# 6.1 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** 6.1.1: 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** 6.1.2: 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** 6.1.3: 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** 6.1.4: 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** 6.1.5: 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** 6.1.6: 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** 6.1.1: Set abstract data type.

502696.2096894.qx3zqy7

Start

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

```
SetAdd(numSet, 74)
SetAdd(numSet, 13)
SetAdd(numSet, 63)
SetAdd(numSet, 74)
```

{ Ex: 1, 2, 3 }

1 2 3



Check Next

# 6.2 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** 6.2.1: 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** 6.2.2: 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** 6.2.3: 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** 6.2.4: 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
   }
}
```



If unable to drag and drop, refresh the page.

**SetFilter(stringSet, StartsWithCapital)** **SetFilter(stringSet, Has6OrFewerCharacters)**

**SetFilter(stringSet, EndsInS)**

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

**Reset**



**PARTICIPATION ACTIVITY** 6.2.5: 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)
}
```



If unable to drag and drop, refresh the page.

**SetMap(numbersSet, Round)** **SetMap(numbersSet, Floor)**

**SetMap(numbersSet, MultiplyBy10)**

| | |
|---|---|
| | { 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** 6.2.6: 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** 6.2.1: Set operations.

502696.2096894.qx3zqy7

Start

Given:
setA = { 60, 21, 37 }
setB = { 77, 21, 92 }

What is SetUnion(setA, setB)?

{ Ex: 1, 2, 3 }

What is SetIntersection(setA, setB)?

{ }

1 2 3 4

Check Next



# 6.3 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 6.3.1: 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** 6.3.1: 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** 6.3.2: 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

# 6.4 Python: Set implementation



## Implementing a set with a BST

A set can be implemented in Python using a binary search tree (BST) to store set elements. 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.

Each node's data stores a set element. References to left child, right child, and parent nodes are also included. External functionality in the Set class implementation provides a way to obtain an element's key from a node's data. Various instance methods such as count(), get_successor(), and replace_child() exist to assist with implementation of the BST.

## Figure 6.4.1: BSTNode class definition.



```
class BSTNode:
    def __init__(self, data, parent, left = None, right = None):
        self.data = data
        self.left = left
        self.right = right
        self.parent = parent


    def count(self):
        leftCount = 0
        rightCount = 0
        if self.left != None:
            leftCount = self.left.count()
        if self.right != None:
            rightCount = self.right.count()
        return 1 + leftCount + rightCount


    def get_successor(self):
        # Successor resides in right subtree, if present
        if self.right != None:
            successor = self.right
            while successor.left != None:
                successor = successor.left
            return successor

        # Otherwise the successor is up the tree
        # Traverse up the tree until a parent is encountered from the
left
        node = self
        while node.parent != None and node == node.parent.right:
            node = node.parent
        return node.parent


    def replace_child(self, current_child, new_child):
        if current_child is self.left:
            self.left = new_child
            if self.left:
                self.left.parent = self
        elif current_child is self.right:
            self.right = new_child
            if self.right:
                self.right.parent = self
```

**PARTICIPATION ACTIVITY** 6.4.1: BSTNode class.

1) Which is not a member of the BSTNode class?
   - data
   - parent
   - key



2) A node's left and right data members default to None if not passed as arguments to the constructor.
   - True
   - False

3) Calling get_successor() for node X will return the node that comes _____ X in the ordering.
   - before
   - after

## BSTIterator class

The set implementation supports iterating through all items in the set. The Set class's __iter__() instance 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 next() (Python versions < 3) or __next__() (Python versions ≥ 3) is called.

Figure 6.4.2: BSTIterator class definition.

```
class BSTIterator:
    def __init__(self, node):
        self.node = node

    # For Python versions >= 3
    def __next__(self):
        return self.next()

    # For Python versions < 3
    def next(self):
        if self.node == None:
            raise StopIteration
        else:
            current = self.node.data
            self.node =
self.node.get_successor()
            return current
```



**PARTICIPATION ACTIVITY** 6.4.2: BSTIterator class.

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

2) BSTIterator's next() and __next__() instance methods have equivalent functionality.
   - True
   - False

3) A call to next() or __next__() raises a StopIteration exception when the iterator's node is None.
   - True
   - False



## Set class __ init__ method

The Set class's __init__() method takes an optional argument for a get-key function. The get-key function takes an element as an argument and returns the element's key. If no get-key function argument is passed to the Set constructor, then a default lambda function is used that takes an element as an

argument and returns the same element as the key. The get-key function is used for comparing elements in the set's BST implementation.

## Python lambda functions

A ***lambda function*** provides a shorthand notation for creating a function object. An expression of the form:

```
get_key = lambda x: x
```

assigns get_key as a function object with functionality equivalent to:

```
def get_key(x):
    return x
```

A lambda function allows a variable to reference a function without declaring the function as global or an instance method.



## Figure 6.4.3: Set class __init__ method.

```
def __init__(self, get_key_function = None):
    self.storage_root = None
    if get_key_function == None:
        # By default, the key of an element is
itself
        self.get_key = lambda el: el
    else:
        self.get_key = get_key_function
```

**PARTICIPATION ACTIVITY** 6.4.3: Set class __init__ method.

1) Initializing a Set object using the code
`set1 = Set()`
initializes set1 with the default get-key function.



