It is a common programming need to keep track of a collection of things. In this case, we do not need to keep track of the order of the collection, but rather what is contained in the collection. Consider, for example, a collector of baseball cards. The collector cares less about the order in which the cards were acquired, rather only what has been acquired. In cases like this, the set is a perfect choice.

Objectives
By the end of this class, you will be able to:

- Define a set as an Abstract Data Type (ADT).
- Describe various operations that are performed on a set.
- Discuss tradeoffs in different set implementation approaches.
- Build a C++ set class supporting the various set operations.
- Use a set to solve real-world problems.

Prerequisites
Before reading this chapter, please make sure you are able to:

- Implement a class in C++ including operator overloading (C++ Object Oriented Programming, Unit 2).
- Implement a generic ADT including the associated iterator (Lesson 00).

Overview
A set is an ADT that stores a collection of values with no repeats. The client of the set typically does not get to specify the order in which things are stored, but most often an order is preserved for efficiency reasons. A set is formally defined as:

<table>
<thead>
<tr>
<th>Set ADT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection of Data Elements</td>
</tr>
<tr>
<td>An unordered collection of unique objects</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Basic Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Construct a set (usually empty)</td>
</tr>
<tr>
<td>- Check to see if the set is empty</td>
</tr>
<tr>
<td>- Insert: Add an element to the set, checking for duplicates</td>
</tr>
<tr>
<td>- Find: Determine if a given element is in the set</td>
</tr>
<tr>
<td>- Erase: Remove an element from the set</td>
</tr>
<tr>
<td>- Clear: Remove all elements from the set</td>
</tr>
</tbody>
</table>
A Formal Definition of Sets

A set is an ADT representing a collection of unique elements. Mathematically, we define these collections with an uppercase letter such as $S$. A set consists of a collection of items called elements. To represent the elements in a set, use the curly braces. For example, the set of even natural numbers less than 10 are:

$$N = \{2, 4, 6, 8\}$$

Recall that for sets the order does not matter. Therefore the following sets are equivalent:

$$\{2, 4, 6, 8\} = \{4, 2, 8, 6\} = \{8, 6, 4, 2\}$$

Another distinguishing property of a set is that redundant or duplicate values are ignored. Therefore the following are equivalent:

$$\{2, 4, 6, 8\} = \{2, 2, 2, 4, 6, 8\} = \{8, 6, 6, 6, 4, 2\}$$

The cardinality of a set refers to the number of elements in the set. Therefore, the cardinality of the following sets are the same: 3

$$\{1, 2, 3\}$$

$$\{a, a, a, b, c\}$$

$$\{Sam, Sue, Stan\}$$

The empty set, of course, is the set with a cardinality of 0. This is represented by the empty-set symbol $\emptyset$ or by the empty curly braces $\{}\$.

Examples

A few sets that you have undoubtedly encountered include:

- The set of all bools, a finite set of cardinality 2:
  $$B = \{ true, false \}$$

- The set of all lowercase letters, a finite set of cardinality 26.
  $$L = \{ a, b, c, d, ... z \}$$

- The set of all whole numbers, an infinite set or a set with infinite cardinality.
  $$W = \{ 0, 1, 2, 3, 4, ... \}$$

- The set of cards in my poker hand, a finite set of cardinality 5.
  $$H = \{ \text{AceSpades, KingSpades, QueenSpades, JackSpades, 10spades} \}$$

Perhaps the final example is most illustrative of sets. A royal flush is a royal flush, regardless of the order in which the cards are arranged.
Set Operations

The most common operations one would want to perform on a set include finding, inserting, and deleting elements. Additional set operations include union, intersection, difference, and complement.

