Chapter 10: Elementary Data Structures

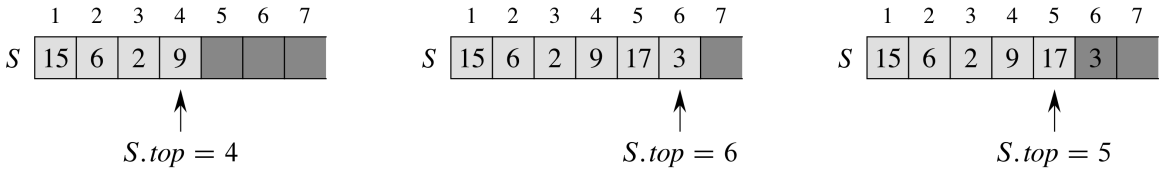
Section 10.1: Stacks and Queues

- **Stack: last-in, first-out data structure.**
- Array representation for a stack:
  - An array $S[1..n]$ is allocated to store elements.
  - An array attribute top[S] points to an array index in which the most recently inserted element resides. Initially, we set top[S] to 0.

```
Stack-Empty(S)
1. if S.top = 0
2.      return true                      O(1)
3. else return false

Push(S, x)       // store at most n elements
1. if (S.top + 1 > S.length)
2.      error "stack overflow"          O(1)
3. else S.top = S.top + 1
4.      S[S.top] = x

Pop(S)
1. if Stack-Empty(S)
2.      error "stack underflow"
3. else S.top = S.top - 1            O(1)
4.      return S[S.top + 1]
```

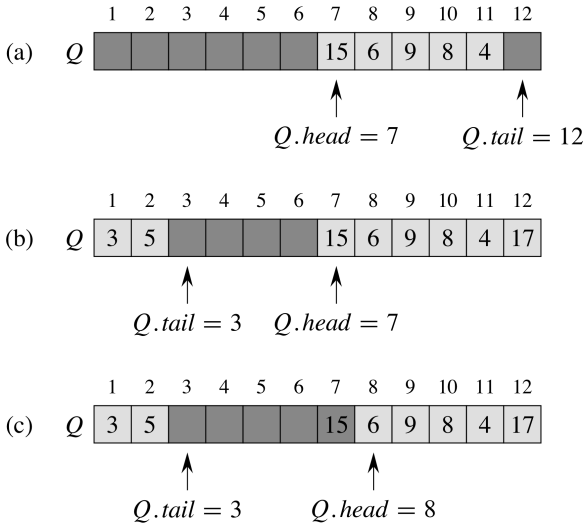
- **Queue: first-in, first-out data structure.**
- A queue has a **head** and a **tail**.
- Array representation for a queue:
  - An array $Q[1..n]$ is allocated to store elements. The array will be considered as a circular array. It will store a queue with at most $n-1$ elements.
  - An array attribute $Q.head$ points to an array index in which the earliest inserted element resides. Another array attribute $Q.tail$ points to an array index where the new element should be inserted.
  - Empty queue: $Q.head = Q.tail$

- **full** queue: $Q.head = Q.tail + 1$ or $Q.head = 1$ and $Q.tail = Q.length$
- An initial empty queue: $Q.head = 1$ and $Q.tail = 1$.



(a) (b) (c)

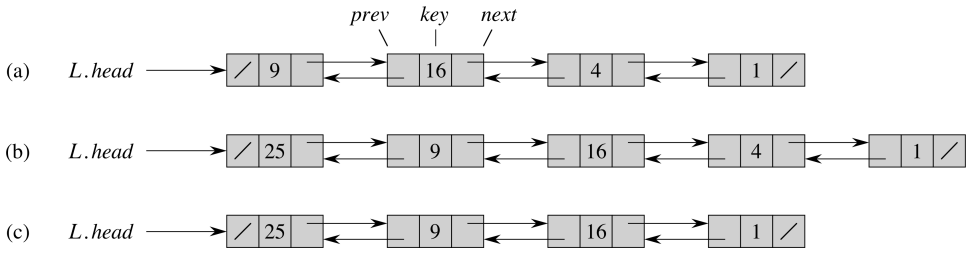
```
Queue-Empty(Q)
1. if Q.head = Q.tail
2.      return true                          O(1)
3. else return false

EnQueue(Q, x)      // store at most n-1 elements
1. if (Q.head = ((Q.tail+1) mod Q.length))
2.      error "queue overflow"
3. else Q[Q.tail] = x                  O(1)
4.      if Q.tail = Q.length
5.           Q.tail = 1
6.       else Q.tail = Q.tail+1
                                                  Give some examples
DeQueue(Q)
1. if Queue-Empty(Q)
2.      error "queue underflow"
3. else x = Q[Q.head]
4.      if Q.head = Q.length                O(1)
5.           Q.head = 1
6.      else Q.head = Q.head+1
7. return x
```



Recommended Exercises: 10.1-1, 10.1-3, 10.1-5. **Challenging**: 10.1-6, 10.1-7.

