Test Requirement Catalog

Generic Clues, Developer Version

PART 1: DATA TYPES

PART 2: OPERATIONS THAT APPLY TO MANY DATA TYPES

This catalog contains general-purpose clues that are useful in a wide range of programs. It is written for developers looking at code, but most of the entries can be used nonprogrammers looking at the external interface. Some of the entries reveal a bias toward testing, but most can also be used to create code inspection checklists.

There are two parts to the catalog. The first part, Data Types, focuses on particular data types and the operations used with them. The second part, Operations That Apply to Many Data Types, contains operations that can't be usefully indexed under a single data type.

The catalog is organized for skimming. The first level of header looks like this:

Major Section Header

You can use major section headers to skip over whole categories of operations and data types, but you'll spend most of your time scanning headers that look like this:

Basic category of data type or operation

You'll read the contents of sections that are relevant to your code, and skip those that are not. Some basic categories are further subdivided into subcategories whose headers look like this:

Some specialized entries within a basic data type or operation

Within any category, there are two types of text. *Italicized text* is used for examples and explanations that will help you understand the category and the test requirements. You'll skip that text after you're experienced. Text in normal font should be read every time you use a category.

When the test requirements need to refer to a variable by name, I'll usually use the name `Var`. When two variables are used, they'll be `Var1` and `Var2`.

Black bars in the margin indicate entries used in my short course on developer testing. Other readers can ignore the bars.

Part 1: Data Types

Simple Data Types

Boolean

- 0 (the false value)
- 1 (normal true value)
- some true value other than 1 CAN'T SET
- If the code is calculating a boolean, consider inputs for which the result is neither true nor false.
  Example: in a text editor, there is a button like this on the toolbar:

  It is both a control and an indicator. As a control, it is used to change the selected text between italic and non-italic. As an indicator, it displays as pressed when the text is italic, and not pressed when the text is not italic. What should it do if half the text is italic?

Cases

A variable takes on one of a small, distinct set of values. The values are distinct because it's likely that code using the values will have to treat each differently (usually with a switch statement). For example, a searching routine might take an enumerated type argument with four values: search forward from the start, search forward from the current position, search backward from the end, or search backward from the current position.

If a variable is an enumerated type, it's almost always a Case variable, but variables declared as integers are often also used as Cases.

- The first possibility.
- The second possibility.
- ... (and so on through all the possibilities)
- Some impossible case. CAN'T SET
  If possible, choose a boundary. If the valid cases are 1, 2, and 3, choose 0 or 4.

Counts

A Count variable ranges from zero to some larger number. The number of letters in your mailbox is a Count, as is the number of typographical errors in this document. Count variables can't legally be negative.

- -1 CAN'T SET
- 0
- 1 USE ONLY
- >1
- maximum possible value
  If you don't know the maximum, experiment to find it, or at least try an "uncomfortably large" value.
- one larger than the maximum possible value CAN'T SET

If "maximum possible value" is true, so is ">1". Both are included because ">1" is easier to accomplish. If you have multiple tests, it may be convenient to have only one test with a maximum value like 4294967296. Others could have values like 3, 5, 12, and so on. The two separate test requirements remind you of that possibility.

Example: If a program deals some number of cards from a standard deck, the requirements for the USE Count "number dealt" would be:

-1 (an error case)
0
1
52
53 (an error case)

Both the ">1" and "maximum possible value" test requirements are satisfied by 52.

If the program is hooked up to a camera and counts the number of cards a human just dealt, the SET test requirements would be:

0
52

-1 and 53 are impossible. 1 is not known to cause failures when it's a pure output value.

Intervals

Intervals describe variables that can take on values in a range of numbers. They may or may not contain their endpoints. For example, you may have a floating point number that is in the interval from zero (including zero) up to (but not including) 12.

On a computer, there's no such thing as an interval to infinity. All intervals have both upper and lower bounds. For example, on 32-bit computers, an unsigned integer word is an Interval from 0 to 4294967296.

