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**, basically a two-dimensional array (or a table), can be represented by one or more one-dimensional arrays. Three of the most common ways to store a matrix are row-major, column-major order, or as multiple one-dimensional arrays.

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.
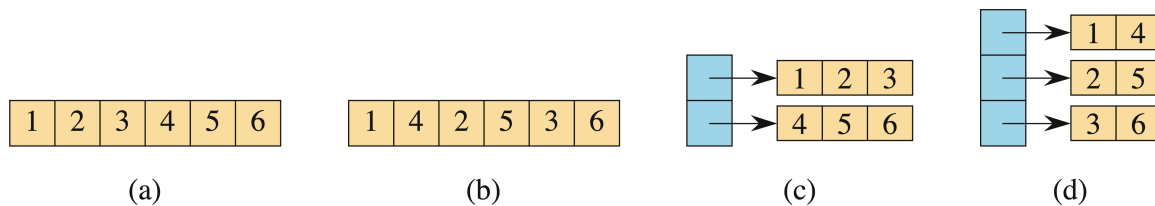
Array of arrays: We can also store matrices using multiple one-dimensional arrays (that is what Java does, for example). For example, each row (or column) can represented by a separate one-dimensional array with a separate array of pointers to connect them together. This representation is referred to as an array of arrays. 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.

Example: Let us show each of the above representation with an example matrix.

Consider the following 2×3 matrix:

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

It can be represented the following four ways. Here (a) shows a row-major order of the matrix in a one-dimensional array, (b) shows a column-major order of the matrix in a one-dimensional array, (c) shows each row represented by a separate one-dimensional array with a separate array of pointers to connect them together, and (d) shows the same with a column in each one-dimensional array.

Here is a Java code snippet that shows how to create a two-dimensional array as an array of arrays. Note that this array is 3×5 but we can make each one-dimensional array be of different size!

```java
Color[] matrix = new Color[3][];
for (int i = 0; i < matrix.length; i++)
    matrix[i] = new Color[5];

// Fill the matrix with some varying shades of gray
// Why are all the colors a shade of gray?
for (int i = 0; i < matrix.length; i++)
    for (int j = 0; j < matrix[i].length; j++)
        matrix[i][j] = new Color(i+j, i+j, i+j);

//How do we get shades of blue? of orange?
```

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 called PUSH, delete operation is called POP, STACK-EMPTY is a query operation.
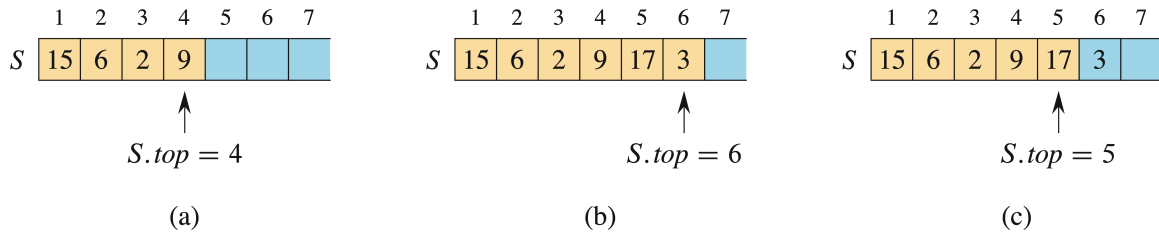
```
STACK-EMPTY(S)
1. if S.top == 0
2.      return TRUE
3. else return FALSE

PUSH(S, x)       // store at most n elements
1. if S.top == S.size
2.      error "overflow"
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
4.      return S[S.top + 1]
```

<table>
<tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
</table>

Each of the operations takes $\Theta(1)$ runtime in the worst case.

