9.1 Counting Sort

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. When $k = O(n)$, the sort runs in $\Theta(n)$ time.

Counting sort determines, for each input element x, the number of elements less than x. It uses this information to place element x directly into its position in the output array. For instance, if 17 elements are less than x, then x belongs in output position 18. We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want to put them all in the same position.

In the code for counting sort, we assume that the input is an array $A[1, .. n]$, and thus $A.length = n$. We require two auxiliary arrays: the array $B[1 .. n]$ holds the sorted output, and the array $C[0 .. k]$ provides temporary working storage.

Note that here we are using a zero-indexed array, C, for the first time, because C holds the counts of each integer from 0 to k present in array A.

COUNTING-SORT(A, B, k)

```
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3        C[i] = 0
 4  for j = 1 to A.length
 5        C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i
 7  for i = 1 to k
 8        C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i
10  for j = A.length downto 1
11        B[C[A[j]]] = A[j]
12        C[A[j]] = C[A[j]] − 1
```

Using the count of each integer, we want to drop the integers in the right positions in array B such that B is sorted. We populate array C such that $C[i]$ contains the number of elements less than or equal to i. In COUNTING-SORT procedure, line 4 to 5 keeps the counts of each integer in array A, while line 7 to 8 populates C such that $C[i]$ contains the number of elements less than or equal to i.

Finally, the **for** loop of lines 10-12 places each element $A[j]$ into its correct sorted position in the output array B. If all n elements are distinct, then when we first enter line 10, for each $A[j]$, the value of $C[A[j]]$ is the correct final position of $A[j]$ in the output array, since there are $C[A[j]]$ elements less than or equal to $A[j]$. Because the elements might not be distinct, we decrement $C[A[j]]$ each time we place a value $A[j]$ into the B array. Decrementing $C[A[j]]$ causes the next input element with a value equal to $A[j]$, if one exists, to go to the position immediately before $A[j]$ in the output array.

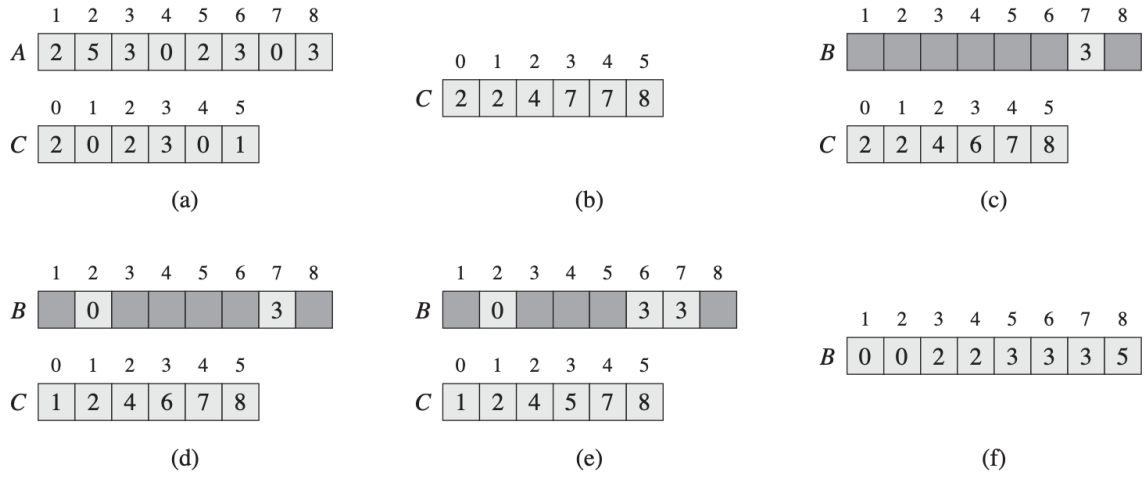

Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

The overall runtime is $\Theta(k+n)$. In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$.

