Sequence Abstract Data Type

Gunnar Gotshalks

2007 December

Table of Contents

1 Introduction

The sequence data type is one of the fundamental data types in computer science. Many other data types such as text files, stacks and queues are variations on the sequence theme. Even strings can be thought of as sequences of characters, although this does not seem to be as useful or natural as it might at first appear. Problems occur, for example, with operations such as sub−string, index and replace. Thus, in our study of sequences we will not include the notion of strings but will treat strings as an independent fundamental structure.

From a theoretical perspective, once the sequence and set abstract data types are available, it is possible to program without reference to arrays and linked lists – although arrays and linked lists are used to implement sequences and sets.

A sequence consists of a homogeneous ordered collection of objects of any type. Note the similarity with the concept of an array. Where the sequence differs from an array is that, in a sequence, we distinguish, and give special names to the two ends of the sequence. Furthermore, we can only access the objects which are at the ends of the sequence. There are no operations which delve into the middle of the sequence to examine and/or modify sequence items.

Stacks, queues, double ended queues and files are examples of sequences as normal operations work only at the ends.

2 Objects for the sequence data type

A sequence consists of five entities.

2.1 The sequence as an object

The first entity is the sequence as a whole. The entire sequence is passed as a parameter to the operations and the sequence is operated upon as a single entity.

The value of a sequence is denoted as a list of items, separated by commas and enclosed in angular brackets, as in the examples shown below.

Examples

1. ⟨⟩ – is the empty sequence containing no members.
2. ⟨x⟩ – is a sequence containing only the object x.
3. ⟨x,y,z⟩ – consists of the three members x, y and z, in that order.
4. $\langle x_1, \ldots, x_n \rangle$ – consists of n members x_1 through x_n inclusive.

2.2 Sequence components

Figure 1 shows the components of a sequence and their relationship.

The first item in a sequence is called the **head**. The last item in a squence is called the **last**. Complementing the head and last items are the sub−sequence **tail** that consists of all of the sequence except for the head item, and the **front** that consists of all of the sequence except for the last item.

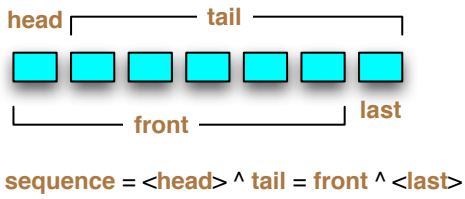

Figure 1: The components of a sequence.

3 Operations on sequences

You should be aware that the names of the various operations are often changed for a specific instantiation of sequences. For example, put_head and take_head could be called push and pop for stacks.

4 Enquiry operations

We only define one operation.

4.0.1 What is the length of a sequence?

```
length( s : SEQUENCE ) : integer
```

require s ≠ void
ensure Result = #s

The length is never a negative integer. The empty sequence, ⟨⟩, always has length 0.

The length of a sequence is defined by the following recursive definition.

```
1  #⟨⟩ = 0
2  #⟨x⟩ = 1
3  #(s^t) = #s + #t
```

Program text is not referenced

4.1 Read operations

There are two read operations.

4.1.1 Read the front object from a sequence

```
read_head(s : SEQUENCE ) : SEQ_TYPE
```

require #s > 0
ensure Result = s(1) = head s

Program text is not referenced

4.1.2 Read the rear object from a sequence

```
read_last( s : SEQUENCE ) : SEQ_TYPE
```

require #s > 0
ensure Result = s(#s) = last s

Program text is not referenced

4.2 Write operations

4.2.1 Create a new sequence

```
create(seq_parameters) : SEQUENCE
```

require The sequence does not exist. seq_parameters contains all the attributes we want the sequence to have including its name and base type.
ensure A empty sequence is created.

Program text is not referenced

4.2.2 Dispose of an existing sequence

```
dispose(s : SEQUENCE )
```

require s ≠ void

ensure The sequence s is removed from the system.