In the following, ε refers to the smallest possible value for the variable. For integers, ε is 1. For floating point numbers, it's a smaller value that depends on the machine.

[LOW, HIGH] an interval that contains both LOW and HIGH

- LOW - ε CAN'T SET
- LOW
- HIGH
- HIGH + ε CAN'T SET

[LOW, HIGH) an interval that contains LOW, but not HIGH

- LOW - ε CAN'T SET
- LOW
- HIGH - ε
- HIGH CAN'T SET

(LOW, HIGH] an interval that does not contain LOW, contains HIGH

- LOW CAN'T SET
- LOW + ε
- HIGH
- HIGH + ε CAN'T SET

(LOW, HIGH) an interval that contains neither LOW nor HIGH

- LOW CAN'T SET
- LOW + ε
- HIGH - ε
- HIGH CAN'T SET

Numbers

Equality tests for floating point numbers

- `Var1` is only slightly different than `Var2`
  Should approximate equality be used instead?

Numbers used in arithmetic operations

Because all numbers are Intervals, use these requirements:

- `Var` is the most positive possible number
- `Var` is the most negative possible number
- `Var` is just beyond the most positive possible number CAN'T SET
- `Var` is just below the most negative possible number CAN'T SET

The following requirements are most useful for more complex arithmetic operations.

- `Var` is 0
- `Var` is a number slightly larger than 0
- `Var` is a number slightly smaller than 0

If there are any other special numbers in your problem domain, such as π, do the same with them.

Percentages

Percentages are Intervals, but they also have one additional test requirement.

- -1 (or a fraction slightly less than zero) CAN'T SET
- 0
- 100
- 101 (or a fraction slightly greater than 100) CAN'T SET
- If the percentage is calculated by the code, attempt to force the denominator to 0.
  As an example, suppose your code calculates the percent of customers late with payment. Will the code work if you have no customers that month?

Pointers

See also "Pairs of Pointers".

- The null pointer
- A pointer to a true object

Strings

Notice that strings are Collections of characters, that they're often treated as Streams of characters, and that programs use Pointers to manipulate them. Keep those other catalog entries in mind.

- The empty string

Comparing strings

- String1 and String2 are the same length, but differ in the last element IMPLEMENTATION
  Example: "abc" and "abd"
- String1 has the same elements as String2, but is one element shorter IMPLEMENTATION
  Example: "ab" and "abc"
- String2 has the same elements as String1, but is one element shorter IMPLEMENTATION
  Example: "abc" and "ab"

Arrays of integers

- Have each element be an illegal index. USE ONLY
  Here's an example:

-100000000	*-1000000000*	*-1000000000*	*343434343434*

This requirement looks for faults where an array element is incorrectly used as an index. Large positive and large negative numbers are more likely to cause obvious failures.

Array indices

Single-dimensioned arrays (vectors)

- `Index` is -1. CAN'T SET.
- `Index` is 0 (the smallest possible index).
- `Index` is the largest valid value.
  Example: Use the index 4 with an array declared as " int array[5]".
- `Index` is one larger than the largest valid value. CAN'T SET

Multidimensional arrays

- All indices are 0 at the same time.
  array[0][0], for example.
- All indices are the largest valid value at the same time.
  For an array declared as "int array[5][5]" be sure to touch element array[4][4].
- The first index is -1, all the rest are valid.
  For an array declared as "int array[5][5]" can inputs be given such that the array reference would evaluate to array[-1][3]?
- The second index is -1, all the rest are valid.
  You don't want more than one invalid index. If the code correctly validates all but one of the indices, trying two invalid indices at once would not discover the fault.
- ... and so on for all the remaining indices.
- The first element is one larger than the largest valid value.
- ... and so on for all the remaining indices.

Collections

Collections are a fundamental data type. Linked lists, arrays, and sets are all collections. Any code that processes groups of objects, each in roughly the same way, is processing a collection. Most loops are associated with collections. Here are some examples of collections:

lines in a file (if the file is processed line by line)
slides in a presentation
a group of slides to be printed (a sub-collection)
paragraphs on a page
highlighted paragraphs on a page (a sub-collection)
the argument list to a function (from the point of view of the compiler)

Spotting collections can require creativity. As important as the collections that are explicitly represented by variables in the code are those implicit collections that might earlier have been overlooked.

