Introduction to Algorithms
Data structures

Dynamic sets & Data structures

Dynamic sets: the sets can grow, shrink, or otherwise change over time.

Dictionary operations:

- Insert
- Delete
- Test membership

Elements of dynamic set:

- key(maybe different, ordered)
- Satellite data

Operations on dynamic sets

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. (Note that this operation takes a pointer to an element x, not a key value.)

Operations on dynamic sets

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.

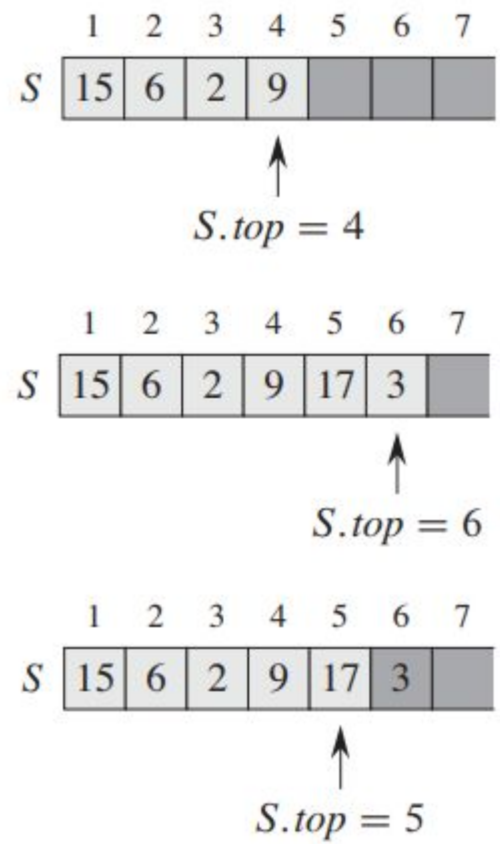
Stacks & Queues

Insert and Remove Policies:

- LIFO: Last In First Out(stack)
- FIFO: First In First Out(queue)

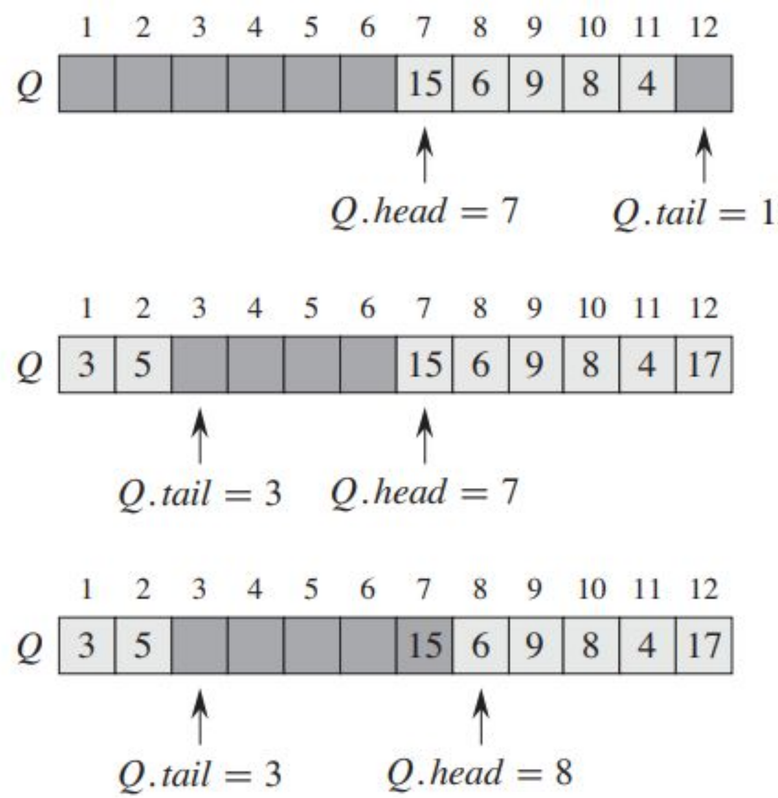
Stacks

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

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

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



Queues

```
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
```



Exercise

10.1-6

Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

10.1-7

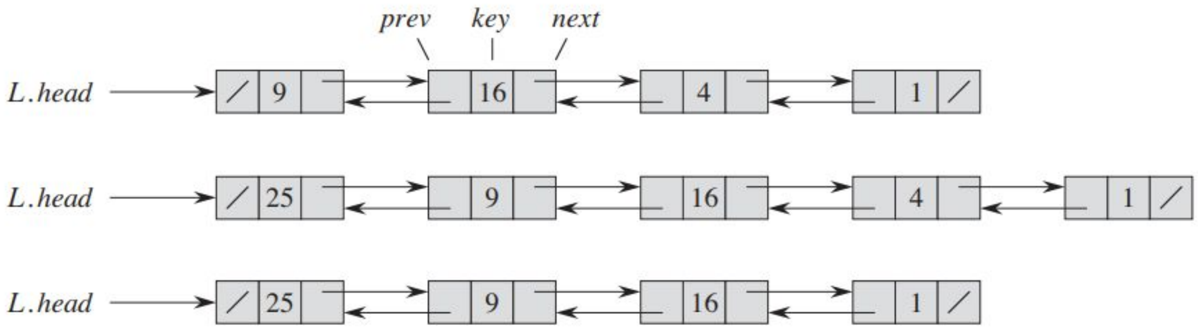
Show how to implement a stack using two queues. Analyze the running time of the stack operations.

