Lecture 23

Binary Search Trees II

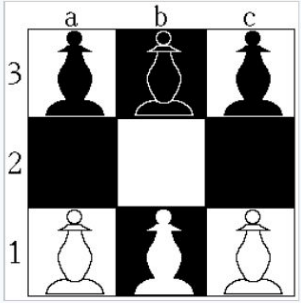
- Lab 7 – Preview
- Binary Search Trees
    - Operations (Part 2)
    - `structure` Package
    - Applications
    - Efficiency
- Self-Balancing Search Trees

Lab 7 – Preview

Hexapawn

From Wikipedia, the free encyclopedia

Hexapawn is a deterministic two-player game invented by Martin Gardner. It is played on a rectangular board of variable size, for example on a 3×3 board or on a chessboard. On a board of size *n*×*m*, each player begins with *m* pawns, one for each square in the row closest to them. The goal of each player is to advance one of their pawns to the opposite end of the board or to prevent the other player from moving.



A 3×3 hexapawn board

Hexapawn on the 3×3 board is a solved game; with perfect play, white will always lose in 3 moves: (1.b2 axb2 2.cxb2 c2 3.a2 c1#). Indeed, Gardner specifically constructed it as a game with a small game tree, in order to demonstrate how it could be played by a heuristic AI implemented by a mechanical computer based on Donald Michie's Matchbox Educable Noughts and Crosses Engine.

A variant of this game is **octopawn**, which is played on a 4×4 board with 4 pawns on each side. In octopawn, if both players play well, the second player to move will always lose.

2. Available for your use are several Java classes:

`HexBoard`. This class describes the state of a board. The default constructor builds the 3×3 starting `HexBoard`. You can ask a board (with its `moves(color)` method) to return a `Vector` of the `HexMoves` that are possible for a particular color (`HexBoard.WHITE` or `HexBoard.BLACK`) from the position. The `win(color)` method allows you to ask a `HexBoard` if the current position is a win for a particular color. A static utility method, `HexBoard.opponent(color)`, takes a color and returns the opposite color.

The `main` method of this class allows a human to play Hex-a-Pawn against a computer that moves randomly. It is a demonstration of how `HexBoards` are manipulated and printed.

`HexMove`. This class describes the movement of a pawn. The result of `HexBoard.moves` is a `Vector` of `HexMove`. Given a `HexBoard` and a `HexMove` one can construct the resulting `HexBoard` using a `HexBoard` constructor. These two classes—`HexBoard` and `HexMove`—are vital in exploring the state-space of the Hex-a-Pawn game.

`GameTree`. This is one of the classes *you* will construct. The `GameTree` nodes will form a large tree of `HexBoard` states, related by player moves. At the root is the starting position, ready for `WHITE` to move. The next level of the tree describes `HexBoard` positions that are the result of a `WHITE` move, ready for `BLACK` to move. The 3×3 game leads to a tree with 252 nodes. We expect that players will traverse a single tree recursively and, if they wish, *prune* the tree to learn from losses.

`Player`. The `Player` interface describes the methods that must be provided by agents that play the game. Every `Player` must have a name and color, accessible through `getName()` and `getColor()`, respectively. In addition, they must support a `play(node,opponent)` method takes a `GameTree` node and an opposing `Player`. This method plays the game by traversing one level of the `GameTree`—if it can—and checking for a win. If the player's move does not lead to a win, it passes control of the game to its opponent. The result of the `play` method is the `Player` who ultimately wins the game.

Read these class files carefully. Please do not modify the classes `HexBoard`, `HexMove`, or `Player`.

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.

```
function find(node, target)
    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)
```

```
function insert(node, value)
    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)
```

Pseudocode for `find` and `insert`.

- `insert`'s first case is only for empty trees.
  In this case, the tree's root node is being created.

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.

