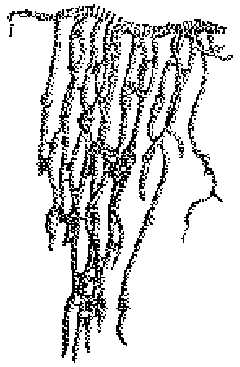
Chapter 2



Representing Sequences by Arrays and Linked Lists

Perhaps the world's oldest data structures were tablets in cuneiform script used more than 5000 years ago by custodians in Sumerian temples. They kept lists of goods, their quantities, owners and buyers. The left picture shows an example. Possibly this was 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 you see in the right picture served a similar purpose in the Inca empire using knots in colored strings arranged sequentially on a master string. Probably it is easier to maintain and use data on tablets than using knotted string, but one would not want to haul stone tablets over Andean mountain trails. Obviously, it makes sense to consider different representations for the same kind of logical 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 $s = \langle e_0, \dots, e_{n-1} \rangle$ are arranged in a linear order — in contrast to the trees and graphs in Chapters **??** and 7, or the unordered hash tables discussed in Chapter 3. There are two basic ways to specify 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 sequence s. Since using arrays is the bread and butter of any imperative programming language, we take *bounded arrays* for granted in this chapter. In a *bounded* data structure, the memory requirements are known in advance, at the latest when the data structure is created. Appendix **??** explains how bounded arrays can can be implemented on top of our machine model. Section 2.1

immediately proceeds to *unbounded arrays* that can adaptively grow and shrink as elements are inserted and removed. The analysis of unbounded arrays introduces the concept of *amortized analysis*.

Another way to specify elements of a sequence is relative to other elements. For example, one could ask for the successor of an element e, for the predecessor of an element e' or for the subsequence $\langle e, \dots, e' \rangle$ of elements between e and e'. Although relative access can be simulated using array indexing, we will see in Section 2.2 that sequences represented using pointers to successors and predecessors are more flexible. In particular, it becomes easier to insert or remove arbitrary pieces of a sequence.

In many algorithms, it does not matter very much whether sequences are implemented using arrays or linked lists because only a very limited set of operations is needed that can be handled efficiently using either representation. Section 2.3 introduces stacks and queues, which are the most common data types of that kind. A more comprehensive account of the zoo of operations available for sequences is given in Section 2.4.

2.1 Unbounded Arrays

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

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

Why are unbounded arrays important? Often, because 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 to actually load it into the array.

In principle, implementing such an unbounded array is easy. We emulate an unbounded array u with n elements by a dynamically allocated bounded array b with $w \geq n$ entries. The first n entries of b are used to store the elements of b. The last $w - n$ entries of b are unused. As long as $w > n$, *pushBack* simply increments n and uses one of the previously unused entries of b for the new element. When $w = n$, the next *pushBack* allocates a new bounded array b' that is a constant factor larger (say a factor two). To reestablish the invariant that u is stored in b, the content of b is 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* looks 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 2.1 gives the complete pseudocode for an unbounded array class. Growing and shrinking is performed using the same utility procedure *reallocate*.

Amortized Analysis of Unbounded Arrays

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, a single insertion into a large array might incur a cost of n. Exercise 2.2 asks you to give an example, where almost every call of *pushBack* and *popBack* is expensive if we make a seemingly harmless change in the algorithm. We now show that such a situation cannot happen for our implementation. Although some isolated procedure calls might be expensive, they are always rare, regardless of what sequence of operations we execute.

Lemma 2.1 *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 $O(m)$.*

If we divide the total cost for the operations in σ by the number of operations, we get a constant. Hence, it makes sense to attribute constant cost to each operation. Such costs are called *amortized costs*. The usage of the term *amortized* is similar to its general usage, but it avoids some common pitfalls. "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 is very small — this investment will amortize" 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 amortized.

In computer science we insist on amortization. We are free to assign arbitrary *amortized costs* to operations but they are only correct if the sum of the amortized costs over any sequence of operations is never below the actual cost. Using the notion of amortized costs, we can reformulate Lemma 2.1 more elegantly to allow direct comparisons with other data structures.