Find

The first common operation performed on the set is to inquire whether a given element belongs to the set or if the element does not belong. The mathematical way to represent the “belongs to” notion is the “element of” symbol (\(\in\)). For example:

\[ 6 \in \{2, 4, 6, 8\} \]

Conversely:

\[ 7 \notin \{2, 4, 6, 8\} \]

We can thus define the find function for a given set \(S\) with:

\[ S.\text{find}(x) = \begin{cases} \text{true}, & \text{if } x \in S \\ \text{false}, & \text{if } x \notin S \end{cases} \]

Insert

Another common operation to perform on a set is to add or insert an element. For now, we will use the plus symbol (+) to denote insert. Since sets do not contain duplicate elements, the first step in any insert operation is to check for the existence of the element in the set:

\[ S.\text{insert}(x) = \begin{cases} S, & \text{if } x \in S \\ S + x, & \text{if } x \notin S \end{cases} \]

Consider the set of all whole numbers below 5 (W):

\[ W = \{0, 1, 2, 3, 4\} \]

First we will attempt to add the number 3 to the set. Since the number 3 is already in the set, \(W\) remains unchanged:

\[ W + 3 = \{0, 1, 2, 3, 4\} \]

Next, we will add the number 10 to the set. Since 10 is not currently in the set, the set will grow by one.

\[ W + 10 = \{0, 1, 2, 3, 4, 10\} \]

The plus symbol is not actually (+) the standard symbol for inserting an item into a set; there is no symbol in fact. Most often, the Union operator (\(\cup\)) is used to denote adding an element (discussed on the next page).

Delete

The opposite of set insertion is set deletion. For now, we will use the minus symbol (-) to denote delete. If we attempt to delete an element that does not exist in the set, then nothing happens. If the element does exist, then it is removed from the collection:

\[ S.\text{delete}(x) = \begin{cases} S - x, & \text{if } x \in S \\ S, & \text{if } x \notin S \end{cases} \]

As with insertion, there is no standard symbol used for deleting an item from a set. The closest thing we have is Intersection and Difference, both discussed on the next page.
**Union**

One less-common operation to do with sets is union (\( \cup \)). Union is a binary operation taking a set on the left-hand-side (lhs) and the right-hand-side (rhs). The result of this operation is a new set consisting of the elements that are either in lhs or rhs. Consider the two sets: \( S_1 \) and \( S_2 \):

\[
S_1 = \{0, 2, 4, 6, 8\} \quad S_2 = \{4, 5, 6\} \\
S_1 \cup S_2 = \{0, 2, 4, 5, 6, 8\}
\]

Observe how the elements present in both \( S_1 \) and \( S_2 \) (\( \{4, 6\} \)) are not duplicated in \( S_1 \cup S_2 \) because sets do not have duplicate elements. Union is commonly used when two or more sets are to be combined.

**Intersection**

Set intersection (\( \cap \)) is defined as the set of elements that two sets share in common. Consider the two sets \( S_1 \) and \( S_2 \):

\[
S_1 = \{0, 2, 4, 6, 8\} \quad S_2 = \{4, 5, 6\} \\
S_1 \cap S_2 = \{4, 6\}
\]

Intersection is used when it is desirable to know what two sets have in common.

**Difference**

The difference operator returns the members that are in one set but not in another. Consider set \( S_1 \) and \( S_2 \):

\[
S_1 = \{0, 2, 4, 6, 8\} \quad S_2 = \{4, 5, 6\} \\
S_1 - S_2 = \{0, 2, 8\}
\]

Unlike the union and intersection operators, the commutative property does not apply to the difference operator. In other words, \( S_1 - S_2 \neq S_2 - S_1 \). Thus:

\[
S_2 - S_1 = \{5\}
\]

**Complement**

The final set operator is complement operator. Perhaps this is best explained by example. Consider the set of all possible dice values (the cardinality is 6 of course) and a set containing the two highest values:

\[
D_{all} = \{1, 2, 3, 4, 5, 6\} \quad D_{5,6} = \{5, 6\}
\]

The complement of \( D_{5,6} \) is all the elements of \( D_{all} \) except those in \( D_{5,6} \):

\[
-D_{5,6} = \{1, 2, 3, 4\}
\]

Note that the complement is similar to the difference operator with one exception: the complement operator is unary (taking a single operand) while the difference operator is binary. You can define the complement as the difference between the set of all elements and the operand:

\[
D_{all} - D_{5,6} = -D_{5,6}
\]

In most situations, we think about the complement operator in the context of finite sets. For example, if you were to play Go Fish!, you must consider the set of all possible cards. The complement of the cards in your hand are the possible cards in your opponent’s hand.
Implementation

Up to this point, we have made no references as to our implementation choices for a set. As long as the contract of the set is honored, the client (or user of the set) should not care about the implementation. This is a fundamental aspect of encapsulation of course!

