1 The gseq protocol

In this section, we describe the **gseq** (for *generalized sequence*) protocol, allowing client code to dynamically add elements to, and delete elements from a sequence.

1.1 The concept of a position

Generalized sequence use the concept of a "position", which has two different meanings in different contexts.

The first meaning is the position at which an element is located in the sequence. In this context, the position must have a value between 0 and $l-1$, where l is the length of the generalized sequence. This meaning is used when an element is to be deleted (`gseq-delete`) or when an element is accessed (`gseq-element`) or replaced (`(setf gseq-element)`).

The second meaning is the position *position between two elements* (or at one of the extreme ends of the generalized sequence). In this context, the position may have a value between 0 and the length of the sequence inclusive. The position 0 means the beginning of the generalized sequence (before the first element, if any), and the position equal to the length means the end of the generalized sequence (after the last element if any). This meaning is used when an element is to be inserted (`gseq-insert`).

1.2 Performance

Accessing and replacing an element are constant-time operations.

We guarantee linear average complexity of a sequence of insert and delete operations provided that the position of two successive operations in the sequence is bounded.

More specifically, the average complexity of an operation is proportional to the difference between the position of the operation and the position of the previous operation.

Here we actually consider distance modulo the length of the generalized sequence so that the distance between the last position and the first is 1. In particular, the longest possible distance between to operations is half of the number of elements in the generalized sequence.

The implementation will allocate more space than is necessary. Whenever space runs out, we allocate a bigger chunk of memory to hold the elements. The new chunk will be bigger than the old by a *constant factor* (say k) as opposed to by a fixed number of elements. The reason for this behavior is that we want to guarantee the linear average complexity of a sequence of insert and delete operations. Typically, k is somewhere between 1.5 and 2. The default value is $k = 1.5$.

We shrink the space whenever the length of the sequence (the number of elements) is significantly smaller than the available space. By default, the definition of "significantly" is that the ratio of length to size must be less than $1/k^2$ in order for the space to be shrunk, and then it is shrunk by a factor k. Using the default value of k, this means that the amount of wasted space can be as large as the length of the sequence in the worst case, but for a sequence of insert operations, the average wasted space is only 25%.

Applications that store a number of elements that does not vary much, can choose a small value for the k to waste less space. Such applications will have to resize the space relatively rarely so performance will not be affected by small values of k.

The space will not shrink below a minimum size (default 5 elements). The reason for this is to avoid to many resize operations for small sequences. It is probably not reasonable to use a value below around 5, since the bookkeeping information takes up at least this much space. This value is also used for the initial size of the sequence. Applications that will typically store a large number of elements can choose a greater value for the minimum size. Doing so also improves performance since fewer resize operations will have to be executed.

1.3 Protocol classes and functions

⇒ `gseq` [*Protocol Class*]

The protocol class for generalized sequences.

⇒ `:initial-contents` [*Initarg*]
⇒ `:element-type` [*Initarg*]

⇒ `:fill-element` [*Initarg*]

⇒ `:expand-factor` [*Initarg*]

⇒ `:min-size` [*Initarg*]

All instantiable subclasses of `gseq` accept these initargs.

The `:initial-contents` initarg is a sequence (list, vector, string) of objects to be stored in the `gseq` from the start.

The `:element-type` initarg determines the type of the elements of the `gseq` (default is `t`).

The `:fill-element` initarg should be an object that is compatible with the `:element-type` initarg and will be used to fill unoccupied space in the generalized sequence (to help the garbage collector). The default value for this initarg will be supplied by the implementation according to the element-type given. The implementation will test `nil`, `0`, and `#\a`. If none of these values will work, the client must supply a value that is compatible with `:element-type`.

The `:expand-factor` initarg is used to determine the factor by which the available space will be multiplied whenever the space for the sequence is full. Default value is 1.5.

The `:min-size` initarg determines the smallest space allocated to hold elements of the sequence. Default value is 5. It is not reasonable to supply values smaller than 5.

The instance created by make-instance will have a length which is that of the sequence given by `:initial-contents` or 0 if no `:initial-contents` was given.

⇒ `standard-lseq` [*Class*]

The standard instantiable subclass of `lseq`.

⇒ `gseq-length` *gseq* [*Generic Function*]

Return the number of elements in the generalized sequence *gseq*.

⇒ `gseq-error` [*Error Condition*]

The base condition for all conditions that may be signaled by the operations on generalized sequences.

