EE2008: Data Structures and Algorithms

Recursive Algorithms, Trees and Sorting

Prof Huang Guangbin

S2.1-B2-06

egbhuang@ntu.edu.sg

Coverage

	Topics	Lecturers
Weeks 1 to 5	Introduction	Low Chor Ping
	Principles of Algorithm Analysis	Low Chor Ping
Weeks 6 to 10	Trees (6 lectures)	Huang Guangbin
	Sorting (9 lectures)	Huang Guangbin
Lectures 11 to 13	Searching (4 lectures)	Huang Guangbin
	Algorithm Design Techniques (3 lectures)	Huang Guangbin
	Review (2 lectures)	Huang Guangbin

Assessments

- **Grading will be based on**
  - ***2 home work assignments (10%)***
  - ***2 practice works (lab) (10%)***
  - ***1 quiz (20%)***
  - ***Final exam (60%)***
- 1st Home Assignment
  - **Given on** *week 4*
  - **Submitted on** *week 5* **during** tutorial session 4
- 2nd Home Assignment
  - **Given on** *week 11*
  - **Submitted on** *week 12* **during** tutorial session 11
- Quiz
  - **Held on** 2nd lecturing session **in** *week 7.*

Texts & References

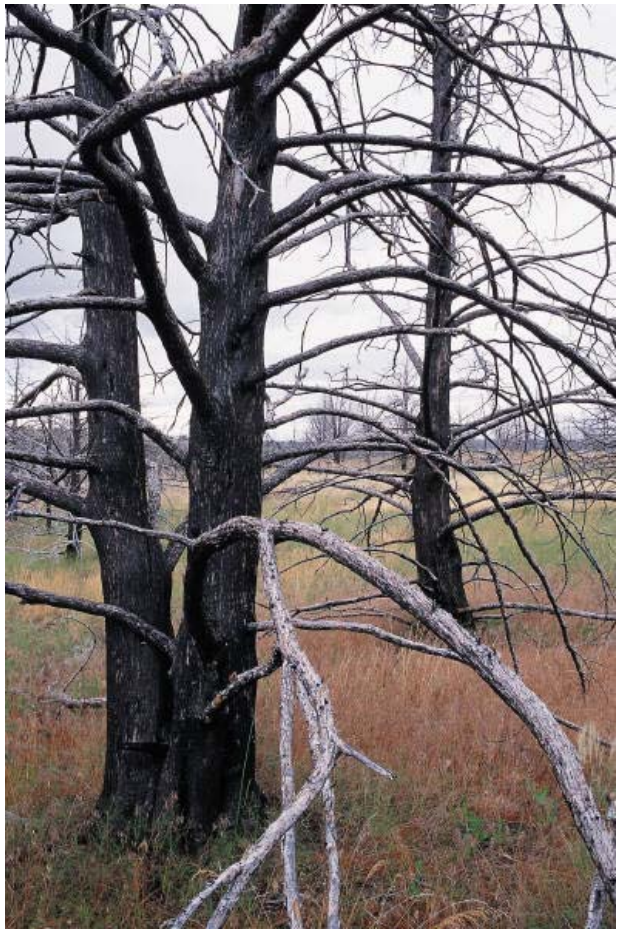
- **Textbook**
  - ✓ **Huang Guangbin and Ng Jim Mee, *Data Structures and Algorithms*, Pearson Education, 2007**
- **Reference**
  - ✓ **Richard Johnsonbaugh & Marcus Schaefer, *Algorithms*, Prentice Hall, 2004**
  - ✓ **Anany Levitin, *Introduction to the Design & Analysis of Algorithms*, Pearson Education, Inc., 2007**

Topics

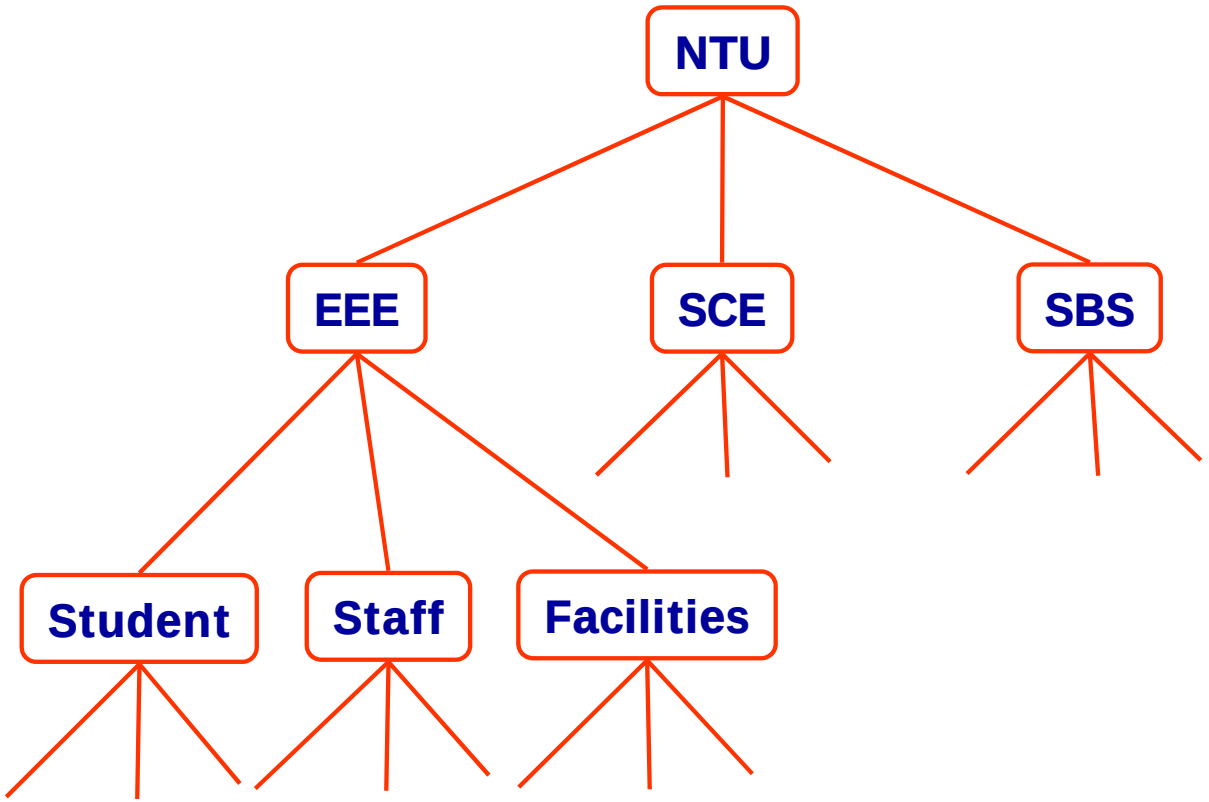
- Recursive Algorithms, Divide and Conquer Algorithms
- Tree data structures
  - Binary tree: Binary search trees, AVL trees, Heaps
- Sorting algorithms
  - Mergesort
  - Quicksort
  - Bucket Sort
  - Radix Sort
  - Heap Sort
- Selection problem and order statistics

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**)



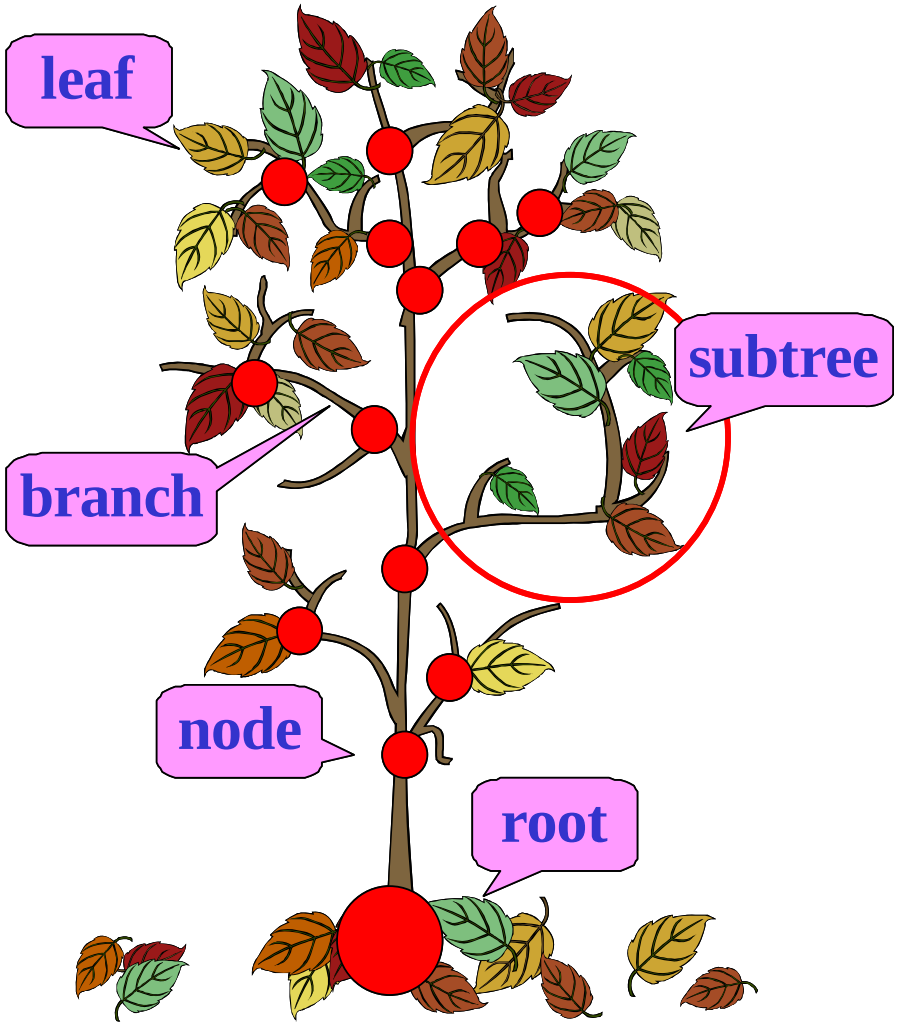
Trees

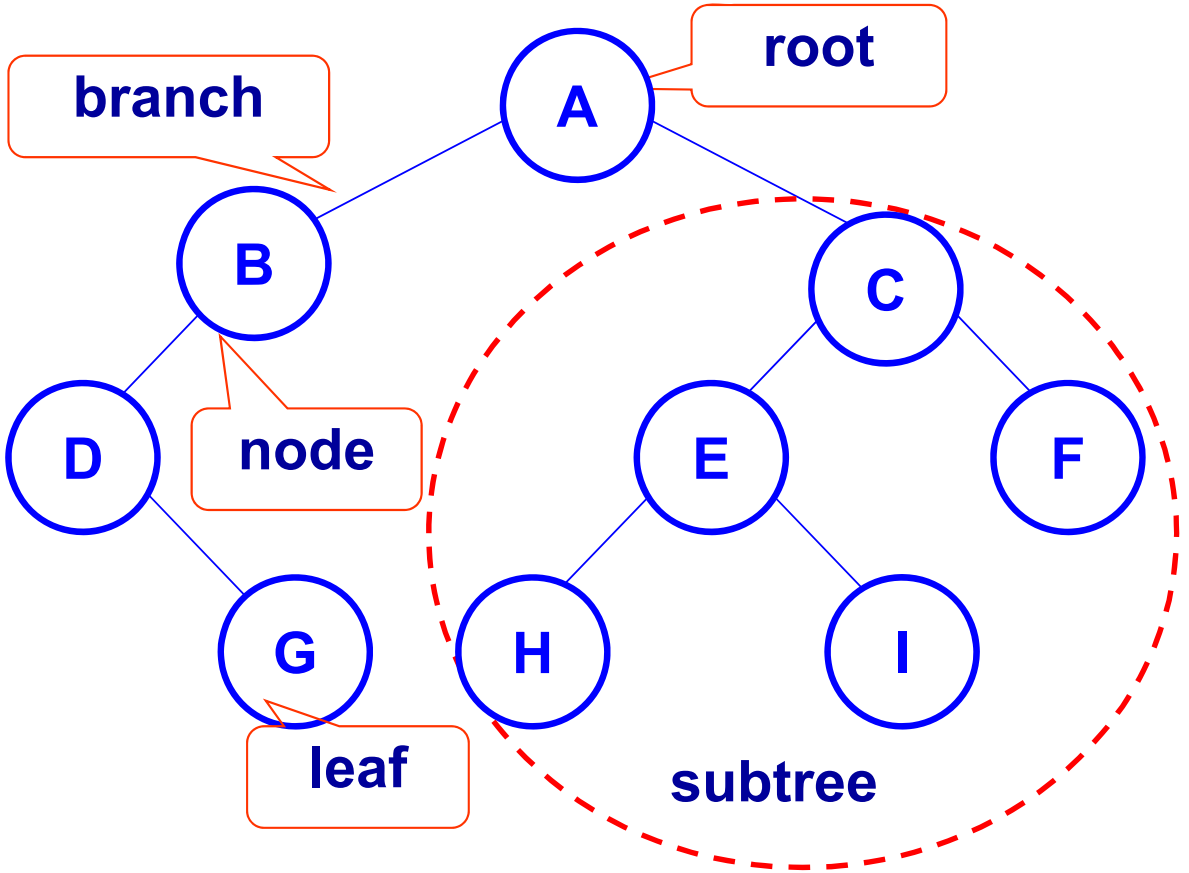


What is a Tree

- 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

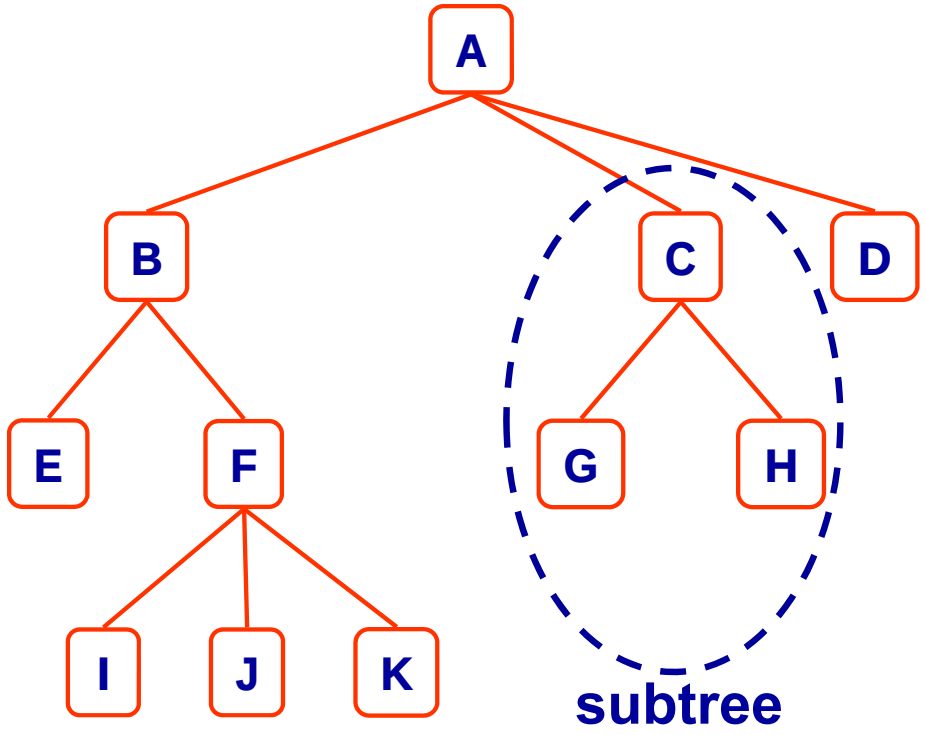




Tree Terminology

- **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



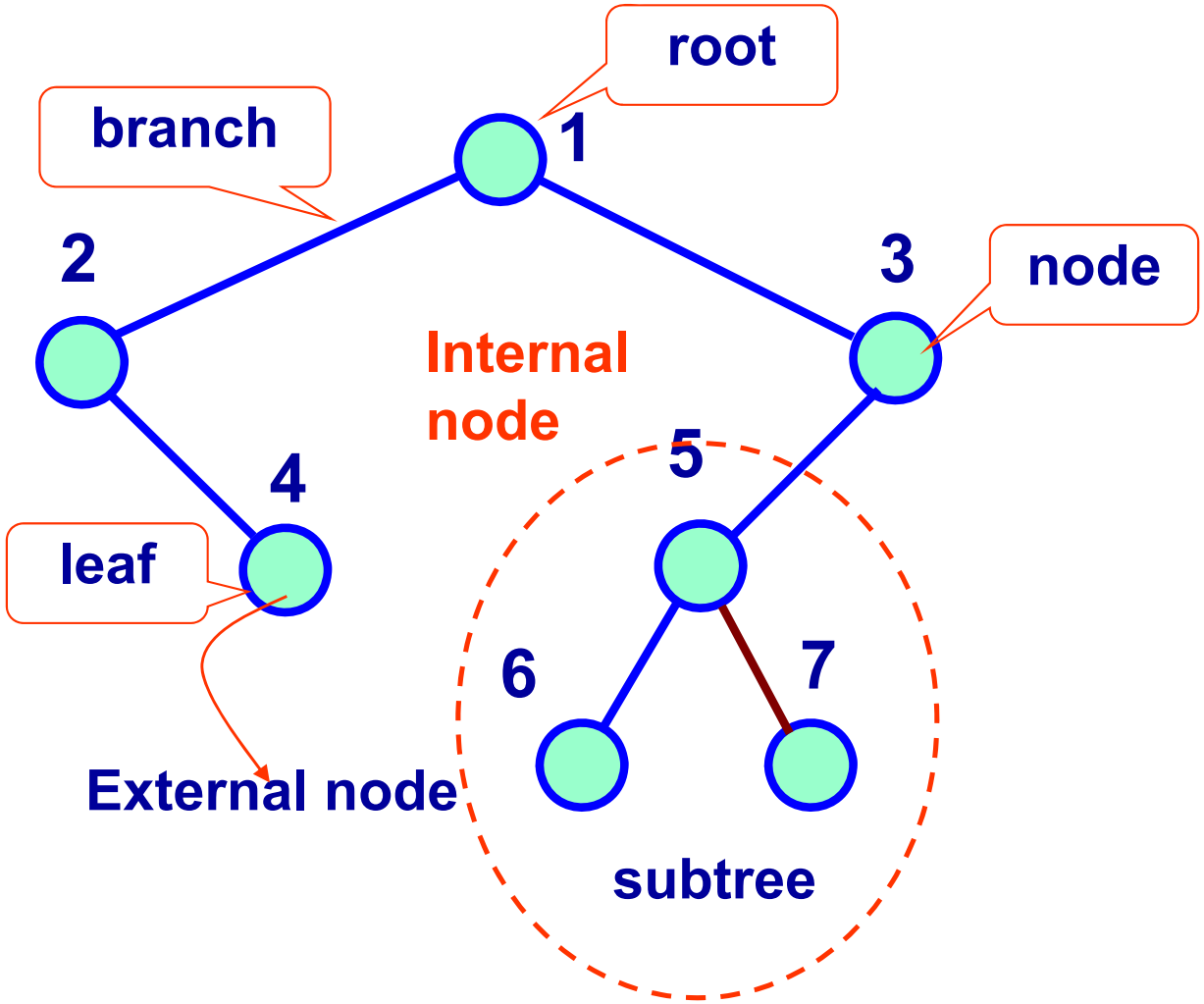
✓ This is a general rooted tree

Binary Trees

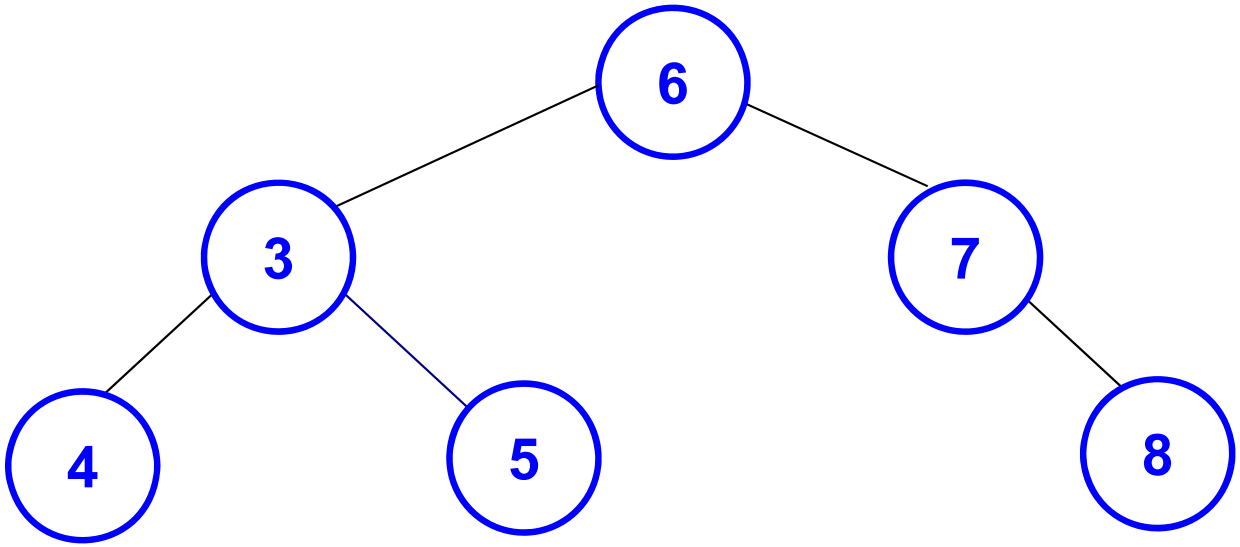
- A binary tree is a rooted tree in which each node has either no children, one child, or two children. --- Binary
  - If a node has one child, that child is designated as either a left child or a right child (but not both).
  - If a node has two children, one child is designated a left child and the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

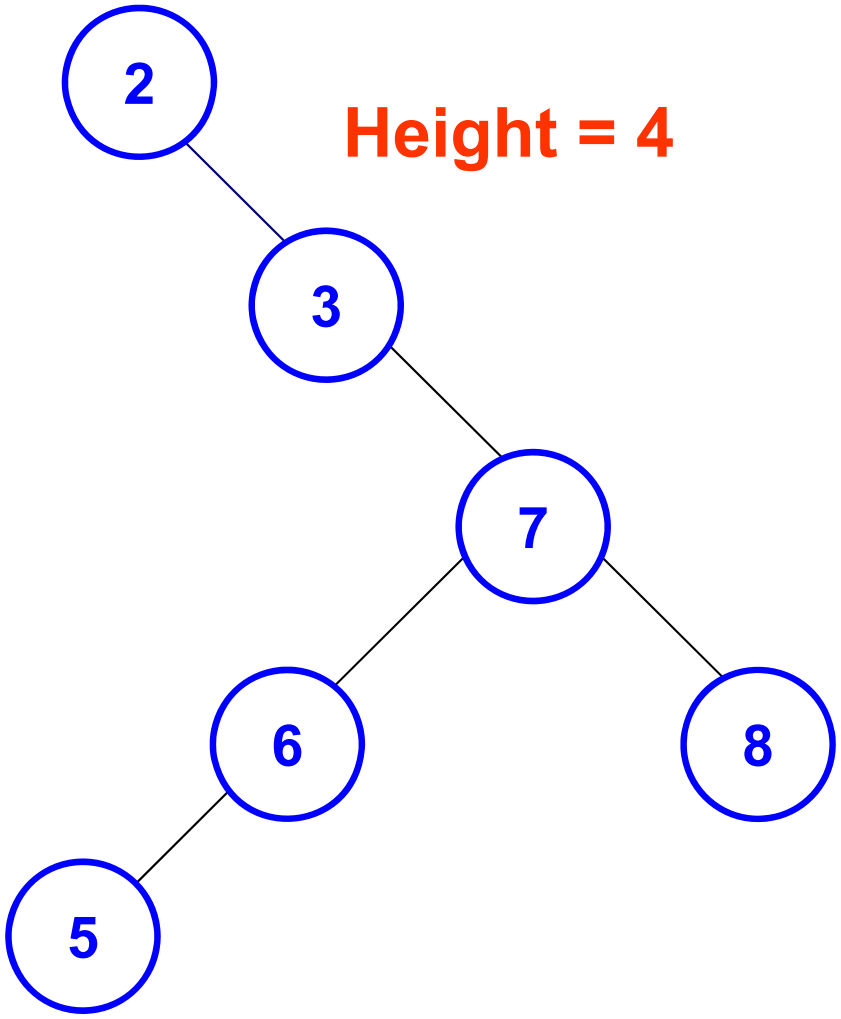
Binary Tree: Examples



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

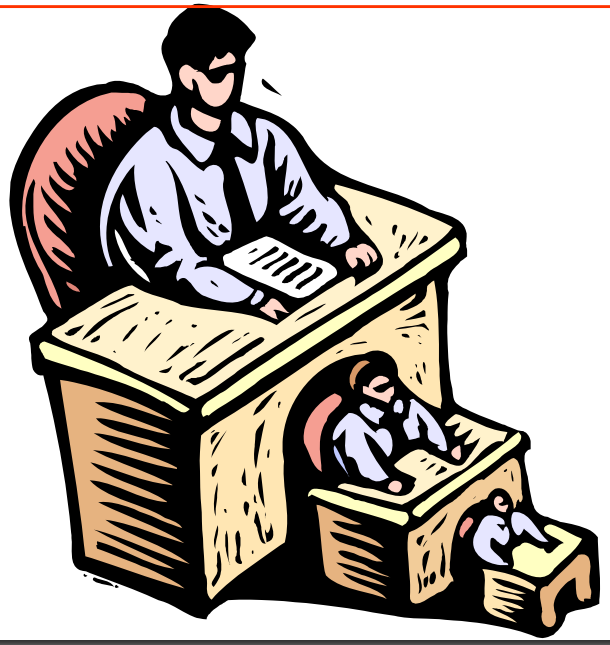
```
factorial( n )=n*(n-1)*…*2*1
```

- Familiar recursive function

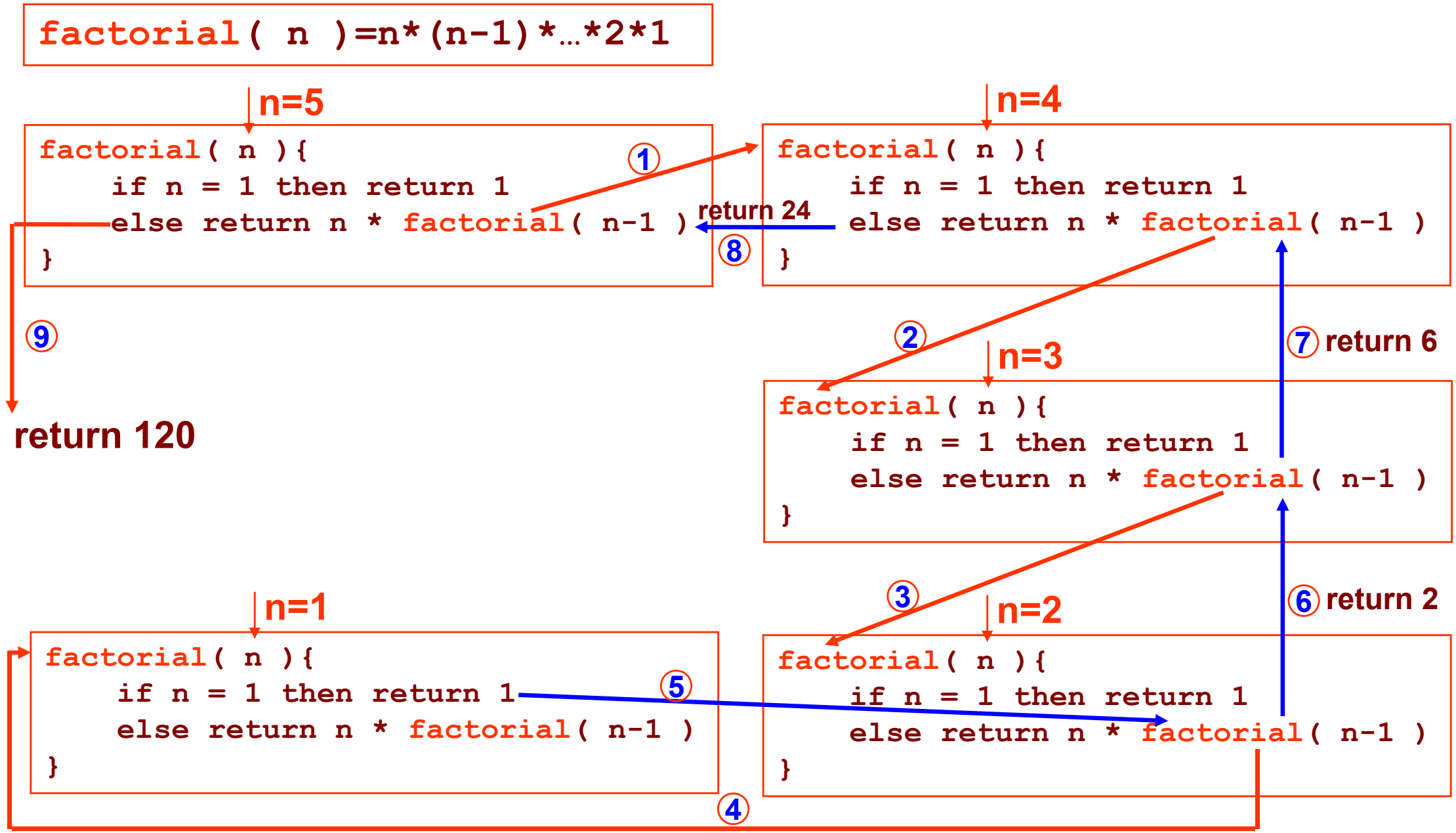
```
factorial( n ){
    if n = 1 then return 1
    else return n * factorial( n-1 )
}
```

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

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	

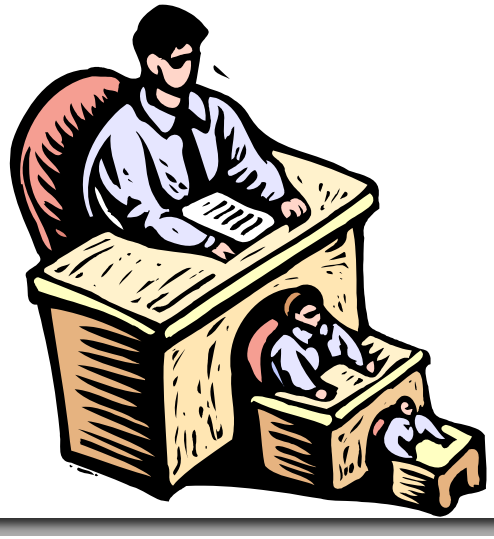


Recursion and recursive



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

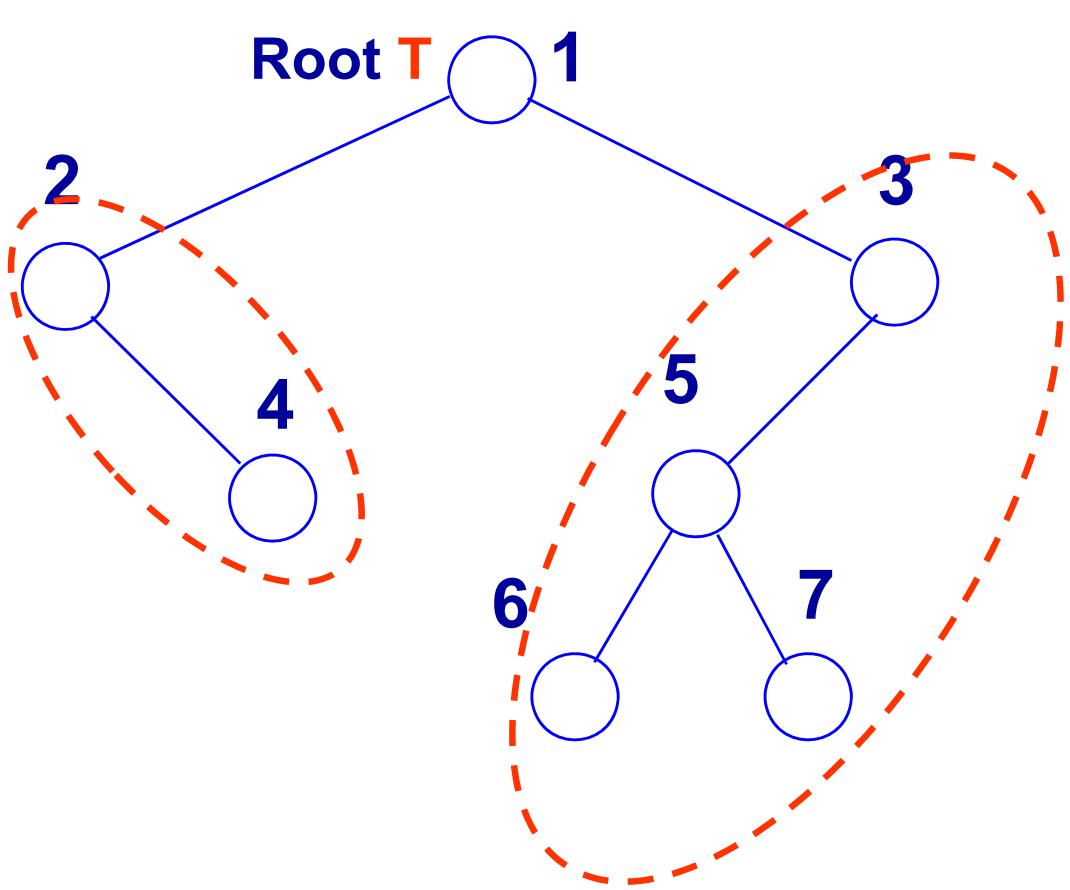
- How do you study a text book?
  - ☞ Reading a book is a recursive procedure

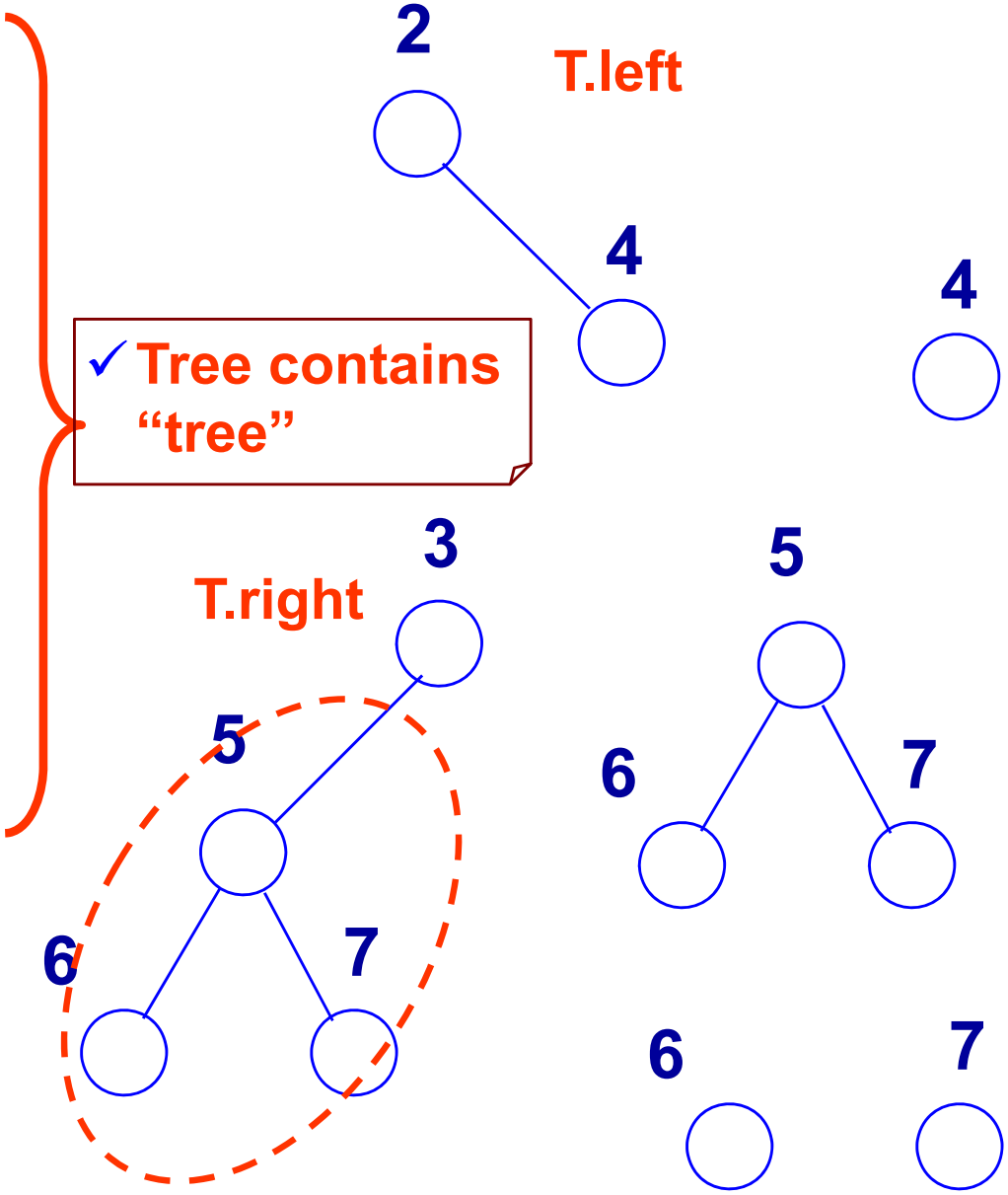
Study the book:

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

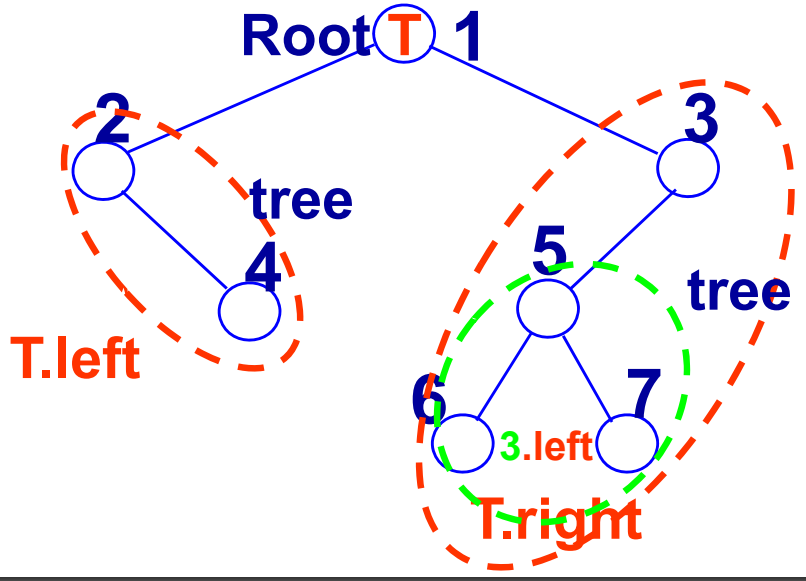
Binary Tree: Definition

- 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)

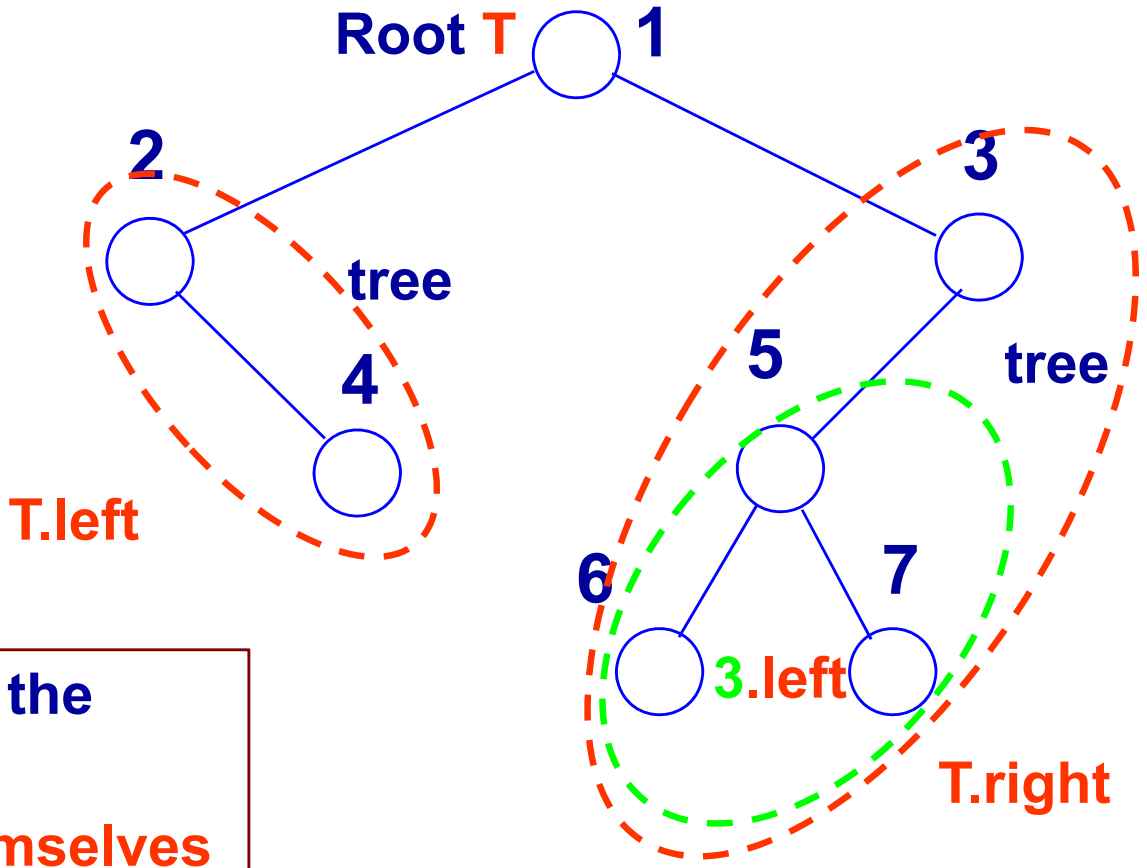
✓The binary tree is defined recursively.



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



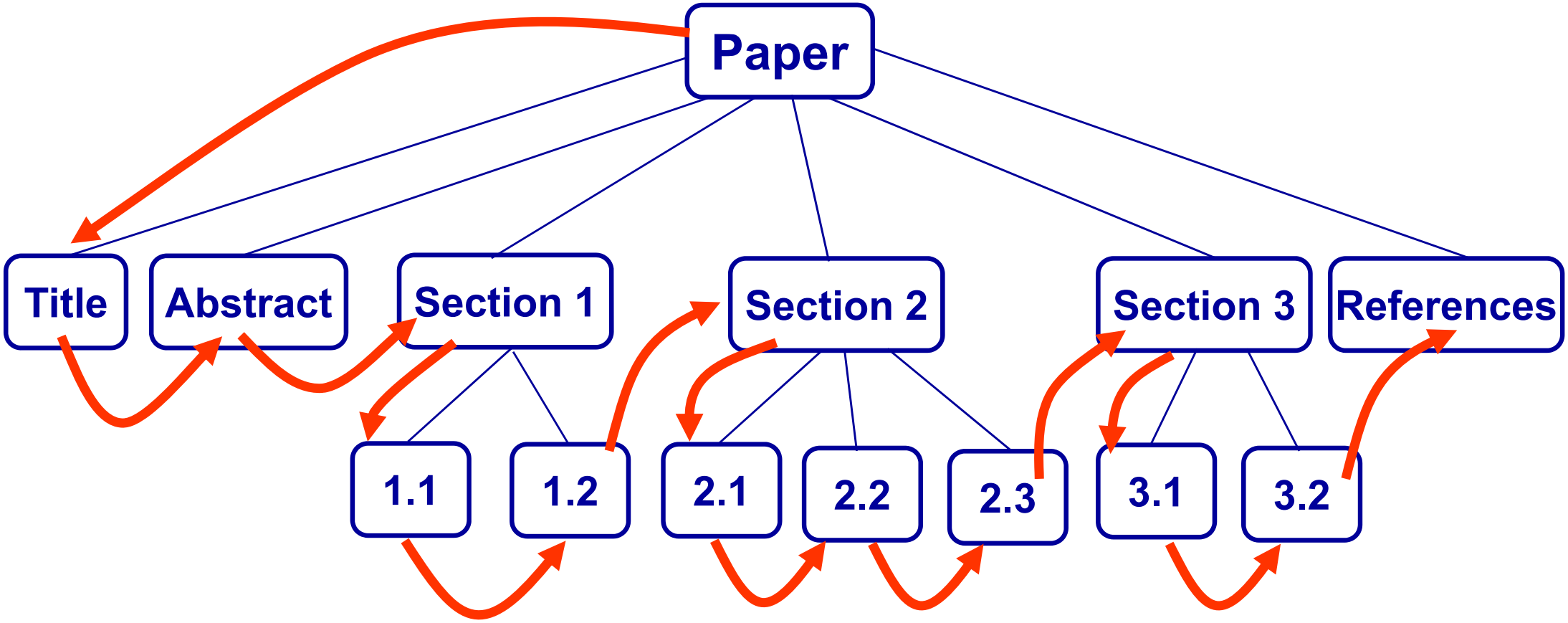
- ✓ Here is a tree that demonstrates the recursiveness of trees;
- ✓ nodes 2, 3 are roots of trees themselves

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

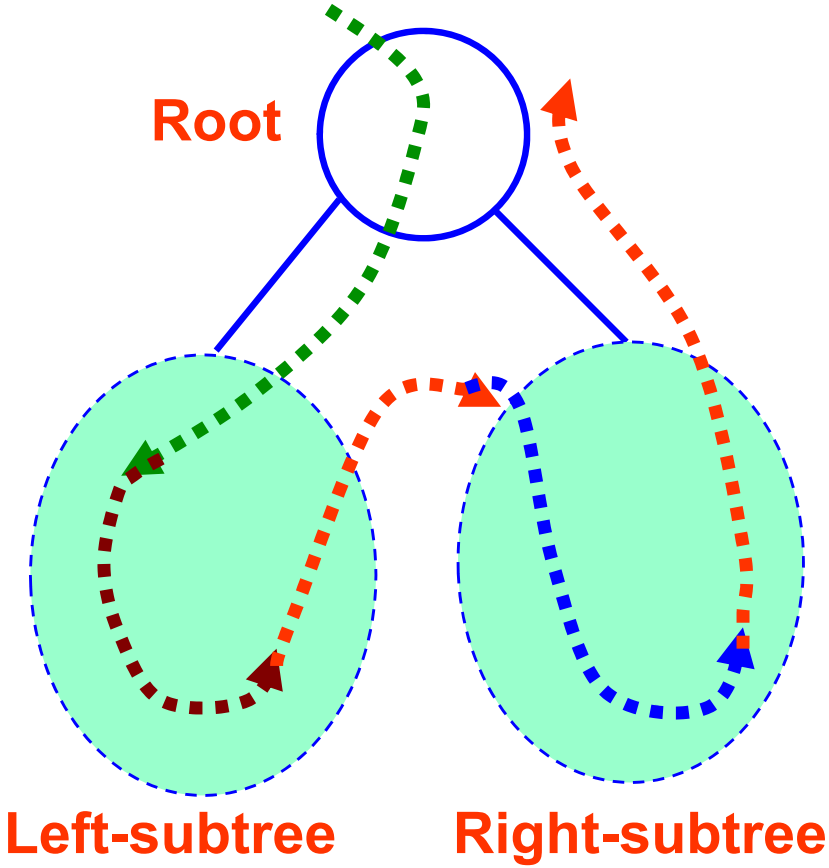
- If you read an entire paper sequentially … …



Preorder traversal: Definition

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

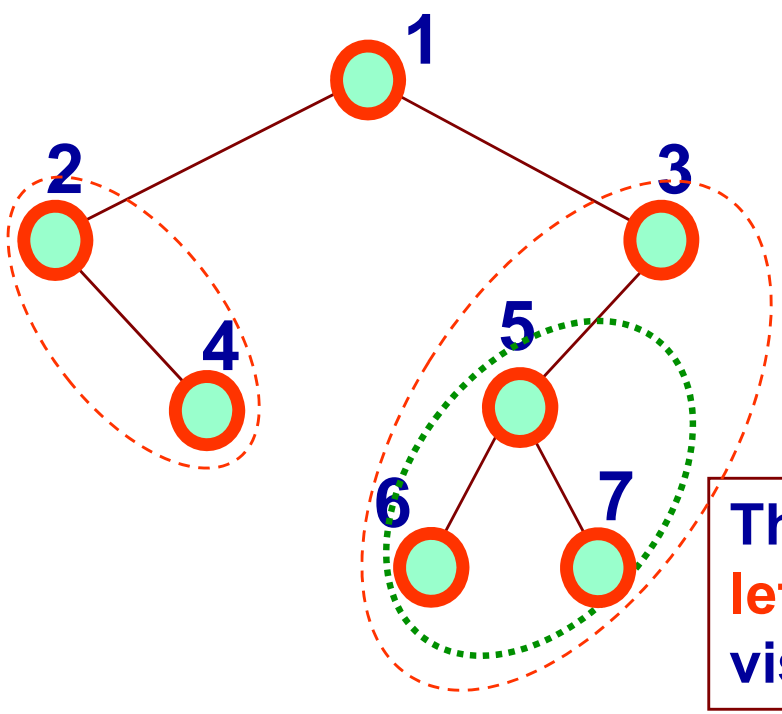
1. If *root* is empty, stop.
2. Visit *root.*
3. Execute *preorder* on the binary tree rooted at the left child of the root.
4. Execute *preorder* on the binary tree rooted at the right child of the root.



✓ 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

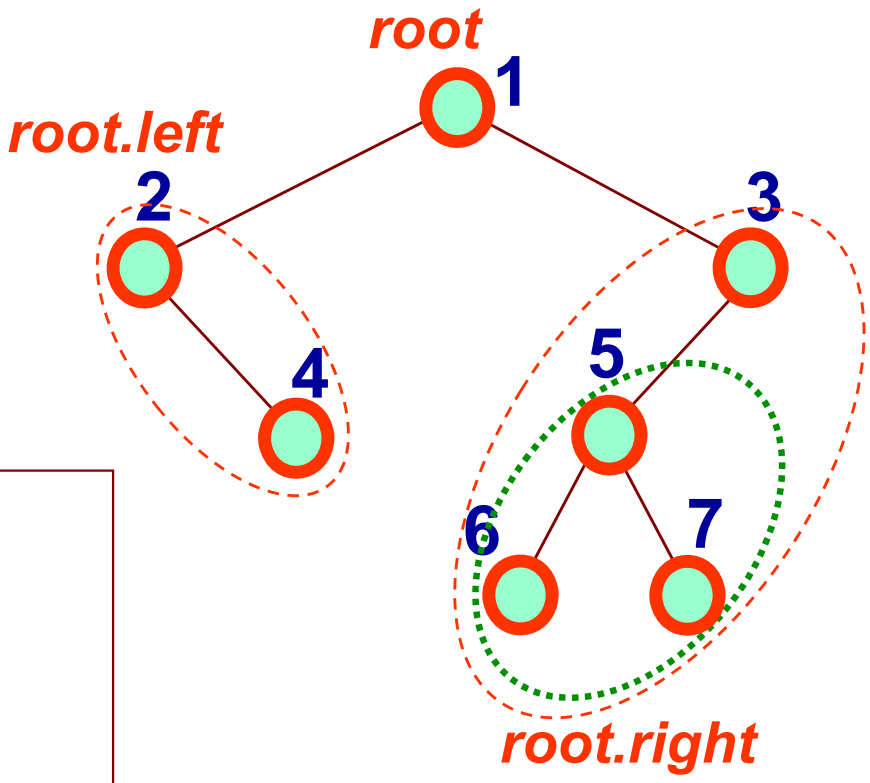
The order of visitation for this subtree is 3, 5, 6, 7.

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*

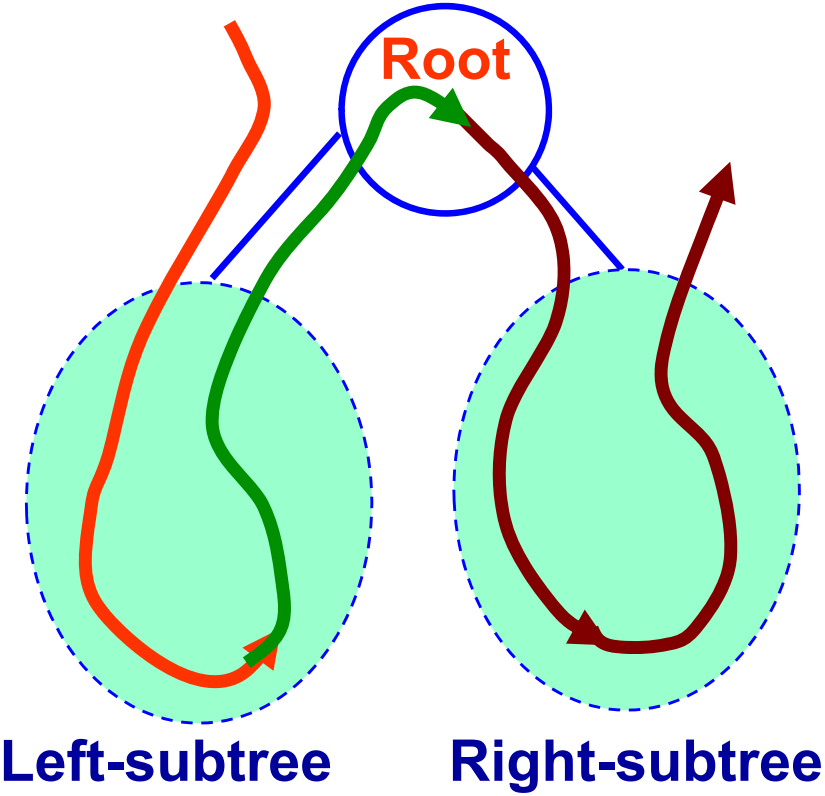
```
Count_nodes (root) {
    if (root== null)
        return 0
    count = 1    // count root
    // add in nodes in left subtree
    count = count + count_nodes(root.left)
    // add in nodes in right subtree
    count = count + count_nodes(root.right)
    return count
}
```

```
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.

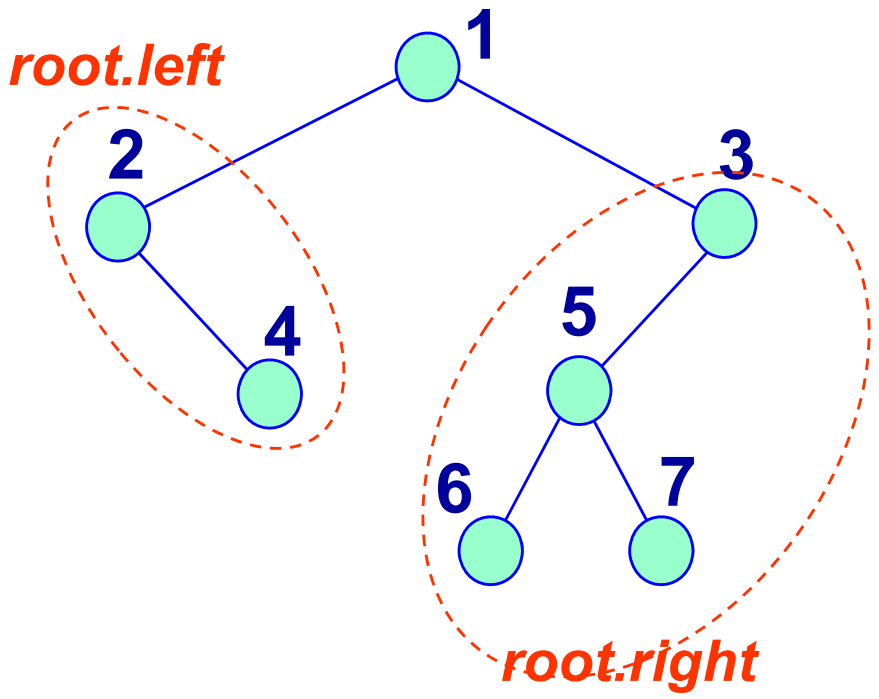


✓ In short, left—root— right

Algorithm Inorder

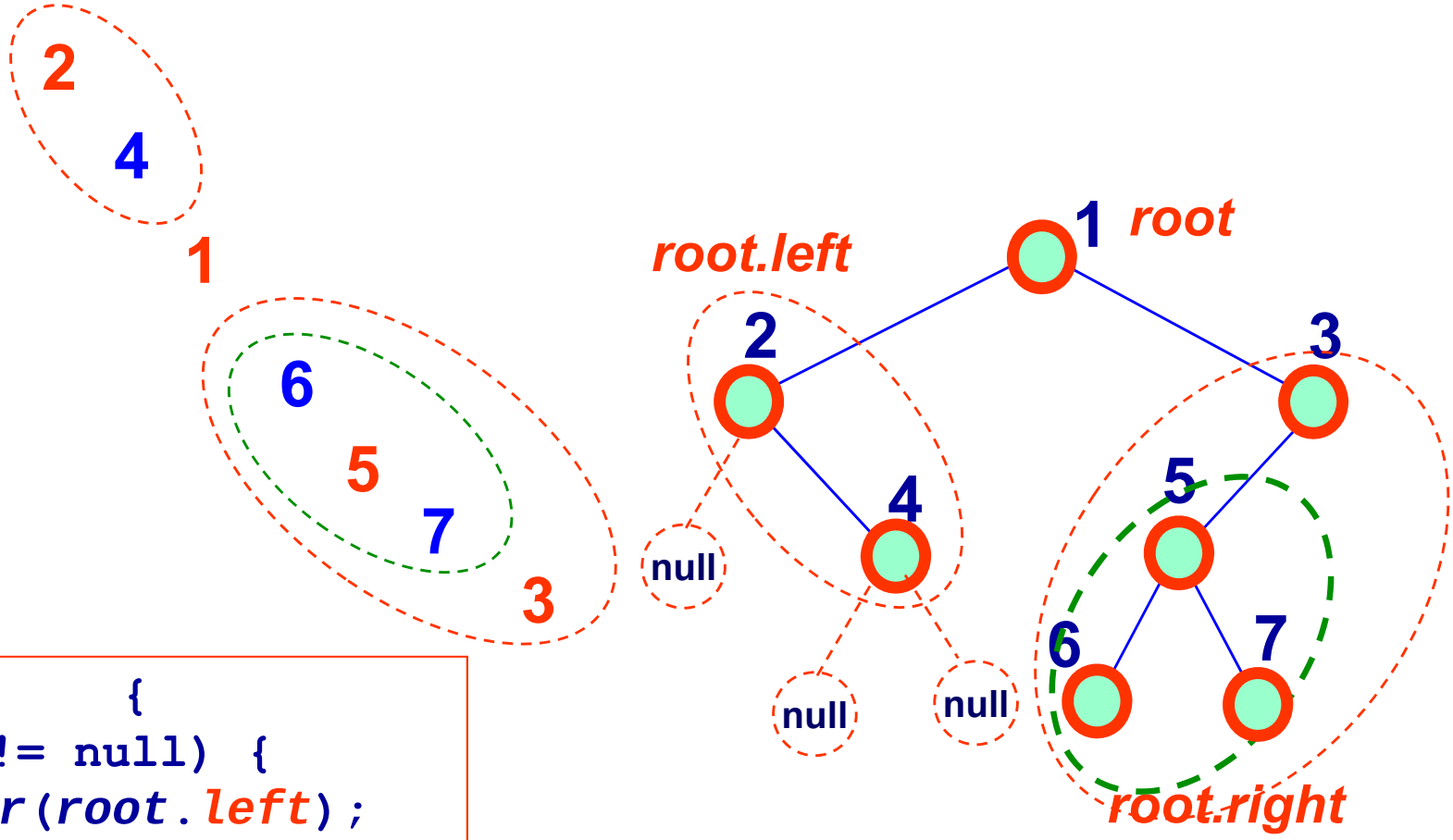
- This algorithm performs an inorder traversal of the binary tree with root *root*.

```
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

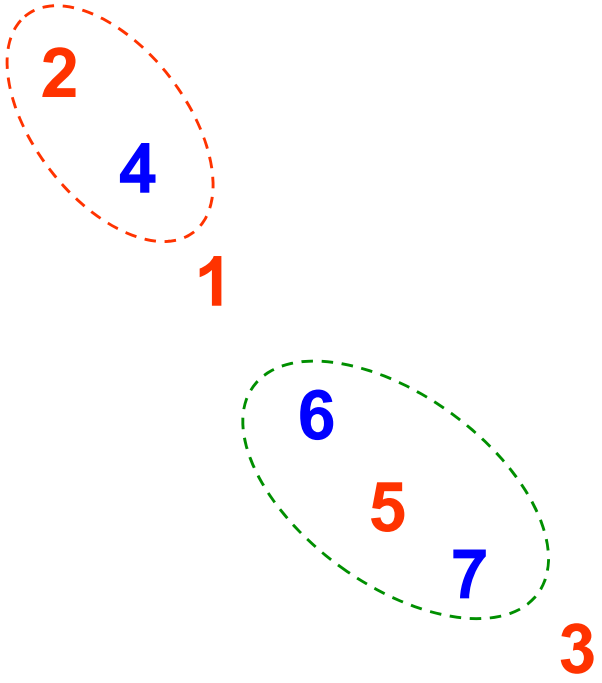


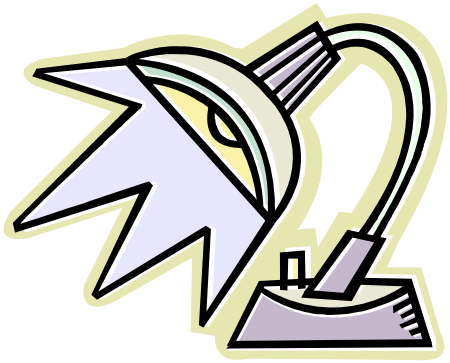

```
inorder(root)     {
    if (root != null) {
        inorder(root.left);
        visit root;
        inorder(root.right);
    }
}
```

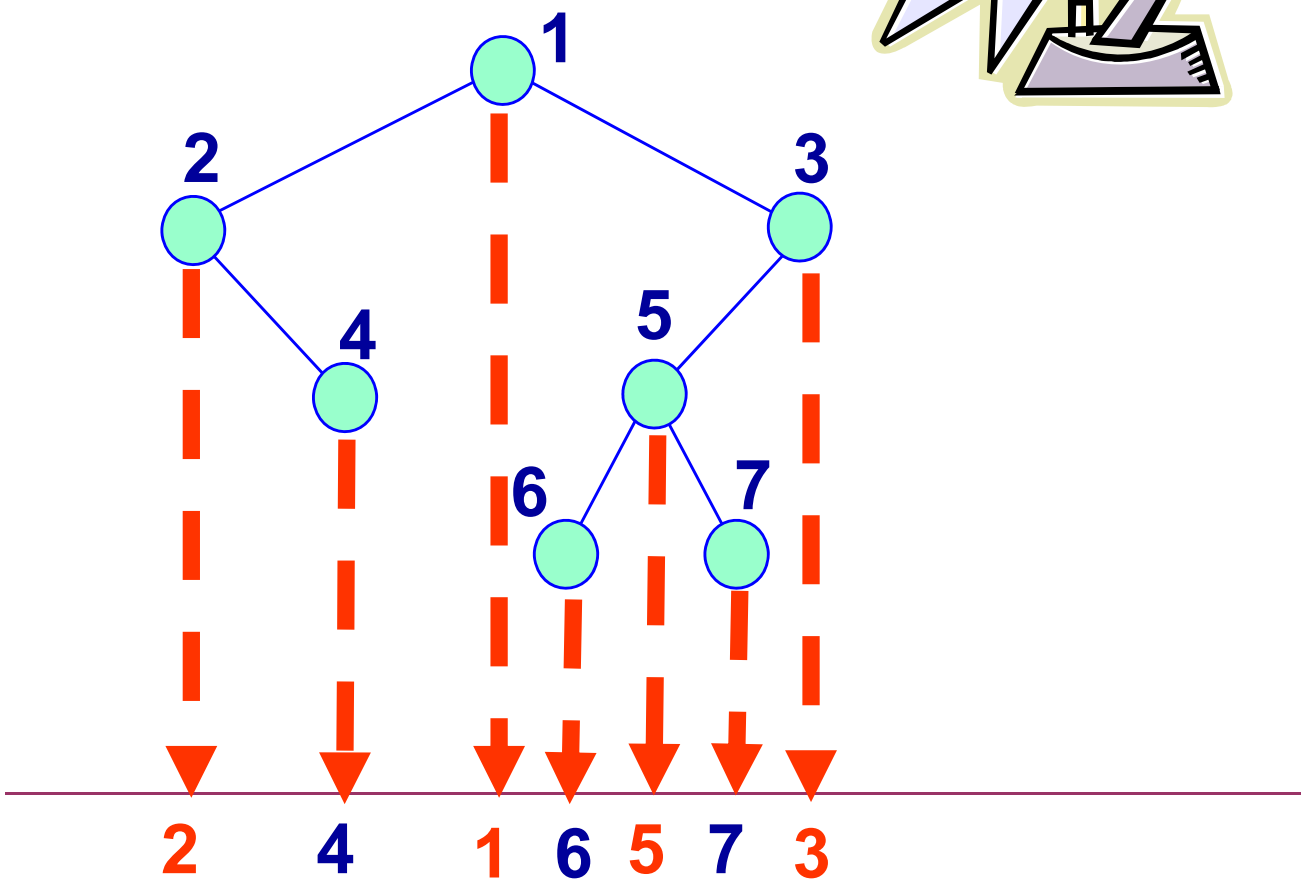
✓ In order of left—root— right

Inorder Traversal By Projection (Squishing)

- An easy way …





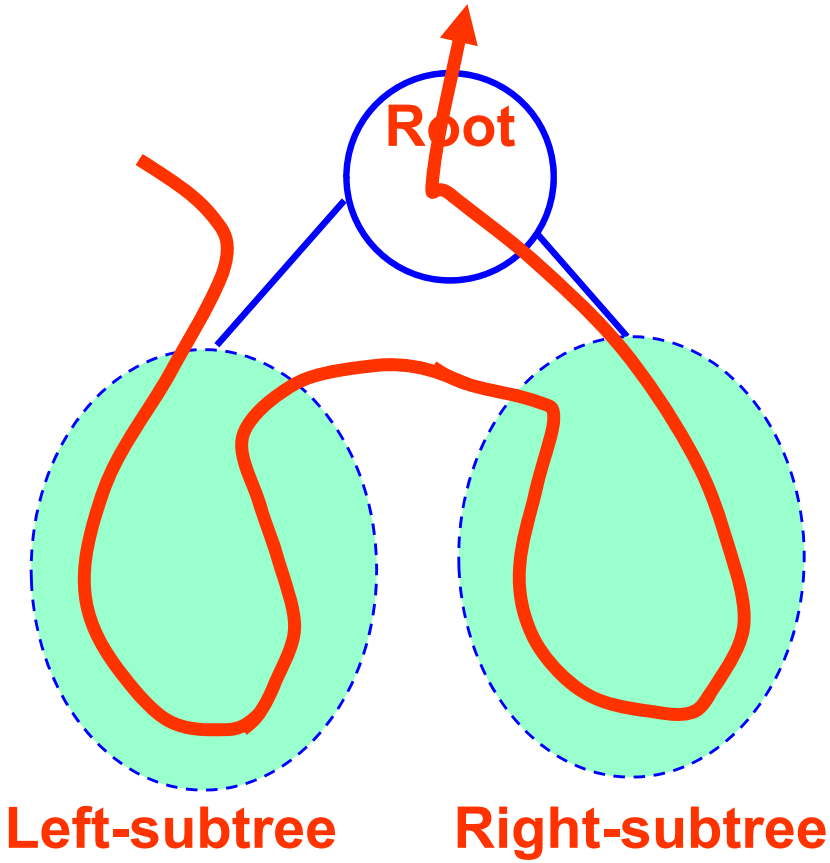


Inorder traversal:

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*.

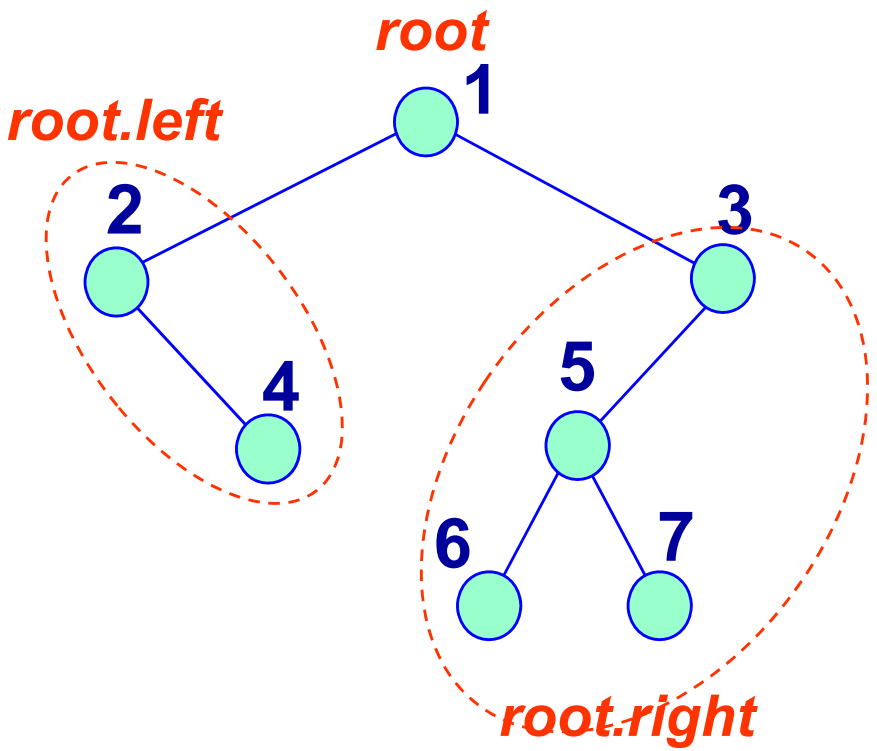


✓ In short, left—right—root

Algorithm Postorder

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

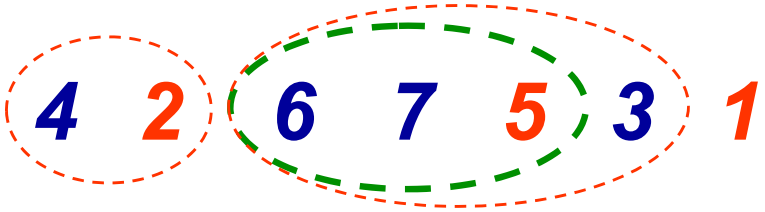
```
postorder(root)     {
    if (root != null) {
        postorder(root.left);
        postorder(root.right);
        visit root;
    }
}
```

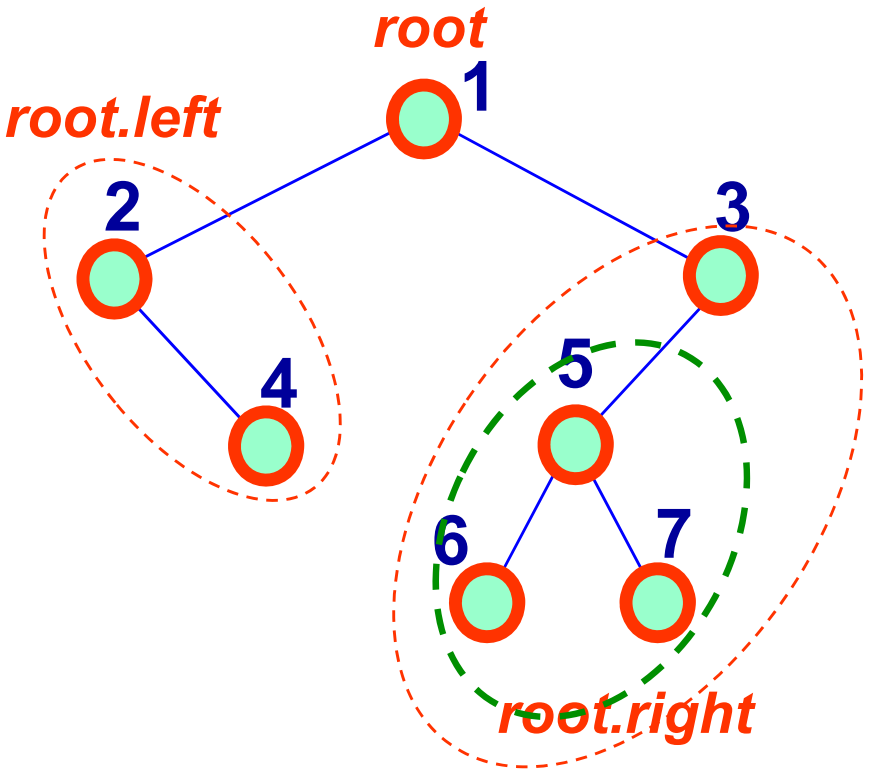


✓ In short, left—right—root

Postorder Traversal: Example

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

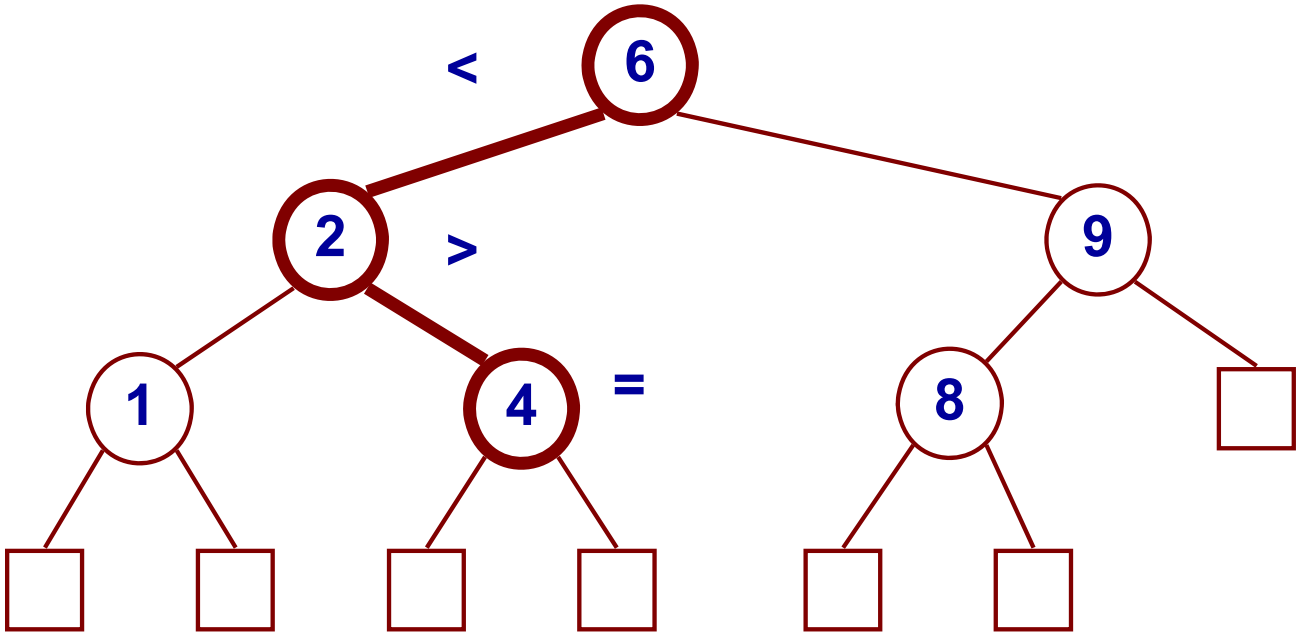





```
postorder(root)     {
    if (root != null) {
        postorder(root.left);
        postorder(root.right);
        visit root;
    }
}
```

✓ In short, left—right—root

Binary Search Trees



What is a Binary Search Tree

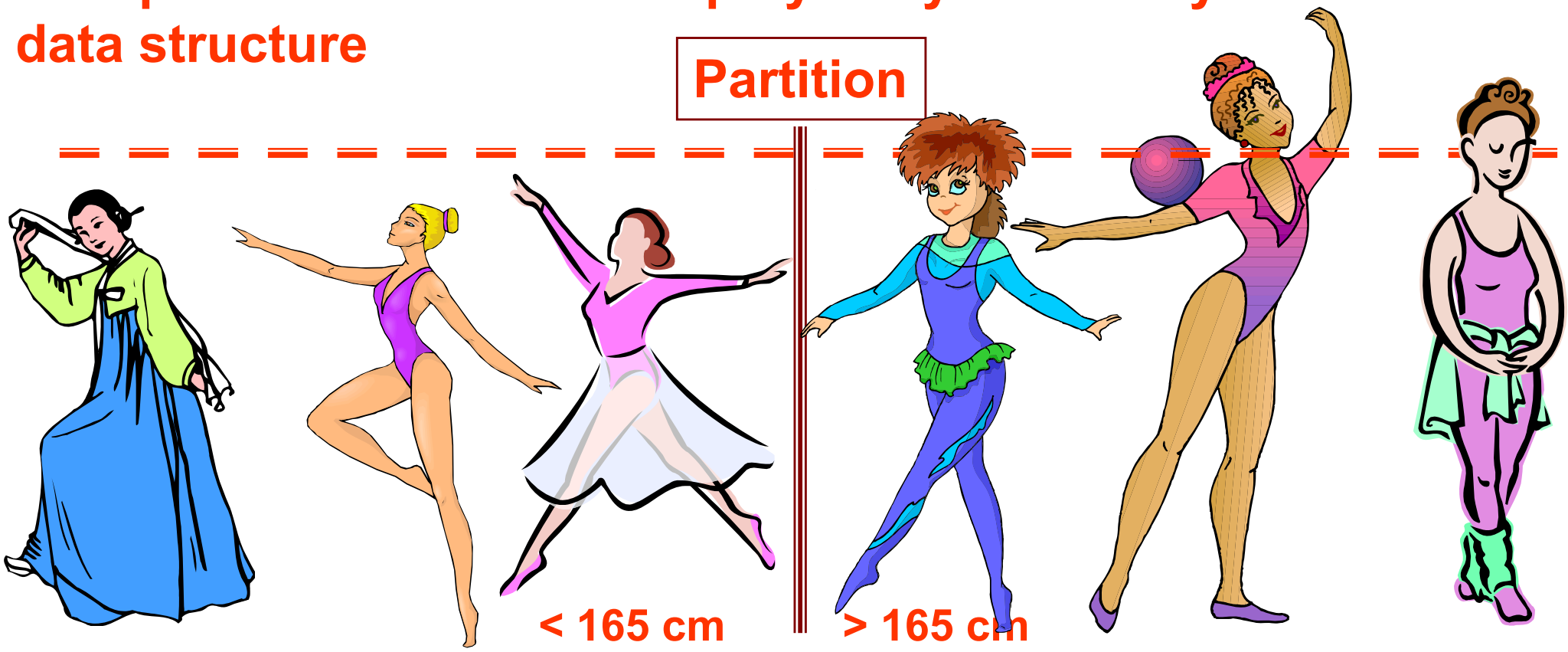
Suppose that you have to sort a group of people by height so that you can easily search for someone by their height later on.

