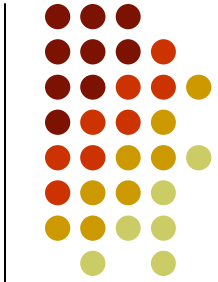
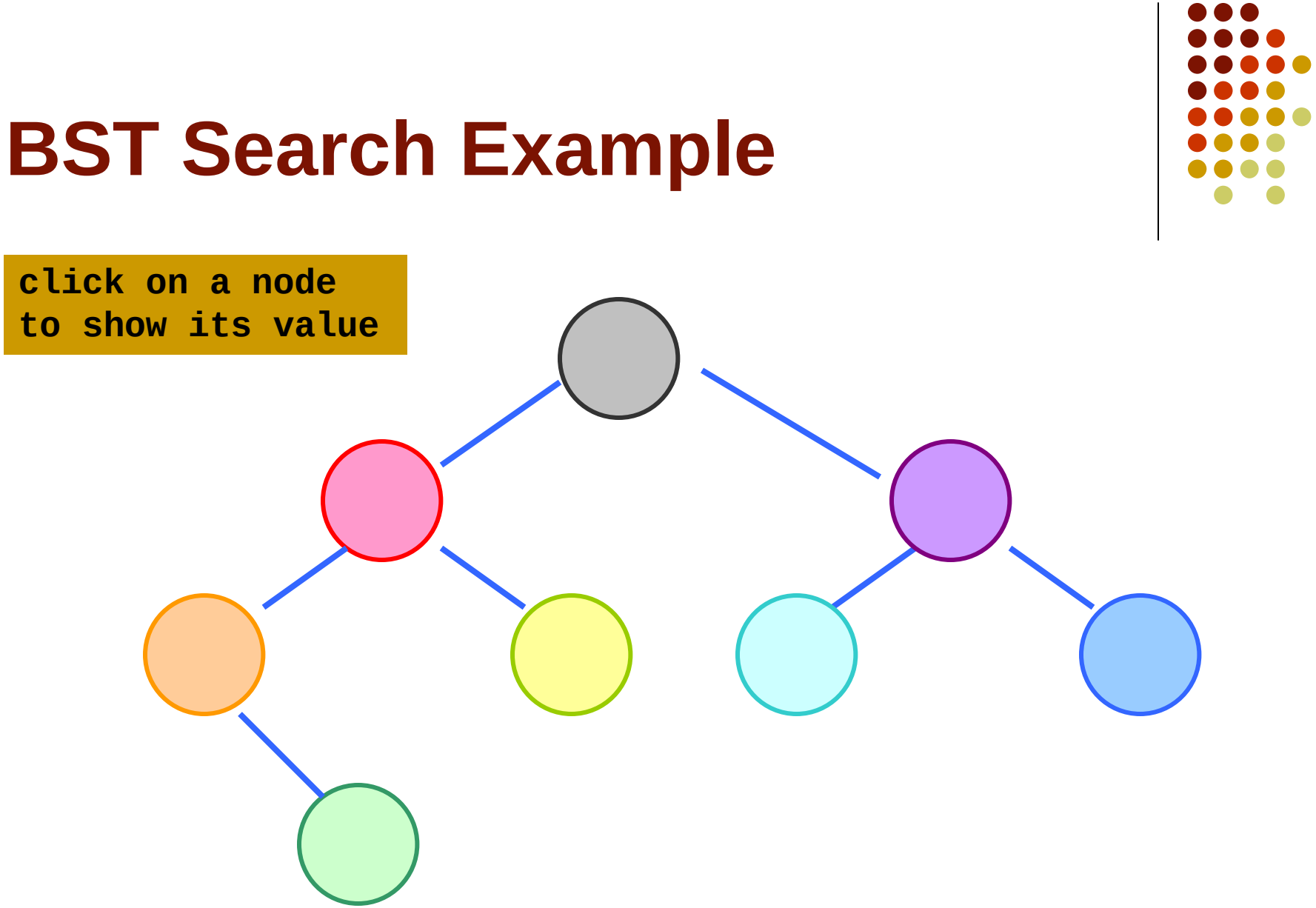
BST Search

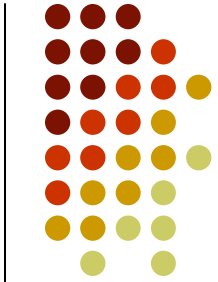


- To find a value in a BST search from the root node:
  - If the target is less than the value in the node search its left subtree
  - If the target is greater than the value in the node search its right subtree
  - Otherwise return data, etc.
  - If null value is reached, return null (“not found”).
- How many comparisons?
  - One for each node on the path
  - Worst case: height of the tree

BST Search Example

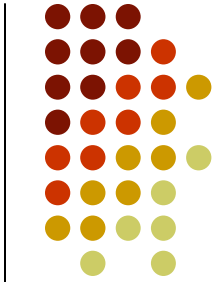


Search algorithm (recursive)



```
T retrieveItem(TreeNode<T extends KeyedItem> n, long searchKey)
  // returns a node containing the item with the key searchKey
  // or null if not found
  {
    if (n == null) {
      return null;
    }
    else {
      if (searchKey == n.getItem().getKey()) {
        // item is in the root of some subtree
        return n.getItem();
      }
      else if (searchKey < n.getItem().getKey()) {
        // search the left subtree
        return retrieveItem(n.getLeft(), searchKey);
      }
      else { // search the right subtree
        return retrieveItem(n.getRight(), searchKey);
      }  // end if
    }  // end if
  }  // end retrieveItem
```