Perhaps the simplest implementation of sets is to use arrays as the underlying data structure. Here, we have two choices: maintain a sorted array or leave the array unsorted. Since the definition of a set is “an unordered collection of unique objects,” any order is equally valid. It thus behooves us to choose an order that is most convenient for implementation and most efficient. Right now, it is not readily apparent which implementation is best. We will therefore explore both an unsorted implementation and a sorted implementation.

Find

The find algorithm will be quite different between the sorted and unsorted implementation. With the unsorted array, the elements in the array could be in any order. This forces us to use a linear search. With the sorted implementation, we can take advantage of the improved speed of a binary search.

Unsorted: A linear search involves looping through every element in the array, comparing each with the searched element in turn. The pseudocode for this algorithm is:

```
find (element)
    FOR i ← 1 ... numElements
        IF element = array[i]
            RETURN i
    RETURN numElements
```

Notice the FOR loop. In the best case, we will encounter a matching element at the beginning of the array. In the worst case, we will not find a matching element and have to loop through the entire array. In the common case, the larger the cardinality of the set, the more iterations through the loop. We call this linear because the cost of searching for an element in the set is directly related to the size of the array. If the array size doubles, then it will take twice as long on average to find a given element.

Sorted: A binary search involves keeping track of the beginning and the end of the range of indices in which the searched element may reside. With each iteration, the indices are moved closer to each other because half of the range is eliminated. The pseudocode for this algorithm is:

```
find (element)
    iBegin ← 0
    iEnd ← numElements – 1

    WHILE iBegin ≤ iEnd
        iMiddle ← (iBegin + iEnd) / 2
        IF element = array[iMiddle]
            RETURN iMiddle
        IF element < array[iMiddle]
            iEnd ← iMiddle – 1
        ELSE
            iBegin ← iMiddle + 1
    RETURN numElements
```

Though this algorithm is considerably more complex, it is much faster. The best case exists when the element is in the middle of the list. The worst case is when the element is not found. In those cases, we will need to search \( \log_2 n \) elements. How much better is \( \log_2 n \) than linear? If \( n = 1,000 \), then \( \log_2 10 = 10 \). If \( n = 1,000,000 \), then \( \log_2 20 \). Thus, \( \log_2 n \) grows much slower than linear as \( n \) gets larger. Slower growth means faster execution time. We will discuss this more in Chapter 10.4 of the text in a few weeks.


**Insert**

Recall that the insert method adds an element to the set if the element is not currently found in the set. This means that two problems need to be solved: a means must be defined for determining if the element is in the set, and a means must be defined for inserting the element into the set.

**Unsorted:** We will start with a set implementation done with an unsorted array. Since the only way to find an element in an unsorted array is to use a linear search, we must write a loop to pass through every element in the array. If the element is found, nothing is done. Otherwise the new element is inserted at the end of the array. The pseudocode for this algorithm is:

```plaintext
insert(element)
   FOR i ← 1 ... num_elements
      IF element = array[i]
         RETURN
      array[num_elements++] ← element
```

The cost of inserting an element into an unsorted set is the same as finding an element in an unsorted set: linear. Might this be faster if we use a sorted array?

**Sorted:** When inserting an element in a sorted array set, there are two steps: first find the location where the element should go, then shift the remaining elements in the array over by one. To do this, assume that our `find()` function above returned an index to the spot in the array where the element would go if found. In this case, the insert pseudocode would be:

```plaintext
insert(element)
   iInsert ← find(element)
   if array[iInsert] ≠ element
      FOR i ← numElements ... iInsert by -1's
         array[i + 1] ← array[i]
      array[iInsert] ← element
      num_elements++
```

The best-case is \( \log_2 n \) which occurs when the element already exists in the list. The cost for this comes from the binary-search in the `find()` function. This is considerably faster than the linear case in the un-sorted implementation. The worst-case occurs when the element is not found and we need to shift the rest of the array. In the case that the spot where element is inserted is at the beginning of the list (\( i_{\text{Insert}} = 0 \)), then the entire array must be shifted. Thus the cost is \( \log_2 n + n \). Because \( n \) is so much larger than \( \log_2 n \), this is still considered a linear algorithm.
Delete

To delete an element from a set, two problems must be solved: the location must be found of the element to be removed, and filling in the spot with another element in the set.