How would you go about doing this efficiently?

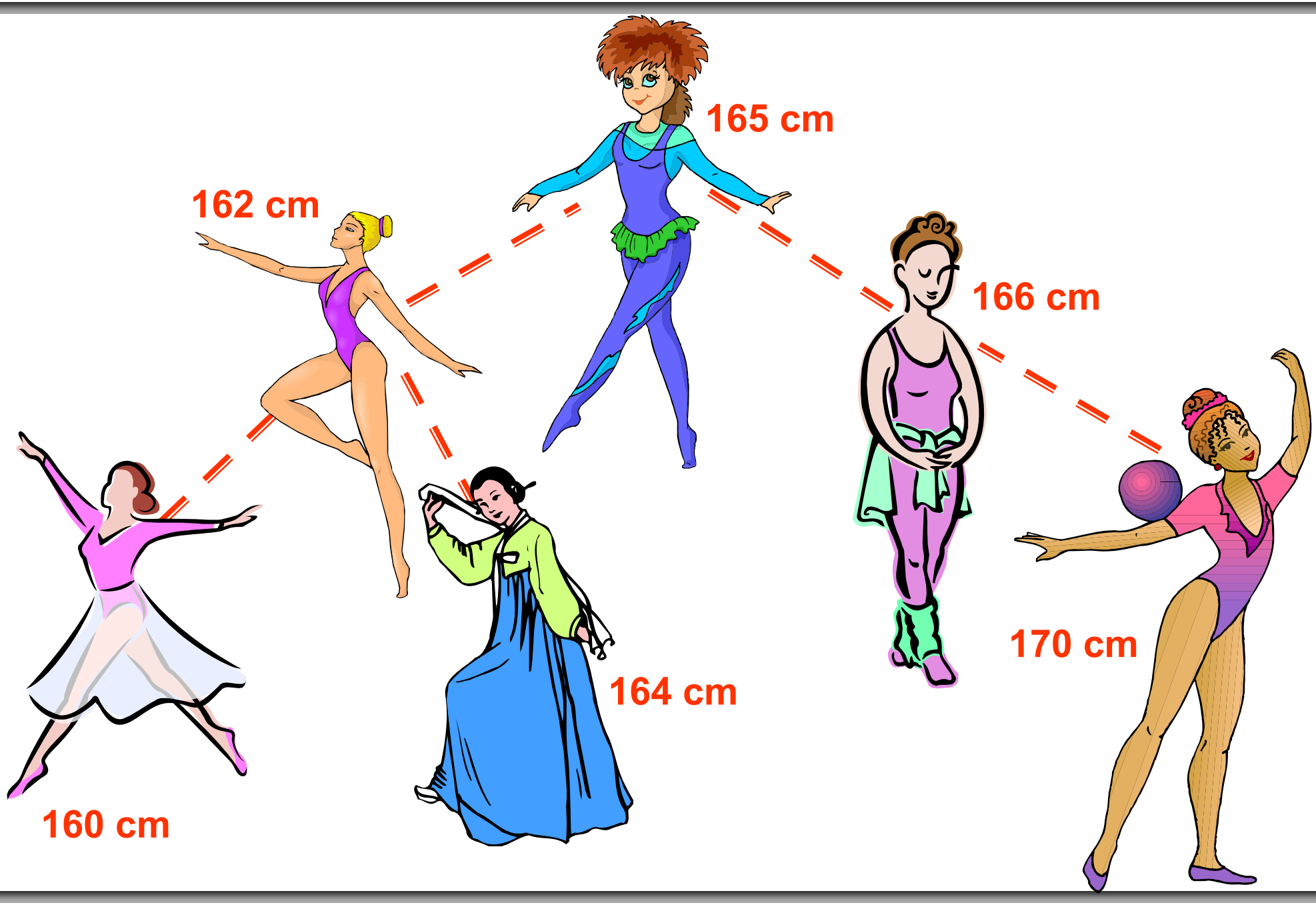
What is a Binary Search Tree

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

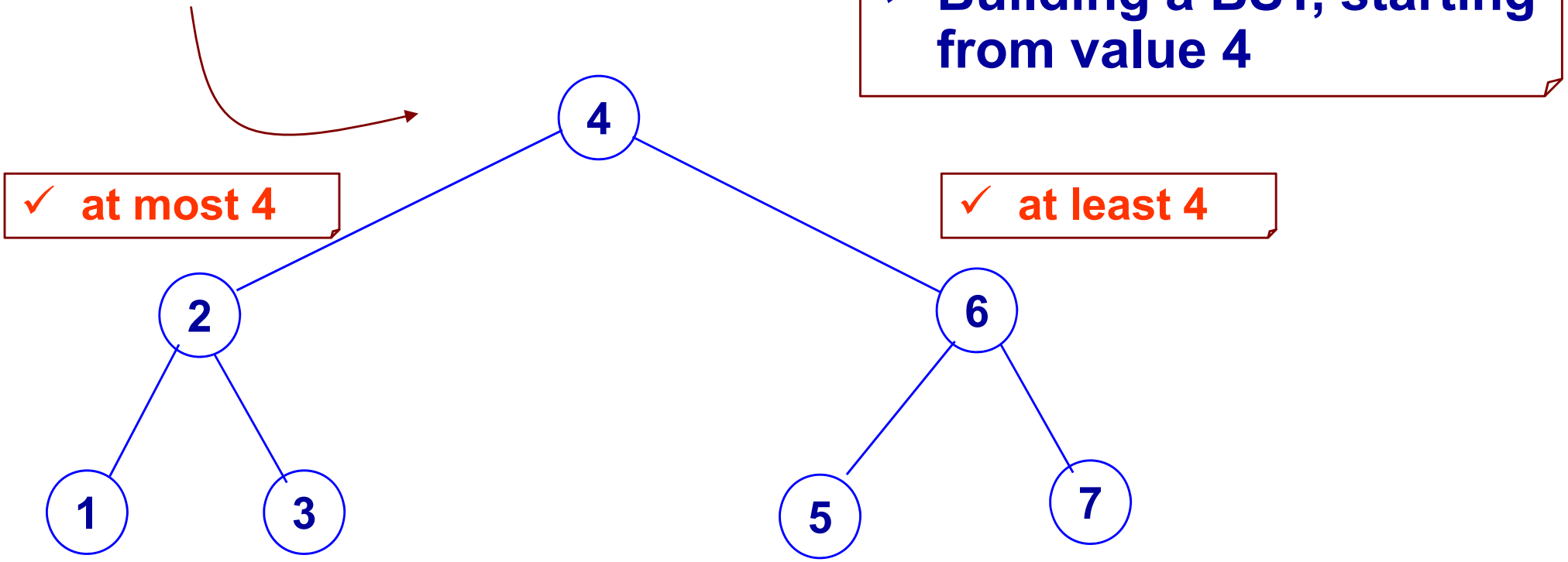
What is a Binary Search Tree

- 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



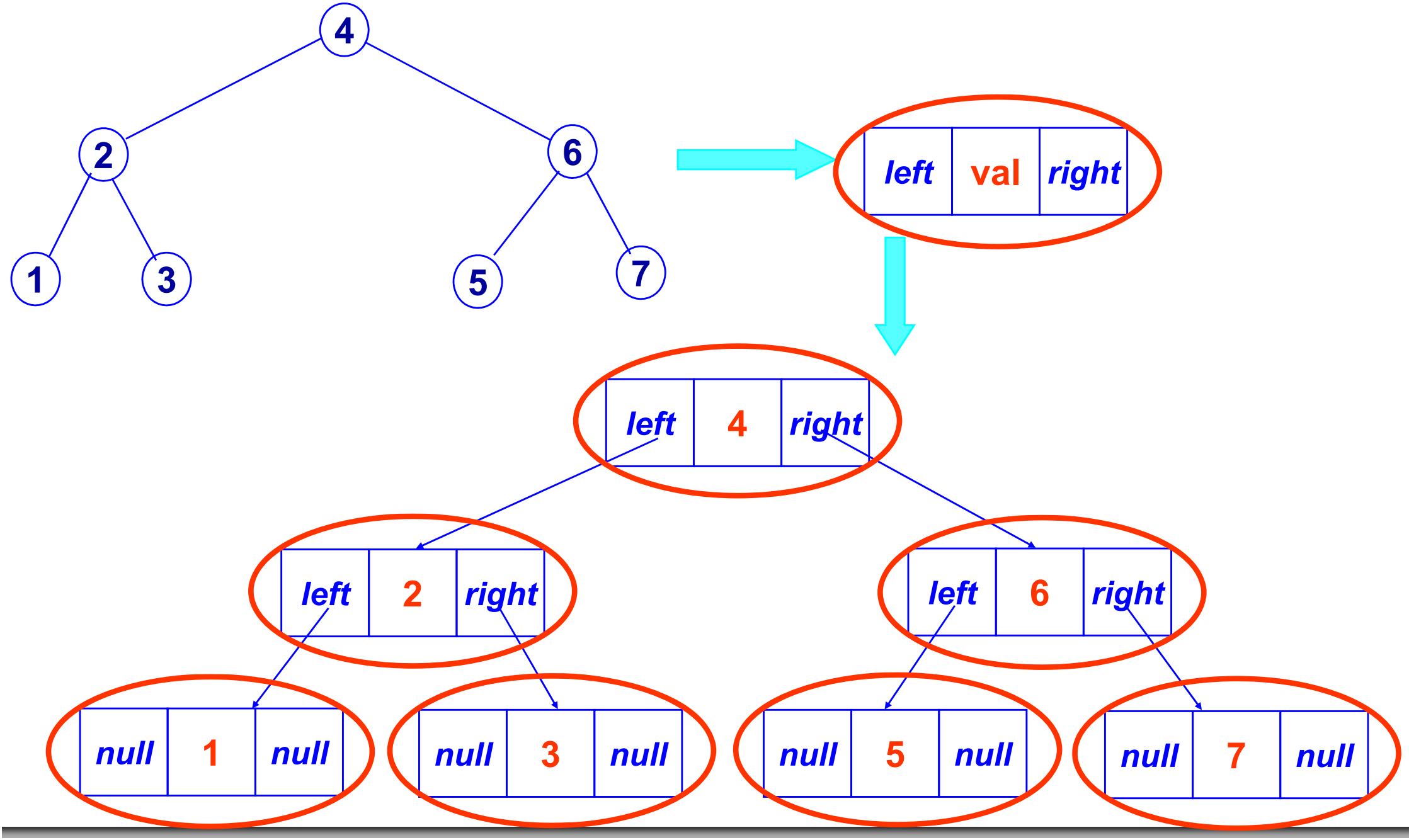
- ✓ Building a BST, starting from value 4
- ✓ at most 4
- ✓ at least 4
- ✓ "Binary search tree property": a node's left child must have a data less than or equal to its parent, and a node's right child must have a data greater than its parent

Insertion and deletion

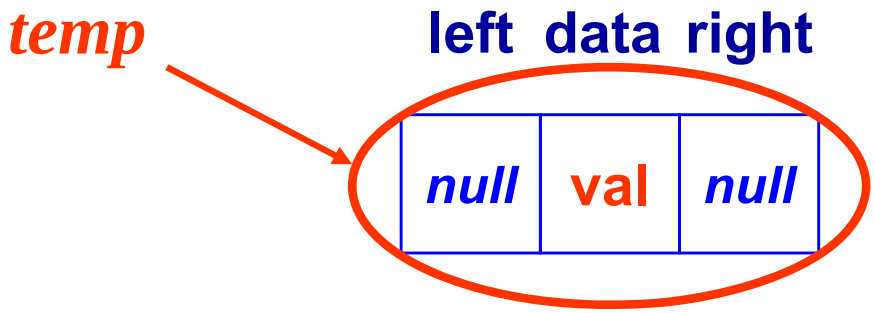
- **The operations of insertion and deletion cause a binary search tree to change**
- **This change is done in such a way that the binary-search-tree property continues to hold.**

BST Insertion

Node Representation



Set up a node to be inserted to tree

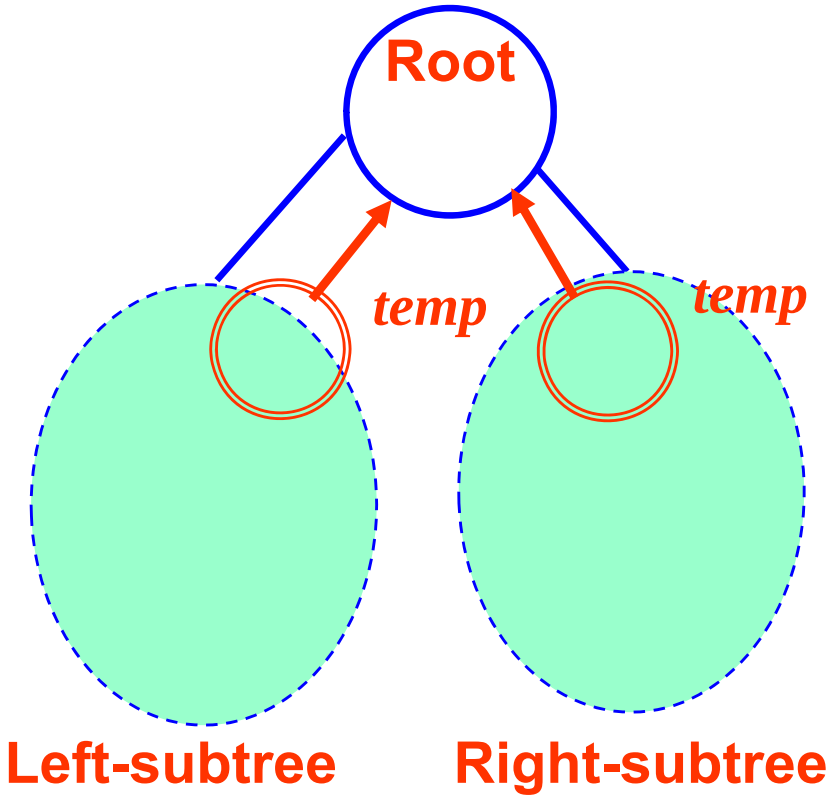



```
// set up node that contains value val to be added to tree

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***

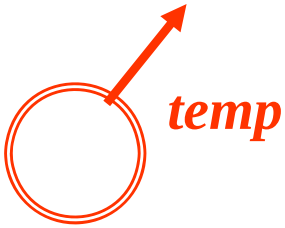
Case 3: ***Insert to a non-empty tree, right subtree empty***

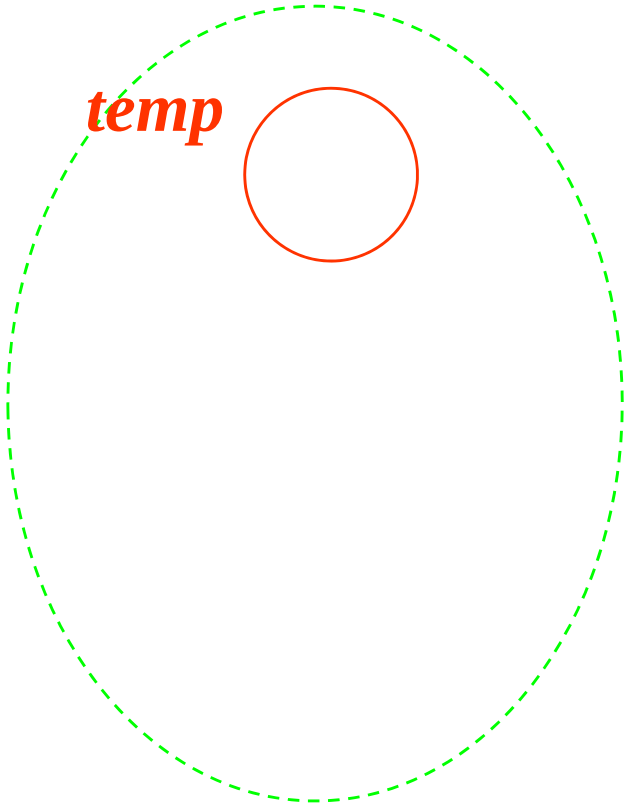
Case 4: ***Insert to a non-empty tree, left subtree non-empty***

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



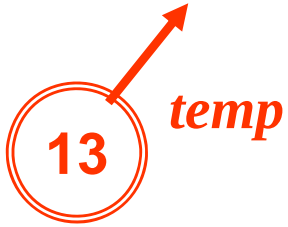



```
BSTinsert (root, val) {
    // set up node to be added to tree
    temp = new node;
    temp.data = val
    temp.left = temp.right = null

    // 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*

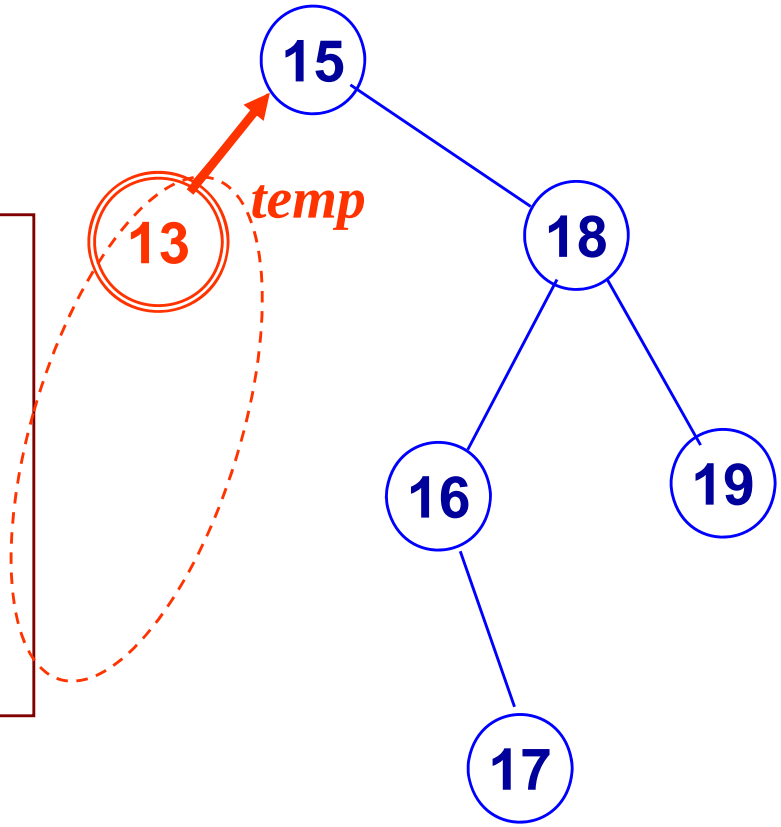
Case 2: Insert to a non-empty tree (into left subtree)

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;
  }

}
```

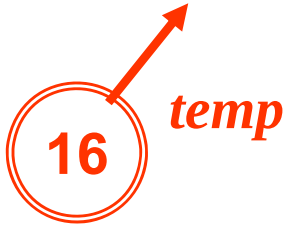


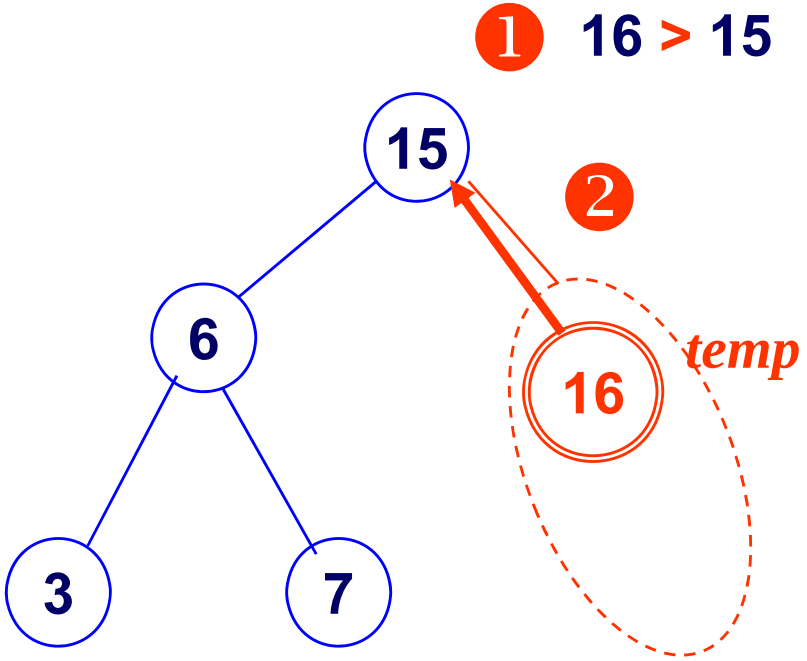
Case 1: *Insert a node to an empty tree*
Case 2: *Insert to a non-empty tree, left subtree empty*
Case 3: *Insert to a non-empty tree, righ subtree empty*
Case 4: *Insert to a non-empty tree, left subtree non-empty*
Case 5: *Insert to a non-empty tree, right subtree non-empty*

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



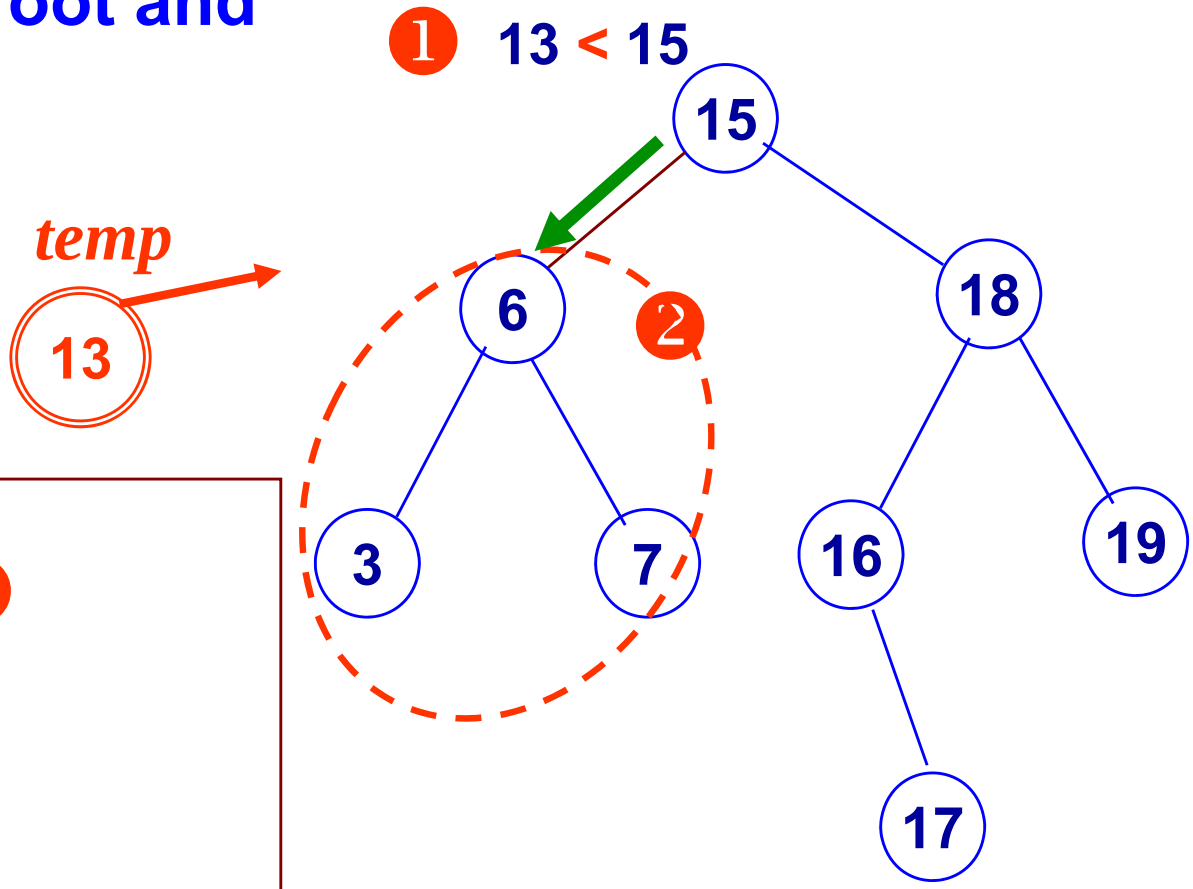



```
BSTinsert_recurs (root, temp) {
  if (temp.data ≤ root.data) {  1
    // insert at left
  }
  else {     // goes to right

    if (root.right = = null)    2
      root.right = temp;        3

  }
}
```

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

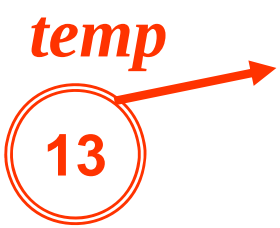


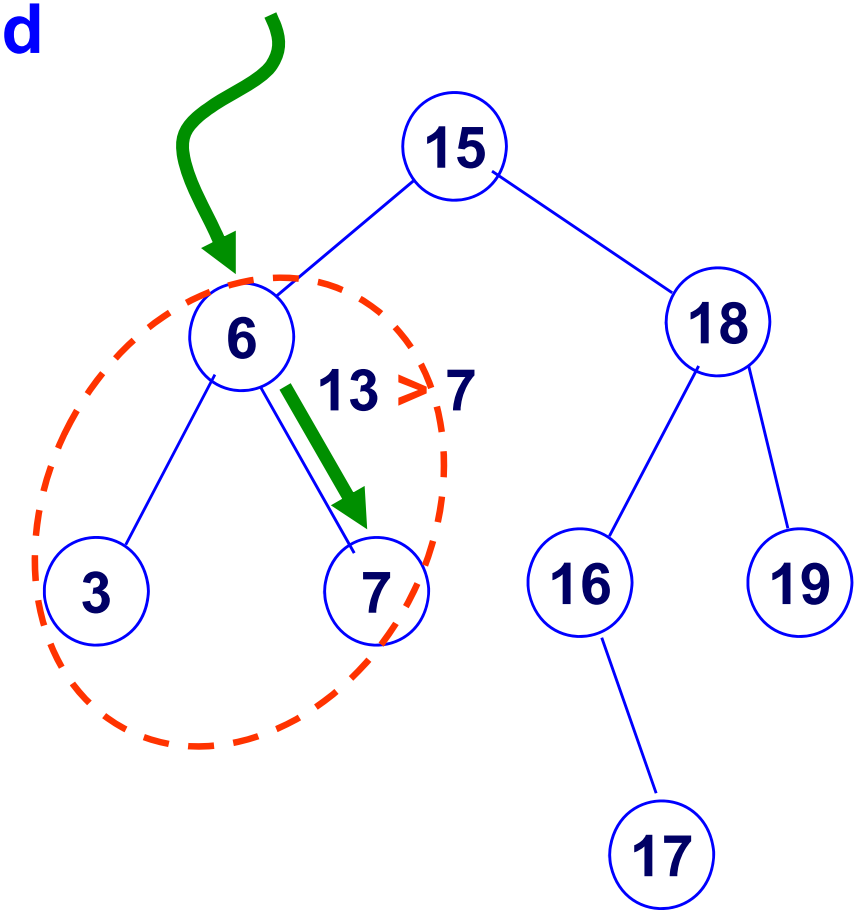

```
BSTinsert_recurs (root, temp) {
  if (temp.data ≤ root.data) {        1
    if (root.left = = null)
      root.left = temp
    else    // non-empty              2
      BSTinsert_recurs (root.left, temp);
  }
  else {     // goes to right
  }
}
```

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

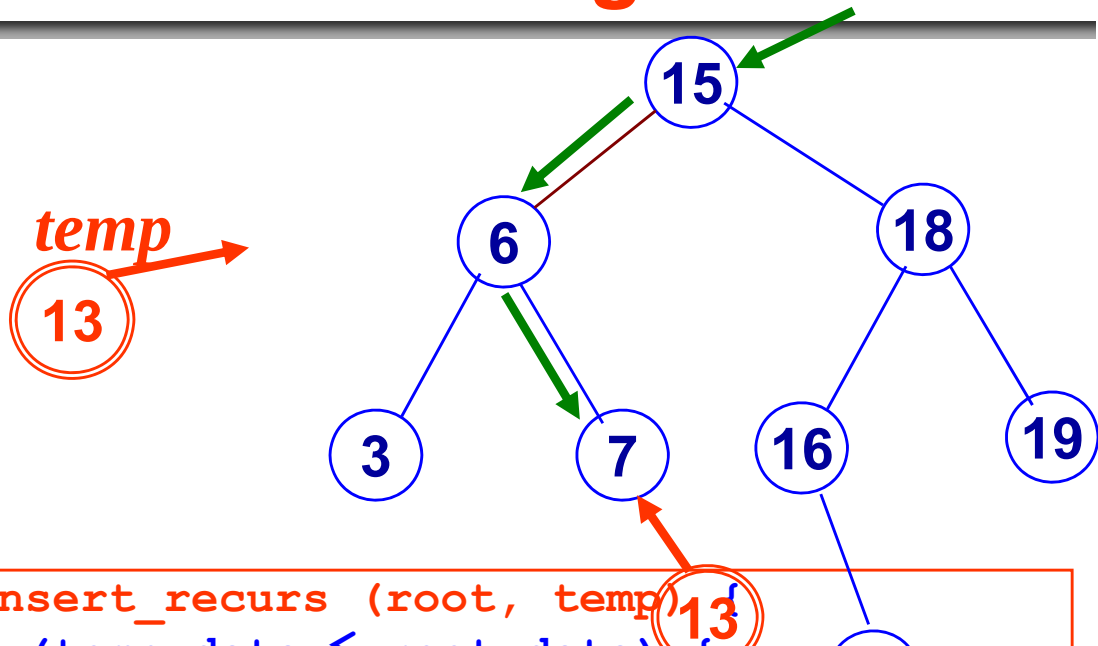
Write a general algorithm that inserts a node into a binary search tree (BST).

Hint:

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

BST Insertion Algorithm: Put all together

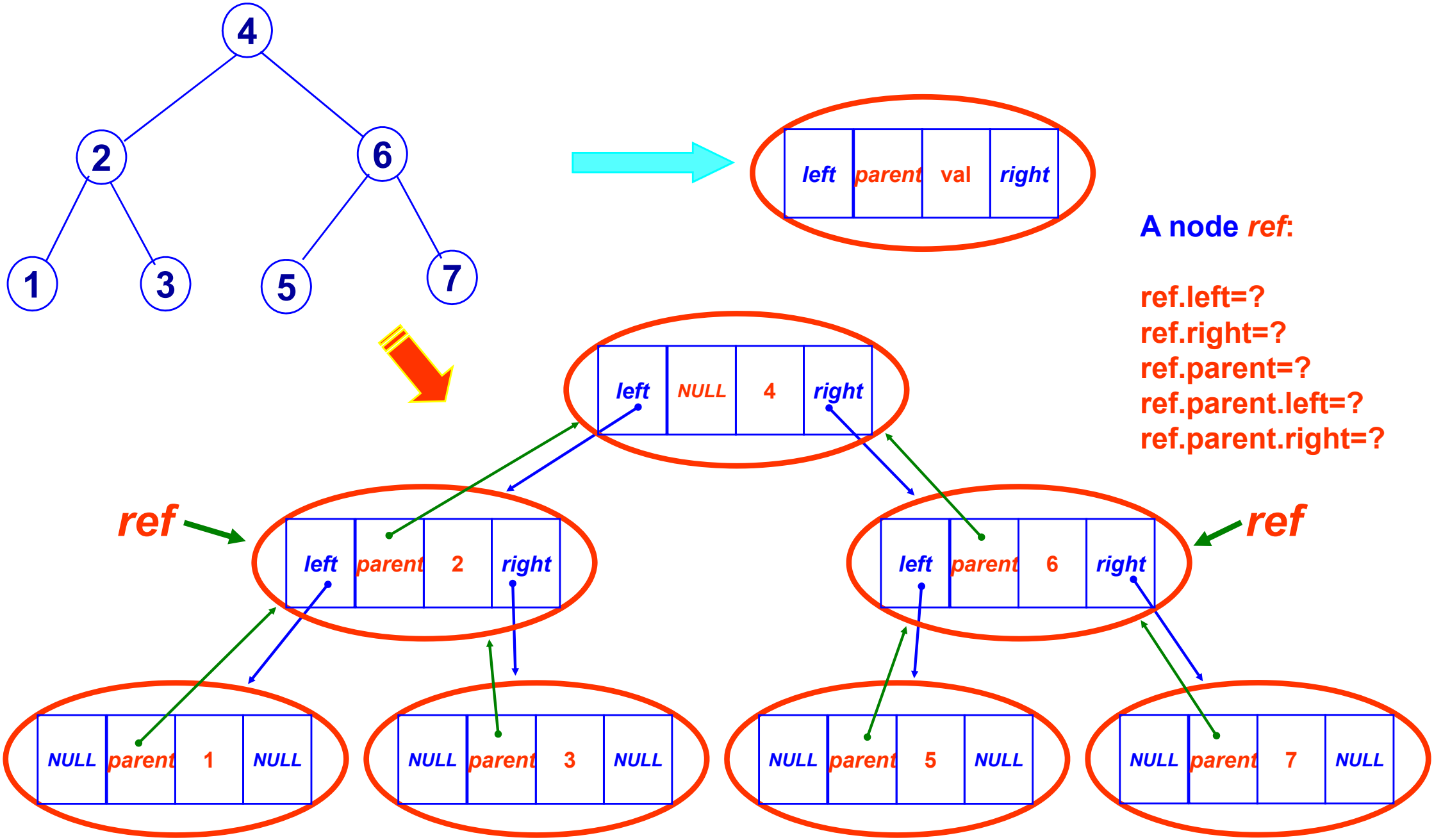
```
BSTinsert (root, val) {
    // set up node to be added to tree
    temp = new node;
    temp.data = val
    temp.left = temp.right = null

    // 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

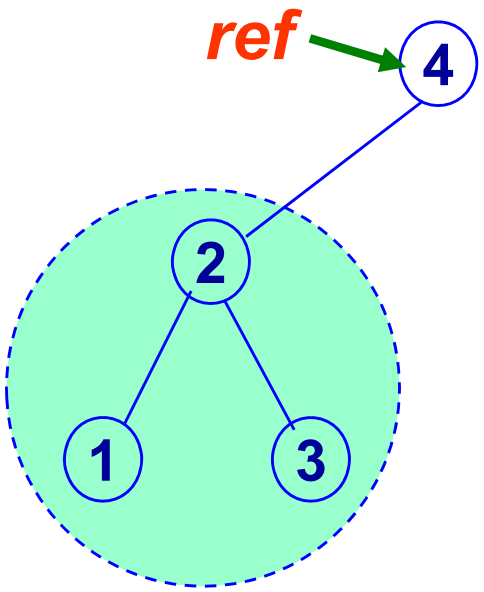
Node Representation

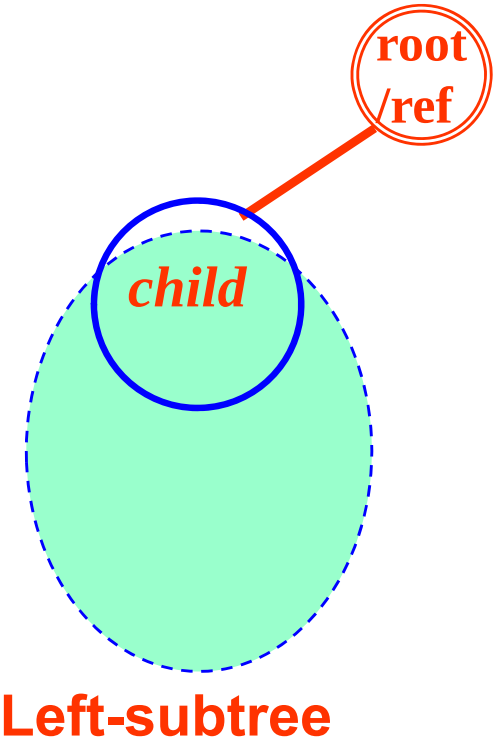


BST Deletion

- *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.

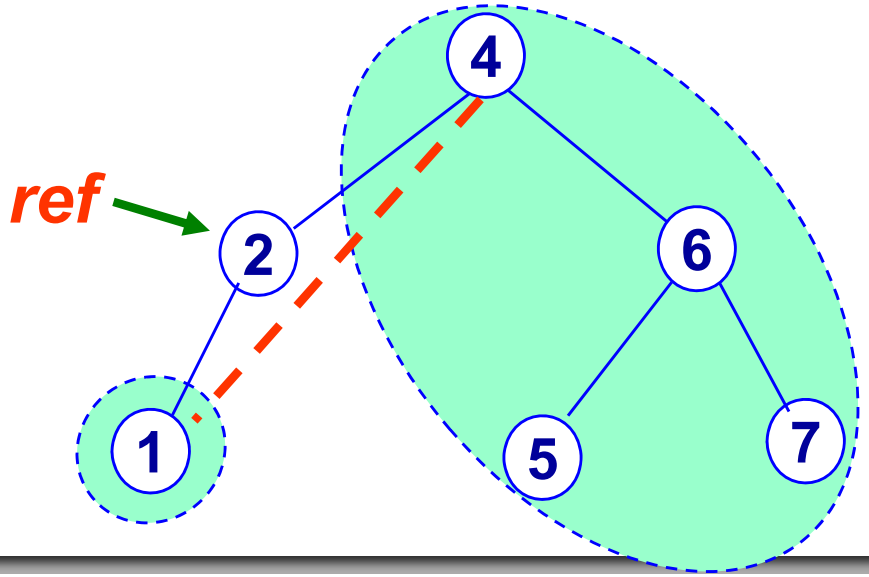


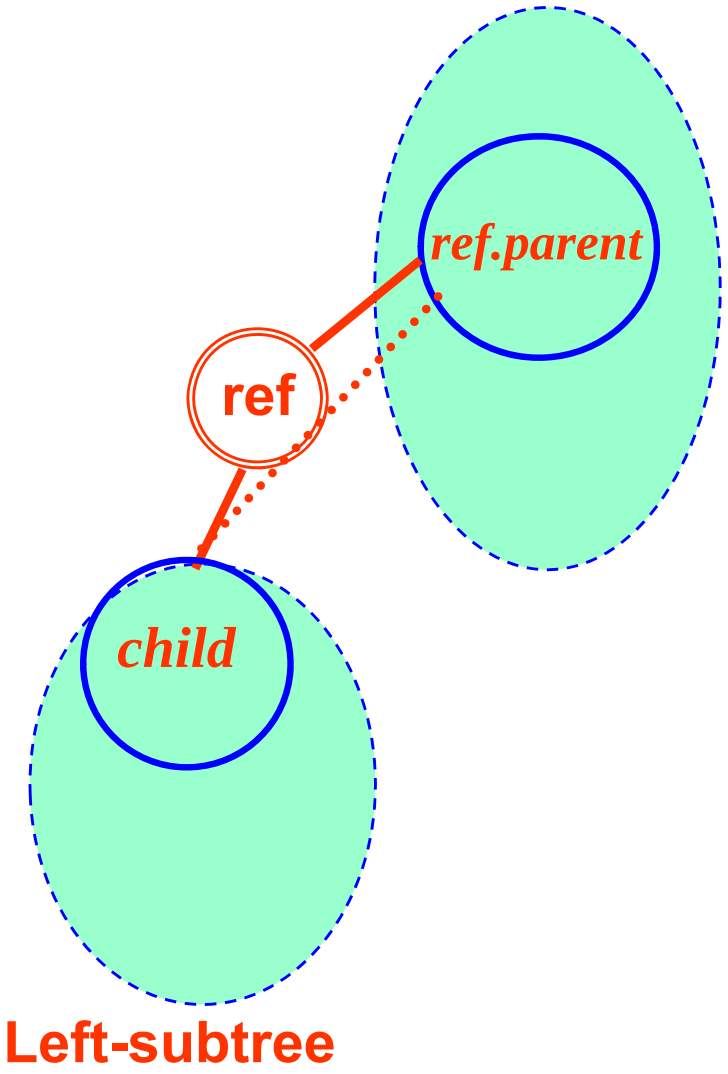


BST Deletion

- *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.

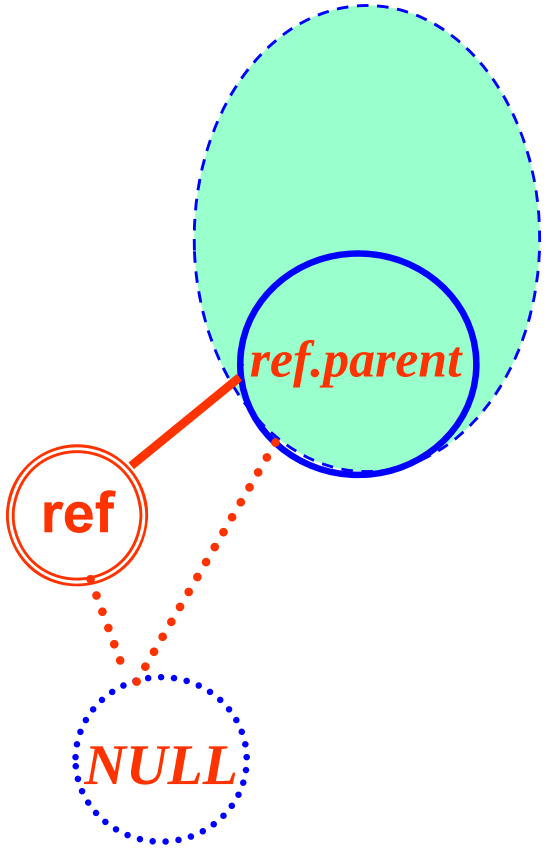


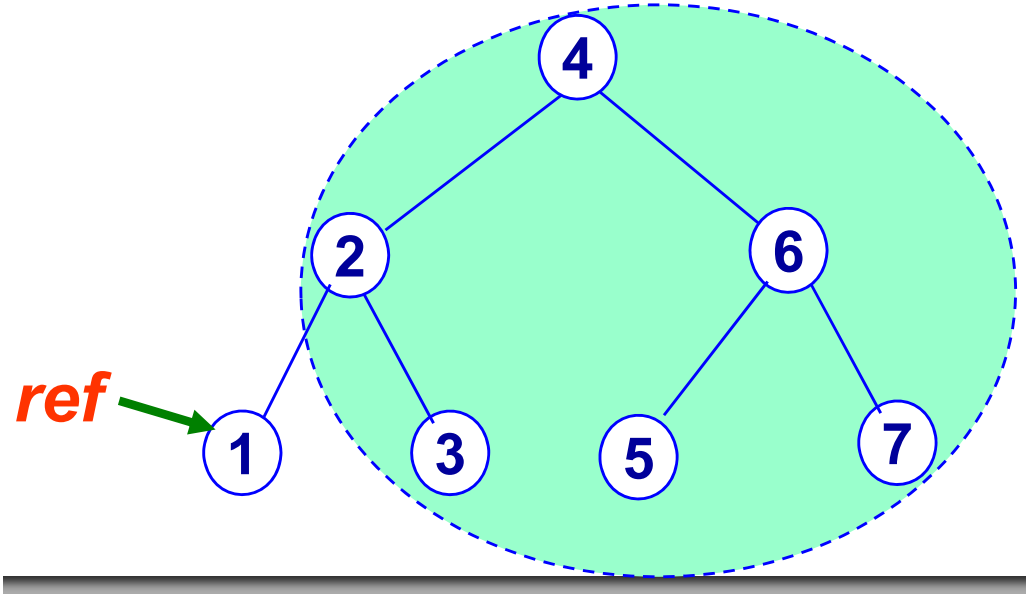


BST Deletion

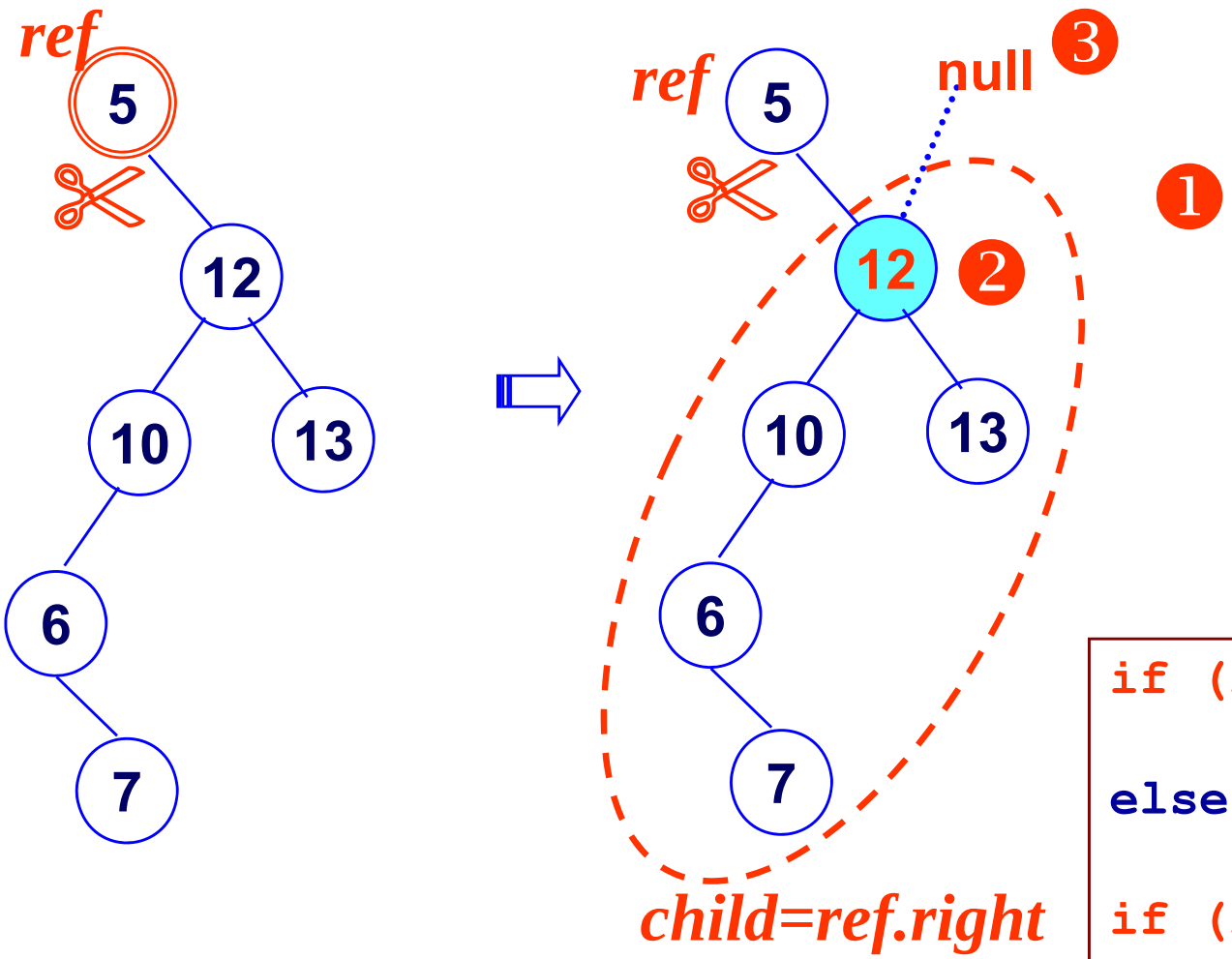
- *BSTreplace(root, ref)*: (for the case *ref* has one or no child)

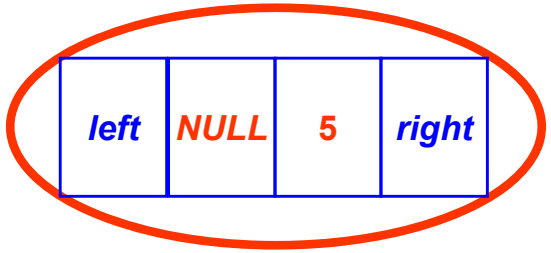
1. If *ref* is *root*, set the child of *ref* as the root, stop.
2. If *ref* is neither *root* nor *leaf*, set the child of *ref* as *ref.parent*'s child, stop.
3. If *ref* is *leaf*, set *ref.parent*'s child NULL, stop.





Deletion Example: only one child (Delete root)



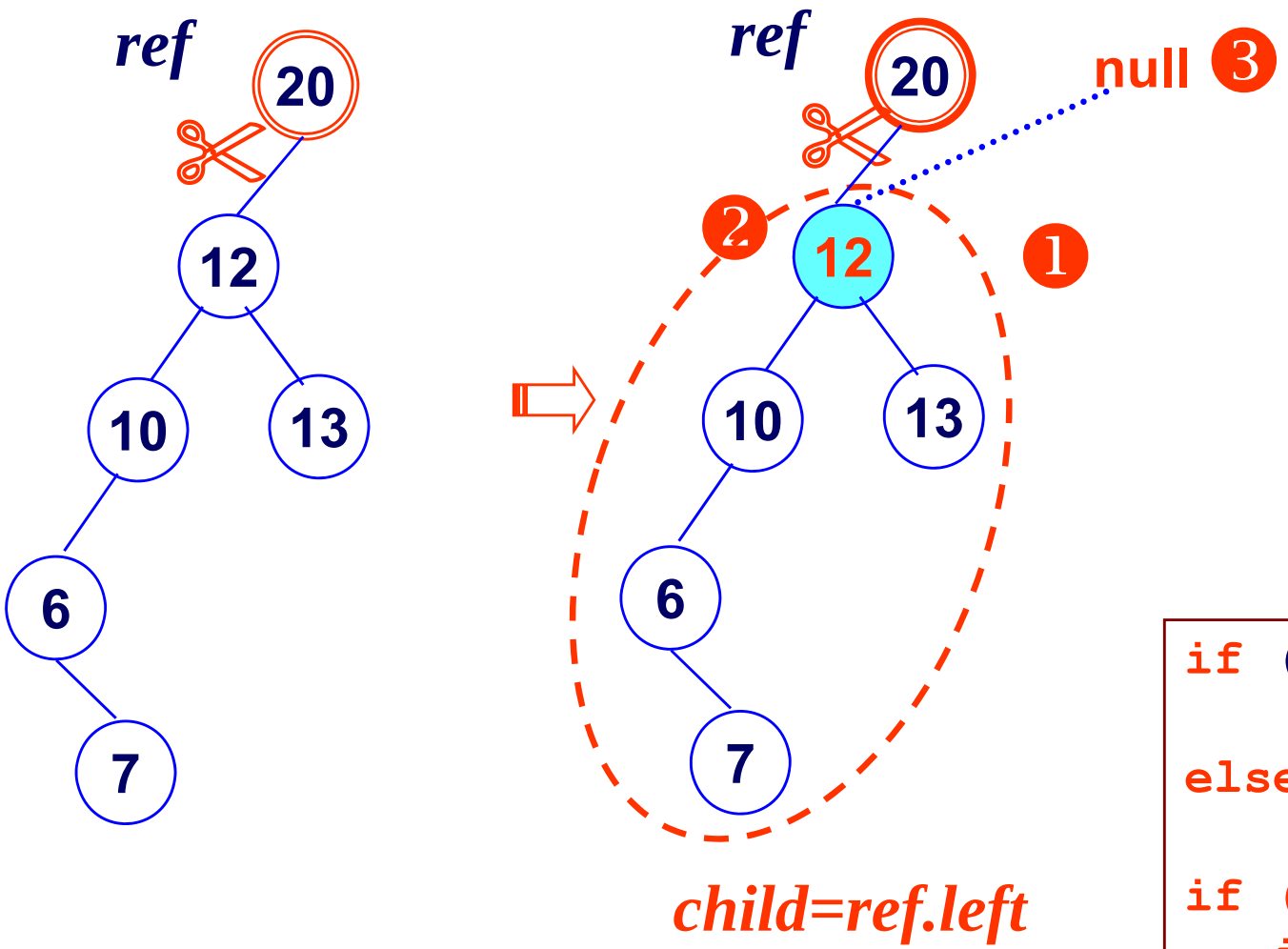


- ✓ First for the case of right child

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

```
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;
}
```

Deletion Example: only one child (Delete root)

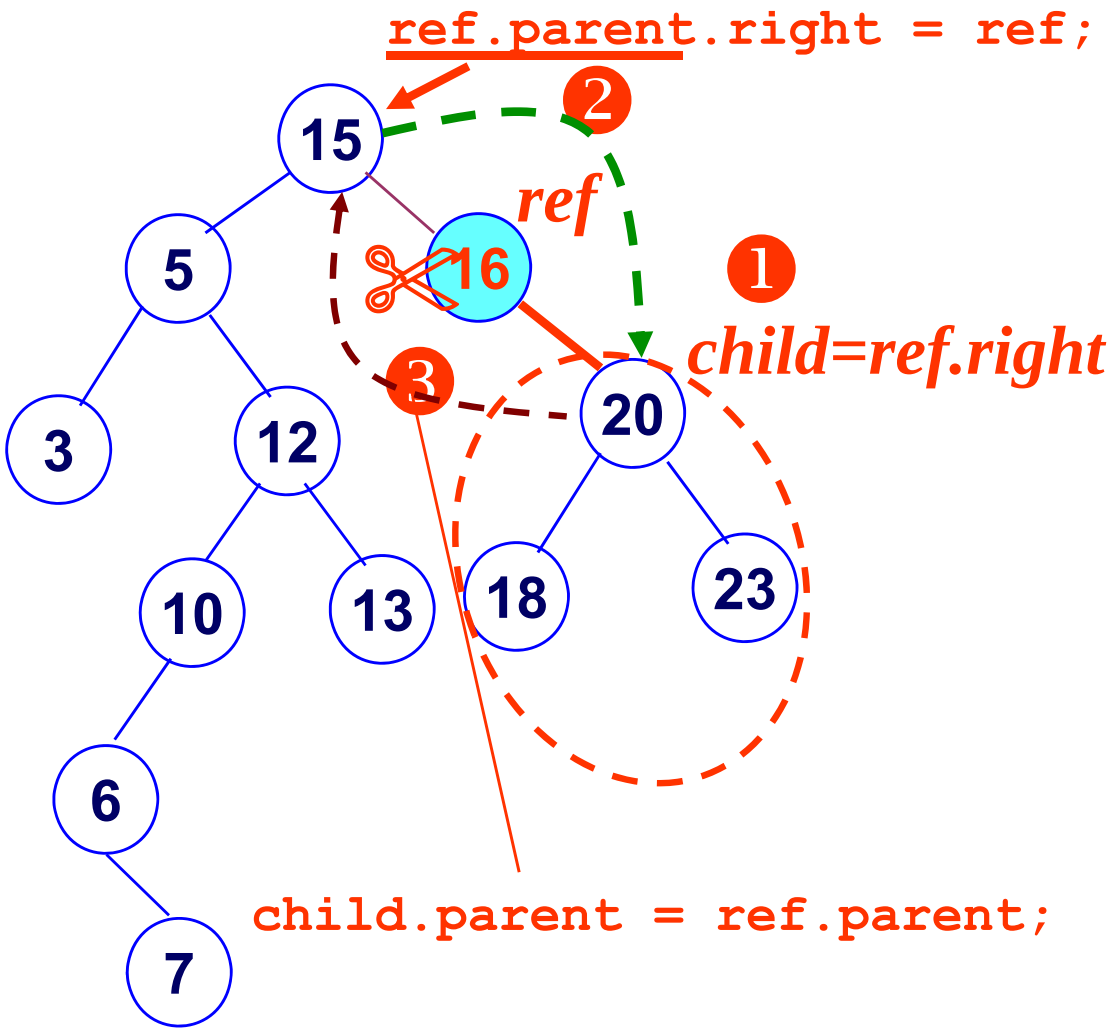


- ✓ Similarly for the case of left child

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

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

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

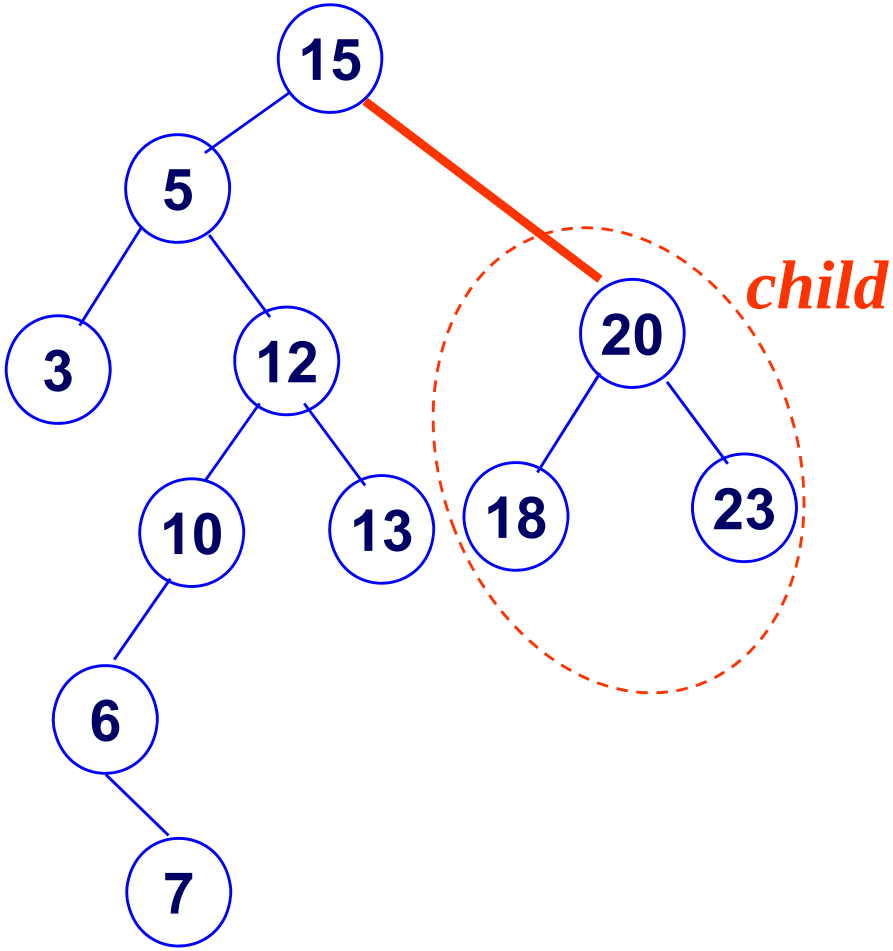
Deletion Example: only one child (not root)




```
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

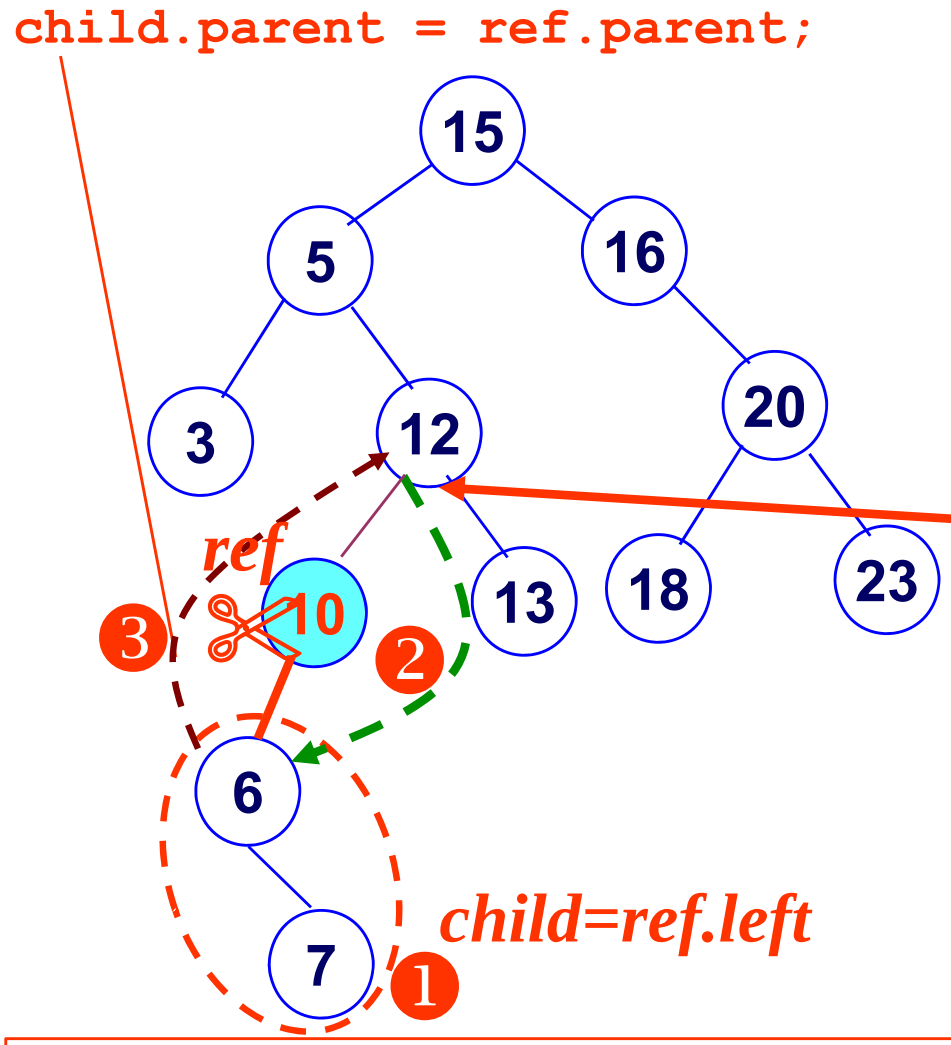
Deletion Example: only one child



✓ Tide up

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

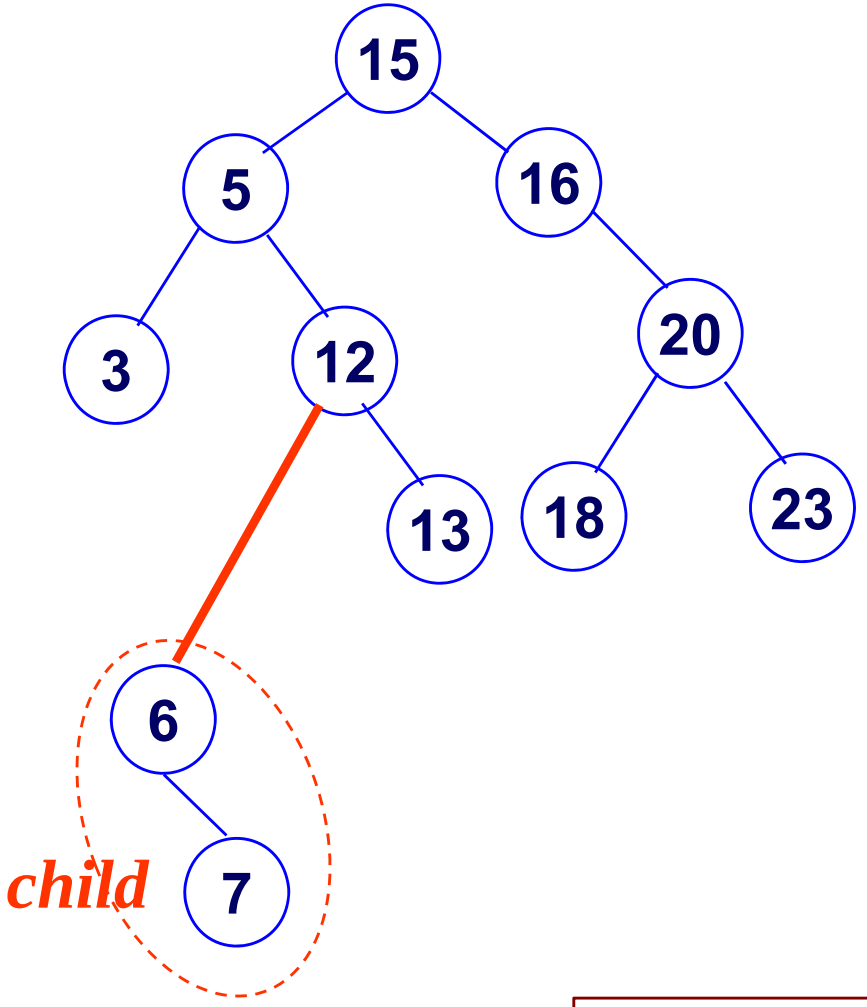
Deletion Example: only one child (not root)




```
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

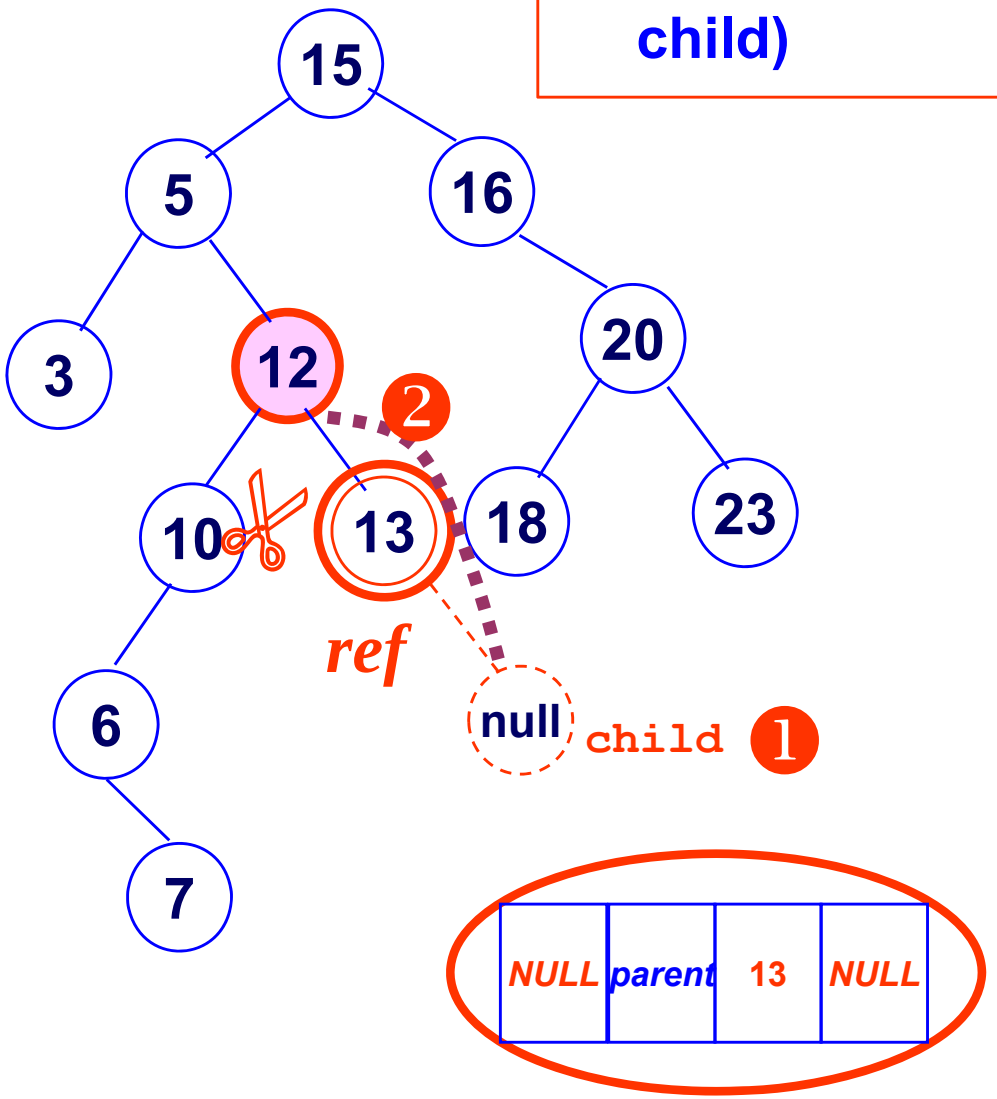


✓ Tide up

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

Deletion Example: no children (Delete leaf)

