Chapter 7: Queues and Deques

After the stack, the next simplest data abstraction is the *queue*. As with the stack, the queue can be visualized with many examples you are already familiar with from everyday life. A simple illustration is a line of people waiting to enter a theater. The fundamental property of the queue is that items are inserted at one end (the rear of the line) and removed from the other (the door to the theater). This means that the order that items are removed matches the order that they are inserted. Just as a stack was described as a LIFO (last-in, first-out) container, this means a queue can be described as FIFO (first in, first out).

A variation is termed the *deque*, pronounced "deck", which stands for *d*ouble-*e*nded *que*ue. In a deque values can be inserted at *either* the front or the back, and similarly the deck allows values to be removed from either the front or the back. A collection of peas in a straw is a useful mental image. (Or, to update the visual image, a series of tapioca buds in a bubble-tea straw).

Queues and deques are used in a number of ways in computer applications. A printer, for example, can only print one job at a time. During the time it is printing there may be many different requests for other output to be printed. To handle these the printer will maintain a queue of pending print tasks. Since you want the results to be produced in the order that they are received, a queue is the appropriate data structure.

The Queue ADT specification

The classic definition of the queue abstraction as an ADT includes the following operations:

addBack (newElement)	Insert a value into the queue
front()	Return the front (first) element in queue
removeFront()	Remove the front (first) element in queue
isEmpty()	Determine whether the queue has any elements

The deque will add the following:

addFront(newElement)	Insert a value at front of deque
back()	Return the last element in the queue
removeBack()	Return the last element in the queue

Notice that the queue is really just a special case of the deque. Any implementation of the deque will also work as an implementation of the queue. (A deque can also be used to implement a stack, a topic we will explore in the exercises).

As with the stack, it is the FIFO (first in, first out) property that is, in the end, the fundamental defining characteristic of the queue, and not the names of the operations. Some designers choose to use names such as “add”, “push” or “insert”, leaving the location unspecified. Similarly some implementations elect to make a single operation that will both return and remove an element, while other implementations separate these two tasks, as well do here. And finally, as in the stack, there is the question of what the effect should be if an attempt is made to access or remove an element from an empty collection. The most common solutions are to throw an exception or an assertion error (which is what we will do), or to return a special value, such as Null.

For a deque the defining property is that elements can only be added or removed from the end points. It is not possible to add or remove values from the middle of the collection.

The following table shows the names of deque and queue operations in various programming languages:

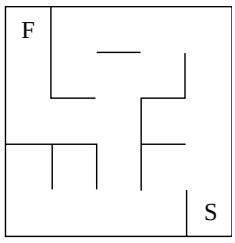
operation	C++	Java	Perl	Pyton
insert at front	push_front	addFirst	unshift	appendleft
Insert at back	Push_back	addLast	Push	append
remove last	pop_back	removeLast	pop	pop
remove first	pop_front	removeFirst	shift	popleft
examine last	back	getLast	$_[-1]	deque[-1]
examine first	front	getFirst	$_[0]	deque[0]

Applications of Queues

Given that queues occur frequently in real life, it is not surprising that they are also frequently used in simulations. To model customers using a bank, for example, you could use a queue of waiting patrons. A simulation might want to ask questions such as the how the average waiting time would change if you added or removed a teller position.

Queues are also used in any time collection where time of insertion is important. We have noted, for example, that a printer might want to keep the collection of pending jobs in a queue, so that they will be printed in the same order that they were submitted.

Depth-first and Breadth-first search

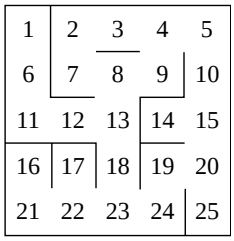


Imagine you are searching a maze, such as the one shown at right. The goal of the search is to move from the square marked S, to the square marked F.

