Binary Search Tree

Revised based on textbook author's notes.

Search Trees

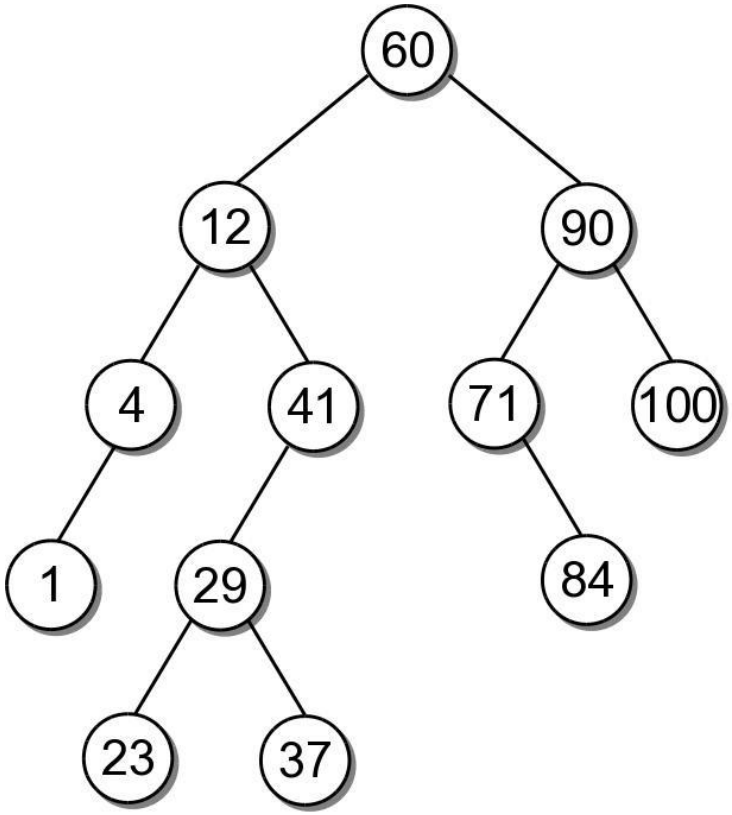
- The tree structure can be used for searching.
  - Each node contains a search key as part of its data or **payload**.
  - Nodes are organized based on the relationship between the keys.
- Search trees can be used to implement various types of containers.
  - Most common use is with the Map ADT.

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



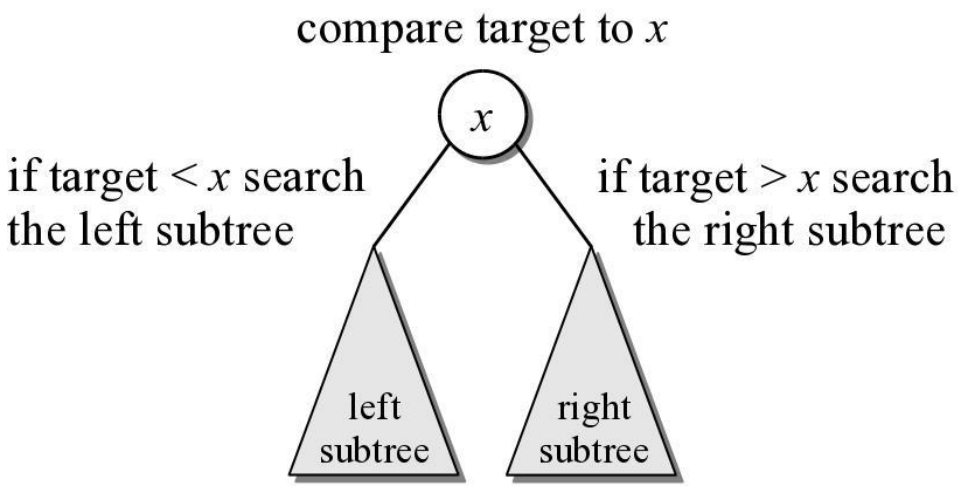
BST – ADT

bst.py

```
# We use an unique name to distinguish this version
# from others in the chapter.
class BST :
  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
```

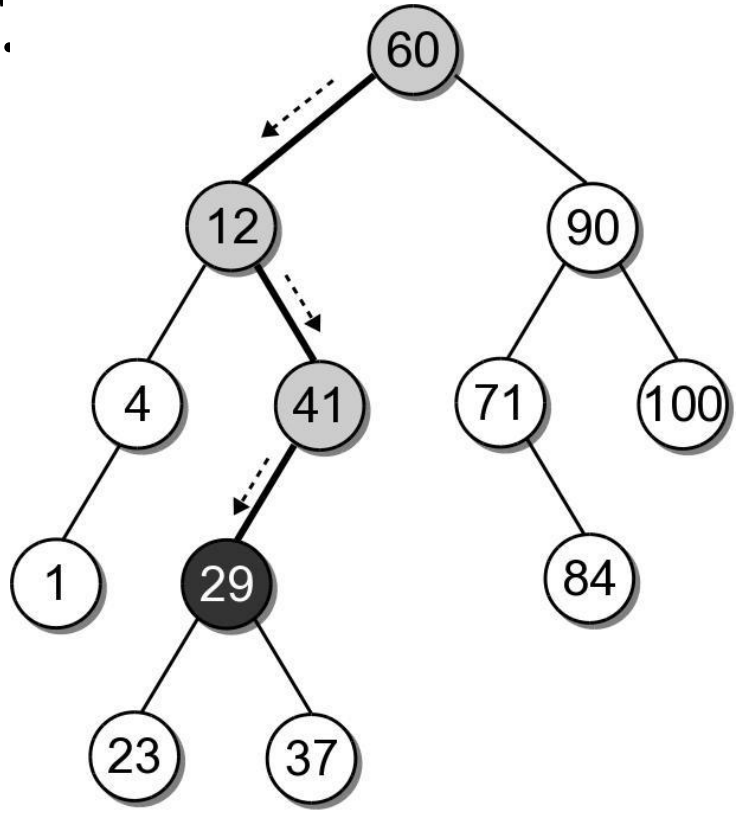
BST – Searching

- 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.



