Lecture 23
Binary Search Trees II

- Lab 7 – Preview
- Binary Search Trees
  - Operations (Part 2)
  - structure Package
  - Applications
  - Efficiency
- Self-Balancing Search Trees
Lab 7 — Preview
The lab is focused on the two-player game hexapawn (or hex-a-pawn) created by Martin Gardner.

- Create a “game tree” of all of the possible moves. The nodes are board configurations and whose turn it is. The root is the initial configuration on white’s turn. The children of a node are the configurations that can be reached by the current player making one move. The game ends at each leaf node, where the current payer cannot make a move.

- To “learn” from a loss, a (computer) player can prune its own version of the game tree, so that it never makes the same last move (which it knows leads to a loss).
Binary Search Trees
continued ...
Exercise: find and insert

Last class, we discussed how to implement find (contains) and insert (add). Now try your best to reproduce these methods.

1. Write recursive pseudocode for find: determine if a target value \( t \) is in a binary search tree.
2. [Time permitting] Write recursive pseudocode for insert: add value \( t \) to a binary search tree.

In this part, you can assume that value \( t \) is not already in the binary search tree.

Write your answer for 3 minutes.
Then trade notes with a neighbor for 2 minutes.

Notes:
- If node is a node in the tree, then you can access its value and children as follows: node.value, node.left, node.right
- What are your base cases?
- Remember to make a new node when inserting.
**Pseudocode for find and insert.**

- **find**
  - if node is null then
    return false
  - if node.value == target then
    return true
  - if target < node.value then
    return find(node.left, target)
  - else
    return find(node.right, target)

- **insert**
  - if node == null then
    node = new node(value)
    return
  - if value < node.value then
    if node.left is null then
      node.left = new node(value)
    else
      insert(node.left, value)
  - else
    if node.right == null then
      node.right = new node(value)
    else
      insert(node.right, value)

Remember that there is no single “correct” pseudocode style.

- Some may prefer to use `true` instead of `yes` (as in previous slides).
- Some may prefer to use `null` instead of `empty` (as in previous slides).
- Some may prefer to use the argument name `node` instead of `root` (as in previous slides).
- Some may prefer to use `.left()` or `left()` instead of `.left().
Operations

(Part 2)
Question: Delete

How can we delete a value from a binary search tree?

- Are there any easy cases?
- Can you convert from a hard case to an easy case?

Recall our operations on binary heaps.

Think about this for 2 minutes.
Think about the quality of your approach.

- What is its run-time? Let \( n \) be the number of values currently in the structure.
- Would it cause subsequent operations (\texttt{find}, \texttt{insert}, or \texttt{delete}) to take longer?

We'll aim for self-contained pseudocode that is similar to the textbook's approach.
Binary Search Tree: Delete (Easy Cases)

Let’s focus on two easy cases:

1. Deleting a leaf. In this case, we just remove it.
2. Deleting a node with one child. In this case, we can move the subtree rooted at the child into the deleted node’s position.

After these deletions, the subtree conditions will still hold at each node.

Notes:

- We could identify more easy cases. We’ll focus on these because they help us solve the remaining cases.
- Are these really different cases? We can use a combined easy case: A node has at most one child.
Let’s focus on two easy cases:

1. **Deleting a leaf.**
   In this case, we just remove it.

2. **Deleting a node with one child.**
   In this case, we can move the subtree rooted at the child into the deleted node’s position.

After these deletions, the subtree conditions will still hold at each node.

Notes:

- We could identify more easy cases. We’ll focus on these because they help us solve the remaining cases.
- Are these really different cases? We can use a combined easy case: A node has at most one child.
In the remaining case, we must delete a node with two children. Let the node's value be \( v \).

Due to its left child, \( v \) isn't the smallest value. The \textit{next smallest value} \( s \) (i.e., largest value \( s \) with \( s < v \)) is the \textit{rightmost descendant} of its left child (i.e., go left once, then right as much as possible).

What if we swap \( v \) and \( s \)?
- The subtree condition will only be violated by \( v \) and \( s \). Why?
- The value \( v \) is now either (a) in a leaf, or (b) in a node that only has one child (since it cannot have a right child).

Therefore, after the swap, it is an easy case to delete the node containing \( v \).
In the remaining case, we must delete a node with two children. Let the node’s value be v.

Due to its left child, v isn’t the smallest value. The next smallest value s (i.e., largest value s with s < v) is the rightmost descendant of its left child (i.e., go left once, then right as much as possible).

What if we swap v and s?
- The subtree condition will only be violated by v and s. Why?
- The value v is now either (a) in a leaf, or (b) in a node that only has one child (since it cannot have a right child).

Therefore, after the swap, it is an easy case to delete the node containing v.
In the remaining case, we must delete a node with two children. Let the node’s value be \( v \).

Due to its left child, \( v \) isn’t the smallest value. The next smallest value \( s \) (i.e., largest value \( s \) with \( s < v \)) is the rightmost descendant of its left child (i.e., go left once, then right as much as possible).

What if we swap \( v \) and \( s \)?

- The subtree condition will only be violated by \( v \) and \( s \). Why?
- The value \( v \) is now either (a) in a leaf, or (b) in a node that only has one child (since it cannot have a right child).

Therefore, after the swap, it is an easy case to delete the node containing \( v \).
In the remaining case, we must delete a node with two children. Let the node’s value be $v$.

