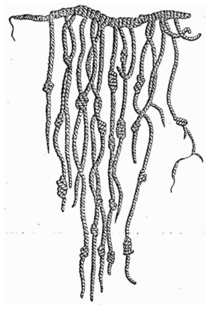
3

Representing Sequences by Arrays and Linked Lists



Perhaps the world's oldest data structures were the tablets in cuneiform script used more than 5000 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.[1] *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 [224] 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 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 ith element of a sequence s. Arrays are the basis of many parallel algorithms. In Sect. 3.1 we explain some of the basic approaches. Our pseudocode directly 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 maximum size is known in advance. In Sect. 3.4, we

[1] This 4600-year-old tablet contains a list of gifts to the high priestess of Adab (see `commons.wikimedia.org/wiki/Image:Sumerian_26th_c_Adab.jpg`).

introduce *dynamic* or *unbounded 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 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. 3.2 that a list-based representation of sequences is more flexible. In particular, it becomes easier to insert or remove arbitrary pieces of a sequence. On the other hand, parallel processing of linked lists is difficult. In Sect. 3.3 we get a glimpse how to do it anyway.

Many algorithms use sequences in a quite limited way. Often only the front and/or the rear of the sequence is read and modified. Sequences that are used in this restricted way are called *stacks*, *queues*, and *deques*. We discuss them in Sects. 3.6 and 3.7. In Sect. 3.8, we summarize the findings of the chapter.

3.1 Processing Arrays in Parallel

Arrays are an important data structure for parallel processing, since we can easily assign operations on different array elements to different PEs. Suppose we want to assign the elements of an array $a[0..n-1]$ to p PEs numbered $0..p-1$. There are many ways for distributing the elements over the PEs. Figure 3.1 gives examples. Perhaps the most natural one – *blocked assignment* – assigns up to $\lceil n/p \rceil$ consecutive array elements to each PE, for example by mapping element $a[i]$ to PE $\lfloor i/\lceil n/p \rceil \rfloor$. This works well if the amount of work required for the different array elements is about the same. Moreover, this assignment is also cache-efficient. To simplify the notation, let us now assume that p divides n.

Another natural assignment is *round robin* (also called *cyclic*) where PE i works on array elements $a[j]$ with $j \bmod p = i$. This is less cache-efficient than blocked assignment but may sometimes assign the work more uniformly.

Exercise 3.1. Since cyclic assignment is less cache-efficient than blocked assignment, one also uses *block cyclic* assignment, where blocks of size B are cyclically assigned to PEs. Work out formulae that specify which elements are assigned to each PE (see Fig. 3.1 for an example).

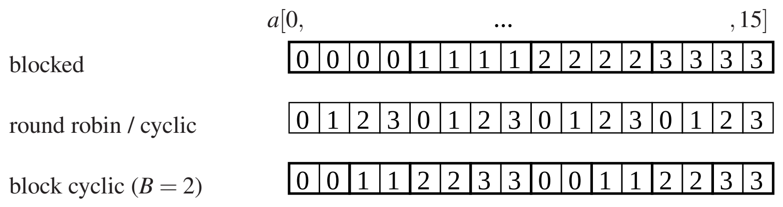


Fig. 3.1. Simple assignments of 16 array elements $a[0..15]$ to PEs 0..3

In Chapter 14 we shall discuss more advanced ways to assign work (e.g., array elements) to PEs in a load balanced way.

On a distributed-memory machine, we usually want to store an array element on the same PE that processes it. This important principle is known as *owner computes*. Hence, our logical array a will be stored in a distributed way – one piece of a is stored on each PE. This distribution is also known as *sharding* or *(horizontal) partitioning*. The distribution principle naturally transfers to more complex data structures such as multidimensional arrays or graphs (see also Sect. 8.6). Note that the "owner computes" principle can be applied to any approach to array partitioning – blocked, cyclic, block cyclic, explicitly load-balanced, ...

Let us look at a simple example. Suppose we want to double all elements of a using blocked assignment. Suppose PE i of a distributed-memory machine stores elements $i \cdot n/p..(i+1)\cdot n/p-1$ in a local array $a[0..n/p-1]$. Then then SPMD pseudocode for this doubling task is

```
**for** $i := 0$ **to** $n/p-1$ **do** $a[i] *= 2$
```

and this takes time $\mathrm{O}(n/p)$, requiring no communication.

Of course, parallel programming is less simple most of the time. In particular, computations usually involve several array elements at once. In Sect. 2.10, we already discussed the task of summing all elements of an array. As another example, suppose we have an array $a[0..n+1]$ and want to compute the average of $a[i-1]$, $a[i]$, and $a[i+1]$ for $i \in 1..n$; the boundary values $a[0]$ and $a[n+1]$ are fixed and are not changed.[2] Such a computation is frequent in the approximate numerical solution of partial differential equations. On a shared-memory machine, the task is quite simple. This time we use explicit loop parallelism. We allocate a second array b and say

```
**for** $i := 1$ **to** $n$ **do**$\|$          // use blocked assignment of loop iterations to PEs
    $b[i] := (a[i-1] + a[i] + a[i+1])/3$.
```

On a distributed-memory machine, we need explicit communication to make sure that all the required data is available. We want PE i to compute components $i\cdot n/p+1$ to $(i+1)\cdot n/p$ of the result vector, as shown below for $n=8$ and $p=2$:

global view	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$	$a[7]$	$a[8]$	$a[9]$
local view, PE 0	$a[0]^*$	$a[1]^*$	$a[2]^*$	$a[3]^*$	$a[4]^*$	$a[5]$				
results computed by PE 0		$b[1]$	$b[2]$	$b[3]$	$b[4]$					
local view, PE 1					$a[0]$	$a[1]^*$	$a[2]^*$	$a[3]^*$	$a[4]^*$	$a[5]^*$
results computed by PE 1						$b[1]$	$b[2]$	$b[3]$	$b[4]$	

We therefore allocate to each PE a local array a with $n/p+2$ cells, where $a[0]$ and $a[n/p+1]$ play a special role. PE i can compute its part of the output if it has access to elements $i\cdot n/p$ to $(i+1)n/p+1$ of the input array, i.e., some elements of the input array must be stored in two local arrays as shown above. However, initially each element of the input array is available on only one of the PEs. We assume

[2] We are computing n values $b[1]$ to $b[n]$ and, for simplicity, want to stick to our convention that p divides n. Therefore, we use an input array of size $n+2$.

that initially the array is distributed over the PEs as follows. Elements 0 to n/p are stored in PE 0, elements $n/p+1$ to $2n/p$ are stored in PE 1, ..., elements $i \cdot n/p+1$ to $(i+1) \cdot n/p$ are stored in PE i, ..., and elements $(p-1) \cdot n/p+1$ to $n+1$ are stored in PE $p-1$. In the figure above, the initial distribution of the array elements is indicated by the symbol *.

As said above, PE i can compute its part of the output if it has access to elements $i \cdot n/p$ to $(i+1)n/p+1$ of the input array. It does not have the first element, which is stored only in PE $i-1$, and it does not have the last element, which is stored only in PE $i+1$. We are now ready for the SPMD pseudocode. In the first line, each PE i sends its local element $a[1]$ to the PE $i-1$ and receives its local element $a[n/p+1]$ from PE $i+1$; in the second line each PE i sends its local element $a[n/p]$ to PE $i+1$ and receives its local element $a[0]$ from PE $i-1$. The code exploits the convention that communication with nonexistent PEs does nothing (see Sect. 2.4.2):[3]

```
$send(i_{\text{proc}}-1, a[1\quad])$ $\|$ $receive(i_{\text{proc}}+1, a[n/p+1])$        // *
$send(i_{\text{proc}}+1, a[n/p])$ $\|$ $receive(i_{\text{proc}}-1, a[0])$             // **
for $i := 1$ to $n/p$ do $b[i] := (a[i-1]+a[i]+a[i+1])/3$
```

The program takes $3n/p$ arithmetic operations and total communication effort $2(\alpha+\beta)$ (exploiting the fact that our distributed-memory machine can send and receive in parallel; see also Sect. 2.4.2). This is efficient in practice if the local computation is large compared with two startup overheads. Below, we shall see that we can sometimes do better.

