Search Trees

Chapter 14

Search Trees

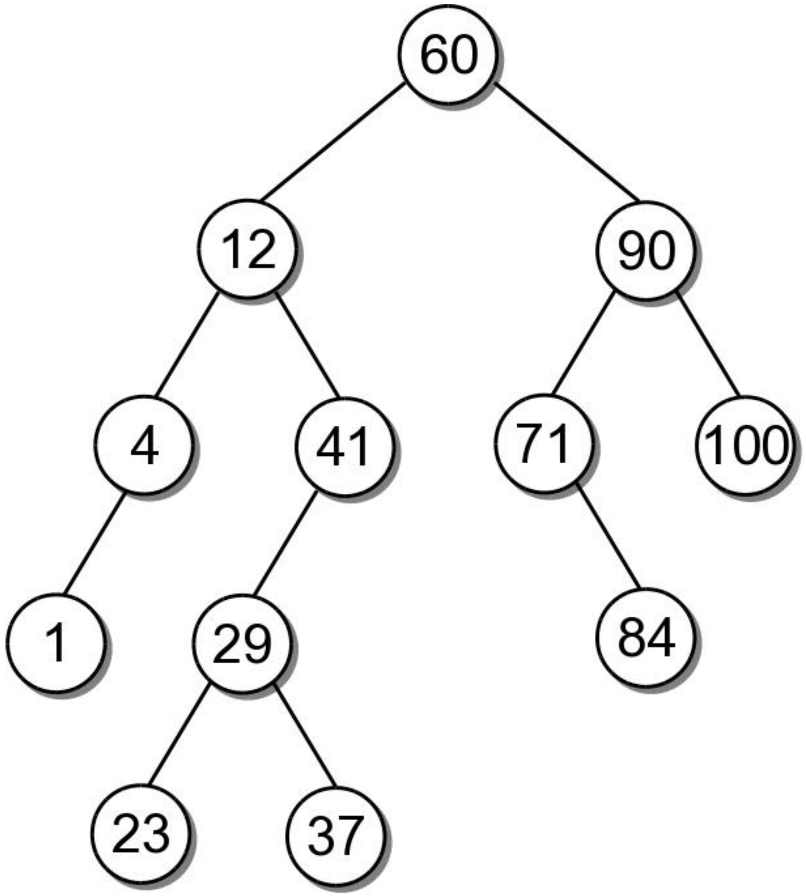
- The tree structure can be used for searching.
  - Each node contains a search key as part of its data or **payload**.
  - Nodes are organized based on the relationship between the keys.
- Search trees can be used to implement various types of containers.
  - Most common use is with the Map ADT.

Binary Search Tree (BST)

- A binary tree in which each node contains a search key and the tree is structured such that for each interior node V:
  - All keys less than the key in node V are stored in the left subtree of V.
  - All keys greater than the key in node V are stored in the right subtree of V.

BST Example

- Consider the example tree



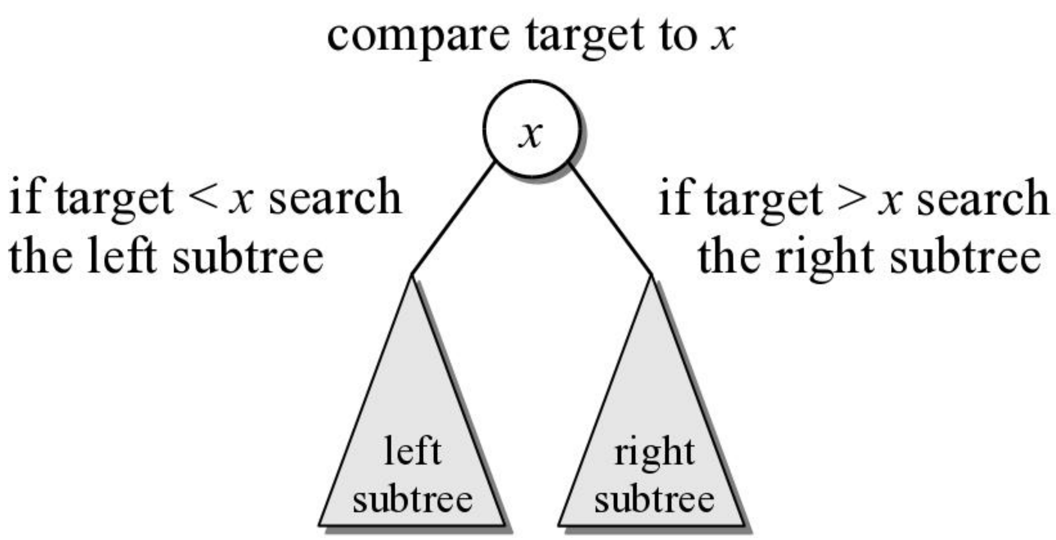
BST – Map ADT

bstmap.py

```
# We use an unique name to distinguish this version
# from others in the chapter.
class BSTMap :
  def __init__( self ):
    self._root = None
    self._size = 0

  def __len__( self ):
    return self._size

  def __iter__( self ):
    return _BSTreeIterator( self._root )
# ...

# Storage class for the binary search tree nodes.
class _BSTNode :
  def __init__( self, key, data ):
    self.key = key
    self.data = data
    self.left = None
    self.right = None
```

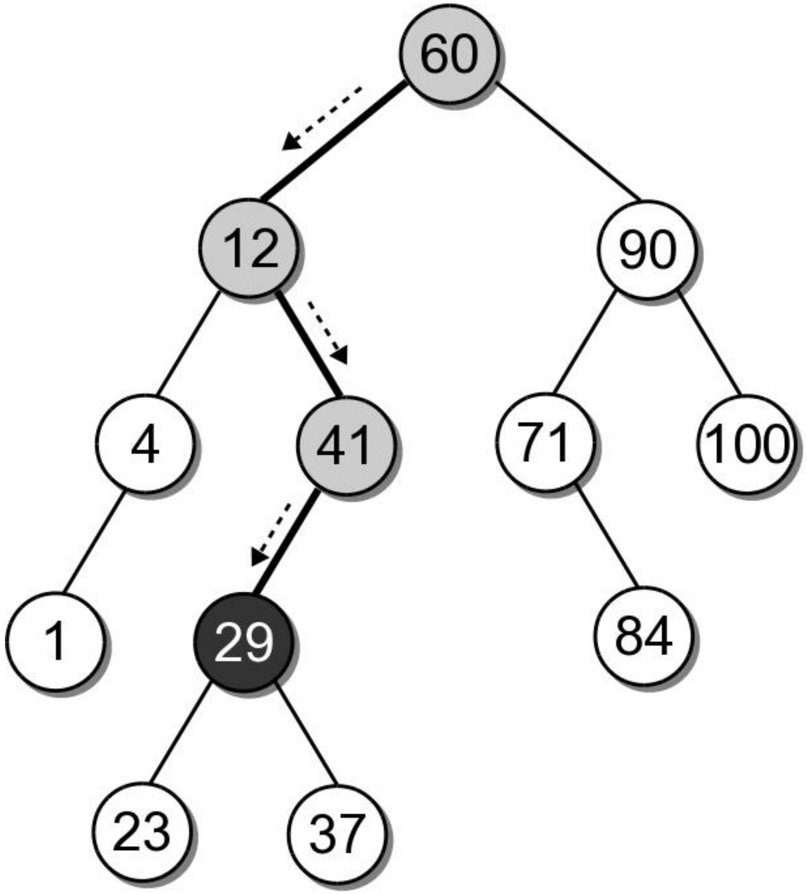
BST – Searching

- A search begins at the root node.
  - The target is compared to the key at each node.
  - The path depends on the relationship between the target and the key in the node.



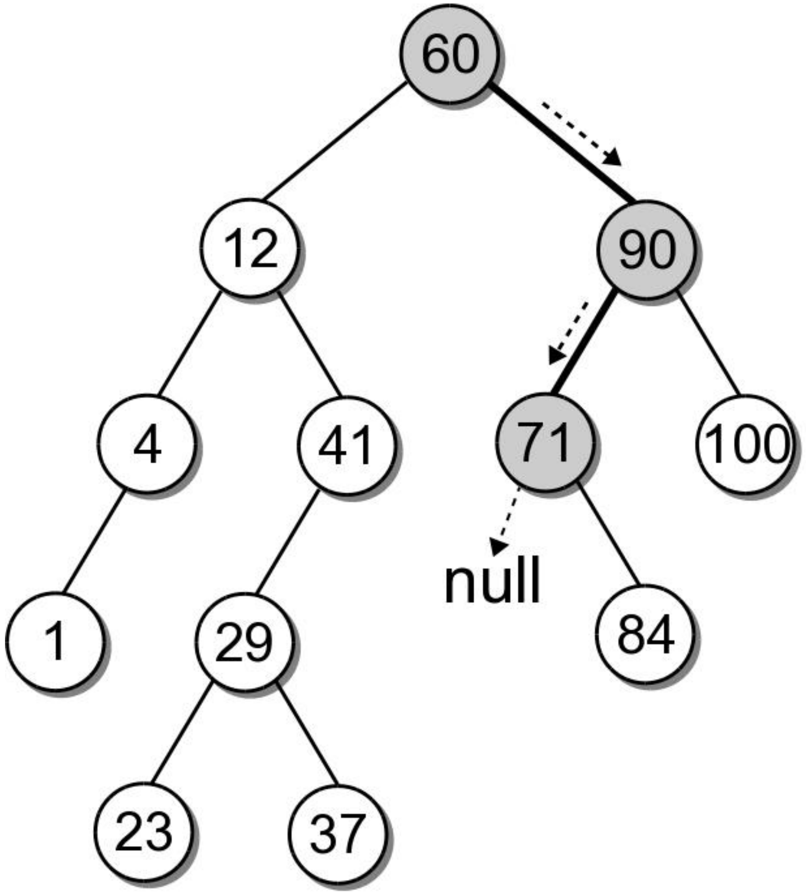
BST – Search Example

- Suppose we want to search for 29 in our BST.



BST – Search Example

- What if the key is not in the tree? Search for key 68 in our BST.



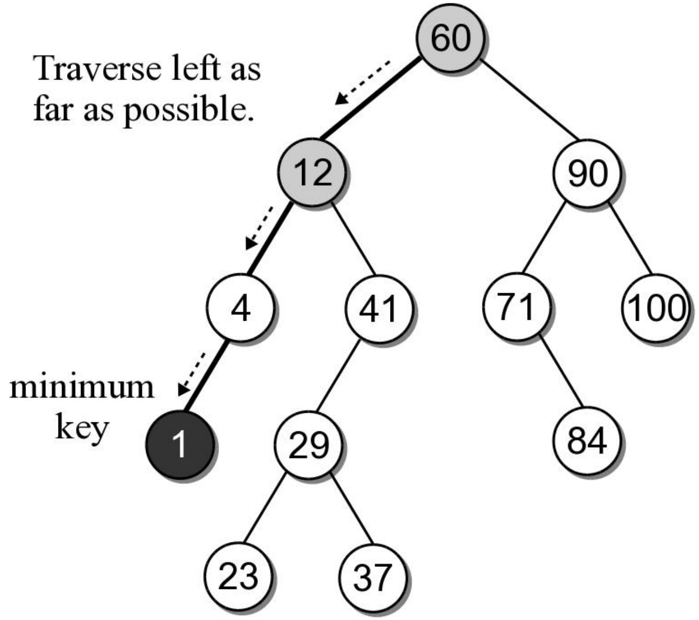
BST – Search Implementation

bstmap.py

```
class BSTMap :
# ...
  def __contains__( self, key ):
    return self._bstSearch( self._root, key ) is not None

  def valueOf( self, key ):
    node = self._bstSearch( self._root, key )
    assert node is not None, "Invalid map key."
    return node.value

  def _bstSearch( self, subtree, target ):
    if subtree is None :
      return None
    elif target < subtree.key :
      return self._bstSearch( subtree.left )
    elif target > subtree.key :
      return self._bstSearch( subtree.right )
    else :
      return subtree
```

BST – Min or Max Key

- Finding the minimum or maximum key within a BST is similar to the general search.
  - Where might the smallest key be located?
  - Where might the largest key be located?



BST – Min or Max Key

- The helper method below finds the node containing the minimum key.

```
class BSTMap :
# ...
  def _bstMinumum( self, subtree ):
    if subtree is None :
      return None
    elif subtree.left is None :
      return subtree
    else :
      return self._bstMinimum( subtree.left )
```

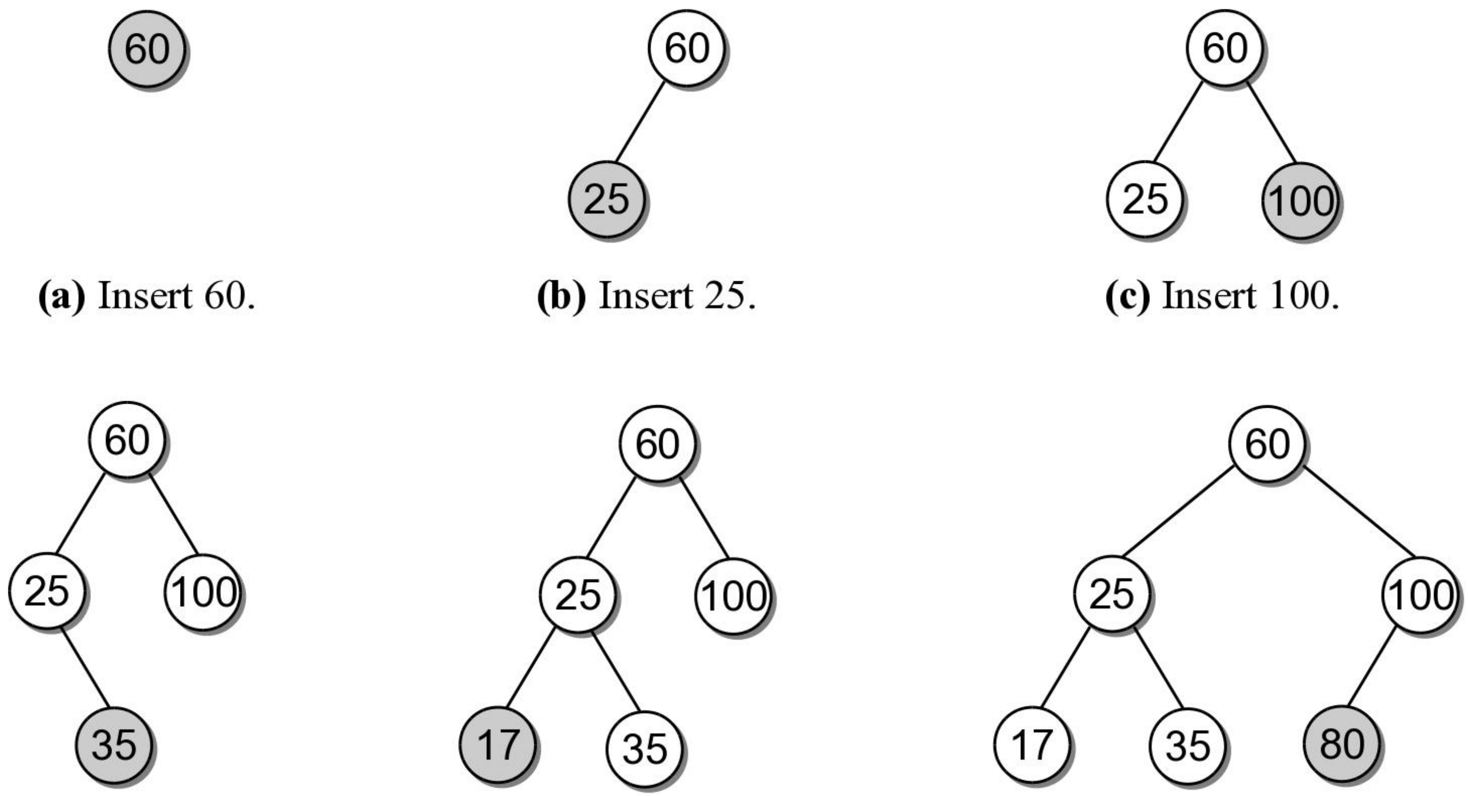
BST – Insertions

- When a BST is constructed, the keys are added one at a time. As keys are inserted
  - A new node is created for each key.
  - The node is linked into its proper position within the tree.
  - The search tree property must be maintained.

