MCS

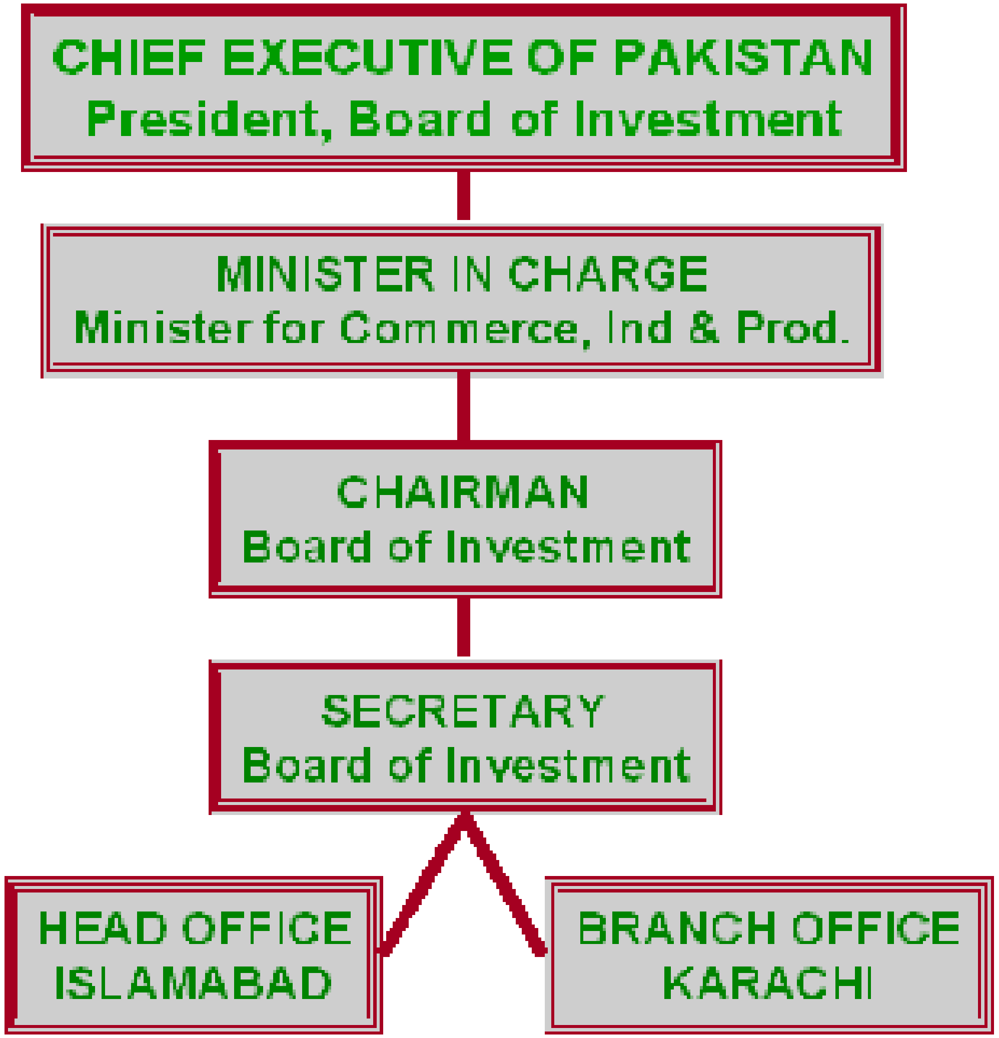
Tree Terminology

Introduction

- Data structure such as Arrays, Stacks, Linked List and Queues are linear data structure. Elements are arranged in linear manner i.e. one after another.
- Tree is a non-linear data structure.
- Tree imposes a ***Hierarchical*** structure, on a collection of items.
- Several Practical applications
  - Organization charts.
  - Family hierarchy
  - Representation of algebraic expression

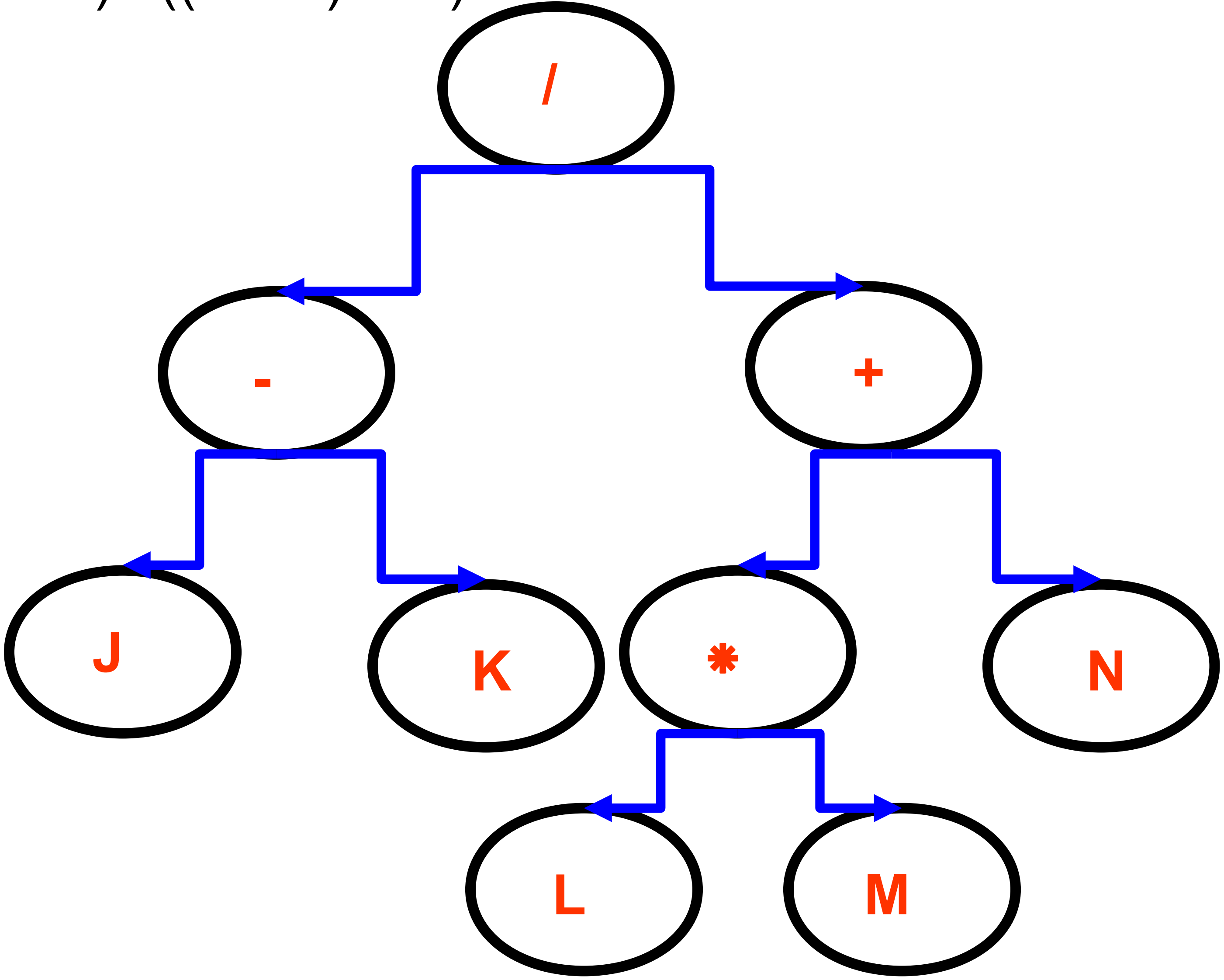
Introduction

- Organizational Chart



Introduction

- Representation of algebraic expression
- Z=(J - K) / ((L * M) + N)



Tree Definition

- A tree is a collection of nodes
  - The collection can be empty
  - If not empty, a tree consists of a distinguished node **R** (the *root*), and zero or more nonempty ***subtrees*** T1, T2, ..., Tk, each of whose roots are connected by a directed ***edge*** from **R.**

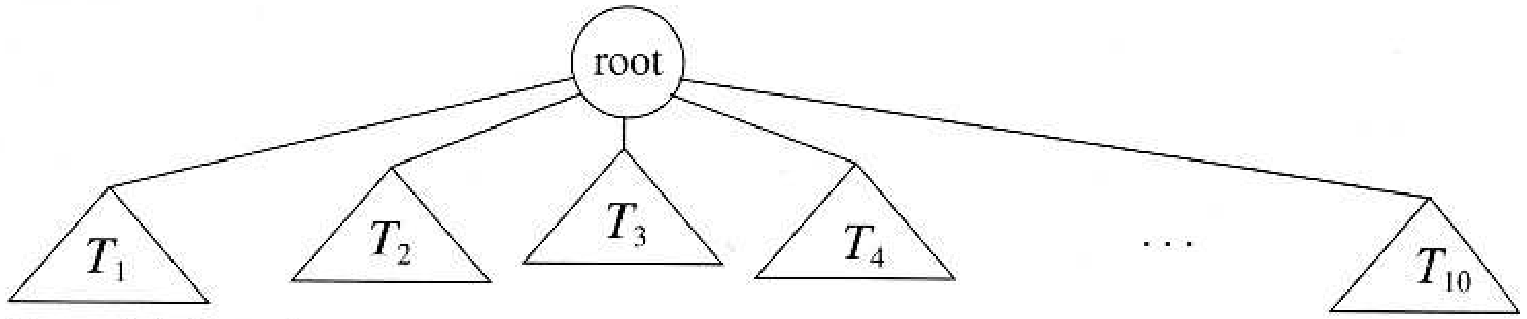


Figure 4.1 Generic tree

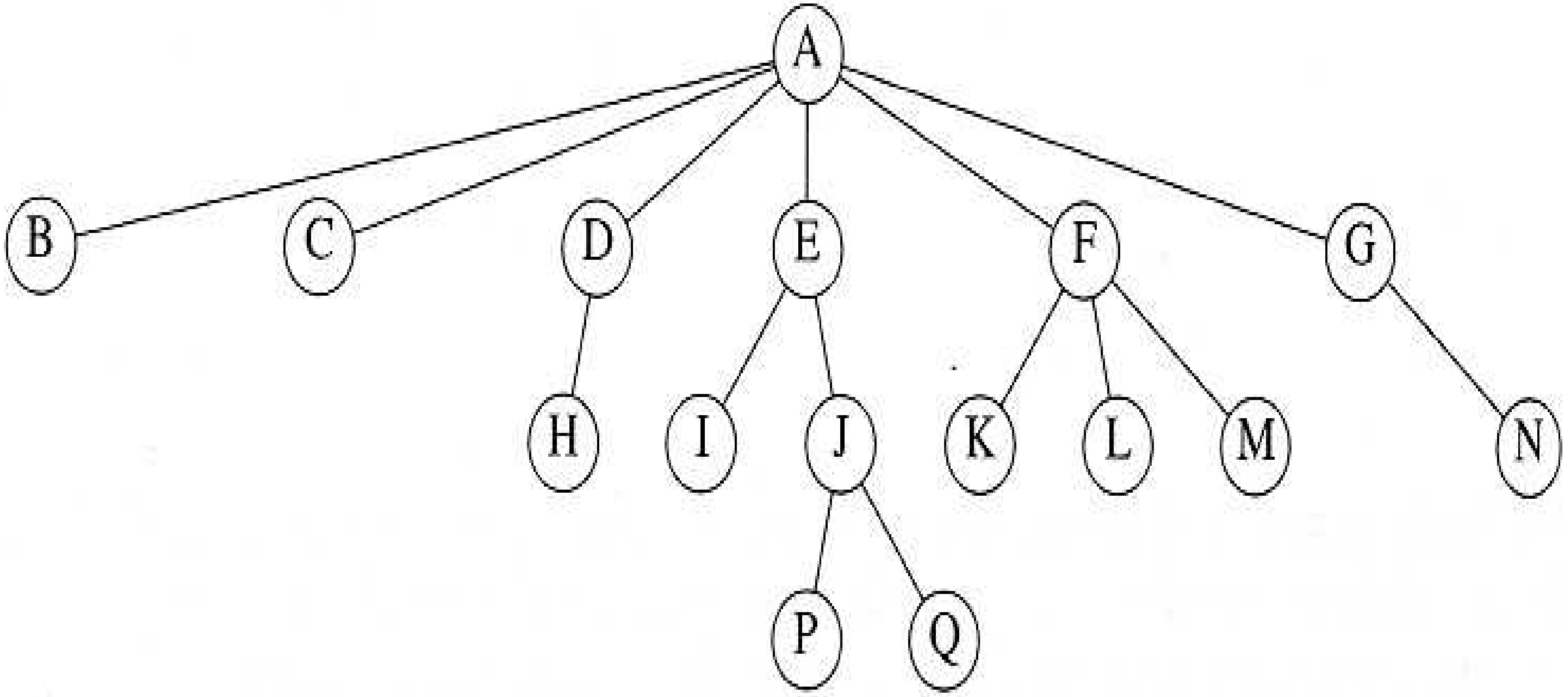
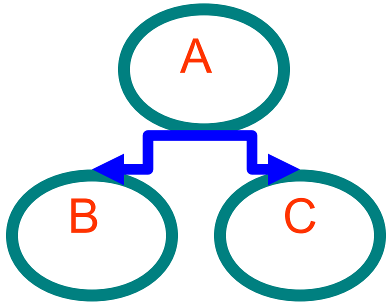


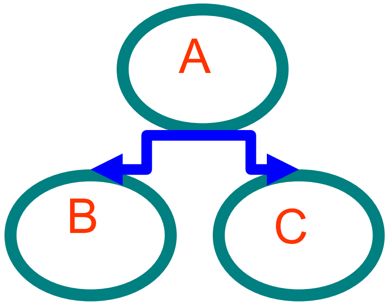
Figure 4.2 A tree

Tree Terminologies

- **Root**
  - It is the mother node of a tree structure. This tree does not have parent. It is the first node in hierarchical arrangement.
- **Node**
  - The node of a tree stores the data and its role is the same as in the linked list. Nodes are connected by the means of links with other nodes.
- **Parent**
  - It is the immediate predecessor of a node. In the figure A is the parent of B and C.



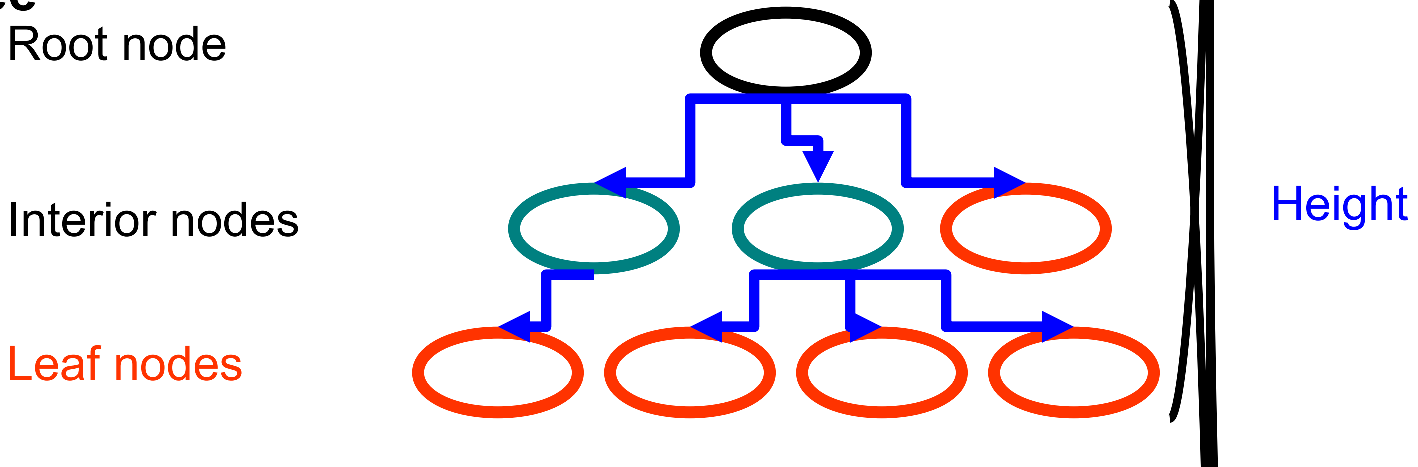
Tree Terminologies



- **Child**
  - When a predecessor of a node is parent then all successor nodes are called child nodes. In the figure B and C are the child nodes of A
- **Link / Edge**
  - An edge connects the two nodes. The line drawn from one node to other node is called edge / link. Link is nothing but a pointer to node in a tree structure.
- **Leaf**
  - This node is located at the end of the tree. It does not have any child hence it is called leaf node.

Tree Terminologies

- **Level**
  - **Level is the rank of tree hierarchy. The whole tree structured is leveled. The level of the root node is always at 0. the immediate children of root are at level 1 and their children are at level 2 and so no.**
- **Height**
  - **The highest number of nodes that is possible in a way starting from the first node (ROOT) to a leaf node is called the height of tree. The formula for finding the height of a tree** $h \quad i_{\max} \quad 1$ **, where h is the height and l is the max level of the tree**



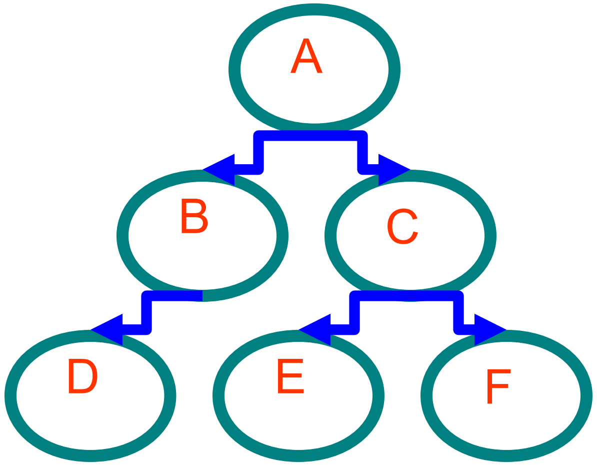
MCS

Tree Terminologies

- **Sibling**
  - **The child node of same parent are called sibling. They are also called brother nodes.**
- **Degree of a Node**
  - **The maximum number of children that exists for a node is called as degree of a node.**
- **Terminal Node**
  - **The node with degree zero is called terminal node or leaf.**
- **Path length.**
  - **Is the number of successive edges from source node to destination node.**
- **Ancestor and descendant**
  - **Proper ancestor and proper descendant**

Tree Terminologies

- **Depth**
  - **Depth of a binary tree is the maximum level of any leaf of a tree.**
- **Forest**
  - **It is a group of disjoint trees. If we remove a root node from a tree then it becomes the forest. In the following example, if we remove a root A then two disjoint sub-trees will be observed. They are left sub-tree B and right sub-tree C.**

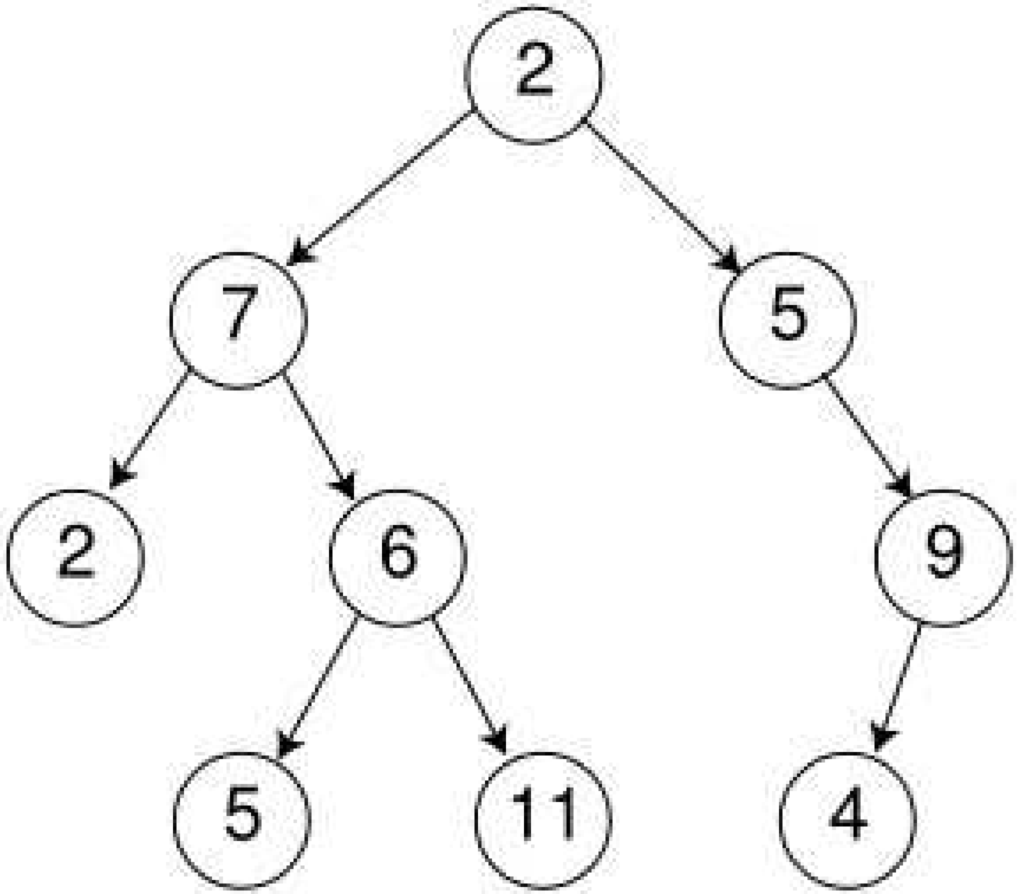


MCS

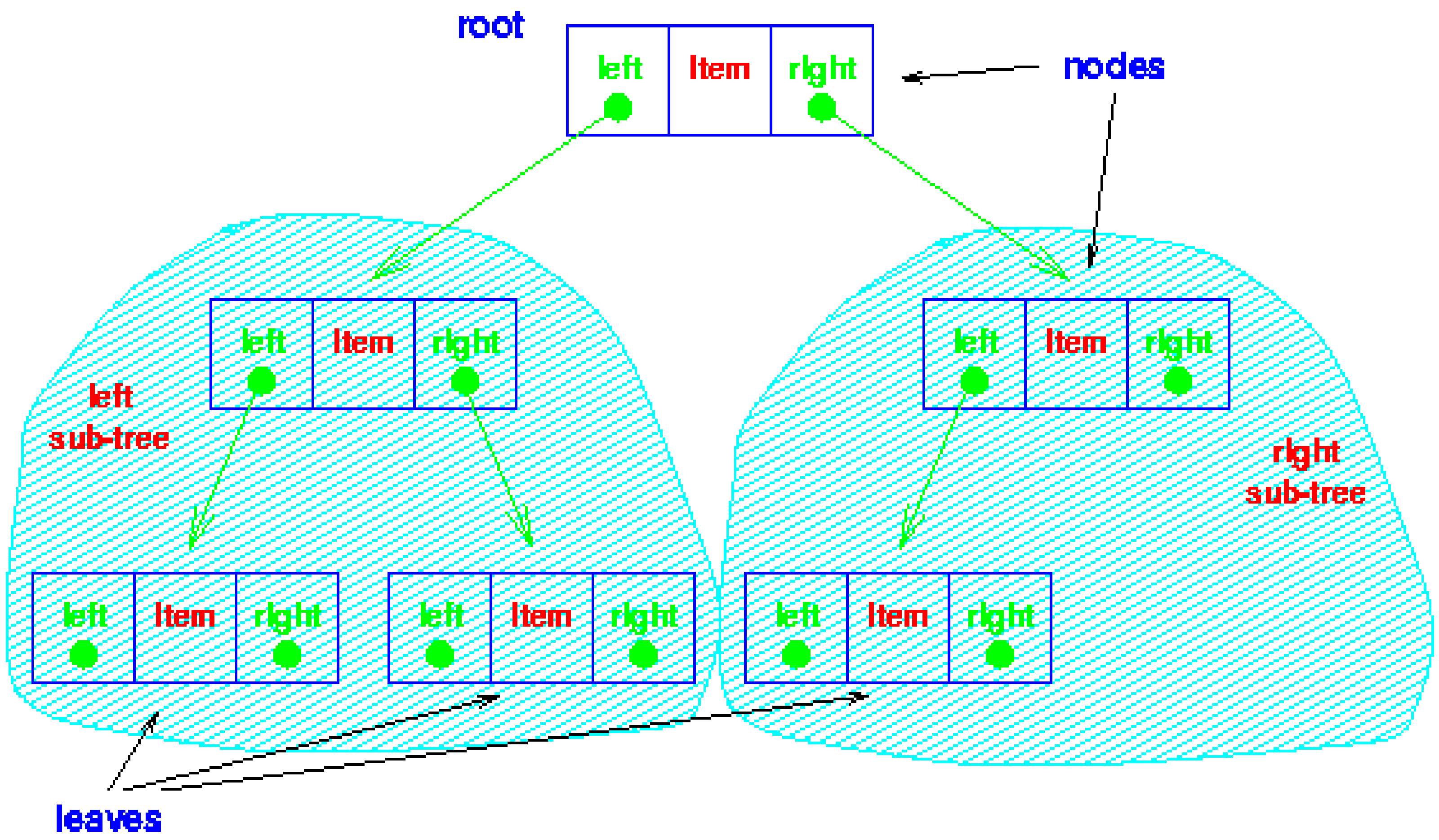
Binary Trees

- The simplest form of tree is a **binary tree**. A binary tree consists of
  - a *node* (called the **root** node) and
  - left and right sub-trees.
    Both the sub-trees are themselves binary trees
- We now have a *recursively defined data structure*.
- Also, a tree is binary if each node of it has a maximum of two branches i.e. a node of a binary tree can have maximum two children.

A Binary Tree



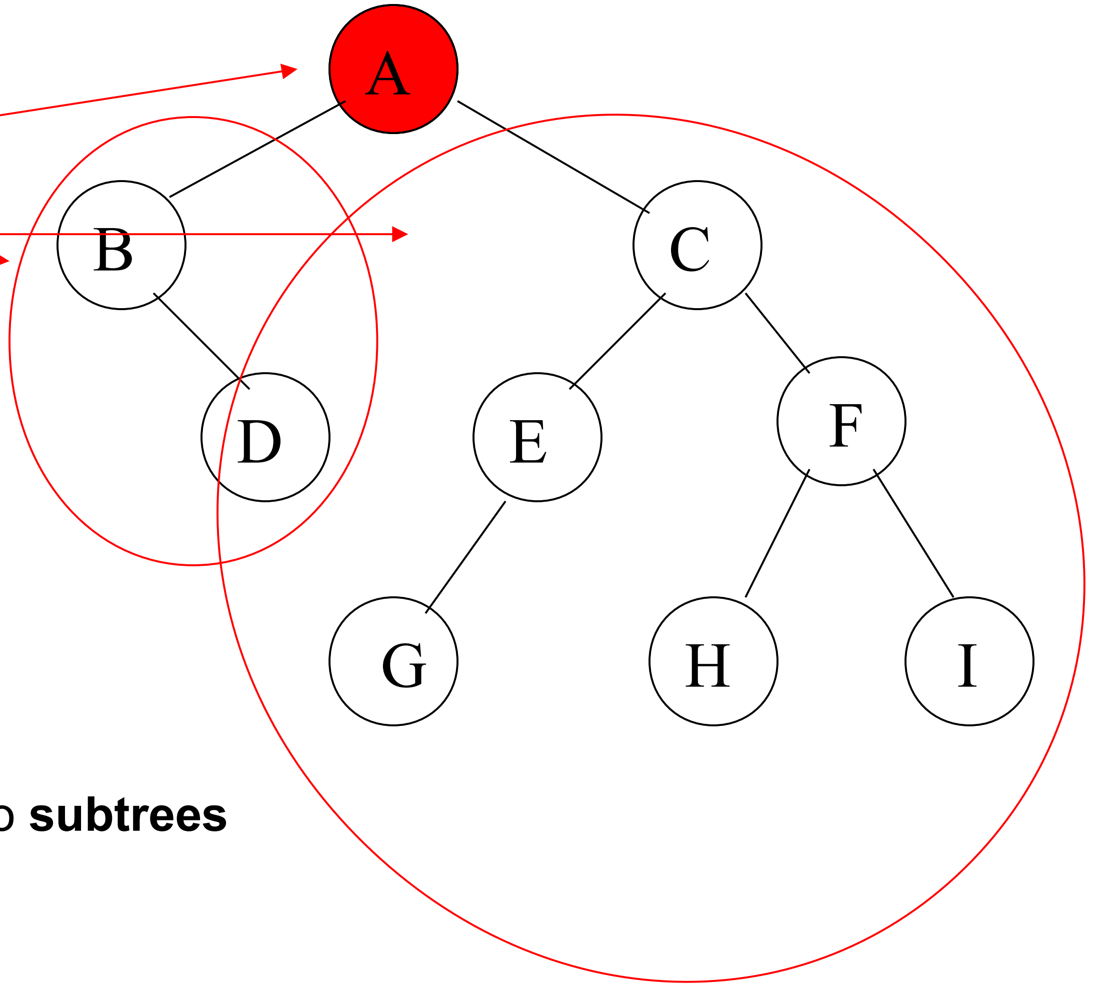
Binary Tree


root
left
Item
right
nodes
left sub-tree
right sub-tree
leaves

Notation

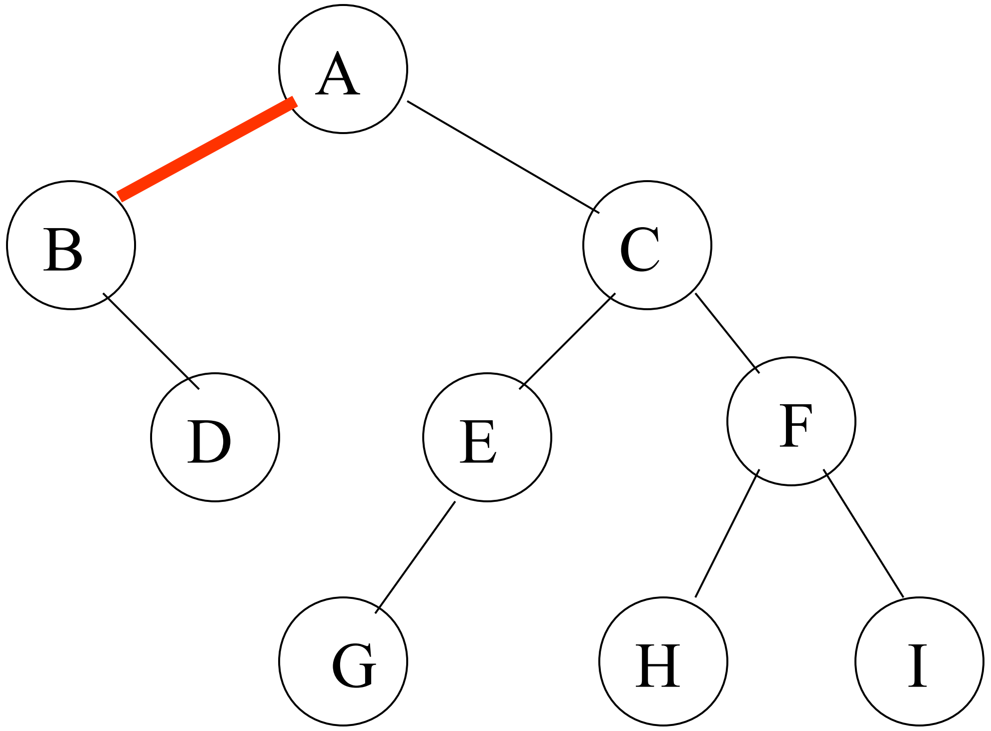
node

- root
  - left subtree
  - right subtree



It consists of a **root** and two **subtrees**

Notation

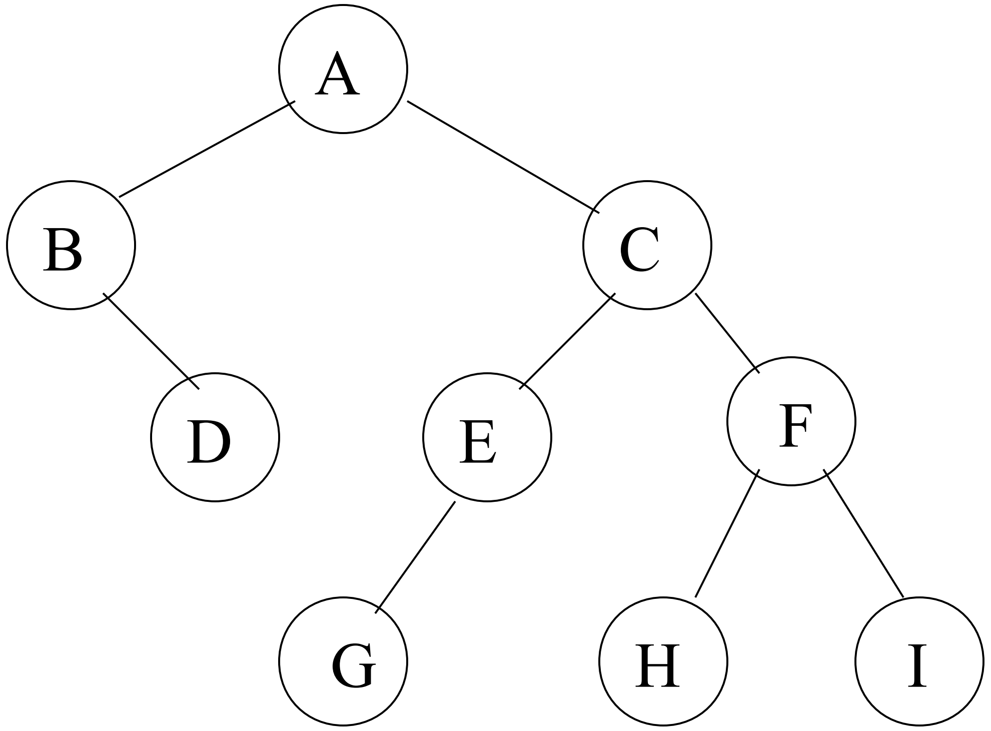


edge –

there is an **edge** from the root to its children

Notation

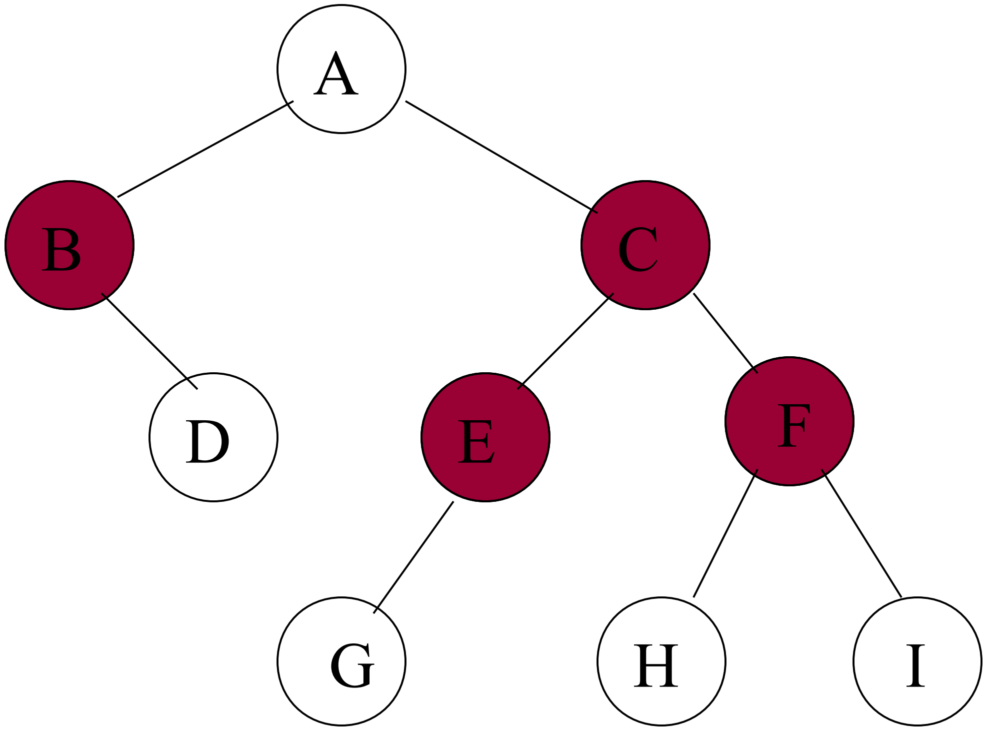
children



Notation

children

Who are node C's children?

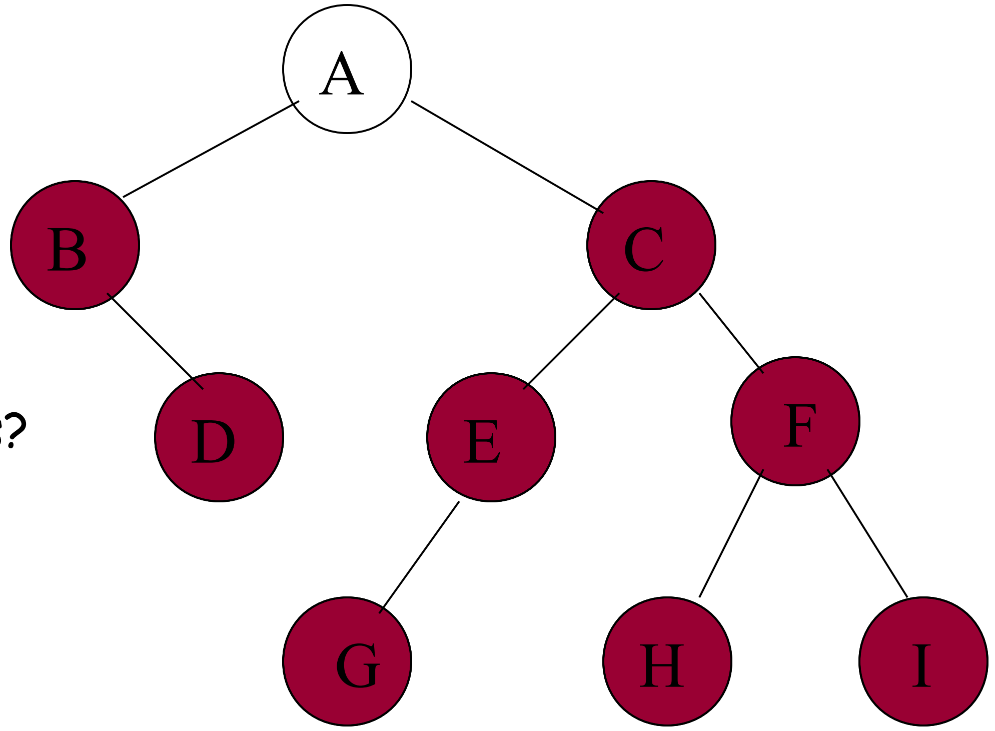


Who are node A's children?

Notation

descendants

Who are node C's descendants?

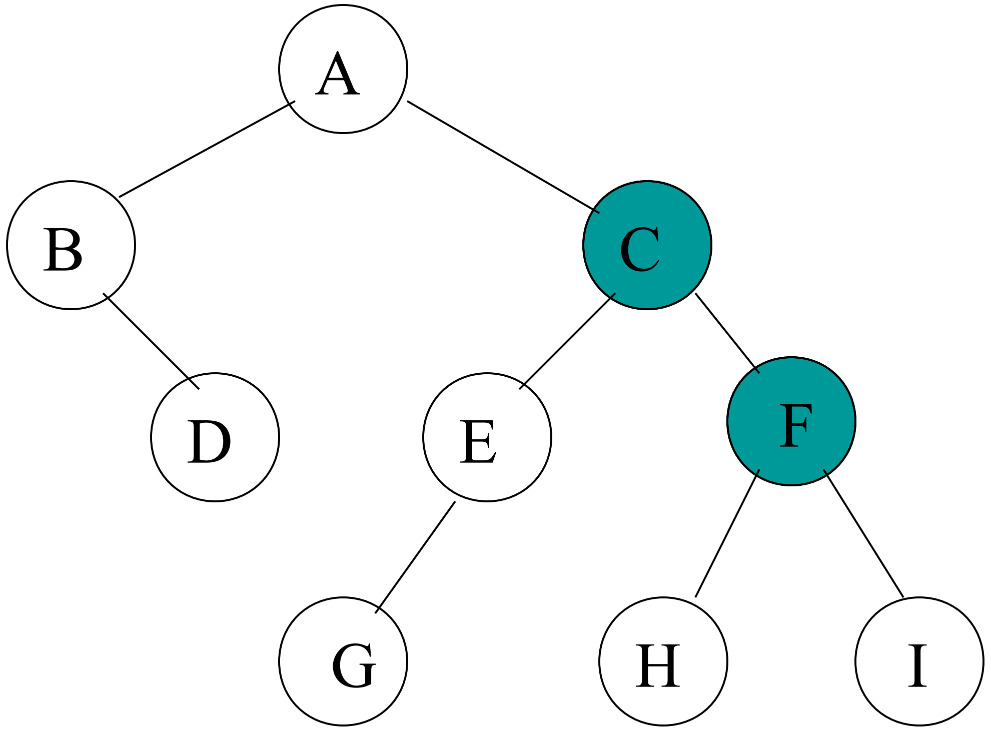


Who are node A's descendants?

Notation

parents

Who is node E's parent?

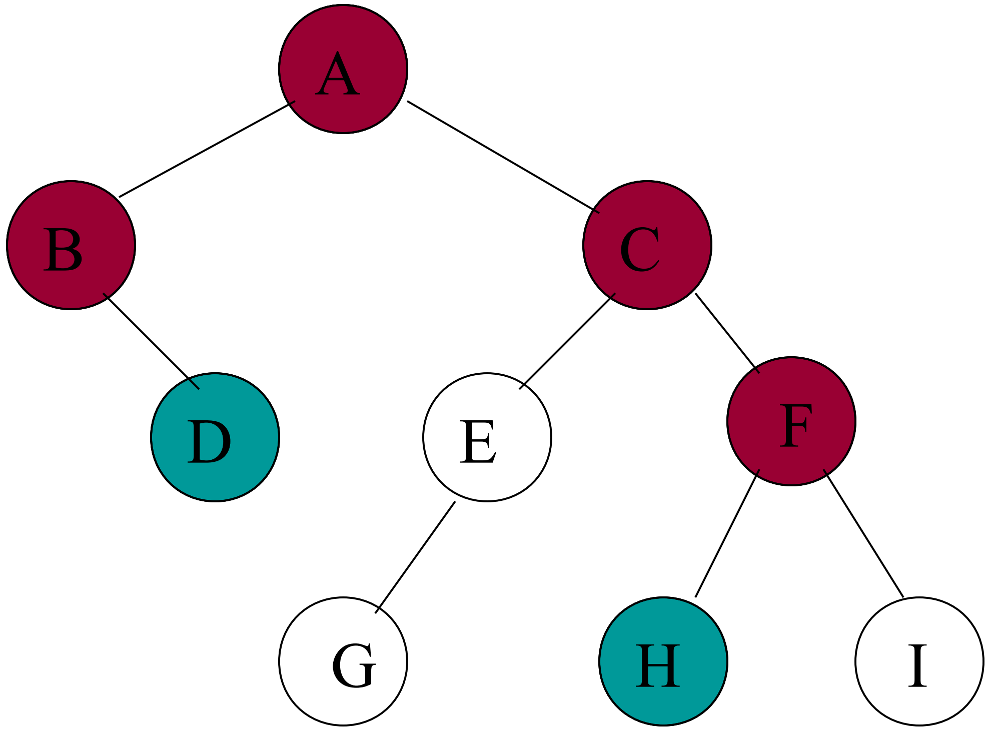


Who is node H's parent?

Notation

ancestors

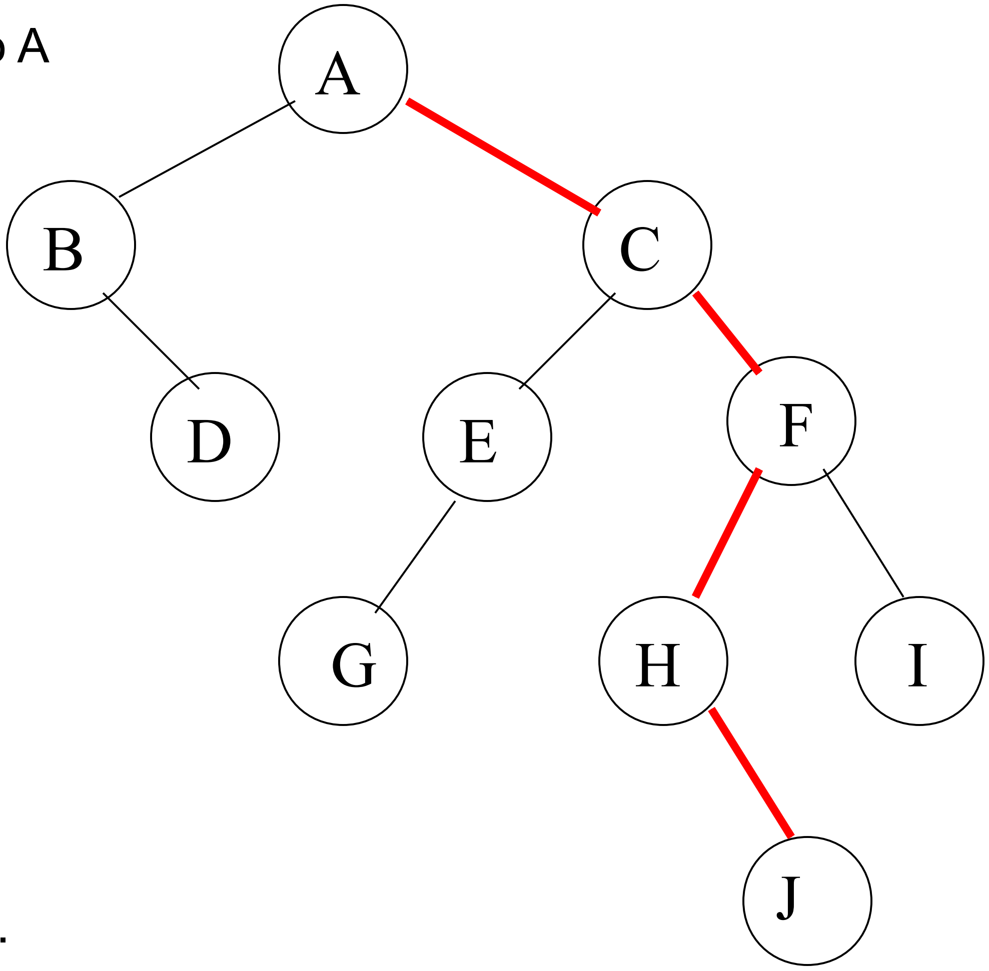
Who are node D's ancestors?



Who are node H's ancestors?

Notation

from J to A

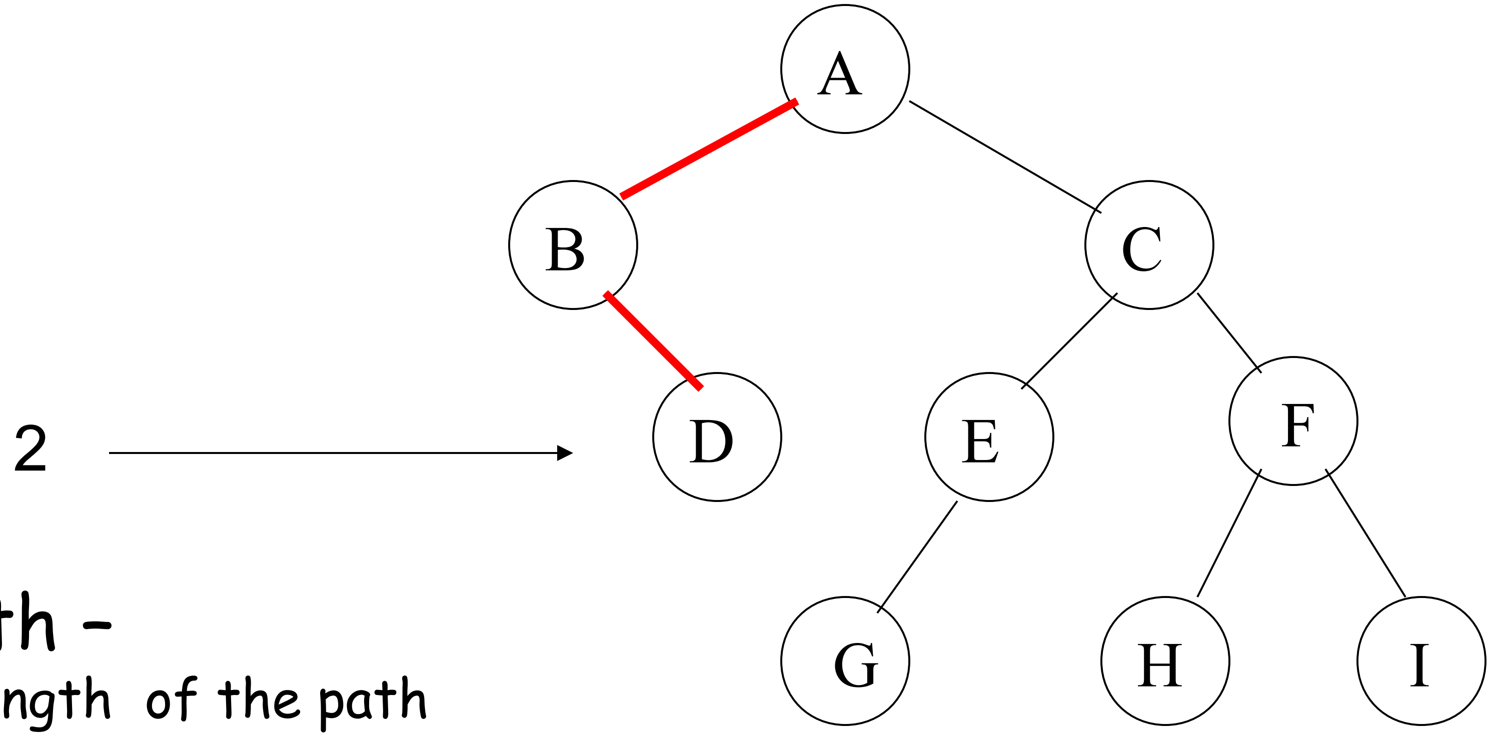


path –

If n1, n2,…nk is a sequence of nodes such that ni is the parent of ni+1, then that sequence is a **path.**

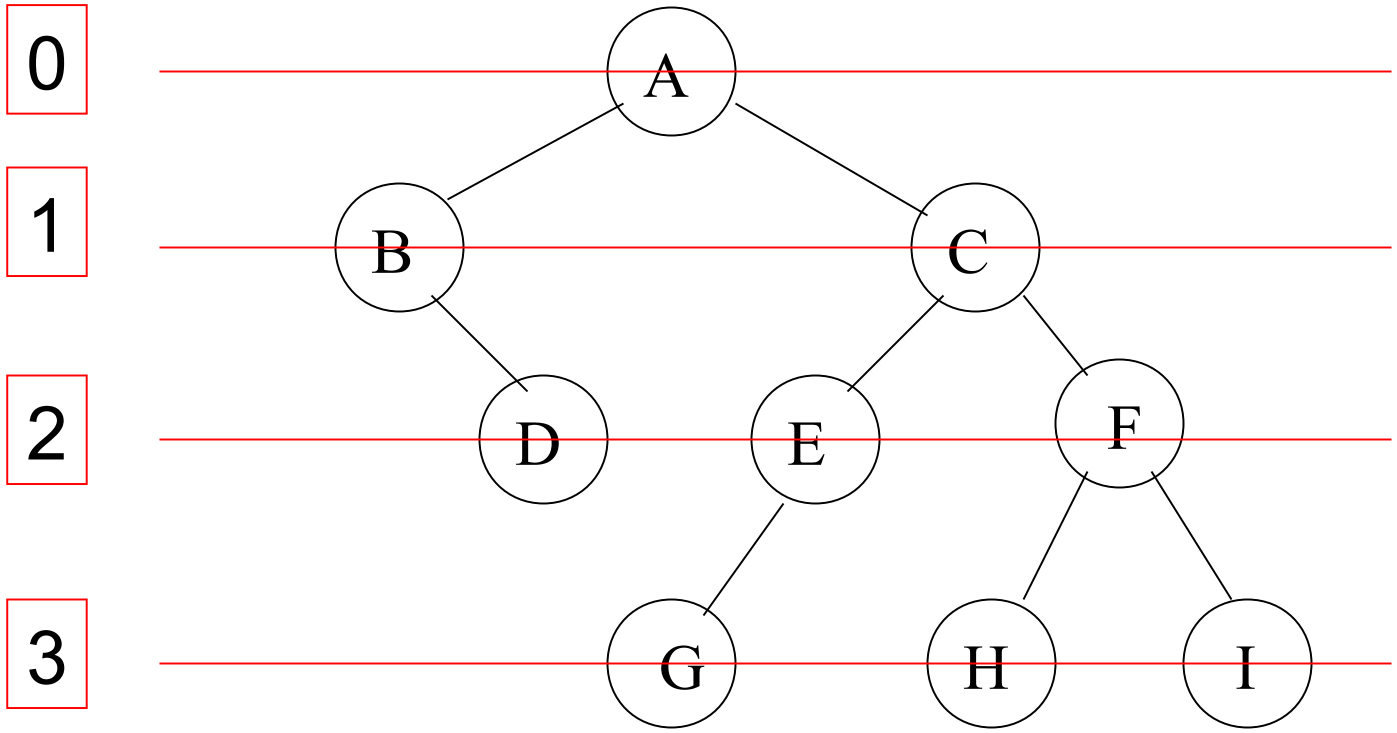
- The **length** of the path is k-1.

Notation



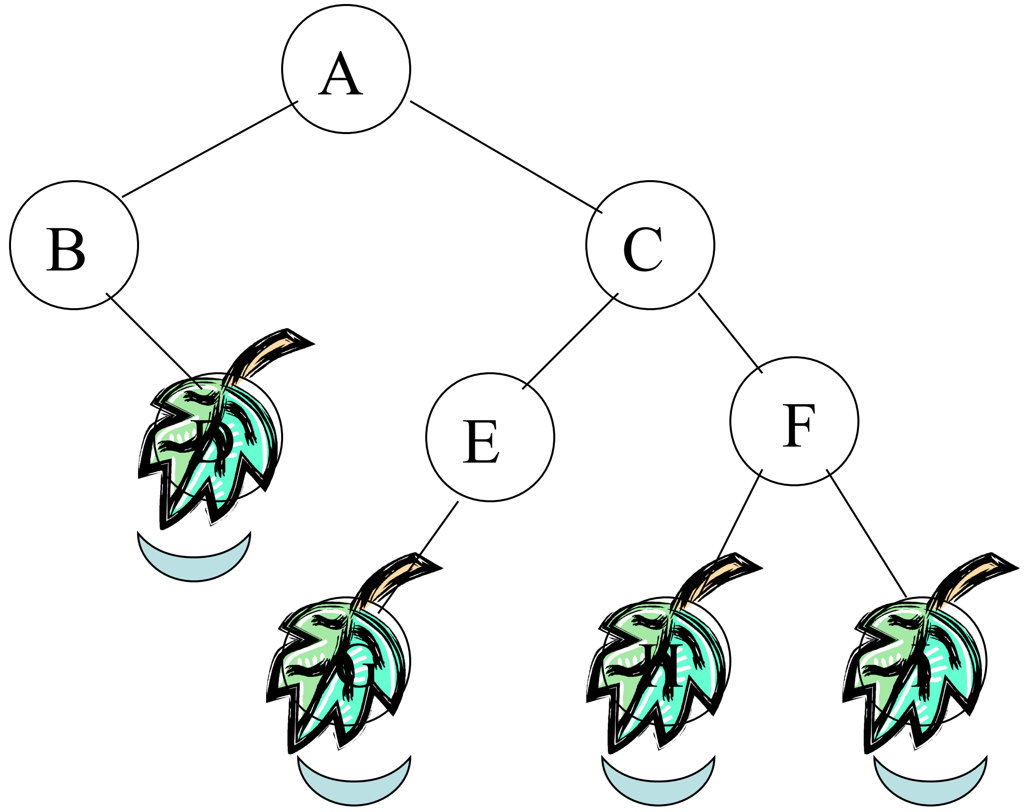
depth –
the length of the path from the root of the tree to the node

Notation



level –
all nodes of depth d are at level d in the tree

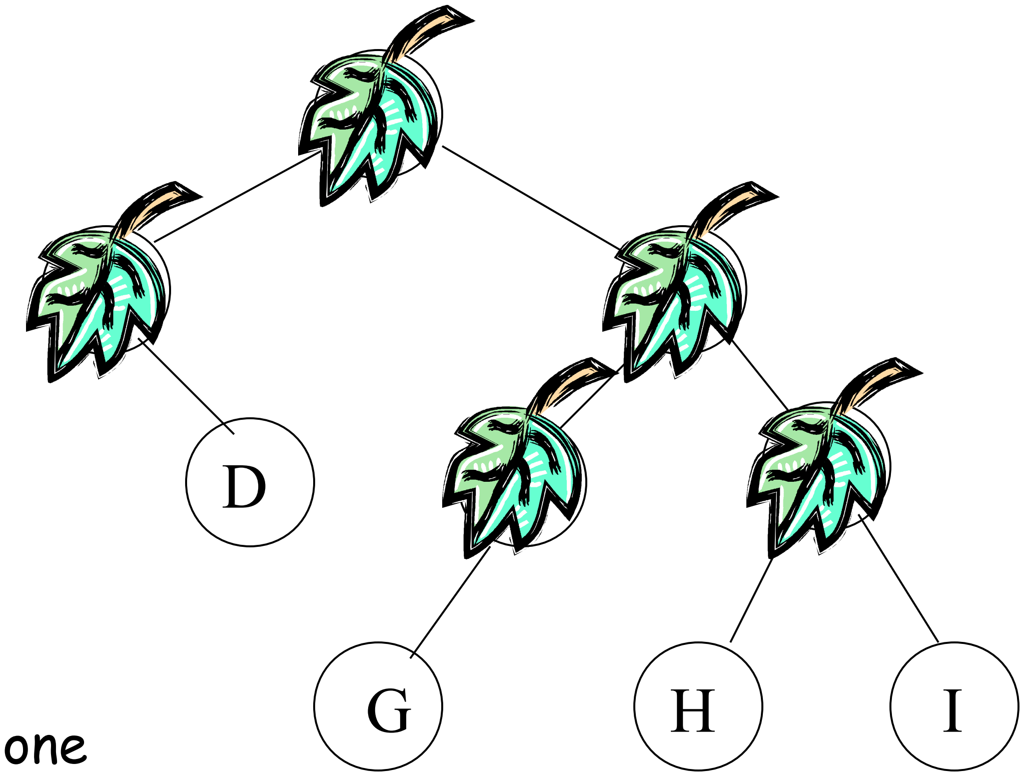
Notation



leaf node –

any node that has two empty children

Notation

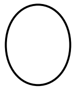


internal node –
any node that has at least one non-emptyChild
Or
An *internal node* of a *tree* is any *node* which has degree greater than one.

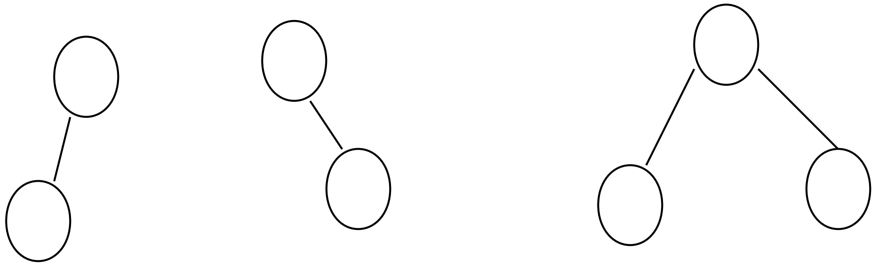
Binary Trees

Some Binary Trees

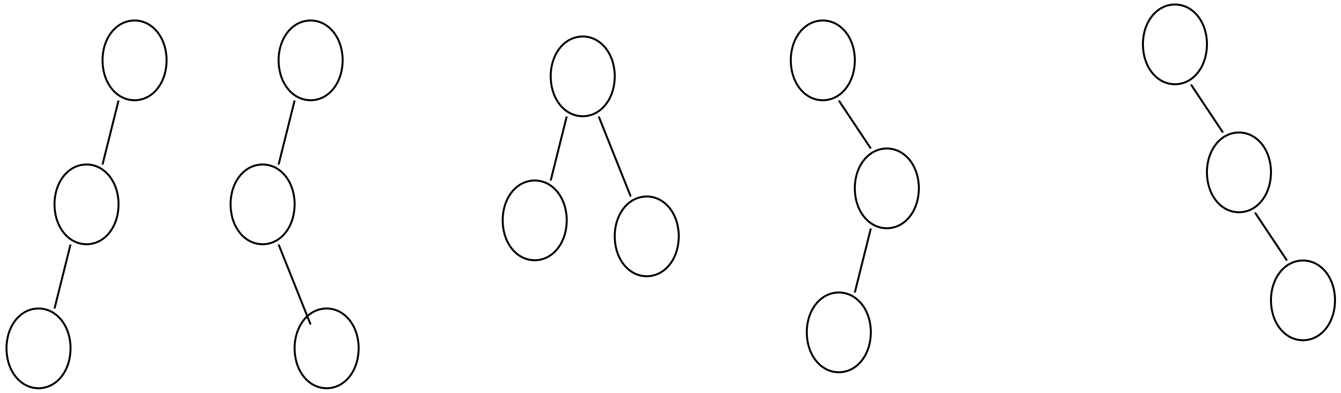
One node



Two nodes

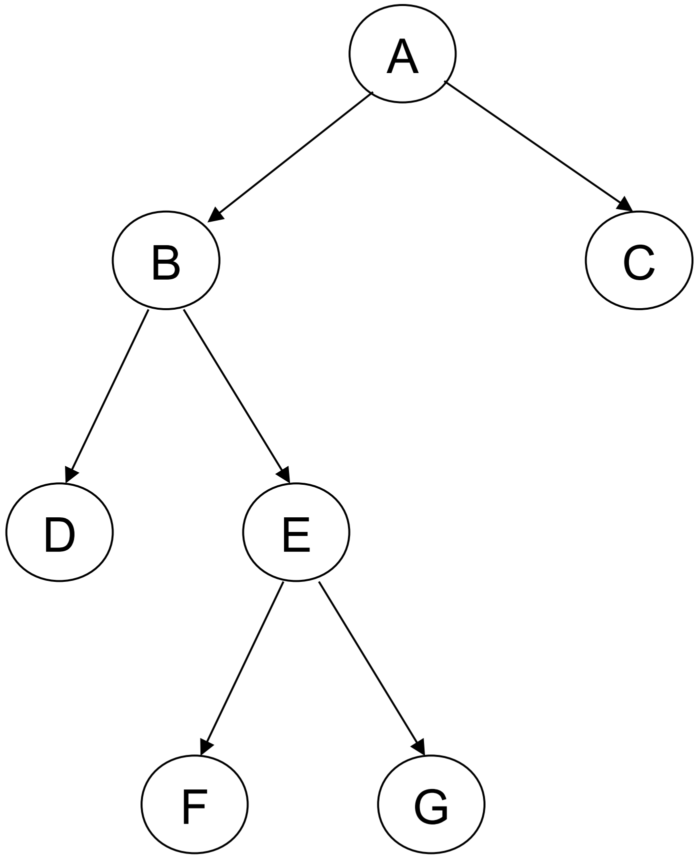


Three nodes



Strictly Binary Tree

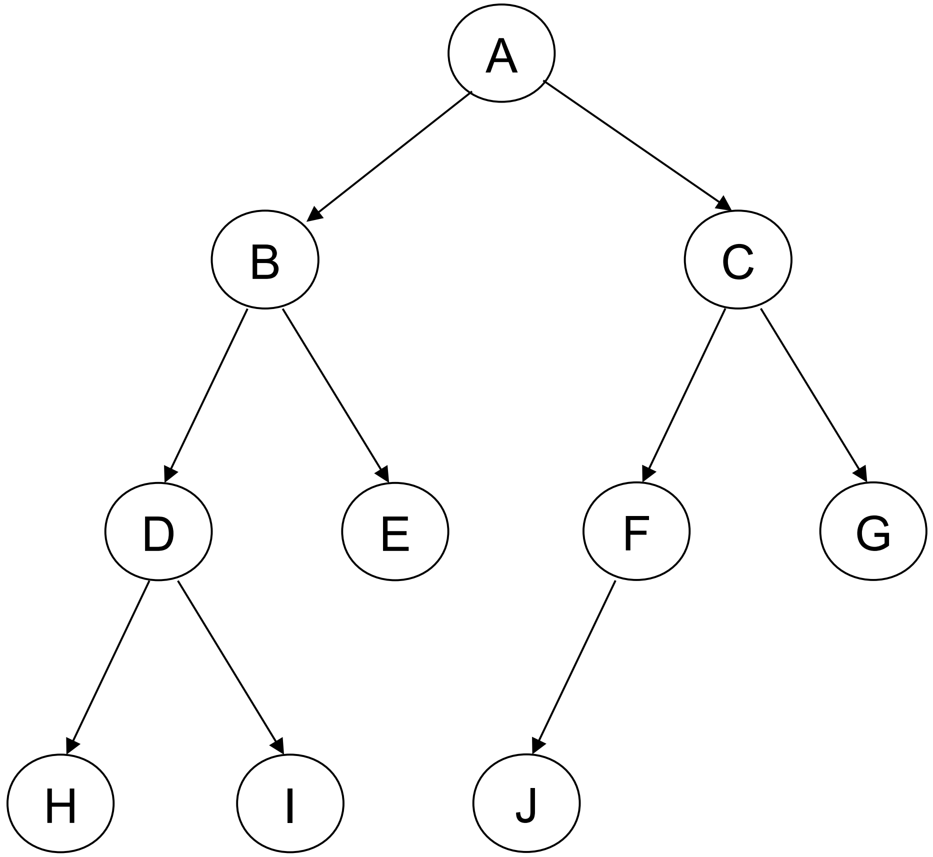
- When every non-leaf node in binary tree is filled with left and right sub-trees, the tree is called strictly binary tree.



Complete Binary Tree

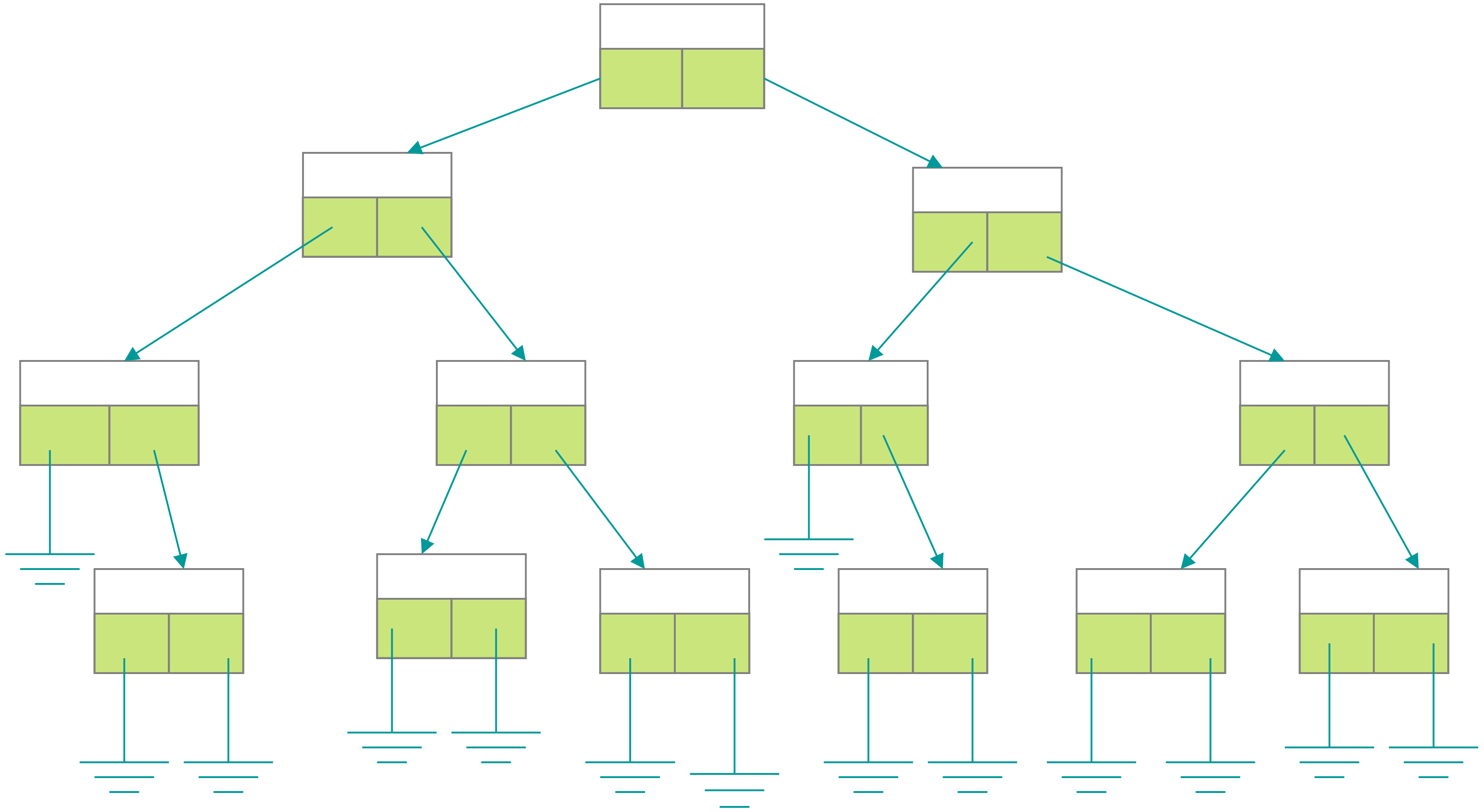
A ***Complete binary*** tree is:

- A tree in which each level of the tree is completely filled.
- Except, possibly the bottom level.



Dynamic Implementation of Binary Tree

Linked Implementation



Structure Definition of Binary Tree Using Dynamic Implementation

- The fundamental component of binary tree is node.
- In binary tree node should consist of three things.
  - **Data**
    - **Stores given values**
  - **Left child**
    - **is a link field and hold the address of its left node**
  - **Right child.**
    - **Is a link field and holds the address of its right node.**

```
struct node
{
    int data
    node *left_child;
    node *right_child;
};
```

Operations on Binary Tree

- **Create**
  - **Create an empty binary tree**
- **Empty**
  - **Return true when binary tree is empty else return false.**
- **Lchild**
  - **A pointer is returned to left child of a node, when a node is without left child, NULL pointer is returned.**
- **Rchild**
  - **A pointer is returned to right child of a node, when a node is without left child, NULL pointer is returned.**
- **Father/Parent**
  - **A pointer to father of a node is returned.**

Operations on Binary Tree

- **Sibling**
  - **A pointer to brother of the node is returned or else NULL pointer is returned.**
- **Tree Traversal**
  - **Inorder Traversal**
  - **Preorder Traversal**
  - **Postorder Traversal**
- **Insert**
  - **To insert a node**
- **Deletion**
  - **To delete a node**
- **Search**
  - **To search a given node**
- **Copy**
  - **Copy one tree into another.**

Example: Expression Trees

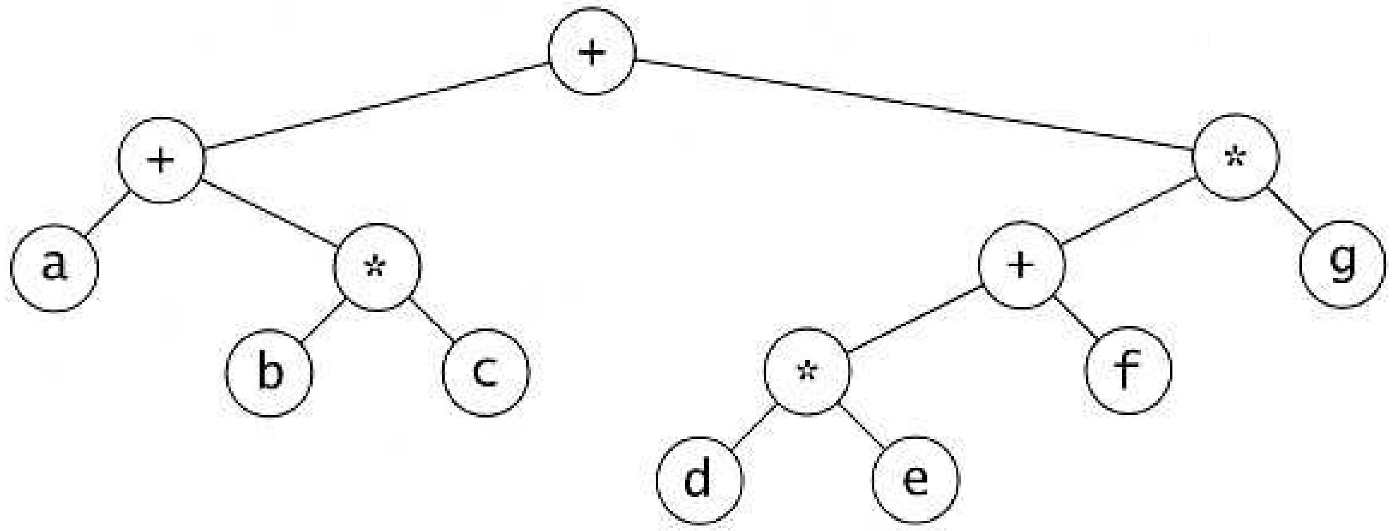


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- Leaves are operands (constants or variables)
- The other nodes (internal nodes) contain operators

Traversal of a Binary Tree

- Used to display/access the data in a tree in a certain order.
- In traversing always right sub-tree is traversed after left sub-tree.
- Three methods of traversing
  - **Preorder Traversing**
    - Root – Left –Right
  - **Inorder Traversing**
    - Left – Root – Right
  - **Postorder Traversing**
    - Left – Right - Root

Preorder Traversal

- **Preorder traversal**
  - **Node – Left – Right**
  - **Prefix expression**
    - **++a*bc*+*defg**

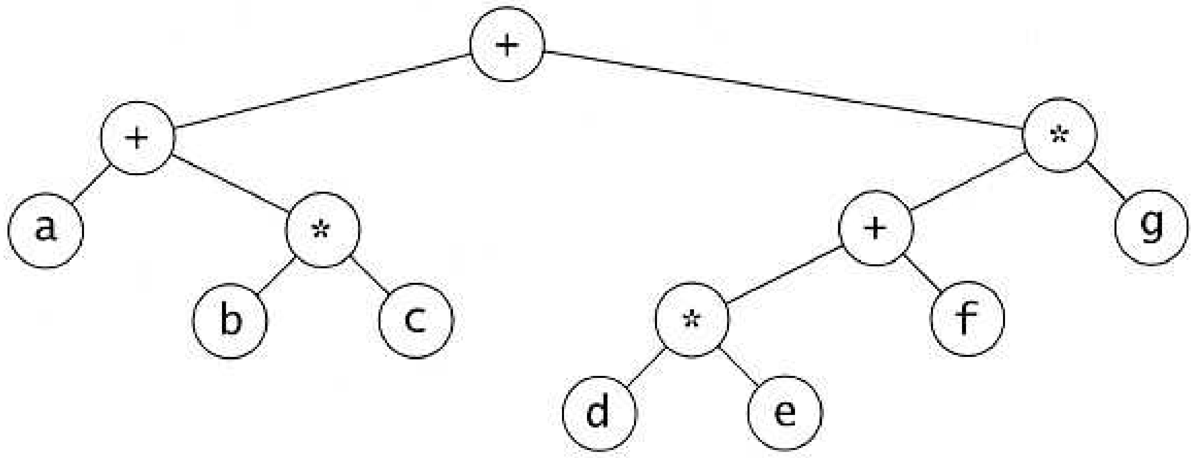


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Inorder Traversal

- **Inorder traversal**
  - **left, node, right.**
  - **infix expression**
    - **a+b*c+d*e+f*g**

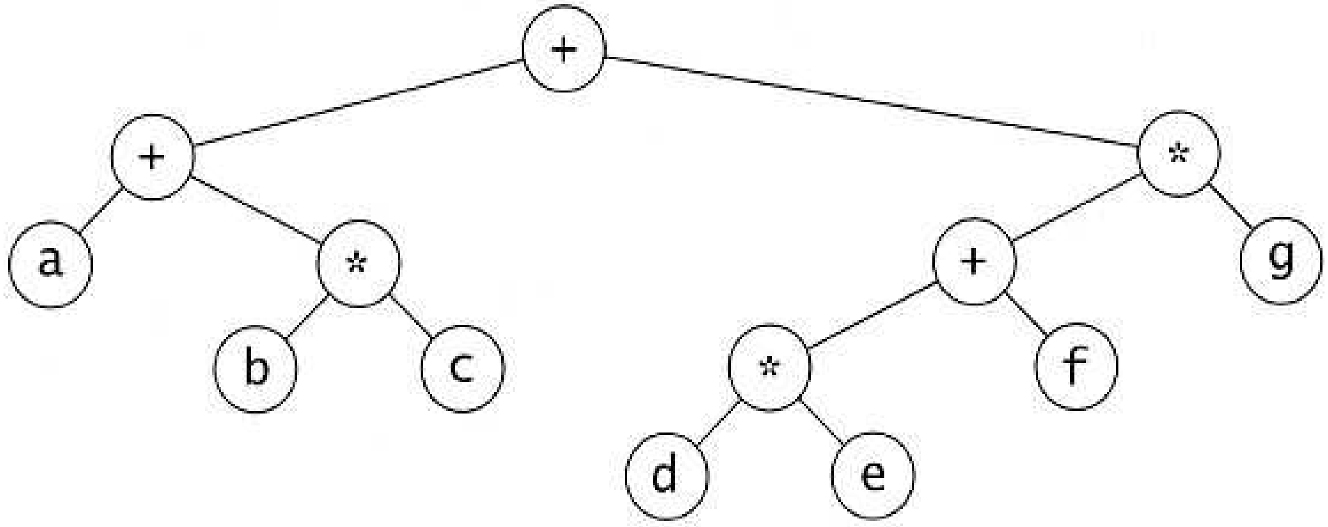


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Postorder Traversal

- **Postorder Traversal**
  - **left, right, node**
  - **postfix expression**
    - **abc*+de*f+g*+**

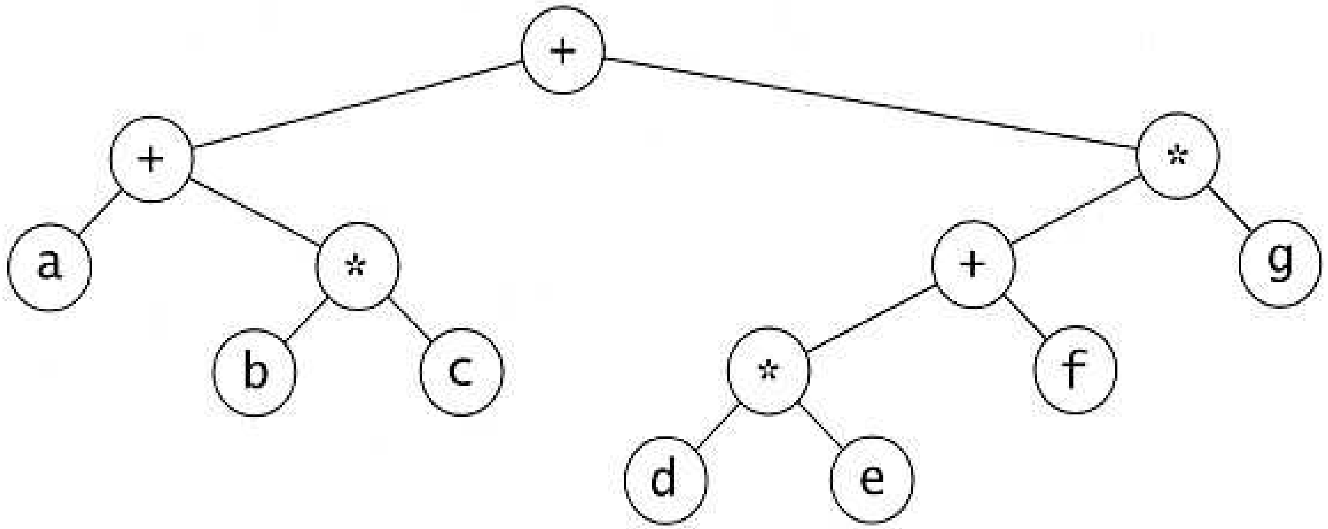
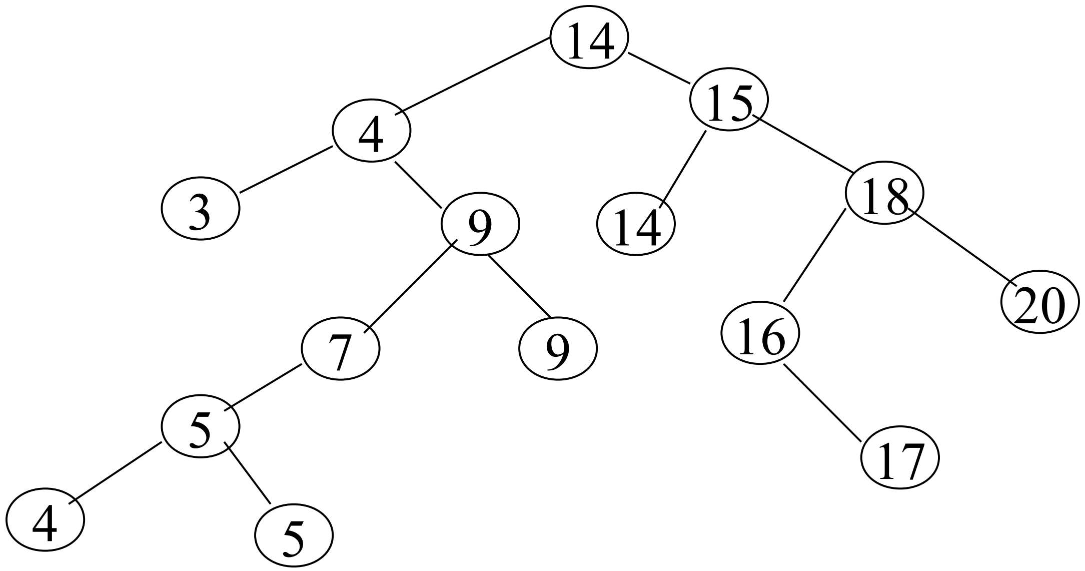


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Traversal Exercise

Traverse the following tree.



Pre-order Traversal?

Post-order Traversal?

In-order Traversal?

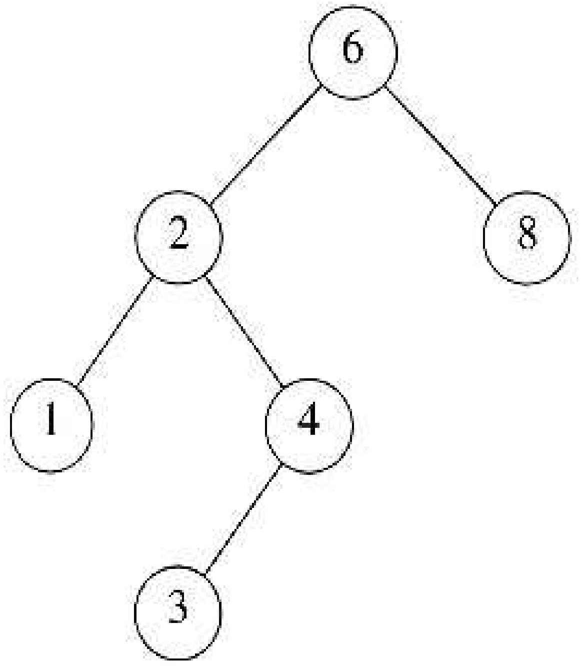
Binary Search Tree

Binary Search Tree

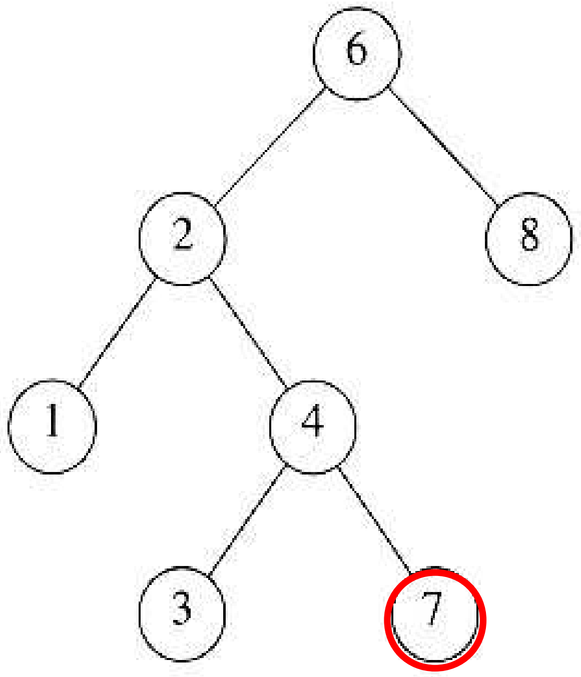
- **Stores keys in the nodes in a way so that searching, insertion and deletion can be done efficiently.**

- **Binary search tree is either empty or each node N of tree satisfies the following property**
  - **The Key value in the left child is not more than the value of root**
  - **The key value in the right child is more than or identical to the value of root**
  - **All the sub-trees, i.e. left and right sub-trees follow the two rules mention above.**

Examples



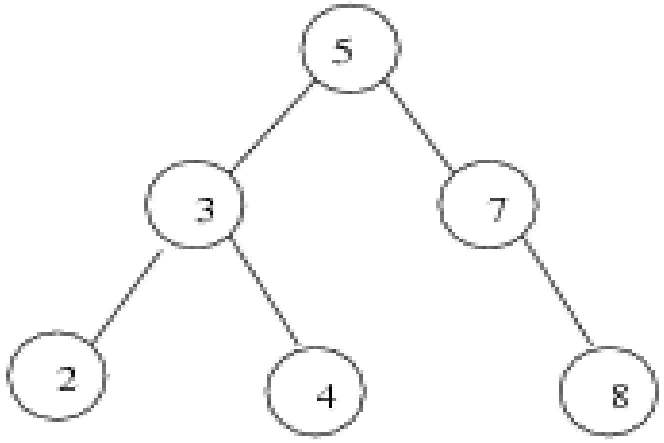
A binary search tree

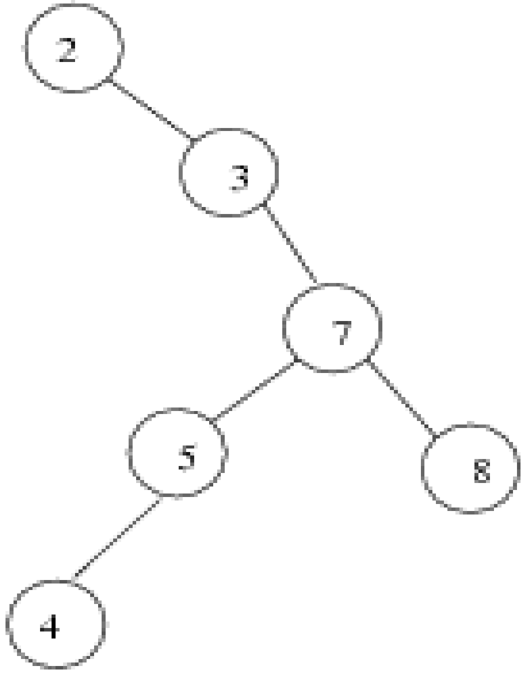


Not a binary search tree

Example 2

Two binary search trees representing the same Data set:

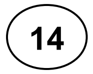




Example

- Input list of numbers:

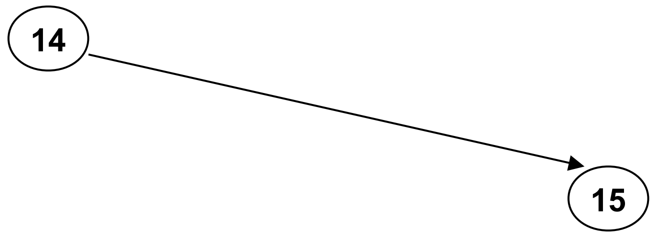
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

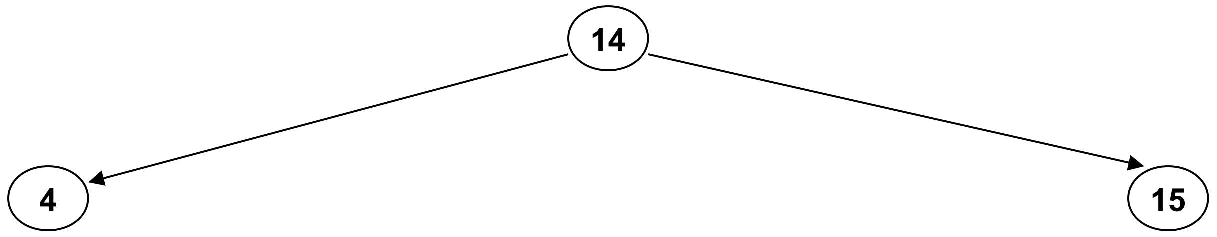
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5


$$\begin{tikzcd}14 \arrow[dr] & \\ & 15\end{tikzcd}$$

Example

- Input list of numbers:

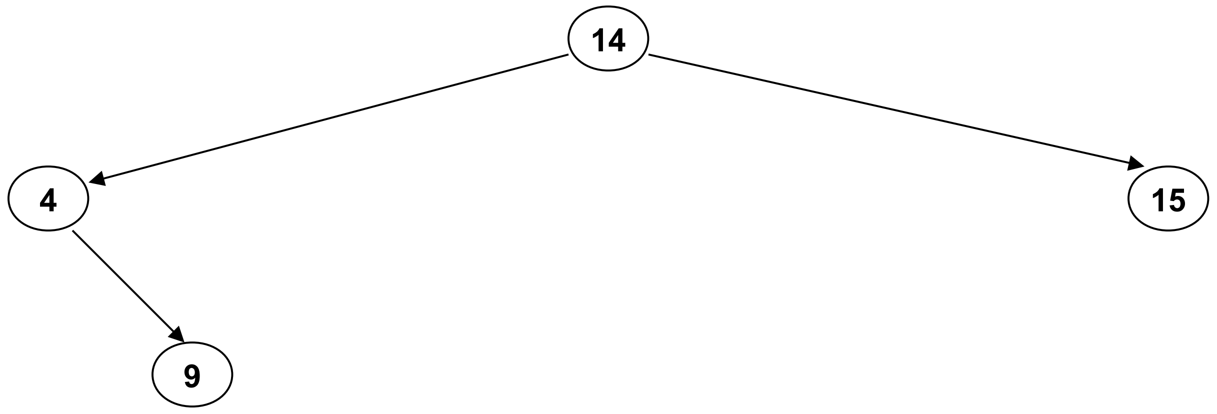
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

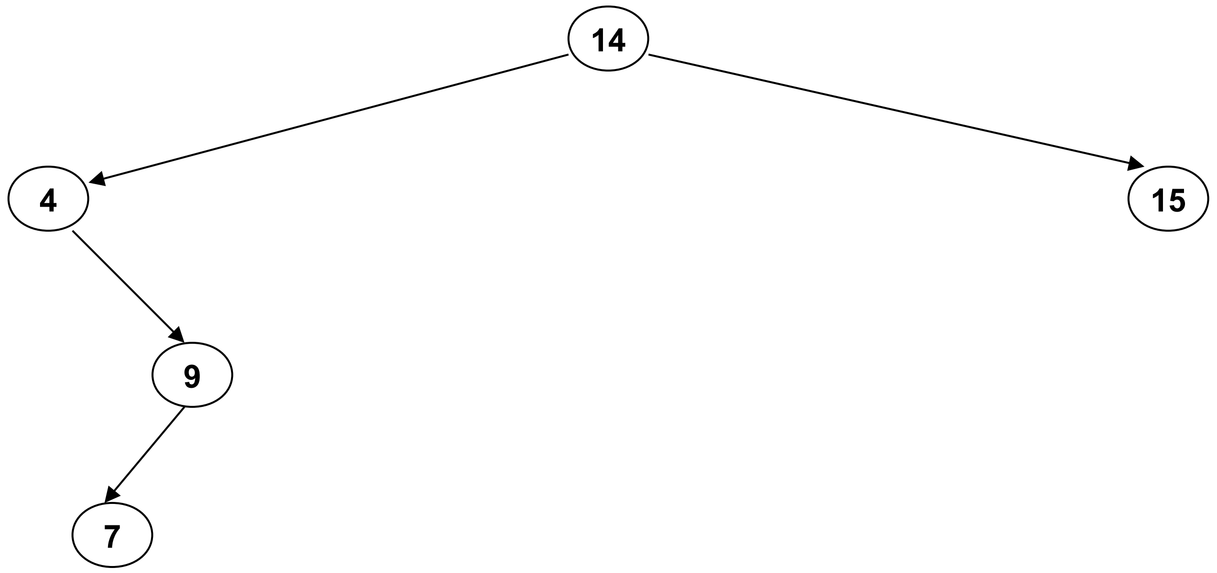
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

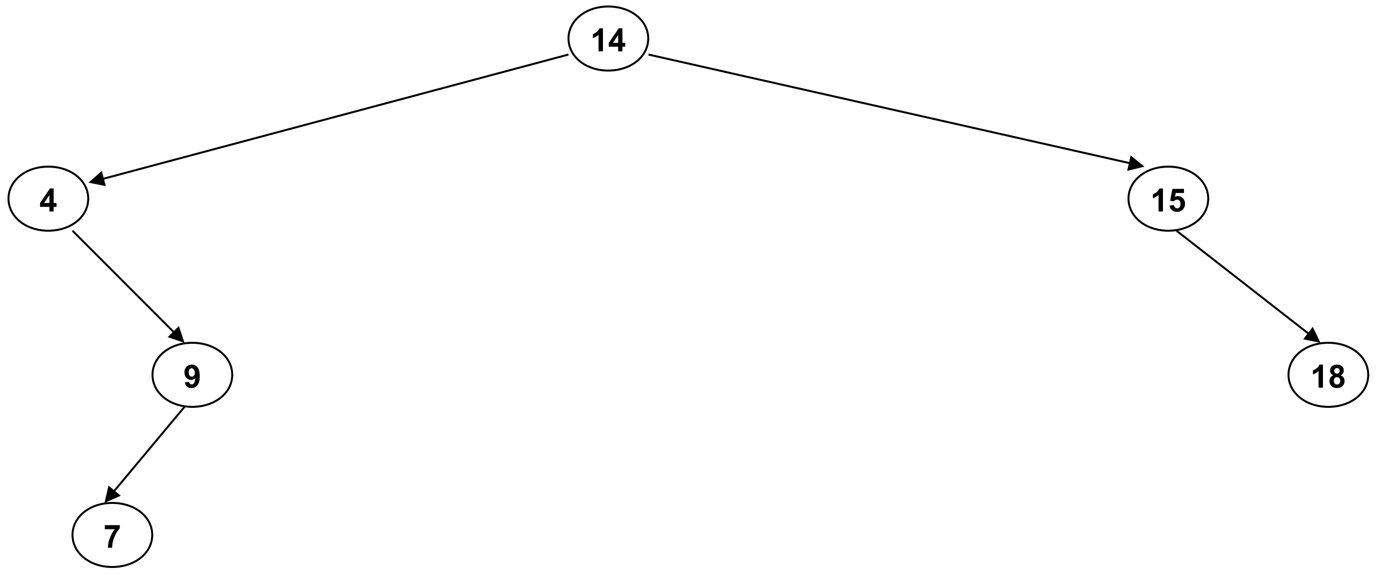
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

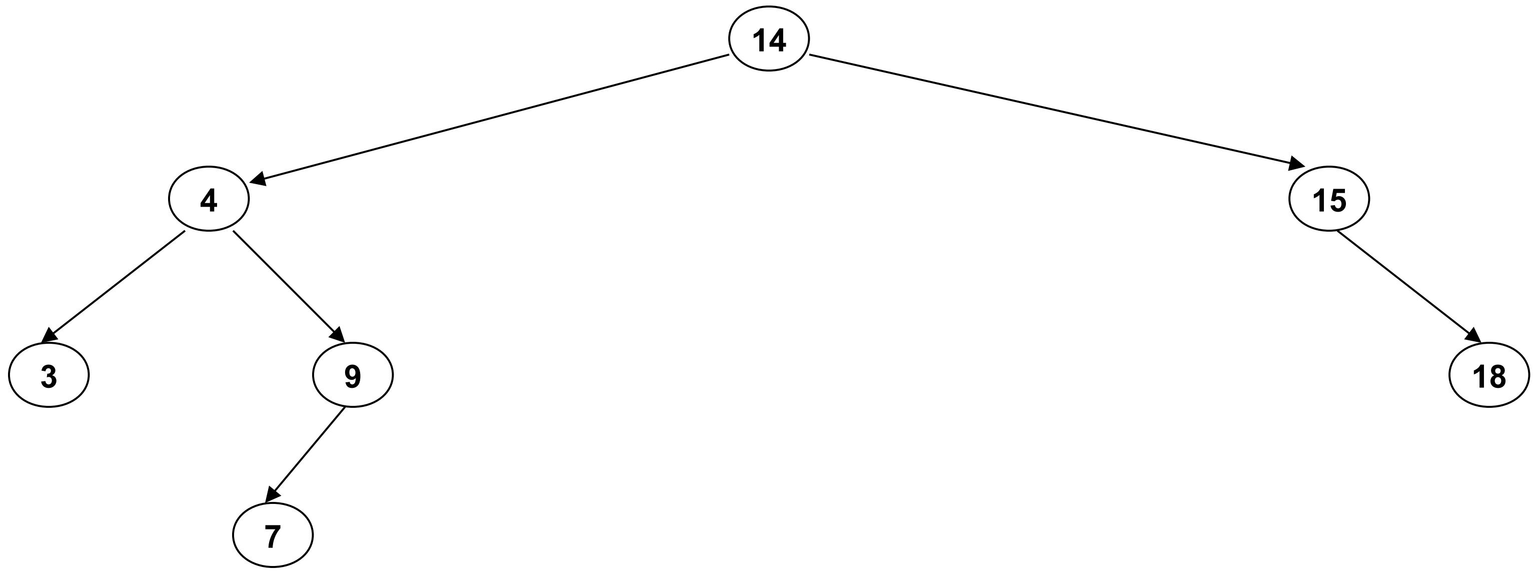
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

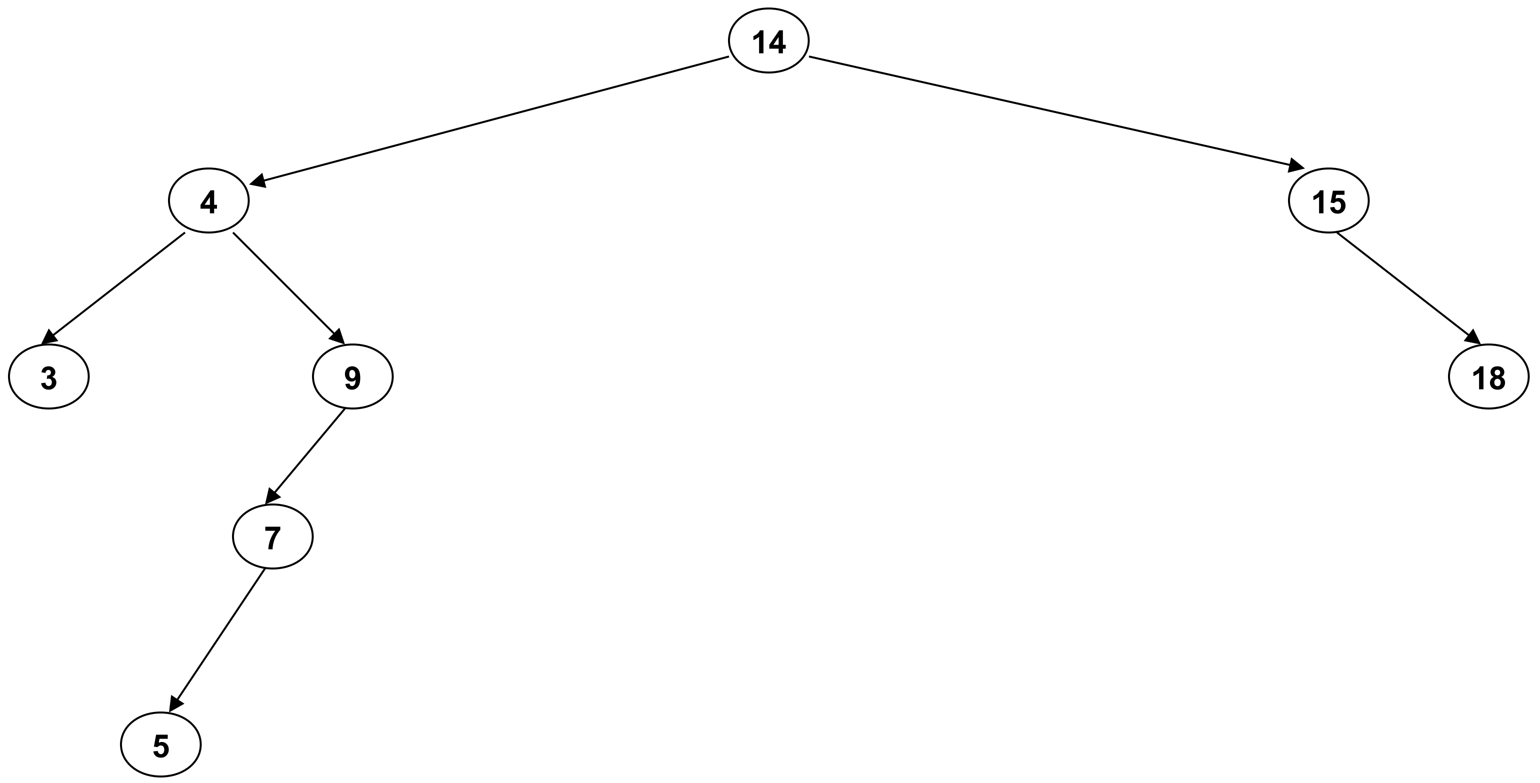
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

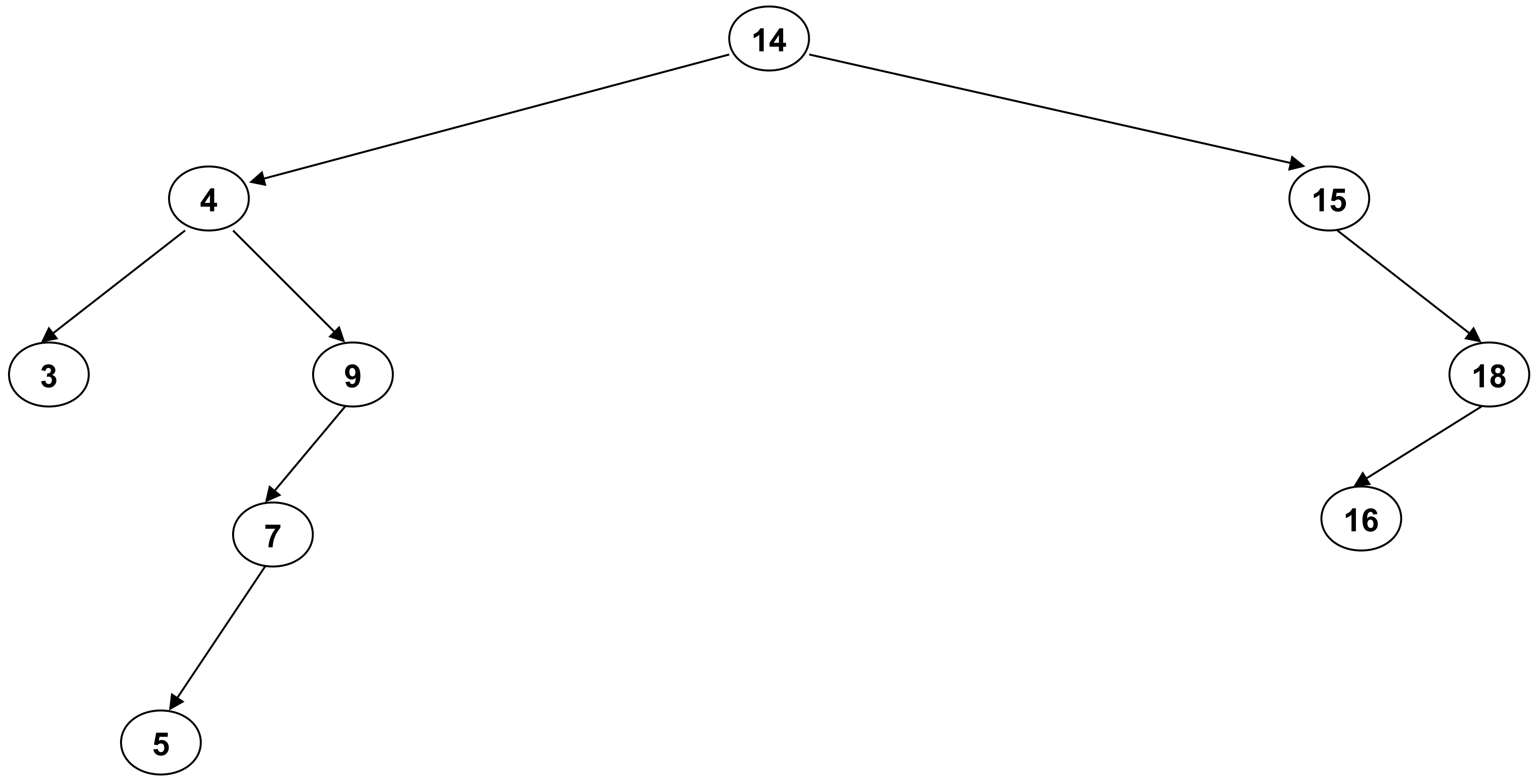
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

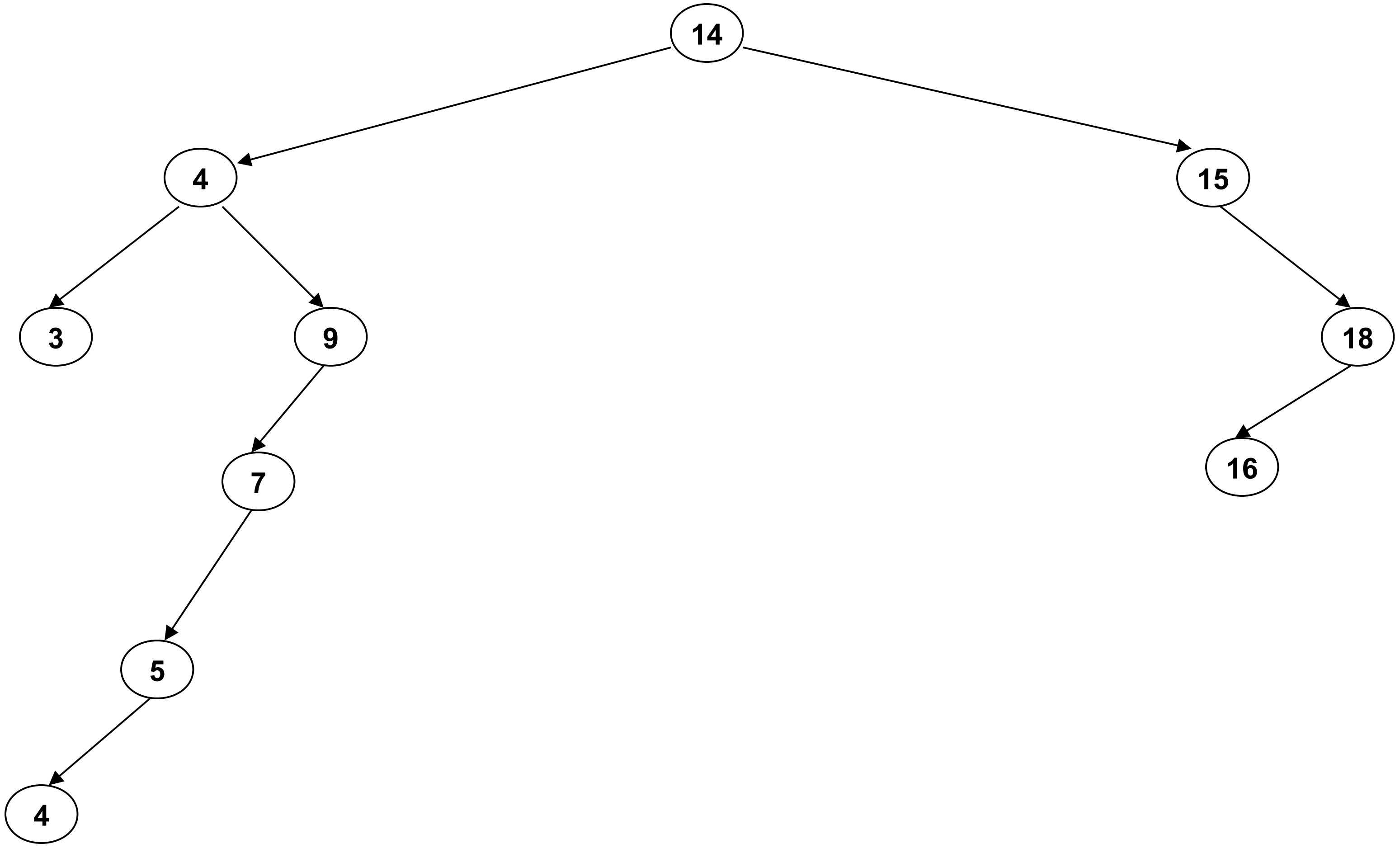
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

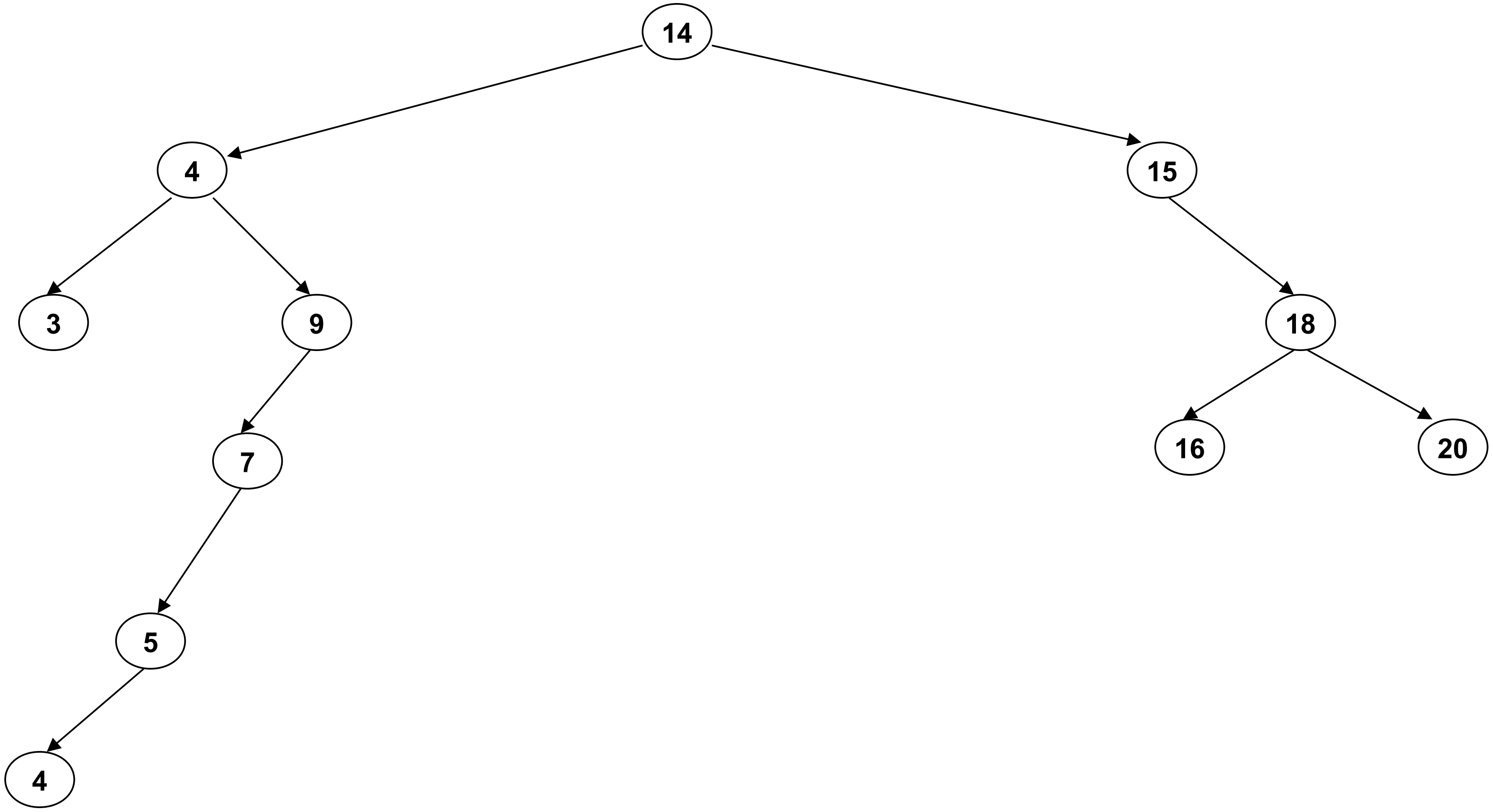
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

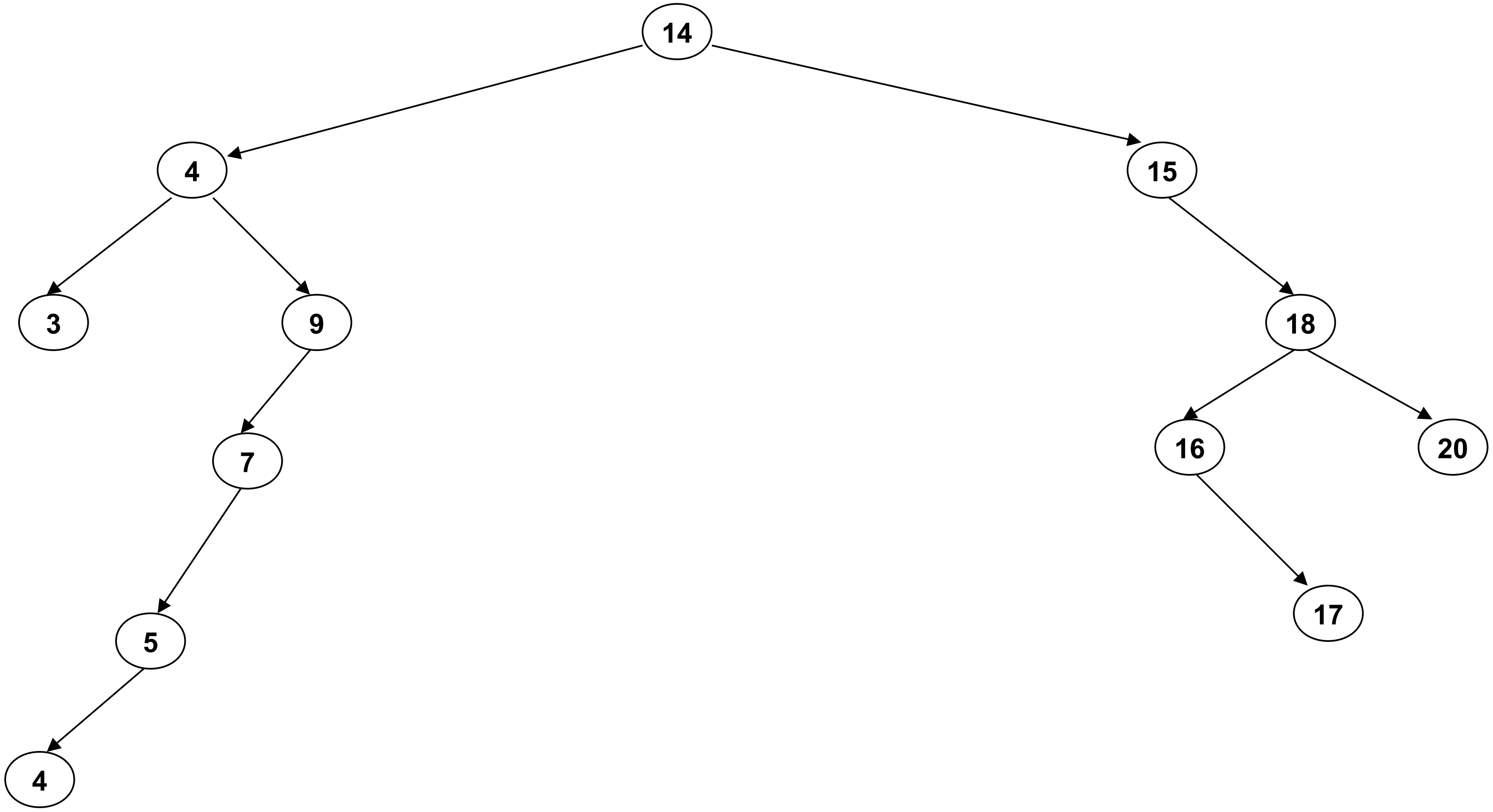
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

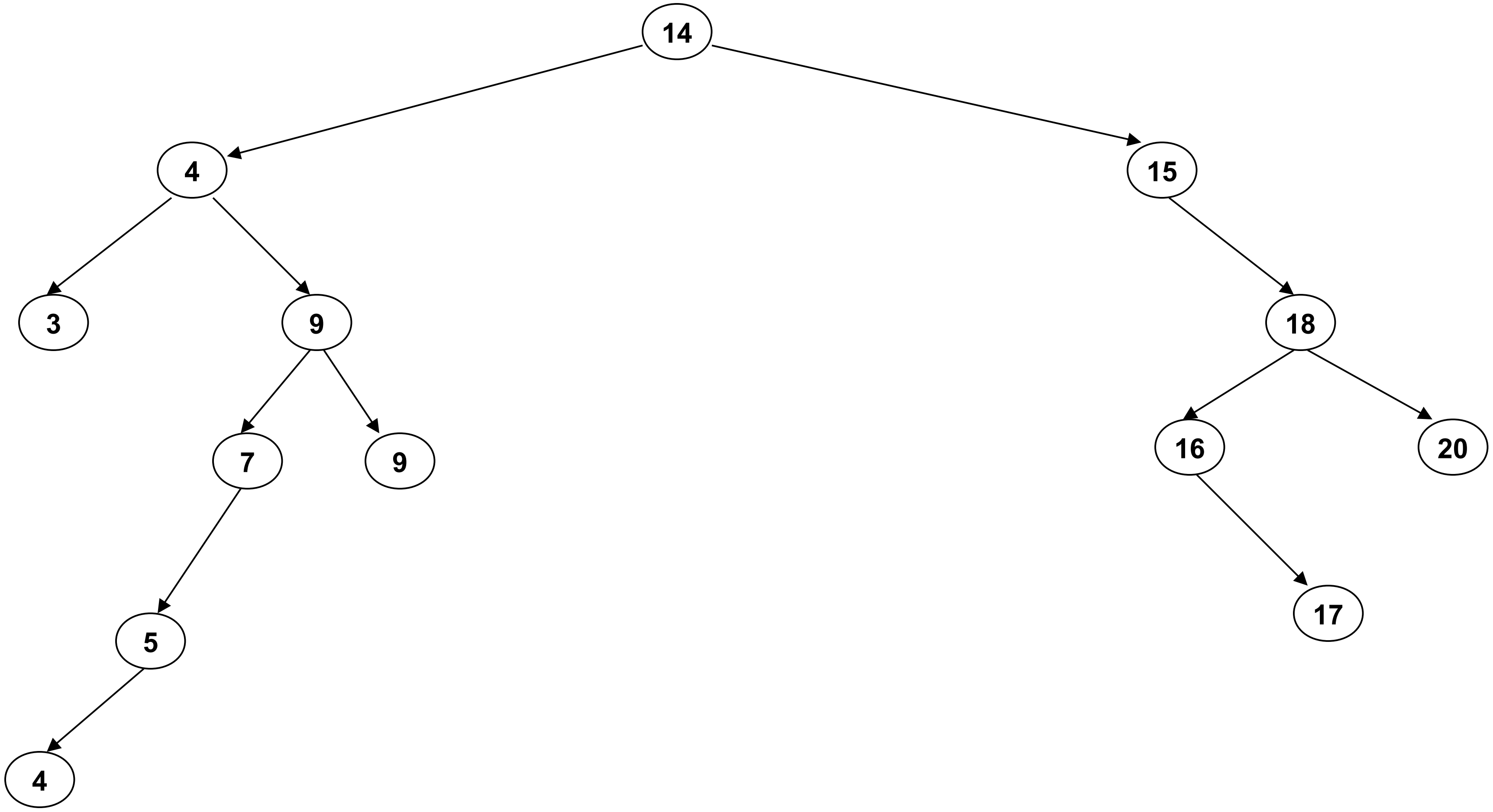
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

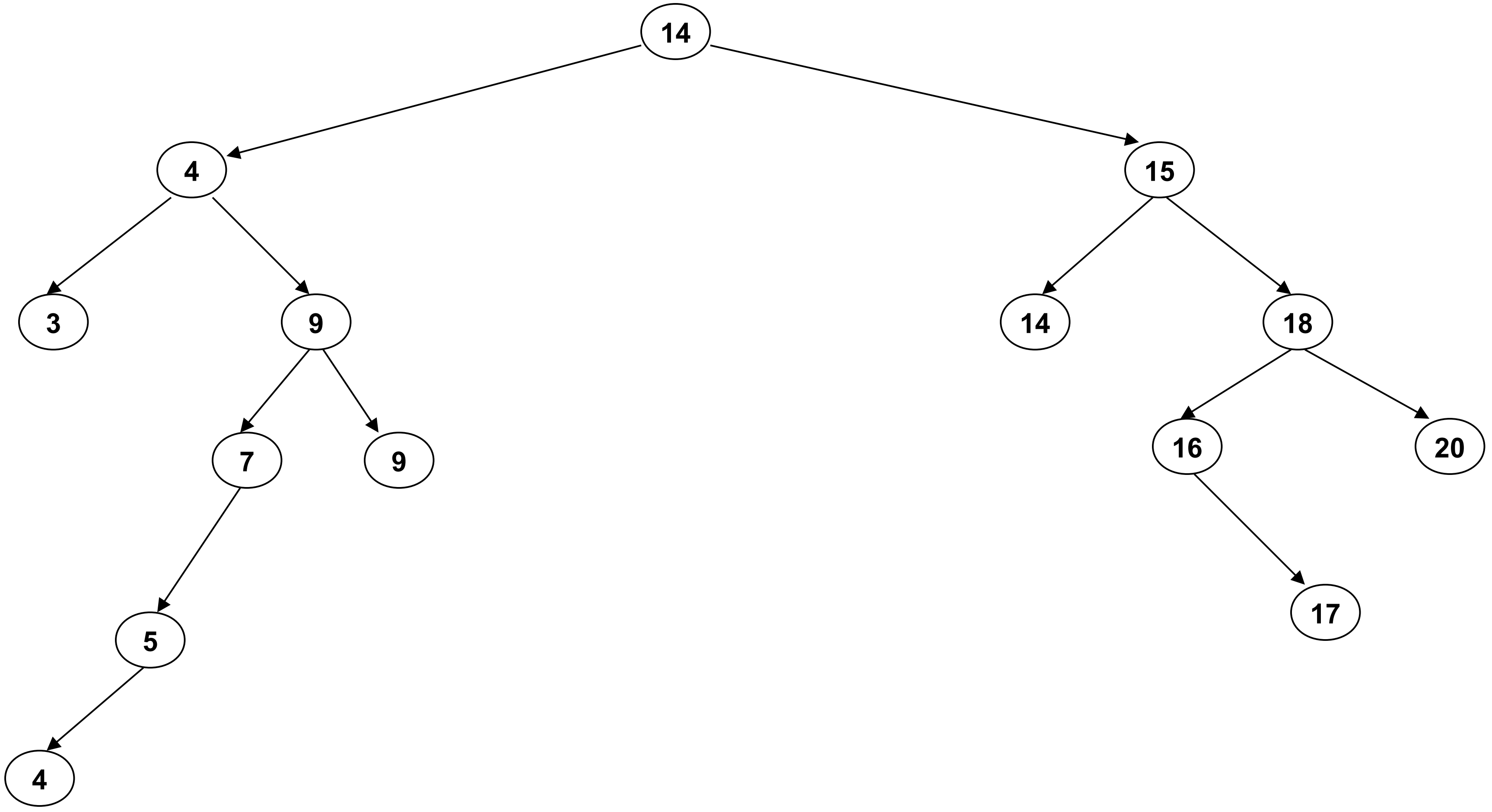
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

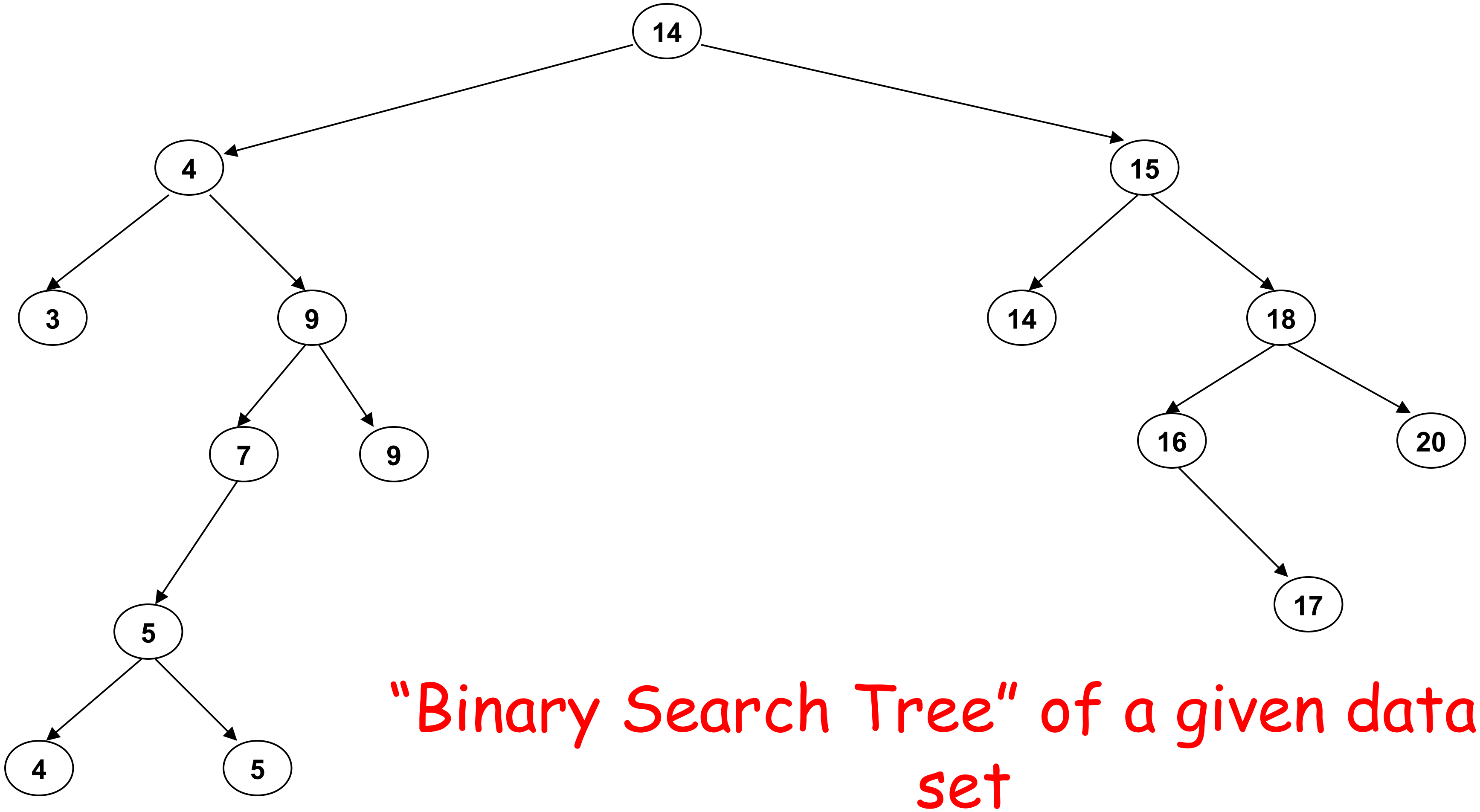
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



"Binary Search Tree" of a given data set

Binary Tree Implementation

```
Class BinaryTree{
      private:
            struct node{
                  int data;
                  node* LTree;
                  node* RTree;
            };
   public:
       node* root;
       BinaryTree( ){ root = NULL; }
      node* Insert(node* , int);
      void Search(node* , int);
      void InorderTraversal(node*);
      void PreorderTraversal(node*);
      void PostorderTraversal(node*);
};
```

Inorder Traversal Function

```
void BinaryTree::InorderTraversal(node* temp)
{
     if(temp!=NULL)
   {
          InorderTraversal(temp->LTree);
          cout<< temp->data;
     InorderTraversal(temp->RTree);
   }
}
```

Postorder Traversal Function

```
void BinaryTree::PostorderTraversal(node* temp)
{
    if(temp!=NULL)
  {
        PostorderTraversal(temp->LTree);
        PostorderTraversal(temp->RTree);
        cout<< temp->data;
  }
}
```

Preorder Traversal Function

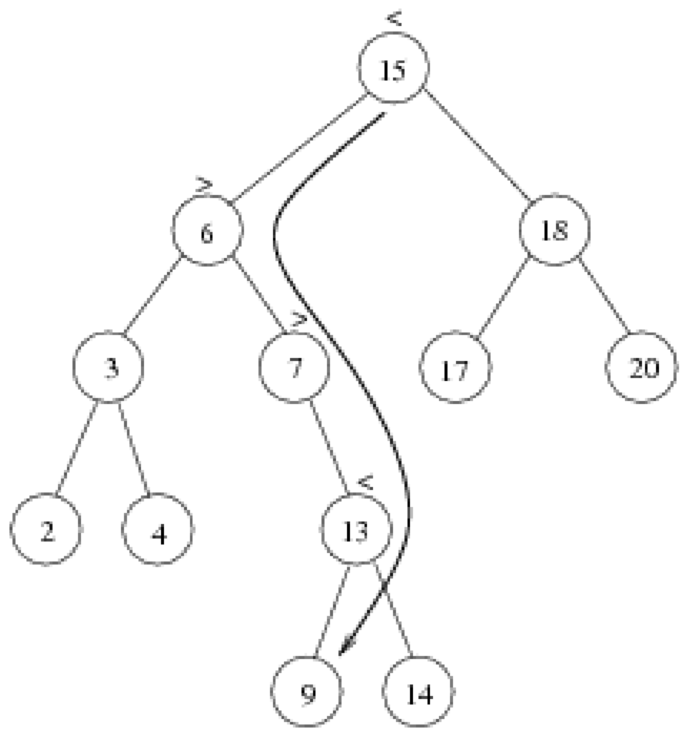
```
void BinaryTree::PreorderTraversal(node* temp)
{
   if(temp!=NULL)
  {
        cout<<temp->data;
        PreorderTraversal(temp->LTree);
        PreorderTraversal(temp->RTree);
  }
}
```

Searching in Binary Search Tree

- **Three steps of searching**
  - **The item which is to be searched is compared with the root node. If the item is equal to the root, then we are done.**
  - **If its less than the root node then we search in the left sub-tree.**
  - **If its more than the root node then we search in the right sub-tree.**

- **The above process will continue till the item is found or you reached end of the tree.**

Example: Search for 9 ...



Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

Search Function

```
void BinaryTree::Search(node* temp, int num)
{
     if(temp==NULL)
          cout<<“Number not found";
   else if(temp->data == num)
               cout<<"Number found";
   else if(temp->data > num)
               Search(temp->LTree, num);
   else if(temp->data < num)
               Search(temp->RTree, num);
}
```

Insertion in BST

- **Three steps of insertion**
  - **If the root of the tree is NULL then insert the first node and root points to that node.**
  - **If the inserted number is lesser than the root node then insert the node in the left sub-tree.**
  - **If the inserted number is greater than the root node then insert the node in the right sub-tree.**

Insertion Function

```
node* BinaryTree::Insert(node* temp, int num)
{
    if ( temp == NULL )
    {
        temp = new node;
        temp->data= num;
    temp->LTree= NULL;
    temp->RTree=NULL;
    }
    else if(num < temp->data)
    temp->LTree = Insert(temp->LTree, num);
    else if( num >= temp->data)
    temp->RTree = Insert(temp->RTree,num);
    return temp;
}
```

Deletion in BST

- When we delete a node, we need to consider how we take care of the children of the deleted node.
  - This has to be done such that the property of the search tree is maintained.

Deletion in BST

Three cases:

(1) The node is a leaf

- **Delete it immediately**

(2) The node has one child

- **Adjust a pointer from the parent to bypass that node**

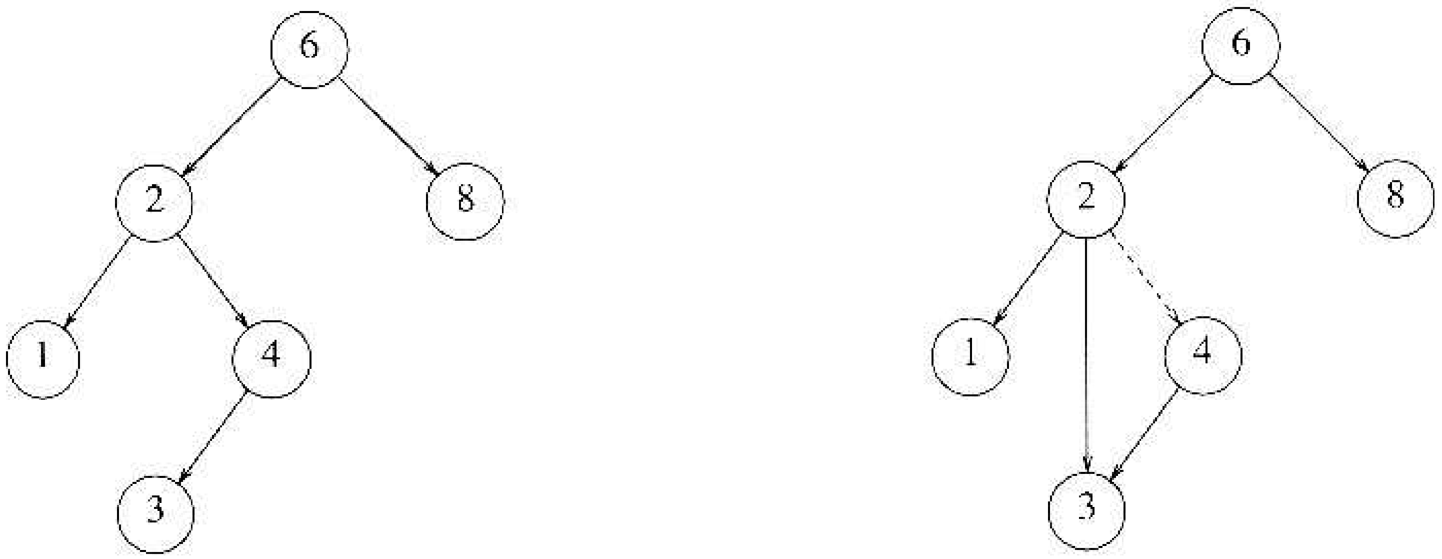


Figure 4.24 Deletion of a node (4) with one child, before and after

Deletion in BST

(3) The node has 2 children

- **Replace the key of that node with the minimum element at the right subtree**
- **Delete the minimum element**
  - **Has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.**

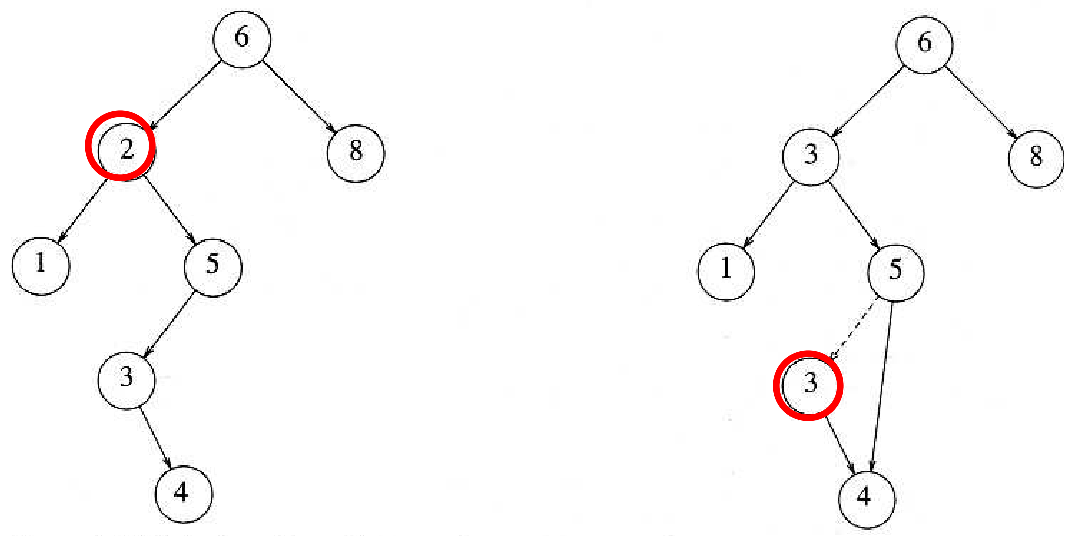


Figure 4.25 Deletion of a node (2) with two children, before and after

Deletion Code

```
void BTree::DeleteNode(node* temp, int num)
{     if (temp==NULL)
      cout<<"Number not Found";
  else if((temp->data == num))
  {     node *parent, *min ;
        int number;
    // if number is found at a leaf node
    if((temp->LTree == NULL) && (temp->RTree == NULL))
    {
          parent=GetParent(root, temp->data, root);  //will return parent node
     if(parent->LTree == temp)
          parent->LTree = NULL;
     else if (parent->RTree == temp)
                parent->RTree = NULL;
           delete temp;
    }
```

```
// if node to be deleted has one child
    else if(((temp->LTree == NULL) && (temp->RTree != NULL)) ||
((temp->LTree != NULL) && (temp->RTree == NULL)))
    {
     parent = GetParent(root, temp->data, root); //will return parent node
          if(temp->LTree != NULL){
               if(parent->LTree == temp)
               parent->LTree = temp->LTree;
          else if (parent->RTree == temp)
                    parent->RTree = temp->LTree;
           }
          else if(temp->RTree != NULL){
               if(parent->LTree == temp)
               parent->LTree = temp->RTree;
          else if (parent->RTree == temp)
                    parent->RTree = temp->RTree;
          }
          delete temp;
//if node to be deleted has two children
else if((temp->LTree != NULL) && (temp->RTree != NULL))
    {
      min = FindMin(temp->RTree);  //will return the min. no. found in RTree
       number = min->data;
      DeleteNode(temp->RTree, min->data);  //calling to itself recursively
      temp->data= number;
      }
  }



  else if (num < temp->data)
     DeleteNode(temp->LTree, num);   //calling to itself recursively
  else if (num > temp->data)
     DeleteNode(temp->RTree, num); //calling to itself recursively
}
```

Full Binary Tree A Binary Tree is full if every node has 0 or 2 children. Following are examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaves have two children.

```
               18
           /       \
         15          30
        /  \        /  \
      40    50    100   40


             18
           /    \
         15      20
        /  \
      40    50
     /   \
   30    50


               18
            /      \
          40        30
                   /  \
                 100   40
```

In a Full Binary, number of leaf nodes is number of internal nodes plus 1

L = I + 1

Where L = Number of leaf nodes, I = Number of internal nodes

Complete Binary Tree: A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible

Following are examples of Complete Binary Trees

```
               18
           /       \
         15         30
        /  \        /  \
      40    50    100   40


               18
           /       \
         15         30
        /  \        /  \
      40    50    100   40
     /  \   /
    8   7  9
```

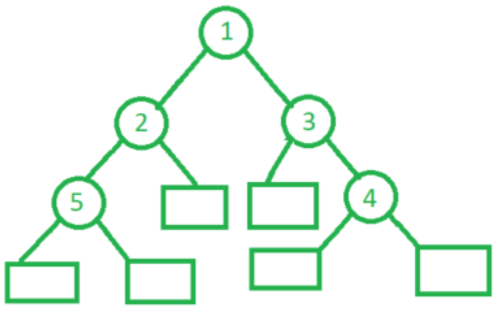
Perfect Binary Tree A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at the same level. Following are examples of Perfect Binary Trees.

```
               18
           /       \
         15         30
        /  \        /  \
      40    50    100   40


               18
           /       \
         15         30
```

A Perfect Binary Tree of height h (where height is the number of nodes on the path from the root to leaf) has $2^h - 1$ node.
Example of a Perfect binary tree is ancestors in the family. Keep a person at root, parents as children, parents of parents as their children.

Extended Binary Tree

Extended binary tree is a type of binary tree in which all the null sub tree of the original tree are replaced with special nodes called **external nodes** whereas other nodes are called **internal nodes**



Extended Binary Tree

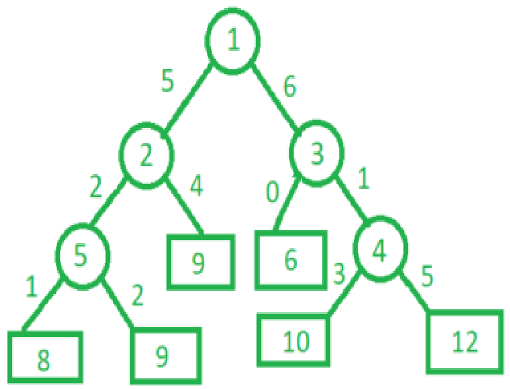
Here the circles represent the internal nodes and the boxes represent the external nodes.

Properties of External binary tree

1. The nodes from the original tree are internal nodes and the special nodes are external nodes.
2. All external nodes are leaf nodes and the internal nodes are non-leaf nodes.
3. Every internal node has exactly two children and every external node is a leaf. It displays the result which is a complete binary tree

Application of extended binary tree:

1. **Calculate weighted path length:** It is used to calculate total path length in case of weighted tree.



Total path length

P = 8*3 + 9*3 + 9*2 + 6*2 + 10*3 + 12*3

=> 147

Here, the sum of total weights is already calculated and stored in the external nodes and thus makes it very easier to calculate the total path length of a tree with given weights. The same technique can be used to update routing tables in a network.

2. **To convert binary tree in Complete binary tree:** The above-given tree having removed all the external nodes, is not a complete binary tree. To introduce any tree as complete tree, external nodes are added onto it. Heap is a great example of a complete binary tree and thus each binary tree can be expressed as heap if external nodes are added to it.

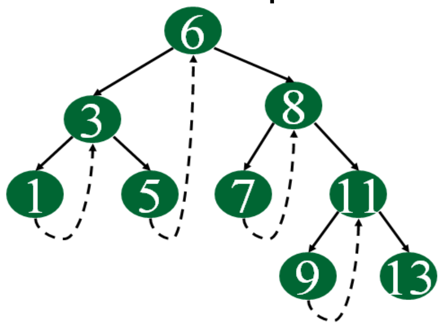
Threaded Binary Tree

Inorder traversal of a Binary tree can either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).
There are two types of threaded binary trees.
Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

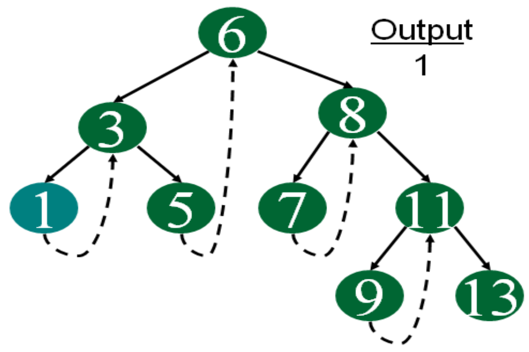
Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

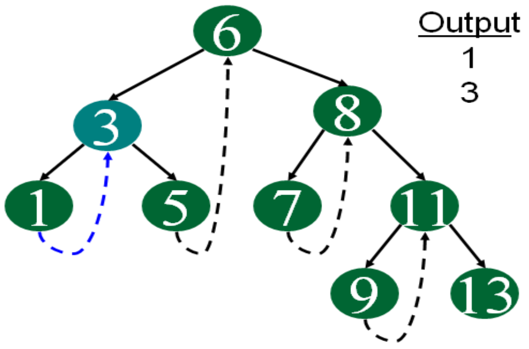
Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



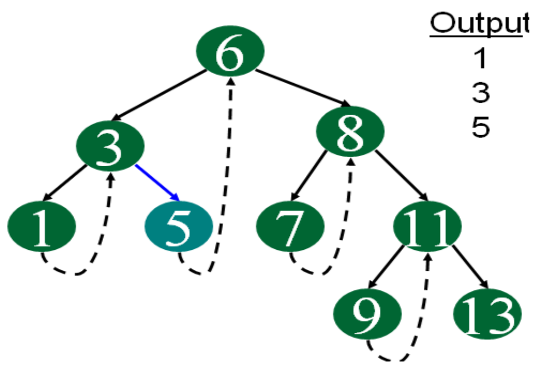
Following diagram demonstrates inorder order traversal using threads.



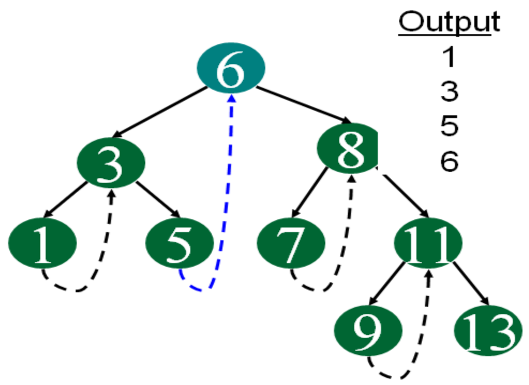
Start at leftmost node, print it



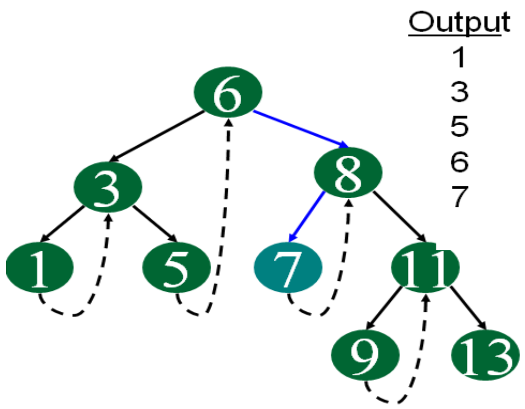
Follow thread to right, print node



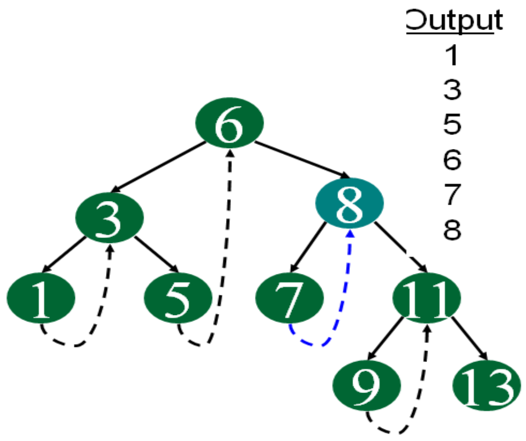
Follow link to right, go to
leftmost node and print



Follow thread to right, print node



Follow link to right, go to
leftmost node and print



Follow thread to right, print node

continue same way for remaining node.....

