CSCI 340 Data Structures and Algorithm Analysis

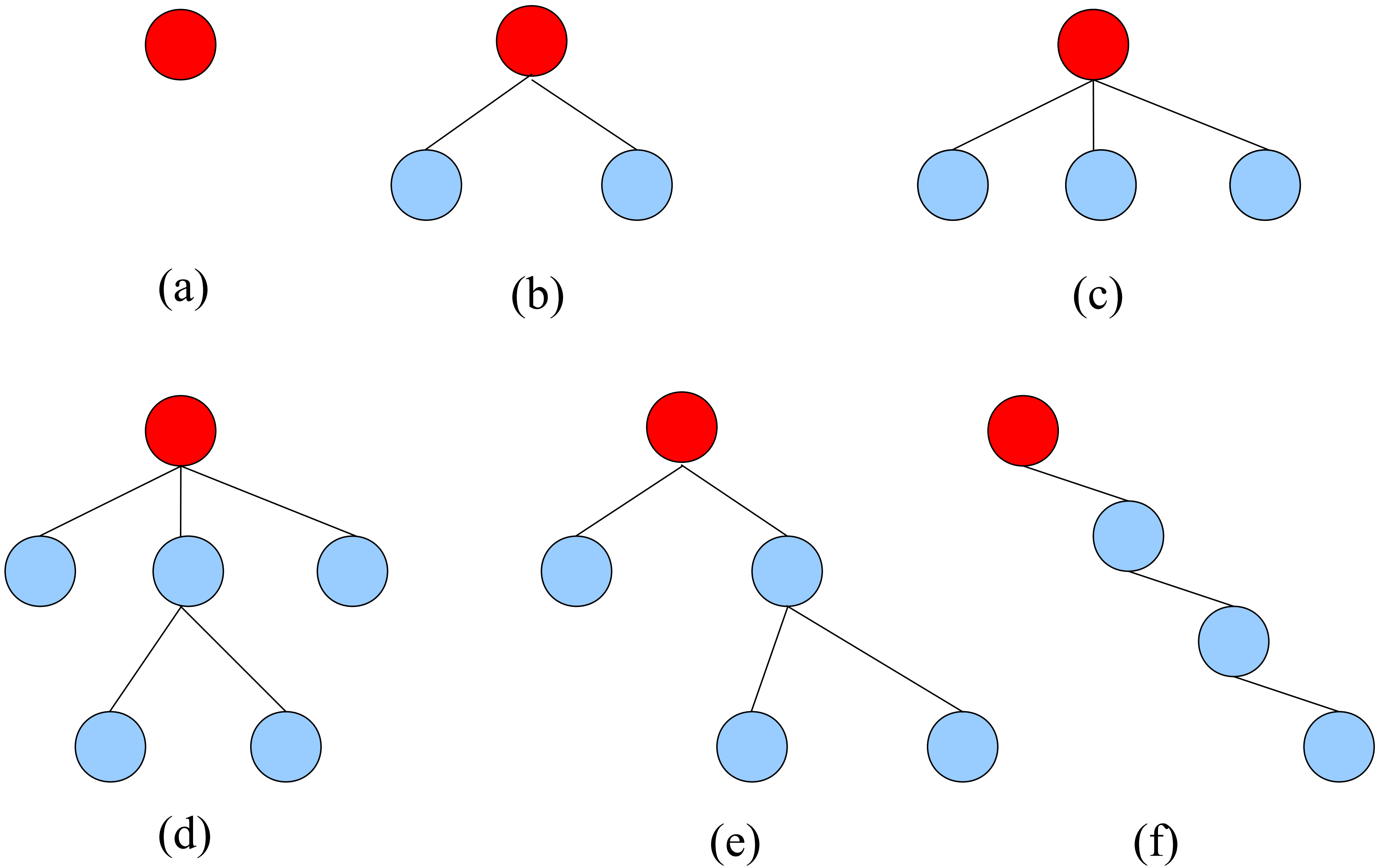
Binary trees (general binary trees, B.S.T.)

Minmei Hou, Ph.D., Dept. Computer Science, NIU

What is tree

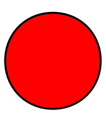
- Tree is a data type that consists of
  - Nodes
    - A node may have child(ren) and a single parent
  - Edges
    - Edges indicate the parent-child relationship
- A tree can be empty and non-empty
  - In case of non-empty tree, it must have one and single one root node which doesn't have a parent
  - The nodes that do not have child(ren) are called leaves.

Example of trees

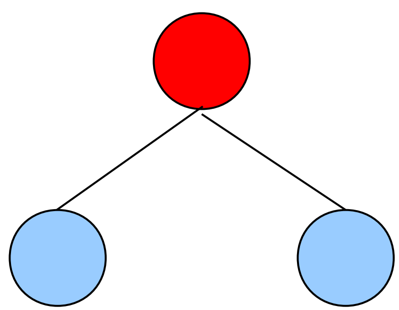


What is binary tree

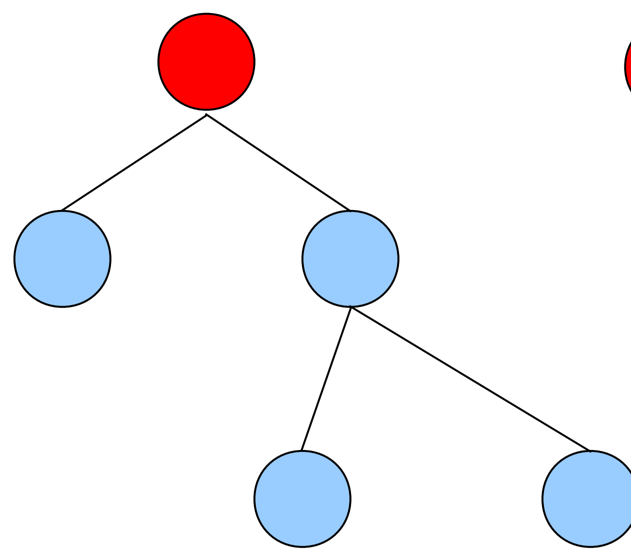
- Binary tree is a tree where any node has at most two children.
  - Left and right child
  - Left and right subtrees
  - Examples:



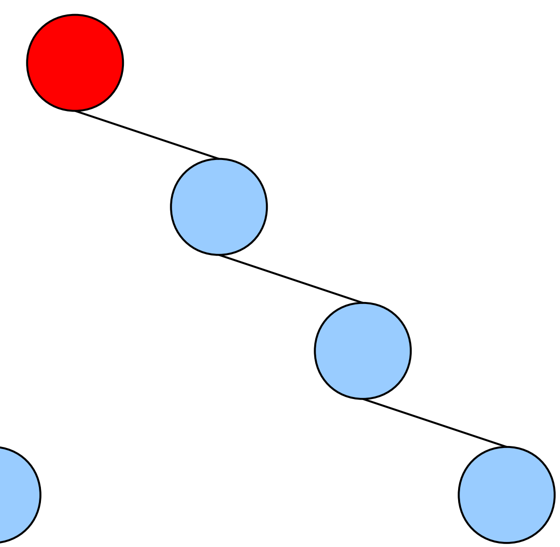
(a)



(b)

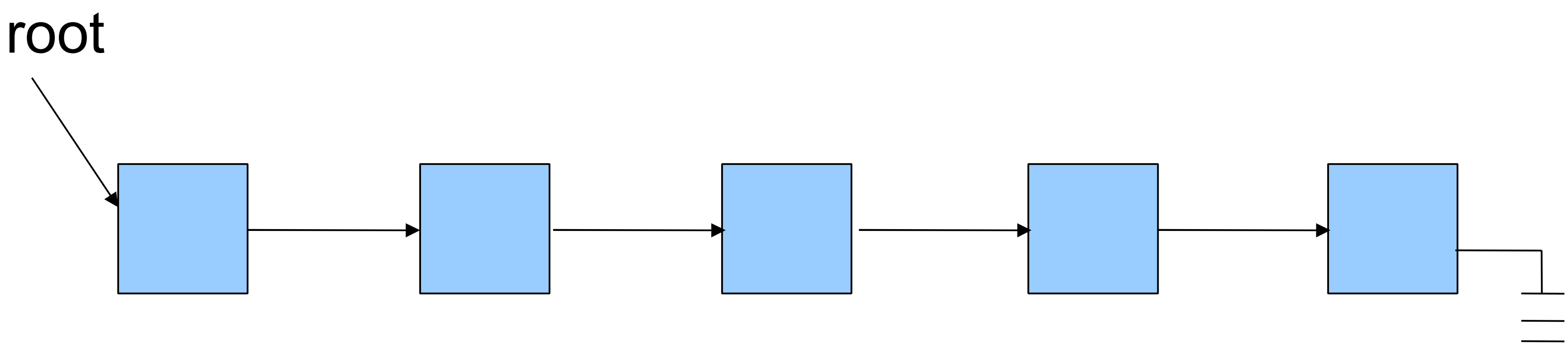


(e)



(f)

Implementing a linked list

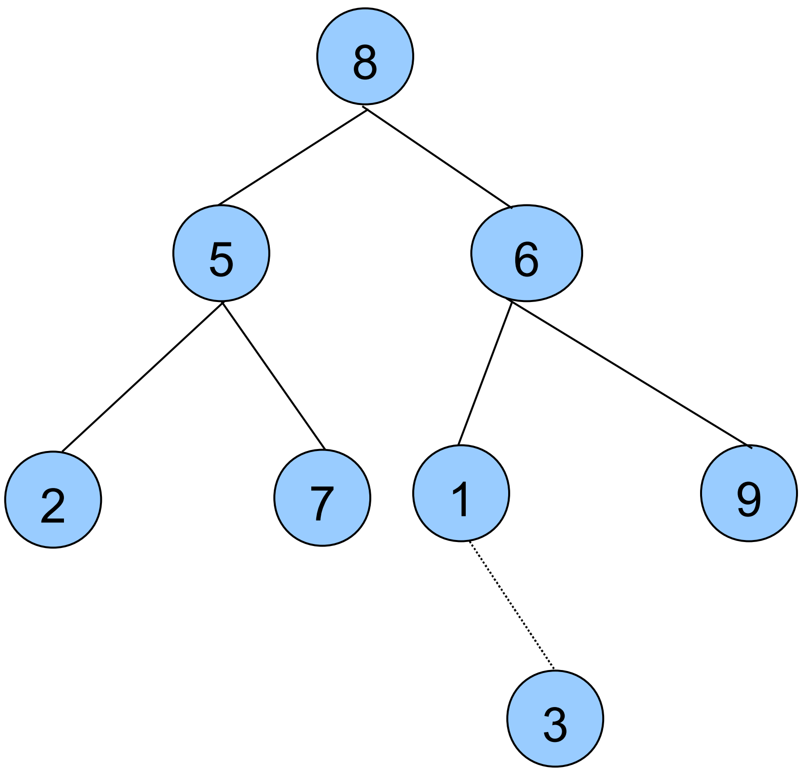


Data fields in Node:
- T value
- Link to the next node

Data fields in the list:
- Root node

Implementing a binary tree

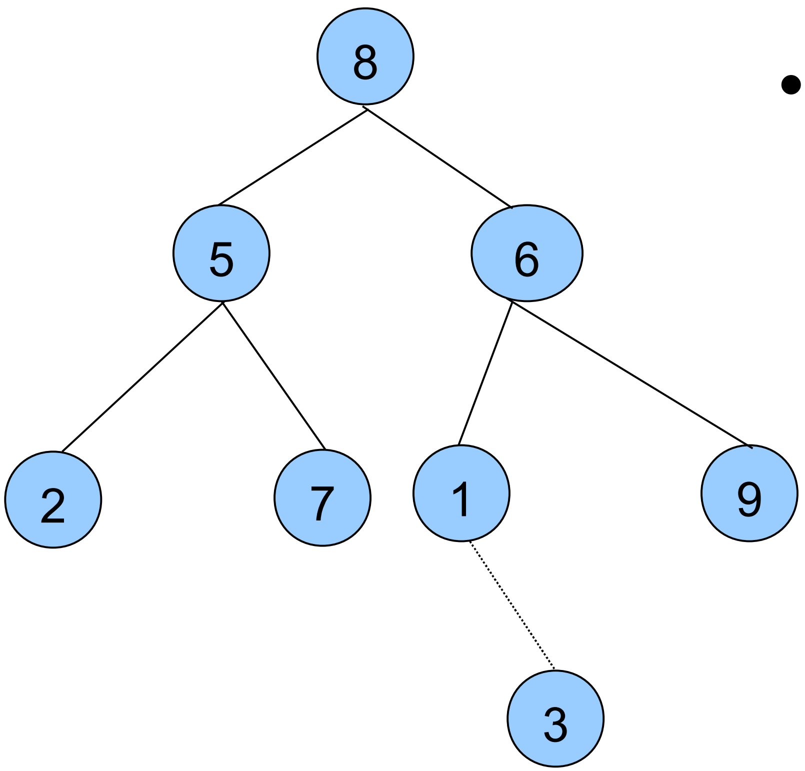
A binary tree



- Minimum data fields of a node:
  - T value
  - Link to the left child
  - Link to the right child
- Minimum data fields of the tree
  - Root node

