EE2008: Data Structures and Algorithms

Recursive Algorithms and Trees

Prof Huang Guangbin

S2.1-B2-06

egbhuang@ntu.edu.sg

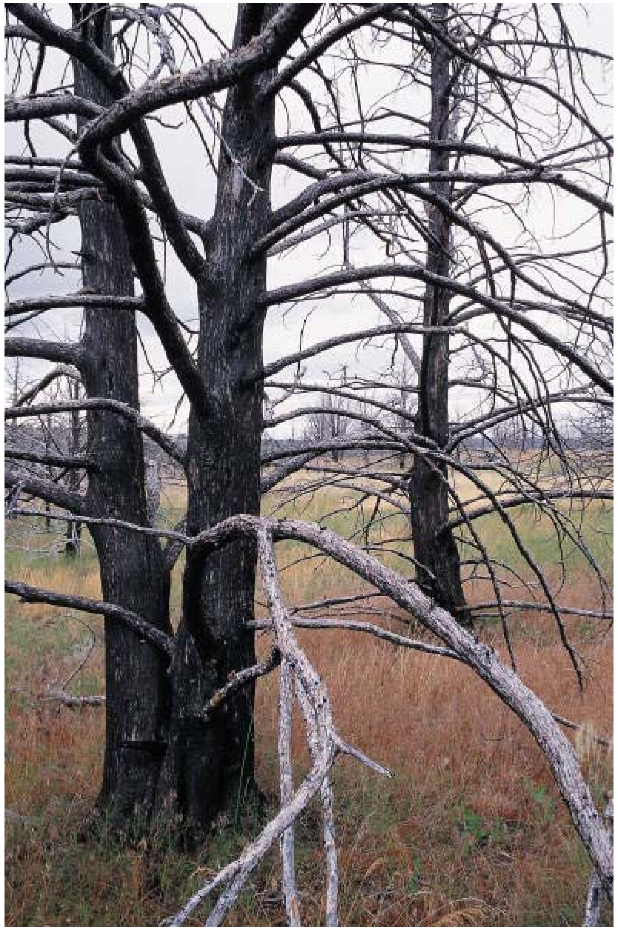
Topics

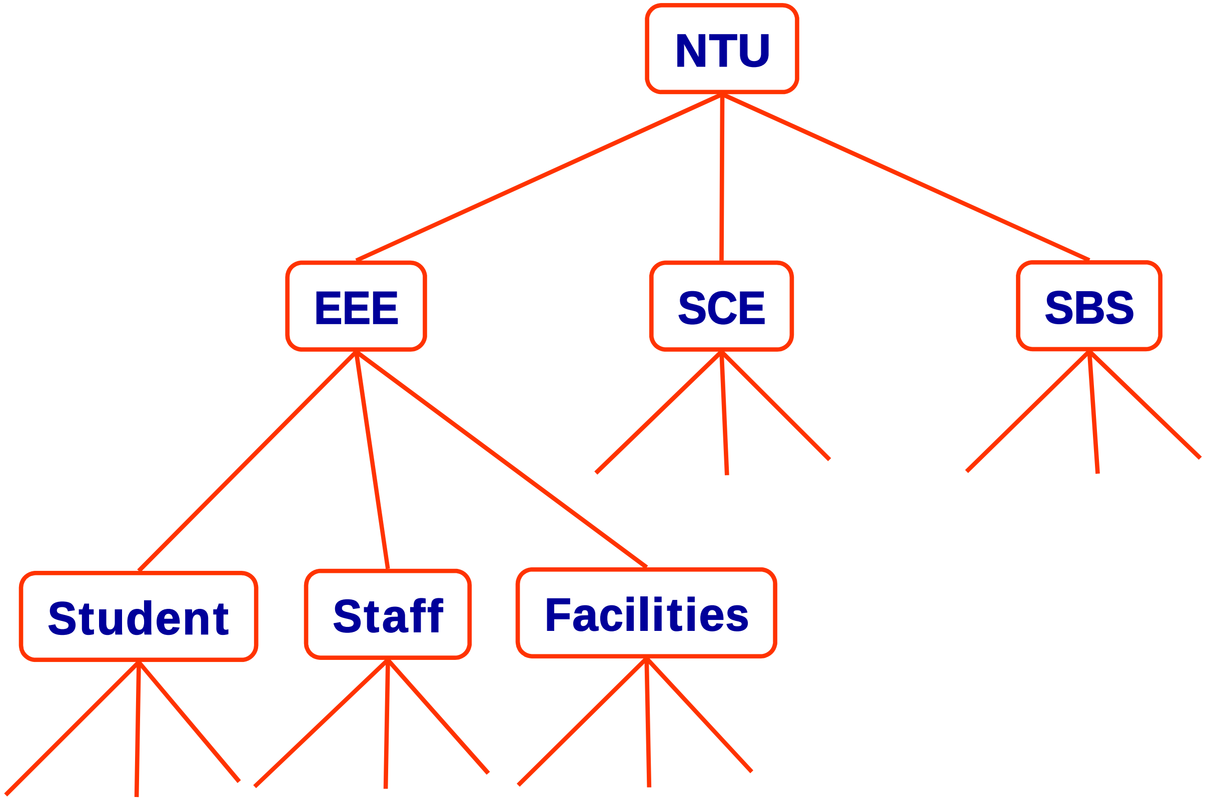
- Recursive Algorithms, Divide and Conquer Algorithms
- Tree data structures
  - Binary tree: Binary search trees, AVL trees, Heaps
- Sorting algorithms
  - Mergesort
  - Quicksort
  - Bucket Sort
  - Radix Sort
  - Heap Sort
- Selection problem and order statistics

Data Structures (review)

- **Definition: Data structure is a way to store and organize data in order to facilitate access and modifications.**
  - **No single data structure works well for all purposes, and so it is important to know the strengths and limitations of different data structures in different applications.**
- **Two kinds of information in a data structure**
  - **Structural (organizational) information (such as “index” for array)**
  - **Data being organized (such as specific data “5” stored in a array)**
- **How do we learn “data structures”: Two key issues**
  - **What are distinguishing “structural” properties (features)**
  - **How to manipulate the structure (algorithms)**

Trees

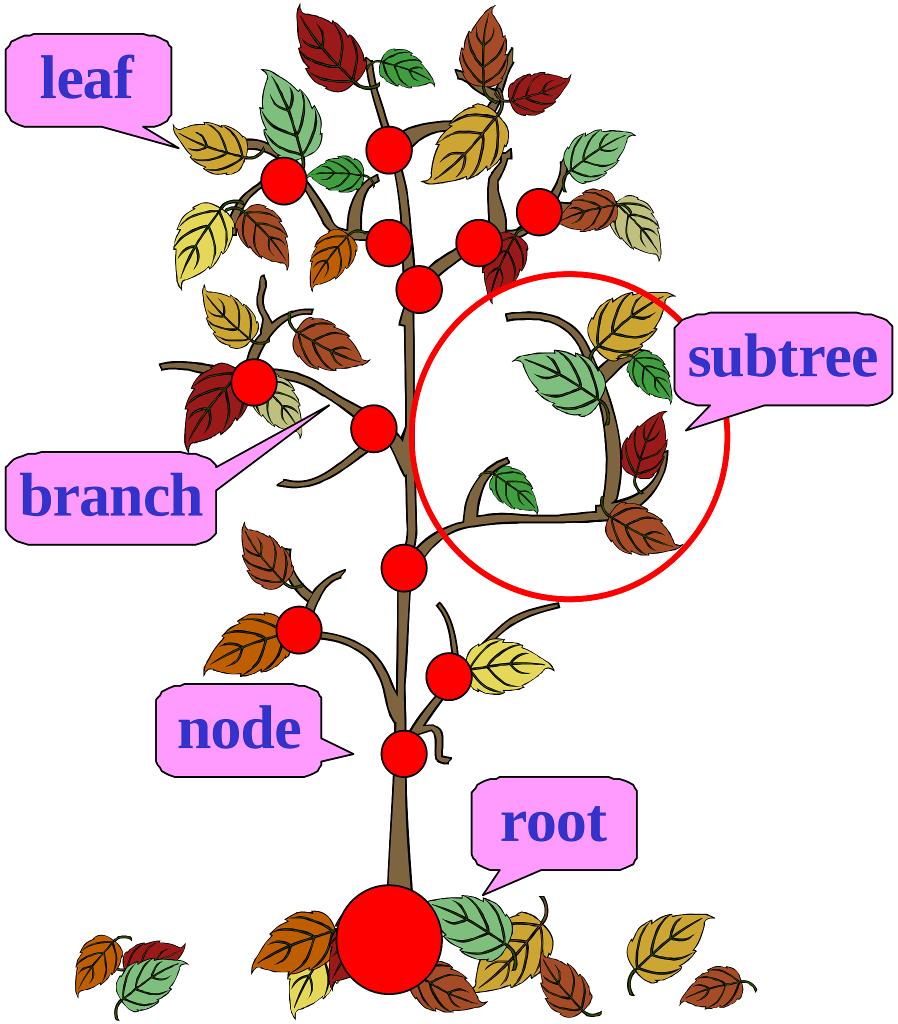


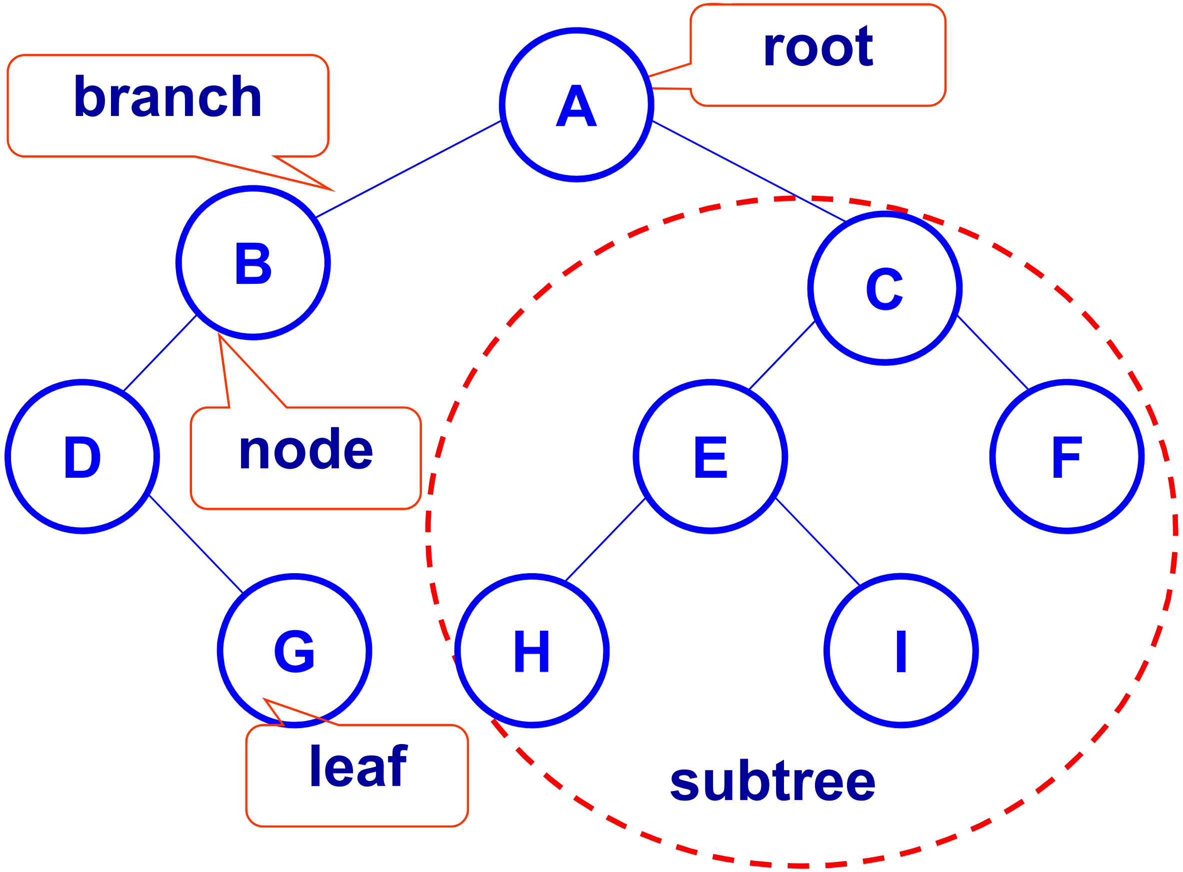


What is a Tree

- **In computer science, a tree is an abstract model of a hierarchical structure**
- **A tree consists of nodes with a parent-child relation**
- **Applications:**
  - ☞ **Organization charts**
  - ☞ **File systems**
    - ✓ **How to efficiently search a file with specific names/types?**

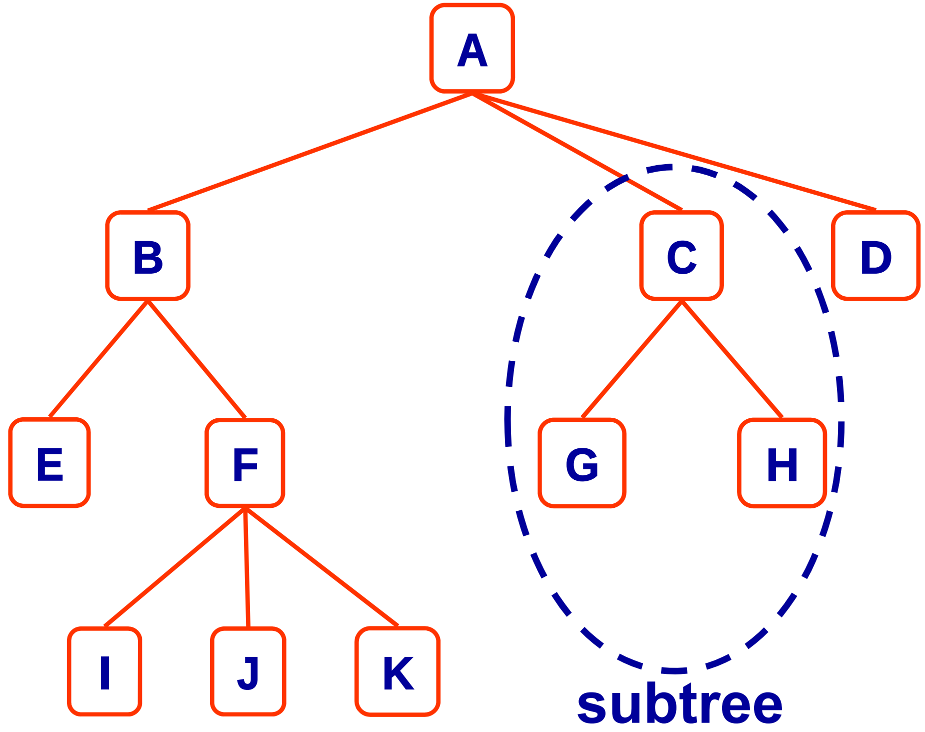
Trees: Terminology





Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (e.g, 3)
- Degree: The number of children of a node
- Subtree: tree consisting of a node and its descendants



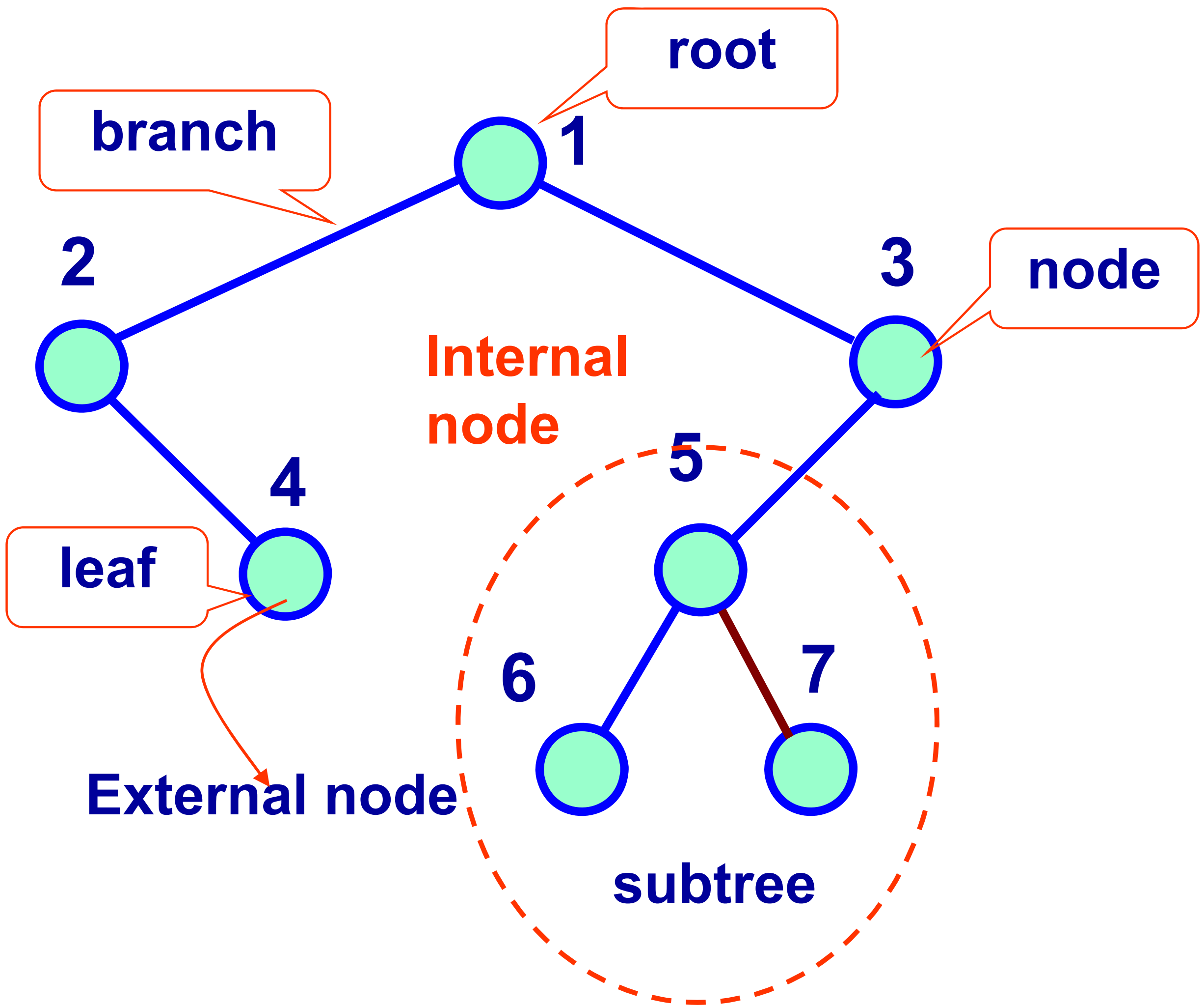
✓ This is a general rooted tree

Binary Trees

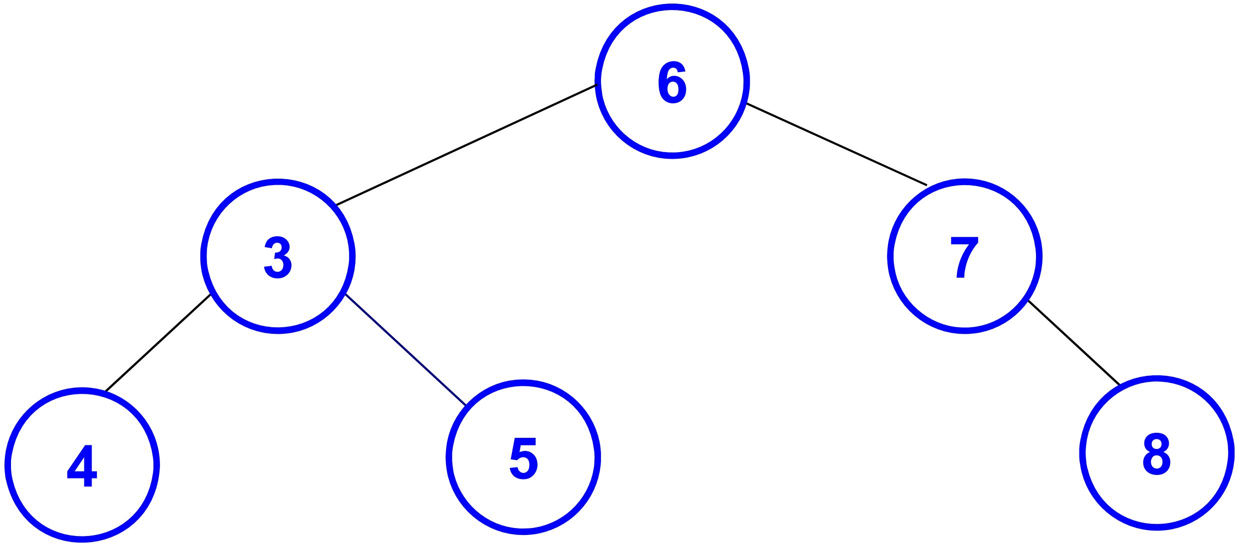
- A binary tree is a rooted tree in which each node has either no children, one child, or two children. --- Binary
  - If a node has one child, that child is designated as either a left child or a right child (but not both).
  - If a node has two children, one child is designated a left child and the other a right child.

✓ When a binary tree is drawn, a left child is drawn to the left and a right child is drawn to the right.

