2
Linear Lists

2.1 Managing Data Storage

In our first steps as programmers, we often write short programs with a quite limited number of variables. Very soon, however, the programming tasks get more involved, and we need some advanced tools to manage efficiently the increasing number of elements dealt with by our programs. This is the role of what has become known as *Data Structures*, which are the subject of this book.

When coming to organize the data we are supposed to handle, we shall deal with entities called *records*, like the one in Figure 2.1, representing, in this chapter's running example, the information about a given student at some university. A record may be divided into *fields*, each standing for a data item relating to the given student, like the name, given name, address, field of study, a list of courses and their grades, etc. One of the fields is particular in that it will serve to identify the records and to distinguish between them. It is emphasized in Figure 2.1, its content is called here ID number, and the assumption will be that different records have different IDs.

Our task is to maintain a large collection of such records, allowing efficient access and updates. Maintaining means here to store the records in such a way that queries about any of their data fields may be processed quickly, but also to facilitate subsequent changes, like insertions of new records, or deletion of those that are not needed any more. Getting back to our example, a student might ask what grade she got on a specific course, a new record has to be allocated if a new student enrolls, and the records of students who finished their studies should be erased from the current list and possibly transferred to another collection, dealing with alumni.

The technical problem of processing records with variable length fields may be overcome by storing the records wherever convenient, and keeping only a list of fixed length (ID, pointer) pairs. We may then restrict our attention to

Name	Given name	ID number	Address	Study field	course 1	grade 1	course2	grade2	...

Figure 2.1. A sample record and its subfields.

processing only the list of IDs, and shall henceforth narrow our discussion to the handling of the identifying field alone.

An immediate question to be dealt with is how to store such a list. A straightforward solution would be to store the records sequentially, in the order they appear in the system. More precisely, we might allocate a certain amount of sequential memory cells, and store there new records on demand, one after the other. This is called *sequential allocation*, and it is clearly advantageous for allowing inserts, as long as the allocated space has not been exhausted, but searches could force a scan of the entire list. To avoid such worst case behavior, one could require the list to be kept in order, for example sorted by increasing ID. This would then enable an improved lookup procedure, known as *binary search*.

Background Concept: Binary Search

Binary search belongs to a family of algorithms called *Divide and Conquer*. The common feature of these algorithms is that they solve a problem by dividing it into several similar, but smaller, subproblems, which can be solved recursively. For instance, when looking for an element x within a sorted array $A[1], \dots, A[n]$, we start by comparing x to the middle element $A[n/2]$. If it is found there, we are done, but if not, we know whether x is smaller or larger, and may restrict any further comparisons to the lower or upper part of the array A accordingly. Continuing recursively, the next comparison will be at the middle element of the subarray we deal with, i.e., either $A[n/4]$ if the array is $A[1], \dots, A[n/2]$, or $A[3n/4]$ if the array is $A[n/2], \dots, A[n]$. The search procedure stops either when x is found, or when the size of the array is reduced to 1 and one can say with certainty that x is not in A.

If $T(n)$ denotes the number of comparisons (in the worst case) to find an element x in an array of size n, we get that $T(1) = 1$, and for $n > 1$:

$$T(n) = 1 + T(n/2). \qquad (2.1)$$

Applying equality (2.1) repeatedly yields

$$T(n) = i + T(n/2^i), \qquad \text{for} \quad i = 1, 2, \dots . \qquad (2.2)$$

To get rid of the T on the right-hand side of the equation, let us choose i large enough to get to the boundary condition, that is $n/2^i = 1$, so that $n = 2^i$, hence $i = \log_2 n$. Substituting in (2.2), we get

$$T(n) = 1 + \log_2 n.$$

We conclude that having sorted the list of records allows us to reduce the number of needed comparisons in the worst case from n to about $\log_2 n$, indeed a major improvement.

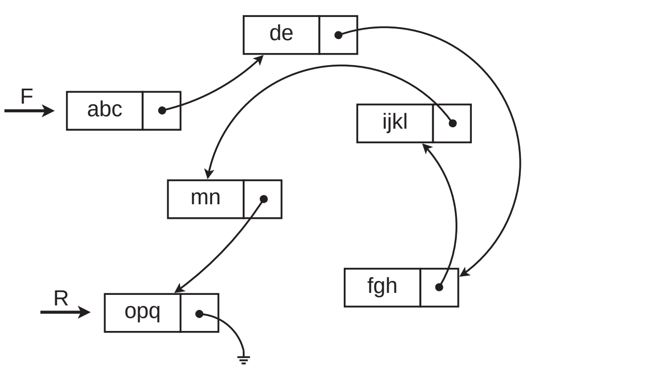
Another problem, however, has now been created: updating the list is not as simple as it used to be. If a new record has to be added, it cannot be simply adjoined at the end, as this might violate the sorted order. Inserting or deleting records may thus incur a cost of shifting a part, possibly as many as half, of the records. Overall, the reduction of the search complexity from n to about $\log n$ came at the cost of increasing the number of steps for an update from 1 to about n.

