Self-Balancing Binary Search Trees

EECS3101 E:
Design and Analysis of Algorithms
Fall 2025

CHEN-WEI WANG

Learning Outcomes of this Lecture

This module is designed to help you understand:

- When the ***Worst-Case RT*** of a ***BST Search*** Occurs
- ***Height-Balance*** Property
- Review: Insertion & Deletion on a BST
- Performing ***Rotations*** to Restore Tree ***Balance***

Implementation: Generic BST Nodes

```
public class BSTNode<E> {
  private int key; /* key */
  private E value; /* value */
  private BSTNode<E> parent; /* unique parent node */
  private BSTNode<E> left; /* left child node */
  private BSTNode<E> right; /* right child node */

  public BSTNode() { ... }
  public BSTNode(int key, E value) { ... }

  public boolean isExternal() {
    return this.getLeft() == null && this.getRight() == null;
  }
  public boolean isInternal() {
    return !this.isExternal();
  }
  public int getKey() { ... }
  public void setKey(int key) { ... }
  public E getValue() { ... }
  public void setValue(E value) { ... }
  public BSTNode<E> getParent() { ... }
  public void setParent(BSTNode<E> parent) { ... }
  public BSTNode<E> getLeft() { ... }
  public void setLeft(BSTNode<E> left) { ... }
  public BSTNode<E> getRight() { ... }
  public void setRight(BSTNode<E> right) { ... }
}
```

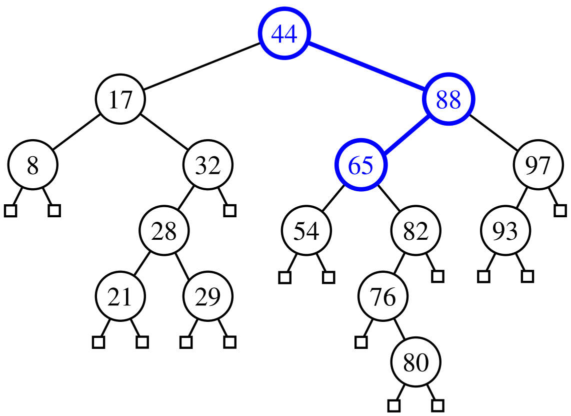

Implementing BST Operation: Searching

Given a *BST* rooted at node ***p***, to locate a particular ***node*** whose ***key*** matches ***k***, we may view it as a ***decision tree***.

```
public BSTNode<E> search(BSTNode<E> p, int k) {
  BSTNode<E> result = null;
  if(p.isExternal()) {
    result = p; /* unsuccessful search */
  }
  else if(p.getKey() == k) {
    result = p; /* successful search */
  }
  else if(k < p.getKey()) {
    result = search(p.getLeft(), k); /* recur on LST */
  }
  else if(k > p.getKey()) {
    result = search(p.getRight(), k); /* recur on RST */
  }
  return result;
}
```

Visualizing BST Operation: Searching (1)

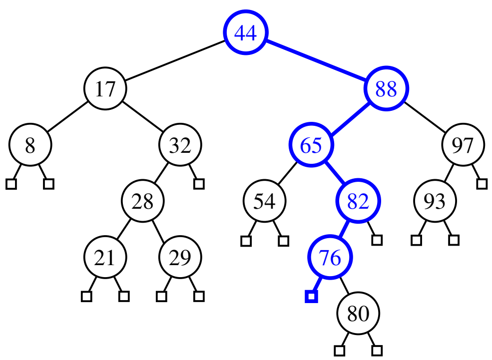
A ***successful*** search for ***key 65***:



The ***internal node*** storing key 65 is **returned**.

Visualizing BST Operation: Searching (2)

- An ***unsuccessful*** search for ***key 68***:



The ***external, left child node*** of the ***internal node*** storing ***key 76*** is **returned**.