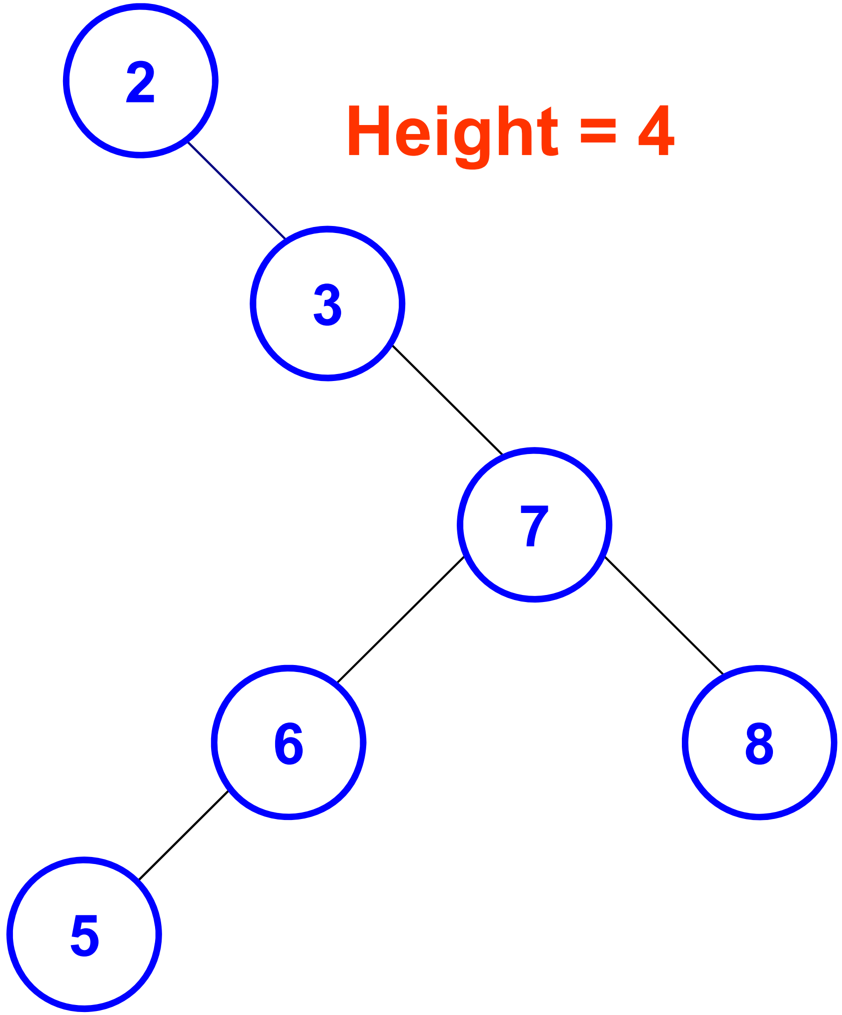
Binary Tree: Examples



Binary Tree: Examples



Height = 2



Height = 4

- ✓ **Height:** **Max number of ancestors**

An empty tree has height –1
A tree with a single node has height 0

Tree: Recursion and recursive

- Recursion as a technique is extensively used

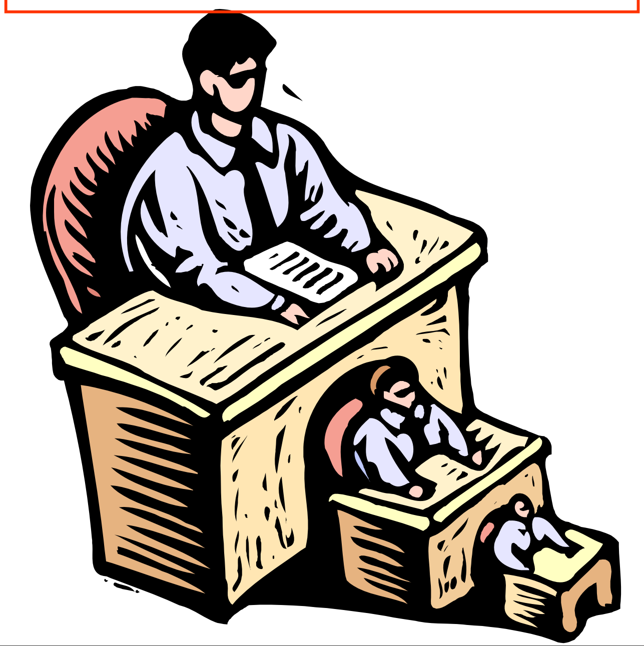
```
factorial( n )=n*(n-1)*…*2*1
```

```
factorial(n){
    k=1
    if n>1 then
        for i=2:n{
            k=k*i
        }
    return k
}
```

- Familiar recursive function

```
factorial( n ){
    if n = 1 then return 1
    else return n * factorial( n-1 )
}
```

factorial(5)	5*factorial(4)	120
factorial(4)	4*factorial(3)	24
factorial(3)	3*factorial(2)	6
factorial(2)	2*factorial(1)	2
factorial(1)	1	



Recursion and recursive

```
factorial( n )=n*(n-1)*…*2*1
```

n=5

```
factorial( n ){
    if n = 1 then return 1
    else return n * factorial( n-1 )
}
```

n=4

```
factorial( n ){
    if n = 1 then return 1
    else return n * factorial( n-1 )
}
```

n=3

```
factorial( n ){
    if n = 1 then return 1
    else return n * factorial( n-1 )
}
```

n=2

```
factorial( n ){
    if n = 1 then return 1
    else return n * factorial( n-1 )
}
```

n=1

```
factorial( n ){
    if n = 1 then return 1
    else return n * factorial( n-1 )
}
```

1 2 3 4 5

6 return 2

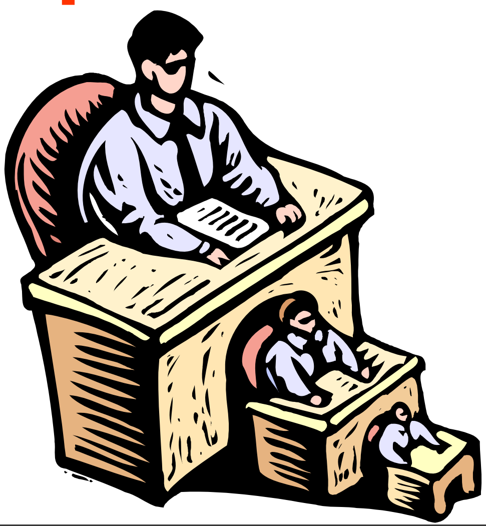
7 return 6

8 return 24

9 return 120

Recursion and recursive

- Recursion is the process that a procedure goes through when one of the steps of the procedure involves re-running the entire same procedure.
  - ☞ A procedure that goes through recursion is said to be recursive
- Simply, a recursive function (definition or algorithm) is one which calls itself as part of the function body (in a smaller scale)



Recursive procedure example

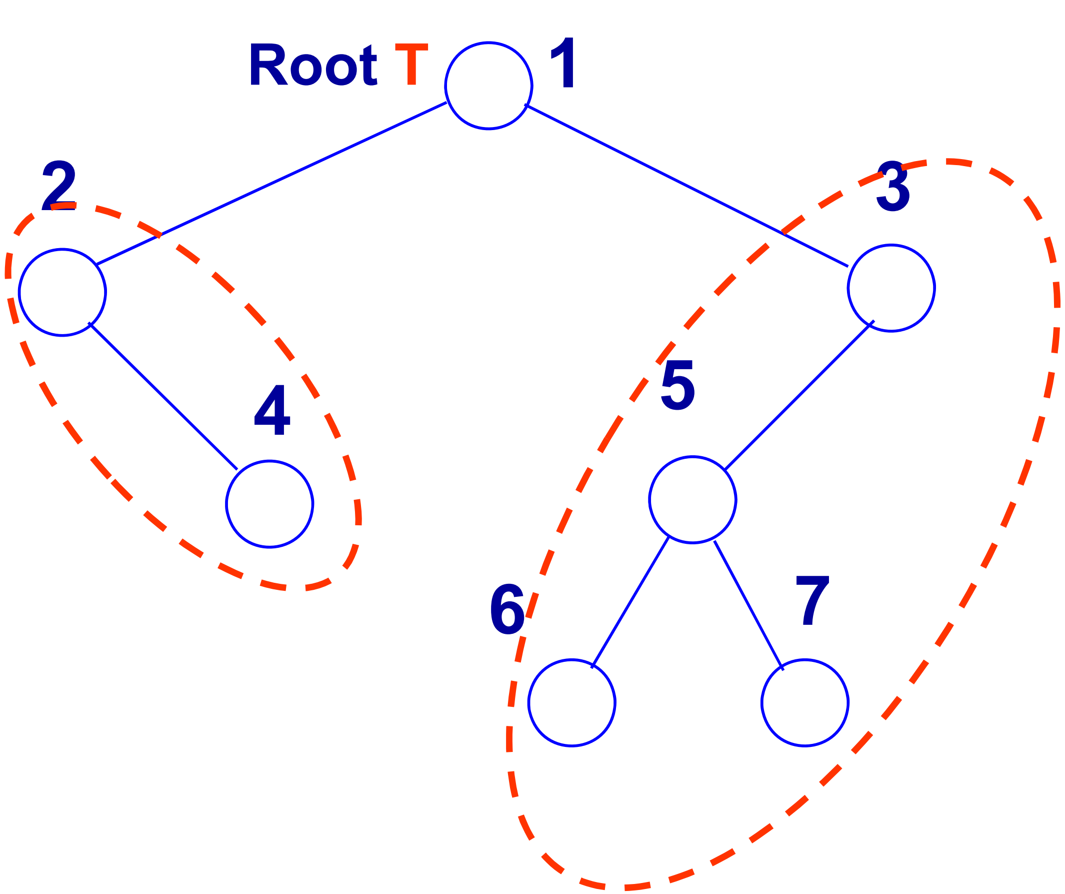
- How do you study a text book?
  - ☞ Reading a book is a recursive procedure

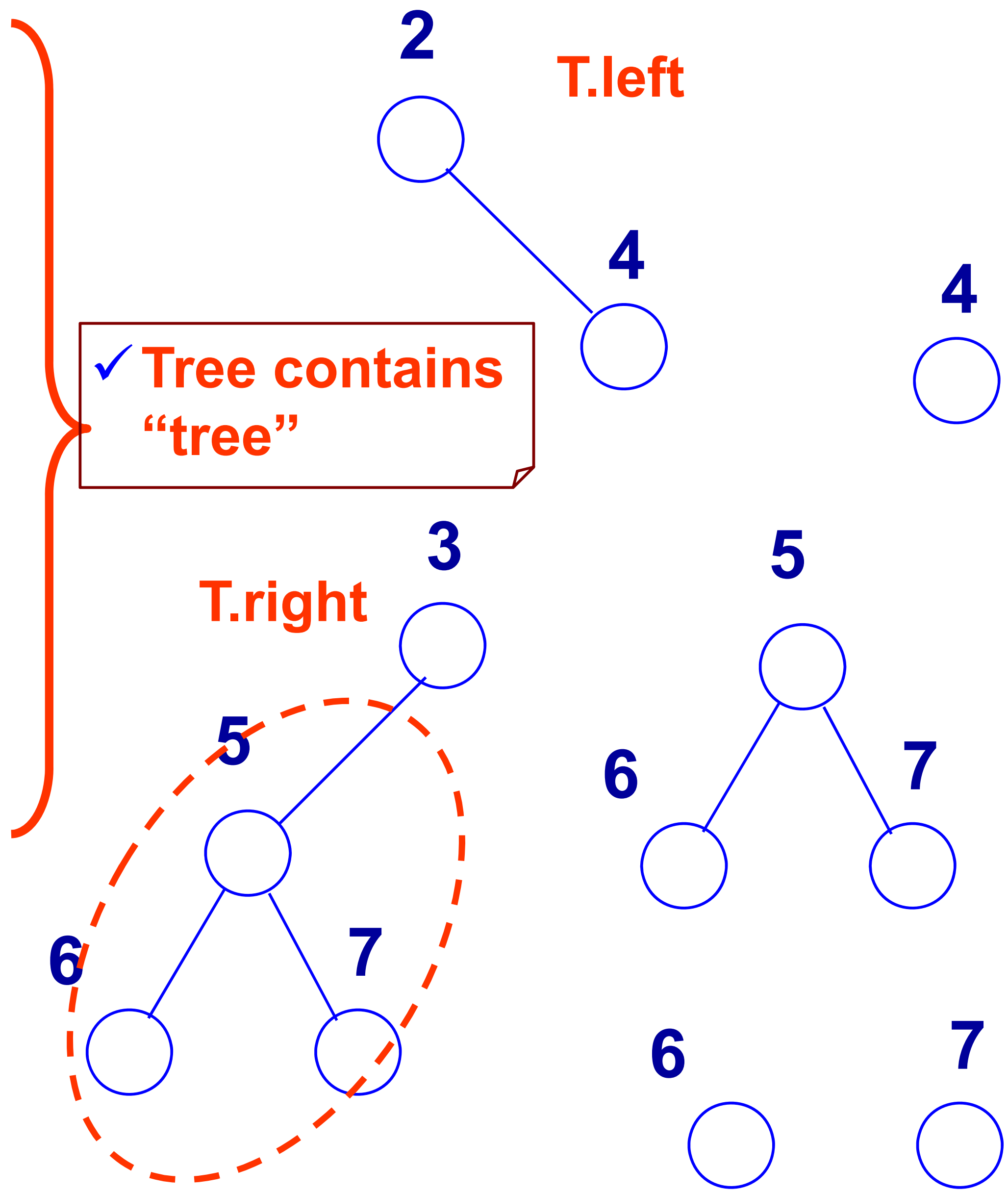
Study the book:

1. If you have reached the end of the book you are done, else
2. Study one page, then study the rest of the book.

Recursive!!

Understanding the *recursive* nature of trees





✓ All parts of a tree are considered as trees

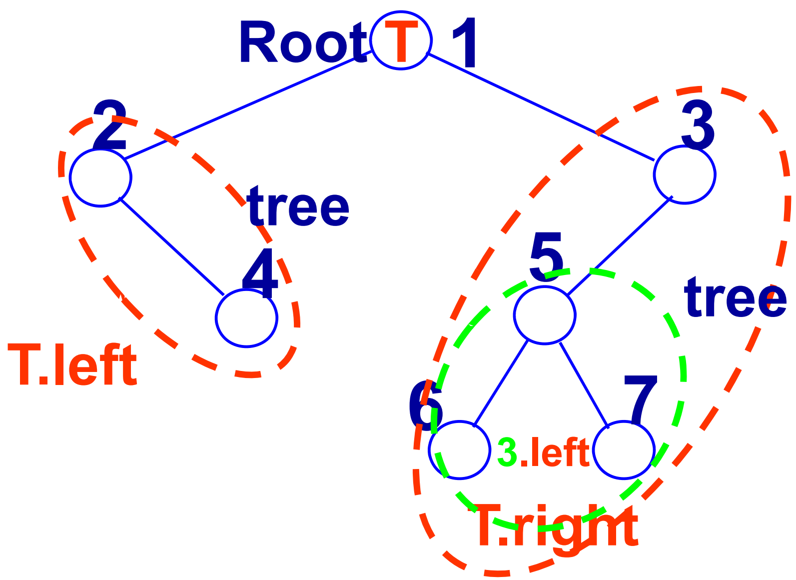
Binary Tree: Definition

- A **binary tree T** is a structure defined on a finite set of nodes that

  Binary Tree:

  1. Either contains no nodes, or
  2. Is composed of **three** disjoint sets of nodes: a root node, its left subtree and its right subtree.
  3. Left and right **subtrees are binary tree** (recursive definition)

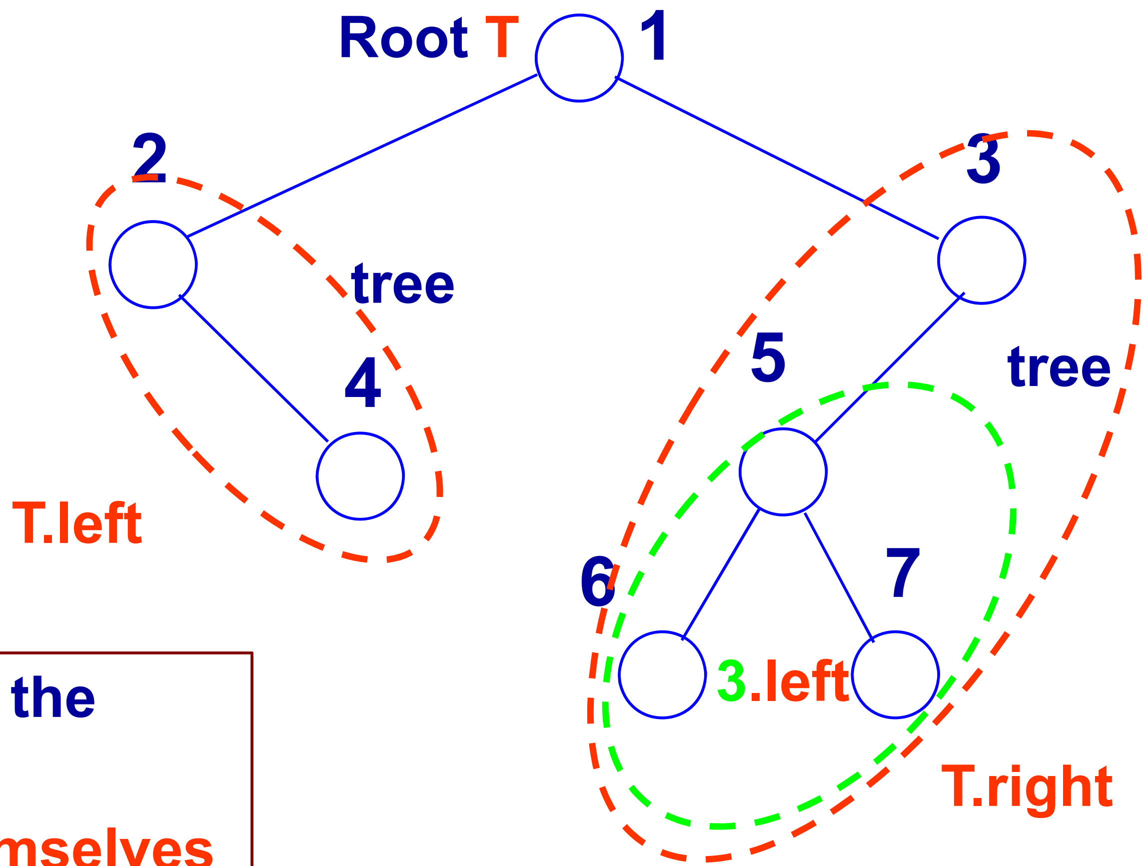
✓The binary tree is defined **recursively**.



The recursive nature of trees

- **Trees are considered a recursive data structure because trees are said to contain themselves**

- ✓ Referencing subtree: dot notation:
  - ✓T.left
  - ✓T.right



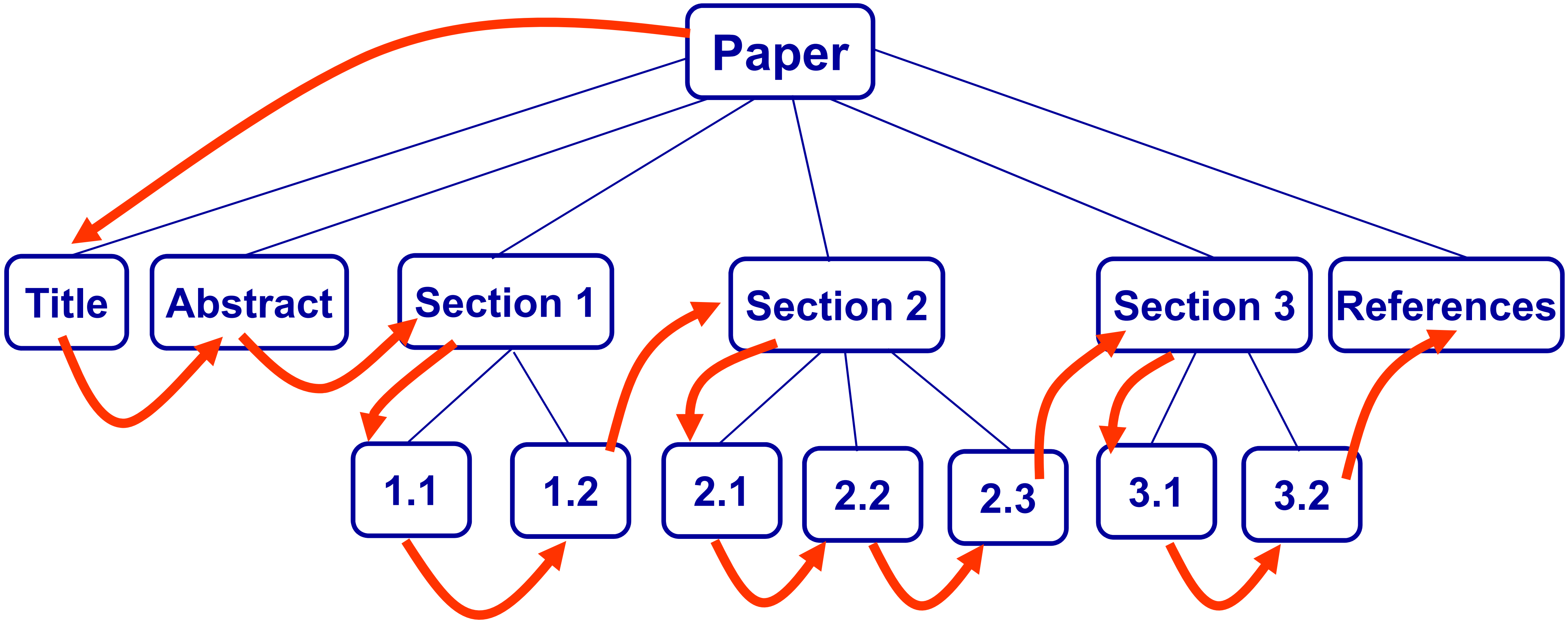
- ✓ Here is a tree that demonstrates the recursiveness of trees;
- ✓ nodes 2, 3 are roots of trees themselves

Binary Tree Traversals

- To traverse a binary tree is to visit each node in the tree in some ***prescribed order***.
  - ☞ Depending on the application, "visit" may be interpreted in various ways.
  - ☞ For example, if we want to print the data in each node in a binary tree, we would interpret "visit" as "print the data."
- The three most common traversal orders are ***preorder, inorder***, and ***postorder***.
  - ☞ Each is most easily defined recursively.