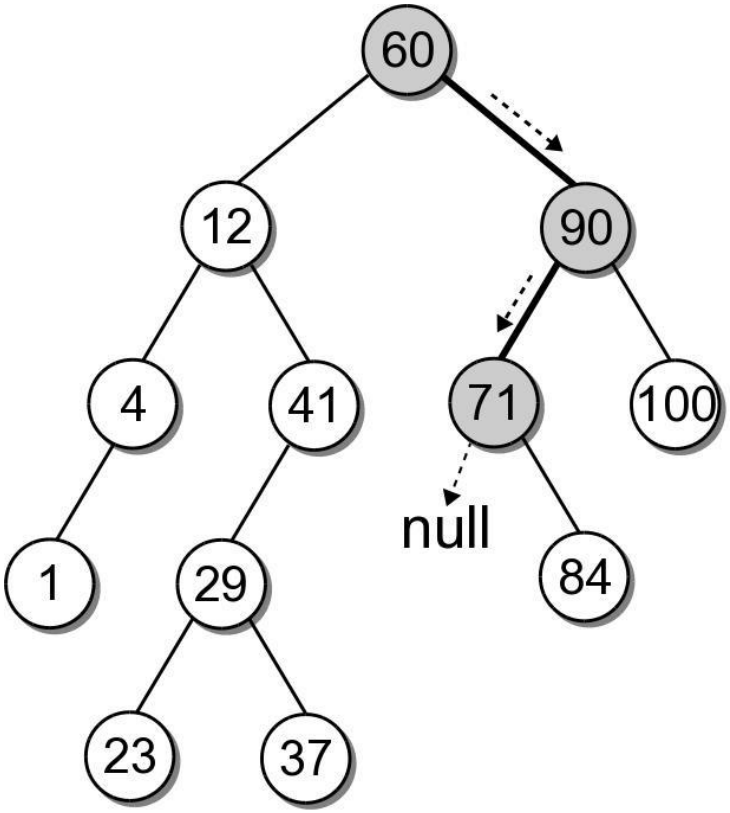
BST – Search Example

- 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.



