AMRITA VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

19CSE212

DATA STRUCTURES & ALGORITHMS

Unit 2

Trees: Tree Definition and Properties – Tree ADT - Basic tree traversals - Binary tree - Data structure for representing trees – Linked Structure for Binary Tree – Array based implementation. Priority queues: ADT – Implementing Priority Queue using List – Heaps. Maps and Dictionaries: Map ADT – List based Implementation – Hash Tables - Dictionary ADT. Skip Lists - Implementation - Complexity. -**BST**

Course Outcome:

Course Outcome's	BTL
CO1, CO2, CO3, CO4 and CO5	1,2,3,4

J.UMA, AP-CSE

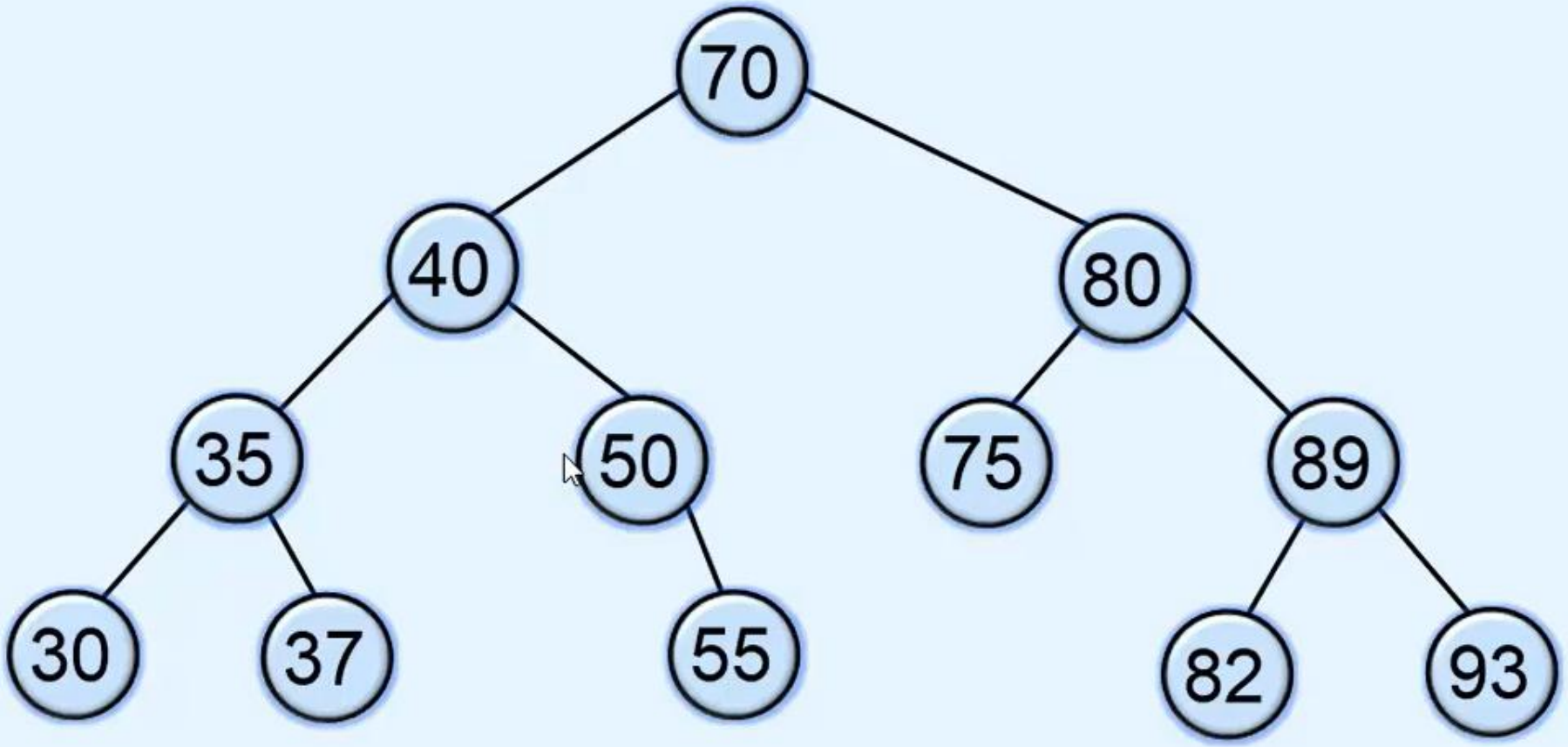
Amrita School of Computing

Binary Search Trees

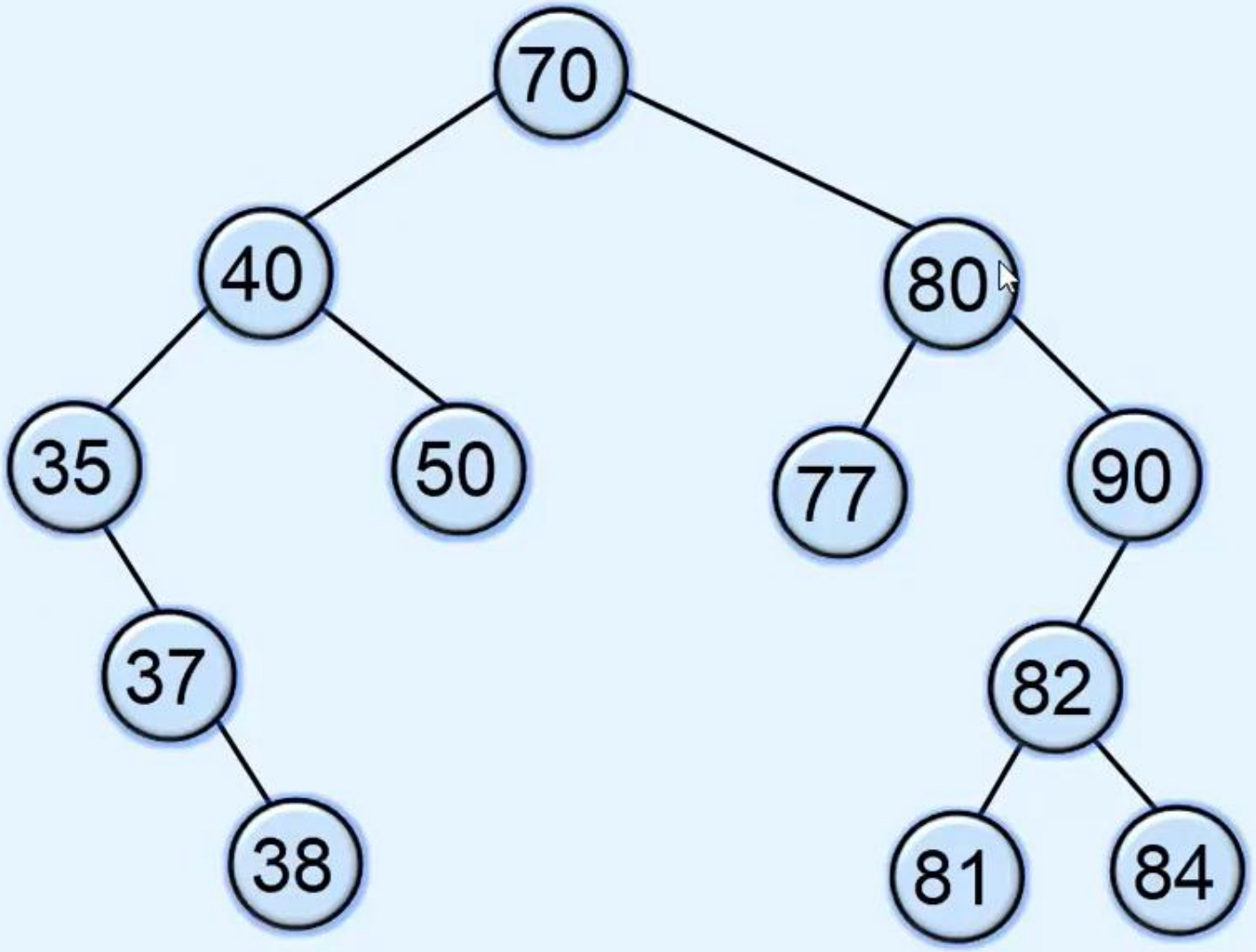
A binary tree that is either empty or has the following properties-

1. All the keys in left subtree of root are less than the key in the root
2. All the keys in right subtree of root are greater than the key in the root
3. Left and right subtrees of root are also binary search trees

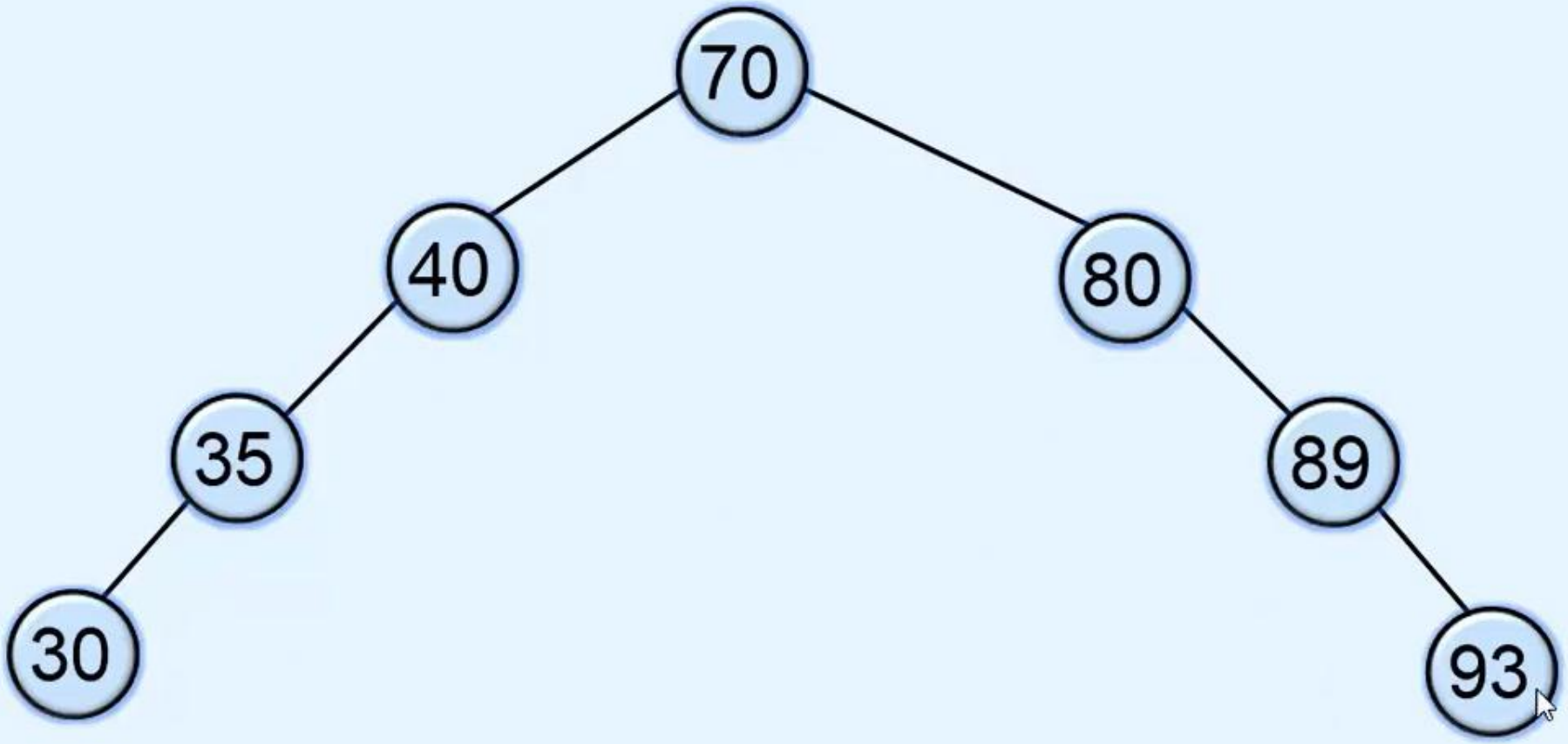
Binary Search Trees



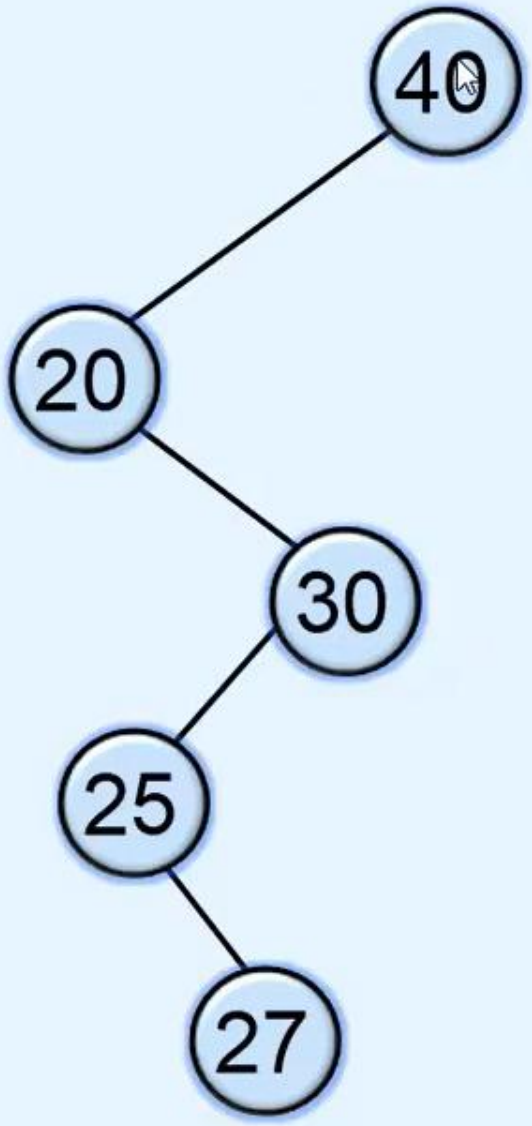
Binary Search Trees



Binary Search Trees



Binary Search Trees



Searching in a Binary Search Tree

x is the key to be searched

Start at the root node and move down the tree

- If x is equal to the key in the current node — Search is successful

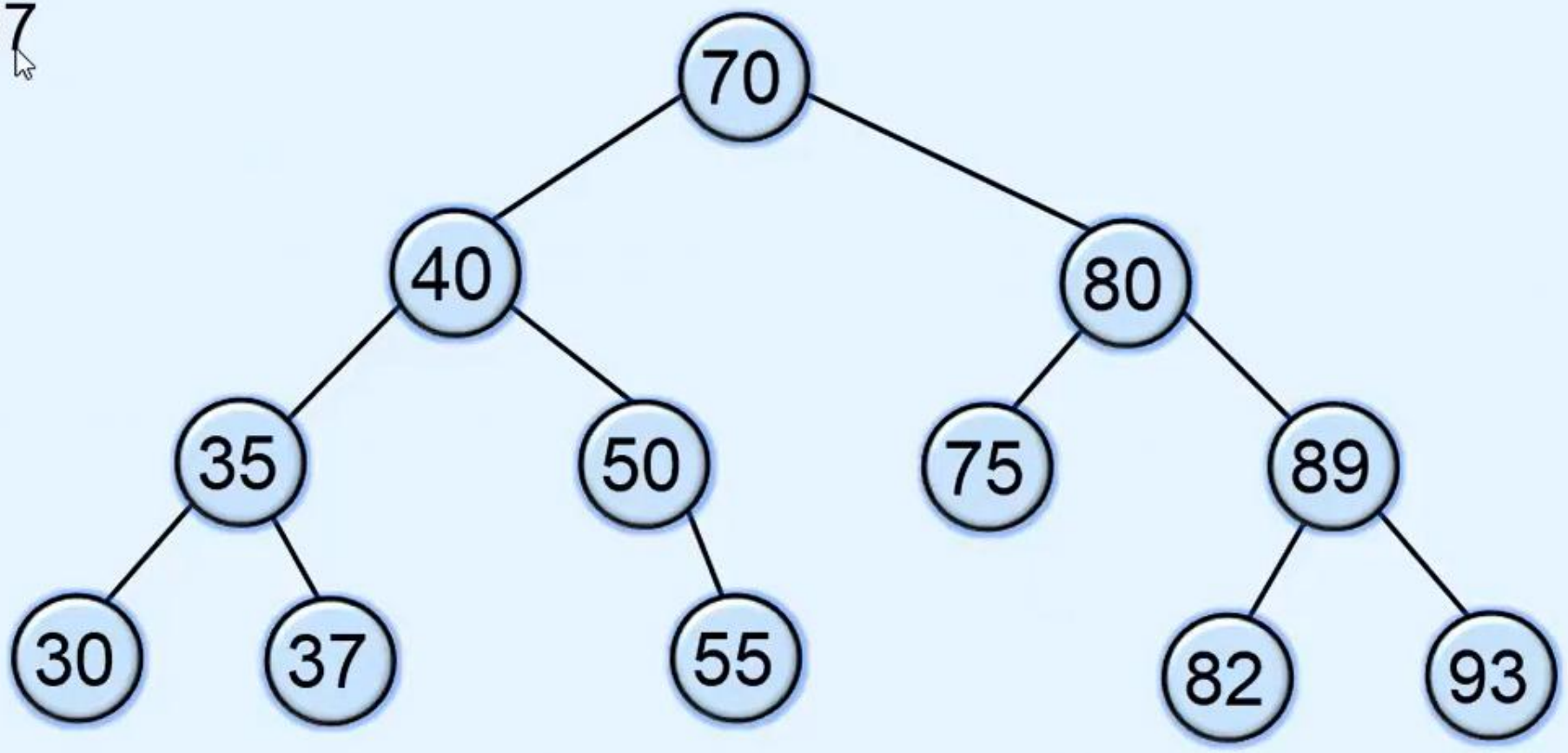
- If x is less than the key in the current node — Move to left child