For the above pseudocode, it is important that the send and receive operations in each line are actually executed in parallel. Suppose we were to replace the " $\|$ " by a ";" – sequential execution. Then all PEs would first send data to the left. However, no PEs are ready to receive that data. Only PE 0 suceeds, as its send operation to PE "-1" is interpreted as doing nothing. Subsequently, PE 0 is ready to receive from PE 1. Only when this first message transfer is finished will PE 1 be ready to receive from PE 2, and so on. Thus it would take time at least $(p-1)\alpha$ to complete line "*". A similar effect would happen in line "**".

Exercise 3.2. Suppose a parallel send and receive operation is not available. How can you change the above pseudocode fragment so that it executes in time $O(\alpha)$, independent of p? Hint: Distinguish between PEs with odd and even number.

3.1.1 *Reducing Latencies by Tiling

Suppose we want to perform the above averaging operation T times (this is typical of iterative numerical computations). The simple algorithm above would need $2T$ message startups and hence time at least $2T\alpha$. For $n/p \ll \alpha/\beta$, startup overheads would thus dominate the running time. We can overcome this bottleneck by partitioning the

[3] There has been considerable work on building parallel programming languages automating the reasoning in this paragraph. For the example presented here, this is possible. However, no such language has been very successful, since the situation is frequently more complex.

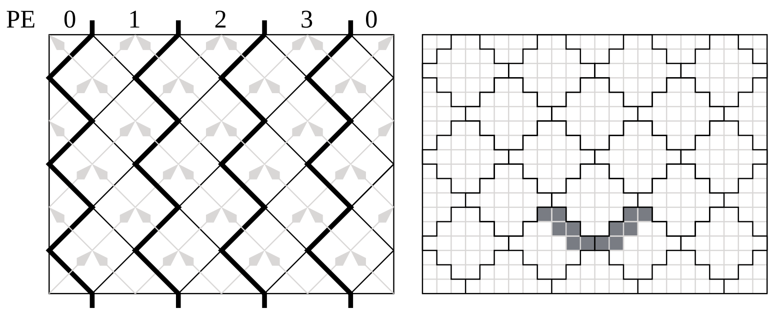


Fig. 3.2. Tiling of a rectangle of computations into diamond-shaped tiles for four PEs. One PE handles the triangular tiles at the left and right borders. *Left*: Continuous drawing for $n \to \infty$. Arrows indicate data dependencies between tiles. Thick lines separate tiles assigned to different PEs. *Right*: The case $n = 24$ leads to tiles of width $w = 24/4 = 6$. The 12 gray cells have to be communicated in order to perform the averaging operations in the tile above them.

averaging operations into more coarse-grained tasks. For this simple, regular computation, a geometrical interpretation is in order. The computations can be arranged as an $n \times T$ matrix, where entry (i,t) stands for computing the tth value of element $a[i]$. This array is partitioned into *tiles* that depend only on the results of a small number of other tiles in such a way that the dependencies form an acyclic graph – a *task DAG*, as described in Sect. 14.6. In the given case, we can minimize the number of startups by using *diamond-shaped* tiles. Figure 3.2 illustrates this situation (at the border of the computation domain, the tiles have a triangular shape); in this figure, time grows from the bottom to the top. The tiles have width $w = n/p$. The computation within a tile depends on $2w$ cells from two tiles below, as indicated in the figure. Since one of these tiles can be assigned to the same PE, only a single message containing the state of w cells must be received by each tile. This happens $2T/w$ times. Overall, we have $2T/w$ message startups and a communication volume of $w \cdot 2T/w = 2T$. Comparing this with the naive algorithm with two startups in each averaging step and two cells received, this reduces the startup overheads by a factor of n/p while keeping the number of exchanged cell states constant.

An alternative way of understanding the scheme is as follows. The area of a tile is essentially w^2. Hence a PE can do work w^2 after receiving a *single* message containing w cells. In the naive scheme, it could do w work w after receiving *two* messages containing two cells. Thus the ratio of work to message has improved by a factor of w.

Exercise 3.3. Explain how to implement diamond tiling using only two local arrays a_0 and a_1 of size $2n/p$ on each PE.

Real applications are more complex than the above simple example. Typically, they iterate on one or several two-dimensional or three-dimensional arrays, and the pattern of cells used for updating an array entry (the *stencil*) and the boundary conditions may vary. Still, the tiling techniques described above can be generalized.

3.2 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. 3.2.1, and singly linked lists in Sect. 3.2.3. 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.

3.2.1 Doubly Linked Lists

Figure 3.3 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 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. 3.4. 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 3.5 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. 3.6. This operation 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

```
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
```

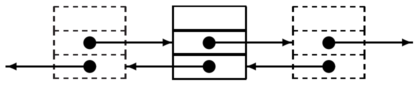


Fig. 3.3. The items of a doubly linked list

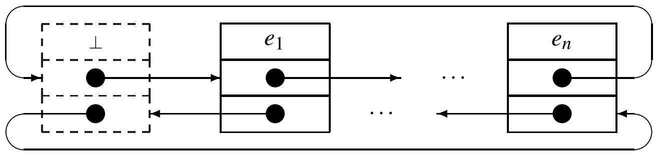
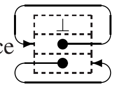


Fig. 3.4. The representation of a sequence $\langle e_1, \ldots, e_n \rangle$ by a doubly linked list. There are $n+1$ items arranged in a ring, a special dummy item h containing no element, and one item for each element of the sequence. The item containing e_i is the successor of the item containing 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.

```
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 : {1,0}; return h.next = this                         // ⟨⟩?
    Function first : Handle; assert ¬isEmpty; return h.next
    Function last : Handle; assert ¬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)
```

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

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 items. 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 3.6 and a short discussion in Sect. 3.9.

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 3.5 and 3.7 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

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

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

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

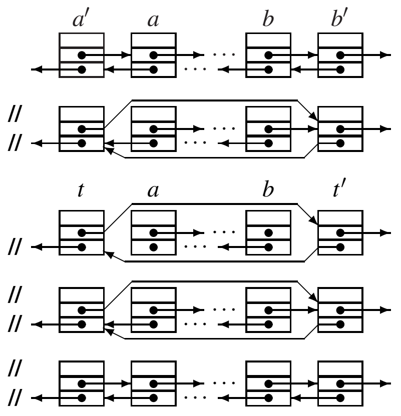


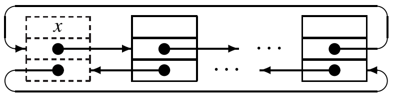
Fig. 3.6. Splicing lists

element are the successor and predecessor, respectively, of h. In order to move an item b to the position 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 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.

Exercise 3.4 (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 item as a *sentinel*. A sentinel is an item 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 item. This ensures that x is present in the list structure and hence a search for it will always terminate. The search will terminate at 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 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 3.5. Implement a procedure *swap* that swaps two sublists in constant time, i.e., sequences $(\langle\ldots,a',a,\ldots,b,b',\ldots\rangle,\langle\ldots,c',c,\ldots,d,d',\ldots\rangle)$ are transformed into $(\langle\ldots,a',c,\ldots,d,b',\ldots\rangle,\langle\ldots,c',a,\ldots,b,d',\ldots\rangle)$. Is *splice* a special case of *swap*?