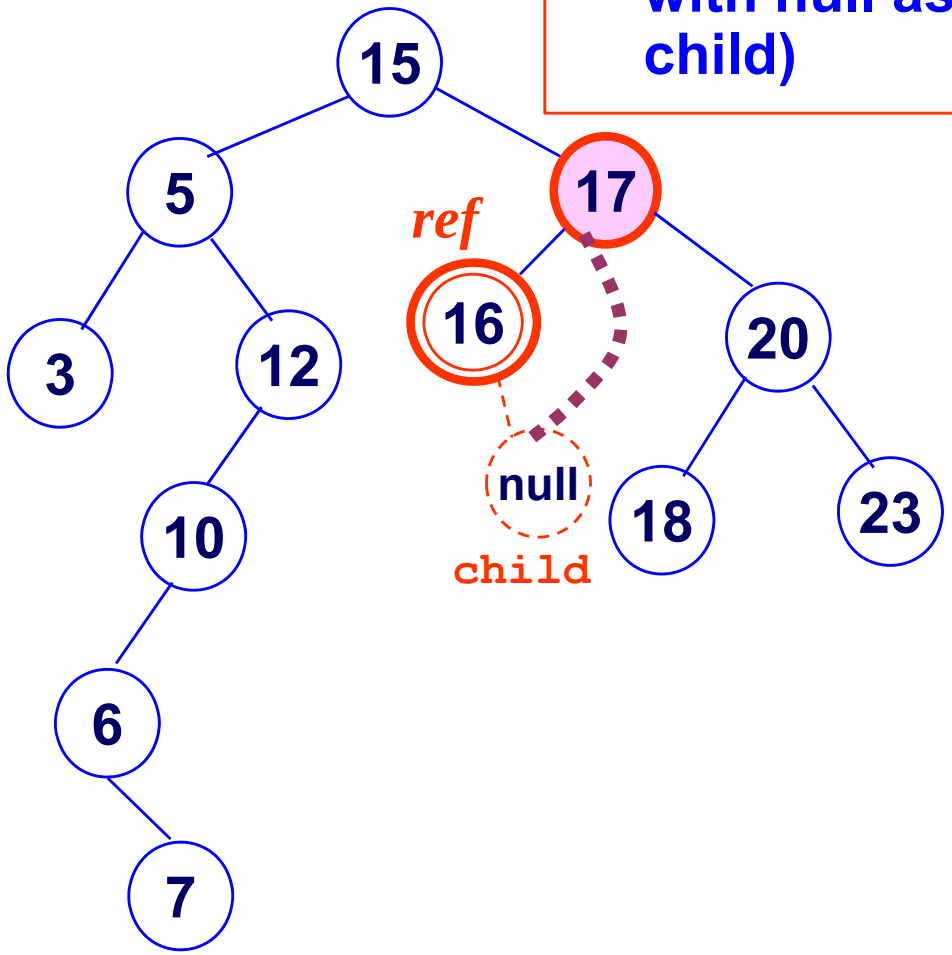
- ✓ If *ref* has no children, modify its parent ref.parent with null as its child (which means, *ref* has a null child)
- ✓ *ref* is the right child of its parent




```
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)

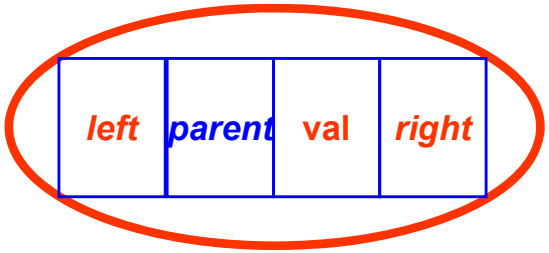


- ✓ *ref* is the left child of its parent

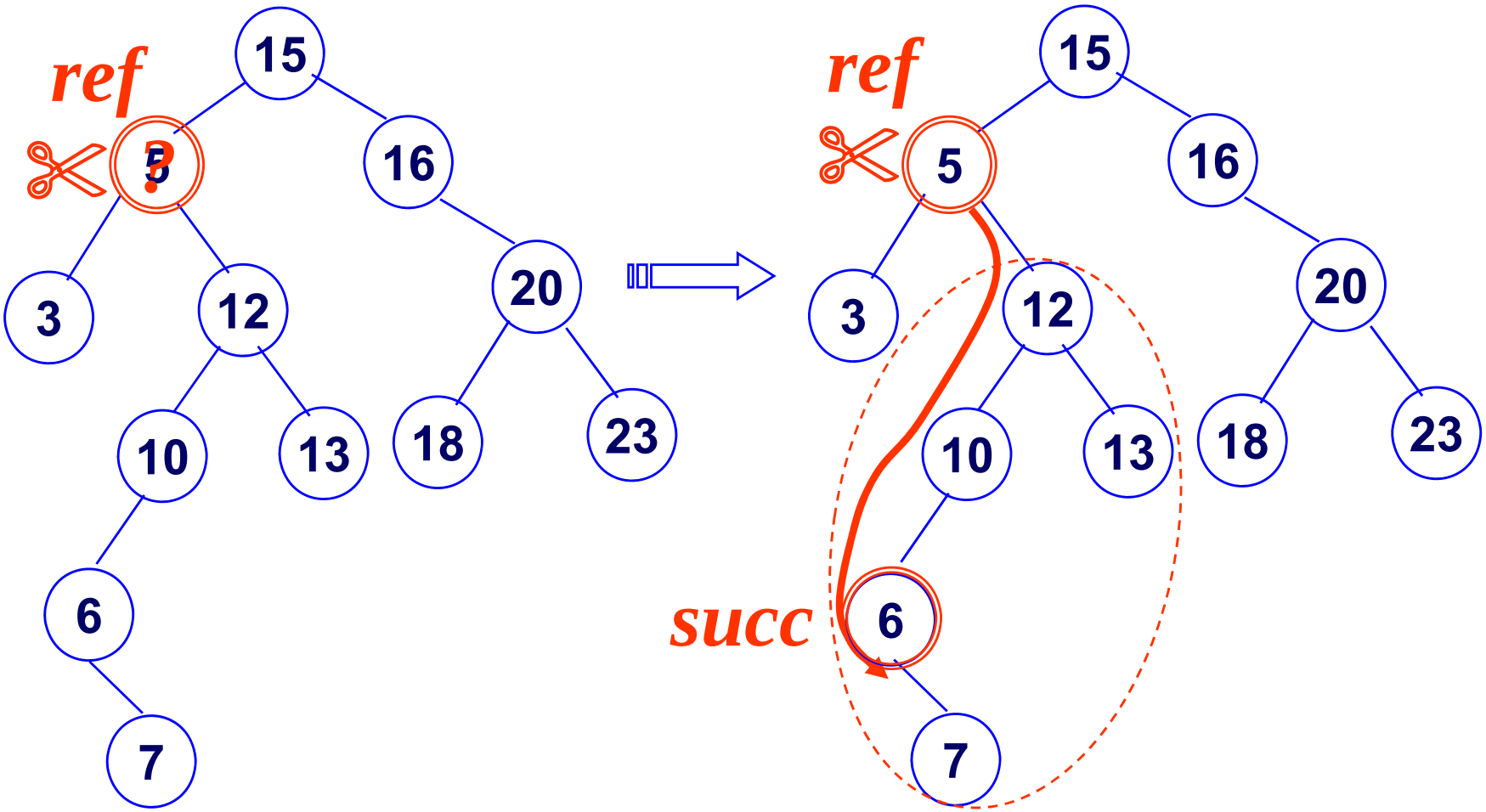
```
if (ref.left == null)
    child = ref.right;
else child = ref.left;

// 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

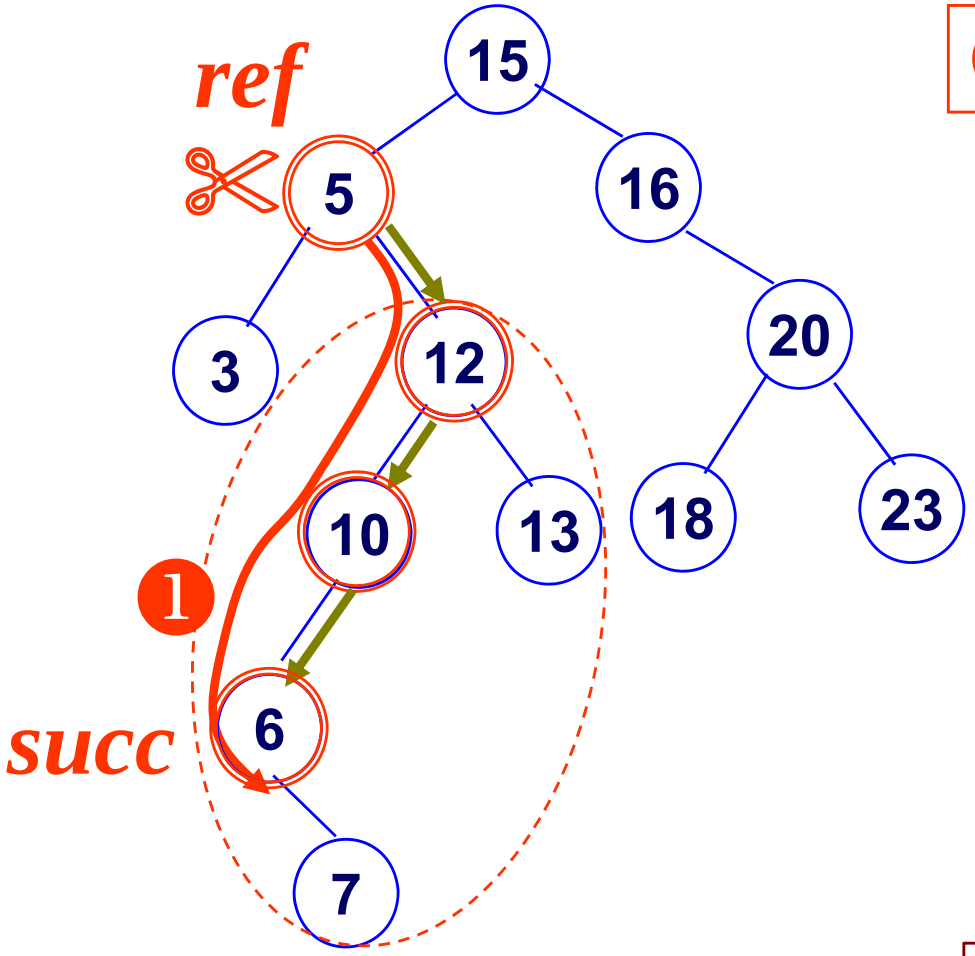
```
BSTreplace(root, ref) {// ref has only one child
    // set child to ref's child, or null if no child
    if (ref.left == null)
        child = ref.right
    else child = ref.left

    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



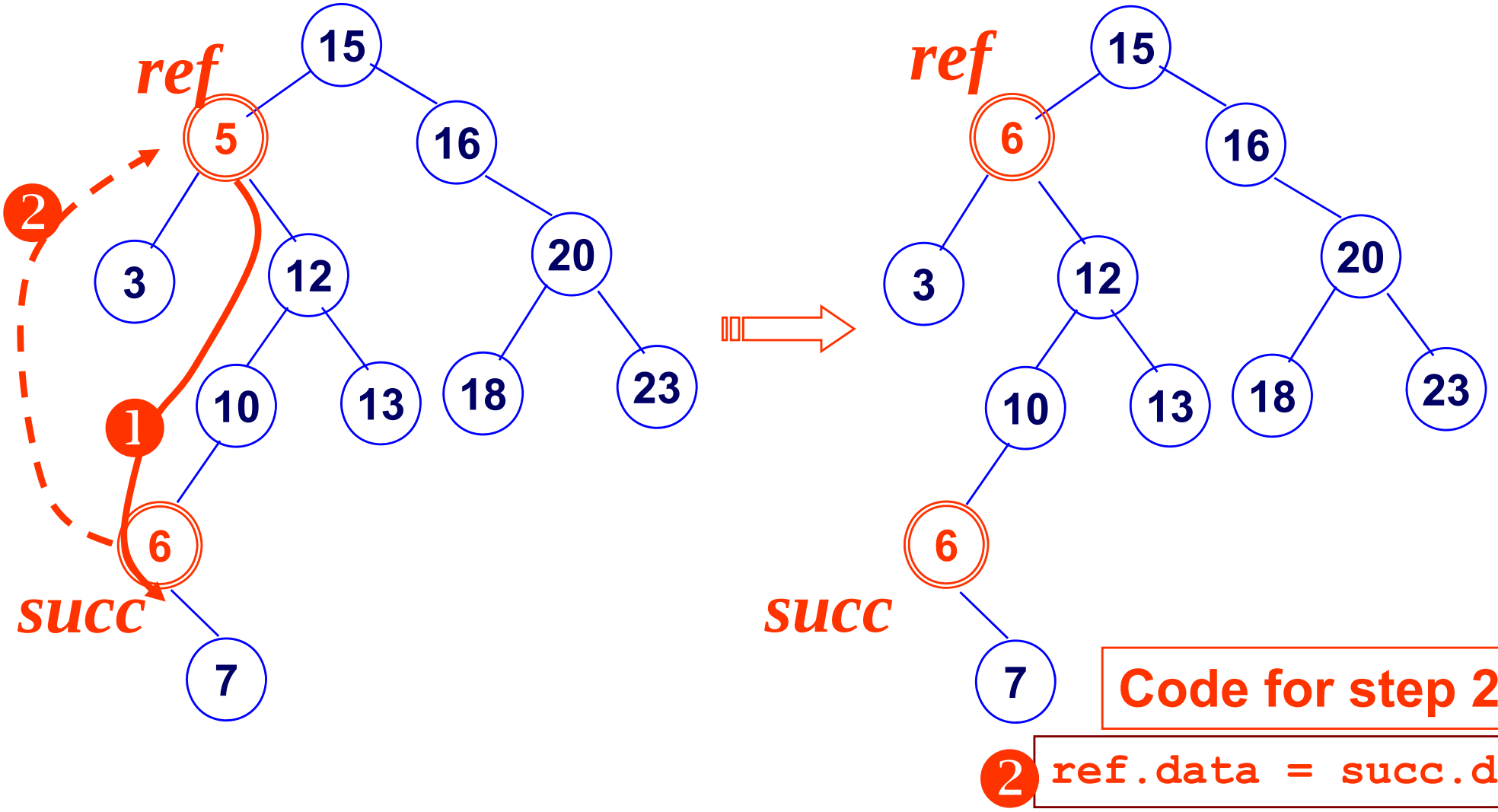
Code for step 1:

```
succ = ref.right
while (succ.left ≠ null)
    succ = succ.left
```
1

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

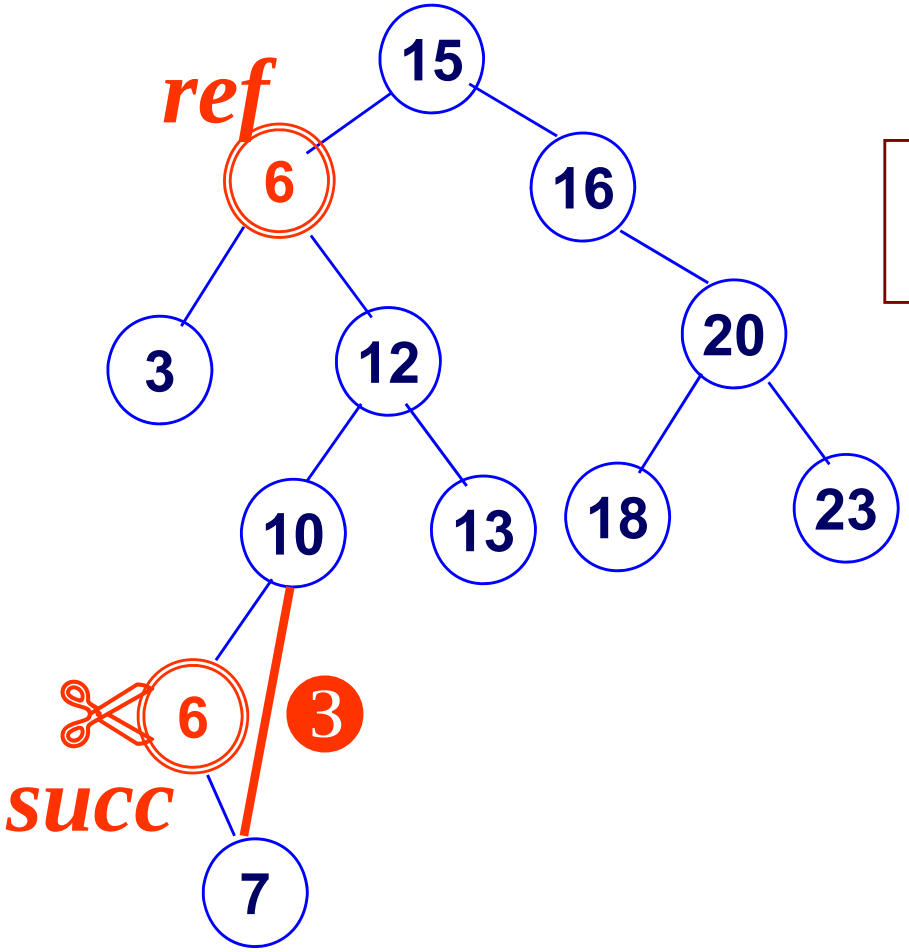


Code for step 2:

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)*

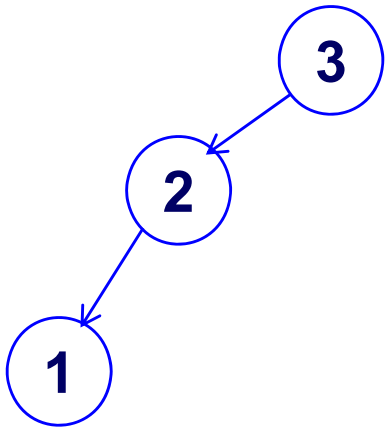
Put all together: Deletion Algorithm

```
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)
}
```

AVL Tree

Unbalanced Tree

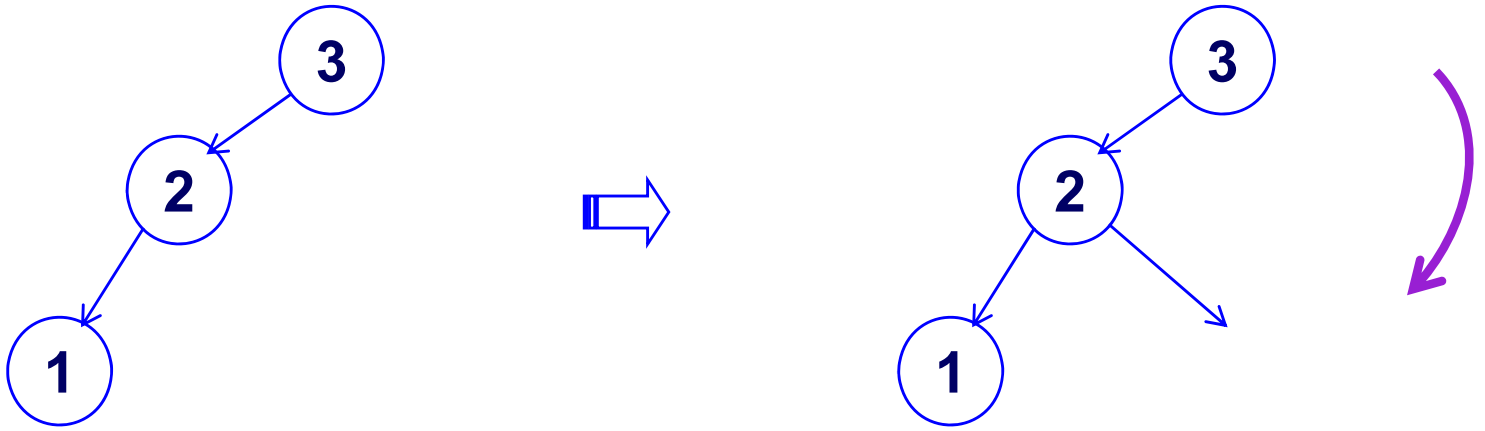
- Suppose we use BSTInsert to insert the data values 3, 2, 1 into an empty BST:



- Same as a linked list!
- Worst case height: O(n) – inefficient for searching

Example

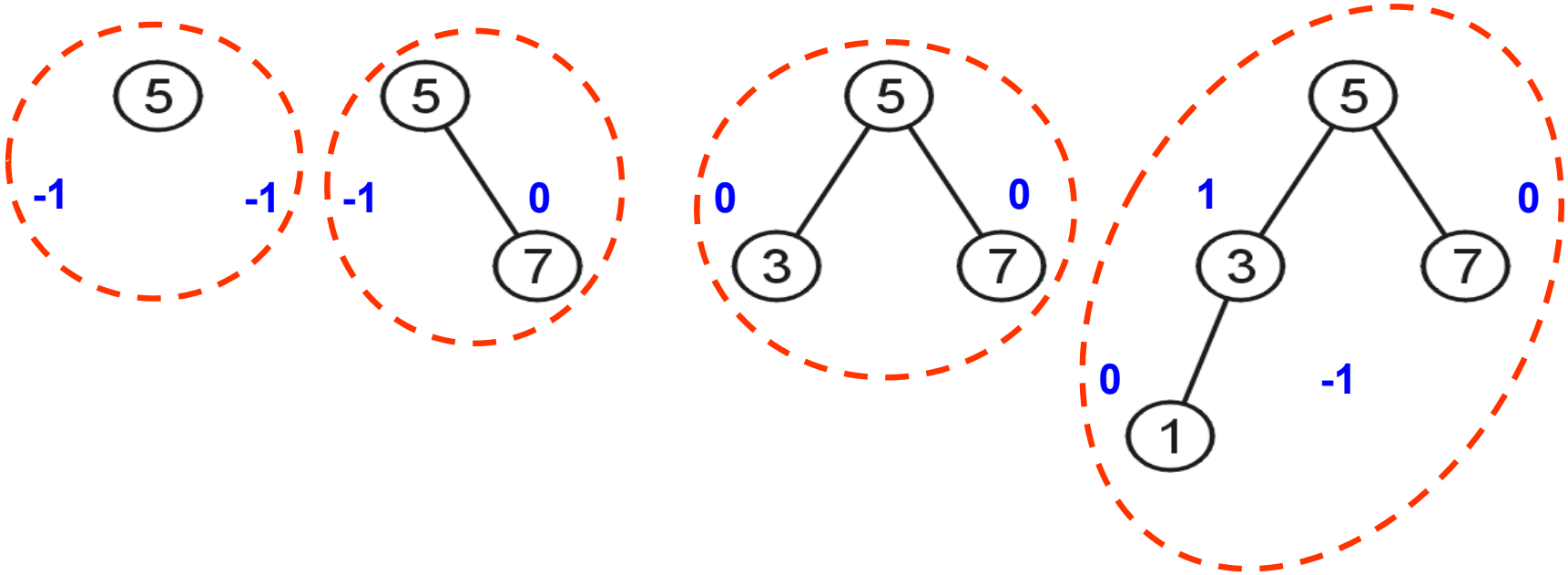
- We can easily fix this and make tree somehow balanced



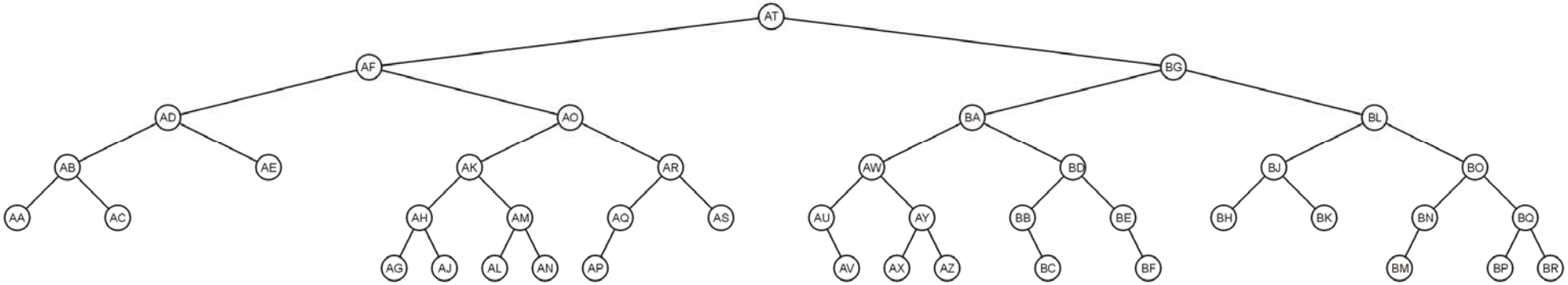
AVL Trees

- Named after Adelson-Velskii and Landis
- Define height of a tree as the number of edges on the longest path from the root to a leaf.
  - An empty tree has height –1
  - A tree with a single node has height 0
- A BST is said to be AVL balanced if the difference in the heights between the left and right sub-trees of any node is at most 1.

AVL trees with 1, 2, 3, and 4 nodes:

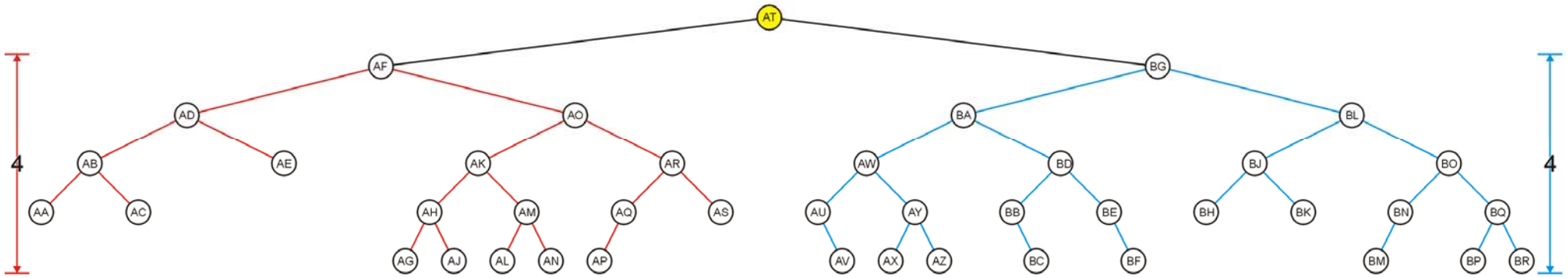


Here is a larger AVL tree (42 nodes):



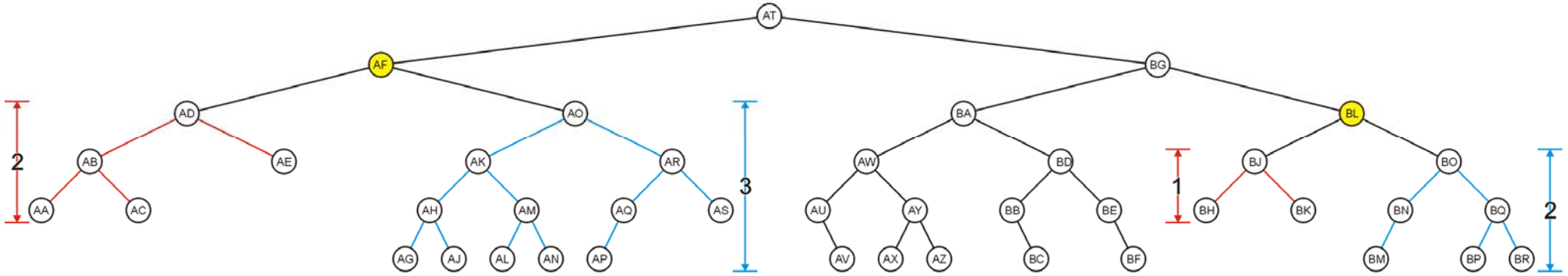
The root node is AVL-balanced:

- ☞ Both sub-trees are of height 4:



All other nodes (*e.g.*, AF and BL) are AVL balanced

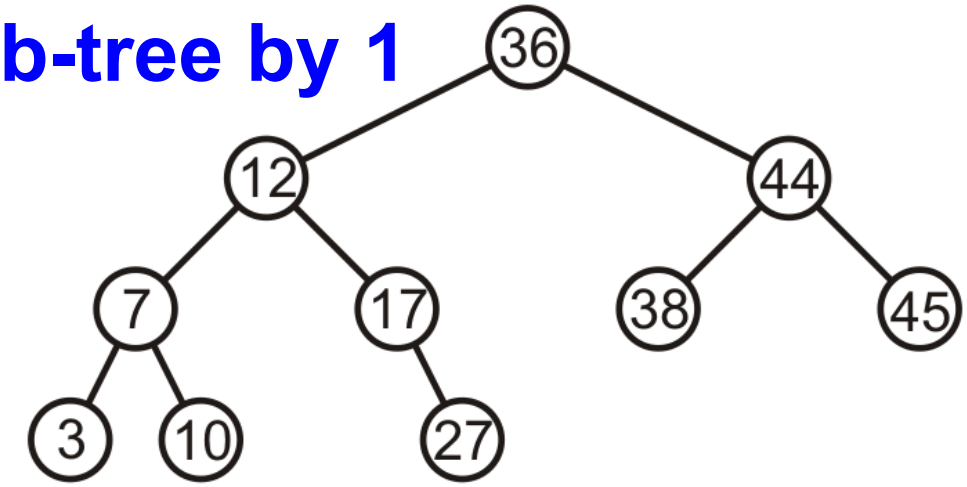
- ☞ The sub-trees differ in height by at most one



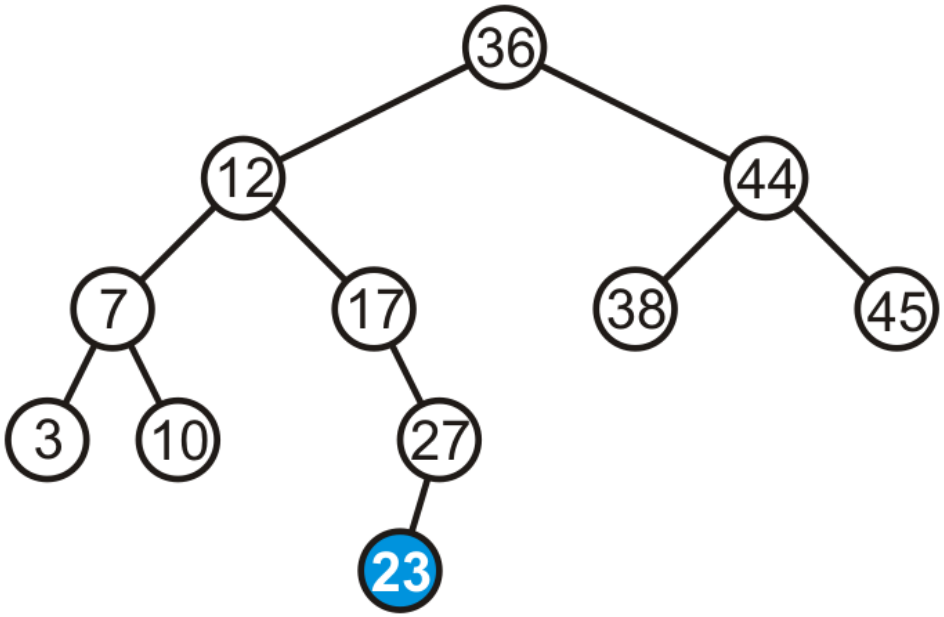
Maintaining Balance

If a tree is AVL balanced, for an insertion to cause an imbalance:

- **The heights of the sub-trees must differ by 1**
- **The insertion must increase the height of the deeper sub-tree by 1**

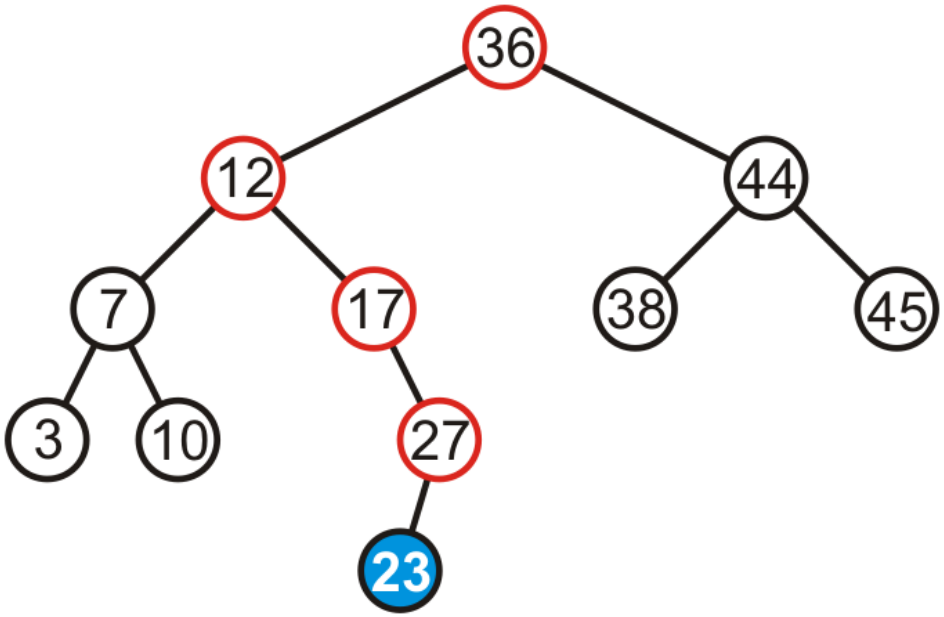


Suppose we insert 23 into our initial tree



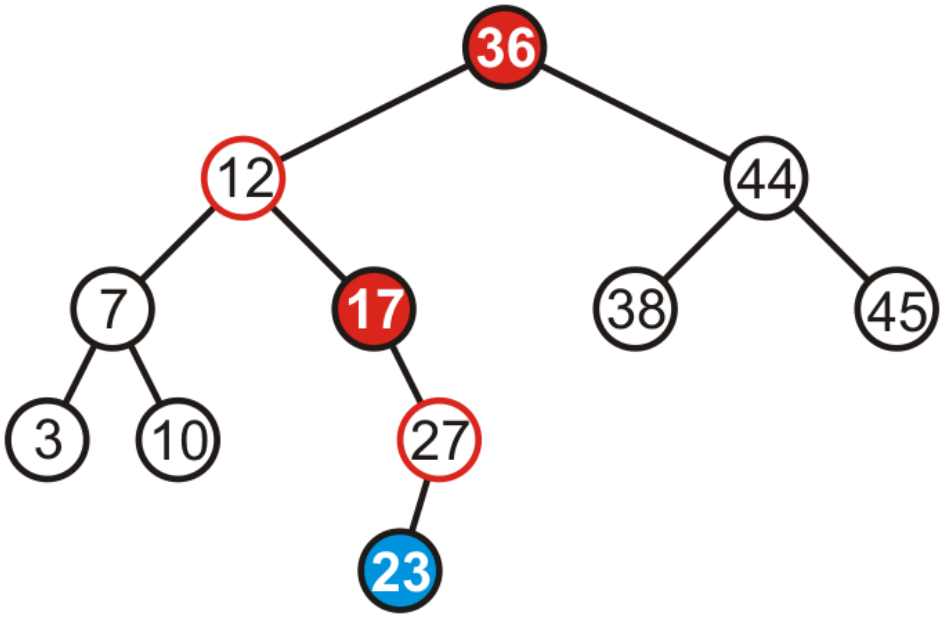
Maintaining Balance

The heights of each of the sub-trees from here to the root are increased by one



Maintaining Balance

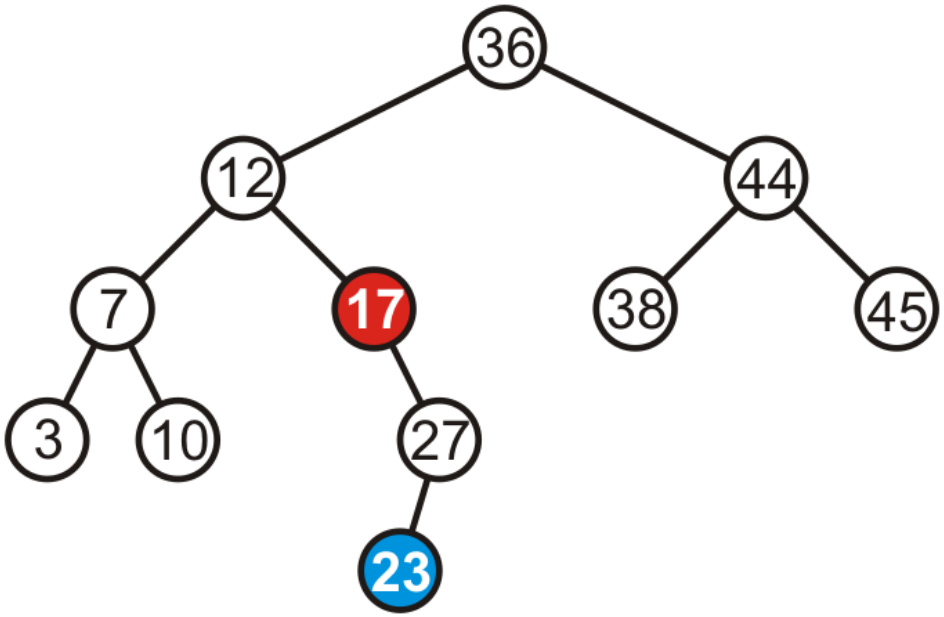
However, only two of the nodes are unbalanced: 17 and 36



Maintaining Balance

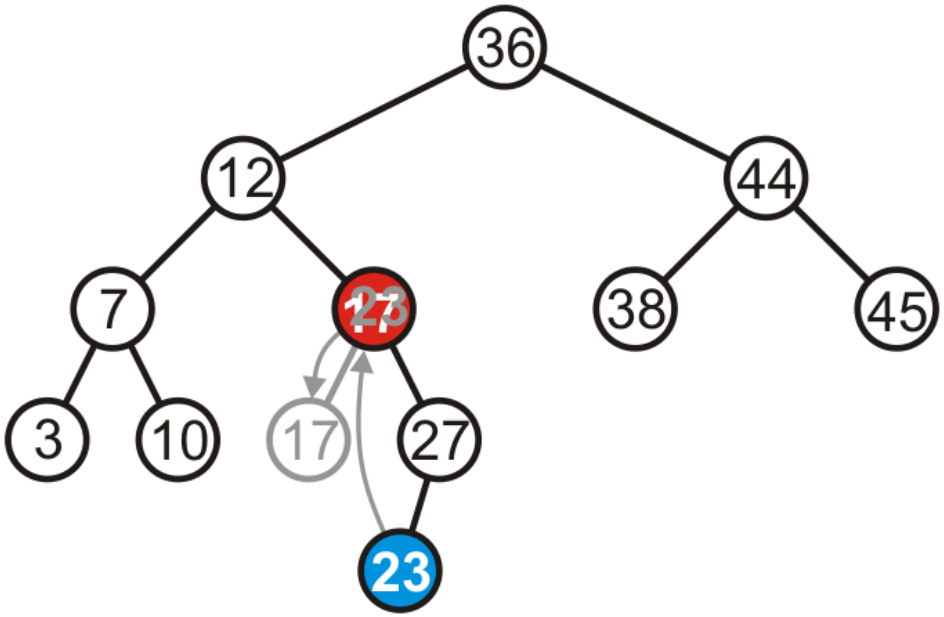
However, only two of the nodes are unbalanced: 17 and 36

- ☞ We only have to fix the imbalance at the lowest node



Maintaining Balance

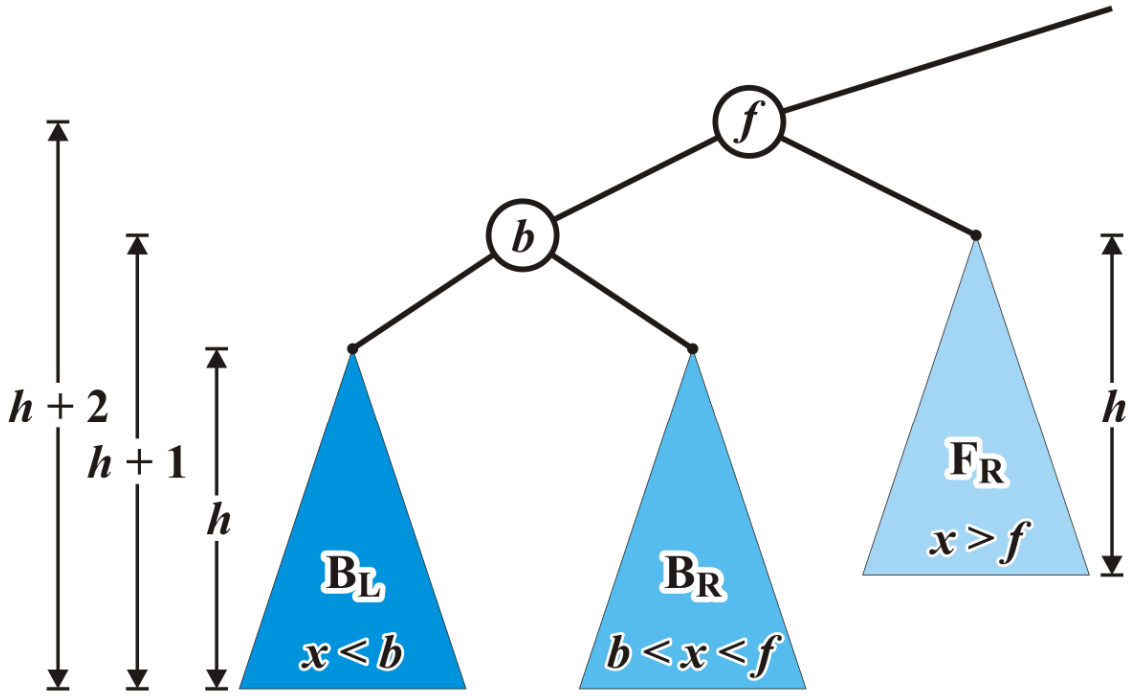
We can promote 23 to where 17 is, and make 17 the left child of 23



Maintaining Balance: Case Left-Left

Consider the following setup

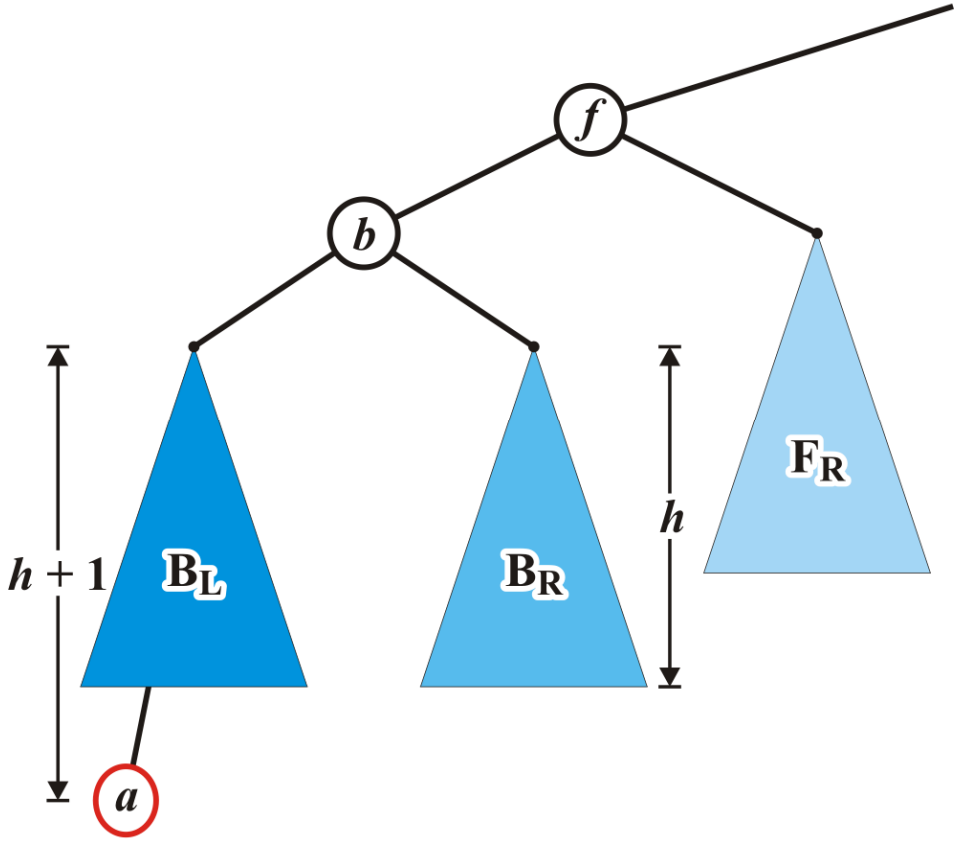
- Each blue triangle represents a tree of height h



Maintaining Balance: Case Left-Left

Insert *a* into this tree: it falls into the left subtree B_L of *b*

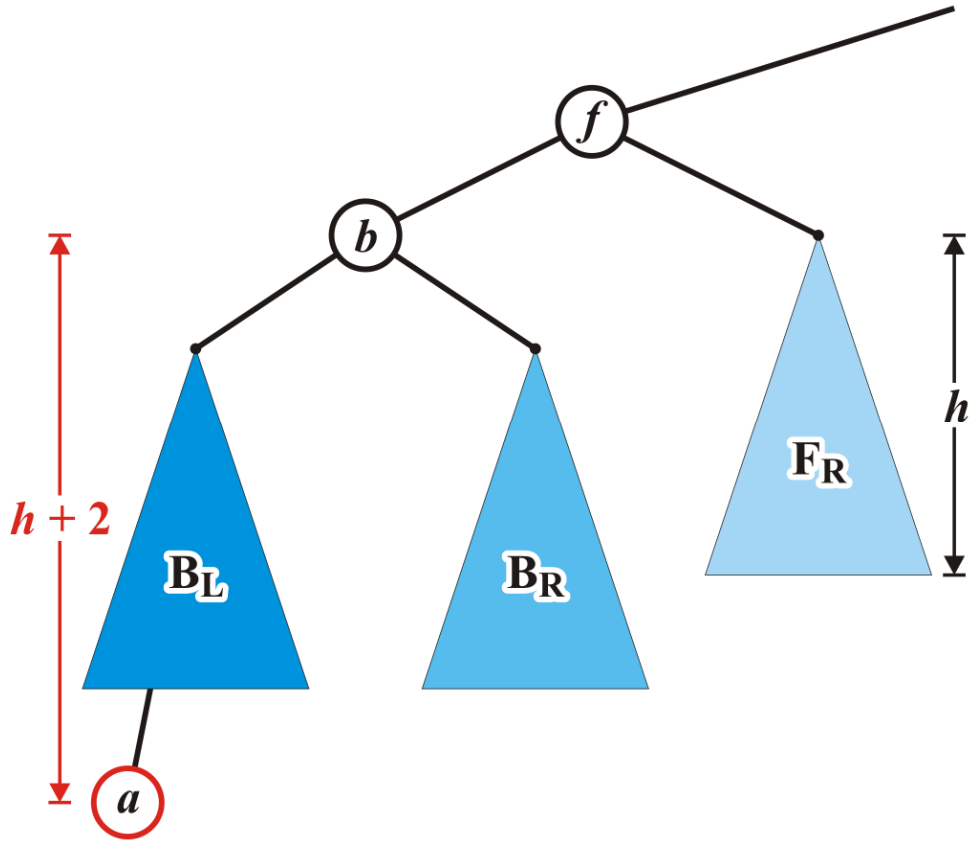
- Assume B_L remains balanced
- Thus, the tree rooted at *b* is also balanced



Maintaining Balance: Case Left-Left

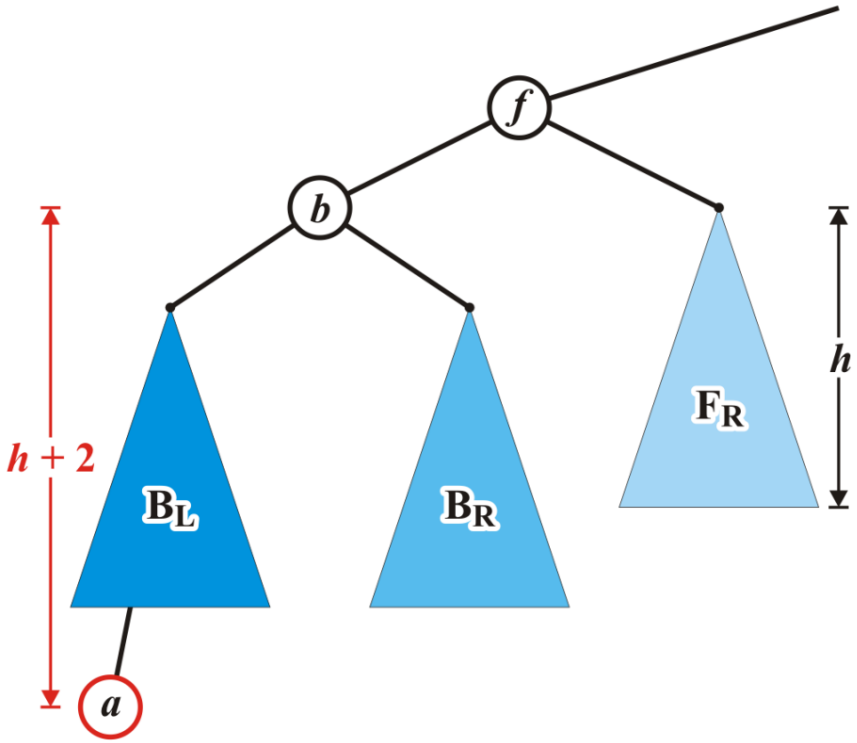
The tree rooted at node *f* is now unbalanced

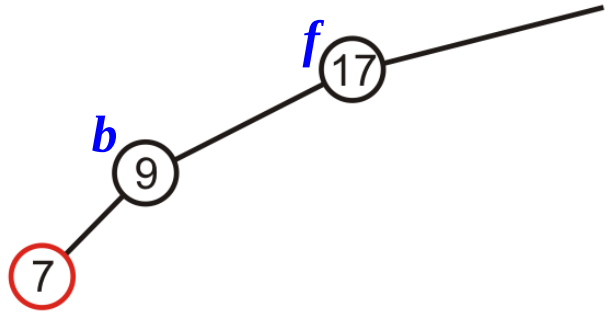
☞ We will correct the imbalance at this node

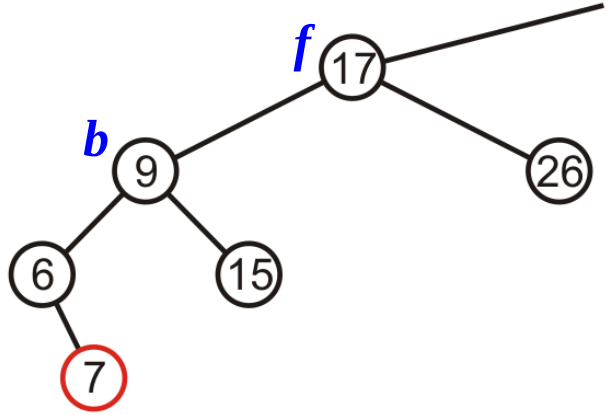


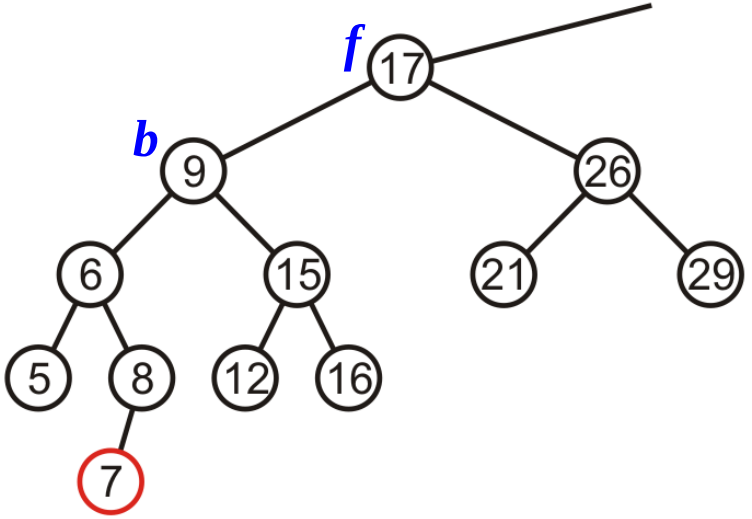
Maintaining Balance: Case Left-Left

Here are examples of when the insertion of 7 may cause this situation when $h = -1, 0$, and 1



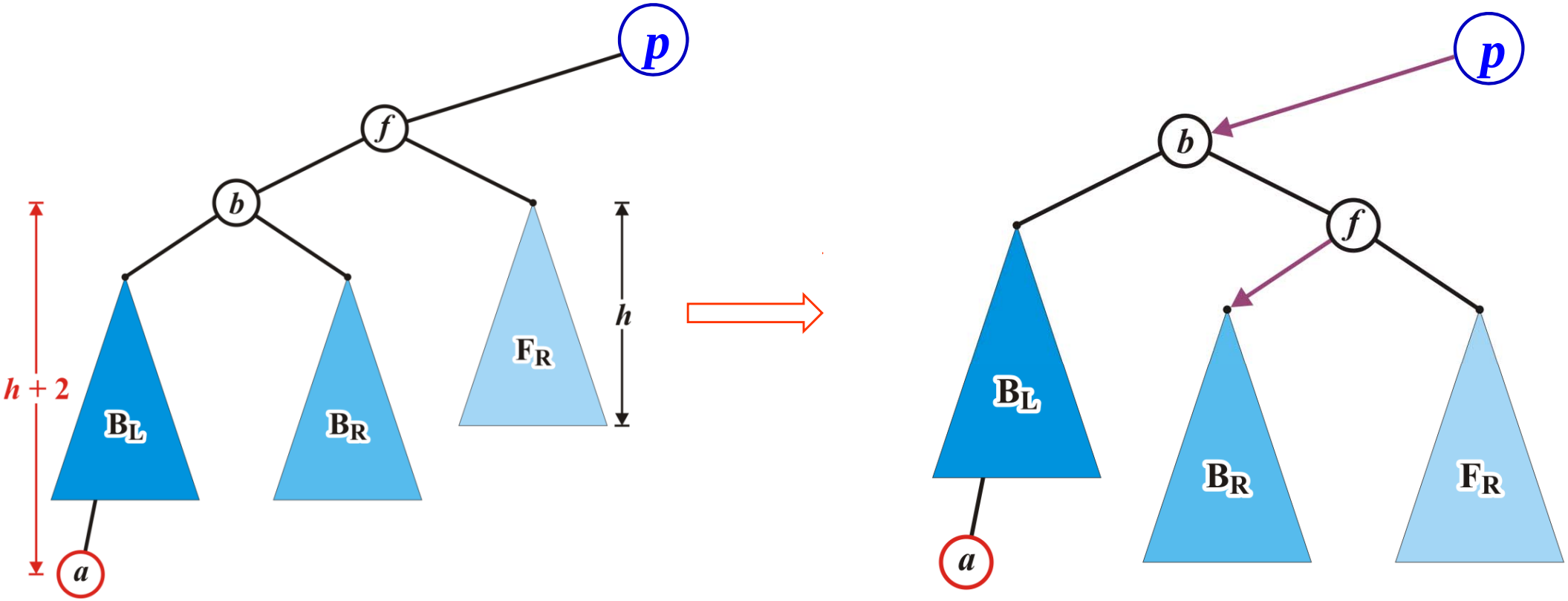






Maintaining Balance: Case Left-Left

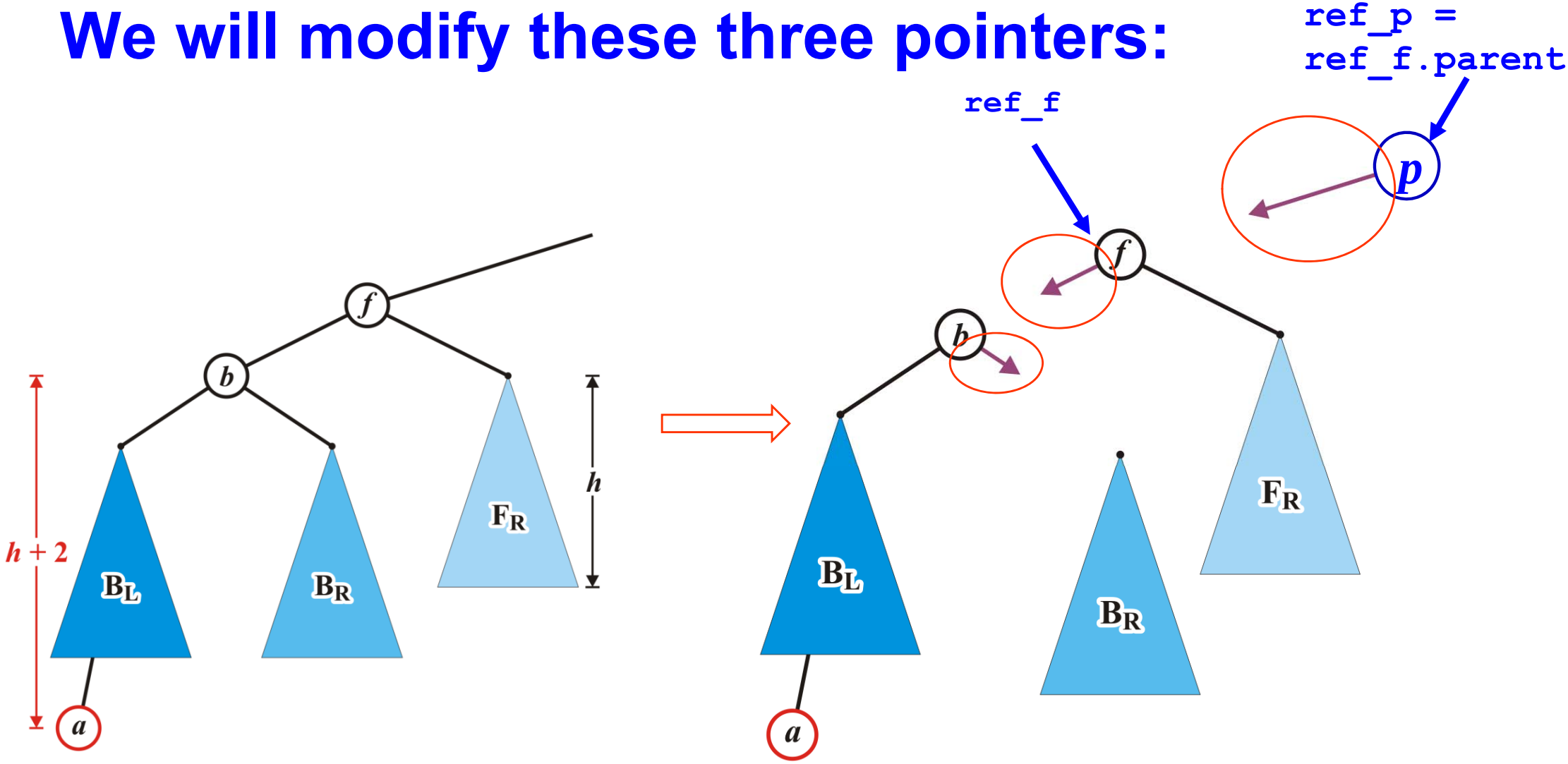
The tree will become balanced if we modify these three pointers:



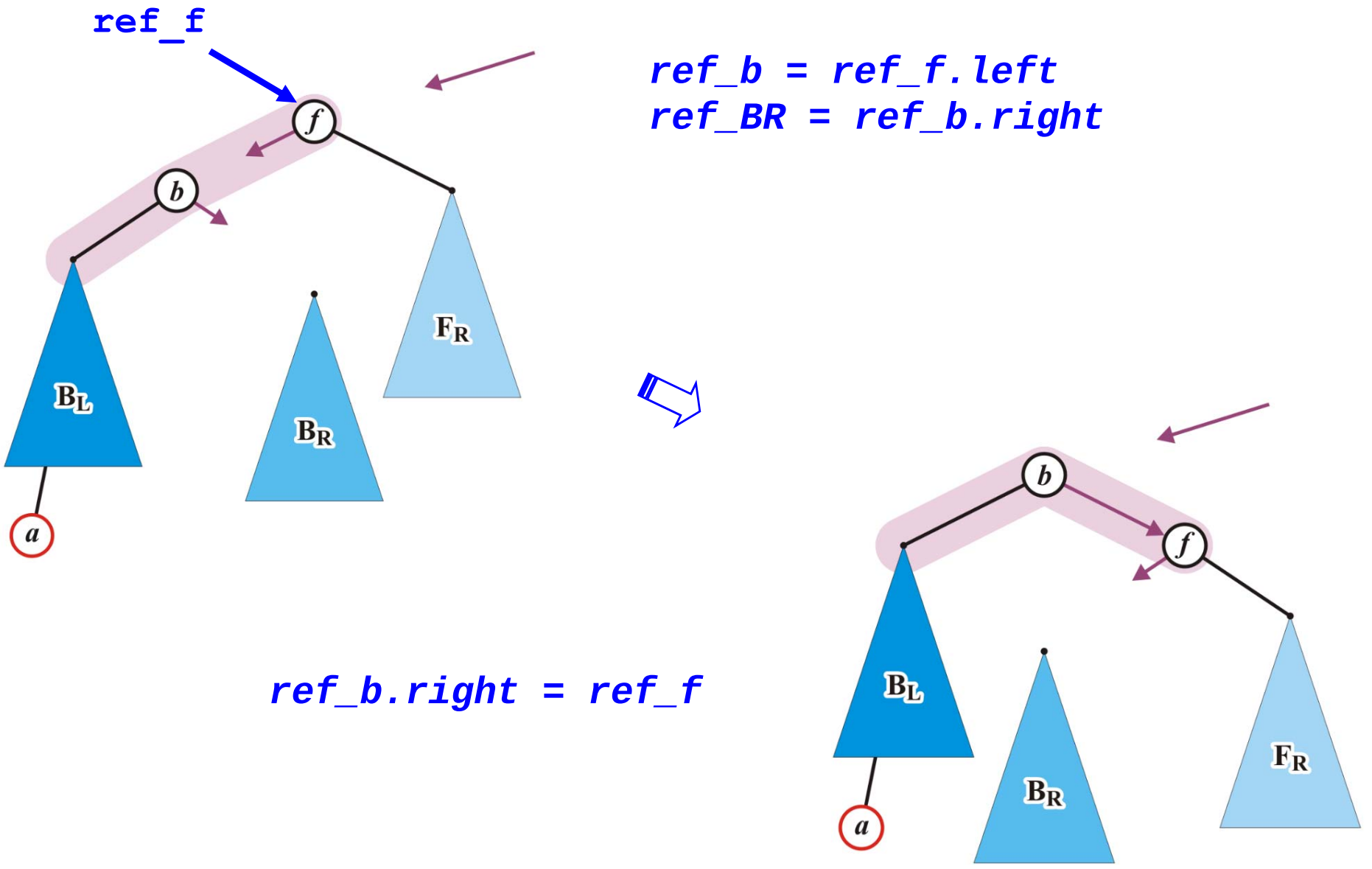
Maintaining Balance: Case Left-Left

Node *f is the lowest unbalanced node.* Suppose its address is given in `ref_f`.

We will modify these three pointers:

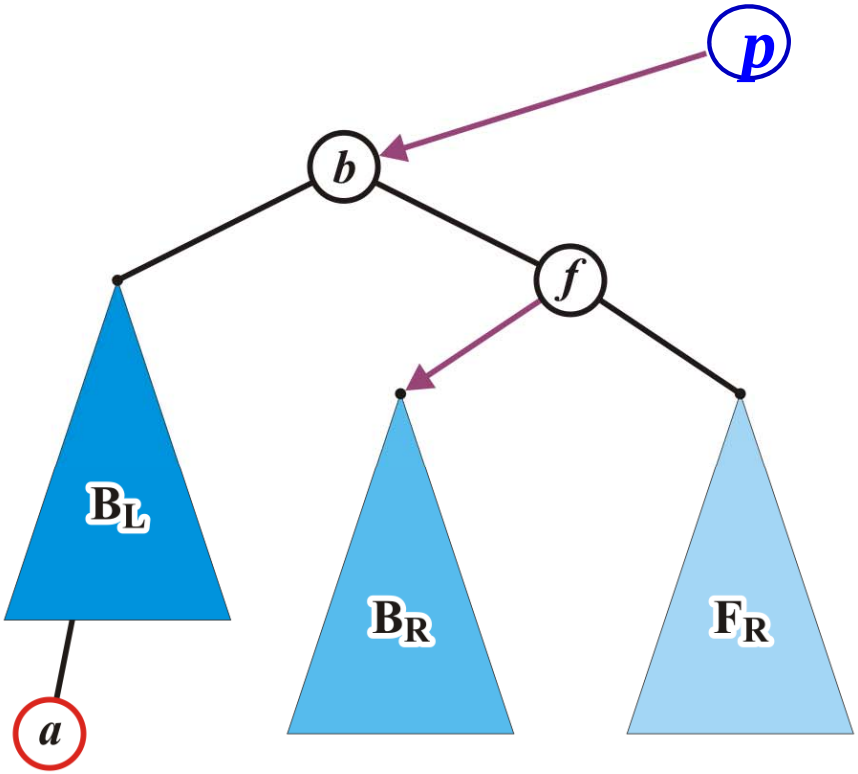


Maintaining Balance: Case Left-Left



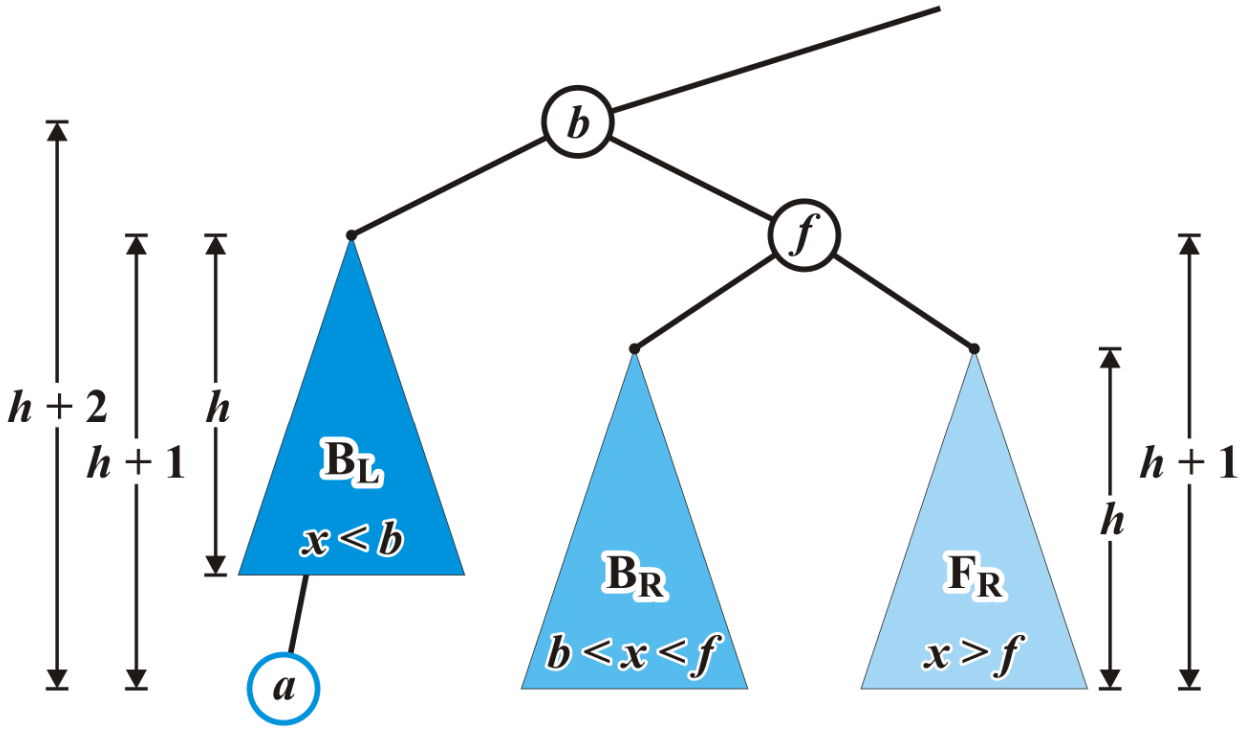
Maintaining Balance: Case Left-Left

