

10.2-6

The dynamic-set operation UNION takes two disjoint sets S_1 and S_2 as input, and it returns a set $S = S_1 \cup S_2$ consisting of all the elements of S_1 and S_2 . The sets S_1 and S_2 are usually destroyed by the operation. Show how to support UNION in $O(1)$ time using a suitable list data structure.

10.2-7

Give a $\Theta(n)$ -time nonrecursive procedure that reverses a singly linked list of n elements. The procedure should use no more than constant storage beyond that needed for the list itself.

10.2-8 ★

Explain how to implement doubly linked lists using only one pointer value $x.np$ per item instead of the usual two ($next$ and $prev$). Assume that all pointer values can be interpreted as k -bit integers, and define $x.np$ to be $x.np = x.next \text{ XOR } x.prev$, the k -bit “exclusive-or” of $x.next$ and $x.prev$. (The value NIL is represented by 0.) Be sure to describe what information you need to access the head of the list. Show how to implement the SEARCH, INSERT, and DELETE operations on such a list. Also show how to reverse such a list in $O(1)$ time.

10.3 Implementing pointers and objects

How do we implement pointers and objects in languages that do not provide them? In this section, we shall see two ways of implementing linked data structures without an explicit pointer data type. We shall synthesize objects and pointers from arrays and array indices.

A multiple-array representation of objects

We can represent a collection of objects that have the same attributes by using an array for each attribute. As an example, Figure 10.5 shows how we can implement the linked list of Figure 10.3(a) with three arrays. The array *key* holds the values of the keys currently in the dynamic set, and the pointers reside in the arrays *next* and *prev*. For a given array index x , the array entries $key[x]$, $next[x]$, and $prev[x]$ represent an object in the linked list. Under this interpretation, a pointer x is simply a common index into the *key*, *next*, and *prev* arrays.

In Figure 10.3(a), the object with key 4 follows the object with key 16 in the linked list. In Figure 10.5, key 4 appears in $key[2]$, and key 16 appears in $key[5]$, and so $next[5] = 2$ and $prev[2] = 5$. Although the constant NIL appears in the *next*

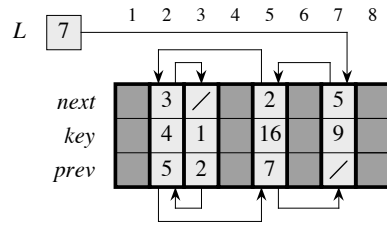


Figure 10.5 The linked list of Figure 10.3(a) represented by the arrays *key*, *next*, and *prev*. Each vertical slice of the arrays represents a single object. Stored pointers correspond to the array indices shown at the top; the arrows show how to interpret them. Lightly shaded object positions contain list elements. The variable L keeps the index of the head.

attribute of the tail and the *prev* attribute of the head, we usually use an integer (such as 0 or -1) that cannot possibly represent an actual index into the arrays. A variable L holds the index of the head of the list.

A single-array representation of objects

The words in a computer memory are typically addressed by integers from 0 to $M - 1$, where M is a suitably large integer. In many programming languages, an object occupies a contiguous set of locations in the computer memory. A pointer is simply the address of the first memory location of the object, and we can address other memory locations within the object by adding an offset to the pointer.

We can use the same strategy for implementing objects in programming environments that do not provide explicit pointer data types. For example, Figure 10.6 shows how to use a single array A to store the linked list from Figures 10.3(a) and 10.5. An object occupies a contiguous subarray $A[j \dots k]$. Each attribute of the object corresponds to an offset in the range from 0 to $k - j$, and a pointer to the object is the index j . In Figure 10.6, the offsets corresponding to *key*, *next*, and *prev* are 0, 1, and 2, respectively. To read the value of $i.\text{prev}$, given a pointer i , we add the value i of the pointer to the offset 2, thus reading $A[i + 2]$.