Exercise 3.6 (memory management). Implement the function *checkFreeList* called by *insertAfter* in Fig. 3.7. Since an individual call of the programming-language primitive **allocate** for every single item might be 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 3.7. Give a constant-time implementation of an algorithm for rotating a list to the right: $\langle a,\ldots,b,c\rangle \mapsto \langle c,a,\ldots,b\rangle$. Generalize your algorithm to rotate $\langle a,\ldots,b,c,\ldots,d\rangle$ to $\langle c,\ldots,d,a,\ldots,b\rangle$ in constant time.

```
// Deleting and inserting elements.
// ⟨…,a,b,c,…⟩ ↦ ⟨…,a,c,…⟩
Procedure remove(b : Handle) moveAfter(b, freeList.head)
Procedure popFront remove(first)
Procedure popBack remove(last)

// ⟨…,a,b,…⟩ ↦ ⟨…,a,e,b,…⟩
Function insertAfter(x : Element; a : Handle) : Handle
    checkFreeList              // make sure freeList is nonempty. See also Exercise 3.6
    a′:=freeList.first                              // Obtain an item a′ to hold x,
    moveAfter(a′,a)                                      // put it at the right place,
    a′→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
// (⟨a,…,b⟩,⟨c,…,d⟩) ↦ (⟨a,…,b,c,…,d⟩,⟨⟩)
Procedure concat(L′ : List)
    splice(L′.first, L′.last, last)

// ⟨a,…,b⟩ ↦ ⟨⟩
Procedure makeEmpty
    freeList.concat(this)                  //
```

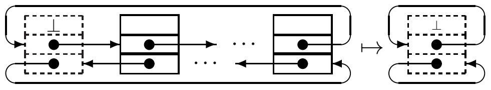


Fig. 3.7. More constant-time operations on doubly linked lists

Exercise 3.8. *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? Why is the relative speed difference dependent on the size of the list?

3.2.2 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 3.9. 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.

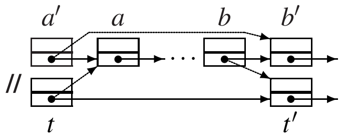
Exercise 3.10. Explain how the operations *remove*, *insertAfter*, and *concat* have to be modified to keep track of the length of a *List*.

3.2.3 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$(a',b,t : SHandle)$
    $\begin{pmatrix} a'\rightarrow next \\ t\rightarrow next \\ b\rightarrow next \end{pmatrix} := \begin{pmatrix} b\rightarrow next \\ a'\rightarrow next \\ t\rightarrow next \end{pmatrix}$
```



Similarly, *findNext* should return not the handle of the next *SItem* containing the search key 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 3.11. 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. 5.3. We may also use singly linked lists to implement free lists of memory managers – even for items in doubly linked lists.

3.3 Processing Linked Lists in Parallel

Linked lists are harder than arrays to process in parallel. In particular, we cannot easily split lists into equal-sized pieces of consecutive elements. The reason is that in arrays, proximity in memory corresponds to proximity in the logical structure. Moreover, arrays support access by index. For linked lists, proximity in memory does not correspond to proximity in the logical structure. The next-pointer of a list item may point to an arbitrary position in memory. Moreover, lists do not support access by index, but only sequential access. The only list element accessible for a given element is the next element, and hence linked lists seem to enfore sequential accessing. As a global rule, it is wise to avoid lists in parallel computing. However, the situation is not completely bleak. We shall see in this section how to convert a linked list into an array. Somewhat surprisingly, this process, which is also known as *list ranking*, is parallelizable. In the algorithms for list ranking, we shall exploit the fact that a parallel algorithm can start traversing the list simultaneously from many positions. The challenge lies in coordinating the different traversals. List ranking plays an important role in many theoretical PRAM algorithms, and parallel list-ranking algorithms are a good showcase of important parallelization ideas such as doubling, contraction, multilevel algorithms, and using inefficient subroutines in an overall efficient algorithm.

3.3.1 *List Ranking by Doubling

We shall work with a singly linked list whose items are stored in an array $L[0..n]$ in any order except that a dummy item is stored in $L[n]$. The dummy item is the last element of the list, and its next-pointer points to itself. We shall compute the distance of each item to the dummy item. Having the distance, reordering is easy. Actually, we shall solve a slightly more general problem. Items have an additional field *rank*. The dummy item has initial rank 0. The initial rank values for the other items are arbitrary; in the list-ranking task, the initial rank of all other items is 1. *The task is to compute for each item the sum of the rank values from the item to the dummy item following the next-pointers*; see Fig. 3.8(a) for an example. *Our algorithm will manipulate the next-pointers and rank values in such a way that this sum remains invariant*. At the end, the *next*-pointer of each item will point to the dummy item directly, and *rank* will contain the desired value.

Exercise 3.12. Give a sequential algorithm that computes the ranks in time $O(n)$.

We begin with an elegant and simple PRAM algorithm that repeatedly replaces the *next*-pointers with the result of following two *next*-pointers and compensates by adding the corresponding *rank* values:

```
Procedure doublingListRanking($L : Array\ [0..n]$ of $Item$)
    for $j := 1$ to $\lceil \log n \rceil$ do
        for $i := 0$ to $n-1$ do||                // Synchronize after each instruction!
            $L[i].rank \mathrel{+}= L[i].next \to rank$
            $L[i].next \ := L[i].next \to next$
```

The algorithm maintains the following invariant: Consider any list item $L[i]$ and consider the sublist of the original list starting at that item and ending just before the item $L[i].next$. Then $L[i].rank$ is the sum of the initial rank fields of the items in this sublist. Moreover, after j iterations, each *next*-pointer either points to the dummy item or jumps 2^j positions in the input sequence to the right. Se Fig. 3.8(a) for an example.

Exercise 3.13. Prove this by induction.

Hence, after $\lceil \log n \rceil$ iterations the *rank* values contain the final result.

The doubling algorithm has span $O(\log n)$ and work $O(n \log n)$. Since a sequential algorithm for list ranking needs only time $O(n)$, the efficiency of the doubling algorithm is only $O(1/\log n)$ even if we simulate several logical PEs on one physical PE. Still, the doubling algorithm is very interesting, since it demonstrates how sequentially following n pointers can be emulated using only $\log n$ iterations of a parallel algorithm. The parallel algorithm traverses the list simultaneously from all locations and uses pointer doubling to halve the distance of each item to the last item in each round. In the next section, we shall see how doubling can be used to accelerate an efficient algorithm.

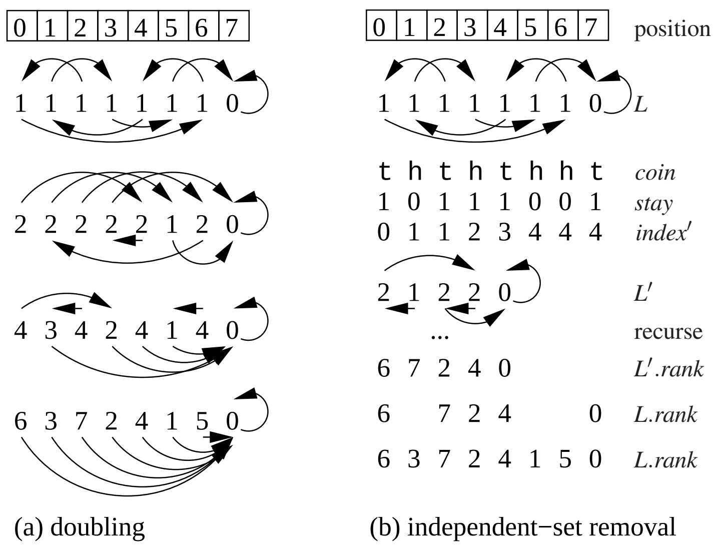


Fig. 3.8. List-ranking algorithms. We have a list of eight items stored in an array $L[0..7]$. The last item is stored in $L[7]$ and points to itself. The next-pointers are shown as arrows. The initial ranks are 1 for all items but the dummy item. The initial rank of the dummy item is 0. The final ranks are shown in the last row. Note that the list element in position $L[6]$ has a rank of 5 because of the linking $L[6] \to L[4] \to L[1] \to L[3] \to L[5] \to L[7]$.

Exercise 3.14. Give a PRAM algorithm with work $O(n)$ and span $O(1)$ that converts an array of n singly-linked-list items into an array of doubly-linked-list items representing the same sequence.

3.3.2 *A Multilevel Algorithm for List Ranking

The pointer-doubling algorithm works on all list items in each round. Since the number of rounds is logarithmic, its work is $\Theta(n \log n)$. If we want an algorithm with linear work, we cannot work on all list items in every iteration of the algorithm. The *independent-set removal* algorithm is based on this idea. It first removes a constant fraction of the list items and builds a list of the remaining items. It then recurses on this *contracted* instance. Finally, it reinserts the removed items. In order to make the first and last steps efficient, an independent set of items is removed in the first step.

The algorithm is a good example of a *multilevel algorithm*: Build a new, smaller instance somehow representing the entire problem, solve the smaller problem recursively, and build the overall solution from the solution of the smaller problem. Multilevel algorithms are related to divide-and-conquer algorithms. One can view multilevel algorithms as divide-and-conquer algorithms which make only a single recursive call (e.g., see the quickselect algorithm in Sect. 5.8). The difference is a matter of interpretation – in a multilevel algorithm, the contracted instance represents the entire original input in some uniform way, whereas a divide-and-conquer algorithms splits off a subproblem by removing irrelevant information.

