Revising past & Advancing trees

Seminar 3.

Reinvent the workshops

	Seminar 1	Seminar 2
All together	1. Merge 2 sorted arrays 2. Basic BST (search, insert)	1. Extended BST (delete) 2. Self-balancing tree
Pro level	1. Extended BST (delete) 2. Self-balancing tree	1. Treap

Reinvent the workshops

tips for Pro

1. First implement delete method for BST
   - Take into account 3 cases
   - Most complicated case - 2 children

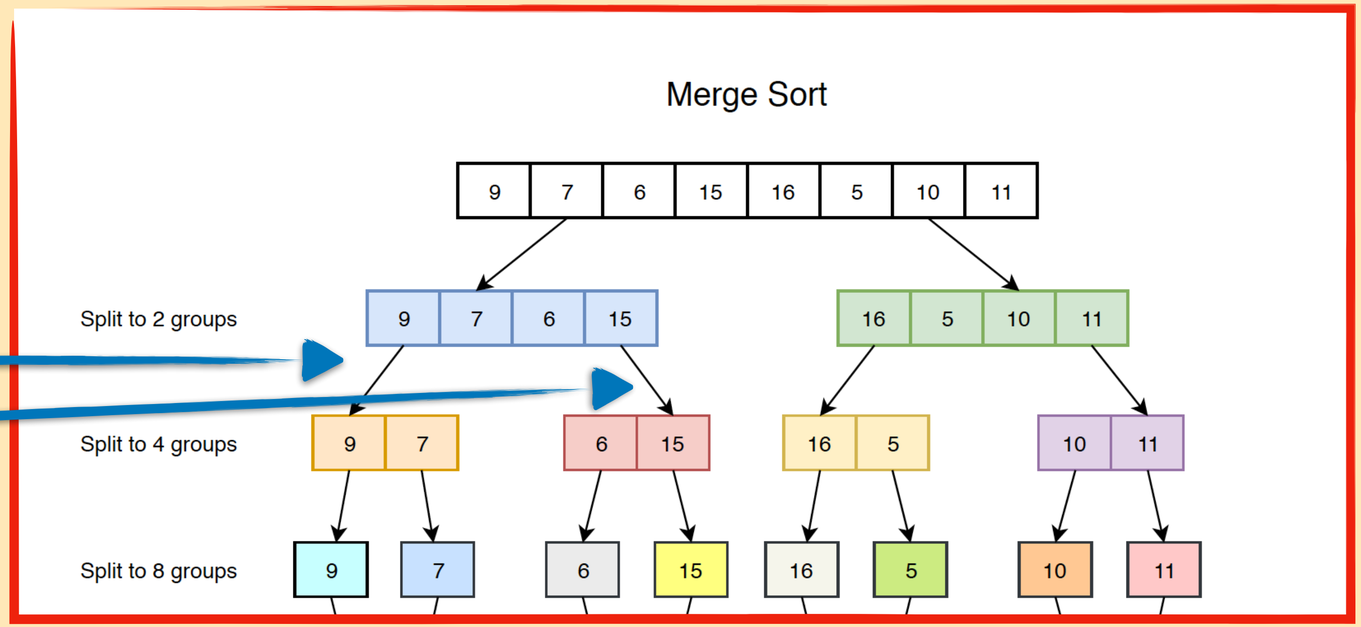
2. Then start AVL
   - Change the Node class
   - Think of all rotation cases
   - Implement rotation methods
   - Implement the insert method

Once again about Merge sort

Pseudo code

```
void mergeSortAlgorithm(int A[], int l, int r)
{
    if(l >= r)
        return
    int mid = l + (r - l)/2
    mergeSortAlgorithm(A, l, mid)
    mergeSortAlgorithm(A, mid + 1, r)
    merge(A, l, mid, r)
}
```

Visualization

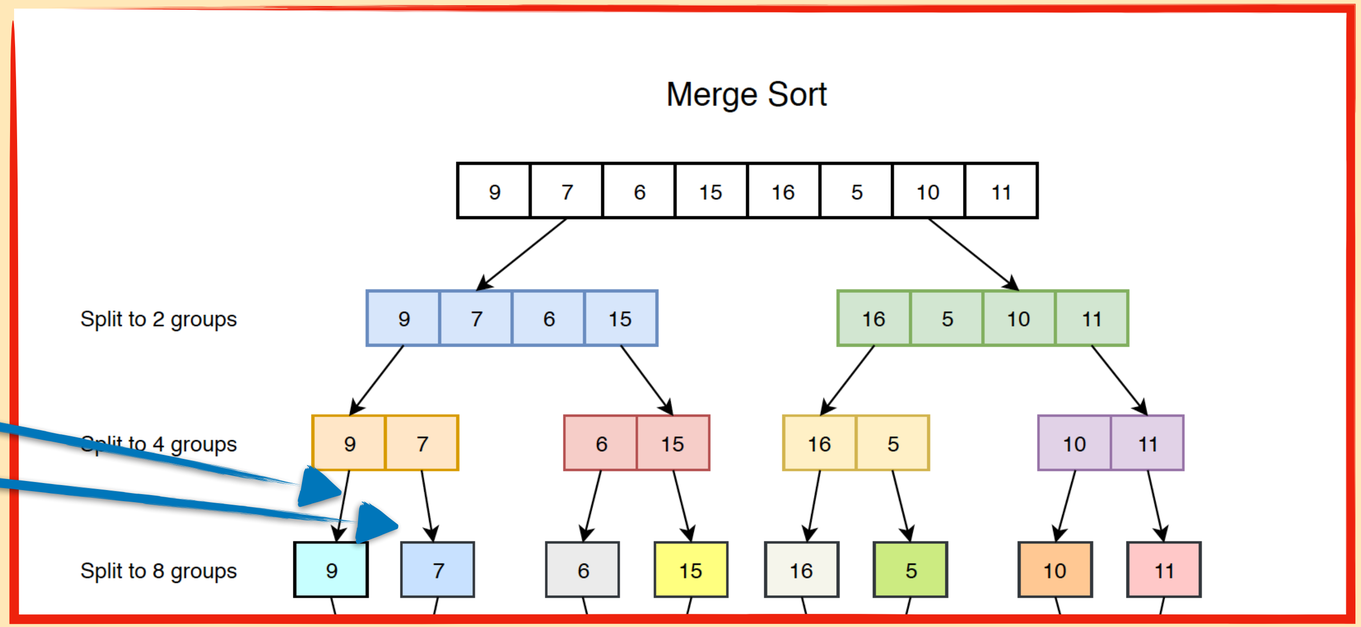


Once again about Merge sort

Pseudo code

```
void mergeSortAlgorithm(int A[], int l, int r)
{
    if(l >= r)
        return
    int mid = l + (r - l)/2
    mergeSortAlgorithm(A, l, mid)
    mergeSortAlgorithm(A, mid + 1, r)
    merge(A, l, mid, r)
}
```

Visualization

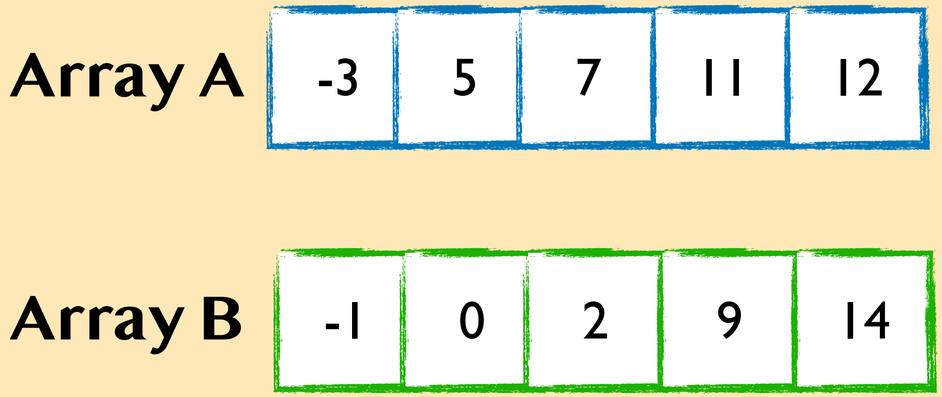


Once again about Merge sort

merge 2 sorted arrays...

Pseudo code

```
void mergeSortAlgorithm(int A[], int l, int r)
{
    if(l >= r)
        return
    int mid = l + (r - l)/2
    mergeSortAlgorithm(A, l, mid)
    mergeSortAlgorithm(A, mid + 1, r)
    merge(A, l, mid, r)
}
```

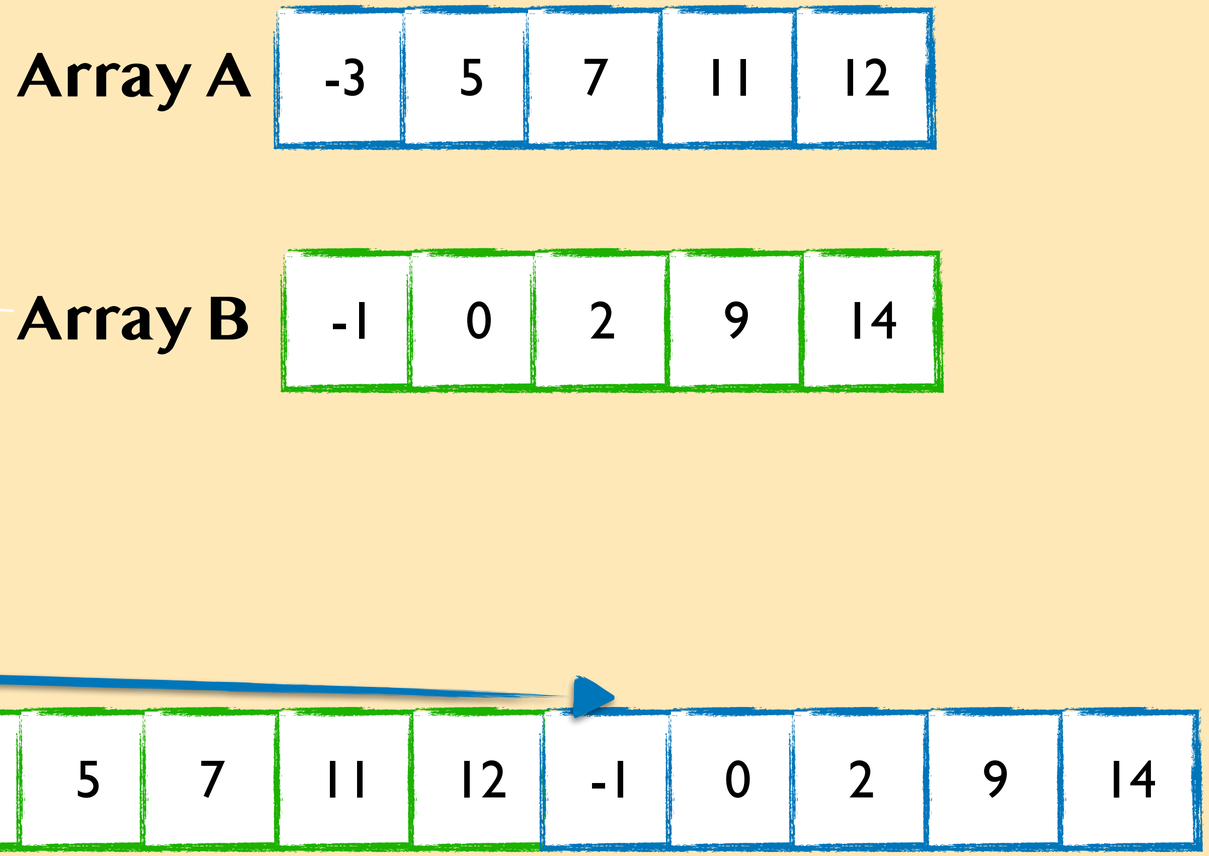


Once again about Merge sort

merge 2 sorted arrays...

Pseudo code

```
void mergeSortAlgorithm(int A[], int l, int r)
{
    if(l >= r)
        return
    int mid = l + (r - l)/2
    mergeSortAlgorithm(A, l, mid)
    mergeSortAlgorithm(A, mid + 1, r)
    merge(A, l, mid, r)
}
```

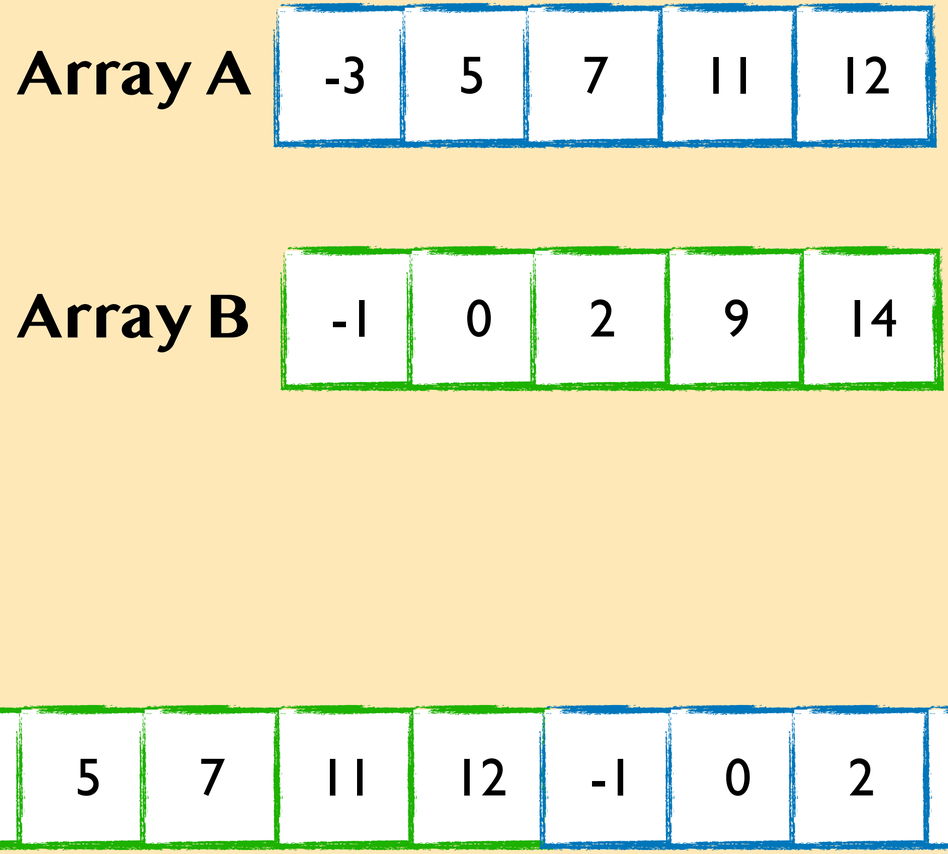


Once again about Merge sort

merge 2 sorted arrays...

Pseudo code

```
void mergeSortAlgorithm(int A[], int l, int r)
{
    if(l >= r)
        return
    int mid = l + (r - l)/2
    mergeSortAlgorithm(A, l, mid)
    mergeSortAlgorithm(A, mid + 1, r)
    merge(A, l, mid, r)
}
```

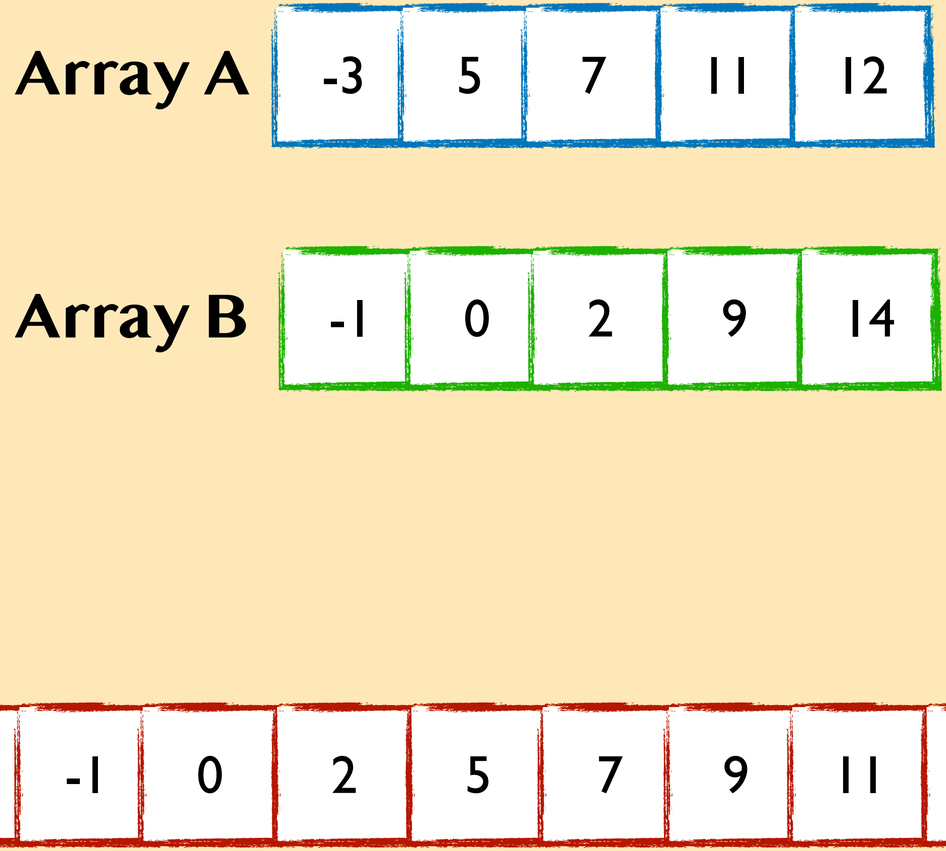


Once again about Merge sort

merge 2 sorted arrays...

