ECE36800 Data structures

Binary search trees (BSTs) and height-balanced BSTs

Chapter 12 (pp. 502-520)

Lu Su

School of Electrical and Computer Engineering
Purdue University

Slides Courtesy: Prof. Cheng-Kok Koh

Overview

- Binary search trees (BSTs)
- Insertion
- Deletion
- Height-balanced BSTs (AVL trees)
  - Rotations

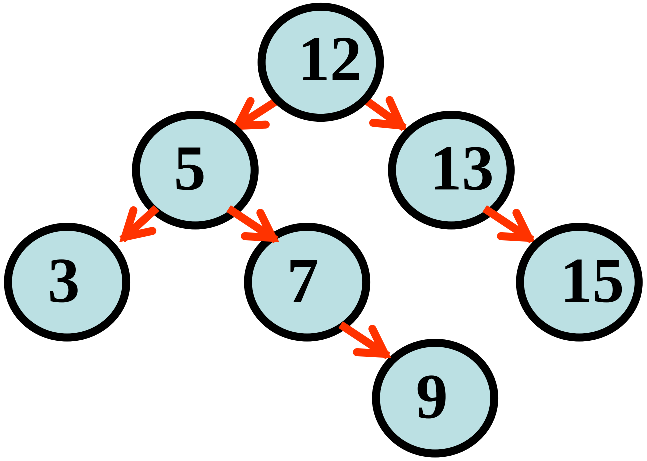
- Reference pages pp. 502-520

Binary search tree (BST)

- Assume that keys are int, and they are stored in the data field of each node
  - The data field may store more than just an integer key
- Keys of descendant nodes on the left of a node are < key of node
  - If we allow duplicate keys in the BST, keys of descendant nodes on the left of a node are ≤ key of node
- Keys of descendant nodes on the right of a node are > key of node
  - If we allow duplicate keys in the BST, Keys of descendant nodes on the right of a node are ≥ key of node
- Inorder traversal yields the keys in ascending order

Binary search

```
Tnode *binary_search(Tnode *root, int key)
{
  Tnode *curr = root;
  while (curr != NULL) {
    if (key == curr->data)
      break;
    else if (key < curr->data)
      curr = curr->left;
    else
      curr = curr->right;
  }
  return curr;
}
```



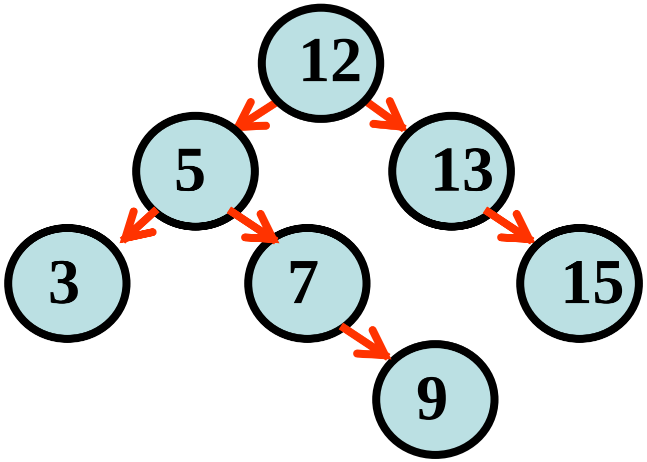
Binary search

- The worst-case time complexity is dependent on the height of binary tree, h
  - In the worst case, the BST is skewed and it is essentially a linked list, h = O(n)
  - When the tree is a complete binary tree, h = O(log n)
- In the best case, we find the key in the first probe, O(1)
- Space complexity is O(1) as it is iterative

Search and insert distinct key

- Search tree for the node that contains the search key
- If the key is in the tree, return the node that contains the key
- If the key is not in the tree, insert a node into the BST with the search key in the data field
  - We must have reached a NULL address to insert a new node
  - Assume function Create() that creates node to be inserted into tree
  - We have to update the location currently storing the NULL address to point to the new node
  - Must use curr and prev to maintain two addresses, just like insertion in linked list
- The search-and-insert function returns the address of the inserted node; if a new node cannot be created, NULL is returned
- Can be generalized to allow duplicate keys

```
Tnode *Search_and_insert(Tnode **root, int key)
{
  Tnode *prev = NULL;
  Tnode *curr = *root;
  while (curr != NULL) {
    if (key == curr->data)
      return (curr);
    prev = curr;
    if (key < curr->data)
      curr = curr->left;
    else
      curr = curr->right;
  } // end while
  Tnode *node = Create(key, NULL, NULL);
  if (node != NULL)
    if (prev == NULL) // empty tree to begin with
      *root = node;   // the root has to be updated
    else
      if (key < prev->data)
        prev->left = node;
      else
        prev->right = node;
  return node;
}
```



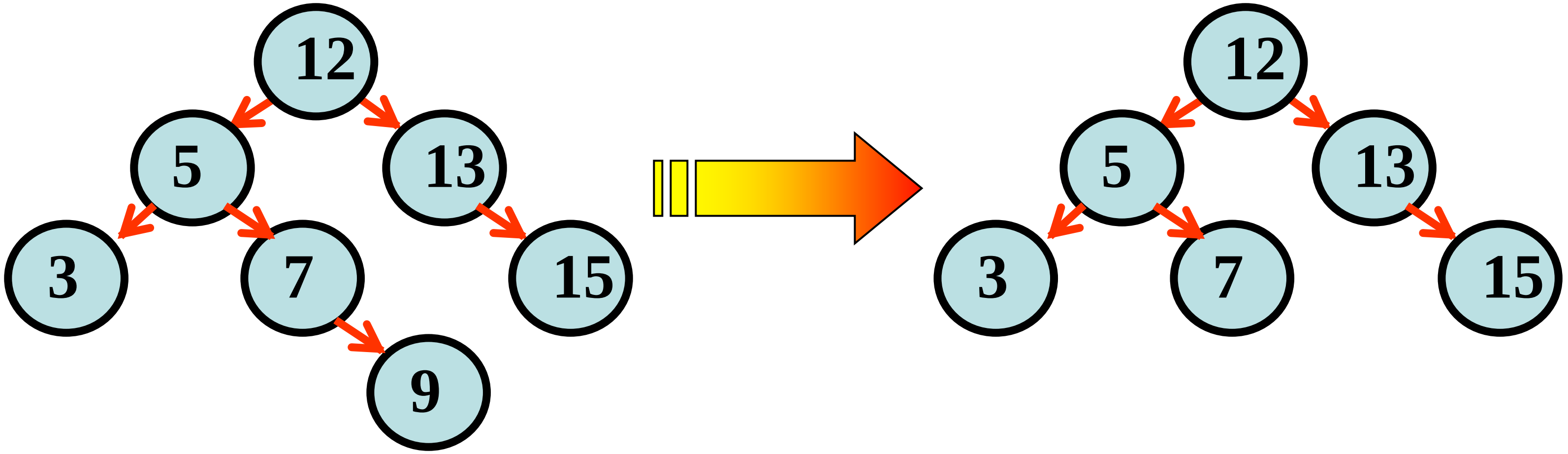
Time complexity and space complexity

- The time complexity is dominated by the search
  - In the worst case, the time complexity is O(h), where h is the tree height, therefore, O(n), where n is the number of nodes in the tree
  - In the best case, we found the key immediately and not have to insert, O(1)
- Space complexity is O(1) as it is still iterative
- If we use the Search_and_insert function to iteratively inserting n distinct integers and then obtain a sorted array (by performing inorder traversal), what are the time complexity and space complexity?