⇒ `gseq-position` [*Error Condition*]

This condition will be signaled by operations that require a position argument whenever that argument is out of range.

⇒ `gseq-insert` *gseq element position* [*Generic Function*]

Insert an element at *position* of the generalized sequence. If *position* is out of range (less than 0 or greater than the length of *gseq*), the `gseq-position` condition will be signaled.

⇒ `gseq-delete` *gseq position* [*Generic Function*]

Delete an element at *position* of the generalized sequence. If *position* is out of range (less than 0 or greater than or equal to the length of *gseq*), the `gseq-position` condition will be signaled.

⇒ `gseq-element` *gseq position* [*Generic Function*]

Return the element at *position* of the *gseq*. If *position* is out of range (less than 0 or greater than or equal to the length of *gseq*), the `gseq-position` condition will be signaled.

⇒ `(setf gseq-element)` *element position gseq* [*Generic Function*]

Replace the element at *position* of *gseq* by *element*. If *position* is out of range (less than 0 or greater than or equal to the length of *gseq*), the `gseq-position` condition will be signaled.

1.4 Stack and queue operations

A `gseq` can be used as a stack or as a queue with very good performance. In this section, we suggest a set of operations to facilitate such use.

⇒ `gseq-push-start` *gseq element* [*Generic Function*]

Insert an element at the beginning of the gseq.

⇒ `gseq-push-end` *gseq element* [*Generic Function*]

Insert an element at the end of the gseq.

⇒ `gseq-pop-start` *gseq* [*Generic Function*]

Pop and return the element at the beginning of the `gseq`

⇒ `gseq-pop-end` *gseq* [*Generic Function*]

Pop and return the element at the end of the `gseq`

⇒ `gseq-rotate` *gseq* `&optional` *(n 1)* [*Generic Function*]

Rotate the elements of the `gseq` so that the element that used to be at position n now is at position 0. With a negative value of n rotate the elements so that the element that used to be at position 0 now is at position n. When the magnitude of n is greater than the length of the `gseq`, the operation wraps around so that it becomes equivalent to the same operation with a value of n modulo the length. When the length of the `gseq` is less than 2, this function does nothing.

2 Implementation of the `gseq` protocol

2.1 Representation

We keep elements in a vector treated as a circular gap buffer. We use the word *position* to refer to the abstract position of an element in a generalized sequence, and the word *index* when we talk about indexes of the gap buffer used in the implementation. We say that an index i is *valid* if $0 \leq i < l$ where l is the size of the vector (the vector is never of size 0, so it is always the case that $l > 0$).

We say that the vector is *full* when it contains as many elements as its length (i.e., the gap has a size of 0), and *empty* when it contains no elements (i.e., the gap is the same size as the vector).

There are three different possible configurations of the gap with respect to the data. Figure 1 shows the case where both the gap and the data are contiguous. Figure 2 shows the case where the data is not contiguous. Finally, figure 3 shows the case where the gap is not contiguous.

The implementation of a generalized sequence allows for the first element to be located at any valid index of the vector. For that reason, we need an index (called `data-start`) which when the vector is not empty indicates the index of the element with a position of 0. `data-start` is always a valid index

A *positional index* is an index in the vector that corresponds to a position in the generalized sequence, except that the positions corresponding to the

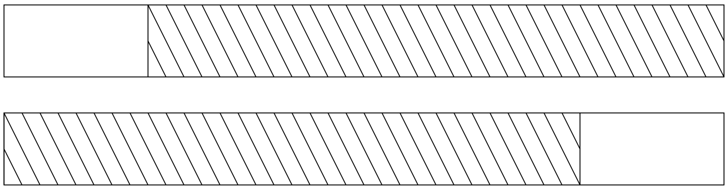

Figure 1: Gap and data are both contiguous

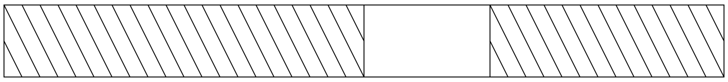

Figure 2: Data is not contiguous

very beginning and the very end of the generalized sequence both map to the index `data-start`. A positional index is either the value of `data-start` or an index of a user object in the vector. The only case where the index `data-start` does not contain any object is when the generalized sequence is empty.

We introduce two different indexes (always valid as well), `gap-start` and `gap-end`. The `gap-start` index is the first index of the gap. When the vector is neither full nor empty, the `gap-start` is always an index containing no element, such that the previous index does contain an element, or 0 if no such index exists. The `gap-end` index is the first index beyond the gap. When the vector is neither full nor empty, the `gap-end` is always an index containing an element, such that the previous index does not contain an element, or 0 if no such index exists. Notice that for certain configurations `gap-start` is smaller than `gap-end` and for certain other configurations, the reverse is true.

