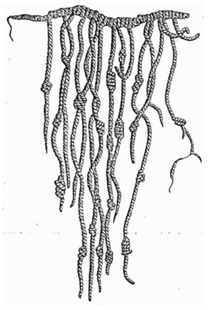
3

Representing Sequences by Arrays and Linked Lists



Perhaps the world's oldest data structures were the tablets in cuneiform script[1] *used more than 5 000 years ago by custodians in Sumerian temples. These custodians kept lists of goods, and their quantities, owners, and buyers. The picture on the left shows an example. This was possibly the first application of written language. The operations performed on such lists have remained the same – adding entries, storing them for later, searching entries and changing them, going through a list to compile summaries, etc. The Peruvian quipu [137] that you see in the picture on the right served a similar purpose in the Inca empire, using knots in colored strings arranged sequentially on a master string. It is probably easier to maintain and use data on tablets than to use knotted string, but one would not want to haul stone tablets over Andean mountain trails. It is apparent that different representations make sense for the same kind of data.*

The abstract notion of a sequence, list, or table is very simple and is independent of its representation in a computer. Mathematically, the only important property is that the elements of a sequence $s = \langle e_0, \ldots, e_{n-1} \rangle$ are arranged in a linear order – in contrast to the trees and graphs discussed in Chaps. 7 and 8, or the unordered hash tables discussed in Chap. 4. There are two basic ways of referring to the elements of a sequence.

One is to specify the index of an element. This is the way we usually think about arrays, where $s[i]$ returns the i-th element of a sequence s. Our pseudocode supports *static* arrays. In a *static* data structure, the size is known in advance, and the data structure is not modifiable by insertions and deletions. In a *bounded* data structure, the maximal size is known in advance. In Sect. 3.2, we introduce *dynamic* or *un-*

[1] The 4 600 year old tablet at the top left is a list of gifts to the high priestess of Adab (see `commons.wikimedia.org/wiki/Image:Sumerian_26th_c_Adab.jpg`).

bounded arrays, which can grow and shrink as elements are inserted and removed. The analysis of unbounded arrays introduces the concept of *amortized analysis*.

The second way of referring to the elements of a sequence is relative to other elements. For example, one could ask for the successor of an element e, the predecessor of an element e', or for the subsequence $\langle e, \ldots, e' \rangle$ of elements between e and e'. Although relative access can be simulated using array indexing, we shall see in Sect. 3.1 that a list-based representation of sequences is more flexible. In particular, it becomes easier to insert or remove arbitrary pieces of a sequence.

Many algorithms use sequences in a quite limited way. Only the front and/or the rear of the sequence are read and modified. Sequences that are used in this restricted way are called *stacks*, *queues*, and *deques*. We discuss them in Sect. 3.4. In Sect. 3.5, we summarize the findings of the chapter.

3.1 Linked Lists

In this section, we study the representation of sequences by linked lists. In a doubly linked list, each item points to its successor and to its predecessor. In a singly linked list, each item points to its successor. We shall see that linked lists are easily modified in many ways: we may insert or delete items or sublists, and we may concatenate lists. The drawback is that random access (the operator $[\cdot]$) is not supported. We study doubly linked lists in Sect. 3.1.1, and singly linked lists in Sect. 3.1.2. Singly linked lists are more space-efficient, and somewhat faster, and should therefore be preferred whenever their functionality suffices. A good way to think of a linked list is to imagine a chain, where one element is written on each link. Once we get hold of one link of the chain, we can retrieve all elements.

3.1.1 Doubly Linked Lists

Figure 3.1 shows the basic building blocks of a linked list. A list *item* stores an element, and pointers to its successor and predecessor. We call a pointer to a list item a *handle*. This sounds simple enough, but pointers are so powerful that we can make a big mess if we are not careful. What makes a consistent list data structure? We

```
Class Handle = Pointer to Item

Class Item of Element                         // one link in a doubly linked list
    e : Element
    next : Handle                              //
    prev : Handle
    invariant next→prev = prev→next = this
```

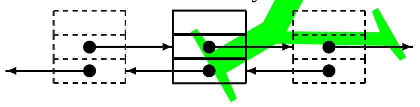


Fig. 3.1. The items of a doubly linked list.

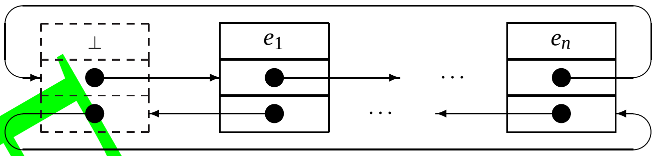


Fig. 3.2. The representation of a sequence $\langle e_1,\ldots,e_n\rangle$ by a doubly linked list. There are $n+1$ items arranged in a ring, a special dummy item h containing no element, and one item for each element of the sequence. The item containing e_i is the successor of the item containing e_{i-1} and the predecessor of the item containing e_{i+1}. The dummy item is between the item containing e_n and the item containing e_1

require that for each item *it*, the successor of its predecessor is equal to *it* and the predecessor of its successor is also equal to *it*.

A sequence of n elements is represented by a ring of $n+1$ items. There is a special dummy item h, which stores no element. The successor h_1 of h stores the first element of the sequence, the successor of h_1 stores the second element of the sequence, and so on. The predecessor of h stores the last element of the sequence; see Fig. 3.2. The empty sequence is represented by a ring consisting only of h. Since there are no elements in that sequence, h is its own successor and predecessor. Figure 3.4 defines a representation of sequences by lists. An object of class *List* contains a single list item h. The constructor of the class initializes the header h to an item containing $\perp$ and having itself as successor and predecessor. In this way, the list is initialized to the empty sequence.

We implement all basic list operations in terms of the single operation *splice* shown in Fig. 3.3. *splice* cuts out a sublist from one list and inserts it after some target item. The sublist is specified by handles a and b to its first and its last element, respectively. In other words, b must be reachable from a by following zero or more next pointers but without going through the dummy item. The target item t can be either in the same list or in a different list; in the former case, it must not be inside the sublist starting at a and ending at b.

splice does not change the number of items in the system. We assume that there is one special list, *freeList*, that keeps a supply of unused elements. When inserting new elements into a list, we take the necessary items from *freeList*, and when removing elements, we return the corresponding items to *freeList*. The function *checkFreeList* allocates memory for new items when necessary. We defer its implementation to Exercise 3.3 and a short discussion in Sect. 3.6.

