BST Operation: Cases of Deletion

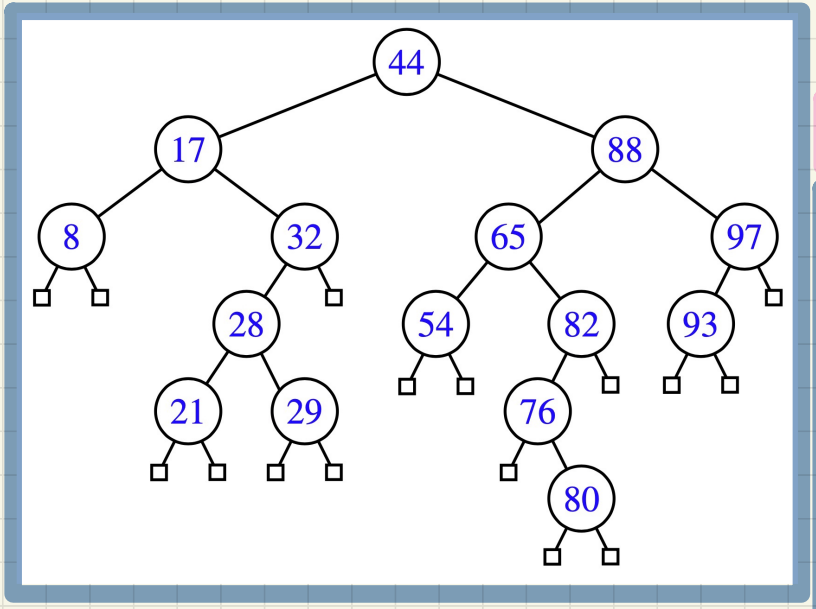
To *delete* an *entry* (with **key** k) from a BST rooted at *node* n:

Let node p be the return value from `search(n, k)`. (root; key of entry to be deleted.)

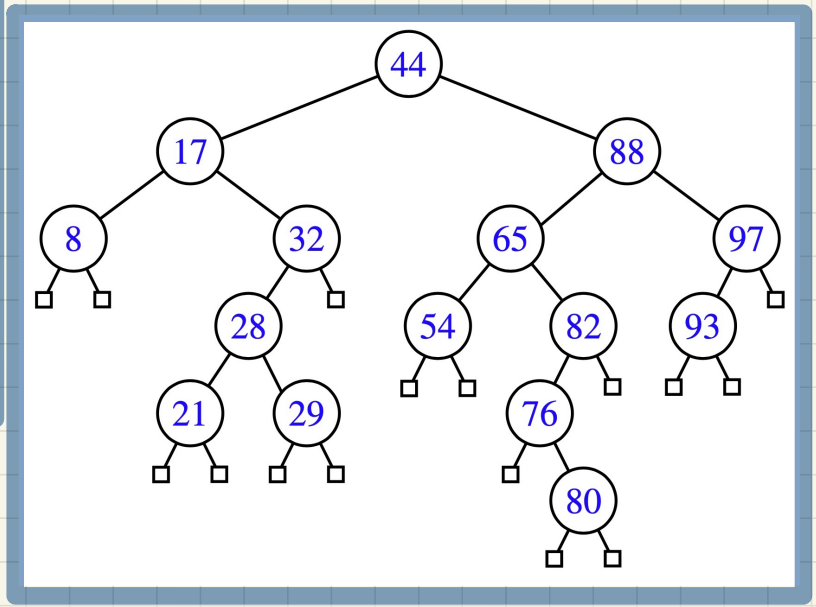
- **Case 1**: Node p is *external*.
  k is not an existing key ⇒ Nothing to remove
- **Case 2**: Both of node p's child nodes are *external*. (internal)
  No "orphan" subtrees to be handled ⇒ Remove p [Still BST?]
- **Case 3**: One of the node p's children, say r, is *internal*.
  - r's sibling is *external* ⇒ Replace node p by node r [Still BST?]
- **Case 4**: Both of node p's children are *internal*.
  - Let r be the *right-most internal node* p's *LST*.
    ⇒ r contains the *largest key s.t. key(r) < key(p)*.
    Exercise: Can r contain the *smallest key s.t. key(r) > key(p)*?
  - Overwrite node p's entry by node r's entry. [Still BST?]
  - r being the *right-most internal node* may have:
    - Two *external child nodes* ⇒ Remove r as in **Case 2**.
    - An *external, RC* & an *internal LC* ⇒ Remove r as in **Case 3**.