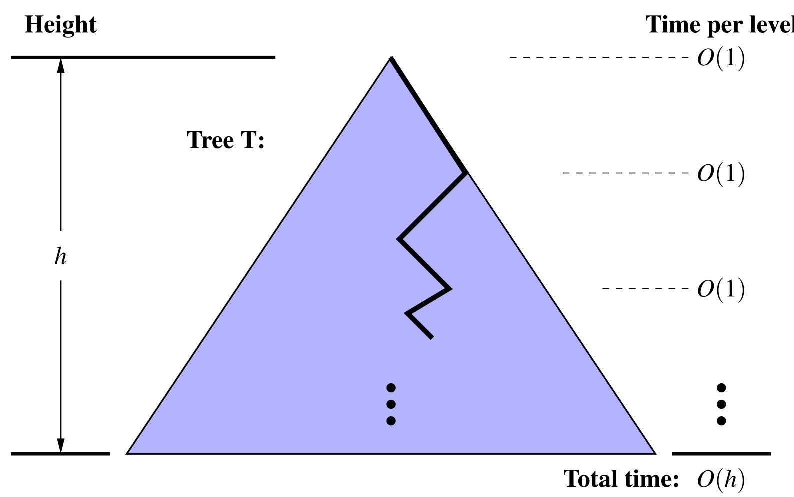
- *Exercise* : Provide ***keys*** for different ***external nodes*** to be **returned**.

Testing BST Operation: Searching

```java
@Test
public void test_binary_search_trees_search() {
  BSTNode<String> n28 = new BSTNode<>(28, "alan");
  BSTNode<String> n21 = new BSTNode<>(21, "mark");
  BSTNode<String> n35 = new BSTNode<>(35, "tom");
  BSTNode<String> extN1 = new BSTNode<>();
  BSTNode<String> extN2 = new BSTNode<>();
  BSTNode<String> extN3 = new BSTNode<>();
  BSTNode<String> extN4 = new BSTNode<>();
  n28.setLeft(n21); n21.setParent(n28);
  n28.setRight(n35); n35.setParent(n28);
  n21.setLeft(extN1); extN1.setParent(n21);
  n21.setRight(extN2); extN2.setParent(n21);
  n35.setLeft(extN3); extN3.setParent(n35);
  n35.setRight(extN4); extN4.setParent(n35);

  BSTUtilities<String> u = new BSTUtilities<>();
  /* search existing keys */
  assertTrue(n28 == u.search(n28, 28));
  assertTrue(n21 == u.search(n28, 21));
  assertTrue(n35 == u.search(n28, 35));
  /* search non-existing keys */
  assertTrue(extN1 == u.search(n28, 17)); /* *17* < 21 */
  assertTrue(extN2 == u.search(n28, 23)); /* 21 < *23* < 28 */
  assertTrue(extN3 == u.search(n28, 33)); /* 28 < *33* < 35 */
  assertTrue(extN4 == u.search(n28, 38)); /* 35 < *38* */
}
```


RT of BST Operation: Searching (1)



RT of BST Operation: Searching (2)

- Recursive calls of `search` are made on a ***path*** which
  - Starts from the **root**
  - Goes down one ***level*** at a time
    RT of deciding from each node to go to LST or RST? [***O(1)***]
  - Stops when the key is found or when a ***leaf*** is reached
    Maximum number of nodes visited by the search? [$h+1$]

  ∴ RT of ***search on a BST*** is $O(h)$
- Recall: Given a BT with ***n*** nodes, the ***height h*** is bounded as:

  $$log(n+1)-1 \leq h \leq n-1$$

  - ***Best*** RT of a ***binary search*** is ***O(log(n))*** [***balanced*** BST]
  - ***Worst*** RT of a ***binary search*** is ***O(n)*** [***ill-balanced*** BST]
- ***Binary search*** on non-linear vs. linear structures:

	Search on a **BST**	Binary Search on a **Sorted Array**
START	Root of BST	Middle of Array
PROGRESS	LST or RST	Left Half or Right Half of Array
BEST RT	***O(log(n))***	***O(log(n))***
WORST RT	***O(n)***	

