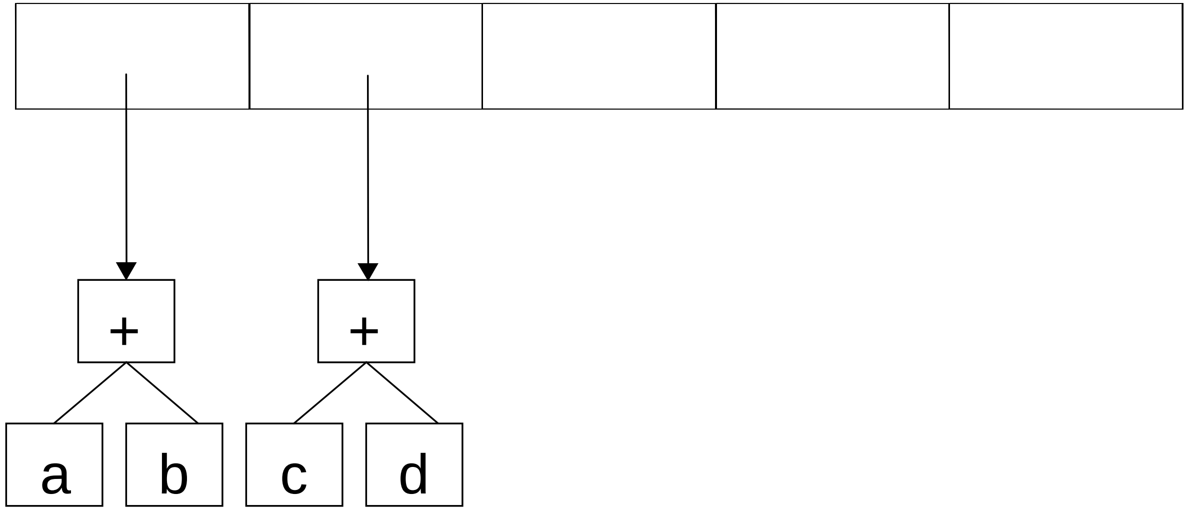
Binary Tree

– Constructing an Expression Tree from a postfix sequence
– E.g. a b + c d + e * *



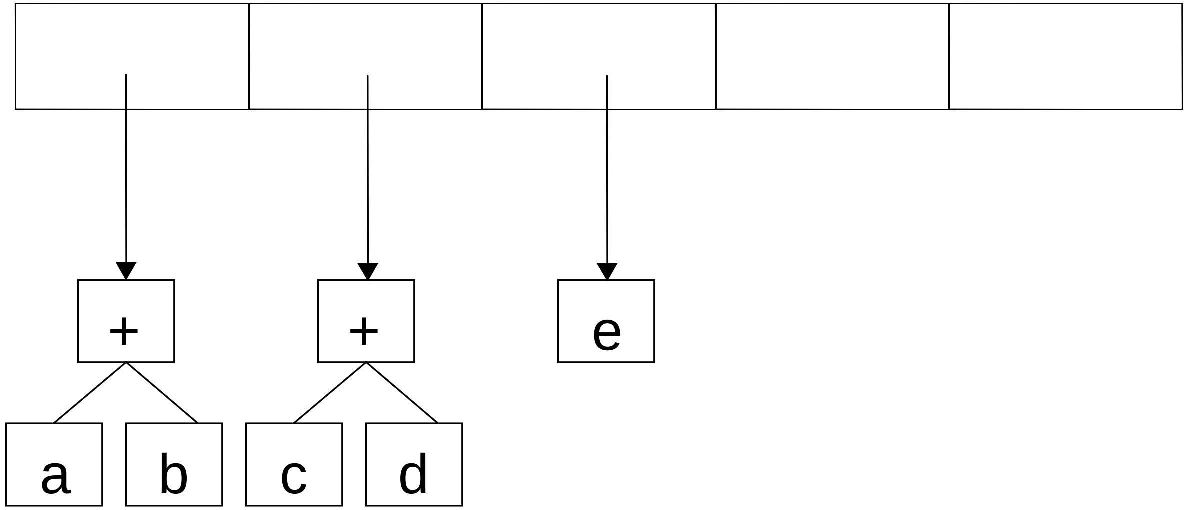
Binary Tree

– Constructing an Expression Tree from a postfix sequence
– E.g. a b + c d + e * *



Binary Tree

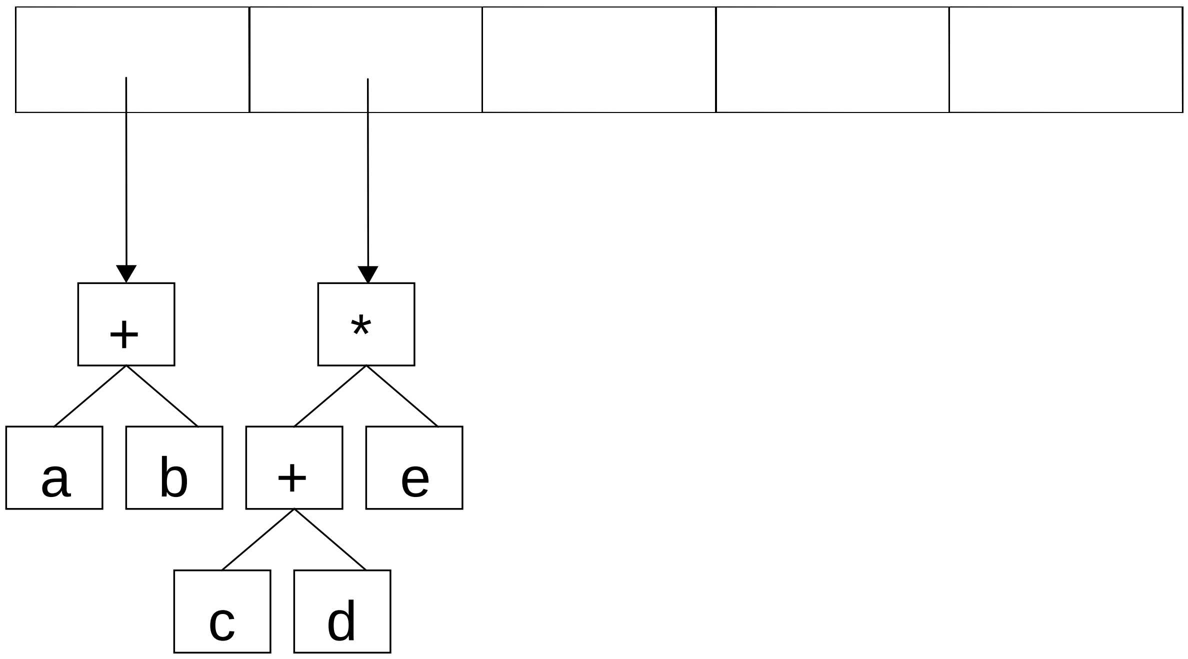
– Constructing an Expression Tree from a postfix sequence
– E.g. a b + c d + e * *



Binary Tree

– Constructing an Expression Tree from a postfix sequence

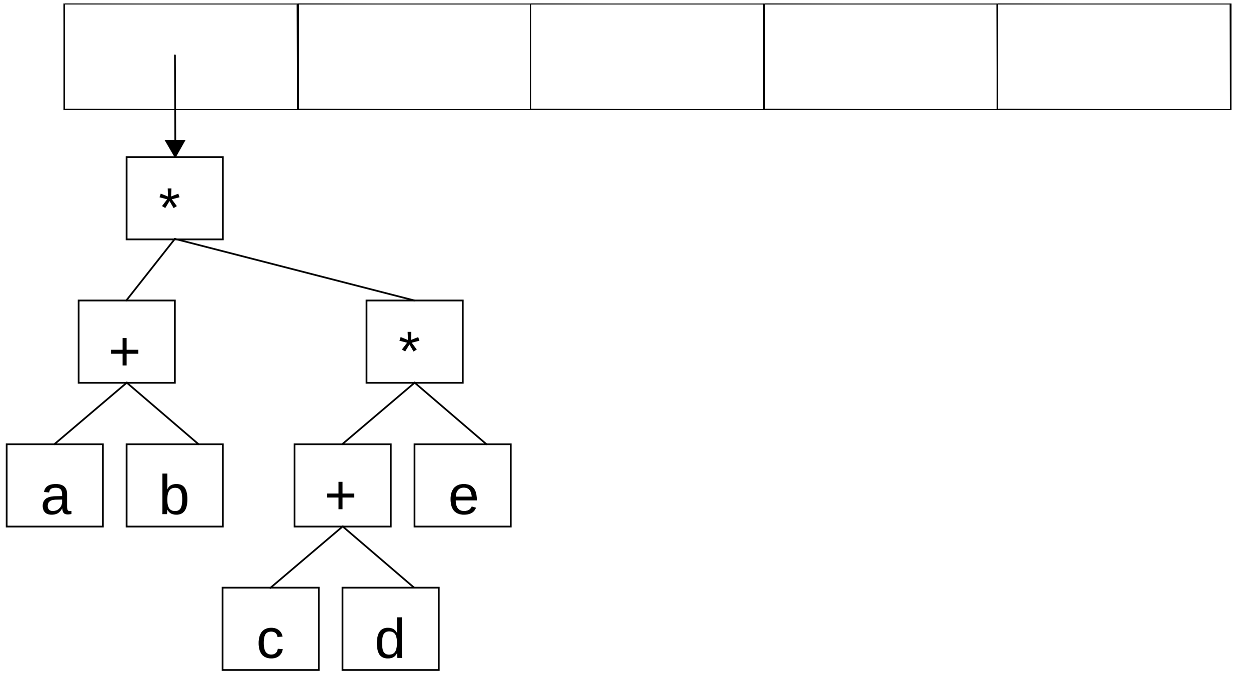
– E.g. a b + c d + e * *



Binary Tree

– Constructing an Expression Tree from a postfix sequence

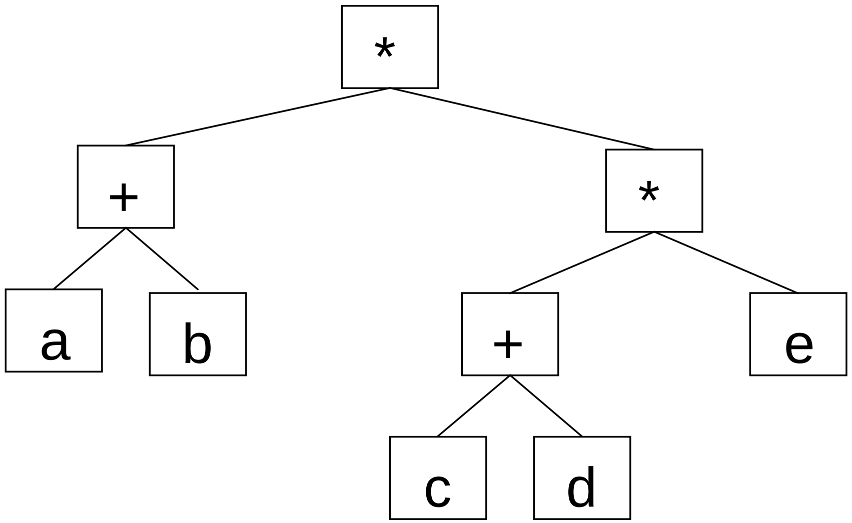
– E.g. a b + c d + e * *



Binary Tree

– Constructing an Expression Tree from a postfix sequence

– E.g. a b + c d + e * *

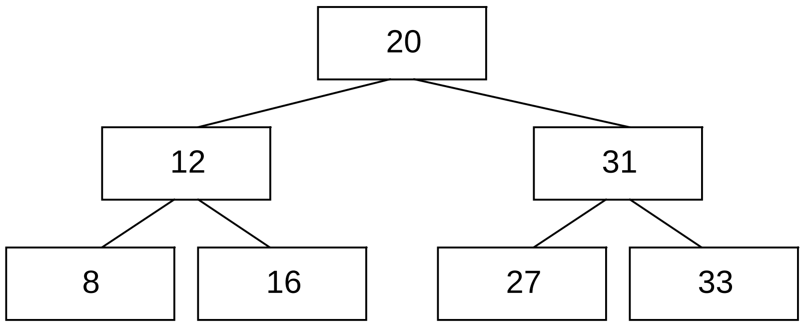


Binary Search Tree

A SEARCH TREE is:

1) The value in each node is greater than or equal to all the values in its left child or any of that child's descendants

2) The value in each node is less than or equal to all the values in its right child or any of that child's descendants



Binary Search Tree

Finding a key

```
function find(key, tree)
    if  tree = nil then return nil      //no match
    else if key < tree^.contents then
        return find(key, tree^.left) //look in left subtree
    else if key > tree^.contents then
        return find(key, tree^.right) //look in right subtree
    else return tree      //match
```

Binary Search Tree

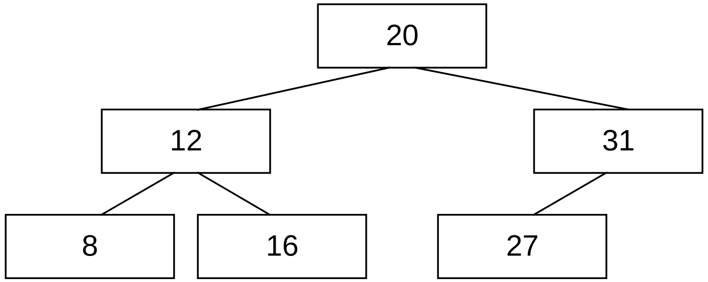
Inserting a key

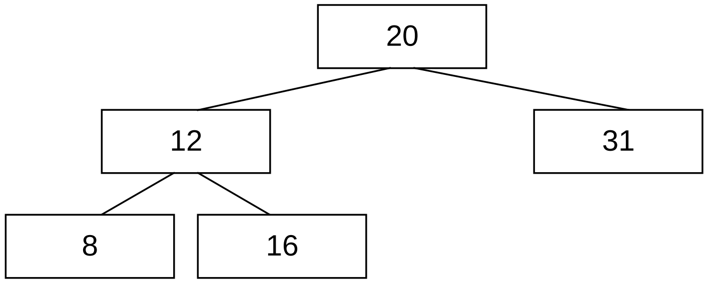
```
function insert(key, tree)
     if  tree = nil then  tree = new node(key, nil, nil)
    else if key < tree^.contents then
           insert(key, tree^.left)
    else if key > tree^.contents then
             insert(key, tree^.right)
      else  ; //do nothing bz key already in tree
```

Binary Search Tree

Removing a key

Case 1: If the node is a leaf just delete it: e.g. delete 27

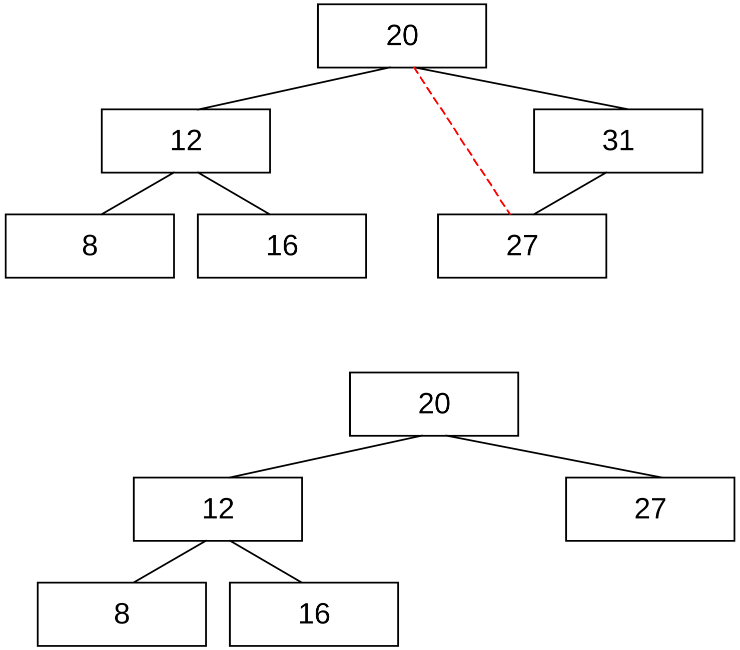




Binary Search Tree

Removing a key

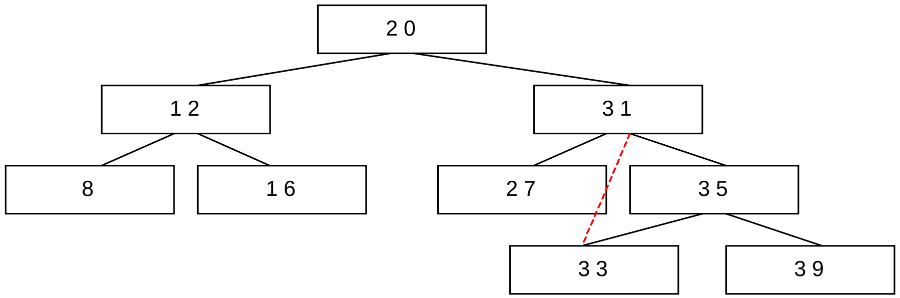
Case 2: If the node has one child relink and delete: e.g. delete 31

