

# CS3500: Object-Oriented Design

## Spring 2014

Canceled Class  
2.4.2014

# Assignments 4

- Deadline extended until Tuesday, February 4 at 11:59pm
- Assignment 5 builds on Assignment 4

# Iterators

[Lewis & Chase]

- An *iterator* is an object that provides the means to iterate over a collection.
- Provide methods that allow the user to acquire and use each element in a collection in turn.

```
//An iterator over a collection.
public interface Iterator<E> {
    // Returns true if the iteration has more
    // elements.
    public boolean hasNext ();

    // Returns the next element in the iteration.
    public E next ();

    // Removes from the underlying collection the last
    // element returned by the iterator (optional).
    public void remove ();
}
```

```

import java.util.Iterator;
import java.util.NoSuchElementException;
/**
 * IntegerIterator iterates through Integers 0-MAX
 */
public class IntegerIterator implements Iterator<Integer>{
    int n; //store the current value to return
    final int MAX = 10; //Max value to return

    /**
     * Initializes the state of the iterator to the starting
     * value of 0
     */
    public IntegerIterator(){
        n = 0;
    }
    /**
     * @return whether there is another integer in the iterator, which is determined by whether n<=MAX
     */
    public boolean hasNext(){
        return (n <= MAX);
    }
    /**
     * Returns the next Integer and updates n
     * @return the next Integer
     */
    public Integer next(){
        if(hasNext()){
            Integer result = new Integer(n);
            n = n + 1;
            return result;
        } else{
            throw new NoSuchElementException();
        }
    }
    /**
     * remove is an Unsupported Operation
     */
    public void remove(){
        throw new UnsupportedOperationException("remove");
    }
}

```

State of iterator (field/s)

Initialize state (field/s)

Does the iterator have another element?

Keys to next method:

- check that next element exists
- return the next element
- update state in preparation for following call to next

# Questions to consider when writing an iterator

- State
  - What will be the state (field/s) of the iterator?
- Constructor
  - What will the state be initialized to?
  - What will be passed to the constructor? (What parameters will the constructor take?)
- hasNext method
  - How do we know if there is another element in the iterator? (What condition can we test?)
- next method
  - Does another element exist?
  - What is the next element to return?
  - How must we update the state in order to prepare for following call to next?

# MyMap Iterator

- Hint: I would suggest considering using an ArrayList to store the MyMap keys and to use for your iterator.
- You will have to consider how to get keys of MyMap into ArrayList and how you will iterate through ArrayList

# Testing Iterator

```
f0 = MyMap.empty();
f1 = f0.include(one, alice);
f2 = f1.include(two, bob);
f5 = f2.include(one, carol);

f = f5;
m = f0;
it = f.iterator();
count = 0;
while (it.hasNext()) {
    Integer k = it.next();
    m = m.include(k, f.get(k));
    count = count + 1;
}
assertTrue("iterator [(1 Carol) (2 Bob) (1 Alice)]",
           f5.equals(m));
assertFalse("iteratorSanity [(1 Carol) (2 Bob) (1 Alice)]",
           it.hasNext());
assertTrue("Iterator count [(1 Carol) (2 Bob) (1 Alice)]",
           f.size() == count);
```



# Resources for Iterators

- The IntegerIterator example in the course directory:  
    /course/cs3500sp14/examples/2014\_01\_28
- Sestoft Section 22.7
- Liskov Chapter 6
- <http://docs.oracle.com/javase/6/docs/api/java/util/Iterator.html>

# Testing Exceptions Example

```
try {  
    // MyMap<Integer, String> f has been declared and  
    // initialized elsewhere  
    Iterator<Integer> it = f.iterator();  
    it.remove();  
    // failed test because remove didn't throw exception  
}  
catch (UnsupportedOperationException e) {  
    // passed test because remove threw  
    // UnsupportedOperationException  
}  
catch (Exception e) {  
    // failed test because remove threw Exception but  
    // didn't throw UnsupportedOperationException  
}
```

```

/**
 * A total ordering of T
 */
public interface Comparator<T>{
    /**
     * Compares its two arguments for order. Returns a negative
     * integer, zero, or a positive integer as the first
     * argument is less than, equal to, or greater than the
     * second.
     * @param o1 the first object to be compared.
     * @param o2 the second object to be compared.
     * @return a negative integer, zero, or a positive integer
     * as the first argument is less than, equal to, or greater
     * than the second
     */
    public int compare(T o1, T o2);

    /**
     * Indicates whether some other object is "equal to" this
     * comparator.
     * @param obj the reference object with which to compare.
     * @return whether specified object is also a comparator and it
     * imposes the same ordering as this comparator
     */
    public boolean equals(Object obj);
}