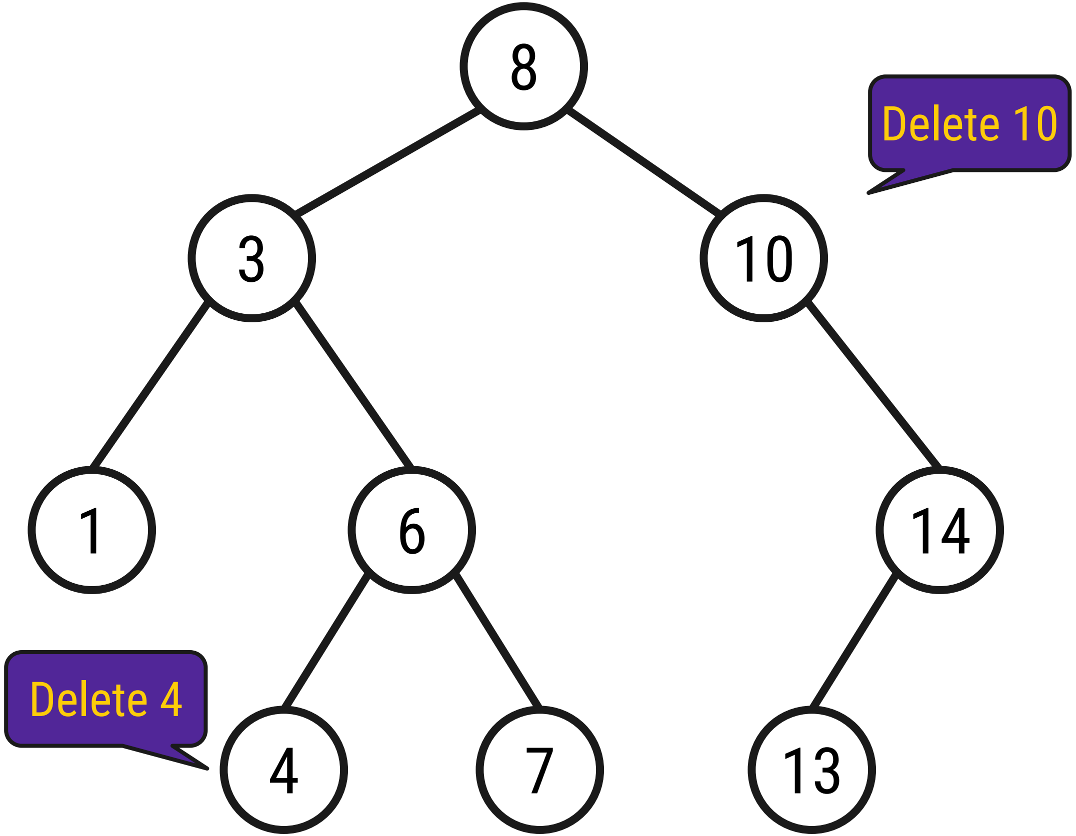
Think about this for 2 minutes.
Then discuss it with your neighbor for 2 minutes.

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 (`find`, `insert`, or `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:

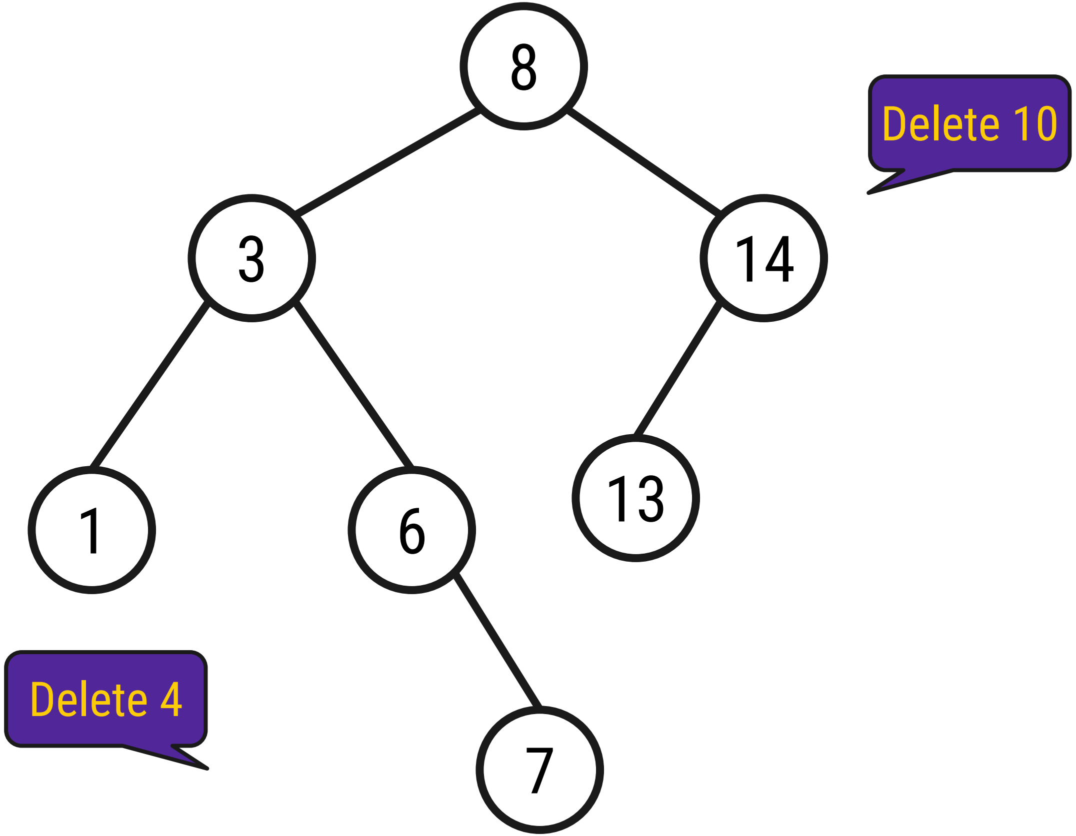
1. Deleting a leaf.
   In this case, we just remove it.
2. Deleting a node with one child.
   In this case, we can move the subtree rooted at the child into the deleted node's position.

