Chapter 21

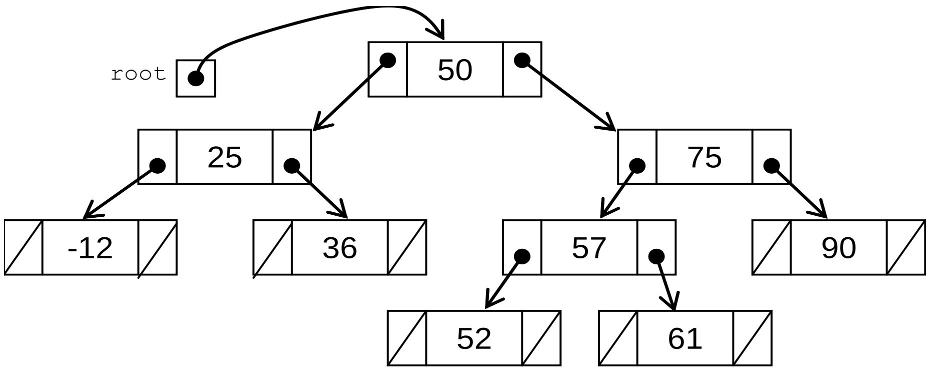
Binary Search Trees

Binary Search Trees

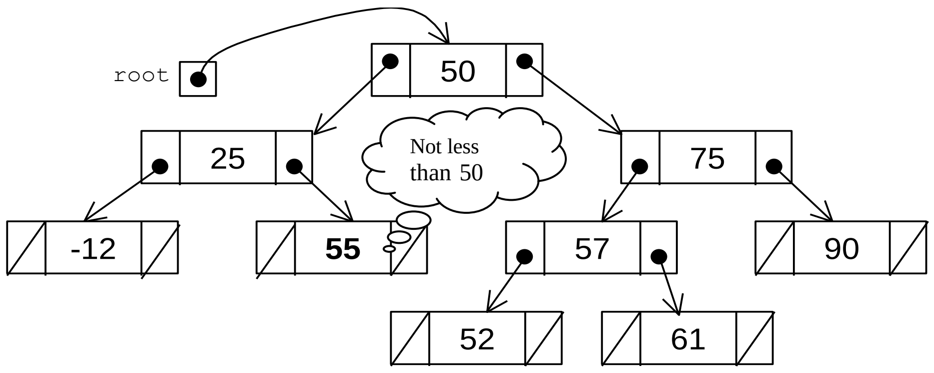
A Binary Search Tree is a binary tree with an ordering property that allows O(log n) retrieval, insertion, and removal of individual elements. Defined recursively, a binary search tree is

1. an empty tree, or
2. consists of a node called the root, and two children, left and right, each of which are themselves binary search trees. Each node contains data at the root that is greater than all values in the left subtree while also being less than all values in the right subtree. No two nodes compare equally. This is called the binary search tree ordering property.

The following two trees represent a binary search tree and a binary tree respectively. Both have the same structure — every node has two children (which may be empty trees shown with `/`) . Only the first tree has the binary search ordering property.



The second does not have the BST ordering property. The node containing 55 is found in the left subtree of 50 instead of the right subtree.



The `BinarySearchTree` class will add the following methods:

`insert`	Add an element to the binary search tree while maintaining the ordering property.
`find`	Return a reference to the element that "equals" the argument according to compareTo
`remove`	Remove the that "equals" while maintaining the ordering property (left as an exercise)

Java generics will make this collection class more type safe. It would be tempting to use this familiar class heading.

```
  public class BinarySearchTree<E>
```

However, to maintain the ordering property, BinarySearchTree algorithms frequently need to compare two elements to see if one element is greater than, less than, or equal to another element. These comparisons can be made for types that have the compareTo method.

