CS 10: Problem solving via Object Oriented Programming

Info Retrieval

ADT Overview

	List
Description	Keep items stored in order by index
Common use	• Grow to hold any number of items
Implementation options	• Linked list • Growing array
Java provided	• LinkedList • ArrayList

ADT Overview

	List	(Binary) Tree
Description	Keep items stored in order by index	Keep hierarchical relationship between nodes
Common use	• Grow to hold any number of items	• Find items quickly by Key • BST *generally* faster than List
Implementation options	• Linked list • Growing array	• BinaryTree • BST • 2-3-4 • Red-Black
Java provided	• LinkedList • ArrayList	

ADT Overview

	List	(Binary) Tree	Set
Description	Keep items stored in order by index	Keep hierarchical relationship between nodes	Keep an unordered set of objects
Common use	• Grow to hold any number of items	• Find items quickly by Key • BST *generally* faster than List	• Prevent duplicates
Implementation options	• Linked list • Growing array	• BinaryTree • BST • 2-3-4 • Red-Black	• List • Tree • Hash table
Java provided	• LinkedList • ArrayList		• TreeSet • HashSet

ADT Overview

	List	(Binary) Tree	Set	Map
Description	Keep items stored in order by index	Keep hierarchical relationship between nodes	Keep an unordered set of objects	Keep a set of Key/Value pairs
Common use	• Grow to hold any number of items	• Find items quickly by Key • BST *generally* faster than List	• Prevent duplicates	• Find items quickly by Key
Implementation options	• Linked list • Growing array	• BinaryTree • BST • 2-3-4 • Red-Black	• List • Tree • Hash table	• List • Tree • Hash table
Java provided	• LinkedList • ArrayList		• TreeSet • HashSet	• TreeMap • HashMap

Agenda

1. Set ADT
2. Map ADT
3. Reading from file/keyboard
4. Search

Key points:

1. **Sets are an unordered collection of items like the mathematical notion of a set**
2. **Sets prevent duplicates**
3. **Can be implemented with trees (Java provides a TreeSet)**

Sets are an unordered collection of items without duplicates

Set ADT

- Model for mathematical definition of a Set
- Like a List, but:
  - *Logically* unordered (no i^{th} item, can't set/get by index)
  - No duplicates allowed
- Operations:
  - `add(E e)` – adds `e` to Set if not already present
  - `contains(E e)` – returns true if `e` in Set, else false
  - `remove(E e)` – removes `e` from Set
  - `size()` – returns number of elements in Set
  - `isEmpty()` – true if no elements in Set, else false
  - `Iterator<E> iterator()` – returns iterator over Set

Sets start out empty

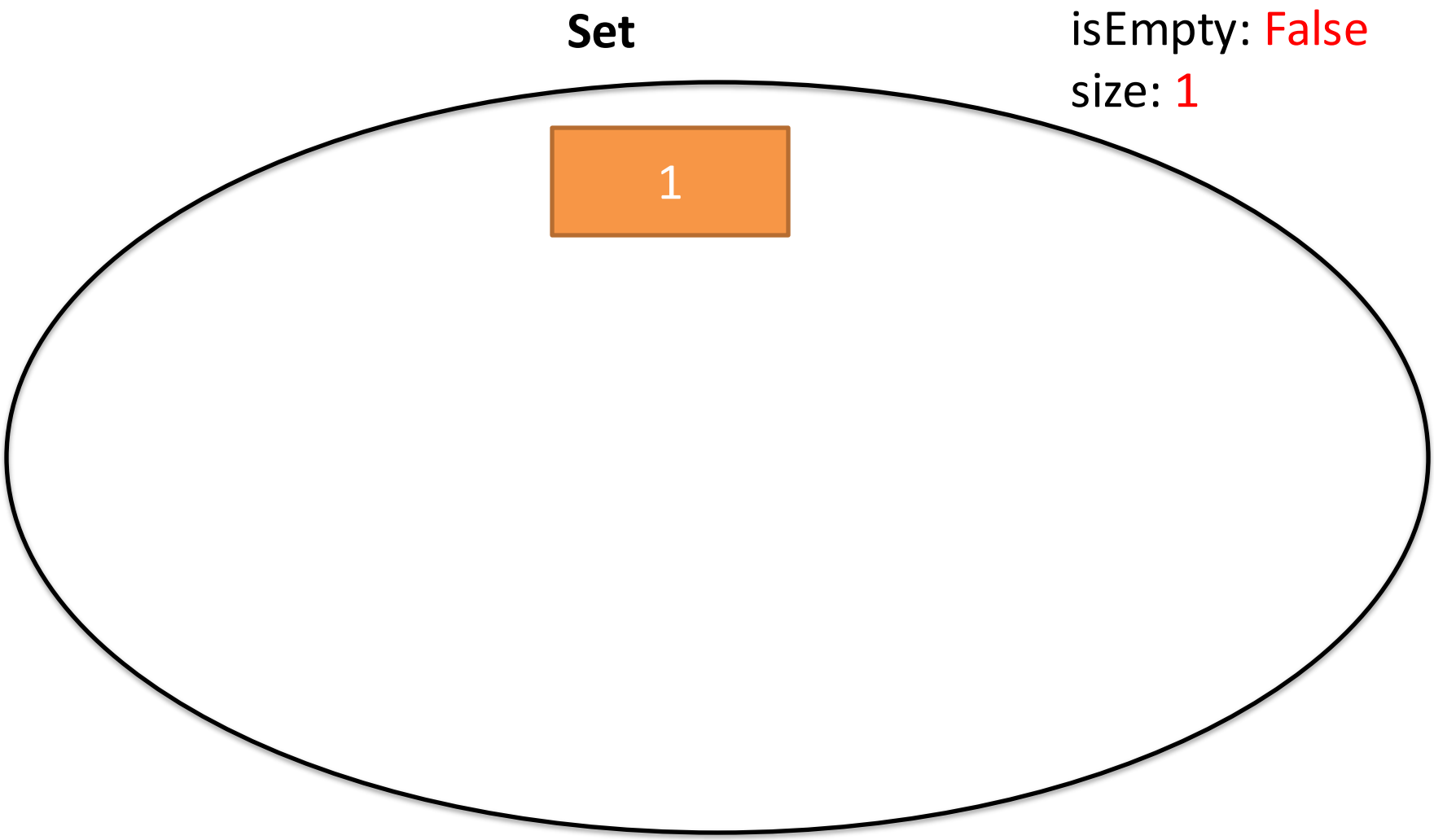
Initial state

Set

isEmpty: True
size: 0

First item added will always create a new entry in the Set (item can't be a duplicate)

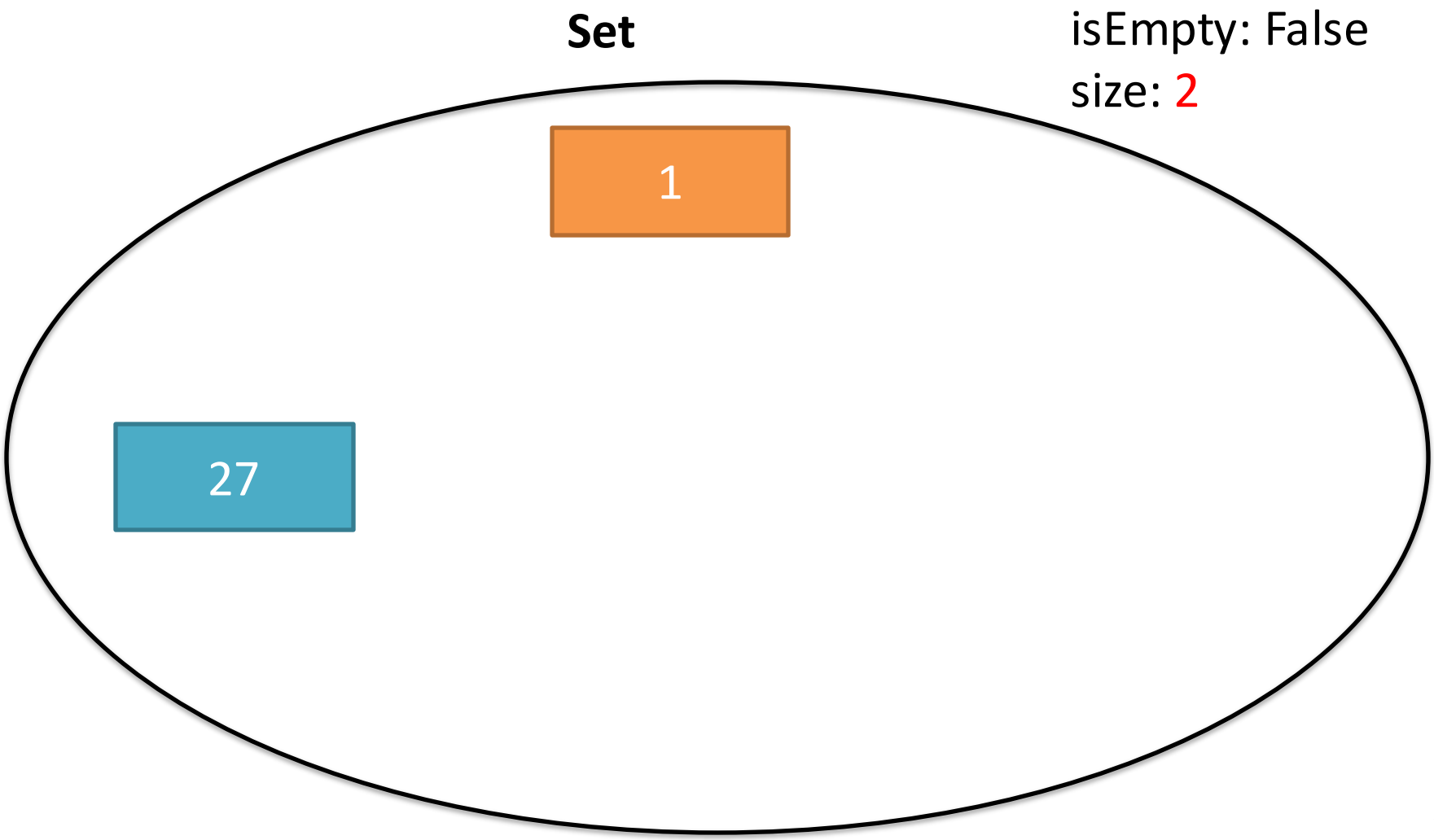
add(1)



isEmpty: False
size: 1

Can think of adding items to Set like adding items to "Bag of items" – no item ordering

add(27)