- If x is greater than the key in the current node — Move to right child

If we reach a None left child or None right child — Search is unsuccessful

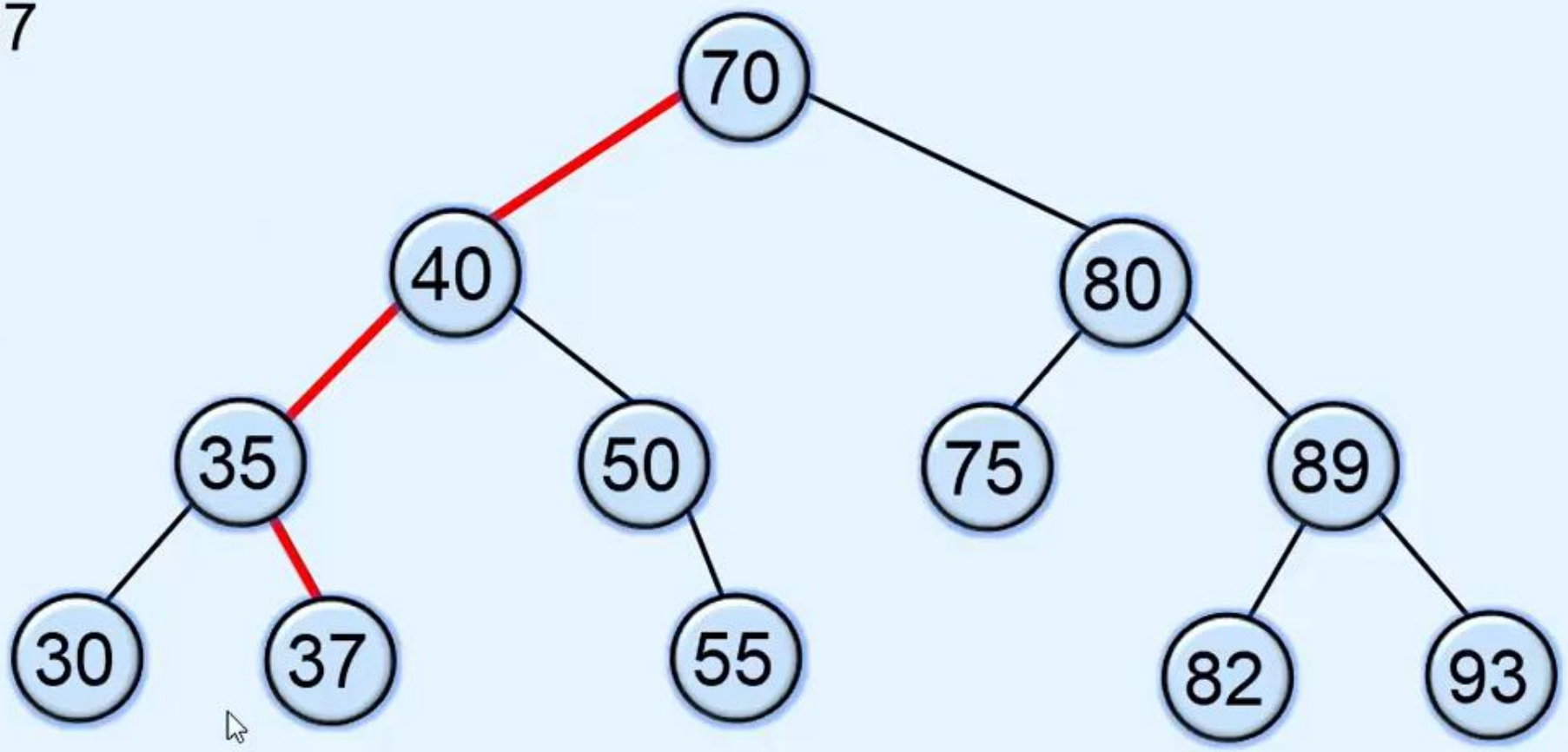
Searching in a Binary Search Tree

Search 37



Searching in a Binary Search Tree

Search 37



Searching in a Binary Search Tree

Searching is more efficient than in linked list

Run time is O(h), where h is the height of the tree

```
def _search(self, p, x):
    if p is None:
        return None  # key not found
    if x < p.info: # search in left subtree
        return self._search(p.lchild, x)
    if x > p.info: # search in right subtree
        return self._search(p.rchild, x)
    return p  # key found
```

RECURSIVE

```
def search1(self,x):
    p = self.root
    while p is not None:
        if x < p.info :
            p = p.lchild  # Move to left child
        elif x > p.info:
            p = p.rchild  # Move to right child
        else:   # x found
            return true
    return false
```

ITERATIVE

```
bst.search(45)
```

```
def search(self,x):
    return self._search(self.root, x) is not None
```

54

```
def _search(self, p, x):
    if p is None:
        return None
    if x < p.info:
        return self._search(p.lchild, x)
    if x > p.info:
        return self._search(p.rchild, x)
    return p
```

42

```
def _search(self, p, x):
    if p is None:
        return None
    if x < p.info:
        return self._search(p.lchild, x)
    if x > p.info:
        return self._search(p.rchild, x)
    return p
```

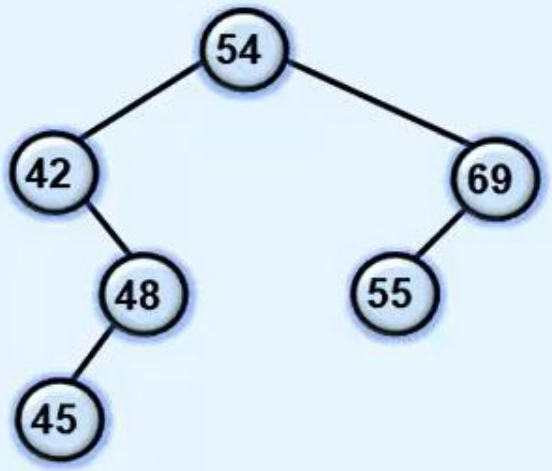
48

```
def _search(self, p, x):
    if p is None:
        return None
    if x < p.info:
        return self._search(p.lchild, x)
    if x > p.info:
        return self._search(p.rchild, x)
    return p
```

45

```
def _search(self, p, x):
    if p is None:
        return None
    if x < p.info:
        return self._search(p.lchild, x)
    if x > p.info:
        return self._search(p.rchild, x)
    return p
```

Search 45



```
bst.search(45)
```

true

```
def search(self,x):
    return self._search(self.root, x) is not None
```

Reference to node with key 45

```
def _search(self, p, x):
    if p is None:
        return None
    if x < p.info:
        return self._search(p.lchild, x)
    if x > p.info:
        return self._search(p.rchild, x)
    return p
```

54

Reference to node with key 45

```
def _search(self, p, x):
    if p is None:
        return None
    if x < p.info:
        return self._search(p.lchild, x)
    if x > p.info:
        return self._search(p.rchild, x)
    return p
```

42

Reference to node with key 45

```
def _search(self, p, x):
    if p is None:
        return None
    if x < p.info:
        return self._search(p.lchild, x)
    if x > p.info:
        return self._search(p.rchild, x)
    return p
```

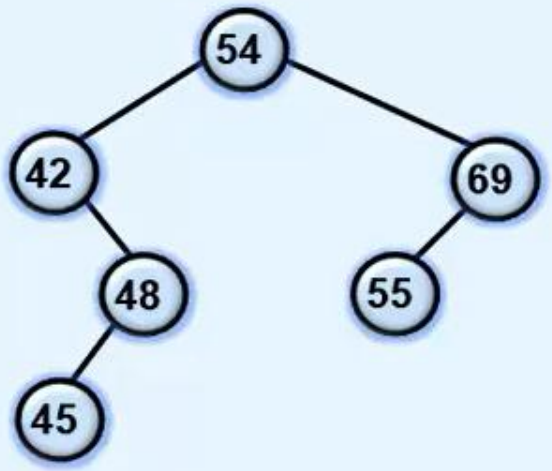
48

Reference to node with key 45

```
def _search(self, p, x):
    if p is None:
        return None
    if x < p.info:
        return self._search(p.lchild, x)
    if x > p.info:
        return self._search(p.rchild, x)
    return p
```

45

Search 45



```
bst.search(58)
```

false

```
def search(self,x):
    return self._search(self.root, x) is not None
```

None

```
def _search(self, p, x):
    if p is None:
        return None
    if x < p.info:
        return self._search(p.lchild, x)
    if x > p.info:
        return self._search(p.rchild, x)
    return p
```

54

None

```
def _search(self, p, x):
    if p is None:
        return None
    if x < p.info:
        return self._search(p.lchild, x)
    if x > p.info:
        return self._search(p.rchild, x)
    return p
```

69

None

```
def _search(self, p, x):
    if p is None:
        return None
    if x < p.info:
        return self._search(p.lchild, x)
    if x > p.info:
        return self._search(p.rchild, x)
    return p
```

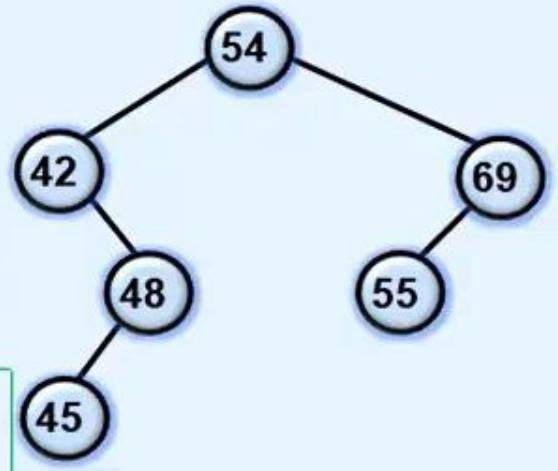
55

None

```
def _search(self, p, x):
    if p is None:
        return None
    if x < p.info:
        return self._search(p.lchild, x)
    if x > p.info:
        return self._search(p.rchild, x)
    return p
```