This problem may be solved by passing to a scheme known as *linked allocation*, in which the records are not necessarily stored in contiguous locations, but are rather linked together, in some order, by means of pointers. The insertion or deletion of a record in a list can then be performed by a simple update of a small number of pointers, without having to move any of the other records. Regretfully, while there is an improvement for the update operations, it is again the search time that is hurt: since we do not know where to find the middle element in the list for the linked model, the use of binary search is not possible, and we are back to a linear search with about n comparisons.

In fact, to get efficient performances for both searches and updates, more sophisticated structures than the linear ones we consider here are needed, and we shall get back to this problem in Chapter 4, dealing with *Trees*. Note also that linked lists are not necessarily built by means of pointers, which are defined in many programming languages. The left part of Figure 2.2 is a schematic of a linked list consisting of 6 elements, and the right part of the figure is an equivalent list, for which the pointers have been replaced by indices in a table. As can be seen, the pointers or indices are incorporated as a part of the records. The index -1 is an indicator of the end of the list, and is equivalent to the ground sign, representing the NIL pointer.

2.2 Queues

Probably the most natural way of organizing a set of records into a structured form is what we commonly call a *queue*. When approaching a bus stop, a bank



1		
2	fgh	7
R 3	opq	-1
F 4	abc	8
5		
6		
7	ijkl	10
8	de	2
9		
10	mn	3

Figure 2.2. Example of a list in linked allocation.

counter or a supermarket cash register, one may assume that the requested services will be provided on a first come, first serve basis, and if one has to wait, a queue, also known as FIFO (First In, First Out), will be formed.

Two pointers are kept: one, R, to the Rear of the queue, where new incoming records should be adjoined, the other, F, to the Front of the queue, indicating the next element to be extracted, that is, the next to be "served." Referring again to Figure 2.2 and considering the list as a queue Q, if an element has to be extracted from Q, it will be the one with value abc, denoted by e_{abc}. In the updated queue, F will then point to e_{de}, which is the successor of e_{abc}. Suppose then that we wish to insert a new element, with value rstu. This should happen at the rear of the queue, so a new element e_{rstu} is created, its successor is defined as NIL, the current rear element, e_{opq} will point to it, and R will also point to this new element. Figure 2.3 gives the formal definitions of (a) inserting an element with value x into a queue Q (at its rear), which shall be denoted by $Q \Longleftarrow x$, and (b) extracting (the front) element of Q and storing its value in a variable x, denoted by $x \Longleftarrow Q$. These operations are often called *enqueue* and *dequeue*, respectively.

A queue can also be kept within an array in sequential allocation. This may save the space for the pointers and simplify the update procedures, but one needs then a good estimate for the maximal number of elements that will be stored. If no reliable prediction of this size is possible, there is a risk of *overflow*.

```
$\underline{Q \Longleftarrow x}$
    allocate$(node)$
    value$(node) \leftarrow x$;        succ$(node) \leftarrow$ NIL
    succ(R) $\leftarrow node$;         R $\leftarrow node$

$\underline{x \Longleftarrow Q}$
    $x \leftarrow$ value(F)
    F $\leftarrow$ succ(F)
```

Figure 2.3. Insertion into and extraction from a queue in linked allocation.

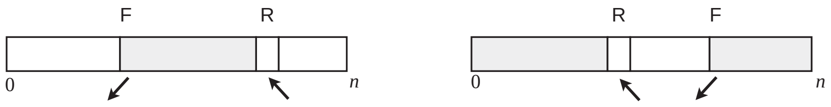


Figure 2.4. A queue in sequential allocation.

Suppose then that we decide that n elements should be enough, so we may initialize an array $Q[0], \ldots, Q[n-1]$. The pointers F and R will now be indices in the array, F will be the index of the front of the queue and it will be convenient to define R as the index of the element *following* the last one in the queue, that is, the index where a new element should be inserted, as seen in the left part of Figure 2.4, in which the elements of the queue appear in gray. The small arrows indicate that updates are performed only at the extremities of the queue.

Extracting or inserting an element then involve increasing the pointers F or R by 1. The queue will thus have a tendency to "move" to the right and will eventually reach the last element indexed $n-1$. We shall therefore consider the array as if it were cyclic by simply performing all increments modulo n. At some stage, the queue may thus look as depicted in the right side of Figure 2.4. The formal update procedures are given in Figure 2.5.

2.2.1 Example: Optimal Prefix Code

We shall see several examples of the usefulness of queues later in this book, and focus here on a possible solution to the following simple problem. We are given a set of n numbers $a_1, \ldots, a_n$, and should repeatedly remove the two smallest ones and add their sum as a new element, until only one element remains. This is a part of the solution of the problem of finding an optimal prefix code, to be studied in Chapter 11.