Sketch of BST Operation: Insertion

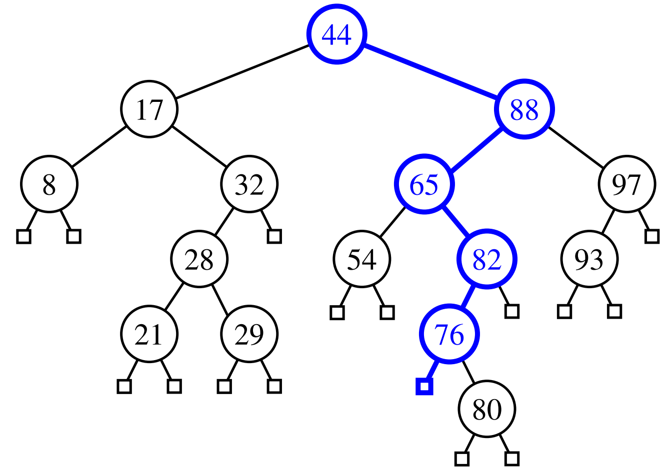
To *insert* an ***entry*** (with **key** *k* & **value** *v*) into a BST rooted at ***node*** *n*:

- Let node ***p*** be the return value from `search(n, k)`.
- If ***p*** is an ***internal node***
  ⇒ Key *k* exists in the BST.
  ⇒ Set *p*'s value to *v*.
- If ***p*** is an ***external node***
  ⇒ Key *k* deos **not** exist in the BST.
  ⇒ Set *p*'s key and value to *k* and *v*.

Running time? [$O(h)$]

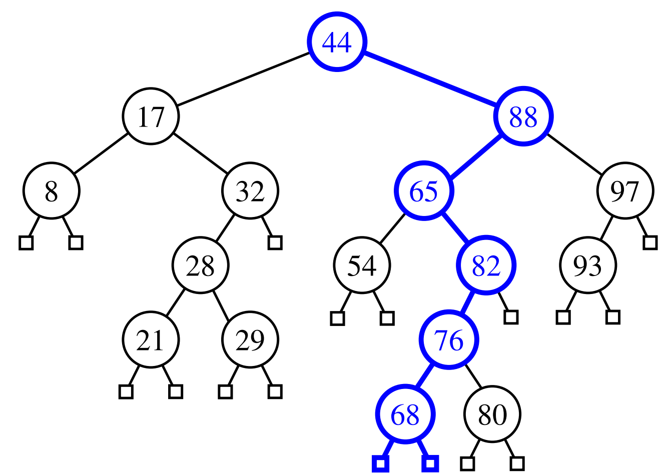
Visualizing BST Operation: Insertion (1)

Before *inserting* an entry with ***key 68*** into the following BST:



Visualizing BST Operation: Insertion (2)

After *inserting* an entry with ***key 68*** into the following BST:



Exercise on BST Operation: Insertion

Exercise: In `BSTUtilities` class, ***implement*** and ***test*** the `void insert(BSTNode<E> p, int k, E v)` method.

Sketch of BST Operation: Deletion

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

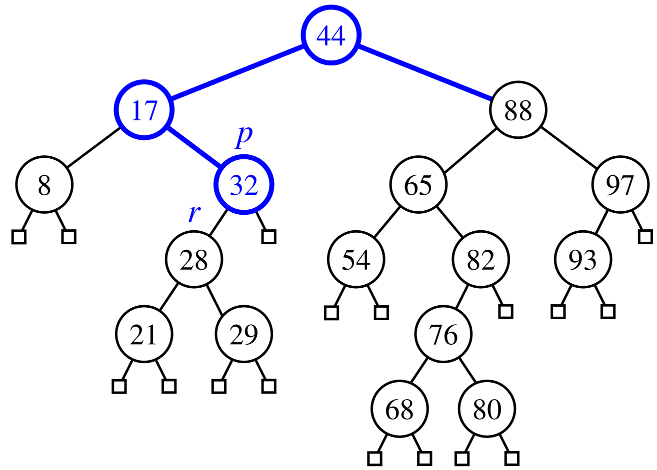
Let node ***p*** be the return value from `search(n, k)`.

- **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***.
  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**.