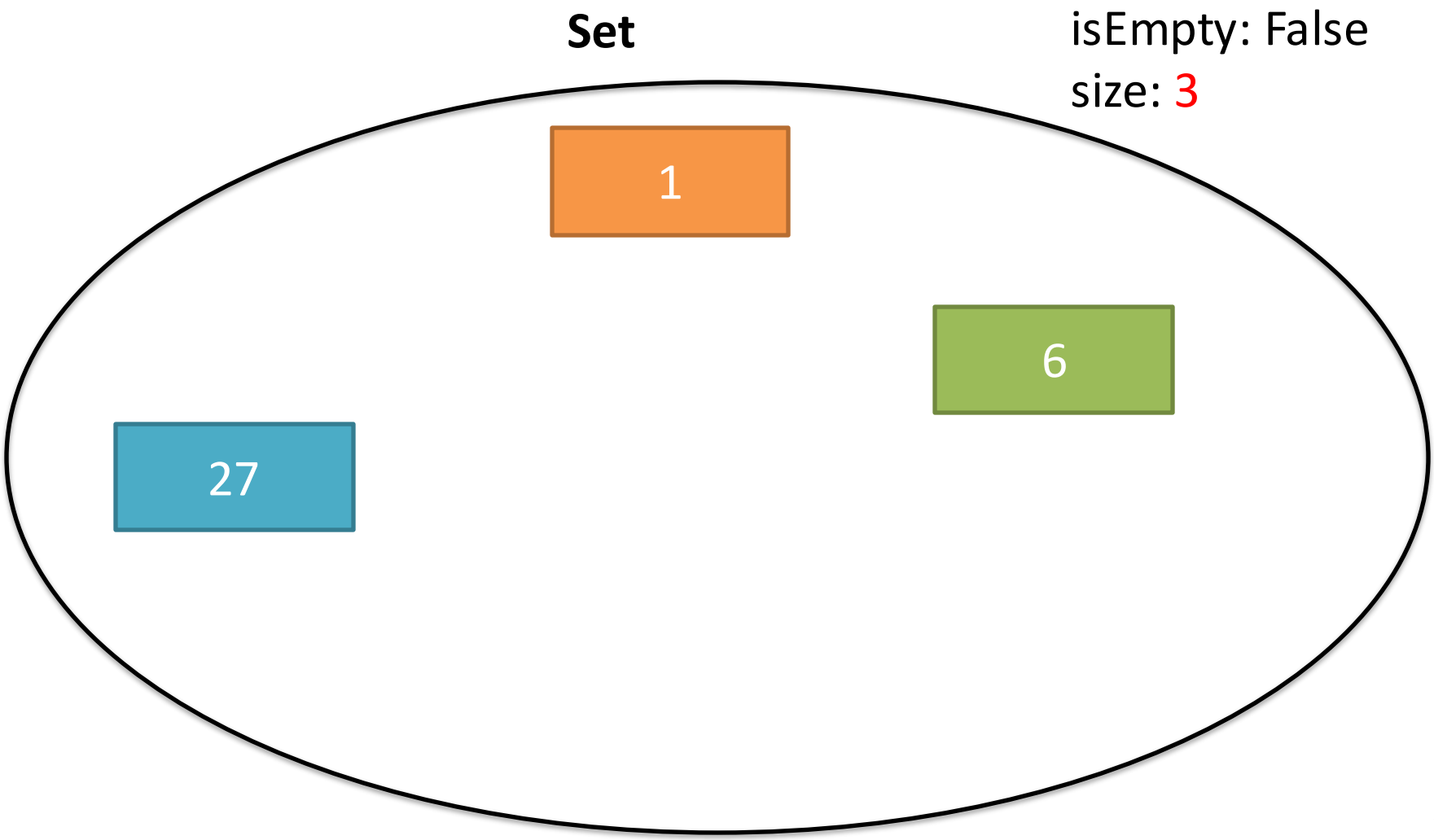


isEmpty: False

size: 2

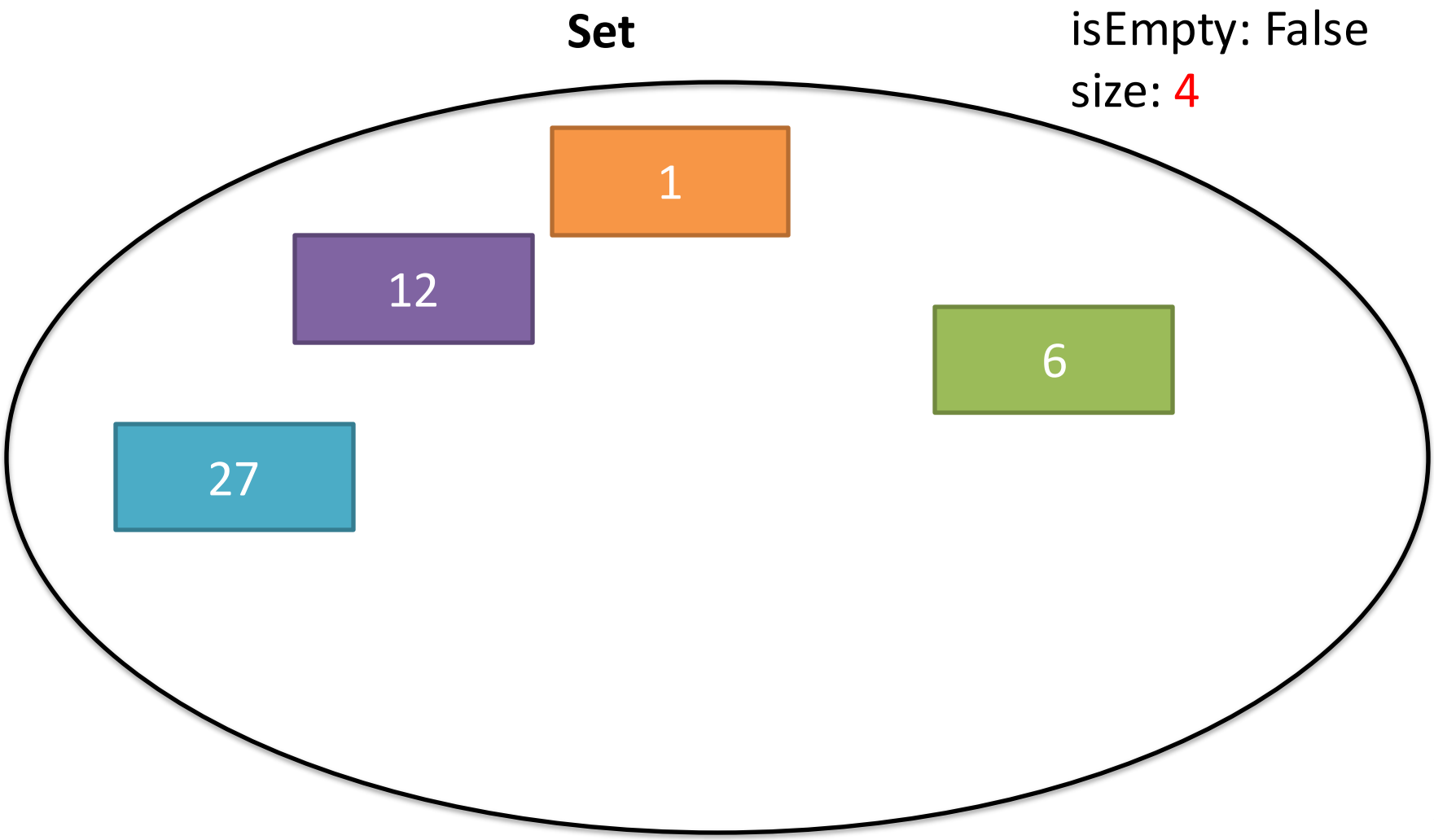
Can think of adding items to Set like adding items to "Bag of items" – no item ordering

add(6)



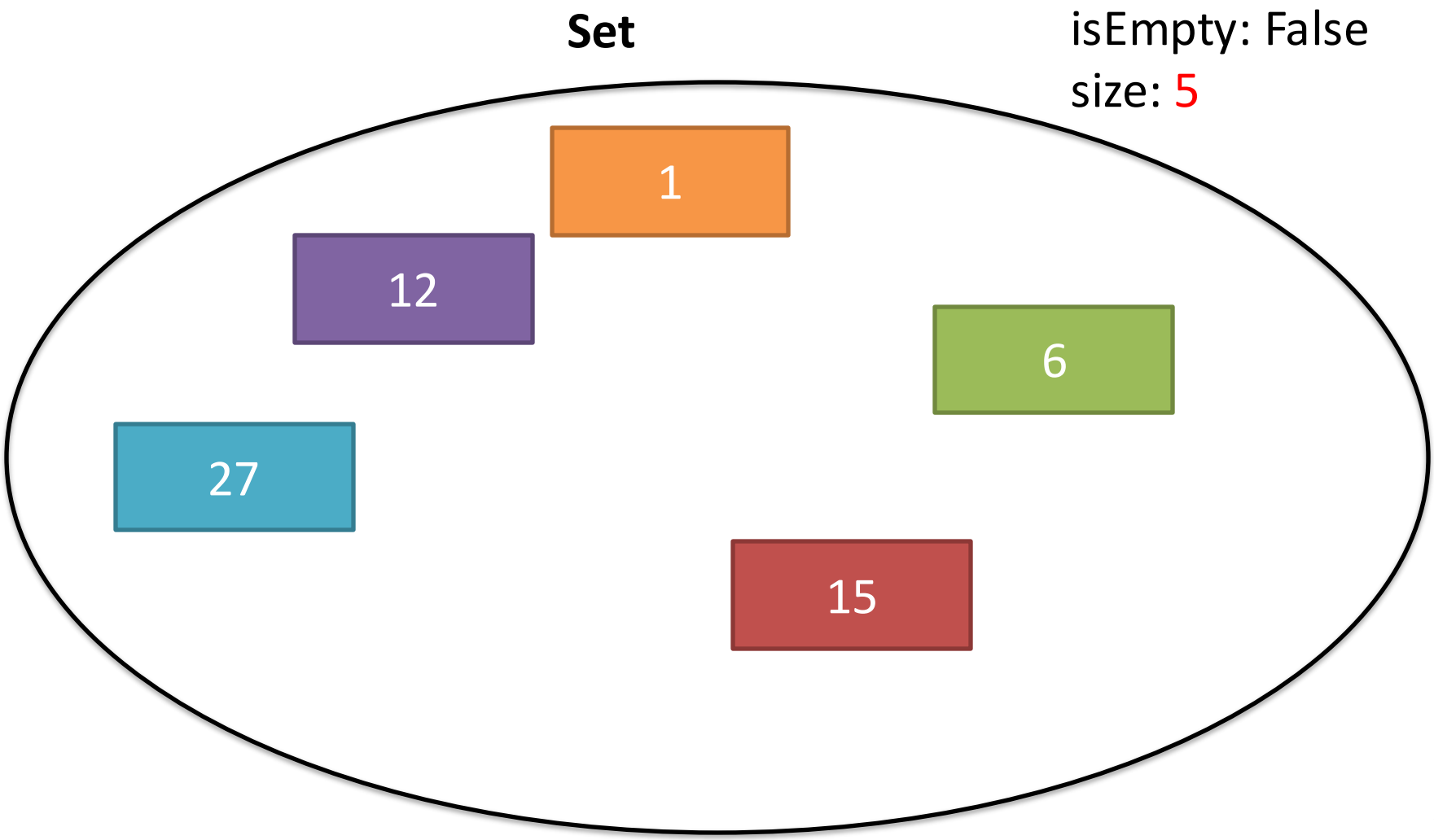
Can think of adding items to Set like adding items to "Bag of items" – no item ordering

add(12)



