10 Lists

10.1 Introduction

Lists are simple linearly ordered collections. Some things we refer to in everyday life as lists, such as shopping lists or laundry lists, are really sets because their order doesn't matter. Order matters in lists. A to-do list really is a list if the tasks to be done are in the order in which they are supposed to be completed (or some other order).

> **List**: An ordered collection.

Because order matters in lists, we must specify a location, or index, of elements when we modify the list. Indices can start at any number, but we will follow convention and give the first element of a list index 0, the second index 1, and so forth.

10.2 The List ADT

Lists are collections of values of some type, so the ADT is *list of T*, where *T* is the type of the elements in the list. The carrier set of this type is the set of all sequences or ordered tuples of elements of type *T*. The carrier set thus includes the empty list, the lists with one element of type *T* (one-tuples), the lists with two elements of type *T* (ordered pairs), and so forth. Hence the carrier set of this ADT is the set of all tuples of type *T*, including the empty tuple.

There are many operations that may be included in a list ADT; the following operations are common. In the list below, *s* and *t* are lists of *T*, *i* is an index, and *e* is a *T* value, *n* is a length, and the value *nil* is a special value that indicates that a result is undefined.

> *size*(*t*)—Return the length of list *t*.
>
> *insert*(*t*,*i*,*e*)—Return a list just like *t* except that *e* has been inserted at index *i*, moving elements with larger indices up in the list if necessary. The precondition of this operation is that *i* be a valid index position: $-size(t) \le i \le size(t)$. When *i* is negative, elements are counted backwards from the end of the list starting at -1, and *e* is inserted after the element; when *i* is 0, *e* is inserted at the front of the list; and when *i* is *size*(*t*), *e* is appended to the end of the list.
>
> *delete_at*(*t*,*i*)—Remove the element at index *i* of *t* and return the resulting (shorter) list. The precondition of this operation is that *i* be a valid index position, with negative indices indicating a location relative to the end of the list: $-size(t) \le i < size(t)$.

$t[i]$—Return the value at index i of t. Its precondition is that i be a valid index position: $-size(t) \leq i < size(t)$.

$t[i]=e$—Replace the element at index i of t and return the resulting (same sized) list. The precondition is that i be a valid index position: $-size(t) \leq i < size(t)$.

index(t,e)—Return the index of the first occurrence of e in t. If e is not in t, return *nil*.

slice(t, i, n)—Return a new list that is a sub-list of t whose first element is the value at index i of t and whose length is n. The precondition of this operation is that i is valid and n is non-negative: $-size(t) \leq i < size(t)$ and $0 \leq n$. The sub-list does not extend past the end of t no matter how large n is.

$t==s$—Return true if and only if s and t have the same elements in the same order.

As with the ADTs we have studied before, an object-oriented implementation of these operations as instance methods will include the list as an implicit parameter so the signatures of these operations will vary somewhat when they are implemented.

10.3 The List Interface

A List interface is a sub-interface of Collection, which is a sub-interface of Container, so it has several operations that it has inherited from its ancestors. The diagram in Figure 1 shows the List interface.

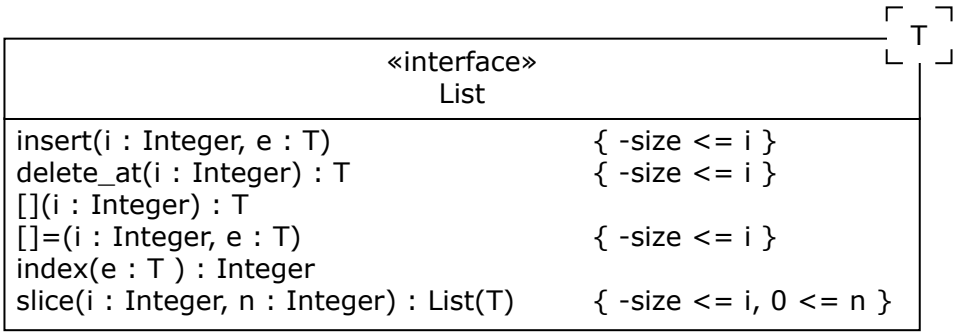

Figure 1: The List Interface

Note that a template parameter is used to generalize the interface for any element type. Note also that preconditions have been added, though they differ slightly from those in the ADT. In particular, when querying a list, if an index is out of bounds, then nil is returned. When inserting or replacing an element in a list, if the index is beyond the upper bound, then the list is extended with nil values until it is large enough, and then the operation is performed. If delete_at() is given an out of bounds index, nothing is changed.