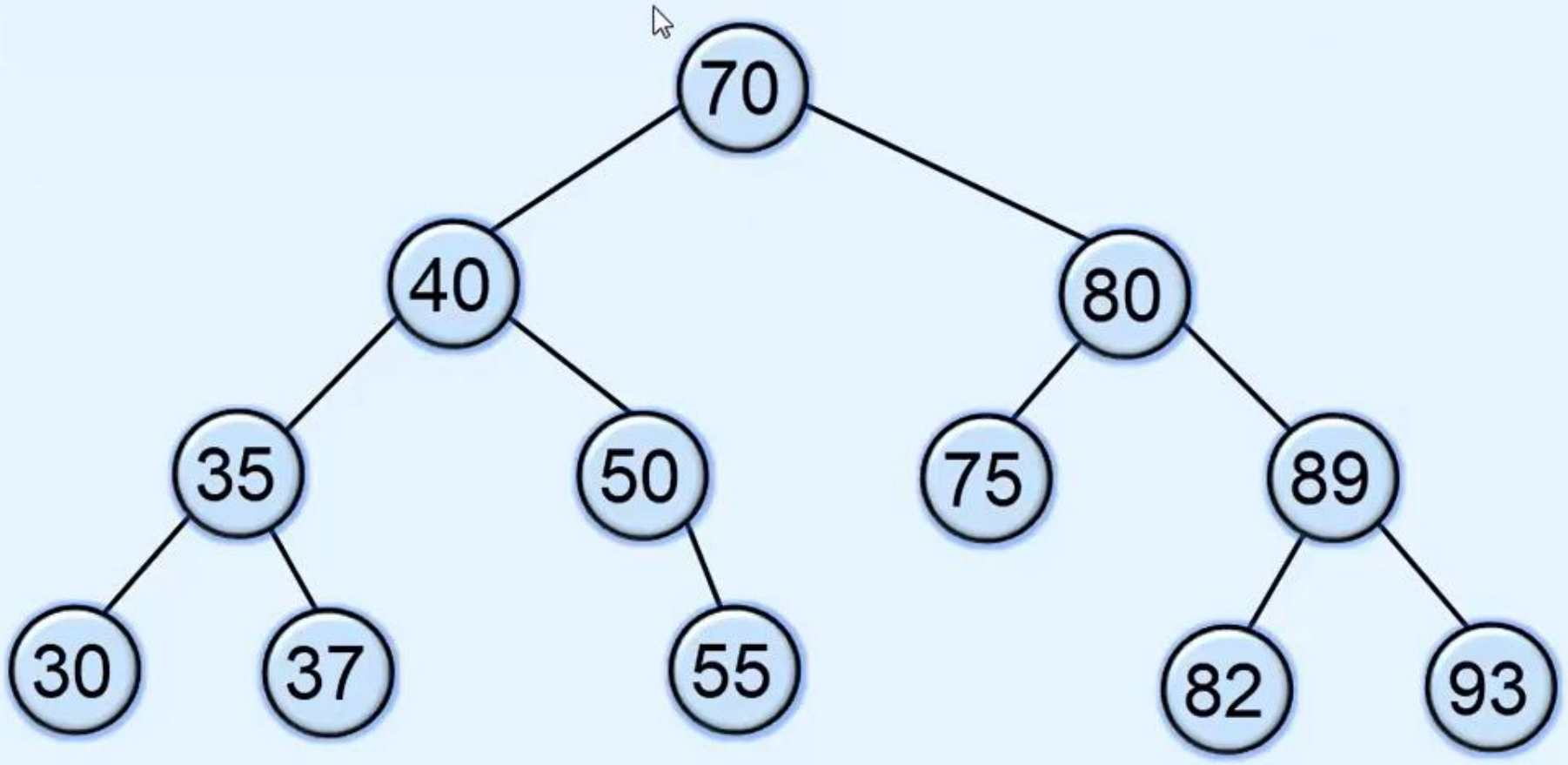
None

Search 58



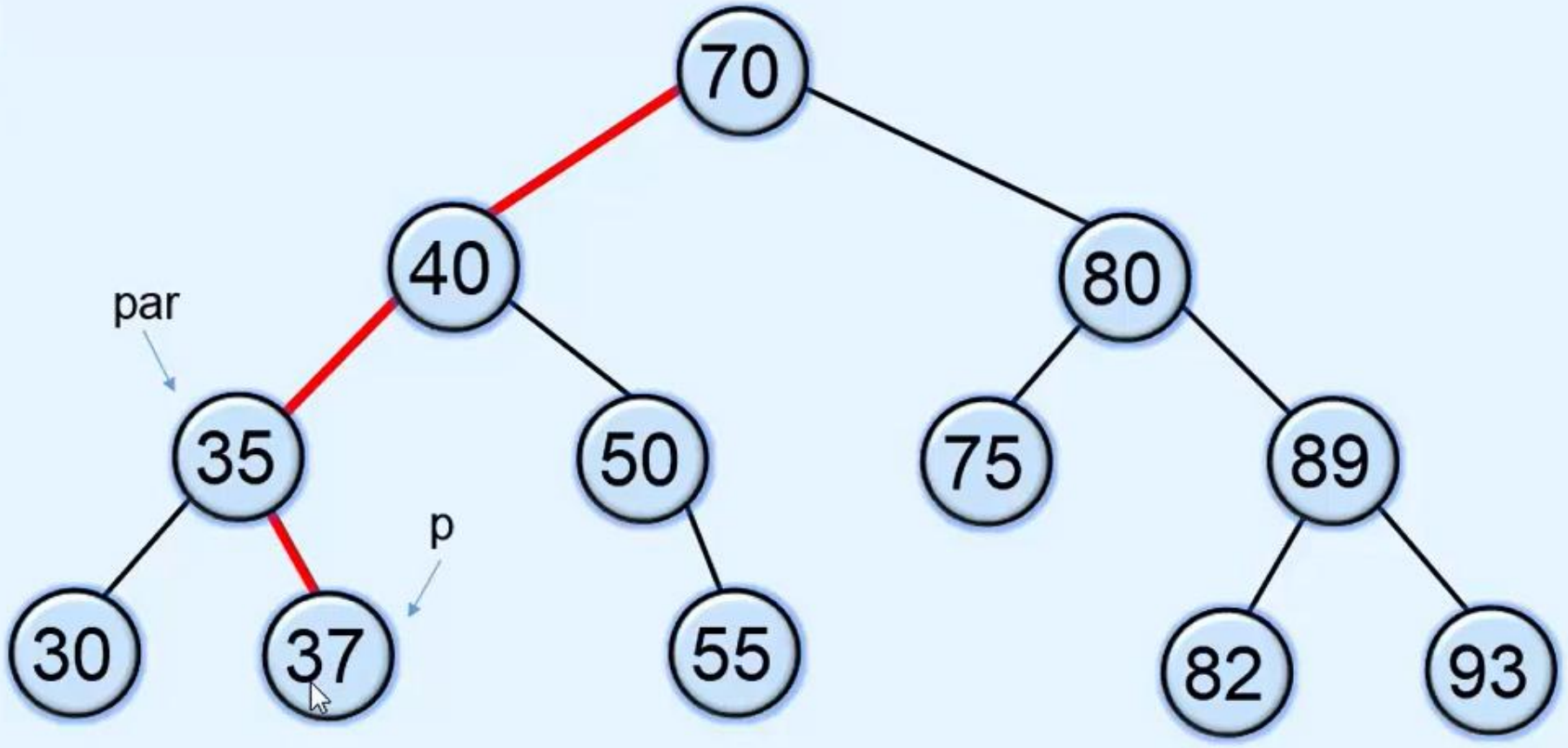
Insertion in a Binary Search Tree

Insert 36



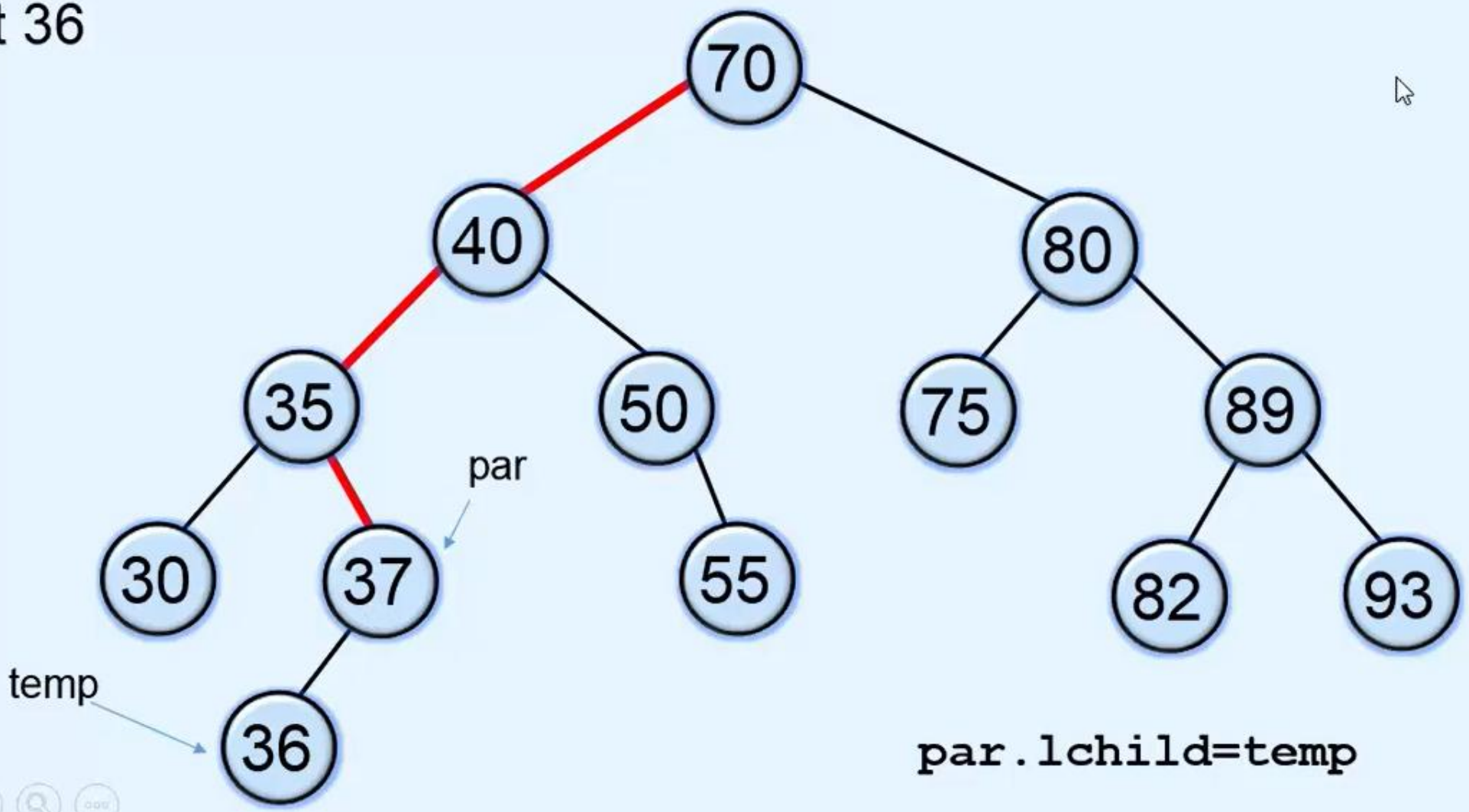
Insertion in a Binary Search Tree

Insert 36



Insertion in a Binary Search Tree

Insert 36




```
par.lchild=temp
```

```
        return p

    def insert1(self,x):
        p = self.root
        par = None
        while p is not None:
            par = p
            if x < p.info :
                p = p.lchild
            elif x > p.info:
                p = p.rchild
            else:
                print(x , " already present in the tree")
                return

        temp = Node(x)

        if par == None:
            self.root = temp
        elif x < par.info:
            par.lchild = temp
        else:
            par.rchild = temp
```

```
bst.insert(50)
```

```
def insert(self,x):
    self.root = self._insert(self.root, x)
```

```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

54

```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

42

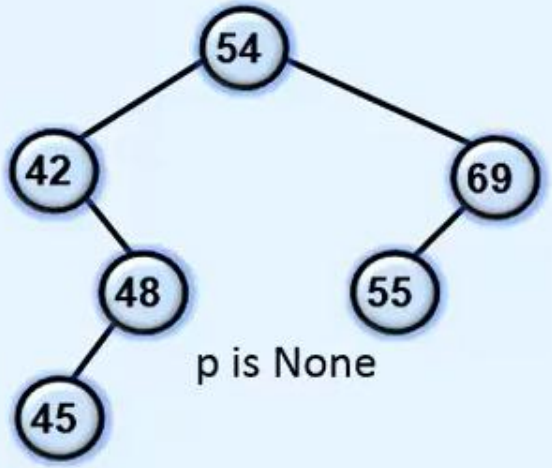
```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

48

```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

None

x = 50



```
bst.insert(50)
```

```
def insert(self,x):
    self.root = self._insert(self.root, x)
```

```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

54

```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

42

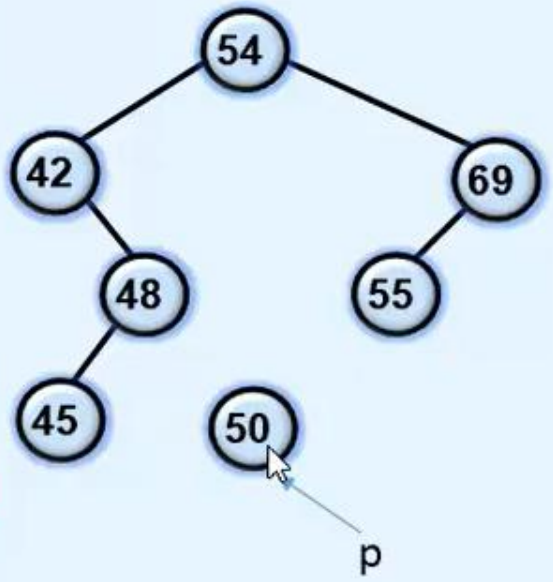
```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

48

```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

None

x = 50



```
bst.insert(50)
```

```
def insert(self,x):
    self.root = self._insert(self.root, x)
```

54

```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

42

```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

48

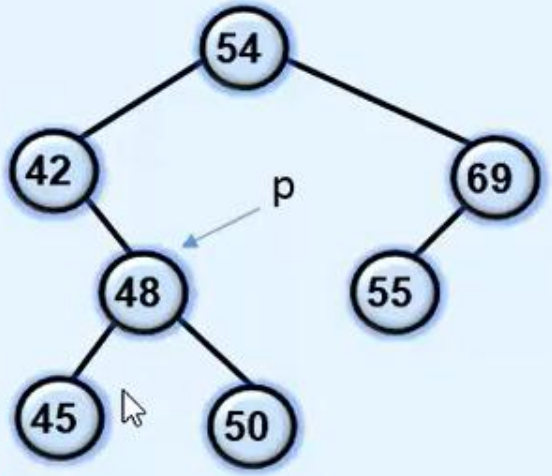
```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

None

```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

Reference to node with key 50

x = 50



```
bst.insert(50)
```

```
def insert(self,x):
    self.root = self._insert(self.root, x)
```

Reference to node with key 54

```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

54

Reference to node with key 42

```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

42

Reference to node with key 48

```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

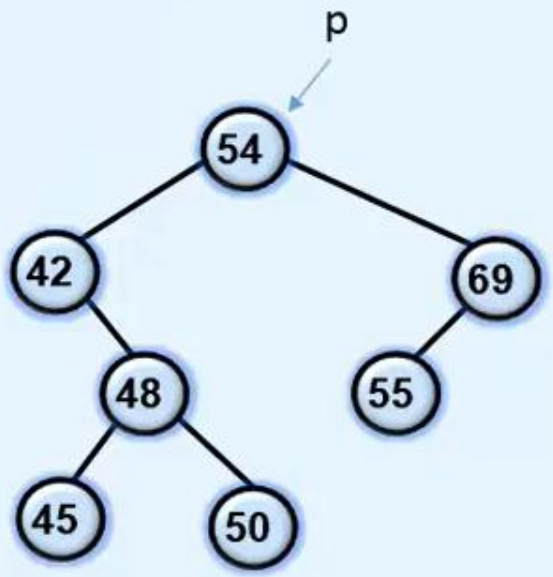
48

Reference to node with key 50

```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

None

x = 50



```
bst.insert(60)
```

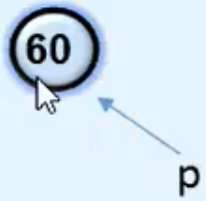
```
def insert(self,x):
    self.root = self._insert(self.root, x)
```

```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

None

x = 60

root is None



```
bst.insert(60)
```

```
def insert(self,x):
    self.root = self._insert(self.root, x)
```

Reference to node
with key 60