Threaded Trees

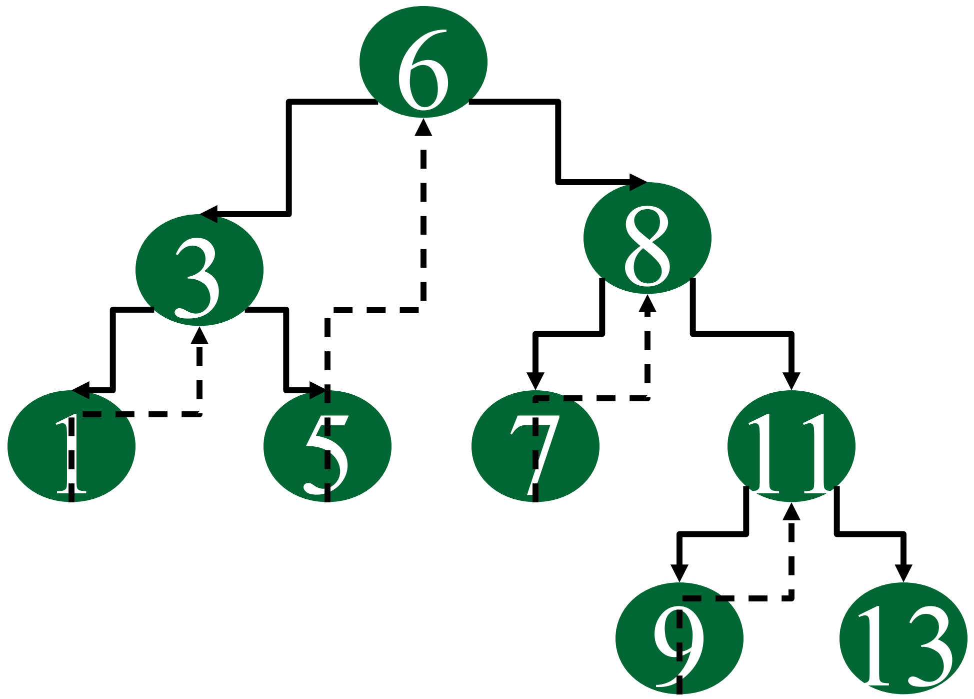
- Binary trees have a lot of wasted space: the leaf nodes each have 2 null pointers
- We can use these pointers to help us in inorder traversals
- We have the pointers reference the next node in an inorder traversal; called *threads*
- We need to know if a pointer is an actual link or a thread, so we keep a boolean for each pointer

Threaded Tree Code

- Example code:

```
class Node {
    Node left, right;
    boolean leftThread, rightThread;
}
```

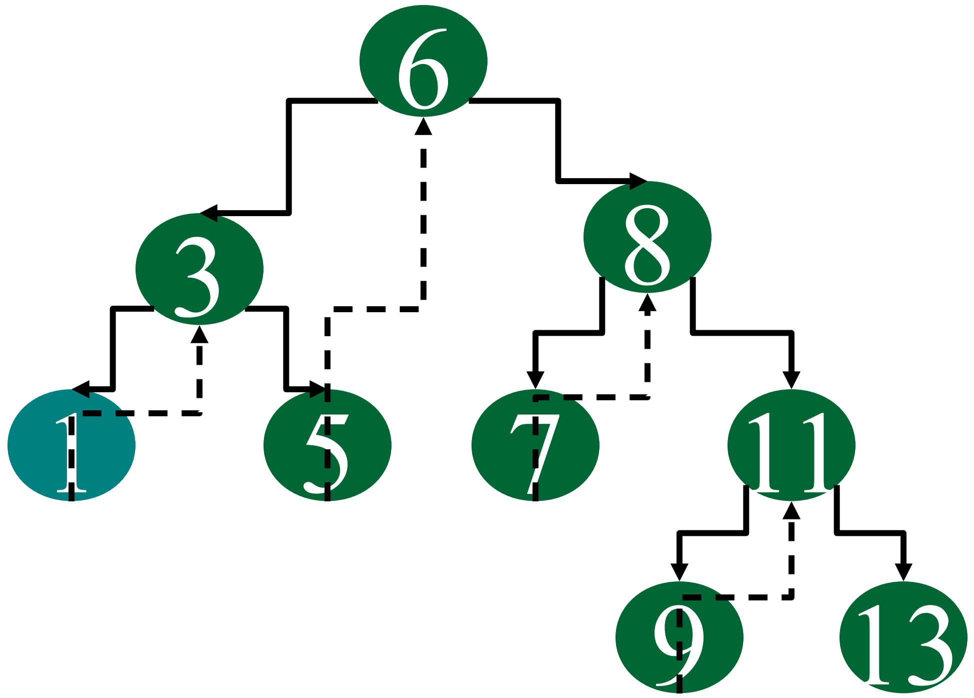
Threaded Tree Example



Threaded Tree Traversal

- We start at the leftmost node in the tree, print it, and follow its right thread
- If we follow a thread to the right, we output the node and continue to its right
- If we follow a link to the right, we go to the leftmost node, print it, and continue

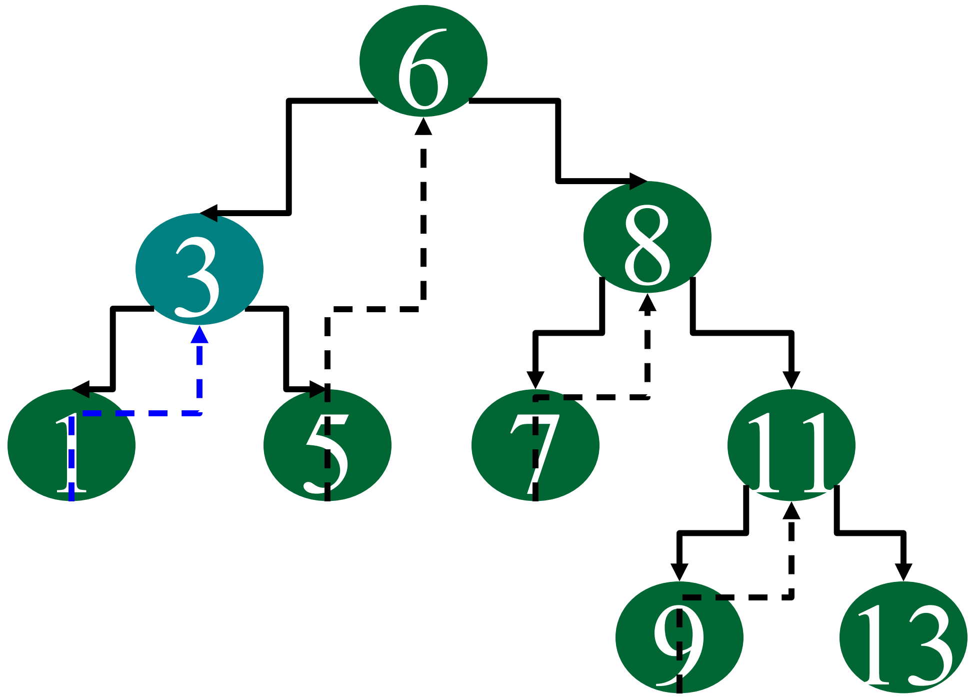
Threaded Tree Traversal



<u>Output</u>
1

Start at leftmost node, print it

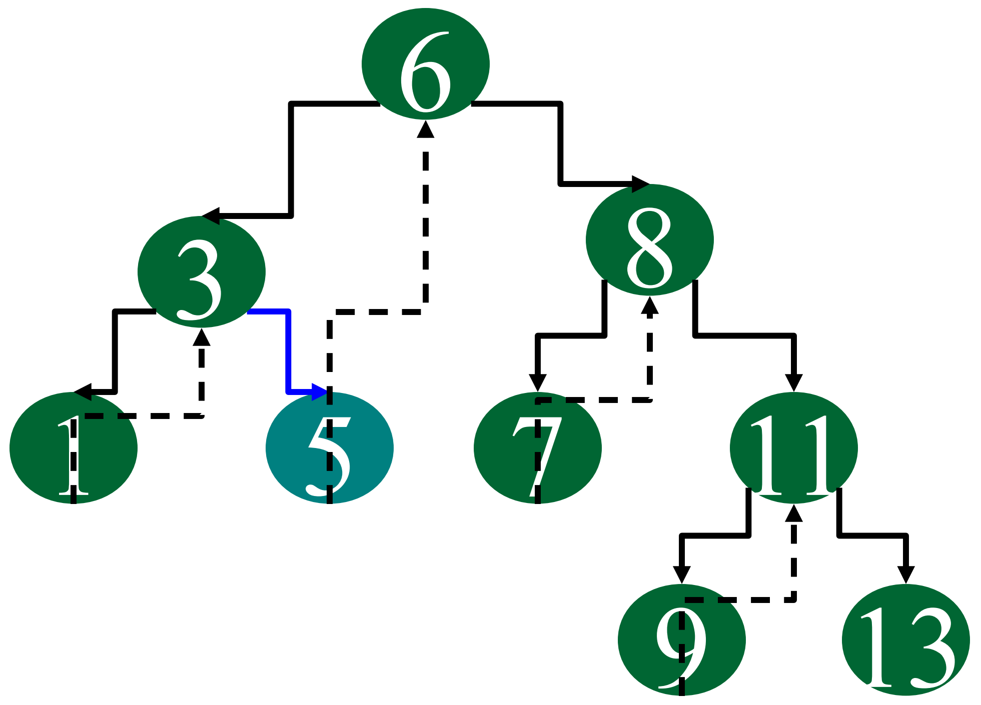
Threaded Tree Traversal



Output
1
3

Follow thread to right, print node

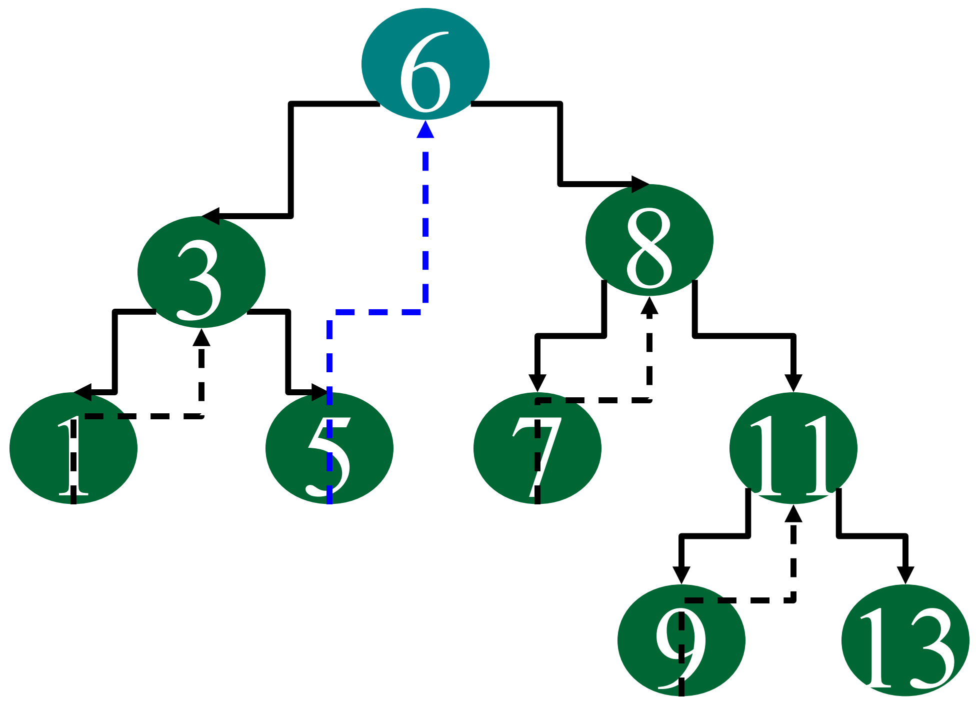
Threaded Tree Traversal



Output
1
3
5

Follow link to right, go to leftmost node and print

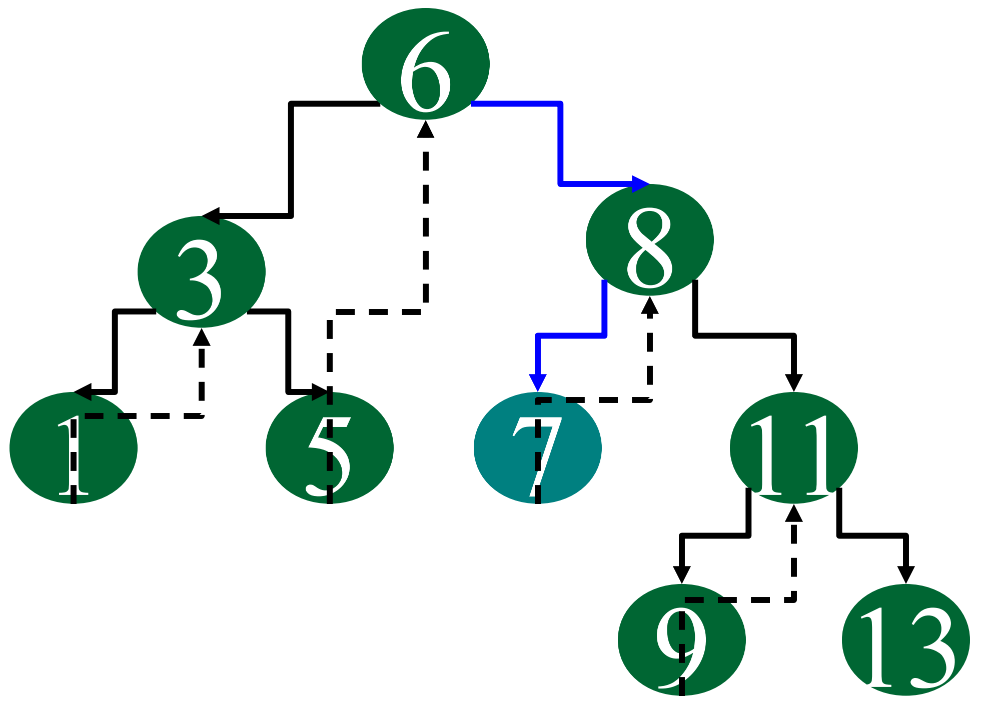
Threaded Tree Traversal



Output
1
3
5
6

Follow thread to right, print node

Threaded Tree Traversal

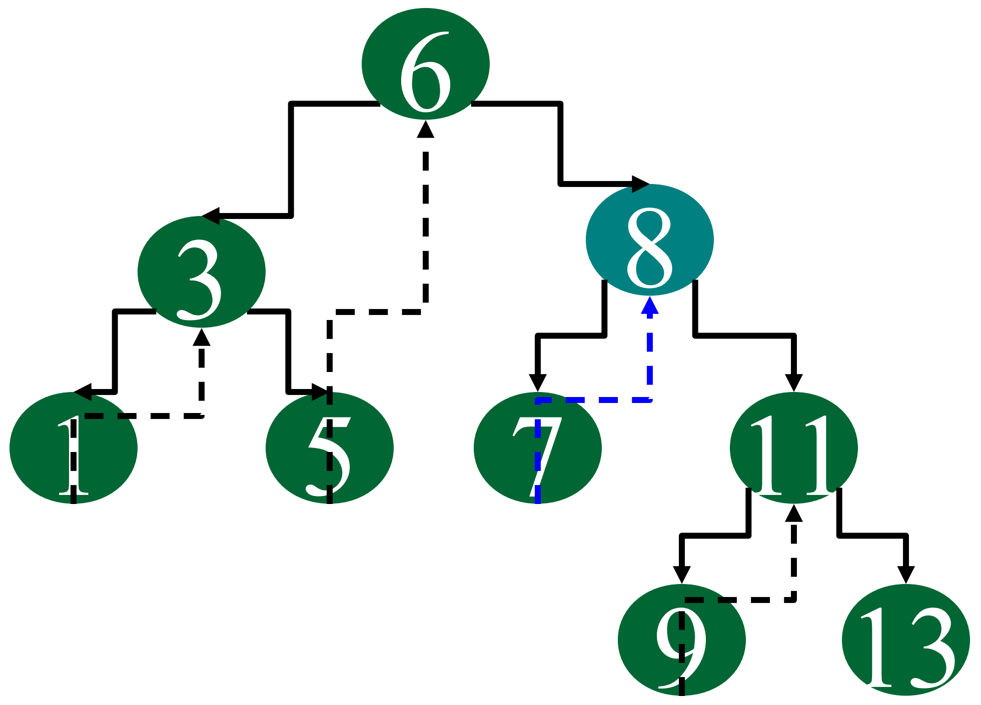


Output

1
3
5
6
7

Follow link to right, go to leftmost node and print

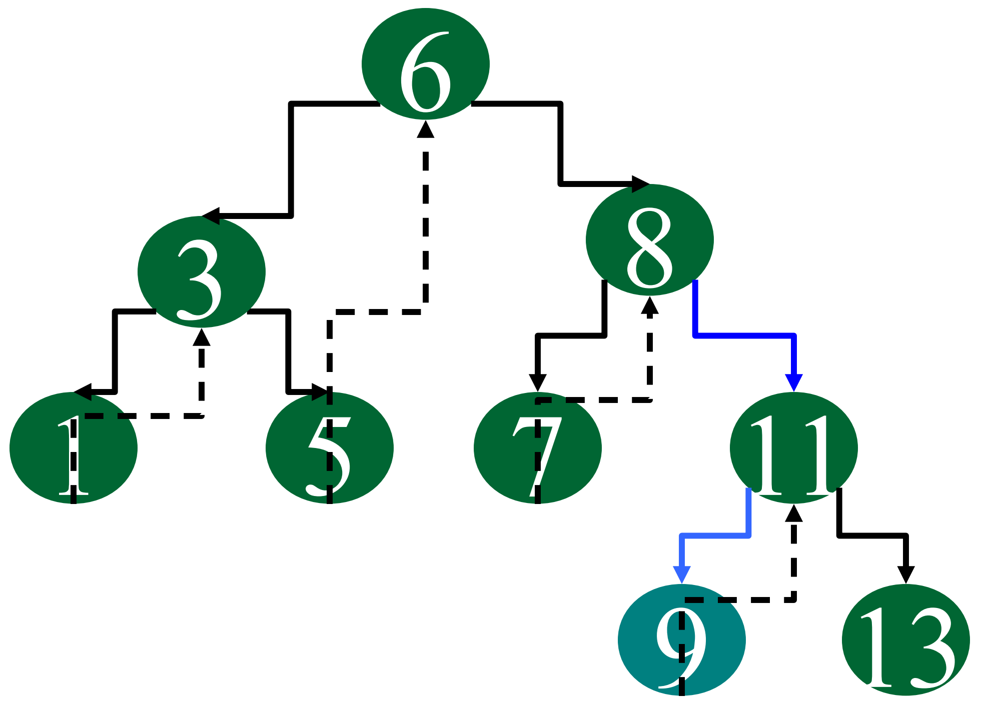
Threaded Tree Traversal



Output
1
3
5
6
7
8

Follow thread to right, print node

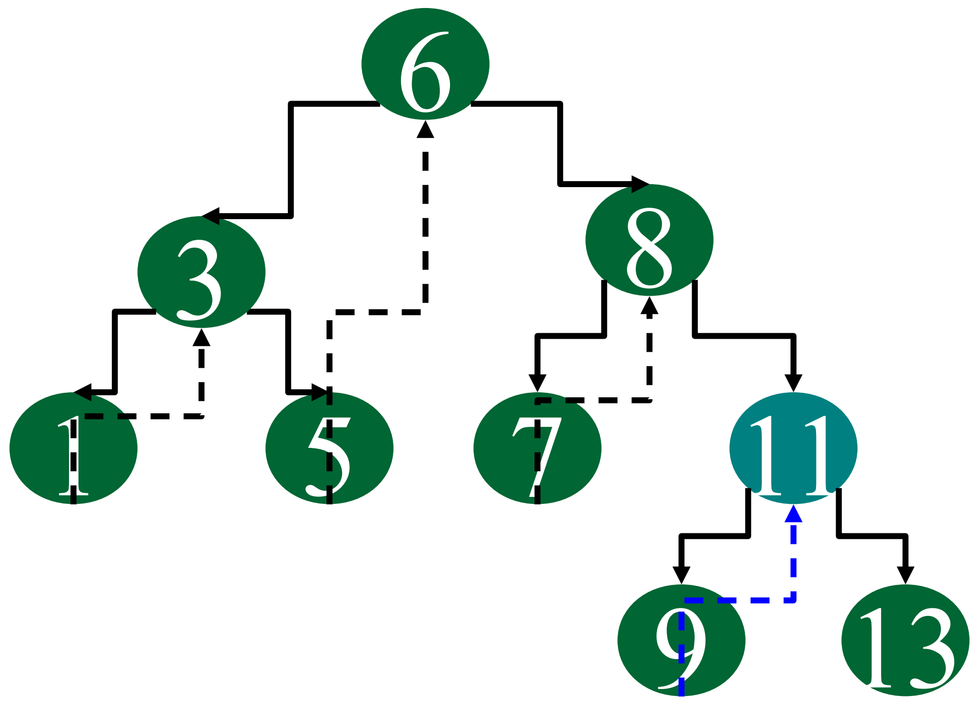
Threaded Tree Traversal



Output
1
3
5
6
7
8
9

Follow link to right, go to leftmost node and print

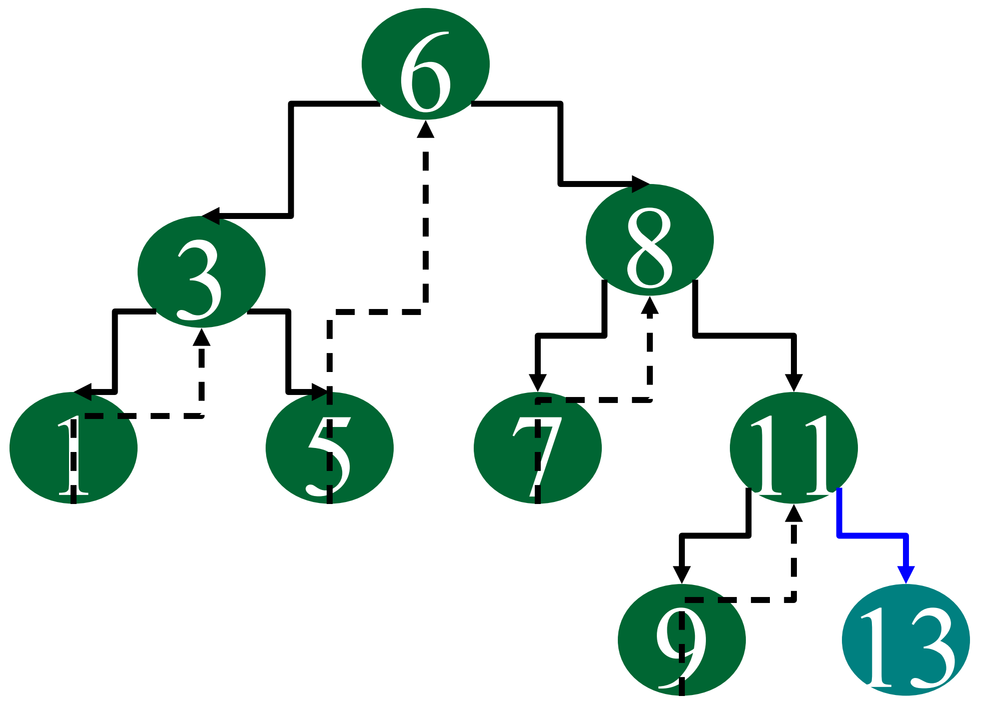
Threaded Tree Traversal



Output
1
3
5
6
7
8
9
11

Follow thread to right, print node

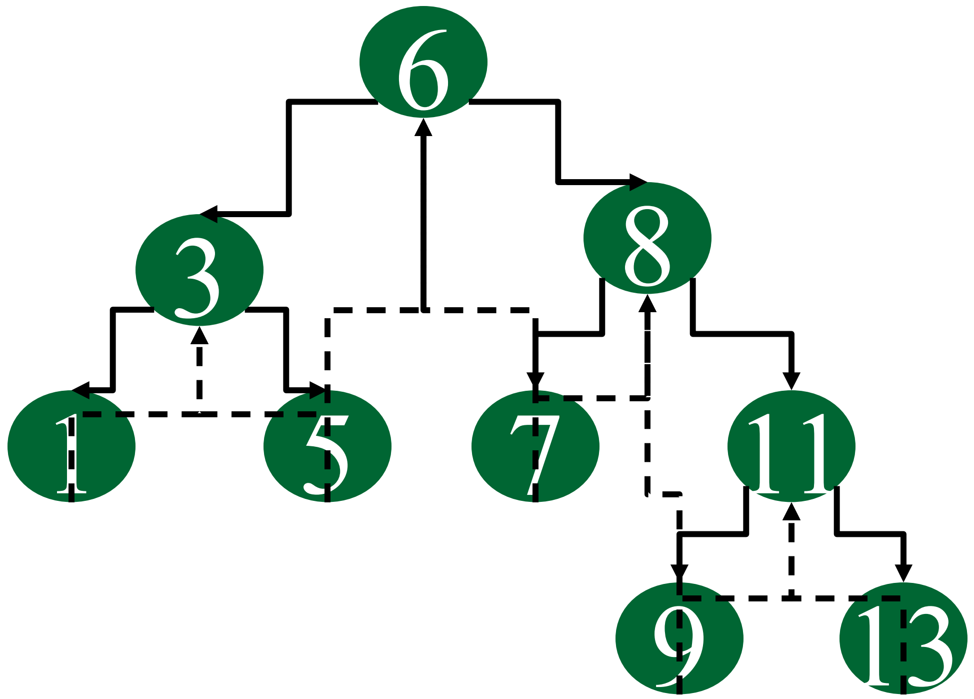
Threaded Tree Traversal



Output
1
3
5
6
7
8
9
11
13

Follow link to right, go to leftmost node and print

Threaded Tree Modification

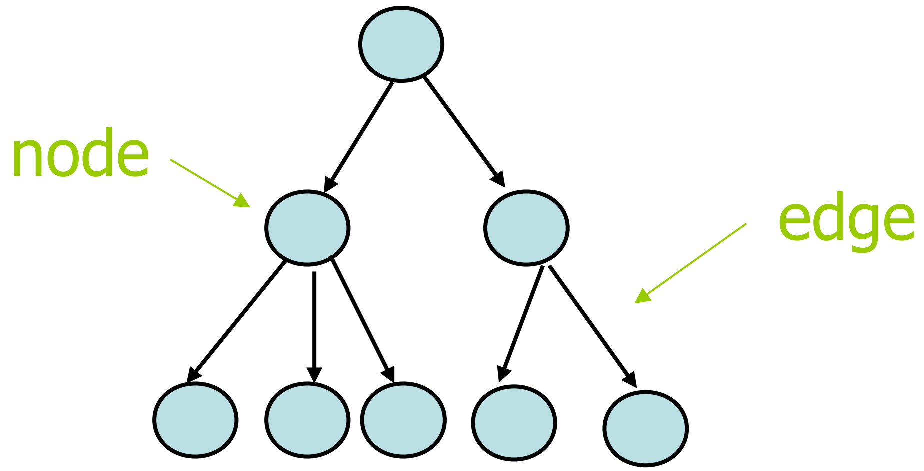


Introduction to Trees and Graphs

Trees

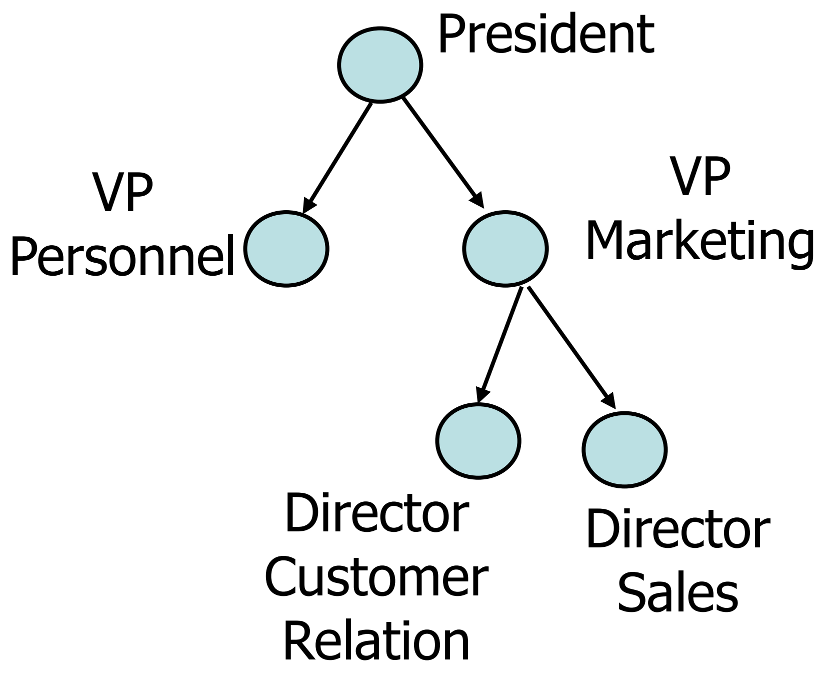
What is a tree?

- Trees are structures used to represent hierarchic al relationship
- Each tree consists of nodes and edges
- Each node represents an object
- Each edge represents the relationship between t wo nodes.

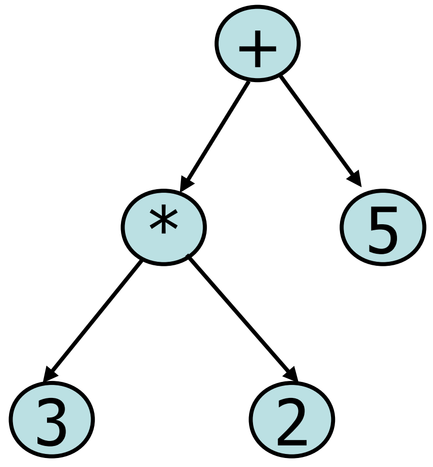


Some applications of Trees

Organization Chart

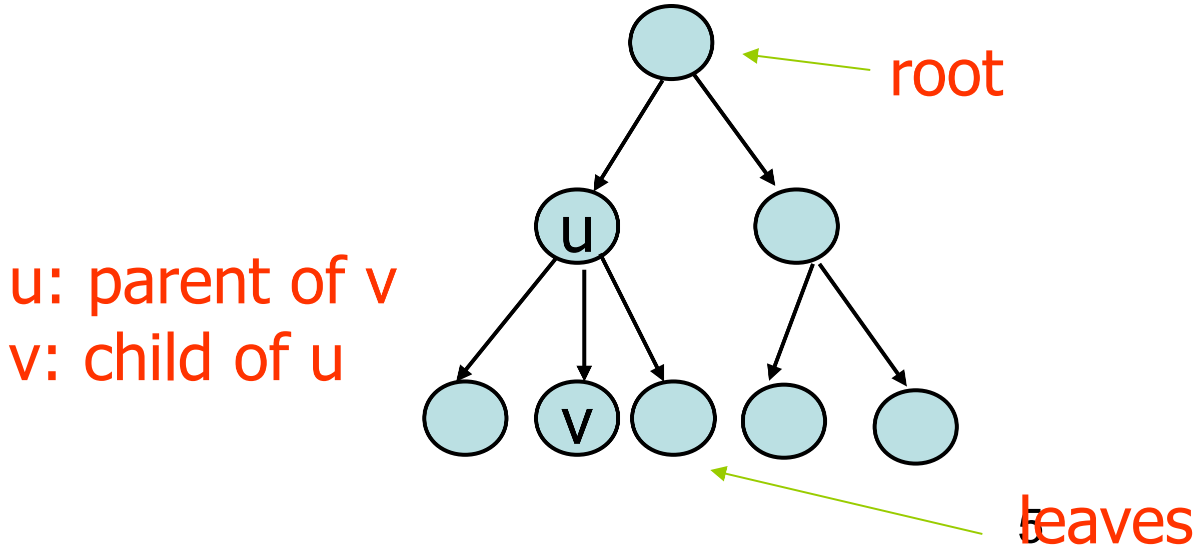


Expression Tree



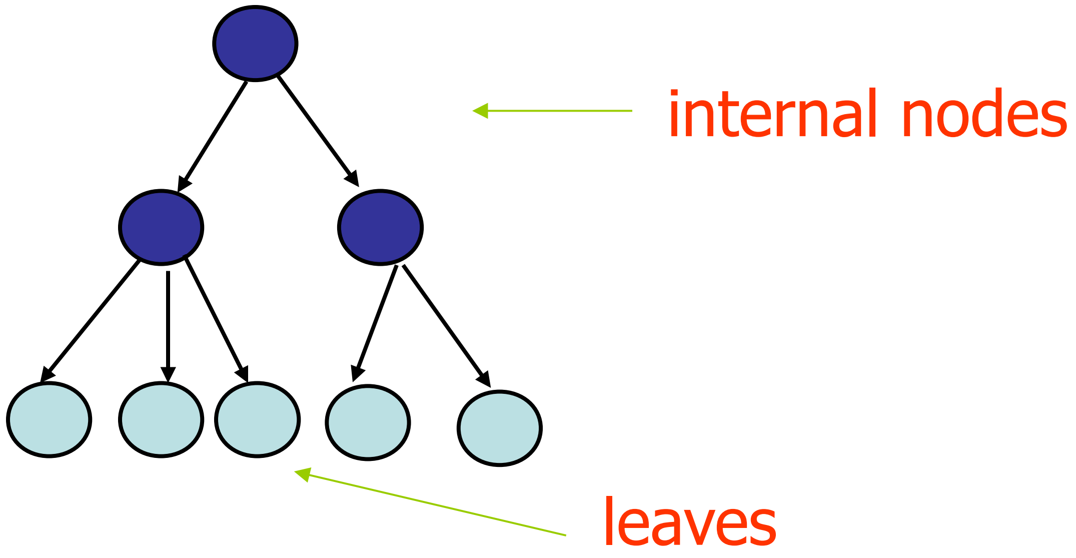
Terminology I

- For any two nodes u and v, if there is an edge po
inting from u to v, u is called the parent of v while
v is called the child of u. Such edge is denoted a
s (u, v).
- In a tree, there is exactly one node without paren
t, which is called the root. The nodes without chil
dren are called leaves.



Terminology II

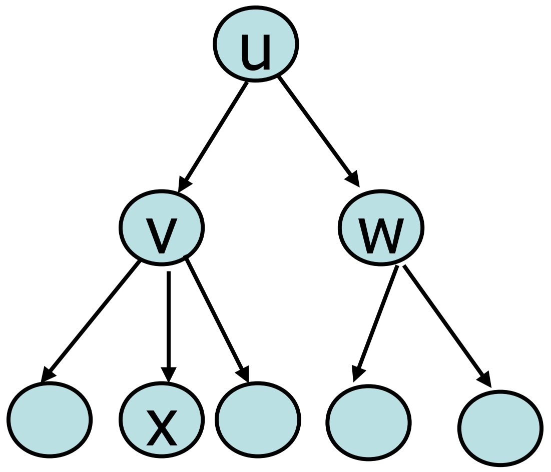
- In a tree, the nodes without children are ca lled leaves. Otherwise, they are called inte rnal nodes.



Terminology III

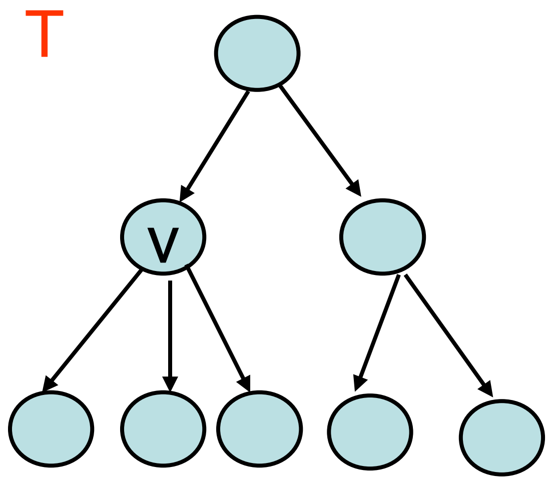
- If two nodes have the same parent, they are sibli ngs.
- A node u is an ancestor of v if u is parent of v or parent of parent of v or …
- A node v is a descendent of u if v is child of v or child of child of v or …

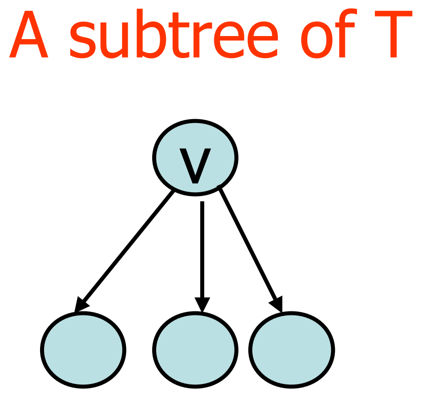
v and w are siblings
u and v are ancestors of x
v and x are descendents of u



Terminology IV

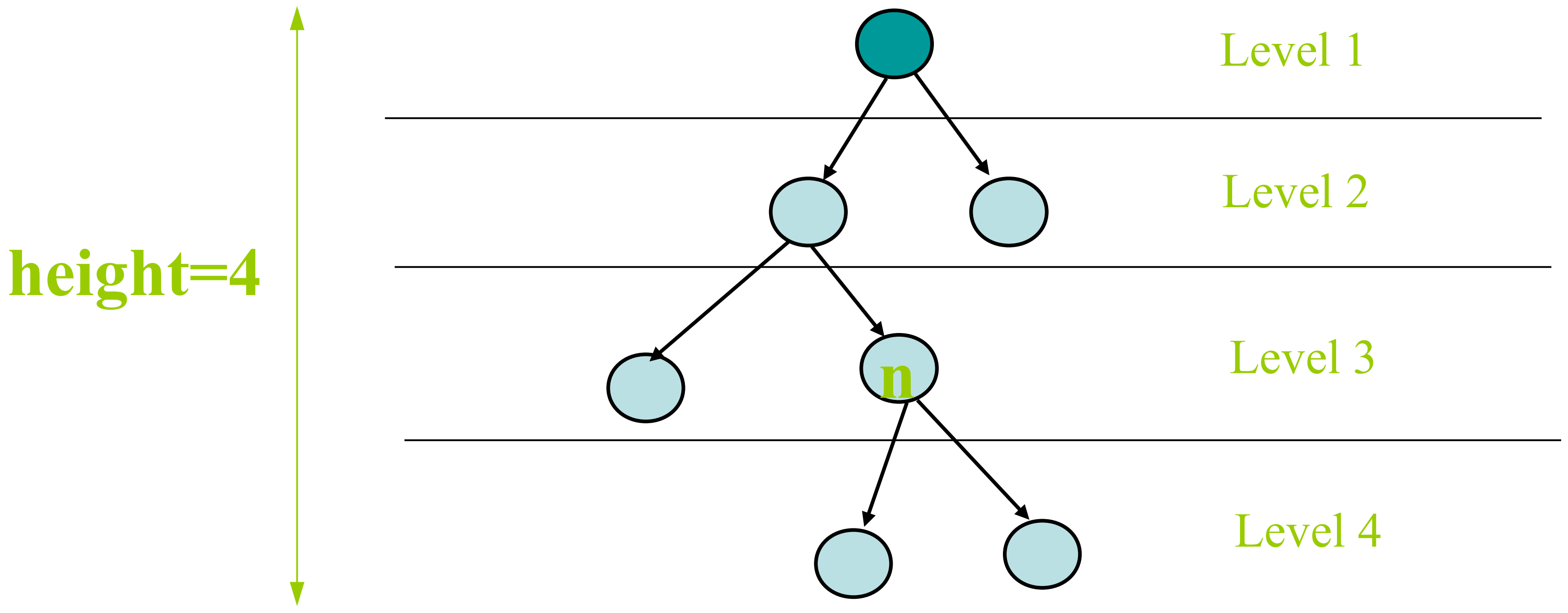
- A subtree is any node together with all its descendants.





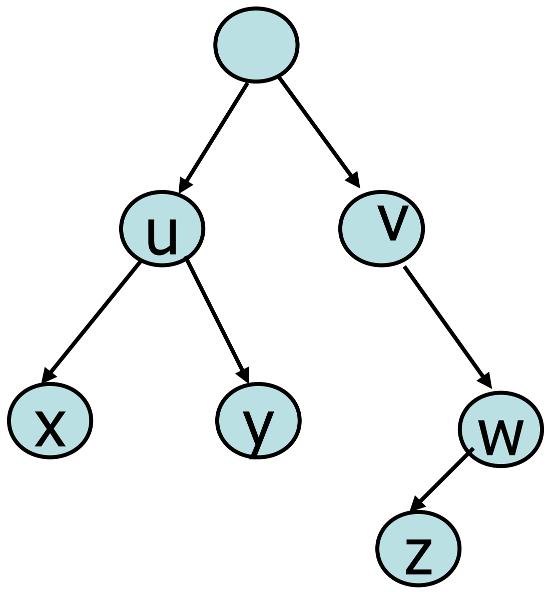
Terminology V

- Level of a node n: number of nodes on the path from roo t to node n
- Height of a tree: maximum level among all of its node



Binary Tree

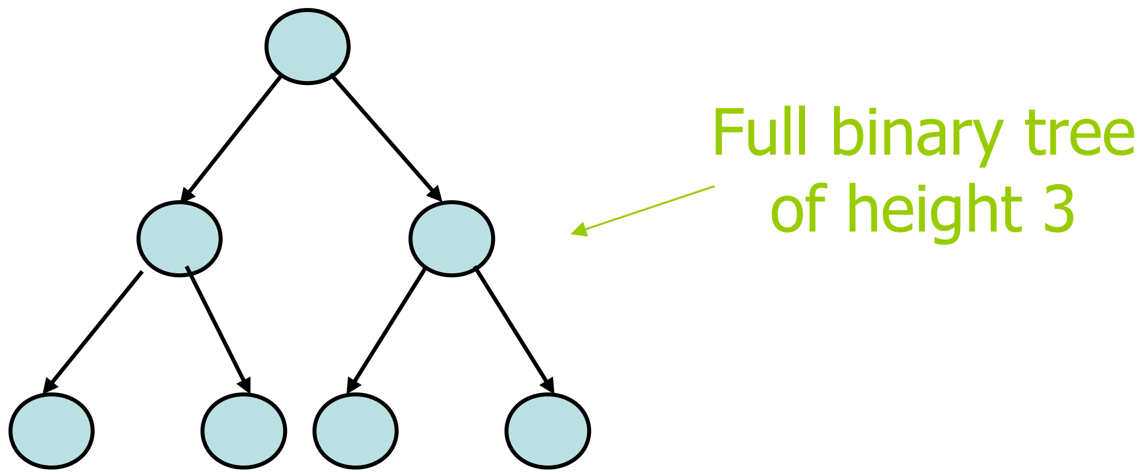
- Binary Tree: Tree in which every node has at mo st 2 children
- Left child of u: the child on the left of u
- Right child of u: the child on the right of u



x: left child of u
y: right child of u
w: right child of v
z: left child of w

Full binary tree

- If T is empty, T is a full binary tree of height 0.
- If T is not empty and of height h >0, T is a full binary tree if both subtrees of the root of T are full binary trees of height h-1.

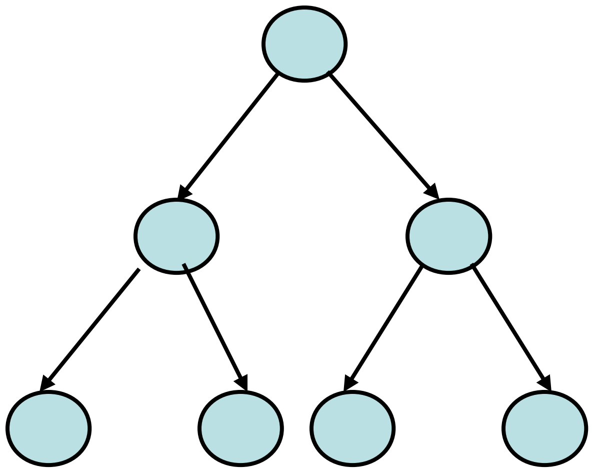


Property of binary tree (I)

- A full binary tree of height h has 2h-1 node s

No. of nodes = 20 + 21 + ... + 2(h-1)

= 2h – 1



Level 1: 20 nodes

Level 2: 21 nodes

Level 3: 22 nodes

Property of binary tree (II)

- Consider a binary tree T of height h. The n umber of nodes of T   2h-1

Reason: you cannot have more nodes than a full binary tree of height h.

Property of binary tree (III)

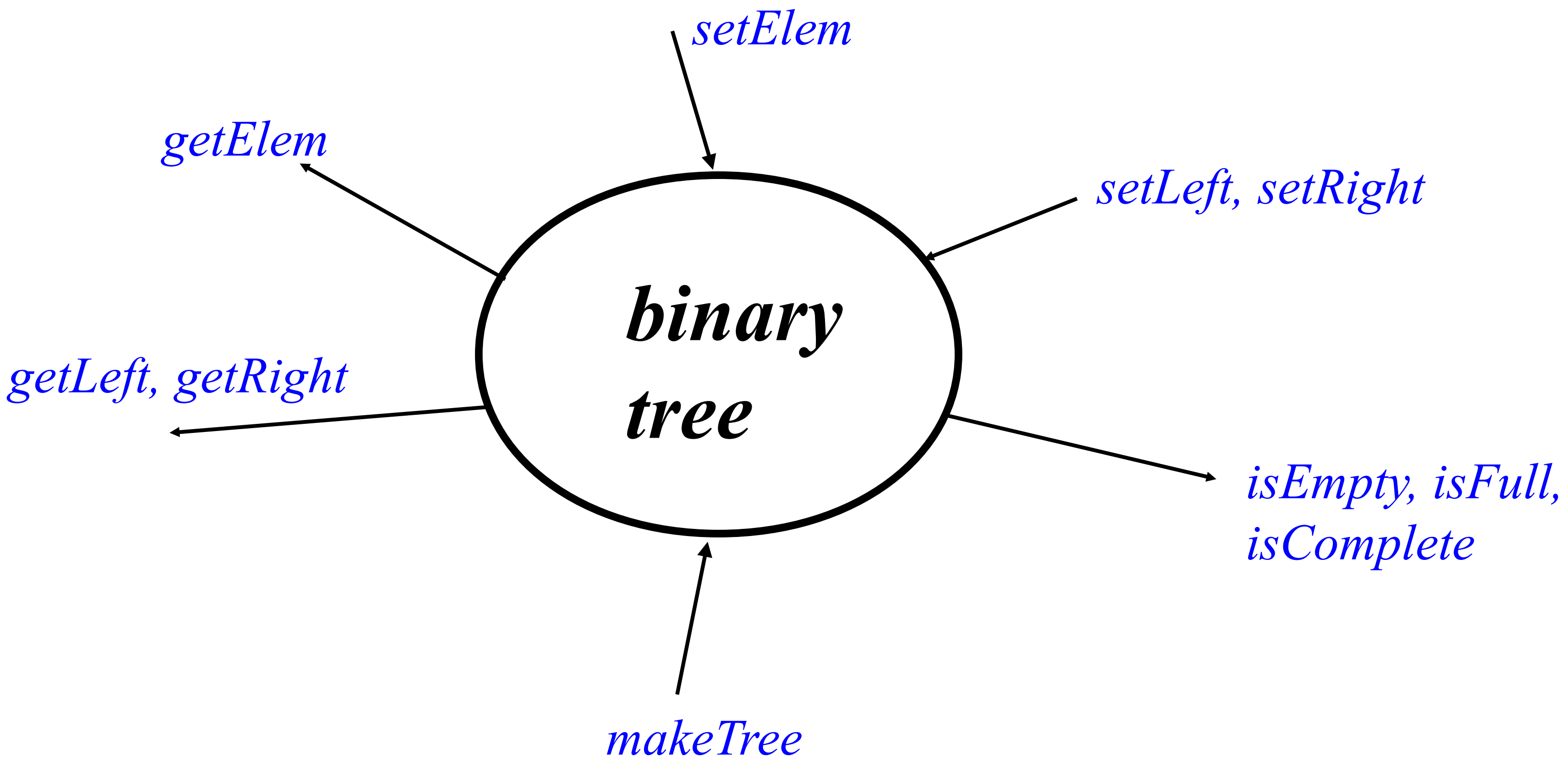
- The minimum height of a binary tree with n nodes is log(n+1)

By property (II), n   2h-1

Thus, 2h   n+1

That is, h   log2 (n+1)

Binary Tree ADT

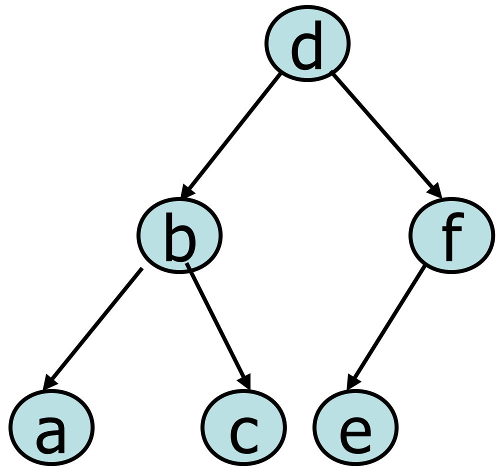


Representation of a Binary Tree

- An array-based representation
- A reference-based representation

An array-based representation

–1: empty tree



nodeNum	item	leftChild	rightChild
0	d	1	2
1	b	3	4
2	f	5	-1
3	a	-1	-1
4	c	-1	-1
5	e	-1	-1
6	?	?	?
7	?	?	?
8	?	?	?
9	?	?	?
...	.....	.....	....

root

$\boxed{0}$

free

$\boxed{6}$

Reference Based Representation

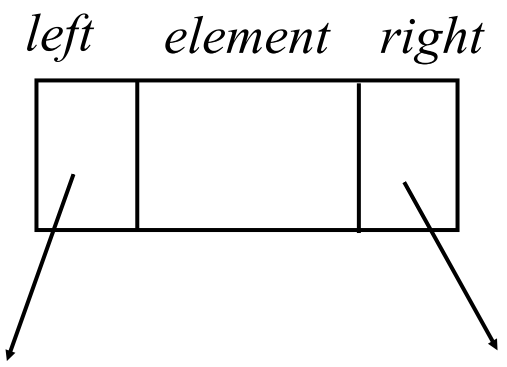
NULL: empty tree

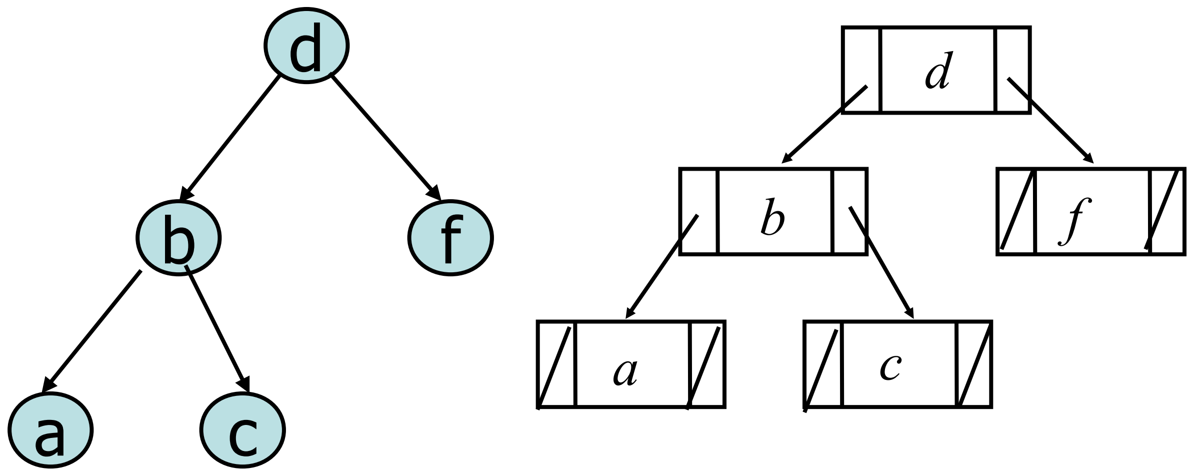
You can code this with a class of three fields:

Object element;

BinaryNode left;

BinaryNode right;





Tree Traversal

- Given a binary tree, we may like to do som e operations on all nodes in a binary tree. For example, we may want to double the v alue in every node in a binary tree.
- To do this, we need a traversal algorithm which visits every node in the binary tree.

Ways to traverse a tree

- There are three main ways to traverse a tree:
  - Pre-order:
    - (1) visit node, (2) recursively visit left subtree, (3) recursively visit right subtree
  - In-order:
    - (1) recursively visit left subtree, (2) visit node, (3) recursively right subtree
  - Post-order:
    - (1) recursively visit left subtree, (2) recursively visit right subtr ee, (3) visit node
  - Level-order:
    - Traverse the nodes level by level
- In different situations, we use different traversal algorithm.

Examples for expression tree

- By pre-order, (prefix)

+ * 2 3 / 8 4

- By in-order, (infix)

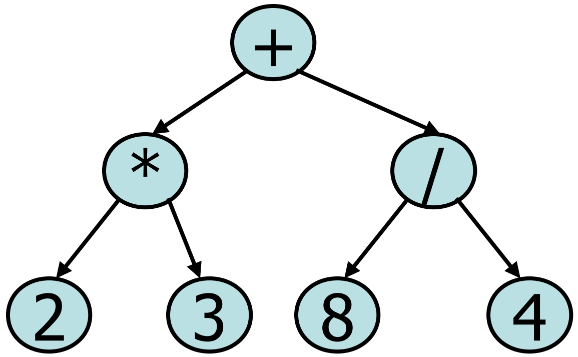
2 * 3 + 8 / 4

- By post-order, (postfix)

2 3 * 8 4 / +

- By level-order,

+ * / 2 3 8 4



- Note 1: Infix is what we read!
- Note 2: Postfix expression can be computed efficiently using stack

Pre-order

```
Algorithm pre-order(BTree x)
If (x is not empty) {
    print x.getItem();          // you can do other things!
    pre-order(x.getLeftChild());
    pre-order(x.getRightChild());
}
```

Pre-order example

```
Pre-order(a);
```

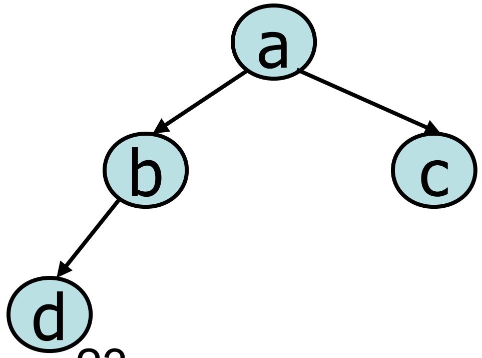
```
Print a;
Pre-order(b);
Pre-order(c);
```

```
Print b;
Pre-order(d);
Pre-order(null);
```

```
Print d;
Pre-order(null);
Pre-order(null);
```

```
Print c;
Pre-order(null);
Pre-order(null);
```

a b d c



Time complexity of Pre-order Tr aversal

- For every node x, we will call pre-order(x) one time, which performs O (1) operations.
- Thus, the total time = O(n).

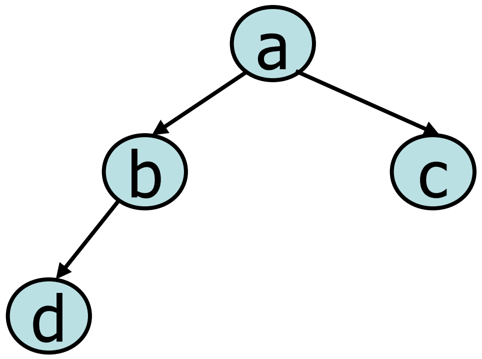
In-order and post-order

```
Algorithm in-order(BTree x)
If (x is not empty) {
    in-order(x.getLeftChild());
    print x.getItem(); // you can do other things!
    in-order(x.getRightChild());
}

Algorithm post-order(BTree x)
If (x is not empty) {
    post-order(x.getLeftChild());
    post-order(x.getRightChild());
    print x.getItem(); // you can do other things!
}
```

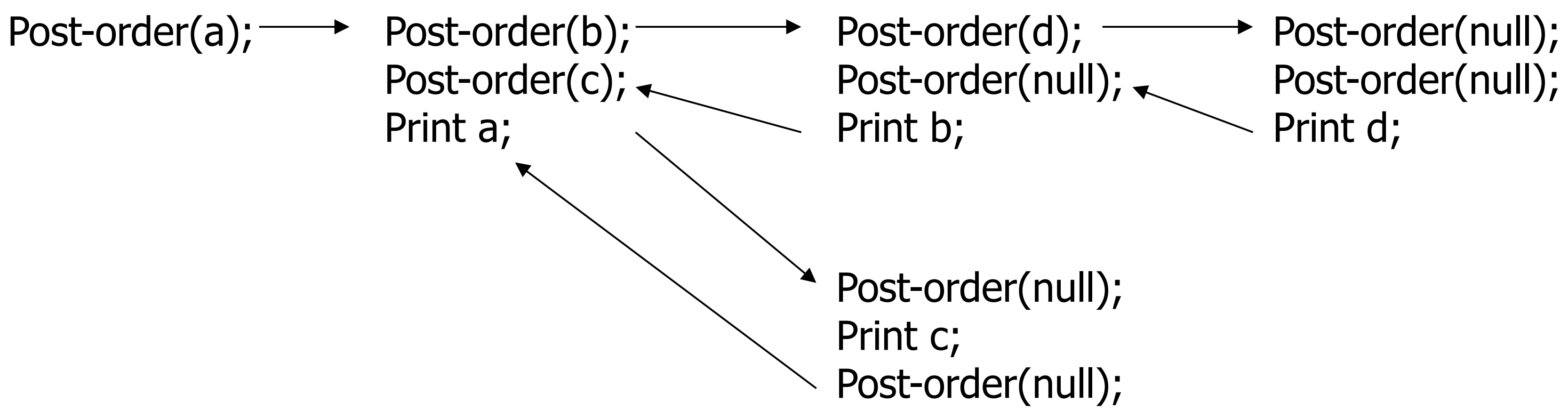
In-order example

```
In-order(a);

In-order(b);
Print a;
In-order(c);

In-order(d);
Print b;
In-order(null);

In-order(null);
Print d;
In-order(null);

In-order(null);
Print c;
In-order(null);
```

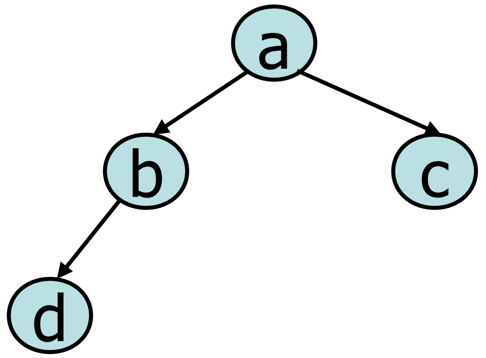
d b a c



Post-order example



d b c a



Time complexity for in-order and post-order

- Similar to pre-order traversal, the time co mplexity is O(n).

Level-order

- Level-order traversal requires a queue!

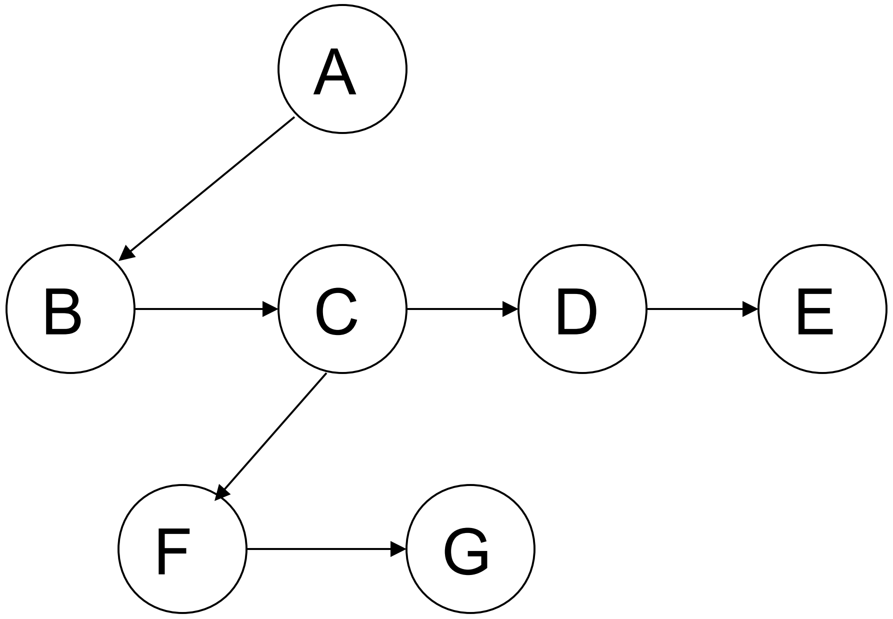
```
Algorithm level-order(BTree t)
 Queue Q = new Queue();
 BTree n;

 Q.enqueue(t);   // insert pointer t into Q

 while (! Q.empty()){
   n = Q.dequeue(); //remove next node from the front of Q

   if (!n.isEmpty()){
       print n.getItem();      // you can do other things
       Q.enqueue(n.getLeft());  // enqueue left subtree on rear of Q
       Q.enqueue(n.getRight()); // enqueue right subtree on rear of Q
   };
 };
```

Time complexity of Level-order t raversal

- Each node will enqueue and dequeue one time.
- For each node dequeued, it only does one print operation!
- Thus, the time complexity is O(n).

General tree implementation

```
struct TreeNode
{
      Object      element
      TreeNode *firstChild
      TreeNode *nextsibling
}
```



because we do not know how many children a node has in advance.

- Traversing a general tree is similar to traversing a binary tree

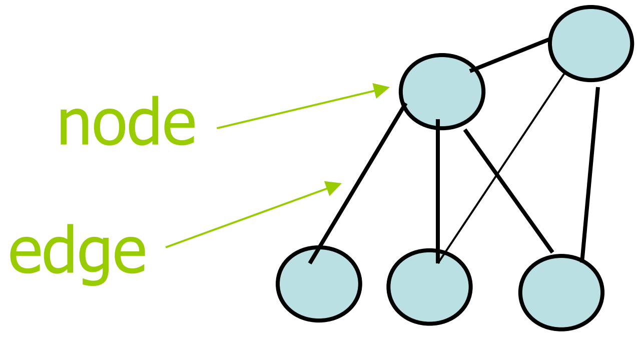
Summary

- We have discussed
  - the tree data-structure.
  - Binary tree vs general tree
  - Binary tree ADT
    - Can be implemented using arrays or references
  - Tree traversal
    - Pre-order, in-order, post-order, and level-order

Graphs

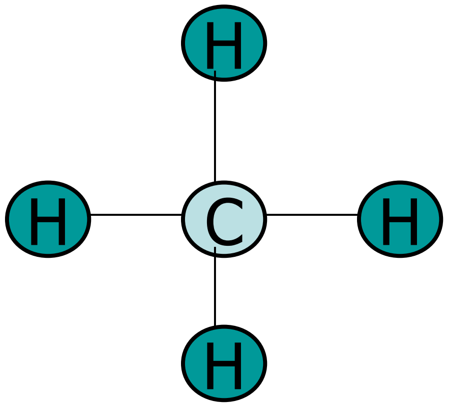
What is a graph?

- Graphs represent the relationships among data i tems
- A graph G consists of
  - a set V of nodes (vertices)
  - a set E of edges: each edge connects two nodes
- Each node represents an item
- Each edge represents the relationship between t wo items

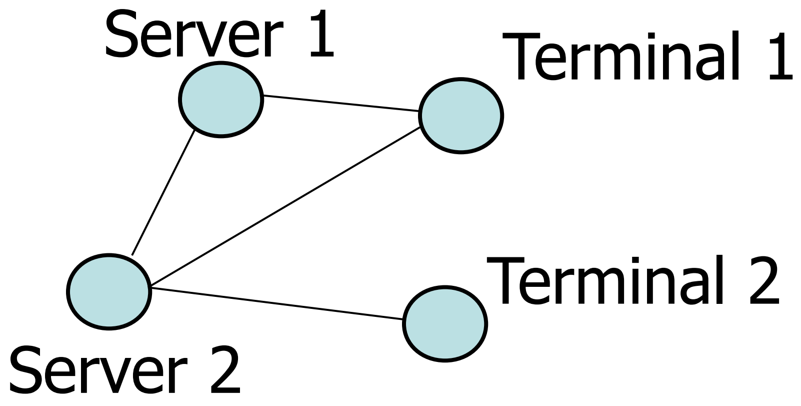


Examples of graphs

Molecular Structure



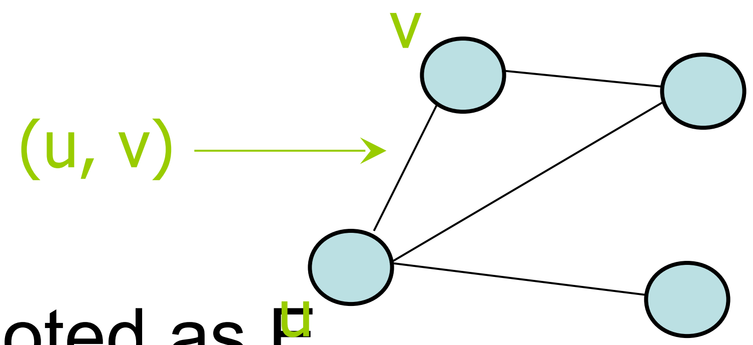
Computer Network



Other examples: electrical and communication networks, airline routes, flow chart, graphs for planning projects

Formal Definition of graph

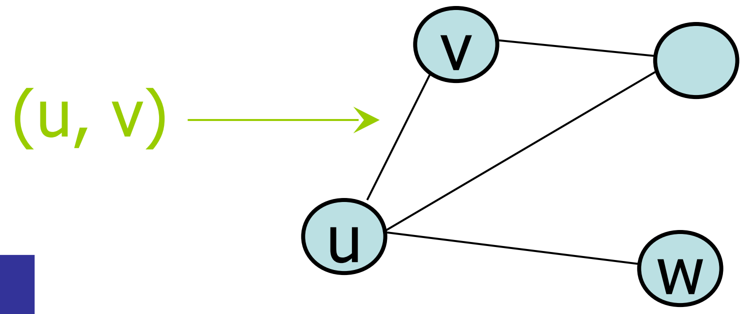
- The set of nodes is denoted as V
- For any nodes u and v, if u and v are connect ed by an edge, such edge is denoted as (u, v)



- The set of edges is denoted as E
- A graph G is defined as a pair (V, E)

Adjacent

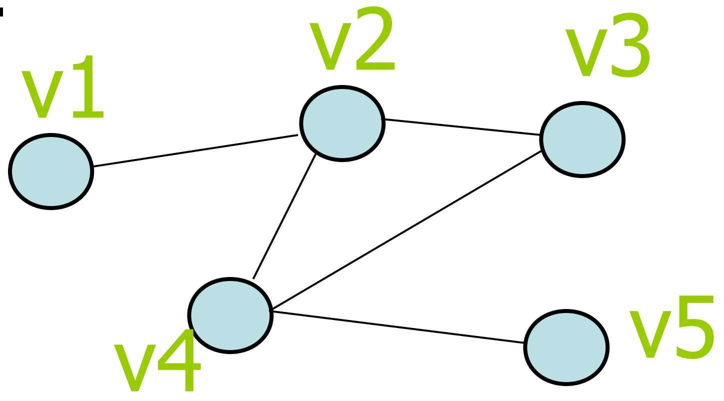
- Two nodes u and v are said to be adjacent if (u, v)   E



u and v are adjacent
v and w are not adjacent

Path and simple path

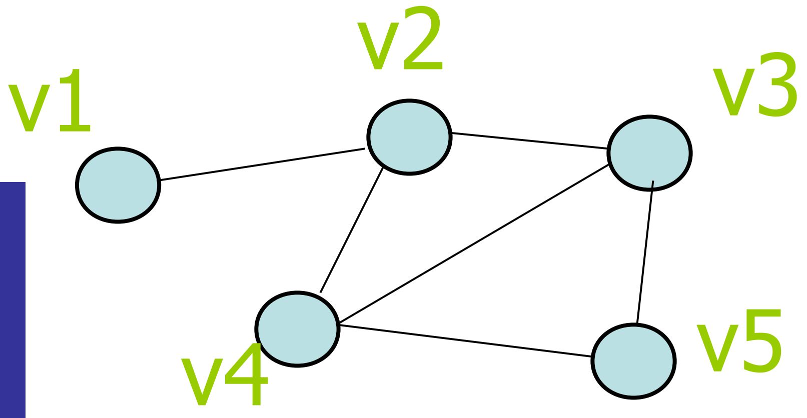
- A path from v1 to vk is a sequence of node s v1, v2, …, vk that are connected by edg es (v1, v2), (v2, v3), …, (vk-1, vk)
- A path is called a simple path if every nod e appears at most once.



- v2, v3, v4, v2, v1 is a path
- v2, v3, v4, v5 is a path, also it is a simple path

Cycle and simple cycle

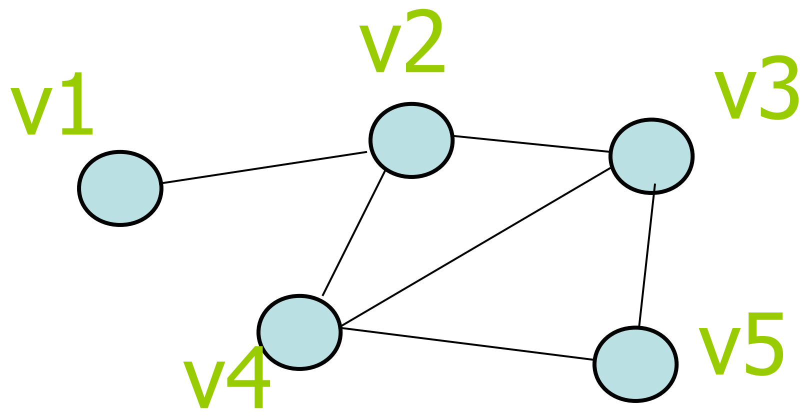
- A cycle is a path that begins and ends at t he same node
- A simple cycle is a cycle if every node app ears at most once, except for the first and the last nodes



- v2, v3, v4, v5 , v3, v2 is a cycle
- v2, v3, v4, v2 is a cycle, it is also a simple cycle

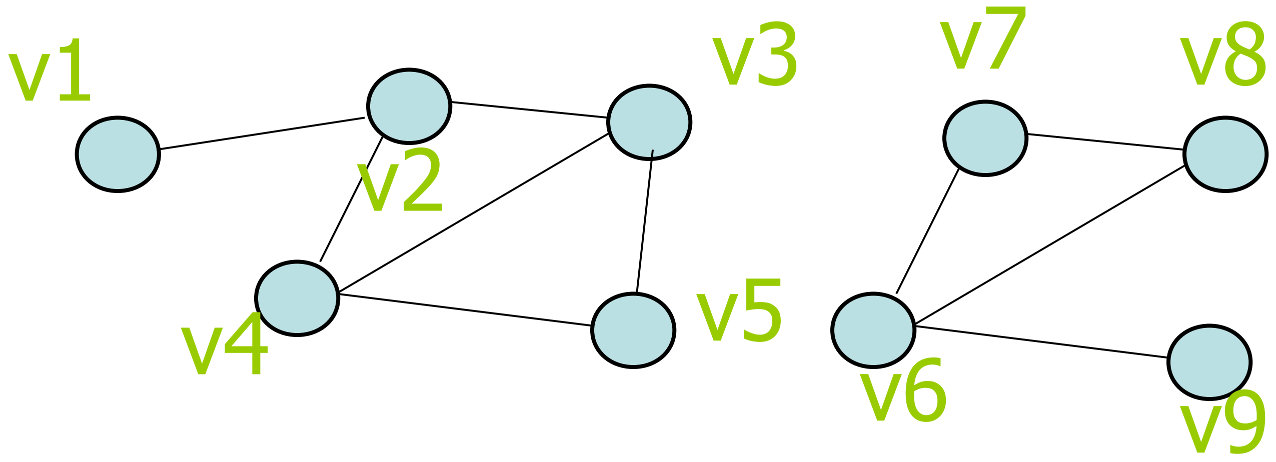
Connected graph

- A graph G is connected if there exists path between every pair of distinct nodes; other wise, it is disconnected



This is a connected graph because there exists path between every pair of nodes

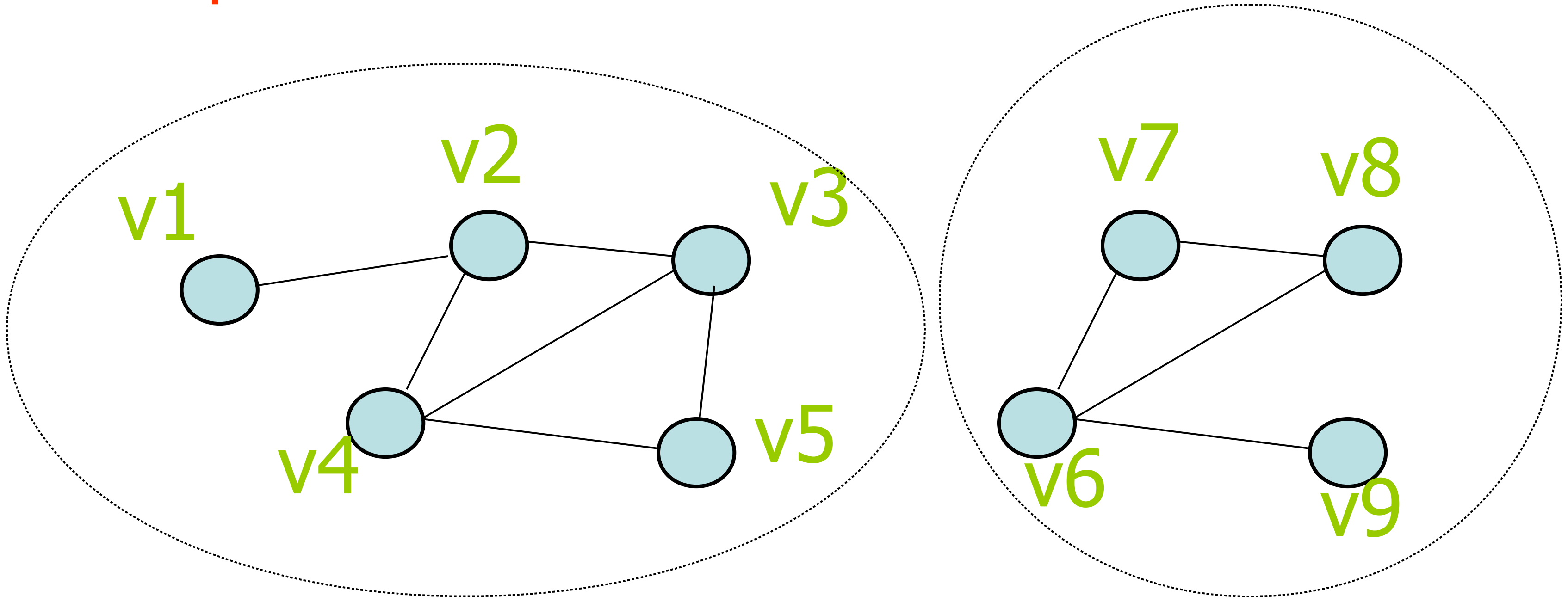
Example of disconnected graph



This is a disconnected graph because there does not exist path between some pair of nodes, says, v1 and v7

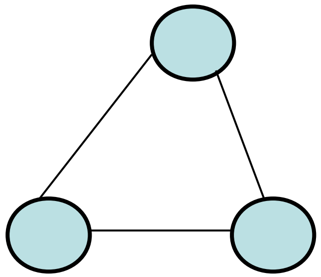
Connected component

- If a graph is disconnect, it can be partitioned into a number of graphs such that each of them is co nnected. Each such graph is called a connected component.

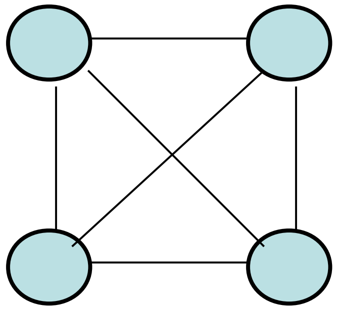


Complete graph

- A graph is complete if each pair of distinct nodes has an edge



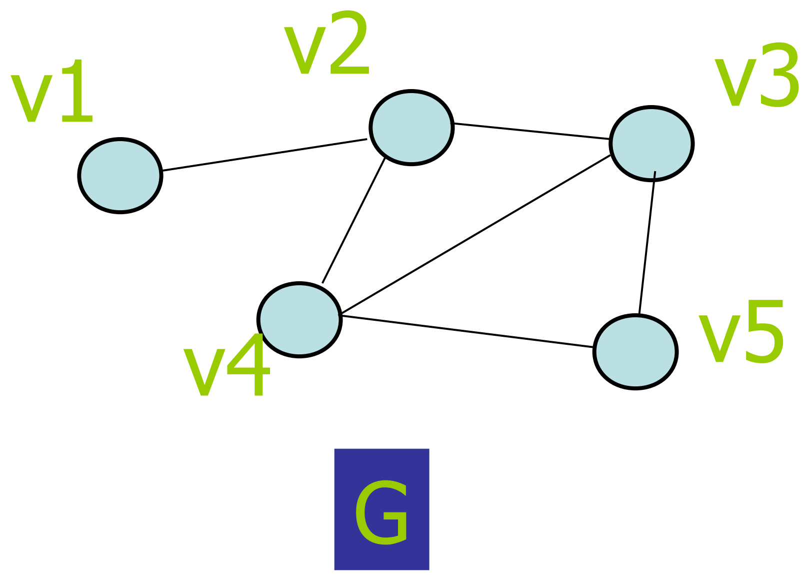
Complete graph with 3 nodes

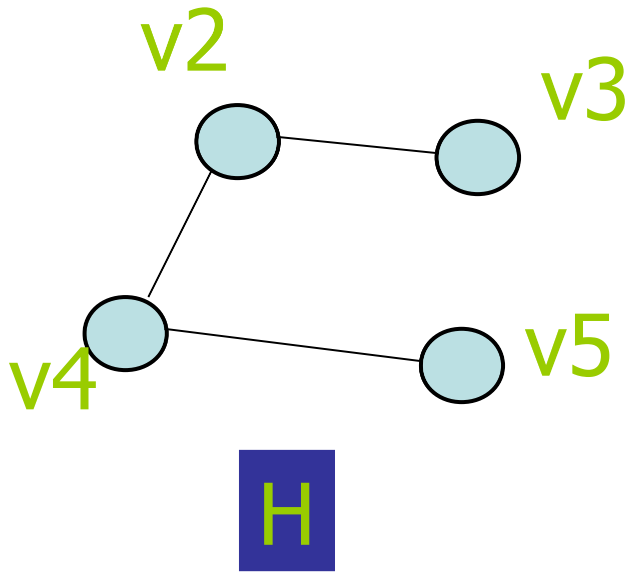


Complete graph with 4 nodes

Subgraph

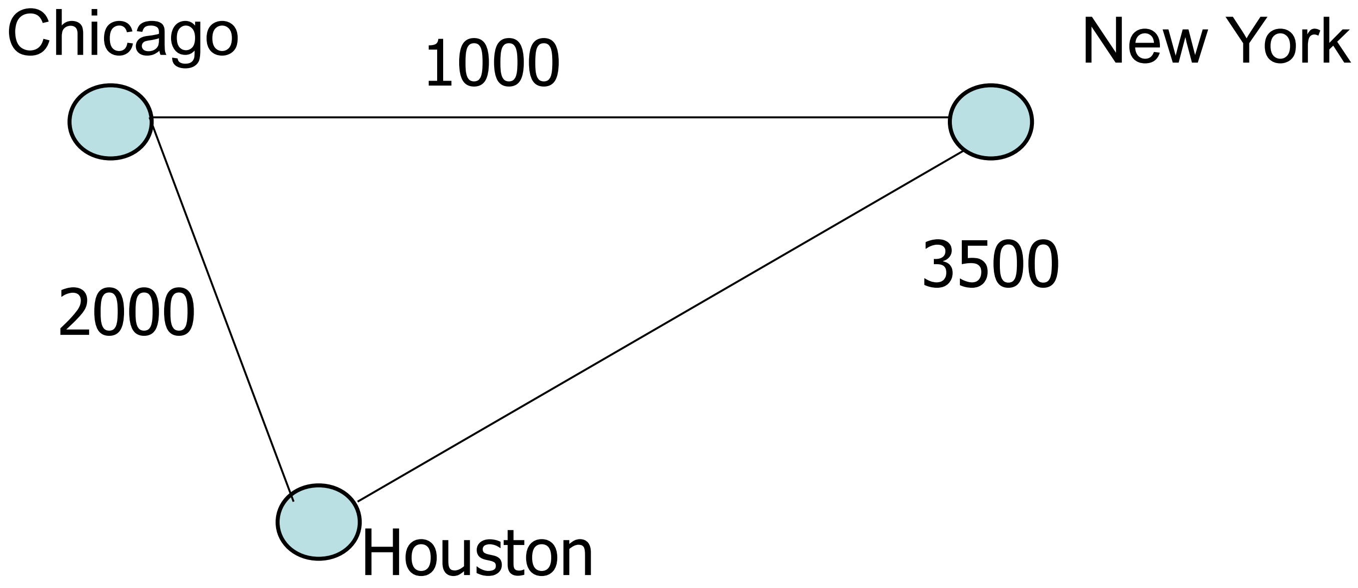
- A subgraph of a graph G =(V, E) is a graph H = (U, F) such that U   V and F   E.





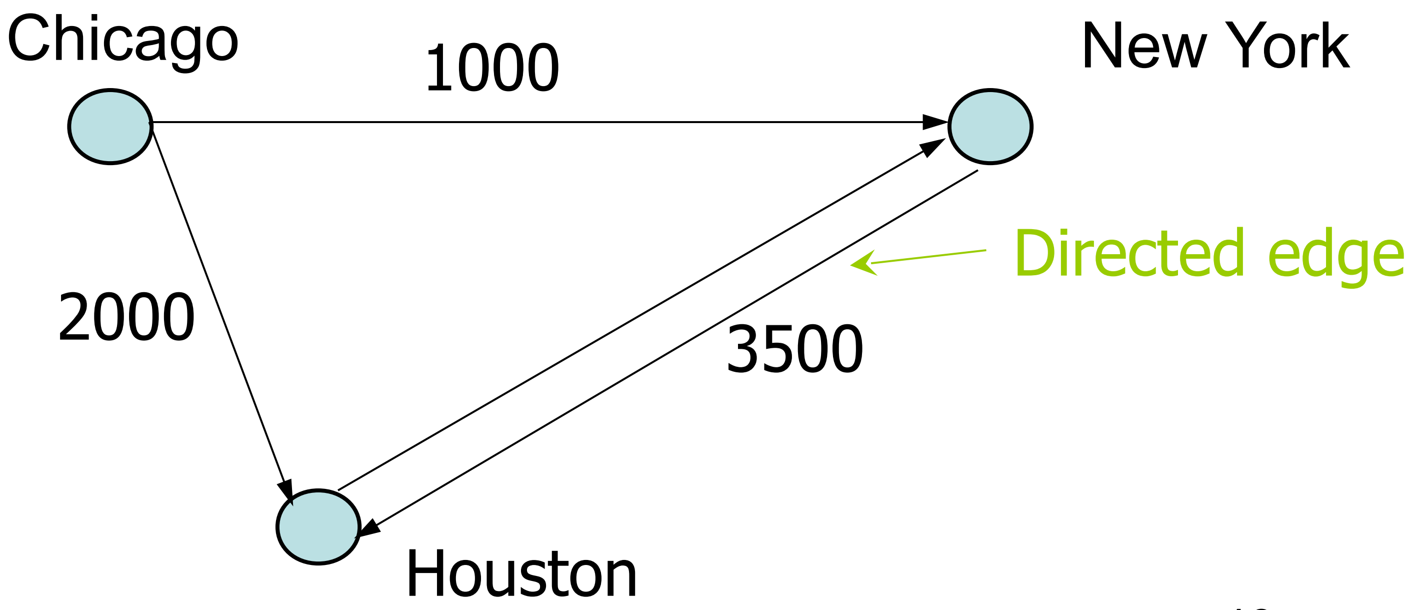
Weighted graph

- If each edge in G is assigned a weight, it i s called a weighted graph

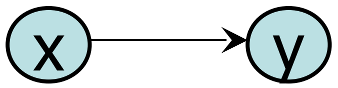


Directed graph (digraph)

- All previous graphs are undirected graph
- If each edge in E has a direction, it is called a directed e dge
- A directed graph is a graph where every edges is a direc ted edge



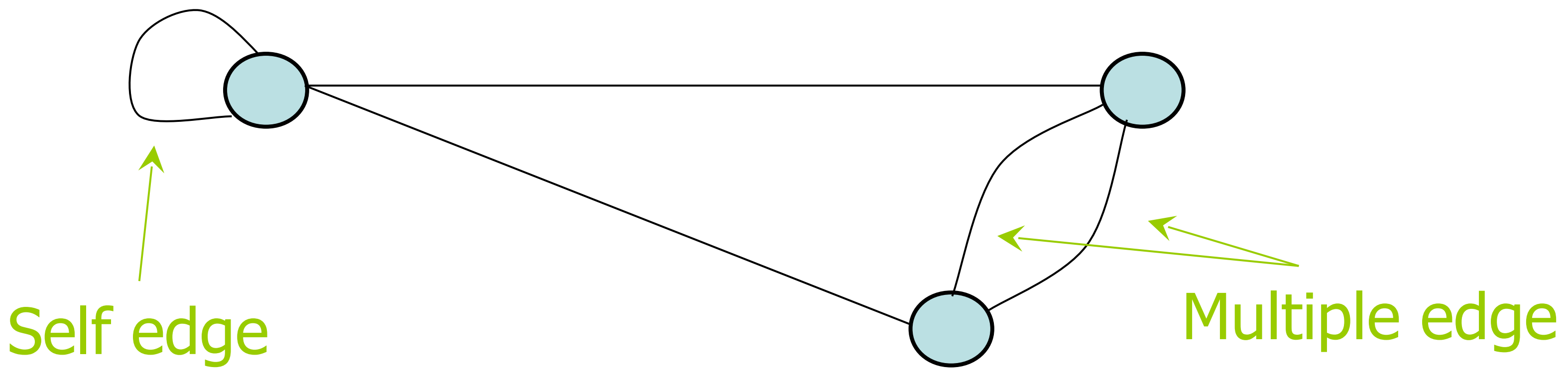
More on directed graph



- If (x, y) is a directed edge, we say
  - y is adjacent to x
  - y is successor of x
  - x is predecessor of y
- In a directed graph, directed path, directed cycle can be defined similarly

Multigraph

- A graph cannot have duplicate edges.
- Multigraph allows multiple edges and self edge (or loop).



Property of graph

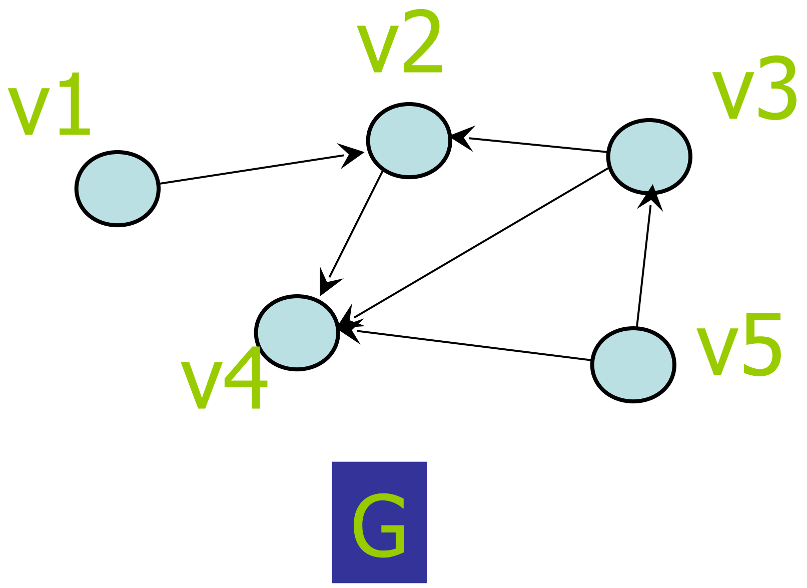
- A undirected graph that is connected and has no cycle is a tree.
- A tree with n nodes have exactly n-1 edge s.
- A connected undirected graph with n node s must have at least n-1 edges.

Implementing Graph

- Adjacency matrix
  - Represent a graph using a two-dimensional ar ray
- Adjacency list
  - Represent a graph using n linked lists where n is the number of vertices

Adjacency matrix for directed graph

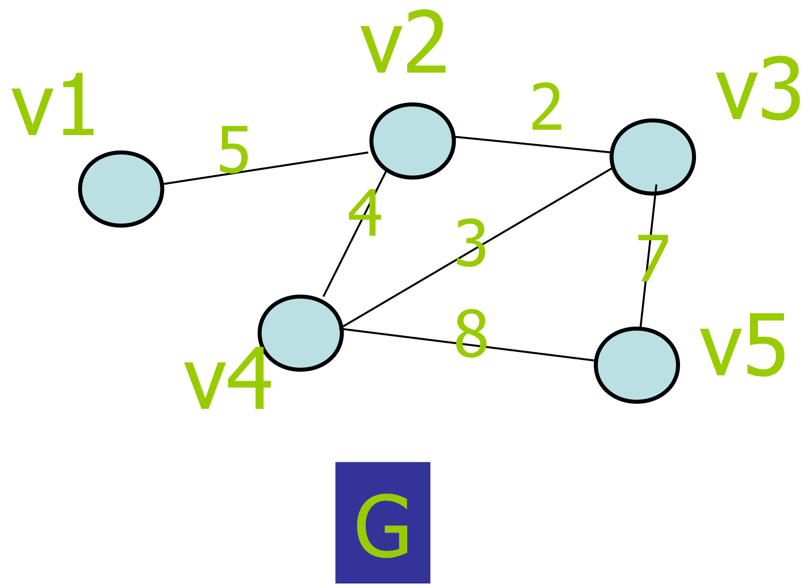
Matrix[i][j] = 1 if (vi, vj) E
0 if (vi, vj) E



		1	2	3	4	5
		v1	v2	v3	v4	v5
1	v1	0	1	0	0	0
2	v2	0	0	0	1	0
3	v3	0	1	0	1	0
4	v4	0	0	0	0	0
5	v5	0	0	1	1	0

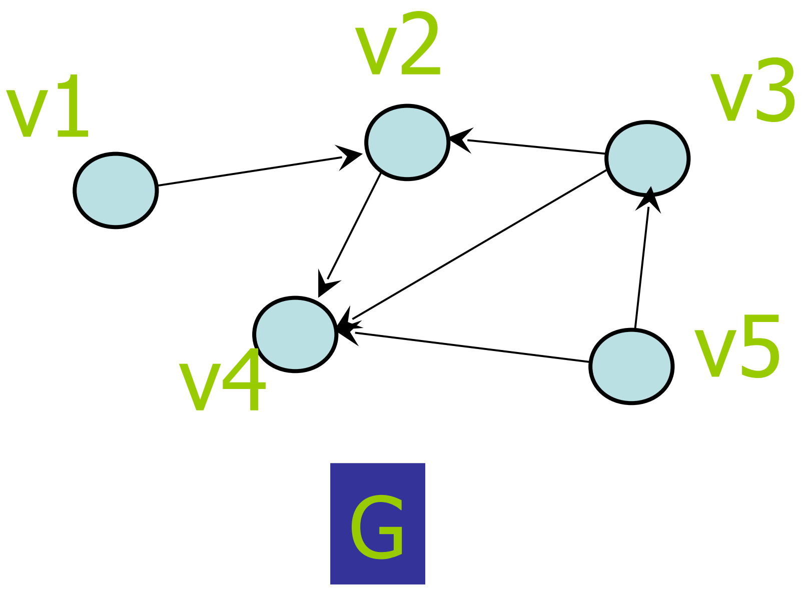
Adjacency matrix for weighted undir ected graph

Matrix[i][j] = w(vi, vj)   if (vi, vj)   E or (vj, vi)   E
∞   otherwise



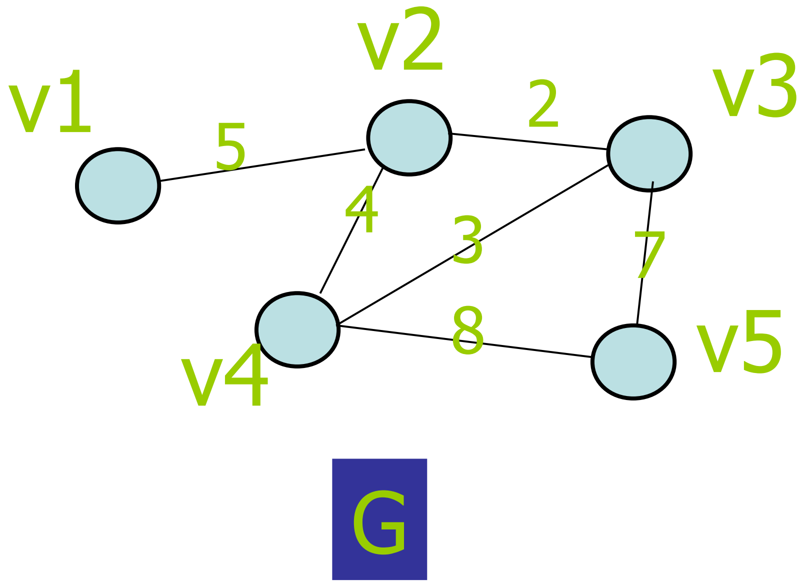
	1 v1	2 v2	3 v3	4 v4	5 v5
1 v1	∞	5	∞	∞	∞
2 v2	5	∞	2	4	∞
3 v3	0	2	∞	3	7

Adjacency list for directed graph



1	v1	v2	
2	v2	v4	
3	v3	v2	v4
4	v4		
5	v5	v3	v4

Adjacency list for weighted undirected graph



1	v 1	v2(5)		
2	v 2	v1(5)	v3(2)	v4(4)
3	v 3	v2(2)	v4(3)	v5(7)
4	v 4	v2(4)	v3(3)	v5(8)
5	v 5	v3(7)	v4(8)	

Pros and Cons

- Adjacency matrix
  - Allows us to determine whether there is an ed ge from node i to node j in O(1) time
- Adjacency list
  - Allows us to find all nodes adjacent to a given node j efficiently
  - If the graph is sparse, adjacency list requires l ess space

Problems related to Graph

- Graph Traversal
- Topological Sort
- Spanning Tree
- Minimum Spanning Tree
- Shortest Path

Graph Traversal Algorithm

- To traverse a tree, we use tree traversal algorith ms like pre-order, in-order, and post-order to visit all the nodes in a tree
- Similarly, graph traversal algorithm tries to visit a ll the nodes it can reach.
- If a graph is disconnected, a graph traversal that begins at a node v will visit only a subset of node s, that is, the connected component containing v.

Two basic traversal algorithms

- Two basic graph traversal algorithms:
  - Depth-first-search (DFS)
    - After visit node v, DFS strategy proceeds along a p
ath from v as deeply into the graph as possible bef
ore backing up
  - Breadth-first-search (BFS)
    - After visit node v, BFS strategy visits every node a
djacent to v before visiting any other nodes

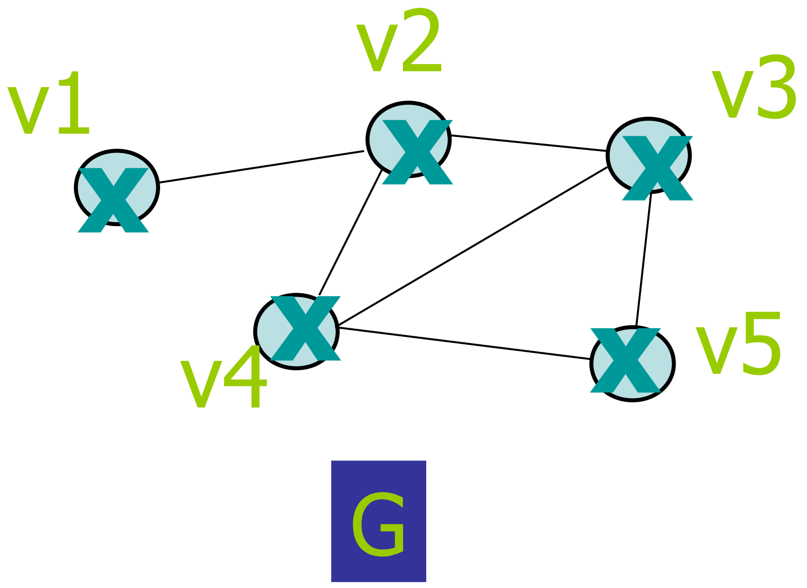
Depth-first search (DFS)

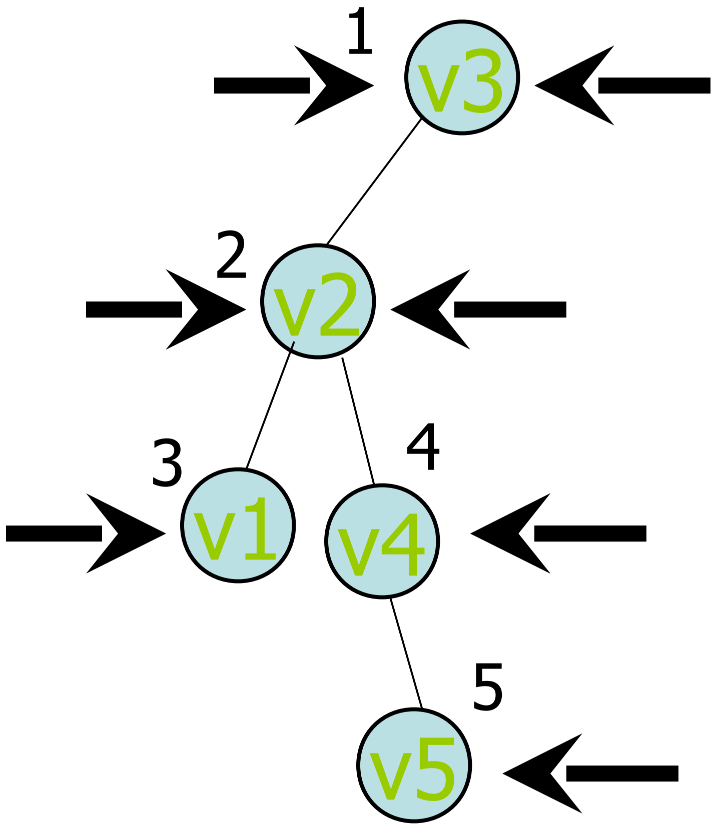
- DFS strategy looks similar to pre-order. From a given no de v, it first visits itself. Then, recursively visit its unvisited neighbours one by one.
- DFS can be defined recursively as follows.

```
Algorithm dfs(v)
print v; // you can do other things!
mark v as visited;
for (each unvisited node u adjacent to v)
    dfs(u);
```

DFS example

- Start from v3



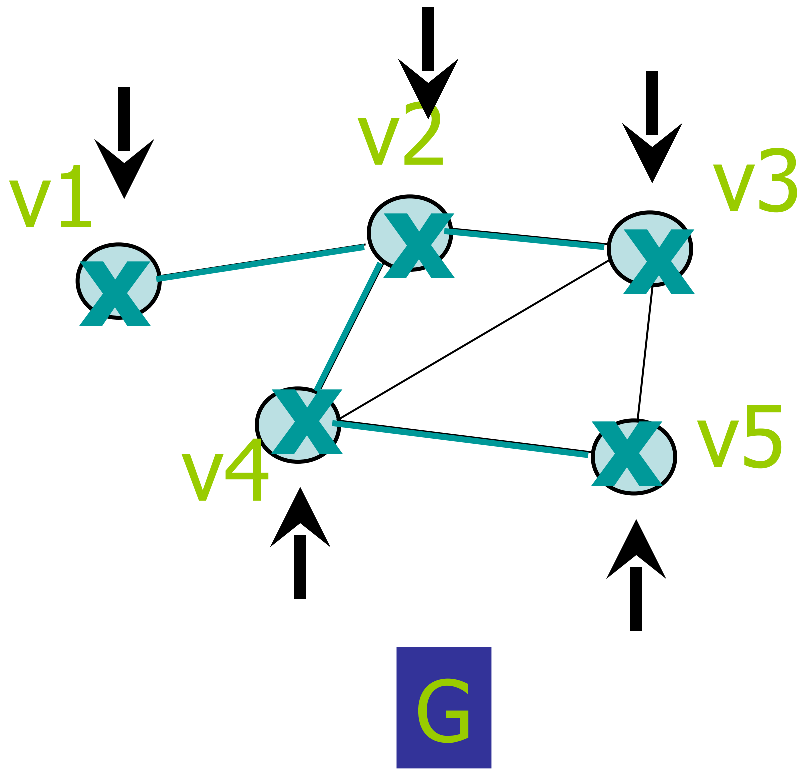


Non-recursive version of DFS al gorithm

```
Algorithm dfs(v)
s.createStack();
s.push(v);
mark v as visited;
while (!s.isEmpty()) {
    let x be the node on the top of the stack s;
    if (no unvisited nodes are adjacent to x)
        s.pop(); // blacktrack
    else {
        select an unvisited node u adjacent to x;
        s.push(u);
        mark u as visited;
    }
}
```

Non-recursive DFS example

visit	stack
v3	v3
v2	v3, v2
v1	v3, v2, v1
backtrack	v3, v2
v4	v3, v2, v4
v5	v3, v2, v4 , v5
backtrack	v3, v2, v4
backtrack	v3, v2
backtrack	v3
backtrack	empty



Breadth-first search (BFS)

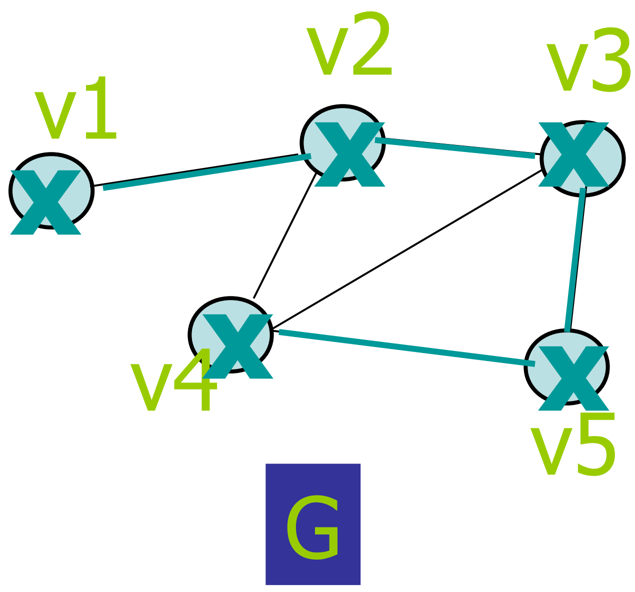
- BFS strategy looks similar to level-order. From a given node v, it first visits itself. Then, it visits ev ery node adjacent to v before visiting any other n odes.
  - 1. Visit v
  - 2. Visit all v’s neigbours
  - 3. Visit all v’s neighbours’ neighbours
  - …
- Similar to level-order, BFS is based on a queue.

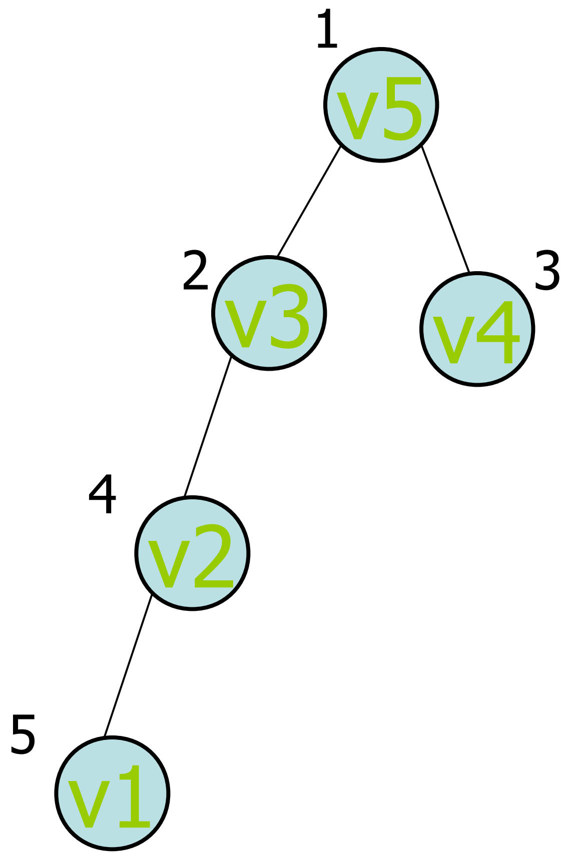
Algorithm for BFS

```
**Algorithm bfs(v)**
q.createQueue();
q.enqueue(v);
mark v as visited;
while(!q.isEmpty()) {
    w = q.dequeue();
    for (each unvisited node u adjacent to w) {
        q.enqueue(u);
        mark u as visited;
    }
}
```

BFS example

- Start from v5

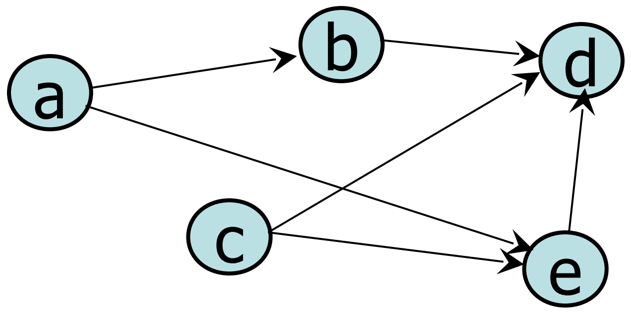




Visit	Queue (front to back)
v5	v5
	empty
v3	v3
v4	v3, v4
	v4
v2	v4, v2
	v2
	empty
v1	v1
	empty

Topological order

- Consider the prerequisite structure for courses:



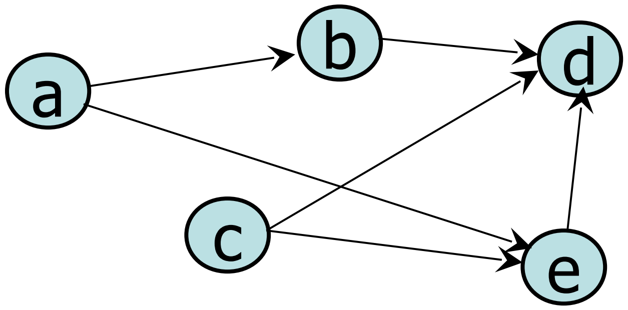
- Each node x represents a course x
- (x, y) represents that course x is a prerequisite to course y
- Note that this graph should be a directed graph without cycles (calle d a directed acyclic graph).
- A linear order to take all 5 courses while satisfying all prerequisites i s called a topological order.
- E.g.
  - a, c, b, e, d
  - c, a, b, e, d

Topological sort

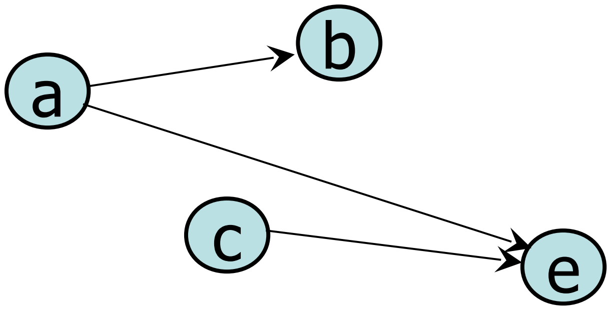
- Arranging all nodes in the graph in a topological order

```
Algorithm topSort
n = |V|;
for i = 1 to n {
    select a node v that has no successor;
    aList.add(1, v);
    delete node v and its edges from the graph;
}
return aList;
```

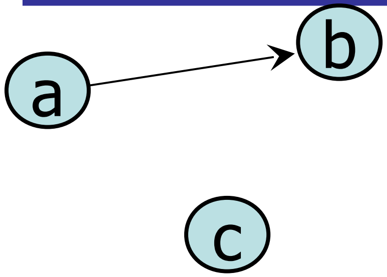
Example



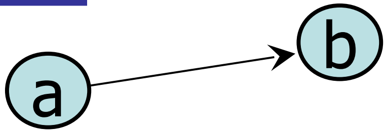
1. d has no successor! Choose d!



2. Both b and e have no successor! Choose e!



3. Both b and c have no successor! Choose c!



4. Only b has no successor! Choose b!

5. Choose a!
The topological order is **a,b,c,e,d**

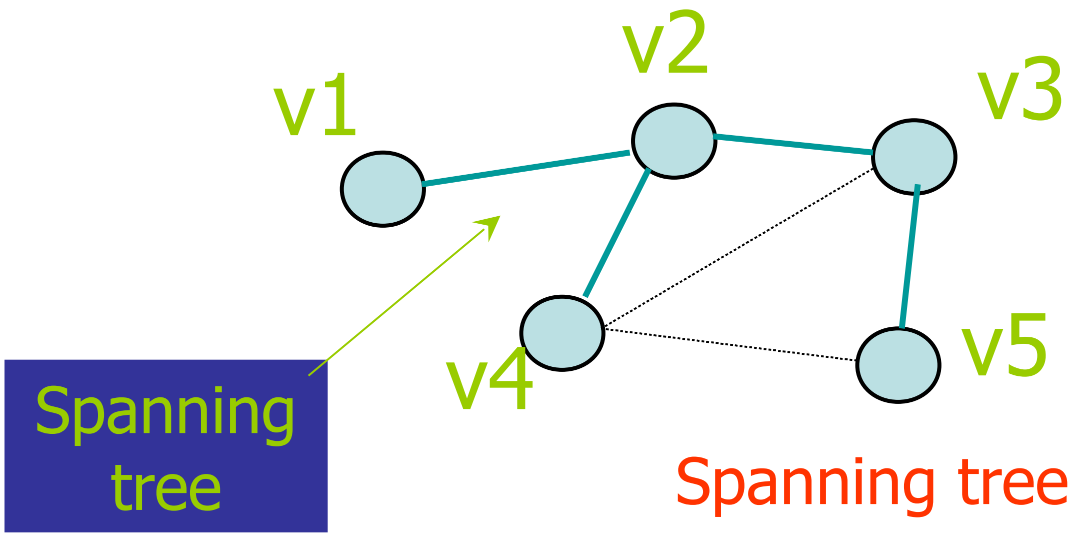
Topological sort algorithm 2

- This algorithm is based on DFS

```
Algorithm topSort2
s.createStack();
for (all nodes v in the graph) {
        if (v has no predecessors) {
                s.push(v);
                mark v as visited;
        }
}
while (!s.isEmpty()) {
        let x be the node on the top of the stack s;
        if (no unvisited nodes are adjacent to x) { // i.e. x has no unvisited successor
                aList.add(1, x);
                s.pop(); // blacktrack
        } else {
                select an unvisited node u adjacent to x;
                s.push(u);
                mark u as visited;
        }
}
return aList;
```

Spanning Tree

- Given a connected undirected graph G, a spanning tree of G is a subgraph of G that contains all of G's nodes and enough of its edges to form a tree.



Spanning tree is not unique!

DFS spanning tree

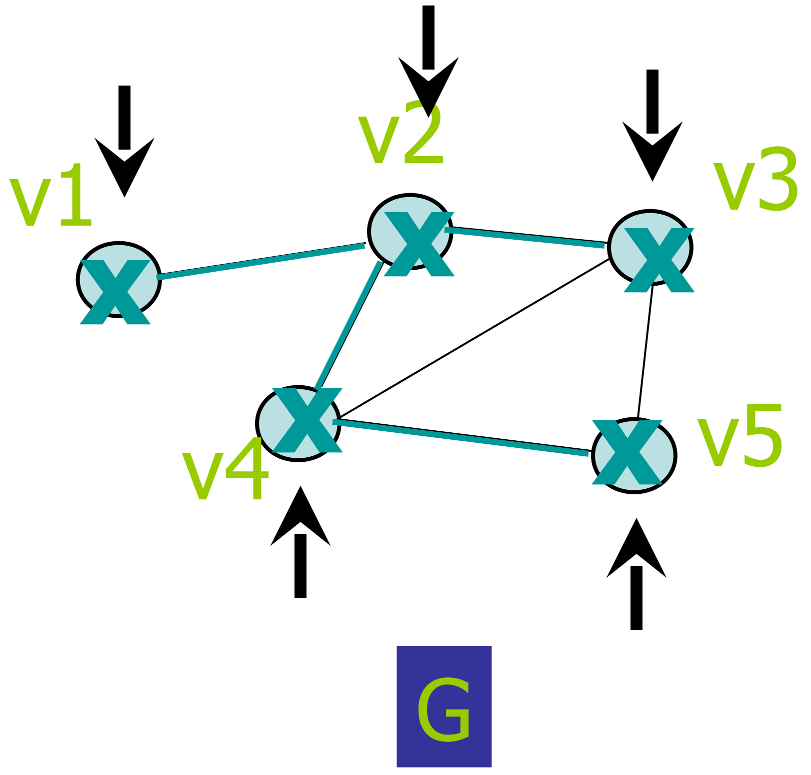
- Generate the spanning tree edge during the DFS traversal.

```
Algorithm dfsSpanningTree(v)
mark v as visited;
for (each unvisited node u adjacent to v) {
    mark the edge from u to v;
    dfsSpanningTree(u);
}
```

- Similar to DFS, the spanning tree edges can be generated based on BFS traversal.

Example of generating spanning tree based on DFS

	stack
v3	v3
v2	v3, v2
v1	v3, v2, v1
backtrack	v3, v2
v4	v3, v2, v4
v5	v3, v2, v4 , v5
backtrack	v3, v2, v4
backtrack	v3, v2
backtrack	v3
backtrack	empty

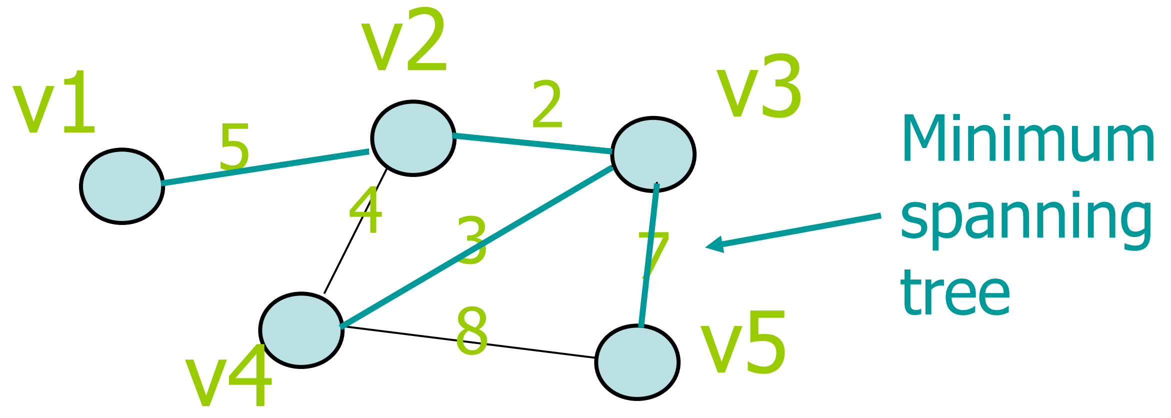


Minimum Spanning Tree

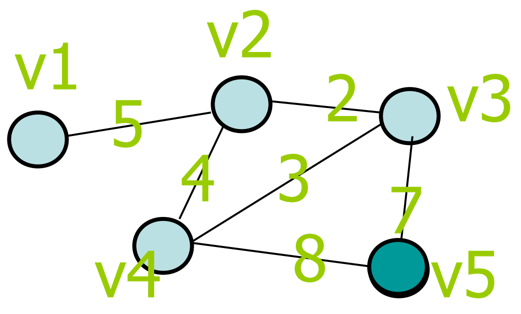
- Consider a connected undirected graph where
  - Each node x represents a country x
  - Each edge (x, y) has a number which measures the c ost of placing telephone line between country x and c ountry y
- Problem: connecting all countries while minimizi ng the total cost
- Solution: find a spanning tree with minimum total weight, that is, minimum spanning tree

Formal definition of minimum spanning tree

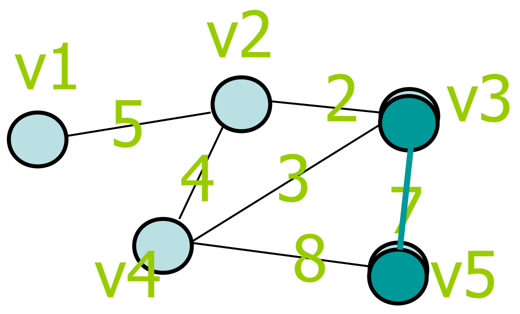
- Given a connected undirected graph G.
- Let T be a spanning tree of G.
- cost(T) = e Tweight(e)
- The minimum spanning tree is a spanning tree T which minimizes cost(T)



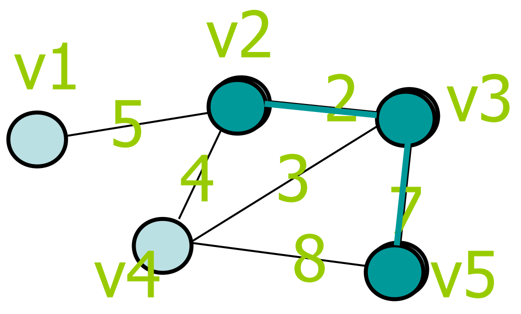
Prim’s algorithm (I)



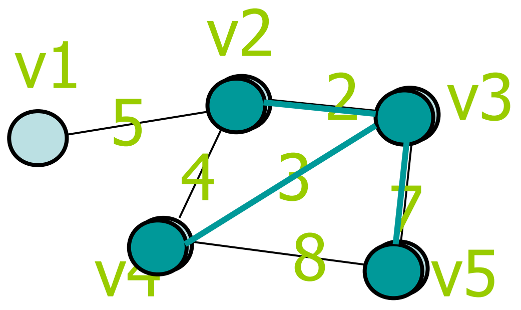
Start from v5, find the minimum edge attach to v5



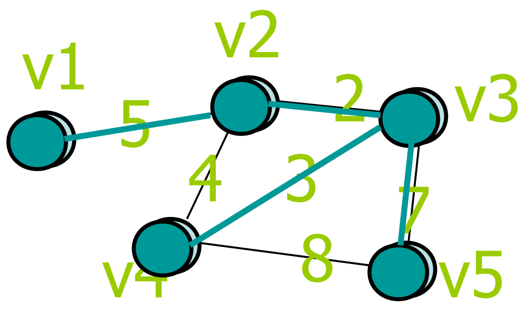
Find the minimum edge attach to v3 and v5



Find the minimum edge attach to v2, v3 and v5



Find the minimum edge attach to v2, v3 , v4 and v5



Prim's algorithm (II)

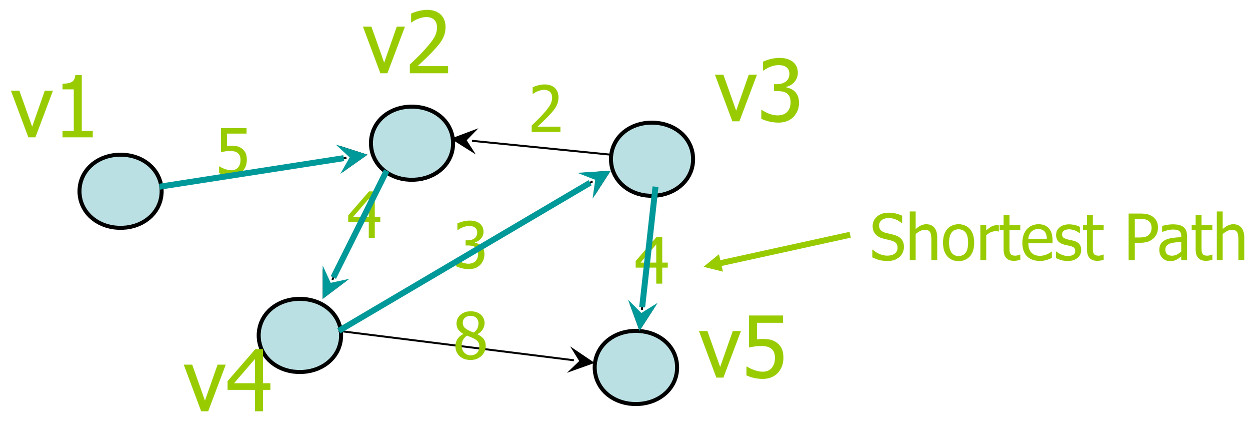
```
Algorithm PrimAlgorithm(v)
• Mark node v as visited and include it in the mini
  mum spanning tree;
• while (there are unvisited nodes) {
    – find the minimum edge (v, u) between a visited node v
      and an unvisited node u;
    – mark u as visited;
    – add both v and (v, u) to the minimum spanning tree;
  }
```

Shortest path

- Consider a weighted directed graph
  - Each node x represents a city x
  - Each edge (x, y) has a number which represent the c ost of traveling from city x to city y
- Problem: find the minimum cost to travel from cit y x to city y
- Solution: find the shortest path from x to y

Formal definition of shortest pat h

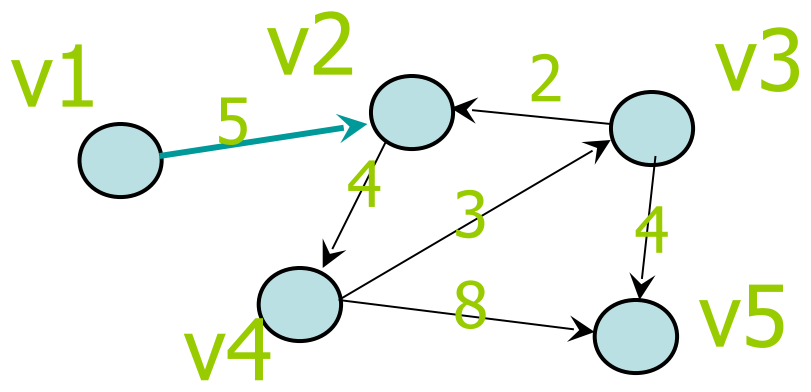
- Given a weighted directed graph G.
- Let P be a path of G from x to y.
- cost(P) = e Pweight(e)
- The shortest path is a path P which minimizes c ost(P)



Dijkstra's algorithm

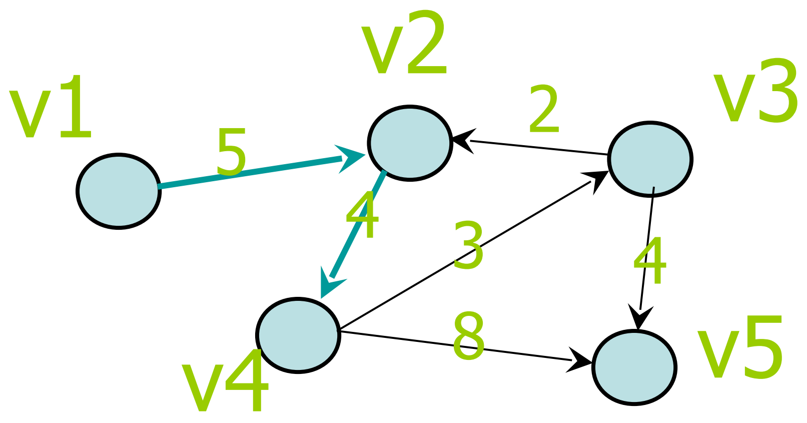
- Consider a graph G, each edge (u, v) has a weight w(u, v) > 0.
- Suppose we want to find the shortest path starting from v1 to any node vi
- Let VS be a subset of nodes in G
- Let cost[vi] be the weight of the shortest p ath from v1 to vi that passes through node s in VS only.

Example for Dijkstra’s algorithm



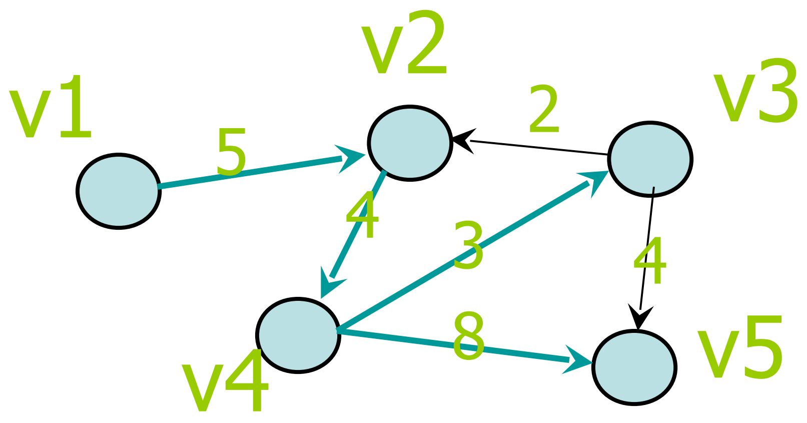
	v	VS	cost[v1]	cost[v2]	cost[v3]	cost[v4]	cost[v5]
1		[v1]	0	5	∞	∞	∞

Example for Dijkstra's algorithm



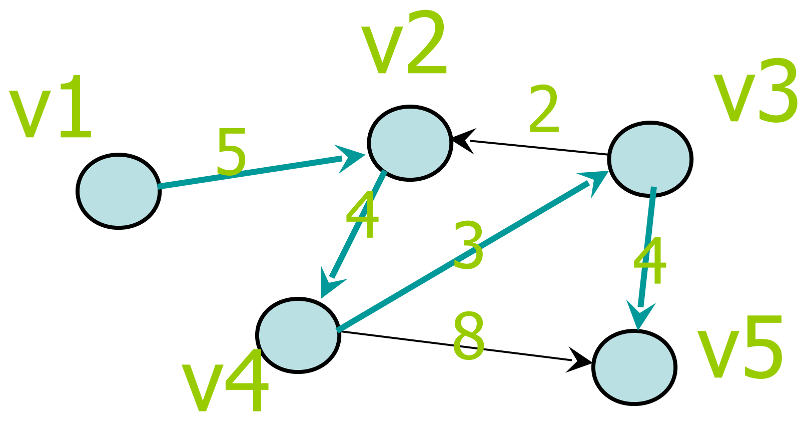
	v	VS	cost[v1]	cost[v2]	cost[v3]	cost[v4]	cost[v5]
1		[v1]	0	5	∞	∞	∞
2	v2	[v1, v2]	0	5	∞	9	∞

Example for Dijkstra’s algorithm



	v	VS	cost[v1]	cost[v2]	cost[v3]	cost[v4]	cost[v5]
1		[v1]	0	5	∞	∞	∞
2	v2	[v1, v2]	0	5	∞	9	∞
3	v4	[v1, v2, v4]	0	5	12	9	17

Example for Dijkstra’s algorithm



	v	VS	cost[v1]	cost[v2]	cost[v3]	cost[v4]	cost[v5]
1		[v1]	0	5	∞	∞	∞
2	v2	[v1, v2]	0	5	∞	9	∞
3	v4	[v1, v2, v4]	0	5	12	9	17
4	v3	[v1, v2, v4, v3]	0	5	12	9	16
5	v5	[v1, v2, v4, v3, v5]	0	5	12	9	16

Dijkstra's algorithm

```
Algorithm shortestPath()
n = number of nodes in the graph;
for i = 1 to n
    cost[vi] = w(v1, vi);
VS = { v1 };
for step = 2 to n {
    find the smallest cost[vi] s.t. vi is not in VS;
    include vi to VS;
    for (all nodes vj not in VS) {
        if (cost[vj] > cost[vi] + w(vi, vj))
            cost[vj] = cost[vi] + w(vi, vj);
    }
}
```

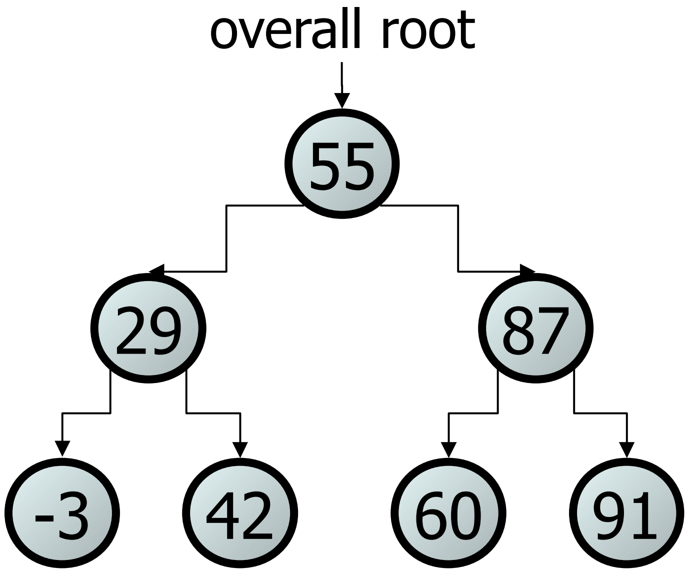
Summary

- Graphs can be used to represent many real-life problems.
- There are numerous important graph algorithms.
- We have studied some basic concepts and algor ithms.
  - Graph Traversal
  - Topological Sort
  - Spanning Tree
  - Minimum Spanning Tree
  - Shortest Path

Binary search trees

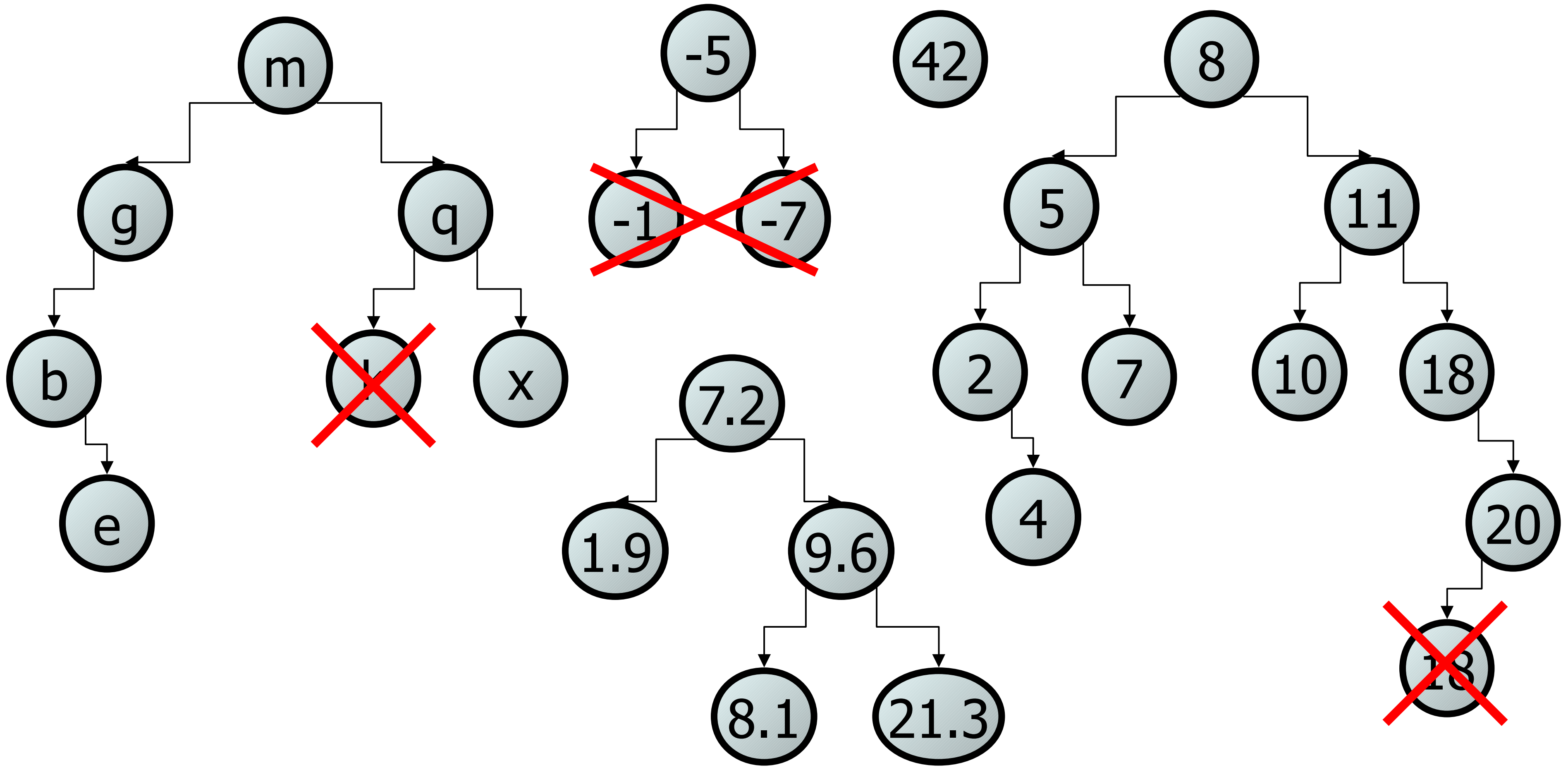
- **binary search tree** ("BST"): a binary tree where each non-empty node R has the following properties:
  - every element of R's left subtree contains data "less than" R's data,
  - every element of R's right subtree contains data "greater than" R's,
  - R's left and right subtrees are also binary search trees.

- BSTs store their elements in sorted order, which is helpful for searching/sorting tasks.



BST examples

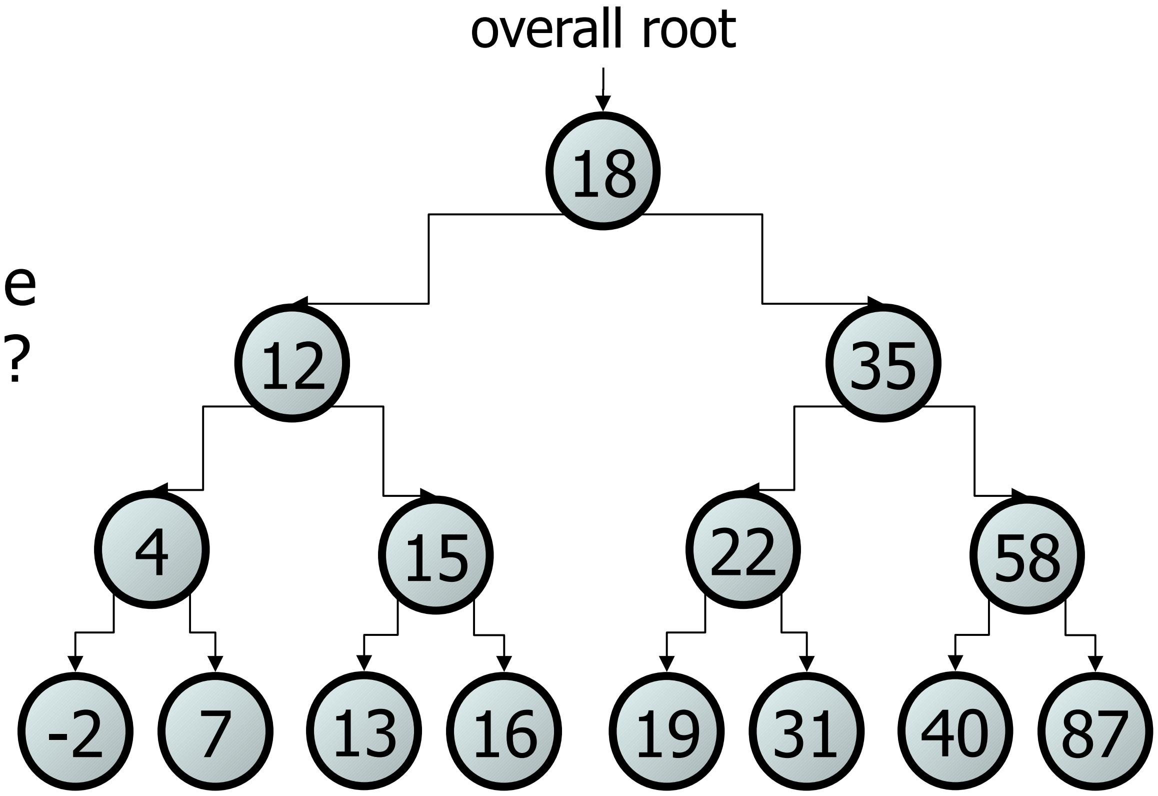
- Which of the trees shown are legal binary search trees?



Searching a BST

- Describe an algorithm for searching a binary search tree.
  - Try searching for the value 31, then 6.

- What is the maximum number of nodes you would need to examine to perform any search?

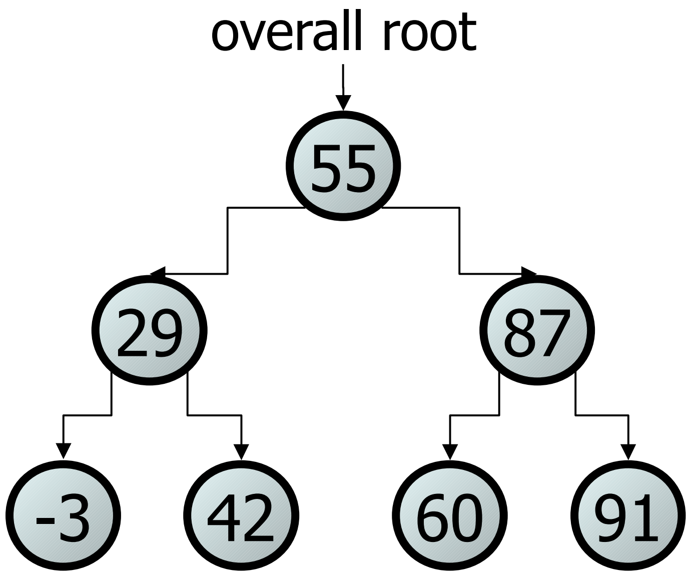


Exercise

- Convert the `IntTree` class into a `SearchTree` class.
  - The elements of the tree will constitute a legal binary search tree.

- Modify `contains` to take advantage of the BST structure.

`tree.contains(29)`	`true`
`tree.contains(55)`	`true`
`tree.contains(63)`	`false`
`tree.contains(35)`	`false`

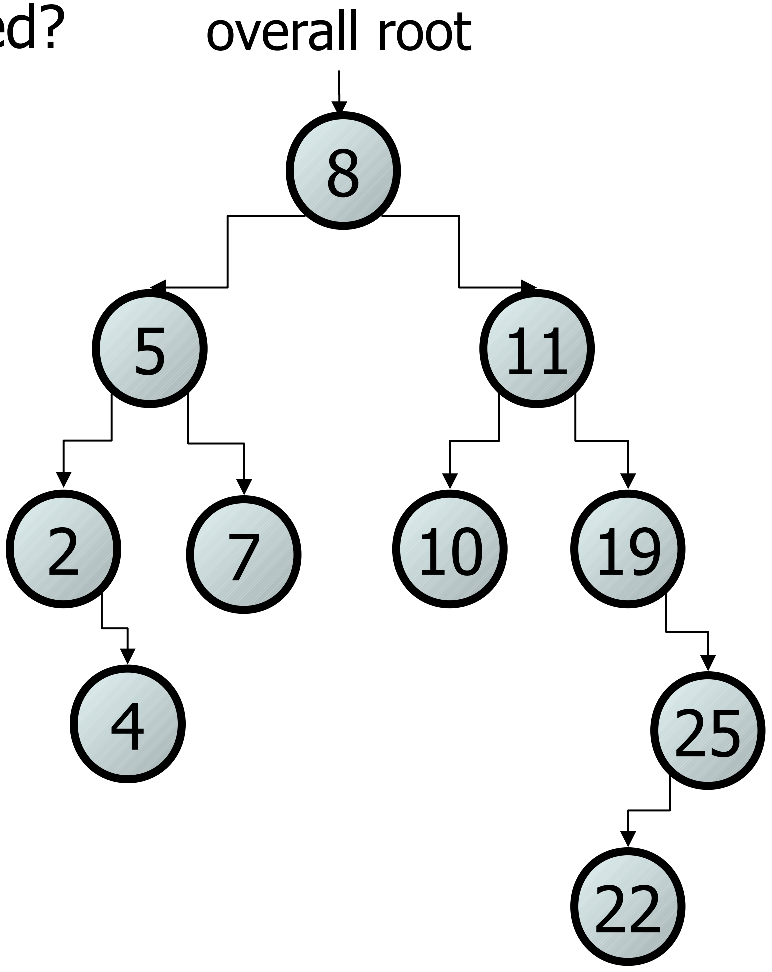


Exercise solution

```
// Returns whether this BST contains the given integer.
public boolean contains(int value) {
    return contains(overallRoot, value);
}

private boolean contains(IntTreeNode node, int value) {
    if (node == null) {
        return false;    // base case: not found here
    } else if (node.data == value) {
        return true;     // base case: found here
    } else if (node.data > value) {
        return contains(node.left, value);
    } else {    // root.data < value
        return contains(node.right, value);
    }
}
```

Adding to a BST

- Suppose we want to add new values to the BST below.
  - Where should the value 14 be added?
  - Where should 3 be added? 7?
  - If the tree is empty, where should a new value be added?
- What is the general algorithm?



Adding exercise

- Draw what a binary search tree would look like if the following values were added to an initially empty tree in this order:

50

20

75

98

80

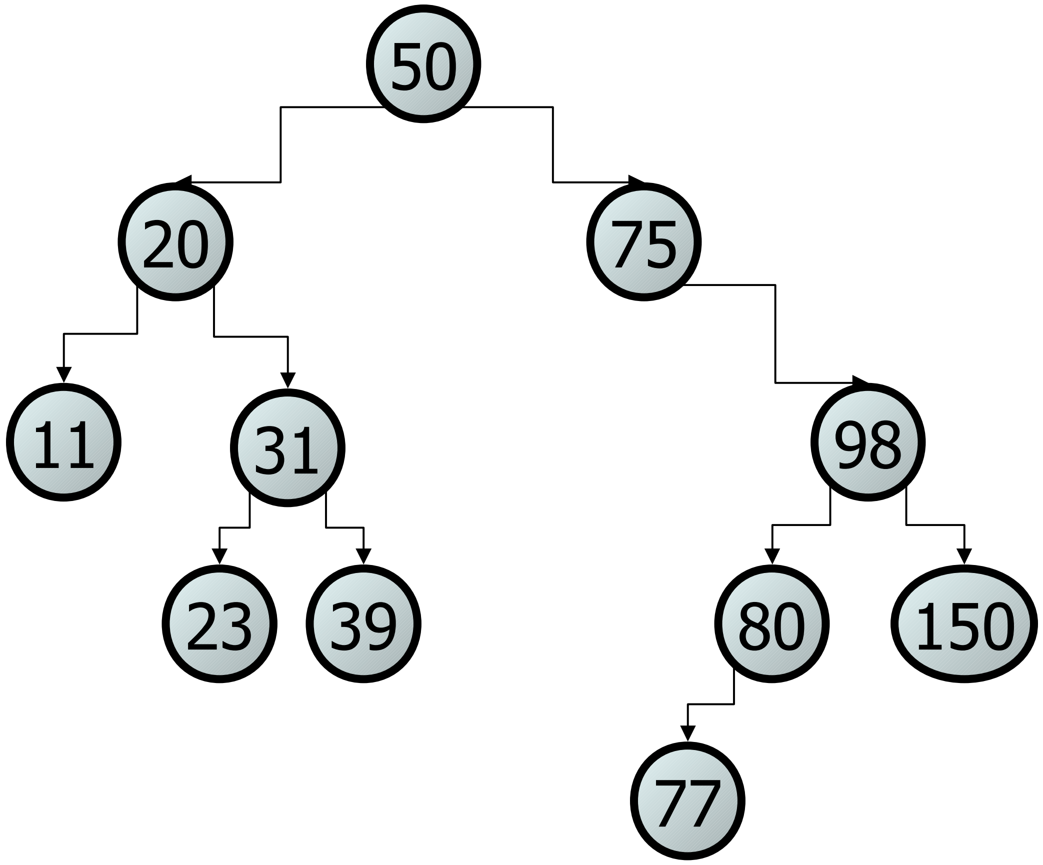
31

150

39

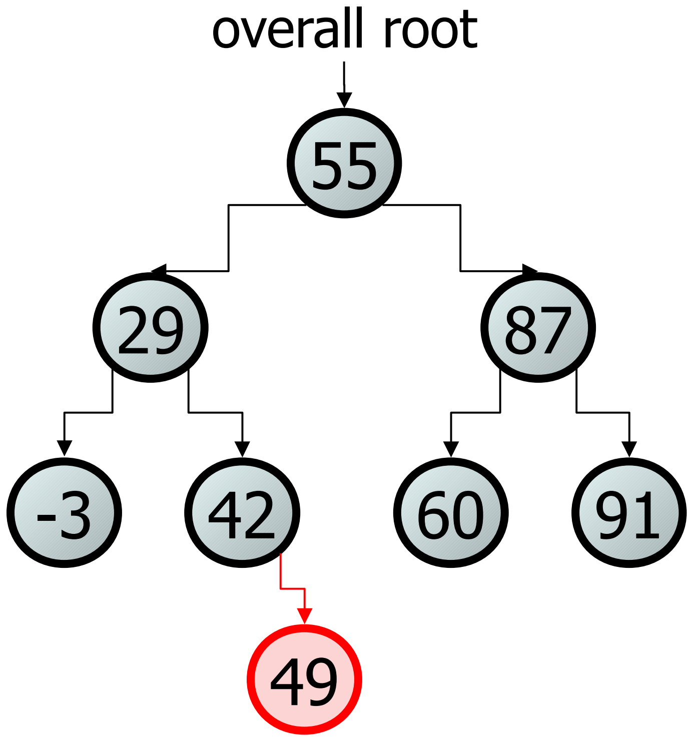
23

11



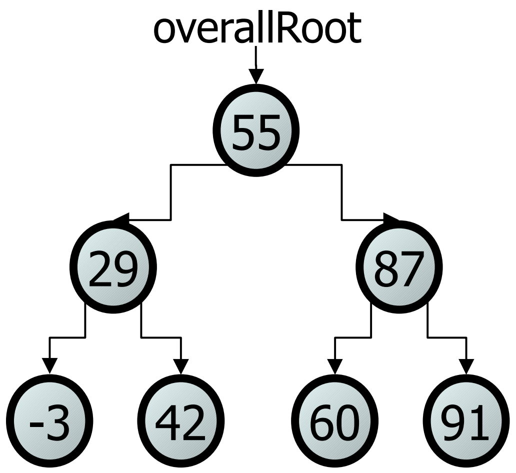
Exercise

- Add a method `add` to the `SearchTree` class that adds a given integer value to the BST.
  - Add the new value in the proper place to maintain BST ordering.
    - `tree.add(49);`



An incorrect solution

```
// Adds the given value to this BST in sorted order.
public void add(int value) {
    add(overallRoot, value);
}

private void add(IntTreeNode node, int value) {
    if (node == null) {
        node = new IntTreeNode(value);
    } else if (node.data > value) {
        add(node.left, value);
    } else if (node.data < value) {
        add(node.right, value);
    }
    // else node.data == value, so
    // it's a duplicate (don't add)
}
```

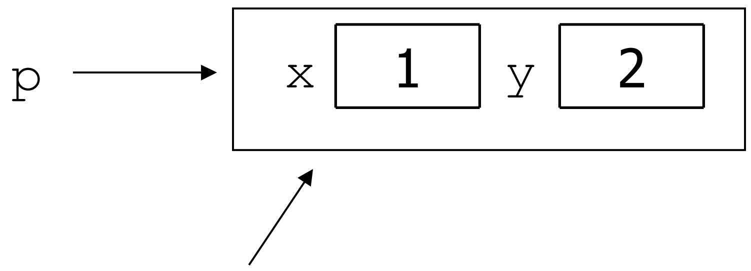


- Why doesn't this solution work?

A tangent: Change a point

- What is the state of the object referred to by `p` after this code?

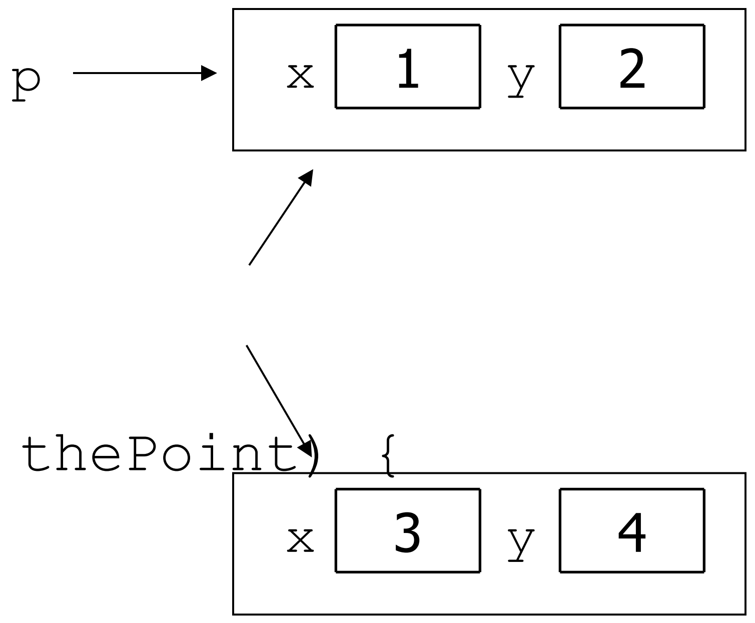
```
public static void main(String[] args) {
    Point p = new Point(1, 2);
    change(p);
    System.out.println(p);
}

public static void change(Point thePoint) {
    thePoint.x = 3;
    thePoint.y = 4;
}

// answer: (3, 4)
```



Change point, version 2

- What is the state of the object referred to by `p` after this code?

```
public static void main(String[] args) {
    Point p = new Point(1, 2);
    change(p);
    System.out.println(p);
}

public static void change(Point thePoint) {
    thePoint = new Point(3, 4);
}
```



```
// answer: (1, 2)
```

Changing references

- If a method *dereferences a variable* (with `.`) and modifies the object it refers to, that change will be seen by the caller.

```
public static void change(Point thePoint) {
    thePoint.x = 3;                    // affects p
    thePoint.setY(4);                  // affects p
```

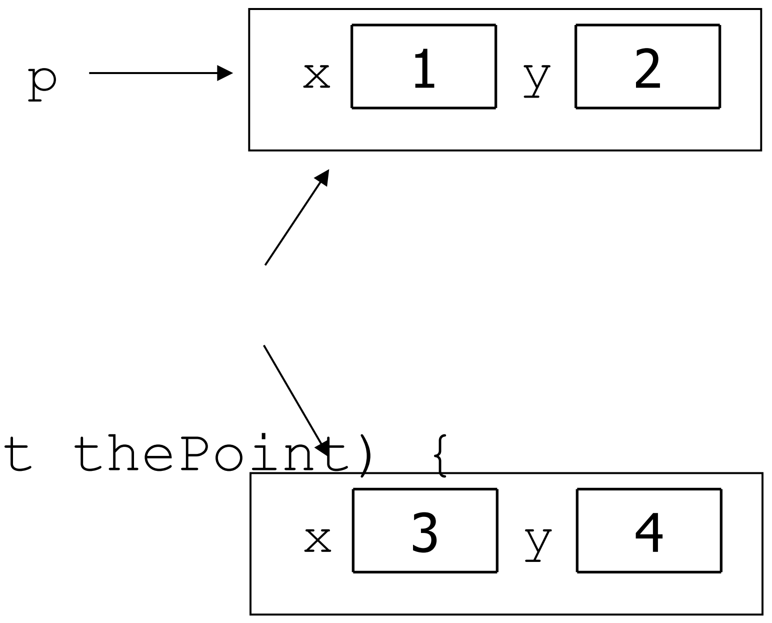
- If a method *reassigns a variable to refer to a new object,* that change will *not* affect the variable passed in by the caller.

```
public static void change(Point thePoint) {
    thePoint = new Point(3, 4);        // p unchanged
    thePoint = null;                   // p unchanged
```

Change point, version 3

- What is the state of the object referred to by `p` after this code?