It beats the lower bound of $\Omega(n \lg n)$ proved with decision model tree in case of comparison sort. You must note that this is not a comparison sort though. There is no comparison used in the entire procedure above. Instead, counting sort uses the actual values of the elements to index into an array. The $\Omega(n \lg n)$ lower bound for sorting does not apply when we depart from the comparison sort model.

Elementary Data Structures

Dynamic sets and dictionary operations

Sets are as fundamental to computer science as they are to mathematics. Whereas mathematical sets are unchanging, the sets manipulated by algorithms can grow, shrink, or otherwise change over time. We call such sets ***dynamic***.

Algorithms may require several different types of operations to be performed on sets. For example, many algorithms need only the ability to insert elements into, delete elements from, and test membership in a set. We call a dynamic set that supports these operations a ***dictionary***.

Elements of a dynamic set

In a typical implementation of a dynamic set, each element is represented by an object whose attributes can be examined and manipulated if we have a pointer to the object. Some kinds of dynamic sets assume that one of the object's attributes is an identifying ***key***. If the keys are all different, we can think of the dynamic set as being a set of key values. The object may contain ***satellite data***, which are carried around in other object attributes but are otherwise unused by the set implementation. It may also have attributes that are manipulated by the set operations; these attributes may contain data or pointers to other objects in the set.

Operations on dynamic sets

Operations on a dynamic set can be grouped into two categories: ***queries***, which simply return information about the set, and ***modifying operations***, which change the set. Here is a list of typical operations.

- SEARCH(S, k): A query that, given a set S and a key value k, returns a pointer x to an element in S such that $x.key = k$, or NIL if no such element belongs to S.
- INSERT(S, x): A modifying operation that augments the set S with the element pointed to by x. We usually assume that any attributes in element x needed by the set implementation have already been initialized.
- DELETE(S, x): A modifying operation that, given a pointer x to an element in the set S, removes x from S.
- MINIMUM(S): A query on a totally ordered set S that returns a pointer to the element of S with the smallest key.
- MAXIMUM(S): A query on a totally ordered set S that returns a pointer to the element of S with the largest key.
- SUCCESSOR(S, x): A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next larger element in S, or NIL if x is the maximum element.
- PREDECESSOR(S, x): A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next smaller element in S, or NIL if x is the minimum element.

9.2 Stacks and queues

Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation prespecified. In a **stack**, the element deleted from the set is the one most recently inserted: the stack implementation a ***last-in, first-out*** or ***LIFO***, policy. Similarly, in a ***queue***, the element deleted is always the one that has been in the set for the longest time: the queue implements a ***first-in, first-out***, or ***FIFO***, policy. We will discuss on implementing stacks and queues using a simple array.

9.2.1 Stacks

The INSEERT operation on a stack is often called PUSH, and the DELETE operation, which does not take an element argument, is often called POP. These names are allusions to physical stacks, such as the spring-loaded stacks of plates used in cafeterias. The order in which plates are popped from the stack is the reverse of the order in which they were pushed onto the stack, since only the top plate is accessible.

As Figure 10.1 shows, we can implement a stack of at most n elements with an array $S[1, . . n]$. The array has an attribute $S.top$ that indexes the most recently inserted element. The stack consists of elements $S[1 . . S.top]$, where $S[1]$ is the element at the bottom of the stack and $S[S.top]$ is the element at the top.

When $S.top = 0$, the stack contains no element and is ***empty***. We can test to see whether the stack is empty by query operation STACK-EMPTY. If we attempt to pop an empty stack, we say the stack ***underflows***, which is normally an error. If $S.top$ exceeds n, the stack ***overflows***. (In our pseudocode implementation, we don't worry about stack overflow.)

```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
3  else
4       return FALSE
```

```
PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x
```

```
POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else
4       S.top = S.top − 1
5       return S[S.top + 1]
```

Figure 10.1 shows the effects of the modifying operations PUSH and POP. Each of the three stack operations takes $O(1)$ time.