```
Procedure $isrListRanking(L : Array\ [0..n]$ of $Item, B : \mathbb{N})$
  if $n \le B$ then $doublingListRanking(L)$;  return                              // base case
  $findIndependentSet(L)$                                                           // which items stay?
  $exclusivePrefixSum(L, stay, index')$                                             // enumerate staying elements
  $L' : Array\ [0..L[n].index']$ of $Item$                                          // contracted instance
  for $i := 0$ to $n$ do$\|$                                                        // build $L'$
    if $L[i].stay$ then                                                             // move $L[i]$ to $L'$
      $i' := L[i].index'$                                                           // position in $L'$
      $r := L[i].next$                                                              // right neighbor
      $L'[i'] := L[i]$                                                              // copy
      $L'[i'].next :=$ addressof $L'[$ if $r \to stay$ then $r \to index'$ else $r \to next \to index']$
      if $\neg r \to stay$ then $L'[i'].rank \mathrel{+}= r \to rank$               // establish invariant for $L'$
  $isrListRanking(L', B)$                                                           // recurse
  for $i := 0$ to $n-1$ do$\|$ if $L[i].stay$ then $L[i].rank := L'[L[i].index'].rank$       // assemble
  for $i := 0$ to $n-1$ do$\|$ if $\neg L[i].stay$ then $L[i].rank \mathrel{+}= L[i].next \to rank$   // solution
```

Fig. 3.9. List ranking by independent-set removal (explicitly parallel)

Figure 3.9 gives pseudocode for the algorithm. The items now have additional fields *stay* and $index'$. The Boolean *stay* indicates whether the item stays in the list ($stay = 1$) or is to be removed ($stay = 0$). For the dummy item, we always have $stay = 1$. For an element that stays, $index'$ is the position of the item in the contracted instance. We shall explain in a moment how it is computed. The subroutine *findIndependentSet* is responsible for finding a (large) independent set of list items that can be removed ($stay = 0$). We never remove two adjacent elements. By requiring that the predecessors and successors of removed elements must stay, we make it easy to build the contracted instance and to reconstruct the overall solution. We defer the description of *findIndependentSet* for now. The position of a staying element in L' can be computed by computing a prefix sum; we have $L[j].index' = \sum_{i=1}^{j-1} L[i].stay$. In Sect. 13.3 it is explained how prefix sums can be computed using linear work and logarithmic span.

But how can L' represent the entire input? The idea is to define the input *rank* values of L' in a way such that for all staying elements the output *rank* value is the same as for L. To this end, we first copy rank values from L to L' and then add the rank of the successor item in L whenever the successor item is removed. After solving L' recursively, it is easy to build the overall solutions. The items of L which stayed in L' can simply take their *rank* value from the corresponding item in L'. The others add their input *rank* value to the *rank*value of their successor. Figure 3.8(b) gives an example.

Let us now analyze *isrListRanking* when the independent set encompasses a constant fraction of the elements. Assume αn elements stay, for some constant $\alpha < 1$.

Then we get the following recurrences for the work and span:

$$\mathrm{work}(n) = \begin{cases} \mathrm{O}(B\log B) & \text{if } n \le B, \\ \mathrm{O}(n + \mathrm{work}(\alpha n)) & \text{else,} \end{cases} \qquad \mathrm{span}(n) = \begin{cases} \mathrm{O}(\log B) & \text{if } n \le B, \\ \mathrm{O}(\log n + \mathrm{span}(\alpha n)) & \text{else.} \end{cases}$$

Roughly, the work shrinks geometrically, so that we get linear work overall plus $\mathrm{O}(B\log B)$ for the base of the recursion. The span is basically the number of levels of recursion times $\log n$; the number of levels of recursion is $\left\lceil \log_{1/\alpha} n/B \right\rceil = \mathrm{O}(\log n/B)$. More precisely, we can prove by induction that

$$\mathrm{work}(n) = \mathrm{O}(n + B\log B) \quad \text{and} \quad \mathrm{span}(n) = \mathrm{O}\Big(\log B + \log n \log \frac{n}{B}\Big).$$

How should we choose the tuning parameter B? Choosing B as a constant yields $\mathrm{span}(n) = \Theta(\log^2 n)$. The span gets smaller for larger values of B. However, if we want linear work, $B\log B$ should be $\mathrm{O}(n)$. This suggests that we should use $B = \Theta(n/\log n)$. We obtain

$$\mathrm{work}(n) = \mathrm{O}(n) \quad \text{and} \quad \mathrm{span}(n) = \mathrm{O}(\log n \log\log n). \tag{3.1}$$

Note that we are using an inefficient algorithm – *doublingListRanking* – to speed up an efficient algorithm – *isrListRanking*. The overall algorithm remains efficient, since we use the inefficient algorithm only when the problem size has been reduced sufficiently to make the inefficiency irrelevant. This is a general principle that works for many parallel algorithms. For example, in Sect. 5.2 we shall design fast, inefficient sorting algorithms that can be used to accelerate the sample sort algorithm described in Sect. 5.13 or the parallel selection algorithm in Sect. 5.9.

3.3.3 Computing an Independent Set

There is a very simple randomized algorithm for computing an independent set. For each item I, we throw a coin. If the coin shows a head and the coin for the successor $I.next$ shows a tail, we put I into the independent set. We always throw a tail for the dummy element. This algorithm has linear work and constant span on a PRAM. Note that there is no need to explicitly check the predecessor $I.prev$: Either we throw a tail for $I.prev$ and we are fine, or $I.prev$ sees that we threw a head for I and therefore stays out of the independent set. Figure 3.8(b) gives an example. Note that $L[3]$ does not enter the independent set even though we threw a head (`h`) for it, because we also threw a head for its successor $L[5]$.

The probability that I goes to the independent set is $\frac{1}{2} \cdot (1 - \frac{1}{2}) = \frac{1}{4}$ except for the predecessor of the dummy element, for which the probability is $\frac{1}{2}$.

Exercise 3.15. Show that using a biased coin does not help: The above probability is never more than $\frac{1}{4}$.

Lemma 3.1. *The expected size of the independent set is* $(n+1)/4$.

Proof. We define the indicator random variable X_i, $i \in 0..n-1$, to be 1 if and only if $L[i]$ goes to the independent set. The size of the independent set is $X := \sum_{i=0}^{n-1} X_i$. We have $\mathrm{prob}(X_i = 1) = \frac{1}{4}$ except for the predecessor of the dummy element, where $\mathrm{prob}(X_i = 1) = \frac{1}{2}$. Using the linearity of expectations we get

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=0}^{n-1} X_i\right] = \sum_{i=0}^{n-1} \mathrm{E}[X_i] = (n-1)\frac{1}{4} + \frac{1}{2} = \frac{n+1}{4}. \qquad \square$$

Unfortunately, the analysis of *isrListRanking* assumes a deterministic algorithm for computing independent sets. There are various ways to fix this problem – none of them very elegant. One view is to ignore technicalities and simply hope that (3.1) is a good approximation of the expected work and span of the algorithm. We shall outline several ways to substantiate this hope. One way is to convert our Monte Carlo algorithm for finding an independent set into a Las Vegas algorithm by actually computing the size of the independent set and repeating until it is "large enough". If we choose the acceptance threshold right, we can prove that a constant number of iterations is enough. Since counting needs logarithmic span, we do not have constant span anymore. But this is no problem in the analysis of *isrListRanking*. However, it is bad style to make an algorithm more expensive just because one is too lazy to perform a tight analysis. We can use a similar argument to that in the analysis of quickselect in Sect. 5.8 to also allow for recursion levels that do not shrink enough. The only argument needed is a constant probability bound such as the following one.

***Exercise 3.16.** Use Markov's inequality (A.5) to prove that with probability at most $\frac{4}{5}$, more than $\frac{15}{16}n$ items stay.

Unfortunately, the constant factor we get out of such an analysis is ridiculously pessimistic. We can do better by proving that the size of the independent set will be very close to its expectation with high probability.

****Exercise 3.17.** Use the bounded difference inequality given in [209] to show that with probability $1 - \mathrm{O}(1/n)$, the independent set will have size $n/4 - o(n)$. Hint: First show that the size of the independent set changes by at most two when we change the outcome of a single coin throw.

***Exercise 3.18.** A *maximal independent set* is one that cannot be enlarged by including additional elements.

(a) Show that a maximal independent set of a list contains at least $n/3$ elements.
(b) Design a randomized parallel algorithm that computes a maximal independent set. Hint: Repeatedly throw coins. Fix the result for the staying items and their successors. Can you make this algorithm work efficiently?

3.4 Unbounded Arrays

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

$$\begin{aligned}\langle e_0,\ldots,e_{n-1}\rangle.\mathit{pushBack}(e) &= \langle e_0,\ldots,e_{n-1},e\rangle,\\ \langle e_0,\ldots,e_{n-1}\rangle.\mathit{popBack} &= \langle e_0,\ldots,e_{n-2}\rangle \qquad \text{(for } n \geq 1\text{)},\\ \mathit{size}(\langle e_0,\ldots,e_{n-1}\rangle) &= n.\end{aligned}$$

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. The solution with unbounded arrays is clearly more elegant, in particular if you can use an implementation provided by a library. Also, there are situations where the input can be read only once.

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 3.10 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 will become clear later.

3.4.1 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 the operator $[\cdot]$ 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 $\mathrm{O}(m)$.