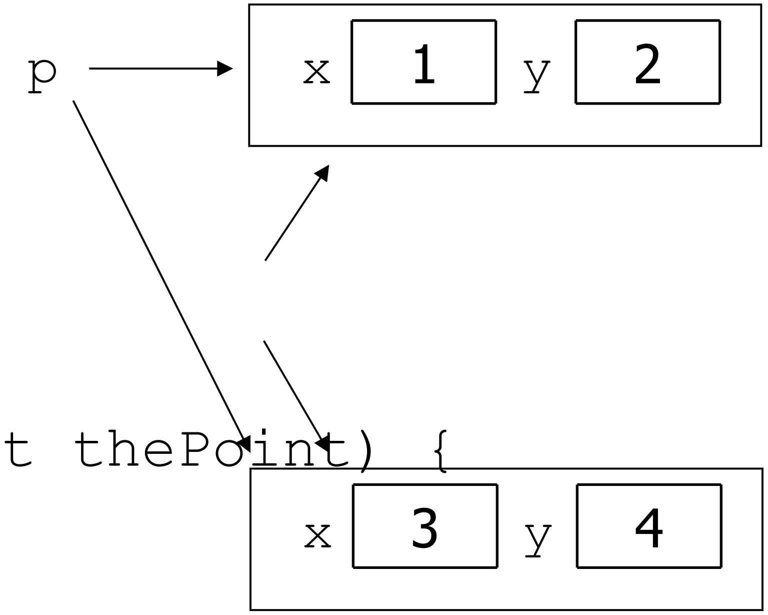
```
public static void main(String[] args) {
    Point p = new Point(1, 2);
    change(p);
    System.out.println(p);
}

public static Point change(Point thePoint) {
    thePoint = new Point(3, 4);
    return thePoint;
}

// answer: (1, 2)
```



Change point, version 4

- What is the state of the object referred to by `p` after this code?

```
public static void main(String[] args) {
    Point p = new Point(1, 2);
    p = change(p);
    System.out.println(p);
}

public static Point change(Point thePoint) {
    thePoint = new Point(3, 4);
    return thePoint;
}
```



// answer: (3, 4)

x = change(x);

- If you want to write a method that can change the object that a variable refers to, you must do three things:

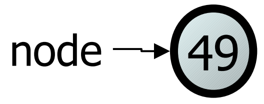
1. **pass** in the original state of the object to the method
2. **return** the new (possibly changed) object from the method
3. **re-assign** the caller's variable to store the returned result

```
    p = change(p);      // in main

public static Point change(Point thePoint) {
    thePoint = new Point(99, -1);
    return thePoint;
```

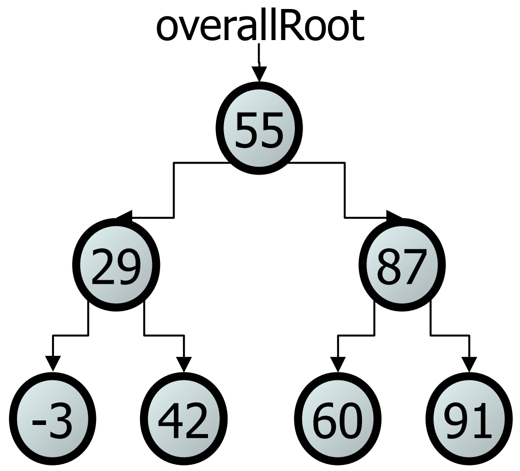
- We call this general algorithmic pattern **x = change(x);**

The problem

- Much like with linked lists, if we just modify what a local variable refers to, it won't change the collection.



```
private void add(IntTreeNode node, int value) {
    if (node == null) {
        node = new IntTreeNode(value);
    }
```



  - In the linked list case, how did we actually modify the list?
    - by changing the `front`
    - by changing a node's `next` field

Applying x = change(x)

- Methods that modify a tree should have the following pattern:
  - input (parameter): old state of the node
  - output (return): new state of the node

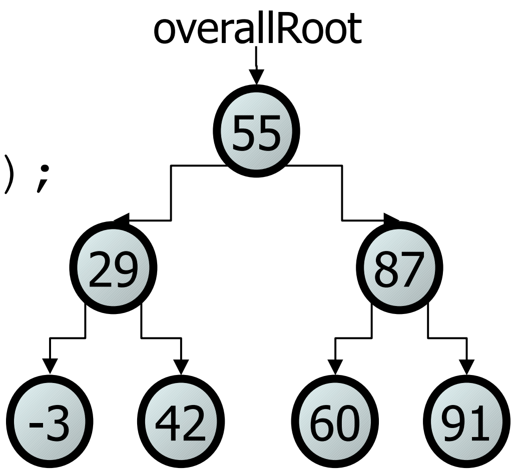
$$\begin{tikzcd}\text{node before} \arrow[r, "\textit{parameter}"] & \boxed{\text{your method}} \arrow[r, "\textit{return}"] & \text{node after}\end{tikzcd}$$

- In order to actually change the tree, you must reassign:

```
            node       = change(node,
parameters);
            node.left  = change(node.left,
parameters);
            node.right = change(node.right,
parameters);
```

A correct solution

```
// Adds the given value to this BST in sorted order.
public void add(int value) {
    overallRoot = add(overallRoot, value);
}

private IntTreeNode add(IntTreeNode node, int value) {
    if (node == null) {
        node = new IntTreeNode(value);
    } else if (node.data > value) {
        node.left = add(node.left, value);
    } else if (node.data < value) {
        node.right = add(node.right, value);
    } // else a duplicate; do nothing

    return node;
}
```

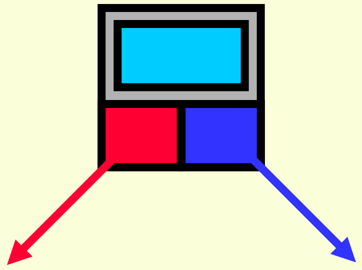


Traversing a Binary Tree
Binary Search Tree Insertion
Deleting from a Binary Search Tree

Traversing a Binary Tree

Inorder Traversal

The Scenario

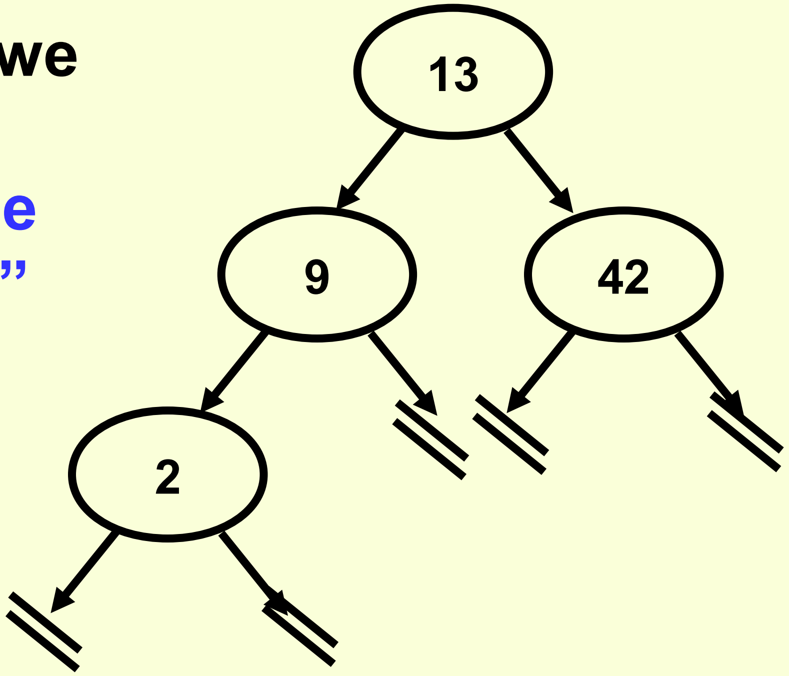


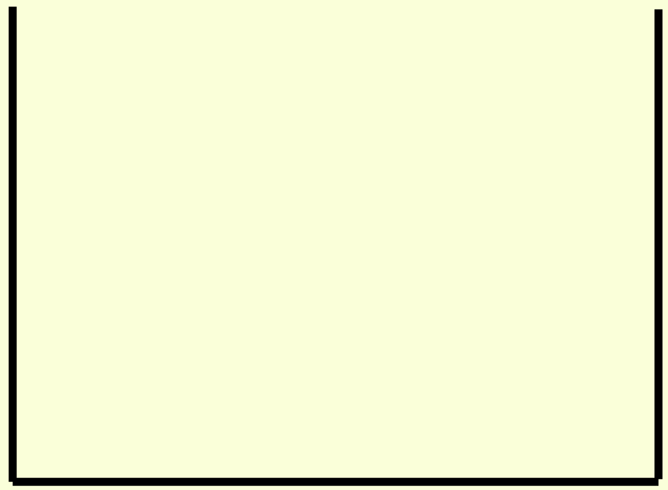
- Imagine we have a binary tree
- We want to traverse the tree
  - It’s not linear
  - We need a way to visit all nodes

- Three things must happen:
  - Deal with the entire left sub-tree
  - Deal with the current node
  - Deal with the entire right sub-tree

Use the Activation Stack

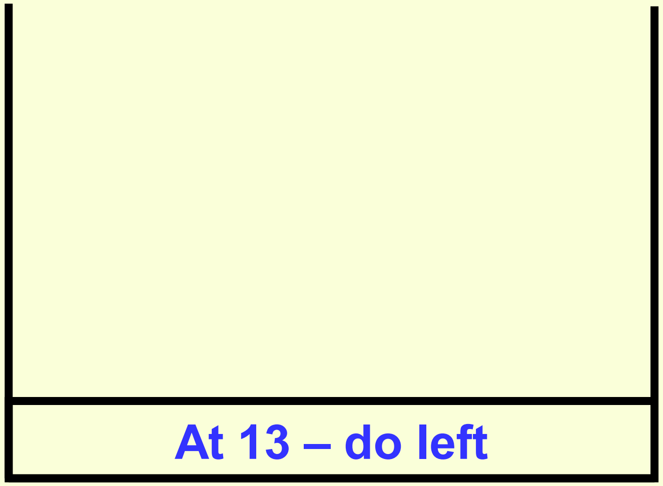
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

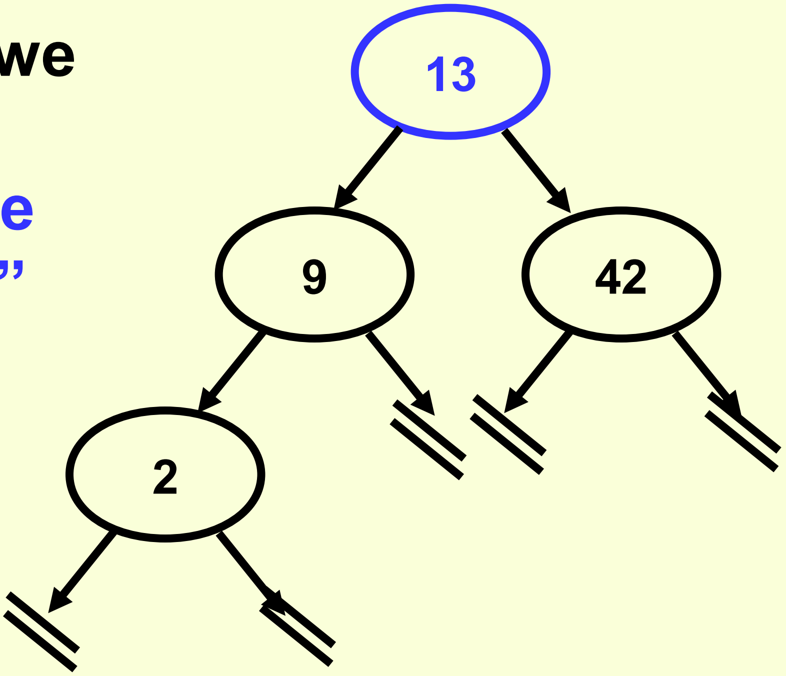




Use the Activation Stack

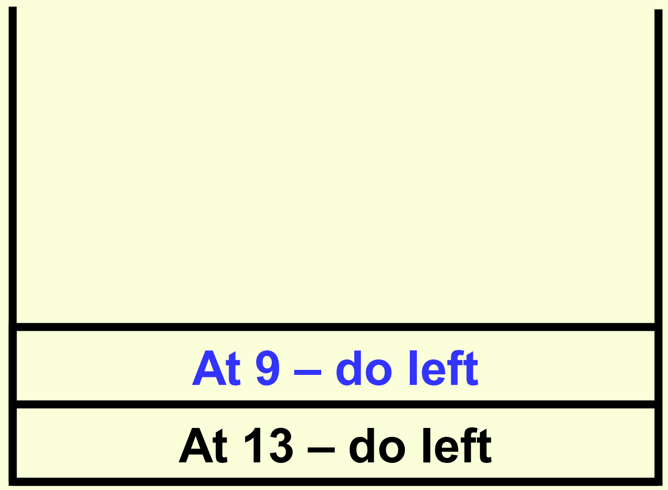
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

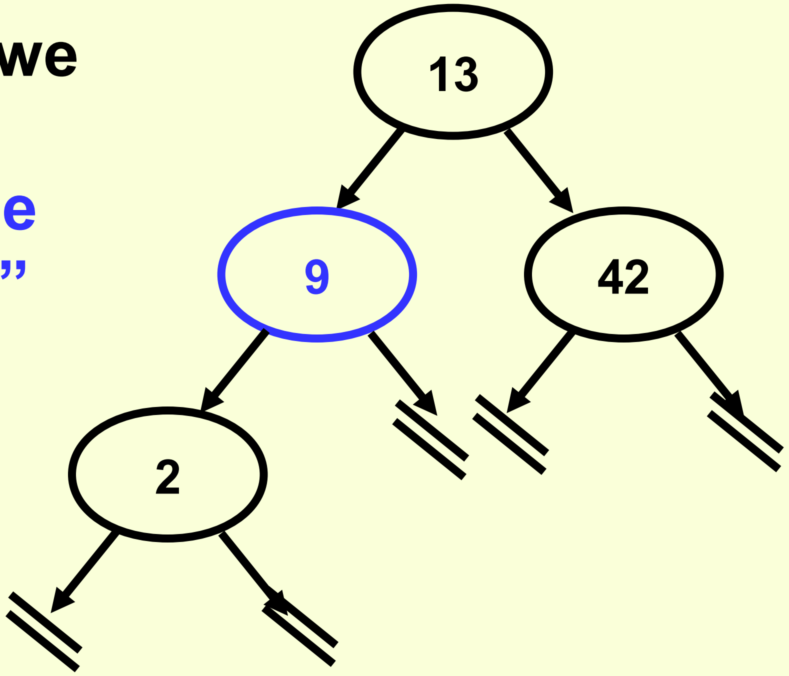




Use the Activation Stack

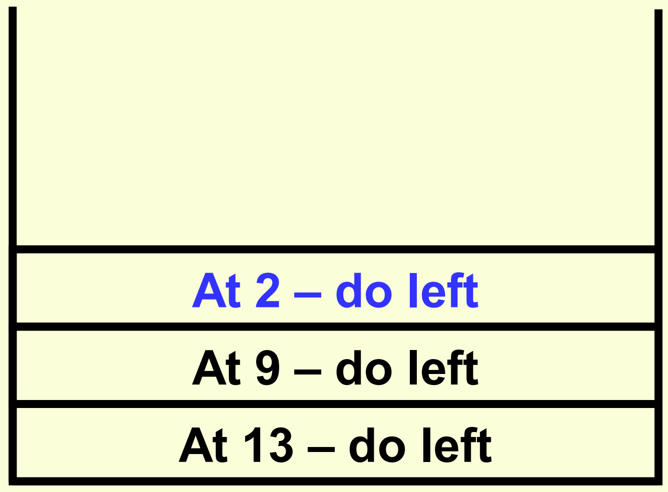
- With a recursive module, we can make use of the activation stack to visit the sub-trees and “remember” where we left off.

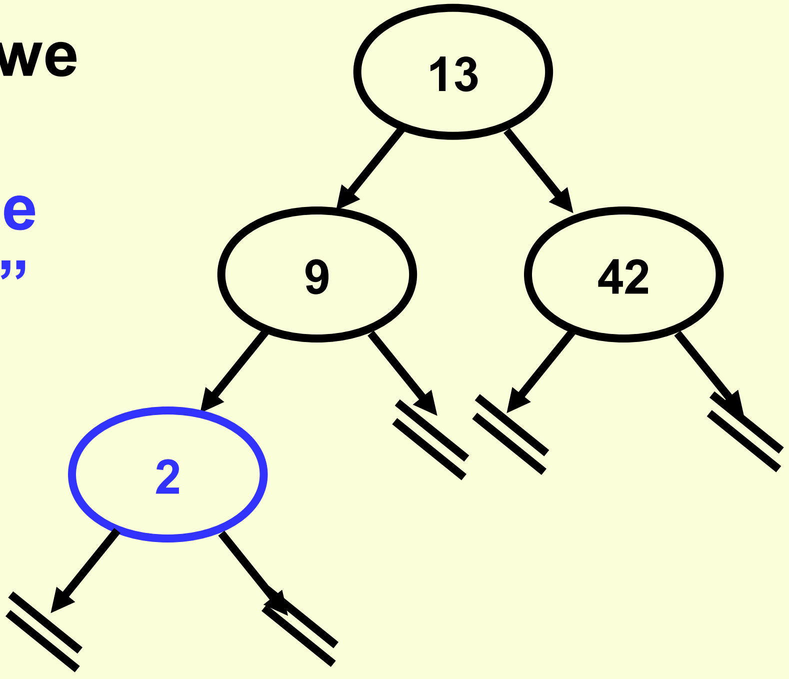




Use the Activation Stack

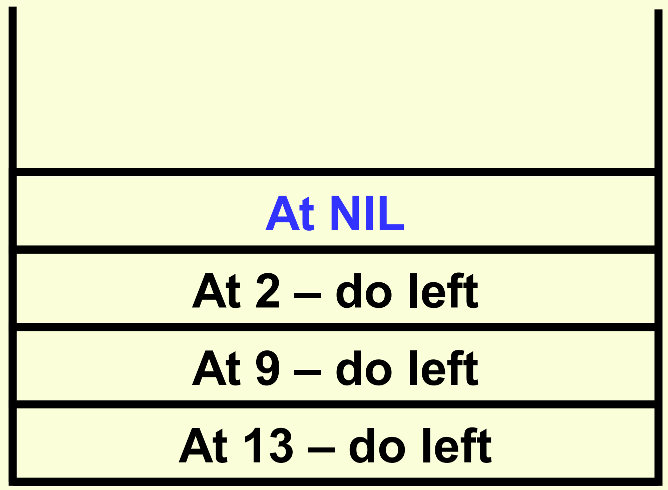
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

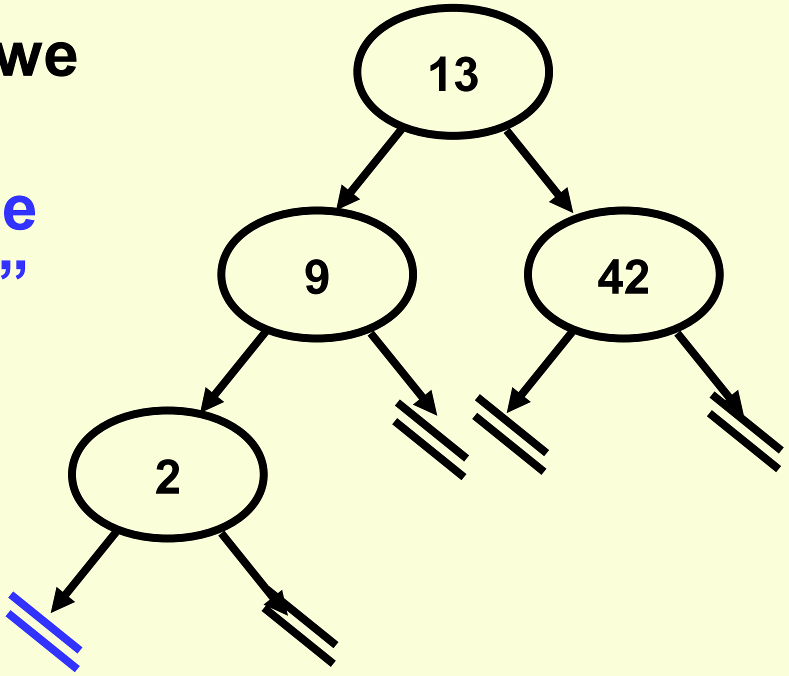




Use the Activation Stack

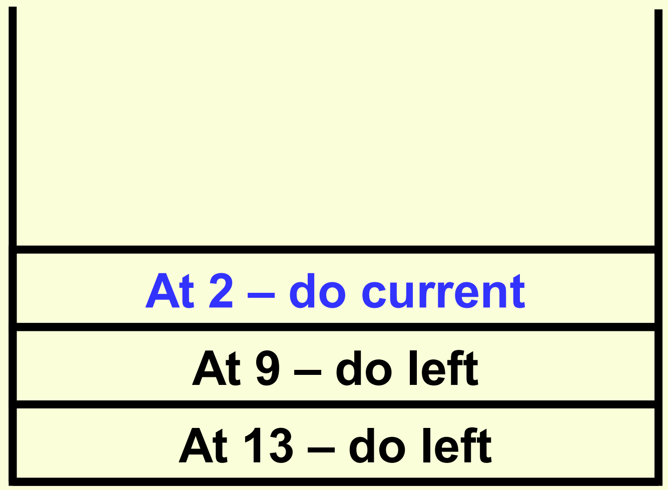
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

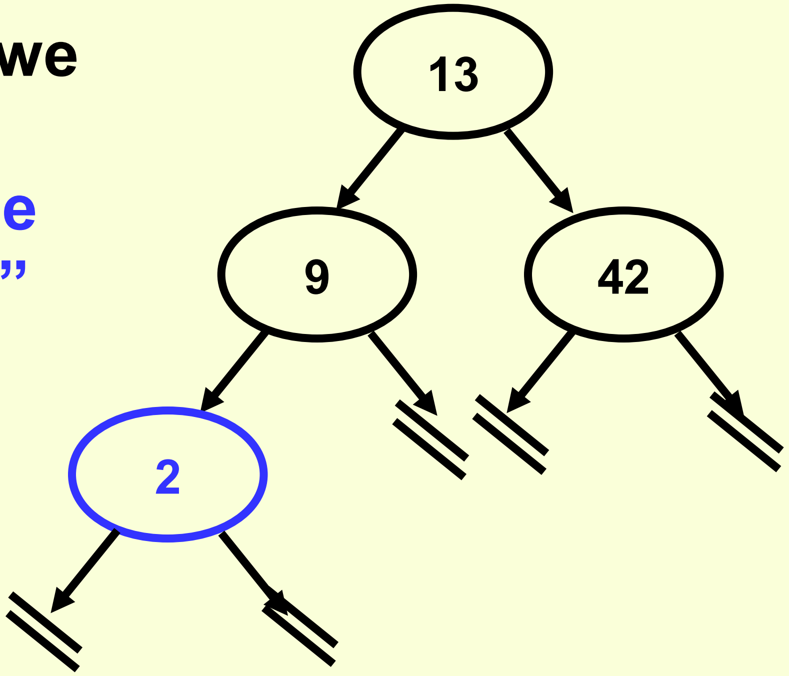




Use the Activation Stack

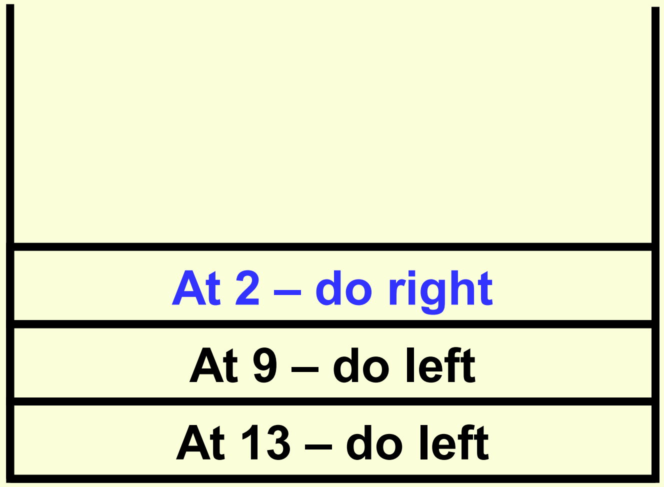
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

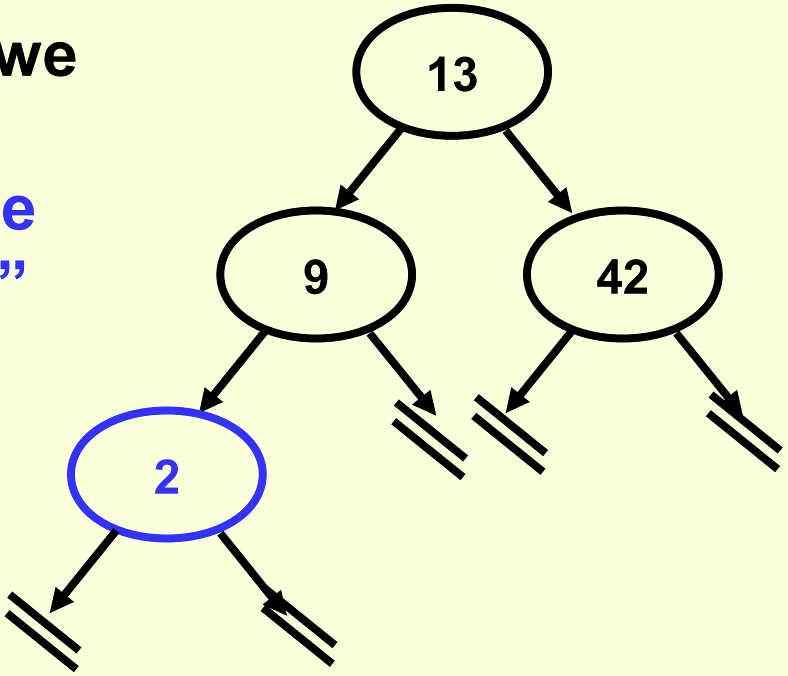




Use the Activation Stack

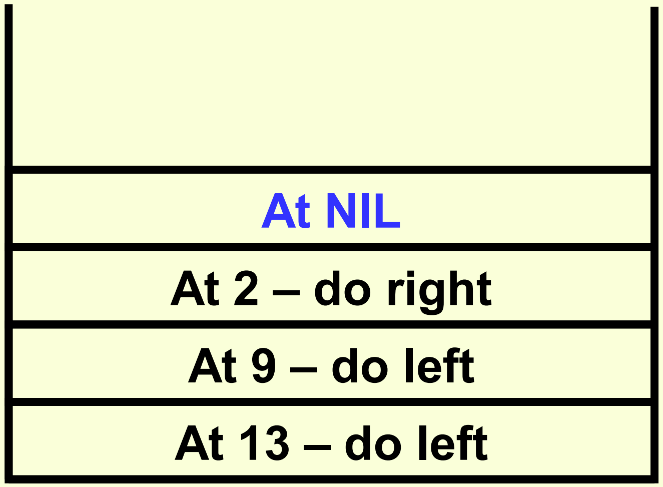
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

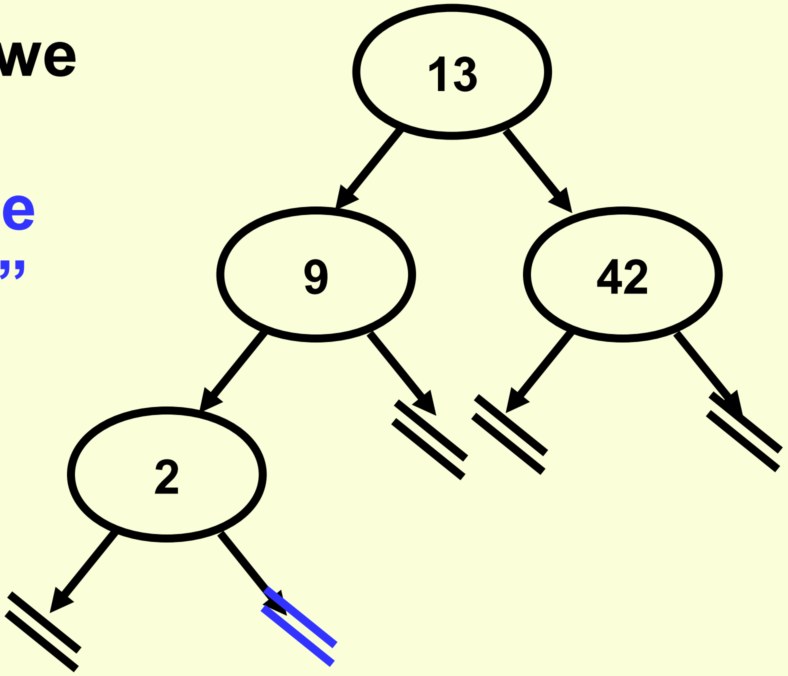




Use the Activation Stack

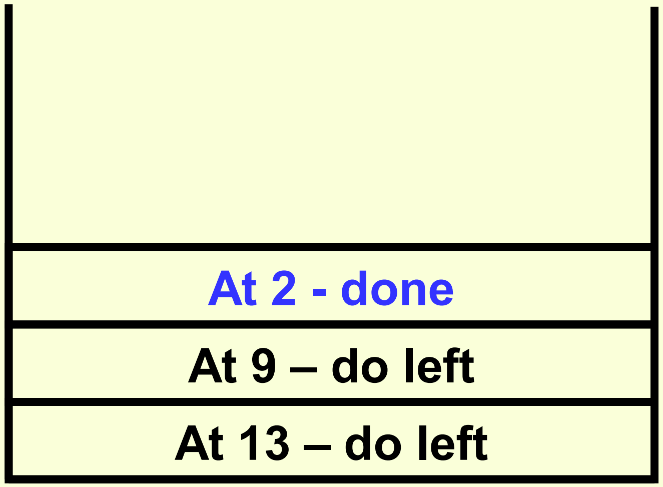
- With a recursive module, we can make use of the activation stack to visit the sub-trees and “remember” where we left off.

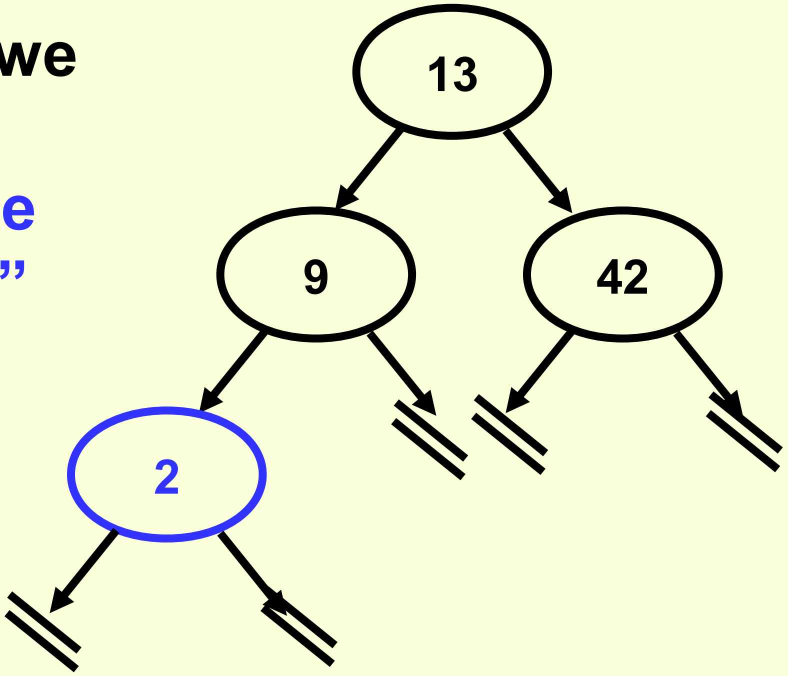




Use the Activation Stack

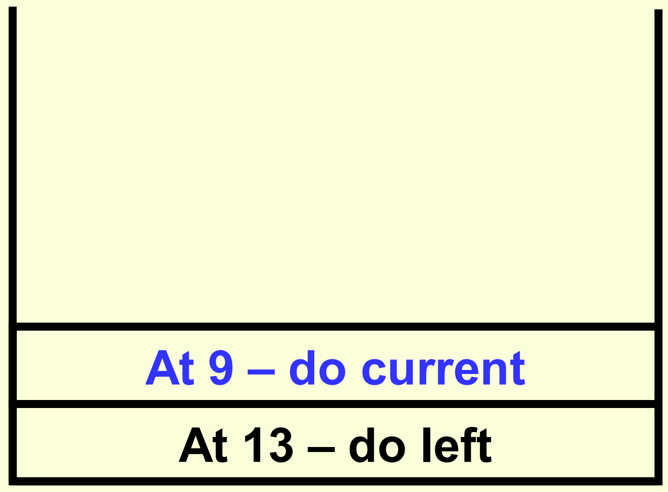
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

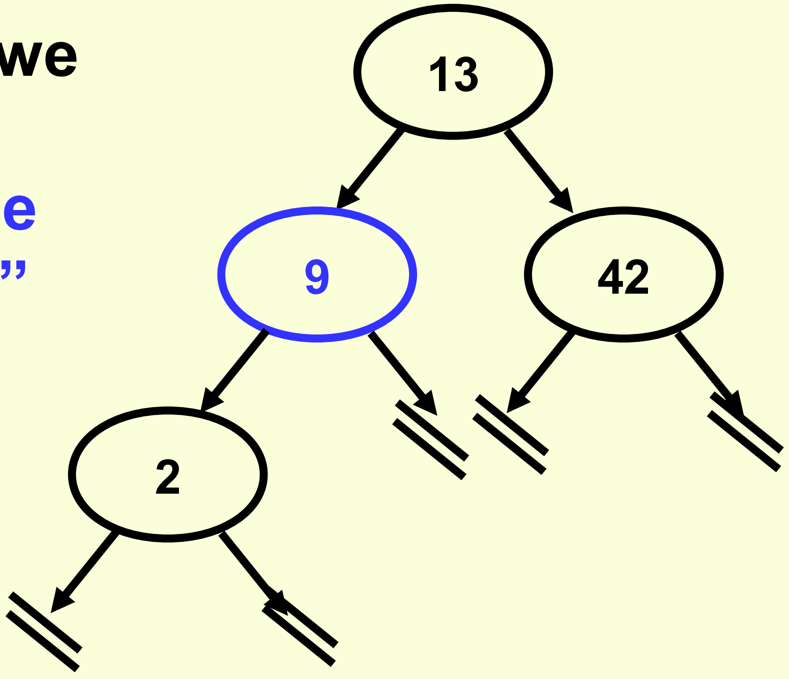




Use the Activation Stack

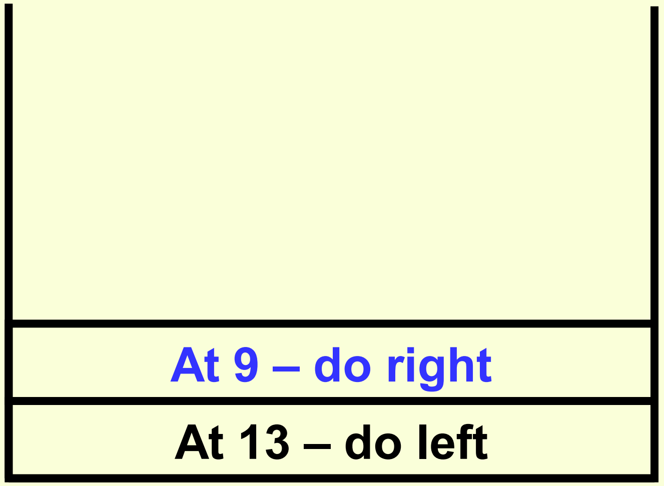
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

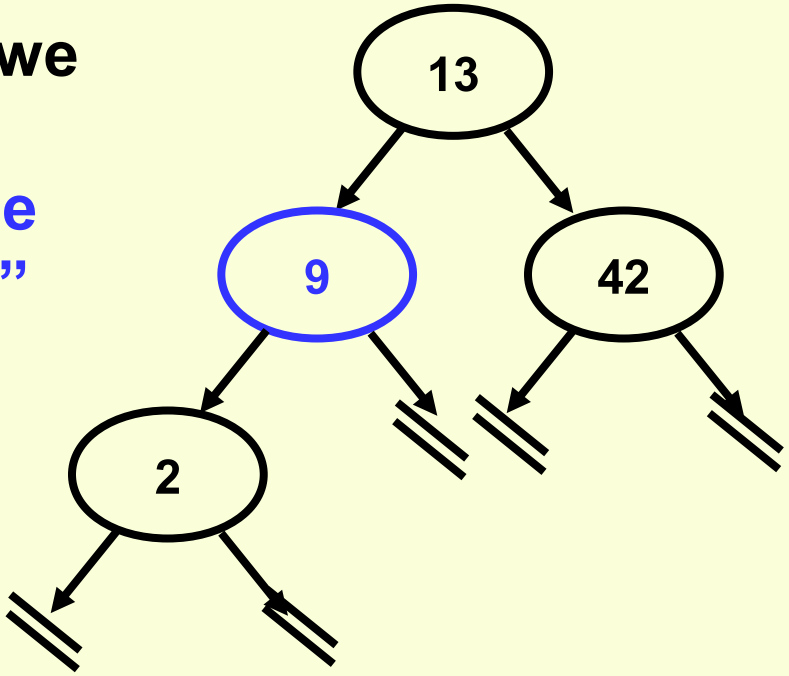




Use the Activation Stack

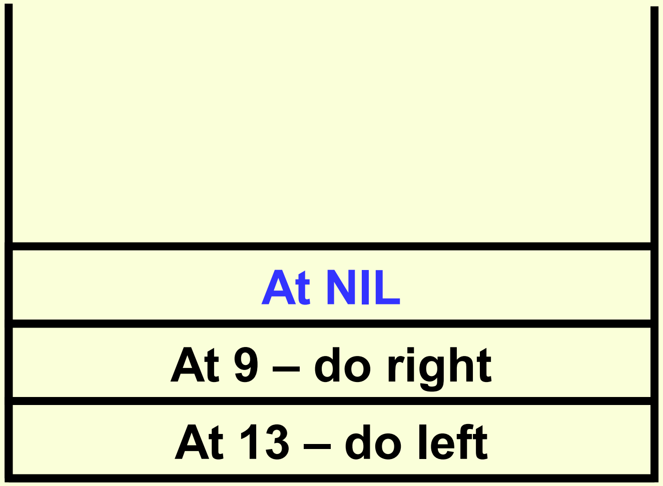
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

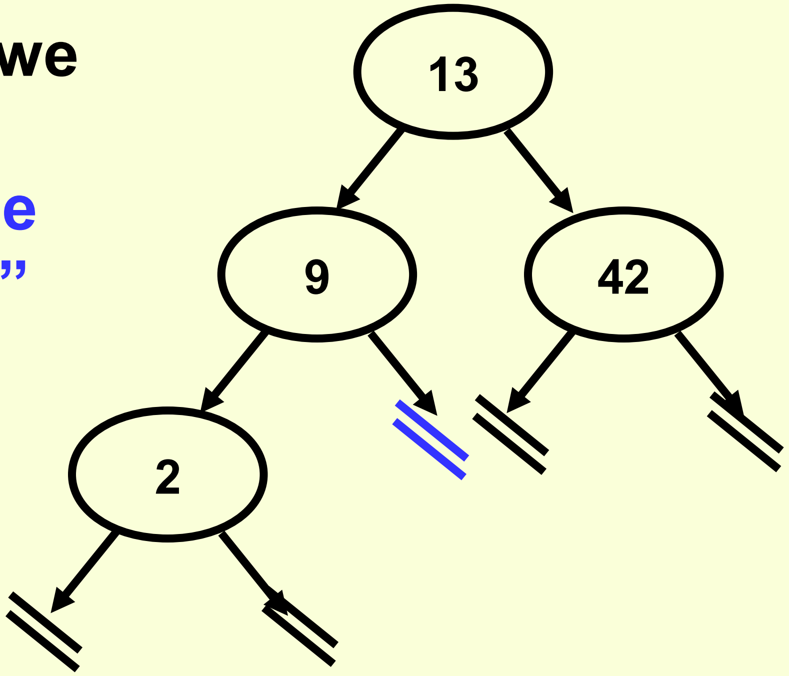




Use the Activation Stack

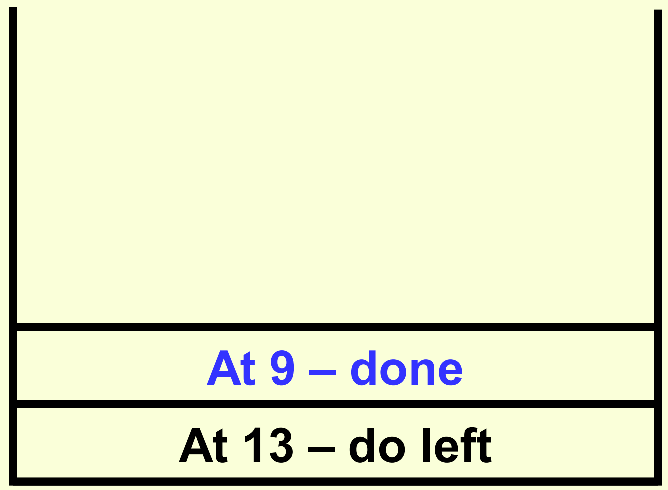
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

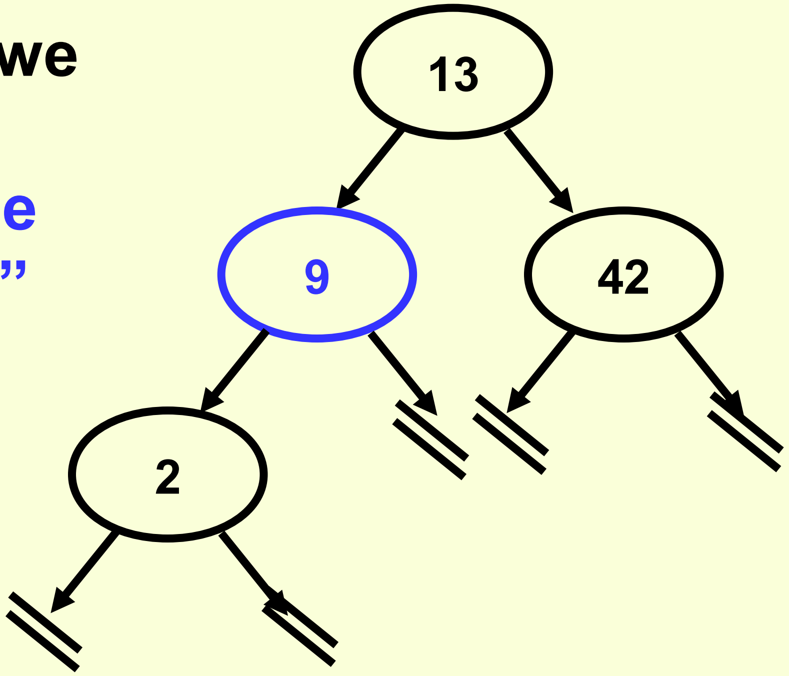




Use the Activation Stack

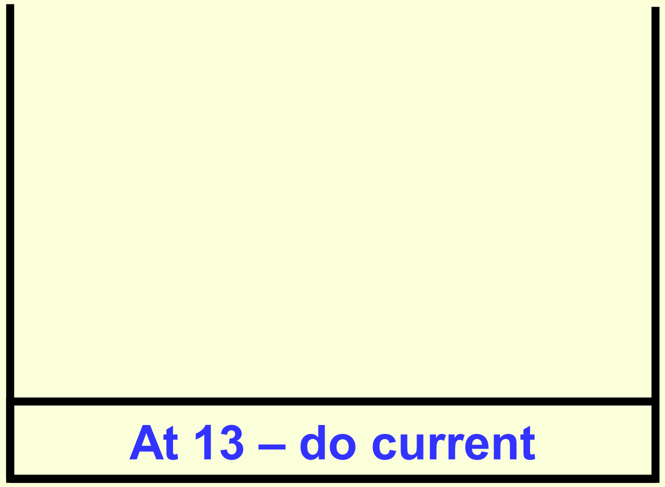
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

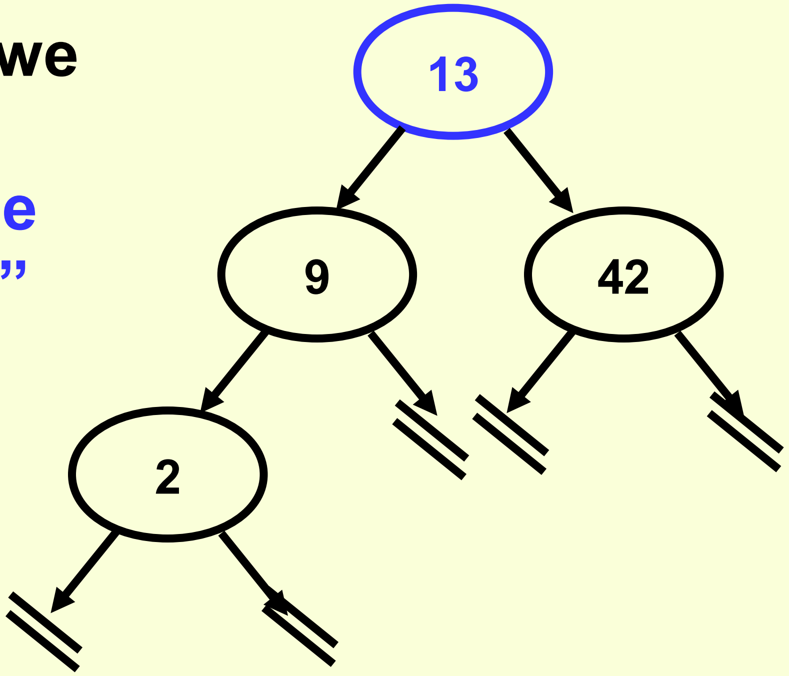




Use the Activation Stack

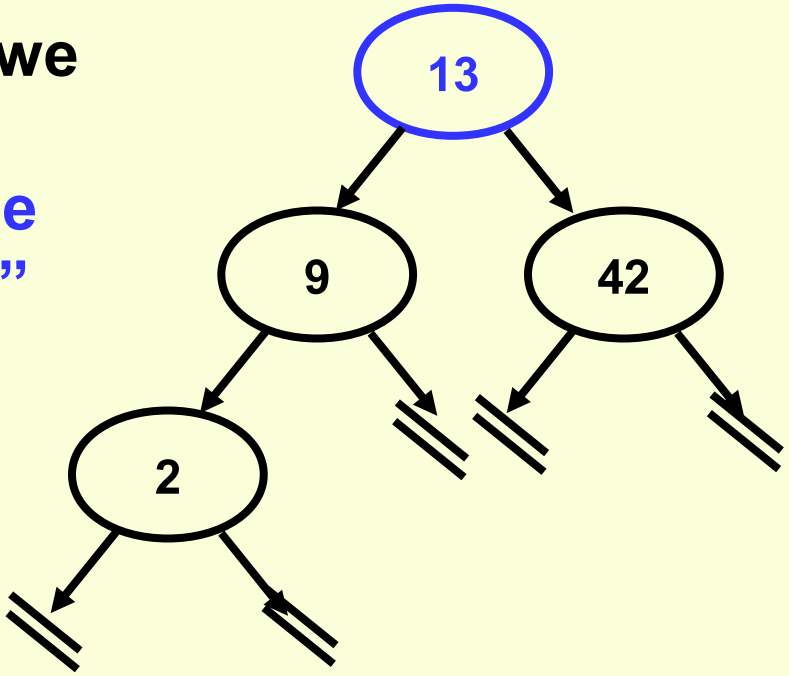
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

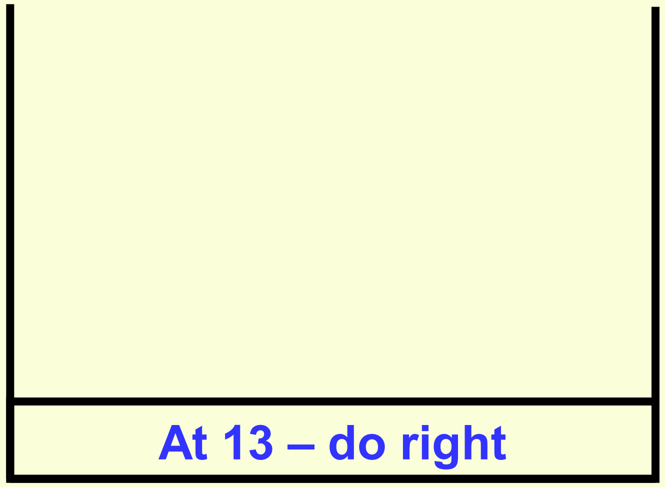




Use the Activation Stack

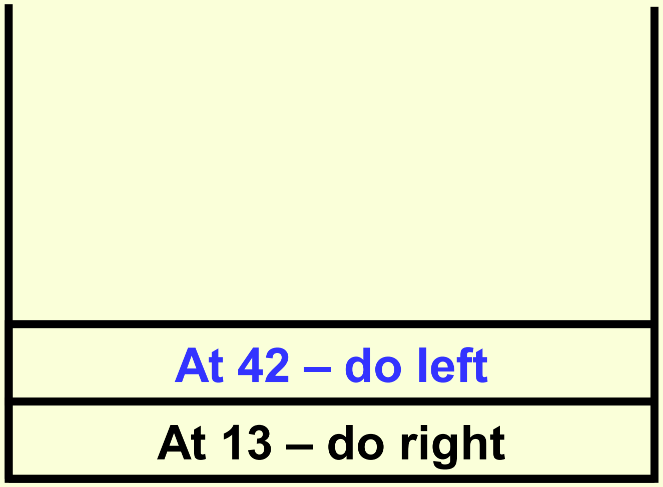
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

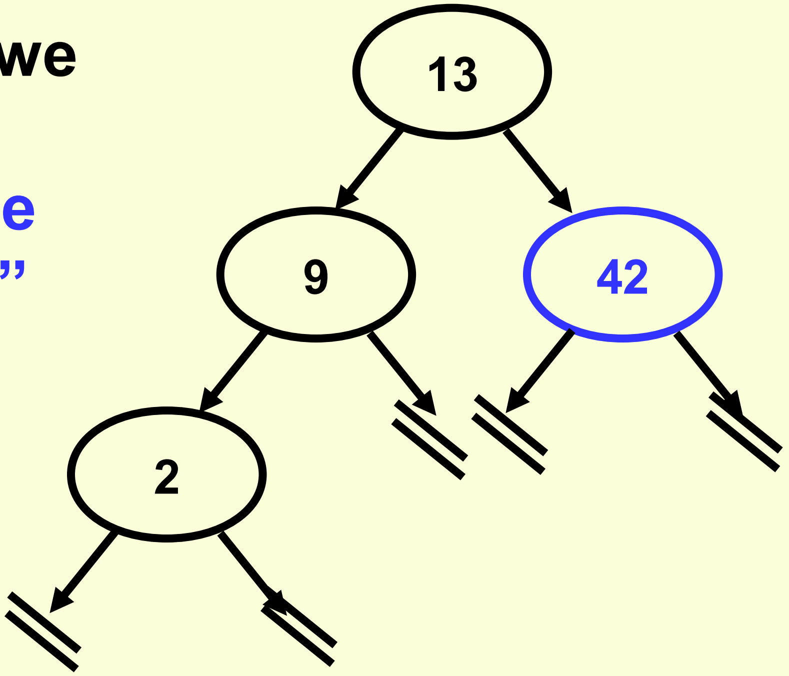




Use the Activation Stack

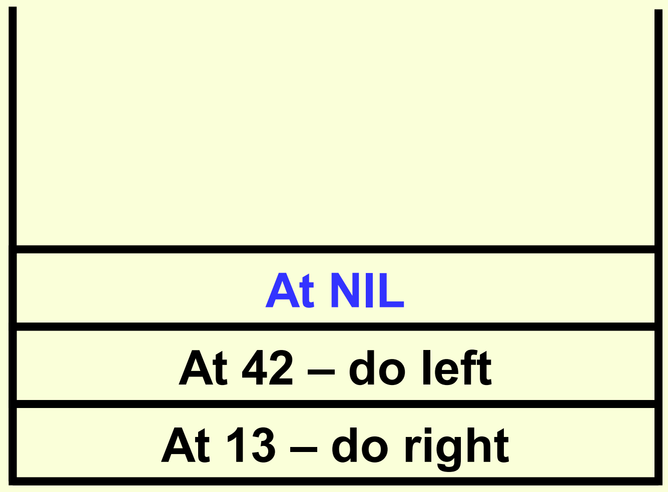
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

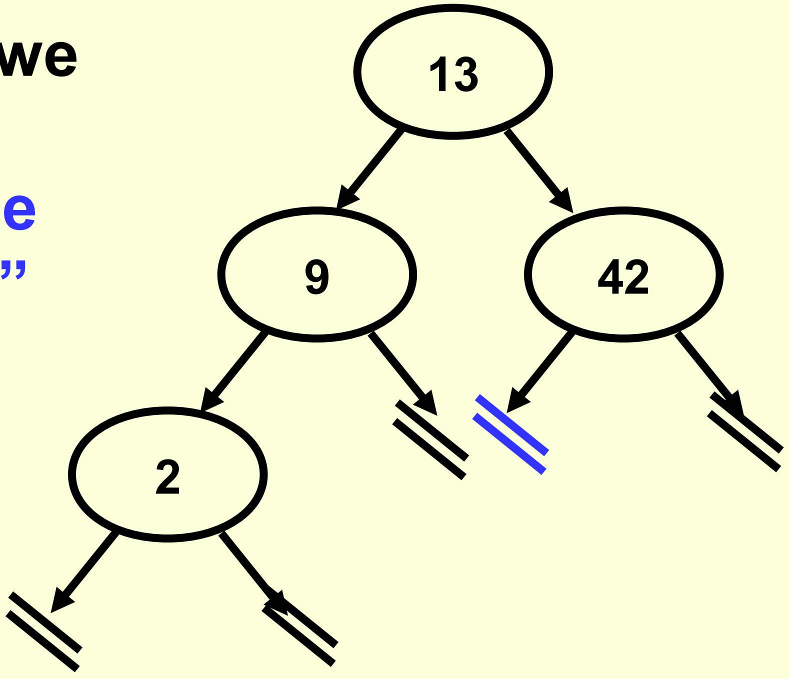




Use the Activation Stack

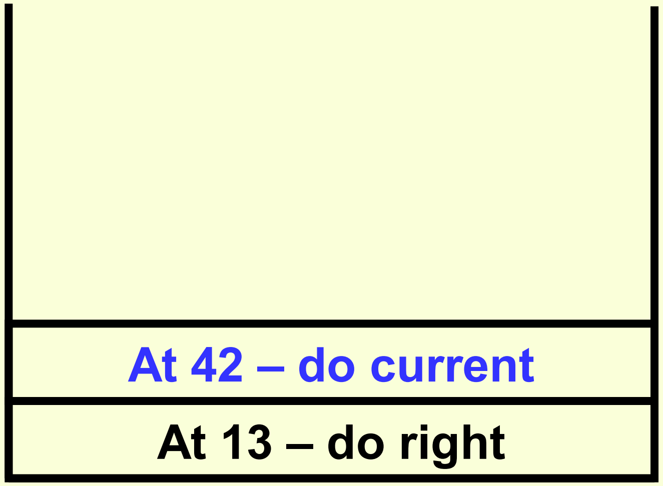
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

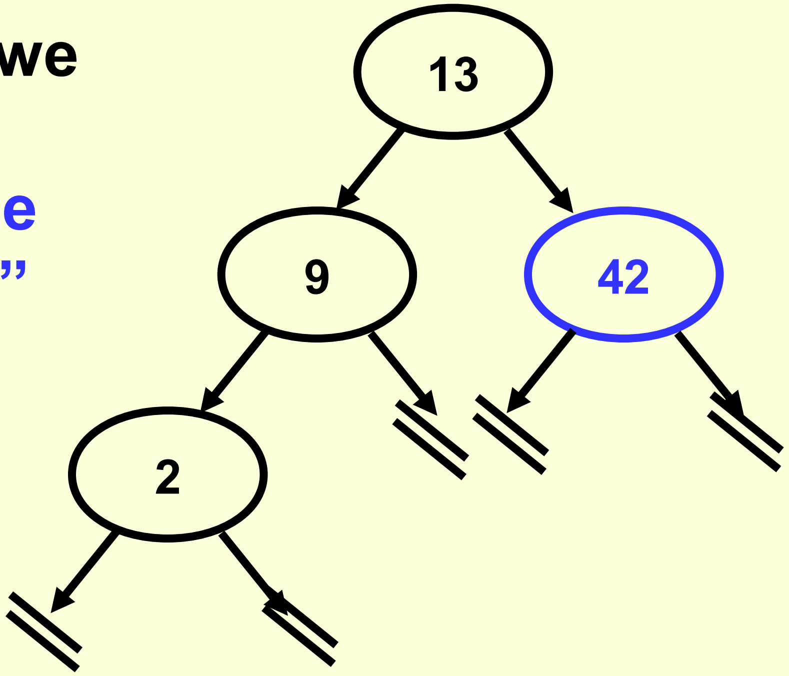




Use the Activation Stack

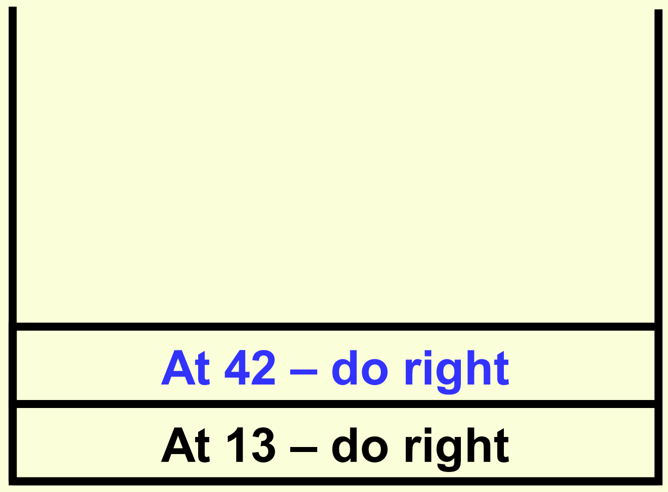
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

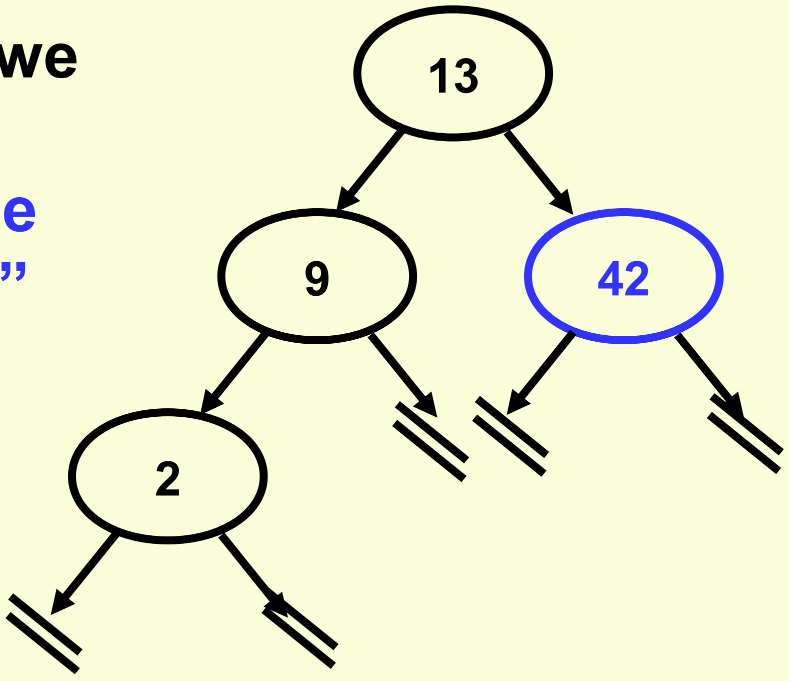




Use the Activation Stack

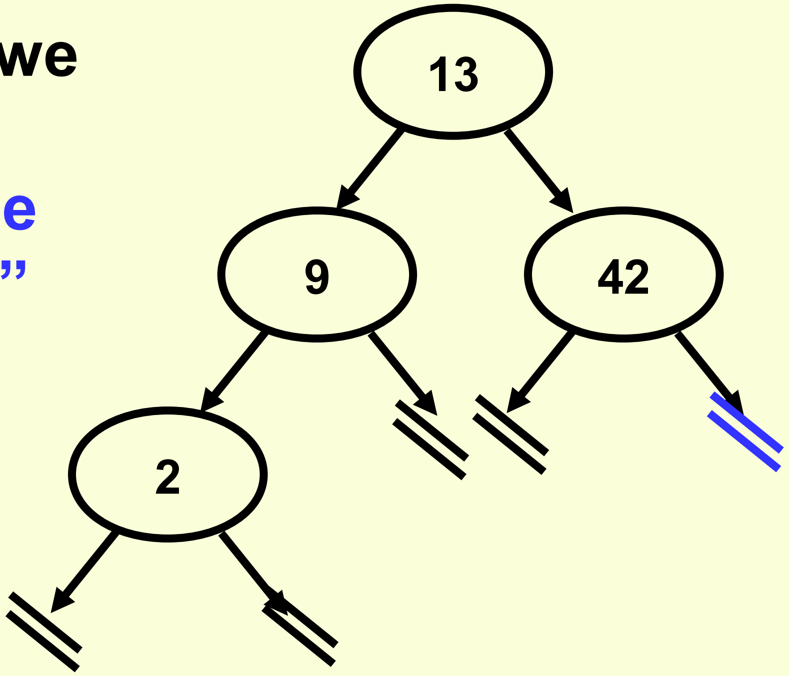
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

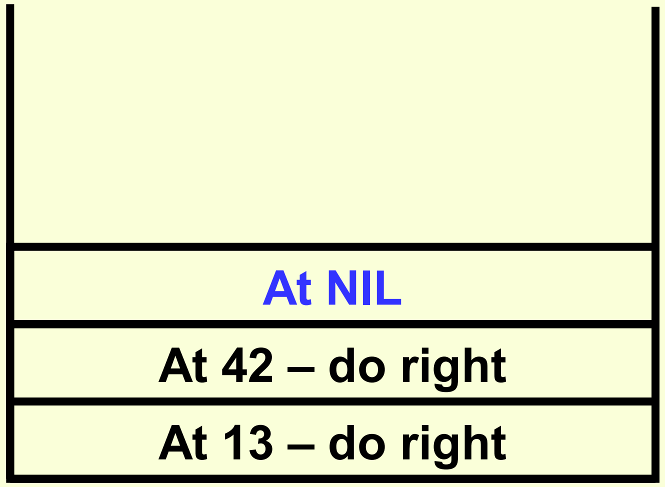




Use the Activation Stack

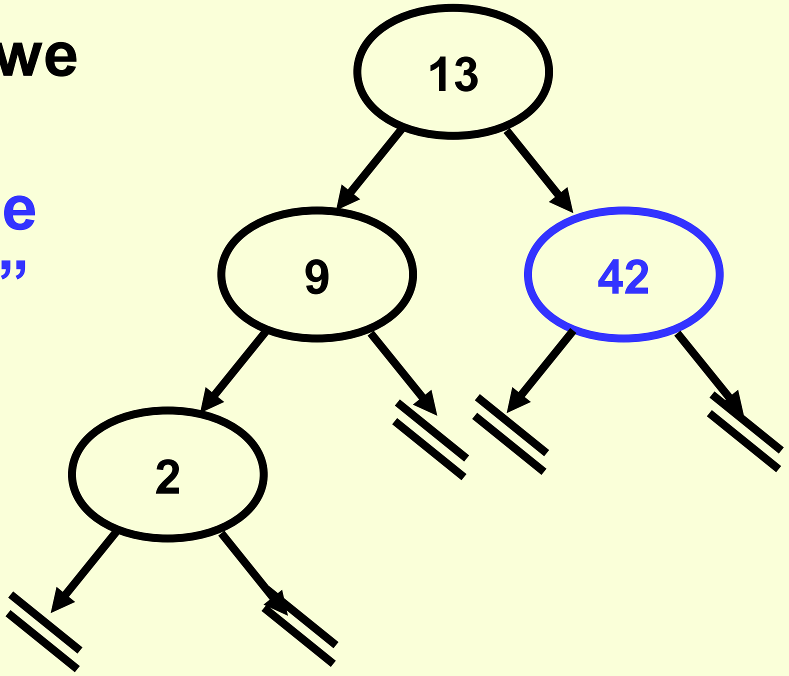
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

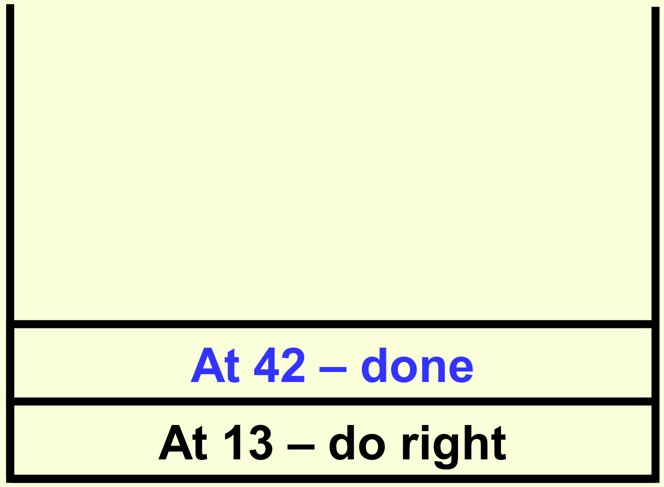




Use the Activation Stack

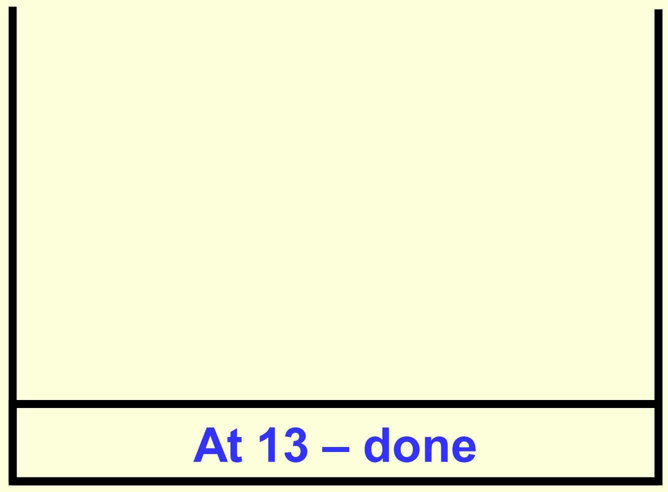
- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

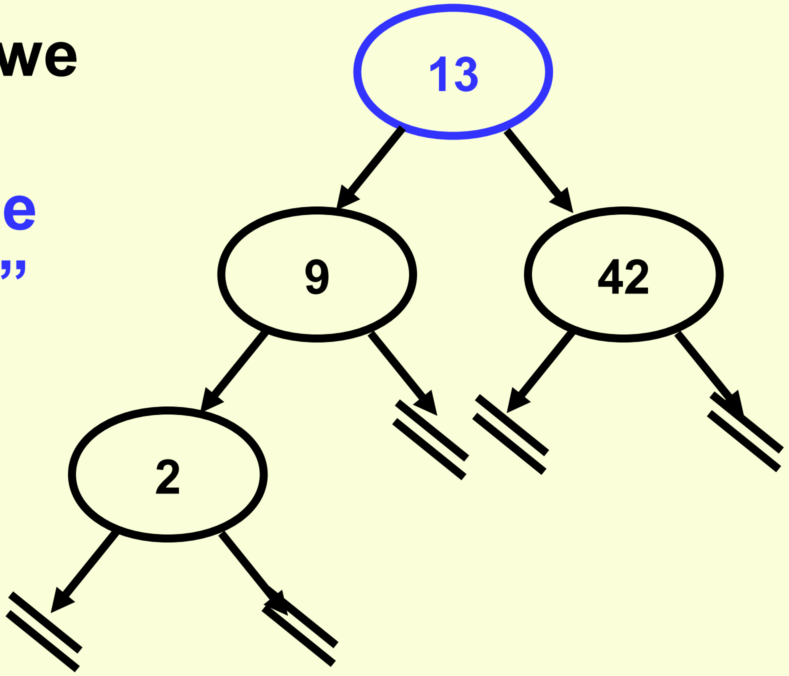




Use the Activation Stack

- With a recursive module, we can make use of the activation stack to visit the sub-trees and "remember" where we left off.

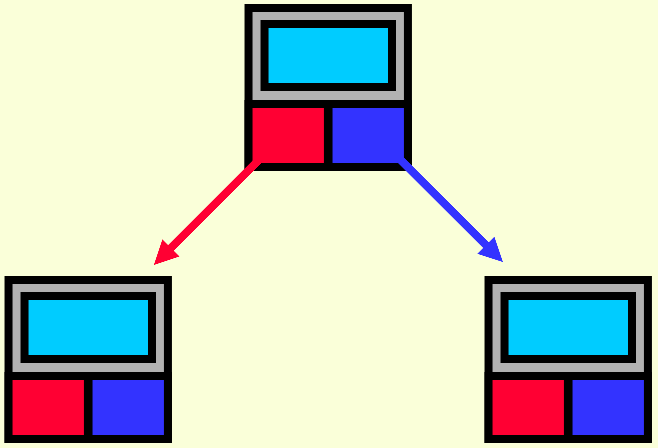




Outline of In-Order Traversal

- Three principle steps:
  - Traverse Left
  - Do work (Current)
  - Traverse Right

- Work can be anything
- Separate work from traversal

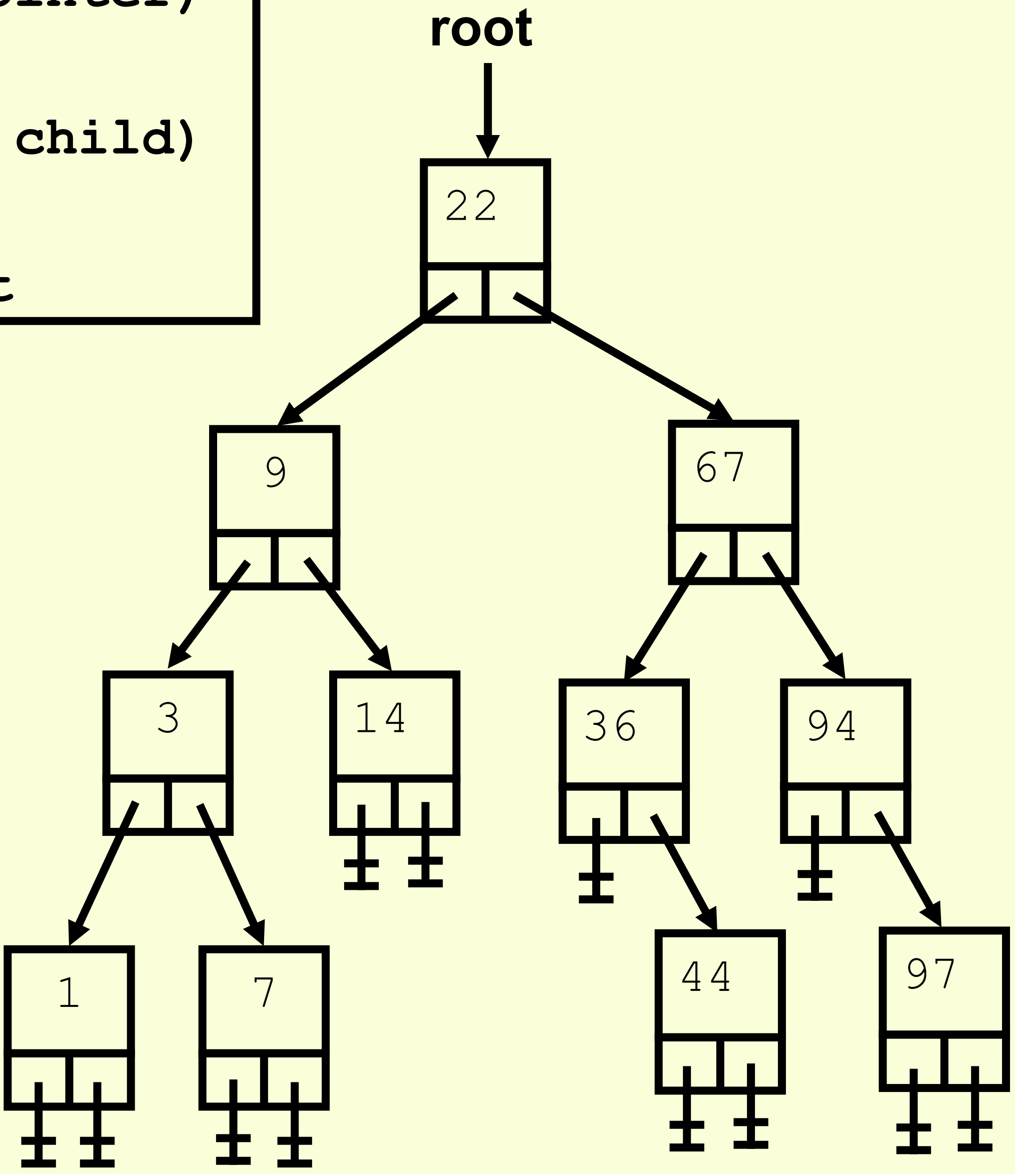


- **Traverse the tree “In order”:**
  - **Visit the tree’s left sub-tree**
  - **Visit the current and do work**
  - **Visit right sub-tree**

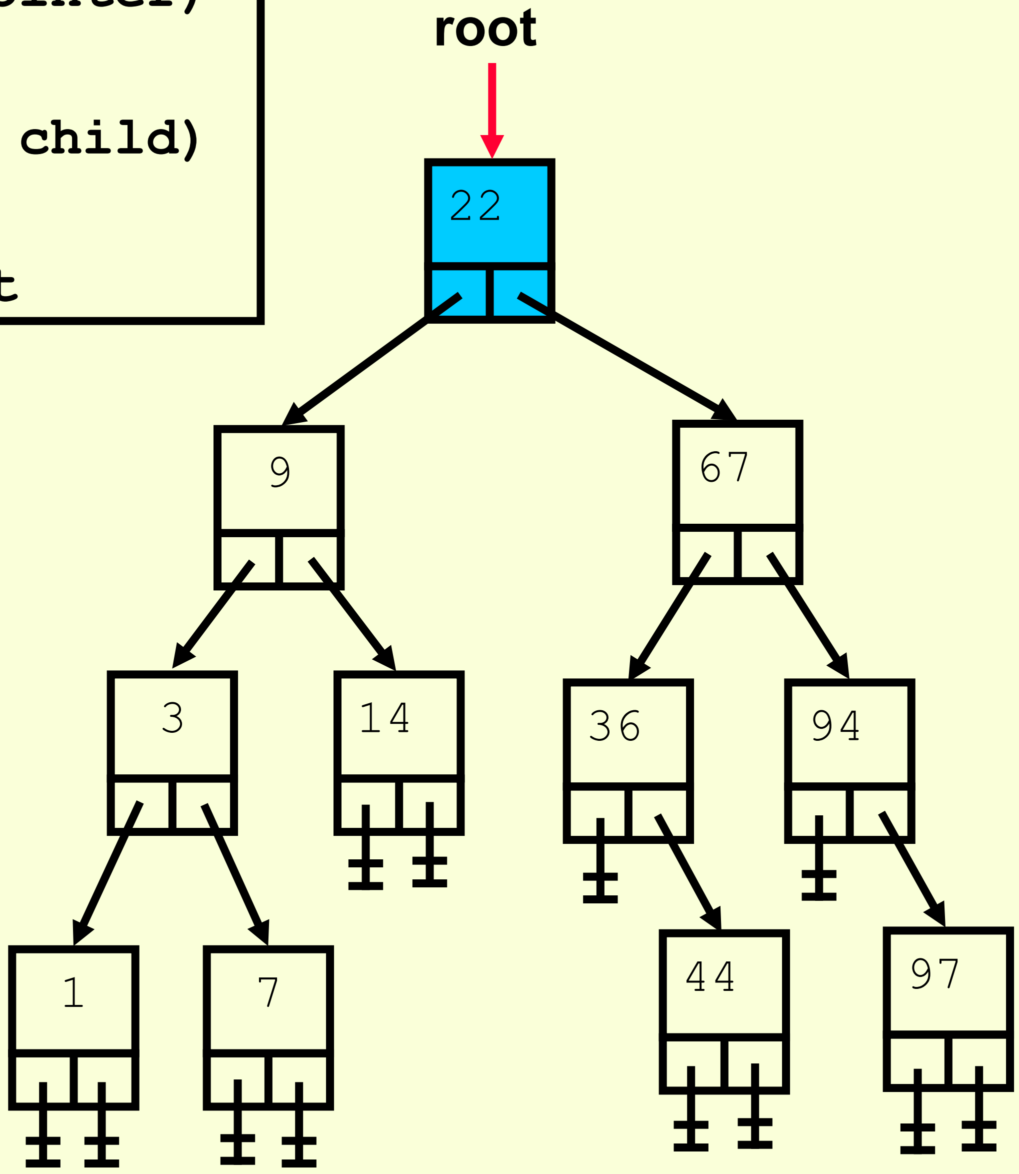
In-Order Traversal Procedure

```
procedure In_Order(cur iot in Ptr toa Tree_Node)
// Purpose: perform in-order traversal, call
//          Do_Something for each node
// Preconditions: cur points to a binary tree
// Postcondition: Do_Something on each tree
//                node in “in-order” order
  if( cur <> NIL ) then
    In_Order( cur^.left_child )
    Do_Something( cur^.data )
    In_Order( cur^.right_child )
  endif
endprocedure   // In_Order
```

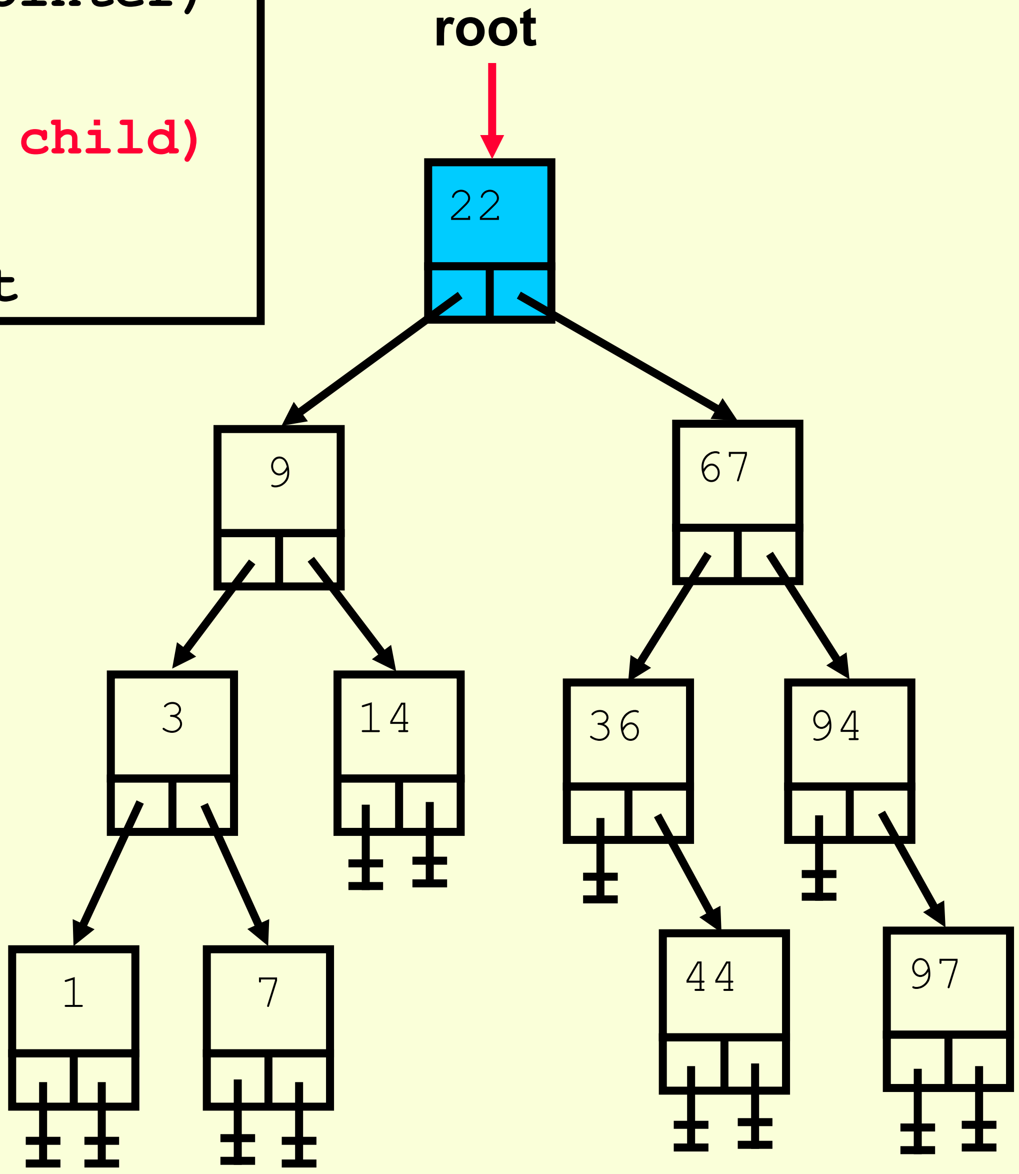
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right
child)
```



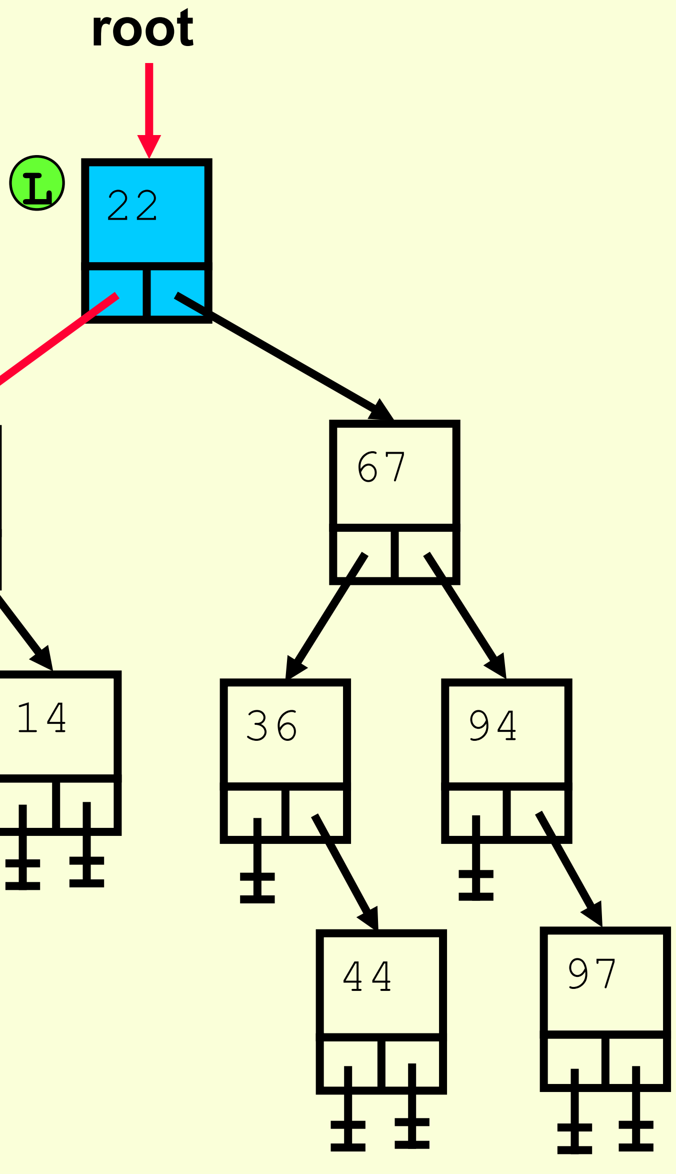
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```



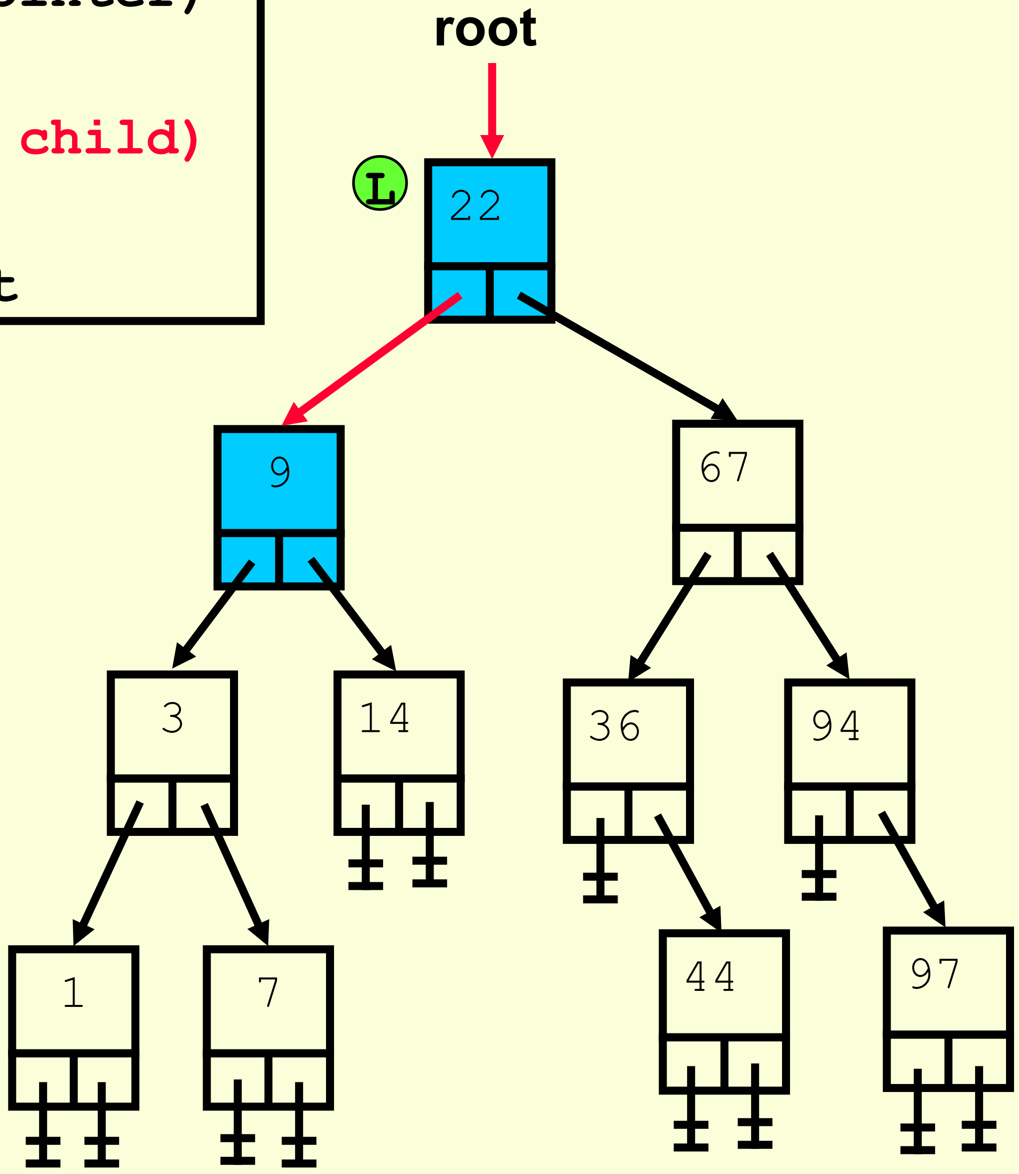
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```



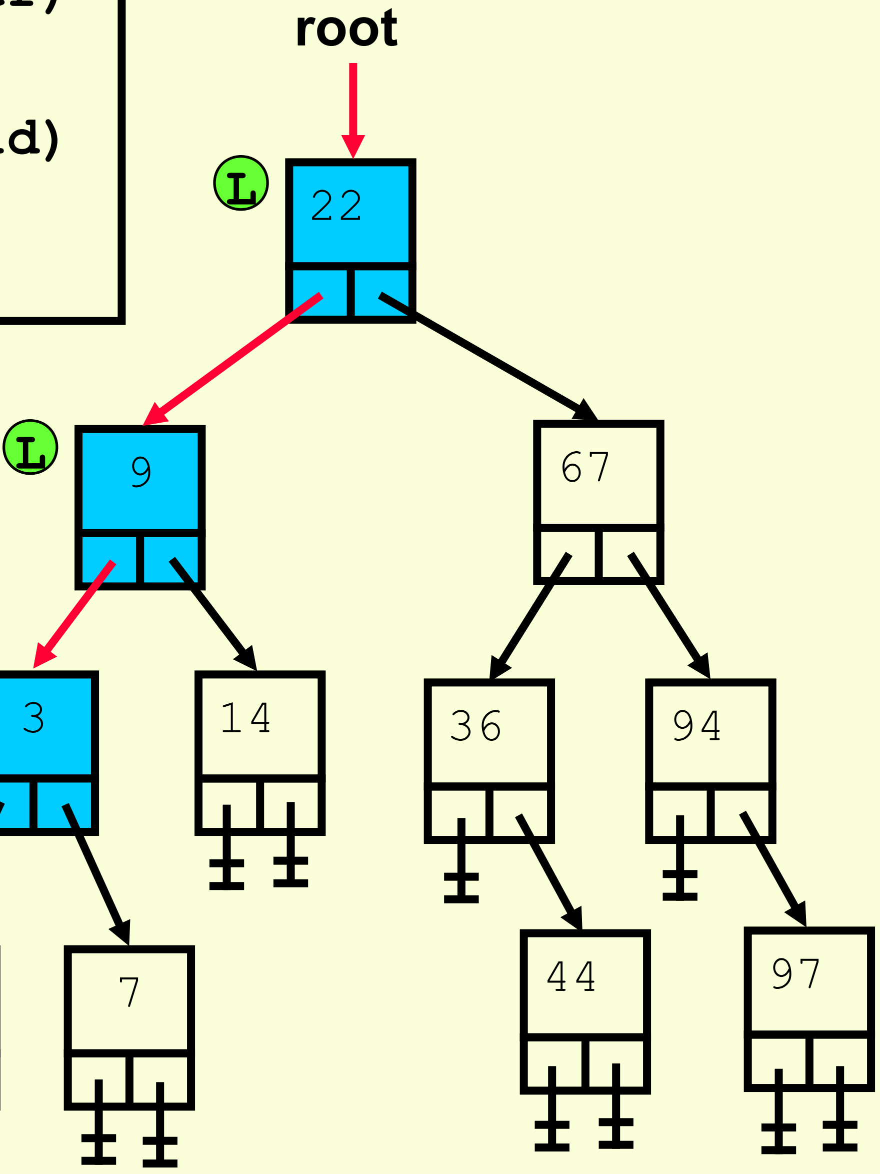
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```



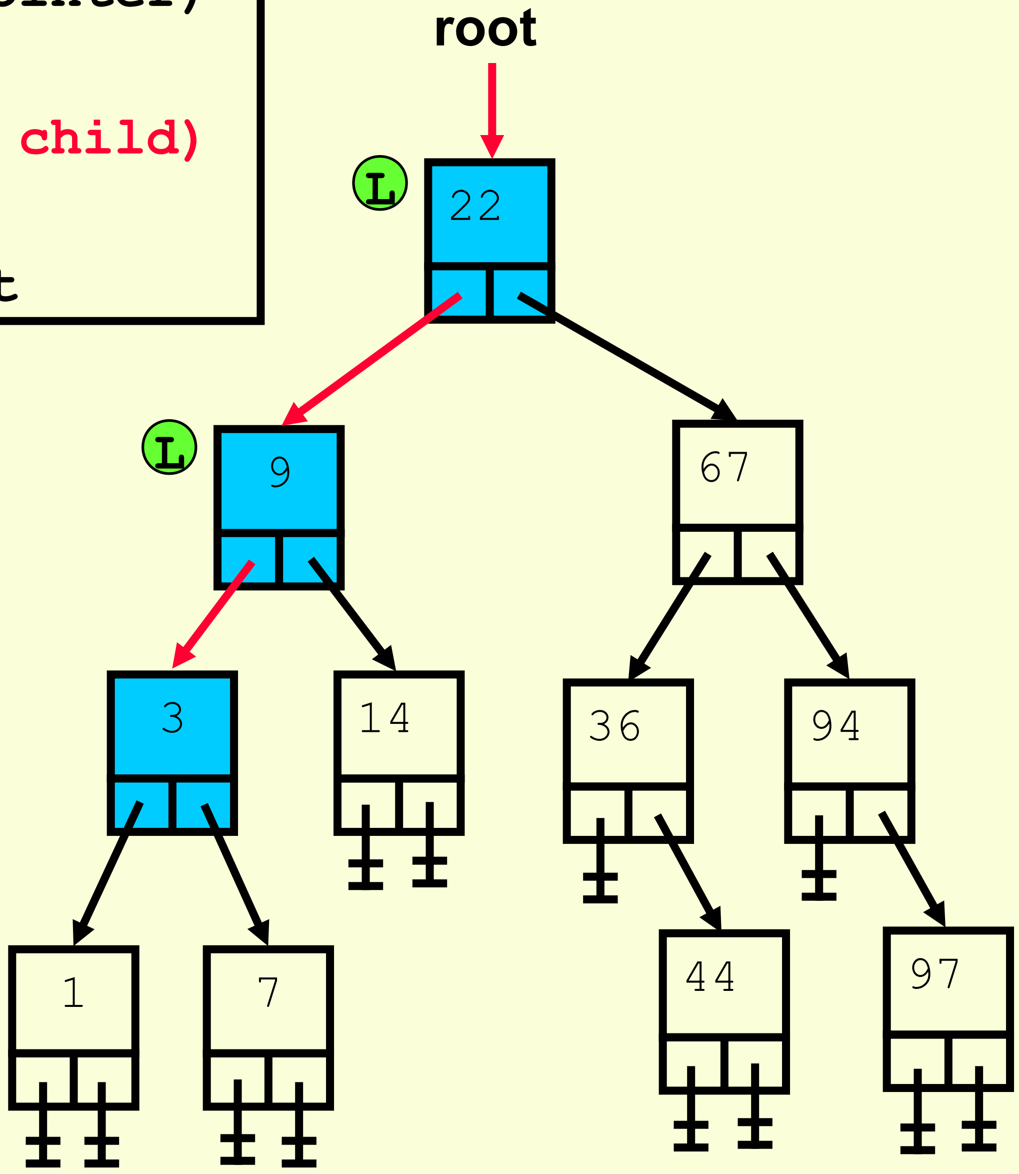
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right
child)
```



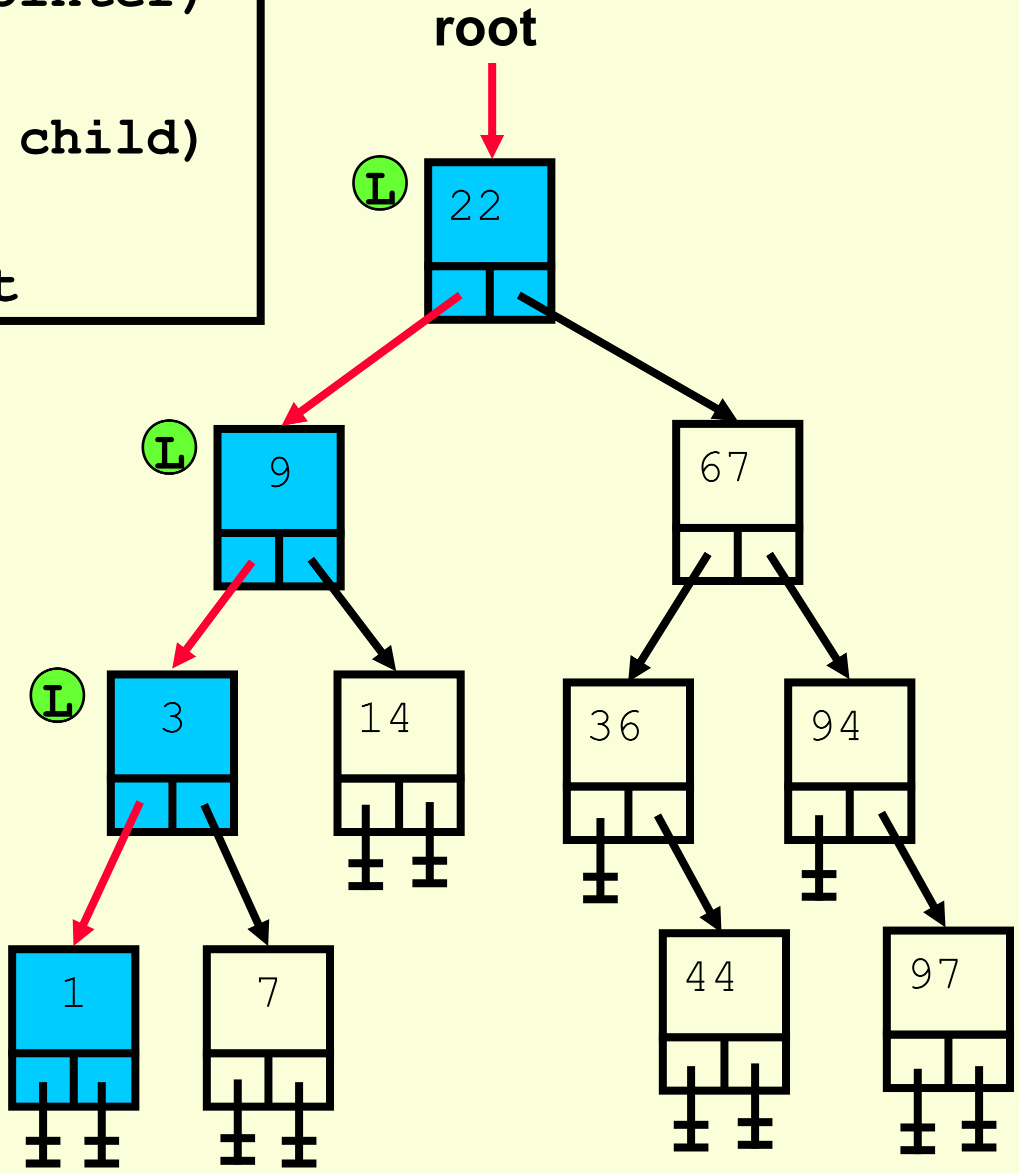
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```



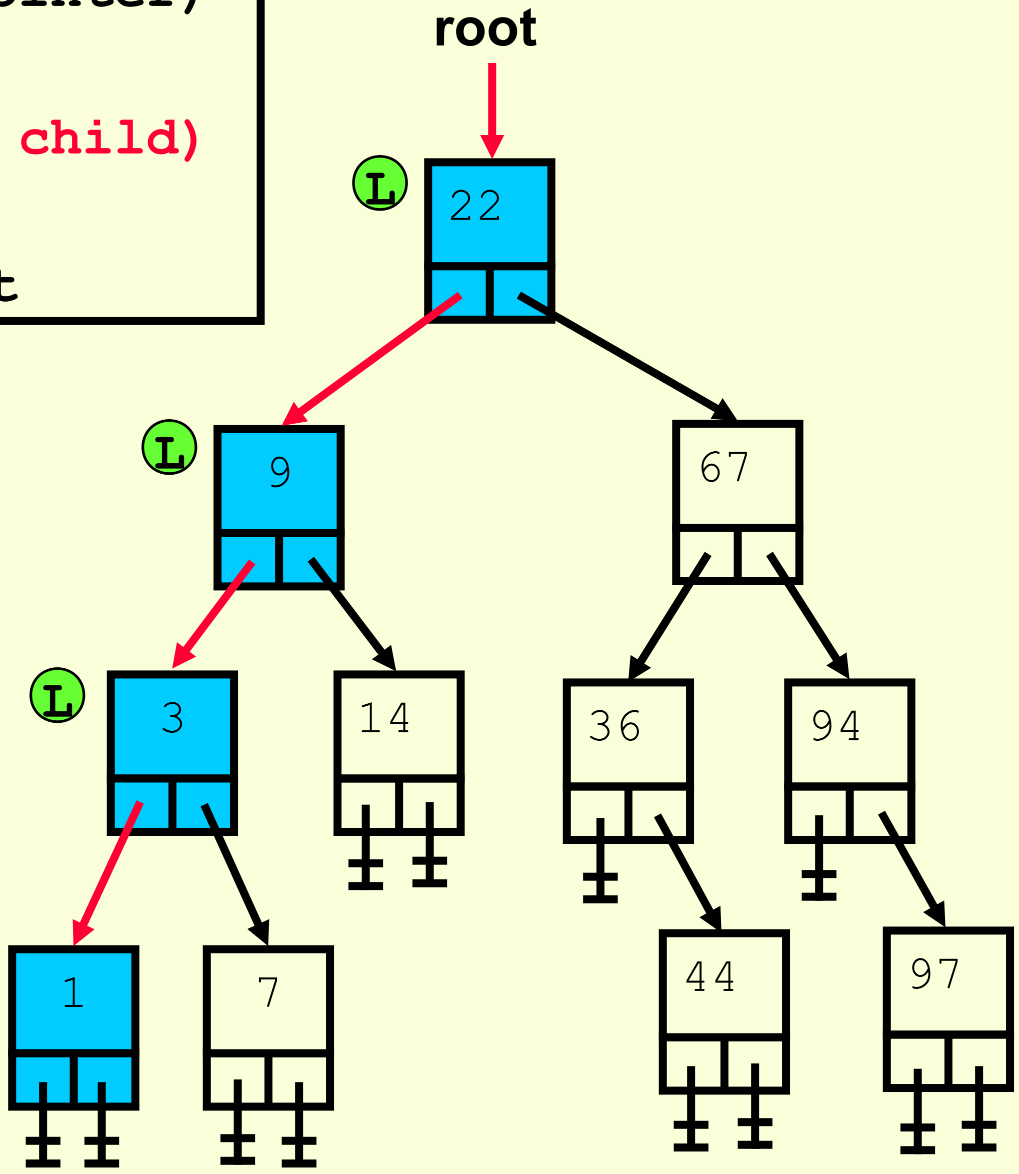
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right
child)
```



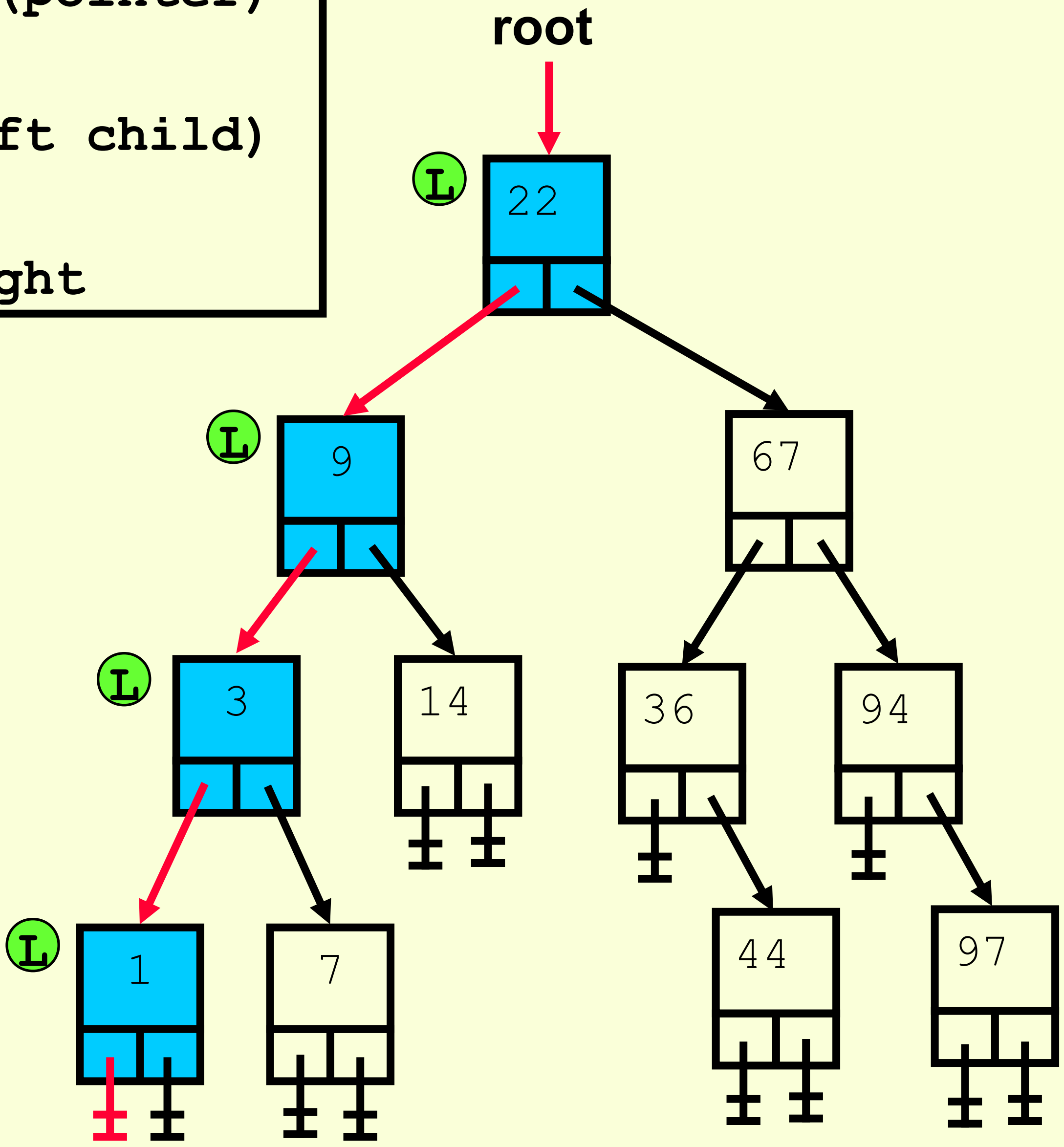
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```



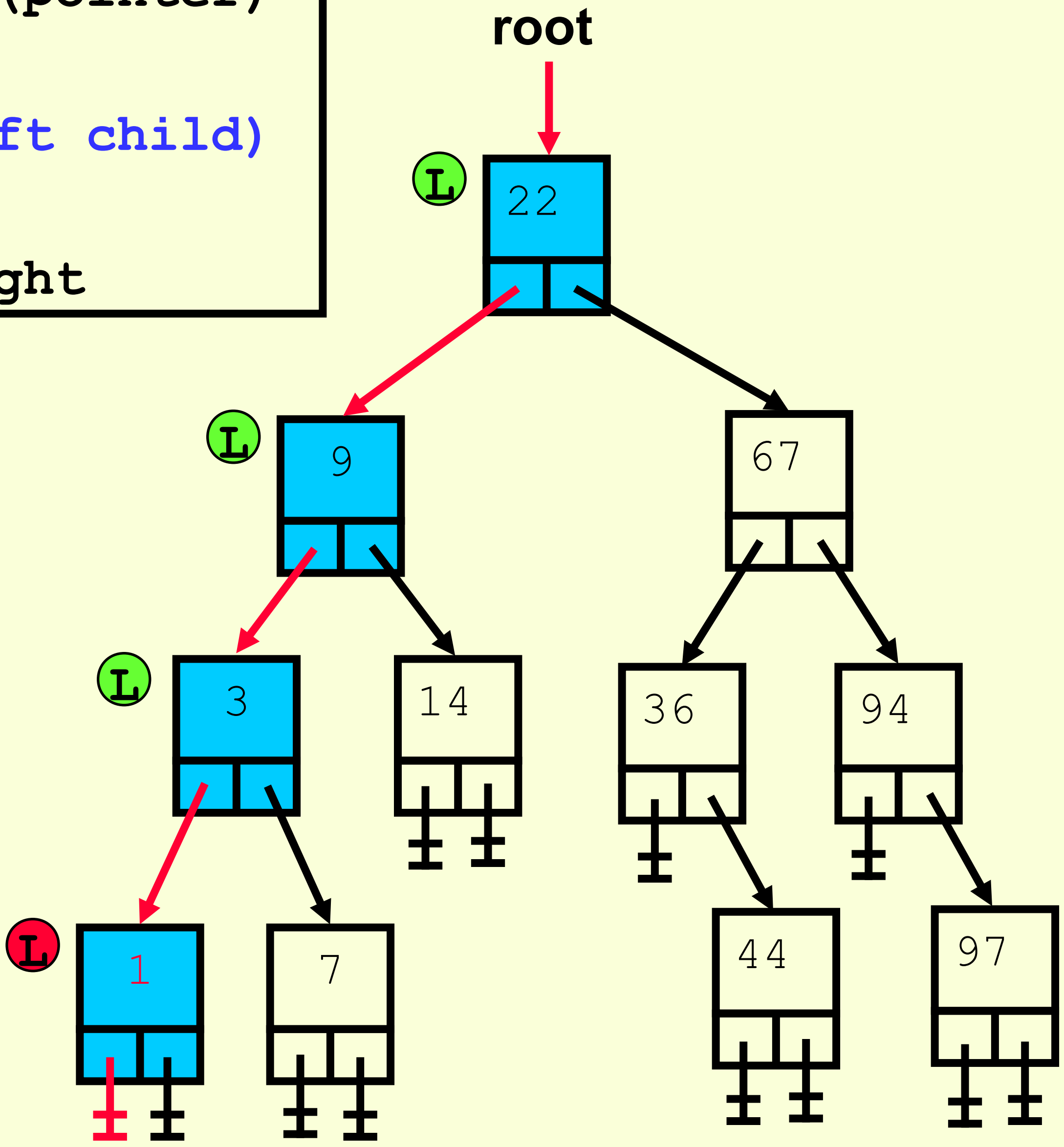
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```



```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right
child)
```

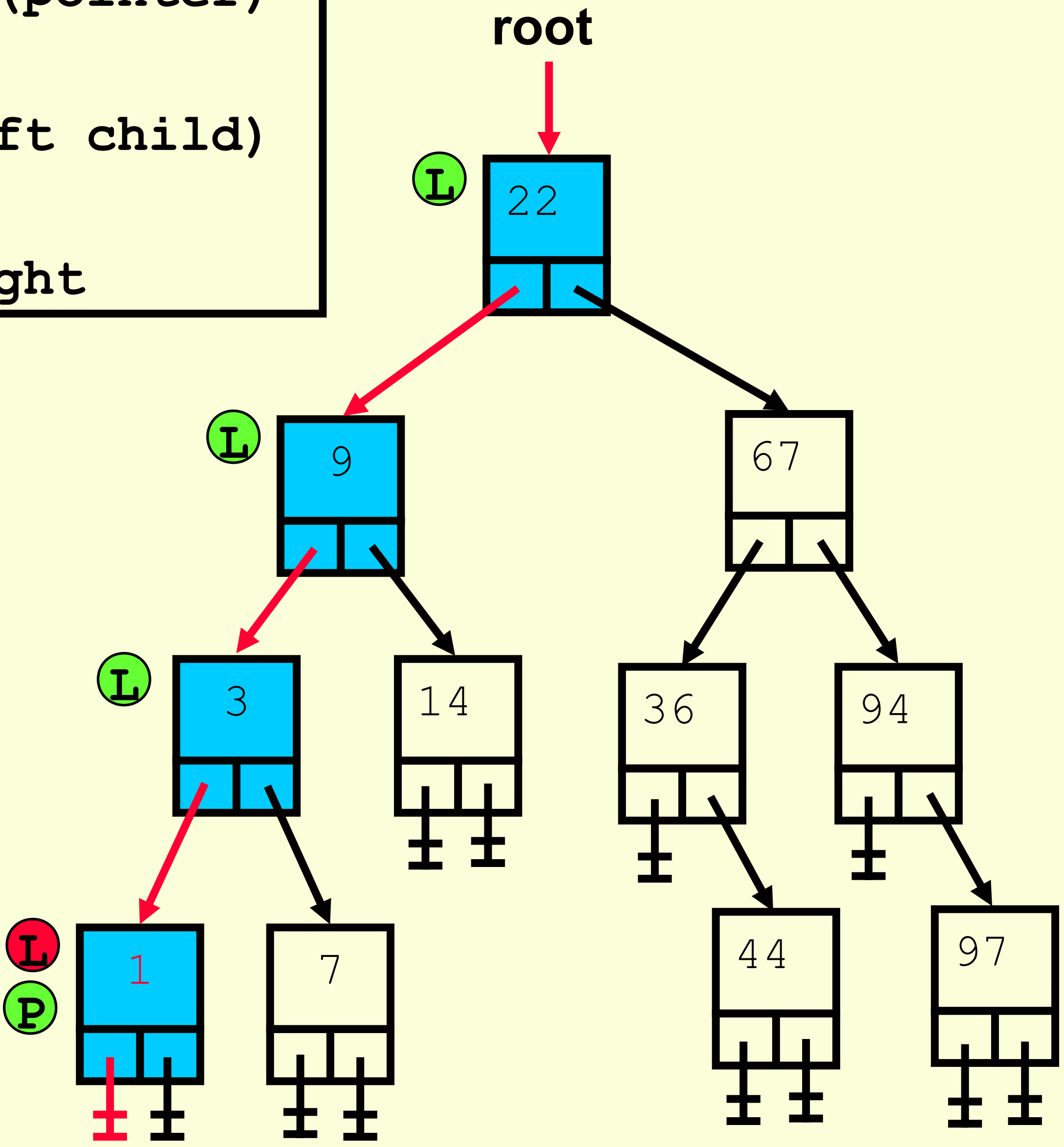


```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right
child)
```



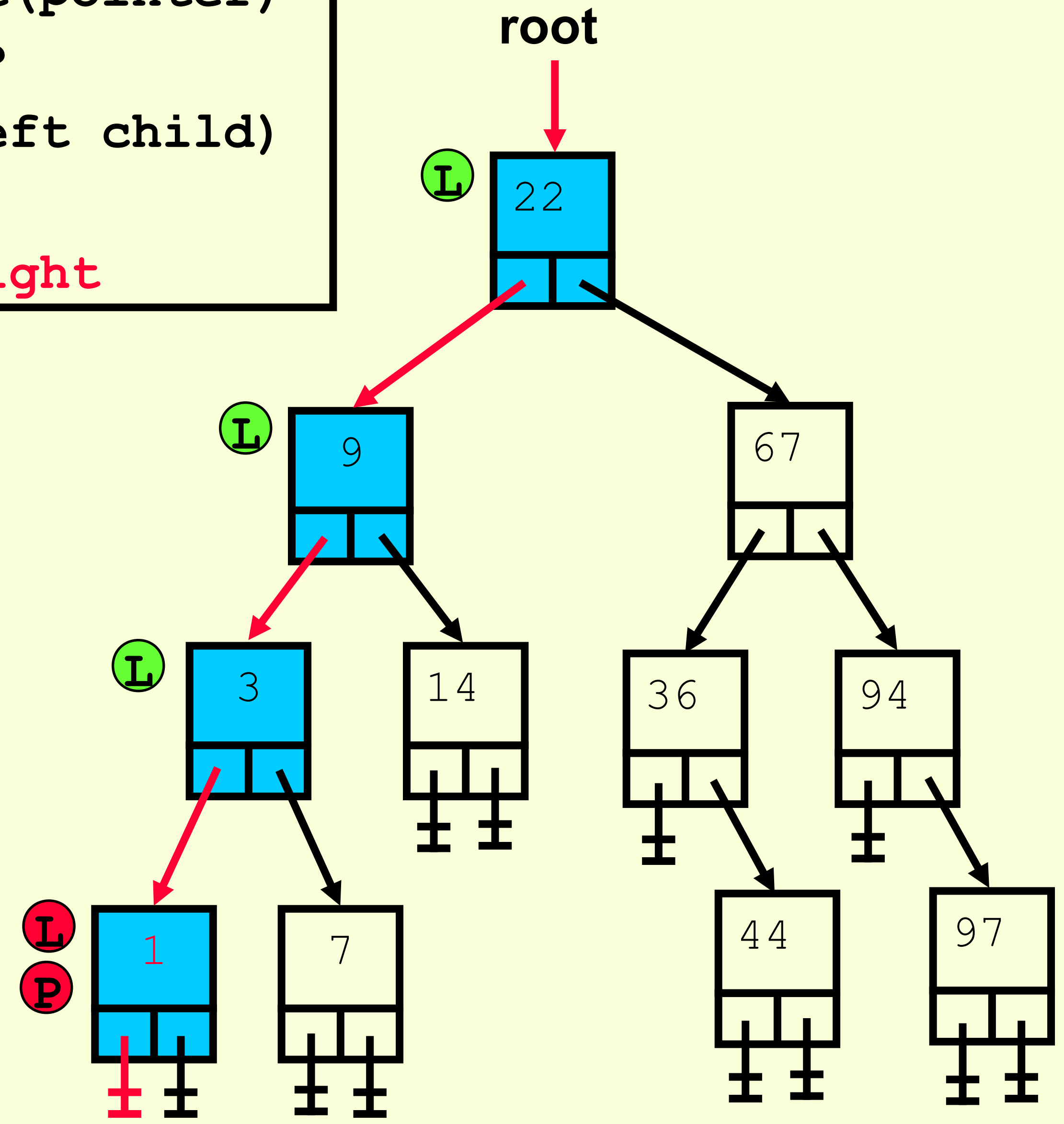
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1
```



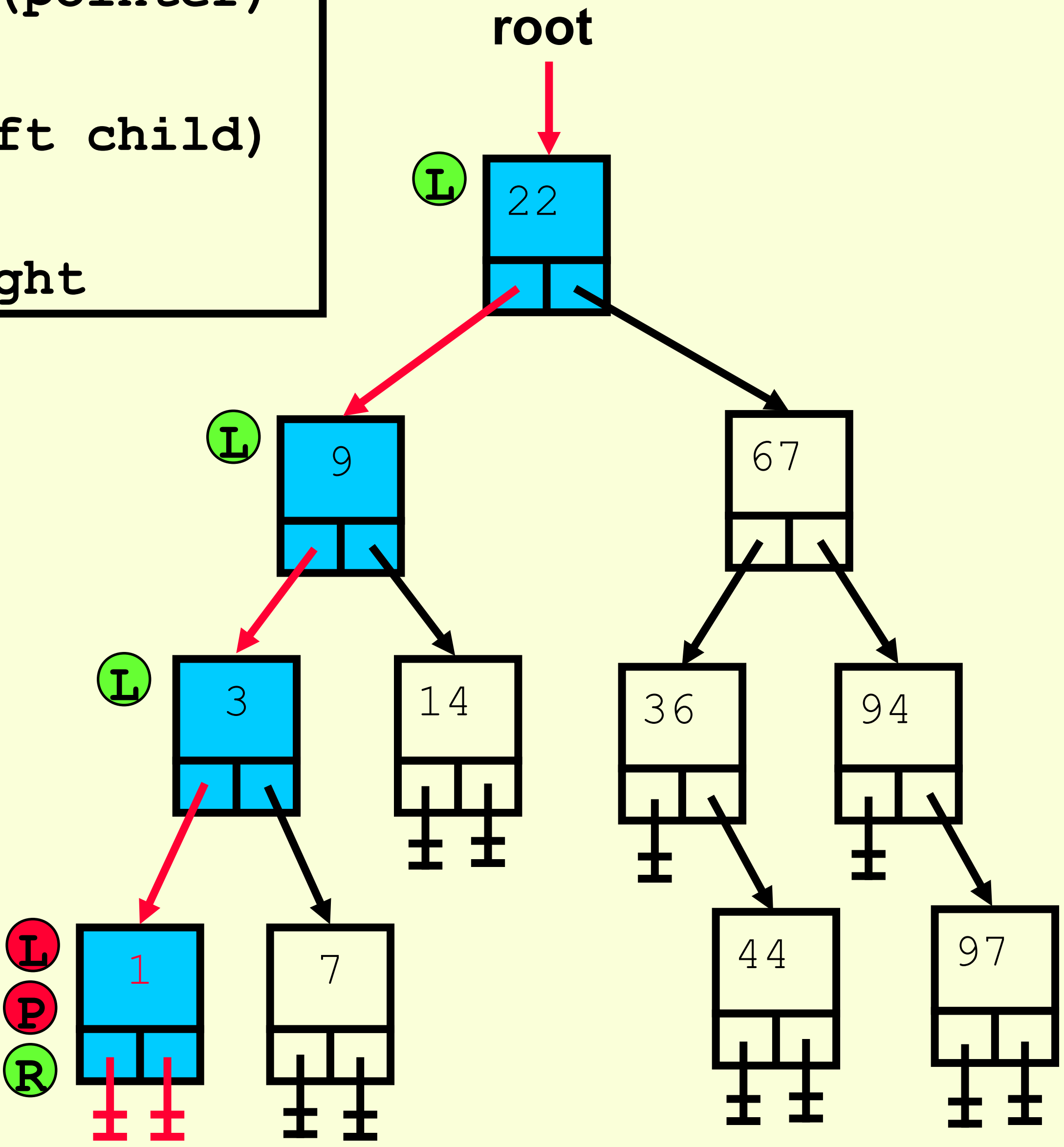
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1
```



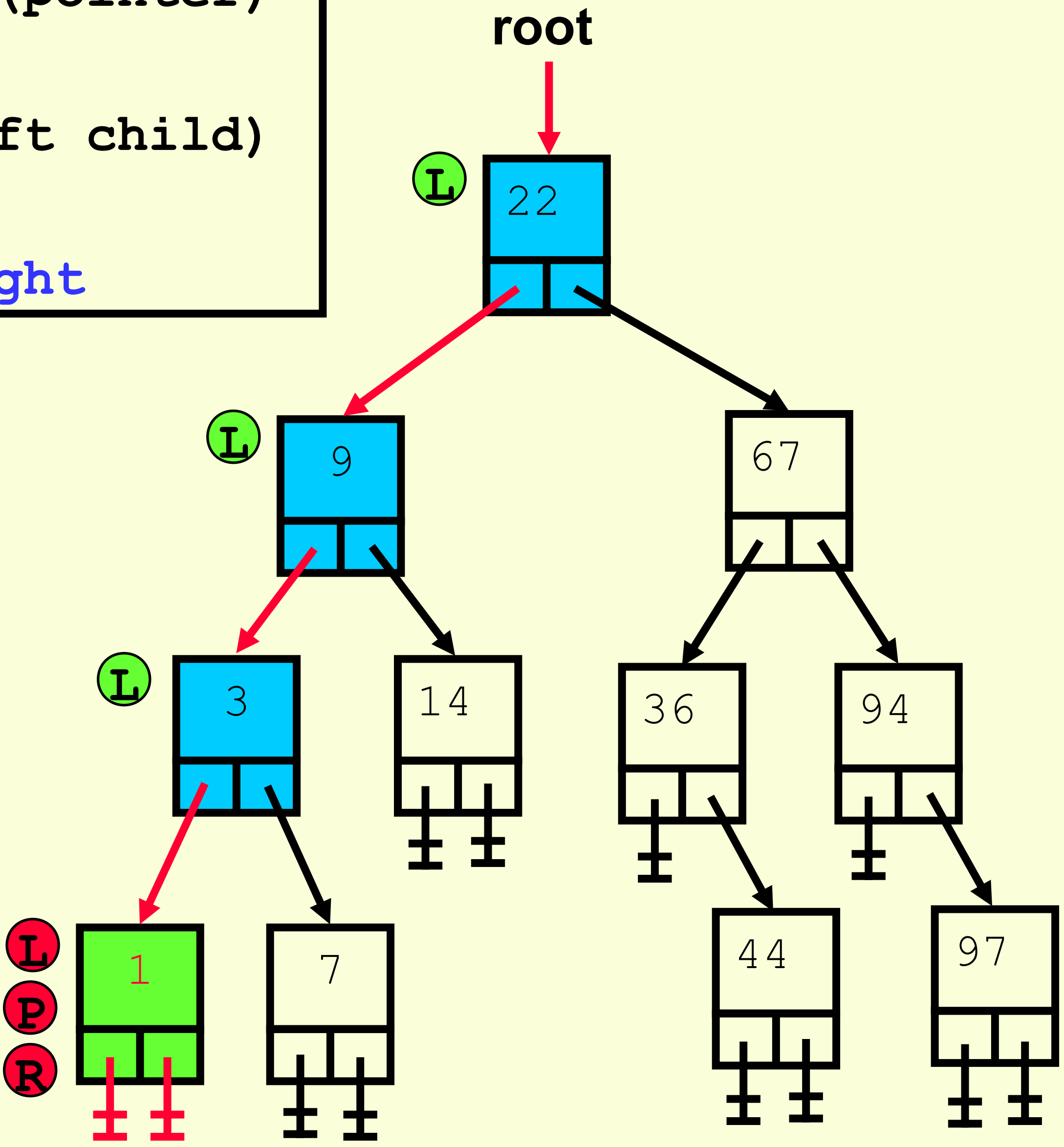
```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
  L InOrderPrint(left child)
  P print(data)
  R InOrderPrint(right
child)
```

```
Output: 1
```



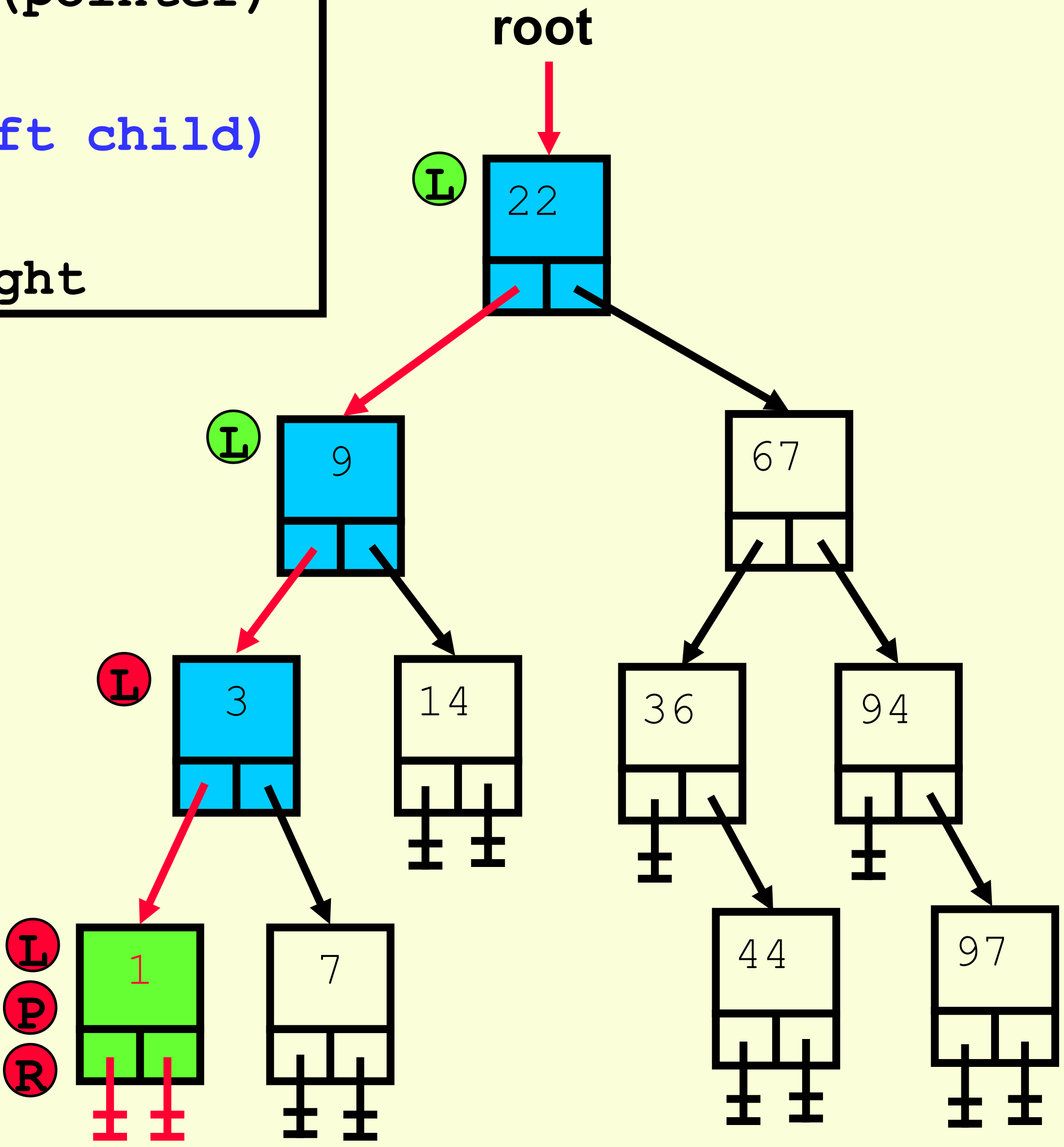
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right child)
```

```
Output: 1
```



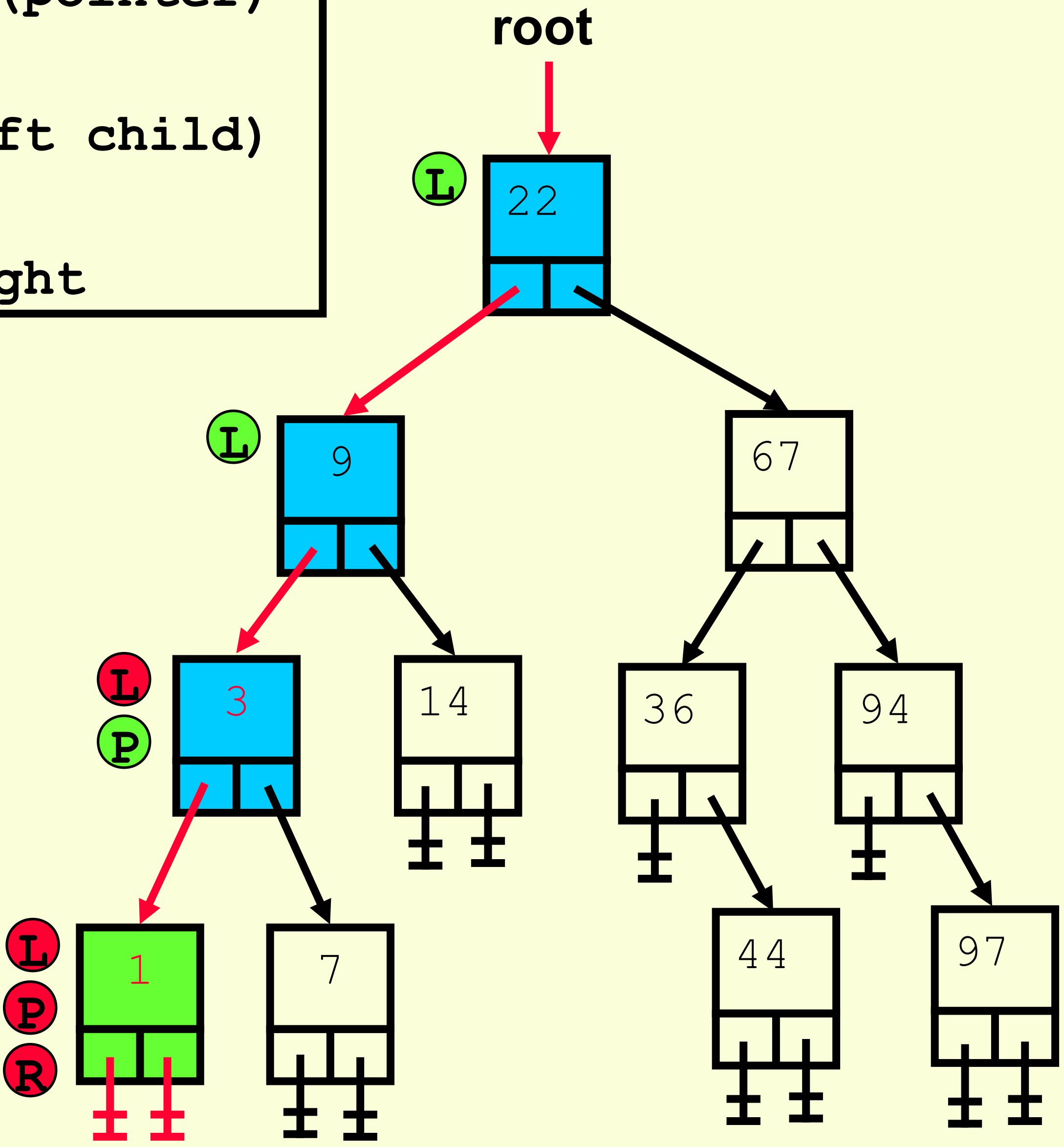
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right
child)
```

```
Output: 1
```



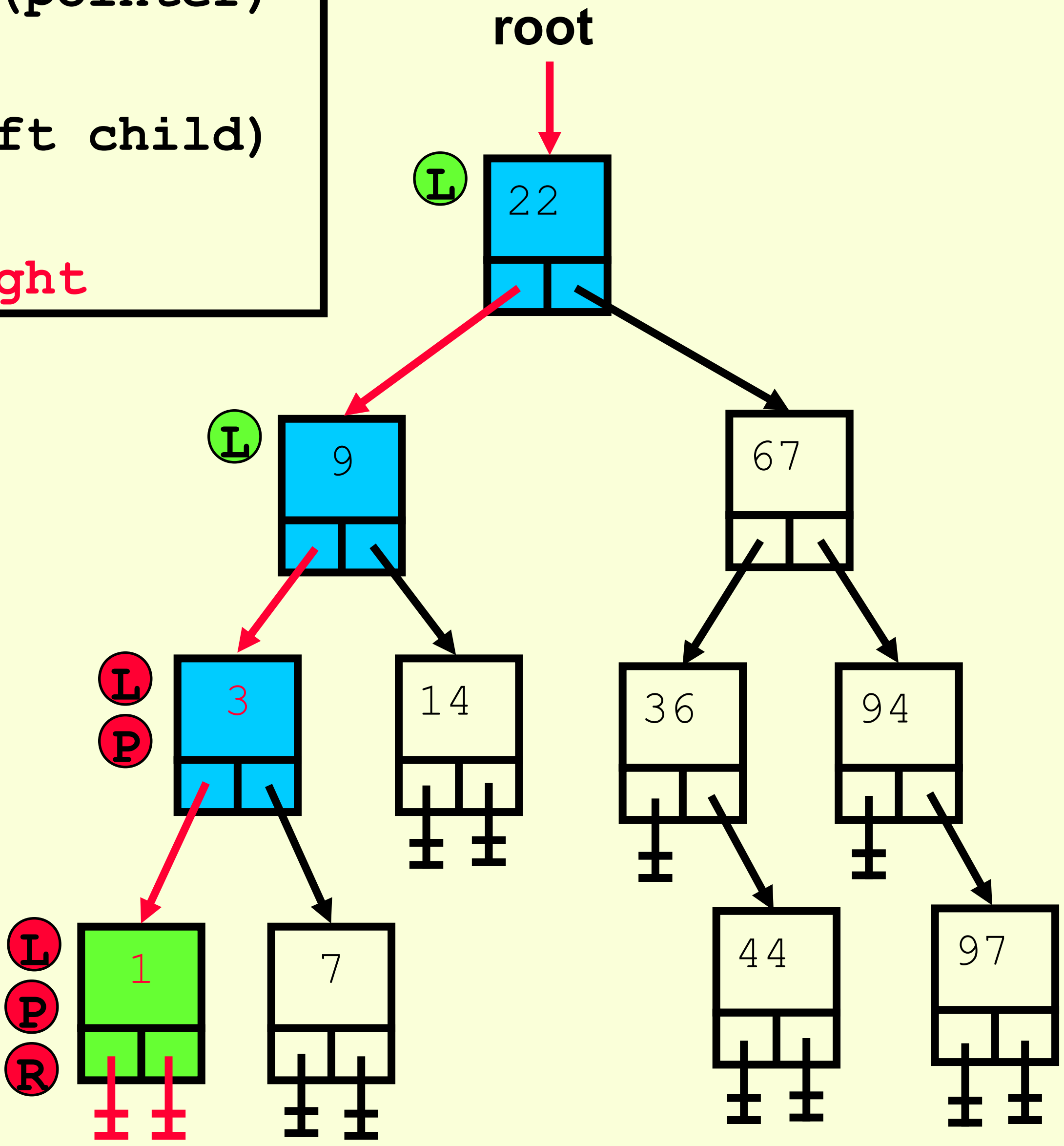
```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right
child)
```

```
Output: 1 3
```



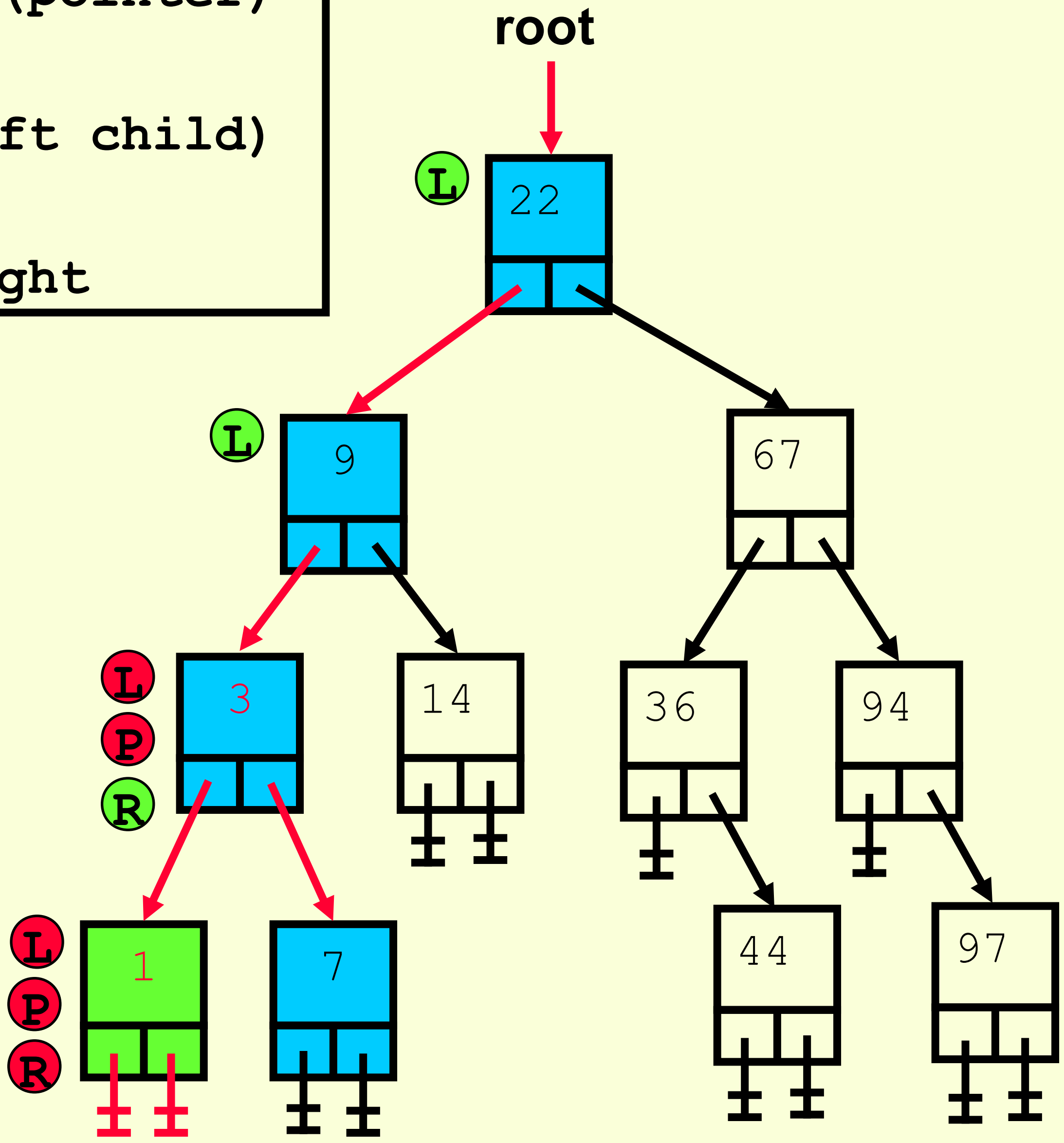
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right child)
```

```
Output: 1 3
```



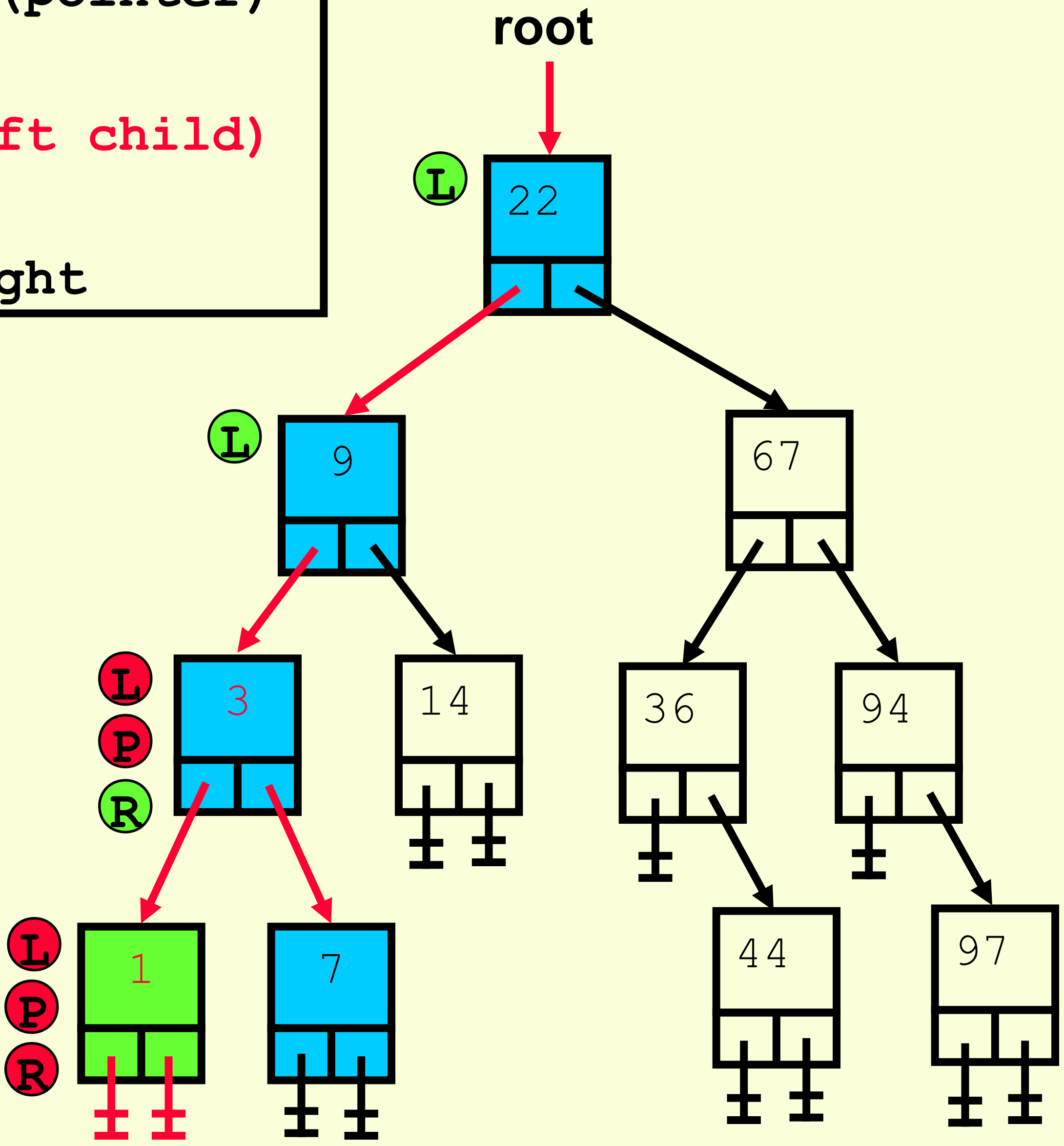
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3
```



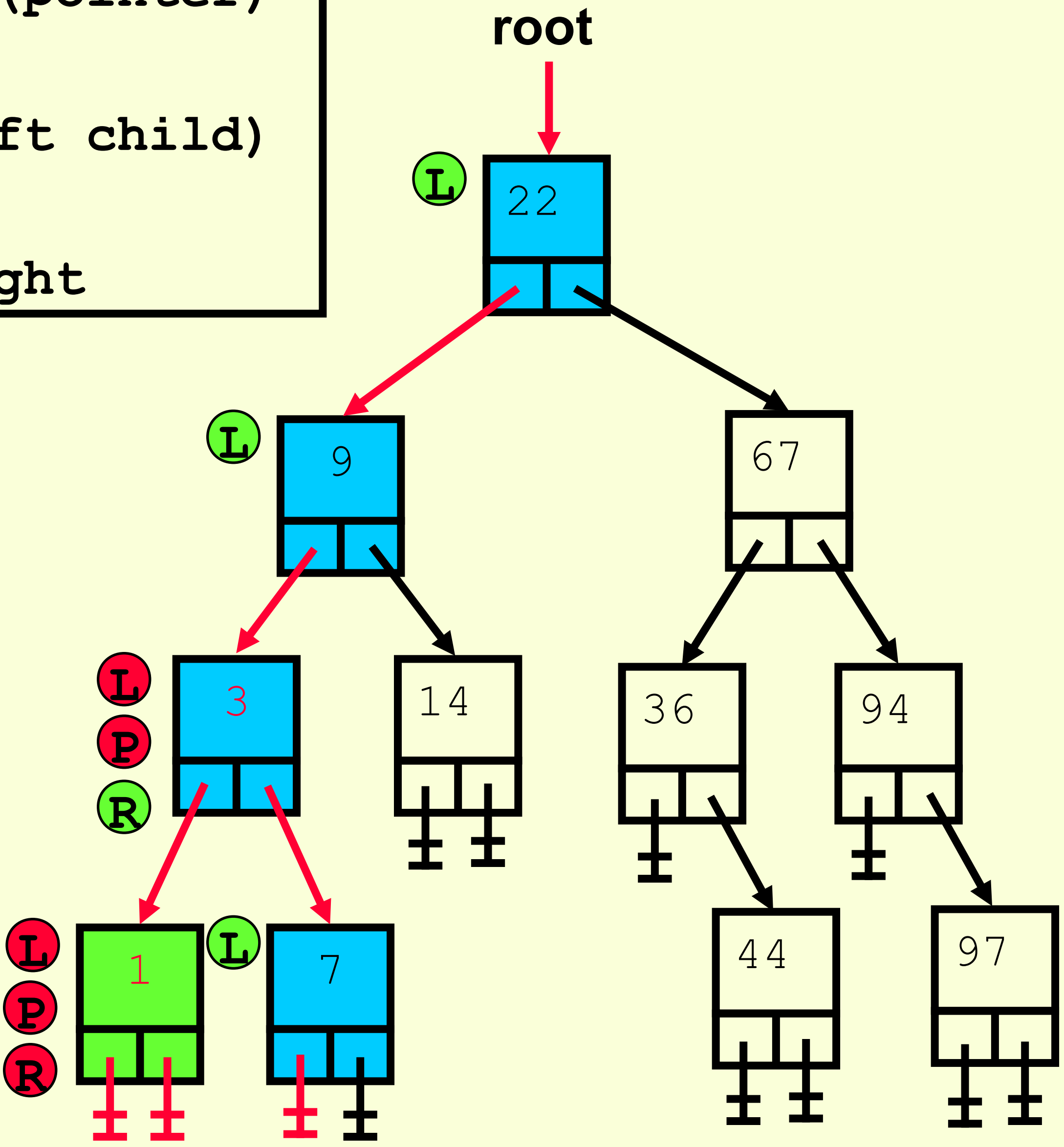
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3
```



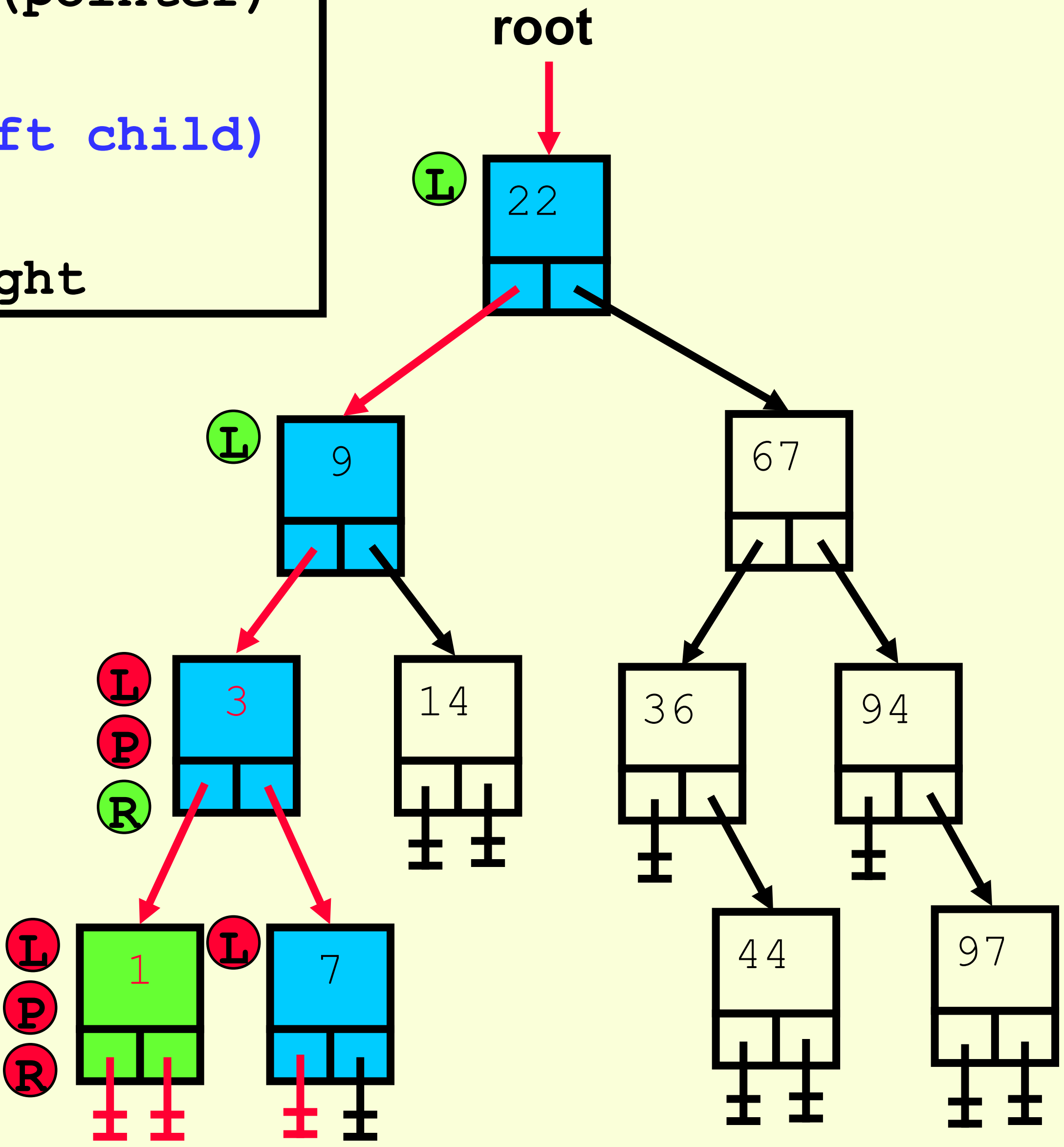
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3
```



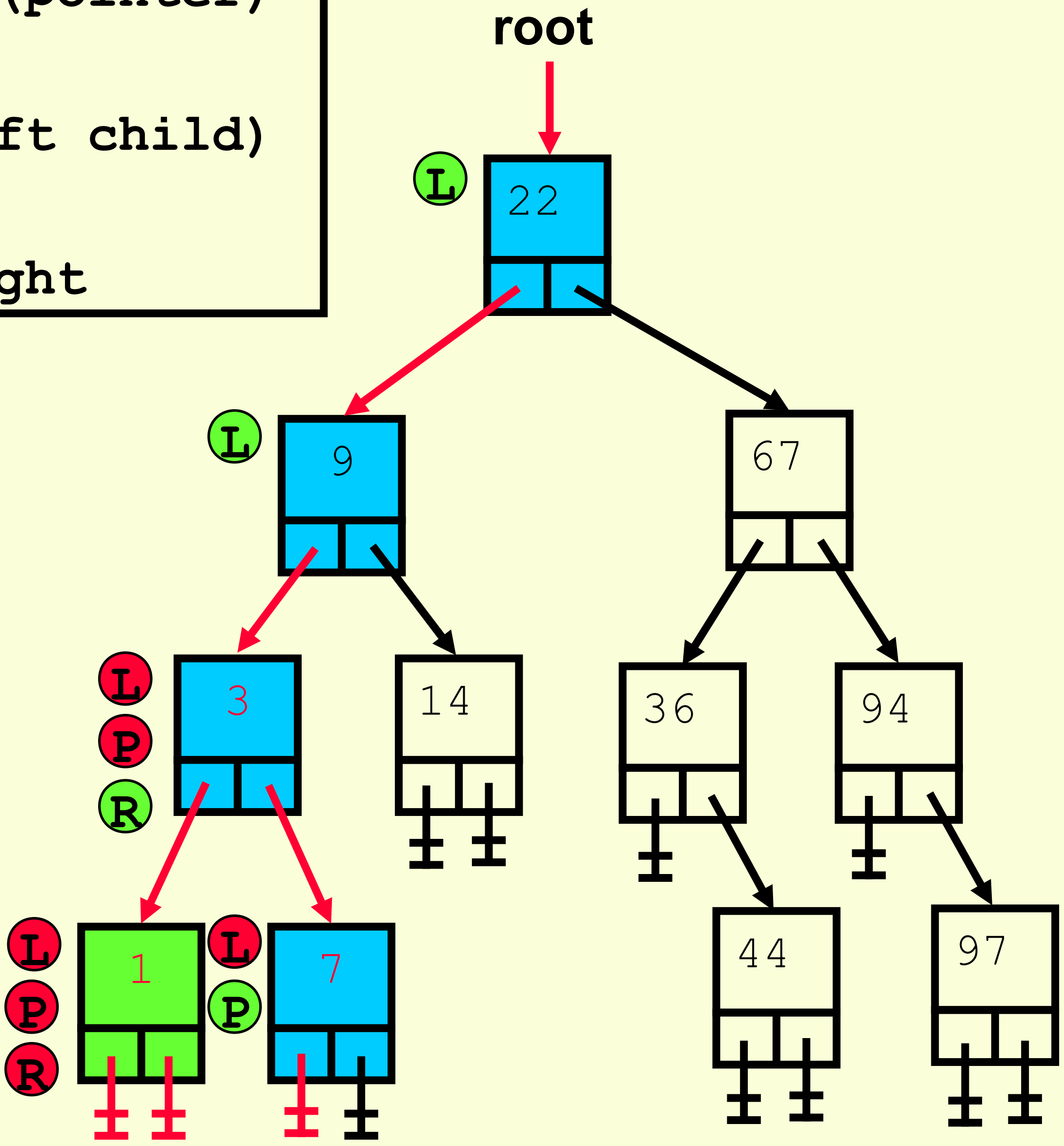
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3
```