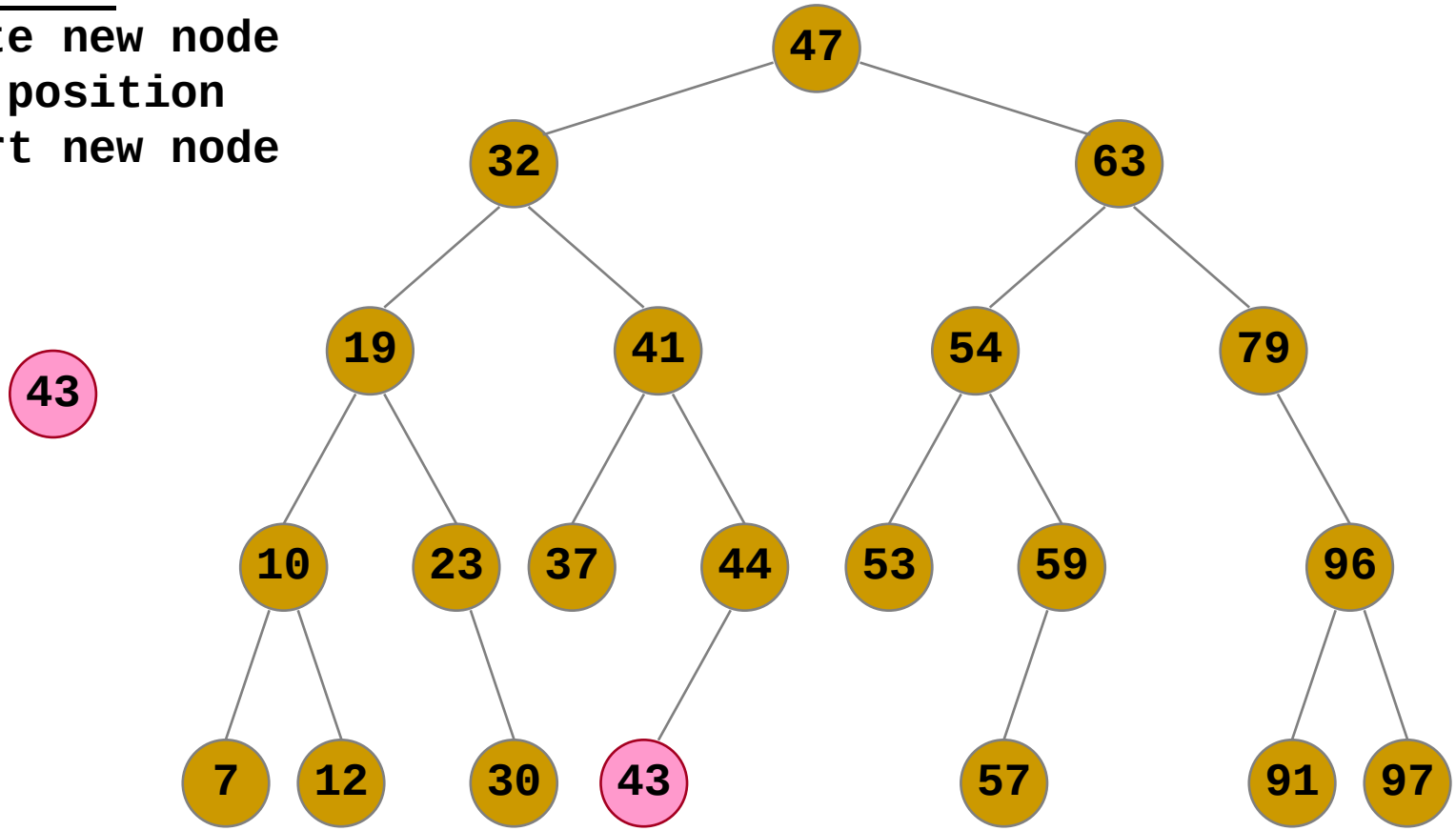
BST Insertion



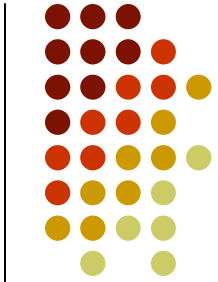
- The BST property must hold **after** insertion
- Therefore the new node must be inserted in the correct position
  - This position is found by performing a search
  - If the search ends at the (null) left child of a node make its left child refer to the new node
  - If the search ends at the (null) right child of a node make its right child refer to the new node
- The cost is about the same as the cost for the search algorithm, O(*height*)

BST Insertion Example

insert 43
create new node
find position
insert new node

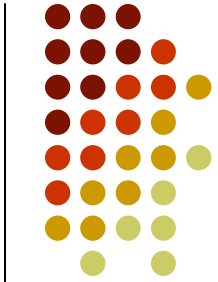


Insertion algorithm (recursive)



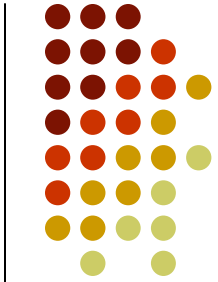
```
TreeNode<T> insertItem(TreeNode<T> n, T newItem)
  // returns a reference to the new root of the subtree rooted in n
  {
    TreeNode<T> newSubtree;
    if (n == null) {
      // position of insertion found; insert after leaf
      // create a new node
      n = new TreeNode<T>(newItem, null, null);
      return n;
    }  // end if

    // search for the insertion position
    if (newItem.getKey() < n.getItem().getKey()) {
      // search the left subtree
      newSubtree = insertItem(n.getLeft(), newItem);
      n.setLeft(newSubtree);
      return n;
    }
    else { // search the right subtree
      newSubtree = insertItem(n.getRight(), newItem);
      n.setRight(newSubtree);
      return n;
    }  // end if
  }  // end insertItem
```

BST Deletion



- After deleting a node the BST property must still hold
- Deletion is not as straightforward as search or insertion
  - So much so that sometimes it is not even implemented!
- There are a number of different cases that have to be considered

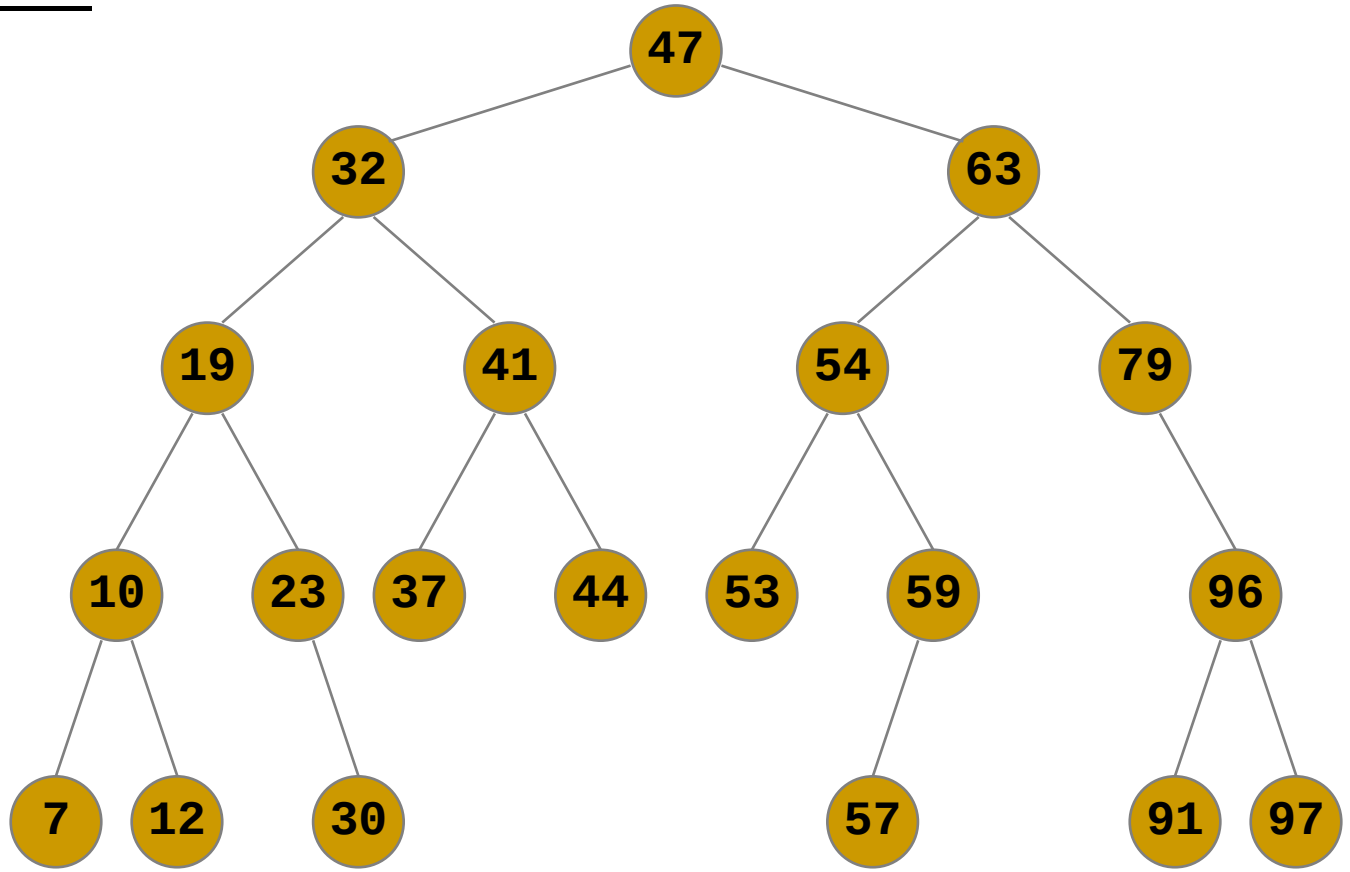
BST Deletion Cases



- The node to be deleted has no children
  - Remove it (assign null to its parent's reference)
- The node to be deleted has one child
  - Replace the node with its subtree
- The node to be deleted has two children
  - Replace the node with its predecessor = the right most node of its left subtree (or with its successor, the left most node of its right subtree)
  - If that node has a child (and it can have at most one child) attach that to the node's parent

