# 11.1 Set abstract data type

## Set abstract data type

A ***set*** is a collection of distinct elements. A set ***add*** operation adds an element to the set, provided an equal element doesn't already exist in the set. A set is an unordered collection. Ex: The set with integers 3, 7, and 9 is equivalent to the set with integers 9, 3 and 7.



**PARTICIPATION ACTIVITY** | 11.1.1: Set abstract data type.

### Animation content:

`undefined`

### Animation captions:

1. Adding 67, 91, and 14 produces a set with 3 elements.
2. Because 91 already exists in the set, adding 91 any number of additional times has no effect.
3. Set 2 is built by adding the same numbers in a different order.
4. Because order does not matter in a set, the 2 sets are equivalent.

**PARTICIPATION ACTIVITY** | 11.1.2: Set abstract data type.

1) Which of the following is not a valid set?

- { 78, 32, 46, 57, 82 }
- { 34, 8, 92 }
- { 78, 28, 91, 28, 15 }

2) How many elements are in a set that is built by adding the element 28 6 times, then the element 54 9 times?



- 1
- 2
- 15

3) Which 2 sets are equivalent?

- { 56, 19, 71 } and { 19, 65, 71, 56 }
- { 88, 54, 81 } and { 81, 88, 54 }
- { 39, 56, 14, 11 } and { 14, 56, 93, 11 }



## Element keys and removal

Set elements may be primitive data values, such as numbers or strings, or objects with numerous data members. When storing objects, set implementations commonly distinguish elements based on an element's ***key value***: A primitive data value that serves as a unique identifier for the element. Ex: An object for a student at a university may store information such as name, phone number, and ID number. No two students will have the same ID number, so the ID number can be used as the student object's key.

Sets are commonly implemented to use keys for all element types. When storing objects, the set retrieves an object's key via an external function or predetermined knowledge of which object property is the key value. When storing primitive data values, each primitive data value's key is itself.

Given a key, a set ***remove*** operation removes the element with the specified key from the set.

**PARTICIPATION ACTIVITY** | 11.1.3: Element keys and removal.

### Animation content:

`undefined`

### Animation captions:

1. Different students at the same university may have the same name or phone number, but each student has a unique ID number.
2. A set for the course roster uses the student ID as the key value, since the exact same student cannot enroll twice in the same course.
3. The call to remove Student C provides only the student ID.



**PARTICIPATION ACTIVITY** | 11.1.4: Element keys and removal.

Refer to the example in the animation above.

1) If the student objects contained a field for GPA, then GPA could be used as the

key value instead of student ID.

- True
- False

2) `SetRemove(courseRosterSet, "Student D")` would remove Student D from the set.



- True
- False

3) SetRemove will not operate properly on an empty set.

- True
- False

## Searching and subsets

Given a key, a set ***search*** operation returns the set element with the specified key, or null if no such element exists. The search operation can be used to implement a subset test. A set X is a ***subset*** of set Y only if every element of X is also an element of Y.

PARTICIPATION ACTIVITY | 11.1.5: SetIsSubset algorithm.

### Animation content:

`undefined`

### Animation captions:

1. To test if set2 is a subset of set1, each element of set 2 is searched for in set1. Elements 19, 22, and 26 are found in set1.
2. Element 34 is in set2 but not set1, so set2 is not a subset of set1.
3. The first element in set3, 88, is not in set1, so set3 is not a subset of set1.
4. All elements of set4 are in set1, so set4 is a subset of set1.
5. No other set is a subset of another.
6. But each set is always a subset of itself.



PARTICIPATION ACTIVITY | 11.1.6: Searching and subsets.

1)

Every set is a subset of itself.

- True
- False

2) For X to be a subset of Y, the number of elements in Y must be greater than or equal to the number of elements in X.



- True
- False

3) The loop in SetIsSubset always performs N iterations, where N is the number of elements in subsetCandidate.

- True
- False

**CHALLENGE ACTIVITY** 11.1.1: Set abstract data type.

361856.2187296.qx3zqy7

Start

Given an empty set numSet, what is numSet after the following operations?

SetAdd(numSet, 28)
SetAdd(numSet, 50)
SetAdd(numSet, 28)
SetAdd(numSet, 28)
SetAdd(numSet, 42)
SetAdd(numSet, 24)
SetAdd(numSet, 24)

{ Ex: 1, 2, 3 }



1 2 3

Check Next

# 11.2 Set operations

## Union, intersection, and difference

The ***union*** of sets X and Y, denoted as X ∪ Y, is a set that contains every element from X, every element from Y, and no additional elements. Ex: { 54, 19, 75 } ∪ { 75, 12 } = { 12, 19, 54, 75 }.