With these conventions in place, a large number of useful operations can be implemented as one-line functions that all run in constant-time. Thanks to the power of *splice*, we can even manipulate arbitrarily long sublists in constant-time. Figures 3.4 and 3.5 show many examples. In order to test whether a list is empty, we simply check whether h is its own successor. If a sequence is nonempty, its first and its last element are the successor and predecessor, respectively, of h. In order to move an item b to the positions after an item a', we simply cut out the sublist starting and ending at b and insert it after a'. This is exactly what $splice(b,b,a')$ does. We move

```
// Remove $\langle a,\dots,b\rangle$ from its current list and insert it after $t$
// $\dots,a',a,\dots,b,b',\dots + \dots,t,t',\dots \mapsto \dots,a',b',\dots + \dots,t,a,\dots,b,t',\dots$
**Procedure** *splice*($a$,$b$,$t$ : *Handle*)
    **assert** *a and b belong to the same list, b is not before a, and* $t \notin \langle a,\dots,b\rangle$

    // cut out $\langle a,\dots,b\rangle$
    $a' := a \to prev$
    $b' := b \to next$
    $a' \to next := b'$                                  //
    $b' \to prev := a'$                                  //

    // insert $\langle a,\dots,b\rangle$ after $t$
    $t' := t \to next$                                   //

    $b \to next := t'$                                   //
    $a \to prev := t$                                    //

    $t \to next := a$                                    //
    $t' \to prev := b$                                   //
```

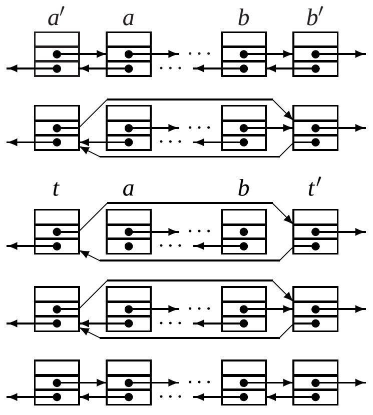


Fig. 3.3. Splicing lists.

```
**Class** *List* **of** *Element*
    // Item $h$ is the predecessor of the first element and the successor of the last element.
    $h = \begin{pmatrix} \perp \\ \textbf{this} \\ \textbf{this} \end{pmatrix}$ : *Item*                // init to empty sequence

    // Simple access functions
    **Function** *head*() : *Handle*; **return address of** $h$        // Pos. before any proper element

    **Function** *isEmpty* : $\{0,1\}$; **return** $h.next$ = **this**                    // $\langle\rangle$?
    **Function** *first* : *Handle*; **assert** $\neg$*isEmpty*; **return** $h.next$
    **Function** *last* : *Handle*; **assert** $\neg$*isEmpty*; **return** $h.prev$

    // Moving elements around within a sequence.
    // $(\langle\dots,a,b,c\dots,a',c',\dots\rangle) \mapsto (\langle\dots,a,c\dots,a',b,c',\dots\rangle)$
    **Procedure** *moveAfter*($b$, $a'$ : *Handle*) *splice*($b,b,a'$)
    **Procedure** *moveToFront*($b$ : *Handle*) *moveAfter*($b$, *head*)
    **Procedure** *moveToBack*($b$ : *Handle*) *moveAfter*($b$, *last*)
```

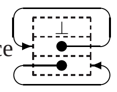


Fig. 3.4. Some constant-time operations on doubly linked lists.

an element to the first or last position of a sequence by moving it after the head or after the last element, respectively. In order to delete an element b, we move it to *freeList*. To insert a new element e, we take the first item of *freeList*, store the element in it, and move it to the place of insertion.

```
// Deleting and inserting elements.
// $\langle \ldots, a, b, c, \ldots \rangle \mapsto \langle \ldots, a, c, \ldots \rangle$
**Procedure** *remove*($b$ : *Handle*) *moveAfter*($b$, *freeList.head*)
**Procedure** *popFront* *remove*(*first*)
**Procedure** *popBack* *remove*(*last*)

// $\langle \ldots, a, b, \ldots \rangle \mapsto \langle \ldots, a, e, b, \ldots \rangle$
**Function** *insertAfter*($x$ : *Element*; $a$ : *Handle*) : *Handle*
    checkFreeList                     // make sure freeList is nonempty. See also Exercise 3.3
    $a' := freeList.first$            // Obtain an item $a'$ to hold $x$,
    moveAfter($a'$, $a$)              // put it at the right place.
    $a' \rightarrow e := x$           // and fill it with the right content.
    **return** $a'$

**Function** *insertBefore*($x$ : *Element*; $b$ : *Handle*) : *Handle* **return** *insertAfter(e, pred(b))*
**Procedure** *pushFront*($x$ : *Element*) *insertAfter(x, head)*
**Procedure** *pushBack*($x$ : *Element*) *insertAfter(x, last)*

// Manipulations of entire lists
// $(\langle a, \ldots, b \rangle, \langle c, \ldots, d \rangle) \mapsto (\langle a, \ldots, b, c, \ldots, d \rangle, \langle \rangle)$
**Procedure** *concat*($L'$ : *List*)
    splice($L'$.first, $L'$.last, last)

// $\langle a, \ldots, b \rangle \mapsto \langle \rangle$
**Procedure** *makeEmpty*
    freeList.concat(**this** )                //
```

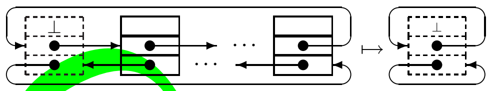
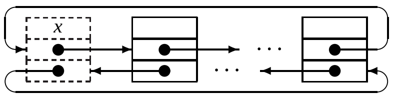


Fig. 3.5. More constant-time operations on doubly linked lists.

Exercise 3.1 (alternative list implementation). Discuss an alternative implementation of *List* that does not need the dummy item h. Instead, this representation stores a pointer to the first list item in the list object. The position before the first list element is encoded as a null pointer. The interface and the asymptotic execution times of all operations should remain the same. Give at least one advantage and one disadvantage of this implementation compared with the one given in the text.

The dummy item is also useful for other operations. For example, consider the problem of finding the next occurrence of an element x starting at an item *from*. If x is not present, *head* should be returned. We use the dummy element as a *sentinel*. A sentinel is an element in a data structure that makes sure that some loop will terminate. In the case of a list, we store the key we are looking for in the dummy element. This ensures that x is present in the list structure and hence a search for it will always terminate. The search will terminate in a proper list item or the dummy item, depending on whether x was present in the list originally. It is no longer necessary, to test whether the end of the list has been reached. In this way, the trick of using the dummy item h as a sentinel saves one test in each iteration and significantly improves the efficiency of the search:

```
**Function** findNext(x : Element; from : Handle) : Handle
    h.e = x                          // Sentinel
    while from → e ≠ x do
        from := from → next
    return from
```



Exercise 3.2. Implement a procedure *swap* that swaps two sublists in constant time, i.e., sequences $(\langle \dots, a', a, \dots, b, b', \dots \rangle, \langle \dots, c', c, \dots, d, d', \dots \rangle)$ are transformed into $(\langle \dots, a', c, \dots, d, b', \dots \rangle, \langle \dots, c', a, \dots, b, d', \dots \rangle)$. Is *splice* a special case of *swap*?

Exercise 3.3 (memory management). Implement the function *checkFreelist* called by *insertAfter* in Fig. 3.5. Since an individual call of the programming-language primitive **allocate** for every single item might be too slow, your function should allocate space for items in large batches. The worst-case execution time of *checkFreeList* should be independent of the batch size. Hint: in addition to *freeList*, use a small array of free items.