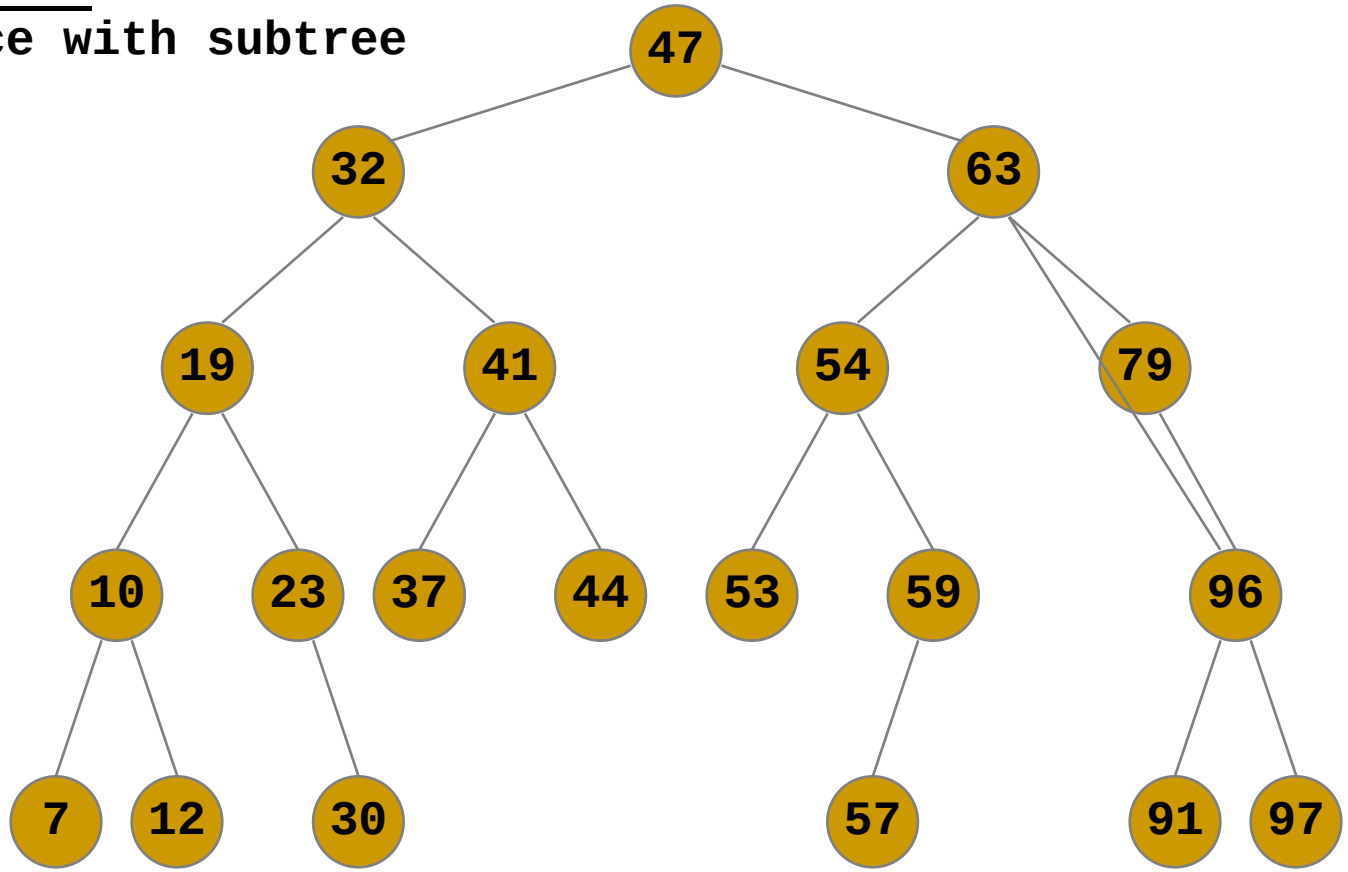
BST Deletion – target is a leaf

delete 30



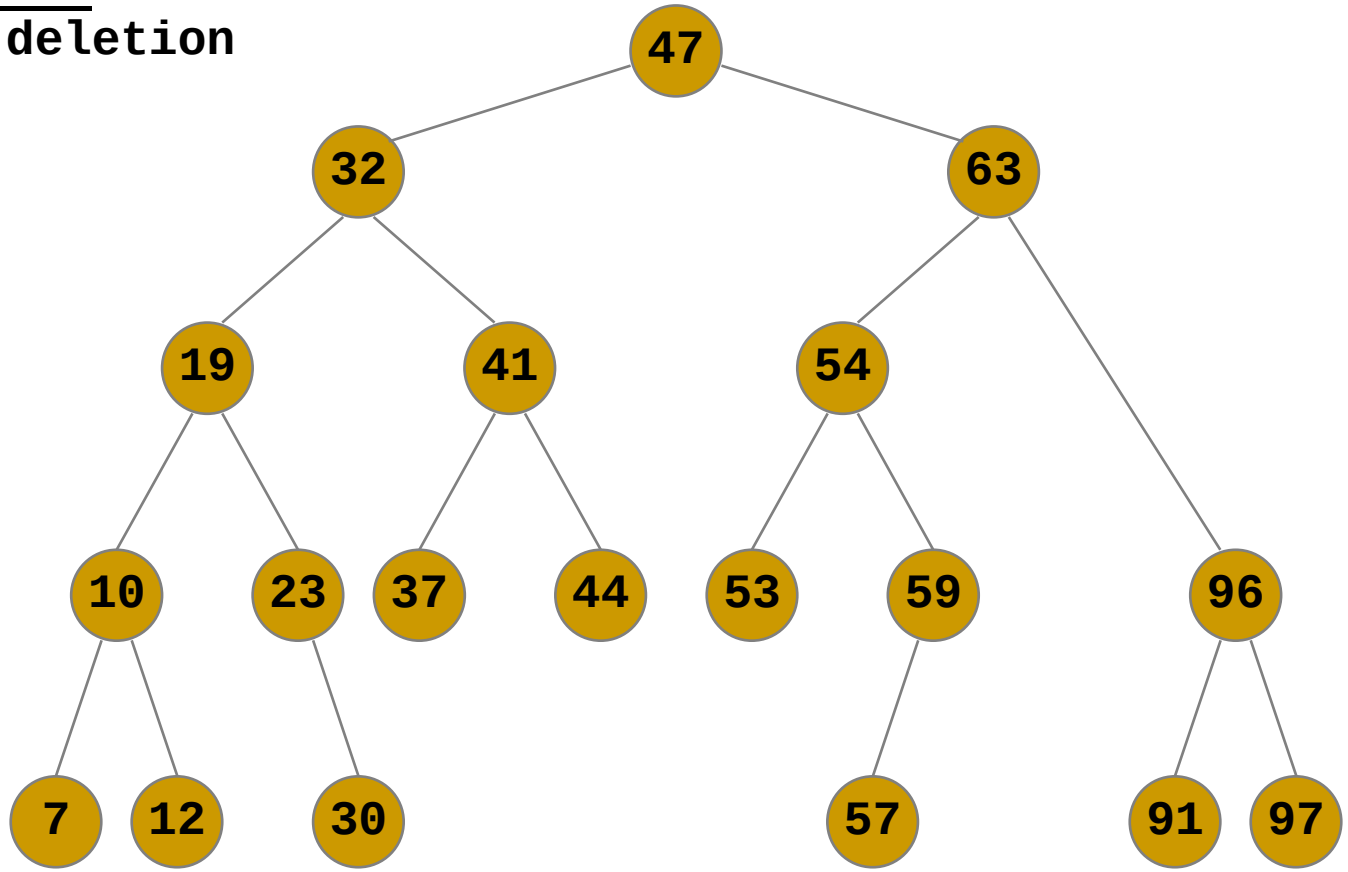
BST Deletion – target has one child

delete 79
replace with subtree



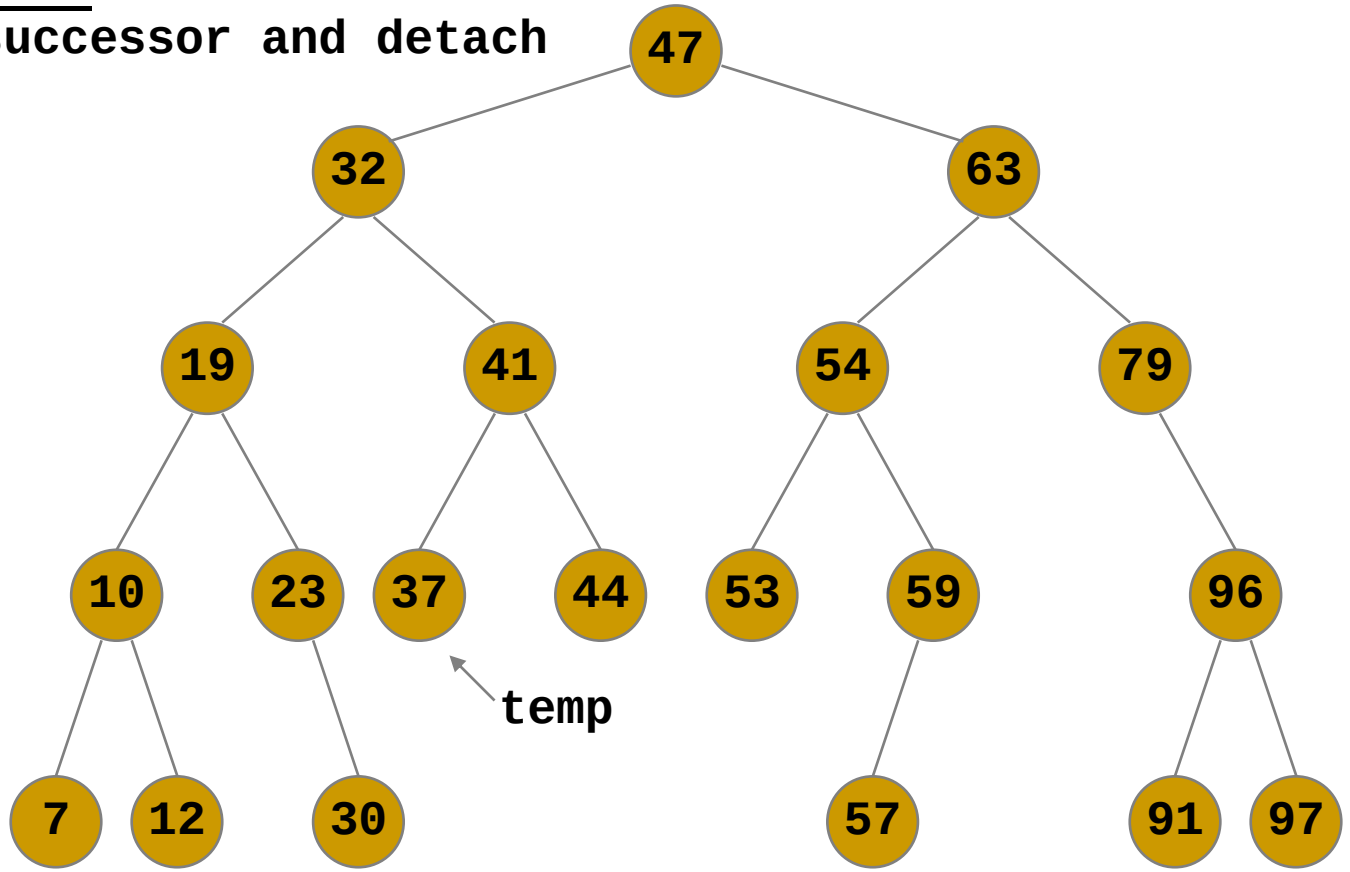
