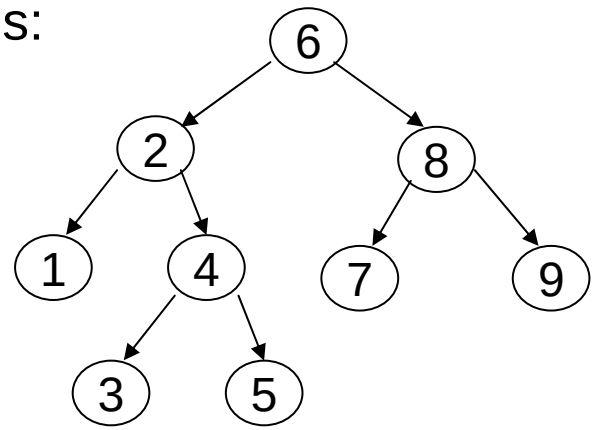
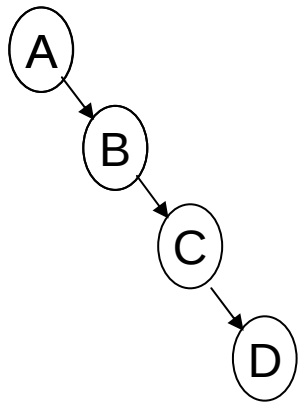
Binary Search Trees (BST)

- **What is a Binary search tree?**
- **Why Binary search trees?**
- **Binary search tree implementation**
- **Insertion in a BST**
- **Deletion from a BST**

Binary Search Trees (Definition)

- A binary search tree (BST) is a binary tree that is empty or that satisfies the BST ordering property:
  1. The key of each node is greater than each key in the left subtree, if any, of the node.
  2. The key of each node is less than each key in the right subtree, if any, of the node.
- Thus, each key in a BST is unique.
- Examples:





Why BST?

- BSTs provide good logarithmic time performance in the best and average cases.
- Average case complexities of using linear data structures compared to BSTs:

Data Structure	Retrieval	Insertion	Deletion
BST	O(log n) FAST	O(log n) FAST	O(log n) FAST
Sorted Array	O(log n) FAST*	O(n) SLOW	O(n) SLOW
Sorted Linked List	O(n) SLOW	O(n) SLOW	O(n) SLOW

*using binary search

Binary Search Tree Implementation

- The BinarySearchTree class inherits the instance variables key, left, and right of the BinaryTree class:

```
public class BinarySearchTree extends BinaryTree
                        implements SearchableContainer {
   private BinarySearchTree getLeftBST(){
      return (BinarySearchTree) getLeft( ) ;
   }

   private BinarySearchTree getRightBST( ){
      return (BinarySearchTree) getRight( ) ;
   }
   // . . .
}
```

Binary Search Tree Implementation (Cont'd)

- The find method of the BinarySearchTree class:

```
public Comparable find(Comparable comparable)
{
    if(isEmpty()) return null;
    Comparable key = (Comparable) getKey();
    if(comparable.compareTo(key)==0)
        return key;
    else if (comparable.compareTo(key)<0)
        return getLeftBST().find(comparable);
    else
        return getRightBST().find(comparable);
}
```

Binary Search Tree Implementation (Cont'd)