Corollary 2.2 *Unbounded arrays implement the operation $[\cdot]$ in worst case constant time and the operations pushBack and popBack in amortized constant time.*

```
Class UArray of Element
    Constant $\beta$=2 : $\mathbb{R}_+$                                  // growth factor
    Constant $\alpha$=$\beta^2$ : $\mathbb{R}_+$                         // worst case memory blowup
    $w$=1 : $\mathbb{N}$                                                 // allocated size
    $n$=0 : $\mathbb{N}$          // current size. invariant $n \leq w$
    $b$ : Array $[0..w-1]$ of Element                       // $b \rightarrow$

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

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

    Procedure pushBack($e$ : Element)                      // Example for $n = w = 4$:
        if $n = w$ then                                      // $b \rightarrow$
            reallocate($\beta n$)                             // $b \rightarrow$
        // For the analysis: pay insurance here.
        $b[n]$ := $e$                                         // $b \rightarrow$
        $n$++                                                 // $b \rightarrow$

    Procedure popBack()                                   // Example for $n = 5$, $w = 16$:
        assert $n > 0$                               // $b \rightarrow$
        $n$−−                                        // $b \rightarrow$
        if $w \geq \alpha n \wedge n > 0$ then                  // reduce waste of space
            reallocate($\beta n$)                             // $b \rightarrow$

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

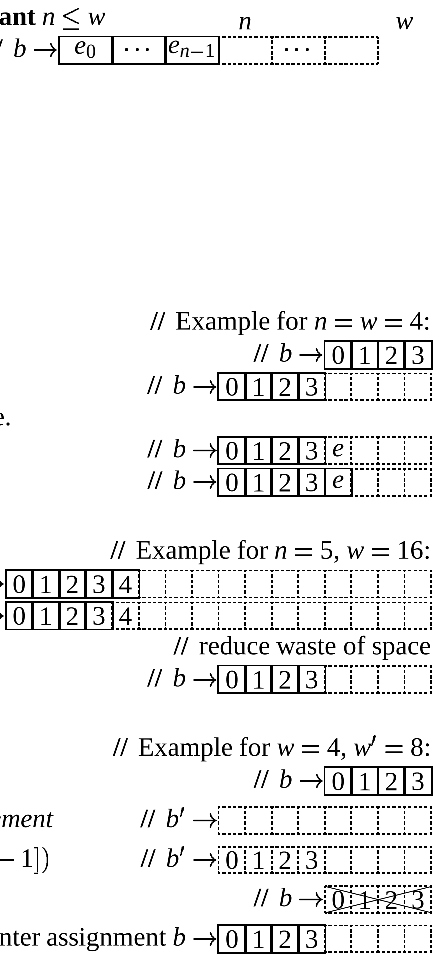


Figure 2.1: Unbounded arrays

To prove Lemma 2.1, we use the *accounting method*. Most of us have already used this approach because it is the basic idea behind an insurance. For example, when you rent a car, in most cases you also have to buy an insurance that covers the ruinous costs you could incur by causing an accident. Similarly, we force all calls to *pushBack* and *popBack* to buy an insurance against a possible call of *reallocate*. The cost of the insurance is put on an account. If a *reallocate* should actually become necessary, the responsible call to *pushBack* or *popBack* does not need to pay but it is allowed to use previous deposits on the insurance account. What remains to be shown is that the account will always be large enough to cover all possible costs.

Proof: Let m' denote the total number of elements copied in calls of *reallocate*. The total cost incurred by calls in the operation sequence σ is $O(m+m')$. Hence, it suffices to show that $m' = O(m)$. Our unit of cost is now the cost of one element copy.

For $\beta = 2$ and $\alpha = 4$, we require an insurance of 3 units from each call of *pushBack* and claim that this suffices to pay for all calls of *reallocate* by both *pushBack* and *popBack*. (Exercise 2.4 asks you to prove that for general β and $\alpha = \beta^2$ an insurance of $\frac{\beta+1}{\beta-1}$ units is sufficient.)

We prove by induction over the calls of *reallocate* that immediately after the call there are at least n units left on the insurance account.

First call of *reallocate*: The first call grows w from 1 to 2 after at least one call of *pushBack*. We have $n = 1$ and $3-1 = 2 > 1$ units left on the insurance account.

For the induction step we prove that $2n$ units are on the account immediately before the current call to *reallocate*. Only n elements are copied leaving n units on the account — enough to maintain our invariant. The two cases in which *reallocate* may be called are analyzed separately.

***pushBack* grows the array:** The number of elements n has doubled since the last *reallocate* when at least $n/2$ units were left on the account by the induction hypothesis. The $n/2$ new elements paid $3n/2$ units giving a total of $2n$ units.

***popBack* shrinks the array:** The number of elements has halved since the last *reallocate* when at least $2n$ units were left on the account by the induction hypothesis. ▮

Exercises

Exercise 2.1 (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, $c = 0$.