```

```

/**
 * A comparator for Integer values.
 */
private static class UsualIntegerComparator implements Comparator<Integer> {
    /**
     * Compares its two arguments for order.
     * @param m first Integer to compare
     * @param n second Integer to compare
     * @return Returns a negative integer, zero, or a positive integer as m is
     *         less than, equal to, or greater than n
     */
    public int compare(Integer m, Integer n) {
        return m.compareTo(n);
    }
    /**
     * Is this <code>Comparator</code> same as the given object
     * @param o the given object
     * @return true if the given object is an instance of this class
     */
    public boolean equals(Object o) {
        return (o instanceof UsualIntegerComparator);
    }
    /**
     * There should be only one instance of this class = all are equal
     * @return the hash code same for all instances
     */
    public int hashCode() {
        return (this.toString().hashCode());
    }
    /**
     * @return name of class
     */
    public String toString() {
        return "UsualIntegerComparator";
    }
}

```

# Resources for Comparators

- <http://docs.oracle.com/javase/6/docs/api/java/util/Comparator.html>
- Sestoft Section 22.8
- UsualIntegerComparator from Slides

# Generic Syntax

**Generic class syntax (Sestoft, p.78):**

```
class-modifiers class C<T1, ..., Tn> class-base-clause  
{ class-body }
```

**Generic method syntax (Sestoft, p.86):**

```
method-modifiers <T1, ..., Tn> returntype m(formal-list)  
{ method-body }
```

**Example:**

```
public abstract class MyMap<K, V> implements Iterable<K>{  
    ...  
    public static <K, V> MyMap<K, V> empty() {  
        return new Empty<K, V>();  
    }  
    ...  
}
```

# Resources for Generic Type Syntax

- <http://docs.oracle.com/javase/tutorial/java/generics/index.html>
- Sestoft Section 21

# Assignment 5

- Due Tuesday, February 11 at 11:59pm
- Implement  $\text{MyMap}\langle K, V \rangle$  ADT more efficiently using binary search tree



# Tree Basics

## [Lewis & Chase]

- Tree: “a non-linear structure in which elements are organized into a hierarchy”
  - Tree contains nodes (elements) and edges (connect nodes)
- Root: single node at top level of tree
- “The nodes at lower levels of the tree are the children of nodes at the previous level. Nodes that have the same parent are called siblings.”
- Leaf: “node that does not have any children”
- Internal node: “node that is not the root and has at least one child”

# Binary Trees

# Labeled Binary Tree (LBT)


- an empty tree
- a node with three components:
  - a label
  - a left subtree, which is a labeled binary tree
  - a right subtree, which is a labeled binary tree

# Binary Search Trees

# Binary Search Tree (BST)

- $t$  is empty
- $t$  is a node
  - a label
  - the left subtree of  $t$  is a BST,
  - the right subtree of  $t$  is a BST,
  - every label within the left subtree of  $t$  is less than the label of  $t$ ,
  - every label within the right subtree of  $t$  is greater than the label of  $t$

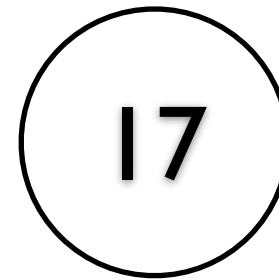
# Insertion into BST

- Insert new element as a leaf
  - Starting at the root of BST
  - Is the BST empty?
    - If so, add element here.
    - Else, compare node element to new element
      - If new element  $<$  node element, move to left subtree.
      - Else if new element  $>$  node element, move to right subtree.
- 