When the vector is full or empty, `gap-start` and `gap-end` are always equal. When the vector is empty, `gap-start` and `gap-end` are always equal to

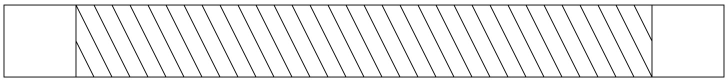

Figure 3: Gap is not contiguous

data-start.

Since we cannot always tell from the values of gap-start and gap-end how many elements there are, we need to keep track of this information separately. For that purpose, we introduce another value named nb-elements.

2.2 Computing and index from a position

Step one in inserting or deleting an element is to determine an index corresponding to the position. Here is how it is done: we compute a value s which is equal to gap-start if gap-start is greater than or equal to data-start. Otherwise s is equal to gap-start plus the size of the vector. The position is added to data-start, giving the value i. If i is greater than or equal to s, the size of the gap is added to i. Finally, if i is greater than or equal to the length of the vector, the length of the vector is subtracted from i (prove that the result is always a positional index). The size of the gap is best computed as the difference between the length of the vector and nb-elements. Call this final value of i the *hot spot*.

2.3 Moving the gap to the right place

After determining the index from a position, we need to determine whether the gap is in the right place. This is the case if and only if the hot spot is equal to gap-end. If that is not the case, we need to move the gap.

There is a case when it is particularly simple to move the gap, namely when the vector is full. In that case, we can just assign both gap-start and gap-end to the value of the hot spot.

There are two ways of getting gap-end to be equal to the hot spot either move everything to the left of the hot spot even further left, or everything to the right of the hot spot (including the hot spot itself) even further right. We always do the one that requires the fewest elements to be moved. One solution will require fewer than half the elements to be moved and the other one at least half.

Moving to the right will require that a number of elements equal to the difference between gap-start and the hot spot to be moved, provided that gap-start is greater than the hot spot. If gap-start is smaller than the hot spot, it is that difference plus the size of the vector. We check whether that value is smaller than half of `nb-elements`.

Moving the elements requires one, two, or three calls to `replace`.

2.3.1 Moving elements to the left

Let us first consider the case of moving elements to the left.

Case 1: If the entire contiguous gap is to the left of the hot spot (as in the upper half of figure 1 or as in figure 2 with the hot spot to the right of the gap), a single call is required.

Case 2:A single call is also required if the highest valid index is inside the gap (as in the lower part of figure 1 and in figure 3) provided that the number of elements to be moved is no greater than the part of the gap that is flush right in the vector.

Case 3: Two calls are needed if the highest valid index is inside the gap (as in the lower part of figure 1 and in figure 3), but the number of elements to be moved is greater than the part of the gap that is flush right in the vector. The first call will fill the part of the gap that is flush right in the vector, giving the situation of the upper half of figure 1. The second call will be as in case 1 above.

Case 4: Two calls are also needed if the data is not contiguous (as in figure 2) and the entire contiguous gap is to the right of the hot spot, but the number of elements to the left of the hot spot (i.e., the index of the hot spot before the move) is no greater than the size of the gap. The first call will move everything to the right of the gap so that the gap will be flush right as in case 2 above. The second call will move the remaining elements.

Case 5: Three calls are needed if the data is not contiguous (as in figure 2) and the entire contiguous gap is to the right of the hot spot, but the number of elements to the left of the hot spot (i.e., the index of the hot spot before the move) is greater than the size of the gap. The first call will move the gap flush right, creating the situation of case 3 above (which then requires another two calls).

2.3.2 Moving elements to the right

Let us now consider moving elements to the right.

Case 1: If the entire contiguous gap is to the right of the hot spot (as in the lower half of figure 1 or as in figure 2 with the hot spot to the left of the gap), a single call is required.

Case 2:A single call is also required if the index 0 is inside the gap (as in the higher part of figure 1 and in figure 3) provided that the number of elements to be moved is no greater than the part of the gap that is flush left in the vector.

Case 3: Two calls are needed if index 0 is inside the gap (as in the lower part of figure 1 and in figure 3), but the number of elements to be moved is greater than the part of the gap that is flush left in the vector. The first call will fill the part of the gap that is flush left in the vector, giving the situation of the lower half of figure 1. The second call will be as in case 1 above.

