Chapter 25 Binary Search Trees

Objectives

- To design and implement a binary search tree (§25.2).
- To represent binary trees using linked data structures (§25.2.1).
- To search an element in binary search tree (§25.2.2).
- To insert an element into a binary search tree (§25.2.3).
- To traverse elements in a binary tree (§25.2.4).
- To delete elements from a binary search tree (§25.3).
- To display binary tree graphically (§25.4).
- To create iterators for traversing a binary tree (§25.5).

Binary Trees

A list, stack, or queue is a linear structure that consists of a sequence of elements. A binary tree is a hierarchical structure. It is either empty or consists of an element, called the root, and two distinct binary trees, called the left subtree and right subtree.
Binary Tree Terms

- A Binary consists of
  - A root
  - A left binary tree (left child)
  - A right binary tree (right child)
- A node without children is a leaf. A node has one parent, except for the root.

Representing Binary Trees

A binary tree can be represented using a set of linked nodes. Each node contains a value and two links named left and right that reference the left child and right child, respectively.

class TreeNode<E> {
    E element;
    TreeNode<E> left;
    TreeNode<E> right;
    public TreeNode(E o) {
        element = o;
    }
}

Binary Search Tree

- A binary search tree of (key, value) pairs, with no duplicate keys, has the following properties
- Every node a left subtree has keys less than the key of the root
- Every node in its right subtree has keys greater than the key of the node.
- (often we only show the keys)
- What is the difference w.r.t heaps?
Searching an Element in a Binary Search Tree

```java
public search(E element) {
    TreeNode<E> current = root; // Start from the root
    while (current != null)
        if (element key less than the key in current.element) {
            current = current.left; // Go left
        }
        else if (element value greater than the value in current.element) {
            current = current.right; // Go right
        }
        else // Element matches current.element
          return found; // Element is found
    return not found; // Element is not in the tree
}
```

Inserting an Element to a Binary Tree

```java
if (root == null)
    root = new TreeNode(element);
else {
    // Locate the parent node
    current = root;
    while (current != null)
        if (element value < the value in current.element) {
            parent = current;
            current = current.left;
        }
        else if (element value > the value in current.element) {
            parent = current;
            current = current.right;
        }
        else
            return false; // Duplicate node not inserted
    // Create the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);
    return true; // Element inserted
}
```

Trace Inserting 101 into the following tree

```java
if (root == null)
    root = new TreeNode(element);
else {
    // Locate the parent node
    current = root;
    while (current != null)
        if (element value < the value in current.element) {
            parent = current;
            current = current.left;
        }
        else if (element value > the value in current.element) {
            parent = current;
            current = current.right;
        }
        else
            return false; // Duplicate node not inserted
    // Create the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);
    return true; // Element inserted
}
```
Trace Inserting 101 into the following tree, cont.

```java
if (root == null) {
    root = new TreeNode(element);
} else {
    // Locate the parent node
    current = root;
    while (current != null) {
        if (element.value < current.element) {
            parent = current;
            current = current.left;
        } else if (element.value > current.element) {
            parent = current;
            current = current.right;
        } else {
            return false; // Duplicate node not inserted
        }
    }
    // Create the new node and attach it to the parent node
    if (element.value < parent.element) {
        parent.left = new TreeNode(element);
    } else {
        parent.right = new TreeNode(element);
    }
    return true; // Element inserted
}
```
Trace Inserting 101 into the following tree, cont.

```java
if (root == null) {
    root = new TreeNode(element);
} else {
    // Locate the parent node
    current = root;
    while (current != null) {
        if (element < current.element) {
            parent = current;
            current = current.left;
        } else if (element > current.element) {
            parent = current;
            current = current.right;
        } else {
            return false; // Duplicate node not inserted
        }
    }
    // Create the new node and attach it to the parent node
    if (element < parent.element) {
        parent.left = new TreeNode(element);
    } else {
        parent.right = new TreeNode(element);
    }
    return true; // Element inserted
}
```

Insert 101 into the following tree.

```
60
55 100
57 45 67 107
```

```
root
current
parent
```

```
101 > 60 true
```
Trace Inserting 101 into the following tree, cont.

```java
if (root == null)
    root = new TreeNode(element);
else {
    // Locate the parent node
    current = root;
    while (current != null)
        if (element < current.element) {
            parent = current;
            current = current.left;
        } else if (element > current.element) {
            parent = current;
            current = current.right;
        } else
            return false; // Duplicate node not inserted
    // Create the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);
    return true; // Element inserted
}
```

Insert 101 into the following tree.

```
60
55 100
57 45 67 107
root
current
parent
```

101 > 60 true

```
60
55 100
57 45 67 107
root
current
parent
```

101 < 100 false

```
60
55 100
57 45 67 107
root
current
parent
```

101 > 100 true
Insert 101 into the following tree.