10.4 An Example of Using Lists

Suppose a calendar program has a to-do list whose elements are ordered by precedence, so that the first element on the list must be done first, the second next, and so forth. The items in the to-do list are all visible in a scrollable display; items can be added, removed, or moved around in the list freely. Mousing over list items displays details about the item, which can be changed. Users can ask to see or print sub-lists (like the last ten items on the list), or they can ask for details about a list item with a certain precedence (like the fifth element).

Clearly, a List is the right sort of container for to-do lists. Iteration over a List to display its contents is easy with an internal or external iterator. The insert() and delete_at() operations allow items to be inserted into the list, removed from it, or moved around in it. The [] operation can be used to obtain a list element for display during a mouse-over event, and the []= operation can replace a to-do item's record if it is changed. The slice() operation produces portions of the list for display or printing, and the index() operation can determine where an item lies in the list.

10.5 Contiguous Implementation of the List ADT

Lists are very easy to implement with arrays. Static arrays impose a maximum list size, while dynamic arrays allow lists to grow without limit. The implementation is similar in either case. An array is allocated to hold the contents of the list, with the elements placed into the array in order so that the element at index *i* of the list is at index *i* of the array. A counter maintains the current size of the list. Elements added at index *i* require that the elements in the slice from *i* to the end of the list be moved upwards in the array to make a "hole" into which the inserted value is placed. When element *i* is removed, the slice from *i*+1 to the end of the list is copied down to close the hole left when the value at index *i* is removed.

Static arrays have their size set at compile time so an implementation using a static array cannot accommodate lists of arbitrary size. In contrast, an implementation using a dynamic array can allocate a larger array if a list exceeds the capacity of the current array during execution. Reallocating the array is an expensive operation because the new, larger array must be created, the contents of the old, smaller array must be copied into the new array, and the old array must be deallocated. To avoid this expense, the number of array reallocations should be kept to a minimum. One popular approach is to double the size of the array whenever it needs to be made larger. For examples, suppose a list begins with a capacity of 10. As it expands, its capacity is changed to 20, then 40, then 80, and so forth. The array never becomes smaller.

Iterators for lists implemented with an array are also very easy to code. The iterator object need merely keep a reference to the list and the current index during iteration, which acts as a cursor marking the current element during iteration.

> **Cursor:** A variable marking a location in a data structure.

Accessing the element at index *i* of an array is almost instantaneous, so the `[]` and `[]=` list operations are very fast using a contiguous implementation. But adding and removing elements requires moving slices of the list up or down in the array, which can be very slow. Hence for applications where list elements are often accessed but not too often added or removed, the contiguous implementation will be very efficient; applications that have the opposite behavior will be much less efficient, especially if the lists are long.

10.6 Linked Implementation of the List ADT

A linked implementation of the list ADT uses a linked data structure to represent values of the ADT carrier set. A singly or multiply linked list may be used, depending on the needs of clients. We will consider using singly or doubly linked lists to illustrate implementation alteratives.

Suppose a singly-linked list is used to hold list elements. It consists of a variable, traditionally called `head`, holding nil when the list is empty and a reference to the first node in the list otherwise. The length of list is also typically recorded.

Most list operations take an index *i* as an argument, so most algorithms to implement these operations will have to begin by walking down the list to locate the node at position *i* or position *i*-1. (Why *i*-1? Because for addition and removal, the link field in the node preceding node *i* will have to be changed.) In any case, if lists are long and there are many changes towards the end of the list, much processing will be done simply finding the right spot in the list to do things.