Preorder traversal: Example

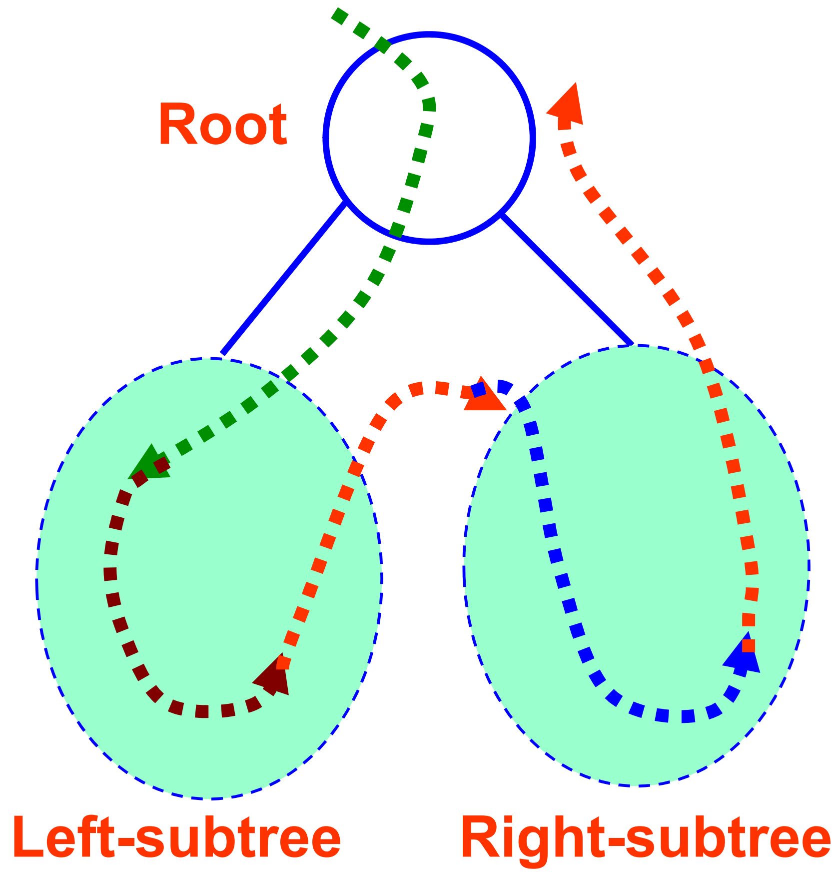
- If you read an entire paper sequentially … …



Preorder traversal: Definition

- ***Preorder traversal*** **of a binary tree rooted at *root* is defined by the rules:**

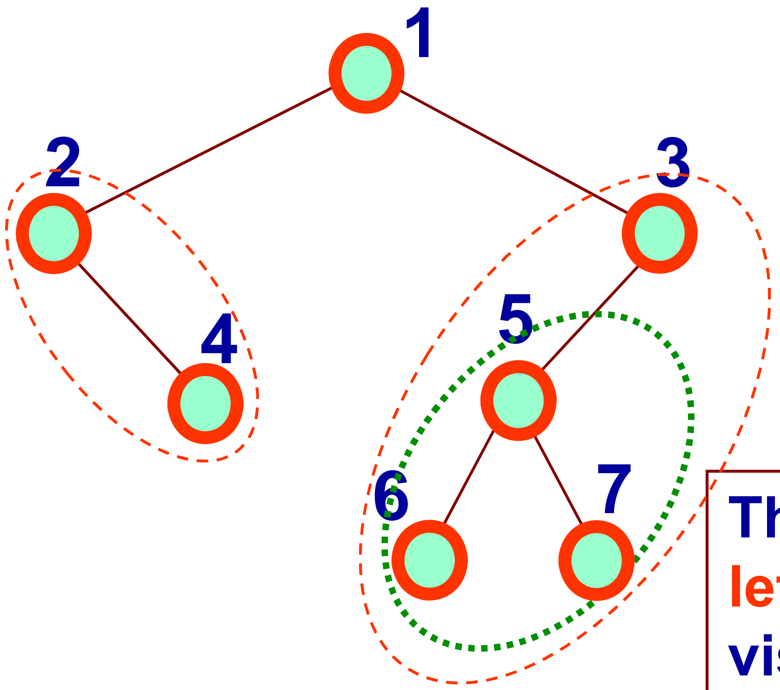
1. **If *root* is empty, stop.**
2. **Visit *root.***
3. **Execute *preorder* on the binary tree rooted at the left child of the root.**
4. **Execute *preorder* on the binary tree rooted at the right child of the root.**



✓ **In a preorder traversal, a node is visited *before* its descendants**

✓ **In short, root—left— right**

Preorder traversal: Example



Beginning with *root* = *1*, we visit 1

We then execute preorder on the subtree rooted at 2 (i.e., the subtree rooted at 1's left child)

The order of visitation of this subtree is 2 (root), (no left subtree), 4 (right subtree). The overall order of visitation so far is 1, 2, 4.

Since we have finished executing preorder on 1's left subtree, we next execute preorder on 1's right subtree

The order of visitation for this subtree is 3, 5, 6, 7.

Therefore, the order of visitation for the tree
1, 2,4, 3, 5, 6, 7.

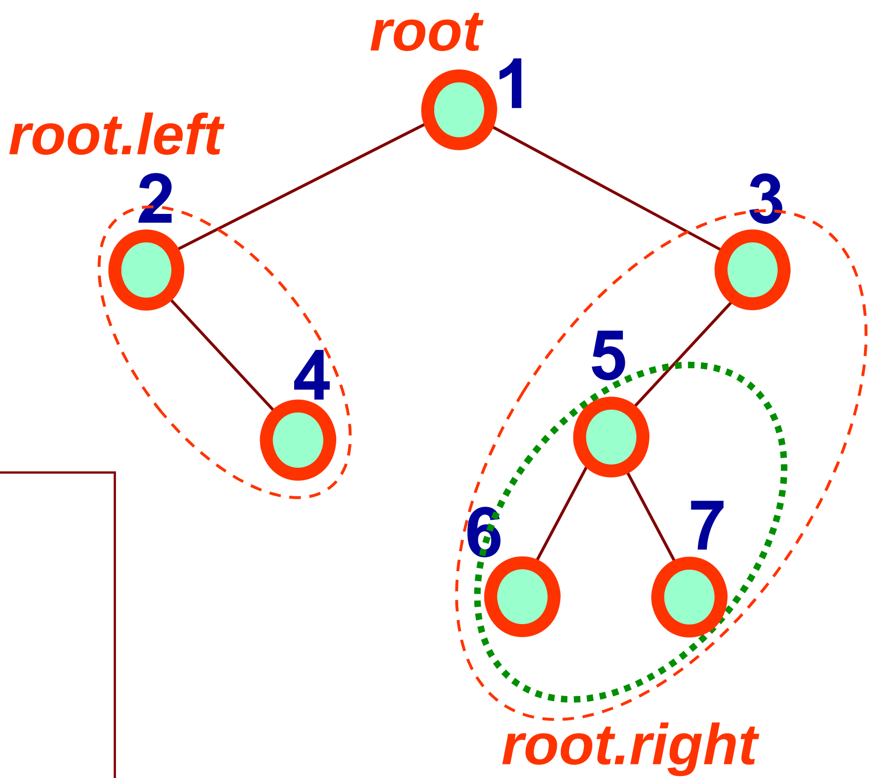
Algorithm Preorder

- This algorithm performs a preorder traversal of the binary tree with root *root*.

1. If *root* is empty, stop.
2. Visit *root.*
3. Execute *preorder* on the binary tree rooted at the left child of the root.
4. Execute *preorder* on the binary tree rooted at the right child of the root.

pseudo code

```
preorder(root)    {
   if (root != null) {
         visit root;
         preorder(root.left);
         preorder(root.right);
   }
}
```



Counting Nodes in a Binary Tree

- **As an application of the preorder algorithm, we adopt it to obtain an algorithm that counts the nodes in a binary tree.**
  - ☞ **"Visit node" is interpreted as "count node."**

Algorithm: Counting Nodes in a Binary Tree

- This algorithm returns the number of nodes in the binary tree with root *root*

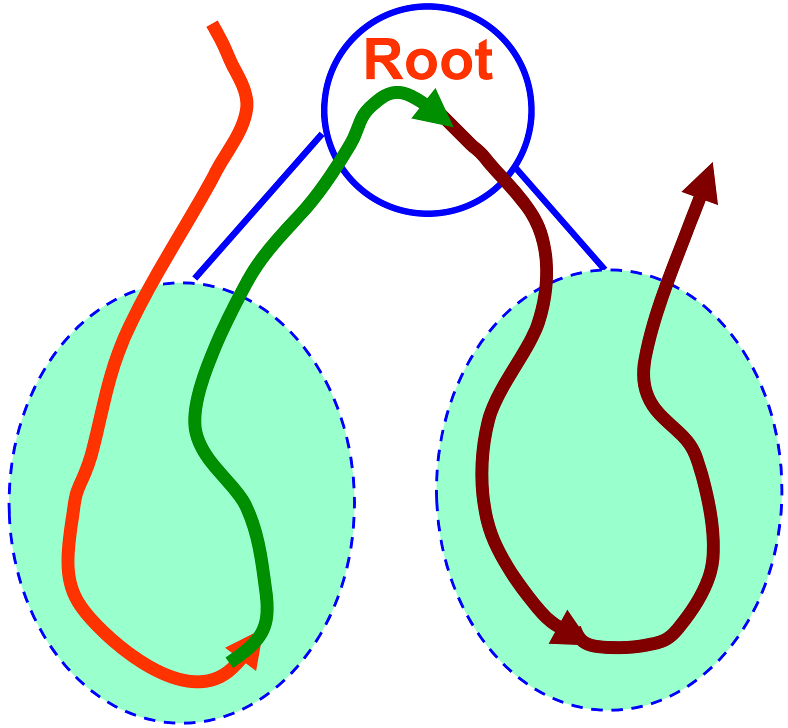
```
Count_nodes (root) {
    if (root== null)
        return 0
    count = 1    // count root
    // add in nodes in left subtree
    count = count + count_nodes(root.left)
    // add in nodes in right subtree
    count = count + count_nodes(root.right)
    return count
}
```

```
preorder(root){
  if (root != null) {
    visit root;
     preorder(root.left);
     preorder(root.right);
  }
}
```

Inorder Traversal: Definition

- **Inorder traversal of a binary tree rooted at *root* is defined by the rules:**

1. **If *root* is empty, stop.**
2. **Execute *inorder* on the binary tree rooted at the left child of the root.**
3. **Visit *root.***
4. **Execute *inorder* on the binary tree rooted at the right child of the root.**

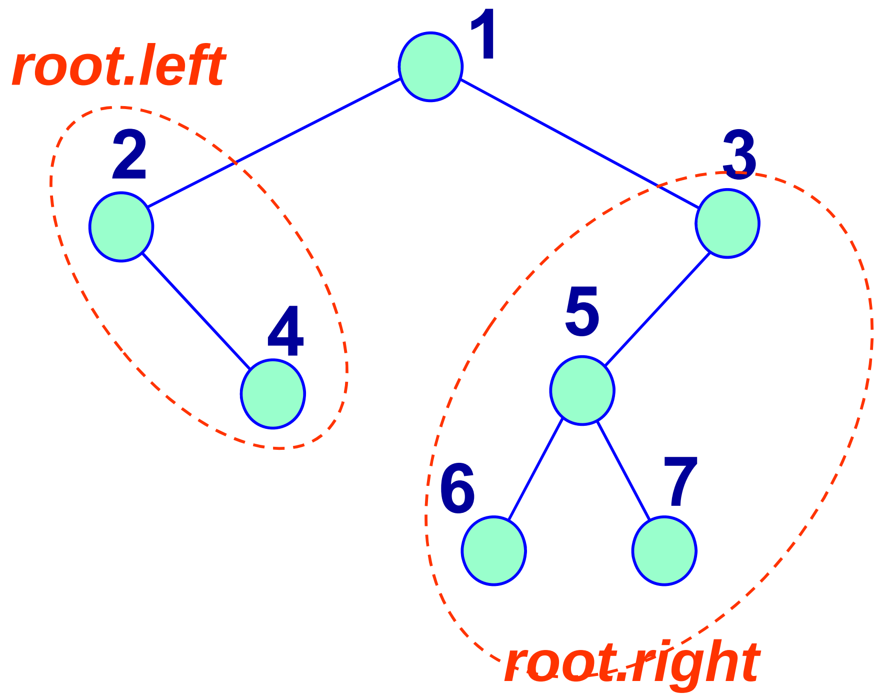


✓ **In short, left—root— right**

Algorithm Inorder

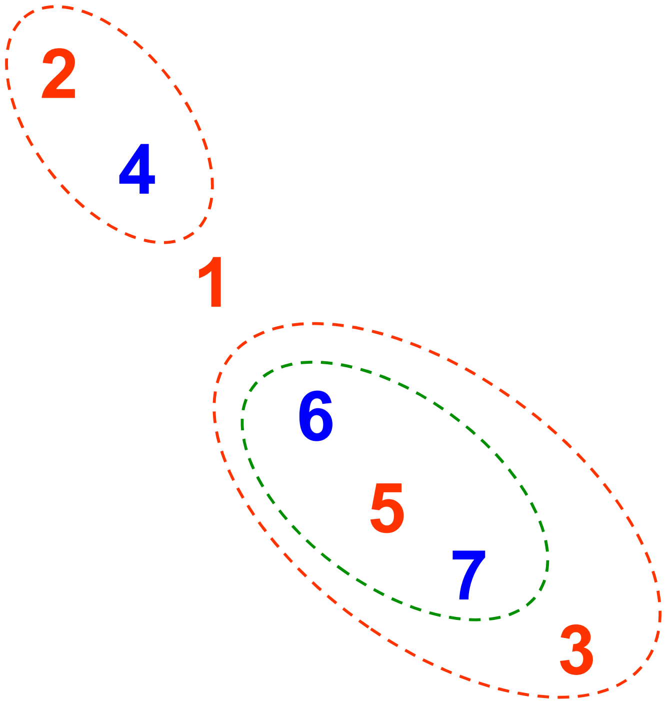
- This algorithm performs an inorder traversal of the binary tree with root *root*.

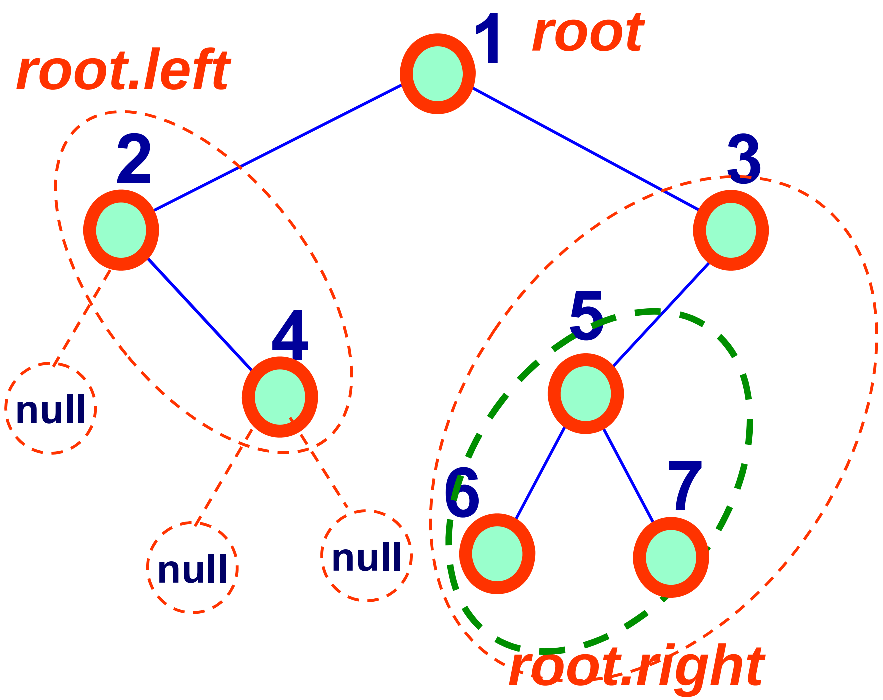
```
inorder(root)    {
    if (root != null) {
        inorder(root.left);
        visit root;
        inorder(root.right);
    }
}
```



Inorder Traversal: Example

- Inorder visits the nodes of the binary tree in the order



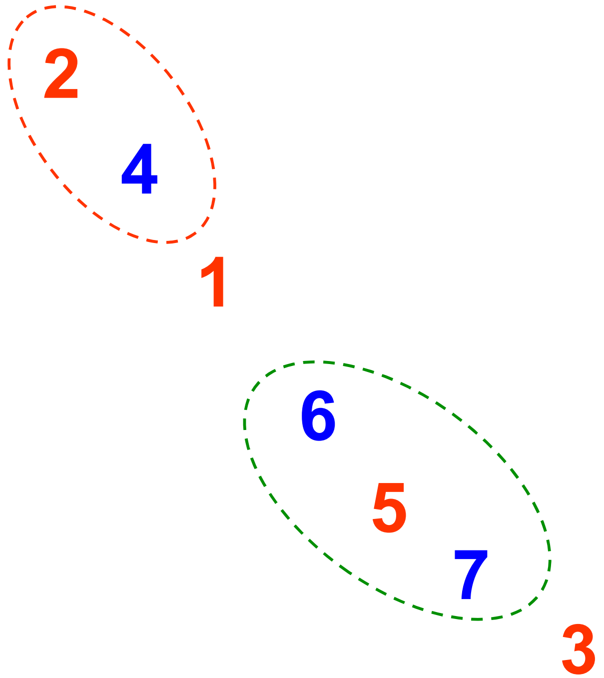


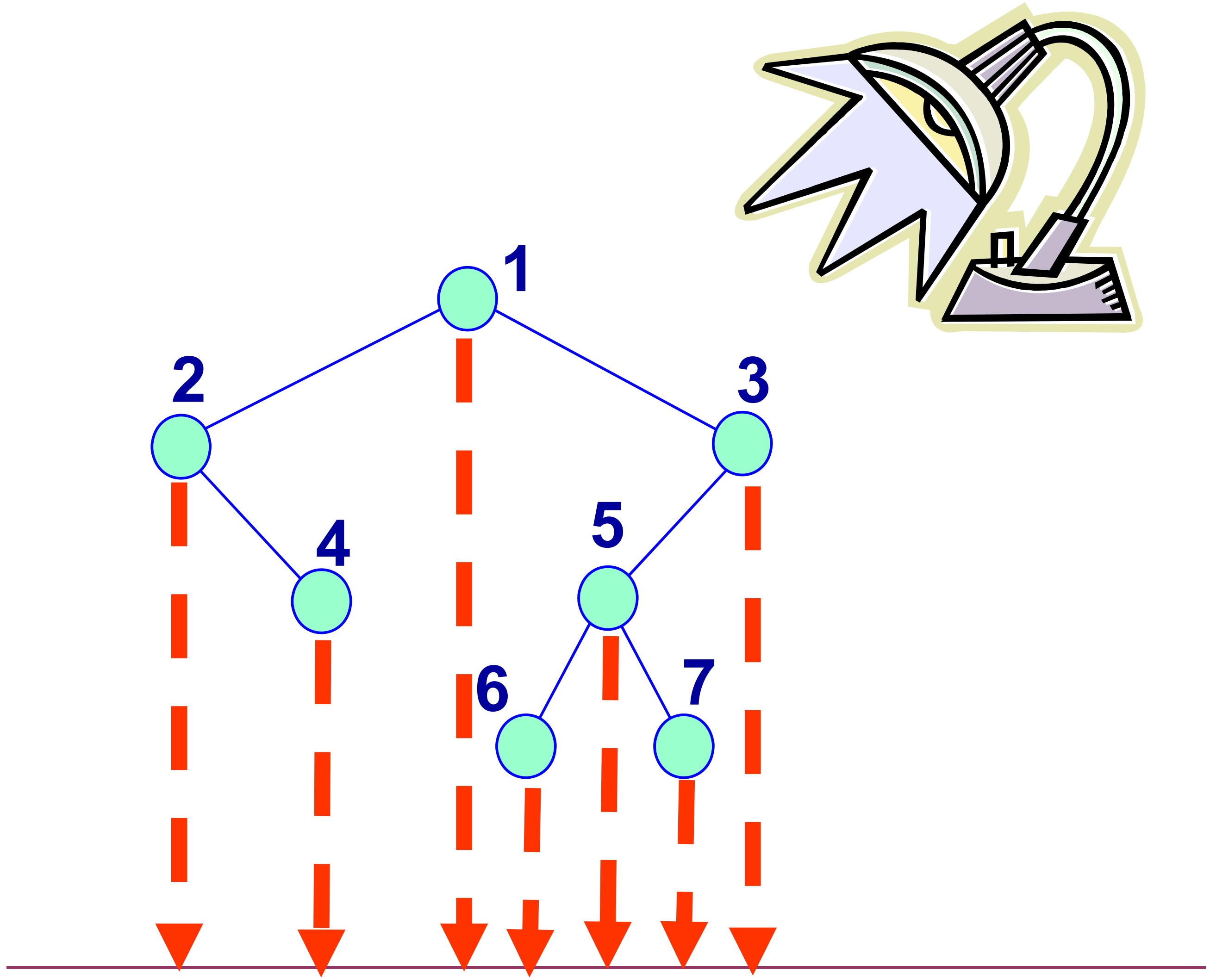

```
inorder(root)    {
    if (root != null) {
        inorder(root.left);
        visit root;
        inorder(root.right);
    }
}
```

✓ In order of left—root— right

Inorder Traversal By Projection (Squishing)