Exercise 3.4. Give a constant-time implementation of an algorithm for rotating a list to the right: $\langle a, \dots, b, c \rangle \mapsto \langle c, a, \dots, b \rangle$. Generalize your algorithm to rotate $\langle a, \dots, b, c, \dots, d \rangle$ to $\langle c, \dots, d, a, \dots, b \rangle$ in constant time.

Exercise 3.5. *findNext* using sentinels is faster than an implementation that checks for the end of the list in each iteration. But how much faster? What speed difference do you predict for many searches in a short list with 100 elements, and in a long list with 10 000 000 elements, respectively? Why is the relative speed difference dependent on the size of the list?

Maintaining the Size of a List

In our simple list data type, it is not possible to determine the length of a list in constant time. This can be fixed by introducing a member variable *size* that is updated whenever the number of elements changes. Operations that affect several lists now need to know about the lists involved, even if low-level functions such as *splice* only need handles to the items involved. For example, consider the following code for moving an element a from a list L to the position after a' in a list L':

```
**Procedure** moveAfter(a, a' : Handle; L, L' : List)
    splice(a, a, a');    L.size--;    L'.size++
```

Maintaining the size of lists interferes with other list operations. When we move elements as above, we need to know the sequences containing them and, more seriously, operations that move sublists between lists cannot be implemented in constant time anymore. The next exercise offers a compromise.

Exercise 3.6. Design a list data type that allows sublists to be moved between lists in constant time and allows constant-time access to *size* whenever sublist operations have not been used since the last access to the list size. When sublist operations have been used, *size* is recomputed only when needed.

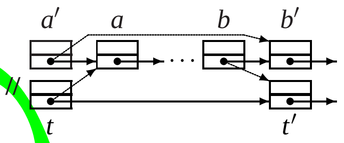
Exercise 3.7. Explain how the operations *remove*, *insertAfter*, and *concat* have to be modified to keep track of the length of a *List*.

3.1.2 Singly Linked Lists

The two pointers per item of a doubly linked list make programming quite easy. Singly linked lists are the lean sisters of doubly linked lists. We use *SItem* to refer to an item in a singly linked list. *SItem*s scrap the predecessor pointer and store only a pointer to the successor. This makes singly linked lists more space-efficient and often faster than their doubly linked brothers. The downside is that some operations can no longer be performed in constant time or can no longer be supported in full generality. For example, we can remove an *SItem* only if we know its predecessor.

We adopt the implementation approach used with doubly linked lists. *SItem*s form collections of cycles, and an *SList* has a dummy *SItem* *h* that precedes the first proper element and is the successor of the last proper element. Many operations on *List*s can still be performed if we change the interface slightly. For example, the following implementation of *splice* needs the *predecessor* of the first element of the sublist to be moved:

```
// $(\langle \ldots, a', a, \ldots, b, b' \ldots \rangle, \langle \ldots, t, t', \ldots \rangle) \mapsto (\langle \ldots, a', b' \ldots \rangle, \langle \ldots, t, a, \ldots, b, t', \ldots \rangle)$
Procedure $splice(a', b, t : SHandle)$
    $\begin{pmatrix} a' \to next \\ t \to next \\ b \to next \end{pmatrix} := \begin{pmatrix} b \to next \\ a' \to next \\ t \to next \end{pmatrix}$
```



Similarly, *findNext* should not return the handle of the *SItem* with the next hit but its *predecessor*, so that it remains possible to remove the element found. Consequently, *findNext* can only start searching at the item *after* the item given to it. A useful addition to *SList* is a pointer to the last element because it allows us to support *pushBack* in constant time.

Exercise 3.8. Implement classes *SHandle*, *SItem*, and *SList* for singly linked lists in analogy to *Handle*, *Item*, and *List*. Show that the following functions can be implemented to run in constant time. The operations *head*, *first*, *last*, *isEmpty*, *popFront*, *pushFront*, *pushBack*, *insertAfter*, *concat*, and *makeEmpty* should have the same interface as before. The operations *moveAfter*, *moveToFront*, *moveToBack*, *remove*, *popFront*, and *findNext* need different interfaces.

We shall see several applications of singly linked lists in later chapters, for example in hash tables in Sect. 4.1 and in mergesort in Sect. 5.2. We may also use singly linked lists to implement free lists of memory managers – even for items in doubly linked lists.

3.2 Unbounded Arrays

Consider an array data structure that, besides the indexing operation $[\cdot]$, supports the following operations *pushBack*, *popBack*, and *size*:

$$\begin{aligned}
\langle e_0,\ldots,e_n\rangle.\mathit{pushBack}(e) &= \langle e_0,\ldots,e_n,e\rangle\ ,\\
\langle e_0,\ldots,e_n\rangle.\mathit{popBack} &= \langle e_0,\ldots,e_{n-1}\rangle\ ,\\
\mathit{size}(\langle e_0,\ldots,e_{n-1}\rangle) &= n\ .
\end{aligned}$$

Why are unbounded arrays important? Because in many situations we do not know in advance how large an array should be. Here is a typical example: suppose you want to implement the Unix command `sort` for sorting the lines of a file. You decide to read the file into an array of lines, sort the array internally, and finally output the sorted array. With unbounded arrays, this is easy. With bounded arrays, you would have to read the file twice: once to find the number of lines it contains, and once again to actually load it into the array.

We come now to the implementation of unbounded arrays. We emulate an unbounded array u with n elements by use of a dynamically allocated bounded array b with w entries, where $w \geq n$. The first n entries of b are used to store the elements of u. The last $w-n$ entries of b are unused. As long as $w > n$, *pushBack* simply increments n and uses the first unused entry of b for the new element. When $w = n$, the next *pushBack* allocates a new bounded array b' that is larger by a constant factor (say a factor of two). To reestablish the invariant that u is stored in b, the contents of b are copied to the new array so that the old b can be deallocated. Finally, the pointer defining b is redirected to the new array. Deleting the last element with *popBack* is even easier, since there is no danger that b may become too small. However, we might waste a lot of space if we allow b to be much larger than needed. The wasted space can be kept small by shrinking b when n becomes too small. Figure 3.6 gives the complete pseudocode for an unbounded-array class. Growing and shrinking are performed using the same utility procedure *reallocate*. Our implementation uses constants α and β, with $\beta = 2$ and $\alpha = 4$. Whenever the current bounded array becomes too small, we replace it by an array of β times the old size. Whenever the size of the current array becomes α times as large as its used part, we replace it by an array of size βn. The reasons for the choice of α and β shall become clear later.

3.2.1 Amortized Analysis of Unbounded Arrays: The Global Argument

Our implementation of unbounded arrays follows the algorithm design principle "make the common case fast". Array access with $[\cdot]$ is as fast as for bounded arrays. Intuitively, *pushBack* and *popBack* should "usually" be fast – we just have to update n. However, some insertions and deletions incur a cost of $\Theta(n)$. We shall show that such expensive operations are rare and that any sequence of m operations starting with an empty array can be executed in time $\mathrm{O}(m)$.