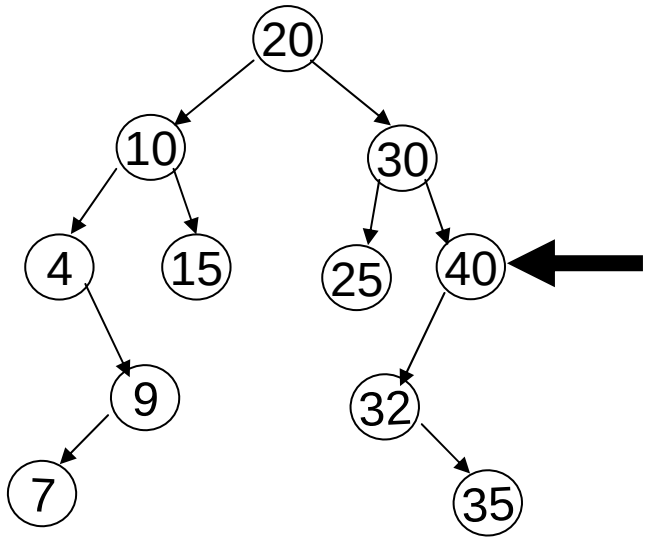
- The findMin method of the BinarySearchTree class:
  - By the BST ordering property, the minimum key is the key of the left-most node that has an empty left-subtree.

```
public Comparable findMin()
{
    if(isEmpty())
        return null;
    if(getLeftBST().isEmpty())
        return (Comparable)getKey();
    else
        return getLeftBST().findMin();
}
```

Binary Search Tree Implementation (Cont'd)

- The findMax method of the BinarySearchTree class:
  - By the BST ordering property, the maximum key is the key of the right-most node that has an empty right-subtree.

```
public Comparable findMax() {
  if(isEmpty())
     return null;
  if(getRightBST().isEmpty())
     return (Comparable)getKey();
  else
     return getRightBST().findMax();
}
```



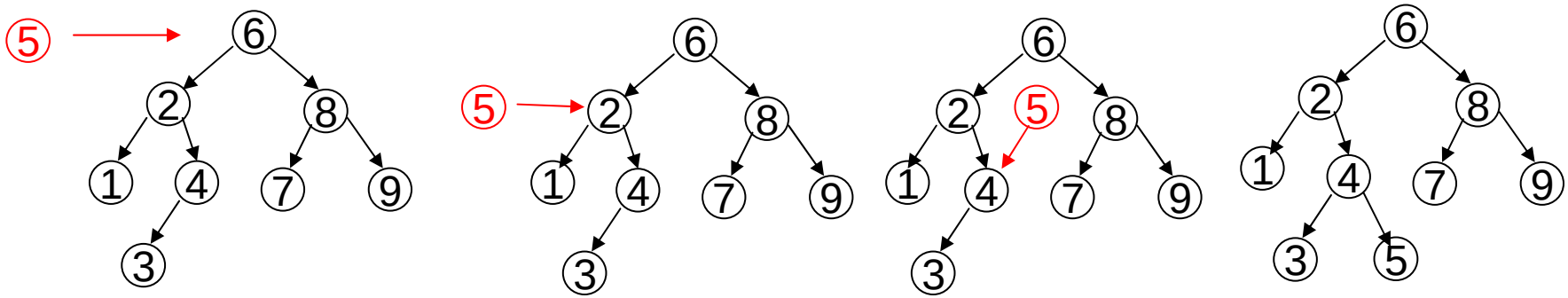
Insertion in a BST

- By the BST ordering property, a new node is always inserted as a leaf node.
- The insert method, given in the next page, recursively finds an appropriate empty subtree to insert the new key. It then transforms this empty subtree into a leaf node by invoking the attachKey method:

```
public void attachKey(Object obj) {
    if(!isEmpty())
        throw new InvalidOperationException();
    else {
        key = obj;
        left = new BinarySearchTree();
        right = new BinarySearchTree();
    }
}
```

Insertion in a BST

```
public void insert(Comparable comparable){
  if(isEmpty())
     attachKey(comparable);
  else {
     Comparable key = (Comparable) getKey();
     if(comparable.compareTo(key)==0)
       throw new IllegalArgumentException("duplicate key");
     else if (comparable.compareTo(key)<0)
       getLeftBST().insert(comparable);
     else
       getRightBST().insert(comparable);
    }
}
```



Deletion in a BST

- There are three cases:
  1. The node to be deleted is a leaf node.
  2. The node to be deleted has one non-empty child.
  3. The node to be deleted has two non-empty children.