(Search) and delete

- Case 1: The node to be deleted does not have child nodes

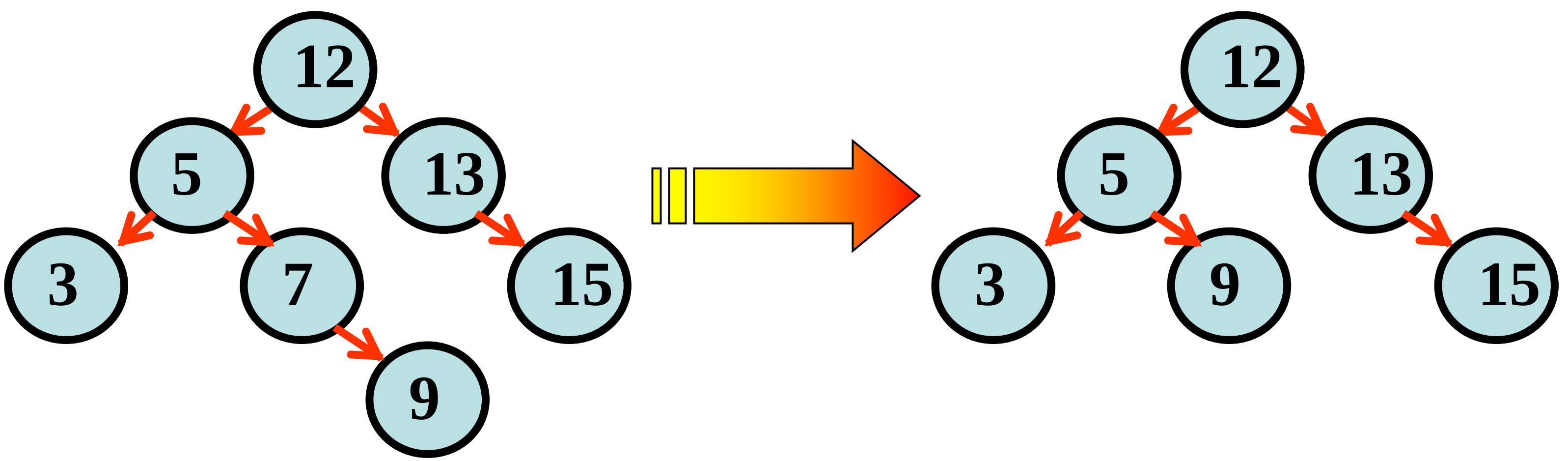
Delete node with key = 9



(Search) and delete

- Case 2: The node to be deleted has one subtree

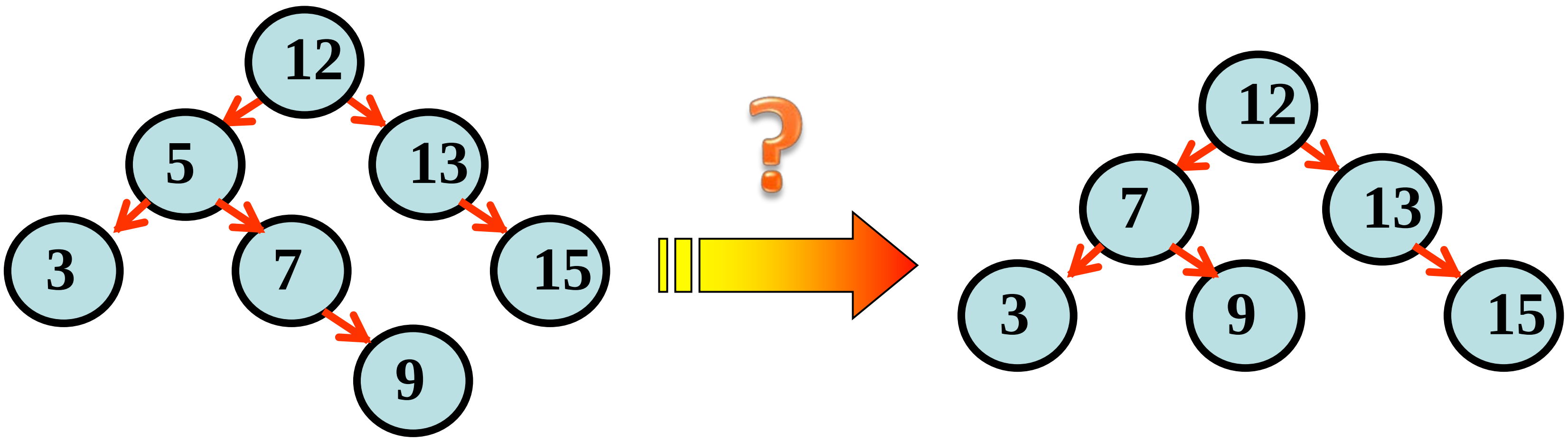
Delete node with key = 7



(Search) and delete

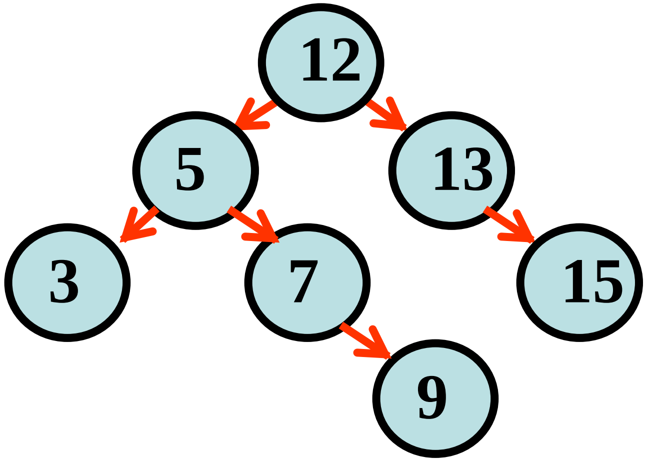
- Case 3: The node to be deleted has two subtrees

Delete node with key = 5



Inorder predecessor and successor

- To find immediate inorder predecessor of a node, go left once and keep going right
  - The predecessor of 12 is 9
  - The deletion of the predecessor node is either case 1 or case 2 since it is the rightmost node, its right field stores NULL
- To find immeidate inorder successor of a node, go right once and keep going left
  - The successor of 12 is 13

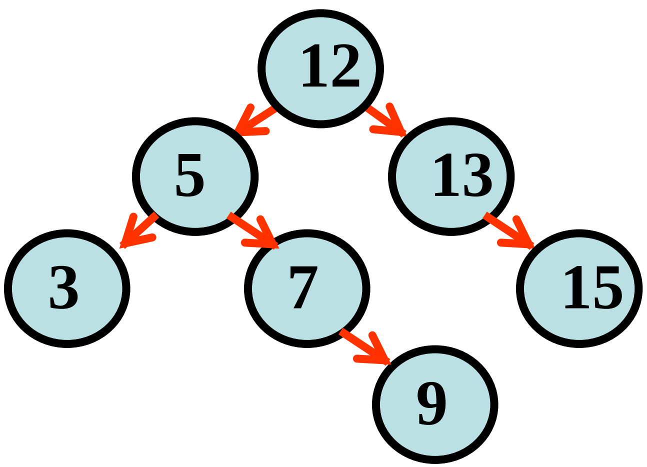


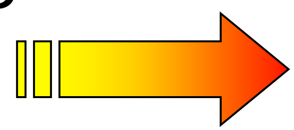
(Search) and delete

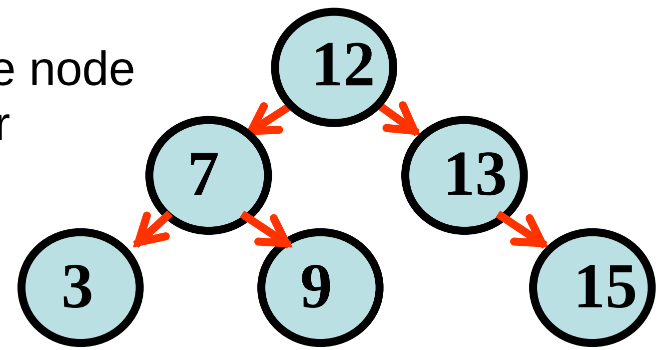
- Case 3: The node to be deleted has two subtrees

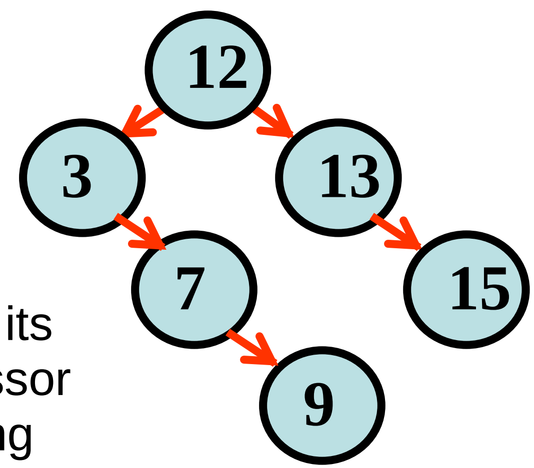
Replace the deleted key with its immediate (inorder) successor and delete the node containing the successor

Delete node with key = 5







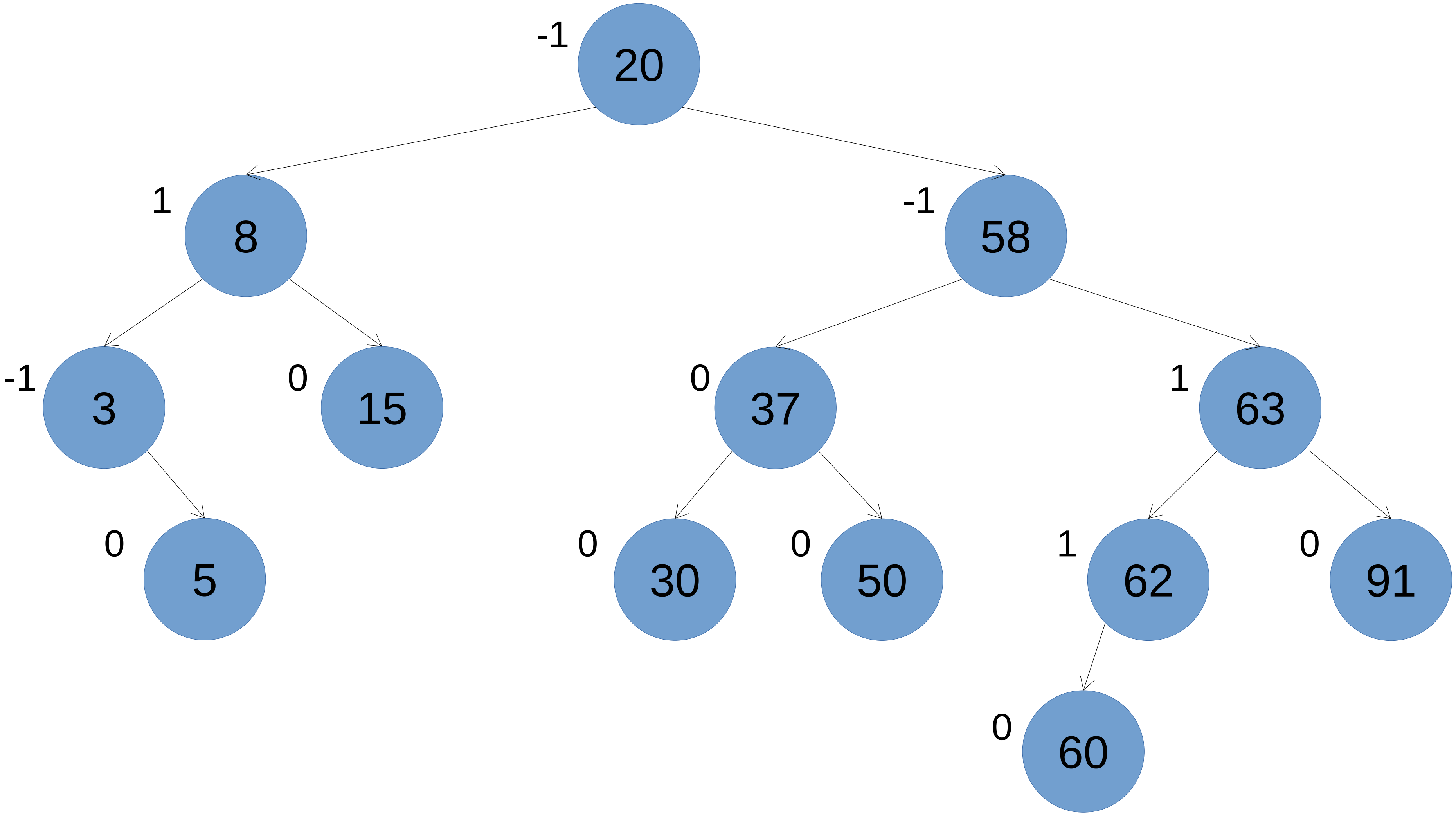


Replace the deleted key with its immediate (inorder) predecessor and delete the node containing the predecessor

