CSCI 241
Lecture 10
Binary Search Trees: Removal, Balanced BSTs
Group Advising Session to Declare the CS Major!

3 - 4 pm on Thurs. Oct. 25 in CF 420

Course planning resources
Prepare for winter registration

Must have completed or be enrolled in CSCI 241, 247, and 301 in fall to be eligible

Major application deadline is Oct. 26 by 5 pm!

Email Mary.Hall@wwu.edu with any questions

For disability resources, contact 360-650-3083
Announcements

• Quiz 2 grades: did you get the memo?
Announcements

Lab 3 printTree:

- up is right!
- left is down!
- nothing is as it seems!

Printing a tree right-side up requires a **level-order** traversal (which isn’t recursive and is implemented using a queue!)
Goals (Monday and Today):

- Know the definition and uses of a binary search tree.
- Be prepared to implement, and know the runtime of, the following BST operations:
  - searching
  - inserting
  - deleting
- Know what a balanced BST is and why we want it.
- Be prepared to implement rotations in BSTs
/** BST: a binary tree, in which:
 * - all values in left are < value
 * - all values in right are > value
 * - left and right are BSTs */

public class BST {
    int value;
    BST parent;
    BST left;
    BST right;
}
Binary Search Tree
Searching a BST

```
search(t, 11)
```

```
t:
10 > 10
search(right, 11)
```

```
11
```

```
8
```

```
4
```

```
9
```

```
16
```

```
17
```

```
11
```

```
11
```

```
4
```

```
9
```

```
16
```

```
17
```

```
11
```

```
11
```

```
11
```

```
11
```

```
11
```

```
11
```

```
11
```

```
11
```

```
11
```
Searching a BST

\[ t: \begin{array}{c}
  10 \\
  8 & 16 \\
  4 & 9 & 11 & 17
\end{array} \]

- \[ \text{search}(t, 11) \]
  - \[ 11 > 10 \]
  - \[ \text{search(right, 11)} \]
  - \[ 11 < 16 \]
  - \[ \text{search(left, 11)} \]
Searching a BST

```
  t: 10
     /  
    /   
  8    16
 /     /   
4  9  11   17

search(t, 11)
11 > 10
search(right, 11)
11 < 16
search(left, 11)
11 == 11
found it! return.
```
Searching a BST - the nonexistent case

\[
\text{t: 10}
\]

\[
\text{search(t, 5)}
\]

\[
\text{5 < 10}
\]

\[
\text{search(left, 5)}
\]
Searching a BST - the nonexistent case

```
search(t, 5)
5 < 10
search(left, 5)
5 < 8
search(left, 5)
```
Searching a BST - the nonexistent case

search(t, 5)
5 < 10
search(left, 5)
5 < 8
search(left, 5)
5 > 4
search(right, 5)
null - not found!
Searching a BST: What’s the runtime?

```java
boolean search(BST t, int v):
    if t == null:
        return false
    if t.value == v:
        return true
    if t.value < v:
        return search(t.left)
    else:
        return search(t.right)
```

Runtime of search is $O(h)$.  Worst: $O(n)$  Best: $O(\log n)$

We want our trees to look more like this than this
Inserting into a BST
Inserting into a BST

\[
\text{t:} \quad \begin{array}{c}
\text{10} \\
\text{8} & \text{16} \\
\text{4} & \text{9} & \text{11} & \text{17}
\end{array}
\]

insert(t, 11)

11 > 10

insert(right, 11)
Inserting into a BST

$t$: 10

4 8 9 11 16 17

insert(t, 11)
11 > 10
insert(right, 11)
11 < 16
insert(left, 11)
Inserting into a BST

\[
\begin{align*}
\text{t:} & \quad 10 \\
8 & \quad 9 \quad 11 \quad 16 \\
4 & \quad 17
\end{align*}
\]

\[
\begin{align*}
\text{insert(t, 11)} \\
11 & > 10 \\
\text{insert(right, 11)} \\
11 & < 16 \\
\text{insert(left, 11)} \\
11 & == 11 \\
\text{found it! no duplicates, allowed; nothing to do.} \\
\text{return.}
\end{align*}
\]
Inserting into a BST - the nonexistent case

