Elementary Data Structures

Devere Anthony Weaver

Introduction

Sets are fundamental structures in computer science and **dynamic sets** for the basis for many useful data structures. Dynamic sets support a number of operations that are usually placed in two groups:

1. queries - operations which return information about the set
2. modifying - operations that change the set

Often, specific applications only require the implementation of only a few of these operations.

Section III of "CLRS" covers some important data structures and set operations that are useful in the study of computer science as well as the implementation of computer programs.

Simple array-based data structures: arrays, matrices, stacks, queues

Arrays

When it comes to standard static arrays, we'll assume they are stored in contiguous memory (as implemented in many C-style programming languages).

If (i) the first element of an array has an index s, (ii) the array starts at memory address a, and (iii) each array element occupies b bytes of memory, then the i^{th} element of the array occupies bytes $a + b(i - s)$ through $a + b(i - s + 1) - 1$ in memory.

Another assumption based on our Word-RAM model of computation is that each element can be accessed randomly thus it takes constant time to access any array element, regardless of the index.

If the elements of a given array are of different sizes, then the above formulas don't hold and we can assume the elements in each are actualized as pointers to objects.

Matrices

Matrices are mathematical structures we're very familiar with from mathematics. Consider an $m \times n$ matrix where $m = \#$ rows and $n = \#$ cols. Matrices stores in **row-major order** store the elements of the matrix row by row and matrices stored in **column-major order** store elements column by column.

Matrices can be represented with single-array representations and multiple-array representations. Typically on modern systems, single-array representations are more efficient. Another common representation scheme is to store matrices using block representation. In block representation, a given matrix is divided into block and then each block, or sub-matrix, is stored contiguously (see p254 of "CLRS" for a visualization).

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, i.e. **last-in, first-out** (LIFO). In a **queue**, the element deleted is always the one that has been in the set for the longest, i.e. **first-in, first-out**, (FIFO).

For a stack, the INSERT operation is also known as PUSH and the DELETE operation is also known as POP. Common attributes for this data structure are *size* and *top* which contain data on the size of an array-based stack and the top of the array-based stack. The following are common stack operations and they take $O(1)$ time, where S is the stack:

- PUSH(S, x) - inserts x on top of the stack
- POP(S) - returns and removes the item from the top of the stack
- PEEK(S) - returns but does not remove item at top of the stack
- ISEMPTY(S) - returns true if the stack has no items
- GETLENGTH(S) - returns the number of items on the stack

Something to keep in mind is that POP and PEEK should NOT be applied to an empty stack, this behavior is undefined. Be sure to check if the stack is empty upon implementation to handle this possibility.

Stacks as an ADT can be implemented using a standard fixed array, a linked list, or a vector. If implemented using a linked list, the head of the list represents the top of the stack. A PUSH operation is performed by creating a new list node and then prepending it to the list. A POP is performed by assigning a local variable with the head node's data, removed the head node from the list, and then returning the local variable.

NOTE: It may be helpful to have a tail attribute for the stack to keep track of the end.

An **unbounded stack** is a stack with no upper limit on the length (this doesn't work for physical machines). A **bounded stack** is a stack with a length that doesn't exceed a maximum value (a bit more realistic). When implementing a bounded stack with an array, there tends to be two different ways:

1. allocate the array to the maximum length upon construction
2. distinguish between allocation size and max length, reallocate as needed until max length is reached

One major difference between these two implementations is the worse-case for the PUSH operation. For the first implementation, the PUSH operation is $O(1)$ since the entire array is allocated at once and never resized. The second implementation can cause the array to reallocate which can result in $O(n)$ time. For both implementations, POP executes in constant-time $O(1)$.

For queue, the INSERT operation is known as ENQUEUE and the DELETE operation is known as DEQUEUE. Since stacks and queues have a pre-specified deletion element, there are no arguments to the DELETE operation. The most common attributes for array-based queues are:

- *size (allocationSize)* - equals the size of the array-based queue
- *head (frontIndex)* - indexes to the head (front or first element) of the queue
- *tail (tailIndex)* - indexes the next location at which a newly arriving element will be inserted into the queue
- *length* - an integer for the number of items in the queue

Some of the common queue operations are, where Q is the queue data structure:

- ENQUEUE(Q, x) - inserts x at the end of the queue, ($O(n)$)

- DEQUEUE(Q) - returns and removes an item at the front of the queue, ($O(1)$)
- PEEK(Q) - returns but does not remove the item at the front of the queue
- ISEMPTY(Q) - returns true if queue has no items
- GETLENGTH(Q) - returns the number of items in the queue

The head of a queue doesn't necessarily need to be the first element since the location of the head will move when an element in the queue is DEQUEUE'd. Similarly, the tail will move whenever a new element is ENQUEUE'd. Attempting to ENQUEUE a queue that is full will cause overflow and attempting to DEQUEUE a queue that is empty will cause underflow. These conditions are met when:

- empty queue - occurs when the $head = tail$
- full queue - occurs when (i) $head = tail + 1$ or (ii) $head = 1$ and $tail = size$

Similar to a stack, a queue can be implemented using an array or a linked list. If using a linked list implementation, the head node represents the front of the queue and the list's tail node represents the queue's end.