Pseudo code

```
void mergeSortAlgorithm(int A[], int l, int r)
{
    if(l >= r)
        return
    int mid = l + (r - l)/2
    mergeSortAlgorithm(A, l, mid)
    mergeSortAlgorithm(A, mid + 1, r)
    merge(A, l, mid, r)
}
```

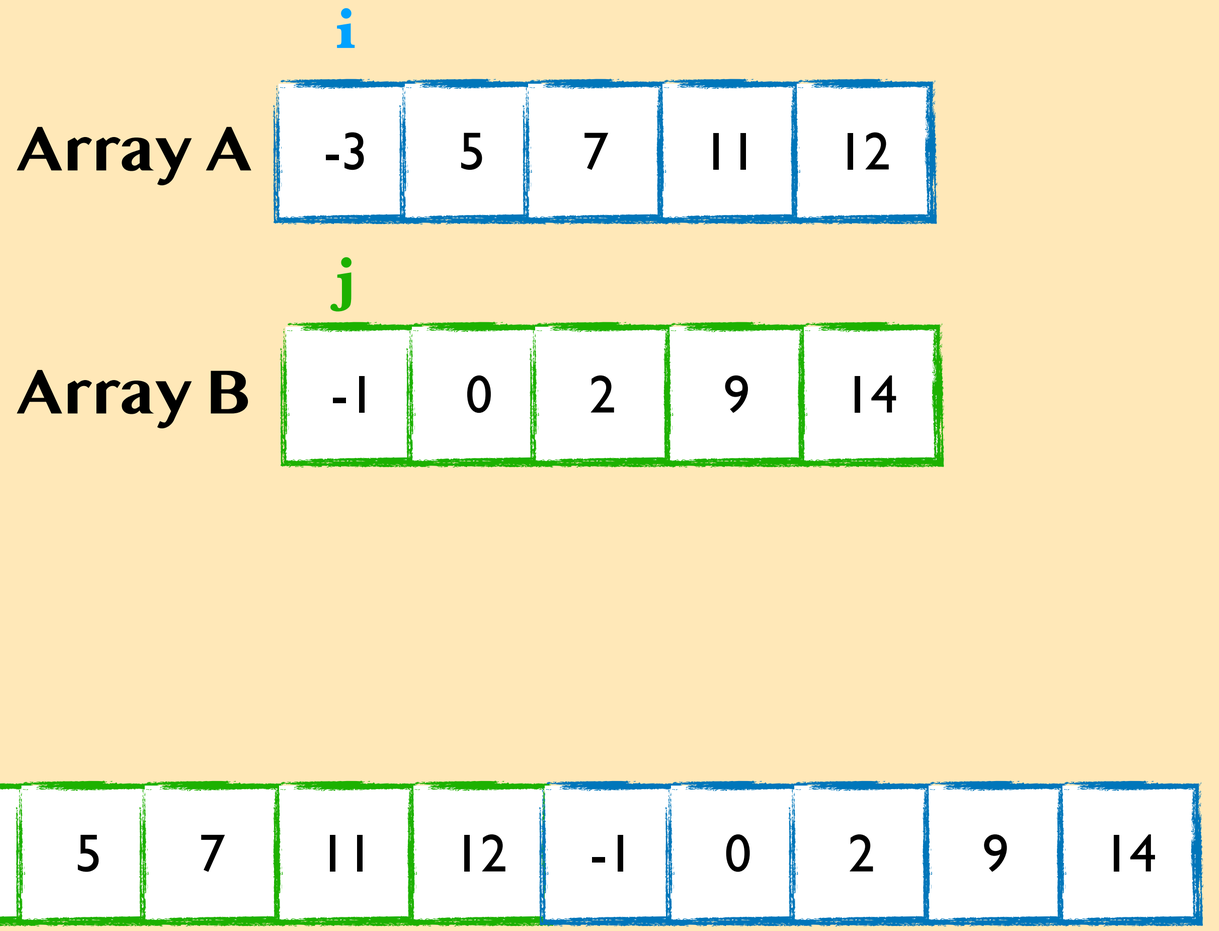


Once again about Merge sort

merge 2 sorted arrays...

Pseudo code

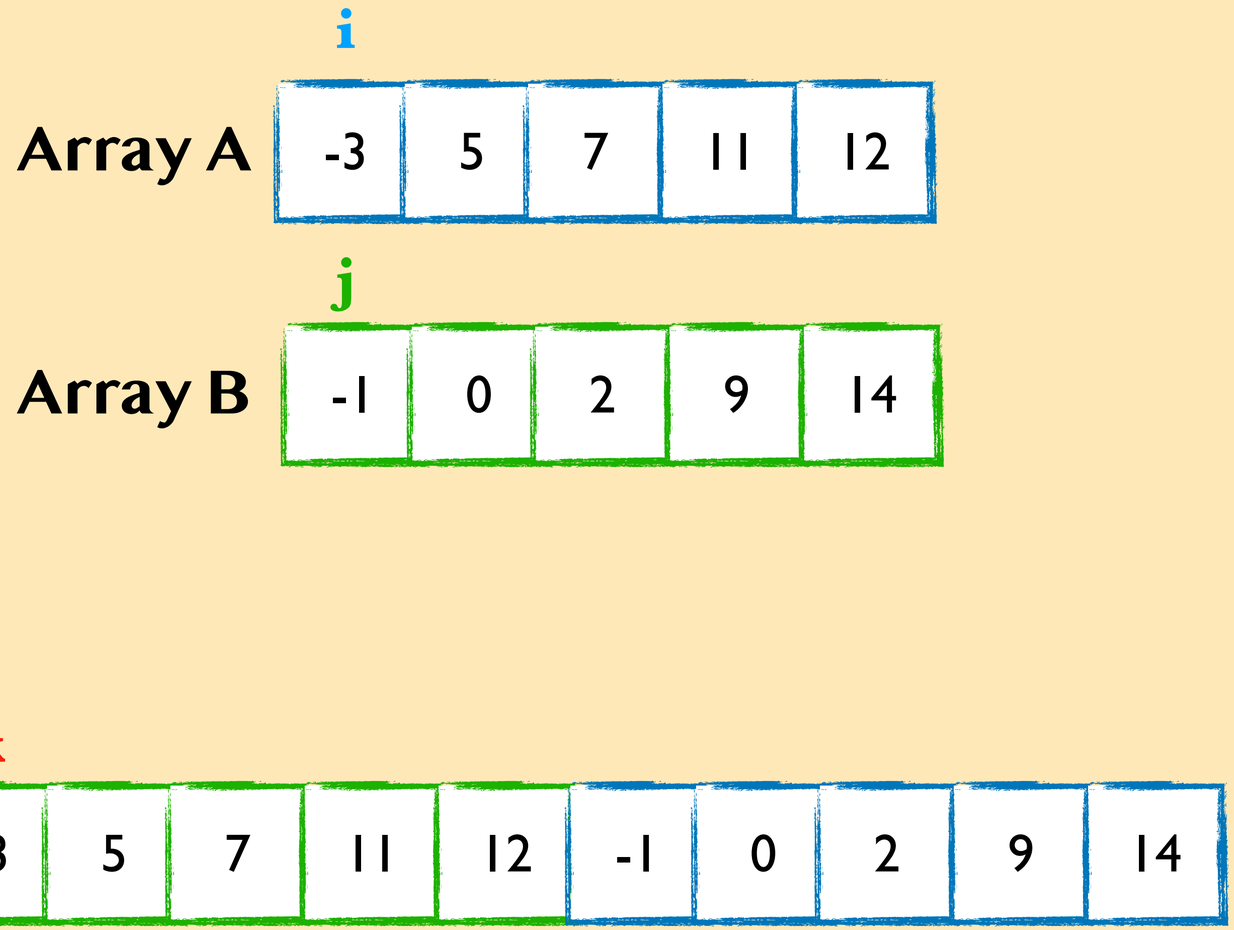
```
void mergeSortAlgorithm(int A[], int l, int r)
{
    if(l >= r)
        return
    int mid = l + (r - l)/2
    mergeSortAlgorithm(A, l, mid)
    mergeSortAlgorithm(A, mid + 1, r)
    merge(A, l, mid, r)
}
```



Once again about Merge sort

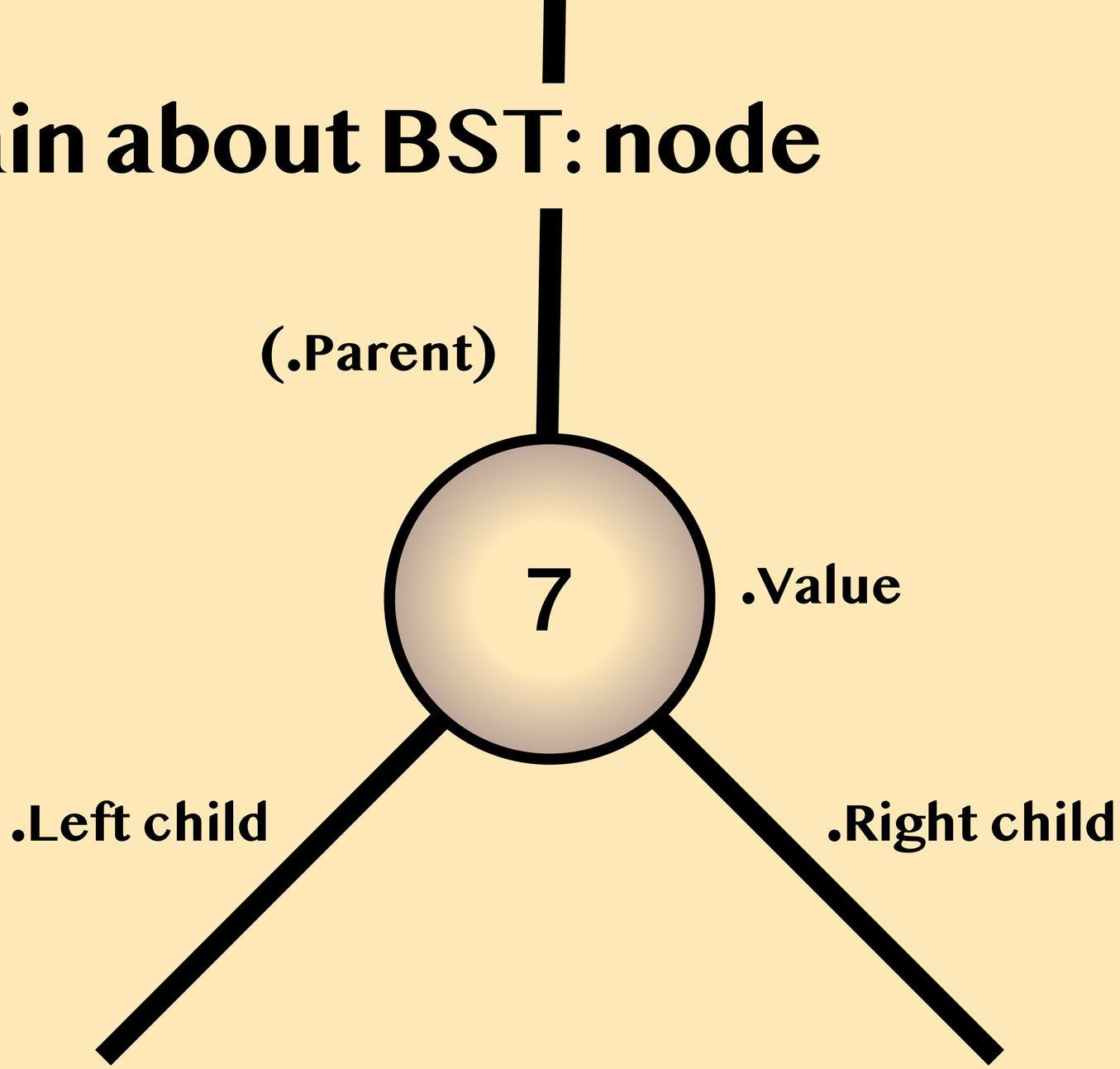
merge 2 sorted arrays...

```
i, j, k = 0, 0, 0
while i < len(left) and j < len(right):
    if left[i] <= right[j]:
        array[k] = left[i]
        i += 1
    else:
        array[k] = right[j]
        j += 1
    k += 1

while i < len(left):
    array[k] = left[i]
    i += 1
    k += 1

while j < len(right):
    array[k] = right[j]
    j += 1
    k += 1
```