BST Deletion – target has one child

delete 79
after deletion



BST Deletion – target has 2 children

delete 32
find successor and detach

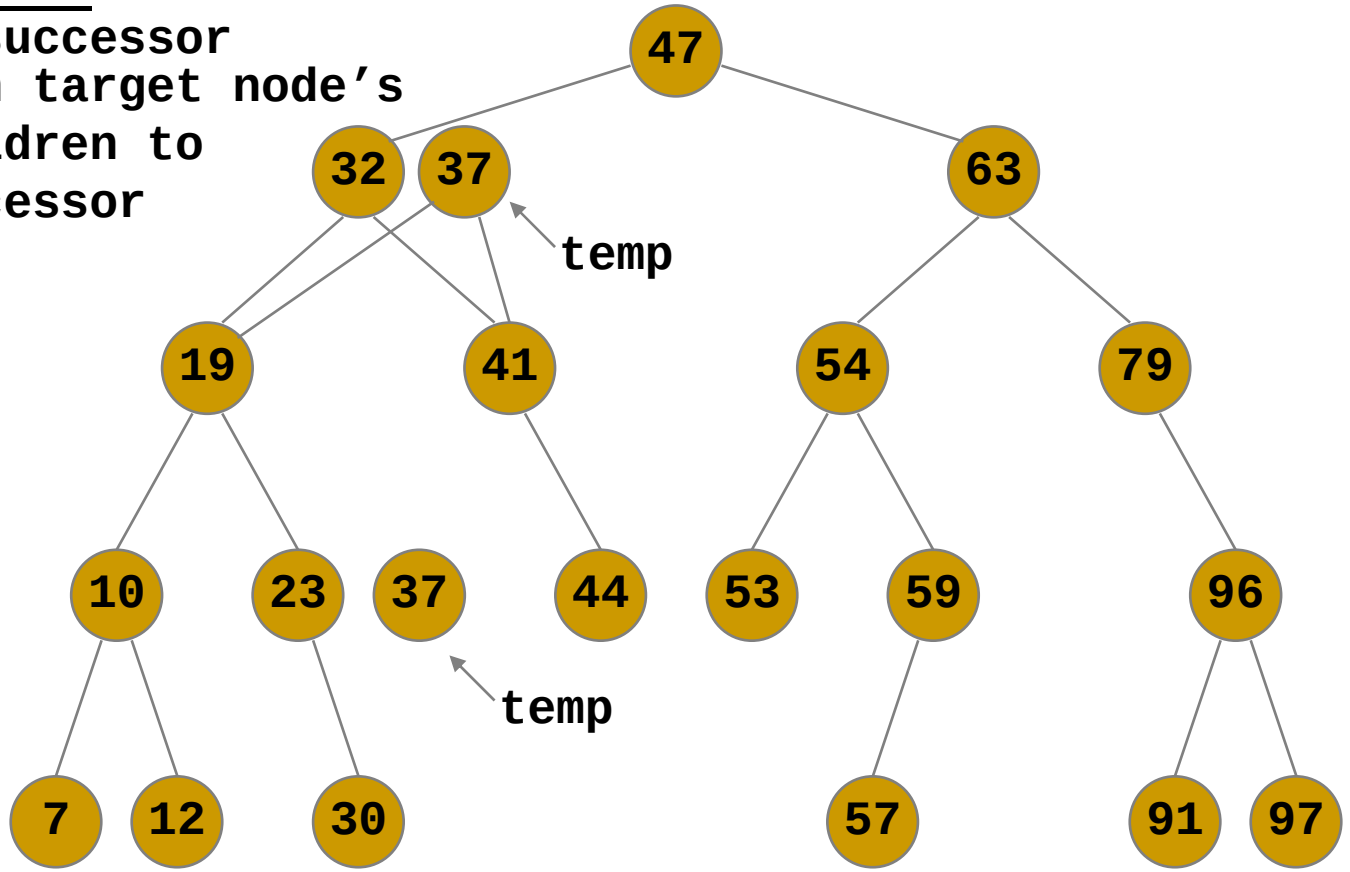


BST Deletion – target has 2 children

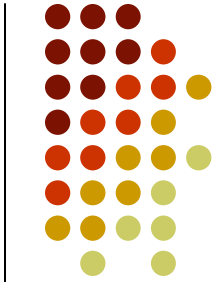
delete 32