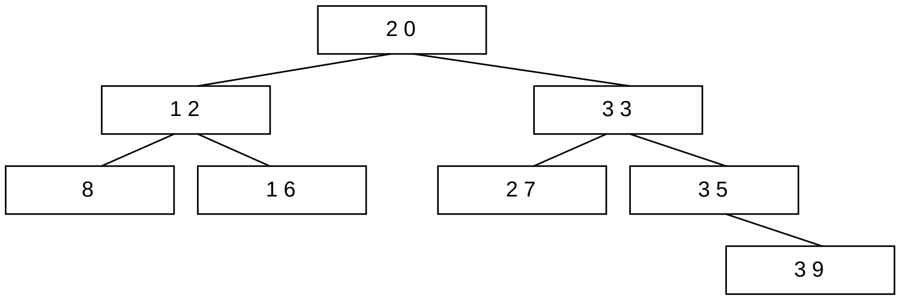


Binary Search Tree

Removing a key***

Case 3: If the node has two children replace contents with contents of minimum value in right subtree and remove that key





Binary Search Trees
– removing a key

```
function remove(key, tree)
  if  tree = nil then  return; //no match, nothing to do
   else if key < tree^.contents then remove(key, tree^.left)
   else if key > tree^.contents then  remove(key, tree^.right)
   else if tree^.left ≠nil & tree^.right≠nil   then
          tree^.contents = find_min(tree^.right)^.contents
           remove(tree^.contents, tree^.right)
    else
            temp = tree
            if tree^.left ≠nil      tree = tree^.left
            else  tree= tree^.right
            delete temp
```

Binary Search Trees
– removing a key: find_min(tree)

```
function find_min(tree)
    if  tree = nil then  return nil;
    if tree^.left = nil then return tree^.contents
    else return find_min(tree^,left)
```

ISSUE and new idea about Binary Search Trees

– Search trees are not usually well balanced
– After insertions and deletions they are typically even less well balanced
– Operations on search trees are $\Theta(\log(n))$ only for balanced trees
– It would be nice to have a "self-balancing" search tree
– AVL trees are such "self-balancing" trees

AVL Trees

An AVL (Adelson-Velski and Landis) tree is
a binary search tree with a balance condition

AVL Trees

An AVL (Adelson-Velski and Landis) tree is
a binary search tree with a balance condition

The condition must be easy to maintain.

AVL Trees

– An AVL (Adelson-Velski and Landis) tree is a binary search tree with a balance condition

– The condition must be easy to maintain.

1. Try “left and right subtrees must be of the same height”

AVL Trees

– An AVL (Adelson-Velski and Landis) tree is a binary search tree with a balance condition

– The condition must be easy to maintain.

1. Try “left and right subtrees must be of the same height”

NO! - **too soft!**

AVL Trees

– An AVL (Adelson-Velski and Landis) tree is a binary search tree with a balance condition

– The condition must be easy to maintain.

1. Try “left and right subtrees must be of the same height”

2. Try “every node must have left and right subtrees of the same

AVL Trees

– An AVL (Adelson-Velski and Landis) tree is a binary search tree with a balance condition

– The condition must be easy to maintain.

1. Try “left and right subtrees must be of the same height”

2. Try “every node must have left and right subtrees of the same

NO! – too hard!

AVL Trees

– An AVL (Adelson-Velski and Landis) tree is a binary search tree with a balance condition

– The condition must be easy to maintain.

1. Try “left and right subtrees must be of the same height”

2. Try “every node must have left and right subtrees of the same

3. Try “every node must have left and right subtrees which differ in height by at most 1”

AVL Trees

– An AVL (Adelson-Velski and Landis) tree is a binary search tree with a balance condition

– The condition must be easy to maintain.

1. Try “left and right subtrees must be of the same height”

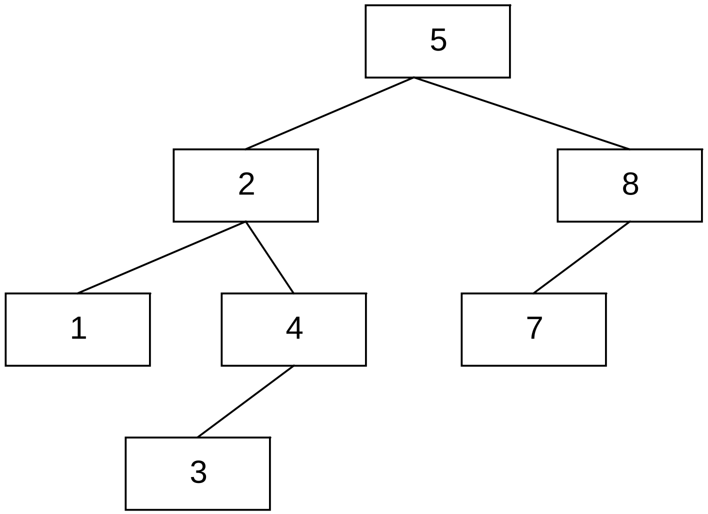
2. Try “every node must have left and right subtrees of the same

3. Try “every node must have left and right subtrees which differ in height by at most 1”

YES! – just right!

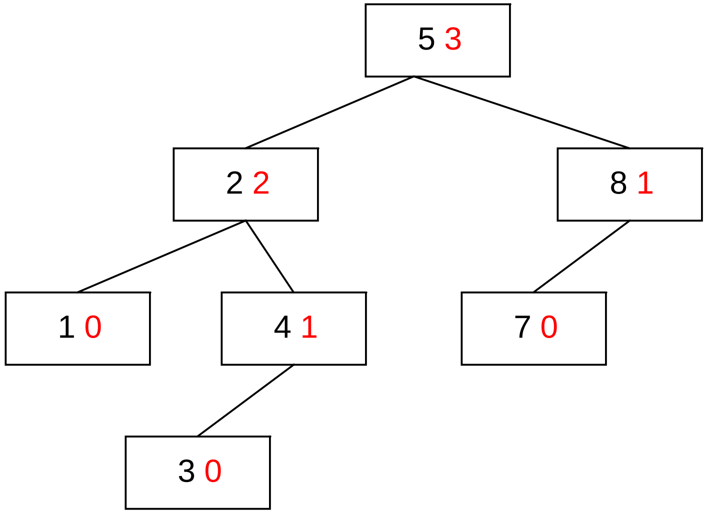
AVL Trees

– **E.g.**



AVL Trees

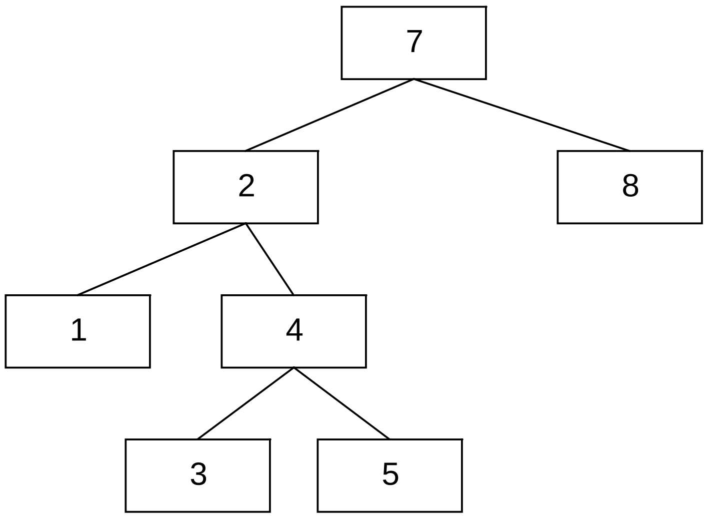
- E.g.



- This **is** an AVL tree (Heights shown in red)

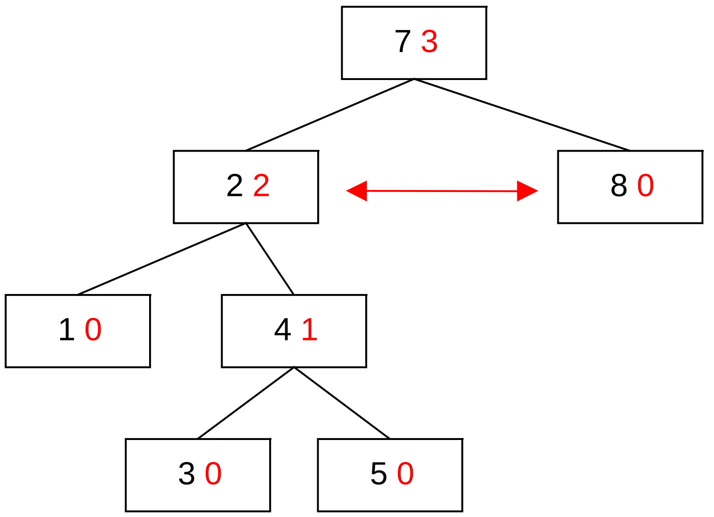
AVL Trees

– **E.g.**



AVL Trees

- **E.g.**
-



- This **is not** an AVL tree (Heights of left and right subtrees differs by 2)

AVL Trees

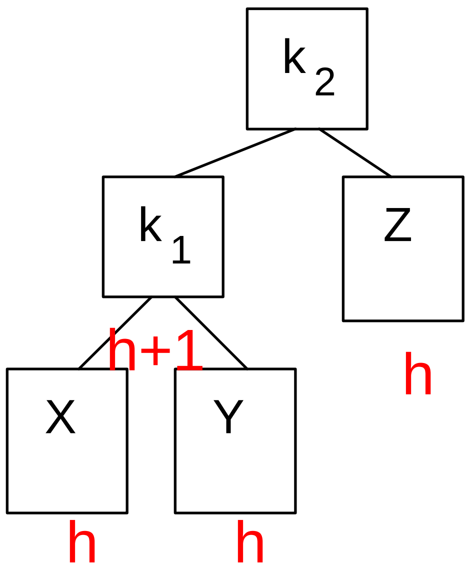
Insertion can unbalance AVL tree node

1. An insertion into the left subtree of the left child of
2. An insertion into the right subtree of the left child of
3. An insertion into the left subtree of the right child of
4. An insertion into the right subtree of the right child of

- Cases 1 and 4 are equivalent, as are cases 2 and 3 (although there are still 4 cases from a coding viewpoint).

AVL Trees

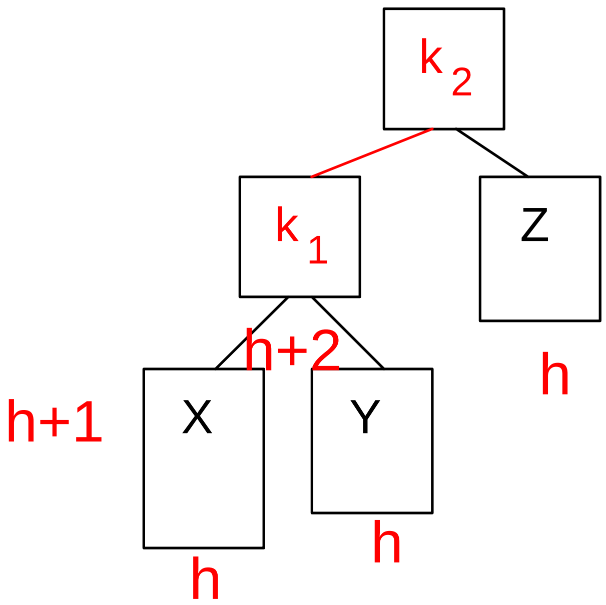
– Case 1: insertion into the left subtree of the left child of k2
– Consider:



-Before insertion of the node (X, Y and Z are sub-trees)

AVL Trees

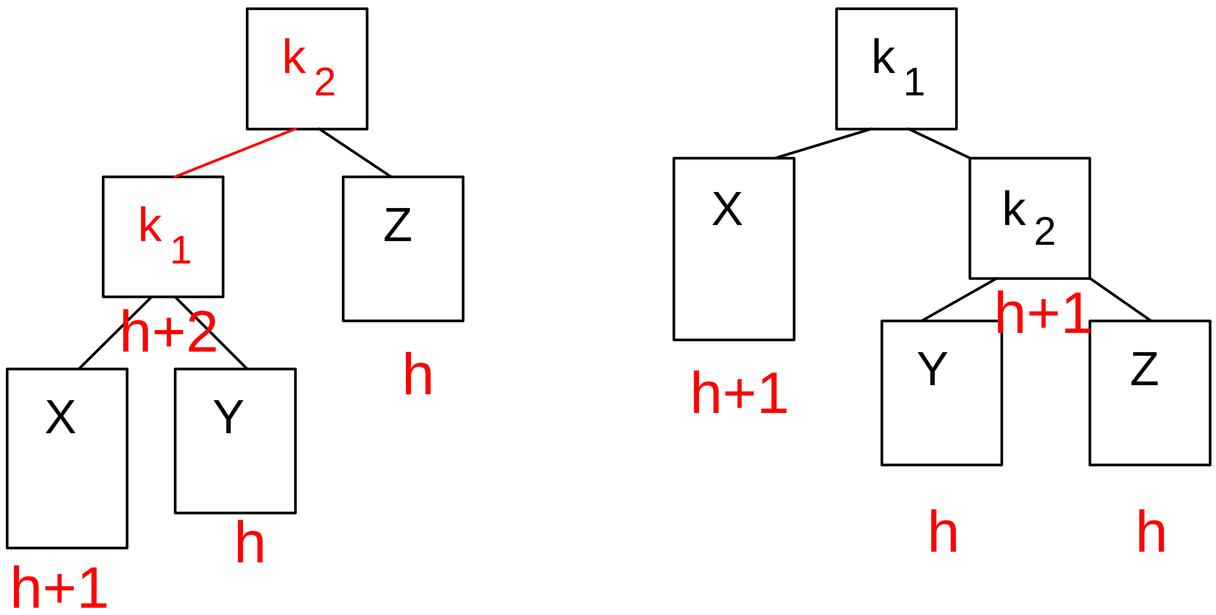
– Case 1: insertion into the left subtree of the left child of k2
– Consider:



– After insertion of the node

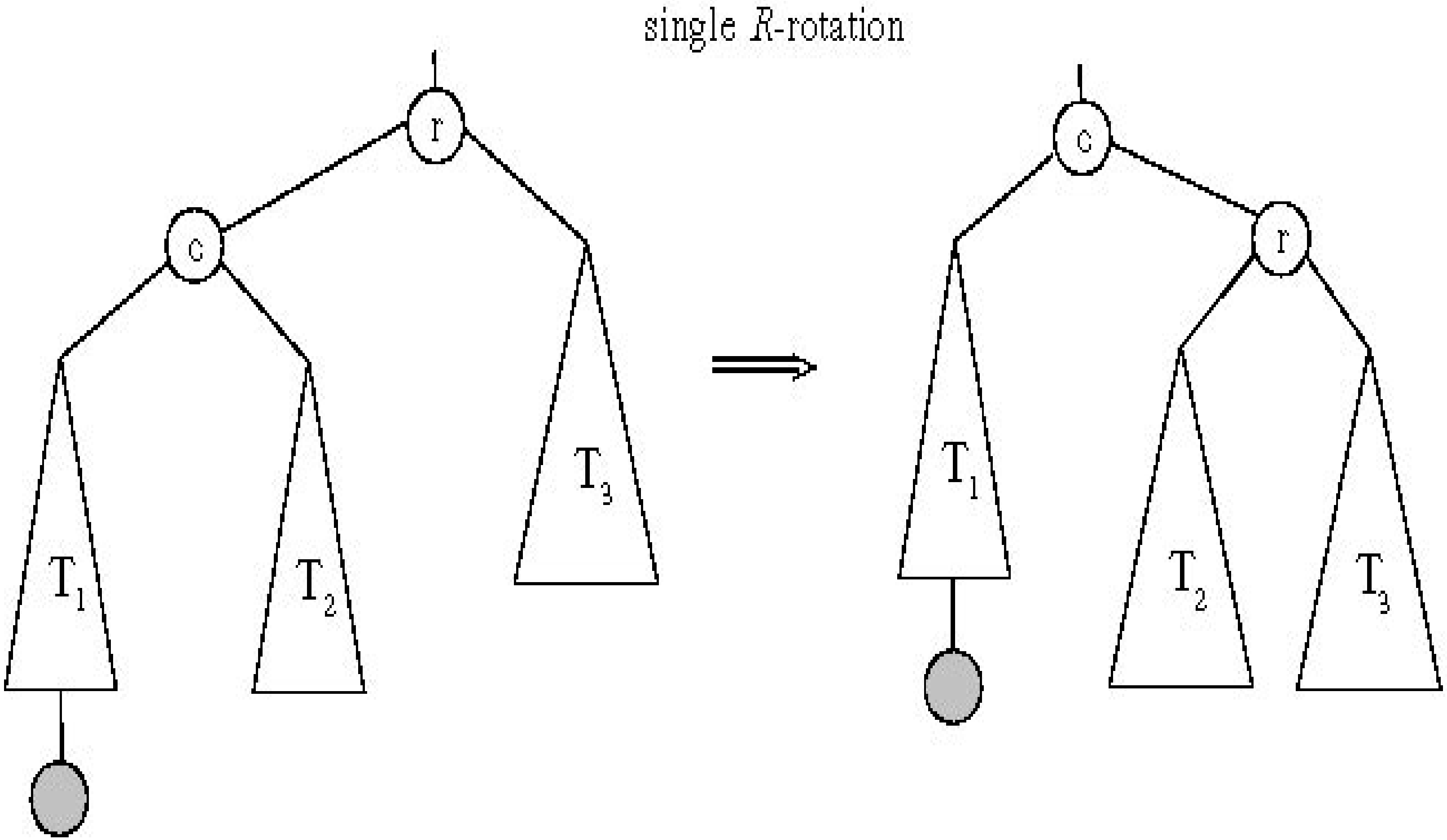
AVL Trees

– Case 1: insertion into the left subtree of the left child of k2
– Consider:



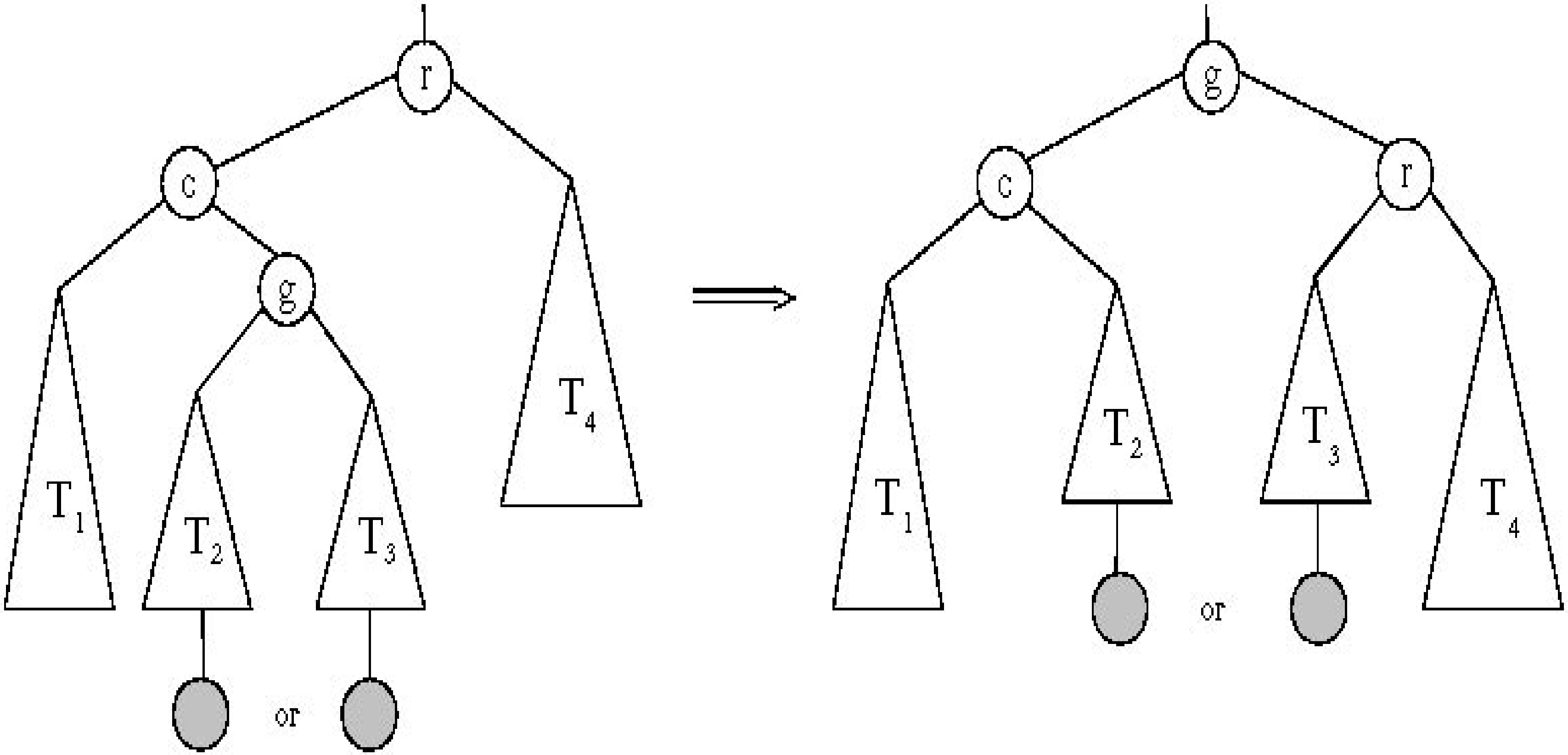
- A single rotation rebalances the tree

AVL Trees (Pattern)



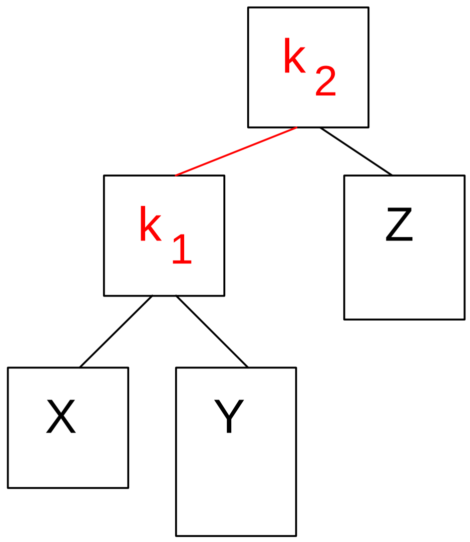
AVL Trees

double *LR*-rotation



AVL Trees

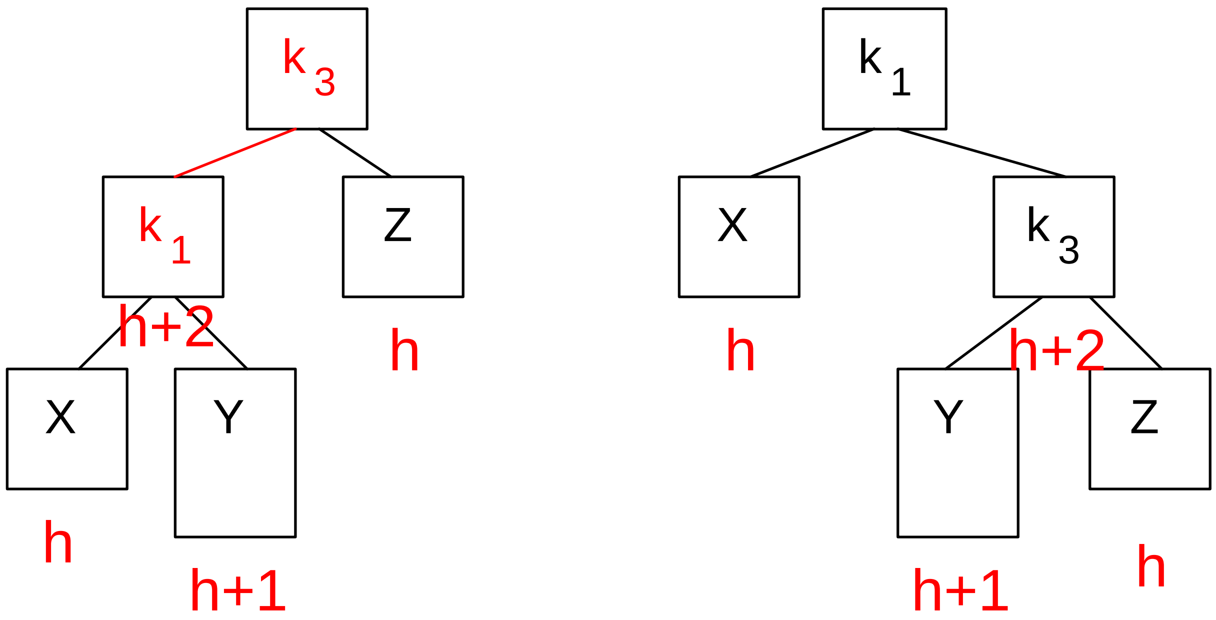
- Case 2: insertion into the right subtree of the left child of k2
- Consider:



- A single rotation does not rebalance the tree.

AVL Trees

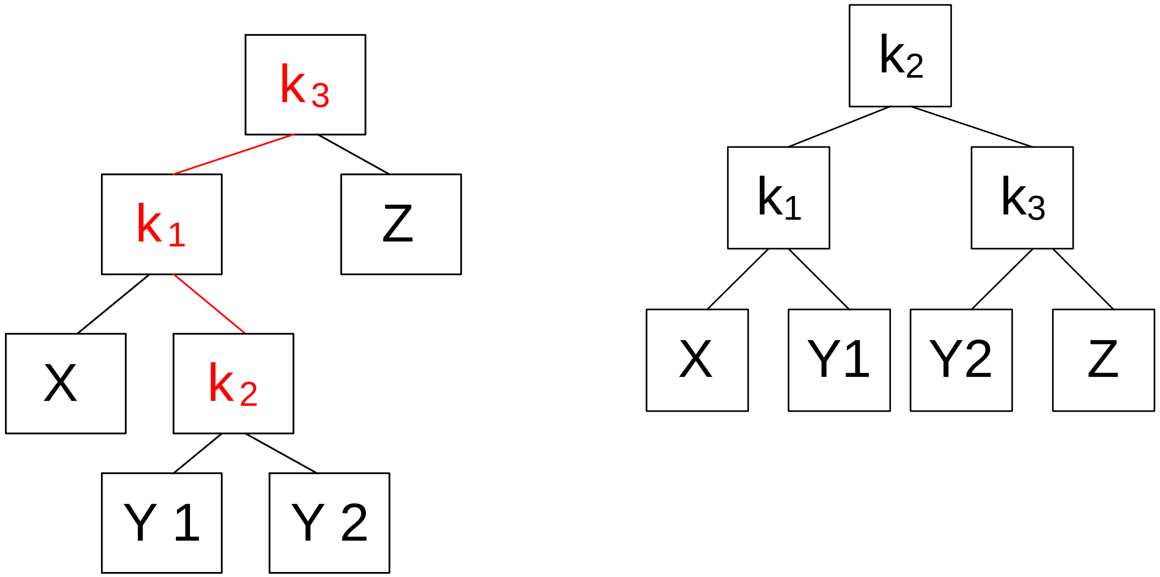
- Case 2: insertion into the right subtree of the left child of k2
- Consider:



- A single rotation does not rebalance the tree.

AVL Trees

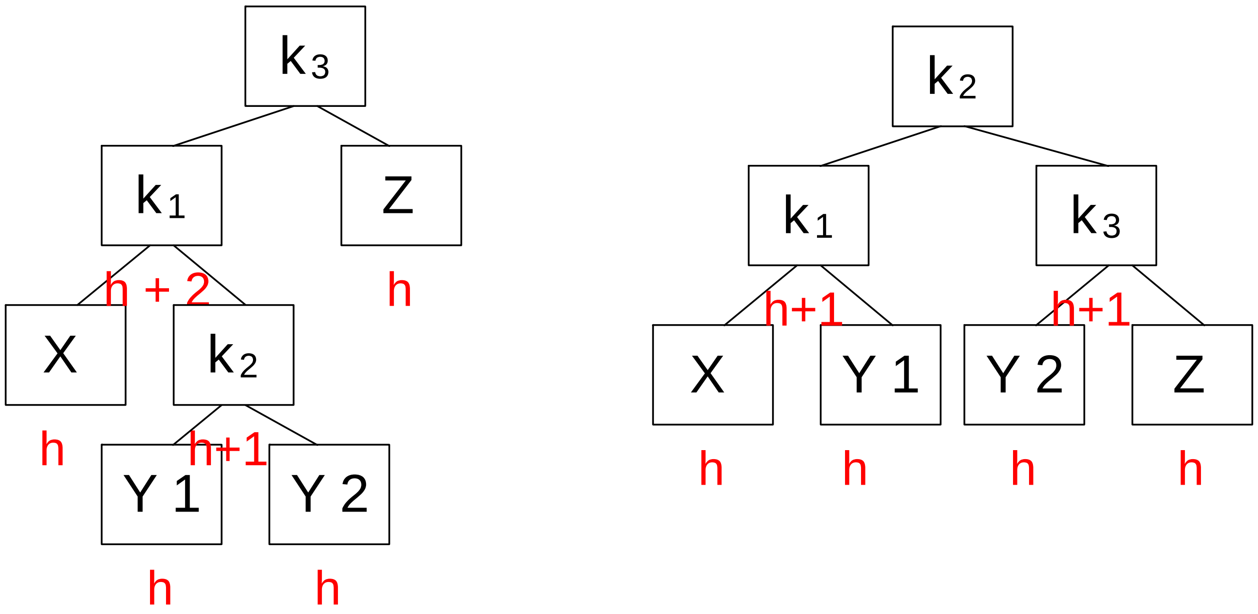
- AVL Trees
  - Case 2: insertion into the right subtree of the left child of
  - Consider:



A double rotation rebalances the tree.

AVL Trees

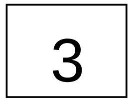
- AVL Trees
  - Case 2: insertion into the right subtree of the left child of
  - Consider:



A double rotation rebalances the tree.