Once again about BST: node

Once again about BST: node

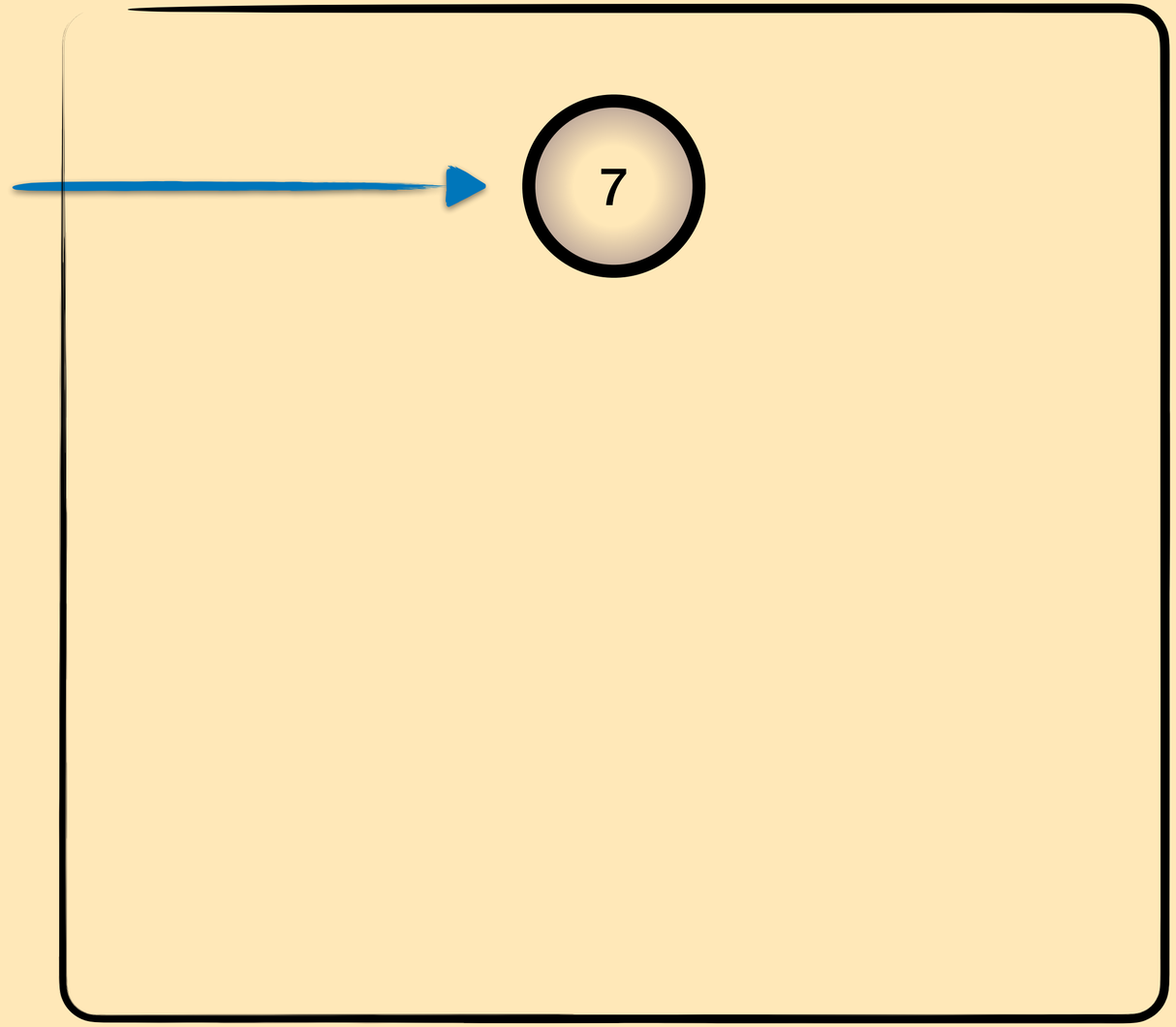


Once again about BST: insert

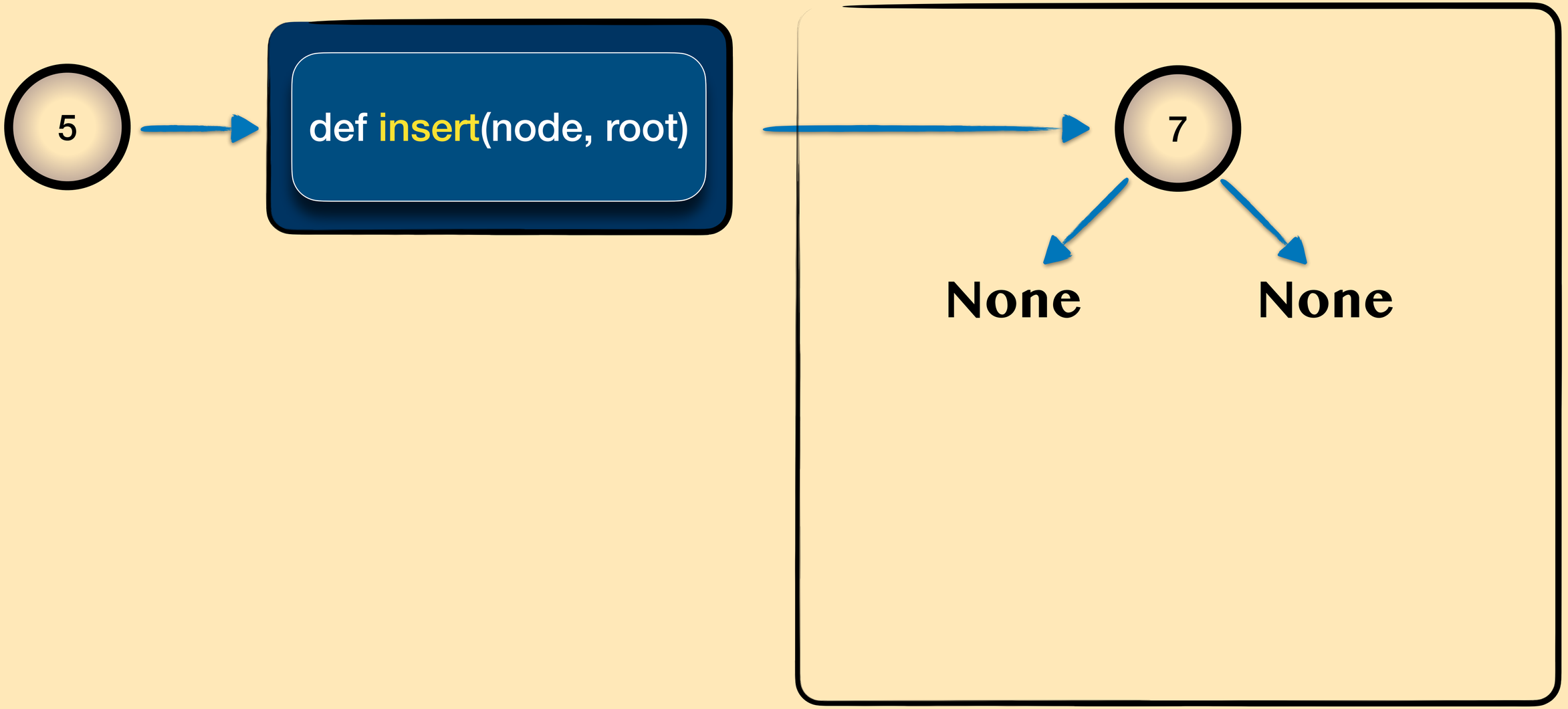
None

Once again about BST: insert

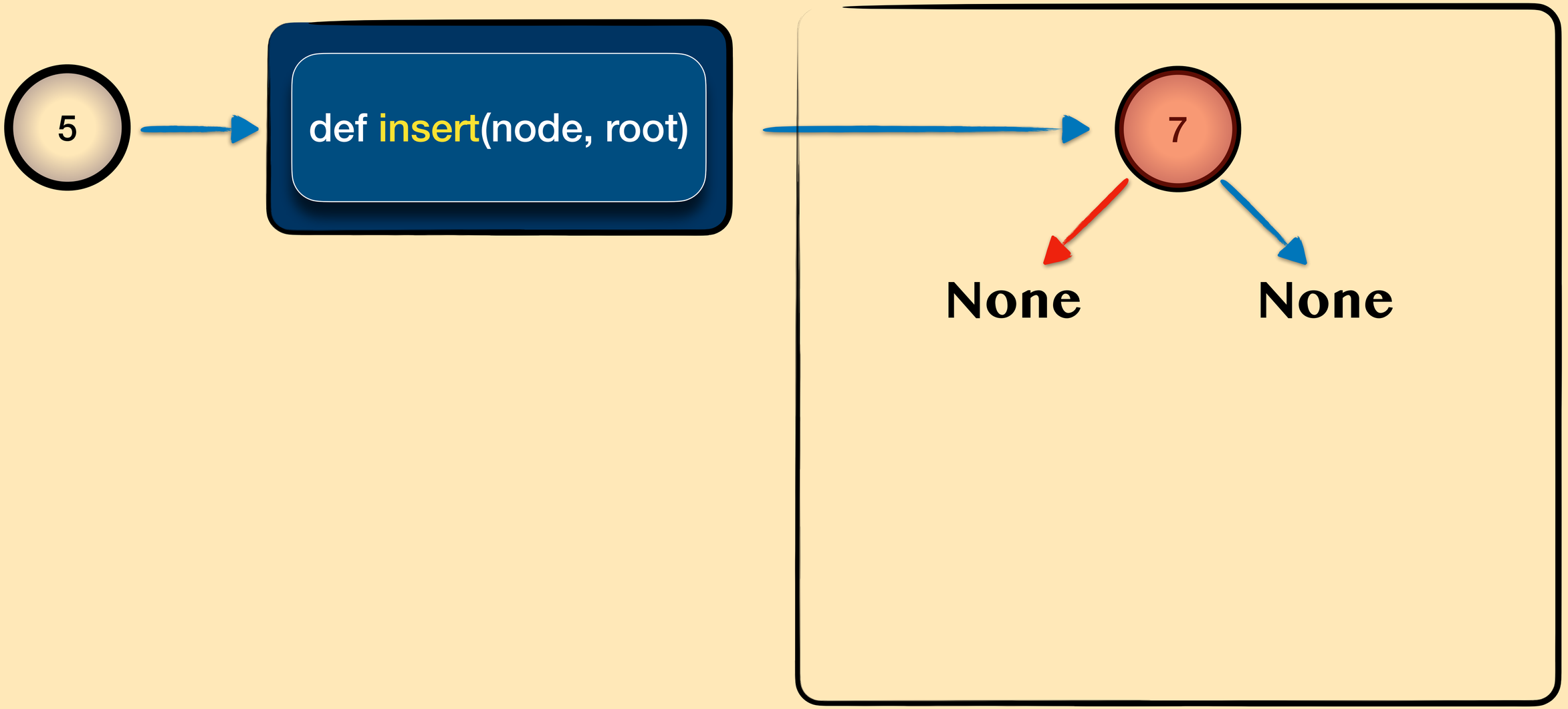
```
def insert(node, root)
```



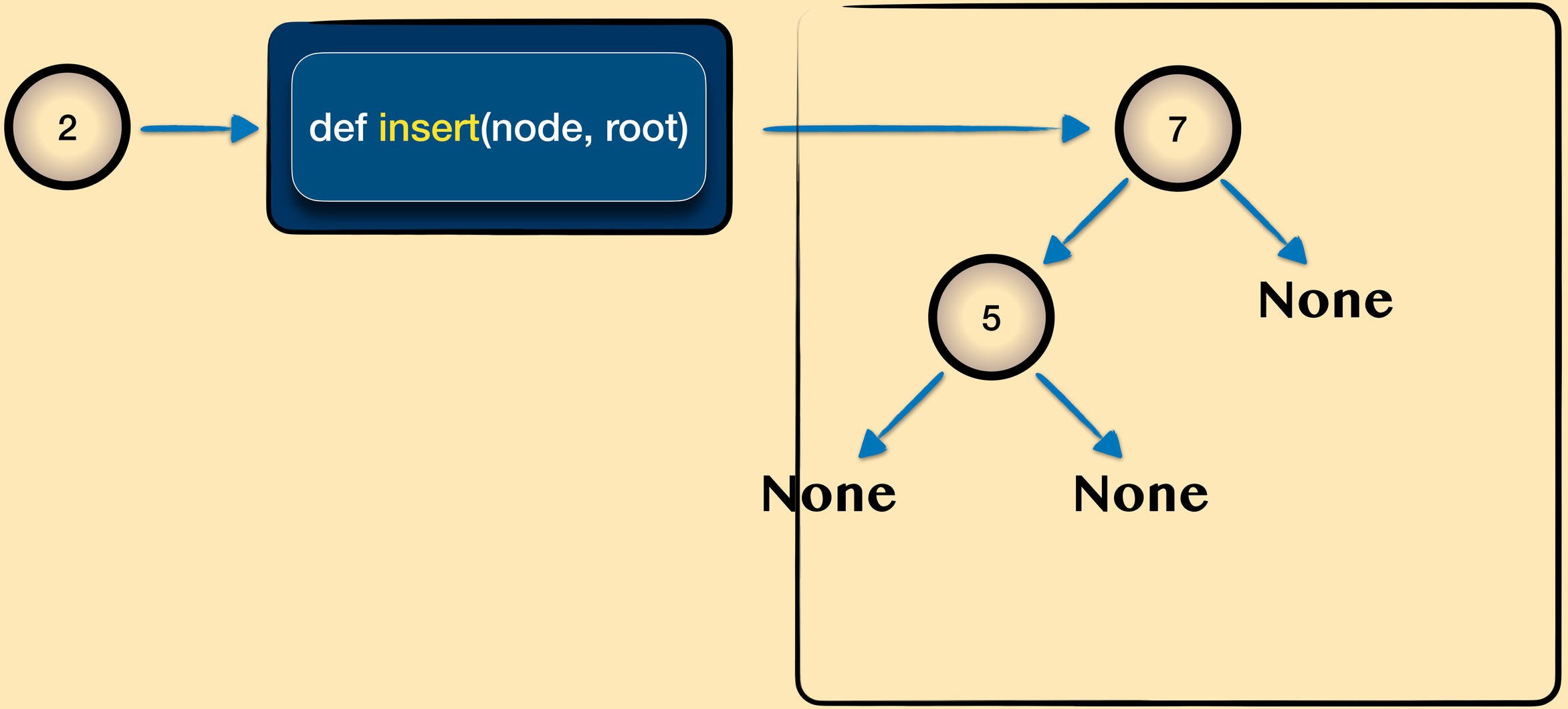
Once again about BST: insert



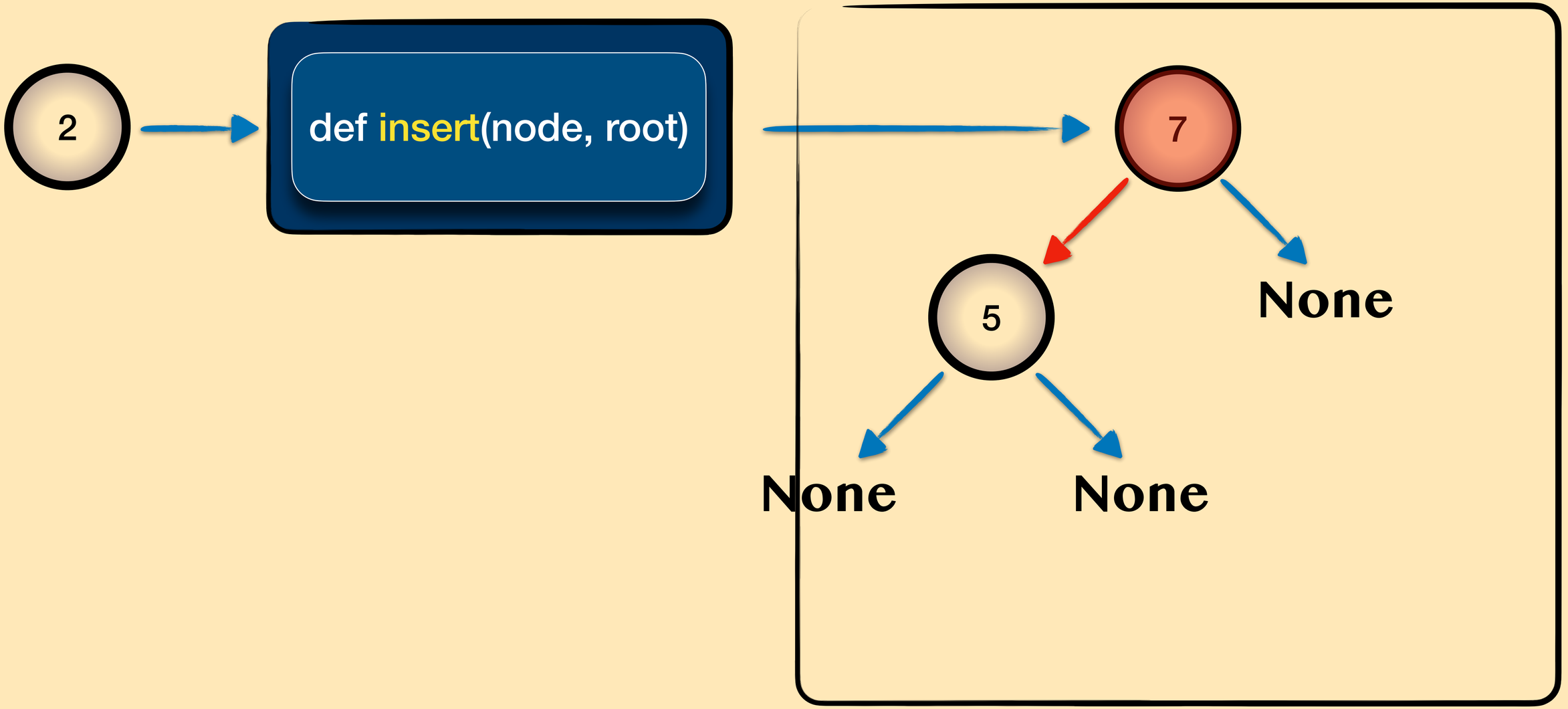
Once again about BST: insert