Building a BST

- Suppose we want to build a BST from the key list

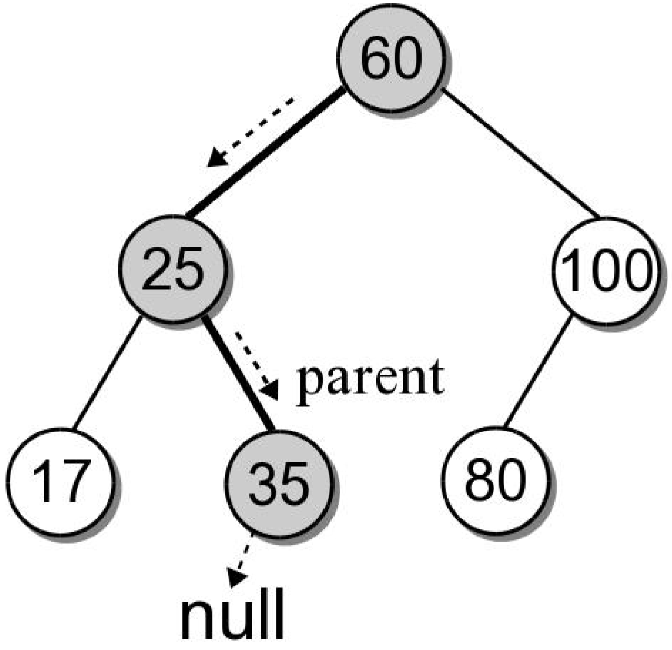
```
60  25  100  35  17  80
```



(a) Insert 60.
(b) Insert 25.
(c) Insert 100.
(d) Insert 35.
(e) Insert 17.
(f) Insert 80.

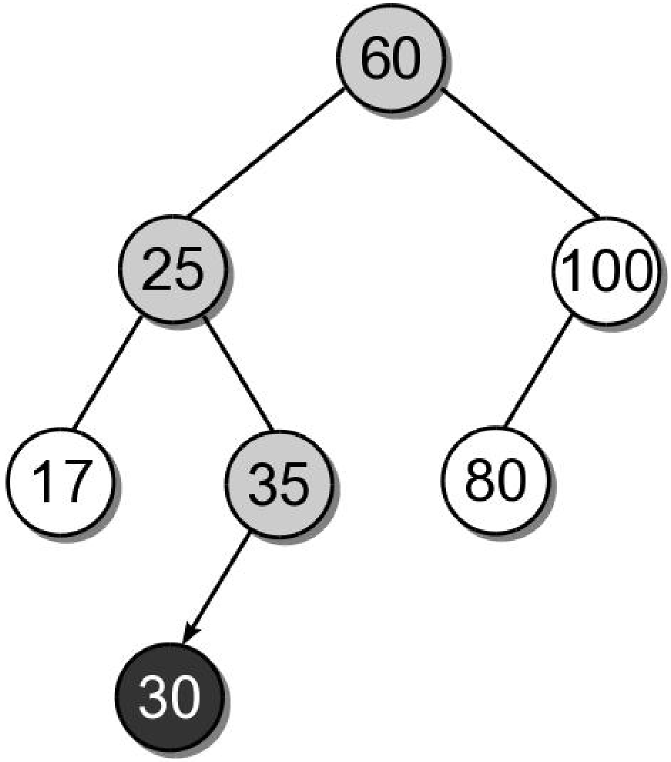
BST – Insertion

- Building a BST by hand is easy. How do we insert an entry in program code?
  - What happens if we use the search method from earlier to search for key 30?



BST – Insertion

- We can insert the new node where the search fell off the tree.



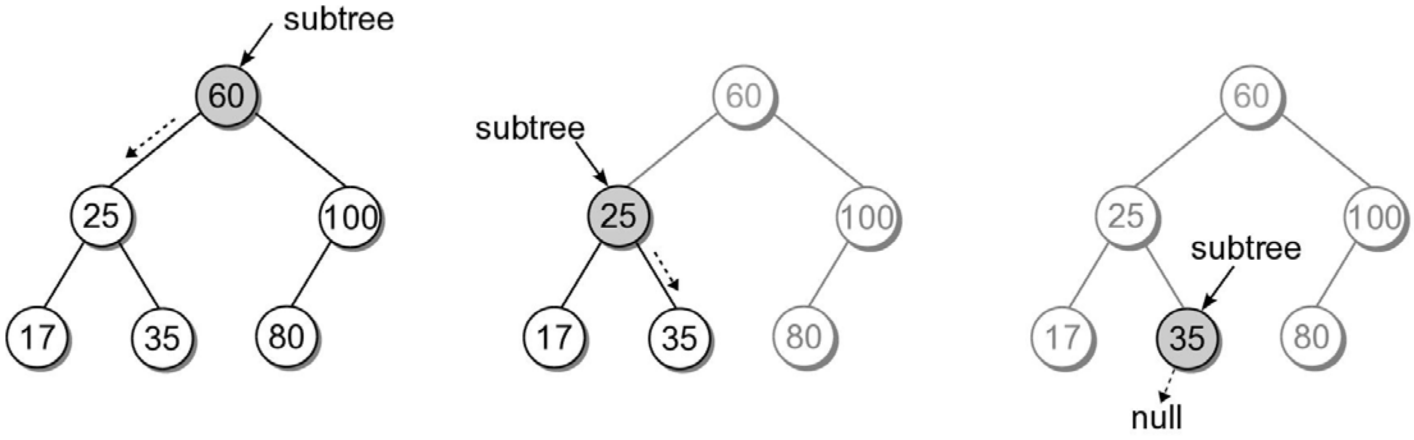
BST – Insert Implementation

bstmap.py

```
class BSTMap :
# ...
  def add( self, key, value ):
    node = self._bstSearch( key )
    if node is not None :
      node.value = value
      return False
    else :
      self._root = self._bstInsert( self._root, key, value )
      self._size += 1
      return True

  def _bstInsert( self, subtree, key, value ):
    if subtree is None :
      subtree = _BSTMapNode( key, value )
    elif key < subtree.key :
      subtree.left = self._bstInsert(subtree.left, key, value)
    elif key > subtree.key :
      subtree.right = self._bstInsert(subtree.right, key, value)
    return subtree
```

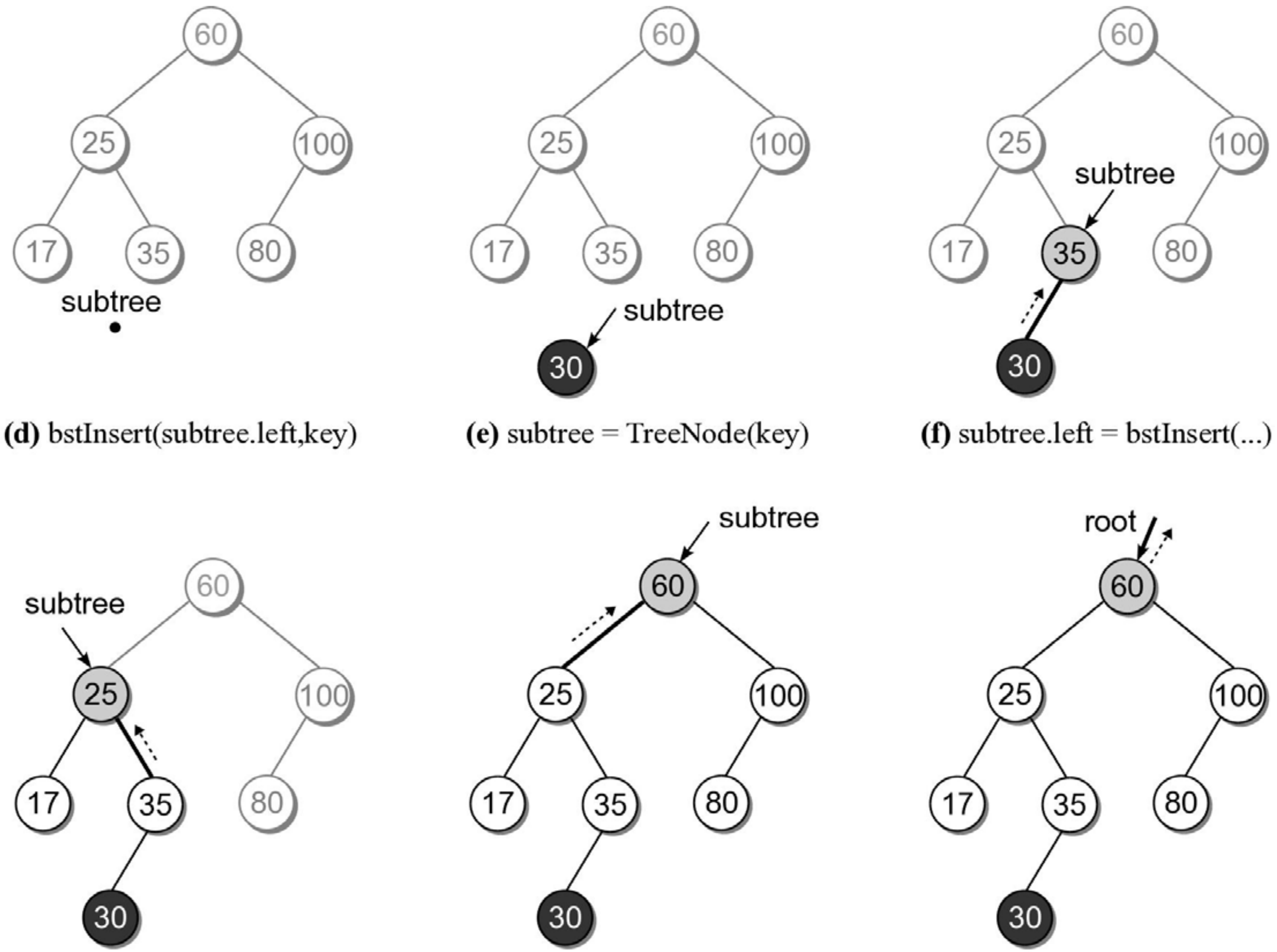
BST – Insert Steps

- Add 30 to our sample BST.



(a) bstInsert(root,30) **(b)** bstInsert(subtree.left,key) **(c)** bstInsert(subtree.right,key)

BST – Insert Steps



(d) bstInsert(subtree.left,key)
(e) subtree = TreeNode(key)
(f) subtree.left = bstInsert(...)
(g) subtree.right = bstInsert(...)
(h) subtree.left = bstInsert(...)
(i) root = bstInsert(...)

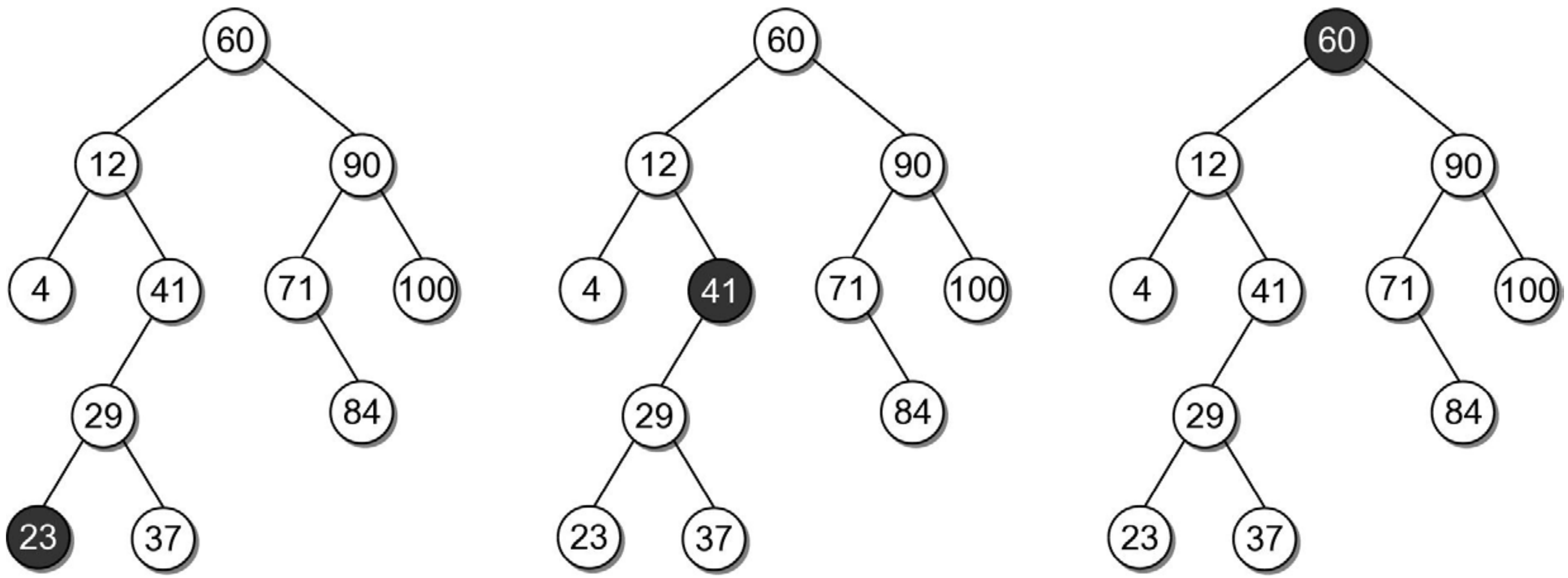
BST – Deletions

- Deleting a node from a BST is a bit more complicated.
  - Locate the node containing the node.
  - Delete the node.

- When a node is removed, the remaining nodes must preserve the search tree property.

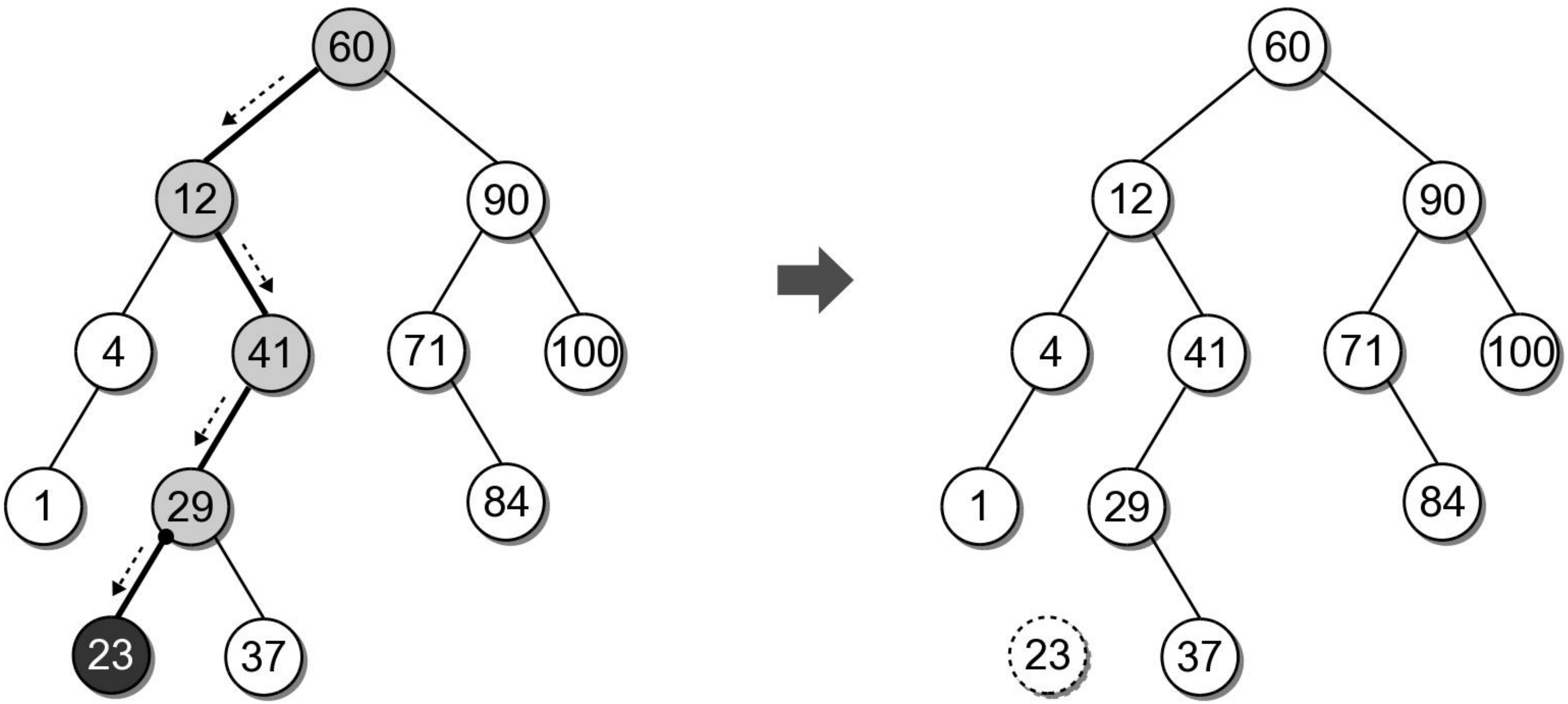
BST – Deletions

- There are three cases to consider:
  - the node is a leaf.
  - the node has a single child
  - the node has two children.