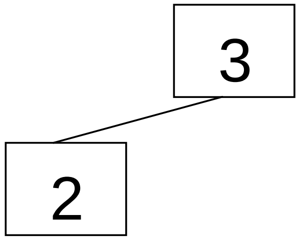
AVL Trees

- AVL Trees
  - An example: Insert 3



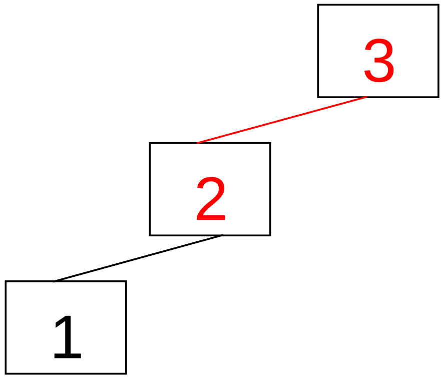
AVL Trees

- AVL Trees
  - An example: Insert 2



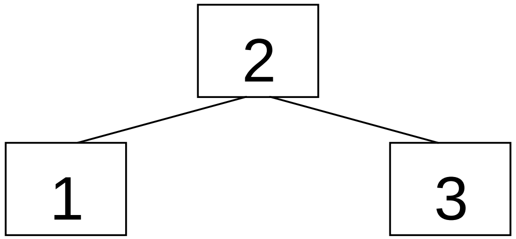
AVL Trees

- AVL Trees
  - An example: Insert 1



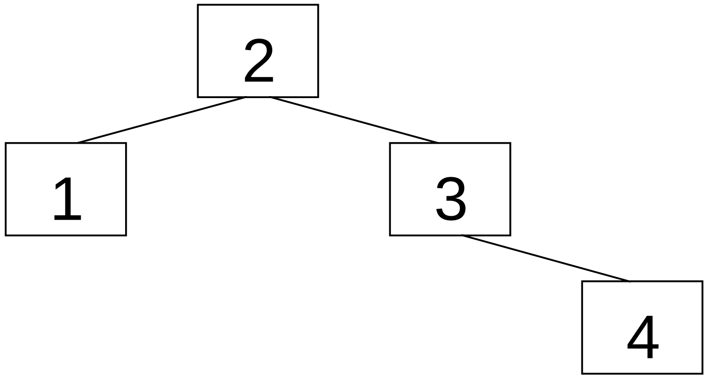
AVL Trees

- AVL Trees
  - An example: rebalance (single rotation)



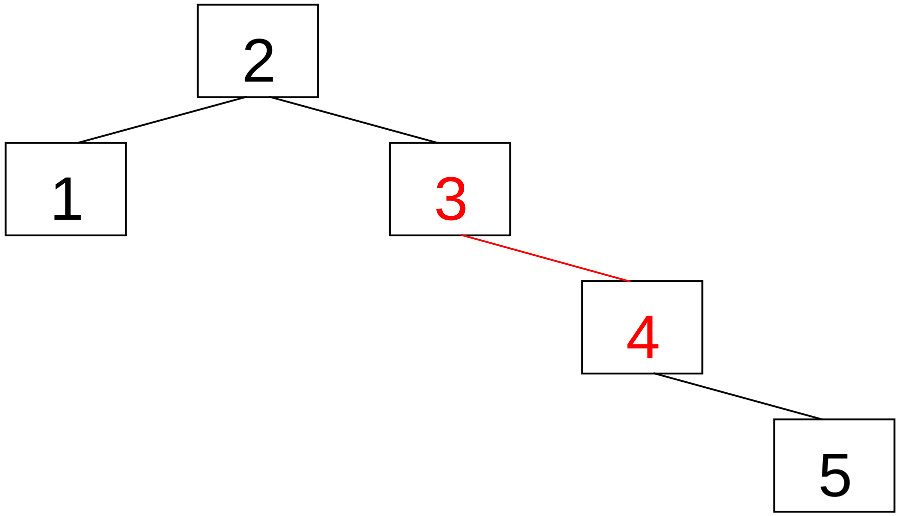
AVL Trees

- AVL Trees
  - An example: insert 4



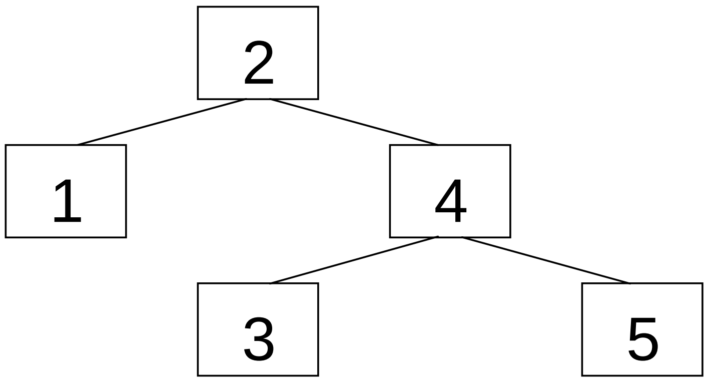
AVL Trees

- AVL Trees
  - An example: insert 5



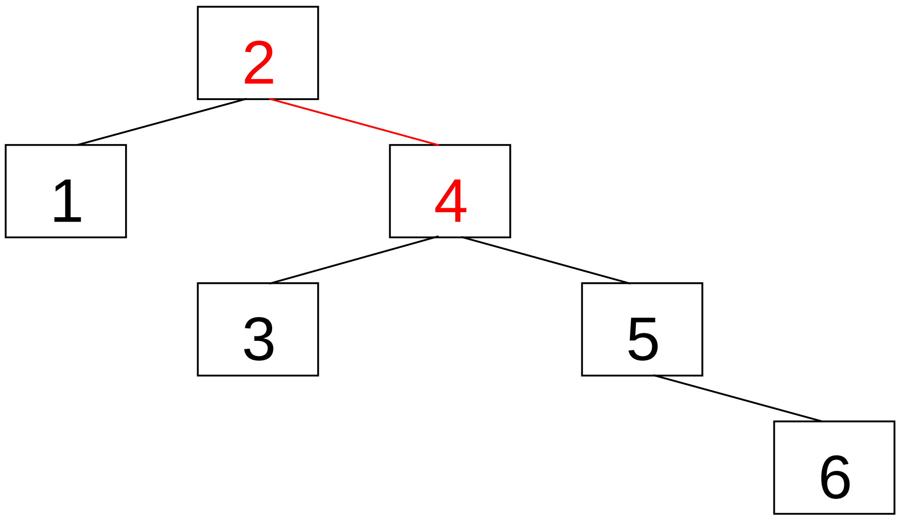
AVL Trees

- AVL Trees
  - An example: rebalance (single rotation)



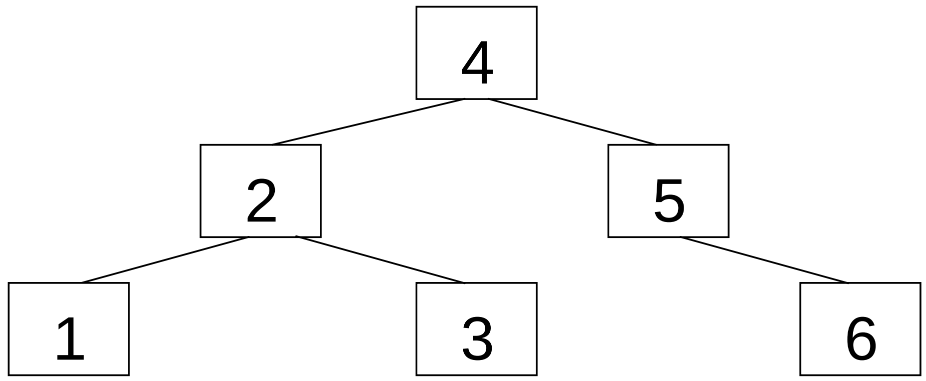
AVL Trees

- AVL Trees
  - An example: insert 6



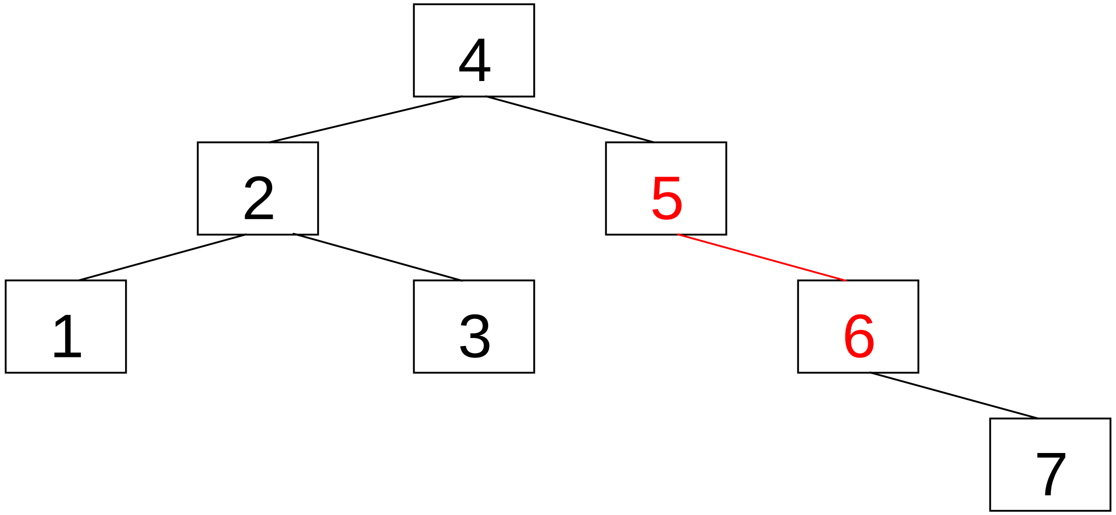
AVL Trees

- AVL Trees
  - An example: rebalance (single rotation)



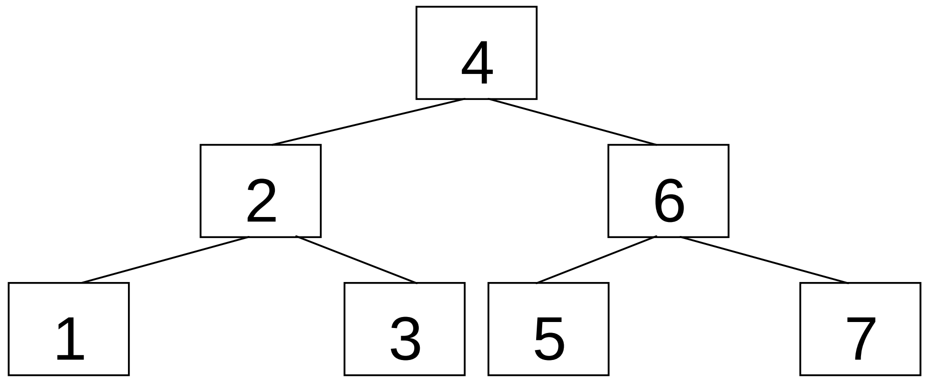
AVL Trees

- AVL Trees
  - An example: insert 7



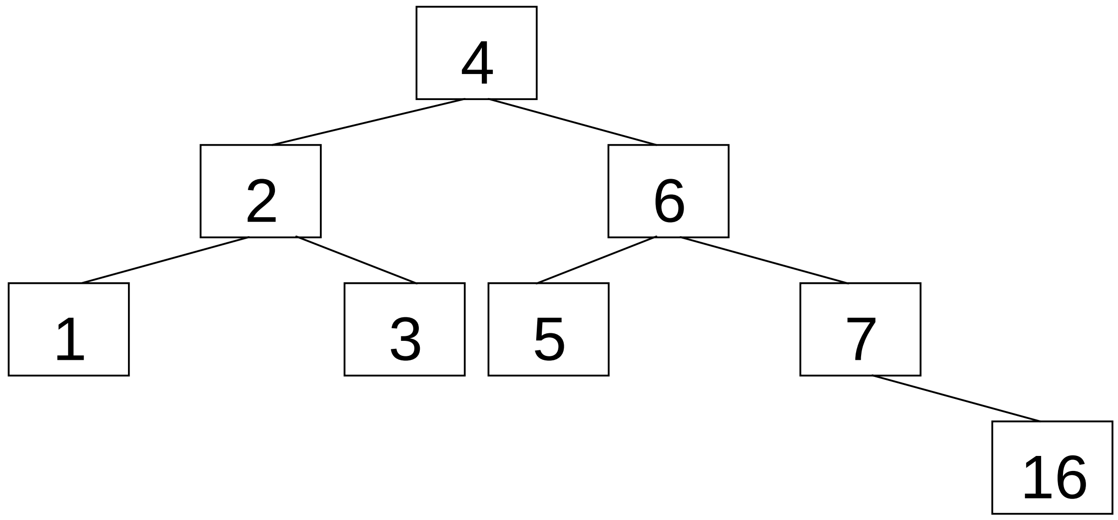
AVL Trees

- AVL Trees
  - An example: rebalance (single rotation)