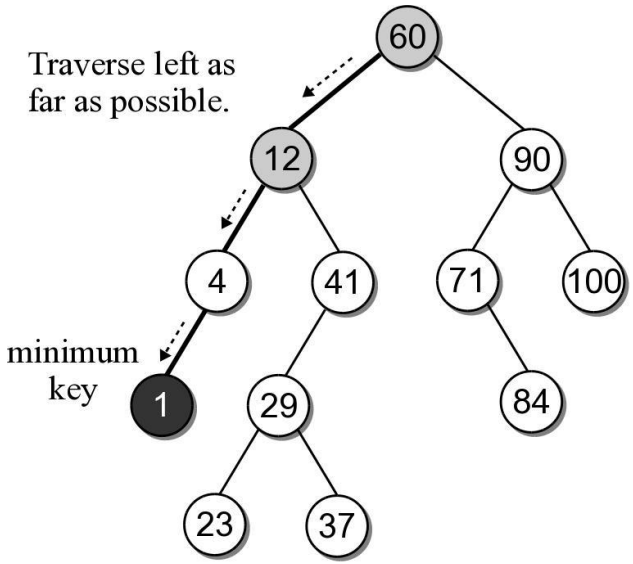
BST – Search Implementation

bst.py

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

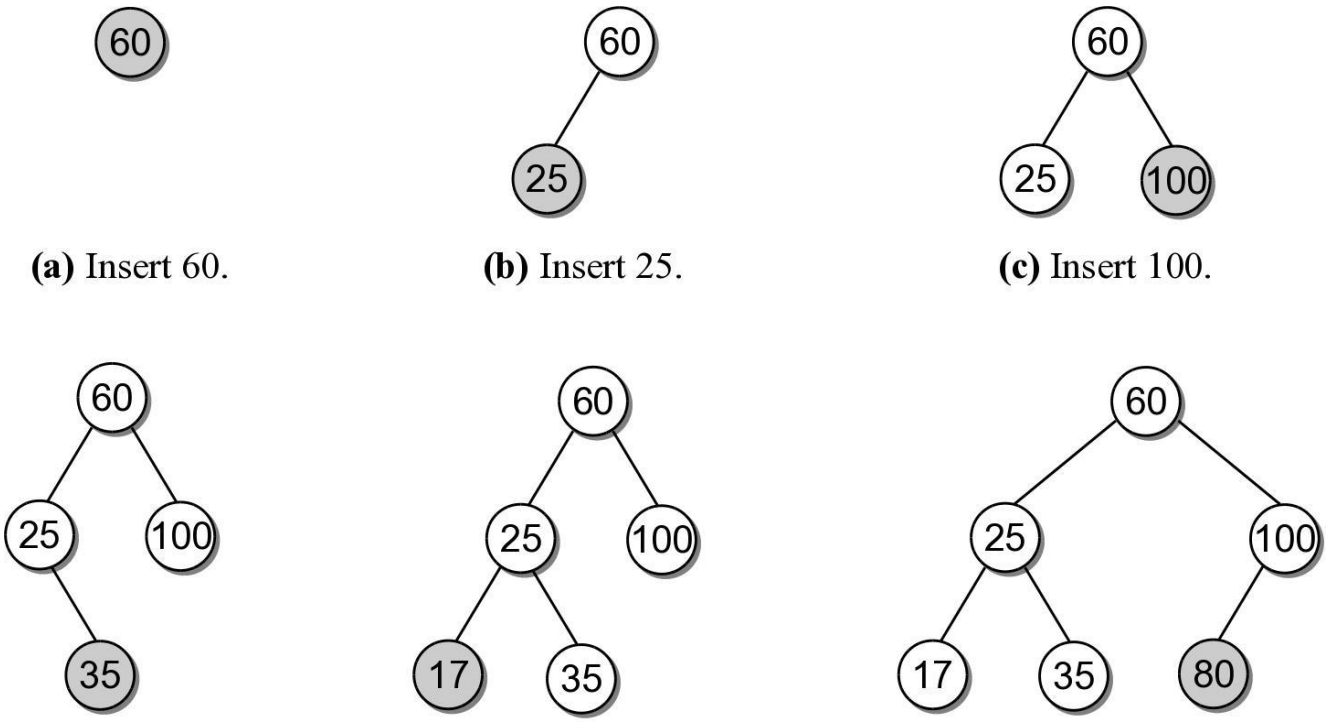
BST – Insertions

- 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
```

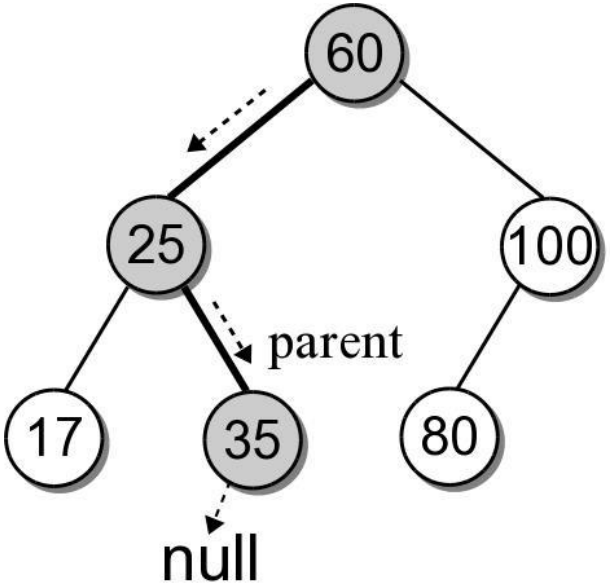


(a) Insert 60. **(b)** Insert 25. **(c)** Insert 100.

(d) Insert 35. **(e)** Insert 17. **(f)** Insert 80.

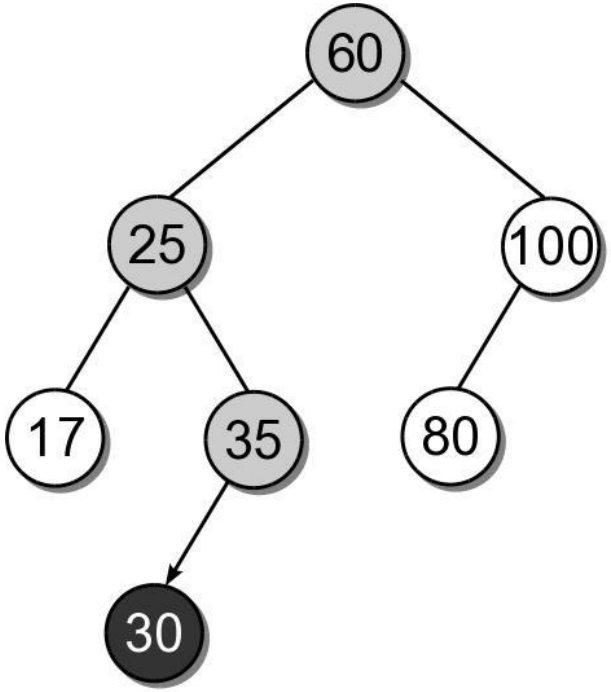
BST – Insertion

- 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.



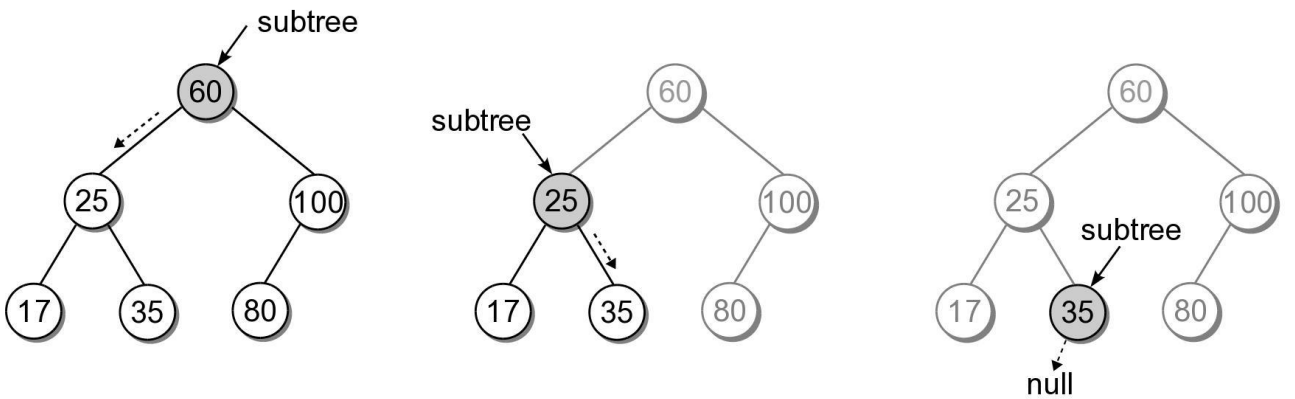
BST – Insert Implementation

bst.py