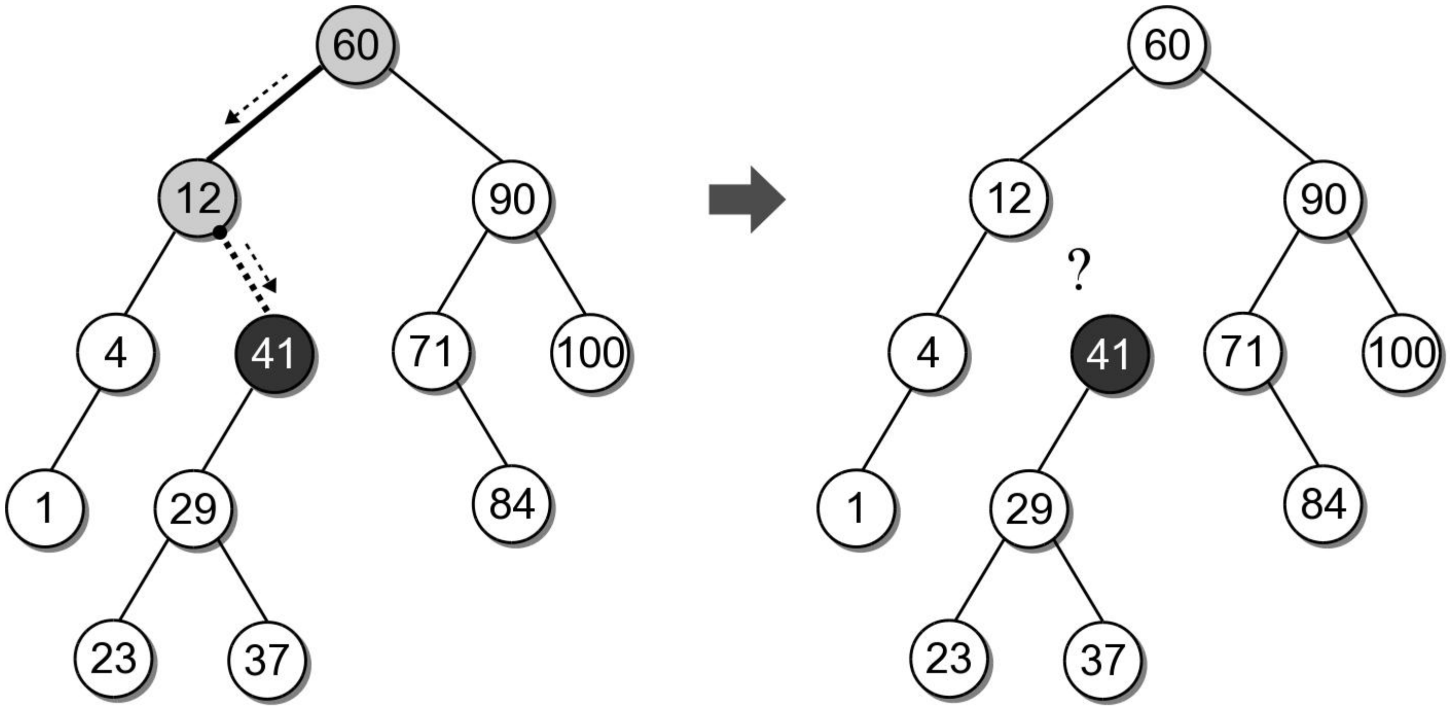
BST – Delete Leaf Node

- Removing a leaf node is the easiest case.
  - Suppose we want to remove 23.



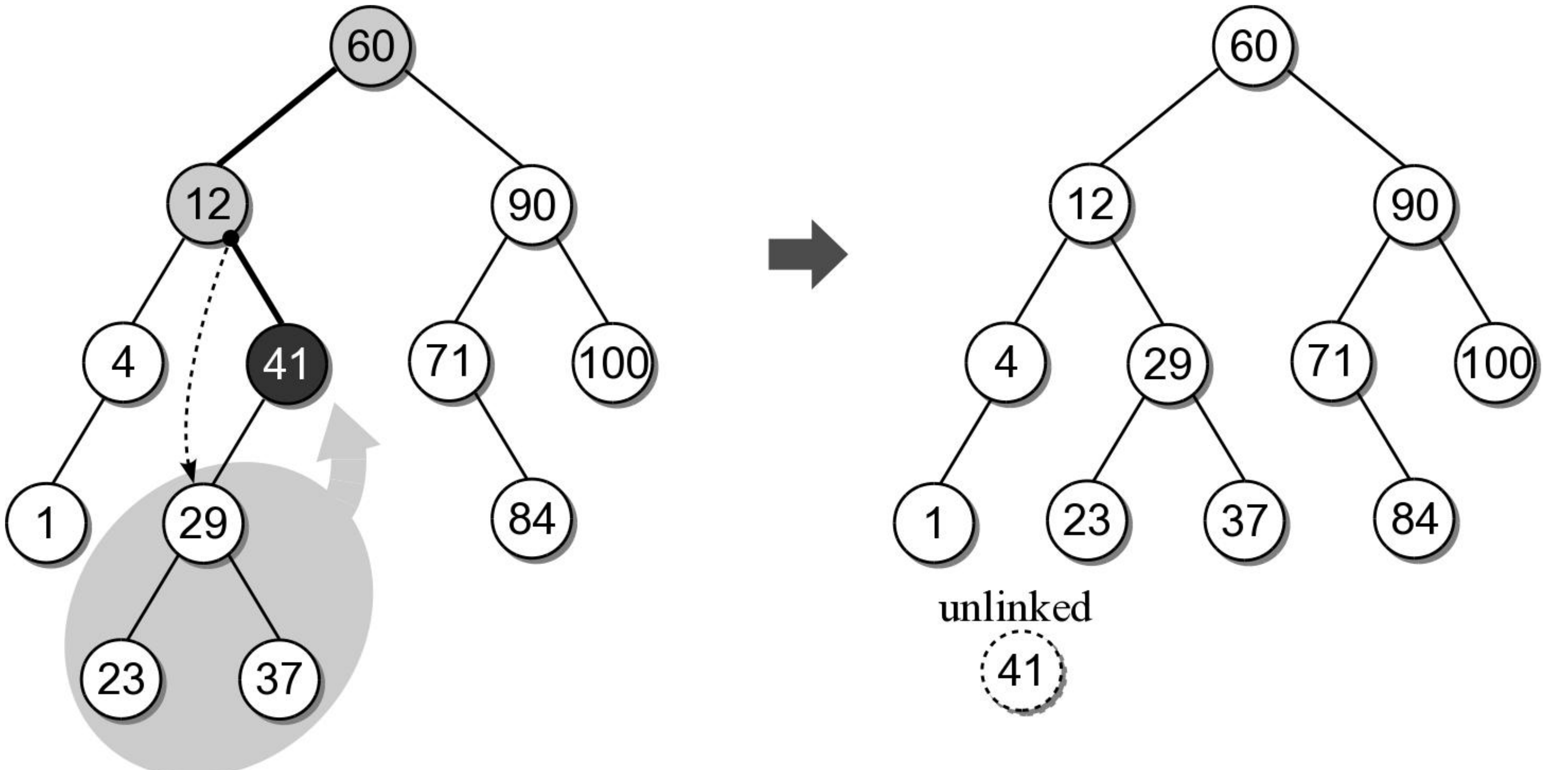
BST – Delete Interior Node

- Removing an interior node with one child.
  - Suppose we want to remove 41.
  - We can not simply unlink the node.



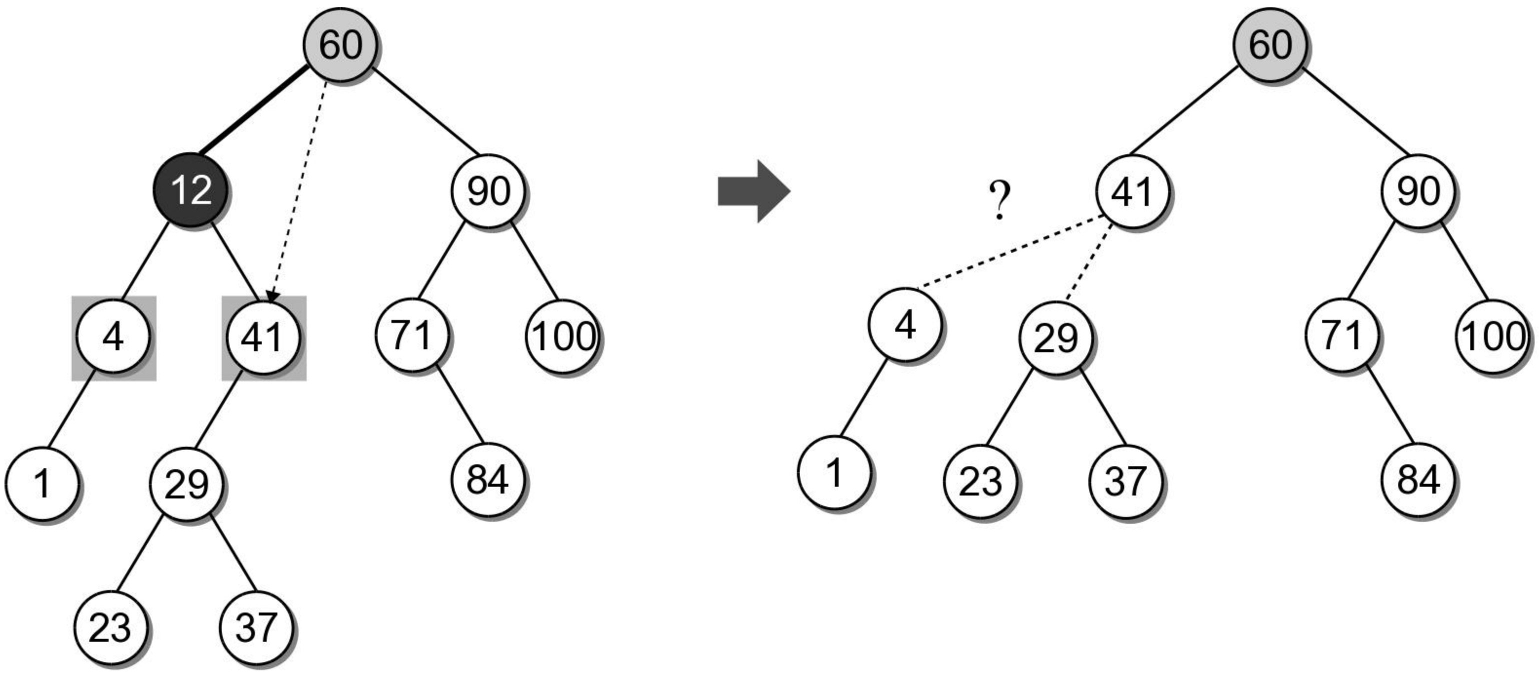
BST – Delete Interior Node

- After locating the node to be removed, it's child must be linked to it's parent.



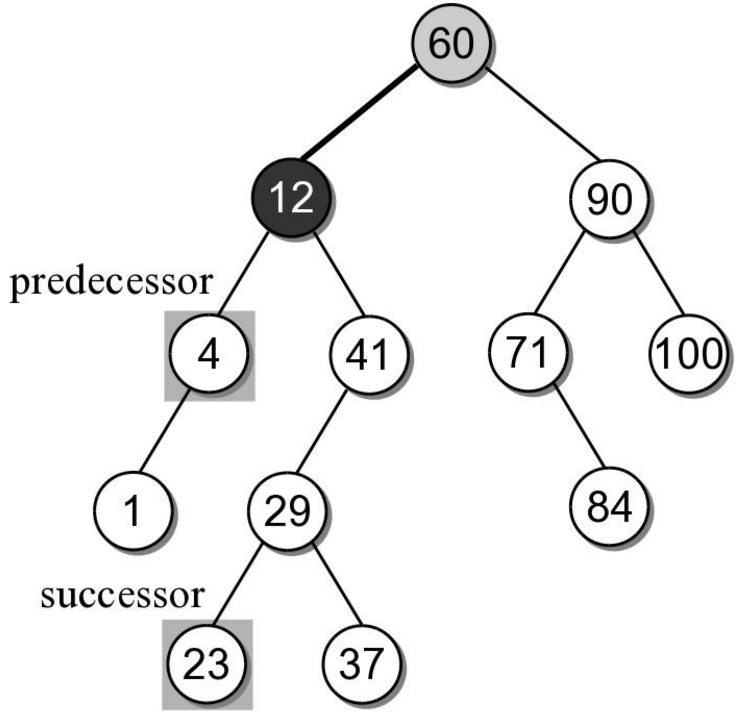
BST – Delete Interior Node

- The most difficult case is deleting a node with two children.
  - Suppose we want to delete node 12.
  - Which child should be linked to the parent?



BST – Delete Interior Node

- Based on the search tree property, each node has a logical predecessor and successor.
  - For node 12, those are 4 and 23.

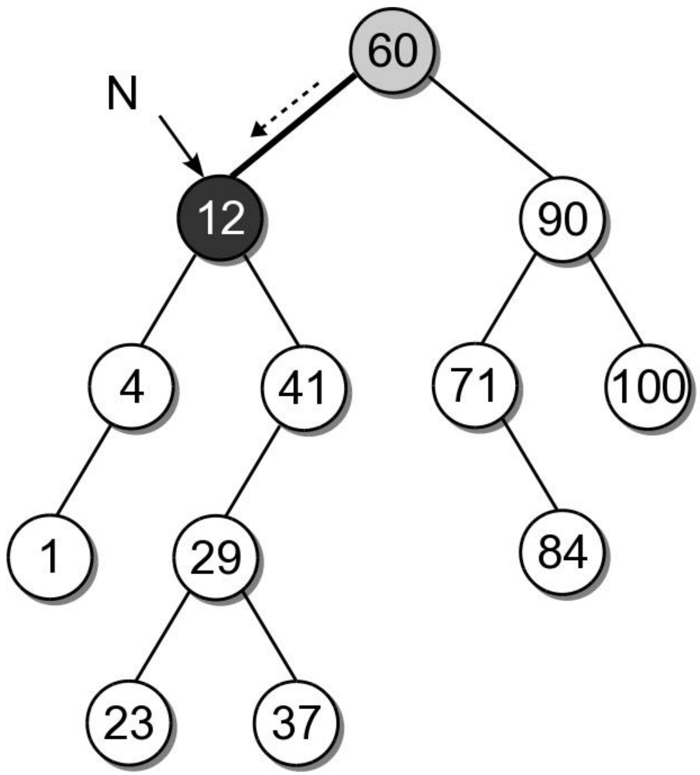


BST – Delete Interior Node

- We can replace to be deleted with either its logical successor or predecessor.
  - Both will either be a leaf or an interior node with one child.
  - We already know how to remove those nodes.

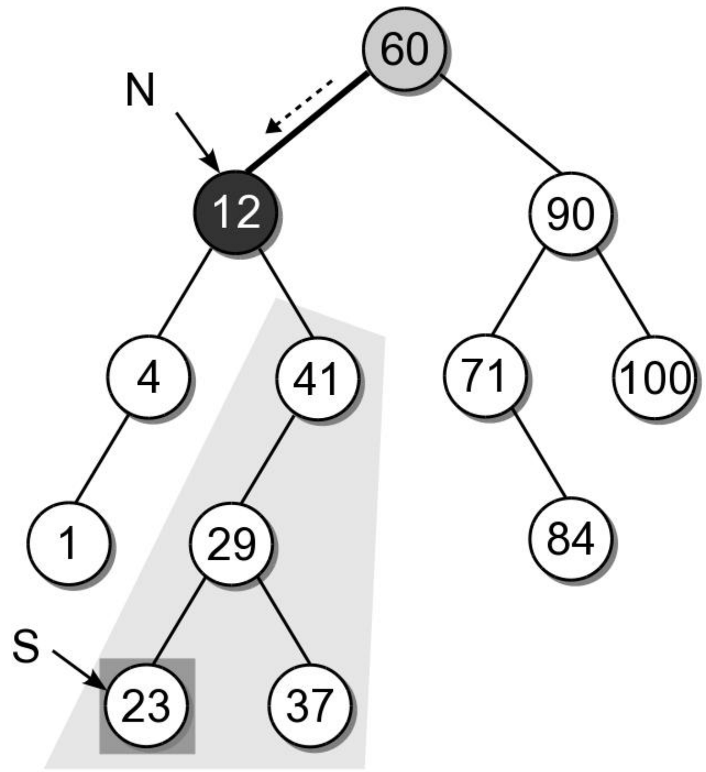
BST – Delete Interior Node

- Removing an interior node with two children requires 4 steps:
  - (1) Find the node to be deleted, N.