a) What is the worst case execution time of an increment or a decrement as a function of m? Assume that you can only work at one bit per step.

b) Prove that the amortized cost of increments is constant if there are no decrements.

c) Give a sequence of m increment and decrement operations that has cost $\Theta(m\log m)$

d) Give a representation of counters such that you can achieve worst case constant time for increment and decrement. (Here you may assume that m is known in advance.)

e) Consider the representation of counters where each digit d_i is allowed to take values from $\{-1,0,1\}$ so that 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.

Exercise 2.2 Your manager asks you whether it is not too wasteful to shrink an array only when already three fourths of b are unused. He proposes to shrink it already when $w = n/2$. 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 were implemented.

Exercise 2.3 (Popping many elements.) Explain how to implement the operation *popB* that removes the last k elements in amortized constant time independent of k. Hint: The existing analysis is very easy to generalize.

Exercise 2.4 (General space time tradeoff) Generalize the proof of Lemma 2.1 for general β and $\alpha = \beta^2$. Show that an insurance of $\frac{\beta+1}{\beta-1}$ units paid by calls of *pushBack* suffices to pay for all calls of *reallocate*.

***Exercise 2.5** We have not justified the relation $\alpha = \beta^2$ in our analysis. Prove that any other choice of α leads to higher insurance costs for calls of *pushBack*. Is $\alpha = \beta^2$ still optimal if we also require an insurance from *popBack*? (Assume that we now want to minimize the maximum insurance of any operation.)

Exercise 2.6 (Worst case constant access time) Suppose for a real time application you need an unbounded array data structure with *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 the small array is completely exhausted.

Exercise 2.7 (Implicitly growing arrays) Implement an unbounded array 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 2.8 (Sparse arrays) Implement a bounded array with constant time for allocating the array and constant amortized time for operation $[\cdot]$. As in the previous exercise, a read access to an array entry that was never written should return $\perp$. Note that you cannot make any assumptions on the contents of a freshly allocated array. Hint: Never explicitly write default values into entries with undefined value. Store the elements that actually have a nondefault value in arbitrary order in a separate data structure. Each entry in this data structure also stores its position in the array. The array itself only stores references to this secondary data structure. The main issue is now how to find out whether an array element has ever been written.

2.2 Linked Lists

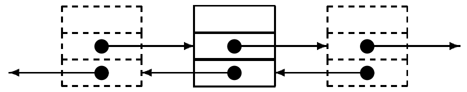
In this section we exploit the approach of representing sequences by storing pointers to successor and predecessor with each list element. A good way to think of such linked lists is to imagine a chain where one element is written on each link. Once we get hold of one link of the chain, we can retrieve all elements by exploiting the fact that the links of the chain are forged together. Section 2.2.1 explains this idea. In Section 2.2.2 we outline how many of the operations can still be performed if we only store successors. This is more space efficient and somewhat faster.

2.2.1 Doubly Linked Lists

Figure 2.2 shows the basic building block of a linked list. A list item (a link of a chain) stores one element and pointers to successor and predecessor. 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 make a simple and innocent looking decision and the basic design of our list data structure will follow from that: The successor of the predecessor of an item must be the original item, and the same holds for the predecessor of a successor.

If all items fulfill this invariant, they will form a collection of cyclic chains. This may look strange, since we want to represent sequences rather than loops. Sequences have a start and an end, wheras loops have neither. Most implementations of linked lists therefore go a different way, and treat the first and last item of a list differently.

