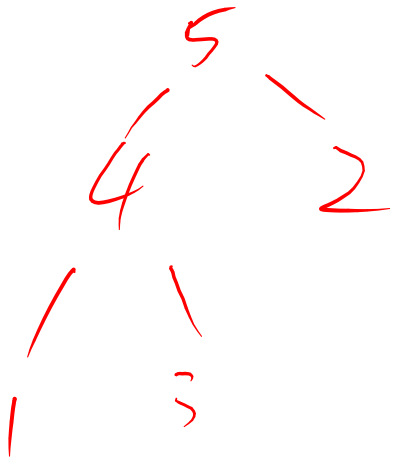
[TAP:LCSGR] Binary Search Tree and Heap

- Is it possible for a binary tree to be a binary search tree and a heap at the same time?

no duplicates



Today's Outline

- Binary Search Tree
  - Basics
  - Operations
  - Implementation

BST Operations

- BSTs will implement the OrderedStructure Interface
  - `add(E item)`
  - `contains(E item)`
  - `get(E item)`
  - `remove(E item)`
  - `iterator()` ← in-order traversal

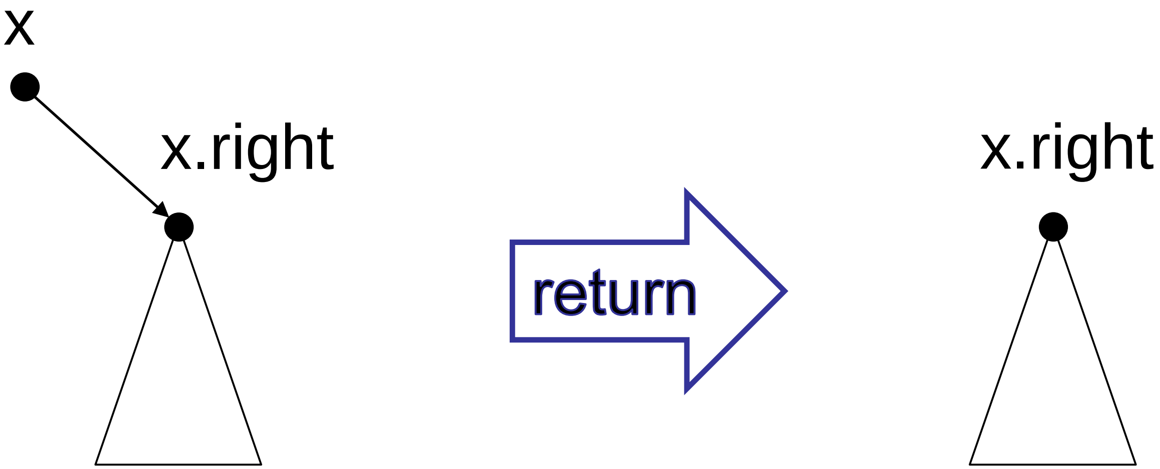
~~O(log n)~~ $\log_2 n \le h \le n$

O(h) ← height of the tree

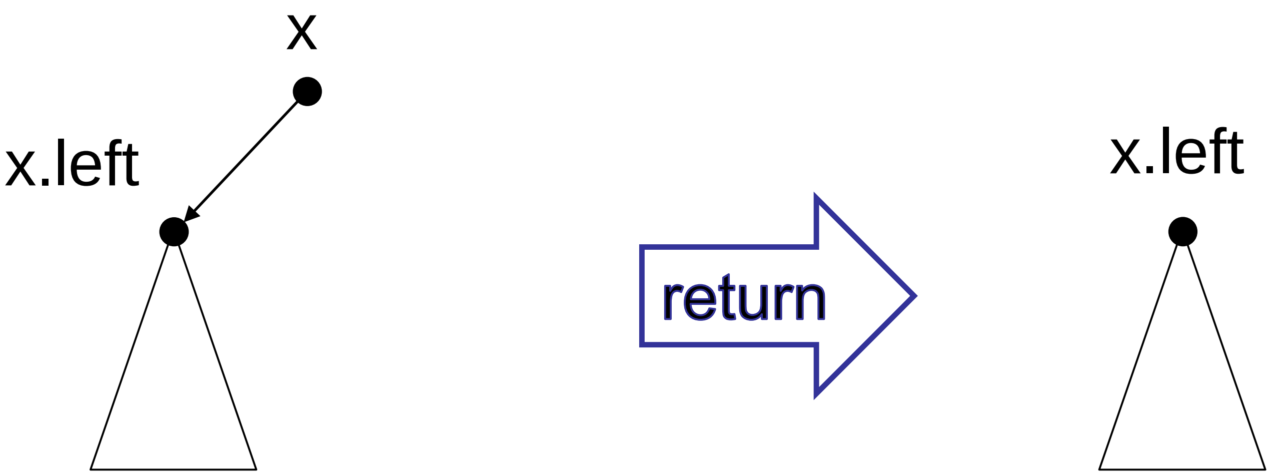
Removal

- Removing the root is a (not so) special case
  - If we can remove the root, we can remove any element in a BST in the same way
- We need to implement:
  - `public E remove(E item)`
  - `protected BT removeTop(BT top)`