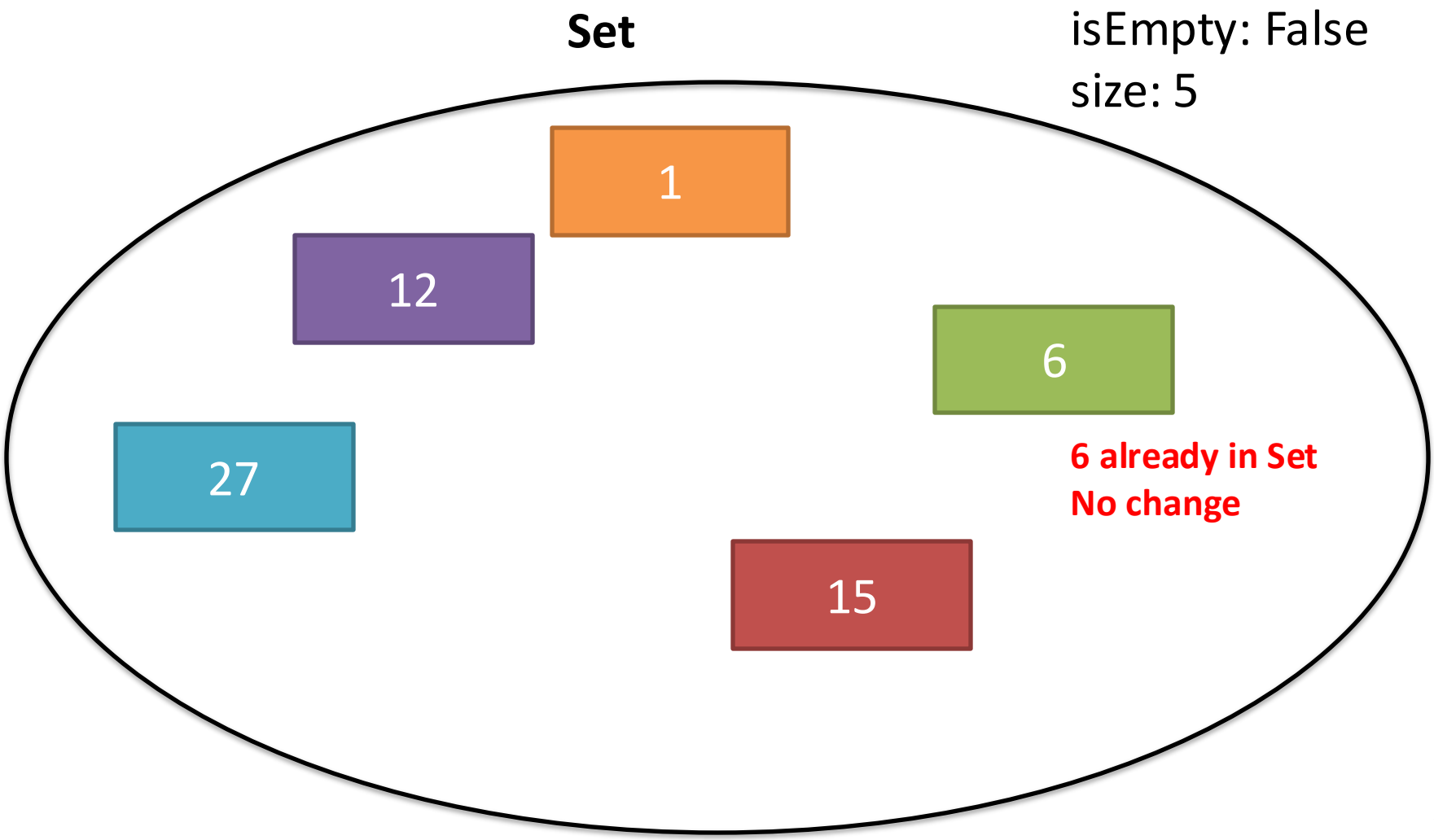
Can think of adding items to Set like adding items to "Bag of items" – no item ordering

add(15)



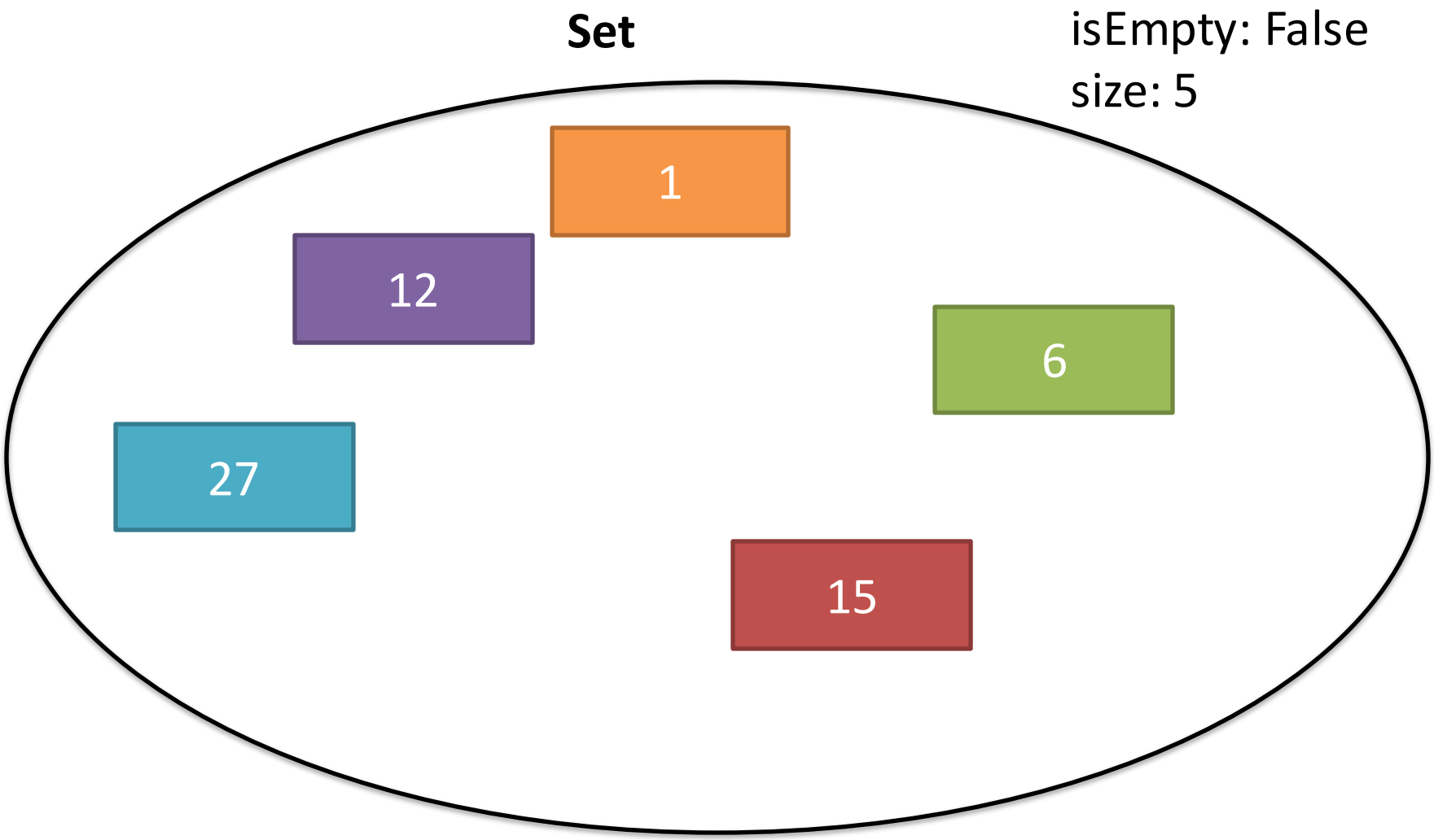
Adding an item that is already in the Set does not change the Set

add(6)



Items can be removed

remove(1)



Items can be removed

remove(1)

Set

isEmpty: False
size: 4

1 removed
size reduced

12

6

27

15

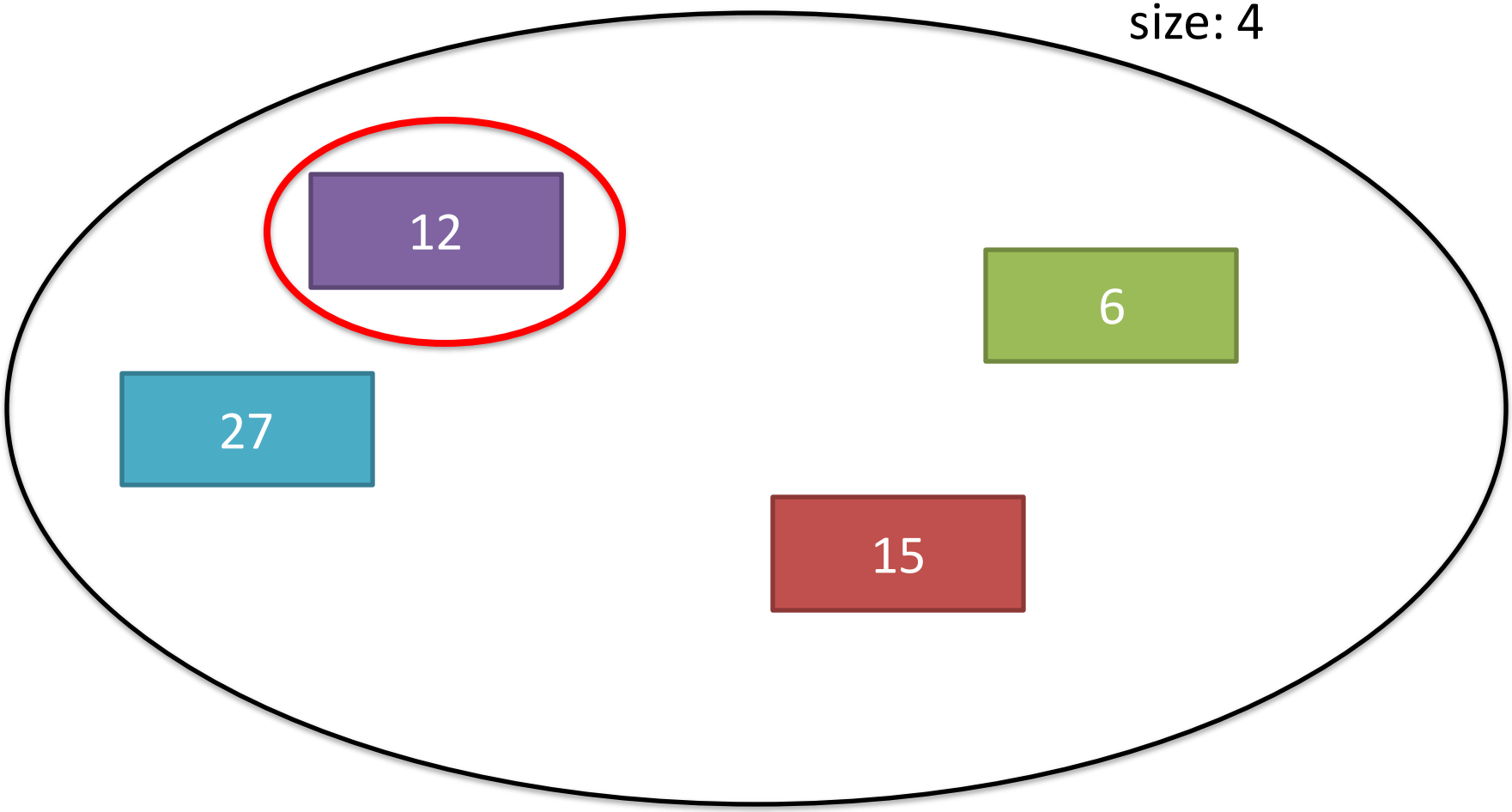
Can also check to see if item is in Set

contains(12)