```
 Type Handle = Pointer to Item
Function info(a : Handle) return $a \rightarrow e$
Function succ(a : Handle) return $a \rightarrow next$
Function pred(a : Handle) return $a \rightarrow prev$

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

    // Remove $\langle a, \ldots, b\rangle$ from its current list and insert it after $t$
    // $\ldots, a', a, \ldots, b, b', \ldots, t, t', \ldots) \mapsto (\ldots, a', b', \ldots, t, a, \ldots, b, t', \ldots)$
    Procedure splice(a,b,t : Handle)
        assert $b$ is not before $a \wedge t \notin \langle a, \ldots, b\rangle$
        // Cut out $\langle a, \ldots, b\rangle$
        $a' := a \rightarrow prev$
        $b' := b \rightarrow next$
        $a' \rightarrow next := b'$                    //
        $b' \rightarrow prev := a'$                    //

        // insert $\langle a, \ldots, b\rangle$ after $t$
        $t' := t \rightarrow next$                     //

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

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



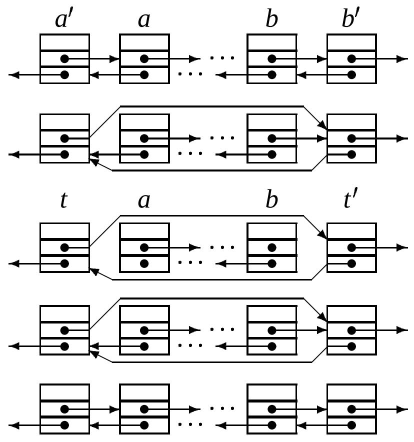


Figure 2.2: Low level list labor.

Unfortunately, this makes the implementation of lists more complicated, more error-prone and somewhat slower. Therefore, we stick to the simple cyclic internal representation. Later we hide the representation from the user interface by providing a list data type that “simulates” lists with a start and an end.

For conciseness, we implement all basic list operations in terms of the single operation *splice* depicted in Figure 2.2. *splice* cuts out a sublist from one list and inserts it after some target item. The target can be either in the same list or in a different list but it must not be inside the sublist.

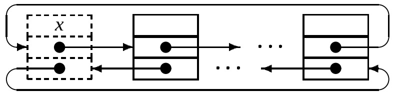
Since *splice* never changes the number of items in the system, we assume that there is one special list *freeList* that keeps a supply of unused elements. When inserting new elements into a list, we take the necessary items from *freeList* and when deleting elements we return the corresponding items to *freeList*. The function *checkFreeList* allocates memory for new items when necessary. We defer its implementation to Exercise 2.11 and a short discussion in Section 2.5.

It remains to decide how to simulate the start and end of a list. The class *List* in Figure 2.3 introduces a dummy item *h* that does not store any element but seperates the first element from the last element in the cycle formed by the list. By definition of *Item*, *h* points to the first “proper” item as a successor and to the last item as a predecessor. In addition, a handle *head*() pointing to *h* can be used to encode a position before the first element or after the last element. Note that there are $n+1$ possible positions for inserting an element into an list with *n* elements so that an additional item is hard to circumvent if we want to code handles as pointers to items.

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. Figure 2.3 gives typical examples.

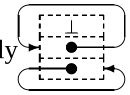
The dummy header can also be useful for other operations. For example consider the following code for finding the next occurence of *x* starting at item *from*. If *x* is not present, *head*() should be returned.

```
**Function** $findNext(x : Element; from : Handle) : Handle$
    $h.e = x$                                // Sentinel
    **while** $from \to e \neq x$ **do**
        $from := from \to next$
    **return** $from$
```