The ***intersection*** of sets X and Y, denoted as X ∩ Y, is a set that contains every element that is in both X and Y, and no additional elements. Ex: { 54, 19, 75 } ∩ { 75, 12 } = { 75 }.

The ***difference*** of sets X and Y, denoted as X \ Y, is a set that contains every element that is in X but not in Y, and no additional elements. Ex: { 54, 19, 75 } \ { 75, 12 } = { 54, 19 }.

The union and intersection operations are commutative, so X ∪ Y = Y ∪ X and X ∩ Y = Y ∩ X. The difference operation is not commutative.

**PARTICIPATION ACTIVITY** 11.2.1: Set union, intersection, and difference.

### Animation content:

`undefined`

### Animation captions:

1. The union operation begins by adding all elements from set1.
2. Each element from set2 is added. Adding elements 82 and 93 has no effect, since 82 and 93 already exist in the result set.
3. The intersection operation iterates through each element in set1. Each element that is also in set2 is added to the result.
4. The difference of set1 and set2, denoted set1 \ set2, iterates through all elements in set1. Only elements 61 and 76 are added to the result, since these elements are not in set2.
5. Set difference is not commutative. SetDifference(set2, set1) produces a result containing only 23 and 46, since those elements are in set2 but not in set1.



**PARTICIPATION ACTIVITY** 11.2.2: Union, intersection, and difference.

1) How many elements are in the set { 83, 5 } ∪ { 9, 77, 83 }?

- 2
- 4

- 5

2) How many elements are in the set { 83, 5 } ∩ { 9, 77, 83 }?
   - 1
   - 2
   - 3



3) { 83, 5 } \ { 9, 77, 83 } = ?
   - { 83 }
   - { 5 }
   - { 83, 5 }

4) { 9, 77, 83 } \ { 83, 5 } = ?
   - { 9, 77 }
   - { 9, 77, 83 }
   - { 5 }

5) Which set operation is not commutative?
   - Union
   - Intersection
   - Difference

6) When X and Y do not have any elements in common, which is always true?
   - X ∪ Y = X ∩ Y
   - X ∩ Y = X \ Y
   - X \ Y = X

7) Which is true for any set X?
   - X ∪ X = X ∩ X
   - X ∪ X = X \ X
   - X \ X = X ∩ X



## Filter and map

A ***filter*** operation on set X produces a subset containing only elements from X that satisfy a particular condition. The condition for filtering is commonly represented by a ***filter predicate***: A function that takes an element as an argument and returns a Boolean value indicating whether or not that element will be in the filtered subset.

A ***map*** operation on set X produces a new set by applying some function F to each element. Ex: If X = {18, 44, 38, 6} and F is a function that divides a value by 2, then SetMap(X, F) = { 9, 22, 19, 3 }.

**PARTICIPATION ACTIVITY** 11.2.3: SetFilter and SetMap algorithms.



## Animation content:

undefined

## Animation captions:

1. SetFilter is called with the EvenPredicate function passed as the second argument.
2. SetFilter calls EvenPredicate for each element. EvenPredicate returns true for each even element, and false for each odd element.
3. Every element for which the predicate returned true is added to the result, producing the set of even numbers from set1.
4. SetFilter(set1, Above90Predicate) produces the set with all elements from set1 that are greater than 90.
5. SetMap is called with the OnesDigit function passed as the first argument. Like SetFilter, SetMap calls the function for each element.
6. The returned value from each OnesDigit call is added to the result set, producing the set of distinct ones digit values.
7. SetMap(set1, StringifyElement) produces a set of strings from a set of numbers.

**PARTICIPATION ACTIVITY** 11.2.4: Using SetFilter with a set of strings.

Suppose stringSet = { "zyBooks", "Computer science", "Data structures", "set", "filter", "map" }. Filter predicates are defined below. Match each SetFilter call to the resulting set.

```
StartsWithCapital(string) {
   if (string starts with capital letter) {
      return true
   }
   else {
      return false
   }
}

Has6OrFewerCharacters(string) {
   if (length of string <= 6) {
      return true
   }
   else {
      return false
   }
}

EndsInS(string) {
   if (string ends in "S" or "s") {
      return true
   }
   else {
      return false
   }
}
```



SetFilter(stringSet, Has6OrFewerCharacters)

SetFilter(stringSet, EndsInS)