The single-array representation is flexible in that it permits objects of different lengths to be stored in the same array. The problem of managing such a heterogeneous collection of objects is more difficult than the problem of managing a homogeneous collection, where all objects have the same attributes. Since most of the data structures we shall consider are composed of homogeneous elements, it will be sufficient for our purposes to use the multiple-array representation of objects.

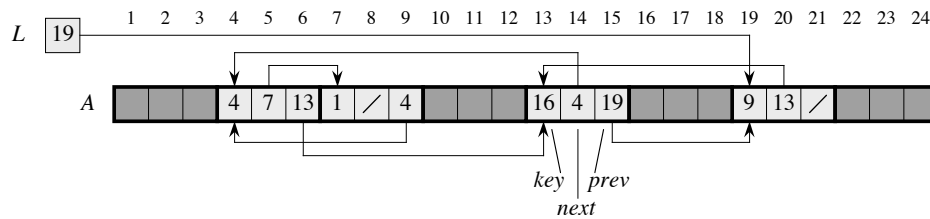


Figure 10.6 The linked list of Figures 10.3(a) and 10.5 represented in a single array *A*. Each list element is an object that occupies a contiguous subarray of length 3 within the array. The three attributes *key*, *next*, and *prev* correspond to the offsets 0, 1, and 2, respectively, within each object. A pointer to an object is the index of the first element of the object. Objects containing list elements are lightly shaded, and arrows show the list ordering.

Allocating and freeing objects

To insert a key into a dynamic set represented by a doubly linked list, we must allocate a pointer to a currently unused object in the linked-list representation. Thus, it is useful to manage the storage of objects not currently used in the linked-list representation so that one can be allocated. In some systems, a **garbage collector** is responsible for determining which objects are unused. Many applications, however, are simple enough that they can bear responsibility for returning an unused object to a storage manager. We shall now explore the problem of allocating and freeing (or deallocating) homogeneous objects using the example of a doubly linked list represented by multiple arrays.

Suppose that the arrays in the multiple-array representation have length m and that at some moment the dynamic set contains $n \leq m$ elements. Then n objects represent elements currently in the dynamic set, and the remaining $m - n$ objects are **free**; the free objects are available to represent elements inserted into the dynamic set in the future.

We keep the free objects in a singly linked list, which we call the **free list**. The free list uses only the *next* array, which stores the *next* pointers within the list. The head of the free list is held in the global variable *free*. When the dynamic set represented by linked list *L* is nonempty, the free list may be intertwined with list *L*, as shown in Figure 10.7. Note that each object in the representation is either in list *L* or in the free list, but not in both.

The free list acts like a stack: the next object allocated is the last one freed. We can use a list implementation of the stack operations PUSH and POP to implement the procedures for freeing and allocating objects, respectively. We assume that the global variable *free* used in the following procedures points to the first element of the free list.