Java generics have a way to ensure that a type has the compareTo method. Rather than accepting any type with <E>, programmers can ensure that the type used to construct an instance does indeed implement the Comparable interface (or any interface that extends the Comparable interface) with this syntax:

```
public class BinarySearchTree <E extends Comparable<E>> {
```

This class heading uses a bounded parameter to restrict the types allowed in a `BinarySearchTree` to `Comparable`s only. This heading will also avoid the need to cast to `Comparable`. Using `<E extends Comparable <E>>` will also avoid cast exceptions errors at runtime. Instead, an attempt to compile a construction with a `NonComparable` — assuming `NonComparable` is a class that does not implement `Comparable` — results in a more preferable compile time error.

```
  BinarySearchTree<String> strings = new BinarySearchTree<String>();
  BinarySearchTree<Integer> integers = new BinarySearchTree<Integer>();
  BinarySearchTree<NonComparable> no = new BinarySearchTree<NonComparable>();
                              ⇑
    Bound mismatch: The type NonComparable is not a valid substitute for the bounded
    parameter <E extends Comparable<E>>
```

So far, most elements have been String or Integer objects. This makes explanations shorter. For example, it is easier to write stringTree.insert("A"); than accountTree.insert(new BankAccount("Zeke Nathanielson", 150.00)); (and it is also easier for authors to fit short strings and integers in the boxes that represent elements of a tree).

However, collections of only strings or integers are not all that common outside of textbooks. You will more likely need to store real-world data. Then the find method seems more appropriate. For example, you could have a binary search tree that stores BankAccount objects assuming BankAccount implements Comparable. Then the return value from find could be used to update the object in the collection, by sending withdraw, deposit, or getBalance messages.

```
accountCollection.insert(new BankAccount("Mark", 50.00));
accountCollection.insert(new BankAccount("Jeff", 100.00));
accountCollection.insert(new BankAccount("Nathan", 150.00));

// Need to create a dummy object that will "equals" the account in the BST
BankAccount toBeMatched = new BankAccount("Jeff", -999);
BankAccount currentReference = accountCollection.find(toBeMatched);
assertNotNull(currentReference);
assertEquals("Jeff", currentReference.getID());

accountCollection.printSideways();
currentReference.deposit(123.45);

System.out.println("After a deposit for Jeff");
accountCollection.printSideways();
```

Output (Notice that the element with ID Jeff changes):

```
    Nathan $150.00
Mark $50.00
    Jeff $100.00
After a deposit for Jeff
    Nathan $150.00
Mark $50.00
    Jeff $223.45
```

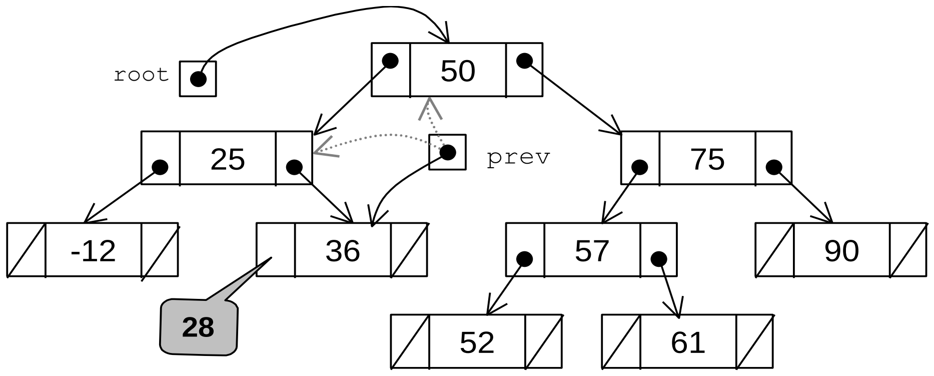
Linked Implementation of a BST

The linked implementation of a binary search tree presented here uses a private inner class `TreeNode` that stores the type `E` specified as the type parameter. This means the nodes can only store the type of element passed as the type argument at construction (which must implement `Comparable` or an interface that extends interface `Comparable`).

```
// This simple class stores a collection of strings in a binary tree.
// There is no add or insert method.Instead a tree will be "hard coded" to
// demonstrate algorithms such as tree traversals, makeMirror, and height.
public class BinarySearchTree<E extends Comparable<E>> {

  private class TreeNode {

    private E data;
    private TreeNode left;
    private TreeNode right;

    TreeNode(E theData) {
      data = theData;
      left = null;
      right = null;
    }
  }

  private TreeNode root;

  public BinarySearchTree() {
    root = null;
  }

  // The insert and find methods will be added here
}
```

boolean insert(E)