If the set is not ordered, then $n-1$ comparisons are required to find and remove the minimal element. The total number of required comparisons is thus

$$\sum_{i=2}^{n-1}(i+(i-1)) = \left(2\sum_{i=1}^{n-1} i\right) - n. \tag{2.3}$$

```
$\underline{Q \Longleftarrow x}$
    if R = F then overflow – STOP
    $Q[\mathsf{R}] \leftarrow x$
    R $\leftarrow$ (R + 1) mod $n$

$\underline{x \Longleftarrow Q}$
    $x \leftarrow Q[\mathsf{F}]$
    F $\leftarrow$ (F + 1) mod $n$
    if R = F then underflow
```

Figure 2.5. Insertion into and extraction from a queue in sequential allocation.

Background Concept: Summing the m First Integers

It might be useful to remember the summation $\sum_{i=1}^{m} i$ on the right-hand side of eq. (2.3), giving the sum of the first m integers, for some $m \geq 1$. Imagine the numbers written in order in a line:

$$1 \qquad 2 \qquad 3 \qquad 4 \qquad \cdots \qquad m-1 \qquad m$$

Now consider another line with the same numbers, but in reversed order:

$$m \qquad m-1 \qquad m-2 \qquad m-3 \qquad \cdots \qquad 2 \qquad 1$$

Adding by columns gives the sum $m+1$ in every column, and since there are m columns, this all adds up to $(m+1)m$. But this is twice the sum we are looking for, so

$$\sum_{i=1}^{m} i = \frac{(m+1)m}{2}.$$

Hence about n^2 comparisons are needed if the set is not ordered, and we shall see that the set can be sorted in only about $n \log n$ steps, and then be stored in a linked list. Extracting the two smallest elements takes then constant time, and the newly formed element has to be inserted in the proper place within the ordered list. The total running time for the $n-1$ iterations will be of the order of n comparisons, if we start the search where to insert a new element at the location of the previous insert.

Here is an alternative procedure, using two queues Q_1 and Q_2 rather than a linked list. The elements of Q_1 will be the original elements of the set, in nondecreasing order. The second queue Q_2 will initially be empty, and will contain only elements that are created by the algorithm, that is, an element whose value is the sum of two previously treated elements. In each iteration, the two smallest numbers in the combined set $Q_1 \cup Q_2$, x and y, are extracted. These are either the two first elements in Q_1, or those in Q_2, or the first elements of both Q_1 and Q_2. The key observation is that there is no need to search where to insert $x+y$: it cannot be smaller than any of the previously created combinations, so its place in the queue Q_2 must be at the rear end. The formal algorithm is given in Figure 2.6, where first(Q) and second(Q) refer to the first two elements of a queue Q.

The overall time to perform the task is thus linear in n, once the elements have been sorted, or of the order of $n \log n$ if the sorting has also to be accounted for.

```
if first$(Q_1)$ > second$(Q_2)$ then
        $x \Longleftarrow Q_2$        $y \Longleftarrow Q_2$
else    if first$(Q_2)$ > second$(Q_1)$ then
            $x \Longleftarrow Q_1$        $y \Longleftarrow Q_1$
        else
            $x \Longleftarrow Q_1$        $y \Longleftarrow Q_2$
$Q_2 \Longleftarrow x + y$
```

Figure 2.6. Iterative insertion of the sum of two smallest elements of a sorted list.

Background Concept: Asymptotic Notation

The following notation is widespread in Computer Science, and will replace the vague phrases like *of the order of* used earlier. It is generally used to describe the complexity of some algorithm as a function of the size n of its input. For two integer functions f and g, we write $f \in O(g)$, which is read as *f is in big-O of g*, if there are constants n_0 and $C > 0$ such that

$$\forall n \geq n_0 \qquad |f(n)| \leq C\,|g(n)|.$$

Roughly speaking, this means that f is bounded above by g, but the strictness of the bound is relaxed in two aspects:

(i) we are only interested in the asymptotic behavior of f, when its argument n tends to infinity, so the ratio of $f(n)$ to $g(n)$ for n smaller than some predefined constant n_0 is not relevant;
(ii) we do not require g itself to be larger than f – it suffices that some constant $C > 0$ multiplied by g be larger than f.

For example, $f(n) = n^2 + 10n\log_2 n + \frac{1000}{n} \in O(n^2)$, as can be checked using the constants $n_0 = 24$ and $C = 3$.

Though $O(g)$ is defined as a set, a prevalent abuse of notation refers to it as if it were a number, writing $f(n) = O(g(n))$. This is not really an equality, in particular, it is not transitive, as you cannot infer from $n^2 = O(n^3)$ and $n^2 = O(n^4)$ that $O(n^3) = O(n^4)$.

There is a similar notation also for lower bounds: $f \in \Omega(g)$, which is read as *f is in big-omega of g*, if there are constants n_0 and $C > 0$ such that

$$\forall n \geq n_0 \qquad |f(n)| \geq C\,|g(n)|.$$

For example, $n^2 - n \in \Omega(n^2)$.