```
ref_p = ref_f.parent
ref_p.left = ref_b
ref_f.left = ref_BR
ref_f.parent = ref_b
```



Maintaining Balance: Case Left-Left

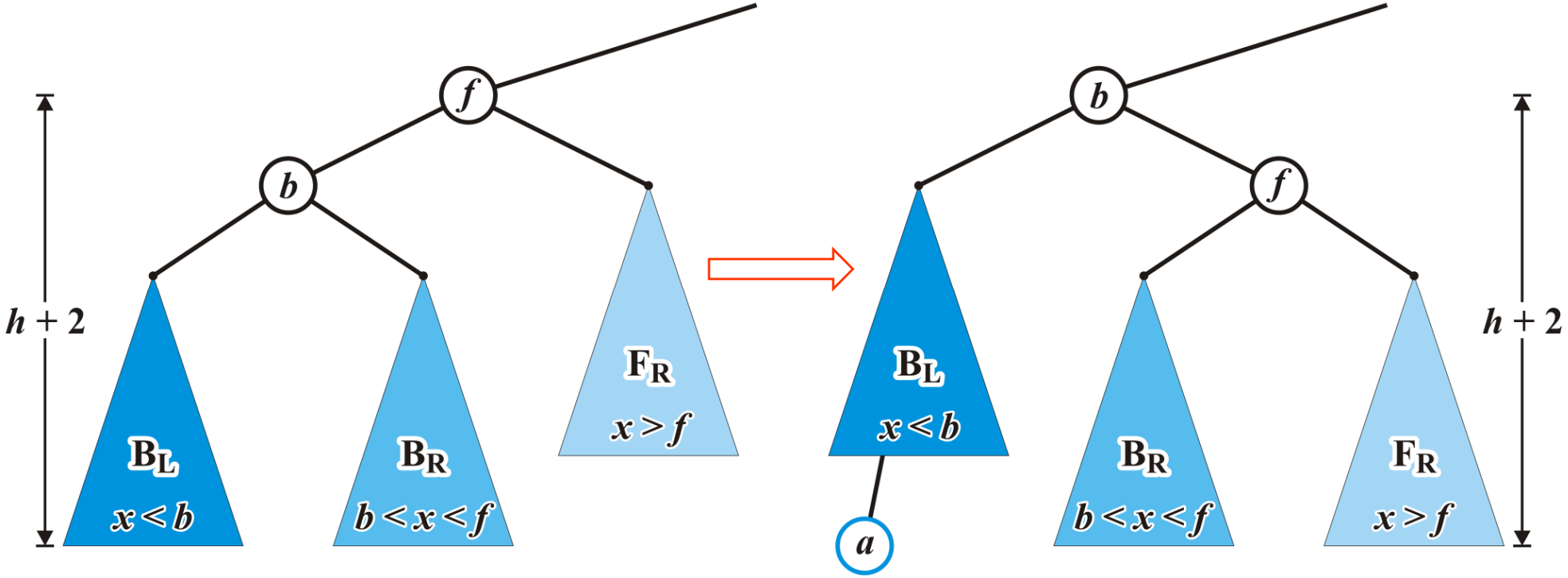
The nodes *b* and *f* are now balanced and all remaining nodes of the subtrees are in their correct positions



Maintaining Balance: Case Left-Left

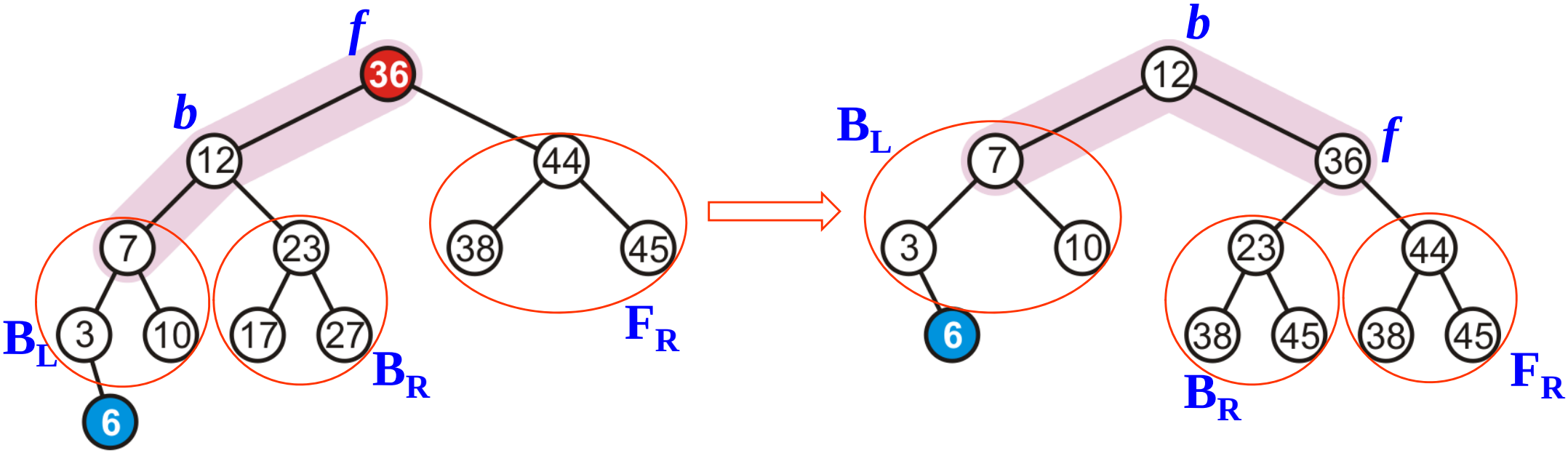
Additionally, height of the tree rooted at b equals the original height of the tree rooted at f

- **Thus, this insertion will no longer affect the balance of any ancestors all the way back to the root**

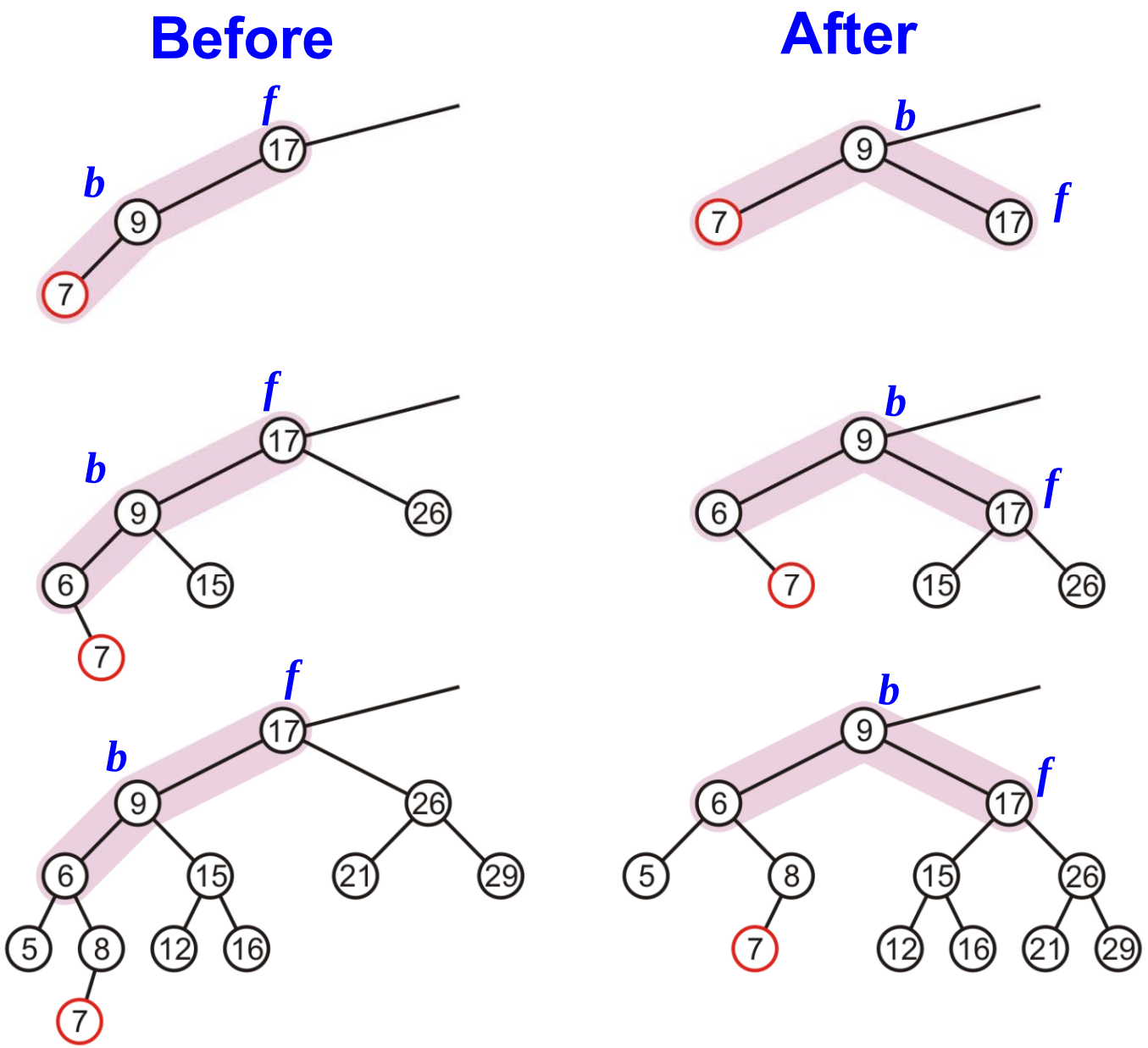


Maintaining Balance: Case Left-Left

In our example case, the correction

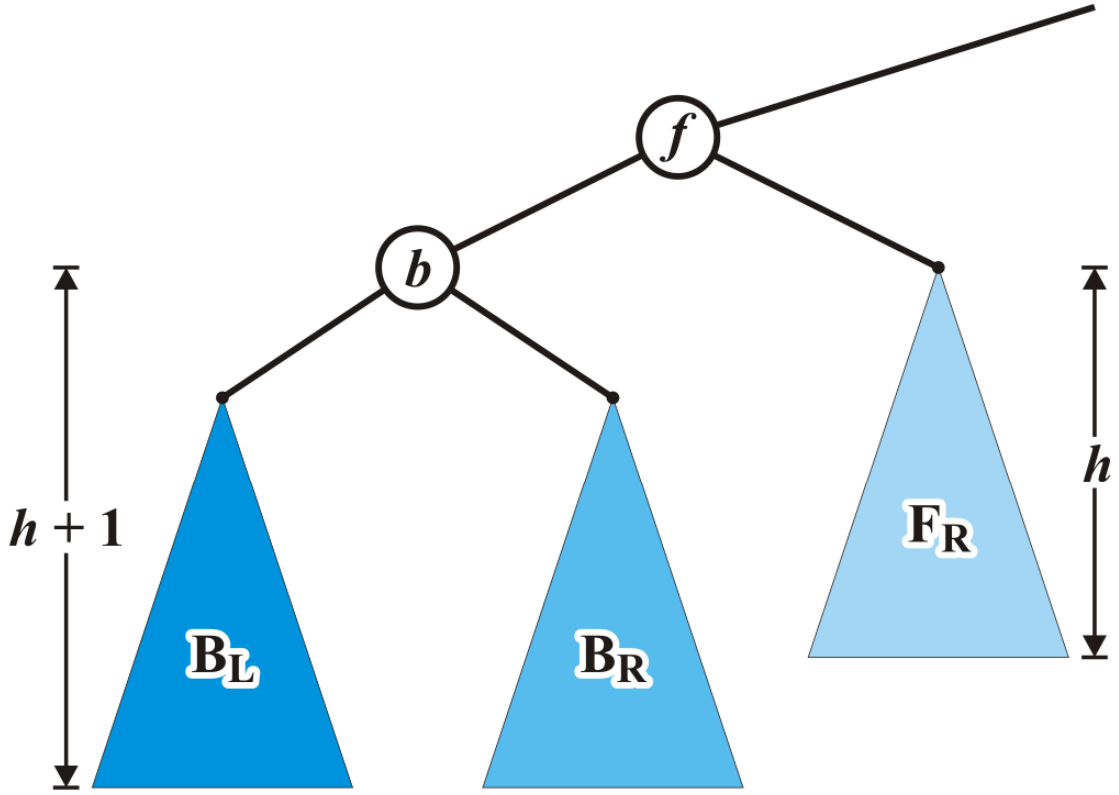


Maintaining Balance: Case Left-Left



Maintaining Balance: Case Left-Right

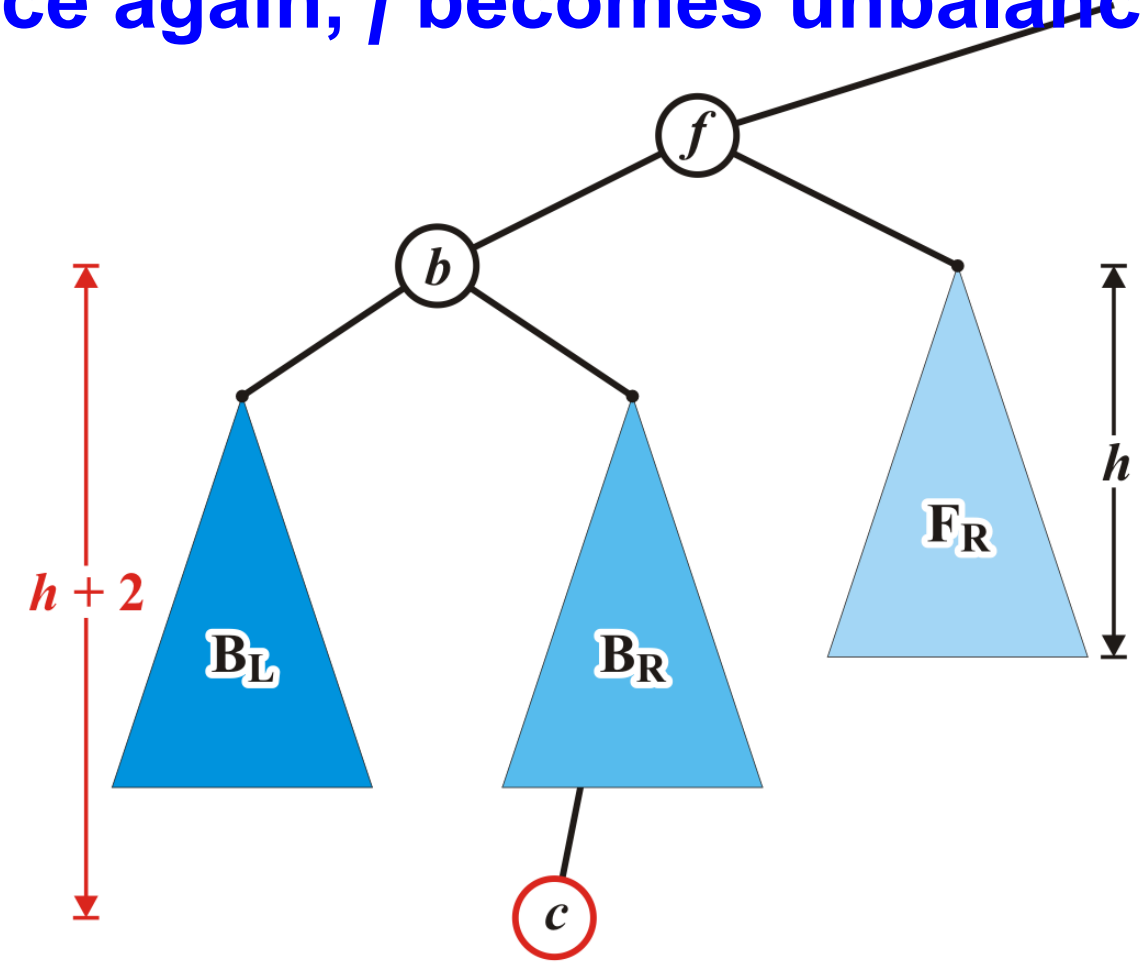
Alternatively, consider the insertion of c where $b < c < f$ into our original tree



Maintaining Balance: Case Left-Right

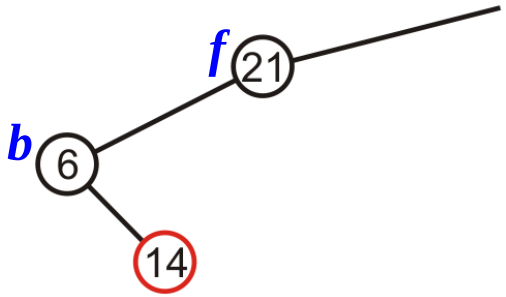
Assume that the insertion of c increases the height of B_R

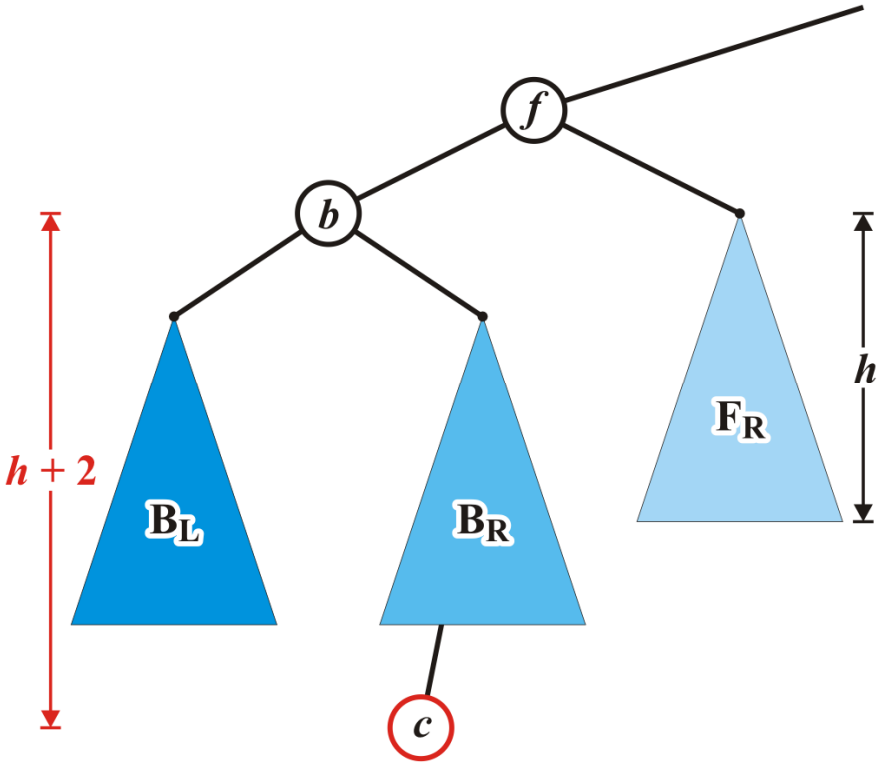
☞ Once again, f becomes unbalanced

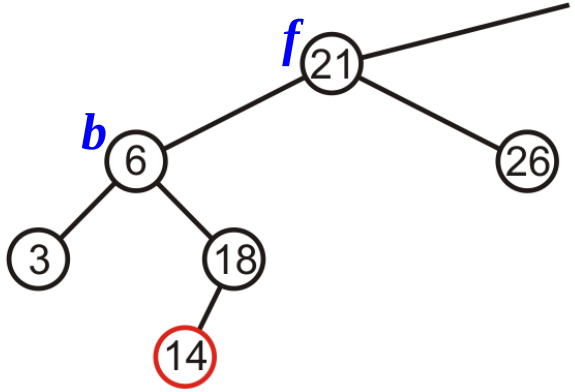


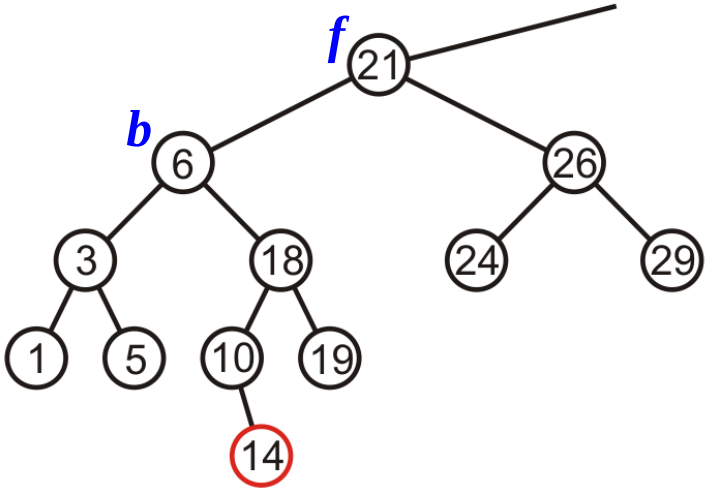
Maintaining Balance: Case Left-Right

Here are examples of when the insertion of 14 may cause this situation when $h = -1, 0,$ and 1



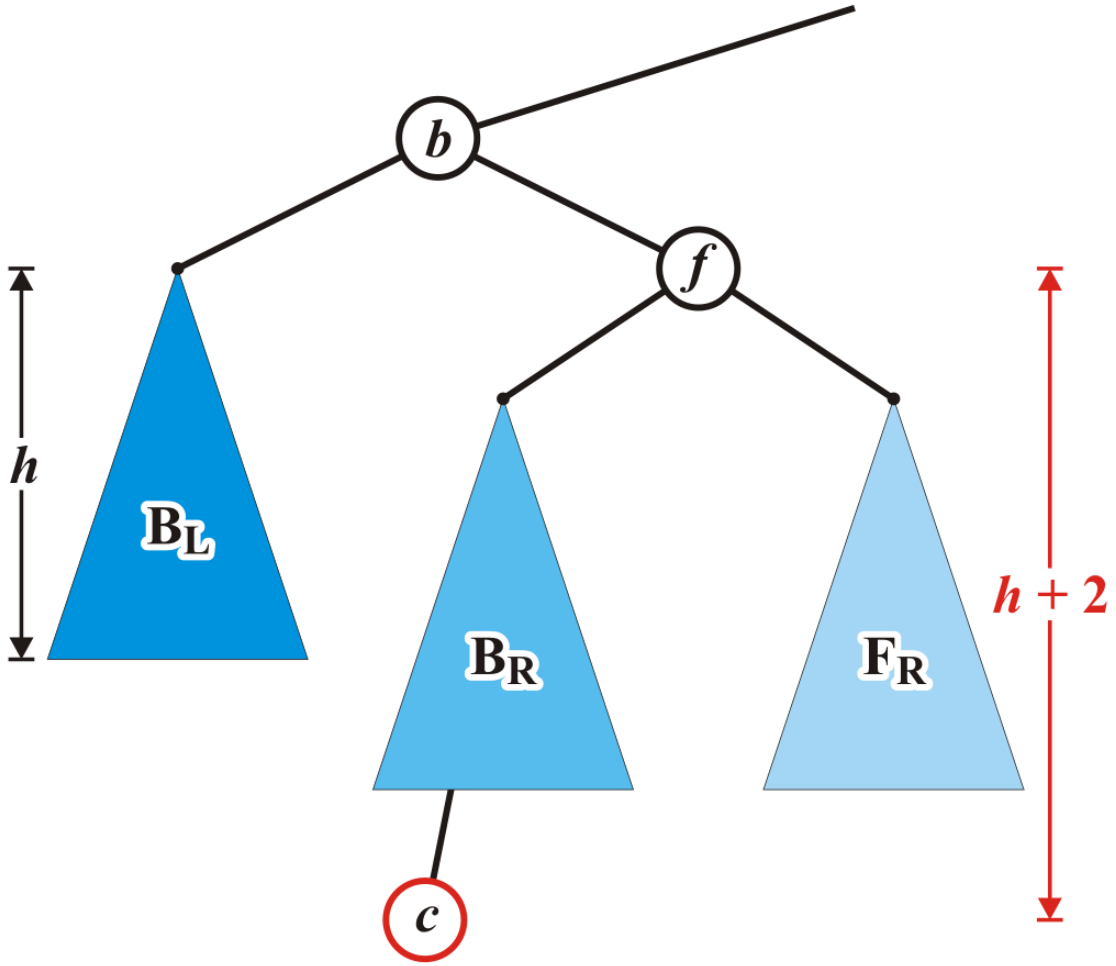






Maintaining Balance: Case Left-Right

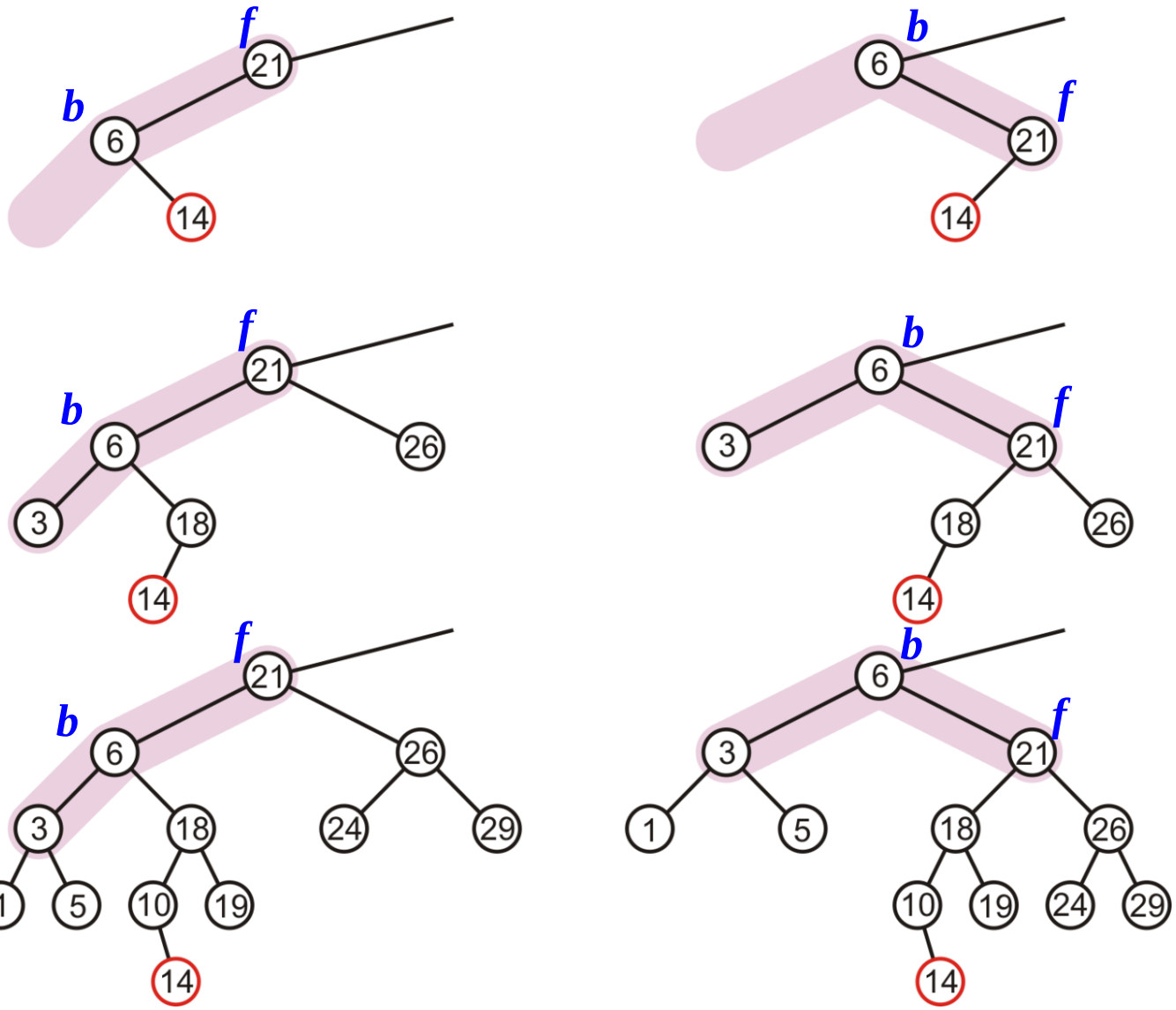
Unfortunately, the previous correction (for case left-left case) does not fix the imbalance at the root of this sub-tree: the new root, b, remains unbalanced



Maintaining Balance: Case Left-Right

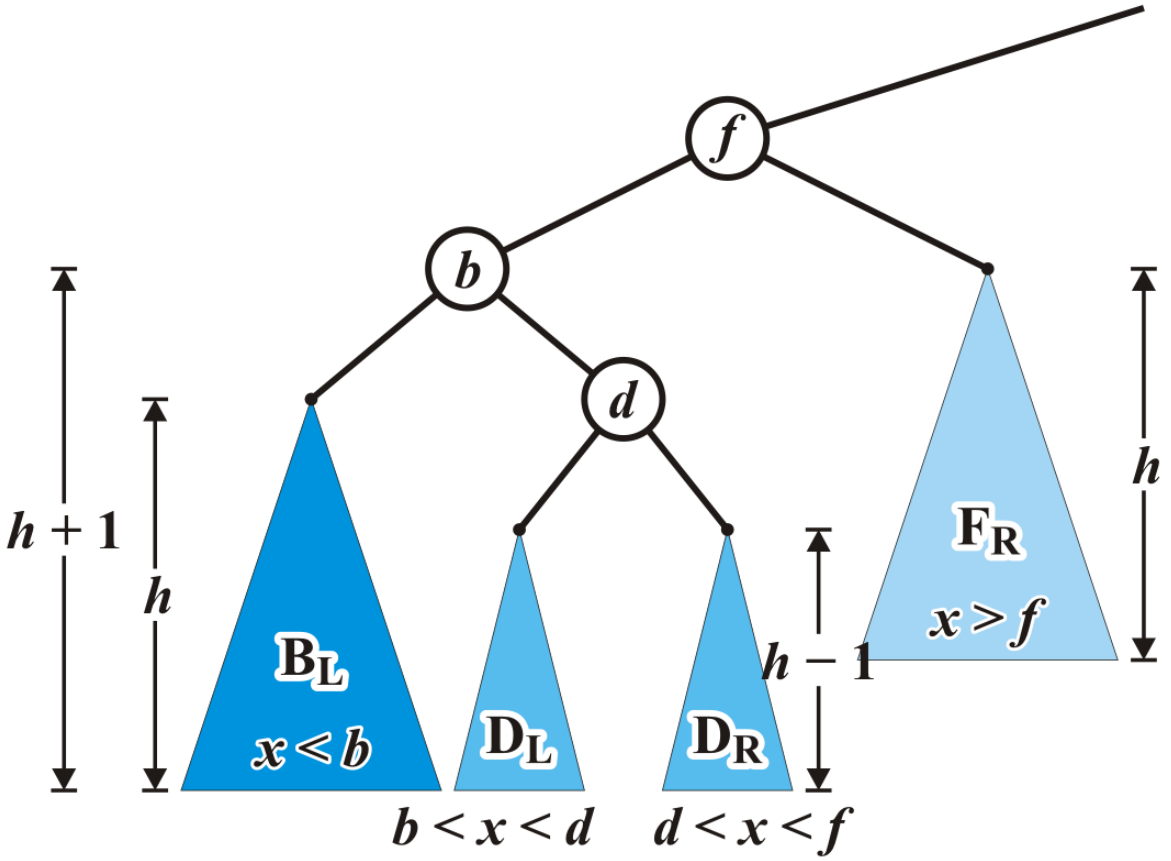
In our three sample cases with $h = -1, 0,$ and 1, doing the same thing as before results in a tree that is still unbalanced…

- The imbalance is just shifted to the other side



Maintaining Balance: Case Left-Right

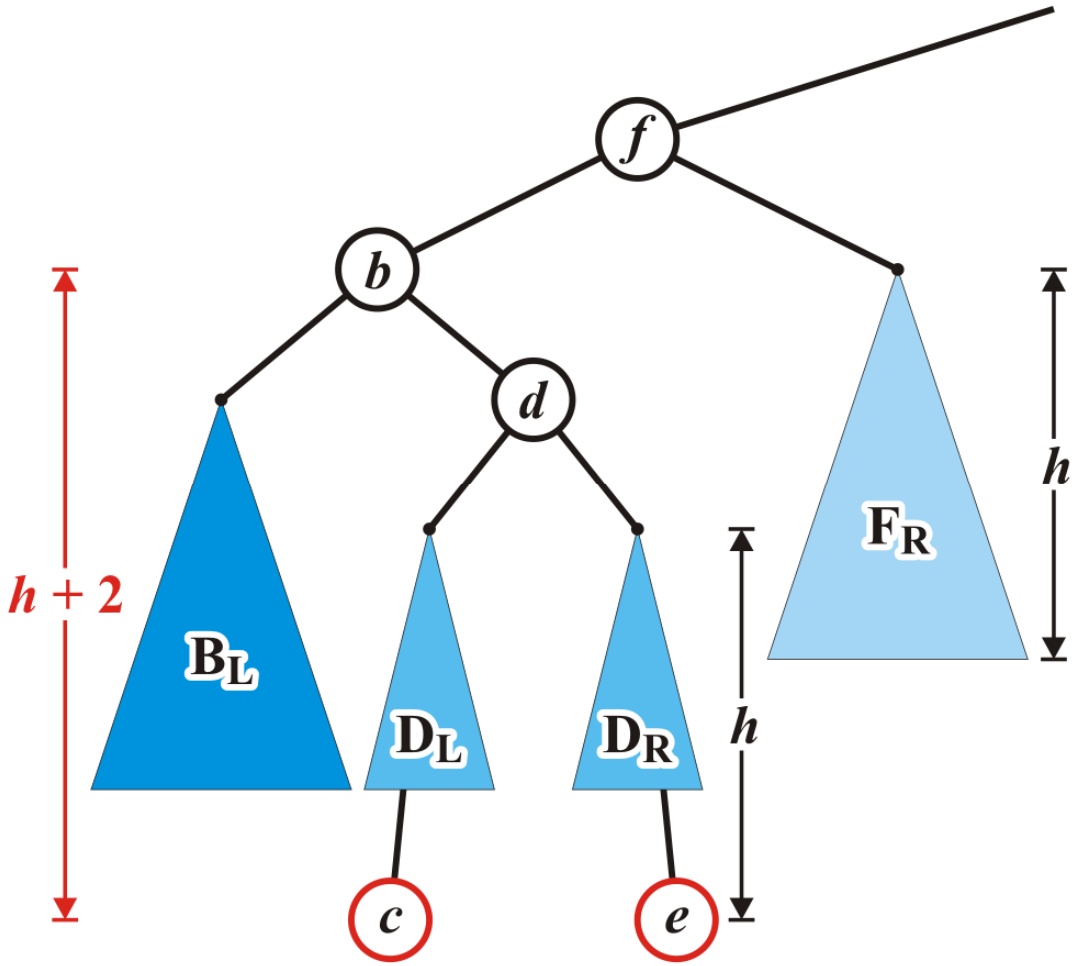
Consider tree rooted at d with two subtrees of height $h - 1$



Maintaining Balance: Case Left-Right

Now an insertion causes an imbalance at *f*

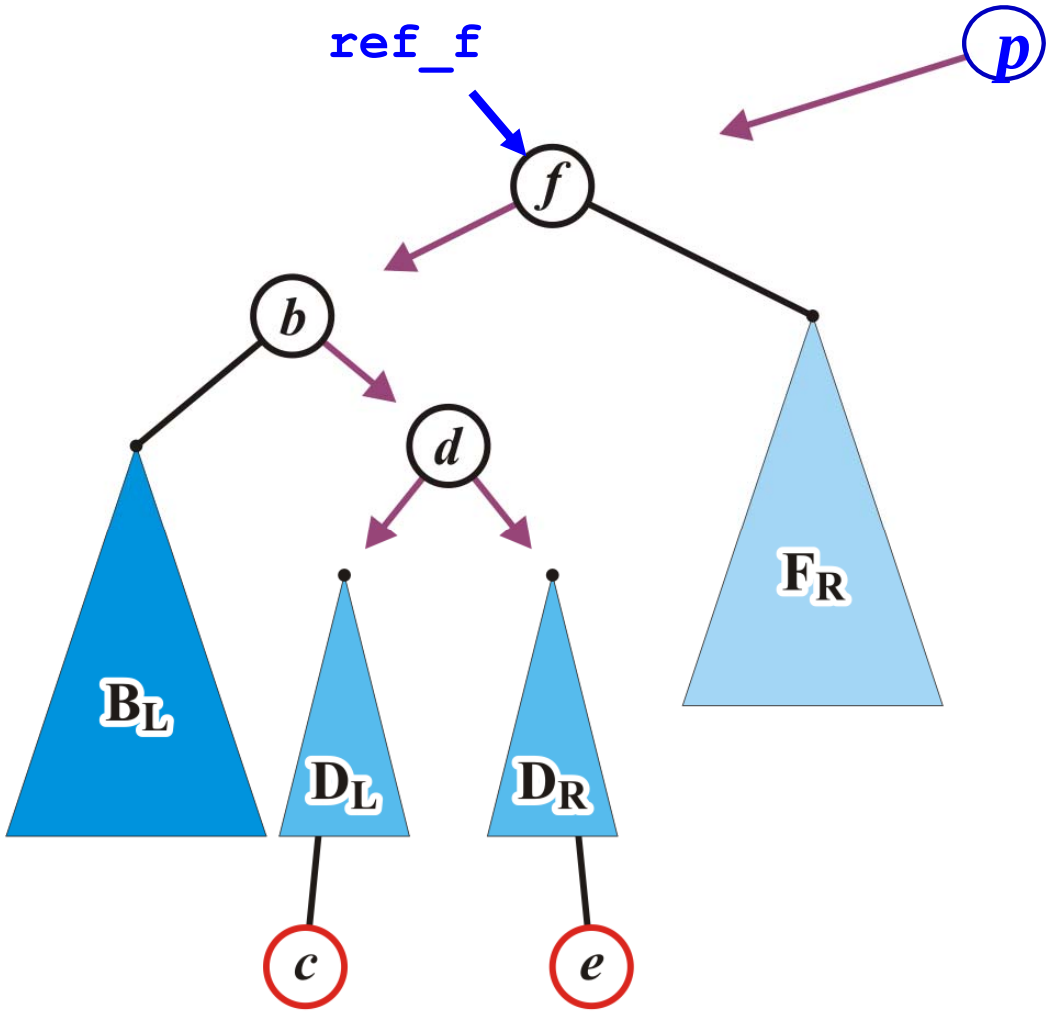
☞ The addition of either *c* or *e* will cause this



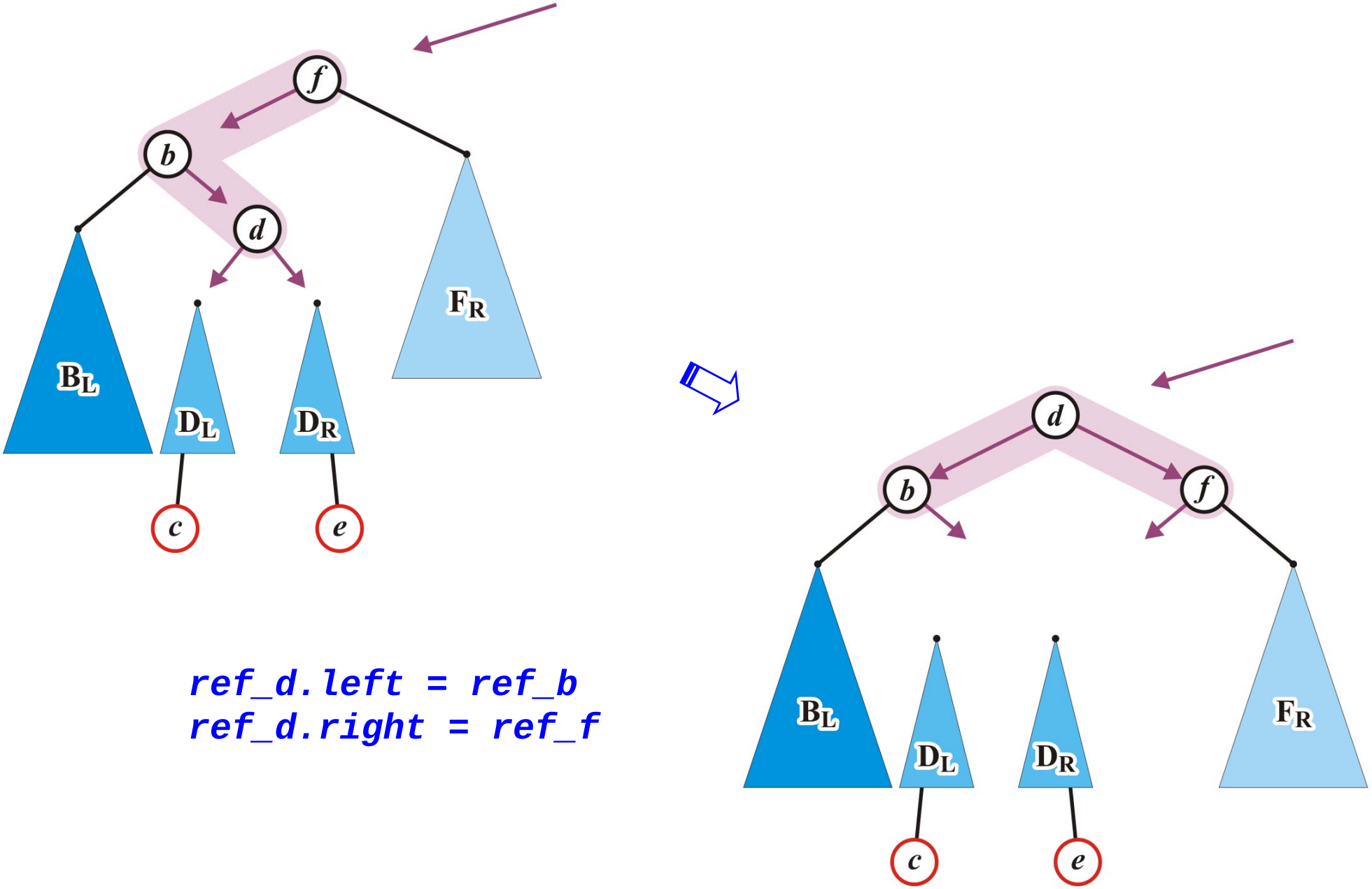
Maintaining Balance: Case Left-Right

Given ref_f, we will reassign the following pointers

```
ref_p = ref_f.parent
ref_b = ref_f.left
ref_d = ref_b.right
ref_DL = ref_d.left
ref_DR = ref_d.right
```



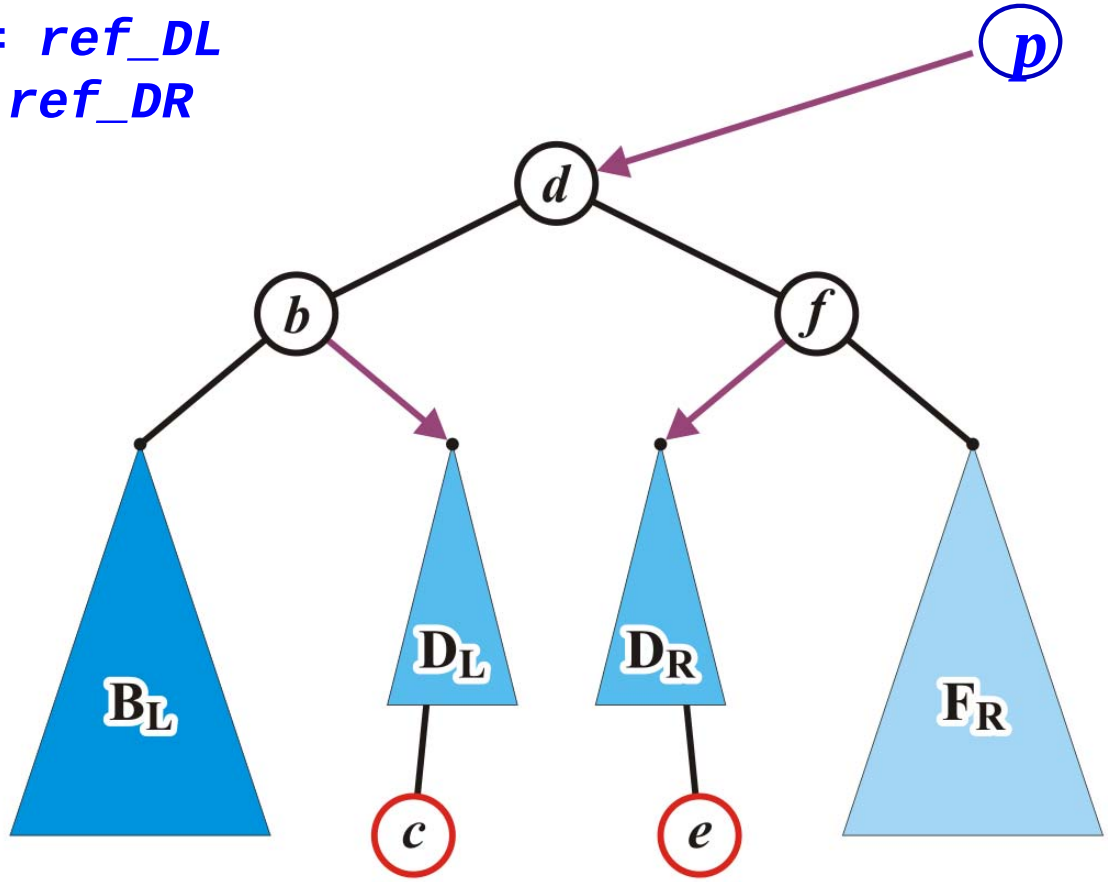
Maintaining Balance: Case Left-Right




```
ref_d.left = ref_b
ref_d.right = ref_f
```

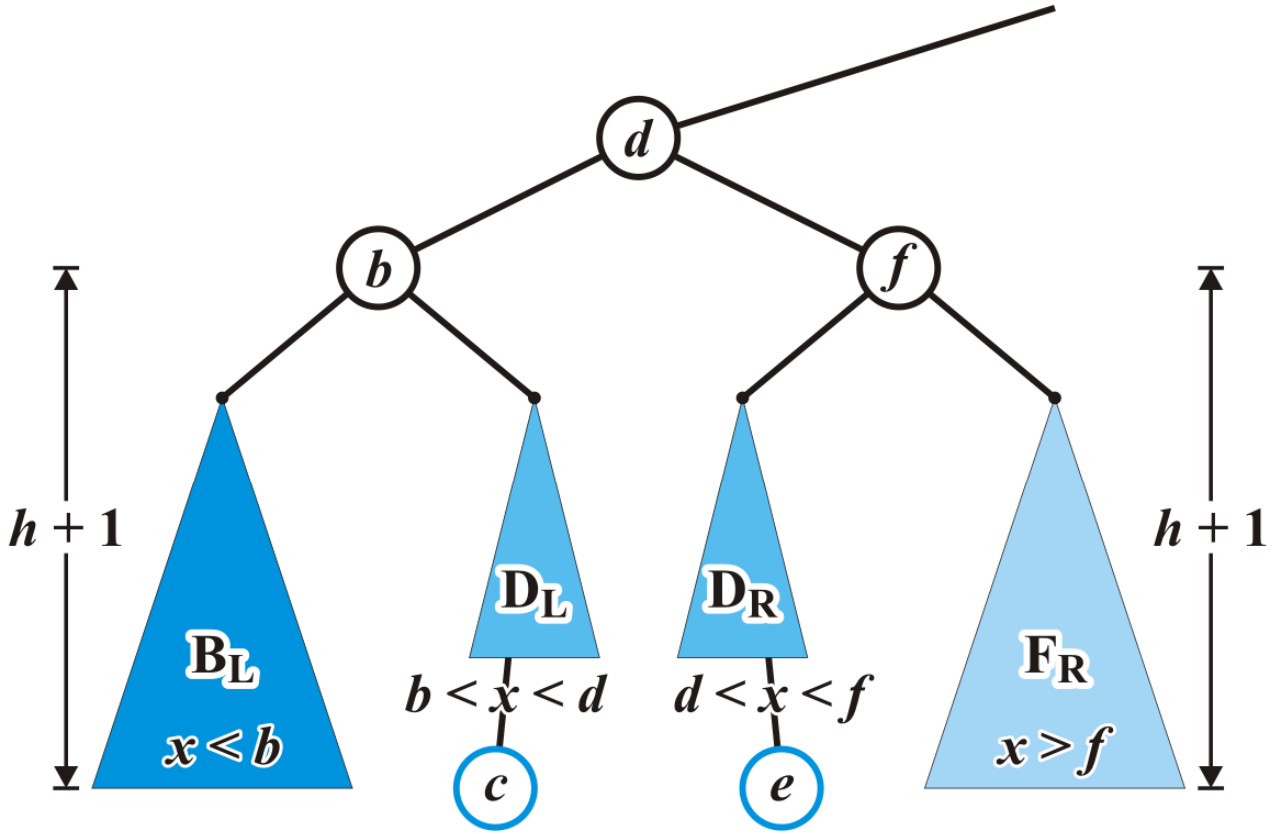
Maintaining Balance: Case Left-Right