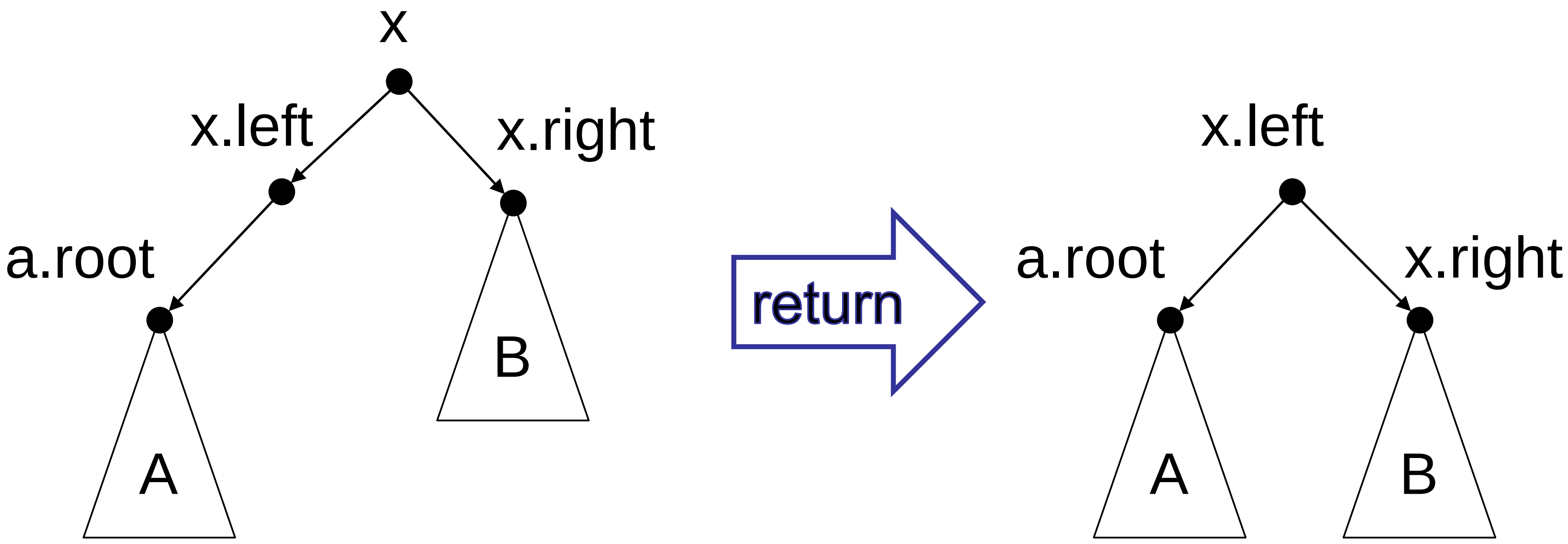
RemoveTop(topNode)
Case 1: No left subtree