SetFilter(stringSet, StartsWithCapital)

| | |
|---|---|
| | { "zyBooks", "Data structures" } |
| | { "Computer science", "Data structures" } |
| | { "set", "filter", "map" } |

Reset



**PARTICIPATION ACTIVITY** 11.2.5: Using SetMap with a set of numbers.

Suppose numbersSet = { 6.5, 4.2, 7.3, 9.0, 8.7 }. Map functions are defined below. Match each SetMap call to the resulting set.

```
MultiplyBy10(number) {
   return number * 10.0
}

Floor(number) {
   return floor(number)
}

Round(number) {
   return round(number)
}
```



SetMap(numbersSet, MultiplyBy10) SetMap(numbersSet, Round)

SetMap(numbersSet, Floor)

| | |
|---|---|
| | { 65.0, 42.0, 73.0, 90.0, 87.0 } |
| | { 7.0, 4.0, 9.0 } |
| | { 6.0, 4.0, 7.0, 9.0, 8.0 } |

Reset

PARTICIPATION ACTIVITY | 11.2.6: SetFilter and SetMap algorithm concepts.

1) A filter predicate must return true for elements that are to be added to the resulting set, and false for elements that are not to be added.

- True
- False

2) Calling SetFilter on set X always produces a set with the same number of elements as X.



- True
- False

3) Calling SetMap on set X always produces a set with the same number

of elements as X.

- True
- False

4) Both SetFilter and SetMap will call the function passed as the second argument for every element in the set.



- True
- False

**CHALLENGE ACTIVITY** 11.2.1: Set operations.

361856.2187296.qx3zqy7

Start

Given:
setA = { 49, 70, 92, 45 }
setB = { 45, 60, 70, 49 }

What is SetUnion(setA, setB)?

{ Ex: 1, 2, 3 }

What is SetIntersection(setA, setB)?

{ }

1 2 3 4

Check Next



# 11.3 Static and dynamic set operations

A ***dynamic set*** is a set that can change after being constructed. A ***static set*** is a set that doesn't change after being constructed. A collection of elements is commonly provided during construction of a static set, each of which is added to the set. Ex: A static set constructed from the list of integers (19, 67, 77, 67, 59, 19) would be { 19, 67, 77, 59 }.

Static sets support most set operations by returning a new set representing the operation's result. The table below summarizes the common operations for static and dynamic sets.



Table 11.3.1: Static and dynamic set operations.

| Operation | Dynamic set support? | Static set support? |
|---|---|---|
| Construction from a collection of values | Yes | Yes |
| Count number of elements | Yes | Yes |
| Search | Yes | Yes |
| Add element | Yes | No |
| Remove element | Yes | No |
| Union (returns new set) | Yes | Yes |
| Intersection (returns new set) | Yes | Yes |
| Difference (returns new set) | Yes | Yes |
| Filter (returns new set) | Yes | Yes |
| Map (returns new set) | Yes | Yes |

**PARTICIPATION ACTIVITY** 11.3.1: Static and dynamic set operations.

1) Static sets do not support union or intersection, since these operations require changing the set.



- True
- False

2) A static set constructed from the list of integers (20, 12, 87, 12) would be { 20,

12, 87, 12 }.

- True
- False

3) Suppose a dynamic set has N elements. Adding any element X and then removing element X will always result in the set still having N elements.



- True
- False

**PARTICIPATION ACTIVITY** 11.3.2: Choosing static or dynamic sets for real-world datasets.

For each real-world dataset, select whether a program should use a static or dynamic set.

1) Periodic table of elements
   - Static
   - Dynamic

2) Collection of names of all countries on the planet
   - Static
   - Dynamic

3) List of contacts for a user
   - Static
   - Dynamic

# 11.4 Java: Set implementation



## Implementing a set with a binary search tree

A set can be implemented in Java using a binary search tree (BST). In practice, a balanced BST or hash table is often used to efficiently implement a set, but a BST will suffice for all necessary set operations.

This section implements a set of integers for simplicity. In practice, a set class is commonly generic, allowing data types other than integers.

The BSTNode class represents a node in a binary search tree. Each node's data stores a set element. References to left child, right child, and parent nodes are also included. The count(), getSuccessor(), and replaceChild() methods assist with implementation of the BST.