Sizes of collections

- an empty collection
- contains exactly one element
- contains more than one element
- the maximum possible size

"Maximum possible size" strictly supercedes "contains more than one". Both are included because "contains more than one" is easier to accomplish. You may have several tests that contain more than one element, but only one where the collection is the maximum possible size.

Contents of collections

- contains duplicate elements

Using an element of a sequenceable collection

A sequenceable collection is one with a first and last element. Typical examples are arrays, non-circular linked lists, and files. For these requirements, it doesn't matter whether collections are processed from first to last or from last to first.

- using the first element IMPLEMENTATION
- using the last element IMPLEMENTATION

"Using an element" means accessing the element's contents in any way.

Using a subset of the collection

- the subset is empty.
- the subset contains exactly one element.
- the subset contains all but one element from the original.
- the subset is the same as the original.

Structured collections

This section concerns itself with collections that have some internal structure.

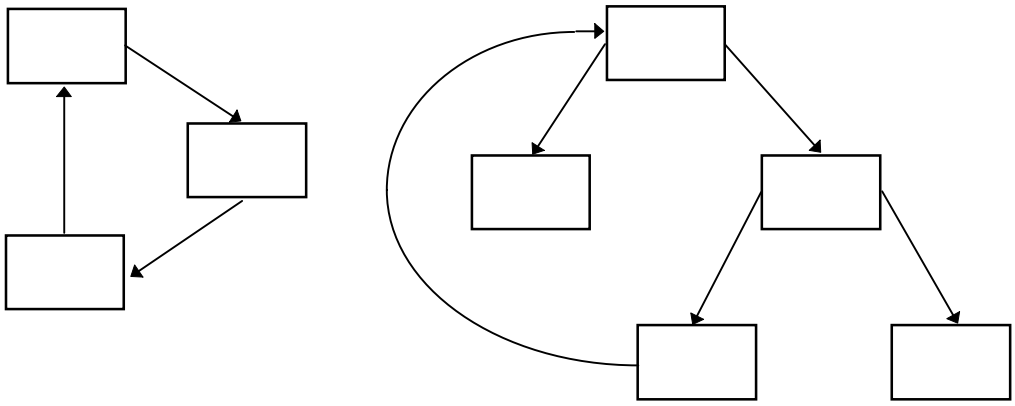
Trees

Trees are Collections. There are also two Counts involved: the children of any given node, and the depth of the tree. Consider Count requirements such as "some node has one child", "some node has more than one child", "the tree has depth 1", "the tree is maximally deep", and so on.

Collections built with references (pointers, indices, etc.)

These are collections where particular elements refer to other elements. The reference is usually by a pointer, but it could also be via an index or some more elaborate way of associating two elements.

- The structure is circular.
  A structure is circular if you can start at some node and follow some chain of references that brings you back to the starting node.



Equality comparison of collections built with references

See the requirements under "Pairs of References".

Pairs of Variables

Pairs of intervals, pairs of counts

If there is more than one Interval or Count in the program, consider adding these requirements:

- `Var1` and `Var2` simultaneously have the smallest possible values.
- `Var1` and `Var2` simultaneously have the largest possible values.

Don't try all combinations of all variables in your code. Consider only pairs of variables that are likely to interact. For example, suppose you have a page_size variable that's an Interval in the range [0,66], and a num_pages variable in the range [0, 1000]. Since the amount of memory needed probably depends on both the page size and the number of pages, the combination would be useful.

If the card dealing program earlier used as an example of Counts took two arguments, the number of cards dealt and the number of players, two useful requirements would be:

> *Zero players get zero cards each.*
> *More than one player gets 52 cards each (this should be an error).*

(It would also be reasonable to consider "total number of cards dealt" as an Interval from 0 to 52.)

