Elementary Data Structures

Chapter 11

CSCE310: Data Structures and Algorithms

`www.cse.unl.edu/~choueiry/S01-310/`

Berthe Y. Choueiry (Shu-we-ri)

Ferguson Hall, Room 104

`choueiry@cse.unl.edu`, Tel: (402)472-5444

Sets are fundamental in CS

Sets are dynamic structures:

they grow, shrinke, change over time

Typical operations:

- insert element
- delete element
- test membership

Objects:

1. key → dynamic set as a set of key values
2. satellite data (unused information)

Two types of operations:

1. Queries: `Search(S, k)`, `Minimum(S)`, `Maximum(S)`, `Successor(S, k)`, `Predecessor(S, k)`
2. Modifications: `Insert(S, k)`, `Delete(S, k)`

Time to execute a set operation generally measured in terms of the <u>size of the set given as one of its parameters</u>

Dynamic sets as simple data structures using pointers

- Stacks
- Queues
- Linked lists
- Rooted trees
- *etc* .

Assume: you are familiar with arrays

Stack and queues

`Delete(S, k)` ≡ `Delete(S)`, deleted element is pre-specified

In stacks: most recent element entered, **LIFO**
In queues: oldest element entered, **FIFO**

Implementation: *e.g.*, array

Stack

Insert operation: `Push (S, x)`

Delete operation: `Pop (S)`

Image: spring-loaded stacks of plates in cafeteria

Order popped is reverse of order pushed (LIFO)

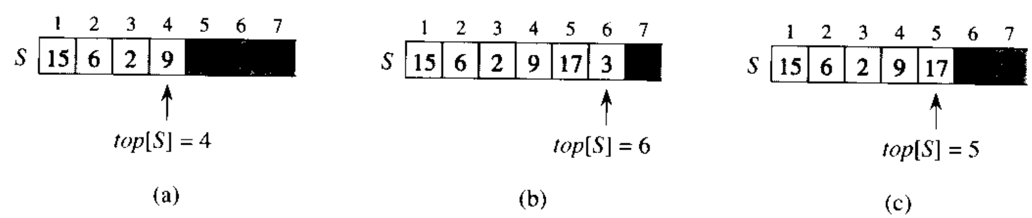


Figure 11.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH(S, 17) and PUSH(S, 3). **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

For a stack of n elements (at most): $S[1..n]$

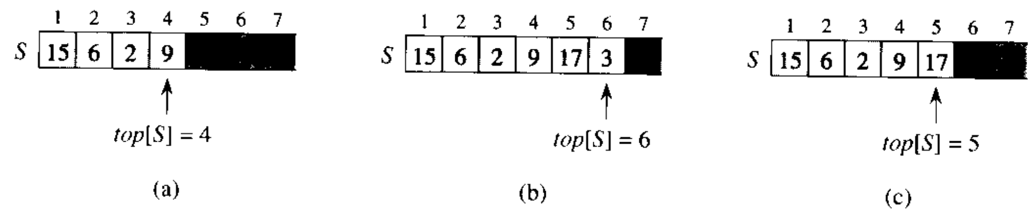


Figure 11.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH(S, 17) and PUSH(S, 3). **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

For a stack of n elements (at most): $S[1..n]$

Attributes of S: `top` (remember `length`?)

`top[S]`: index of most recently inserted element

Stacks elements: `S[1.. top[S]]`

Bottom of stack: `S[1]`

Top of stack: `S[top[S]]`

Empty stack: `top[S]` $= 0$

Test for emptiness: `Stack-empty(S)`

```
STACK-EMPTY(S)
if top[S] = 0
    then return TRUE
    else return FALSE
```

Popping on empty stack: underflow

```
PUSH(S, x)
top[S] ← top[S] + 1
S[top[S]] ← x
```

Pushing on full stack: overflow (`top[S]]` $> n$)

```
POP(S)
if STACK-EMPTY(S)
    then error "underflow"
    else top[S] ← top[S] − 1
         return S[top[S] + 1]
```