if (root == null)
    root = new TreeNode(element);
else {
    // Locate the parent node
    current = root;
    while (current != null)
        if (element value < the value in current.element) {
            parent = current;
            current = current.left;
        } else if (element value > the value in current.element) {
            parent = current;
            current = current.right;
        } else
            return false; // Duplicate node not inserted
    // Create the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);
    return true; // Element inserted
}
Trace Inserting 101 into the following tree, cont.

if (root == null)
    root = new TreeNode(element);
else {
    // Locate the parent node
    current = root;
    while (current != null)
    if (element < current.element) {
        parent = current;
        current = current.left;
    } else if (element > current.element) {
        parent = current;
        current = current.right;
    } else
        return false; // Duplicate node not inserted
    // Create the new node and attach it to the parent node
    if (element < parent.element)
        parent.left = new TreeNode(element);
    else
        parent.right = new TreeNode(element);
    return true; // Element inserted
}

Insert 101 into the following tree.

55
\_ 60
  \_ 100
    \_ 57
    \_ 45
    \_ 67
    \_ 107
\_ root
\_ current
\_ parent

60
55 100
57 45 67 107
root
current
parent

101 < 107 true
Trace Inserting 101 into the following tree, cont.

```java
if (root == null) {
    root = new TreeNode(element);
} else {
    // Locate the parent node
    current = root;
    while (current != null) {
        if (element < the value in current.element) {
            parent = current;
            current = current.left;
        } else if (element > the value in current.element) {
            parent = current;
            current = current.right;
        } else {
            return false; // Duplicate node not inserted
        }
    }
    // Create the new node and attach it to the parent node
    if (element < parent.element) {
        parent.left = new TreeNode(element);
    } else {
        parent.right = new TreeNode(element);
    }
    return true; // Element inserted
}
```

Insert 101 into the following tree.

```
60
55 100
57 45 67 107
root
Since current.left is null, current becomes null
```

```
101 < 107 true
```

Trace Inserting 101 into the following tree, cont.

```java
if (root == null) {
    root = new TreeNode(element);
} else {
    // Locate the parent node
    current = root;
    while (current != null) {
        if (element.value < current.element) {
            parent = current;
            current = current.left;
        } else if (element.value > current.element) {
            parent = current;
            current = current.right;
        } else {
            return false; // Duplicate node not inserted
        }
        // Create the new node and attach it to the parent node
        if (element.value < parent.element) {
            parent.left = new TreeNode(element);
        } else {
            parent.right = new TreeNode(element);
        }
        return true; // Element inserted
    }
}
```

Insert 101 into the following tree.

```
60
55 100
57 45 67 107
root
parent
101
```

Inserting 59 into the Tree

```java
if (root == null) {
    root = new TreeNode(element);
} else {
    // Locate the parent node
    current = root;
    while (current != null) {
        if (element.value < current.element) {
            parent = current;
            current = current.left;
        } else if (element.value > current.element) {
            parent = current;
            current = current.right;
        } else {
            return false; // Duplicate node not inserted
        }
        // Create the new node and attach it to the parent node
        if (element.value < parent.element) {
            parent.left = new TreeNode(element);
        } else {
            parent.right = new TreeNode(element);
        }
        return true; // Element inserted
    }
}
```

```
60
55 100
57 45 67 107
root
parent
59 101
```

Tree Traversal

Tree traversal is the process of visiting each node in the tree exactly once. There are several ways to traverse a tree. This section presents depth-first: in-, pre-, post order and breadth-first: level order traversals.

**InOrder**
- The inorder traversal is to visit the left subtree of the current node first recursively, then the current node itself, and finally the right subtree of the current node recursively.

**Postorder**
- The postorder traversal is to visit the left subtree of the current node first, then the right subtree of the current node, and finally the current node itself.
Tree Traversal, cont.

- **Preorder**
  - The preorder traversal is to visit the current node first, then the left subtree of the current node recursively, and finally the right subtree of the current node recursively.

- **Level order**
  - The level order (breadth-first) traversal is to visit the nodes level by level. First visit the root, then all children of the root from left to right, then grandchildren of the root from left to right, and so on.

---

Breadth-first traversal (BFS)

- Breadth-first processes the tree **level by level** starting at the root and handling all the nodes at a particular level from **left to right**.