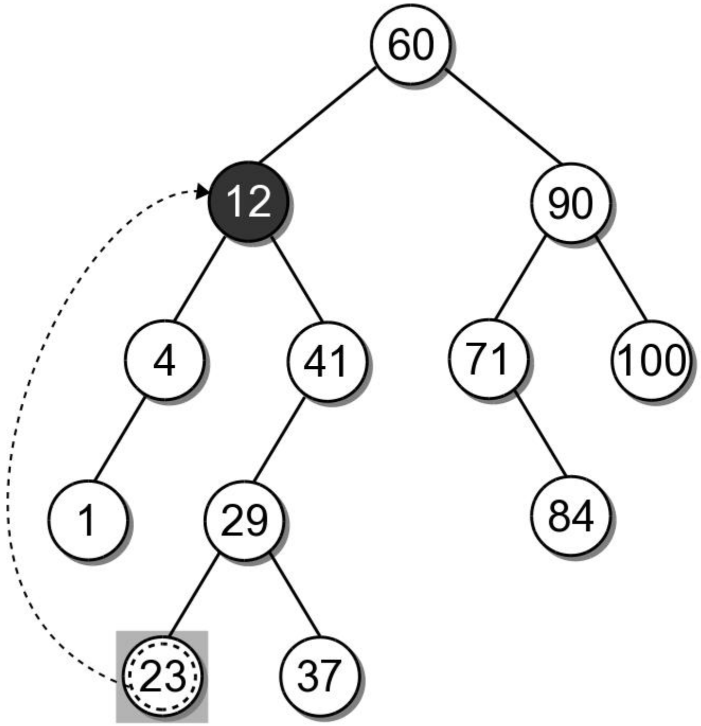
BST – Delete Interior Node

- (2) Find the successor, S, of node N.



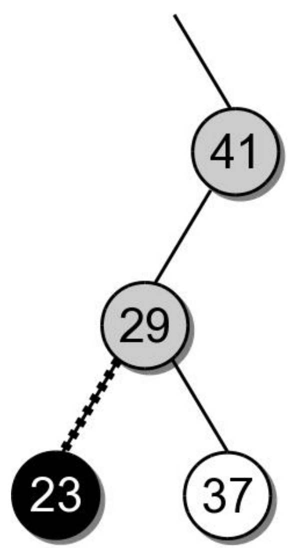
BST – Delete Interior Node

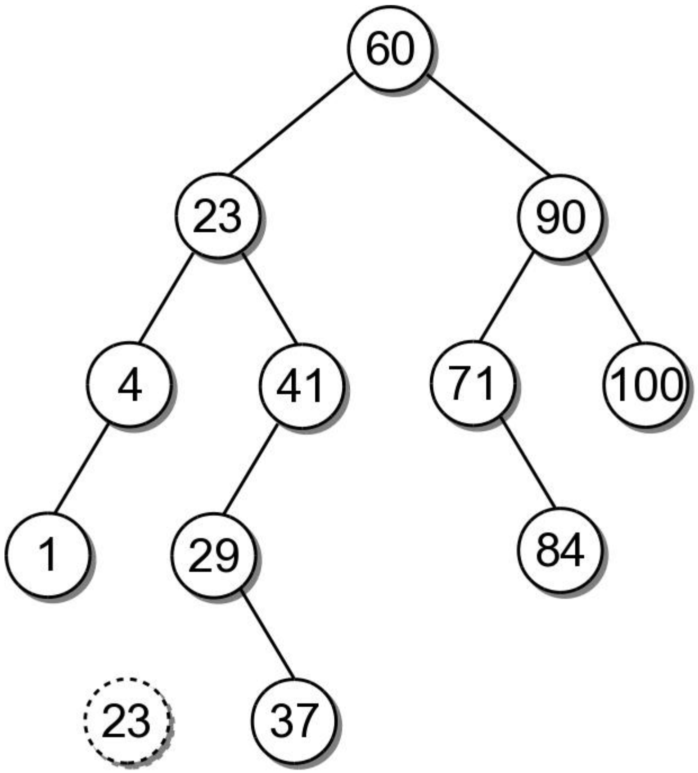
- (3) Copy the payload from node S to node N.



BST – Delete Interior Node

- (4) Remove node S from the tree.





BST – Delete Interior Node

- Removing an interior node with two children requires 4 steps:
  - Find the node to be deleted, N.
  - Find the logical successor, S, of node N.
  - Copy the payload from node S to node N.
  - Remove node S from the tree.

BST – Delete Implementation

bstmap.py

```
class BSTMap :
# ...
  def remove( self, key ):
    assert key in self, "Invalid map key."
    self._root = self._bstRemove( self._root, key )
    self._size -= 1
```

BST – Delete Implementation

bstmap.py

```
class BSTMap :
# ...
  def _bstRemove( self, subtree, target ):
    if subtree is None :
      return subtree
    elif target < subtree.key :
      subtree.left = self._bstRemove( subtree.left, target )
      return subtree
    elif target > subtree.key :
      subtree.right = self._bstRemove( subtree.right, target )
      return subtree
    else :
        ......
```

BST – Delete Implementation

bstmap.py

```
class BSTMap :
# ...
  def _bstRemove( self, subtree, target ):
         ......
    else :
      if subtree.left is None and subtree.right is None :
        return None
      elif subtree.left is None or subtree.right is None :
        if subtree.left is not None :
          return subtree.left
        else :
          return subtree.right
      else
        successor = self._bstMinimum( subtree.right )
        subtree.key = successor.key
        subtree.value = successor.value
        subtree.right = self._bstRemove( subtree.right,
                                         successor.key )
        return subtree
```

BST – Efficiency

Operation	Worst Case
_bstSearch(root, k)	O(n)
_bstMinimum(root)	O(n)
_bstInsert(root, k)	O(n)
_bstDelete(root, k)	O(n)
traversal	O(n)

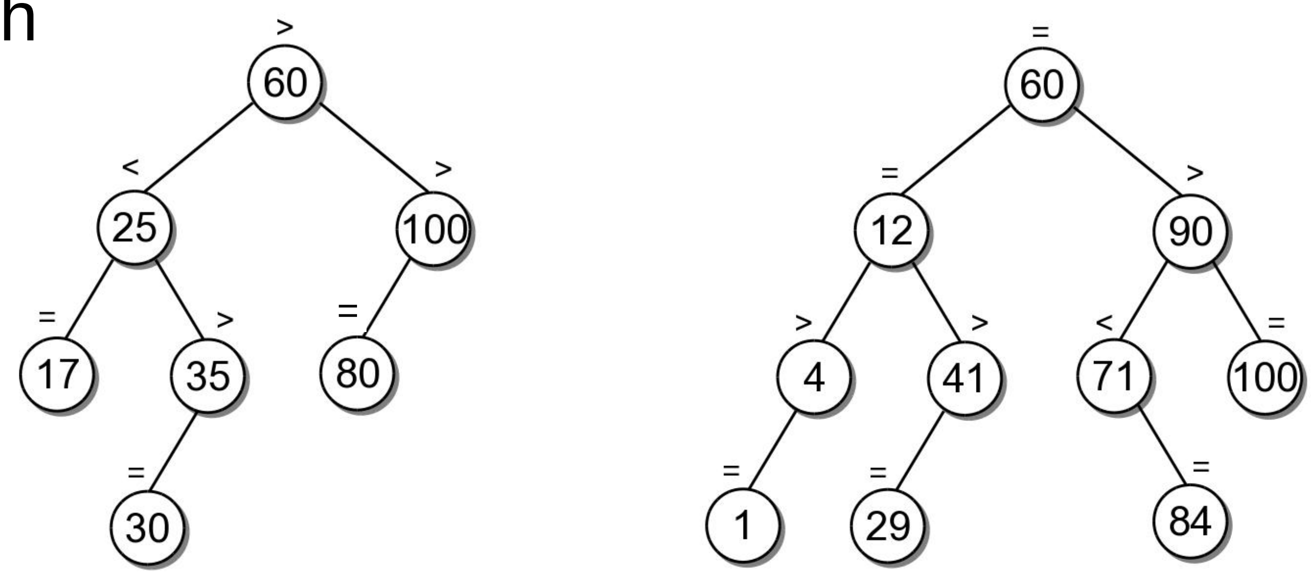
Balanced Binary Search Tree (AVL)

- Improves on the binary search tree by always guaranteeing the tree is height balanced.
  - Allows for more efficient operations.
  - Developed by **A**del'son-Velskhii and **L**andis in 1962.

- **balanced** – the heights of the left and right subtrees of every node differ by at most 1.

Balance Factor

- Associated with each node and indicates the height difference between the left and right branch.
  - left-high
  - equal-high
  - right-high

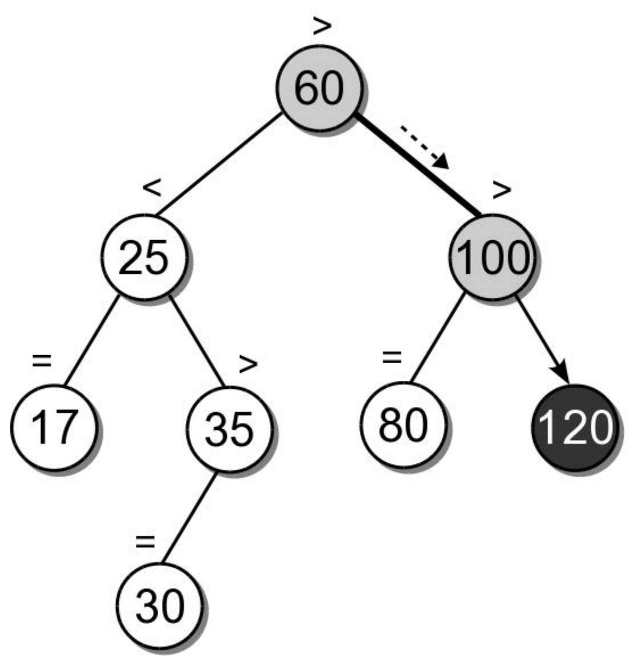


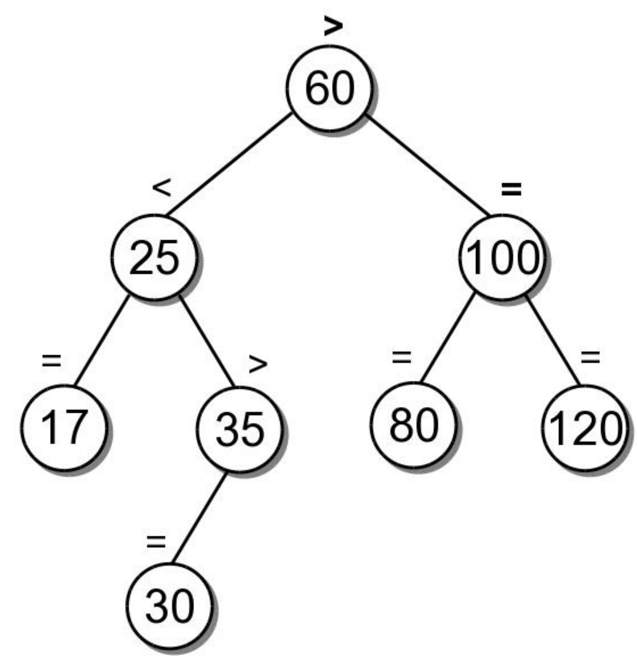
AVL Operations

- Search and traversal are the same as with the binary search tree.
- Insertion and deletion must be modified.
  - Maintain the balance property as keys are added and/or removed.
  - Ensures height never exceeds *1.44 log n*.
  - Provides *O(log n)* worst case time.

AVL Insertions

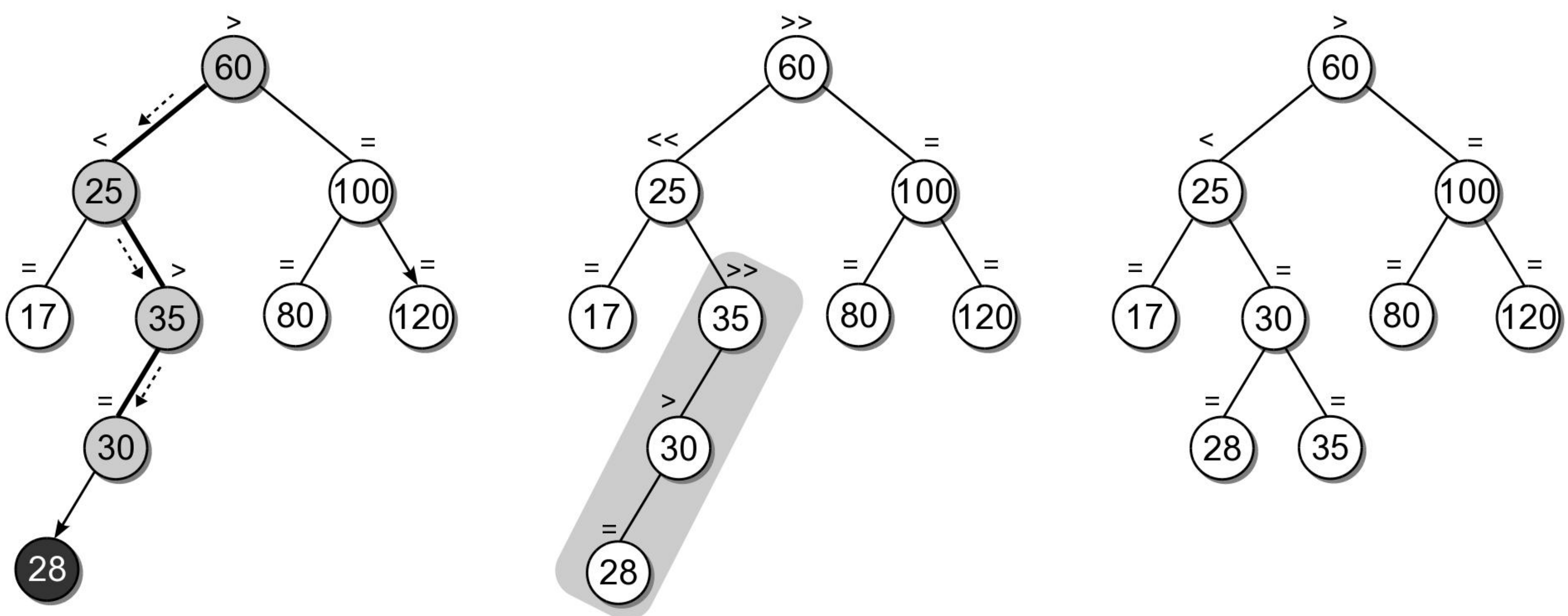
- Begins with the same process used with a BST.
  - Must rebalance the tree if the insertion causes it to become unbalanced.
  - Example: insert key 120





AVL Insert Example

- Suppose we add key 28 to the AVL tree.



AVL Rotation

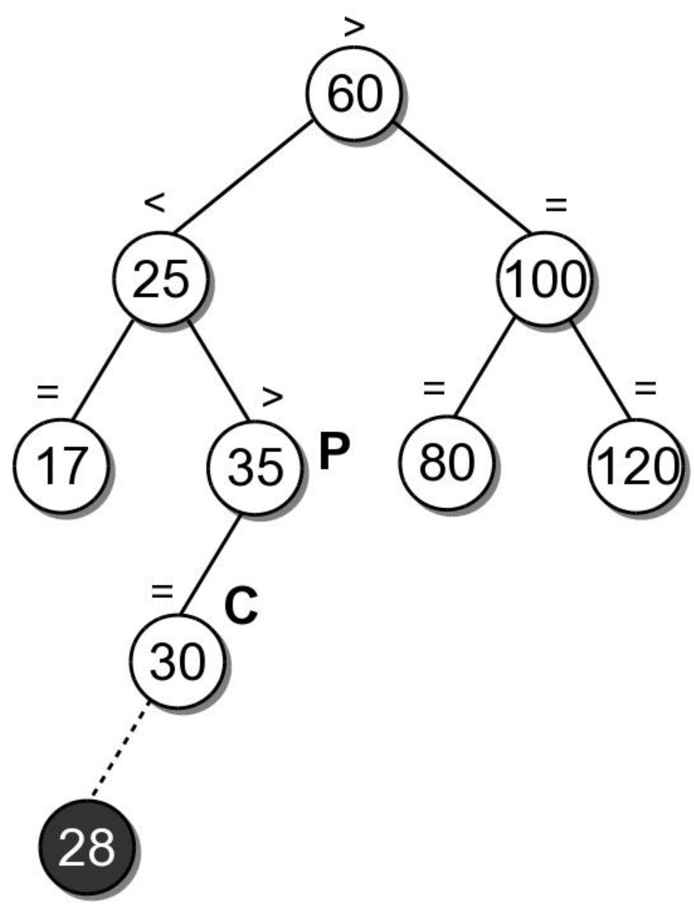
- Multiple subtrees can become unbalanced after an inserting a new key.
  - Limited to the nodes along the insertion path.
  - Balance factors are adjusted during the recursion unwinding.
  - **pivot node** – root node of the first out of balance subtree encountered.