True

Set

isEmpty: False
size: 4



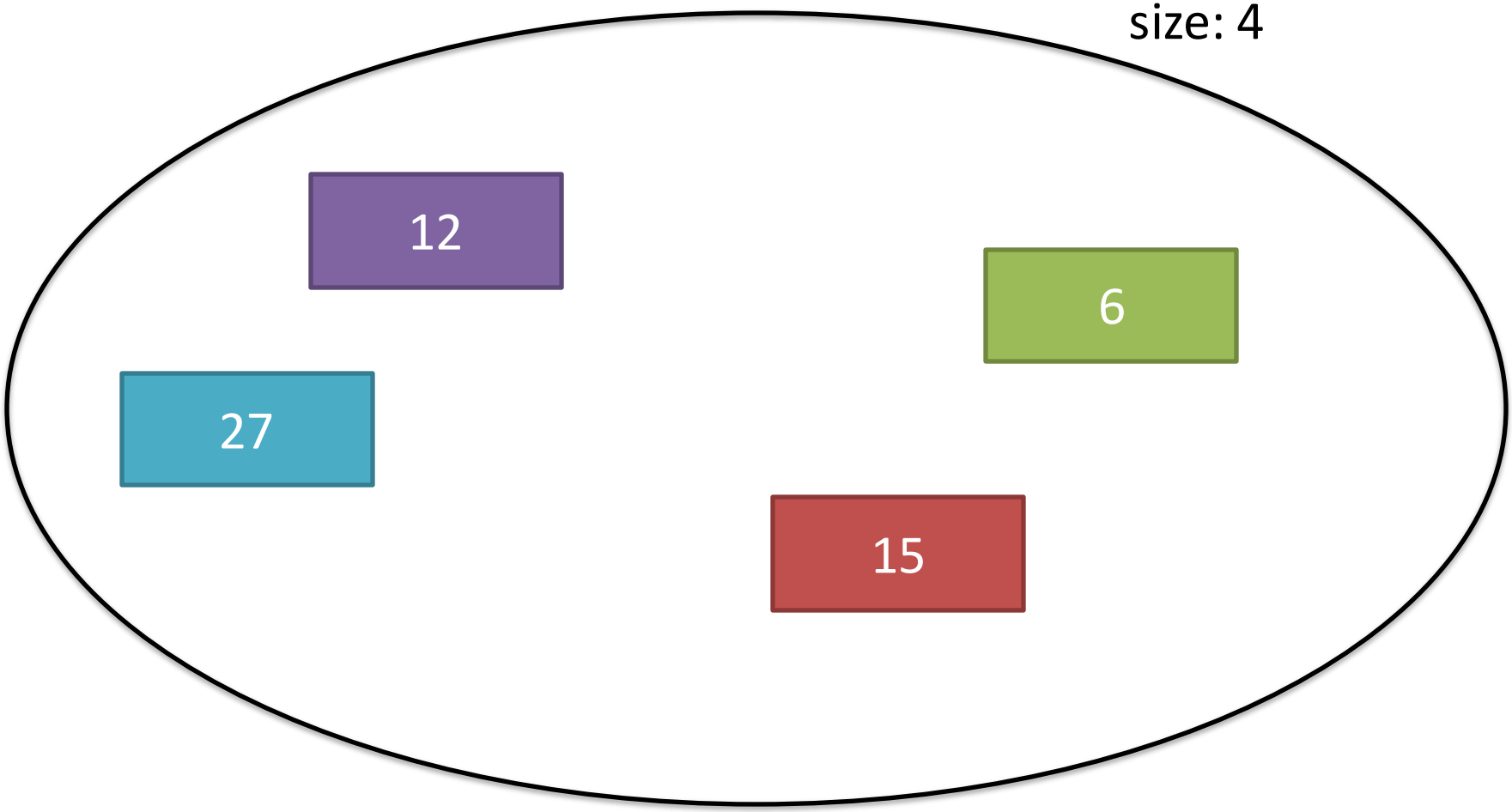
Can also check to see if item is in Set

contains(13)

False

Set

isEmpty: False
size: 4



Trees are one way to implement the Set ADT

Sets implemented with Trees

- Could implement as a List, but linear search time
- Trees are a natural way to think about implementation
- If the Set is implemented with a tree

Trees are one way to implement the Set ADT

Sets implemented with Trees

- Could implement as a List, but linear search time
- Trees are a natural way to think about implementation
- If the Set is implemented with a tree

Operation	Run-time	Notes
add(e)	O(h)	• Search for node until found or hit leaf • If not found, add new leaf (if found do nothing) • Might have to add node on longest path • Can't be more than h+1 checks

Trees are one way to implement the Set ADT

Sets implemented with Trees

- Could implement as a List, but linear search time
- Trees are a natural way to think about implementation
- If the Set is implemented with a tree

Operation	Run-time	Notes
add(e)	O(h)	• Search for node until found or hit leaf • If not found, add new leaf (if found do nothing) • Might have to add node on longest path • Can't be more than h+1 checks
contains(e)	O(h)	• Search for node until found or hit leaf • Might have to search longest path • Can't be more than h+1 checks

Trees are one way to implement the Set ADT

Sets implemented with Trees

- Could implement as a List, but linear search time
- Trees are a natural way to think about implementation
- If the Set is implemented with a tree

Operation	Run-time	Notes
add(e)	O(h)	• Search for node until found or hit leaf • If not found, add new leaf (if found do nothing) • Might have to add node on longest path • Can't be more than h+1 checks
contains(e)	O(h)	• Search for node until found or hit leaf • Might have to search longest path • Can't be more than h+1 checks
remove(e)	O(h)	• Traverse tree to find element, then delete it

Trees are one way to implement the Set ADT

Sets implemented with Trees

- Could implement as a List, but linear search time
- Trees are a natural way to think about implementation
- If the Set is implemented with a tree

Operation	Run-time	Notes
add(e)	O(h)	• Search for node until found or hit leaf • If not found, add new leaf (if found do nothing) • Might have to add node on longest path • Can't be more than h+1 checks
contains(e)	O(h)	• Search for node until found or hit leaf • Might have to search longest path • Can't be more than h+1 checks
remove(e)	O(h)	• Traverse tree to find element, then delete it

- Soon we will see another, more efficient way to implement a Set using a hash table

Can use a Set to easily identify the unique words in a body of text

Text from which to identify unique words

"Pretend that this string was loaded from a web page. We won't go to all that trouble here. This string contains multiple words. And multiple copies of multiple words. And multiple words with multiple copies. It is to be used as a test to demonstrate how sets work in removing redundancy by keeping only one copy of each thing. Is it very very redundant in having more than one copy of some words?"

Pseudocode

- Create Set with String as element
- Loop over each word in text
- Add to Set
- Print Set when done

Set <String>

- **Add each word in text to Set**
- **Duplicates not maintained**

Can use a Set to easily identify the unique words in a body of text

Text from which to identify unique words

"**Pretend** that this string was loaded from a web page. We won't go to all that trouble here. This string contains multiple words. And multiple copies of multiple words. And multiple words with multiple copies. It is to be used as a test to demonstrate how sets work in removing redundancy by keeping only one copy of each thing. Is it very very redundant in having more than one copy of some words?"

Pseudocode

- Create Set with String as element
- Loop over each word in text
- Add to Set
- Print Set when done

Set <String>

Pretend

Can use a Set to easily identify the unique words in a body of text

Text from which to identify unique words

"Pretend **that** this string was loaded from a web page. We won't go to all that trouble here. This string contains multiple words. And multiple copies of multiple words. And multiple words with multiple copies. It is to be used as a test to demonstrate how sets work in removing redundancy by keeping only one copy of each thing. Is it very very redundant in having more than one copy of some words?"

Pseudocode

- Create Set with String as element
- Loop over each word in text
- Add to Set
- Print Set when done

Set <String>

Pretend

that

Can use a Set to easily identify the unique words in a body of text

Text from which to identify unique words

"Pretend that this string was loaded from a web page. We won't go to all **that** trouble here. This string contains multiple words. And multiple copies of multiple words. And multiple words with multiple copies. It is to be used as a test to demonstrate how sets work in removing redundancy by keeping only one copy of each thing. Is it very very redundant in having more than one copy of some words?"

Pseudocode

- Create Set with String as element
- Loop over each word in text
- Add to Set
- Print Set when done

Set <String>

Pretend

that

this

string

was

loaded

…

- **"that" seen again**
- **Already in Set, so Set does not change**
- **At the end the Set will contain all the unique words in the text**

UniqueWords.java: Use a Set to easily identify the unique words in a body of text

```
10     public static void main(String[] args) {
11         String page = "Pretend that this string was loaded from a web page.  We
12                 + "won't go to all that trouble here.  This string contains mult
13                 + "words. And multiple copies of multiple words.  And multiple "
14                 + "words with multiple copies.  It is to be used as a test to "
15                 + "demonstrate how sets work in removing redundancy by keeping c
16         String[] allWords = page.split("[ .,?!]+"); // split on punctuation and
17
18         // Declare new Set to hold unique words
19         Set<String> uniqueWords = new TreeSet<String>();
20
21         // Loop over all the words split out of the string, adding to set
22         for (String s: allWords) {
23             uniqueWords.add(s.toLowerCase()); // Calling add() method for duplic
24         }
25
26         System.out.println(allWords.length + " words"); //note: this is not the
27         System.out.println(uniqueWords.size() + " unique words"); //this is the
28         System.out.println(uniqueWords); //print the unique words
29     }
30 }
```

Large amount of text simulates webpage

split() makes an array with entry for each word (including duplicates)

Java has Set implementation based on Red/Black Tree

Implements Set interface

Set elements are Strings here

Add all words to Set, discarding duplicates

No duplicate words

Print calls *toString* on TreeSet class

Why is output alphabetical? *toString* does In-order tree traversal!