- **An easy way …**



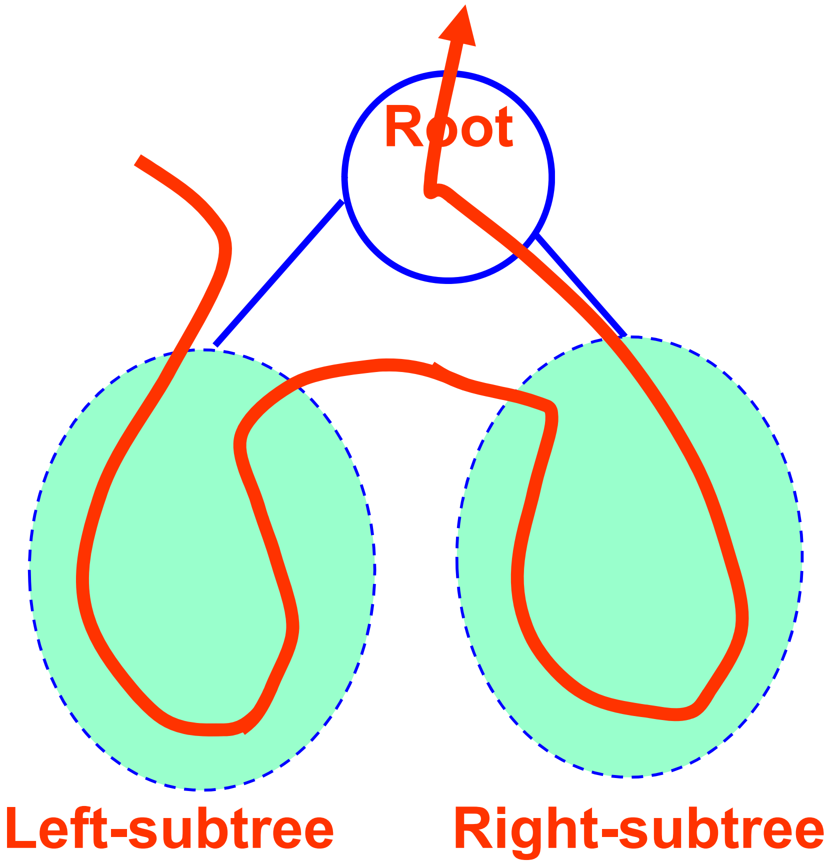


Inorder traversal: 2 4 1 6 5 7 3

Postorder traversal: Definition

- ***Postorder traversal*** **of a binary tree rooted at *root* is defined by the rules**

1. **If *root* is empty, stop.**
2. **Execute *postorder* on the binary tree rooted at the left child of the root.**
3. **Execute *postorder* on the binary tree rooted at the right child of the root.**
4. **Visit *root*.**

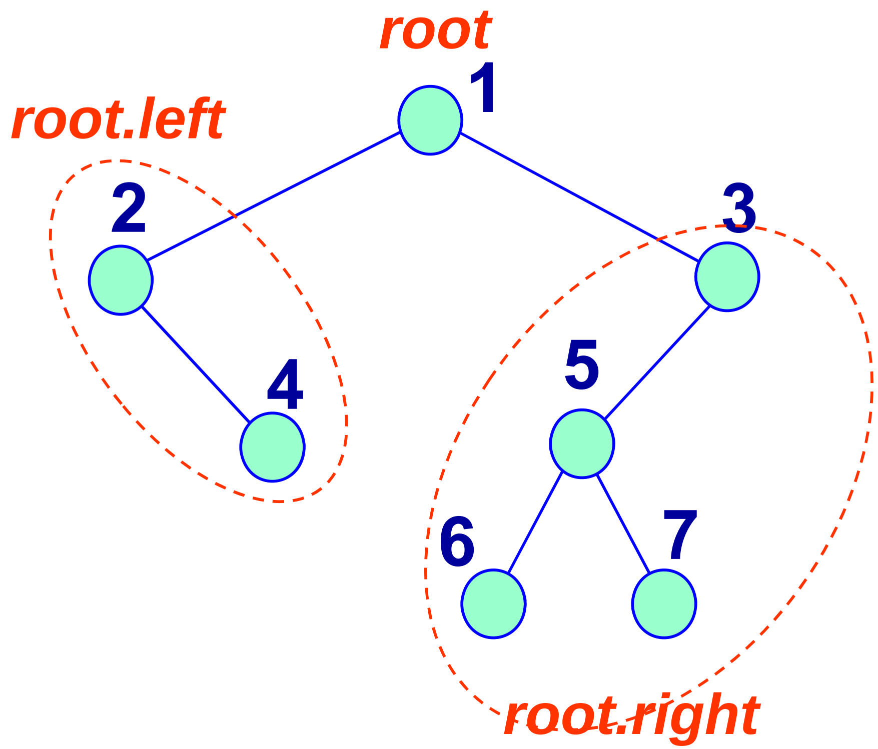


✓ **In short, left—right—root**

Algorithm Postorder

- This algorithm performs a **postorder** traversal of the binary tree with root *root*.

```
postorder(root)    {
    if (root != null) {
        postorder(root.left);
        postorder(root.right);
        visit root;
    }
}
```

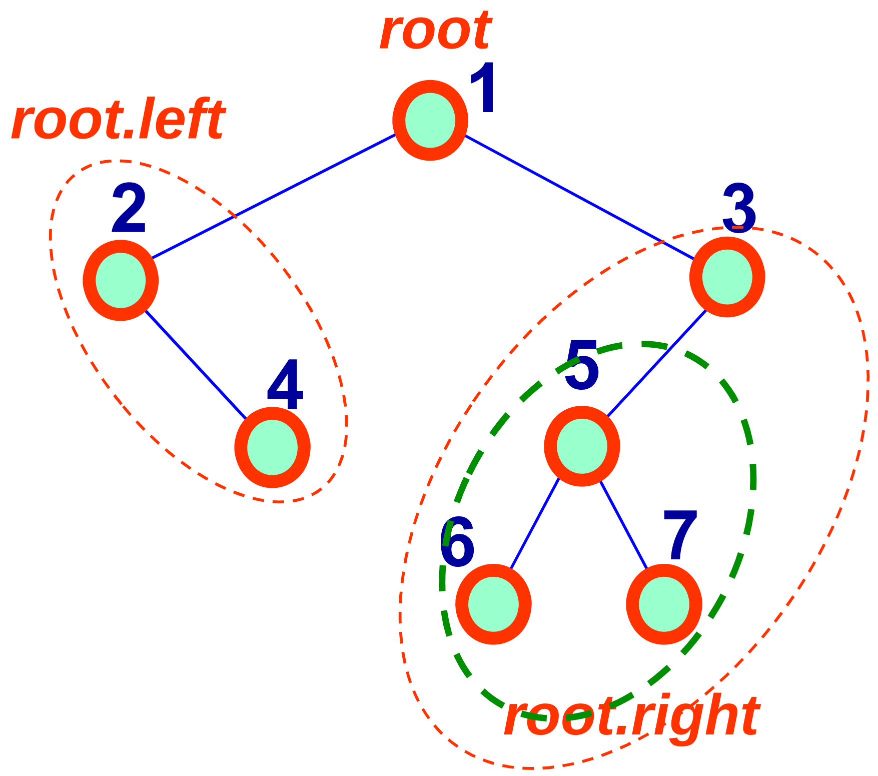


✓ In short, left—right—root

Postorder Traversal: Example

- **Postorder** visits the nodes of the binary tree in the order

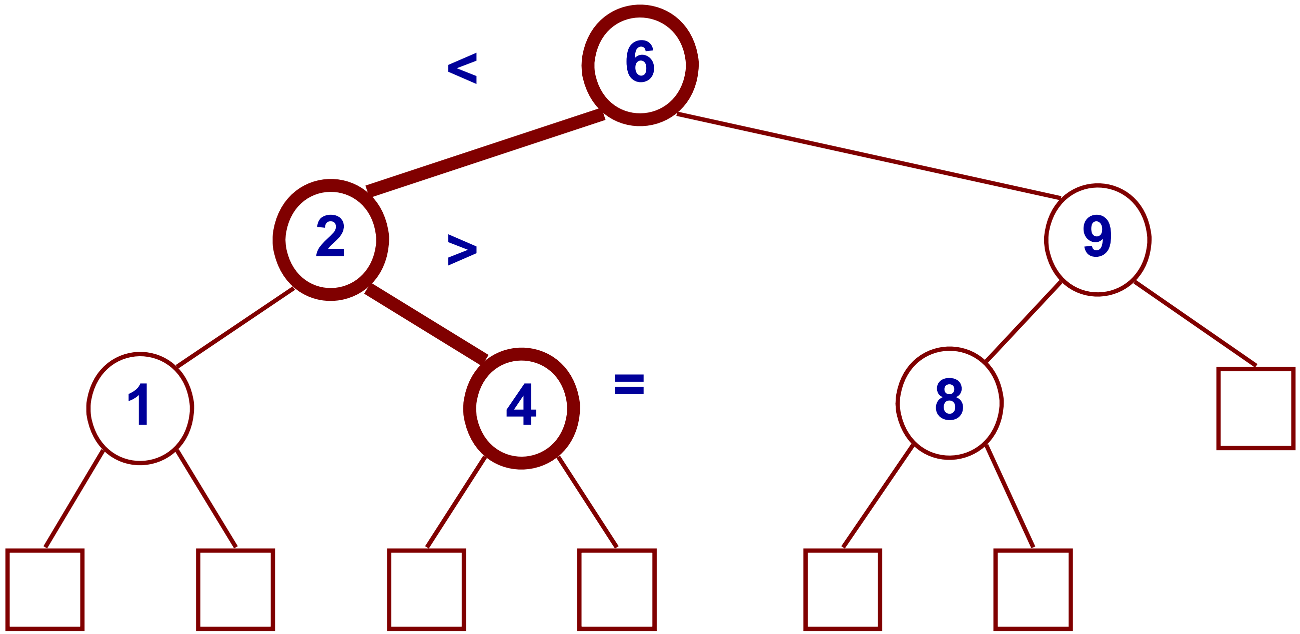
4 2 6 7 5 3 1




```
postorder(root)    {
    if (root != null) {
        postorder(root.left);
        postorder(root.right);
        visit root;
    }
}
```

✓ In short, left—right—root

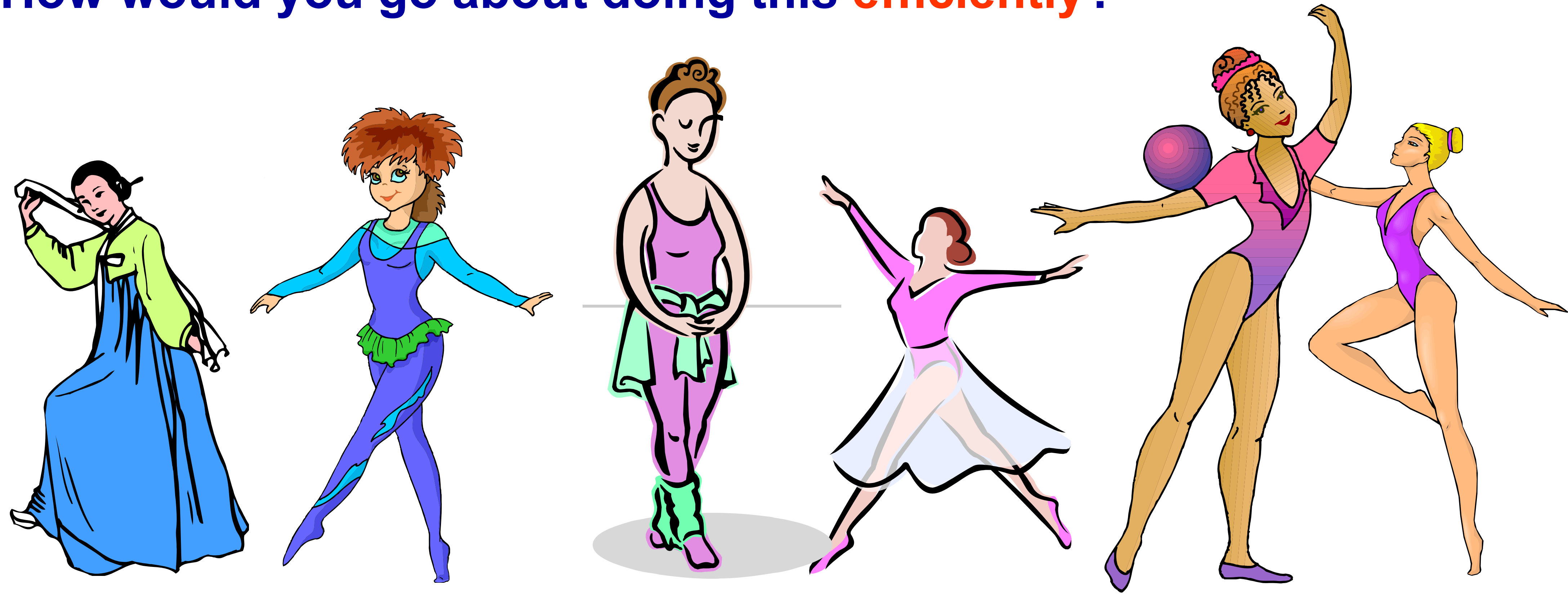
Binary Search Trees



What is a Binary Search Tree

Suppose that you have to sort a group of people by height so that you can easily search for someone by their height later on.

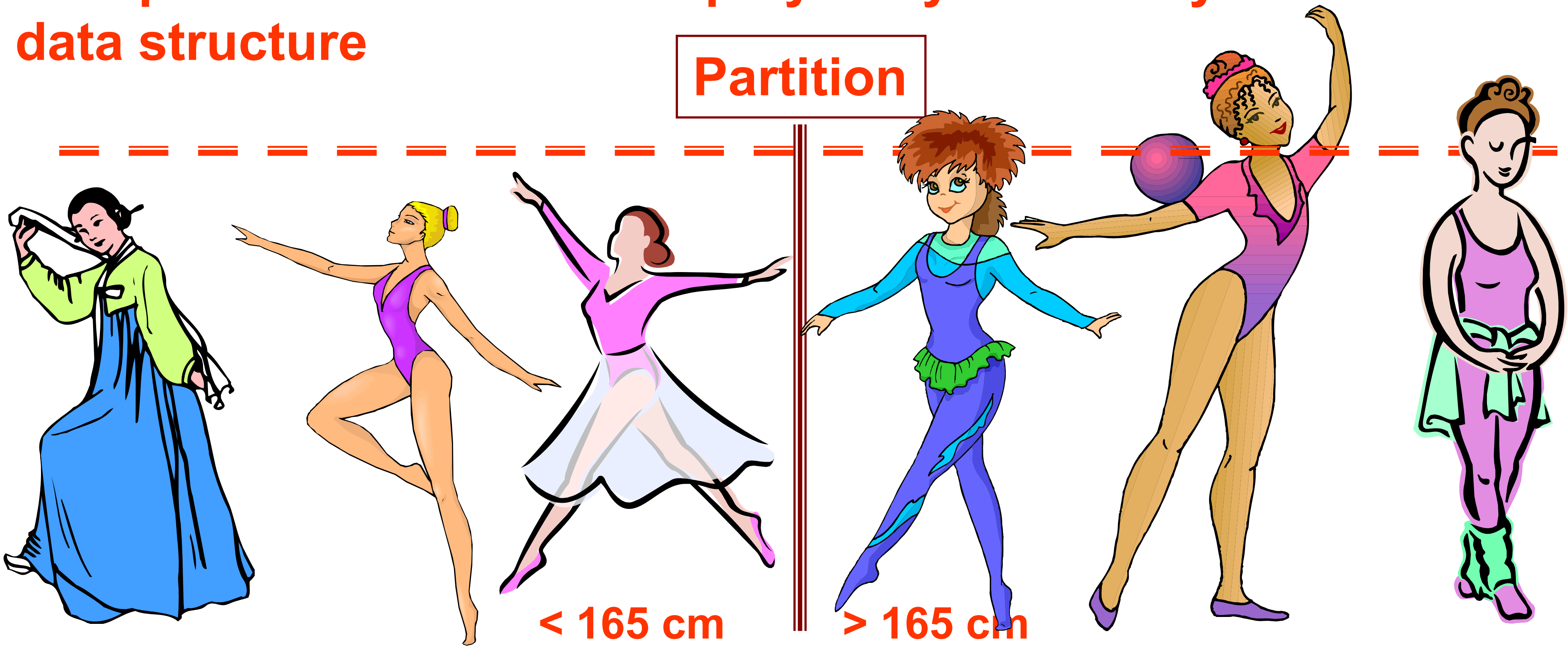
How would you go about doing this efficiently?



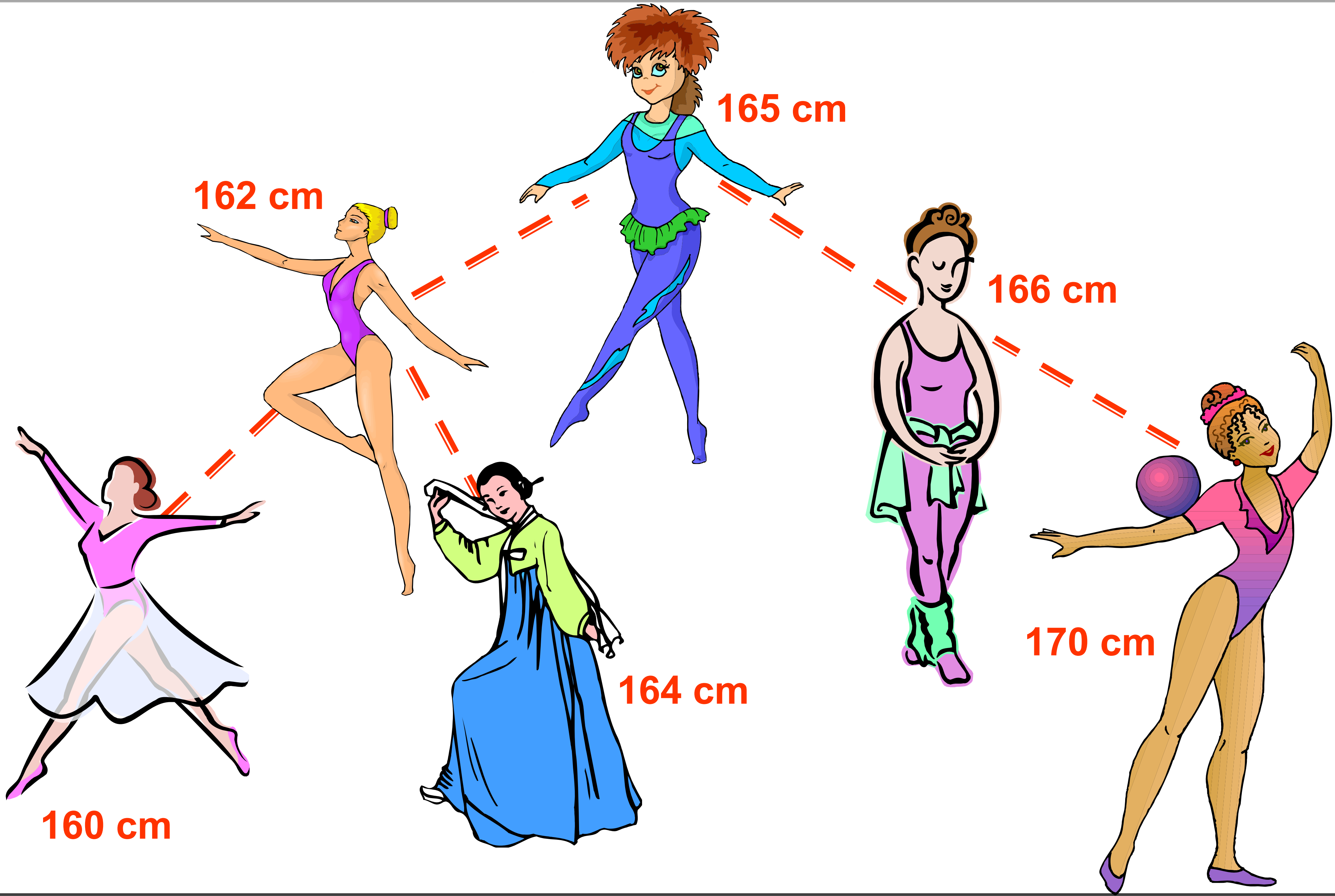
What is a Binary Search Tree

Pick a midpoint (say, 165cm) and look at the first person in line. If s/he is below that height, you move him/her to the left, otherwise to the right.

This partition method is employed by the binary search tree data structure



What is a Binary Search Tree



What is a Binary Search Tree

- In a binary search tree, data are stored in the nodes.
- Data items can be compared; that is, <, ≤, and so on, are defined on the data
  - ☞ Compare data of a node with its parent

What is a Binary Search Tree

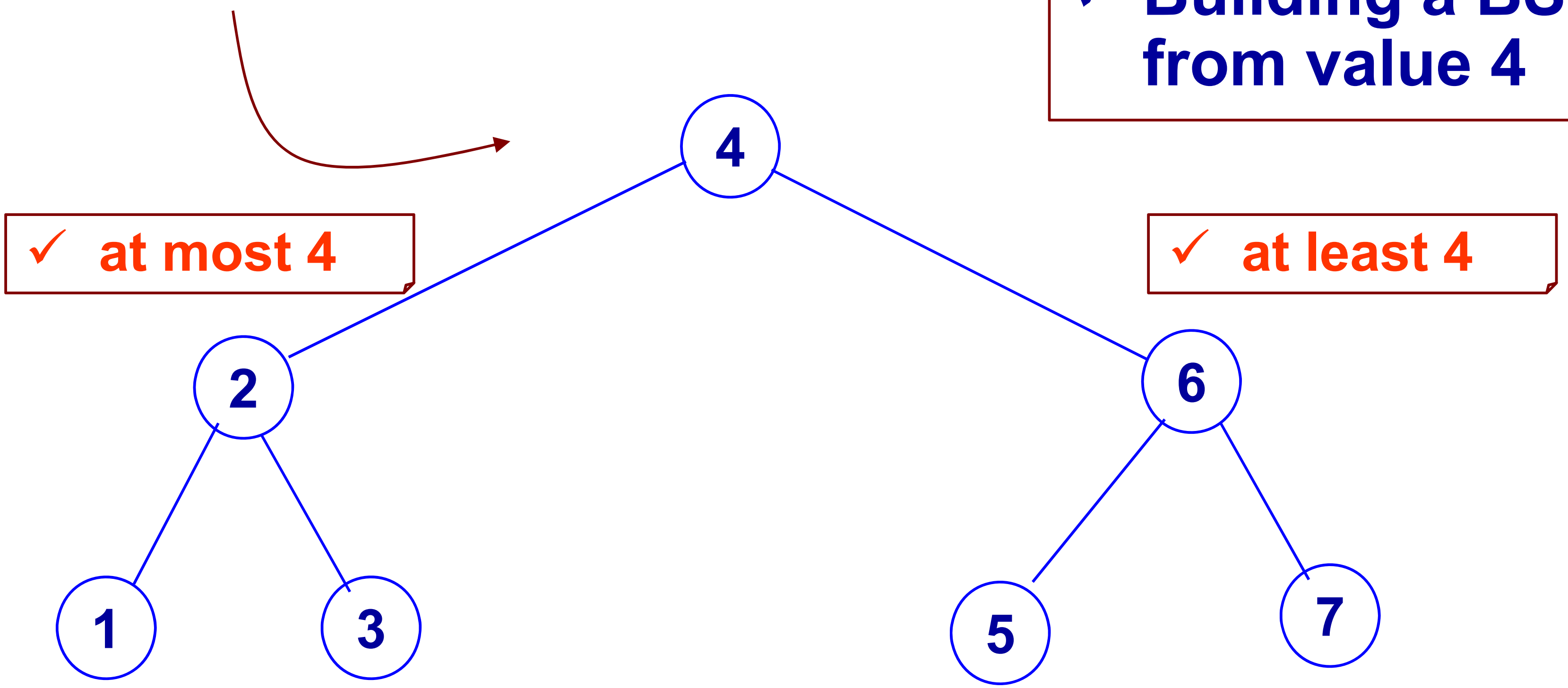
- The data are placed in a binary search tree so that the following "binary search tree property" is satisfied:
  - for every node *v*, each data item in v's left subtree, if any, is less than or equal to the data item in *v*, and each data item in v's right subtree, if any, is greater than the data item in *v*.