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.

Binary Search Tree: Delete (Easy Cases)



Let's focus on two easy cases:

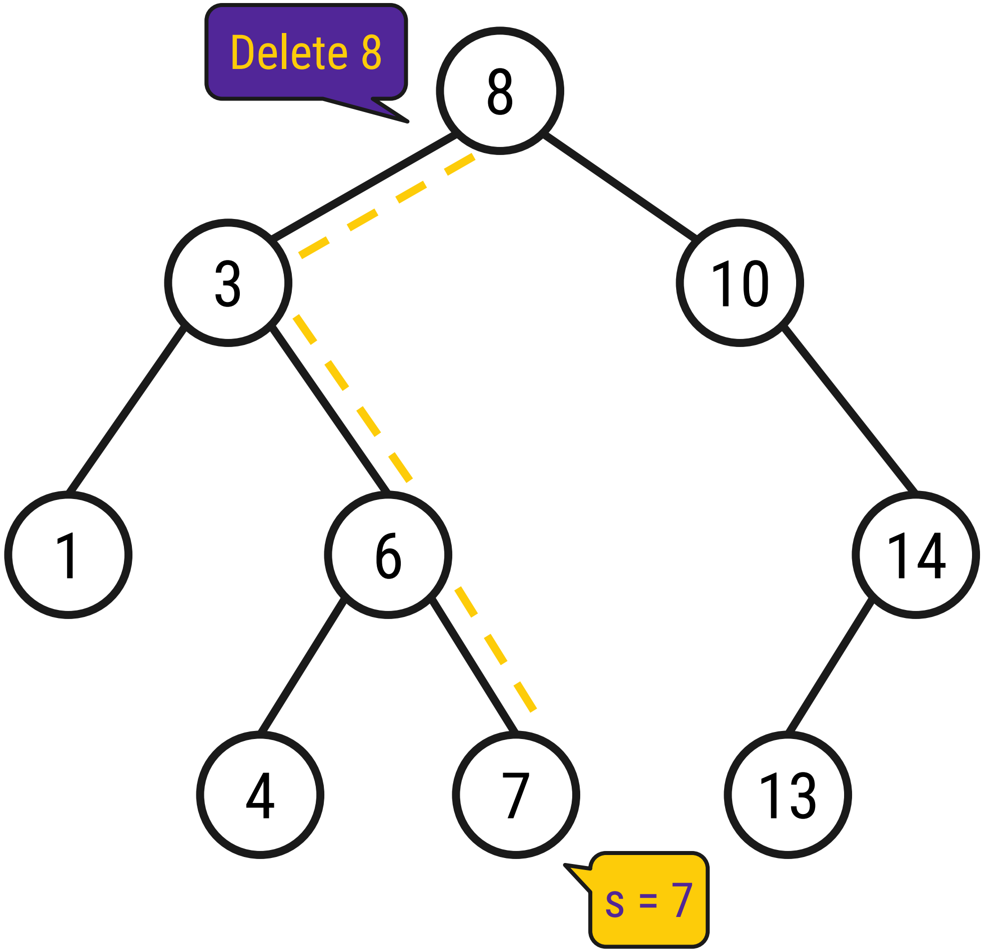
1. Deleting a leaf.
   In this case, we just remove it.
2. Deleting a node with one child.
   In this case, we can move the subtree rooted at the child into the deleted node's position.

