CSE 373: Data Structures and Algorithms

Lecture 9: 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><code>add(value)</code></td>
<td>adds the given value to the set</td>
</tr>
<tr>
<td><code>contains(value)</code></td>
<td>returns true if the given value is found in this set</td>
</tr>
<tr>
<td><code>remove(value)</code></td>
<td>removes the given value from the set</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>removes all elements of the set</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>returns the number of elements in list</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>returns true if the set's size is 0</td>
</tr>
<tr>
<td><code>toString()</code></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

**A \cup B**  Union  

**A \cap B**  Intersection  

**A - B**  Difference  

<table>
<thead>
<tr>
<th>Method</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 true if this set contains every element from given set</td>
</tr>
<tr>
<td><code>equals(set)</code></td>
<td>returns true if given other set contains the same elements</td>
</tr>
<tr>
<td><code>iterator()</code></td>
<td>returns an object used to examine set's contents</td>
</tr>
<tr>
<td><code>removeAll(collection)</code></td>
<td>removes all elements in the given collection from this set</td>
</tr>
<tr>
<td><code>retainAll(collection)</code></td>
<td>removes elements <em>not</em> found in 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 (type name : collection) {
    statements;
}

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

Set<Double> grades = new TreeSet<Double>();
...

for (double grade : grades) {
    System.out.println("Student grade: " + grade);
}

– 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>$\Theta(n)$</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>$\Theta(n)$</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;
}