Lemma 3.1. *Consider an unbounded array u that is initially empty. Any sequence* $\sigma = \langle\sigma_1,\ldots,\sigma_m\rangle$ *of pushBack or popBack operations on u is executed in time* $\mathrm{O}(m)$.

```
Class $UArray$ of $Element$
    Constant $\beta = 2 : \mathbb{R}_+$                          // growth factor
    Constant $\alpha = 4 : \mathbb{R}_+$                    // worst case memory blowup
    $w = 1 : \mathbb{N}$                                        // allocated size
    $n = 0 : \mathbb{N}$                                          // current size.
    invariant $n \le w < \alpha n$ or $n = 0$ and $w \le \beta$                 $n$          $w$
    $b : Array\ [0..w-1]$ of $Element$          // $b \rightarrow \boxed{e_0}\boxed{\cdots}\boxed{e_{n-1}}\boxed{\ \ }\boxed{\cdots}\boxed{\ \ }$

    Operator $[i : \mathbb{N}] : Element$
        assert $0 \le i < n$
        return $b[i]$

    Function $size : \mathbb{N}$     return $n$

    Procedure $pushBack(e : Element)$                   // Example for $n = w = 4$:
        if $n = w$ then                                       // $b \rightarrow \boxed{0}\boxed{1}\boxed{2}\boxed{3}$
            $reallocate(\beta n)$                   // $b \rightarrow \boxed{0}\boxed{1}\boxed{2}\boxed{3}\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }$
        $b[n] := e$                                  // $b \rightarrow \boxed{0}\boxed{1}\boxed{2}\boxed{3}\boxed{e}\boxed{\ }\boxed{\ }\boxed{\ }$
        $n{+}{+}$                                      // $b \rightarrow \boxed{0}\boxed{1}\boxed{2}\boxed{3}\boxed{e}\boxed{\ }\boxed{\ }\boxed{\ }$

    Procedure $popBack$                           // Example for $n = 5$, $w = 16$:
        assert $n > 0$            // $b \rightarrow \boxed{0}\boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }$
        $n{-}{-}$                 // $b \rightarrow \boxed{0}\boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }$
        if $\alpha n \le w \wedge n > 0$ then                    // reduce waste of space
            $reallocate(\beta n)$                   // $b \rightarrow \boxed{0}\boxed{1}\boxed{2}\boxed{3}\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }$

    Procedure $reallocate(w' : \mathbb{N})$                  // Example for $w = 4$, $w' = 8$:
        $w := w'$                                             // $b \rightarrow \boxed{0}\boxed{1}\boxed{2}\boxed{3}$
        $b' :=$ allocate $Array\ [0..w'-1]$ of $Element$     // $b' \rightarrow \boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }$
        $(b'[0], \ldots, b'[n-1]) := (b[0], \ldots, b[n-1])$   // $b' \rightarrow \boxed{0}\boxed{1}\boxed{2}\boxed{3}\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }$
        dispose $b$                                           // $b \rightarrow \cancel{\boxed{0}\boxed{1}\boxed{2}\boxed{3}}$
        $b := b'$                    // pointer assignment $b \rightarrow \boxed{0}\boxed{1}\boxed{2}\boxed{3}\boxed{\ }\boxed{\ }\boxed{\ }\boxed{\ }$
```

Fig. 3.6. Pseudocode for unbounded arrays

Lemma 3.1 is a nontrivial statement. A small and innocent-looking change to the program invalidates it.

Exercise 3.9. Your manager asks you to change the initialization of α to $\alpha = 2$. He argues that it is wasteful to shrink an array only when three-fourths of it are unused. He proposes to shrink it when $n \le w/2$. Convince him that this is a bad idea by giving a sequence of m *pushBack* and *popBack* operations that would need time $\Theta(m^2)$ if his proposal was implemented.

Lemma 3.1 makes a statement about the amortized cost of *pushBack* and *popBack* operations. Although single operations may be costly, the cost of a sequence of m operations is $\mathrm{O}(m)$. If we divide the total cost of the operations in σ by the number of operations, we get a constant. We say that the *amortized cost* of each operation is constant. Our usage of the term "amortized" is similar to its usage in everyday language, but it avoids a common pitfall. "I am going to cycle to work every day from now on, and hence it is justified to buy a luxury bike. The cost per ride will be very small – the investment will be amortized." Does this kind of reasoning sound familiar to you? The bike is bought, it rains, and all good intentions are gone. The bike has not been amortized. We shall instead insist that a large expenditure is justified by savings in the past and not by expected savings in the future. Suppose your ultimate goal is to go to work in a luxury car. However, you are not going to buy it on your first day of work. Instead, you walk and put a certain amount of money per day into a savings account. At some point, you will be able to buy a bicycle. You continue to put money away. At some point later, you will be able to buy a small car, and even later you can finally buy a luxury car. In this way, every expenditure can be paid for by past savings, and all expenditures are amortized. Using the notion of amortized costs, we can reformulate Lemma 3.1 more elegantly. The increased elegance also allows better comparisons between data structures.

Corollary 3.2. *Unbounded arrays implement the operation* $[\cdot]$ *in worst-case constant time and the operations pushBack and popBack in amortized constant time.*

To prove Lemma 3.1, we use the *bank account* or *potential* method. We associate an *account* or *potential* with our data structure and force every *pushBack* and *popBack* to put a certain amount into this account. Usually, we call our unit of currency a *token*. The idea is that whenever a call of *reallocate* occurs, the balance in the account is sufficiently high to pay for it. The details are as follows. A token can pay for moving one element from b to b'. Note that element copying in the procedure *reallocate* is the only operation that incurs a nonconstant cost in Fig. 3.6. More concretely, *reallocate* is always called with $w' = 2n$ and thus has to copy n elements. Hence, for each call of *reallocate*, we withdraw n tokens from the account. We charge two tokens for each call of *pushBack* and one token for each call of *popBack*. We now show that these charges suffice to cover the withdrawals made by *reallocate*.

The first call of *reallocate* occurs when there is one element already in the array and a new element is to be inserted. The element already in the array has deposited two tokens in the account, and this more than covers the one token withdrawn by *reallocate*. The new element provides its tokens for the next call of *reallocate*.

After a call of *reallocate*, we have an array of w elements: $n = w/2$ slots are occupied and $w/2$ are free. The next call of *reallocate* occurs when either $n = w$ or $4n \le w$. In the first case, at least $w/2$ elements have been added to the array since the last call of *reallocate*, and each one of them has deposited two tokens. So we have at least w tokens available and can cover the withdrawal made by the next call of *reallocate*. In the second case, at least $w/2 - w/4 = w/4$ elements have been removed from the array since the last call of *reallocate*, and each one of them has deposited one token. So we have at least $w/4$ tokens available. The call of *reallocate* needs at most $w/4$ tokens, and hence the cost of the call is covered. This completes the proof of Lemma 3.1. $\square$

Exercise 3.10. Redo the argument above for general values of α and β, and charge $\beta/(\beta-1)$ tokens for each call of *pushBack* and $\beta/(\alpha-\beta)$ tokens for each call of *popBack*. Let n' be such that $w = \beta n'$. Then, after a *reallocate*, n' elements are occupied and $(\beta-1)n' = ((\beta-1)/\beta)w$ are free. The next call of *reallocate* occurs when either $n = w$ or $\alpha n \leq w$. Argue that in both cases there are enough tokens.