Visualizing BST Operation: Deletion

Case 4.1: Delete Entry with Key 17



Case 4.2: Delete Entry with Key 88



After Deletion: Continuous Trinode Restructuring

- *Recall*: ***Deletion*** from a BST results in
  removing a node with zero or one ***internal*** child node.
- After ***deleting*** an existing node, say its child is *n*:
  Case 1: Nodes on *n*'s ***ancestor path*** remain ***balanced***. ⇒ No rotations
  Case 2: At least one of *n*'s ***ancestors*** becomes ***unbalanced***.
  1. Get the **first**/**lowest** ***unbalanced*** node *a* on *n*'s ***ancestor path***.
  2. Get *a*'s ***taller*** child node *b*. [$b \notin$ *n*'s ***ancestor path***]
  3. Choose *b*'s child node *c* as follows:
     - *b*'s two child nodes have ***different*** heights ⇒ *c* is the ***taller*** child
     - *b*'s two child nodes have ***same*** height ⇒ *a*, *b*, *c* slant the ***same*** way
  4. Perform rotation(s) based on the ***alignment*** of *a*, *b*, and *c*:
     - Slanted the ***same*** way ⇒ ***single*** *rotation* on the **middle** node *b*
     - Slanted ***different*** ways ⇒ ***double*** *rotations* on the **lower** node *c*
- As *n*'s ***unbalanced ancestors*** are found, keep applying **Case 2**,
  until **Case 1** is satisfied. [***rotations***]

Trinode Restructuring after Deletion: Single Rotation

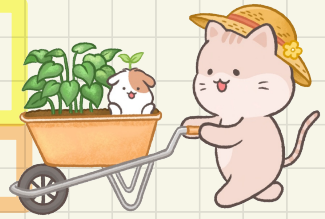
- Insert the following sequence of keys into an empty BST:

  <44, 17, 62, 32, 50, 78, 48, 54, 88>

- Delete 32 from the BST.

Trinode Restructuring after Deletion: Multiple Rotations

- Insert the following sequence of keys into an empty BST:
  <50, 25, 10, 30, 5, 15, 27, 1, 75, 60, 80, 55>
- Delete 80 from the BST.



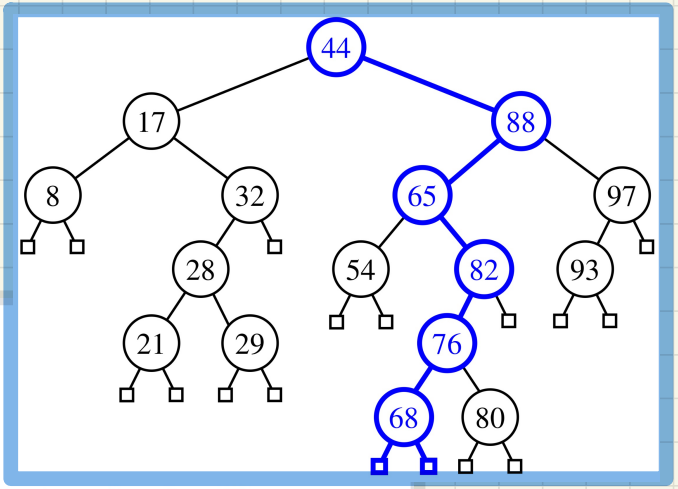
Exercise: BST Construction

Study: *constructExampleTree*

```
public class BSTNode<E> {
   private int key;
   private E value;

   private BSTNode<E> parent;
   private BSTNode<E> left;
   private BSTNode<E> right;

   . . .

   public void setLeft(BSTNode<E> left) {
      this.left = left;
      left.setParent(this);
   }

   public void setRight(BSTNode<E> right) {
      this.right = right;
      right.setParent(this);
   }

}
```

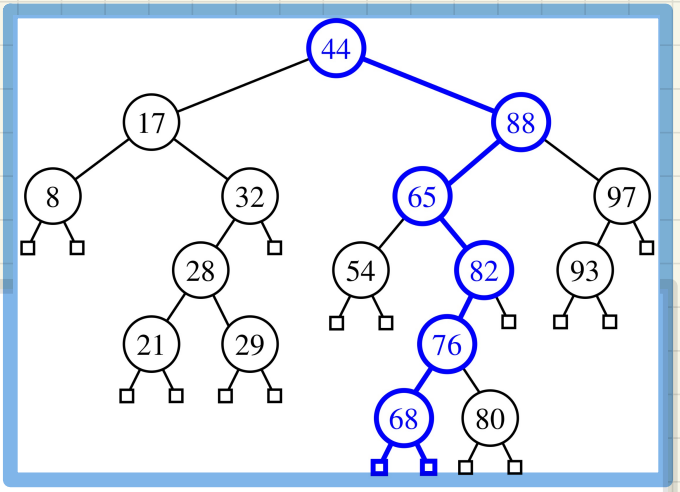
```
BSTNode<String> n44 = new BSTNode<>(44, "Yuna");
BSTNode<String> extN1 = new BSTNode<>();
BSTNode<String> extN2 = new BSTNode<>();
n44.setLeft(extN1);
n44.setRight(extN2);
```