Running time? [$O(h)$]

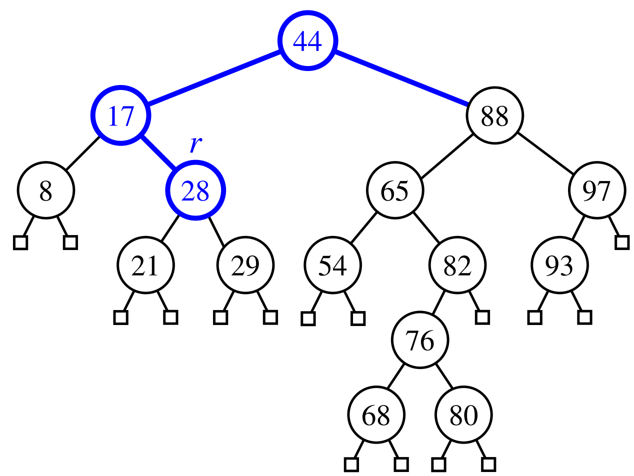
Visualizing BST Operation: Deletion (1.1)

(**Case 3**) Before *deleting* the node storing ***key 32***:



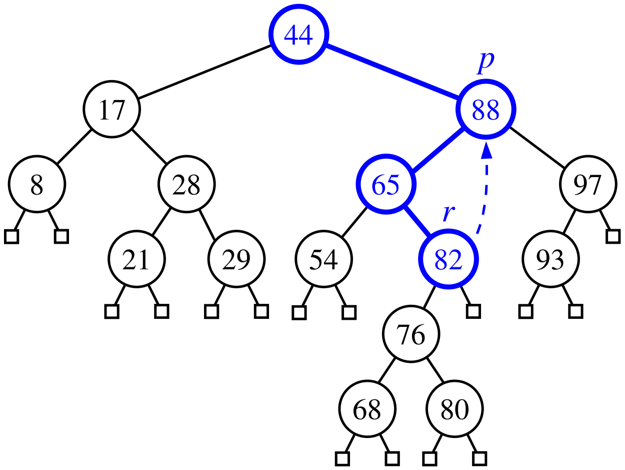
Visualizing BST Operation: Deletion (1.2)

(**Case 3**) After *deleting* the node storing ***key 32***:



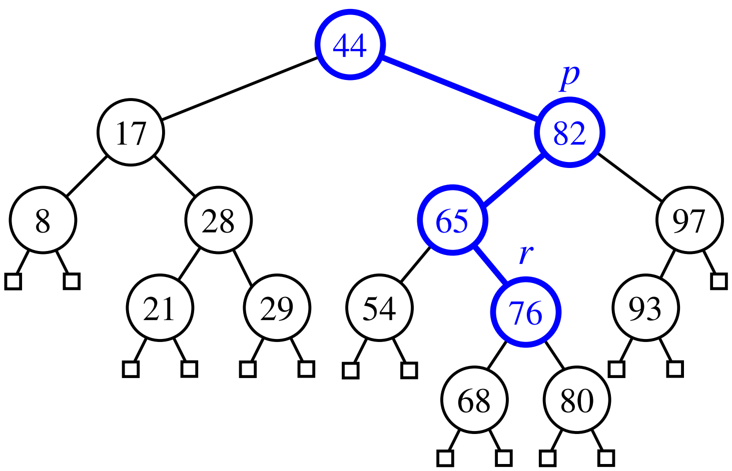
Visualizing BST Operation: Deletion (2.1)

(**Case 4**) Before *deleting* the node storing ***key 88***:



Visualizing BST Operation: Deletion (2.2)

(**Case 4**) After *deleting* the node storing ***key 88***:



Exercise on BST Operation: Deletion

Exercise: In `BSTUtilities` class, ***implement*** and ***test*** the `void delete(BSTNode<E> p, int k)` method.

Balanced Binary Search Trees: Motivation

- After ***insertions*** into a BST, the ***worst-case RT*** of a ***search*** occurs when the ***height*** h is at its ***maximum***: ***O(n)***:
  - e.g., Entries were inserted in an decreasing order of their keys
    $\langle 100, 75, 68, 60, 50, 1 \rangle$
    $\Rightarrow$ ***One-path***, ***left-slanted*** BST
  - e.g., Entries were inserted in an increasing order of their keys
    $\langle 1, 50, 60, 68, 75, 100 \rangle$
    $\Rightarrow$ ***One-path***, ***right-slanted*** BST
  - e.g., Last entry's key is in-between keys of the previous two entries
    $\langle 1, 100, 50, 75, 60, 68 \rangle$
    $\Rightarrow$ ***One-path***, ***side-alternating*** BST
- To avoid the worst-case RT ($\because$ a ***ill-balanced tree***), we need to take actions *as soon as* the tree becomes ***unbalanced***.