```
def _insert(self,p, x):
    if p is None:
        p = Node(x)
    elif x < p.info :
        p.lchild = self._insert(p.lchild, x)
    elif x > p.info :
        p.rchild = self._insert(p.rchild, x)
    else: print(x, " already present in the tree")
    return p
```

None

x = 60

root

Deletion in a Binary Search Tree

Case A : Node has no child, it is leaf node

Case B: Node has exactly 1 child

Case C: Node has exactly 2 children

Case A : Node is Leaf node

For deleting a leaf node N

↓

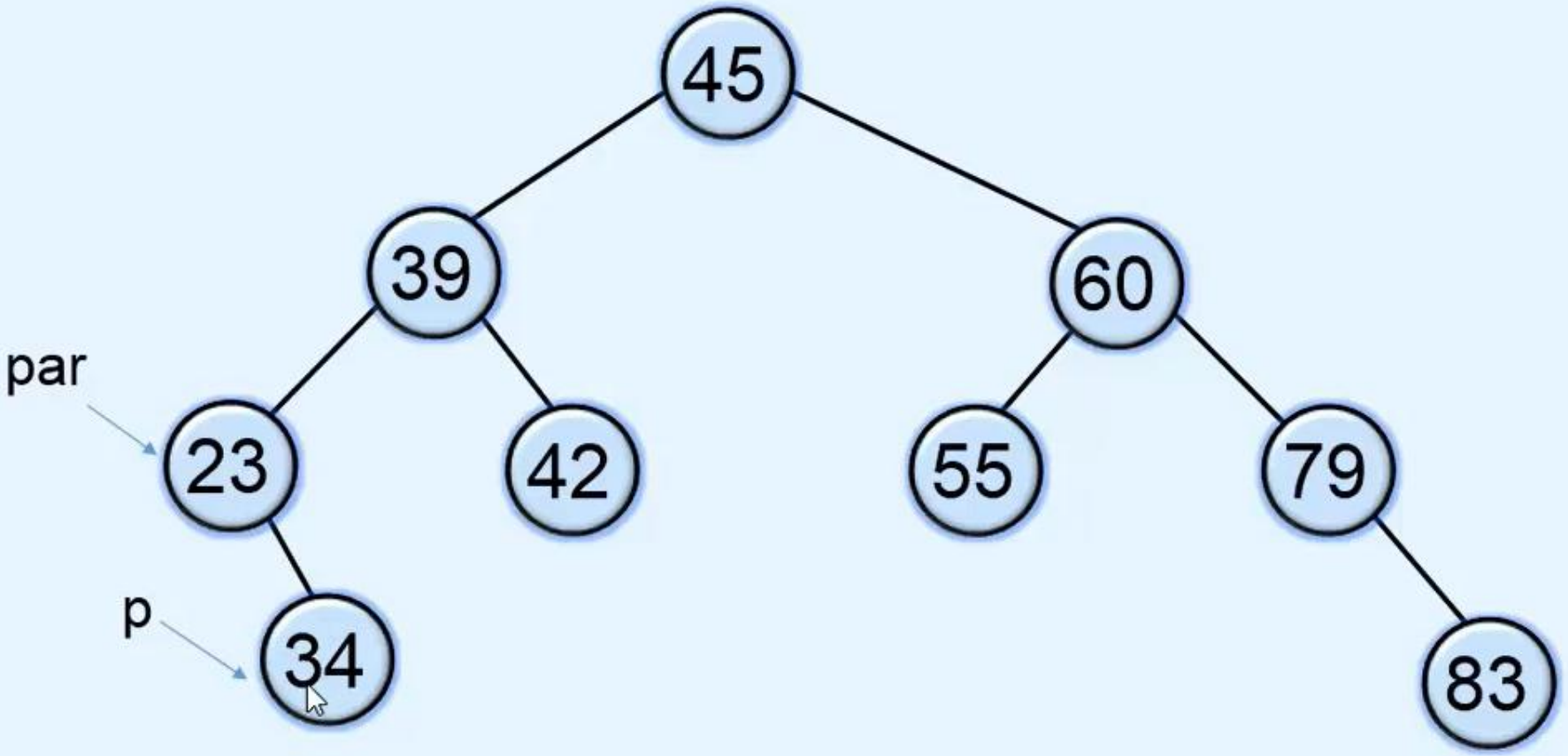
Link to node N in its parent is replaced by None

If N is left child → Left child of parent becomes None

If N is right child → Right child of parent becomes None