find successor

attach target node's children to successor



BST Deletion – target has 2 children

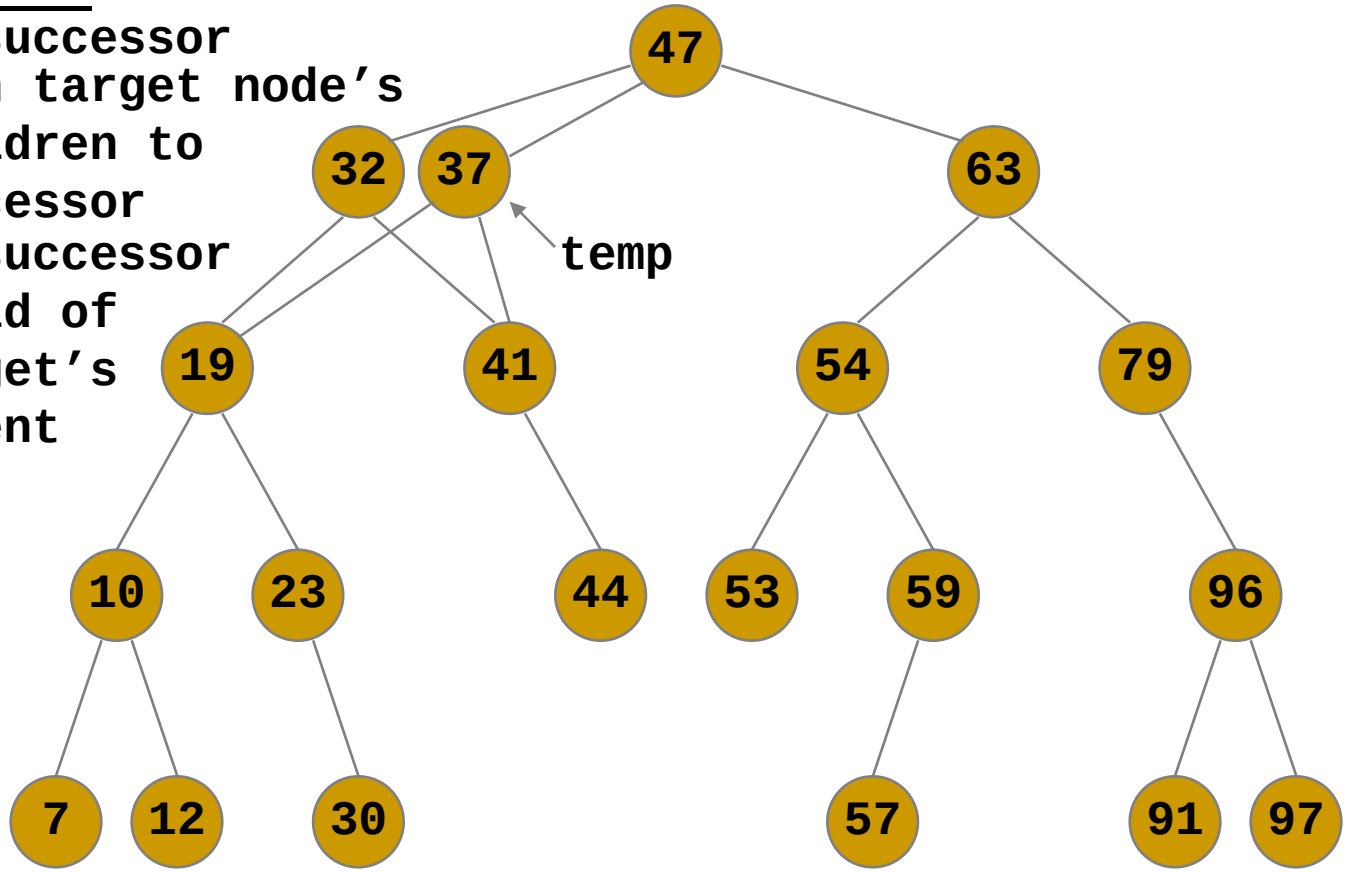


delete 32

find successor

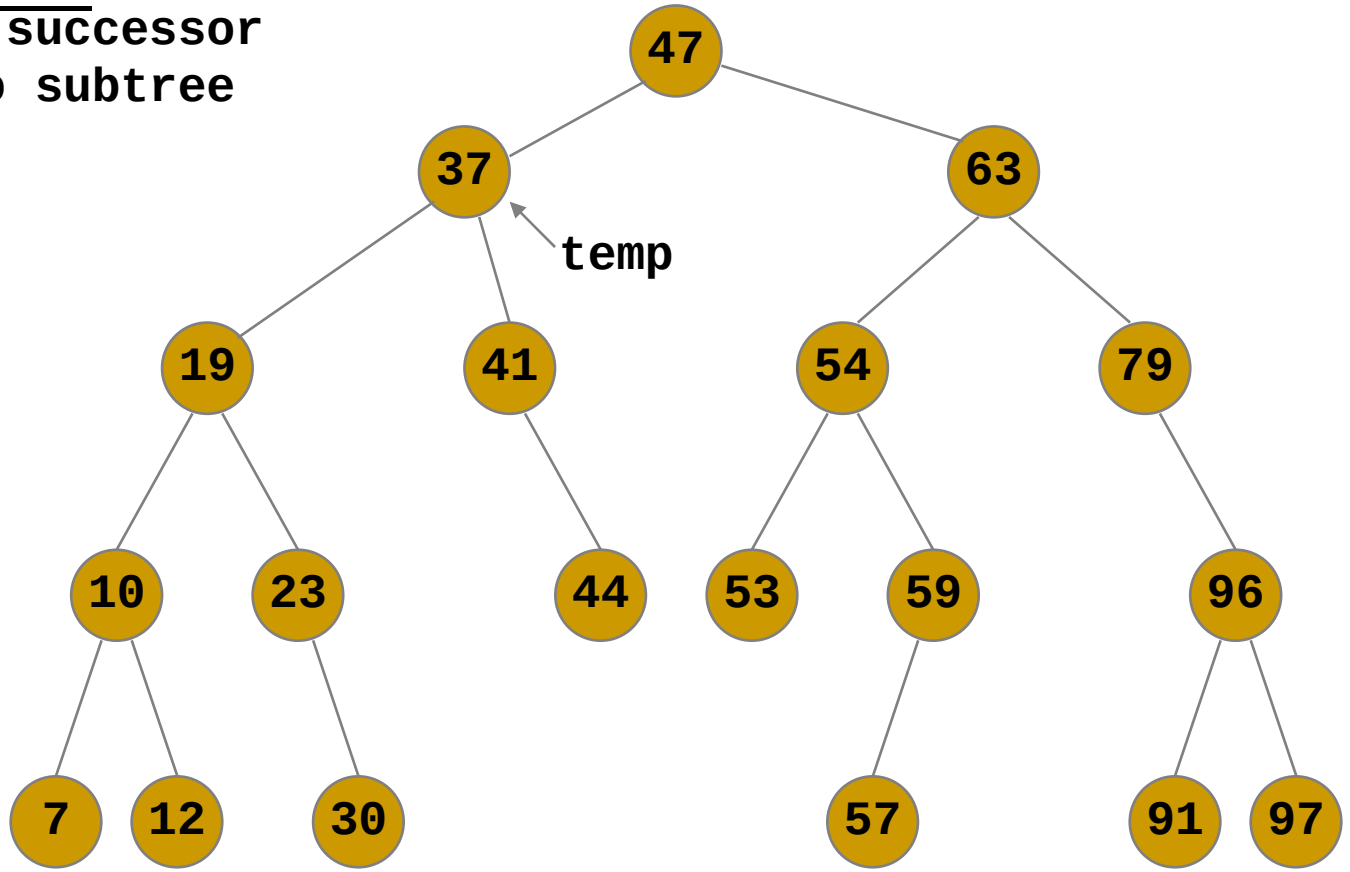
attach target node's children to successor