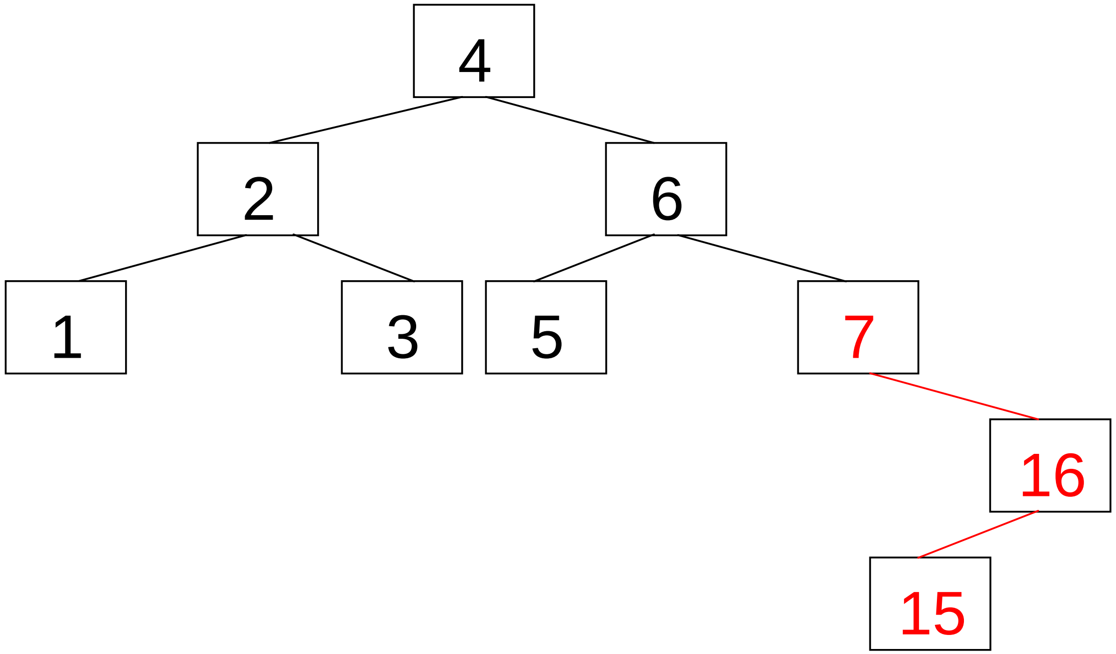
AVL Trees

- AVL Trees
  - An example: insert 16



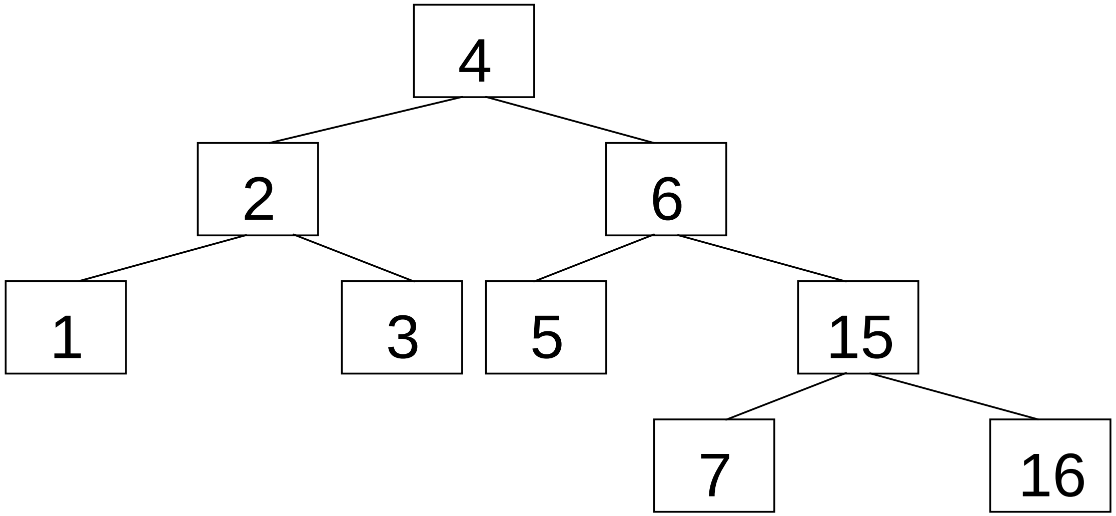
AVL Trees

- AVL Trees
  - An example: insert 15



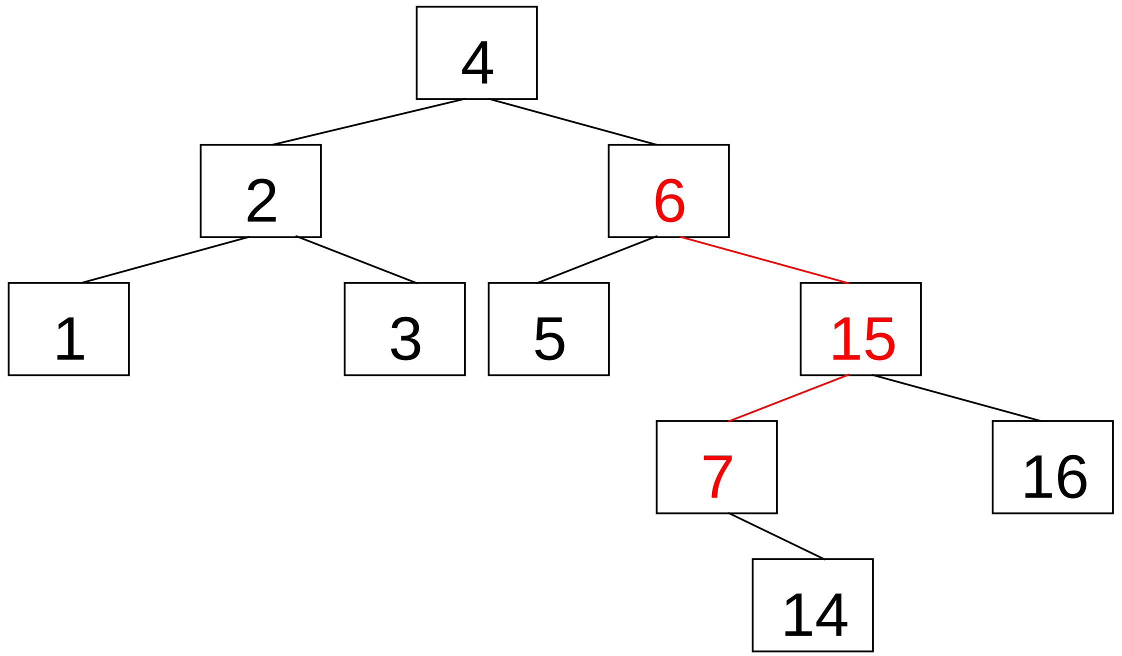
AVL Trees

- AVL Trees
  - An example: rebalance (double rotation)

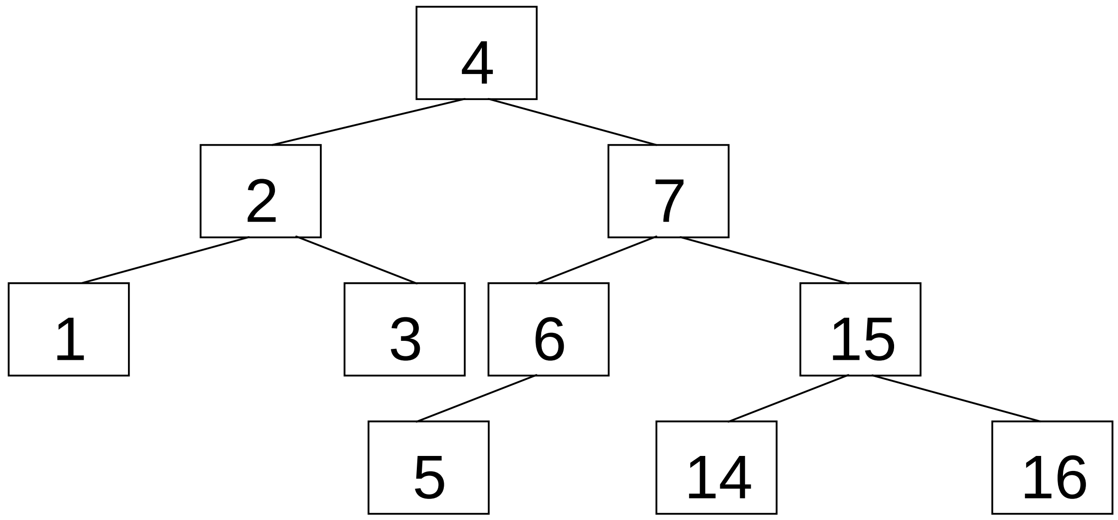


AVL Trees

- AVL Trees
  - An example: insert 14

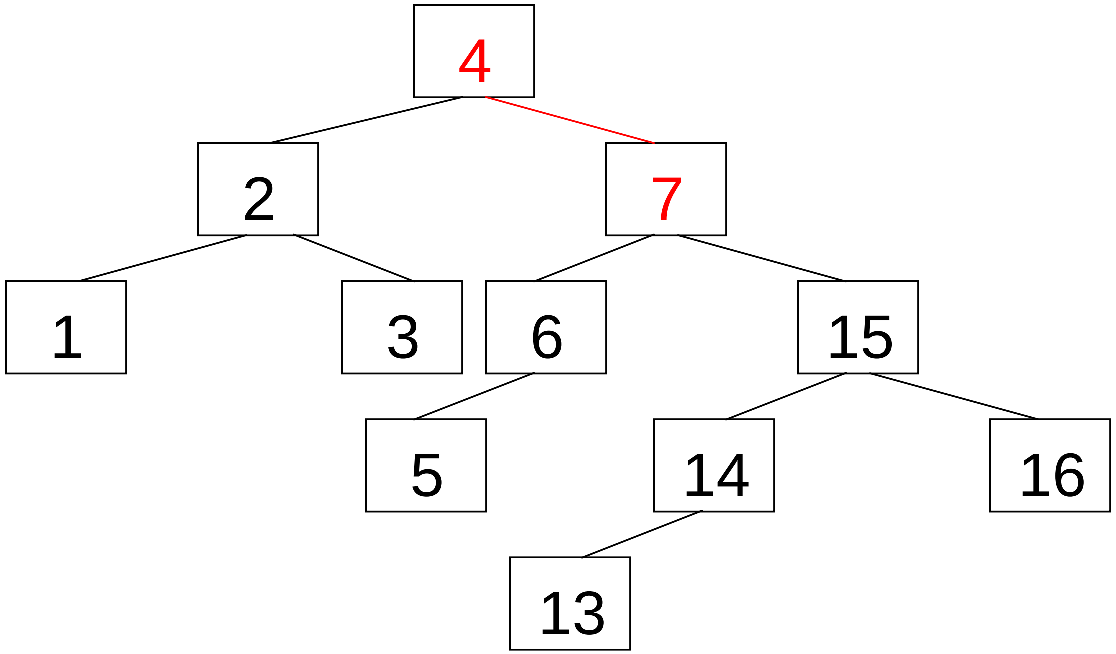


- AVL Trees
  - An example: rebalance (double rotation)



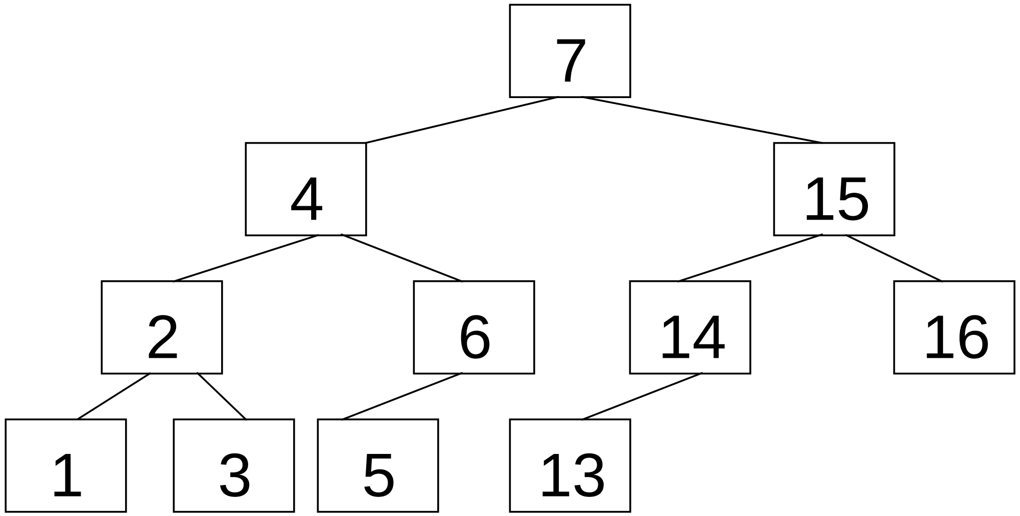
AVL Trees

- AVL Trees
  - An example: insert 13



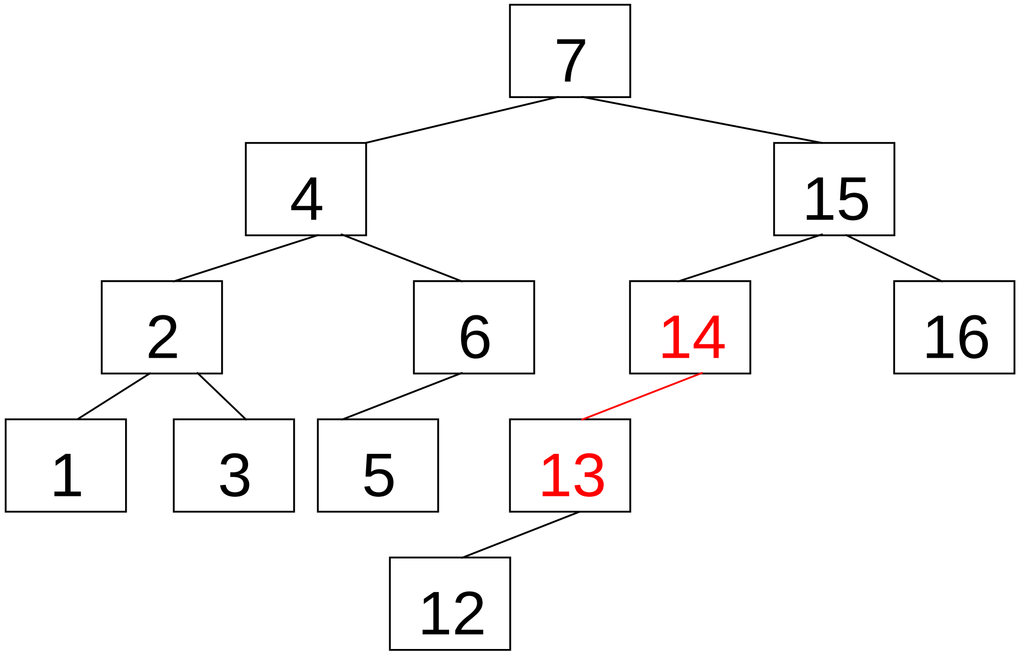
AVL Trees

- AVL Trees
  - An example: rebalance (single rotation)

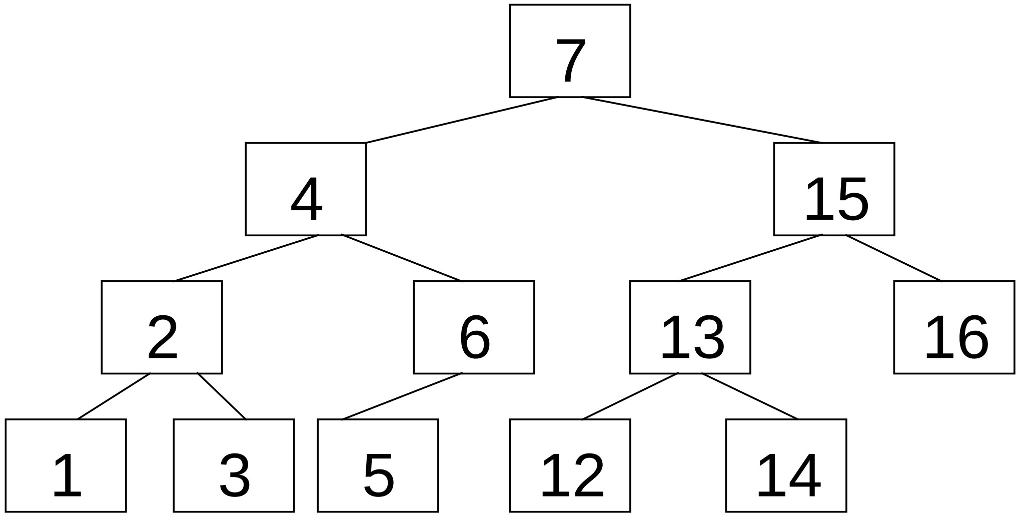


AVL Trees

- AVL Trees
  - An example: insert 12

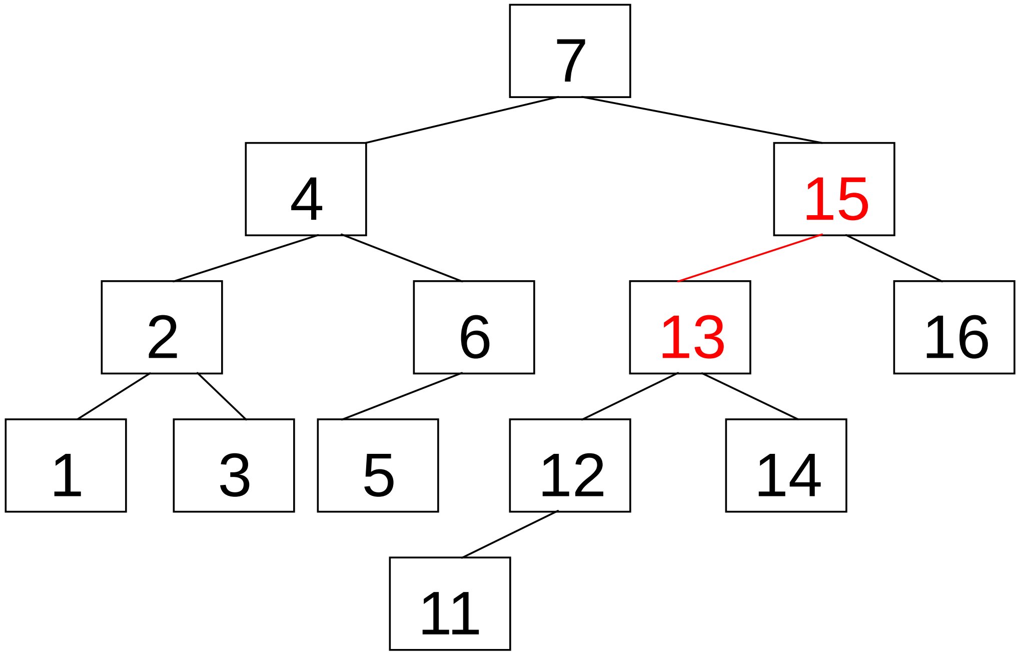


- AVL Trees
  - An example: rebalance (single rotation)