Thus the ENQUEUE operation is performed by creating a new list node, assigning the data with the item, and then appending it to the list. This entails having to point the previous last node's next member to this newly created node and the ENQUEUE'd node's next member to null.

Likewise, the DEQUEUE operation simply stores the head node's data member value in a local variable, deletes the node itself, and then returns the local variables that is storing the deleted node's data.

A **bounded queue** is a queue with a length that doesn't exceed a specified maximum value. It typically is given an additional attribute that contains information on the maximum possible length for this value, e.g. *maxLength*. Naturally a bounded queue is full when the length of the queue is equal to the maxLength attribute. Similarly an **unbounded queue** is a queue that can grow indefinitely (impractical mostly, maybe some theoretical uses).

For both stacks and queues, we can create a single implementation that supports both bounded and unbounded versions of each structure. Consult the implementation code in my github repos as needed.

A **deque** (double-ended queue) is an ADT in which items can be inserted and removed at both the front and the back of the queue. Like a normal queue, deque can be implemented via a static array or a linked list.

Some common deque ADT operations include the following, where D is a deque:

- PUSHFRONT(D, x) - inserts x at the front of the queue
- PUSHBACK(D, x) - inserts x at the back of the queue
- POPFRONT(D) - returns and removes item at the front of queue
- POPBACK(D) - returns and removes item at the back of queue
- PEEKFRONT(D) - returns item at the front of the queue
- PEEKBACK(D) - returns item at the back of the queue
- ISEMPTY(D) - returns true is the deque is empty
- GETLENGTH(D) - returns the number of items in deque

Linked Lists

A **linked list** is a data structure in which the objects are arranged in a linear order. The order in a linked list is determined by a pointer in each object. In addition to pointer(s), each node in the list also often contains satellite data. A linked list also has an attribute that points to the head of the linked list (the first element in the list).

There are a few variations of linked lists. Some common ones include:

- doubly linked list - each object in the list contains a pointer to its successor (next) and its predecessor (previous) objects
- singly linked list - each object in the list contains a pointer only to its successor (next)
- sorted - the linear order of the list depends on the order of the keys
- unsorted - elements are stored in any order, not based on the keys
- circular list - the previous pointer to the head of the list points to the tail and the next pointer of the tail points to the head (ring)

Common list ADT operations

- APPEND(L, x) - inserts x at the end of the list
- LIST-PREPEND(L,x) (a.k.a PREPEND) - inserts x at the start of the list
- LIST-INSERT(x, y) - inserts elements x after element y is a list
- PRINT(L) - prints a list's items in order
- LIST-SEARCH(L, x) - search a list L for a given key x
- LIST-DELETE(L, x) (REMOVE) - delete element x from list L
- PRINT-REVERSE(L) - prints a list's items in ascending order
- SORT(L) - sorts the lists items in ascending order
- IS-EMPTY(L) - returns true if the list has no items
- GET-LENGTH(L) - returns the number of items in the list

Searching a linked list

The LIST-SEARCH(L, k) operation finds the first element with key k in the list L by a simple linear search, returning a pointer to the element. If the operation doesn't find the key in the list, then it returns a pointer to NULL. The worst-case runtime is $\Theta(n)$, since LIST-SEARCH is simply a linear-search and it may have to search the entire list before finding or not finding the key.

Inserting into a linked list

There are two general ways to insert new elements into a linked list. The two operations defined here are:

- LIST-PREPEND(L, x) - prepends element x to the list L; $O(1)$
- LIST-INSERT(x, y) - inserts element x after element y in the list; $O(1)$

Deleting from a linked list

- LIST-DELETE(L, x) - removes an element x from a linked list L; $O(1)$

While LIST-DELETE itself runs in $O(1)$ time, it may need to call LIST-SEARCH to get a pointer to the element that needs to be deleted which is $\Theta(n)$. So, the worst-case runtime will depend on if the implementation makes a call to LIST-SEARCH itself or if LIST-SEARCH is called before LIST-DELETE.

Sentinels

A **sentinel** is a dummy object that allows us to simplify boundary conditions. For the example of a linked list, the sentinel is the object $L.null$ which represents NULL but is not actually NULL itself. Instead, this sentinel is actually an element of the linked list with the same attributes as the rest of the elements in the list. Thus, $L.null.next$ points to the head of the list and $L.null.prev$ points to the tail. When utilized it creates a **circular, doubly linked list with a sentinel**, a.k.a a mouthful.

If a sentinel element is used in a a linked list, then there is no need for a $L.head$ attribute for the list. One benefit to using a sentinel element is that it simplifies the implementation details for LIST-DELETE and LIST-INSERT and it can reduce their worst-case runtime by a constant factor; however, their use doesn't typically improve asymptotic running time. In fact, if our application utilizes many small lists, sentinels can waste memory. Consider only using them if there is a good reason to justify its existence.

As a note, other DSA resources may refer to a sentinel as a **dummy node** when dealing with linked lists.