Finally, for a more precise evaluation, we may use a combination of the bounds: $f \in \theta(g)$, which is read as *f is in theta of g*, if both $f \in O(g)$ and $f \in \Omega(g)$, obviously with different constants, that is, there are constants n_0, $C_1 > 0$ and $C_2 > 0$ such that

$$\forall n \geq n_0 \qquad C_1\ |g(n)| \leq |f(n)| \leq C_2\ |g(n)|.$$

For example, $\log n + \cos(n) \in \theta(\log n)$.

This symbolism can also be extended to the reals and to letting the argument tend to some constant, often 0, rather than to ∞.

2.3 Stacks

The counterpart of the First-In-First-Out paradigm of queues is known as LIFO, which stands for Last In, First Out. This might clash with our natural sense of fairness when applied to waiting in line, but is nevertheless useful in other scenarios. Possible examples are

- a pile of books, from which the elements can only be removed in the reverse of the order by which they have been added;
- a narrow, dead-ended parking lot, accommodating several cars in its length, but at most one in its width;
- luggage space in an airplane, so that the always late travelers arriving at the last moment at the check-in counter, will be the first ones to get their suitcases at destination. . .

Figure 2.7 is a schematic of a stack in both linked and sequential allocation. Insertions and deletions are now performed at the same end of the stack, identified by a pointer or index T (Top). In the upper part of Figure 2.7, the elements are numbered in order of their insertion into the stack. We shall use a similar notation as for queues, $S \Longleftarrow x$ standing for the insertion of an element with

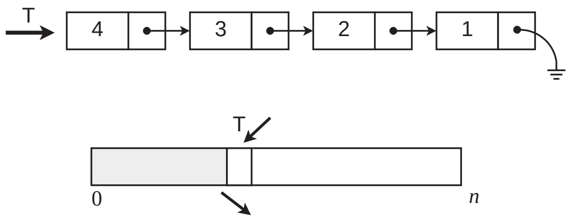


Figure 2.7. A stack in linked and sequential allocation.

```
$\underline{S \Longleftarrow x}$ (linked)
    allocate($node$)
    value($node$) $\leftarrow x$;  succ($node$) $\leftarrow$ T
    T $\leftarrow node$

$\underline{x \Longleftarrow S}$ (linked)
    $x \leftarrow$ value(T)
    T $\leftarrow$ succ(T)

$\underline{S \Longleftarrow x}$ (sequential)
    if T $= n$ then
        overflow – STOP
    $S$[T] $\leftarrow x$
    T $\leftarrow$ T $+ 1$

$\underline{x \Longleftarrow S}$ (sequential)
    if T $= 0$ then
        underflow – STOP
    T $\leftarrow$ T $- 1$
    $x \leftarrow S$[T]
```

Figure 2.8. Insertion into and extraction from a stack in linked and sequential allocation.

value x into the stack S, and $x \Longleftarrow S$ for the removal of the top element from the stack S and storing its value into x. These operations are often referred to as *push* and *pop*, respectively, and are formally given in Figure 2.8.

2.3.1 Example: Arithmetic Expression

Consider an arithmetic expression like those any programmer encounters on a daily basis, for example

$$4 + 3 \times 5.$$

While certain cheap pocket calculators still evaluate this to be 35 (when the $\times$ key is pressed, the display already shows 7), the more advanced ones correctly recognize that multiplication has priority over addition, so that the result should rather be 19. If the intention was indeed to add 4 to 3 before multiplying, parentheses should have been used, as in

$$(4 + 3) \times 5.$$

In fact, arithmetic expression are written the way they are probably mainly for historic reasons, as this way is not consistent with the customary form of writing functions. We usually write, e.g., $f(x)$ or $\log x$, preceding the argument x by the function name f or log, with or without parentheses. When several variables are involved, we write $f(x, y)$ rather than $x\ f\ y$. Similarly, we should have written the displayed expressions as add$(4,$ multiply$(3, 5))$ or multiply(add$(4, 3), 5)$.

This way of writing expressions is called *Polish notation*. Note that the parentheses are then redundant, so that the preceding expressions can be written unambiguously as

$$+\ 4\ \times\ 3\ 5 \qquad\qquad \text{or} \qquad\qquad \times\ +\ 4\ 3\ 5.$$

There exists also a *Reversed Polish notation*, in which the arguments precede the function name instead of following it. This makes even more sense as once the operation to be performed is revealed, its arguments are already known. The preceding in reversed Polish notation would be

$$4\ 3\ 5\ \times\ + \qquad\qquad \text{or} \qquad\qquad 4\ 3\ +\ 5\ \times.$$

Because of the positions of the operators, these notations are often referred to as prefix or postfix, whereas the standard way of writing is infix.