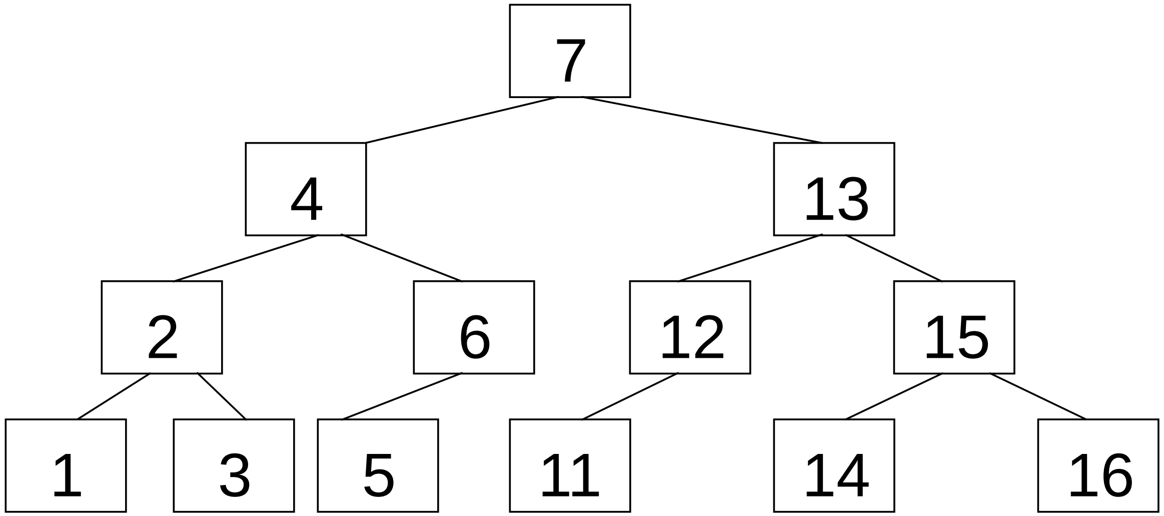
AVL Trees

- AVL Trees
  - An example: insert 11



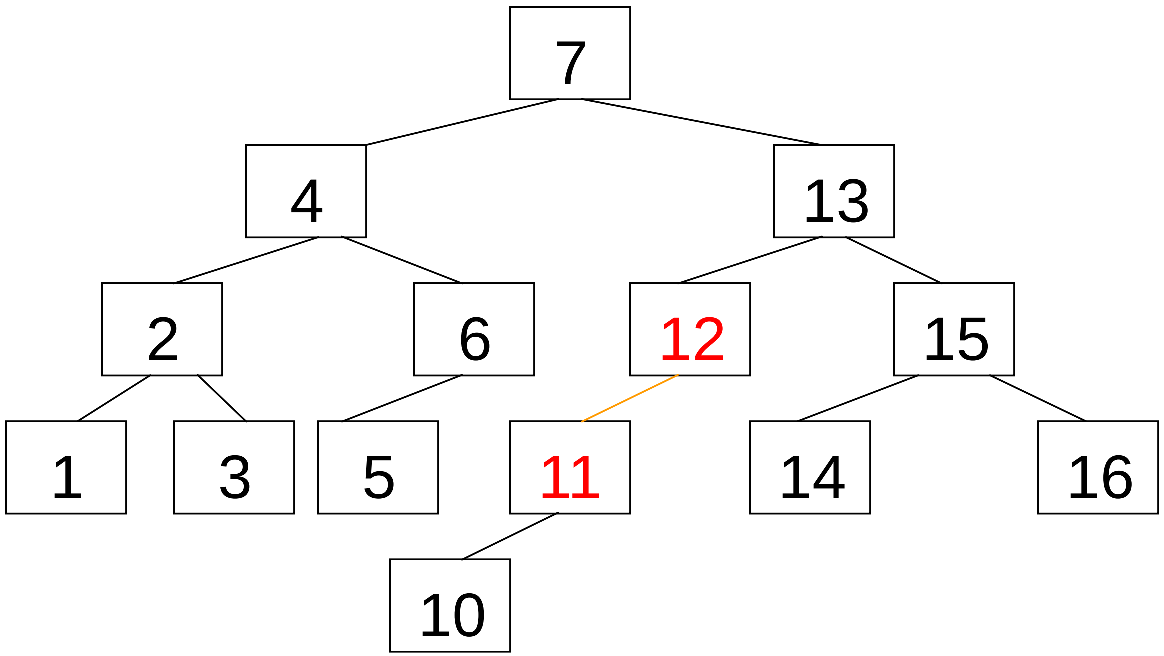
AVL Trees

- AVL Trees
  - An example: rebalance (single rotation)



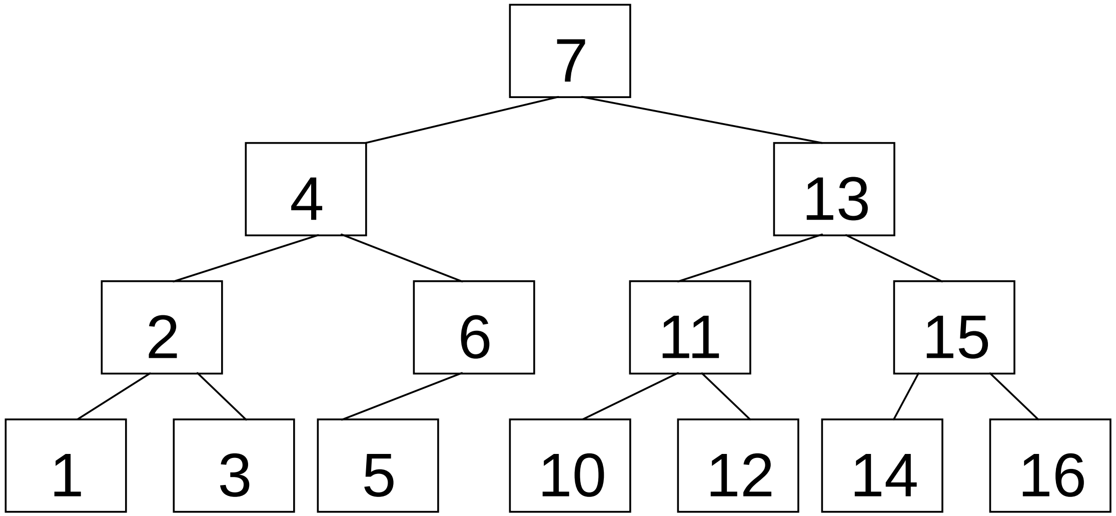
AVL Trees

- AVL Trees
  - An example: Insert 10



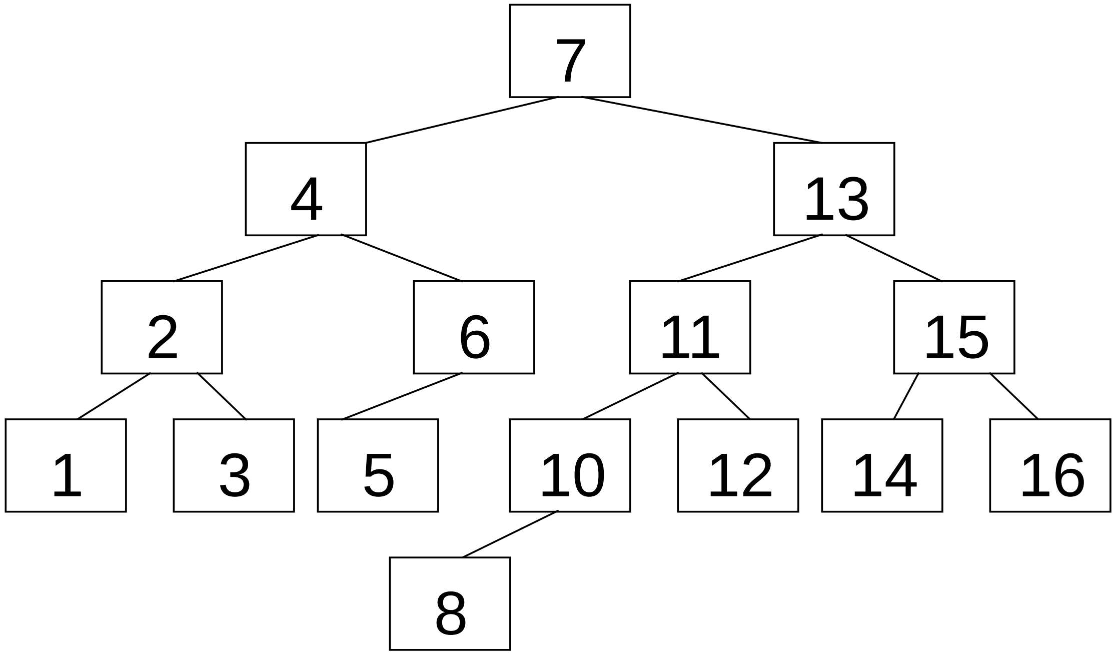
AVL Trees

- AVL Trees
  - An example: rebalance (single rotation)



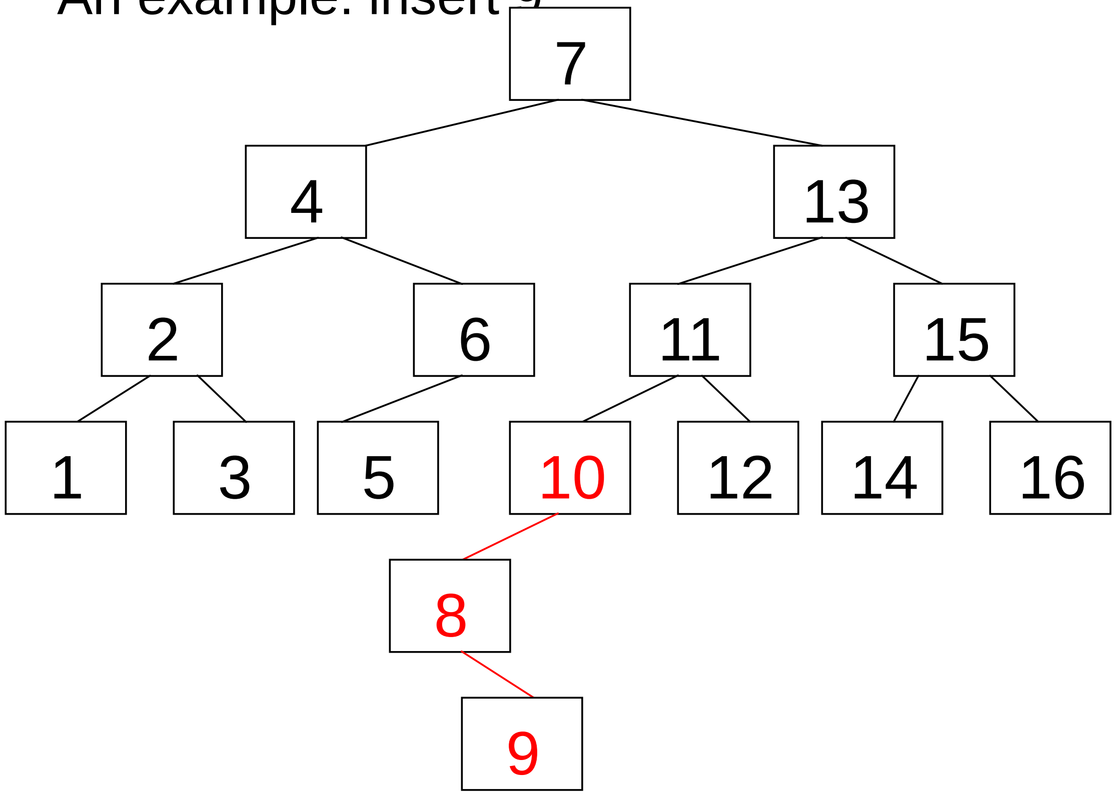
AVL Trees

- AVL Trees
  - An example: insert 8



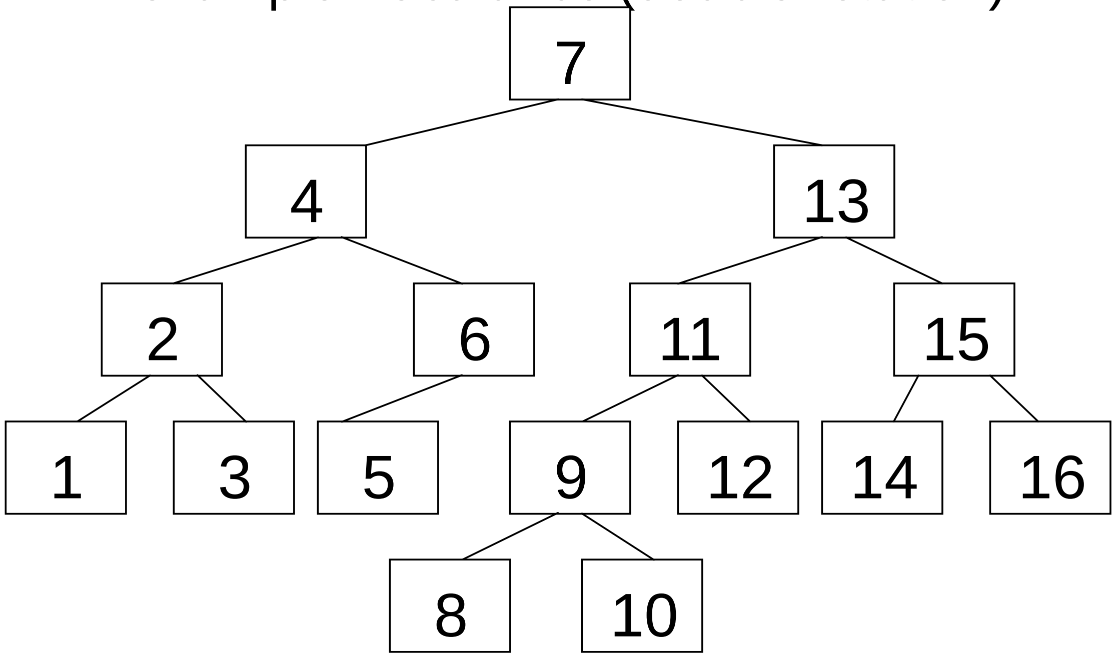
AVL Trees

- AVL Trees
  - An example: insert 9



AVL Trees

- AVL Trees
  - An example: rebalance (double rotation)



AVL Trees

- AVL Trees – Implementation

- type avl_node = record
  value: stuff
  left: ^avl_node
  right: ^avl_node
  height: int

AVL Trees

- AVL Trees – Implementation
- function avl_insert(key, tree)

```
function avl_insert(key, tree)
   if (tree = nil) then
      tree = new avl_node(key, nil, nil, 0)
   else if  key < tree^.value then
      avl_insert(key, tree^.left)
      if (tree^.left)^.height – (tree^.right)^.height) = 2 then
         if  key < (tree^.left)^.value then
            rotate_left_child(tree) // case 1
         else
            double_left_child(tree) // case 2
   else if  key > tree^.value then
      avl_insert(key, tree^.right)
         if (tree^.right)^.height – (tree^.left)^.height) = 2 then
            if  key < (tree^.right)^.value then
               double_right_child(tree) // case 3
            else
               rotate_right_child(tree) // case 4
   tree^.height = max((tree^.left)^.height, (tree^.right)^.height) + 1
```

AVL Trees

- AVL Trees – Implementation
- function rotate_left_child(k2)

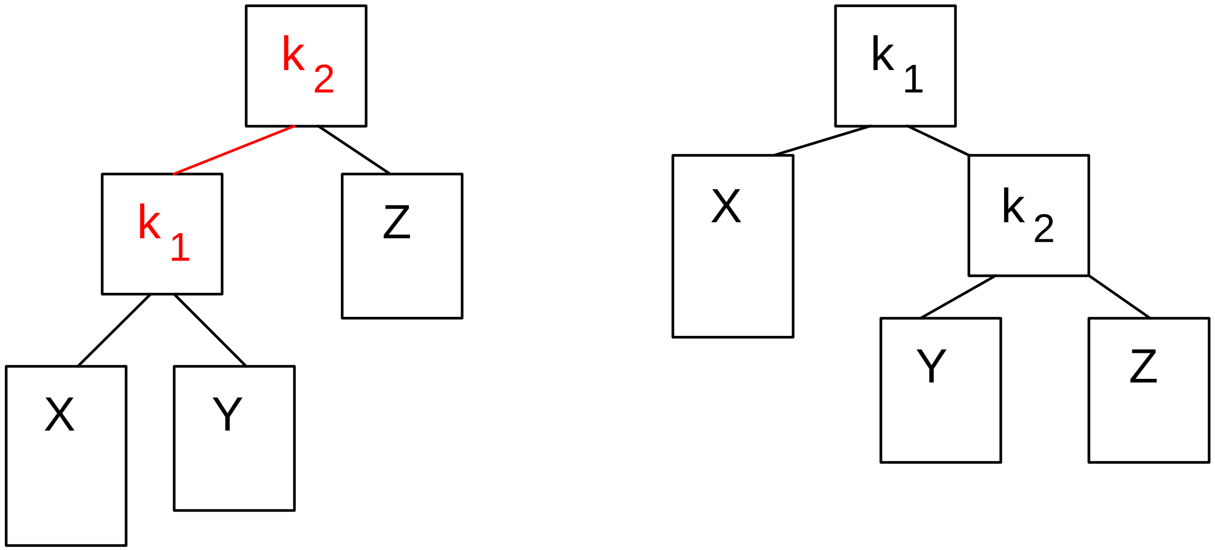
```
k1 = k2^.left
k2^.left = k1^.right
k1^.right = k2
k2^.height = max((k2^.left)^.height), (k2^.right)^.height) + 1
k1^.height = max((k1^.left)^.height), k2^.height) + 1
k2 = k1
```