We use the header as a *sentinel*. A sentinel is a dummy element in a data structure that makes sure that some loop will terminate. By storing the key we are looking for in the

```
**Class** $List$ **of** $Element$
    // Item $h$ is the predecessor of the first element
    // Item $h$ is the successor of the last element.
    $h = \begin{pmatrix} \perp \\ \textbf{this} \\ \textbf{this} \end{pmatrix} : Item$            // empty list $\langle\rangle$ with dummy item only
    // Simple access functions
    **Function** $head() : Handle$; **return address of** $h$ // Pos. before any proper element

    **Function** $isEmpty()$ : **boolean** ; **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.
    // $(\langle\ldots,a,b,c\ldots,a',c',\ldots\rangle) \mapsto (\langle\ldots,a,c\ldots,a',b,c',\ldots\rangle)$
    **Procedure** $moveAfter(b,a' : Handle)\ splice(b,b,a')$
    **Procedure** $moveToFront(b : Handle)\ moveAfter(b, head())$
    **Procedure** $moveToBack(b : Handle)\ moveAfter(b, last())$

    // 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** $popBack()\ remove(last())$

    // $\langle\ldots,a,b,\ldots\rangle \mapsto \langle\ldots,a,e,b,\ldots\rangle$
    **Procedure** $insertAfter(x : Element; a : Handle)$
        $checkFreeList()$      // make sure $freeList$ is nonempty. See also Exercise 2.11
        $moveAfter(freeList.first(), a)$
        $a \to next \to e = x$

    **Procedure** $insertBefore(x : Element; b : Handle)\ insertAfter(e, pred(b))$
    **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** $concat(o : List)$
        $splice(o.first(), o.last(), head())$

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



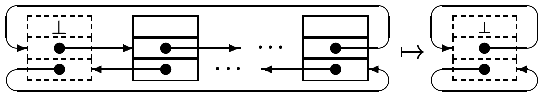


Figure 2.3: Some constant time operations on doubly linked lists.

header, we make sure that the search terminates even if x is originally not present in the list. This trick saves us an additional test in each iteration whether the end of the list is reached.

Maintaining the Size of a List

In our simple list data type it not possible to find out the number of elements 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 like *splice* would only need handles of the items involved. For example, consider the following code for moving an element from one list to another:

```
// $(\langle\ldots,a,b,c\ldots\rangle,\langle\ldots,a',c',\ldots\rangle) \mapsto (\langle\ldots,a,c\ldots\rangle,\langle\ldots,a',b,c',\ldots\rangle)$
Procedure moveAfter(b, a' : Handle; o : List)
    splice(b,b,a')
    size−−
    o.size++
```

Interfaces of list data types should require this information even if *size* is not maintained so that the data type remains interchangable with other implementations.

A more serious problem is that operations that move around sublists beween lists cannot be implemented in constant time any more if *size* is to be maintained. Exercise 2.15 proposes a compromise.

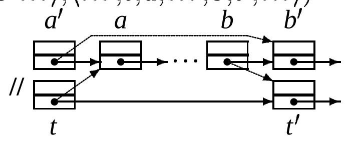
2.2.2 Singly Linked Lists

The many pointers used by doubly linked lists makes programming quite comfortable. Singly linked lists are the lean and mean sisters of doubly linked lists. *SItem*s scrap the *prev* pointer and only store *next*. This makes singly linked lists more space efficient and often faster than their doubly linked brothers. The price we pay is that some operations can no longer be performed in constant time. For example, we cannot remove an *SItem* if we do not know its predecessor. Table 2.1 gives an overview of constant time operations for different implementations of sequences. We will see several applications of singly linked lists, e.g., in Section 3.1, . In particular, we can use singly linked lists to implement free lists of memory managers — even for items of doubly linked lists.

We can adopt the implementation approach from 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 of *List*s can

still be performed if we change the interface. 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)
    $\begin{pmatrix} a'\rightarrow next \\ t\rightarrow next \\ b\rightarrow next \end{pmatrix} := \begin{pmatrix} b\rightarrow next \\ a'\rightarrow next \\ t\rightarrow next \end{pmatrix}$
```



Similarly, *findNext* should not return the handle of the *SItem* with the next fit but its *predecessor*. This way it remains possible to remove the element found. A useful addition to *SList* is a pointer to the last element because then we can support *pushBack* in constant time. We leave the details of an implementation of singly linked lists to Exercise 2.17.