```
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 → |e_0|···|e_{n-1}|   ···   |

  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 → |0|1|2|3|
      reallocate($\beta n$)                              // b → |0|1|2|3| | | | |
    $b[n] := e$                                          // b → |0|1|2|3|e| | | |
    $n$++                                                // b → |0|1|2|3|e| | | |

  Procedure popBack                                        // Example for $n = 5$, $w = 16$:
    assert $n > 0$                    // b → |0|1|2|3|4| | | | | | | | | | | |
    $n$−−                             // b → |0|1|2|3|4| | | | | | | | | | | |
    if $\alpha n \le w \land n > 0$ then                            // reduce waste of space
      reallocate($\beta n$)                              // b → |0|1|2|3| | | | |

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

Fig. 3.10. Pseudocode for unbounded arrays

Lemma 3.2. *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* $O(m)$.

Lemma 3.2 is a nontrivial statement. A small and innocent-looking change to the program invalidates it.

Exercise 3.19. 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 3.2 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 $O(m)$. 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 3.2 more elegantly. The increased elegance also allows better comparisons between data structures.

Corollary 3.3. *Unbounded arrays implement the operation $[\cdot]$ in worst-case constant time and the operations pushBack and popBack in amortized constant time.*

To prove Lemma 3.2, 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. 3.10. 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 guarantee that the balance of the account stays nonnegative and hence suffice to cover the withdrawals made by *reallocate*.

The first call of *reallocate* occurs when there is already one element 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 $4n \leq w$. 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 3.2. $\square$

Exercise 3.20. 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 learn alternative proof methods[4] for amortized analysis. 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 for *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 3.21 (continuation of Exercise 3.20). 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 such that $\beta/(\alpha-\beta) \leq \beta/(\beta-1)$.

3.4.2 Amortized Analysis of Unbounded Arrays: The Local Argument

We now describe our second modification of the proof. 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 executed. Immediately after the first *reallocate*, there is one token in the account and the invariant requires none ($n = w/2 = 1$). A *pushBack* (ignoring the potential call of *reallocate*) increases n by one and adds two tokens. So the invariant is maintained. A *popBack* (again ignoring the potential call of *reallocate*) removes one element and adds one token. So the invariant is again maintained. We next turn to calls of *reallocate*. 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.

[4] Some induction proofs become easier if they are formulated in terms of a smallest counterexample. It is useful to know both methods. The situation is similar here.

Exercise 3.22. Charge three tokens for a *pushBack* and no tokens for a *popBack*. Argue that the account always contains at least $n + \max(2(n - w/2), w/2 - n) = \max(3n - w, w/2)$ tokens.

Exercise 3.23 (popping many elements). Implement an operation *popBack*(k) that removes the last k elements in amortized constant time. Of course, $0 < k \leq n$, but k is arbitrary otherwise.

Exercise 3.24 (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: In an initial solution, support only [.] and *pushBack*. Store the elements in up to two arrays. Start moving elements to a larger array well before the small array is completely exhausted. How do you generalize this approach if *popBack* must also be supported?

Exercise 3.25 (implicitly growing arrays). Implement unbounded arrays where the operation $[\cdot]$ accepts any positive index i as its argument. 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 3.26 (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.

3.4.3 Amortized Analysis of Binary Counters

Amortized analysis is so important that it deserves a second introductory example. We consider 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 0. Its representation is a string of 0's. We define the cost of incrementing the counter as 1 plus the number of trailing 1's 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, its final value is m. The representation of the number m requires $L = 1 + \lceil \log m \rceil$ bits. Among the numbers from 0 to $m-1$, there are at most 2^{L-k-1} numbers whose binary representation ends with a 0 followed by k many 1's. For each one of them, an increment costs $1 + k$. Thus the total cost of the m increments is bounded by

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

where the last equality uses (A.15). Hence, the amortized cost of an increment is $O(1)$.