- function rotate_right_child(k2)

```
k1 = k2^.right
k2^.right = k1^.left
k1^.left = k2
k2^.height = max((k2^.left)^.height), (k2^.right)^.height) + 1
k1^.height = max(k2^.height, (k1^.right)^.height), ) + 1
k2 = k1
```

AVL Trees

- AVL Trees
  - Case 1: insertion into the left subtree of the left child of k_2
  - rotate_left_child(k2)



  - A single rotation rebalances the tree

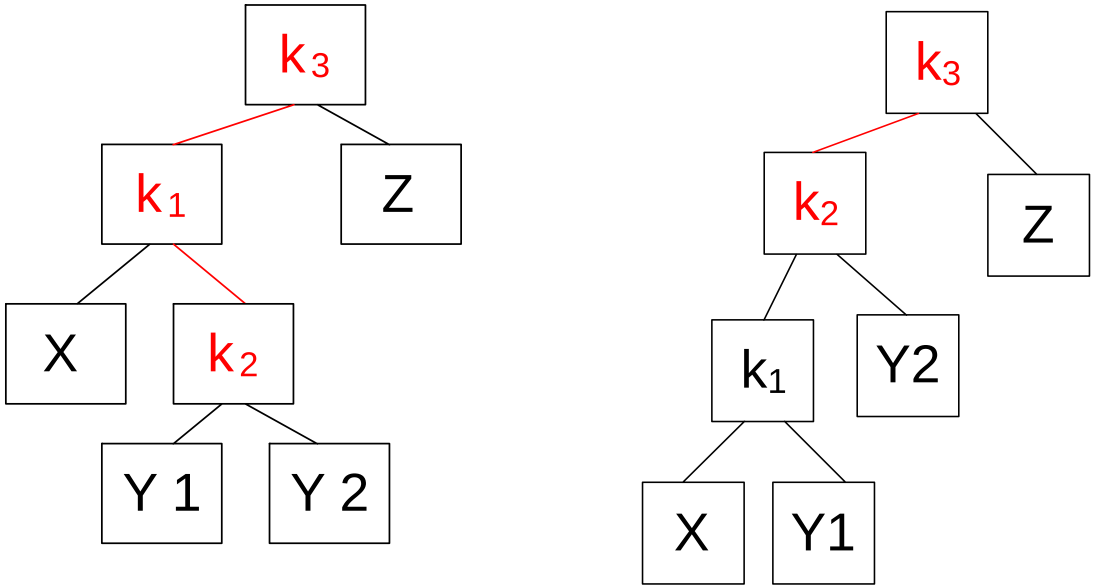
AVL Trees

- AVL Trees – Implementation
- function double_left_child(k3)
  rotate_right_child(k3^.left)
  rotate_left_child(k3)

  function double_right_child(k3)
  rotate_left_child(k3^.right)
  rotate_right_child(k3)

AVL Trees

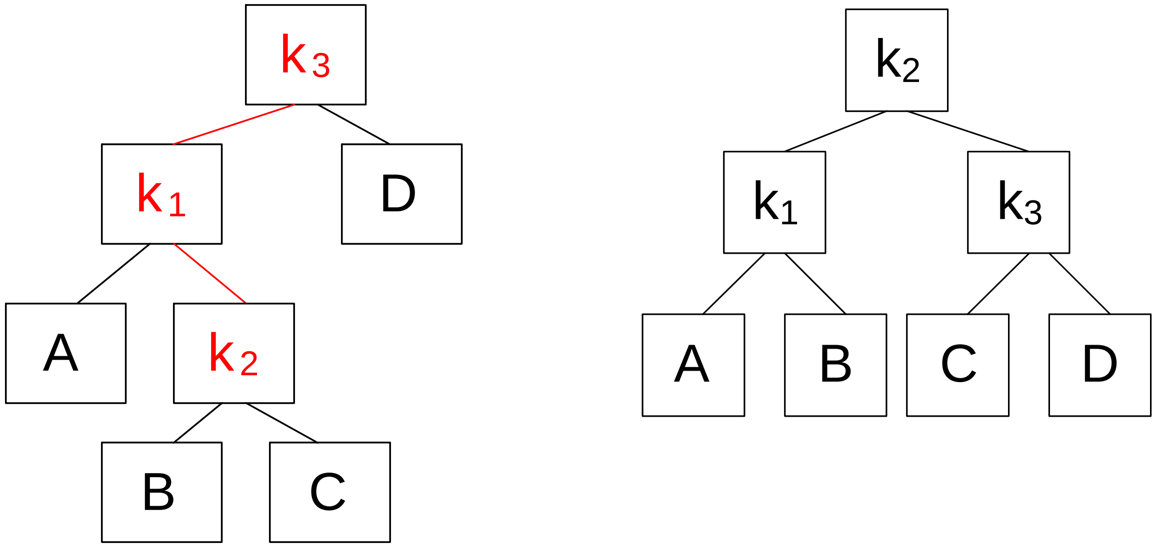
- AVL Trees
  - Case 2: insertion into the right subtree of the left child of k3
  - double_left_child(k3)
    rotate_right_child(k3^.left)
    rotate_left_child(k3)



  - A double rotation rebalances the tree.

AVL Trees

- AVL Trees
  - Case 2: insertion into the right subtree of the left child of k3
  - double_left_child(k3)
    rotate_right_child(k3^.left)
    rotate_left_child(k3)



  - A double rotation rebalances the tree.

AVL Trees

- Deletion from AVL-Trees
  - Unlike insertion, deletion can seriously unbalance AVL-Trees
    - A single (or double) rotation may not fix it up
  - We can often get away with a cheat
    - “Lazy deletion”
    - Don’t delete the node, just flag it
    - Re-use the node if we can at a later insertion

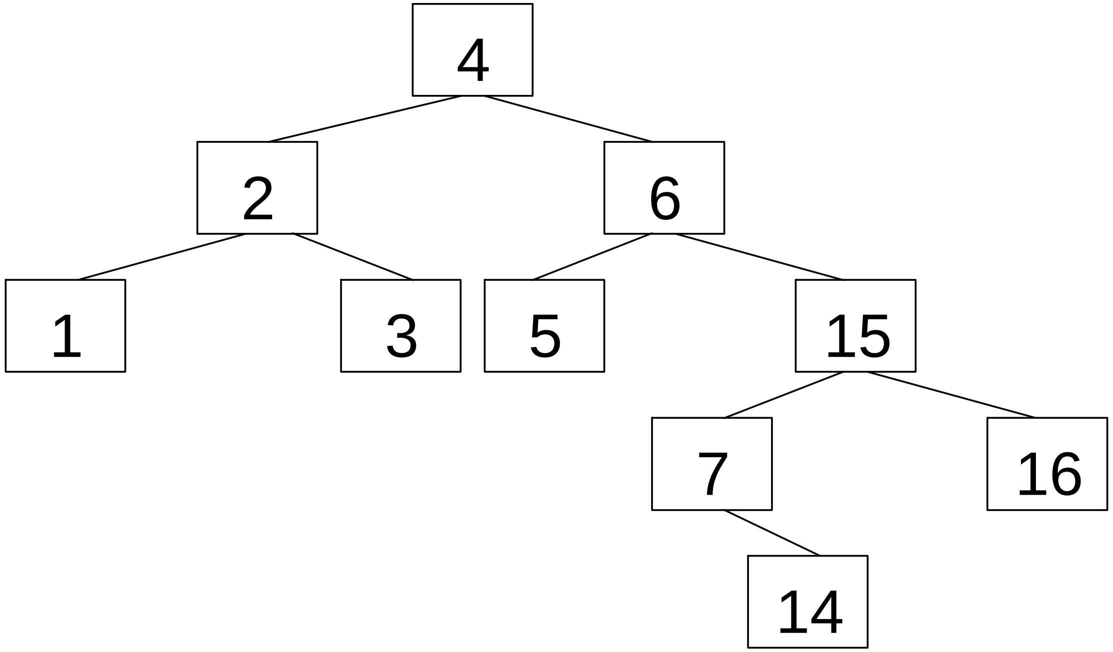
AVL Trees

An algorithm for doing AVL rebalances by hand after an insertion

1. Find the unbalanced node closest to the insertion point (by labeling with heights)
2. Determine whether the insertion occurred beneath the left or right child of the unbalanced node
3. Highlight the corresponding left or right edge
4. Determine whether the insertion occurred in the left or right subtree of the child node
5.If the direction in step 4 was different from the direction in step 3, highlight the corresponding right or left edge.
6. Do a single or double rotation as indicated by the highlighted edges

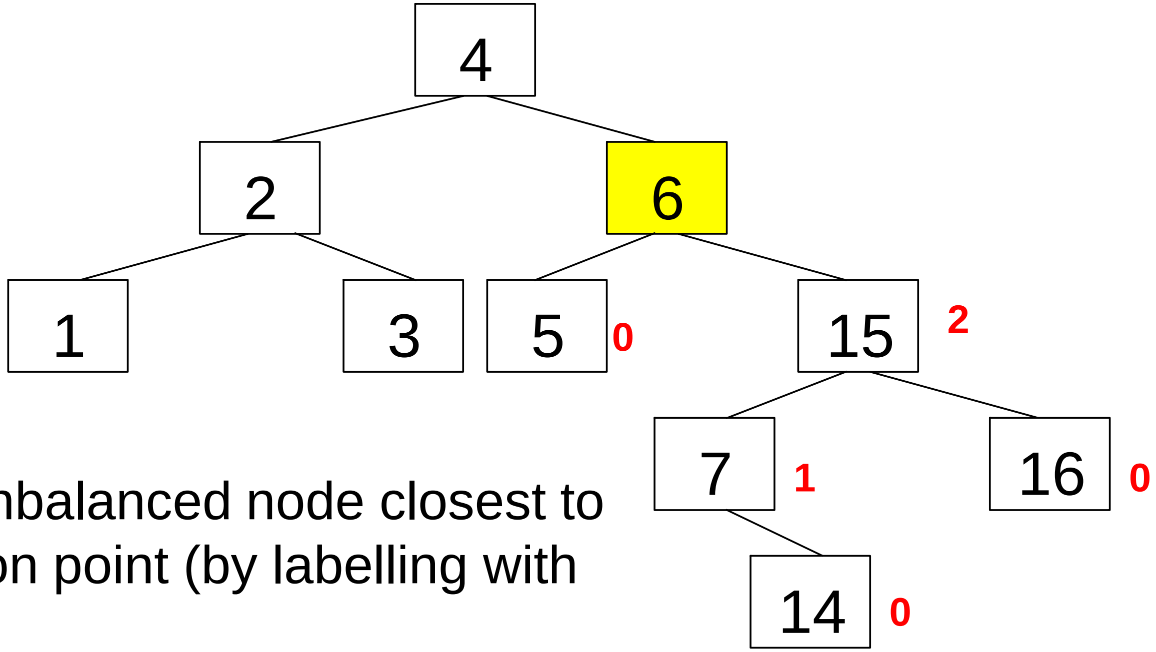
AVL Trees

- AVL Trees
  - An example: insert 14



AVL Trees

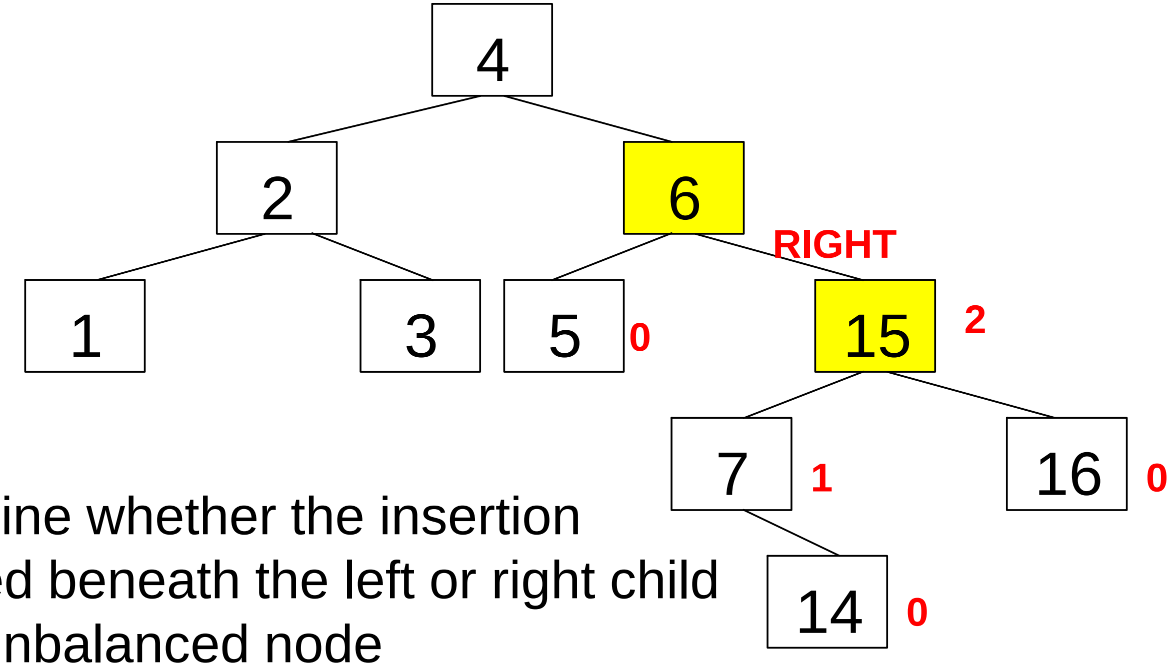
- AVL Trees
  - An example: insert 14



Find the unbalanced node closest to the insertion point (by labelling with heights)

AVL Trees

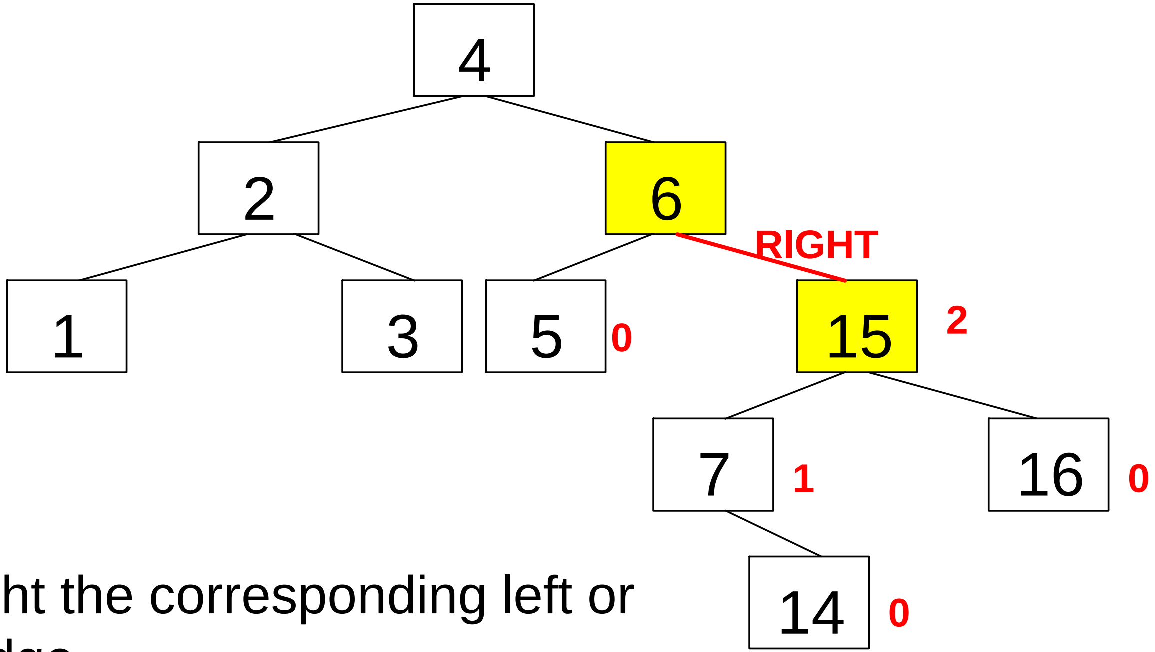
- AVL Trees
  - An example: insert 14



Determine whether the insertion occurred beneath the left or right child of the unbalanced node