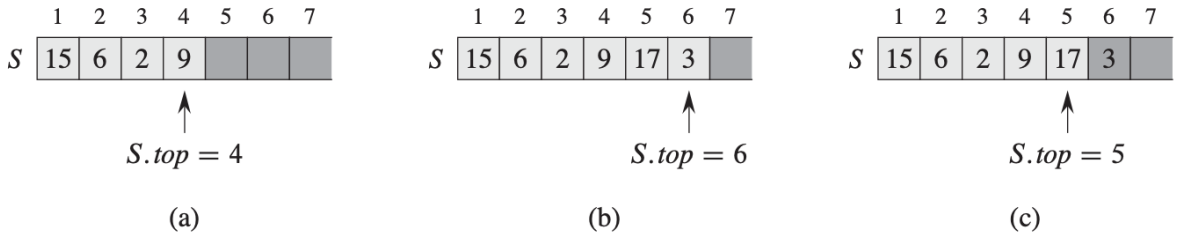

Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH$(S, 17)$ and PUSH$(S, 3)$. **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

9.2.2 Queues

We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE; like the stack operation POP, DEQUEUE takes no element argument. The FIFO property of a queue causes it to operate like a line of customers waiting to pay a cashier. The queue has a ***head***, and a ***tail***. When an element is enqueued, it takes its place at the tail of the queue, just as a newly arriving customer takes a place at the end of the line. The element dequeued is always the one at the head of the queue, like the customer at the head of the line who has waited the longest.

Figure 10.2 shows one way to implement a queue of at most $n - 1$ elements using an array $Q[1..n]$. The queue has an attribute $Q.head$ that indexes, or points to, its head. The attribute $Q.tail$ indexes the next location at which a newly arriving element will be inserted into the queue. The elements in the queue reside in locations $Q.head$, $Q.head + 1$, ..., $Q.tail - 1$, where we "wrap around" in the sense that the location 1 immediately follows location n in a circular order. When $Q.head = Q.tail$, the queue is empty. Initially, we have $Q.head = Q.tail = 1$. If we attempt to dequeue an element from an empty queue, the queue underflows. When $Q.head = Q.tail + 1$, the queue is full, and if we attempt to enqueue an element, then the queue overflows. When $Q.head = Q.tail + 1$, the queue is full, and if we attempt to enqueue an element, then the queue overflows.