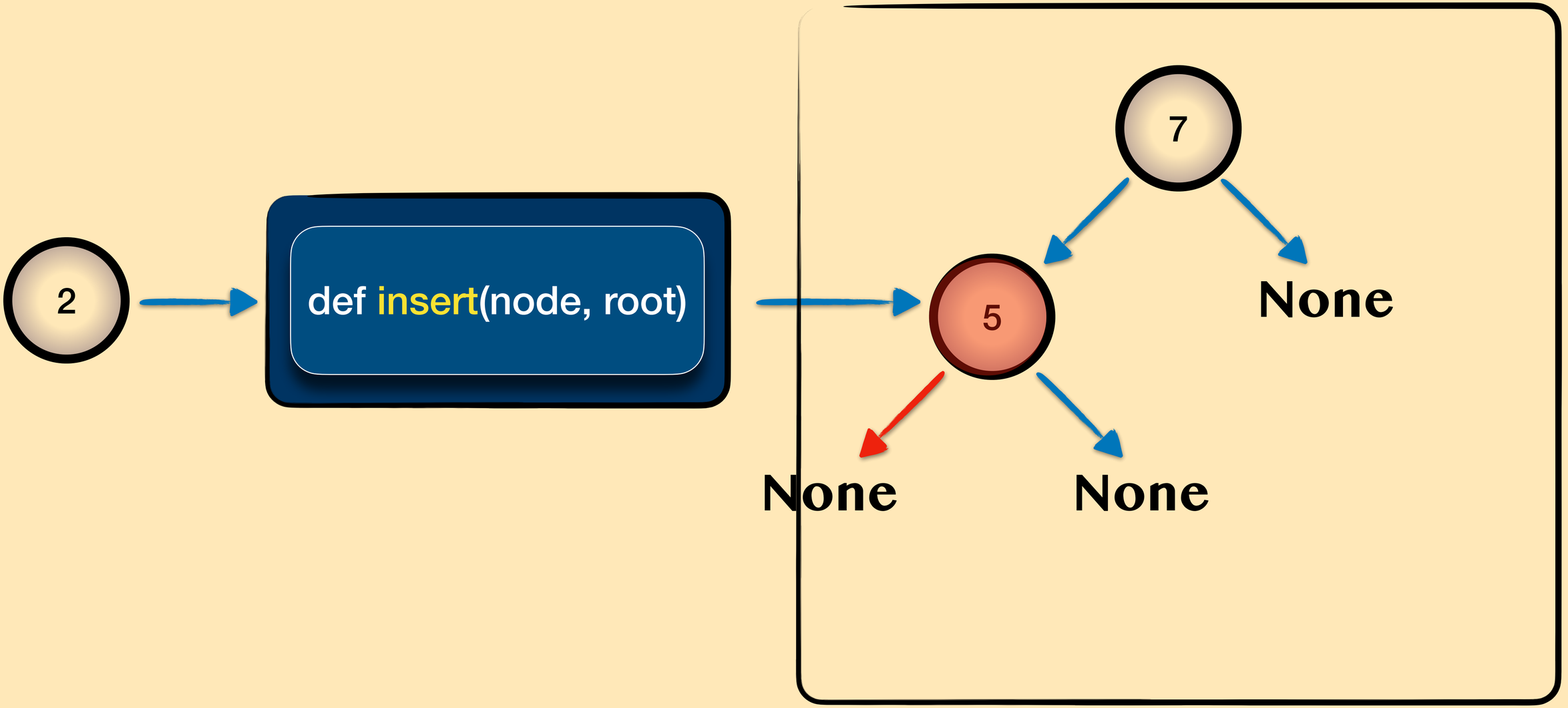
Once again about BST: insert



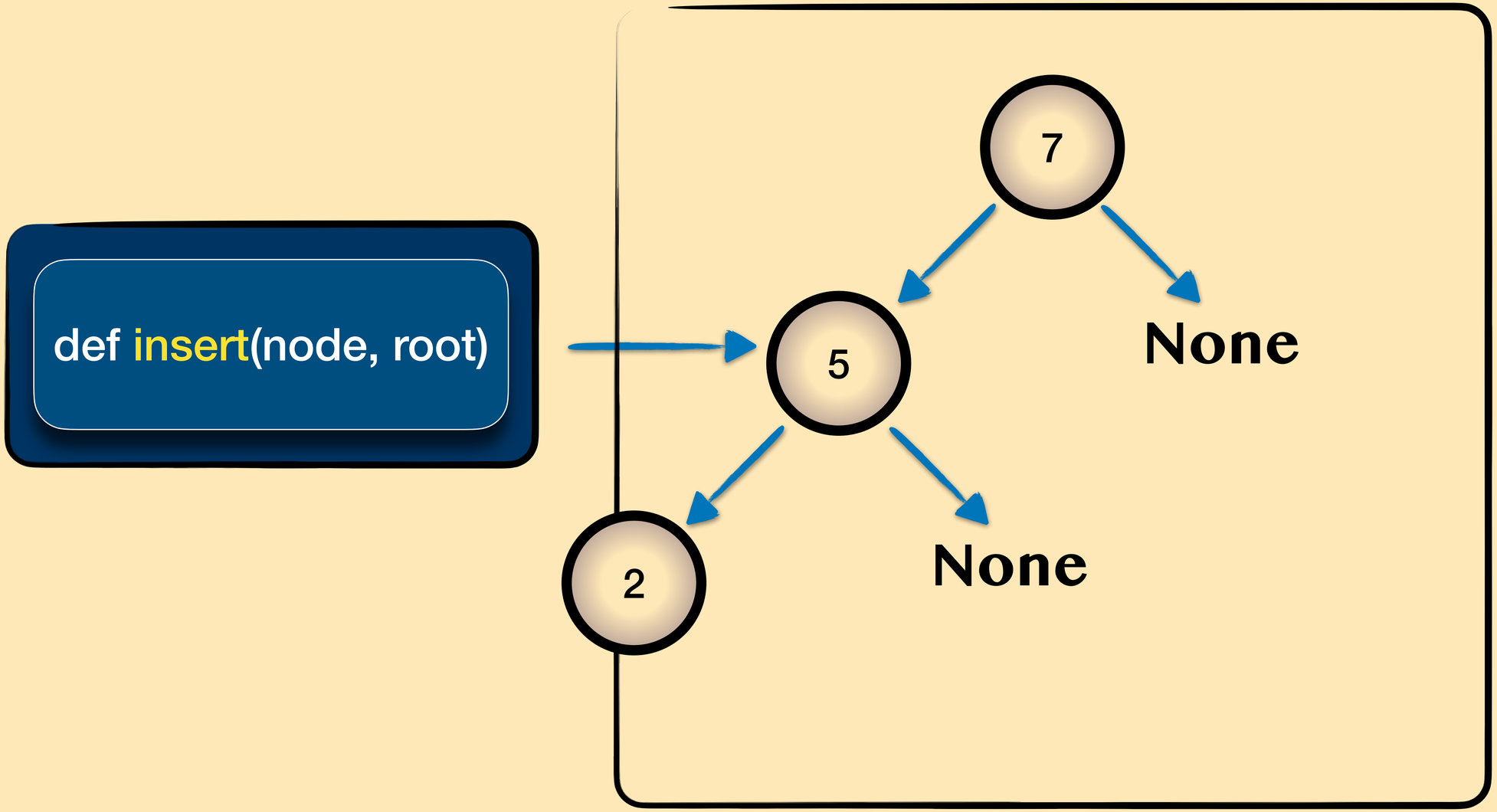
Once again about BST: insert



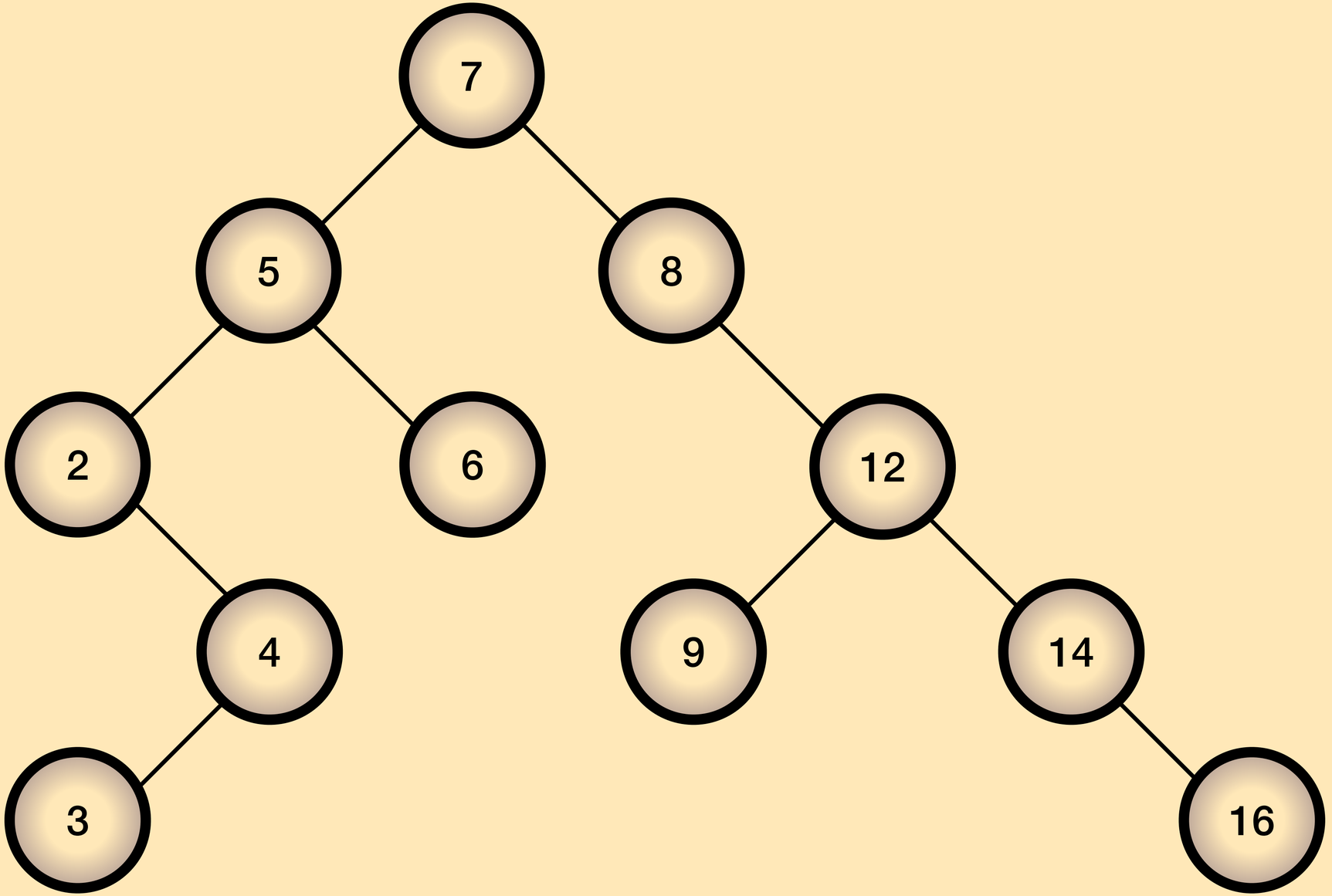
Once again about BST: insert



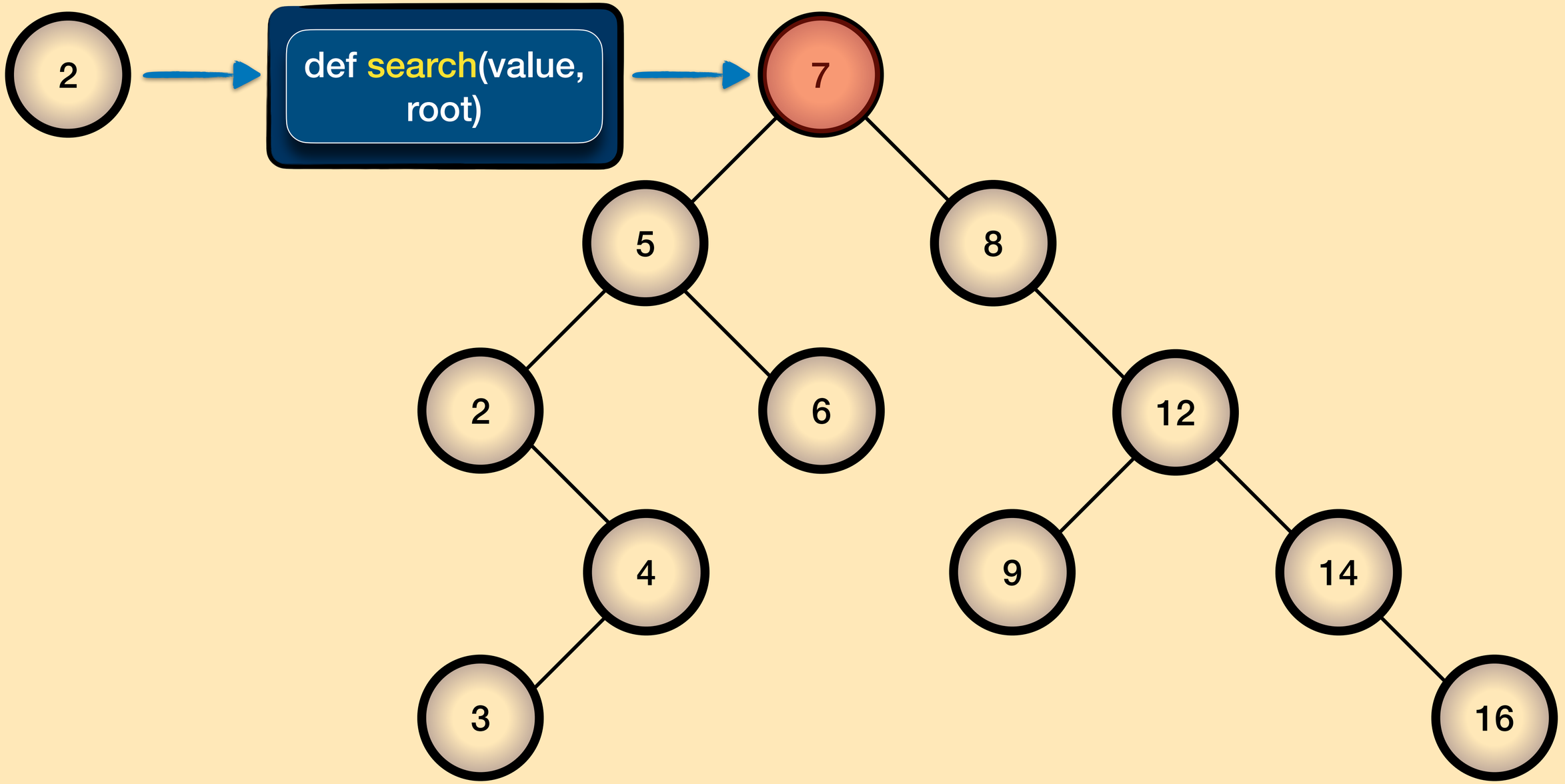
Once again about BST: insert



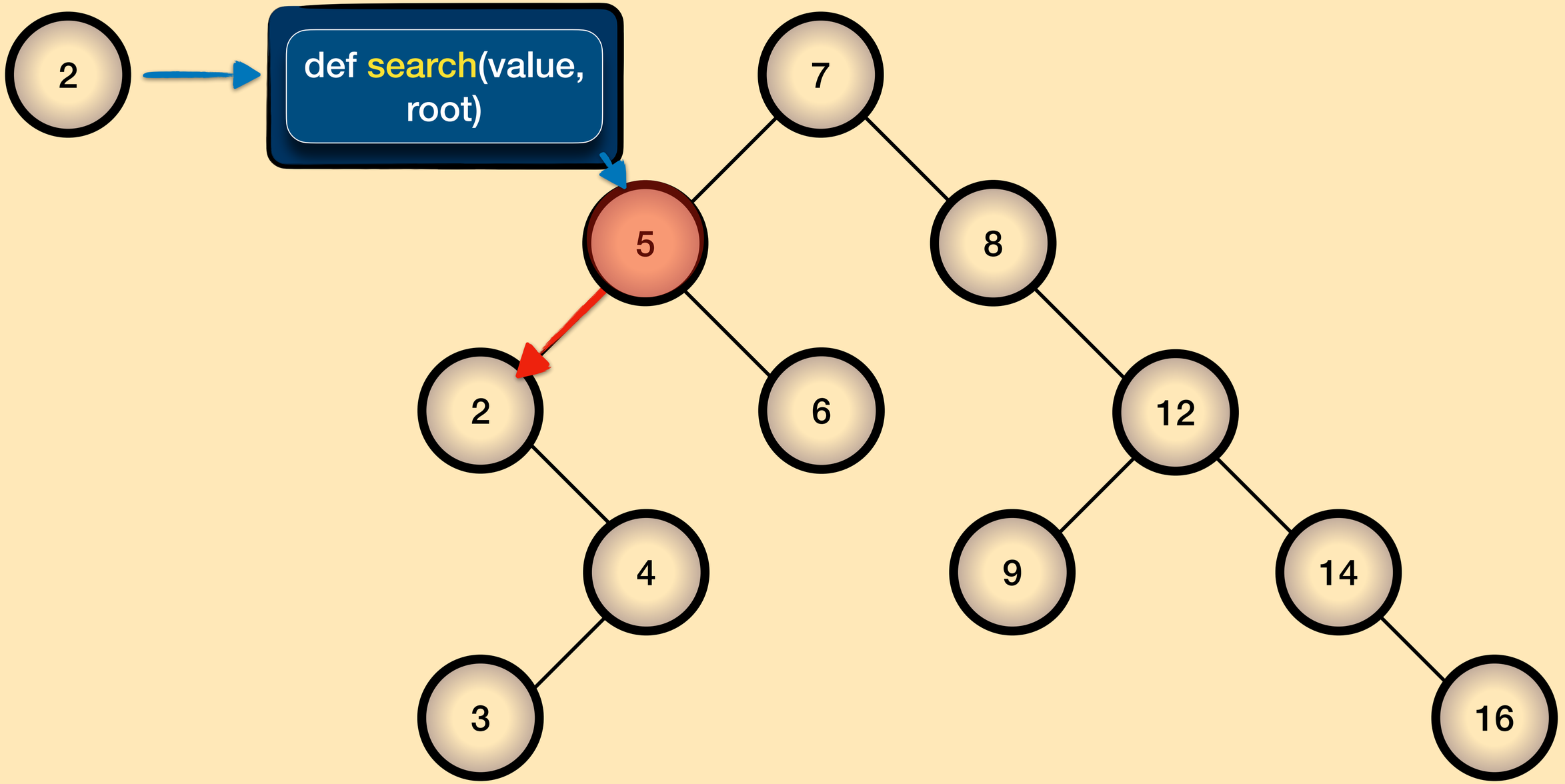
Once again about BST: search



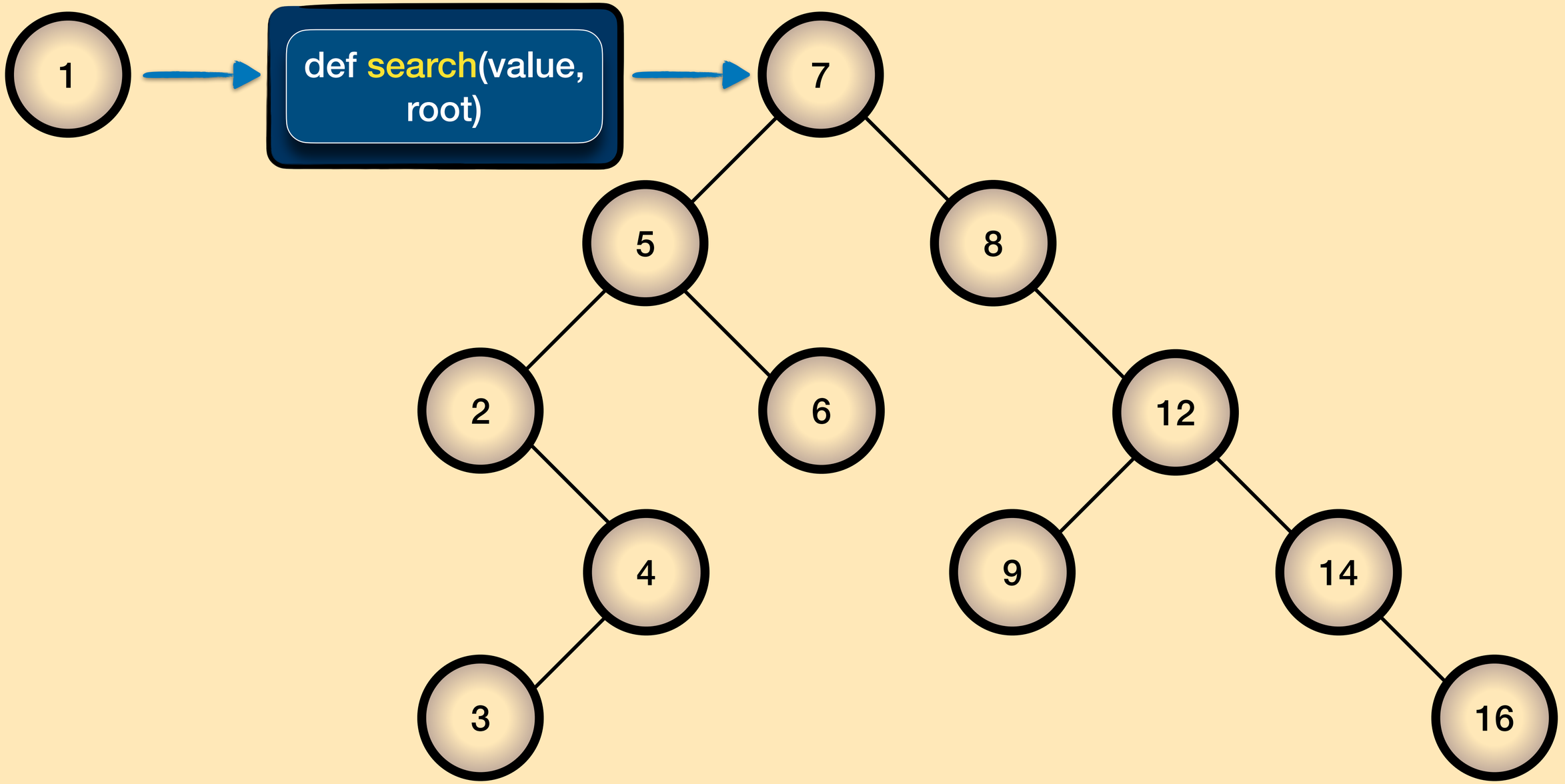