Problems | Javadoc | Declaration | Console | Debug | Expressions | Error Log | Call Hierarchy

```
terminated> UniqueWords [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Jan 2, 2018, 6:59:33 AM)
[a, all, and, as, be, by, contains, copies, copy, demonstrate, each, from, go, having
```

Agenda

1. Set ADT
2. Map ADT
3. Reading from file/keyboard
4. Search

Key points:

1. **Maps look items up by Key and return Value**
2. **Python programmers, think dictionaries**
3. **Can be implemented with trees (Java provides a TreeMap)**

Map ADT associates Keys with Values

Map ADT

- Key is used to look up a Value (ex., student ID finds student record)
- Python programmers can think of Maps as Dictionaries
- Value could be an object (e.g., a person object or student record)
- Duplicate *Values* allowed, but not duplicate Keys

Operations:

- `containsKey(K key)` – true if `key` in Map, else false
- `containsValue(V value)` – true if *one or more* entries have `value`
- `get(K key)` – returns `value` for specified `key` or null otherwise
- `put(K key, V value)` – store `key`/`value` in Map; overwrite existing value if `key` found (NOTE: no `add` operation in Map ADT)
- `remove(K key)` – removes `key` from Map and returns `value`
- `keySet()` – returns `Set` of Keys in Map (which has iterator)
- `size()` – returns number of elements in Map
- `isEmpty()` – true if no elements in Map, else false

Like Sets, Maps initially start out empty

Map	
Key <StudentID>	**Value <Student Name>**

isEmpty: True
size: 0

Items are adding to a Map using *put(Key,Value)*

put(123, “Charlie”)

Map	
Key <StudentID>	**Value <Student Name>**
123	Charlie

isEmpty: False

size: 1

Items are adding to a Map using *put(Key,Value)*

put(987, “Alice”)

Map	
Key <StudentID>	**Value <Student Name>**
123	Charlie
987	Alice

isEmpty: False

size: 2

Items are adding to a Map using *put(Key,Value)*

put(456, “Bob”)

Map	
Key <StudentID>	**Value <Student Name>**
123	Charlie
987	Alice
456	Bob

isEmpty: False

size: 3

Items are adding to a Map using *put(Key,Value)*

put(456, “Bob”)

Map	
Key <StudentID>	**Value <Student Name>**
123	Charlie
987	Alice
456	Bob

isEmpty: False

size: 3

Items are adding to a Map using *put(Key,Value)*

put(456, "Bob")

Map	
Key <StudentID>	**Value <Student Name>**
123	Charlie
987	Alice
456	Bob

isEmpty: False

size: 3

- **NOTE: Keys are not necessarily kept in order**
- **Implementation details left to the designer**
- **Today we use a tree, but we will discuss another option next class**

If an item already exits, *put(Key,Value)* will update the Value for that Key

put(987, “Ally”)

Map	
Key <StudentID>	**Value <Student Name>**
123	Charlie
987	Alice
456	Bob

isEmpty: False
size: 3

If an item already exits, *put(Key,Value)* will update the Value for that Key

put(987, "Ally")

Map	
Key <StudentID>	**Value <Student Name>**
123	Charlie
987	Ally
456	Bob

isEmpty: False
size: 3

***put* overwrites Value if item with Key is already in Map**

Can remove items by Key and get Value for that Key (or null if Key not found)

remove(987) => “Ally”

Map	
Key <StudentID>	**Value <Student Name>**
123	Charlie
987	Ally
456	Bob

isEmpty: False
size: 3

Removes item with Key and returns Value

Can remove items by Key and get Value for that Key (or null if Key not found)

remove(987) => null

Map	
Key <StudentID>	**Value <Student Name>**
123	Charlie
456	Bob

isEmpty: False

size: 2

Returns null if Key not found
Does not throw Exception

keyset() returns a Set of Keys in the Map

keyset() => Set {123, 456}

Map	
Key <StudentID>	**Value <Student Name>**
123	Charlie
456	Bob

isEmpty: False
size: 2

Set has an iterator which can be used to iterate over all Keys in Map

get(Key) returns the Value for the Key (or null if Key not found)

get(456) => “Bob”

Map	
Key <StudentID>	**Value <Student Name>**
123	Charlie
456	Bob

isEmpty: False
size: 2

get(Key) returns the Value for the Key (or null if Key not found)

get(987) => null

Map	
Key <StudentID>	**Value <Student Name>**
123	Charlie
456	Bob

isEmpty: False
size: 2

Returns null if Key not found
Does not throw Exception

containsKey(Key) returns True if Key in Map, False otherwise

containsKey(123) => True

Map	
Key <StudentID>	**Value <Student Name>**
123	Charlie
456	Bob

isEmpty: False
size: 2

containsKey(Key) returns True if Key in Map, False otherwise

containsKey(987) => False

Map	
Key <StudentID>	**Value <Student Name>**
123	Charlie
456	Bob

isEmpty: False
size: 2

containsValue(Value) returns True if Value in Map, False otherwise

containsValue("Bob") => True

Map	
Key <StudentID>	**Value <Student Name>**
123	Charlie
456	Bob

isEmpty: False
size: 2

containsValue(Value) returns True if Value in Map, False otherwise

containsValue("Alice") => False

Map	
Key <StudentID>	**Value <Student Name>**
123	Charlie
456	Bob

isEmpty: False
size: 2

Trees are one way to implement the Map ADT

Maps implemented with Trees

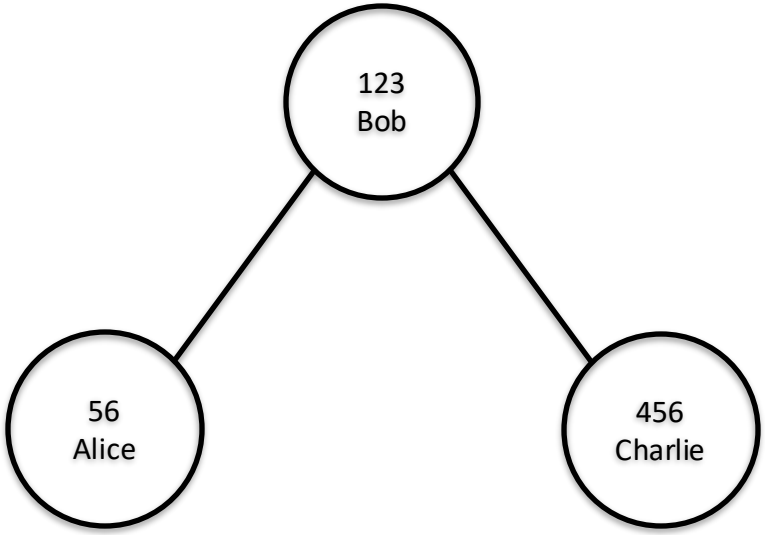
- Could implement as a List, but linear search time
- Like Sets, Trees are natural way to think about Map implementation
- Problem: no easy way to implement `containsValue()` because Tree searches for Keys not Values (but `containsKey()` is easy!)
  - Could search entire Tree for Value
    - Problem: linear time

  - Idea: keep a Set of values, update on each `put` and then search Set
    - Problem: the same Value could be stored with different keys, so if delete Key from Map, can't necessarily delete Value from Set

  - Better idea: keep a second Tree with Values as Keys and counts of each Value
    - When adding a Value, increment its count in the second Tree
    - When deleting a Key, decrement Value count, delete Value in second Tree if count goes to zero
    - Now have *O(h)* time search for `containsValue()`
    - Uses more memory, but has better speed

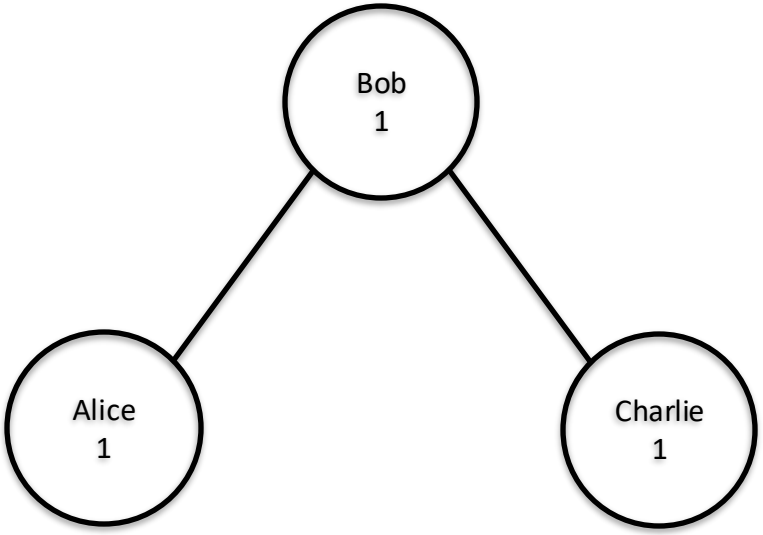
containsValue() keep two trees: trade memory for speed

Tree with Key and Value



- Each node has Key and Value
- Duplicate <u>Values</u> allowed, duplicate <u>Keys</u> not allowed
- Easy to do *containsKey(key)*
  - Search Tree for *key*
  - Return false if hit leaf and *key* not found, else true

Tree with Value and count

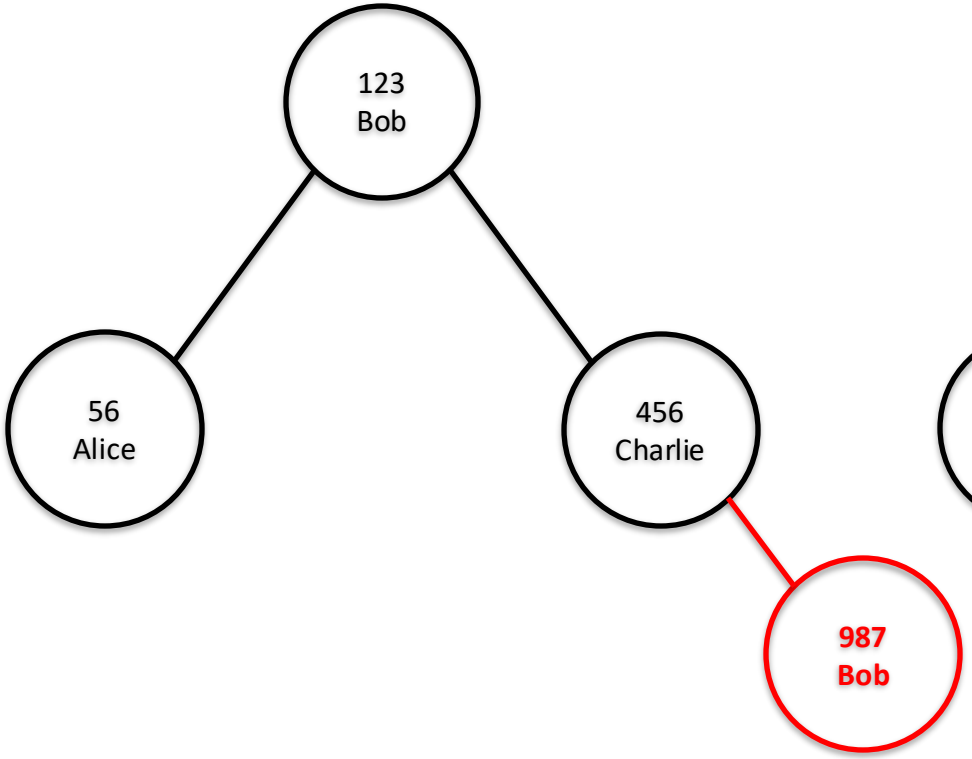


- Each node has Value and <u>count</u> of how many times Value in Map
- Easy to do *containsValue(value)*
  - Search Tree for *value*
  - Return false if hit leaf and *value* not found, else true
- **Approach trades memory for speed**