Exercise: BST In-Order Traversal



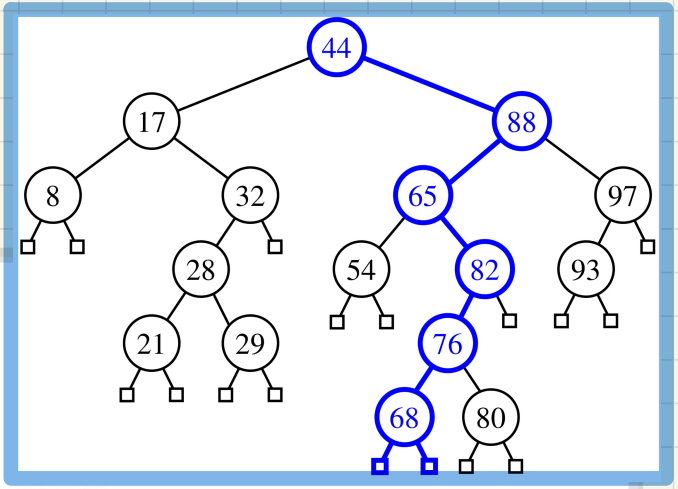

```
@Test
public void test_bst_in_order_traversal() {
    constructExampleTree();

    BSTUtilities<String> u = new BSTUtilities<>();
    ArrayList<BSTNode<String>> inOrderList = u.inOrderTraversal(n44);
    assertTrue(inOrderList.size() == 16 + 17); /* 16 internal nodes + 17 external nodes */
    ArrayList<BSTNode<String>> expectedOrder = new ArrayList<>(Arrays.asList(
                    extN1, n8, extN2,
                n17,
                    extN10, n21, extN11, n28, extN12, n29, extN13, n32, extN3,
            n44,
                    extN5, n54, extN6, n65, extN14, n68, extN15, n76, extN16, n80, extN17, n82, extN7,
                n88,
                    extN8, n93, extN9, n97, extN4
    ));
    assertEquals(expectedOrder, inOrderList);
}
```

Exercise: Trinode Restructuring via a Right Rotation




```
@Test
public void test_bst_right_rotation_1() {
    constructExampleTree();

    BSTUtilities<String> u = new BSTUtilities<>();
    u.rightRotate(n44, n17, n8);

    ArrayList<BSTNode<String>> inOrderList = u.inOrderTraversal(n17);
    assertTrue(inOrderList.size() == 16 + 17); /* 16 internal nodes + 17 external nodes */
    ArrayList<BSTNode<String>> expectedOrder = new ArrayList<>(Arrays.asList(
            extN1, // T1
            n8, // c
            extN2, // T2
            n17, // b
            extN10, n21, extN11, n28, extN12, n29, extN13, n32, extN3, // T3
            n44, // a
            extN5, n54, extN6, n65, extN14, n68, extN15, n76, extN16, n80, extN17, n82, extN7, n88, extN8, n93, extN9, n97, extN4 // T4
    ));
    assertEquals(expectedOrder, inOrderList);
}
```

Exercise: Trinode Restructuring via a Right Rotation




```
@Test
public void test_bst_right_rotation_2() {
    constructExampleTree();

    BSTUtilities<String> u = new BSTUtilities<>();
    u.rightRotate(n32, n28, n21);

    ArrayList<BSTNode<String>> inOrderList = u.inOrderTraversal(n44);
    assertTrue(inOrderList.size() == 16 + 17); /* 16 internal nodes + 17 external nodes */
    ArrayList<BSTNode<String>> expectedOrder = new ArrayList<>(Arrays.asList(
            extN1, n8, extN2, n17,
            extN10, // T1
            n21, // c
            extN11, // T2
            n28, // b
            extN12, n29, extN13, // T3
            n32, // a
            extN3, // T4
            n44, extN5, n54, extN6, n65, extN14, n68, extN15, n76, extN16, n80, extN17, n82, extN7, n88, extN8, n93, extN9, n97, extN4
    ));
    assertEquals(expectedOrder, inOrderList);
}
```