```
ref_p.left = ref_d
ref_b.right = ref_DL
ref_f.left = ref_DR
```



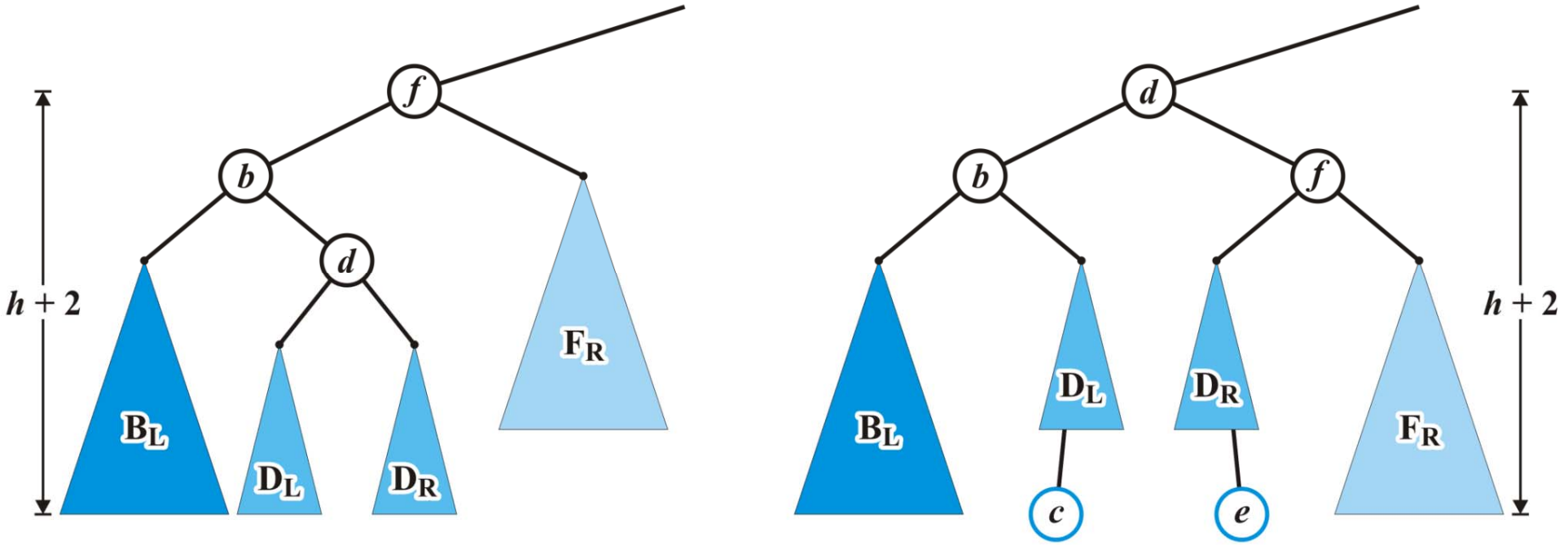
Maintaining Balance: Case Left-Right

Now the tree rooted at d is balanced

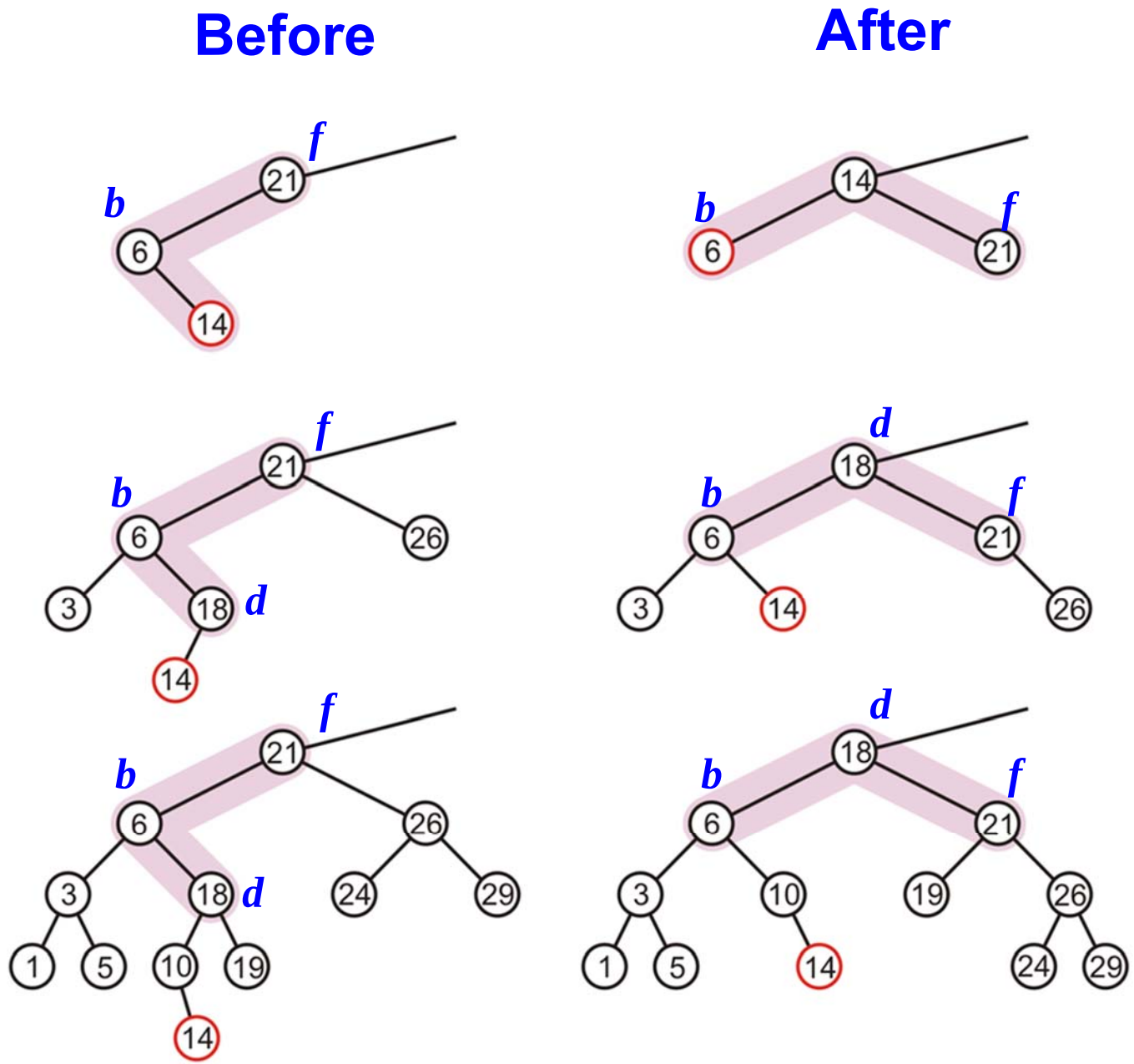


Maintaining Balance: Case Left-Right

Again, the height of the root did not change

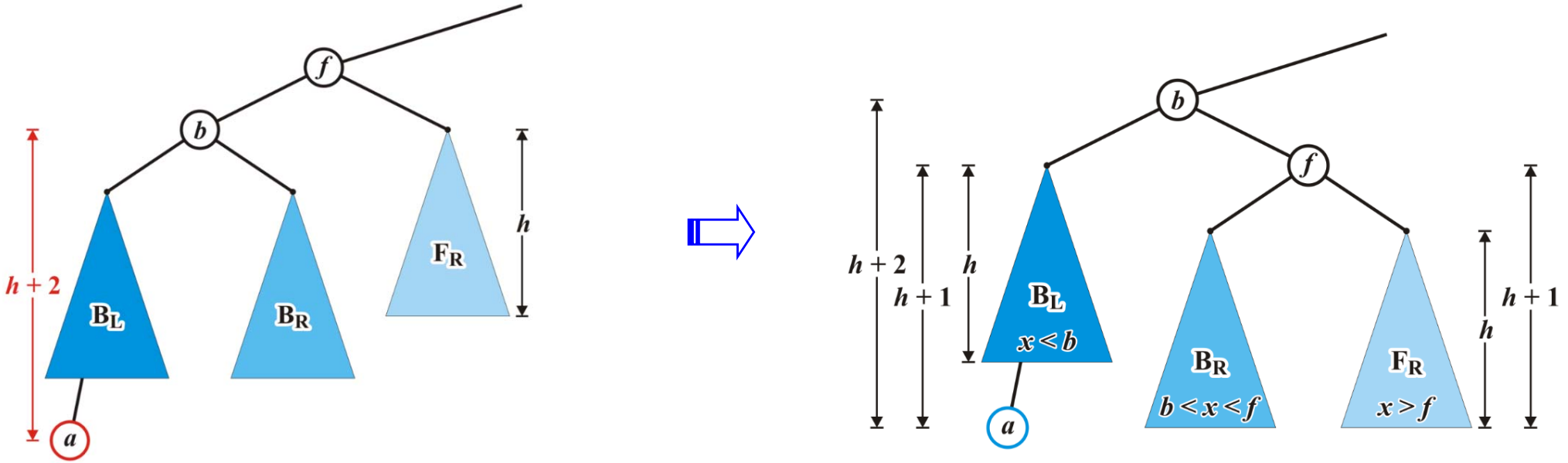


Maintaining Balance: Case Left-Right



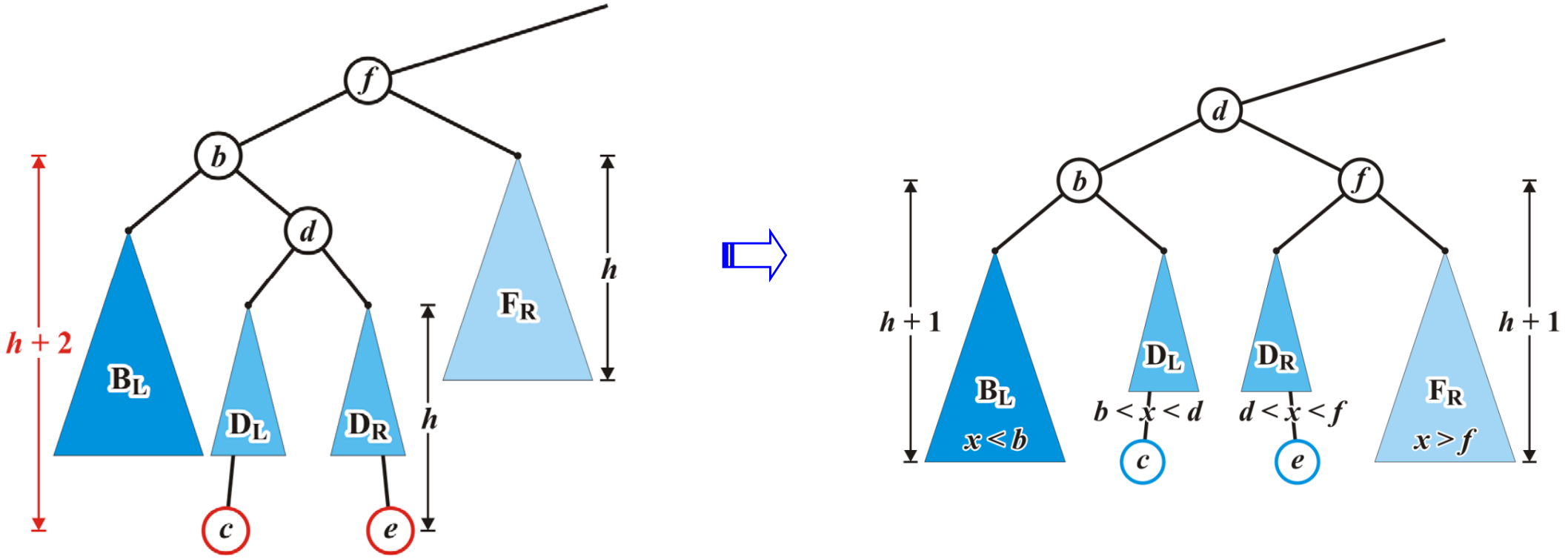
Maintaining balance: Summary

Case Left-Left: promotes first intermediate node b to root



Maintaining balance: Summary

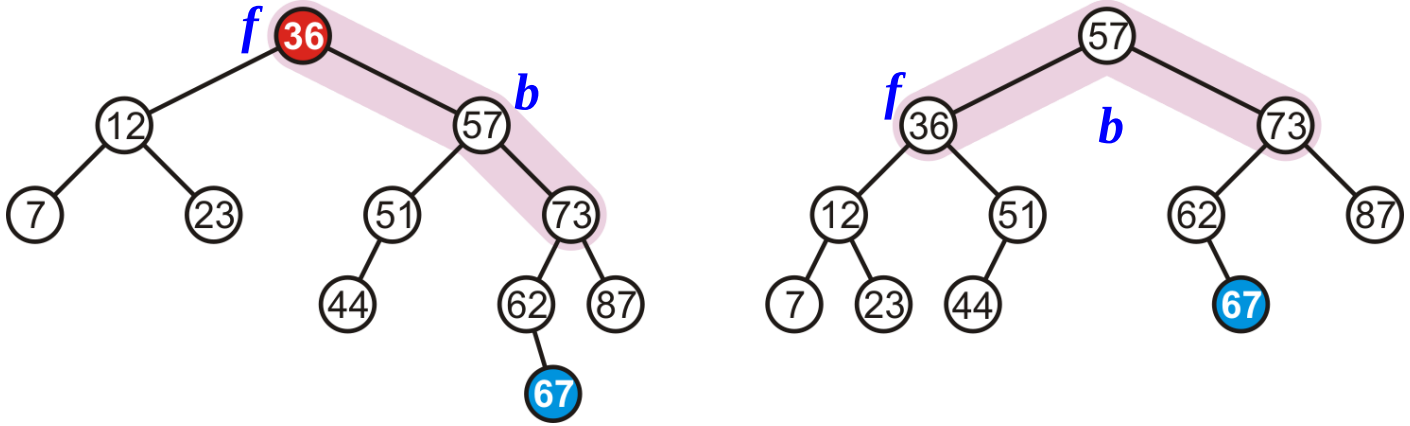
Case Left-Right: promotes second intermediate node d to root



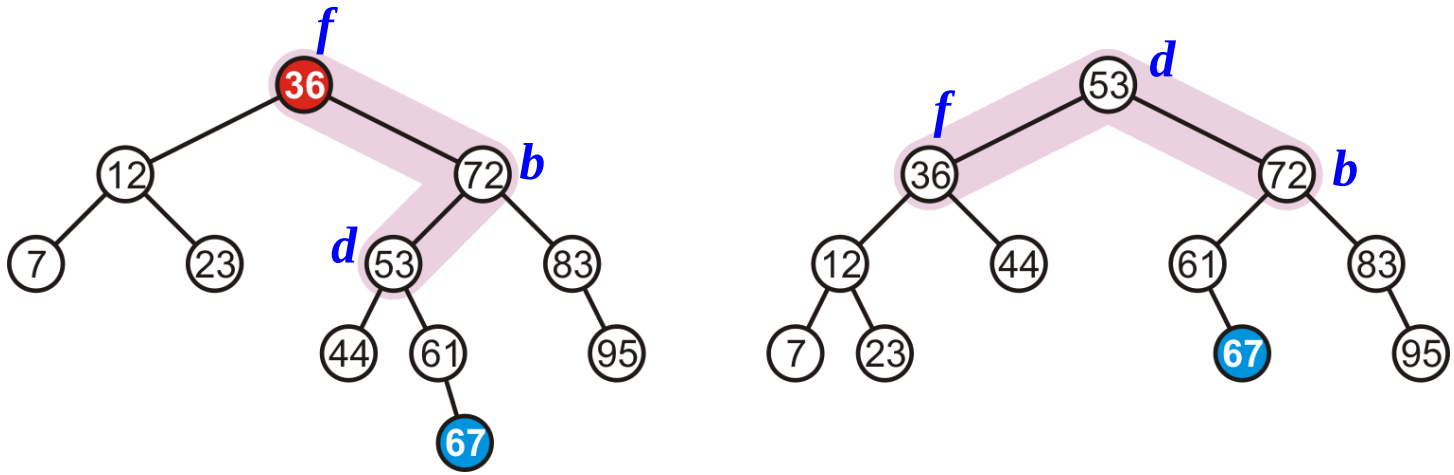
Maintaining balance: Summary

There are two symmetric cases to those we have examined:

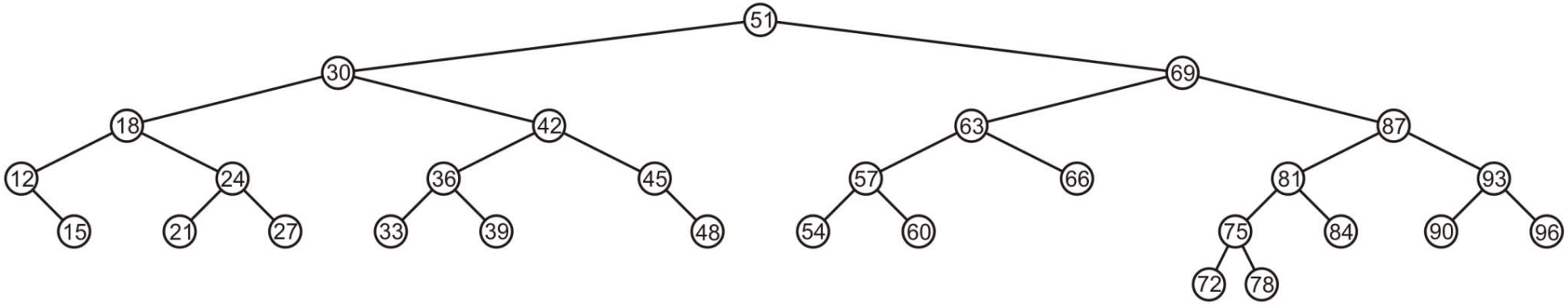
- Insertions into the right-right sub-tree: similar to left-left



- Insertions into either the right-left sub-tree: similar to left-right

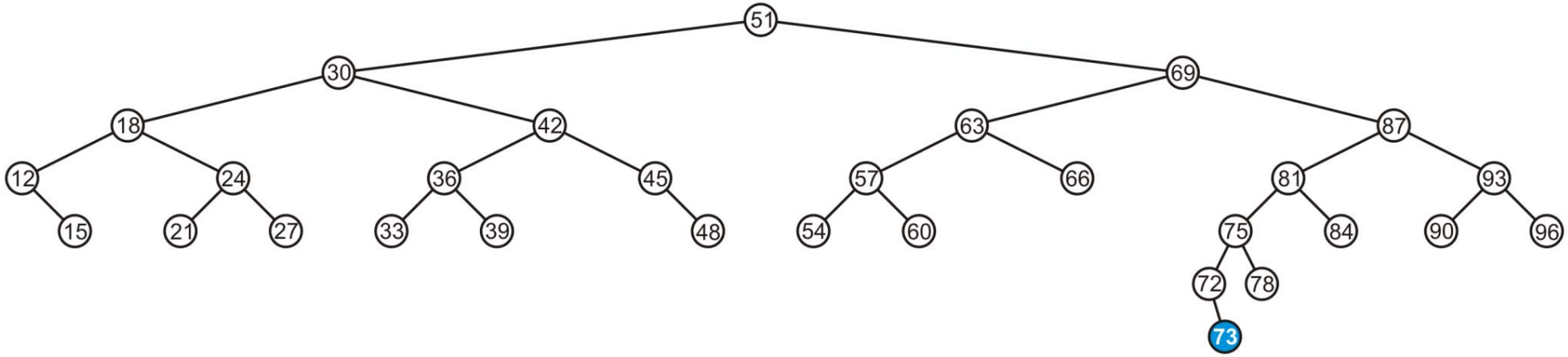


Insertion Example



Insertion Example I

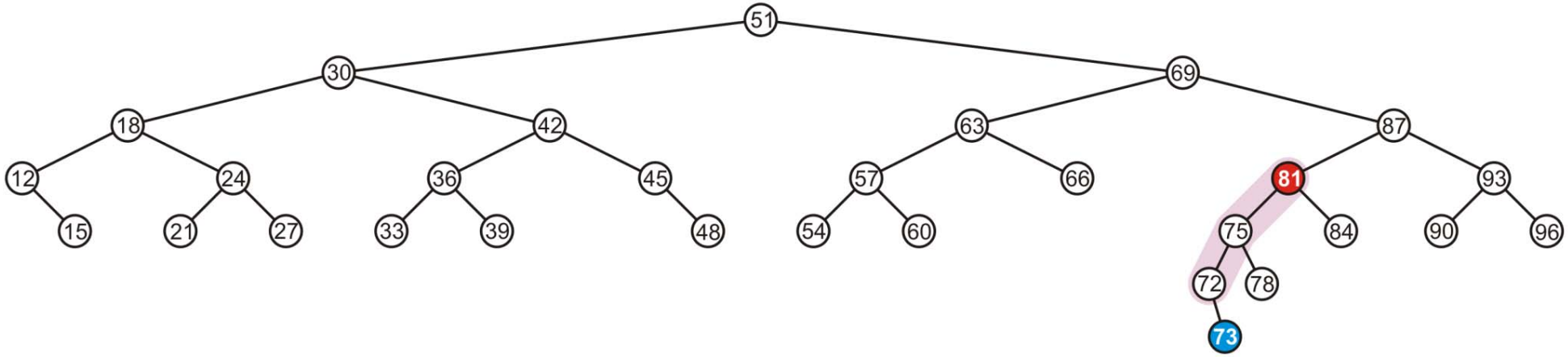
Insert 73



Insertion Example I

The node 81 is unbalanced

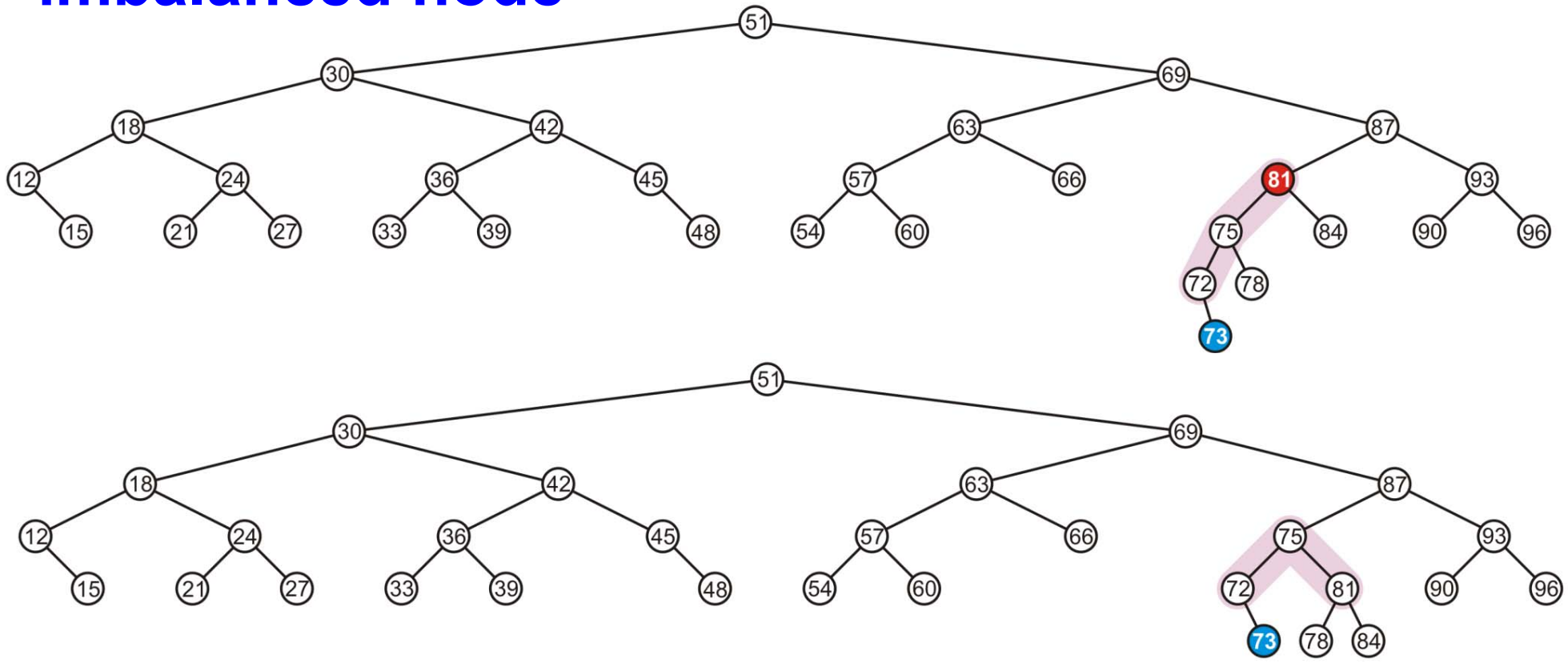
- A left-left imbalance



Insertion Example I

The node 81 is unbalanced

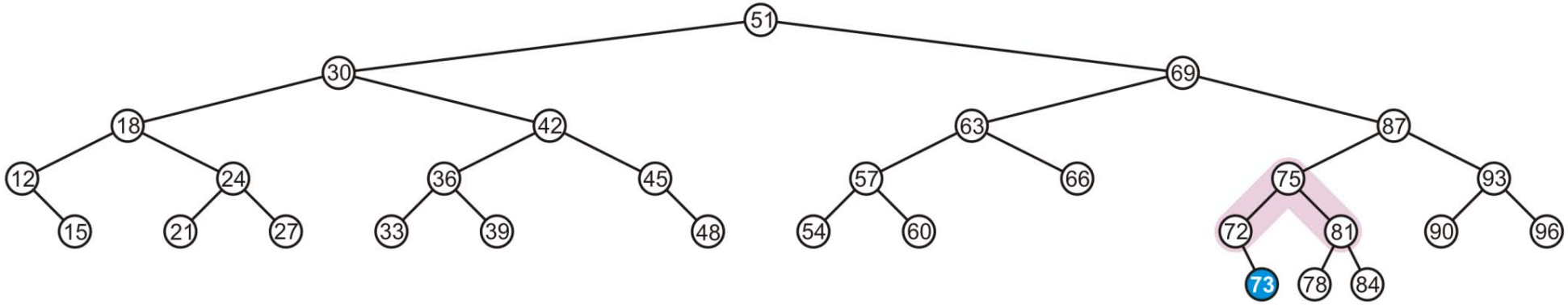
- A left-left imbalance
- Promote the intermediate node 75 to the imbalanced node



Insertion Example I

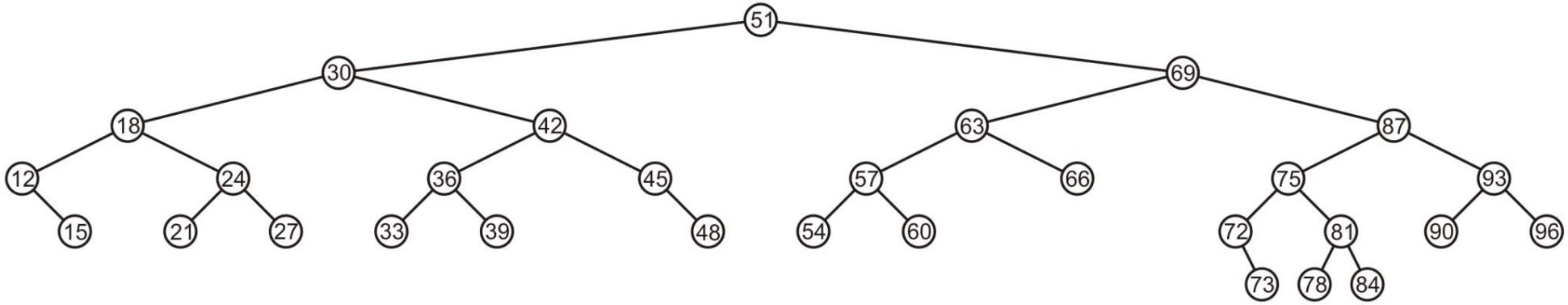
The node 81 is unbalanced

- A left-left imbalance
- Promote the intermediate node 75 to the imbalanced node



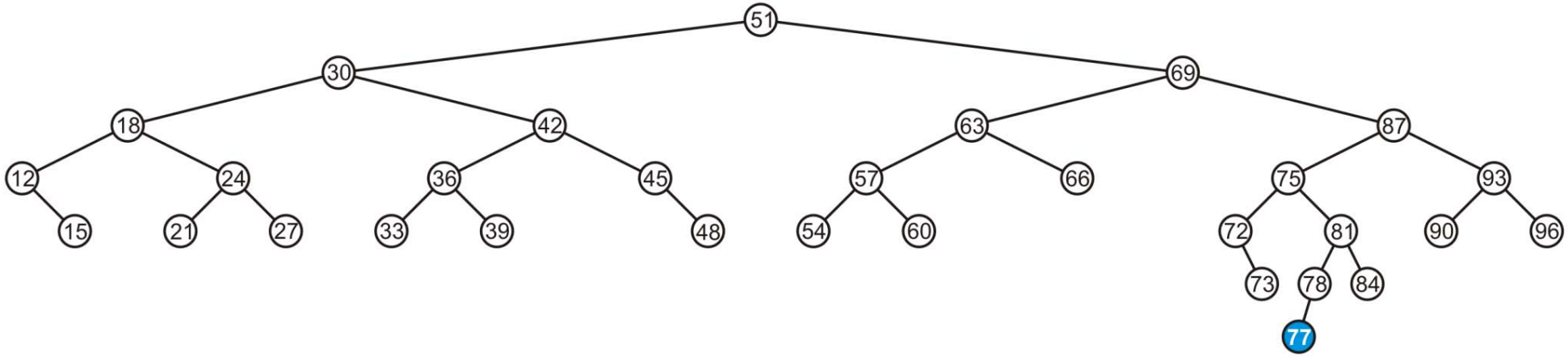
Insertion Example I

The tree is AVL balanced



Insertion Example II

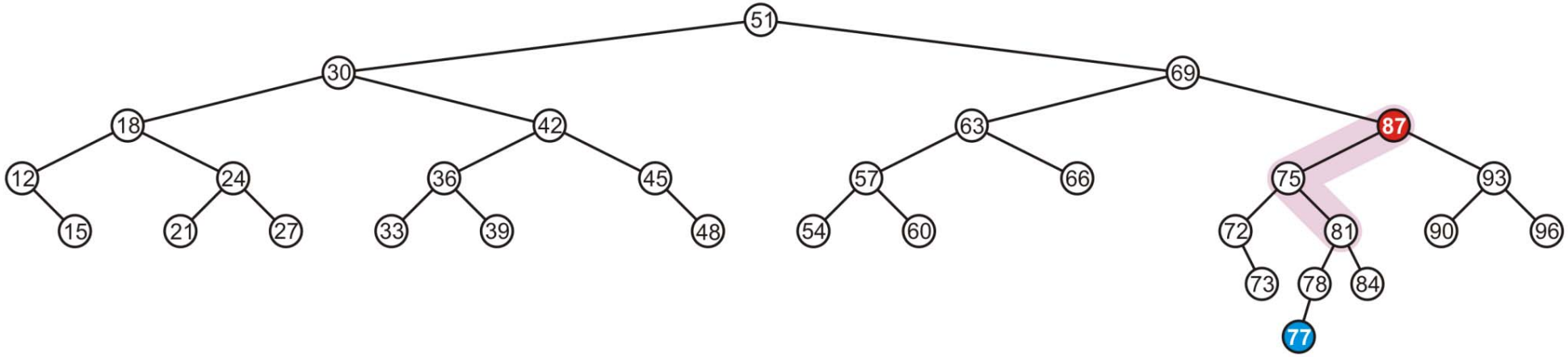
Insert 77



Insertion Example II

The node 87 is unbalanced

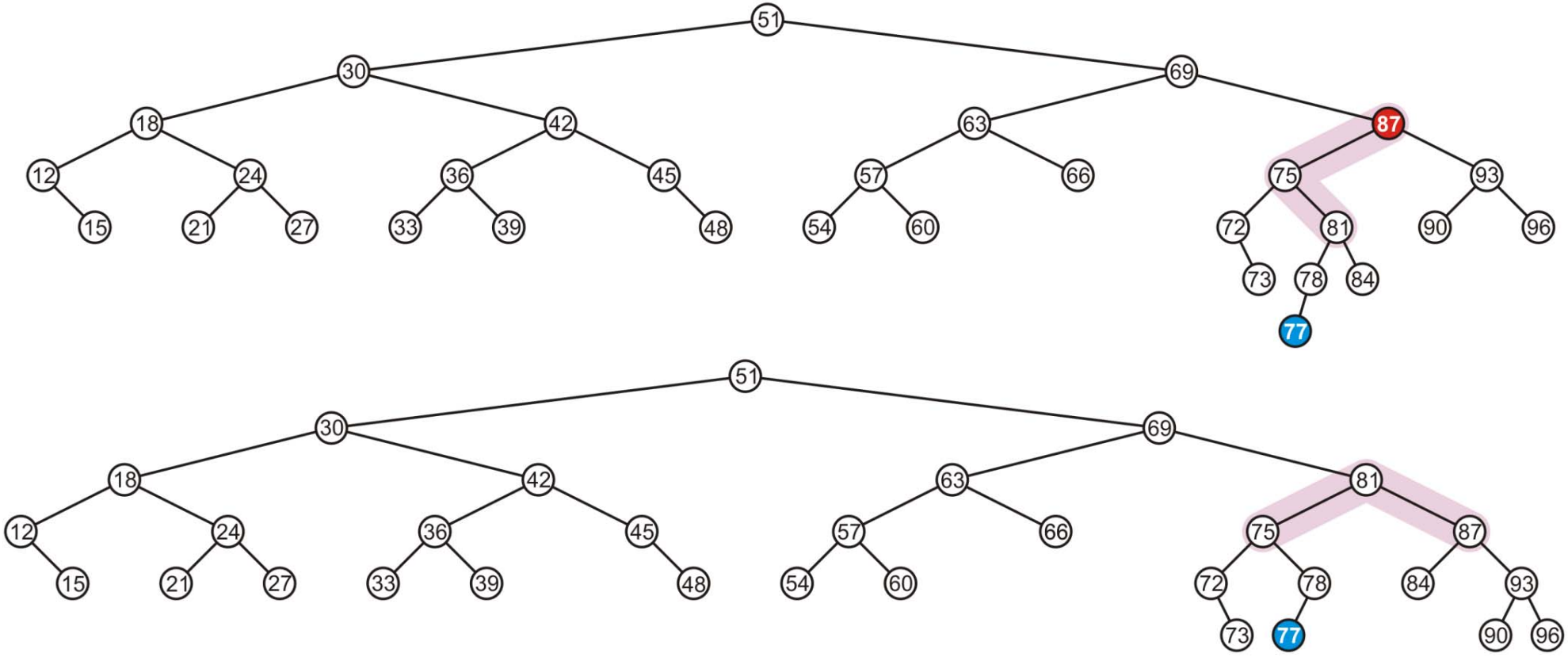
- A left-right imbalance



Insertion Example II

The node 87 is unbalanced

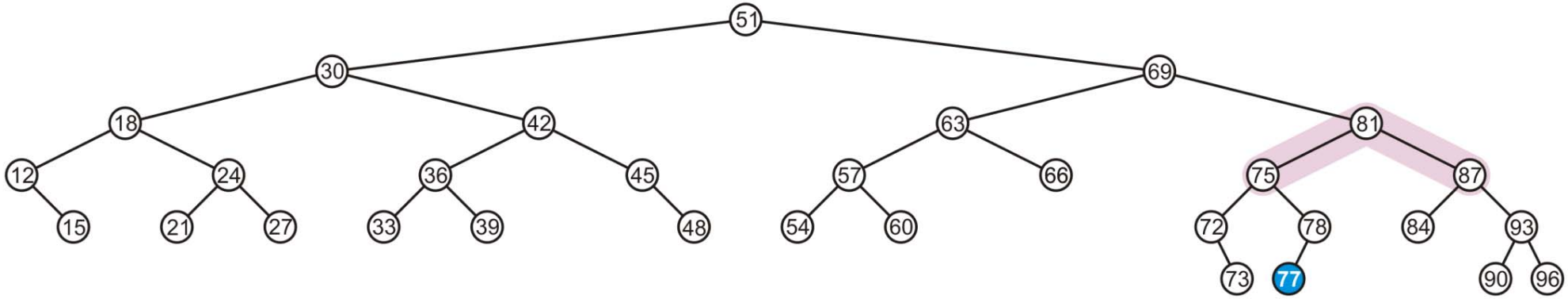
- A left-right imbalance
- Promote the intermediate node 81 to the imbalanced node



Insertion Example II

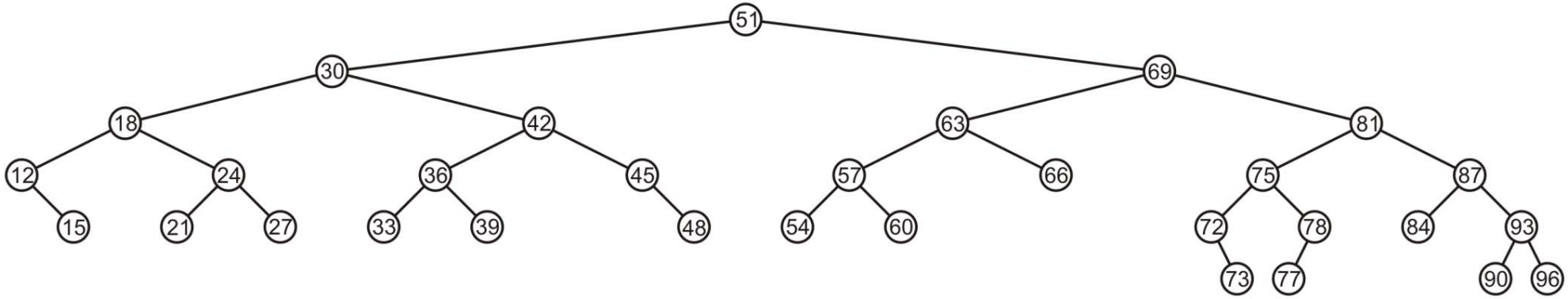
The node 87 is unbalanced

- A left-right imbalance
- Promote the intermediate node 81 to the imbalanced node



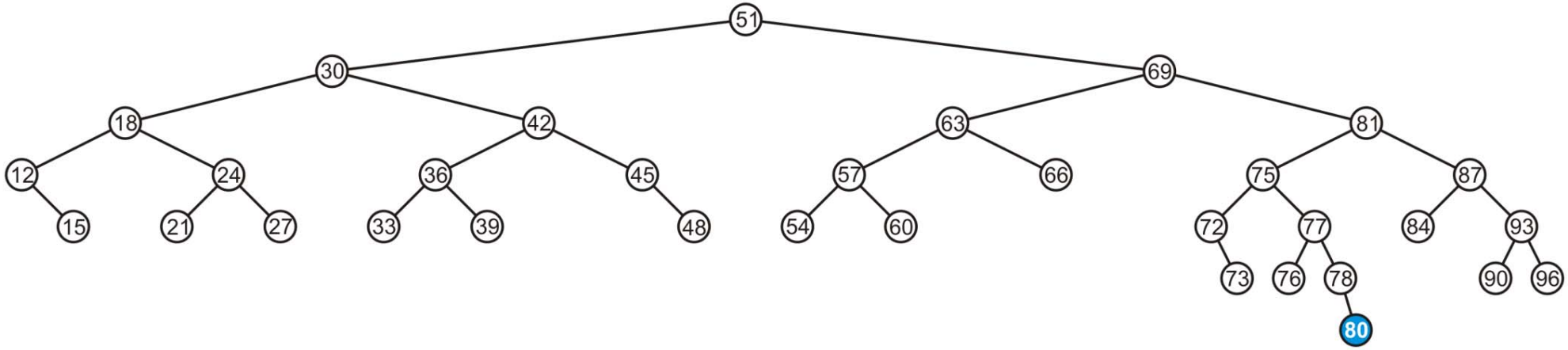
Insertion Example II

The tree is balanced



Insertion Example III

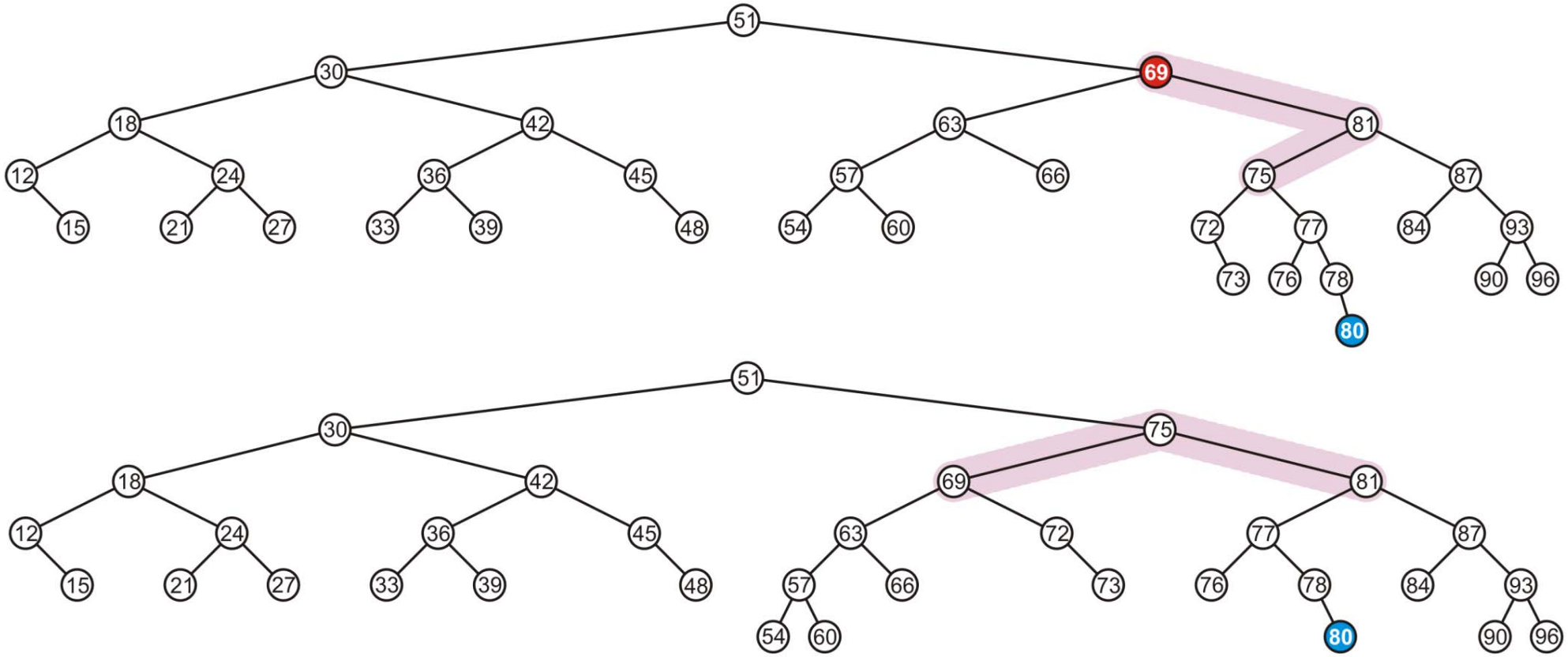
Insert 80



Insertion Example

The node 69 is unbalanced

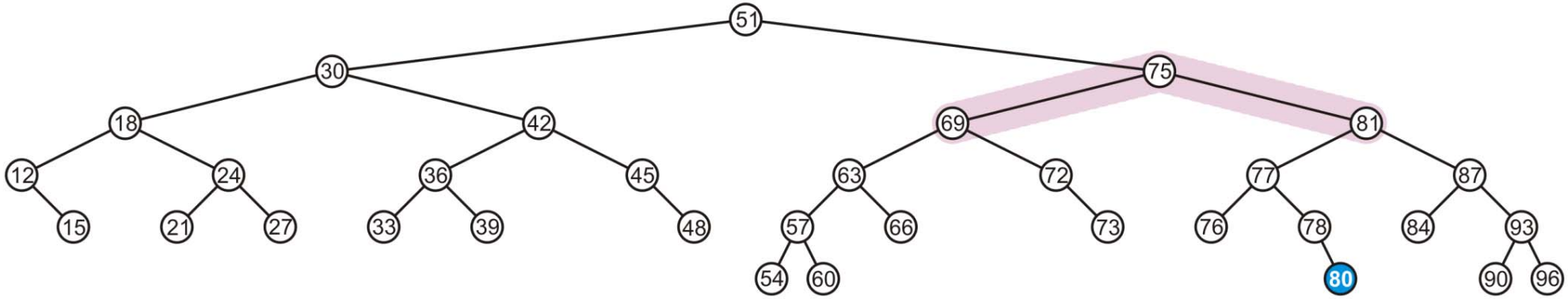
- A right-left imbalance
- Promote the intermediate node 75 to the imbalanced node



Insertion Example III

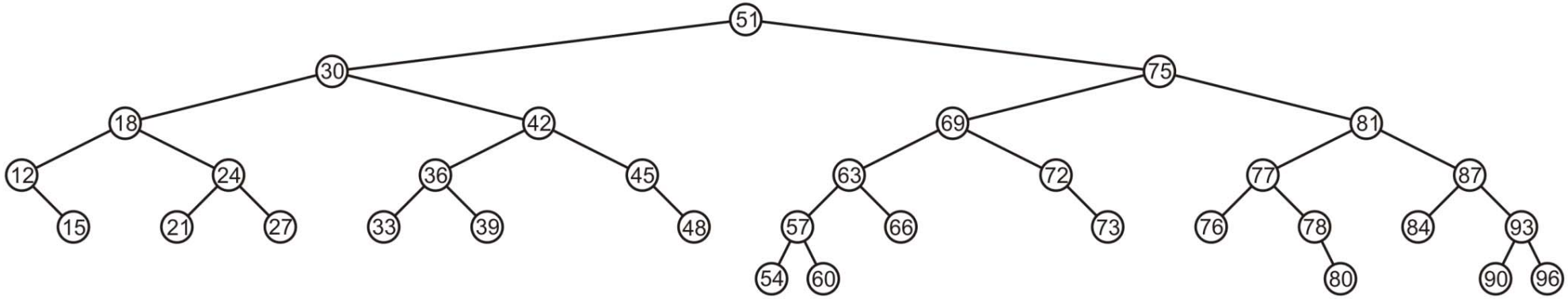
The node 69 is unbalanced

- A left-right imbalance
- Promote the intermediate node 75 to the imbalanced node



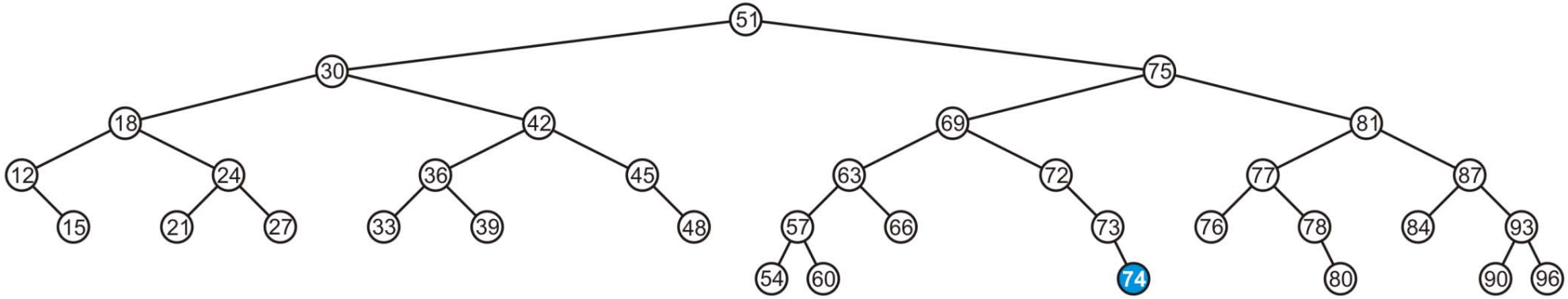
Insertion Example III

Again, balanced



Insertion Example IV

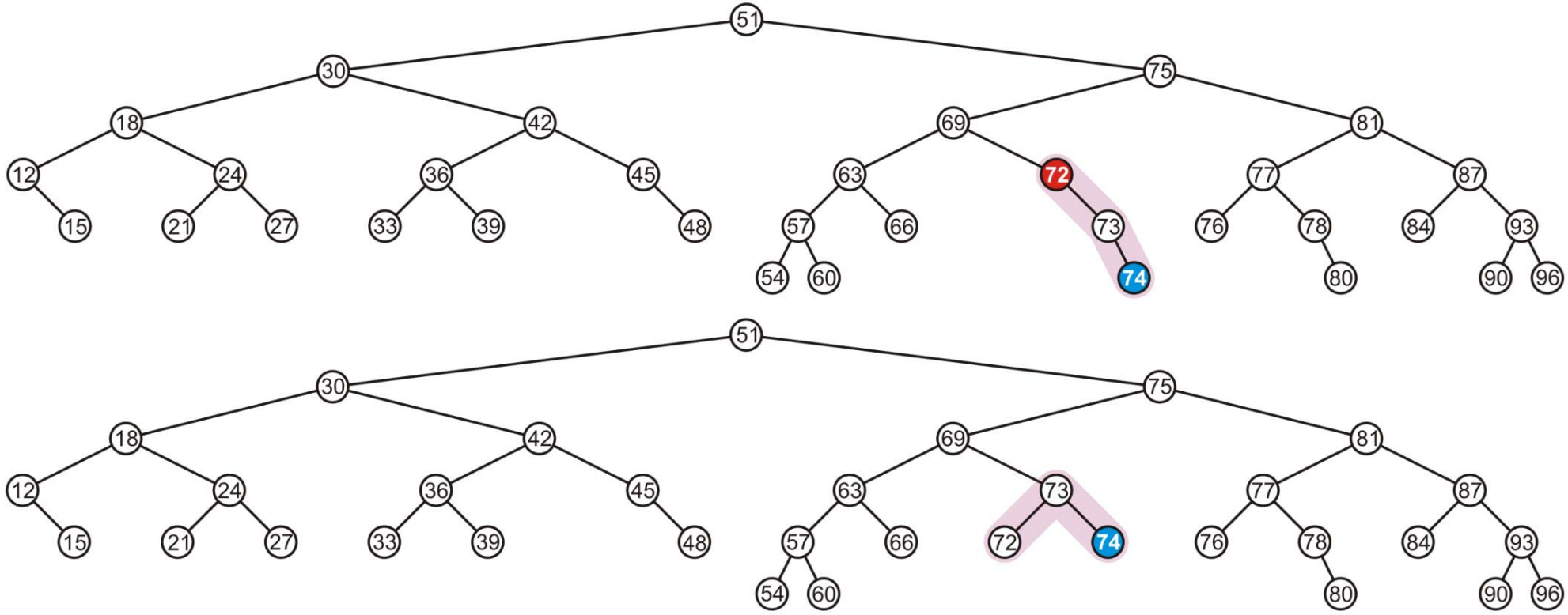
Insert 74



Insertion Example IV

The node 72 is unbalanced

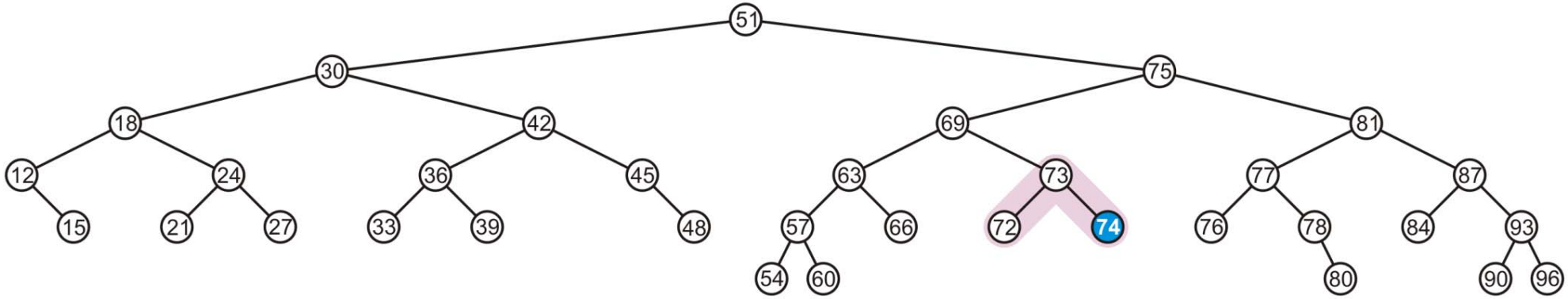
- A right-right imbalance
- Promote the intermediate node 73 to the imbalanced node



Insertion Example IV

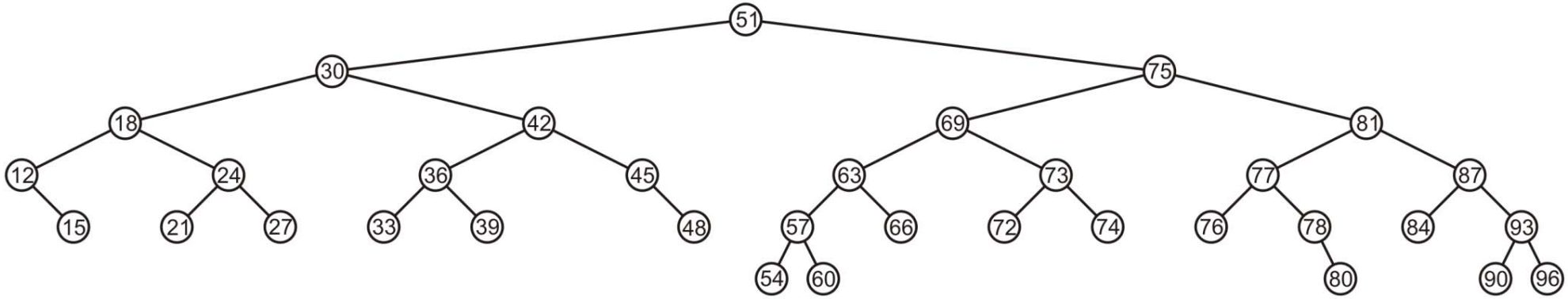
The node 72 is unbalanced

- A right-right imbalance
- Promote the intermediate node to the imbalanced node



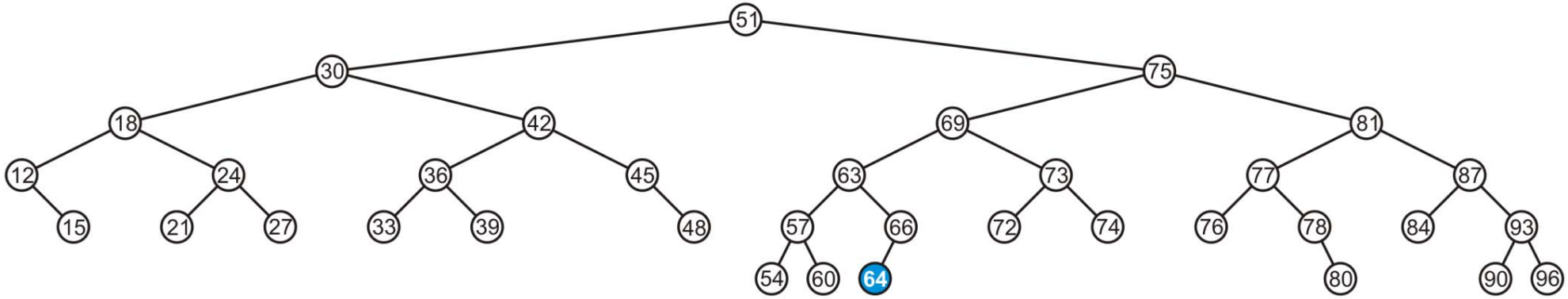
Insertion Example IV

Again, balanced



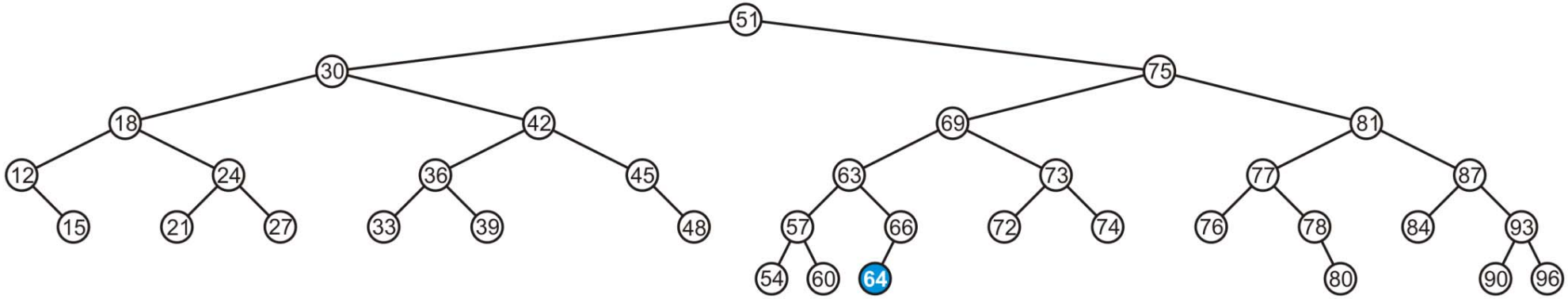
Insertion Example V

Insert 64



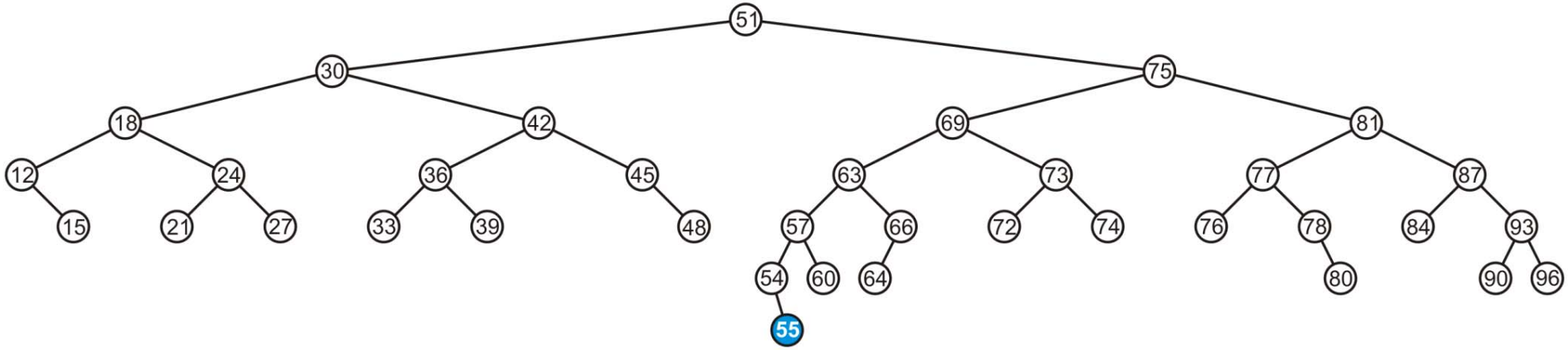
Insertion Example V

This causes no imbalances



Insertion Example VI

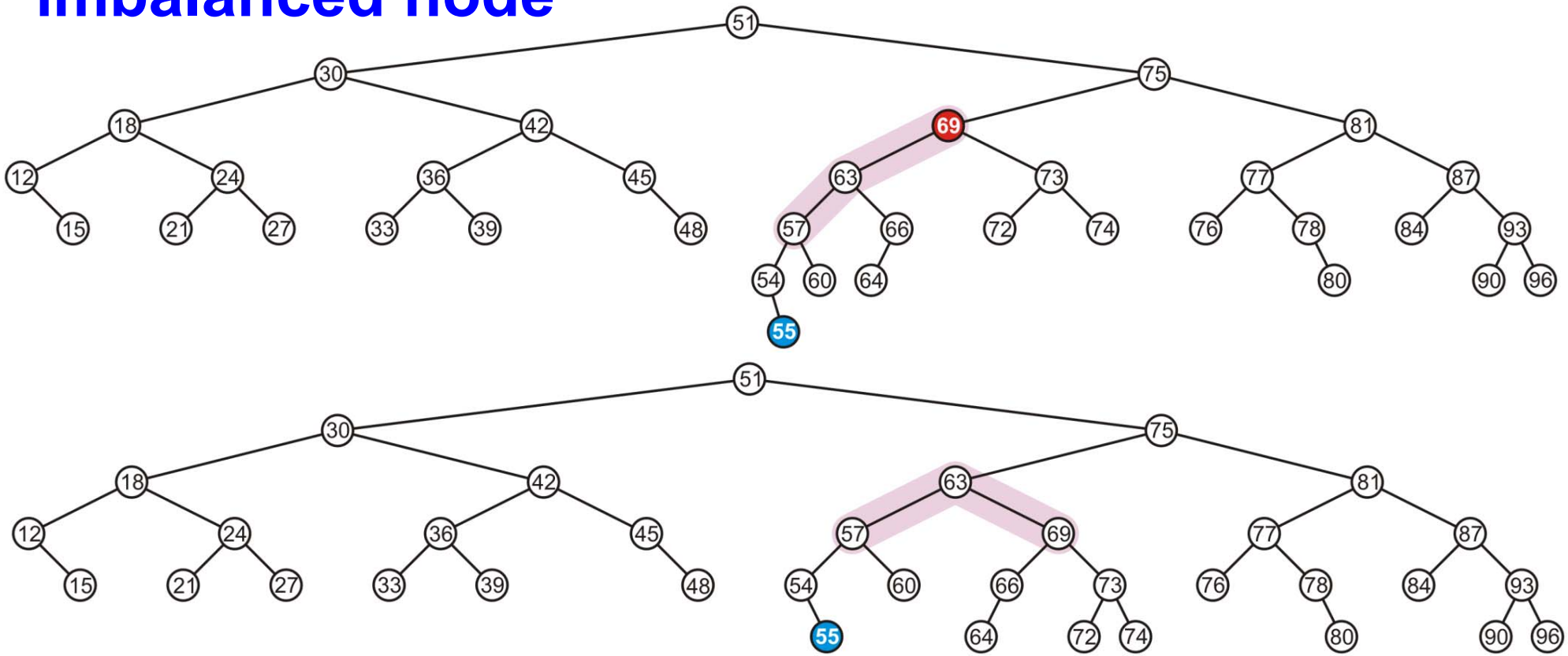
Insert 55



Insertion Example VI

The node 69 is imbalanced

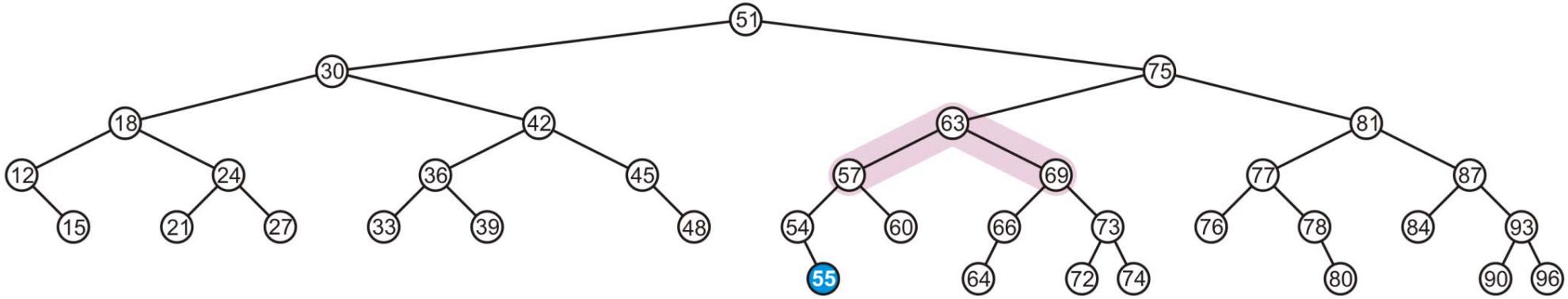
- A left-left imbalance
- Promote the intermediate node 63 to the imbalanced node



Insertion Example VI

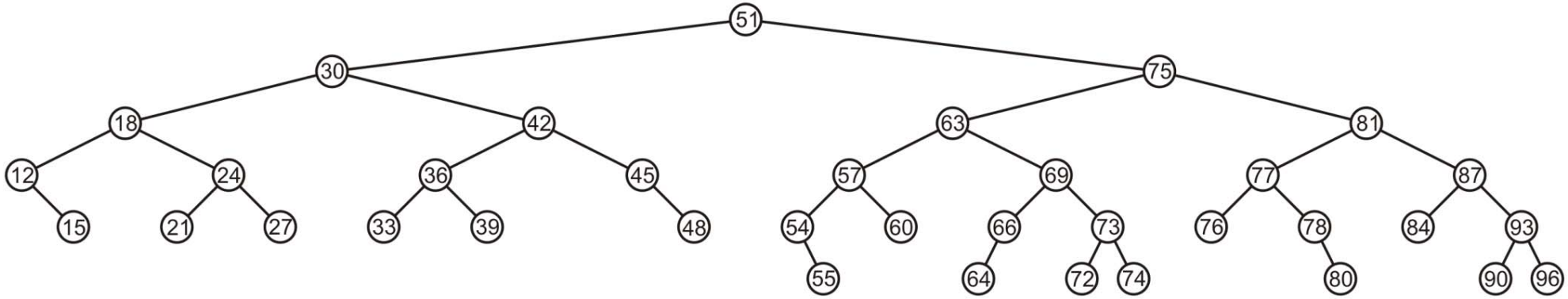
The node 69 is imbalanced

- A left-left imbalance
- Promote the intermediate node 63 to the imbalanced node



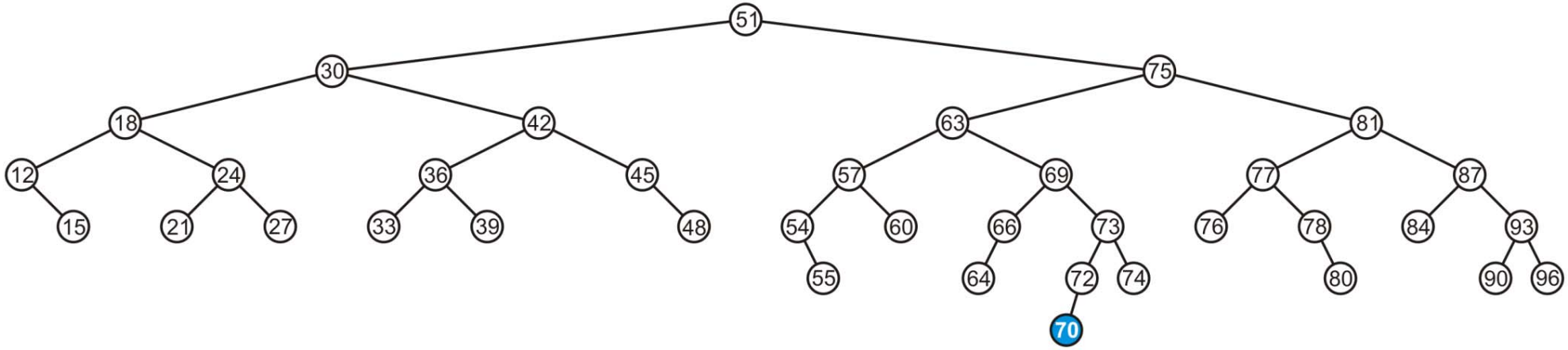
Insertion Example VI

The tree is now balanced



Insertion Example VII

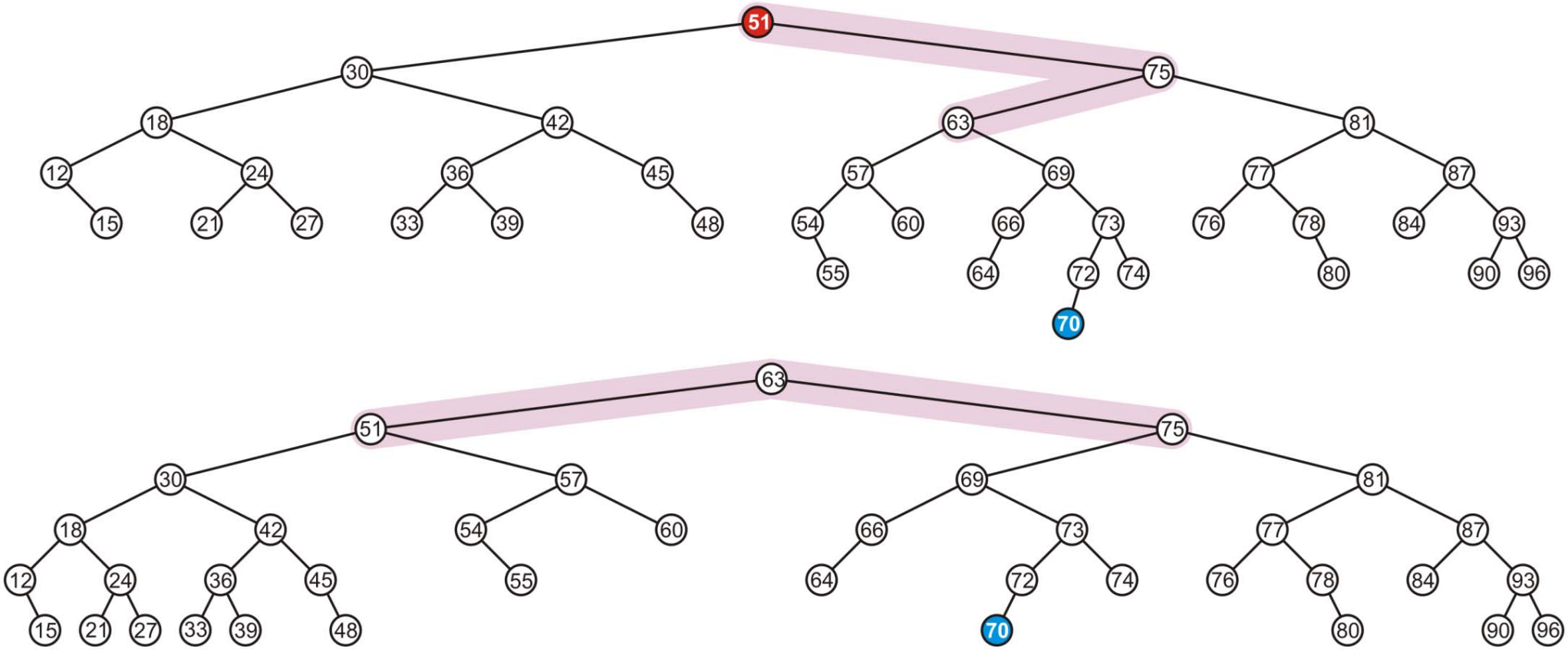
Insert 70



Insertion Example VII

The root node is now imbalanced

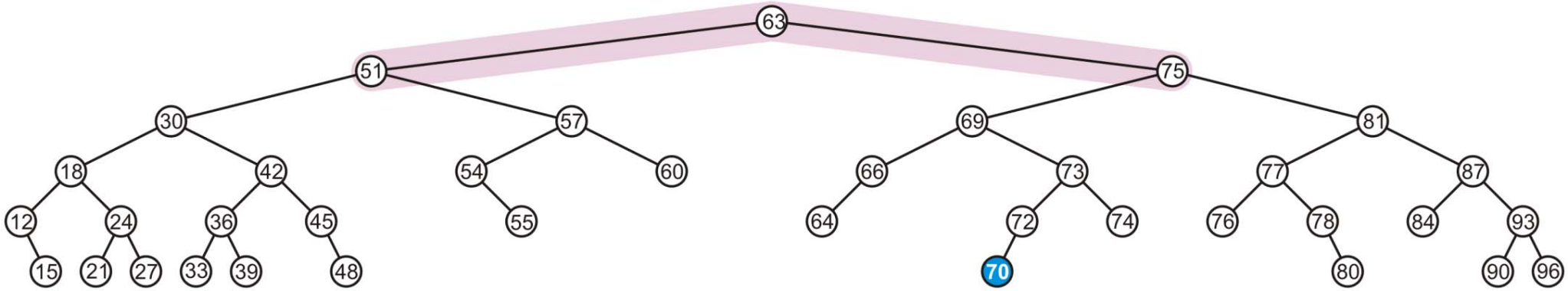
- A right-left imbalance
- Promote the intermediate node 63 to the root



Insertion Example VII

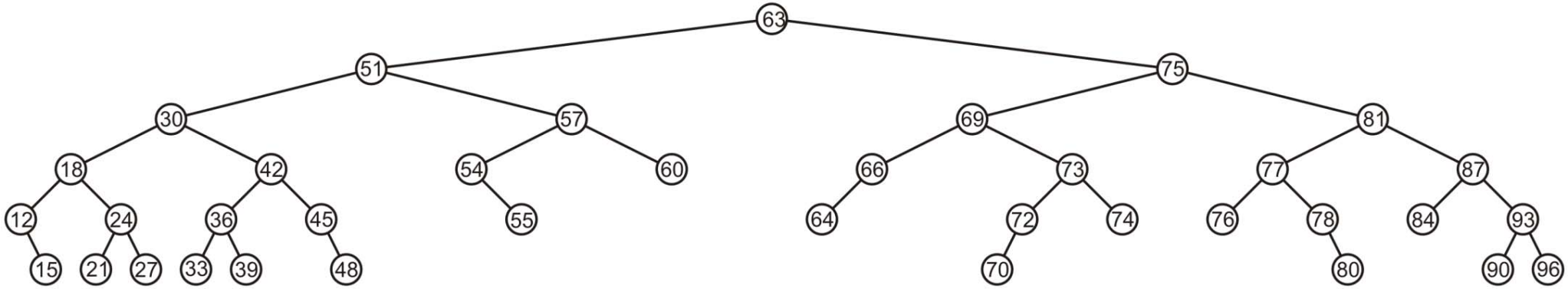
The root node is imbalanced

- A right-left imbalance
- Promote the intermediate node 63 to the root

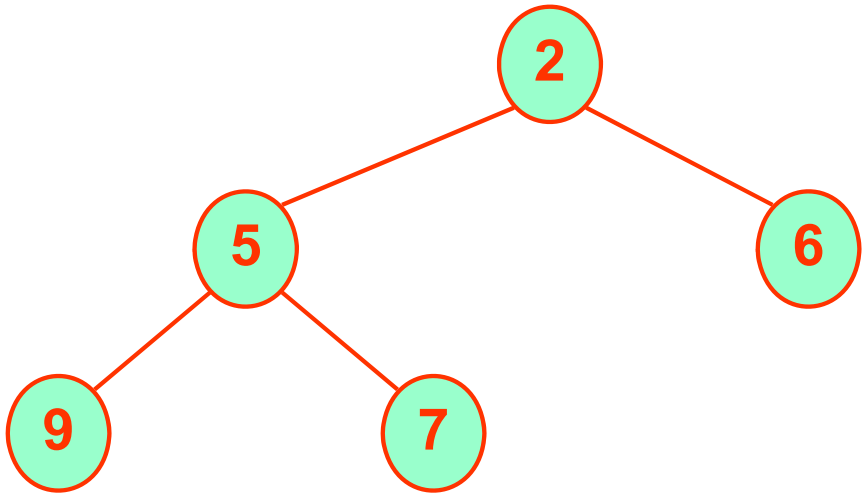


Insertion Example VII

The result is AVL balanced



Heaps



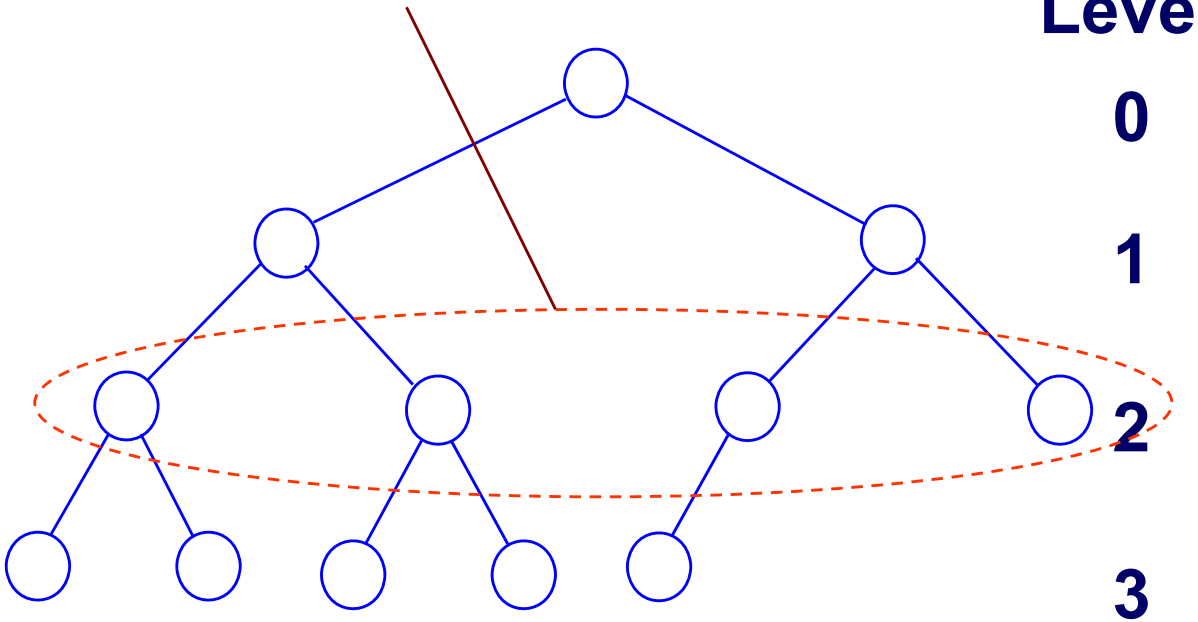
Defining a heap structure

- A *heap structure* is a binary tree in which all levels, except possibly the last (bottom) level, have as many nodes as possible. On the last level, all of the leaves are at the left.
  - ☞ *This is a Heap Structural Property*

- ✓ A heap must be a *nearly complete* binary tree.
- ✓ levels *0, 1, 2, …, h-1* (except the last level) have the maximum number of the nodes possible (i.e., level *i* has 2^i node for $0 \le i \le h-1$
- ✓ The lowest level is filled from the left up to a point

Binary Heap: Example

Level *i* has 2^i *nodes*

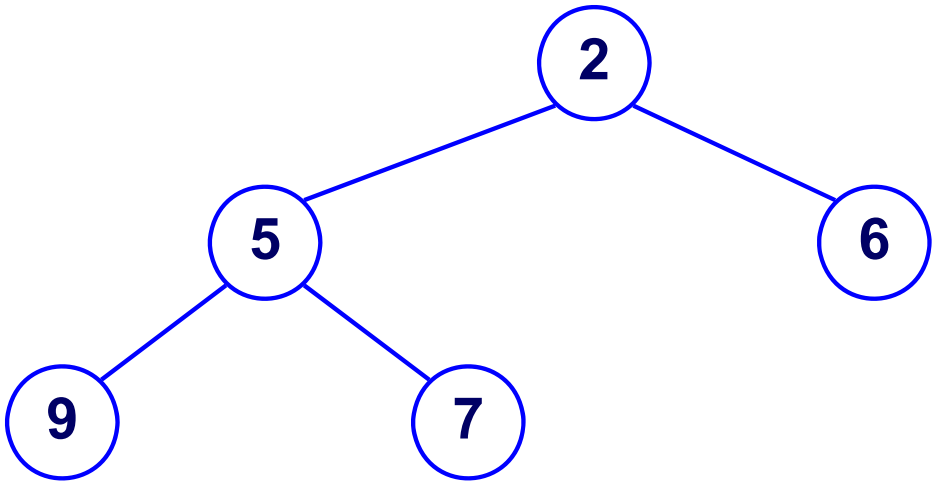


- ✓ All levels, except the last level, 3, have as many nodes as possible.
- ✓ On the last but one level, there is at most one node with one child, which must be a left child
- ✓ No vacancies at level 0 to n-1

- ✓ Levels 0 (the root), 1, and 2 (the next-to-last level) have as many nodes as possible.
- ✓ On level 3 (the last level), there is at most one node with no sibling, which must be a left child --- also called left-complete binary tree

A binary minheap: Definition

- A binary *minheap* is a heap structure in which values are assigned to the nodes so that the value of each node is less than or equal to the values of its children (if any).
  - ☞ *This is a Heap Order Property*

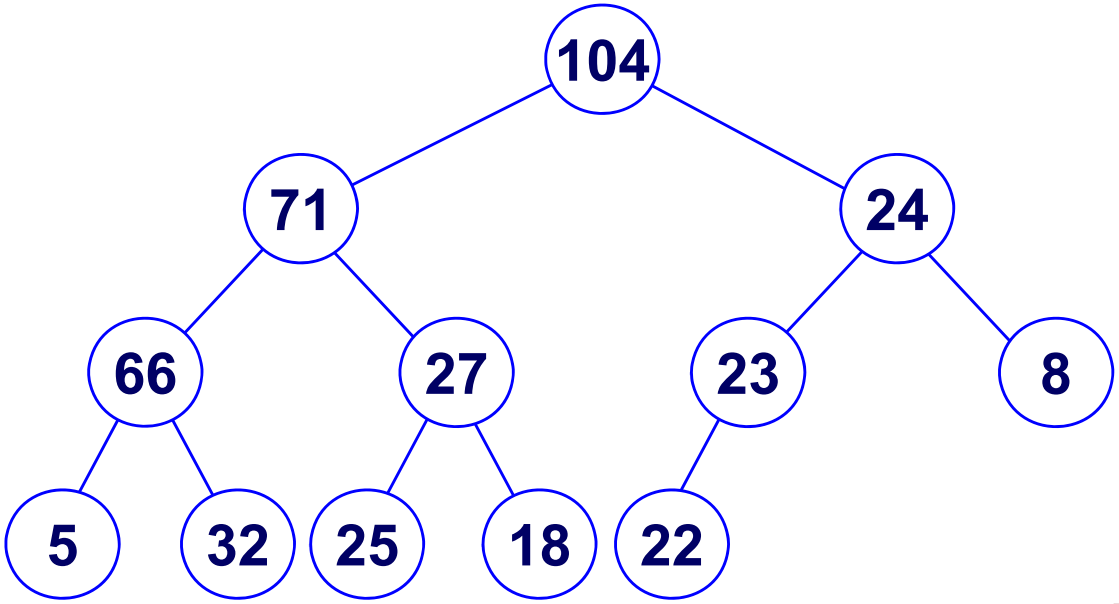


Minheap property:
value of each node ≤ value of its children

✓ A minheap must maintain two properties: structural and order property

A binary maxheap: Definition

- A binary *maxheap* is a heap structure in which values are assigned to the nodes so that the value of each node is greater than or equal to the values of its children (if any)



Maxheap property:
value of each node ≥ value of its children

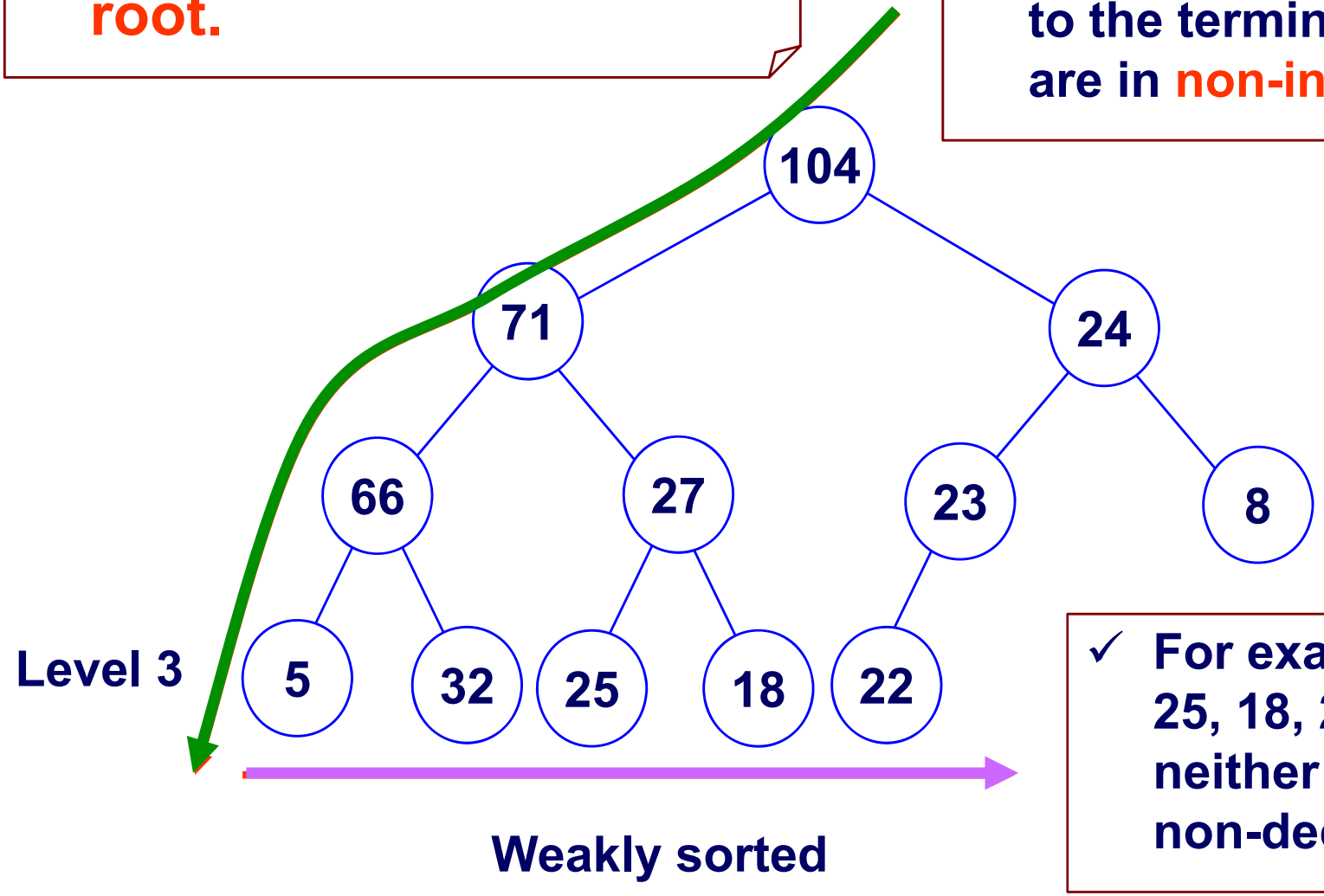
"Weakly sorted" Heap

- A maxheap is "weakly sorted" in the sense that the values along a path from the root to a terminal node are in non-increasing order.
- At the same time, the values along a *level* are, in general, in no particular order

"Weakly sorted" maxheap

- ✓ In a maxheap, the maximum value is at the root.

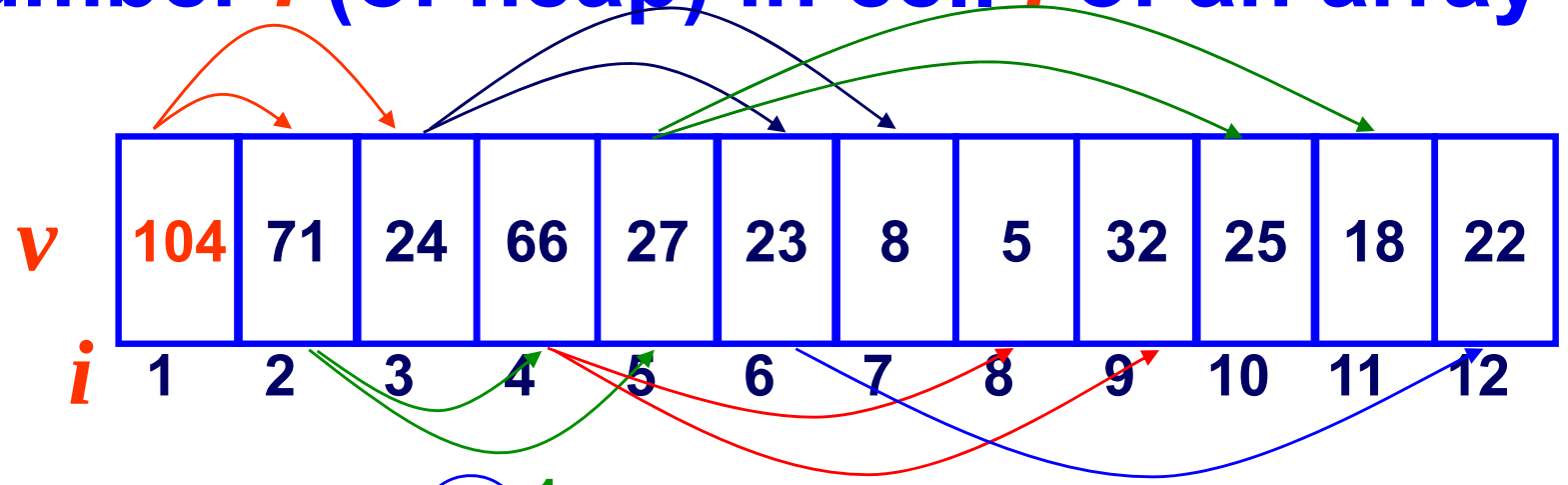
- ✓ For example, the values 104, 71, 66, 5, along the path from the root to the terminal node with value 5, are in non-increasing order.

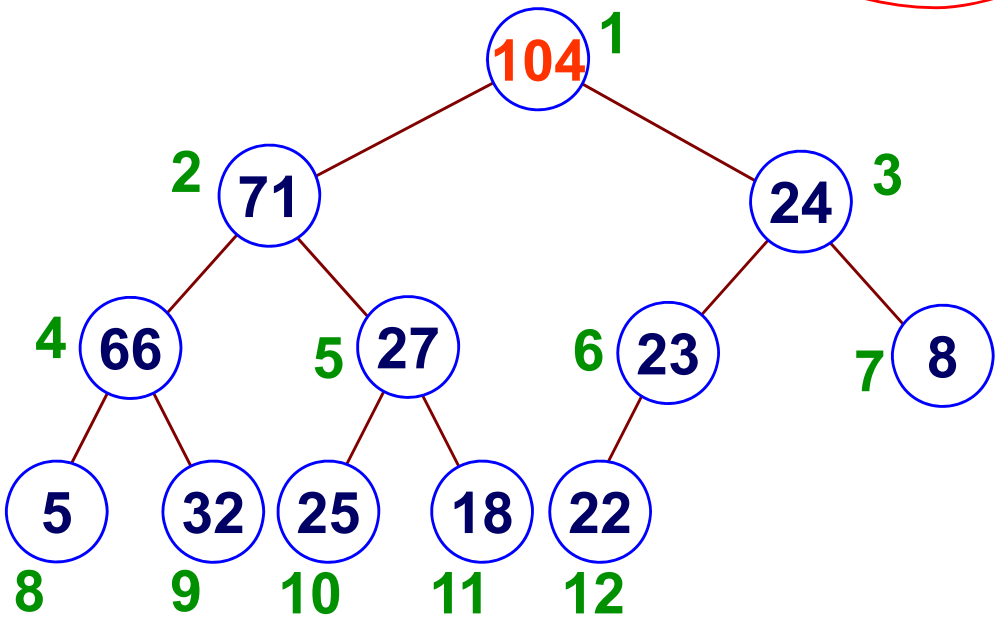


- ✓ For example, the values 5, 32, 25, 18, 22 on level 3 are in neither non-increasing nor non-decreasing order.

Representing a heap using An Array

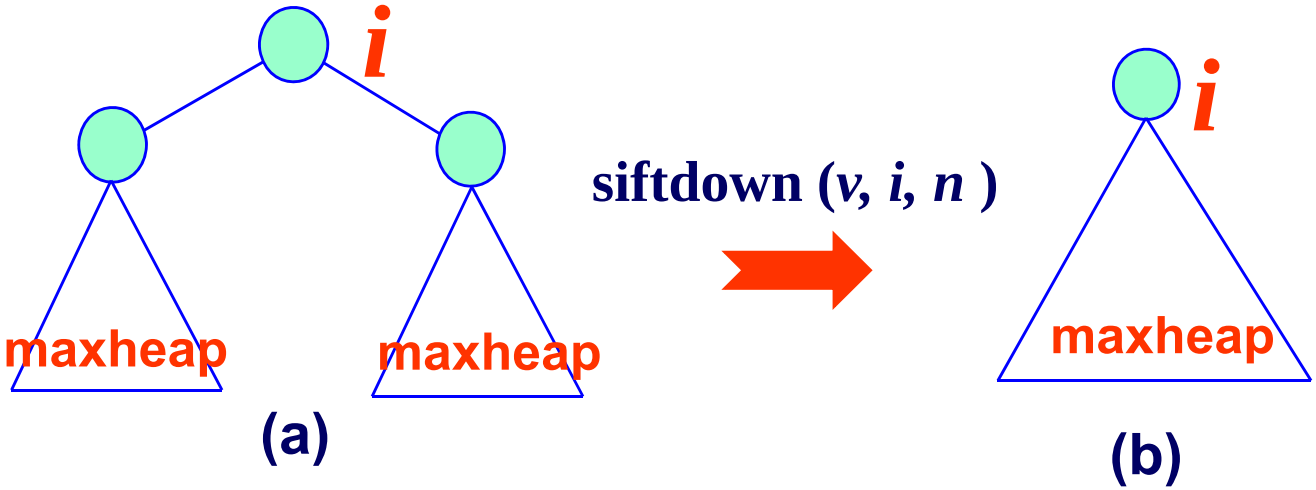
- To represent a heap, we store the value in node number *i* (of heap) in cell *i* of an array





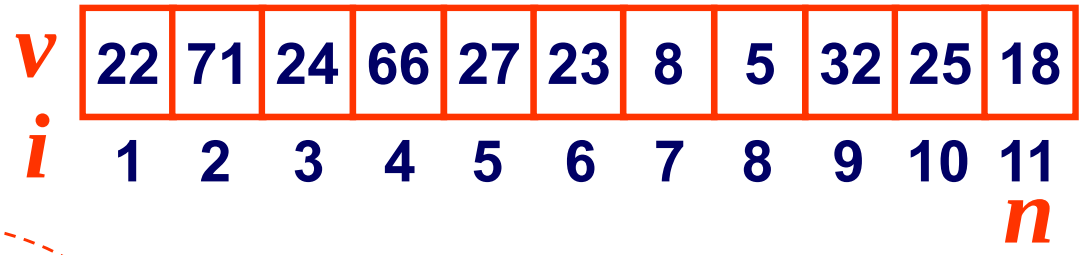
Algorithm siftdown

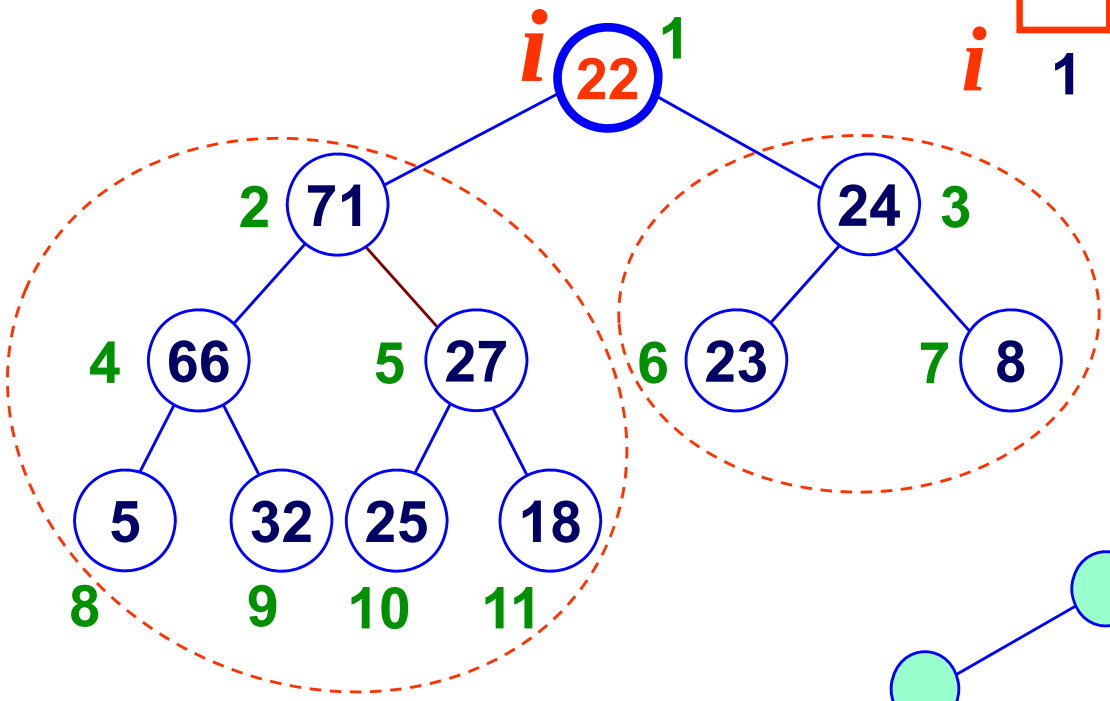
- **Assume:**
  - Overall a heap structure rooted at *i*, but the order property does not hold
  - For node *i*, the left subtree and right subtree are maxheap
  - *v* is an array [*1* ..*n*] representing a heap structure

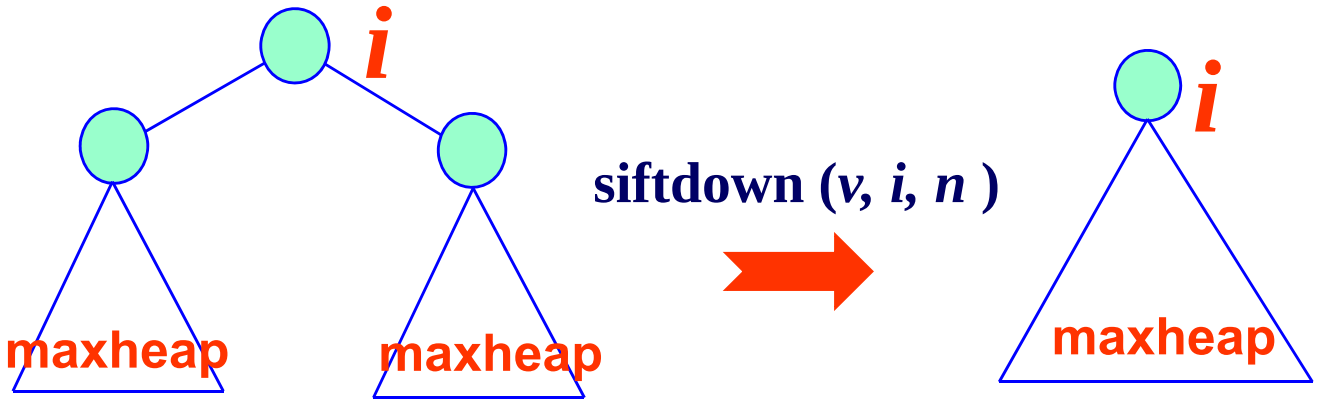


Algorithm siftdown: Example

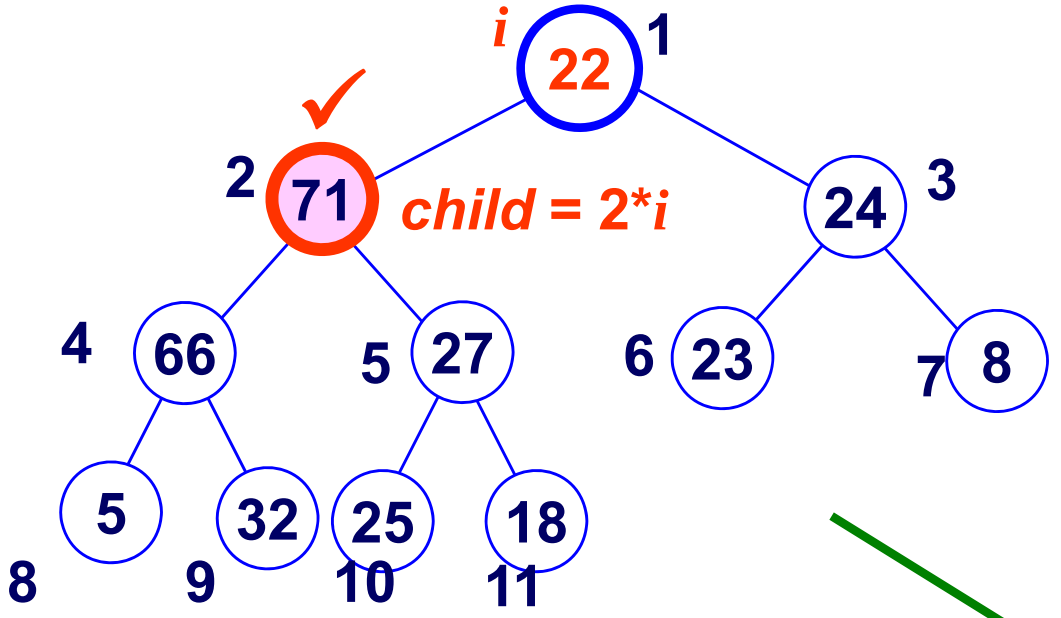
- Idea: repeatedly swap a value with its larger child.



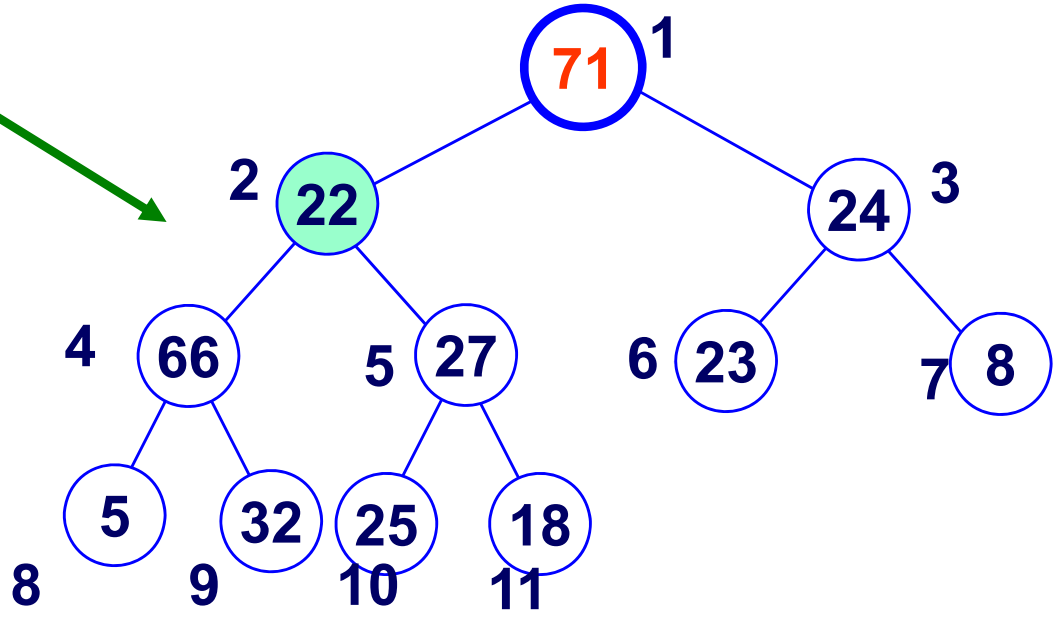




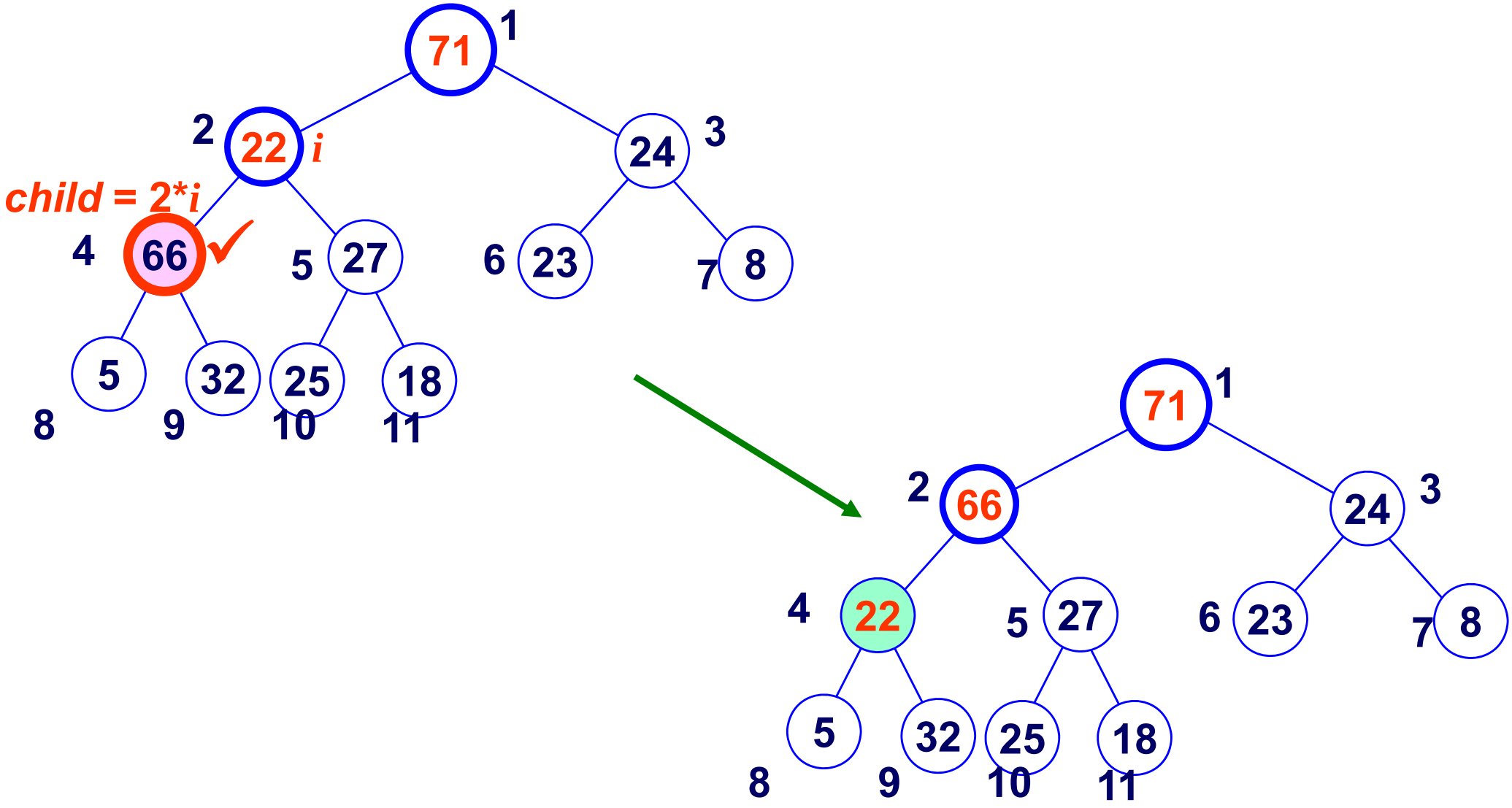
Repeatedly Swapping (Siftdown)



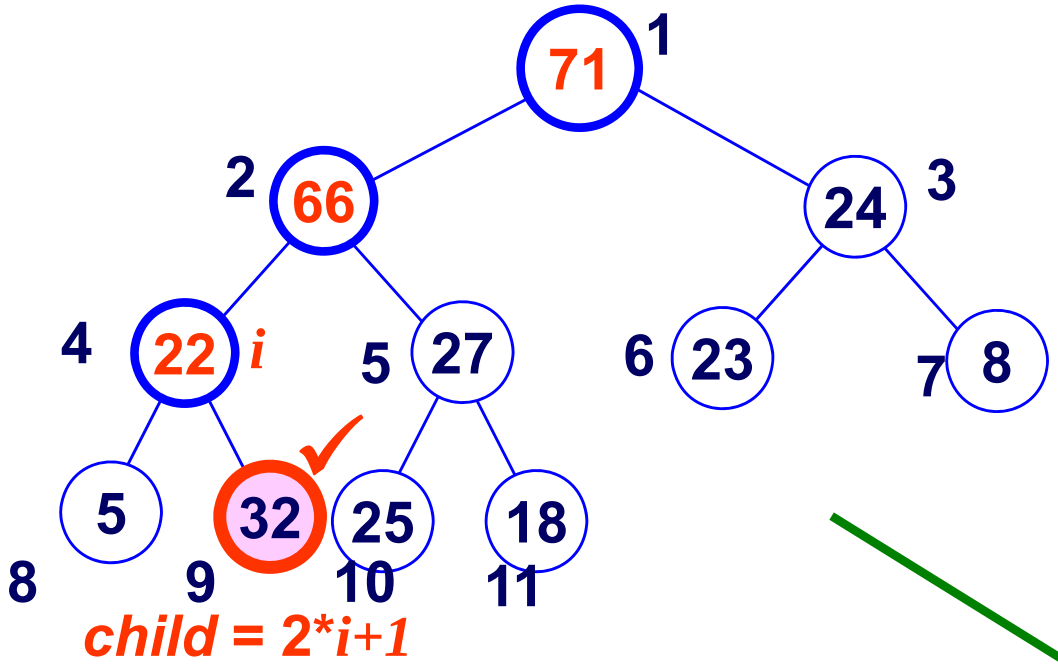
1. Find larger child
2. Swap



Repeatedly Swapping (Siftdown)

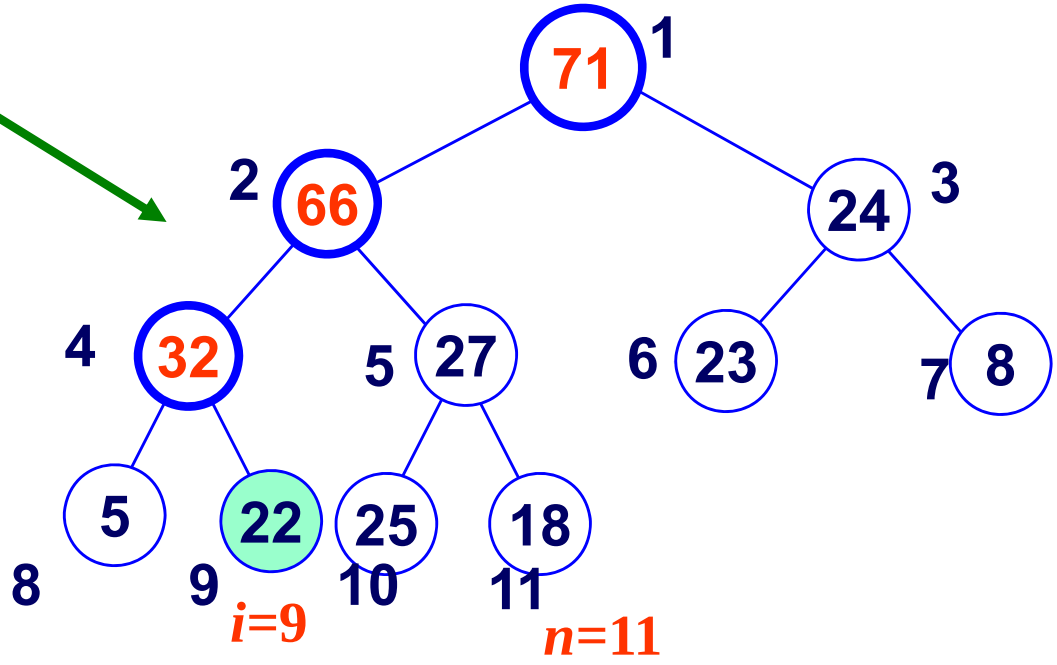


Repeatedly Swapping (Siftdown)



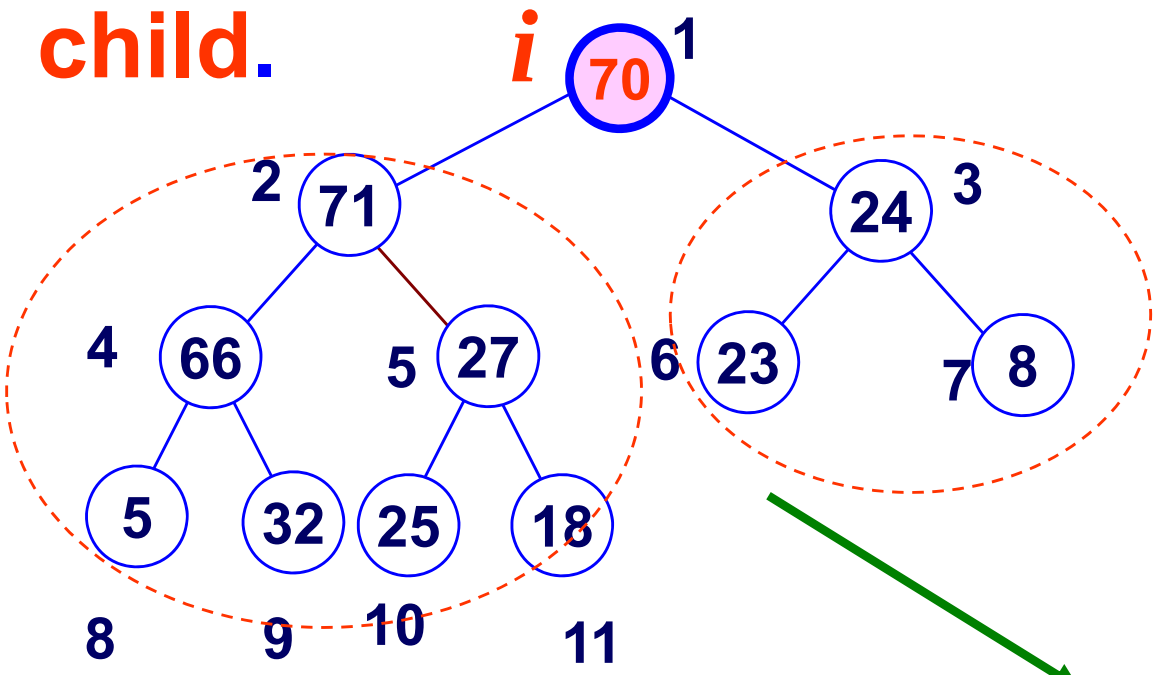
When swapping repeat ends?

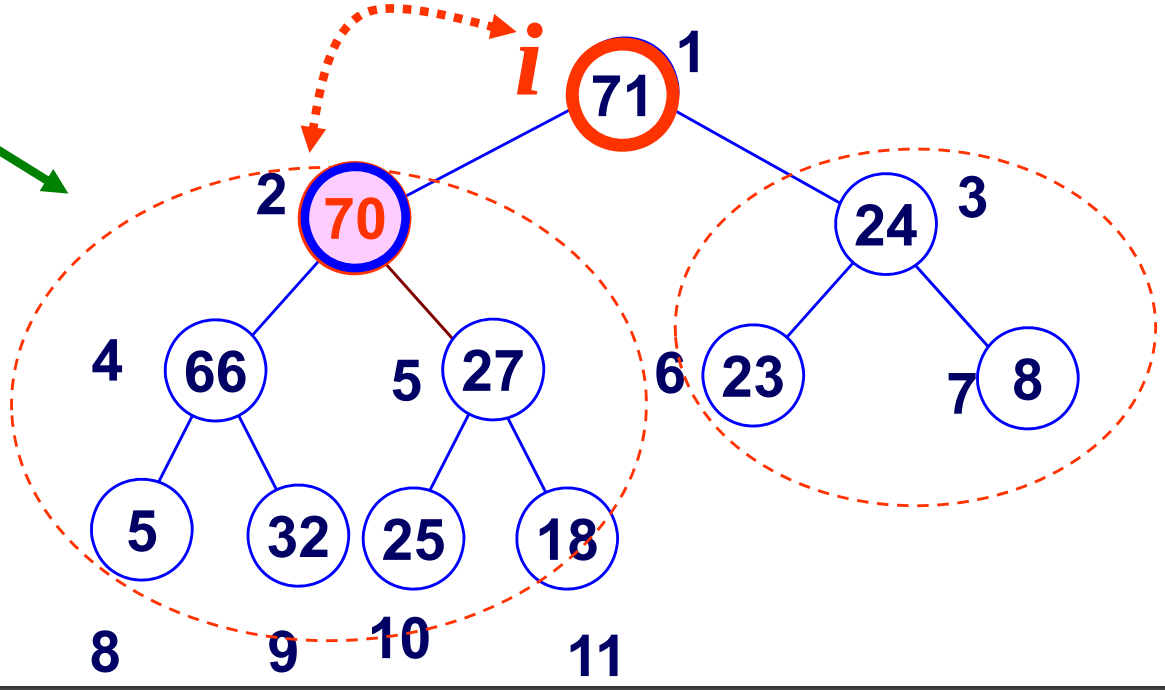
1) Node has no child (2**i*>*n*)
2) Node is already larger than children, exit the while loop



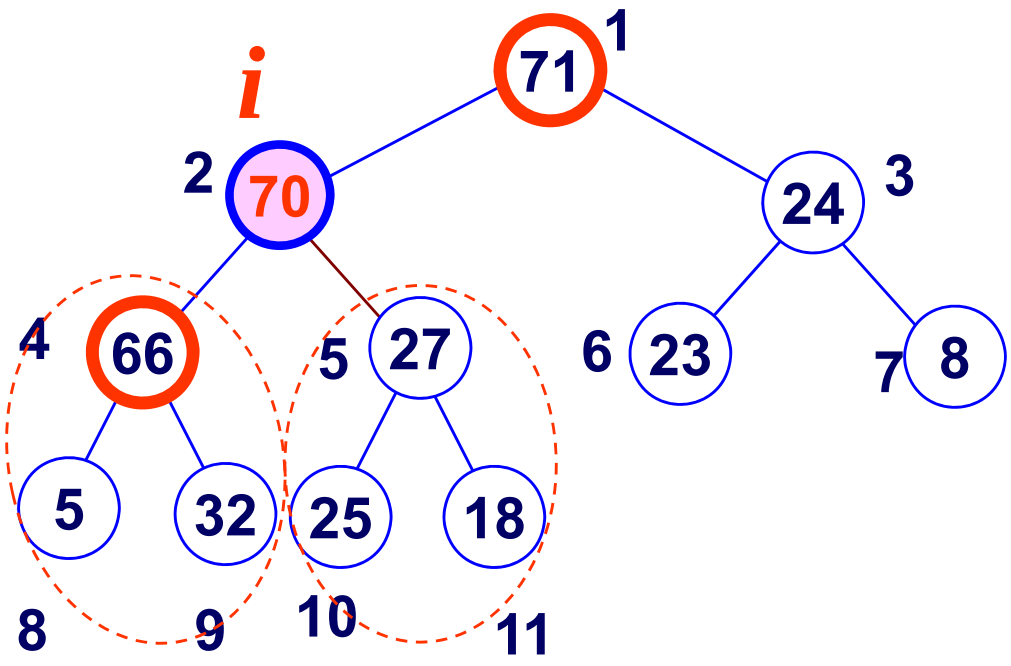
Algorithm siftdown: Example

- Idea: repeatedly swap a value with its larger child.





Algorithm siftdown



When the larger child is NOT greater than the parent, stop

Put all together: Algorithm Siftdown