AVL Trees

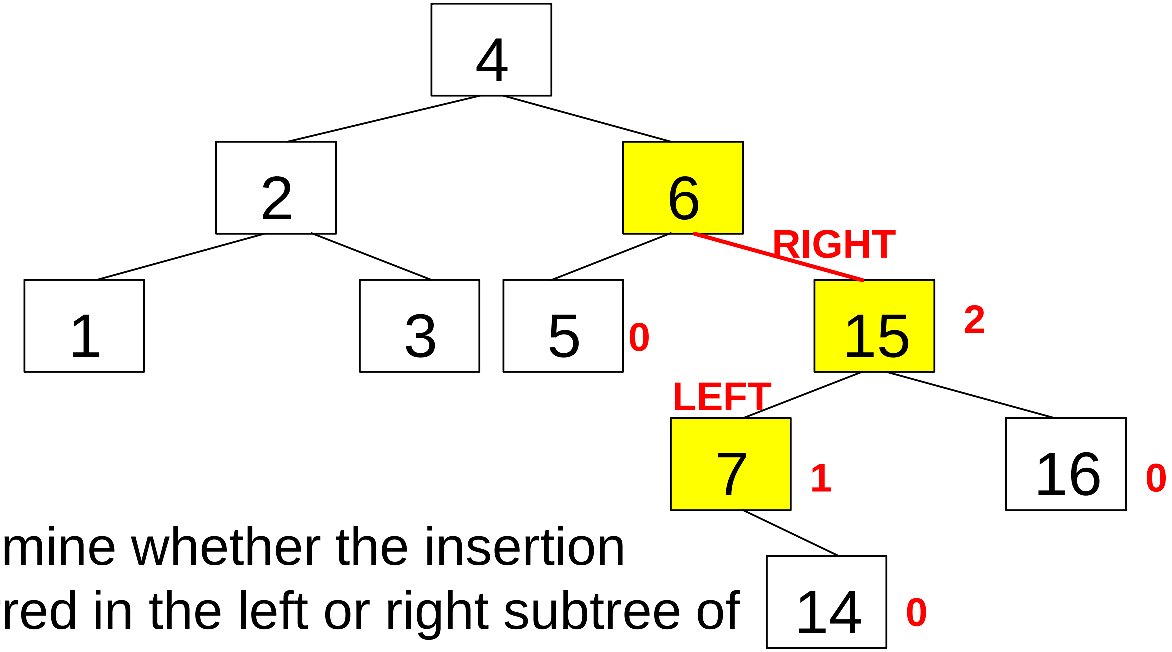
- AVL Trees
  - An example: insert 14



Highlight the corresponding left or right edge

AVL Trees

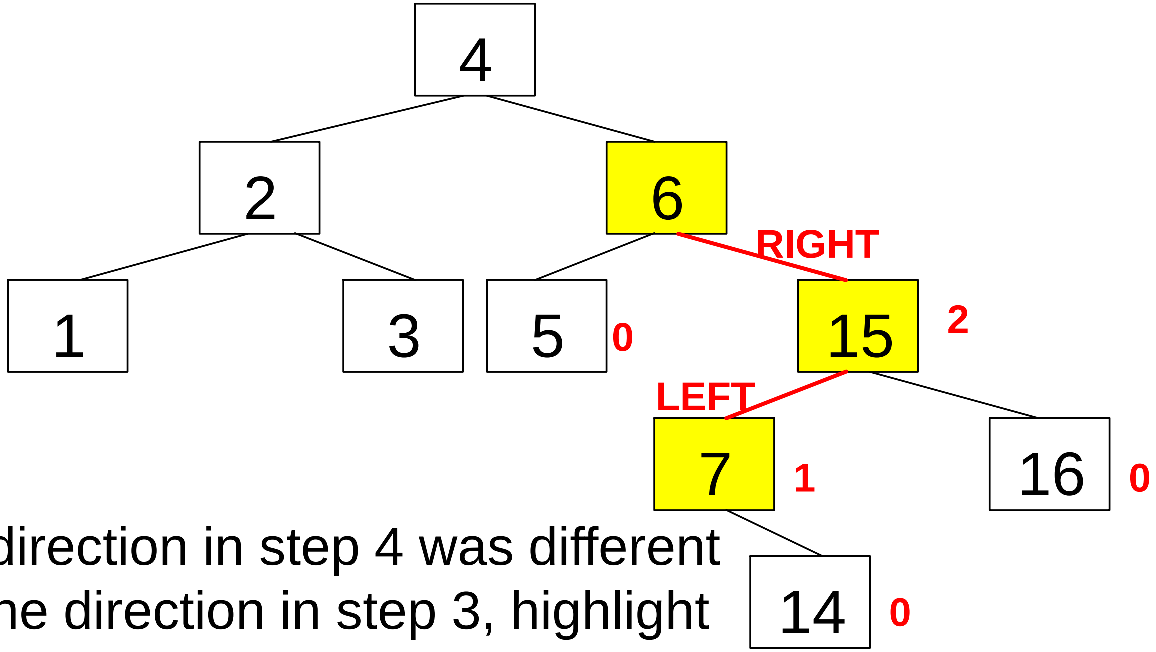
- AVL Trees
  - An example: insert 14



Determine whether the insertion occurred in the left or right subtree of the child node

AVL Trees

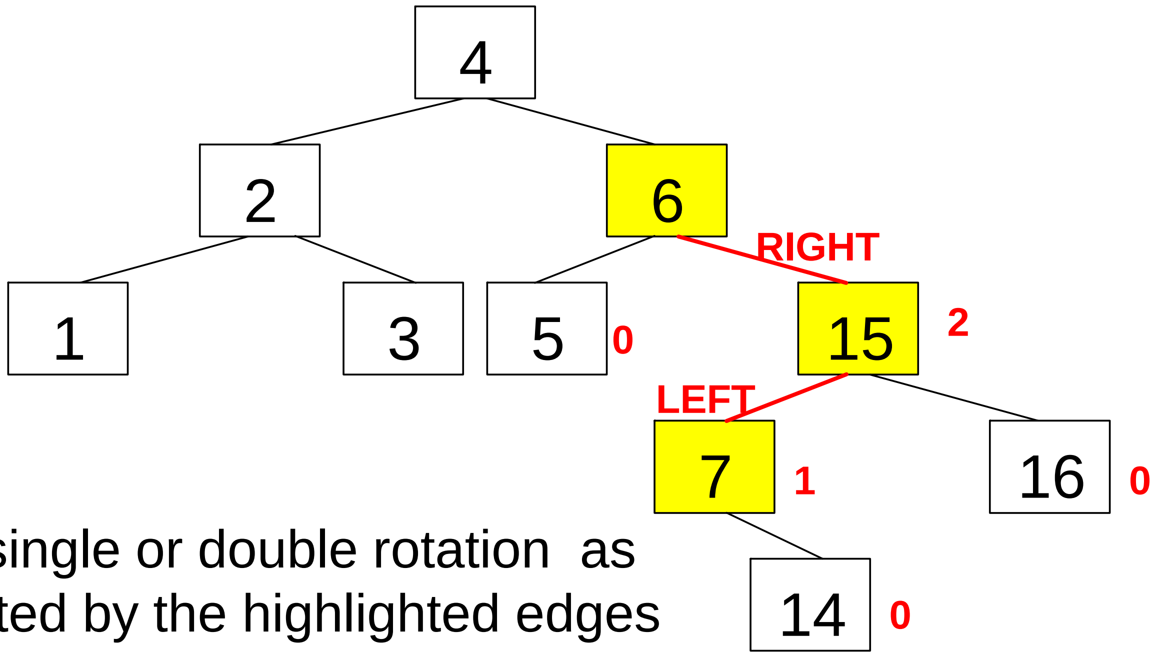
- AVL Trees
  - An example: insert 14



If the direction in step 4 was different from the direction in step 3, highlight the corresponding right or left edge.

AVL Trees

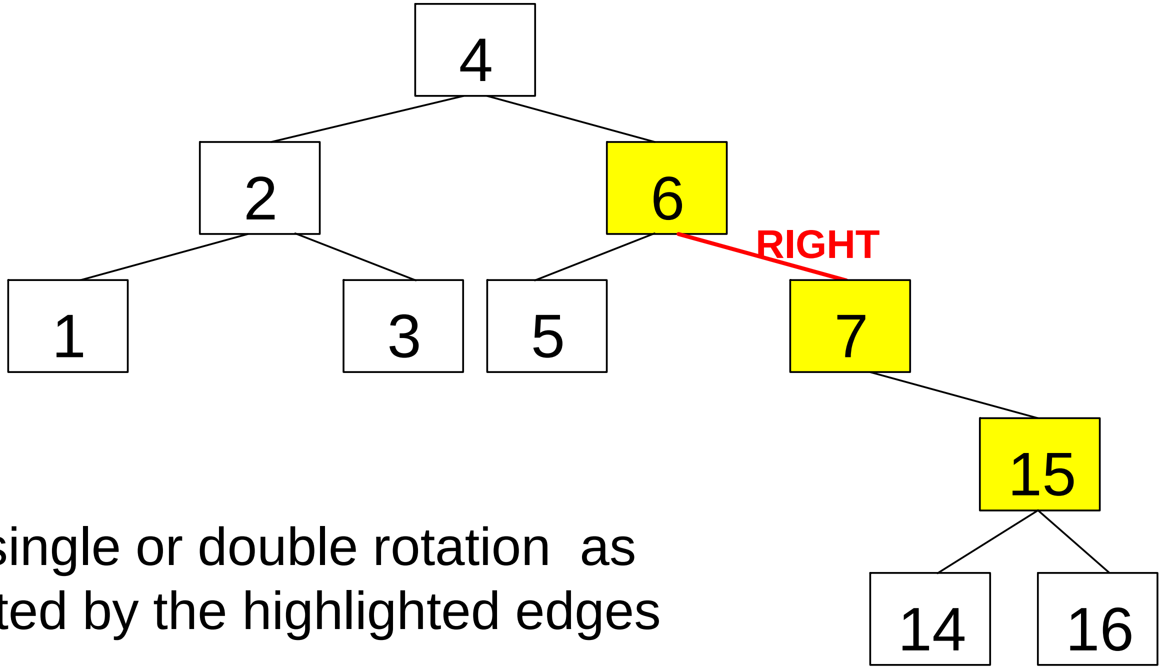
- AVL Trees
  - An example: insert 14



Do a single or double rotation as indicated by the highlighted edges

AVL Trees

- AVL Trees
  - An example: insert 14

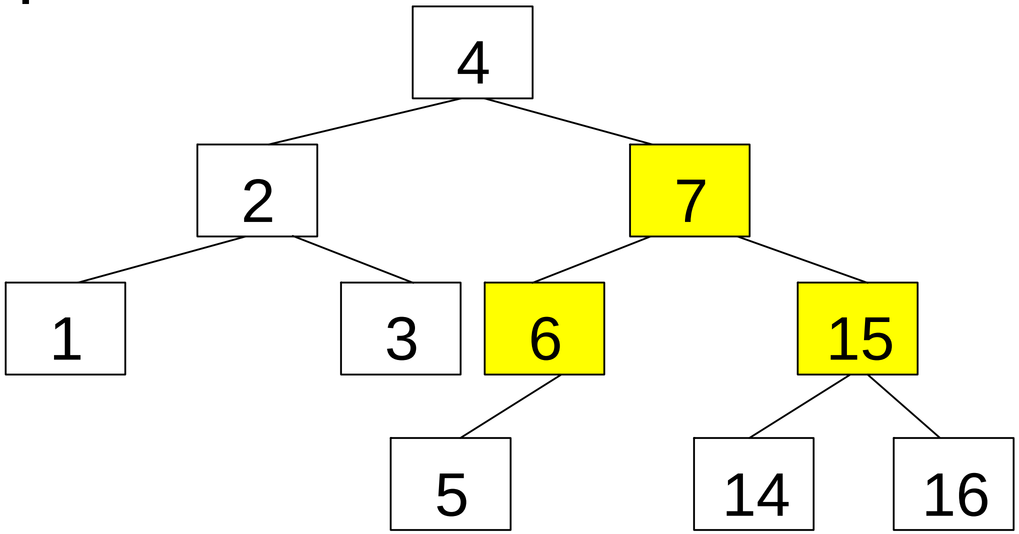


Do a single or double rotation as indicated by the highlighted edges

AVL Trees

- AVL Trees

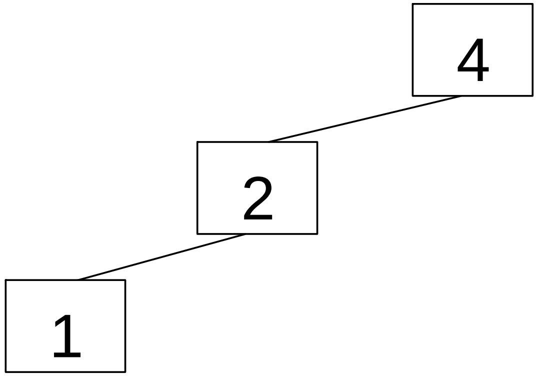
An example: insert 14



Do a single or double rotation as
indicated by the highlighted edges

AVL Trees

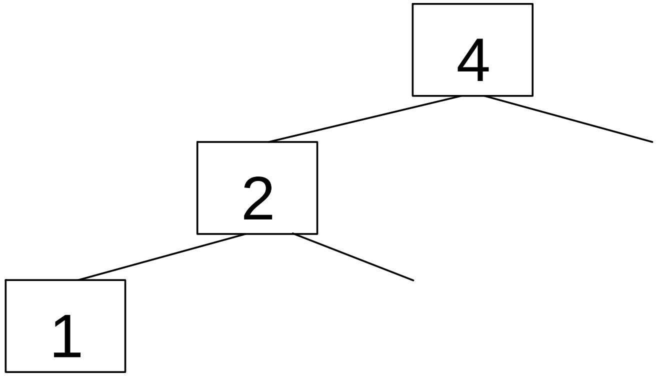
- AVL Trees
  - An example: insert 1



Make sure you get the correct unbalanced node in these situations

AVL Trees

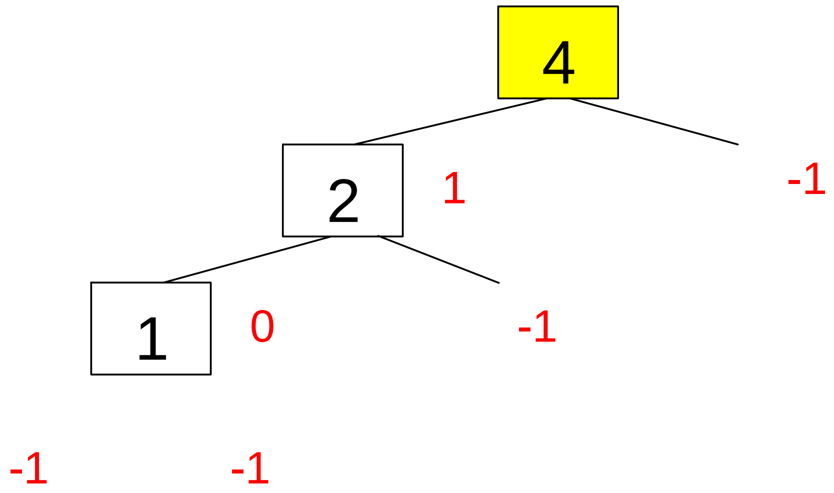
- AVL Trees
  - An example: insert 1



Make sure you get the correct unbalanced node in these situations

AVL Trees

- AVL Trees
  - An example: insert 1



By treating nil pointers as having height -1

Tutorial : Implement virtual functions
for three types of traversal

Homework: : Implement virtual functions
for building AVL tree
for storing a list of numbers