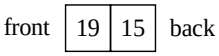
A simple algorithm would be the following:

How to search a maze:
Keep a list of squares you have visited, initially empty.
Keep a stack of squares you have yet to visit. Put the starting square in this stack
While the stack is not empty:
    Remove an element from the stack
    If it is the finish, then we are done
    Otherwise if you have already visited this square, ignore it
    Otherwise, mark the square on your list of visited positions, and
    add all the neighbors of this square to your stack.
If you eventually reach an empty stack and have not found the start, there is no solution

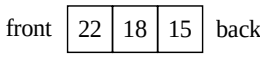
To see the working of this algorithm in action, let us number of states of our maze from 1 to 25, as shown. There is only one cell reachable from the starting position, and thus after the first step the queue contains only one element:



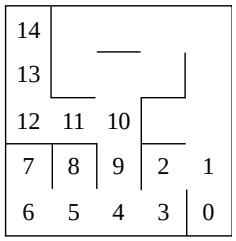
This value is pulled from the stack, and the neighbors of the cell are inserted back. This time there are two neighbors, and so the stack will have two entries.



Only one position can be explored at any time, and so the first element is removed from the stack, leaving the second waiting for later exploration. Two steps later we again have a choice, and both neighbors are inserted into the stack. At this point, the stack has the following contents:



The solution is ultimately found in fifteen steps. The following shows the path to the solution, with the cells numbered in the order in which they were considered.



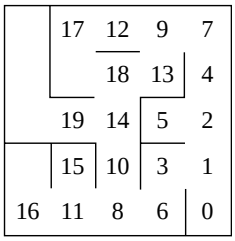
The strategy embodied in this code doggedly pursues a single path until it either reaches a dead end or until the solution is found. When a dead end is encountered, the most recent alternative path is reactivated, and the search continues. This approach is called a *depth-first search*, because it moves deeply into the structure before examining alternatives. A depth-first search is the type of search a single individual might perform in walking through a maze.

Suppose, on the other hand, that there were a group of people walking together. When a choice of alternatives was encountered, the group might decide to split itself into smaller groups, and explore each alternative simultaneously. In this fashion all potential paths are investigated at the same time. Such a strategy is known as a *breadth-first* search.

What is intriguing about the maze-searching algorithm is that the exact same algorithm can be used for both, changing only the underlying data structure. Imagine that we change the stack in the algorithm to a queue:

> **How to search a maze using breadth-first search**:
> Keep a list of squares you have visited, initially empty.
> Keep a queue of squares you have yet to visit. Put the starting square in this queue
> While the queue is not empty:
> Remove an element from the queue
> If it is the finish, then we are done
> Otherwise if you have already visited this square, ignore it
> Otherwise, mark the square on your list of visited positions, and
> add all the neighbors of this queue to your stack.
> If you eventually reach an empty queue and have not found the start, there is no solution

As you might expect, a breadth-first search is more thorough, but may require more time than a depth-first search. While the depth-first search was able to find the solution in 15 steps, the breadth-first search is still looking after 20. The following shows the search at this point. Trace with your finger the sequence of steps as they are visited. Notice how the search jumps all over the maze, exploring a number of different alternatives at the same time. Another way to imagine a breadth-first first is as what would happen if ink were poured into the maze at the starting location, and slowly permeates every path until the solution is reached.



We will return to a discussion of depth-first and breadth-first search in a later chapter, after we have developed a number of other data structures, such as graphs, that are useful in the representation of this problem.

Queue Implementation Techniques

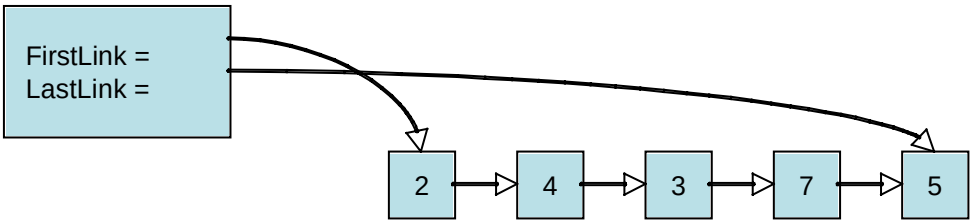
As with the stack, the two most common implementation techniques for a queue are to use a linked list or to use an array. In the worksheets you will explore both of these alternatives.

Worksheet 18	Linked List Queue, Introduction to Sentinels
Worksheet 19	Linked List Deque
Worksheet 20	Dynamic Array Queue

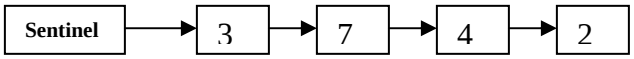
Building a Linked List Queue

A stack only needs to maintain a link to one end of the chain of values, since both insertions and removals occur on the same side. A queue, on the other hand, performs

insertions on one side and removals from the other. Therefore it is necessary to maintain a links to both the front and the back of the collection.



