CSCI 470: Counting Sort, Stack and Queue

Vijay Chaudhary

Wed. September 27

Department of Electrical Engineering and Computer Science
Howard University

Overview

1. Quick Update
2. Counting Sort
3. Stack
4. Queue

Quick Update

Quick update

- HW 01 has been graded. You can collect it after the class.
- We are grading your exam.
- HW 02 deadline has been extended.

Counting Sort

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.

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

- **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.
- In the code for counting sort, we assume that the input is an array $A[1..n]$, and thus $A.length = n$

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.
- 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.

Counting sort: pseudocode

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
```

Illustration

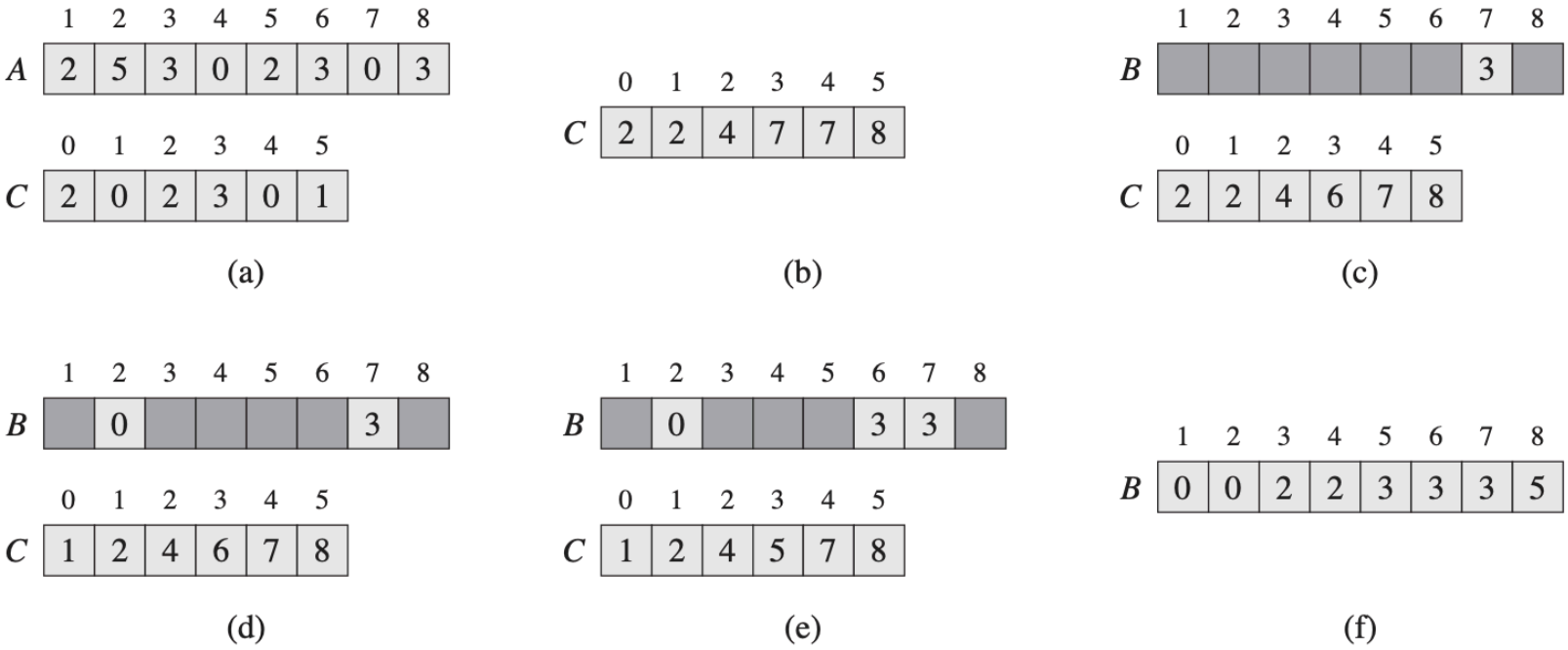

Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after 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.

Counting sort: runtime

- 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.

Stack

Stack

- 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.

Stack: Stack-Empty

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

Stack: Push

PUSH(S, x)

1 $S.top = S.top + 1$
2 $S[S.top] = x$

Stack: Pop

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

Stack: illustration

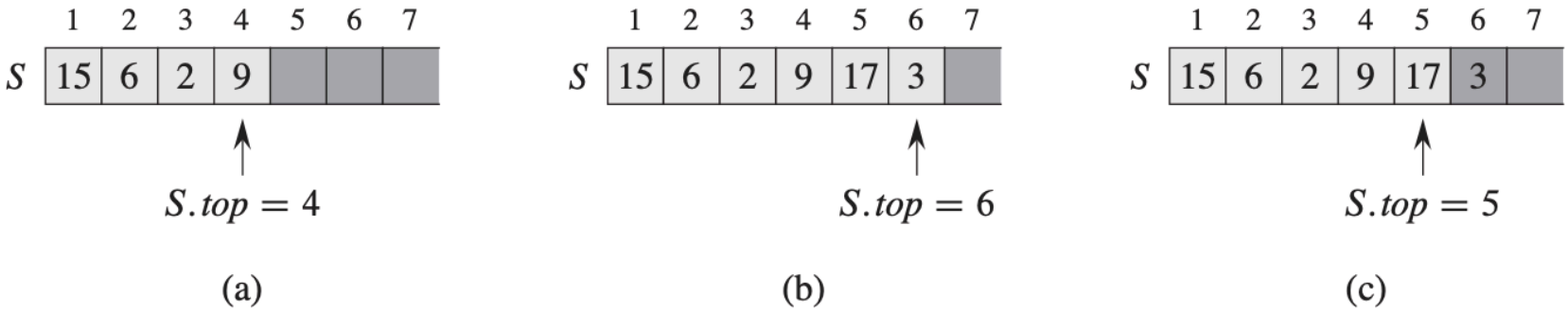

Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH$(S, 17)$ and PUSH$(S, 3)$. **(c)** Stack S after the call POP(S) has returned the element 3, which is 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.

Queue

Queue

- 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.

Queue: Enqueue

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
```