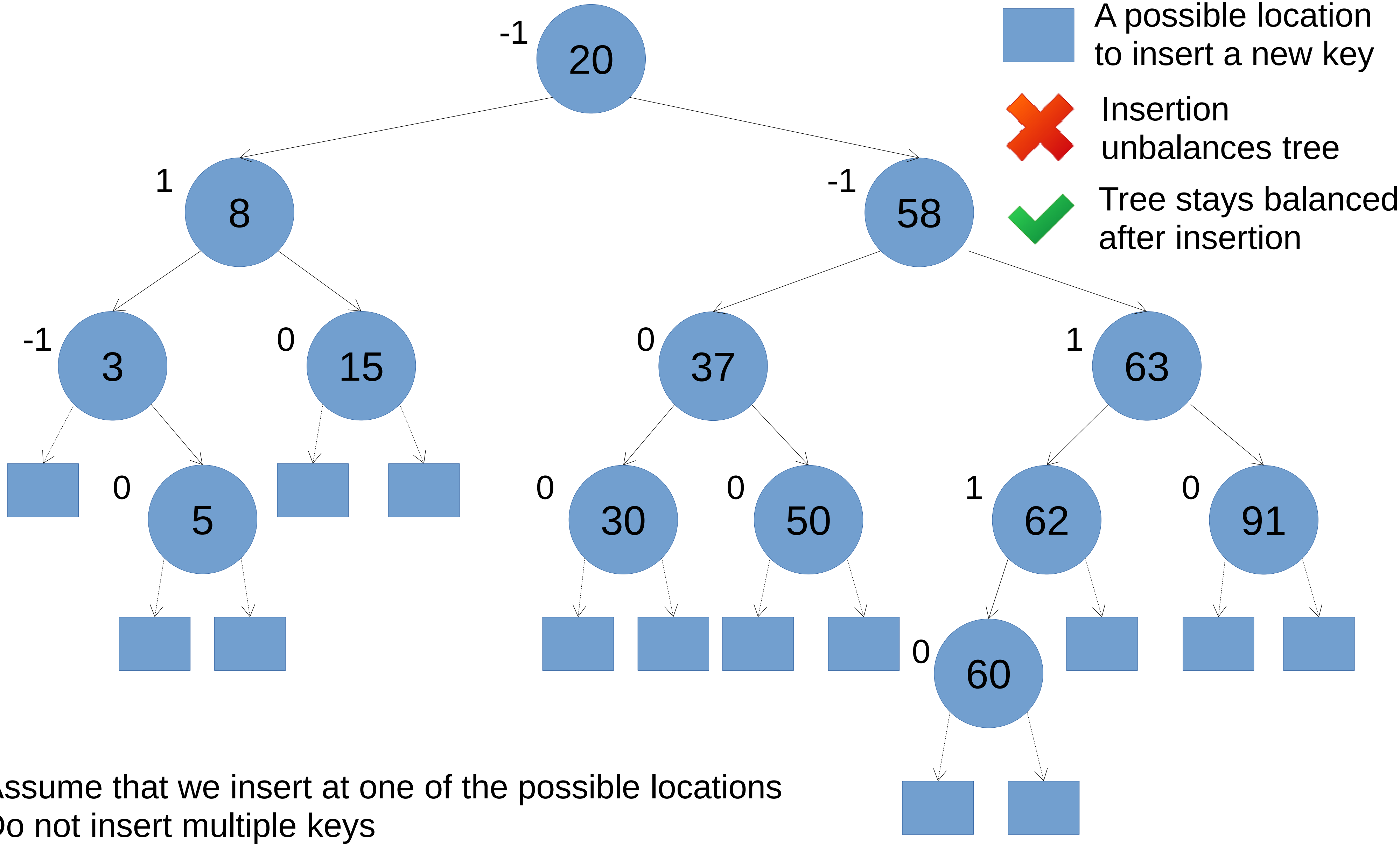
Height balanced BSTs

- Also called AVL trees (Adelson-Velsky and Landis)
- Having a BST that is “balanced” allows the search process to have a worst-case time complexity of O(log n)
- Balance of a node is height of left subtree minus height of right subtree
- A tree is height-balanced if every node has balance 0, 1, or –1
  - Need only two bits to store the balance of a node
- Insertion or deletion of a node may make a height-balanced tree unbalanced, must re-balance the tree

Height-balanced tree example

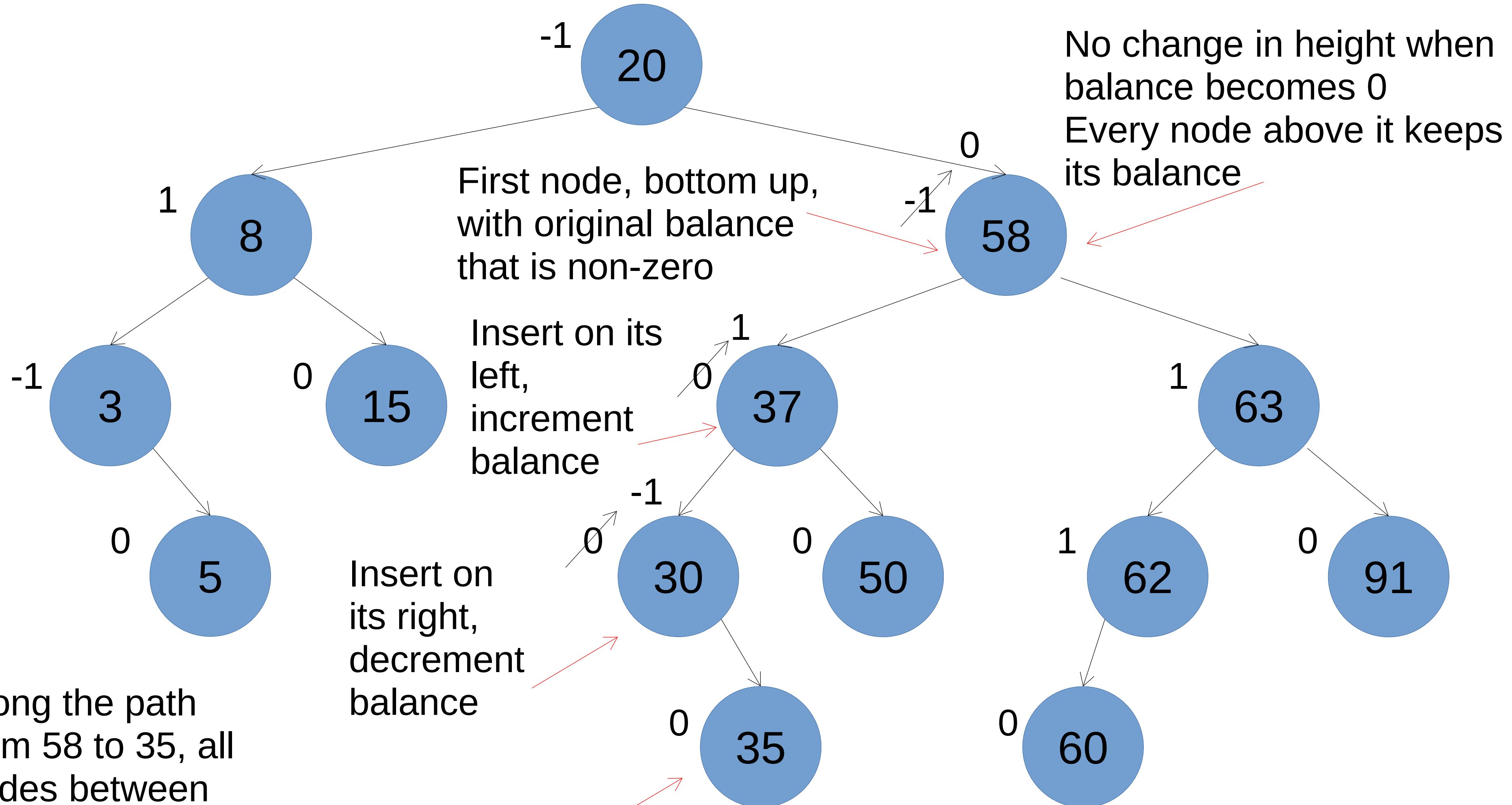


Insert key into Height-balanced BST



Assume that we insert at one of the possible locations
Do not insert multiple keys