Once again about BST: search



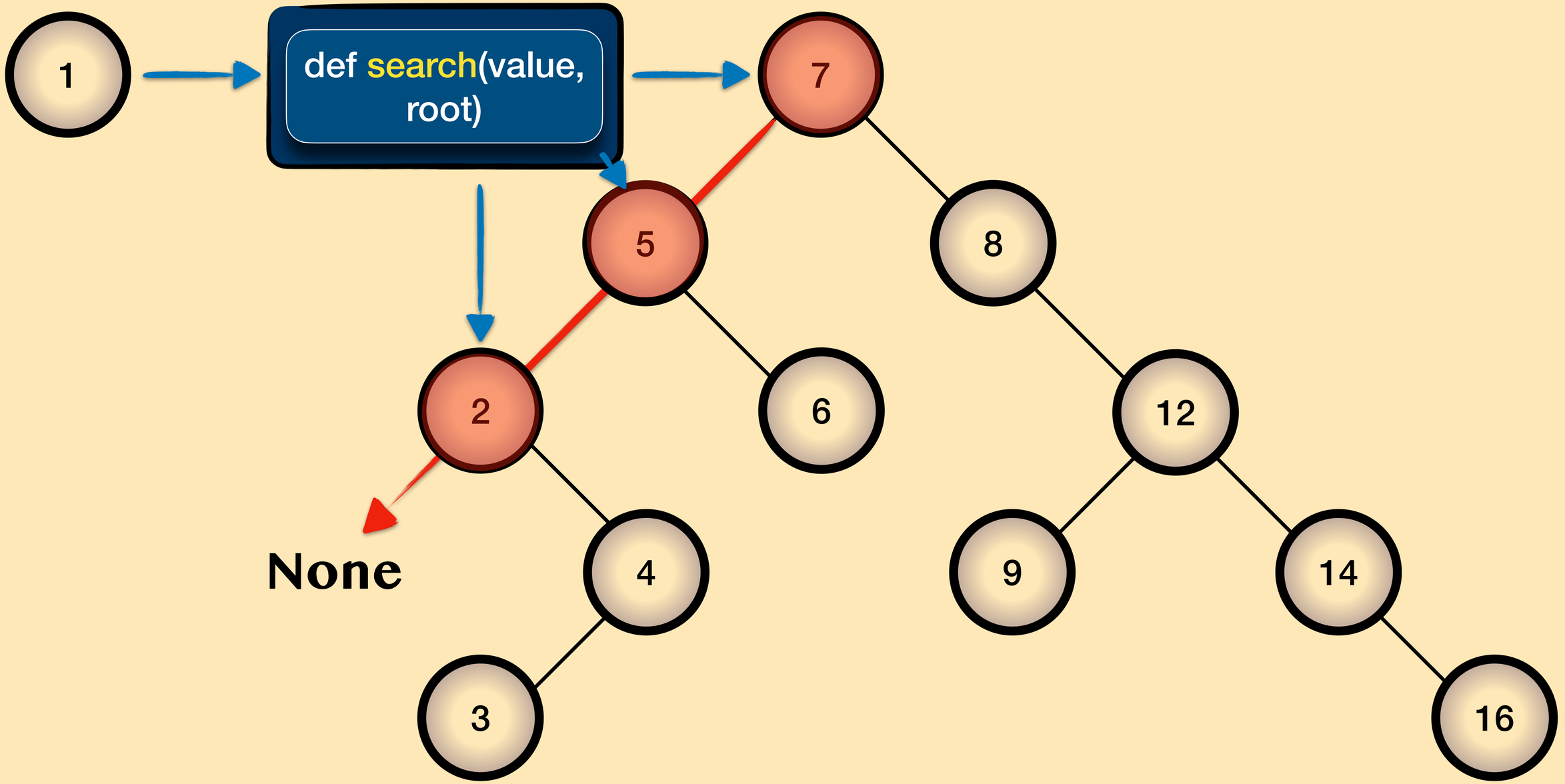
Once again about BST: search



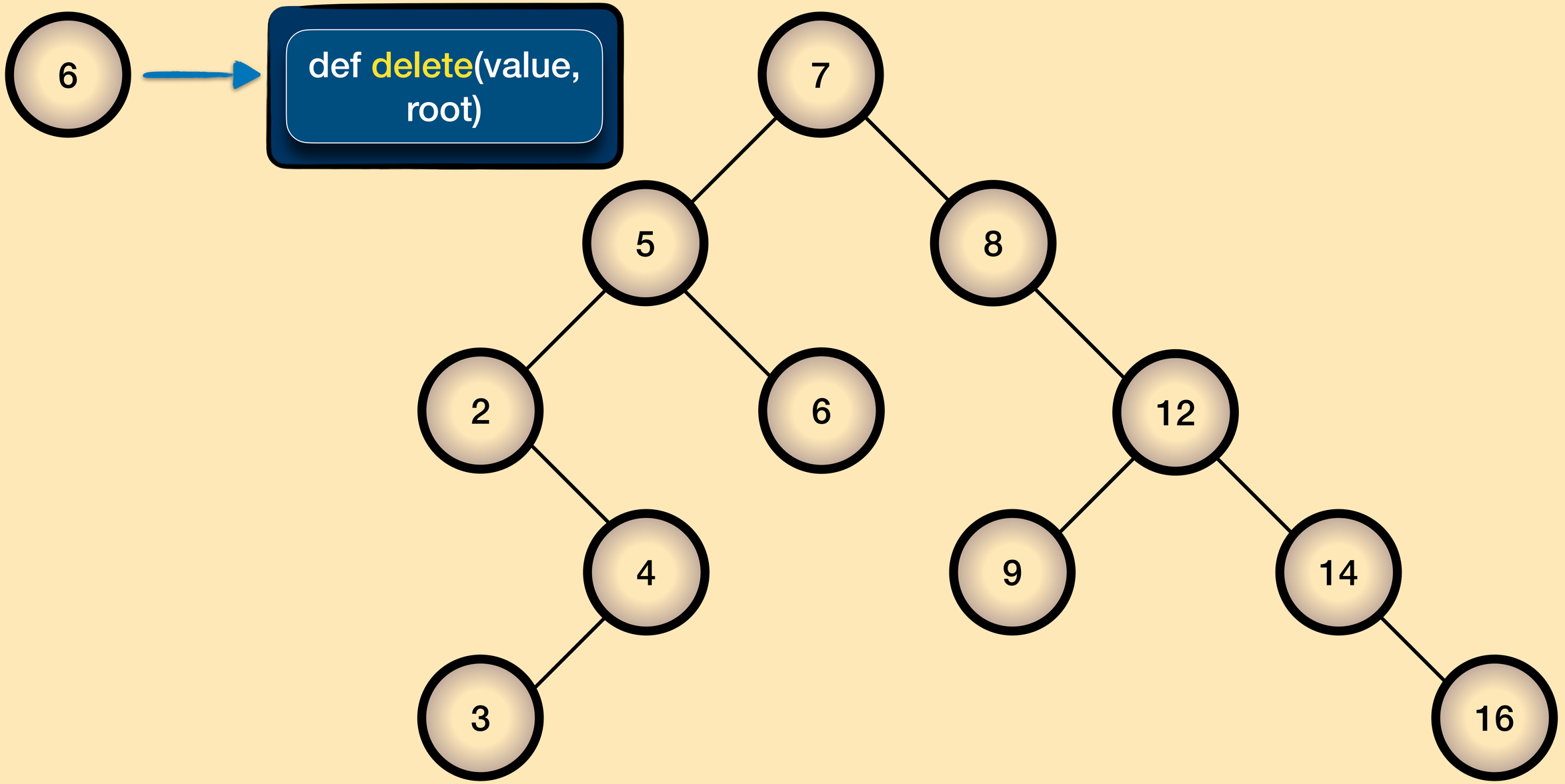
Once again about BST: search



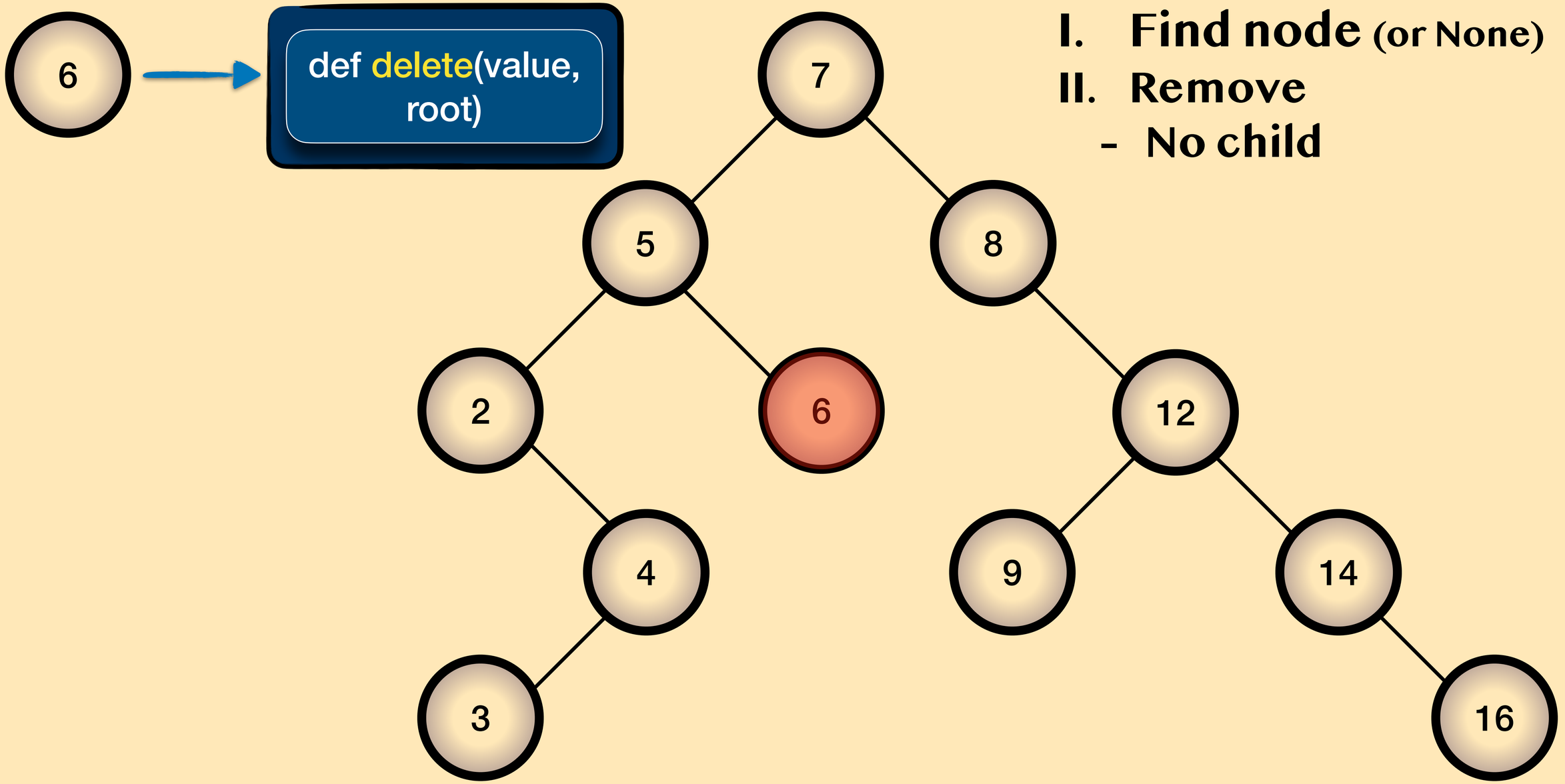
Once again about BST: search



Once again about BST: delete

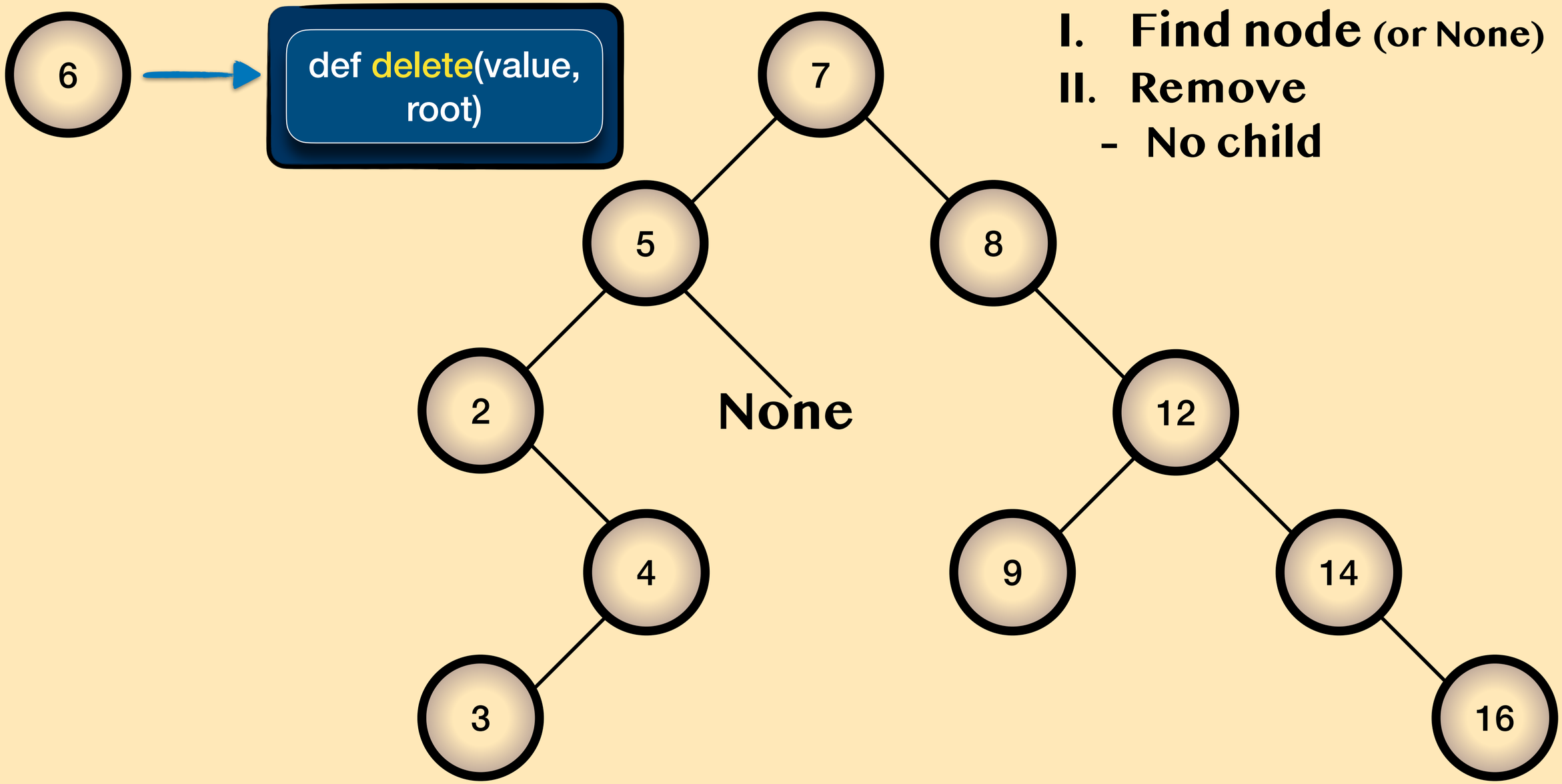


Once again about BST: delete



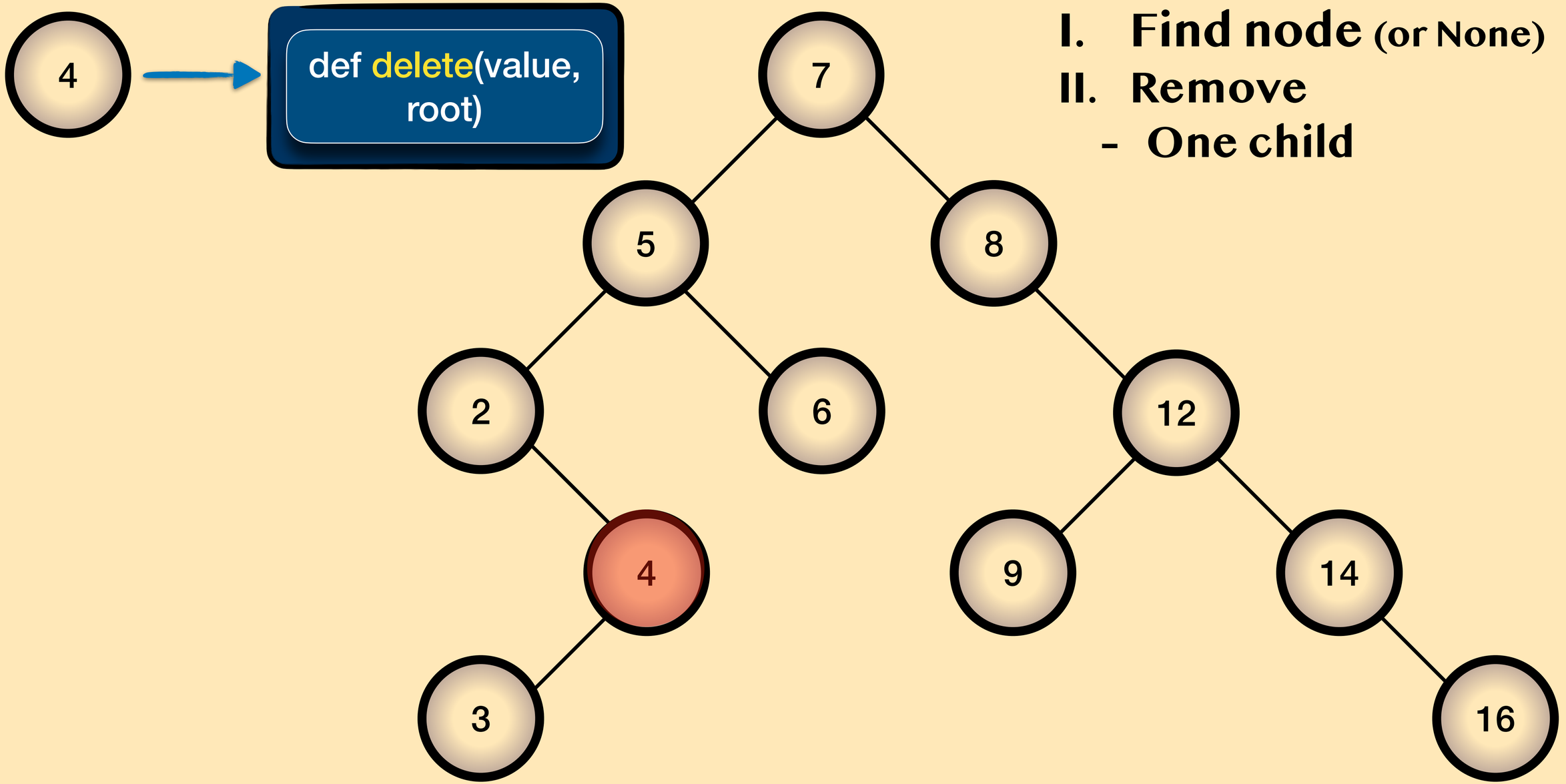
I. **Find node** (or None)