Exercises

Exercise 2.9 Prove formally that items of doubly linked lists fulfilling the invariant $next \rightarrow prev = prev \rightarrow next = \textbf{this}$ form a collection of cyclic chains.

Exercise 2.10 Implement a procudure *swap* similar to *splice* that swaps two sublists in constant time
$(\langle\ldots,a',a,\ldots,b,b',\ldots\rangle,\langle\ldots,c',c,\ldots,d,d',\ldots\rangle) \mapsto$
$(\langle\ldots,a',c,\ldots,d,b',\ldots\rangle,\langle\ldots,c',a,\ldots,b,d',\ldots\rangle)$.
Can you view *splice* as a special case of *swap*?

Exercise 2.11 (Memory mangagement for lists) Implement the function *checkFreelist* called by *insertAfter* in Figure 2.3. 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 2.12 Give a constant time implementation for rotating a list right: $\langle a,\ldots,b,c\rangle$ $\langle c,a,\ldots,b\rangle$. Generalize your algorithm to rotate sequence $\langle a,\ldots,b,c,\ldots,d\rangle$ to $\langle c,\ldots,d$ in constant time.

Exercise 2.13 (Acyclic list implementation.) Give an alternative implementation of *List* that does not need the dummy item h and encodes *head*() 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 disadvantag of this implementation compared to the algorithm from Figure 2.3.

Exercise 2.14 *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 small list with 100 elements, or for a large list with 10 000 000 elements respectively? Why is the relative speed difference dependent on the size of the list?

Exercise 2.15 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 only recomputed when needed.

Exercise 2.16 Explain how the operations *remove*, *insertAfter*, and *concat* have to be modified to keep track of the length of a *List*.

Exercise 2.17 Implement classes *SHandle*, *SItem*, and *SList* for singly linked lists in analogy to *Handle*, *Item*, and *List*. Support all functions that can be implemented to run in constant time. Operations *head*, *first*, *last*, *isEmpty*, *popFront*, *pushFront*, *pushBack*, *insertAfter*, *concat*, and *makeEmpty* should have the same interface as before. Operations *moveAfter*, *moveToFront*, *moveToBack*, *remove*, *popFront*, and *findNext* need different interfaces.

2.3 Stacks and Queues

Sequences are often used in a rather limited way. Let us again start with examples from precomputer days. Sometimes a clerk tends to work in the following way: There is a *stack* of files on his desk that he should work on. New files are dumped on the top of the stack. When he processes the next file he also takes it from the top of the stack. The easy handling of this "data structure" justifies its use as long as no time-critical jobs are forgotten. In the terminology of the preceeding sections, a stack is a sequence that only supports the operations *pushBack*, *popBack*, and *last*. In the following we will use the simplified names *push*, *pop*, and *top* for these three operations on stacks.

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

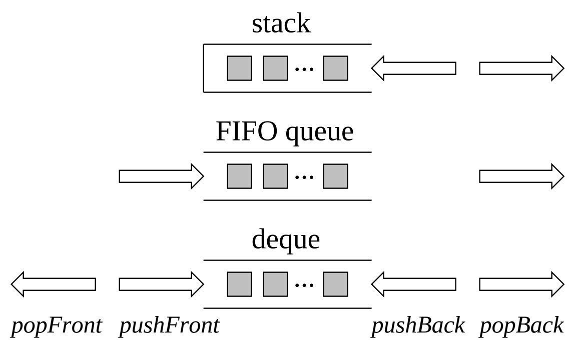


Figure 2.4: Operations on stacks, queues, and double ended queues (deques).

The more general *deque*[1], or *double ended queue*, that allows operations *first*, *last*, *pushFront*, *pushBack*, *popFront* and *popBack* might also be observed at a post office, when some not so nice guy jumps the line, or when the clerk at the counter gives priority to a pregnant woman at the end of the queue. Figure 2.4 gives an overview of the access patterns of stacks, queues and deques.