Pairs of collections

These requirements will supercede some of the requirements gotten by considering each collection in isolation.

- Both collections are empty. USE ONLY
- The first collection has one element; the other has no elements. USE ONLY
- The first collection has no elements; the second collection has one. USE ONLY
- Both collections have more than one element. USE ONLY

Equal-sized collections

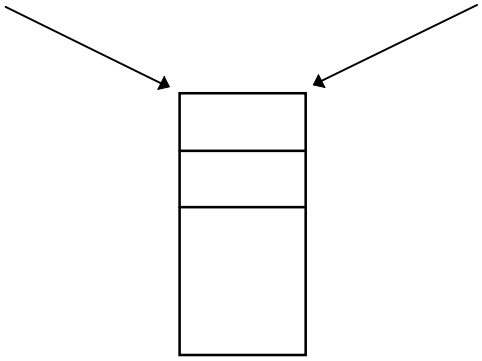
If the collections are supposed to have the same number of elements, add these error cases:

- The first collection has one more element than the second; both have more than one element. CAN'T SET
- The first collection has one fewer element than the second; both have more than one element. CAN'T SET

Pairs of references (usually pointers)

Programmers often neglect the case where two references refer to the same object. They then overwrite what the first refers to, not realizing they've also modified what the second refers to.

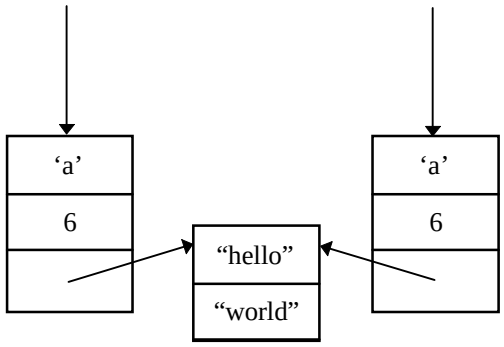
- Two references refer to the same object.



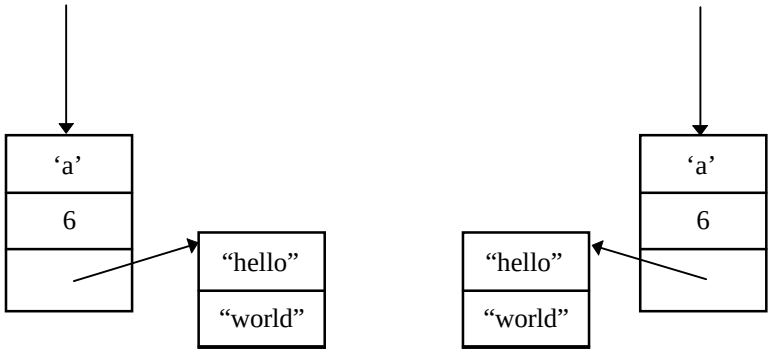
Object equality

In the case where references are tested for equality, programmers can become confused about what "equality" means. The above test requirement checks one meaning of equality. Here are two others:

- Two reference arguments refer to different objects with the same contents.



- Two reference arguments refer to different objects whose non-reference contents are equal. Further, their reference contents refer to objects that are recursively equal.



Part 2: Operations That Apply to Many Data Types

Searching

In all cases, there should be more than one element in the collection being searched.

- Match not found.
  If possible, a matching element should be placed just past the bounds of the search. If the boundaries are handled incorrectly, this increases the chance of an observable failure.
- Exactly one match.
- More than one match in the collection.
- Single match found in the first position. IMPLEMENTATION
- Single match found in the last position.

If the search is backward, the "last" element is actually the first in the collection.

It may seem odd that "single match found in the first position" is marked IMPLEMENTATION and "single match found in the last position" is not. Suppose you're calling a routine to search an array. Such a routine will, in C, have an argument to limit the search. It either gives the number of elements to search OR indicates the last element to search. You might call the routine with one interpretation when it expects the other. Similar errors are possible for other types of collections; they're caused by confusion about whether the search should be up to the terminating element or up to and including the terminating element.