```
class BST :
# ...
  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
```

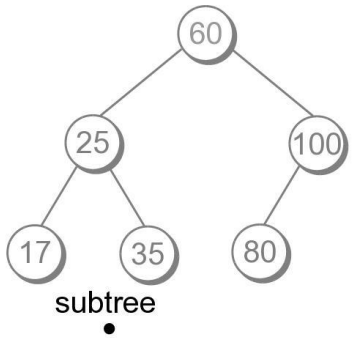
BST – Insert Steps

- Add 30 to our sample BST.

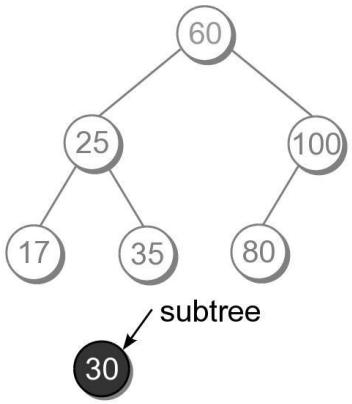


(a) bstInsert(root,30) **(b)** bstInsert(subtree.left,key) **(c)** bstInsert(subtree.right,key)

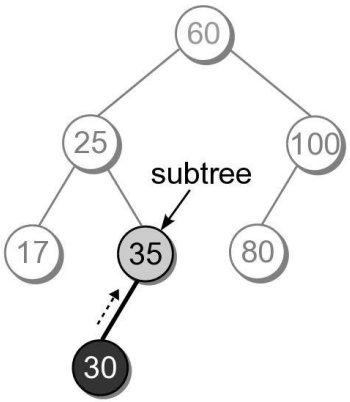
BST – Insert Steps



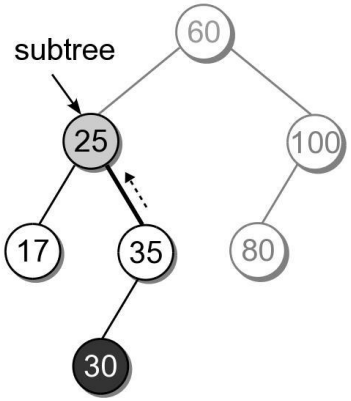
(d) bstInsert(subtree.left,key)



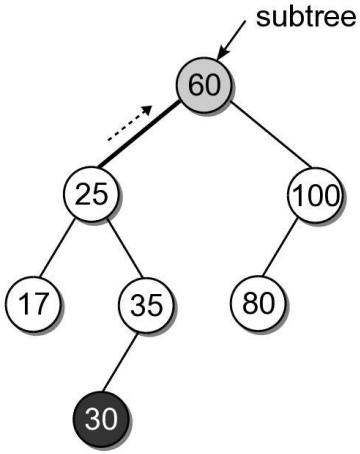
(e) subtree = TreeNode(key)



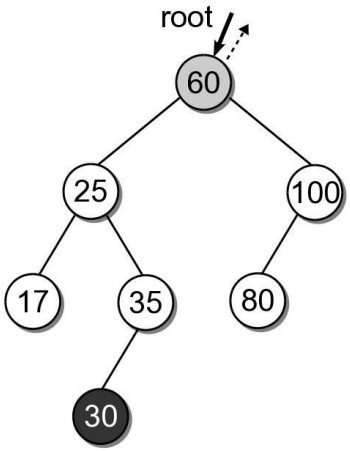
(f) subtree.left = bstInsert(...)



(g) subtree.right = bstInsert(...)



(h) subtree.left = bstInsert(...)



(i) root = bstInsert(...)