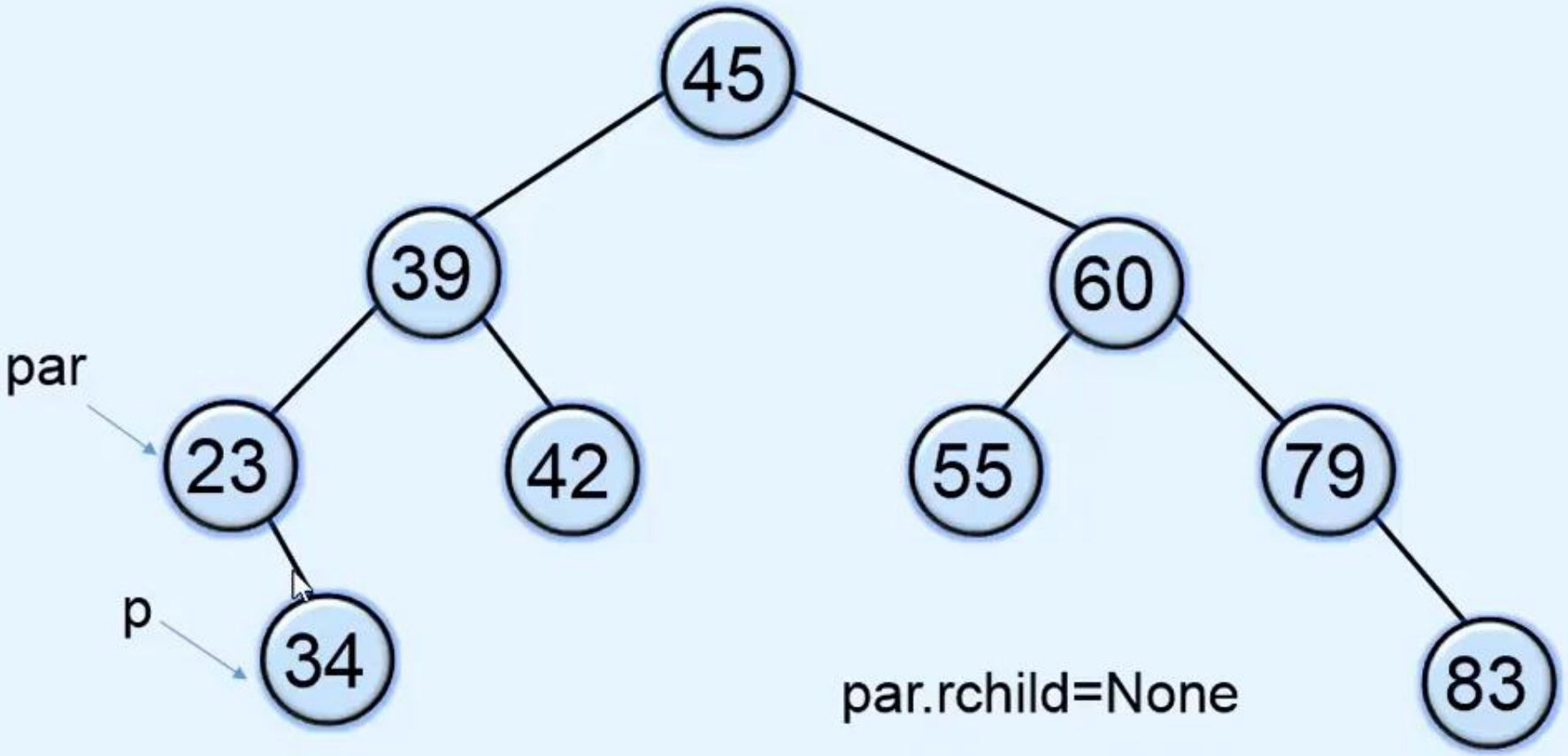
Case A : Node is Leaf node

Delete 34



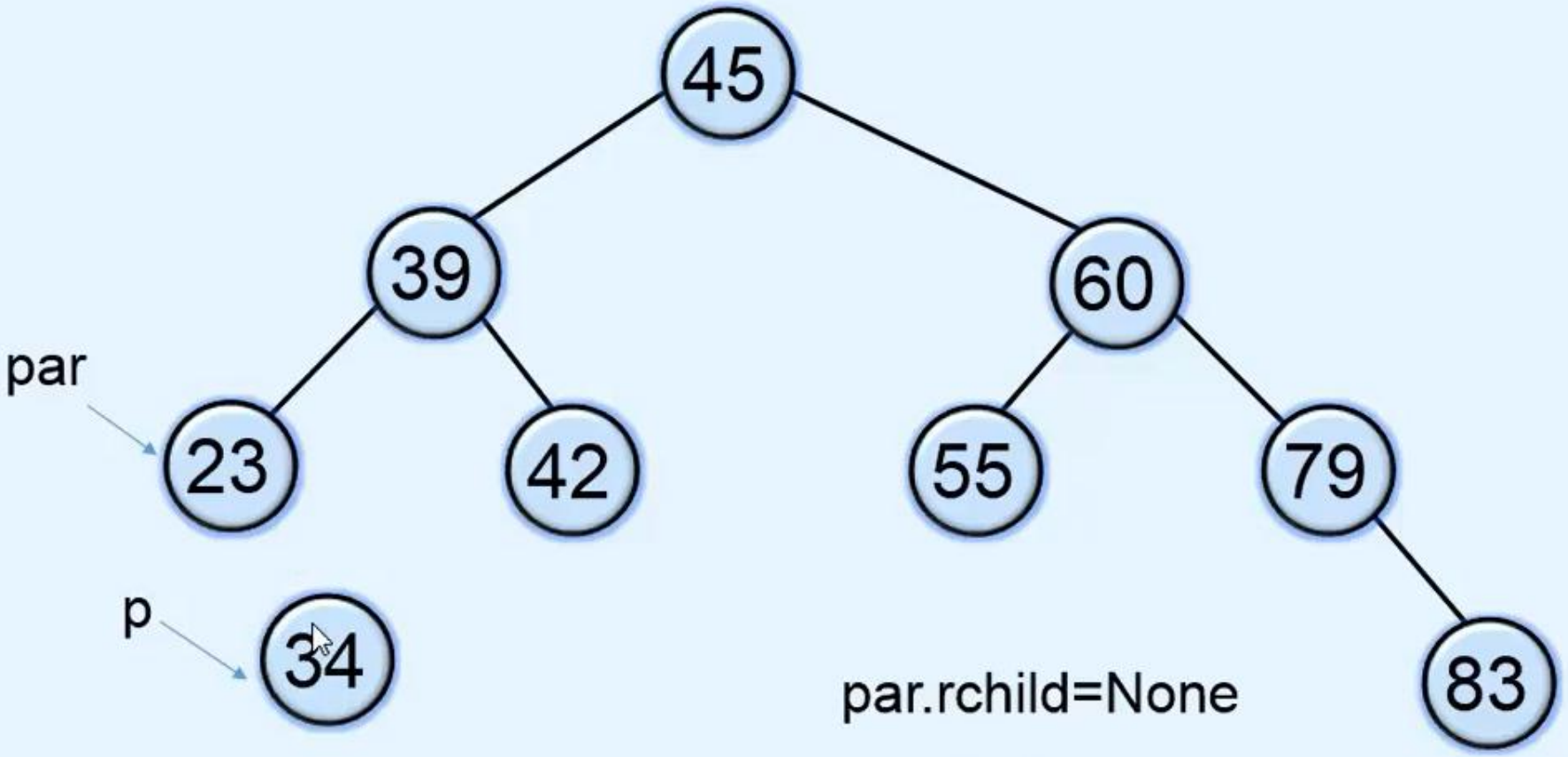
Case A : Node is Leaf node

Delete 34



Case A : Node is Leaf node

Delete 34



Case B : Node has only one child

After deletion, single child will come at the place of deleted node

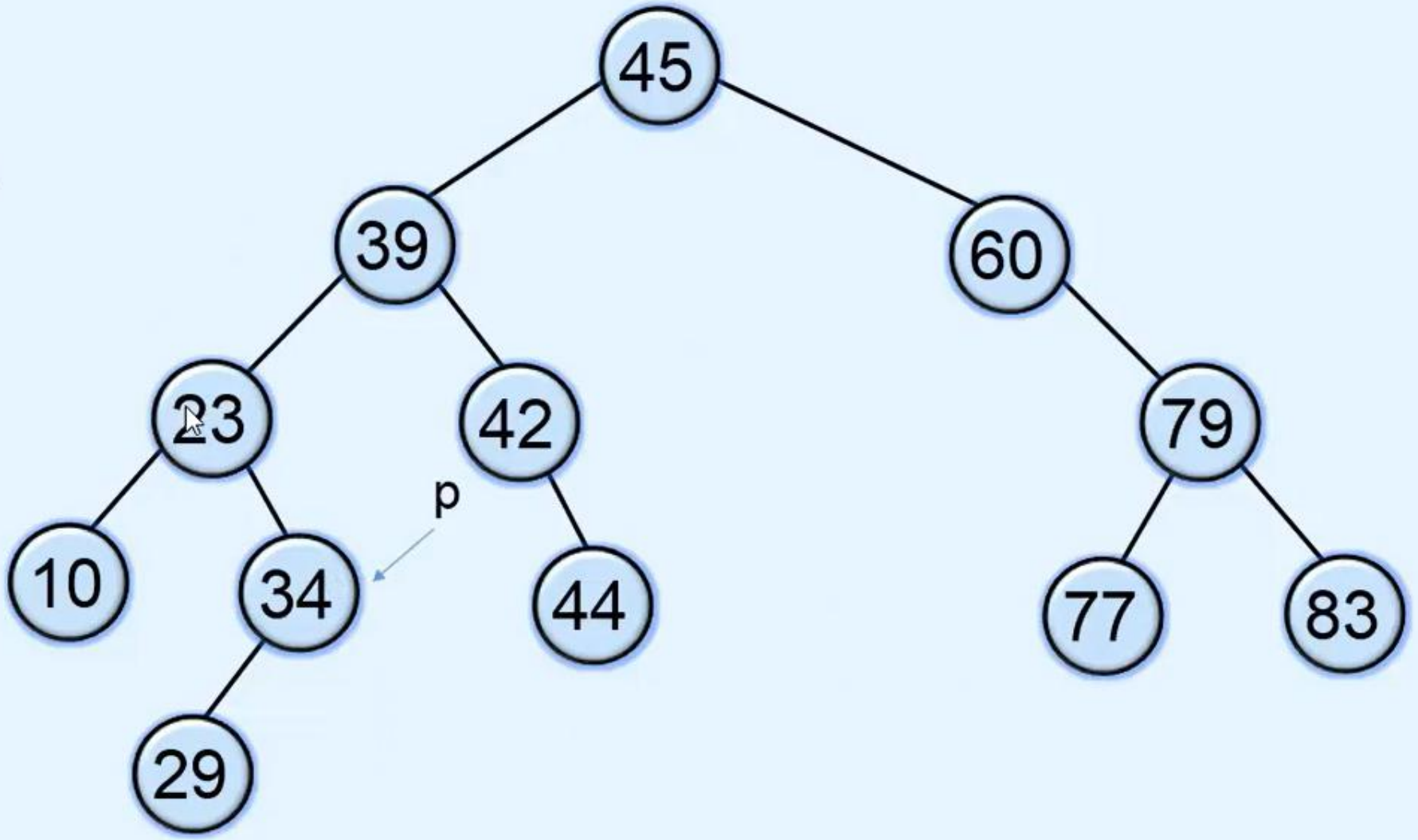
P	Node to be deleted
PR	Parent of the node that has to be deleted
CH	Child of the node that has to be deleted

If P is left child of PR → CH becomes left child of PR

If P is right child of PR → CH becomes right child of PR

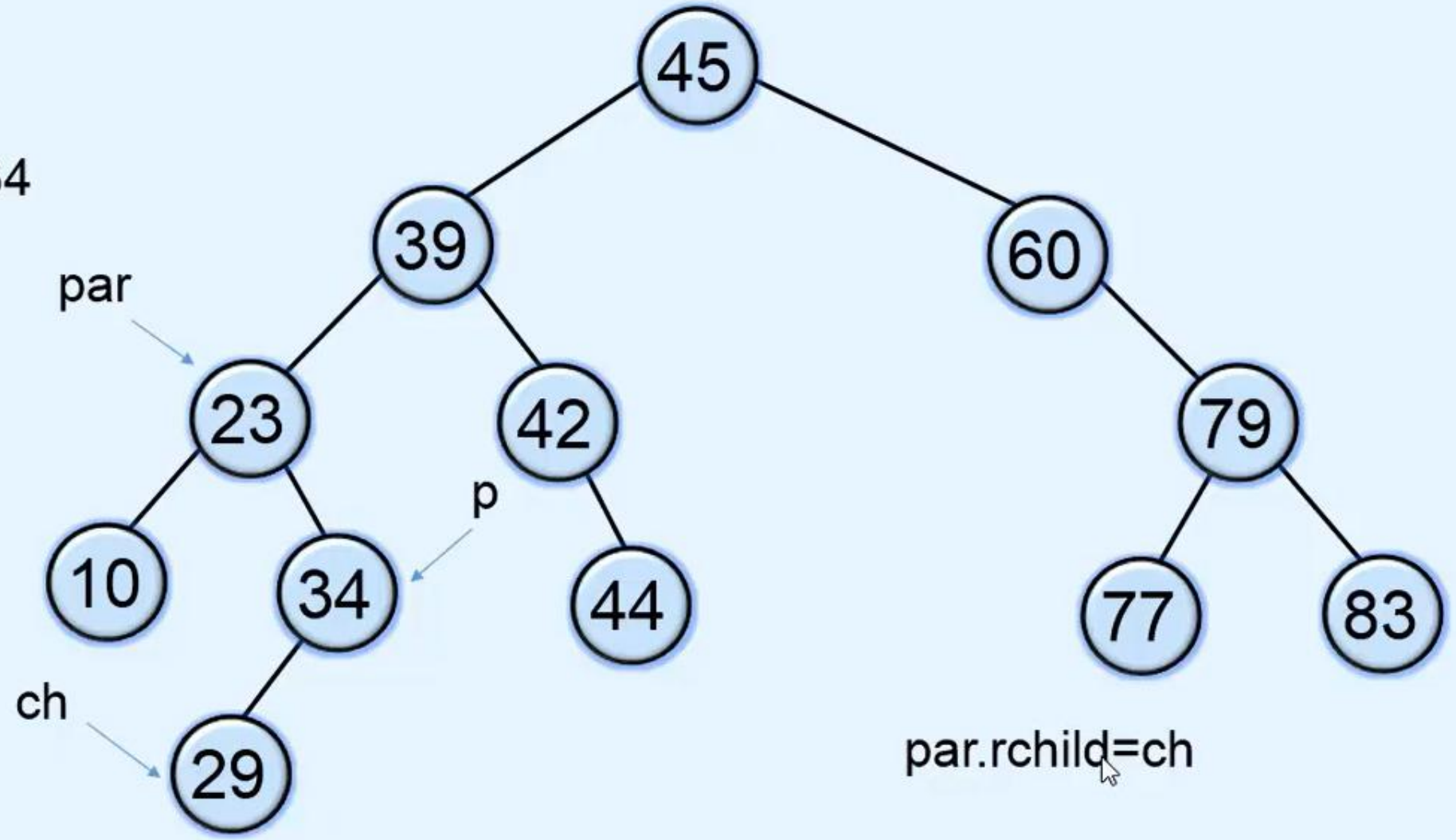
Case B : Node has only one child

Delete 34



Case B : Node has only one child

