# CSE 143
# Lecture 25

Set ADT implementation; hashing

read 11.2

slides created by Marty Stepp
http://www.cs.washington.edu/143/

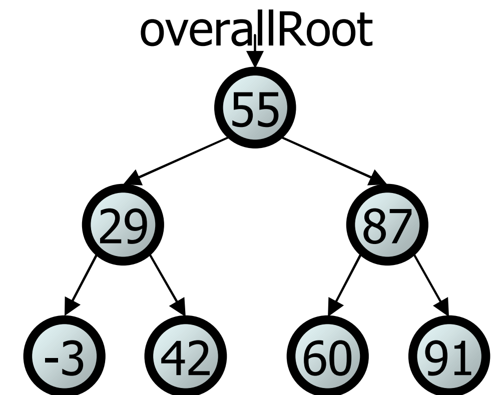
# IntTree as set

- We implemented a class `IntTree` to store a BST of `int`s:

- Our BST is essentially a set of integers.

  Operations we support:
  - `add`
  - `contains`
  - `remove` (not written in lecture)

    ...




  - Problems:
    - **The tree carries around a clunky extra node class.**
    - The tree can store only `int` elements, not any type of value.
    - There are other ways to implement a set. We should be able to treat different implementations of sets the same way.

# Tree node inner class

```
public class IntTreeSet {
    private IntTreeNode overallRoot;
    ...
    // inner (nested) class
    private class IntTreeNode {
        public int data;              // data stored at this node
        public IntTreeNode left;      // left subtree
        public IntTreeNode right;     // right subtree

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

        // Constructs leaf or branch with given data and links.
        public IntTreeNode(int d, IntTreeNode l, IntTreeNode r) {
            this.data = d;
            this.left = l;
            this.right = r;
        }
    }
}
```

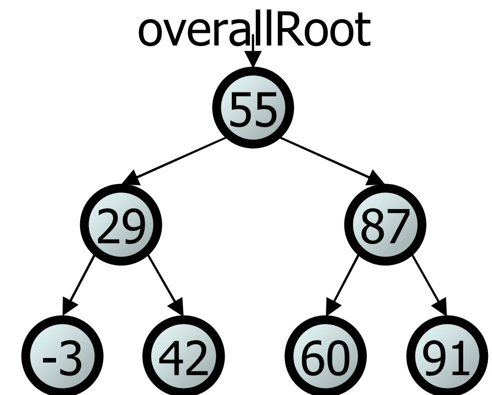
# `IntTree` as set

- We implemented a class `IntTree` to store a BST of `int`s:

- Our BST is essentially a set of integers.

  Operations we support:
  - `add`
  - `contains`
  - `remove` (not written in lecture)

    ...




  - Problems:
    - The tree carries around a clunky extra node class.
    - **The tree can store only `int` elements, not any type of value.**
    - There are other ways to implement a set. We should be able to treat different implementations of sets the same way.

# Problem with generics

```
public class TreeSet<E> {
    ...
    // Recursive helper to search given subtree for given value.
    private boolean contains(IntTreeNode root, E value) {
        if (root == null) {
            return false;
        } else if (root.data == value) {
            return true;
        } else if (root.data > value) {  // too large; go left
            return contains(root.left, value);
        } else {                          // too small; go right
            return contains(root.right, value);
        }
    }
}
```

- You cannot use the `<` or `>` operator on objects. How to fix it?
- It still doesn't work if you write the following. Why not?

```
        } else if (root.data.compareTo(value) > 0) {
```

# Constrained type params.

```
public class name<Type extends Type2> {
    ...
}
```

- places a constraint on what type can be given by the client; client can supply only **Type2** or any of its subclasses

- **Type2** can be an interface (we don't write "implements")
  - any class that implements the interface can be supplied

- **Type2** can itself be parameterized if necessary (nested <>)

# Correct generic tree code

```
public class TreeSet<E extends Comparable<E>> {
    ...
    // Recursive helper to search given subtree for given value.
    private boolean contains(IntTreeNode root, E value) {
        if (root == null) {
            return false;
        } else if (root.data == value) {
            return true;
        } else if (root.data.compareTo(value) > 0) {
            return contains(root.left, value);
        } else {
            return contains(root.right, value);
        }
    }
}
```

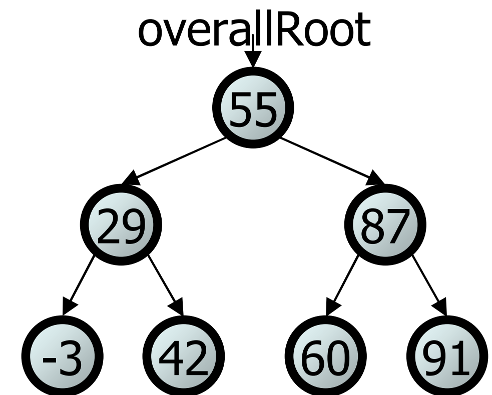
# `IntTree` as set

- We implemented a class `IntTree` to store a BST of `int`s:

- Our BST is essentially a set of integers.