This difficulty can be alleviated by keeping a special cursor consisting of an index number and a reference into the list. The cursor is used to find the node that some operation needs to do its job. The next time an operation is called, it may be able to use the existing value of the cursor, or use it with slight changes, thus saving time. For example, suppose that a value is added at the end of the list. The cursor is used to walk down to the end of the list and make the addition; when this task is done, the cursor marks the node at, let us say, location *size*-2 in the list. If another addition is made, the cursor only needs to be moved forward one node to the end of the list to mark the node whose link field must be changed—the walk down the list from its beginning has been avoided.

It may be useful to maintain a reference to the end of the list. Then operations at the end of the list can be done quickly in exchange for the slight extra effort of maintaining the extra reference. If a client does many operations at the end of a list, the extra work will be justified.

Another way to make list operations faster is to store elements in a doubly linked list in which each node (except those at the ends) has a link to both its successor and its predecessor nodes. Figure 2 below illustrates this setup. Using a cursor with a doubly linked list can speed things up considerably because the links make it possible to move both backwards and forwards in the list. If an operation needs to get to node *i* and the cursor marks node *j*, which is closer to node *i* than node *i* is to the head of the list, following links from the cursor can get to node *i* more quickly than following links from the head of the list. Keeping a reference to the end of the list makes things faster still: it is possible to start walking toward a node from three points: the front of the list, the back of the list, or the cursor, which is often somewhere in the middle of the list.

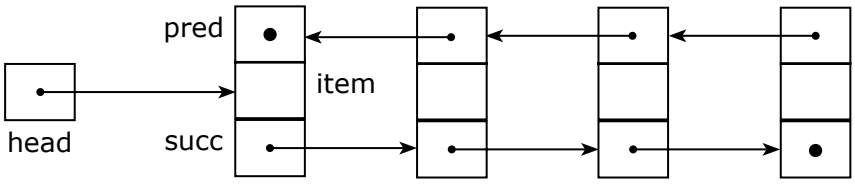

Figure 2: A Doubly Linked List

Another trick is to make the list **circular**: have the `pred` link of the first node point to the last node rather than containing nil, and have the `succ` link of the last node point to the fist node rather than containing nil. Then there is no need for a separate reference to the end of the list: the `head` reference can be used to get to both the front and the rear of the list.

Iterators for the linked implementation of lists must obtain a reference to the head of the list, which can be passed to the instance constructor. Then it is merely a question of maintaining a cursor and walking down the list as the Iterator.next() operation is called.

Modifying linked implementations of lists is very fast once the nodes to operate on have been found, and using doubly linked lists with cursors can make node finding fairly fast. A linked implementation is a good choice when a variety of list operations are needed.

10.7 Implementing Lists in Ruby

The Ruby array is already very similar to a contiguously implemented list; it lacks only `contains?()` and `iterator()` operations. Hence the simplest way to implement an `ArrayList` in Ruby is to make the `ArrayList` a sub-class of the built-in Ruby `Array` class and add the two missing operations. Of course, the `iterator()` factory method must return an instance of an `Iterator` that traverses the `ArrayList`, so an `ArrayListIterator` class must also be written.

A `LinkedList` can be implemented in Ruby by making linked lists using a `Node` class containing data and link fields and writing operations to perform list operations as discussed above. In this case `LinkedList` is a sub-class of `List`, an interface that is a sub-class of `Collection`, which in turn is a sub-class of `Container`.

10.8 Summary and Conclusion

The contiguous implementation of lists is very easy to program and quite efficient for element access, but quite slow for element addition and removal. The linked implementation is considerably more complex to program, and can be slow if operations must always locate nodes by walking down the list from its head, but if double links and a cursor are used, list operations can be quite fast across the board. The contiguous implementation is preferable for lists that don't change often but must be accessed quickly, while the linked implementation is better when these characteristics don't apply.