The evaluation of a long arithmetic expression given in infix seems to be quite involved: there are priorities, parentheses to change them if necessary, and defaults, like $a - b - c$ standing for $(a - b) - c$; but using $\uparrow$ for exponentiation, $a \uparrow b = a^b$, the default of $a \uparrow b \uparrow c$ is $a \uparrow (b \uparrow c)$, because $(a \uparrow b) \uparrow c = \left(a^b\right)^c = a^{bc}$. When the expression is given in postfix, on the other hand, the following simple procedure can be used to evaluate it by means of a single scan from left to right and using a stack S.

Operands are pushed into the stack in the order of appearance; when an operator X is encountered, the two last elements, y and z are popped from the stack, $z\ X\ y$ is calculated (note that the order of the operands is reversed), and its value is pushed again into S. At the end, the result is in the only element of S. Formally,

```
while end of expression has not been reached
    $X \leftarrow$ next element of the expression (operator or operand)
    if $X \notin$ {Operators}
        $S \Longleftarrow X$
    else
        $y \Longleftarrow S$;        $z \Longleftarrow S$
        $r \Longleftarrow$ compute($z\ \ X\ \ y$)
        $S \Longleftarrow r$
```

Figure 2.9 shows the stack after having read an operator, as well as at the end, when the expression to be evaluated, $(4 + 3) * \big(2 \uparrow (14 - 8)/2\big)$, has been converted into its corresponding postfix form $4\ \ 3\ +\ 2\ \ 14\ \ 8\ -\ 2\ /\ \uparrow\ *$.

The question is therefore how to convert an expression from infix to postfix. This is again done using a stack, which, in contrast to the stack in the example of Figure 2.9, holds the operators rather than the operands. We also need a table

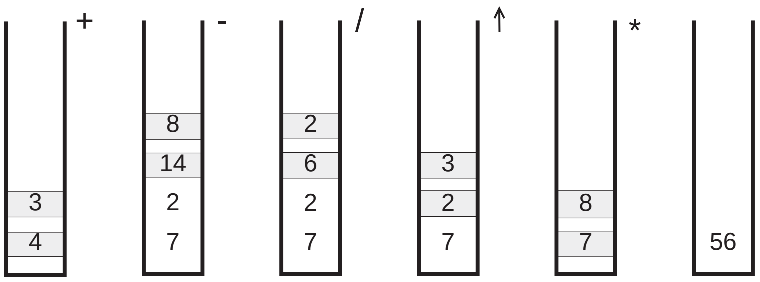


Figure 2.9. Evaluating an arithmetic expression using a stack.

of priorities, and actually, two such tables, called STACK-PRIORITY and INPUT-PRIORITY, will be used to deal with the special cases. Parentheses have highest priority in the input, since their purpose is to change the default priorities, but once in the stack, their priority should be the lowest. The two tables also enable the correct treatment of exponentiation, which associates to the right, as seen earlier. A dummy element with lowest priority is inserted into the stack as *sentinel*, to facilitate the treatment of boundary conditions.

The algorithm scans the infix expression left to right and transfers the operands directly to the output, in the order of their appearance. Operators, including left parentheses, are pushed into the stack, and are popped according to the order imposed by their priorities. The priority tables and the formal algorithm are given in Figure 2.10. The tables may be extended to deal also with

operator	sentinel	(	+ -	* /	↑	)
STACK-PRIORITY	-1	1	2	3	4	–
INPUT-PRIORITY	–	6	2	3	5	2

```
$S \Longleftarrow$ sentinel
while end of expression has not been reached
    $X \leftarrow$ next element of the expression
    if end of input then    while Top $\neq$ sentinel
                                  $y \Longleftarrow S$;     print $y$
    else    if $X \notin$ {Operators} then print $X$
            else    while STACK-PRIORITY(Top) $\geq$ INPUT-PRIORITY($X$)
                          $y \Longleftarrow S$;     print $y$
                    if $X$ is not right parenthesis ')' then
                          $S \Longleftarrow X$
                    else    dummy $\Longleftarrow S$    // pop matching left parenthesis
```

Figure 2.10. Priority tables and algorithm for conversion from infix to postfix.

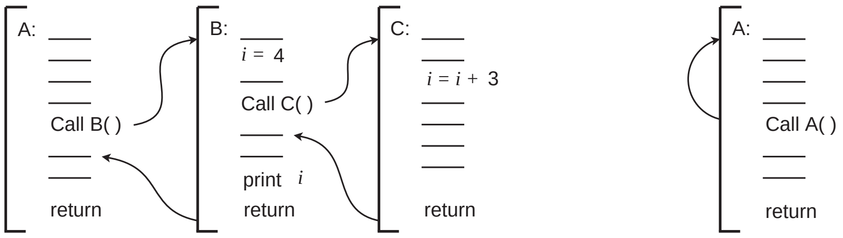


Figure 2.11. Chain of function calls.

other operators defined by the given programming language, such as assignments =, logical operations & | or shifts ≫.