  Operations we support:
  - `add`
  - `contains`
  - `remove` (not written in lecture)

    ...




  - Problems:
    - The tree carries around a clunky extra node class.
    - The tree can store only `int` elements, not any type of value.
    - **There are other ways to implement a set. We should be able to treat different implementations of sets the same way.**

# How to implement a set?

- Elements of a `TreeSet` (`IntTree`) are in BST sorted order.
  - We need this in order to add or search in O(log N) time.

- But it doesn't really matter what order the elements appear in a set, so long as they can be added and searched quickly.

- Consider the task of storing a set in an array.
  - What would make a good ordering for the elements?

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| value | 7 | 11 | 24 | 49 | 0 | 0 | 0 | 0 | 0 | 0 |

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| value | 0 | 11 | 0 | 0 | 24 | 0 | 0 | 7 | 0 | 49 |

# Hashing

- **hash**: To map a value to an integer index.
  - **hash table**: An array that stores elements via hashing.

- **hash function**: An algorithm that maps values to indexes.

$$HF(I) \rightarrow I \mathbin{\%} length$$

```
set.add(11);       // 11 % 10 == 1
set.add(49);       // 49 % 10 == 9
set.add(24);       // 24 % 10 == 4
set.add(7);        //  7 % 10 == 7
```

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| value | 0 | 11 | 0 | 0 | 24 | 0 | 0 | 7 | 0 | 49 |

# Efficiency of hashing

```
public static int HF(int i) {        // hash function
    return Math.abs(i) % elementData.length;
}
```

- Add: simply set `elementData[HF(i)] = i;`
- Search: check if `elementData[HF(i)] == i`
- Remove: set `elementData[HF(i)] = 0;`

- What is the runtime of `add`, `contains`, and `remove`?
  - **O(1)!** OMGWTFBBQFAST

- Are there any problems with this approach?

# Collisions

- **collision**: When a hash function maps two or more elements to the same index.

```
set.add(11);
set.add(49);
set.add(24);
set.add(7);
set.add(54);   // collides with 24!
```

- **collision resolution**: An algorithm for fixing collisions.

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| value | 0 | 11 | 0 | 0 | 24 | 0 | 0 | 7 | 0 | 49 |

# Probing

- **probing**: Resolving a collision by moving to another index.
  - **linear probing**: Moves to the next index.

```
set.add(11);
set.add(49);
set.add(24);
set.add(7);
set.add(54);   // collides with 24
```

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| value | 0 | 11 | 0 | 0 | 24 | **54** | 0 | 7 | 0 | 49 |

  - Is this a good approach?

# Clustering

- **clustering**: Clumps of elements at neighboring indexes.
  - slows down the hash table lookup; you must loop through them.

