WIA2005: Algorithm Design and Analysis

Semester 2, Session 2016/17

Lecture 7: Stacks, Queues and Linked List

Learning objectives

- Know and understand:
  - Stacks
  - Queues
  - Linked List

Stacks and queues

- Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is pre-specified.
- In a ***stack***, the element deleted from the set is the one most recently inserted: the stack implements 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.

Stack – Insert (Push) and Delete (Pop)

- The INSERT operation on a stack is often called PUSH, and the DELETE operation, which does not take an element argument, is often called POP.

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

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

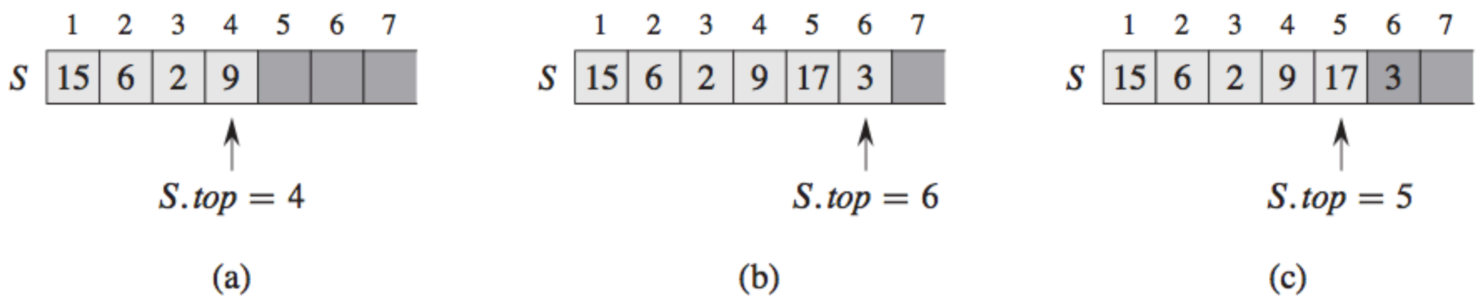
POP(S)
1 **if** STACK-EMPTY(S)
2 **error** "underflow"
3 **else** $S.top = S.top - 1$
4 **return** $S[S.top + 1]$

Imagine stack as a stack of plates, put the last one on top, take the from the top as well.

Each of the three stack operations takes O(1) time.

Stacks Operation

- 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.



Queues – Insert (Enqueue) and Delete (Dequeue)

- We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE.
- The queue has a ***head*** and a ***tail***.
- When an element is en- queued, it takes its place at the tail of the queue.
- The element dequeued is always the one at the head of the queue.

Imagine queue as, well, a queue! Enqueue from the back(tail), dequeue from the front (head)

Each of the operations takes O(1) time.

ENQUEUE(Q, x)

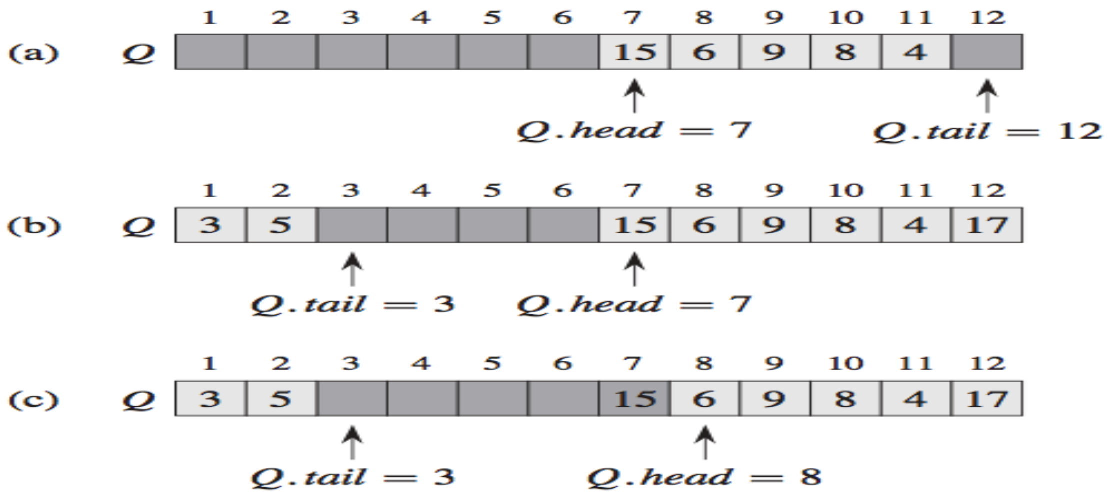
$Q[Q.tail] = x$
if $Q.tail == Q.length$
    $Q.tail = 1$
else $Q.tail = Q.tail + 1$

DEQUEUE(Q)

$x = Q[Q.head]$
if $Q.head == Q.length$
    $Q.head = 1$
else $Q.head = Q.head + 1$
return x

Queue Operation

- 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.