Why should we care about these specialized types of sequences if *List* can implement them all? There are at least three reasons. A program gets more readable and easier to debug if special usage patterns of data structures are made explicit. Simple interfaces also allow a wider range of implementations. In particular, the simplicity of stacks and queues allows for specialized implementions that are more space efficient than general *List*s. We will elaborate this algorithmic aspect in the remainder of this section. In particular, we will strive for implementations based on arrays rather than lists. Array implementations may also be significantly faster for large sequences because sequential access patterns to stacks and queues translate into good reuse of cache blocks for arrays. In contrast, for linked lists it can happen that each item access causes a cache miss.

Bounded stacks, where we know the maximal size in advance can easily be implemented with bounded arrays. For unbounded stacks we can use unbounded arrays. Stacks based on singly linked lists are also easy once we have understood that we can use *pushFront*, *popFront*, and *first* to implement *push*, *pop*, and *top* respectively.

Exercise 2.19 gives hints how to implement unbounded stacks and queues that support worst case constant access time and are very space efficient. Exercise 2.20 asks you to design stacks and queues that even work if the data will not fit in main memory. It goes without saying that all implementations of stacks and queues described here

[1]Deque is pronounced like "deck".

can easily be augmented to support *size* in constant time.

```
Class BoundedFIFO(n : ℕ) of Element
  b : Array [0..n] of Element
  h=0 : ℕ                              // index of first element
  t=0 : ℕ                              // index of first free entry
  Function isEmpty() : boolean ; return $h = t$
  Function first() : Element; assert ¬isEmpty(); return $b[h]$
  Function size() : ℕ; return $(t - h + n)$ mod $(n + 1)$
  Procedure pushBack(x : Element)
    assert size()< n
    $b[t] := x$
    $t := (t + 1)$ mod $(n + 1)$
  Procedure popFront() assert ¬isEmpty(); $h := (h + 1)$ mod $(n + 1)$
```

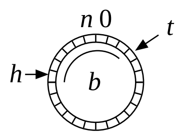


Figure 2.5: A bounded FIFO queue using arrays.

FIFO queues are easy to implement with singly linked lists with a pointer to the last element. Figure 2.5 gives an implementation of bounded FIFO queues using arrays. The general idea is to view the array as a cyclic structure where entry zero is the successor of the last entry. Now it suffices to maintain two indices delimiting the range of valid queue entries. These 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.[2] Our implementation always leaves one entry of the array empty because otherwise it would be difficult to distinguish a full queue from an empty queue. Bounded queues can be made unbounded using similar techniques as for unbounded arrays in Section 2.1.

Finally, deques cannot be implemented efficiently using singly linked lists. But the array based FIFO from Figure 2.5 is easy to generalize. Circular arrary can also support access using $[\cdot]$.

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

[2]On some machines one might get signifi cant speedups by choosing the array size as a power of two and replacing **mod** by bit operations.

Exercises

Exercise 2.18 (The towers 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 these diamond needles according to Brahma's immutable law: No disk may be placed on a smaller disk. In 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 will come the end of the world and all will turn to dust. [25].*[3]

Describe the problem formally for any number k of disks. Write a program that uses three stacks for the poles 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 2.19 (Lists of arrays) Here we want 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 for large queues than either of them. 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 space consumption and execution time to linked lists and unbounded arrays for large stacks and some random sequence of pushes and pops

Exercise 2.20 (External memory stacks and queues) Design a stack data structure that needs $O(1/B)$ I/Os per operation in the I/O model from Section **??**. It suffices to keep two blocks in internal memory. What can happen in a naive implementaiton with only one block in memory? Adapt your data structure to implement FIFOs, again using two blocks of internal buffer memory. Implement deques using four buffer blocks.

Exercise 2.21 Explain how to implement a FIFO queue using two stacks so that each FIFO operations takes amortized constant time.

2.4 Lists versus Arrays

Table 2.1 summarizes the execution times of the most important operations discussed in this chapter. Predictably, arrays are better at indexed access whereas linked lists have their strenghts at sequence manipulation at arbitrary positions. However, both basic approaches can implement the special operations needed for stacks and queues

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

Table 2.1: Running times of operations on sequences with n elements. Entries have an implicit $O(\cdot)$ around them.