- True
- False

2) Initialization of the storage_root member does not depend on the get_key_function argument.
   - True
   - False

3) The get-key function is expected to return an object that supports comparison via the >, <, and == operators.
   - True
   - False



## Set class definition

The Set class implements standard BST functionality in the add(), remove(), and search() methods. The __len__() method returns the set's length by counting the number of nodes in the BST. The remaining methods implement common set operations such as difference(), filter(), intersection(), map(), and union().

Figure 6.4.4: Set class definition.

```
class Set:
    def __init__(self, get_key_function=None):
        self.storage_root = None
        if get_key_function == None:
            # By default, the key of an element is itself
            self.get_key = lambda el: el
        else:
            self.get_key = get_key_function

    def __iter__(self):
        if self.storage_root == None:
            return BSTIterator(None)
        minNode = self.storage_root
        while minNode.left != None:
            minNode = minNode.left
        return BSTIterator(minNode)

    def add(self, new_element):
        new_elementKey = self.get_key(new_element)
        if self.node_search(new_elementKey) != None:
            return False

        newNode = BSTNode(new_element, None)
        if self.storage_root == None:
            self.storage_root = newNode
        else:
            node = self.storage_root
            while node != None:
                if new_elementKey < self.get_key(node.data):
                    # Go left
                    if node.left:
                        node = node.left
                    else:
                        node.left = newNode
                        newNode.parent = node
                        return True
                else:
                    # Go right
                    if node.right:
                        node = node.right
                    else:
                        node.right = newNode
                        newNode.parent = node
                        return True

    def difference(self, other_set):
        result = Set(self.get_key)
        for element in self:
            if other_set.search(self.get_key(element)) == None:
                result.add(element)
        return result

    def filter(self, predicate):
        result = Set(self.get_key)
        for element in self:
            if predicate(element):
                result.add(element)
        return result

    def intersection(self, other_set):
```

```
        result = Set(self.get_key)
        for element in self:
            if other_set.search(self.get_key(element)) != None:
                result.add(element)
        return result

    def __len__(self):
        if self.storage_root == None:
            return 0
        return self.storage_root.count()

    def map(self, map_function):
        result = Set(self.get_key)
        for element in self:
            new_element = map_function(element)
            result.add(new_element)
        return result

    def node_search(self, key):
        # Search the BST
        node = self.storage_root
        while (node != None):
            # Get the node's key
            node_key = self.get_key(node.data)

            # Compare against the search key
            if node_key == key:
                return node
            elif key > node_key:
                node = node.right
            else:
                node = node.left
        return node

    def remove(self, key):
        self.remove_node(self.node_search(key))

    def remove_node(self, node_to_remove):
        if node_to_remove != None:
            # Case 1: Internal node with 2 children
            if node_to_remove.left != None and node_to_remove.right != None:
                successor = node_to_remove.get_successor()

                # Copy the data value from the successor
                dataCopy = successor.data

                # Remove successor
                self.remove_node(successor)

                # Replace node_to_remove's data with successor data
                node_to_remove.data = dataCopy

            # Case 2: Root node (with 1 or 0 children)
            elif node_to_remove is self.storage_root:
                if node_to_remove.left != None:
                    self.storage_root = node_to_remove.left
                else:
                    self.storage_root = node_to_remove.right

                if self.storage_root:
                    self.storage_root.parent = None
```

```
                self.storage_root.parent = None

            # Case 3: Internal node with left child only
            elif node_to_remove.left != None:
                node_to_remove.parent.replace_child(node_to_remove,
node_to_remove.left)

            # Case 4: Internal node with right child only, or leaf node
            else:
                node_to_remove.parent.replace_child(node_to_remove,
node_to_remove.right)

    def search(self, key):
        # Search the BST
        node = self.node_search(key)
        if node != None:
            return node.data
        return None

    def union(self, other_set):
        result = Set(self.get_key)
        for element in self:
            result.add(element)
        for element in other_set:
            result.add(element)
        return result
```



**PARTICIPATION ACTIVITY** 6.4.4: Set class instance methods.

If unable to drag and drop, refresh the page.

intersection | filter | union | __iter__ | difference | map

| | |
|---|---|
| | Builds a new set by adding elements returned from a function. The function is passed as an argument and is called for each element in the instance set. |
| | Allows a Set object to be iterated through in a for loop. |
| | 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 6.4.1: Python: Set implementation.

Definitions for the Set class are included in the Set_BSTBased.py file. Try altering the sam in main.py to experiment with the Set class's functionality.

Current file: main.py Load default templat

```
1  from Set_BSTBased import Set
2
3  def show_set(set, set_name):
4      print(set_name + ": ", end="")
5      for element in set:
6          print(element, end=" ")
7      print("")
8
9  def even_predicate(element):
10     return (element % 2) == 0
11
12 def over50_predicate(element):
13     return element > 50
14
15 def map_times_10(element):
```

Run



# 6.5 LAB: Implementing StaticSet using a Python set

This section's content is not available for print.



©zyBooks 11/20/23 11:02 1048447
Eric Quezada
UTEPCS2302ValeraFall2023