Figure 11.4.1: BSTNode class definition.



```
class BSTNode {
   public int data;
   public BSTNode parent;
   public BSTNode left;
   public BSTNode right;

   public BSTNode(int data, BSTNode parent, BSTNode left, BSTNode right) {
      this.data = data;
      this.parent = parent;
      this.left = left;
      this.right = right;
   }

   public BSTNode(int data, BSTNode parent) {
      this(data, parent, null, null);
   }

   public int count() {
      int leftCount = 0;
      int rightCount = 0;
      if (left != null) {
         leftCount = left.count();
      }
      if (right != null) {
         rightCount = right.count();
      }
      return 1 + leftCount + rightCount;
   }

   public BSTNode getSuccessor() {
      // Successor resides in right subtree, if present
      if (right != null) {
         BSTNode successor = right;
         while (successor.left != null) {
            successor = successor.left;
         }
         return successor;
      }

      // Otherwise the successor is up the tree
      // Traverse up the tree until a parent is encountered from the left
      BSTNode node = this;
      while (node.parent != null && node == node.parent.right) {
         node = node.parent;
      }
      return node.parent;
```

```
   }

   public void replaceChild(BSTNode currentChild, BSTNode newChild) {
      if (currentChild == left) {
         left = newChild;
         if (left != null) {
            left.parent = this;
         }
      }
      else if (currentChild == right) {
         right = newChild;
         if (right != null) {
            right.parent = this;
         }
      }
   }
}
```



**PARTICIPATION ACTIVITY** | 11.4.1: BSTNode class.

1) The statement below _____.

`BSTNode newNode = new BSTNode(21, null);`

- causes a compilation error
- causes a runtime error
- allocates a node with data set to 21 and parent, left, and right set to null

2) Calling getSuccessor() for node X will return the node that comes _____ X in the ordering.

- before
- after

3) `currentNode.getSuccessor()` _____.

- always returns a descendant of currentNode
- always returns an ancestor of currentNode
- returns either an ancestor or descendant of currentNode

## BSTIterator class

The set implementation supports iterating through all items in the set. The Set class's iterator() method constructs and returns a BSTIterator object referencing the minimum node in the tree. The BSTIterator class implements an iterator that stores a reference to a node in the BST and supports moving to the successor node when the next() method is called.



Figure 11.4.2: BSTIterator class definition.

```
class BSTIterator implements Iterator<Integer> {
   private BSTNode nextNode;

   public BSTIterator(BSTNode startNode) {
      nextNode = startNode;
   }

   public boolean hasNext() {
      return nextNode != null;
   }

   public Integer next() {
      Integer returnMe = null;
      if (nextNode != null) {
         returnMe = nextNode.data;
         nextNode = nextNode.getSuccessor();
      }
      return returnMe;
   }
}
```

**PARTICIPATION ACTIVITY** 11.4.2: BSTIterator class.

1) BSTIterator keeps a reference to all nodes in the tree.

- True
- False



2) When nextNode is null, BSTIterator's next() method throws an exception to indicate completion of iteration.

- True
- False

3) The tree's root node is passed to

BSTIterator's constructor to construct an iterator that iterates through the set's elements in ascending order.

- True
- False

## Set class: BST methods



The Set class implements standard BST functionality in the add(), remove(), and nodeSearch() methods. The contains() method uses nodeSearch() and returns true if an element is in the set, false if the element is not in the set. The size() method returns the number of elements in the set by counting the number of nodes in the BST.