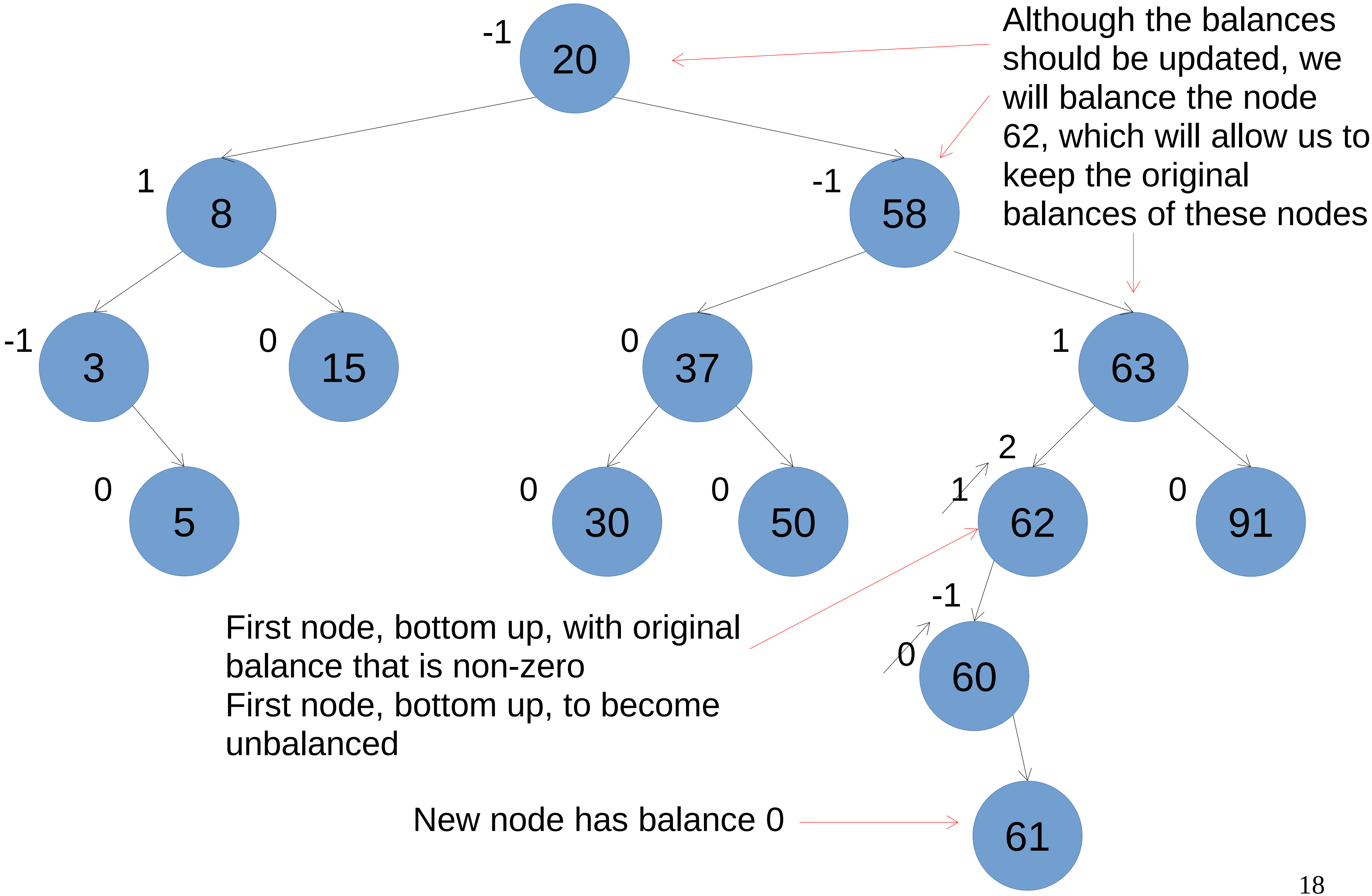
Insertion 35: Update node balances



Along the path from 58 to 35, all nodes between the end points have original balance of 0

New node has balance 0

Insert 61: Update node balances



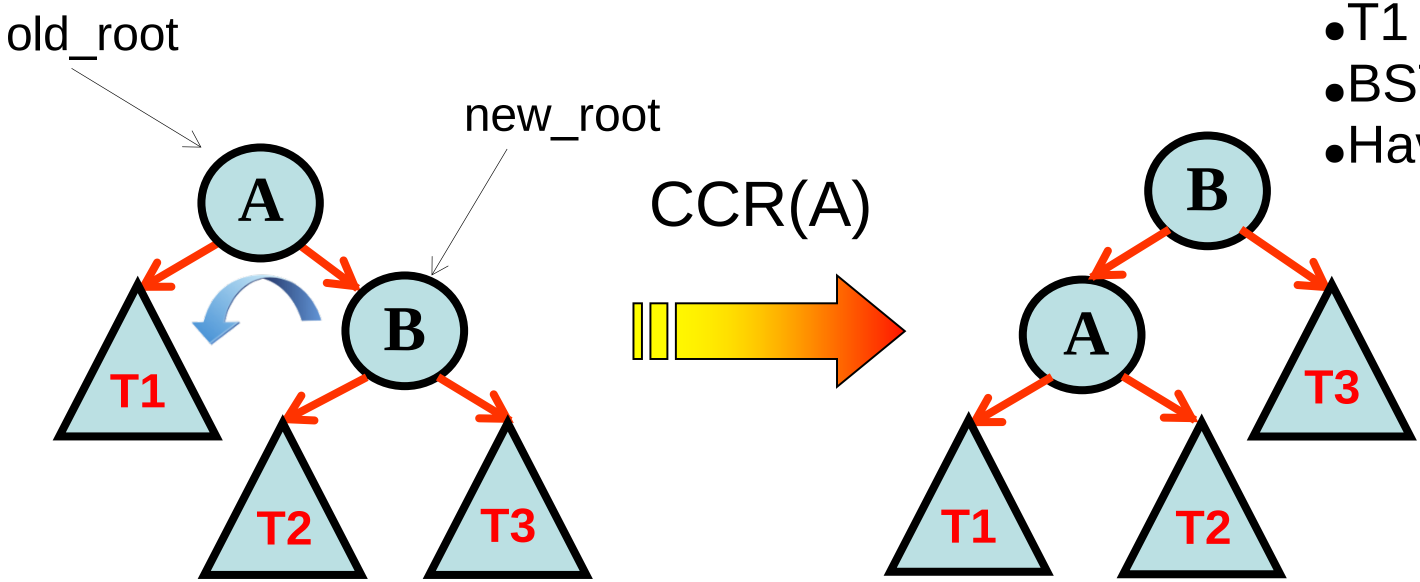
When can a node become unbalanced?

- Assume that an insertion occurs on a node's left or right
- If the original balance of the node is 0, it can only become 1 or -1 after the insertion
  - Still balanced
  - The height of the tree rooted at this node increases
- If the original balance is 1 or -1, the balance becomes 0 if the insertion is on its right or left, respectively
  - Still balanced
  - the height of the subtree rooted at this node stays the same
- If the original balance is 1 or -1, the balance becomes 2 or -2 if the insertion is on its left or right, respectively
  - The node becomes unbalanced
  - The height of the tree rooted at this node increases
- While many nodes may become unbalanced, we will balance the unbalanced node that is closest to the inserted node
  - This is the **youngest ancestor** of the inserted node that has become unbalanced
  - After balancing, all other unbalanced ancestor nodes of the inserted node will automatically become balanced again, and their balances stay intact

Rotations

- We perform **rotation(s)** to balance an unbalanced tree
- Rotation must preserve the property of binary search tree (inorder traversal is in ascending order) and restore height balanceness (all nodes are height balanced)
- Two types of rotation
  - **Counter-clockwise rotation** (**CCR**) or left rotation
  - **Clockwise rotation** (**CR**) or right rotation

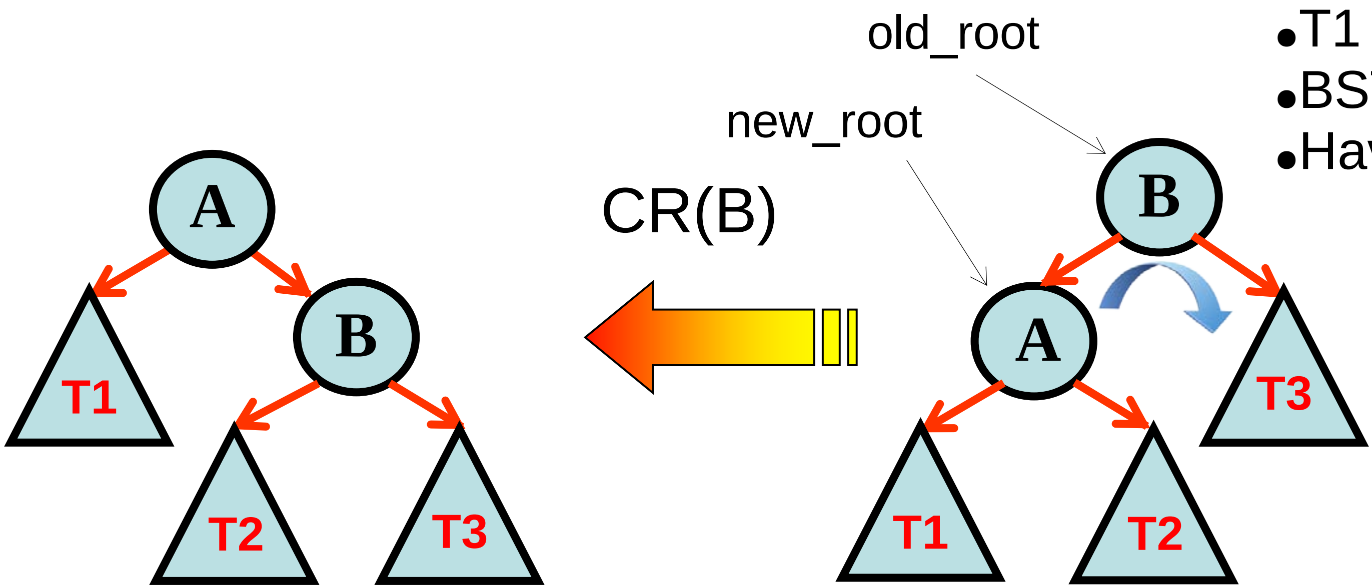
Counter-clockwise rotation (CCR)



- Assume distinct keys:
- T1 < A < T2 < B < T3
- BST property satisfied
- Have to update balances

```
CCR(old_root) {
  new_root = old_root->right;
  old_root->right = new_root->left;
  new_root->left = old_root;
  return new_root;
}
```