**Unsorted:** To delete an element from a set implemented with an un-sorted array, it is necessary to first find the element to be deleted and then to shift all the elements over by one slot. This requires two linear algorithms: one to find the element and the second to shift the rest of the array. Note that we can do an interesting optimization. Instead of shifting the array, we can simply move the last element in the set to the slot formerly occupied by the deleted element.

```cpp
define delete(element)
  FOR i ← 1 ... numElements
      IF element = array[i]
          array[i] ← array[--numElements]
      RETURN
```

On average, we will have to loop through half the elements in the array before the desired element is found. The best case is when the first element in the set is to be deleted, the worst is when the element is not found. Since the cost increases linearly with the number of elements in the set, the cost is linear.

**Sorted:** These same basic operations must be performed when we have a sorted array implementation. First we locate the element to be deleted using our \( \log_2 n \) \( \text{find}() \) function, then shift the rest of the list over by one spot. In many ways, the algorithm is similar to that of our insert function:

```cpp
#define delete(element)
  iDelete = find(element)
  if array[iDelete] = element
      FOR i ← iDelete ... numElements
          array[i] = array[i + 1]
      numElements--
```

This algorithm is also linear. In the best case, we fail to find the element in \( \log_2 n \) time and do not need to modify the list. In the worst case, we find the element in \( \log_2 n \) time and then need to shift the entire list (linear).
Union

The union operator involves creating a new set that consists of all the elements in either set1 or set2.

Unsorted: There are several ways to implement the union operator. The first is to copy the first set then insert the elements in the second set one-by-one. The algorithm is:

```pseudo
union(set1, set2)
    setReturn = set1
    FOR i = 0 ... set2.numElements
        setReturn.insert(set2.array[i])
    RETURN setReturn
```

This algorithm looks very simple, but is actually quite expensive. The first operation (setReturn = set1) will require a loop to pass through all the elements in set1 and copy them to setReturn. We did not discuss this algorithm, but at best it is done in linear time with respect to set1.num_elements. After this, we need to pass through each element in set2 and, for each element, insert it into the setReturn. Recall that each call to insert is linear. Since this needs to be done \( n \) times, the cost is \( n^2 \). How expensive is \( n^2 \)? If we have 100 elements in the set, it will take on the order of 10,000 iterations. If we have 1,000 elements, then a million iterations will be performed. In other words, this is quite expensive!

Sorted: A more efficient algorithm can be followed if we use the sorted array implementation. We will start with an index referring to the head of each array (set1.array and set2.array). Since both arrays are sorted, we know the lowest of the first element from setReturn must be in set1.array[0] or set2.array[0]. This gives us three cases:

- **set1.array[i1] == set2.array[i2]**: insert one of them to the beginning of setReturn. Since they are the same, it does not matter which you choose. After it is inserted, then advance the index in both set1 and set2 with \( (i1++ \text{ and } i2++) \).
- **set1.array[i1] < set2.array[i2]**: insert set1.array[i1] into setReturn and advance the index for set1 with \( (i1++) \).
- **set1.array[i1] > set2.array[i2]**: insert set2.array[i2] into setReturn and advance the index for set2 with \( (i2++) \).

This process is repeated until all the elements from set1 and set2 have been inserted into setReturn. The pseudocode is:

```pseudo
union(set1, set2)
    iSet1 = 0
    iSet2 = 0
    WHILE (iSet1 < set1.numElements OR iSet2 < set2.numElements)
        IF iSet1 == set1.numElements
            setReturn.addToEnd(set2.array[iSet2++])
        ELSE IF iSet2 == set2.num_elements
            setReturn.addToEnd(set1.array[iSet1++])
        ELSE IF set1.array[iSet1] == set2.array[iSet2]
            setReturn.addToEnd(set1.array[iSet1])
            iSet1++
            iSet2++
        ELSE IF set1.array[iSet1] < set1.array[iSet2]
            setReturn.addToEnd(set1.array[iSet1++])
        ELSE
            setReturn.addToEnd(set2.array[iSet2++])
    RETURN setReturn
```

While this is quite a bit more complex than the earlier implementation, it is also quite a bit more efficient. Since every element in set1 and set2 is viewed exactly once, the total cost of the function is \( (n + m) \) where \( n = set1.numElements \) and \( m = set2.numElements \). This is also linear.
**Intersection**

Implementation of the intersection operator is much like that of the union. In both cases, each element of the first set must be compared against each element of the second.