make successor child of target's parent



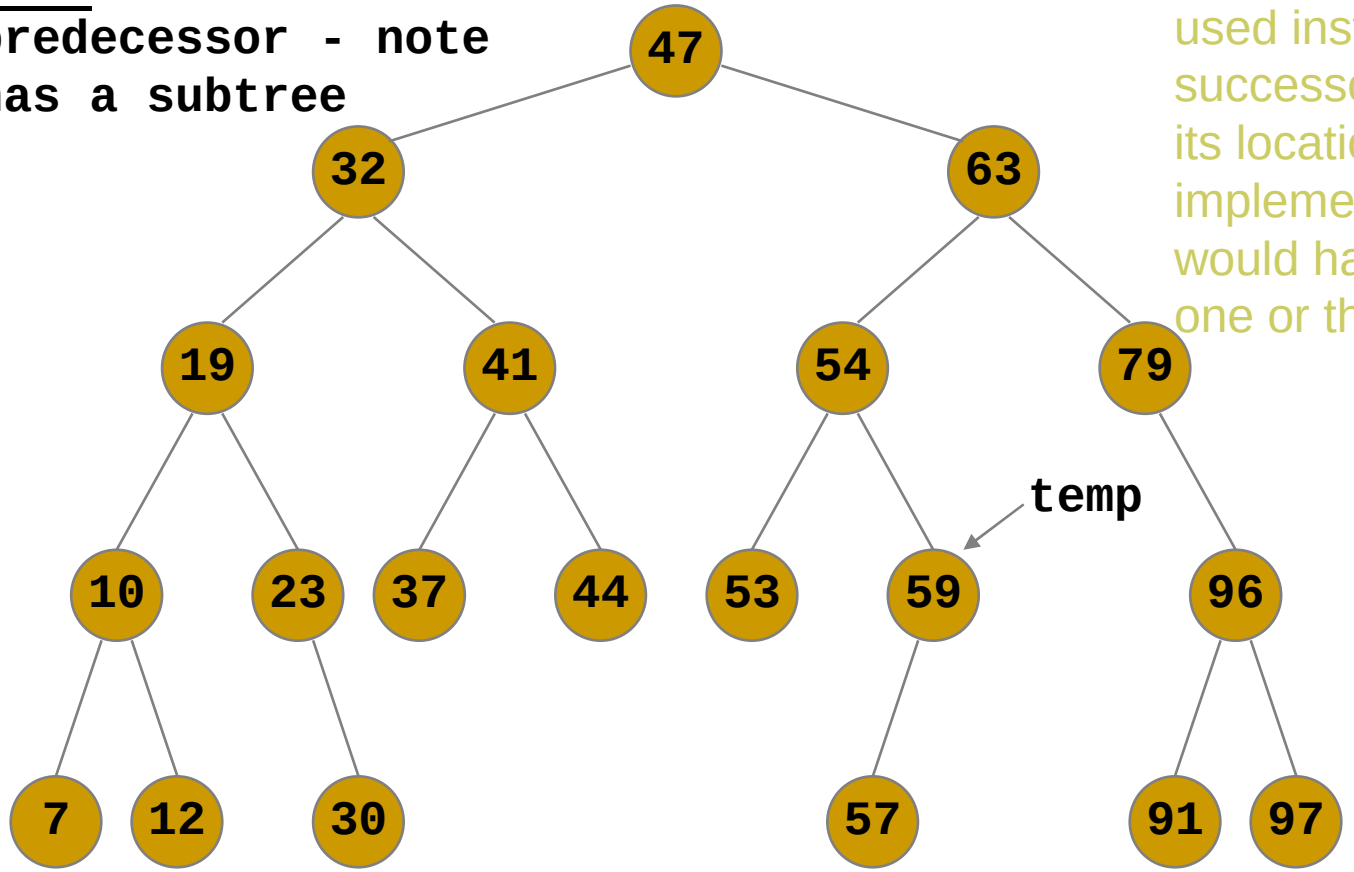
BST Deletion – target has 2 children

delete 32
note: successor
had no subtree



BST Deletion – target has 2 children

delete 63
find predecessor - note it has a subtree



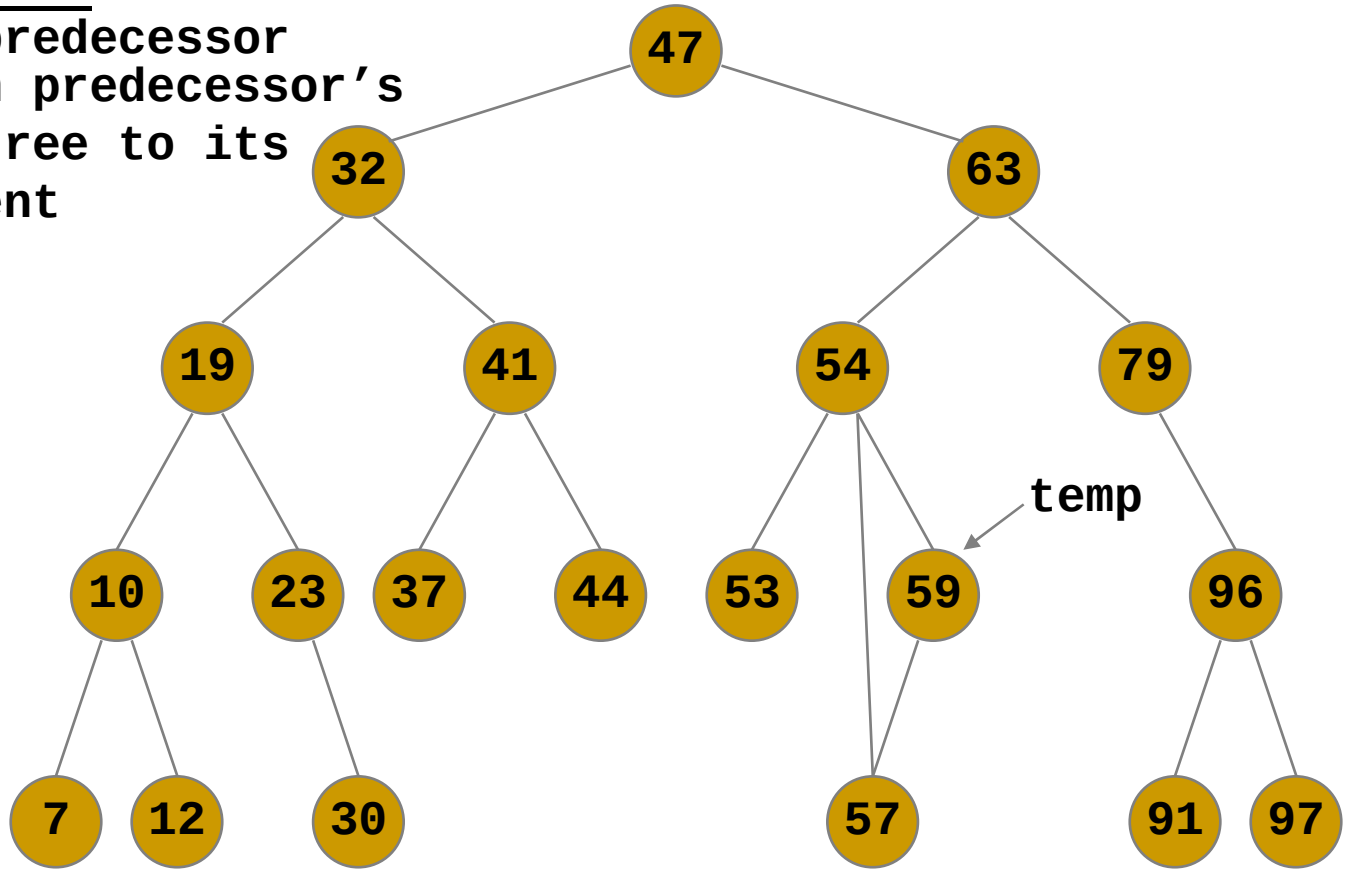
Note: predecessor used instead of successor to show its location - an implementation would have to pick one or the other

