# Common Lisp Container Library

## 1. ABSTRACT

Common Lisp ships with a set of powerful built in data structures including the venerable list, full featured arrays, and hash-tables. The Common Lisp Container Library (CL-Containers) enhances and builds on these structures in two ways:

1. By adding containers that are not available in native Lisp (for example: binary search trees, red-black trees, sparse arrays and so on).
2. By standardizing the container interface so that they are simpler to use and so that changing design decisions becomes significantly easier.

This document discusses the design motivations of **CL-Containers** and documents the structure and contents of the library in detail.

## 2. MOTIVATION

A significant portion of almost any non-trivial programming effort goes into the manipulation of structured data. Many data structures can be viewed as containers. These may be ordered or unordered, associative (indexed) or sequential and so forth. The power and beauty of some programming languages derives in no small part from their ability to handle container classes well (Smalltalk comes to mind) and other languages have gone to great lengths to incorporate flexible containers into their frameworks (C++'s STL for example). Although Common Lisp includes a number of "containers" (hash-tables, lists and arrays) there are many useful data structures that are not built-in (e.g., binary search trees) and the existing containers have a fragmented interface that shows Lisp's peripatetic evolution.

The Common Lisp Container Library (**CL-Containers**) extends Lisp in two ways: it adds new container functionality and, by standardizing container interfaces, it makes using them significantly easier. The standard interface also lends itself to more a flexible design and development process since data structure decisions are easier to change as the program evolves. **CL-Containers** thereby expends the power of Lisp and makes it an even better tool for both rapid prototyping *and* production code.

The remainder of this document will cover the container types available in **CL-Containers**, the methods applicable to these containers, information about the internal design of **CL-Containers**and examples of containers in use. As will become evident, **CL-Containers**'s design borrows from the Standard Template Library, Smalltalk and existing Lisp idioms wherever possible.

## 3. BASIC CONTAINER KINDS

The containers in **CL-Containers** can be divided into three basic types: Ordered, Unordered and Associative. The main examples of each type are given in table **??**. The three container types can be distinguished as follows: Ordered containers store things in (some) order. The order may be based on how items were inserted into the container or it may depend on an explicit sorting. Unordered containers also store things and share much of Ordered containers interface. However, the items in an Unordered container do not maintain any particular arrangment. Finally, Associative containers store items *associated* with some index or key. The key may be simple (for example, a one-dimensional array indexed by an integer) or complex (for example, a nested hash table indexed by color, size and object class).

Before describing the methods and implementation of **CL-Containers** in greater detail, a few examples should generate a better sense of how it fits into Lisp. FIrst, we use **CL-Containers** for a job that would often be accomplished using an associated list: linking one small group of things with another.

```
;; a lookup table associating tokens with notes.
(defvar *tokens->notes*
        (make-container 'alist-container))

(defun assign-note (token note)
  "Associate a token with a note."
  (setf (item-at *tokens->notes* token) note))

(defun clear-note-assignments ()
  "Clear all associations."
  (empty! *tokens->notes*))
```

| Category | Container | Description |
|---|---|---|
| Unordered | | |
| | Bag-container | Unsorted collection of items |
| | Set-container | Unsorted collection of items ignoring duplicates |
| Ordered | | |
| | list-container | Equivalent to a Lisp list |
| | stack-container | A standard Stack |
| | queue-container | A standard Queue |
| | priority-queue-container | A standard priority queue |
| | vector-container | An extendable one dimensional array (like STL's vector) |
| | binary-search-tree | A standard Binary Tree |
| | red-black-tree | A balanced binary tree |
| | ring-buffer | A standard ring-buffer (useful for fixed size queues) |
| Associative | | |
| | alist-container | Equivalent to a Lisp assoc list |
| | array-container | Equivalent to a Lisp array |
| | associative-array-container | (not-implemented) |
| | associative-container | Equivalent to using nested Lisp hash-tables. |
| | simple-associative-container | Equivalent to a Lisp hash-table |
| | sparse-array-container | An array that only allocates space when necessary. |

**Table 1: CL-Containers main container types**

The advantage of **CL-Containers** is that if we later learn that our association must handle dozens or hundreds of objects, we can switch to a more efficient hash table based association by making only one change to the code:

```
;; a lookup table associating tokens with notes.
;; was 'alist-container
(defvar *tokens->notes*
        (make-container 'simple-associative-container))
```

A second example comes from a Hierarchical Agglomerative Clustering (HAC) application. At each step in the clustering, we merge the pair of items that "goes together" best. We store the candidate pairs in a priority queue ordered by their distance (closer items are more likely to result in a successful merge). Of course, there are many sorts of priority queues and the one to use will depend on actual items being clustered. **CL-Containers** lets us hide the queue implementation and defer the choice of implementation to the last possible moment: when we actual run the clustering algorithm.

```
;; The priority-queue-on-container container
;; lets you specify which container type to use
;; in storing the items in the queue (as long
;; as it supports the necessary operations).
(make-container
  'priority-queue-on-container
  :key #'group-average-score
  :test #'(lambda (candidate1 candidate2)
             (compute-distance candidate1 candidate2))
  :container-type 'binary-search-tree)
```

In general, a heap-based or `binary-search-tree` based container may be the best to use for the priority-queue. However, if we believe that many of the items will have the same distance from one another, then a `red-black-tree` may be the most appropriate because it will stay balanced. If too many of the same items are stored in a `binary-search-tree`, then it will degenerate into essentially a list!