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3        Q.tail = 1
4  else
5        Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else
5        Q.head = Q.head + 1
6  return x
```

Figure 10.2 shows the effects the ENQUEUE and DEQUEUE operations. Each operation takes $O(1)$ time.

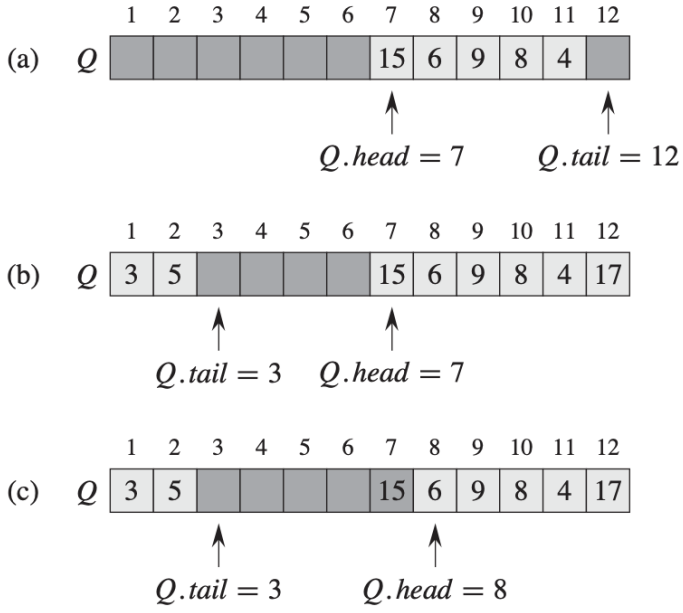

Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$, and ENQUEUE$(Q, 5)$. **(c)** The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.

9.3 Linked List

A ***linked list*** is a data structure in which the objects are arranged in a linear order. Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object.

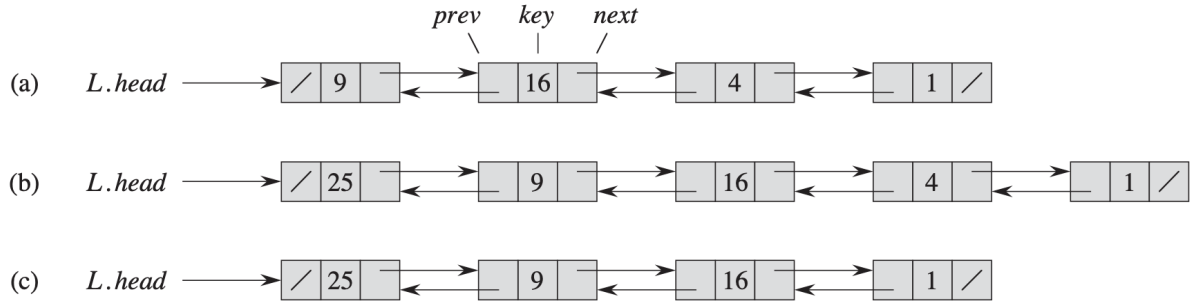

Figure 10.3 **(a)** A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute $L.head$ points to the head. **(b)** Following the execution of LIST-INSERT(L, x), where $x.key = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. **(c)** The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

As shown in Figure 10.3, each element of a ***doubly linked list*** L is an object with an attribute key and two other pointer attributes: $next$ and $prev$. Given an element x in the list, $x.next$ points to its successor in the linked list, and $x.prev$ points to its predecessor. If $x.prev = \text{NIL}$, the element x has no predecessor and is therefore the first element, or ***head***, of the list. If $x.next = \text{NIL}$, the element x has no successor and is therefore the last element, or ***tail***, of the list. An attribute $L.head$ points to the first element of the list. If $L.head = \text{NIL}$, the list is empty. There are several forms of a linked list such as ***singly linked list***, ***circular list***, and ***doubly linked list***. We will discuss ***unsorted*** and ***doubly linked*** list in the following section.

9.3.1 Searching a linked list

The procedure LIST-SEARCH(L, k) finds the first element with key k in list L by a simple linear search, return a pointer to this element. If no object with key k appears in the list, then the procedure returns NIL. Figure 10.3(a), the call LIST-SEARCH$(L, 4)$ returns a pointer to the third element, and the call LIST-SEARCH$(L, 7)$ returns NIL.

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3        x = x.next
4  return x
```

To search a list of n objects, the LIST-SEARCH procedure takes $\Theta(n)$ time in the worst case, since it may have to search the entire list.

9.3.2 Inserting into a linked list

Given an element x whose *key* attribute has already been set, the LIST-INSERT procedure "splices" x onto the front of the linked list, as shown in Figure 10.3(b).

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3        L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

The running time for LIST-INSERT on a list of n elements is $O(1)$.

9.3.3 Deleting from a linked list

The procedure LIST-DELETE removes an element x from a linked list L. It must be given a pointer to x, and it then "splices" x out of the list by updating pointers. If we wish to delete an element with a given key, we must first call LIST-SEARCH to retrieve a pointer to the element.

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

Figure 10.3(c) shows how an element is deleted from a linked list. LIST-DELETE runs in $O(1)$ time, but if we wish to delete an element with a given key, $\Theta(n)$ time is required in the worst case because we must first call LIST-SEARCH to find the element.