AVL Rotation

- An AVL subtree is rebalanced by performing a **rotation** around the pivot node.
  - Rearrange links of the pivot node, its children and at most one of its grandchildren.
  - There are four possible cases.

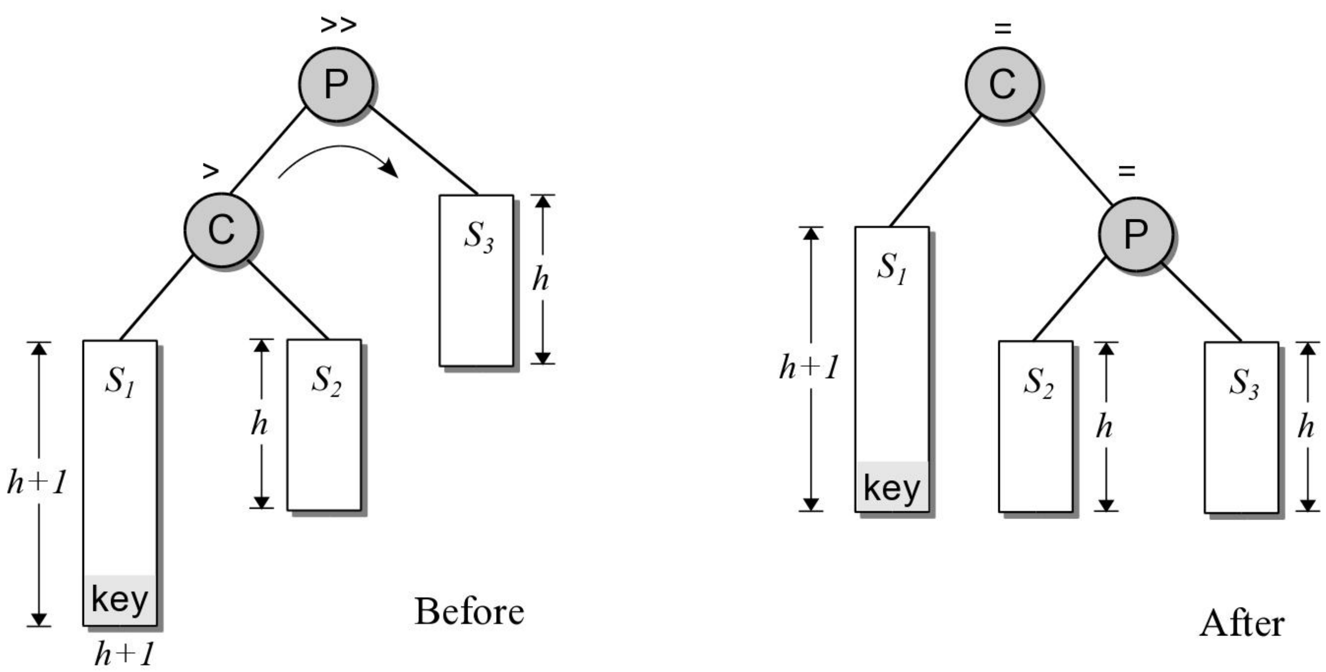
AVL Rotation (Case 1)

- The balance factor of the pivot node (P) is left-high before the insertion into the left subtree (C) of P.



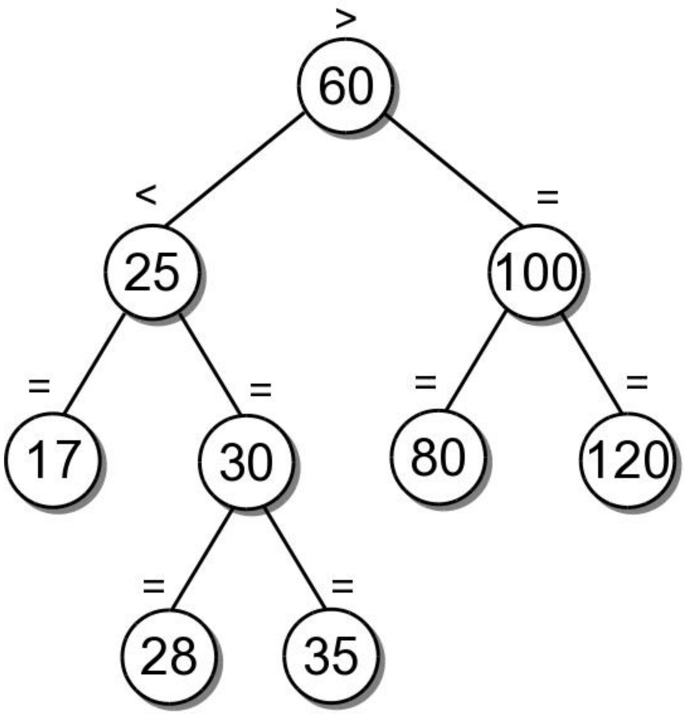
AVL Rotation (Case 1)

- The pivot node has to be rotated right over its left child.
  - P becomes the right child of C.
  - Right child of C becomes the left child of P.



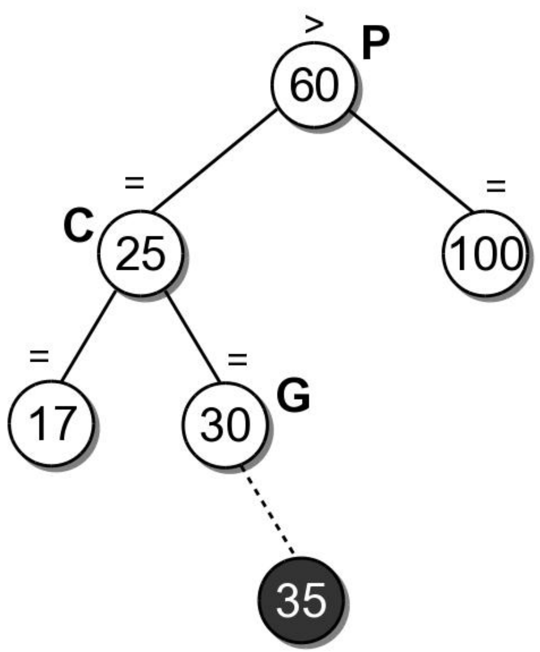
AVL Rotation (Case 1)

- Result after the left rotation.



AVL Rotation (Case 2)

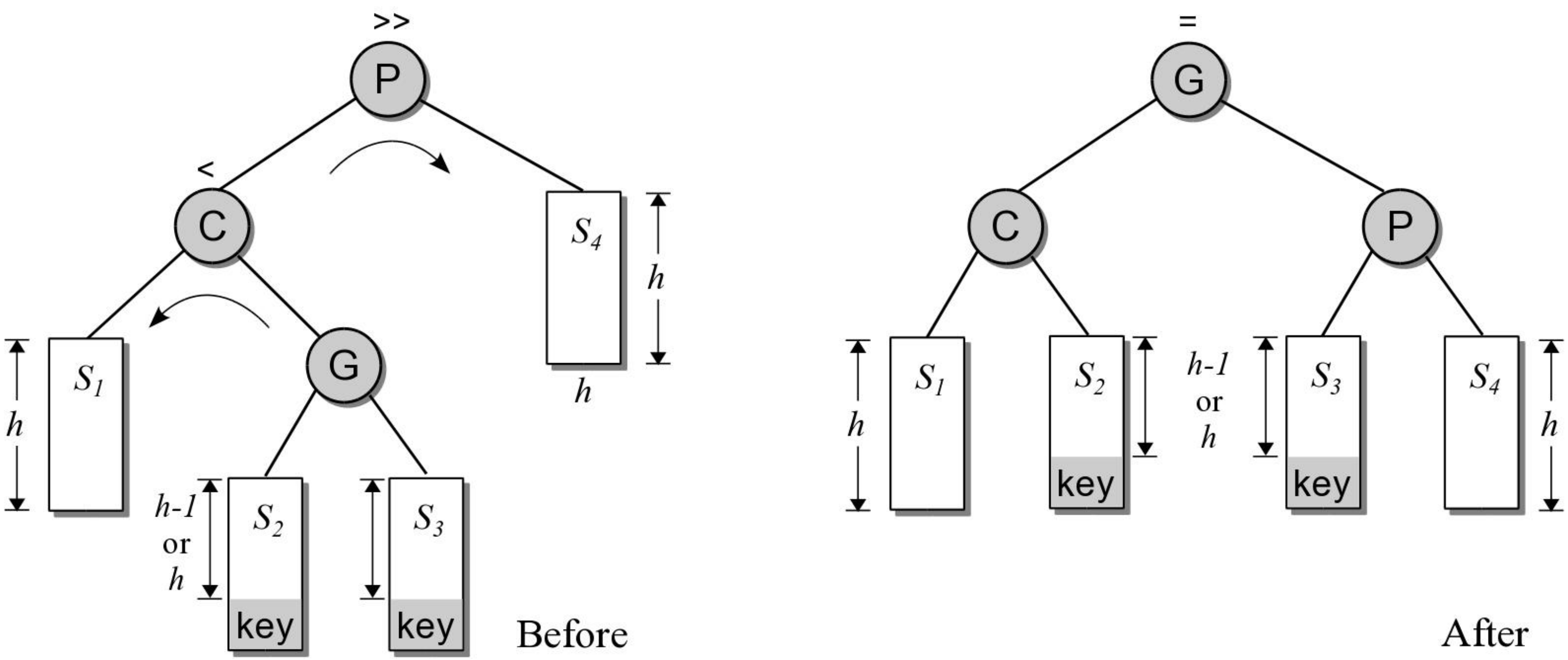
- Involves three nodes: pivot (P), left child (C) of P and the right child (G) of C.
  - Balance factor of P is left-high before the insertion.
  - Inserted into either left or right subtree of G.



AVL Rotation (Case 2)

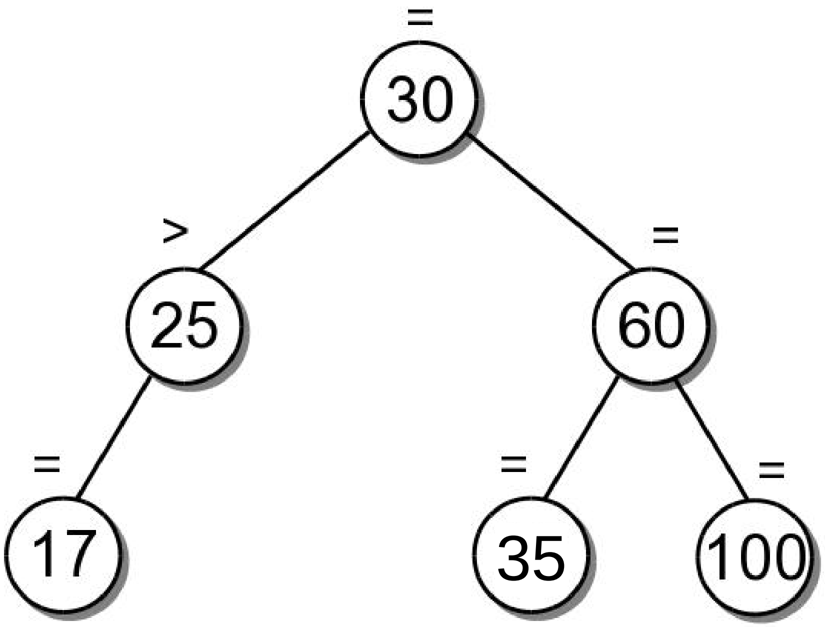
- Requires a double rotation:
  - node C has to be rotated left over node G.
  - pivot node has to be rotated right over its left child.
- Link modifications:
  - right child of G becomes the new left child of P.
  - left child of G becomes the new right child of C.
  - C becomes the new left child of G.
  - P becomes the new right child of G.

AVL Rotation (Case 2)



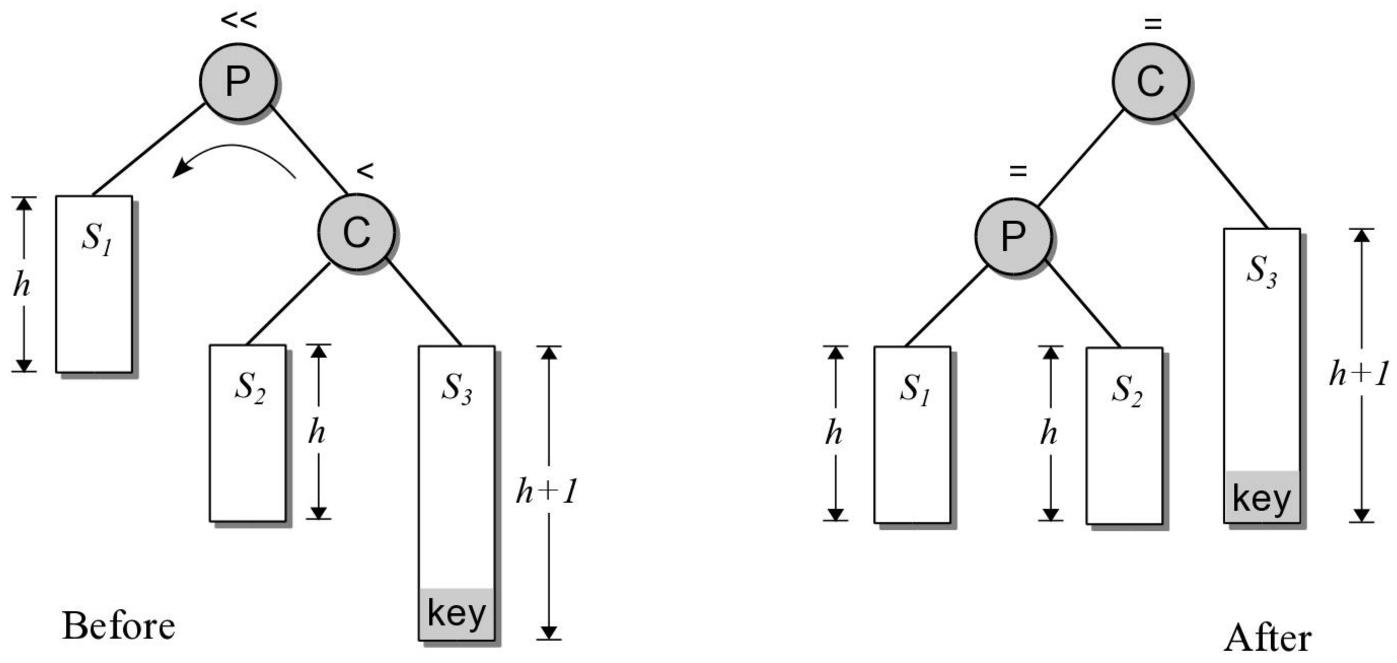
AVL Rotation (Case 2)

- Result after the two rotations.