Implementing a binary tree

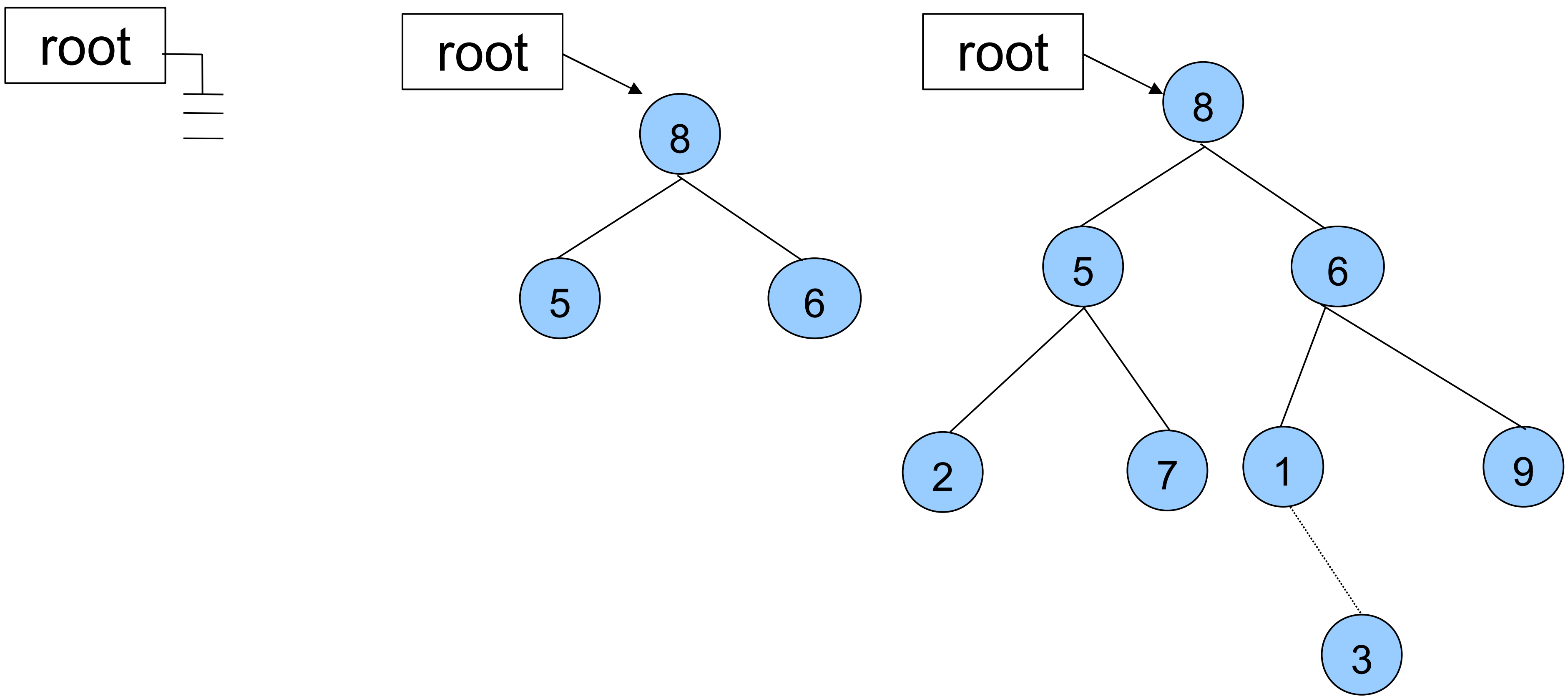
A binary tree



- Operations on the tree
  - IsEmpty
  - Tree traversal
  - Insert an element
  - Delete an element
  - clear
  - (search in B.S.T.)
  - (balancing for height balanced tree)

Implementing a binary tree

bool IsEmpty( )

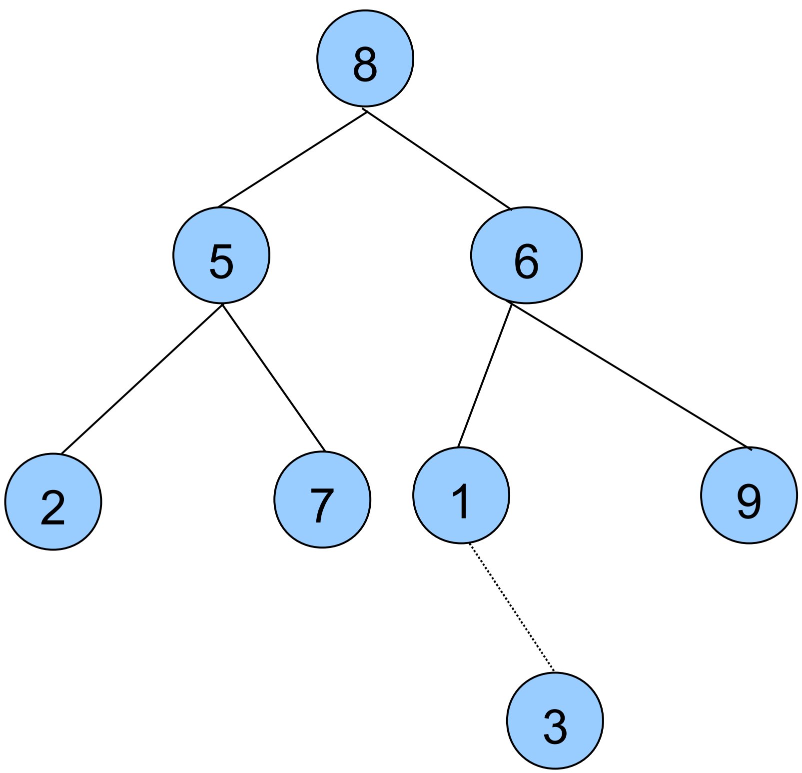


Tree traversal

- Breadth-first traversal
  - Visit each node starting from the highest (or lowest) level and moving down (or up) level by level
  - Visiting nodes on each level from left to right (or right to left).
  - ==> 4 possibilities:
    - Top-down, left-to-right (most common one)
    - Top-down, right-to-left
    - Bottom-up, left-to-right
    - Bottom-up, right-to-left
- Depth-first traversal

Top-down, left-to-right breadth-first traversal implementation

A binary tree



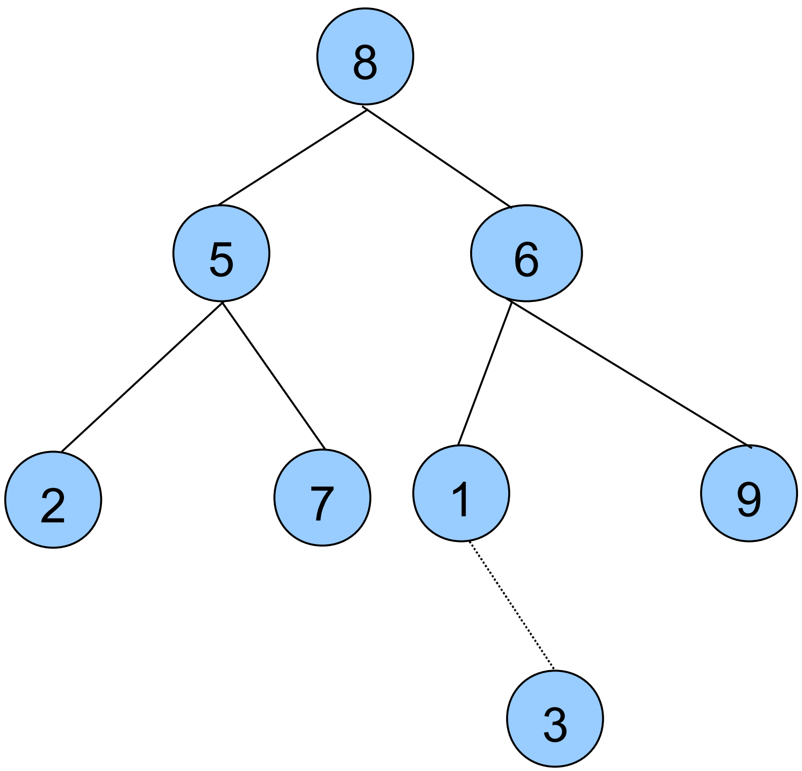
Visit order:
8 5 6 2 7 1 9 3

- Use a queue
- Pseudocode

```
push the root to the queue Q
while ( Q is not empty )
    P ← front & pop of Q
    visit P
    If ( P's left child L is not empty )
        push L to Q
    If ( P's right child R is not empty)
        push R to Q
```

Depth-first traversal

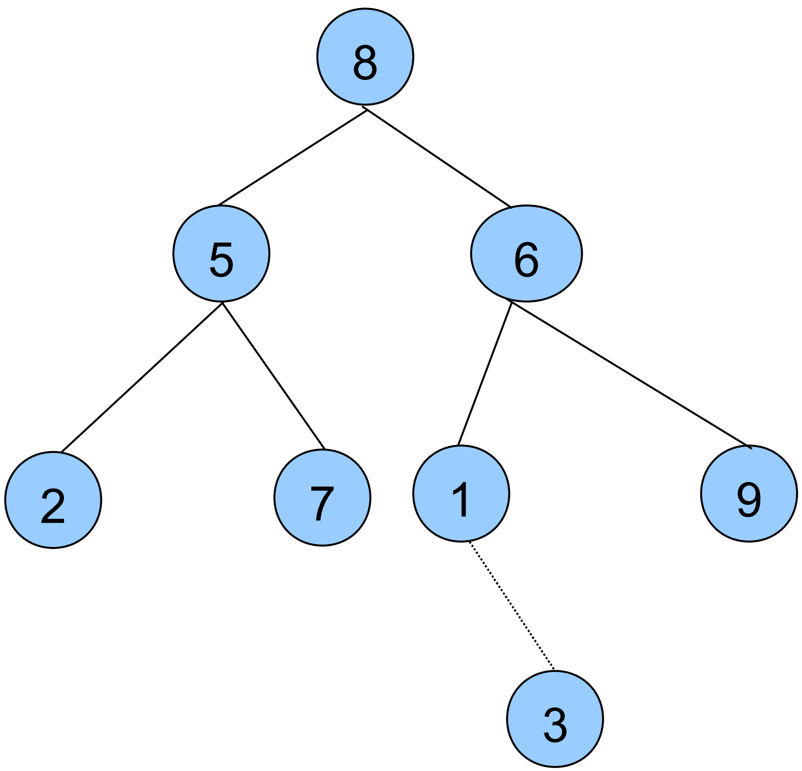
A binary tree



- PreOrder
  - Visit the current node before traversing sub-trees
  - 8 5 2 7 6 1 3 9
- InOrder
  - Traverse left sub-tree first, then visit the current node, then traverse right sub-tree
  - 2 5 7 8 1 3 6 9
- PostOrder
  - Visit the current node after traversing sub-trees
  - 2 7 5 3 1 9 6 8