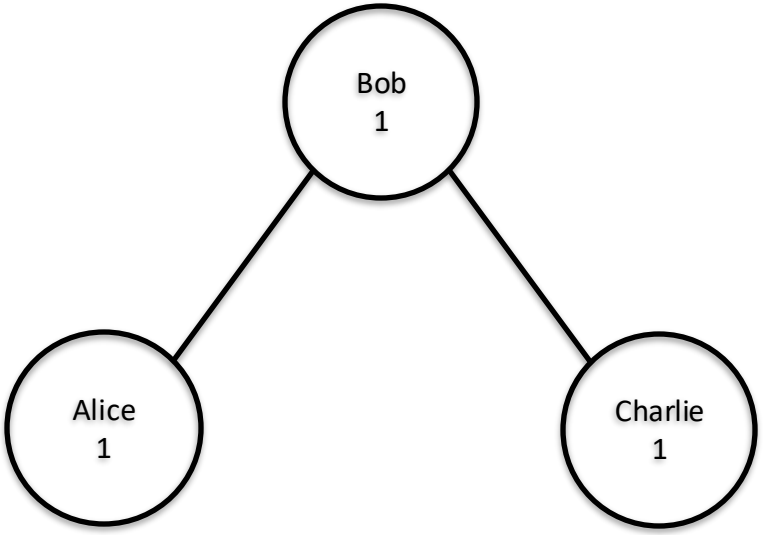
On *put(key,value)*, add Key/Value to Tree, increment count (if needed)

put(987, "Bob")

Tree with Key and Value



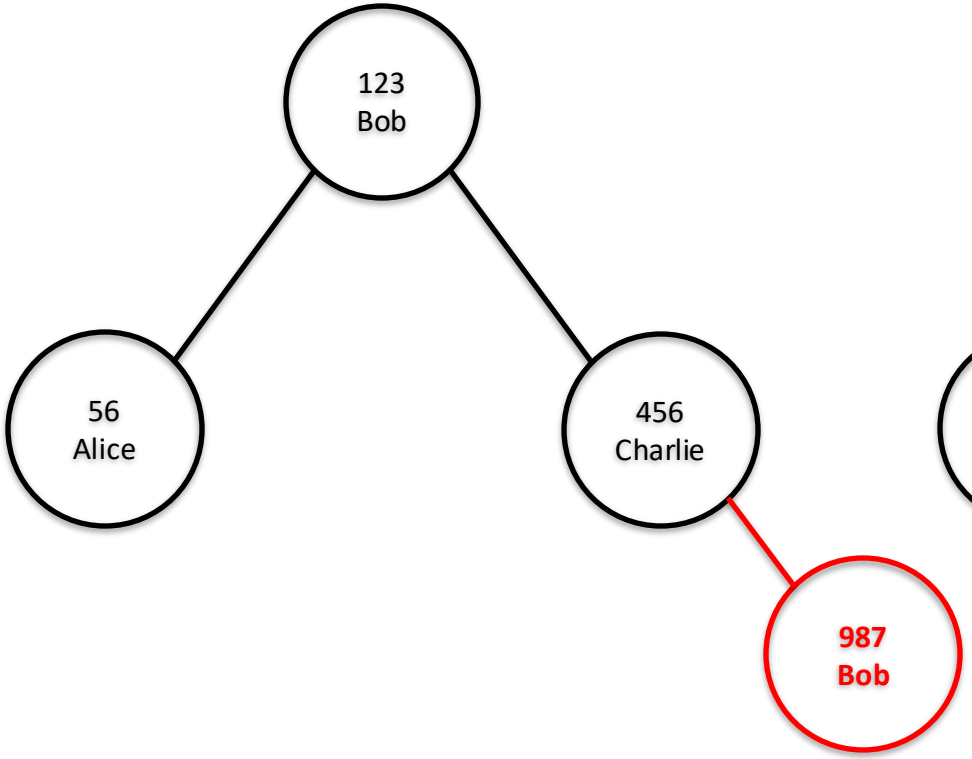
Tree with Value and count



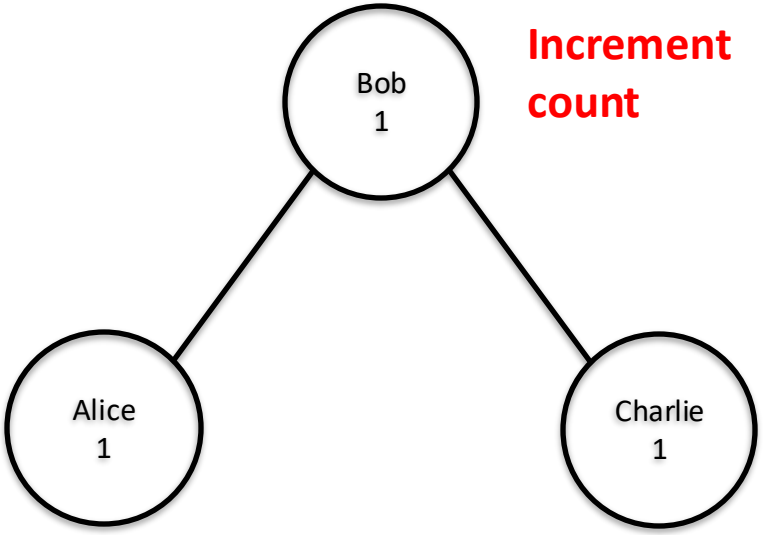
On *put(key,value)*, add Key/Value to Tree, increment count (if needed)

put(987, "Bob")

Tree with Key and Value



Tree with Value and count

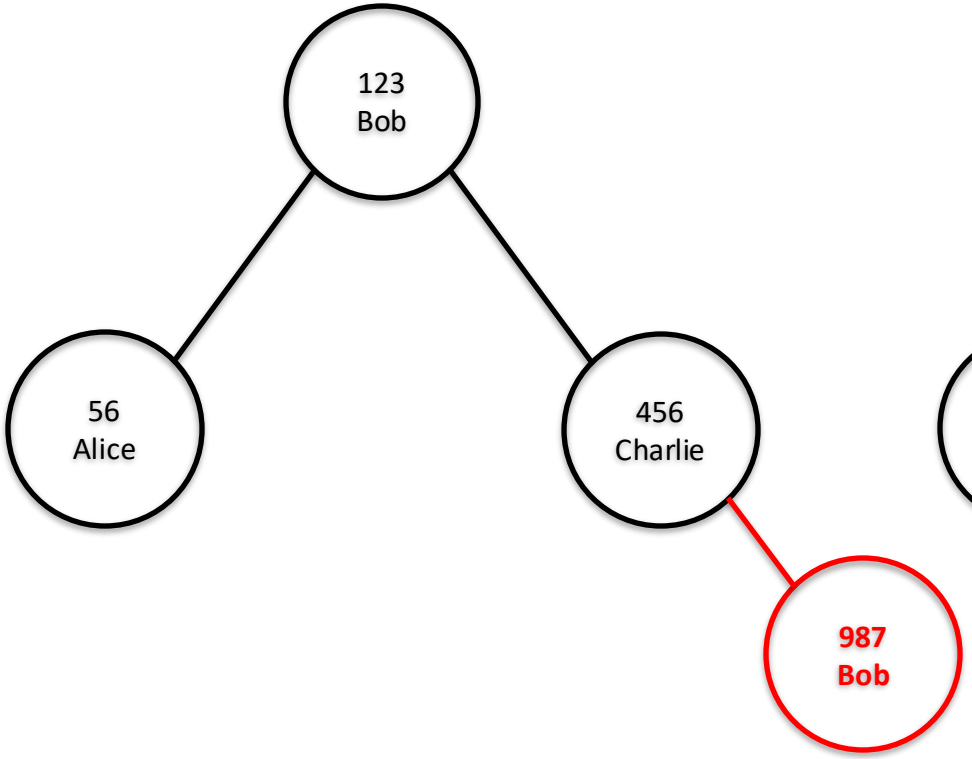


Increment count

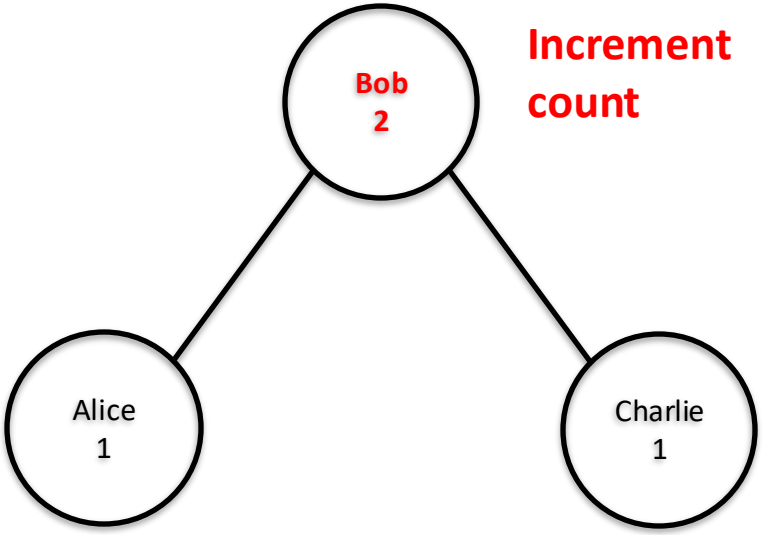
On *put(key,value)*, add Key/Value to Tree, increment count (if needed)

put(987, "Bob")