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.

Binary Search Tree: Delete (Hard Case)



In the remaining case, we must delete a node with two children. Let the node's value be v.

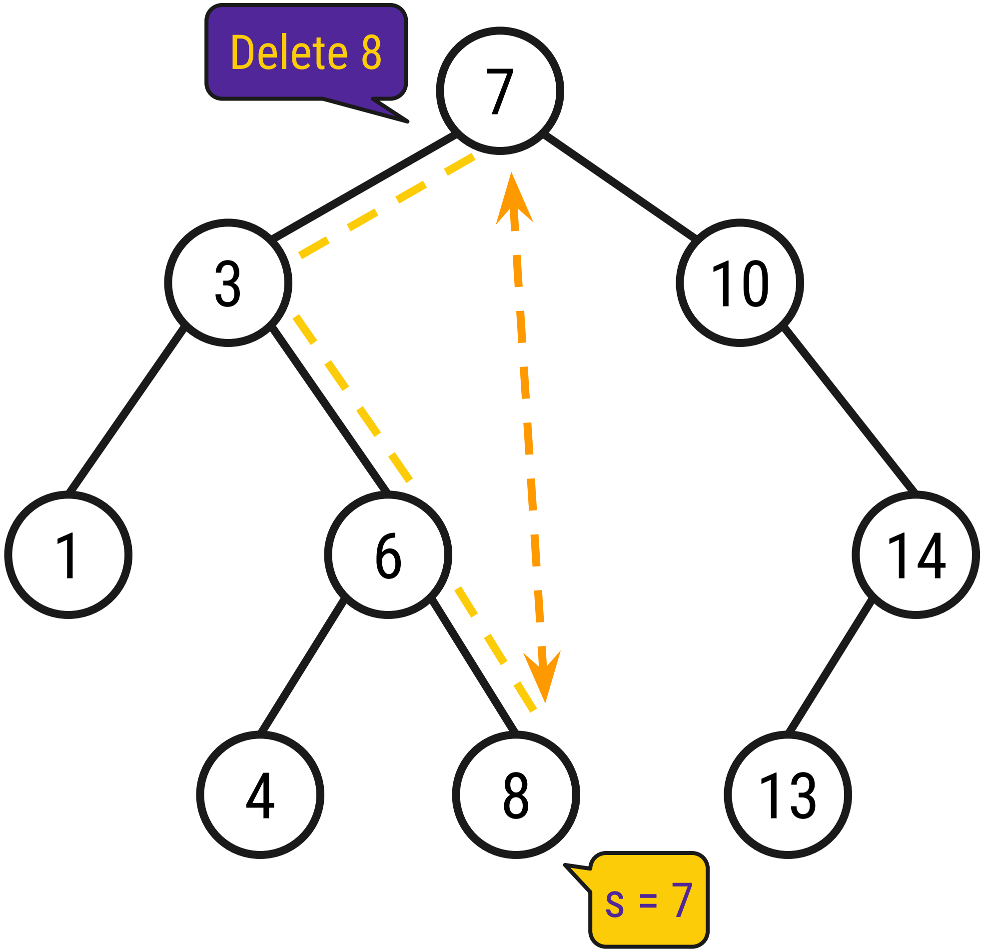
Due to its left child, v isn't the smallest value. The *next smallest value* s (i.e., largest value s with s < v) is the *rightmost descendant* of its left child (i.e., go left ocne, then right as much as possible).

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.

Binary Search Tree: Delete (Hard Case)



In the remaining case, we must delete a node with two children. Let the node's value be v.