II. **Remove**
   - **No child**

Once again about BST: delete



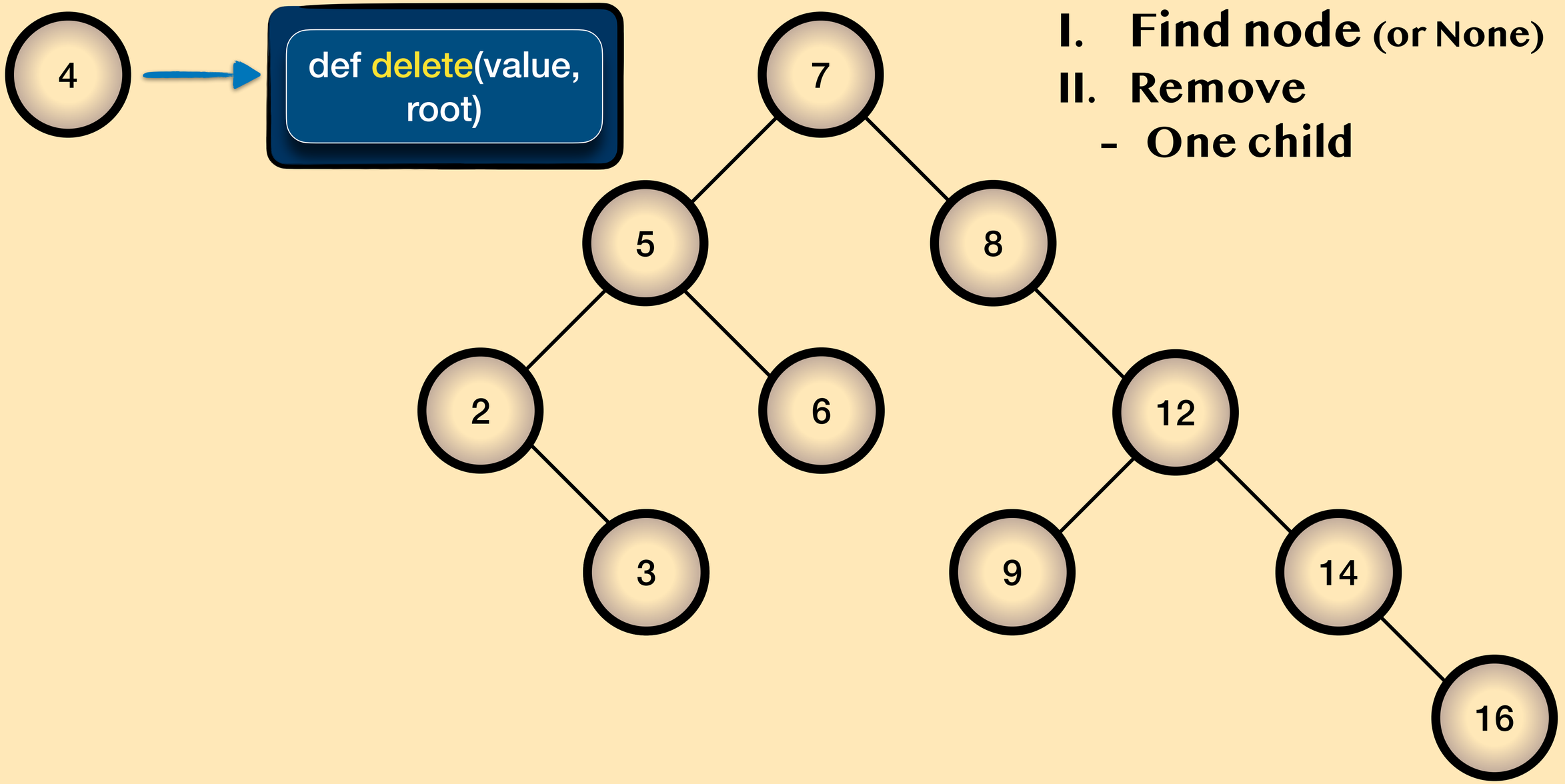
I. Find node (or None)
II. Remove
   - No child

Once again about BST: delete



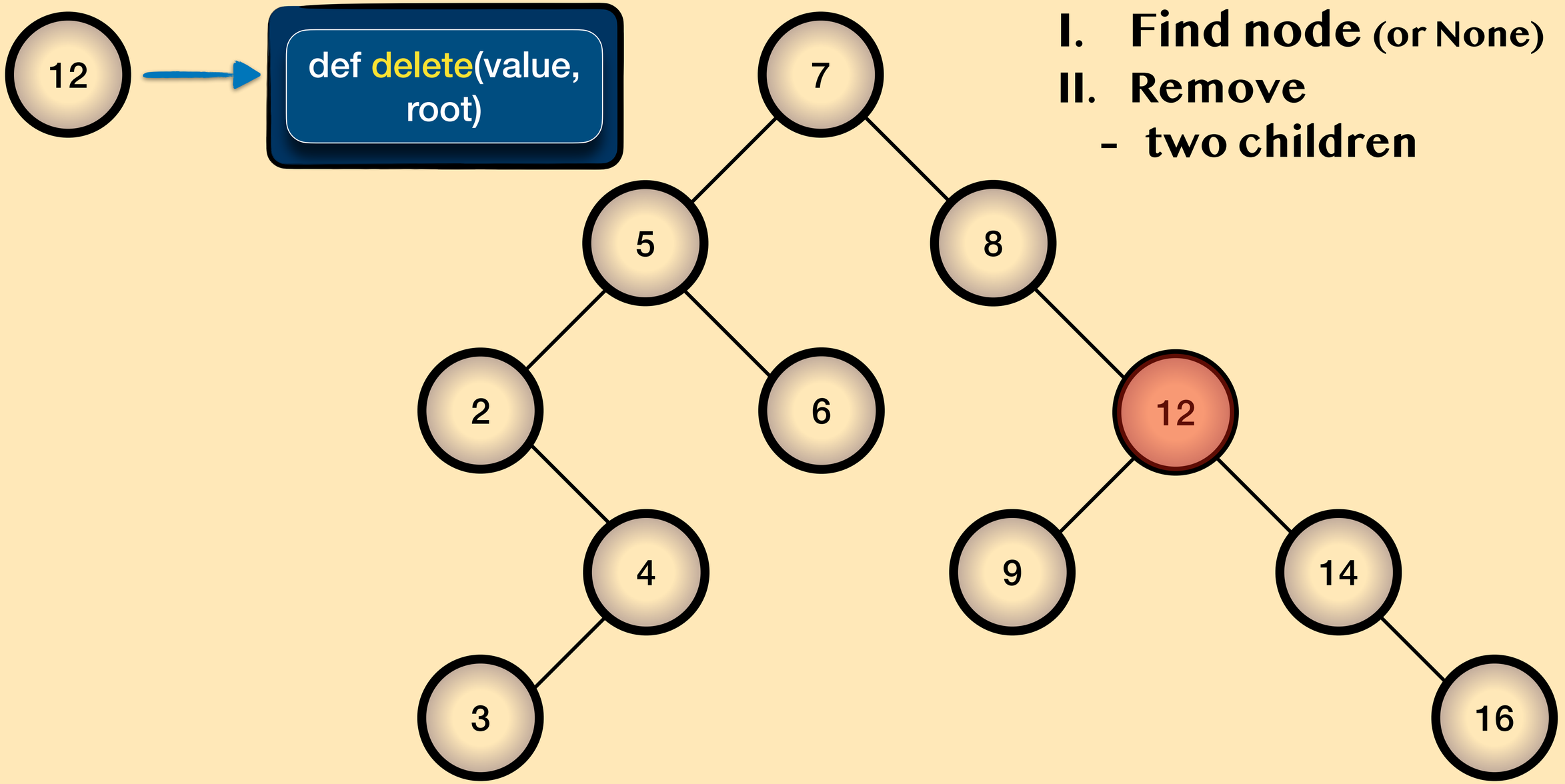
I. Find node (or None)
II. Remove
   - One child

Once again about BST: delete



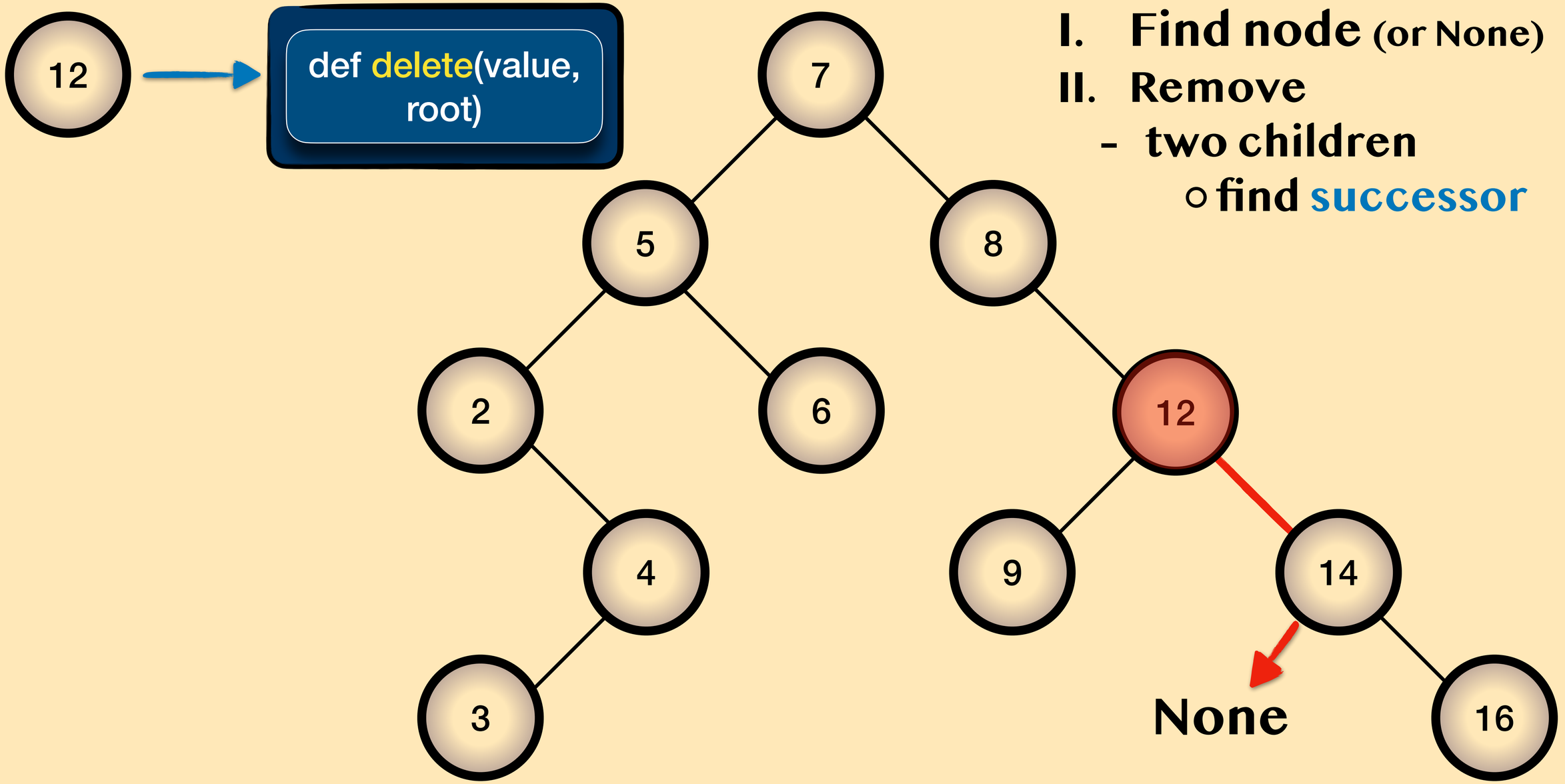
I. Find node (or None)
II. Remove
   - One child