We will add another variation to our container. A *sentinel* is a special link, one that does not contain a value. The sentinel is used to mark either the beginning or end of a chain of links. In our case we will use a sentinel at the front. This is sometimes termed a *list header*. The presence of the sentinel makes it easier to handle special cases. For example, the list of links is never actually empty, even when it is logically empty, since there is always at least one link (namely, the sentinel). A new value is inserted after the end, after the element pointed to by the last link field. Afterwards, the last link is updated to refer to the newly inserted value.

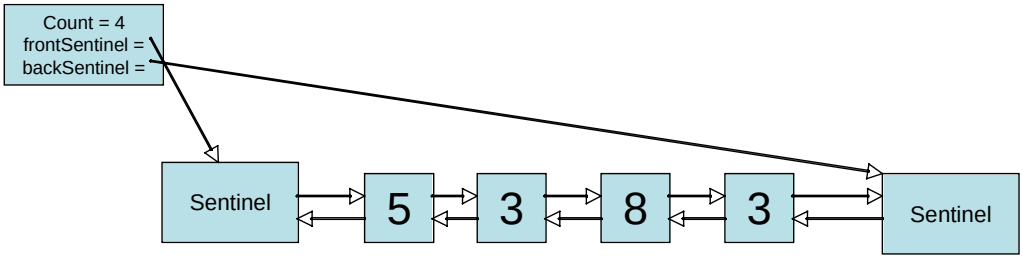


Values are removed from the front, as with the stack. But because of the sentinel, these will be the element right after the sentinel.

You will explore the implementation of the queue using linked list techniques in Worksheet 18.

A Linked List Deque – using Double Links

You may have noticed that removal from the end of a ListQueue is difficult because with only a single link it is difficult to “back up”. That is, while you have a pointer to the end sentinel, you do not have an easy way to back up and find the link immediately preceding the sentinel.



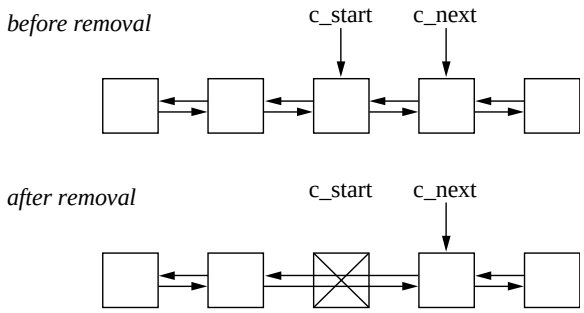
One solution to this problem is to use a *doubly-linked* list. In a doubly-linked list, each link maintains two pointers. One link, the forward link, points forward toward the next link in the chain. The other link, the prev link, points backwards towards the previous

element. Anticipating future applications, we now also keep a count of the number of elements in the list.

With this picture, it is now easy to move either forward or backwards from any link. We will use this new ability to create a linked list deque. In order to simplify the implementation, we will this time include sentinels at both the beginning and the end of the chain of links. Because of the sentinels, both adding to the front and adding to the end of a collection can be viewed as special cases of a more general "add to the middle of a list" operation. That is, perform an insertion such as the following:

picture

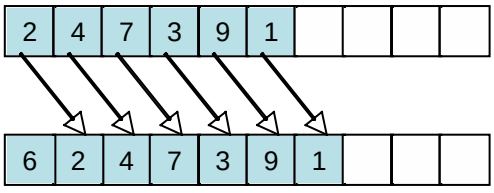
Similarly, removing a value (from other the front or the back) is a special case of a more general remove operation:



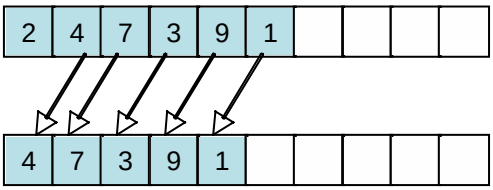
In worksheet 19 you will complete the implementation of the list deque based on these ideas.