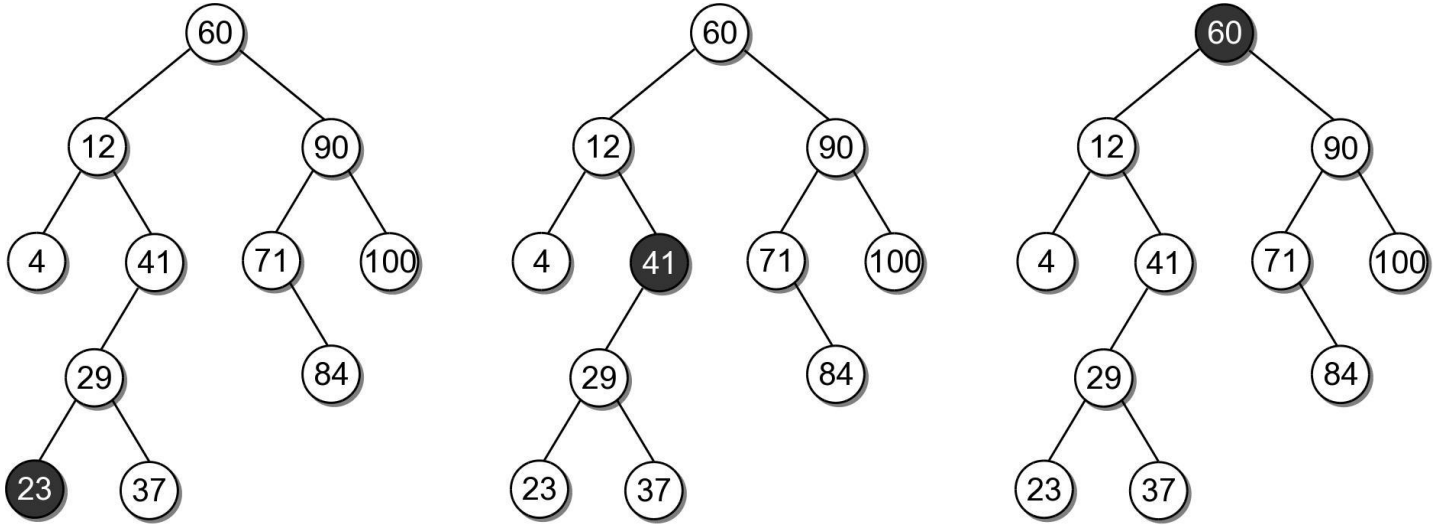
BST – Deletions

- 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.

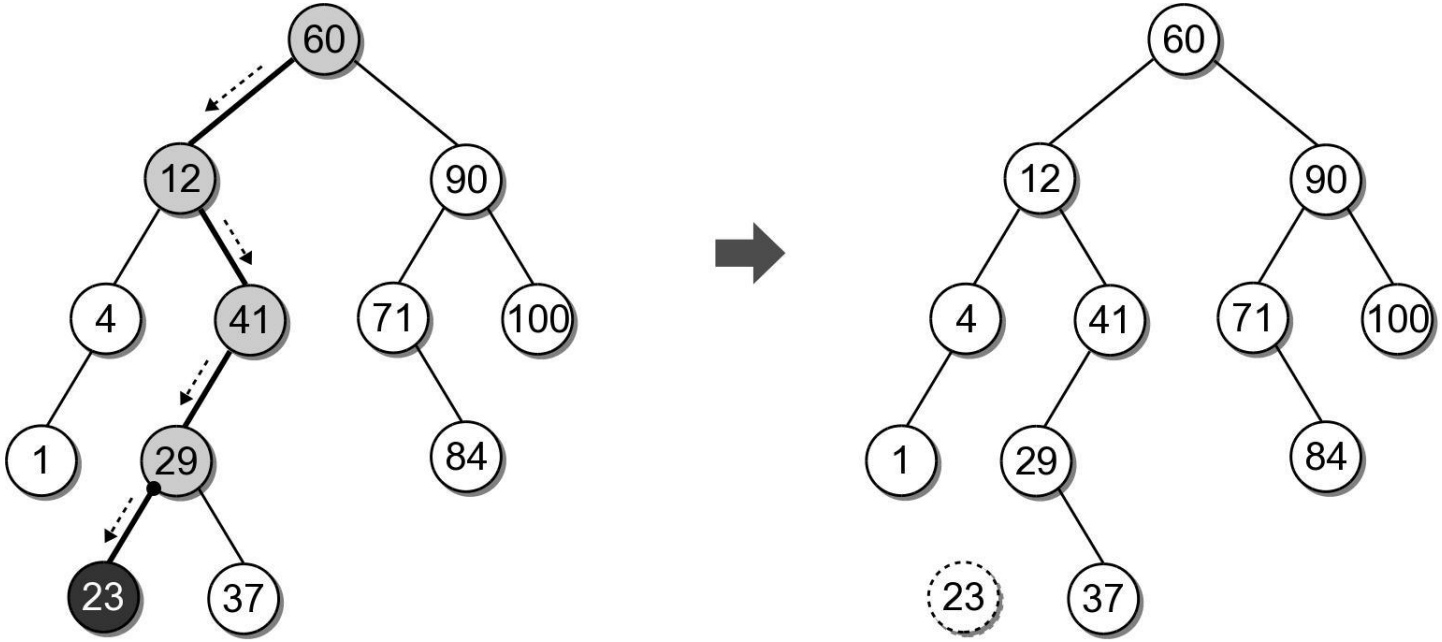
BST – Deletions

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



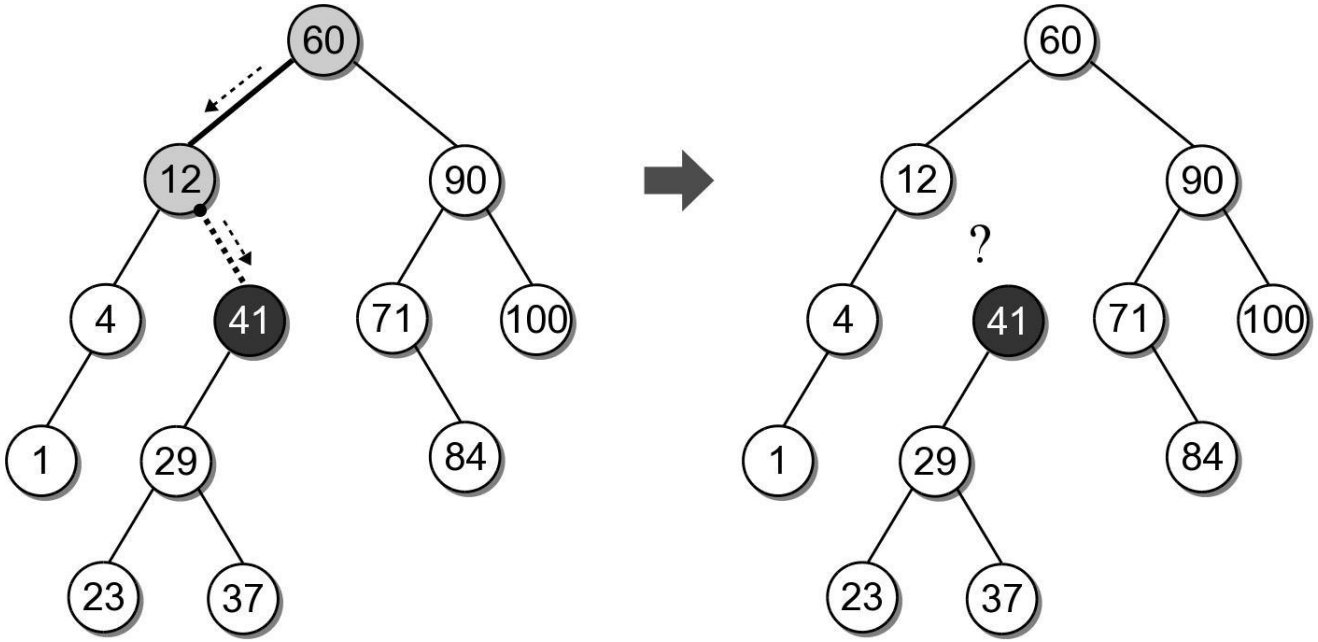
BST – Delete Leaf Node

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



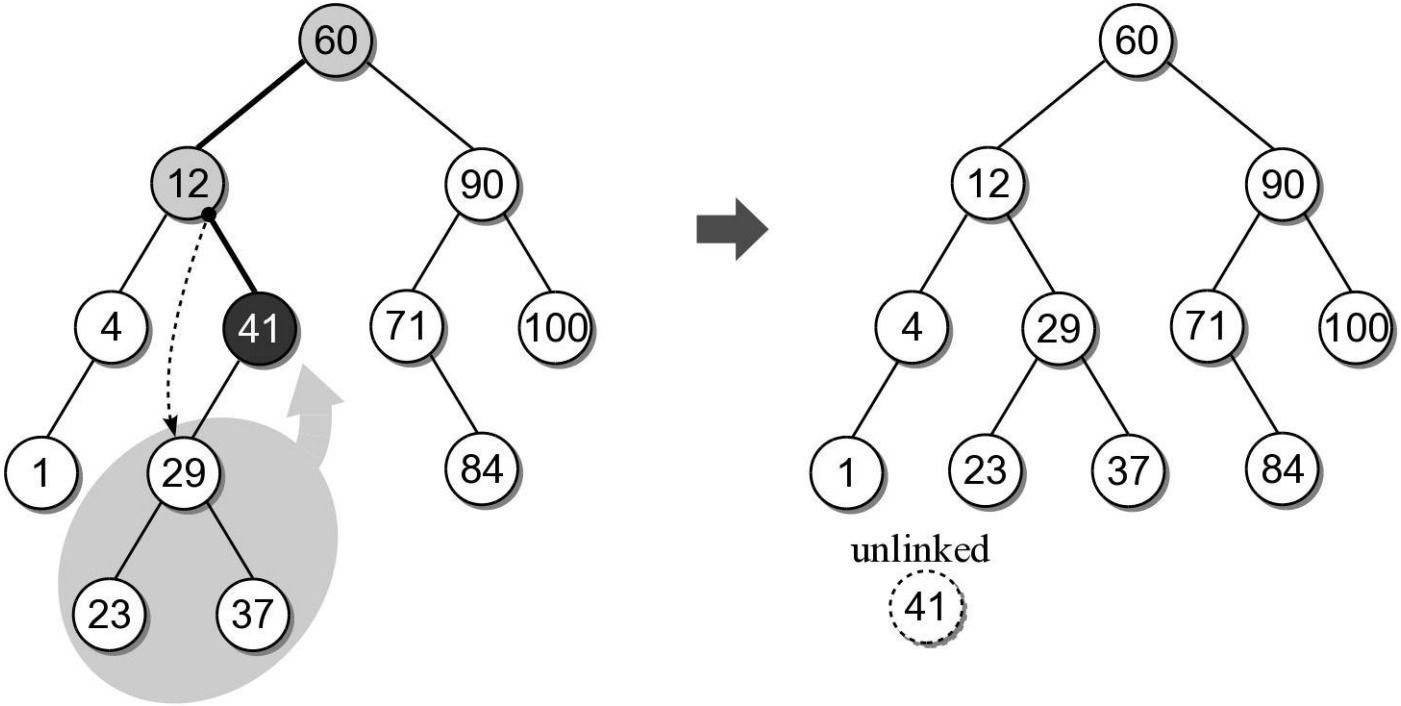
BST – Delete Interior Node

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



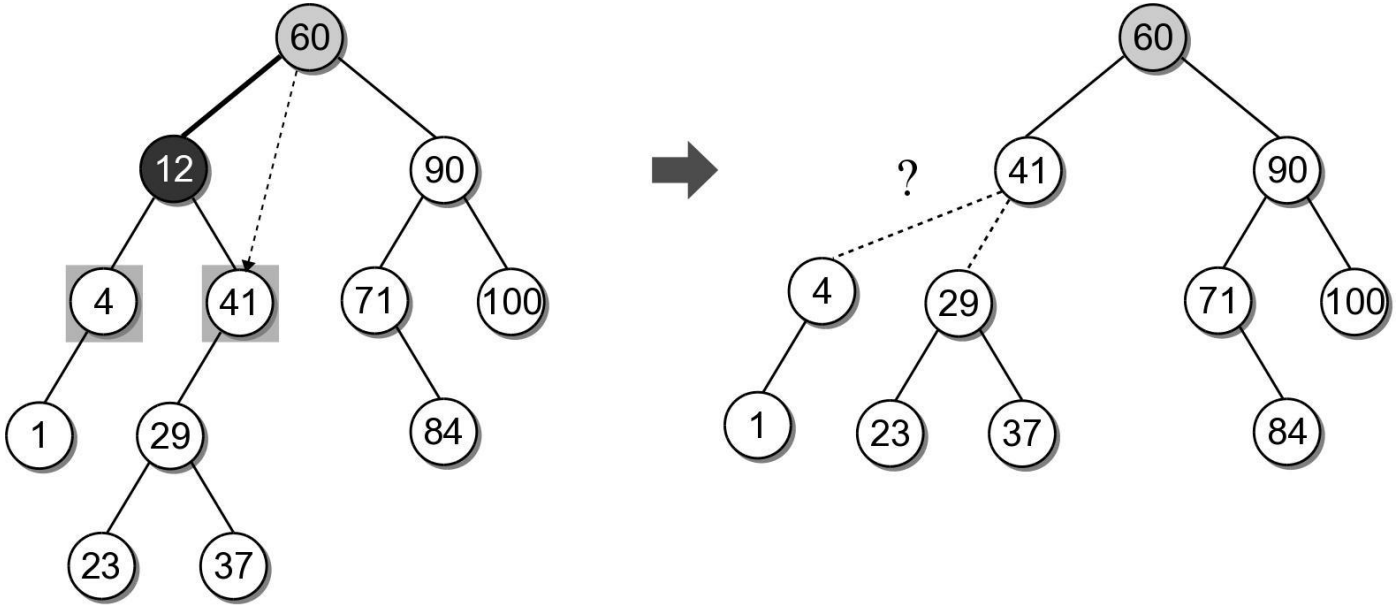
BST – Delete Interior Node

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



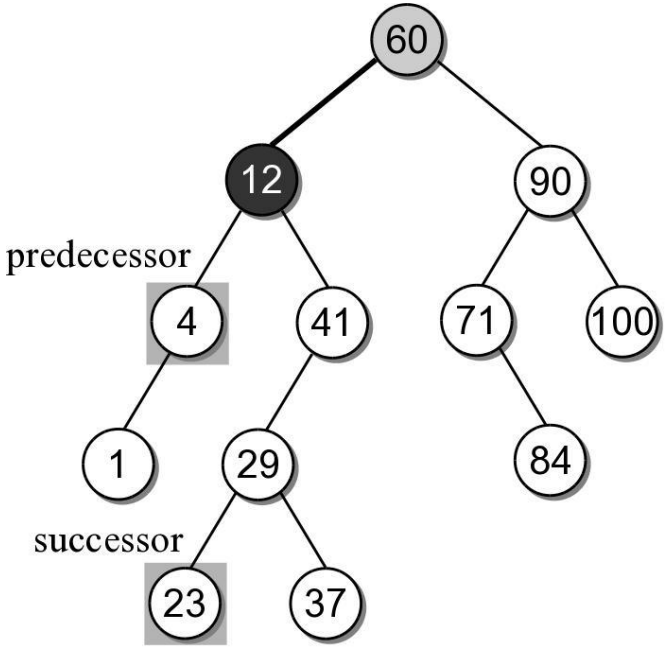
BST – Delete Interior Node

- 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.