Queue: Dequeue

```
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
```

Queue: illustration

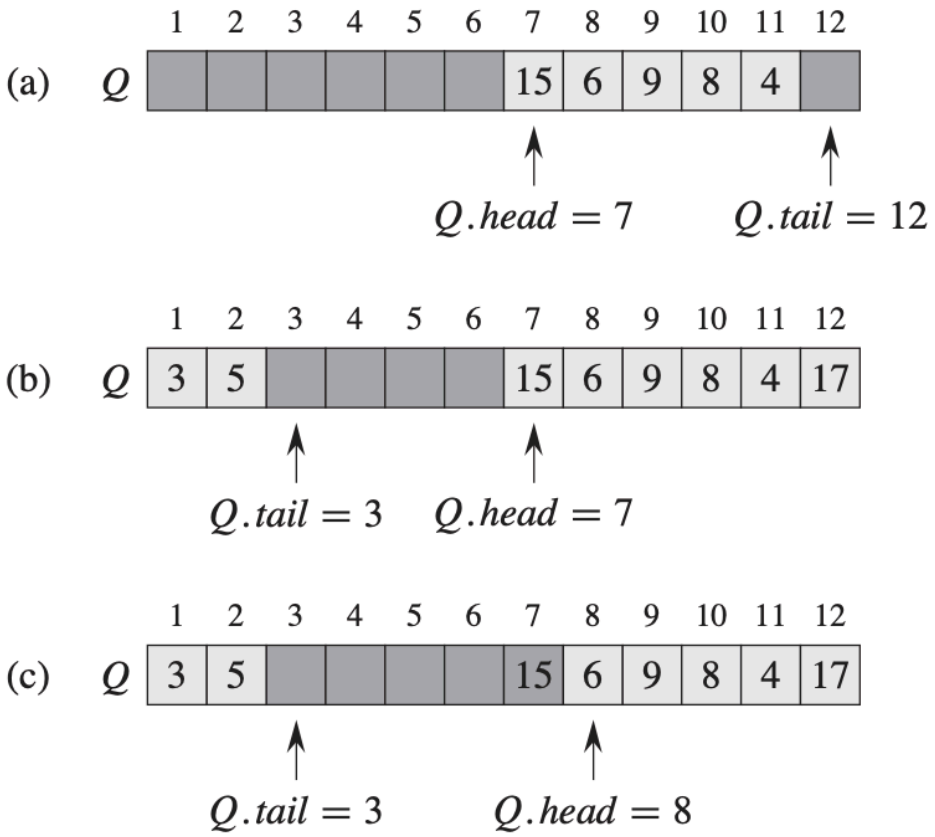

Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$, and ENQUEUE$(Q, 5)$. **(c)** The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