✓ That is, the value of left child is at most the value of its parent, and a right child is at least its parent.

Binary search trees: Example

4, 2, 6, 5, 1, 3, 7

✓ Building a BST, starting from value 4



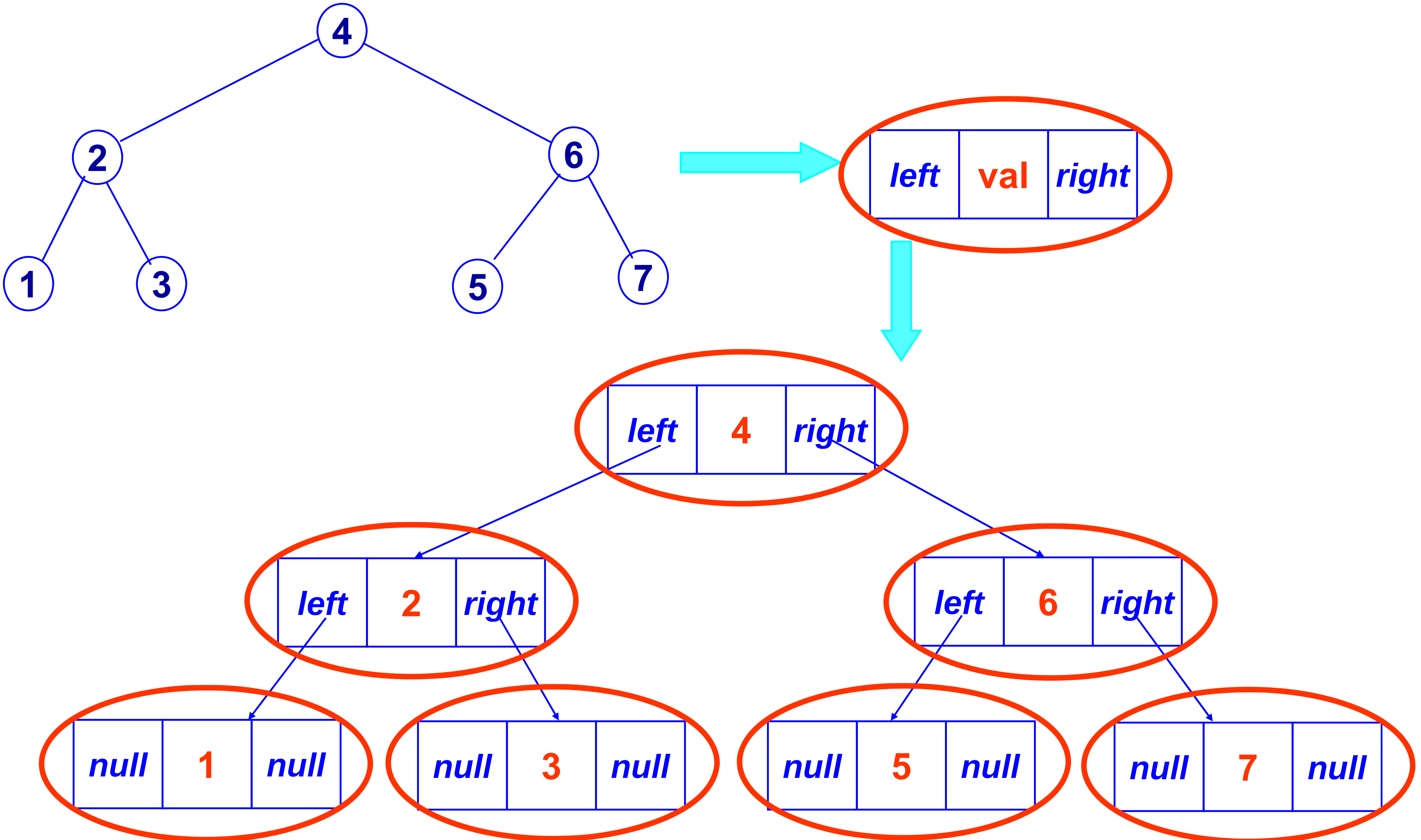
✓ **"Binary search tree property"**: a node's left child must have a data less than or equal to **its parent**, and a node's right child must have a data greater than **its parent**

Insertion and deletion

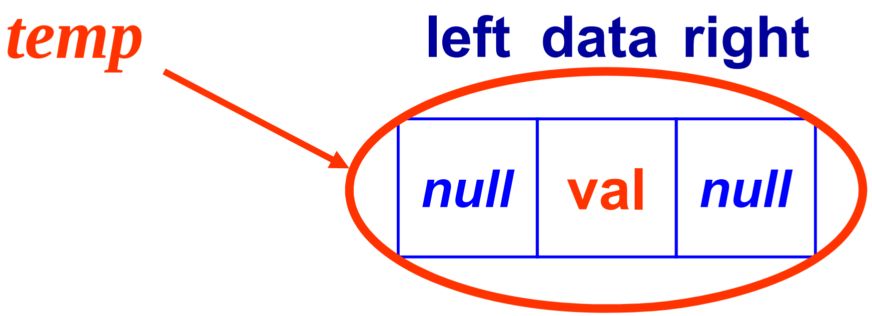
- **The operations of insertion and deletion cause a binary search tree to change**
- **This change is done in such a way that the binary-search-tree property continues to hold.**

BST Insertion

Node Representation



Set up a node to be inserted to tree

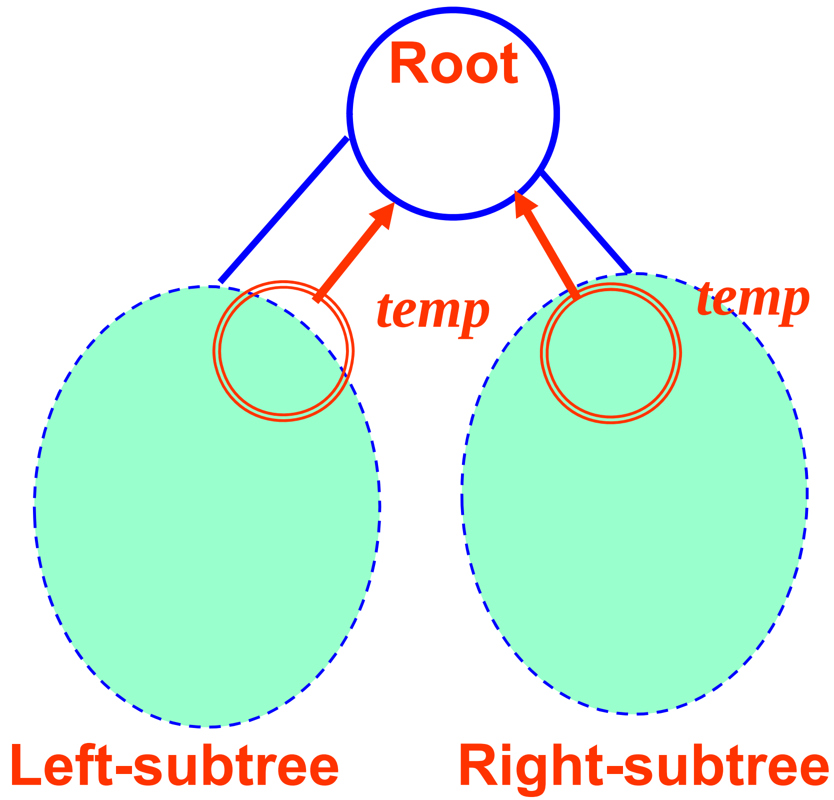



```
// set up node that contains value val to be added to tree

temp = new node;
temp.data = val
temp.left = temp.right = null
```

BST Insertion

❑ *BSTinsert:*

1. If *root* is empty, return *temp* as the root of the BST, stop.
2. Execute *BSTinsert_recurs* procedure on tree root
   - 2.1 if *temp.data<=root.data* and left child of root is empty, add *temp* as the left child of root
   - 2.2 if *temp.data>root.data* and right child of root is empty, add temp as the right child of root
   - 2.3 if *temp.data<=root.data* and left child of root is not empty, apply *BSTinsert_recurs* in its left subtree.
   - 2.4 if *temp.data>root.data* and right child of root is not empty, apply *BSTinsert_recurs* in its right subtree.



5 Insertion Cases

Case 1: ***Insert a node to an empty tree***

Case 2: ***Insert to a non-empty tree, left subtree empty***

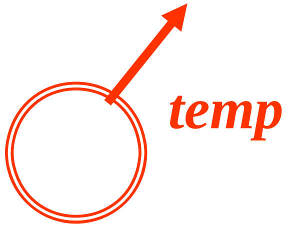
Case 3: ***Insert to a non-empty tree, right subtree empty***

Case 4: ***Insert to a non-empty tree, left subtree non-empty***

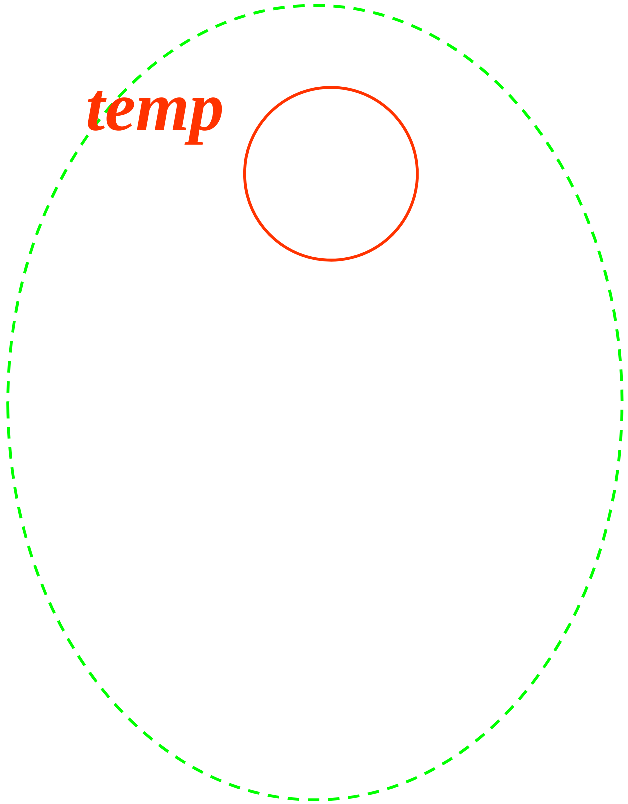
Case 5: ***Insert to a non-empty tree, right subtree non-empty***

Case 1: Insert a node to an empty tree

- ✓ Since the tree is empty, this newly inserted node becomes the root



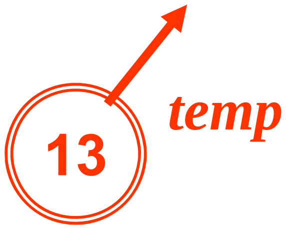

```
BSTinsert (root, val) {
   // set up node to be added to tree
   temp = new node;
   temp.data = val
   temp.left = temp.right = null

   // special case: empty tree
   if (root = = null)
      return temp;
   else  // for all other cases
     BSTinsert_recurs (root, temp);
   return root;
}
```



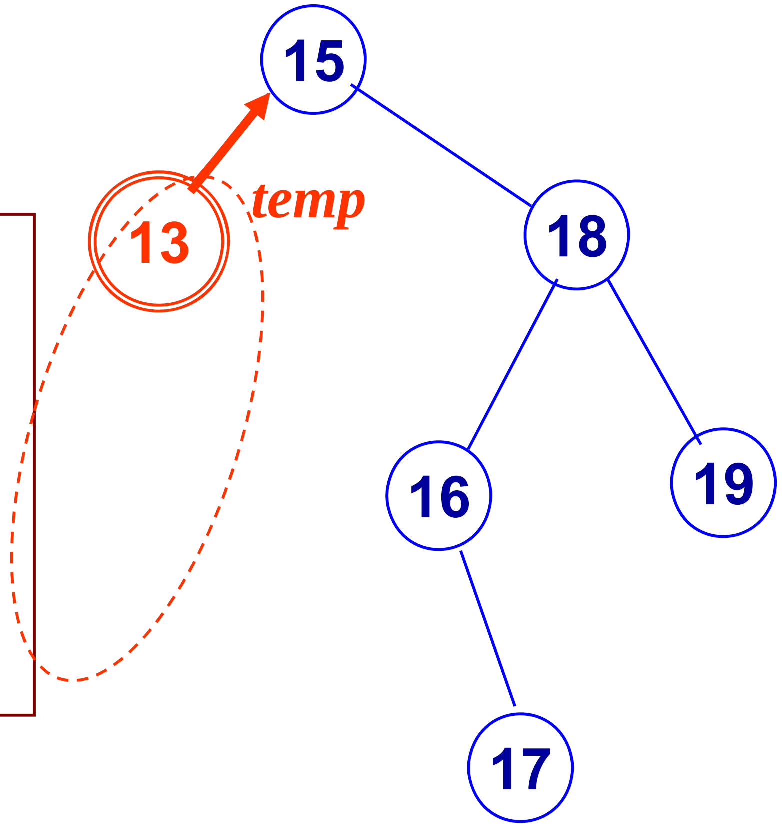
Case 1: *Insert a node to an empty tree*
Case 2: *Insert to a non-empty tree, left subtree empty*
Case 3: *Insert to a non-empty tree, righ subtree empty*
Case 4: *Insert to a non-empty tree, left subtree non-empty*
Case 5: *Insert to a non-empty tree, right subtree non-empty*

Case 2: Insert to a non-empty tree (into left subtree)

1) Inserted data is less than root and
2) left subtree is empty



13 < 15



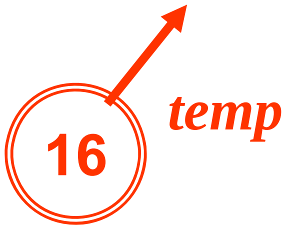

```
BSTinsert_recurs (root, temp) {

  if (temp.data ≤ root.data) {
    if (root.left = = null)
      root.left = temp;
  }

}
```

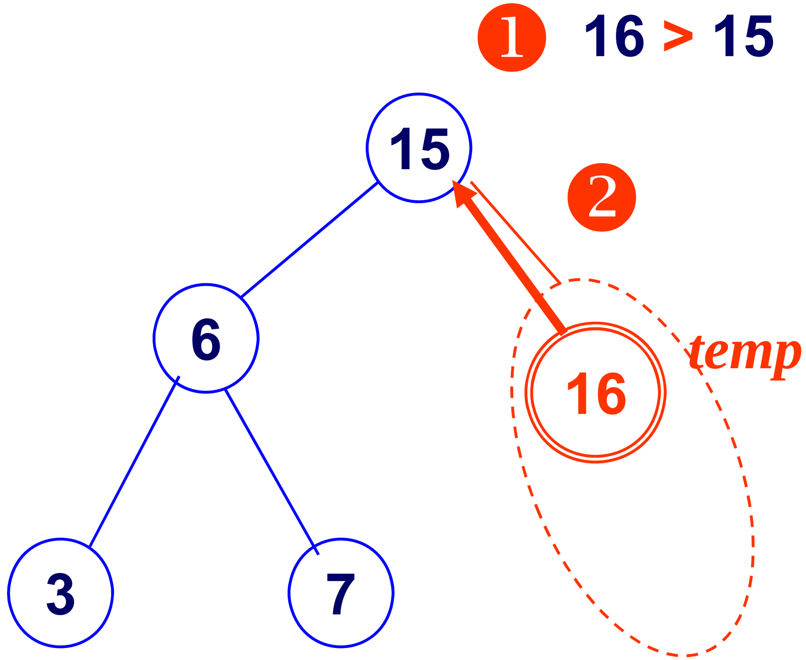
Case 1: *Insert a node to an empty tree*
Case 2: *Insert to a non-empty tree, left subtree empty*
Case 3: *Insert to a non-empty tree, righ subtree empty*
Case 4: *Insert to a non-empty tree, left subtree non-empty*
Case 5: *Insert to a non-empty tree, right subtree non-empty*

Case 3: Insert to a non-empty tree (into right subtree)

1) **Inserted data is larger than root and**
2) **right subtree is empty**

> ✓ **Similar to the case for left subtree**



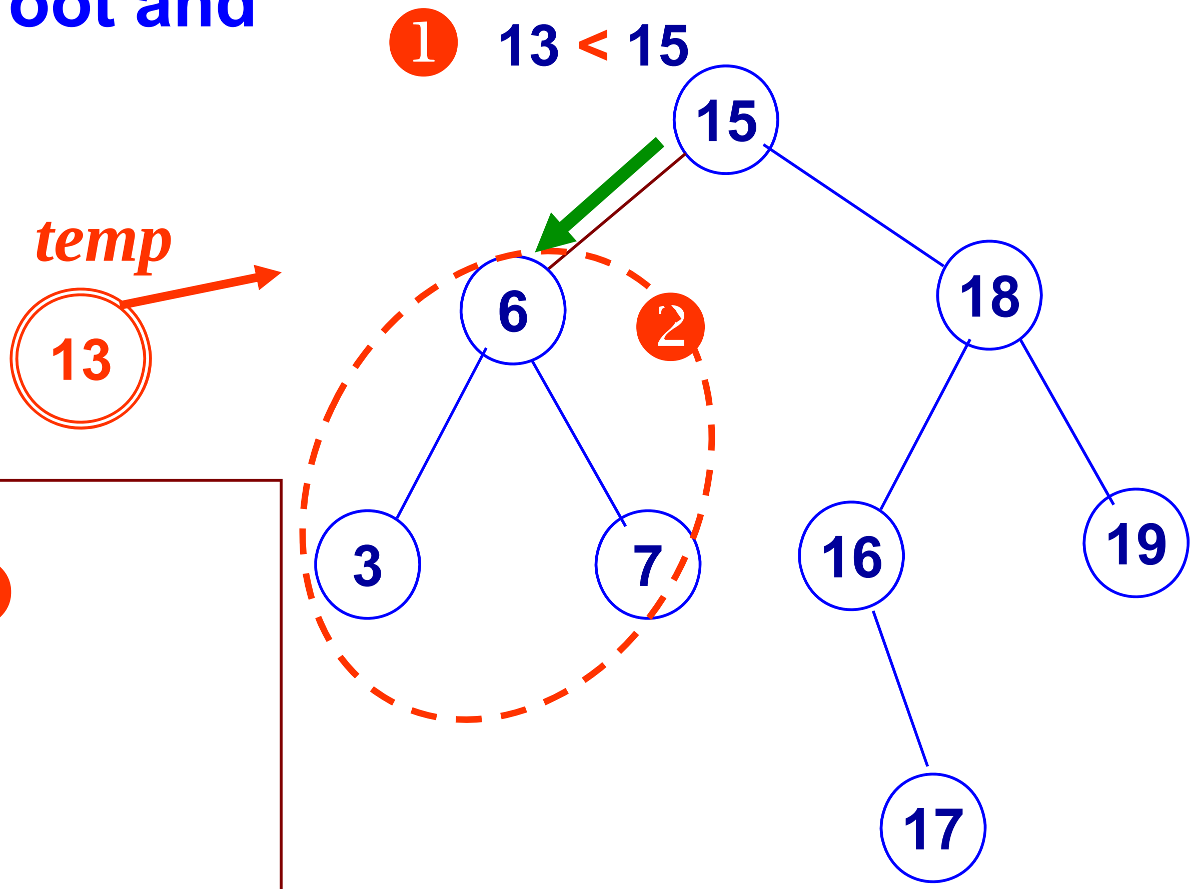



```
BSTinsert_recurs (root, temp) {
  if (temp.data ≤ root.data) {   ❶
    // insert at left
  }
  else {    // goes to right

    if (root.right = = null)    ❷
      root.right = temp;    ❸

  }
}
```

Case 1: *Insert a node to an empty tree*
Case 2: *Insert to a non-empty tree, left subtree empty*
Case 3: *Insert to a non-empty tree, righ subtree empty*
Case 4: *Insert to a non-empty tree, left subtree non-empty*
Case 5: *Insert to a non-empty tree, right subtree non-empty*