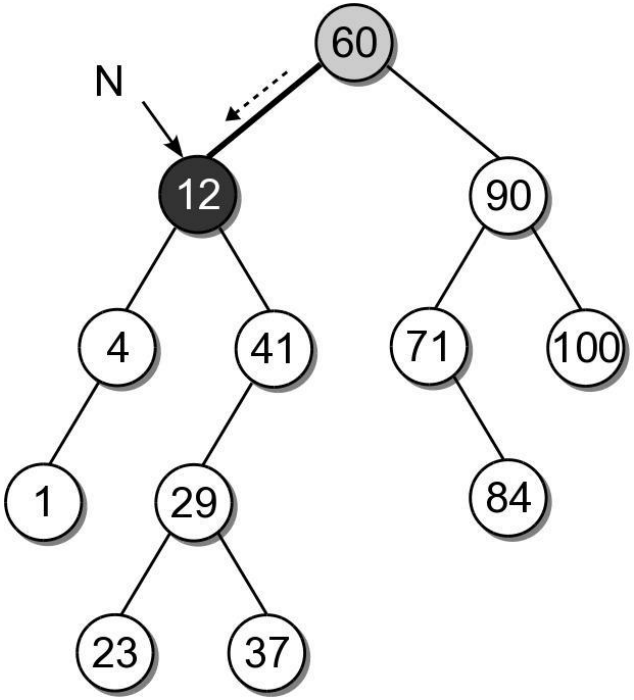


BST – Delete Interior Node

- 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.

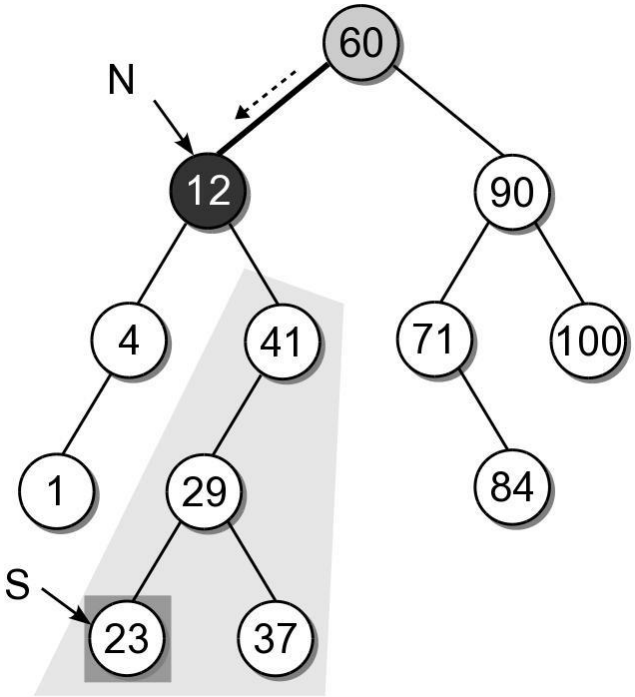
BST – Delete Interior Node

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



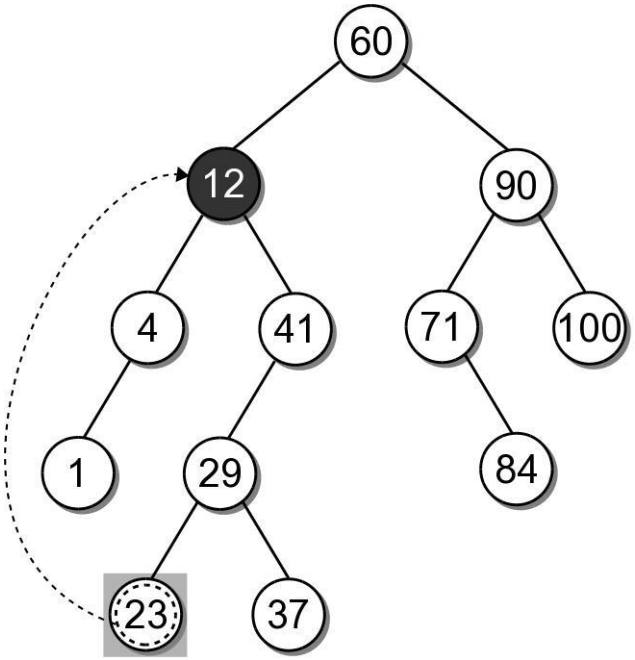
BST – Delete Interior Node

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



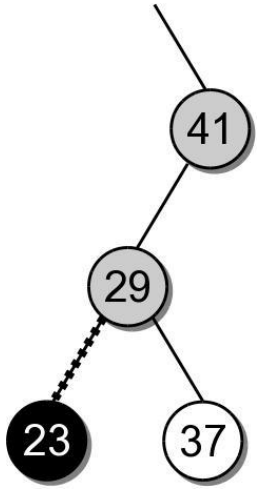
BST – Delete Interior Node

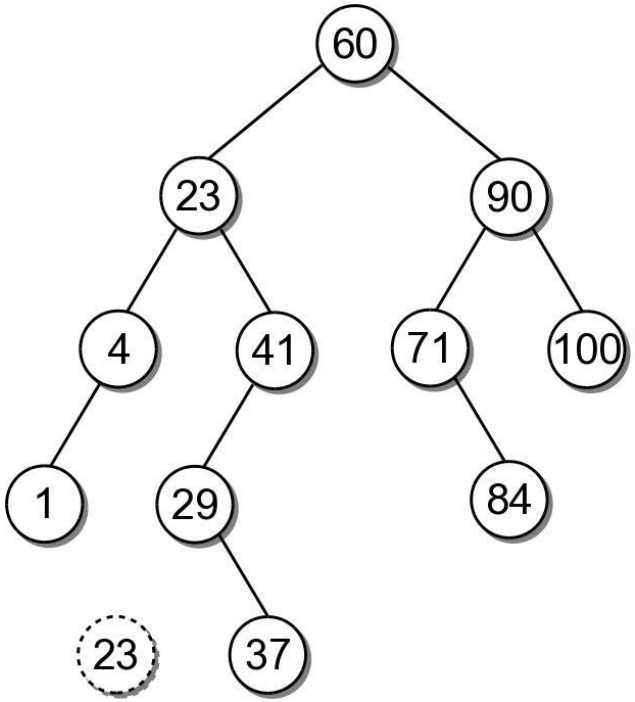
- (3) Copy the payload from node S to node N.



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

bst.py

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

BST – Delete Implementation

bst.py

```
class BST :
# ...
  def _bstRemove( self, subtree, target ):
    if subtree is None :  # not found
      return subtree
    elif target < subtree.key :  # search for the left tree
      subtree.left = self._bstRemove( subtree.left, target )
      return subtree
    elif target > subtree.key :  # search for the right tree
      subtree.right = self._bstRemove( subtree.right, target )
      return subtree
    else :   # target == subtree.key, delete the node
        ......
```

BST – Delete Implementation

bst.py

```
class BST :
# ...
  def _bstRemove( self, subtree, target ):
          ......
    else :
      if subtree.left is None and subtree.right is None :
        return None     # leaf node
      elif subtree.left is None or subtree.right is None :
      # has one child only
        if subtree.left is not None :
          return subtree.left
        else :
          return subtree.right
      else : # has both children
        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)
_bstRemove(root, k)	O(n)
traversal	O(n)

Exercises

1. The `remove()` method uses a `bst_find_min(subtree)` method to search and return the node with minimum key in a tree rooted at `subtree`. Implement this method.
2. Write a method `bst_sort_reverse()` to sort a binary search tree rooted at `subtree` in reverse order.
3. Test your methods by running the program `testbst.py` on the course website.