2.3.2 Example: Handling Recursion

It is good programming practice to partition a given task into several small logical units, and large programs often consist of numerous procedures, creating chains of functions $f_1, f_2, f_3, \ldots$, where f_1 is calling f_2, which in turn calls f_3, etc. These functions are usually written by different programmers, and each comes with its own set of parameters and local variables. The challenge is to keep track of their correct values. Consider, for example, the function calls on the left side of Figure 2.11, we would not like the increment $i = i + 3$ in procedure C to affect the value of variable i in procedure B.

The problem is aggravated if the chain of calling functions includes more than one copy of a given function. This is called *recursion*, where, in its simplest form, a function calls itself, as depicted on the right-hand side of Figure 2.11. The commands of the recursive function appear physically only once in the memory of our computer, there is therefore a need to differentiate between the values the same variable gets at different generations of the function calls. This is done by means of a *program stack*.

Each entry of the program stack corresponds to a single function call and holds a record consisting of the following fields: the input parameters, the local variables, and an address from which to continue when returning after the execution of the current function call. The following actions are taken when a function call Call B( ) is encountered within a function A:

Call 1: Push a new entry into the stack.
Call 2: Update parameters and initialize local variables.
Call 3: In the return-address field, store the address of the command following the function call in the calling routine A.
Call 4: Continue executing commands from the beginning of B.

When exiting a function, generally via a return command, the actions are:

Ret 1: Read the return address field of the Top element into a variable adr.
Ret 2: Discard the Top element by Pop.
Ret 3: Continue executing commands from adr.

As example, we show what happens when executing a Mergesort procedure on a small input vector.

Background Concept: Mergesort

Mergesort is another example of the *Divide and Conquer* family. To sort an array $A[1], \dots, A[n]$, we apply mergesort recursively on the left and right halves of the array, that is on $A[1], \dots, A[\frac{n}{2}]$ and $A[\frac{n}{2}+1], \dots, A[n]$, and then use a function merge(i, k, j), which supposes that the subarrays $A[i], \dots, A[k]$ and $A[k+1], \dots, A[j]$ are already sorted, to produce a merged array $A[i], \dots, A[j]$. The merging may be implemented with the help of three pointers, two to the parts of the array that are to be merged, and one to an auxiliary array B, holding temporarily the sorted output, before it is moved back to the array A. The formal definition, in which the element following the last is defined by $A[j+1] \leftarrow \infty$ to deal with the boundary condition of one of the subarrays being exhausted, is given by:

```
merge($i, k, j$)
        $p_1 \leftarrow i$;      $p_2 \leftarrow k+1$;      $p_3 \leftarrow i$
        if $p_1 \le k$ and $A[p_1] < A[p_2]$ then
                $B[p_3] \leftarrow A[p_1]$;      $p_1$++;      $p_3$++
        else    $B[p_3] \leftarrow A[p_2]$;      $p_2$++;      $p_3$++
        $A[i] \cdots A[j] \leftarrow B[i] \cdots B[j]$
```

The number of comparisons for the merge is clearly bounded by $j - i$. The formal definition of mergesort(i, j), which sorts the subarray of A from index i to, and including, index j, is then

```
mergesort($i, j$)
        if $i < j$ then
                $k \leftarrow \lfloor (i+j)/2 \rfloor$
                mergesort($i, k$)
           L1:  mergesort($k+1, j$)
           L2:  merge($i, k, j$)
```

The analysis is similar to the one we did earlier for binary search. Let $T(n)$ denote the number of comparisons needed to sort an array of size n by mergesort, we get that $T(1) = 0$, and for $n > 1$:

$$T(n) = 2\,T(n/2) + n. \tag{2.4}$$

Applying equality (2.4) repeatedly yields

$$T(n) = 2^i\,T(n/2^i) + i\,n, \qquad \text{for} \quad i = 1, 2, \ldots. \tag{2.5}$$

To get rid of the T on the right-hand side of the equation, let us choose i large enough to get to the boundary condition, that is $n/2^i = 1$, so that $i = \log_2 n$. Substituting in (2.5), we get

$$T(n) = n \log_2 n.$$

Suppose our input vector A, indexed 1 to 5, contains the elements 9,2,1,5,3, in the given order. Somewhere in the calling program, the sorting procedure is invoked by

```
        ...
        mergesort(1, 5)
    L4: ...
```

When the call is executed, a new entry is created in the stack, as depicted in the lowest line of part (a) in Figure 2.12. Initially, the line contains only the parameters i and j, 1 and 5 in our case, and the return address L4 from which the program execution will continue after having finished with mergesort. The execution of the program continues from the beginning of the recursive function. Since $i = 1 < 5 = j$, the value 3 is assigned to k, and the value of k in the entry in the stack will be updated only at this stage. The next command is again a (recursive) call to mergesort. Hence a new entry is pushed onto the stack, with parameters 1 and 3, and return address L1. The process repeats twice until a fourth entry is created, with parameters 1 and 1, and return address L1.