Once again about BST: delete



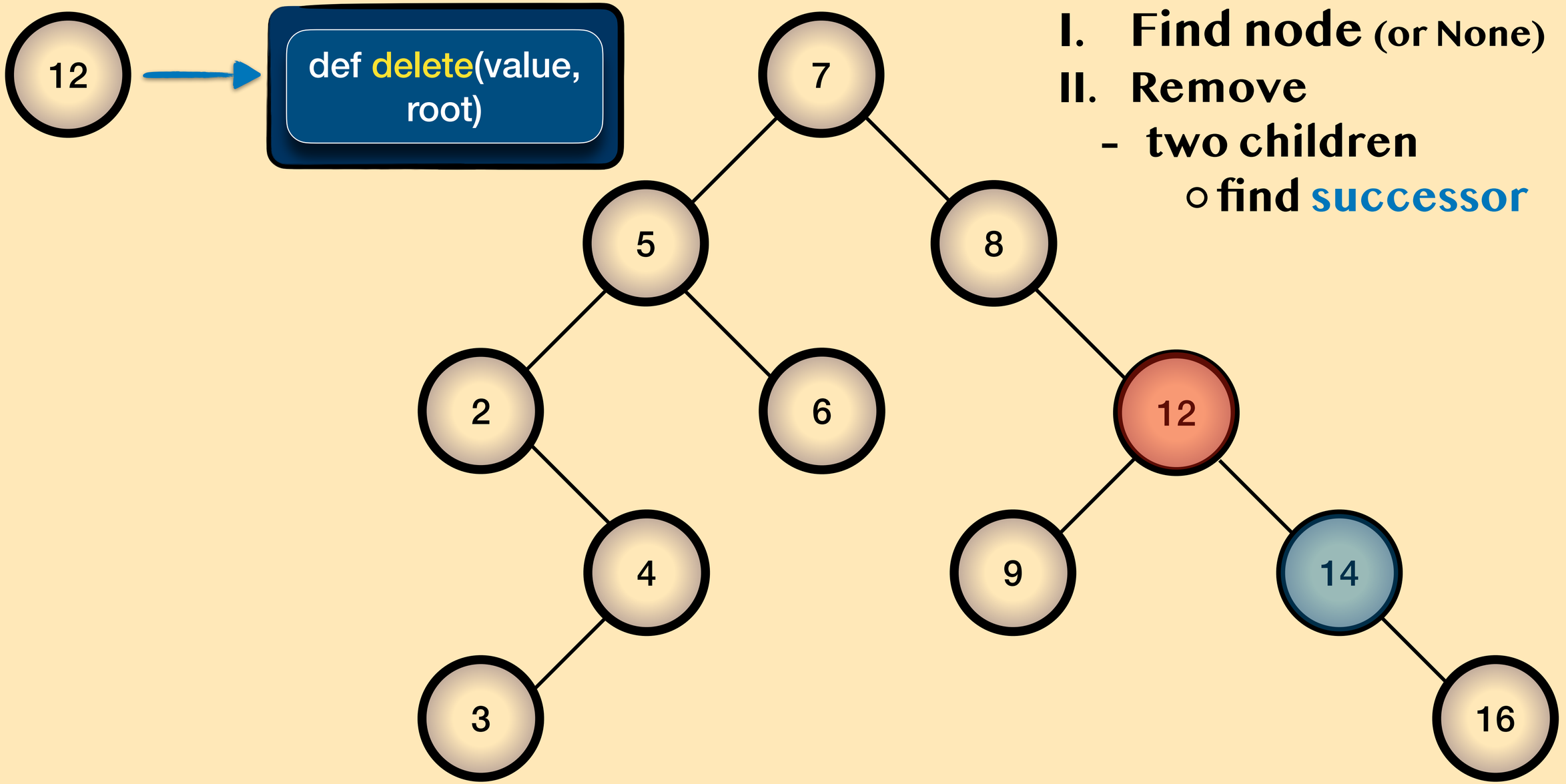
I. Find node (or None)
II. Remove
- two children

Once again about BST: delete



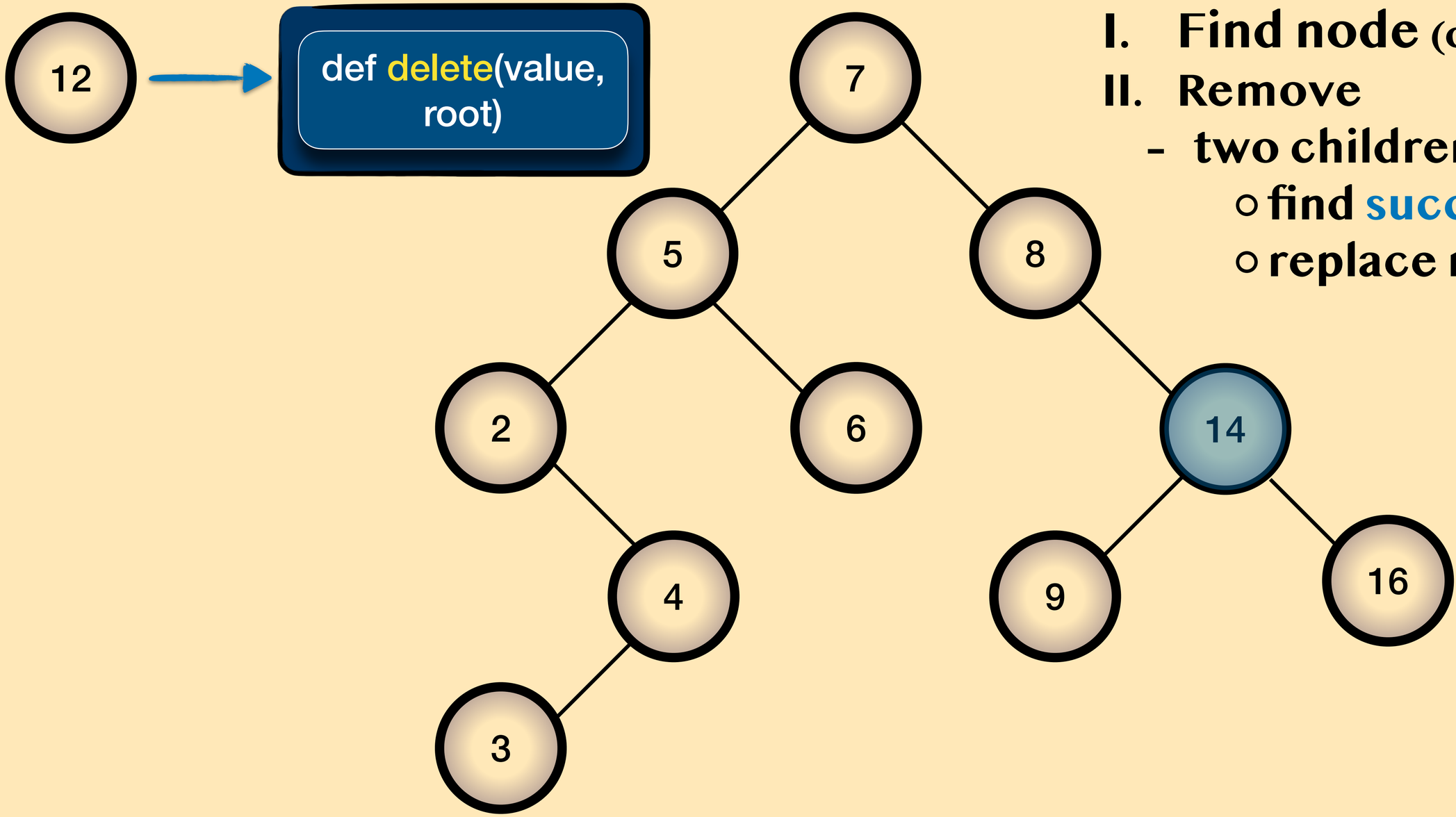
I. Find node (or None)
II. Remove
- two children
  - find successor

Once again about BST: delete



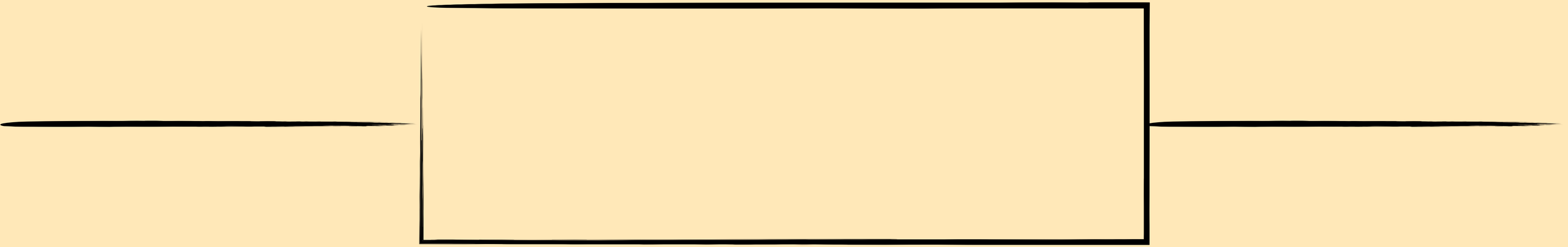
I. Find node (or None)
II. Remove
- two children
  - find successor

Once again about BST: delete



I. Find node (or None)
II. Remove
- two children
  - find successor
  - replace node

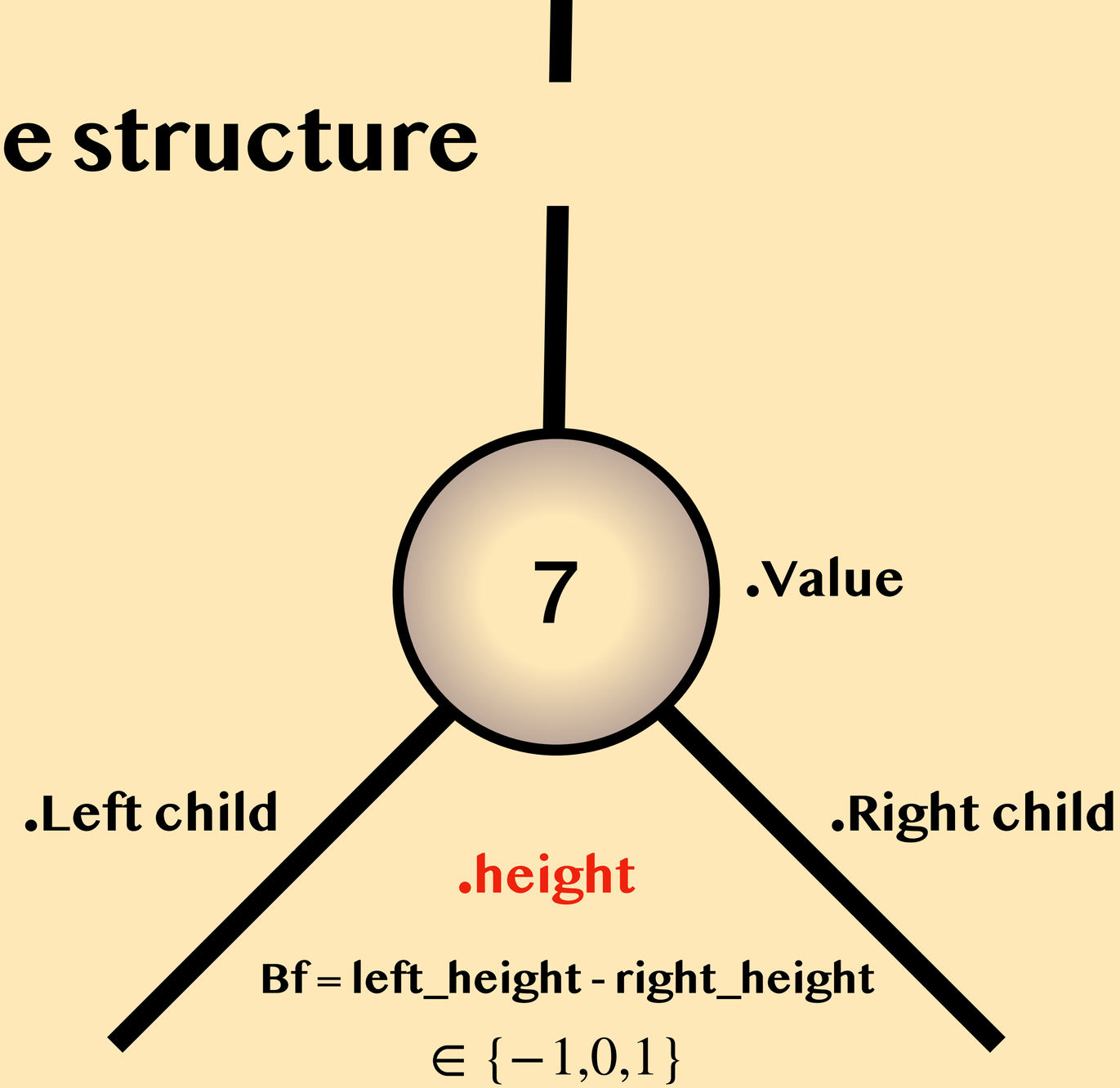
Self-balancing Tree



Self-balancing Tree

Heights of right & left subtrees of each node differ by at most one

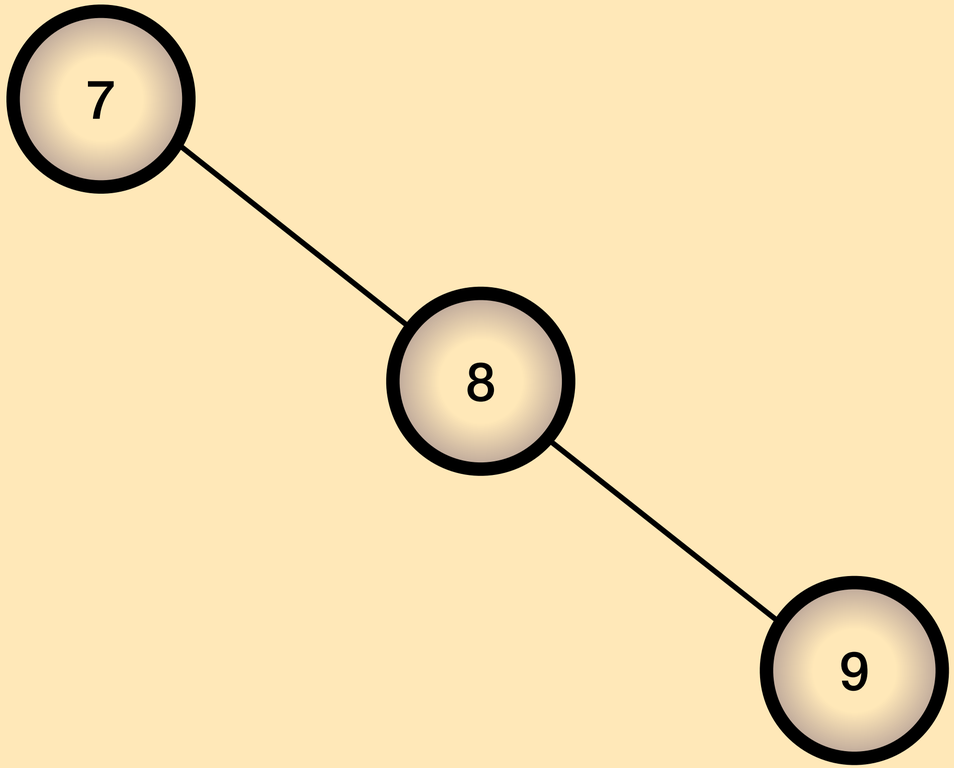
AVL: node structure



AVL: rotation cases

Basic

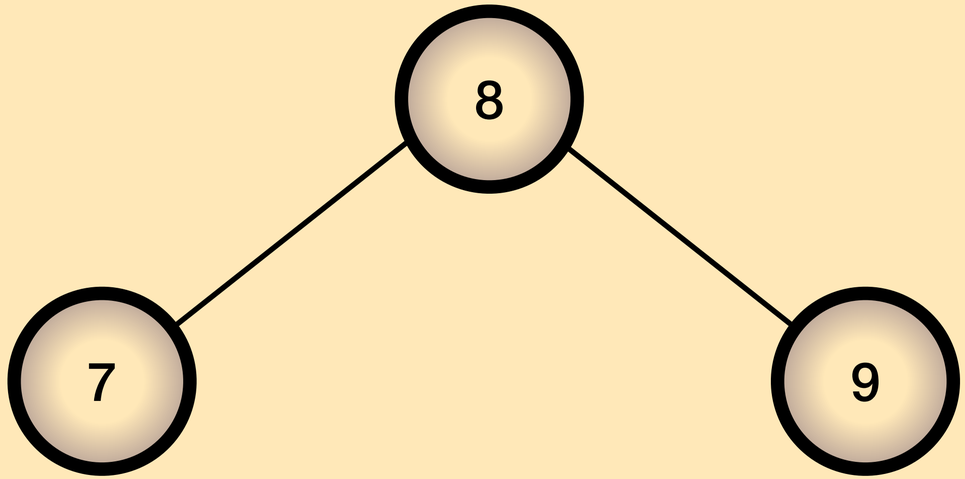
Rotation to left (LL)