# Example: BST of Integers with Usual Ordering

## Insert

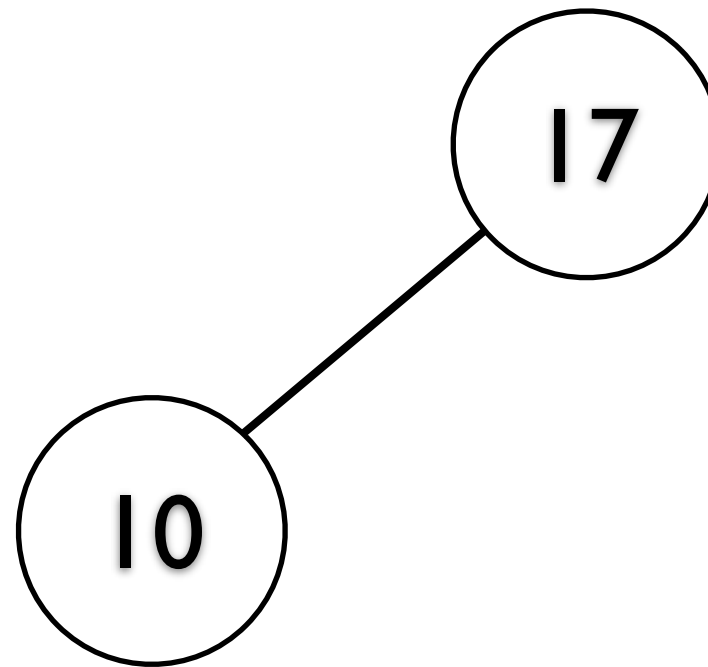
- 17
- 10
- 37
- 23
- 4
- 40
- 13



# Example: BST of Integers with Usual Ordering

## Insert

- 17
- **10**
- 37