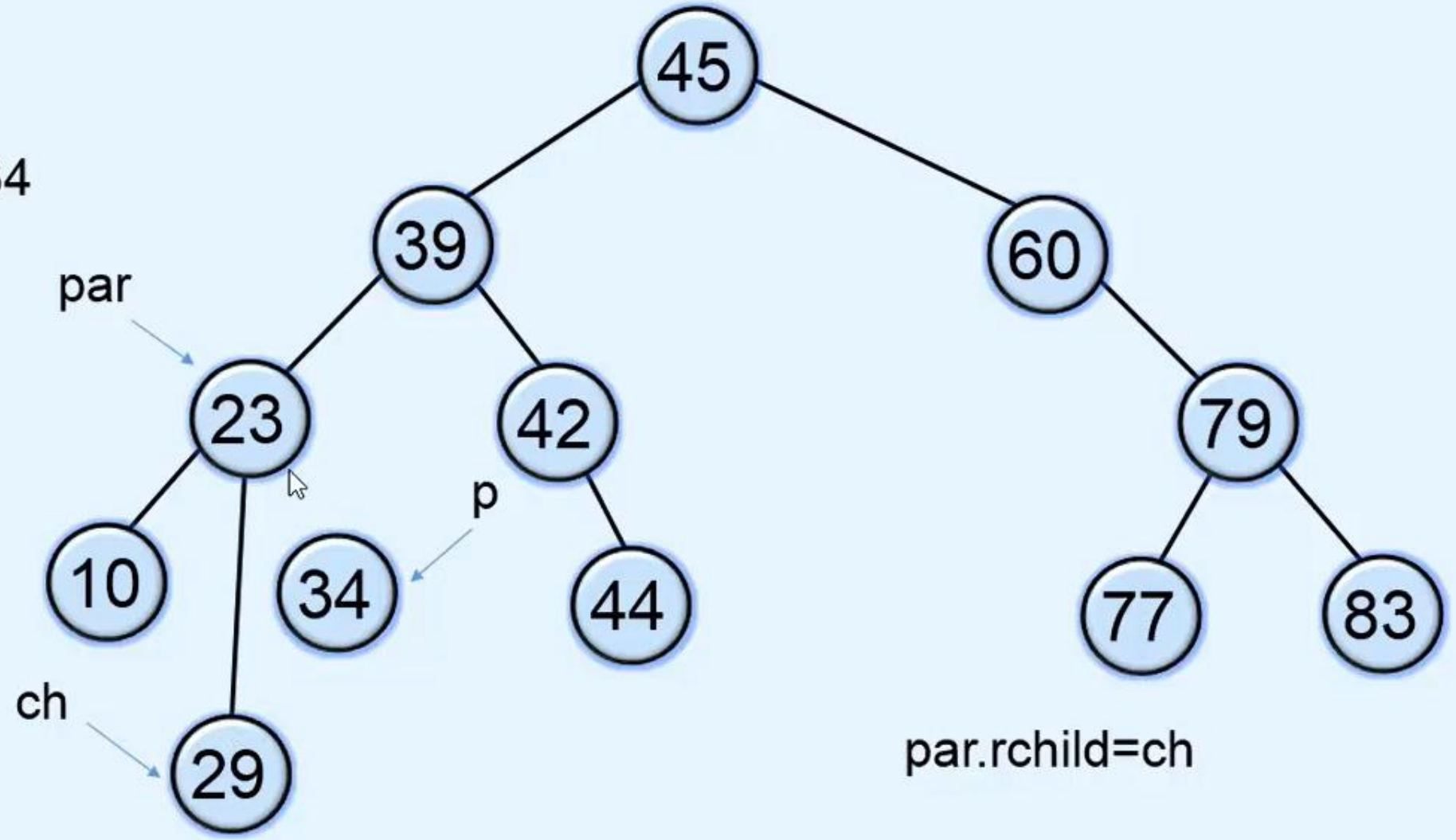
Delete 34



par.rchild=ch

Case B : Node has only one child

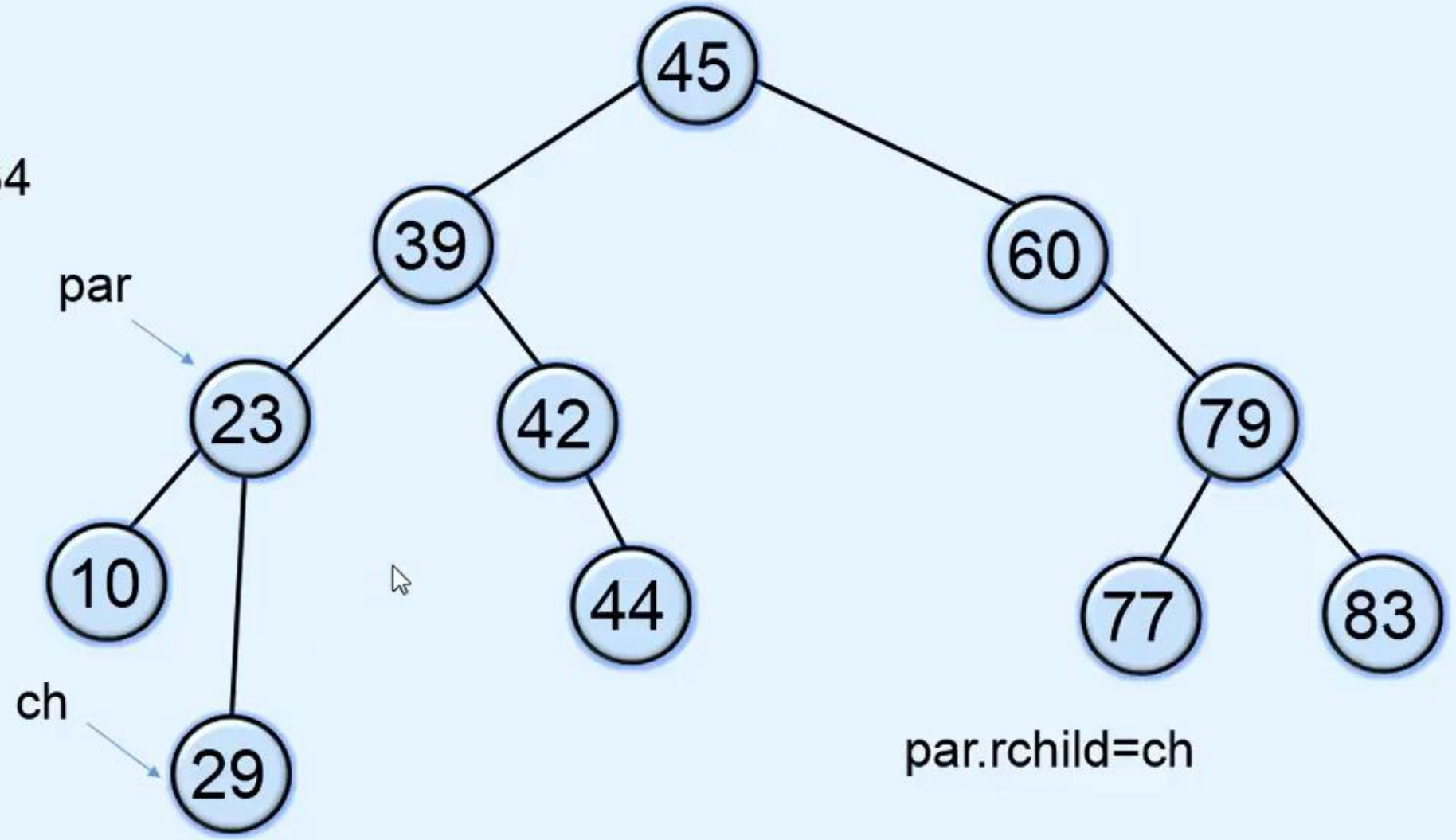
Delete 34



par.rchild=ch

Case B : Node has only one child

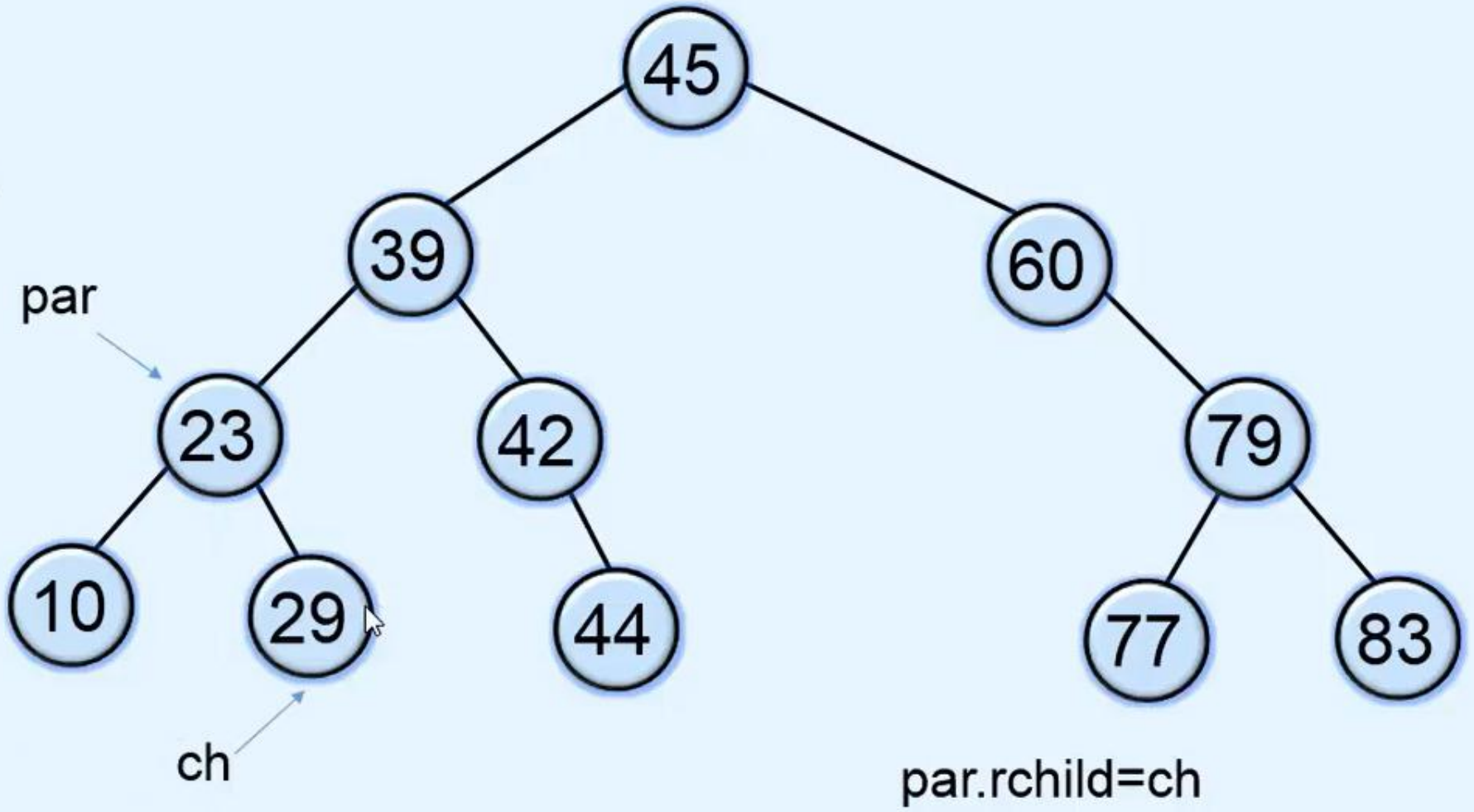
Delete 34



par.rchild=ch

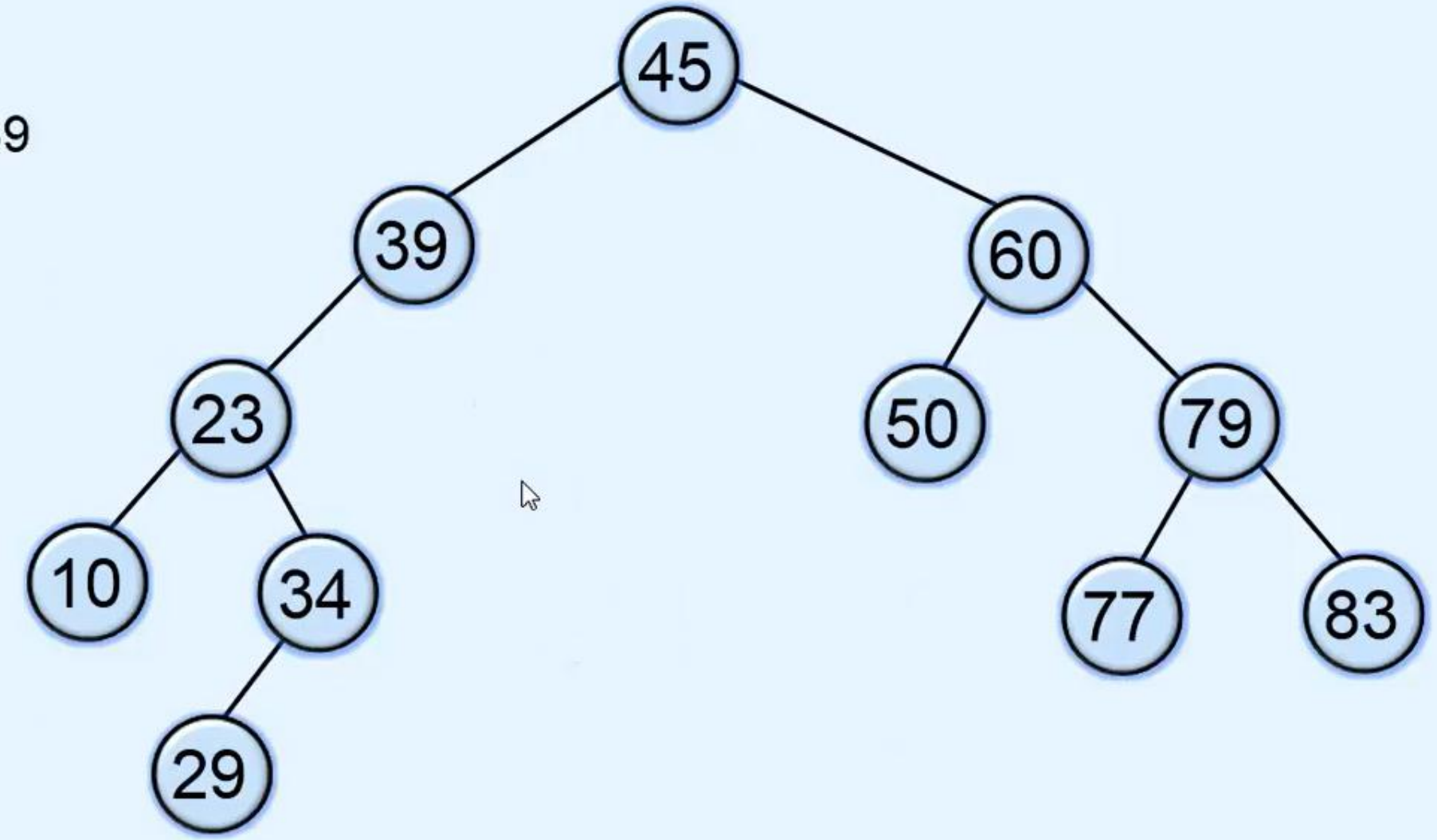
Case B : Node has only one child

Delete 34



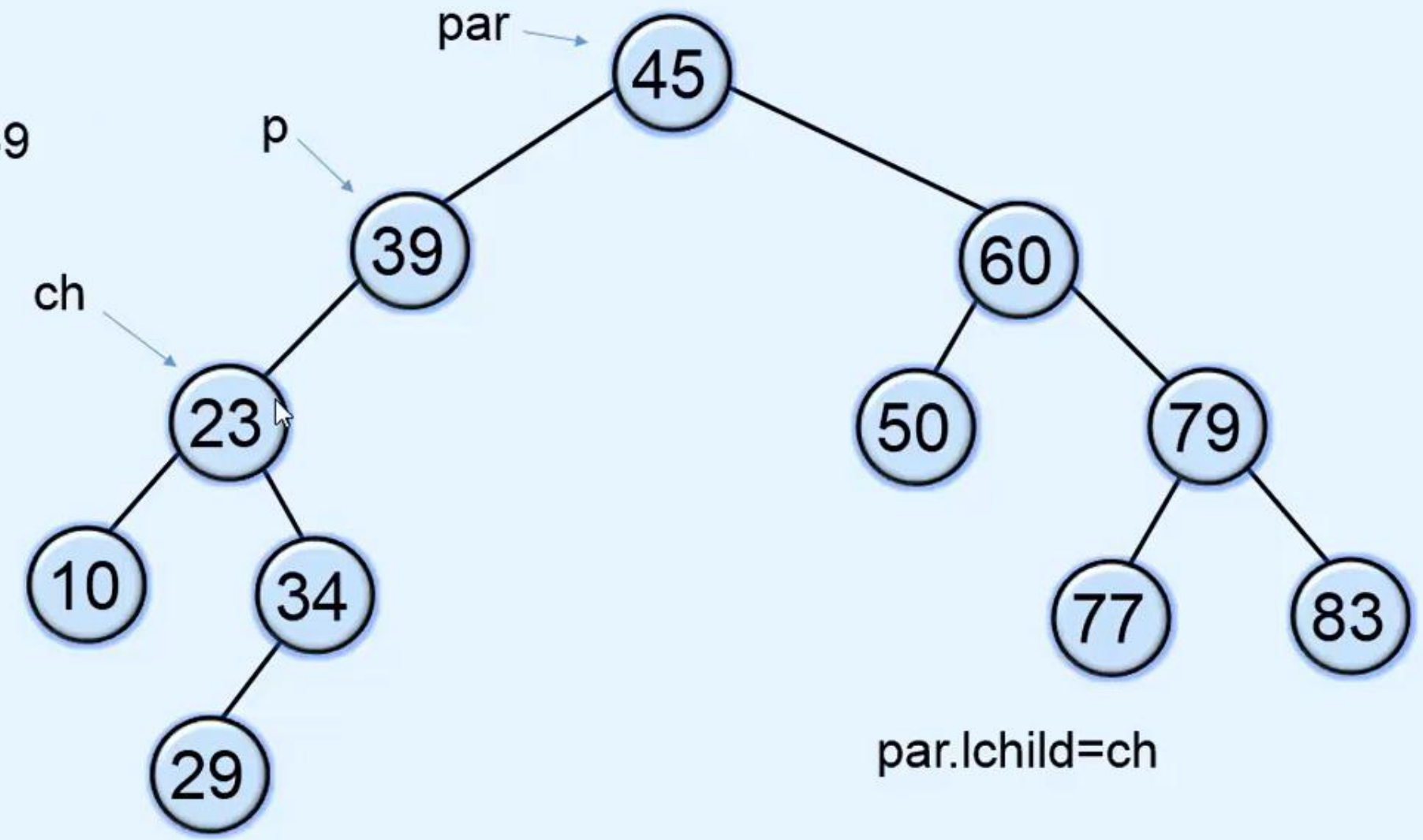
Case B : Node has only one child

Delete 39



Case B : Node has only one child

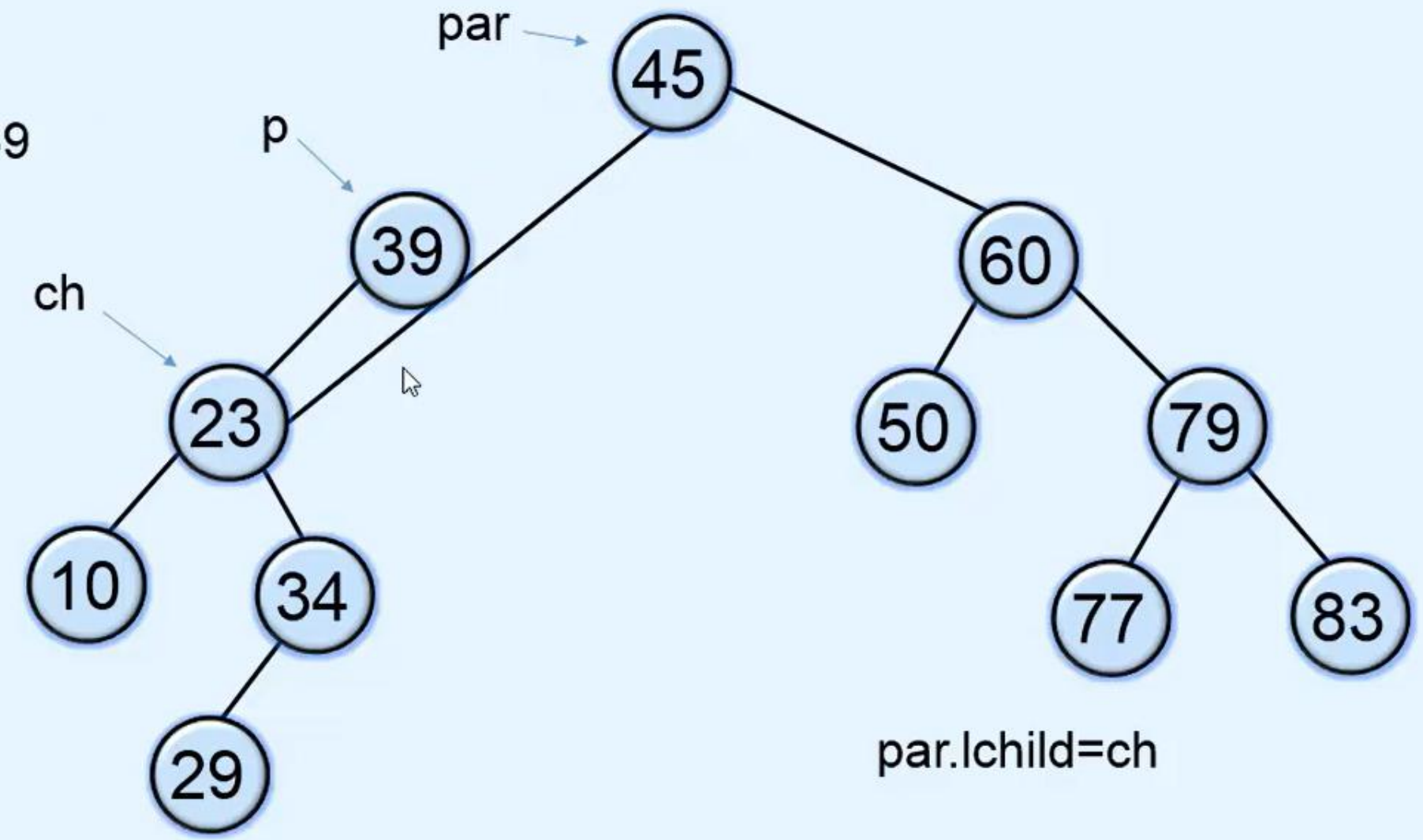
Delete 39



par.lchild=ch

Case B : Node has only one child

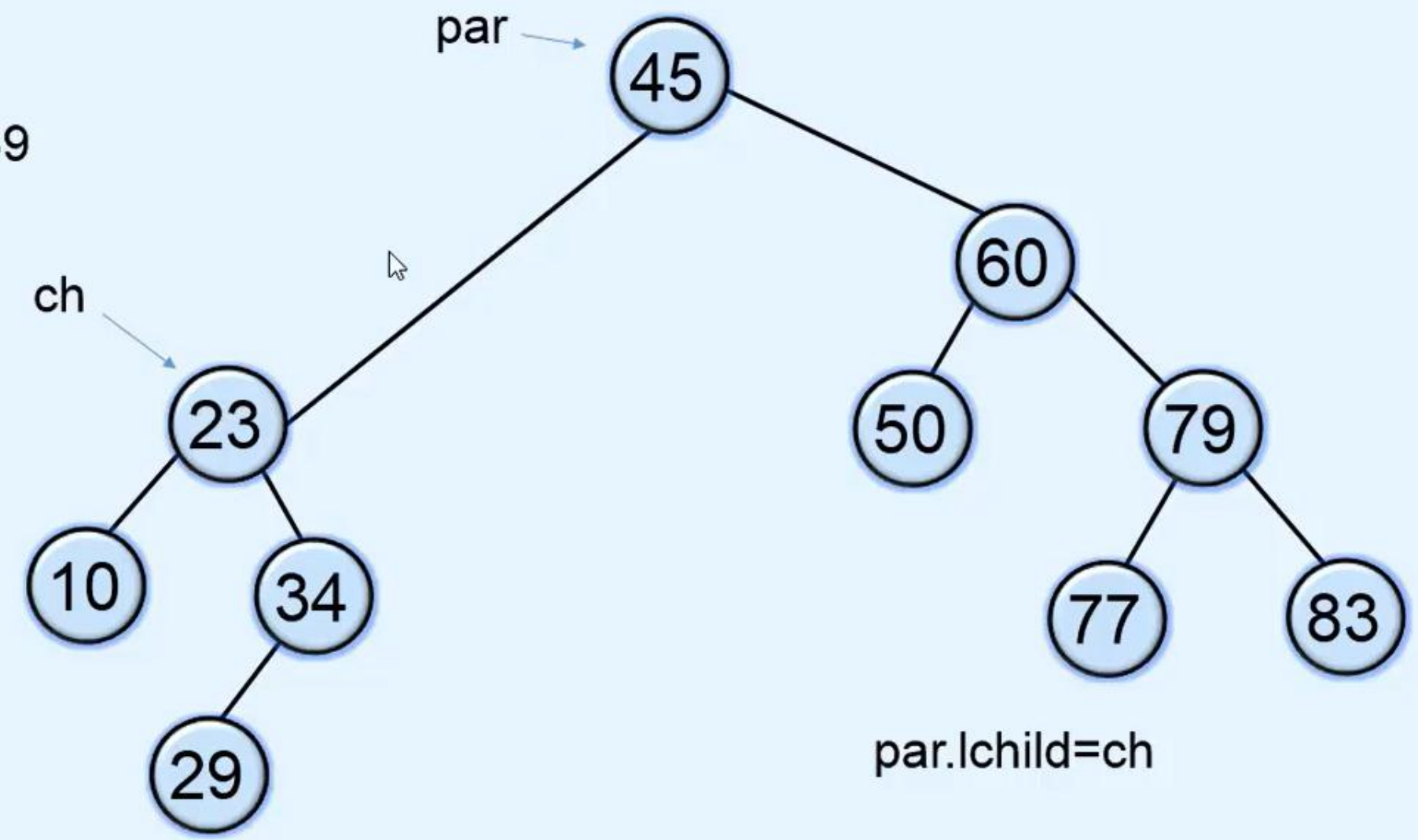
Delete 39



par.lchild=ch

Case B : Node has only one child