Program text is not referenced

4.2.3 Initialize a sequence to the empty state

```
init(s : SEQUENCE )
```

require s ≠ void
ensure s = ⟨⟩

Program text is not referenced

4.2.4 Take the first item from a sequence

```
take_head(s : SEQUENCE )
```

require #s ≥ 0
ensure s' = *tail* s

Program text is not referenced

4.2.5 Take the last item from a sequence

```
take_rear(s : SEQUENCE )
```

require #s ≥ 0
ensure s' = *front* s

Program text is not referenced

4.2.6 Put a new item at the front of a sequence

```
put_head(s : SEQUENCE ; item : SEQ_TYPE)
```

require s ≠ void
ensure s' = ⟨item⟩ ^ s

Program text is not referenced

4.2.7 Put a new item at the rear of a sequence

```
put_last(s : SEQUENCE ; item : SEQ_TYPE)
```

require s ≠ void
ensure s' = s ^ ⟨item⟩

Program text is not referenced

4.2.8 Concatenate two sequences

```
append( s , t : SEQUENCE )
```

require s ≠ void ∧ t ≠ void
ensure s' = s ^ t

The append operator obeys the following laws, where s, t and u are sequences.

L1 s ^ ⟨⟩ = ⟨⟩ ^ s = s
L2 s ^ (t ^ u) = (s ^ t) ^ u
L3 s ^ t = s ^ u ≡ t = u
L4 t ^ s = u ^ s ≡ t = u
L5 s ^ t = ⟨⟩ ≡ s=⟨⟩ ∧ t=⟨⟩

Program text is not referenced

4.2.9 Exercises

What sequence operations would correspond to the following data structures operations?

1. Stack operations push and pop.
2. Qeueue operations enqueue and dequeue.
3. File operations read and write.

5 Dictionary

All examples assume that s = ⟨x,y,z⟩.

- **head** is the first item in a sequence – *front* s = x.
- **tail** is the sequence without the first item – *tail* s = ⟨y, z⟩.
- **last** is the last item in a sequence. – *last* s = z.
- **front** is the sequence without the last item – *front* s = ⟨x, y⟩.
- #s is length of the sequence s – #s = 3.
  See the specification of the operation length for a formal definition of the length of a sequence.
- ⟨⟩ is the empty sequence. It has length zero.
- *rev* s is reverse of the sequence s – *rev* s = ⟨ z, y, x⟩.
- ^ is the concat operator; it means append – s ^ ⟨a, b, c⟩ = ⟨x, y, z, a, b, c⟩.

6 Operation interactions

The following is a sequence of axioms that show the the results of using multiple sequence operators.

1. read_head(put_head(s,x)) = x

   Putting item x to the head of a sequence and then reading the head of the sequence we have the item x.
   Note that in an axiomatic representation all procedures are assumed to be functions that return the modified sequence.
2. read_last(put_last(s,x)) = x

   Putting item x to the last of a sequence, then reading the last of the sequence gives the item x.
3. put_head(take_head(s), read_head(s)) = s

   Putting the head of a sequence s to tail of the sequence s gives the sequence s. (See Figure 1).
4. put_last(take_last(s), read_last(s)) = s
5. create = ⟨⟩ = init
6. read_head(⟨⟩) = error
7. read_read(⟨⟩) = error
8. take_head(⟨⟩) = error
9. take_last(⟨⟩) = error

7 Physical representations for sequences

Sequences in memory are represented using arrays and linked lists.

7.1 Array implementation

In mathematics we think of a sequence as being a vector or one−dimensional matrix as shown in Figure 2.

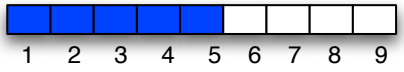

Figure 2: Sequence reqpresented as a vector.

When taking from the head of the sequence, the array elements are shifted as shown in Figure 3. The operation is O(n) in time.

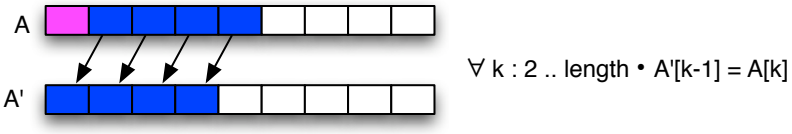

Figure 3: Array sequence take_head operation.

When putting to the head of the sequence, the array elements are shifted as shown in Figure 4. The operations is O(n) in time.

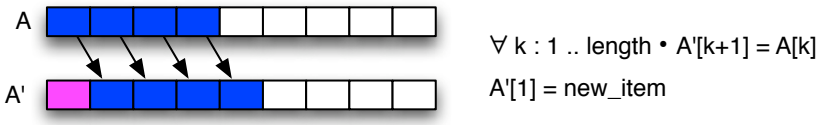

Figure 4: Array sequence put_head operation.