- 23
- 4
- 40
- 13

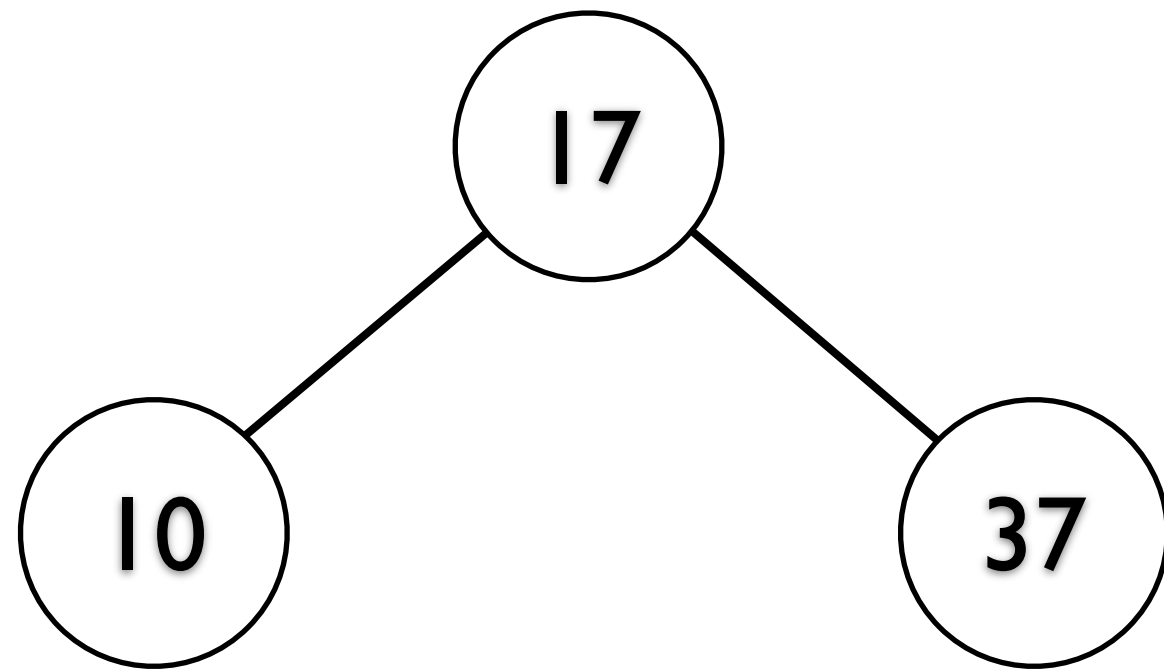




# Example: BST of Integers with Usual Ordering

## Insert

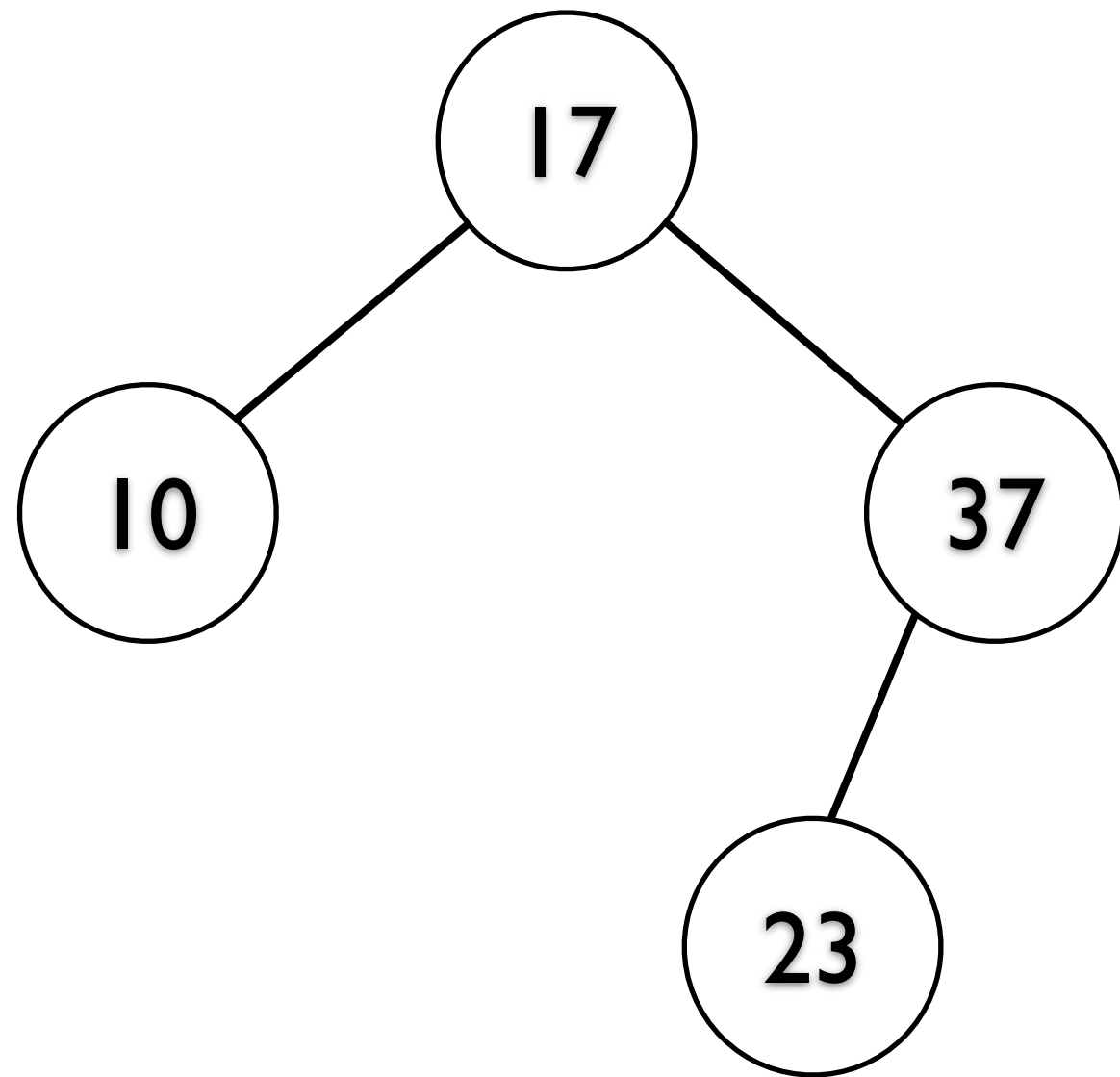
- 17
- 10
- **37**
- 23
- 4
- 40
- 13