Using the position of the matching element

In this case, the code uses the position of the matching element in later processing. The IMPLEMENTATION tag is removed from the requirements that mention the position of the match.

- Single match found in the first position.
- Single match found in the last position.

Continued searching

Searching is resumed from the last element found.

- Further match is immediately adjacent to the last match.
  For example, suppose your code is searching for $-delimited keywords in text. A good test case would be "some text $key1$$key2$ some more text". An off-by-one error in the code could cause the second keyword to be missed.

Streaming (Processing a Sequence of Elements)

Streaming operations process a sequence of elements one at a time, with no backtracking. Code that processes input from files or terminals or networks is streaming code. Code that adds up all the elements of an array is streaming over the array.

Terminology: the collection of elements that's processed is called the "stream". The code that processes those elements is the "streaming code".

Notation: In the examples, stream elements are written as tokens in parentheses, like (a b c). Streaming code would first process "a", then "b", and finally "c". Duplicate tokens mean the elements are the same. For example, if the stream (a b b) represents network packets, the second and third packets are the same. The first packet is different from them. "The same" and "different" are with respect to the code's intended behavior. If it is only to look at the checksum field in the packet, two packets with identical checksums are "the same", even if the rest of their contents are different.

For clarity, elements that illustrate a particular point may be bolded and capitalized. That has no significance, other than to catch your eye.

Streams can also be considered Collections, so you should consider those test requirements.

Several of the categories below might match your code.

Streams that may contain repeated elements

These requirements apply to code that should treat identical elements specially. For example, if the elements are machines to send updates to, the correct behavior might be to ignore repeated elements. That should be checked. If, on the other hand, the elements are just numbers to be added, repeated elements don't make a difference and aren't worth checking.

- No duplicates in the stream.
  Examples: (a b c), (c a b d)
- All elements are duplicates.
  Examples: (a a), (b b b b b b)
- An element appears at least three times.
  *Examples: (b b b), (**A** b **A** c **A** d)*
- More than one element is duplicated.
  *Examples: (a a b b), (**A** m **B** o p q **A** **B** **C** d **C**)*
- An element appears immediately after its duplicate.
  *Examples: (a a), (b **A** **A** c)*
- There's at least one other element between two duplicates.
  *Examples (**A** b **A**), (x **A** b d e g **A**)*

More than one of these requirements can apply to a single stream. For example, the stream (A A m A) satisfies "an element appears at least three times", "an element appears immediately after its duplicate", and "there's at least one other element between two duplicates".

Streams with only two kinds of values

The "two values" might be explicit, as in a stream of bits. They might also be a result of the interpretation the streaming code places on the actual elements. For example, in a stream of machine names, the two values might be whether the machine was available or not available. There are really a wide variety of elements (the machine names), but only two meaningful values.

- Two or more elements, each with the first value.
  Examples: (0 0), (0 0 0)
- Two or more elements, each with the second value.
  Examples: (1 1), (1 1 1)
- Two or more elements, the first value appears after the second.
  *Examples: (**1 0**), (**1 0** 1), (0 1 **1 0** 1)*
- Two or more elements, the second value appears after the first.
  *Examples: (**0 1**) (1 **0 1**) (**0 1** 0 0)*

*More than one of these requirements can apply to a single stream. For example, the stream (**0 1 1 0** 1) satisfies both of the last two requirements.*

Streams with error handling

These streams may contain elements that trigger error handling. For example, the program may cease processing if a filename is unreadable.

- A stream containing only an error.
  *Example: (**ERROR**)*
- One or more correct elements, an error, one or more correct elements.
  *Examples: (correct **ERROR** correct),*
  *(correct correct **ERROR** correct correct correct)*
  The correct elements before the error check what effect the error handling has on previous processing. If it was supposed to undo previous work, does it? If it wasn't, does it leave that work alone? The correct elements after the error check later processing. If streaming is supposed to resume, does it? If it's supposed to stop, does it?