Figure 11.4.3: Set class methods for implementing a BST.

```
class Set implements Iterable<Integer> {
   private BSTNode root;

   public Set() {
      root = null;
   }

   public boolean add(Integer newElement) {
      if (contains(newElement)) {
         return false;
      }

      BSTNode newNode = new BSTNode(newElement, null);

      // Special case for empty set
      if (root == null) {
         root = newNode;
         return true;
      }

      BSTNode node = root;
      while (node != null) {
         if (newElement < node.data) {
            // Go left
            if (node.left != null) {
               node = node.left;
            }
            else {
               node.left = newNode;
               newNode.parent = node;
               node = null;
            }
         }
         else {
            // Go right
            if (node.right != null) {
```

```
            if (node.right != null) {
               node = node.right;
            }
            else {
               node.right = newNode;
               newNode.parent = node;
               node = null;
            }
         }
      }
      return true;
   }

   public boolean contains(Integer element) {
      return nodeSearch(element) != null;
   }

   public Iterator<Integer> iterator() {
      // Special case for empty set
      if (root == null) {
         return new BSTIterator(null);
      }

      // Start the iterator at the minimum node
      BSTNode minNode = root;
      while (minNode.left != null) {
         minNode = minNode.left;
      }
      return new BSTIterator(minNode);
   }

   private BSTNode nodeSearch(Integer element) {
      // Search the BST
      BSTNode node = root;
      while (node != null) {
         // Compare node's data against the search element
         if (element.equals(node.data)) {
            return node;
         }
         else if (element > node.data) {
            node = node.right;
         }
         else {
            node = node.left;
         }
      }
      return node;
   }

   public void remove(Integer element) {
      removeNode(nodeSearch(element));
   }

   private void removeNode(BSTNode nodeToRemove) {
      if (nodeToRemove == null) {
          return;
      }

      // Case 1: Internal node with 2 children
      if (nodeToRemove.left != null && nodeToRemove.right != null) {
```

```
         BSTNode successor = nodeToRemove.getSuccessor();

         // Copy the data value from the successor
         int dataCopy = successor.data;

         // Remove successor
         removeNode(successor);

         // Replace nodeToRemove's data with successor data
         nodeToRemove.data = dataCopy;
      }

      // Case 2: Root node (with 1 or 0 children)
      else if (nodeToRemove == root) {
         if (nodeToRemove.left != null) {
            root = nodeToRemove.left;
         }
         else {
            root = nodeToRemove.right;
         }

         if (root != null) {
            root.parent = null;
         }
      }

      // Case 3: Internal node with left child only
      else if (nodeToRemove.left != null) {
         nodeToRemove.parent.replaceChild(nodeToRemove, nodeToRemove.left);
      }

      // Case 4: Internal node with right child only, or leaf node
      else {
         nodeToRemove.parent.replaceChild(nodeToRemove, nodeToRemove.right);
      }
   }

   public int size() {
      if (root == null) {
         return 0;
      }
      return root.count();
   }
}
```



**PARTICIPATION ACTIVITY** 11.4.3: Set class: BST methods.



1) The size() method returns ____ when the root is null.

Check Show answer

2) The remove() method calls both the removeNode() and _____ methods.

Check Show answer



3) The add() method first calls the _____ method to see if the element already exists in the set.

Check Show answer

4) The _____ method returns an iterator that allows iteration through the Set's elements using an enhanced for loop.

Check Show answer

## Set class: methods for set operations

The remaining methods in the Set class implement common set operations: difference(), filter(), intersection(), map(), and union().

Figure 11.4.4: difference(), filter(), intersection(), map(), and union() methods.

```
class Set implements Iterable<Integer> {
   public Set difference(Set otherSet) {
      Set result = new Set();
      for (Integer element : this) {
         if (!otherSet.contains(element)) {
            result.add(element);
         }
      }
      return result;
   }

   public Set filter(Predicate<Integer> predicate) {
      Set result = new Set();
      for (Integer element : this) {
         if (predicate.test(element)) {
            result.add(element);
         }
      }
      return result;
   }

   public Set intersection(Set otherSet) {
      Set result = new Set();
      for (Integer element : this) {
         if (otherSet.contains(element)) {
            result.add(element);
         }
      }
      return result;
   }

   public Set map(Function<Integer, Integer> mapFunction) {
      Set result = new Set();
      for (Integer element : this) {
         Integer newElement = mapFunction.apply(element);
         result.add(newElement);
      }
      return result;
   }

   public Set union(Set otherSet) {
      Set result = new Set();
      for (Integer element : this) {
         result.add(element);
      }
      for (Integer element : otherSet) {
         result.add(element);
      }
      return result;
   }
}
```

**PARTICIPATION ACTIVITY** 11.4.4: Set class methods.

intersection() filter() difference() map() union()



| | |
|---|---|
| | Builds a new set by adding elements returned from a Function object's apply() method. apply() is called for each element in the instance set. |
| | Builds and returns the set of elements that exist only in both sets. |
| | Builds and returns the set of elements that exist in 1 or both sets. |
| | Builds and returns the set of elements that exist only in the instance set, and not in the other set. |
| | Builds and returns the set of elements from the instance set that satisfy the predicate. |

Reset

## zyDE 11.4.1: Java: Set implementation.

Definitions for the Set class are included in the Set.java file. Try changing the set elemer declarations on the first two lines in main() and see if you can calculate the result befor running the code.



Current file: SetDemo.java Load default templ

```
import java.util.function.Function;
import java.util.function.Predicate;

class EvenPredicate implements Predicate<Integer> {
   public boolean test(Integer value) {
      return (int)value % 2 == 0;
   }
```

```java
}

class Over50Predicate implements Predicate<Integer> {
   public boolean test(Integer value) {
      return value > 50;
   }
}

class MapTimes10 implements Function<Integer, Integer> {
   public Integer apply(Integer value) {
```

Run