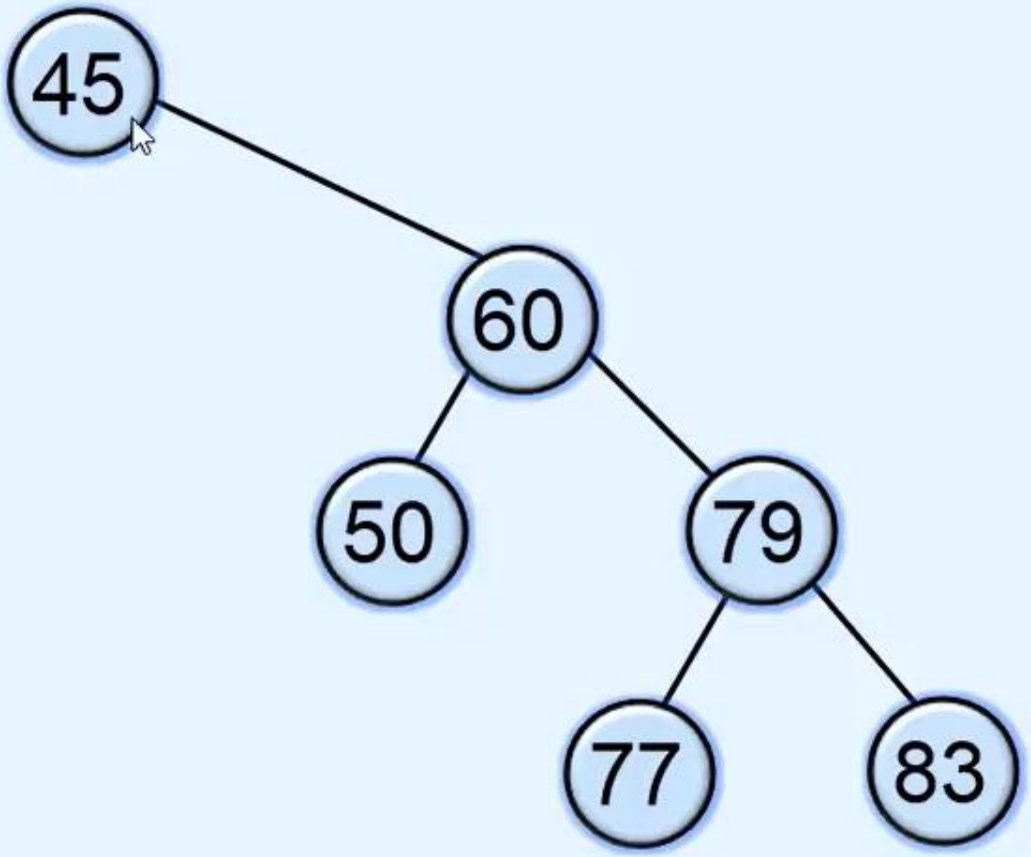
Delete 39



par.lchild=ch

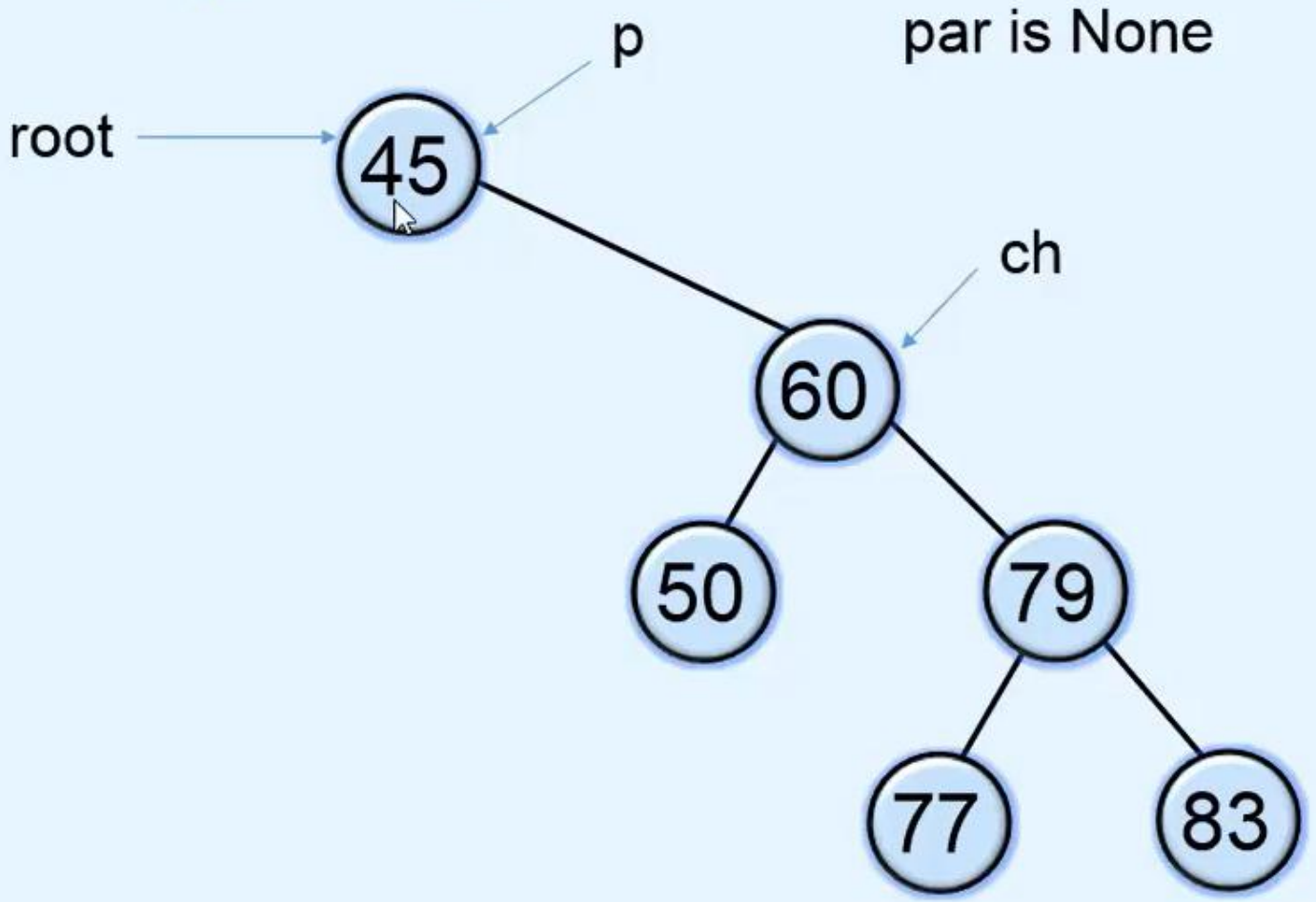
Case B : Node has only one child

Delete 45



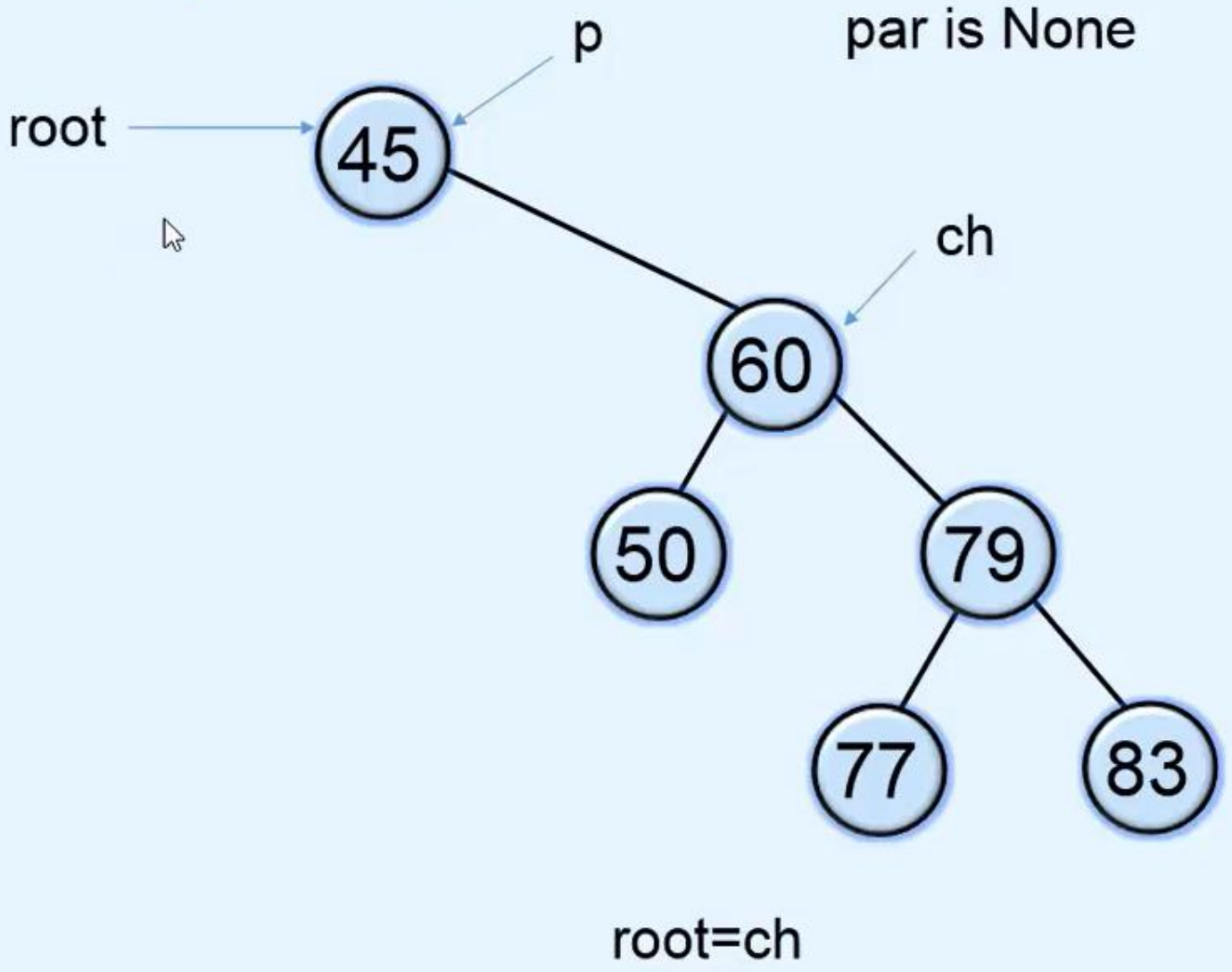
Case B : Node has only one child

Delete 45



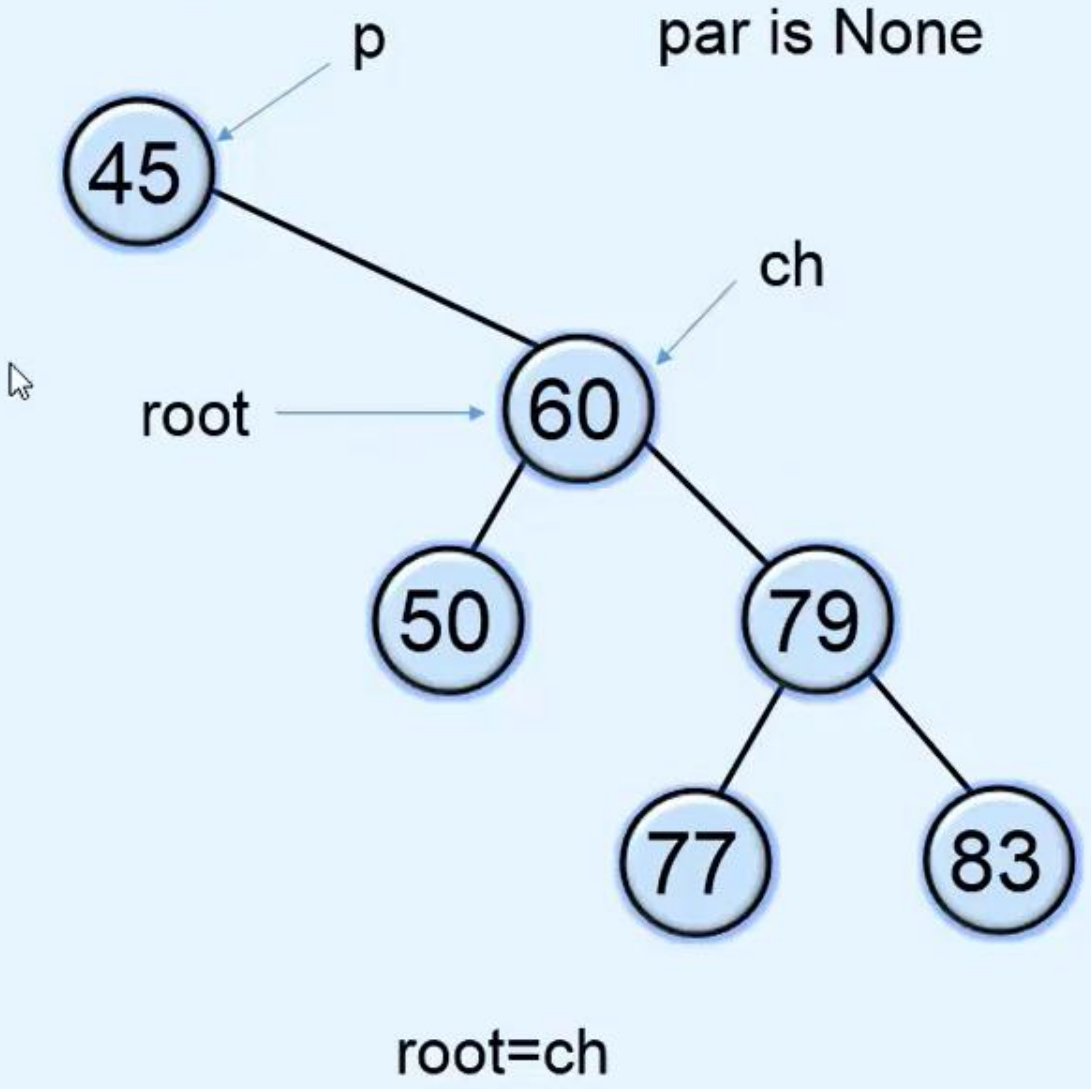
Case B : Node has only one child

Delete 45



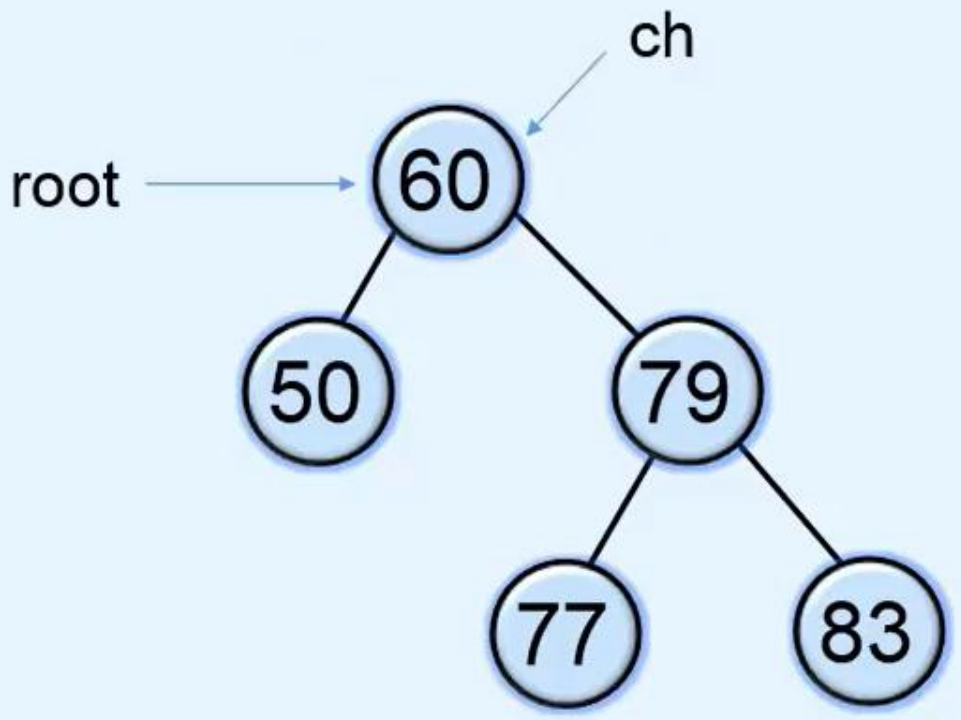
Case B : Node has only one child

Delete 45



Case B : Node has only one child

Delete 45

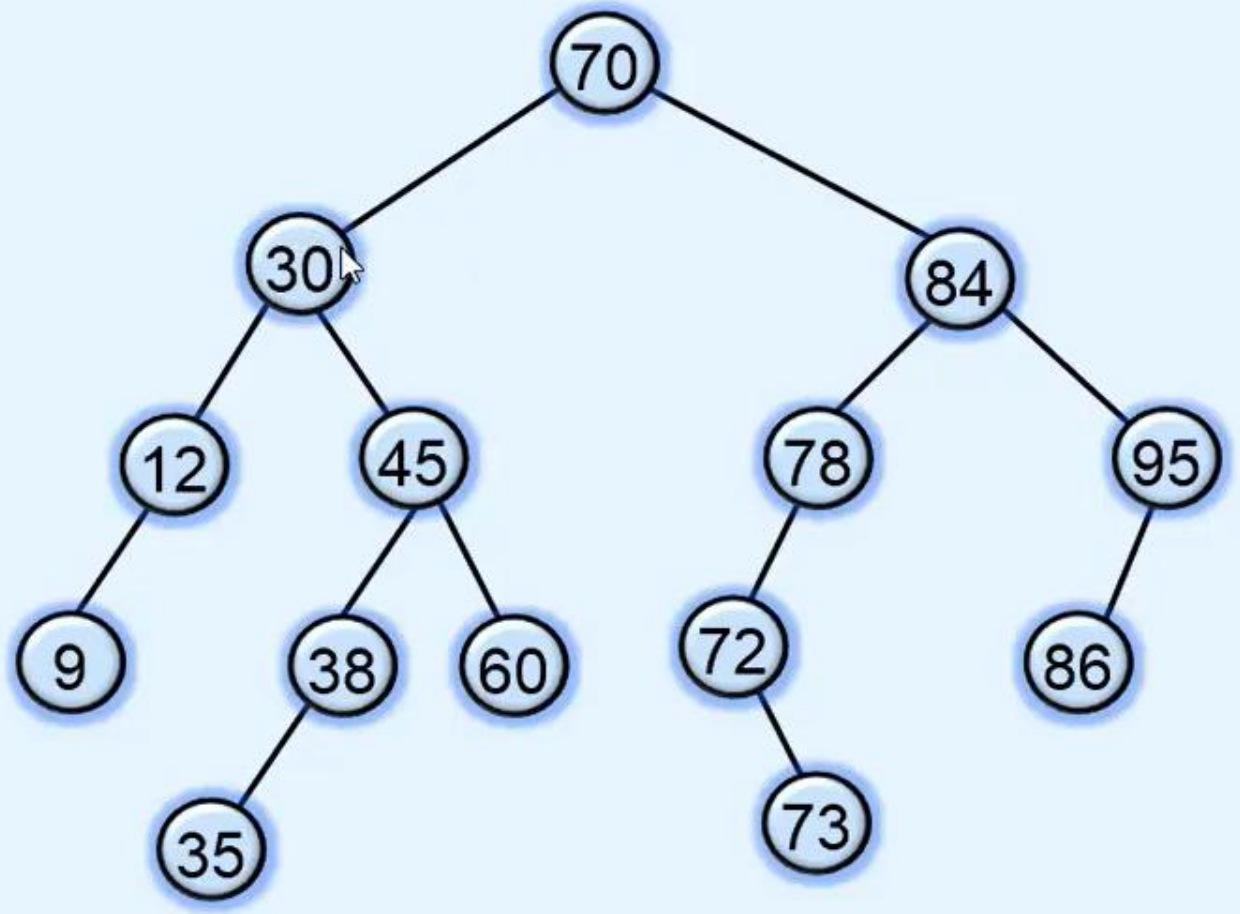


root=ch

Case C : Node has two children

Find the inorder successor of the node to be deleted

Inorder successor of N: Leftmost node in the right subtree of N



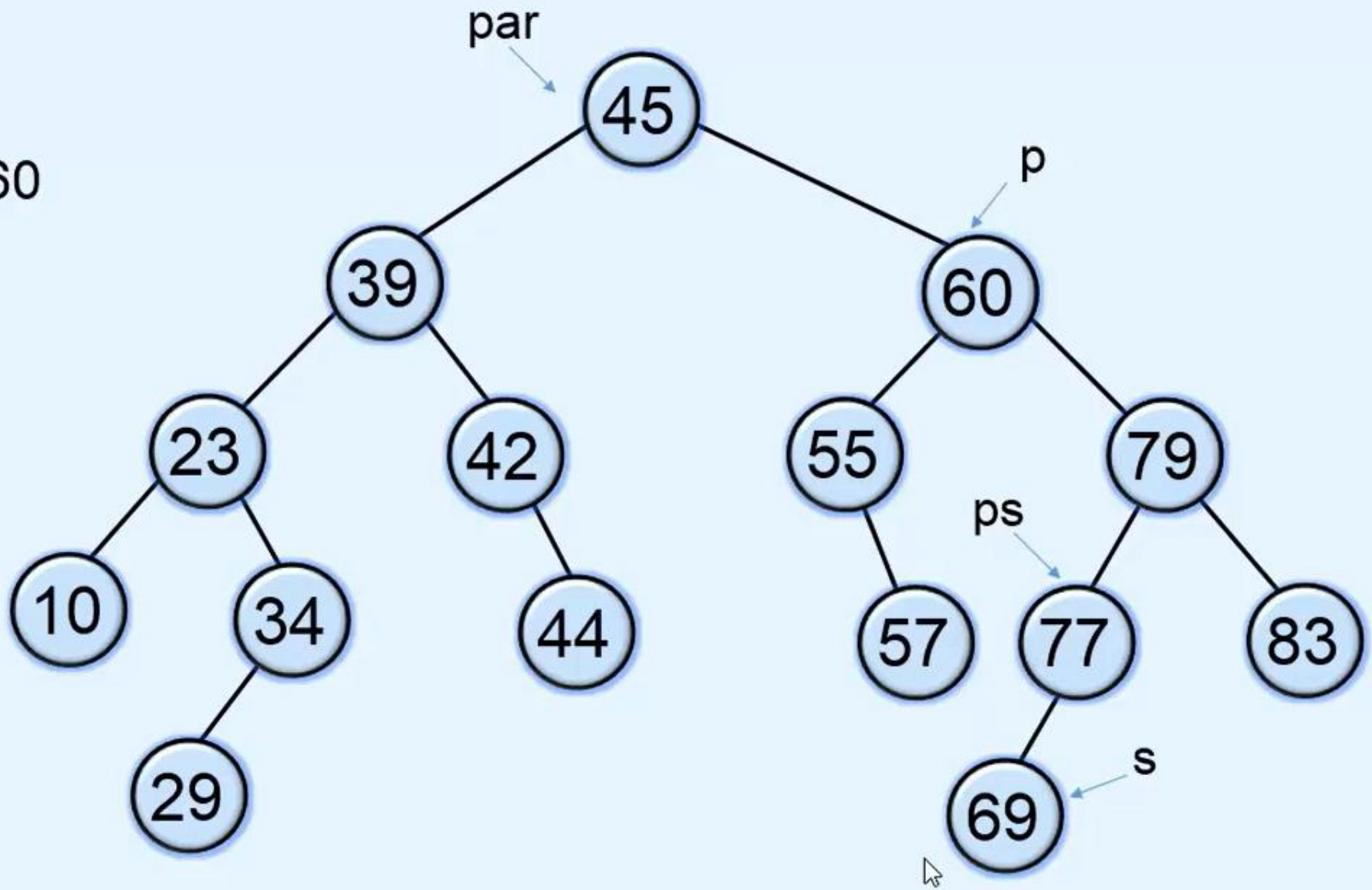