10.9 Review Questions

1. Does it matter where list element numbering begins?
2. What does the list ADT *index*(*t*,*e*) operation return when *e* is not in list *t*?
3. What is a cursor?
4. Under what conditions does the contiguous implementation of the list ADT not perform well?
5. What advantage does a doubly linked list provide over a singly linked list?
6. What advantage does a circular doubly-linked list provide over a non-circular list?
7. What advantage does a cursor provide in the linked implementation of lists?
8. In an application where long lists are changed infrequently but access to the middle of the lists are common, would a contiguous or linked implementation be better?

10.10 Exercises

1. Do we really need iterators for lists? Explain why or why not.
2. Would it be worthwhile to maintain a cursor for a contiguously implemented list? Explain why or why not.
3. What should happen if a precondition of a List operation is violated?
4. In the List interface, why do the delete_at() and []() operations not have preconditions while the insert(), []=(), and slice() operations do have preconditions?
5. An Iterator must have a reference to the Collection with which it is associated. How is such a connection made if a concrete iterator class is separate from its associated collection class? For example, suppose the ArrayListIterator is a class entirely separate from the ArrayList class. How can an ArrayListIterator instance obtain a connection to the particular ArrayList instance it is supposed to traverse?
6. A ListIterator is a kind of Iterator that (a) allows a client to start iteration at the end of list and go through it backwards, (b) change the direction of iteration during traversal, and (c) obtain the index as well as the value of the current element. Write an interface for ListIterator that is a sub-interface of Iterator.
7. Write ArrayList and ArrayListIterator implementations in Ruby. Note that the List interface is almost entirely implemented by the Ruby `Array` class: all that is missing is the iterator() and contains?() operations.

8. Write an ArrayListListIterator to go along with the implementation in the last exercise.
9. Write a LinkedList implementation in Ruby that uses a singly-linked list, no reference to the end of the list, and a cursor. Write a LinkedListIterator to go along with it.
10. Write a LinkedListListIterator to go along with the LinkedList class in the previous exercise.
11. Write a LinkedList implementation in Ruby that uses a doubly-linked circular list and a cursor. Write a LinkedListIterator to along with it.
12. Write a LinkedListListIterator to go along with the LinkedList class in the previous exercise.

10.11 Review Question Answers

1. It does not matter where list element numbering begins: it may begin at any value. However, it is usual in computing to start at zero, and in everyday life to start at one, so one of these values is preferable.
2. The list ADT *index*(*t*,*e*) operation cannot return an index when *e* is not in list *t*. It should return a value indicating that the result is undefined, namely *nil*.
3. A cursor is a variable marking a location in a data structure. In the case of a List, a cursor is a data structure marking a particular element in the List. For an ArrayList, a cursor might be simply an index. For a LinkedList, it is helpful for the cursor to hold both the index and a reference to the node where the item is stored.
4. A contiguous implementation of the list ADT does not perform well when the list is long and it is changed near its beginning often. Every change near the beginning of a long contiguously implemented list requires that many list elements be copied up or down the list, which is expensive.
5. A doubly linked list makes it faster to move around than in a singly linked list, which can speed up the most expensive part of linked list operations: finding the nodes in the list where the operation must do its job.
6. A circular doubly-linked list makes it possible to follow links quickly from the list head to the end of the list. This can only be done in a non-circular list if a reference to the end of the list is maintained.
7. A cursor helps to speed up linked list operations by often making it faster to get to the nodes where the operations must do their work. Even in a circular doubly-linked list, it is expensive to get to the middle of the list. If a cursor is present and it ends up near the middle of a list after some list operation, then it can be used to get to a node in the middle of the list very quickly.
8. In an application where long lists are changed infrequently but are accessed near their middle often, a contiguous implementation will likely be better than a linked implementation because access to the middle of a contiguously implemented list (no matter how long) is instantaneous, while access to the middle of a linked list will almost always be slower, and could be extremely slow (if link following must begin at one end of the list).