- **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
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
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
5.              Q.head = 1
6.      else Q.head = Q.head + 1
7. return x
```
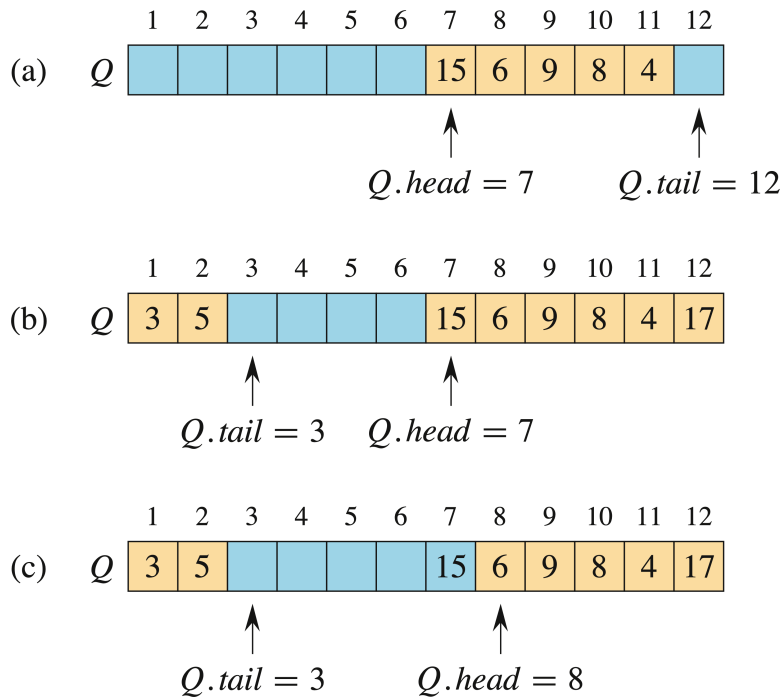
(a) Q

1 2 3 4 5 6 7 8 9 10 11 12

15 6 9 8 4

$Q.head = 7$ $Q.tail = 12$

(b) Q

1 2 3 4 5 6 7 8 9 10 11 12

3 5 15 6 9 8 4 17

$Q.tail = 3$ $Q.head = 7$

(c) Q

1 2 3 4 5 6 7 8 9 10 11 12
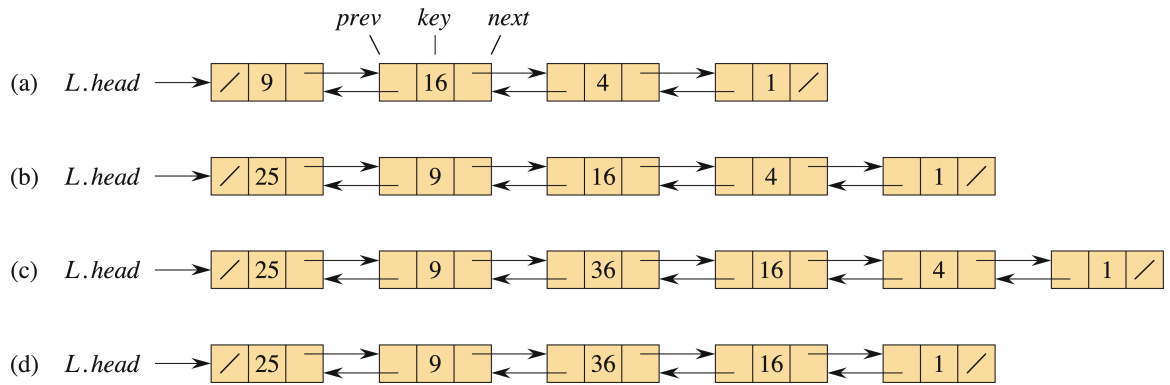
3 5 15 6 9 8 4 17

$Q.tail = 3$ $Q.head = 8$

Each of the queue operations takes $\Theta(1)$ worst-case runtime.

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

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 contains 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.

4

prev   key   next

(a) *L.head* → [/ | 9 | _] ⇄ [_ | 16 | _] ⇄ [_ | 4 | _] ⇄ [_ | 1 | /]

(b) *L.head* → [/ | 25 | _] ⇄ [_ | 9 | _] ⇄ [_ | 16 | _] ⇄ [_ | 4 | _] ⇄ [_ | 1 | /]

(c) *L.head* → [/ | 25 | _] ⇄ [_ | 9 | _] ⇄ [_ | 36 | _] ⇄ [_ | 16 | _] ⇄ [_ | 4 | _] ⇄ [_ | 1 | /]

(d) *L.head* → [/ | 25 | _] ⇄ [_ | 9 | _] ⇄ [_ | 36 | _] ⇄ [_ | 16 | _] ⇄ [_ | 1 | /]

For the following procedures, we will use **doubly, unsorted and non-circular** linked lists.