Pseudocode of recursive implementation of depth-first binary tree traversal

A binary tree



InOrder (a node N)
    if N is not empty
        InOrder(N's left child)
        visit N
        InOrder(N's right child)

PreOrder (a node N)
    If N is not empty
        visit N
        PreOrder (N's left child)
        PreOrder (N's right child)

PostOrder (a node N)

- If N is not empty
  - PostOrder (N's left child)
  - PostOrder (N's right child)

  Visit N

Minmei Hou, Ph.D., Dept. Computer Science, NIU

Implementing tree traversal

- public member function of the tree class:

```
void PreOrder( ) { PreOrder(root); }
```

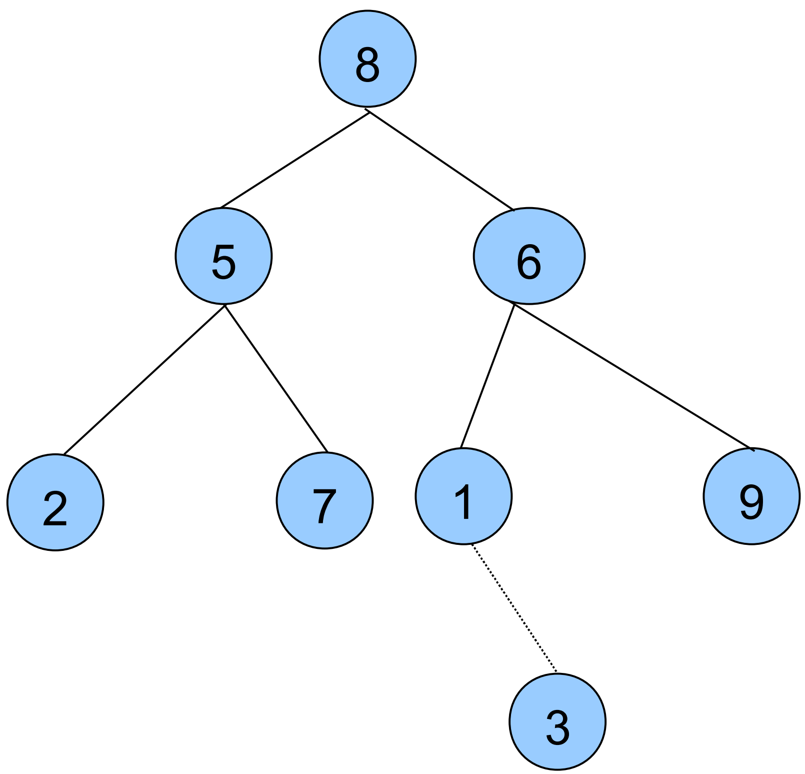
- private member function of the tree class:

```
void PreOrder(Node* n);
```

Nonrecursive implementation of PreOrder binary tree traversal

- Use a stack *S*
- Pseudocode

A binary tree

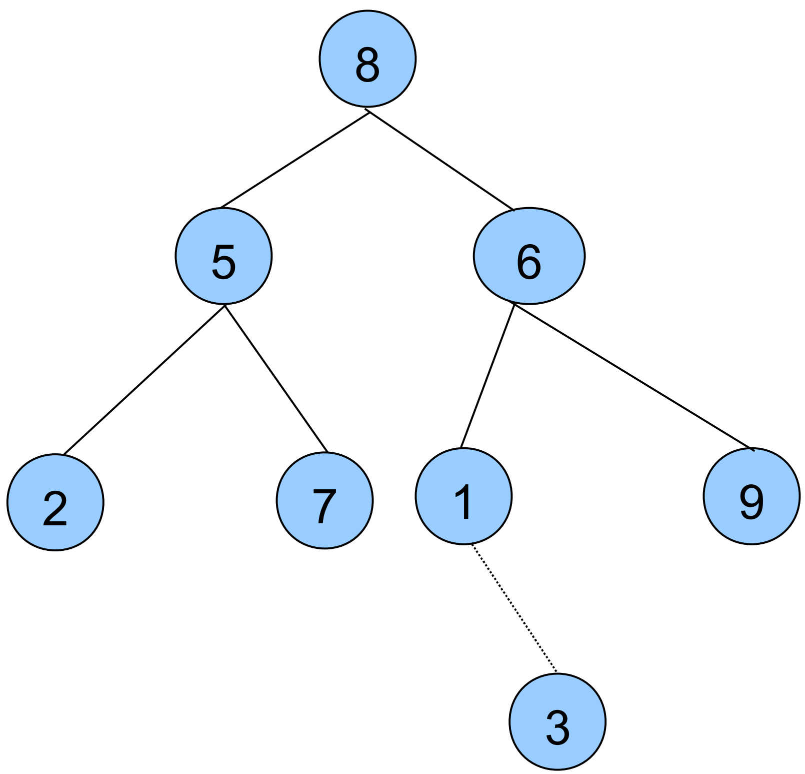


$p \leftarrow$ root of the tree
if (p is not empty)
  push p to S
  while (S is not empty)
    $p \leftarrow$ top and pop S
    visit p
    if (p's right child R is not empty)
      push R to S
    if (p's left child L is not empty)
      push L to S

Nonrecursive implementation of PostOrder binary tree traversal

- Use a stack S
- Pseudocode

A binary tree

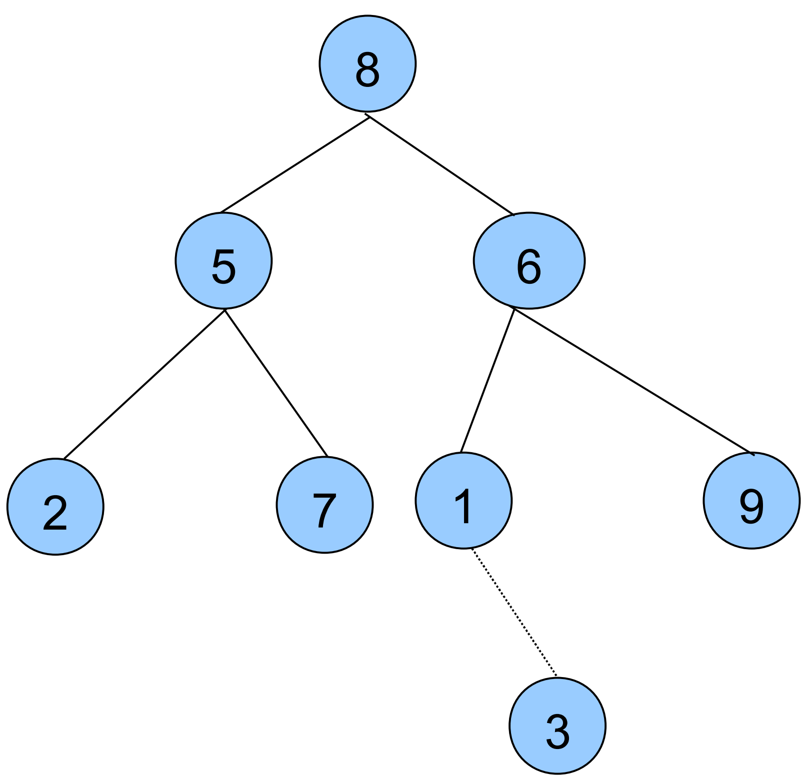



```
p ← q ← root of the tree
while ( p is not empty )
    while ( p's left child L is not empty)
        push p to S
        p ← L
    while ( p is not empty AND
        (p's right child is empty OR
         p's right child is q) )
        visit p
        q ← p
        If ( S is empty )
            stop
        p ← top and pop S
    push p to S
    p ← p's right child
```

Nonrecursive implementation of InOrder binary tree traversal

- Use a stack *S*
- Pseudocode

A binary tree




```
p ← root of the tree
while ( p is not empty )
    while ( p is not empty)
        if ( p's right child R is not empty)
            push R to S
        push p to S
        p ← p's left child
    p ← top & pop S
    while ( S is not empty AND p's
                right chid R is empty)
        visit p
        P ← top & pop S
    visit p
    if ( S is not empty )
        p ← top & pop S
    else
        p ← null
```

Minmei Hou, Ph.D., Dept. Computer Science, NIU

Note

For depth-first tree traversal, only recursive implementations are required (for your exam).

Nonrecursive implementations of depth-first tree traversal are for your reference only.

Application of tree traversals

- Find the number of nodes in the tree
- Find the height of the tree
- Find the number of leaves in the tree
- Make a copy of the tree

Clear a tree

- Use a recursion in private member method:

```
ClearTree ( R )  // R is the root at the first call
    if R is not empty
        ClearTree R's left child
        ClearTree R's right child
        Deallocate R
```

- A public member method simply invoke:

```
ClearTree( root ) // root is the data member
root ← empty      // reset root pointer
```

Make a clone

- Deep copy vs. shallow copy
- Recursive private method:

```
Clone (Curr)
    If Curr is not empty
        L ← Clone Curr's left child
        R ← Clone Curr's right child
        N ← Create new node with Curr's value, L and R
        N is clone of Curr
    Else
        Clone of Curr is empty
```

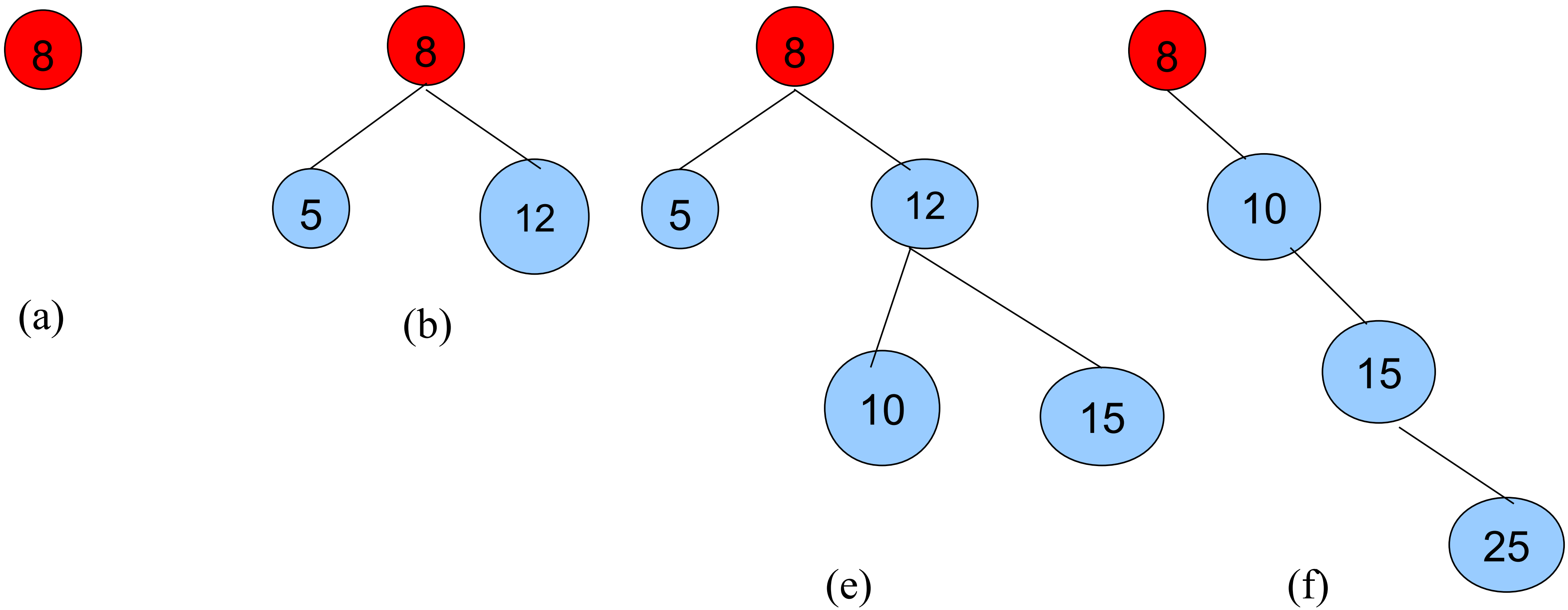
- Public method simply invoke:

```
Clone( src.root ) // where src is source tree object
```