Amortized analysis is an extremely versatile tool, and so we think that it is worthwhile to know some alternative proof methods. We shall now give two variants of the proof above.

Above, we charged two tokens for each *pushBack* and one token for each *popBack*. Alternatively, we could charge three tokens for each *pushBack* and not charge *popBack* at all. The accounting is simple. The first two tokens pay for the insertion as above, and the third token is used when the element is deleted.

Exercise 3.11 (continuation of Exercise 3.10). Show that a charge of $\beta/(\beta-1) + \beta/(\alpha-\beta)$ tokens for each *pushBack* is enough. Determine values of α such that $\beta/(\alpha-\beta) \leq 1/(\beta-1)$ and $\beta/(\alpha-\beta) \leq \beta/(\beta-1)$, respectively.

3.2.2 Amortized Analysis of Unbounded Arrays: The Local Argument

We now describe our second modification of the proof. In the argument above, we used a global argument in order to show that there are enough tokens in the account before each call of *reallocate*. We now show how to replace the global argument by a local argument. Recall that, immediately after a call of *reallocate*, we have an array of w elements, out of which $w/2$ are filled and $w/2$ are free. We argue that at any time after the first call of *reallocate*, the following token invariant holds:

$$\text{the account contains at least } \max(2(n-w/2), w/2-n) \text{ tokens.}$$

Observe that this number is always nonnegative. We use induction on the number of operations. Immediately after the first *reallocate*, there is one token in the account and the invariant requires none. A *pushBack* increases n by one and adds two tokens. So the invariant is maintained. A *popBack* removes one element and adds one token. So the invariant is again maintained. When a call of *reallocate* occurs, we have either $n = w$ or $4n \leq w$. In the former case, the account contains at least n tokens, and n tokens are required for the reallocation. In the latter case, the account contains at least $w/4$ tokens, and n are required. So, in either case, the number of tokens suffices. Also, after the reallocation, $n = w/2$ and hence no tokens are required.

Exercise 3.12. Charge three tokens for a *pushBack* and no tokens for a *popBack*. Argue that the account contains always at least $n + \max(2(n-w/2), w/2-n) = \max(3n-w, w/2)$ tokens.

Exercise 3.13 (popping many elements). Implement an operation *popBack*(k) that removes the last k elements in amortized constant time independent of k.

Exercise 3.14 (worst-case constant access time). Suppose, for a real-time application, you need an unbounded array data structure with a *worst-case* constant execution time for all operations. Design such a data structure. Hint: store the elements in up to two arrays. Start moving elements to a larger array well before a small array is completely exhausted.

Exercise 3.15 (implicitly growing arrays). Implement unbounded arrays where the operation $[i]$ allows any positive index. When $i \geq n$, the array is implicitly grown to size $n = i+1$. When $n \geq w$, the array is reallocated as for *UArray*. Initialize entries that have never been written with some default value $\perp$.

Exercise 3.16 (sparse arrays). Implement bounded arrays with constant time for allocating arrays and constant time for the operation $[\cdot]$. All array elements should be (implicitly) initialized to $\perp$. You are not allowed to make any assumptions about the contents of a freshly allocated array. Hint: use an extra array of the same size, and store the number t of array elements to which a value has already been assigned. Therefore $t = 0$ initially. An array entry i to which a value has been assigned stores that value and an index j, $1 \leq j \leq t$, of the extra array, and i is stored in that index of the extra array.

3.2.3 Amortized Analysis of Binary Counters

In order to demonstrate that our techniques for amortized analysis are also useful for other applications, we shall now give a second example. We look at the amortized cost of incrementing a binary counter. The value n of the counter is represented by a sequence $\ldots\beta_i\ldots\beta_1\beta_0$ of binary digits, i.e., $\beta_i \in \{0,1\}$ and $n = \sum_{i\geq 0}\beta_i 2^i$. The initial value is zero. Its representation is a string of zeros. We define the cost of incrementing the counter as one plus the number of trailing ones in the binary representation, i.e., the transition

$$\ldots 01^k \to \ldots 10^k$$

has a cost $k+1$. What is the total cost of m increments? We shall show that the cost is $\mathrm{O}(m)$. Again, we give a global argument first and then a local argument.

If the counter is incremented m times, the final value is m. The representation of the number m requires $L = 1 + \lceil \log m \rceil$ bits. Among the numbers 0 to $m-1$, there are at most 2^{L-k-1} numbers whose binary representation ends with a zero followed by k ones. For each one of them, an increment costs $1+k$. Thus the total cost of the m increments is bounded by

$$\sum_{0\leq k<L}(k+1)2^{L-k-1} = 2^L \sum_{1\leq k\leq L} k/2^k \leq 2^L \sum_{k\geq 1} k/2^k = 2\cdot 2^L \leq 4m\,,$$

where the last equality uses (A.14). Hence, the amortized cost of an increment is $\mathrm{O}(1)$.

The argument above is global, in the sense that it requires an estimate of the number of representations ending in a zero followed by k ones. We now give a local argument which does not need such a bound. We associate a bank account with the counter. Its balance is the number of ones in the binary representation of the counter. So the balance is initially zero. Consider an increment of cost $k+1$. Before the increment, the representation ends in a zero followed by k ones, and after the increment, the representation ends in a one followed by $k-1$ zeros. So the number of ones in the representation decreases by $k-1$, i.e., the operation releases $k-1$ tokens from the account. The cost of the increment is $k+1$. We cover a cost of $k-1$ by the tokens released from the account, and charge a cost of two for the operation. Thus the total cost of m operations is at most $2m$.

3.3 *Amortized Analysis

We give here a general definition of amortized time bounds and amortized analysis. We recommend that one should read this section quickly and come back to it when needed. We consider an arbitrary data structure. The values of all program variables comprise the state of the data structure; we use S to denote the set of states. In the first example in the previous section, the state of our data structure is formed by the values of n, w, and b. Let s_0 be the initial state. In our example, we have $n=0$, $w=1$, and b is an array of size one in the initial state. We have operations to transform the data structure. In our example, we had the operations *pushBack*, *popBack*, and *reallocate*. The application of an operation X in a state s transforms the data structure to a new state s' and has a cost $T_X(s)$. In our example, the cost of a *pushBack* or *popBack* is 1, excluding the cost of the possible call to *reallocate*. The cost of a call *reallocate*(βn) is $\Theta(n)$.