Due to its left child, $v$ isn’t the smallest value. The *next smallest value* $s$ (i.e., largest value $s$ with $s < v$) is the rightmost descendant of its left child (i.e., go left once, then right as much as possible).

What if we swap $v$ and $s$?

- The subtree condition will only be violated by $v$ and $s$. Why?
- The value $v$ is now either (a) in a leaf, or (b) in a node that only has one child (since it cannot have a right child).

Therefore, after the swap, it is an easy case to delete the node containing $v$.

Delete also works with the *next largest value* (i.e., smallest $\ell$ with $\ell > v$) in the leftmost descendant of its right child.
In the remaining case, we must delete a node with two children. Let the node's value be v.

Due to its left child, v isn’t the smallest value. The next smallest value s (i.e., largest value s with s < v) is the rightmost descendant of its left child (i.e., go left once, then right as much as possible).

What if we swap v and s?
- The subtree condition will only be violated by v and s. Why?
- The value v is now either (a) in a leaf, or (b) in a node that only has one child (since it cannot have a right child).

Therefore, after the swap, it is an easy case to delete the node containing v.

Delete also works with the next largest value (i.e., smallest ℓ with ℓ > v) in the leftmost descendant of its right child.
In the remaining case, we must delete a node with two children. Let the node's value be $v$.

Due to its left child, $v$ isn't the smallest value. The next smallest value $s$ (i.e., largest value $s$ with $s < v$) is the rightmost descendant of its left child (i.e., go left once, then right as much as possible).

What if we swap $v$ and $s$?
- The subtree condition will only be violated by $v$ and $s$. Why?
- The value $v$ is now either (a) in a leaf, or (b) in a node that only has one child (since it cannot have a right child).

Therefore, after the swap, it is an easy case to delete the node containing $v$.

Delete also works with the next largest value (i.e., smallest $\ell$ with $\ell > v$) in the leftmost descendant of its right child.
Binary Search Tree: delete

This pseudocode implements the `delete` operation recursively and returns if it was successful.

```plaintext
function delete(node, target)
    // Base case: Target not in tree.
    if node == null then return false

    // The target is not in this node.
    if node.value < target then
        return delete(node.left, target)
    else if node.value > target then
        return delete(node.right, target)

    // Determine if node is a child.
    isLeft = false
    isRight = false
    parent = node.parent
    if parent ≠ null then
        isLeft  = (parent.left  == node)
        isRight = (parent.right == node)
    return

    // Combined easy case: At most one child.
    if node.left == null then
        if isLeft then parent.left = node.right
        if isRight then parent.right = node.right
        return true
    else if node.right == null then
        if isLeft then parent.left = node.left
        if isRight then parent.right = node.left
        return true

    // Hard case: Find next smallest s.
    s = node.left
    while s.right == null
        s = s.right
    // Then swap values and finish recursively.
    node.value = s.value
    s.value = target
    return delete(s, target)
```

This approach uses `.parent` references. The deletion may require the data structure’s root reference to be updated (not shown).
structure Package
The textbook’s approach for `delete` (remove) has some similarities and differences:

- Figure 14.2 (a)–(b) are the combined easy case: A node has at most one child. Figure 14.2 (c) is an additional easy case that is needed using this approach.
- `remove` calls protected methods including `locate` (which returns a node to remove) and `removeTop` (which returns the modified subtree); `remove` fixes the parent references.
Data Structures & Advanced Programming

Williams College

CSCI 136

remove and its helper functions in the structure5 package.

```java
/**
 * @pre root and value are non-null
 * @post returned: 1 - existing tree node with the desired value, or
 * 2 - the node to which value should be added
 */
protected BinaryTree<E> locate(BinaryTree<E> root, E value) {
    E rootValue = root.value();
    BinaryTree<E> child;

    // found at root: done
    if (rootValue.equals(value)) return root;
    // look left if less-than, right if greater-than
    if (ordering.compare(rootValue, value) < 0) {
        child = root.right();
    } else {
        child = root.left();
    }

    // no child there: not in tree, return this node,
    // else keep searching
    if (child.isEmpty()) {
        return root;
    } else {
        return locate(child, value);
    }
}
```
protected BinaryTree<E> predecessor(BinaryTree<E> root) {
    Assert.pre(!root.isEmpty(), "No predecessor to middle value.");
    Assert.pre(!root.left().isEmpty(), "Root has left child.");
    BinaryTree<E> result = root.left();
    while (!result.right().isEmpty()) {
        result = result.right();
    }
    return result;
}

protected BinaryTree<E> successor(BinaryTree<E> root) {
    Assert.pre(!root.isEmpty(), "Tree is non-null.");
    Assert.pre(!root.right().isEmpty(), "Root has right child.");
    BinaryTree<E> result = root.right();
    while (!result.left().isEmpty()) {
        result = result.left();
    }
    return result;
}

remove and its helper functions in the structure5 package.
remove and its helper functions in the structure5 package.
Applications
Applications

Binary search trees have many applications:

- Tree sorting. Insert all of the values, then perform an in-order traversal. Expected run-time is $O(n \log n)$-time, but this is not true in the worst-case.

- Symbol table. The keys are ordered and each key has an associated value. Find, insert, and remove in expected $O(\log n)$-time.