Pseudo-code for `Push`(`Pop`) increment (decrement) counter

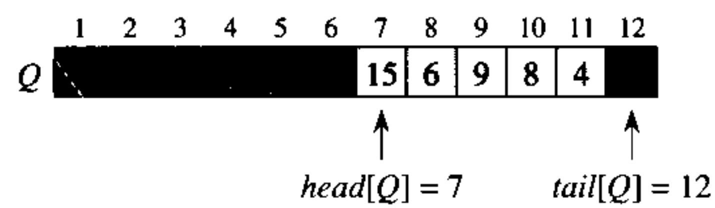
Queues

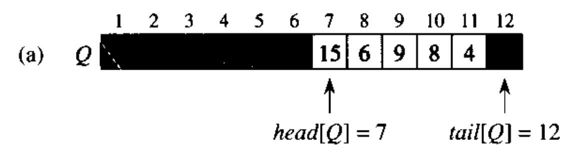
Insert operation: `Enqueue (S, x)`

Delete operation: `Dequeue (S)`

Image: line of people at the post office

Order popped is reverse of order pushed (LIFO)

(a) 

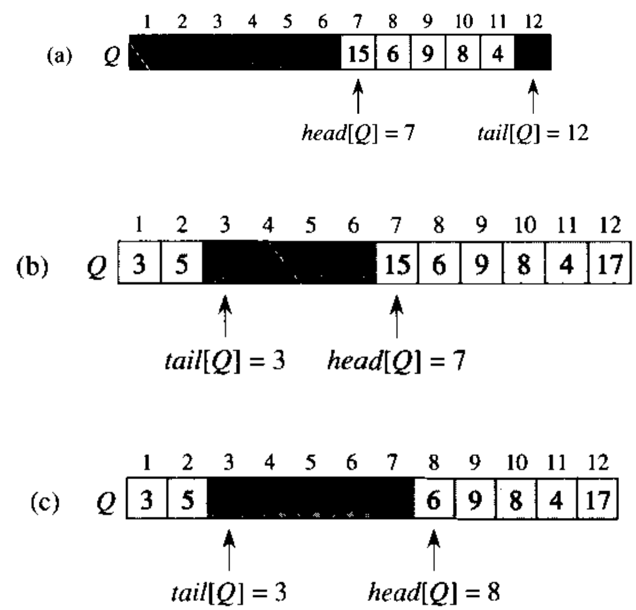


Attributes of S: `head` and `tail`

Enqueued: added at the tail, dequeued: always from head

Elements in queue are in positions:

`head[Q]`, `head[Q]`$+1$, ..., `tail[Q]`-1



Wrap around (circular): position 1 directly follows position n

Empty queue: `head[Q]`=`tail[Q]`

Full queue: `head[Q]`=`tail[Q]`+1

Initially: `head[Q]`=`tail[Q]`=1

Dequeue empty list: underflow

```
DEQUEUE(Q)
x ← Q[head[Q]]
if head[Q] = length[Q]
    then head[Q] ← 1
    else head[Q] ← head[Q] + 1
return x
```

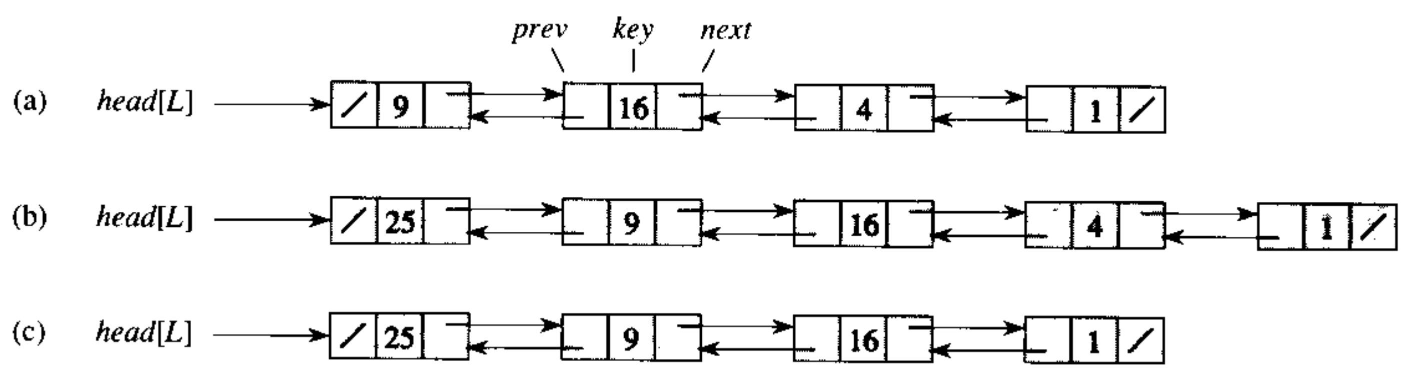
Enqueue full queue: overflow

```
ENQUEUE(Q, x)
Q[tail[Q]] ← x
if tail[Q] = length[Q]
    then tail[Q] ← 1
    else tail[Q] ← tail[Q] + 1
```