At this point, no command is executed during the recursive call, and upon exiting from mergesort, the top entry is popped and the execution flow continues from L1. The values of k and j, 1 and 2, are taken from the entry which is now at the top of the stack. The next command is again a call to mergesort, and the program stack after this call is depicted in part (b) of Figure 2.12. When the current top element is popped, one gets to label L2, which invokes the merge procedure. This is the first access to the input vector A itself. In this case, subvectors of size 1 are merged, which is equivalent to just reordering

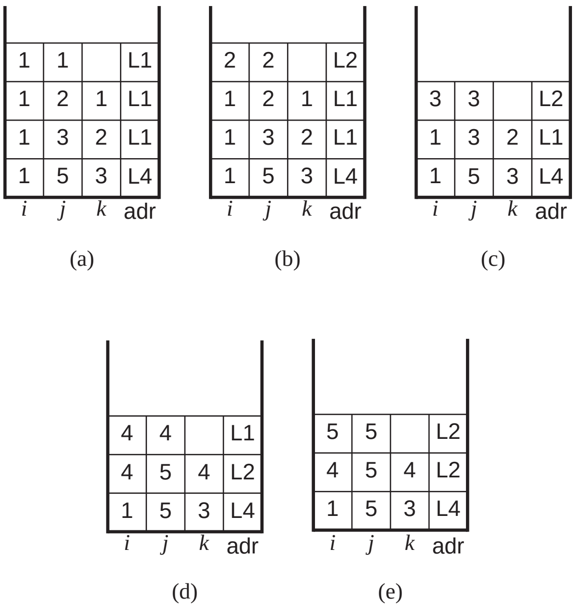


Figure 2.12. Mergesort: program stack when called on input vector 9, 2, 1, 5, 3.

the two elements involved. The vector A after the merge will thus contain the elements 2, 9, 1, 5, 3, and we have reached the end of one of the function calls. Execution thus continues at label L1, with the program stack containing the two lowest lines of Figure 2.12(c).

After the following call to merge, with parameters $(i, k, j) = (1, 2, 3)$, the vector A will contain 1, 2, 9, 5, 3, the line 1,3,2,L1 will be popped and the stack will contain only a single entry. The reader is encouraged to continue this simulation through parts (d) and (e) of the figure. The next merge, with parameters (4, 4, 5), should transform the vector A into 1, 2, 9, 3, 5, and then finally into 1, 2, 3, 5, 9.

2.4 Other Linear Lists

In the general family called *linear lists*, of which queues and stacks are special cases, the records are ordered according to some criterion. This order is given either implicitly by the position of the record in a sequential allocation, or explicitly by means of pointers in a linked allocation. There are many more linear lists and we shall mention only the following. Generally, the decision whether to prefer one type over another will be guided by the details of the

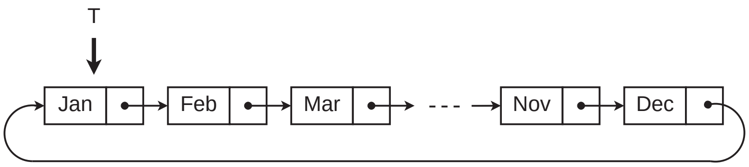


Figure 2.13. Circular list.

specific application at hand. One can then adapt the choice of the data structures to the tradeoff between the resources which best fits their purpose.

2.4.1 Circular Lists

A circular list is a linear list in which the last element points to the first one, thus exploiting the wasted space of the NIL pointer. This may be advantageous for lists that are intrinsically periodic like the months of the year. If one wishes to change the calendar year to the school or academic year, all one needs to do is to move the pointer T in Figure 2.13 to September or October.

There are, however, also applications for which it would seem natural to use a noncyclic linear list. Nevertheless, closing a cycle may permit the execution of certain operations more efficiently, for example, having the possibility to reach any record x by starting a linear search from any other record y. Another feature of a circular list is that only one pointer is needed (to the last element), which is useful to enable the efficient concatenation of lists.

The algorithms for inserting or deleting records from circular lists are straightforward , but special care should be taken when dealing with an empty list. A standard way, not only for circular lists, of facilitating the handling of special cases such as empty lists is the use of a so-called *sentinel* element: this is a record like the others of the given data structures, just that it does not carry any information. For large enough structures, the waste caused by such a dummy element can be tolerated and is generally outweighed by the benefits of the simplified processing.

2.4.2 Doubly Linked Lists

Additional flexibility may be gained by adding also a link from each record to its predecessor, in addition to the link to its successor. Whether the increased overhead can be justified will depend on the intended application: navigating through the list for searches will be facilitated, whereas updates when inserting or deleting records are becoming more involved. Figure 2.14 shows a doubly

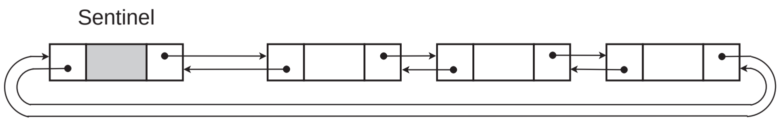


Figure 2.14. Doubly linked list with sentinel element.

linked list with three records and an additional sentinel element acting both as the predecessor of the first element, and as the successor of the last one.