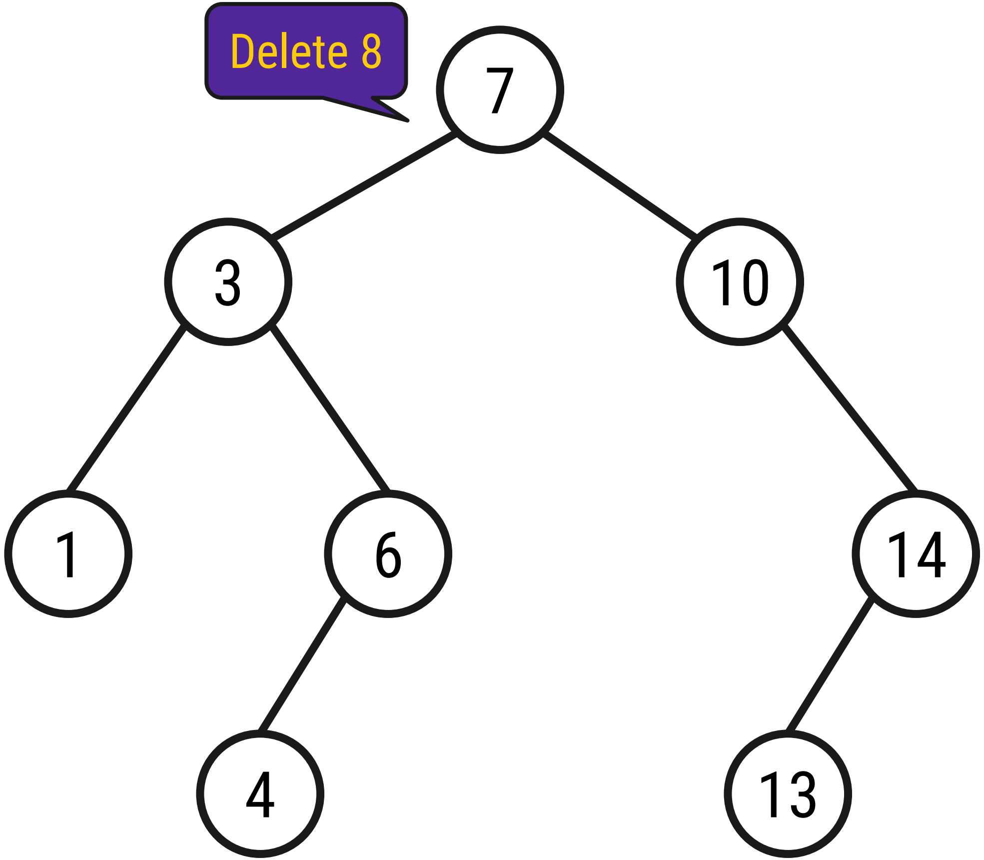
Due to its left child, v isn't the smallest value. The *next smallest value* s (i.e., largest value s with s < v) is the *rightmost descendant* of its left child (i.e., go left once, then right as much as possible).

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.

Binary Search Tree: Delete (Hard Case)



In the remaining case, we must delete a node with two children. Let the node's value be v.

