CSE 373: Data Structures and Algorithms

Lecture 8: Trees
Set ADT

- **set**: A collection that does not allow duplicates
  - We don't think of a set as having indexes; we just add things to the set in general and don't worry about order

- **basic set operations**:
  - **insert**: Add an element to the set (order doesn't matter).
  - **remove**: Remove an element from the set.
  - **search**: Efficiently determine if an element is a member of the set.

```java
set.contains("to")  // true
set.contains("be")  // false
```
Sets in computer science

• Databases:
  – set of records in a table

• Search engines:
  – set of URLs/webpages on the Internet

• Real world examples:
  – set of all products for sale in a store inventory
  – set of friends on Facebook
  – set of email addresses
Using Sets

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(value)</td>
<td>adds the given value to the set</td>
</tr>
<tr>
<td>contains(value)</td>
<td>returns true if the given value is found in this set</td>
</tr>
<tr>
<td>remove(value)</td>
<td>removes the given value from the set</td>
</tr>
<tr>
<td>clear()</td>
<td>removes all elements of the set</td>
</tr>
<tr>
<td>size()</td>
<td>returns the number of elements in list</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>returns true if the set's size is 0</td>
</tr>
<tr>
<td>toString()</td>
<td>returns a string such as &quot;[3, 42, -7, 15]&quot;</td>
</tr>
</tbody>
</table>

List<String> list = new ArrayList<String>();
...
Set<Integer> set = new TreeSet<Integer>(); // empty
Set<String> set2 = new HashSet<String>(list);

- can construct an empty set, or one based on a given collection
More Set operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addAll(collection)</code></td>
<td>Adds all elements from the given collection to this set</td>
</tr>
<tr>
<td><code>containsAll(coll)</code></td>
<td>Returns <code>true</code> if this set contains every element from the given set</td>
</tr>
<tr>
<td><code>equals(set)</code></td>
<td>Returns <code>true</code> if the given other set contains the same elements</td>
</tr>
<tr>
<td><code>iterator()</code></td>
<td>Returns an object used to examine the set's contents</td>
</tr>
<tr>
<td><code>removeAll(coll)</code></td>
<td>Removes all elements in the given collection from this set</td>
</tr>
<tr>
<td><code>retainAll(coll)</code></td>
<td>Removes elements <em>not</em> found in the given collection from this set</td>
</tr>
<tr>
<td><code>toArray()</code></td>
<td>Returns an array of the elements in this set</td>
</tr>
</tbody>
</table>
Accessing elements in a Set

for (\texttt{type name : collection}) {
  statements;
}

- Provides a clean syntax for looping over the elements of a Set, List, array, or other collection

\begin{verbatim}
Set<Double> grades = new TreeSet<Double>();
...
for (double grade : grades) {
  System.out.println("Student grade: "+ grade);
}
\end{verbatim}

- needed because sets have no indexes; can't get element i
Sets and ordering

• **HashSet**: elements are stored in an unpredictable order

```java
Set<String> names = new HashSet<String>();
names.add("Jake");
names.add("Robert");
names.add("Marisa");
names.add("Kasey");
System.out.println(names);
// [Kasey, Robert, Jake, Marisa]
```

• **TreeSet**: elements are stored in their "natural" sorted order

```java
Set<String> names = new TreeSet<String>();

// [Jake, Kasey, Marisa, Robert]
```
## Implementing Set ADT

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Remove</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsorted array</strong></td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td><strong>Sorted array</strong></td>
<td>$\Theta(\log(n) + n)$</td>
<td>$\Theta(\log(n) + n)$</td>
<td>$\Theta(\log(n))$</td>
</tr>
<tr>
<td><strong>Linked list</strong></td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>
Trees

• **tree**: A directed, acyclic structure of linked nodes.
  – *directed*: Has one-way links between nodes.
  – *acyclic*: No path wraps back around to the same node twice.
  – *binary tree*: One where each node has at most two children.

• A binary tree can be defined as either:
  – empty (*null*), or
  – a **root** node that contains:
    • data,
    • a **left** subtree, and
    • a **right** subtree.
    – (The left and/or right subtree could be empty.)
Trees in computer science

- folders/files on a computer
- family genealogy; organizational charts
- AI: decision trees
- compilers: parse tree
  - $a = (b + c) \times d$;
- cell phone T9
Terminology

• **node**: an object containing a data value and left/right children
• **root**: topmost node of a tree
• **leaf**: a node that has no children
• **branch**: any internal node; neither the root nor a leaf

• **parent**: a node that refers to this one
• **child**: a node that this node refers to
• **sibling**: a node with common parent
// A StringTreeNode object is one node in a binary tree of Strings.
public class StringTreeNode {
    public String data; // data stored at this node
    public StringTreeNode left; // reference to left subtree
    public StringTreeNode right; // reference to right subtree

    // Constructs a leaf node with the given data.
    public StringTreeNode(String data) {
        this(data, null, null);
    }

    // Constructs a leaf or branch node with the given data and links.
    public StringTreeNode(String data, StringTreeNode left,
                          StringTreeNode right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }
}
Binary search trees

• **binary search tree** ("BST"): a binary tree that is either:
  – empty (null), or
  – a root node R such that:
    • every element of R's left subtree contains data "less than" R's data,
    • every element of R's right subtree contains data "greater than" R's,
    • R's left and right subtrees are also binary search trees.