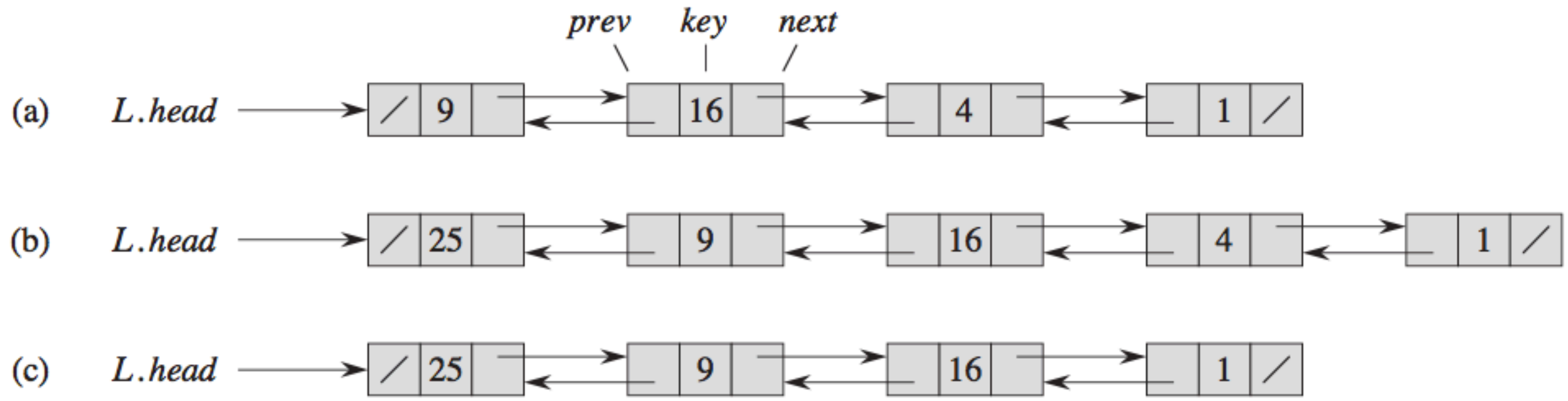


Linked List

- 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.
- A list may have one of several forms:
  - singly linked or doubly linked.
  - sorted or not.
  - circular or not.

Doubly linked list

- **(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 *pre* 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.



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, returning a pointer to this element.
- If no object with key k appears in the list, then the procedure returns NIL.
- For the linked list in (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)
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

Can you calculate the running time for this function?
(Try before you go to the next slides)

Running time for LIST-SEARCH

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

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 (b).

```
LIST-INSERT(L, x)
x.next = L.head
if L.head ≠ NIL
    L.head.prev = x
L.head = x
x.prev = NIL
```

What about this one. Can you calculate the running time for this function? (Try before you go to the next slides)