**Unsorted:** As with the union operator, there are several ways to implement the intersection operator. Perhaps the simplest involves searching for each member of `set1` in `set2`. If it exists, then insert it into `setReturn`:

```cpp
void intersection(set1, set2)
    FOR i = 0 ... set1.numElements
        IF set2.find(set1.array[i])
            setReturn.insert(set1.array[i])
    RETURN setReturn
```

The cost of this algorithm is quite high. Since we need to go through each element in `set1`, then the cost is at least linear with respect to `set1.num_elements`. However, for each element in `set1`, we need to search through `set2`. This would also be linear. If `set1` and `set2` are comparable in size, then we have $n^2$ which is quite expensive.

**Sorted:** As you can imagine, with a sorted implementation of `set1` and `set2`, a more efficient algorithm is possible. There are essentially three cases:

- `set1.array[i1] == set2.array[i2]`: insert one of them to the beginning of `setReturn`. Since they are the same, it does not matter which you choose. After it is inserted, then advance the index `i1++` and `i2++`.
- `set1.array[i1] < set2.array[i2]`: advance the index `i1++`.

The pseudocode is:

```cpp
void intersection(set1, set2)
    iSet1 ← 0
    iSet2 ← 0
    WHILE iSet1 < set1.numElements OR iSet2 < set2.numElements
        IF iSet1 == set1.numElements
            RETURN setReturn
        ELSE IF iSet2 == set2.numElements
            RETURN setReturn
        ELSE IF set1.array[iSet1] == set2.array[iSet2]
            setReturn.addToEnd(set1.array[iSet1])
            iSet1++
            iSet2++
        ELSE IF set1.array[iSet1] < set2.array[iSet2]
            iSet1++
        ELSE
            iSet2++
    RETURN setReturn
```

As with Union, the cost of this algorithm is $(n + m)$. 
Problem 1

Consider the following sets:

\[ S_1 = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 \} \]

\[ S_2 = \{ 2, 3, 5, 7, 11, 13, 17, 19 \} \]

\[ S_3 = \{ 1, 2, 4, 8, 16 \} \]

What is the result of the following:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a.</td>
<td>[ S_1.\text{find}(7) ]</td>
</tr>
<tr>
<td>b.</td>
<td>[ S_1.\text{find}(8) ]</td>
</tr>
<tr>
<td>c.</td>
<td>[ S_2.\text{find}(7) ]</td>
</tr>
<tr>
<td>d.</td>
<td>[ S_2.\text{find}(8) ]</td>
</tr>
<tr>
<td>e.</td>
<td>[ S_3.\text{find}(7) ]</td>
</tr>
<tr>
<td>f.</td>
<td>[ S_3.\text{find}(8) ]</td>
</tr>
</tbody>
</table>

Please see page 3 for questions

Problem 2

Consider the following sets:

\[ S_1 = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 \} \]

\[ S_2 = \{ 2, 3, 5, 7, 11, 13, 17, 19 \} \]

\[ S_3 = \{ 1, 2, 4, 8, 16 \} \]

What is the result of the following:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a.</td>
<td>[ S_1.\text{insert}(7) ]</td>
</tr>
<tr>
<td>b.</td>
<td>[ S_1.\text{insert}(8) ]</td>
</tr>
<tr>
<td>c.</td>
<td>[ S_2.\text{insert}(7) ]</td>
</tr>
<tr>
<td>d.</td>
<td>[ S_2.\text{insert}(8) ]</td>
</tr>
<tr>
<td>e.</td>
<td>[ S_3.\text{insert}(7) ]</td>
</tr>
<tr>
<td>f.</td>
<td>[ S_3.\text{insert}(8) ]</td>
</tr>
</tbody>
</table>

Please see page 3 for questions
Consider the following sets:

\[ S_1 = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 \} \]
\[ S_2 = \{ 2, 3, 5, 7, 11, 13, 17, 19 \} \]
\[ S_3 = \{ 1, 2, 4, 8, 16 \} \]

What is the result of the following:

a. \( S_1.\text{delete}(7) \)
b. \( S_1.\text{delete}(8) \)
c. \( S_2.\text{delete}(7) \)
d. \( S_2.\text{delete}(8) \)
e. \( S_3.\text{delete}(7) \)
f. \( S_3.\text{delete}(8) \)

Please see page 3 for a hint.