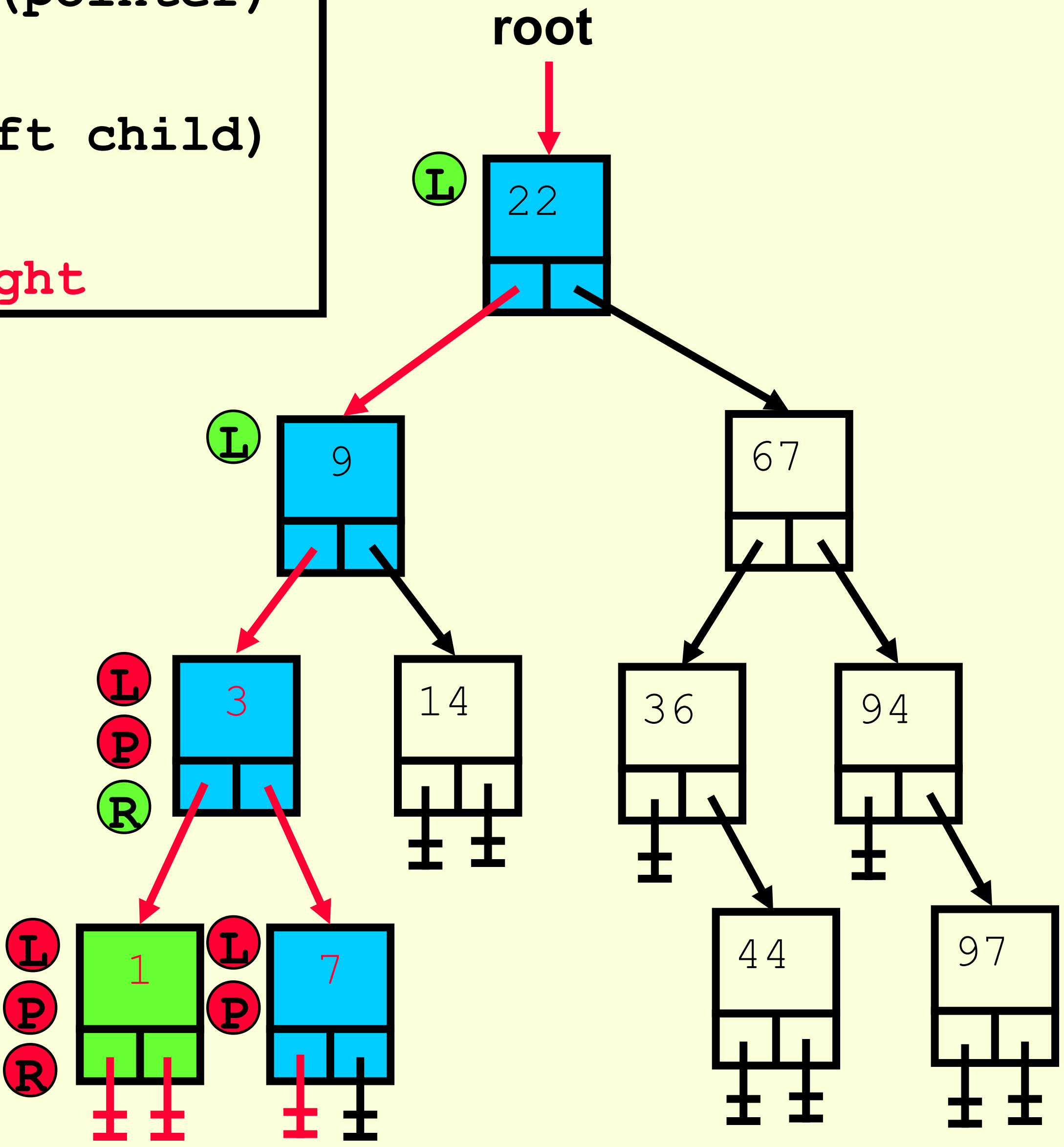
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7
```



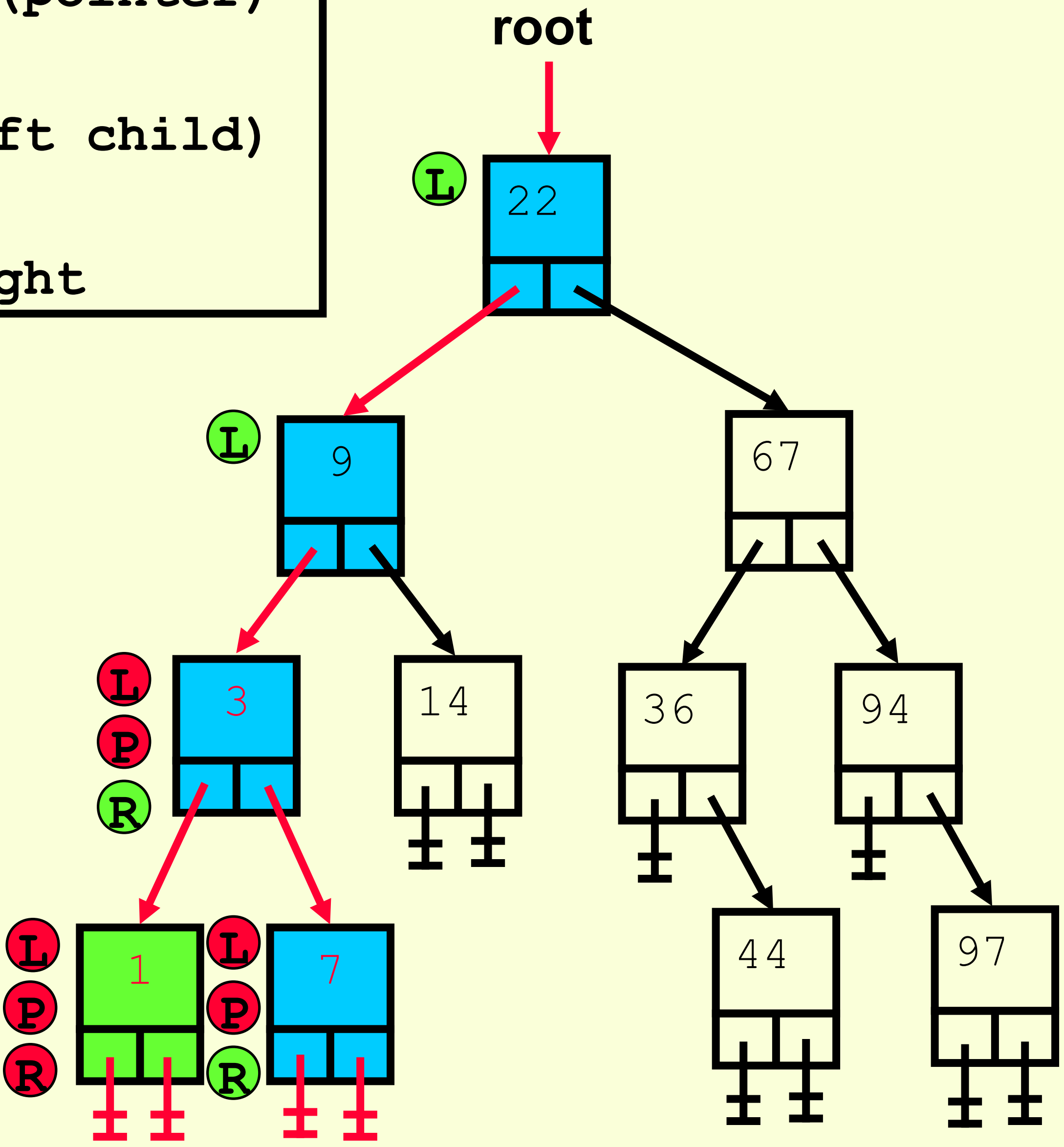
```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L   InOrderPrint(left child)
P   print(data)
R   InOrderPrint(right child)
```

```
Output: 1 3 7
```



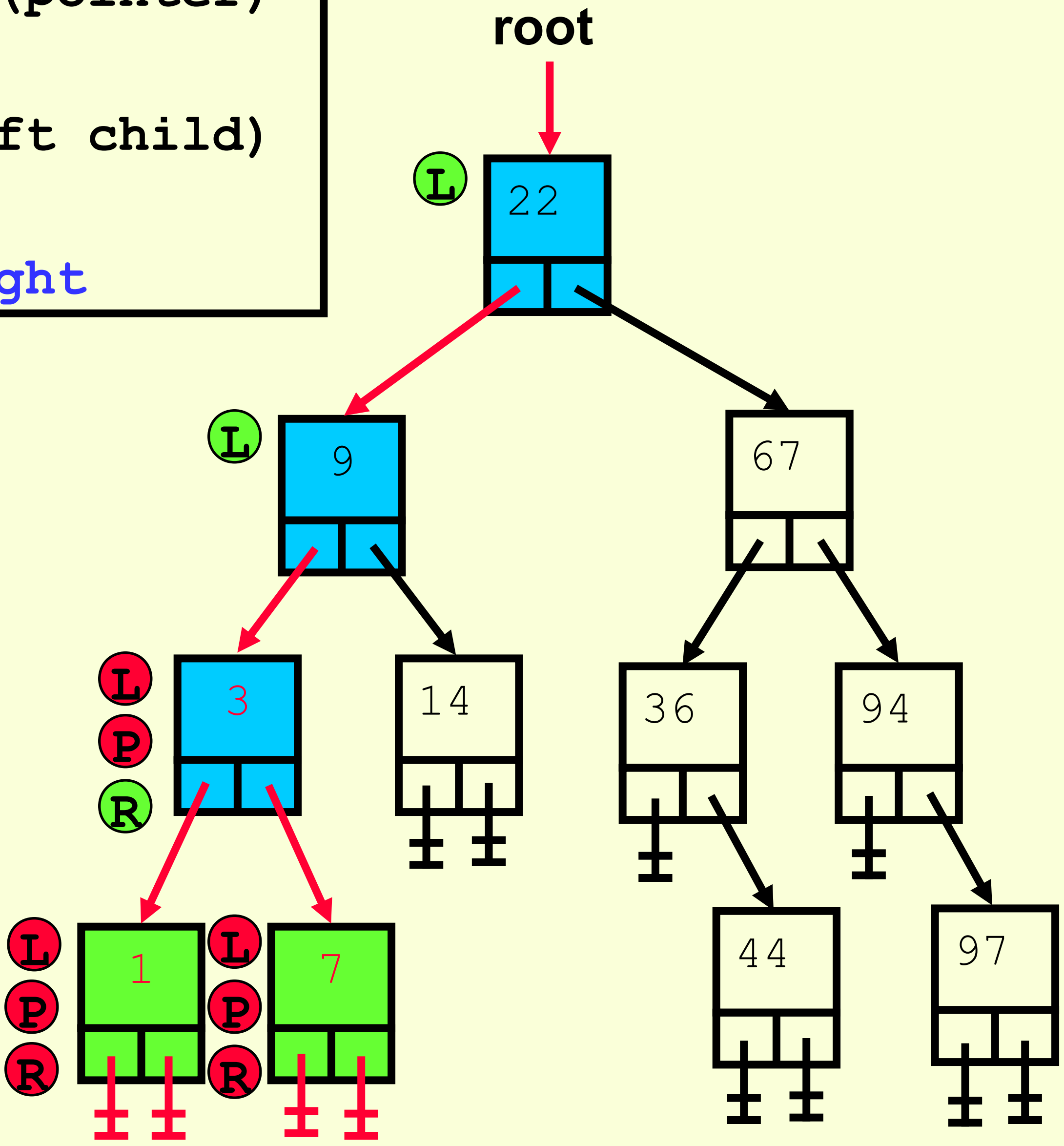
```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right
child)
```

```
Output: 1 3 7
```



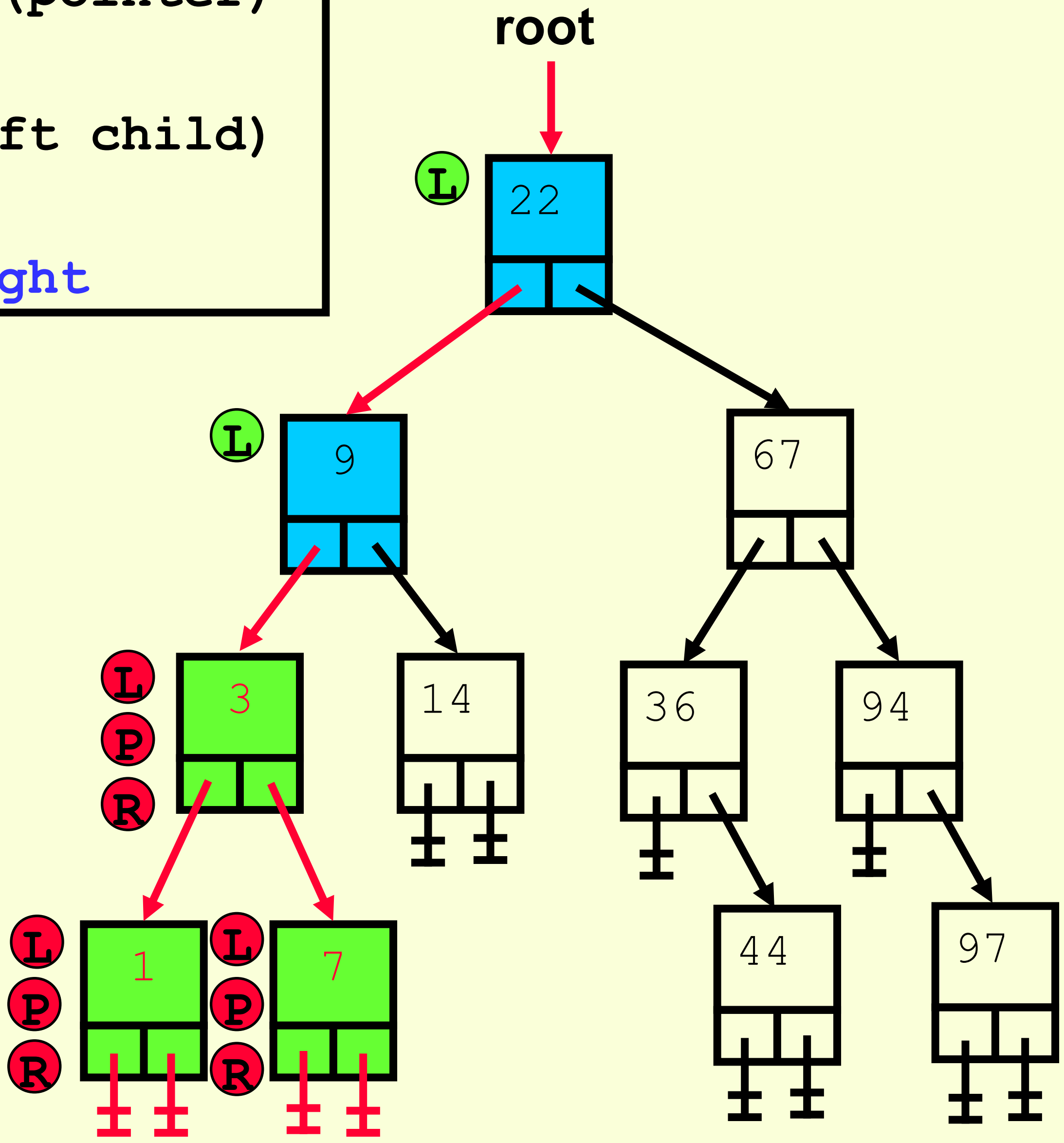
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right child)
```

```
Output: 1 3 7
```



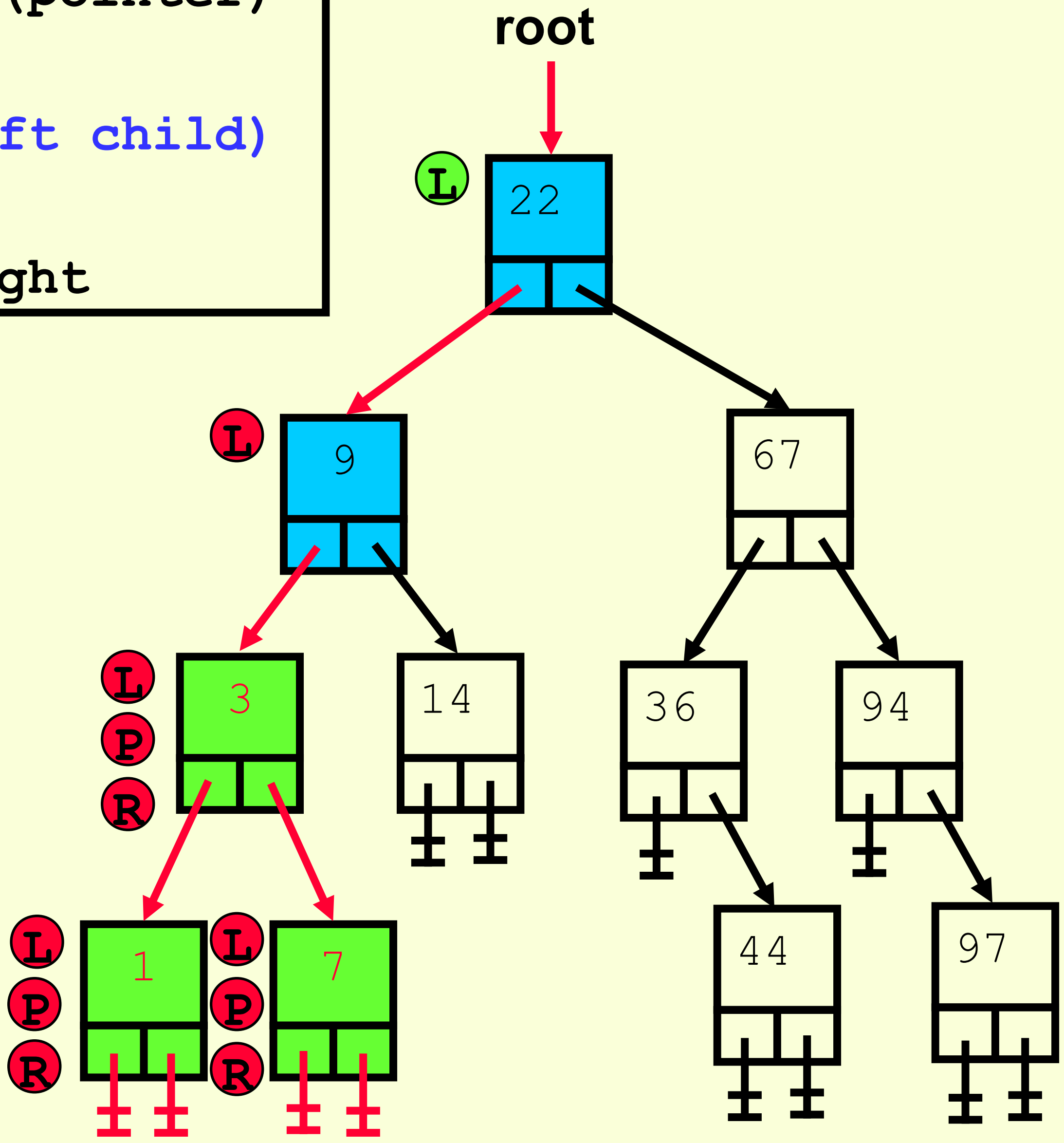
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7
```



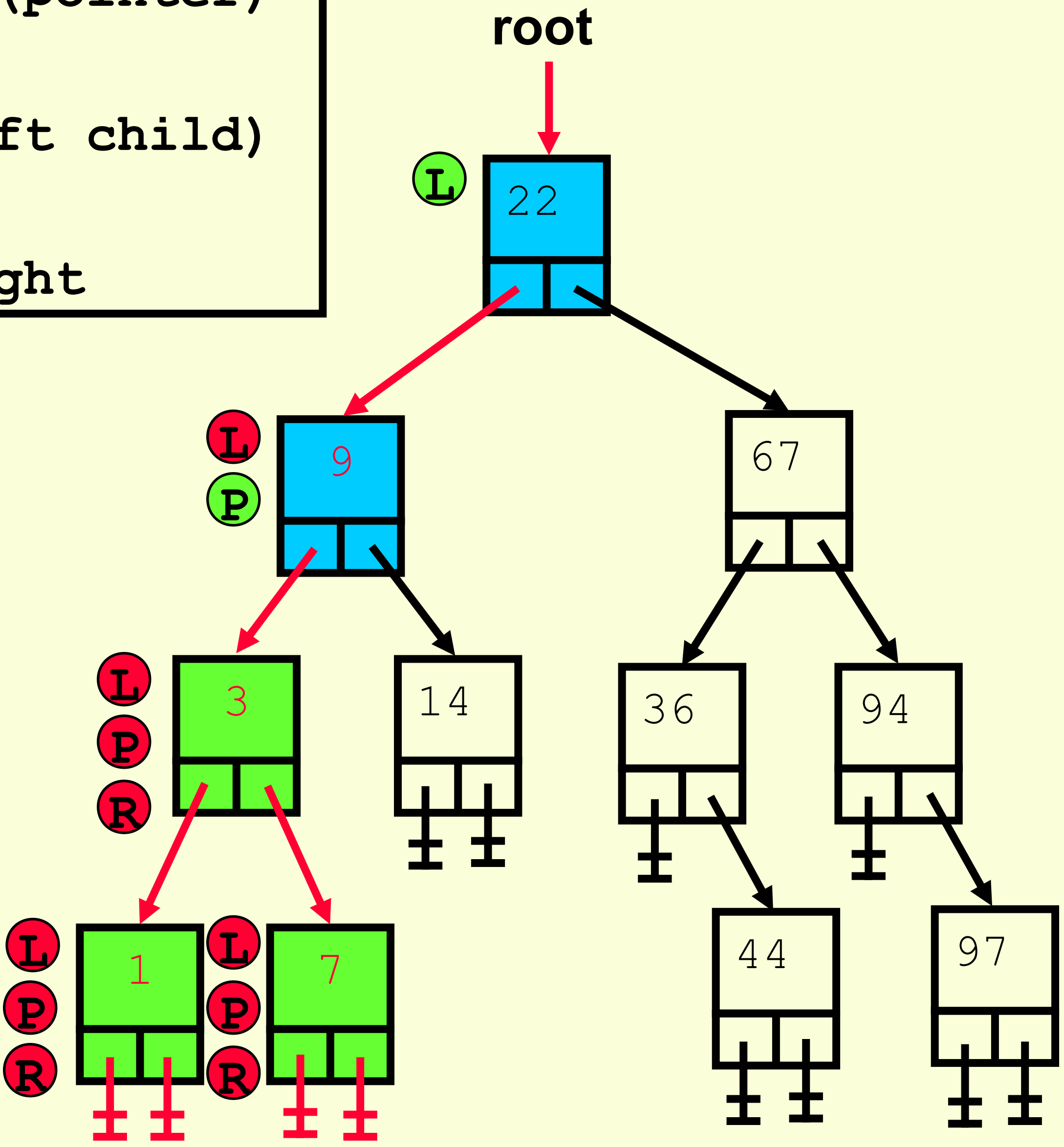
```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
 L InOrderPrint(left child)
 P print(data)
 R InOrderPrint(right
child)
```

```
Output: 1 3 7
```



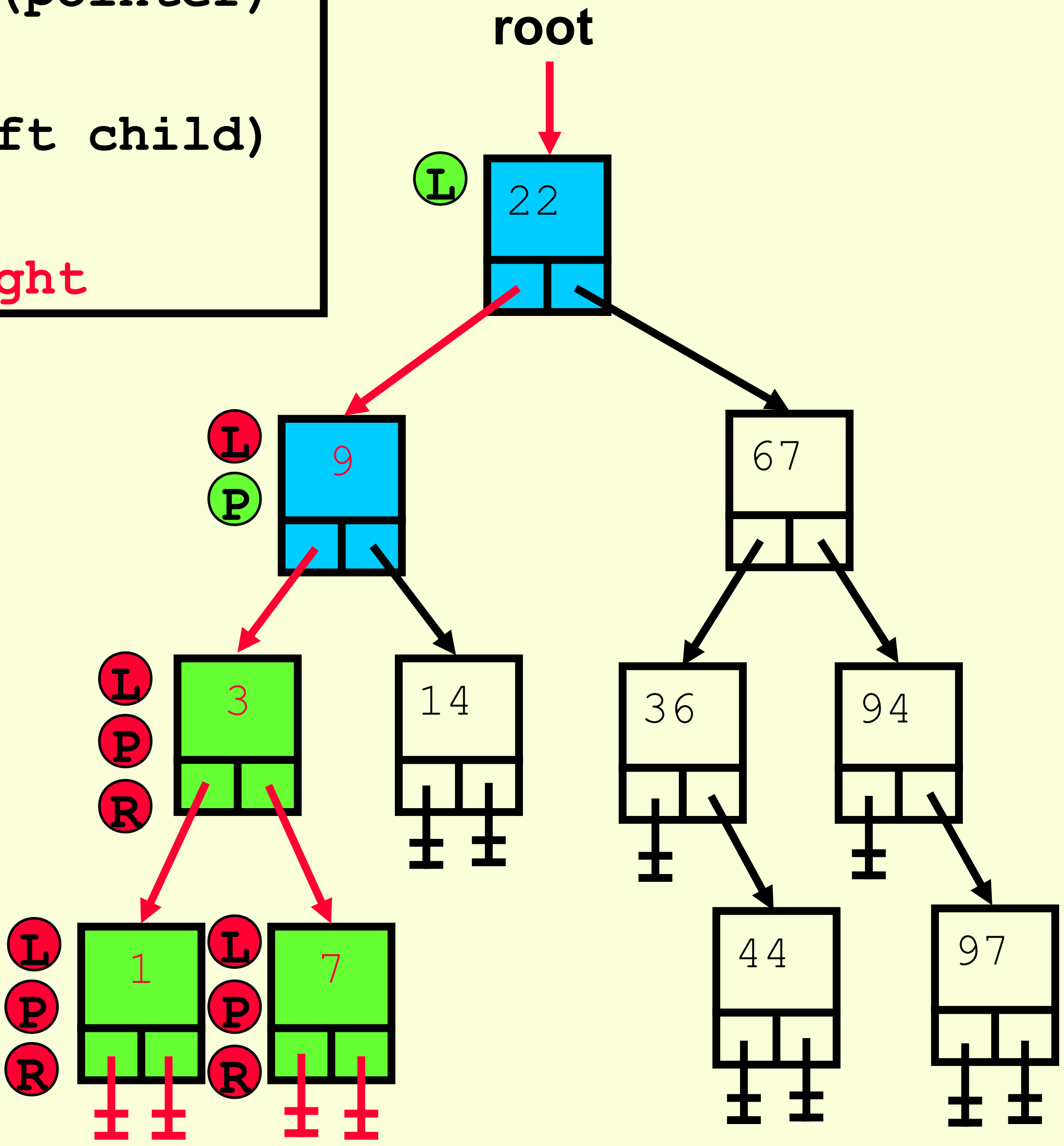
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9
```



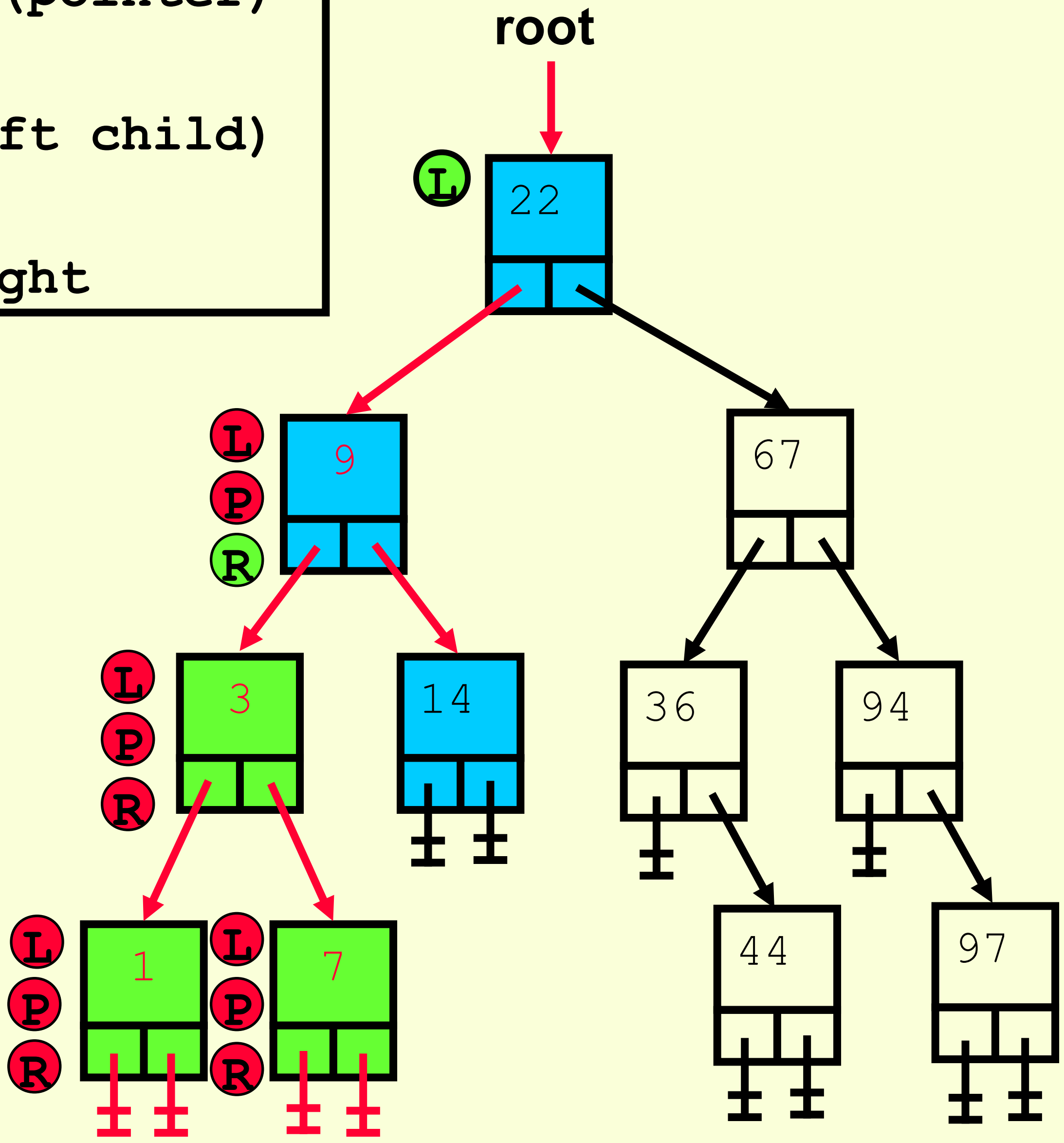
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9
```



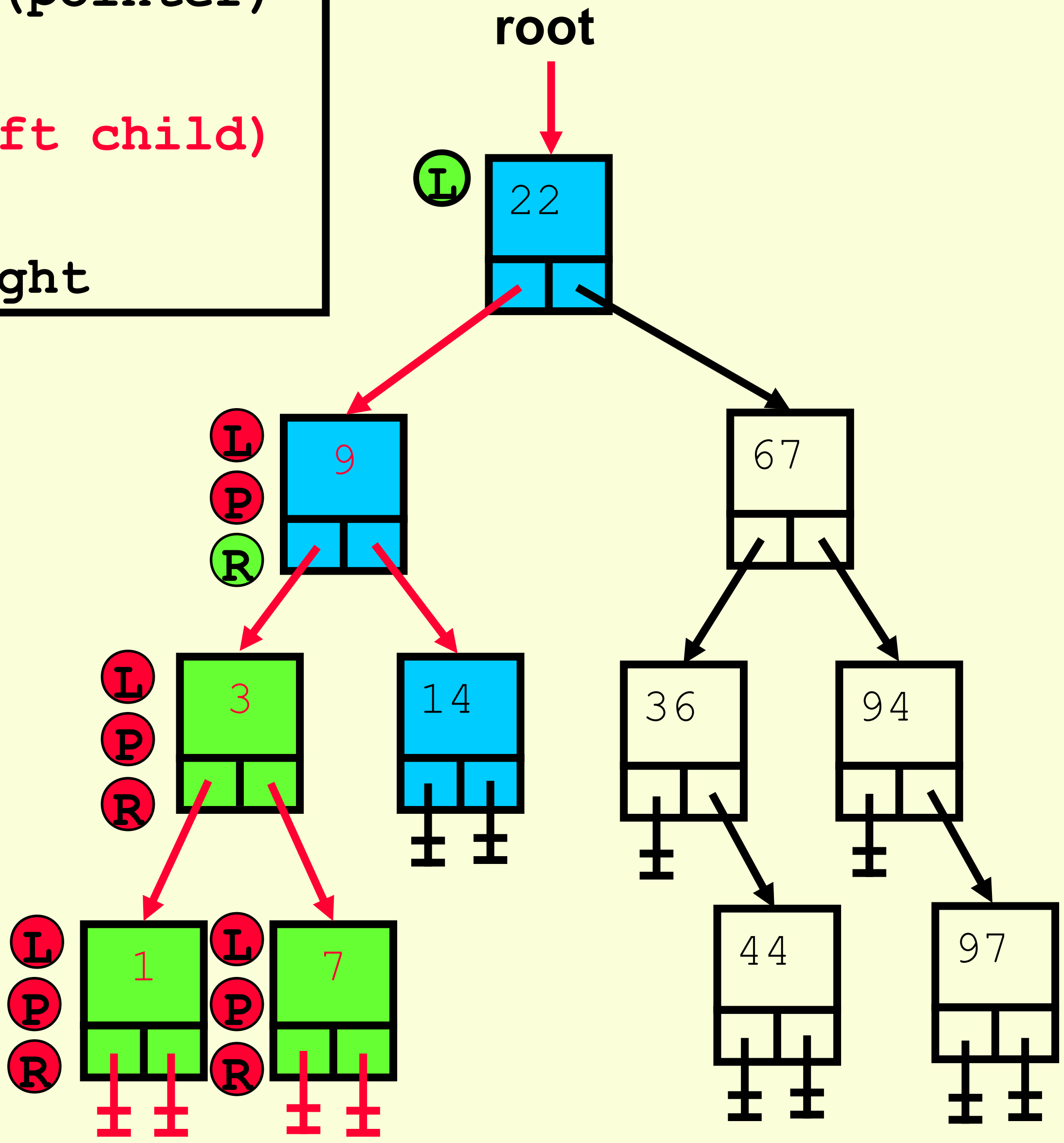
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9
```



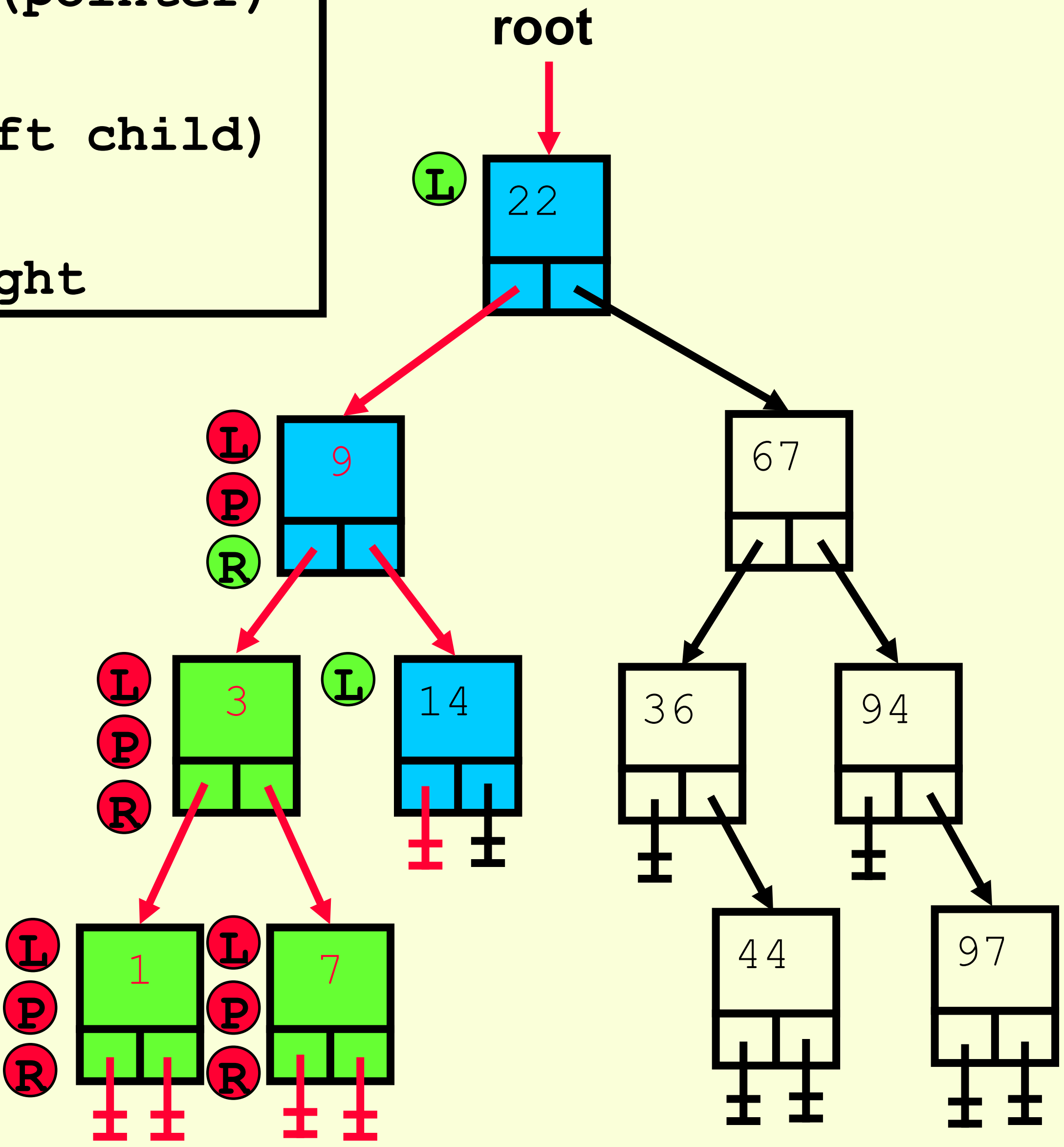
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right
child)
```

```
Output: 1 3 7 9
```



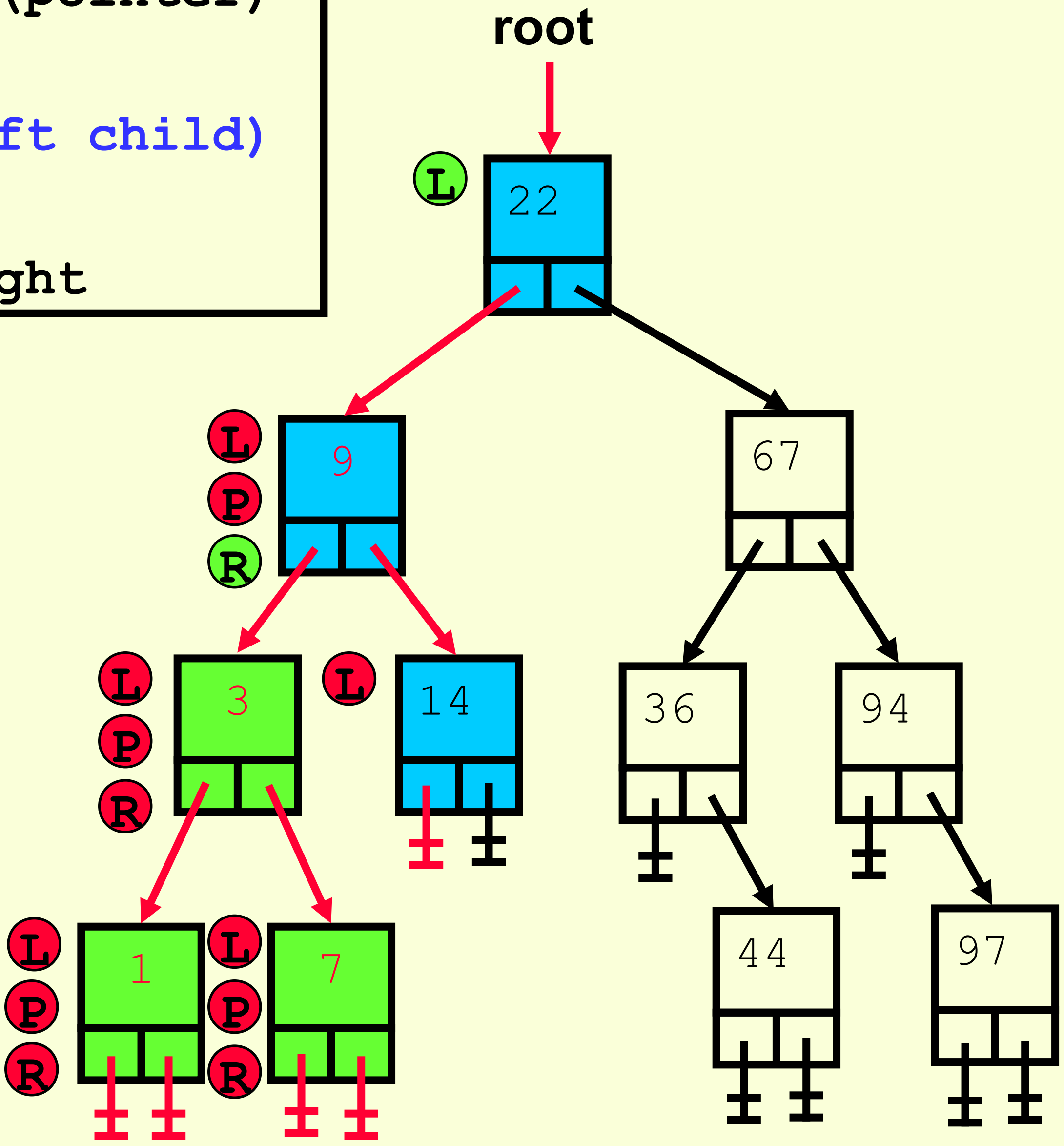
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9
```



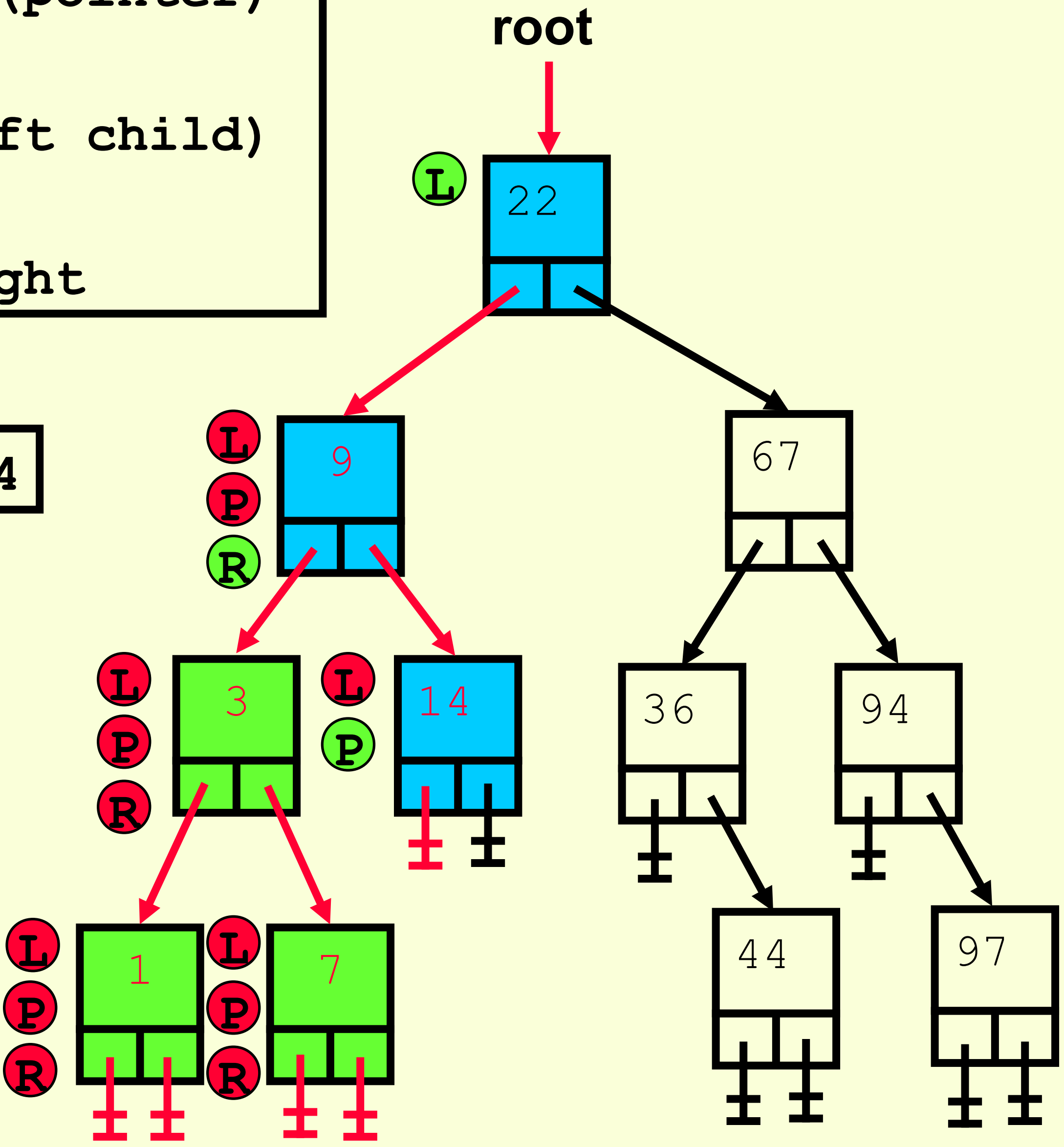
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right child)
```

```
Output: 1 3 7 9
```



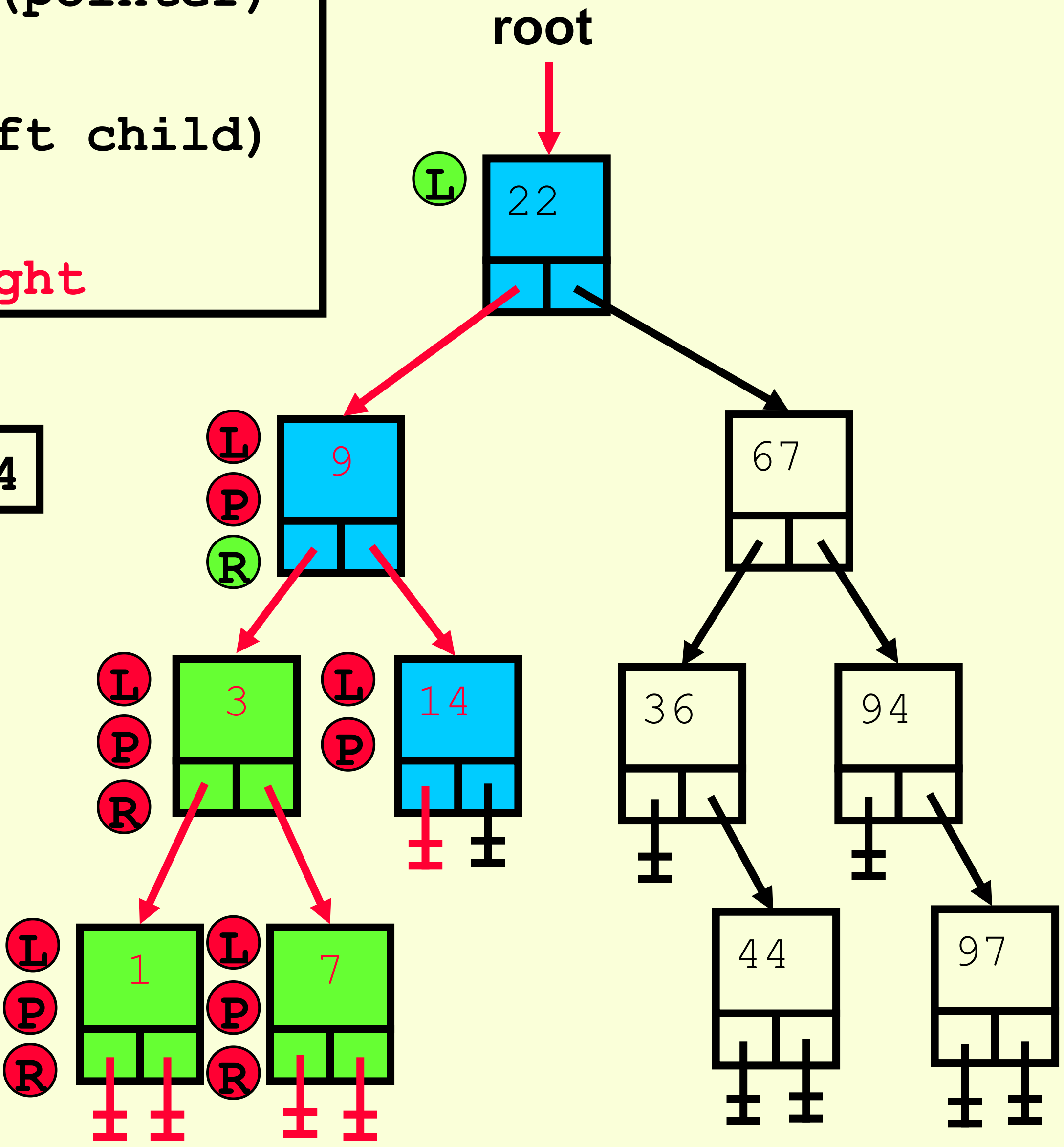
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14
```



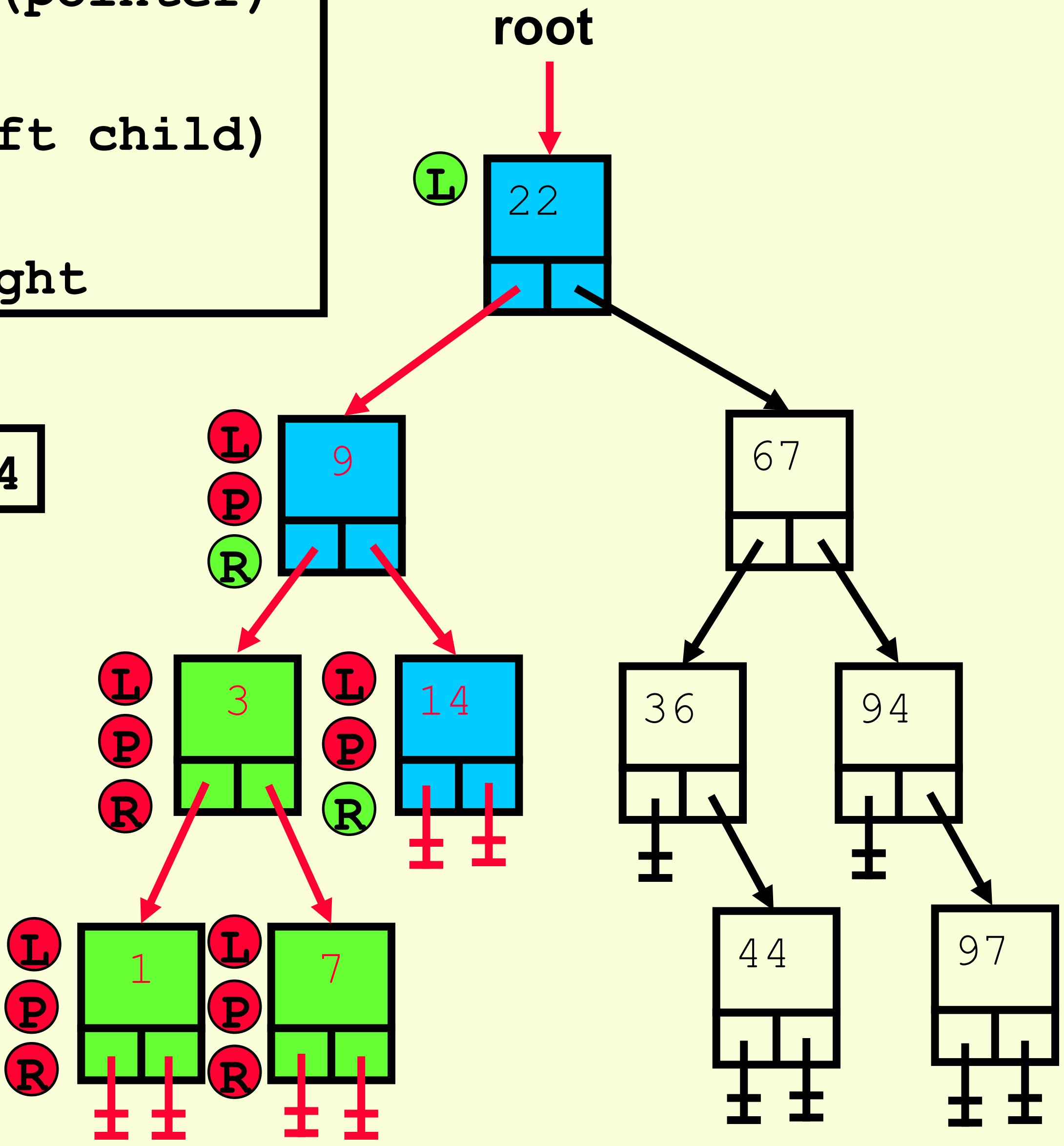
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right child)
```

```
Output: 1 3 7 9 14
```



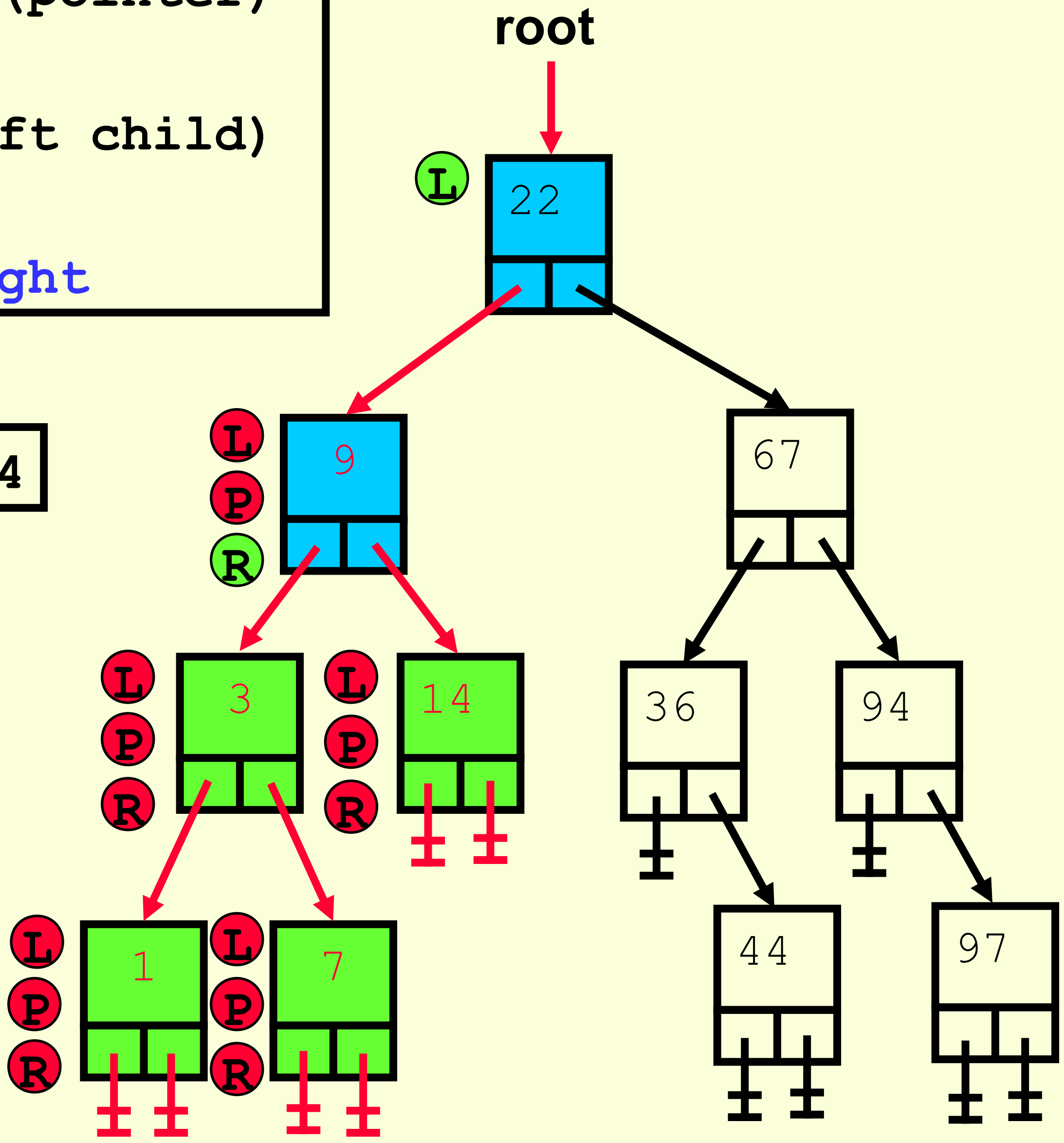
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14
```



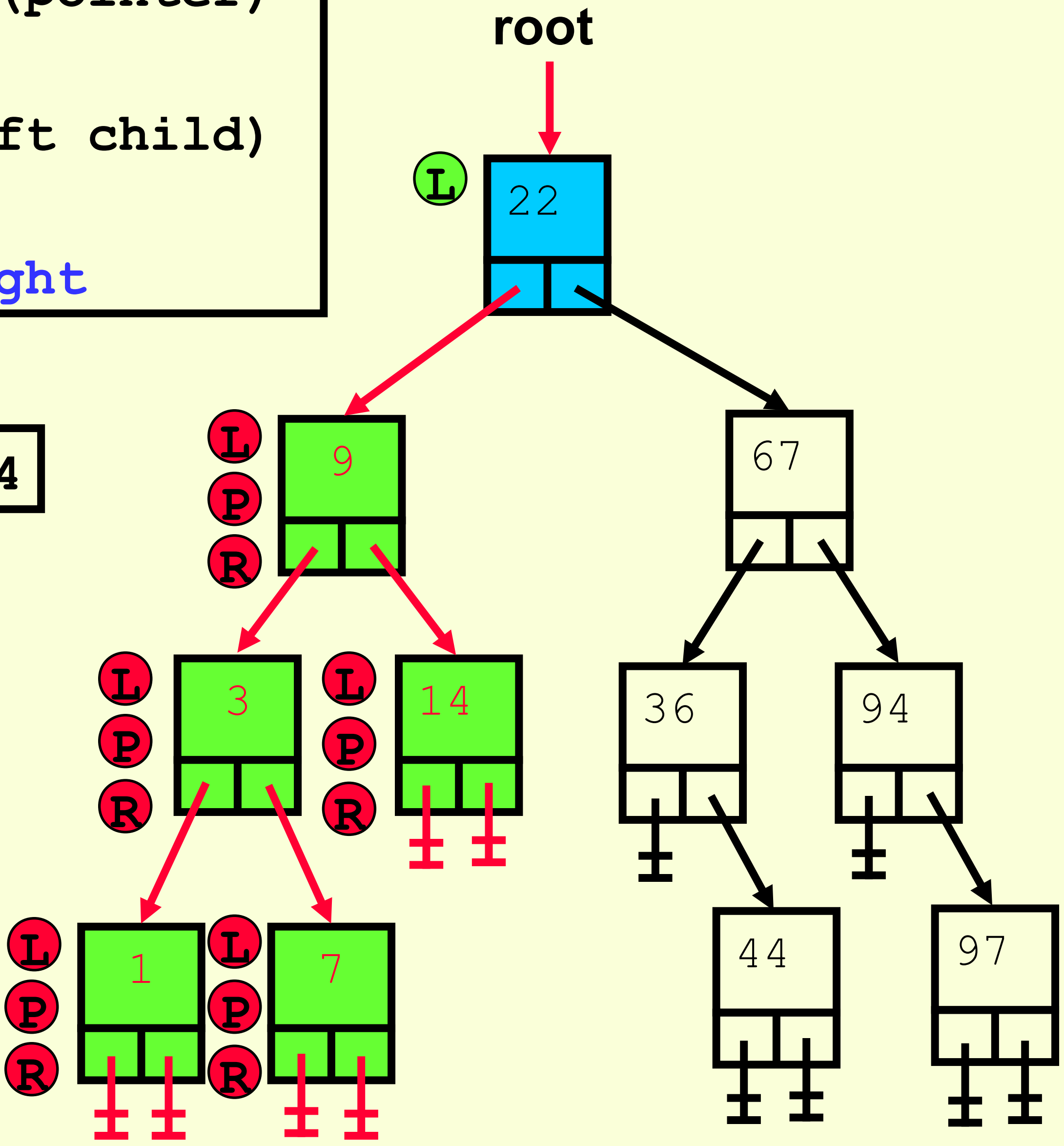
```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L   InOrderPrint(left child)
P   print(data)
R   InOrderPrint(right child)
```

```
Output: 1 3 7 9 14
```



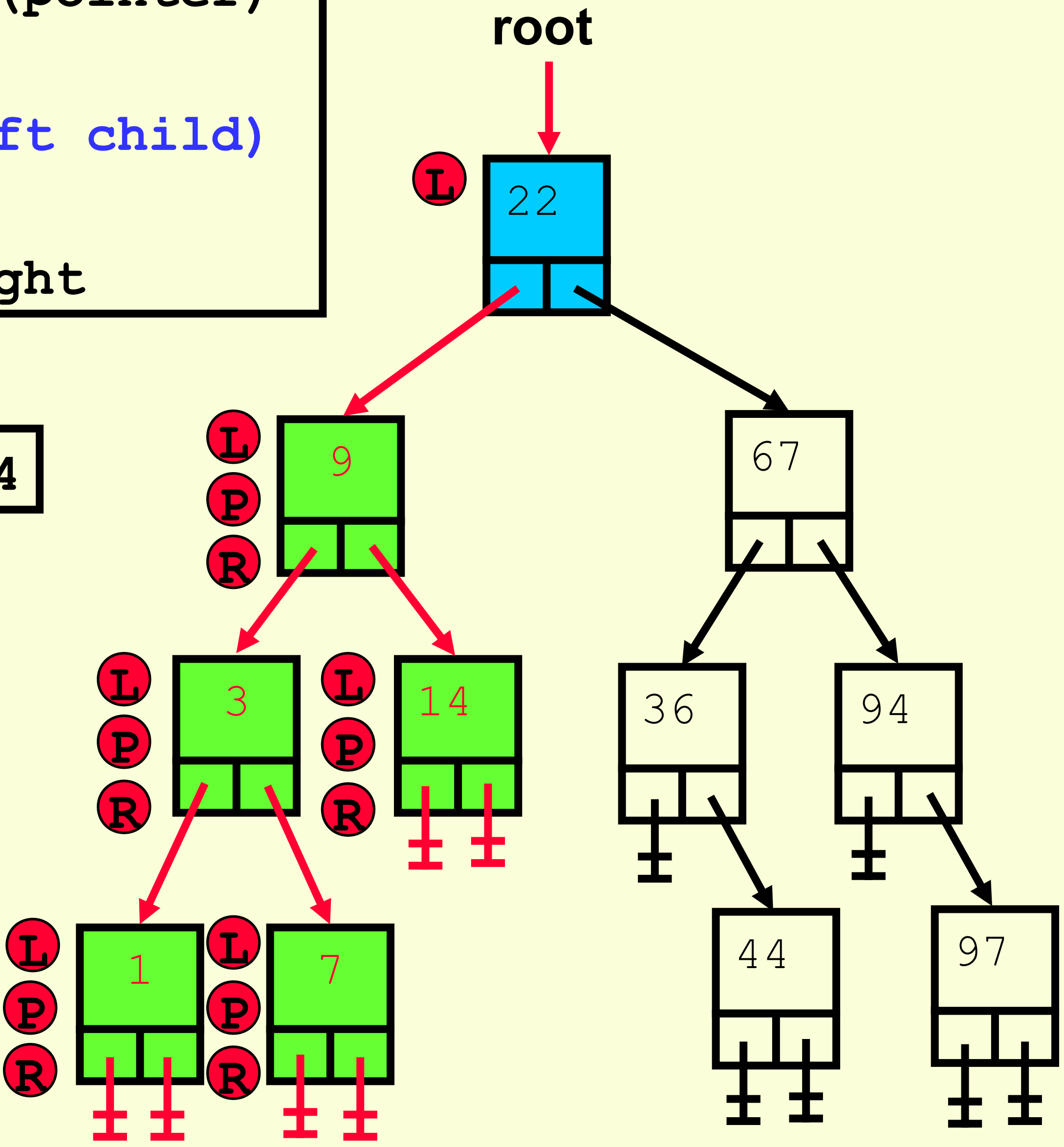
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right child)
```

```
Output: 1 3 7 9 14
```



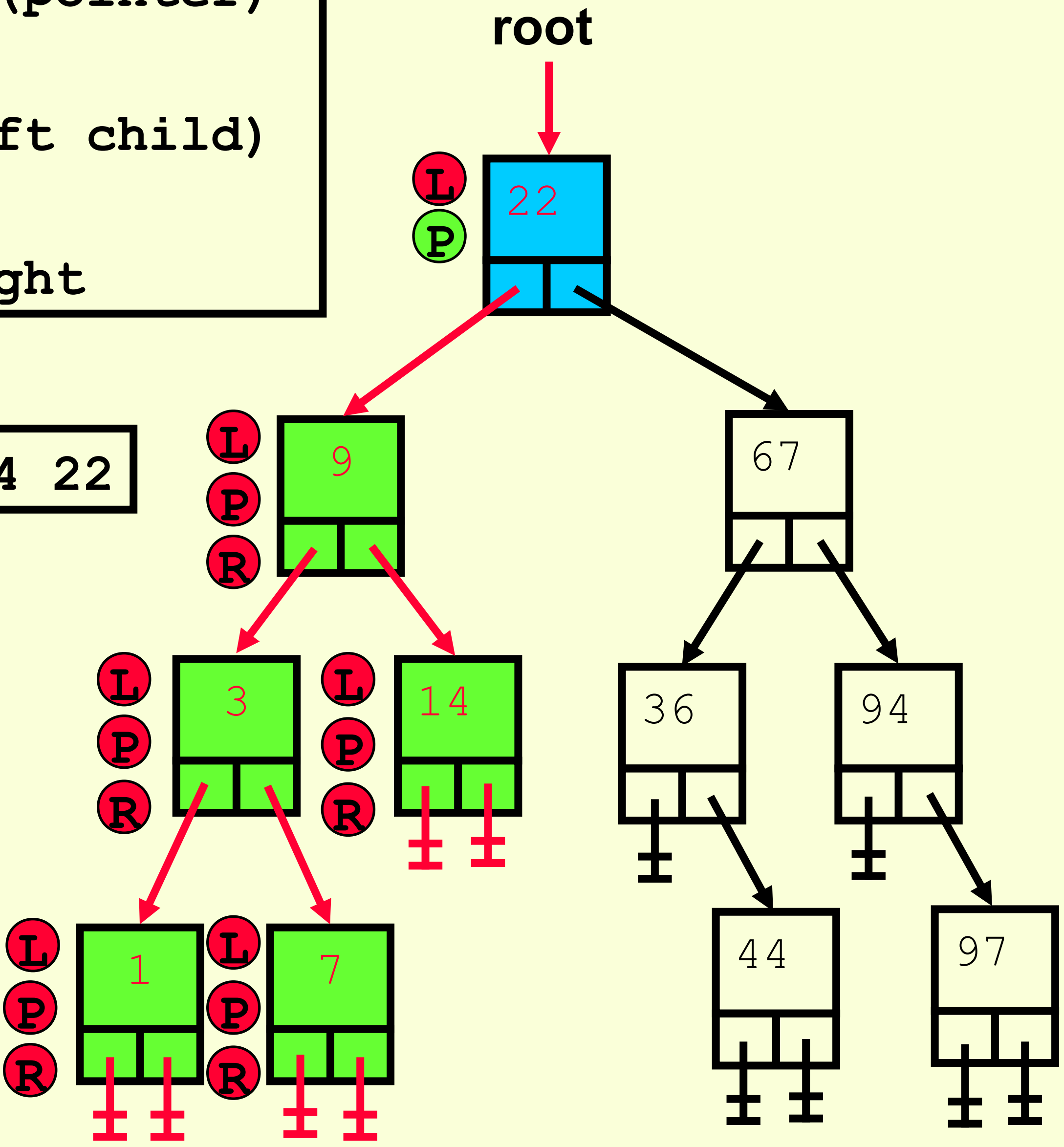
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14
```



```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
```



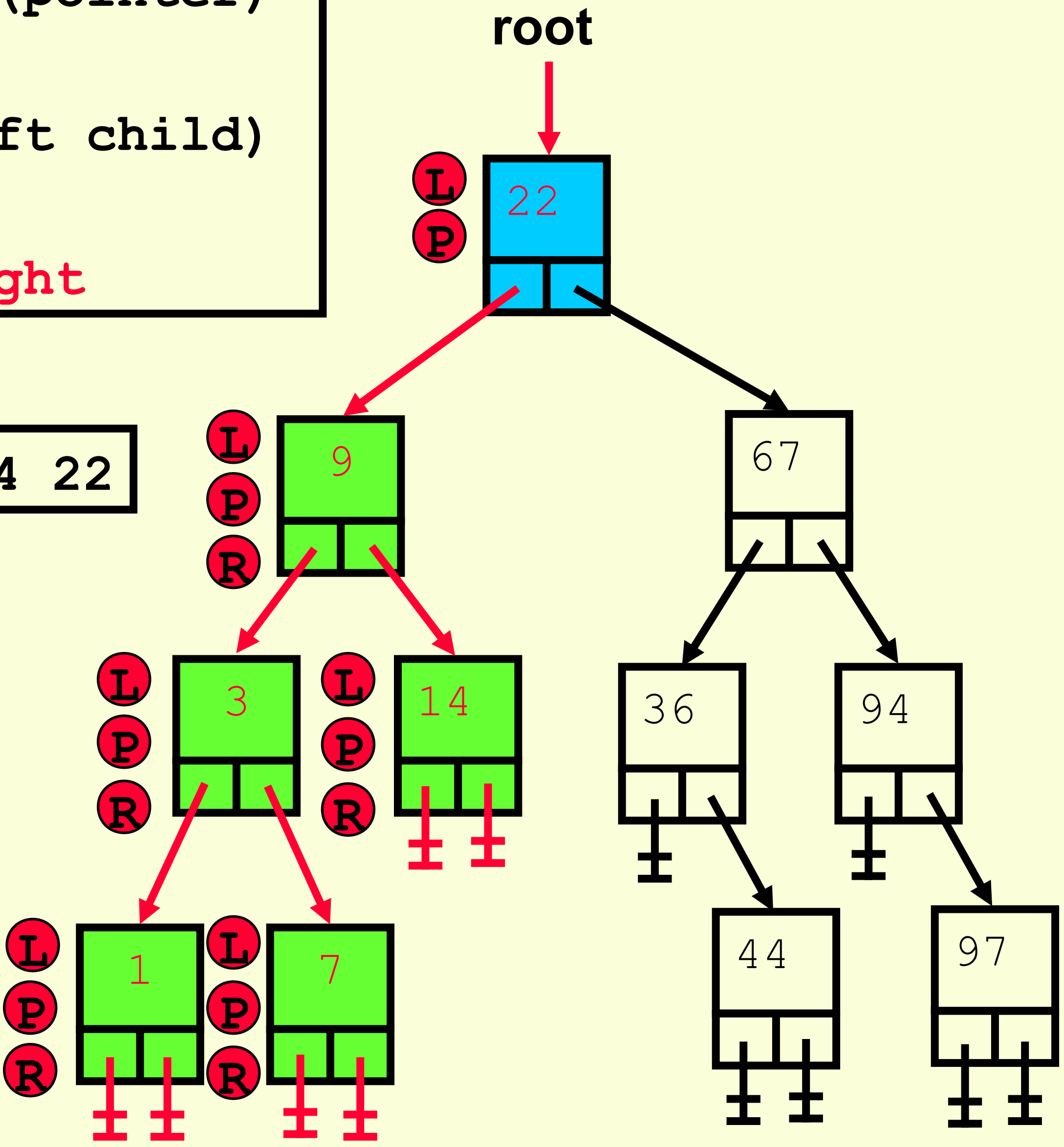
Continue?

Yes!

Enough Already!

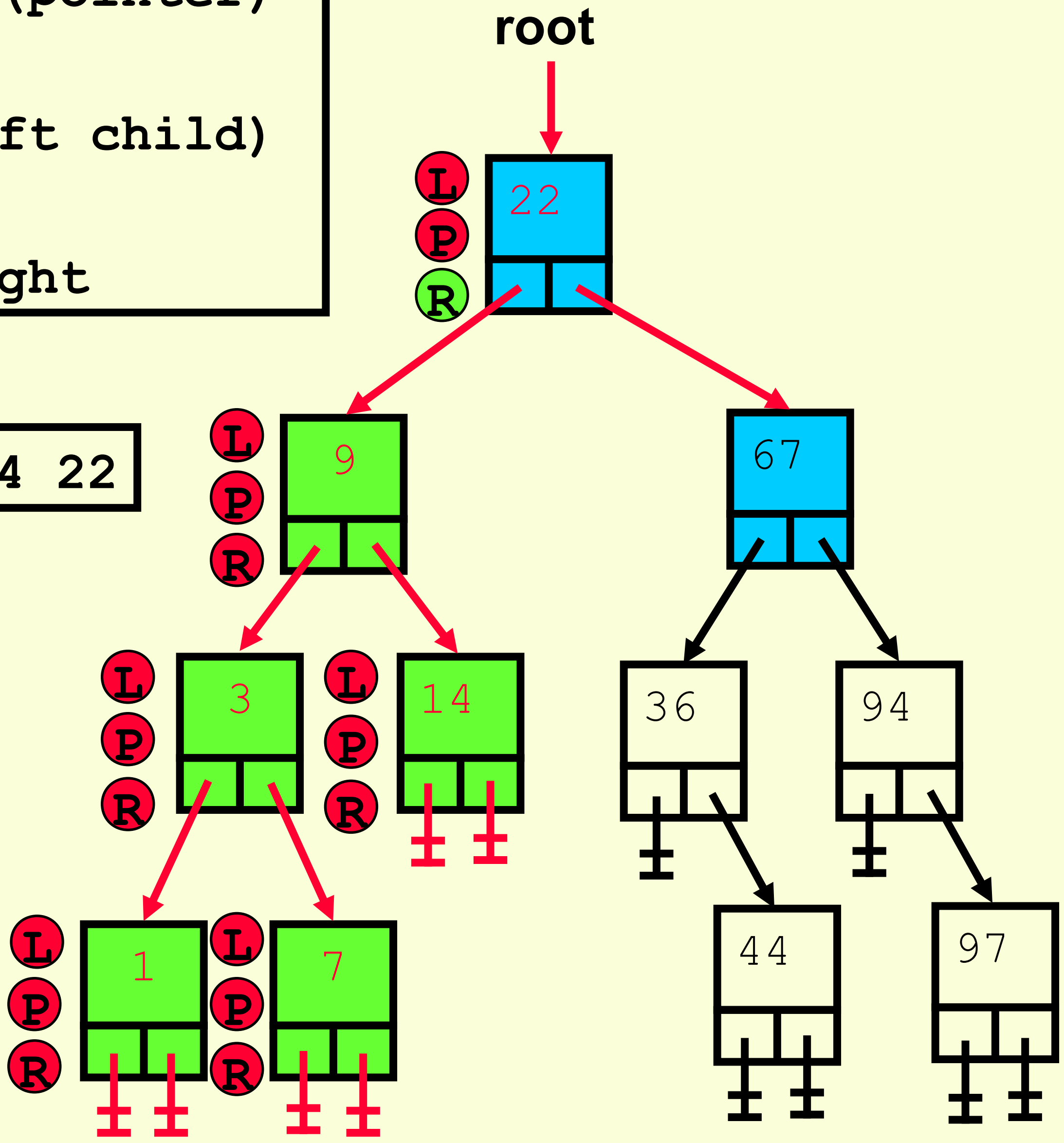
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
```



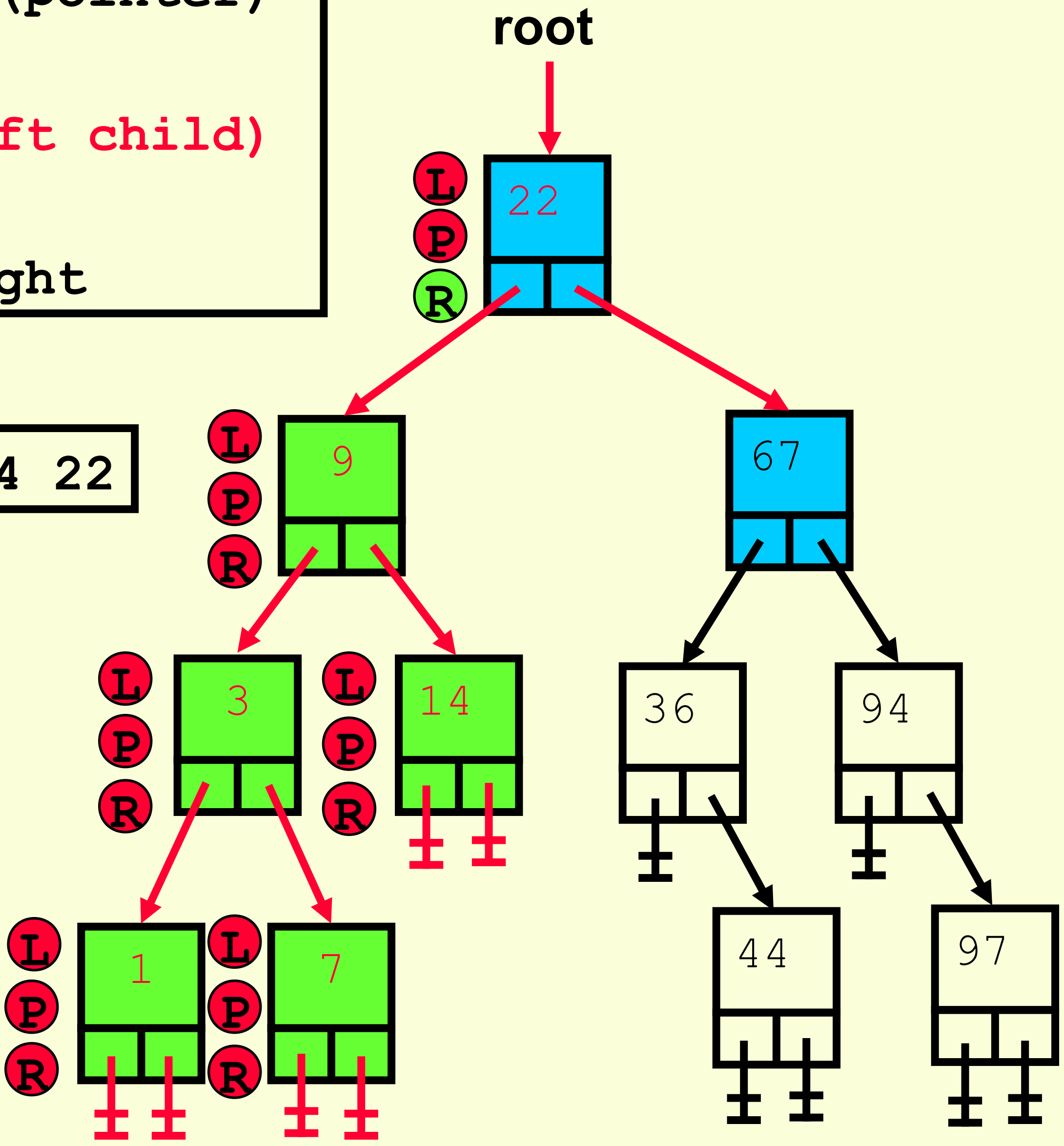
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
```



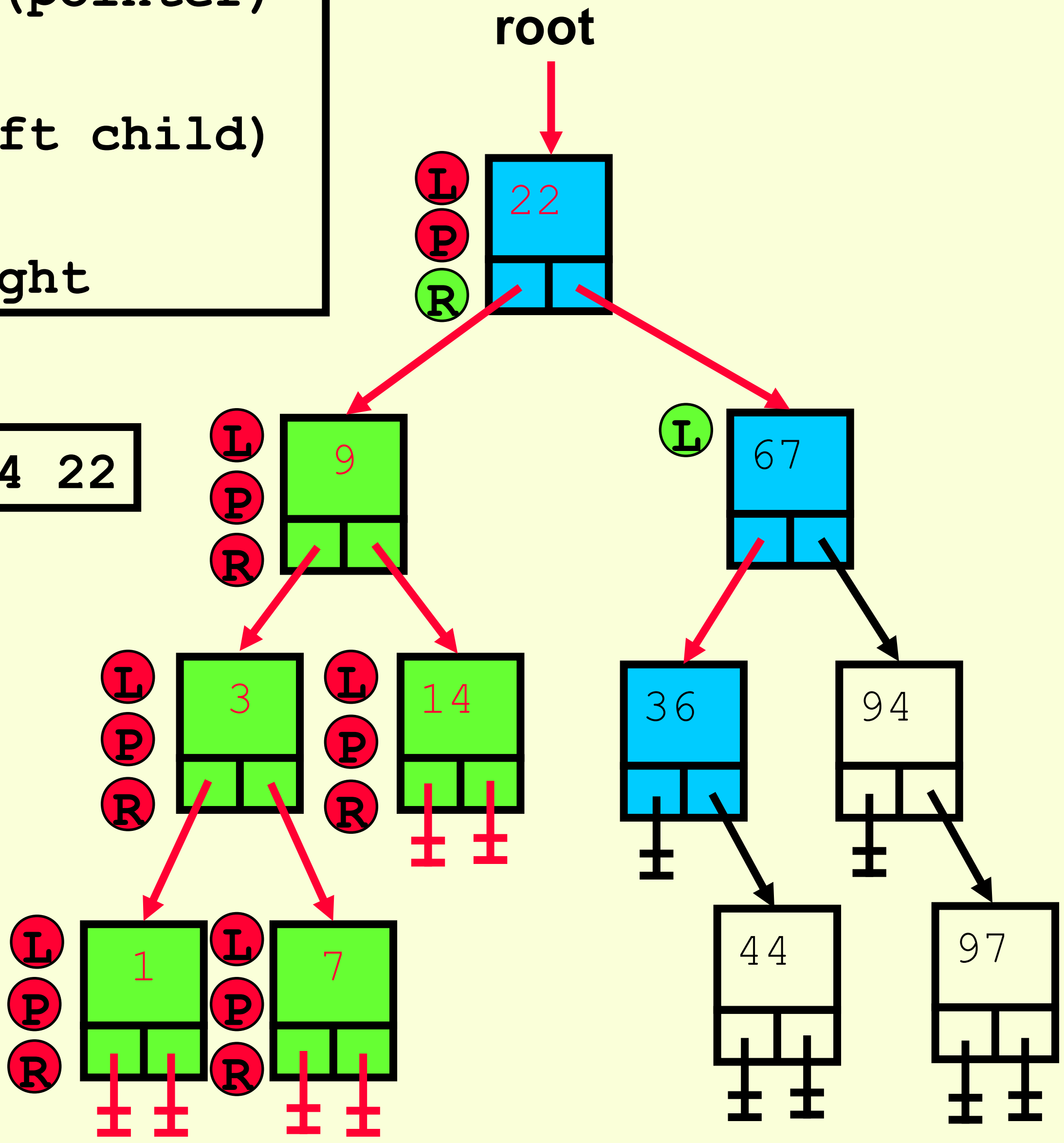
```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L   InOrderPrint(left child)
P   print(data)
R   InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
```



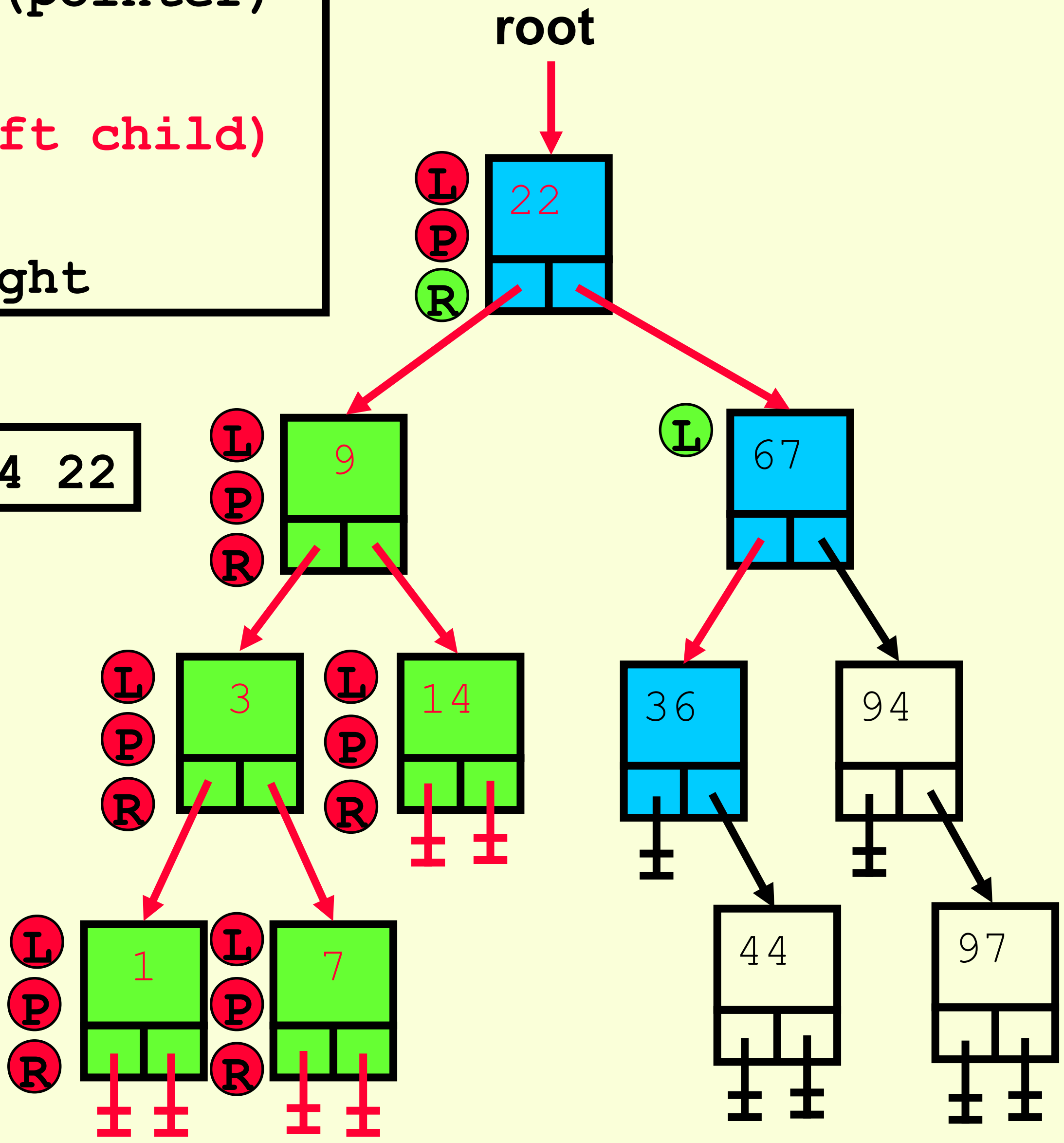
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
```



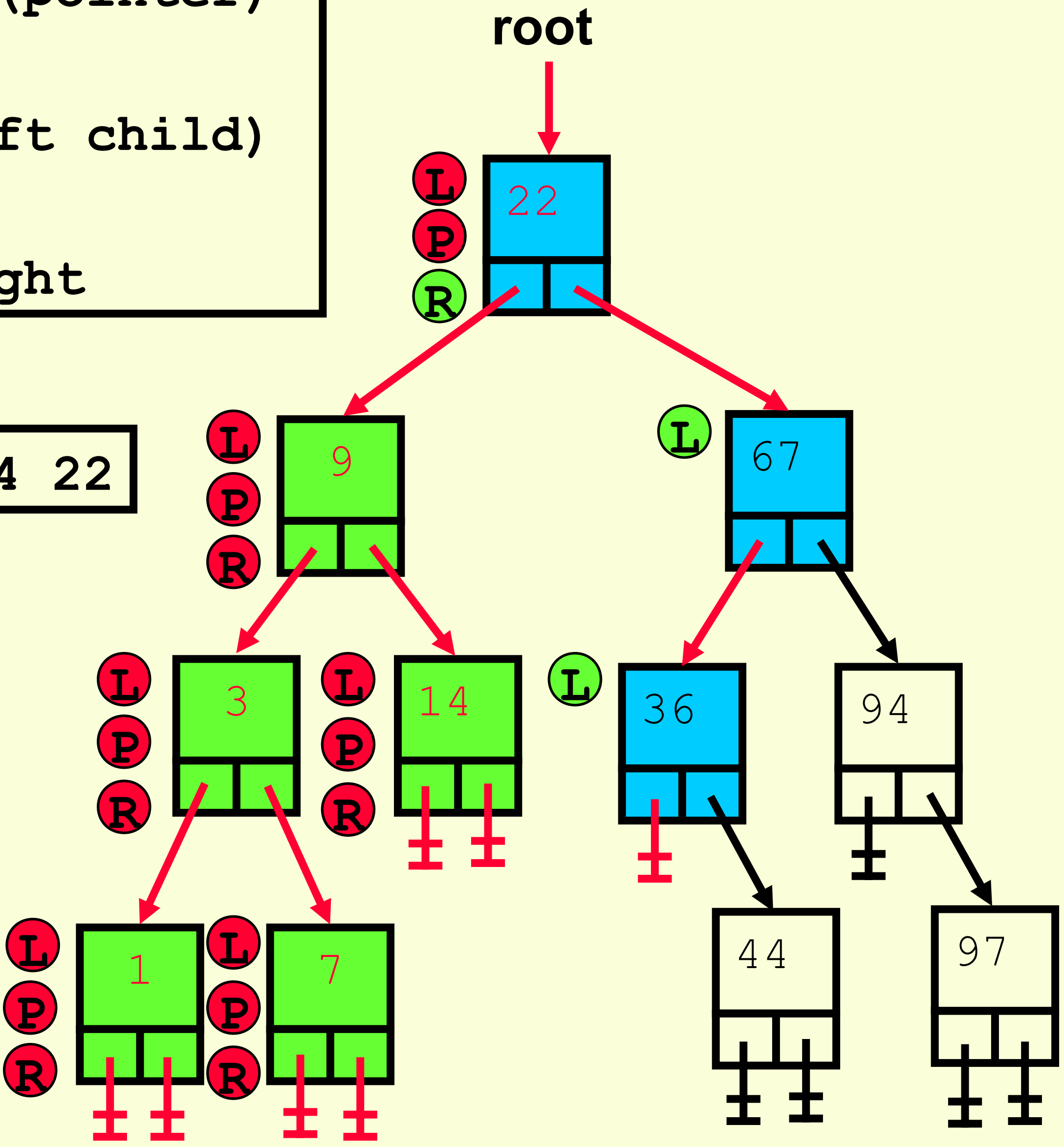
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
```



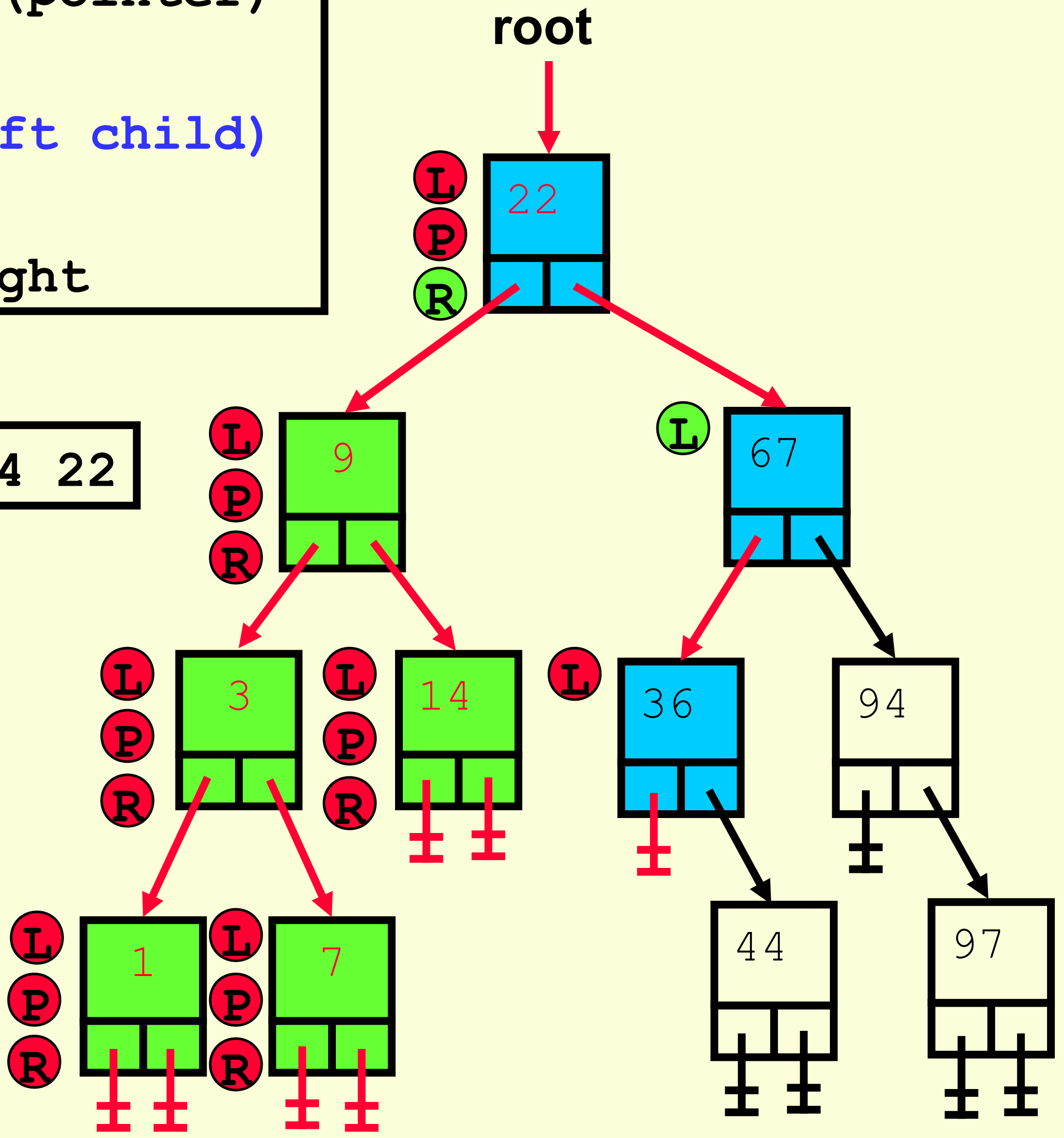
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
```



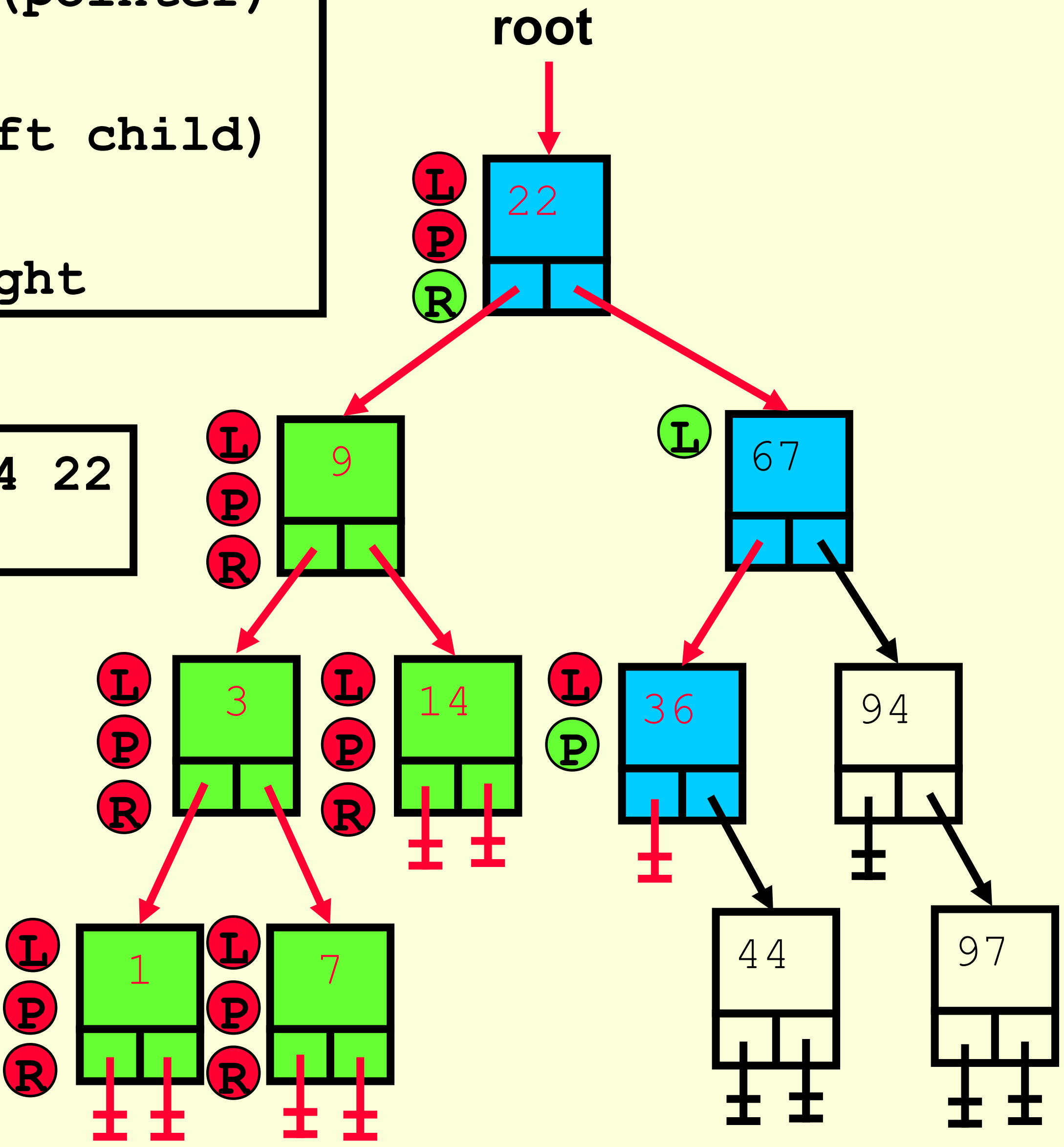
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
```



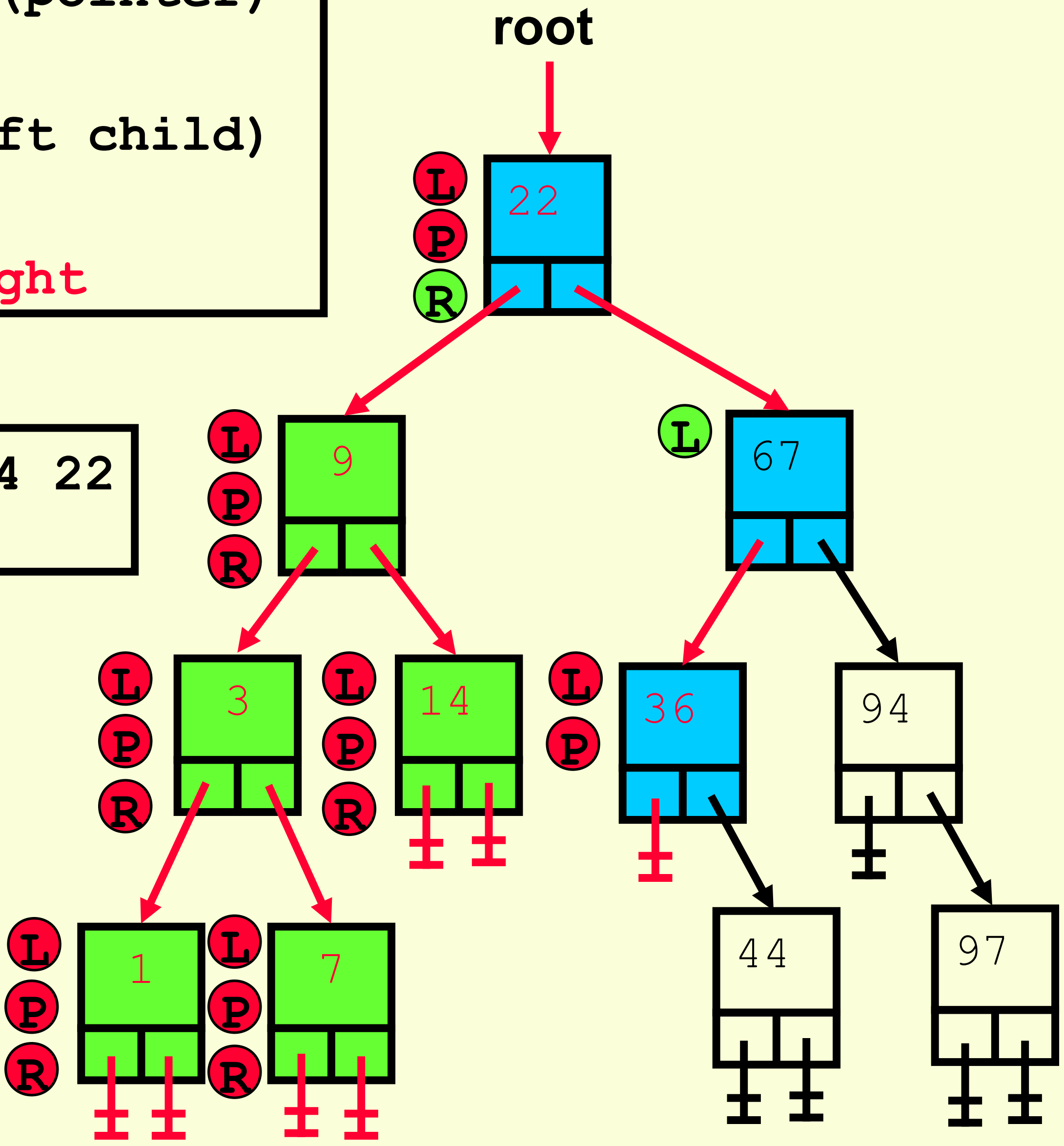
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36
```



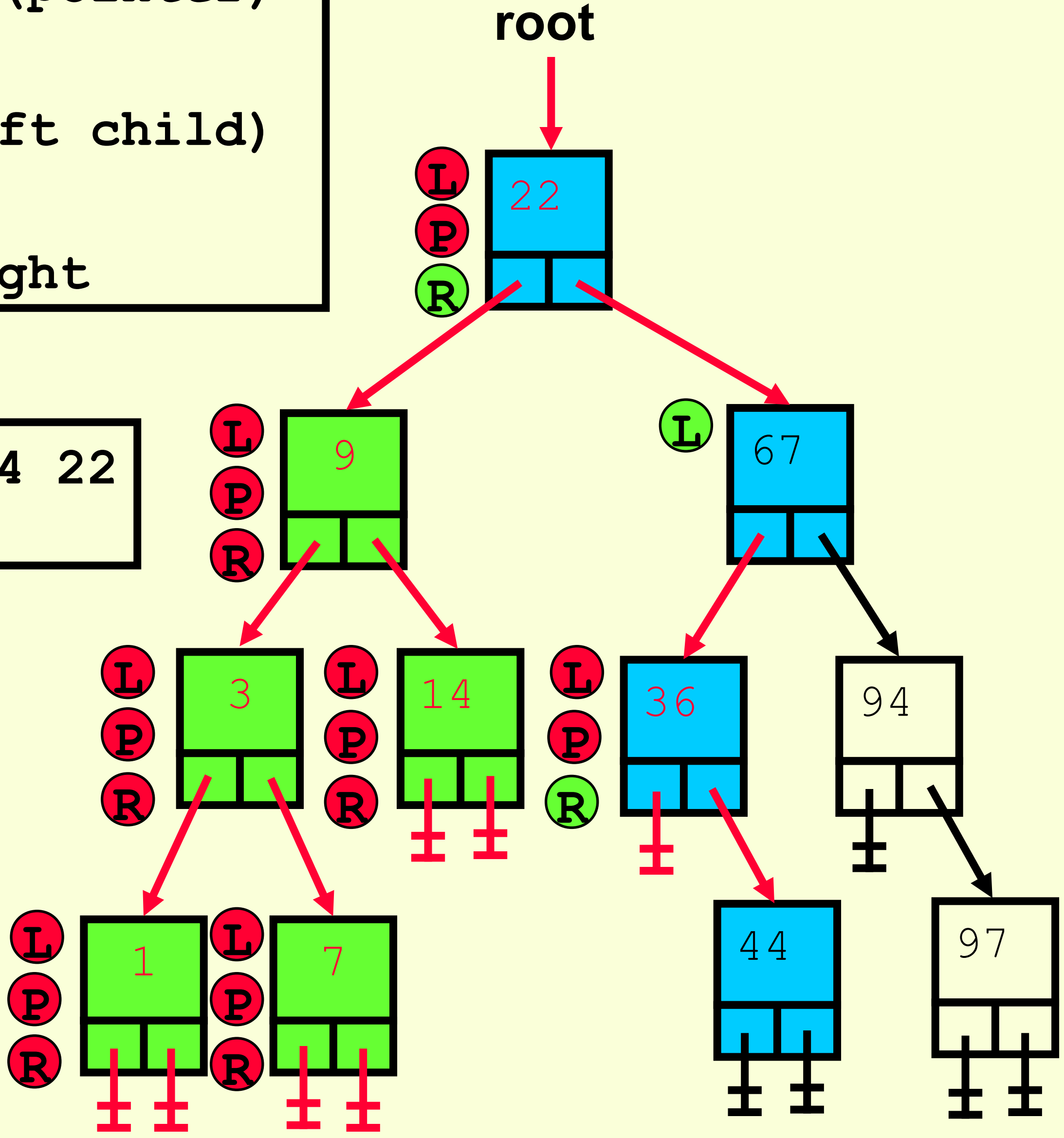
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36
```



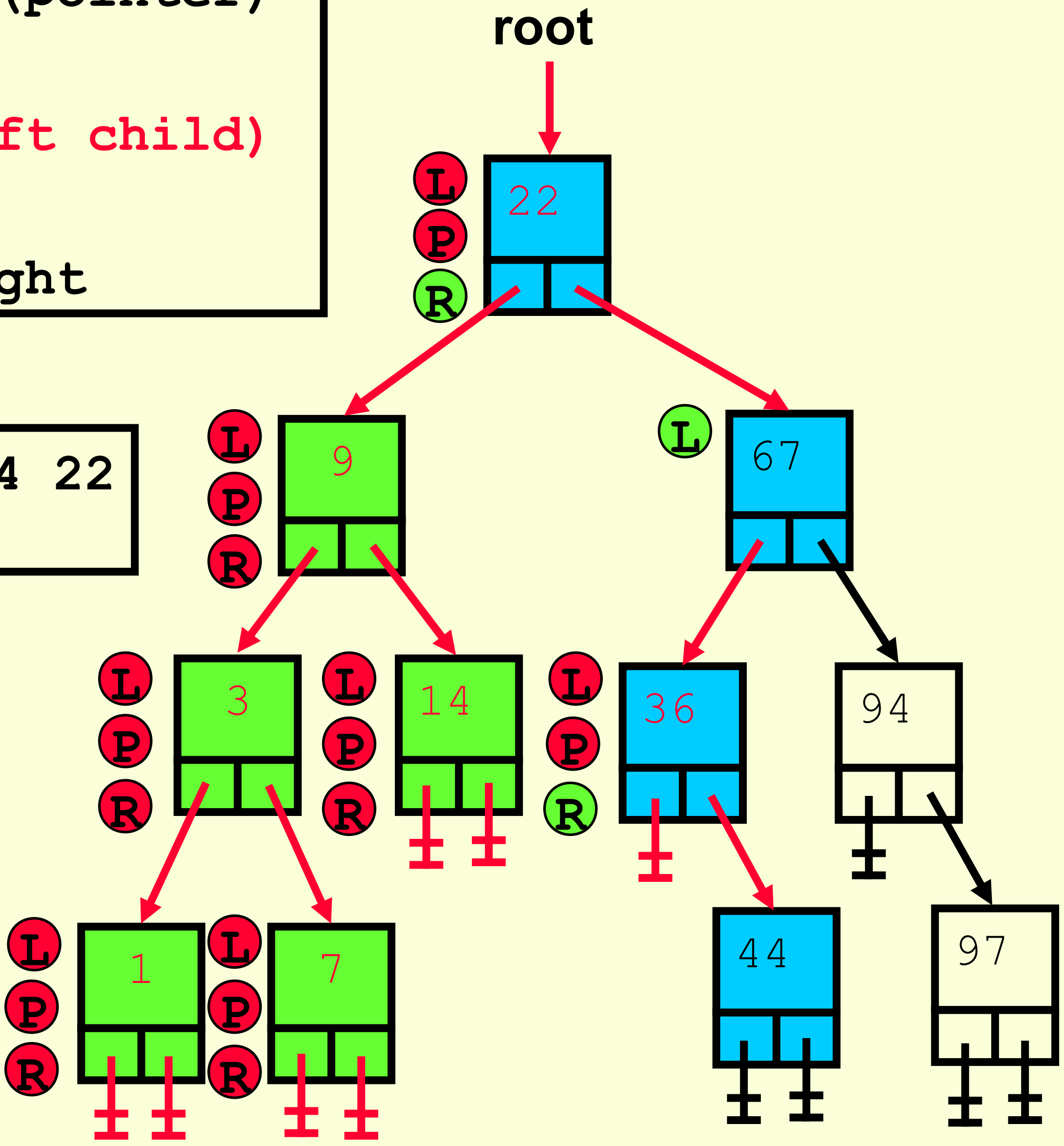
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36
```



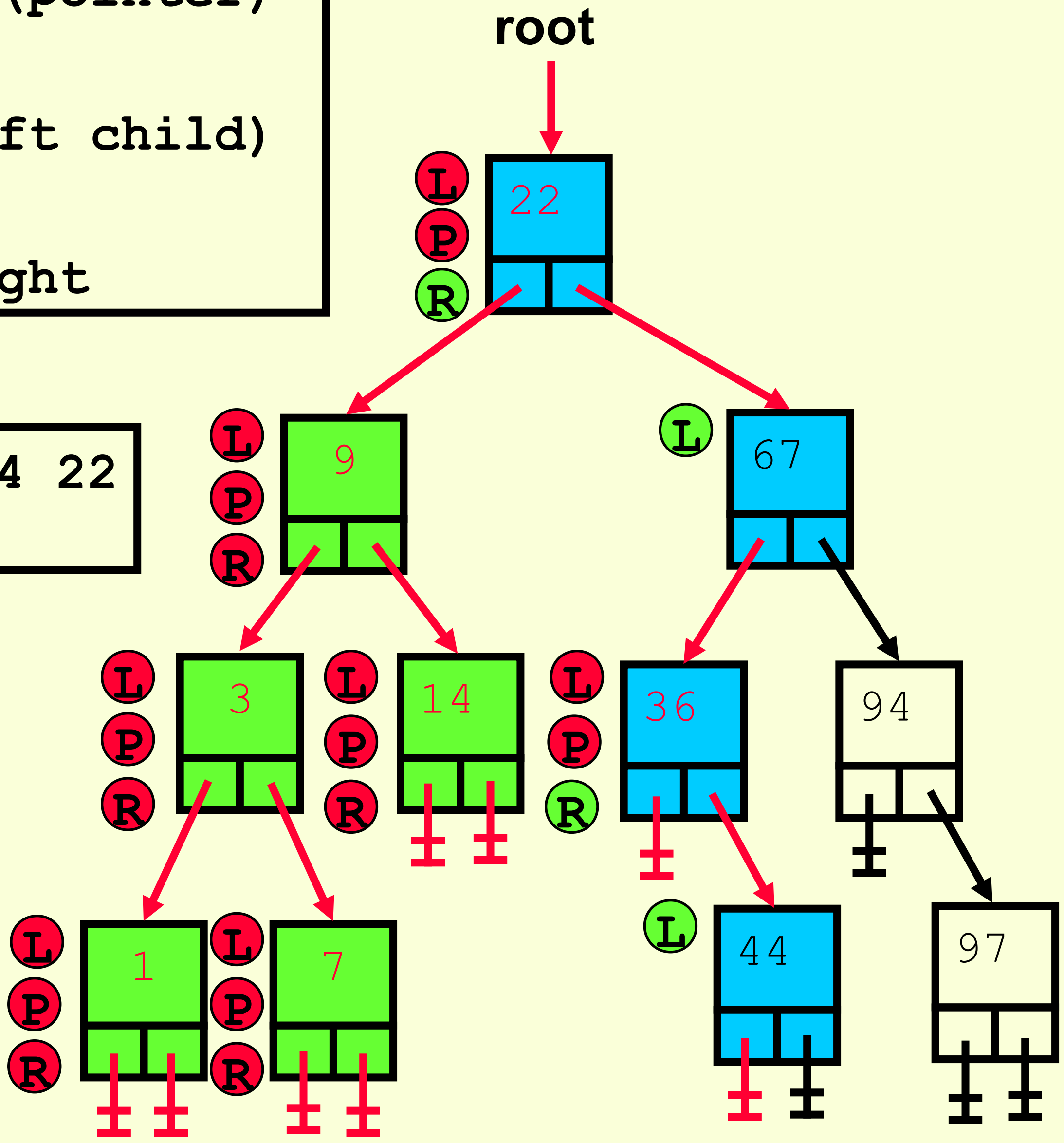
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36
```



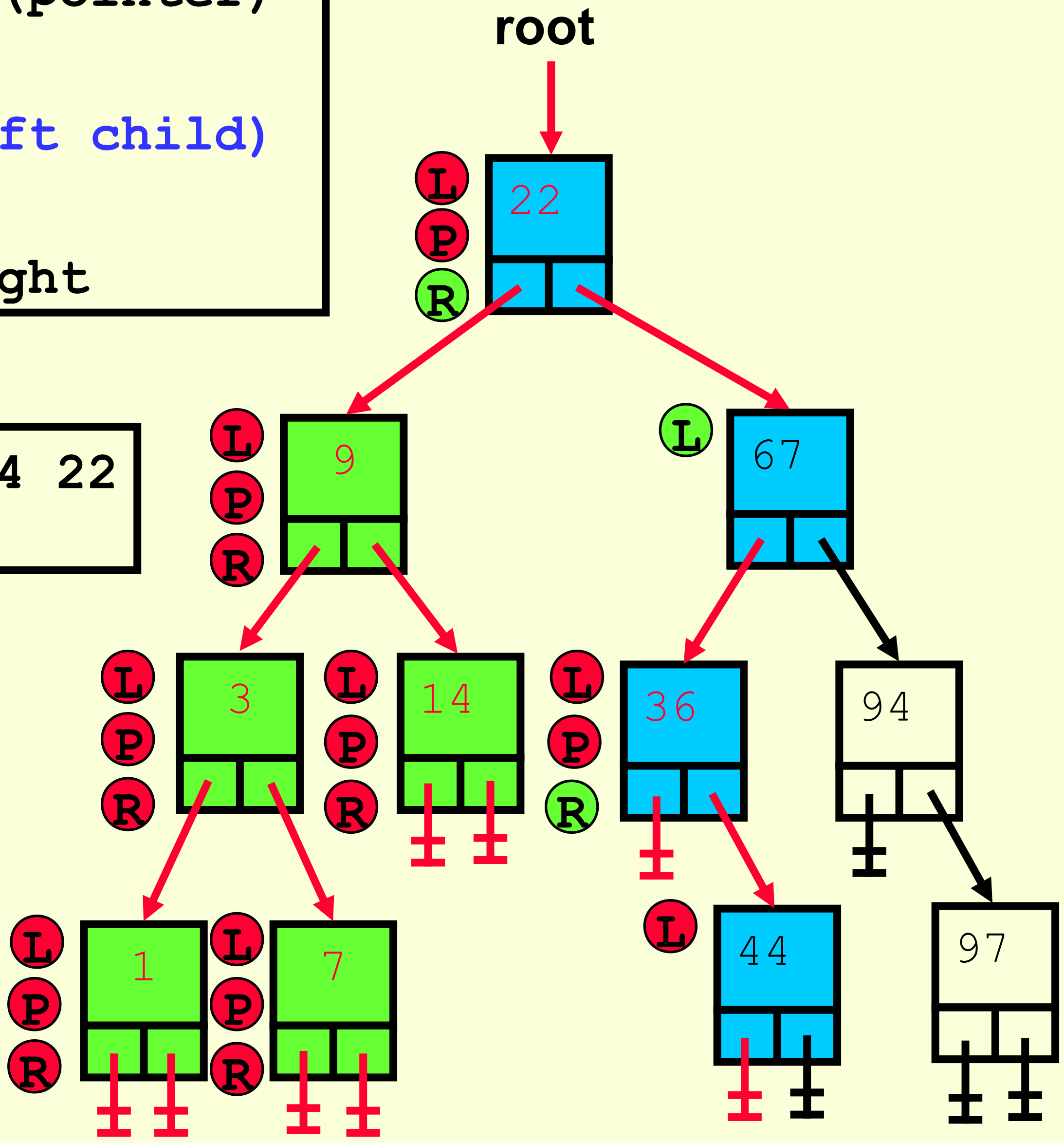
```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36
```



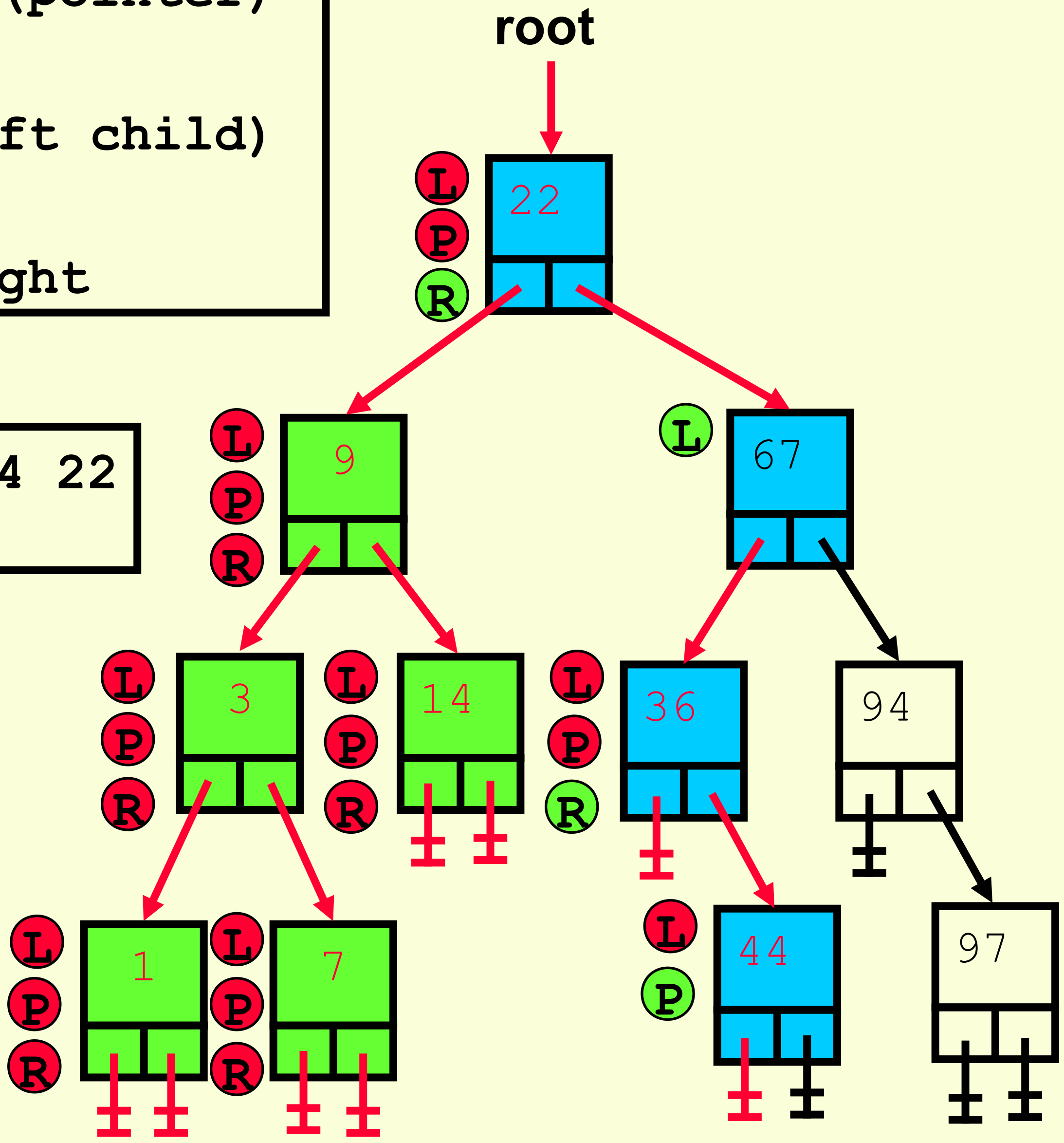
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36
```



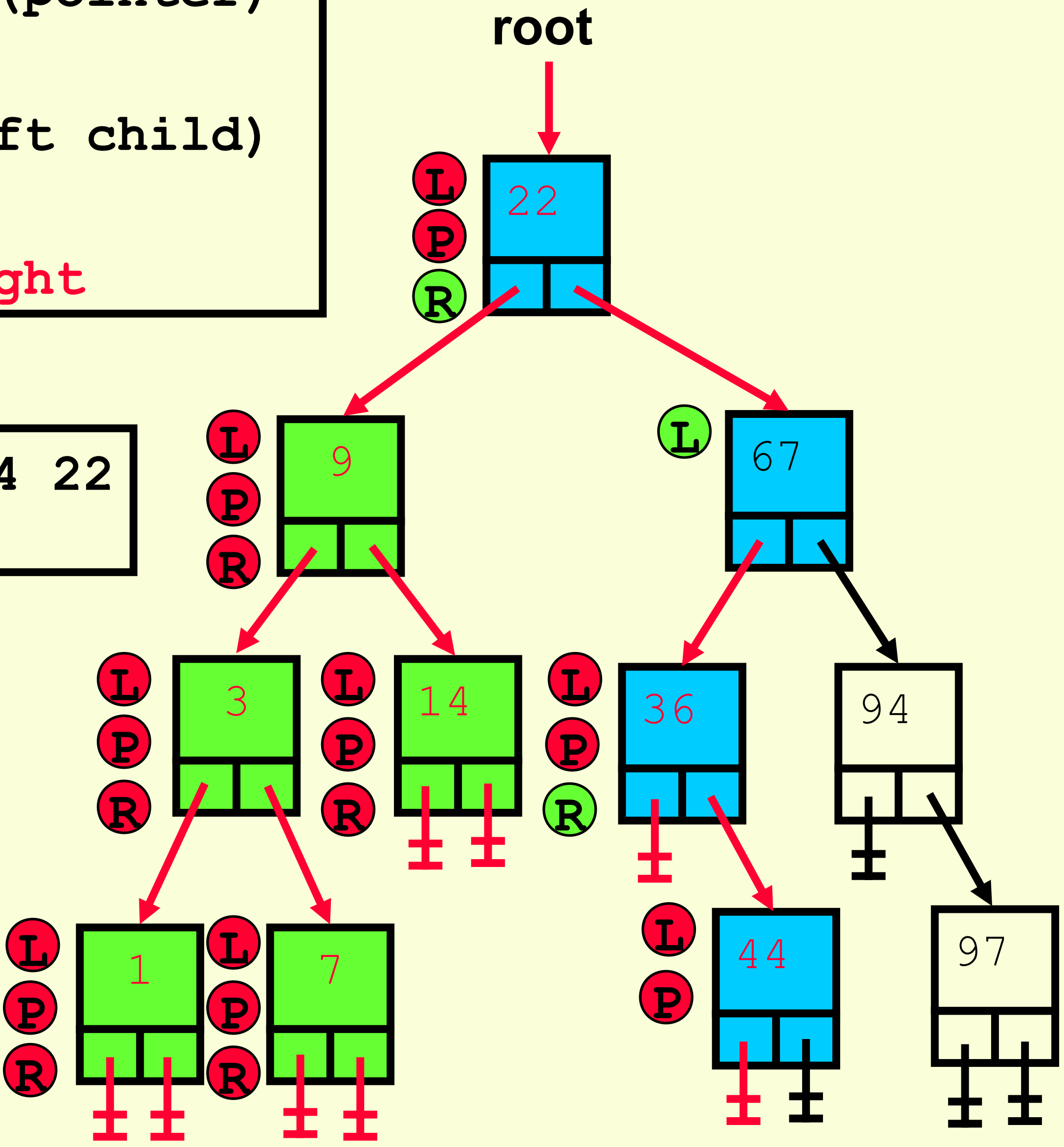
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

```
Output: 1 3 7 9 14 22
        36 44
```



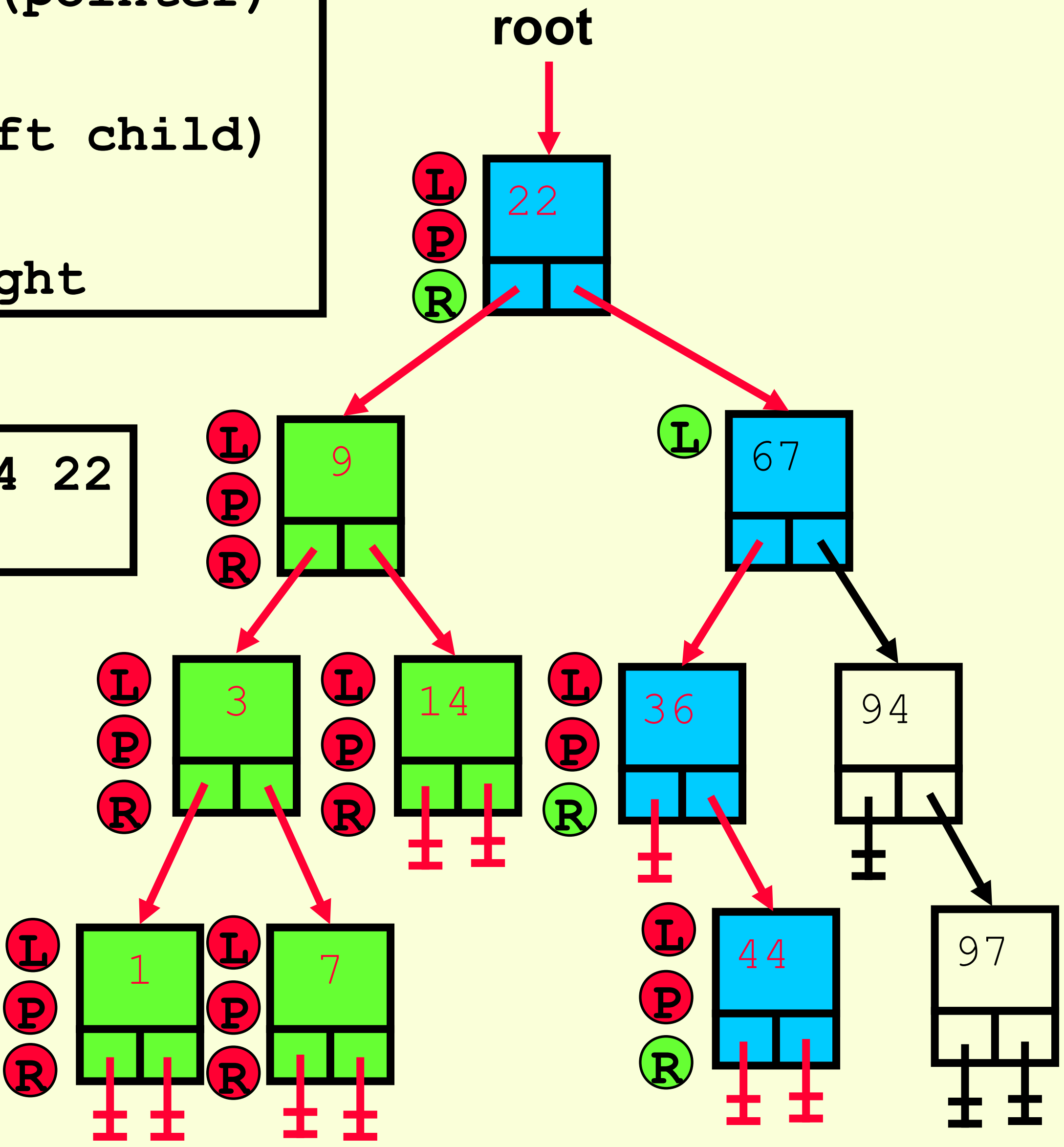
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44
```



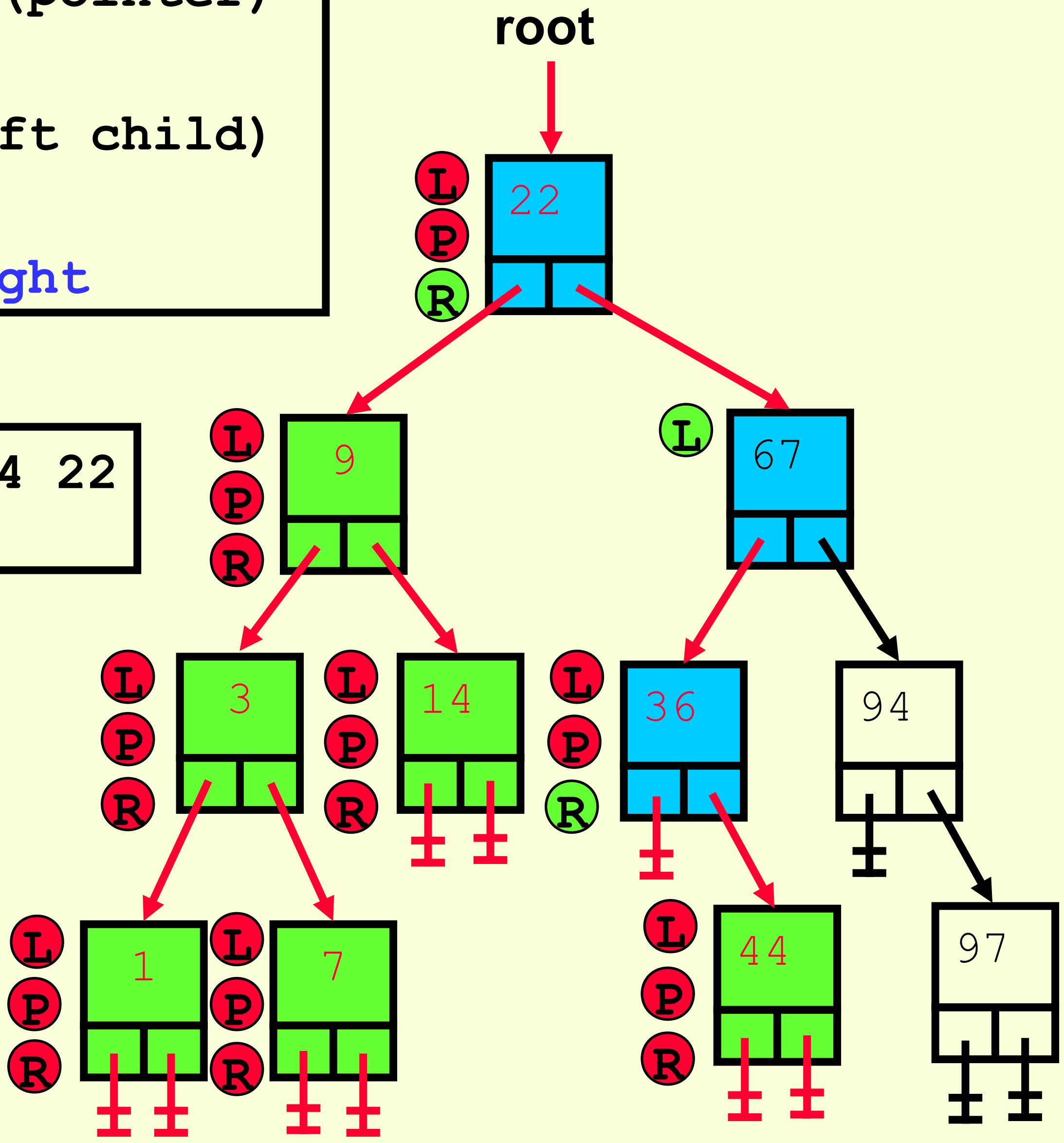
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44
```



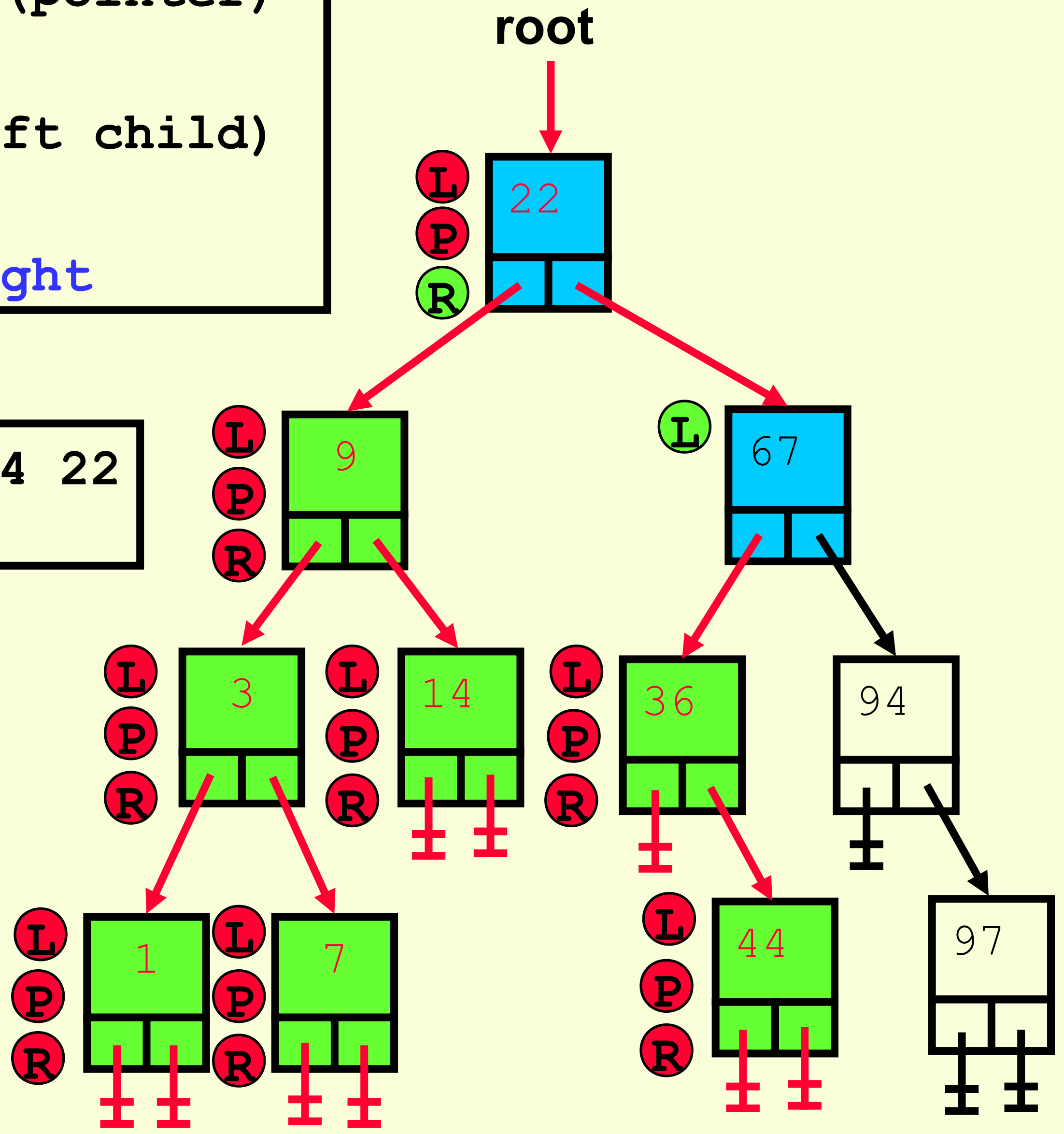
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44
```



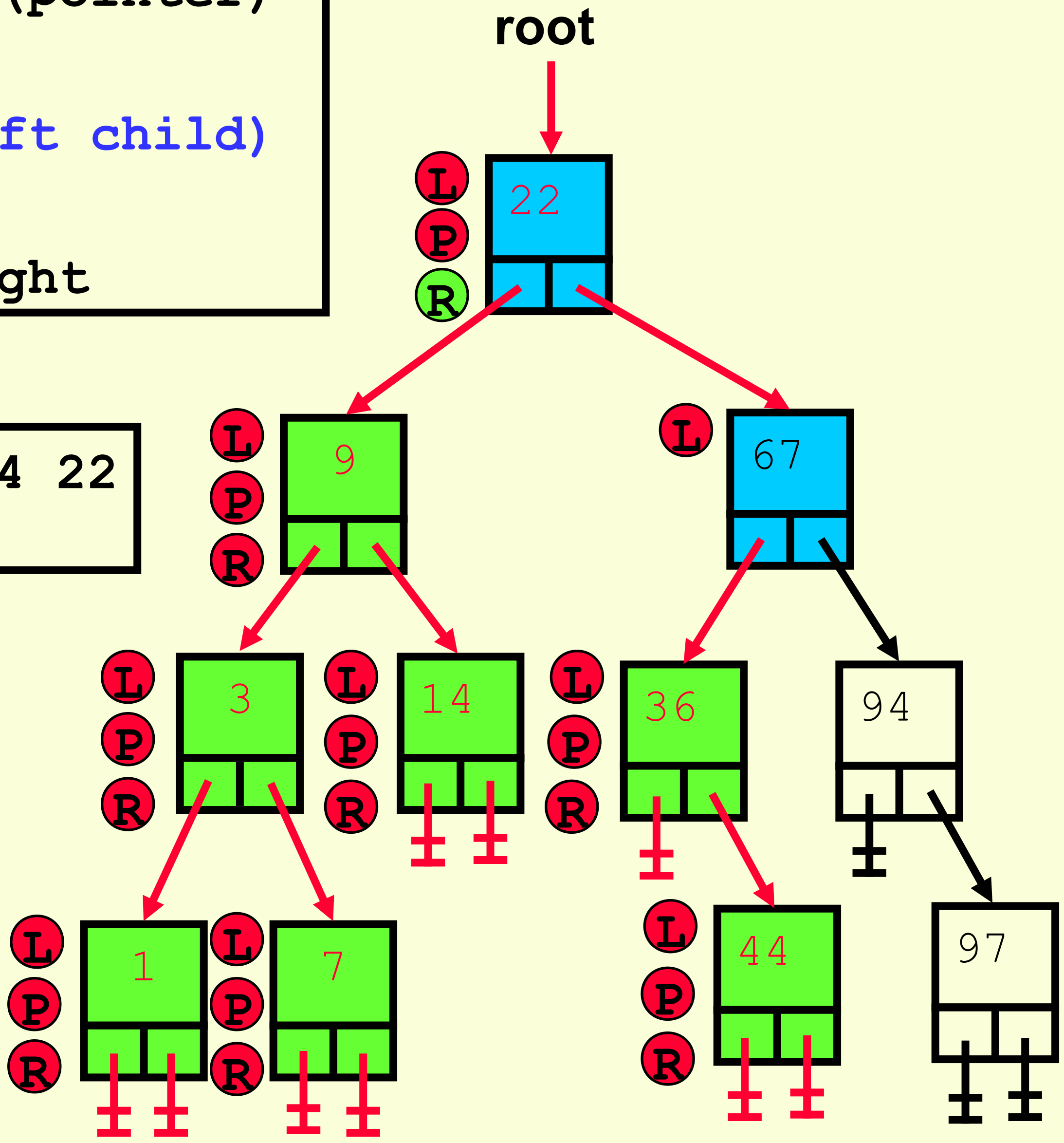
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44
```



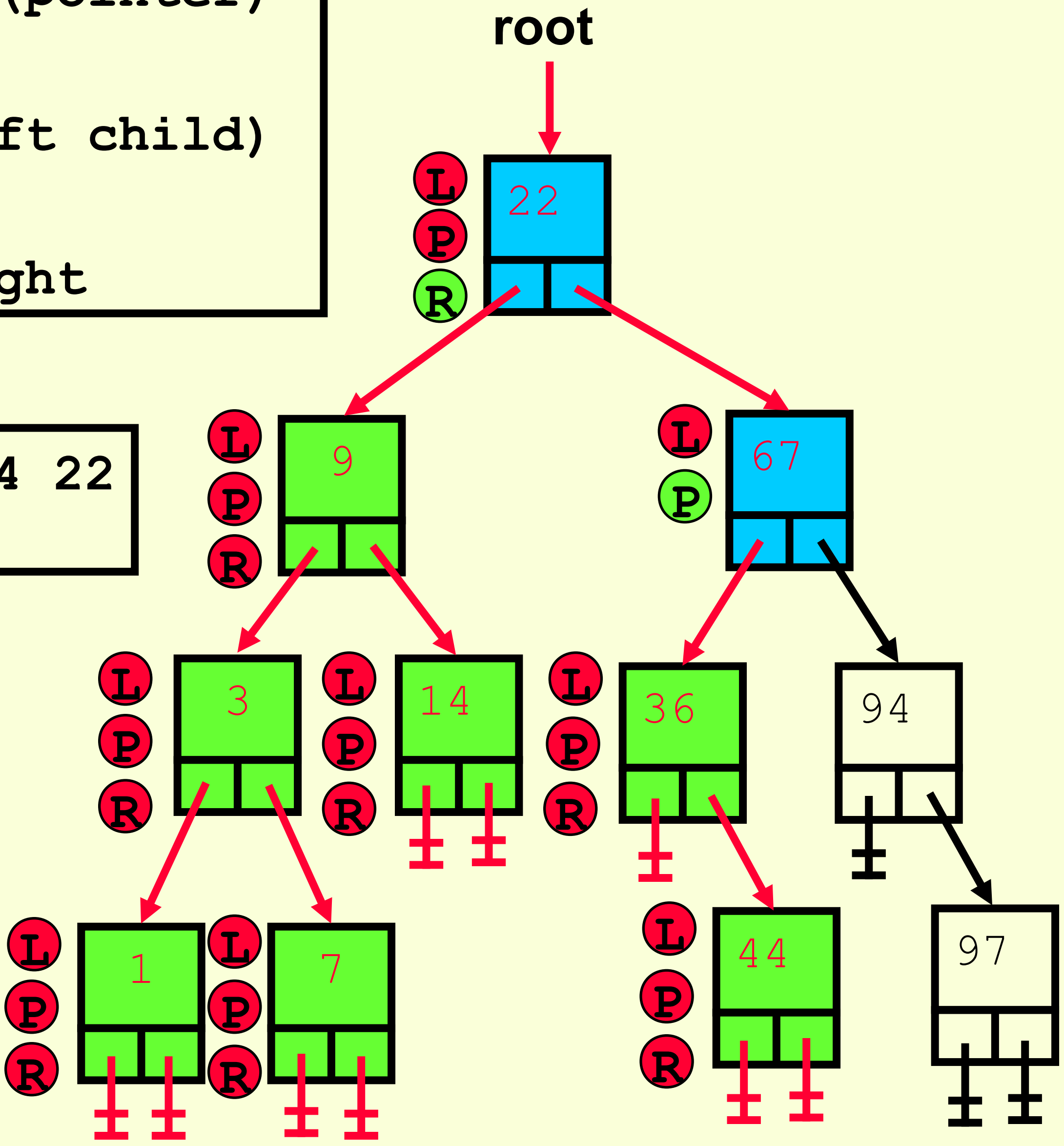
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

```
Output: 1 3 7 9 14 22
        36 44
```



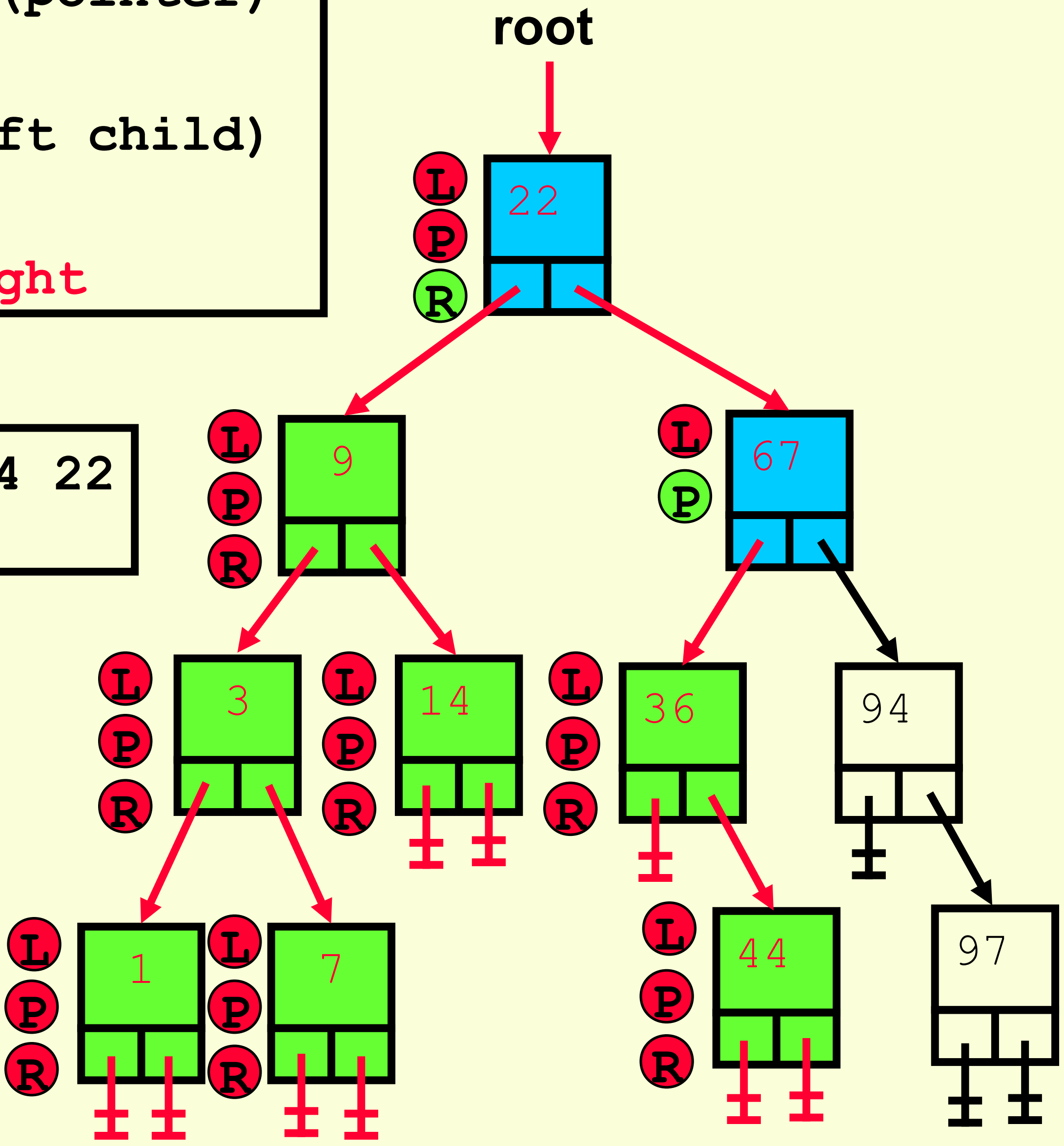
```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L   InOrderPrint(left child)
P   print(data)
R   InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44 67
```



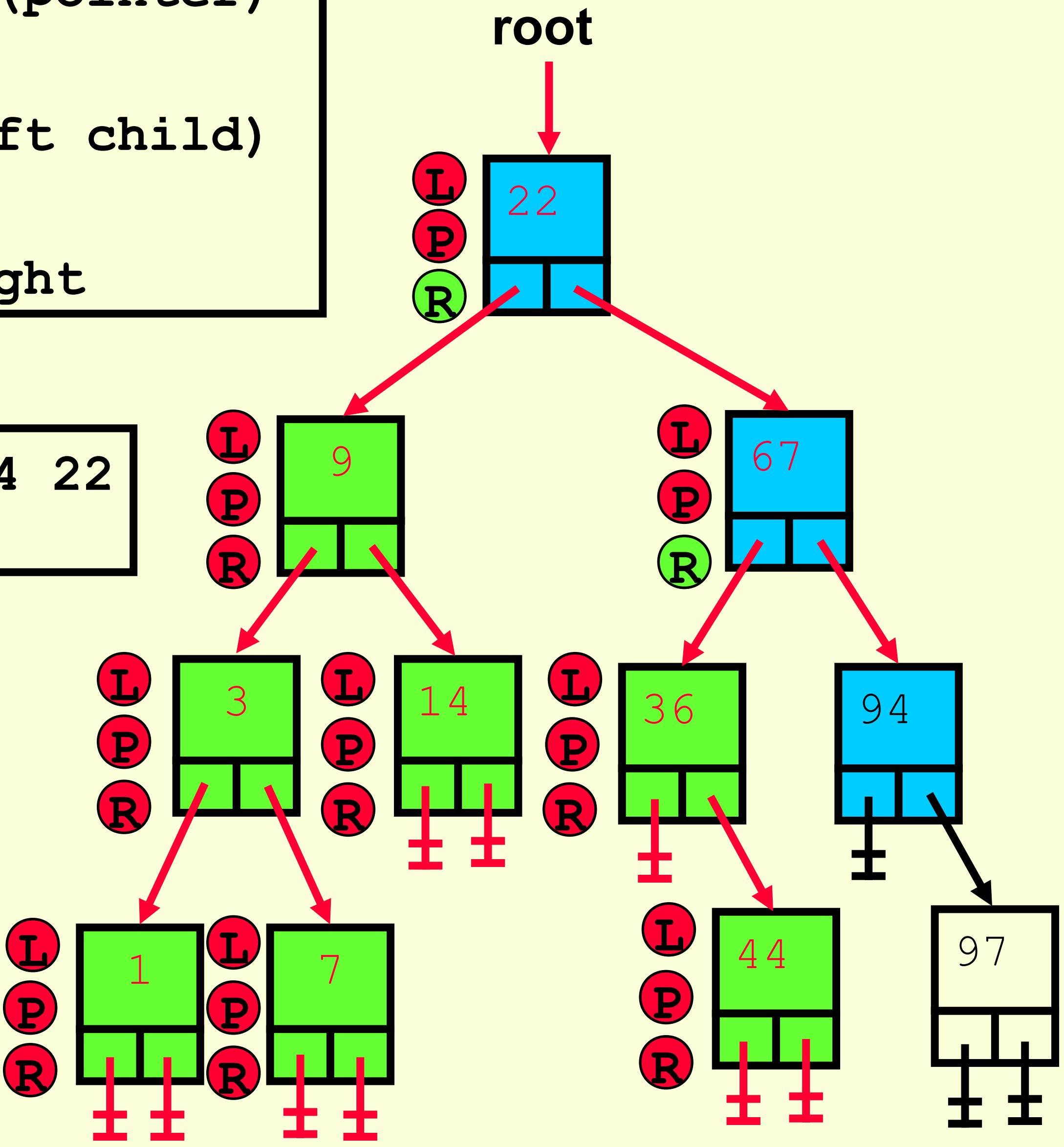
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44 67
```