$t: \quad 10$

$\begin{align*}
\text{insert}(t, 5) \\
5 < 10 \\
\text{insert}(\text{left}, 5)
\end{align*}$
Inserting into a BST - the nonexistent case

\[
\begin{align*}
t &: 10 \\
8 &\quad 16 \\
4 &\quad 9 &\quad 11 &\quad 17
\end{align*}
\]

- \text{insert}(t, 5)
  - \text{5 < 10}
  - \text{insert(left, 5)}
  - \text{5 < 8}
  - \text{insert(left, 5)}
Inserting into a BST - the nonexistent case

```
insert(t, 5)

5 < 10
insert(left, 5)

5 < 8
insert(left, 5)

5 > 4
insert(right, 5)
null - not found. insert it here!
```
Warm-up

Write a method to find the smallest value in a BST:

1. Spec
   /**< min value in n’s subtree.  
   * pre: n is not null */
   *public* int minimum(Node n) {
   
2. Base case

Warm-up

Write a method to find the smallest value in a BST:

1. Spec
/** min value in n’s subtree.
 * pre: n is not null */
public int minimum(Node n) {
    if (n.left == null) 2. Base case
        return n.value;
}

3. Recursive definition:
Smallest(n) is the smallest value in the left subtree, or n.value if no left subtree exists.

4. Implement using recursive call
Warm-up

Write a method to find the smallest value in a BST:

1. Spec
   /** min value in n’s subtree.  
    * pre: n is not null */
   public int minimum(Node n) {
      if (n.left == null) 2. Base case
         return n.value;
      return minimum(n.left); 4. Implement using recursive call
   }

3. Recursive definition:
   Smallest(n) is the smallest value in the left subtree, or n.value if no left subtree exists.
Write a method to find the smallest value in a BST:

/** min value in n’s subtree.  
 * pre: n is not null */
public int minimum(Node n) {
    if (n.left == null)
        return n.value;
    return minimum(n.left);
}
Deleting a node from a BST

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children
Deleting a node from a BST: Case 1

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n is a leaf)
   replace parent’s child with null
Deleting a node from a BST: Case 2

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has exactly one child)
replace parent’s child with n’s child
replace n’s child’s parent with n’s parent
Deleting a node from a BST: Case 2

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

If (n has exactly one child)
   replace parent’s child with n’s child
   replace n’s child’s parent to n’s parent
Deleting a node from a BST: Case 2

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has exactly one child)
   replace parent’s child with n’s child
   replace n’s child’s parent to n’s parent
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
let k = min node in right subtree
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)  
let k = \text{min node in right subtree}  
replace n’s value with k’s value
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
   let k = min node in right subtree
   replace n’s value with k’s value

Can we do that?
• k is n’s successor (next in an in-order traversal)
• Everything else in n’s right subtree is bigger than it
• Everything in n’s left subtree is smaller than it
• k’s value can safely replace n’s…but now we have a duplicate.
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
let k = min node in right subtree
replace n’s value with k’s value
remove k from n’s right subtree
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
let k = min node in right subtree
replace n’s value with k’s value
remove k from n’s right subtree
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
   let k = min node in right subtree
   replace n’s value with k’s value
   remove k from n’s right subtree (recursively!)
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
let \( k = \text{min node in right subtree} \)
replace n’s value with k’s value
remove k from n’s right subtree

this has to be either Case 1 or Case 2!
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
   let k = min node in right subtree
   replace n’s value with k’s value
   remove k from n’s right subtree

this has to be either Case 1 or Case 2!

Why?
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
   let k = min node in right subtree
   replace n’s value with k’s value
   remove k from n’s right subtree
   this has to be either Case 1 or Case 2!