Linked lists

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. Linked lists provide a simple, flexible representation for dynamic sets.

Types:

- Singly linked
- Duobly linked
- Sorted
- Unsorted
- Circular list



Linked list operations

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

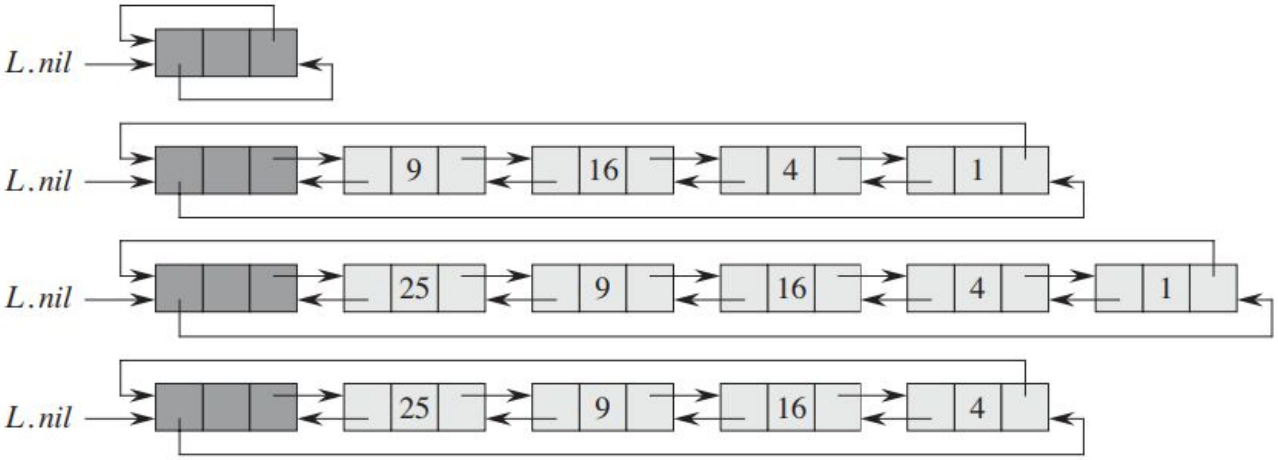
```
LIST-INSERT(L, x)
x.next = L.head
if L.head ≠ NIL
    L.head.prev = x
L.head = x
x.prev = NIL
```

```
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
```

```
LIST-DELETE′(L, x)
x.prev.next = x.next
x.next.prev = x.prev
```

Sentinels

- **L.nil** lies between the head and tail.
- **L.nil.next** points to the head of the list
- **L.nil.pre** points to the tail



Sentinels

```
LIST-SEARCH'(L, k)
x = L.nil.next
while x ≠ L.nil and x.key ≠ k
    x = x.next
return x
```

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

```
LIST-INSERT'(L, x)
x.next = L.nil.next
L.nil.next.prev = x
L.nil.next = x
x.prev = L.nil
```

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

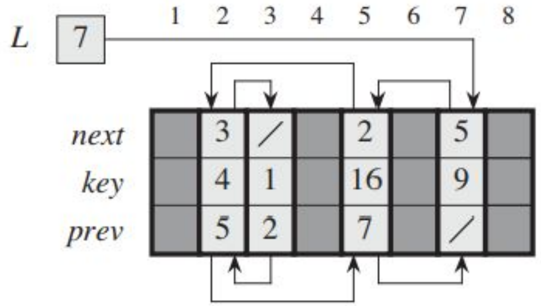
Exercise

10.2-1
Can you implement the dynamic-set operation INSERT on a singly linked list in $O(1)$ time? How about DELETE?

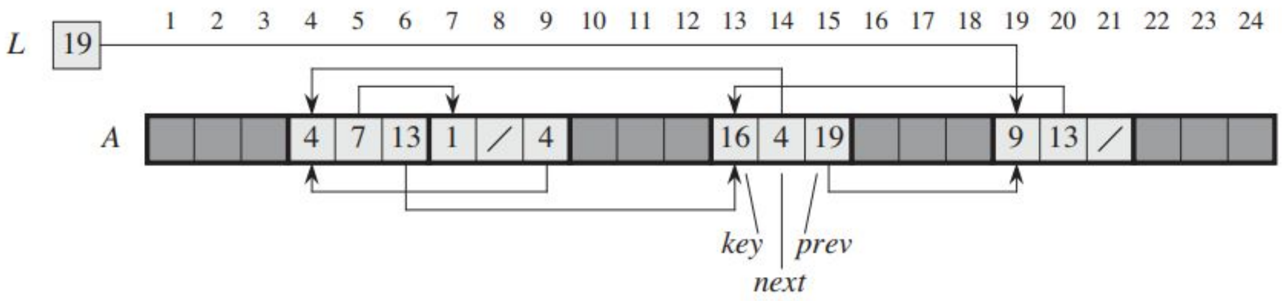
10.2-2
Implement a stack using a singly linked list L. The operations PUSH and POP should still take $O(1)$ time.

Implementing pointers and objects

- Multi-array representation



- Single-array representation



Exercise

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.