A new node will always be inserted as a leaf. The insert algorithm begins at the root and proceeds as if it were searching for that element. For example, to insert a new `Integer` object with the value of 28 into the following binary search tree, 28 will first be compared to 50. Since 28 is less than the root value of 50, the search proceeds down the left subtree. Since 28 is greater than 25, the search proceeds to the right subtree. Since 28 is less than 36, the search attempts to proceed left, but stops. The tree to the left is empty. At this point, the new element should be added to the tree as the left child of the node with 36.



The search to find the insertion point ends under either of these two conditions:

1. A node matching the new value is found.
2. There is no further place to search. The node can then be added as a leaf.

In the first case, the insert method could simply quit without adding the new node (recall that binary search trees do not allow duplicate elements). If the search stopped due to finding an empty tree, then a new `TreeNode` with the integer 28 gets constructed and the reference to this new node replaces one of the empty trees (the `null` value) in the leaf last visited. In this case, the reference to the new node with 28 replaces the empty tree to the left of 36.

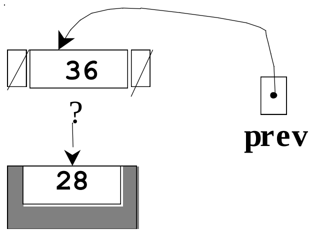
One problem to be resolved is that a reference variable (named `curr` in the code below) used to find the insertion point eventually becomes `null`. The algorithm must determine where it should store the reference to the new node. It will be in either the left link or the right link of the node last visited. In other words, after the insertion spot is found in the loop, the code must determine if the new element is greater than or less than its soon to be parent.

Therefore, two reference variables will be used to search through the binary search tree. The `TreeNode` reference named `prev` will keep track of the previous node visited. (Note: There are other ways to implement this).

The following method is one solution to insertion. It utilizes the Binary Search Tree ordering property. The algorithm checks that the element about to be inserted is either less than or greater than each node visited. This allows the appropriate path to be taken. It ensures that the new element will be inserted into a location that keeps the tree a binary search tree. If the new element to be inserted compares equally to the object in a node, the insert is abandoned with a `return` statement.

```
public boolean insert(E newElement) {
    // newElement will be added and this will still be a
    // BinarySearchTree. This tree will not insert newElement
    // if it will compareTo an existing element equally.
    if (root == null)
      root = new TreeNode(newElement);
    else {
      // find the proper leaf to attach to
      TreeNode curr = root;
      TreeNode prev = root;
      while (curr != null) {
        prev = curr;
        if (newElement.compareTo(curr.data) < 0)
          curr = curr.left;
        else if (newElement.compareTo(curr.data) > 0)
          curr = curr.right;
        else {
          System.out.println(newElement + " in this BST");
          return false;
        }
      }

      // Correct leaf has now been found. Determine whether to
      // link the new node came from prev.left or prev.right
      if (newElement.compareTo(prev.data) < 0)
        prev.left = new TreeNode(newElement);
      else
        prev.right = new TreeNode(newElement);
    }
    return true;
  } // end insert
```

When `curr` finally becomes `null`, it must be from either `prev`'s left or right.



This situation is handled by the code at the end of insert that compares `newElement` to `prev.data`.