What is binary search tree

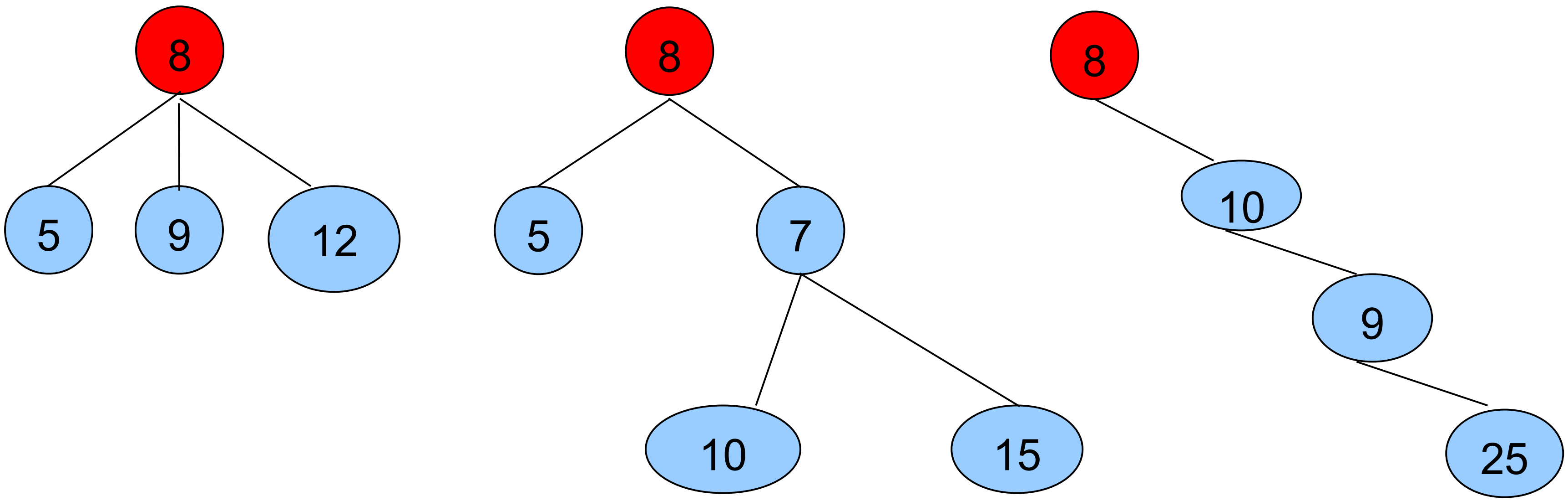
- First of all, it's a binary tree
- Assuming there is no redundant values, it satisfies the following property:
  - For any node *n* of the tree, all values stored in *n*'s left sub-tree are less that the value stored in *n*,
  - And all values stored in the right sub-tree are greater than the value stored in *n*.

Examples of B.S.T.



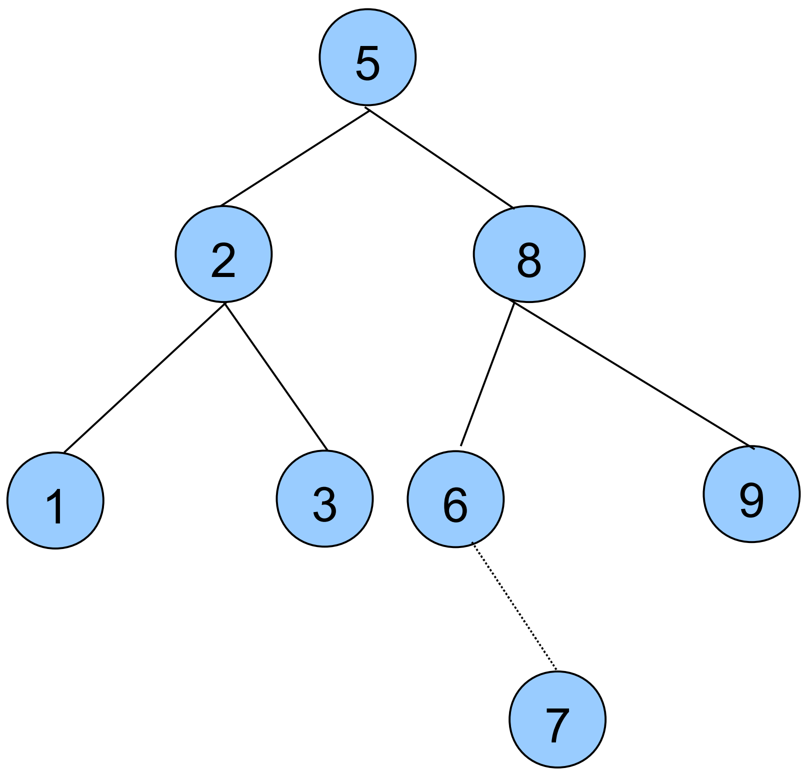
Minmei Hou, Ph.D., Computer Science Department, NIU

These are not B.S.T.



Depth-first traversals of B.S.T.

A B.S.T.



- PreOrder

5 2 1 3 8 6 7 9

- InOrder

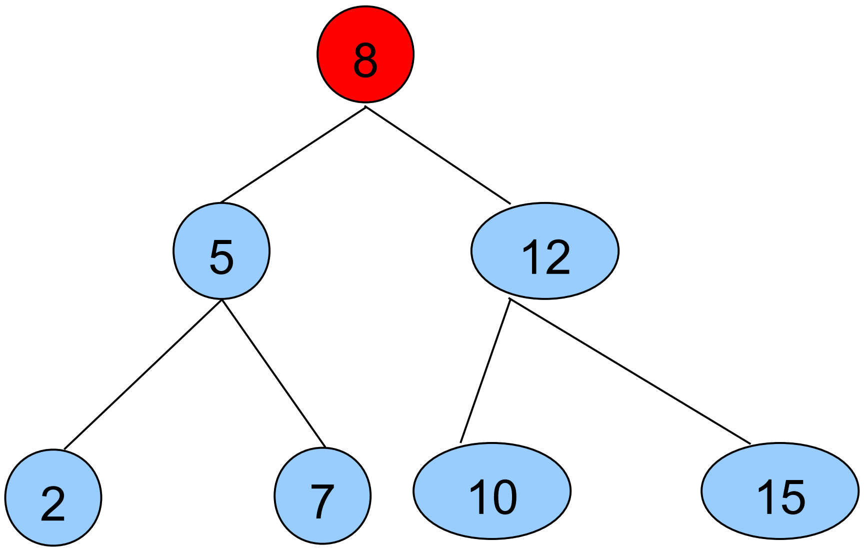
1 2 3 5 6 7 8 9

- PostOrder

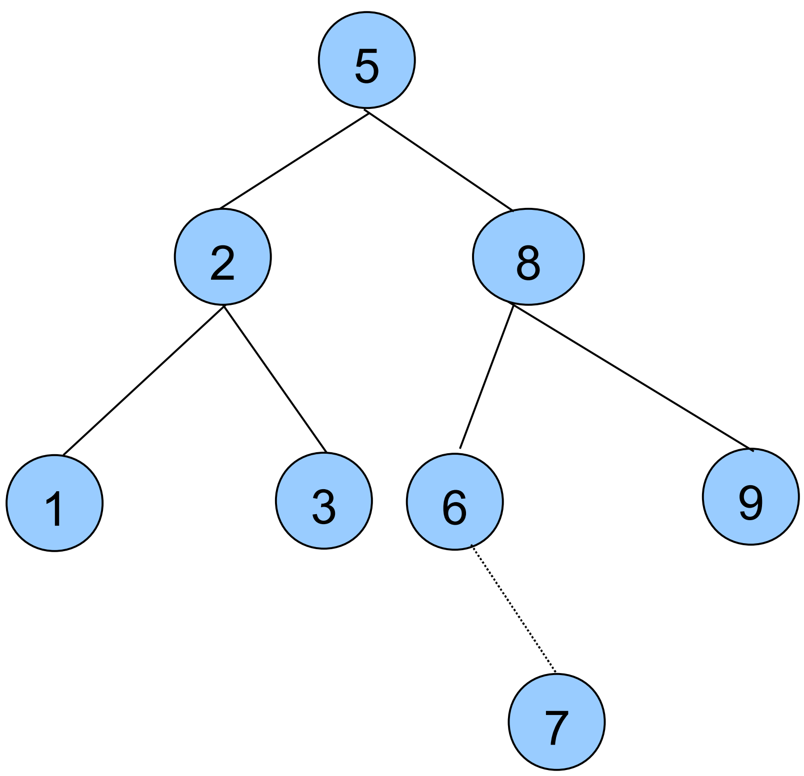
1 3 2 7 6 9 8 5

Search in a B.S.T.

- A search in B.S.T. starts from the root
- Follows a path from the root to a leaf
- It may stop in the middle of the path in case the element is found
- The length of the path determines the efficiency of the search operation

