Array Translation Algorithm

Jouni S. Puuronen

20.4.2023

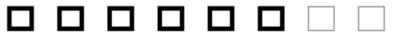
Abstract

We find a container algorithm that is based on performing small translations of arrays, and that supports insert, search and delete operations with $\log(n)$ computational complexity.

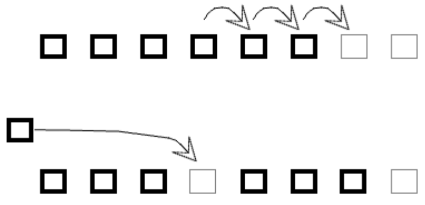
Our objective is to find a container algorithm that supports insert, search and delete operations efficiently. The container must store (key, data) pairs, where the keys are some integer numbers.

We initialize the container by allocating space for an array. We can visualize this space as:

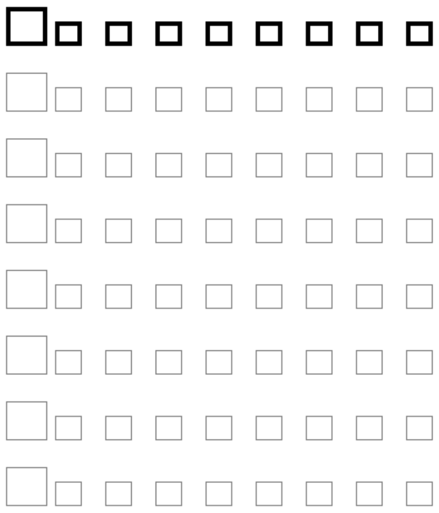
We have to choose some constant that determines the arrays' maximum length, and we can let it be 8 in our example. In the beginning we start maintaining our container as an ordinary sorted array. For example, suppose we are in a situation, where an array is 6 elements long, and 2 places remain unused:



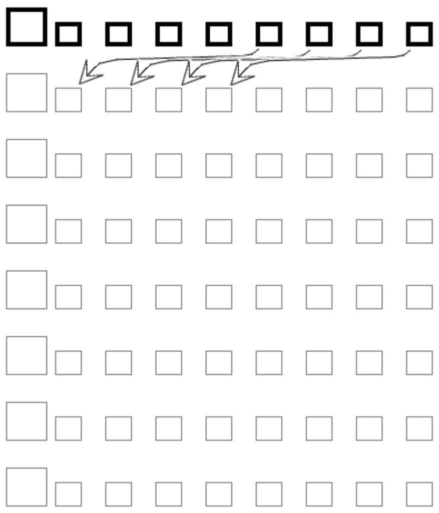
Suppose we want to insert a new element whose key is between the keys of 3rd and 4th elements. We translate the elements from 4th onwards to right, and insert the new element in the created free space:

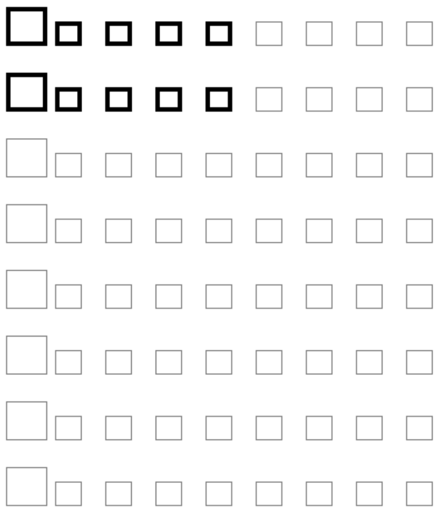


If the array reaches the maximal length, we allocate space for a new array:



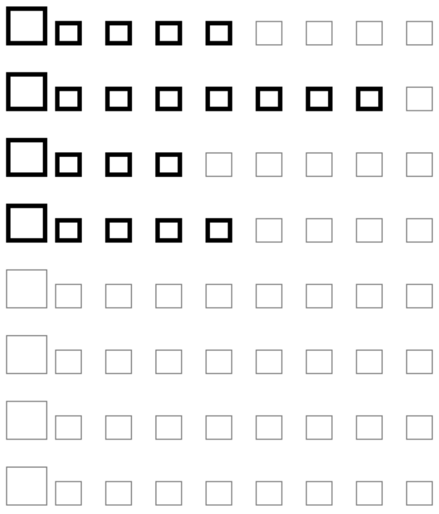
Here the new array is visualized as a vertical column, and the elements of this new array are pointers to horizontal row arrays. First the first row contains the old array. Then we split the array into two pieces, and the second half gets relocated on the next horizontal row array:



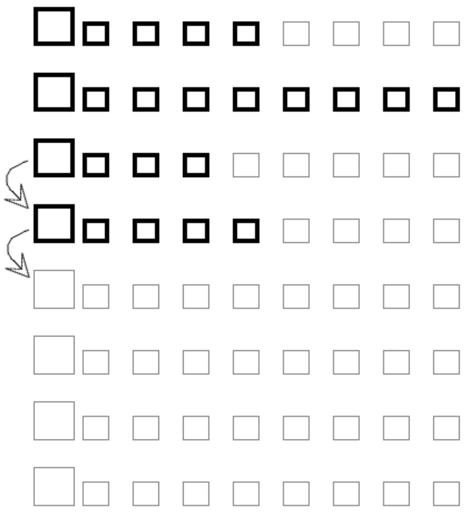


The elements of the vertical column array should contain as keys the first keys of the corresponding horizontal row arrays. Then, if we want to insert a new element into the container, we use these keys first to determine the correct row, and then the correct place in the row.

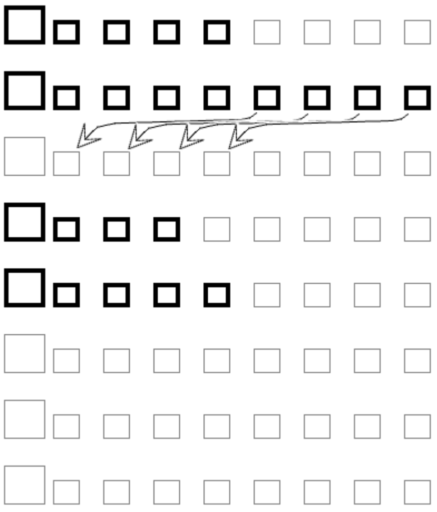
Suppose we have a situation:

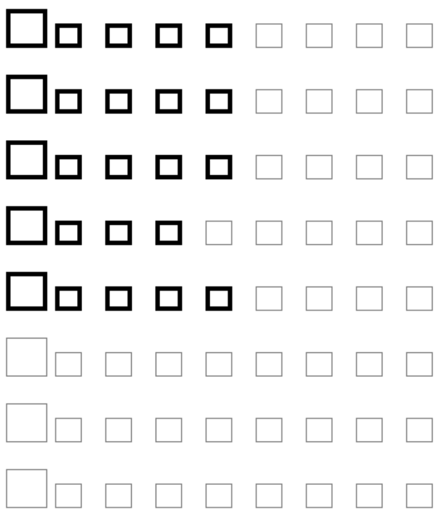


and that we want to add an element on the 2nd row. Then the 2nd row reaches maximal length, so we create some space with vertical translations:

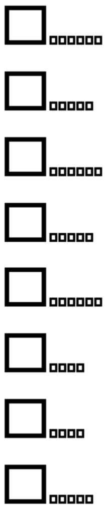


and split the 2nd row into two pieces:

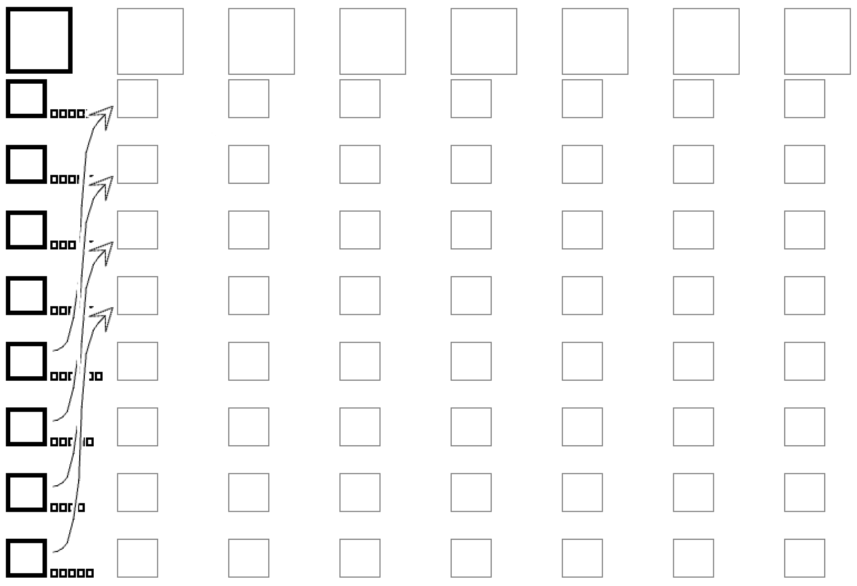


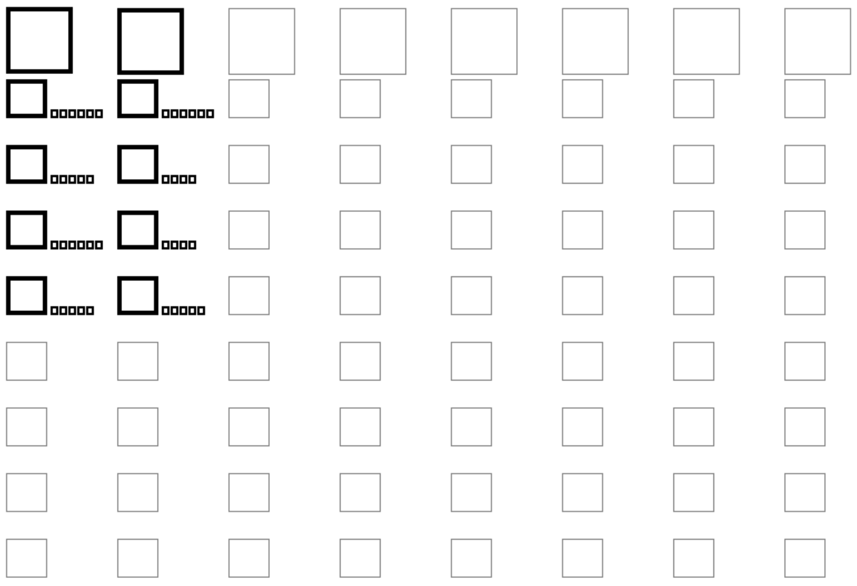


Suppose that the vertical column array eaches maximal length. We can visualize it like this:



In this case we allocate space for a new array, and visualize it as a horizontal row. Elements of this new row will contain pointers to vertical columns arrays. Then we split the full column into two pieces:

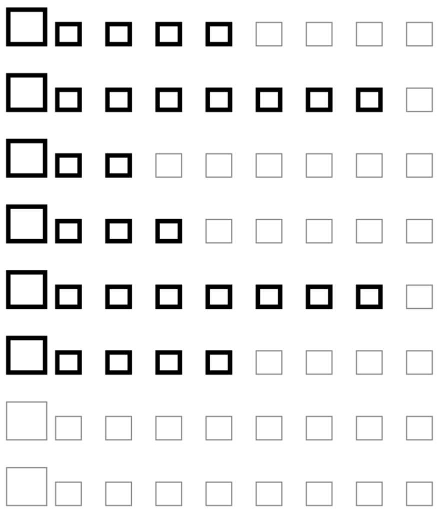




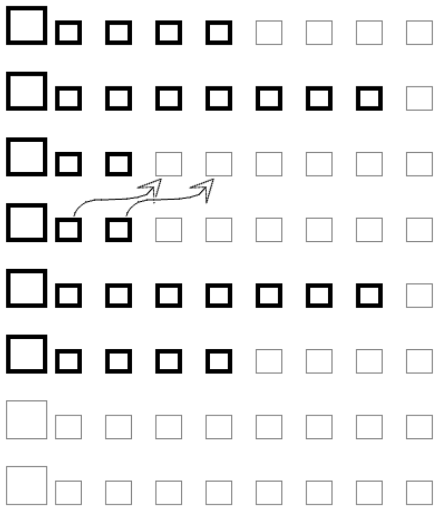
The new array elements will contain as keys the first keys of the corresponding vertical column arrays. If we want to insert a new element into the container, we can use these keys to first find the correct column, then a correct row, and finally the correct place in the row.

These arrays form a tree structure, and we can continue like this indefinitely to arbitrary container sizes.