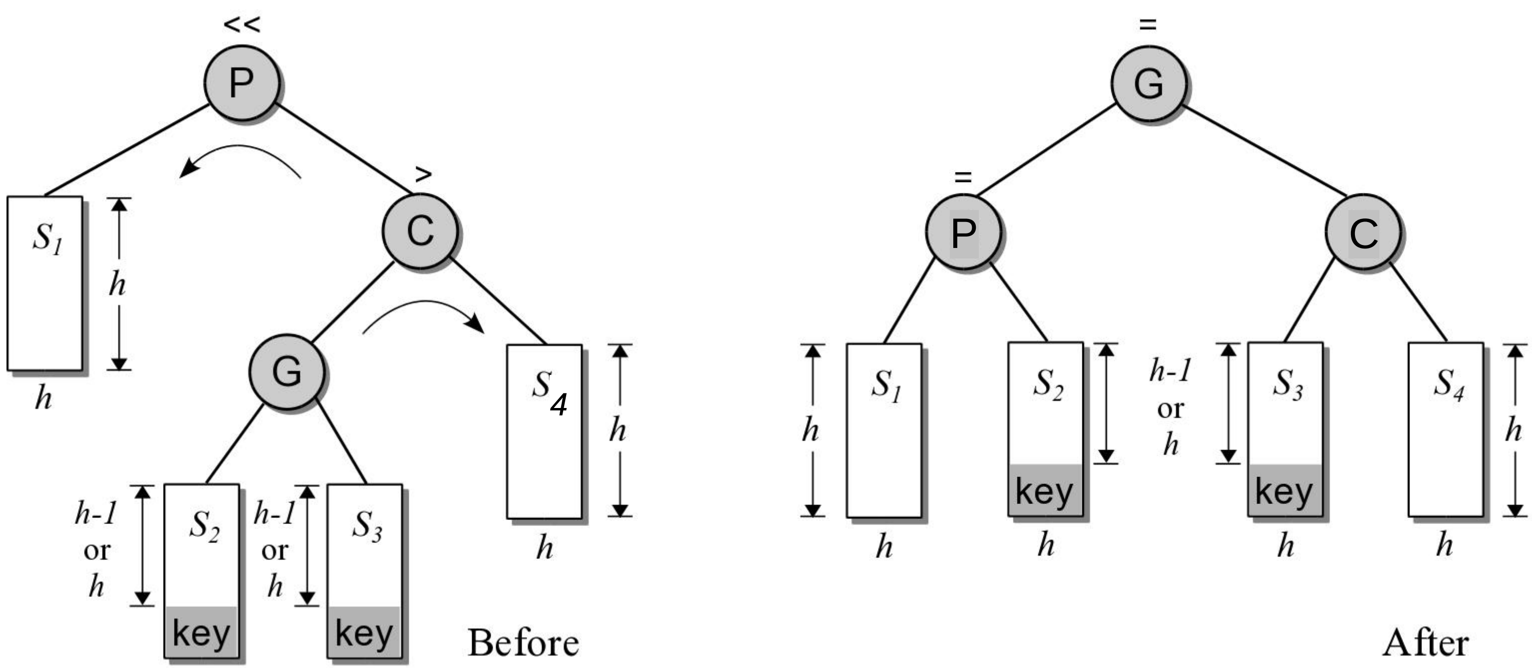
AVL Rotation (Case 3)

- A mirror image of the first case.
  - P is right-high.
  - The new key is inserted in the right subtree of C.



AVL Rotation (Case 4)

- A mirror image of the second case.
  - P is right-high.
  - G is the left child of C instead of the right.



New Balance Factors

- The balance factors of the nodes along the insertion path may have to be modified.
  - Performed in reverse order as the recursion unwinds.
- The new balance factor of a node depends on its current factor and the subtree into which the new node was inserted.

current factor	left subtree	right subtree
>	>>	=
=	>	<
<	=	<<

New Balance Factors

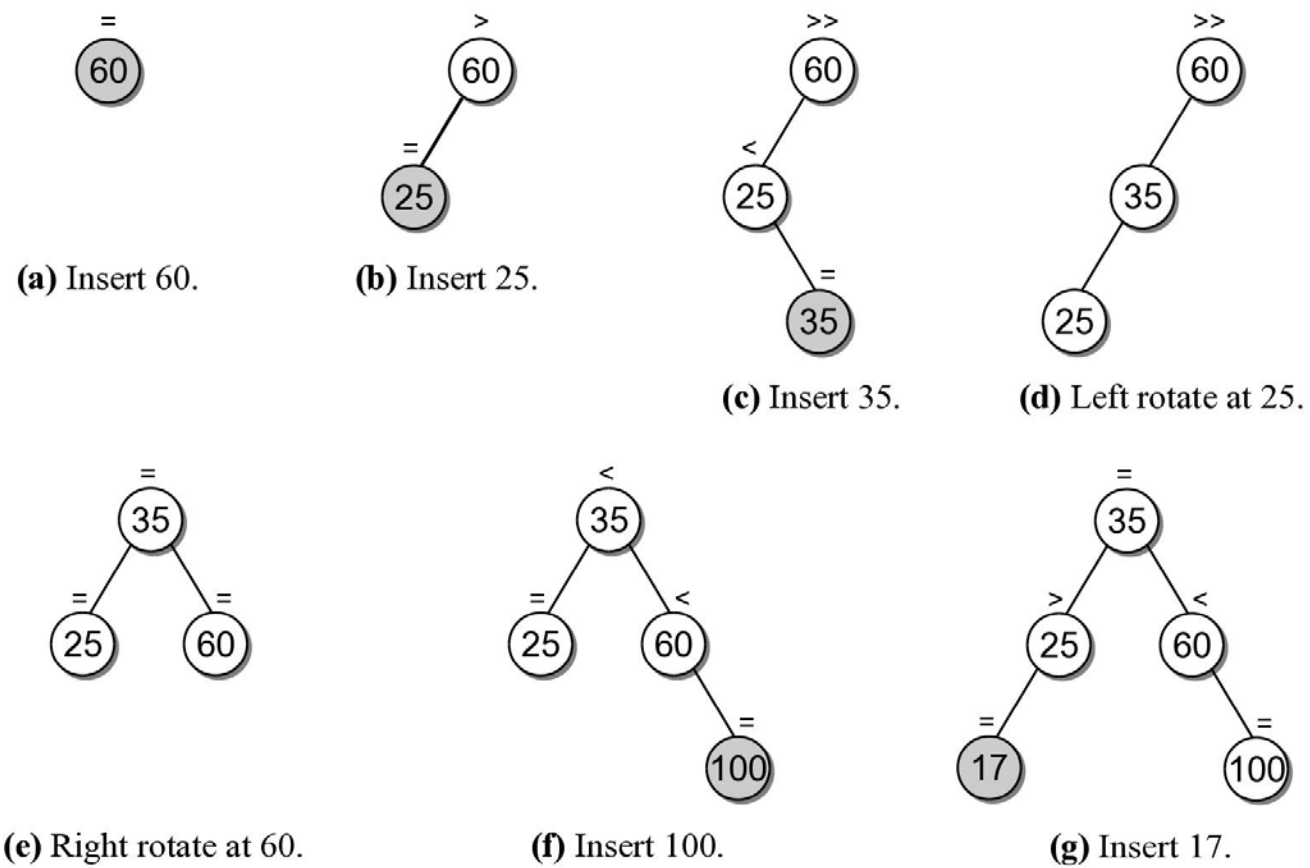
- The balance factors of the nodes impacted by a rotation have to be modified.
  - The new balance factors depends on the case that triggered the rotation.

	G	new P	new L	new R	new G
case 1		=	=		
case 2	>	<	=		=
	=	=	=		=
	<	=	>		=
case 3		=		=	
case 4	>	=		=	<
	=	=		=	=
	<	=		=	>

Building an AVL Tree

- Suppose we want to build an AVL tree from the list.

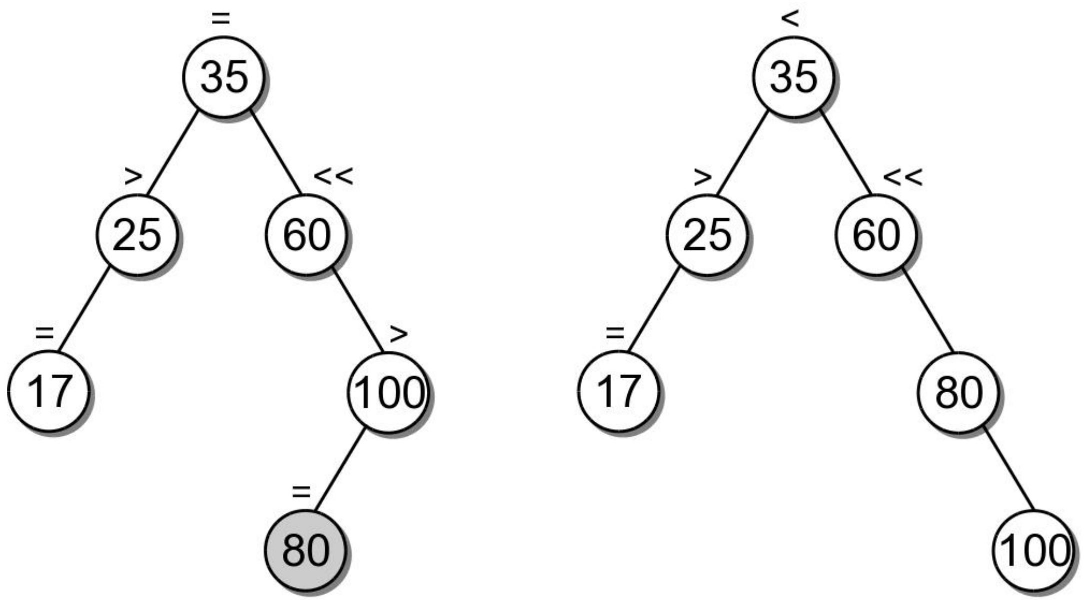
```
60  25  100  35  17  80
```



(a) Insert 60. (b) Insert 25. (c) Insert 35. (d) Left rotate at 25.

(e) Right rotate at 60. (f) Insert 100. (g) Insert 17.

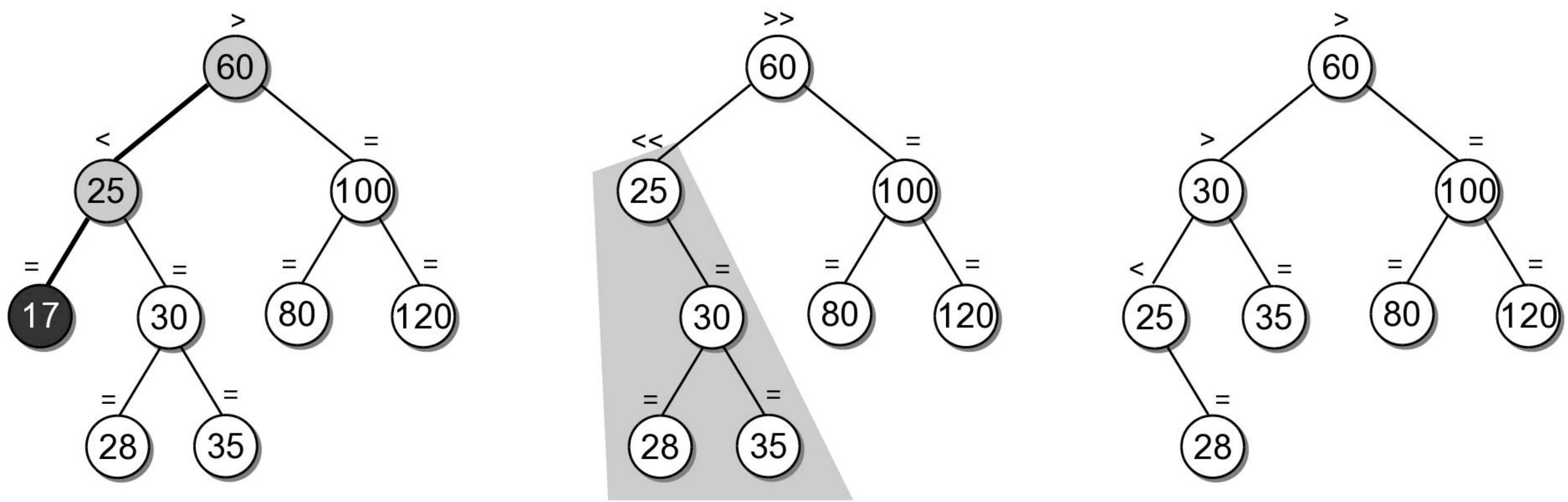
Building an AVL Tree



(h) Insert 80. **(i)** Right rotate at 100. **(j)** Left rotate at 60.

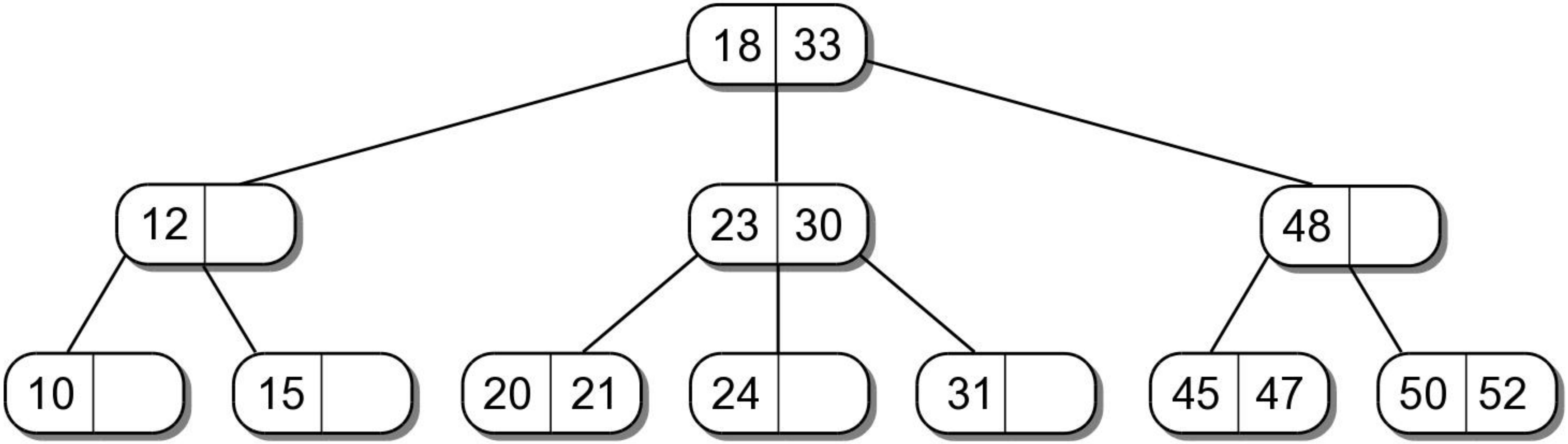
AVL Deletions

- When an entry is removed, we must ensure the balance property is maintained.
  - Use the same technique as with a BST.
  - After the node is removed, subtrees may have to be rebalanced.



2-3 Trees

- A multi-way search tree that can have up to three children.
  - Provides fast operations.
  - Gets its name from the max number of keys (2) and the max number of children (3) each node can have.

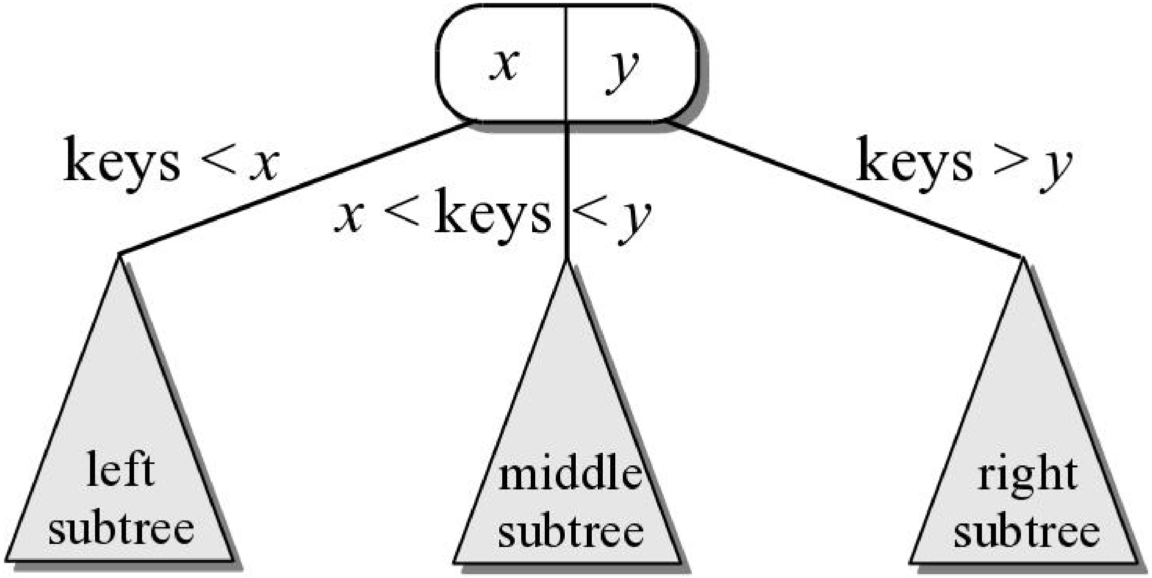


2-3 Tree Definition

- A search tree that is always balanced and whose shape and structure is as follows:
  - Every node has capacity for one or two keys.
  - Every node has capacity for up to three children.
  - All leaf nodes are at the same level.
  - Every interior node must contain two or three children.
    - one key = two children
    - two keys = three children

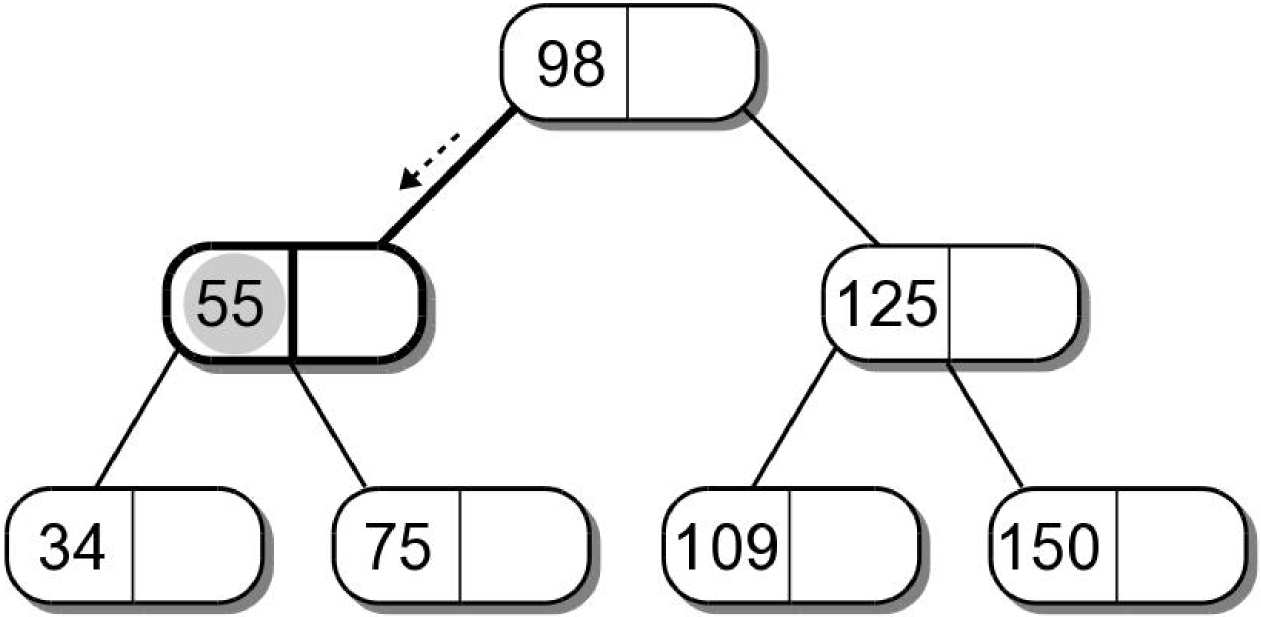
2-3 Tree Search Property

- The organization of the keys in a 2-3 tree are based on the search property.