7.2 Circular array implementation

Since arrays are fixed in size, we can logically think of wrapping the array in a circle with last element following preceeding the first element as shown in Figure 5. The array bounds are set to 0 .. n−1 to simplify the arithmetic. The put_head and take_head operations are shown in Figure 5. The operations are O(1) in time.

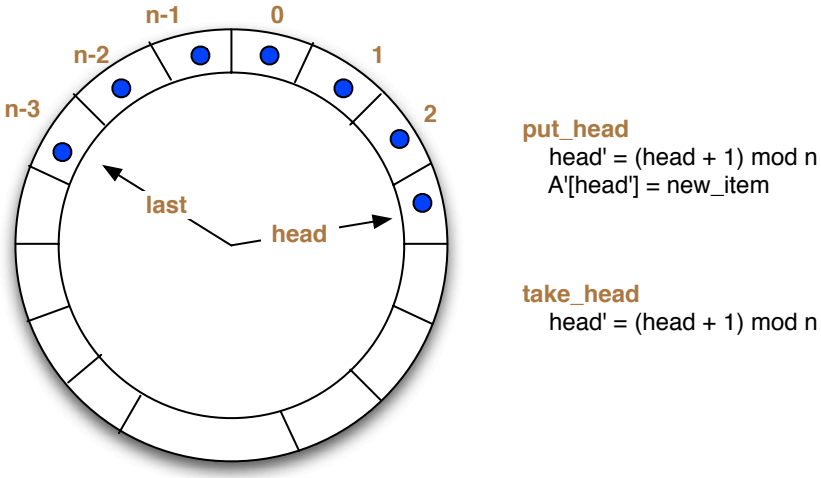

Figure 5: Circular array sequence put and take head operations.

7.2.1 Exercise

Write the put_last and take_last circular array operations.

7.3 Linked list implementation

The elements of the sequence are physically separated but are logically linked with pointers indicating the sequence order, as shown in Figure 6. Both Singly and doubly linked lists require O(n) extra space for the pointers, even though doubly linked lists have twice the extra space overhead of singly linked lists.

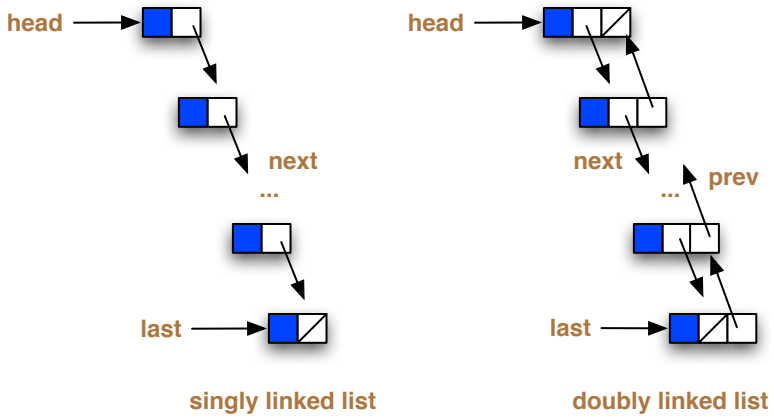

Figure 6: Singly and Doubly linked list implementations of sequences.

7.4 Singly linked list put & take operations

The put_head and take_head operations are shown in Figure 5. The operations are O(1) in time.

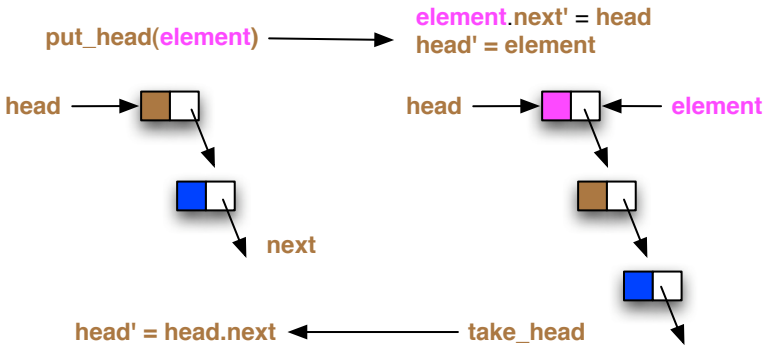

Figure 7: Singly linked list operations at head.

The put_last operation is shown in Figure 8. The operation is O(1) in time.