Pseudo-code for `Enqueue`(`Dequeue`) increment (decrement) tail (head)

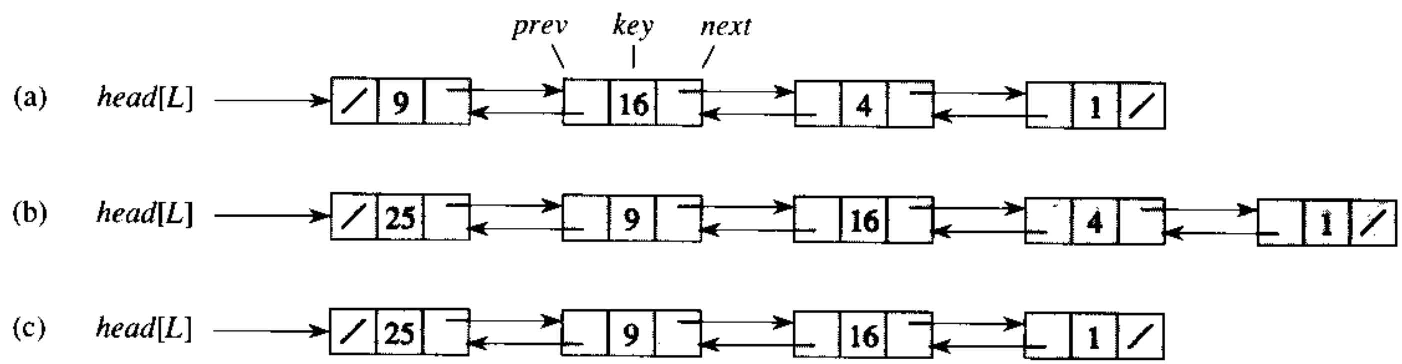
Linked list

Objects arranged in a linear order according to a pointer in each object



Each element `x`:

- 1 key field
- 2 pointers fields (`next[x]`, `prev[x]`)

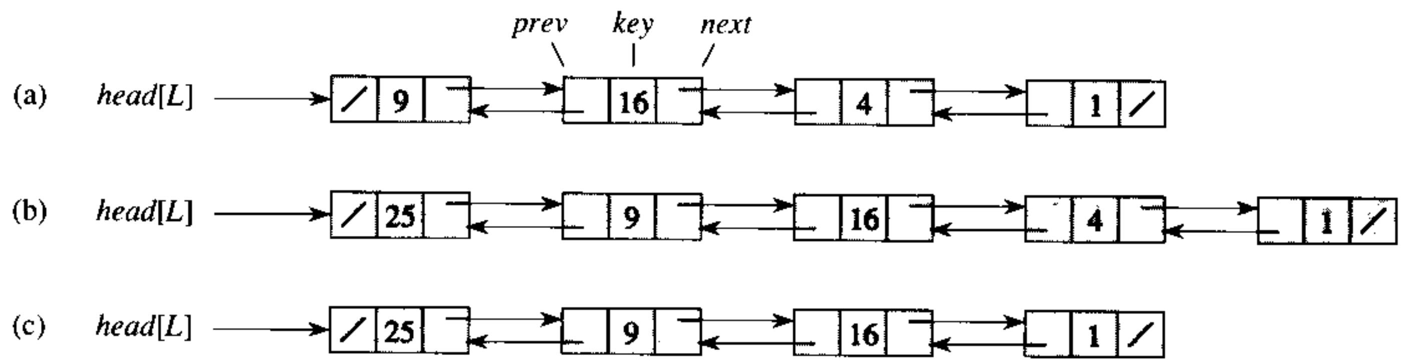


First element, head: `prev[x]` = nil

Last element, tail: `next[x]` = nil

Attribute `Head[L]` points to first element in list

Empty list: `Head[L]` = nil



Simply linked list (no `prev`) vs. doubly linked list

Circular vs. non-circular list

Sorted list (vs. unsorted)

- linear order of items ≡ linear order of keys
- minimum item is head, maximum item is tail

Circular list (ring of elements):

prev of head is tail, and next of tail is head