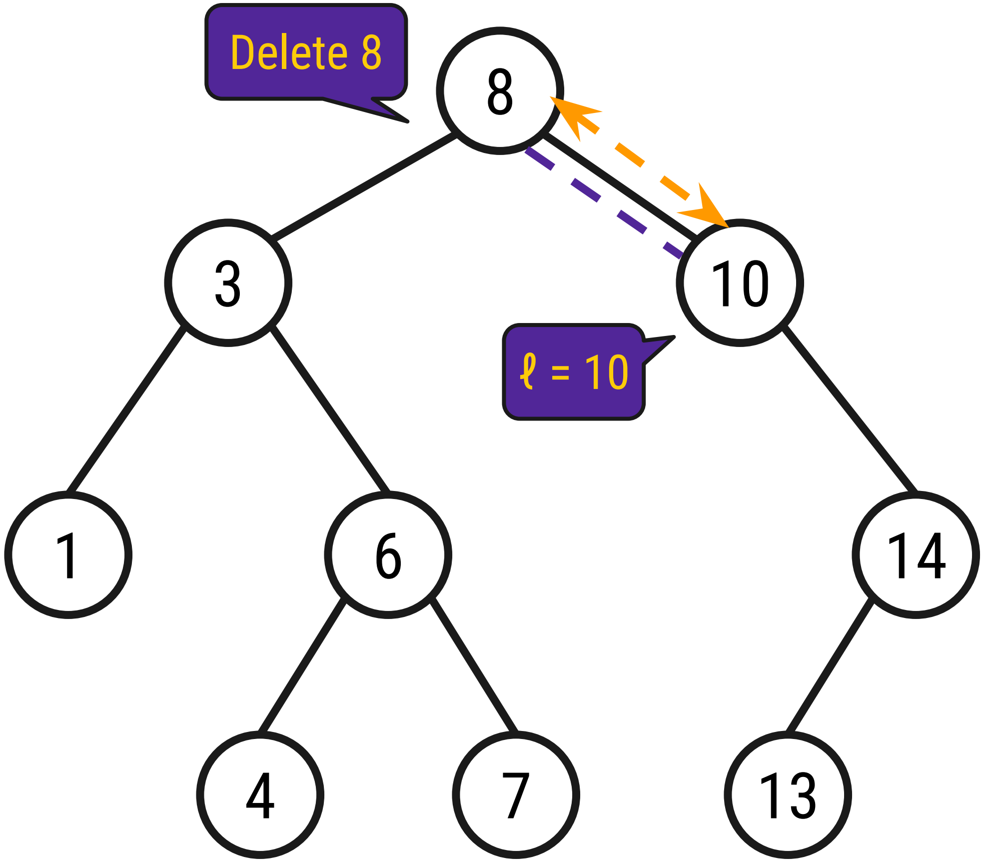
Due to its left child, v isn't the smallest value. The *next smallest value* s (i.e., largest value s with s < v) is the *rightmost descendant* of its left child (i.e., go left once, then right as much as possible).

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.

Binary Search Tree: Delete (Hard Case)



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).

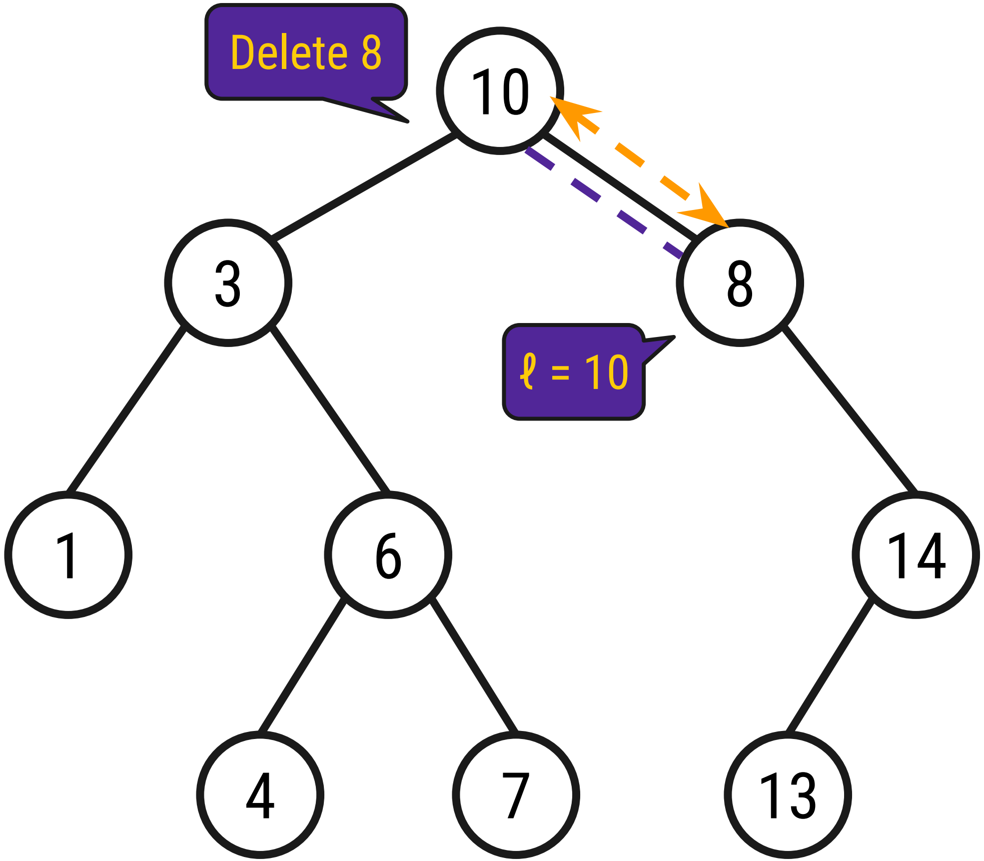
What if we swap v and s?