# Example: BST of Integers with Usual Ordering

## Insert

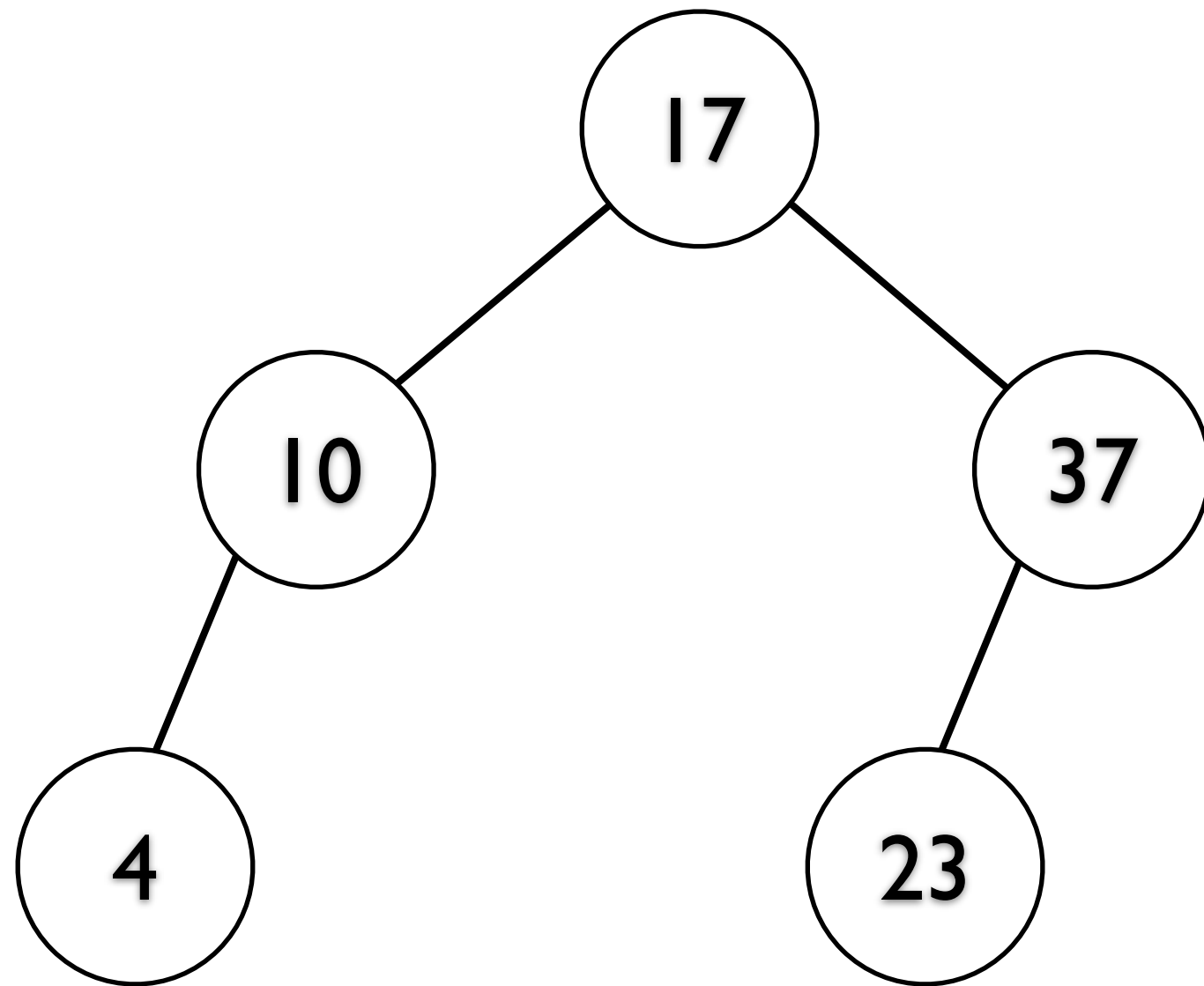
- 17
- 10
- 37
- **23**
- 4
- 40
- 13



# Example: BST of Integers with Usual Ordering

## Insert

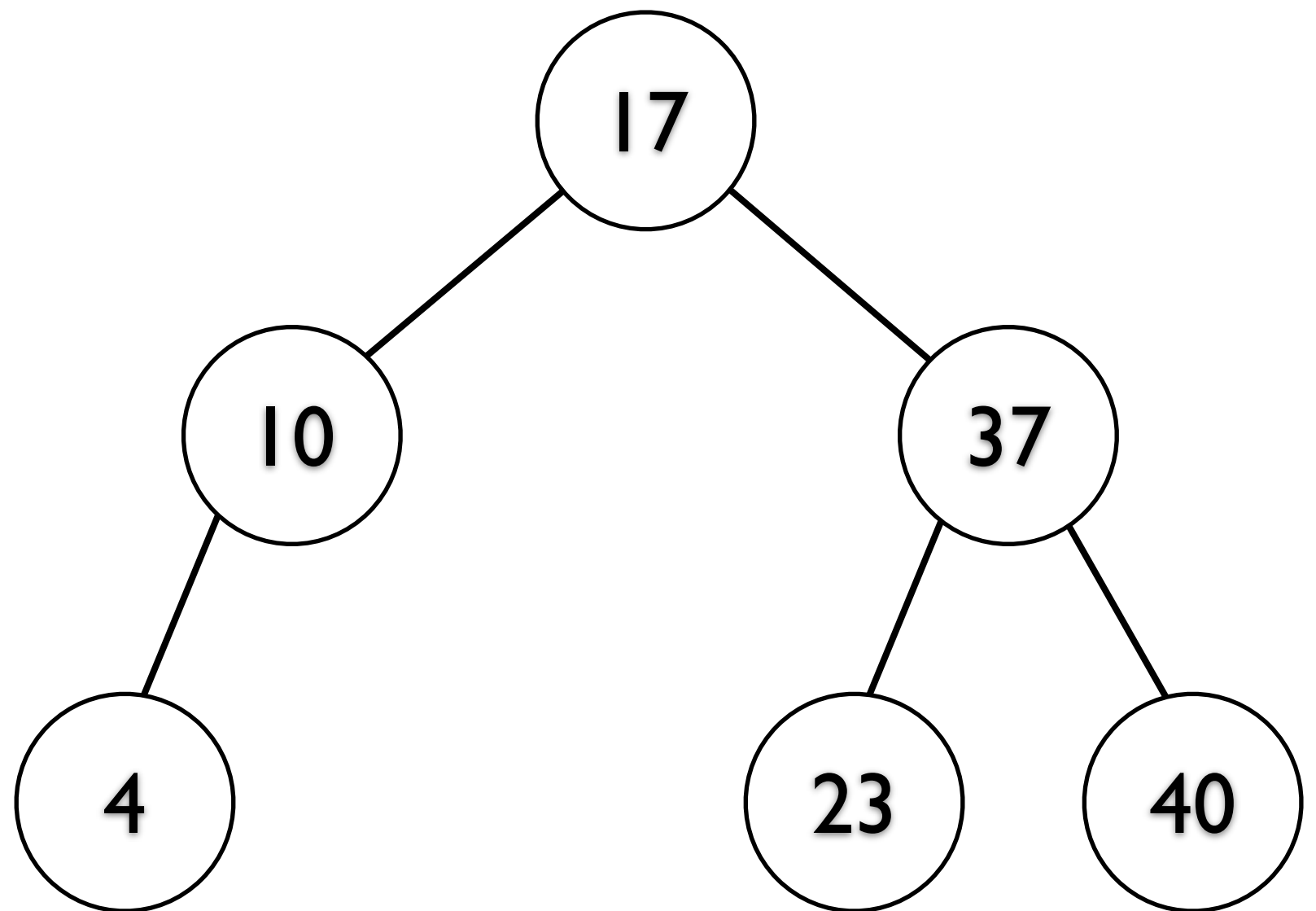
