- What distinguishes a tree from a linked list? What gets stored at each tree Node?
- What technique becomes useful for implementing operations on trees? Why?
- What is our motivations for looking at trees again? e.g. Why not just stick to ArrayList/LinkedList?
- New: How might one implement an iterator for a tree?
Binary Search Tree Property

A binary tree where every node $N$ in the BST

- Any data in the tree rooted at $N$.left sort before $N$.data
- Any data in the tree rooted at $N$.right sort after $N$.data
Comparisons

How does java guarantee comparability?

**Comparable**
Data can implement Comparable

```java
int c = x.compareTo(y);
// neg for x < y, right order
// 0 for x = y, don’t care
// pos for x > y, wrong order
```

**Comparator**
Use a Comparator object to do comparisons

```java
Comparator<Thing> cmp =
        new ...;
int c = cmp.compare(x,y);
// neg for x < y, right order
// 0 for x = y, don’t care
// pos for x > y, wrong order
```

Presence of both hints at a fundamental problem
Define `bst.find()`

- `find(T x)` is publicly accessible
  
  `tree.find("Mario");`

- **Define**
  
  `find(T x, Node<T> t)` which works on a given start node

- **Compare via** `Comparable`:
  
  `if(x.compareTo(t.data) < 0)`

Give 2 versions

- **Recursive**
  
  ```java
  public class BinarySearchTree
      <T extends Comparable<T>>
  {
      protected Node<T> root;
      // Return x if in tree, // null otherwise
      public T find(T x)
      {
          Node<T> result =
              find(x, this.root);
          if(result == null){ return null;} else{ return result.data; }
      }
  }
  // Find node containing x // starting at node t // Return null if not found
  private static Node<T> find(T x, Node<T> t){
      // DEFINE ME
  }
  ```

- **Iterative**
Recursive find(x, node)

Use key of data to search through tree

- Left for less than
- Right for greater than

// pseudocode
Node<T> find(x, t)
{
    if (t == null)
    {
        return null;
    }

    int diff = x.compareTo(t.data);
    if (diff < 0)  // x < t
    {
        return find(x, t.left);
    } else if (diff > 0)  // x > t
    {
        return find(x, t.right);
    } else  // x==t.data
    {
        return t.data;  // found
    }
}
Iterative `find(x,node)`

See `weiss/nonstandard/BinarySearchTree.java`

```java
private static BinaryNode<T> find(T x, BinaryNode<T> t){
    while( t != null ) {
        if( x.compareTo( t.data ) < 0 )
            t = t.left;
        else if( x.compareTo( t.data ) > 0 )
            t = t.right;
        else
            return t; // Match
    }
    return null; // Not found
}
```
What is the worst-case complexity of \texttt{find}(x) in terms of tree properties?

Construct a tree with this worst case complexity.
Draw the tree that results from the following sequence of insertions.

MyBST<String> t = new MyBST<String>();
t.insert("Mario");
t.insert("Goomba");
t.insert("Luigi");
t.insert("Toad");
t.insert("Wario");
t.insert("Princess");
t.insert("Bowser");
t.insert("Chain Chomp");
May need to change a left or right pointers, redefine root

No duplication, define a TreeSet, exception on duplicate insert

Define Recursive Insert

```java
class BinarySearchTree<T> {
    Node<T> root=null; int size=0;
    public void insert( T x ){
        root = insert( x, root );
        this.size++;
    }
    private static Node<T> insert( T x, Node<T> t ){
        // DEFINE ME
    }
}
```

Define Iterative Insert if You’re Brave

```java
public void insert( T x ){
    // DEFINE ME
}
```
Recursive insert(x,t)

From weiss/nonstandard/BinarySearchTree.java

class BinarySearchTree<T> {
    Node<T> root;
    public void insert( T x ){
        root = insert( x, root );
    }
    private static Node<T> insert( T x, Node<T> t )
    {
        if( t == null )
            t = new Node<T>( x );
        else if( x.compareTo( t.data ) < 0 )
            t.left = insert( x, t.left );
        else if( x.compareTo( t.data ) > 0 )
            t.right = insert( x, t.right );
        else
            throw new DuplicateItemException( x.toString( ) );
        return t;
    }
}

Binary Search Tree remove(x)

// Public method, eliminate x if present in tree
public void remove(T x);

// Recursive helper method
private Node<T> remove(T x, Node<T> t);

- Get rid of a node with data x in a binary tree; throw exception if not present (or ignore request)
- More involved than find/insert
- Preserve Tree Structure
- Recursion greatly eases implementation
Consider Mario Tree