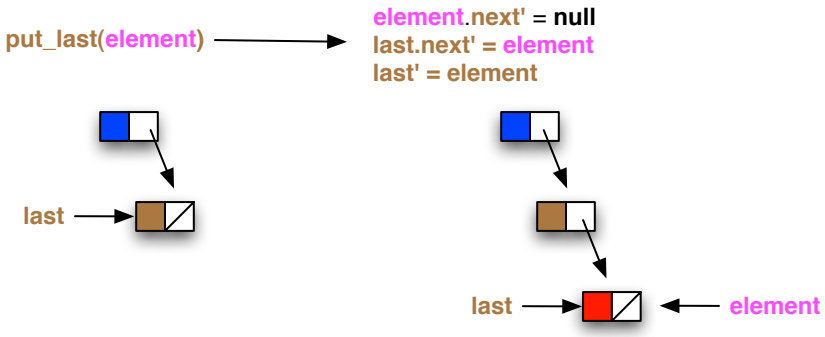

Figure 7: Singly linked list operation put_last.

The take_last operation is shown in Figure 9. In this case an algorithm is given, as only an algorithm can show the sequence of state changes needed to achieve the post_condition. The operation is O(n) in time.

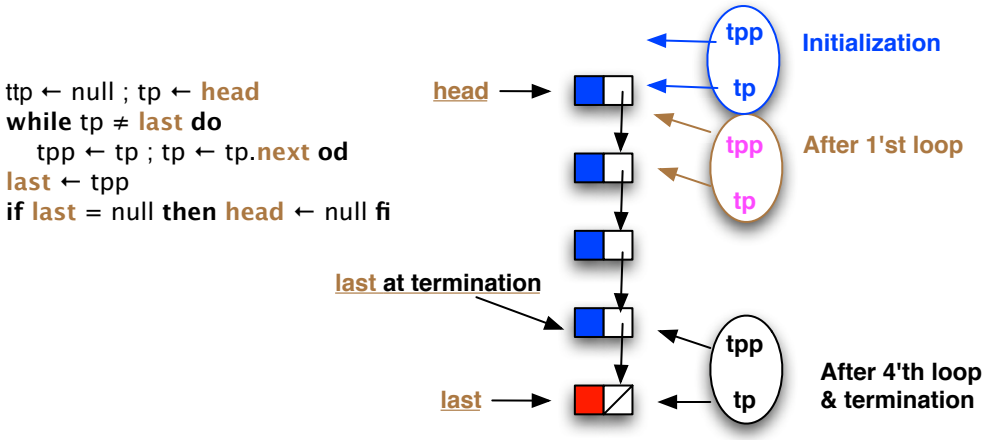

Figure 9: Singly linked list operation take_last.

7.5 Doubly linked list put & take operations

The put_head and take_head operations are shown in Figure 10. The operations are O(1) in time.

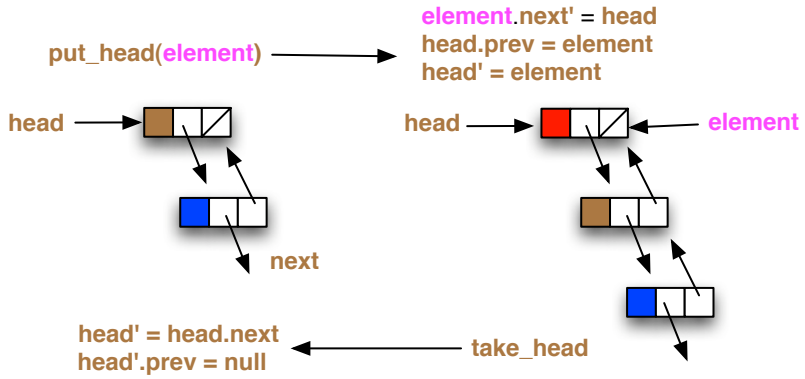

Figure 10: Doubly linked list operations at head.

The put_last and take_last operations are shown in Figure 11. The operations are O(1) in time.