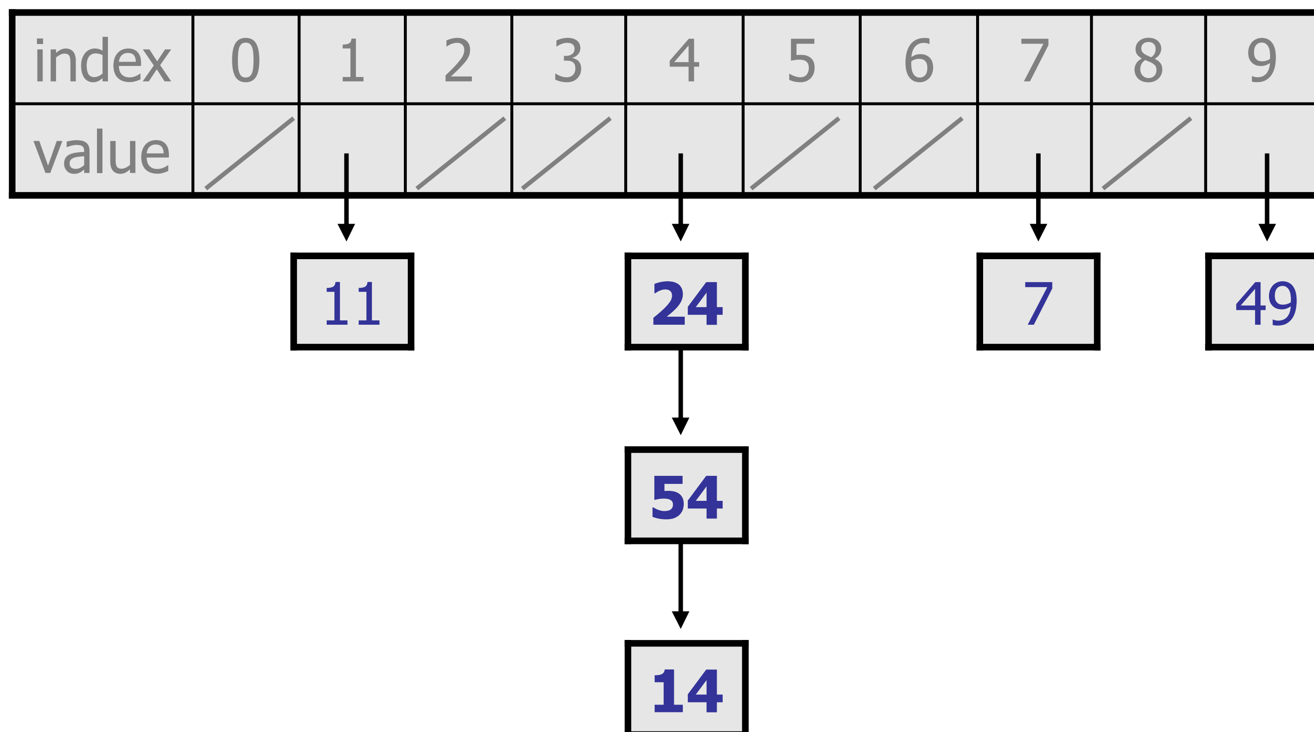
```
set.add(11);
set.add(49);
set.add(24);
set.add(7);
set.add(54);   // collides with 24
set.add(14);   // collides with 24, then 54
set.add(86);   // collides with 14, then 7
```

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| value | 0 | 11 | 0 | 0 | **24** | **54** | **14** | **7** | **86** | 49 |

- Now a lookup for 94 must look at 5 out of 10 total indexes.

# Chaining

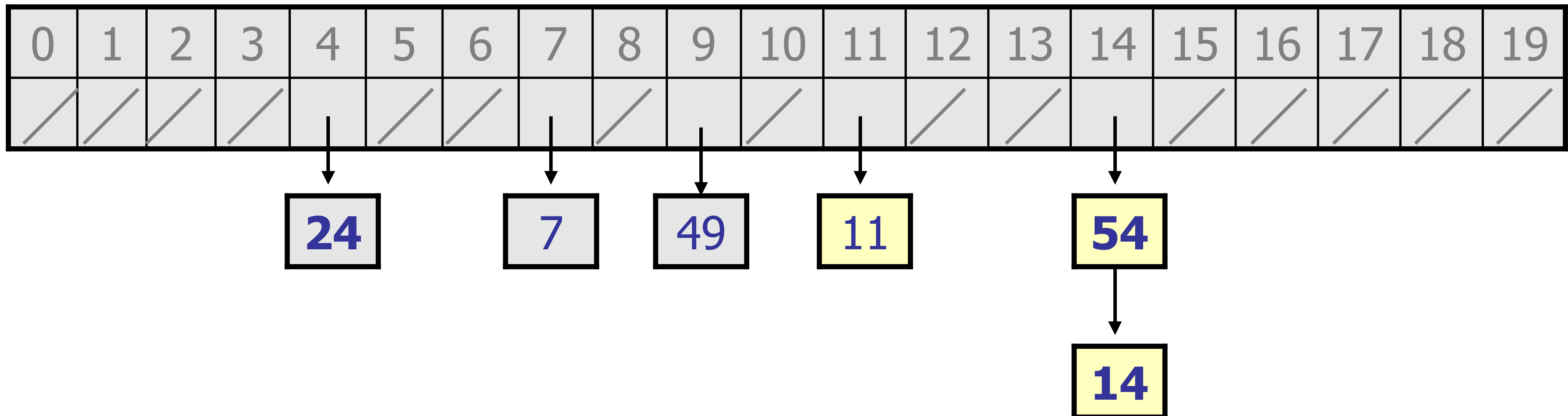
- **chaining**: Resolving collisions by storing a list at each index.
  - add/search/remove must traverse lists, but the lists are short
  - impossible to "run out" of indexes, unlike with probing



# Rehashing

- **rehash**: Growing to a larger array when the table is too full.
  – Cannot simply copy the old array to a new one. (Why not?)

- **load factor**: ratio of ( *# of elements* ) / ( *hash table length* )
  – many collections rehash when load factor ≅ .75
  – can use big prime numbers as hash table sizes to reduce collisions



# Hashing objects

- It is easy to hash an integer I (use index *I % length* ).
  - How can we hash other types of values (such as objects)?

