Comp215: Trees

Dan S. Wallach (Rice University)

Copyright © 2015, Dan S. Wallach. All rights reserved.
Sometimes, a list isn’t good enough

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>List</th>
<th>Binary Tree</th>
<th>Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insert</strong></td>
<td>O(1) unordered</td>
<td>O(1) unordered</td>
<td>O(log n) expected</td>
<td>O(1) expected</td>
</tr>
<tr>
<td></td>
<td>O(n) ordered</td>
<td>O(n) ordered</td>
<td>O(n) worst case</td>
<td>O(n) worst case</td>
</tr>
<tr>
<td><strong>Search</strong></td>
<td>O(n) unordered</td>
<td>O(n)</td>
<td>O(log n) expected</td>
<td>O(1) expected</td>
</tr>
<tr>
<td></td>
<td>O(log n) binary search</td>
<td>O(n)</td>
<td>O(n) worst case</td>
<td>O(n) worst case</td>
</tr>
<tr>
<td><strong>Index (get i-th element in order)</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>not easily supported</td>
<td>not easily supported</td>
</tr>
<tr>
<td><strong>Delete specific element</strong></td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(log n) expected</td>
<td>O(k), k = packing per bin</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>O(n) worst case</td>
<td>O(n) worst case</td>
</tr>
<tr>
<td><strong>Range Query (e.g., all elements &gt; i )</strong></td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(log n) expected</td>
<td>not easily supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>O(n) worst case</td>
<td></td>
</tr>
<tr>
<td><strong>Copy</strong></td>
<td>O(n)</td>
<td>O(1) functional</td>
<td>O(1) functional</td>
<td>O(n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O(n) mutating</td>
<td>O(n) mutating</td>
<td></td>
</tr>
</tbody>
</table>
Binary trees

Intuition: each element has a value, and two children trees left and right
Everything less than value is on the left, everything greater is on the right.
Wander from the root to the leaves, insert when you hit a leaf.
Example: inserting (5, 9, 2, 4, 7, 3, 8, 6)
Binary trees

Intuition: each element has a value, and two children trees left and right

Everything less than value is on the left, everything greater is on the right.

Wander from the root to the leaves, insert when you hit a leaf.

Example: inserting (5, 9, 2, 4, 7, 3, 8, 6)
Binary trees

Intuition: each element has a *value*, and two children trees *left* and *right*.

Everything less than *value* is on the left, everything greater is on the right.

**Wander from the root to the leaves, insert when you hit a leaf.**

Example: inserting (5, 9, 2, 4, 7, 3, 8, 6)
Binary trees

Intuition: each element has a value, and two children trees left and right

Everything less than value is on the left, everything greater is on the right.

Wander from the root to the leaves, insert when you hit a leaf.

Example: inserting (5, 9, 2, 4, 7, 3, 8, 6)
Binary trees

Intuition: each element has a value, and two children trees left and right.

Everything less than value is on the left, everything greater is on the right.

Wander from the root to the leaves, insert when you hit a leaf.

Example: inserting (5, 9, 2, 4, 7, 3, 8, 6)
Binary trees

Intuition: each element has a value, and two children trees left and right
Everything less than value is on the left, everything greater is on the right.

Wander from the root to the leaves, insert when you hit a leaf.
Example: inserting (5, 9, 2, 4, 7, 3, 8, 6)
Binary trees

Intuition: each element has a value, and two children trees left and right
Everything less than value is on the left, everything greater is on the right.
Wander from the root to the leaves, insert when you hit a leaf.
Example: inserting (5, 9, 2, 4, 7, 3, 8, 6)
Binary trees

Intuition: each element has a value, and two children trees left and right
Everything less than value is on the left, everything greater is on the right.

Wander from the root to the leaves, insert when you hit a leaf.
Example: inserting (5, 9, 2, 4, 7, 3, 8, 6)
Binary trees

Intuition: each element has a value, and two children trees left and right
Everything less than value is on the left, everything greater is on the right.

Wander from the root to the leaves, insert when you hit a leaf.
Example: inserting (5, 9, 2, 4, 7, 3, 8, 6)
No mutation!
What’s a functional tree like?

Functional trees look a lot like our functional list code, e.g.:

```java
ITree<String> emptyTree = Tree.Empty.create();
ITree<String> tree1 = emptyTree.insert("Alice")
    .insert("Bob")
    .insert("Charlie")
    .insert("Dorothy")
    .insert("Eve");

ITree<String> tree2 = tree1.remove("Alice");
```

In this example, tree2 is tree1 without "Alice".