Case 4: Insert to a non-empty tree, left subtree non-empty

1) Inserted data is less than root and
2) left subtree is non-empty

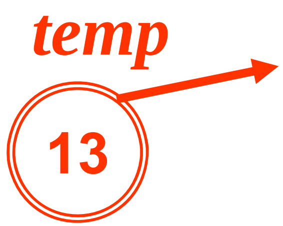


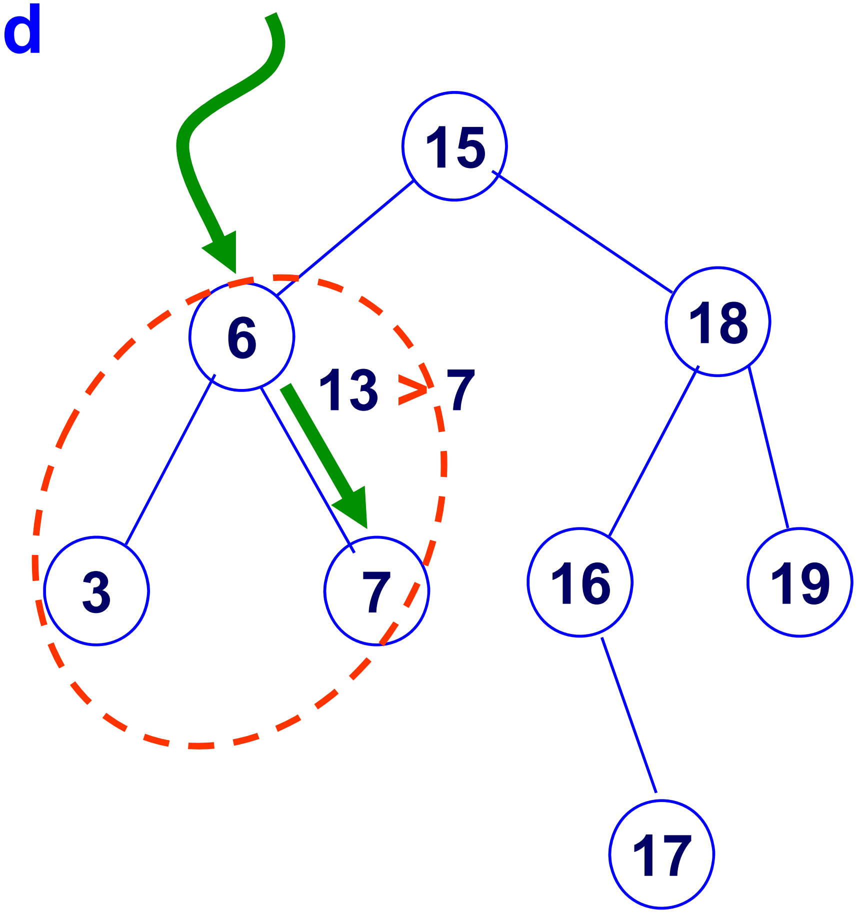

```
BSTinsert_recurs (root, temp) {
  if (temp.data ≤ root.data) {     1
    if (root.left = = null)
      root.left = temp
    else    // non-empty         2
      BSTinsert_recurs (root.left, temp);
  }
  else {     // goes to right
  }
}
```

Case 1: *Insert a node to an empty tree*
Case 2: *Insert to a non-empty tree, left subtree empty*
Case 3: *Insert to a non-empty tree, righ subtree empty*
Case 4: *Insert to a non-empty tree, left subtree non-empty*
Case 5: *Insert to a non-empty tree, right subtree non-empty*

Case 5: Insert to a non-empty tree, right subtree non-empty

1) Inserted data is larger than root and
2) right subtree is non-empty






```
BSTinsert_recurs (root, temp) {
  if (temp.data ≤ root.data) {
     // insert at left
  }
  else {  // goes to right
    if (root.right = = null)
      root.right = temp
    else
      BSTinsert_recurs (root.right, temp)
  }
}
```

Case 1: *Insert a node to an empty tree*
Case 2: *Insert to a non-empty tree, left subtree empty*
Case 3: *Insert to a non-empty tree, righ subtree empty*
Case 4: *Insert to a non-empty tree, left subtree non-empty*
Case 5: *Insert to a non-empty tree, right subtree non-empty*

An exercise

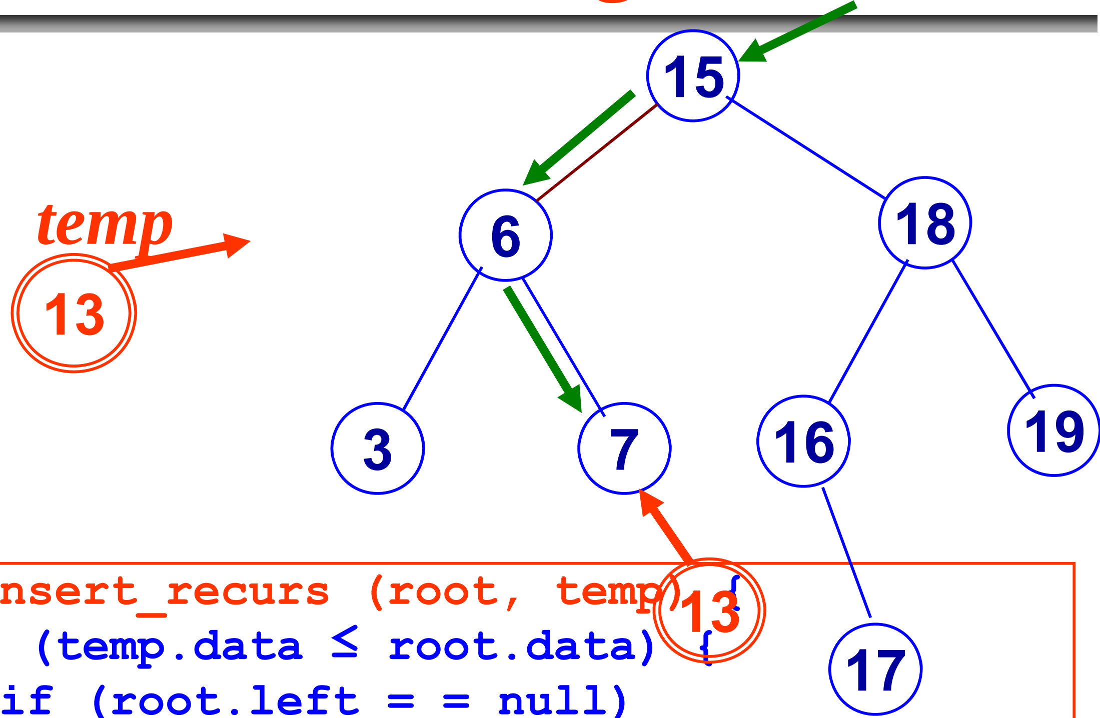
Write a general algorithm that inserts a node into a binary search tree (BST).

Hint:

Put all the code for all five cases previously discussed together.

BST Insertion Algorithm: Put all together

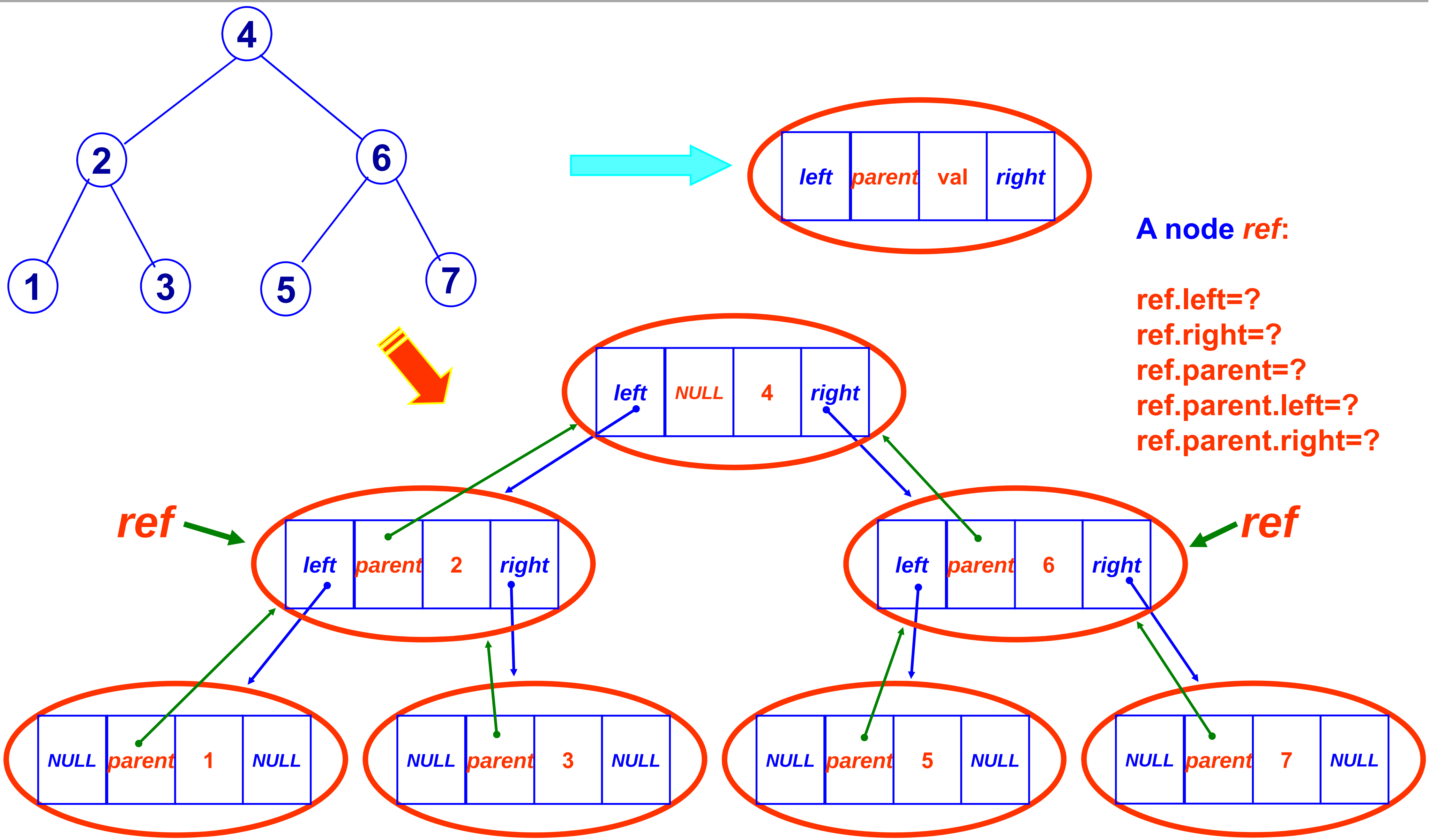
```
BSTinsert (root, val) {
   // set up node to be added to tree
   temp = new node;
   temp.data = val
   temp.left = temp.right = null

   // special case: empty tree
   if (root = = null)
     return temp;
   else
     // for all other cases
     BSTinsert_recurs (root, temp);
   return root;
}
```




```
BSTinsert_recurs (root, temp) {
  if (temp.data ≤ root.data) {
    if (root.left = = null)
      root.left = temp
    else
      BSTinsert_recurs (root.left, temp);
  }
  else {  // goes to right
    if (root.right = = null)
      root.right = temp
    else
      BSTinsert_recurs (root.right, temp)
  }
}
```

BST Deletion

Node Representation



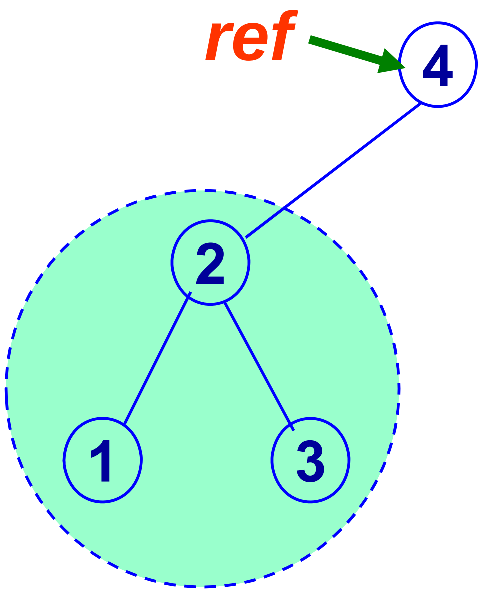
A node *ref*:

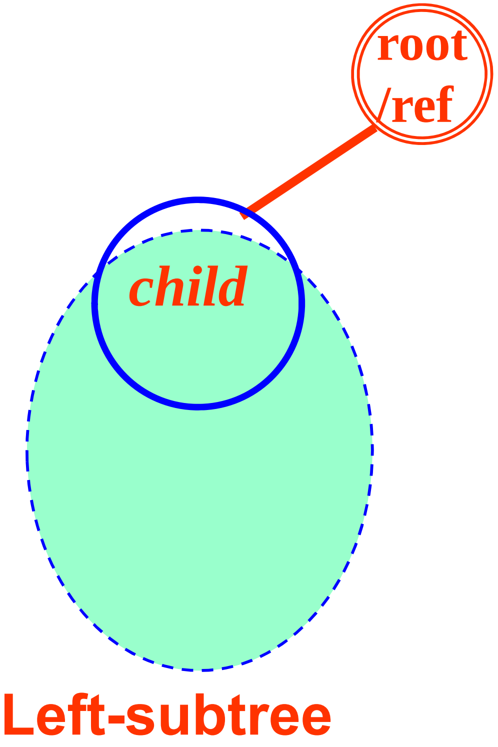
ref.left=?
ref.right=?
ref.parent=?
ref.parent.left=?
ref.parent.right=?

BST Deletion

- ***BSTreplace(root, ref)*: (for the case *ref* has one or no child)**

1. **If *ref* is *root*, set the child of *ref* as the root, stop.**
2. **If *ref* is neither *root* nor *leaf*, set the child of *ref* as *ref.parent*'s child, stop.**
3. **If *ref* is *leaf*, set *ref.parent*'s child NULL, stop.**

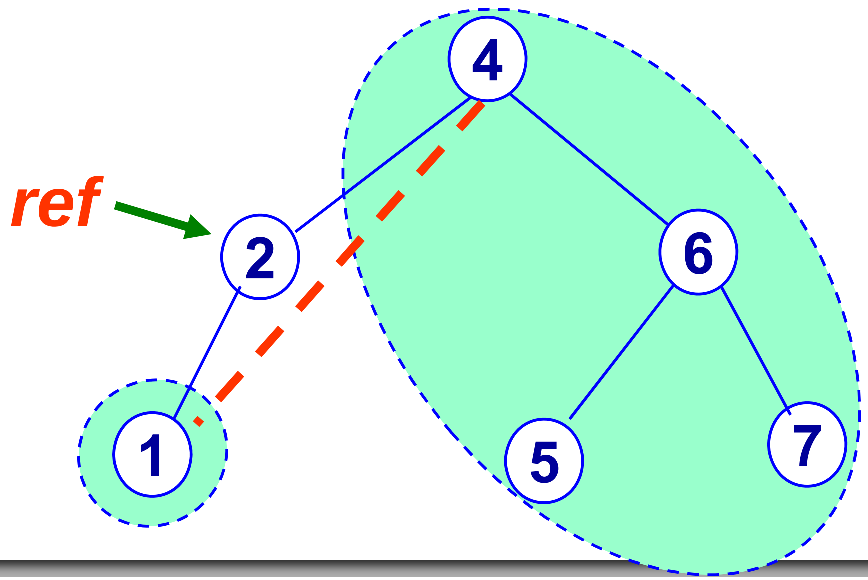


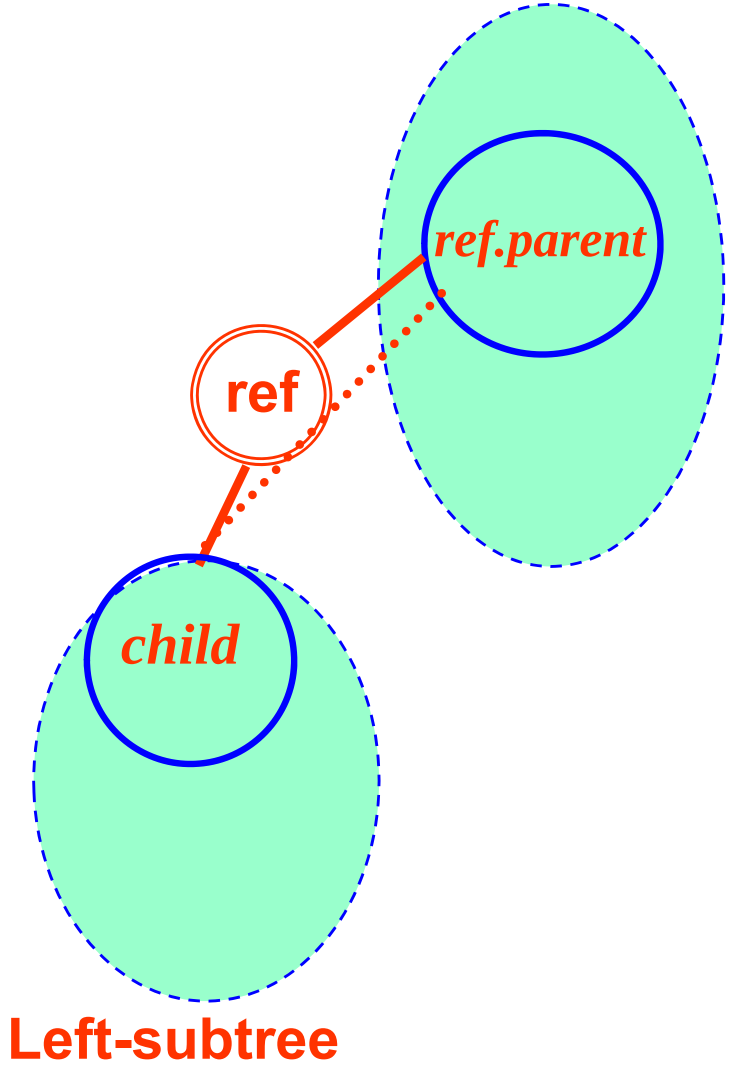


BST Deletion

- ***BSTreplace(root, ref)*: (for the case *ref* has one or no child)**

1. **If *ref* is *root*, set the child of *ref* as the root, stop.**
2. **If *ref* is neither *root* nor *leaf*, set the child of *ref* as *ref.parent*'s child, stop.**
3. **If *ref* is *leaf*, set *ref.parent*'s child NULL, stop.**