- The `Object` class defines the following method:

```
public int hashCode()
```

  Returns an integer hash code for this object.

  - We can call `hashCode` on any object to find its preferred index.

- How is `hashCode` implemented?
  - Depends on the type of object and its state.
    - Example: a String's `hashCode` adds the ASCII values of its letters.
  - You can write your own `hashCode` methods in classes you write.

# Final hash set code

```
import java.util.*;    // for List, LinkedList

// All methods assume value != null;  does not rehash
public class HashSet<E> implements Set<E> {
    private static final int INITIAL_CAPACITY = 137;
    private List<E>[] elements;

    // constructs new empty set
    public HashSet() {
        elements = (List<E>[]) (new List[INITIAL_CAPACITY]);
    }

    // adds the given value to this hash set
    public void add(E value) {
        int index = hashFunction(value);
        if (elements[index] == null) {
            elements[index] = new LinkedList<E>();
        }
        elements[index].add(value);
    }

    // hashing function to convert objects to indexes
    private int hashFunction(E value) {
        return Math.abs(value.hashCode()) % elements.length;
    }
    ...
```

# Final hash set code 2

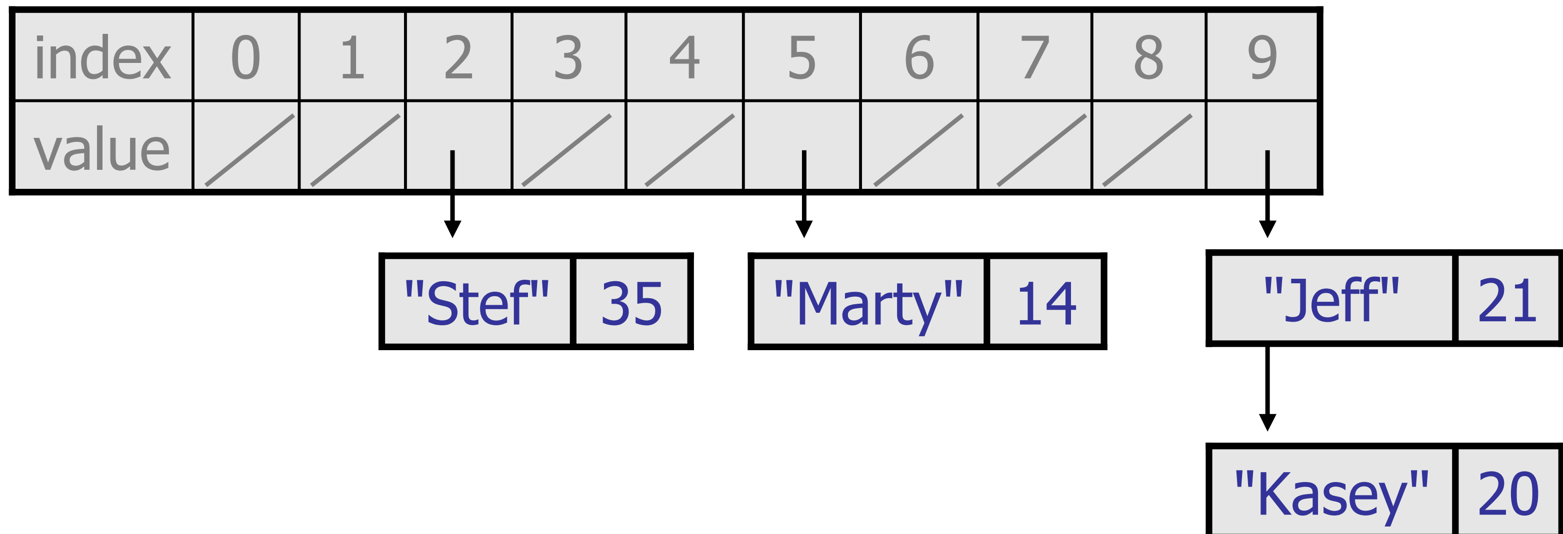
```
    ...
    // Returns true if this set contains the given value.
    public boolean contains(E value) {
        int index = hashFunction(value);
        return elements[index] != null &&
               elements[index].contains(value);
    }

    // Removes the given value from the set, if it exists.
    public void remove(E value) {
        int index = hashFunction(value);
        if (elements[index] != null) {
            elements[index].remove(value);
        }
    }
}
```

# Implementing maps

- A map is just a set where the lists store key/value pairs:

```
//           key      value
map.put("Marty", 14);
map.put("Jeff",  21);
map.put("Kasey", 20);
map.put("Stef",  35);
```




  - Instead of a `List<E>`, write an inner `Entry` class with `key` and `value` fields and make a `List<Entry>`