Why? Rewind to before we removed it:
Deleting a node from a BST: Case 3

Three possible cases:
1. n has no children (is a leaf)
2. n has one child
3. n has two children

if (n has two children)
   let k = min node in right subtree
   replace n’s value with k’s value
   remove k from n’s right subtree

Why? Rewind to before we removed it:
• k is the smallest node in n’s right subtree.
• if it had a left child, that child would have to be smaller!
Details

- Need to update root pointer if root is removed.
- Often can’t assume n.parent isn’t null - n may be root
- To update parent’s child pointer, you need to know which (L or R) child pointer to update.
- The approach presented differs from that in CLRS and some other resources.
30 second kitten break
Searching a BST:
What’s the runtime?

```java
boolean search(BST t, int v):
    if t == null:
        return false
    if t.value == v:
        return true
    if t.value < v:
        return search(t.left)
    else:
        return search(t.right)
```

Runtime of search is $O(h)$.  
We want our trees to look more like this than this

Worst: $O(n)$  
Best: $O(\log n)$
Same values, different trees
Same values, different trees

Balance Factor(n) = height(n.right) - height(left)
Can we improve balance?

Balance Factor(n) = height(n.right) - height(left)
Can we improve balance?

Balance Factor(n) = height(n.right) - height(left)
Can we improve balance?

Balance Factor\((n)\) = height\((n.right)\) - height\((left)\)
Can we improve balance?

Balance Factor(n) = height(n.right) - height(left)
Can we improve balance?

**Balance Factor(n)** = height(n.right) - height(left)
Can we improve balance?

Balance Factor(n) = height(n.right) - height(left)
Tree Rotations
modify the structure without violating the BST property.

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree
Tree Rotations

modify the structure without violating the BST property.

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

Details: need to update child, parent, and (possibly) root pointers.
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

**Details:** need to update child, parent, and (possibly) root pointers.
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. **Transfer β**: x’s right subtree becomes y’s old left subtree (β)
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

x.R gets y.L
y.L.p gets x
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

x.R gets y.L
y.L.p gets x

(only rearranged the picture)
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. **Transfer the parent:** y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

$x.R$ gets $y.L$
$y.L.p$ gets x

$y.p$ gets $x.p$
$p.[L/R]$ gets y
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. **Transfer the parent:** y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

$\ x \ R \ gets \ y \ L$
$\ y \ L \ . p \ gets \ x$

$\ y . p \ gets \ x . p$
$\ p . [L/R] \ gets \ y$

(what if $\rho$ is null / x was root?)
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer \( \beta \): x’s right subtree becomes y’s old left subtree (\( \beta \))
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

- x.R gets y.L
- y.L.p gets x
- y.p gets x.p
- p.[L/R] gets y

(only rearranged the picture)
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. Transfer the parent: y’s parent becomes x’s old parent
3. **Transfer x itself**: x becomes y’s left subtree

$x.R$ gets $y.L$
$y.L.p$ gets x
$y.p$ gets $x.p$
p.[L/R] gets y

$y.L$ gets x
$x.p$ gets y
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. Transfer the parent: y’s parent becomes x’s old parent
3. **Transfer x itself**: x becomes y’s left subtree

- $x.R$ gets $y.L$
- $y.L.p$ gets $x$
- $y.p$ gets $x.p$
- $p.[L/R]$ gets $y$

- $y.L$ gets $x$
- $x.p$ gets $y$
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer $\beta$: x’s right subtree becomes y’s old left subtree ($\beta$)
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

$$x.R \text{ gets } y.L$$
$$y.L.p \text{ gets } x$$
$$y.p \text{ gets } x.p$$
$$p.[L/R] \text{ gets } y$$

(only rearranged the picture)
Tree Rotations

Steps in left rotation (move y up to its parent’s position):
1. Transfer β: x’s right subtree becomes y’s old left subtree (β)
2. Transfer the parent: y’s parent becomes x’s old parent
3. Transfer x itself: x becomes y’s left subtree

Overall Transformation:

x.R gets y.L
y.L.p gets x
y.p gets x.p
p.[L/R] gets y
y.L gets x
x.p gets y
Pseudocode from CLRS

```
LEFT-ROTATE(T, x)
1. xfer β
2. x.right = y.left
3. if y.left ≠ T.nil
4. y.left.p = x
5. y.p = x.p
6. if x.p == T.nil
7. T.root = y
8. elseif x == x.p.left
9. x.p.left = y
10. else x.p.right = y
11. y.left = x
12. x.p = y
```

1. xfer β
2. xfer parent
3. xfer x

// set y
// turn y’s left subtree into x’s right subtree
// link x’s parent to y
// put x on y’s left