Chapter 10: Elementary Data Structures

Section 10.1.1: Arrays

Most languages store array as a contiguous sequence of bytes in memory and require each element to be of the same size. This allows $\Theta(1)$ access time to the ith element.

Section 10.1.2: Matrices

The mathematical notion of a **matrix** or a two-dimensional array can be represented by one or more one-dimensional arrays.Two of the most common ways to store a matrix are row-major and column-major order.

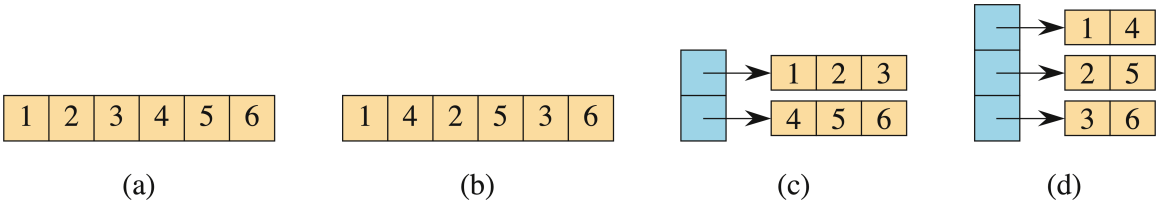
Row-Major order: A given $n \times n$ matrix is stored row by row in a one dimensional array. **Column-Major order**: A given $n \times n$ matrix is stored column by column in a one dimensional array.

We can also store matrices using multiple one-dimensional arrays (that is what Java does, for example). This is more flexible at the expense of a slight loss in access speed. For example, each row (or column) can be a different length so we can have ragged shapes.

For example, the 2×3 matrix:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

can be represented the following four ways:



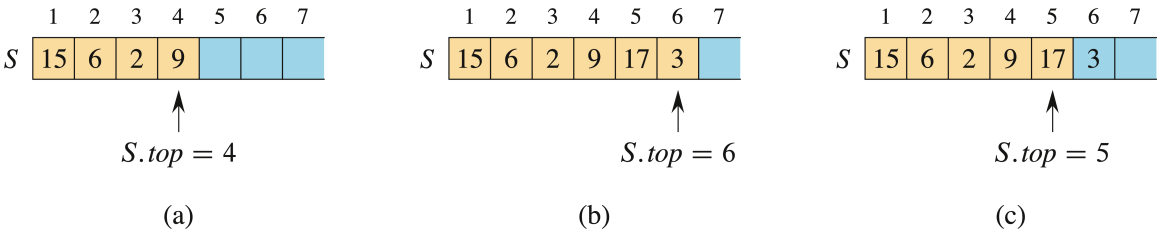
(a) (b) (c) (d)

Section 10.1.3: Stacks and Queues

- **Stack: Last-In, First-Out (LIFO) data structure.**
- Array representation for a stack
  - An array $S[1..n]$ is allocated to store elements. The size of the stack is $S.size = n$.

- An array attribute $S.top$ points to an array index in which the most recently inserted element resides. Initially, we set $S.top$ to 0.
- With respect to set operations, Insert operation is call PUSH, delete operation is called POP, STACK-EMPTY is a query operation.

```
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.size)
2.       error "overflow"                     O(1)
3. else S.top = S.top + 1
4.       S[S.top] = x

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

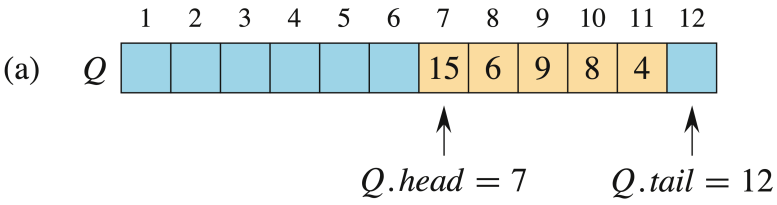


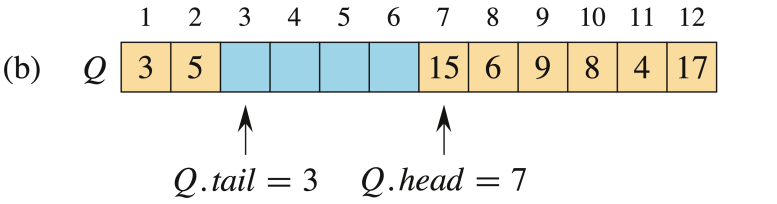
(a) (b) (c)

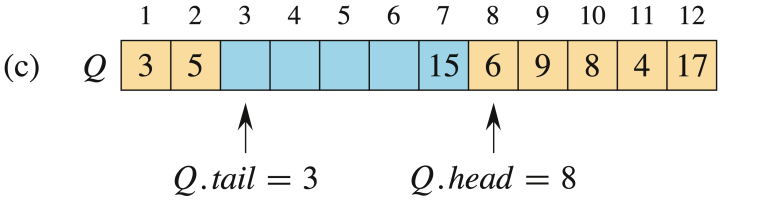
- **Queue: First-In, First-Out (FIFO) data structure.**
- A queue has a **head** and a **tail**.
- Array representation for a queue:
  - An array $Q[1..n]$ is allocated to store elements, so $Q.size = n$. The array will be considered as a circular array. It will store a queue with at most $n-1$ elements.
  - 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 $Q.tail = Q.size$

– An initial empty queue: $Q.head = 1$ and $Q.tail = 1$.

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

EnQueue(Q, x)       // stores at most n-1 elements
1. if (Q.head == ((Q.tail+1) mod Q.size))
2.      error "queue overflow"
3. else Q[Q.tail] = x                    O(1)
4.      if Q.tail == Q.size
5.           Q.tail = 1
6.       else Q.tail = Q.tail + 1

DeQueue(Q)
1. if Queue-Empty(Q)
2.      error "queue underflow"
3. else x = Q[Q.head]
4.      if Q.head == Q.size                    O(1)
5.           Q.head = 1
6.      else Q.head = Q.head + 1
7. return x
```