Section 10.2: Linked Lists

- Linked lists provide the dynamic storage versus the fixed storage of arrays.
- In a doubly linked list L, an element (object) x contain at least 3 fields:
  1. `key`[x] return the key value of x.
  2. `next`[x] return the pointer to the next element after x.
  3. `prev`[x] return the pointer to the previous element before x.
- For the list L, an attribute `head`[L] points to the first element in L. We may also keep track of the tail of the list.
- In a singly linked list L, an element x has at least two fields: `x.key` and `x.next`.
- For a linked list, it may be singly or doubly, sorted or not sorted, circular or non-circular.



Assume that **doubly, unsorted and non-circular** linked lists are used for the following procedures.

```
// Searching a linked list: go through the list one element at a time.
//                      return a node pointer if found; otherwise return nil.
```

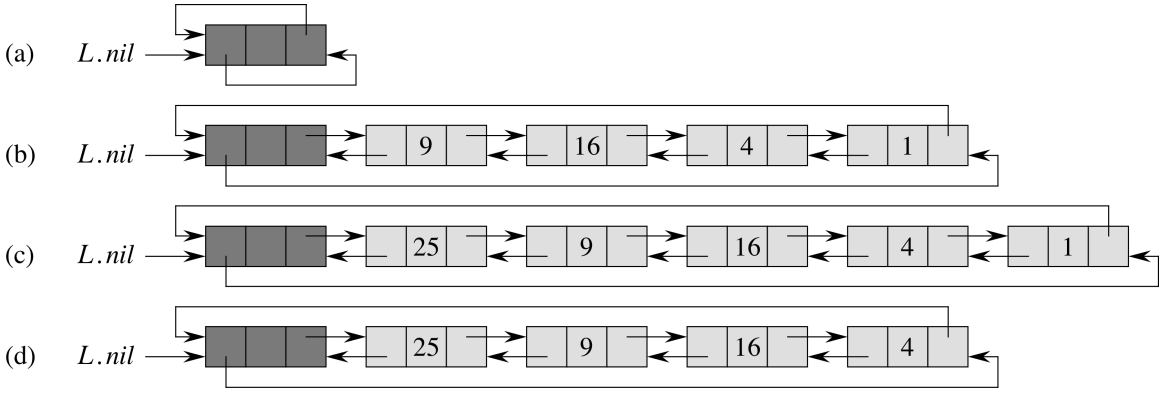
```
List-Search(L, k)  // k is a key value
1. x = L.head
2. while x != nil and x.key != k         // Theta(n) in worst-case
3.        x = x.next
4. return x

// Insertion: insert a new element x in front of the list
List-Insert-Front(L, x)
1. x.next = L.head
2. if L.head != nil
3.    then L.head.prev = x            // O(1)
4. L.head = x
5. x.prev = nil

// Deletion: remove an element x from the list.
//           Assume x is indeed in the list.
List-Delete(L, x)
1. if x.prev != nil
2.    then x.prev.next = x.next
3.    else L.head = x.next            // O(1)
4. if x.next != nil
5.    then x.next.prev = x.prev
```

To delete a given key k rather than a given node from a list, it requires to call List-search(L, k) to find the node pointer x, and then call List-Delete(L, x).

Totally, it takes $\Theta(n)$ time in the worst-case.

Sentinels: Using sentinels to simplify the linked list procedures. In this case, the sentinel is an empty object *L.nil* that represents the NIL but has all the attributes of the other objects in the list.

Wherever we have a reference to NIL, we replace it with a reference to the sentinel *L.nil*. This turns our list into a circular, doubly-linked list with a sentinel (between the tail an the head)



```
// Searching a linked list: go through the list one element at a time.
//                      return a node pointer if found; otherwise return nil.
List-Search'(L, k)  // k is a key value
1. x = L.nil.next
2. while x != L.nil and x.key != k         // Theta(n) in worst-case
3.        x = x.next
4. return x

// Insertion: insert a new element x in front of the list
List-Insert-Front(L, x)
1. x.next = L.nil.next
2. L.nil.next.prev = x            // O(1)
3. L.nil.next = x
4. x.prev = L.nil

// Deletion: remove an element x from the list.
//           Assume x is indeed in the list.
List-Delete(L, x)
1. x.prev.next = x.next
2. x.next.prev = x.prev            // O(1)
```

Sentinels rarely reduce the asymptotic time bounds of data structure operations but they can reduce constant factors. The gain from using sentinels is usually a matter of clarity of code rather than speed.

However, we should use sentinels judiciously. When there are many small lists, the extra storage used by their sentinels can represent significant wasted memory.

Recommended Exercises: 10.2-1, 10.2-2, 10.2-3, 10.2-4, 10.2-6, 10.2-7 (use circular lists with sentinels here!).