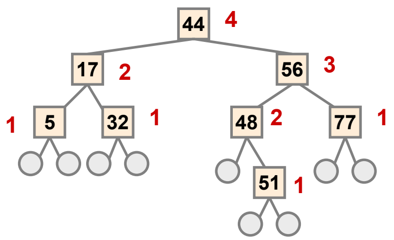
Balanced Binary Search Trees: Definition

- Given a node p, the ***height*** of the subtree rooted at p is:

$$\textit{height}(p) = \begin{cases} 0 & \text{if } p \text{ is } \textit{external} \\ 1 + \textbf{MAX}\left(\{\ \textit{height}(c) \mid \textit{parent}(c) = p\ \}\right) & \text{if } p \text{ is } \textit{internal} \end{cases}$$

- A ***balanced*** BST T satisfies the ***height-balance property*** :

  For every ***internal node*** n, ***heights*** of n's child nodes differ $\leq$ *1*.

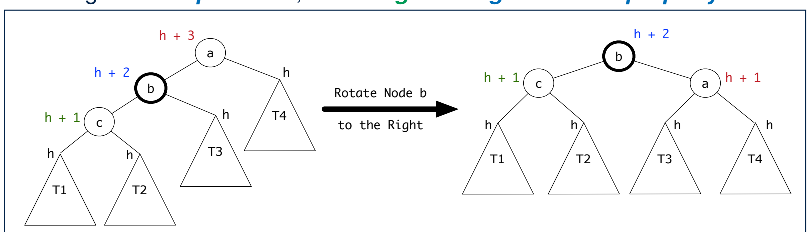


Q: Is the above tree a ***balanced BST***? ✓
Q: Will the tree remain ***balanced*** after inserting 55? ×
Q: Will the tree remain ***balanced*** after inserting 63? ✓

Fixing Unbalanced BST: Rotations

A tree ***rotation*** is performed:

- When the latest **insertion**/**deletion** creates ***unbalanced*** nodes, along the ***ancestor path*** of the node being inserted/deleted.
- To change the ***shape*** of tree, ***restoring*** the ***height-balance property***



Q. An ***in-order traversal*** on the resulting tree?
A. Still produces a sequence of ***sorted keys*** $\langle T_1,\ c,\ T_2,\ b,\ T_3,\ a,\ T_4 \rangle$

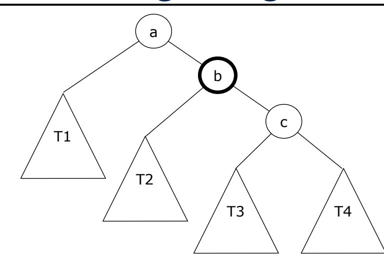
- After ***rotating*** node b to the right:
  - Heights of ***descendants*** (b, c, T_1, T_2, T_3) and ***sibling*** (T_4) stay ***unchanged***.
  - Height of ***parent*** (a) is ***decreased by 1***.
  - ⇒ ***Balance*** of node a was ***restored*** by the ***rotation***.

After Insertions: Trinode Restructuring via Rotation(s)

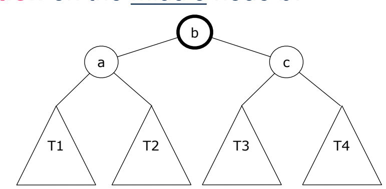
After ***inserting*** a new node n:

- **Case 1**: Nodes on n's ***ancestor path*** remain ***balanced***.
  ⇒ No rotations needed
- **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 child node b in n's ***ancestor path***.
  3. Get b's child node c in n's ***ancestor path***.
  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

Trinode Restructuring: Single, Left Rotation



After a ***left rotation*** on the middle node b:



BST property maintained? $\langle T_1,\ a,\ T_2,\ b,\ T_3,\ c,\ T_4 \rangle$

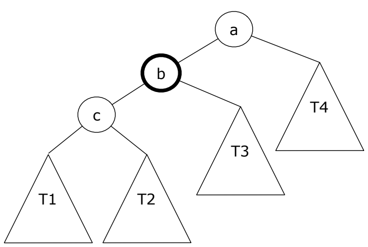
Left Rotation

- ***Insert*** the following sequence of nodes into an empty BST:

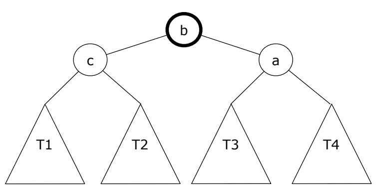
$$\langle 44, 17, 78, 32, 50, 88, 95 \rangle$$

- Is the BST now ***balanced***?
- ***Insert*** 100 into the BST.
- Is the BST still ***balanced***?
- Perform a ***left rotation*** on the appropriate node.
- Is the BST again ***balanced***?

