CS 310: Binary Search Trees

Chris Kauffman

Week 11-1
Logistics

Reading
- Weiss Ch. 7 Recursion
- Weiss Ch. 19 BSTs

Today
- Tree traversals
- Mention Tree Iterators
- Comparable vs Comparator
- Binary Search Trees

HW2
- Due Thursday
- Questions?

HW3
- Will post and discuss on Thursday
- Last HW of the course
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 before y
// 0 for x equals y
// pos for x after y
```

### Comparator

Use a `Comparator` object to do comparisons

```java
Comparator<Thing> cmp = new ...;
int c = cmp.compare(x,y);
// neg for x before y
// 0 for x equals y
// pos for x after y
```

- Generalizes subtraction: `c = x-y;`
- Presence of both hints at a fundamental problem
Comparators allow multiple sorting criteria

```java
public class RCComp<RowCol> {
    public int compare(RowCol a, RowCol b) {
        // compare a.row to b.row
        // if tied, compare a.col to b.col
    }
}

public class CRComp<RowCol> {
    public int compare(RowCol a, RowCol b) {
        // compare a.col to b.col
        // if tied, compare a.row to b.row
    }
}

main(){
    RowCol x = ...;
    RowCol y = ...;
    Comparator<RowCol> rc = new RCComp();
    if(rc.compare(x,y) < 0){
        // x before y according to rc
    } else if(rc.compare(x,y) > 0){
        // x after y according to rc
    } else{
        // x equals y according to rc
    }
    Comparator<RowCol> cr = new CRComp();
    if(cr.compare(x,y) < 0){
        // x before y according to cr
    } else if(cr.compare(x,y) > 0){
        // x after y according to cr
    } else{
        // x equals y according to cr
    }
}
```
Comparison Type Conventions

`java.util.TreeSet` and similar trees EITHER

1. Contain objects that implement `Comparable` (things in tree can compare to one another via `x.compareTo(y)`
2. Constructed with a `Comparator` as in

   ```java
   Comparator<Something> cmp = ...;
   TreeSet<Something> t = new TreeSet<Something>(cmp);
   ```

In example code, we’ll presume objects are `Comparable` for convenience

Generics get Funky

```java
class Tree< T extends Comparable<? super T> >
```

Tree of something stuff that *descends from something Comparable to T*

```java
class Tree< T extends Comparable<T> >
```

Tree of stuff that is comparable to itself only
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

- Iterative

```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){
        if(x.compareTo(t.data) < 0)
            return find(x, t.left);
        else if(x.compareTo(t.data) > 0)
            return find(x, t.right);
        else
            return t;
    }
}
```
Recursive `find(x,node)`

Use key of data to search through tree
- Left for less than
- Right for greater than

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

See weiss/nonstandard/BinarySearchTree.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.
Examples of Insert

Play with MyBST.java in JGrasp and look at the pretty pictures after multiple insert(x) calls
Insertion: Similar to \texttt{find(x)}

- May need to change a left or right pointers, redefine root
- No duplication, define a TreeSet, exception on duplicate insert

Define Recursive Insert


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

Define Iterative Insert if You’re Brave

public void insert( T x ){
    // DEFINE ME
}
Exercise: 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;
    }
}
class BinarySearchTree<T> {
    Node<T> root;
    public void insert(T x) {
        if (this.root == null) { // New root?
            this.root = new Node<T>(x); // Yes
            return;
        } //
        Node<T> t = this.root; // Root exists
        while (true) { // Traverse tree
            int diff = x.compareTo(t.data); // Traverse tree
            if (diff == 0) { // Check duplicate
                throw new DuplicateItemException(x.toString());
            }
            if (diff < 0) { // Go left
                if (t.left == null) { // Found insertion point?
                    t.left = new Node<T>(x); // Yes
                    return;
                } //
                else { // No: Go farther left
                    t = t.left;
                }
            }
            else { // Go right
                if (t.right == null) { // Found insertion point?
                    t.right = new Node<T>(x); // Yes
                    return;
                } //
                else { // No: Go farther left
                    t = t.right;
                }
            }
        }
    }
}
// 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
Prelims

Consider Mario Tree

- Describe which 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 nodes 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

![Tree (a)](image1)

![Tree (b)](image2)

**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 \( min.data \)
4. Remove min

![Tree (a)](image3)

![Tree (b)](image4)
Recursive Implementation: Think Locally

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

\[
T \text{ findMin}(\text{Node}<T> \ t); \quad \text{Node}<T> \text{ removeMin}(\text{Node}<T> \ t)
\]

- \( t \) has 0 children, get rid of \( t \)
  
  return null;

- \( t \) has 1 child, pass over \( t \)
  
  \((t.left!=null) \ ? \ return \ t.left : return \ t.right;\)

- \( t \) has 2 children, replace with \textit{next} or prev
  
  \( t.data = \text{findMin}(t.right); \)
  \( t.right = \text{removeMin}(t.right); \)
  return \( t \);

- How are \textit{findMin}(\( t \)) and \textit{removeMin}(\( t \)) implemented?
  
  - Where is the minimum node in a tree?
  
  - How many children does it have?
remove(x)

Adapted from weiss/nonstandard/BinarySearchTree.java

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 $Height \approx \log(\text{Size})$

Next Time

- Balanced BSTs: AVL and Red-Black Trees
- Reading: Weiss Ch. 19.4-5