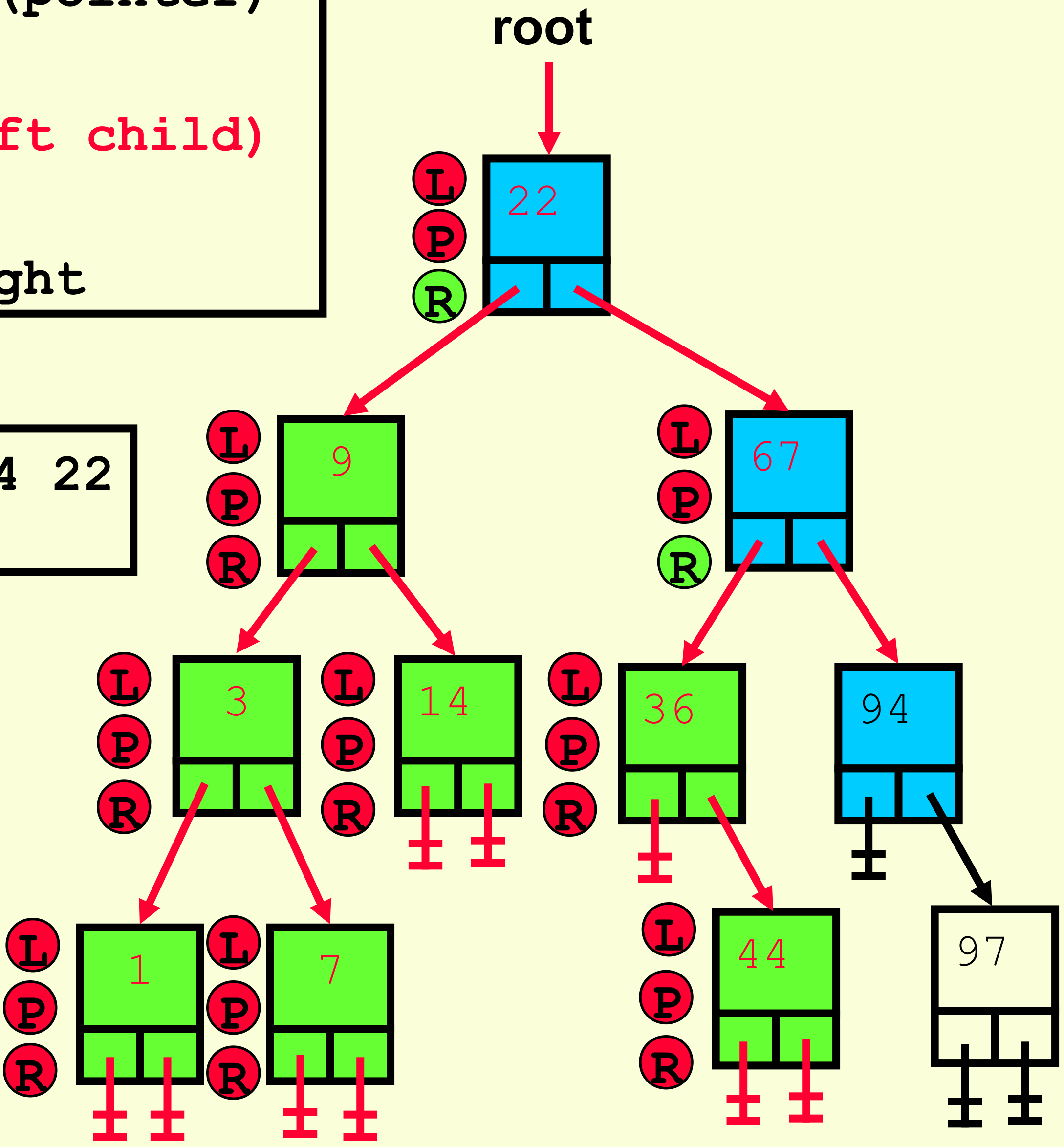
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

```
Output: 1 3 7 9 14 22
        36 44 67
```



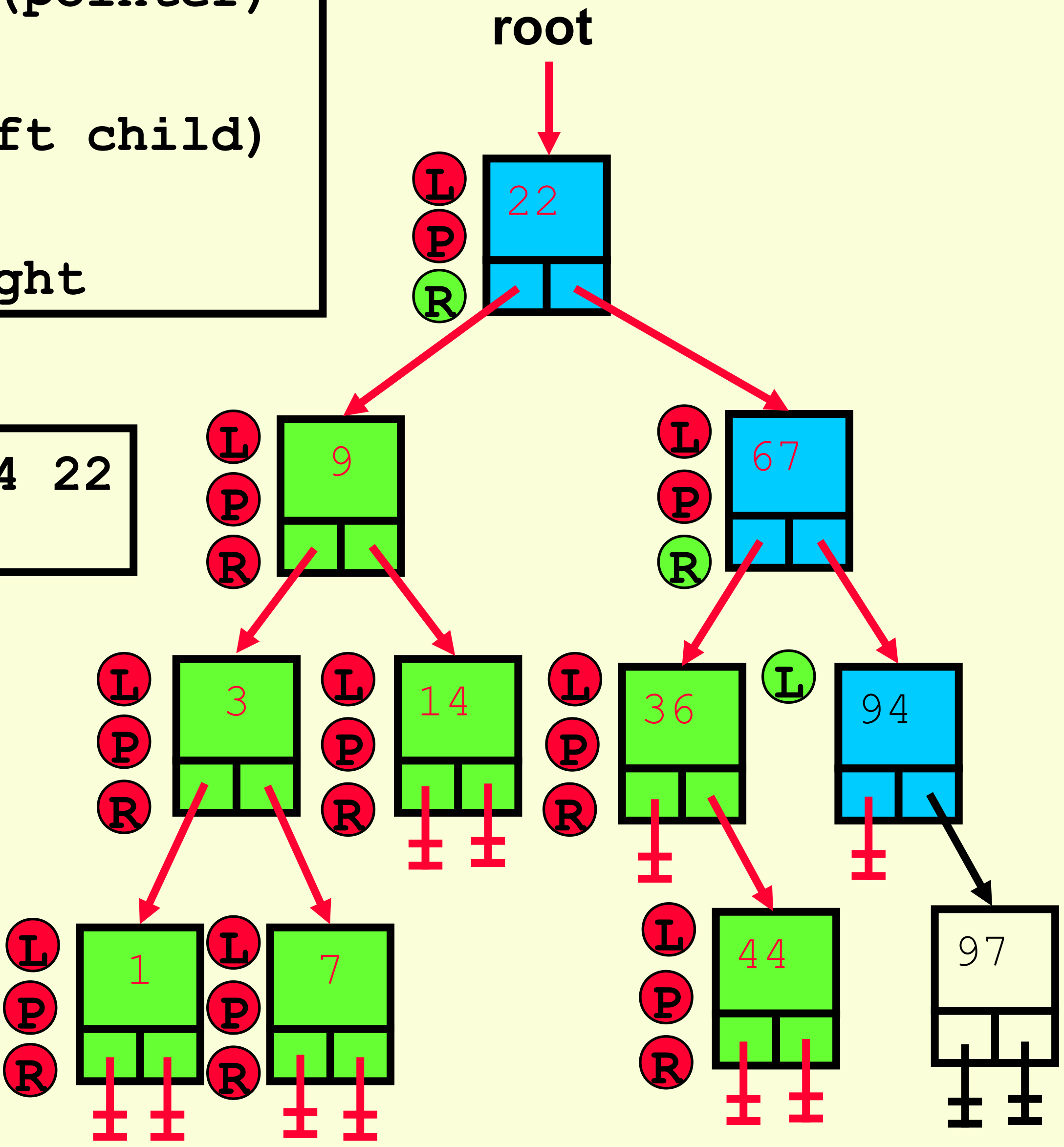
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44 67
```



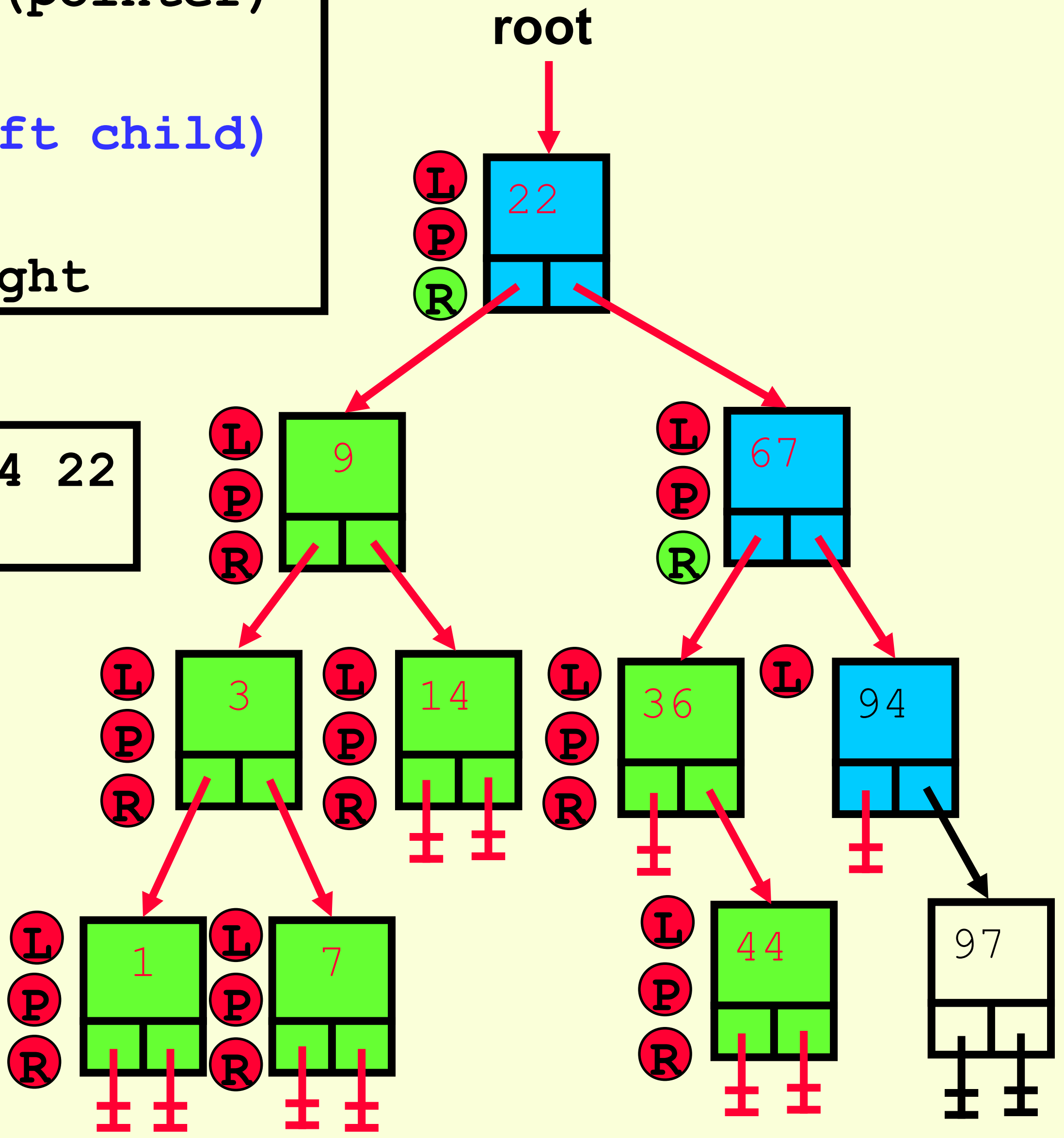
```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44 67
```



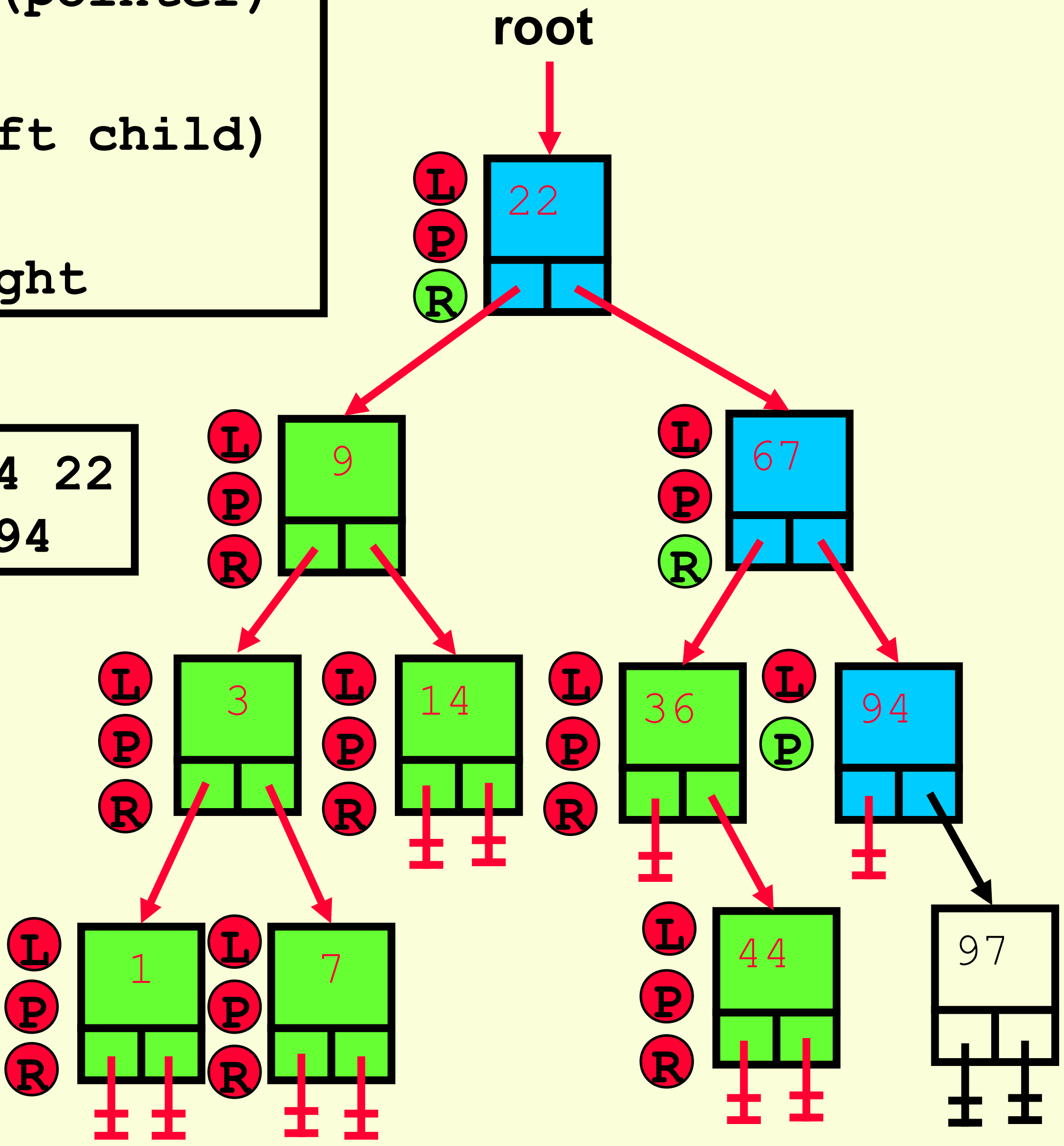
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44 67
```



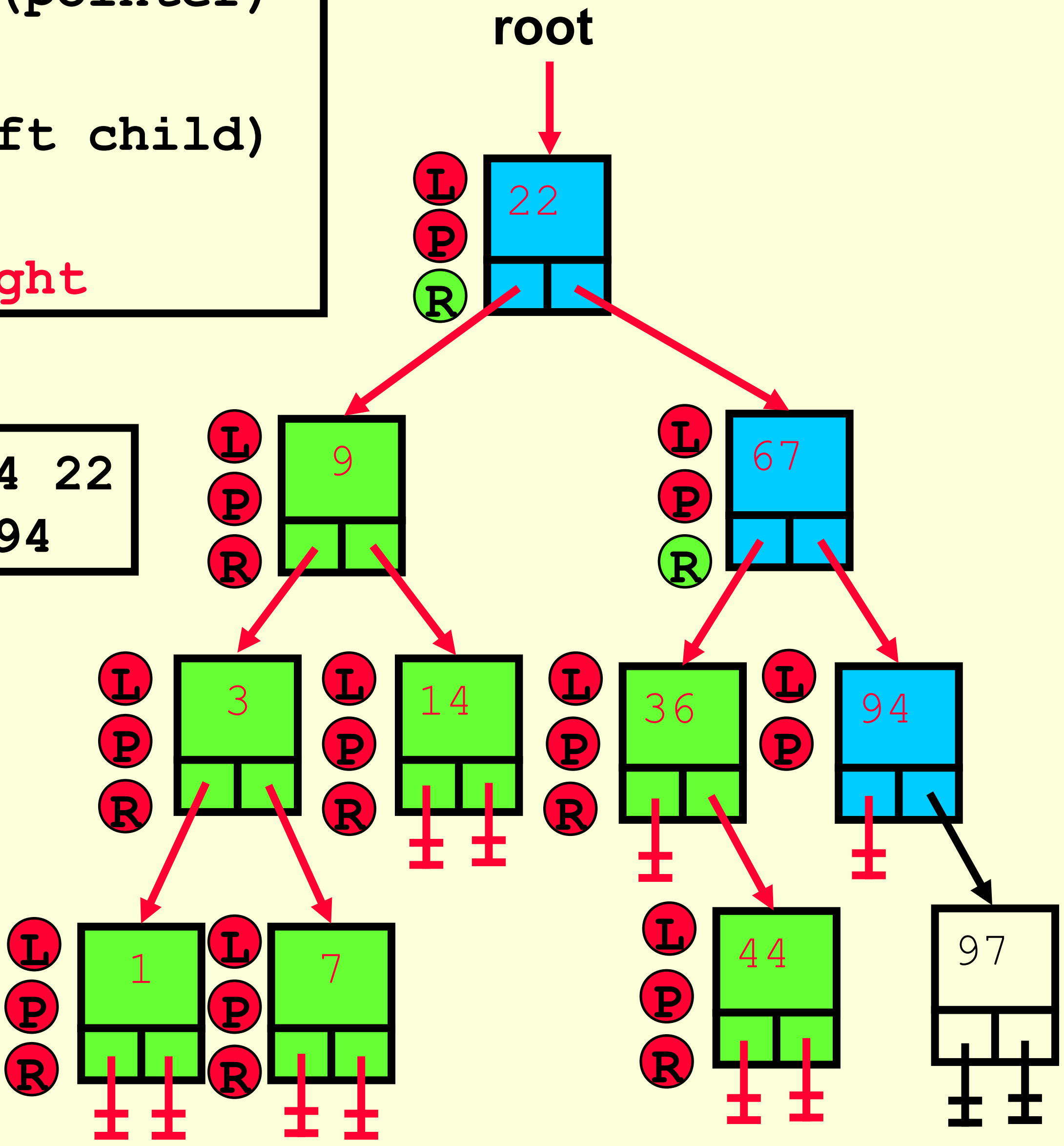
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44 67 94
```



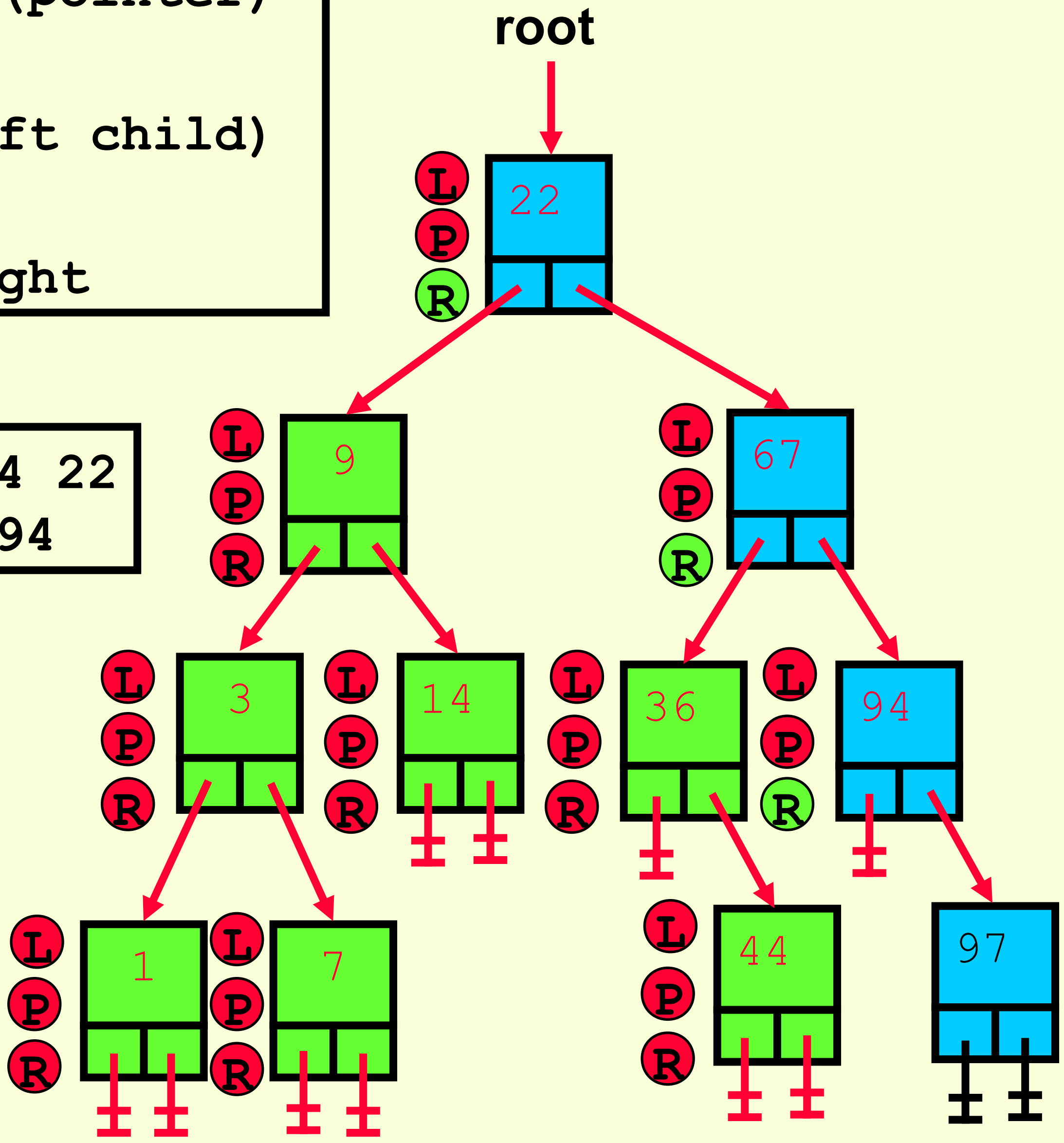
```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44 67 94
```



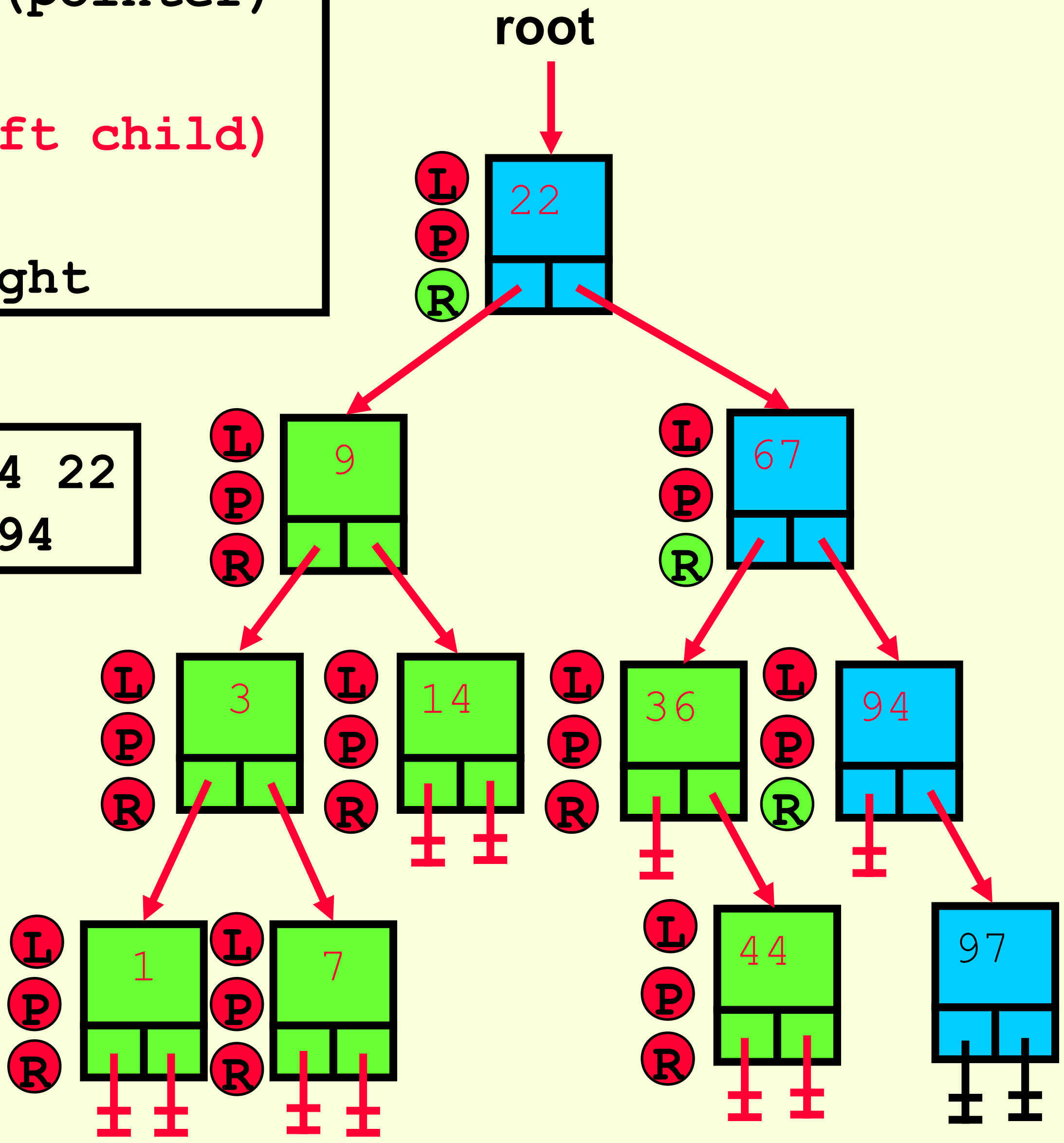
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44 67 94
```



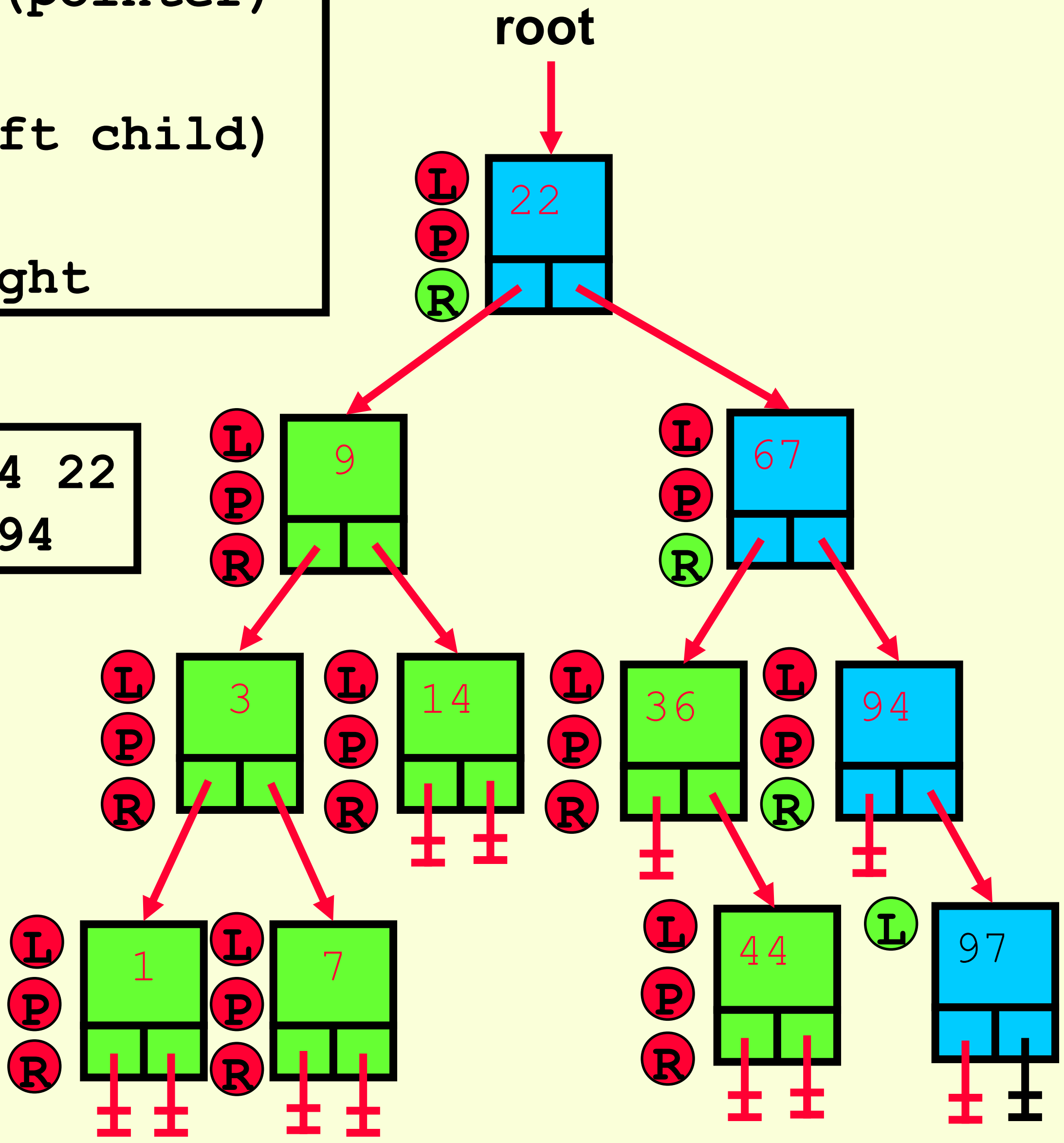
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44 67 94
```



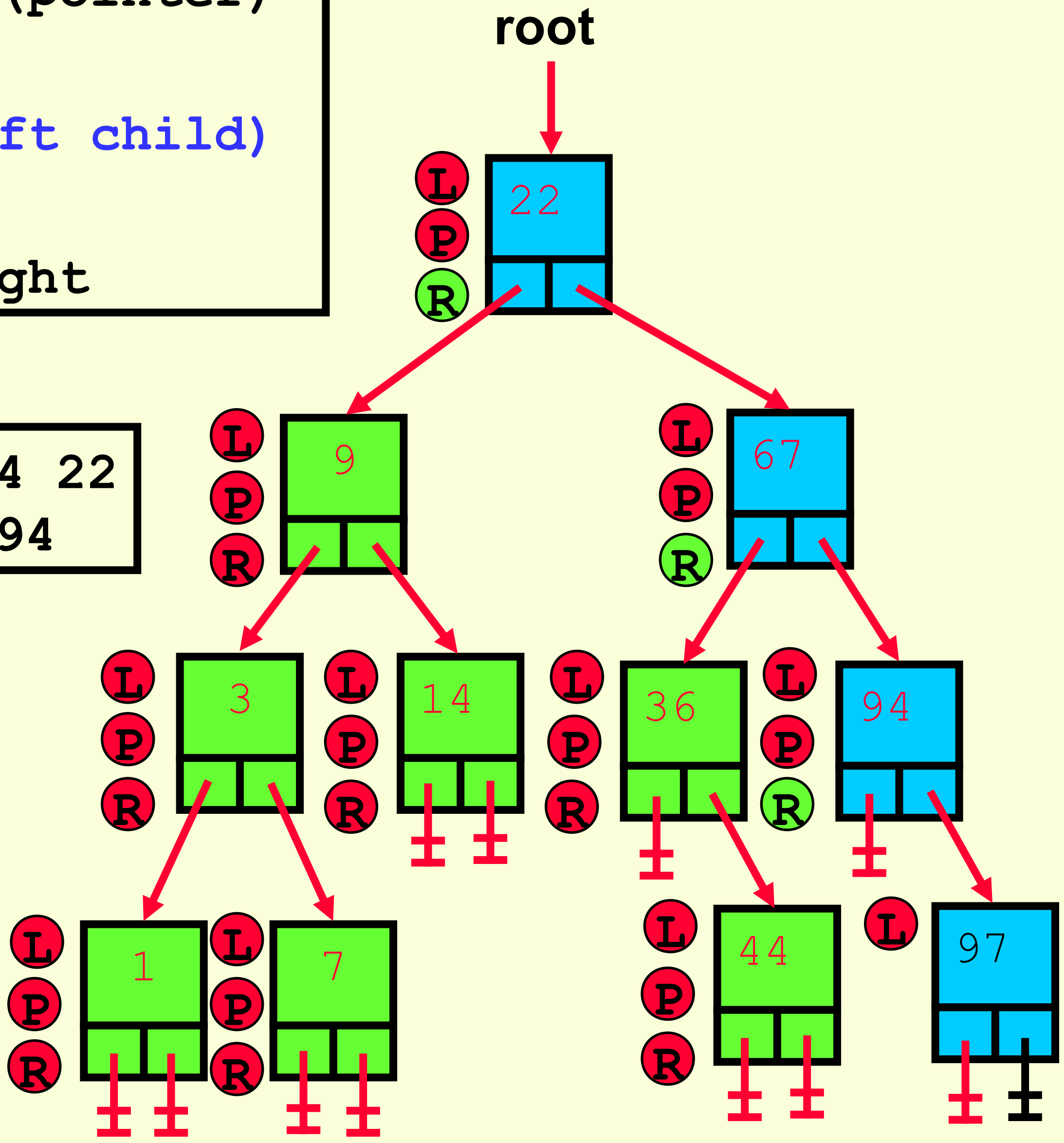
```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44 67 94
```



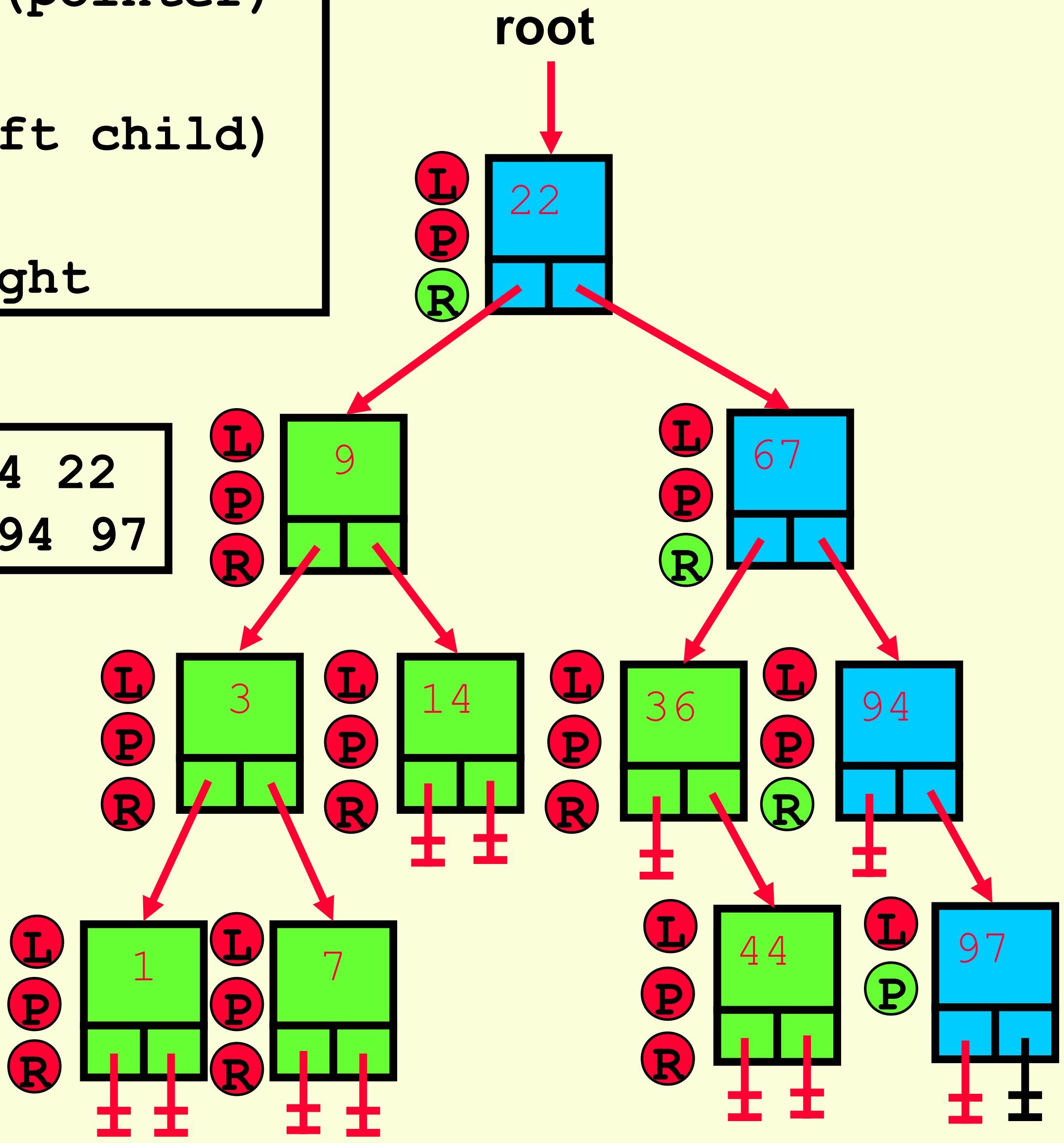
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44 67 94
```



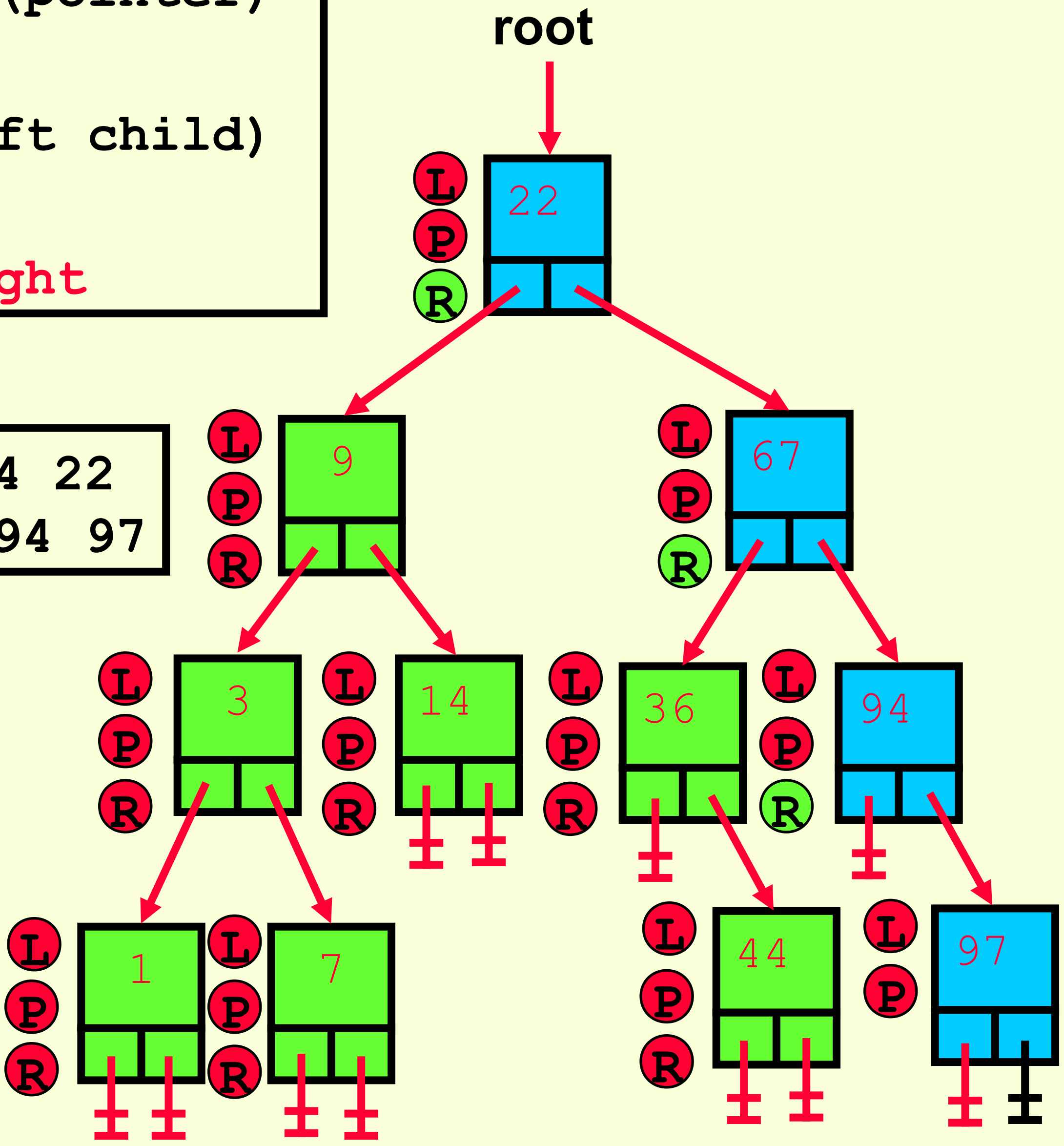
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44 67 94 97
```



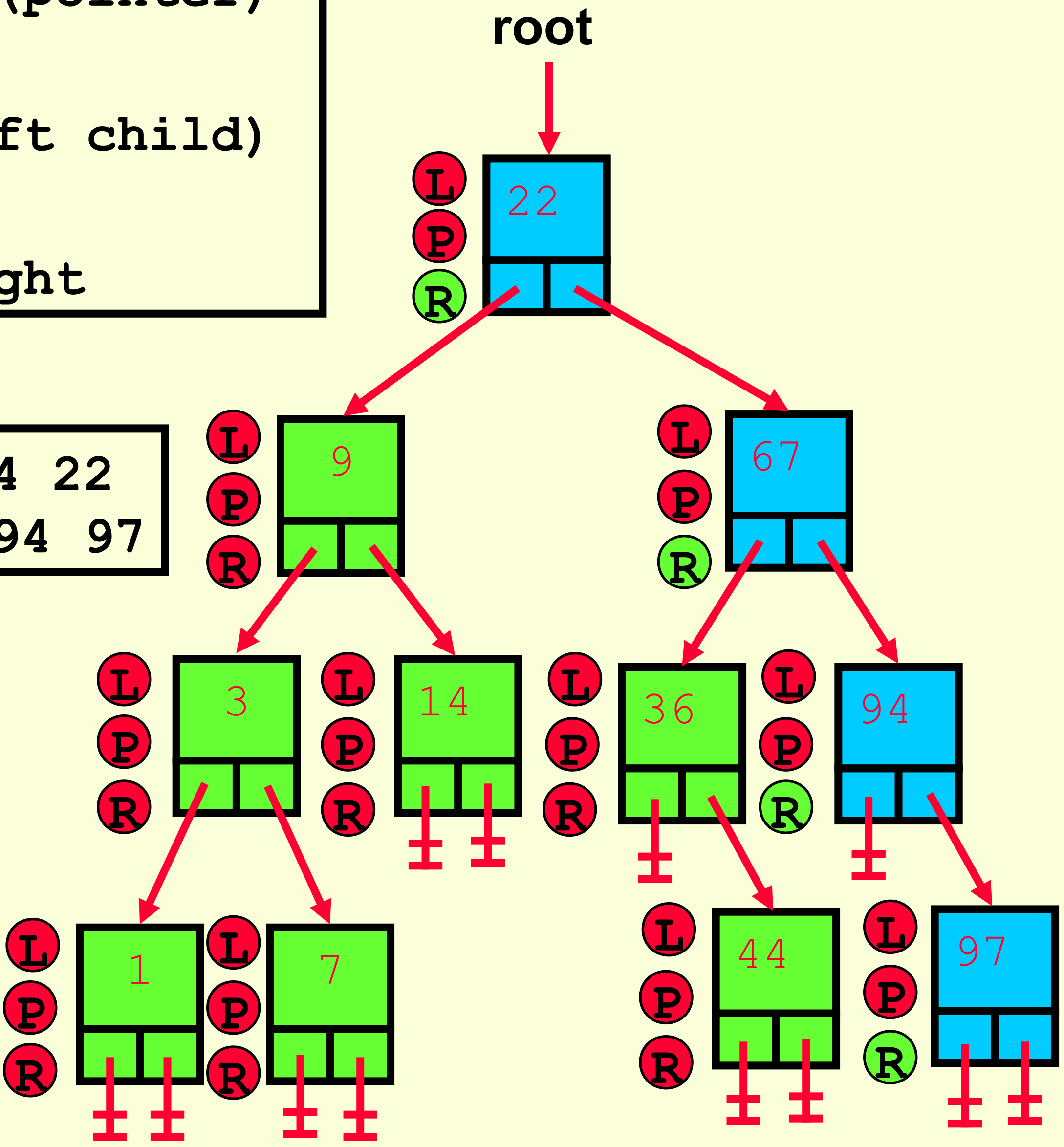
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44 67 94 97
```



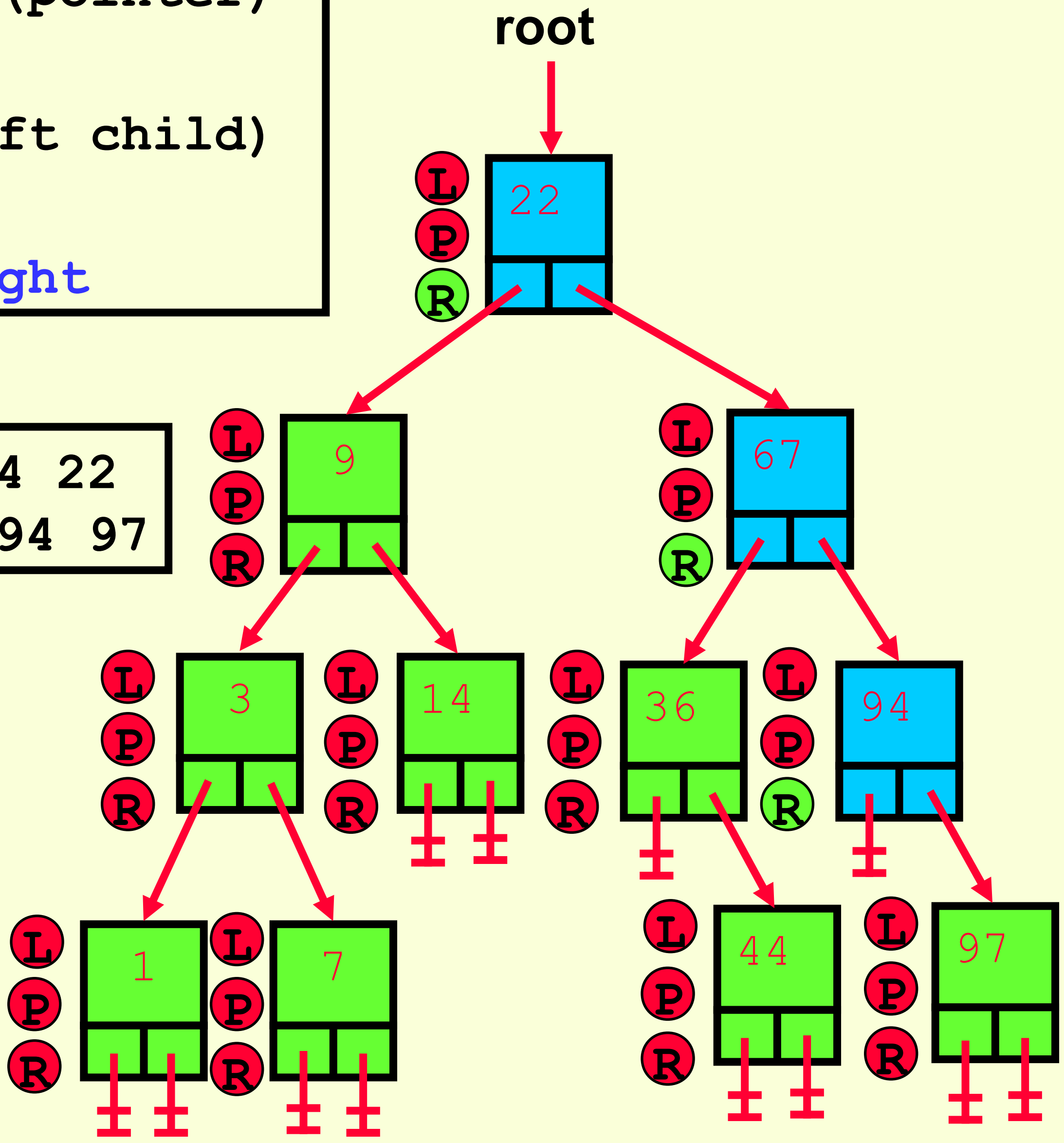
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44 67 94 97
```



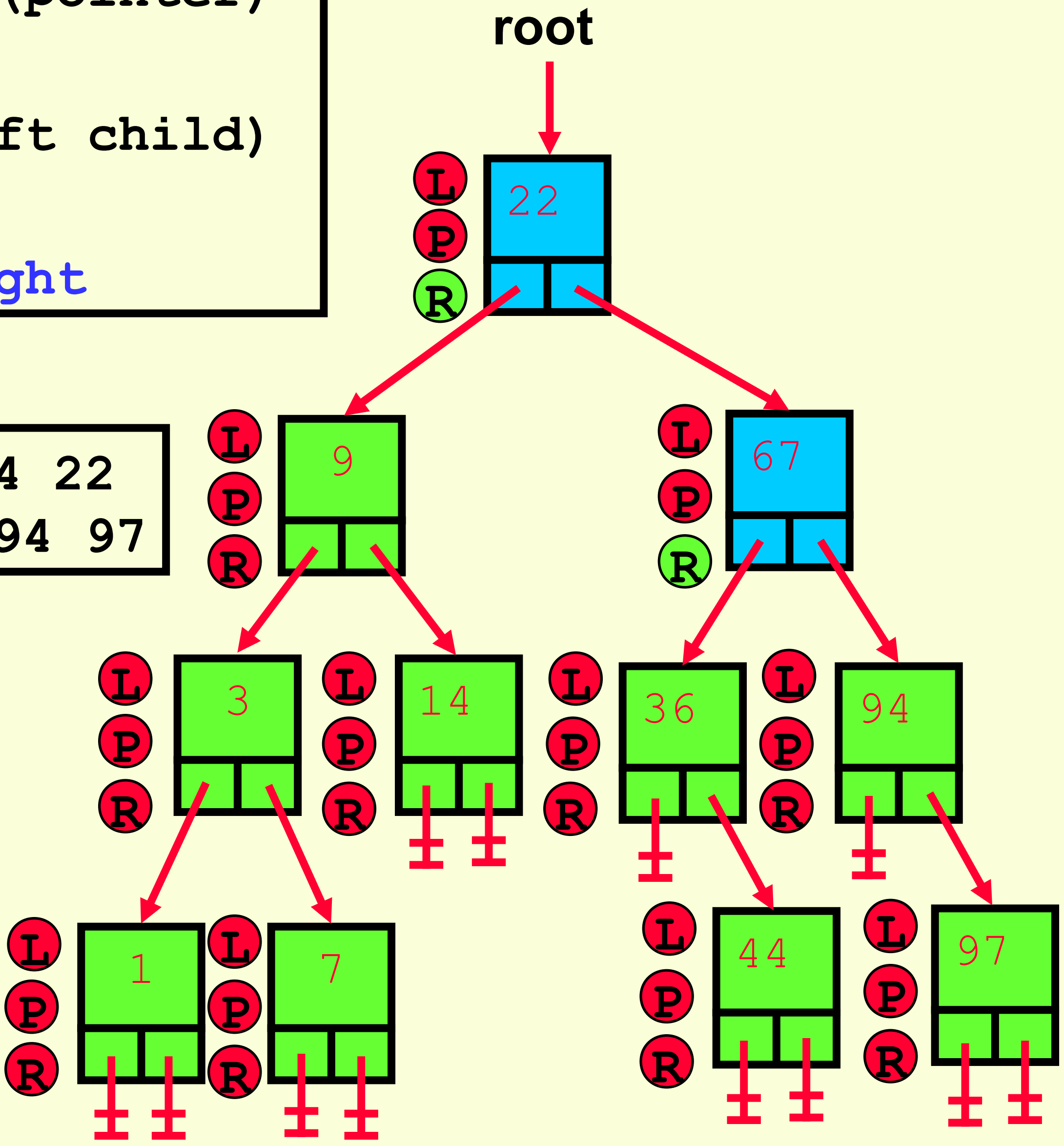
```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L   InOrderPrint(left child)
P   print(data)
R   InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44 67 94 97
```



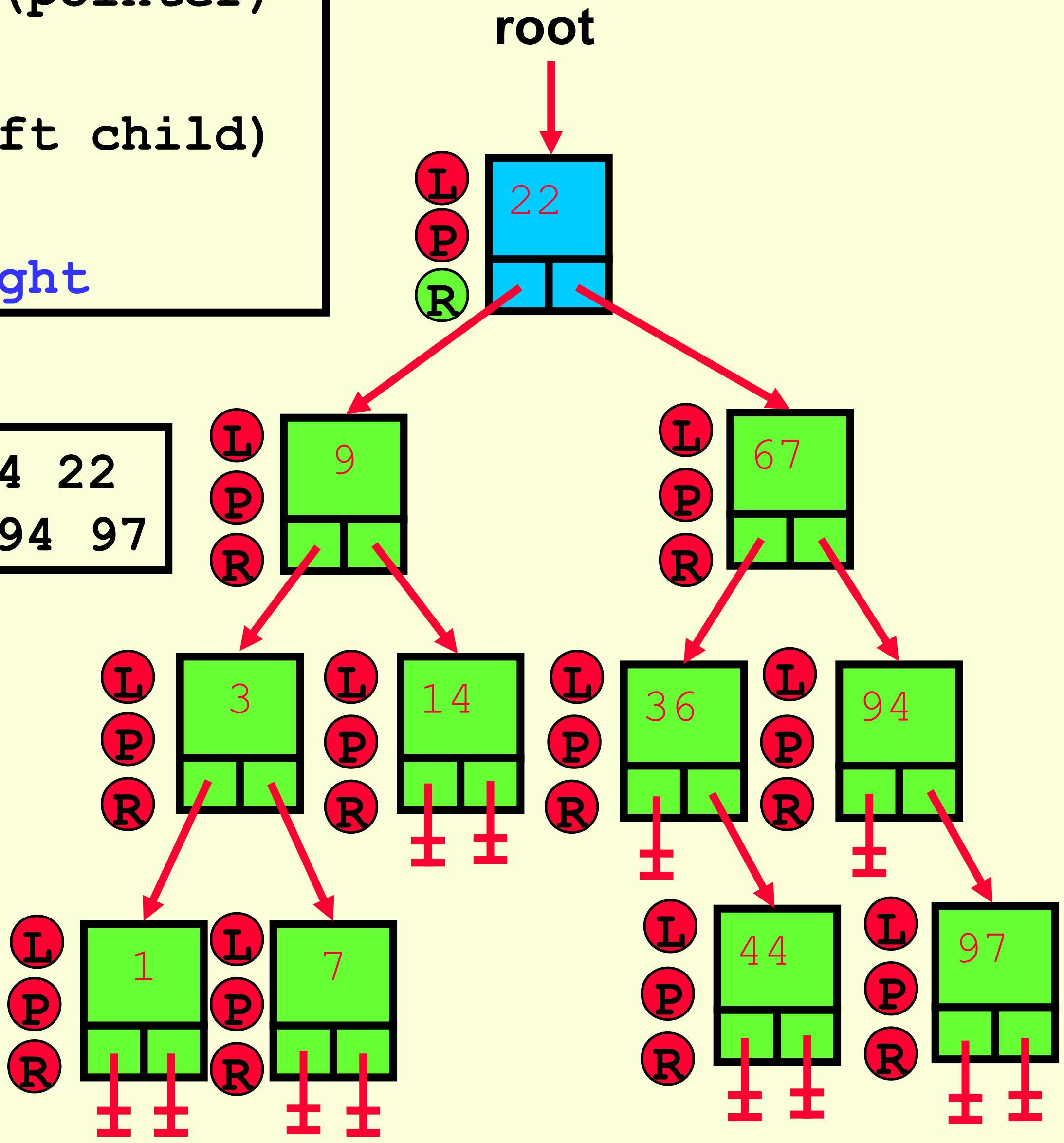
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

```
Output: 1 3 7 9 14 22
        36 44 67 94 97
```



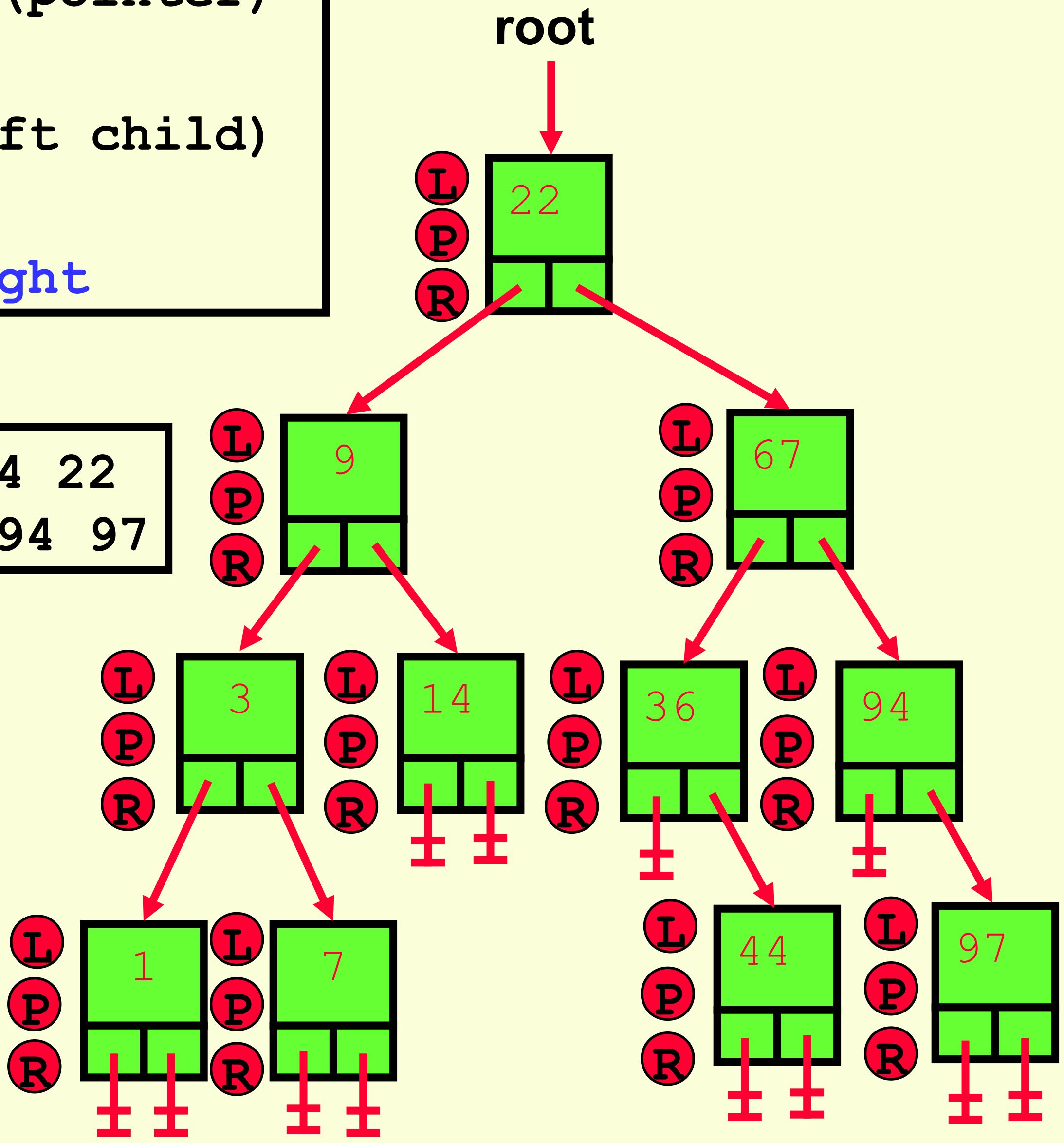
```
Proc InOrderPrint(pointer)
  pointer NOT NIL?
L  InOrderPrint(left child)
P  print(data)
R  InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44 67 94 97
```



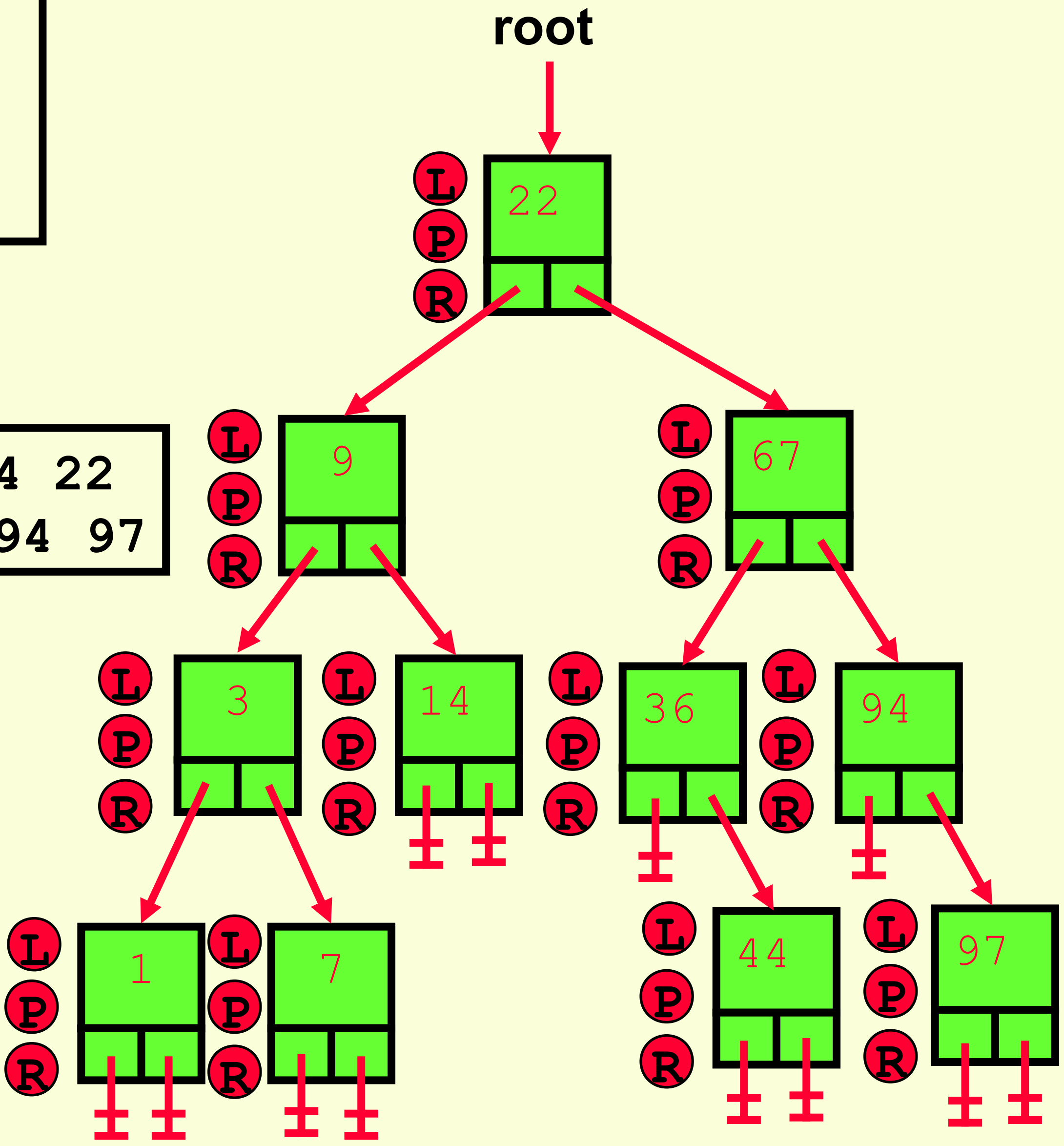
```
Proc InOrderPrint(pointer)
 pointer NOT NIL?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right
child)
```

```
Output: 1 3 7 9 14 22
        36 44 67 94 97
```



```
Algorithm Example
. . .
InOrderPrint(root
)
. . .
```

```
Output: 1 3 7 9 14 22
        36 44 67 94 97
```



Summary

- An In-Order traversal visits every node
  - Recurse left first
  - Do something with current
  - Recurse right last
- The “left, current, right” logic is repeated recursively at every node.
- For a BST, an in-order traversal accesses the elements in ascending order.

Questions?

Binary Search Tree Insertion

Tree Node Defined

In general:

```
Node definesa record
   data isoftype <type>
   left, right isoftype ptr toa Node
endrecord
```

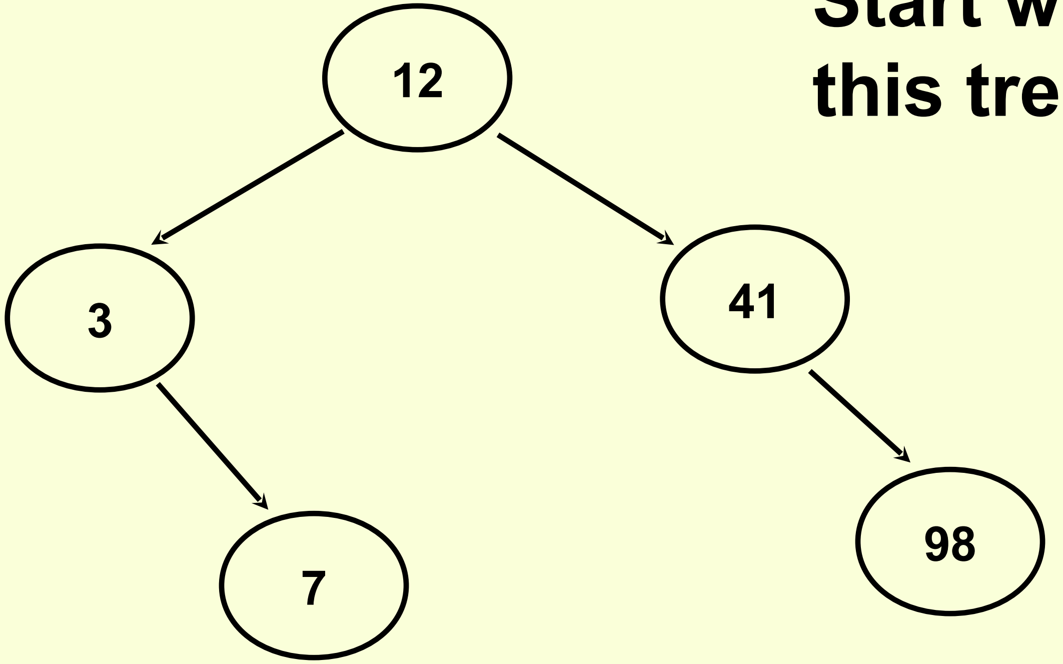
In this example:

```
Node definesa record
   data isoftype num
   left, right isoftype ptr toa Node
endrecord
```

Scenario

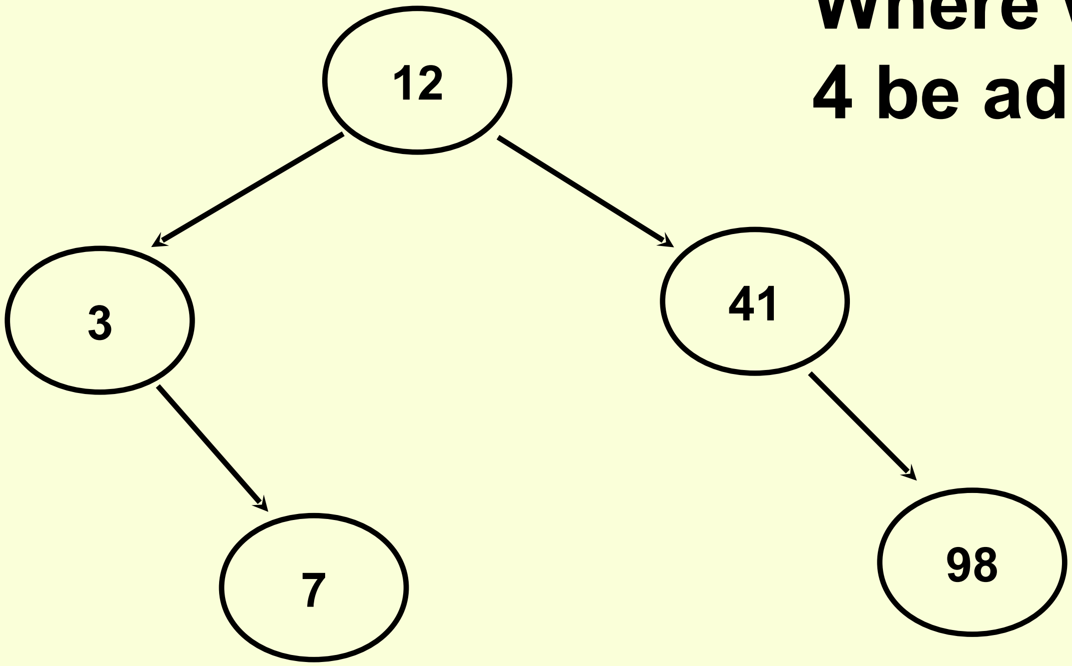
- We have a Binary Search Tree
  - It can be empty
  - Or have some elements in it already
- We want to add an element to it
  - Inserting/adding involves 2 steps:
    - Find the correct location
    - Do the steps to add a new node
- Must maintain “search” structure

Finding the Correct Location



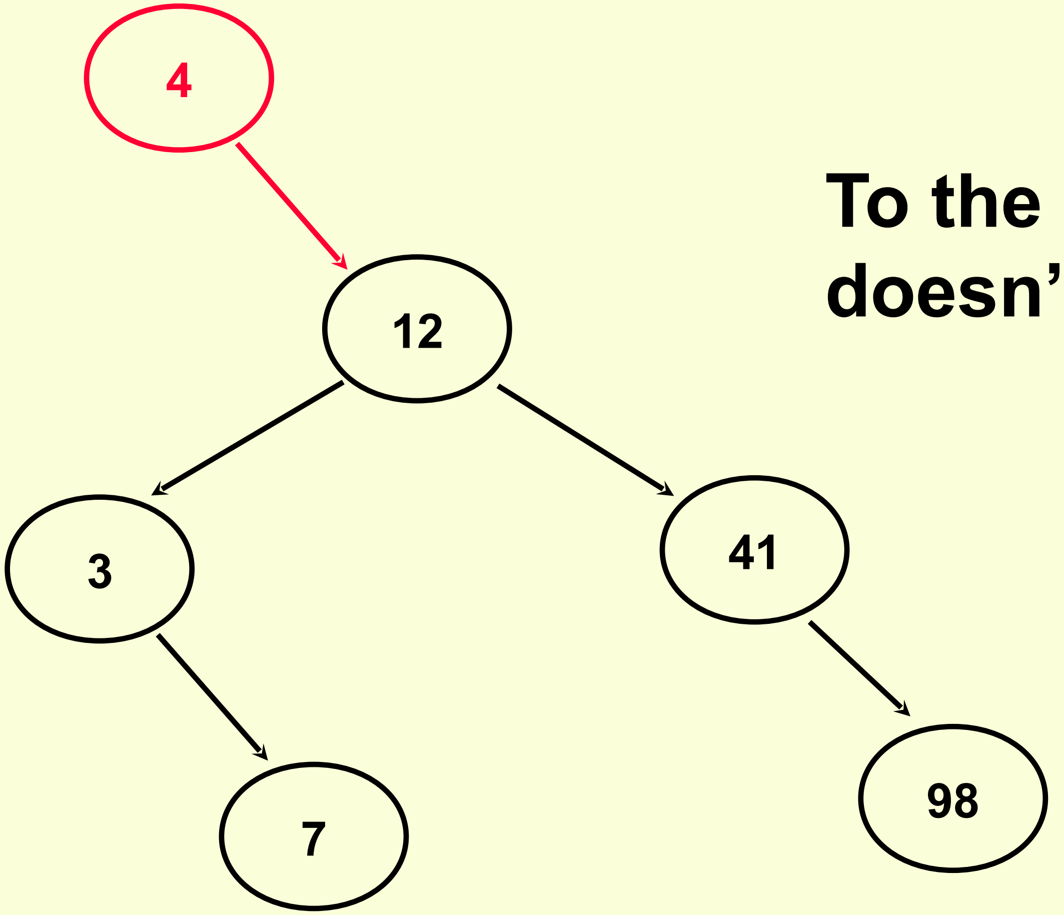
Start with this tree

Finding the Correct Location



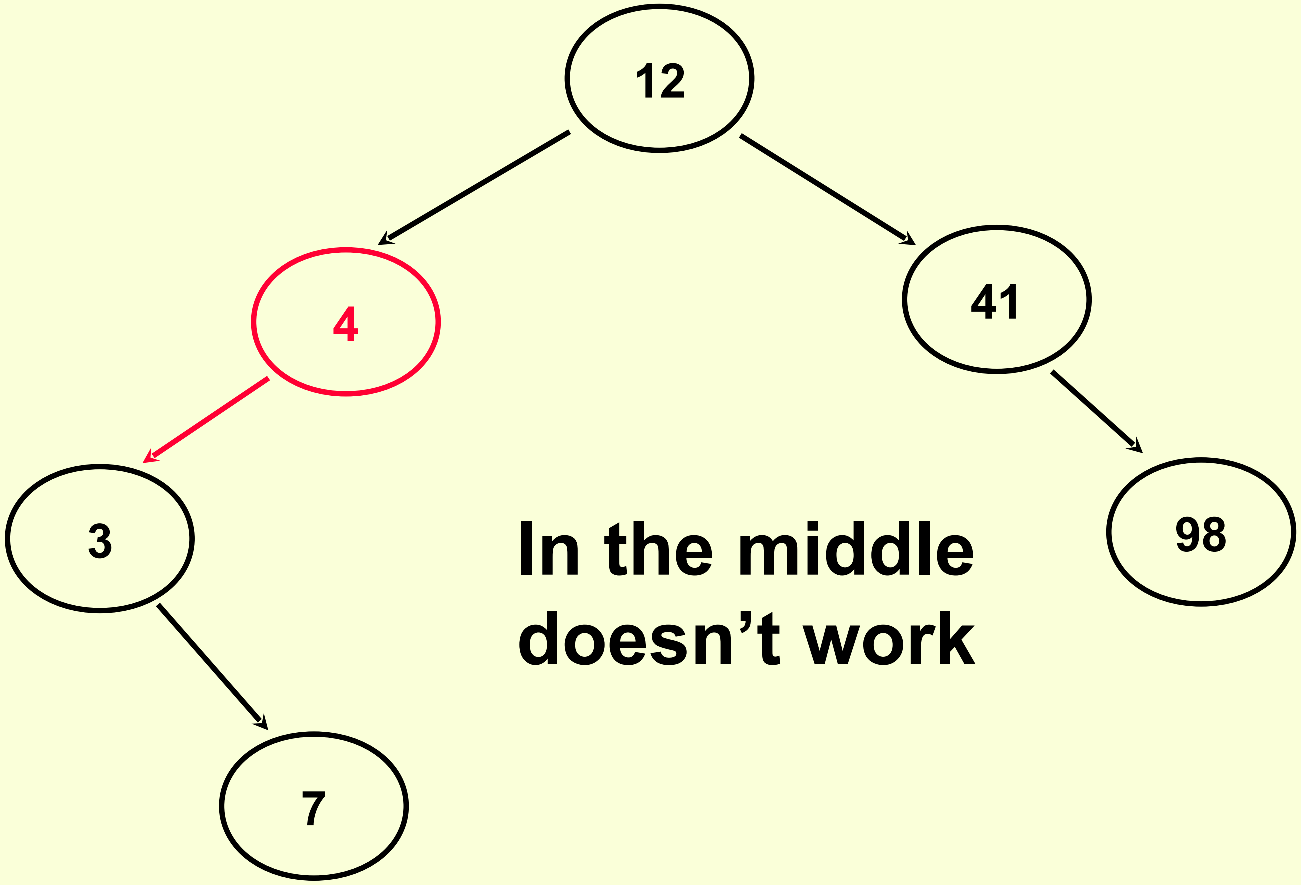
Where would 4 be added?

Finding the Correct Location



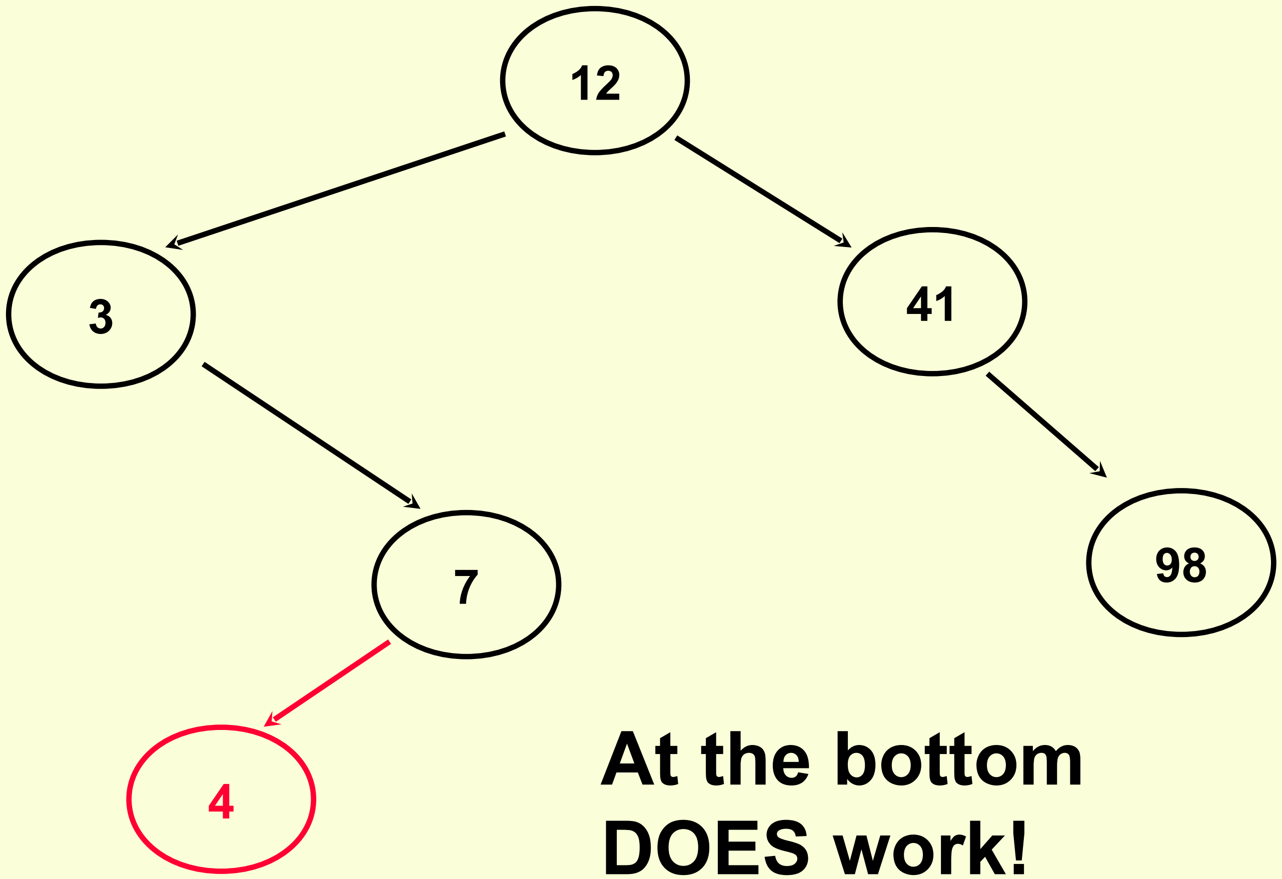
To the top doesn't work

Finding the Correct Location



In the middle doesn’t work

Finding the Correct Location



At the bottom DOES work!

Finding the Correct Location

- Must maintain “search” structure
  - Everything to left is less than current
  - Everything to right is greater than current
- Adding at the “bottom” guarantees we keep search structure.
- We’ll recurse to get to the “bottom” (i.e. when current = nil)

Finding the Correct Location

```
if (current = nil)
  DO “ADD NODE” WORK HERE
elseif (current^.data > value_to_add) then
  // recurse left
  Insert(current^.left, value_to_add)
else
  // recurse right
  Insert(current^.right, value_to_add)
endif
```

Adding the Node

- Current is an in/out pointer
  - We need information IN to evaluate current
  - We need to send information OUT because we’re changing the tree (adding a node)

- Once we’ve found the correct location:
  - Create a new node
  - Fill in the data field
    (with the new value to add)
  - Make the left and right pointers point to nil
    (to cleanly terminate the tree)

Adding the Node

```
current <- new(Node)
current^.data <- value_to_add
current^.left <- nil
current^.right <- nil
```

The Entire Module

```
procedure Insert(cur iot in/out Ptr toa Node,
            data_in iot in num)
  if(cur = NIL) then
    cur <- new(Node)
    cur^.data <- data_in
    cur^.left <- NIL
    cur^.right <- NIL
  elseif(cur^.data > data_in)
    Insert(cur^.left, data_in)
  else
    Insert(cur^.right, data_in)
  endif
endprocedure // Insert
```

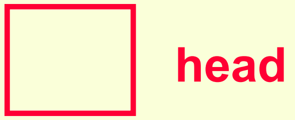
Tracing Example

The following example shows a trace of the BST insert.

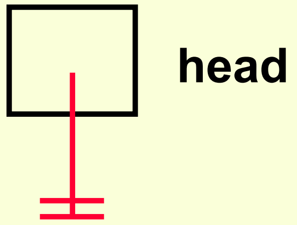
- Begin with an empty BST (a pointer)
- Add elements 42, 23, 35, 47 in the correct positions.

```
Head iot Ptr toa Node
head <- NIL
Insert(head, 42)
```

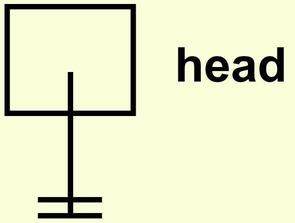
```
Head iot Ptr toa Node
head <- NIL
Insert(head, 42)
```



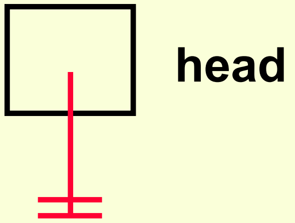
```
Head iot Ptr toa Node
head <- NIL
Insert(head, 42)
```



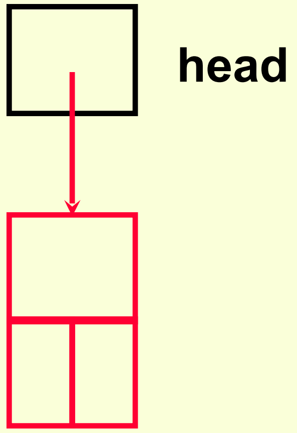
```
Head iot Ptr toa Node
head <- NIL
Insert(head, 42)
```



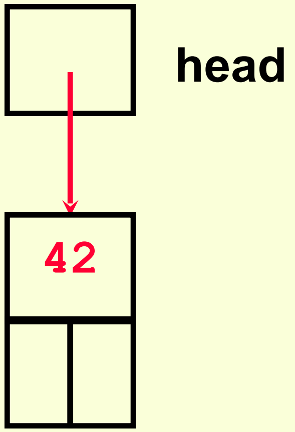
```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



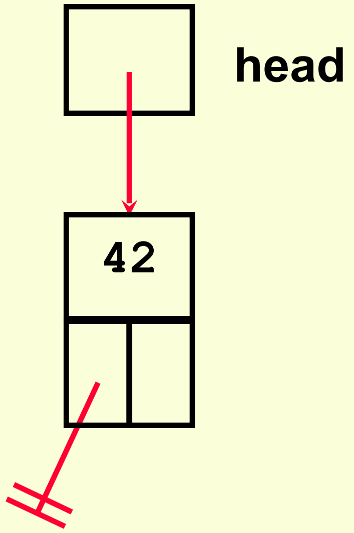
```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



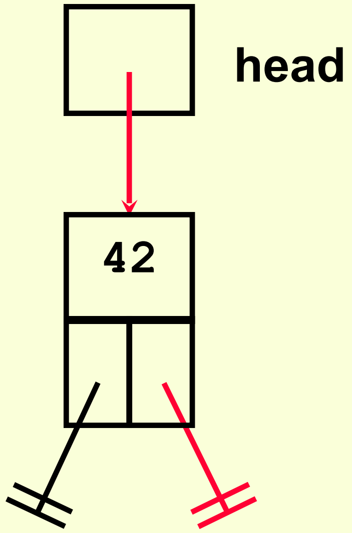
```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



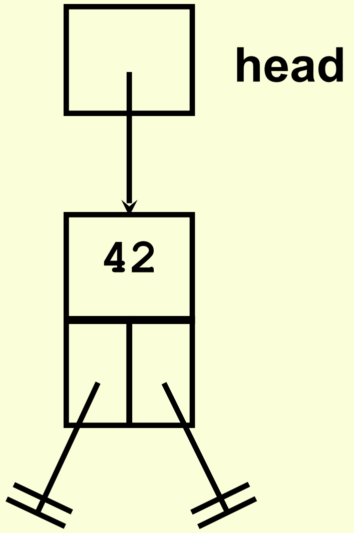
```
procedure Insert(
  cur iot in/out Ptr toa Node,
  data_in iot in num)
  if(cur = NIL) then
    cur <- new(Node)
    cur^.data <- data_in
    cur^.left <- NIL
    cur^.right <- NIL
  elseif(cur^.data > data_in)
    Insert(cur^.left, data_in)
  else
    Insert(cur^.right, data_in)
  endif
endprocedure // Insert
```



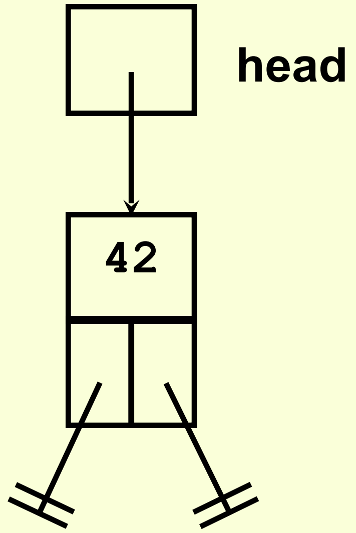
```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



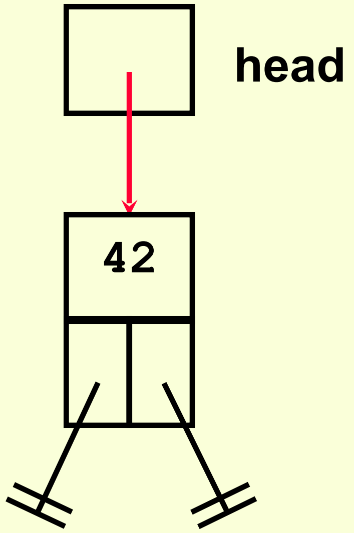
```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



```
.
.
Insert(head, 23)
Insert(head, 35)
Insert(head, 47)
.
.
```

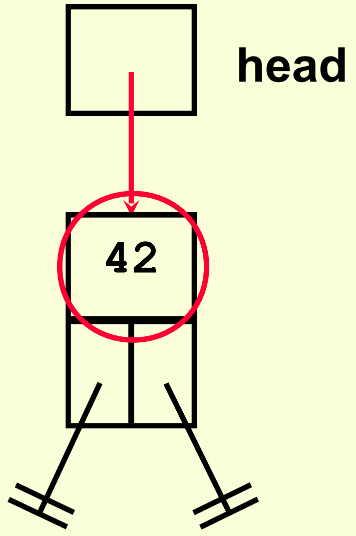


```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



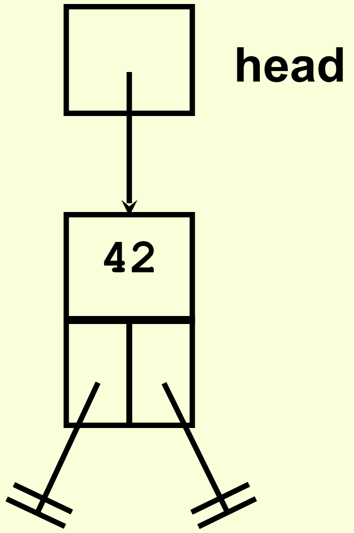
data_in = 23

```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



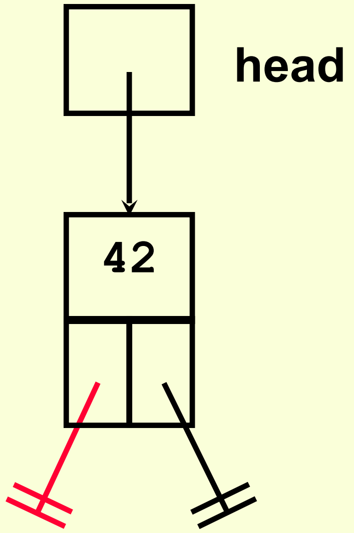
data_in = 23

```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



data_in = 23

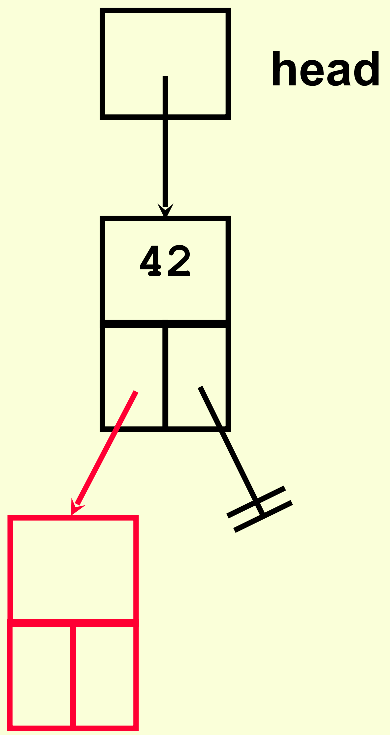
```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



data_in = 23

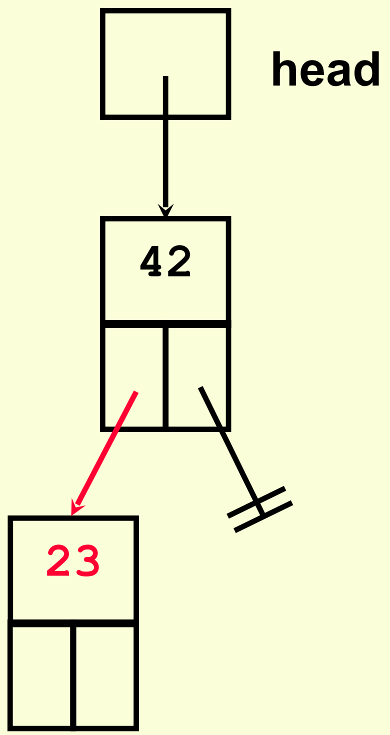
```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```

data_in = 23



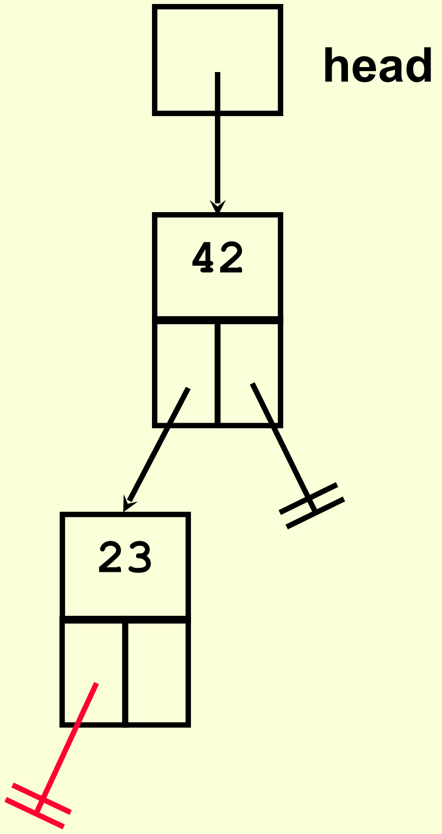
```
procedure Insert(
  cur iot in/out Ptr toa Node,
  data_in iot in num)
  if(cur = NIL) then
    cur <- new(Node)
    cur^.data <- data_in
    cur^.left <- NIL
    cur^.right <- NIL
  elseif(cur^.data > data_in)
    Insert(cur^.left, data_in)
  else
    Insert(cur^.right, data_in)
  endif
endprocedure // Insert
```

data_in = 23



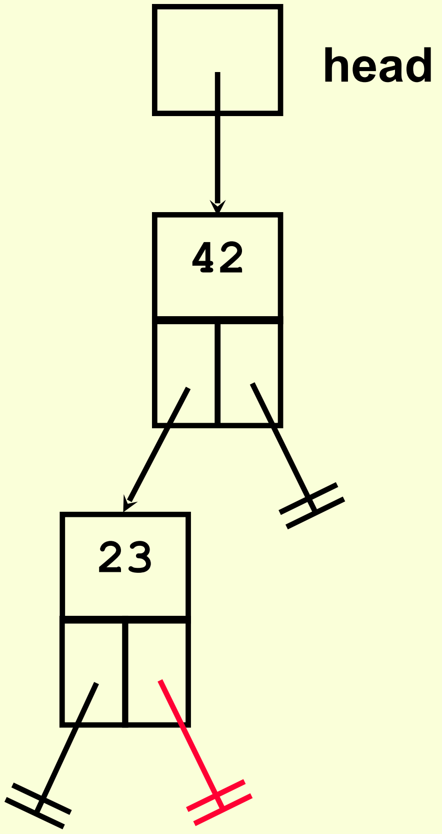
```
procedure Insert(
  cur iot in/out Ptr toa Node,
  data_in iot in num)
  if(cur = NIL) then
    cur <- new(Node)
    cur^.data <- data_in
    cur^.left <- NIL
    cur^.right <- NIL
  elseif(cur^.data > data_in)
    Insert(cur^.left, data_in)
  else
    Insert(cur^.right, data_in)
  endif
endprocedure // Insert
```

```
data_in = 23
```



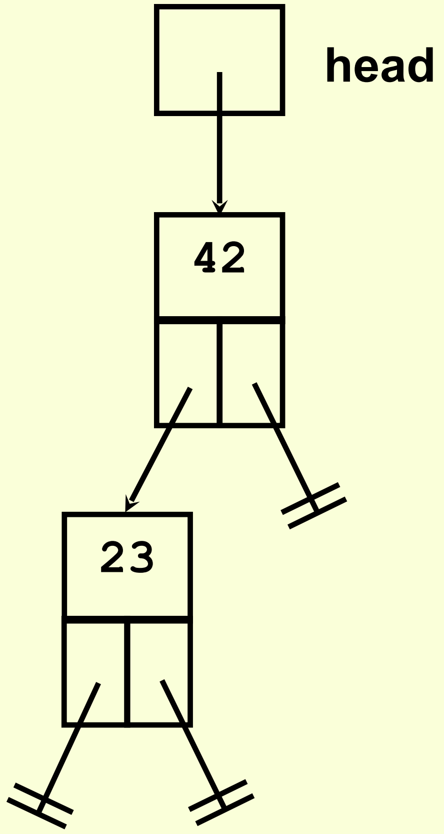
```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```

data_in = 23

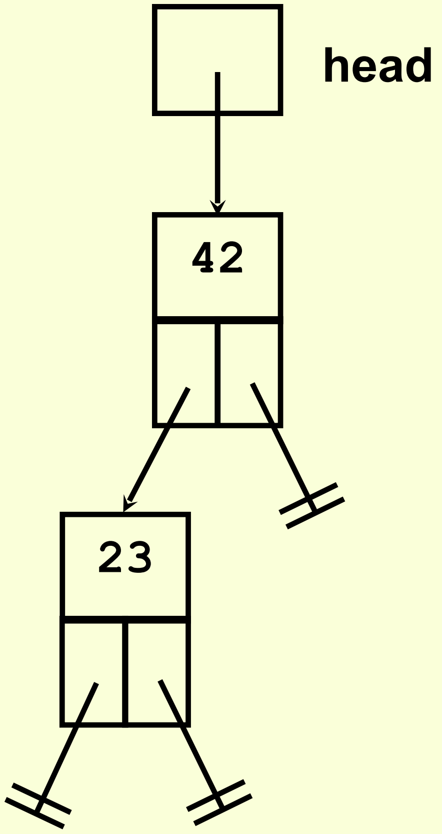


```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```

data_in = 23

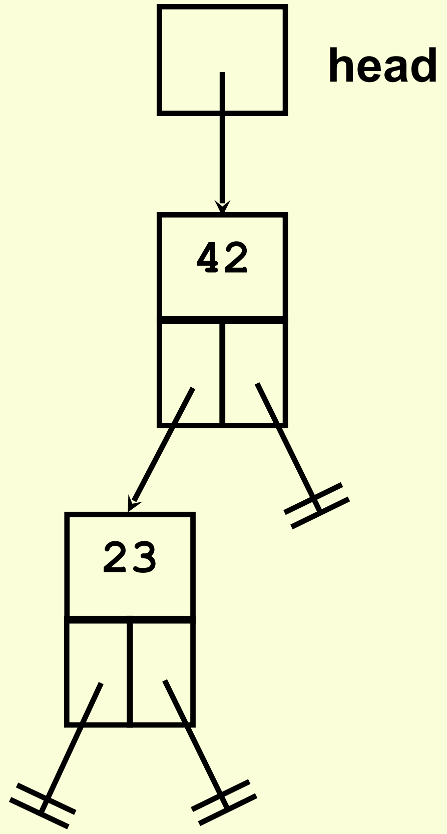


```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```

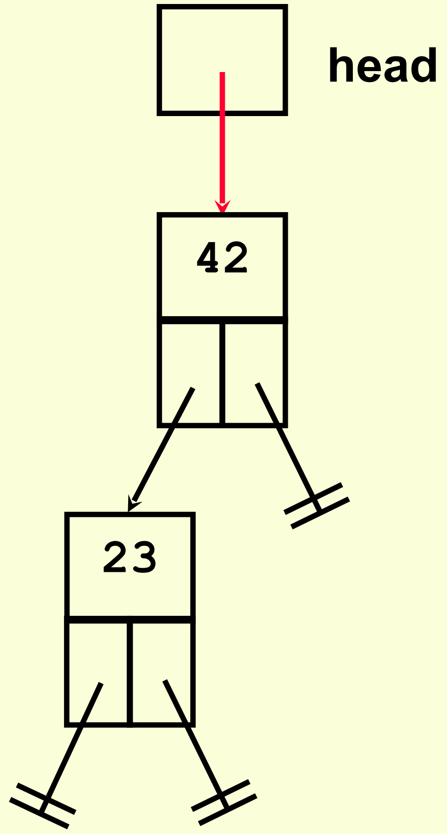


data_in = 23

```
.
.
Insert(head, 23)
Insert(head, 35)
Insert(head, 47)
.
.
```

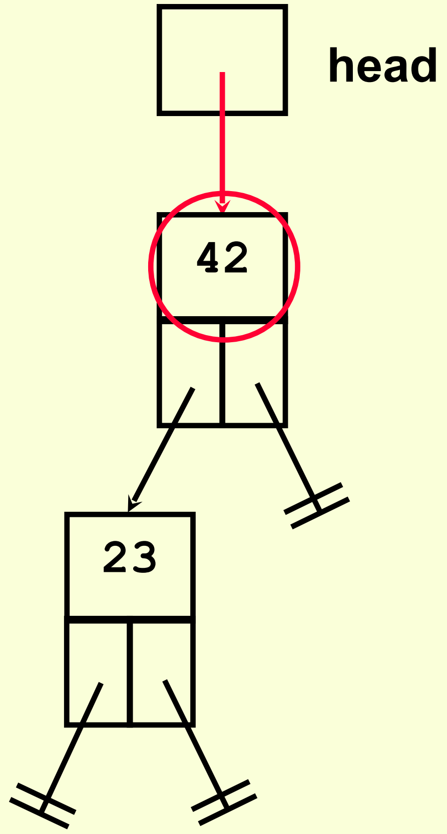


```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



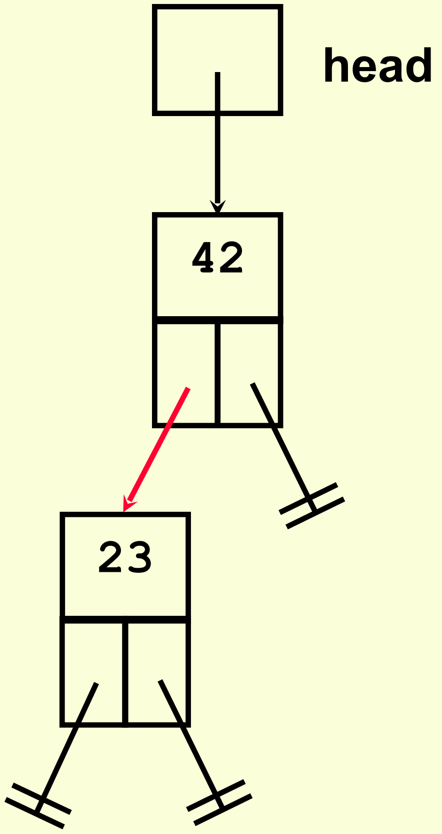
data_in = 35

```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



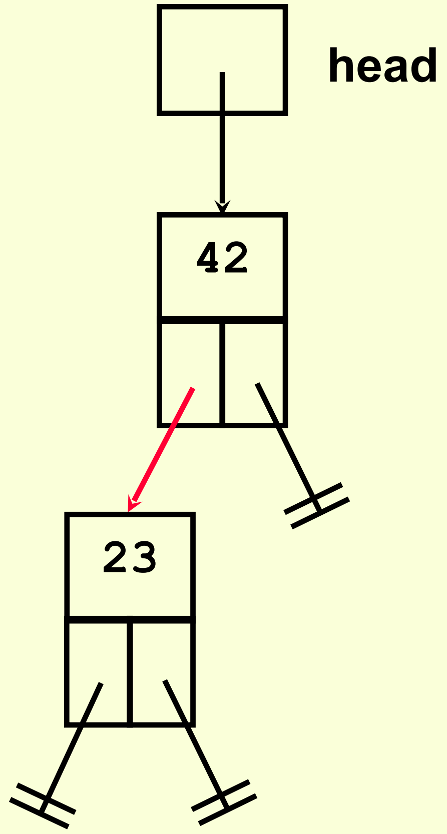
```
data_in = 35
```

```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



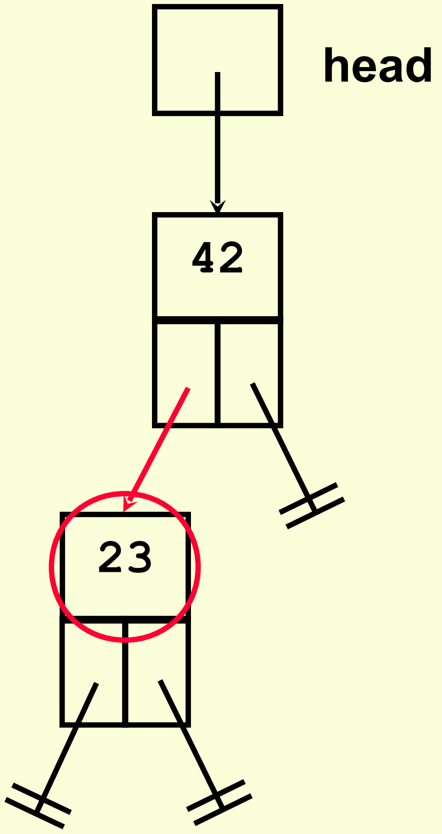
data_in = 35

```
procedure Insert(
  cur iot in/out Ptr toa Node,
  data_in iot in num)
  if(cur = NIL) then
    cur <- new(Node)
    cur^.data <- data_in
    cur^.left <- NIL
    cur^.right <- NIL
  elseif(cur^.data > data_in)
    Insert(cur^.left, data_in)
  else
    Insert(cur^.right, data_in)
  endif
endprocedure // Insert
```



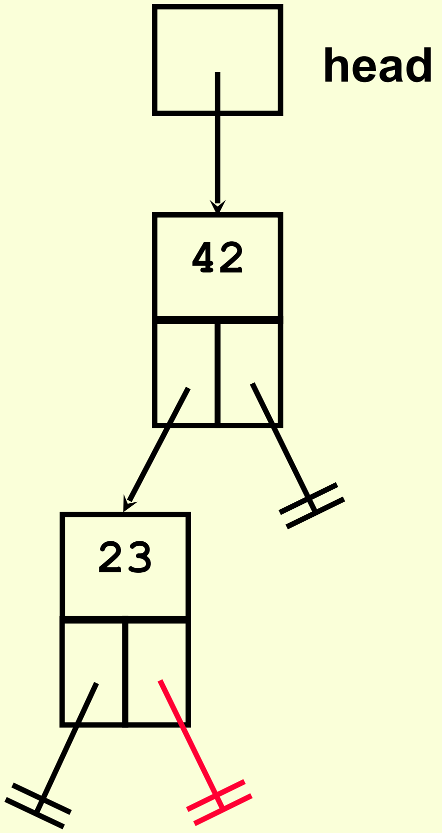
data_in = 35

```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



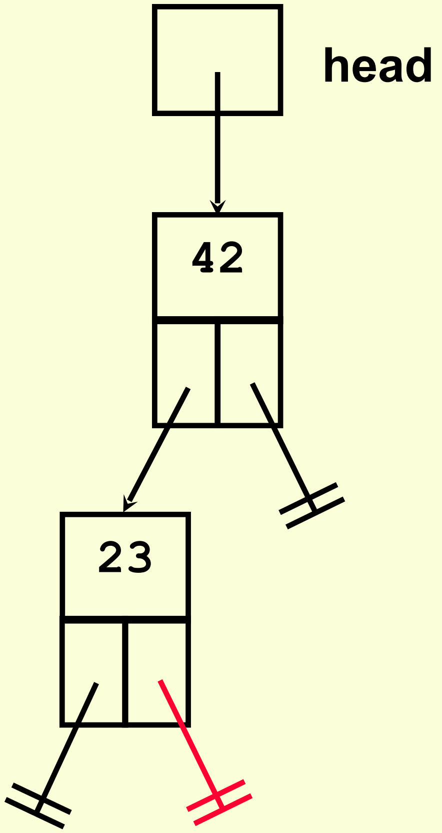
data_in = 35

```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



data_in = 35

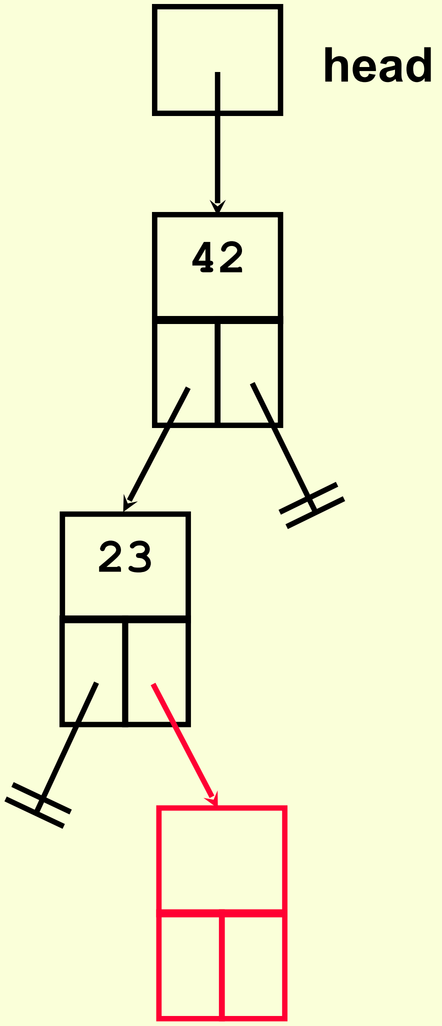
```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



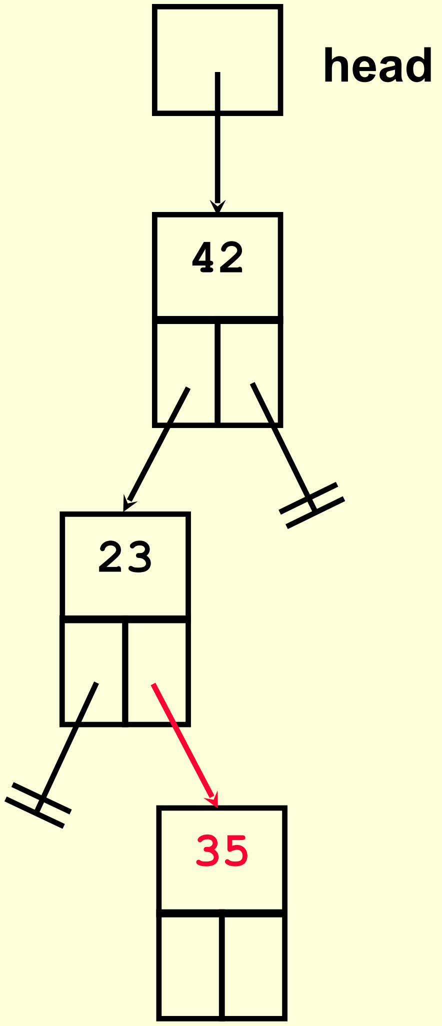
data_in = 35

```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```

data_in = 35

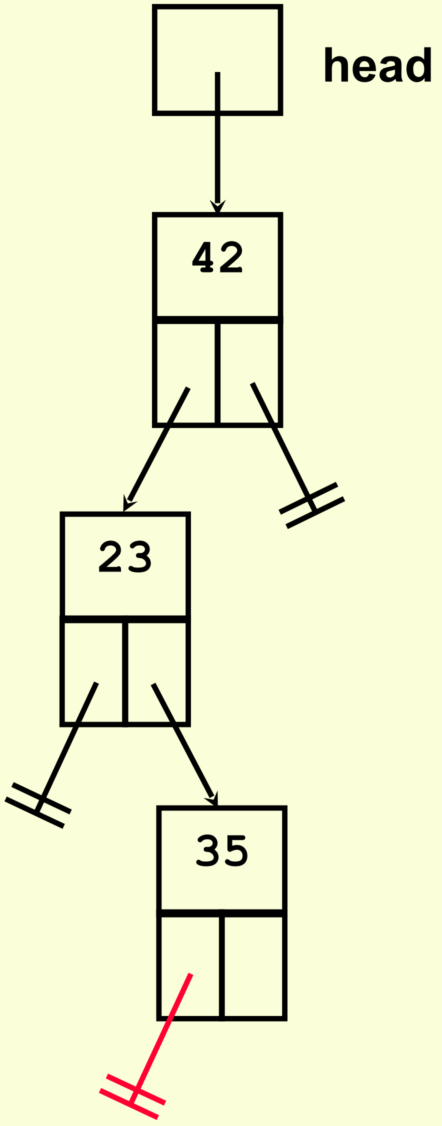


```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



data_in = 35

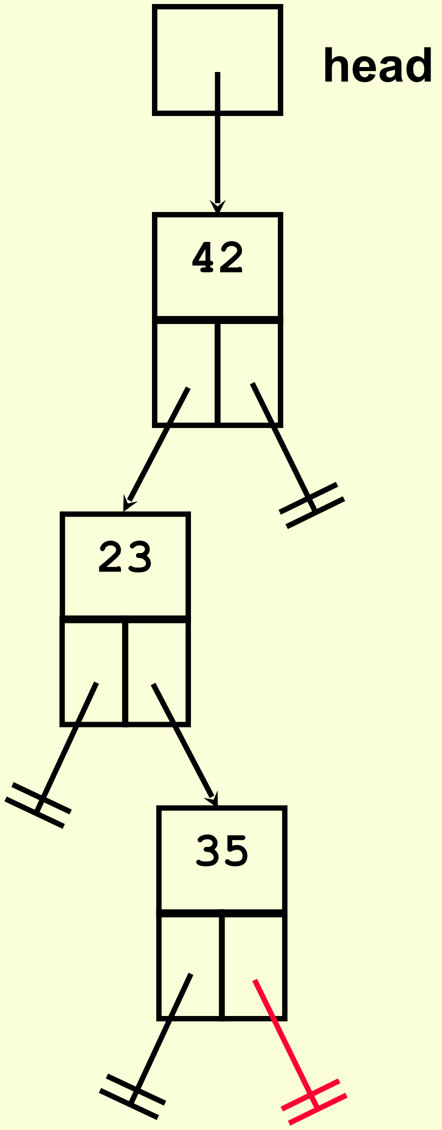
```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



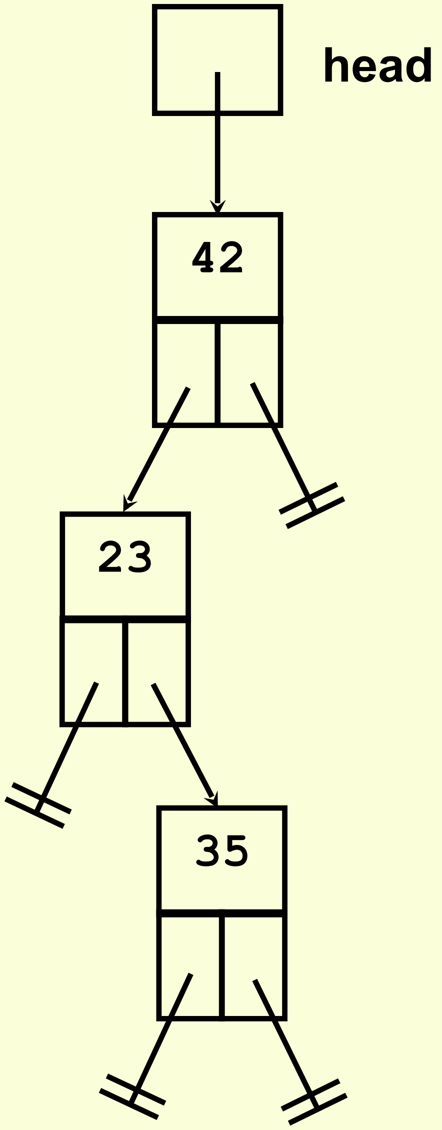
data_in = 35

```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```

data_in = 35

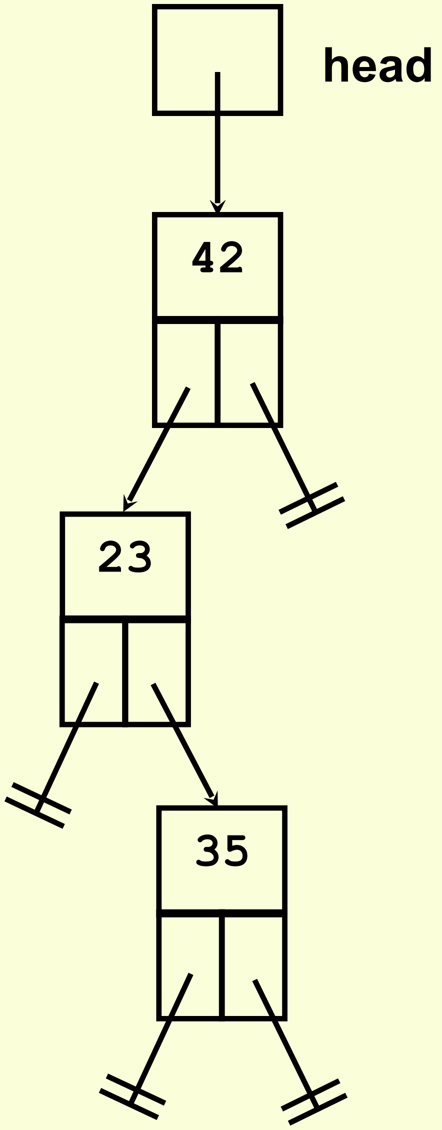


```
procedure Insert(
  cur iot in/out Ptr toa Node,
  data_in iot in num)
  if(cur = NIL) then
   cur <- new(Node)
   cur^.data <- data_in
   cur^.left <- NIL
   cur^.right <- NIL
  elseif(cur^.data > data_in)
   Insert(cur^.left, data_in)
  else
   Insert(cur^.right, data_in)
  endif
endprocedure // Insert
```



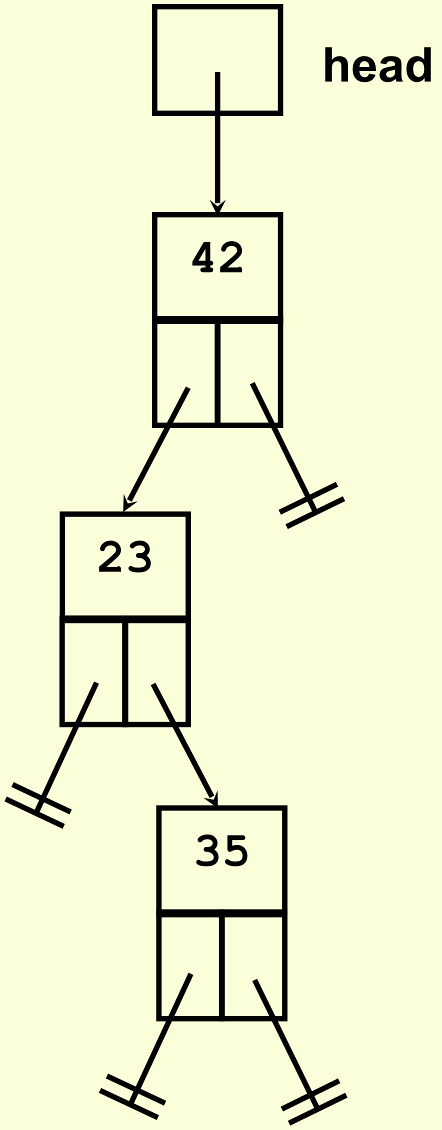
data_in = 35

```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



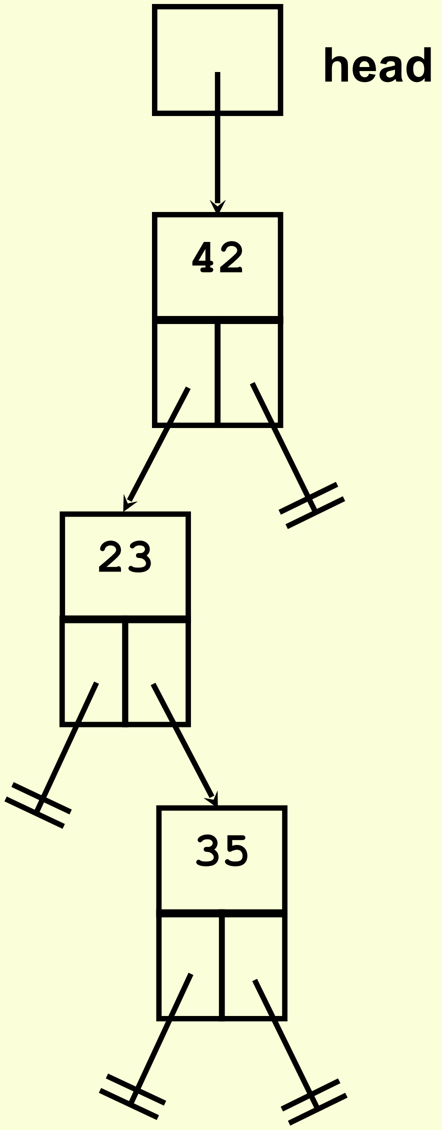
```
data_in = 35
```

```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



data_in = 35

```
.
.
Insert(head, 23)
Insert(head, 35)
Insert(head, 47)
.
.
```

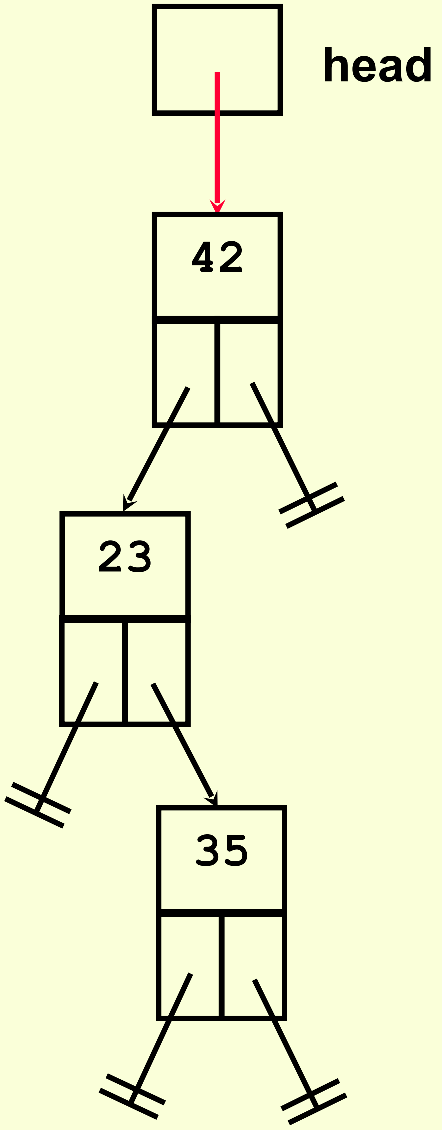


Continue?

yes...

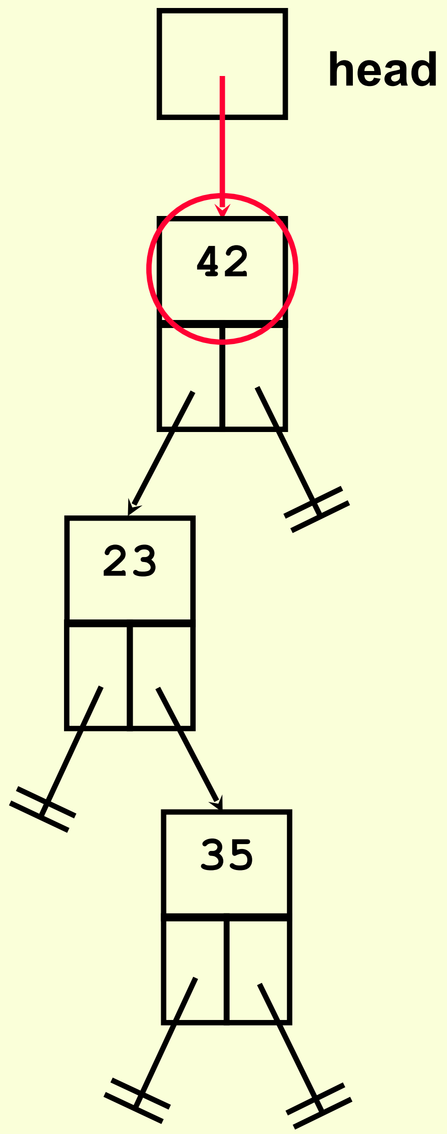
I’ve had enough!

```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



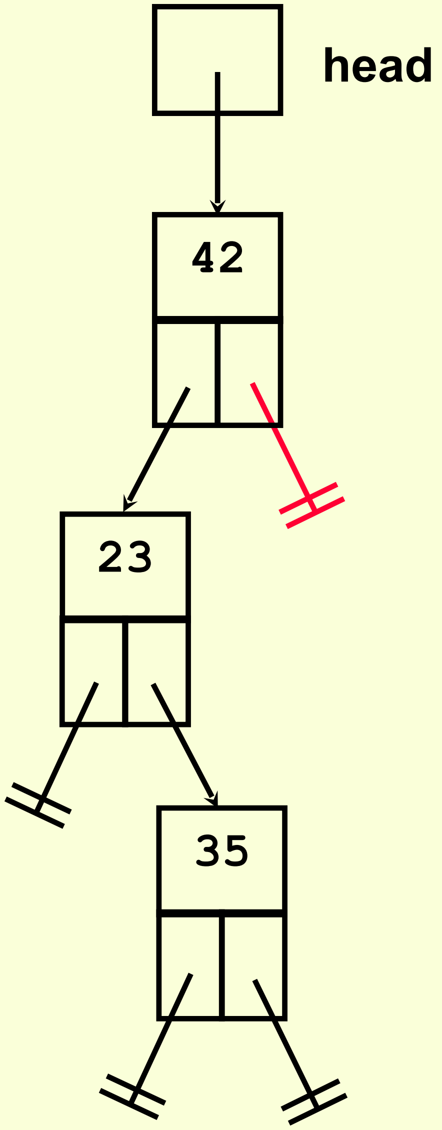
data_in = 47

```
procedure Insert(
  cur iot in/out Ptr toa Node,
  data_in iot in num)
  if(cur = NIL) then
    cur <- new(Node)
    cur^.data <- data_in
    cur^.left <- NIL
    cur^.right <- NIL
  elseif(cur^.data > data_in)
    Insert(cur^.left, data_in)
  else
    Insert(cur^.right, data_in)
  endif
endprocedure // Insert
```



data_in = 47

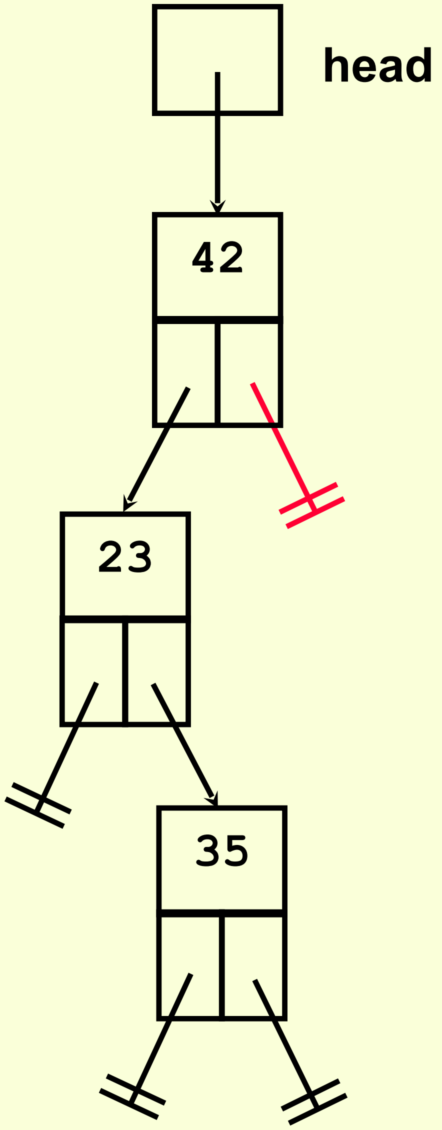
```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



data_in = 47

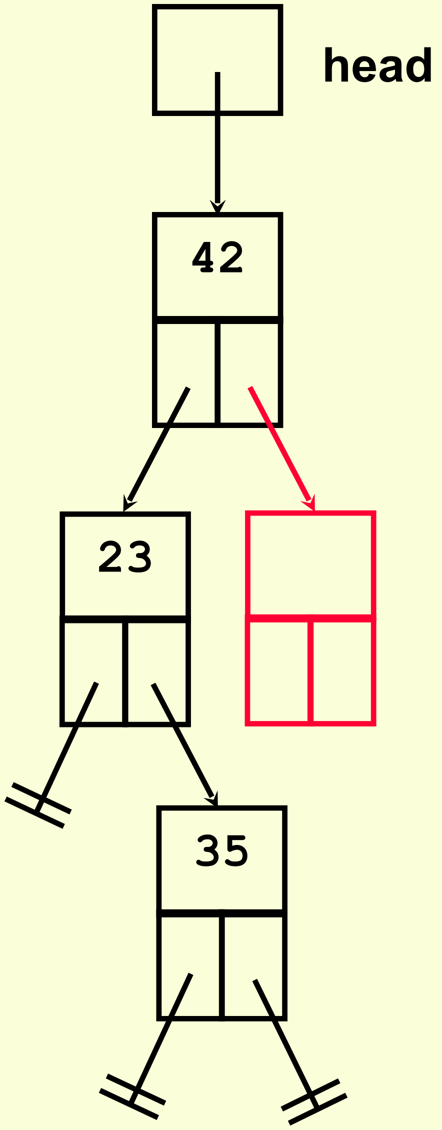
```
procedure Insert(
  cur iot in/out Ptr toa Node,
  data_in iot in num)
  if(cur = NIL) then
    cur <- new(Node)
    cur^.data <- data_in
    cur^.left <- NIL
    cur^.right <- NIL
  elseif(cur^.data > data_in)
    Insert(cur^.left, data_in)
  else
    Insert(cur^.right, data_in)
  endif
endprocedure // Insert
```

data_in = 47



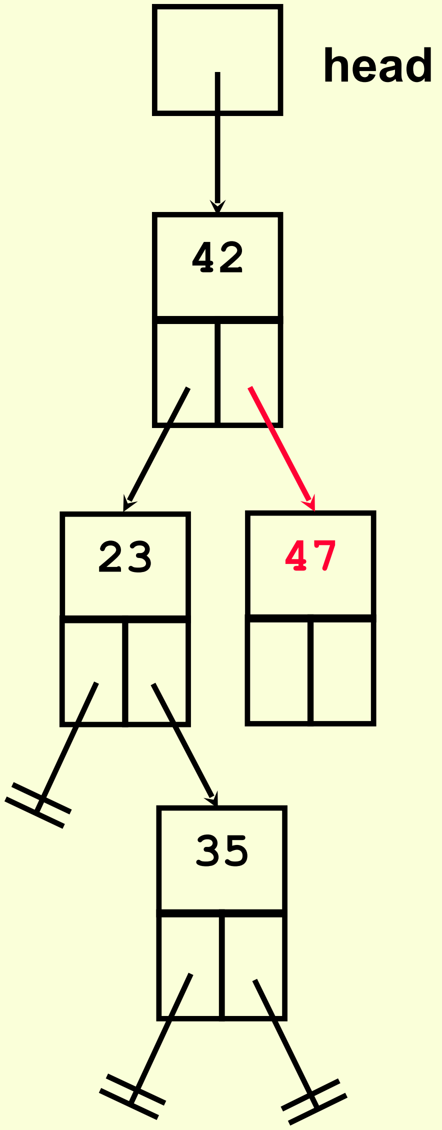
```
procedure Insert(
  cur iot in/out Ptr toa Node,
  data_in iot in num)
  if(cur = NIL) then
    cur <- new(Node)
    cur^.data <- data_in
    cur^.left <- NIL
    cur^.right <- NIL
  elseif(cur^.data > data_in)
    Insert(cur^.left, data_in)
  else
    Insert(cur^.right, data_in)
  endif
endprocedure // Insert
```

data_in = 47



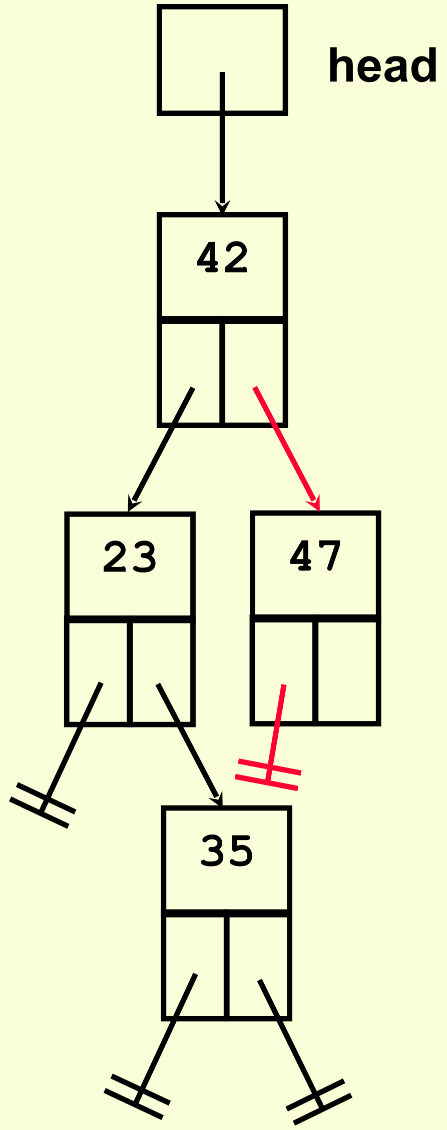
```
procedure Insert(
  cur iot in/out Ptr toa Node,
  data_in iot in num)
  if(cur = NIL) then
    cur <- new(Node)
    cur^.data <- data_in
    cur^.left <- NIL
    cur^.right <- NIL
  elseif(cur^.data > data_in)
    Insert(cur^.left, data_in)
  else
    Insert(cur^.right, data_in)
  endif
endprocedure // Insert
```

data_in = 47

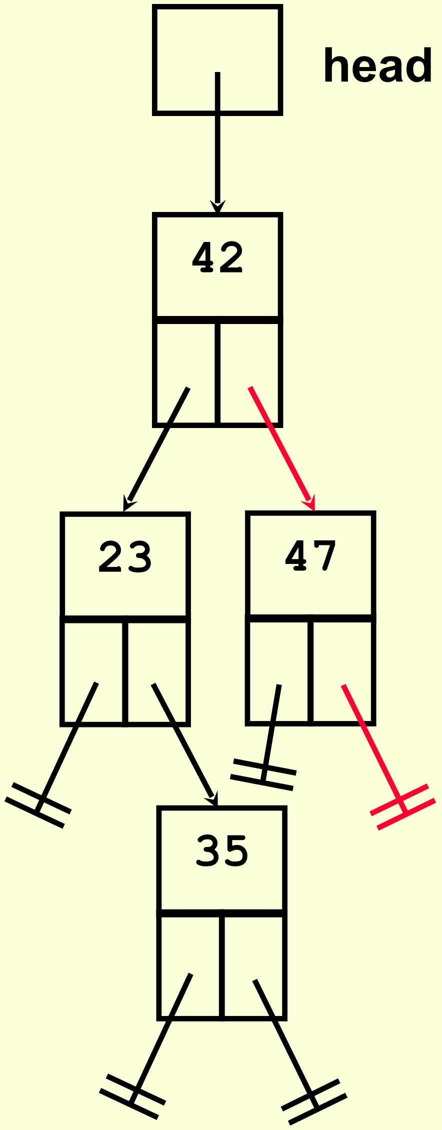


```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```

data_in = 47

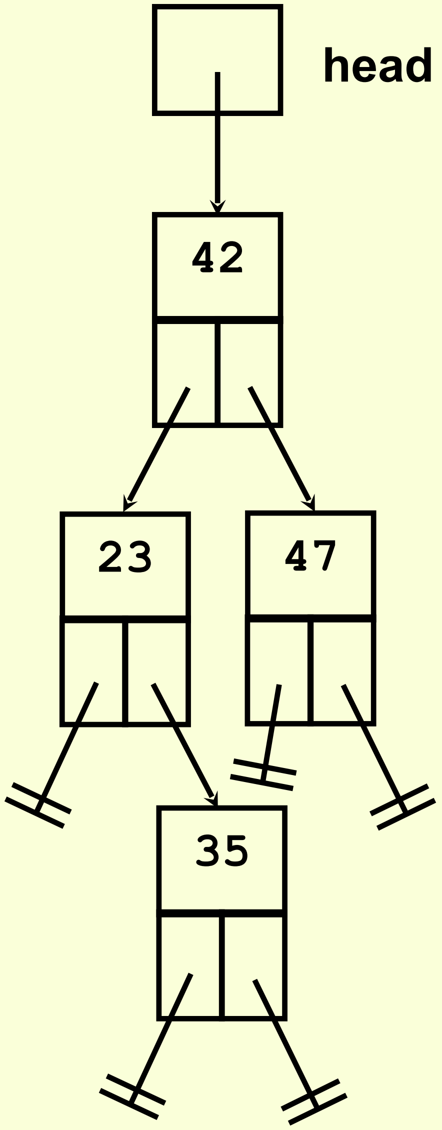


```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



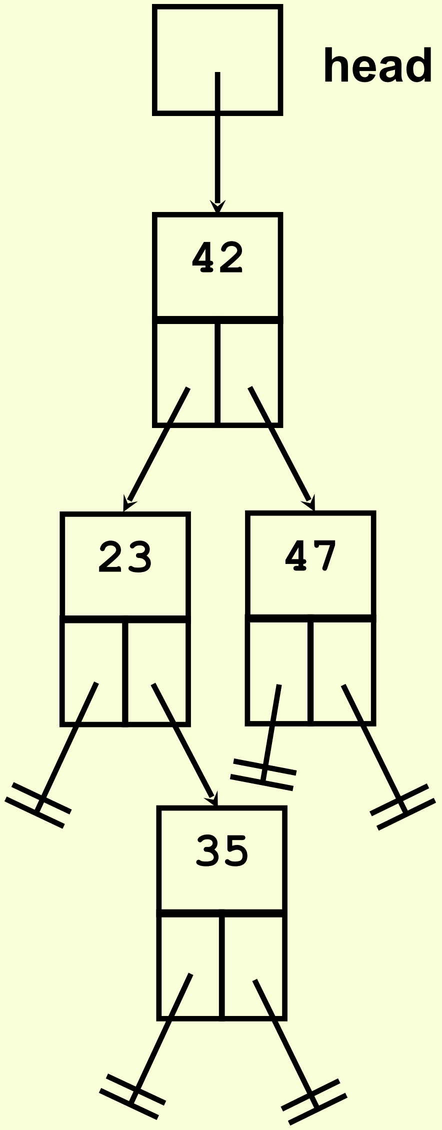
data_in = 47

```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



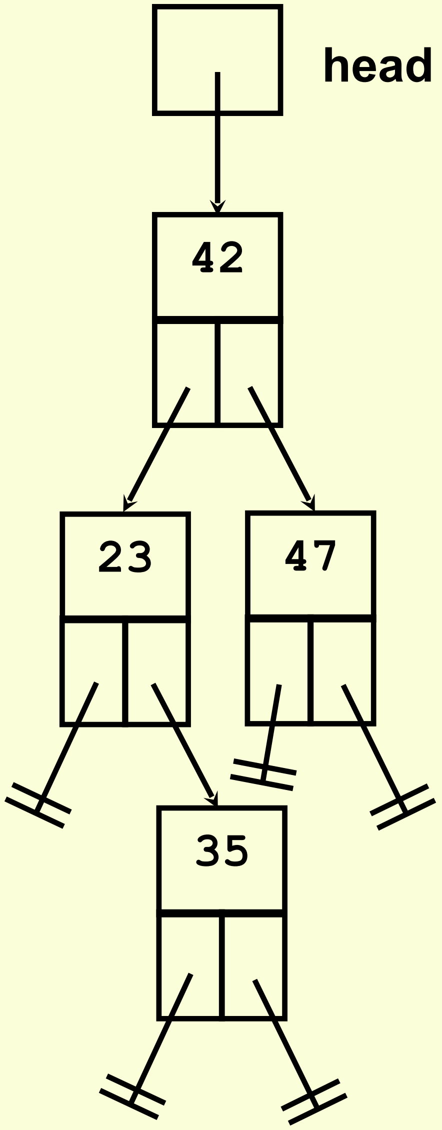
data_in = 47

```
procedure Insert(
 cur iot in/out Ptr toa Node,
 data_in iot in num)
 if(cur = NIL) then
  cur <- new(Node)
  cur^.data <- data_in
  cur^.left <- NIL
  cur^.right <- NIL
 elseif(cur^.data > data_in)
  Insert(cur^.left, data_in)
 else
  Insert(cur^.right, data_in)
 endif
endprocedure // Insert
```



data_in = 47

```
.
.
Insert(head, 23)
Insert(head, 35)
Insert(head, 47)
.
.
```



Summary

- Preserve “search” structure!
- Inserting involves 2 steps:
  - Find the correct location
    - For a BST insert, always insert at the “bottom” of the tree
  - Do commands to add node
    - Create node
    - Add data
    - Make left and right pointers point to nil

Questions?

Deleting from a Binary Search Tree

(BST)

The Scenario

- We have a Binary Search Tree and want to remove some element based upon a match.

- Must preserve “search” property
- Must not lose any elements (i.e. only remove the one element)

BST Deletion

- Search for desired item.
- If not found, then return NIL or print error.
- If found, perform steps necessary to accomplish removal from the tree.

Four Cases for Deletion

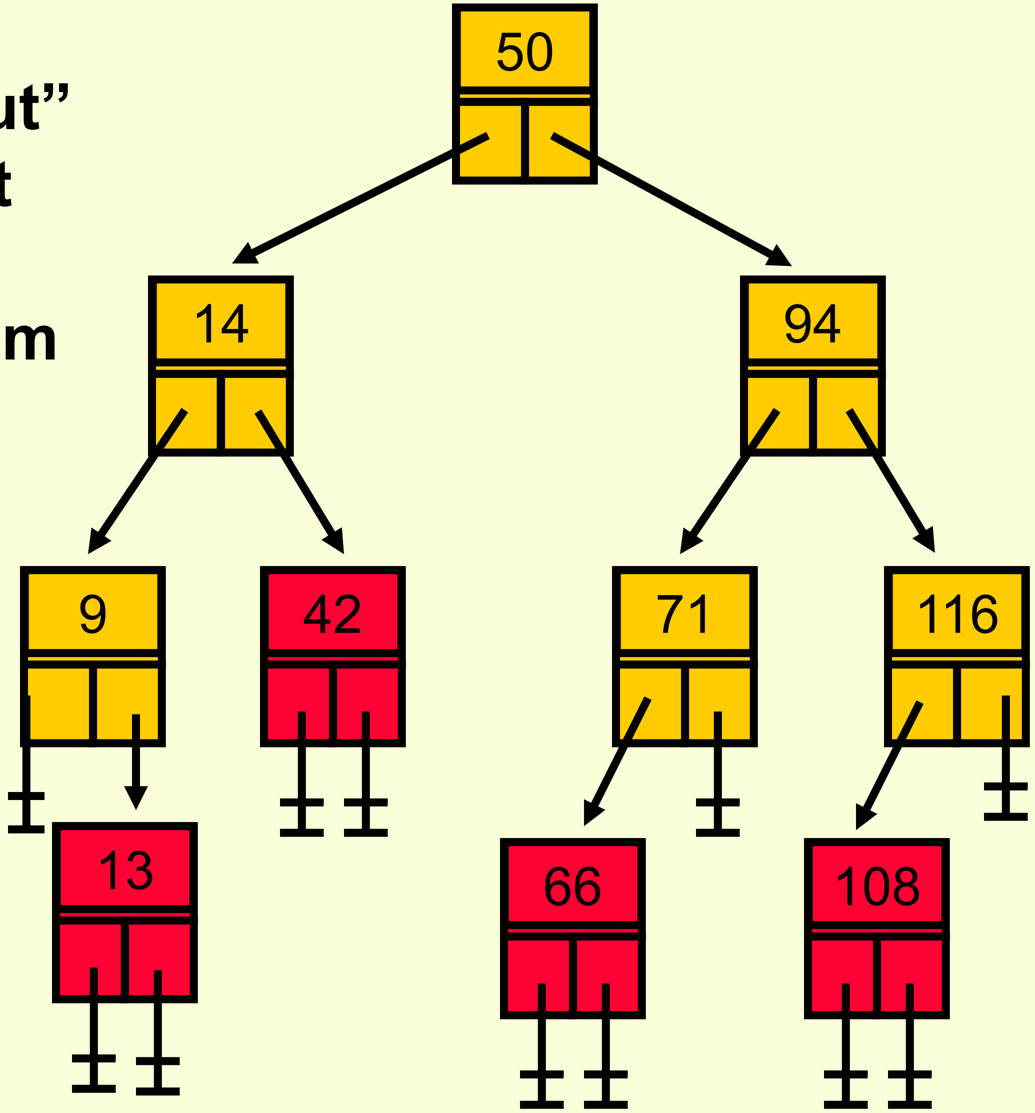
- Delete a leaf node
- Delete a node with only one child (left)
- Delete a node with only one child (right)
- Delete a node with two children

Cases 2 and 3 are comparable and only need slight changes in the conditional statement used

Delete a Leaf Node

Simply use an "in/out" pointer and assign it to "nil". This will remove the node from the tree.

```
cur <- nil
```

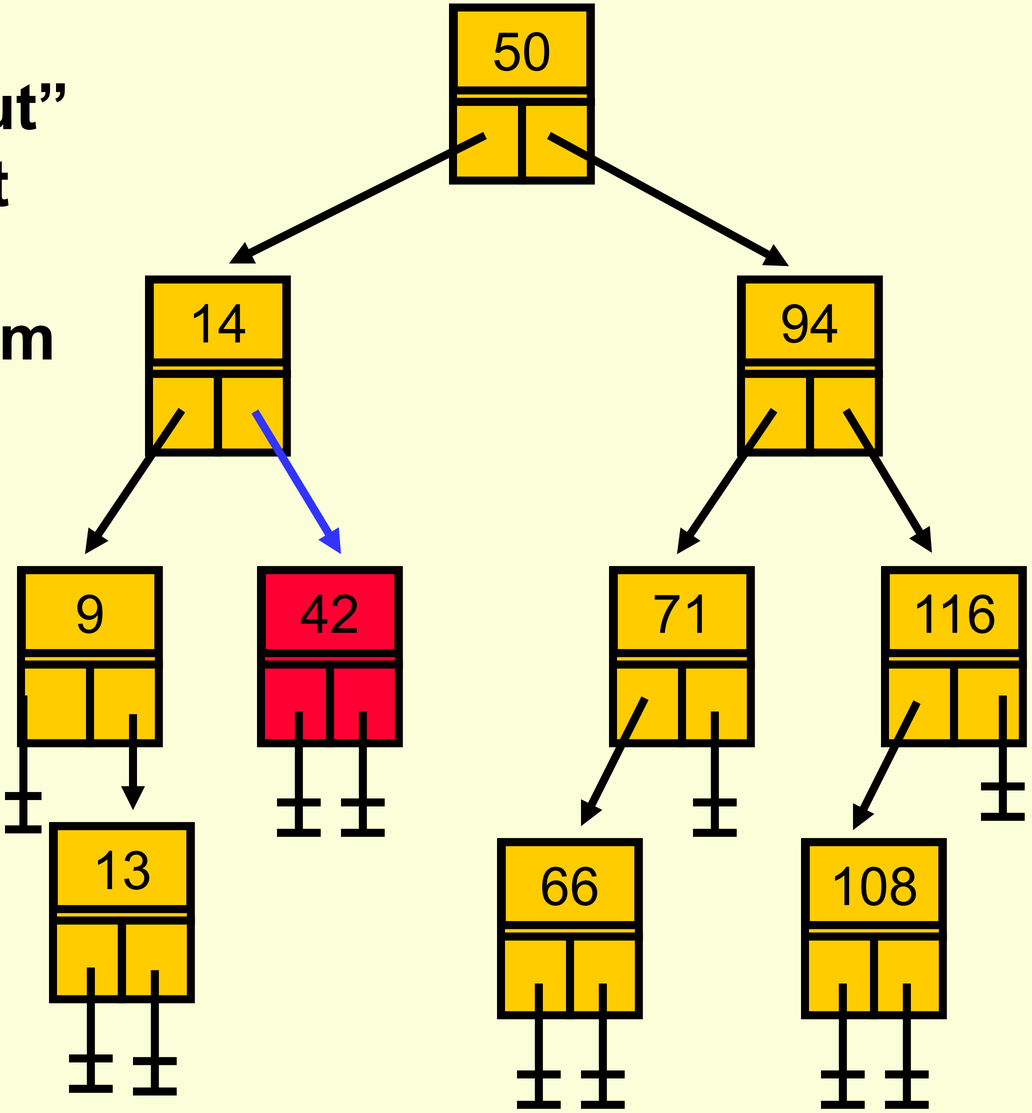


Delete a Leaf Node

Simply use an "in/out" pointer and assign it to "nil". This will remove the node from the tree.