Trinode Restructuring: Single, Right Rotation



After a ***right rotation*** on the middle node *b*:



BST property maintained? $\langle T_1,\ a,\ T_2,\ b,\ T_3,\ c,\ T_4 \rangle$

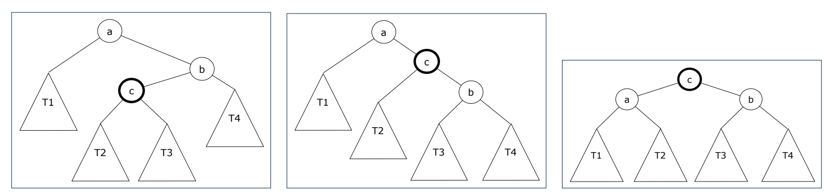
Right Rotation

- ***Insert*** the following sequence of nodes into an empty BST:

$$\langle 44, 17, 78, 32, 50, 88, 48 \rangle$$

- Is the BST now ***balanced***?
- ***Insert*** 46 into the BST.
- Is the BST still ***balanced***?
- Perform a ***right rotation*** on the appropriate node.
- Is the BST again ***balanced***?

Trinode Restructuring: Double, R-L Rotations



Perform a ***Right Rotation*** on Node *c* | Perform a ***Left Rotation*** on Node *c* | After Right-Left Rotations

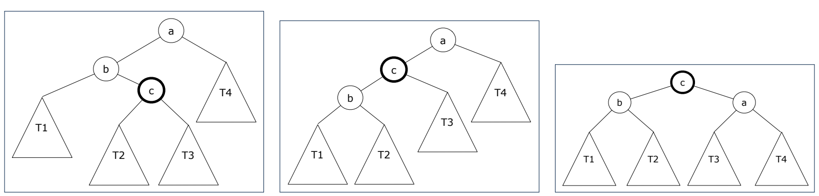
BST property maintained? $\langle T_1,\ a,\ T_2,\ c,\ T_3,\ b,\ T_4 \rangle$

R-L Rotations

- ***Insert*** the following sequence of nodes into an empty BST:

  $\langle 44, 17, 78, 32, 50, 88, 82, 95 \rangle$
- Is the BST now ***balanced***?
- ***Insert*** 85 into the BST.
- Is the BST still ***balanced***?
- Perform the ***R-L rotations*** on the appropriate node.
- Is the BST again ***balanced***?

Trinode Restructuring: Double, L-R Rotations



Perform a ***Left Rotation*** on Node *c* — Perform a ***Right Rotation*** on Node *c* — After Left-Right Rotations

BST property maintained? $\langle T_1, b, T_2, c, T_3, a, T_4 \rangle$

L-R Rotations

- ***Insert*** the following sequence of nodes into an empty BST:

  $\langle 44, 17, 78, 32, 50, 88, 48, 62 \rangle$
- Is the BST now ***balanced***?
- ***Insert*** 54 into the BST.
- Is the BST still ***balanced***?
- Perform the ***L-R rotations*** on the appropriate node.
- Is the BST again ***balanced***?

After Deletions: 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. [$O(h) = O(\log n)$ ***rotations***]

Single Trinode Restructuring Step

- ***Insert*** the following sequence of nodes into an empty BST:

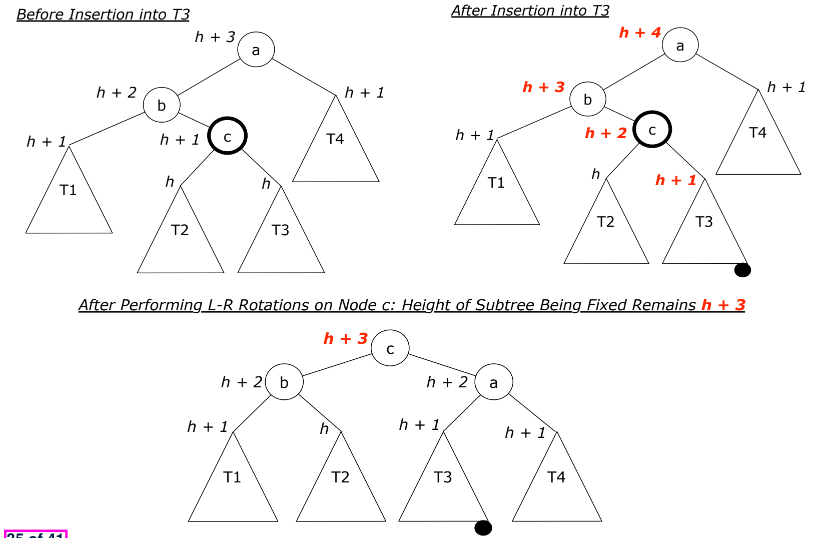
  $\langle 44, 17, 62, 32, 50, 78, 48, 54, 88 \rangle$
- Is the BST now ***balanced***?
- ***Delete*** 32 from the BST.
- Is the BST still ***balanced***?
- Perform a ***left rotation*** on the appropriate node.
- Is the BST again ***balanced***?

Multiple Trinode Restructuring Steps

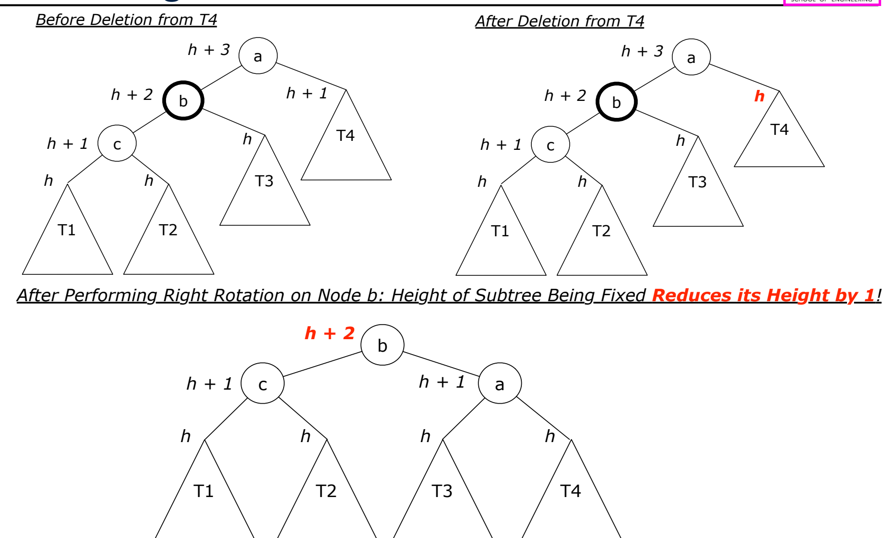
- ***Insert*** the following sequence of nodes into an empty BST:

  $\langle 50, 25, 10, 30, 5, 15, 27, 1, 75, 60, 80, 55 \rangle$
- Is the BST now ***balanced***?
- ***Delete*** 80 from the BST.
- Is the BST still ***balanced***?
- Perform a ***right rotation*** on the appropriate node.
- Is the BST now ***balanced***?
- Perform another ***right rotation*** on the appropriate node.
- Is the BST again ***balanced***?

Restoring Balance from Insertions



Restoring Balance from Deletions



Restoring Balance: Insertions vs. Deletions

- Each ***rotation*** involves only ***POs*** of setting parent-child references.
  - ⇒ ***O(1)*** running time for each tree ***rotation***
- After each ***insertion***, a ***trinode restructuring*** step can ***restore the balance*** of the subtree rooted at the first ***unbalanced*** node.
  - ⇒ ***O(1)*** rotations suffices to restore the balance of tree
- After each ***deletion***, one or more ***trinode restructuring*** steps may ***restore the balance*** of the subtree rooted at the first ***unbalanced*** node.
  - ⇒ May take ***O(log n)*** rotations to restore the balance of tree

Index (1)

Index (2)

Index (3)

Index (4)