```
siftdown(v,i,n) {
    // 2 * i ≤ n tests for a left child
    while (2 * i ≤ n) {
        child = 2 * i

        // if there is a right child and it is
        // bigger than the left child, update child
        if (child < n && v[child + 1] > v[child])
            child = child + 1

        // need to swap?
        if (v[child] > v[i])
            swap (v[i], v[child]);
        else  // done, exit
            break // if node is already larger, exit while loop
        i= child
    }
}
```

✓ As long as a node has the child, repeat …

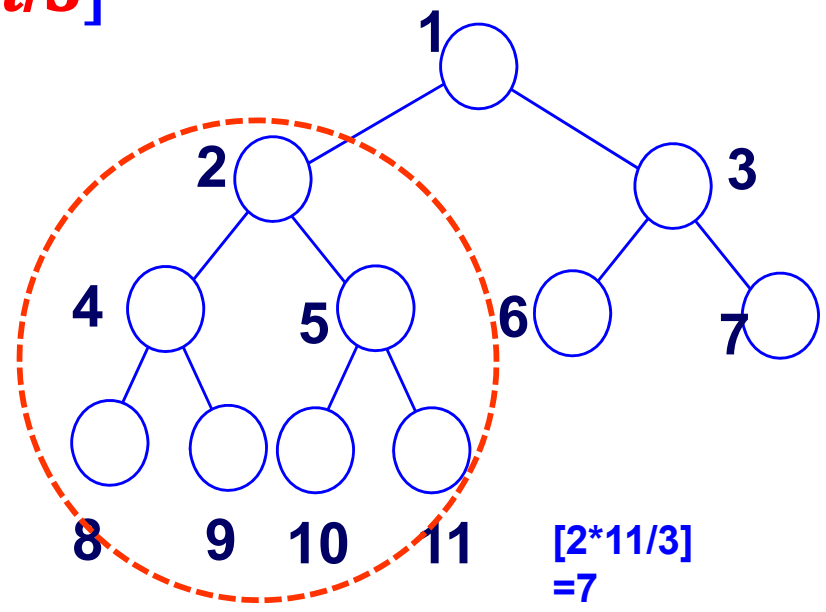
✓ *child* always refers to the larger child

✓ Swapping

✓ Consider next

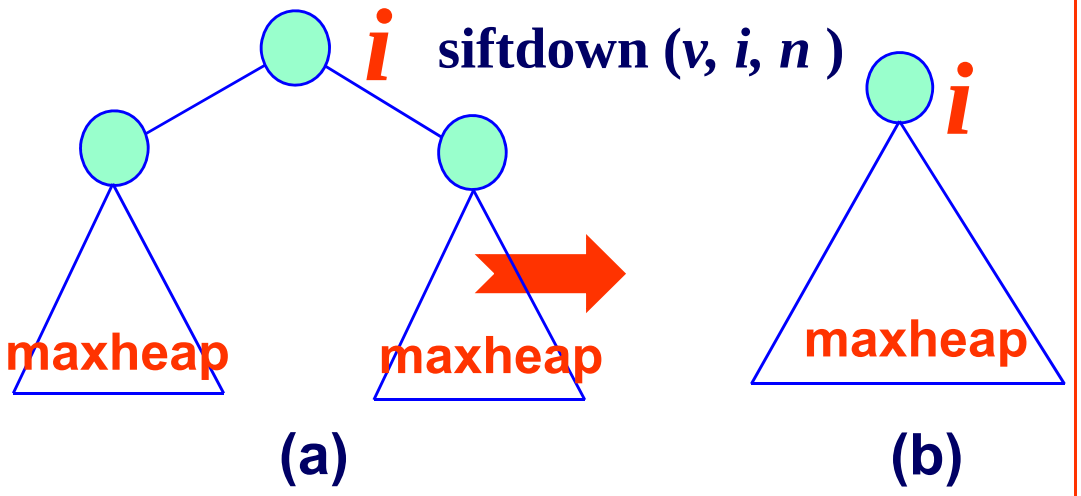
Summary: Algorithm Siftdown

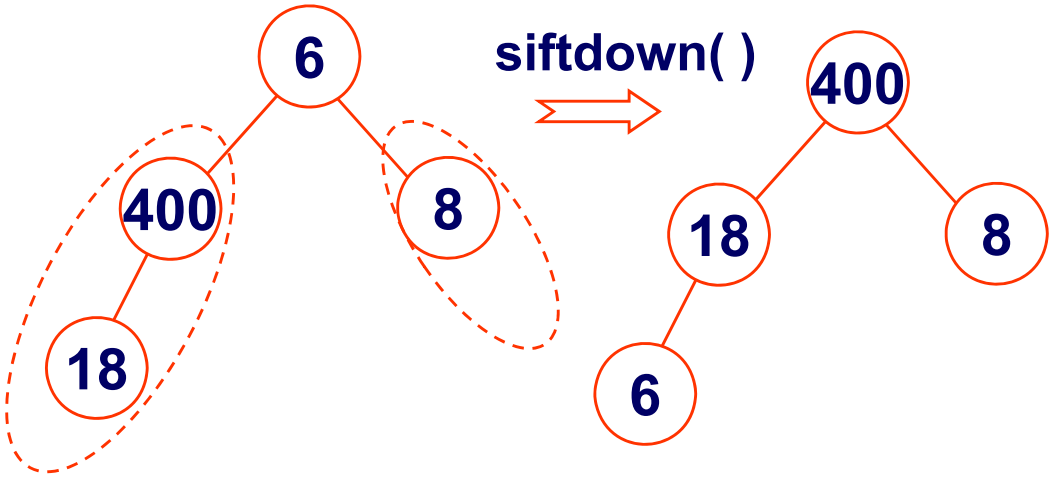
- Suppose the subtree at node *i* has *n* nodes.
- Takes O(1) time to determine which of *i*, left(*i*) and right(*i*) is largest, and perform swapping.
- Call siftdown on one of the children's subtree.
- Worst case: bottom level of tree is exactly half full, i.e., the two children's subtrees differ most in number of nodes.
- Size of child subtree in worst case: $\lfloor 2n/3 \rfloor$
- $T(n) \leq T(2n/3) + O(1)$
- Exercise: show that $T(n) = O(\log n)$



Summary: Algorithm Siftdown

- The array *v* represents a heap structure indexed from *1* to *n*. The left and right subtrees of node *i* are maxheaps. After *siftdown(v, i, n)* is called, the tree rooted at *i* is a maxheap.

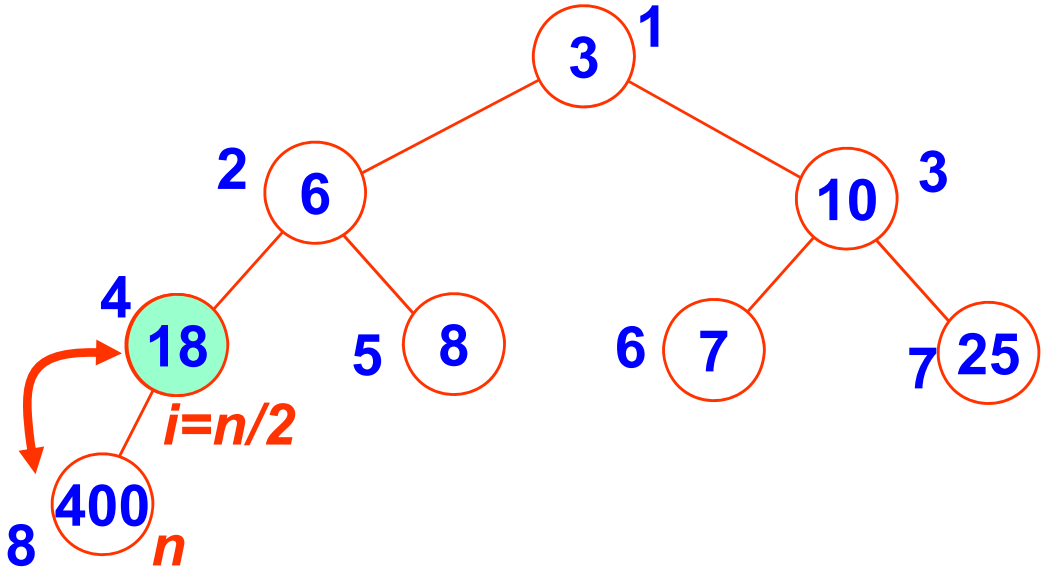


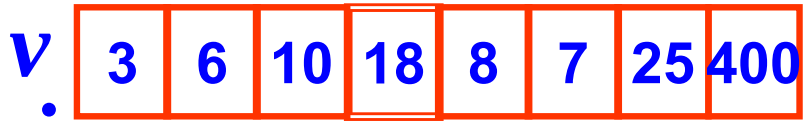


- ✓ Initially, the left and right subtrees of node *i* are maxheaps (a).
- ✓ After siftdown is called, the tree rooted at *i* is a heap (b).

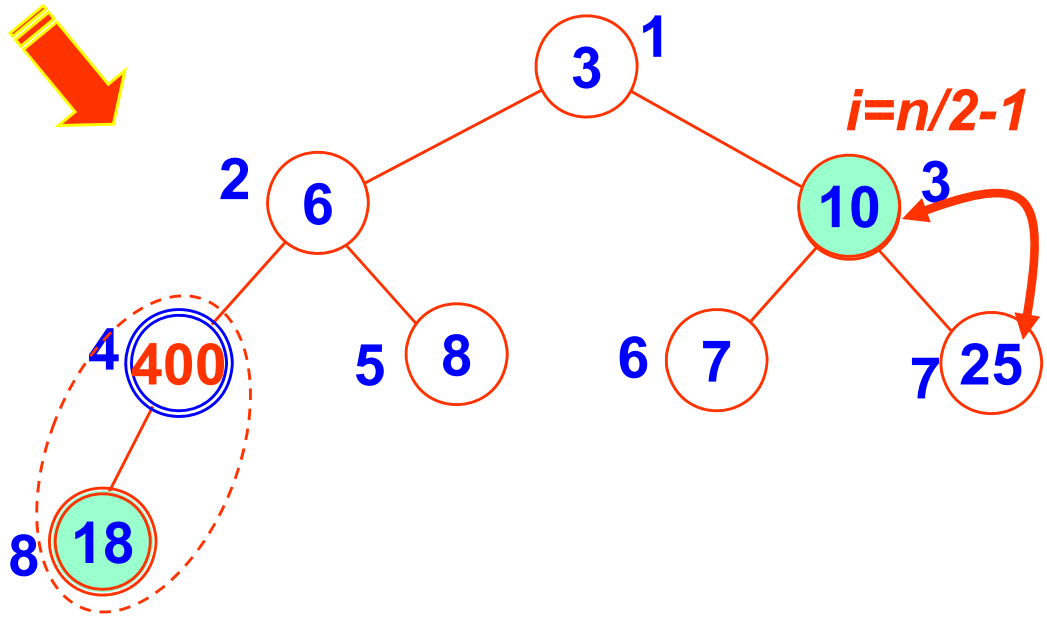
Making a maxheap: Heapify

- The problem of organizing the data into a maxheap or minheap is called *heapify*

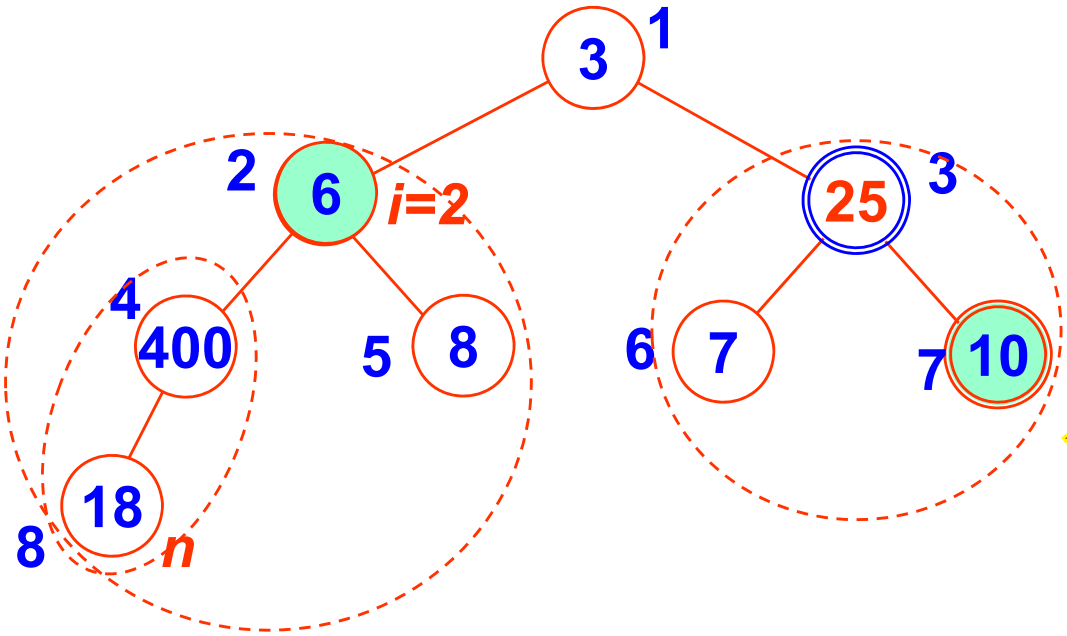




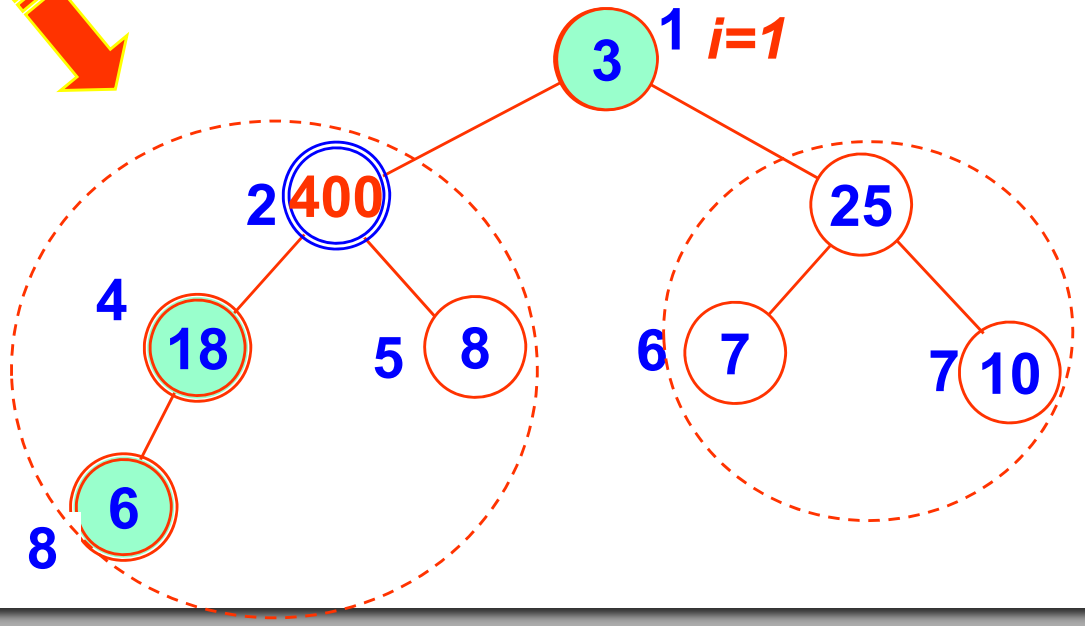

```
for i = n/2 downto 1
      siftdown()
```



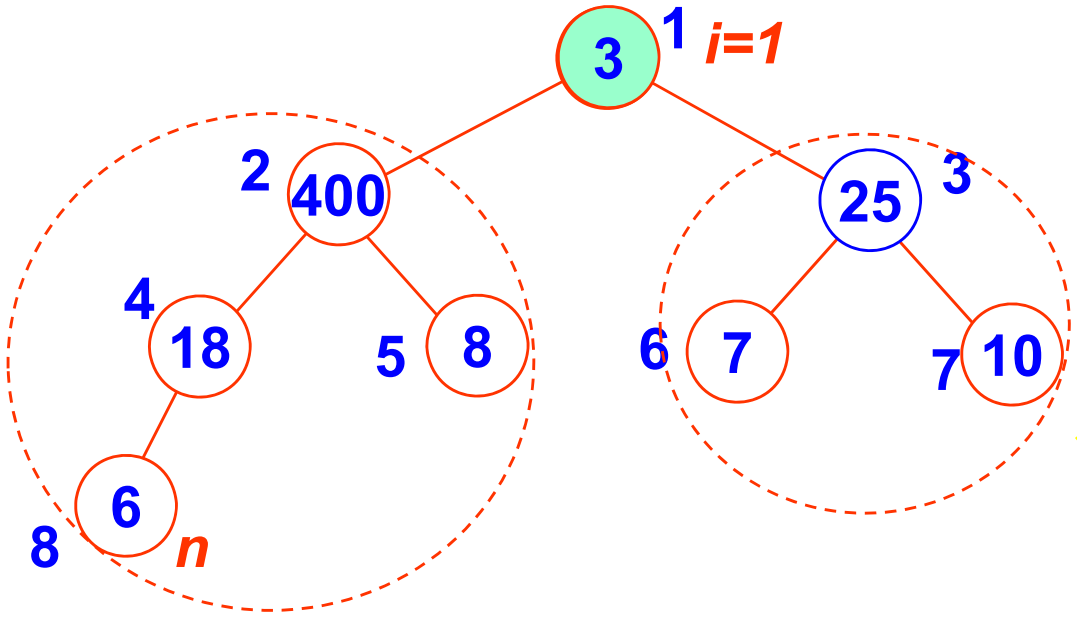
Making a maxheap: Heapify




```
for i = n/2 downto ... .
    siftdown()
```

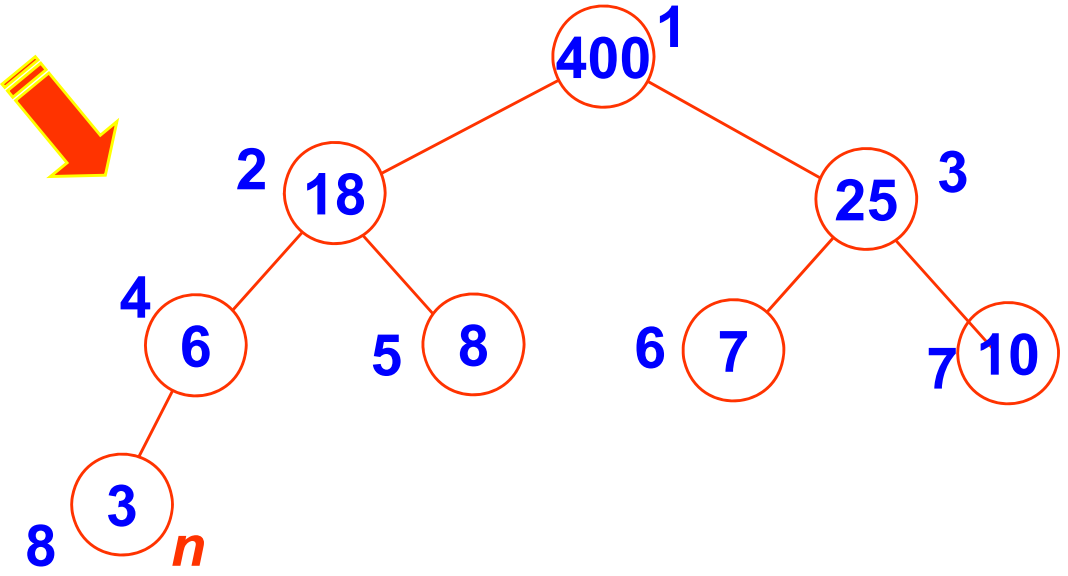


Making a maxheap: Heapify




```
for i = n/2 downto ... .
    siftdown()
```

When index 1, it terminates

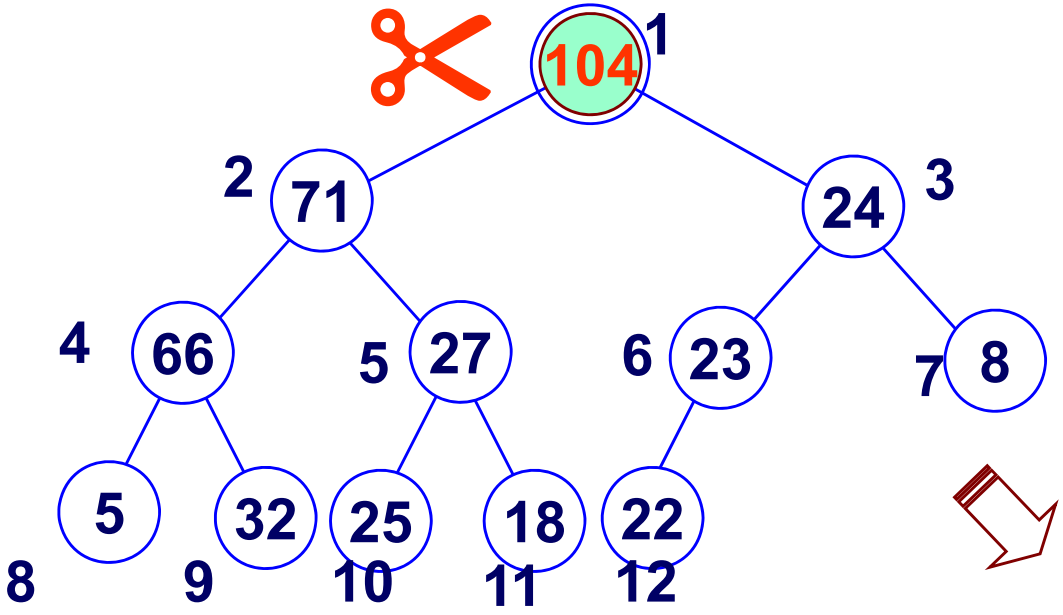


Summary Algorithm Heapify

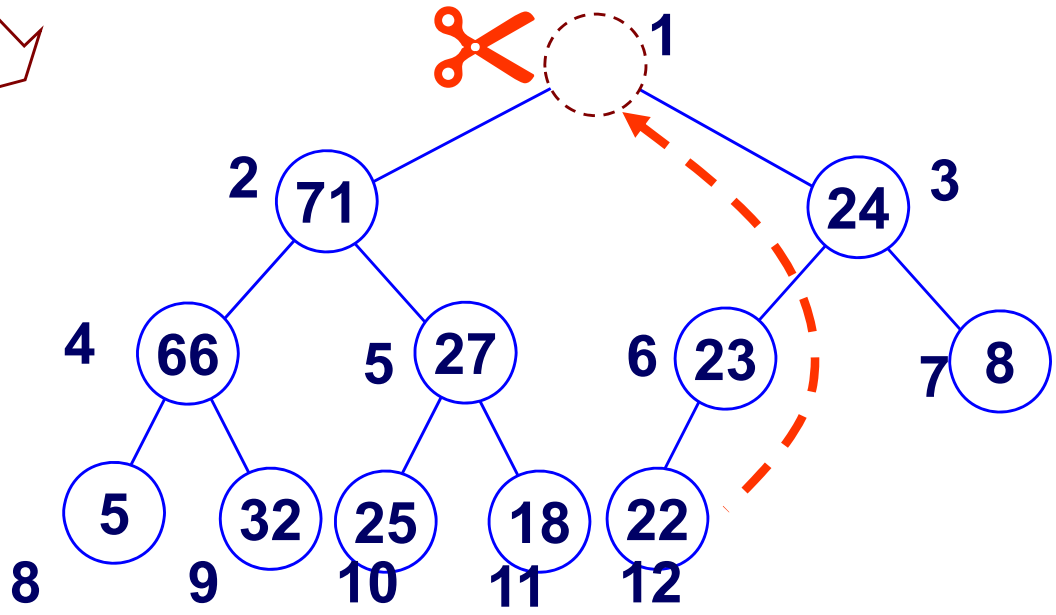
- We state the algorithm to make a maxheap/minheap as heapify( ).
  - ☞ This algorithm rearranges the data in the array *v*, indexed from *1* to *n*, so that it represents a heap.

```
heapify(v,n) {
    // n/2 is the index of the parent of the last node
    for i = n/2 downto 1
        siftdown(v,i,n)
}
```

Deleting root from a maxheap

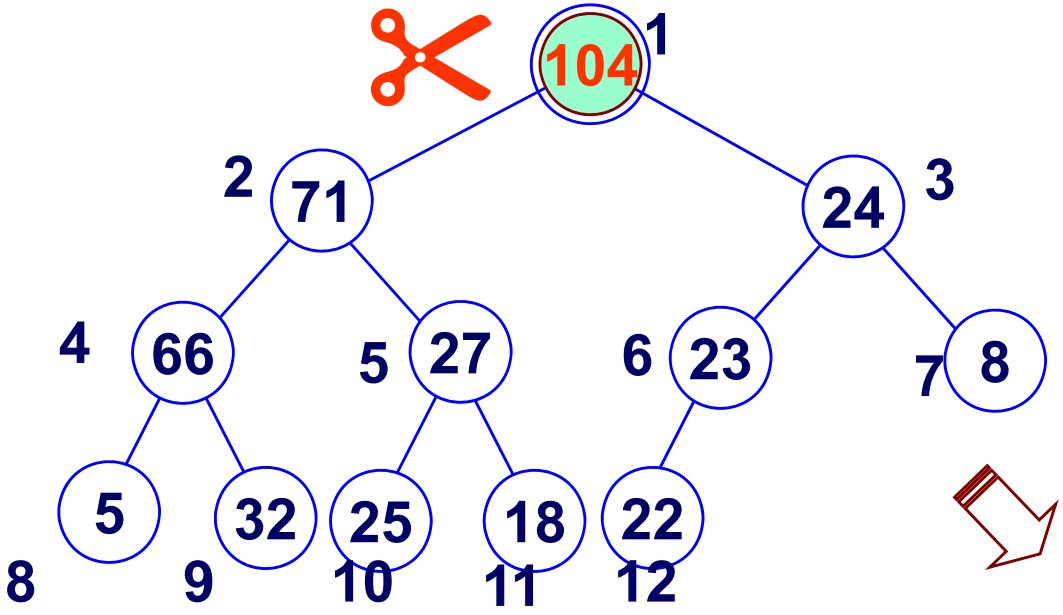


- ✓ The root that contains the largest value, 104, is to be deleted
- ✓ Idea, we move the value at the bottom level, farthest right, 22, to the root

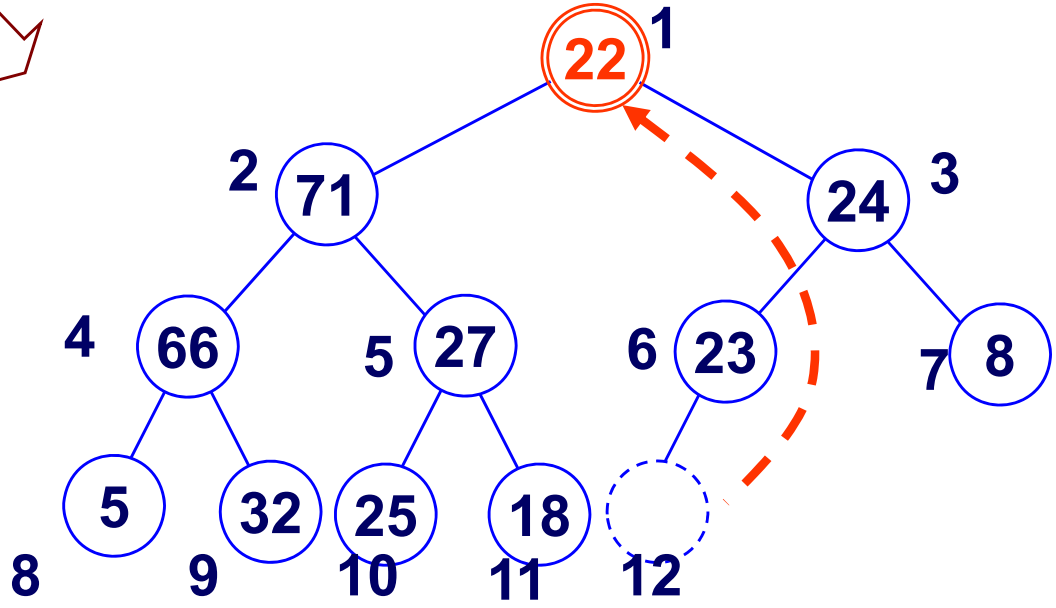


- ✓ After deleting the root, we have to recover the structure

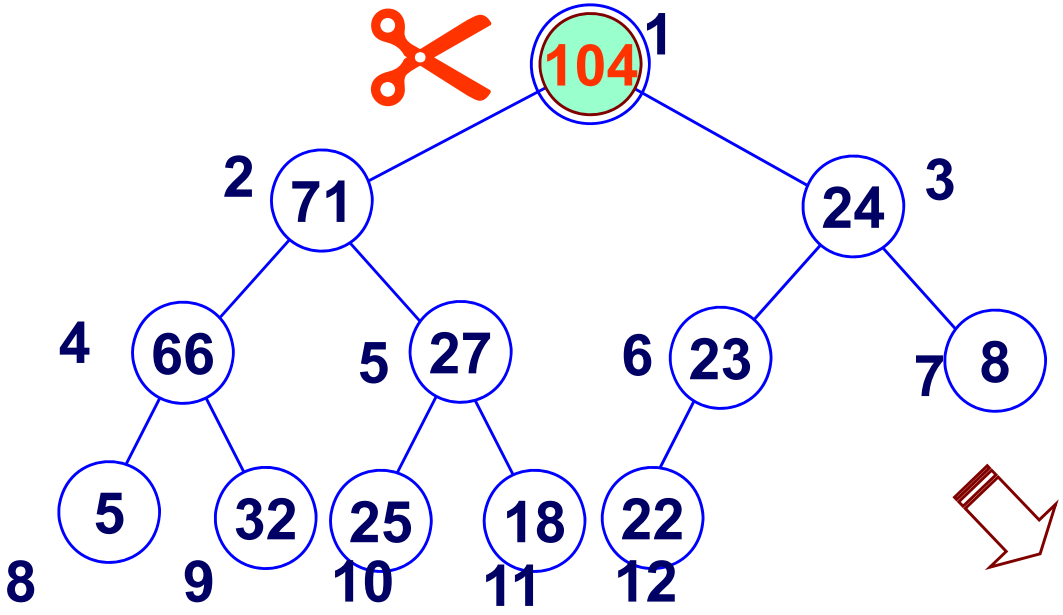
Deleting root from a maxheap



- ✓ **Idea**, we move the value at the bottom level, farthest right, 22, to the root

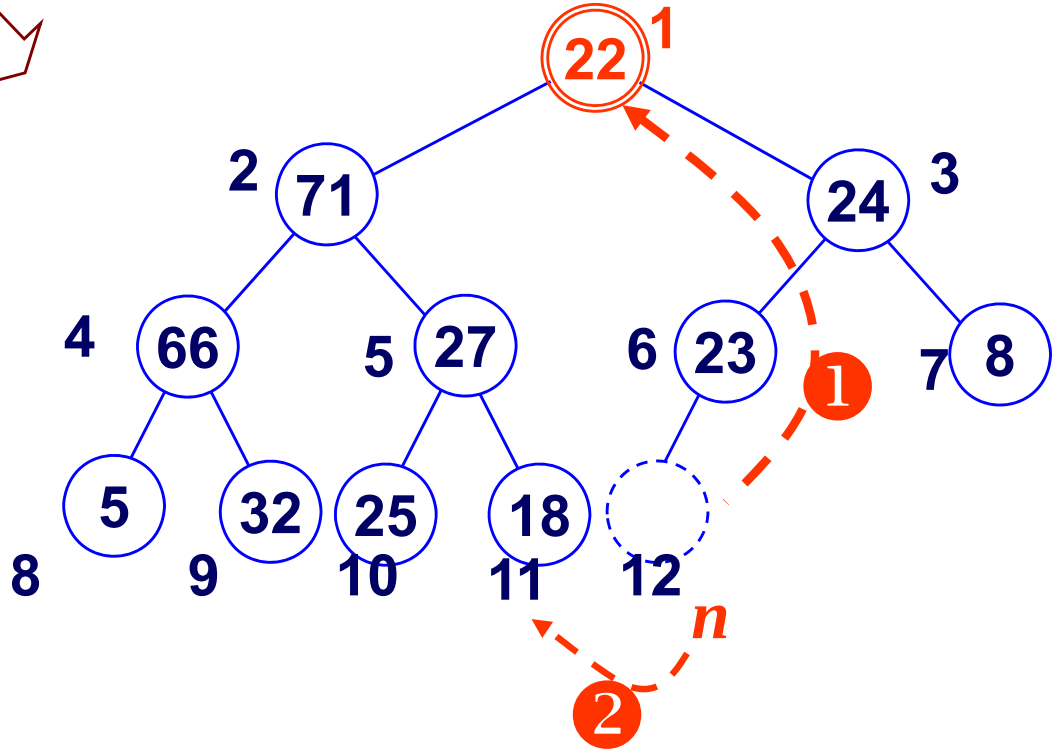


Deleting root from a maxheap



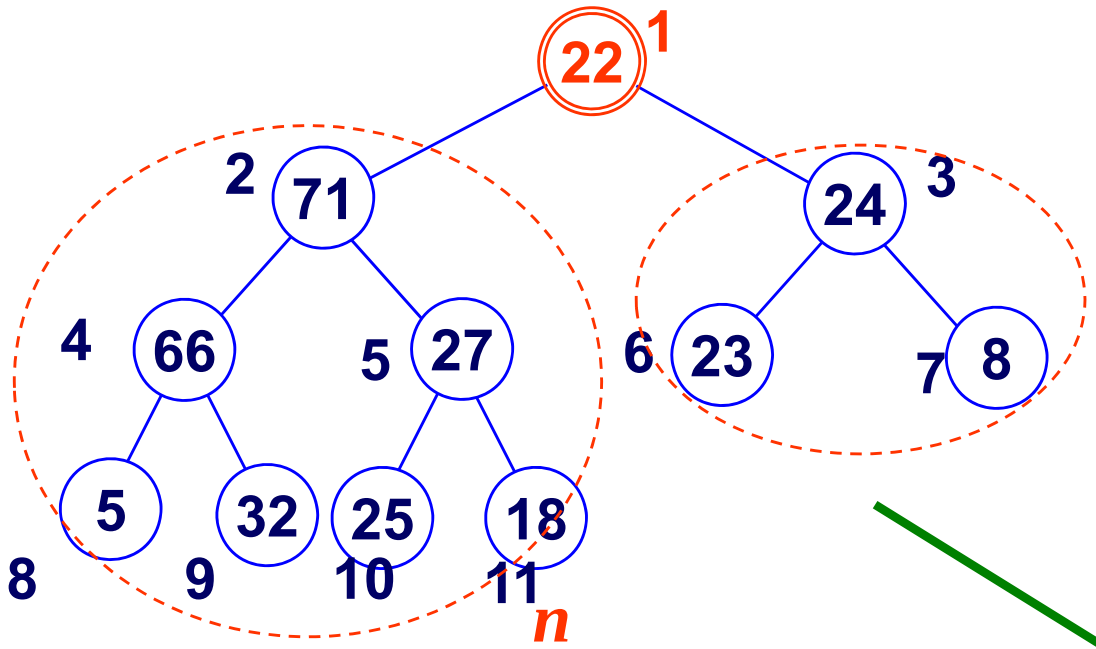
Code

```
v[1] = v[n]      1
n = n - 1        2
```

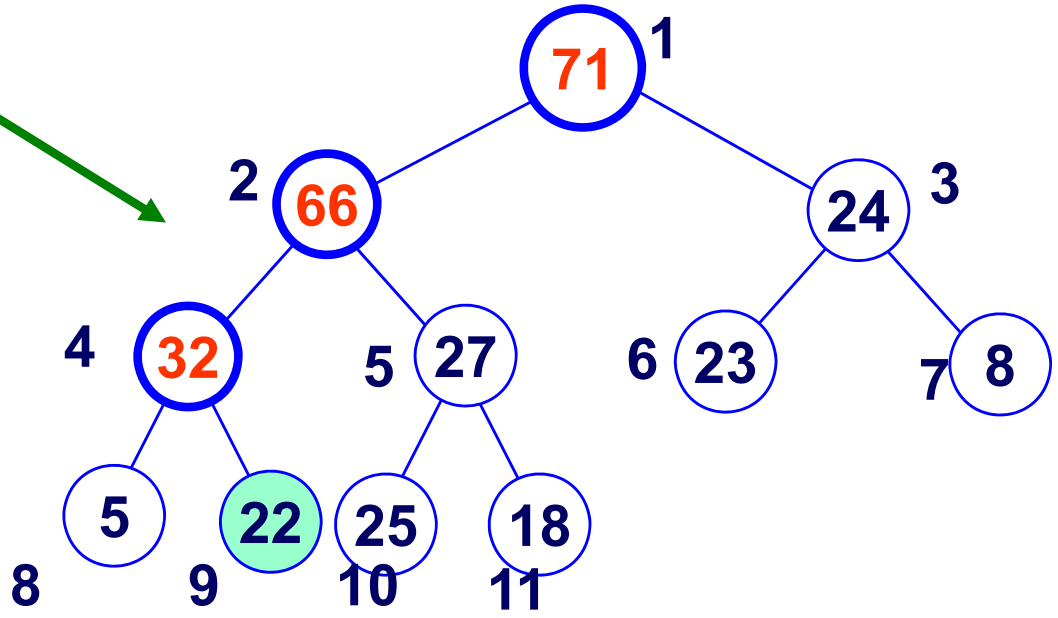


✓ **Idea**, we move the value at the bottom level, farthest right, 22, to the root

Deleting root from a maxheap



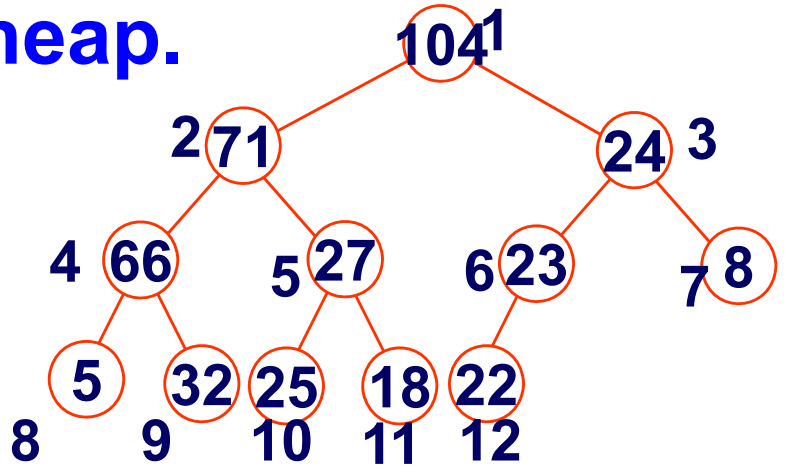
- ✓ Need to recover to a maxheap
- ✓ siftdown( ) ❸

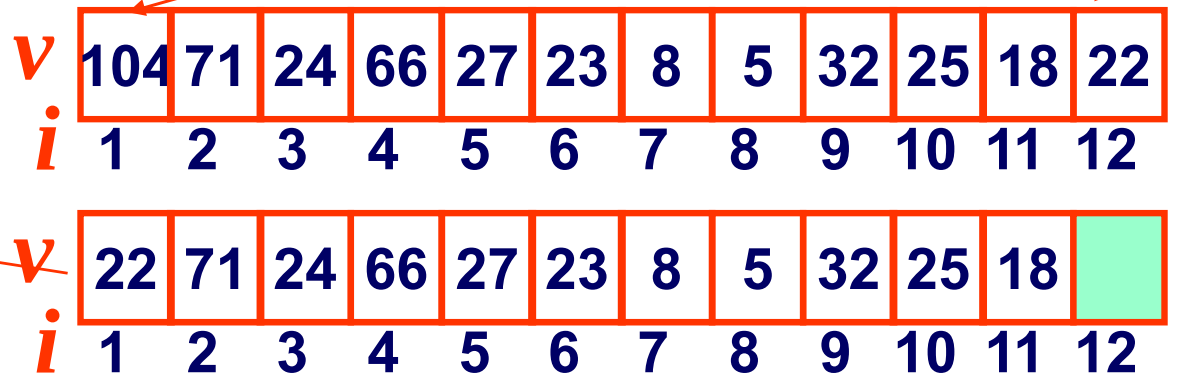


Summary: Algorithm Delete the root from a maxheap

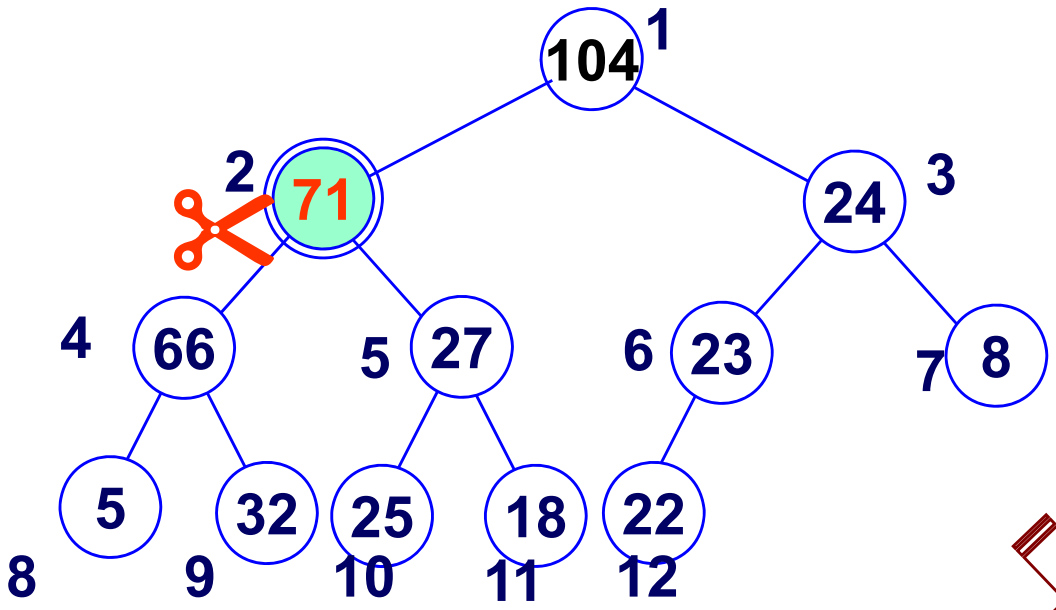
- This algorithm deletes the root (the item with largest value) from a heap containing n elements.
  - The array v represents the heap.

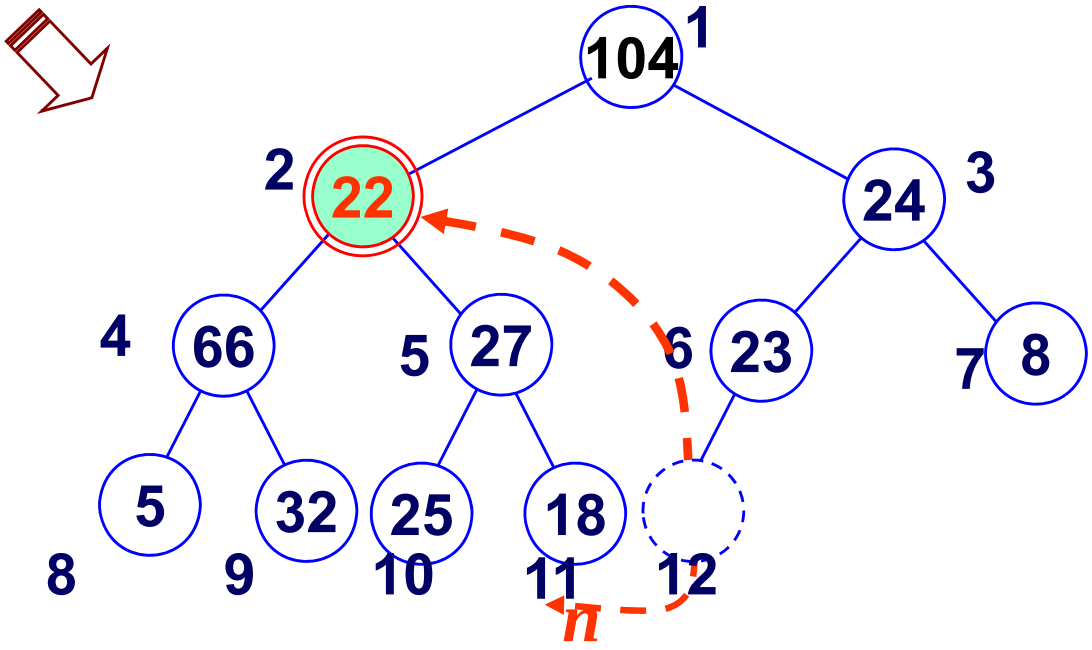
```
heap_delete(v,n) {
   // move the item with
the largest index to root
    v[1] = v[n]          1
    n = n - 1            2
 // maintain a heap
    siftdown(v,1,n)      3
}
```





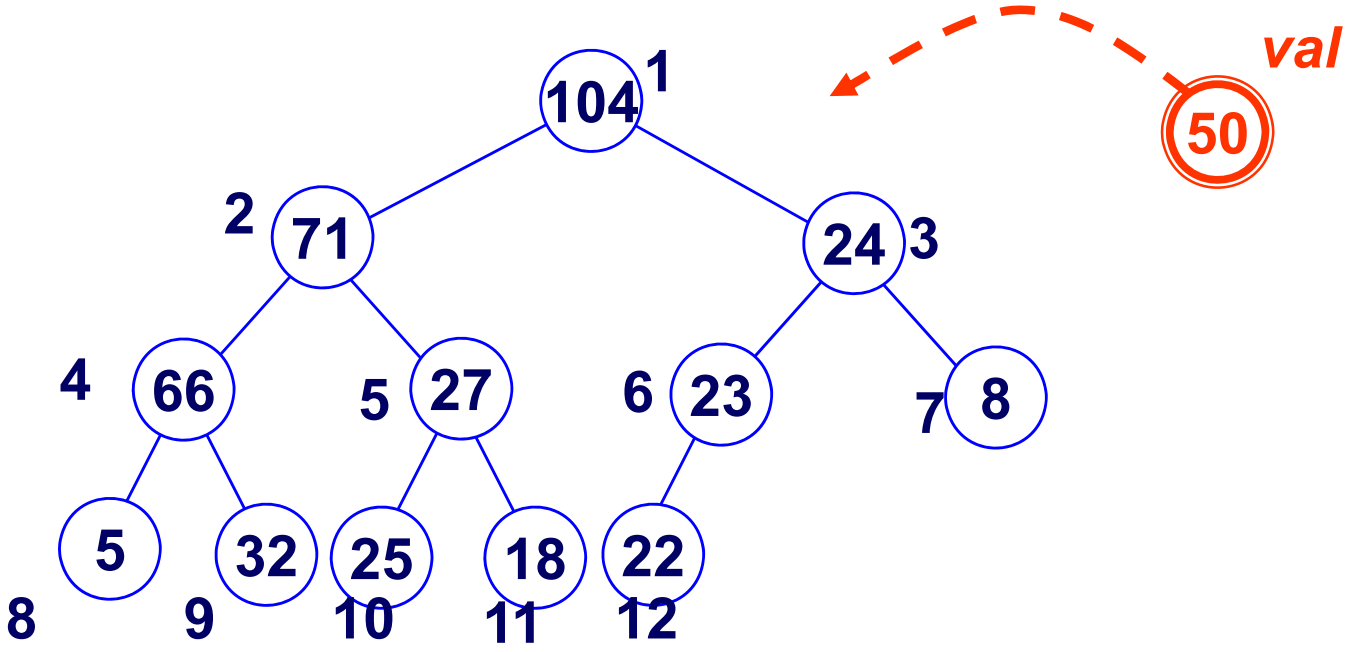
Deleting any node from a maxheap





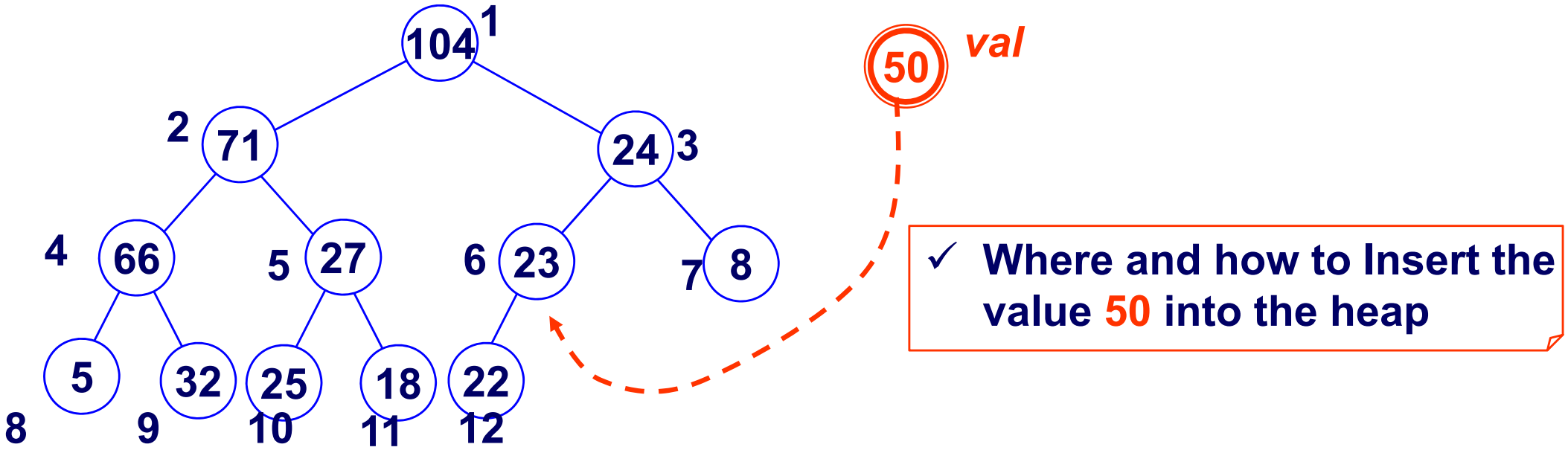
Insert a value into the maxheap

- ✓ Where and how to Insert the value 50 into the heap

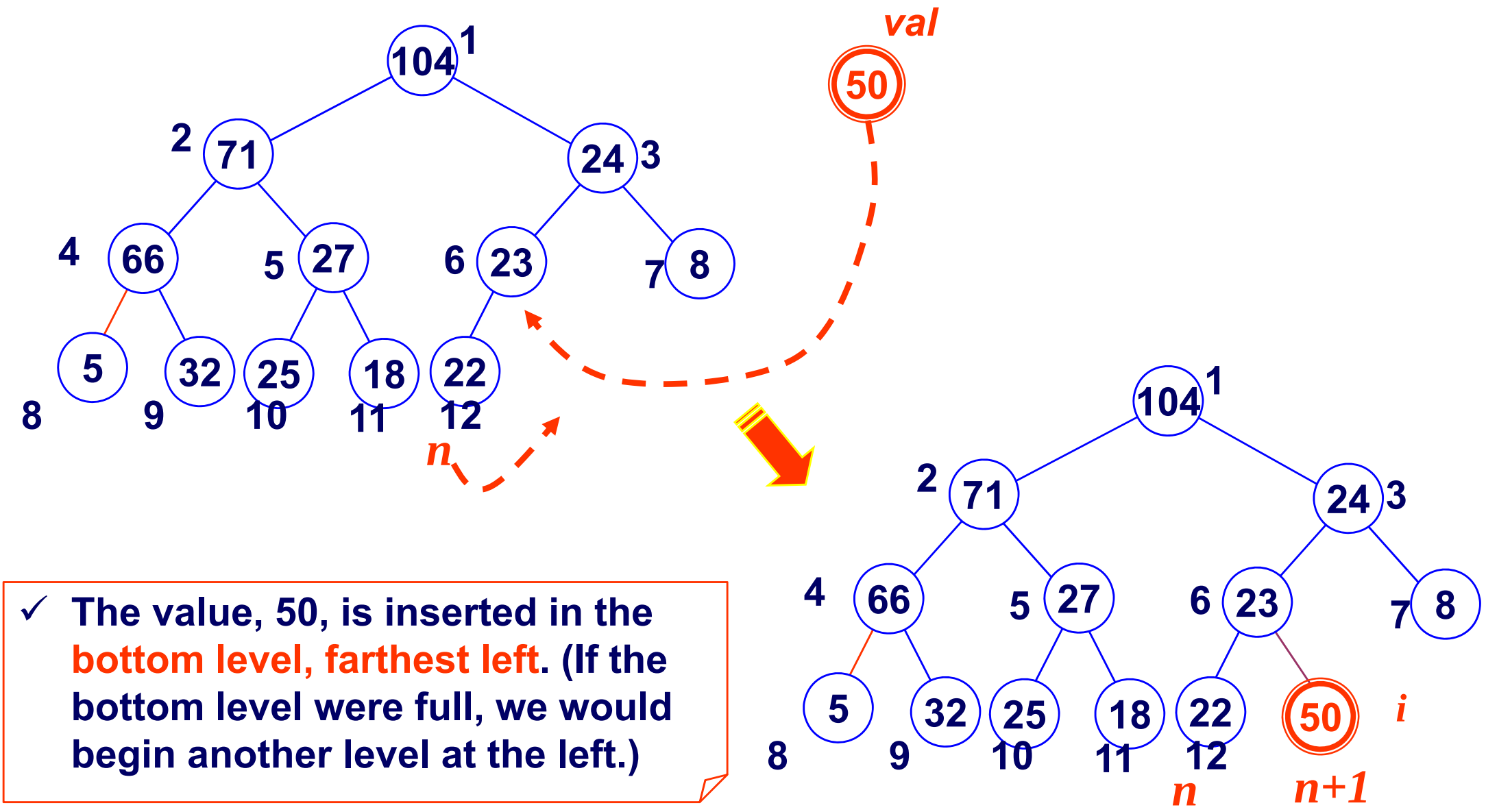


Insert a value into the maxheap

- **Idea:**
  - ☞ When inserting a value into the maxheap, we first insert it in the bottom level, farthest left, then repeatedly move the parent down, maintaining the maxheap structure and property,

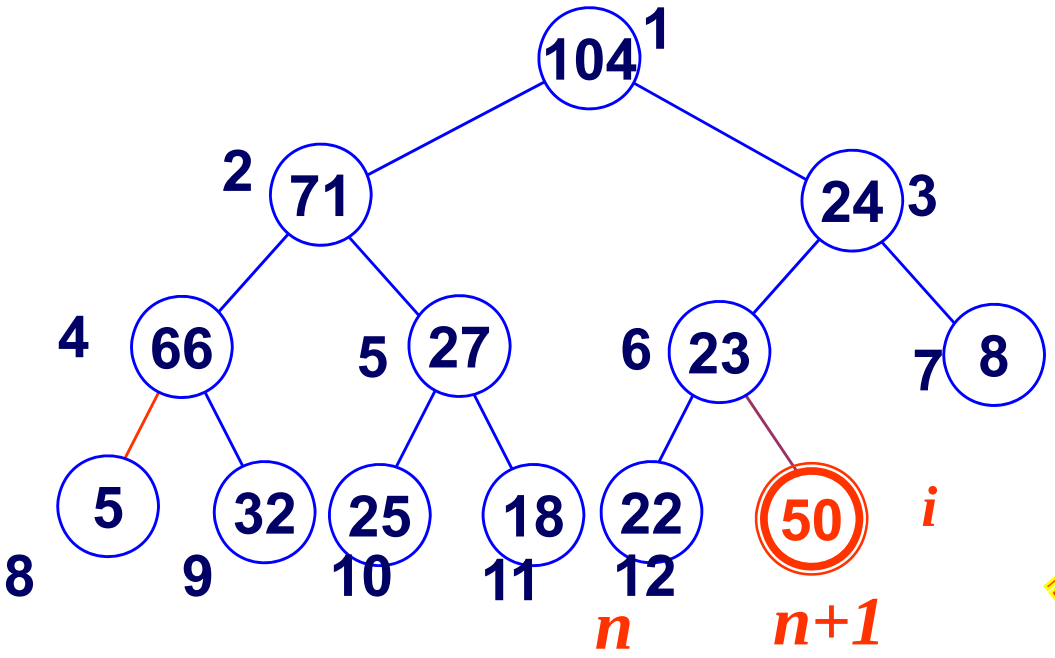


Inserts the value val into a maxheap



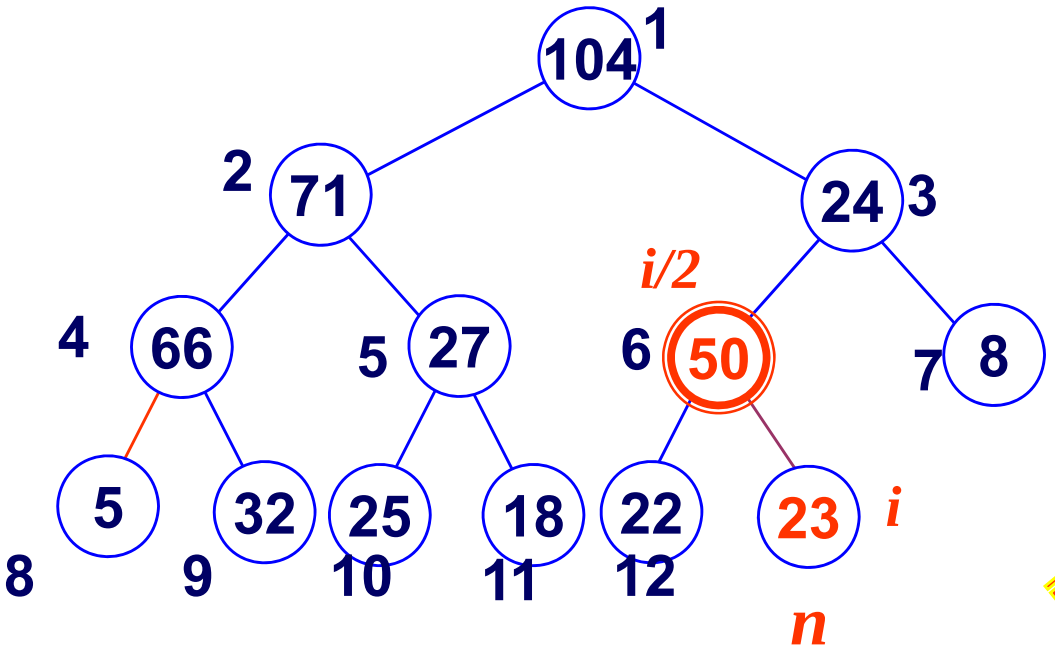
- ✓ The value, 50, is inserted in the bottom level, farthest left. (If the bottom level were full, we would begin another level at the left.)

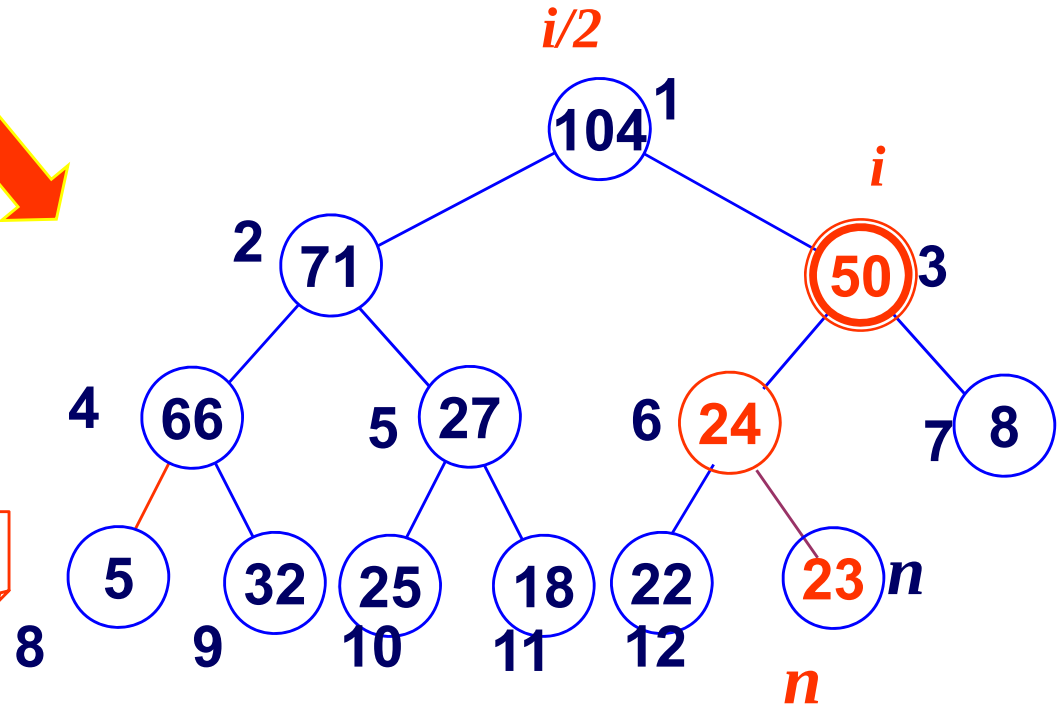
Inserts the value val into a maxheap



- ✓ The value, 50, is inserted in the bottom level, farthest left. (If the bottom level were full, we would begin another level at the left.)

Inserts the value val into a maxheap





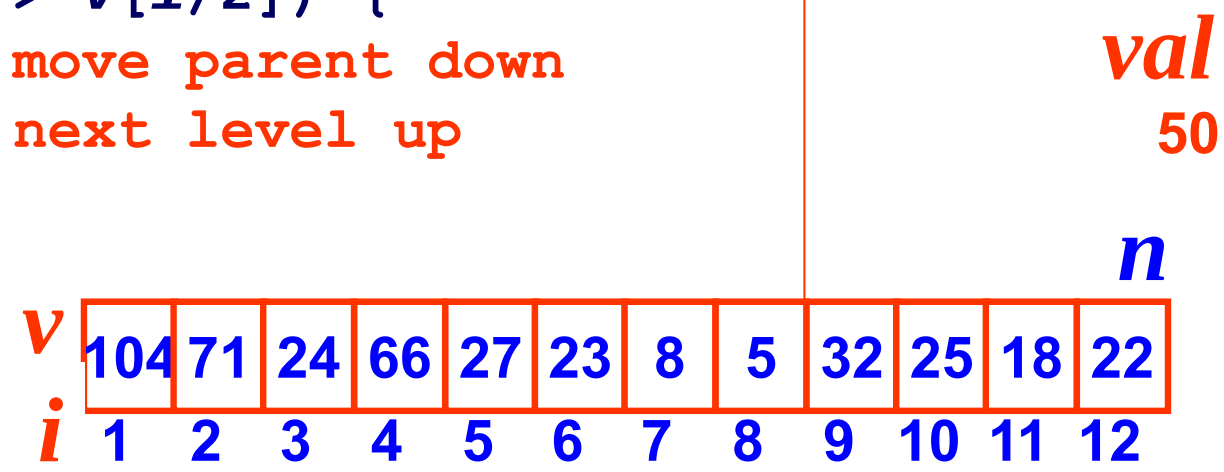
✓ algorithm terminates.

Summary: Algorithm Insert into the maxheap

- The algorithm repeatedly move the parent down, maintaining the heap structure and property

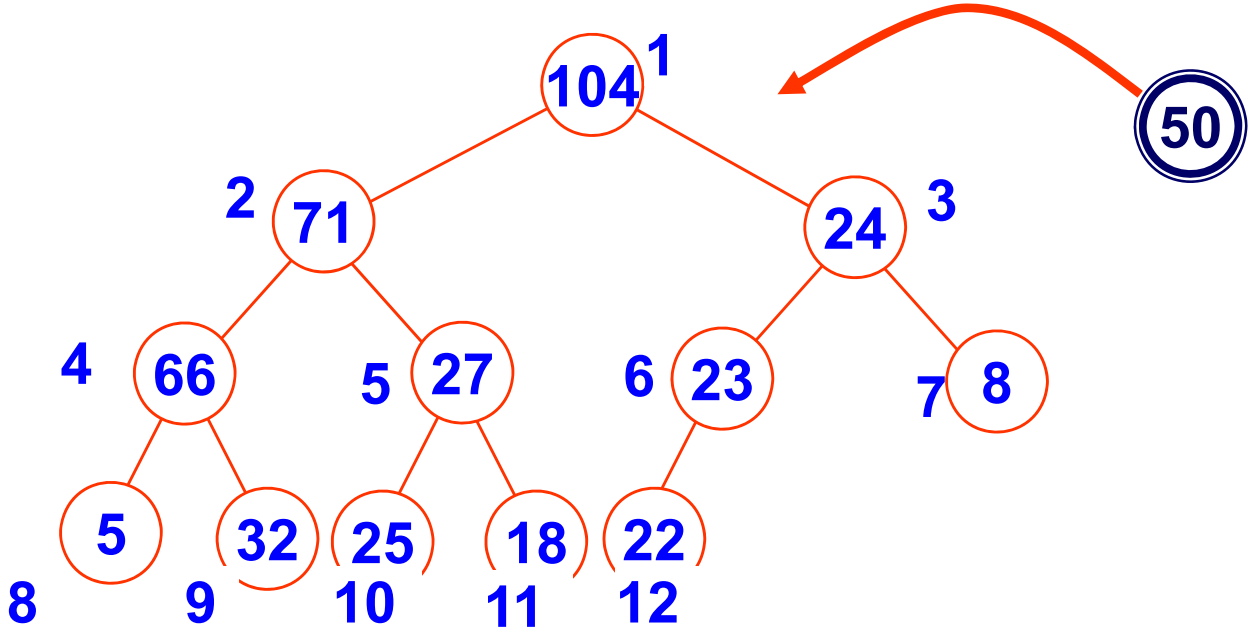
```
heap_insert(val,v,n) {
    i = n + 1
    n = n +1
    // i is the child and i/2 is the parent.
    // If i > 1, i is not the root.
    while (i > 1 and val > v[i/2]) {
        v[i] = v[i/2]   // move parent down
        i = i/2         // next level up
    }
    v[i] = val
}
```

// a new space at bottom level, farthest left



Homework reading:

Algorithm walkthrough --- *heap_insert*



Problem Solving Session for

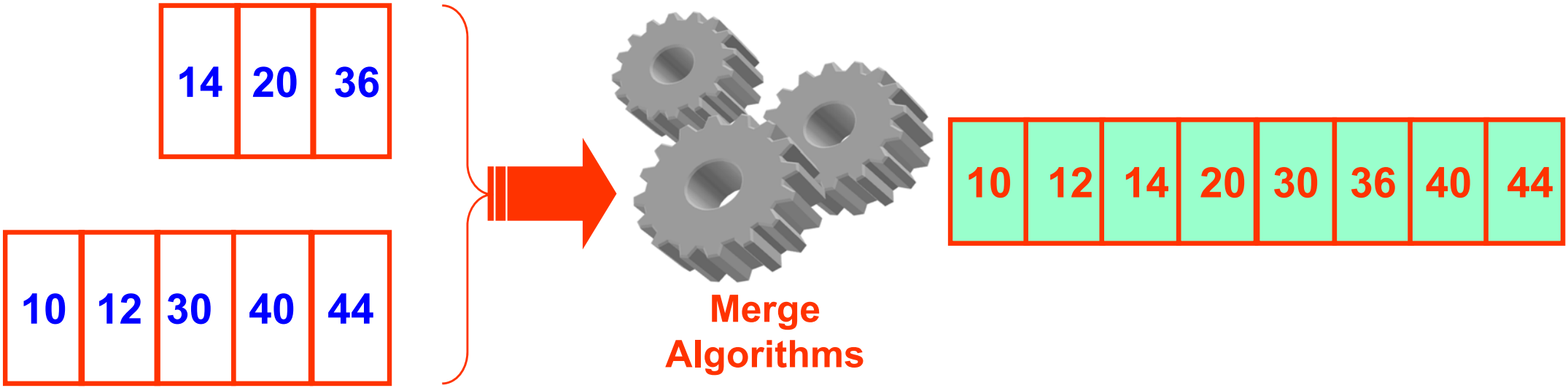
Binary Trees, Binary Search Trees, and Heaps Related Applications

Topics

- Divide and Conquer
- Mergesort
- Quicksort
- Heapsort
- Counting Sort
- Radix Sort
- Bucket Sort
- Selection problem and order statistics

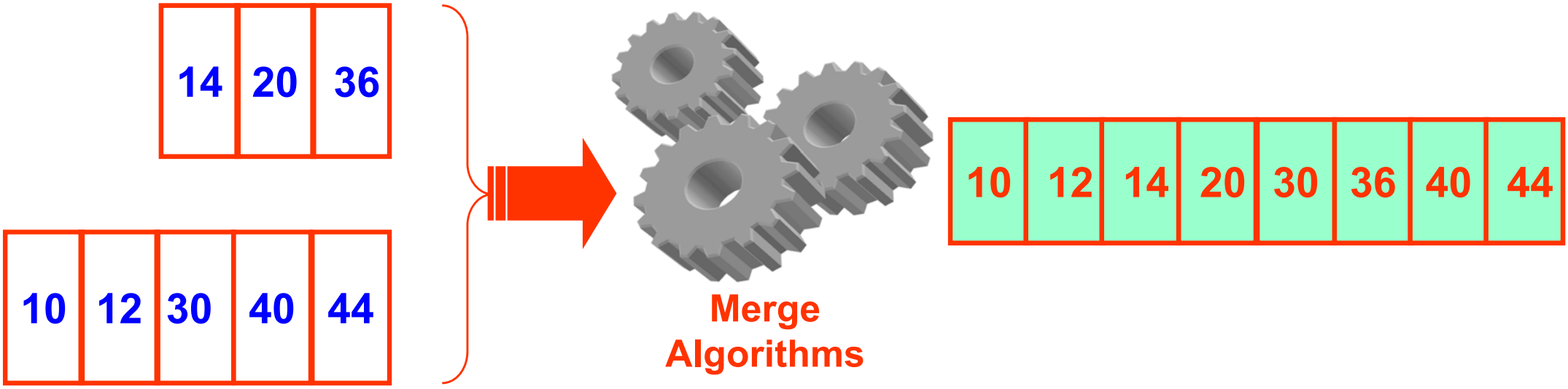
Merge Problem

- **Input:**
  - Two *nearly equal* sorted arrays
- **Output**
  - One sorted array.

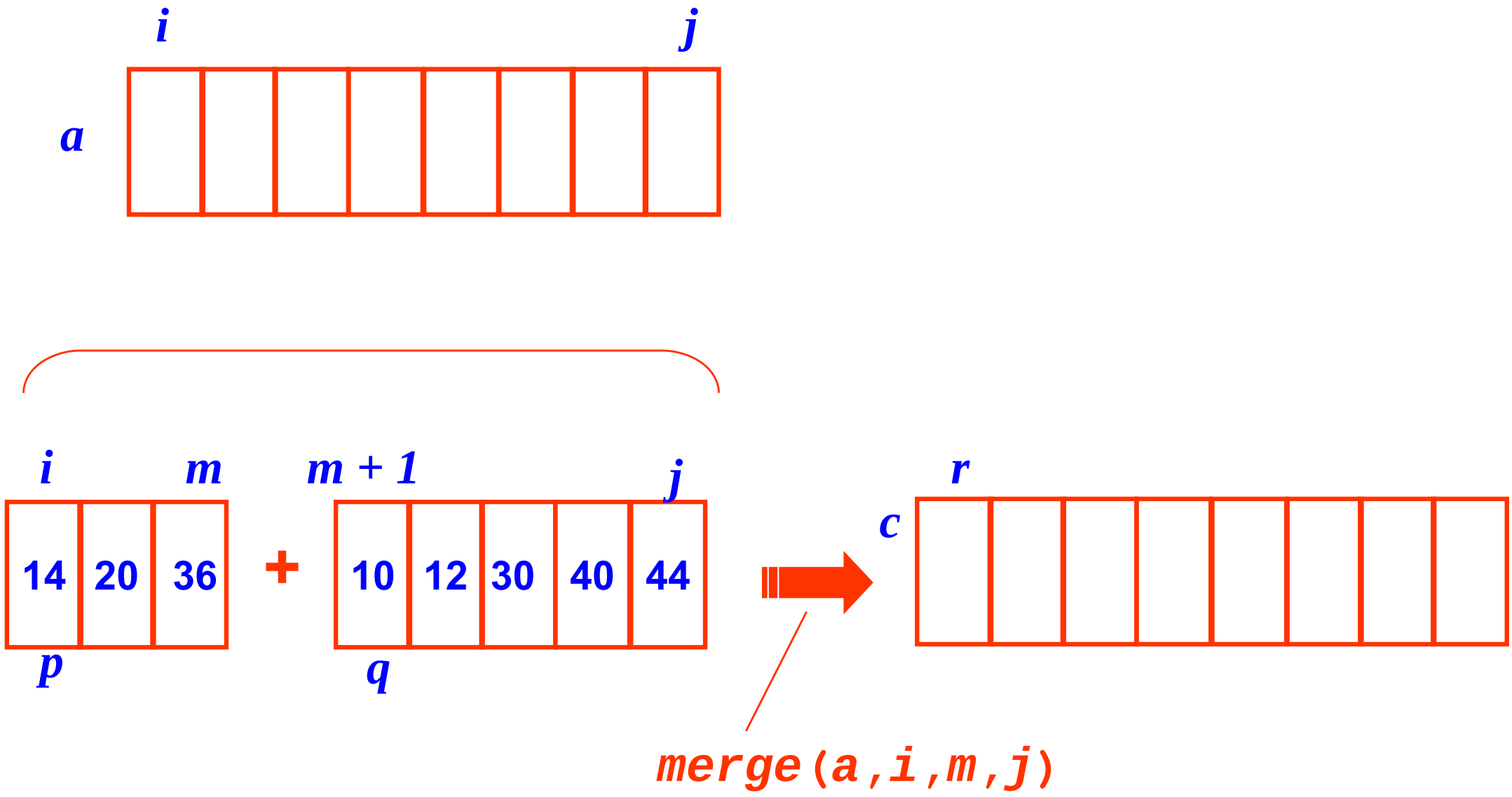


Merge Algorithm

- **Idea:**
  - ☞ Keep track of smallest element in each sorted half.
  - ☞ Insert smallest of two elements into auxiliary array.
  - ☞ Repeat until done

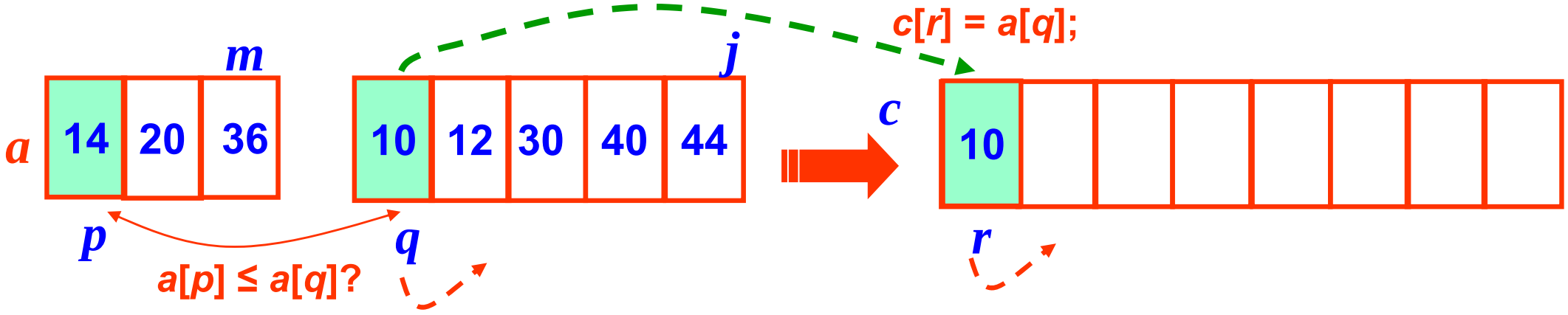


Merging Two Sorted Arrays: Code



Merging Two Sorted Arrays

We begin by examining the first element in each array



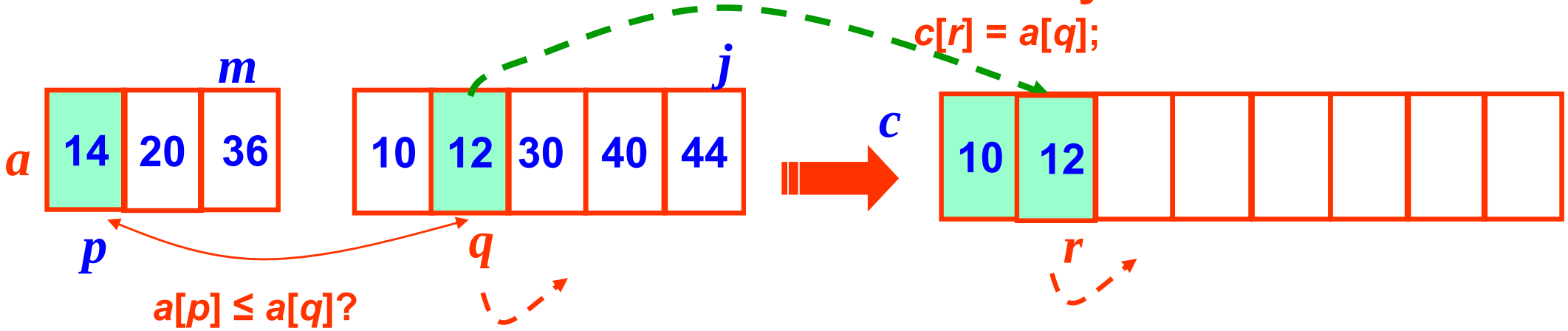
✓ Since 10 < 14 and each array is sorted, 10 is smallest, so copied to output

✓ copy smaller value to *c*