Running time for LIST-INSERT

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

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)
if x.prev ≠ NIL
    x.prev.next = x.next
else L.head = x.next
if x.next ≠ NIL
    x.next.prev = x.prev
```

Calculate the running time for this function. What if we need to delete a certain key?
(Try before you go to the next slides)

Running time for LIST-DELETE

- LIST-DELETE runs in O(1) time, but if we wish to delete an element with a given key, O(n) time is required in the worst case because we must first call LIST-SEARCH to find the element.

Sentinels

- The code for LIST-DELETE would be simpler if we could ignore the boundary conditions at the head and tail of the list.
- A ***sentinel*** is a dummy object that allows us to simplify boundary conditions.
- We should use sentinels judiciously. When there are many small lists, the extra storage used by their sentinels can represent significant wasted memory.

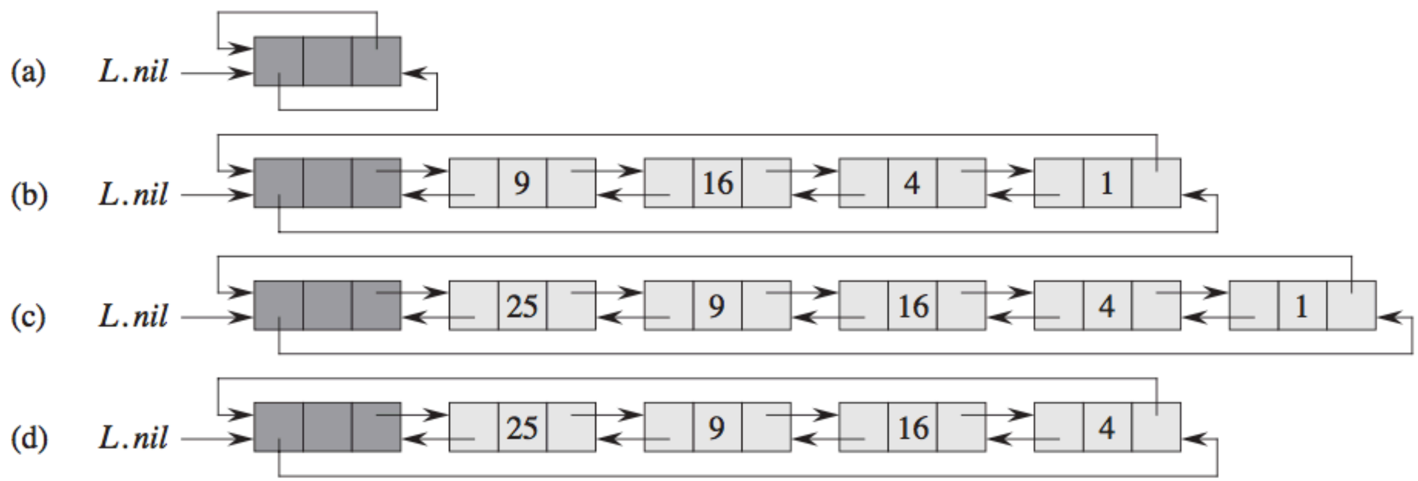
LIST-DELETE$'(L, x)$

1 $x.prev.next = x.next$

2 $x.next.prev = x.prev$

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 doubly linked list (a) (previous slides), with key 9 at the head and key 1 at the tail.
- **(c)** The list after executing LIST-INSERT0(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.



Search with a sentinel

- The code for LIST-SEARCH remains the same as before, but with the references to NIL and L.*head* had changed.

LIST-SEARCH$'(L, k)$

$x = L.nil.next$
while $x \neq L.nil$ and $x.key \neq k$
  $x = x.next$
return x

Delete and insert with a sentinel

- We use the two-line procedure LIST-DELETE’ from before to delete an element from the list.

- The following procedure inserts an element into the list:

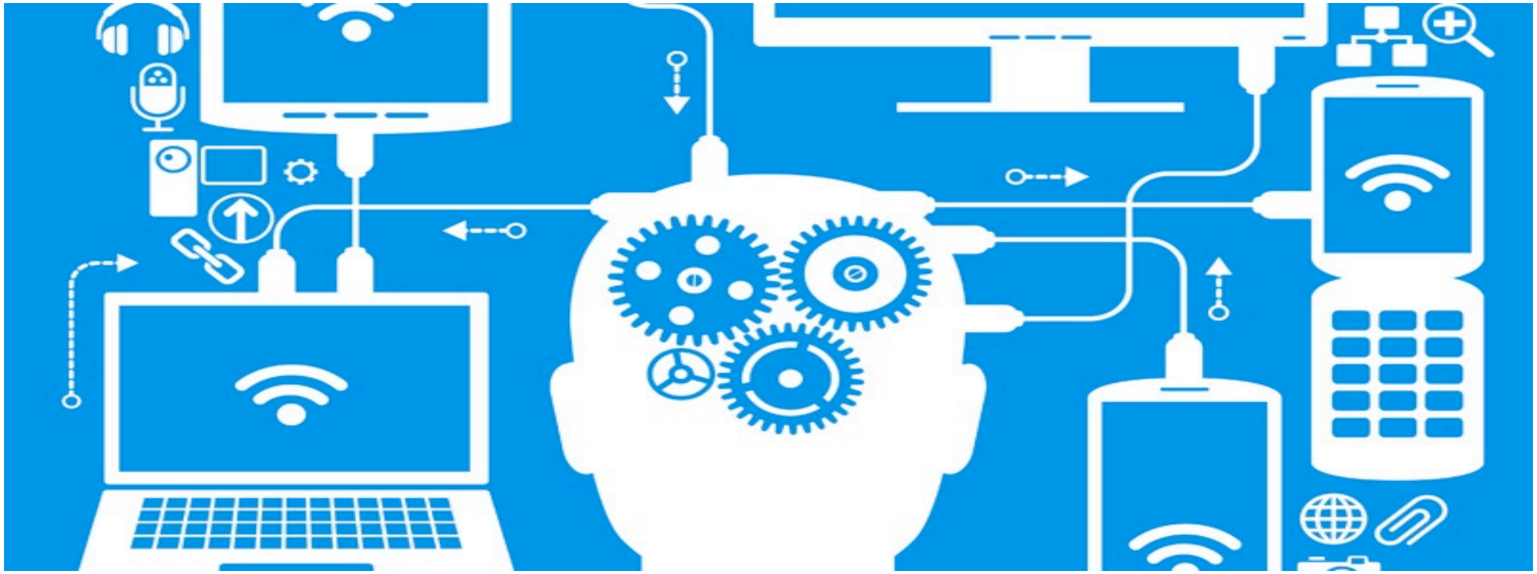
$\text{LIST-INSERT}'(L, x)$

$x.next = L.nil.next$
$L.nil.next.prev = x$
$L.nil.next = x$
$x.prev = L.nil$

Reference

- Cormen, Lieserson and Rivest, Introduction to Algorithms, Third Edition, MIT Press, 2009.

In the next lecture..



Lecture 8: Hash tables