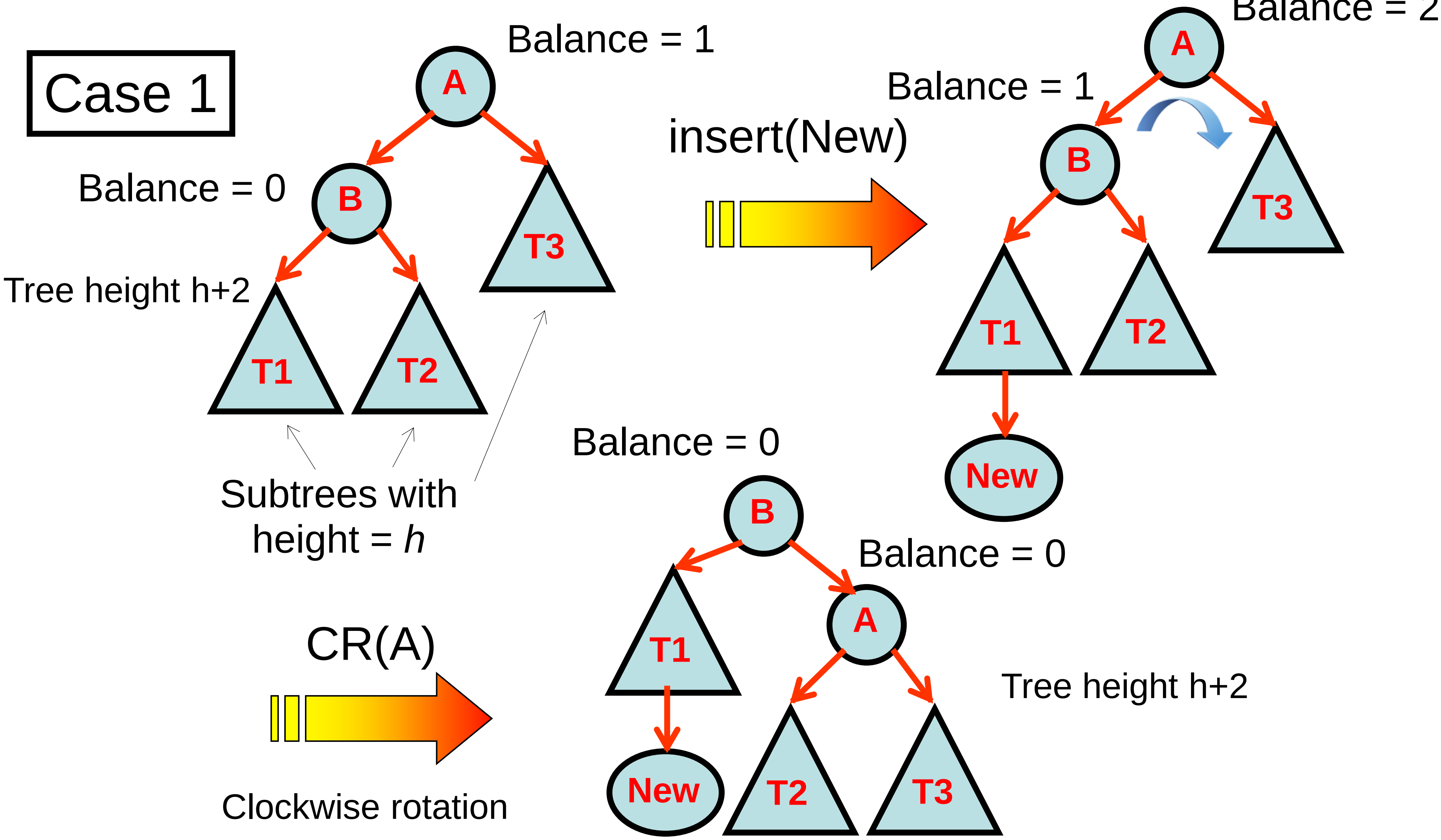
Clockwise rotation (CR)



- Assume distinct keys:
- T1 < A < T2 < B < T3
- BST property satisfied
- Have to update balances

```
CR(old_root) {
  new_root = old_root->left;
  old_root->left = new_root->right;
  new_root->right = old_root;
  return new_root;
}
```

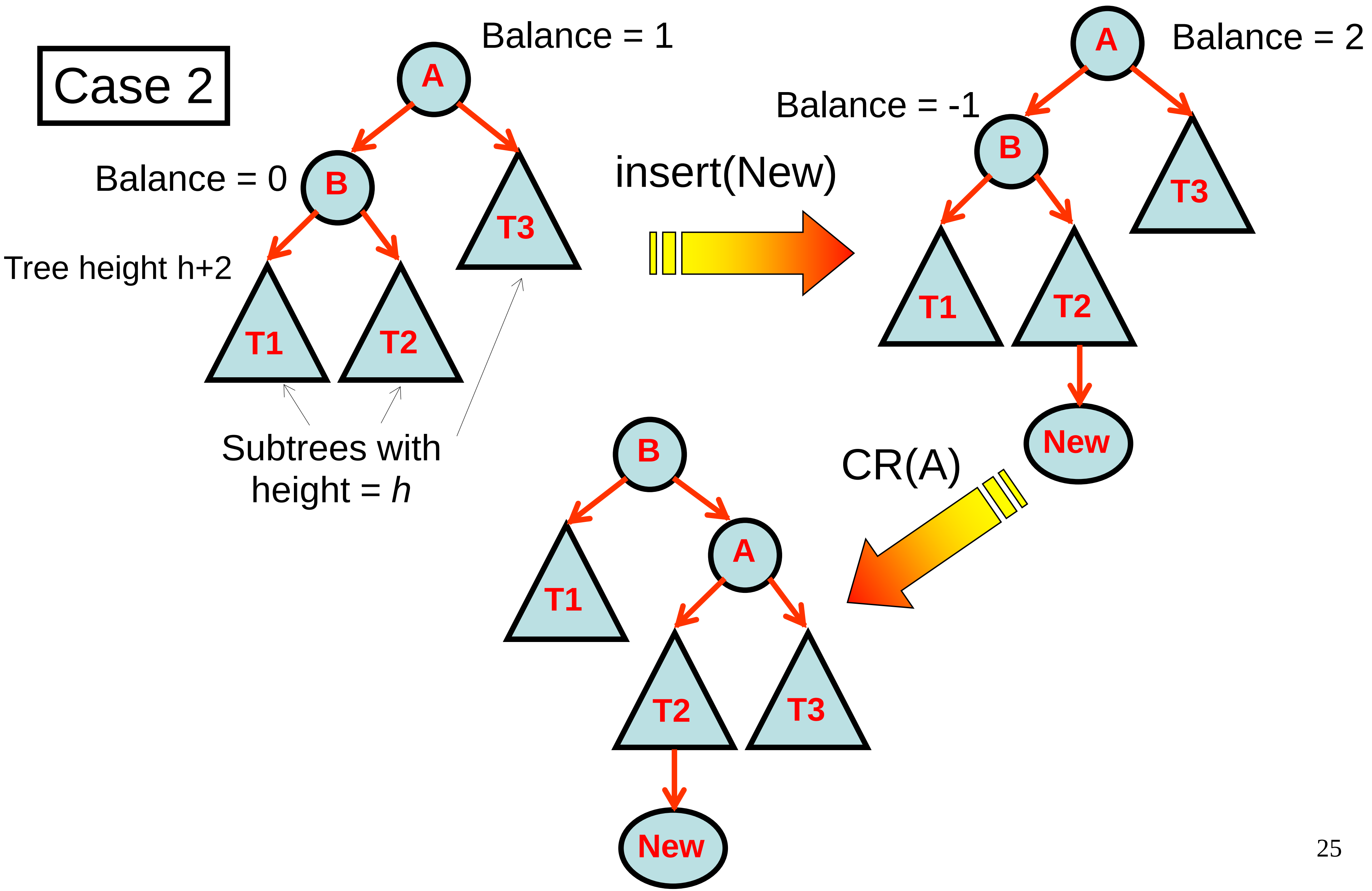
Balancing an unbalanced tree



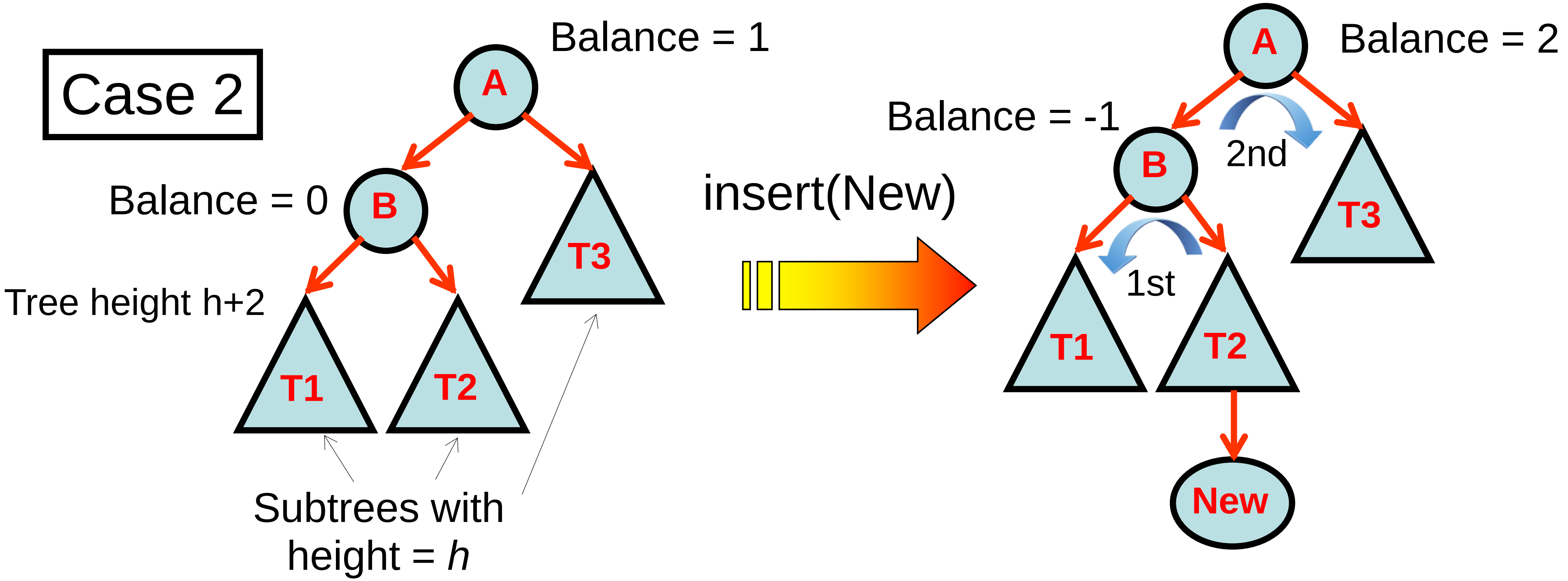
Balancing: Case 1

- A is the youngest ancestor that has become unbalanced, B is the child node of A in the direction where the new node is inserted
- The balances of A and B have the **same polarity** (+2 and +1), i.e., they are both taller on the left
  - Insertion occurs on the same side of A and B
- After rotation, we have a new root of the tree, i.e., B
- If A has a parent before rotation, the parent must now point to B instead
- After rotation, the new tree rooted at B has the **same height** as the old tree rooted at A, i.e. h+2
- All ancestors of B or all old ancestors of A can keep **the same balances**