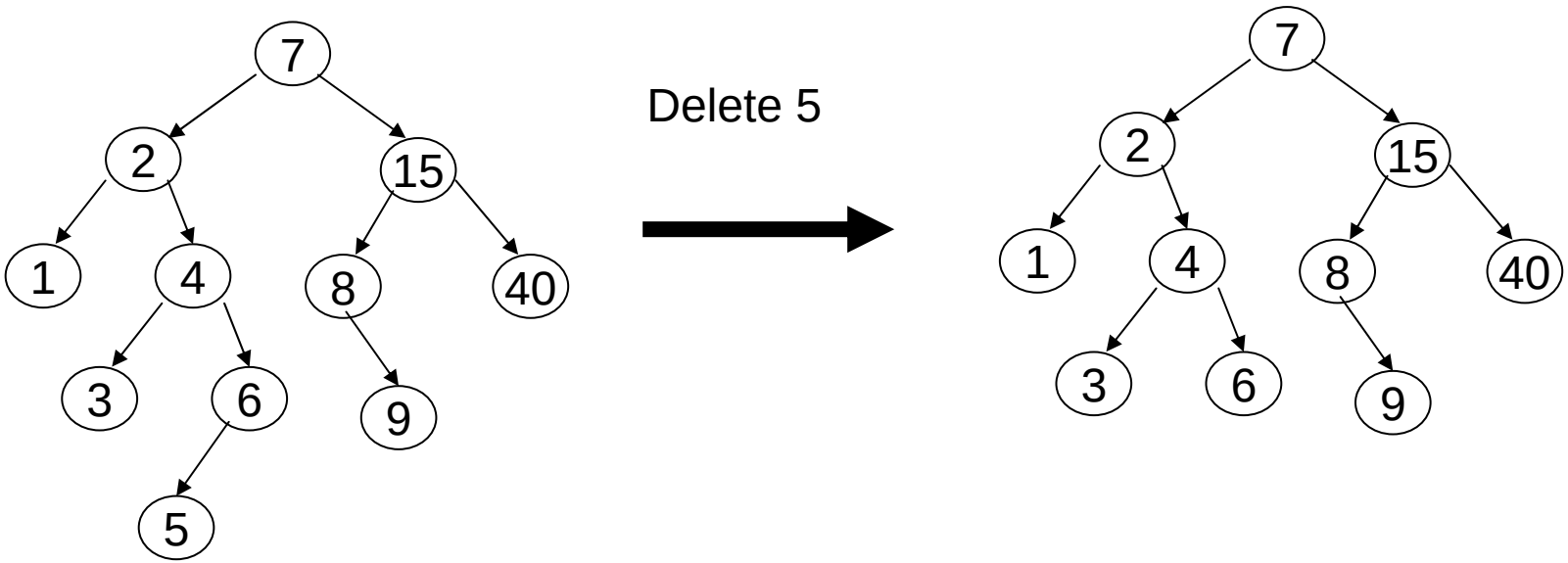
CASE 1: DELETING A LEAF NODE

Convert the leaf node into an empty tree by using the detachKey method:

```
// In Binary Tree class
public Object  detachKey( ){
   if(! isLeaf( )) throw new InvalidOperationException( ) ;
   else {
      Object obj = key ;
      key = null ;
      left = null ;
      right = null ;
      return obj ;
   }
}
```

Deleting a leaf node (cont'd)

- Example: Delete 5 in the tree below:



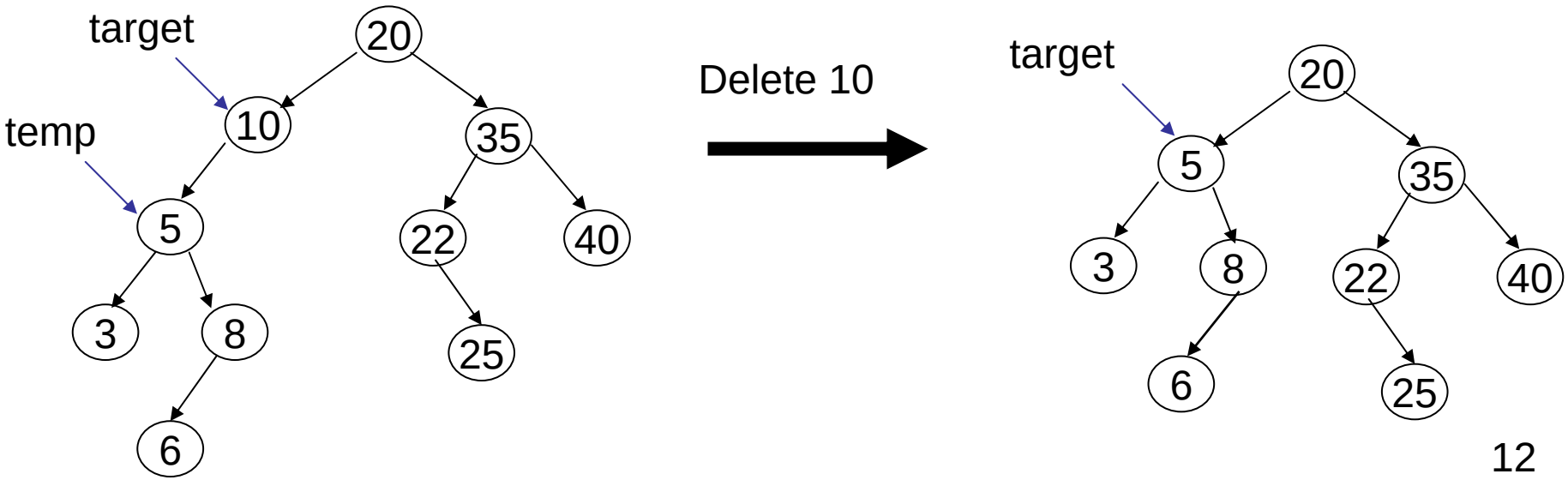
Deleting a one-child node

- **CASE 2: THE NODE TO BE DELETED HAS ONE NON-EMPTY CHILD**

  (a) The right subtree of the node x to be deleted is empty.

```
// Let target be a reference to the node x.
BinarySearchTree temp = target.getLeftBST();
target.key = temp.key;
target.left = temp.left;
target.right = temp.right;
temp = null;
```

- Example:

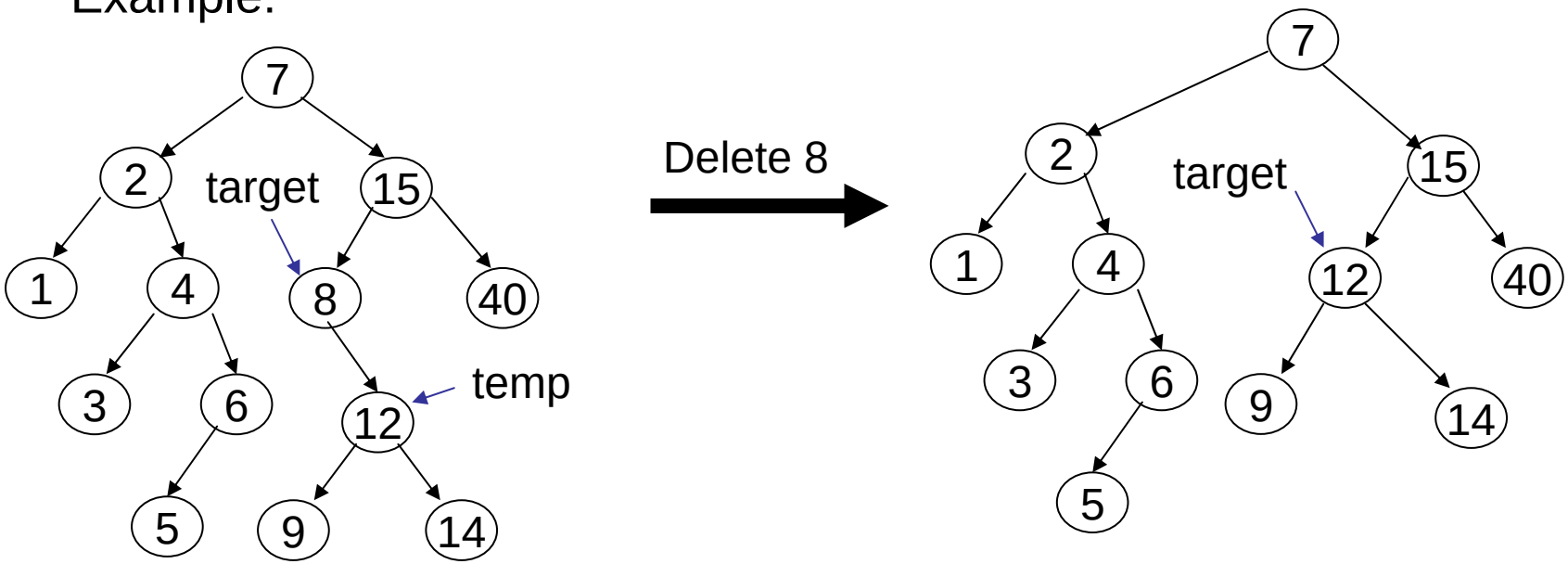


Deleting a one-child node (cont'd)

(b) The left subtree of the node x to be deleted is empty.

```
// Let target be a reference to the node x.
BinarySearchTree temp = target.getRightBST();
target.key = temp.key;
target.left = temp.left;
target.right = temp.right;
temp = null;
```

Example:

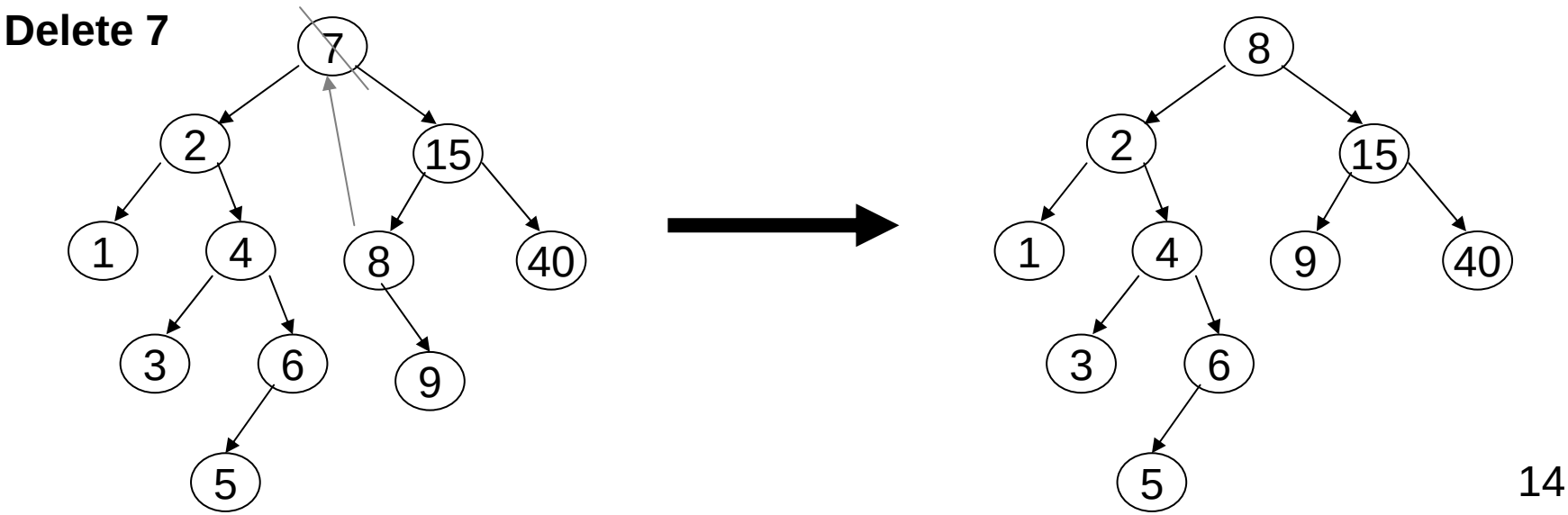


CASE 3: DELETING A NODE THAT HAS TWO NON-EMPTY CHILDREN

DELETION BY COPYING: METHOD#1

Copy the **minimum** key in the **right** subtree of x to the node x, then delete the one-child or leaf-node with this minimum key.