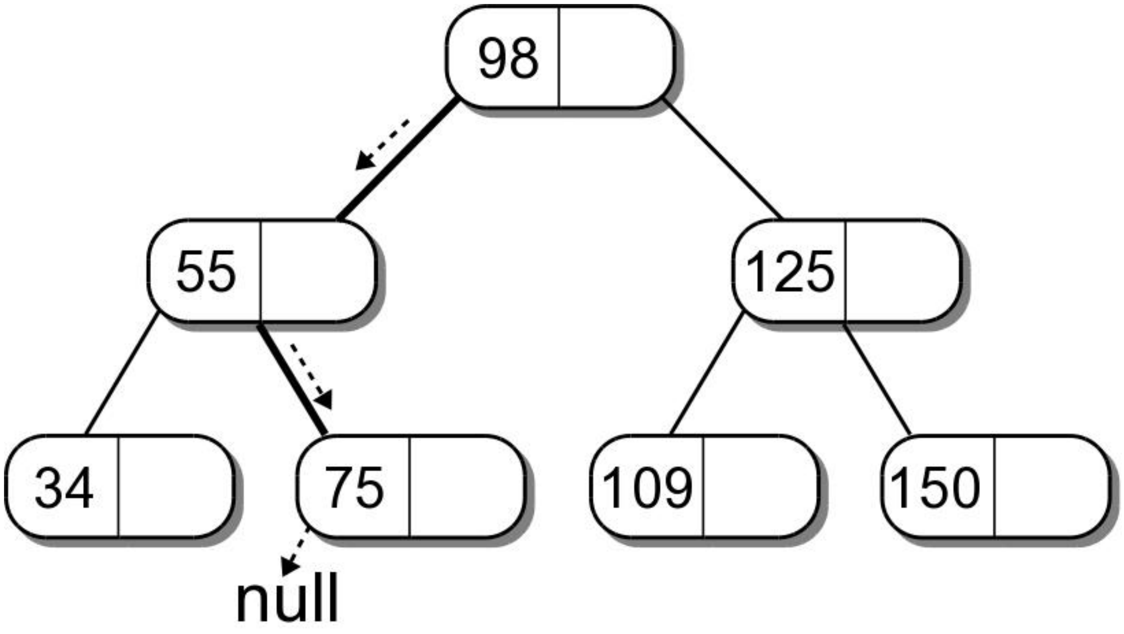
2-3 Tree Searching

- Searching a 2-3 tree is similar to that of a BST.
  - Start at the root node and follow the appropriate branch.
  - The target has to be compared against both keys.



2-3 Tree Searching

- Searching for a non-existent key is also the same as in a BST.

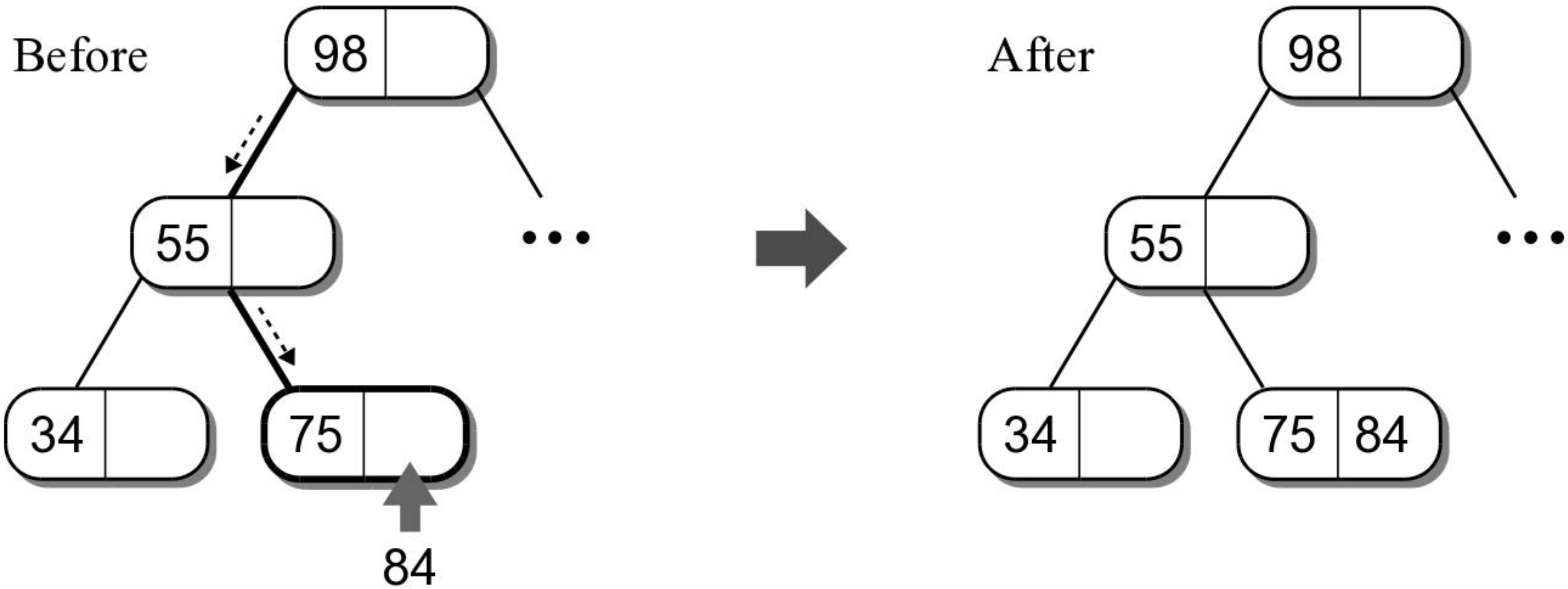


2-3 Tree Insertions

- Inserting a key into a 2-3 tree is similar to a BST, but a bit more complicated.
  - Search for the key as if it were in the tree.
  - If there is space in the leaf for the new key, insert it into the leaf node.

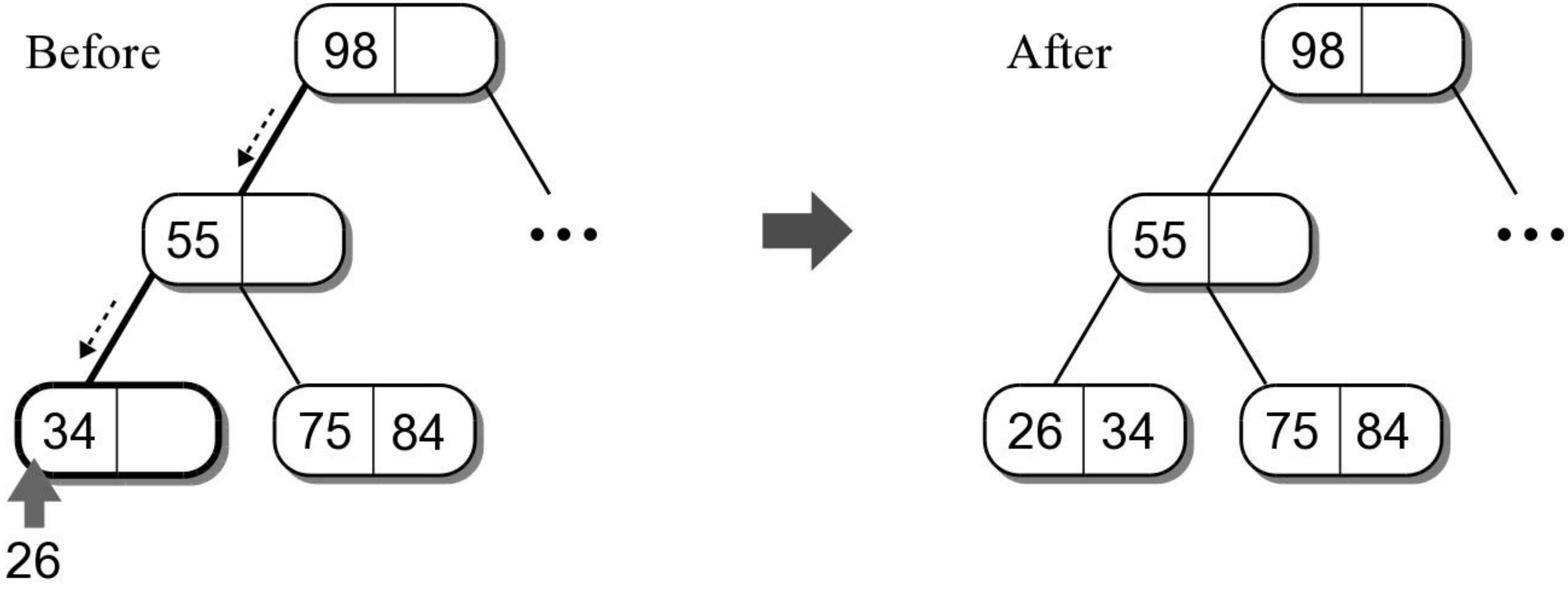
2-3 Tree Insertions

- Insert key 84 into our sample tree.
  - Key 84 is larger than 75 and becomes key2



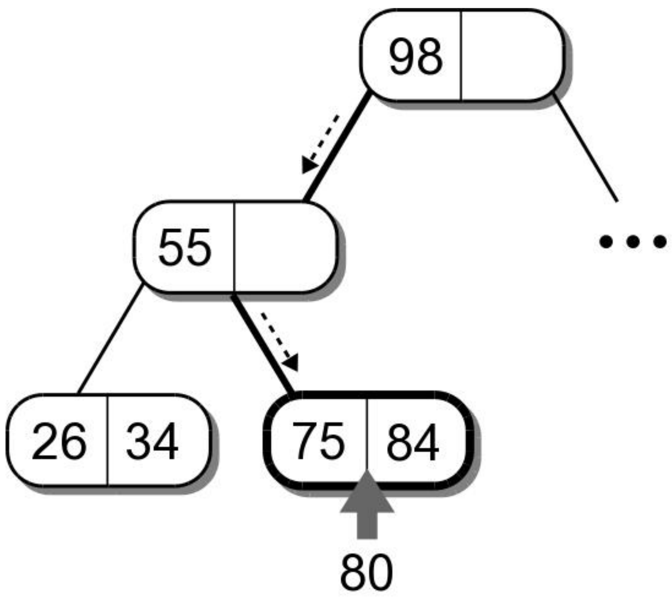
2-3 Tree Insertions

- Insert key 26 into our sample tree.
  - Key 26 is less than 34 and becomes key 1.



2-3 Tree Insertions

- Things become more complicated if the leaf node is full.



Splitting Leaf Node

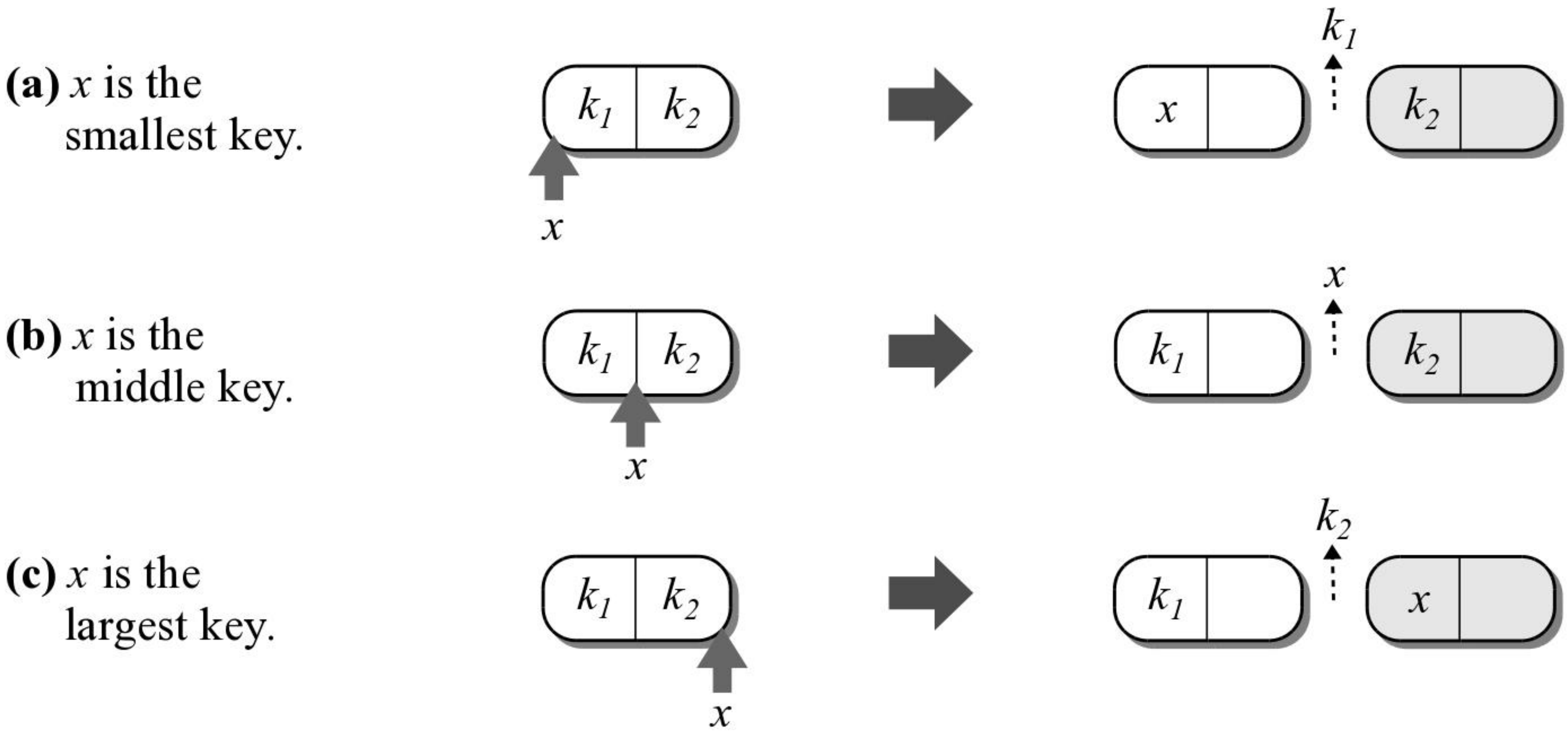
- All leaf nodes must be at the same level.
  - Can not simply create a new node and attach it to the leaf.
  - The leaf node has to be split and inserted at the same level.

Splitting Leaf Node

- The splitting process involves two steps:
  - a new node is created
  - the new key is compared to the two keys in the original leaf node.
    - smallest is inserted into the original node.
    - largest is inserted into the new node.
    - middle value is promoted to the parent along with a reference to the new node.

Splitting Leaf Node

- Illustration of the three cases.