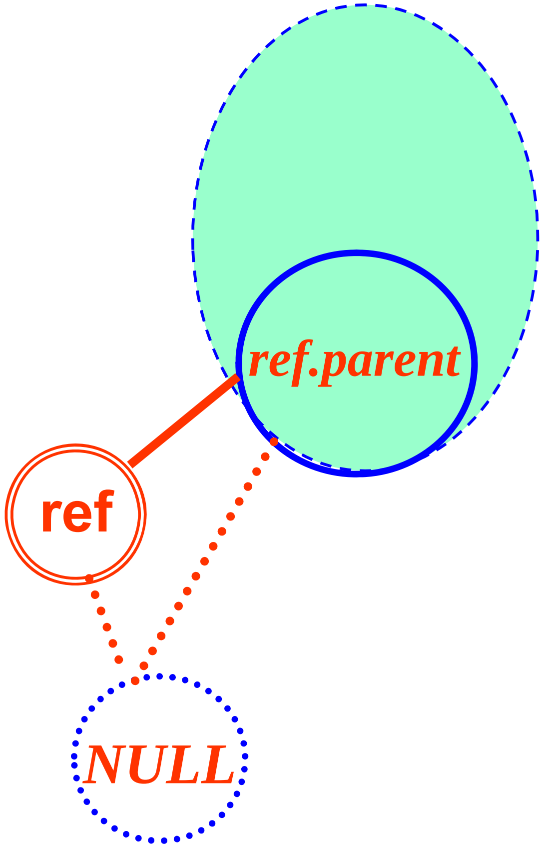


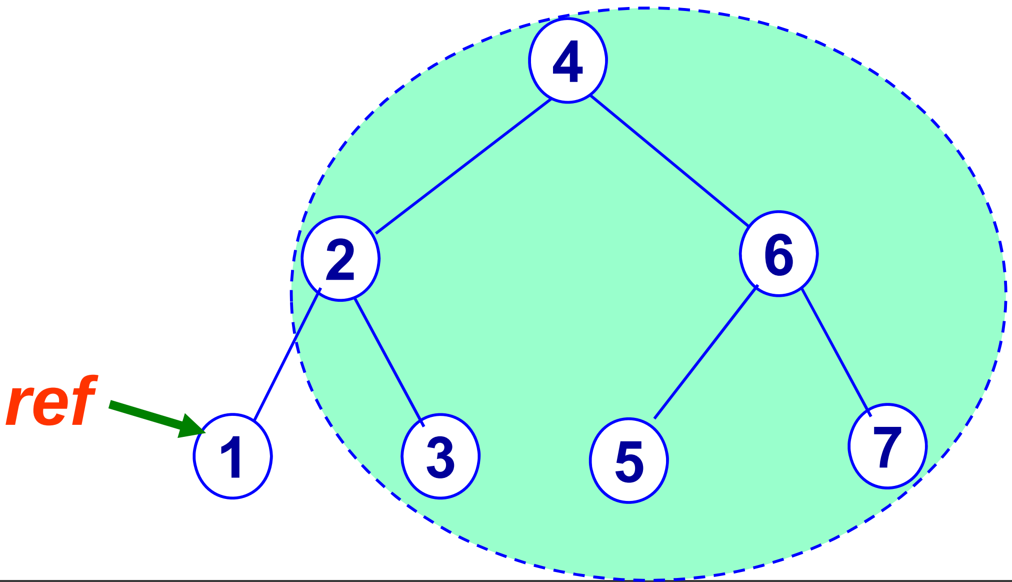


BST Deletion

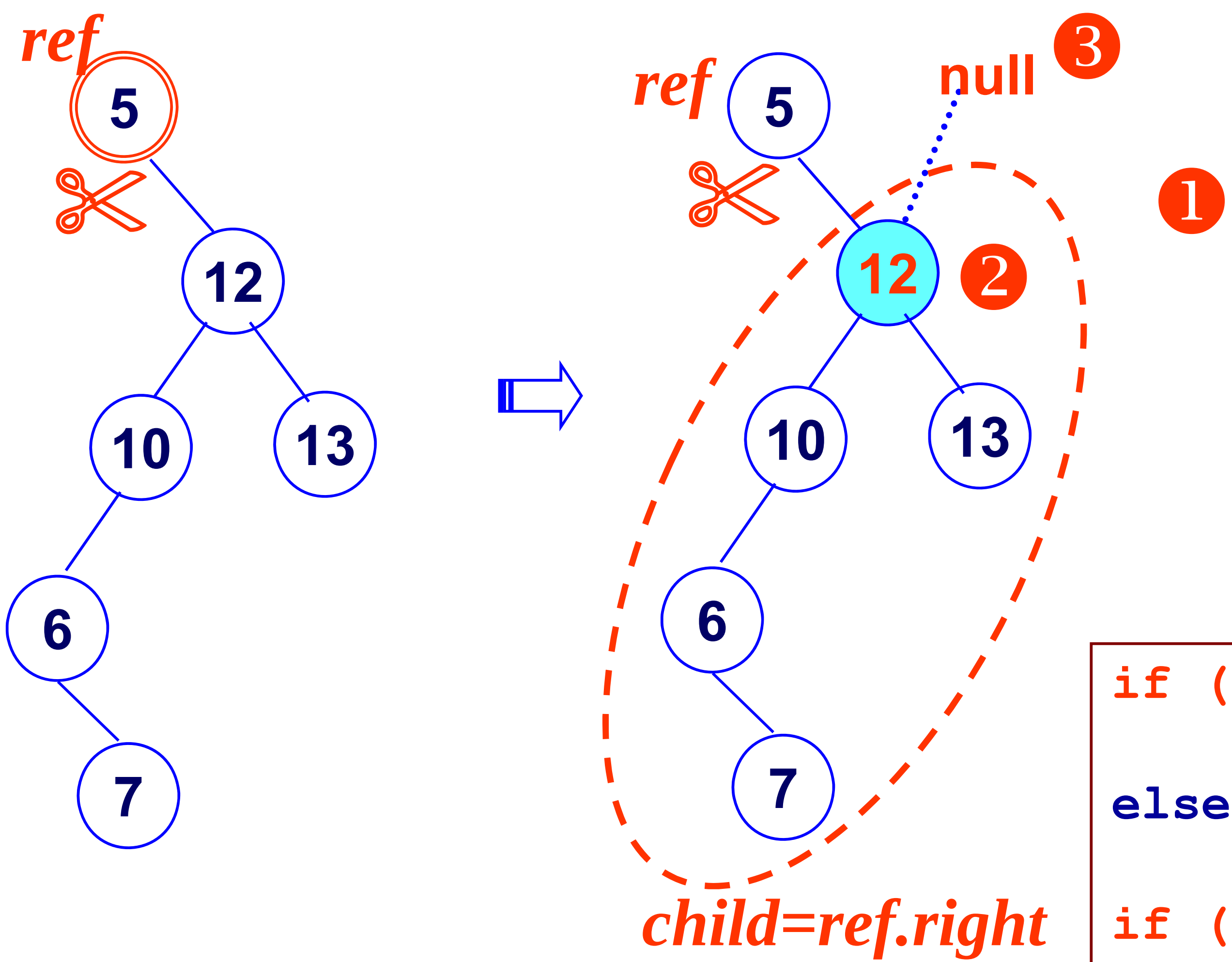
- ***BSTreplace(root, ref)*: (for the case *ref* has one or no child)**

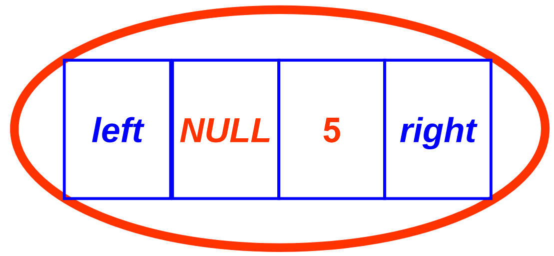
1. **If *ref* is *root*, set the child of *ref* as the root, stop.**
2. **If *ref* is neither *root* nor *leaf*, set the child of *ref* as *ref.parent*'s child, stop.**
3. **If *ref* is *leaf*, set *ref.parent*'s child NULL, stop.**





Deletion Example: only one child (Delete root)



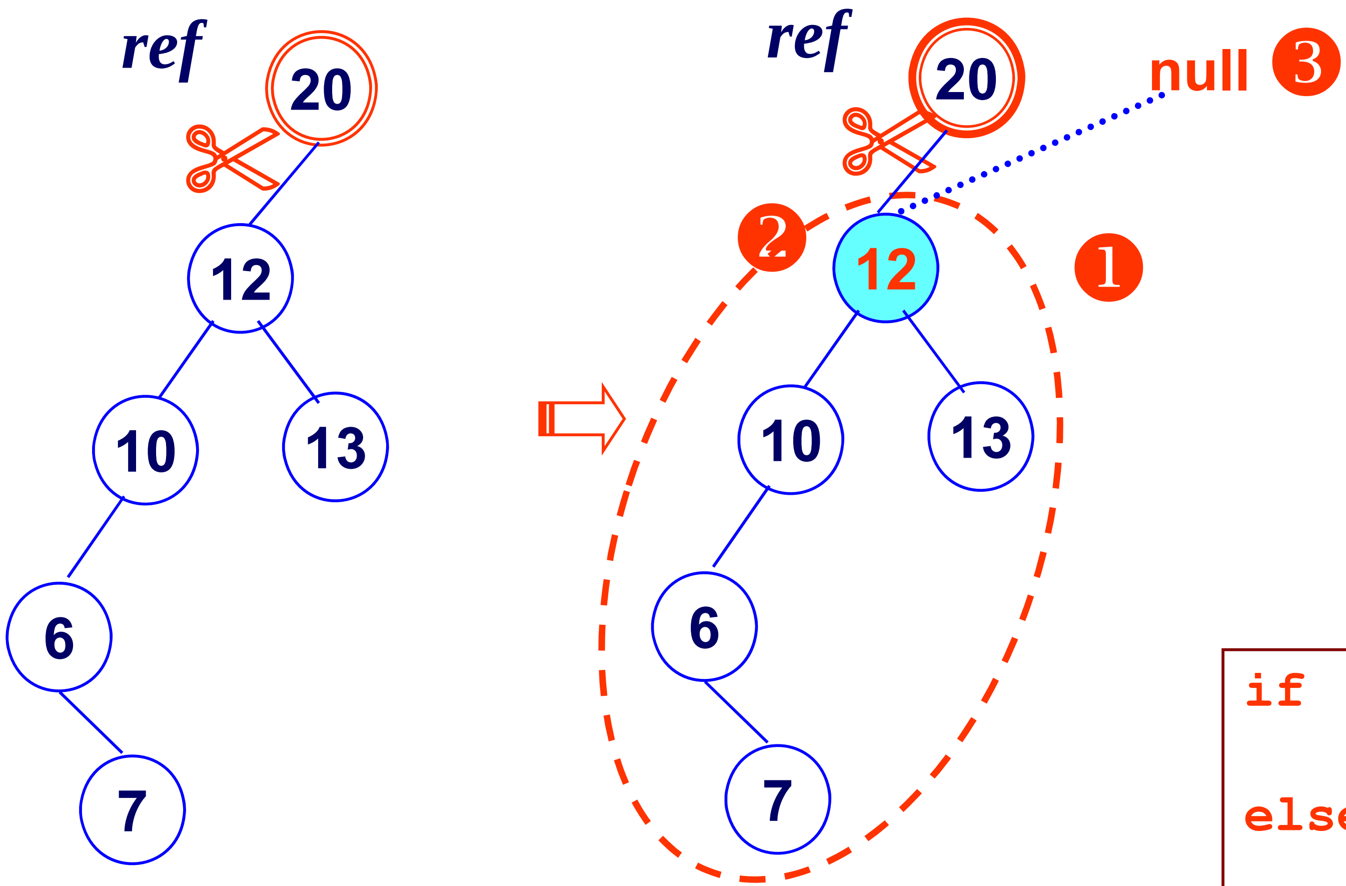


- ✓ First for the case of right child

```
if (ref.left = = null)                       1
        child = ref.right
else child = ref.left

if (ref == root) {
    if (child ≠ null)                        2
        child.parent = null;                 3
    return child;
}
```

- ✓ Keep track its child 1
- ✓ modify its parent child.parent with null 3

Deletion Example: only one child (Delete root)



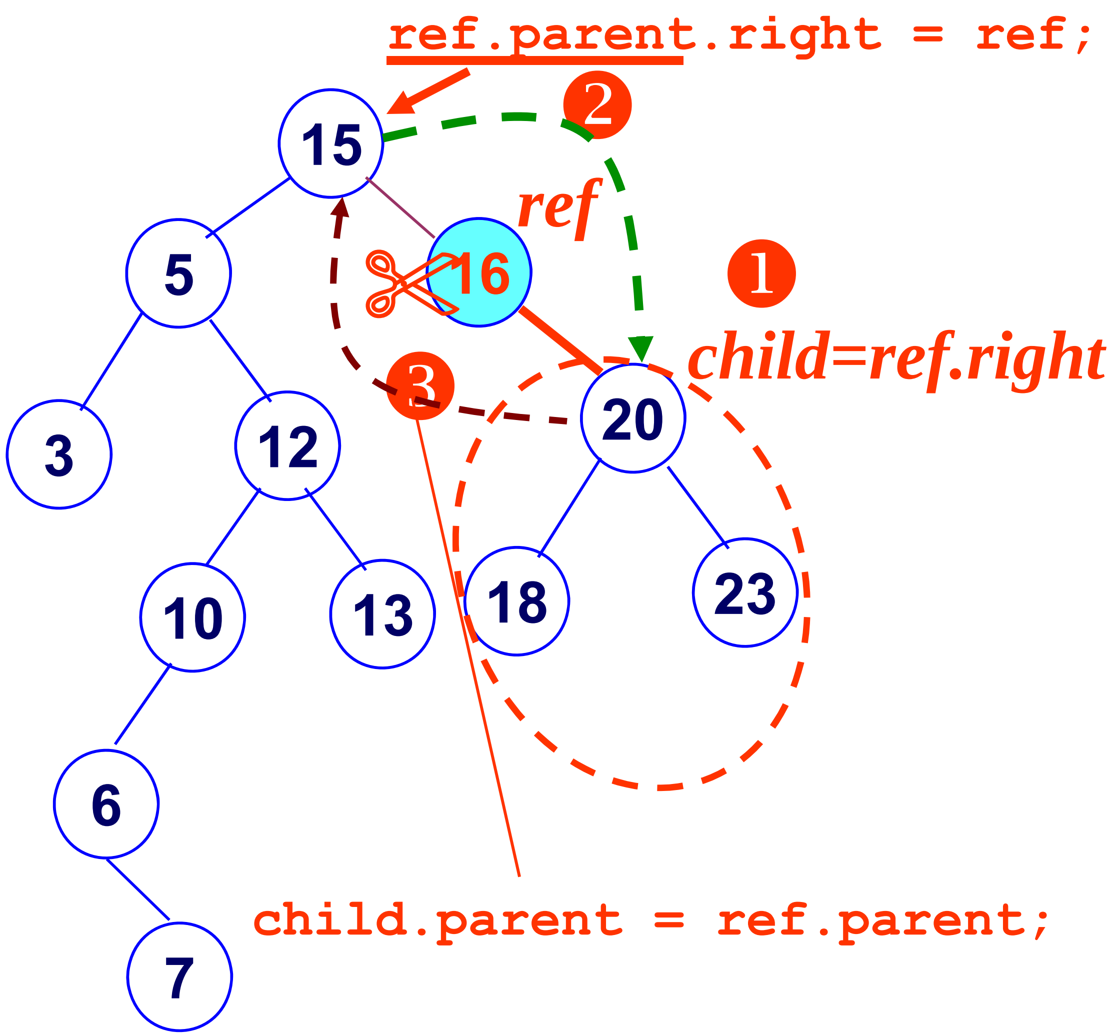
child=ref.left

- ✓ Similarly for the case of left child

```
if (ref.left = = null)
        child = ref.right;
else child = ref.left;   ❶

if (ref == root) {
    if (child ≠ null) ❷
        child.parent = null;❸
    return child
}
```

- ✓ Keep track its child
- ✓ modify its parent child.parent with null

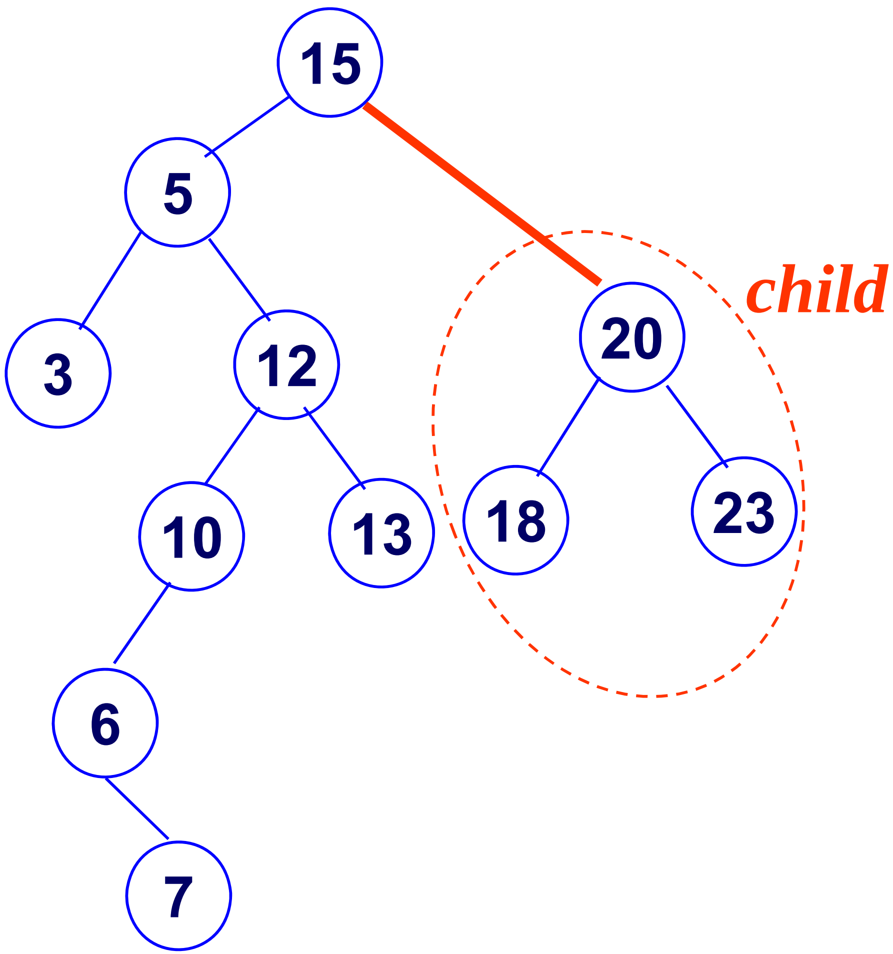
Deletion Example: only one child (not root)




```
if (ref.left == null)
    child = ref.right;  1
else child = ref.left;

// is ref left child
if (ref.parent.left == ref )
    ref.parent.left = child;
else  2
    ref.parent.right = child;

if (child ≠ null)    3
    child.parent = ref.parent;
return root;
```

- ✓ **Keep track its child**
- ✓ **modify its parent child.parent**

Deletion Example: only one child

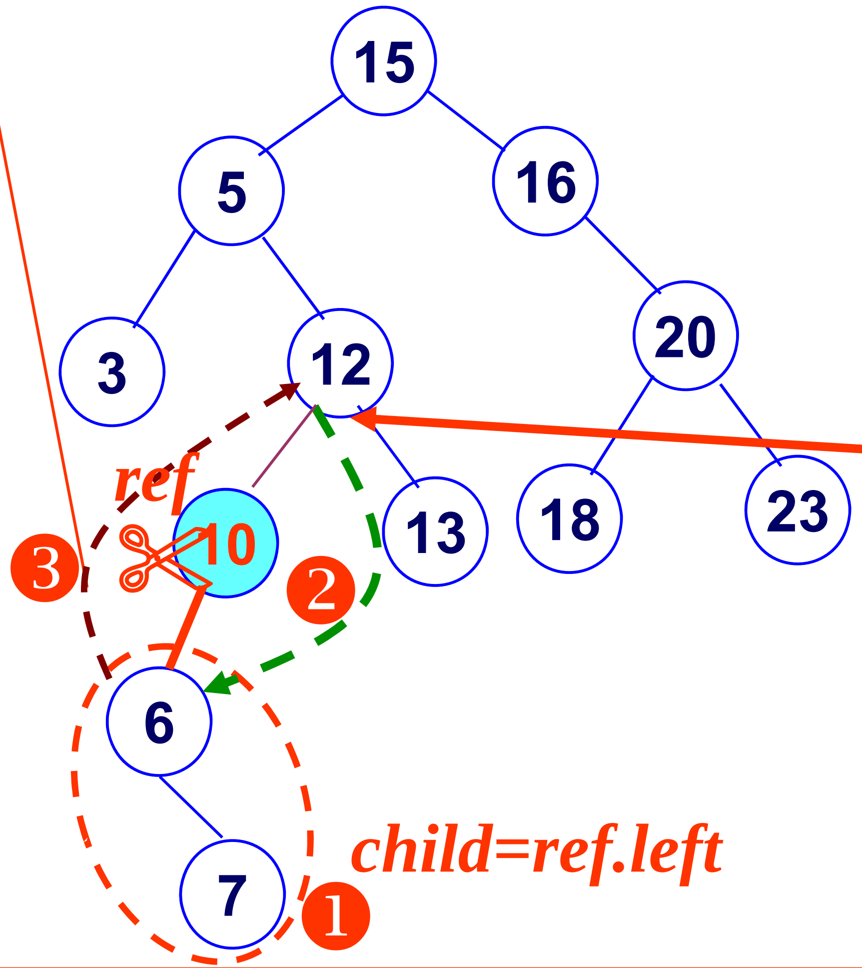


✓ Tide up

✓ That is, if **ref** has only one child, we replace it by its children

Deletion Example: only one child (not root)

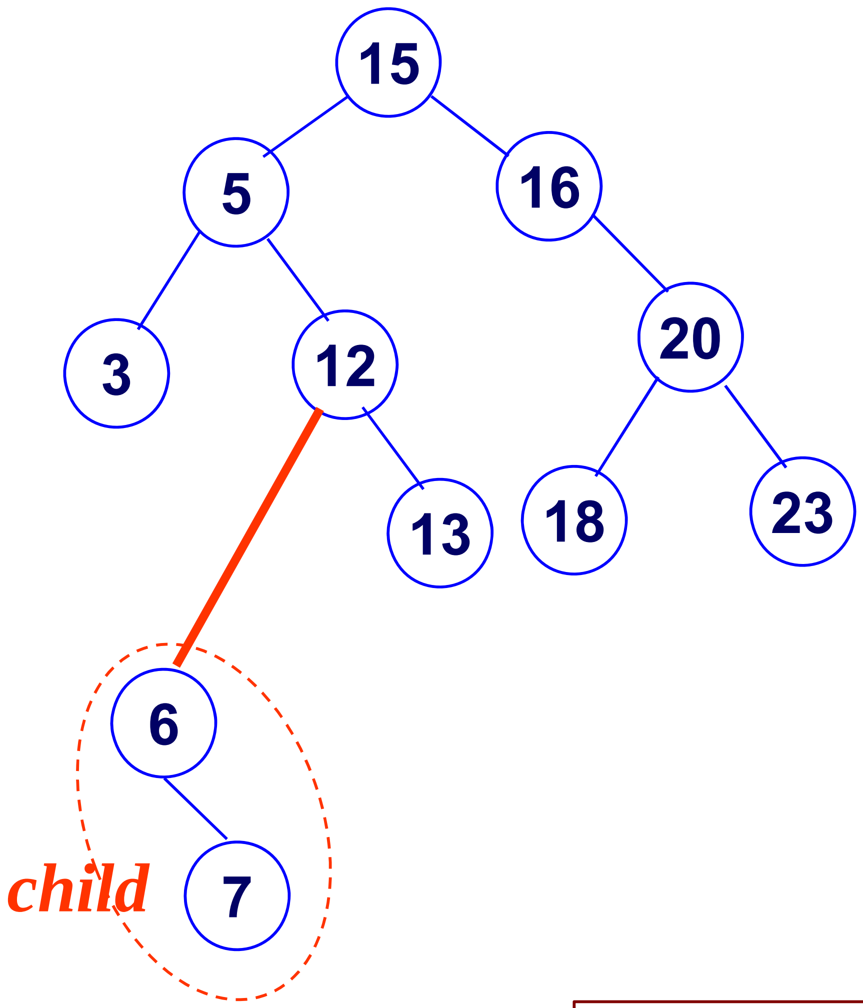
`child.parent = ref.parent;`




```
if (ref.left == null)
    child = ref.right;
else child = ref.left;  ❶

// is ref left child
if (ref.parent.left == ref )
    ref.parent.left = child; ❷
else
    ref.parent.right = child;

if (child ≠ null)        ❸
    child.parent = ref.parent;
return root;
```