Just like our functional lists, we’ll allocate new nodes, but never mutate.
Binary tree insertion walkthrough

We have to allocate new nodes as we work our way down

$O(\log n)$ allocations as we go

Example: inserting (6)
Binary tree insertion walkthrough

We have to allocate new nodes as we work our way down

$O(\log n)$ allocations as we go

Example: inserting (6)
Binary tree insertion walkthrough

We have to allocate new nodes as we work our way down

$O(\log n)$ allocations as we go

Example: inserting (6)
Binary tree insertion walkthrough

We have to allocate new nodes as we work our way down $O(\log n)$ allocations as we go

Example: inserting (6)
Binary tree insertion walkthrough

We have to allocate new nodes as we work our way down

$O(\log n)$ allocations as we go

Example: inserting (6)
Overlapping trees?

The new tree allocated $O(\log n)$ new nodes, shares the rest.

If you drop the old pointer, just keep the new one, you get $O(\log n)$ garbage.
Overlapping trees?

The new tree allocated $O(\log n)$ new nodes, shares the rest.

If you drop the old pointer, just keep the new one, you get $O(\log n)$ garbage.
Overlapping trees?

The new tree allocated $O(\log n)$ new nodes, shares the rest.
If you drop the old pointer, just keep the new one, you get $O(\log n)$ garbage.
Overlapping trees?

The new tree allocated $O(\log n)$ new nodes, shares the rest.
If you drop the old pointer, just keep the new one, you get $O(\log n)$ garbage.

Modern GC performance: $O(\text{live data})$. Short-lived garbage is cheap.
public interface ITree<T extends Comparable<T>> {
    /**
     * returns the number of elements in the tree
     */
    int size();

    /**
     * returns whether or not the current tree is empty
     */
    boolean empty();

    /**
     * returns a new tree equal to the current tree with the new element inserted into it
     */
    ITree<T> insert(T newbie);

    /**
     * returns a new tree with all elements greater than the floor value, either inclusive or exclusive
     */
    ITree<T> greaterThan(T floor, boolean inclusive);

    /**
     * returns a new tree equivalent to the original, but absent the value if it's present
     */
    ITree<T> remove(T value);
public interface ITree<T extends Comparable<T>> {
    /**
     * returns the number of elements in the tree
     */
    int size();

    /**
     * returns whether or not the current tree is empty
     */
    boolean empty();

    /**
     * returns a new tree equal to the current tree with the new element inserted into it
     */
    ITree<T> insert(T newbie);

    /**
     * returns a new tree with all elements greater than the floor value, either inclusive or exclusive
     */
    ITree<T> greaterThan(T floor, boolean inclusive);

    /**
     * returns a new tree equivalent to the original, but absent the value if it's present
     */
    ITree<T> remove(T value);
}

Type constraint: T must implement the Comparable<T> interface
A tree is:
A “comparable” value
A left tree of values “less than” this value
A right tree of values “greater than” this value

Or:
An empty tree

```java
public class Tree<T extends Comparable<T>> implements ITree<T> {
    private final ITree<T> left, right;
    private final T value;

    // external tree users: don't use this; insert to an empty tree instead
    Tree(T value, ITree<T> left, ITree<T> right) {
        this.left = left;
        this.right = right;
        this.value = value;
    }
```
A tree is:
A “comparable” value
A left tree of values “less than” this value
A right tree of values “greater than” this value

Or:
An empty tree

```java
public class Tree<T extends Comparable<T>> implements ITree<T> {
    private final ITree<T> left, right;
    private final T value;

    // external tree users: don't use this; insert to an empty tree instead
    Tree(T value, ITree<T> left, ITree<T> right) {
        this.left = left;
        this.right = right;
        this.value = value;
    }
}
```

class Tree<T> must implement ITree<T>
**Tree: Data definition**

**A tree is:**
- A “comparable” value
- A left tree of values “less than” this value
- A right tree of values “greater than” this value