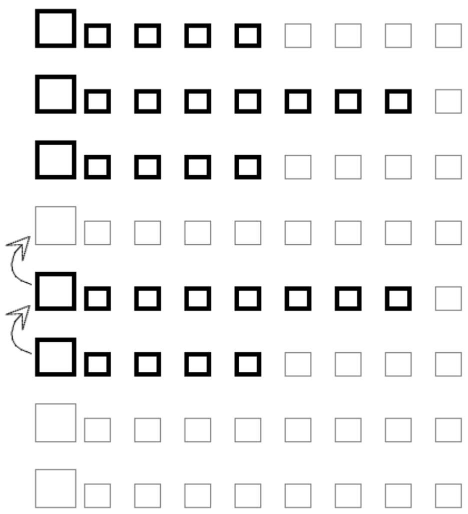
Next, let's have a look at how deleting elements from the container works. We will need some threshold value for the smallest allowed number of elements in two neighboring arrays, and we can let it be 4 in our example. Suppose we have a situation:

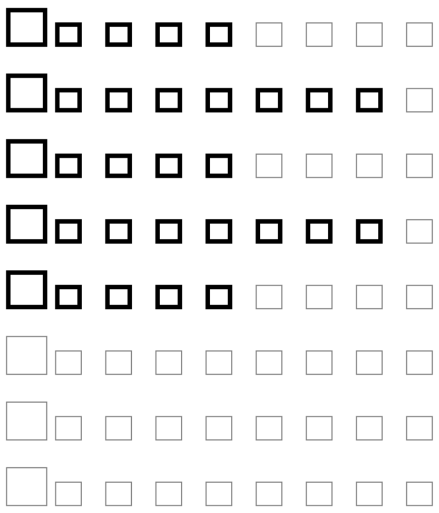


and that we want to delete an element from the 4th row. After the deletion the 4th row and the previous row together have only 4 elements, so the threshold is reached, and we merge the two rows:

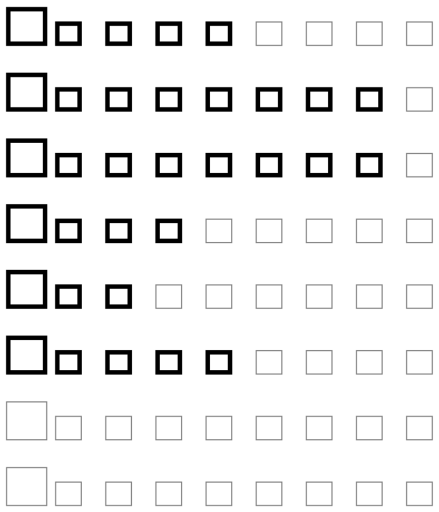


and translate further rows backwards:

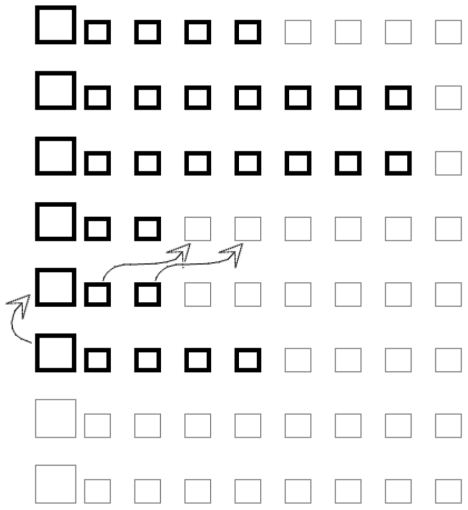




Suppose we instead have a situation:

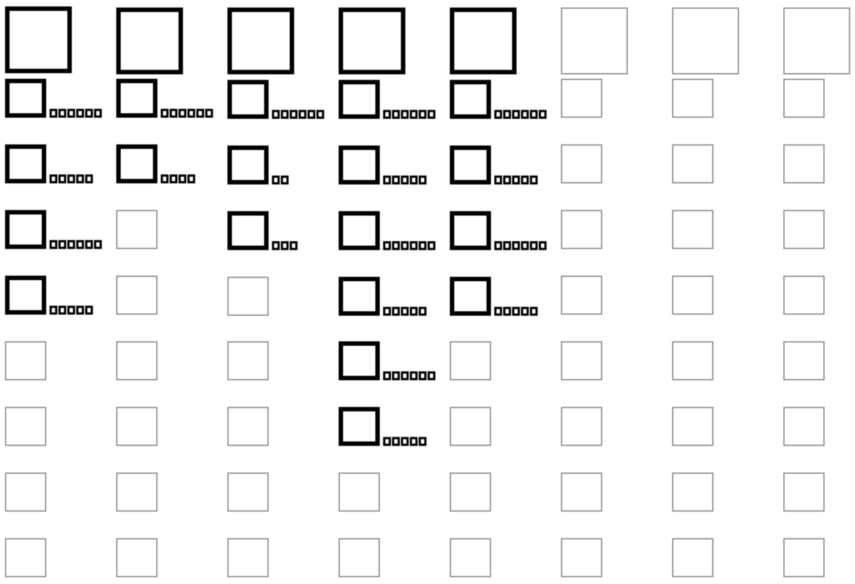


and that we again want to delete an element from the 4th row. This time we recognize that after deletion the 4th row and the next row together have only 4 elements, so the threshold is reached this way. We merge these rows and translate further rows backwards:

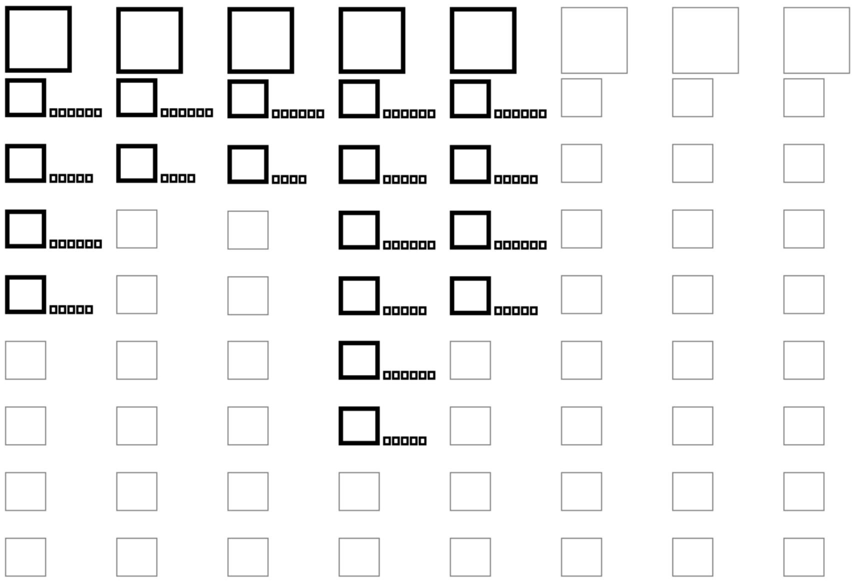


It is also possible that some row becomes empty without reaching the threshold with a neighbor row. These situations can be recognized and dealt with by removing the empty row with a translation of rows.

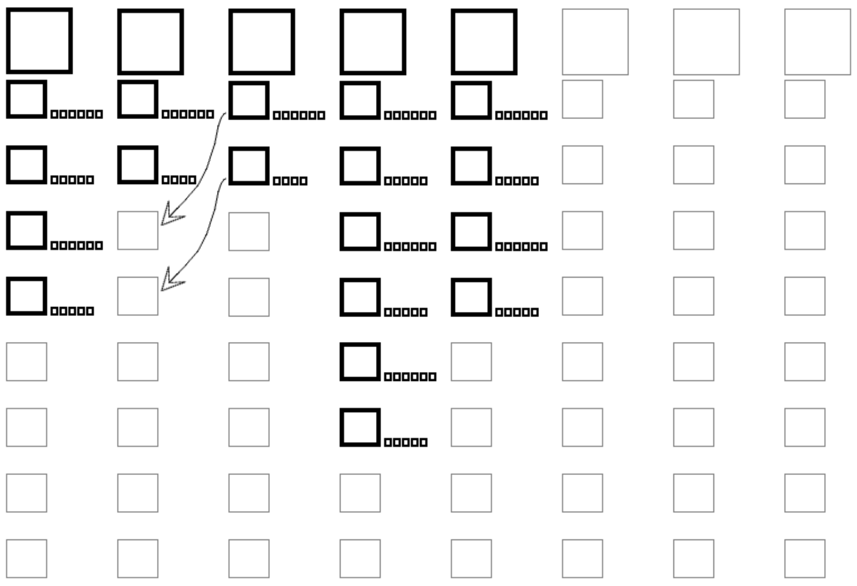
When an element is deleted from a container, we perform recursive search calls to find the element from the tree structure. When we return from the recursive calls, thresholds for merging must be checked after every return. For example, suppose we have a situation:

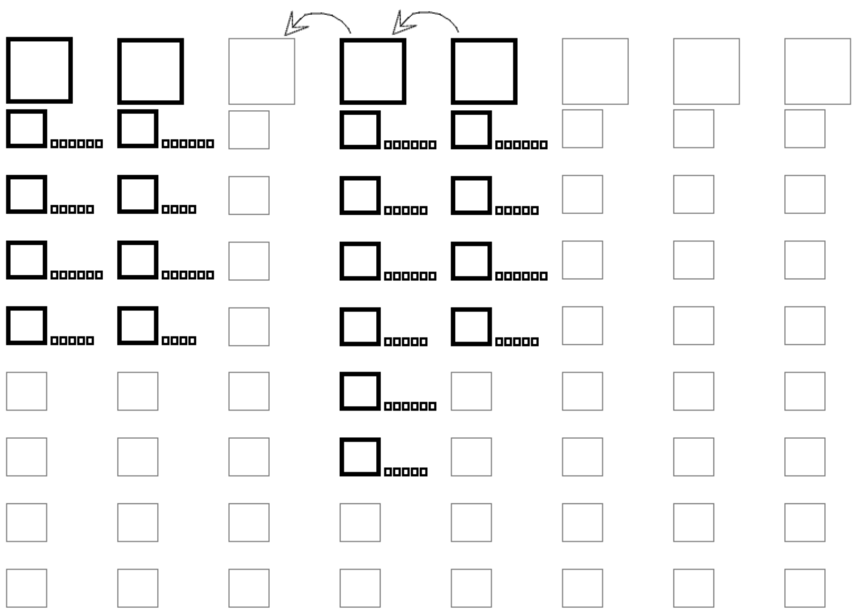


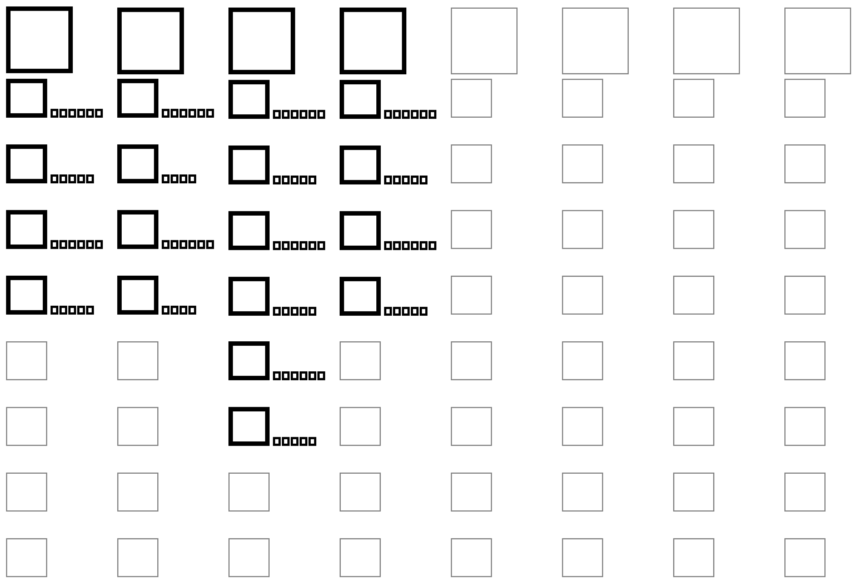
and that we want to delete an element that is in 3rd row of the 3rd column. After the deletion the 3rd row and the previous row have only 4 elements, so they will be merged. Then the situation is:



At this point we must recognize that now the 3rd column and the previous column have only 4 elements, so the threshold was reached in this way too. The columns must be merged:







If the array on the first level of the tree structure contains only one element, the depth of the tree can be reduced by one.

By implementing these ideas we get a container, where insert, search and delete operations have computational complexity proportional to the depth of the tree structure, and where the depth is proportional to the logarithm of the number of elements in the container.