Case 4: Two calls are also needed if the data is not contiguous (as in figure 2) and the entire contiguous gap is to the left of the hot spot, but the number of elements to the right of the hot spot (i.e., the index of the hot spot before the move) is no greater than the size of the gap. The first call will move everything to the left of the gap so that the gap will be flush left as in case 2 above. The second call will move the remaining elements.

Case 5: Three calls are needed if the data is not contiguous (as in figure 2) and the entire contiguous gap is to the left of the hot spot, but the number of elements to the right of the hot spot is greater than the size of the gap. The first call will move the gap flush left, creating the situation of case 3 above (which then requires another two calls).

2.4 Increasing the size of the vector

We increase the size of the vector whenever it is full and another element needs to be added.

When this call is made, `gap-start` and `gap-end` have the same value. We must preserve the position of the gap in the new vector.

A new vector with the size of the current one multiplied by `size-multiplier` is first allocated.

Next, we copy (using a single call to `replace`) all elements before the gap to the start of the new vector. Then we copy (using another single call to `replace`) all elements after the gap to the end of the new vector. The value of `gap-end` is incremented by the difference in size of the two vectors, as is `data-start` if it was greater than or equal to `gap-end`.

2.5 Decreasing the size of the vector

A new vector with the size of the current one divided by `size-multiplier` is first allocated.

Next, we copy (using a single call to `replace`) all elements before the gap to the start of the new vector. Then we copy (using another single call to `replace`) all elements after the gap to the end of the new vector. The value of `gap-end` is decremented by the difference in size of the two vectors, as is `data-start` if it was greater than or equal to `gap-end`.

2.6 Inserting an element

The insertion operation is given a position. The semantics of the insertion operation require that all elements having a position greater than or equal to the one given as argument to the insertion operation be "moved to the right" i.e., that they have their positions incremented by one.

After moving the hot spot to the right place, the value of `gap-end` is the index corresponding to the position supplied by the call. It should be noted that the same index will result from a position of 0 and from a position equal to the current length of the sequence. We shall treat all calls with a position greater than 0 the same way, and a call with a position of 0 differently.

But first, we need to make sure the vector is not full. If it is, we call the function to increase its size.

For a position greater than 0, we shall suspect that the next call will be with a position one greater than this one. In order not to have to move elements a second time for this case, we place the element to be inserted at the index of `gap-start` and then increment `gap-start`. If this operation gives a `gap-start` equal to the size of the vector, then it is set to 0.

For a position of 0 we might suspect that the next call will also be with a position of 0. If that is not the case, we will make the mistake only once. In that case, `gap-end` and `data-start` are the same index. In this case, we start by decrementing `gap-end` and `data-start`. If this operation gives a value less than 0, then the value is set to the highest valid index. Finally, the element at the position of `data-start` is set to the one supplied by the call.

Finally, We increment `nb-elements`.

2.7 Deleting an element

After moving the hot spot to the right place, we need to delete the element at `gap-end`. We do this by replacing it by the `fill-element` so as to avoid holding on to it in case it is no longer referenced. Then we increment `gap-end` (and also `data-start` if the were the same). We decrement `nb-elements`.

Finally, we check whether the size of the vector should be decreased.

2.8 Stack and queue operations

The stack and queue operations are implemented very efficiently. The `push` and `pop` operations simply call the corresponding `insert` and `delete` operations.

The `rotate` operation just modifies the `data-start` slot.

3 The `lseq` protocol

A *locator sequence* (lseq for short) is like a generalized sequence, but it also keeps around a bunch of "locators".

3.1 The concept of a locator

A locator is an object that corresponds to a position in the sequence. Whenever an element is inserted before the position of a locator, the position of the locator will be incremented. Conversely, whenever an element is deleted from a position below that of a locator, the position of the locator is decremented.

3.2 Mixing `lseq` and `gseq` operations

The user can freely mix editing operations from the `lseq` and the `gseq` protocol. When an editing operation from the `gseq` protocol is used on an `lseq` object, the locators of the `lseq` object are updated accordingly.

3.3 Performance

We assume that the number of locators of a particular sequence is relatively small. Although under normal circumstances simple operations like insert and delete will not take a time proportional to the number of locators, it is possible that locators need to be updated from time to time. We do not promise any bound on complexity for a large number of locators.

3.4 Protocol classes and functions

⇒ `lseq` [*Protocol Class*]

This is a subclass of `gseq`.