Operation	*List*	*SList*	*UArray*	*CArray*	explanation of '*'
$[\cdot]$	n	n	1	1	
$\|\cdot\|$	1^*	1^*	1	1	not with inter-list *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

roughly equally well. Where both approaches work, arrays are more cache efficient whereas linked lists provide worst case performance guarantees. This is particularly true for all kinds of operations that scan through the sequence; *findNext* is only one example.

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 elements are much larger than one or two pointers. For small *Element* types, arrays have the potential to be more compact because there is no overhead for pointers. This is certainly true if the size of the arrays is known in advance so that bounded arrays can be used. Unbounded arrays have a tradeoff between space efficiency and copying overhead during reallocation.

2.5 Implementation Notes

C++

Unbounded arrays are implemented as class *vector* in the standard library. Class *vector*⟨*Element*⟩ is likely to be more efficient than our simple implementation. It gives you additional control over the allocated size w. 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 builtin C style arrays. An added benefit of *vector*s is that they are automatically destructed when the variable gets out of scope. Furthermore, during debugging you can easily switch to implementations with bound checking.

There are some additional performance 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. Perhaps a very clever compiler could perform this optimization automatically, but we doubt that this happens in practice. Another low level optimization is to implement *reallocate* using the standard `C` function *realloc*

```
b = realloc(b, sizeof(Element));
```

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 used. If reallocations cannot be ruled out, you can use array indices rather than pointers.

The C++ standard library and LEDA offer doubly linked lists in the class *list*⟨*Element* and singly linked lists in the class *slist*⟨*Element*⟩. The implementations we have seen perform rather well. Their memory management uses free lists for all object of (roughly) the same size, rather than only for objects of the same class. Nevertheless, you might have to implement list like data structures yourself. Usually, the reason will be that your elements are part of complex data structures, and being arranged in a list is only one possible state of affairs. In such implementations, memory management is often the main challenge. Note that the operator *new* can be redefined for each class. The standard library class *allocator* offers an interface that allows you to roll your own memory management while cooperating with the memory managers of other classes.

The standard C++ library implements classes *stack*⟨*Element*⟩ and *deque*⟨*Element*⟩ for stacks and double ended queues respectively. C++ *deque*s also allow constant time indexed access using $[\cdot]$. LEDA offers classes *stack*⟨*Element*⟩ and *queue*⟨*Element*⟩ 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 C++ standard library that implement our abstract view of sequences independent of the particular representation.

Java

The *util* package of the Java 2 platform provides *Vector* for unbounded arrays, *LinkedList* for doubly linked lists, and *Stack* for stacks. There is a quite elaborate hierarchy of abstractions for sequence like data types.

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 nonsimple 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 needed any more.

Java does not allow the specification of container classes like lists and arrays for a particular class *Element*. Rather, containers always contain *Object*s and the application program is responsible for performing appropriate casts. Java extensions for better support of generic programming are currently a matter of intensive debate.

2.6 Further Findings

Most 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 concepts predate computers.

Amortization is as old as the analysis of algorithms. The accounting method and the even more general *potential method* were introduced in the beginning of the 80s by R.E. Brown, S. Huddlestone, K. Mehlhorn. D.D. Sleator und R.E. Tarjan [11, 26, 52, 53]. The overview article [55] popularized the term *amortized analysis*.

There is an array-like data structure, that supports indexed access in constant time and arbitrary element insertion and deletion in amortized time $O(\sqrt{n})$ The trick is relatively simple. The array is split into subarrays of size $n' = \Theta(\sqrt{n})$. Only the last subarray may contain less elements. The subarrays are maintained as cyclic arrays as described in Section 2.3. Element i can be found in entry i **mod** n' of subarray $\lfloor i/n' \rfloor$. A new element is inserted in its subarray in time $O(\sqrt{n})$. 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 $O(n/n') = O(\sqrt{n})$ 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. Katajainen and Mortensen have designed more sophisticated implementations of deques and present an implementation study [32].