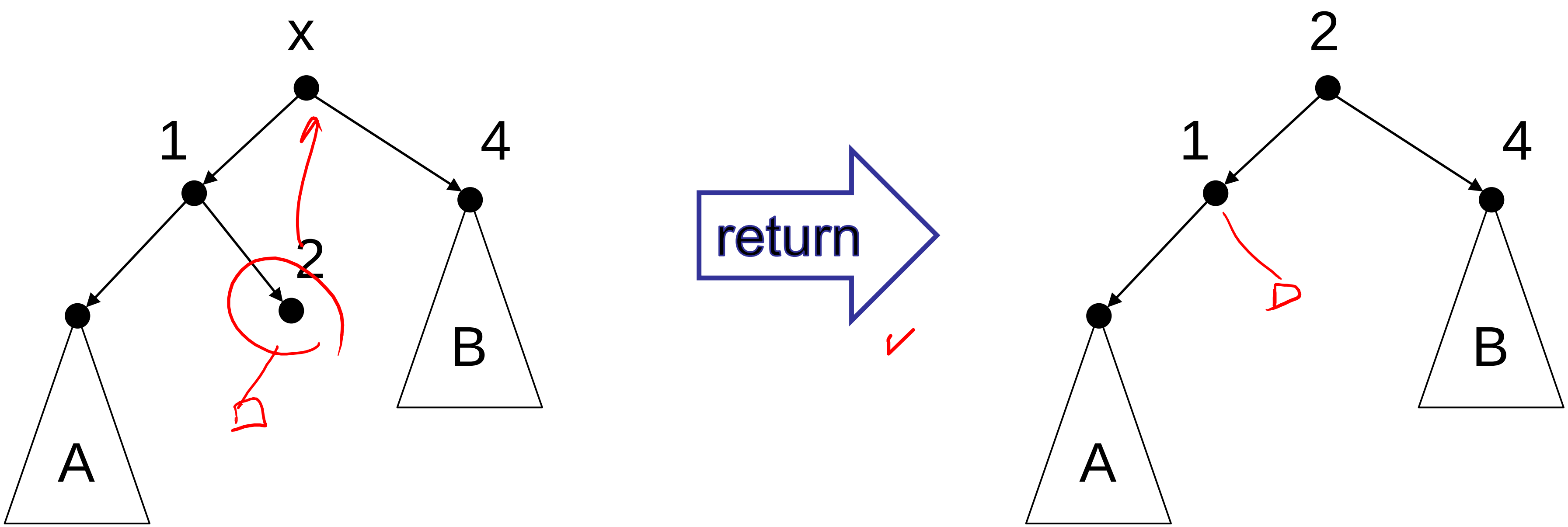
RemoveTop(topNode)
Case 2: No right subtree