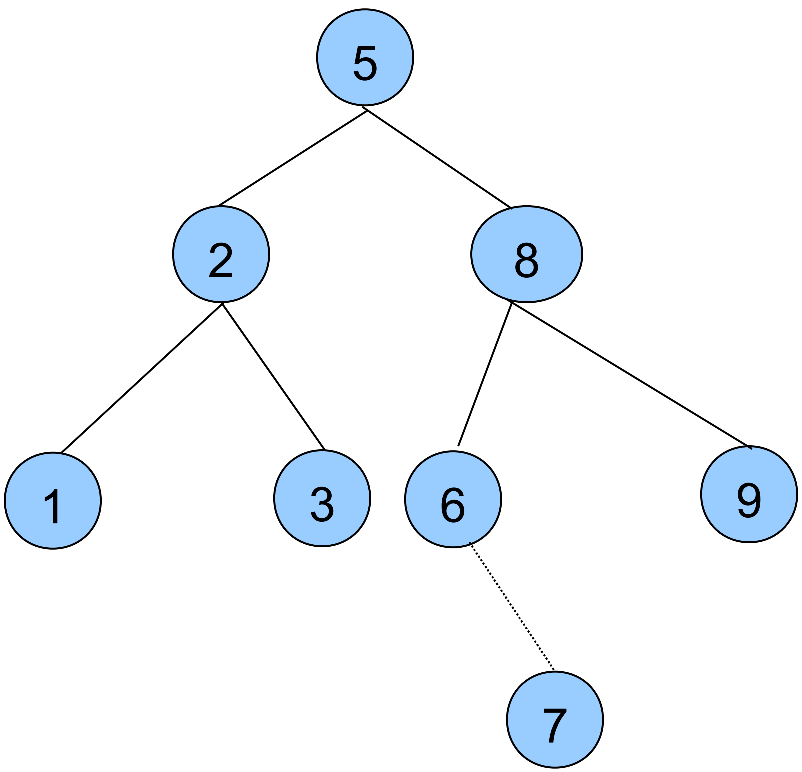


Implementing B.S.T. Search



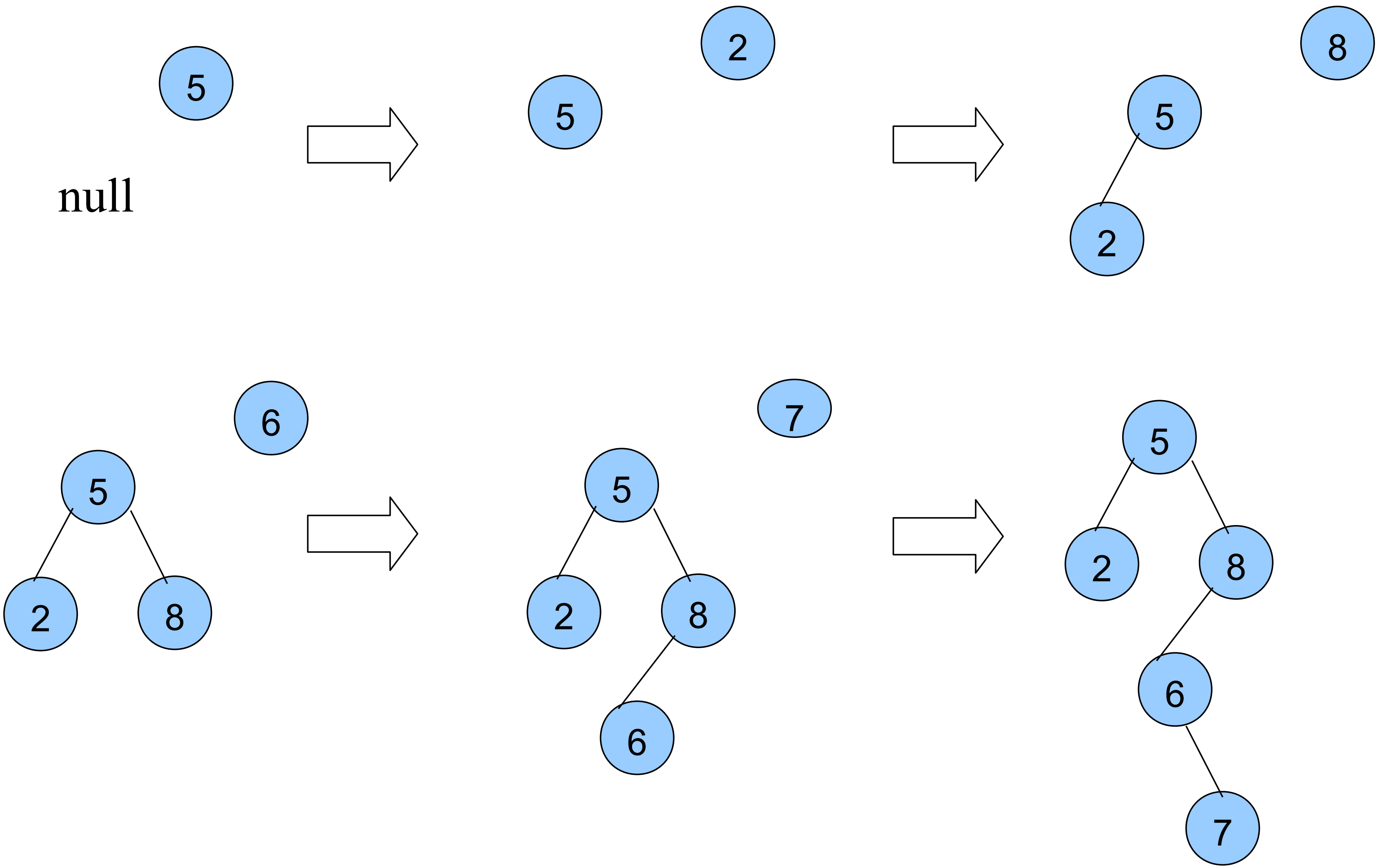
BSTSearch(N, val)
    If N is not empty
        If N's value is val
            Val is found
        If N's value > val
            Search in N's left sub-tree
        Else
            Search in N's right sub-tree
    Else
        Val is not found in the tree

Implementing B.S.T. search



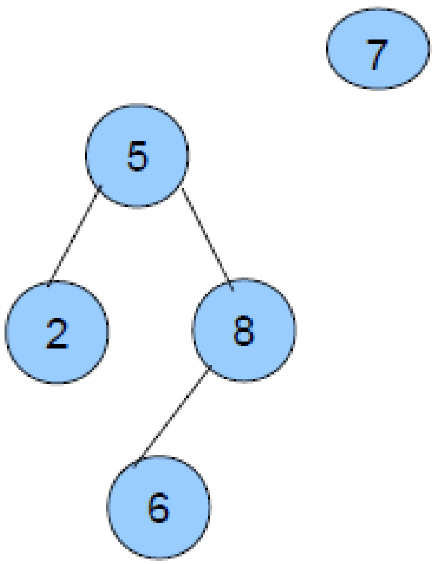

```
BSTSearch( N, val)
    while N is not empty
        if N's value == val
            Val is found
        else if N's value > val
            N ← N's left link
        else
            N ← N's right link
    if N is empty
        Val is not found
```

Insertion in B.S.T.



Minmei Hou, Ph.D., Dept. Computer Science, NIU

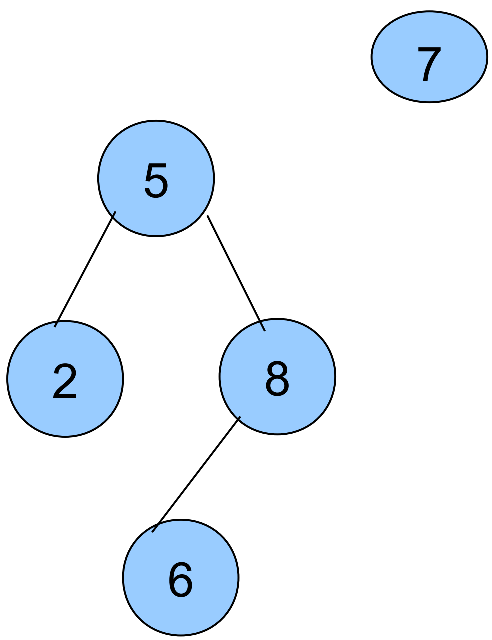
Insertion in B.S.T.



- Recursive insertion

BSTInsert(n, val)
If n is null
    n ← new node with val
Else if n's data > val
    BSTInsert(n->left, val)
Else
    BSTInsert(n->right, val)

Non-recursive Insertion in B.S.T.



BSTInsert(*val*)
    N ← create node with *val*
    if (*root* is empty)
        root ← *N*
        STOP
    curr ← *root*
    p*rev* ← null
    while (*curr* is not empty)
        p*rev* ← *curr*
        if *curr*'s value is less than *val*
            curr ← *curr*'s right link
        else
            curr ← *curr*'s left link
    if prev's value is less than *val*
        prev's right link ← *N*
    else
        prev's left link ← *N*

Deletion in B.S.T.

Three cases:

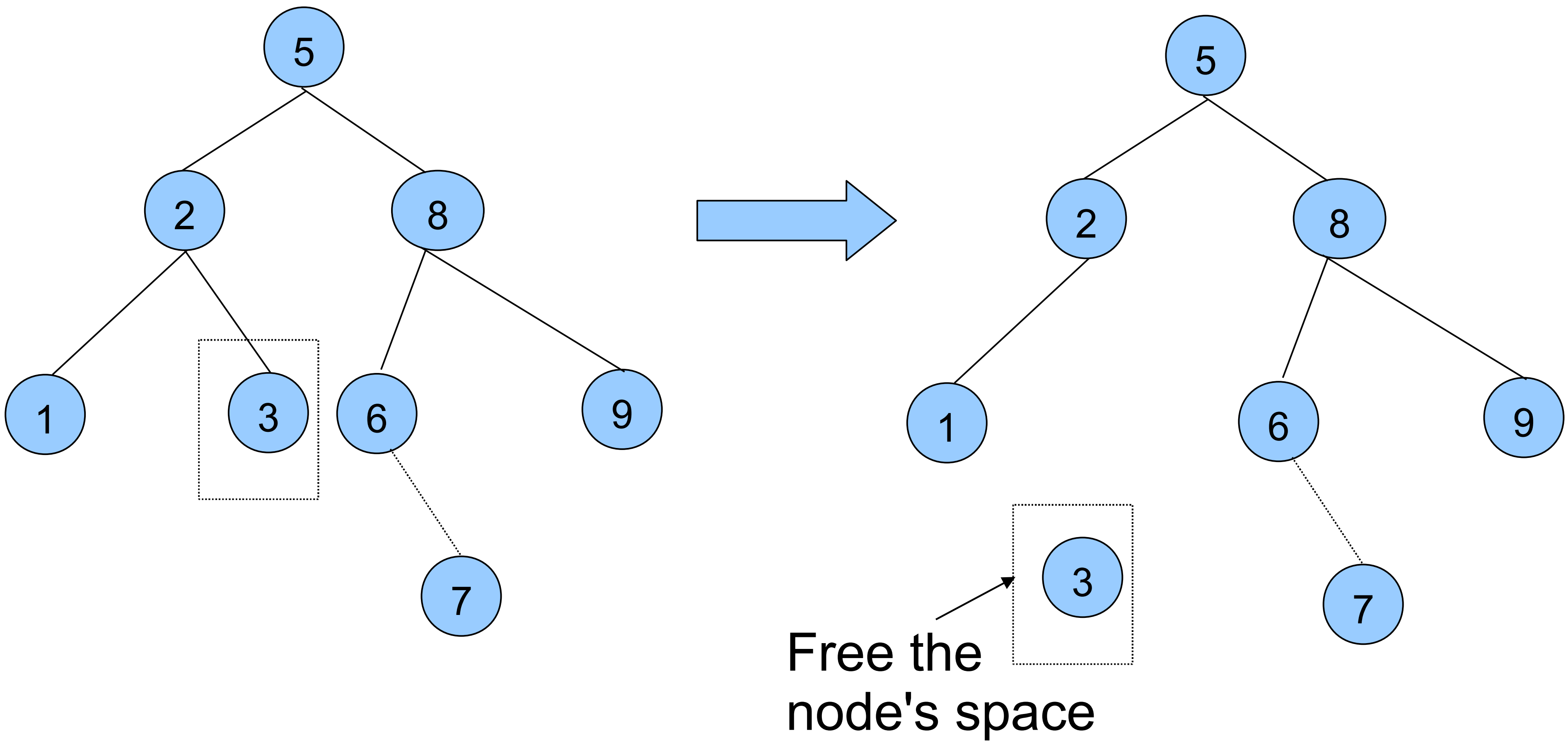
1. the node to be deleted is a leaf

2. the node to be deleted has one child/subtree

3. the node to be deleted has two children/subtrees

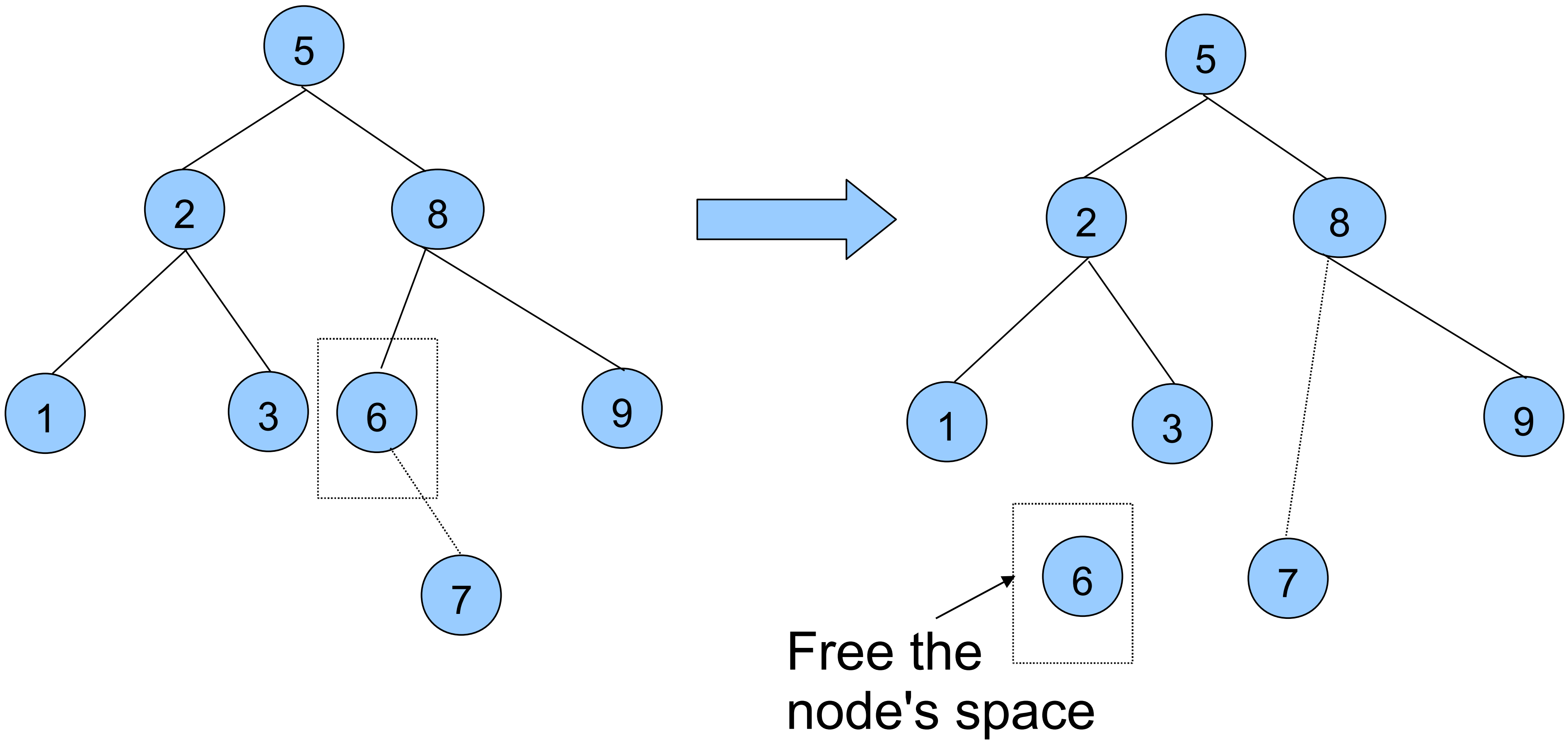
Deletion in B.S.T.

1st case: Deleting a leaf.



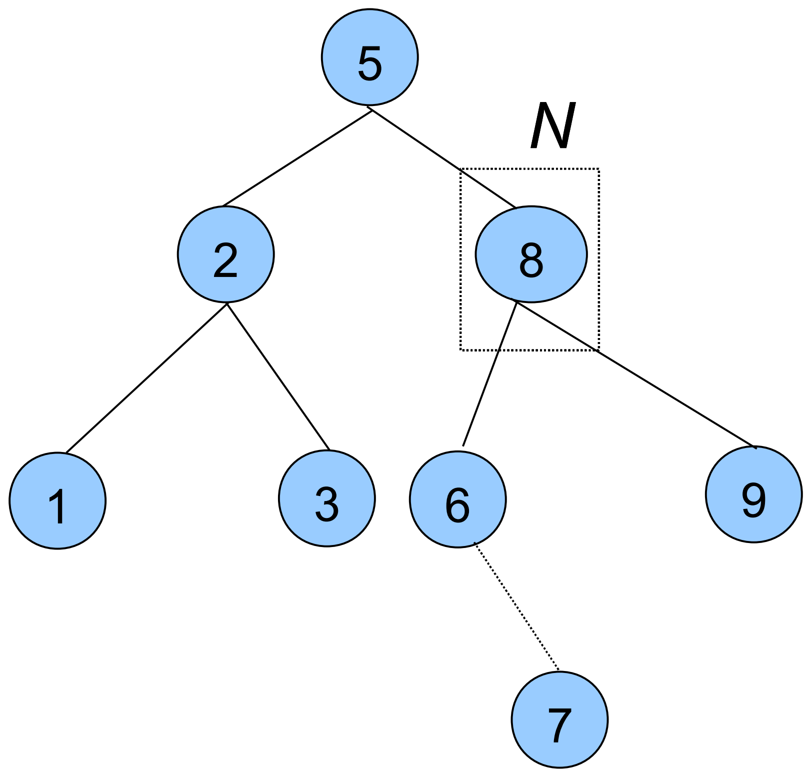
Deletion in B.S.T.

2nd case: Deleting a node with one child/subtree.



Deletion in B.S.T.

3^{rd} case: Deleting a node *N* with two children/subtrees.



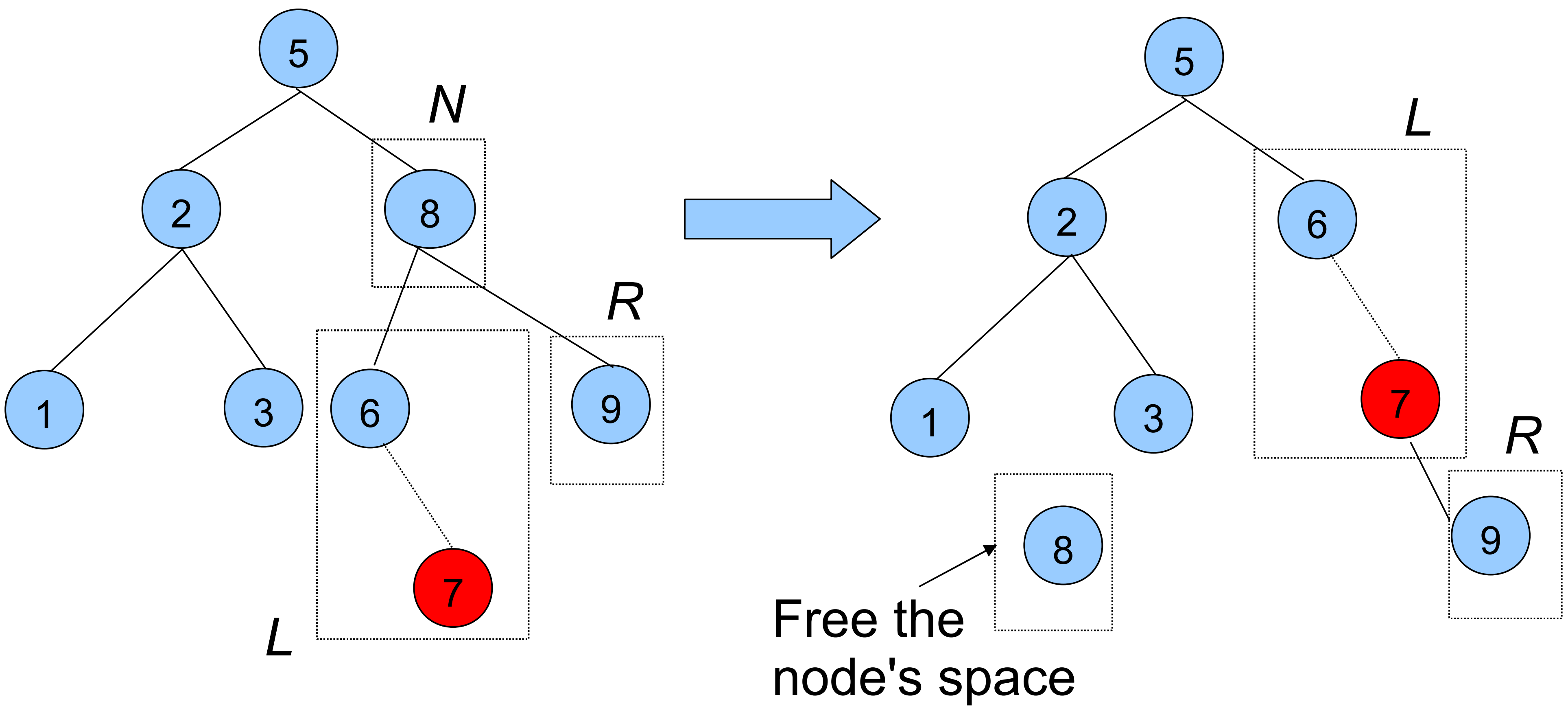
Two approaches:

- Deletion by merging
- Deletion by copying

Deletion in B.S.T.

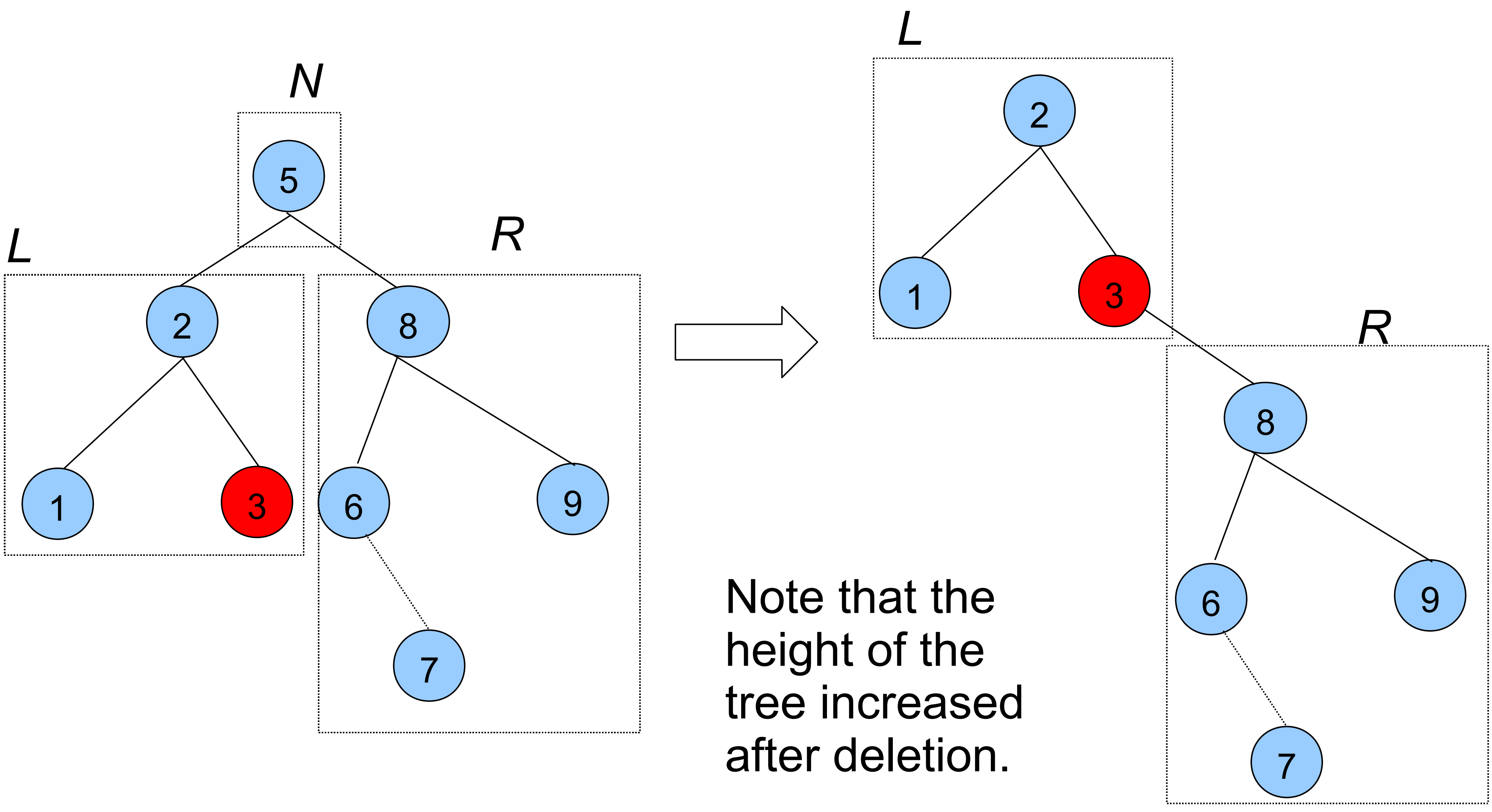
Approach I: deletion by merging.