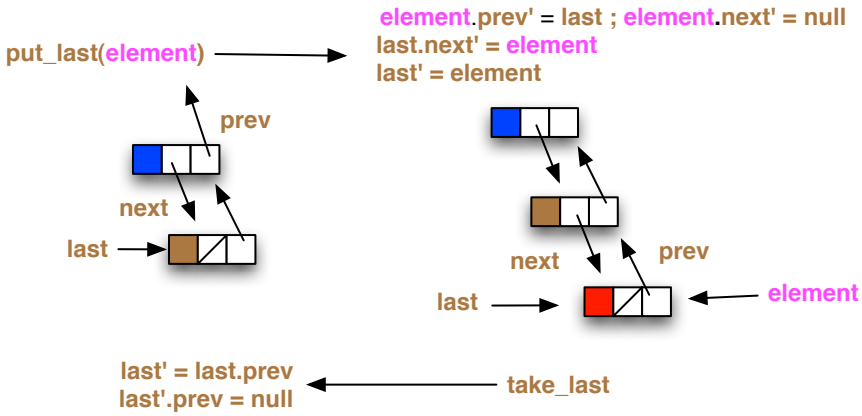

Figure 11: Doubly linked list operations at last.

8 Sequence Interface

Sequences are containers. Figure 12 shows the relationship between sequences and containers in BON (Business Object Notation). The diagrams assume that the operations put_last and take_last are implemented by the add and remove routines in the CONTAINER class.

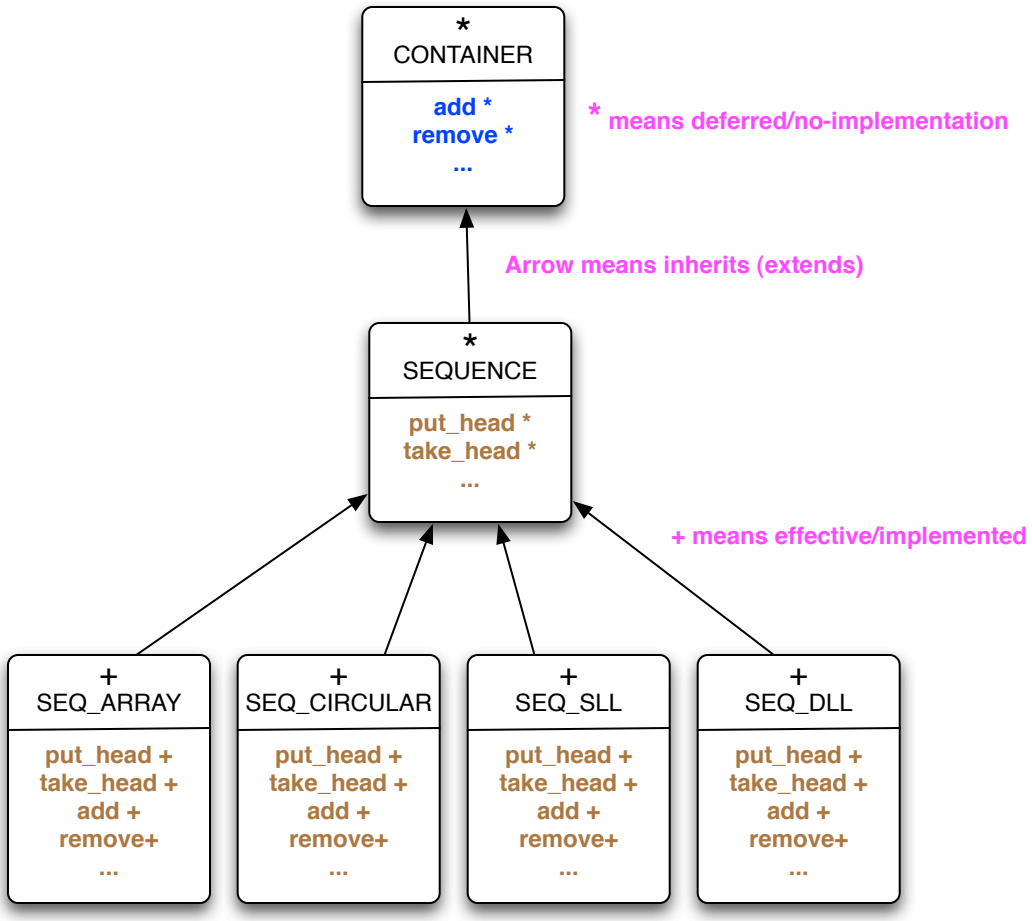

Figure 12: BON structure diagram for sequences.

The following program text shows how the interface could look in Java.

```
public interface SEQUENCE extends CONTAINER {
  void put_head(Object object);
  void take_head();
  void add(Object object);     // Implement as put_last
  void remove();               // Implement as take_last

  // Other operations that could be in CONTAINER and SEQUENCE
}
```

Program text is not referenced