⇒ `standard-lseq` [*Class*]

The standard instantiable subclass of `lseq`

⇒ `locator` [*Protocol Class*]

The protocol class for all locators

⇒ `lseq` [*Initarg*]

This initarg determines the lseq with which the locator is associated.

⇒ `standard-locator` [*Class*]

The standard instantiable subclass of `lseq`.

⇒ `lseq` *locator* [*Generic Function*]

Return the underlying locator sequence of the locator given as argument.

⇒ `clone-locator` *locator* [*Generic Function*]

Create a locator that is initially at the same location as the one given as argument.

⇒ `locator-position` *locator* [*Generic Function*]

Return the position of the locator. This operation may or may not be efficient depending on how the lseq is implemented.

⇒ `at-beginning-p` *locator* [*Generic Function*]

Return true if the locator is at the beginning of the lseq (i.e., if it has a position of 0). This operation is guaranteed to be executed in O(1) time.

⇒ `at-beginning` [*Error Condition*]

This condition is signaled whenever an attempt is made to move a locator beyond the beginning of the lseq.

⇒ `at-end-p` *locator* [*Generic Function*]

Return true if the locator is at the end of the lseq (i.e., if it has a position equal to the length of the lseq). This operation is guaranteed to be executed in O(1) time.

⇒ `at-end` [*Error Condition*]

This condition is signaled whenever an attempt is made to move a locator beyond the end of the lseq.

⇒ `forward` *locator* `&optional` *(n 1)* [*Generic Function*]

More the locator forward n positions. If the locator is at a position greater than len - n where len is the length of the lseq, then the condition at-end will be signaled.

⇒ `backward` *locator* `&optional` *(n 1)* [*Generic Function*]

More the locator backward n positions. If the locator is at a position less than n, then the condition at-beginning will be signaled.

⇒ `incompatible-object-type` [*Error Condition*]

This condition is signaled whenever an attempt is made to insert an object of an incompatible type into an lseq.

⇒ `insert-elem` *object locator* [*Generic Function*]

Insert an object after the locator and then advance the locator by one position. Locators initially located at positions strictly higher than that of the locator argument will have their positions incremented by one. Other locators are unaffected.

If the type of the object does not match the type accepted by the underlying lseq, the `incompatible-object-type` condition is signaled.

⇒ `insert-sequence` *sequence locator* [*Generic Function*]

The effect is the same as if each element of the sequence were inserted using the insert generic function.

⇒ `delete-elem` *locator* `&optional` *(n 1)* [*Generic Function*]

Delete n objects after the locator. A locator initially located at a position p strictly higher than that of the locator argument will have its position decremented so that its new position is (max pl (- p n)) where pl is the initial position of the locator argument.

⇒ `erase-elem` *locator* `&optional` *(n 1)* [*Generic Function*]

Delete n objects before the locator. ...

A sequence of insert, delete, and erase operations with is guaranteed to be efficient if the locations of successive operations are not too far apart as measured by the shortest distance of the lseq viewed as a circular list. Thus, the beginning and the end of the lseq are considered close, which makes it feasible to use the lseq to implement an efficient queue.

⇒ `with-editing-operations` *locator* `&body` *body* [*Macro*]

This macro can be used to group a bunch of editing operations (insert, delete, erase) into a body. The sequence remains locked for the duration of invocation. Other locators of the underlying lseq are updated only after the last operation has been completed, thus making it more efficient to use this macro than to use individual editing operations.

⇒ `elem-at` *locator* [*Generic Function*]

Return the element immediately after the locator. If the locator is at the end, an at-end condition will be signaled.

⇒ `(setf element)` *locator* [*Generic Function*]

Replace the element immediately after the locator by the one given as argument.

4 Implementation of the `lseq` protocol

Locators are stored as lists of weak references so that they can be recycled when no longer referenced by client code.

A locator contains its *index in the vector* as opposed to its *position in the sequence*. This method avoids most updates of locators at each insert and delete operation. Most locators need only be updated whenever the gap moves.

After a delete operation, locators with indexes equal to the old value of `gap-end` need to be updated to the new value of `gap-end`, including the locator that was used in the call.

After an insert operation, only the locator used in the call needs to be incremented. All other locators remain the same.

Mixing of `lseq` and `gseq` editing operations is possible thanks to an internal protocol for moving the gap. The `lseq` code uses :before, :after, and :around methods on the `gseq` editing operations as well as on the code for moving the gap to update the locators accordingly. This way, an `lseq` editing operation translates directly to a `gseq` editing operation with no extra code.