E find(E)

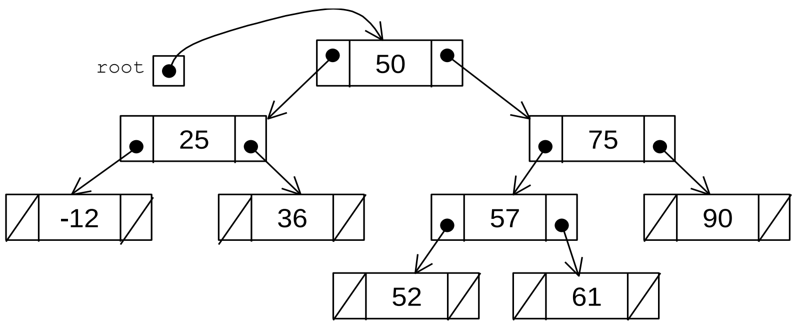
This `BinarySearchTree` needed some way to insert elements before `find` could be tested so insert could be tested, a bit of illogicality. Both will be tested now with a unit test that begins by inserting a small set of integer elements. The `printSideways` message ensures the structure of the tree has the BST ordering property.

```
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.Before;

public class BinarySearchTreeTest {

  private BinarySearchTree<Integer> aBST;
  // Any test method can use aBST with the same 9 integers shown in @Before as
  // setUpBST will be called before each @Test
  @Before
  public void setUpBST() {
    aBST = new BinarySearchTree<Integer>();
    aBST.insert(50);
    aBST.insert(25);
    aBST.insert(75);
    aBST.insert(-12);
    aBST.insert(36);
    aBST.insert(57);
    aBST.insert(90);
    aBST.insert(52);
    aBST.insert(61);
    aBST.printSideways();
  }
}
```

Output

```
        90
    75
            61
        57
            52
50
        36
    25
        -12
```

Picture the state of aBST



The first test method ensures that elements that can be added result in true and those that can't result in false. Programmers could use this to ensure the element was added or the element already existed.

```
@Test
public void testInsertDoesNotAddExistingElements() {
  assertTrue(aBST.insert(789));
  assertTrue(aBST.insert(-789));
  assertFalse(aBST.insert(50));
  assertFalse(aBST.insert(61));
}
```

This test method ensures that the integers are found and that the correct value is returned.

```
@Test
public void testFindWhenInserted() {
  assertEquals(50, aBST.find(50));
  assertEquals(25, aBST.find(25));
  assertEquals(75, aBST.find(75));
  assertEquals(-12,aBST.find(-12));
  assertEquals(36, aBST.find(36));
  assertEquals(57, aBST.find(57));
  assertEquals(90, aBST.find(90));
  assertEquals(52, aBST.find(52));
  assertEquals(61, aBST.find(61));
}
```

And this test method ensures that a few integers not inserted are also not found.

```
@Test
public void testFindWhenElementsNotInserted() {
  assertNull(aBST.find(999));
  assertNull(aBST.find(0));
}
```

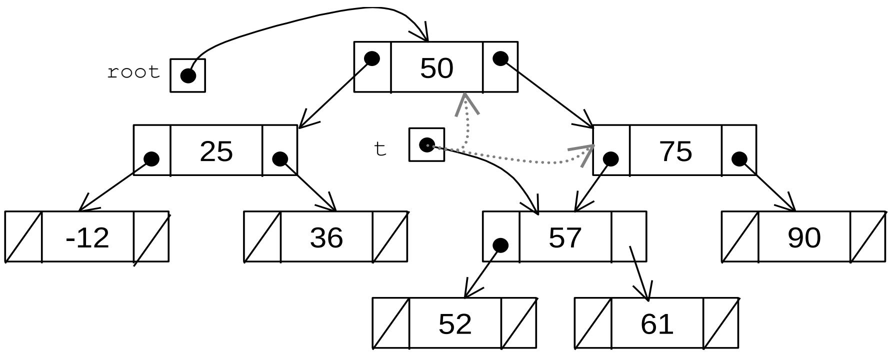
The search through the nodes of a aBST begins at the root of the tree. For example, to search for a node that will compareTo 57 equally, the method first compares 57 to the root element, which has the value of 50. Since 57 is greater than 50, the search proceeds down the *right* subtree (recall that nodes to the right are greater). Then 57 is compared to 75. Since 57 is less than 75, the search proceeds down the *left* subtree of 75. Then 57 is compared to the node with 57. Since these compare equally, a reference to the element is returned to the caller. The binary search continues until one of these two events occur:

1. The element is found
2. There is an attempt to search an empty tree (nowhere to go -- the node is not in the tree)

In the first case, the reference to the data in the node is returned to the sender. In the second case, the method returns null to indicate that the element was not in the tree. Here is an implementation of find method.

```
// Return a reference to the object that will compareTo
// searchElement equally. Otherwise, return null.
public E find(E searchElement) {
  // Begin the search at the root
  TreeNode ref = root;
  // Search until found or null is reached
  while (ref != null) {
    if (searchElement.compareTo(ref.data) == 0)
      return ref.data; // found
    else if (searchElement.compareTo(ref.data) < 0)
      ref = ref.left; // go down the left subtree
    else
      ref = ref.right; // go down the right subtree
  }
  // Found an empty tree. SearchElement was not found
  return null;
}
```

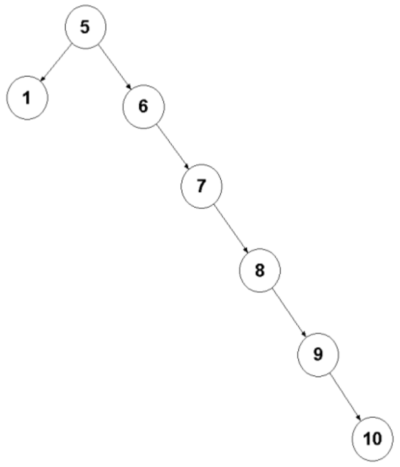
The following picture shows the changing values of the external reference t as it references the three different nodes in its search for 57:



One of the reasons that binary search trees are frequently used to store collections is the speed at which elements can be found. In a manner similar to a binary search algorithm, half of the elements can be eliminated from the search in a BST at each loop iteration. When you go left from one node, you ignore all the elements to the right, which is usually about half of the remaining nodes. Assuming the BinarySearchTree is fairly complete, searching in a binary search tree is O(log n). For example, in the previous search, t referred to only three nodes in a collection of size 9. A tree with 10 levels could have a maximum size of 1,024 nodes. It could take as few as 10 comparisons to find something on level 10.

Efficiency

Much of the motivation for the design of trees comes from the fact that the algorithms are more efficient than those with arrays or linked structures. It makes sense that the basic operations on a binary search tree should require O(log n) time where n is the number of elements of the tree. We know that the height of a balanced binary tree is $\log_2 n$ where *n* is the number elements in the tree. In this case, the cost to find the element should be on the order of O(log *n*). However, with a tree like the following one that is not balanced, runtime performance takes a hit.



If the element we were searching for was the right-most element in this tree (10), the search time would be O(n), the same as a singly linked structure.

Thus, it is very important that the tree remain balanced. If values are inserted randomly to a binary search tree, this condition may be met, and the tree will remain adequately balanced so that search and insertion time will be O(log *n*).

The study of trees has been very fruitful because of this problem. There are many variants of trees, e.g., red-black trees, AVL trees, B-trees, that solve this problem by re-balancing the tree after operations that unbalance it are performed on them. Re-balancing a binary tree is a very tedious task and is beyond the scope of this book. However, it should be noted that having to rebalance a binary tree every now and then adds overhead to the runtime of a program that requires a binary search tree. But if you are mostly searching, which is often the case, the balanced tree might be appropriate.

The table below compares a binary search tree's performance with a sorted array and singly linked structure (the remove method for BST is left as an exercise).

	Sorted Array	Singly Linked	Binary Search Tree
remove	O(n)	O(n)	O(log n)
find	O(log n)	O(n)	O(log n)
insert	O(n)	O(n)	O(log n)