- *N*: the node to be deleted; *L*: *N*'s left subtree; *R*: *N*'s right subtree
- *R* is attached to *L*'s right-most node (*N*'s immediate predecessor) as the right child
- *N* is replaced by its left child



Deletion in B.S.T.

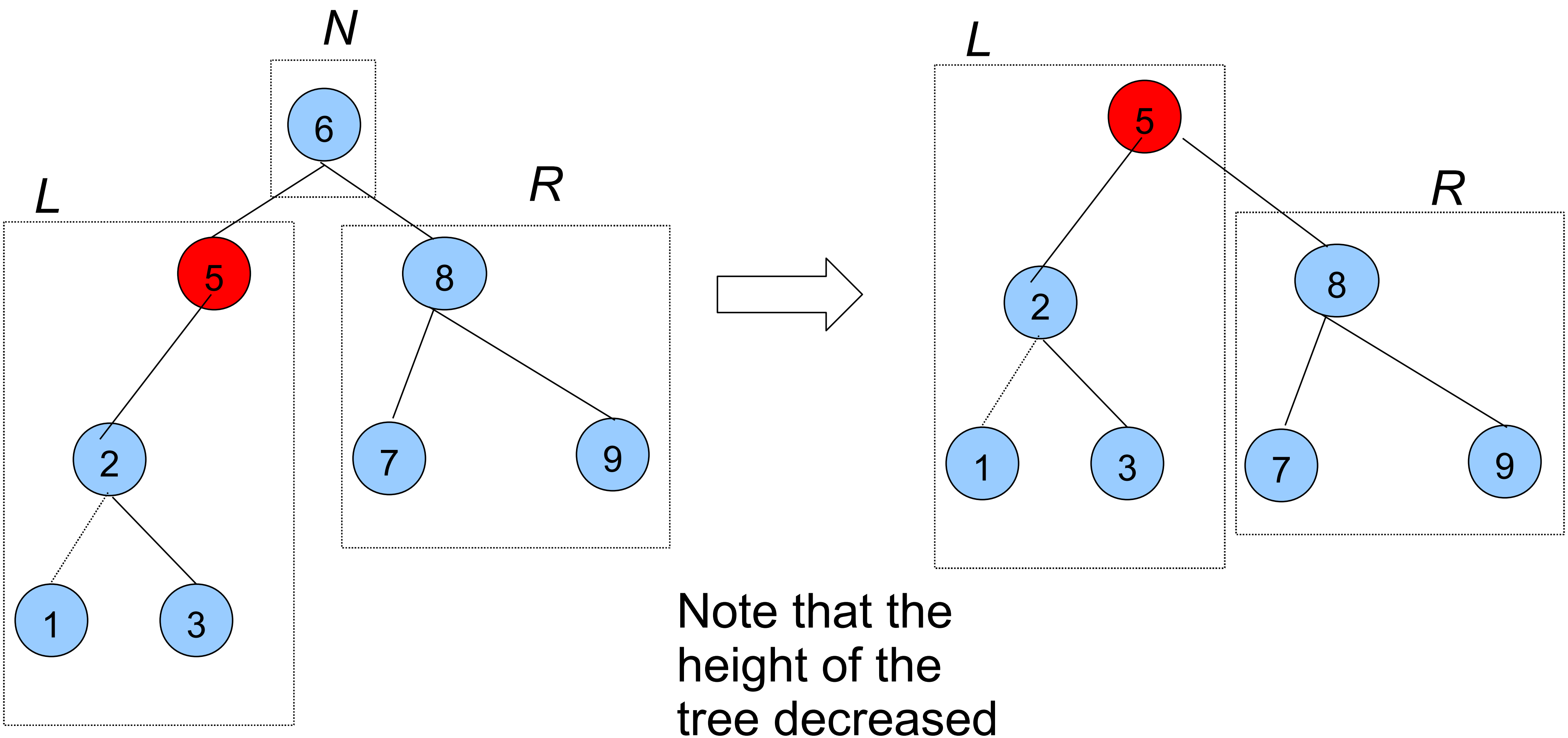
More examples of deletion by merging:



Note that the height of the tree increased after deletion.

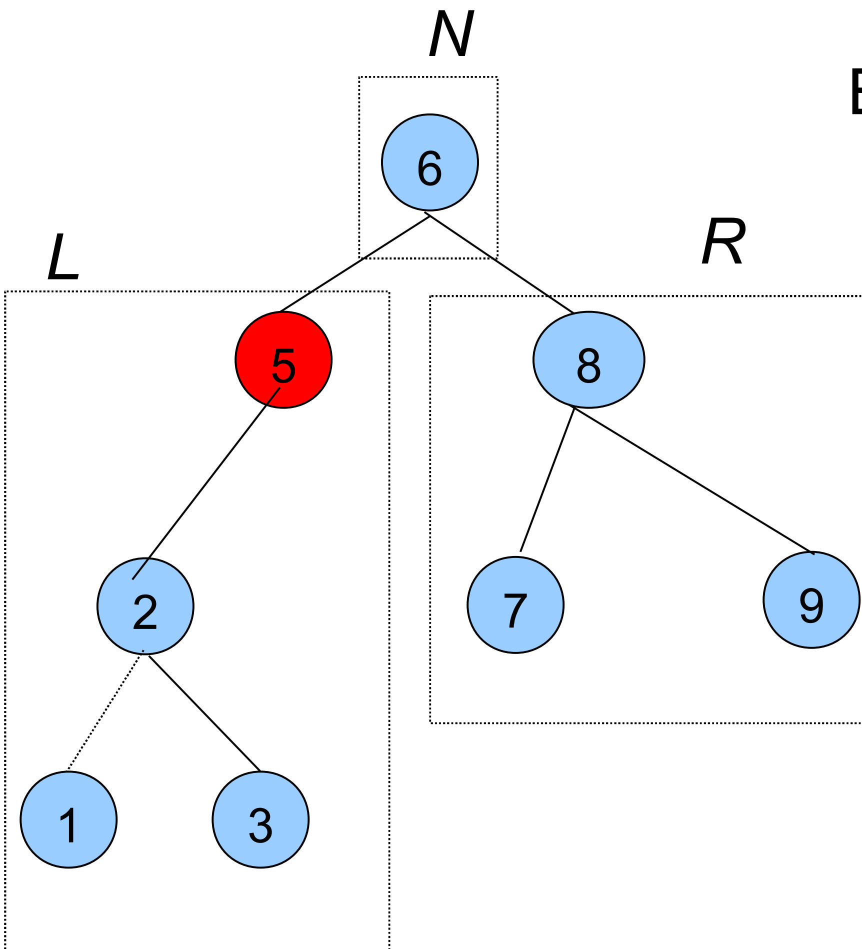
Deletion in B.S.T.

More examples of deletion by merging:



Note that the height of the tree decreased after deletion.

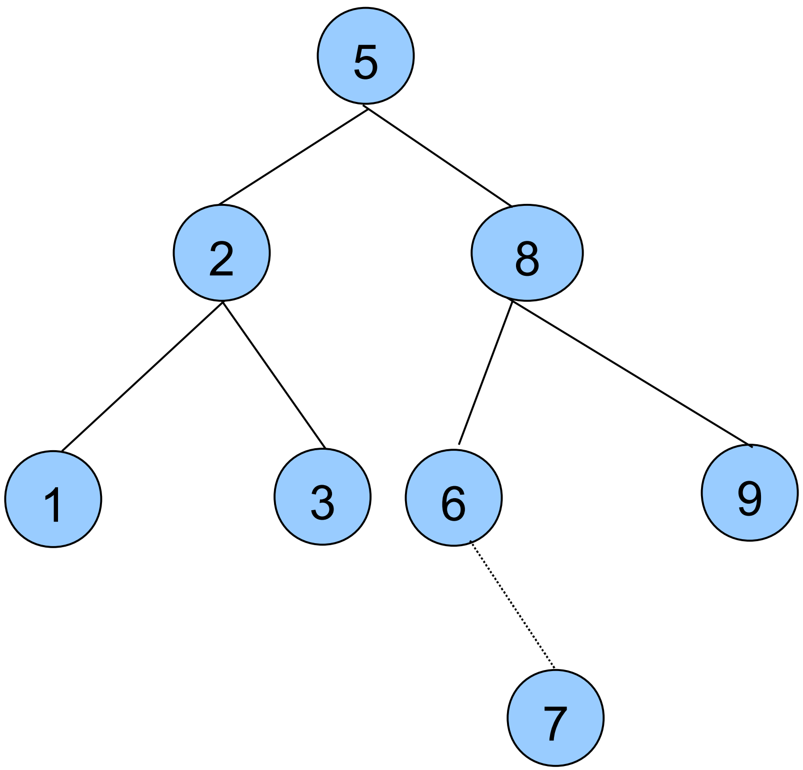
Pseudocode of deletion by merging



BSTDeletionByMerging (*N*)
    // N is passed by “passing-by-reference”
    if *N* is empty
        STOP
    tmp ← *N* // for deallocation later
    if *N*'s right child is empty
        N ← *N*'s left child
    else if *N*'s left child is empty
        N ← *N*'s right child
    else // in case *N* has two children
        pred ← immediate predecessor of *N*
        pred's right child ← *N*'s right child
        N ← *N*'s left child
    deallocate *tmp*

Minmei Hou, Ph.D., Dept. Computer Science, NIU

Pseudocode of deletion by merging



How to find **immediate predecessor** of node *N* in case *N* has two children?

> $p \leftarrow N$'s left child
> **while** p's right child is not empty
> $p \leftarrow p$'s right child
> p is the predecessor of node *N*

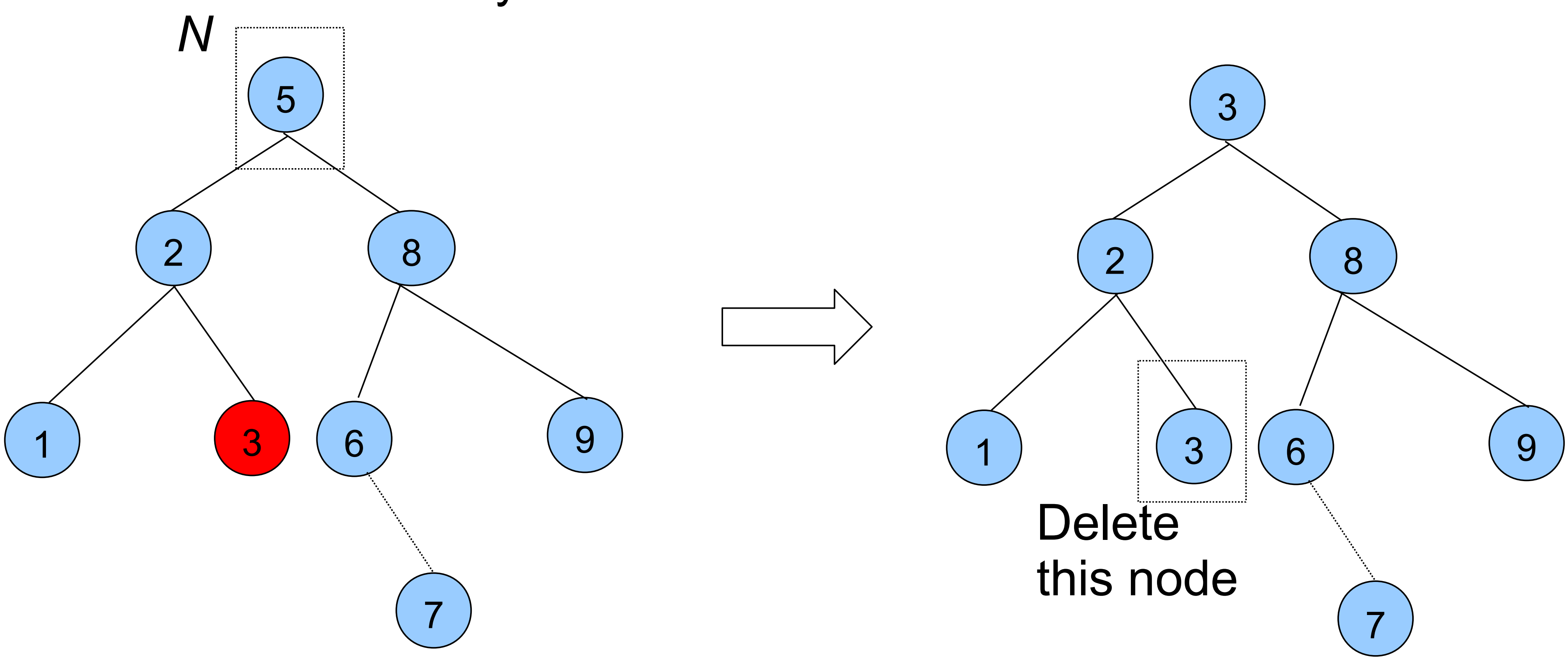
How do you find **immediate successor** of node *N*?