```
cur <- nil
```

Let's delete 42.

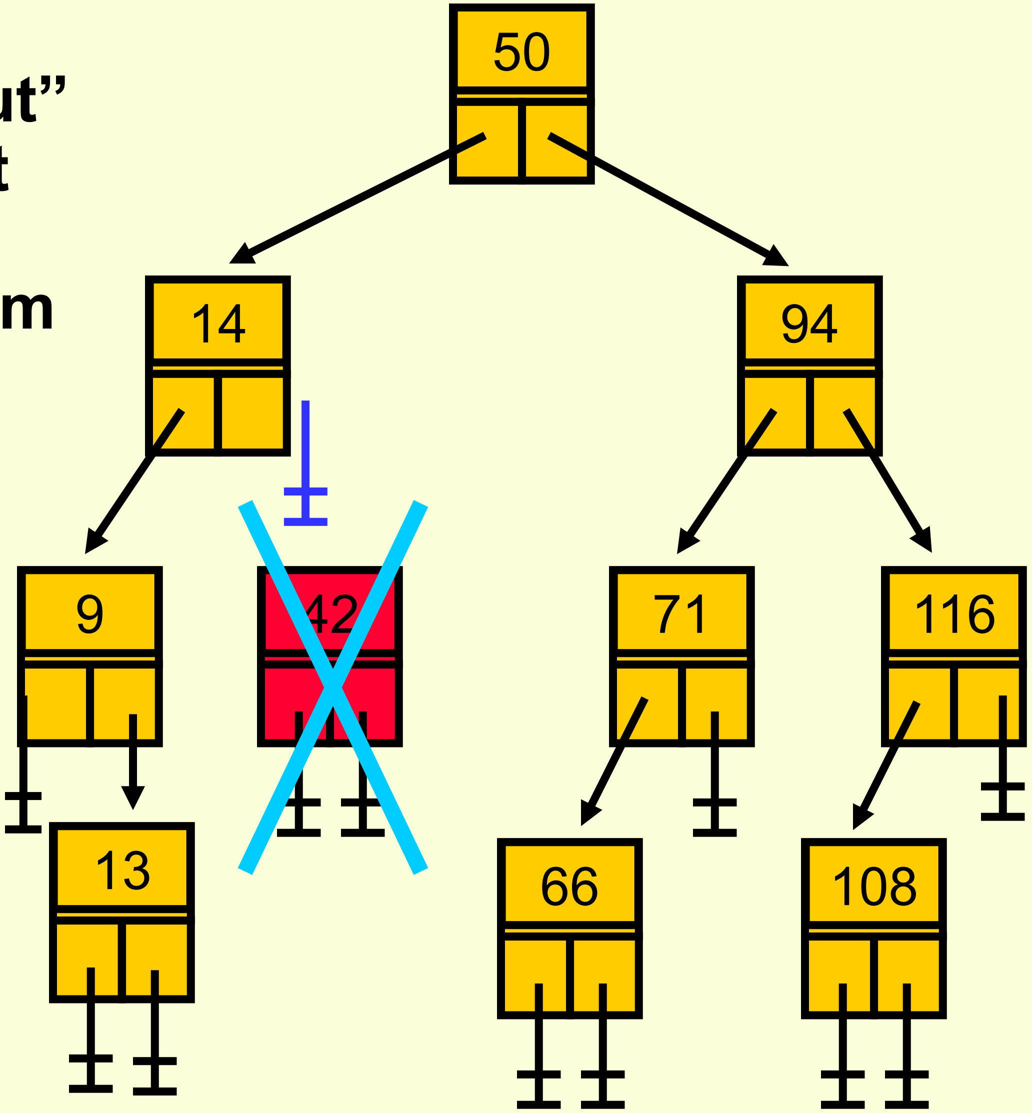


Delete a Leaf Node

Simply use an “in/out” pointer and assign it to “nil”. This will remove the node from the tree.

```
cur <- nil
```

Move the pointer; now nothing points to the node.

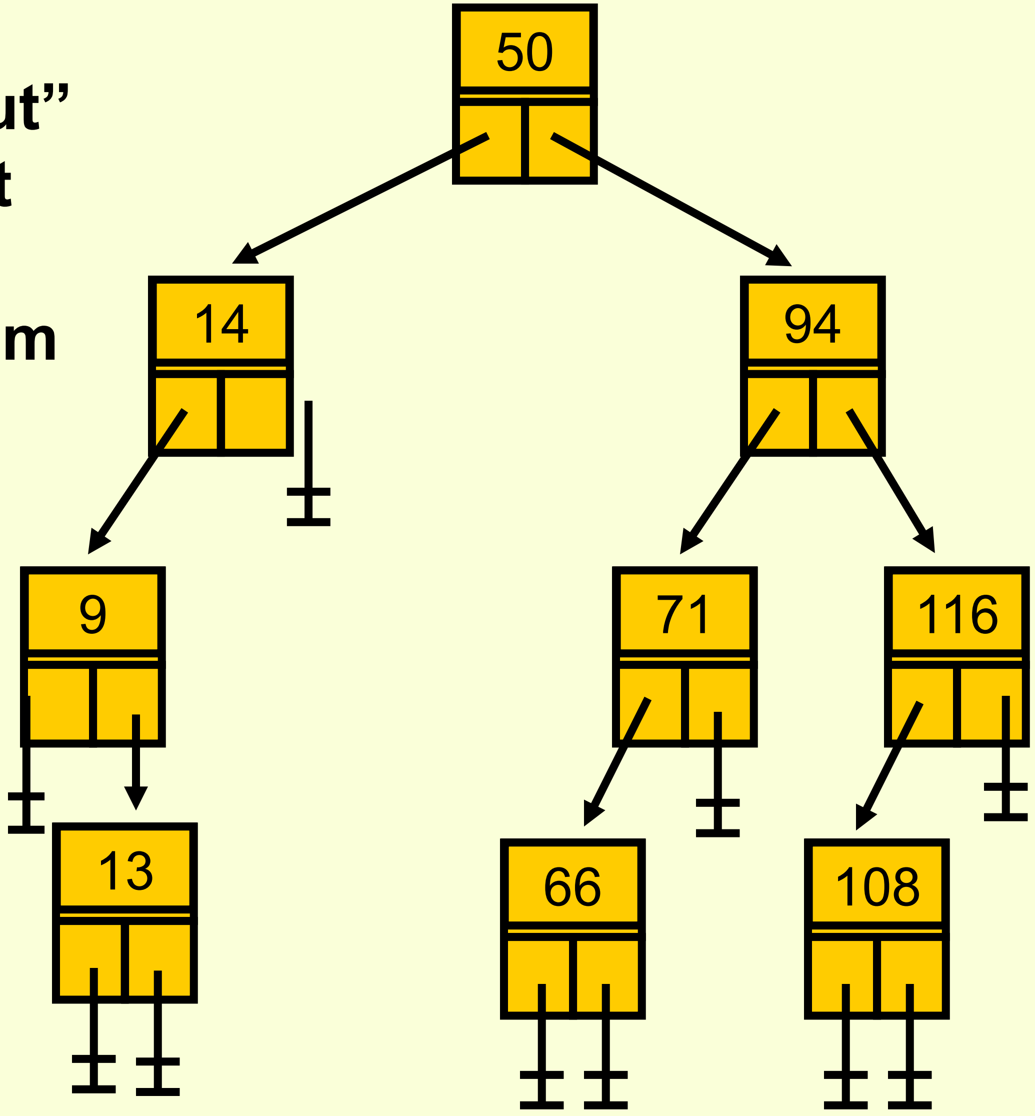


Delete a Leaf Node

Simply use an "in/out" pointer and assign it to "nil". This will remove the node from the tree.

```
cur <- nil
```

The resulting tree.



Delete a Node with One Child

Use an “in/out” pointer.

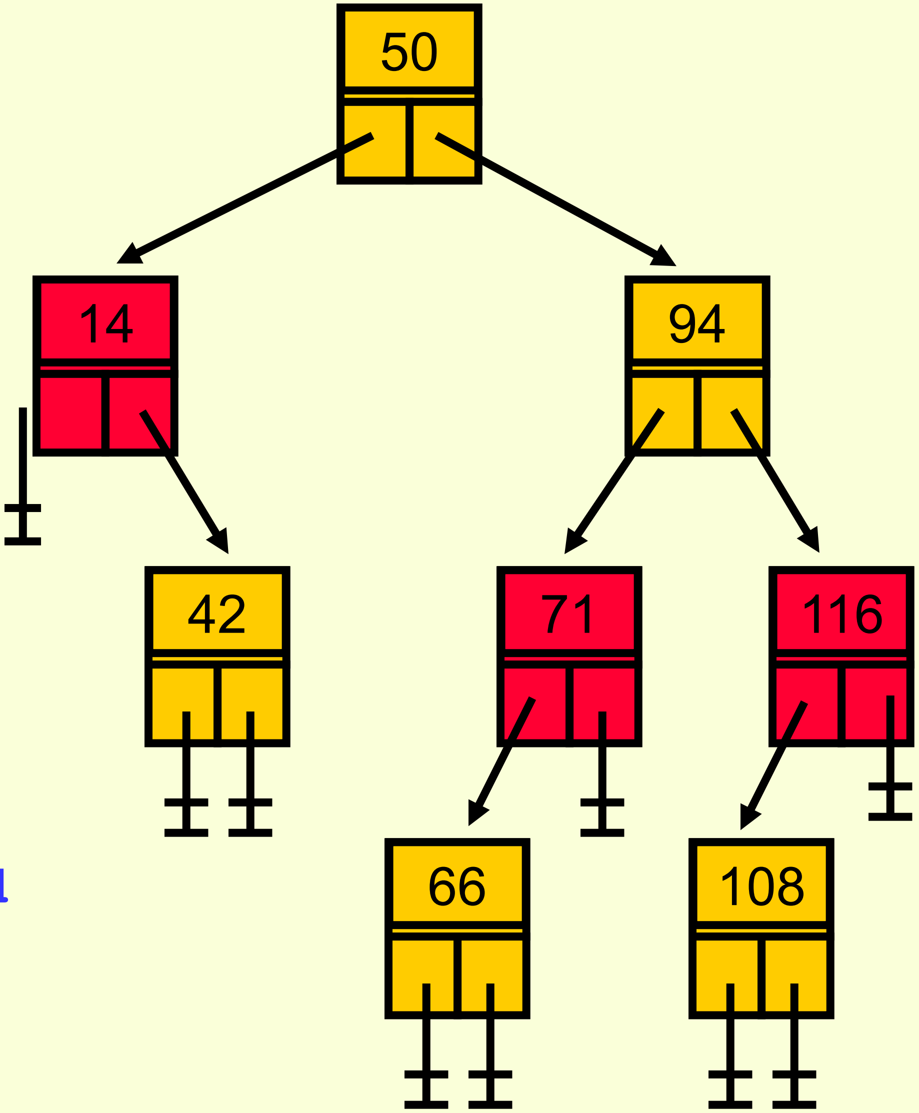
Determine if it has a left or a right child.

Point the current pointer to the appropriate child:

```
cur <- cur^.left_child
```

or

```
cur <- cur^.right_child
```



Delete a Node with One Child

Use an "in/out" pointer.

Determine if it has a left or a right child.

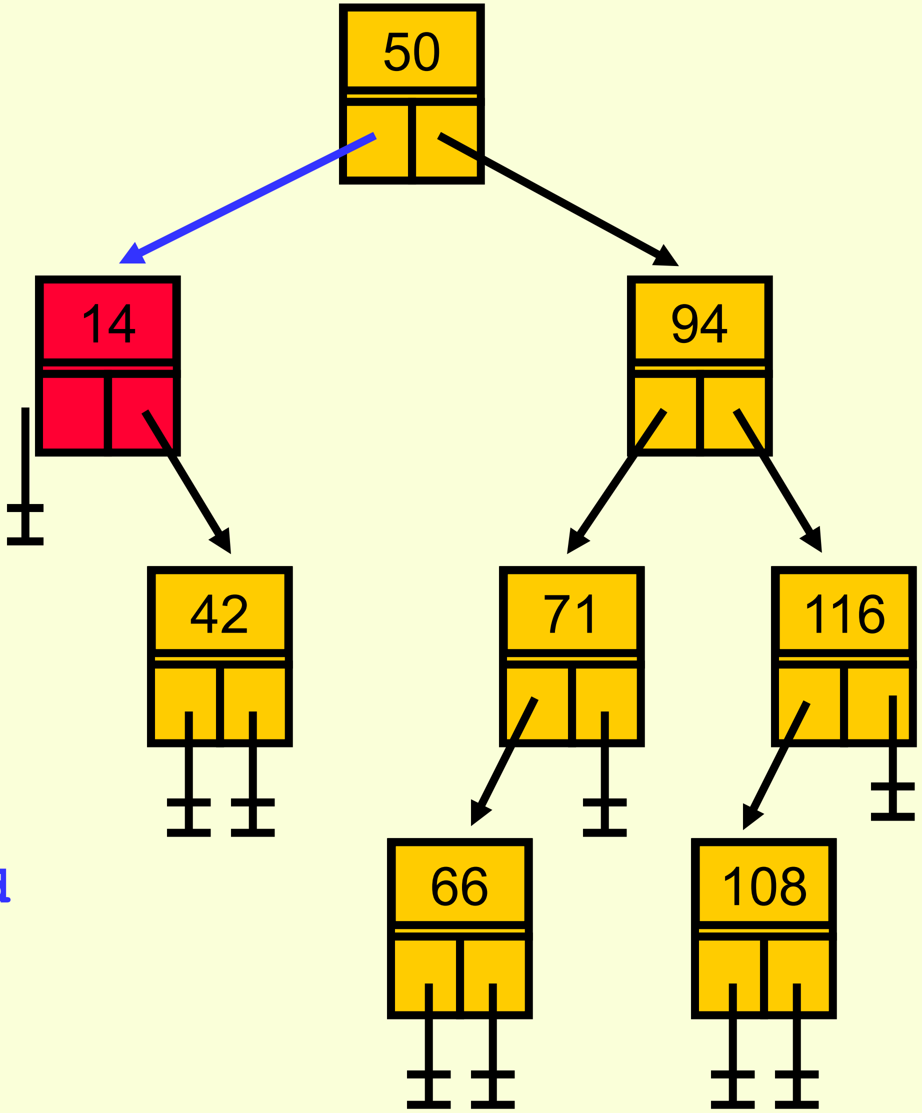
Point the current pointer to the appropriate child:

```
cur <- cur^.left_child
```

or

```
cur <- cur^.right_child
```

Let's delete 14.



Delete a Node with One Child

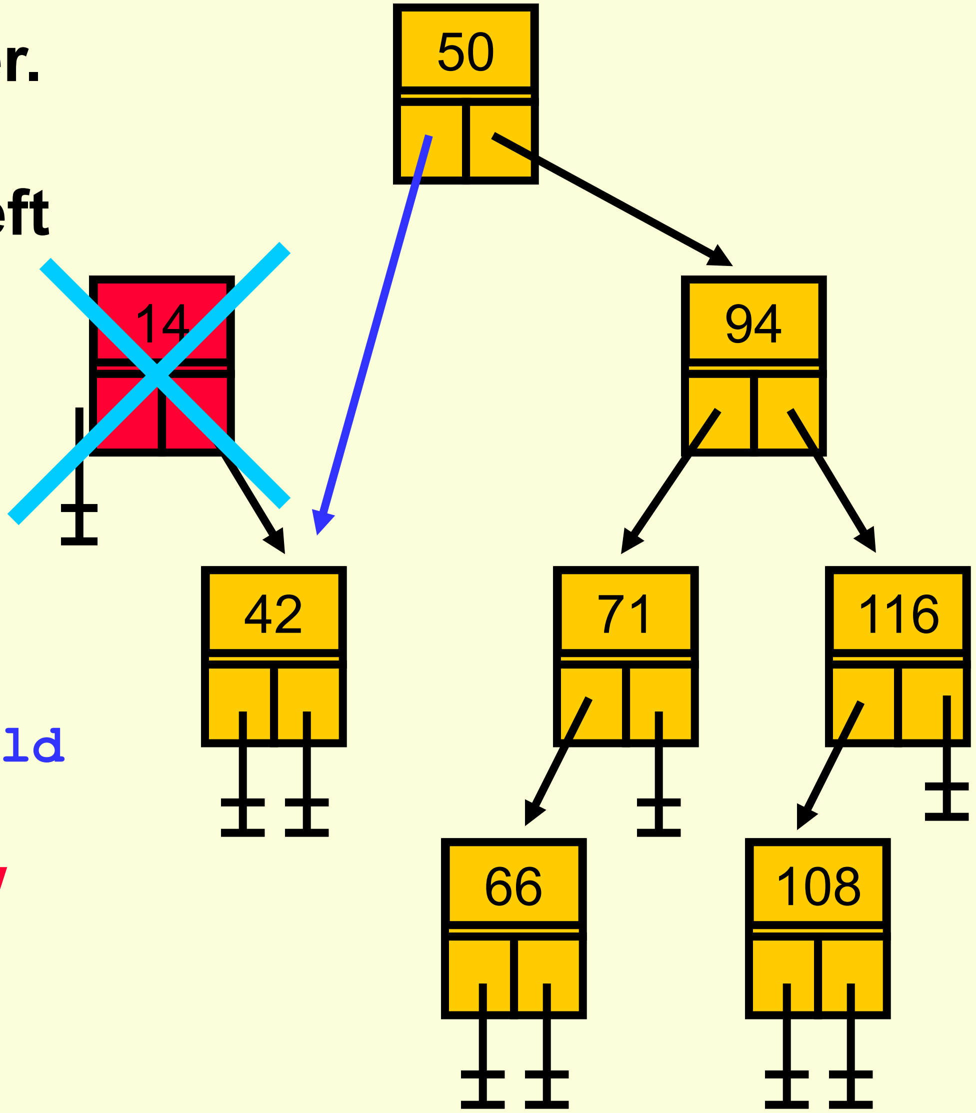
Use an “in/out” pointer.

Determine if it has a left or a right child.

Point the current pointer to the appropriate child:

```
cur <- cur^.right_child
```

Move the pointer; now nothing points to the node.



Delete a Node with One Child

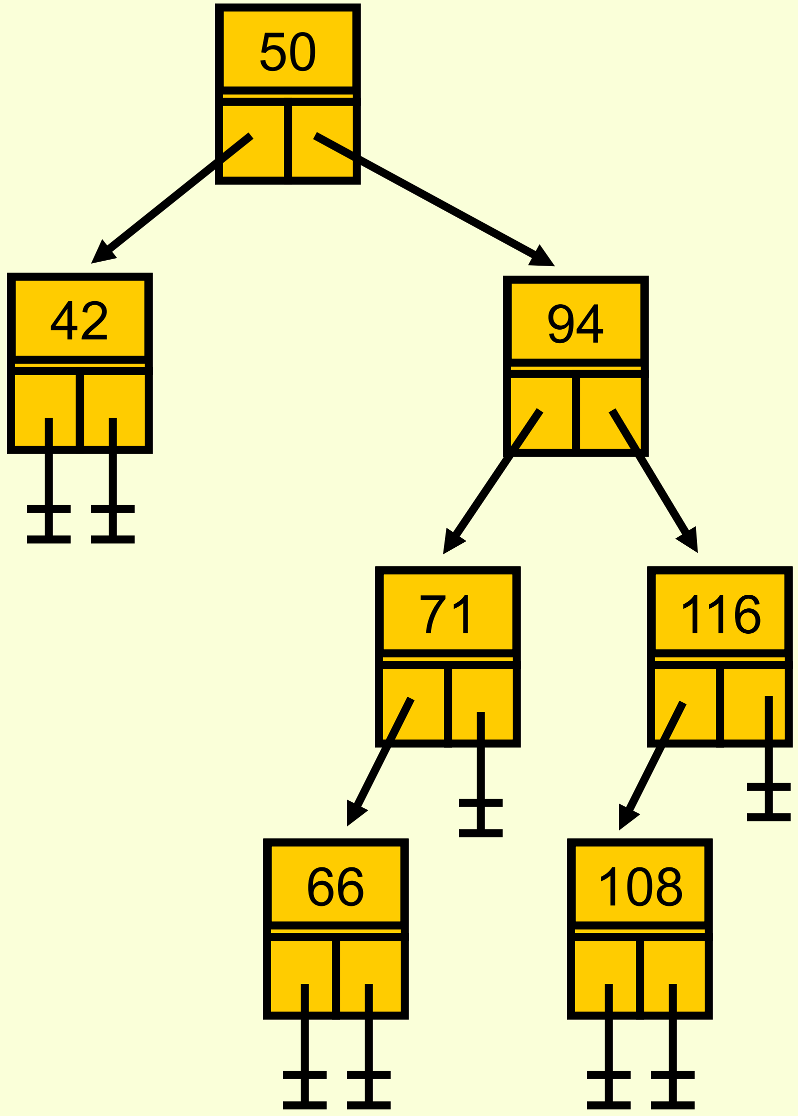
Use an “in/out” pointer.

Determine if it has a left or a right child.

Point the current pointer to the appropriate child:

```
cur <- cur^.right_child
```

The resulting tree.



Delete a Node with One Child

Use an “in/out” pointer.

Determine if it has a left or a right child.

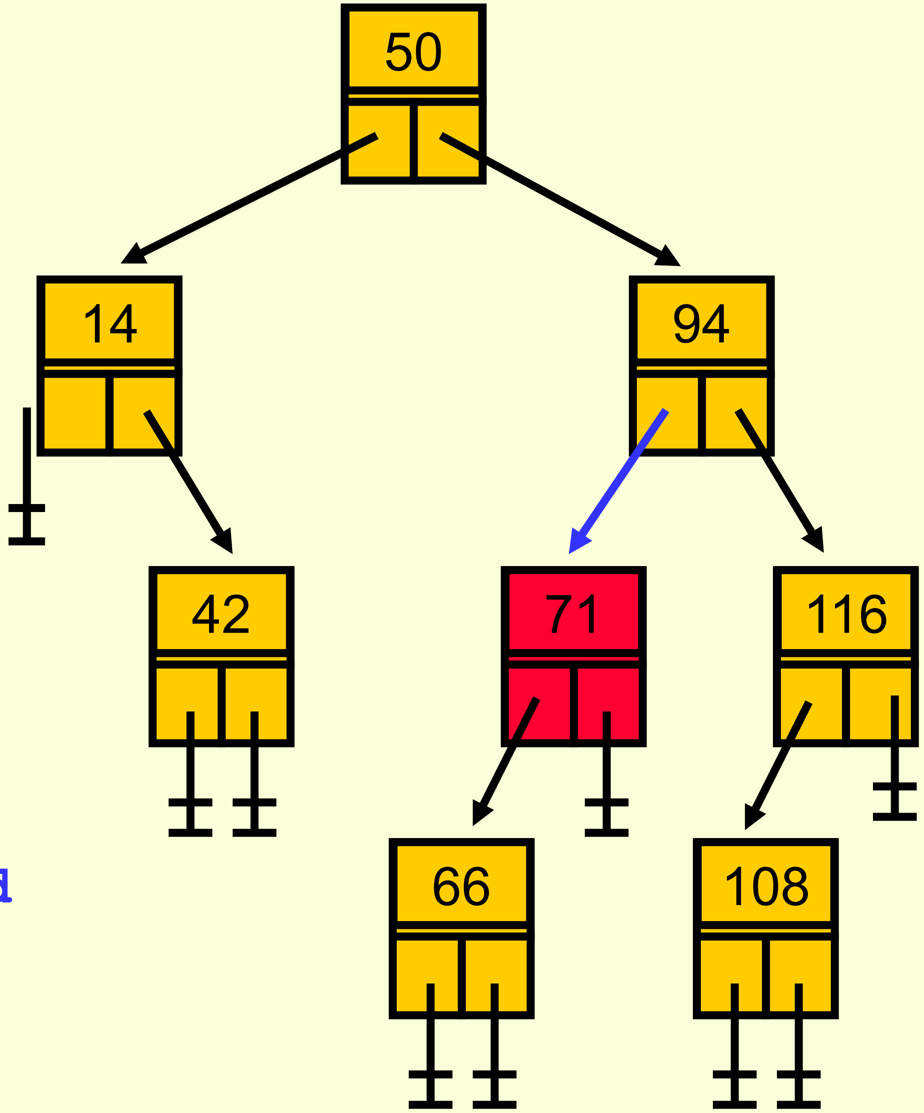
Point the current pointer to the appropriate child:

```
cur <- cur^.left_child
```

or

```
cur <- cur^.right_child
```

Let’s delete 71.



Delete a Node with One Child

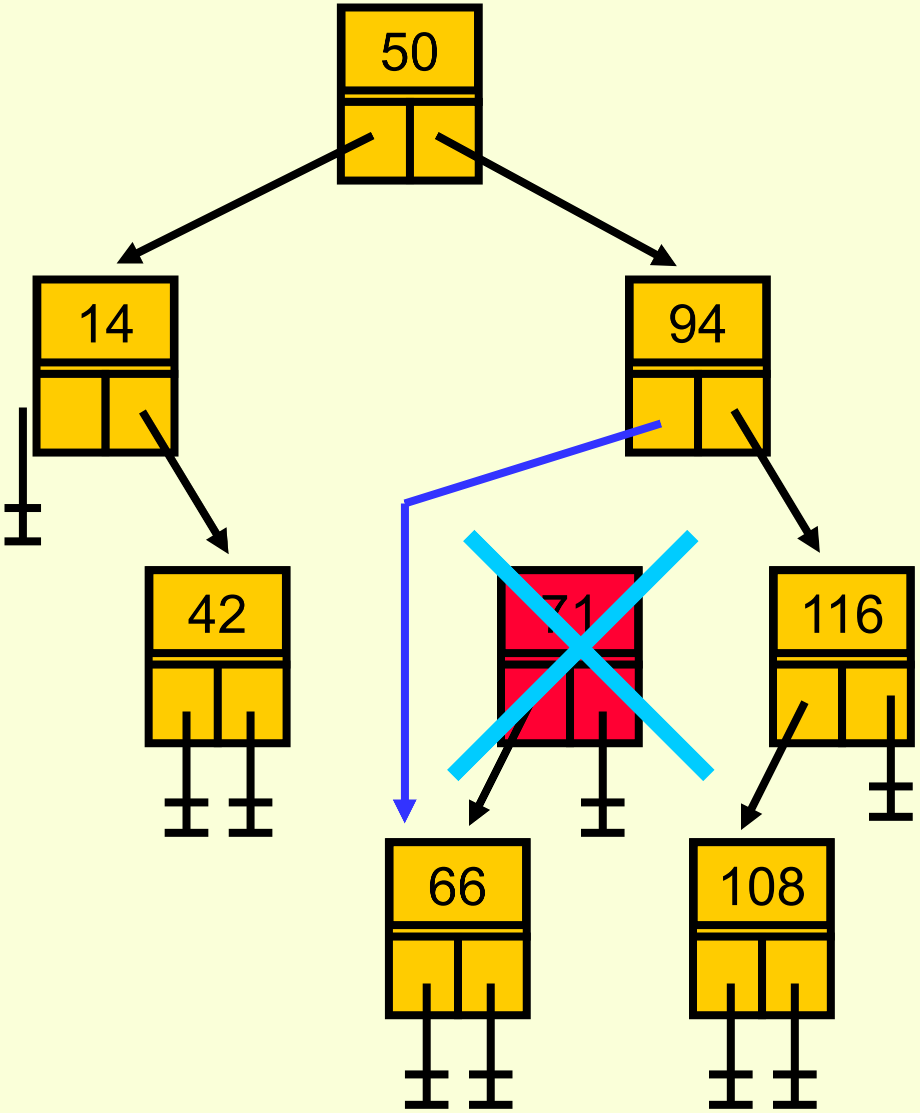
Use an “in/out” pointer.

Determine if it has a left or a right child.

Point the current pointer to the appropriate child:

```
cur <- cur^.left_child
```

Move the pointer; now nothing points to the node.



Delete a Node with One Child

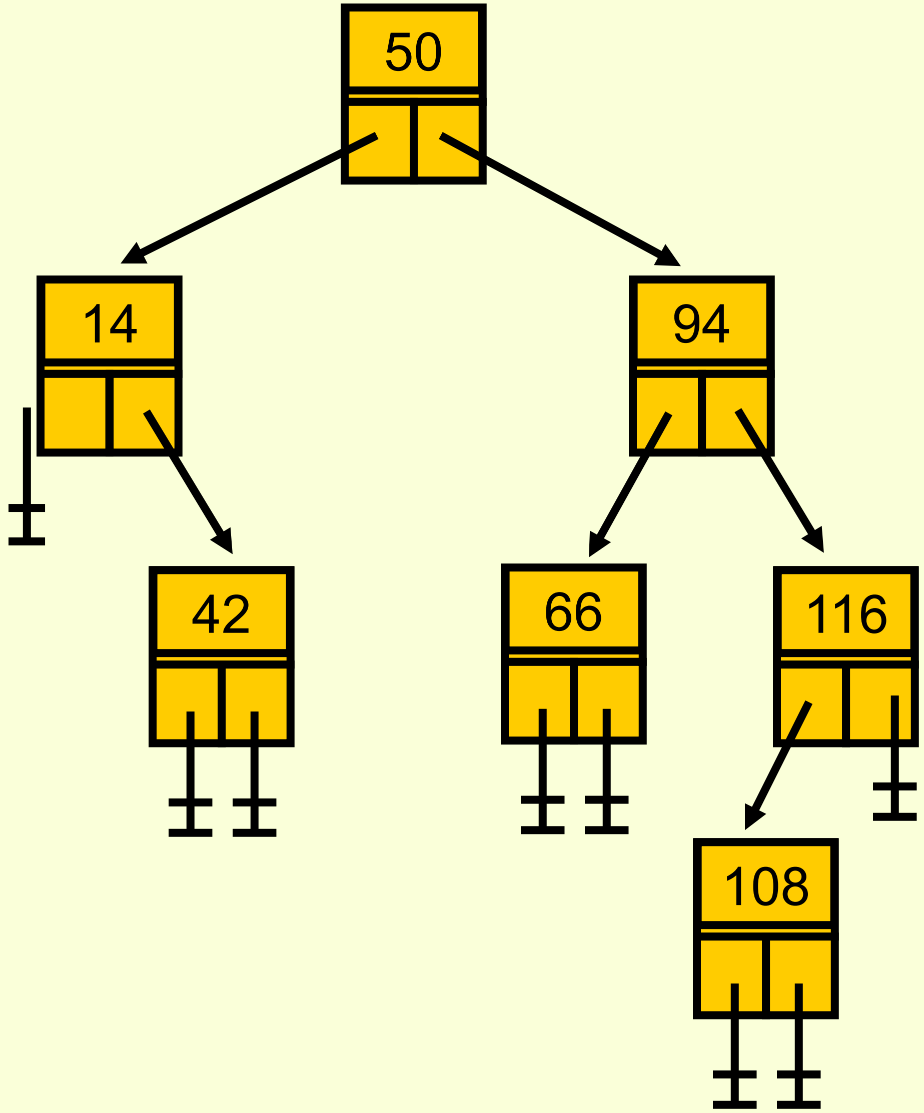
Use an "in/out" pointer.

Determine if it has a left or a right child.

Point the current pointer to the appropriate child:

```
cur <- cur^.left_child
```

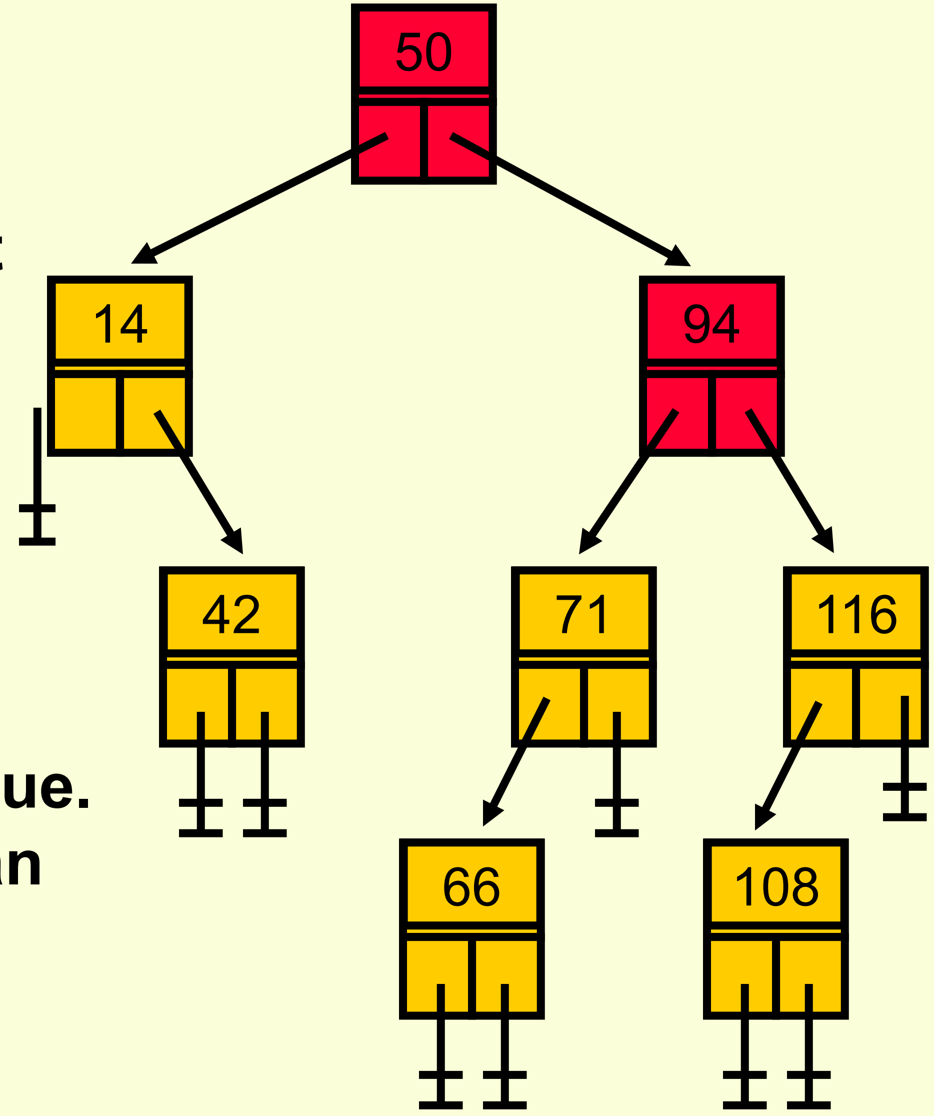
The resulting tree.



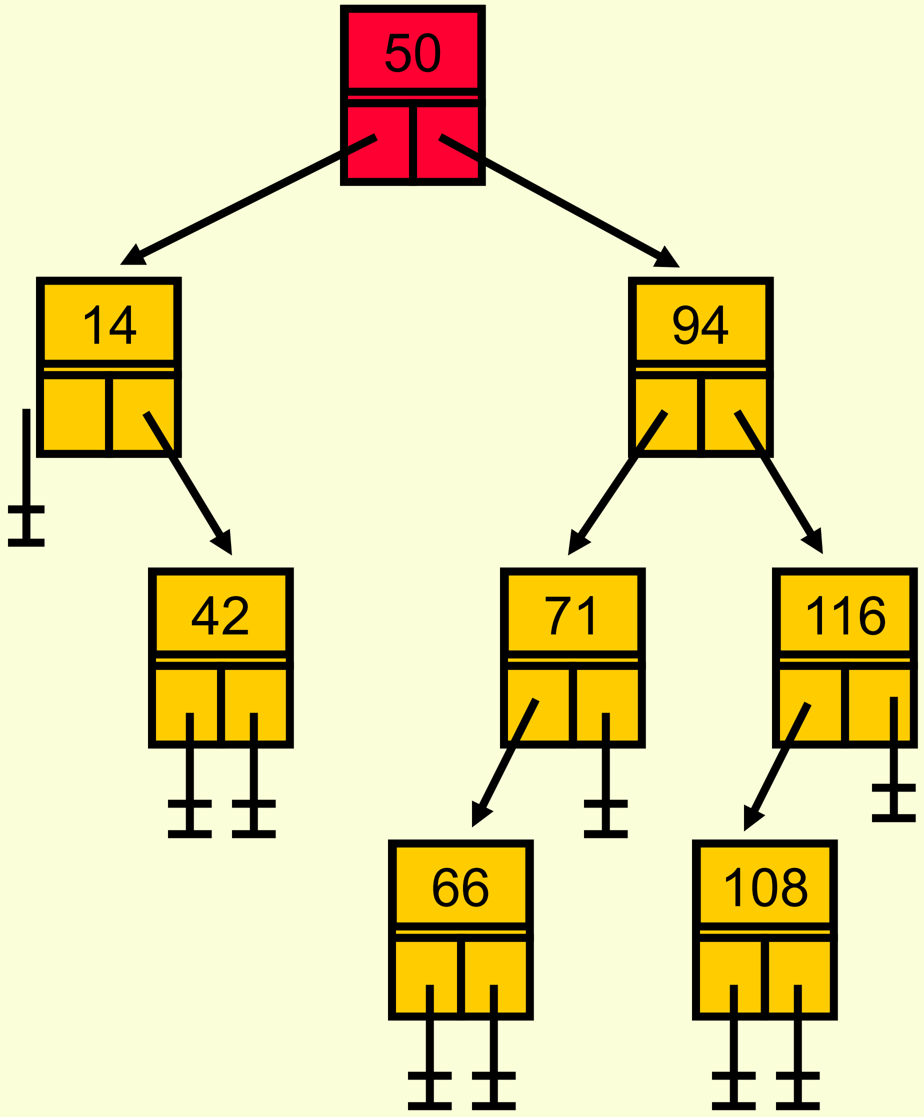
Delete a Node with Two Children

Copy a replacement value from a descendant node.
- Largest from left
- Smallest from right

Then delete that descendant node to remove the duplicate value.
- We know this will be an easier case.

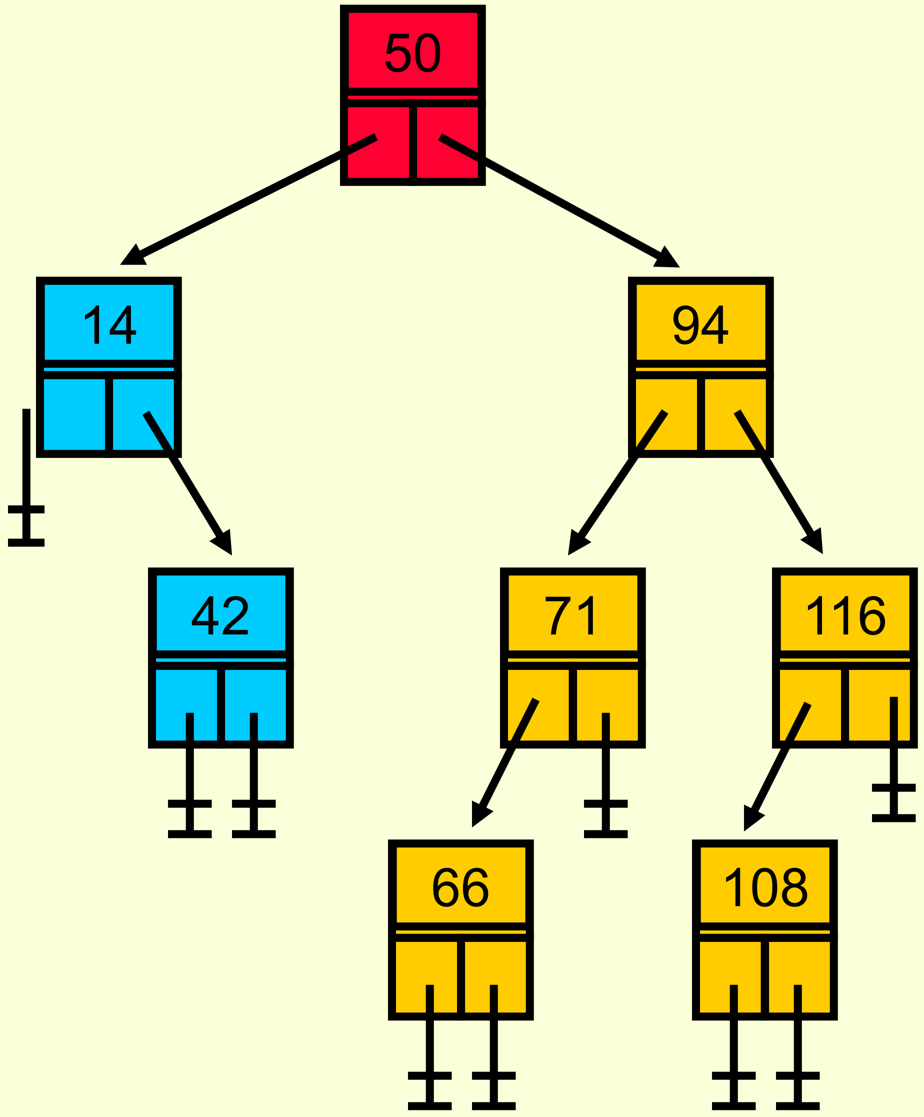


Delete a Node with Two Children



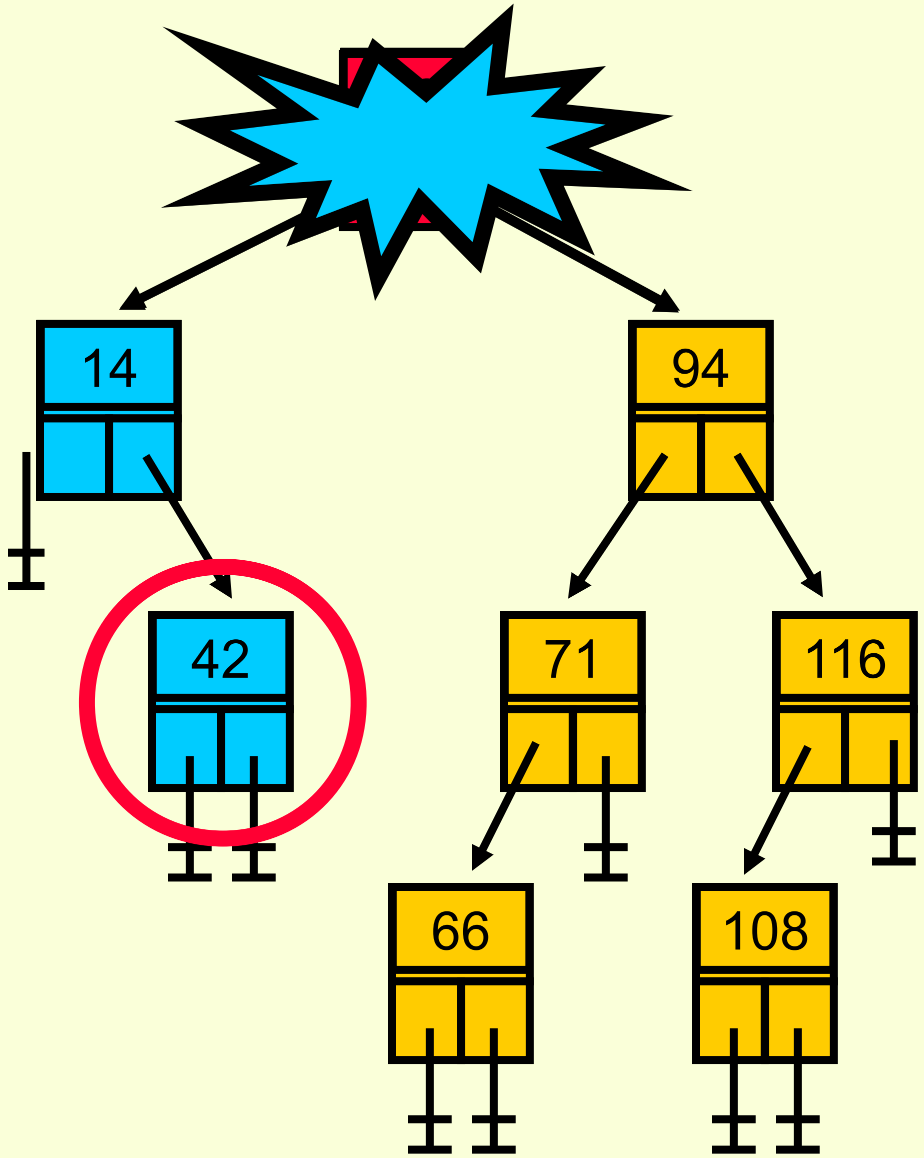
Let's delete 50.

Delete a Node with Two Children



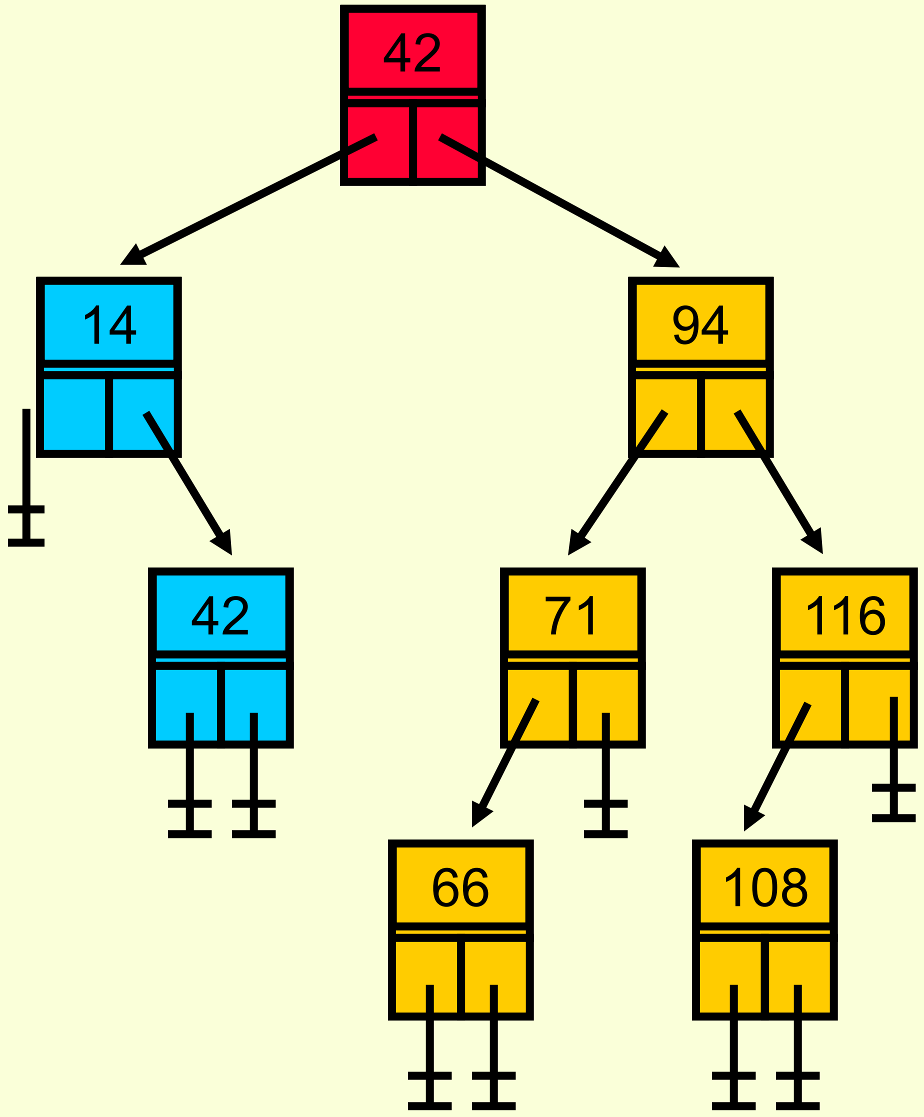
Look to the left sub-tree.

Delete a Node with Two Children



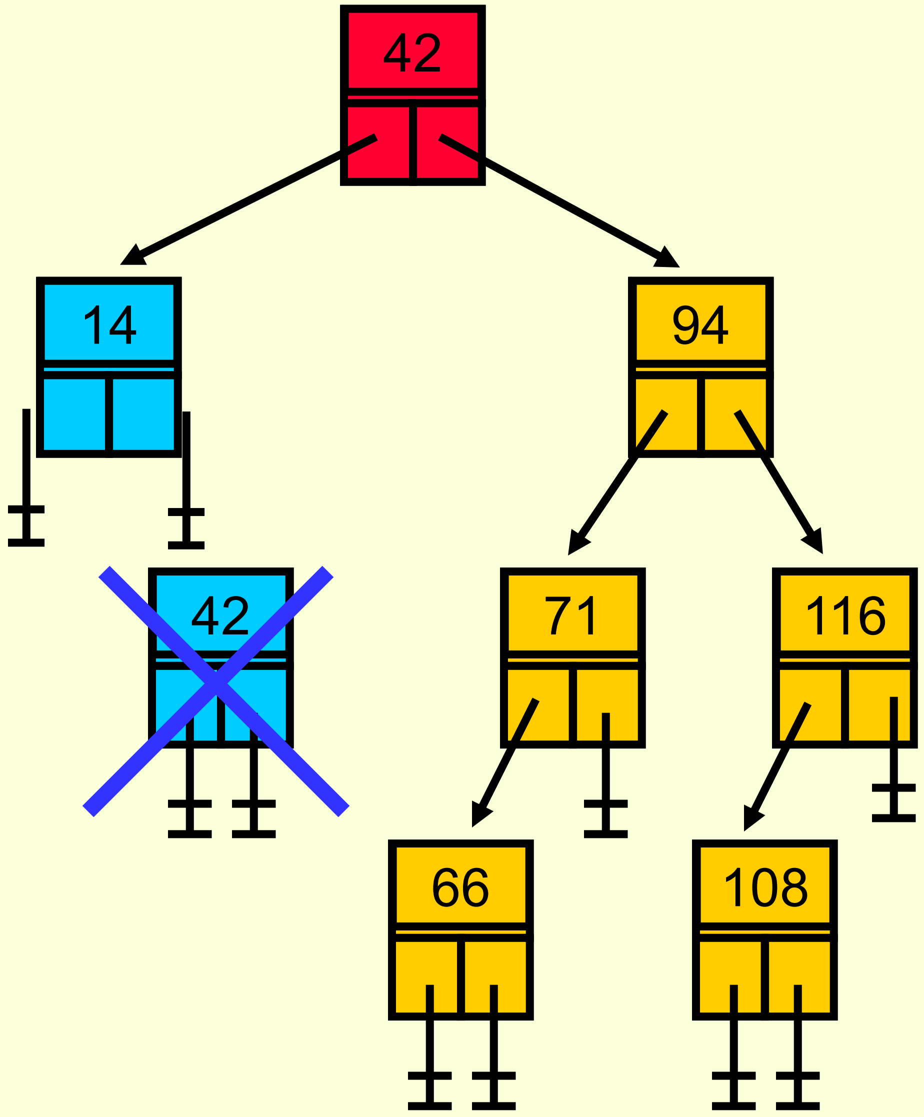
Find and copy the largest value (this will erase the old value but creates a duplicate).

Delete a Node with Two Children



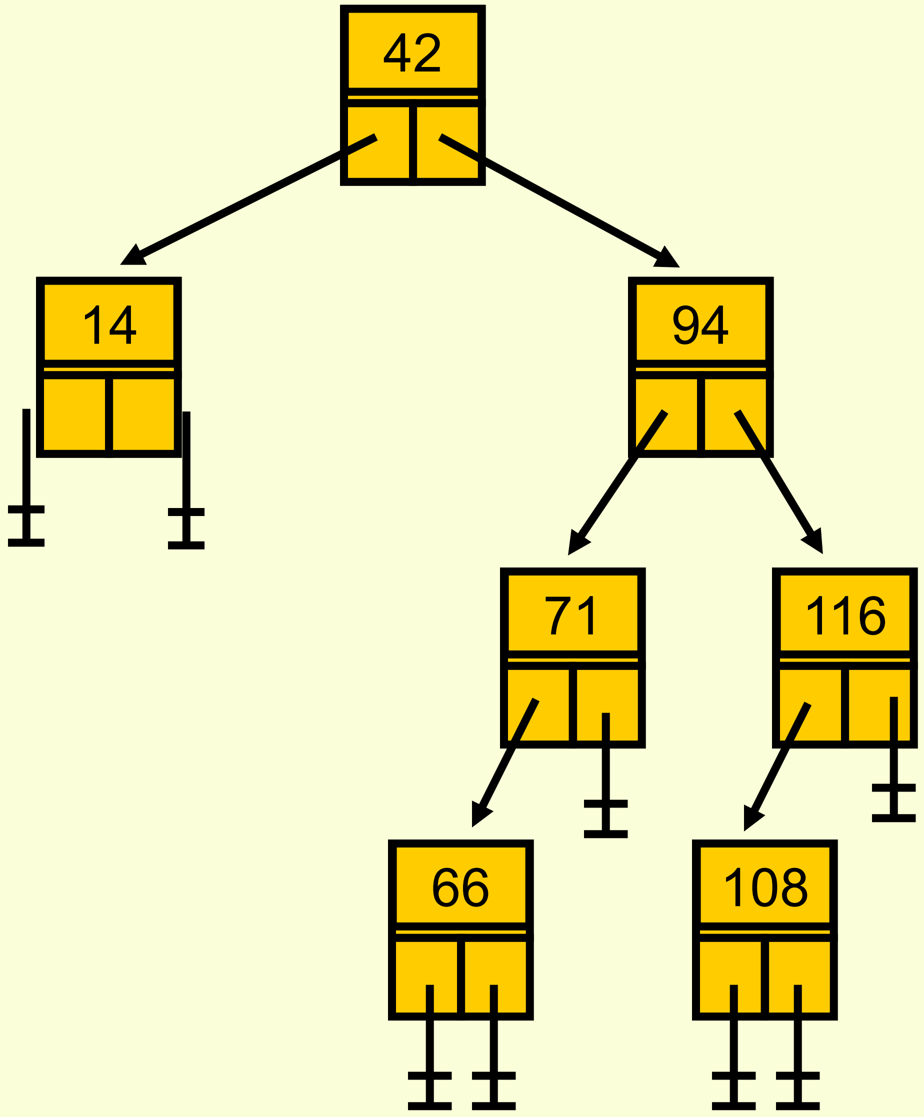
The resulting tree so far.

Delete a Node with Two Children



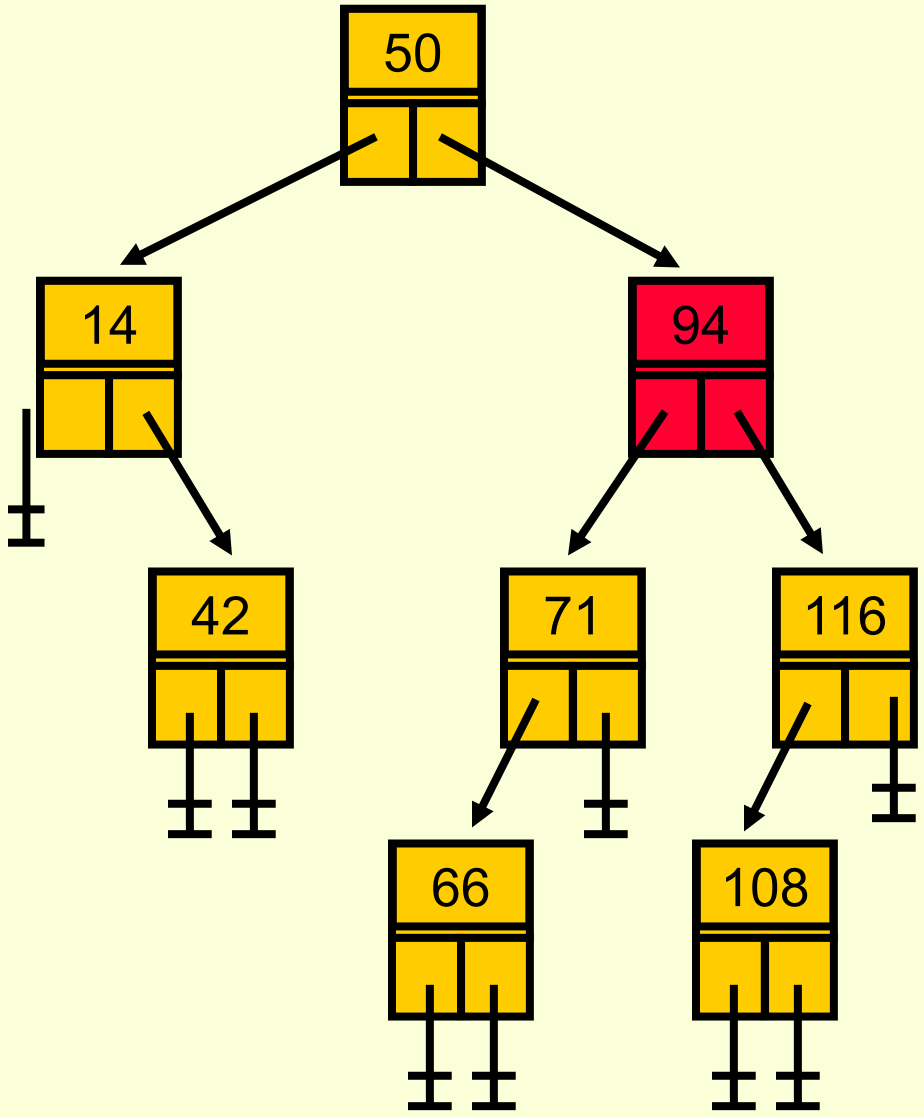
Now delete the duplicate from the left sub-tree.

Delete a Node with Two Children



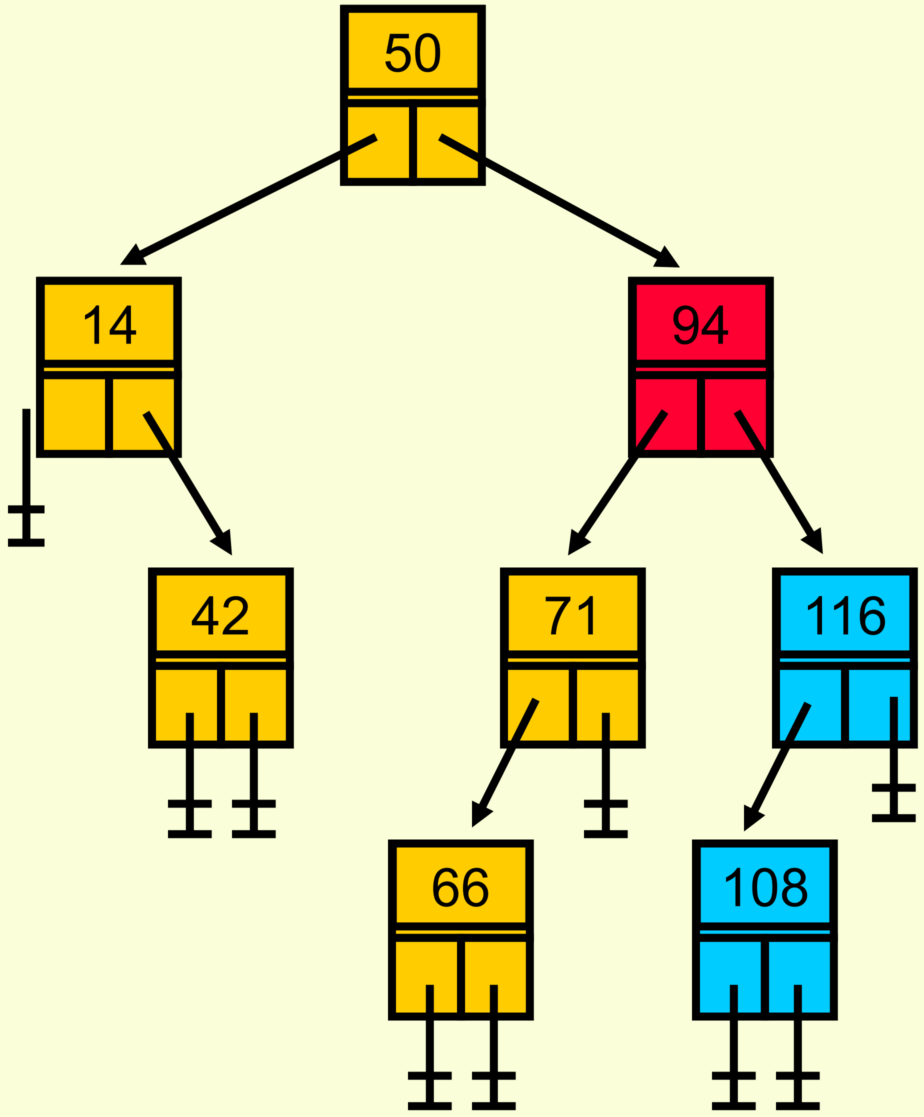
The final resulting tree – still has search structure.

Delete a Node with Two Children



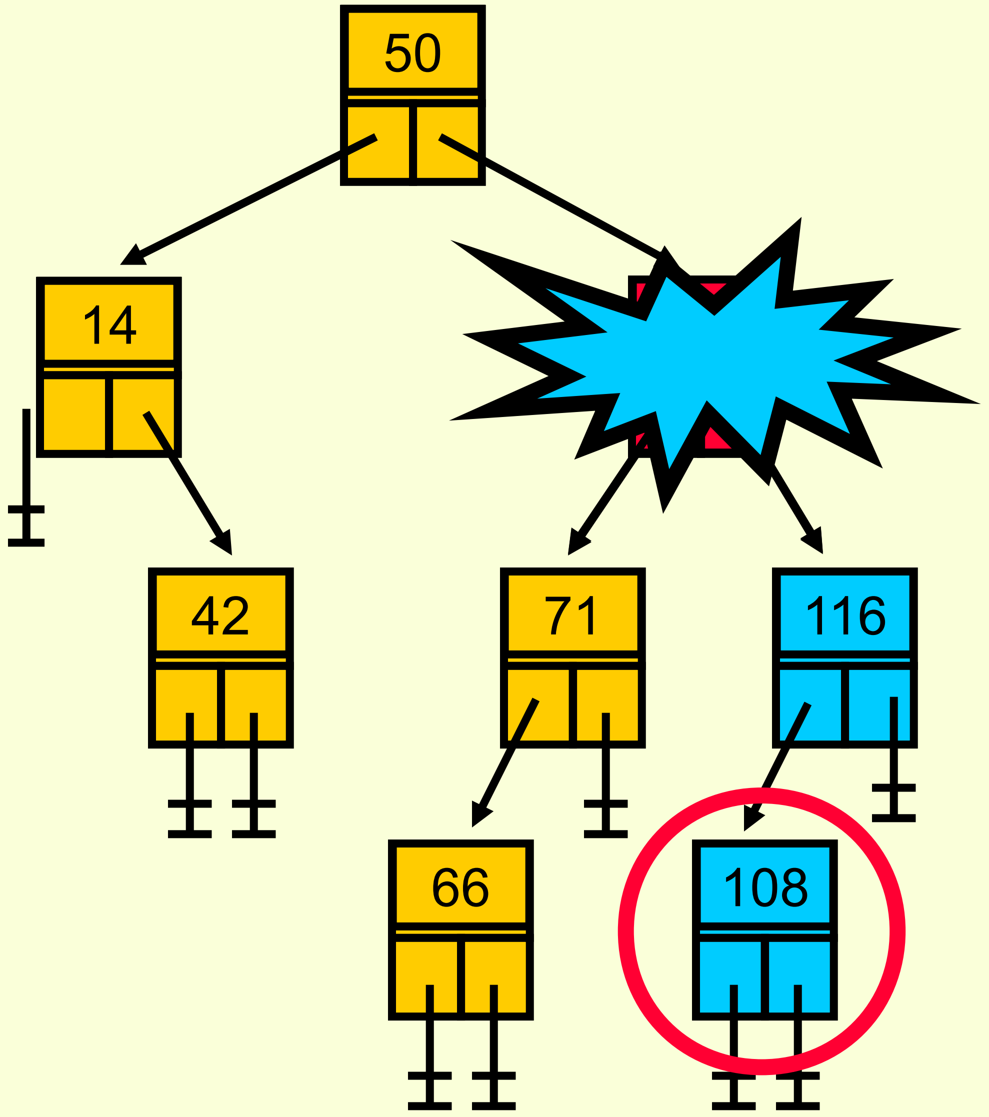
Let’s delete 94.

Delete a Node with Two Children



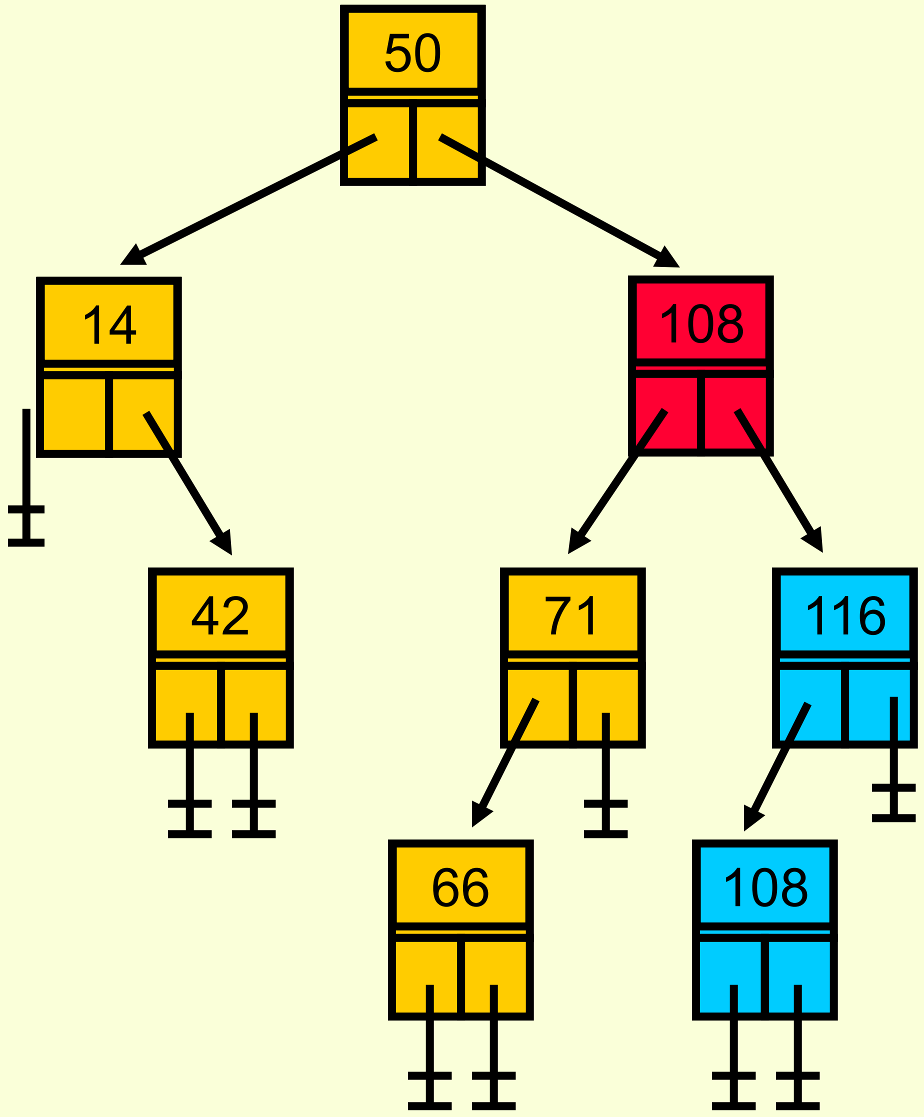
Look to the right sub-tree.

Delete a Node with Two Children



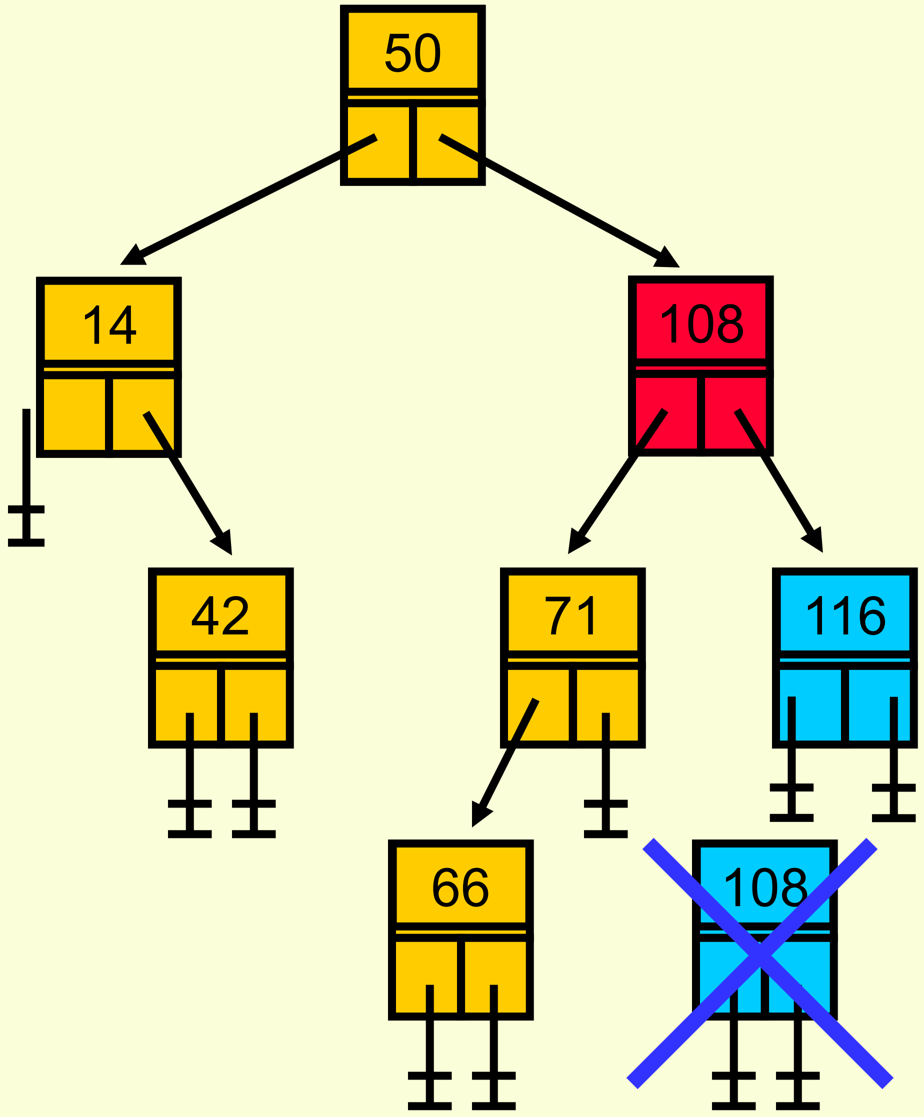
Find and copy the smallest value (this will erase the old value but creates a duplicate).

Delete a Node with Two Children



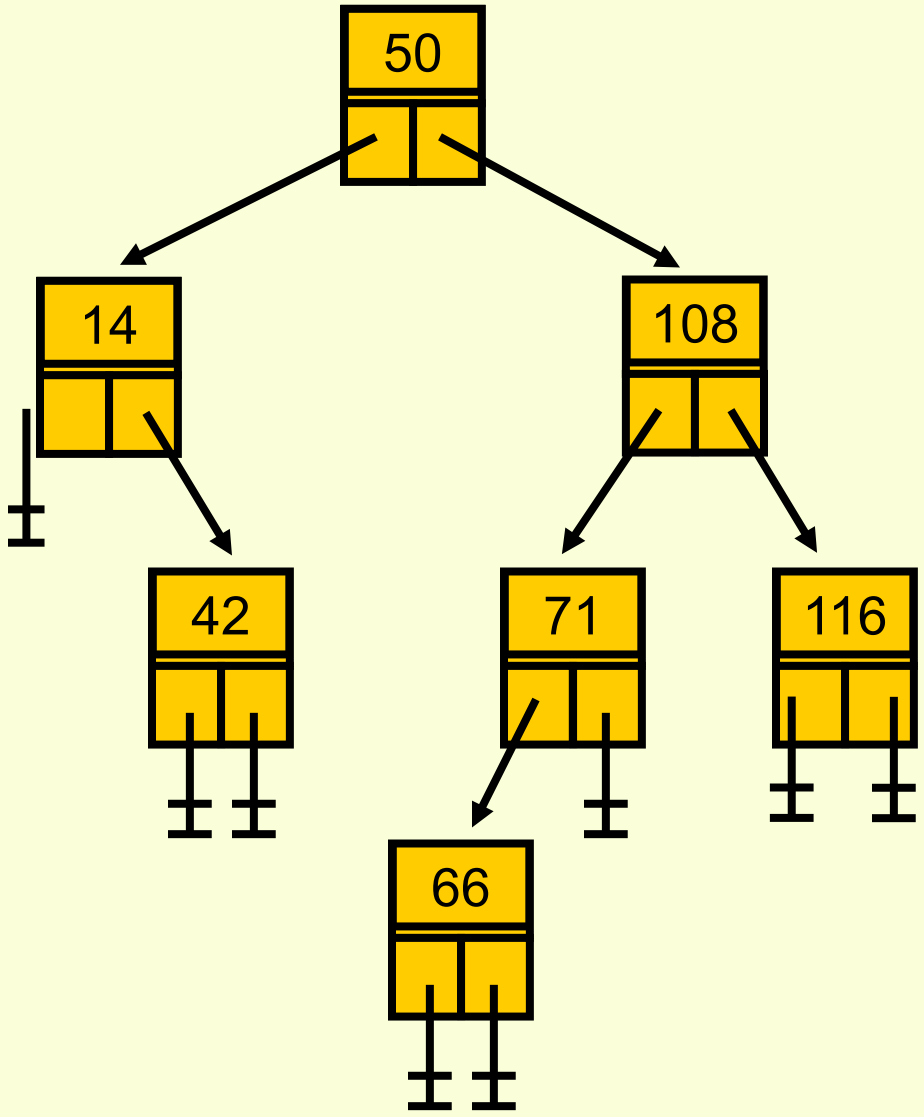
The resulting tree so far.

Delete a Node with Two Children



Now delete the duplicate from the left sub-tree.

Delete a Node with Two Children



The final resulting tree – still has search structure.

Summary

- Deleting a node from a binary search tree involves two steps:
  - Search for the element
  - Then perform the deletion
- We must preserve the search structure and only delete the element which matches.
- Four cases:
  - Deleting a leaf node
  - Deleting a node with only the left child
  - Deleting a node with only the right child
  - Deleting a node with both children

Questions?

Algorithms

AVL Tree

Balanced binary tree

- The disadvantage of a binary search tree is that its height can be as large as N-1
- This means that the time needed to perform insertion and deletion and many other operations can be O(N) in the worst case
- We want a tree with small height
- A binary tree with N node has height at least $\Theta(\log N)$
- Thus, our goal is to keep the height of a binary search tree O(log N)
- Such trees are called balanced binary search trees. Examples are AVL tree, red-black tree.

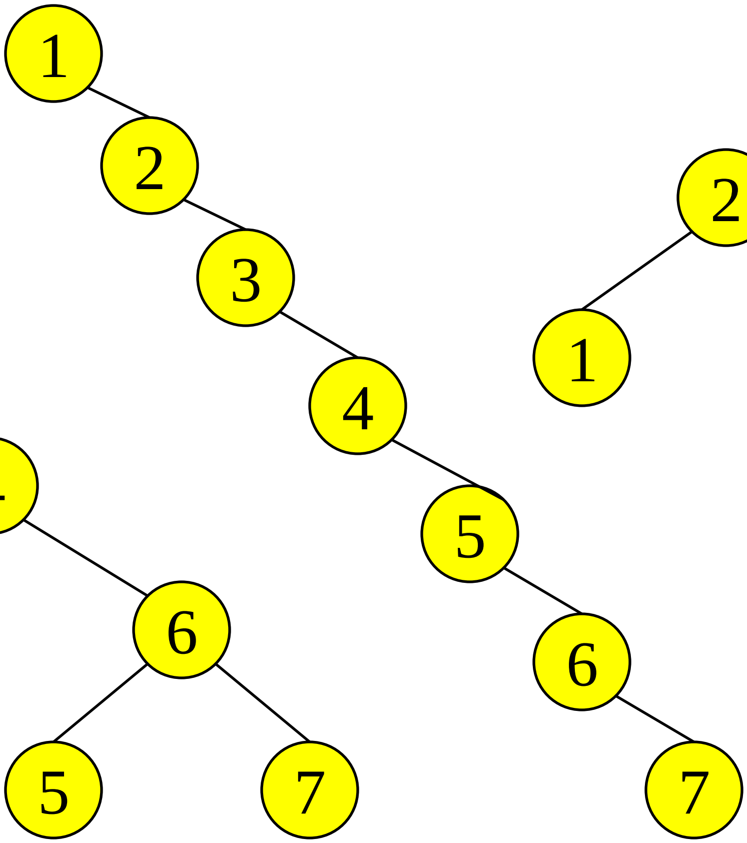
Binary Search Tree - Best Time

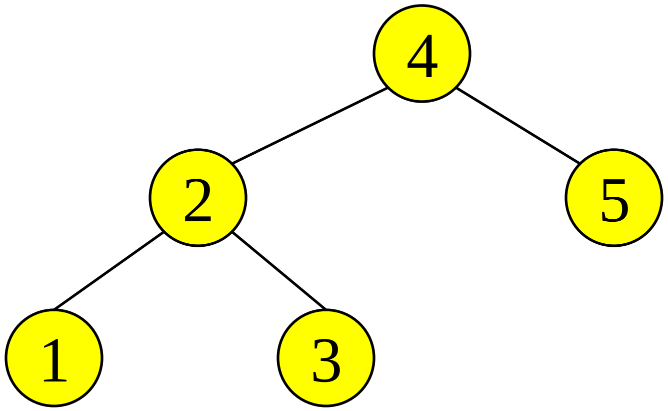
- All BST operations are O(h), where d is tree depth
- minimum d is $h = \lfloor \log_2 N \rfloor$ for a binary tree with N nodes
  - What is the best case tree?
  - What is the worst case tree?
- So, best case running time of BST operations is O(log N)

Binary Search Tree - Worst Time

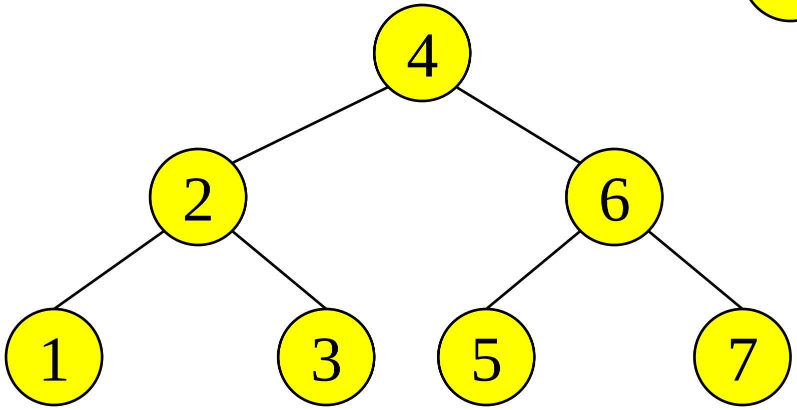
- Worst case running time is O(N)
  - What happens when you Insert elements in ascending order?
    - Insert: 2, 4, 6, 8, 10, 12 into an empty BST
  - Problem: Lack of "balance":
    - compare depths of left and right subtree
  - Unbalanced degenerate tree

Balanced and unbalanced BST





Is this "balanced"?



Approaches to balancing trees

- Don't balance
  - May end up with some nodes very deep
- Strict balance
  - The tree must always be balanced perfectly
- Pretty good balance
  - Only allow a little out of balance
- Adjust on access
  - Self-adjusting

Balancing Binary Search Trees

- Many algorithms exist for keeping binary search trees balanced
  - Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
  - Splay trees and other self-adjusting trees
  - B-trees and other multiway search trees

AVL Tree is…

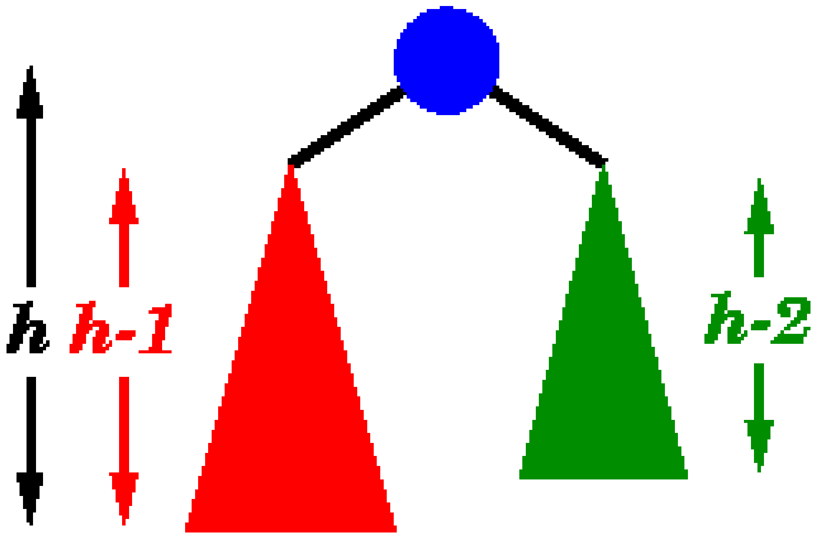
- Named after **A**delson-**V**elskii and **L**andis
- the first dynamically balanced trees to be propose
- Binary search tree with **balance condition** in which the sub-trees of each node can differ by <u>at most 1</u> in their height

Definition of a balanced tree

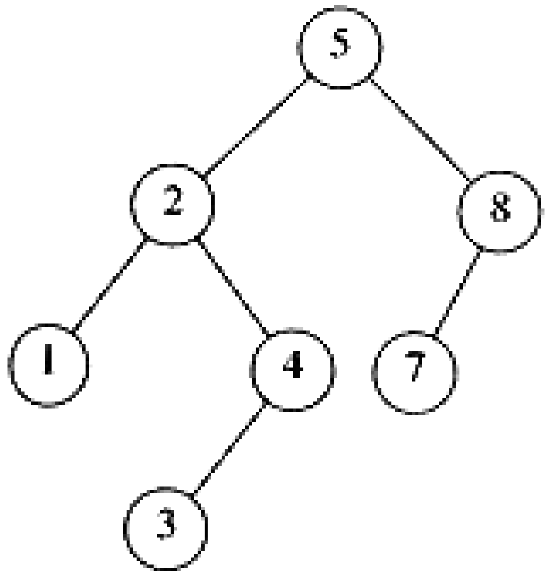
- Ensure the depth = O(log N)
- Take O(log N) time for searching, insertion, and deletion
- Every node must have left & right sub-trees of the same height

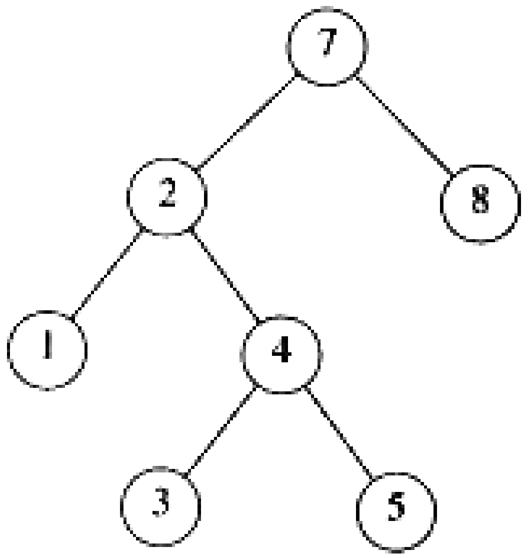
An AVL tree has the following properties:

1. Sub-trees of each node can differ by at most 1 in their height
2. Every sub-trees is an AVL tree



AVL tree?





YES

Each left sub-tree has height 1 greater than each right sub-tree

NO

Left sub-tree has height 3, but right sub-tree has height 1

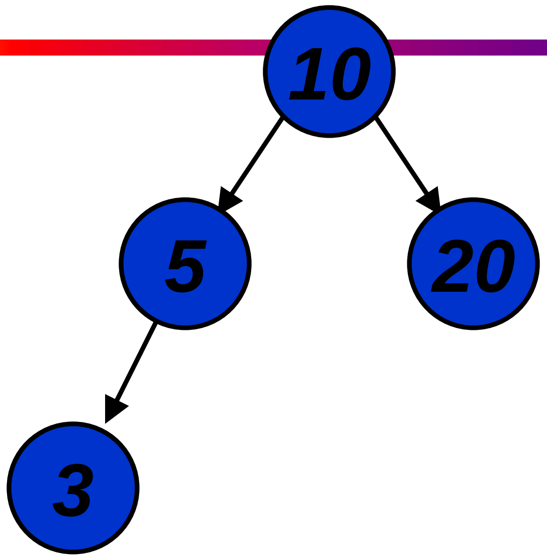
AVL tree

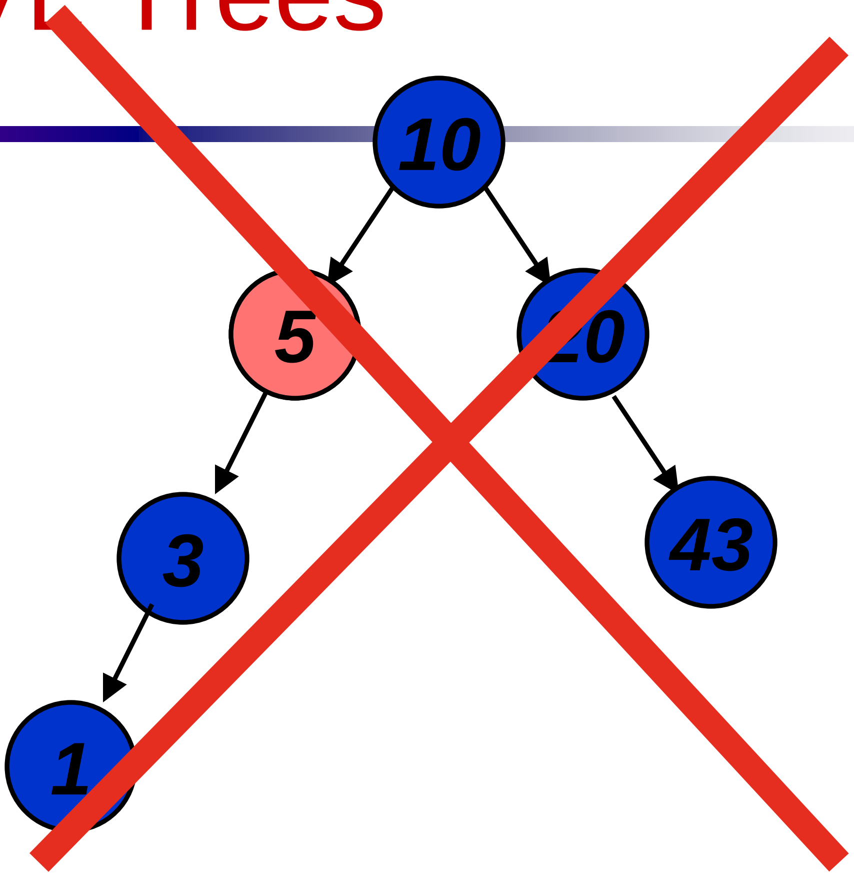
Height of a node

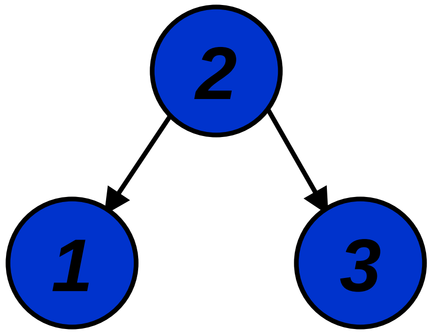
- The height of a leaf is 1. The height of a null pointer is zero.
- The height of an internal node is the maximum height of its children plus 1

Note that this definition of height is different from the one we defined previously (we defined the height of a leaf as zero previously).

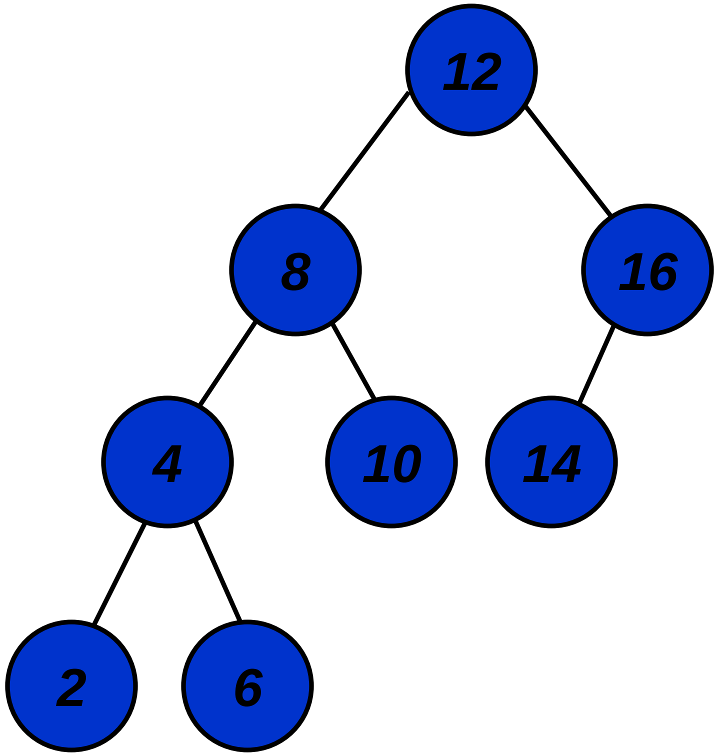
AVL Trees



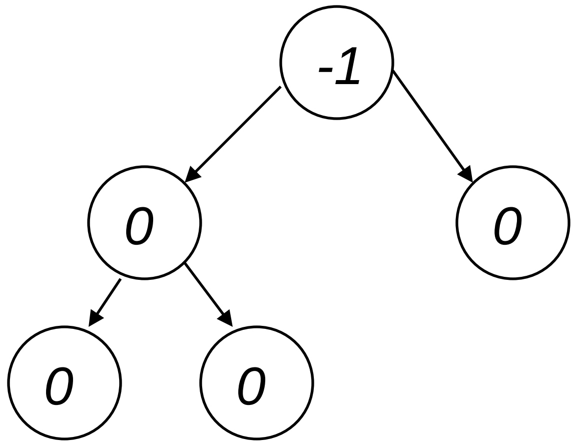




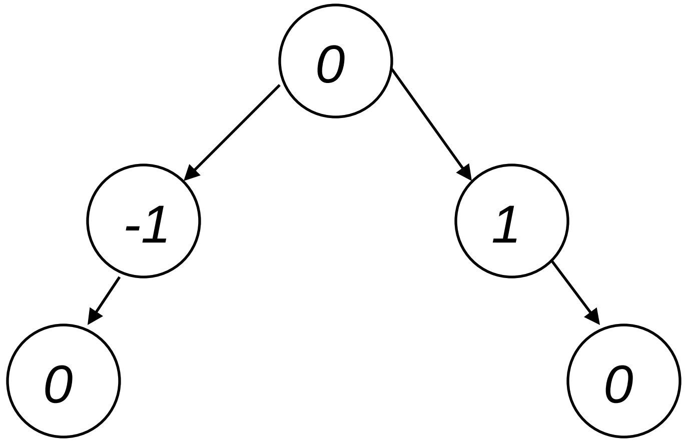
AVL Trees



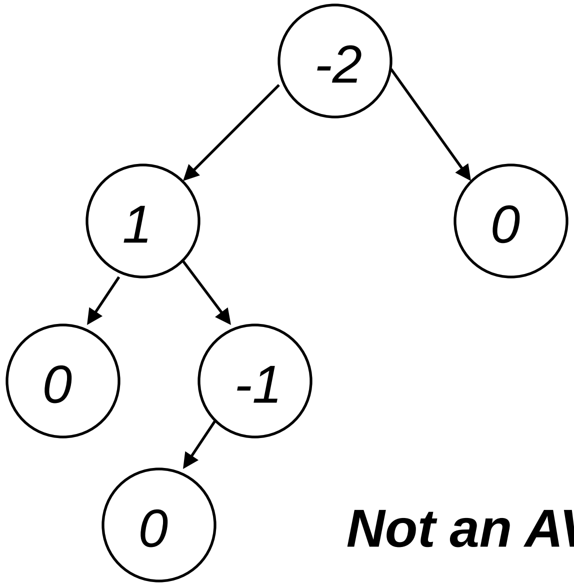
AVL Tree



AVL Tree

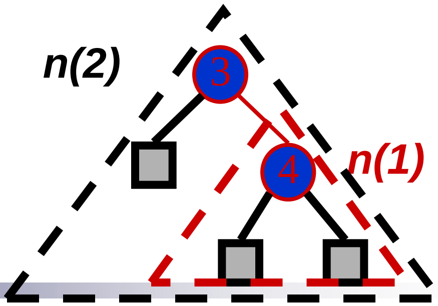


AVL Tree



Not an AVL Tree

Height of an AVL Tree



- **Fact**: The **height** of an AVL tree storing n keys is O(log n).
- **Proof**: Let us bound **n(h):** the minimum number of internal nodes of an AVL tree of height h.
- We easily see that n(1) = 1 and n(2) = 2
- For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height n-1 and another of height n-2.
- That is, n(h) = 1 + n(h-1) + n(h-2)
- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So

  n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (by induction),

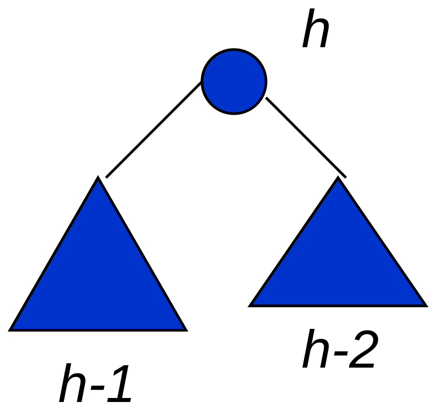
  n(h) > 2^in(h-2i)
- Solving the base case we get: n(h) > $2^{h/2-1}$
- Taking logarithms: h < 2log n(h) +2
- Thus the height of an AVL tree is O(log n)

AVL - Good but not Perfect Balance

- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - height(left subtree) - height(right subtree)
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right subtree can differ by no more than 1
  - Store current heights in each node

Height of an AVL Tree

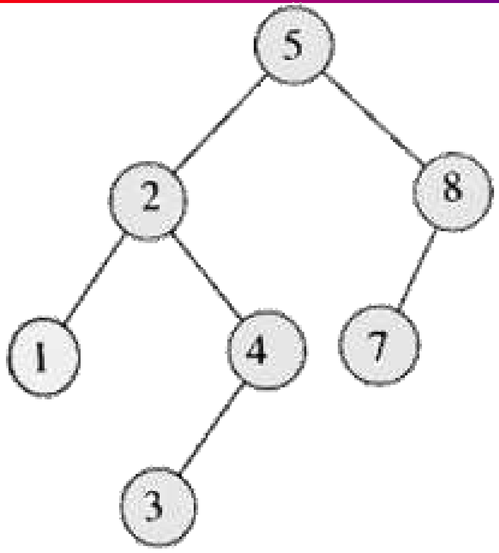
- N(h) = minimum number of nodes in an AVL tree of height h.
- Basis
  - N(0) = 1, N(1) = 2
- Induction
  - N(h) = N(h-1) + N(h-2) + 1
- Solution (recall Fibonacci analysis)
  - $N(h) \geq \phi^h \quad (\phi \approx 1.62)$



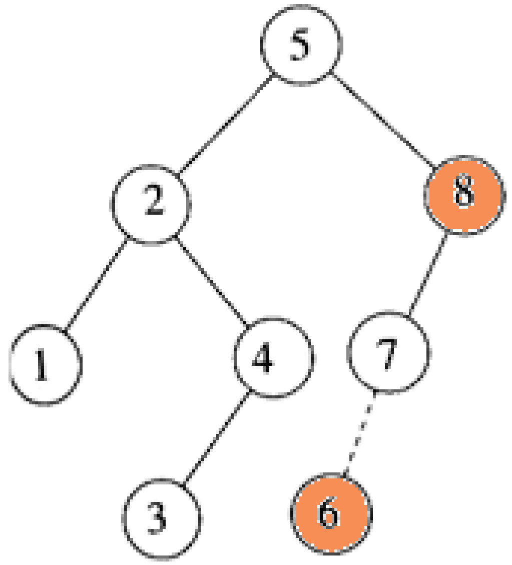
Height of an AVL Tree

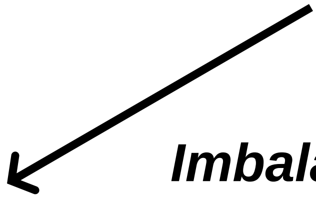
- $N(h) \geq \phi^h$ $(\phi \approx 1.62)$
- Suppose we have n nodes in an AVL tree of height h.
  - $n \geq N(h)$ (because N(h) was the minimum)
  - $n \geq \phi^h$ hence $\log_\phi n \geq h$ (relatively well balanced tree!!)
  - $h \leq 1.44 \log_2 n$ (i.e., Find takes O(log n))

Insertion

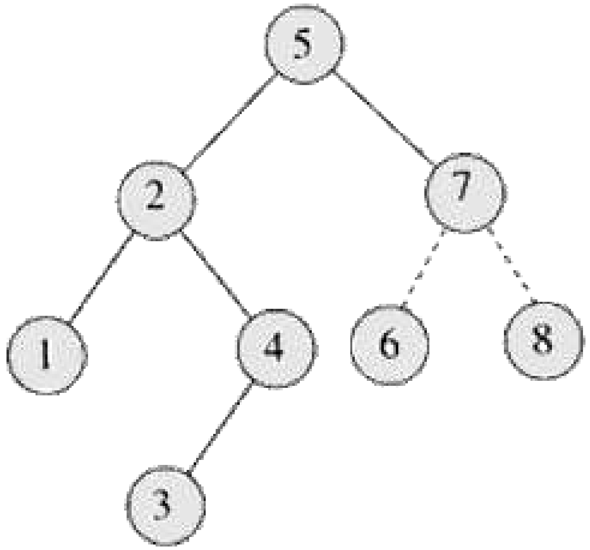


Insert 6

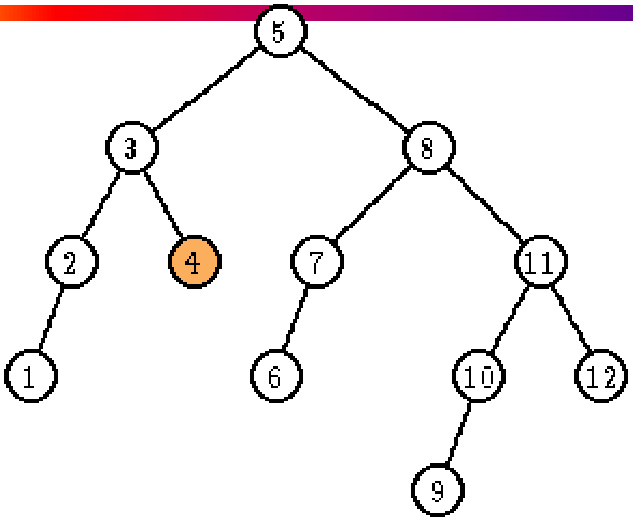




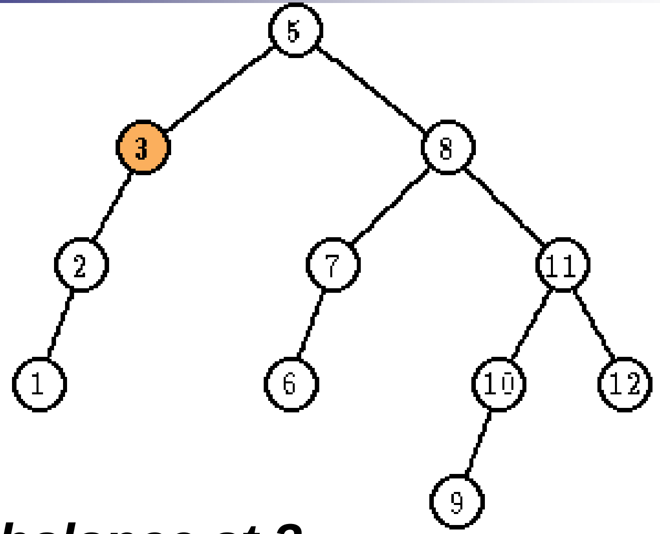
Imbalance at 8
Perform rotation with 7



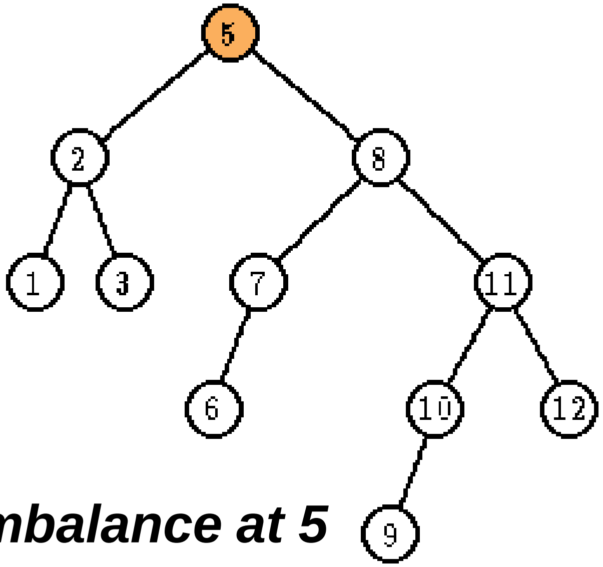
Deletion



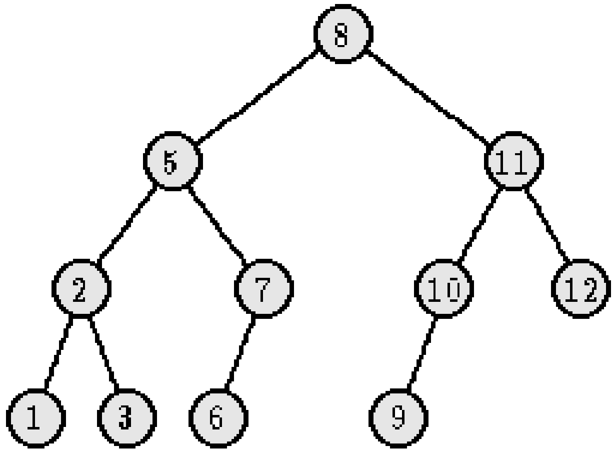
Delete 4



Imbalance at 3
Perform rotation with 2



Imbalance at 5
Perform rotation with 8



Key Points

- AVL tree remain balanced by applying rotations, therefore it guarantees O(log N) search time in a dynamic environment
- Tree can be re-balanced in at most O(log N) time

Searching AVL Trees

- Searching an AVL tree is exactly the same as searching a regular binary tree
  - all descendants to the right of a node are greater than the node
  - all descendants to the left of a node are less than the node

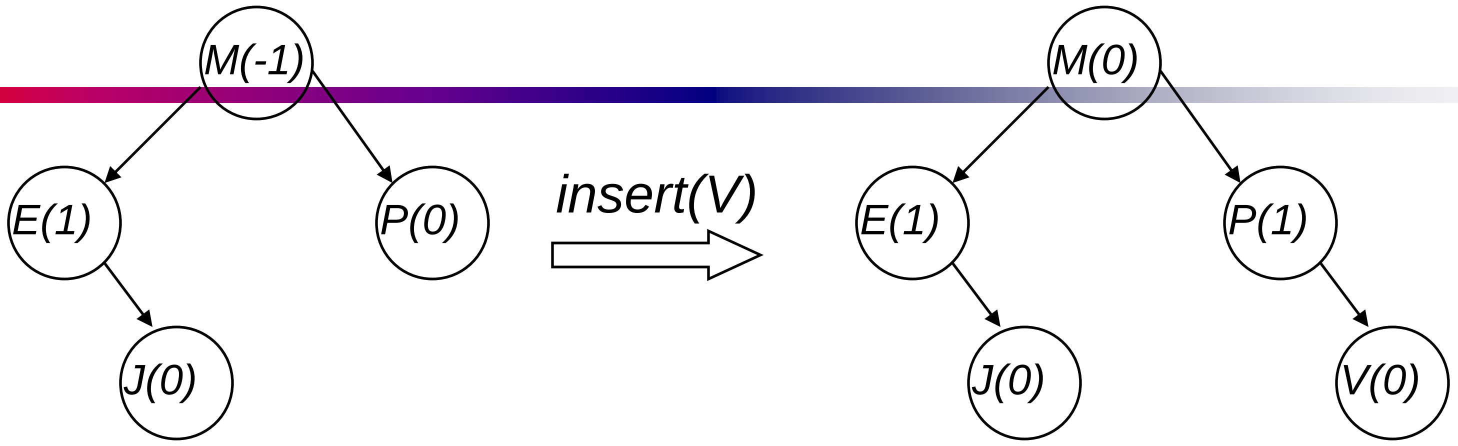
Inserting in AVL Tree

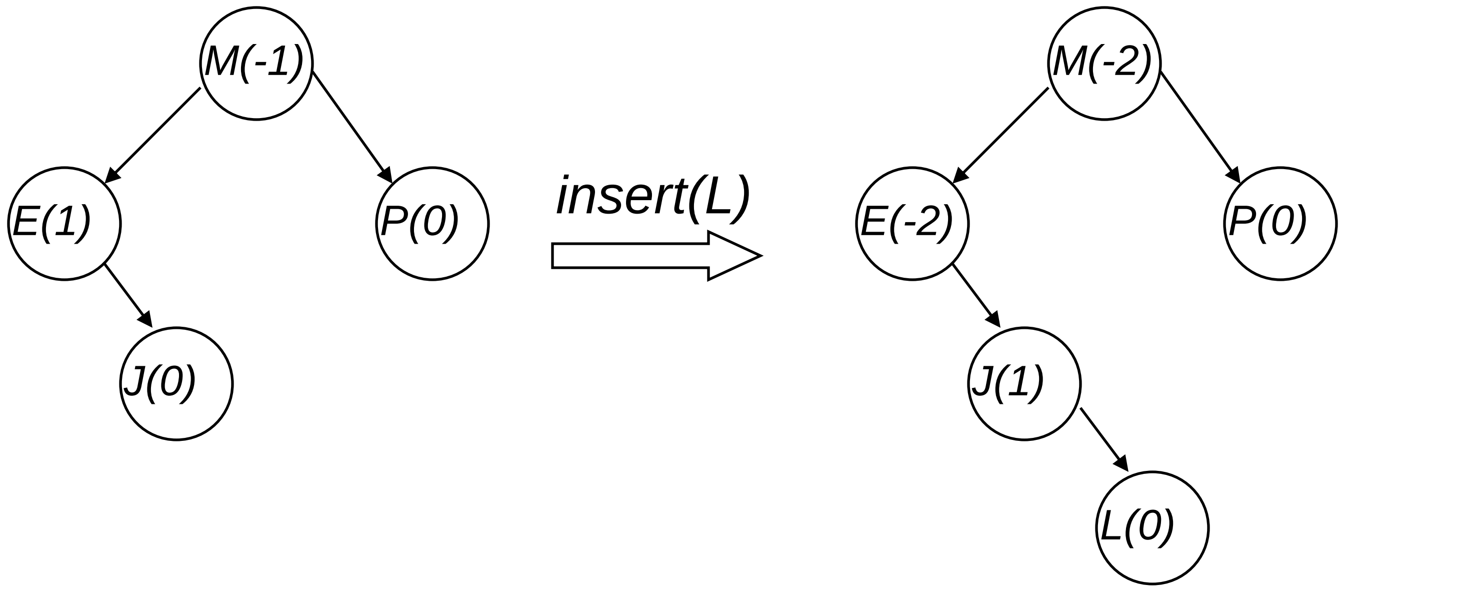
- Insertion is similar to regular binary tree
  - keep going left (or right) in the tree until a null child is reached
  - insert a new node in this position
    - an inserted node is always a leaf to start with
- Major difference from binary tree
  - must check if any of the sub-trees in the tree have become too unbalanced
    - search from inserted node to root looking for any node with a balance factor of 2

Inserting in AVL Tree

- A few points about tree inserts
  - the insert will be done recursively
  - the insert call will return true if the height of the sub-tree has changed
    - since we are doing an insert, the height of the sub-tree can only increase
  - if insert() returns true, balance factor of current node needs to be adjusted
    - balance factor = height(right) – height(left)
      - left sub-tree increases, balance factor decreases by 1
      - right sub-tree increases, balance factor increases by 1
  - if balance factor equals   2 for any node, the sub-tree must be rebalanced

Inserting in AVL Tree





This tree needs to be fixed!

Re-Balancing a Tree

- To check if a tree needs to be rebalanced
  - start at the parent of the inserted node and journey up the tree to the root
    - if a node's balance factor becomes   2 need to do a rotation in the sub-tree rooted at the node
    - once sub-tree has been re-balanced, guaranteed that the rest of the tree is balanced as well
      - can just return false from the insert() method
  - 4 possible cases for re-balancing
    - only 2 of them need to be considered
      - other 2 are identical but in the opposite direction

Insertions in AVL Trees

Let the node that needs rebalancing be α.

There are 4 cases:

Outside Cases (require single rotation) :

1. *Insertion into left subtree of left child of α.*
2. *Insertion into right subtree of right child of α.*

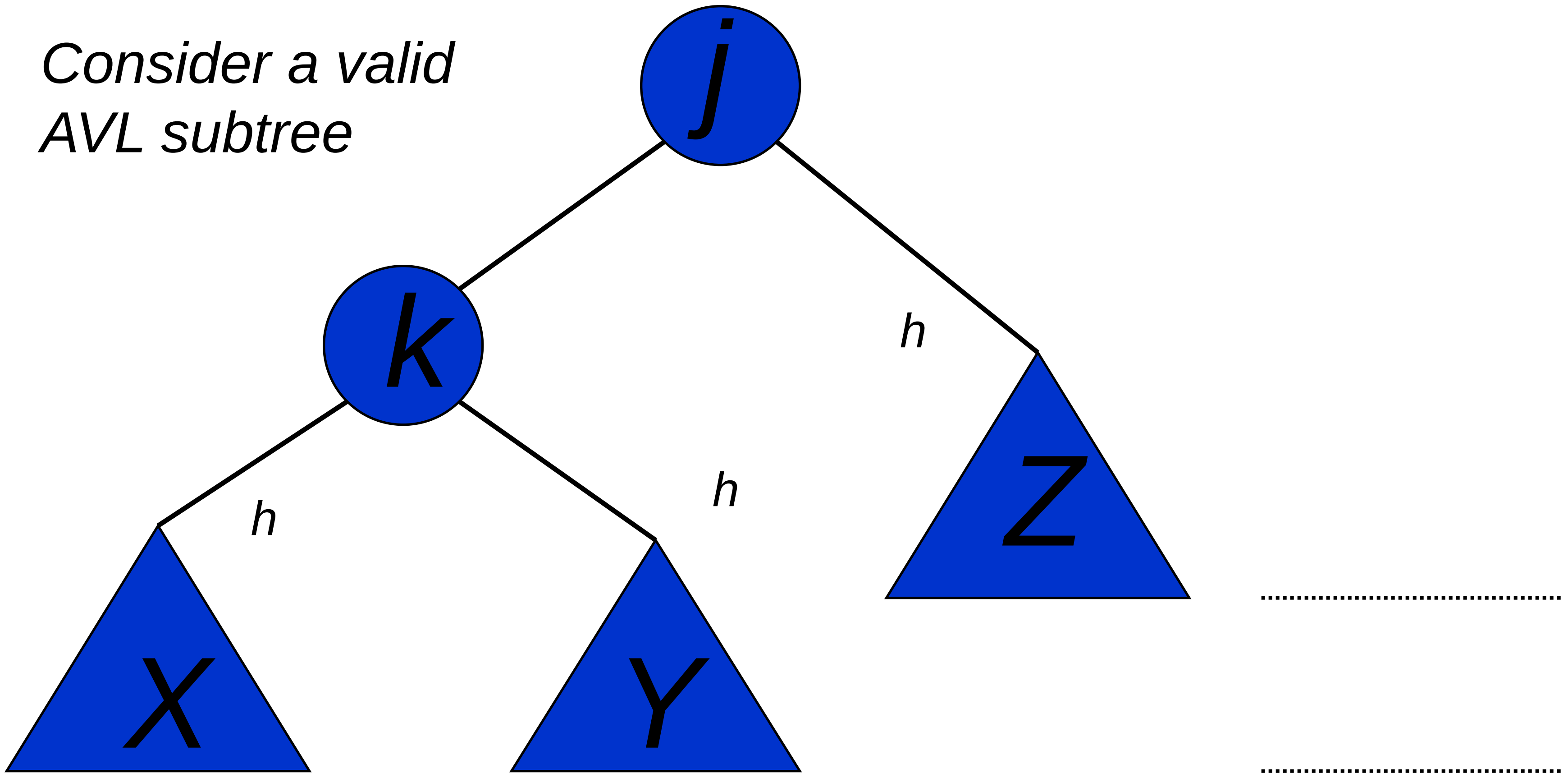
Inside Cases (require double rotation) :

3. *Insertion into right subtree of left child of α.*
4. *Insertion into left subtree of right child of α.*

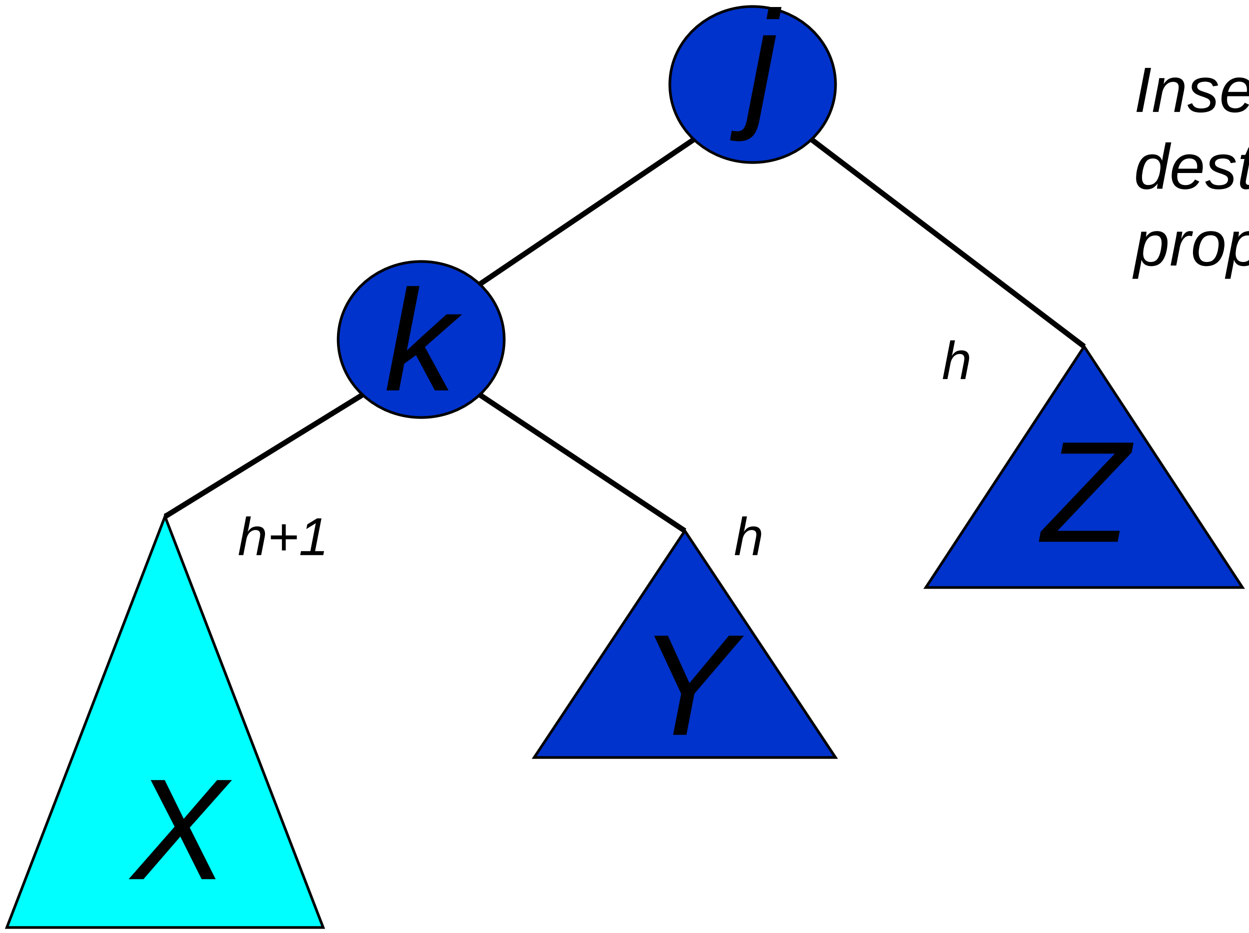
The rebalancing is performed through four separate rotation algorithms.

AVL Insertion: Outside Case

Consider a valid AVL subtree

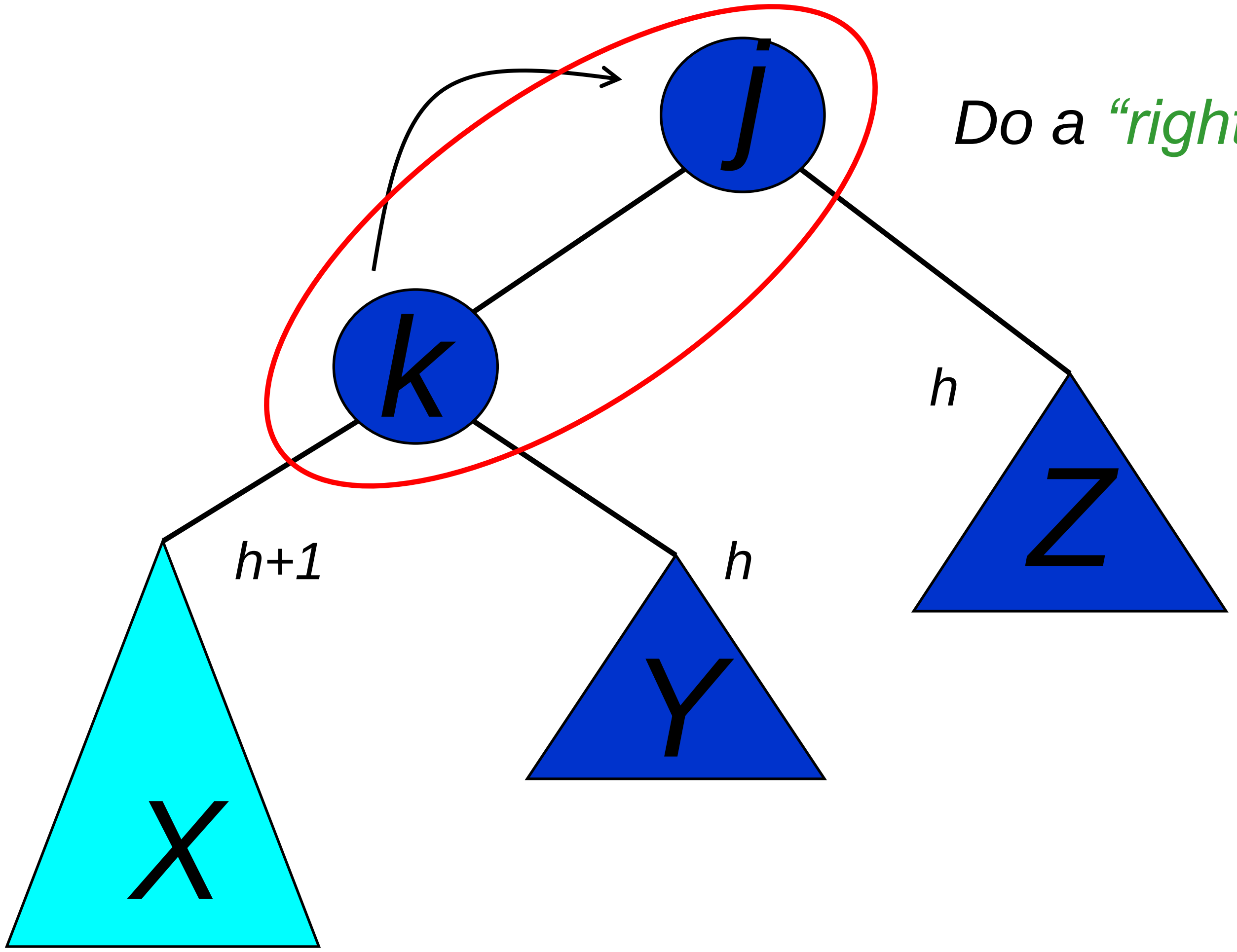


AVL Insertion: Outside Case



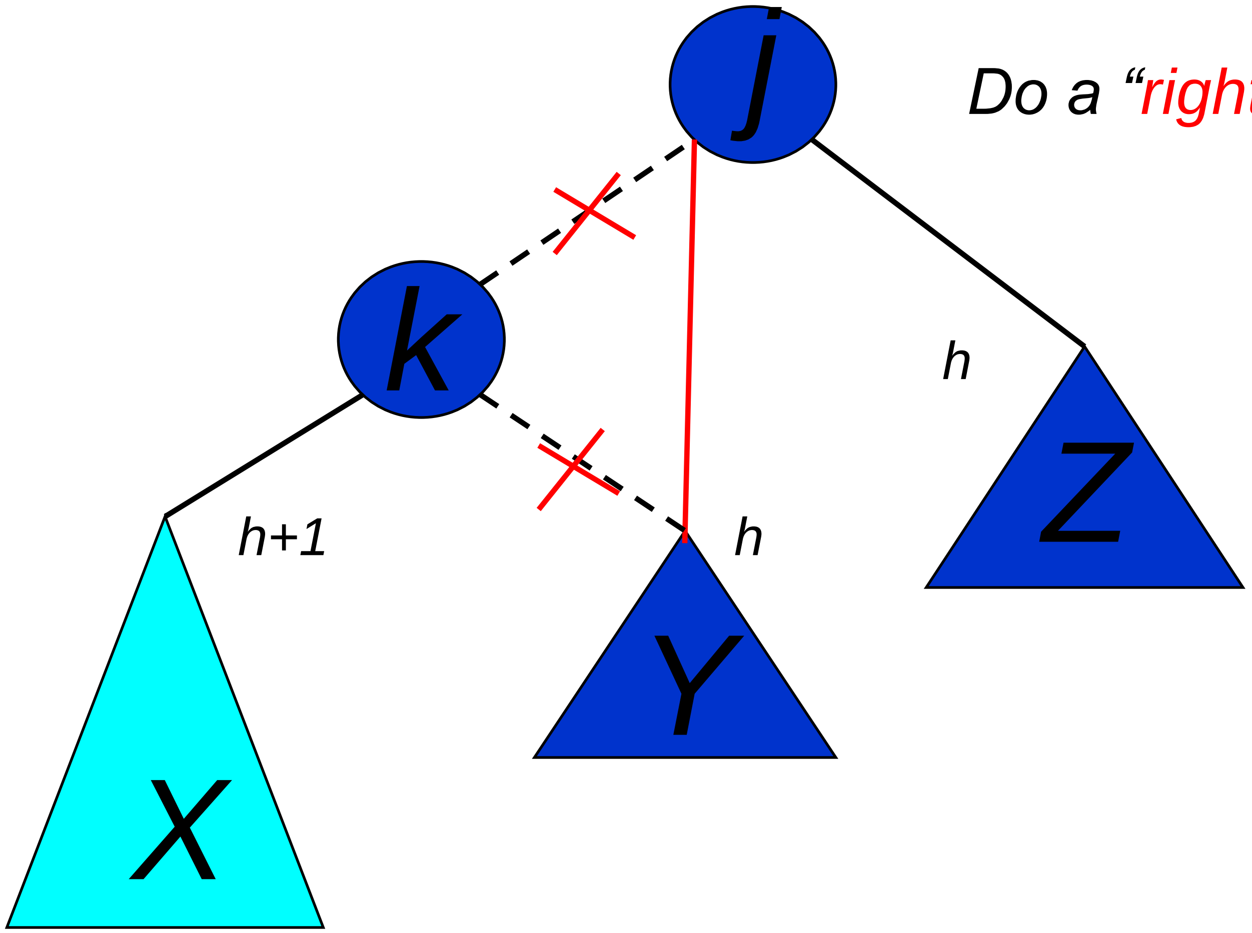
Inserting into X destroys the AVL property at node j

AVL Insertion: Outside Case



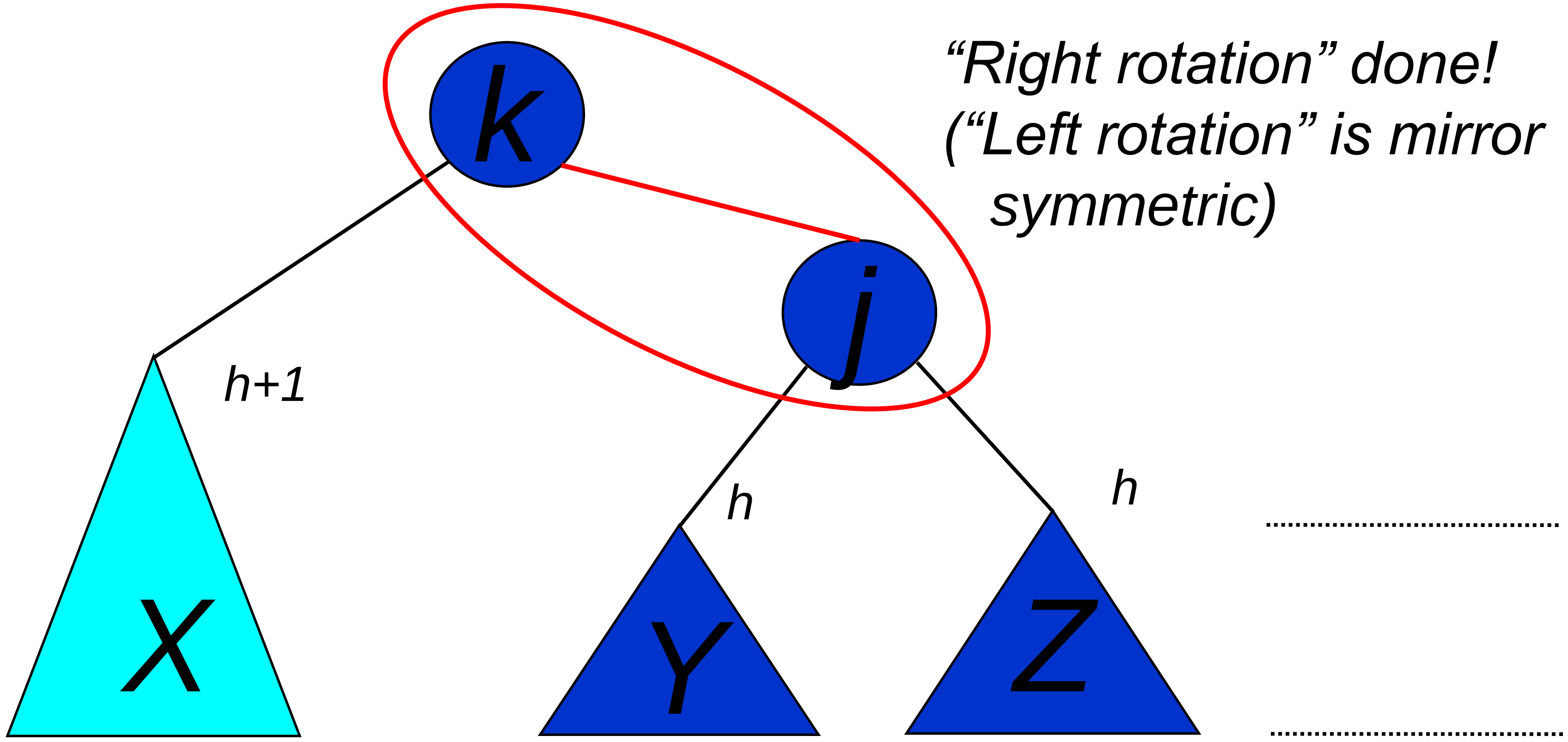
Do a *“right rotation”*

Single right rotation



Do a "right rotation"

Outside Case Completed

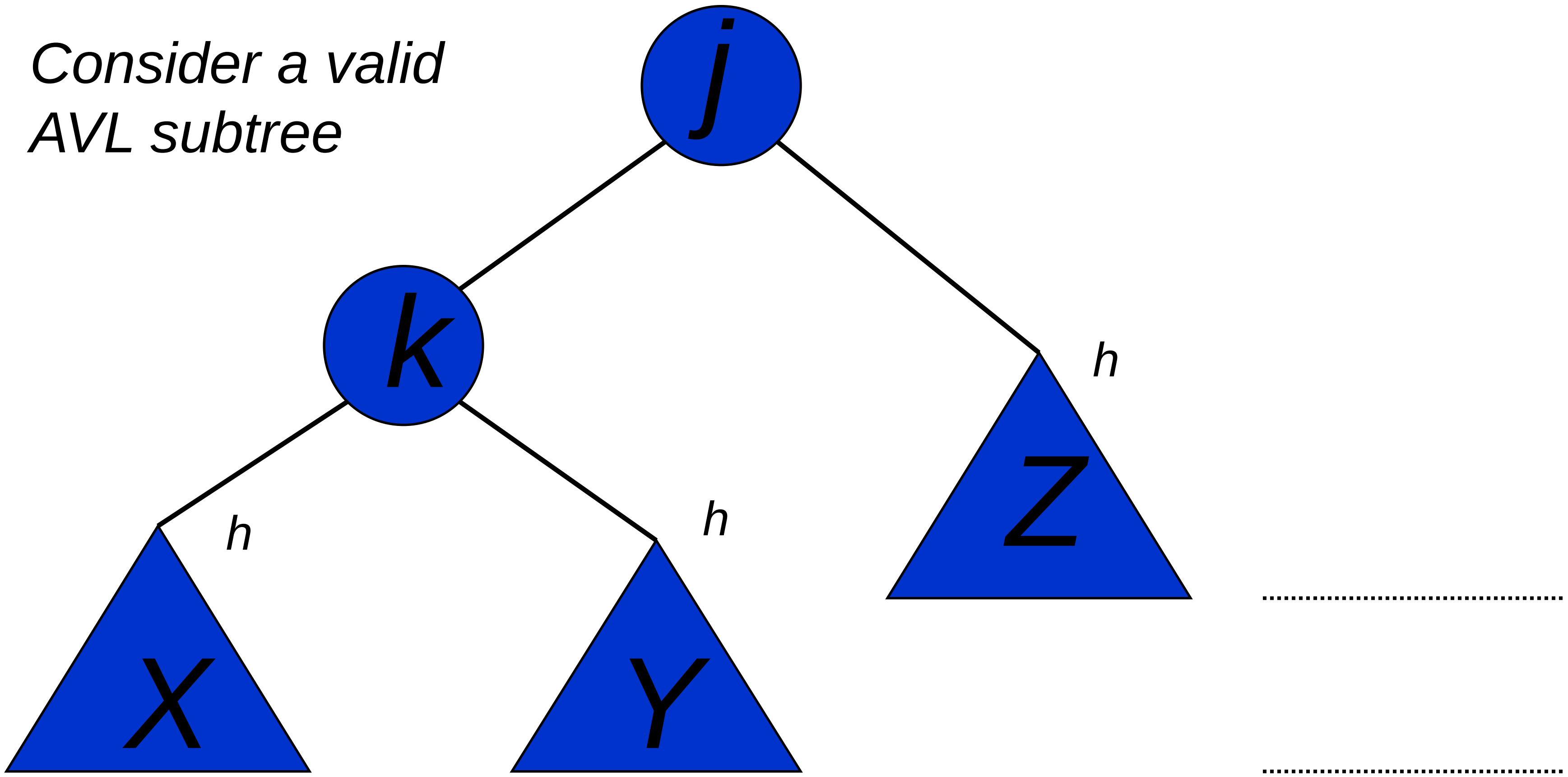


"Right rotation" done!
("Left rotation" is mirror symmetric)

AVL property has been restored!

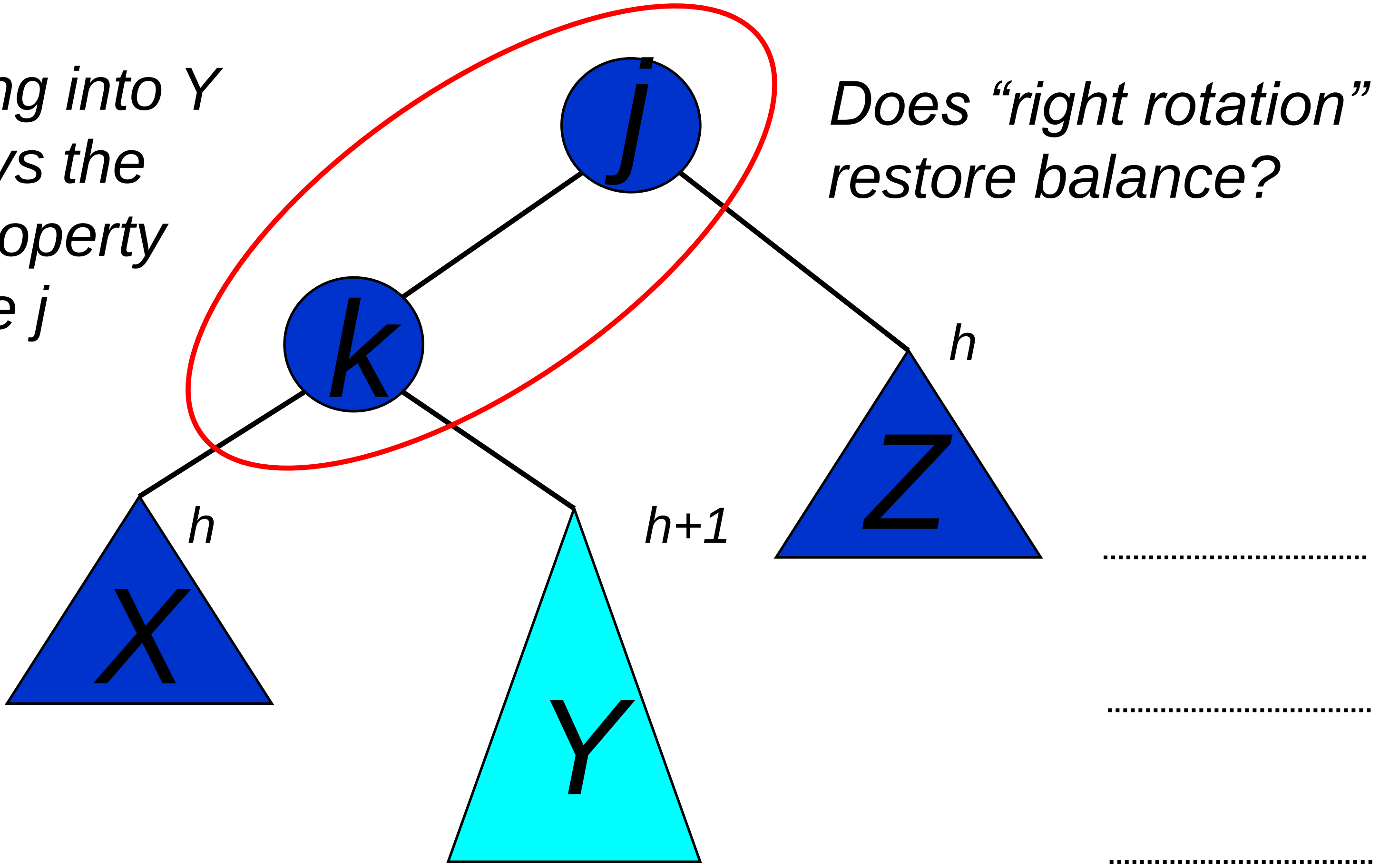
AVL Insertion: Inside Case

Consider a valid AVL subtree



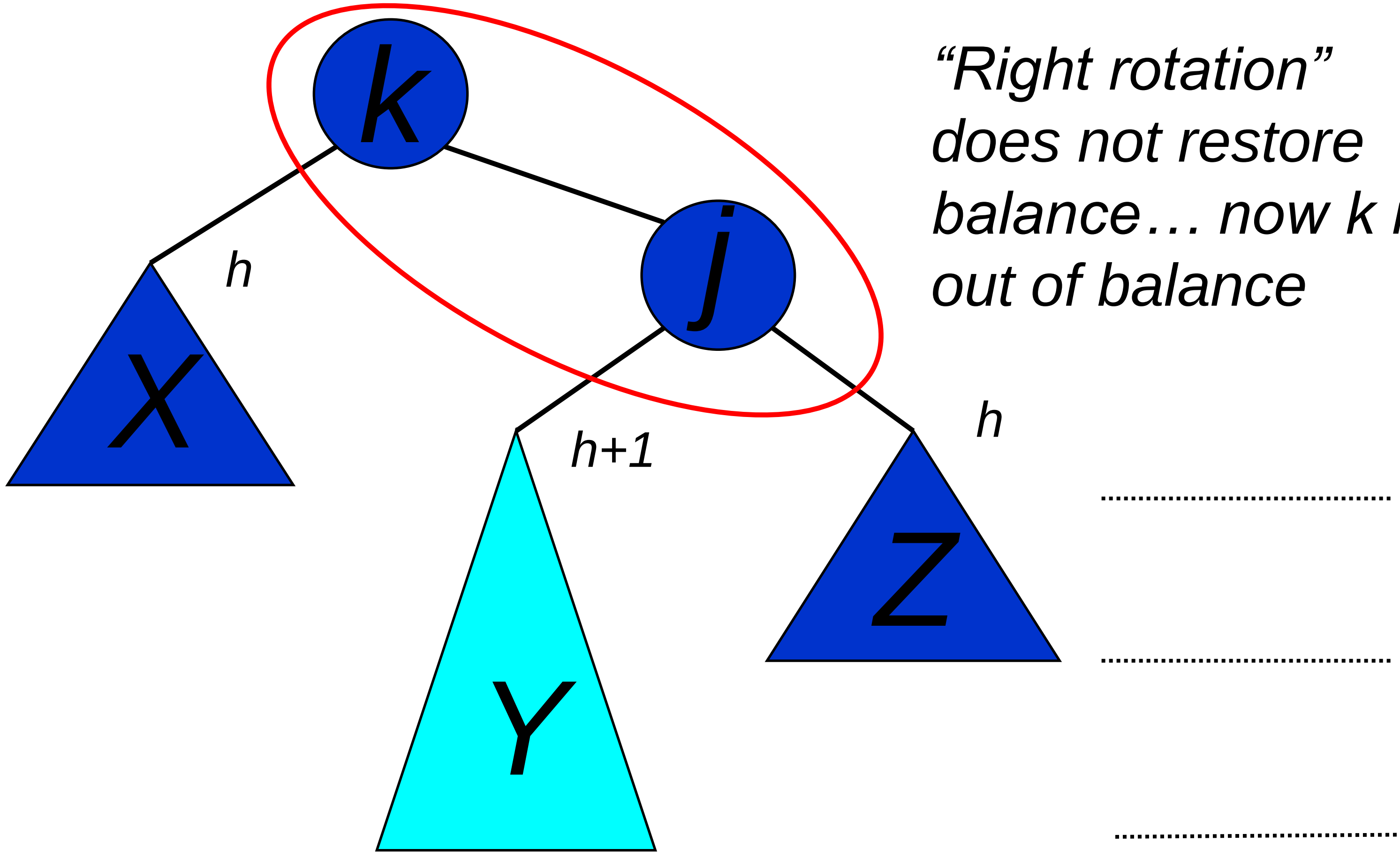
AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j



Does "right rotation" restore balance?

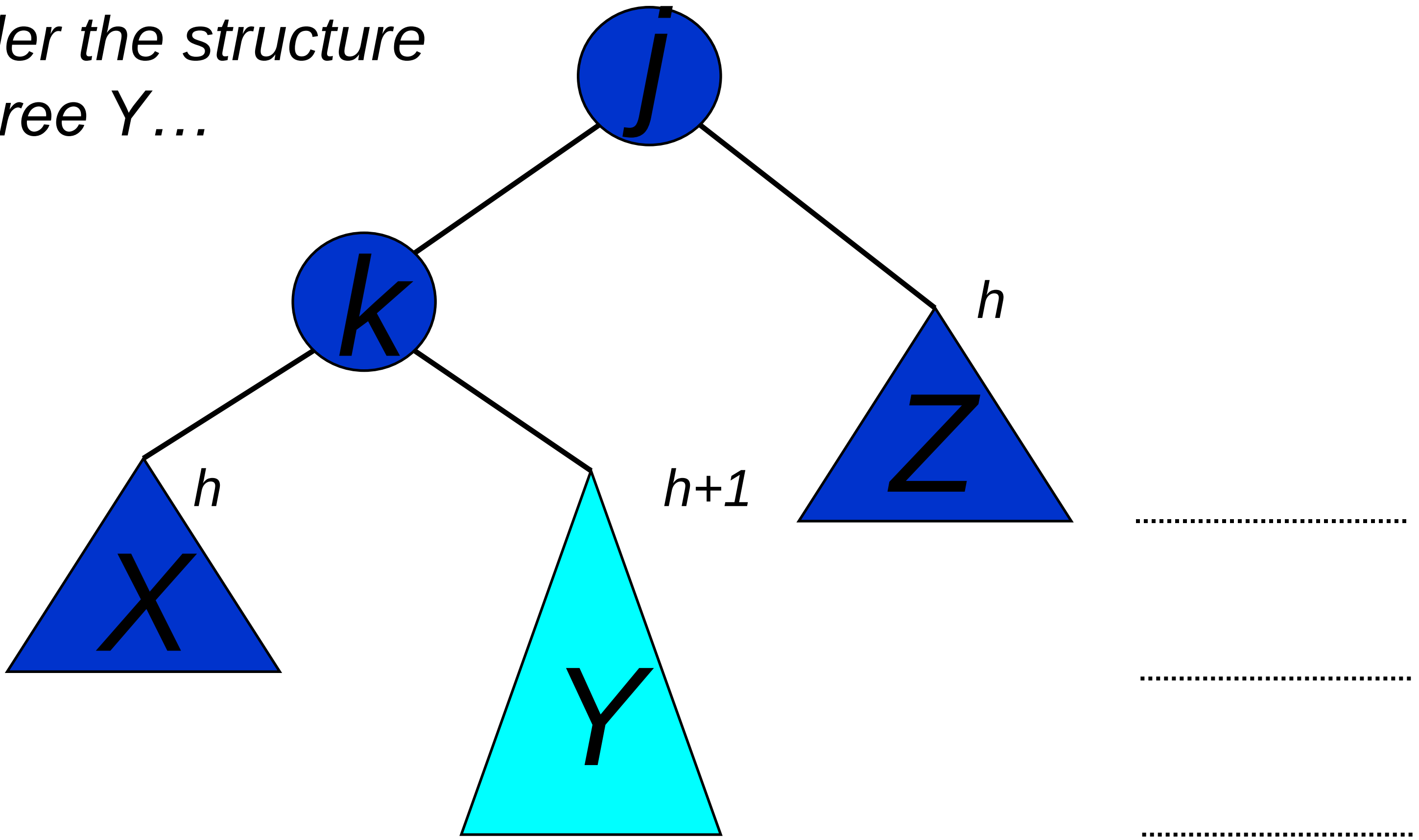
AVL Insertion: Inside Case



"Right rotation" does not restore balance… now k is out of balance

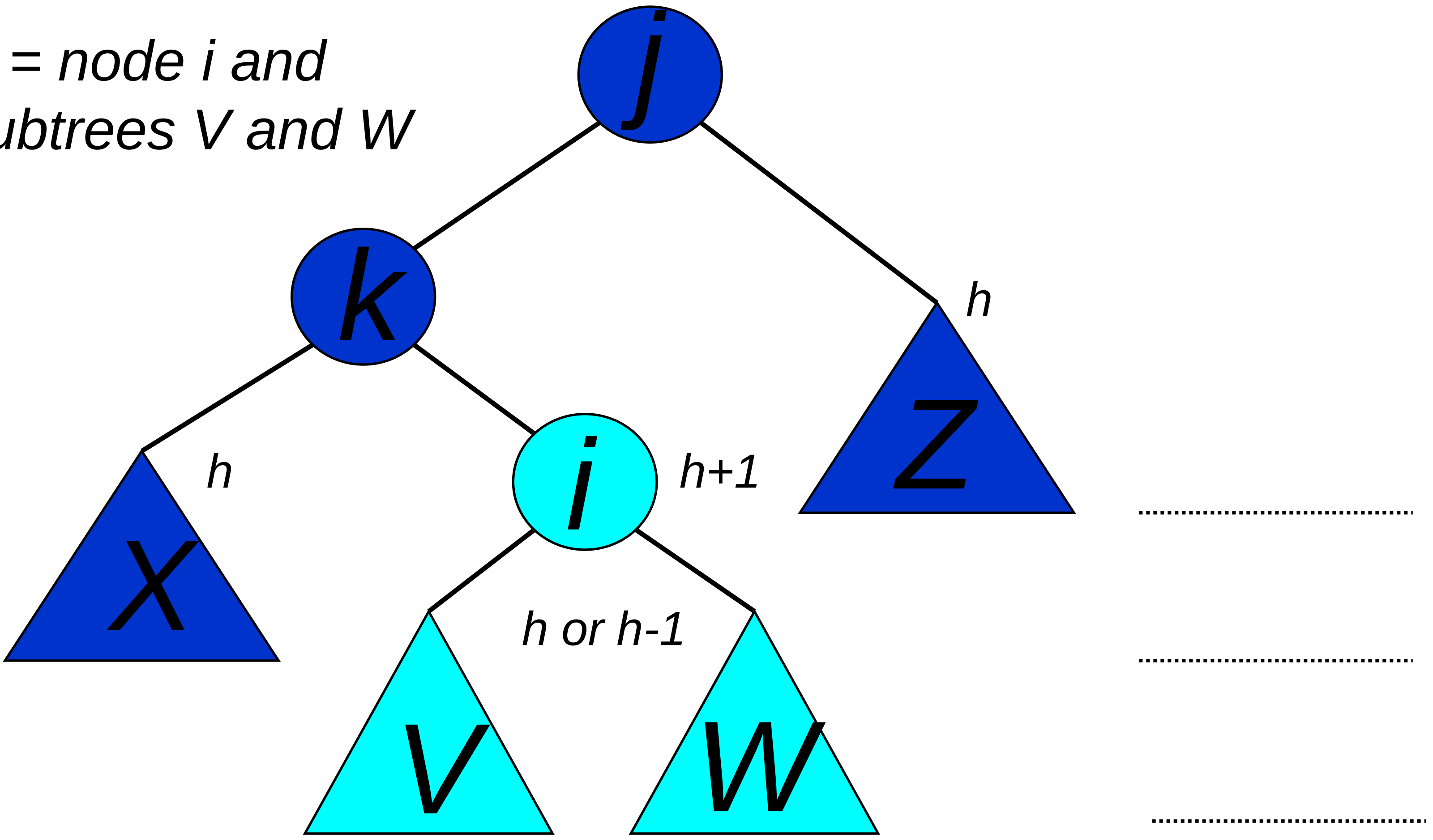
AVL Insertion: Inside Case

Consider the structure of subtree Y…

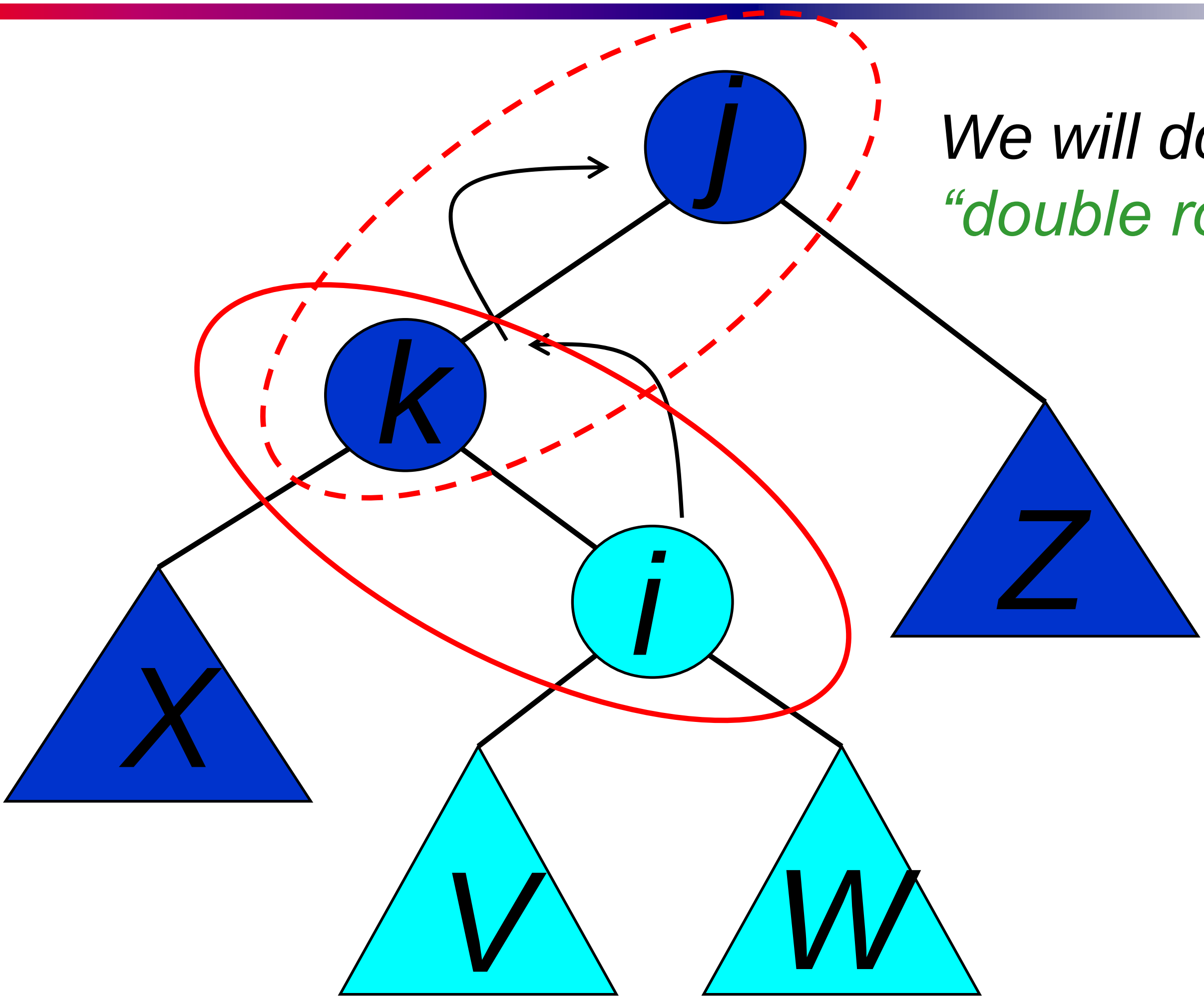


AVL Insertion: Inside Case

Y = node i and subtrees V and W

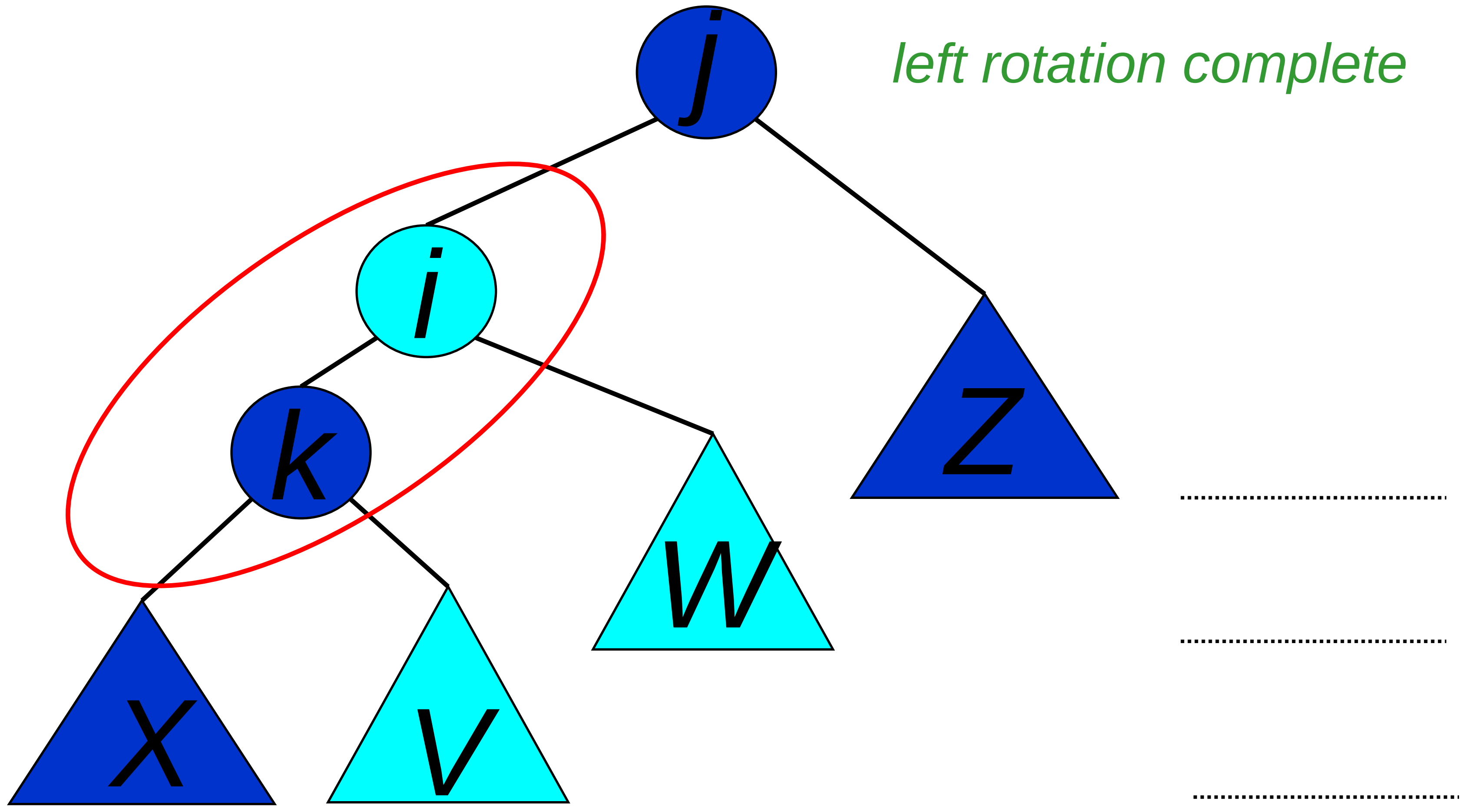


AVL Insertion: Inside Case



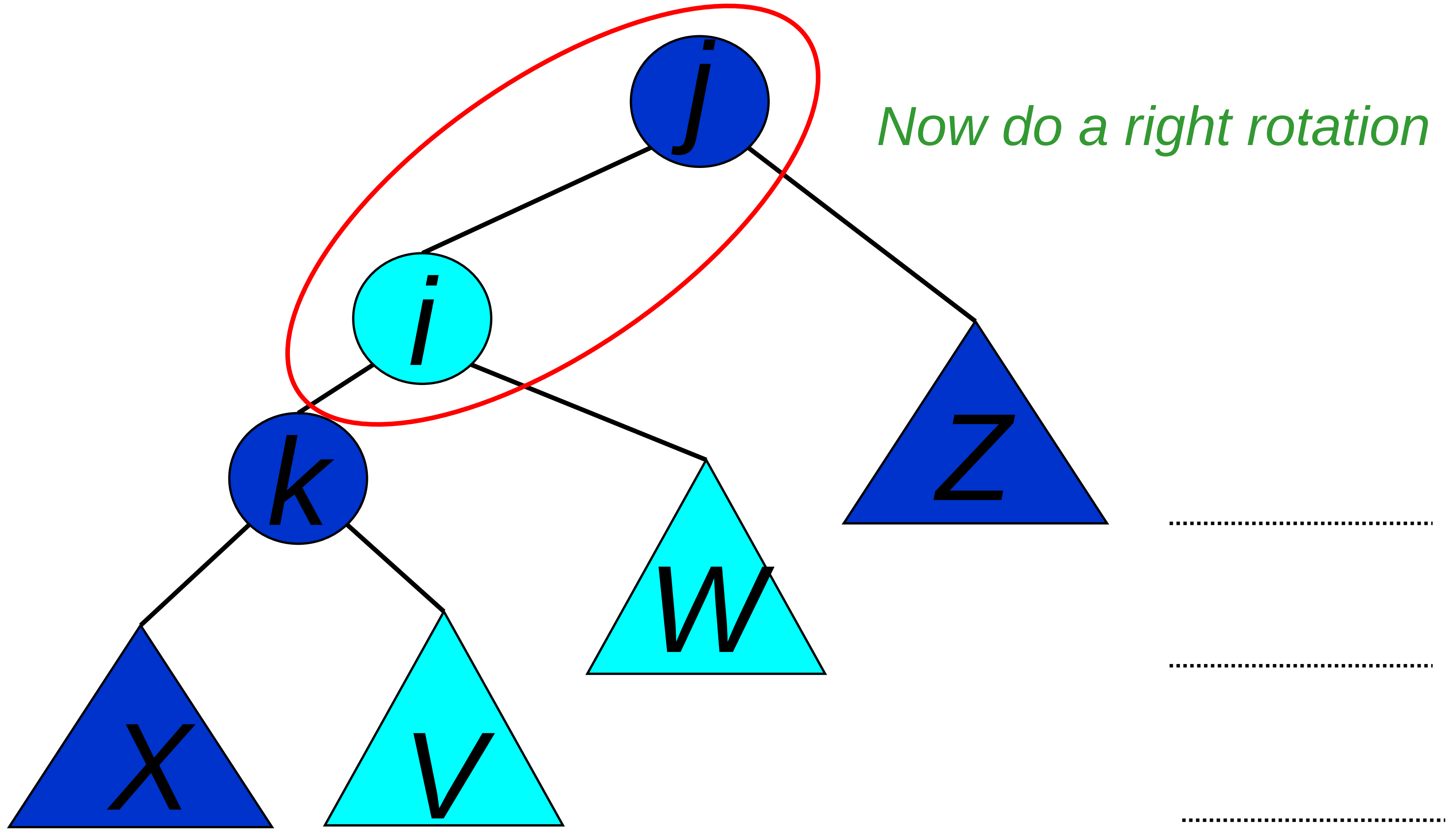
We will do a left-right "double rotation" . . .

Double rotation : first rotation



left rotation complete

Double rotation : second rotation

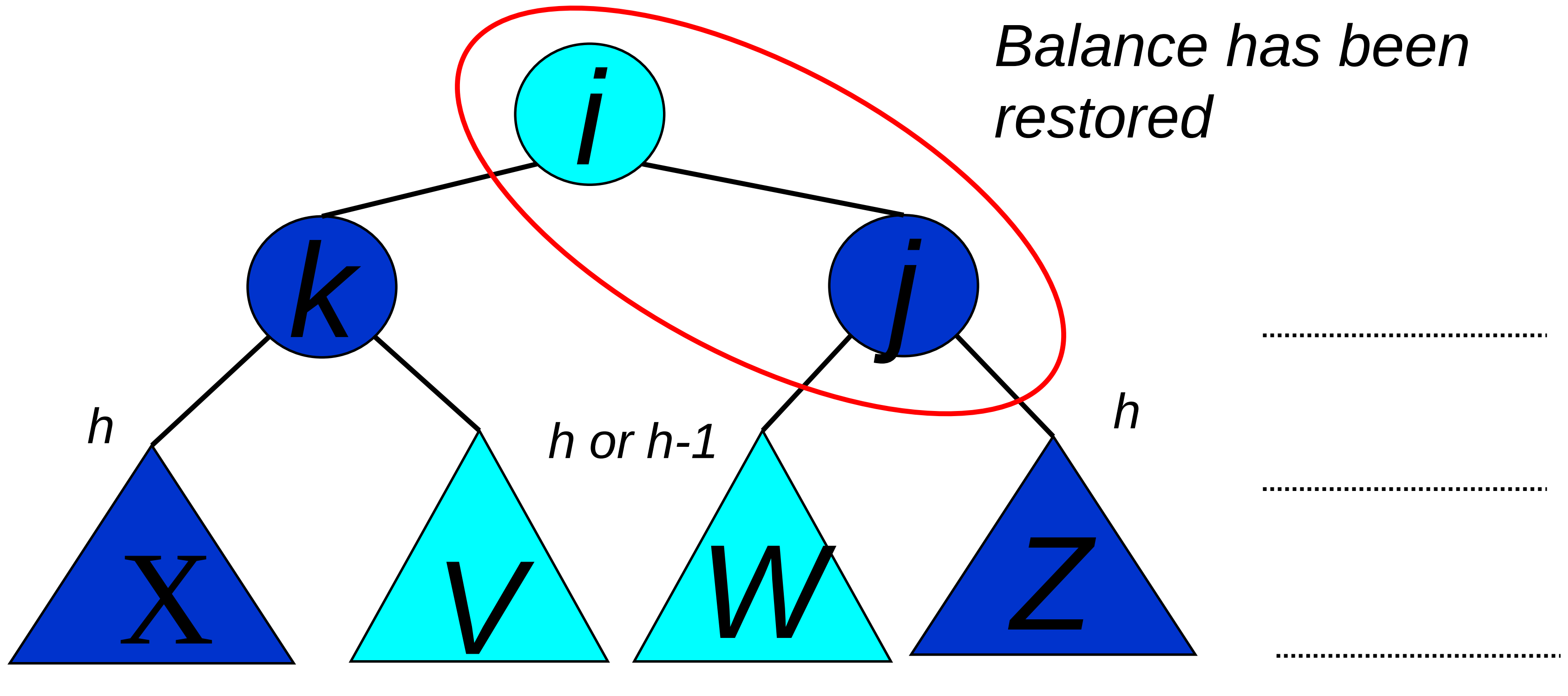


Now do a right rotation

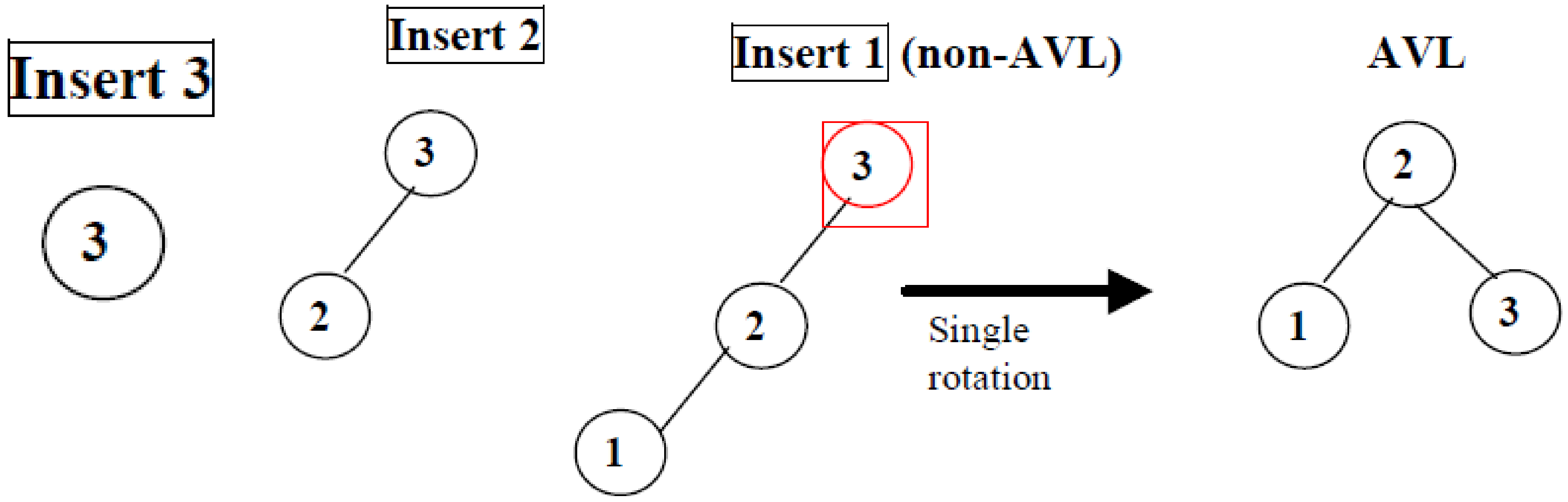
Double rotation : second rotation

right rotation complete

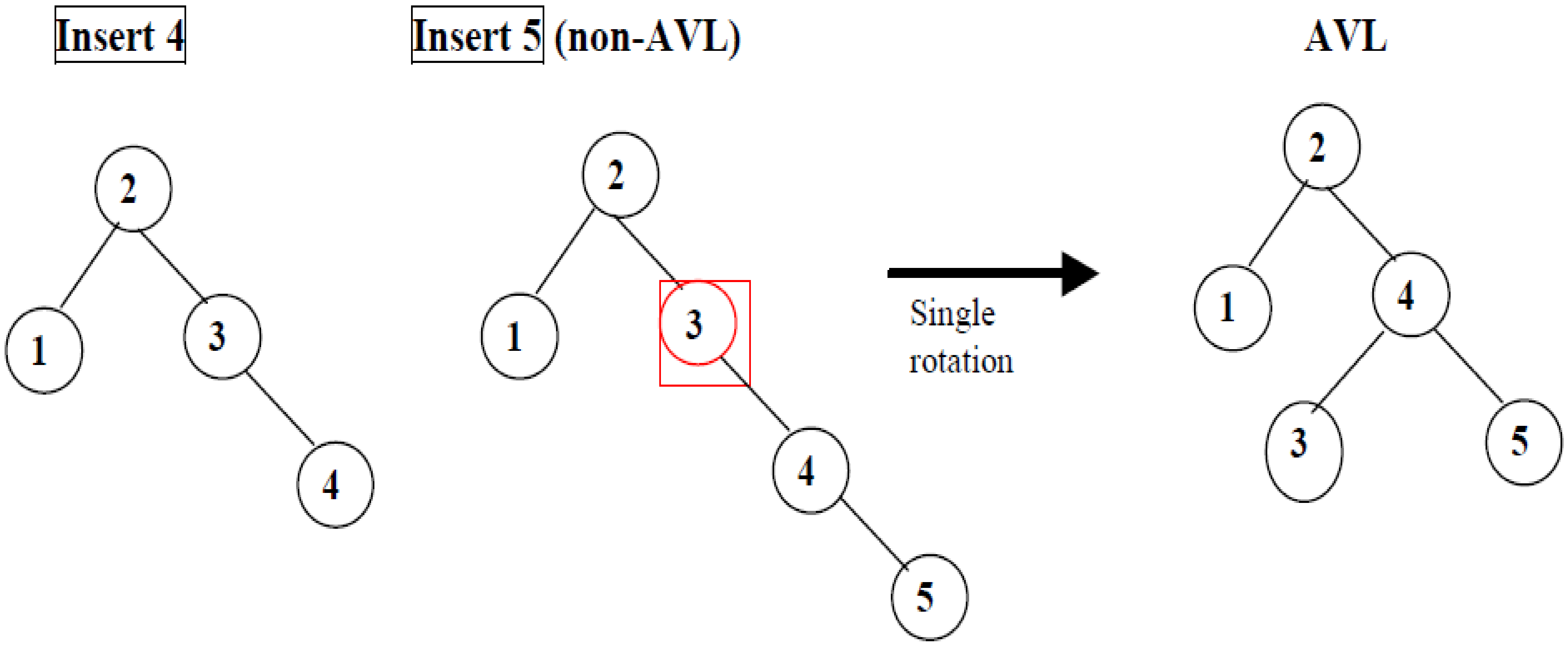
Balance has been restored



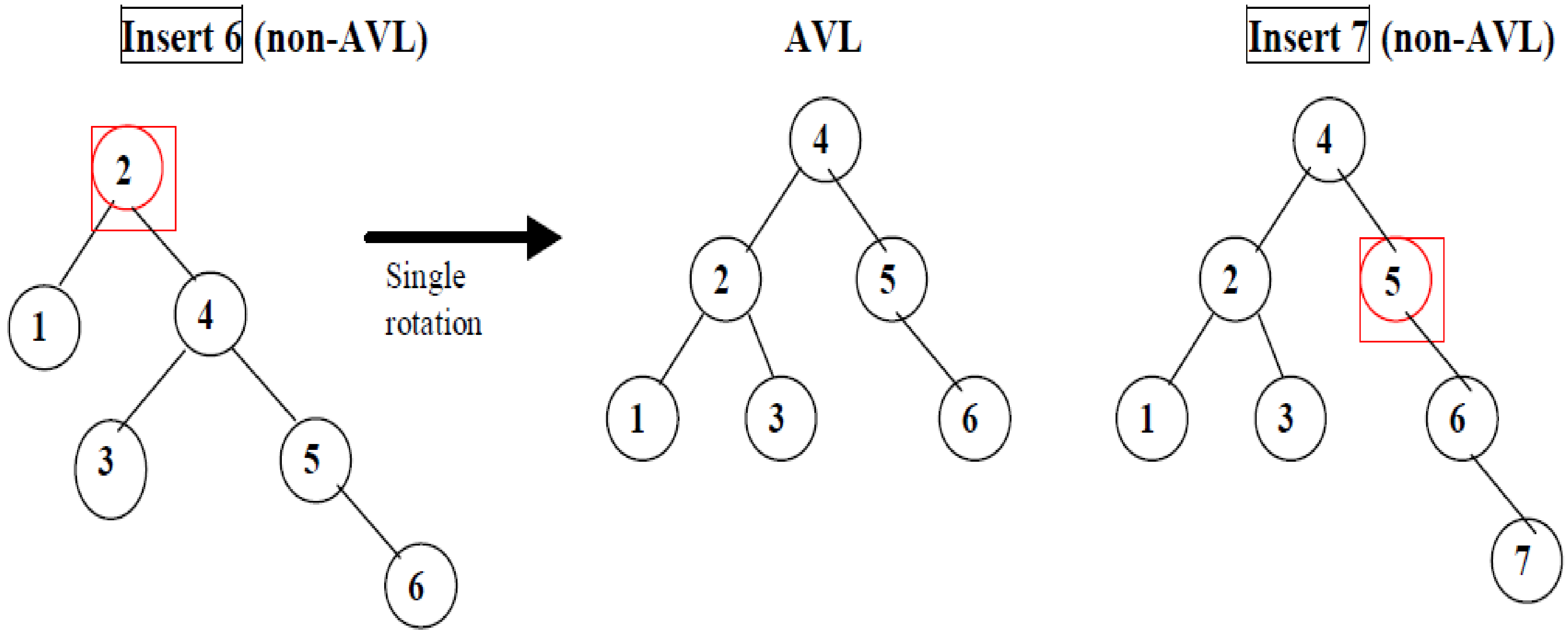
AVL Trees Example



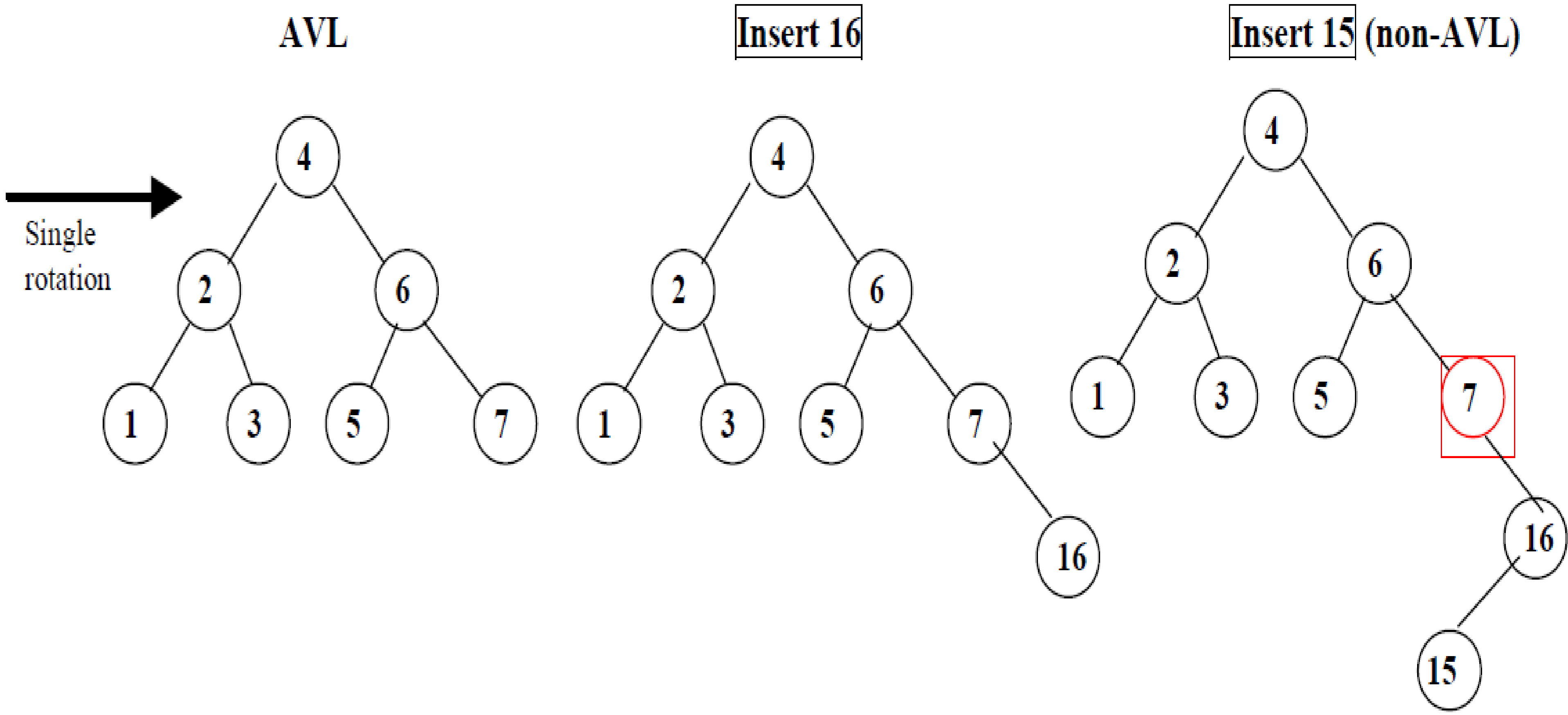
AVL Trees Example



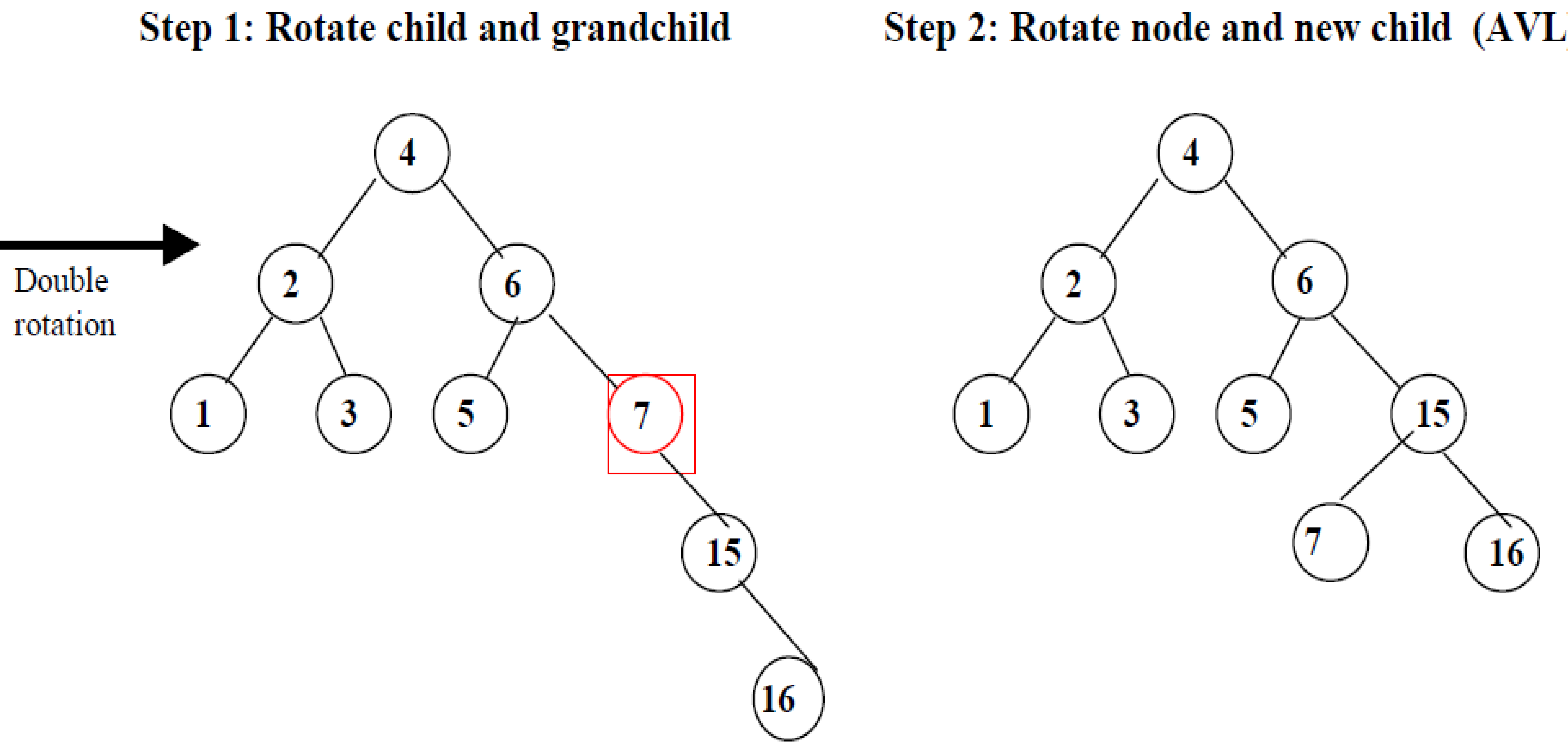
AVL Trees Example



AVL Trees Example

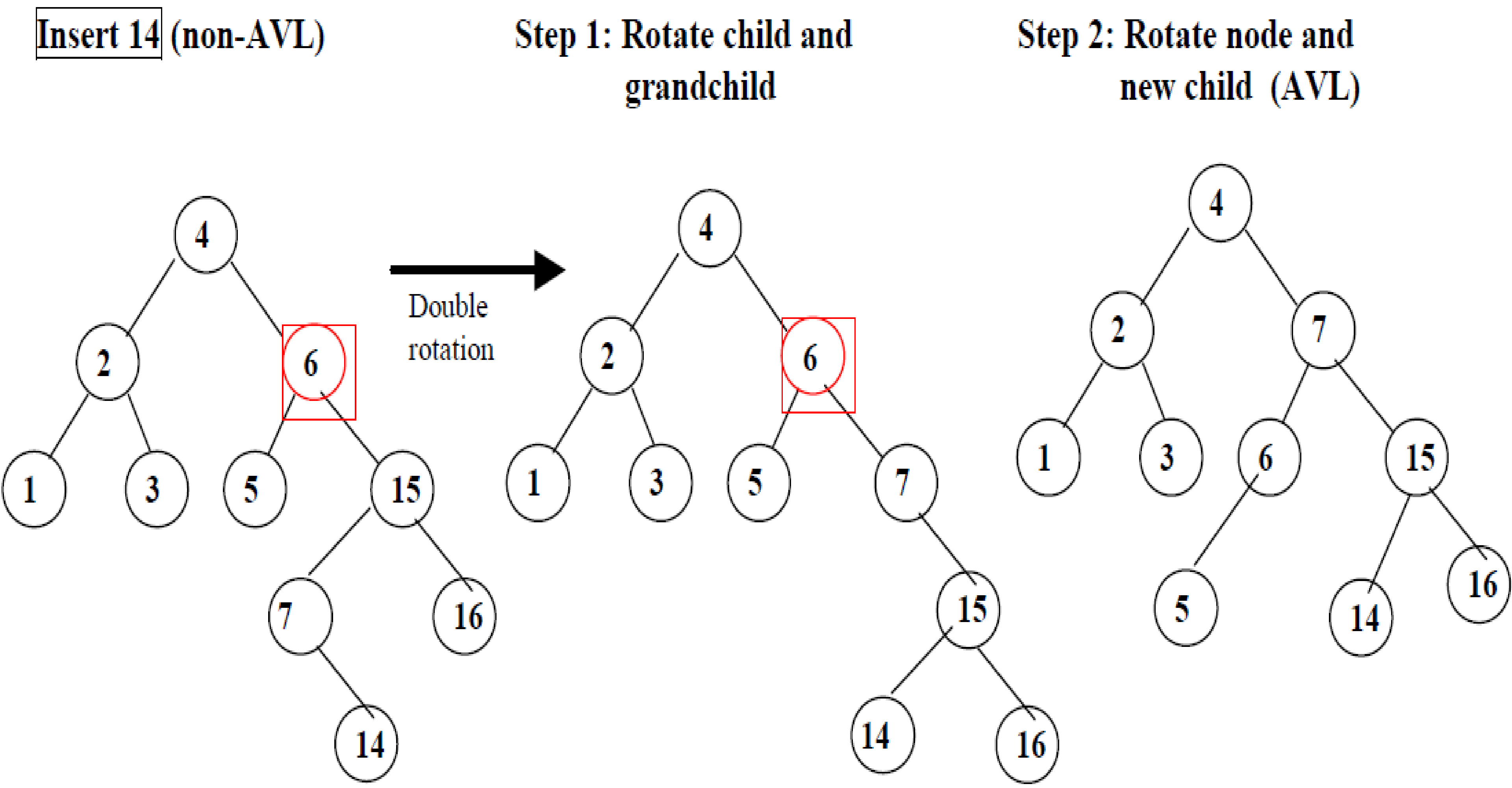


AVL Trees Example



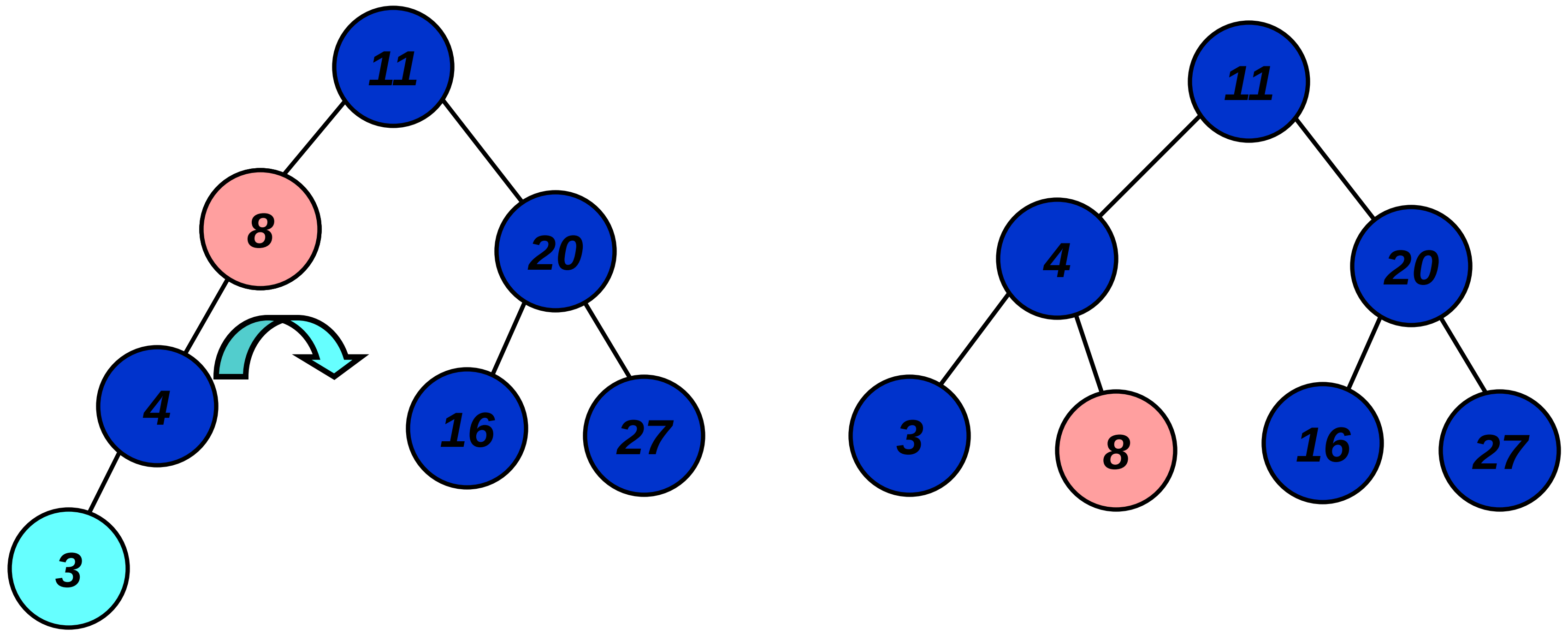
Step 1: Rotate child and grandchild
Step 2: Rotate node and new child (AVL
Double rotation
4
2
6
1
3
5
7
15
16
4
2
6
1
3
5
15
7
16

AVL Trees Example


Insert 14 (non-AVL)
Step 1: Rotate child and grandchild
Step 2: Rotate node and new child (AVL)
Double rotation
4
2
6
1
3
5
15
7
16
14
4
2
6
1
3
5
7
15
14
16
4
2
7
1
3
6
15
5
14
16

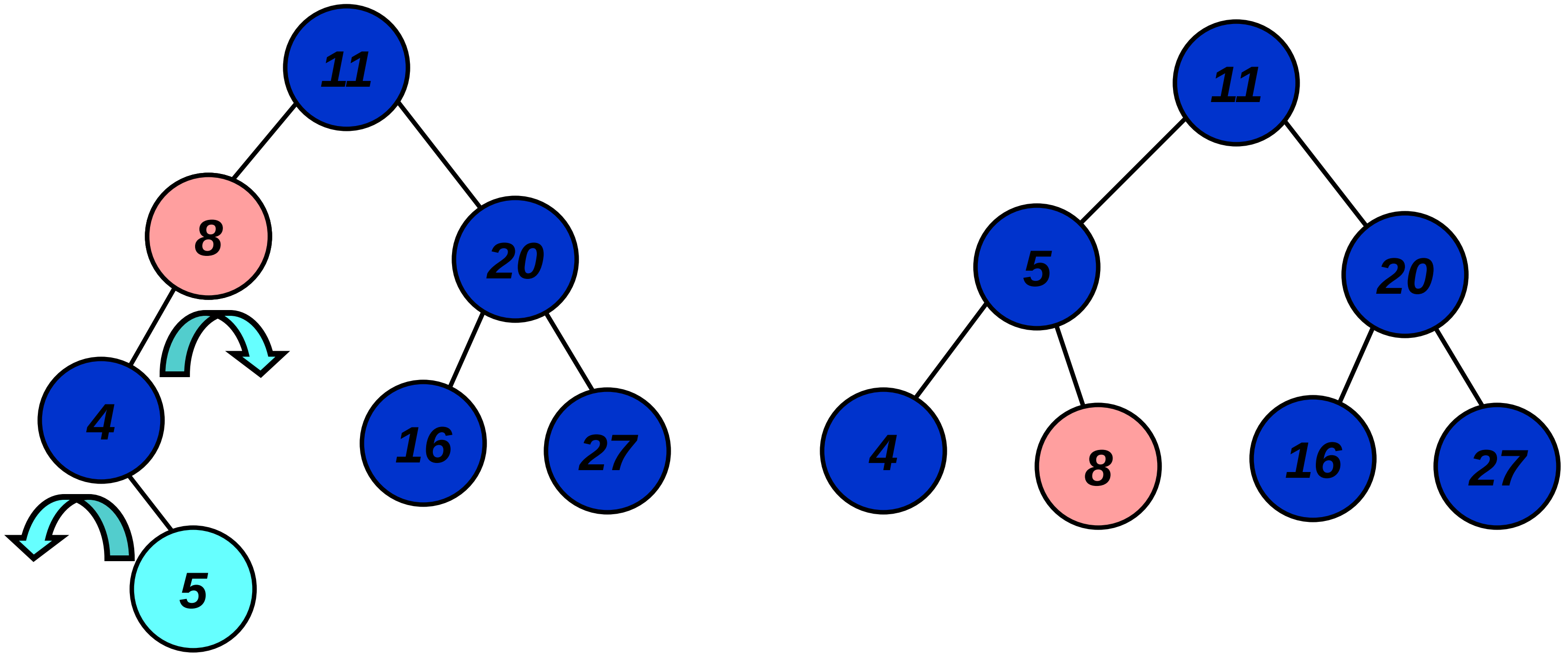
Example

- Insert 3 into the AVL tree



Example

- Insert 5 into the AVL tree



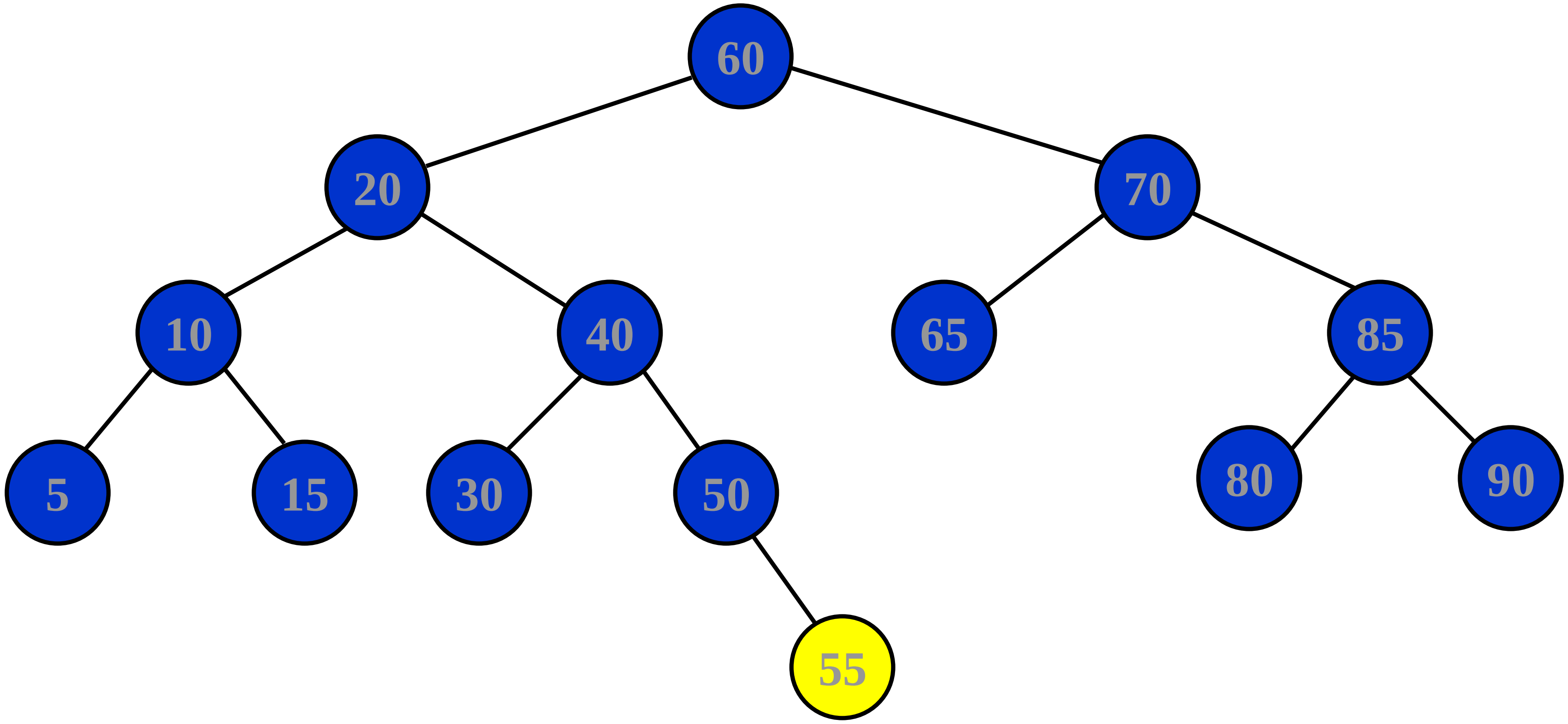
AVL Trees: Exercise

- Insertion order:
  - 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55

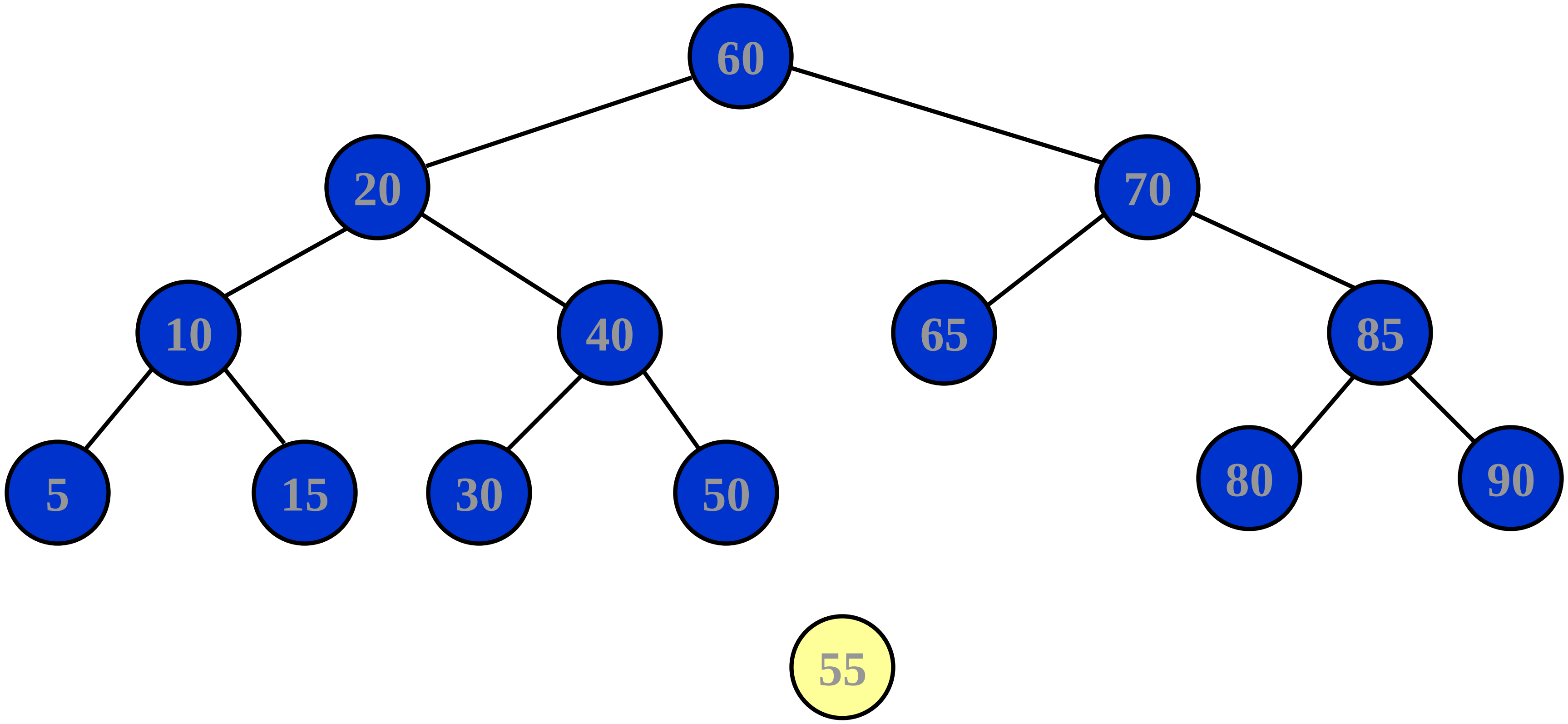
Deletion X in AVL Trees

- Deletion:
  - Case 1: if X is a leaf, delete X
  - Case 2: if X has 1 child, use it to replace X
  - Case 3: if X has 2 children, replace X with its inorder predecessor (and recursively delete it)
- Rebalancing

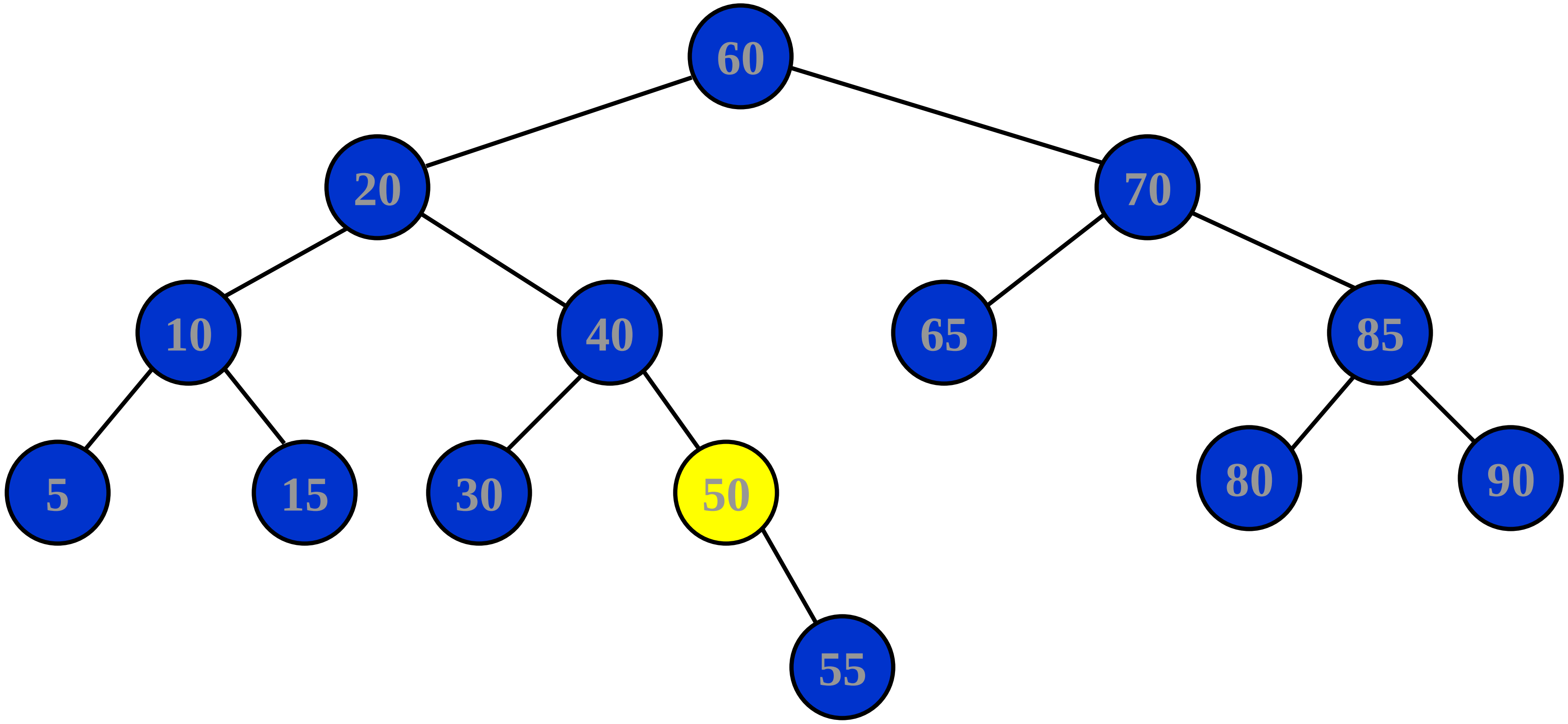
Delete 55 (case 1)



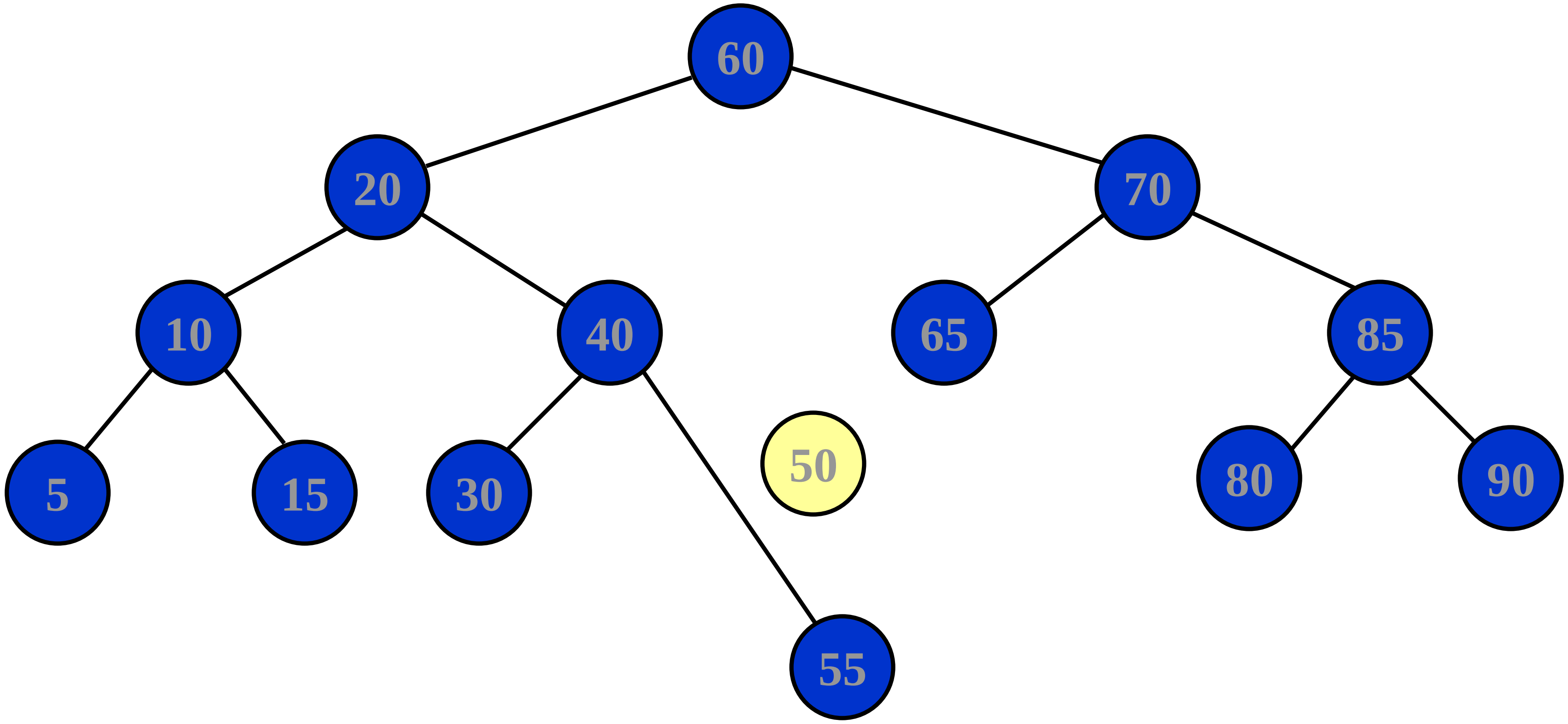
Delete 55 (case 1)



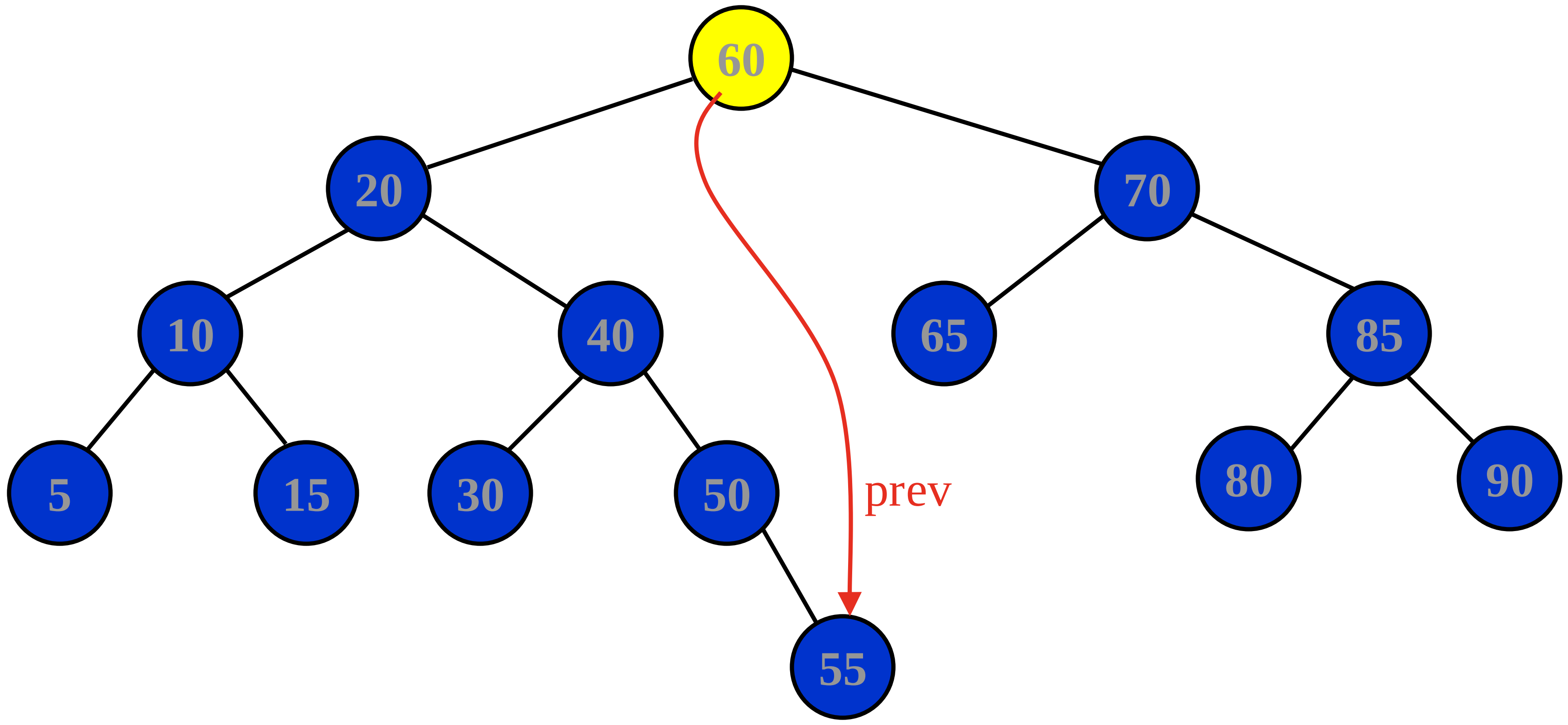
Delete 50 (case 2)



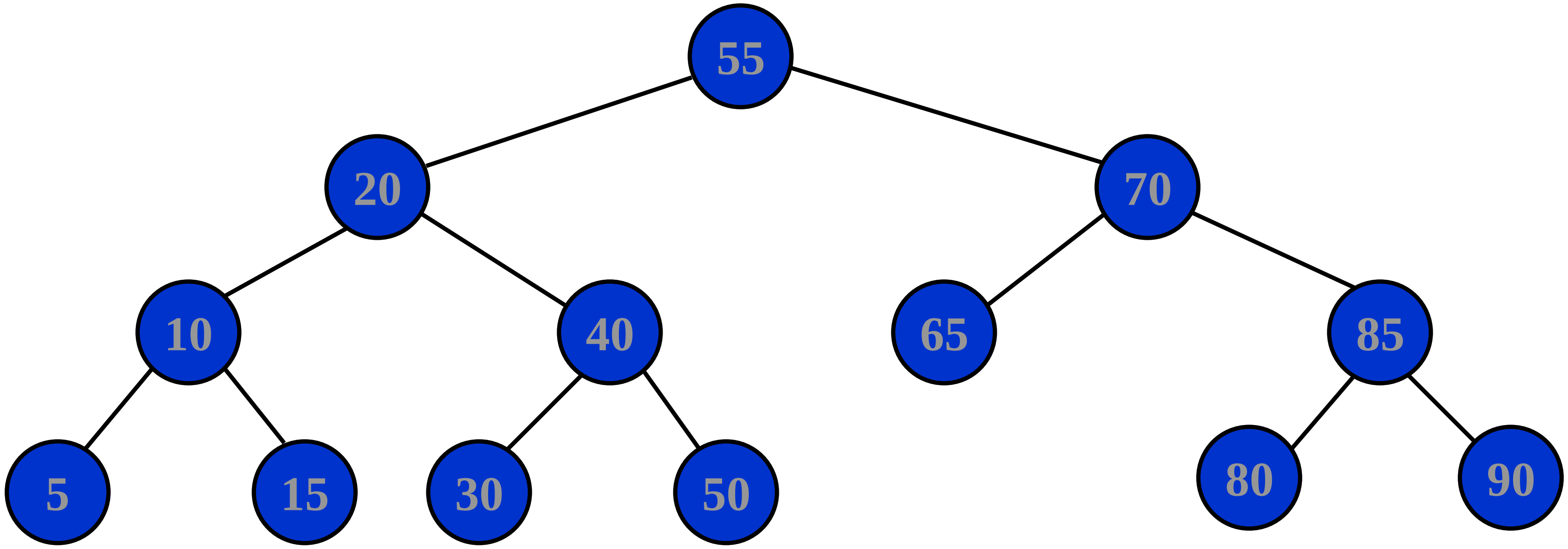
Delete 50 (case 2)



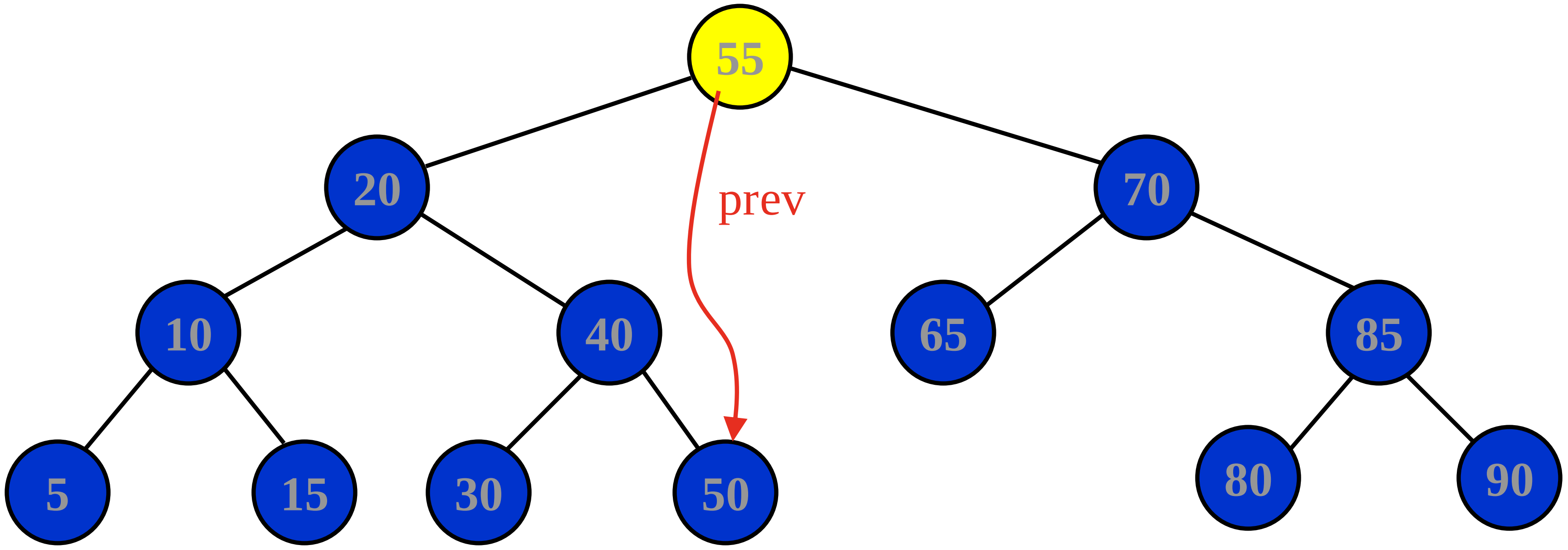
Delete 60 (case 3)



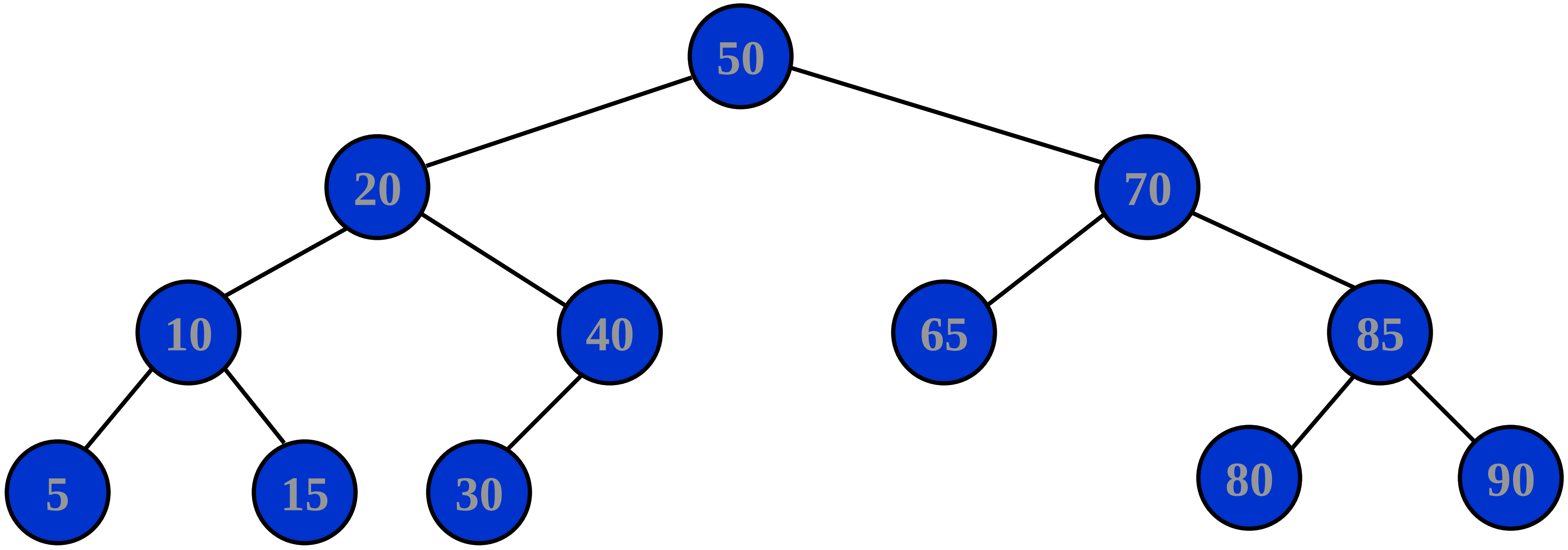
Delete 60 (case 3)



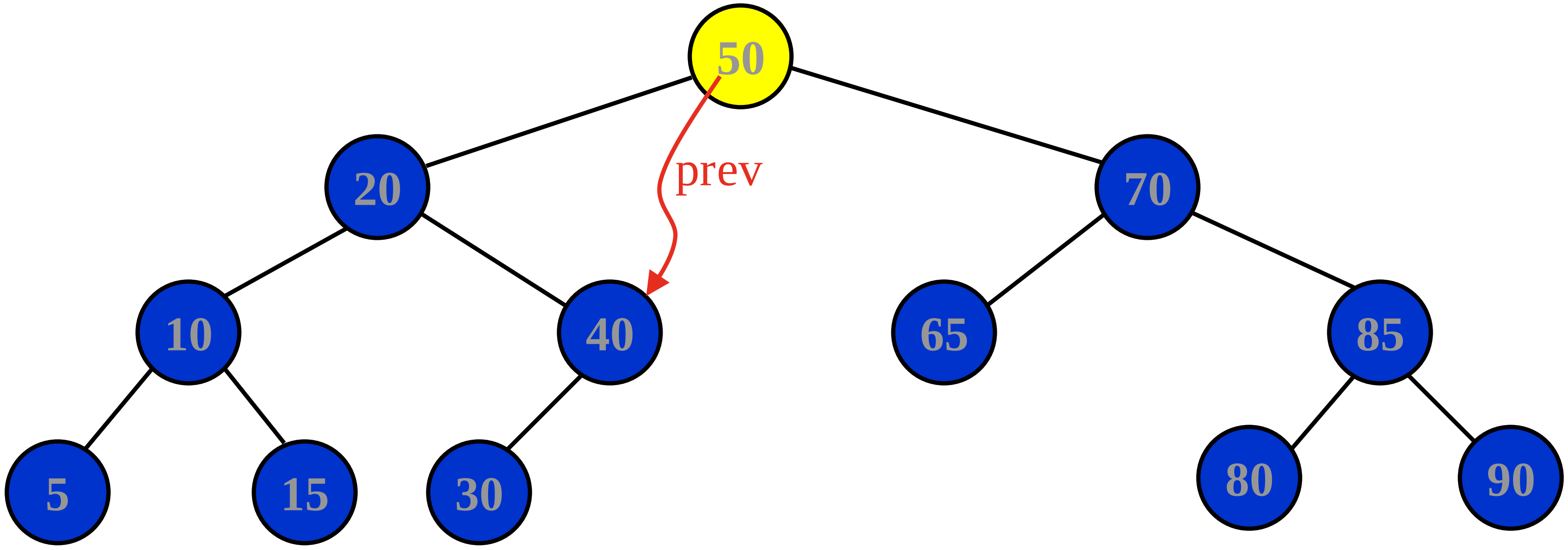
Delete 55 (case 3)



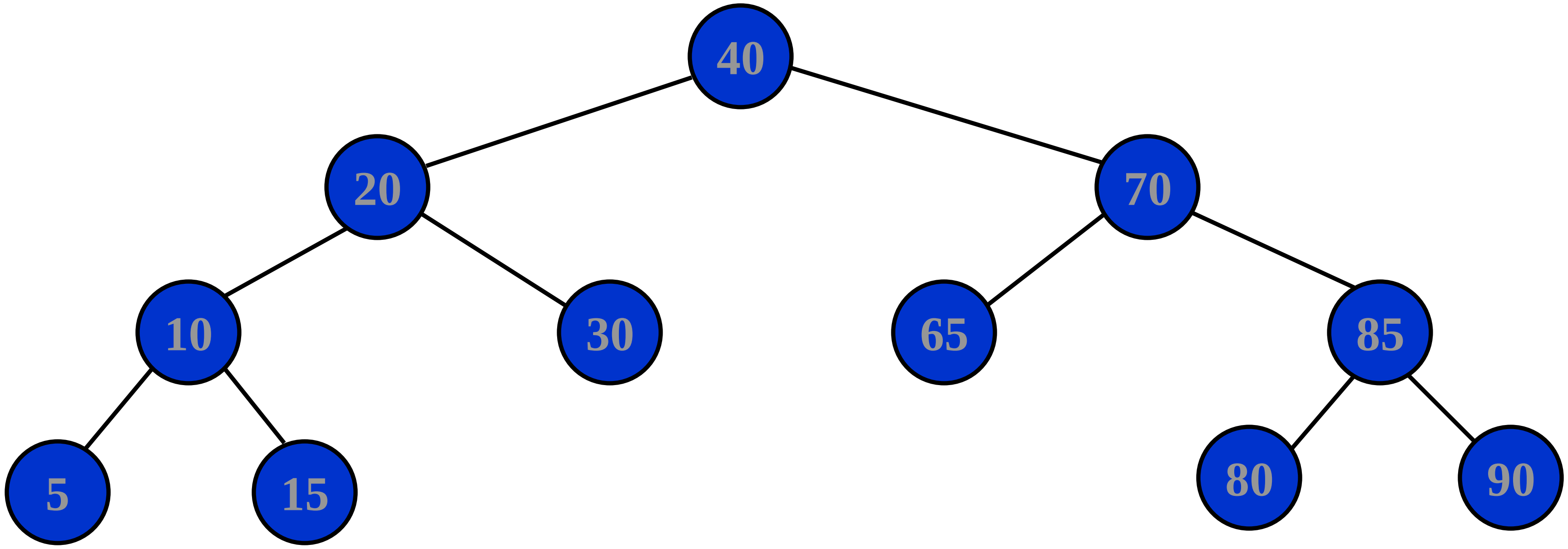
Delete 55 (case 3)



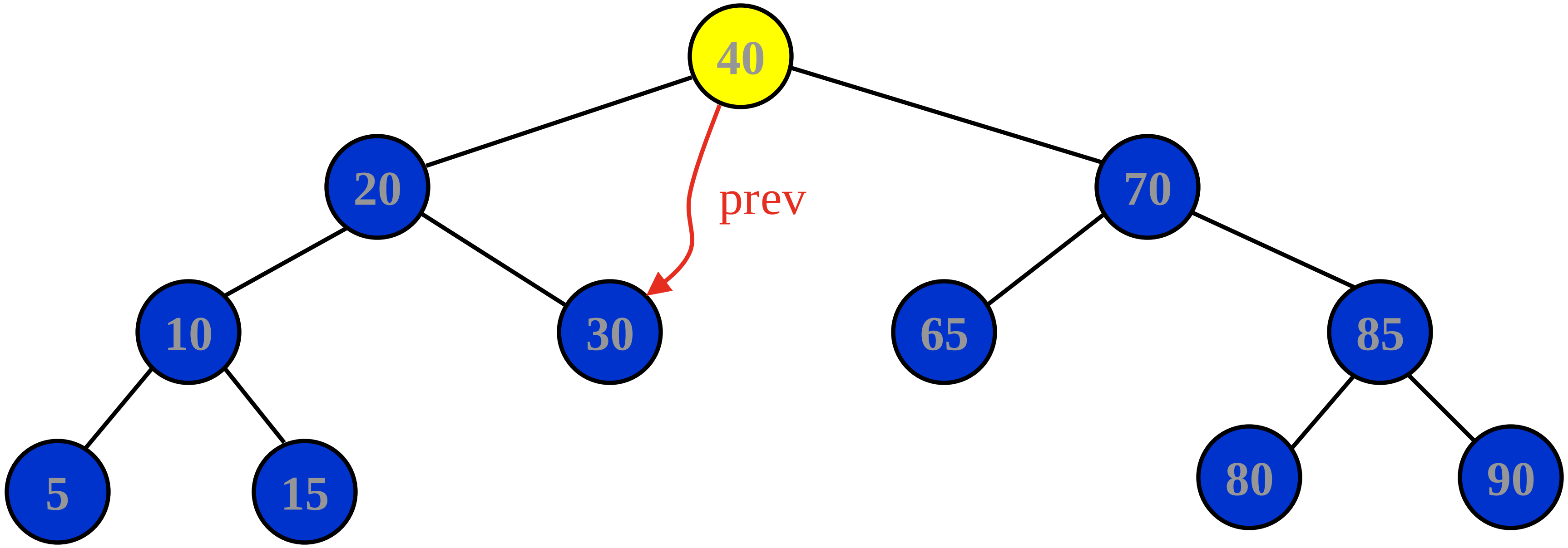
Delete 50 (case 3)



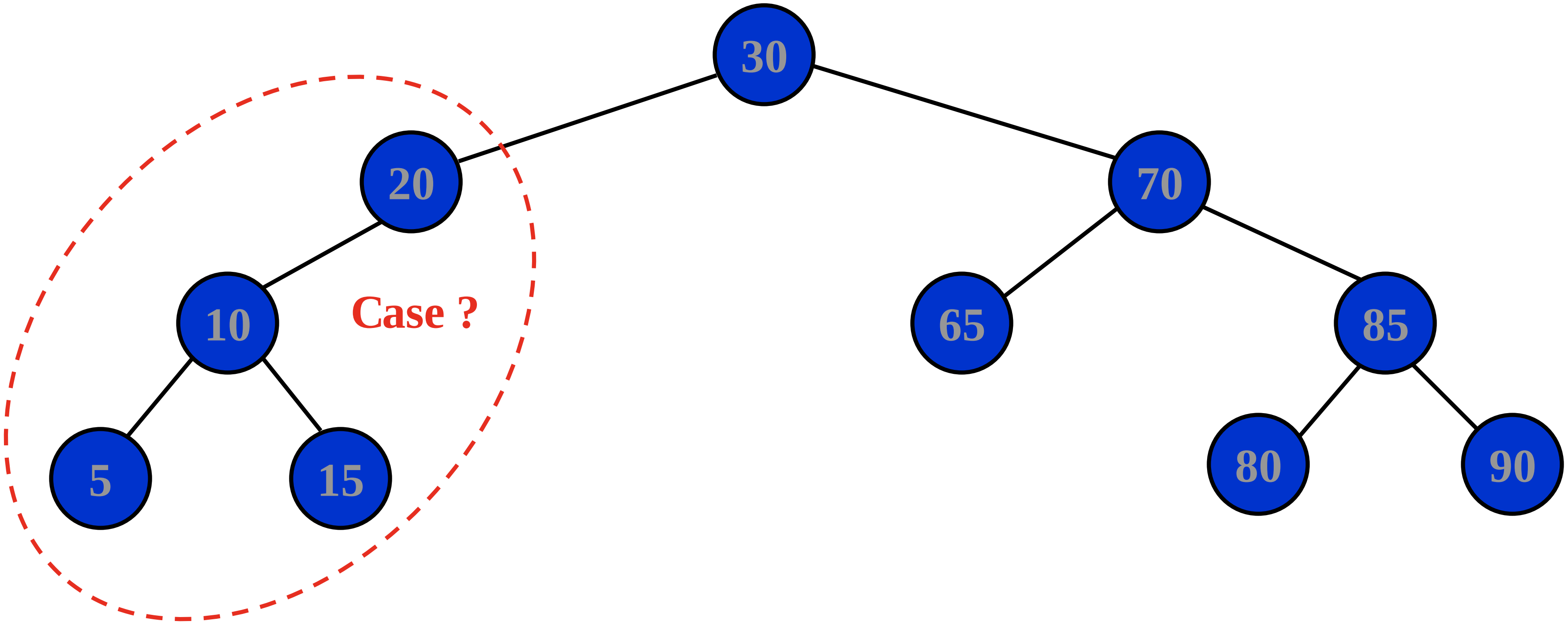
Delete 50 (case 3)



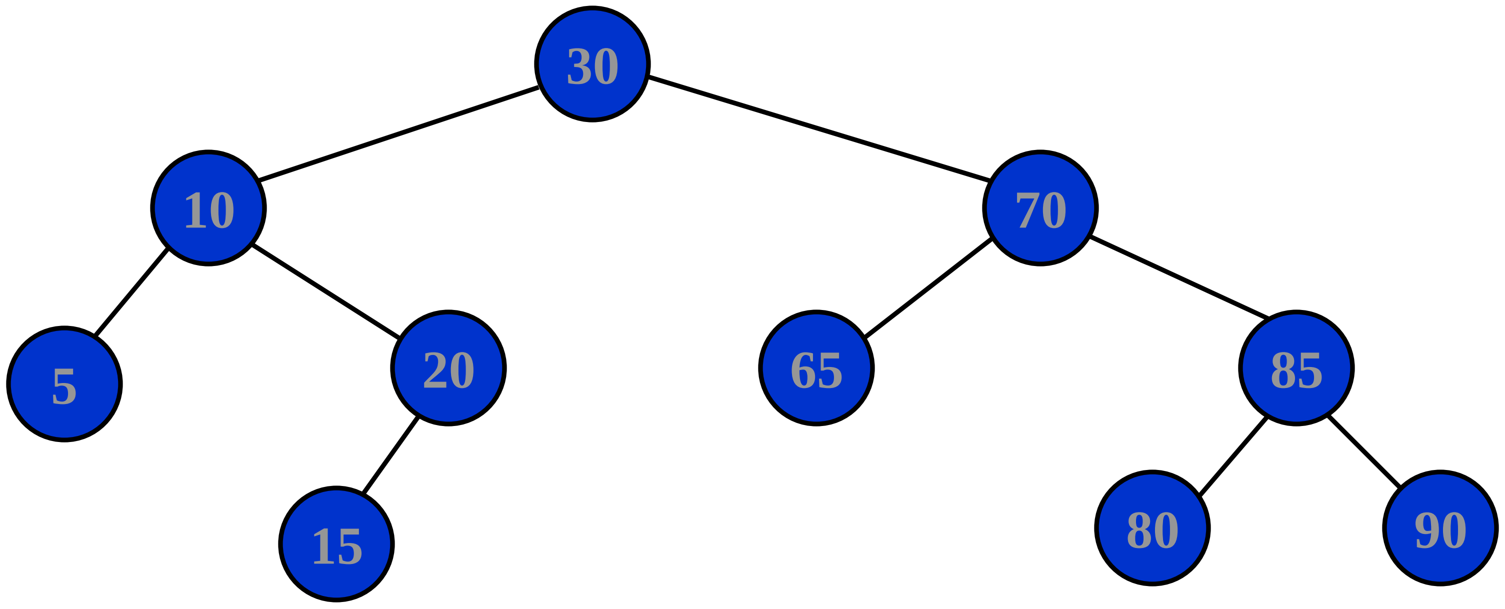
Delete 40 (case 3)



Delete 40 : Rebalancing



Delete 40: after rebalancing



Single rotation is preferred!

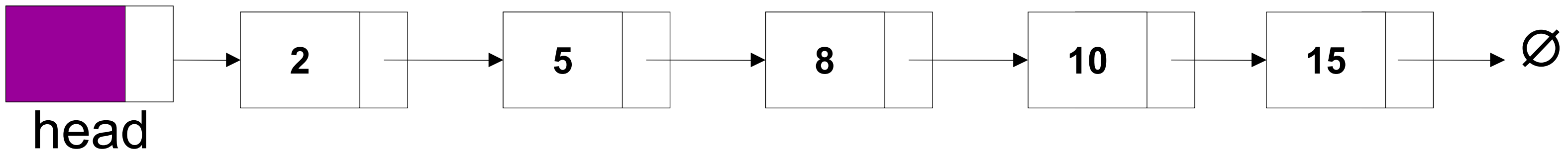
AVL Tree: analysis

- The depth of AVL Trees is at most logarithmic.
- So, all of the operations on AVL trees are also logarithmic.
- The worst-case height is at most 44 percent more than the minimum possible for binary trees.

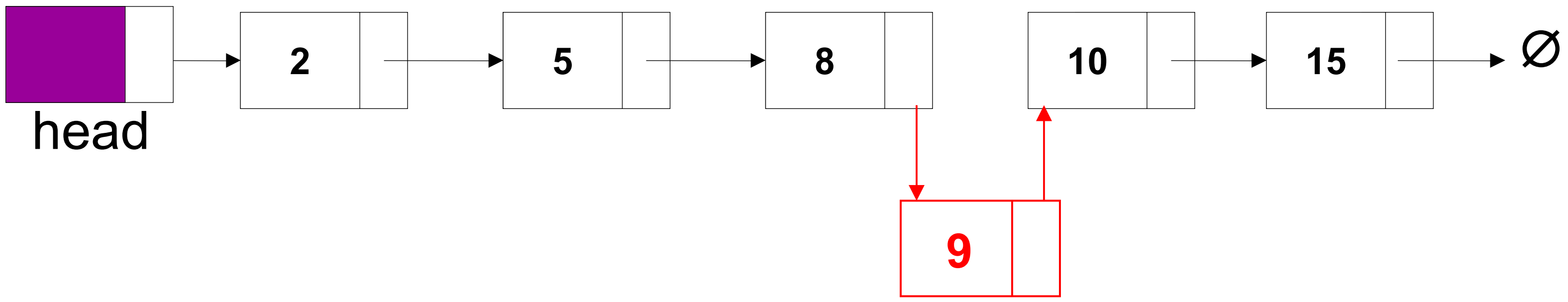
HEAPS

Priority Queues

Linked-list



- Insert



Priority Queues

Supports the following operations.

Insert element x.

Return min/max element.

Return and delete minimum/maximum element.

Increase/Decrease key of element x to k.

Max/Min-HEAPIFY

BUILD-Max/Min-HEAP

HEAPSORT

Applications.

Dijkstra's shortest path algorithm.

Prim's MST algorithm.

Event-driven simulation.

Huffman encoding.

Heapsort.

. . .

Time Complexity

Insert:	O(n)
Delete:	O(1)
n deletions and insertions:	$O(n^2)$

Heap

A max (min) heap is a complete binary tree such that the data stored in each node is greater (smaller) than the data stored in its children, if any.

Heap Types

Max-heaps (largest element at root), have the *max-heap property:*

for all nodes i, excluding the root:

$$A[\text{PARENT}(i)] \geq A[i]$$

Min-heaps (smallest element at root), have the *min-heap property:*

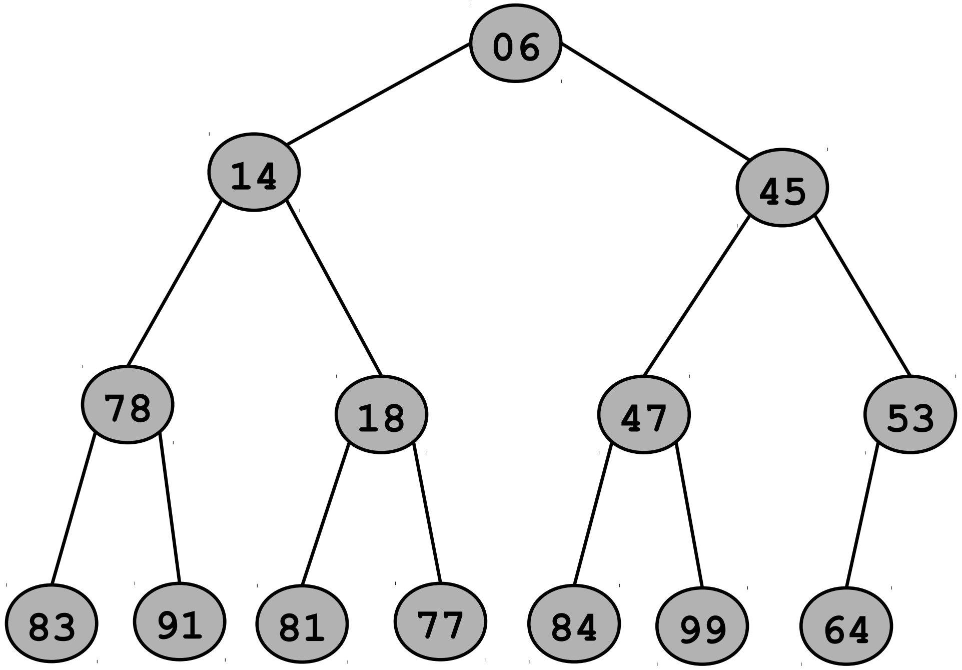
for all nodes i, excluding the root:

$$A[\text{PARENT}(i)] \leq A[i]$$

Binary Heap: Definition

Binary heap.

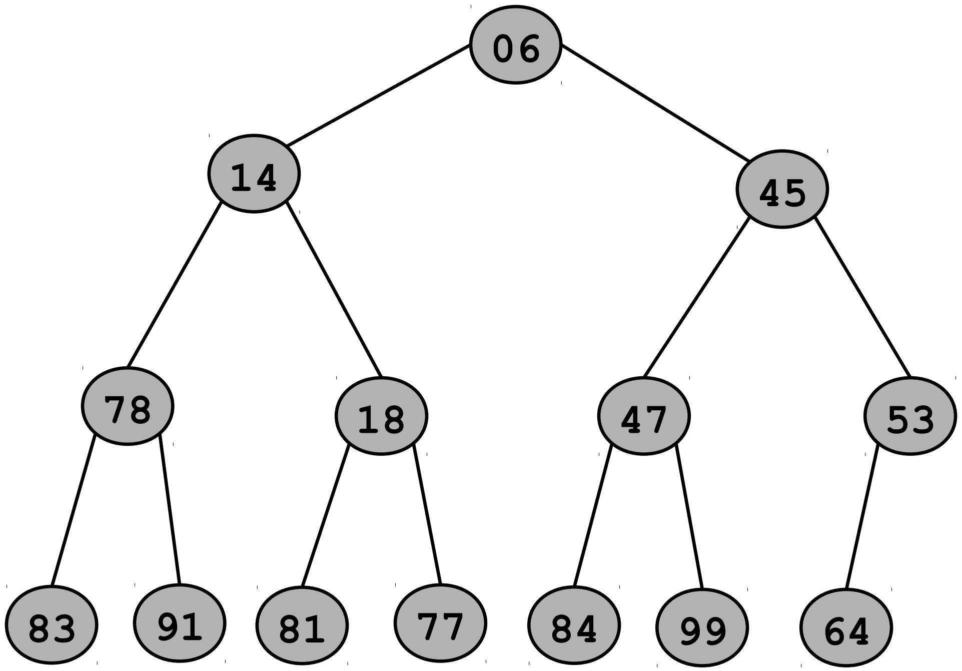
- Almost complete binary tree.
  - filled on all levels, except last, where filled from left to right
- Min-heap ordered.
  - every child greater than (or equal to) parent



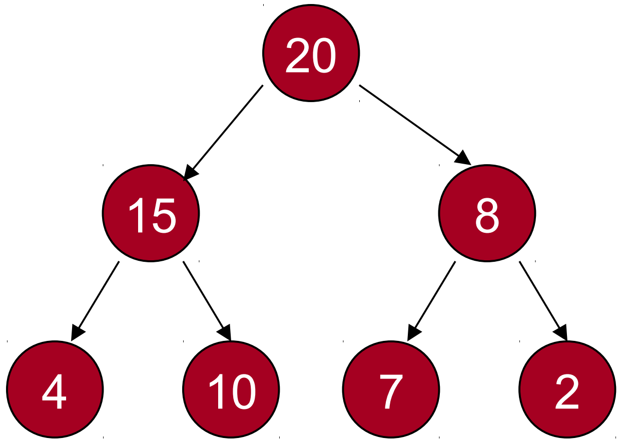
Binary Heap: Properties

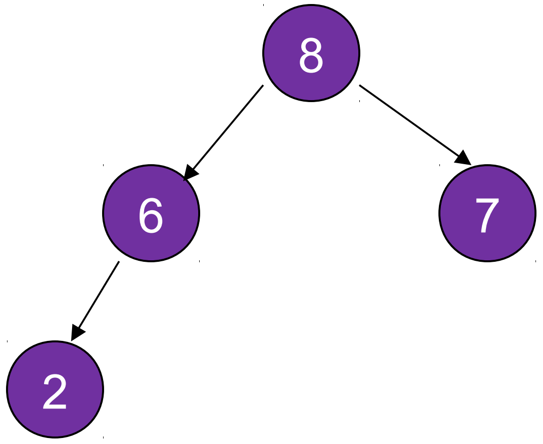
Properties.

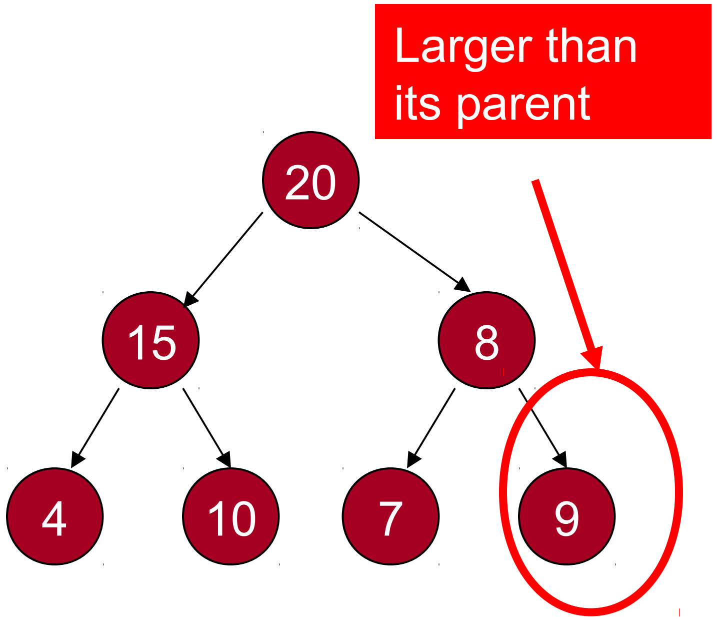
- Min element is in root.
- Heap with N elements has height = $\lfloor \log_2 N \rfloor$.



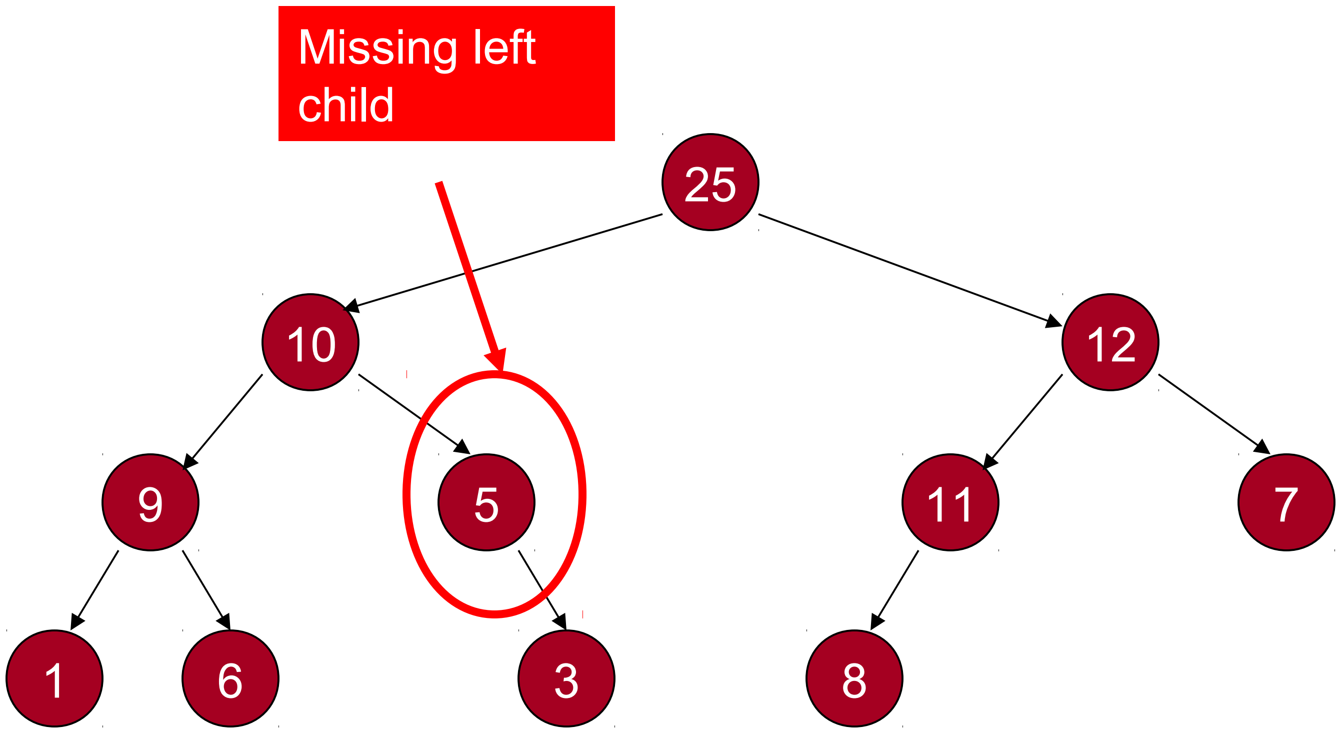
N = 14
Height = 3


20
15
8
4
10
7
2


8
6
7
2

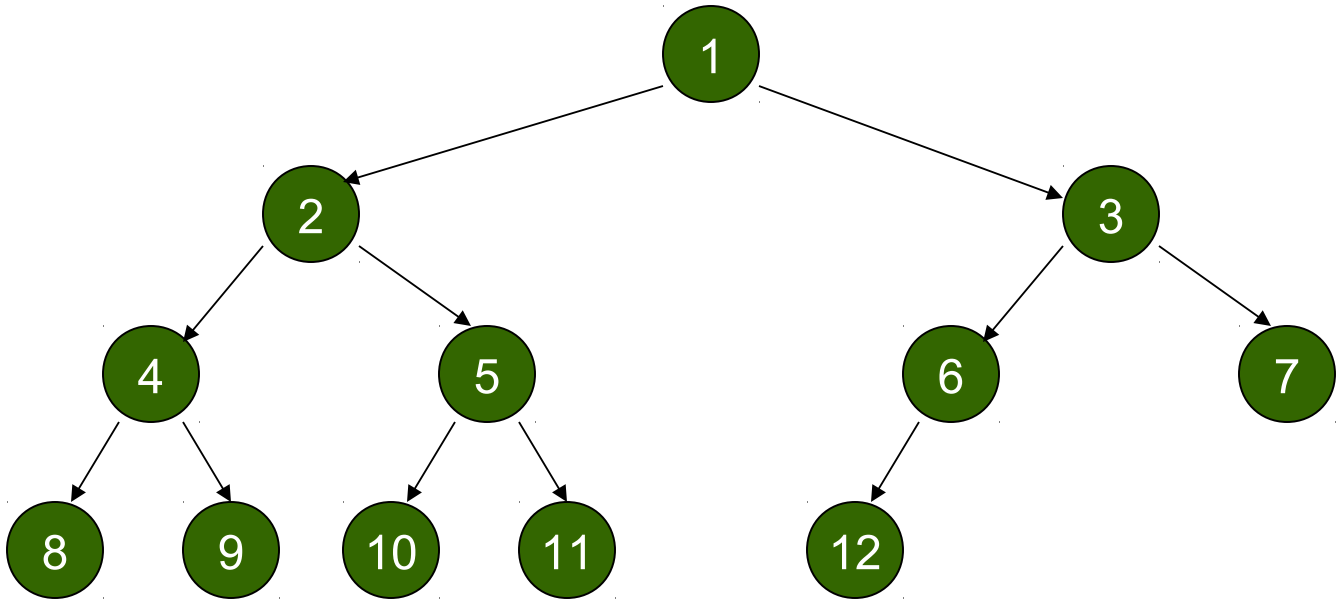


Not a heap



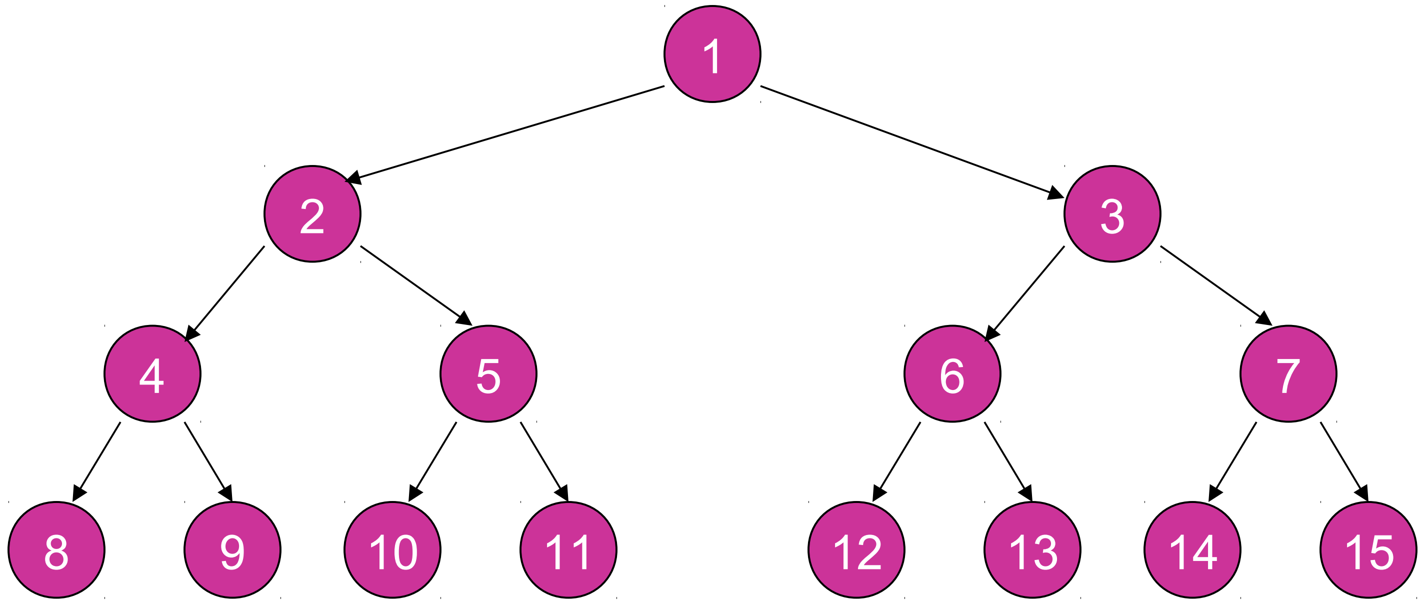
Not a complete tree – Not a heap

Complete Binary Tree



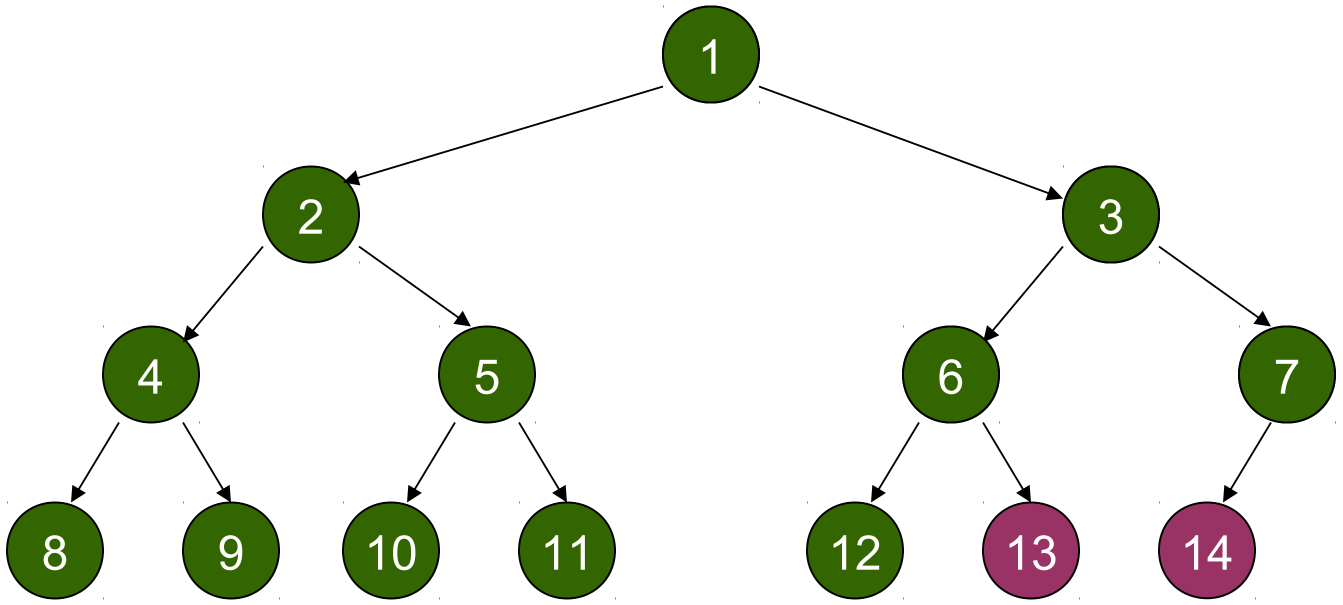
Where to add the next node?

How to find it?



1	0001	6	0110	11	1011
2	0010	7	0111	12	1100
3	0011	8	1000	13	1101
4	0100	9	1001	14	1110
5	0101	10	1010	15	1111

After the first 1 from left: 0 – left; 1 – right



Next 13

1101

Next 14

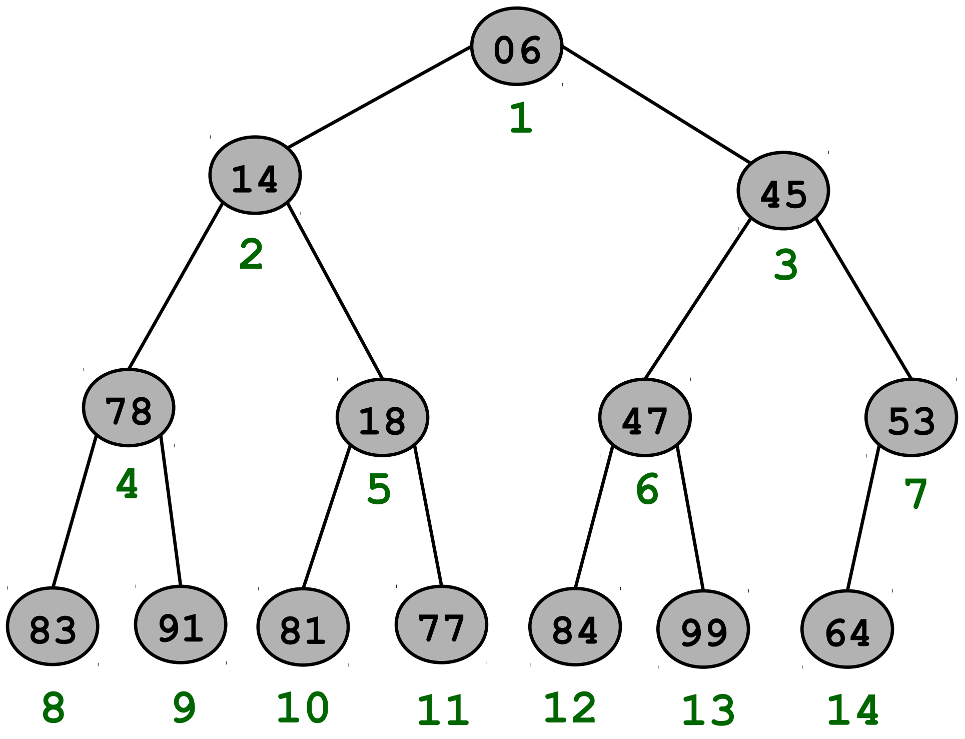
1110

Binary Heaps: Array Implementation

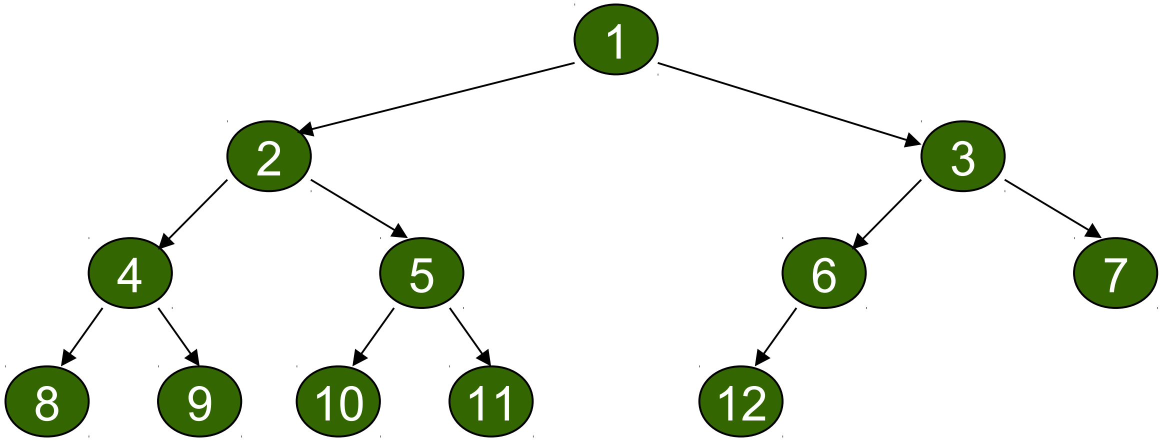
Implementing binary heaps.

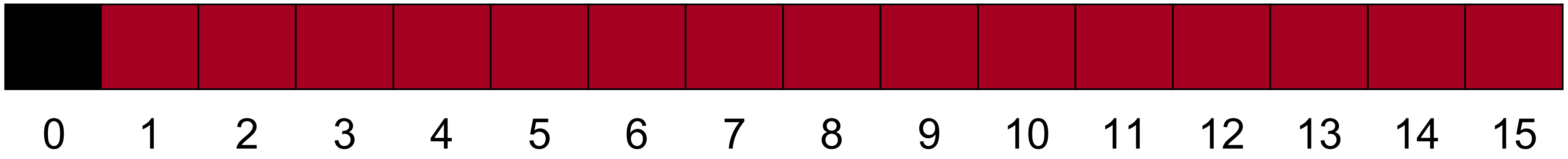
Use an array: no need for explicit parent or child pointers.

- `Parent(i)  = ⌊i/2⌋`
- `Left(i)    = 2i`
- `Right(i)   = 2i + 1`



Storing a Complete Binary Tree in an Array





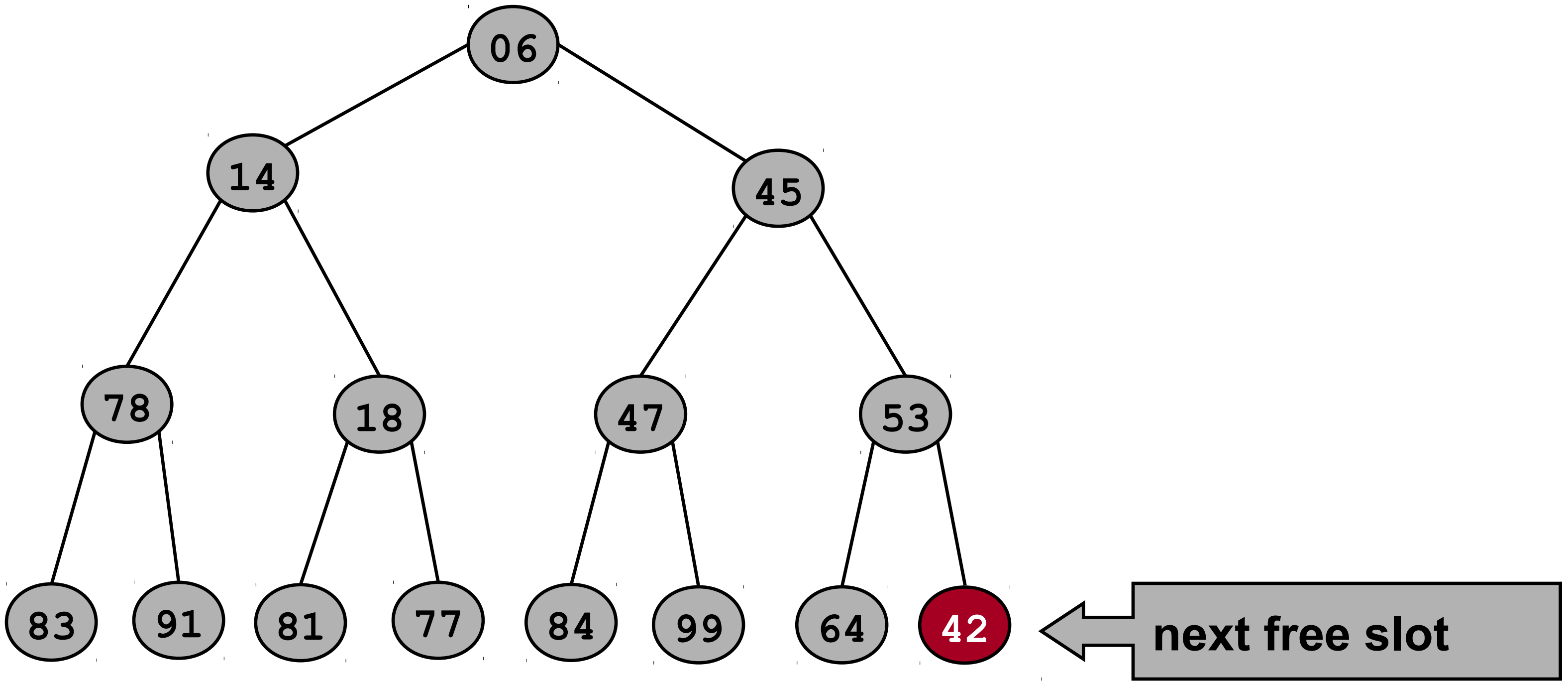
for any node at index i

parent = i/2 left child = i * 2 right child = i*2+1

Binary Heap: Insertion

Insert element x into heap.

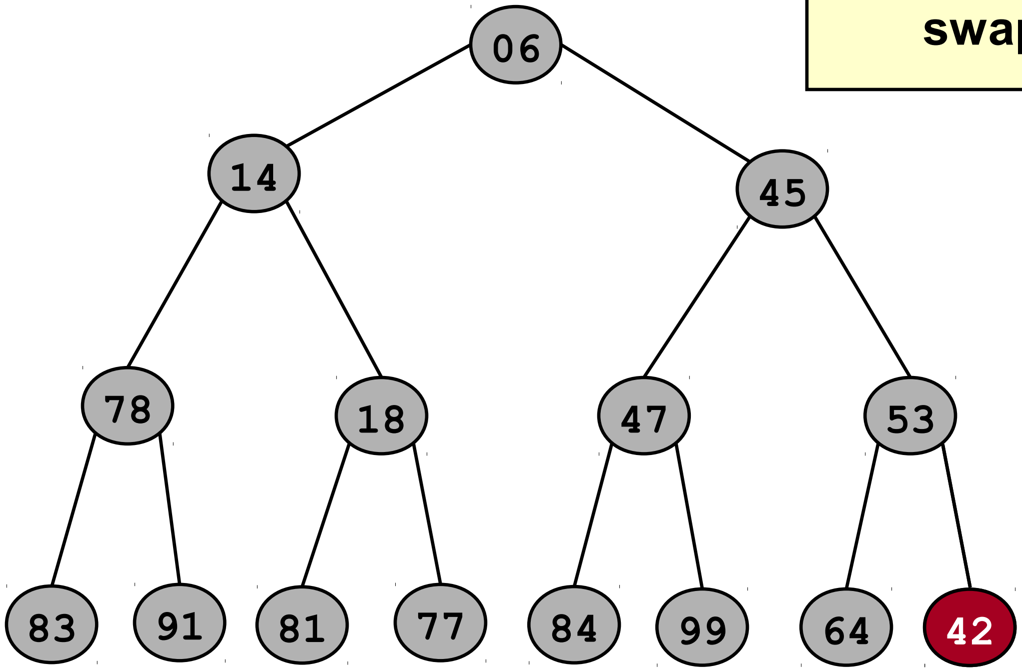
- Insert into next available slot.
- Bubble up until it's heap ordered.
  - Peter principle: nodes rise to level of incompetence



Binary Heap: Insertion

Insert element x into heap.

- Insert into next available slot.
- Bubble up until it's heap ordered.
  - Peter principle: nodes rise to level of incompetence



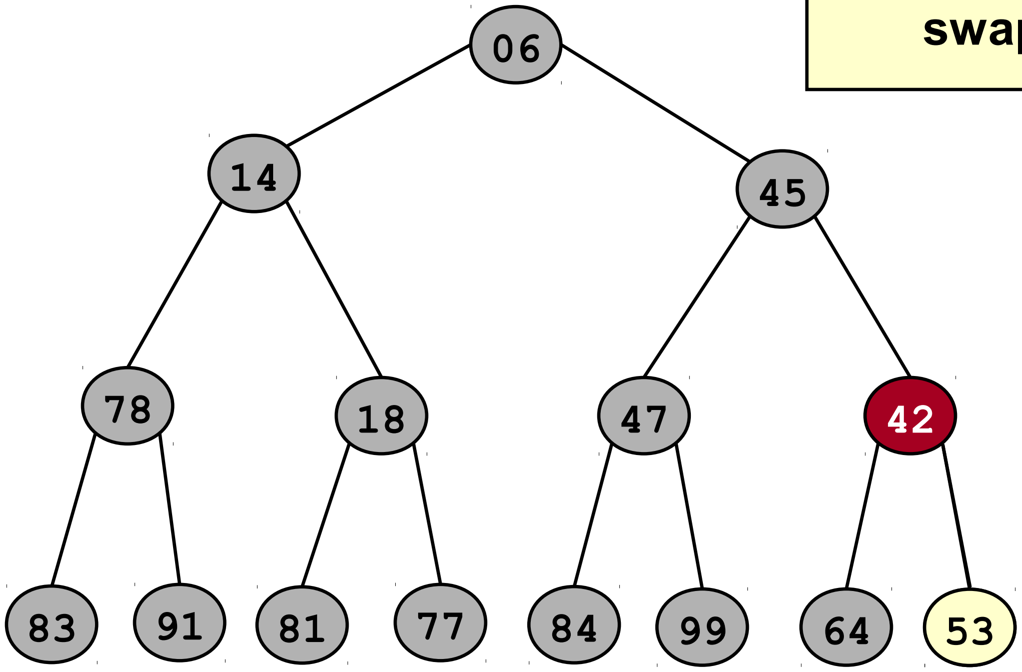
swap with parent

Binary Heap: Insertion

Insert element x into heap.

- Insert into next available slot.
- Bubble up until it's heap ordered.
  - Peter principle: nodes rise to level of incompetence

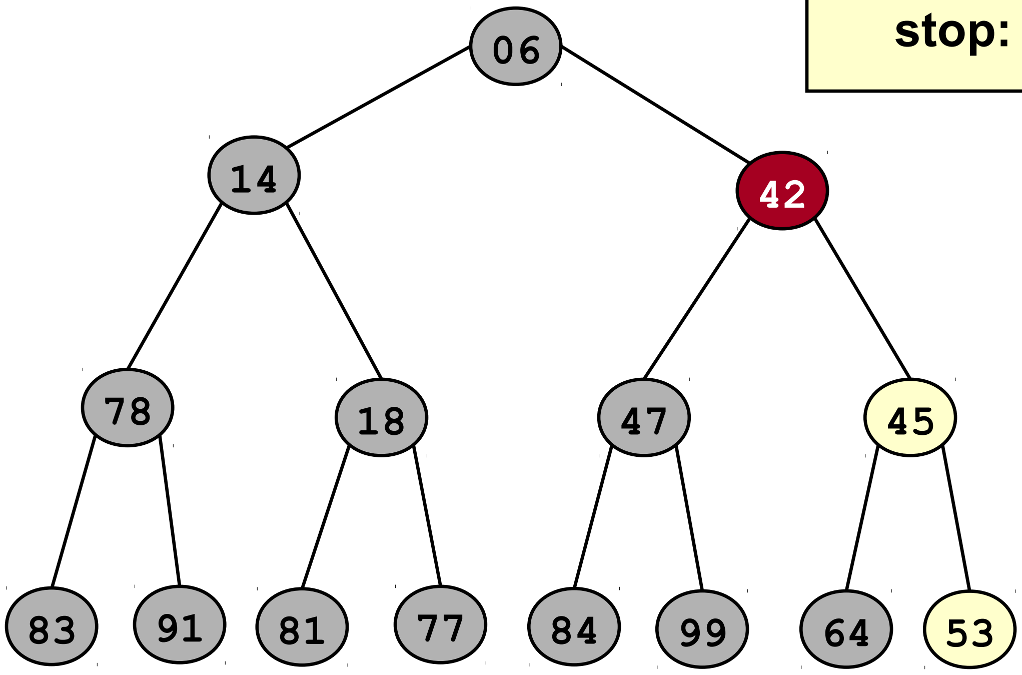
swap with parent



Binary Heap: Insertion

Insert element x into heap.

- Insert into next available slot.
- Bubble up until it's heap ordered.
  - Peter principle: nodes rise to level of incompetence



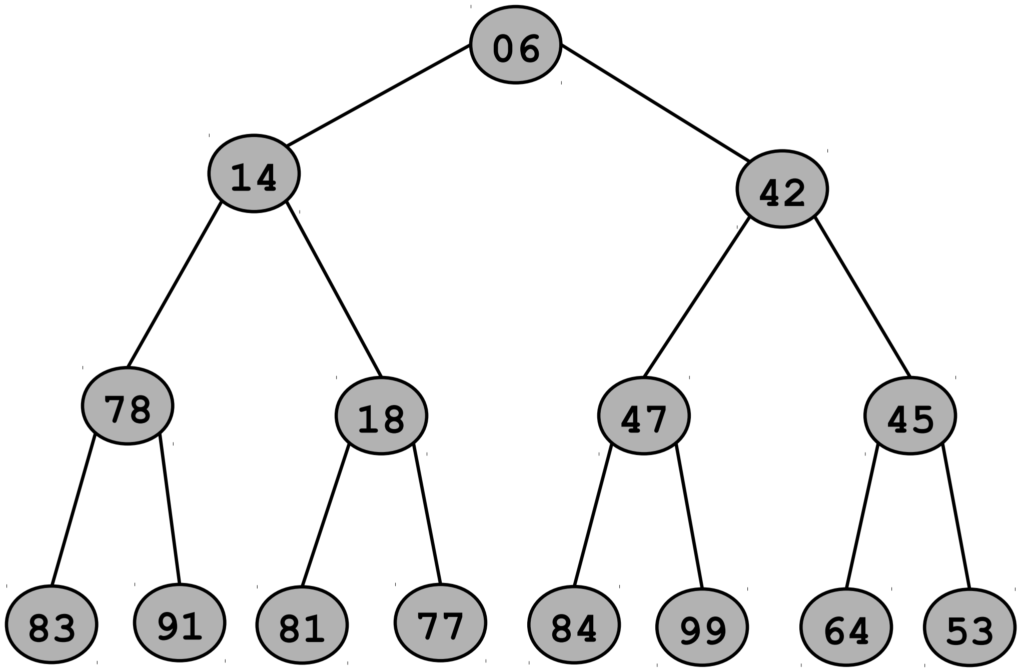
stop: heap ordered

O(log N) operations.

Binary Heap: Decrease Key

Decrease key of element x to k.

Bubble up until it's heap ordered.

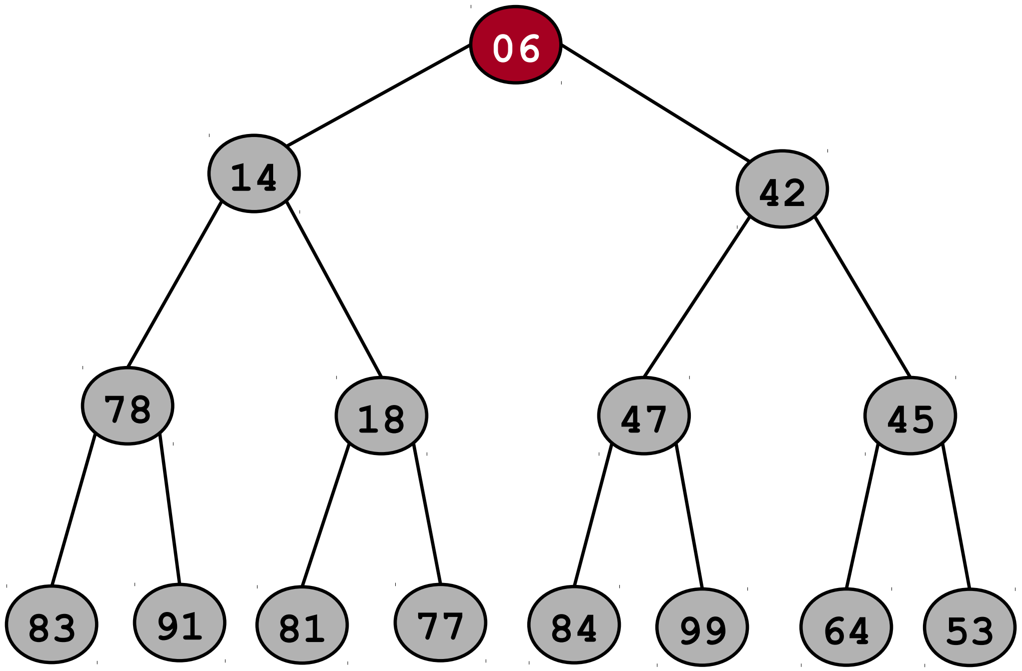


O(log N) operations.

Binary Heap: Delete Min

Delete minimum element from heap.

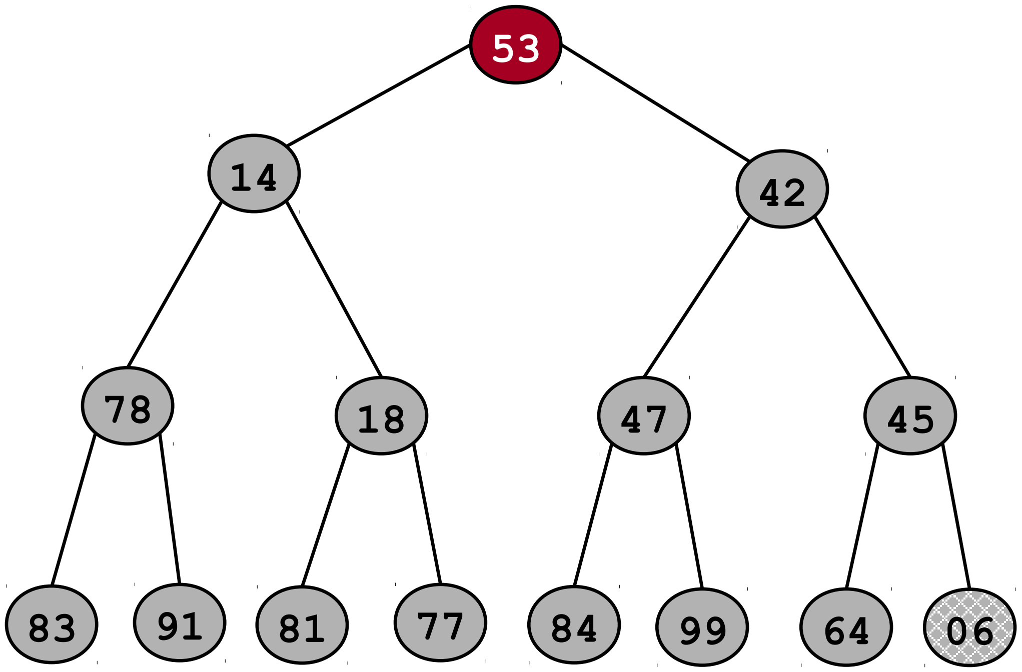
- **Exchange root with rightmost leaf.**
- **Bubble root down until it's heap ordered.**
  - **power struggle principle: better subordinate is promoted**



Binary Heap: Delete Min

Delete minimum element from heap.

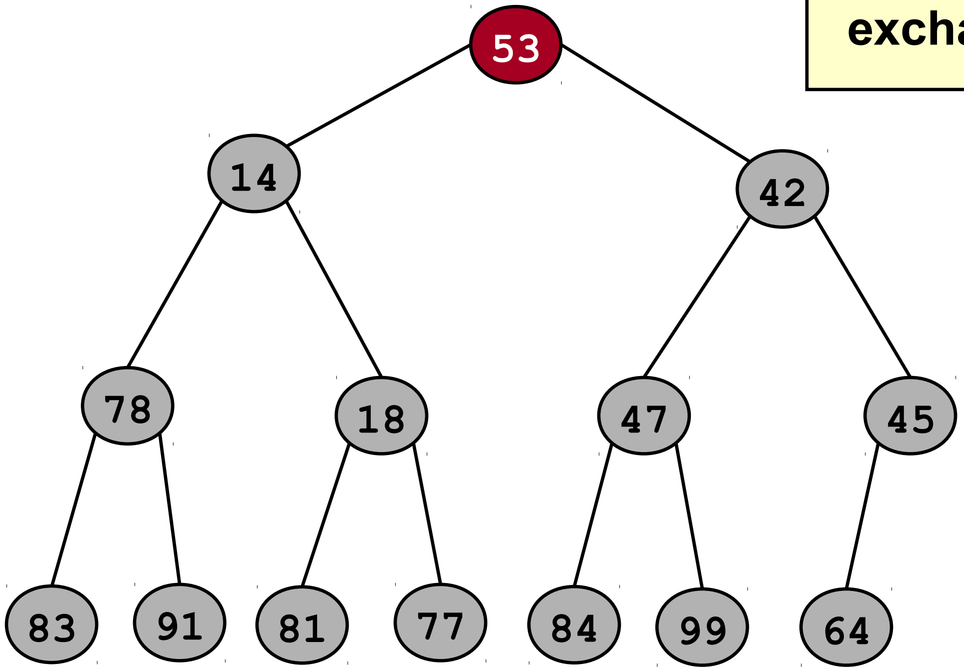
- Exchange root with rightmost leaf.
- Bubble root down until it's heap ordered.
  - power struggle principle: better subordinate is promoted



Binary Heap: Delete Min

Delete minimum element from heap.

- Exchange root with rightmost leaf.
- Bubble root down until it's heap ordered.
  - power struggle principle: better subordinate is promoted



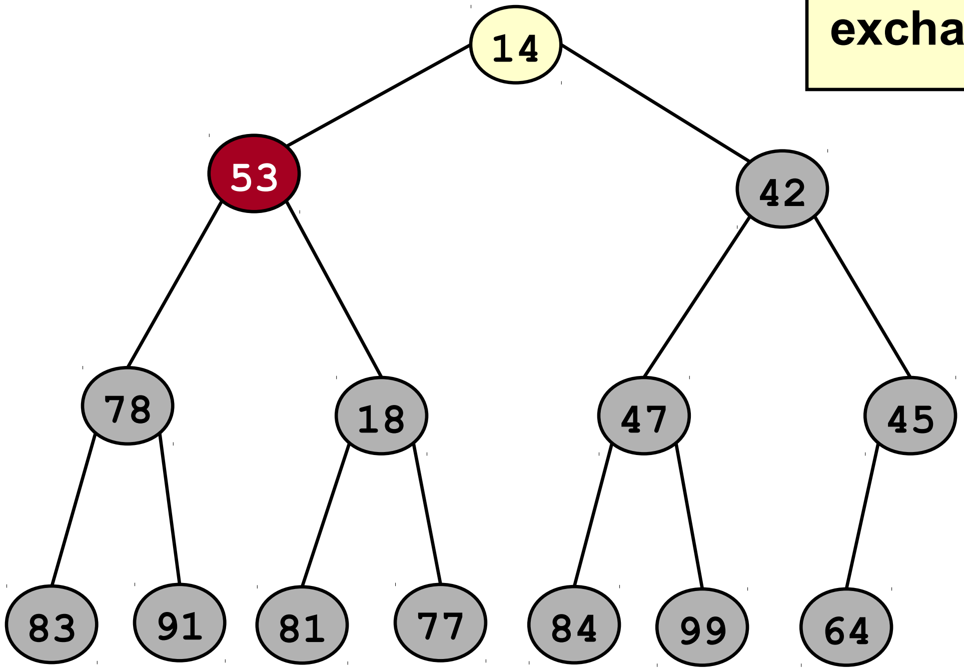
exchange with left child

Binary Heap: Delete Min

Delete minimum element from heap.

- Exchange root with rightmost leaf.
- Bubble root down until it's heap ordered.
  - power struggle principle: better subordinate is promoted

exchange with right child

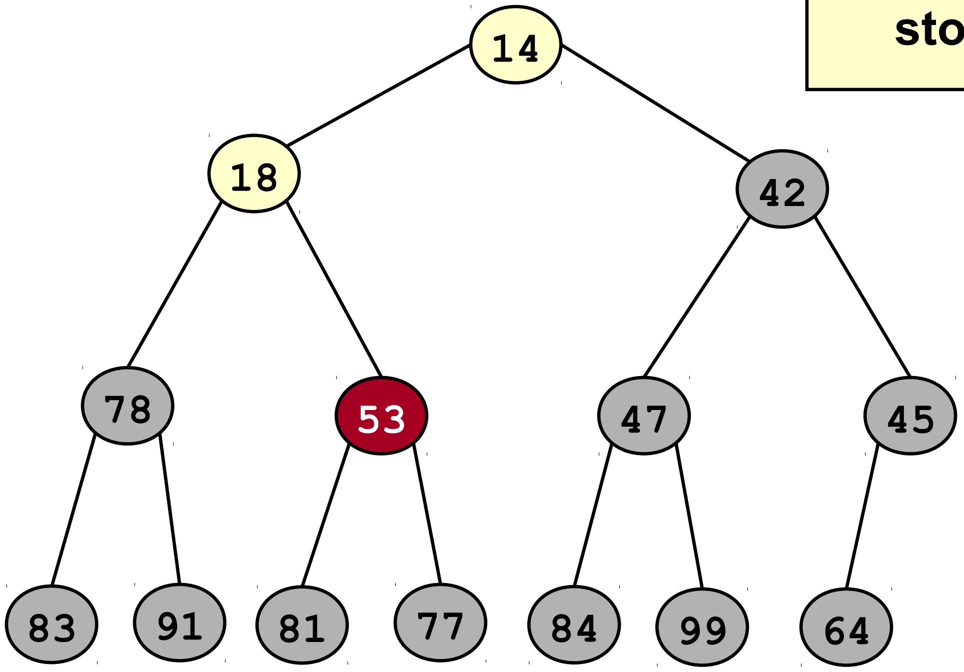


Binary Heap: Delete Min

Delete minimum element from heap.

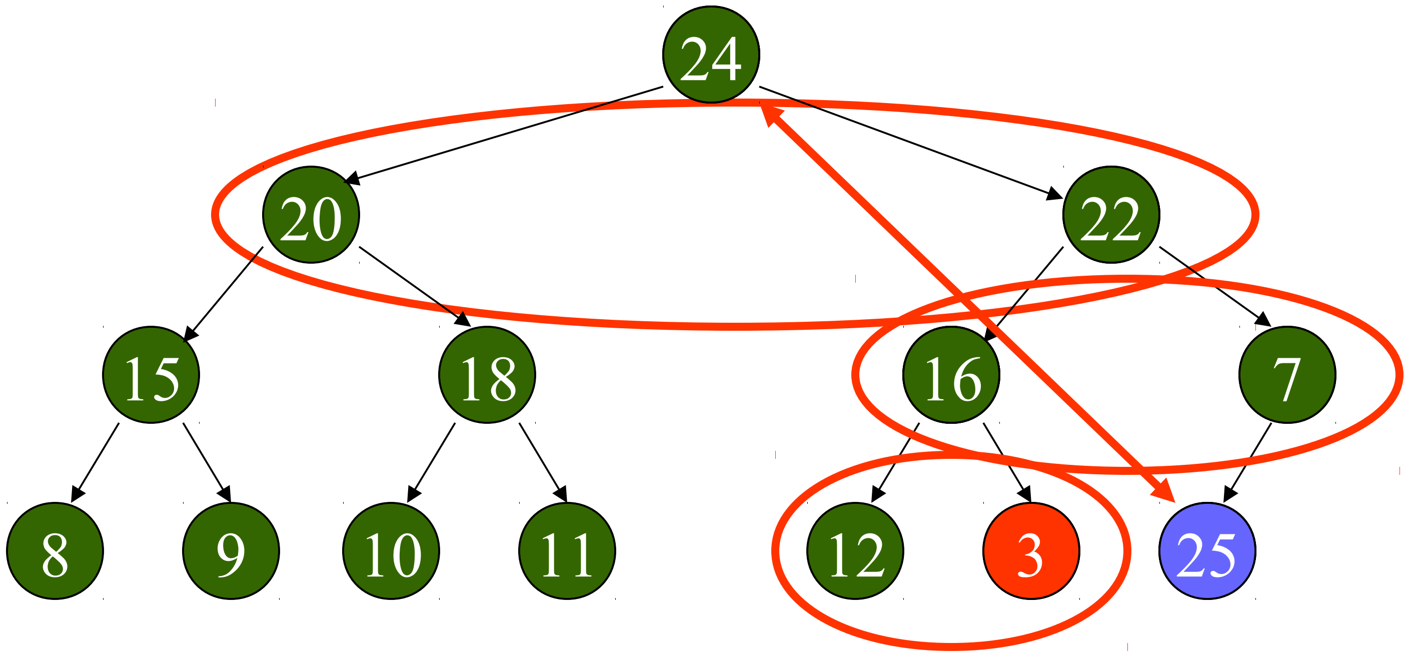
- Exchange root with rightmost leaf.
- Bubble root down until it's heap ordered.
  - power struggle principle: better subordinate is promoted

stop: heap ordered



O(log N) operations.

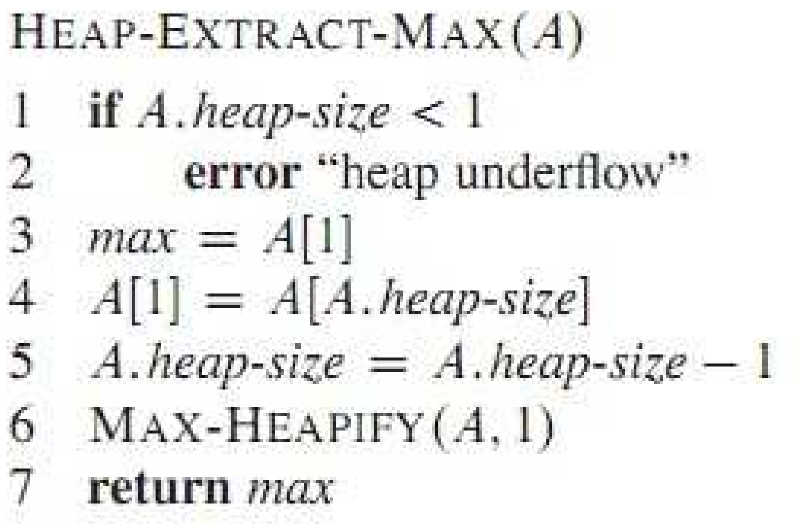
Binary Heap: Delete /Extract Max



delete

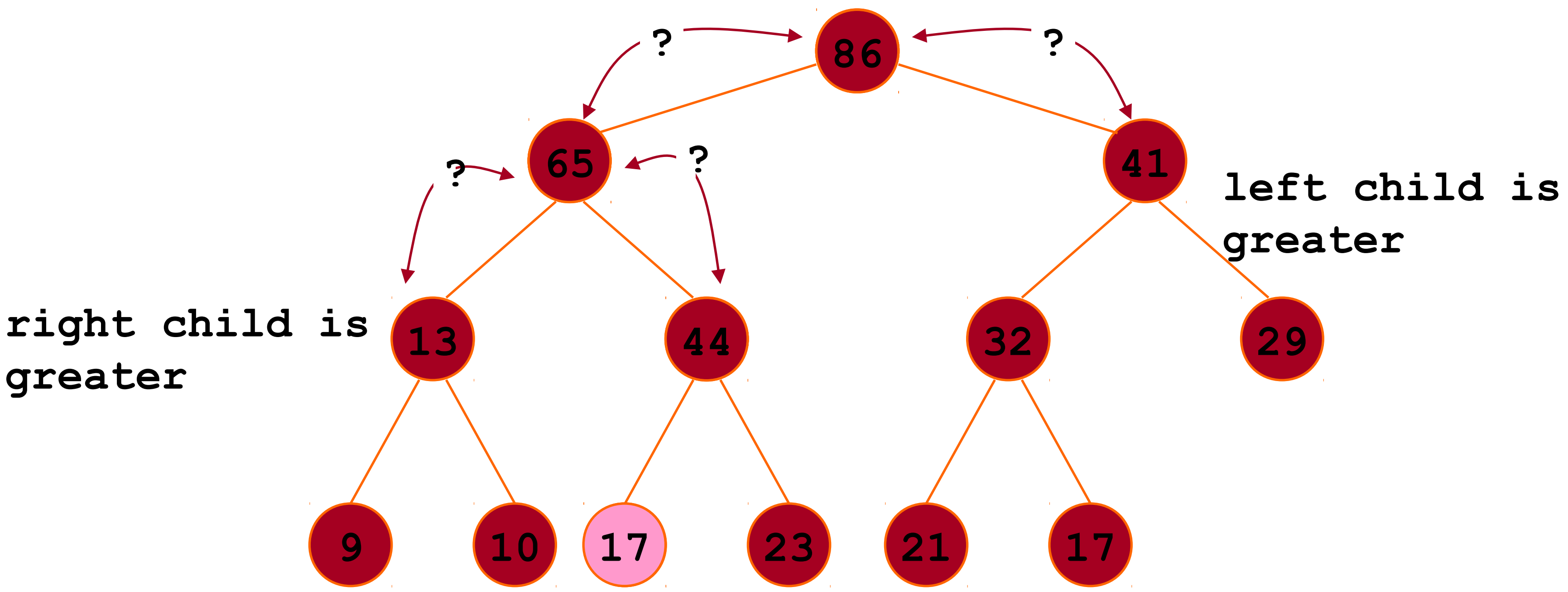
Last Node# 14 1110

Binary Heap: Delete /Extract Max



```
HEAP-EXTRACT-MAX($A$)
if $A.heap\text{-}size < 1$
        error "heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size = A.heap\text{-}size - 1$
MAX-HEAPIFY($A$, 1)
return $max$
```

Max-Heapify Example



children of 2oo2*22*2*22±1+14= 5, 3

index	1	2	3	4	5	6	7	8	8	10	11	12
value	86	65	41	13	44	32	29	9	10	17	23	21

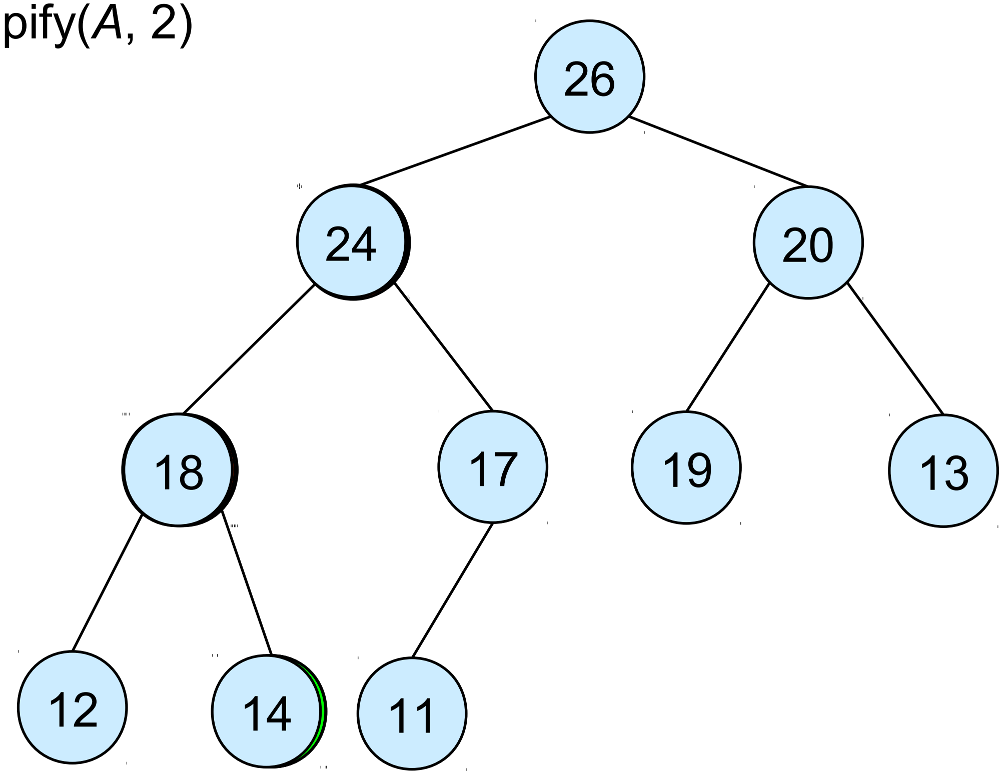
Procedure MaxHeapify

```
MaxHeapify(A, i)
1.  l = left(i);
2.  r = right(i);
3.  if (l ≤ heap-size[A] && A[l] > A[i])
4.              largest = l;
5.      else    largest = I;
6.  if (r ≤ heap-size[A] && A[r] > A[largest])
7.              largest = r;
8.  if (largest != i)
9.              Swap(A[i], A[largest])
10.             MaxHeapify(A, largest)
```

Assumption:
Left(*i*) and Right(*i*) are max-heaps.

MaxHeapify – Example

MaxHeapify(A, 2)



Running Time for MaxHeapify

```
MaxHeapify(A, i)
1.  l = left(i);
2.  r = right(i);
3.  if (l ≤ heap-size[A] && A[l] > A[i])
4.          largest = l;
5.      else largest = I;
6.  if (r ≤ heap-size[A] && A[r] > A[largest])
7.          largest = r;
8.  if (largest != i)
9.          Swap(A[i], A[largest])
10.         MaxHeapify(A, largest)
```

$\left.\right\}$ Time to fix node *i* and its children = **Θ(1)**

PLUS

$\left.\right\}$ Time to fix the subtree rooted at one of *i*'s children = *T*(size of subree at *largest*)

Running Time for MaxHeapify(A, n)

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

$largest \leq 2n/3$ (worst case occurs when the last row of tree is exactly half full)

$T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\lg n)$

Alternately, MaxHeapify takes O(h) where h is the height of the node where MaxHeapify is applied.

Building a heap

Use *MaxHeapify* to convert an array *A* into a max-heap.

How?

Call MaxHeapify on each element in a bottom-up manner.

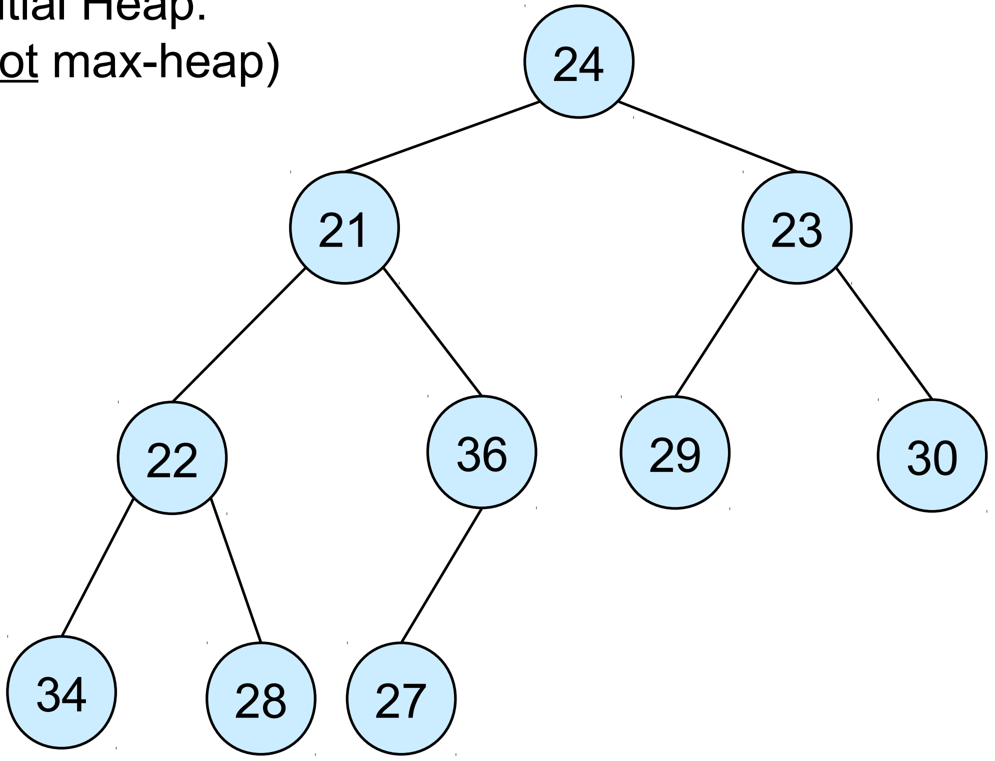
```
BuildMaxHeap(A)
1.  $heap\text{-}size[A] = length[A]$
2.  for $i = \lfloor length[A]/2 \rfloor$ down to 1
3.      do $MaxHeapify(A, i)$
```

BuildMaxHeap – Example

Input Array:

24	21	23	22	36	29	30	34	28	27

Initial Heap:
(<u>not</u> max-heap)



BuildMaxHeap – Example

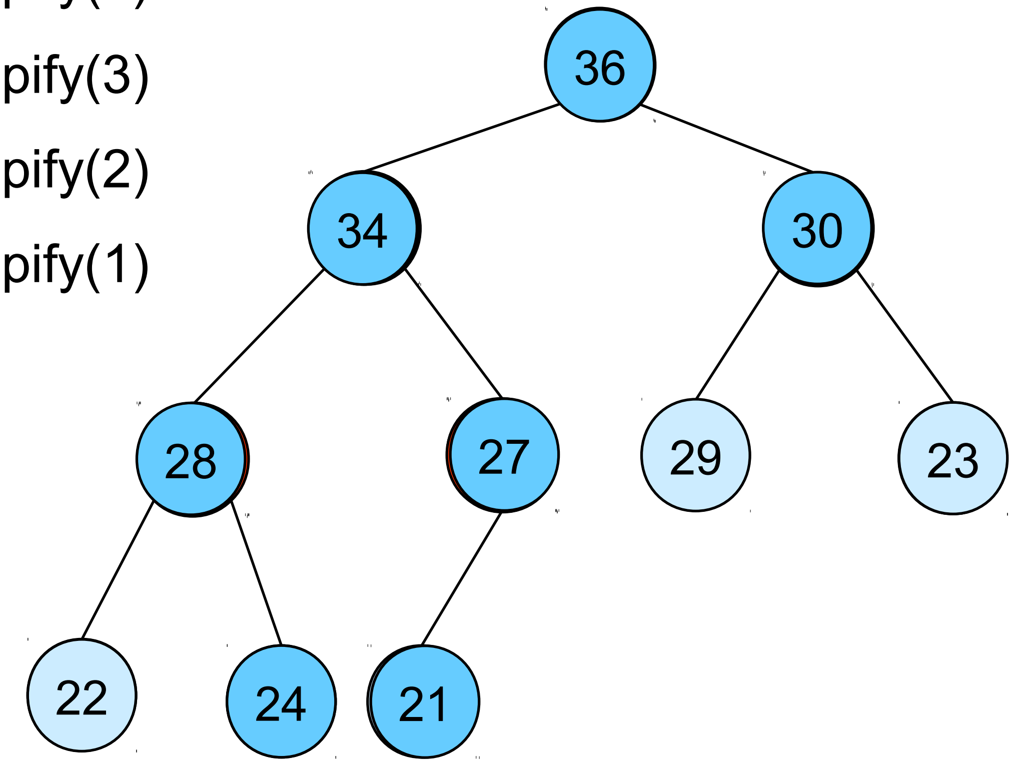
MaxHeapify(⌊10/2⌋ = 5)

MaxHeapify(4)

MaxHeapify(3)

MaxHeapify(2)

MaxHeapify(1)



Running Time of *BuildMaxHeap*

```
BuildMaxHeap(A)
1. heap-size[A] = length[A]
2. for i = ⌊length[A]/2⌋ down to 1
3.       do MaxHeapify(A, i)
```

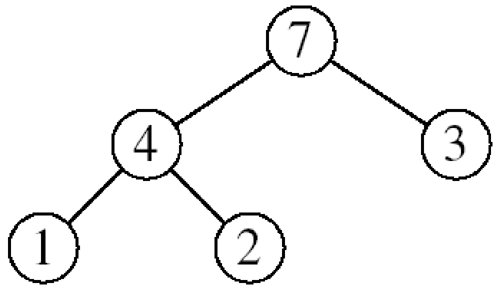
$\Theta(\log n)$

$\Theta(n \log n)$

Cost of a *MaxHeapify* call × No. of calls to *MaxHeapify*

$O(\lg n) \times O(n) = O(n \lg n)$

Heapsort



Goal:

Sort an array using heap representations

Idea:

Build a max-heap from the array

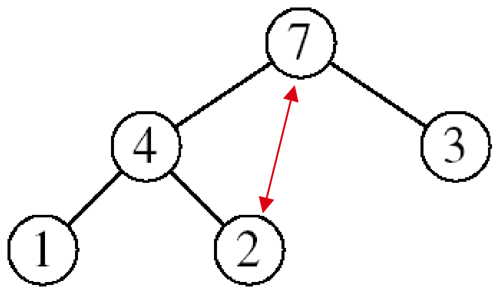
Swap the root (the maximum element) with the last element in the array

“Discard” this last node by decreasing the heap size

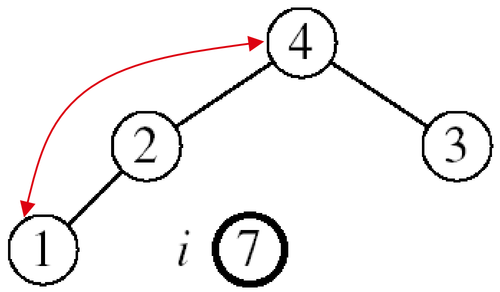
Call MAX-HEAPIFY on the new root

Repeat this process until only one node remains

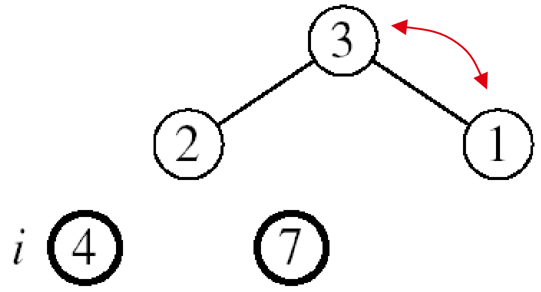
Example: A=[7, 4, 3, 1, 2]



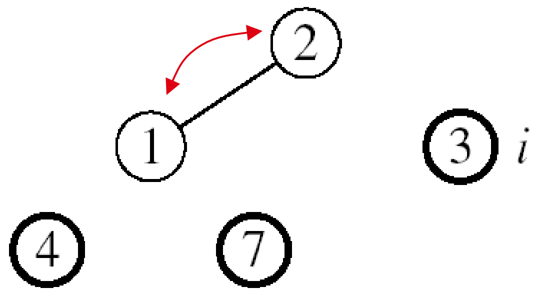
MAX-HEAPIFY(A, 1, 4)



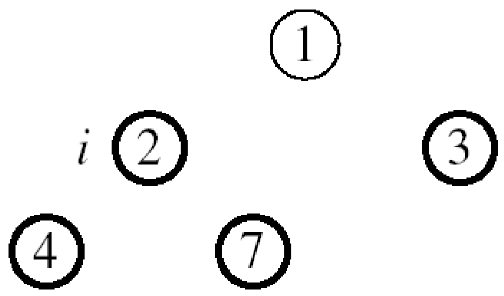
MAX-HEAPIFY(A, 1, 3)



MAX-HEAPIFY(A, 1, 2)



MAX-HEAPIFY(A, 1, 1)



A	1	2	3	4	7

Heapsort(*A*)

```
HeapSort(A)
1.  Build-Max-Heap(A)
2.  for i = length[A] downto 2
3.      swap( A[1], A[i] );
4.      heap-size[A] = heap-size[A] – 1;
5.      MaxHeapify(A, 1);
```

Algorithm Analysis

```
HeapSort(A)
1.  Build-Max-Heap(A)                          } Θ(n)
2.  for i = length[A] downto 2
3.          swap( A[1], A[i] );
4.            heap-size[A] = heap-size[A] – 1;
5.            MaxHeapify(A, 1);                } Θ(logn)
```

Θ(n -1)

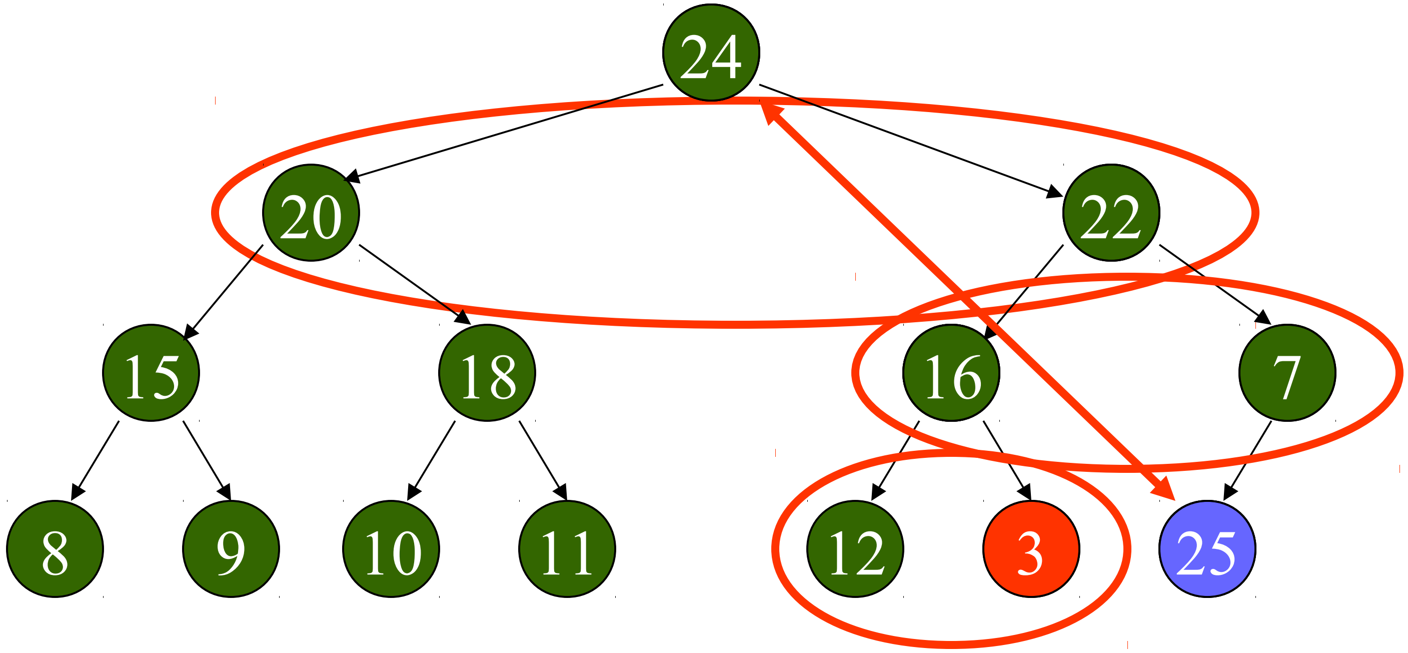
Θ(n logn)

In-place

Not Stable

Build-Max-Heap takes *O*(*n*) and each of the *n-1* calls to Max-Heapify takes time *O*(lg *n*).

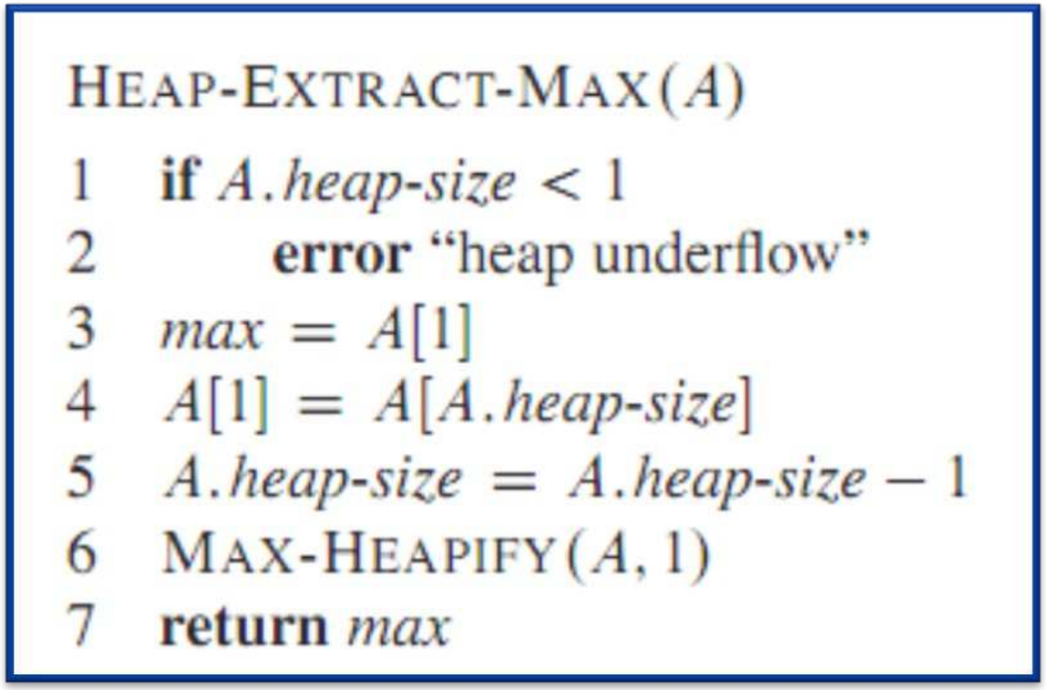
Binary Heap: Delete /Extract Max



delete

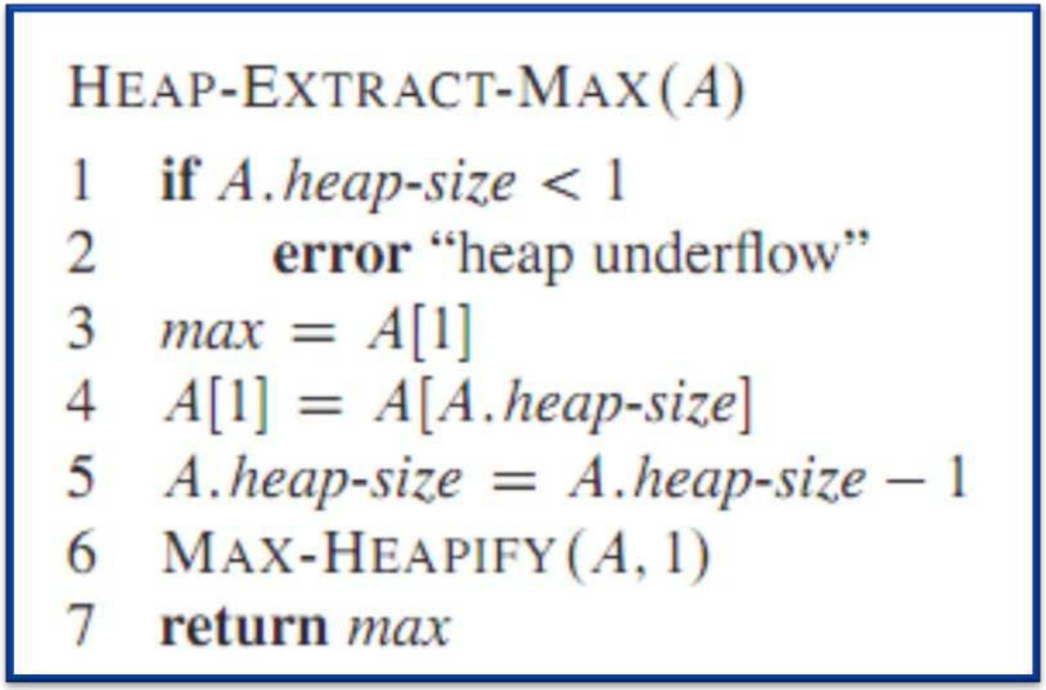
Last Node#   14   1110

Binary Heap: Delete /Extract Max



```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2       error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size - 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

Analysis Binary Heap: Delete /Extract Max



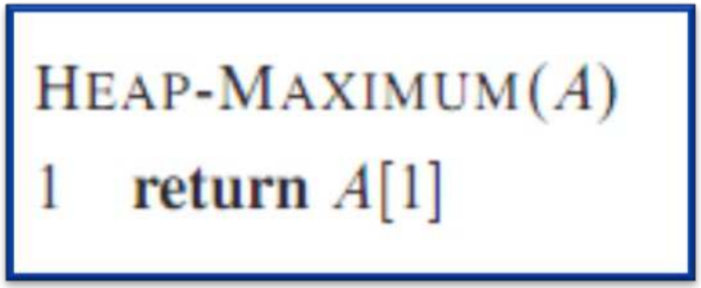
```
HEAP-EXTRACT-MAX(A)
if $A.heap\text{-}size < 1$
    error "heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size = A.heap\text{-}size - 1$
MAX-HEAPIFY(A, 1)
return max
```

$\Theta(1)$

$\Theta(\log n)$

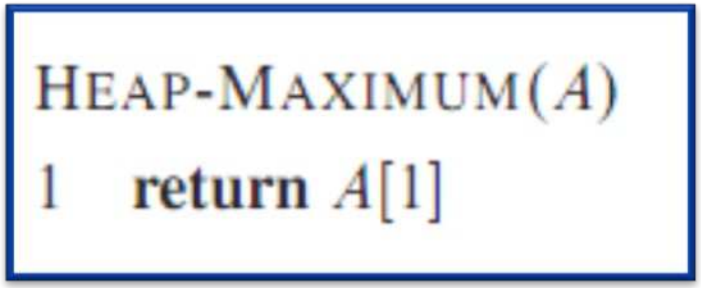
$\Theta(\log n)$

Binary Heap: Return/Find Max



```
HEAP-MAXIMUM(A)
1  return A[1]
```

Analysis Binary Heap: Return/Find Max

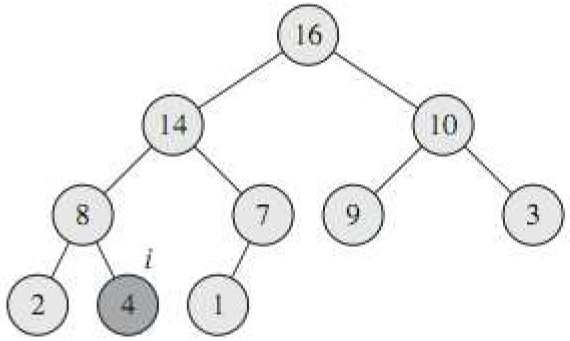


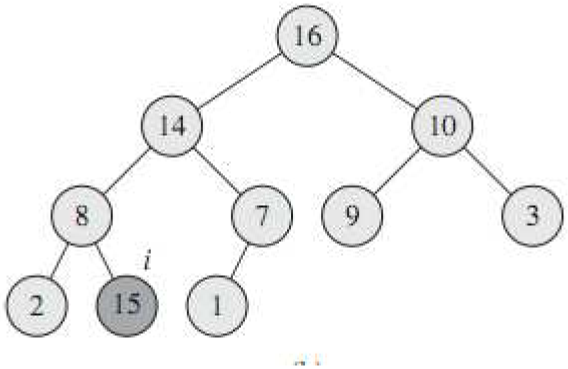
```
HEAP-MAXIMUM(A)
1  return A[1]
```

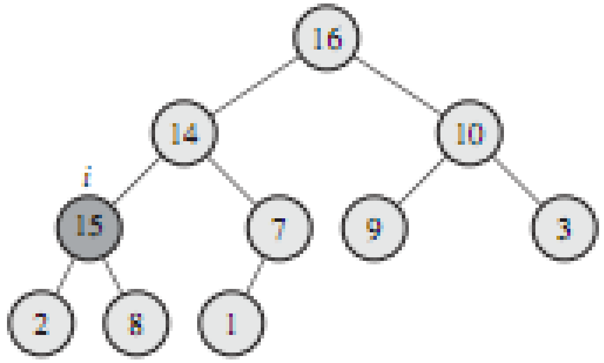
$\Theta(1)$

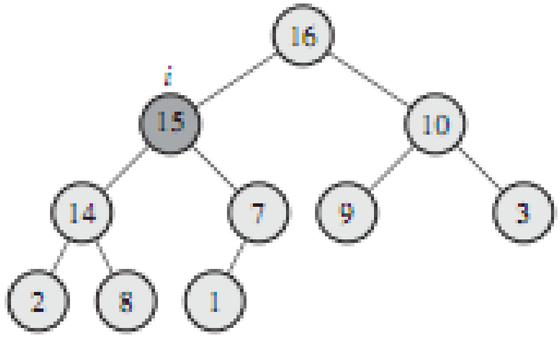
$\Theta(1)$

Binary Heap: Increase Key



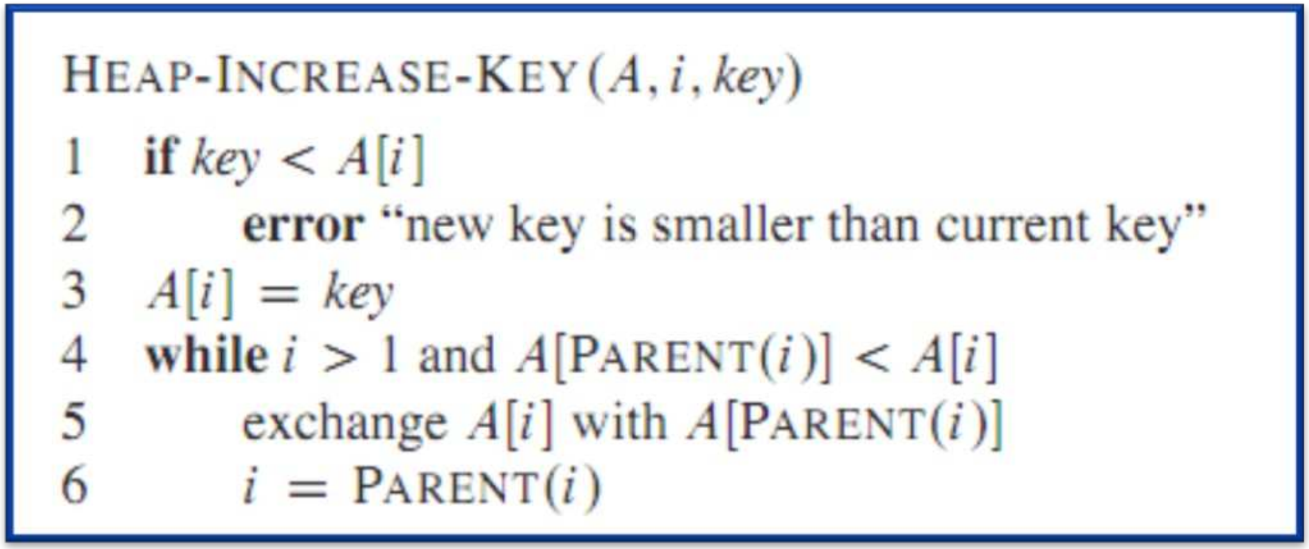






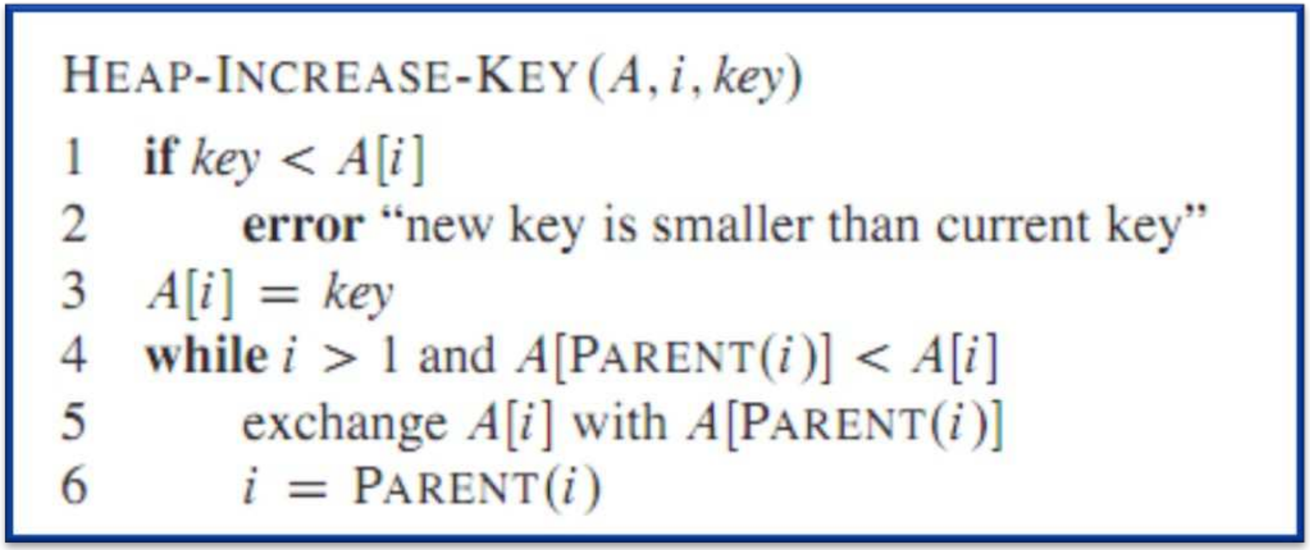
The node i has its key increased to 15.