Tree with Key and Value



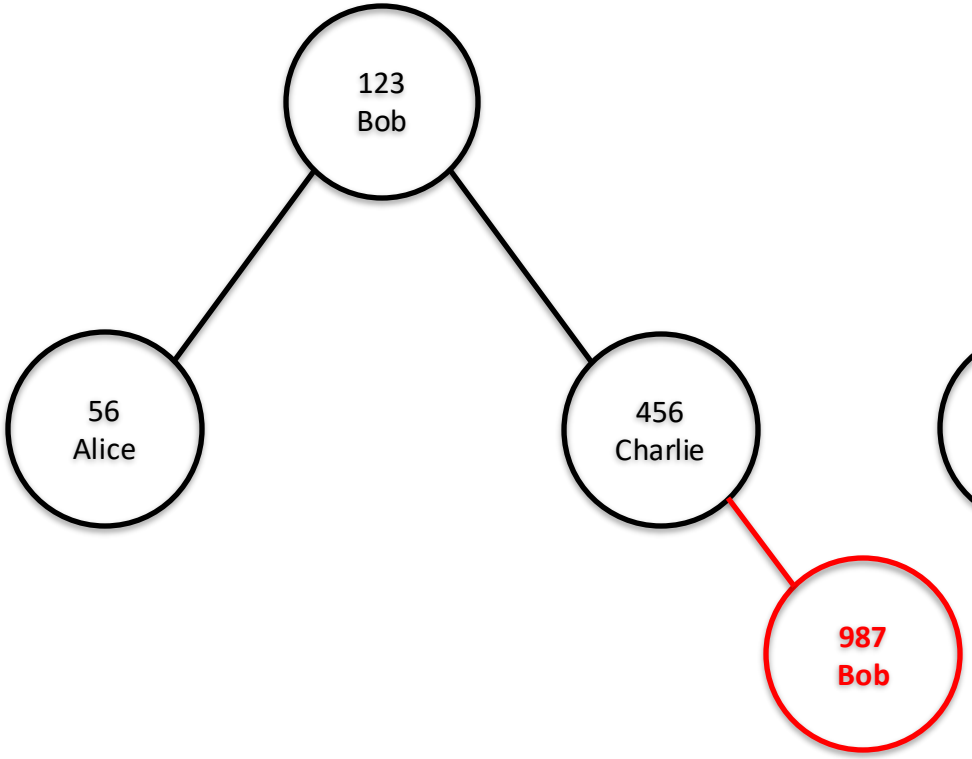
Tree with Value and count



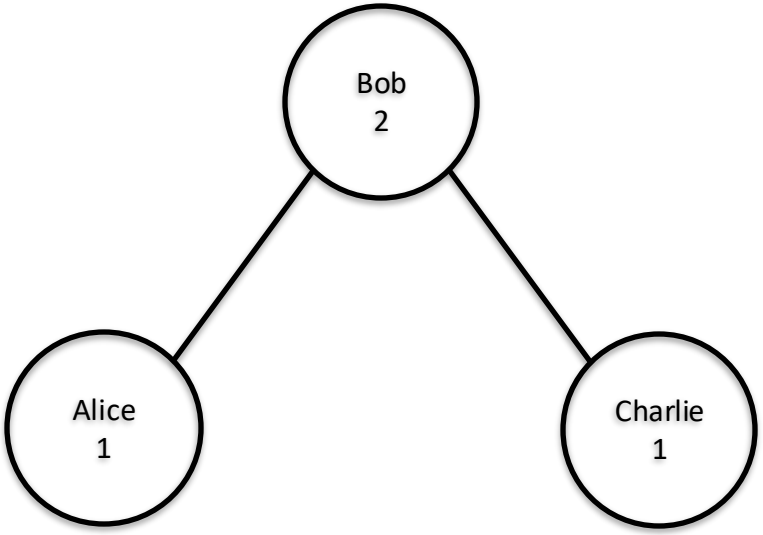
On *remove(key)*, delete Key/Value and decrement count

remove(987)

Tree with Key and Value



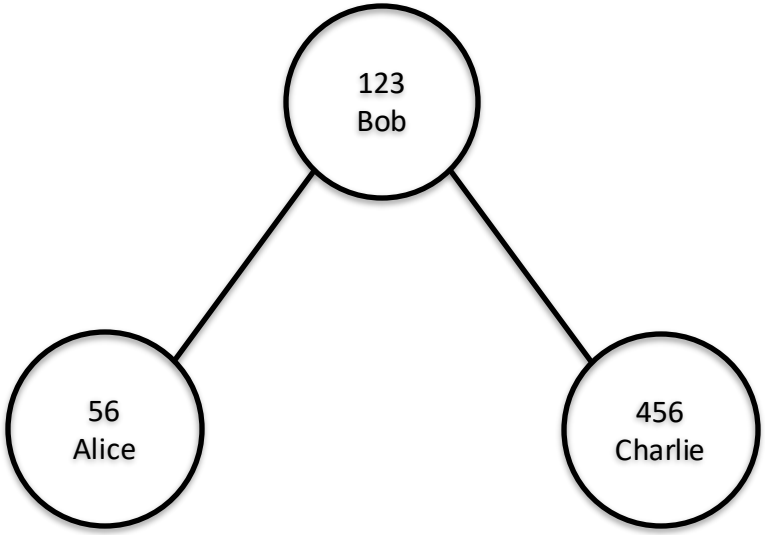
Tree with Value and count



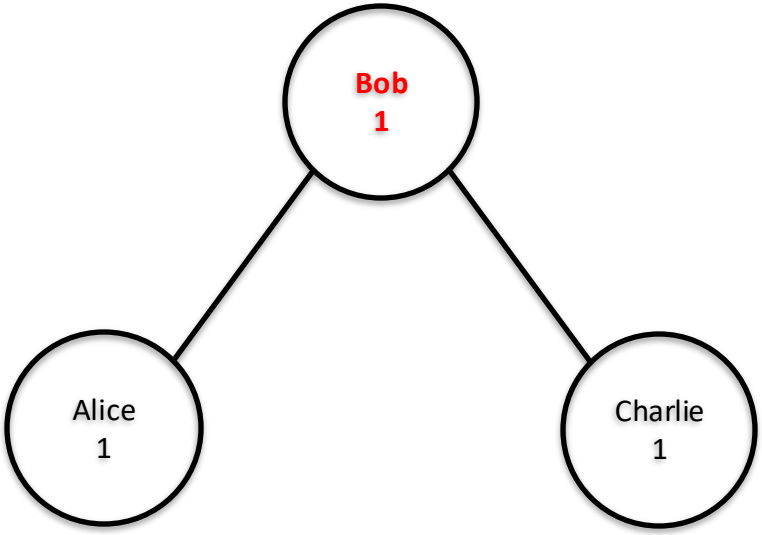
On *remove(key)*, delete Key/Value and decrement count

remove(987)

Tree with Key and Value



Tree with Value and count

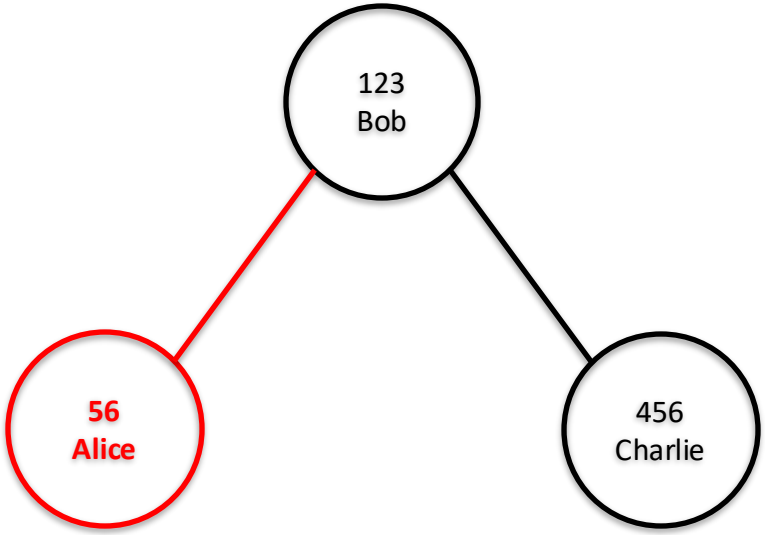


- **Know there is still one "Bob" in the Tree**
- **Don't delete node "Bob" from this tree**

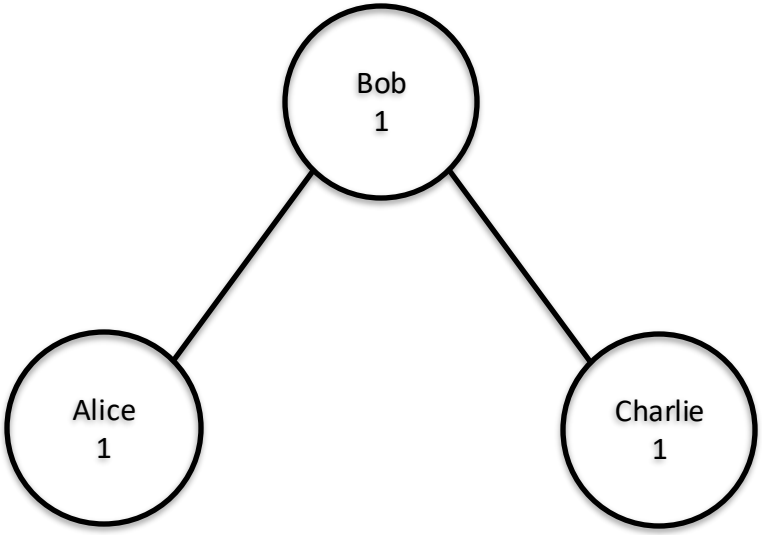
On *remove(key)*, delete Key/Value and decrement count

remove(56)

Tree with Key and Value



Tree with Value and count

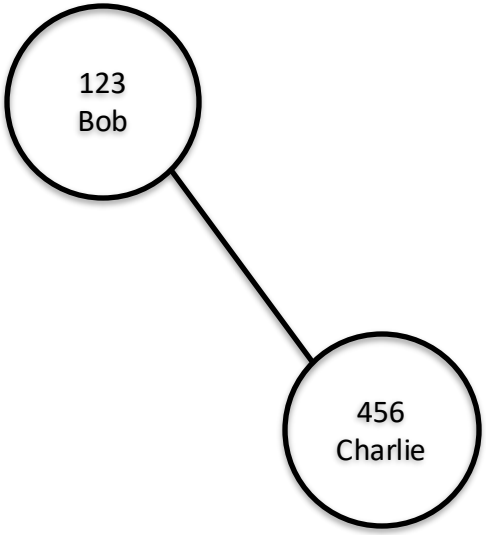


Remove "Alice"

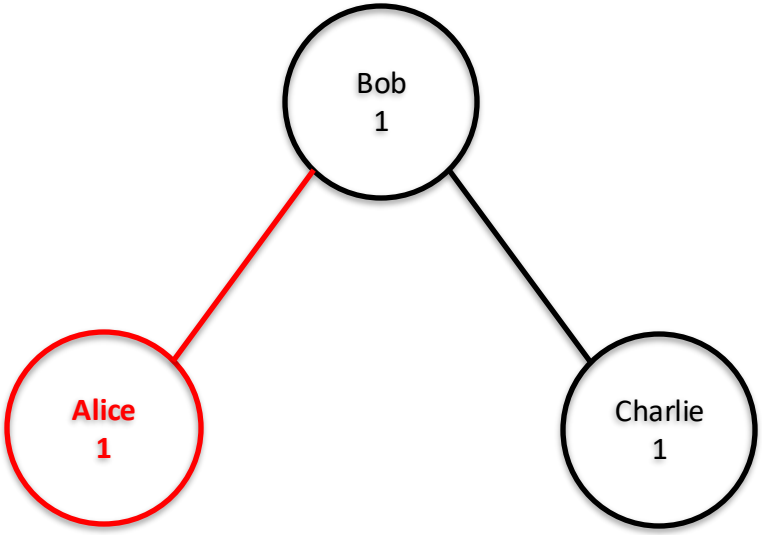
On *remove(key)*, delete Key/Value and decrement count

remove(56)

Tree with Key and Value



Tree with Value and count



Because count goes to 0, remove "Alice" here too

Must also update counts if a *put()* replaces a value

Key point: trade memory for speed!