- Example:

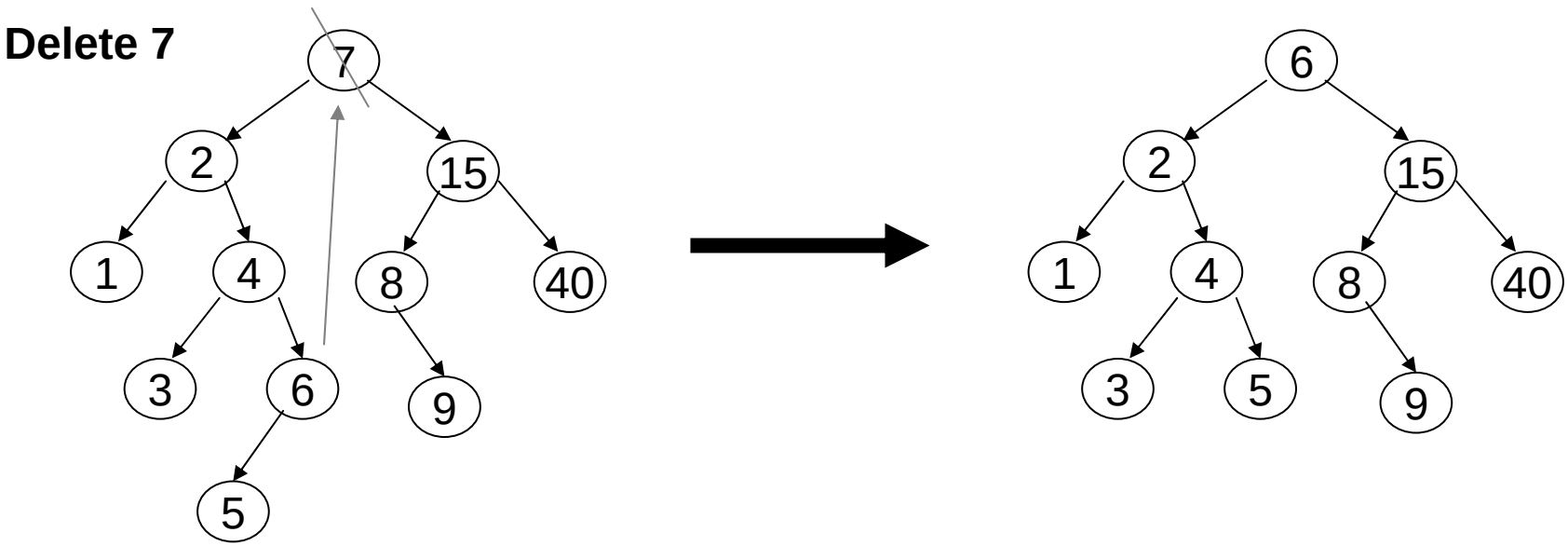


DELETING A NODE THAT HAS TWO NON-EMPTY CHILDREN

DELETION BY COPYING: METHOD#2

Copy the **maximum** key in the **left** subtree of x to the node x, then delete the one-child or leaf-node with this **maximum** key.

- Example:



Two-child deletion method#1 code

```
// find the minimum key in the right subtree of the target node
Comparable min = target.getRightBST().findMin();

// copy the minimum value to the target
target.key = min;

// delete the one-child or leaf node having the min
target.getRightBST().withdraw(min);
```

All the different cases for deleting a node are handled in the **withdraw (Comparable key)** method of **BinarySearchTree** class