The argument above is global, in the sense that it requires an estimate of the number of representations ending in a 0 followed by k many 1's. 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 1's in the binary representation of the counter. So, the balance is initially 0. Consider an increment of cost $k+1$. Before the increment, the representation ends in a zero followed by k many 1's, and after the increment, the representation ends in a 1 followed by k many 0's. So, the number of 1's 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 $k-1$ with the tokens released from the account, and charge a cost of two to the operation. Thus the total cost of m operations is at most $2m$.

3.5 *Amortized Analysis

We give here a general definition of amortized time bounds and amortized analysis. We recommend that you 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 1 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 $\langle Op_1, Op_2, Op_3, \ldots, Op_m\rangle$. Starting at the initial state s_0, F takes us through a sequence of states to a final state s_m:

$$s_0 \xrightarrow{Op_1} s_1 \xrightarrow{Op_2} s_2 \xrightarrow{Op_3} \cdots \xrightarrow{Op_m} s_m.$$

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

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

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,

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

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.

This definition is a very general formulation of the bank account method. We start with a balance of c tokens and then execute the sequence of operations. If an operation X is executed in state s, we deposit $A_X(s)$ tokens into the account and also withdraw $T_X(s)$ tokens to pay for the execution of the operation. The functions A_X form a family of amortized time bounds if the balance of the account can never become negative. In order to use the bank account method, one has to define the functions A_X and the constant c and then *prove* that the balance can never become negative. The balance after the execution of a sequence F of operations is $c+A(F)-T(F)$.

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 $A_X(s)$ (with small function values) that form a family of amortized time bounds. In our example, the functions $A_{pushBack}(s)=2$, $A_{popBack}(s)=1$, $A_{[\cdot]}(s)=1$, and $A_{reallocate}(s)=0$ for all s form a family of amortized time bounds (with $c=0$). In order to prove that a set of functions is indeed a family of amortized time bounds, one uses induction with a suitable invariant which bounds from below the balance of the account after the execution of a sequence F of operations. For our example, the invariant states that after the execution of a sequence F of operations leading to the state (n,w), the balance is at least $\max(2(n-w/2), w/2-n)$.

Some readers may find it counterintuitive that the amortized cost of *reallocate* is stated as 0. After a call of *reallocate* we have an array of size w, in which exactly half of the slots are occupied. The other half is free, i.e., $n=w/2$. According to the invariant, the balance of the account may be as low as 0 after the operation. The cost of the operation is $w/2$, as this is the number of elements that have to be moved. Before the call, we had either an array of $w/2$ slots, all of which were full ($n_{\text{before}} = w_{\text{before}} = w/2$), or an array of $2w$ slots, a quarter of which were full ($n_{\text{before}} = w/2$ and $w_{\text{before}} = 2w$). In the former case, the balance before the operation is at least $2(n_{\text{before}} - w_{\text{before}}/2) = w/2$. In the latter case, the balance before the operation is at least $w_{\text{before}}/2 - n_{\text{before}} = w - w/2 = w/2$. Thus, in either case, the cost of the operation is $w/2$ and the balance of the account is at least $w/2$. We can therefore completely pay for the cost of the operation out of the account and there is no need to charge any amortized cost.

3.5.1 The Potential Method for Amortized Analysis

Here, we introduce a powerful general technique for obtaining amortized time bounds: the potential method for amortized analysis. In Sect. 3.4.3, we analyzed the binary counter by associating with each bit string (state of the data structure) the number of 1-bits in the bit string (the potential of the state) and then using this potential to compute the charges required for the counter operations. We now formalize and generalize this method. The essence of the method is a function *pot* that associates a nonnegative potential with every state of the data structure, i.e., $pot\colon S \longrightarrow \mathbb{R}_{\geq 0}$. We call $pot(s)$ the potential of 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 3.4 (potential method). *Let S be the set of states of a data structure, let s_0 be the initial state, and let $pot\colon S \longrightarrow \mathbb{R}_{\geq 0}$ 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 with $c = pot(s_0)$.

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

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

since $pot(s_m) \geq 0$. Thus $T(F) \leq A(F) + pot(s_0)$ and the definition of amortized time bounds is satisfied with $c = pot(s_0)$. Note that c is a constant independent of F. □

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 3.22, 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 of $\max(3-2,1) = 1$. 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 of 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 of $w/4$. 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 $O(1)$ and the amortized cost of *reallocate* is 0 or less. Thus a sequence of m operations on an unbounded array has cost $O(m)$.

Exercise 3.27 (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. 3.4.

(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 1's 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 0?

3.5.2 Universality of the Potential Method

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

Theorem 3.5. *Let $B_X(s)$ be a family of amortized time bounds. Then there is 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 3.4.*

Proof. For a sequence $F = \langle Op_1, \ldots, Op_m \rangle$ of operations which generates the sequence $s_0, s_1, \ldots, s_m$ from the start state s_0, define $B(F) = \sum_{1 \leq i \leq m} B_{Op_i}(s_{i-1})$. Let c be a constant such that $T(F) \leq B(F) + c$ for any such sequence F.

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 here, 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 $A_X(s)$ defined according to Theorem 3.4 form a family of amortized time bounds.

We need to show that $A_X(s) \leq B_X(s)$ for all X and s. Let $\varepsilon > 0$ be arbitrary. We shall show that $A_X(s) \leq B_X(s) + \varepsilon$. Since ε is arbitrary, this proves that $A_X(s) \leq B_X(s)$.

Fix $\varepsilon > 0$, let s be an arbitrary state, and let X be an operation. Let F be a sequence with final state s and $B(F) + c - T(F) \leq pot(s) + \varepsilon$. The operation X transforms s into some state s'. Let F' be F followed by X, i.e.,

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

Then $pot(s') \leq B(F') + c - T(F')$ by the definition of $pot(s')$, $pot(s) \geq B(F) + c - T(F) - \varepsilon$ 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

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

□

3.5.3 Amortization in Parallel Processing

Amortized solutions are also useful in parallel processing, but there is a potential pitfall that forces us to use them with more care. This is because local amortization may have global consequences. For example, assume that each PE performs a *pushBack* operation on a local unbounded array and that the PEs have to synchronize after the *pushBack*. Then, if one of the PEs has to copy its array, all other PEs have to wait for it, and the amortized analysis breaks down.

Exercise 3.28. Show that the following SPMD pseudocode takes time $\Omega(pn)$:

```
$a$ : $UArray$ **of** $\mathbb{N}$                          // one on each PE!
**for** $i := 1$ **to** $n$ **do**                     // synchronize after each iteration
    **if** $i > i_{\mathrm{proc}}$ **then** $a.pushBack(i)$
```

Hint: Investigate when different PEs perform expensive *pushBack*s.

In order to avoid this pitfall, either we should avoid synchronization to such an extent that delays due to expensive events provably cannot lead to excessive waiting times, or we should synchronize the PEs such that all expensive events happen on all PEs at the same time. When amortization is used globally in the first place, we often want a global trigger for rare expensive events and we then synchronize all PEs to work collectively on them. Examples are epoch FIFO queues, described in Sect. 3.7, where we use a probabilistic asynchronous trigger in order to avoid contention due to the triggering mechanism, and the unbounded distributed-memory hash table described in Sect. 4.6.1 where we use explicit synchronization and collective communication in order to find the exact global size of the data structure.