```
if (a[p] ≤ a[q]) {
    c[r] = a[p];
    p = p + 1;
}
else {
    c[r] = a[q];
    q = q + 1;
}
r = r +1;
```

Merging Two Sorted Arrays

We move to the next item in the second array

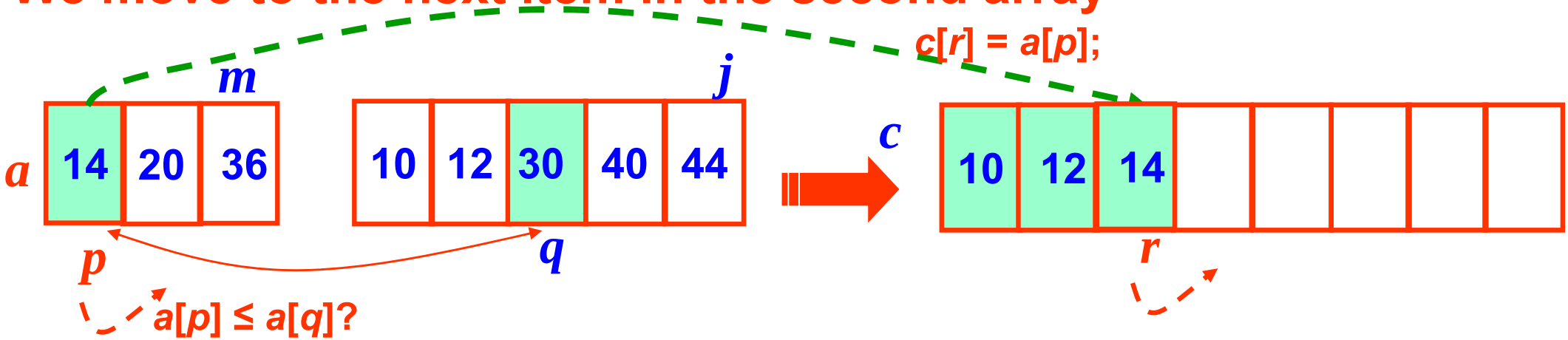


✓ copy smaller value to *c*

```
if (a[p] ≤ a[q]) {
    c[r] = a[p];
    p = p + 1;
}
else {
    c[r] = a[q];
    q = q + 1;
}
r = r +1;
```

Merging Two Sorted Arrays

We move to the next item in the second array

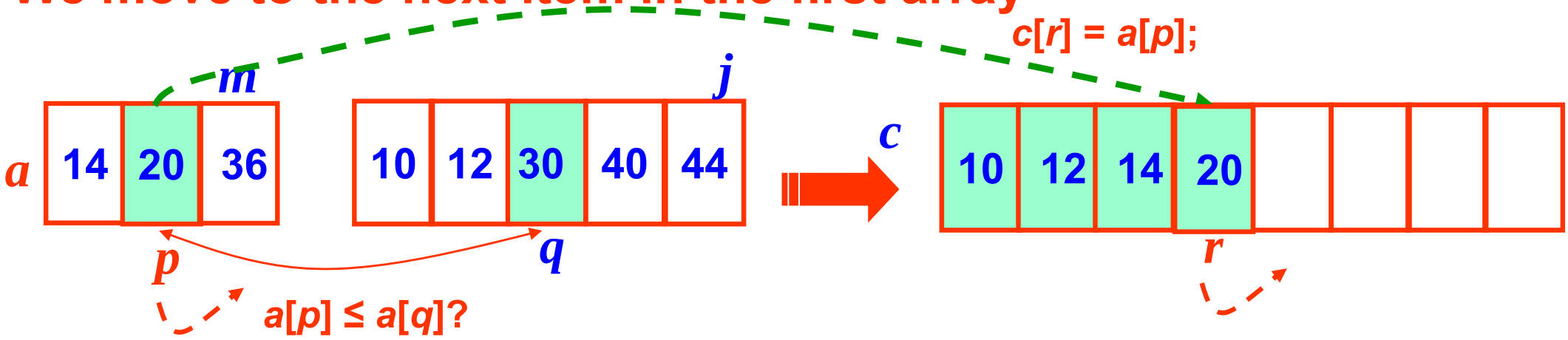


- ✓ copy smaller value to *c*

```
if (a[p] ≤ a[q]) {
    c[r] = a[p];
    p = p + 1;
}
else {
    c[r] = a[q];
    q = q + 1;
}
r = r +1;
```

Merging Two Sorted Arrays

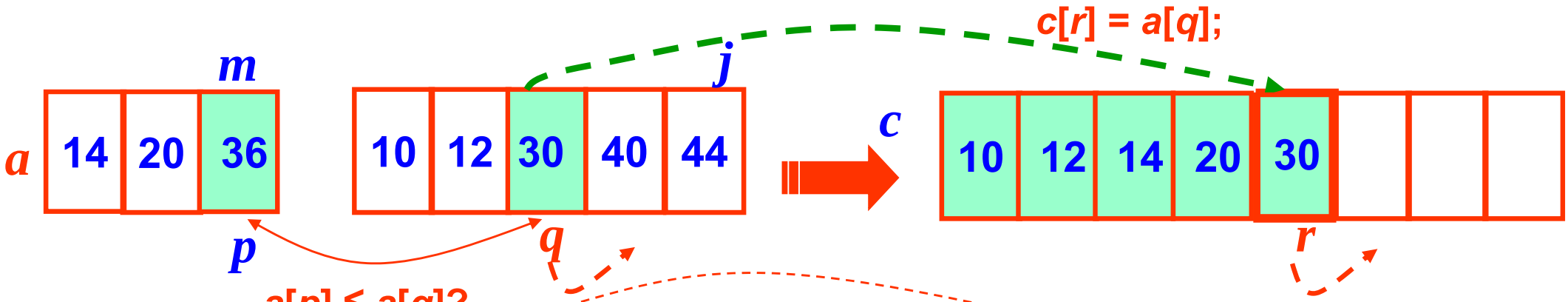
We move to the next item in the first array



- ✓ copy smaller value to *c*

```
if (a[p] ≤ a[q]) {
    c[r] = a[p];
    p = p + 1;
}
else {
    c[r] = a[q];
    q = q + 1;
}
r = r +1;
```

Merging Two Sorted Arrays



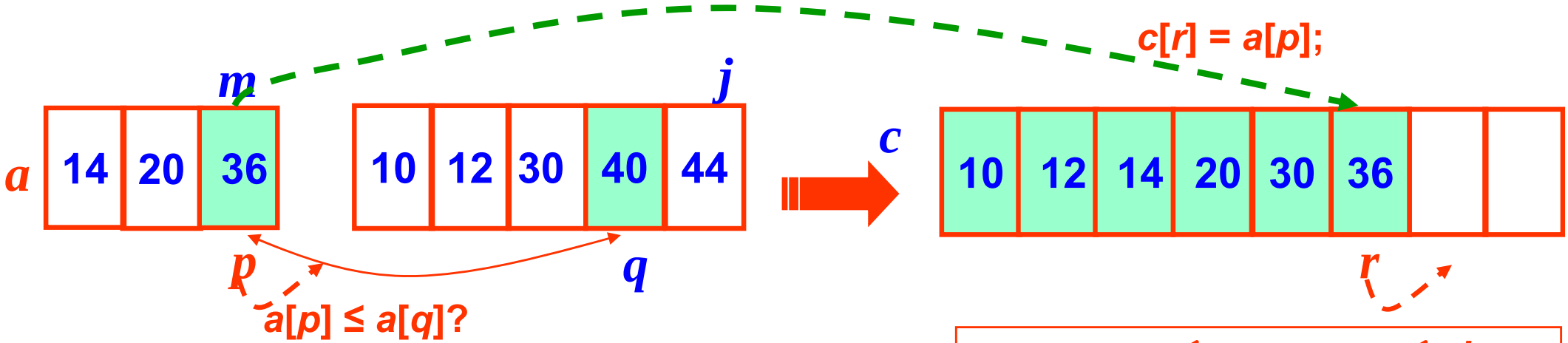

```
while (p ≤ m and q ≤ j) {


}
```

✓ We continue as long as $p \le m$ and $q \le j$

```
if (a[p] ≤ a[q]) {
    c[r] = a[p];
    p = p + 1;
}
else {
    c[r] = a[q];
    q = q + 1;
}
r = r +1;
```

✓ copy smaller value to c

Merging Two Sorted Arrays




```
while (p ≤ m and q ≤ j) {


}
```

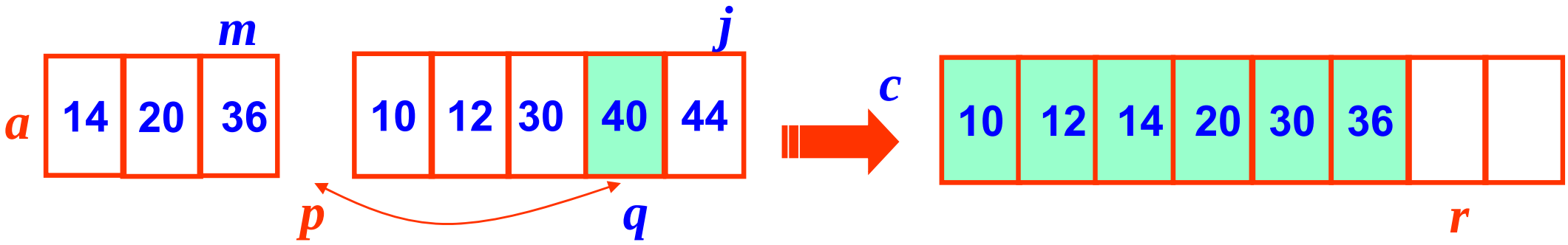
✓ We continue as long as $p \le m$ and $q \le j$

```
if (a[p] ≤ a[q]) {
    c[r] = a[p];
    p = p + 1;
}
else {
    c[r] = a[q];
    q = q + 1;
}
r = r +1;
```

✓ copy smaller value to c

Merging Two Sorted Arrays

All of the data from the first array have been copied to the output array



✓ We continue as long as $p \le m$ and $q \le j$

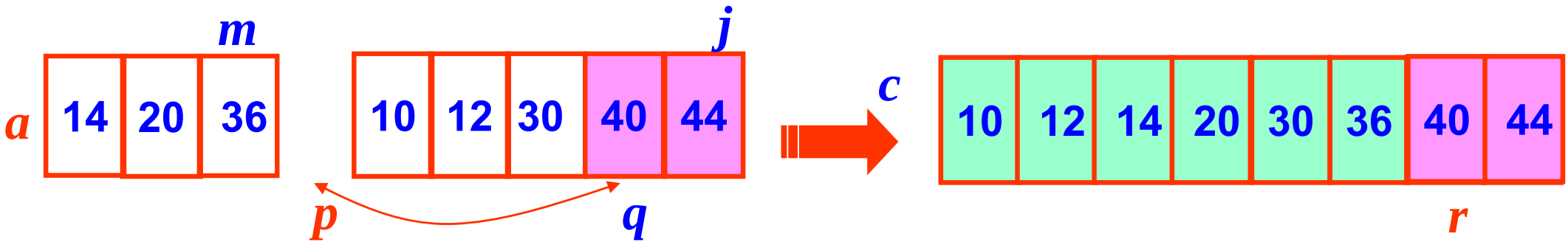
```
while (p ≤ m and q ≤ j) {


}
```

```
if (a[p] ≤ a[q]) {
    c[r] = a[p];
    p = p + 1;
}
else {
    c[r] = a[q];
    q = q + 1;
}
r = r +1;
```

✓ copy smaller value to c

Merging Two Sorted Arrays

Copy the remainder



- ✓ all of the data from the first array have been copied to the output array; we conclude by copying the remainder of the second array to the output array
- ✓ What is the condition: $p > m$ *or* $q > j$

- ✓ copy the remainder to c

```
while (q ≤ j) {
    c[r] = a[q];
    q = q + 1;
    r = r + 1;
}
```

Summary: Algorithm Merge

- This algorithm receives as input indexes *i*, *m*, and *j*, and an array *a*, where *a[i]*, ... , *a[m]* and *a[m +1]*, ... , *a[j]* are each sorted in non-decreasing order.
  - ☞ These two non-decreasing subarrays are merged into a single nondecreasing array

Put all together: Algorithm Merge

```
merge (a,i,m,j) {
    p = i           // index in a[i], ... , a[m]
    q = m + 1       // index in a[m + 1], ... , a[j]
    r = i           // index in a local array c
    while (p ≤ m and q ≤ j) {
        // copy smaller value to c
        if (a[p] ≤ a[q]) {
                c[r] = a[p]
                p = p + 1       // next
        }
        else {
                c[r] = a[q]
                q = q + 1       // next
        }
        r = r + 1
    }
    ...
    // copy the remainder
```

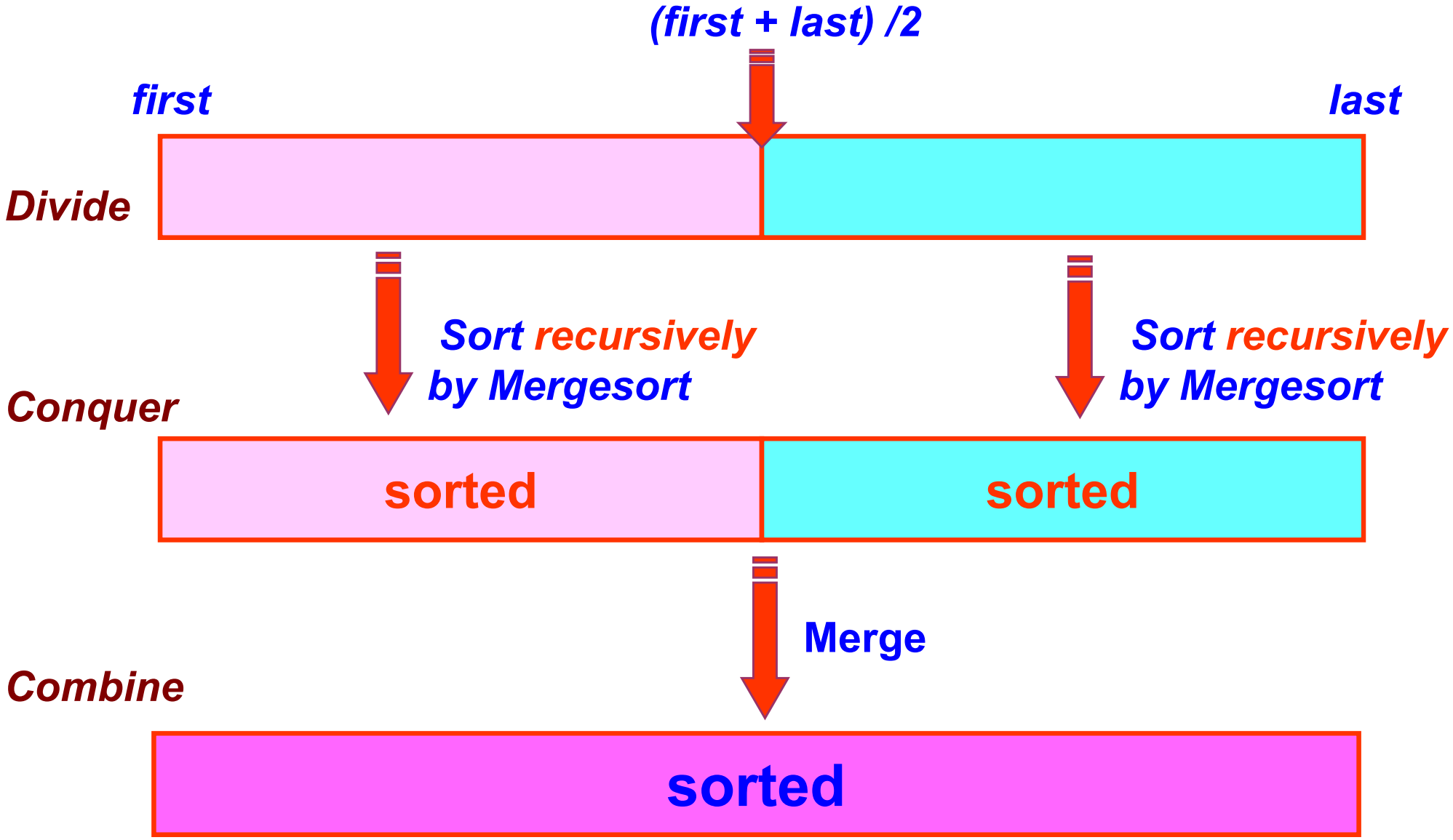
... Algorithm Merge

```
    ...
    // copy remainder, if any, of first sub-array to c
    while (p ≤ m) {
        c[r] = a[p]
        p = p + 1
        r = r + 1
    }
    // copy remainder, if any, of second sub-array to c
    while (q ≤ j) {
        c[r] = a[q]
        q = q + 1
        r = r + 1
    }
    // copy c back to a
    for r = i to j
        a[r] = c[r]
}
```

Merge sort

- Merge algorithm can be used for sorting
  - Merge-sort
- Mergesort
  - If the array has single element, stop
  - Divide the array to be sorted into two *nearly* equal parts.
  - Each part is then sorted using mergesort.
  - The two sorted subarrays are then merged into one sorted array

Merge-Sort

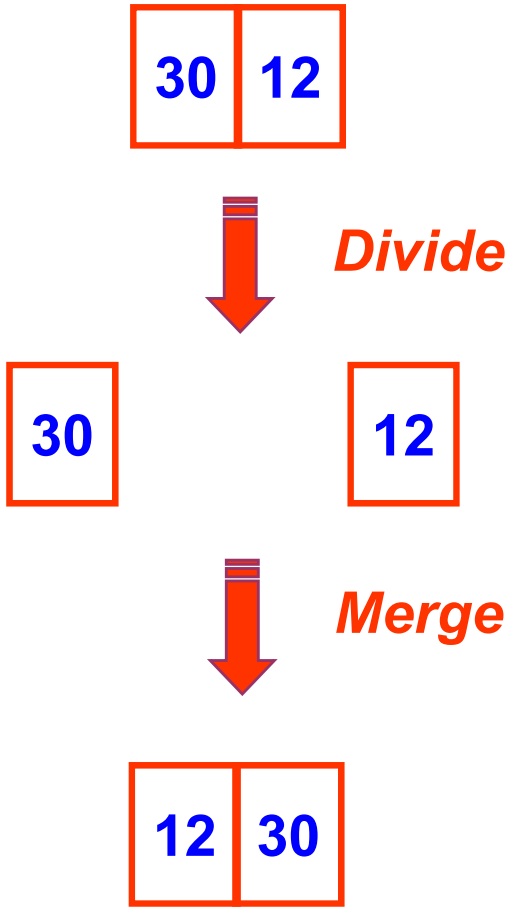

(first + last) /2
first
last
Divide
Sort recursively
by Mergesort
Sort recursively
by Mergesort
Conquer
sorted
sorted
Merge
Combine
sorted

Merge Sort: Problem-solving strategy

- The merge-sort algorithm closely follows the divide-and-conquer paradigm

1. *Divide:* Divide the *n*-element sequence to be sorted into two subsequences of *n/2* element each
2. *Conquer:* Sort the two subsequences recursively using merge-sort.
3. *Combine*: Merge the two sorted subsequences to produce the sorted answer.

Merge Sort : Example

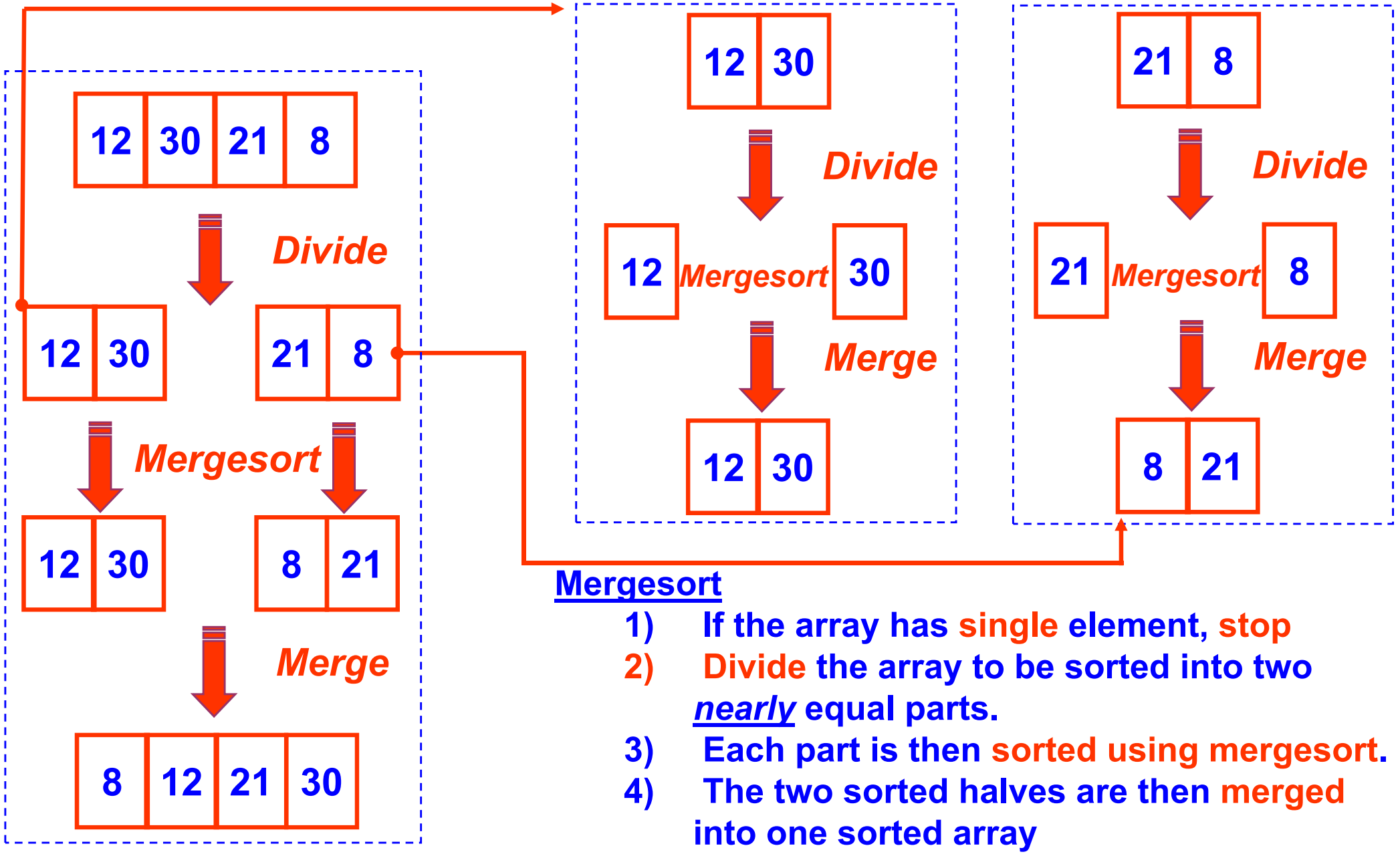


Sort each part recursively by Mergesort

Mergesort

1) If the array has single element, stop
2) Divide the array to be sorted into two *nearly* equal parts.
3) Each part is then sorted using mergesort.
4) The two sorted halves are then merged into one sorted array

Merge Sort : Example



Mergesort

1) If the array has single element, stop
2) Divide the array to be sorted into two *nearly* equal parts.
3) Each part is then sorted using mergesort.
4) The two sorted halves are then merged into one sorted array

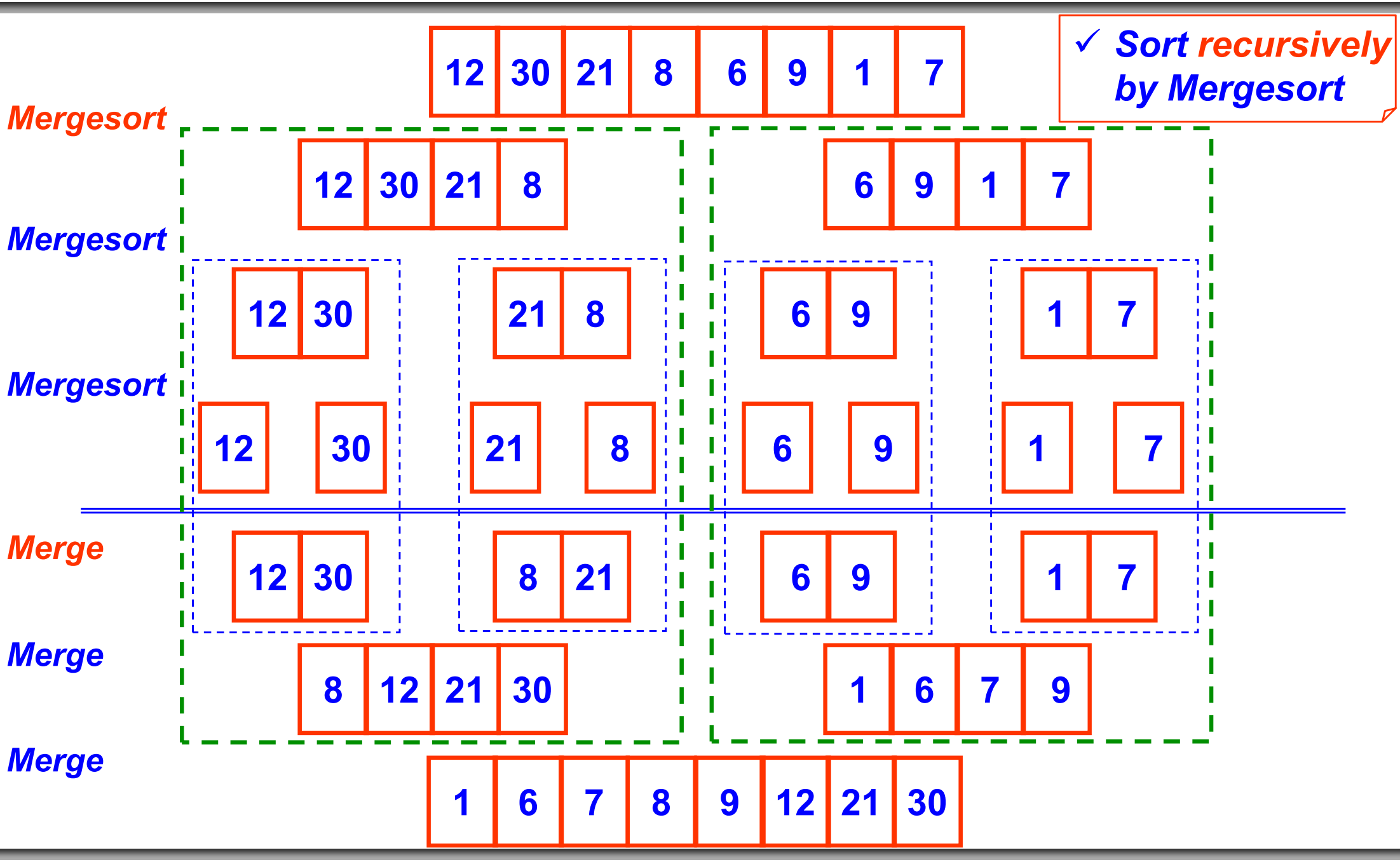
Algorithm Mergesort

This algorithm sorts the array *a[i], ... , a[j]* in nondecreasing order. It uses the merge algorithm

```
mergesort (a,i,j) {
    // if only one element, just return
    if (i == j)
        return
    // divide a into two nearly equal parts
    m = (i + j)/2
    // merge sort each half
    mergesort(a,i,m);
    mergesort(a,m + 1,j);
    // merge the two sorted halves
    merge(a,i,m,j)
}
```

For each part, you have to execute the entire algorithm!

Merge Sort : Example



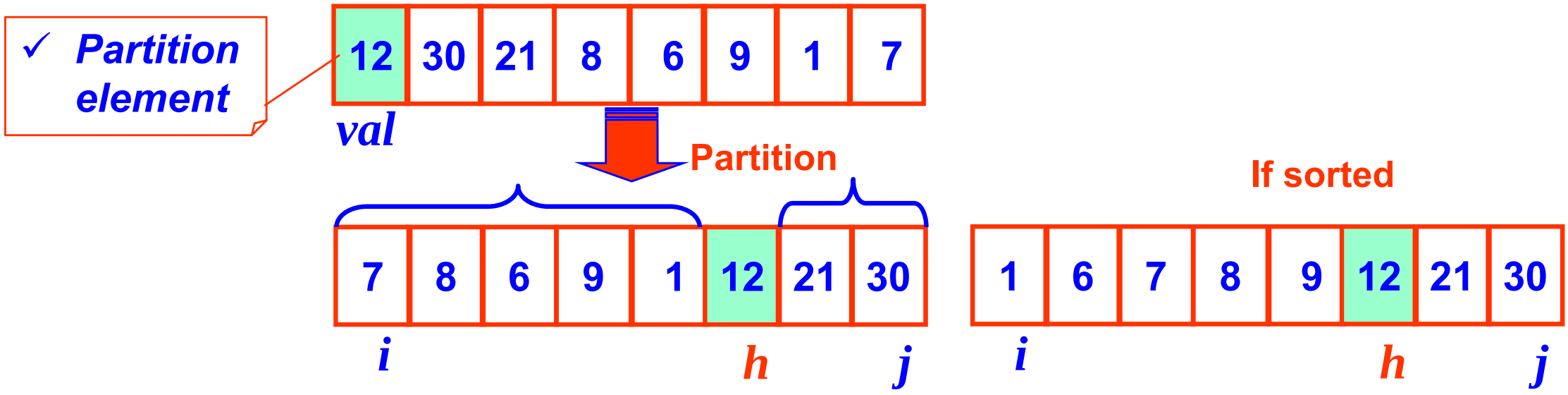
Partition

- The partition divides the array into two parts and sizes of the two parts can range from nearly equal to highly unequal.
  - ☞ Unlike mergesort, which divides the array into two nearly equal parts.
- The division depends on a particular element, called the partition element, that is chosen.

> ✓ The partition element is also called a pivot

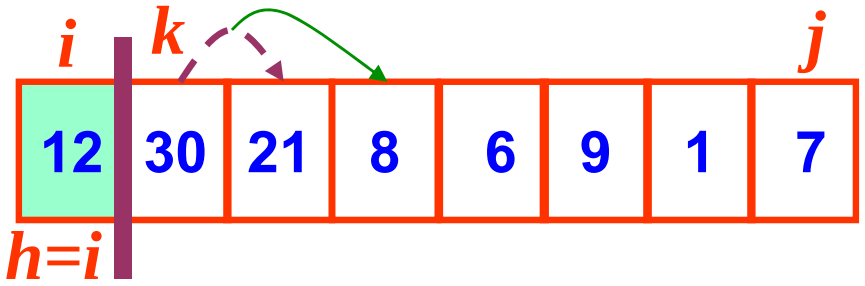
What is Algorithm Partition

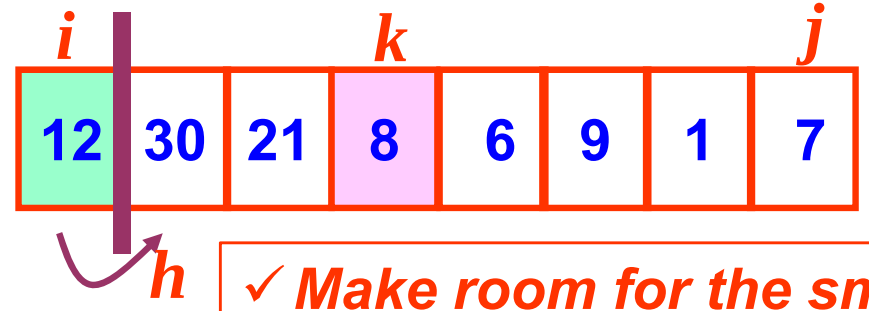
- ✓ This algorithm partitions the array *a*[*i*], ... , *a*[*j*] by inserting *val* = *a*[*i*] at the index *h* where it would be if the array was sorted.
- ✓ When the algorithm concludes, values at indexes less than *h* are less than *val*, and values at indexes greater than *h* are greater than or equal to *val*.
- ✓ The algorithm returns the index *h*.



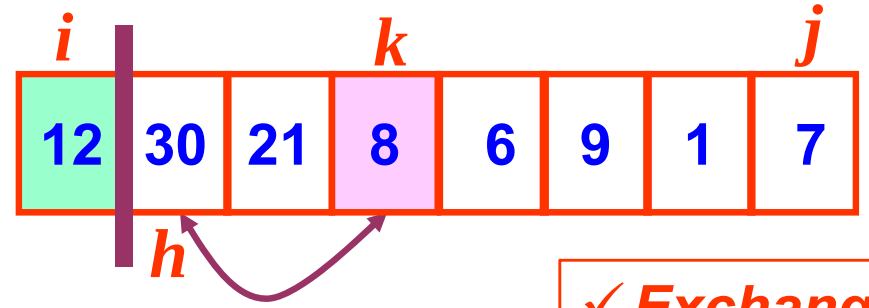
Partition: Idea

✓ Scan for the smaller





✓ Make room for the smaller one



✓ Exchange

Idea:

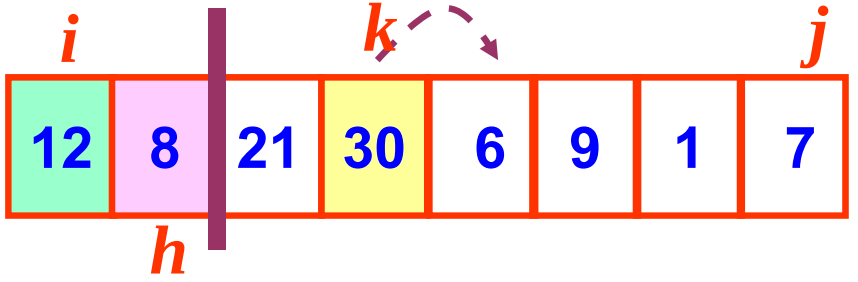
scan from left for element smaller than *val* = a[*i*]

Increase *h (make room for smaller)*

exchange

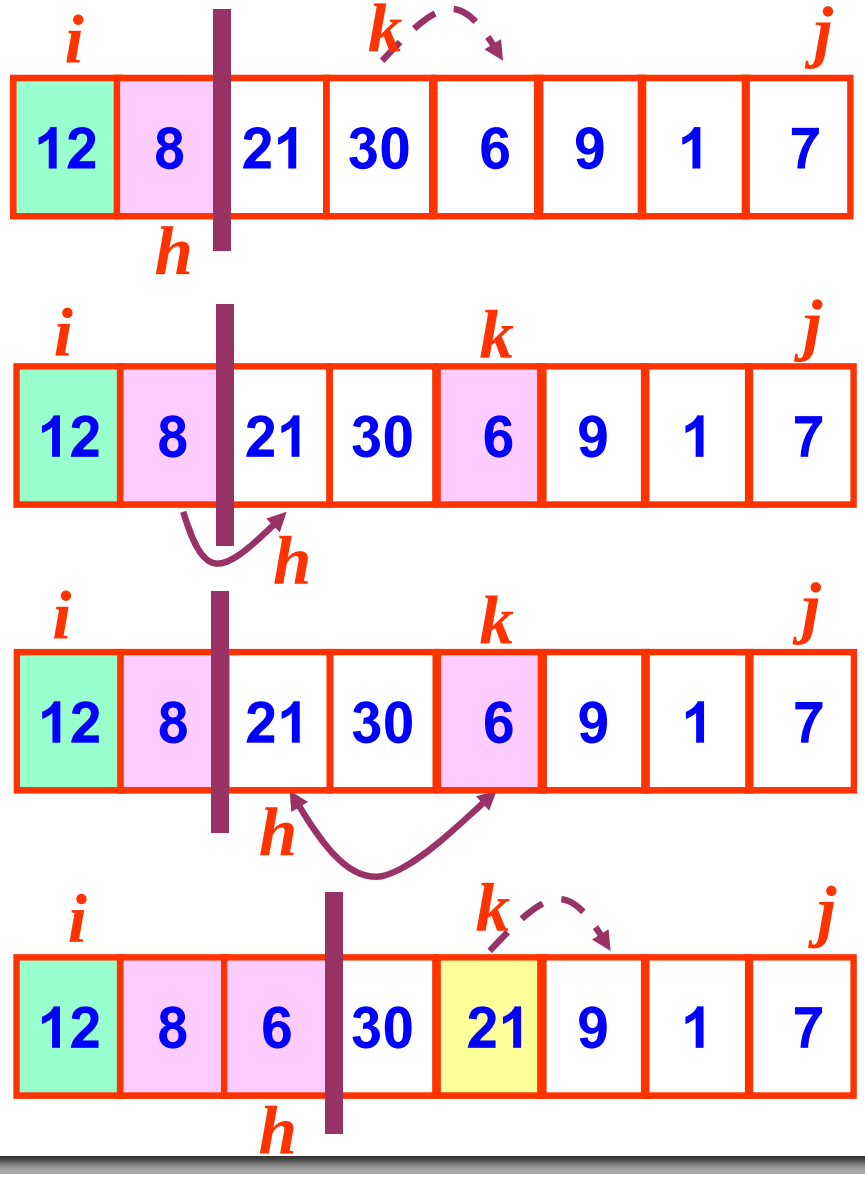
repeat until the last element of the array

```
val=a[i]; h=i;
for k = i + 1 to j
    if (a[k] < val) {
        h = h + 1;
        swap(a[h],a[k]);
    }
```



Partition: Idea

✓ Scan for the smaller



Idea:

scan from left for element smaller than *val* = a[*i*]

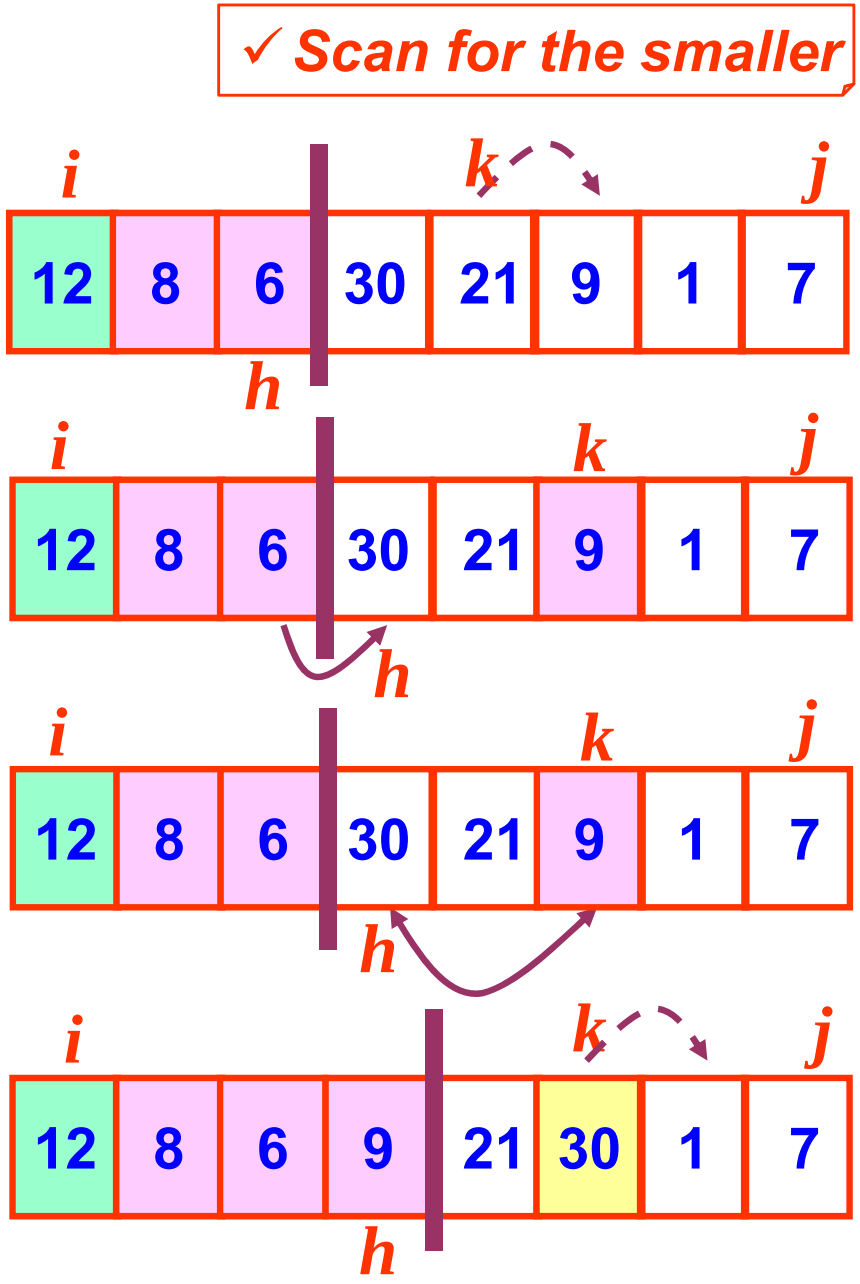
Increase *h (make room for smaller)*

exchange

repeat until the last element of the array

```
val=a[i]; h=i;
for k = i + 1 to j
    if (a[k] < val) {
        h = h + 1;
        swap(a[h],a[k]);
    }
```

Partition: Idea



Idea:

scan from left for element smaller than *val* = a[*i*]

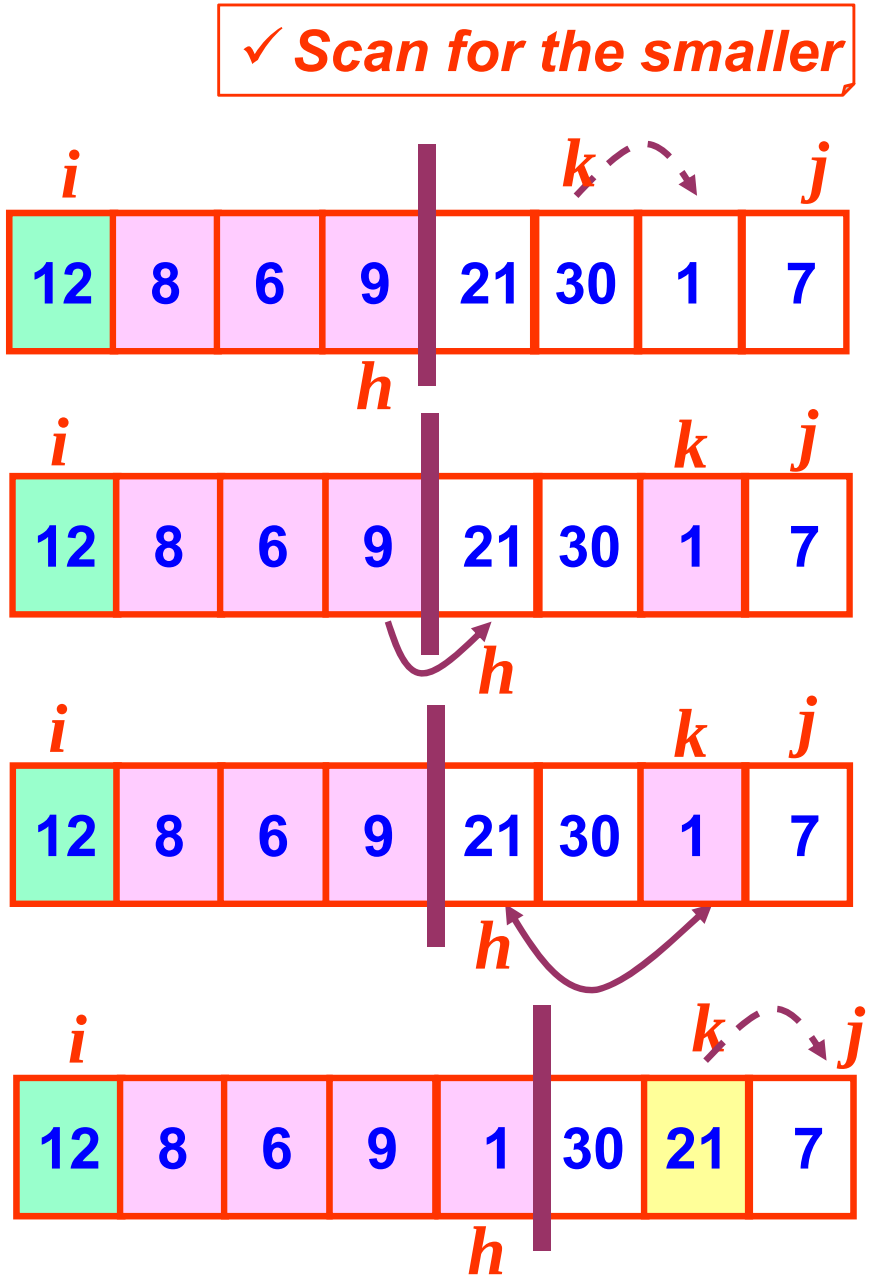
Increase *h (make room for smaller)*

exchange

repeat until the last element of the array

```
val=a[i]; h=i;
for k = i + 1 to j
    if (a[k] < val) {
        h = h + 1;
        swap(a[h],a[k]);
    }
```

Partition: Idea



Idea:

scan from left for element smaller than *val* = a[*i*]

Increase *h (make room for smaller)*

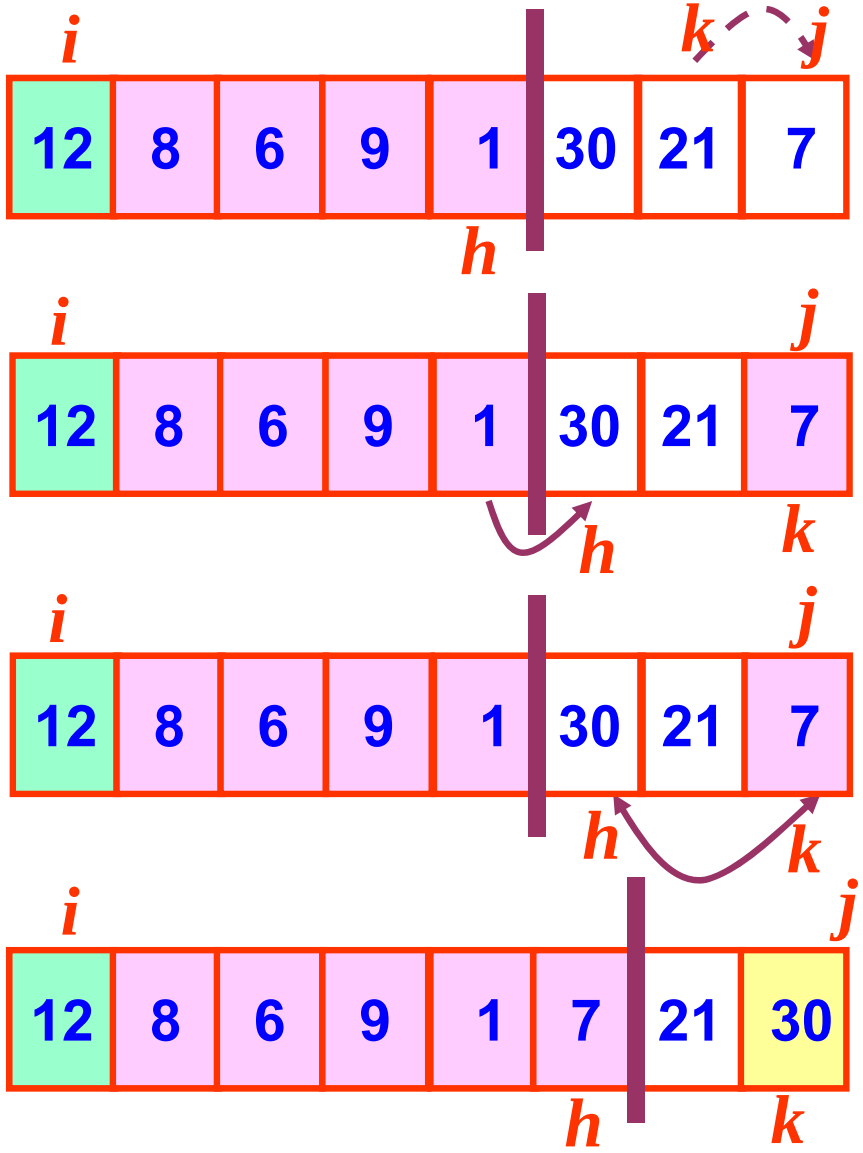
exchange

repeat until the last element of the array

```
val=a[i]; h=i;
for k = i + 1 to j
    if (a[k] < val) {
        h = h + 1;
        swap(a[h],a[k]);
    }
```

Partition: Idea

✓ Scan for the smaller



Idea:

scan from left for element smaller than *val* = a[*i*]

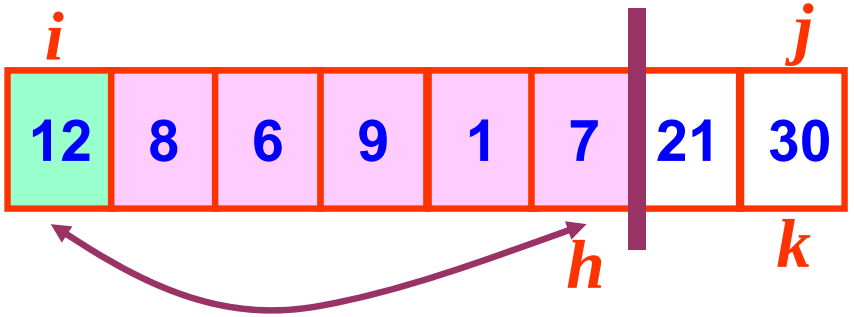
Increase *h (make room for smaller)*

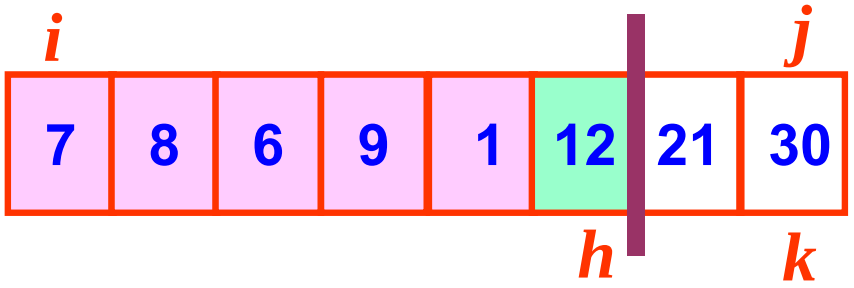
exchange

repeat until the last element of the array

```
val=a[i]; h=i;
for k = i + 1 to j
    if (a[k] < val) {
        h = h + 1;
        swap(a[h],a[k]);
    }
```

Partition: Idea





✓ *Exchange*

Idea:

scan from left for element smaller than *val* = *a[i]*

Increase *h (make room for smaller)*

exchange

repeat until the last element of the array

```
val=a[i]; h=i;
for k = i + 1 to j
   if (a[k] < val) {
      h = h + 1;
      swap(a[h],a[k]);
   }
swap (a[i],a[h])
      return h
```

Partition Algorithm

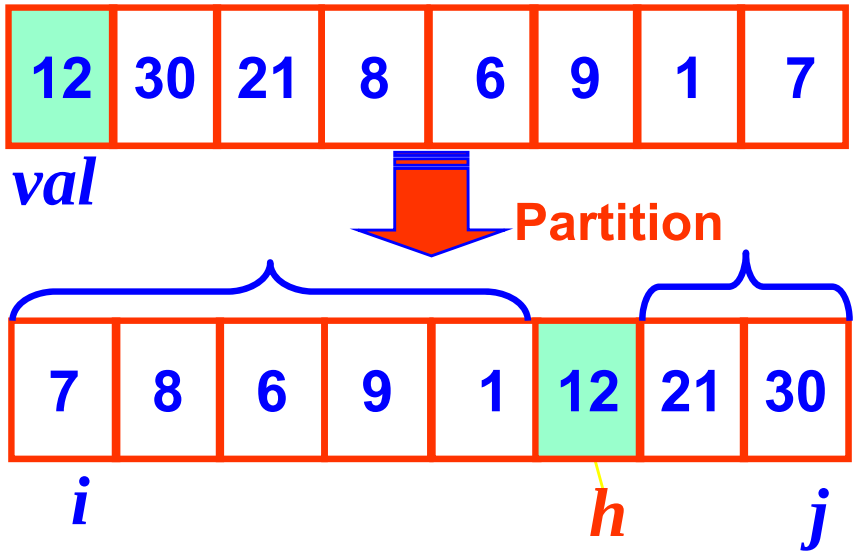
Idea:

scan from left for element smaller than *val* = a[*i*]

Increase *h (make room for smaller)*

exchange

repeat until the last element of the array




```
partition(a,i,j) {
    val = a[i];
    h = i
    for k = i + 1 to j
        if (a[k] < val) {
            h = h + 1
            swap(a[h],a[k])
        }
    swap (a[i],a[h])
    return h
}
```

✓ scan

✓ For smaller

✓ swap

Quick Sort

- The quick-sort, like merge-sort, closely follows the divide-and-conquer paradigm

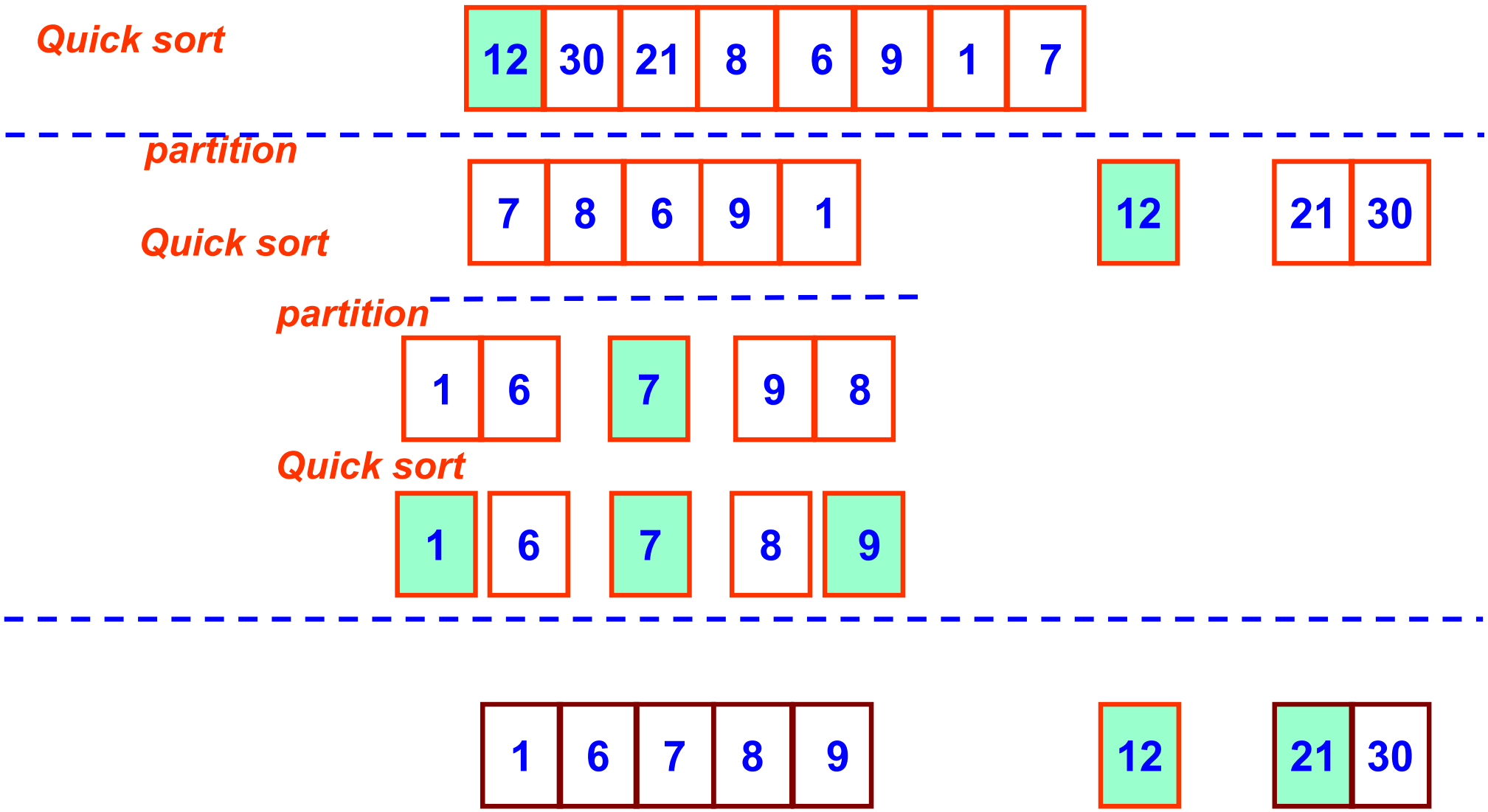
1. *Divide:* Partition (rearrange) the array $A[p..r]$ into two (possibly empty) subarray $A[p..h-1]$ and $A[h+1..r]$ such that each element of $A[p..h-1]$ is less than or equal $A[h]$, which is in turn less than or equal to each element of $A[h+1..r]$. This partition procedure returns the index h
2. *Conquer:* Sort the two subarrays $A[p..h-1]$ and $A[h+1..r]$ by recursively calls to quick-sort.
3. *Combine:* Since the subarrays are sorted in place, no work is needed to combine them: the entire array $A[p..r]$ is now sorted

Algorithm Quicksort

This algorithm sorts the array *a[p], ... , a[r]* by using the partition algorithm