A typical application of a doubly linked list would be for the management of dynamic allocations of memory space.

2.4.3 Deques

Unifying the properties of both queues and stacks, a *deque* allows inserts and deletes at both ends of the list, which for symmetry reasons will now be called Left and Right rather than Front and Rear. Like queues and stacks, deques can be managed in sequential allocation, as shown in Figure 2.15, or in linked allocation, usually as a special case of a doubly linked list.

Think of a deque as a model simulating the use of large elevators as found in large public facilities like hospitals, where one may enter or exit on two opposing sides. Similarly, certain airplanes may be boarded or left through front or rear doors.

2.4.4 Higher Order Linear Lists

All these ideas can be generalized to d dimensions, with $d \geq 2$. Even though we usually do not consider a two-dimensional matrix as a linear object, the linearity we refer to in this chapter relates to the fact that records can be arranged in order, in which each element has a well-defined successor and predecessor, and this may be true for each dimension. The number of pointers included in each record will increase with d, but this will often be compensated for by the more efficient algorithms.

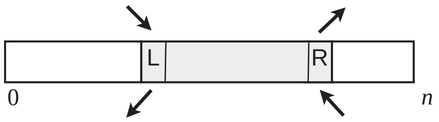


Figure 2.15. A deque in sequential allocation.

An example could be representing a sparse matrix (i.e., a matrix with a large number of elements that are 0) by linked lists of records defined only for the nonzero elements. The records will be linked together in both their rows and columns (and further dimensions, if necessary). This leads to more efficient handling of these and other special matrices, such as triangular.

Exercises

2.1 You have a linked list of n records, each consisting of a value field and a next-pointer. The length n of the list is not known, but may be assumed to be a multiple of 3. Show how to print the values of the records in the middle third of the list using only two pointers and without using a counter.

Apply the same idea to solve the following problem: given is a list like the preceding, show how to check in time $O(n)$ whether the list contains a cycle.

2.2 Let M be a matrix of dimension $n \times n$, and assume that only $O(n)$ of the n^2 elements of M are nonzero. Use a linked representation of the nonzero elements of M to calculate the *trace* of M^2 in time $O(n)$ (the trace of a matrix $A = (a_{ij})$ is the sum of the elements in its diagonal, $\sum_{i=1}^{n} a_{ii}$).

2.3 Let A be an array of n numbers. You know that there exists some index k, $1 \le k < n$, such that $A[1], \ldots, A[k]$ are all positive, and $A[k+1], \ldots, A[n]$ are all negative. Show how to find k:

(a) in time $O(\log n)$;
(b) in time $O(\min(\log n, k))$;
(c) in time $O(\log k)$.

2.4 A column of n cars is approaching a highway from an access road. The cars are numbered 1 to n, but not necessarily in order. We would like to rearrange the cars so that they enter the highway by increasing order of their indices. The problem is that the access road is so narrow that only one car fits in its width. However, just before the road meets the highway, there is a dead-ended parking lot, long enough to accommodate all cars, but again not wide enough for two cars side by side. Can the task of rearranging the cars be completed with the help of the parking lot, for every possible input permutation? If so, prove it, if not, give a counter-example.

2.5 A known game chooses the winner among n children in the following way: the children stand in a circle and transfer clockwise, at a regular pace, some object from one to another. At irregular intervals, at the sound of some bell, the child holding the object passes it to the next one and leaves the game, so that the circle shrinks. The winner is the last child remaining in the circle.

To program the game, simulate the bell by generating a random integer t between 1 and some given upper bound K; t will be the number steps the object will be forwarded until the next sound of the bell.

(a) What is the time complexity of the process until the winner is found, as a function of n and K, if the players are represented as records in a doubly linked cyclic deque?
(b) What is the time complexity of the process until the winner is found, as a function of n and K, if the players are represented as elements of an array?
(c) Compare the two methods, in particular for the case $K > n$.

2.6 Write the algorithms for inserting and deleting elements in circular doubly linked lists with a sentinel element. Take care in particular how to deal with the empty list.

2.7 Find constants n_0 and C to complete the examples given in the description of the asymptotic notation, showing that

(a) $n^2 - n \in \Omega(n^2)$
(b) $\log n + \cos(n) \in \theta(\log n)$.

2.8 The closed form for $T(n)$ defined by the recurrence relation of eq. (2.1) was $T(n) \in O(\log n)$.

(a) Show that if the recursion is $T(n) = 1 + T(\sqrt{n})$, with the same boundary condition as earlier, then the closed form is $T(n) \in O(\log \log n)$.
(b) What is the function $f(n)$ for which the recurrence relation $T(n) = 1 + T(f(n))$, again with the same boundary condition, yields as closed form $T(n) \in O(\log \log \log n)$?
(c) What is the closed form, if the recurrence is
$T(n) = 1 + T(\log n)$?