If the streaming code is to recover from the error and continue processing, add this requirement:

- A stream with two errors in a row, then a correct item.
  *Examples: (**ERROR ERROR** correct), (correct **ERROR ERROR** correct)*

Streams with a fixed number of elements

The number of elements may be predetermined, or it may be calculated from one of the early elements.

- A stream that ends one element prematurely.
  This requirement is especially good for file I/O streams, which often don't check for end-of-file and therefore treat it like valid input.

Streams with strictly increasing elements

In these streams, each new element must be strictly larger than the previous. (The "larger" relationship is not necessarily numeric. For example, each stream element might be a record, one of whose fields was a timestamp string in the format "MM-DD-YY: HHMMSS". The stream itself is the total record of a single customer's transactions, so it should be strictly increasing in time.)

- The stream is strictly increasing.
  Examples: (a b c d), (a b d f z)
  Try to make at least one of the increases as small as possible. In a stream of characters, having 'c' follow 'b' is better than having 'd' follow it.
- The stream contains an element equal to its predecessor. (The stream has a plateau.)
  *Example: (a **B B** d)*
  This is an error case.

Streams with increasing elements

This is like the previous clue, except that the elements need not be strictly increasing. Two successive elements may have equal values.

- The stream contains an element equal to its predecessor. (The stream has a plateau.)
  *Example (a **B B** d)*
- The stream contains an element smaller than its predecessor.
  *Example: (a c **B** d)*
  Try to make the decrease as small as possible. In a stream of characters, having 'b' follow 'c' is better than having 'a' follow it.
  This is an error case.

Streaming Over Two Sequences

Here, two sequences of elements are processed. Each is processed in order (its second element after its first, and so on). However, processing of the two sequences may be interleaved in different ways. For example, one order might be:

1. ***First** element of the **first** sequence.*
2. ***Second** element of the **first** sequence.*
3. *First element of the second sequence.*
4. *Second element of the second sequence.*

A different interleaving would be

1. ***First** element of the **first** sequence.*
2. *First element of the second sequence.*
3. ***Second** element of the **first** sequence.*
4. *Second element of the second sequence.*

These test requirements explore the interleavings.

Searching for matches

These requirements apply if elements in the two sequences match. The implication is that elements will be processed until a match is found, then the match is processed, then the code searches for the next match.

- Two matches in a row USE ONLY
  Here's an example that uses two arrays:

1	*2*	...

1	*2*	...

- A match followed by a mismatch USE ONLY

1	*A*	...

1	*2*	...

- A mismatch followed by a match USE ONLY

A	*2*	...

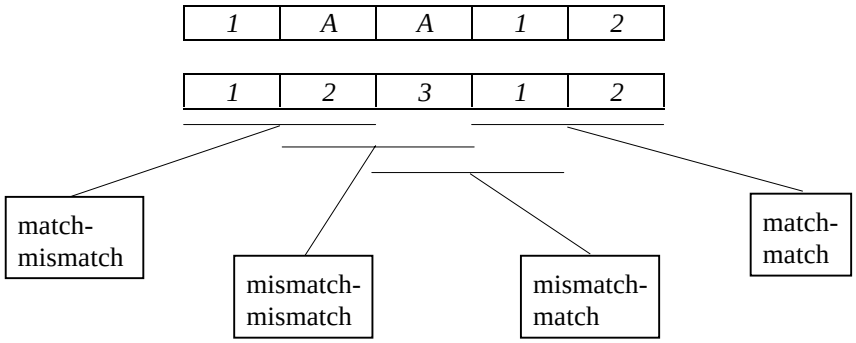
1	*2*	...

- Two mismatches in a row USE ONLY

A	*B*	...

1	*2*	...

The examples above all satisfy the test requirements with the first two elements of the array. That's not required. Here's an array that satisfies all the test requirements:



The following requirements may also apply. In the case where the collections are of different lengths, the "last element" is the last element of the shorter collection.