RemoveTop(topNode)
Case 3: Left has no right subtree

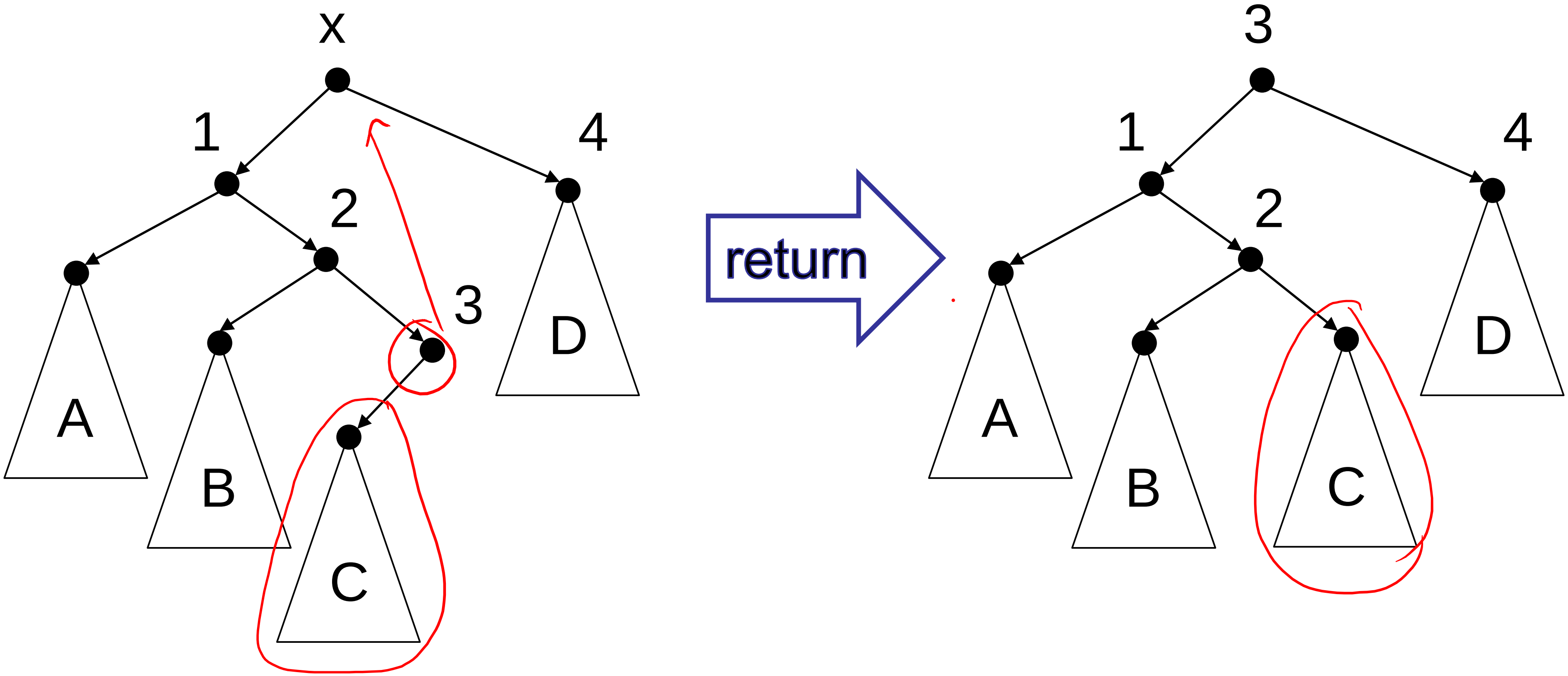


RemoveTop(topNode) Case 4: General Case

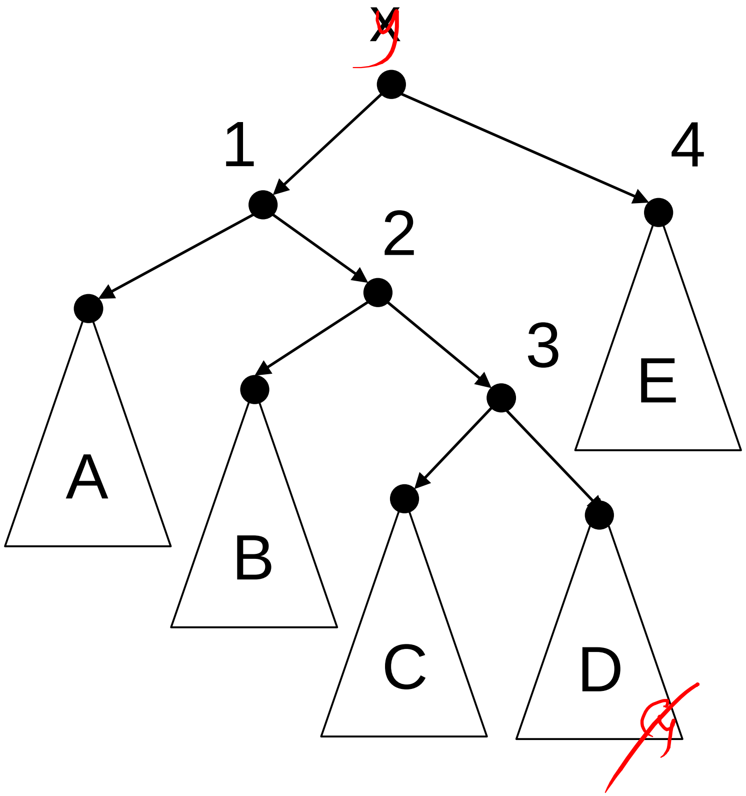


return

RemoveTop(topNode) Case 4: General Case



[Exercise] Draw the tree after removing x



Today’s Outline

- Binary Search Tree
  - Basics
  - Operations
  - Implementation

BST Implementation

- The BST holds the following items
  - `BinaryTree root`: the root of the tree
  - `int count`: the number of nodes in the BST
  - `Comparator<E> ordering`: for an alternative way to comparing nodes
- Two constructors: One takes a Comparator

BST Implementation: locate

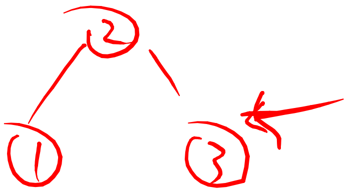
- Several methods search the tree:
  - `add`, `remove`, `contains`, …
- We factor out common code: `locate` method
- *protected* locate(BinaryTree<E> *node*, E *v*)
  - Returns a `BinaryTree<E>` *n* in the subtree whose root is *node* such that :

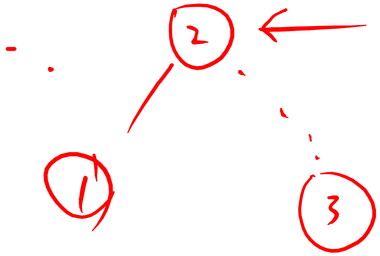
Can be distinguished by comparing v and n.val

1
      - *n* has its value equal to *v* or

2
      - *v* is not in this subtree and *n* should be *v*'s parent

3





The code : locate

```
protected BinaryTree<E> locate(BinaryTree<E> root, E value) {
      E   rootVal = root.value();
      BinaryTree<E> child;

      if ( rootVal.equals(value))
            return root;  // case ①

      if (ordering.compare(rootVal, value) < 0 )
            child = root.right();
      else
            child = root.left();

      if (child.isEmpty())
              return root;  // case ②
      else
            return locate(chid, value);

}
```

Contains

```
public boolean contains(E value){

        if (root.isEmpty())
            return false;

      return value.equals(locate(root, value).value());

}
```