Balancing an unbalanced tree

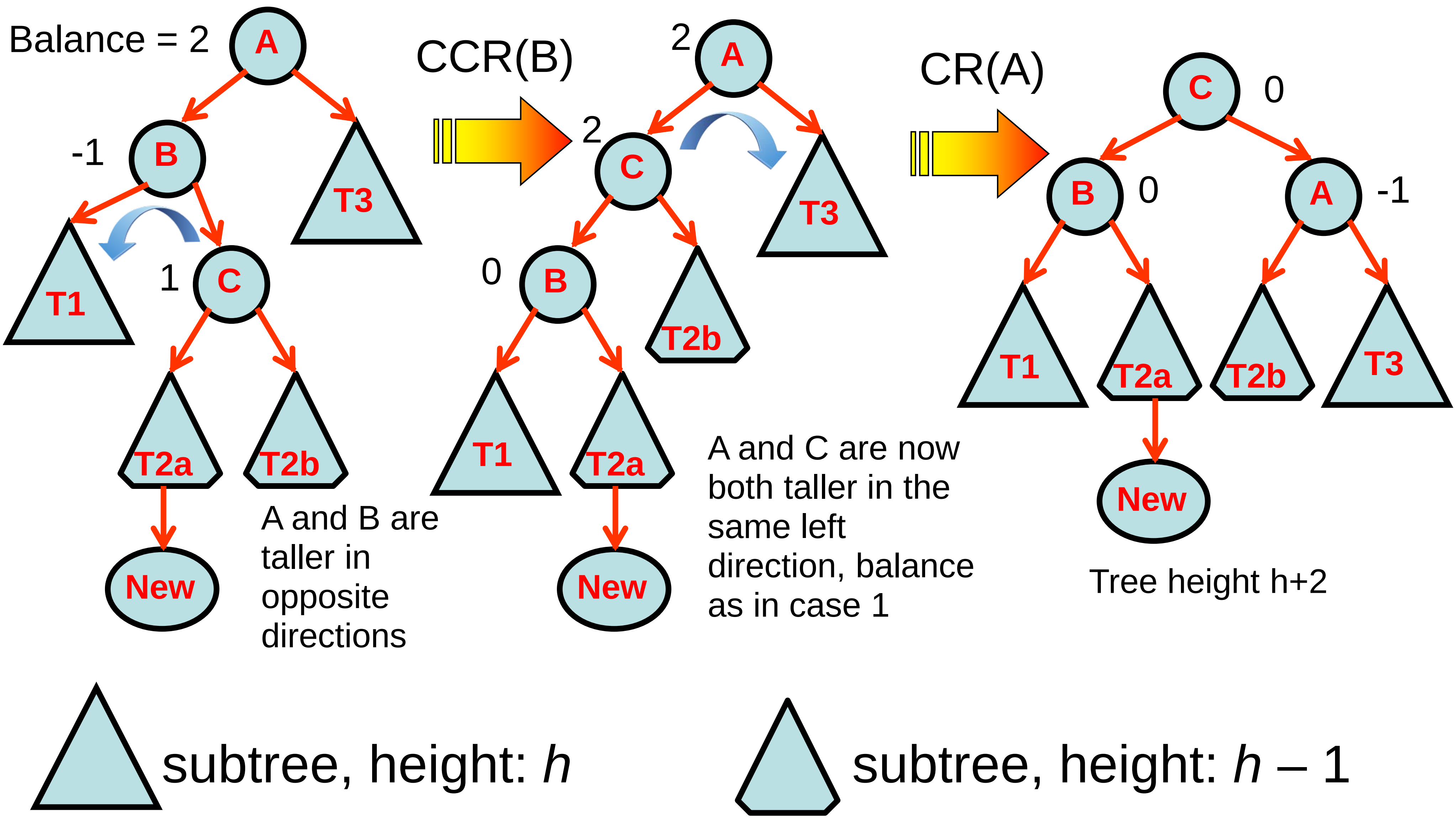


Balancing an unbalanced tree

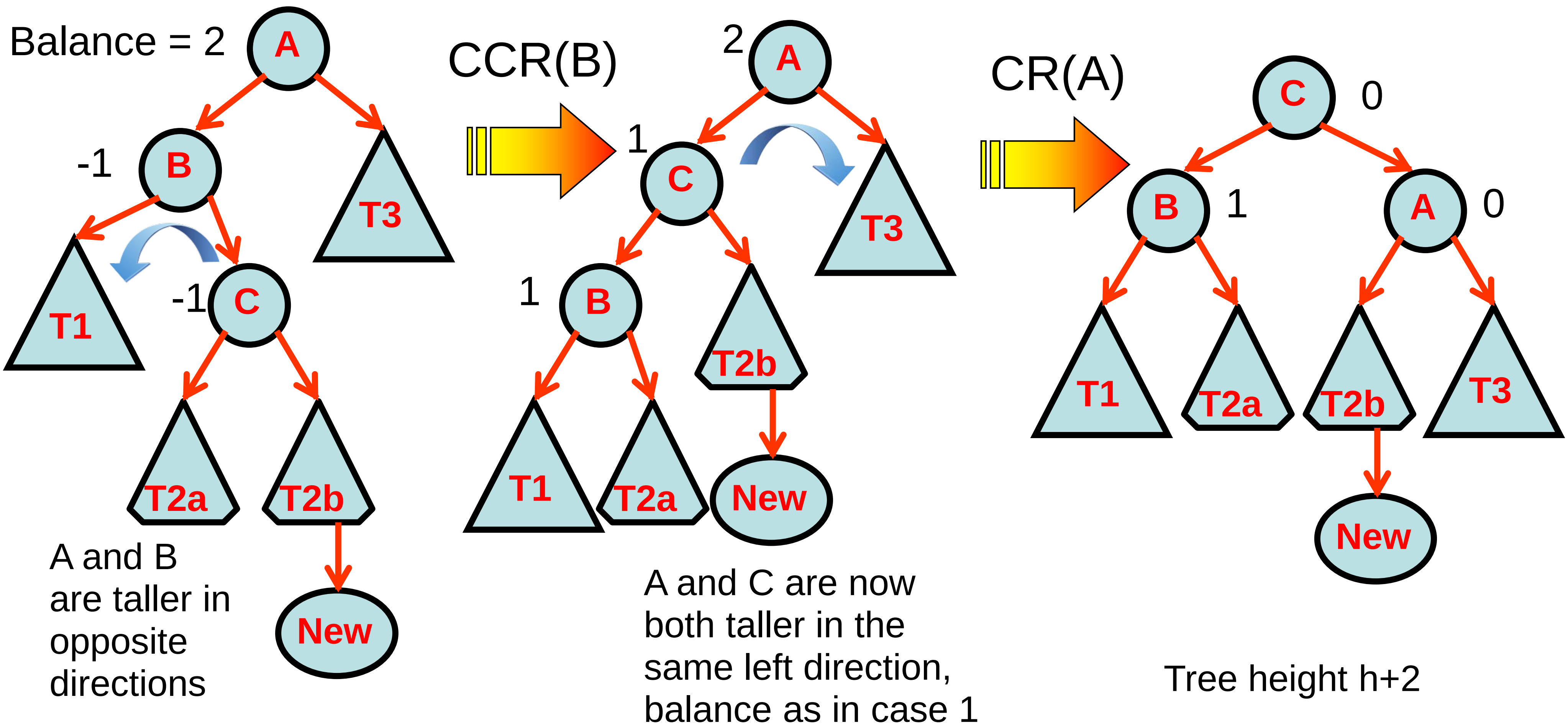


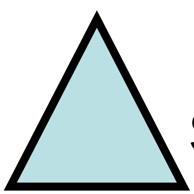
- A is the youngest ancestor that has become unbalanced, B is the child node of A in the direction where the new node is inserted
- Insertion occurs on opposite sides of A and B, i.e., they are taller in opposite sides/directions
- A is taller on its left and B is taller on its right, the balances have different signs
- Requires two rotations, first CCR(B), then CR(A) (rotations in opposite directions)
- Three subcases to update the balances