Consider the following sets:

\[ S_1 = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 \} \]
\[ S_2 = \{ 2, 3, 5, 7, 11, 13, 17, 19 \} \]
\[ S_3 = \{ 1, 2, 4, 8, 16 \} \]

What is the result of the following:

a. \( S_1 \cup S_2 \)
b. \( S_1 \cup S_3 \)
c. \( S_2 \cup S_3 \)
d. \( S_1 \cup S_1 \)
e. \( S_1 \cup \emptyset \)
f. \( S_1 \cup S_2 \cup S_3 \)

Please see page 4 for a hint.
Problem 5
Consider the following sets:

\[ S_1 = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 \} \]
\[ S_2 = \{ 2, 3, 5, 7, 11, 13, 17, 19 \} \]
\[ S_3 = \{ 1, 2, 4, 8, 16 \} \]

What is the result of the following:

a. \( S_1 \cap S_2 \)
b. \( S_1 \cap S_3 \)
c. \( S_2 \cap S_3 \)
d. \( S_1 \cap S_1 \)
e. \( S_1 \cap \emptyset \)
f. \( S_1 \cap S_2 \cap S_3 \)

Please see page 4 for a hint.

Problem 6
Consider the following sets:

\[ S_1 = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 \} \]
\[ S_2 = \{ 2, 3, 5, 7, 11, 13, 17, 19 \} \]
\[ S_3 = \{ 1, 2, 4, 8, 16 \} \]

What is the result of the following:

a. \( S_1 - S_2 \)
b. \( S_2 - S_1 \)
c. \( S_1 - S_3 \)
d. \( S_2 - S_3 \)
e. \( S_1 - S_1 \)
f. \( S_1 - \emptyset \)

Please see page 4 for a hint.
Problem 7
Consider the following set:

\[ S = \{ a, e, i, o, u \} \]

a. Following the pseudocode for the `insert()` method for an unsorted set implementation presented on page 6, what would be the result of adding the letter \( b \) to the set?

b. Following the pseudocode for the `insert()` method for a sorted set implementation presented on page 6, what would be the result of adding the letter \( b \) to the set?

Please see page 6 for a hint.

Problem 8
Consider the following set:

\[ S = \{ a, e, i, o, u \} \]

a. Following the pseudocode for the `delete()` method for an unsorted set implementation presented on page 7, what would be the result of deleting the letter \( e \) from the set?

b. Following the pseudocode for the `delete()` method for a sorted set implementation presented on page 7, what would be the result of deleting the letter \( e \) from the set?

Please see page 7 for a hint.

Problem 9
Consider the following sets represented as sorted arrays:

\[ S_1 = \begin{array}{cccccc}
  a & e & i & o & u & y \\
  0 & 1 & 2 & 3 & 4 & 5 \\
\end{array} \]

\[ S_2 = \begin{array}{cccccc}
  a & b & c & d & e & f \\
  0 & 1 & 2 & 3 & 4 & 5 \\
\end{array} \]

Following the `union()` algorithm for sorted arrays presented on page 8, consider the case where the algorithm is in the middle of running. In other words, fill in the next item in a desk check. The values of the current state of the algorithm are the first three columns. What is the next step in the execution? Fill in the new values for `iSet1`, `iSet2`, and `setReturn`.

<table>
<thead>
<tr>
<th>iSet1</th>
<th>iSet2</th>
<th>setReturn</th>
<th>iSet1</th>
<th>iSet2</th>
<th>setReturn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>{}</td>
<td>1</td>
<td>1</td>
<td>{ a }</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>{ a, b }</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>{ a, b, c, d }</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>{ a, b, c, d, e, f, i }</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please see page 8 for a hint.