- 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.

Binary Search Tree: Delete (Hard Case)



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).

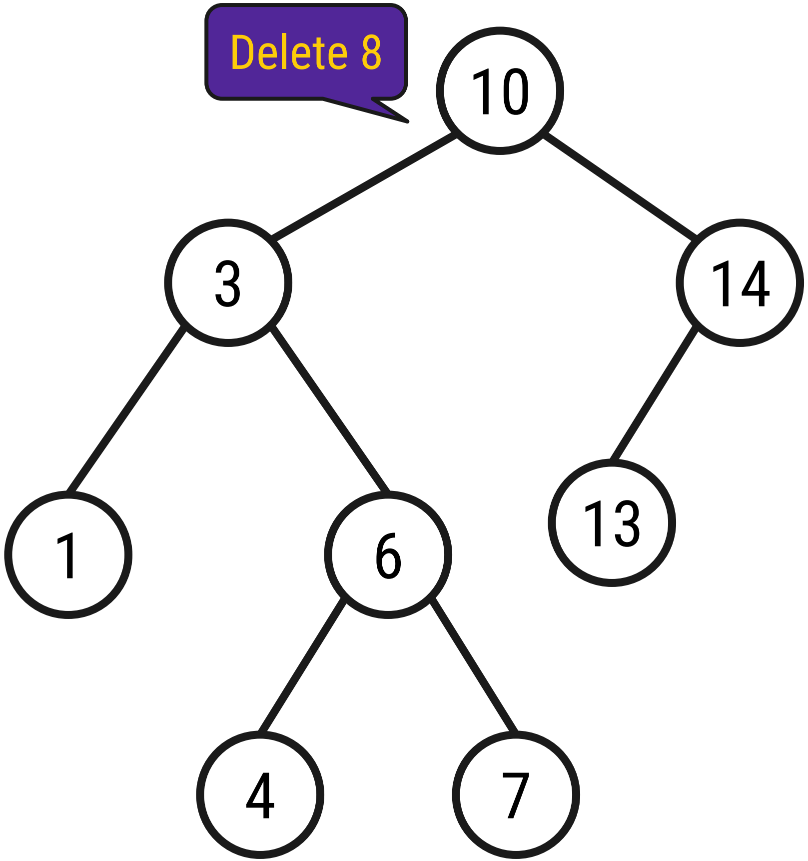
What if we swap v and s?

- 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.

Binary Search Tree: Delete (Hard Case)



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.

Binary Search Tree: `delete`

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

```
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)
```

```
// 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)
```

Alternatively, we could find the next largest ℓ.

This approach uses `.parent` references.

The deletion may require the data structure's `root` reference to be updated (not shown).