3.6 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 their 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*, *last*, and *isEmpty*. We shall use the simplified names *push*, *pop*, and *top* for the three main stack operations.

The behavior is different when people are standing 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*, *popFront*, and *isEmpty*.

The more general *deque* (pronounced "deck"), or *double-ended queue*, allows the operations *first*, *last*, *pushFront*, *pushBack*, *popFront*, *popBack*, and *isEmpty*; it 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 3.11 illustrates the access patterns of stacks, queues, and deques.

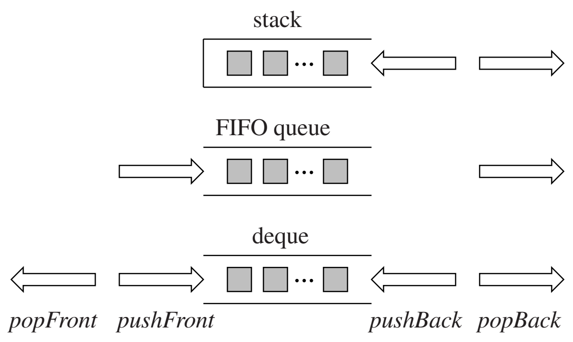


Fig. 3.11. Operations on stacks, queues, and double-ended queues (deques).

Exercise 3.29 (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* [153].[5]

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 transforms the state $(\langle k,\ldots,1\rangle,\langle\rangle,\langle\rangle)$ into the state $(\langle\rangle,\langle\rangle,\langle k,\ldots,1\rangle)$.

Exercise 3.30. Explain how to implement a FIFO queue using two stacks so that each FIFO operation takes amortized constant time.

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

```
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 : {1,0}; return h = t

    Function first : Element; assert ¬isEmpty; return b[h]

    Function size : 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 ¬isEmpty; h := (h + 1) mod (n + 1)
```

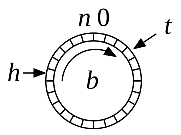


Fig. 3.12. An array-based bounded FIFO queue implementation

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 of stacks and queues translate into good reuse of cache blocks when stacks and queues are represented by arrays.

Bounded stacks, where we know the maximum 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. 3.12. We view an array as a cyclic structure where entry 0 follows the last entry. In other words, we have array indices 0 to n, and view the indices modulo $n+1$. We 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 arithmetic modulo the array size.[6] We always leave at least one

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

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** $b[i+h \bmod (n+1)]$

Bounded queues and deques can be made unbounded using techniques similar to those used for unbounded arrays in Sect. 3.4.

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 3.31 (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 3.32 (external-memory stacks and queues). Design a stack data structure that needs $\mathrm{O}(1/B)$ I/Os per operation in the I/O model described in Sect. 2.2. It 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.

3.7 Parallel Queue-Like Data Structures

All operations on stacks, queues, and deques concentrate on one or two logical positions. These positions constitute potential bottlenecks and thus can be problematic for parallel processing. Hence, queue-like data structures should only be used with great care. However, there are situations where we need them and where they can even simplify parallelization. We shall discuss such situations in this section.

The straightforward implementation of queue-like data structures on a shared-memory machine protects the data structure with a lock, which has to be acquired before performing any operation. The lock serializes all operations on the queue. Although it seems hard to avoid serialization in the worst case, for example for a queue that alternates between being empty and being nonempty, it is undesirable for parallel computing. Therefore there has been considerable work on better implementations [150] for special situations. What exactly can be done depends not only on the structure (stack, queue, deque) but also on which PEs are allowed to perform which operations. We shall now concentrate on FIFO queues as a concrete example. We will therefore use the abbreviations *push* for *pushFront* and *pop* for *popBack* here. Stacks are much less important for parallelization. At the end of this section, we shall discuss an important special case of a deque.

3.7.1 Single-Producer/Single-Consumer FIFO Queues

Let us call PEs allowed to push elements *producers* and PEs allowed to pop elements *consumers*. A very simple case of a shared-memory FIFO queue is a *single-producer/single-consumer FIFO queue* (1/1-FIFO queue). This case is important since 1/1-FIFOs queues can be used to decouple PEs dedicated to different functions. This way, we can build a *pipeline* that processes a stream of tasks in several stages, with one PE dedicated to each stage. For example, in the UNIX shell, the operator "`|`" *pipes* data from one command to the next. In 1/1-FIFO queues, locks can be completely avoided. We explain the FastForward queue [124], which is a variant of the bounded circular array FIFO queue shown in Fig. 3.12. *We assume that full and empty queue entries can be distinguished.* As a consequence, there is no need to access h (head) and t (tail) to find out whether the queue is full or empty. The code simplifies and the performance increases. Also, there is no need anymore for an extra location to distinguish an empty from a full queue. However, we now need to explicitly return error conditions, since there is no safe way to find out beforehand whether an operation will be successful. We describe the operations.

```
Function push($x$ : Element) : $\{\texttt{OK}, \texttt{FULL}\}$
    if $\neg b[t].isEmpty$ then return FULL
    $b[t] := x$;    $t := t + 1 \bmod n$;    return OK
```

The function *pop* looks very similar except that it has to explicitly empty the entry just popped:

```
Function pop($x$ : Element) : $\{\texttt{OK}, \texttt{EMPTY}\}$
    if $b[h].isEmpty$ then return EMPTY
    $x := b[h]$;    $b[h] := \texttt{EMPTY}$;    $h := h + 1 \bmod n$;    return OK
```

3.7.2 Relaxed FIFO Queues and Bulk Operations

More general FIFO queues allowing fully concurrent access are much more complicated, and it is astonishing how much can go wrong (see [150]). The trouble comes from making sure that the distributed execution is equivalent to a serial execution in which each operation is atomically executed at some point in time between the start and the end of the distributed execution. This behavior is not guaranteed by currently available memory consistency models (see also Sect. B.3). Moreover, even achieving a global ordering does not mean that the order in which elements are processed corresponds to the actual time when the operations are called – this would be almost impossible to ensure, for fundamental physical reasons. We view these difficulties as an indication that the semantics of a FIFO queue is unnecessarily strict. If the strict FIFO property is not required, significant simplications are possible. The FIFO semantics is violated when an element is pushed before another element but is popped after it. Then the second element *overtakes* the first. Figure 3.13 shows an example. We can quantify the degree of violation of the FIFO property by the number of elements that can overtake another element or how long the time lapses can be.

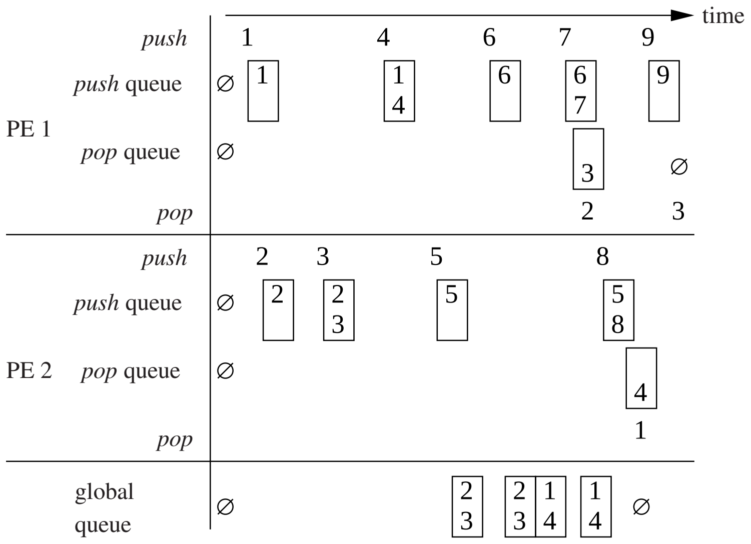


Fig. 3.13. Nine *push* operations (elements 1..9) and three *pop* operations on a batched FIFO queue with $B = 2$ and two PEs. First PE 1 pushes 1, then PE 2 pushes 2 and 3, then PE 1 pushes 4, and PE 2 pushes 5. The elements 2 and 3 fill a block. This block is moved to the global queue when PE 2 pushes 5. Next PE 1 pushes 6, and the block containing 1 and 4 is moved to the global queue. The global queue now contains two blocks. When a *pop* is executed and the PE has no block in its pop buffer, it fetches a buffer from the global queue. So when PE 1 performs a *pop*, it fetches the block containing 2 and 3, and when PE 2 performs a *pop*, it fetches the block containing 1 and 4. PE 1 then serves the *pop* operations from the block in its *pop* buffer.