Let F be a sequence of operations $Op_1, Op_2, Op_3, \ldots, Op_n$. Starting at the initial state s_0, F takes us through a sequence of states to a final state s_n:

$$s_0 \xrightarrow{Op_1} s_1 \xrightarrow{Op_2} s_2 \xrightarrow{Op_3} \cdots \xrightarrow{Op_n} s_n \ .$$

The cost $T(F)$ of F is given by

$$T(F) = \sum_{1 \le i \le n} T_{Op_i}(s_{i-1}) \ .$$

A family of functions $A_X(s)$, one for each operation X, is called a *family of amortized time bounds* if, for every sequence F of operations,

$$T(F) \le A(F) := c + \sum_{1 \le i \le n} A_{Op_i}(s_{i-1})$$

for some constant c not depending on F, i.e., up to an additive constant, the total actual execution time is bounded by the total amortized execution time.

There is always a trivial way to define a family of amortized time bounds, namely $A_X(s) := T_X(s)$ for all s. The challenge is to find a family of simple functions $A_X(s)$ that form a family of amortized time bounds. In our example, the functions $A_{pushBack}(s) = A_{popBack}(s) = A_{[\cdot]}(s) = \mathrm{O}(1)$ and $A_{reallocate}(s) = 0$ for all s form a family of amortized time bounds.

3.3.1 The Potential or Bank Account Method for Amortized Analysis

We now formalize the technique used in the previous section. We have a function *pot* that associates a nonnegative potential with every state of the data structure, i.e., $pot : S \longrightarrow \mathbb{R}_{\geq 0}$. We call $pot(s)$ the potential of the state s, or the balance of the savings account when the data structure is in the state s. It requires ingenuity to come up with an appropriate function *pot*. For an operation X that transforms a state s into a state s' and has cost $T_X(s)$, we define the amortized cost $A_X(s)$ as the sum of the potential change and the actual cost, i.e., $A_X(s) = pot(s') - pot(s) + T_X(s)$. The functions obtained in this way form a family of amortized time bounds.

Theorem 3.3 (potential method). *Let S be the set of states of a data structure, let s_0 be the initial state, and let $pot : S \longrightarrow \mathbb{R}_{\geq 0}$ be a nonnegative function. For an operation X and a state s with $s \xrightarrow{X} s'$, we define*

$$A_X(s) = pot(s') - pot(s) + T_X(s).$$

The functions $A_X(s)$ are then a family of amortized time bounds.

Proof. A short computation suffices. Consider a sequence $F = \langle Op_1, \ldots, Op_n \rangle$ of operations. We have

$$\begin{aligned}\sum_{1\leq i\leq n} A_{Op_i}(s_{i-1}) &= \sum_{1\leq i\leq n} (pot(s_i) - pot(s_{i-1}) + T_{Op_i}(s_{i-1})) \\ &= pot(s_n) - pot(s_0) + \sum_{1\leq i\leq n} T_{Op_i}(s_{i-1}) \\ &\geq \sum_{1\leq i\leq n} T_{Op_i}(s_{i-1}) - pot(s_0),\end{aligned}$$

since $pot(s_n) \geq 0$. Thus $T(F) \leq A(F) + pot(s_0)$. □

Let us formulate the analysis of unbounded arrays in the language above. The state of an unbounded array is characterized by the values of n and w. Following Exercise 3.12, the potential in state (n, w) is $\max(3n - w, w/2)$. The actual costs T of *pushBack* and *popBack* are 1 and the actual cost of *reallocate*(βn) is n. The potential of the initial state $(n, w) = (0, 1)$ is $1/2$. A *pushBack* increases n by 1 and hence increases the potential by at most 3. Thus its amortized cost is bounded by 4. A *popBack* decreases n by 1 and hence does not increase the potential. Its amortized cost is therefore at most 1. The first *reallocate* occurs when the data structure is in the state $(n, w) = (1, 1)$. The potential of this state is $\max(3 - 1, 1/2) = 2$, and the actual cost of the *reallocate* is 1. After the *reallocate*, the data structure is in the state $(n,w) = (1,2)$ and has a potential $\max(3-2,1) = 1$. Therefore the amortized cost of the first *reallocate* is $1-2+1 = 0$. Consider any other call of *reallocate*. We have either $n = w$ or $4n \leq w$. In the former case, the potential before the *reallocate* is $2n$, the actual cost is n, and the new state is $(n,2n)$ and has a potential n. Thus the amortized cost is $n-2n+n = 0$. In the latter case, the potential before the operation is $w/2$, the actual cost is n, which is at most $w/4$, and the new state is $(n,w/2)$ and has a potential $w/4$. Thus the amortized cost is at most $w/4-w/2+w/4 = 0$. We conclude that the amortized costs of *pushBack* and *popBack* are $\mathrm{O}(1)$ and the amortized cost of *reallocate* is zero or less. Thus a sequence of m operations on an unbounded array has cost $\mathrm{O}(m)$.

Exercise 3.17 (amortized analysis of binary counters). Consider a nonnegative integer c represented by an array of binary digits, and a sequence of m increment and decrement operations. Initially, $c = 0$. This exercise continues the discussion at the end of Sect. 3.2.

(a) What is the worst-case execution time of an increment or a decrement as a function of m? Assume that you can work with only one bit per step.
(b) Prove that the amortized cost of the increments is constant if there are no decrements. Hint: define the potential of c as the number of ones in the binary representation of c.
(c) Give a sequence of m increment and decrement operations with cost $\Theta(m \log m)$.
(d) Give a representation of counters such that you can achieve worst-case constant time for increments and decrements.
(e) Allow each digit d_i to take values from $\{-1,0,1\}$. The value of the counter is $c = \sum_i d_i 2^i$. Show that in this *redundant ternary* number system, increments and decrements have constant amortized cost. Is there an easy way to tell whether the value of the counter is zero?

3.3.2 Universality of Potential Method

We argue here that the potential-function technique is strong enough to obtain any family of amortized time bounds.

Theorem 3.4. *Let $B_X(s)$ be a family of amortized time bounds. There is then a potential function pot such that $A_X(s) \leq B_X(s)$ for all states s and all operations X, where $A_X(s)$ is defined according to Theorem 3.3.*

Proof. Let c be such that $T(F) \leq B(F) + c$ for any sequence of operations F starting at the initial state. For any state s, we define its potential $pot(s)$ by

$$pot(s) = \inf\{B(F) + c - T(F) : F \text{ is a sequence of operations with final state } s\} .$$

We need to write inf instead of min, since there might be infinitely many sequences leading to s. We have $pot(s) \geq 0$ for any s, since $T(F) \leq B(F) + c$ for any sequence F. Thus *pot* is a potential function, and the functions $A_X(s)$ form a family of amortized time bounds. We need to show that $A_X(s) \le B_X(s)$ for all X and s. Let $\varepsilon > 0$ be arbitrary. We shall show that $A_X(s) \le B_X(s) + \varepsilon$. Since ε is arbitrary, this proves that $A_X(s) \le B_X(s)$.