`structure` Package

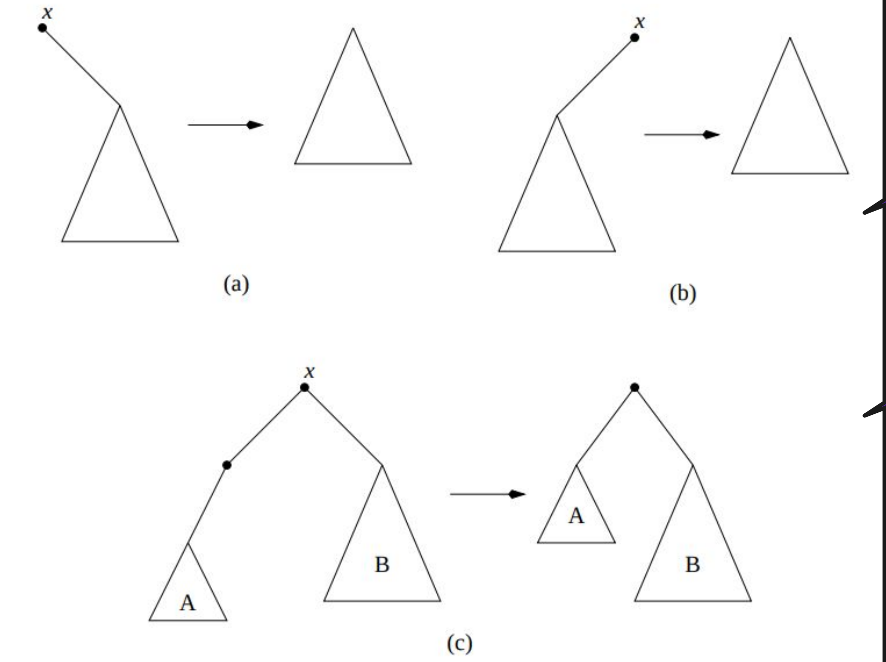


Figure 14.2 The three simple cases of removing a root value from a tree.

(a) and (b) are the combined easy case.

(c) is another easy case required by this approach

The hard case.

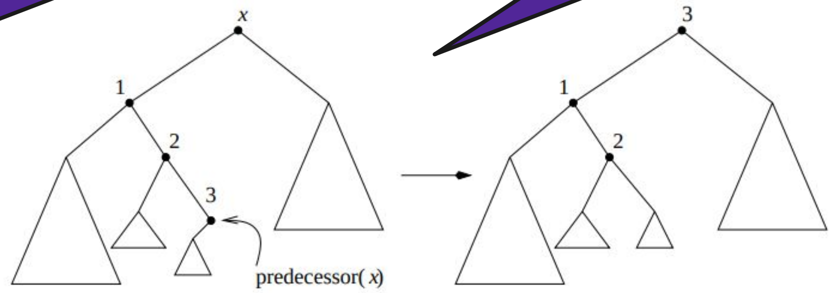


Figure 14.3 Removing the root of a tree with a rightmost left descendant.

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.

```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);
    }
}
```

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

```
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.

```java
/**
 * Remove an value "equals to" the indicated value.  Only one value
 * is removed, and no guarantee is made concerning which of duplicate
 * values are removed.  Value returned is no longer part of the
 * structure
 *
 * @post Removes one instance of val, if found
 *
 * @param val Value sought to be removed from tree
 * @return Actual value removed from tree
 */
public E remove(E value)
{
    if (isEmpty()) return null;

    if (value.equals(root.value())) // delete root value
    {
        BinaryTree<E> newroot = removeTop(root);
        count--;
        E result = root.value();
        root = newroot;
        return result;
    }
    else
    {
        BinaryTree<E> location = locate(root,value);

        if (value.equals(location.value())) {
            count--;
            BinaryTree<E> parent = location.parent();
            if (parent.right() == location) {
                parent.setRight(removeTop(location));
            } else {
                parent.setLeft(removeTop(location));
            }
            return location.value();
        }
    }
    return null;
}
```

```java
 * Removes the top node of the tree rooted, performs the necessary
 * rotations to reconnect the tree.
 *
 * @pre topNode contains the value we want to remove
 * @post We return an binary tree rooted with the predecessor of topnode.
 * @param topNode Contains the value we want to remove
 * @return The root of a new binary tree containing all of topNodes
 * descendents and rooted at topNode's predecessor
 */
protected BinaryTree<E> removeTop(BinaryTree<E> topNode)
{
    // remove topmost BinaryTree from a binary search tree
    BinaryTree<E> left  = topNode.left();
    BinaryTree<E> right = topNode.right();
    // disconnect top node
    topNode.setLeft(EMPTY);
    topNode.setRight(EMPTY);
    // Case a, no left BinaryTree
    //   easy: right subtree is new tree
    if (left.isEmpty()) { return right; }
    // Case b, no right BinaryTree
    //   easy: left subtree is new tree
    if (right.isEmpty()) { return left; }
    // Case c, left node has no right subtree
    //   easy: make right subtree of left
    BinaryTree<E> predecessor = left.right();
    if (predecessor.isEmpty()) {
        left.setRight(right);
        return left;
    }
    // General case, slide down left tree
    //   harder: successor of root becomes new root
    //           parent always points to parent of predecessor
    BinaryTree<E> parent = left;
    while (!predecessor.right().isEmpty()) {
        parent = predecessor;
        predecessor = predecessor.right();
    }
    // Assert: predecessor is predecessor of root
    parent.setRight(predecessor.left());
    predecessor.setLeft(left);
    predecessor.setRight(right);
    return predecessor;
}
```

`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.