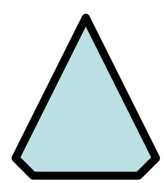
Balancing: Case 2a



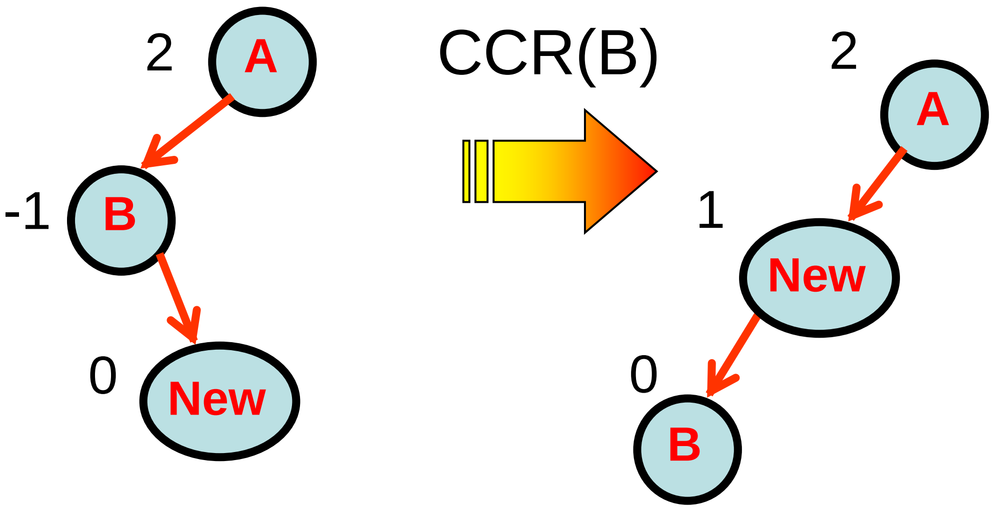
Balancing: Case 2b

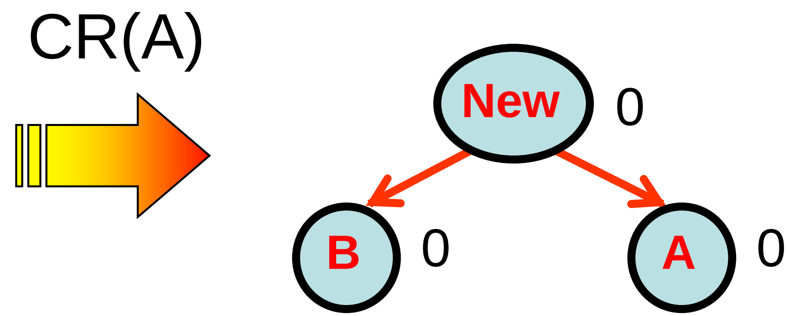


 subtree, height: h

 subtree, height: $h - 1$

Balancing: Case 2c





Height of balanced tree is the same as the height before insertion

A and B are taller in opposite directions

A and New are now both taller in the same left direction, balance as in case 1

Balancing an unbalanced tree

- A is the youngest ancestor that has become unbalanced, B is the child node of A in the direction where the new node is inserted
- There are also Case 3 (mirror of Case 1) with balances of A and B being -2 and -1 and Cases 4a, 4b, 4c (mirrors of Cases 2a, 2b, 2c) with balances of A and B being -2 and 1
- Summary on balancing an unbalanced tree
  - We must perform a final rotation at the youngest ancestor A
- If the balance of the youngest ancestor is +2 (-2), clockwise rotation (counter-clockwise rotation)
  - Check whether we have to perform a rotation prior to that
  - Compare the polarity of the balances of A and B, the child node of the youngest ancestor in the direction the new node is inserted
- If the balances have different signs (+2 and -1, or -2 and +1), we must perform a rotation at B prior to the rotation at A
- The direction of the rotation at B is opposite to that at A
- A rotation takes O(1)

Algorithm for search and insert

- If search is unsuccessful, insert the new key, followed by balancing (or rotations) if necessary
  - Must keep track of the youngest ancestor that has an original balance that is non-zero
  - The youngest ancestor may become unbalanced and we have to perform rotation(s) involving the youngest ancestor
- Update the balances from the youngest ancestor to the parent node of the newly inserted node
- Check whether youngest ancestor has become unbalanced
- If youngest ancestor is unbalanced, check which of the 4 cases applies and apply appropriate rotation(s)
- Update the balances of the nodes involved in the rotation(s)
- The time complexity is dominated by search
  - As he tree is “balanced”, the height is O(log n), the time complexity is O(log n) at worst

```
ya = curr = root; // ya: youngest ancestor with balance 1/-1
pya = pcurr = NULL;  // pya: parent of youngest ancestor
while (curr != NULL)
  if (key == curr->data)  // insert only distinct keys
    return (curr);
  if (key < curr->data)
    q = curr->left;
  else
    q = curr->right;
  if (q != NULL & q->bal != 0) // keep track of youngest
    pya = curr;                // ancestor and its parent
    ya = q;
  pcurr = curr;
  curr = q;
q = Create(key, NULL, NULL); // assume node can be created
q->bal = 0;
If (pcurr == NULL)
  root = q;
else
  if key < pcurr->data
    pcurr->left = q;
  else
    pcurr->right = q;
```

```
// update the balance from youngest ancestor to
// parent of the new node
curr = ya;
while (curr != q)
  if (key < curr->data)
    curr->bal += 1;      // insert on left
    curr = curr->left;
  else
    curr->bal -= 1;      // insert on right
    curr = curr->right;

// check if balancing is required
// balance of ya = -1, 0, or 1, no balancing
if ((ya->bal < 2) && (ya->bal > -2))
  return q;

// find the child into which q is inserted
if (key < ya->data)
  child = ya->left;
else
  child = ya->right
```

```
// the subtree rooted at ya needs balancing
// curr points to the new root of the subtree
// pya has to point to curr after rebalancing

// both ya and child are unbalanced in the same
// direction

if ((ya->bal == 2) && (child->bal == 1))
  curr = CR(ya); // clockwise rotation
  ya->bal = 0;
  child->bal = 0;

if ((ya->bal == -2) && (child->bal == -1))
  curr = CCR(ya); // counter-clockwise rotation
  ya->bal = 0;
  child->bal = 0;
```

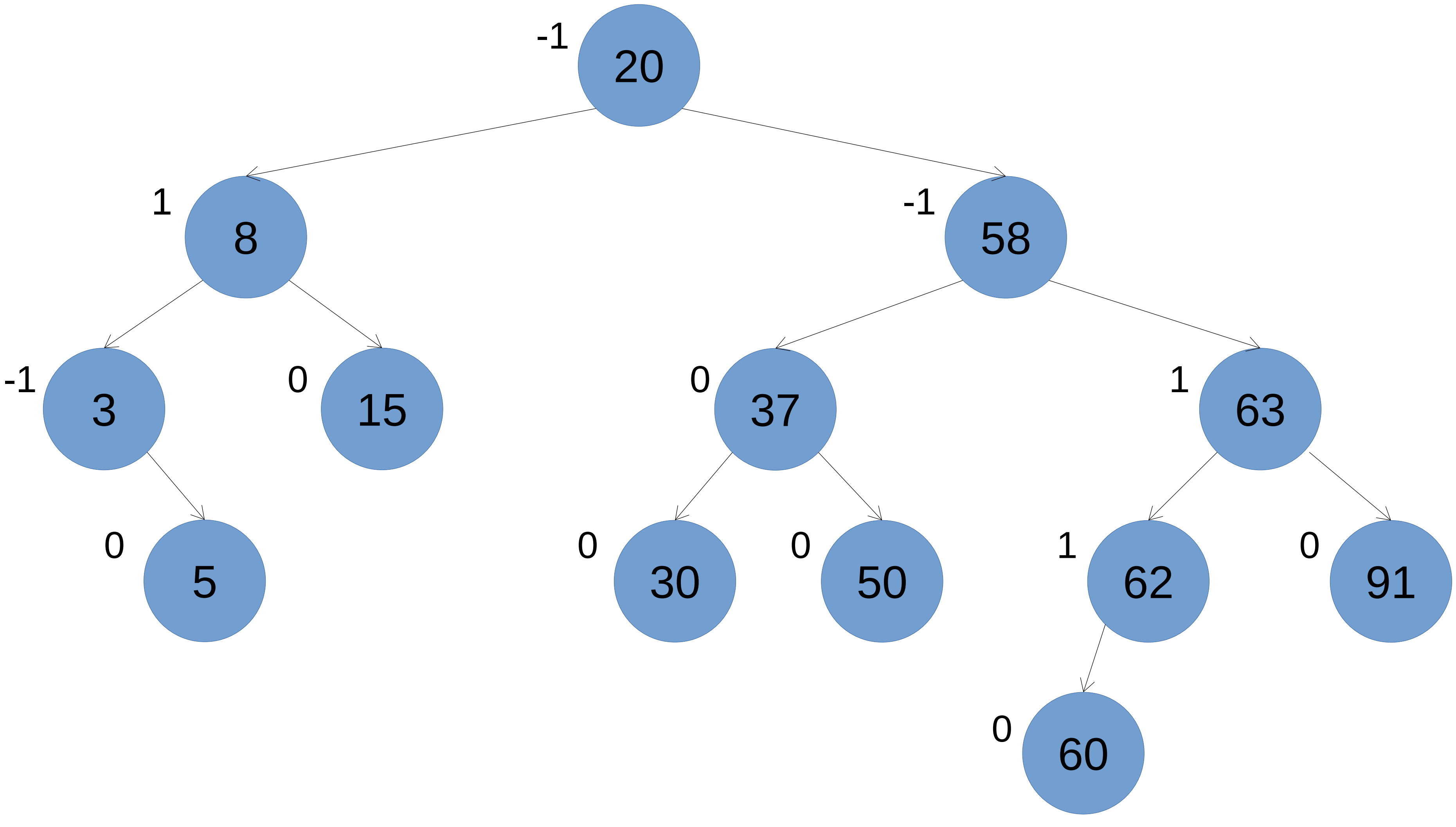
```
// ya and child are unbalanced in opp. Directions

if ((ya->bal == 2) && (child->bal == -1))
  ya->left = CCR(child); // counter-clockwise rotation
  curr = CR(ya);         // clockwise rotation

  if (curr->bal == 0)   // curr is inserted node q
    ya->bal = 0;
    child->bal = 0;
  else
    if (curr->bal == 1) // ori. left subtree of curr
      ya->bal = -1;     // contains q
      child->bal = 0;
    else                // ori. right subtree of curr
      ya->bal = 0;      // contains q
      child->bal = 1;
    curr->bal = 0;      // new root is balanced
```

```
if ((ya->bal == -2) && (child->bal == 1))
  ya->right = CR(child);      // clockwise rotation
  curr = CCR(ya);             // counter-clockwise rotation
  if (curr->bal == 0)    // curr is inserted node q
    ya->bal = 0;
    child->bal = 0;
  else
    if (curr->bal == -1) // ori. right subtree of curr
      ya->bal = 1;       // contains q
      child->bal = 0;
    else                 // ori. left subtree of curr
      ya->bal = 0;       // contains q
      child->bal = -1;
    curr->bal = 0;       // new root is balanced
if (pya == NULL)
  root = curr;
else
  if (key < pya->data)
    pya->left = curr;
  else
    pya->right = curr;
return q;
```