- 17
- 10
- 37
- 23
- 4
- 40
- 13



# Example: BST of Integers with Usual Ordering

## Insert

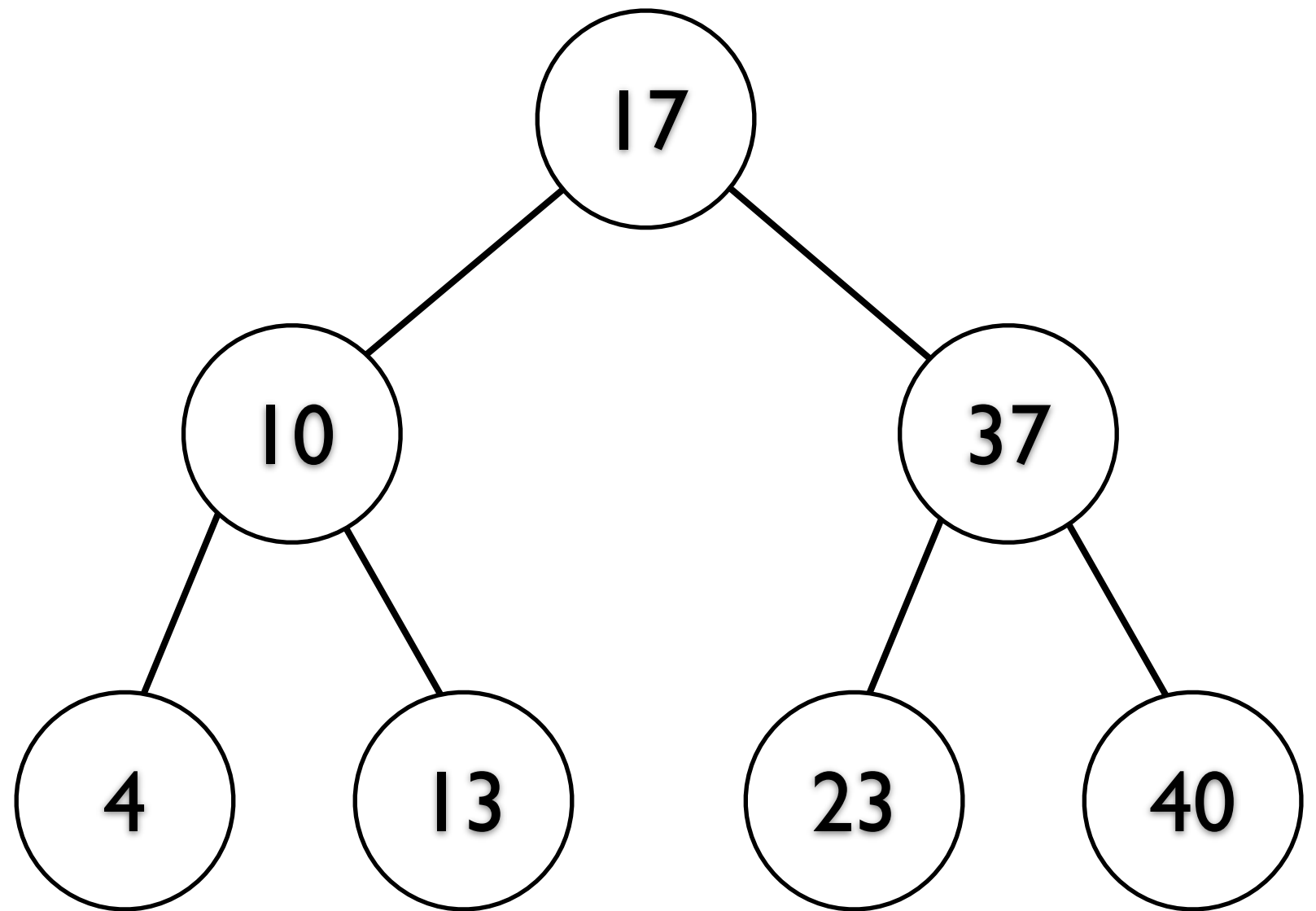
- 17
- 10
- 37
- 23
- 4
- **40**
- 13



# Example: BST of Integers with Usual Ordering

## Insert

- 17
- 10
- 37
- 23
- 4
- 40
- 13



# Algebraic Specs

- How would you specify an immutable binary search tree using algebraic specs?
- What operations would you need?