**Or:**
- An empty tree

```java
public class Tree<
    T extends Comparable<T>> implements ITree<T> {
    private final ITree<T> left, right;
    private final T value;

    // external tree users: don't use this; insert to an empty tree instead
    Tree(T value, ITree<T> left, ITree<T> right) {
        this.left = left;
        this.right = right;
        this.value = value;
    }
}
```
public class Tree<T extends Comparable<T>> implements ITree<T> {
    public ITree<T> insert(T newbie) {
        int comparison = newbie.compareTo(value);
        if (comparison < 0)
            return new Tree<>(value, left.insert(newbie), right);
        if (comparison > 0)
            return new Tree<>(value, left, right.insert(newbie));

        // if the newbie is exactly the same as what’s there, then we’re done
        if (this.value == newbie)
            return this;

        // This is a curious case. We have two different objects that have the same
        // value, so they’re “equal” but not the same. We’ll treat this as an update, which
        // will be useful later on.
        return new Tree<>(newbie, left, right);
    }

    ...

    public static class Empty<T extends Comparable<T>> extends Tree<T> {
        public ITree<T> insert(T value) {
            return new Tree<>(value, this, this);
        }
    }
}
public class Tree<T extends Comparable<T>> implements ITree<T> {
    public ITree<T> insert(T newbie) {
        int comparison = newbie.compareTo(value);
        if (comparison > 0)
            return new Tree<>(value, left, right.insert(newbie));
        if (comparison < 0)
            return new Tree<>(value, left.insert(newbie), right);
        // if the newbie is exactly the
        if (this.value == newbie)
            return this;
        // This is a curious case. We have two different objects that have the same
        // value, so they’re “equal” but not the same. We’ll treat this as an update, which
        // will be useful later on.
        return new Tree<>(newbie, left, right.insert(newbie));
    }
}

public static class Empty<T extends Comparable<T>> extends Tree<T> {
    public ITree<T> insert(T value) {
        return new Tree<>(value, this, this);
    }
}
public class Tree<T extends Comparable<T>> implements ITree<T> {
    public ITree<T> insert(T newbie) {
        int comparison = newbie.compareTo(value);
        if (comparison < 0)
            return new Tree<>(value, left.insert(newbie), right);
        if (comparison > 0)
            return new Tree<>(value, left, right.insert(newbie));
        // if the newbie is exactly the same as what’s there, then we’re done
        if (this.value == newbie)
            return this;
        // This is a curious case. We have two different objects that have the same
        // value, so they’re “equal” but not the same. We’ll treat this as an update, which
        // will be useful later on.
        return new Tree<>(newbie, left, right);
    }
}

public static class Empty<T extends Comparable<T>> extends Tree<T> {
    public ITree<T> insert(T value) {
        return new Tree<>(value, this, this);
    }
}
Tree insertion

```java
class Tree<T extends Comparable<T>> implements ITree<T> {
    public ITree<T> insert(T newbie) {
        int comparison = newbie.compareTo(value);
        if (comparison < 0)
            return new Tree<>(value, left.insert(newbie), right);
        if (comparison > 0)
            return new Tree<>(value, left, right.insert(newbie));
        // if the newbie is exactly the same as what’s there, then we’re done
        if (this.value == newbie)
            return this;
        // This is a curious case. We have two different objects that have the same
        // value, so they’re “equal” but not the same. We’ll treat this as an update, which
        // will be useful later on.
        return new Tree<>(newbie, left, right);
    }
}
```

insertion into an empty tree gives a tree with empty left/right

```java
class Empty<T extends Comparable<T>> extends Tree<T> {
    public ITree<T> insert(T value) {
        return new Tree<>(value, this, this);
    }
}
```
Tree insertion

```java
public class Tree<T extends Comparable<T>> implements ITree<T> {
    public ITree<T> insert(T newbie) {
        int comparison = newbie.compareTo(value);
        if (comparison < 0)
            return new Tree<>(value, left.insert(newbie), right);
        if (comparison > 0)
            return new Tree<>(value, left, right.insert(newbie));

        // if the newbie is exactly the same as what’s there, then we’re done
        if (this.value == newbie)
            return this;

        // This is a curious case. We have two different objects that have the same
        // value, so they’re “equal” but not the same. We’ll treat this as an update, which
        // will be useful later on.
        return new Tree<>(newbie, left, right);
    }
}
...

public static class Empty<T extends Comparable<T>> extends Tree<T> {
    public ITree<T> insert(T value) {
        return new Tree<>(value, this, this);
    }
}
```