AVL: rotation cases

Basic

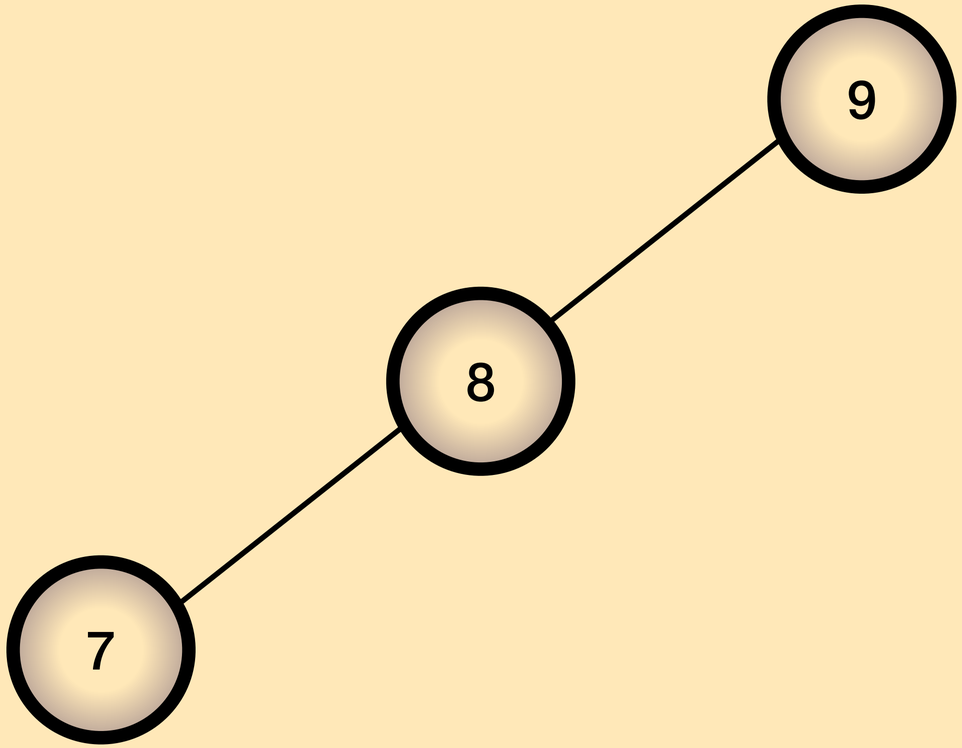
Rotation to left (LL)



AVL: rotation cases

Basic

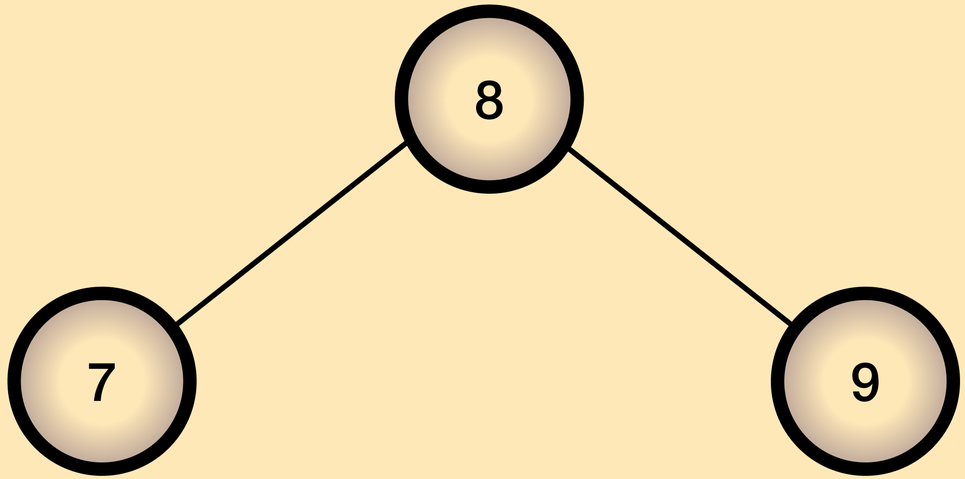
Rotation to right (RR)



AVL: rotation cases

Basic

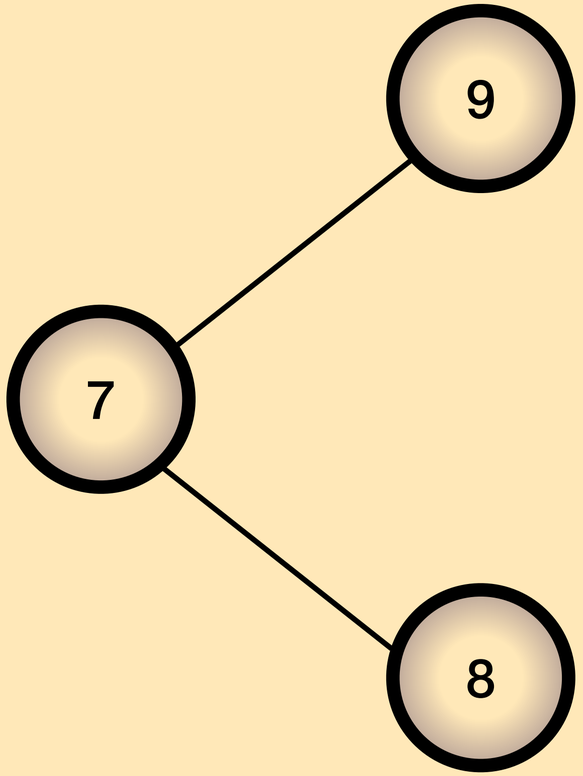
Rotation to right (RR)



AVL: rotation cases

Advanced

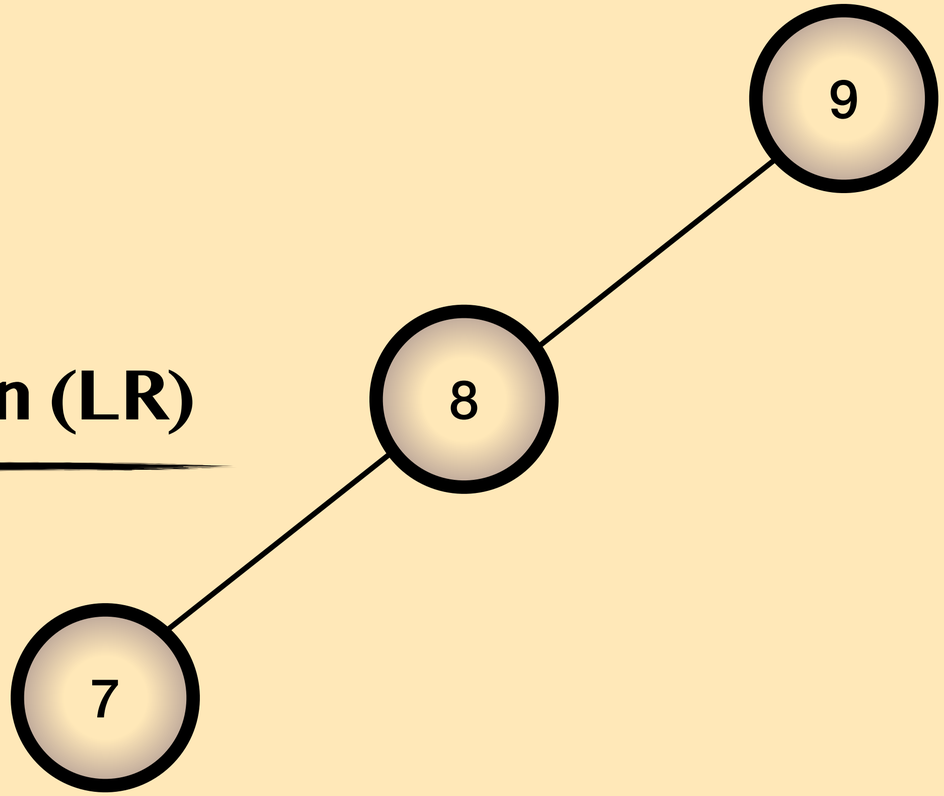
Left right rotation (LR)



AVL: rotation cases

Advanced

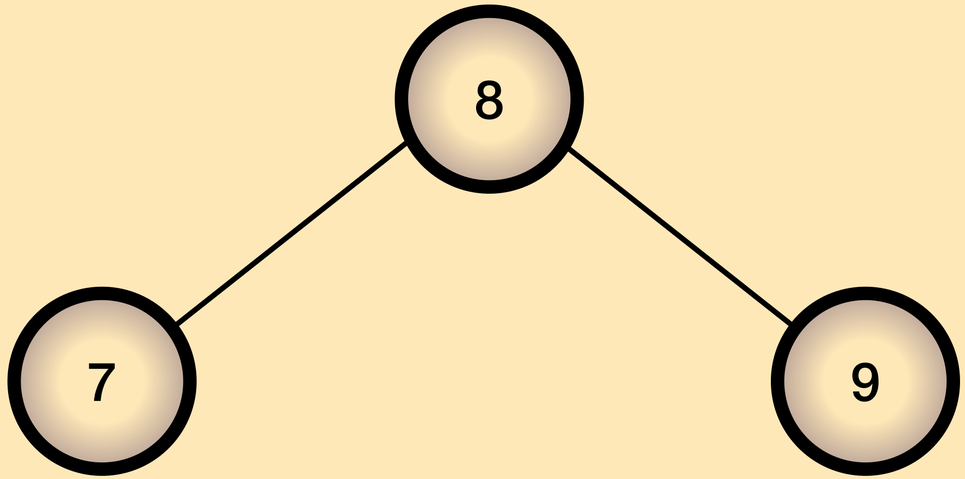
Left right rotation (LR)



AVL: rotation cases

Advanced

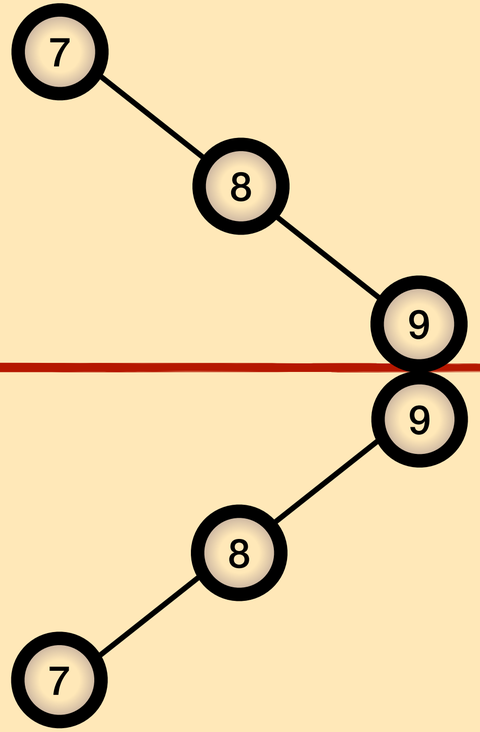
Left right rotation (LR)

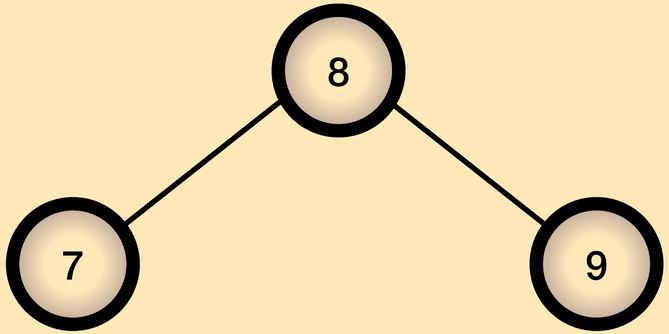


AVL: rotation cases

Inference

Rotation to left (LL)





Rotation to right (RR)

Left right rotation (LR)

Right left rotation (RL)

AVL: rotation cases

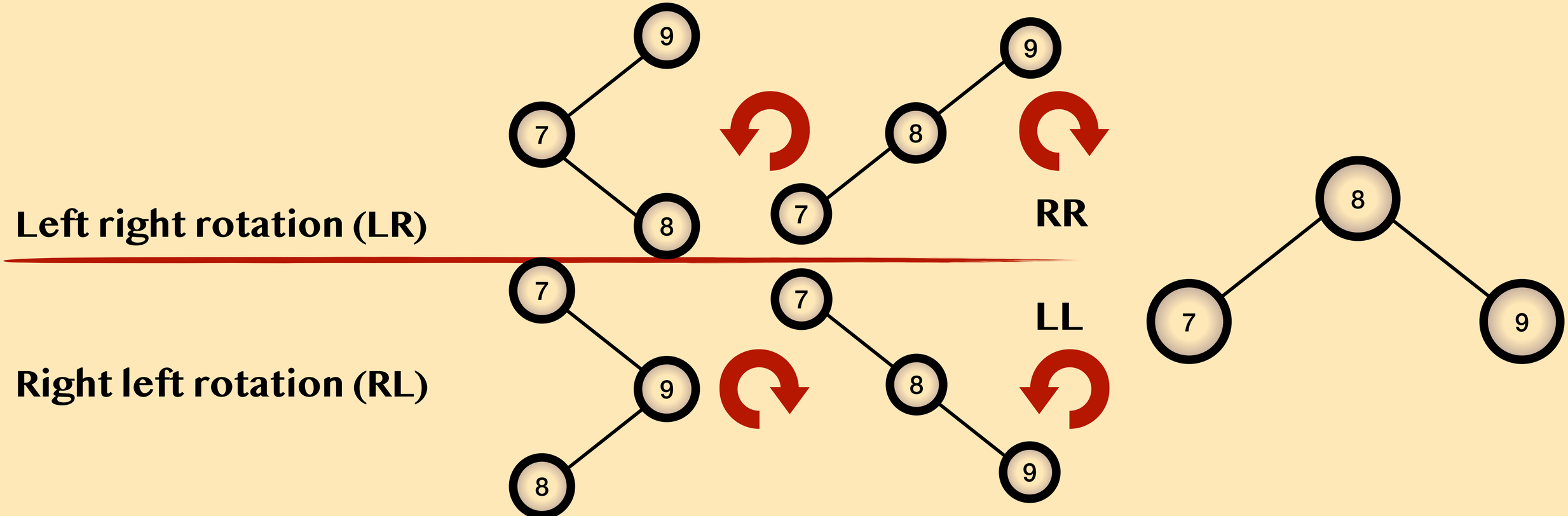
Inference

Rotation to left (LL)

Rotation to right (RR)

Left right rotation (LR)

Right left rotation (RL)



AVL: insert method

How to code?

1. **Insert the node**
2. **Check for BF complicity**

```
computeHeightFromChildren(current)    \\update current's height

leftH = current→left→height
rightH = current→right→height

if leftH > rightH + 1 then            \\Left subtree is too tall
    leftleftH = current→left→left→height
    leftrightH = current→left→right→height

    if leftleftH >= leftrightH then
        return rightRotate(current)          \\left-outer grandchild is taller
    else
        return leftRightRotate(current)      \\left-inner grandchild is taller
    end if
end if

if rightH > leftH + 1 then            \\Right subtree is too tall
    rightleftH = current→right→left→height
    rightrightH = current→right→right→height

    if rightrightH >= rightleftH then
        return leftRotate(current)           \\right-outer grandchild is taller
    else
        return rightLeftRotate(current)      \\right-inner grandchild is taller
    end if
end if

return current    \\No rotation, so root is the same
end function
```