9.3.4 Sentinels (Optional)

The code for LIST-DELETE would be simpler if we could ignore the boundary conditions at the head and tail of the list:

```
LIST-DELETE'(L, x)
1  x.prev.next = x.next
2  x.next.prev = x.prev
```

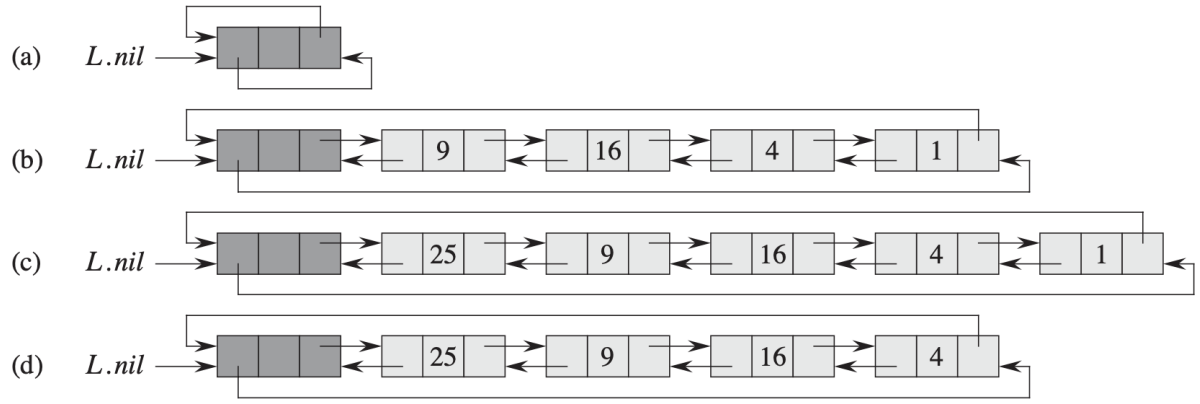

Figure 10.4 A circular, doubly linked list with a sentinel. The sentinel $L.nil$ appears between the head and tail. The attribute $L.head$ is no longer needed, since we can access the head of the list by $L.nil.next$. **(a)** An empty list. **(b)** The linked list from Figure 10.3(a), with key 9 at the head and key 1 at the tail. **(c)** The list after executing LIST-INSERT$'(L, x)$, where $x.key = 25$. The new object becomes the head of the list. **(d)** The list after deleting the object with key 1. The new tail is the object with key 4.

A ***sentinel*** is a dummy object that allows us to simplify boundary conditions. For example, suppose that we provide with list L an object $L.nil$ that represents NIL but has all the attributes of the other objects in the list. Whenever we have a reference to NIL in list code, we replace it by a reference to the sentinel $L.nil$. As shown in Figure 10.4, this change turns a regular doubly linked list into a ***circular, doubly linked list with a sentinel***, in which the sentinel $L.nil$ lies between the head and tail. The attribute $L.nil.next$ points to the head of the list, and $L.nil.prev$ points to the tail. Similarly, both the $next$ attribute of the tail and the $prev$ attribute of the head point to $L.nil$. Since $L.nil.next$ points to the head, we can eliminate the attribute $L.head$ altogether, replacing references to it by references to $L.nil.next$. Figure 10.4(a) shows that an empty list consists of just the sentinel, and both $L.nil.next$ and $L.nil.prev$ point to $L.nil$.

The code of LIST-SEARCH remains the same as before, but with the references to NIL and $L.head$ changed as specified above:

LIST-SEARCH'(L, x)

1 $x = L.nil.next$
2 **while** $x \neq L.nil$ and $x.key \neq k$
3 $\quad x = x.next$
4 **return** x

LIST-INSERT'(L, x)

1 $x.next = L.nil.next$
2 $L.nil.next.prev = x$
3 $L.nil.next = x$
4 $x.prev = L.nil$

Figure 10.4 shows the effects of LIST-INSERT' and LIST-DELETE' on a sample list.

- Sentinels rarely reduce the asymptotic time bounds of data structure operations, but they can reduce constant factors.
- When there are many small lists, the extra storage used by their sentinels can represent significant wasted memory.
- We use sentinels only when they truly simplify the code.