The general pattern: return a new Tree, different arguments, sometimes recursive.
public interface ITree<T extends Comparable<T>> {
    default ITree<T> insertList(IList<? extends T> values) {
        return values.foldl((tree, elem) -> tree.insert(elem), this);
    }
}

public class Tree<T extends Comparable<T>> implements ITree<T> {
    @SafeVarargs
    public static <T extends Comparable<T>> ITree<T> of(T... values) {
        ITree<T> emptyTree = Tree.Empty.create();
        return emptyTree.insertList(LazyList.fromArray(values));
    }
}

ITree<Integer> tree = Tree.of(5, 27, 3, 1, 12);
public interface ITree<T extends Comparable<T>> {
    default ITree<T> insertList(IList<? extends T> values) {
        return values.foldl((tree, elem) -> tree.insert(elem), this);
    }
}

public class Tree<T extends Comparable<T>> implements ITree<T> {
    @SafeVarargs
    public static <T extends Comparable<T>> ITree<T> of(T... values) {
        ITree<T> emptyTree = Tree.Empty.create();
        return emptyTree.insertList(LazyList.fromArray(values));
    }
}

ITree<Integer> tree = Tree.of(5, 27, 3, 1, 12);

Goal: Have a cool varargs way of building a tree.
public interface ITree<T extends Comparable<T>> {
    default ITree<T> insertList(IList<? extends T> values) {
        return values.foldl((tree, elem)->tree.insert(elem), this);
    }
}

public class Tree<T extends Comparable<T>> implements ITree<T> {
    @SafeVarargs
    public static <T extends Comparable<T>> ITree<T> of(T... values) {
        ITree<T> emptyTree = Tree.Empty.create();
        return emptyTree.insertList(LazyList.fromArray(values));
    }
}

ITree<Integer> tree = Tree.of(5,27,3,1,12);

Same as we did with lists:
public interface ITree<T extends Comparable<T>> {
    default ITree<T> insertList(IList<? extends T> values) {
        return values.foldl((tree, elem) -> tree.insert(elem), this);
    }
}

...}

public class Tree<T extends Comparable<T>> implements ITree<T> {
    @SafeVarargs
    public static <T extends Comparable<T>> ITree<T> of(T... values) {
        ITree<T> emptyTree = Tree.Empty.create();
        return emptyTree.insertList(LazyList.fromArray(values));
    }
}

...

ITree<Integer> tree = Tree.of(5, 27, 3, 1, 12);
**Trees + Lists + Fold-Left = Bulk Insertion**

```java
public interface ITree<? extends T extends Comparable<T>> {
    default ITree<T> insertList(IList<? extends T> values) {
        return values.foldl((tree, elem)->tree.insert(elem), this);
    }
}
```

```
public class Tree<T extends Comparable<T>> implements ITree<T> {
    @SafeVarargs
    public static <T extends Comparable<T>> ITree<T> of(T... values) {
        ITree<T> emptyTree = Tree.Empty.create();
        return emptyTree.insertList(LazyList.fromArray(values));
    }
}
```

```
ITree<Integer> tree = Tree.of(5,27,3,1,12);
```

Why in the interface? This works for *any* tree!
Trees + Lists + Fold-Left = Bulk Insertion

```java
public interface ITree<T extends Comparable<T>> {
    default ITree<T> insertList(IList<? extends T> values) {
        return values.foldl((tree, elem)->tree.insert(elem), this);
    }
}
```

```java
public class Tree<T extends Comparable<T>> implements ITree<T> {
    @SafeVarargs
    public static <T extends Comparable<T>> ITree<T> of(T... values) {
        ITree<T> emptyTree = Tree.Empty.create();
        return emptyTree.insertList(LazyList.fromArray(values));
    }
}
```

```java
ITree<Integer> tree = Tree.of(5,27,3,1,12);
```

It's okay if we're inserting subclasses of T.