Once we accept a relaxed semantics, we can use it to reduce contention. For example, we can process elements in batches, performing expensive access to shared variables only occasionally for an entire batch of elements. Assume each PE maintains separate local push and pop buffers taking up to B elements. When a push buffer is full, it is moved to a global FIFO queue F that takes batches of B elements. Similarly, rather than popping individual elements, a PE pops an entire batch of size B from F into its *pop* buffer and uses this to answer up to B single-element pops. The advantage of this approach is that the cost and contention of accessing the global queue are amortized over B elements. Assuming that elements are produced and consumed continuously, incurring only constant delays between operations, our simple batched FIFO queue with $B = \Theta(p)$ allows at most $O(p^2)$ elements to overtake another one. Figure 3.13 gives an example. Note that the first element pushed (1 on PE 2) is popped after element 2. If the above assumptions are not fulfilled, things can be much worse. For example, a PE might push an element x into an empty buffer, and then will be delayed for a long time while all other PEs rapidly *push* and *pop* an unbounded number of elements that all overtake x.

3.7.3 Distributed-Memory FIFO Queues

On a distributed-memory machine, the message startup overhead is so big that even for a 1/1-FIFO queue, bulk operations make sense. Indeed, 1/1-FIFO queues with bulk operations have a semantics very similar to point-to-point message passing. Most message-passing systems ensure that messages exchanged between a fixed pair of PEs are exchanged in FIFO order – there is no overtaking. A more general FIFO queue allowing arbitrary producers and consumers could be managed by a designated PE f, which receives all *push* and *pop* requests. The order in which these requests are received represents a natural global ordering. If the queue becomes large or if large batches of elements are involved, PE f can delegate the communication and storage of the actual queue content to further PEs. For example, it could just maintain a global counter c, which assigns the cth enqueued batch to PE $b := c \bmod p$. A *push* by a PE a would then involve three messages: a request from a to f, a reply from f to a telling it the value of counter c, and a message from a to b delivering the actual data. PE b would then maintain a local queue for the batches delivered to it. In order to ensure the global ordering determined by PE f, PE b has to reorder incoming messages according to their c-values. Since b can receive batches from any other PE, there is no guarantee that the batches will arrive in the order in which they were sent. Hence the necessity for reordering.

3.7.4 *The Epoch FIFO Queue

We discuss here a scalable, fully distributed relaxed FIFO queue called the epoch FIFO queue. We achieve scalability by using efficient distributed mechanisms, such as prefix sums (Sect. 13.3) and work stealing (Sect. 14.5), for global control. Pushs are done in *epochs* and the guarantee is that all elements pushed in one epoch are popped before all elements pushed in later epochs. Each PE maintains a local push buffer taking all of the elements it pushes during the current epoch. When an epoch ends, all elements pushed during this epoch are moved to a single global array, which is pushed as one big batch into a global queue Q. We discuss below how the length of an epoch is determined. Long epochs reduce the amount of communication required, and short epochs guarantee a smaller violation of the FIFO property. Note that the length of an epoch bounds the number of elements that can overtake any element.

Exercise 3.33. Explain how the elements can be moved efficiently using prefix sums.

All elements from a batch are popped before switching to the next batch. Assigning the elements from a particular batch to the PEs is a special case of the loop-scheduling problem discussed in Sect. 14.5, in which a subinterval of the current batch is assigned to each PE. When this local interval is exhausted, it can steal pieces of intervals from other PEs. It can be shown that this can be done in expected time $\mathrm{O}(\log p + B/p)$ for a batch of size B using randomized work stealing. This is work-efficient if we ensure that the average batch size is $\Omega(p \log p)$. Since shorter batches guarantee a smaller violation of the FIFO property, our goal is to have batch sizes $\Theta(p \log p)$.

The easiest way to control the batch size is to maintain a global counter c which counts the number of push operations performed in the current epoch. Keeping an exact count is a source of contention. Fortunately, it suffices to keep an approximate count. To avoid contention, c is only incremented with probability $\Theta(1/p)$ when a push is performed. When c exceeds a limit $\ell = \Omega(\log p)$, the next epoch is triggered. In this way, an epoch contains $\Theta(p\ell) = \Theta(p \log p)$ pushes with high probability. Since the work per epoch is $O(p \log p)$, the amortized cost per operation is constant.

The epoch length bounds the number of elements that can overtake any element. Thus only $O(p \log p)$ elements can overtake any element – much less than the $\Theta(p^2)$ overtaking elements for simple batched FIFO queues, and independent of additional assumptions about delays between operations. The disadvantage of epoch FIFO queues is that they require global synchronization of all PEs in order to collectively perform the operations needed for switching from one epoch to the next one.[7]

Epoch FIFO queues can also be implemented efficiently on distributed memory. The global counter c is maintained by PE 0, to which all increment requests are sent. Since PE 0 receives only a logarithmic number of requests per epoch, it does not constitute a bottleneck. PE 0 notifies the other PEs about the end of an epoch using an asynchronous broadcast (see Sect. 13.1). To avoid moving elements around, each entry of the global queue Q is split into p pieces – one for each PE holding the elements pushed by that PE. These elements also constitute the initial interval assigned to the PE. Possible imbalances will then be equalized during work stealing.

3.7.5 Deques for Work Stealing

A variant of deques which is important for the work-stealing load balancers described in Sect. 14.5 has a single PE that uses the deque like a stack (*pushBack*, *popBack*), and any number of PEs that are allowed to do *popFront*s. A lock-free implementation is given in [20] that exploits the special set of operations and additional properties of the application.

3.8 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.

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.

[7] In a more sophisticated variant of the epoch FIFO queue, this work could also be done in the background without working threads noticing, by using $\Theta(p)$ additional server threads which are triggered by the end of an epoch or the exhaustion of the elements in the batch.

Table 3.1. Running times of operations on sequences with n elements. The entries have an implicit $O(\cdot)$ 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

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.

3.9 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 list-like data structures yourself, for example when your objects are members of several linked lists. In such implementations, memory management is often a major challenge.

3.9.1 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 of 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 may 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 [194] 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 list-like 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 the operator $[\cdot]$. 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.

3.9.2 Java

Since version 6 of Java, the *util* package 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 book on Java proudly announce that Java has no pointers, so 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 collection of all objects that are not referenced anymore.

3.10 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 Brown, Huddlestone, Mehlhorn, Sleator, and Tarjan [57, 158, 299, 300]. The overview article [308] popularized the term *amortized analysis*, and Theorem 3.5 first appeared in [213].

There is an array-like 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. 3.6. Element i can be found in entry $i \bmod n'$ 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 [177] for more sophisticated implementations of deques and an implementation study.