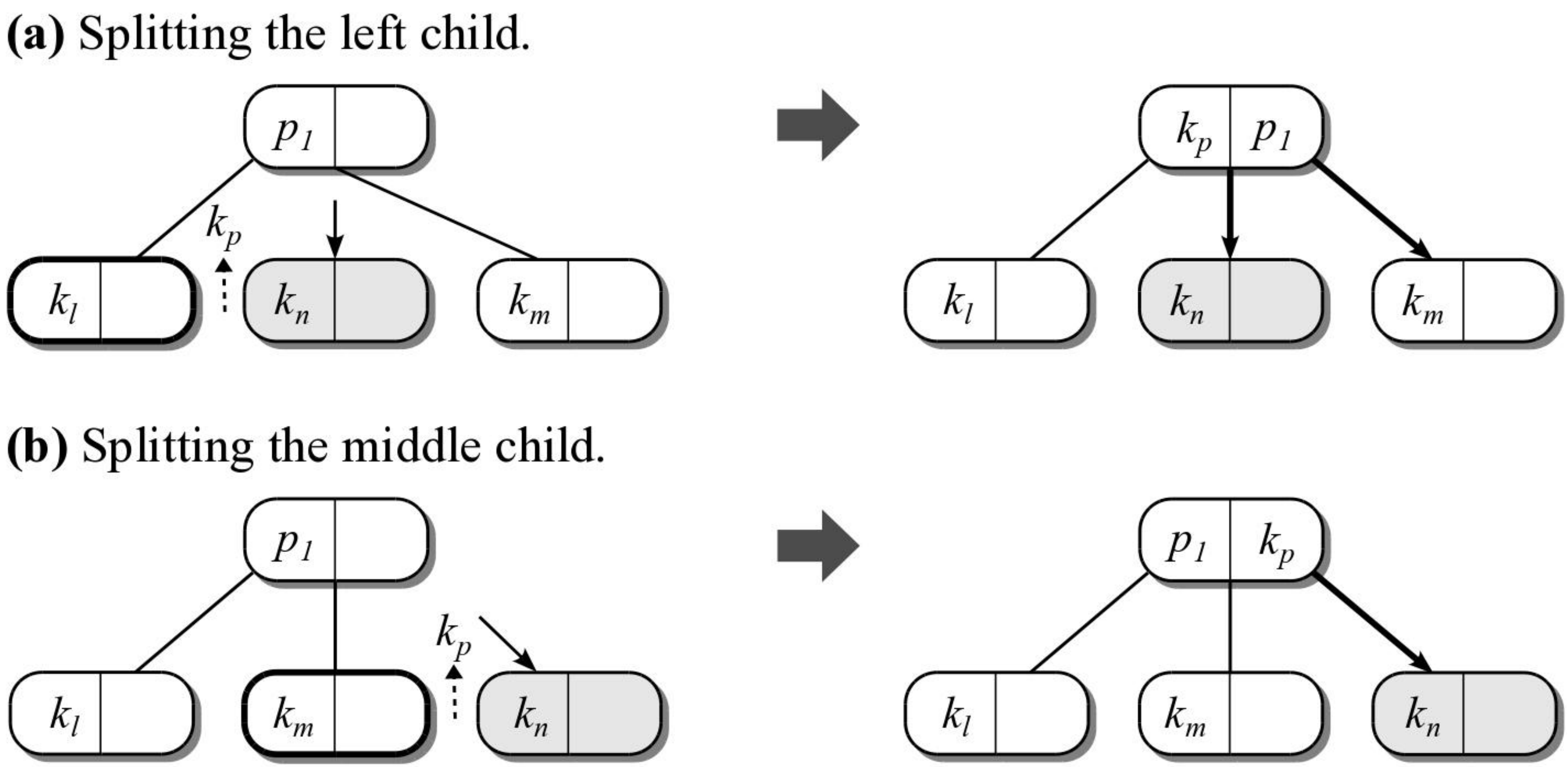


Key Promotion

- When a key is promoted to the parent, it has to be inserted into the parent's node.
  - A link reference to the new node is also passed up.
  - The link also has to be inserted into the parent node.

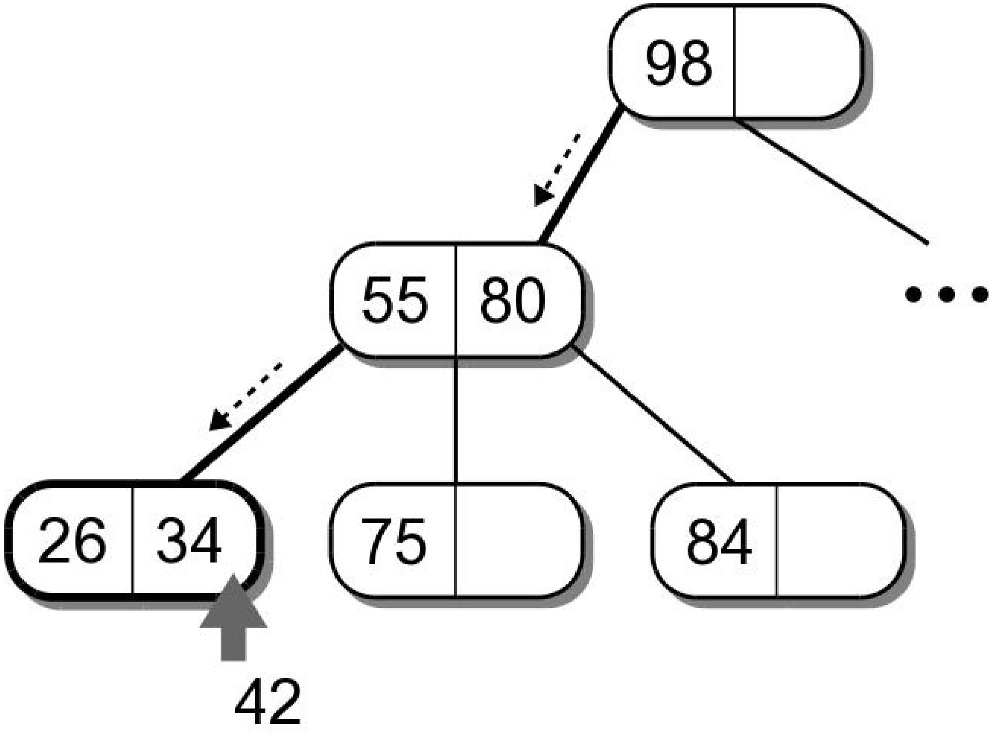
Key Promotion

- Inserting the key and reference is simply if the parent contains a single key.



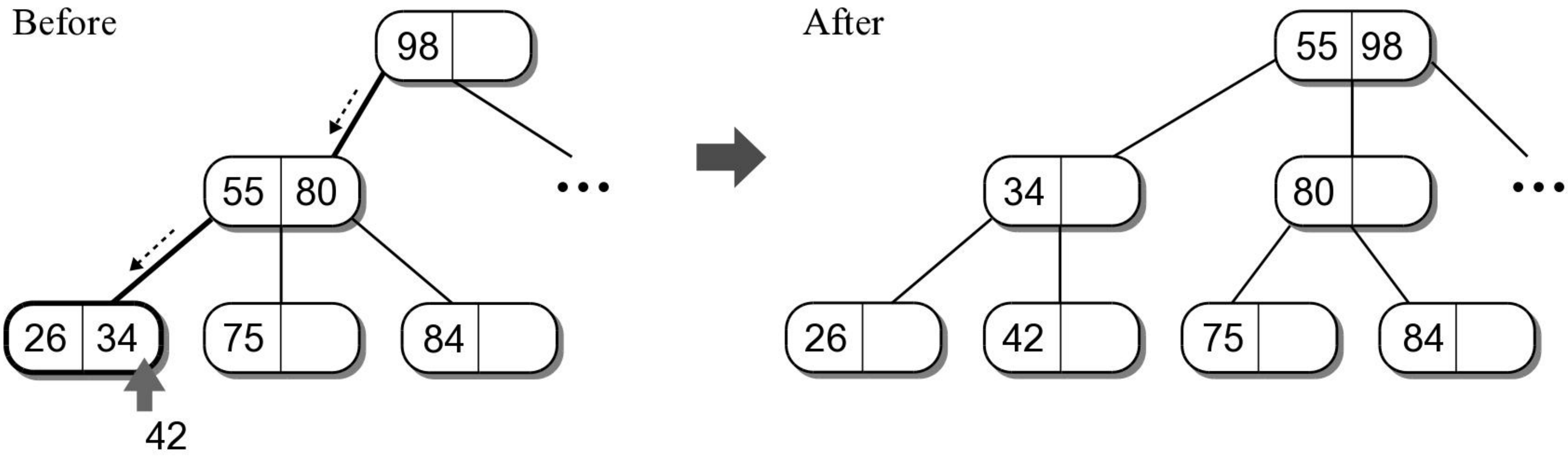
Full Parent Node

- What happens if a node is split and its parent contains three children?



Splitting Parent Node

- The parent node has to be split in a similar fashion as a leaf node.

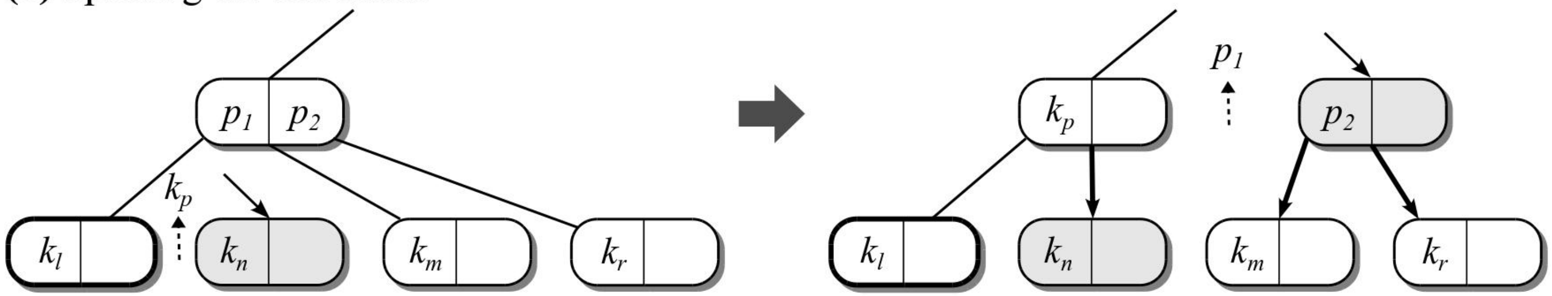


Splitting Parent Node

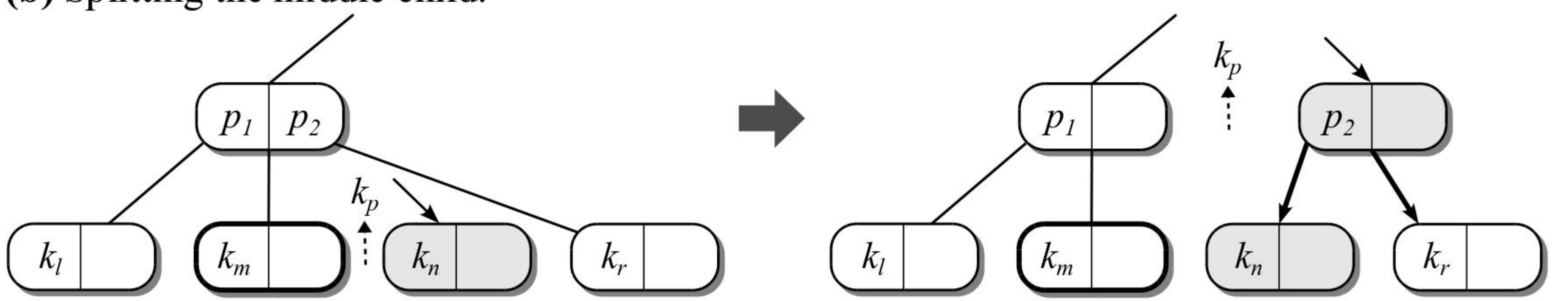
- When the parent node is split, a new parent node is created at the same level.
  - The two keys in the original parent and the promoted key have to be distributed.
  - Connections between the parents and children have to be modified.
  - There are three cases.

Splitting Parent Node

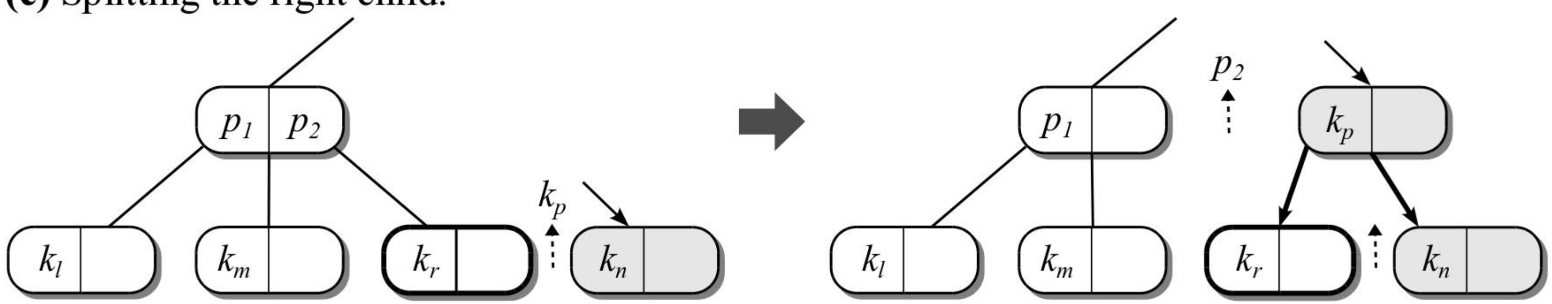
(a) Splitting the left child.



(b) Splitting the middle child.



(c) Splitting the right child.

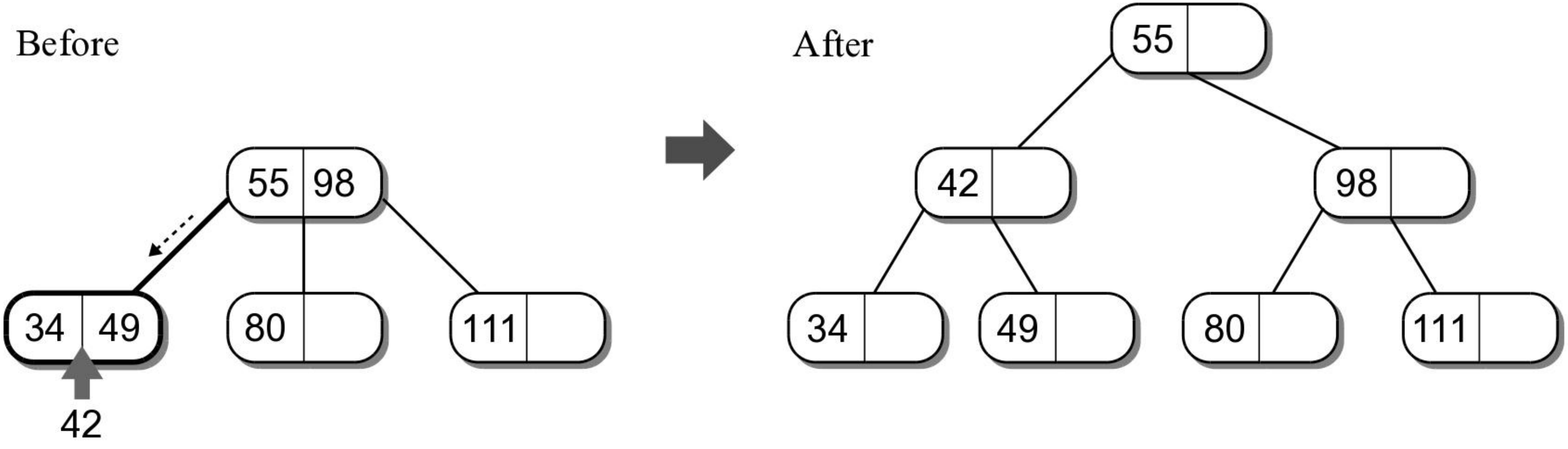


Recursive Operation

- The splitting process can continue up the tree until either a non-full parent node or the root is located.
- Splitting the root node is a special case.
  - The root node is split like any parent or leaf node.
  - A new root node is created into which the promoted key is stored.

Splitting Root Node

- The child links of the new root node must be set:
  - The original root becomes the left child.
  - The new node created from the original split of the root becomes the middle child.



2-3 Tree Efficiency

- The efficiency depends on the height of the tree.
  - minimum height = $\log_3 n$
  - maximum height = $\log_2 n$

Operation	Worst Case
search	O(log n)
insert	O(log n)
delete	O(log n)
traversal	O(n)