More about generic type wildcards coming in week 6. For now, don't panic.
public interface ITree<T extends Comparable<T>> {
    default ITree<T> insertList(IList<? extends T> values) {
        return values.foldl((tree, elem)->tree.insert(elem), this);
    }
}
public interface ITree<T extends Comparable<T>> {
    default ITree<T> insertList(IList<? extends T> values) {
        return values.foldl((tree, elem)->tree.insert(elem), this);
    }
}
public interface ITree<T extends Comparable<T>> {
    default ITree<T> insertList(IList<? extends T> values) {
        return values.foldl((tree, elem) -> tree.insert(elem), this);
    }
}
public interface ITree<T extends Comparable<T>> {
    default ITree<T> insertList(IList<? extends T> values) {
        return values.foldl((tree, elem) -> tree.insert(elem), this);
    }
}
public interface ITree<T extends Comparable<T>> {
    default ITree<T> insertList(IList<? extends T> values) {
        return values.foldl((tree, elem) -> tree.insert(elem), this);
    }
}

Equivalent statements:
ITree<Integer> tree = Tree.of(5, 27, 3, 1, 12);
ITree<Integer> tree = emptyTree.insert(5).insert(27).insert(3).insert(1).insert(12);

Runtime: O(n log n)
Digression: performance cost of functional programming

“Beautiful” functional code

```java
default ITree<T> insertList(IList<? extends T> values) {
    return values.foldl((tree, elem)->tree.insert(elem), this);
}
```

“Traditional” iterative code

```java
ITree<Integer> tree = Tree.Empty.create();
for(IList<Integer> x = values; !x.empty(); x = x.tail())
    tree = tree.insert(x.head());
```

Measured performance (μs per insert, JDK8u51, best of 5 runs)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree insert from list</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(10K random integers)</td>
<td>0.373 μs</td>
<td>0.582 μs</td>
<td>0.192 μs</td>
<td>0.310 μs</td>
</tr>
<tr>
<td>Tree insert from list</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1M random integers)</td>
<td>4.228 μs</td>
<td>1.674 μs</td>
<td>1.116 μs</td>
<td>0.953 μs</td>
</tr>
</tbody>
</table>
Digression: performance cost of functional programming

“Beautiful” functional code

```java
default ITree<T> insertList(IList<T> values) {
    return values.foldl((tree, elem) -> tree.insert(elem), this);
}
```

“Traditional” iterative code

```java
ITree<Integer> tree = Tree.Empty.create();
for (IList<Integer> x = values; !x.empty(); x = x.tail())
    tree = tree.insert(x.head());
```

Measured performance (μs per insert, JDK8u51, best of 5 runs)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree insert from list</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(10K random integers)</td>
<td>0.373 μs</td>
<td>0.582 μs</td>
<td>0.192 μs</td>
<td>0.310 μs</td>
</tr>
<tr>
<td>(1M random integers)</td>
<td>4.228 μs</td>
<td>1.674 μs</td>
<td>1.116 μs</td>
<td>0.953 μs</td>
</tr>
</tbody>
</table>

MacBook Air: 1.6GHz Intel Core i5 (dual core)
MacPro: 3.5GHz Intel Xeon (6 core)

Performance difference: 2-4x (single core)
"Beautiful" functional code

```java
default ITree<T> insertList(IList<? extends T> values) {
    return values.foldl((tree, elem)->tree.insert(elem), this);
}
```

"Traditional" iterative code

```java
ITree<Integer> tree = Tree.Empty.create();
for(IList<Integer> x = values; !x.empty(); x = x.tail())
    tree = tree.insert(x.head());
```

Measured performance (μs per insert, JDK8u51, best of 5 runs)

<table>
<thead>
<tr>
<th></th>
<th>&quot;Beautiful&quot; 2011 MacBook Air</th>
<th>&quot;Traditional&quot; 2011 MacBook Air</th>
<th>&quot;Beautiful&quot; 2014 MacPro</th>
<th>&quot;Traditional&quot; 2014 MacPro</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree insert from list (10K random integers)</td>
<td>0.373 μs</td>
<td>0.582 μs</td>
<td>0.192 μs</td>
<td>0.310 μs</td>
</tr>
<tr>
<td>Tree insert from list (1M random integers)</td>
<td>4.228 μs</td>
<td>1.674 μs</td>
<td>1.116 μs</td>
<td>0.953 μs</td>
</tr>
</tbody>
</table>

Big-O always matters. Beyond that, it’s hard to predict.
On Wednesday, we’ll introduce treaps

Kinda like trees, but balanced; no worst-case $O(n)$ behavior

Implementing treaps will be your project this week.
Sets & Maps
Moving onward: Sets