Binary Heap: Increase Key



```
HEAP-INCREASE-KEY($A, i, key$)
1  if $key < A[i]$
2      error "new key is smaller than current key"
3  $A[i] = key$
4  while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
5      exchange $A[i]$ with $A[\text{PARENT}(i)]$
6      $i = \text{PARENT}(i)$
```

Analysis Binary Heap: Increase Key



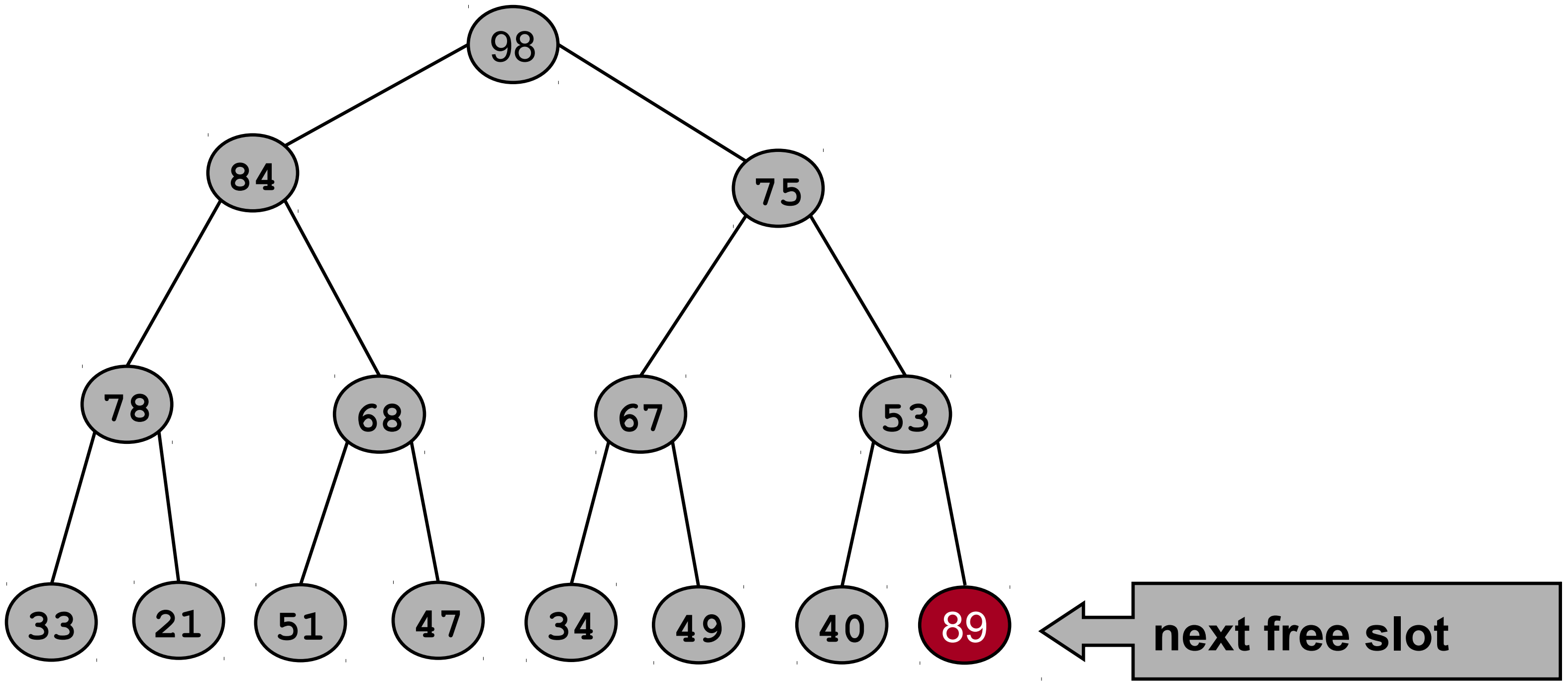
```
HEAP-INCREASE-KEY$(A, i, key)$
if $key < A[i]$
        error "new key is smaller than current key"
$A[i] = key$
while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
        exchange $A[i]$ with $A[\text{PARENT}(i)]$
        $i = \text{PARENT}(i)$
```

The running time of HEAP-INCREASE-KEY on an n-element heap is O(lg n), since the path traced from the node to the root has length O(lg n).

Binary Heap: Insertion

Insert element x into heap.

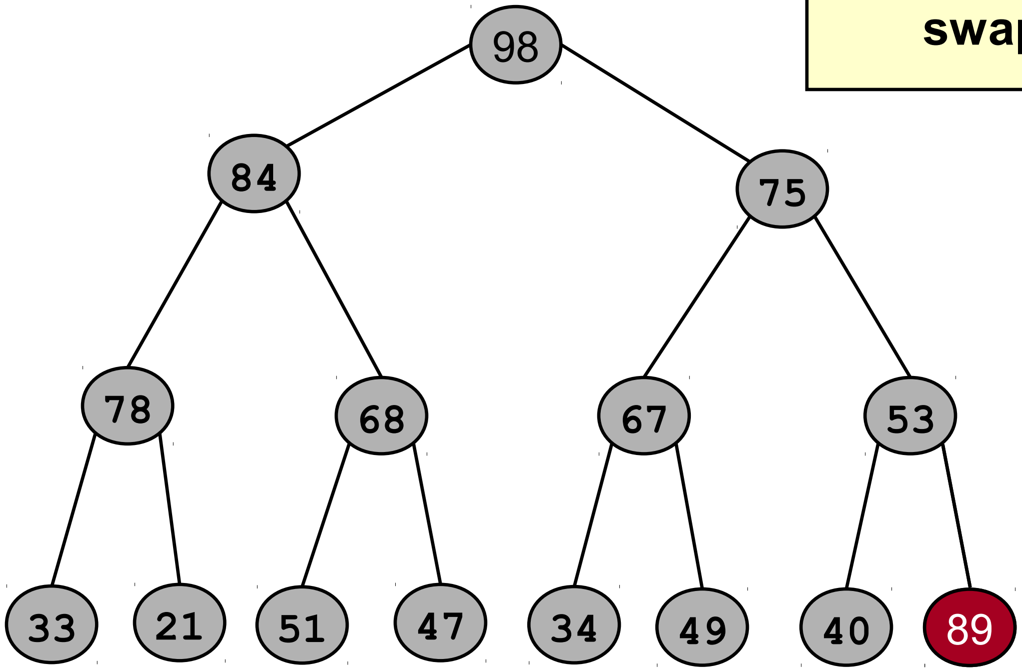
- Insert into next available slot.
- Bubble up until it's heap ordered.
  - Peter principle: nodes rise to level of incompetence



Binary Heap: Insertion

Insert element x into heap.

- **Insert into next available slot.**
- **Bubble up until it's heap ordered.**
  - **Peter principle: nodes rise to level of incompetence**



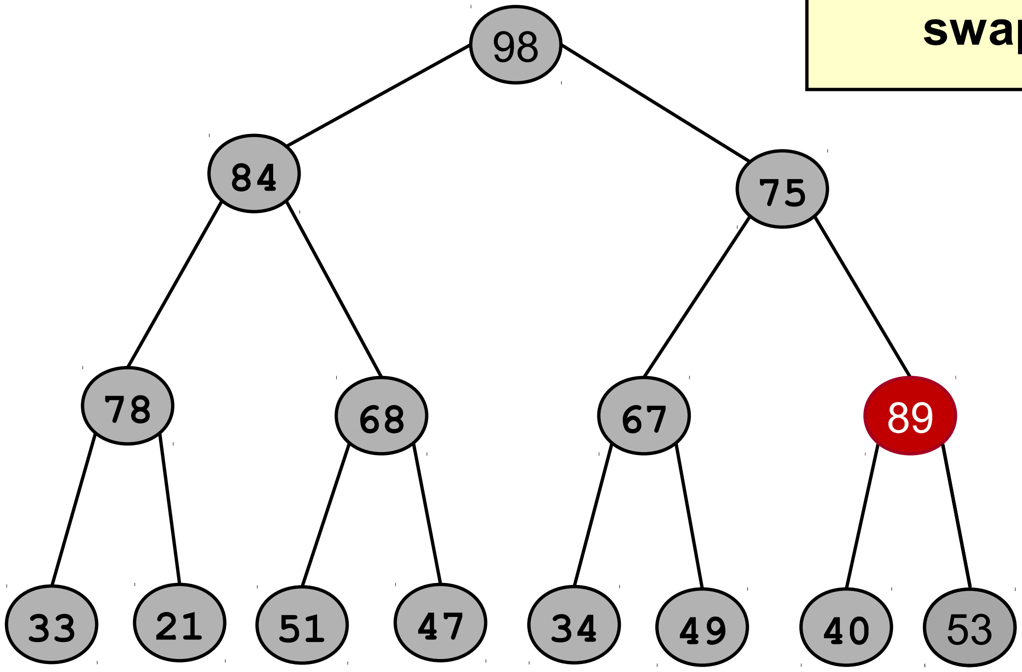
swap with parent

Binary Heap: Insertion

Insert element x into heap.

- Insert into next available slot.
- Bubble up until it's heap ordered.
  - Peter principle: nodes rise to level of incompetence

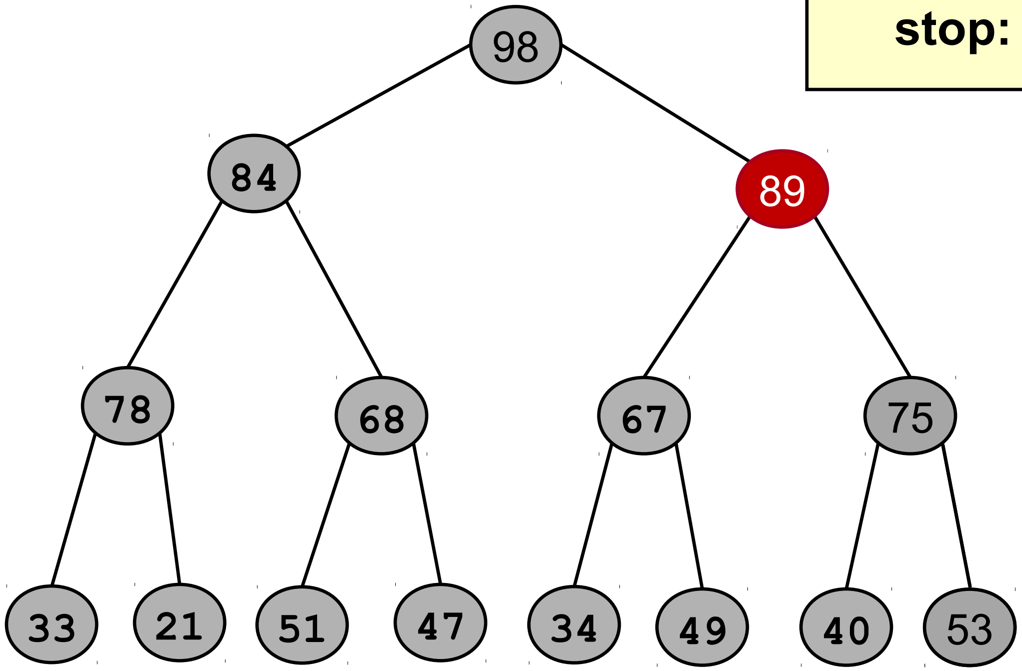
swap with parent



Binary Heap: Insertion

Insert element x into heap.

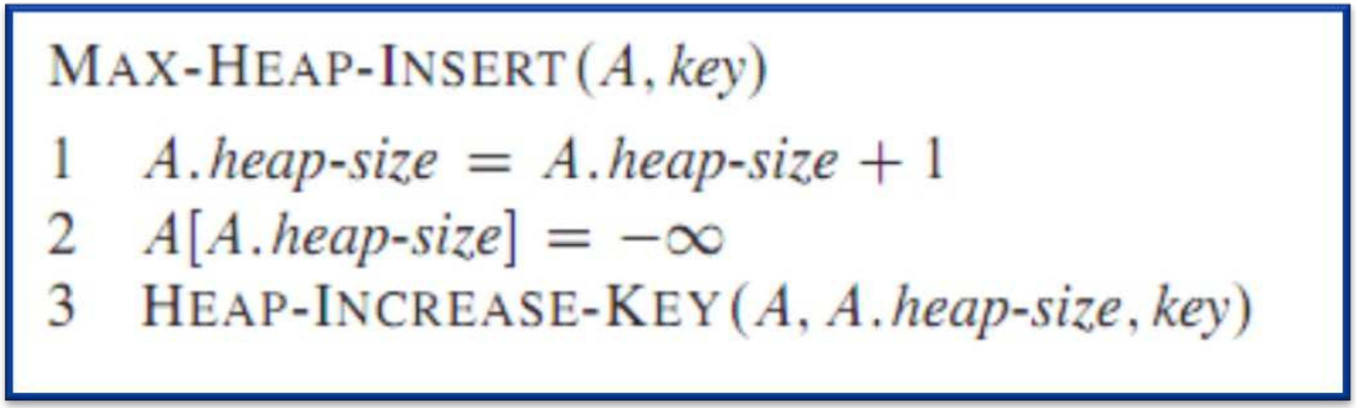
- **Insert into next available slot.**
- **Bubble up until it's heap ordered.**
  - **Peter principle: nodes rise to level of incompetence**



stop: heap ordered

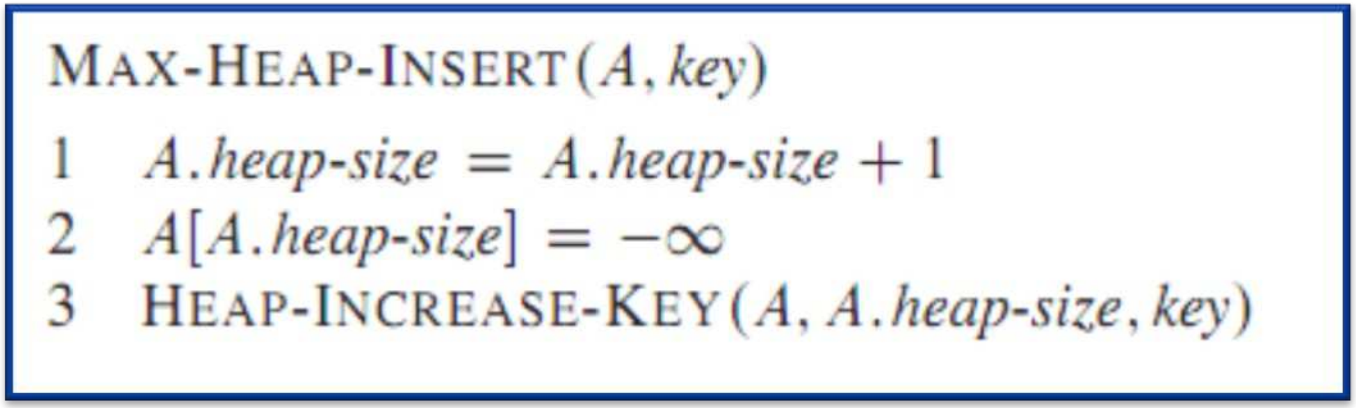
O(log N) operations.

Binary Heap: Insertion



```
MAX-HEAP-INSERT$(A, key)$
  $A.heap\text{-}size = A.heap\text{-}size + 1$
  $A[A.heap\text{-}size] = -\infty$
  HEAP-INCREASE-KEY$(A, A.heap\text{-}size, key)$
```

Analysis Binary Heap: Insertion



```
MAX-HEAP-INSERT(A, key)
1   $A.heap\text{-}size = A.heap\text{-}size + 1$
2   $A[A.heap\text{-}size] = -\infty$
3   HEAP-INCREASE-KEY(A, A.heap-size, key)
```

The running time of Insert on an n-element heap is O(lg n), since the path traced from the node to the root has length O(lg n).

Priority Queues

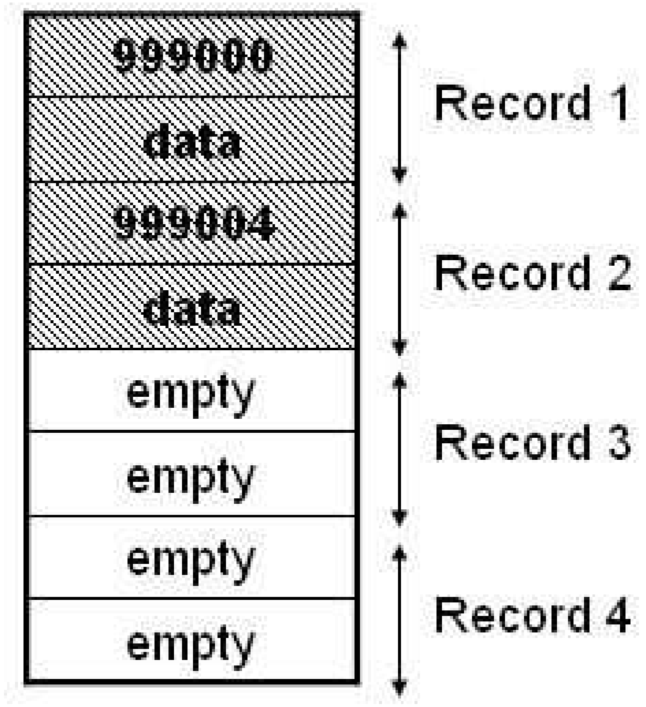
Operation	
Build -heap	N
insert	log N
find-min	1
delete-min	log N
union	N
decrease-key	log N
delete	log N
is-empty	1

B-Trees

- Disk Storage
- What is a multiway tree?
- What is a B-tree?
- Why B-trees?
- Insertion in a B-tree
- Deletion in a B-tree

Disk Storage

- Data is stored on disk (i.e., secondary memory) in blocks.
- A block is the smallest amount of data that can be accessed on a disk.
- Each block has a fixed number of bytes – typically 512, 1024, 2048, 4096 or 8192 bytes
- Each block may hold many data records.

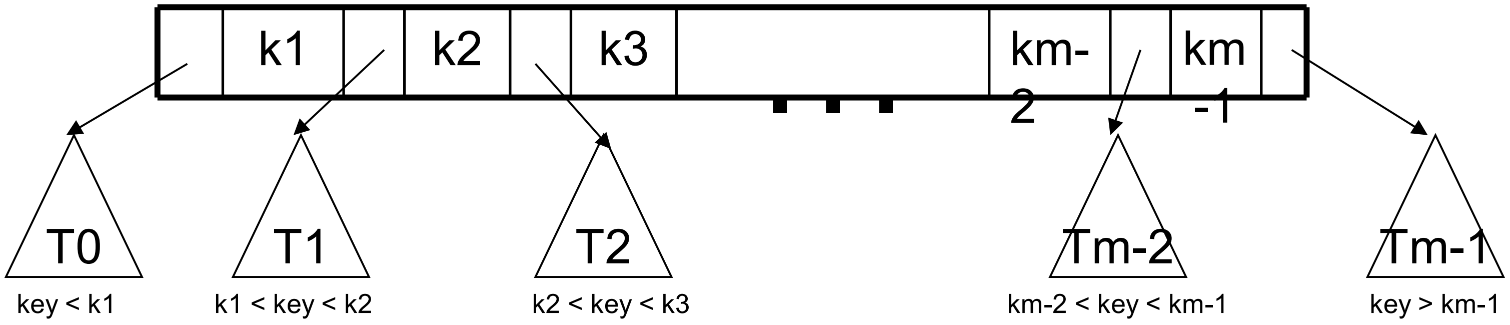


Motivation for studying Multi-way and B-trees

- A disk access is very expensive compared to a typical computer instruction (mechanical limitations) - One disk access is worth about 200,000 instructions.
- Thus, When data is too large to fit in main memory the number of disk accesses becomes important.
- Many algorithms and data structures that are efficient for manipulating data in primary memory are not efficient for manipulating large data in secondary memory because they do not minimize the number of disk accesses.
- For example, AVL trees are not suitable for representing huge tables residing in secondary memory.
- The height of an AVL tree increases, and hence the number of disk accesses required to access a particular record increases, as the number of records increases.

What is a Multi-way tree?

- A multi-way (or m-way) search tree of order m is a tree in which
  - Each node has at-most **m** subtrees, where the subtrees **<u>may be empty</u>**.
  - Each node consists of at least **1** and at most **m-1** distinct keys
  - The keys in each node are sorted.

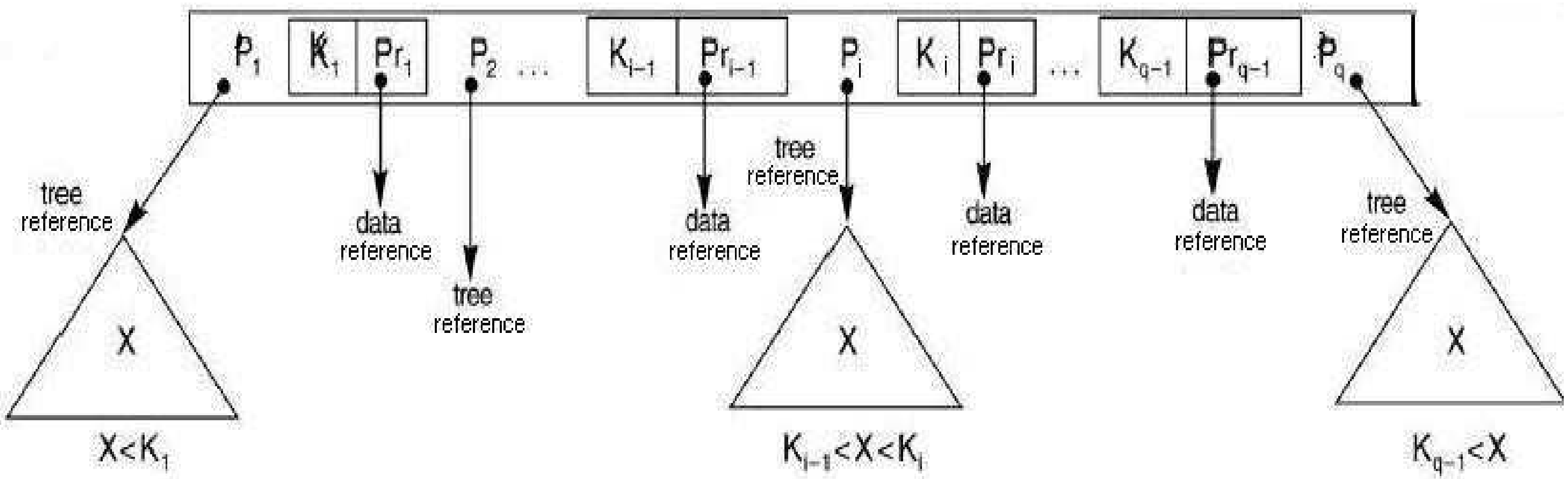


- The keys and subtrees of a non-leaf node are ordered as:

  T0, k1, T1, k2, T2, . . . , km-1, Tm-1 such that:

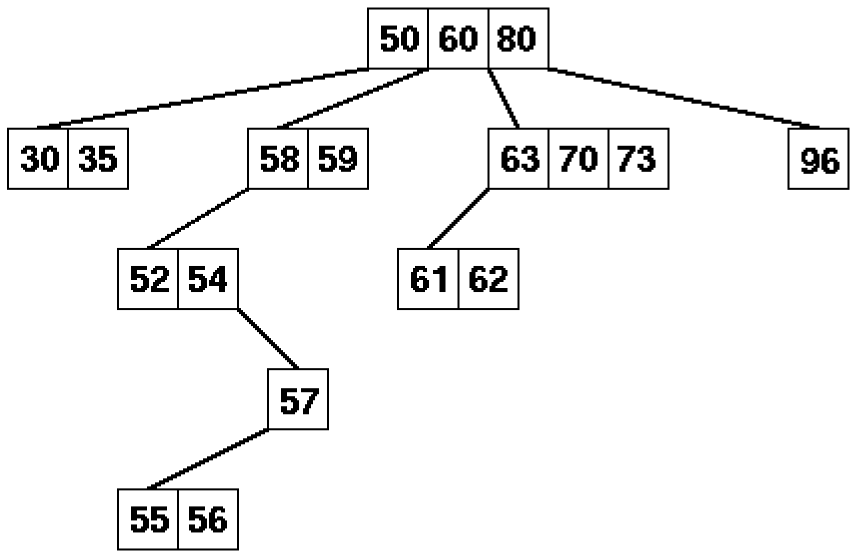
  - All keys in subtree T0 are less than k1.
  - All keys in subtree Ti , 1 <= i <= m - 2, are greater than ki but less than ki+1.
  - All keys in subtree Tm-1 are greater than km-1

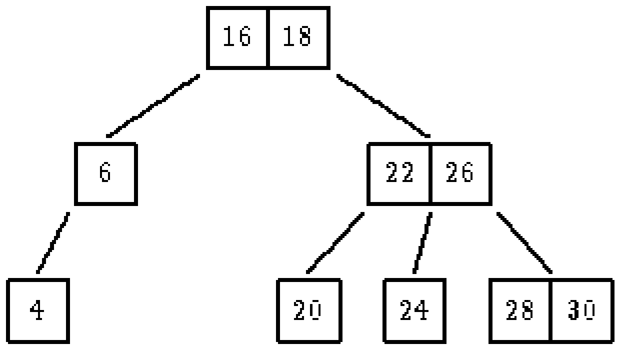
The node structure of a Multi-way tree



- Note:
  - Corresponding to each key there is a data reference that refers to the data record for that key in secondary memory.
  - In our representations we will omit the data references.
  - The literature contains other node representations that we will not discuss.

Examples of Multi-way Trees





- Note: In a multiway tree:
  - The leaf nodes need not be at the same level.
  - A non-leaf node with **n** keys may contain less than **n + 1** non-empty subtrees.

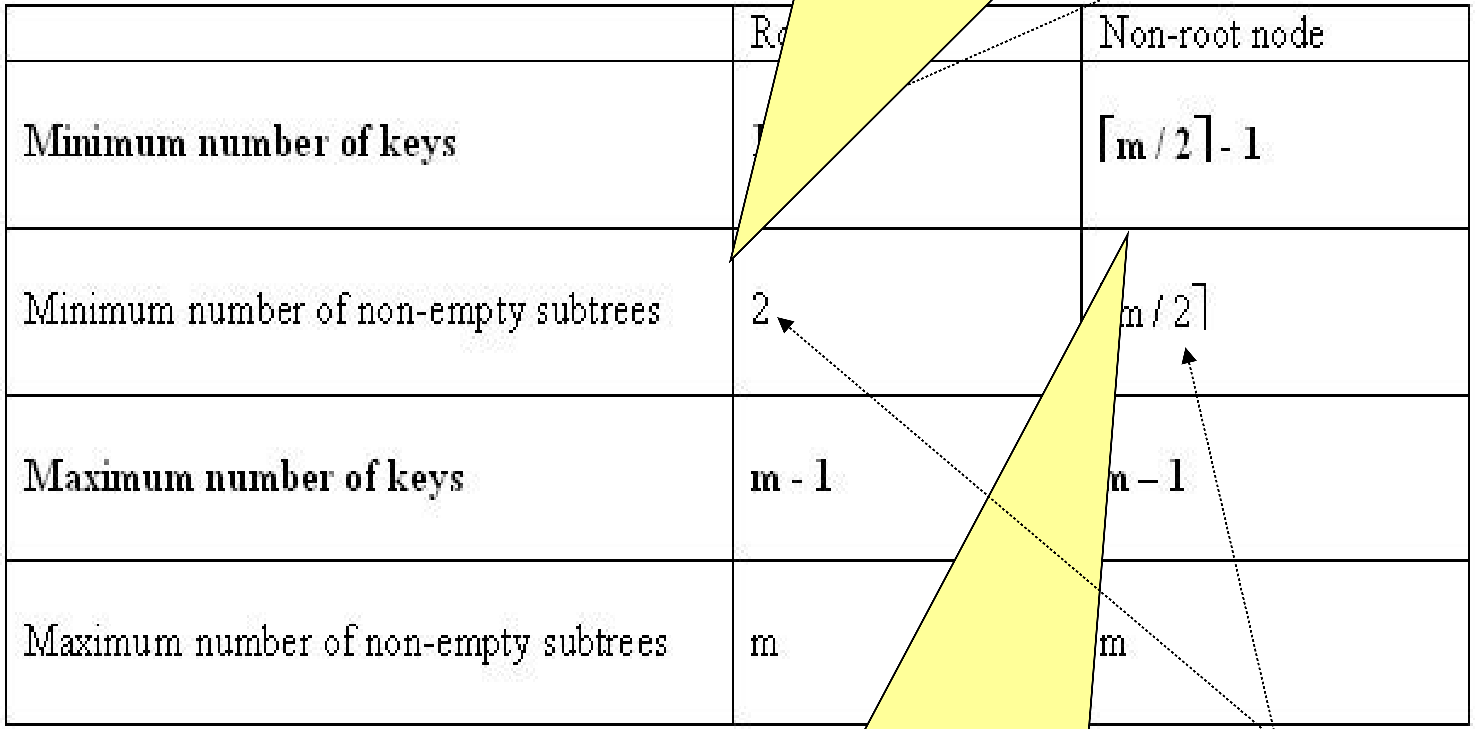
What is a B-Tree?

- A B-tree of order m (or branching factor m), where m > 2, is either an empty tree or a multiway search tree with the following properties:
  - The root is either a leaf or it has at least two non-empty subtrees and at most m non-empty subtrees.
  - Each non-leaf node, other than the root, has at least m/2 non-empty subtrees and at most m non-empty subtrees. (Note: x is the lowest integer > x).
  - The number of keys in each non-leaf node is one less than the number of non-empty subtrees for that node.
  - All leaf nodes are at the same level; that is the tree is perfectly balanced.

What is a B-tree? (cont’d)

For a non-empty B-tree of order m:

This may be zero, if the node is a leaf as well

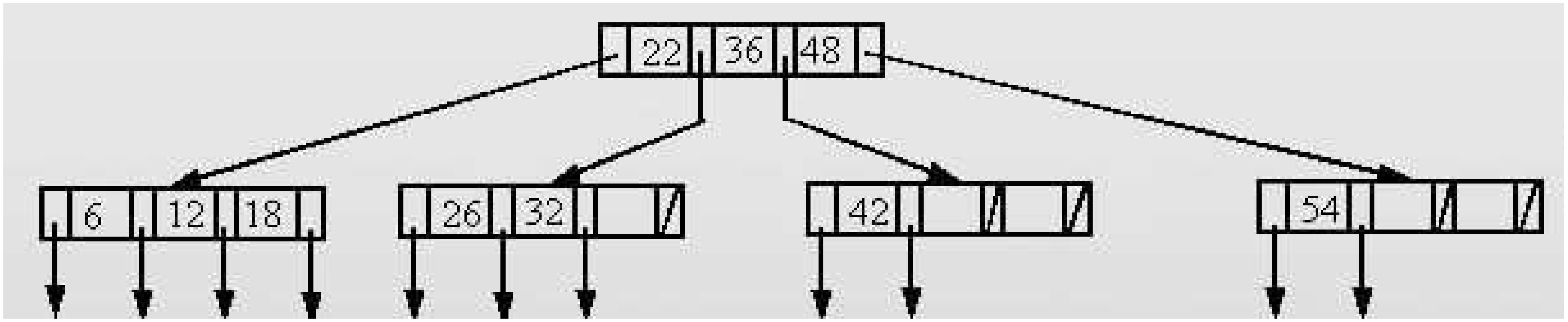


	Ro	Non-root node
Minimum number of keys	1	$\lceil m / 2 \rceil - 1$
Minimum number of non-empty subtrees	2	$m / 2\rceil$
Maximum number of keys	**m - 1**	**n – 1**
Maximum number of non-empty subtrees	m	m

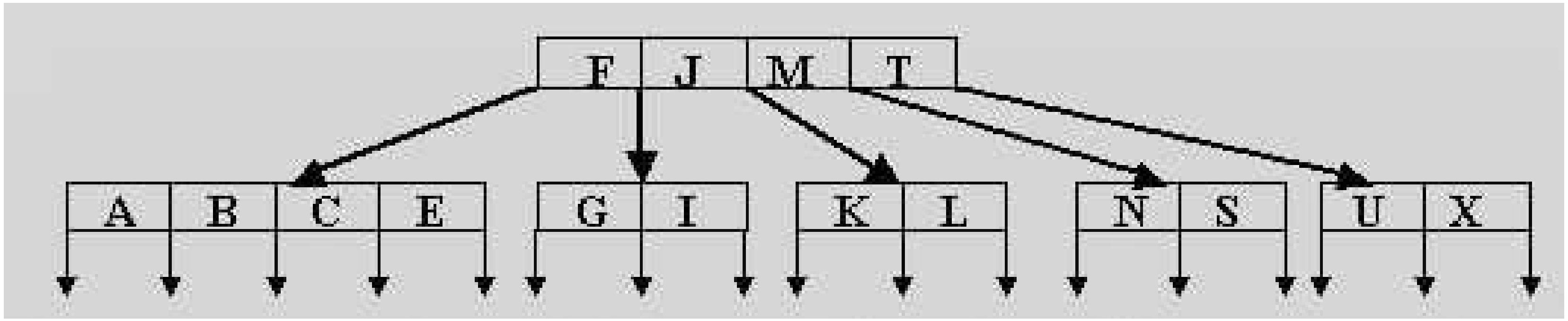
These will be zero if the node is a leaf as well

B-Tree Examples

Example: A B-tree of order 4



Example: A B-tree of order 5



Note:

- The data references are not shown.
- The leaf references are to empty subtrees

More on Why B-Trees

- B-trees are suitable for representing huge tables residing in secondary memory because:
   1. With a large branching factor m, the height of a B-tree is low resulting in fewer disk accesses.
      Note: As m **increases the amount of computation at each node increases; however this cost is negligible compared to hard-drive accesses.**

   3. The branching factor can be chosen such that a node corresponds to a block of secondary memory.

   4. The most common data structure used for database indices is the B-tree. An **index** is any data structure that takes as input a property (e.g. a value for a specific field), called the search key, and ***quickly*** finds all records with that property.

Comparing B-Trees with AVL Trees

- The height h of a B-tree of order m, with a total of n keys, satisfies the inequality:

$h <= 1 + \log_{m/2} ((n + 1) / 2)$

- If m = 300 and n = 16,000,000 then h ≈ 4.
- Thus, in the worst case finding a key in such a B-tree requires 3 disk accesses (assuming the root node is always in main memory).
- The average number of comparisons for an AVL tree with n keys is log n + 0.25 where n is large.
- If n = 16,000,000 the average number of comparisons is 24.
- Thus, in the average case, finding a key in such an AVL tree requires 24 disk accesses.

Insertion in B-Trees

- **OVERFLOW CONDITION:**

  A root-node or a non-root node of a B-tree of order m overflows if, after a key insertion, it contains m keys.

- **Insertion algorithm:**

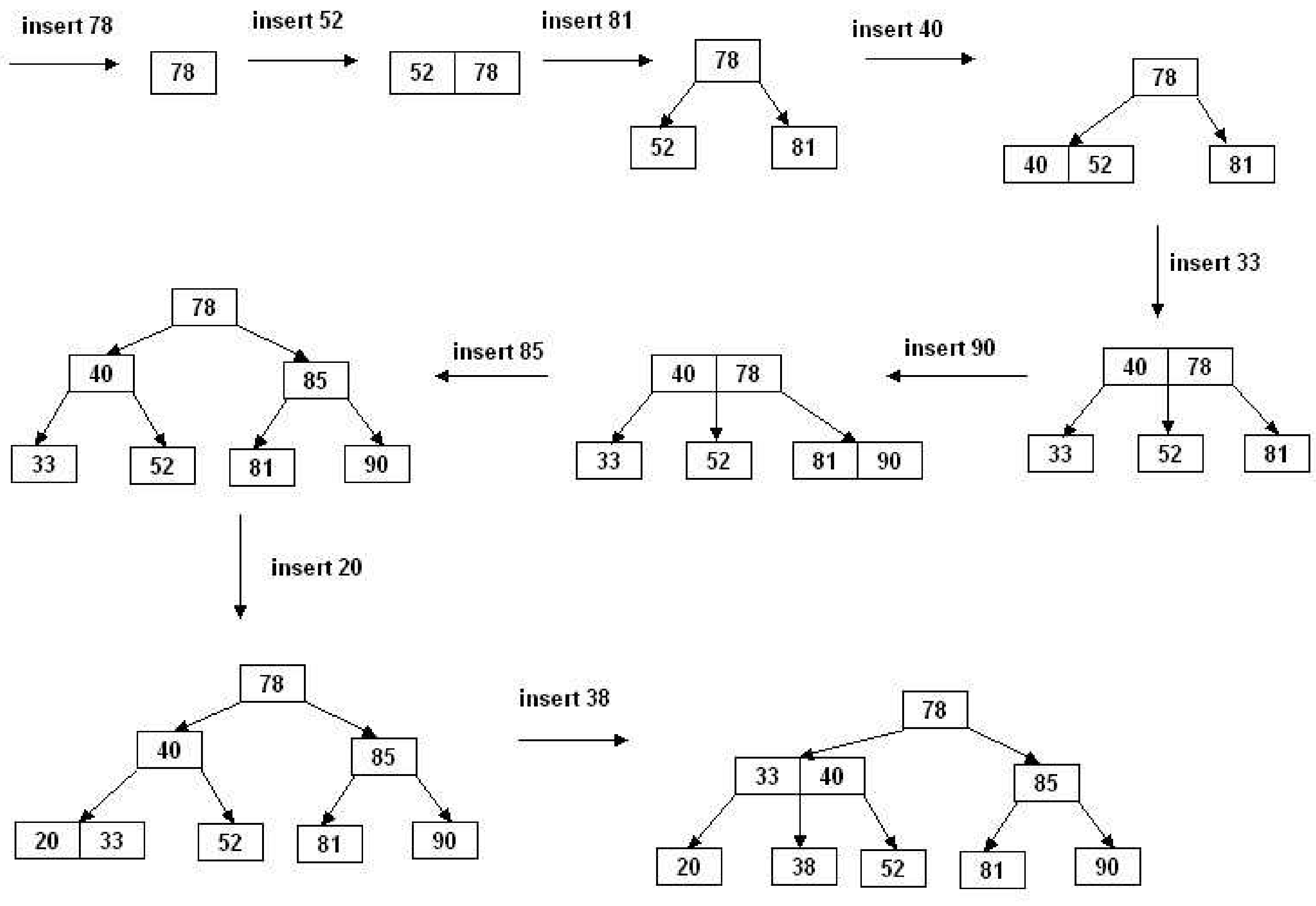
  If a node overflows, split it into two, propagate the "middle" key to the parent of the node. If the parent overflows the process propagates upward. If the node has no parent, create a new root node.

- **Note: Insertion of a key always <u>starts</u> at a leaf node.**

Insertion in B-Trees

- **Insertion in a B-tree of <u>odd</u> order**

- Example: Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in this order in an initially empty B-tree of order 3



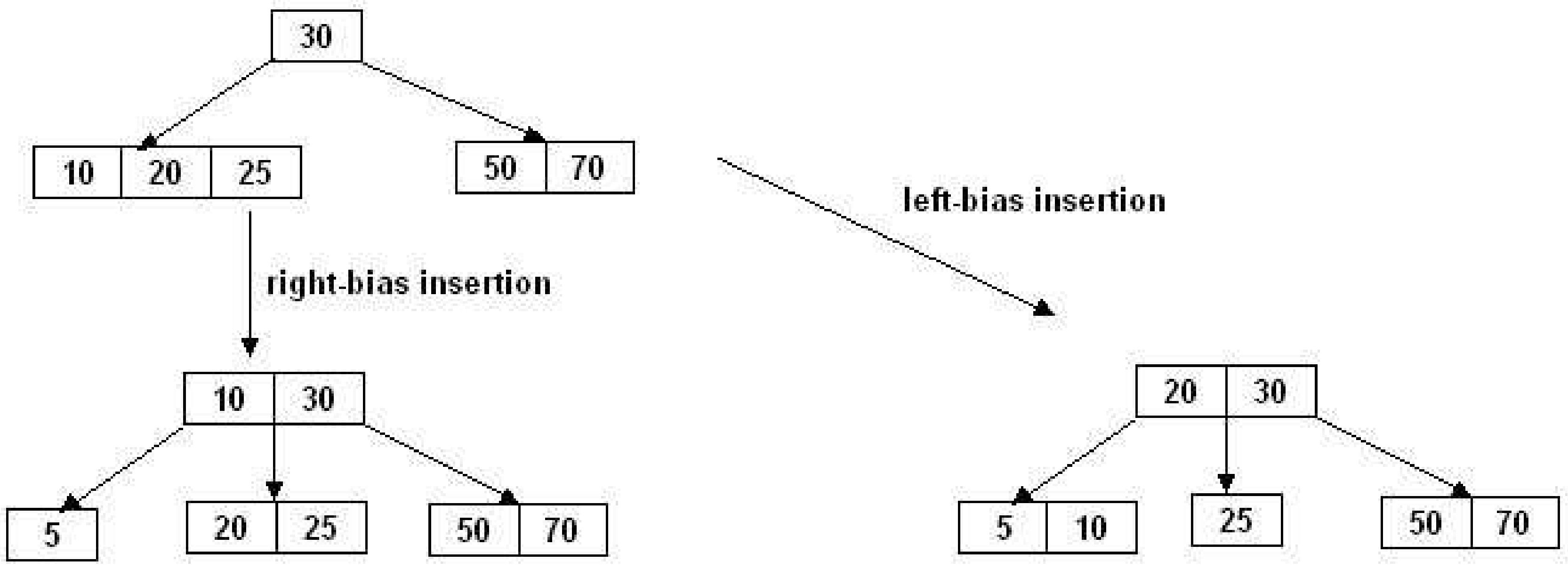
Insertion in B-Trees

- **Insertion in a B-tree of <u>even</u> order**

At each node the insertion can be done in two different ways:

- **right-bias:** The node is split such that its right subtree has more keys than the left subtree.
- **left-bias:** The node is split such that its left subtree has more keys than the right subtree.

- **Example**: Insert the key **5** in the following B-tree of order **4**:



B-Tree Insertion Algorithm

```
insertKey (x){
   if(the key **x** is in the tree)
       throw an appropriate exception;

   let the insertion leaf-node be the currentNode;
   insert **x** in its proper location within the node;

   if(the currentNode does not overflow)
     return;
    done = false;
    do{
       if (m is odd) {
              split currentNode into two siblings such that the right sibling **rs** has m/2 right-most keys,
              and the left sibling **ls** has m/2 left-most keys;
              Let **w** be the middle key of the splinted node;
       }
       else {           // m is even
              split currentNode into two siblings by any of the following methods:
                •    right-bias: the right sibling **rs** has m/2 right-most keys, and the left sibling **ls** has (m-1)/2 left-most keys.
                •    left-bias: the right sibling **rs** has (m-1)/2 right-most keys, and the left sibling **ls**  has m/2 left-most keys.
             let **w** be the "middle" key of the splinted node;
       }
      if (the currentNode is not the root node) {
             insert **w** in its proper location in the parent **p** of the currentNode;
          if (**p** does not overflow)
                done = true;
         else
                let **p** be the currentNode;
       }
  } while (! done && currentNode is not the root node);
```

B-Tree Insertion Algorithm - Contd

```
 if (! done) {
      create a new root node with w as its only key;
      let the right sibling rs be the right child of the new root;
      let the left sibling ls be the left child of the new root;
  }
  return;
}
```

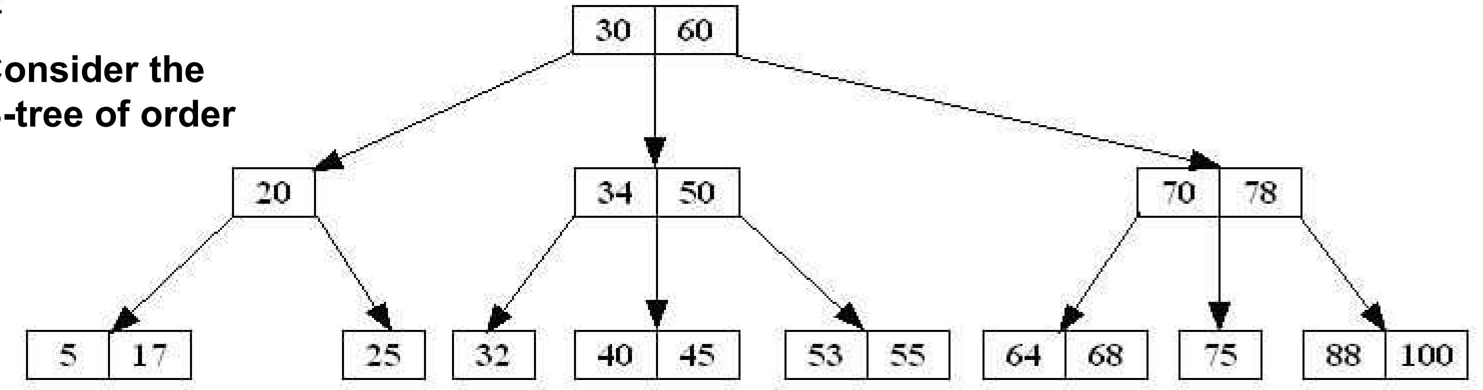
Deletion in B-Tree

- **Like insertion, deletion must be on a leaf node. If the key to be deleted is not in a leaf, swap it with either its successor or predecessor (each will be in a leaf).**

- The successor of a key **k** is the smallest key greater than **k**.
- The predecessor of a key **k** is the largest key smaller than **k**.

- IN A B-TREE THE SUCCESSOR AND PREDECESSOR, IF ANY, OF ANY KEY IS IN A LEAF NODE

Example: Consider the following B-tree of order 3:



successor	predecessor	key
25	17	20
32	25	30
40	32	34
53	45	50
64	55	60
75	68	70
88	75	78

Deletion in B-Tree

- **UNDERFLOW CONDITION**
- A non-root node of a B-tree of order m underflows if, after a key deletion, it contains **m / 2 - 2** keys

- The root node does not underflow. If it contains only one key and this key is deleted, the tree becomes empty.

Deletion in B-Tree

- **Deletion algorithm:**

  If a node underflows, **rotate** the appropriate key from the **adjacent** right- or left-sibling if the sibling contains at least **m / 2** keys; otherwise perform a **merging**.

⇒ A key rotation must always be attempted before a merging

There are **five** deletion cases:

1. The leaf does not underflow.

2. The leaf underflows and the adjacent right sibling has at least m / 2 keys.

   perform a left key-rotation

3. The leaf underflows and the adjacent left sibling has at least m / 2 keys.

   perform a right key-rotation

4. The leaf underflows and each of the adjacent right sibling and the adjacent left sibling has at least m / 2 keys.

   perform either a left or a right key-rotation

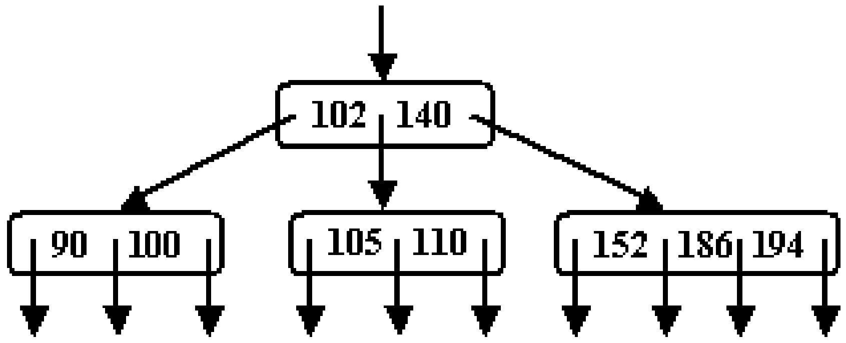
5. The leaf underflows and each adjacent sibling has m / 2 - 1 keys.

Deletion in B-Tree

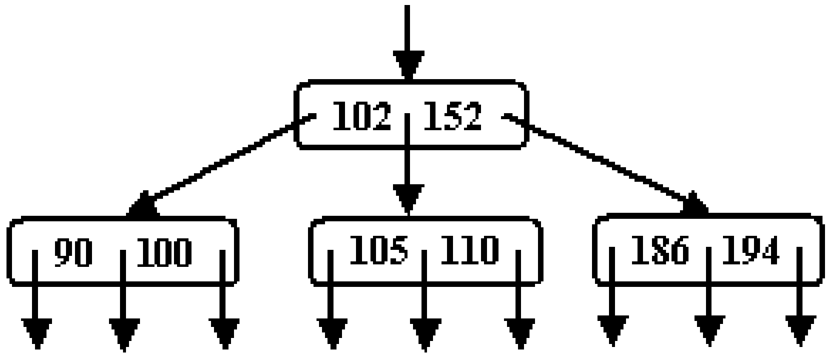
Case1: The leaf does not underflow.

Example:

B-tree of order 4



Delete 140

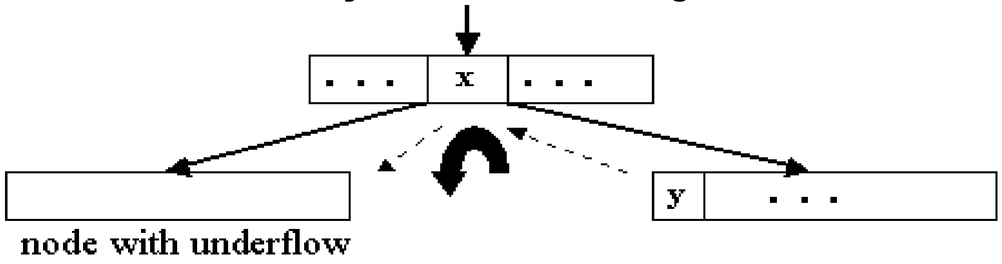


Deletion in B-Tree (cont'd)

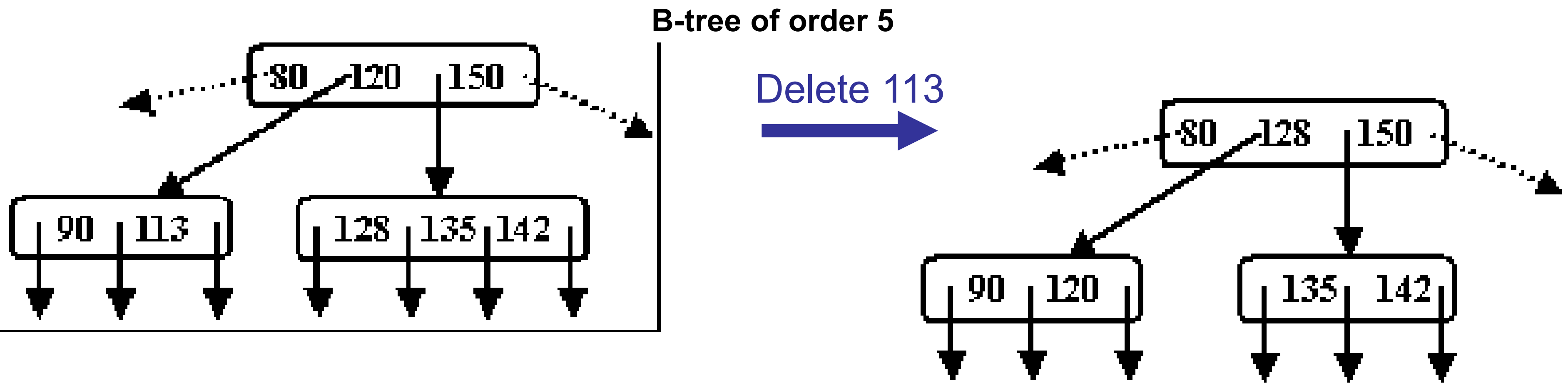
Case2: The leaf underflows and the adjacent right sibling has at least m / 2 keys.

Perform a left key-rotation:

1. Move the parent key **x** that separates the siblings to the node with underflow
2. Move **y**, the minimum key in the right sibling, to where the key **x** was
3. Make the old left subtree of **y** to be the new right subtree of **x**.



Example:

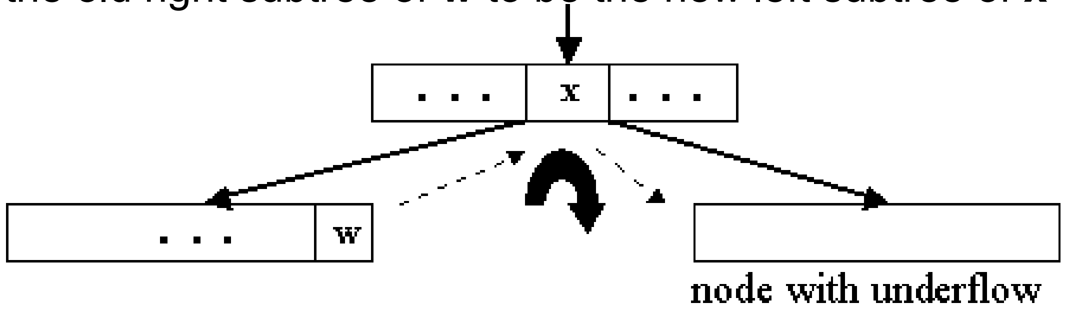


Deletion in B-Tree (cont'd)

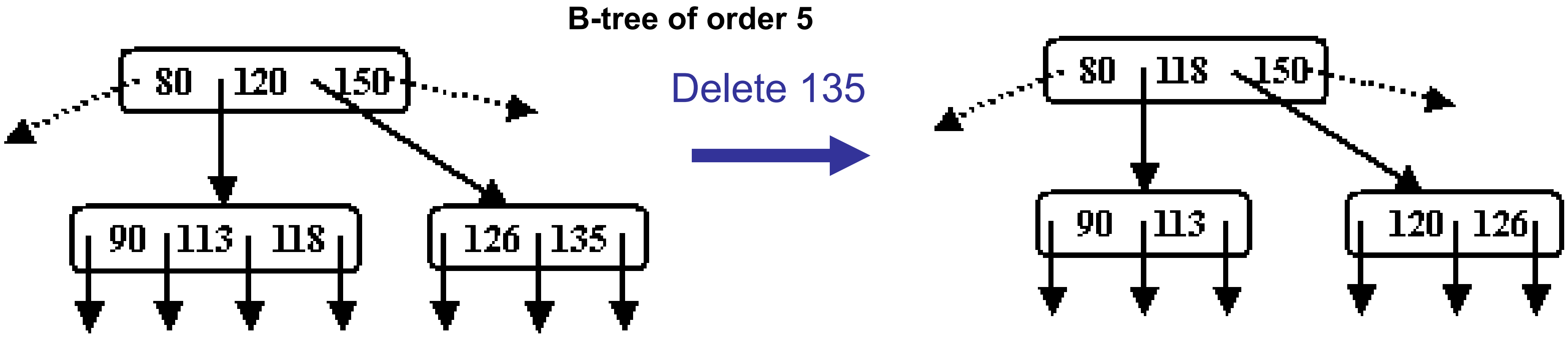
Case 3: The leaf underflows and the adjacent left sibling has at least m / 2 keys.

Perform a right key-rotation:

1. Move the parent key **x** that separates the siblings to the node with underflow
2. Move **w**, the maximum key in the left sibling, to where the key **x** was
3. Make the old right subtree of **w** to be the new left subtree of **x**

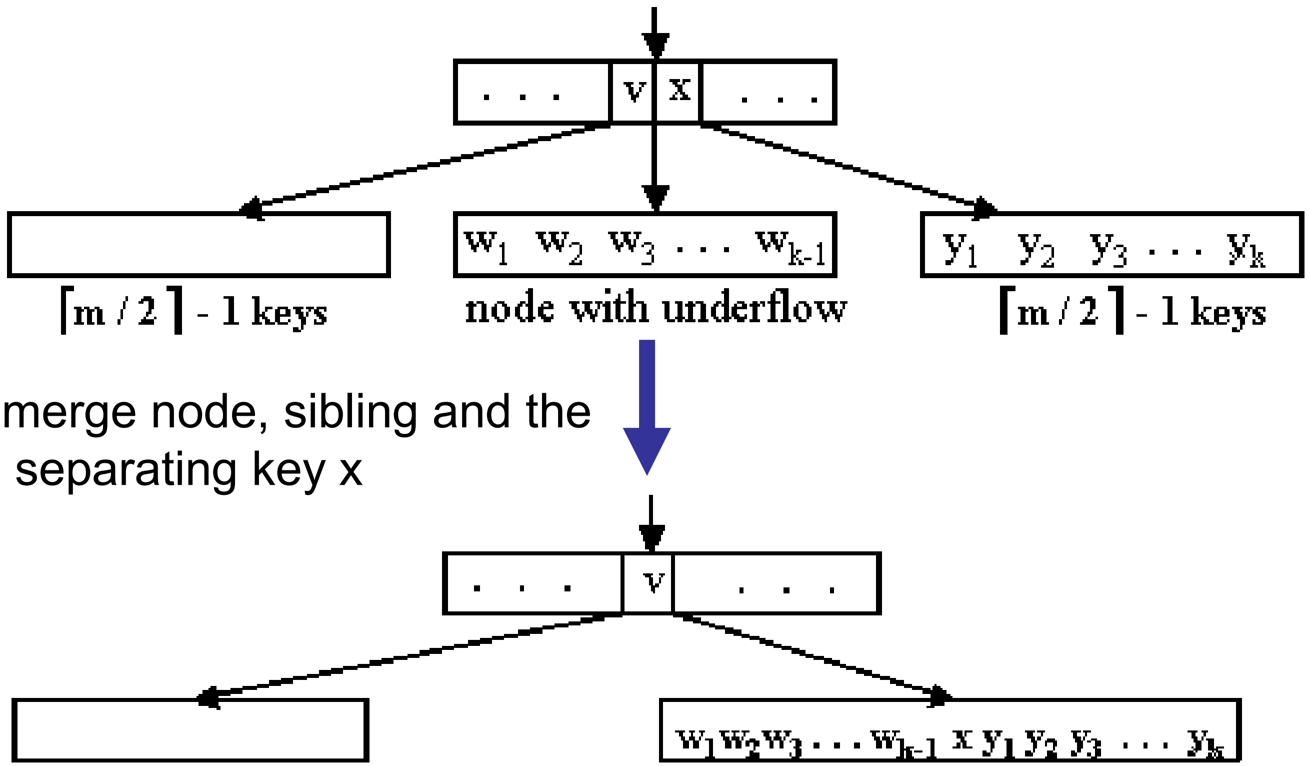


Example:



Deletion in B-Tree (cont'd)

Case 5:The leaf underflows and each adjacent sibling has m / 2 - 1 keys.



. . . v x . . .
w_1 w_2 w_3 . . . w_{k-1}
y_1 y_2 y_3 . . . y_k
$\lceil m / 2 \rceil$ - 1 keys
node with underflow
$\lceil m / 2 \rceil$ - 1 keys
. . . v . . .
$w_1 w_2 w_3$ . . . w_{k-1} x y_1 y_2 y_3 . . . y_k

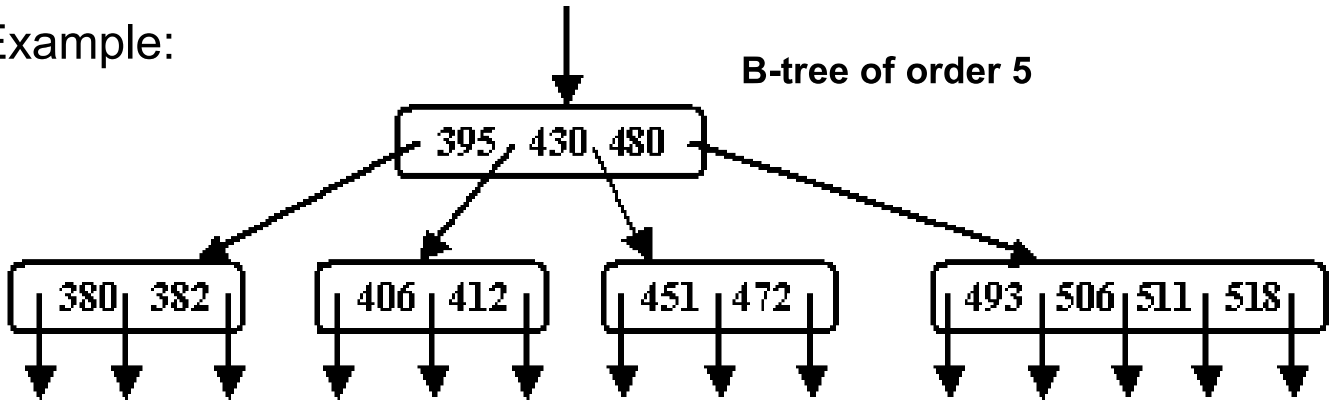
merge node, sibling and the separating key x

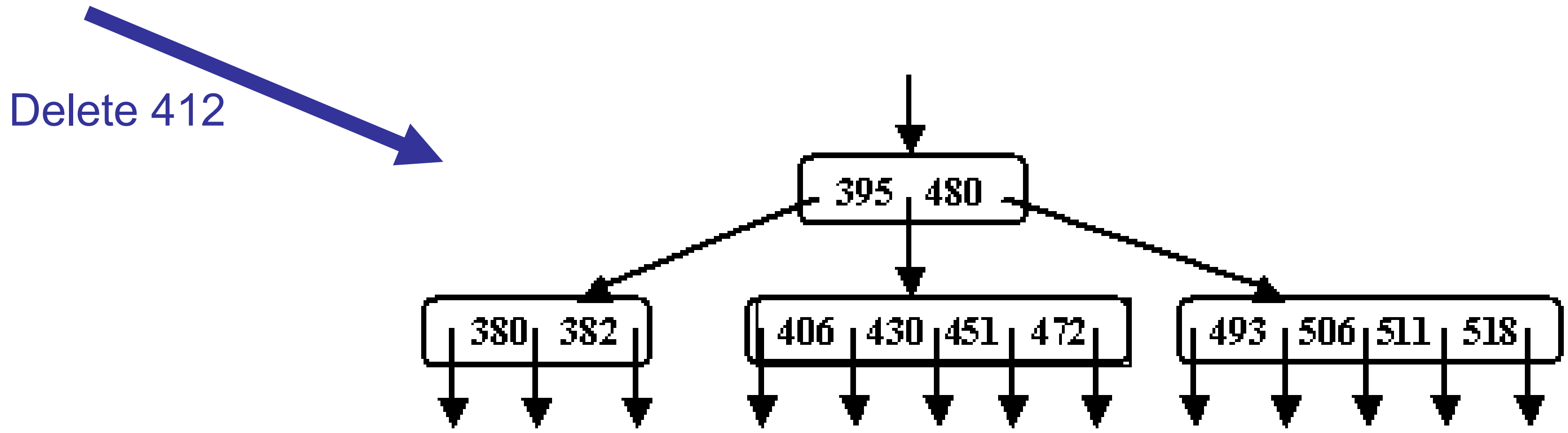
If the parent of the merged node underflows, the merging process propagates upward. In the limit, a root with one key is deleted and the height decreases by one.

Note: The merging could also be done by using the left sibling instead of the right sibling.

Deletion in B-Tree (cont'd)

Example:

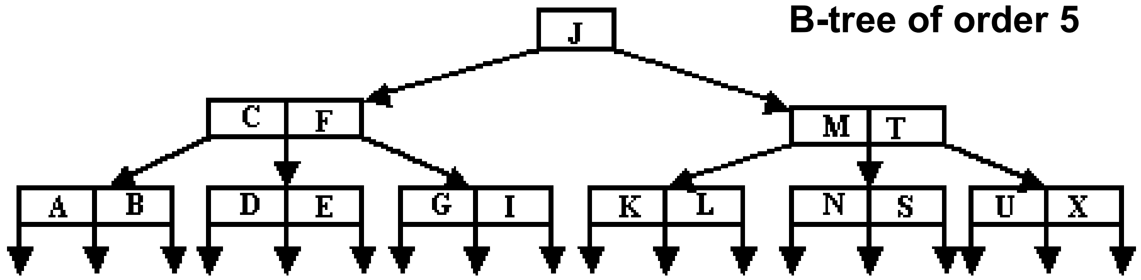


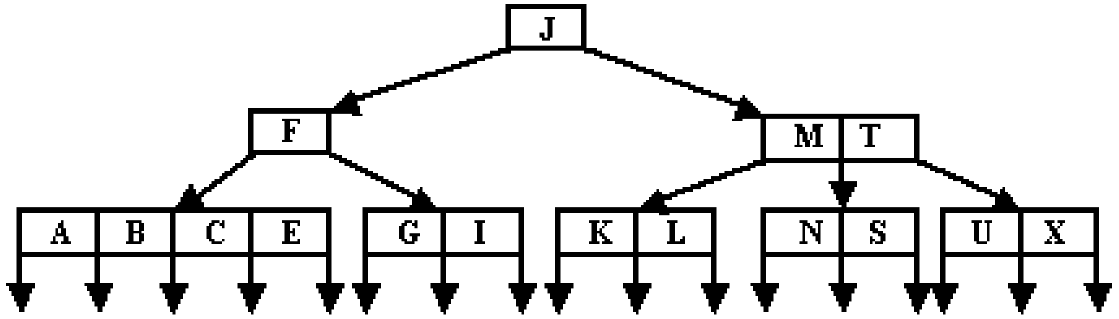


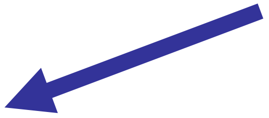
The parent of the merged node does not underflow. The merging process does not propagate upward.

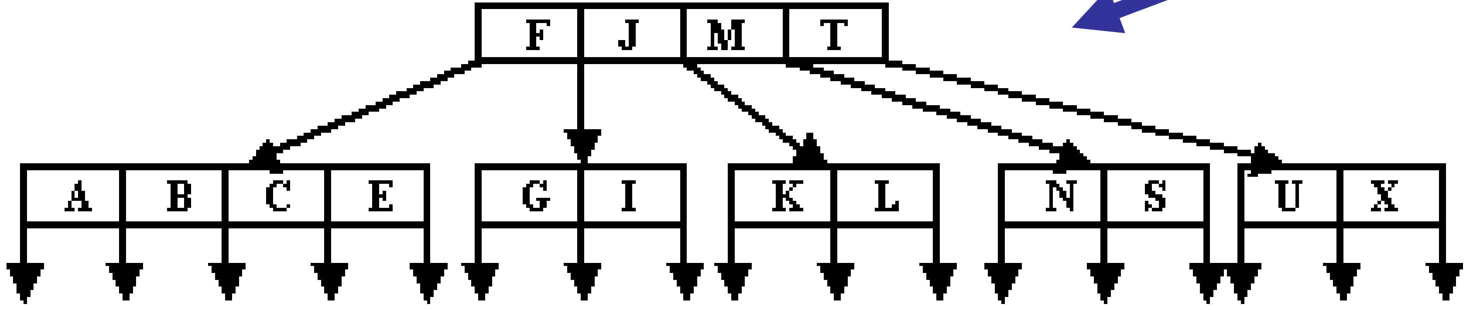
Deletion in B-Tree (cont'd)

Example:



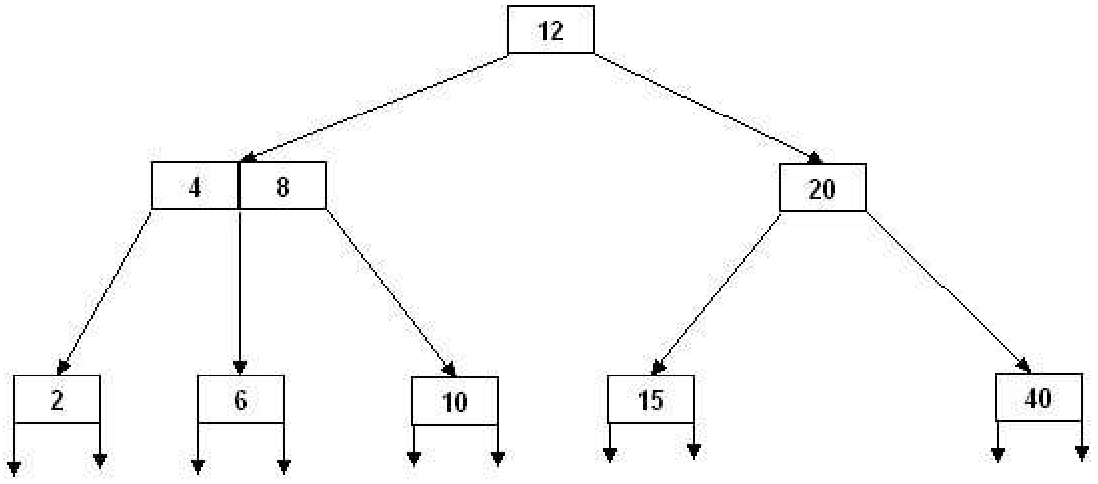


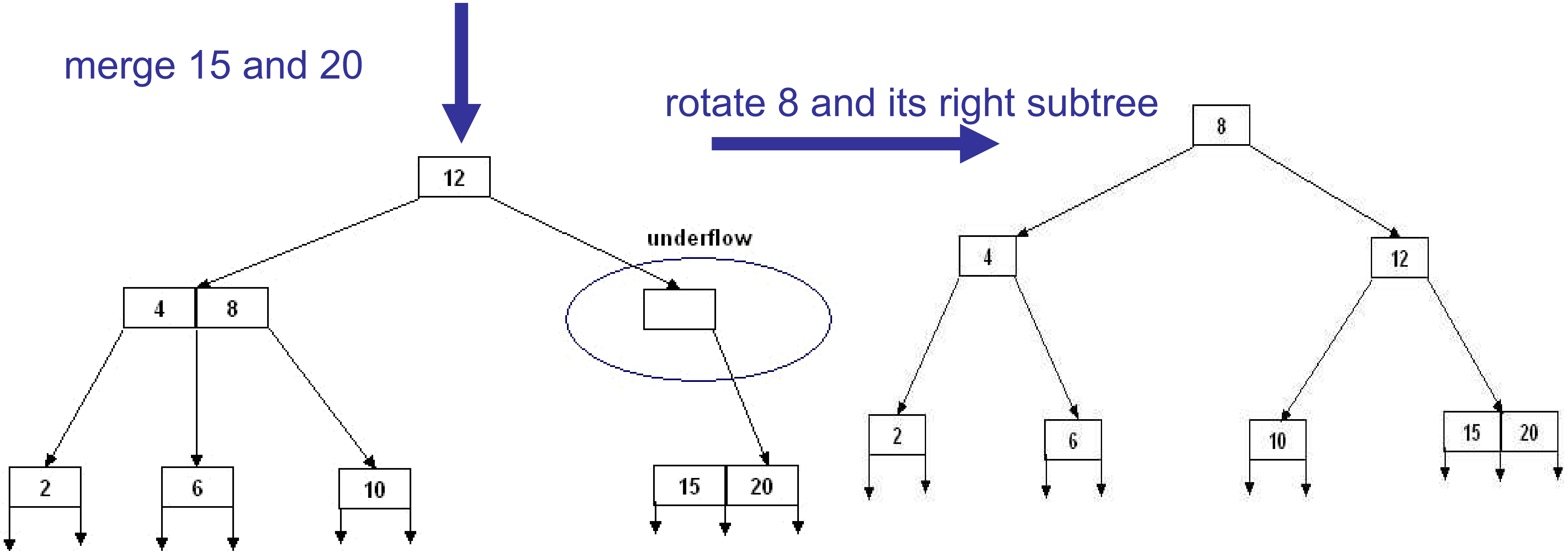




Deletion : Special Case, involves rotation and merging

Example: Delete the key **40** in the following B-tree of order **3**:

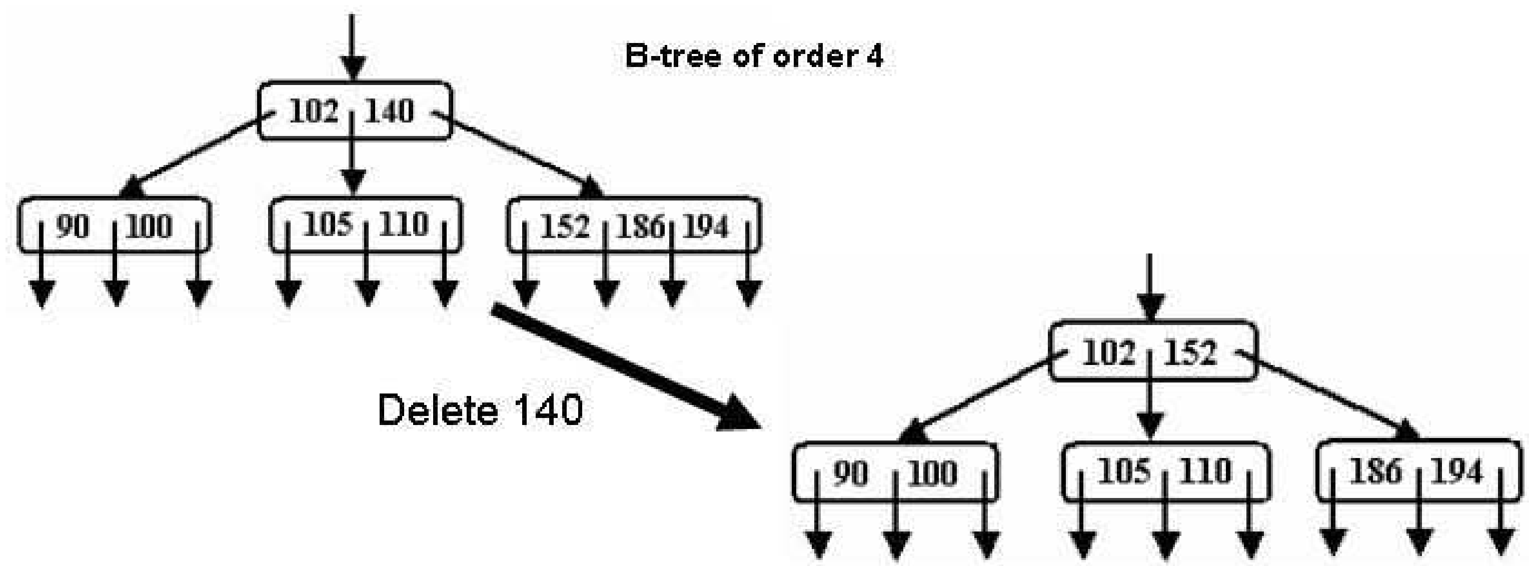




Deletion of a non-leaf node

Deletion of a non-leaf key can always be done in two different ways: by first swapping the key with its successor or predecessor. The resulting trees may be similar or they may be different.

Example: Delete the key 140 in the following partial B-tree of order **4:**



B-Tree Deletion Algorithm

```
deleteKey (x) {
   if (the key x to be deleted is not in the tree)
        throw an appropriate exception;
   if (the tree has only one node) {
     delete x ;
     return;
   }
   if (the key x is not in a leaf node)
         swap x with its successor or predecessor;// each will be in a leaf node
   delete x from the leaf node;
   if(the leaf node does not underflow) // after deletion numKeys      m / 2  - 1
        return;
    let the leaf node be the CurrentNode;
    done = false;
```

B-Tree Deletion Algorithm

```
 while (! done && numKeys(CurrentNode)      m / 2   - 1) {   // there is underflow
       if (any of the adjacent siblings t of the CurrentNode has at least   m / 2   keys) { // ROTATION CASE
              if (t is the adjacent right sibling) {
              •   rotate the separating-parent key w of CurrentNode and t to CurrentNode;
              •   rotate the minimum key of t to the previous parent-location of w;
              •   rotate the left subtree of t, if any, to become the right-most subtree of CurrentNode;
               }
                else {      // t is the adjacent left sibling
              •   rotate the separating-parent key w between CurrentNode and t to CurrentNode;
              •   rotate the maximum key of t to the previous parent-location of w;
              •   rotate the right subtree of t , if any, to become the left-most subtree of CurrentNode;
              }
              done = true;
        }
        else { // MERGING CASE: the adjacent or each adjacent sibling has   m / 2   - 1 keys
select any adjacent sibling t of CurrentNode;

create a new sibling by merging currentNode, the sibling t, and their parent-separating key ;

If (parent node p is the root node) {

         if (p is empty after the merging)

             make the merged node the new root;

                done = true;
              } else
          let parent p be the CurrentNode;
          }
     }  // while
     return;
```

Max Heap Datastructure

Heap data structure is a specialized binary tree-based data structure. Heap is a binary tree with special characteristics. In a heap data structure, nodes are arranged based on their values. A heap data structure some times also called as Binary Heap.

There are two types of heap data structures and they are as follows...

1. **Max Heap**
2. **Min Heap**

Every heap data structure has the following properties...

Property #1 (Ordering): Nodes must be arranged in an order according to their values based on Max heap or Min heap.

Property #2 (Structural): All levels in a heap must be full except the last level and all nodes must be filled from left to right strictly.

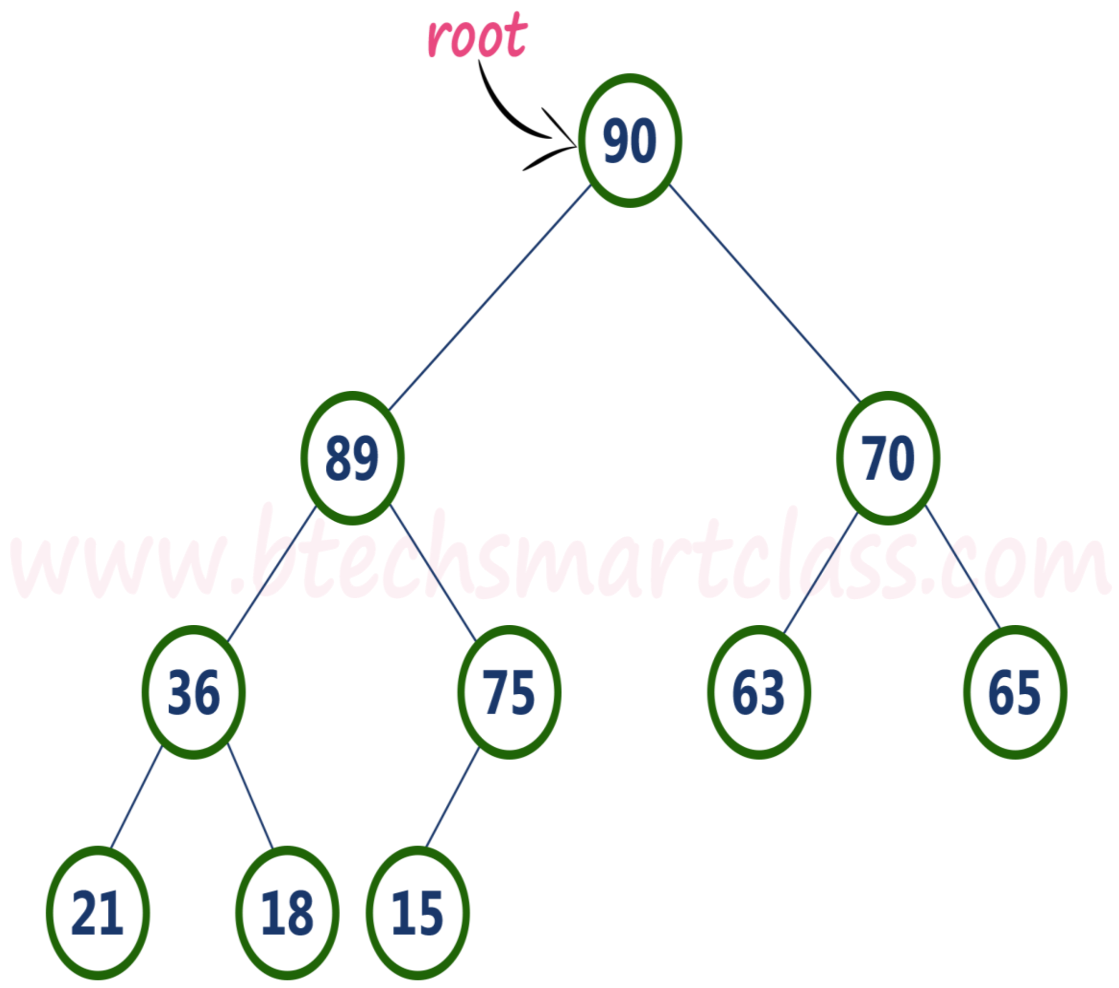
Max Heap

Max heap data structure is a specialized full binary tree data structure. In a max heap nodes are arranged based on node value.

Max heap is defined as follows...

Max heap is a specialized full binary tree in which every parent node contains greater or equal value than its child nodes.

Example



Above tree is satisfying both Ordering property and Structural property according to the Max Heap data structure.

Operations on Max Heap

The following operations are performed on a Max heap data structure...

1. **Finding Maximum**
2. **Insertion**
3. **Deletion**

Finding Maximum Value Operation in Max Heap

Finding the node which has maximum value in a max heap is very simple. In a max heap, the root node has the maximum value than all other nodes. So, directly we can display root node value as the maximum value in max heap.

Insertion Operation in Max Heap

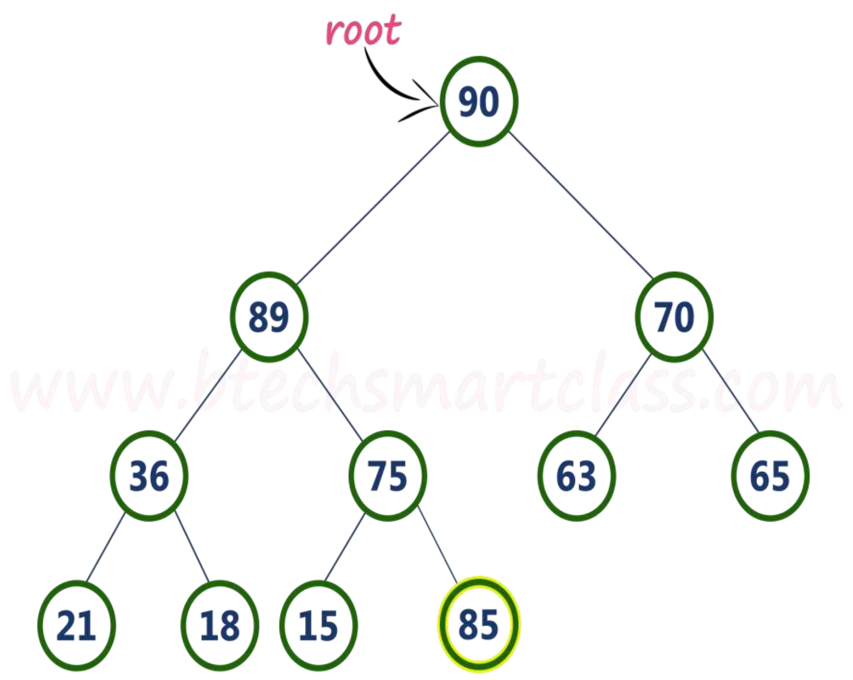
Insertion Operation in max heap is performed as follows...

- **Step 1 -** Insert the **newNode** as **last leaf** from left to right.
- **Step 2 -** Compare **newNode value** with its **Parent node**.
- **Step 3 -** If **newNode value is greater** than its parent, then **swap** both of them.
- **Step 4 -** Repeat step 2 and step 3 until newNode value is less than its parent node (or) newNode reaches to root.

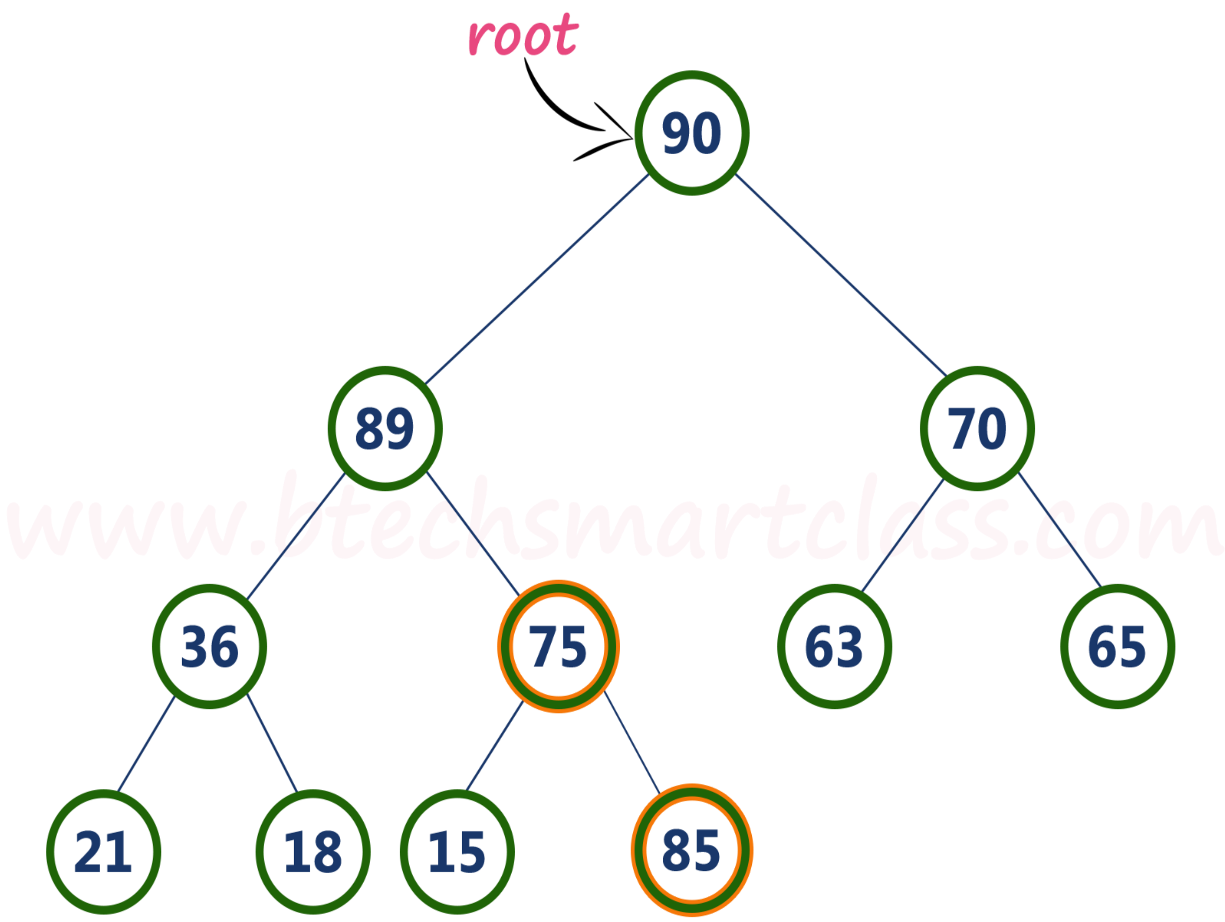
Example

Consider the above max heap. **Insert a new node with value 85.**

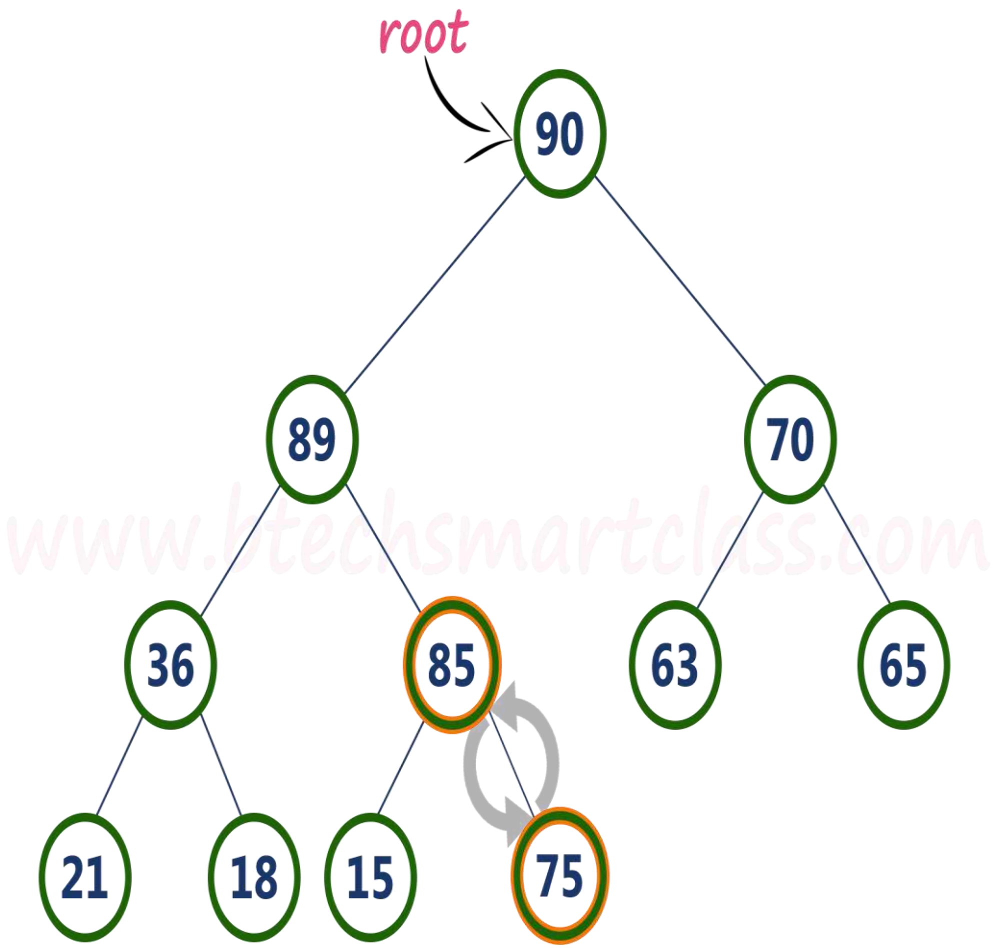
- **Step 1 -** Insert the **newNode** with value 85 as **last leaf** from left to right. That means newNode is added as a right child of node with value 75. After adding max heap is as follows...



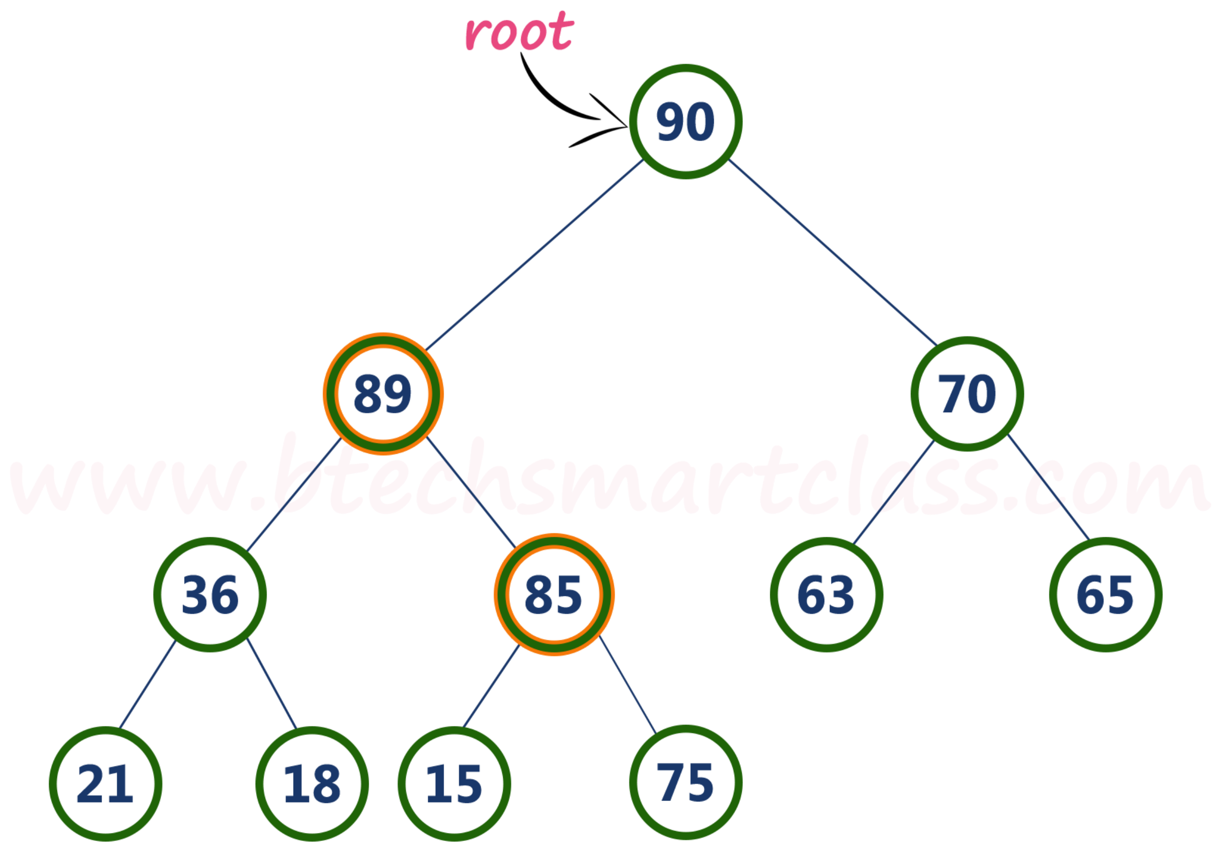
- **Step 2 -** Compare **newNode value (85)** with its **Parent node value (75)**. That means **85 > 75**



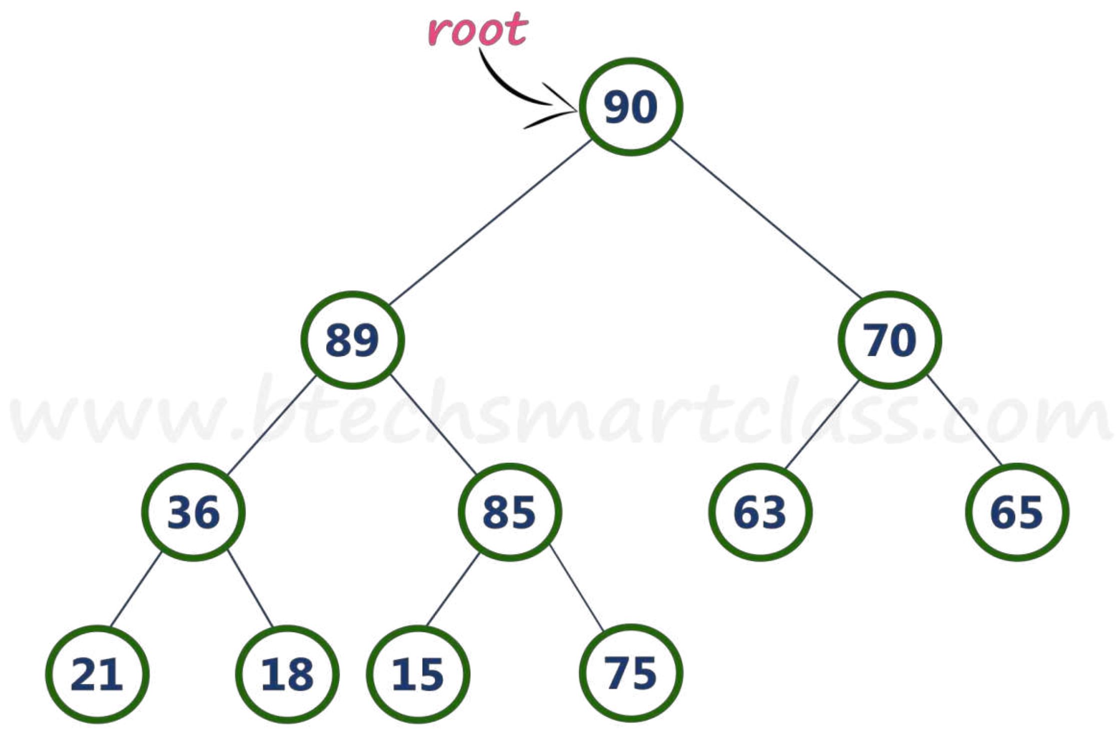
- **Step 3 -** Here **newNode value (85) is greater** than its **parent value (75)**, then **swap** both of them. After swapping, max heap is as follows...



- **Step 4 -** Now, again compare newNode value (85) with its parent node value (89).



Here, newNode value (85) is smaller than its parent node value (89). So, we stop insertion process. Finally, max heap after insertion of a new node with value 85 is as follows...



Deletion Operation in Max Heap

In a max heap, deleting the last node is very simple as it does not disturb max heap properties.

Deleting root node from a max heap is little difficult as it disturbs the max heap properties. We use the following steps to delete the root node from a max heap...

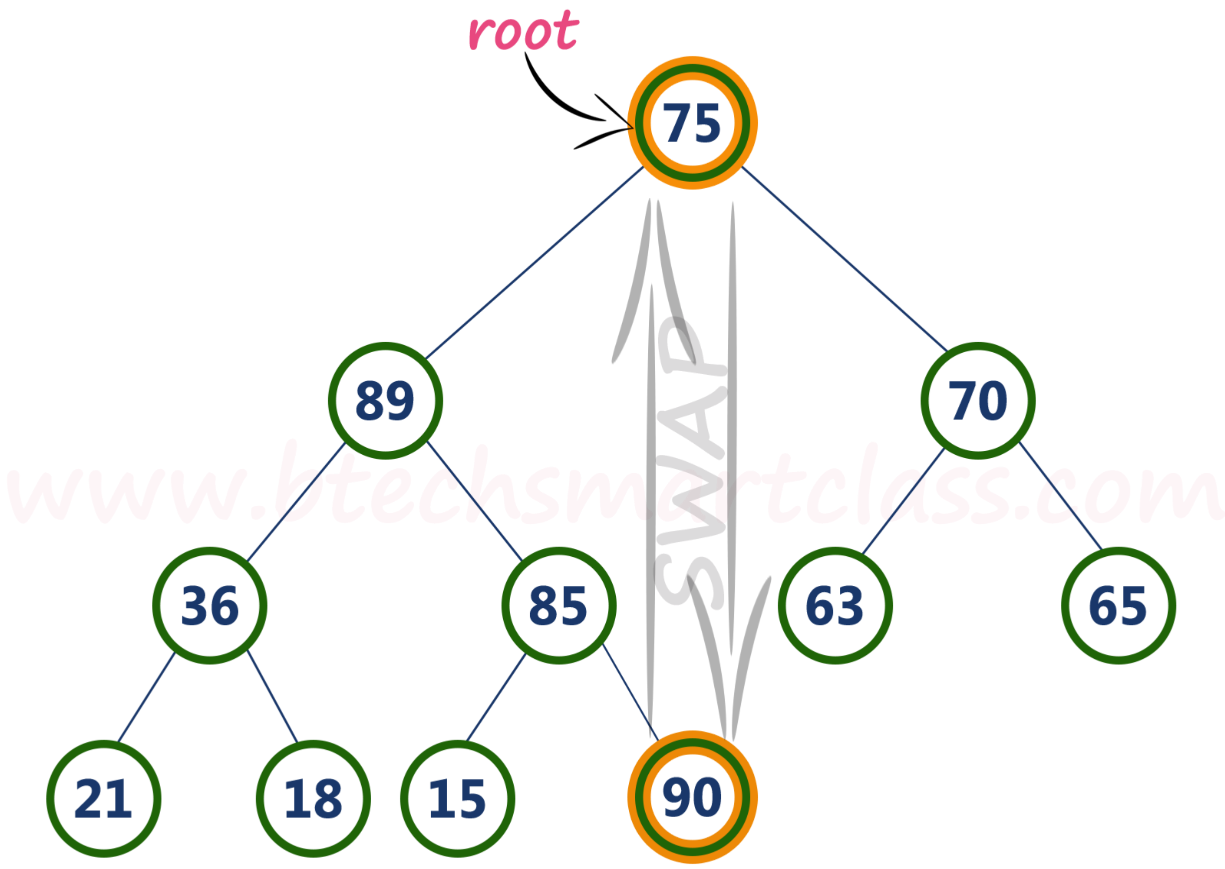
- **Step 1 - Swap** the **root** node with **last** node in max heap
- **Step 2 - Delete** last node.
- **Step 3 -** Now, compare **root value** with its **left child value**.
- **Step 4 -** If **root value is smaller** than its left child, then compare **left child** with its **right sibling**. Else goto **Step 6**

- **Step 5 -** If **left child value is larger** than its **right sibling**, then **swap root** with **left child** otherwise **swap root** with its **right child**.
- **Step 6 -** If **root value is larger** than its left child, then compare **root value** with its **right child** value.
- **Step 7 -** If **root value is smaller** than its **right child**, then **swap root** with **right child** otherwise **stop the process**.
- **Step 8 -** Repeat the same until root node fixes at its exact position.

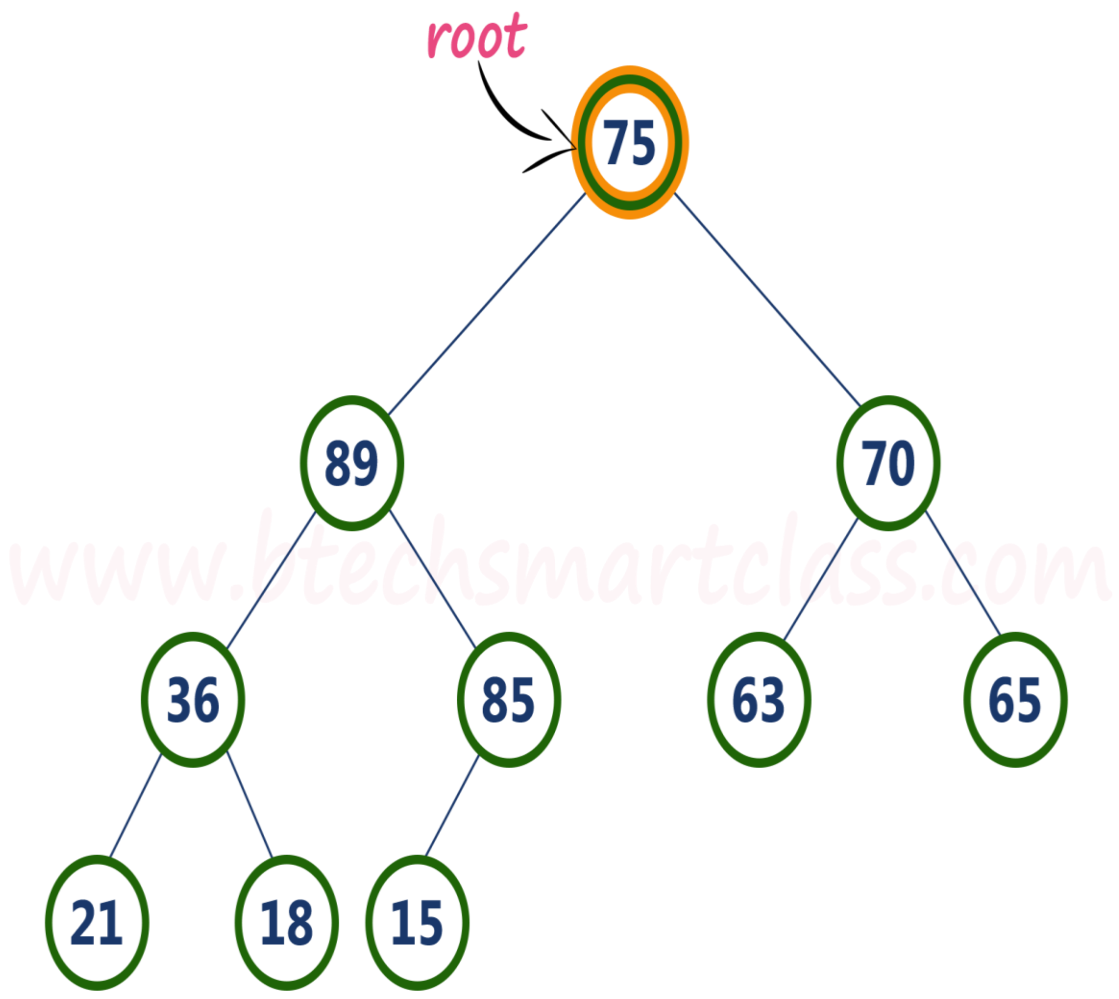
Example

Consider the above max heap. **Delete root node (90) from the max heap.**

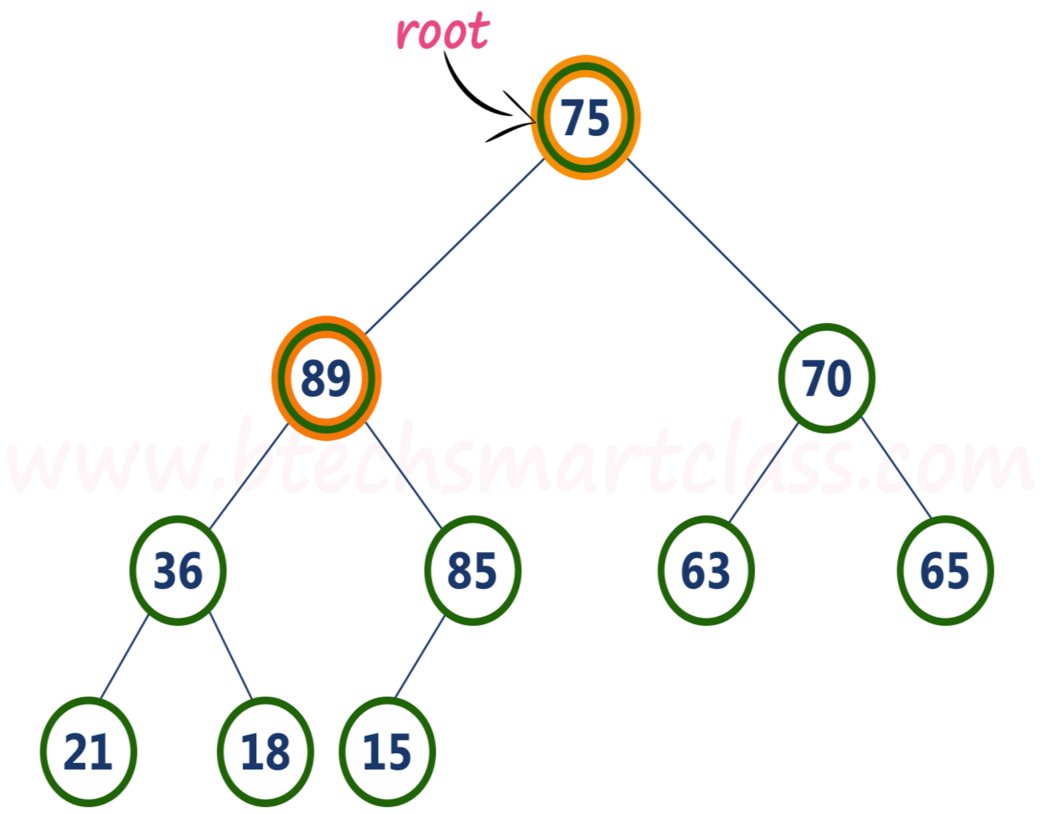
- **Step 1 - Swap** the **root node (90)** with **last node 75** in max heap. After swapping max heap is as follows...



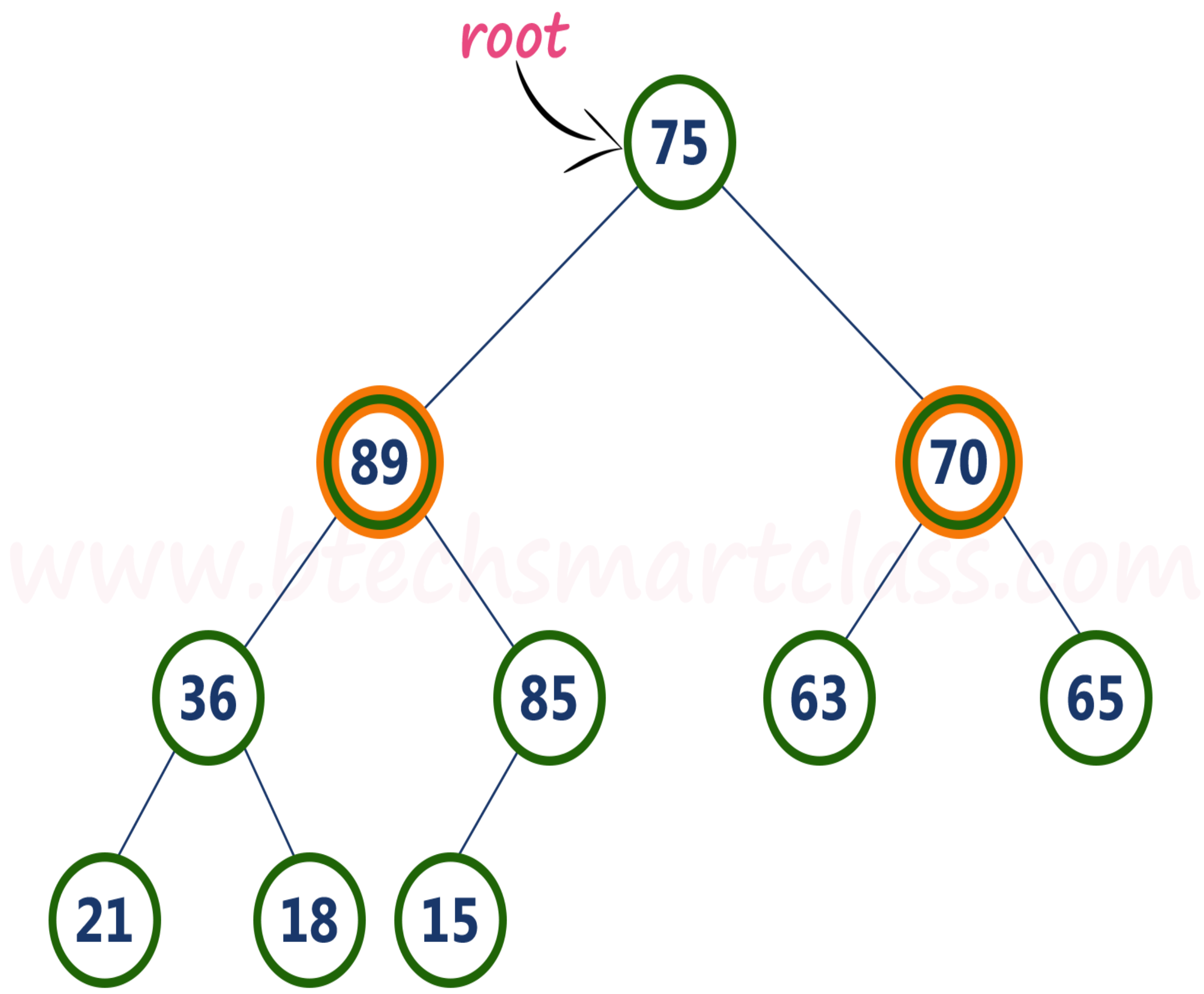
- **Step 2 - Delete** last node. Here the last node is 90. After deleting node with value 90 from heap, max heap is as follows...



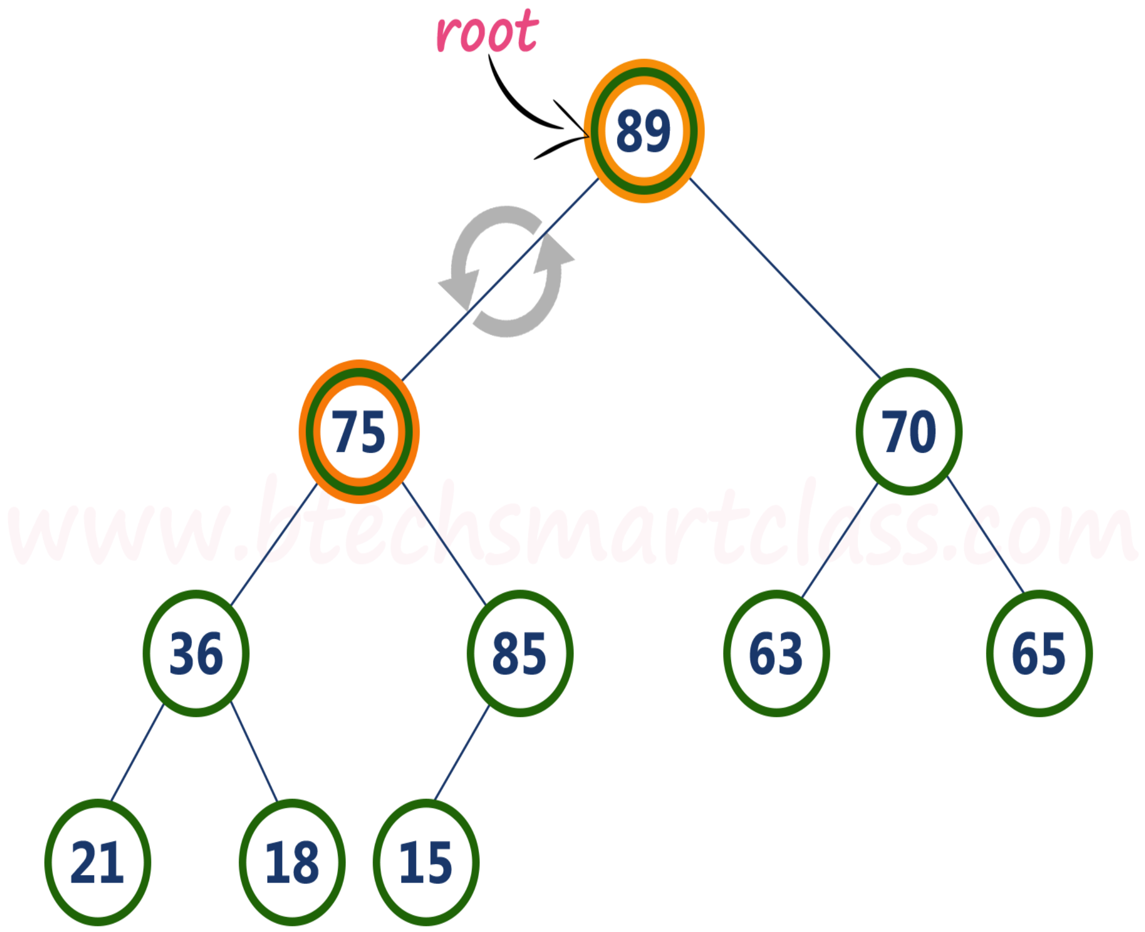
- **Step 3 -** Compare **root node (75)** with its **left child (89).**



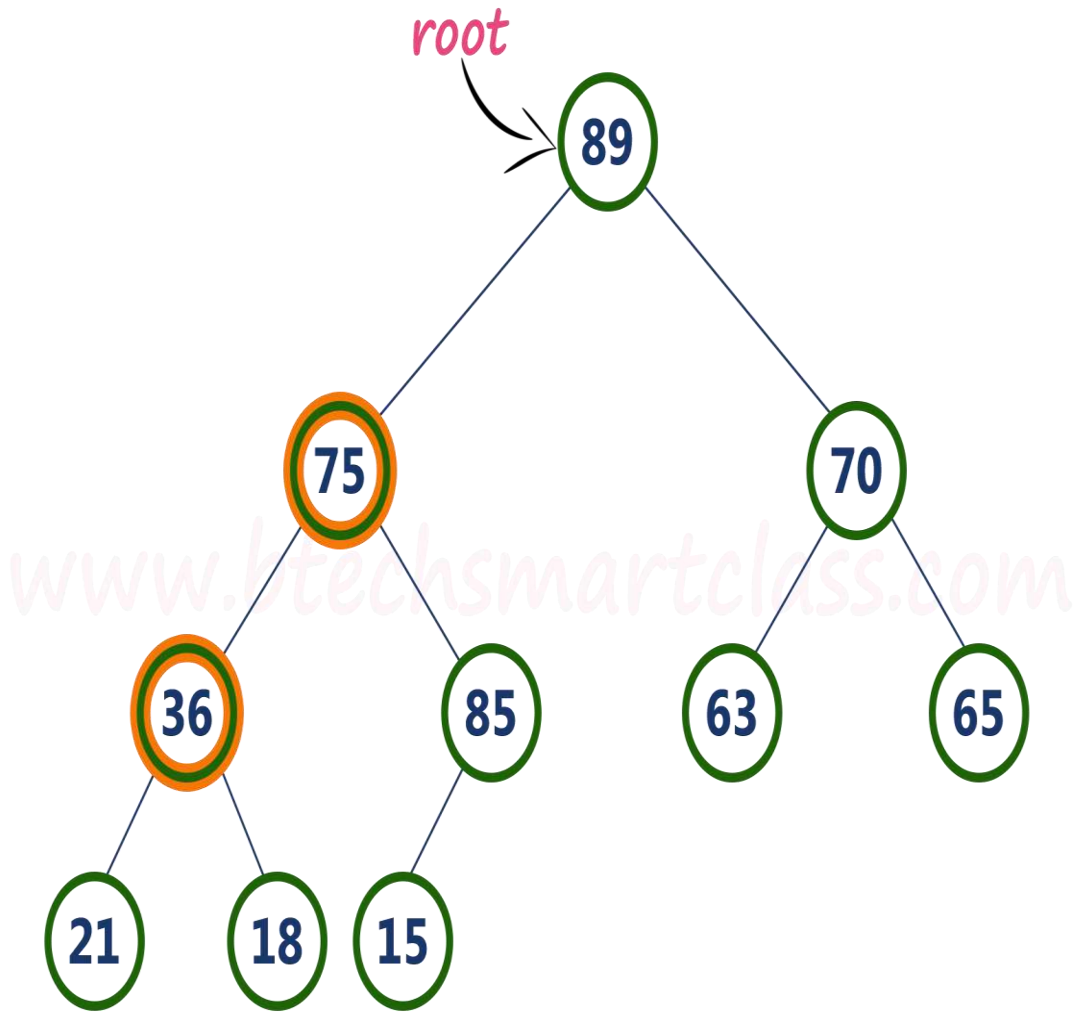
Here, **root value (75) is smaller** than its left child value (89). So, compare left child (89) with its right sibling (70).



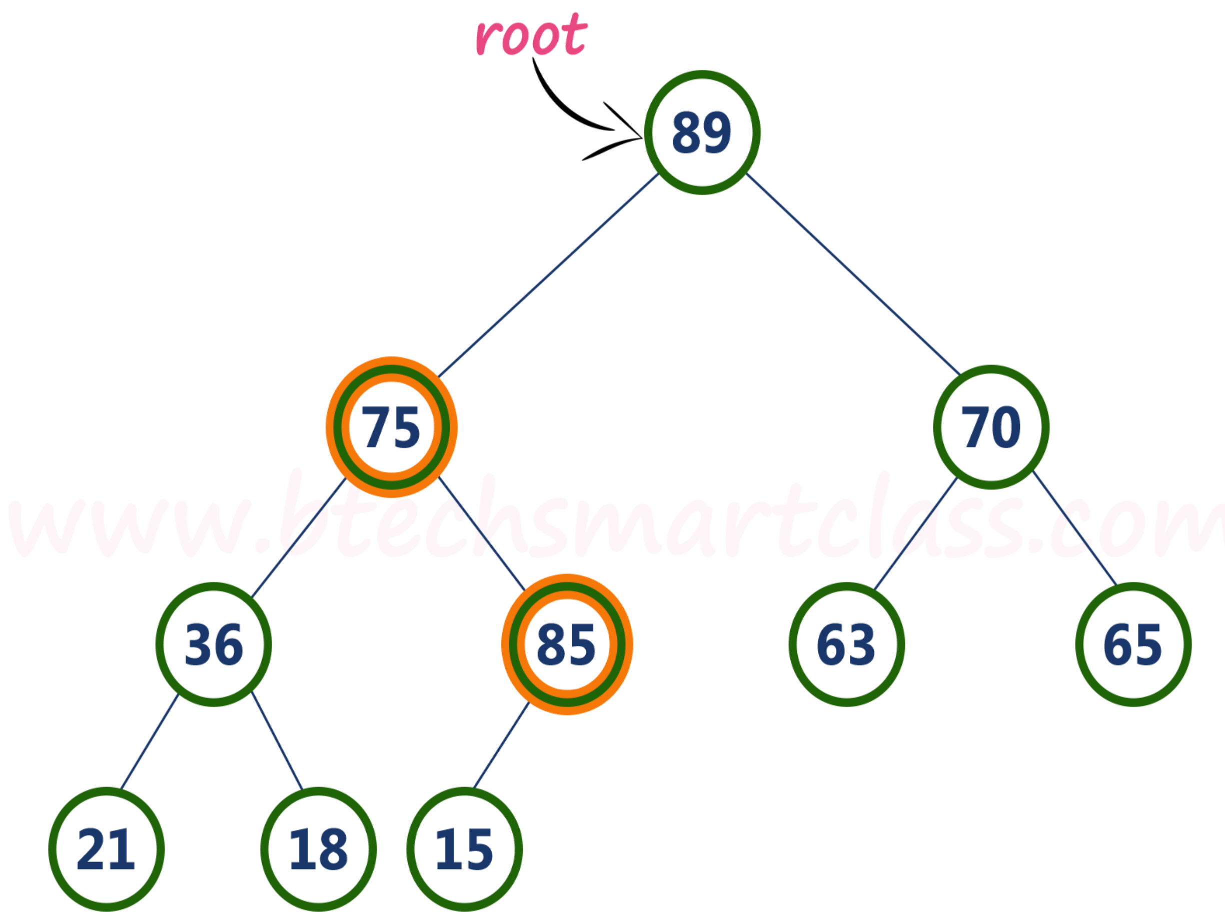
- **Step 4 -** Here, **left child value (89) is larger** than its **right sibling (70)**, So, **swap root (75)** with **left child (89).**



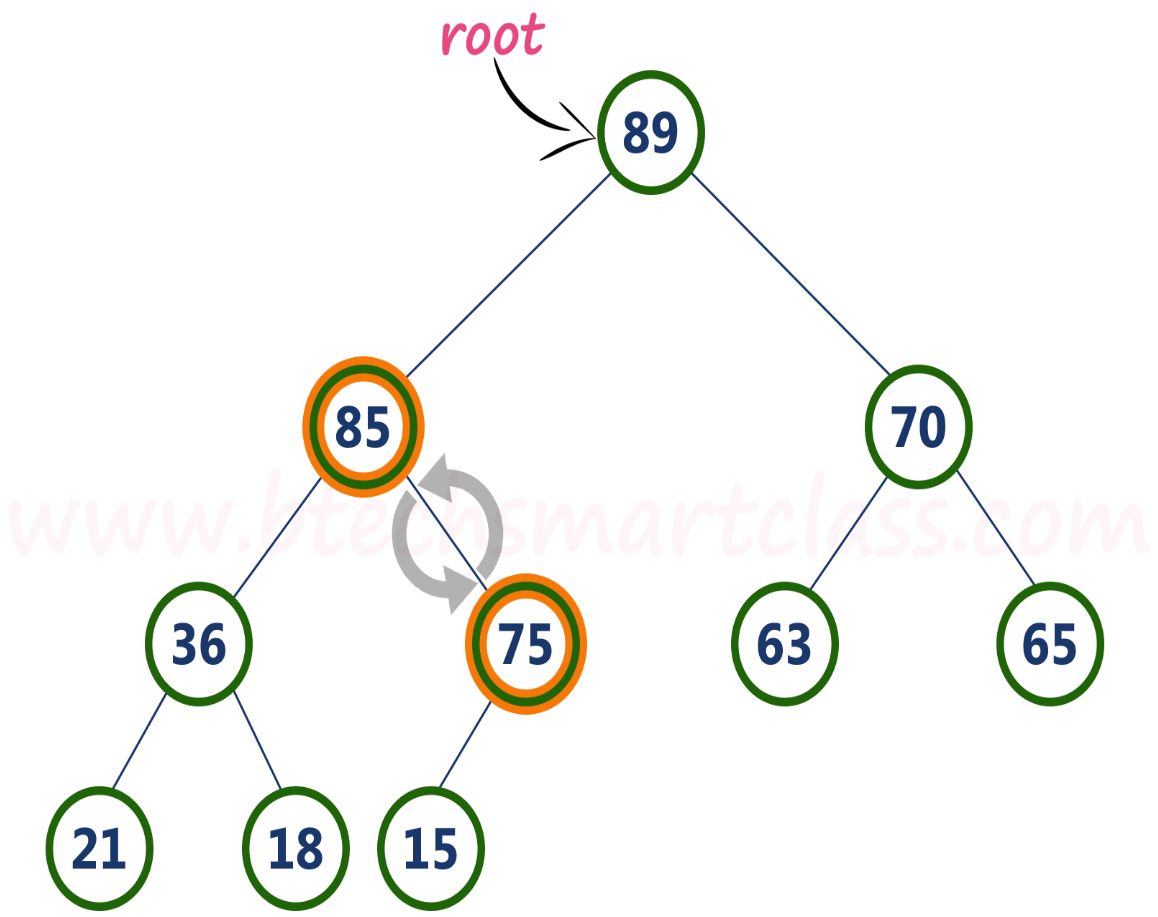
- **Step 5 -** Now, again compare **75** with its **left child (36)**.



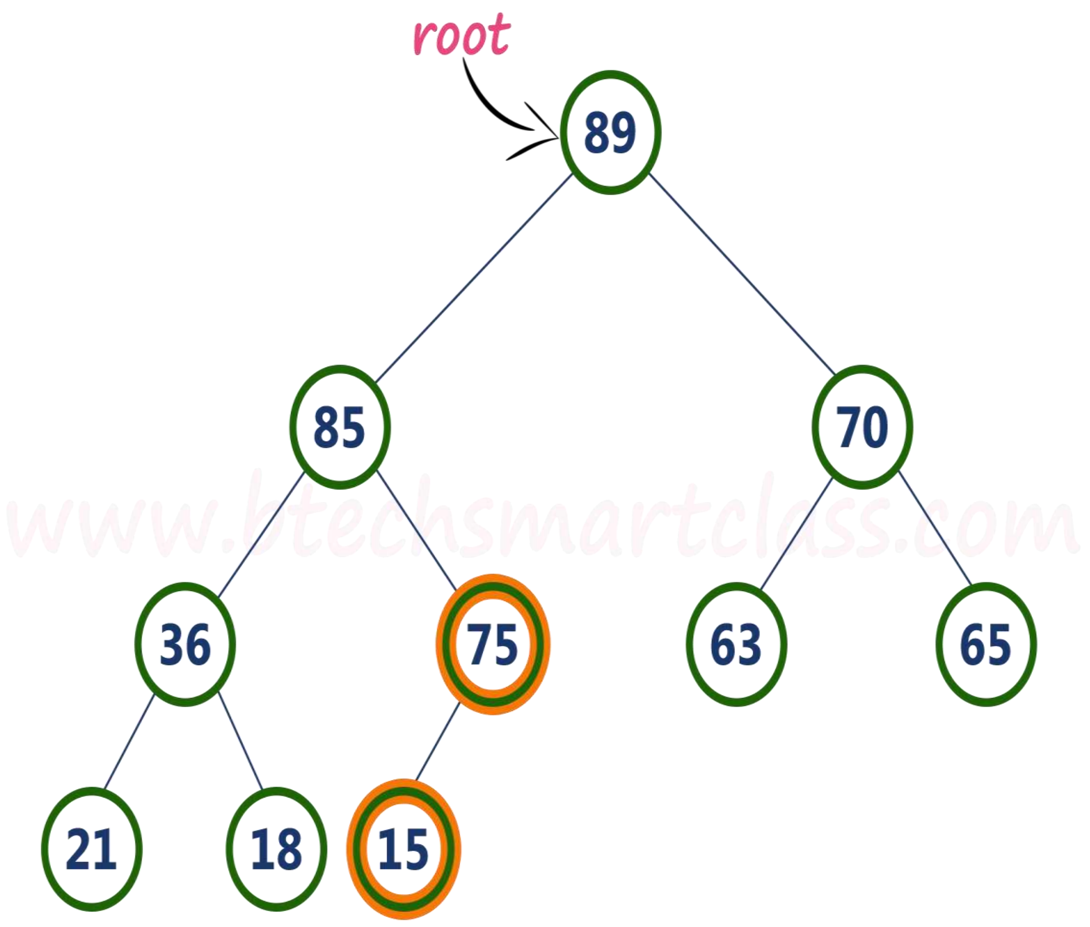
Here, node with value **75** is larger than its left child. So, we compare node **75** with its right child **85.**



- **Step 6 -** Here, node with value **75** is smaller than its **right child (85)**. So, we swap both of them. After swapping max heap is as follows...

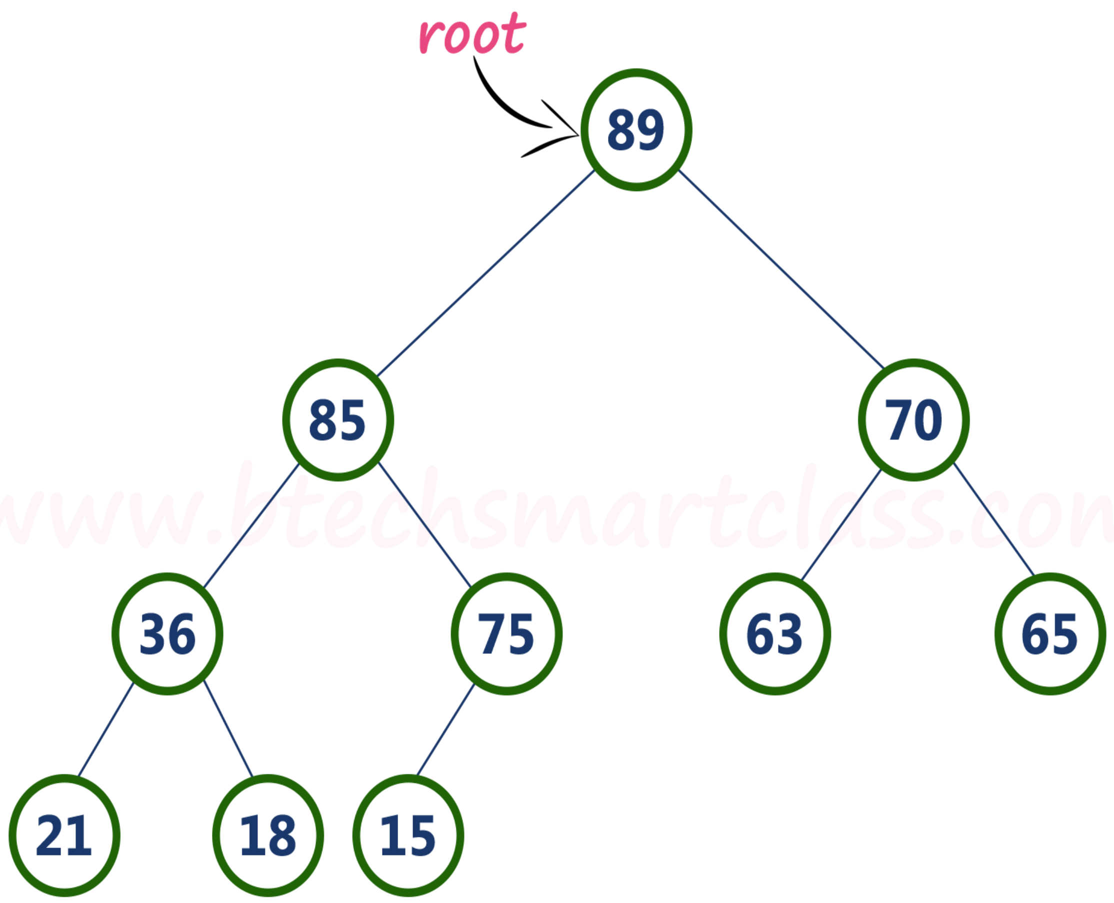


- **Step 7 -** Now, compare node with value **75** with its left child (**15**).



Here, node with value **75** is larger than its left child (**15**) and it does not have right child. So we stop the process.

Finally, max heap after deleting root node (**90**) is as follows...


root
89
85
70
36
75
63
65
21
18
15