- Describe which \texttt{cases} exist \texttt{tree.remove(x)}?
- Which of these do you anticipate being easy/hard to code for?
Cases for `t.remove(x)`

1. `x` not in tree
   - Leave tree as is or raise an exception

2. `x` at a node with no children
   - Get rid of node containing `x`

3. `x` at a node with 1 child
   - "Pass over" node containing `x`

4. `x` at a node with 2 children
   - Find a next node in sorting order
   - Replace `x` with next node's data
   - Remove next node
   - Next is minimum of right subtree
class BST<T> {
    private Node<T> root;

    // Public facing method, find minimum element and return it
    public T findMin(){ return this.findMin(this.root); }

    // Private helper method return the smallest element in the
    // tree rooted at t
    private T findMin(Node<T> t){
        // DEFINE ME
    }

    // Public facing method, eliminate the smallest data in tree
    public void removeMin(){ this.root = removeMin(this.root); }

    // Recursive helper; remove the node with the smallest data
    // in it in the tree rooted at t. The node returned is used
    // to alter the structure of the tree.
    private Node<T> removeMin(Node<T> t){
        // DEFINE ME
    }
}
Warm-up Questions

1. What is the Binary Search Tree property?
2. Are all trees binary trees? Do all binary trees have the BST property? (give counter-examples)
3. Where is the biggest data element in a BST? The smallest?
4. What are the runtime complexities of BST tree.find(x) and tree.insert(x)?
5. Which kinds of nodes are easy to remove from BSTs? Which kinds are more difficult?
6. What is a useful strategy for removing difficult nodes?
Children Cases for \texttt{remove(t,x)}

**One Child: Remove 5**

1. Find node \( t \) with data \( x \)
2. Replace with only child

**Two Children: Remove 2**

1. Find node \( t \) with data \( x \)
2. Find min node of \( t \).right:
   - min must have 0/1 child
3. Replace \( t \).data with \( \text{min} \).data
4. Remove \( \text{min} \)
Lesson from insert()

- Recall in `insert(x, t)`, did stuff like
  
  ```
  t.right = insert(x, t.right);
  // a new/existing node is returned by insert()
  ```

- Take the same approach for `remove(x, t)`
- Assume these helpers are Available
  
  ```
  T findMin(Node<T> t); Node<T> removeMin(Node<T> t)
  ```

Implement Recursive `remove(x, t)`

- How to know if `t` is the node?
- What to do if `t` isn't the node?
- If `t` is the node, are there separate cases for action?
Cases for recursive remove()

1. t is null
   - Throw an exception
     throw new ItemNotFoundException();
   - Or do nothing to the tree
     return null;

2. x less than t.data (recurse left)
   - t.left = remove(t.left, x);

3. x greater than t.data (recurse right)
   - t.right = remove(t.right, x);

4. x equals t.data (remove t)
   - t has 0 children, get rid of t
   - t has 1 child, pass over t
   - t has 2 children, replace with next/prev
Case 4: $x$ equals $t$.data (remove $t$)

Helper methods defined elsewhere

```java
T findMin(Node<T> t);    Node<T> removeMin(Node<T> t)
```

- $t$ has 0 children, get rid of $t$
  ```java
  return null;
  ```
- $t$ has 1 child, pass over $t$
  ```java
  (t.left!=null) ? return t.left : return t.right;
  ```
- $t$ has 2 children, replace with next or prev
  ```java
  t.data = findMin(t.right);
  t.right = removeMin(t.right);
  return t;
  ```
- How are $\text{findMin}(t)$ and $\text{removeMin}(t)$ implemented?
  - Where is the minimum node in a tree?
  - How many children does it have?
private Node<T> remove( T x, Node<T> t ){
    if( t == null )
        throw new ItemNotFoundException( x.toString( ) );
    if( x.compareTo( t.data ) < 0 )
        t.left = remove( x, t.left );
    else if( x.compareTo( t.data ) > 0 )
        t.right = remove( x, t.right );
    // Found at this node
    else if( t.left != null && t.right != null ){
        // Two children
        t.data = findMin( t.right );
        t.right = removeMin( t.right );
    }
    else // One child or no children
        t = ( t.left != null ) ? t.left : t.right;
    return t;
}
So Far

Binary Search Trees

- Defined find() / insert() / remove()
- Helpers: findMin() / findMax() / removeMin() / removeMax()
- All ops runtime complexity $O(Height)$
- Discuss balancing trees to ensure that $Height \approx \log(Size)$