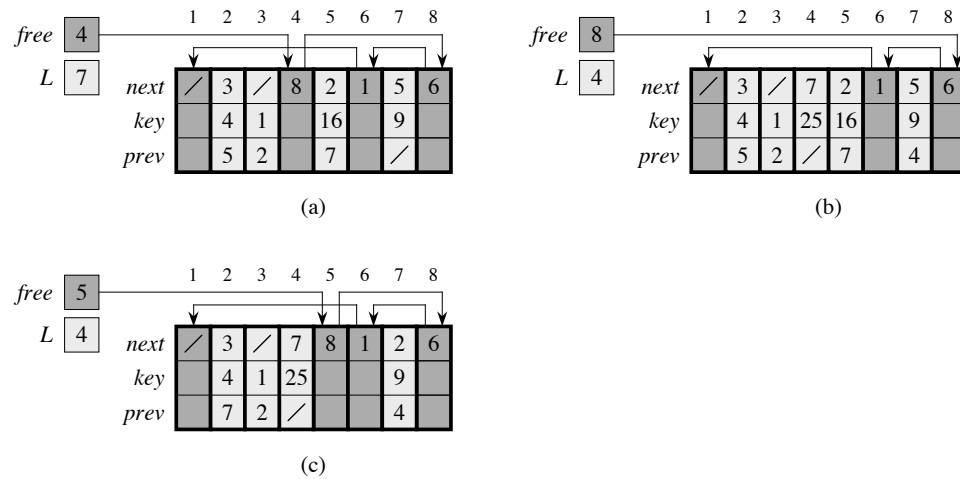


Figure 10.7 The effect of the `ALLOCATE-OBJECT` and `FREE-OBJECT` procedures. (a) The list of Figure 10.5 (lightly shaded) and a free list (heavily shaded). Arrows show the free-list structure. (b) The result of calling `ALLOCATE-OBJECT()` (which returns index 4), setting `key[4]` to 25, and calling `LIST-INSERT(L, 4)`. The new free-list head is object 8, which had been `next[4]` on the free list. (c) After executing `LIST-DELETE(L, 5)`, we call `FREE-OBJECT(5)`. Object 5 becomes the new free-list head, with object 8 following it on the free list.

`ALLOCATE-OBJECT()`

```

1  if free == NIL
2      error "out of space"
3  else x = free
4      free = x.next
5      return x

```

`FREE-OBJECT(x)`

```

1  x.next = free
2  free = x

```

The free list initially contains all n unallocated objects. Once the free list has been exhausted, running the `ALLOCATE-OBJECT` procedure signals an error. We can even service several linked lists with just a single free list. Figure 10.8 shows two linked lists and a free list intertwined through `key`, `next`, and `prev` arrays.

The two procedures run in $O(1)$ time, which makes them quite practical. We can modify them to work for any homogeneous collection of objects by letting any one of the attributes in the object act like a `next` attribute in the free list.

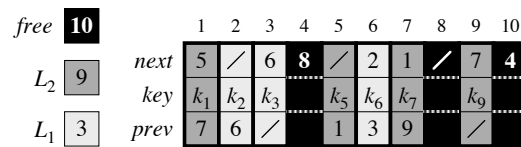


Figure 10.8 Two linked lists, L_1 (lightly shaded) and L_2 (heavily shaded), and a free list (darkened) intertwined.

Exercises

10.3-1

Draw a picture of the sequence $\langle 13, 4, 8, 19, 5, 11 \rangle$ stored as a doubly linked list using the multiple-array representation. Do the same for the single-array representation.

10.3-2

Write the procedures `ALLOCATE-OBJECT` and `FREE-OBJECT` for a homogeneous collection of objects implemented by the single-array representation.

10.3-3

Why don't we need to set or reset the *prev* attributes of objects in the implementation of the `ALLOCATE-OBJECT` and `FREE-OBJECT` procedures?

10.3-4

It is often desirable to keep all elements of a doubly linked list compact in storage, using, for example, the first m index locations in the multiple-array representation. (This is the case in a paged, virtual-memory computing environment.) Explain how to implement the procedures `ALLOCATE-OBJECT` and `FREE-OBJECT` so that the representation is compact. Assume that there are no pointers to elements of the linked list outside the list itself. (*Hint*: Use the array implementation of a stack.)

10.3-5

Let L be a doubly linked list of length n stored in arrays *key*, *prev*, and *next* of length m . Suppose that these arrays are managed by `ALLOCATE-OBJECT` and `FREE-OBJECT` procedures that keep a doubly linked free list F . Suppose further that of the m items, exactly n are on list L and $m - n$ are on the free list. Write a procedure `COMPACTIFY-LIST(L, F)` that, given the list L and the free list F , moves the items in L so that they occupy array positions $1, 2, \dots, n$ and adjusts the free list F so that it remains correct, occupying array positions $n + 1, n + 2, \dots, m$. The running time of your procedure should be $\Theta(n)$, and it should use only a constant amount of extra space. Argue that your procedure is correct.