Let F be a sequence with final state s and $B(F) + c - T(F) \le pot(s) + \varepsilon$. Let F' be F followed by X, i.e.,

$$s_0 \xrightarrow{F} s \xrightarrow{X} s' \,.$$

Then $pot(s') \le B(F') + c - T(F')$ by the definition of $pot(s')$, $pot(s) \ge B(F) + c - T(F) - \varepsilon$ by the choice of F, $B(F') = B(F) + B_X(s)$ and $T(F') = T(F) + T_X(s)$ since $F' = F \circ X$, and $A_X(s) = pot(s') - pot(s) + T_X(s)$ by the definition of $A_X(s)$. Combining these inequalities, we obtain

$$\begin{aligned} A_X(s) &\le (B(F') + c - T(F')) - (B(F) + c - T(F) - \varepsilon) + T_X(s) \\ &= (B(F') - B(F)) - (T(F') - T(F) - T_X(s)) + \varepsilon \\ &= B_X(s) + \varepsilon \,. \end{aligned}$$

□

3.4 Stacks and Queues

Sequences are often used in a rather limited way. Let us start with some examples from precomputer days. Sometimes a clerk will work in the following way: the clerk keeps a *stack* of unprocessed files on her desk. New files are placed on the top of the stack. When the clerk processes the next file, she also takes it from the top of the stack. The easy handling of this "data structure" justifies its use; of course, files may stay in the stack for a long time. In the terminology of the preceding sections, a stack is a sequence that supports only the operations *pushBack*, *popBack*, and *last*. We shall use the simplified names *push*, *pop*, and *top* for the three stack operations.

The behavior is different when people stand in line waiting for service at a post office: customers join the line at one end and leave it at the other end. Such sequences are called *FIFO (first in, first out) queues* or simply *queues*. In the terminology of the *List* class, FIFO queues use only the operations *first*, *pushBack*, and *popFront*.

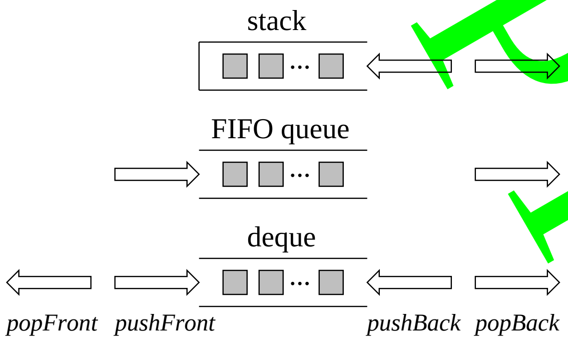


Fig. 3.7. Operations on stacks, queues, and double-ended queues (deques)

The more general *deque* (pronounced "deck"), or *double-ended queue*, allows the operations *first*, *last*, *pushFront*, *pushBack*, *popFront*, and *popBack* and can also be observed at a post office when some not so nice individual jumps the line, or when the clerk at the counter gives priority to a pregnant woman at the end of the line. Figure 3.7 illustrates the access patterns of stacks, queues, and deques.

Exercise 3.18 (the Tower of Hanoi). *In the great temple of Brahma in Benares, on a brass plate under the dome that marks the center of the world, there are 64 disks of pure gold that the priests carry one at a time between three diamond needles according to Brahma's immutable law: no disk may be placed on a smaller disk. At the beginning of the world, all 64 disks formed the Tower of Brahma on one needle. Now, however, the process of transfer of the tower from one needle to another is in mid-course. When the last disk is finally in place, once again forming the Tower of Brahma but on a different needle, then the end of the world will come and all will turn to dust*, [93].[2]

Describe the problem formally for any number k of disks. Write a program that uses three stacks for the piles and produces a sequence of stack operations that transform the state $(\langle k,\ldots,1\rangle,\langle\rangle,\langle\rangle)$ into the state $(\langle\rangle,\langle\rangle,\langle k,\ldots,1\rangle)$.

Exercise 3.19. Explain how to implement a FIFO queue using two stacks so that each FIFO operation takes amortized constant time.

Why should we care about these specialized types of sequence if we already know a list data structure which supports all of the operations above and more in constant time? There are at least three reasons. First, programs become more readable and are easier to debug if special usage patterns of data structures are made explicit. Second, simple interfaces also allow a wider range of implementations. In particular, the simplicity of stacks and queues allows specialized implementations that are more space-efficient than general *List*s. We shall elaborate on this algorithmic aspect in the remainder of this section. In particular, we shall strive for implementations based on arrays rather than lists. Third, lists are not suited for external-memory use because any access to a list item may cause an I/O operation. The sequential access patterns to stacks and queues translate into good reuse of cache blocks when stacks and queues are represented by arrays.

Bounded stacks, where we know the maximal size in advance, are readily implemented with bounded arrays. For unbounded stacks, we can use unbounded arrays. Stacks can also be represented by singly linked lists: the top of the stack corresponds to the front of the list. FIFO queues are easy to realize with singly linked lists with a pointer to the last element. However, deques cannot be represented efficiently by singly linked lists.

We discuss next an implementation of bounded FIFO queues by use of arrays; see Fig. 3.8. We view an array as a cyclic structure where entry zero follows the last entry. In other words, we have array indices 0 to n, and view the indices modulo $n+1$. We

[2] In fact, this mathematical puzzle was invented by the French mathematician Edouard Lucas in 1883.

```
Class BoundedFIFO($n : \mathbb{N}$) of Element
    $b$ : Array $[0..n]$ of Element
    $h = 0 : \mathbb{N}$                // index of first element
    $t = 0 : \mathbb{N}$                // index of first free entry

    Function isEmpty : $\{0,1\}$; return $h = t$

    Function first : Element; assert $\neg$isEmpty; return $b[h]$

    Function size : $\mathbb{N}$; return $(t - h + n + 1)$ mod $(n + 1)$

    Procedure pushBack($x$ : Element)
        assert size$< n$
        $b[t] := x$
        $t := (t + 1)$ mod $(n + 1)$

    Procedure popFront assert $\neg$isEmpty; $h := (h + 1)$ mod $(n + 1)$
```

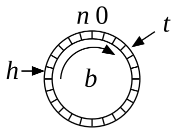


Fig. 3.8. An array-based bounded FIFO queue implementation.

maintain two indices h and t that delimit the range of valid queue entries; the queue comprises the array elements indexed by $h..t-1$. The indices travel around the cycle as elements are queued and dequeued. The cyclic semantics of the indices can be implemented using arithmetics modulo the array size.[3] We always leave at least one entry of the array empty, because otherwise it would be difficult to distinguish a full queue from an empty queue. The implementation is readily generalized to bounded deques. Circular arrays also support the random access operator $[\cdot]$:

```
    Operator $[i : \mathbb{N}]$ : Element; return $b[i + h$ mod $n]$
```

Bounded queues and deques can be made unbounded using techniques similar to those used for unbounded arrays in Sect. 3.2.

We have now seen the major techniques for implementing stacks, queues, and deques. These techniques may be combined to obtain solutions that are particularly suited for very large sequences or for external-memory computations.

Exercise 3.20 (lists of arrays). Here we aim to develop a simple data structure for stacks, FIFO queues, and deques that combines all the advantages of lists and unbounded arrays and is more space-efficient than either lists or unbounded arrays. Use a list (doubly linked for deques) where each item stores an array of K elements for some large constant K. Implement such a data structure in your favorite programming language. Compare the space consumption and execution time with those for linked lists and unbounded arrays in the case of large stacks.