```
// Searching a linked list:
// Walk 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
3.         x = x.next
4. return x
```

List-Search operation takes $\Theta(n)$ worst-case runtime.

```
// 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
4. L.head = x
```

LIST-PREPEND operation takes $\Theta(1)$ worst-case runtime.

```
// 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
4. y.next = x
```

LIST-INSERT operation takes $\Theta(1)$ worst-case runtime.
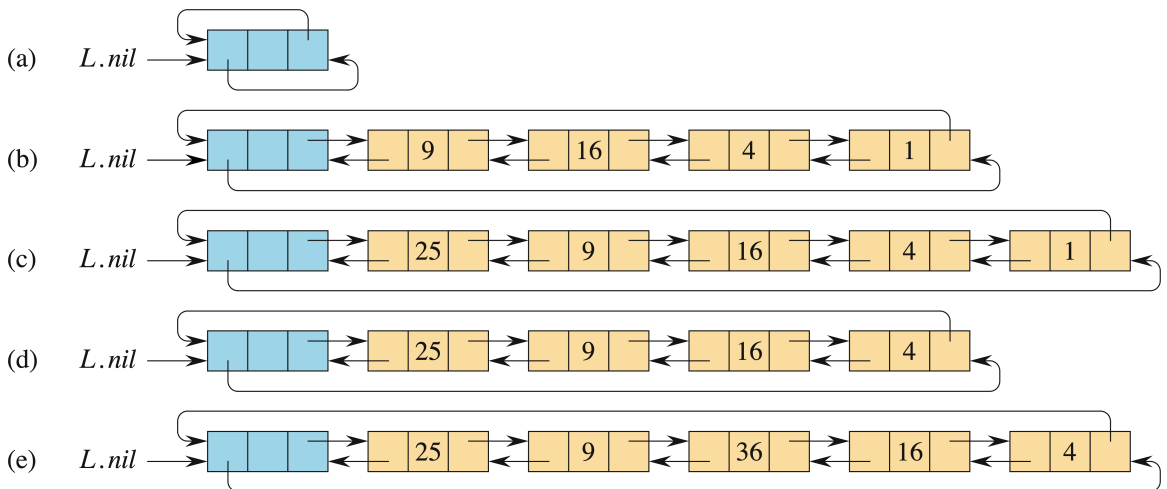
```
// Deletion: remove an element x from the list.
//           Assume x is indeed in the list.
LIST-DELETE(L, x)
1. if x.prev != NIL
2.     x.prev.next = x.next
3. else L.head = x.next
4. if x.next != NIL
5.     x.next.prev = x.prev
```

LIST-DELETE operation takes $\Theta(1)$ worst-case runtime.

To delete an element with 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). In this case, the delete takes $\Theta(n)$ runtime in the worst-case.

Sentinels: Using sentinels to simplify the linked list procedures. A sentinel is an extra element that acts as a guard to simplify loops and conditions. 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 and the head)

```
// Searching a linked list:
// Walk 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. L.nil.key = k //store the key in sentinel to
                 //guarantee it is in the list
2. x = L.nil.next //start at the head of the list
3. while x.key != k
4.      x = x.next
5. if x == L.nil   //found k in the sentinel
6.      return NIL //k was not really in the list
7. else return x


// Insertion: insert a new element x following object y
// Insert in front: Make y be L.nil
// Insert at end: Make y be L.nil.prev
LIST-INSERT'(x, y)
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.
//           Assume x is indeed in the list.
LIST-DELETE'(L, x)
1. x.prev.next = x.next
2. x.next.prev = x.prev
```

The worst-case run-times are the same as above for linked lists without sentinels. 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!).