- ✓ **Keep track its child**
- ✓ **modify its parent child.parent**

Deletion Example: only one child

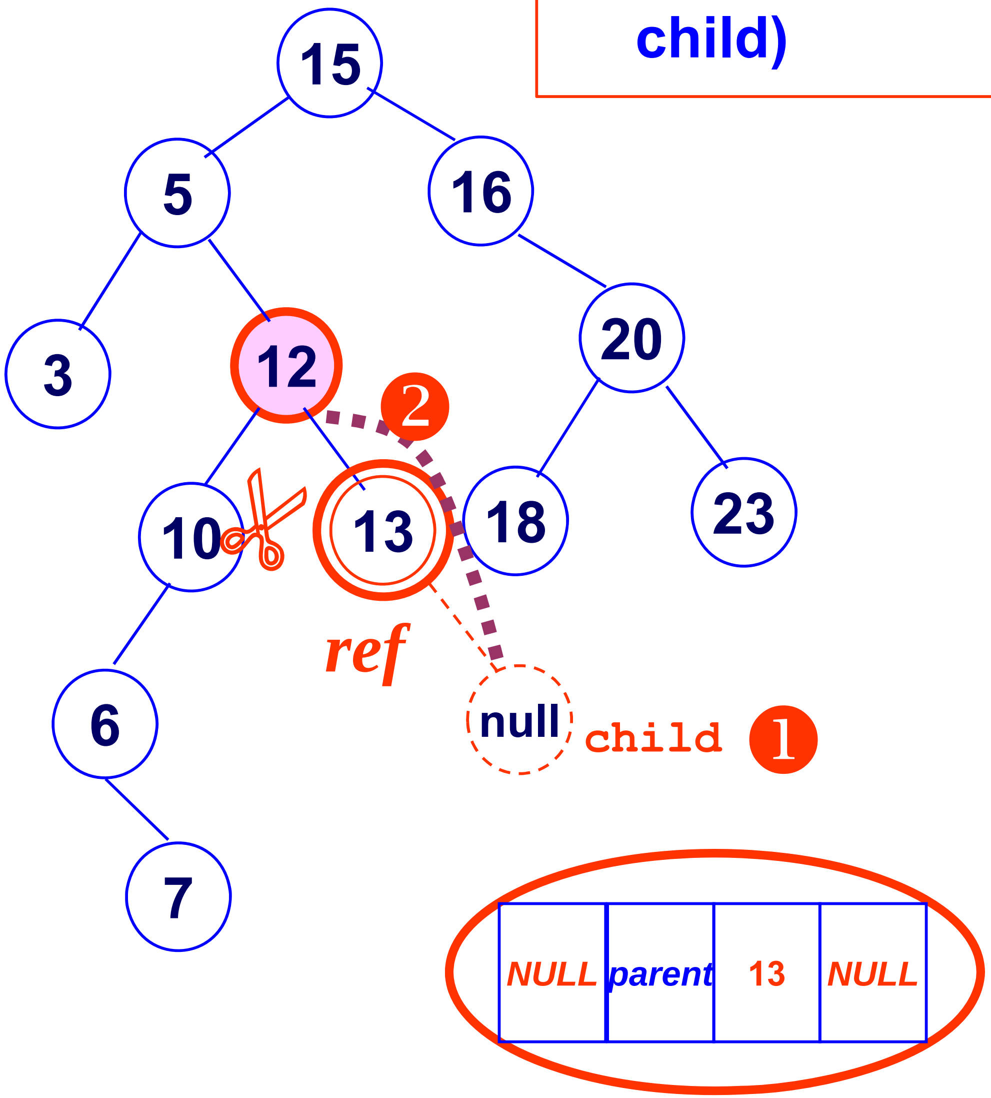


✓ Tide up

✓ That is, if **ref** has only one child, we replace it by its children

Deletion Example: no children (Delete *leaf*)

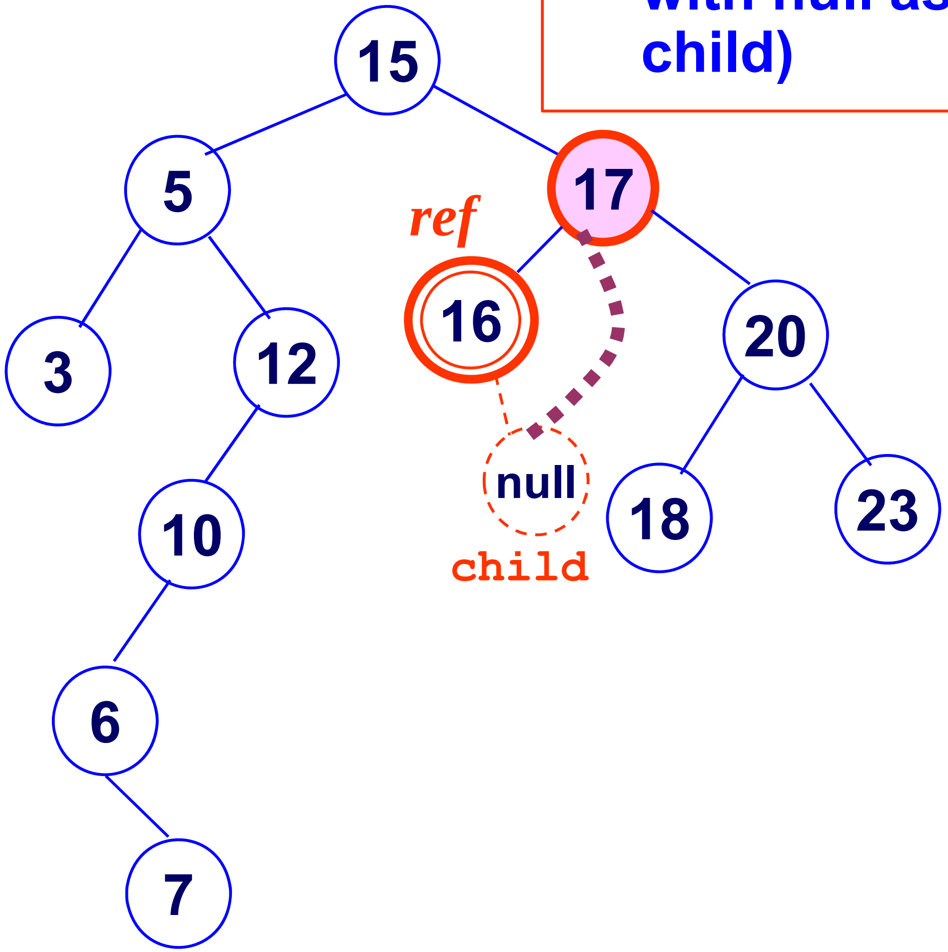
- ✓ If *ref* has no children, modify its parent ref.parent with null as its child (which means, *ref* has a null child)



- ✓ *ref* is the right child of its parent

```
if (ref.left == null)
    child = ref.right; ①
else child = ref.left;

// is ref left child
if (ref.parent.left == ref )
    ref.parent.left = child;
else
    ref.parent.right = child; ②

if (child ≠ null)
    child.parent = ref.parent;
return root;
```

Deletion Example: no children (Delete leaf)



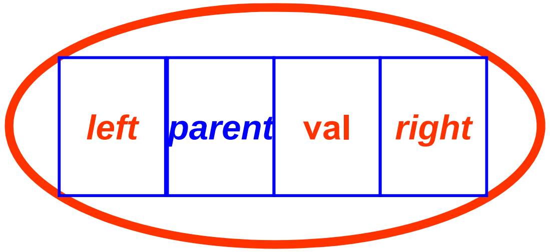
- ✓ If *ref* has no children, modify its parent **ref.parent** with null as its child (which means, *ref* has a null child)

- ✓ *ref* is the left child of its parent

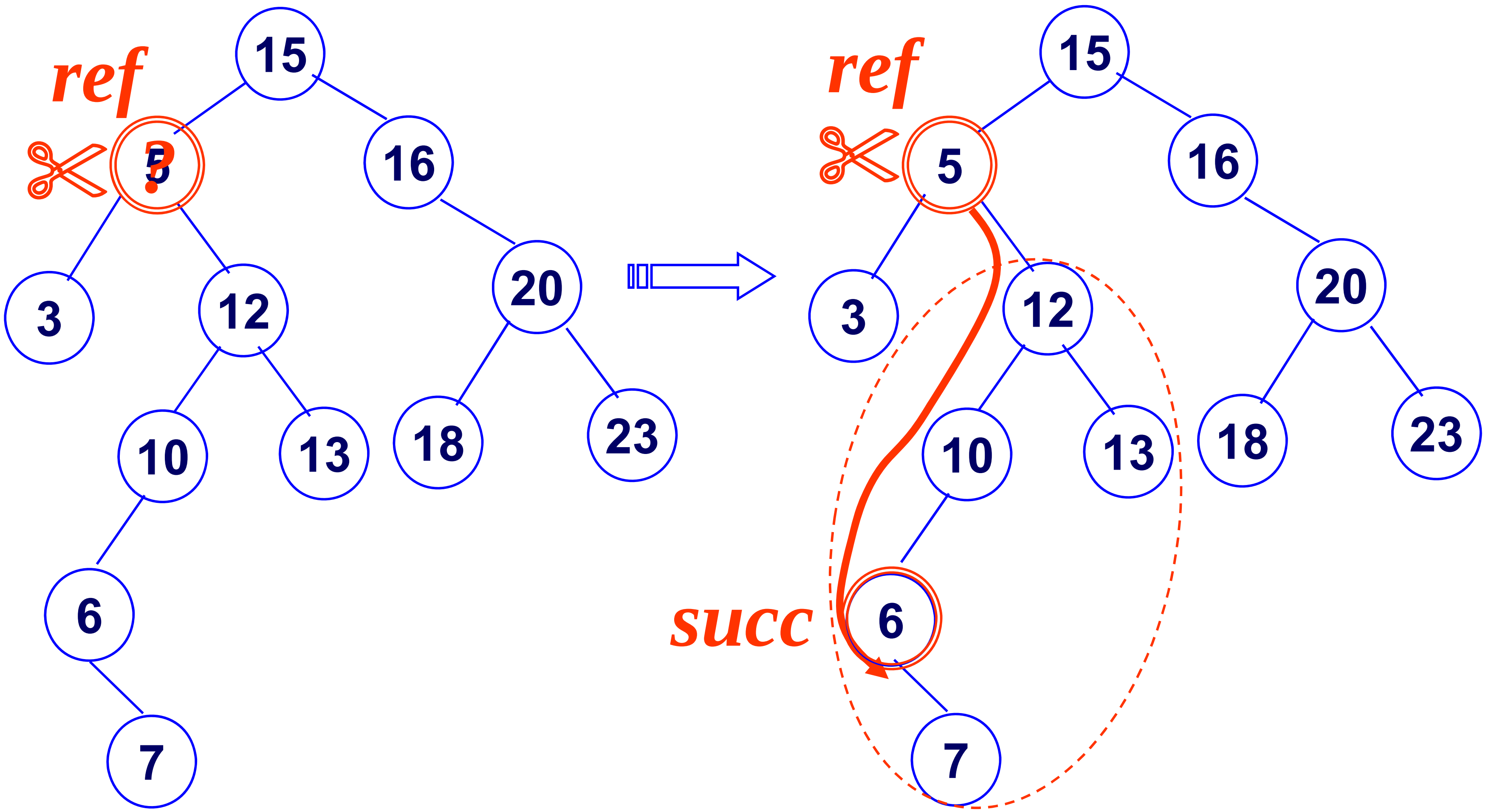
```
if (ref.left == null)
   child = ref.right;
else child = ref.left;

// is ref left child
if (ref.parent.left == ref )
   ref.parent.left = child;
else
   ref.parent.right = child;

if (child ≠ null)
   child.parent = ref.parent;
return root;
```

Put all together-- Deletion: No child or only one child

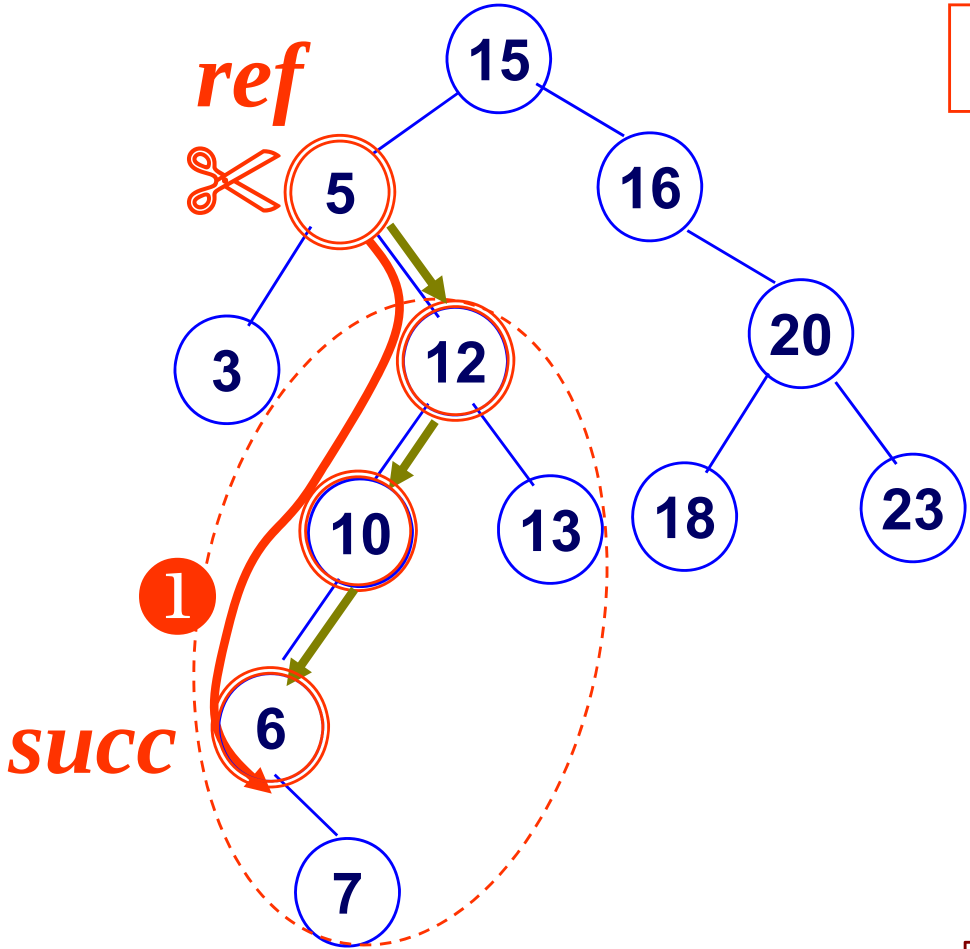
```
BSTreplace(root, ref) {// ref has only one child
    // set child to ref's child, or null if no child
    if (ref.left == null)
        child = ref.right
    else child = ref.left

    if (ref == root) {// delete root
        if (child ≠ null)
            child.parent = null
        return child
    }

    if (ref.parent.left == ref ) // is ref left child
        ref.parent.left = child
    else
        ref.parent.right = child

    if (child ≠ null)
        child.parent = ref.parent
    return root
}
```



Deletion Example: Two children



Deletion Example: Two children



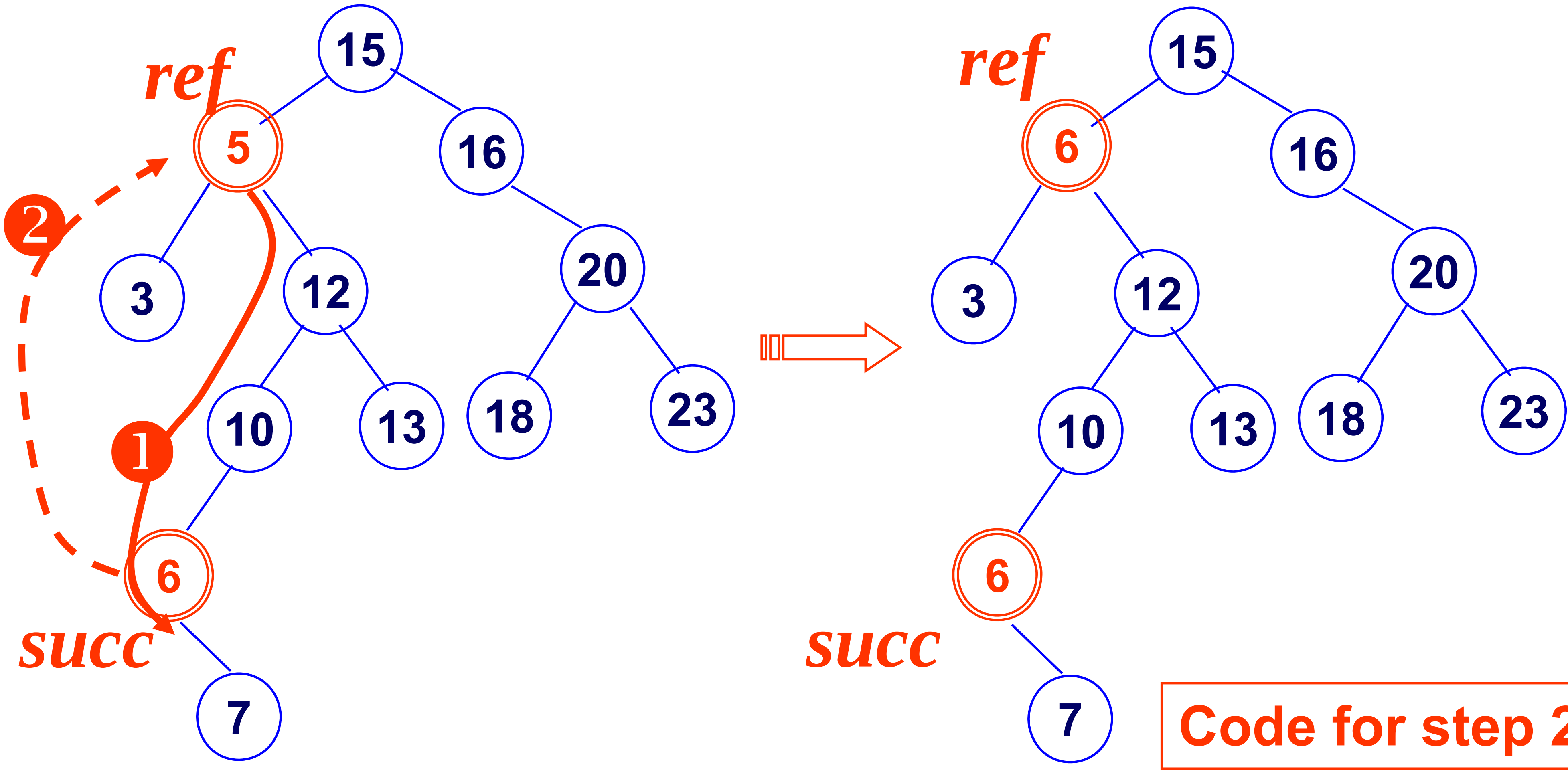
Code for step 1:

```
succ = ref.right
while (succ.left ≠ null)
    succ = succ.left        ❶
```

The smaller data is on **left subtree**, we search for **mini** Until *succ* has no left child

1. We locate the node *succ* containing the **mini** data, 6, in its **right subtree**, (*succ* **must have no left child**)

Deletion Example: Two children

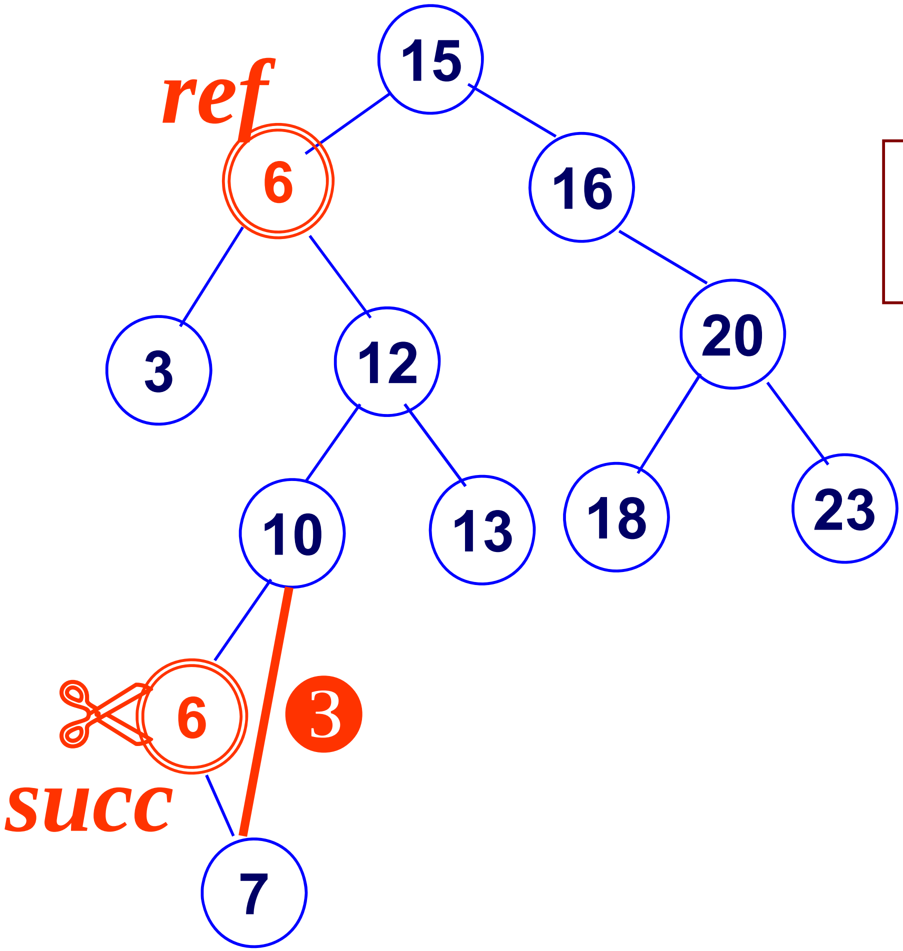


Code for step 2:

2 `ref.data = succ.data`

2. Replace the value of the node to be deleted (5) by the minimum value *succ* (6).

Deletion Example: Two children



Code for step 3:

```
// delete succ
   return BSTreplace (root, succ)
```

✓ As we discussed early, BSTreplace can delete a node in various cases

3. Delete *succ* (*succ* must have no left child) *(we have already discussed on how to delete)*

Put all together: Deletion Algorithm

```
BSTdelete(root, ref) {
    // if zero or one children, use Algorithm BSTreplace
    if (ref.left == null || ref.right == null)
        return BSTreplace (root, ref)

    //find node succ containing a minimum data item
    //   in ref's right subtree
    succ = ref.right                                    ❶
    while (succ.left ≠ null)
        succ = succ.left

    // "move" succ to ref, thus deleting ref              ❷
    ref.data = succ.data

    // delete succ                   ❸
    return BSTreplace (root, succ)
}
```