- To achieve we use a Queue, because the parent child references are not sufficient.
Breadth-first traversal

LevelOrder

<table>
<thead>
<tr>
<th>Step</th>
<th>Queue</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[A]</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>[B,C]</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>[C,D]</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>[E,F,G,H]</td>
<td>A C D</td>
</tr>
<tr>
<td>5</td>
<td>[F,G,H]</td>
<td>A C D E</td>
</tr>
<tr>
<td>7</td>
<td>[H,I]</td>
<td>A B C D E F G</td>
</tr>
<tr>
<td>8</td>
<td>[I]</td>
<td>A B C D E F G H</td>
</tr>
<tr>
<td>9</td>
<td>()</td>
<td>A B C D E F G H</td>
</tr>
</tbody>
</table>

The Tree Interface

The Tree interface defines common operations for trees.

- `search(e: E): boolean` - Returns true if the specified element is in the tree.
- `insert(e: E): boolean` - Returns true if the element is added successfully.
- `delete(e: E): boolean` - Returns true if the element is removed from the tree successfully.
- `inorder(): void` - Prints the nodes in inorder traversal.
- `preorder(): void` - Prints the nodes in preorder traversal.
- `postorder(): void` - Prints the nodes in postorder traversal.
- `getSize(): int` - Returns the number of elements in the tree.
- `isEmpty(): boolean` - Returns true if the tree is empty.
- `clear(): void` - Removes all elements from the tree.

Override the `add`, `isEmpty`, `remove`, `containsAll`, `addAll`, `removeAll`, `retainAll`, `toArray()`, and `toArray(T[])` methods defined in `Collection` using default methods.
The BST Class

Let's define the binary tree class, named BST with A concrete BST class can be defined to extend AbstractTree.

```
interface Tree<E>

class BST<E extends Comparable<E>>

root: TreeNode<E>
size: int

+BST()
+BST(objects: E[]):
+path(e: E): java.util.List<TreeNode<E>>
```

The root of the tree.
The number of nodes in the tree.

+BST(): Creates a default BST.

+BST(objects: E[]): Creates a BST from an array of elements.

+path(e: E): Returns the path of nodes from the root leading to the node for the specified element. The element may not be in the tree.

Deleting Elements in a Binary Search Tree

- Locate the node that contains the element and its parent node.
- Let current point to the node that contains the element in the binary tree and parent point to the parent of the current node. (notice: parent can be the root reference)
- The current node may be a left child or a right child of the parent node.
- There are two cases.

Deleting Elements in a Binary Search Tree

Case 1: The current node does not have a left child, as shown in this figure (a). Simply connect the parent with the right child of the current node, as shown in this figure (b).
Deleting Elements in a Binary Search Tree

For example, to delete node 10 in Figure 25.9a. Connect the parent of node 10 with the right child of node 10, as shown in Figure 25.9b.

Deleting Elements in a Binary Search Tree

Case 2: The current node has a left child.

- Let rightMost point to the node that contains the largest element in the left subtree and parentOfRightMost point to its parent node.
- Note that the rightMost node cannot have a right child, but may have a left child.
- Replace the element value in the current node with the one in the rightMost node.
- Connect the parentOfRightMost node with the left child of the rightMost node, and delete the rightMost node.

Deleting Elements in a Binary Search Tree

Case 2 diagram
Deleting Elements in a Binary Search Tree

Case 2 example, delete 20

Examples

Delete this node

Examples

Delete this node
Examples

Alternative, more balanced approach

Cases to Consider
- Delete something that is not there
  - Throw exception
- Delete a leaf
  - Easy, just set link from parent to null
- Delete a node with one child
- Delete a node with two children

Delete

Case 1: one child
Delete
Case 2: two children

Which are valid replacement nodes?

4 and 6, WHY?

max of left, min of right

what would be a good one here?

6, WHY?

Digression: inorder traversal of BST

- In order:
  - go left
  - visit the node
  - go right
- The keys of an inorder traversal of a BST are in sorted order!

Replace with successor

Replace root with its leftmost right descendant and replace that node with its right child, if necessary (an easy delete case). That node is the inorder successor of the root.

Can that node have two children? A left child?
Replace with predecessor

\[ \text{delete}(5) \]

\[
\begin{array}{c}
\text{5} \\
\text{2} & \text{8} \\
\text{1} & \text{4} & \text{6} & \text{9} \\
\end{array}
\]

\[
\begin{array}{c}
\text{2} \\
\text{4} & \text{8} \\
\text{1} & \text{6} & \text{9} \\
\end{array}
\]

Replace root with its rightmost left descendant and replace that node with its left child, if necessary (an easy delete case). That node is the inorder predecessor of the root.

Can that node have two children? A right child?

Delete

Case 2: two children

1. Find the *inorder successor or predecessor M* of N’s search key.
   - The node whose search key comes immediately after or before N’s search key
2. Copy the item of M, to the deleting node N.
3. Remove the node M from the tree.

Iterators

An *iterator* is an object that provides a uniform way for traversing the elements in a container such as a set, list, binary tree, etc.