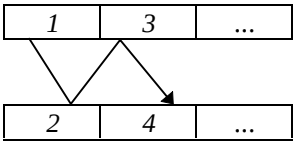
- The first elements match USE ONLY, IMPLEMENTATION
- The first elements mismatch USE ONLY, IMPLEMENTATION
- The last elements match USE ONLY, IMPLEMENTATION
- The last elements mismatch USE ONLY, IMPLEMENTATION
- One or more extra elements in the first sequence
- One or more extra elements in the second sequence

Sorted sequences

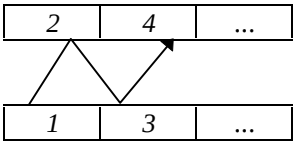
In this situation, elements of two sequences are processed in order. The elements of each sequence are already sorted.

The key here is forcing processing to switch between sequences in ways that may trigger failures. Test requirements are hard to explain in words, so concentrate on the pictures.

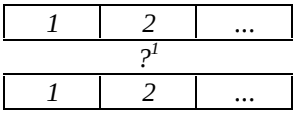
- Back and forth between sequences; first sequence first USE ONLY



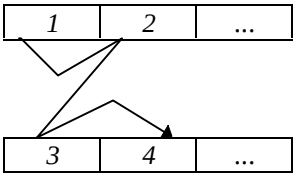
- Back and forth between sequences; second sequence first USE ONLY



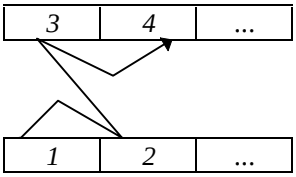
- Pairs of identical elements in the sequences USE ONLY



- First sequence first; then second USE ONLY

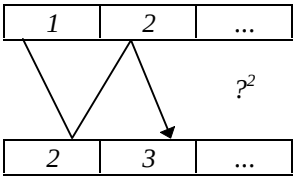


- Second sequence first; then first USE ONLY

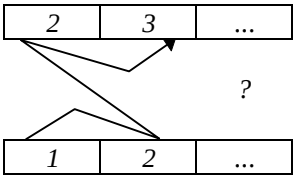


[1] The '?' indicates that the order of processing is not obvious. Does it matter which is done? What's correct behavior?

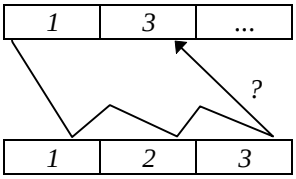
- Single shared element, moving from first to second sequence USE ONLY



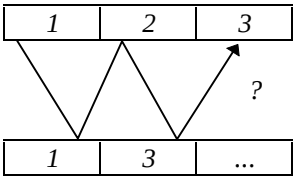
- Single shared element, moving from second to first sequence USE ONLY



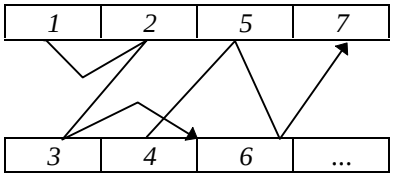
- Common elements with gap in first sequence USE ONLY



- Common elements with gap in second sequence USE ONLY



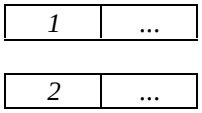
More than one of these requirements can be satisfied in a single test. Be careful to compare elements in processing order. In the following example, the element '4' is both the end of one test requirement and the beginning of the next.



[2] The picture gives one possible order of processing. There's another. Is either correct?

In addition, use these test requirements.

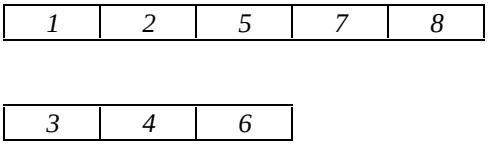
- The first sequence's first element is smaller than the second sequence's first element.



- The first sequence's first element is larger than the second sequence's first element.
- The first sequence's first element is equal to the second sequence's first element.