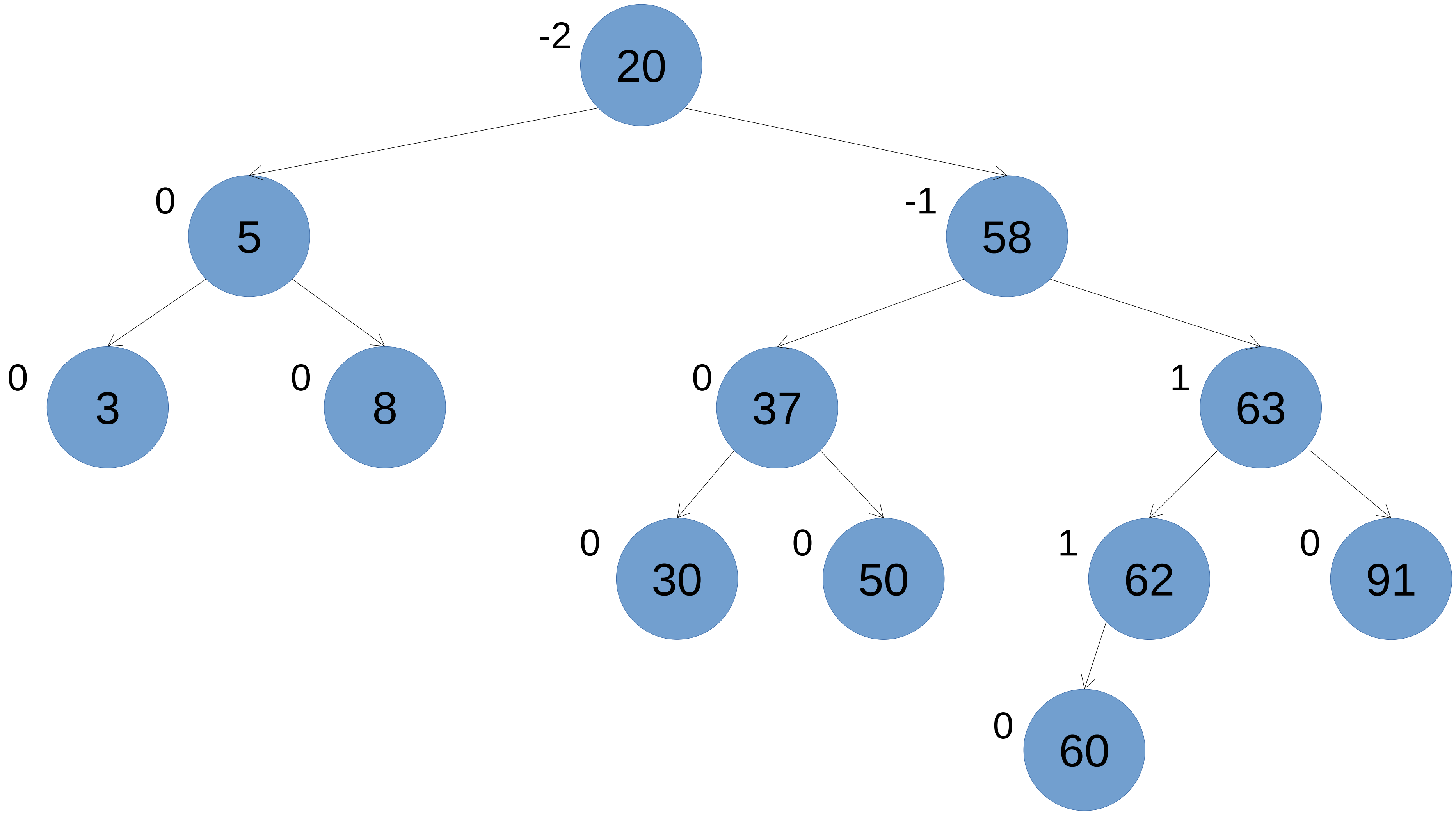
Search and delete

- Search and insert requires at most 2 rotations to maintain height balanceness
  - Balancing at the youngest ancestor node that has become unbalanced is sufficient to keep the height of the subtree the same as before insertion
- Deletion may involve rotations at every level from the parent of the deleted node to the root node
  - The balance of the parent node is incremented (decremented) if the deletion happens on its right (left)
  - If it becomes unbalanced, rotation(s) may result in a subtree that is shorter than the original tree rooted at the parent node
    - If the subtree is shorter, must update the balance the grandparent node and check for height balanceness
    - May have to do that all the way to the root node

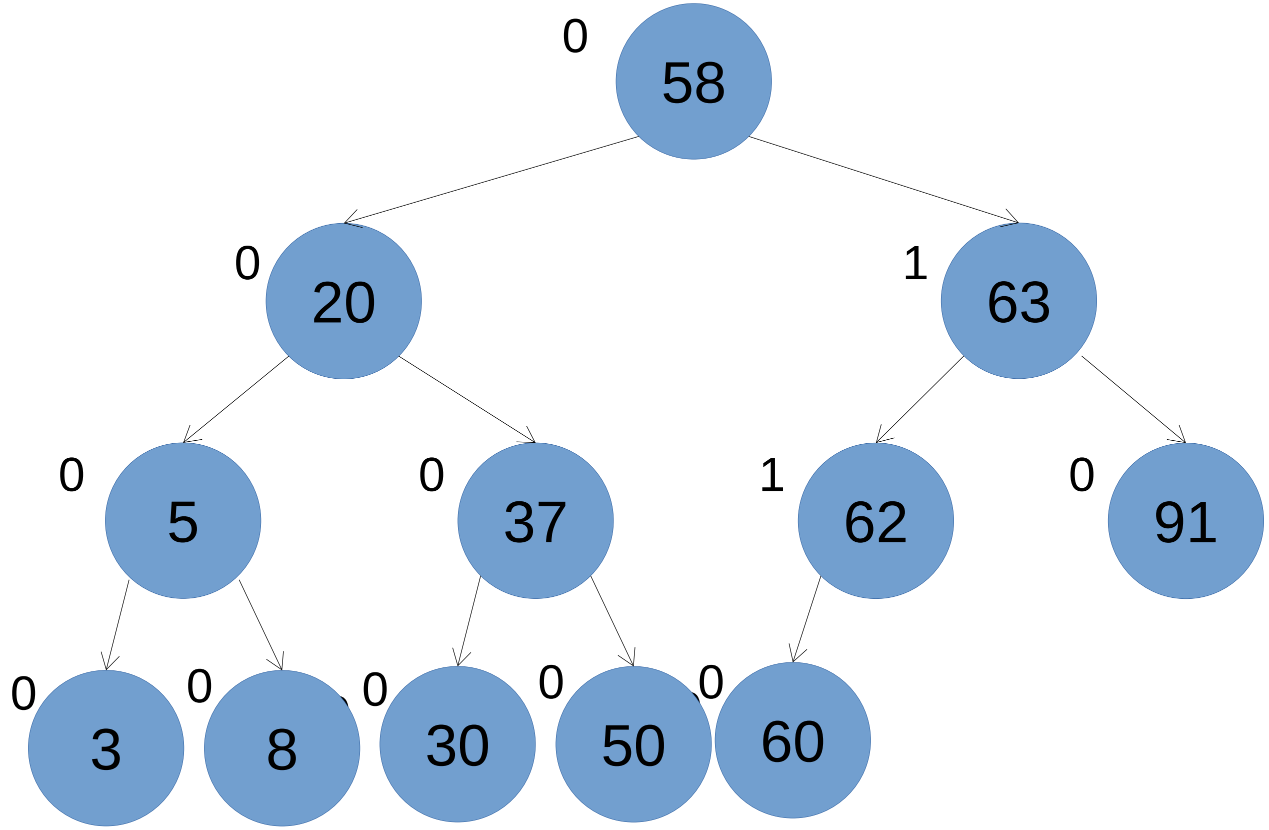
Height-balanced tree example



Height-balanced tree example



Height-balanced tree example



Search and delete

- How do you decide the rotation(s) to be performed at an unbalanced node?
  - Similar to insertion, use the balance of the unbalanced node to determine the final rotation direction
  - To check whether it is necessary to perform a prior rotation, if the balance of the unbalanced node is 2 (-2), check the balance of the left child (right child)
    - Unlike insertion where the balance of the child node could be -1 or 1, the balance of the child node here could be -1, 0, 1
  - If the balances are of different signs, two rotations (as in insertion with the first rotation performed in the opposite direction of the final rotation)
    - The balances 2 and 0 (-2 and 0) are of the same sign

Search and delete

- How do you determine whether there is a change in height?
  - Examine the change in the balance of the root node of a tree before deletion and the balance of the root node of the tree obtained after deletion (and necessary rotations)
- For insertion
  - If the balance changes from 0 to -1 or 1, there is a change in height (subtrees have same height originally, one side grows taller, so overall tree height changes)
  - If the balance changes from -1 or 1 to 0, there is no change in height (subtrees have different heights originally, shorter side grows taller, but overall tree height stays the same)
- For deletion, apply similar logic
  - If the balance changes from 0 to -1 or 1, there is no change in height (subtrees have same height originally, one side becomes shorter, but overall, same height)
  - If the balance changes from -1 or 1 to 0, there is a change in height (subtrees have different heights originally, taller side becomes shorter, so overall tree height changes)

Search and delete

- The time complexity is again dominated by search, which is O(log n) as the height is O(log n)
- There may be 2 rotations at each level, which is again O(log n)
- There is a need to remember the path from the parent of the deleted node to the root node
  - A recursive search and delete using call stack has space complexity of O(log n)
  - A user-defined stack can also be used, space complexity is still O(log n)