Case C : Node has two children

Find the inorder successor of the node to be deleted

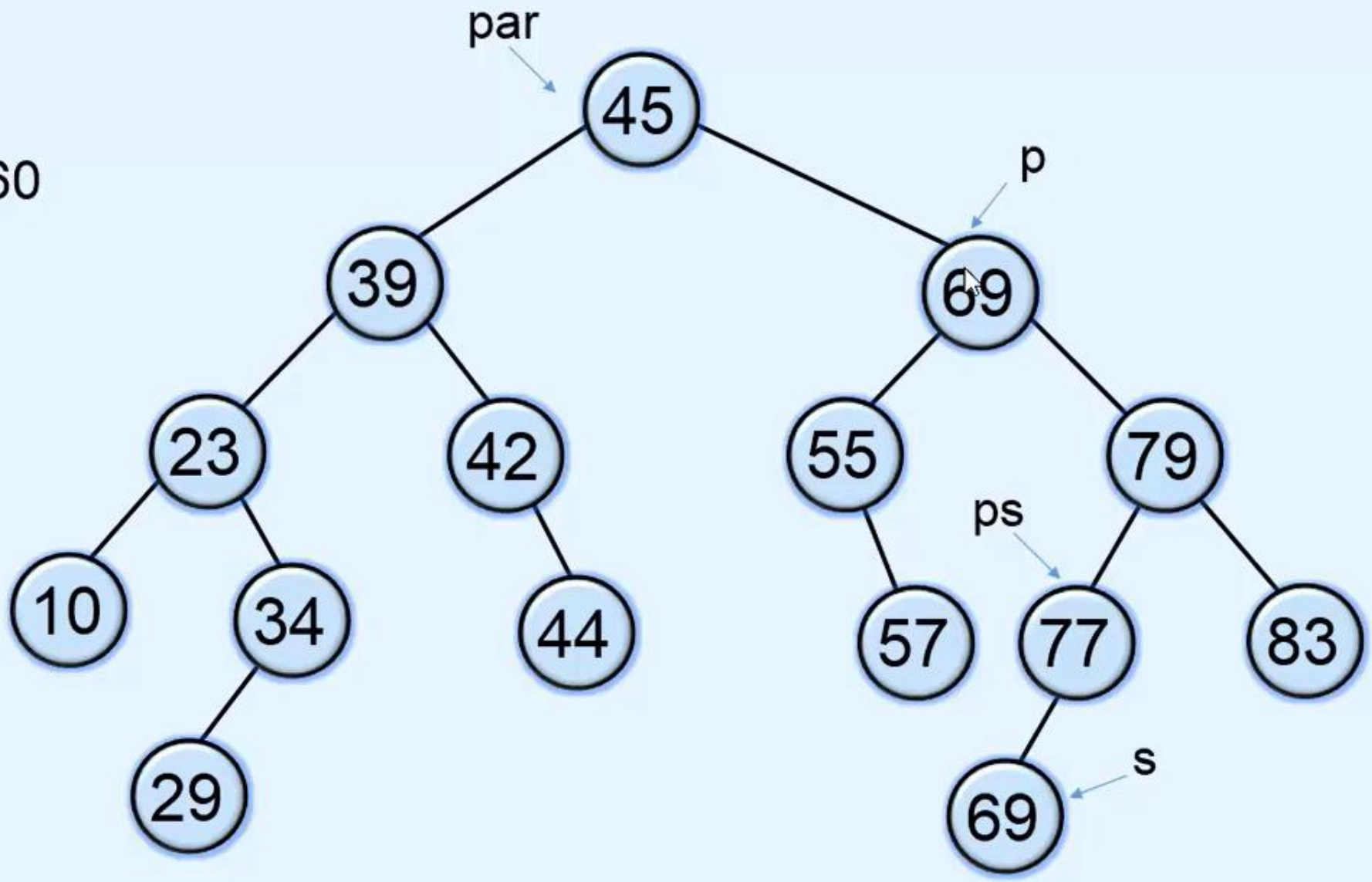
Copy the data of the inorder successor to the node

Delete the inorder successor from the tree → Case A / Case B

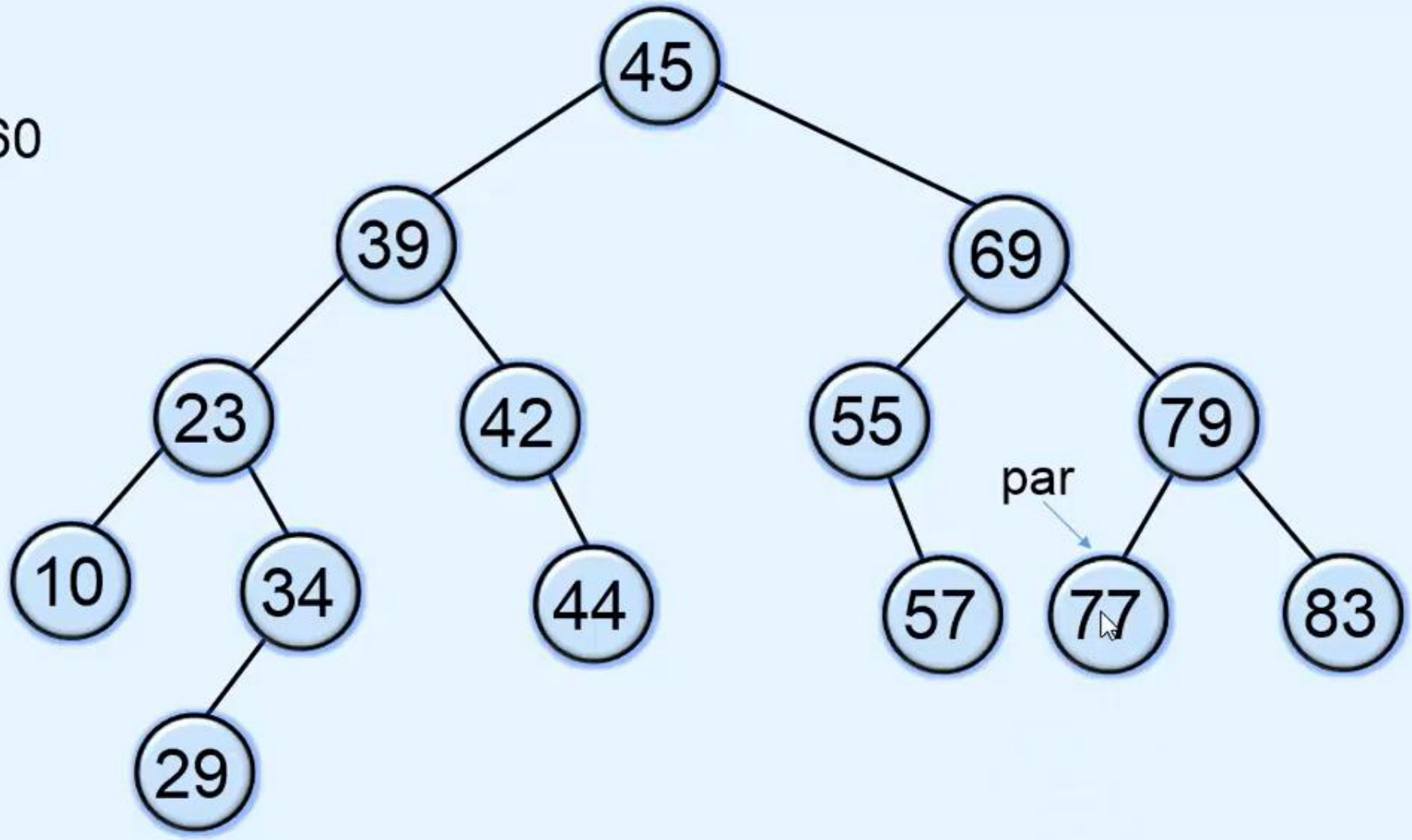
Delete 60



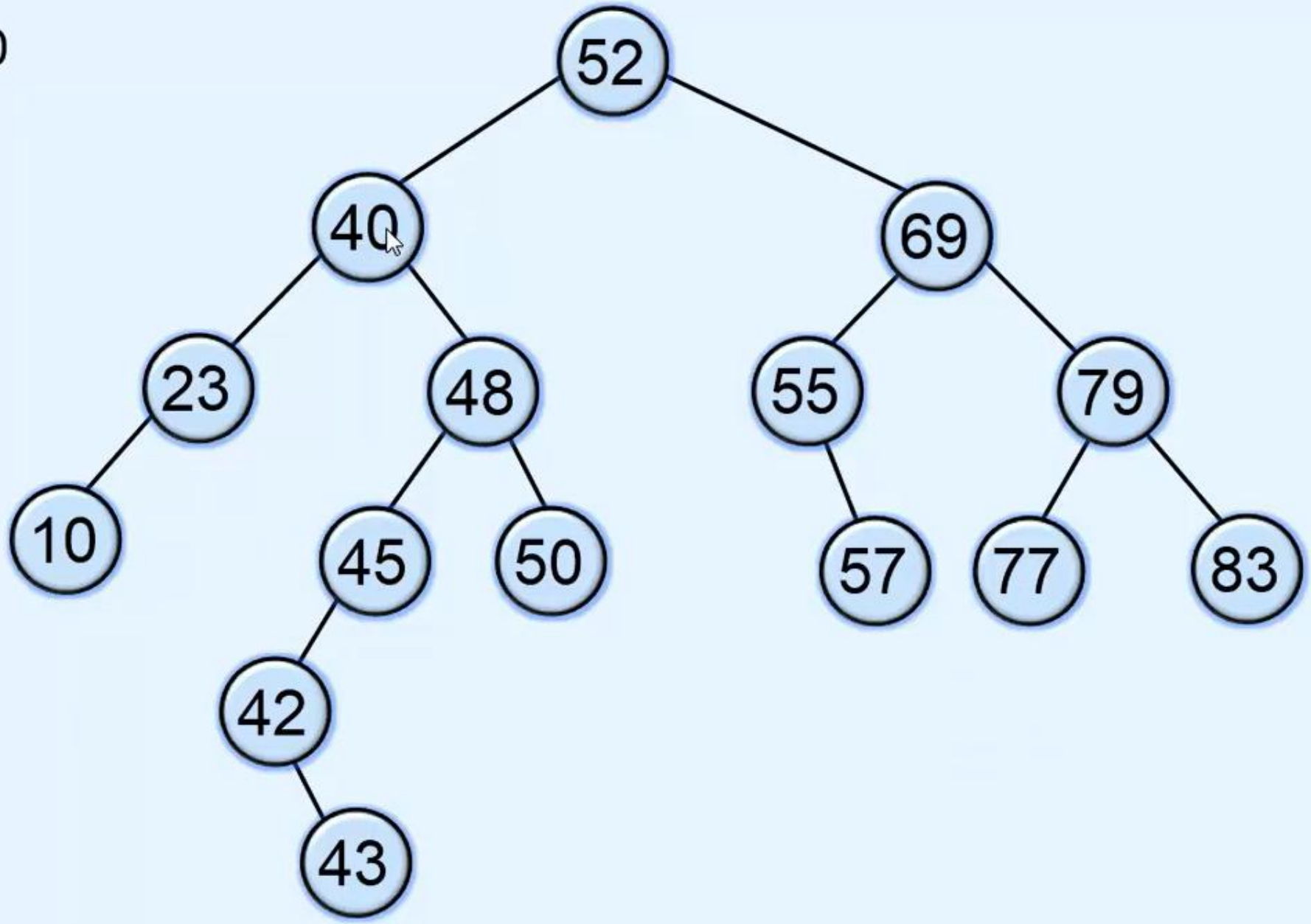
Delete 60



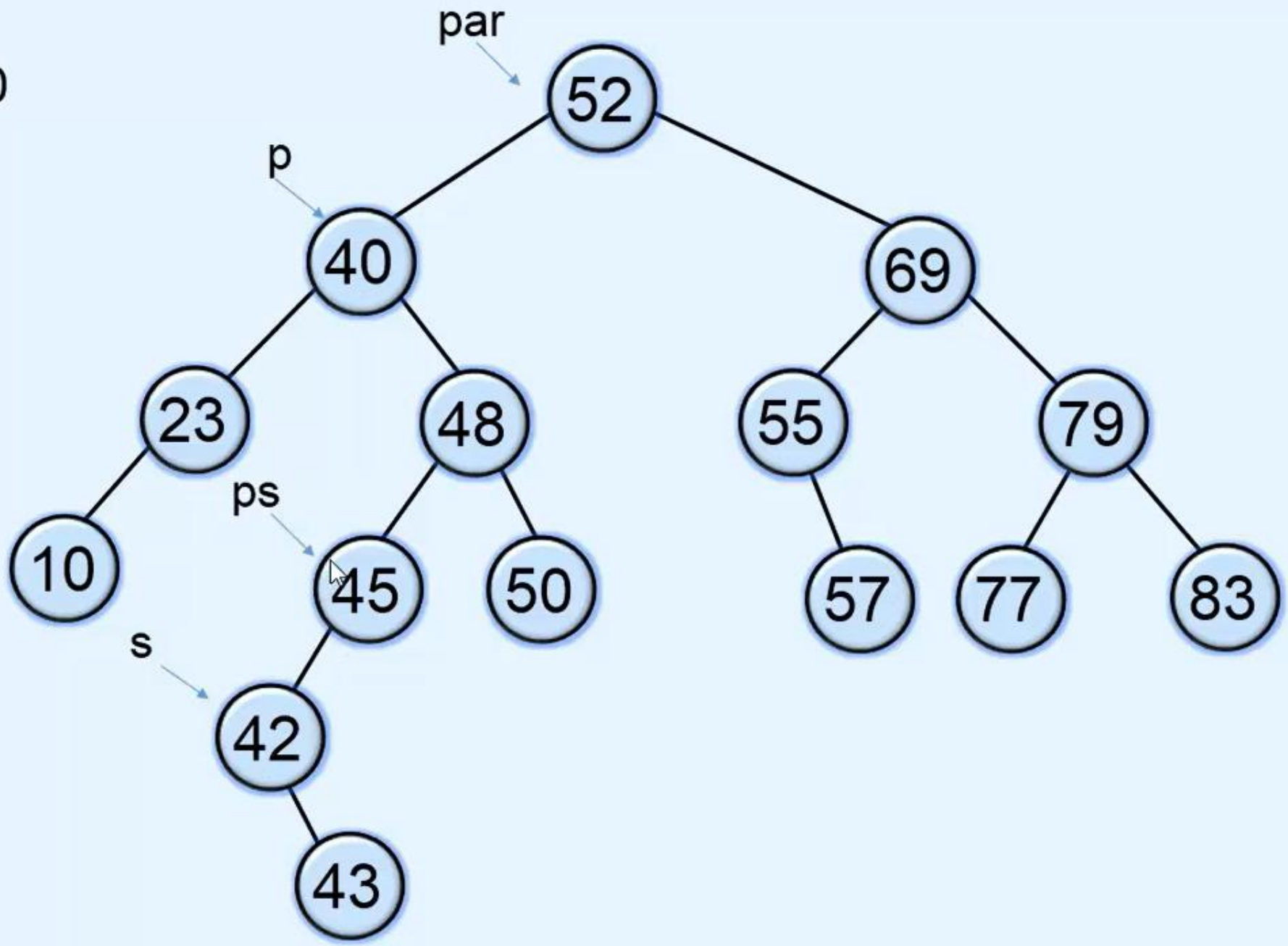
Delete 60



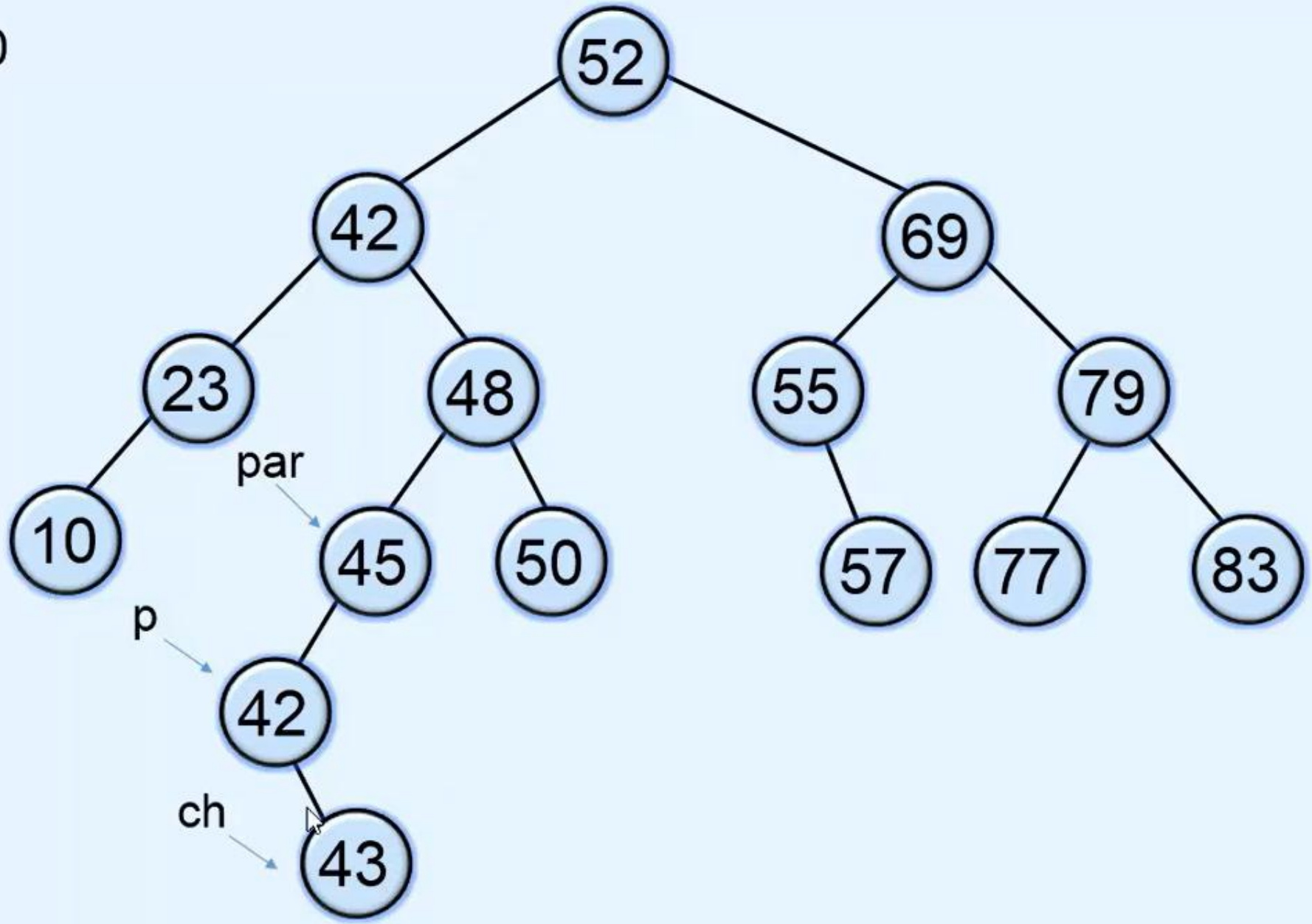
Delete 40



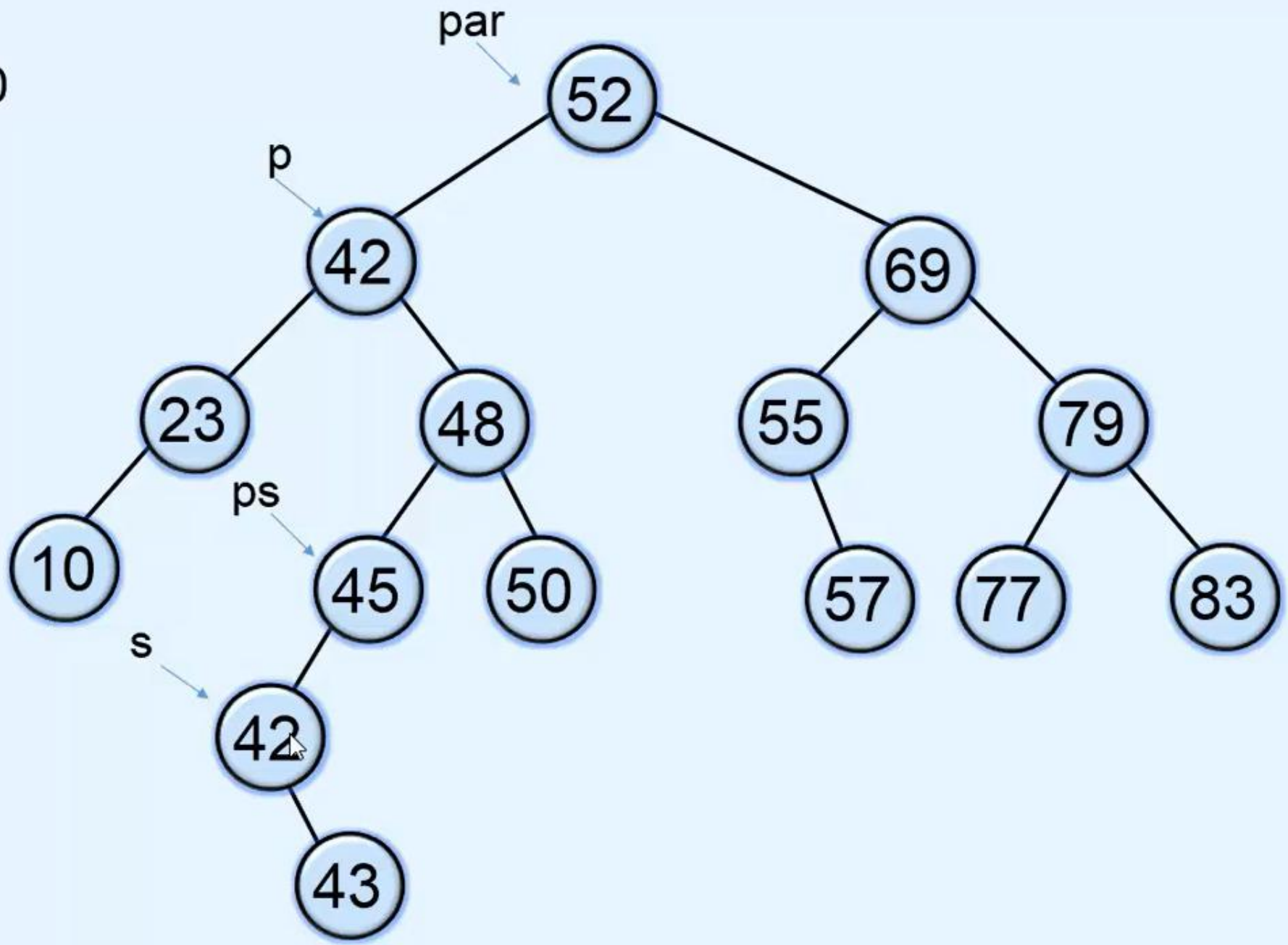
Delete 40



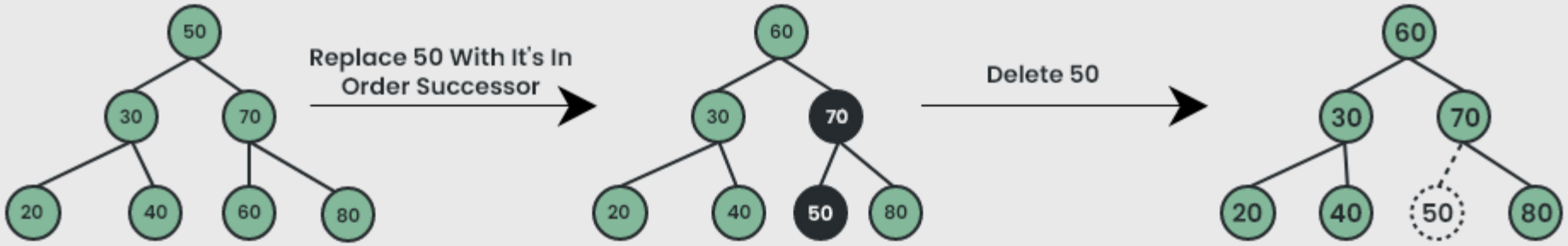
Delete 40



Delete 40



Case 3 : Delete A Node With Both Children In BST



Delete Node 50

After Deletion

Deletion In BST