A set is something like an unordered list
We want efficient ways to test membership, add things, take union and intersection, etc.

```java
public interface ISet<T extends Comparable<T>> {
    /**
     * Returns a new set, adding the additional value. If the
     * value is already present, the prior value is replaced.
     */
    ISet<T> add(T value);

    /**
     * Returns a new set, adding the additional set. If the
     * value is already present, the prior value is "removed"
     * and the "merged" value, with mergeOp, is inserted.
     */
    ISet<T> merge(T value, BinaryOperator<T> mergeOp);

    /**
     * returns a new set, without the value, if present.
     * If the value is absent, this returns the same set.
     */
    ISet<T> remove(T value);
```
There are many kinds of equality in Java
Reference equality ($x == y$): do they point to the same exact object in memory
`Object.equals(Object)`: do they have the “same value”
`Comparable.compareTo()`: provides ordering, but (hopefully) compatible with `equals`

We’ll internally use `compareTo()`
Requiring ordering allows us to use trees for efficient storage
And useful methods like “greaterThan” to query a range of values in the set

But what if two objects are “equal” but differ in some meaningful way?
Solution 1: when “adding” to the set, replace the old value with the new one
Solution 2: keep both values
Solution 3: “merge” the values
Implementation: TreapSet

Coding tactic: delegate to an internal treap

```java
public class TreapSet<T extends Comparable<T>> implements ISet<T> {
    private final ITree<T> treap;

    // not for external use; start from the empty TreapSet instead
    TreapSet(ITree<T> treap) {
        this.treap = treap;
    }

    @Override
    public ISet<T> add(T value) {
        return new TreapSet<>(treap.insert(value));
    }
}
```
Delegation vs. inheritance

Why not have TreapSet extend Treap?
A “set” is not the same thing as a “tree”
A “set” has operations that a tree might not (union, intersection, etc.)

General rule:
Use inheritance if you’re specializing something (e.g., an empty tree is a specialized form of a regular tree)
- Same interface.
- Same behaviors (from a client’s perspective, anyway).

Use delegation if you’re using one thing to help you build something else (e.g., there are lists inside a functional queue)
- Different interfaces.
- Different behaviors.
One step further: Maps

A map is a (mathematical) function from “keys” to “values”
Not just a set of key/value pairs, because each key can only occur once

```java
public interface IMap<K extends Comparable<K>, V> {
    /**
     * Returns a new map, adding the additional key/value pair. If the
     * key is already present, the prior value for that key is replaced.
     */
    default IMap<K, V> add(K key, V value) {
        return add(new KeyValue<>(key, value));
    }

    /**
     * Returns a new map, adding the additional key/value pair. If the
     * key is already present, the prior value for that key is merged
     * with mergeOp and the new value.
     */
    IMap<K, V> merge(K key, V value, BinaryOperator<V> mergeOp);
}
```
TreapMap: delegating to a TreapSet

We need to worry about equality again
Solution: KeyValue pairs only test equality on the keys, not on the values
So, if we’re tracking age of people {“Alice”, 10} and {“Alice”, 15} are “equal”

```java
public class KeyValue<K extends Comparable<K>, V> implements Comparable<KeyValue<K, V>> {
    private final K key;
    private final V value;

    public int compareTo(KeyValue<K, V> other) {
        return this.key.compareTo(other.key);
    }
}
```

With this definition of “equal”, we can now store maps in a set, e.g.:

```java
public class TreapMap<K extends Comparable<K>, V> implements IMap<K, V> {
    private final ISet<KeyValue<K, V>> set;

    public Optional<V> oget(K key) {
        // look up the key in the set (this will be "equal"), then extract the value
        return set.oget(new KeyValue<>(key, null)).map(kv->kv.getValue());
    }
}
```
Performance

We’re paying a significant penalty for our functional-style map

- edu.rice.tree.TreapMap: 1000000 inserts, 10000 queries; 2.486 μs per insert
- java.util.TreeMap: 1000000 inserts, 10000 queries; 0.829 μs per insert
- java.util.HashMap: 1000000 inserts, 10000 queries; 0.219 μs per insert

- edu.rice.tree.TreapMap: 10000 range queries, 1.990 μs per query
- java.util.TreeMap: 10000 range queries, 0.285 μs per query

But we’ll need this later (week 9, our adventure game)

Makes it cheap and easy to remember the entire history of the game.

Also, range queries on mutating trees are weird and flakey
If the underlying tree changes, the query results change as well.
Live coding

Tree query

Tree traversal (recursive + lazy list)

Tree range query