• BSTs store their elements in sorted order, which is helpful for searching/sorting tasks.
Exercise

• Which of the trees shown are legal binary search trees?
Programming with Binary Trees

• Many tree algorithms are recursive
  – Process current node, recur on subtrees
  – Base case is empty tree (null)

• traversal: An examination of the elements of a tree.
  – A pattern used in many tree algorithms and methods

• Common orderings for traversals:
  – pre-order: process root node, then its left/right subtrees
  – in-order: process left subtree, then root node, then right
  – post-order: process left/right subtrees, then root node
Tree Traversal (in order)

// Returns a String representation of StringTreeSet with elements in
// their "natural order" (e.g., [Jake, Kasey, Marisa, Robert]).
public String toString() {
    String str = "[" + toString(root);
    if (str.length() > 1) { str = str.substring(0, str.length()-2); }
    return str + "]";
}

// recursive helper; in-order traversal
private String toString(StringTreeNode root) {
    String str = "";
    if (root != null) {
        str += toString(root.left);
        str += root.data + ", ";
        str += toString(root.right);
    }
    return str;
}
Implementing Set with BST

• Each Set entry adds a node to tree
  – Node contains String element, references to left/right subtree

• Tree organized for binary search
  – Quickly search or place to insert/remove element
Implementing Set with BST (cont.)

```java
public interface StringSet {
    public boolean add(String value);
    public boolean contains(String value);
    public void print();
    public boolean remove(String value);
    public int size();
}
```
StringTreeSet class

// A StringTreeSet represents a Set of Strings.
public class StringTreeSet {
    private StringTreeNode root;  // null for an empty set

    // methods
}
– Client code talks to the
    StringTreeSet, not to the node
    objects inside it

– Methods of the StringTreeSet
    create and manipulate the nodes,
    their data and links between them
Set implementation: search

```java
public boolean contains(String value) {
    return contains(root, value);
}

private boolean contains(StringTreeNode root, String value) {
    if (root == null) {
        return false; // not in set
    } else if (root.data.compareTo(value) == 0) {
        return true; // found!
    } else if (root.data.compareTo(value) > 0) {
        return contains(root.left, value); // search left
    } else {
        return contains(root.right, value); // search right
    }
}
```
Set implementation: insert

• **Starts like** `contains`
  – Trace out path where node should be

• **Add node as new leaf**
  – Don't change any other nodes or references
  – Correct place to maintain binary search tree property
Set implementation: insert

public boolean add(String value) {
    int oldSize = size();
    this.root = add(root, value);
    return oldSize != size();
}

private StringTreeNode add(StringTreeNode root, String value) {
    if (root == null) {
        root = new StringTreeNode(value);
        numElements++;
    } else if (root.data.compareTo(value) == 0) {
        return root;
    } else if (root.data.compareTo(value) > 0) {
        root.left = add(root.left, value);
    } else {
        root.right = add(root.right, value);
    }
    return root;
}
Set implementation: remove

- Possible states for the node to be removed:
  - a leaf: replace with null
  - a node with a left child only: replace with left child
  - a node with a right child only: replace with right child
  - a node with both children: replace with min value from right

```
set.remove("L");
```
Set implementation: remove

```java
public boolean remove(String value) {
    int oldSize = size();
    root = remove(root, value);
    return oldSize != size();
}

private StringTreeNode remove(StringTreeNode root, String value) {
    if (root == null) { return root; }
    else if (root.data.compareTo(value) > 0) {
        root.left = remove(root.left, value);
    } else if (root.data.compareTo(value) < 0) {
        root.right = remove(root.right, value);
    } else {
        numElements--;
        if (root.right != null && root.left != null) {
            root.data = findMin(root.right).data;
            root.right = remove(root.right, root.data);
        } else if (root.right != null) {
            root = root.right;
        } else {
            root = root.left;
        }
    }
    return root;
```
Evaluate Set as BST

• Space used
  – Overhead of two references per entry
  – BST adds nodes as needed; no excess capacity

• Runtime
  – add, contains take time proportional to tree height
  – height expected to be $O(\log N)$
A Balanced Tree

- Values: 2 8 14 15 18 20 21
  - Order added: 15, 8, 2, 20, 21, 14, 18
- Different tree structures possible
  - Depends on order inserted
- 7 nodes, expected height log 7 ≈ 3
- Perfectly balanced

```
• 2
  • 8
    • 2
  • 14
  • 15
    • 20
       • 18
          • 21
    • 14
```

root
Mostly Balanced Tree

- Same Values: 2 8 14 15 18 20 21
  - Order added: 20, 8, 21, 18, 14, 15, 2
- Mostly balanced, height 4/5
Degenerate Tree

- Same Values: 2 8 14 15 18 20 21
  - Order added: 2, 8, 14, 15, 18, 20, 21
- Totally unbalanced, height 7
Binary Trees: Some Numbers

Recall: height of a tree = length of longest path from the root to a leaf.

For binary tree of height $h$:

- max # of leaves: $2^h$
- max # of nodes: $2^{(h + 1)} - 1$
- min # of leaves: $1$
- min # of nodes: $h + 1$

We’re not going to do better than $\log(n)$ height, and we need something to keep us away from $n$. 
## Implementing Set ADT (Revisited)

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Remove</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unssorted array</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted array</td>
<td>$\Theta(\log(n)+n)$</td>
<td>$\Theta(\log(n) + n)$</td>
<td>$\Theta(\log(n))$</td>
</tr>
<tr>
<td>Linked list</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>BST (if balanced)</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>