BST Deletion – target has 2 children

delete 63

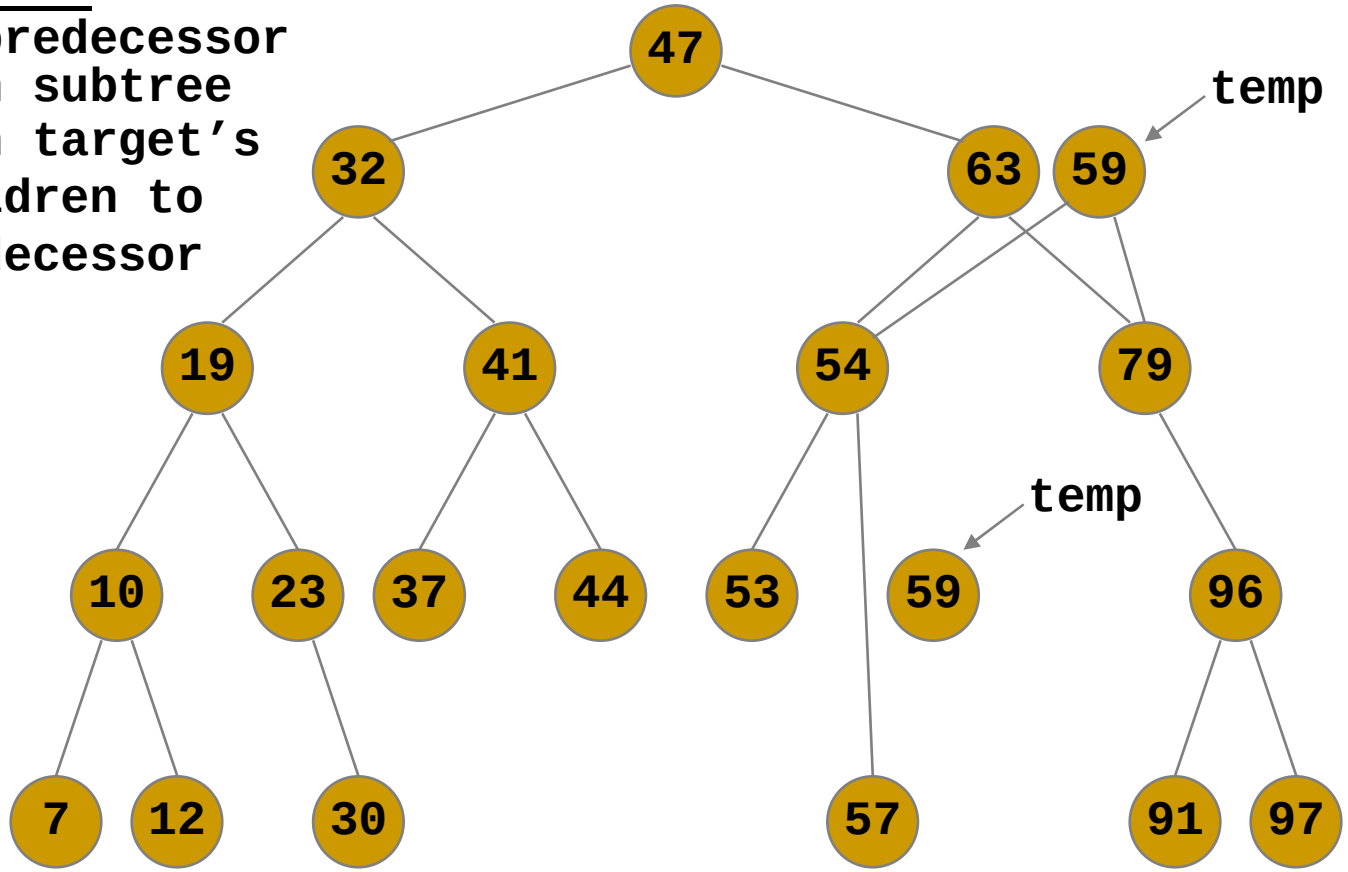
find predecessor

attach predecessor's subtree to its parent



BST Deletion – target has 2 children

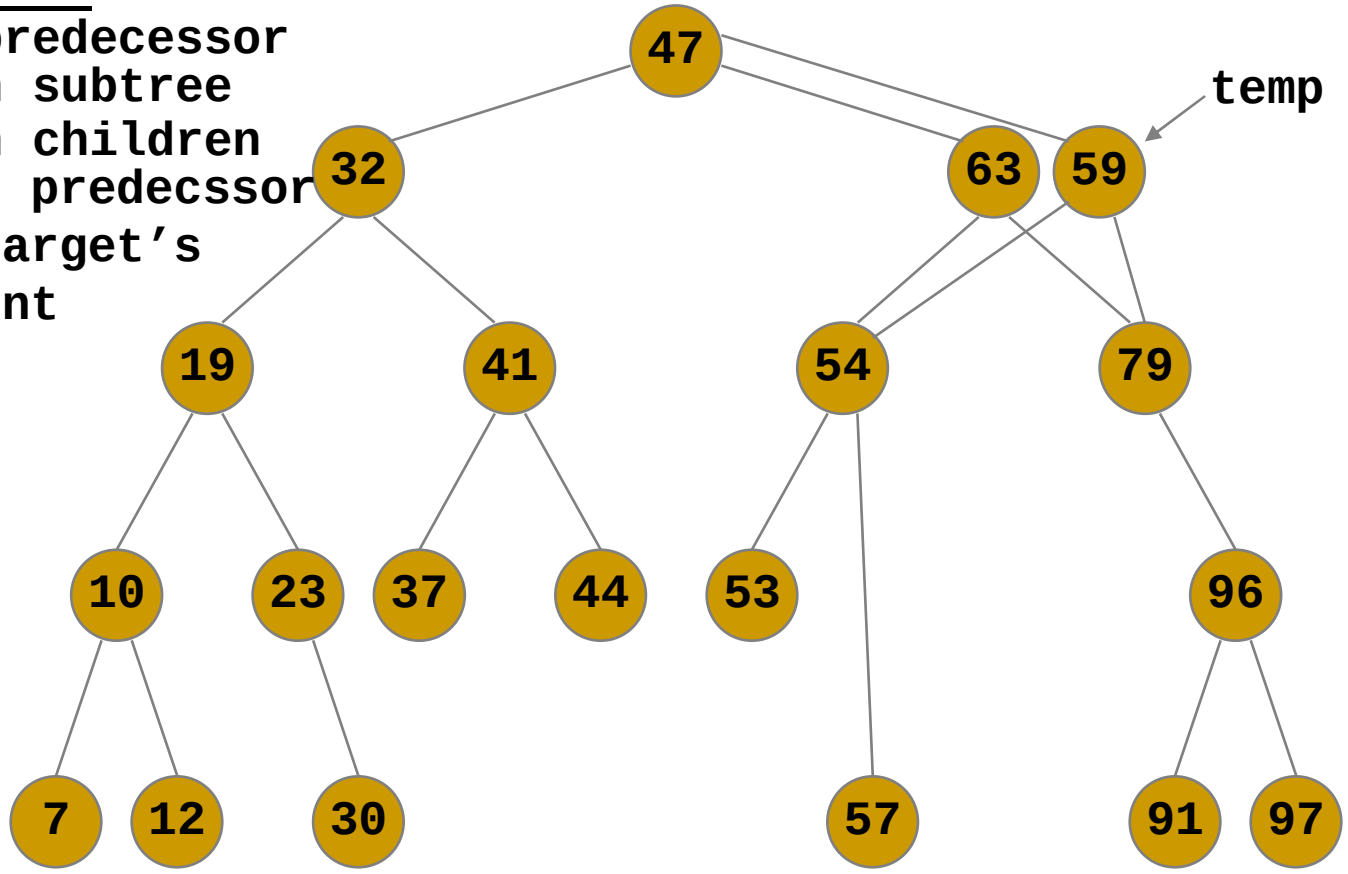
delete 63
find predecessor
attach subtree
attach target's
  children to
  predecessor



BST Deletion – target has 2 children

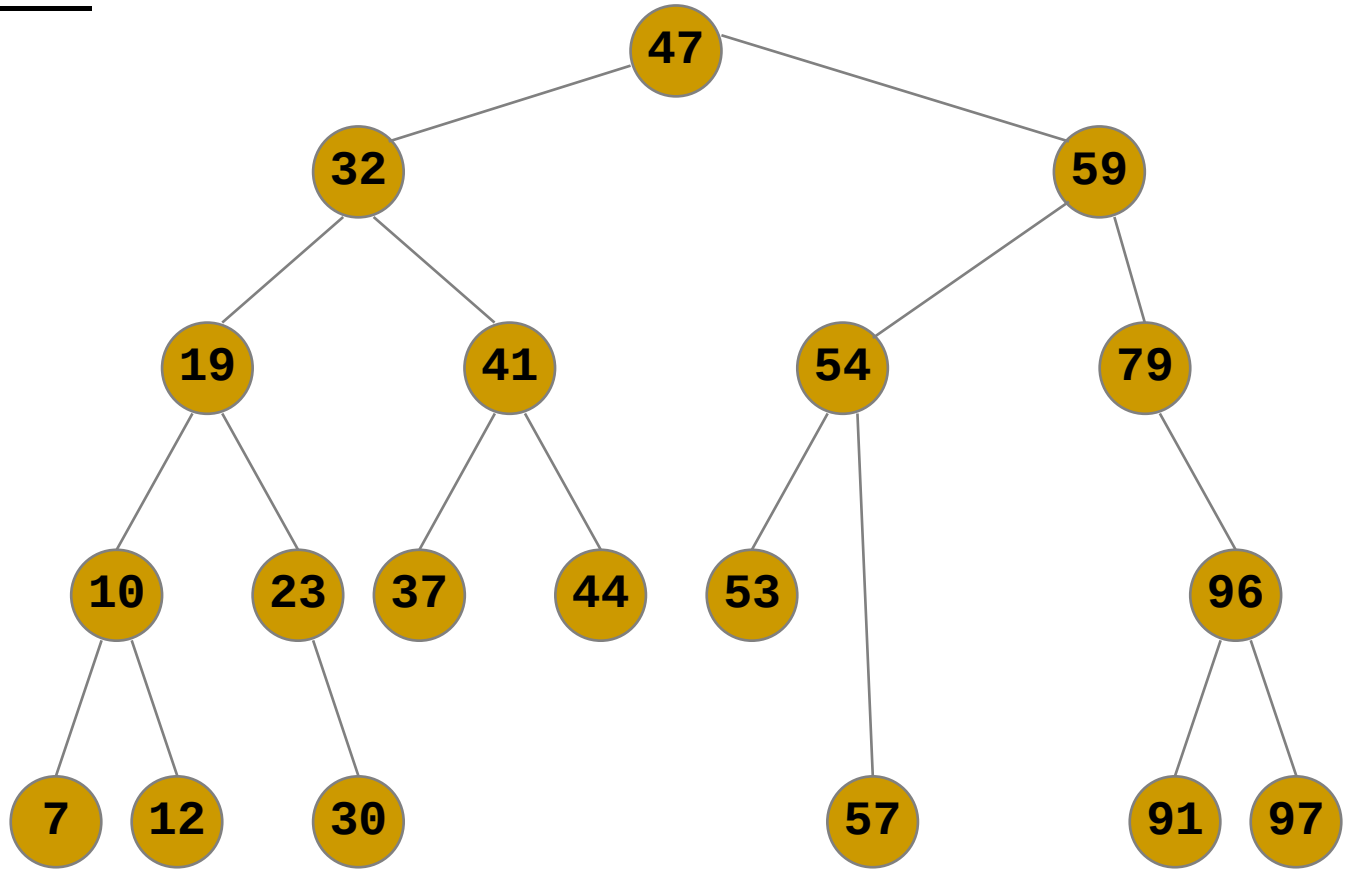
delete 63

find predecessor
attach subtree
attach children
attach predecssor
  to target's
  parent

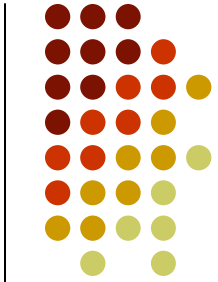


BST Deletion – target has 2 children

delete 63



Deletion algorithm – Phase 1: Finding Node



```
TreeNode<T> deleteItem(TreeNode<T> n, long searchKey) {
  // Returns a reference to the new root.
  // Calls: deleteNode.
    TreeNode<T> newSubtree;
    if (n == null) {
    throw new TreeException("TreeException: Item not found");
    }
    else {
      if (searchKey==n.getItem().getKey()) {
        // item is in the root of some subtree
        n = deleteNode(n);  // delete the node n
      }
      // else search for the item
      else if (searchKey<n.getItem().getKey()) {
        // search the left subtree
        newSubtree = deleteItem(n.getLeft(), searchKey);
        n.setLeft(newSubtree);
      }
      else { // search the right subtree
        newSubtree = deleteItem(n.getRight(), searchKey);
        n.setRight(newSubtree);
      }  // end if
    }  // end if
    return n;
  }  // end deleteItem
```