### 3.1 Container Mixins

Tables **??** and **??** list the main mixins used by **CL-Containers** . A typical *concrete* container (i.e., one that you would actually use in your application) will inherit from several of these mixins. For example, binary-search-tree inherits from sorted-container-mixin, findable-container-mixin and iteratable-container-mixin. As is generally true of CLOS class design, the mixin hierarchy provides generic implementation of common operations and helps to make clear the abilities of each specific container class. For example, the class hierarchy indicates that the binary-search-tree above sorts its contents, supports a fast find operation and can be iterated over. Effort has been made to split apart the various container operations in such a way that creating new containers is relatively easy.

## 4. BASIC CONTAINER METHODS

Table **??** is a list of many of the methods defined on containers. Many of these are not defined for all containers. For example, insert-item does not make any sense for associative containers. More information on exactly which methods are defined for which containers is included below.

## 5. USING CL-Containers

**CL-Containers** must be loaded into your Lisp environment before it can be used. This is accomplished by loading the file containers.lisp.

To use **CL-Containers** in your code, use make-container instead of the traditional Lisp methods (e.g., make-hash-table) then use methods like insert-item, search-for-item, empty! and so on as necessary. The methods you use will

| Class | Description |
|---|---|
| abstract-container | Inherited by all container classes, this is a good place to put those pesky superclasses you need everyone to inherit. |
| typed-container-mixin | Elements have a specified (Lisp) type set a creation time. |
| bounded-container-mixin | Containers are created with a fixed size. Supports total-size. |
| indexed-container-mixin | Container elements can be accessed via an index. Supports item-at. |
| initial-element-mixin | Container supports initial-element and initial-element-fn. |
| initial-contents-mixin | Containers can be created with initial-contents (like a Lisp array). |
| iteratable-container-mixin | Elements of the container can be iterated over. Supports iterate-container. |
| searchable-container-mixin | The container can be searched (not necessarily quickly). Supports search-for-item and search-for-match. |
| findable-container-mixin | The container incorporates a pre-defined search function into its structure. The function must be specified at creation time. Supports find-item. |

**Table 2: CL-Containers container mixins**

obviously depend on the purpose of your containers. You may find the additional examples including in the extras directory helpful as you get started.

## 6. STUFF THAT NEEDS DOCUMENTATION

**CL-Containers** includes some powerful and funky iterators constructs. **CL-Containers** all container types and methods **CL-Containers** installation **CL-Containers** testing

## 7. FUTURE WORK

**CL-Containers** provides a useful set of data structure abstactions that enable faster prototyping and more flexible design. As always, more remains to be done. For **CL-Containers** , this more falls into the following categories: additional container types, better integration with Lisp types and improvements to the existing containers.

**CL-Containers** is missing some containers types (e.g., `splay-tree`) In addition, some of the existing containers fail to implement (or implement poorly) operations that they ought. To ameiliorate these problems, we are investigating incorporating STL-like iterators into the design of **CL-Containers**. (This has been partly completed)

**CL-Containers**fulfils most of its goal's admirably. It provides Lisp with useful and non-intrusive container support and it makes Lisp a better tool for both rapid prototyping and production development.

| Class | Description |
|---|---|
| ordered-container-mixin | The elements in the container are ordered. Supports first-item, item-after, item-before, last-item, delete-first, delete-list, and insert-item. |
| sorted-container-mixin | The elements in the container are sorted by a sort function which must be specified at creation time. |
| keyed-container-mixin | The container has a key functin used to specify an accessor to its elements. Used by findable-container-mixin and sorted-container-mixin. |
| uses-contents-mixin | The elements in the container are stored in a slot named contents. This mixin and its three sub-classes (see below) abstract a group of common operations. |
| contents-as-array-mixin | The contents of the container are stored in a Lisp array. |
| contents-as-list-mixin | The contents of the container are stored in a Lisp list. |
| contents-as-hashtable-mixin | The contents of the container are stored in a Lisp hash table. |

**Table 3: More CL-Containers container mixins**

| Class | Description |
|---|---|
| abstract-queue | The container supports the Queue interface. Supports enqueue and dequeue. |
| abstract-stack | The container supports the Stack interface. Supports pop-item and push-item. |

**Table 4: CL-Containers Generic Container Templates**

| Method Name | Description |
|---|---|
| Generic Operations | |
| make-container | create a new container |
| size | returns the number of items currently in the container |
| empty-p | returns **t** if the container is empty |
| empty! | removes everything from the container |
| Insertion and Deletion | |
| insert-item | Adds a new item to a container. |
| delete-item | Removes an item from a container. |
| item-at | returns the item at a specified index (settable). This is only applicable for indexed-containers. |
| Search and Iteration | |
| iterate-container | calls a function on each item in a container |
| search-for-item | hunts for an item in a container (generally by iterating over all of the items. |
| find-item | finds an item in a container using the containers underlying structure (usually faster than search-for-item if the container supports it) |
| keys-of-container | Returns a list of all of the indexes of an associative container. |
| iterate-key-value | Similar to iterate-container, this method calls a function for each (key, value) pair in an associative container. It is analogous to maphash. |
| Navigation | |
| first-item | returns the first item in a container (settable) |
| last-item | returns the last item in a container (settable) |
| successor | Returns the item after the specified item in the container. |
| predecessor | Returns the item before the specified item in the container. |
| Specialized Operations | |
| total-size | returns the maximum number of items that a container can contains (only applicable for bounded-containers) |
| dequeue | synonym for delete-first (for queues) |
| enqueue | synonym for insert-item (for queues) |
| pop-item | synonym for delete-first (for stacks) |
| push-item | synonym for insert-item (for stacks) |

**Table 5: CL-Containers Container methods**