```
Input Parameters: a,p,r
Output Parameters: a

quicksort(a,p,r) {
    if (p < r) {
        h = partition(a,p,r);      ❶
        quicksort(a,p,h - 1);      ❷
        quicksort(a,h + 1,r);      ❸
    }
}
```

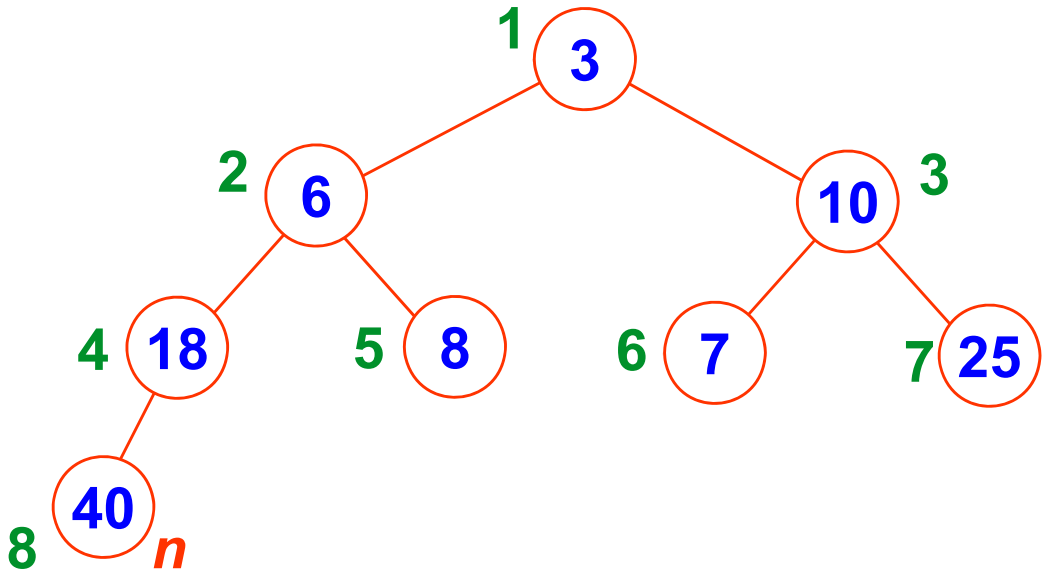
Quick sort


Quick sort
12 30 21 8 6 9 1 7
partition
Quick sort
7 8 6 9 1
12
21 30
partition
1 6
7
9 8
Quick sort
1 6
7
8 9
1 6 7 8 9
12
21 30

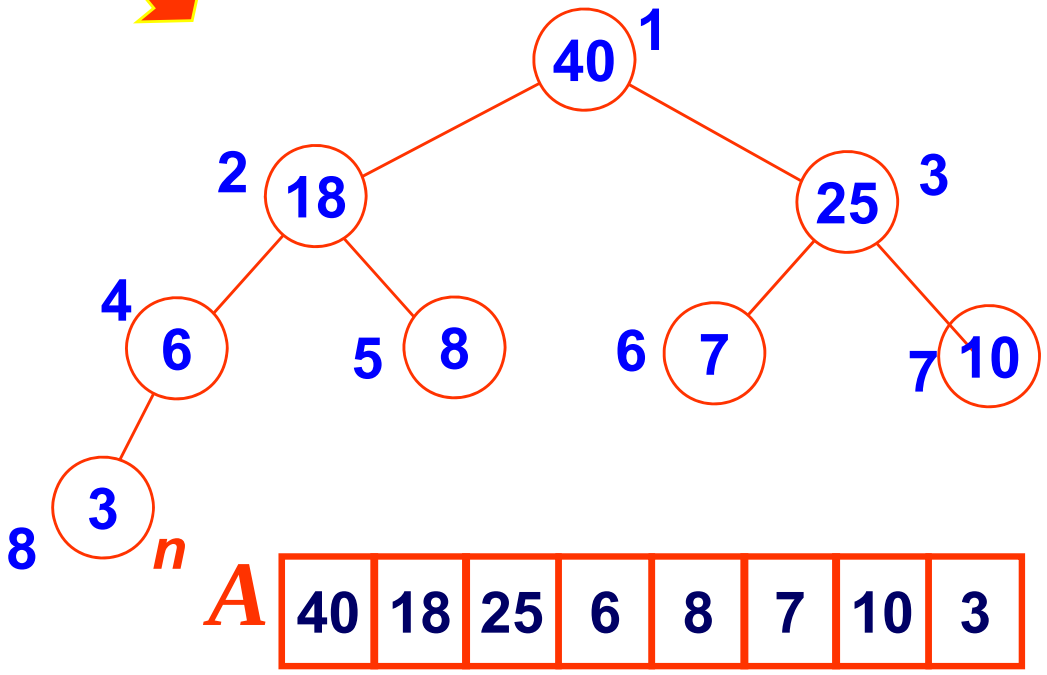
Heapsort

- Recall: mergesort has good worst case complexity $O(n \log n)$, but requires auxiliary arrays to be created.
- We now look at another sorting algorithm: heapsort
  - same worst case complexity $O(n \log n)$
  - can be done in place

Heapsort: idea



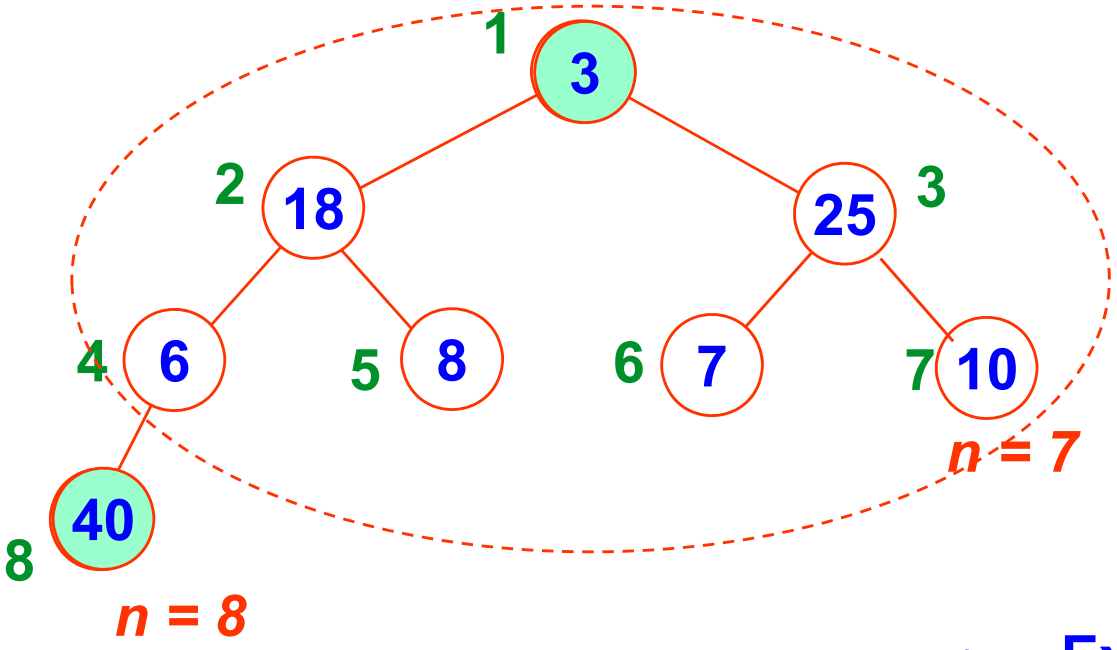
Make a maxheap:
heapify(A)

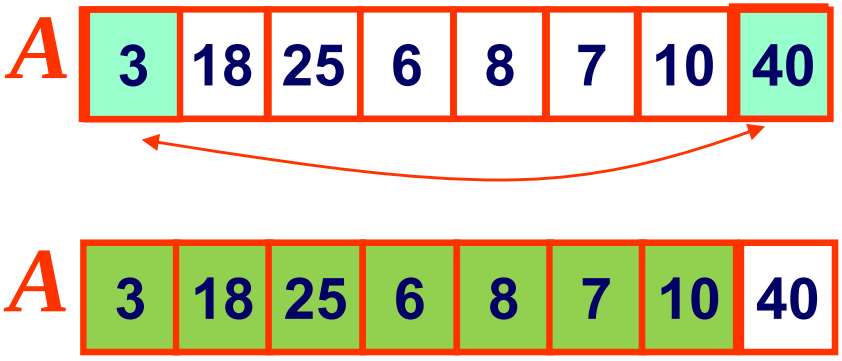


A	3	6	10	18	8	7	25	40

A	40	18	25	6	8	7	10	3

Heapsort: idea

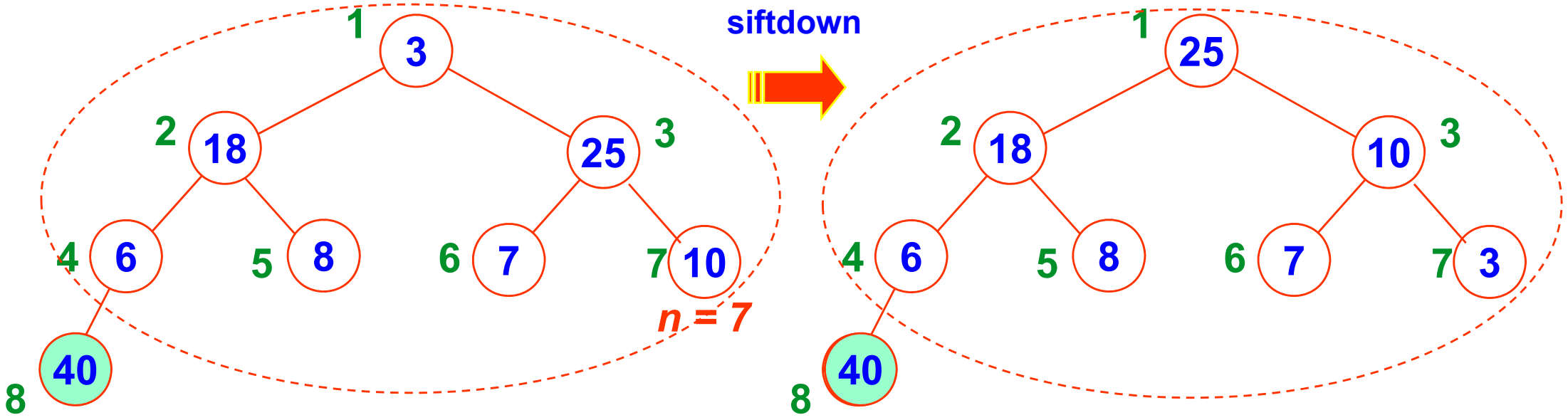




- Exchange A[1] with A[n]
- Treat A[1..n-1] as new heap.
- Repeat procedure.

Similar to deleting root of A

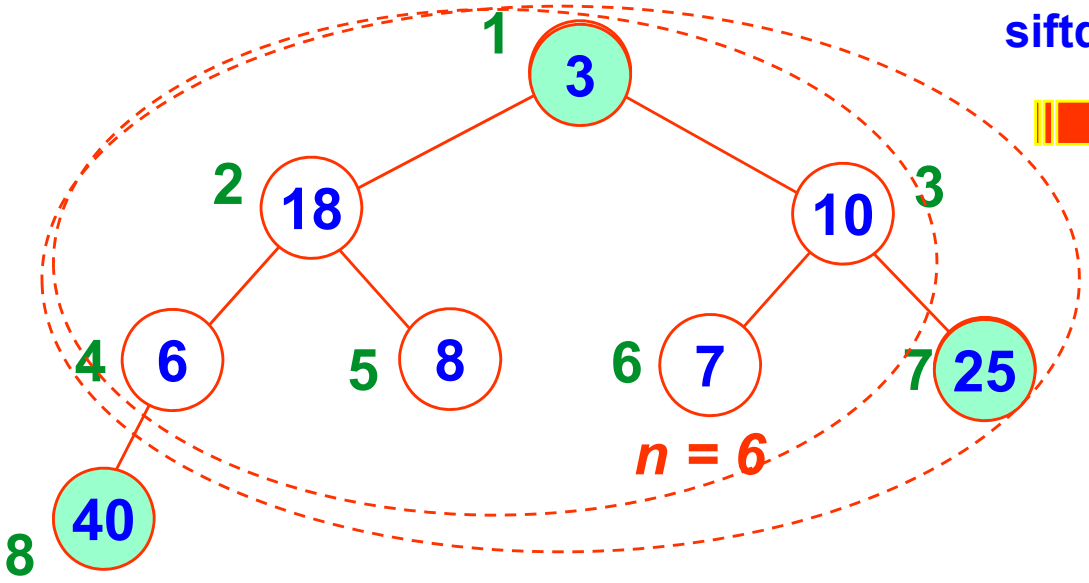
Heapsort: idea



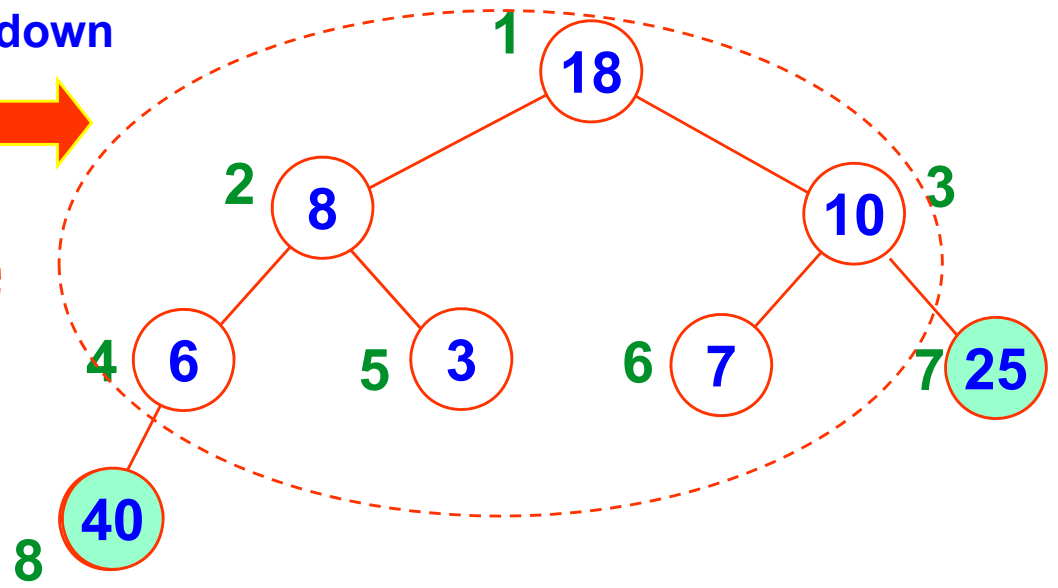
- Exchange A[1] with A[n]
- Treat A[1..n-1] as new heap.
- Repeat procedure.

Similar to deleting root of A

Heapsort: idea



siftdown

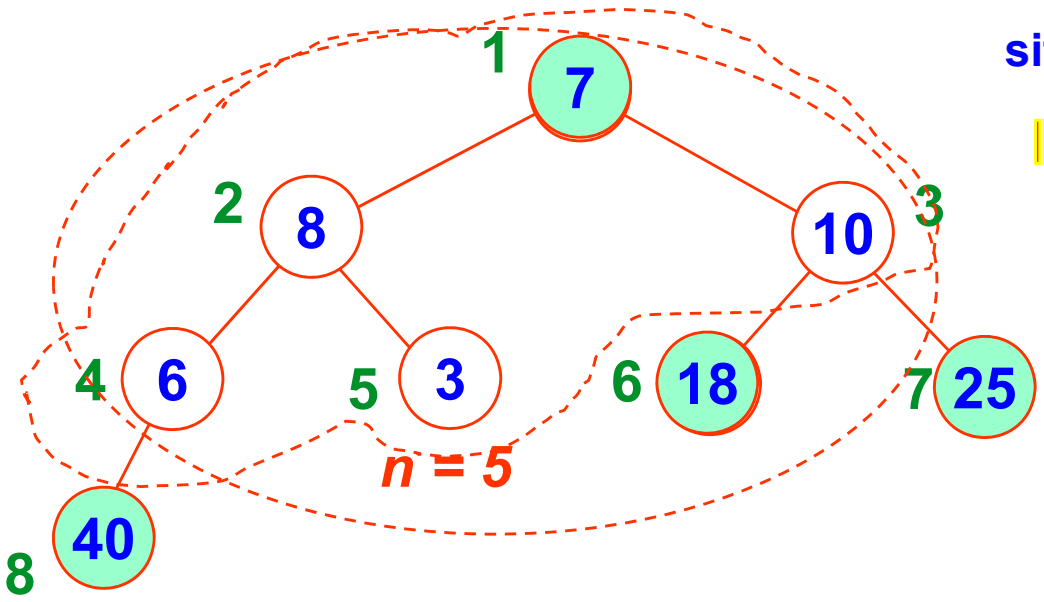


- Exchange A[1] with A[n]
- Treat A[1..n-1] as new heap.
- Repeat procedure.

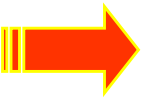
Similar to deleting root of A

siftdown

Heapsort: idea

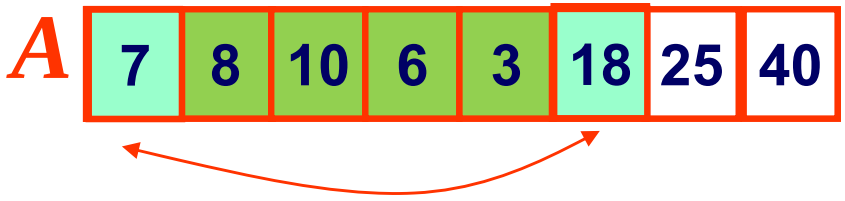


siftdown

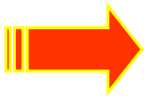


- Exchange A[1] with A[n]
- Treat A[1..n-1] as new heap.
- Repeat procedure.

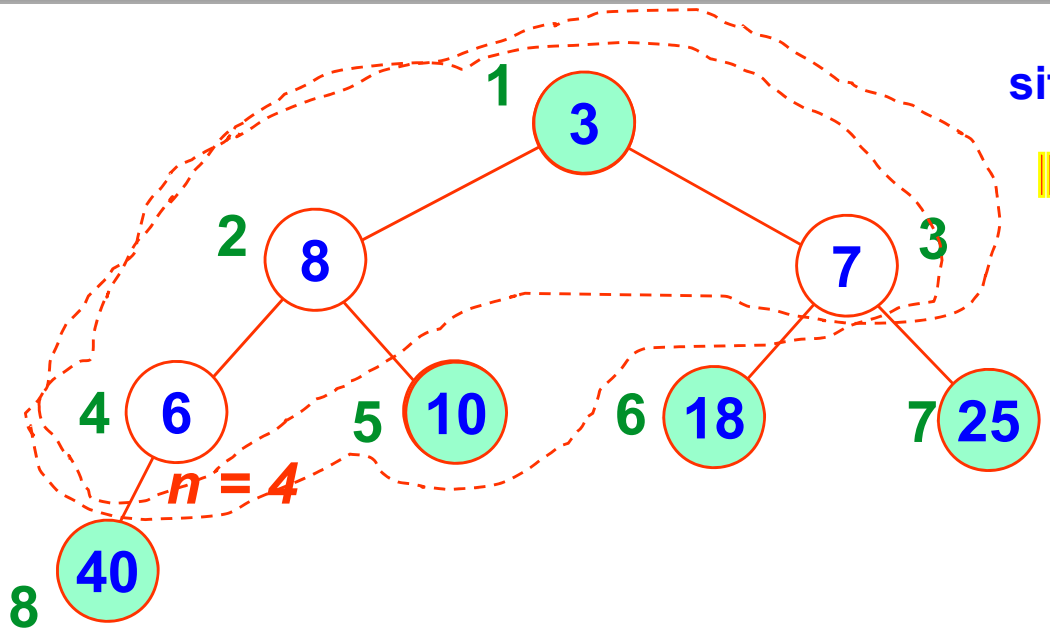
Similar to deleting root of A



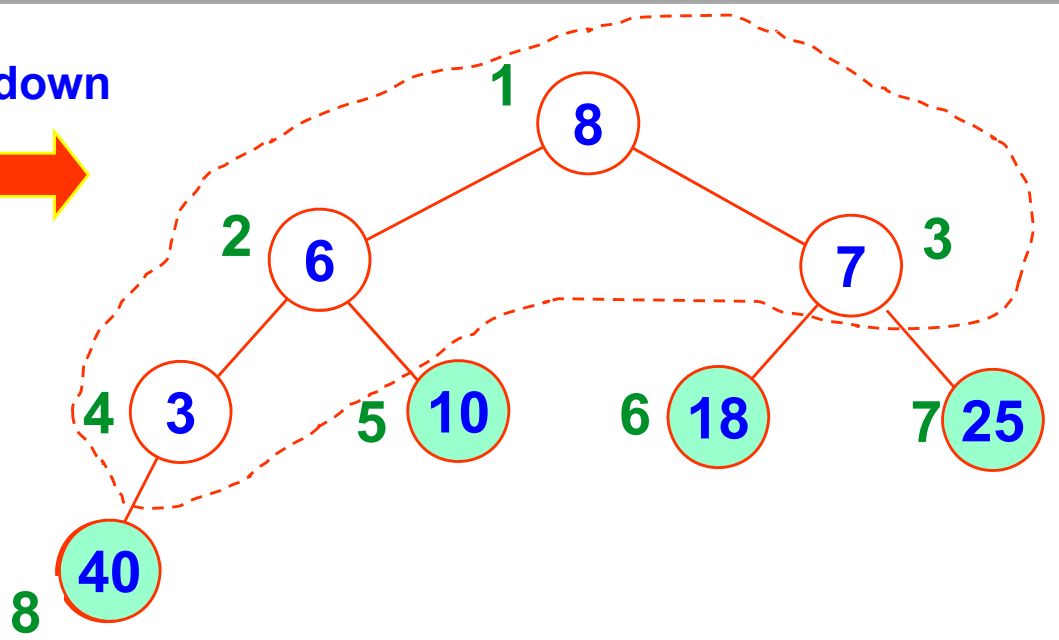
siftdown



Heapsort: idea



siftdown

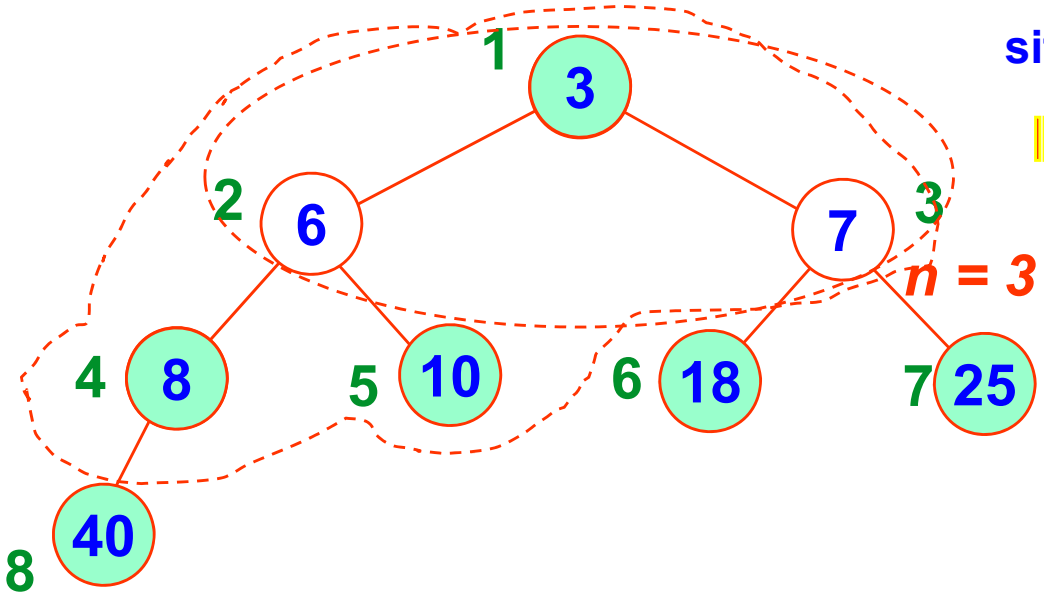


- Exchange A[1] with A[n]
- Treat A[1..n-1] as new heap.
- Repeat procedure.

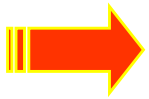
Similar to deleting root of A

siftdown

Heapsort: idea



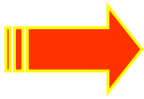
siftdown

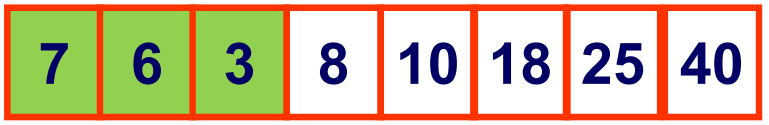


- Exchange A[1] with A[n]
- Treat A[1..n-1] as new heap.
- Repeat procedure.

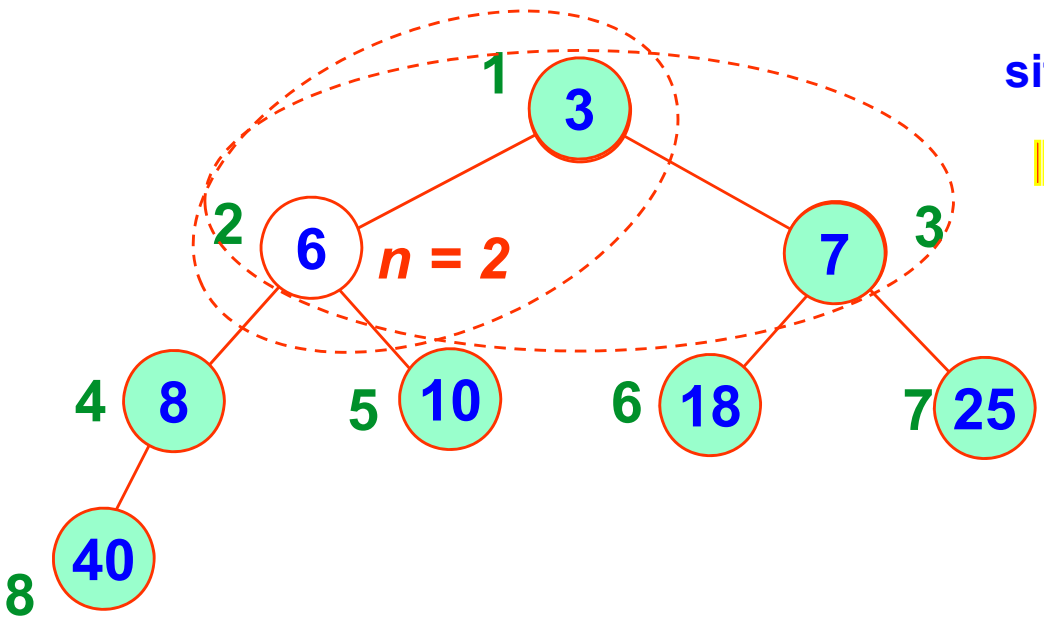
Similar to deleting root of A

siftdown

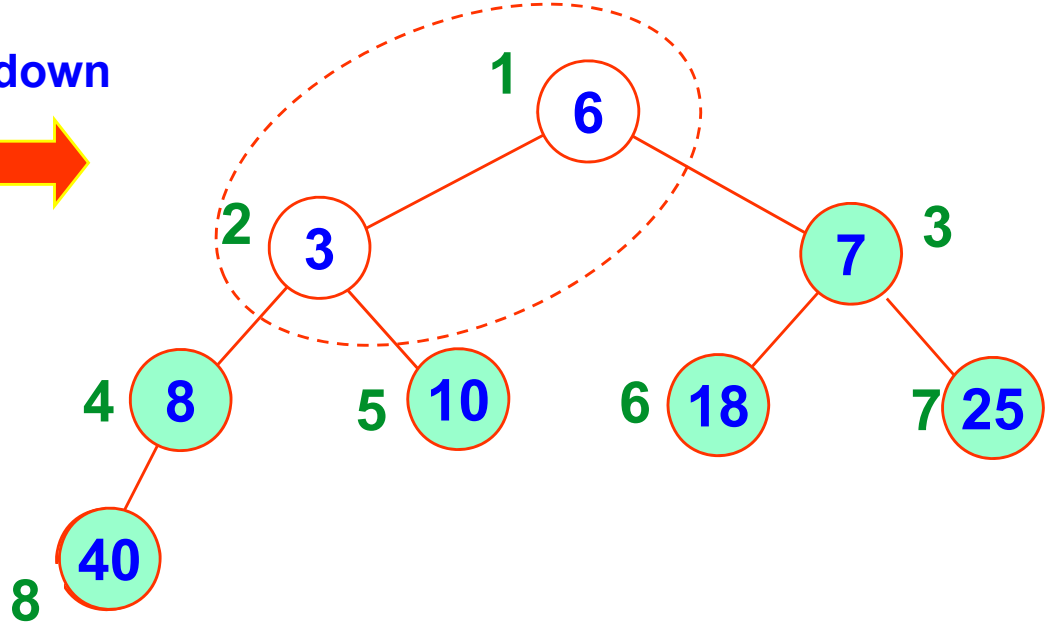




Heapsort: idea

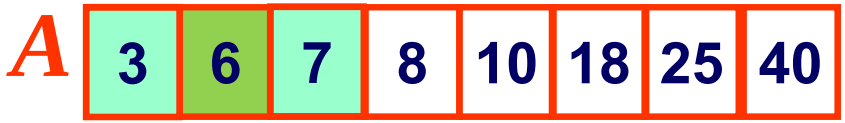


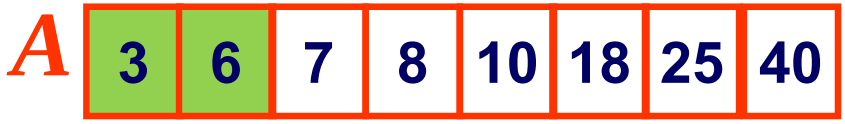
siftdown



- Exchange A[1] with A[n]
- Treat A[1..n-1] as new heap.
- Repeat procedure.

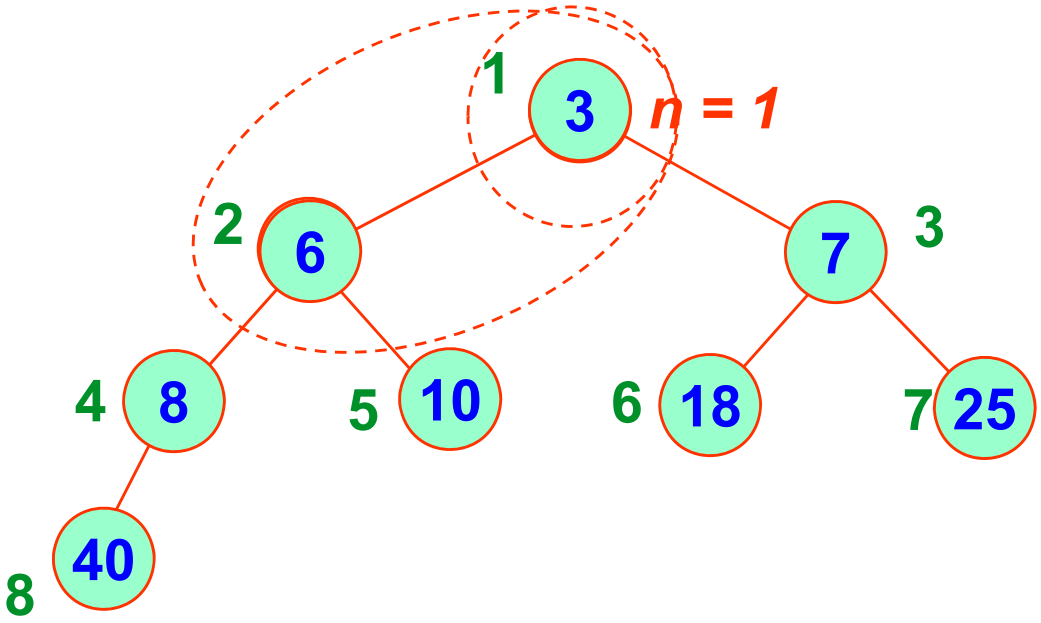
Similar to deleting root of A

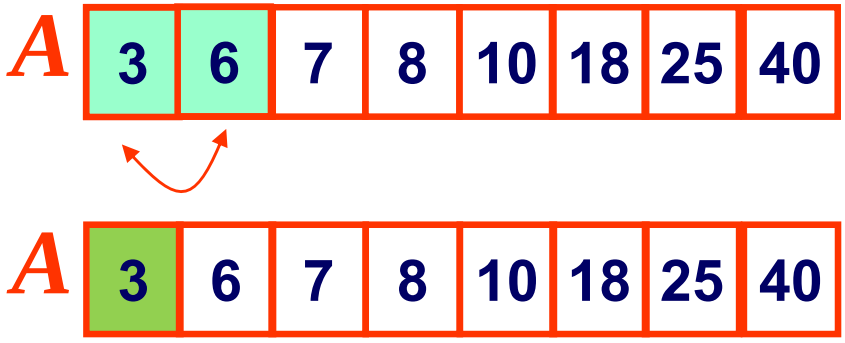




siftdown

Heapsort: idea





- Exchange A[1] with A[n]
- Treat A[1..n-1] as new heap.
- Repeat procedure.

Similar to deleting root of A

Heapsort

```
heapsort(A) {
  // change A into a maxheap
  heapify(A)
  for (i = A.last downto 2) {
    // shift ith largest element
    // to its correct place
    swap(A,1,i)
    // reduce size of heap
    A.heapsize = A.heapsize - 1
    // maintain maxheap
    siftdown(A,1)
  }
}
```

$O(n)$

$O(1)$

$O(n \log n)$

$O(\log n)$

Total: $O(\log n)$

Learning Takeaway

- Heaps are very useful data structures.
  - Get "weakly sorted" array.

- Main operations have low complexity:
  - siftdown $O(log\ n)$
  - heapify $O(n)$
  - delete and insert $O(log\ n)$

- Applications: heapsort, priority queues, …

Sorting in Linear Time

Counting Sort

- A[1..*n*] input elements are integers from 1 to *k*.
  - For example, A[1, …, 100] with integers 1, …, 10
- How to sort the array in linear time?
- Idea: if j elements are ≤ A[i], then A[i] can be put in the j-th position of the array.
  - For example, suppose 100 elements ≤ A[i], then where should we put A[i]?
  - After sorting, right location for A[i] is position 100 (100th element) in the array.

Counting Sort

- **Question I: If the number (j) of elements ≤ A[i] has been found (for all i), where to store these number information (j)?**
  - Use *count* array to store the number of elements ≤ A[i], e.g, if A[i]=m, element m in *count* array counts how many elements ≤ m.
  - For example, if A[i]=5, element 5 in *count* array denotes the number of elements ≤ m.

- **Question II: How to find the number of elements ≤ A[i] efficiently?**

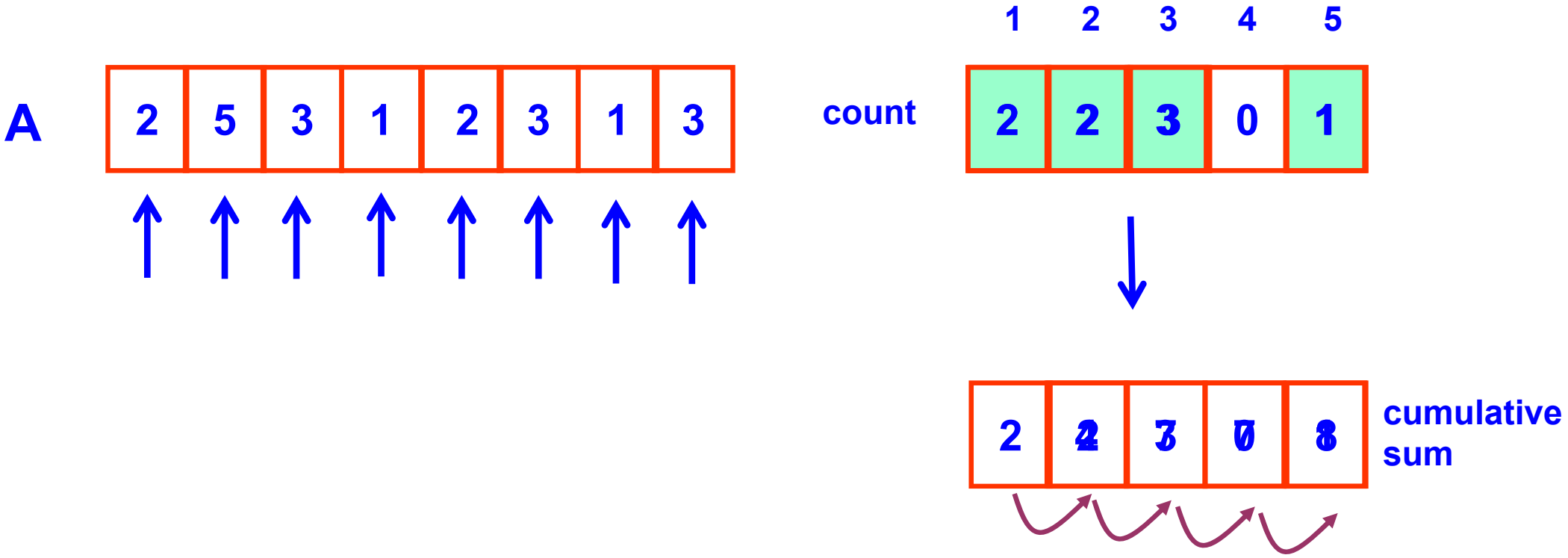
Counting Sort

- Initialize another array count[1, …, *k*]
- Go through A[1, ..., *n*]
  - ☞ increase count[m] by 1 ⟵ this can be done because A[i] is a positive integer (e.g, m)
- *count*[m] now contains the number of elements of A of which values are equal to m
- To get the number of elements ≤ m : cumulative sum

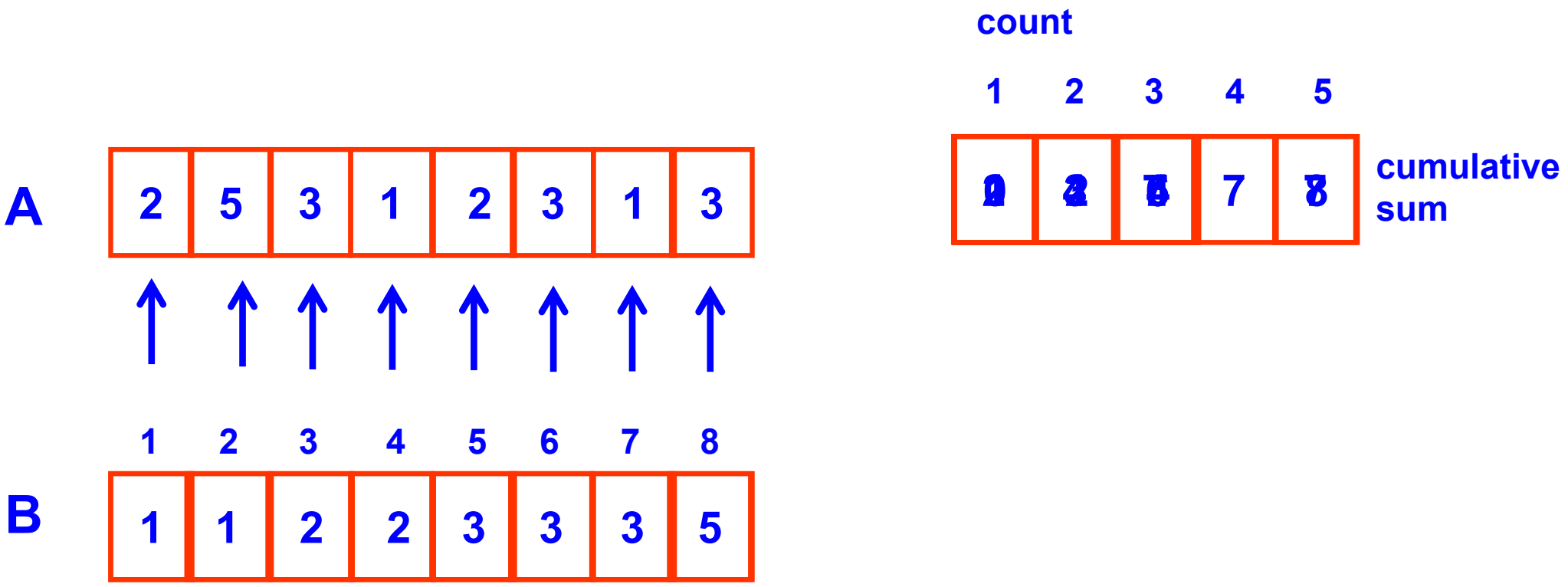
```
for m = 1 to
    count[m] = count[m -1] + count[m]
```

- Use *count* array to sort A[1, … , *n*] by placing each A[i] (=m) in the right location given by *count*[m].

Counting Sort Example



Counting Sort Example



This is an important property for many applications.
E.g. used in radix sort later.

We start from the back of A so that numbers with the same value appear in B in the same order that they appear in A. This sorting is said to be *stable*.

counting_sort pseudo code

```
A[1..n] – input array of n integers in the range 1 to k
B[1..n] – output array containing the sorted elements from A

counting_sort(A,B,k) {
        n = A.last
        for i = 1 to k
                count[i] = 0                                    } O(k)
        // count[i] now contains the no. of elements = i
        for j = 1 to n
                 m=A[j]; count[m] = count[m] + 1                } O(n)
        // count[i] now contains no. of elements ≤ i
        for i = 1 to k
                count[m] = count[m-1] + count[m]                } O(k)

        for j = n downto 1 {
                m=A[j]; i=count[m]                              } O(n)
                B[i] = m; count[m] = count[m]-1
        }                                                       O(n+k)
}
```

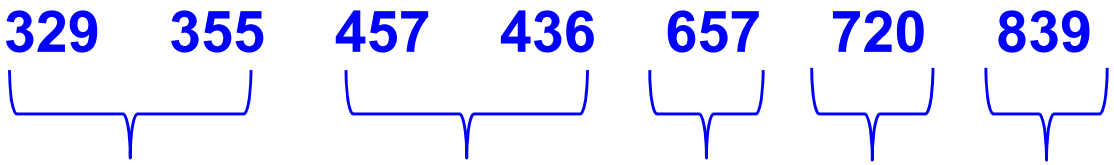
Radix Sort

- Aim: sort *n* words each with *d* digits

329	329
457	355
657	436
839	457
436	657
720	720
355	839

sort *n*=7 words each with *d*=3 digits 0 .. 9

- Human: sort according to most significant digits first

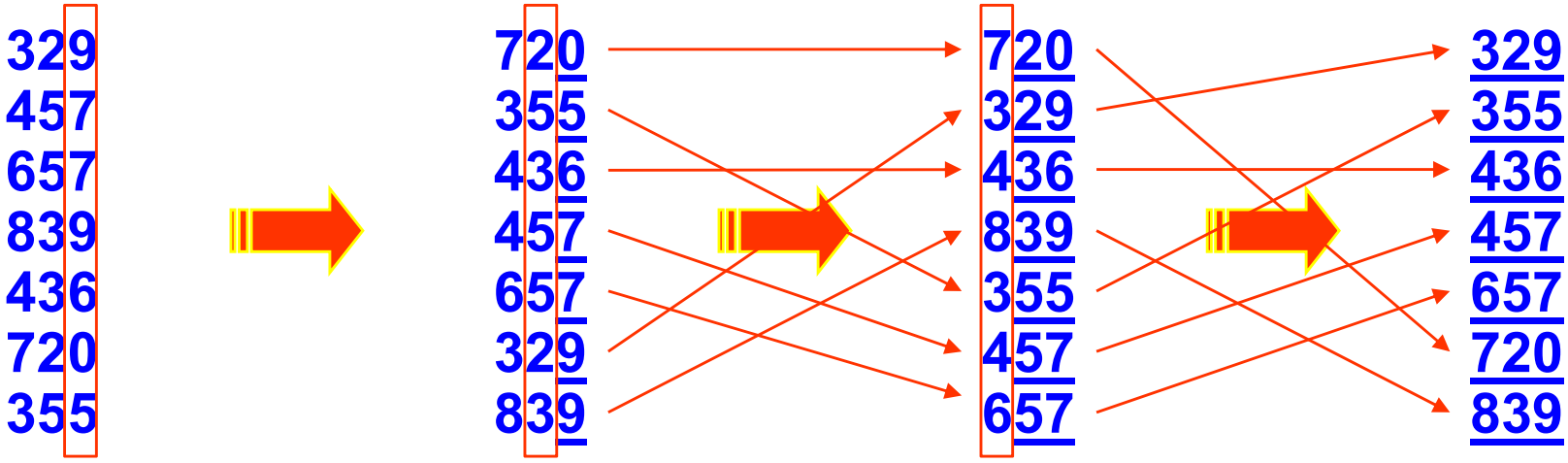


Sort each pile according to second significant digits and so on.

- Con: Need to keep track of many piles of numbers.

Radix Sort

- Amazingly, radix sort performs sorting on the *least significant* digit first.



- Starting from least significant digit, sort the digit using a stable sort like counting sort.
- Repeat till the most significant digit. When the higher significant digits are same, remain the order already sorted in lower significant digits.

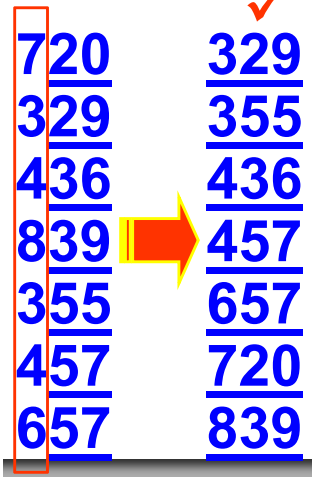
Radix Sort

```
radix_sort(A,d) {
    for i = 1 to d {
        sort A on digit i using a stable sort
    }
}
```

- Suppose we use counting sort to sort each digit in radix sort. Each digit is in range 1..k
  - Each counting sort - O(n+k)
  - d counting sorts are used - O(d(n+k))

Radix Sort Correctness

- Why does it work? Key: stable sort is used in each iteration.
- Proof by induction on the number of digits:
  - Basis step: d=1, counting sort gives correct sorting.
  - Inductive step: Suppose that radix sort is correct for numbers with $d-1$ digits.
  - After $d-1$ iterations, the numbers are sorted according to their low-order $d-1$ digits. The final sort orders the numbers by their d-th digit. Consider 2 numbers a and b with d-th digit a_d and b_d respectively.
    - ✓ If $a\_d<b\_d$, then $a<b$ regardless, so final sort is correct.
    - ✓ If $a\_d>b\_d$, then $a>b$ regardless, so final sort is correct.
    - ✓ If $a\_d=b\_d$, since final sort is stable, the relative order of a and b remains unchanged.
      - ❖ Correct ordering is determined by low-order $d-1$ digits since they have same d-th digit.
      - ❖ Induction hypothesis: a and b are in the correct ordering.



Breaking into digits

- How to break given elements into "digits"?
- n words, b bits/word
  - break into r-bit digits $\Longrightarrow$ $d=b/r$ digits/word
  - each digit is then in the range 0.. $k=2^r-1$
- Complexity: $O(b/r\ (n+2^r\ ))$
  - If $r=lg\ n$: $O(b/(lg\ n)\ (n+n))=O(bn/(lg\ n))$
  - If $r \gg lg\ n$: 2^r term dominates r term in denominator so complexity is at least $\Omega(bn/(lg\ n))$.
  - If $r \ll lg\ n$: $n+2^r=O(n)$ but b/r increases, so $\Omega(bn/(lg\ n))$.
- In general, choose $r \approx lg\ n$

Comparison with Quicksort

- Suppose we want to sort 2^16 32-bit numbers.
  - n=2^16, b=32
  - Radix sort: choose $r=lg$ 2^16=16, then $d=b/r=2$

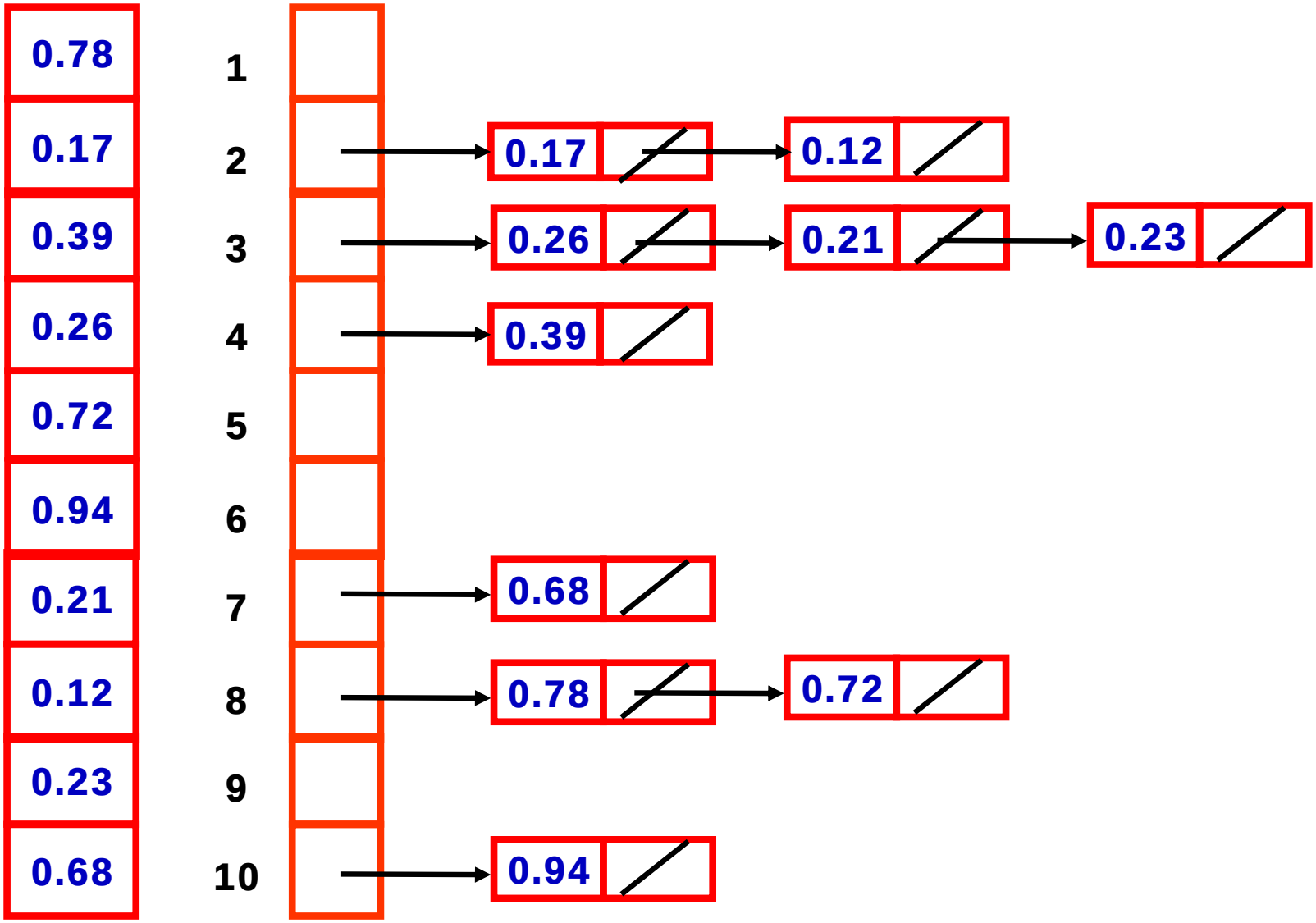
    $\Rightarrow$ 2 passes over the n numbers
  - Quicksort: $lg\ n$=16 passes over the n numbers
- What is the magic?
  - Counting sort uses the fact that keys are integers, not just comparison of keys.
  - Keys are used as array indices.
- However, actual speed depends on:
  - characteristics of the implementations
  - underlying machine (e.g., quicksort often uses hardware caches more effectively than radix sort)
- Counting sort also not in-place: storage issues.

Bucket Sort

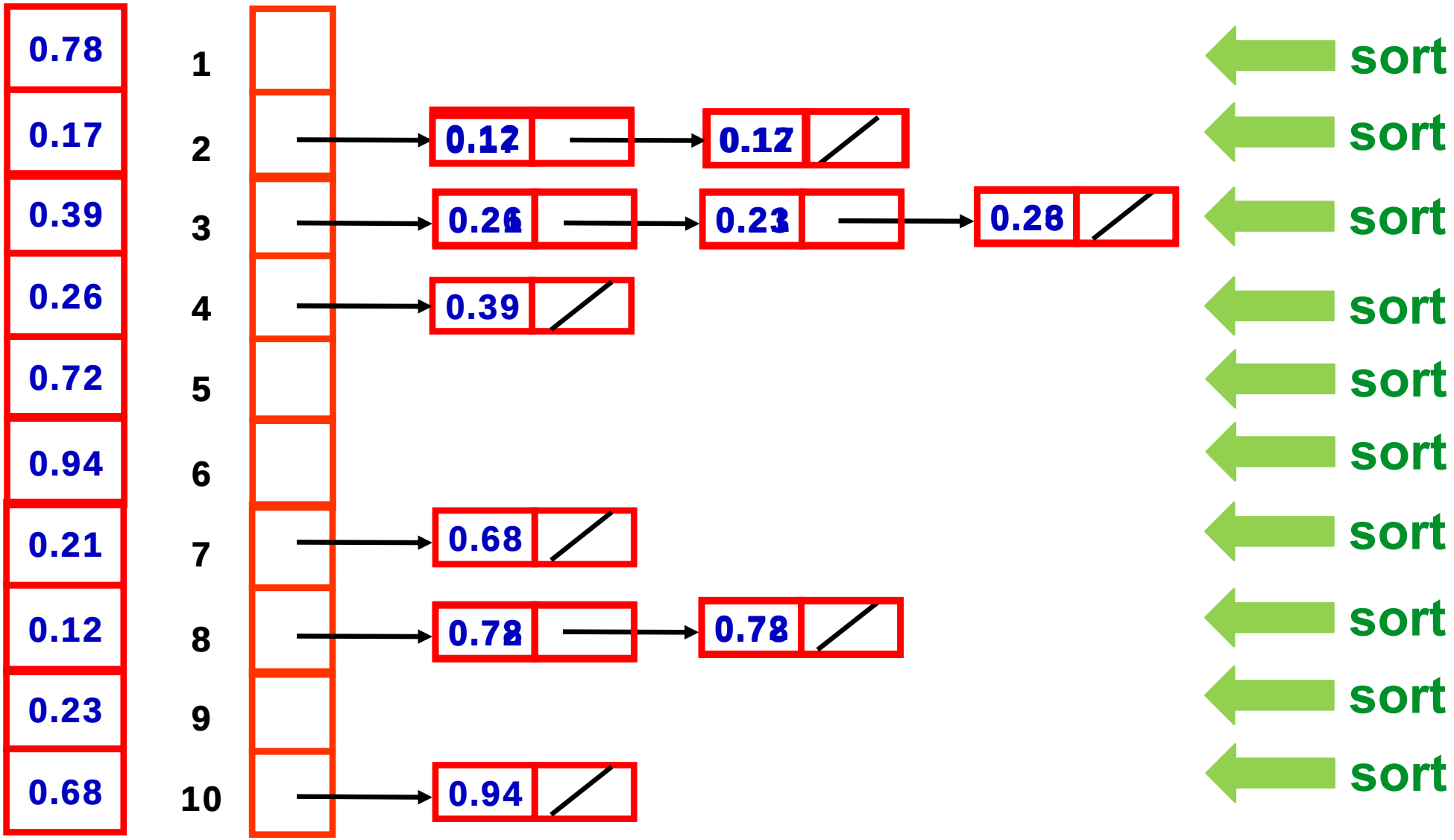
- Assumes *n* input is generated by a random process that distributes *n* values uniformly over [0,1).

- Idea:
  - divide [0,1) into n equal sized buckets represented by B[1..*n*]
  - distribute the given n inputs into the buckets
    - ✓ maintain a linked list for each bucket
  - sort each bucket (or linked list)
  - concatenate B[1], …, B[*n*] together to form final output

Bucket Sort


0.78
0.17
0.39
0.26
0.72
0.94
0.21
0.12
0.23
0.68
1
2
3
4
5
6
7
8
9
10
0.17
0.12
0.26
0.21
0.23
0.39
0.68
0.78
0.72
0.94

Bucket Sort


0.78
0.17
0.39
0.26
0.72
0.94
0.21
0.12
0.23
0.68
1
2
3
4
5
6
7
8
9
10
0.12
0.12
0.26
0.23
0.28
0.39
0.68
0.78
0.78
0.94
sort
sort
sort
sort
sort
sort
sort
sort
sort
sort

Bucket Sort Exercise

- Values distributed uniformly over [0,1).
- Each link list is on average small, so average complexity per list is O(1) (regardless of the sorting algorithm used, as long as it is reasonable. E.g., use insertion sort for simplicity).
- Therefore, overall average complexity : O(n)
- Exercises:
  - write the pseudo-code for insertion sort on a singly linked list
  - write the pseudo-code for bucket sort.

Selection problem and order statistics

Selection in Context

- There are a number of applications in which we are interested in identifying a single element of its rank relative to the ordering of the entire set
- Example
  - ☞ The minimum and maximum elements

✓ In general, queries that ask for an element with a given rank are called "order statistics"

Worst Case Time Complexity of Sorting

- mergesort: O(n log$n$), and also requires auxiliary arrays of size n to be created.
- quicksort: O(n^2), but it does not require auxiliary arrays to be created.
- heapsort: O(n log$n$), but it does not require auxiliary arrays to be created.
- Linear time sorting:
  - The worst case complexity of counting sort is O(n+k), but it does require auxiliary arrays to be created. Can counting sort be used in general cases?
  - Radix sort and bucket sort only available for specific cases

Definition

- The *k-th* order statistic of a set of *n* elements is its *k-th* smallest element

Example:

A={11, 10, 14, 4, 2, 1, 5, 7}

✓ If they are sorted {1, 2, 4, 5, 7, 10, 11, 14}

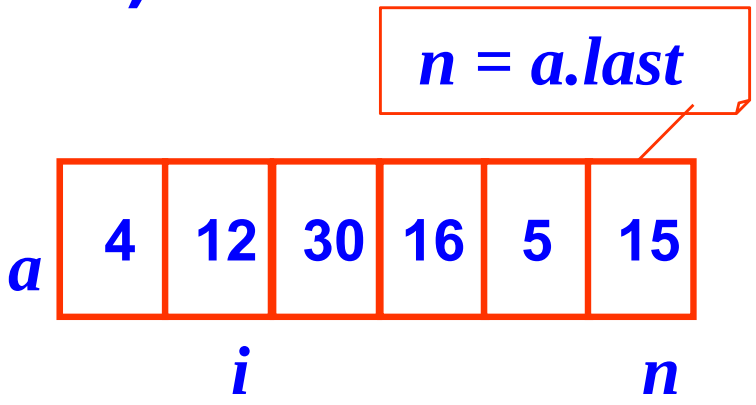
1-st order statistic = minimum(A) = 1

3-rd order statistic = 4

n-th order statistic = maximum(A) = 14

Selecting the minimum and maximum

- Find *1-st* order statistic (minimum)

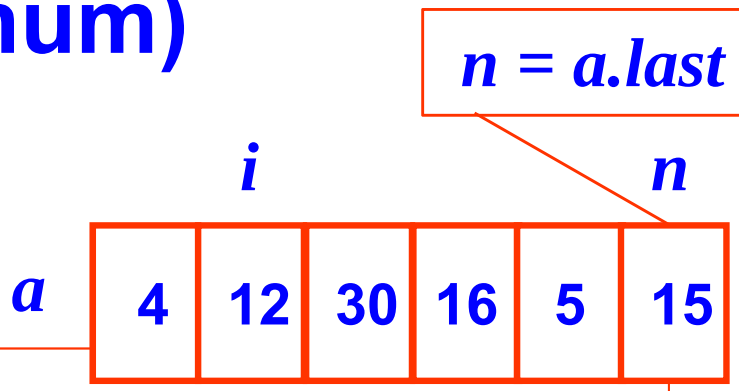


Idea:

✓ *Examine each element of the array in turn and keep track of the smallest element seen so far.*

Selecting the minimum and maximum

- **Find *1-st* order statistic (minimum)**
  - ☞ ***n-1* comparisons**




```
minimum (a) {
   min = a[1]      // start with 1st, assume it smallest
   for i = 2 to a.last
      if (a[i] < min )      // smaller value found
         min = a[i]         // smallest so far
   return min
}
```

✓ *Examine each element of the array in turn and keep track of the smallest element seen so far.*

✓ **Finding the maximum is similar with *n-1* comparison.**

Median

- A median, informally, is the "half way point" of the set if the set is sorted
  - When *n* is odd, the median is unique, occurring at

$$i = (n+1) / 2$$

  - When *n* is even, there are two median, occurring at

$$i = n / 2 \quad \text{and} \quad i = n / 2 + 1$$

✓ Median is the element such that half of the other elements are smaller and the remaining half are larger

Median Example

Consider the incomes listed below:

260,750 25,160 72,815 30,570 137,230 55,300 77,800

What is the median?

If sorted …

4

25,160	30,570	55,300	72,815	77,800	137,230	260,750

The median income is the income that would be in the middle if the incomes were sorted --- the median income is $ 72,815.

The Selection problem

- The problem of selecting the *k-th* order statistic from a set of *n* distinct number
- The selection problem is: Given an array *a* and an integer *k*, find the *k-th* smallest element.

Input: a set *A* of *n* (distinct) numbers and an integer $1 \leq k \leq n$

Output: the element x in S that is larger than (exactly) $k - 1$ other elements of S

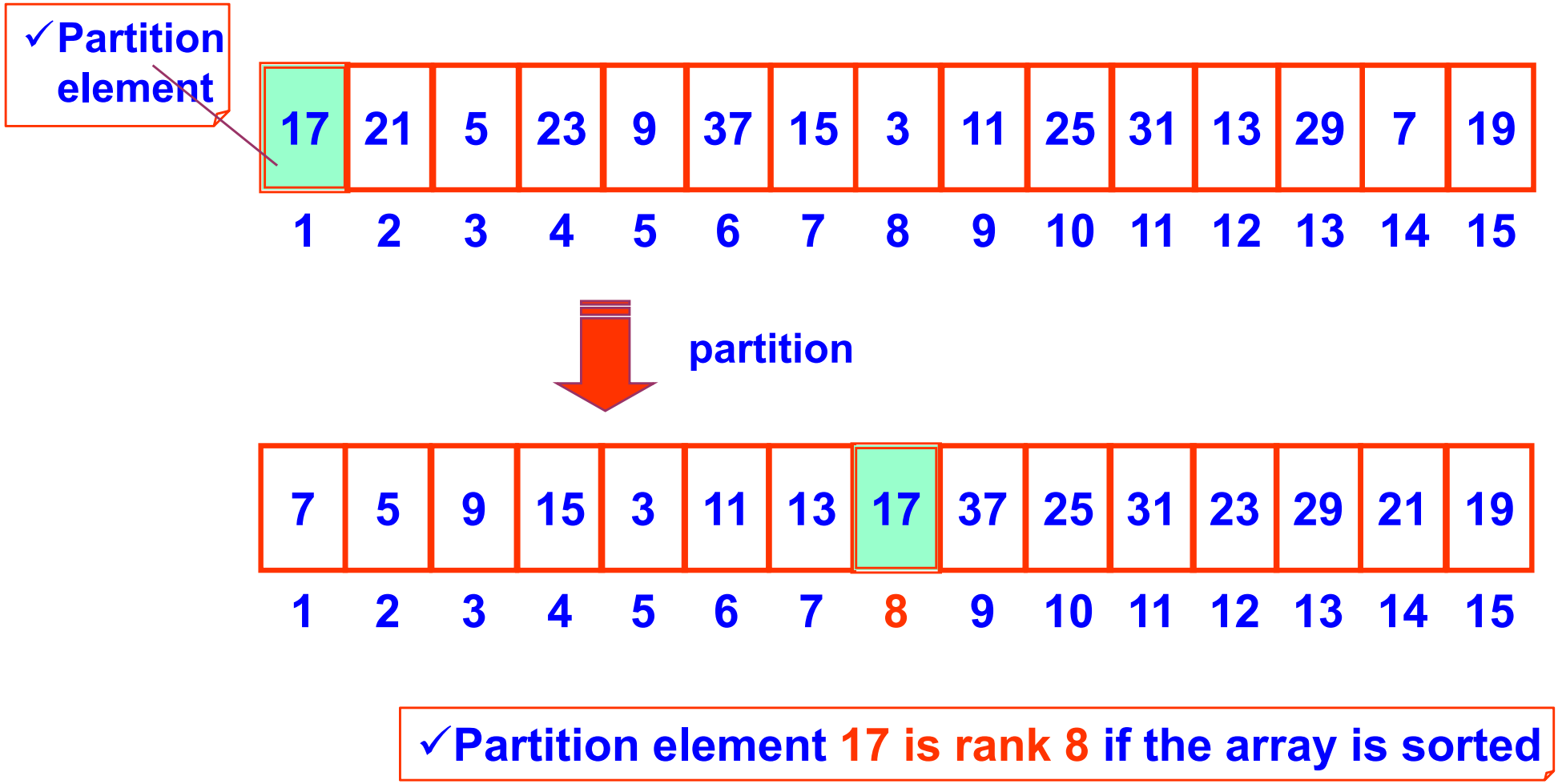
Selection Problem

- Since the selection problem does *not* require that the array be sorted, the goal is to solve the problem faster by doing less work than sorting the entire array.
  - ☞ Using partition

✓ Note that we do not sort the entire array

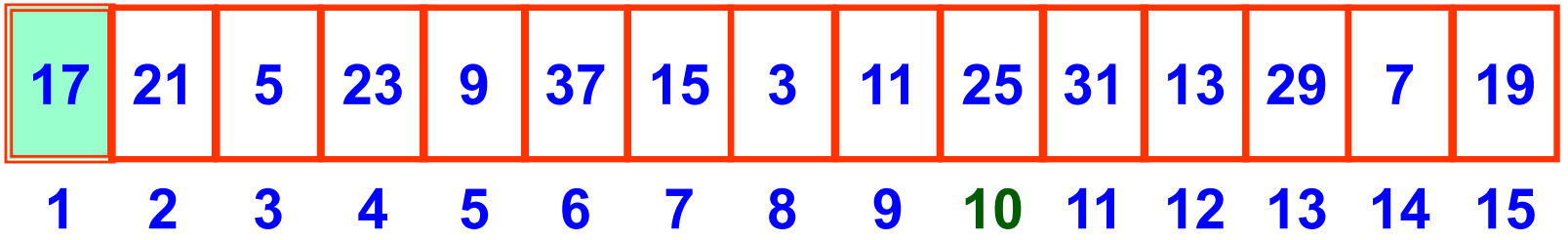
Recall: The property of partition algorithm

- The partition algorithm returns the rank of the partition element.



Selecting by Partition: Example

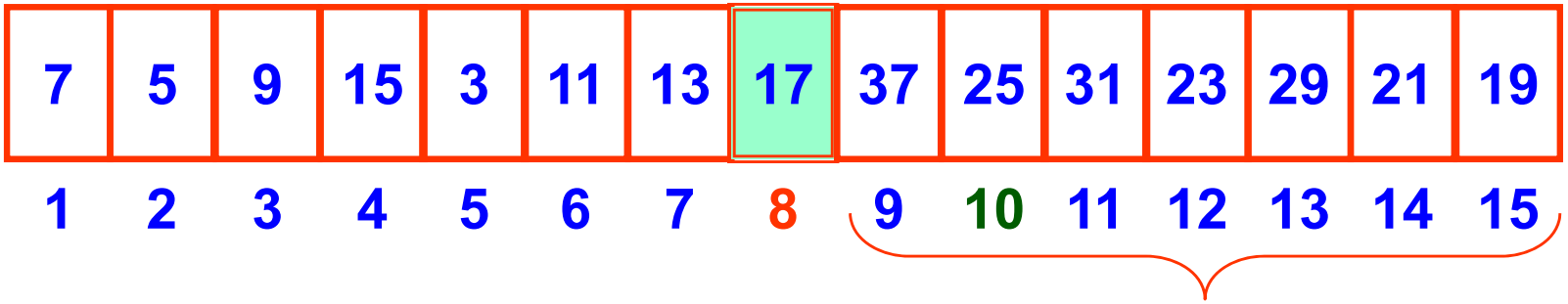
Suppose that we want to find the *10th* smallest element in the array (i.e., rank 10)



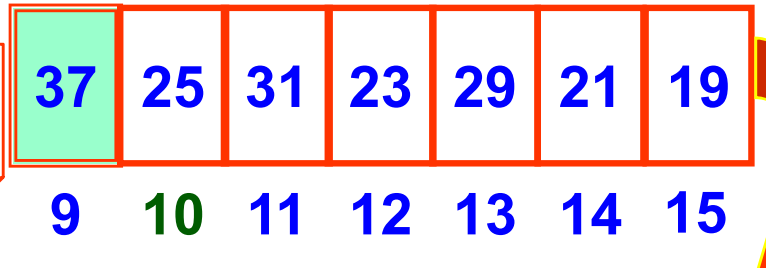
Using 17 as partition element

Partition

After partition, we got rank 8

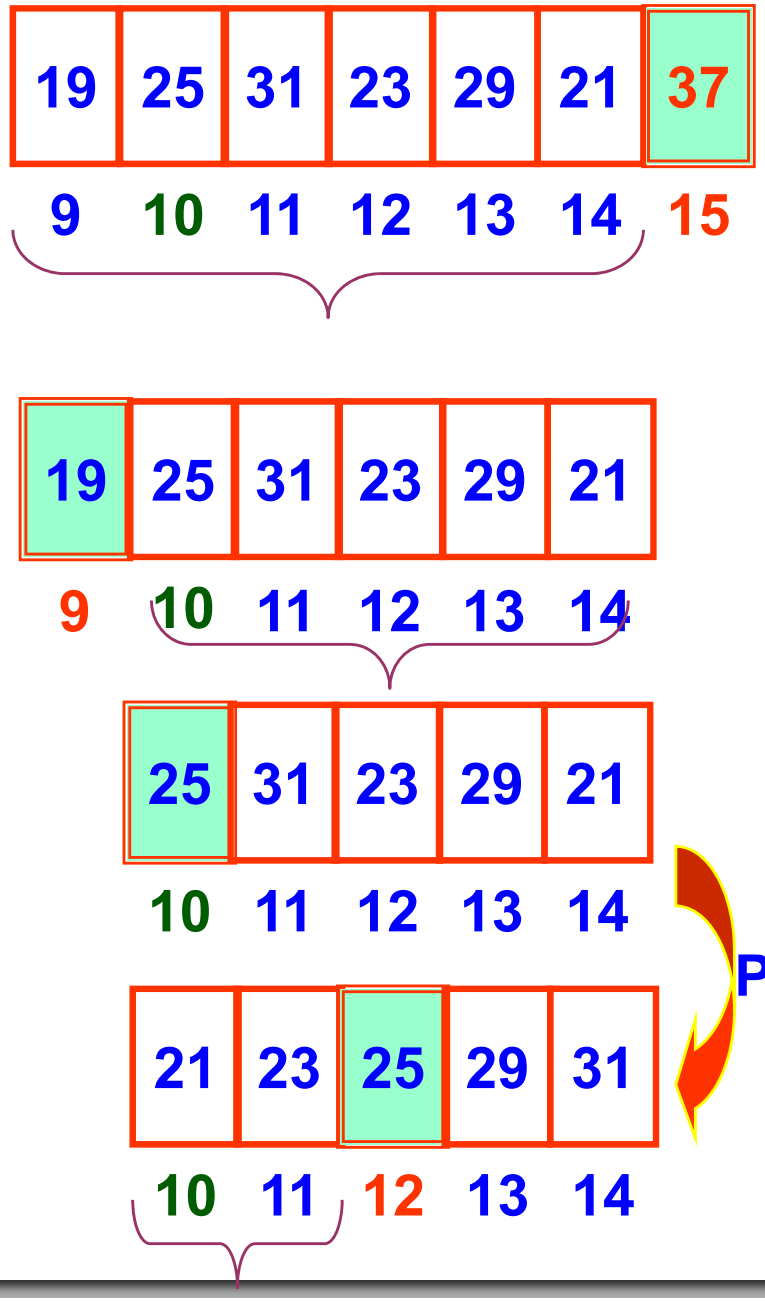


Since 8 < 10, we next call partition on the part of the array to the right of 17



Partition

Selecting by Partition: Example



✓37 is rank 15

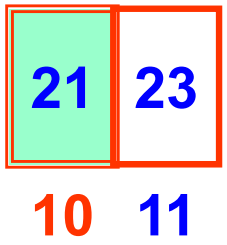
✓19 is rank 9

✓Since rank 9 < 10, we next partition on the part of the array to the right of 19

✓25 is rank 12

Selecting by Partition: Example

Since 10 < 12, we next call partition on the part of the array to the left of 25:



The partition element 21 is placed at index 10. The algorithm thus terminates, having found 21 as the *10th* smallest element in the array

Selecting by Partition: Algorithm

- Translate the idea of selection into an algorithm in pseudo code!

```
selection(A,i,j,k)     {
    if i>j return NULL //not found
    h = partition(A,i,j);
    if k=h return A[h];
    if k<h selection(A,i,h-1,k);
    else selection(A,h+1,j,k);
}
```

Problem Solving Session for

Divide-and-Conquer, Sorting and Selection Related Applications