Can use a Map to count word occurrences in a body of text

Text from which to identify unique words

"**Pretend** that this string was loaded from a web page. We won't go to all that trouble here. This string contains multiple words. And multiple copies of multiple words. And multiple words with multiple copies. It is to be used as a test to demonstrate how sets work in removing redundancy by keeping only one copy of each thing. Is it very very redundant in having more than one copy of some words?"

Pseudocode

- Create Map with String Key and Integer Value
- Loop over each `word` in text
- If Map `contains(word)`
  - Increment count Value
  - Else `put(word)` with Value 1
- Print Map when done

Map	
Key <String>	**Value <Integer>**
Pretend	1

Can use a Map to count word occurrences in a body of text

Text from which to identify unique words

"Pretend **that** this string was loaded from a web page. We won't go to all that trouble here. This string contains multiple words. And multiple copies of multiple words. And multiple words with multiple copies. It is to be used as a test to demonstrate how sets work in removing redundancy by keeping only one copy of each thing. Is it very very redundant in having more than one copy of some words?"

Pseudocode

- Create Map with String Key and Integer Value
- Loop over each `word` in text
- If Map `contains(word)`
  - Increment count Value
  - Else `put(word)` with Value 1
- Print Map when done

Map	
Key <String>	**Value <Integer>**
Pretend	1
that	1

Can use a Map to count word occurrences in a body of text

Text from which to identify unique words

"Pretend that **this** string was loaded from a web page. We won't go to all that trouble here. This string contains multiple words. And multiple copies of multiple words. And multiple words with multiple copies. It is to be used as a test to demonstrate how sets work in removing redundancy by keeping only one copy of each thing. Is it very very redundant in having more than one copy of some words?"

Pseudocode

- Create Map with String Key and Integer Value
- Loop over each `word` in text
- If Map `contains(word)`
  - Increment count Value
  - Else `put(word)` with Value 1
- Print Map when done

Map	
Key <String>	**Value <Integer>**
Pretend	1
that	1
this	1

Can use a Map to count word occurrences in a body of text

Text from which to identify unique words

"Pretend that this string was loaded from a web page. We won't go to all **that** trouble here. This string contains multiple words. And multiple copies of multiple words. And multiple words with multiple copies. It is to be used as a test to demonstrate how sets work in removing redundancy by keeping only one copy of each thing. Is it very very redundant in having more than one copy of some words?"

Pseudocode

- Create Map with String Key and Integer Value
- Loop over each `word` in text
- If Map `contains(word)`
  - Increment count Value
  - Else `put(word)` with Value 1
- Print Map when done

Map	
Key <String>	**Value <Integer>**
Pretend	1
that	2
this	1
…	

UniqueWordCounts.java: Use Map to count word occurrences in a body of text

```
9  public class UniqueWordsCounts {
10     public static void main(String[] args) {
11         String page = "Pretend that this string was loaded from a web pa
12         String[] allWords = page.split("[ .,?!]+");
13         // Declare new Map to hold count of each word
14         Map<String,Integer> wordCounts = new TreeMap<String,Integer>();
15         // Loop over all the words split out of the string, adding to ma
16         for (String s: allWords) {
17             String word = s.toLowerCase();
18             // Check to see if we have seen this word before, update wor
19             if (wordCounts.containsKey(word)) {
20                 // Have seen this word before, increment the count
21                 wordCounts.put(word, wordCounts.get(word)+1);
22             }
23             else {
24                 // Have not seen this word before, add the new word (Jav
25                 wordCounts.put(word, 1);
26             }
27         }
28         // Print word counts
29         System.out.println(wordCounts);
```

Large amount of text simulates webpage

Split into words (aka tokens)

Java has Map based on Trees

Implements Map interface

String Key, Integer Value

Loop over all words

Check if word seen previously

Update word counts

We have seen this word before, increments Value for this Key

Have not seen this word before, *put()* into Map with a value of 1 for word Key

Printing Map calls *toString()* on TreeMap class

Problems @ Javadoc Declaration Console Debug Expressions Error Log Call Hierarchy

```
terminated> UniqueWordsCounts [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Jan 2, 2018, 7:53:53 AM)
{a=2, all=1, and=2, as=1, be=1, by=1, contains=1, copies=2, copy=2, demonstr
```

Maps can also contain Objects such as a List as their Value

- Track position where each word appears (first word is at index 0)
- Word may appear in multiple positions (e.g., 7th and 41st index)
- Need a way to track many items for each word (word is Key in Map)
- Use Map with a *List* as the Value instead of Object representation of a primitive type (e.g., Integer)
- Map will hold many Lists, one List for each Key
- Here each List element is Integer, represents index where word found

Map	
Key<String>	**Value <List <Integer>>**
Pretend	head → 0 → \
that	head → 1 → 15 → \
this	head → 2 → 18 → \
…	

Values as objects is a powerful concept indeed!

UniqueWordPositions.java: Maps can also contain Objects such as a List as their Value

```
 9 public class UniqueWordsPositions {
10     public static void main(String[] args) {
11         String page = "Pretend that this string was loaded from a web page.  We wo
12         String[] allWords = page.split("[ .,?!]+");
13         // Declare new Map, each entry in the Map is a List that will hold the ind
14         Map<String,List<Integer>> wordPositions = new TreeMap<String,List<Integer>
15         // Loop over all the words split out of the string, adding their positions
16         for (int i=0; i<allWords.length; i++) {
17             String word = allWords[i].toLowerCase();
18             // Check to see if we have seen this word before, update wordCounts ap
19             if (wordPositions.containsKey(word)) {
20                 // Now each item in the Map is a List of Integers, add the positio
21                 wordPositions.get(word).add(i);
22             }
23             else {
24                 // Add the new word with a new list containing just this position
25                 List<Integer> positions = new ArrayList<Integer>();
26                 positions.add(i);
27                 wordPositions.put(word, positions);
28             }
29         }
30         System.out.println(wordPositions);
31     }
32 }
```

Create Map with String as Key and List of Integers as Value

Loop over all words

Check if word seen previously

- **If Map has this word as a Key then *add()* position where word found to List**
- ***get()* returns Value which is a List here**

Update word positions

- **Create a new List if we haven't seen this word before**
- ***add()* word to new List**
- **Then *put(word, List)* into Map**

Problems | Javadoc | Declaration | Console | Debug | Expressions | Error Log | Call Hierarchy

```
terminated> UniqueWordsPositions [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Jan 2, 2018, 11:00:29 AM)
a=[7, 41], all=[14], and=[23, 29], as=[40], be=[38], by=[51], contains=[20], copies=[2
```

Agenda

1. Set ADT
2. Map ADT
3. Reading from file/keyboard
4. Search

Key points:

1. **Java provides a FileReader class for reading files**
2. **Java provides a Scanner class for reading from the keyboard**

UniqueWordPositionsFile.java: Read words from a file instead of hard-coded String

```
12 public class UniqueWordsPositionsFile {
13     /**
14      * Collects all the lines from a file into a single string
15      */
16     private static String loadFileIntoString(String filename) throws Exception {
17         BufferedReader in = new BufferedReader(new FileReader(filename));
18         String str = "", line;
19         while ((line = in.readLine()) != null) str += line;
20         in.close();
21         return str;
22     }
23
24     public static void main(String[] args) throws Exception {
25         String page = loadFileIntoString("inputs/text.txt");
26         String[] allWords = page.split("[ .,?!]+");
27
28         // Declare new Map, each entry in the Map is a List that will hold the index w
29         Map<String,List<Integer>> wordPositions = new TreeMap<String,List<Integer>>();
30
31         // Loop over all the words split out of the string, adding their positions in
32         for (int i=0; i<allWords.length; i++) {
33             String word = allWords[i].toLowerCase();
34
35             // Check to see if we have seen this word before, update wordCounts approp
36             if (wordPositions.containsKey(word)) {
37                 // Now each item in the Map is a List of Integers, add the position of
38                 wordPositions.get(word).add(i);
39             }
40             else {
```

- NOTE: Throws exception
- What would happen if file not found?
- Here would pass exception to caller (may end execution)

BufferedReader can read from a file on disk

Append each line from file onto String *str*

Don't forget to close file

- Load String page from a file
- Rest of the code is the same as UniqueWordsPosition.java

```
Problems  @ Javadoc  Declaration  Console  Debug  Expressions  Error Log  Call Hierarchy
<terminated> UniqueWordsPositionsFile [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Jan 2, 2018, 11:25:26 AM)
{a=[7, 25, 50], all=[14], and=[32, 38], as=[49], at=[18], be=[47], by=[60], contains=[29],
```

A scanner can be used to read input from keyboard

```java
import java.util.Scanner;

public class ScannerTest {

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String line;
        int i;
        //scanners read from the keyboard
        //they can parse input for different types
        System.out.println("Enter String");
        line = in.nextLine();
        System.out.println("Got String: " + line);
        //now try reading an integer
        System.out.println("Enter integer");
        i = in.nextInt();
        System.out.println("Got int: " + i);
    }
}
```

Declare *Scanner* to read from keyboard

Parses input to match assigned type (e.g., read input as a String with *nextLine()*) Execute pauses until user presses Enter key

Parse input as an integer with *nextInt()*

Problems | Javadoc | Declaration | Console | Debug | Expressions | Error Log | Call Hierarchy

```
<terminated> ScannerTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Jan 2, 2018,
Enter String
test
Got String: test
Enter integer
7
Got int: 7
```

Agenda

1. Set ADT
2. Map ADT
3. Reading from file/keyboard
4. Search

Search.java: Make different data structures to help answer questions

Shakespeare works

- Hamlet
- Julius Caesar
- King Lear
- Macbeth
- Midsummer
- Othello
- Romeo & Juliet
- Tempest

Read

file2WordCounts

- Use filename as Key
- Store how many times each word appears in file
- Map of Maps!

Key <String> filename	Value Map<<String>,<Integer>> word	count
hamlet.txt	forbear	1
	the	1,150
	…	
juliusCaesar.txt	the	606

numWords

- Map filename to number of words in file

Key <String> filename	Value <Integer> number words
hamlet.txt	32,831
juliusCaesar.txt	21,183

numFiles: # of files word is in

Key <String> word	Value <Integer> number files
forbear	3
forsooth	3
the	8

totalCounts: How many total times word appears

Key <String> word	Value <Integer> total count
forbear	6
forsooth	5
the	5,716

Demo: Search.java uses Scanner and data structures to answer questions

Type a word to see how many times it appears in each file

- Love
- Forbear
- Forsooth
- Audience suggestion

 # n to get n most common words

- Try top 10 words with # 10, then # 100
- **Try bottom 10 words with # -10, then # -100**

Can restrict to just a single file with # n (e.g., # 10 hamlet.txt)

Search multiple words, does an AND

Play around on your own

Key points

1. Sets are an unordered collection of items like the mathematical notion of a set
2. Sets prevent duplicates
3. Can be implemented with trees (Java provides a TreeSet)
4. Maps look items up by Key and return Value
5. Python programmers, think dictionaries
6. Can be implemented with trees (Java provides a TreeMap)
7. Java provides a FileReader class for reading files
8. Java provides a Scanner class for reading from the keyboard