Exercise 3.21 (external-memory stacks and queues). Design a stack data structure that needs $O(1/B)$ I/Os per operation in the I/O model described in Sect. 2.2. It

[3] On some machines, one might obtain significant speedups by choosing the array size to be a power of two and replacing **mod** by bit operations.

suffices to keep two blocks in internal memory. What can happen in a naive implementation with only one block in memory? Adapt your data structure to implement FIFO queues, again using two blocks of internal buffer memory. Implement deques using four buffer blocks.

3.5 Lists Versus Arrays

Table 3.1 summarizes the findings of this chapter. Arrays are better at indexed access, whereas linked lists have their strength in manipulations of sequences at arbitrary positions. Both of these approaches realize the operations needed for stacks and queues efficiently. However, arrays are more cache-efficient here, whereas lists provide worst-case performance guarantees.

Table 3.1. Running times of operations on sequences with n elements. The entries have an implicit $\mathrm{O}(\cdot)$ around them. *List* stands for doubly linked lists, *SList* stands for singly linked lists, *UArray* stands for unbounded arrays, and *CArray* stands for circular arrays

Operation	*List*	*SList*	*UArray*	*CArray*	Explanation of "*"
$[\cdot]$	n	n	1	1	
size	1^*	1^*	1	1	Not with interlist *splice*
first	1	1	1	1	
last	1	1	1	1	
insert	1	1^*	n	n	*insertAfter* only
remove	1	1^*	n	n	*removeAfter* only
pushBack	1	1	1^*	1^*	Amortized
pushFront	1	1	n	1^*	Amortized
popBack	1	n	1^*	1^*	Amortized
popFront	1	1	n	1^*	Amortized
concat	1	1	n	n	
splice	1	1	n	n	
findNext,...	n	n	n^*	n^*	Cache-efficient

Singly linked lists can compete with doubly linked lists in most but not all respects. The only advantage of cyclic arrays over unbounded arrays is that they can implement *pushFront* and *popFront* efficiently.

Space efficiency is also a nontrivial issue. Linked lists are very compact if the elements are much larger than the pointers. For small *Element* types, arrays are usually more compact because there is no overhead for pointers. This is certainly true if the sizes of the arrays are known in advance so that bounded arrays can be used. Unbounded arrays have a trade-off between space efficiency and copying overhead during reallocation.

3.6 Implementation Notes

Every decent programming language supports bounded arrays. In addition, unbounded arrays, lists, stacks, queues, and deques are provided in libraries that are available for the major imperative languages. Nevertheless, you will often have to implement listlike data structures yourself, for example when your objects are members of several linked lists. In such implementations, memory management is often a major challenge.

3.6.1 C++

The class *vector*$\langle$*Element*$\rangle$ in the STL realizes unbounded arrays. However, most implementations never shrink the array. There is functionality for manually setting the allocated size. Usually, you will give some initial estimate for the sequence size n when the *vector* is constructed. This can save you many grow operations. Often, you also know when the array will stop changing size, and you can then force $w = n$. With these refinements, there is little reason to use the built-in C-style arrays. An added benefit of *vector*s is that they are automatically destroyed when the variable goes out of scope. Furthermore, during debugging, you may switch to implementations with bound checking.

There are some additional issues that you might want to address if you need very high performance for arrays that grow or shrink a lot. During reallocation, *vector* has to move array elements using the copy constructor of *Element*. In most cases, a call to the low-level byte copy operation *memcpy* would be much faster. Another low-level optimization is to implement *reallocate* using the standard C function *realloc*. The memory manager might be able to avoid copying the data entirely.

A stumbling block with unbounded arrays is that pointers to array elements become invalid when the array is reallocated. You should make sure that the array does not change size while such pointers are being used. If reallocations cannot be ruled out, you can use array indices rather than pointers.

The STL and LEDA [118] offer doubly linked lists in the class *list*$\langle$*Element*$\rangle$, and singly linked lists in the class *slist*$\langle$*Element*$\rangle$. Their memory management uses free lists for all objects of (roughly) the same size, rather than only for objects of the same class.

If you need to implement a listlike data structure, note that the operator *new* can be redefined for each class. The standard library class *allocator* offers an interface that allows you to use your own memory management while cooperating with the memory managers of other classes.

The STL provides the classes *stack*$\langle$*Element*$\rangle$ and *deque*$\langle$*Element*$\rangle$ for stacks and double-ended queues, respectively. *Deque*s also allow constant-time indexed access using $[\cdot]$. LEDA offers the classes *stack*$\langle$*Element*$\rangle$ and *queue*$\langle$*Element*$\rangle$ for unbounded stacks, and FIFO queues implemented via linked lists. It also offers bounded variants that are implemented as arrays.

Iterators are a central concept of the STL; they implement our abstract view of sequences independent of the particular representation.

3.6.2 Java

The *util* package of the Java 6 platform provides *ArrayList* for unbounded arrays and *LinkedList* for doubly linked lists. There is a *Deque* interface, with implementations by use of *ArrayDeque* and *LinkedList*. A *Stack* is implemented as an extension to *Vector*.

Many Java books proudly announce that Java has no pointers so that you might wonder how to implement linked lists. The solution is that object references in Java are essentially pointers. In a sense, Java has *only* pointers, because members of non-simple type are always references, and are never stored in the parent object itself.

Explicit memory management is optional in Java, since it provides garbage collections of all objects that are not referenced any more.

3.7 Historical Notes and Further Findings

All of the algorithms described in this chapter are “folklore”, i.e., they have been around for a long time and nobody claims to be their inventor. Indeed, we have seen that many of the underlying concepts predate computers.

Amortization is as old as the analysis of algorithms. The *bank account* and *potential* methods were introduced at the beginning of the 1980s by R. E. Brown, S. Huddlestone, K. Mehlhorn, D. D. Sleator, and R. E. Tarjan [32, 95, 182, 183]. The overview article [188] popularized the term *amortized analysis*, and Theorem 3.4 first appeared in [127].

There is an arraylike data structure that supports indexed access in constant time and arbitrary element insertion and deletion in amortized time $\mathrm{O}(\sqrt{n})$. The trick is relatively simple. The array is split into subarrays of size $n' = \Theta(\sqrt{n})$. Only the last subarray may contain fewer elements. The subarrays are maintained as cyclic arrays, as described in Sect. 3.4. Element i can be found in entry $i \textbf{ mod } n'$ of subarray $\lfloor i/n' \rfloor$. A new element is inserted into its subarray in time $\mathrm{O}(\sqrt{n})$. To repair the invariant that subarrays have the same size, the last element of this subarray is inserted as the first element of the next subarray in constant time. This process of shifting the extra element is repeated $\mathrm{O}(n/n') = \mathrm{O}(\sqrt{n})$ times until the last subarray is reached. Deletion works similarly. Occasionally, one has to start a new last subarray or change n' and reallocate everything. The amortized cost of these additional operations can be kept small. With some additional modifications, all deque operations can be performed in constant time. We refer the reader to [107] for more sophisticated implementations of deques and an implementation study.