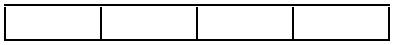
- The first sequence's last element is smaller than the second sequence's last element.
- The first sequence's last element is larger than the second sequence's last element.
- The first sequence's last element is equal to the second sequence's last element.

For the last three test requirements, the sequences don't have to be the same length. In the following example, the first sequence's last element ('8') is larger than the second sequence's last element ('6'). This is true even though, strictly speaking, '8' should never be compared to anything in the second sequence. (That sequence is "used up" when '6' is compared to '7'.)

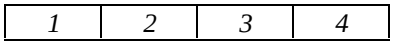


Changing the Contents of a Collection

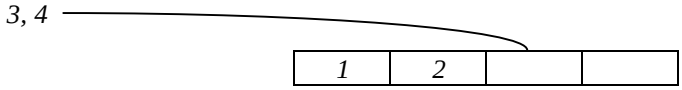
The earlier Collection test requirements assumed the collection was constant. These additional requirements cover changes to collections. In the examples, I'll use a Collection of four elements. Here's its initial (empty) state:



When filled with numbers, it looks like this:

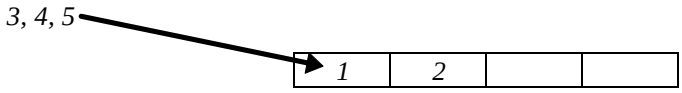


I will represent adding elements to the collection like this:

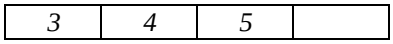


Here, I am adding the entries "3" and "4" to a collection already containing "1" and "2".

I will represent overwriting elements in the collection like this:



Here, I overwrite "1" and "2", yielding this collection:

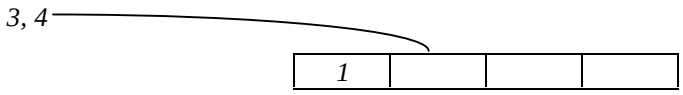


Appending a single element

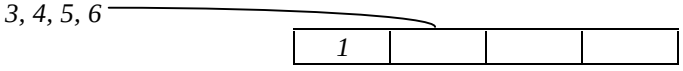
- The collection is initially empty.
- The collection is not initially empty.
- Add the last element that can fit (the collection is now full).
- Attempt to add one more element to a full collection.

Appending zero, one, or more elements at once

- The collection is initially empty; add at least one new element.
- The container is not initially empty; add at least one new element.
- Add zero new elements.
- Add multiple elements, almost filling up the collection IMPLEMENTATION
  This can catch later small additions that "don't need" boundary checking.



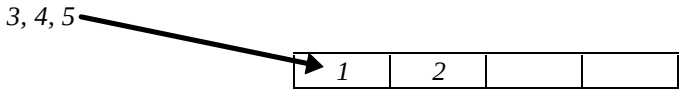
- Add multiple elements, filling up the collection.
- Try to add one element to a full collection.
- The collection is not full. Try to add one too many elements (adding more than one).



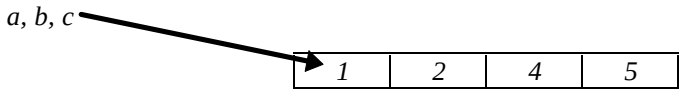
Overwriting the previous contents

These requirements apply whether or not you're adding more than one element at once.

- The new contents leave room for exactly one element. IMPLEMENTATION
  If you're overwriting with one element, that implies the collection can contain at most two elements. More often, you'd be adding several elements to a larger collection, as in this example:

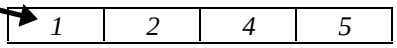


- One or more elements are added, but fewer than in the previous contents.
  Should the previous contents be completely erased? In the following example, should the final "5" be erased or retained? Whichever is appropriate, does your code do it?



Overwriting the previous contents with zero, one, or several elements at once

- New contents have one more element than will fit
  a, b, c, d, e



- New contents just fit
- Zero elements are added.
  Are the old contents cleared? Should they be?

Deleting elements from any position

- The collection has one element. IMPLEMENTATION
- The collection has no elements (nothing to delete).