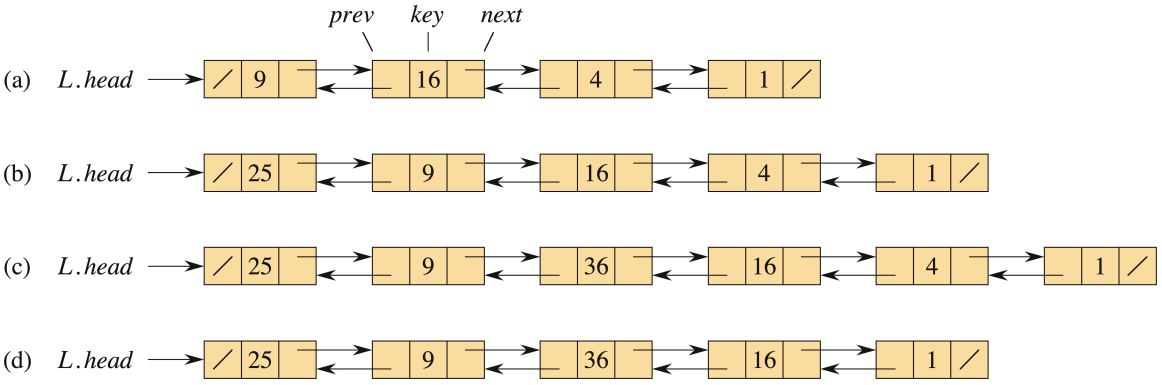


See `examples/circular-queue` in the CS321-resources repository for an iplementation.

Recommended Exercises: 10.1-2, 10.1-4, 10.1-6. **Challenging**: 10.1-7, 10.1-8.

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. `x.key` returns the key value of x.
  2. `x.next` return the pointer to the next element after x.
  3. `x.prev` return the pointer to the previous element before x.
- For the list L, an attribute `L.head` 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
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-Prepend(L, x)
```

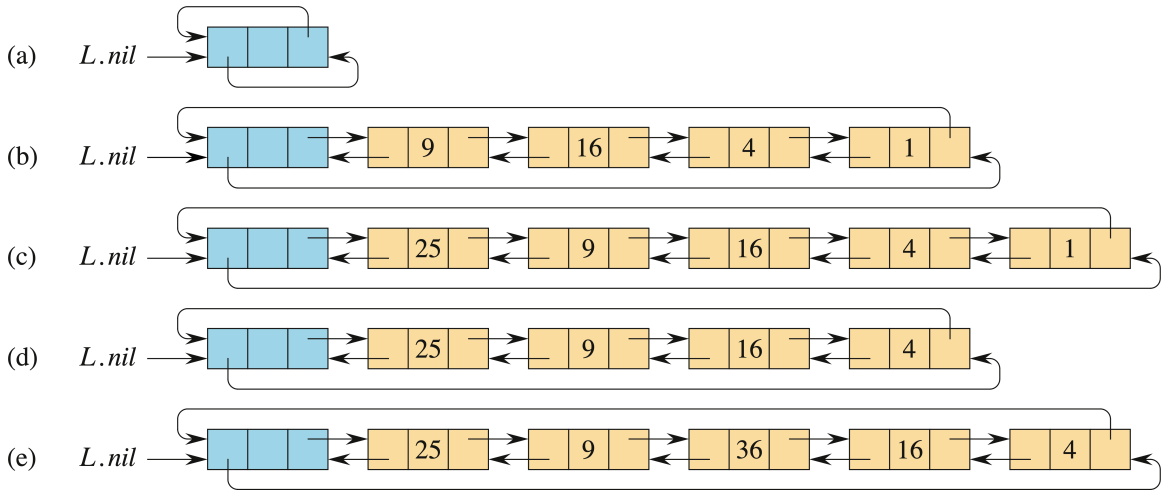
```
1. x.next = L.head
2. x.prev = NIL
2. if L.head != NIL
3.     L.head.prev = x            // O(1)
4. L.head = x

// Insertion: insert a new element x immediately after element y
List-Insert(x, y)
1. x.next = y.next
2. x.prev = y
2. if y.next != NIL
3.     y.next.prev = x            // O(1)
4. y.next = x

// 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.    x.prev.next = x.next
3. else L.head = x.next            // O(1)
4. if x.next != NIL
5.    x.next.prev = x.prev
```

To delete a given key k rather than a given node from a list, it requires us 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.
// Theta(n) in the worst-case

List-Search'(L, k)  // k is a key value
1. L.nil.key = k //store the key in sentinel to guarantee it is in the
2. x = L.nil.next //start at the head of the list
2. while x.key != k
3.     x = x.next
4. if x == L.nil  //found k in the sentinel
5.     return NIL //k was not really in the list
6. else return x

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

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

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, 10.2-5 (good one!).