Minmei Hou, Ph.D., Dept. Computer Science, NIU

Deletion in B.S.T.

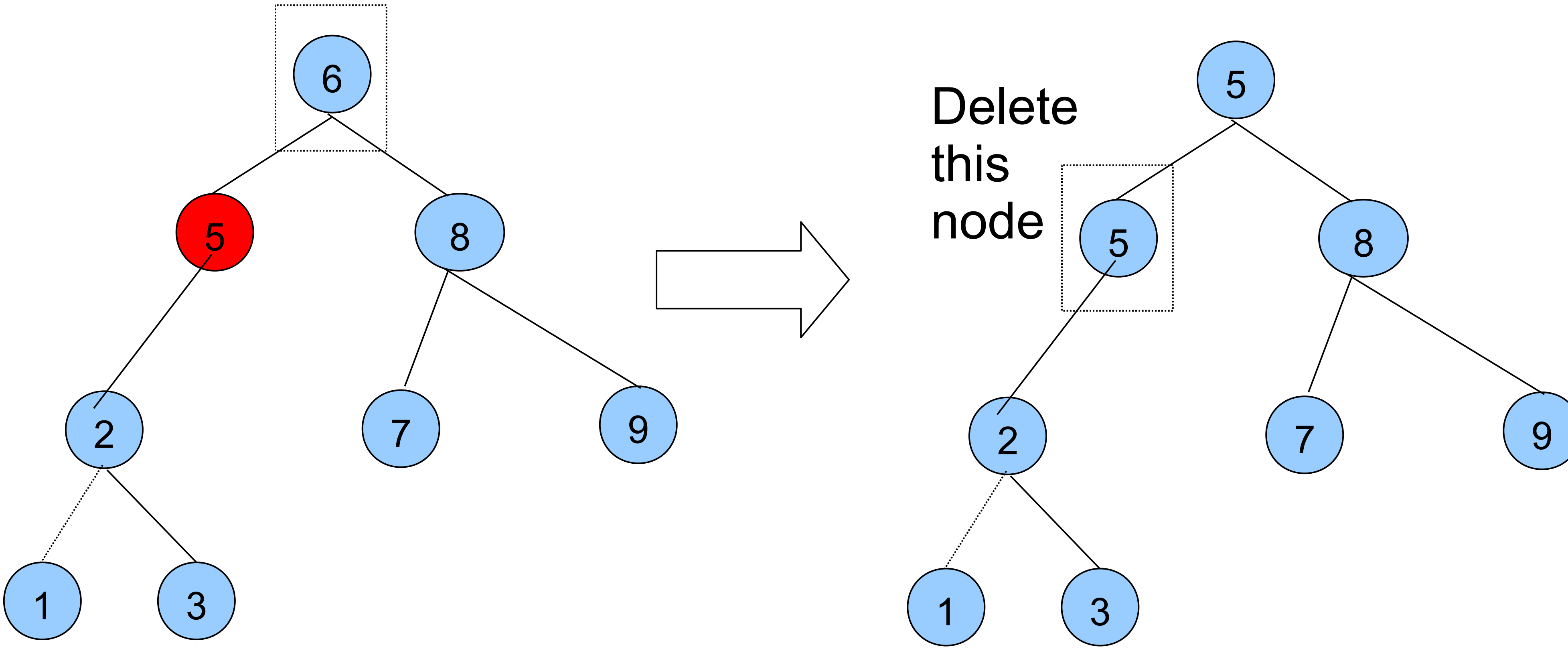
Approach II: deletion by copying.

- Replace N's value by N's immediate predecessor's value
- The problem becomes to delete N's immediate predecessor
  – It is a leaf
  – It has only one child.

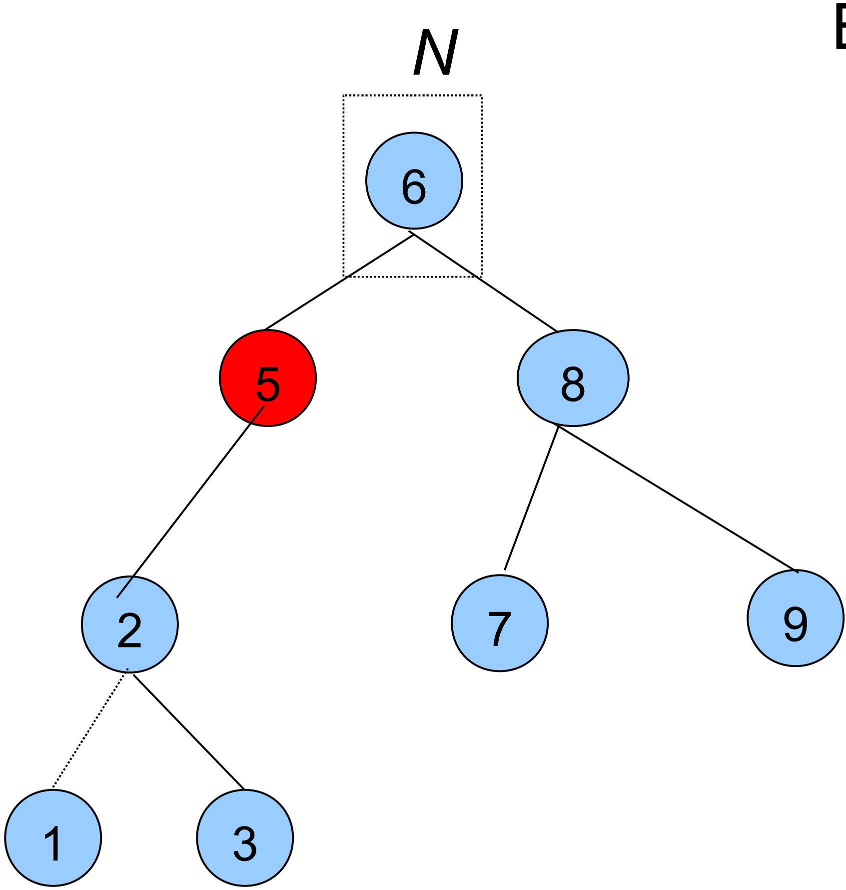


Deletion by copying

More example

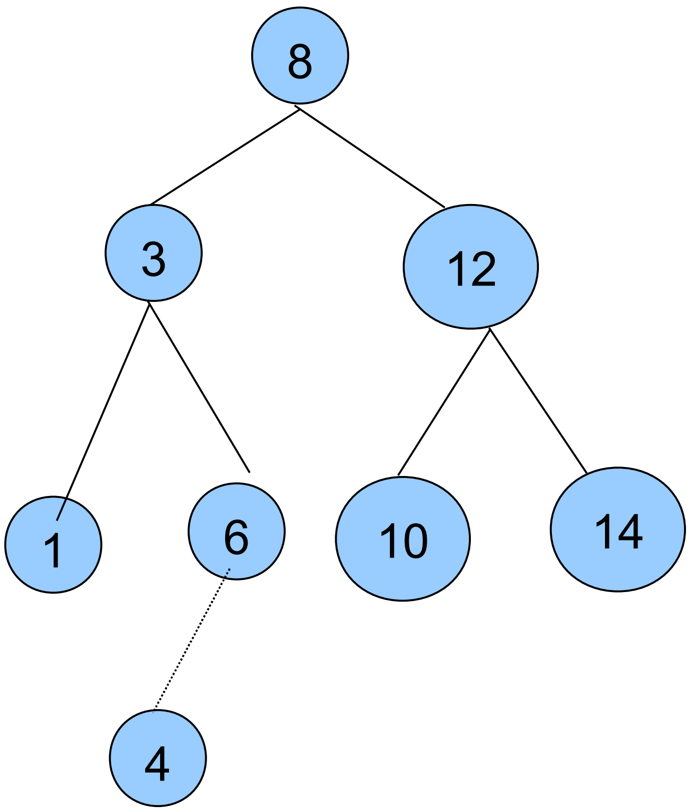


Pseudocode of deletion by copying



BSTDeletionByCopying (*N*)
    // N is passed by "passing-by-reference"
    if *N* is empty
        STOP
    if *N*'s right child is empty
        N ← *N*'s left child
    else if *N*'s left child is empty
        N ← *N*'s right child
    else // in case *N* has two children
        pred ← immediate predecessor of *N*
        N's value ← *pred*'s value
        tmp ← *pred*   // for deallocation later
        if *pred*'s right child is empty
            pred ← *pred*'s left child
        else if *pred*'s left child is empty
            pred ← *pred*'s right child
        deallocate *tmp*

Practice questions

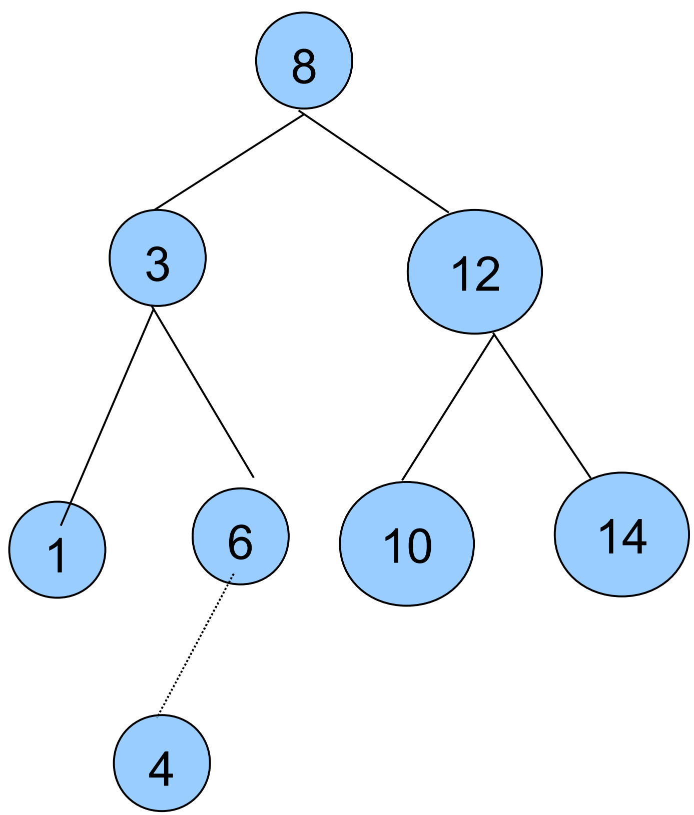


Questions:

What is the immediate predecessor of node 12?

What is the resulted tree if node 12 is deleted by merging?

Practice questions



Questions:

What is the immediate predecessor of node 8?

What is the resulted tree if node 8 is deleted by copying?