A Dynamic Array Deque

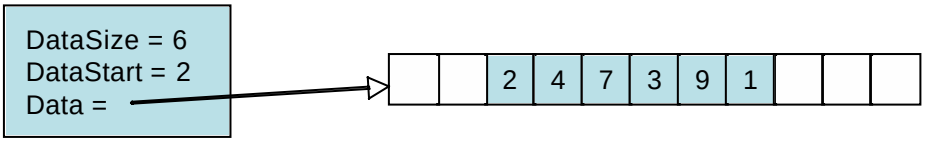
The dynamic array that we used in implementing the ArrayStack will not work for either a queue or a deque. The reason is that inserting an element at the front of the array requires moving every element over by one position. A loop to perform this movement would require O(n) steps, far too slow for our purposes.



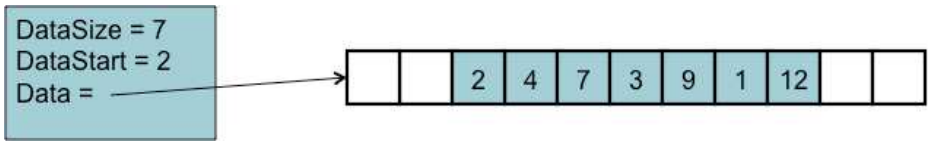
If we tried adding elements to the end (as we did with the stack) then the problem is reversed. Now it is the remove operation that is slow, since it requires moving all values down by one position



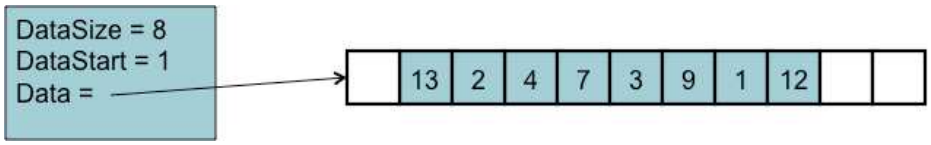
The root of the problem is that we have fixed the start of the collection at array index position zero. If we simply loosen that requirement, then things become much easier. Now in addition to the data array our collection will maintain two integer data fields; a size (as before) and a starting location.



Adding a value to the end is simlar to the ArrayStack. Simply increase the size, and place the new element at the end.

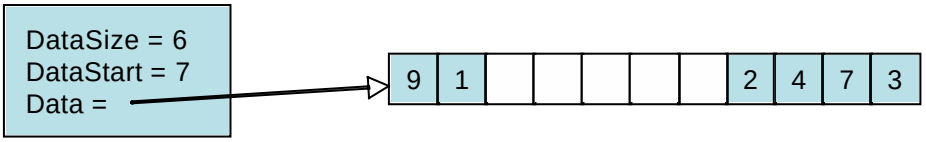


But adding a value to the front is also simple. Simply decrement the starting point by one, then add the new element to the front:



Removing elements undo these operations. As before, if an insertion is performed when the size is equal to the capacity, the array must be doubled in capacity, and the values copied into the new array.

There is just one problem. Nothing prevents the data values from wrapping around from the upper part of the array to the lower:



To accommodate index values must be computed carefully. When an index wraps around the end of the array it must be altered to point to the start of the array. This is easily done by subtracting the capacity of the array. That is, suppose we try to index the fifth element in the picture above. We start by adding the index, 5, to the starting location, 7. The resulting sum is 12. But there are only eleven values in the collection. Subtracting 11 from 12 yields 1. This is the index for the value we seek.

In worksheet 20 you will explore the implementation of the dynamic array deque constructed using these ideas. A deque implemented in this fashion is sometimes termed a *circular buffer*, since the right hand side circles around to begin again on the left.

Self Study Questions

1. What are the defining characteristics of the queue ADT?

2. What do the letters in FIFO represent? How does this describe the queue?

3. What does the term deque stand for?

4. How is the deque ADT different from the queue abstraction?

5. What will happen if an attempt is made to remove a value from an empty queue?

6. What does it mean to perform a depth-first search? What data structure is used in performing a depth-first search?

7. How is a breadth-first search different from a depth-first search? What data structure is used in performing a breadth-first search?

8. What is a sentinel in a linked list?

9. What does it mean to say that a list is singly-linked?

10. How is a doubly-linked list different from a singly-linked list? What new ability does the doubly-linked feature allow?

11. Why is it difficult to implement a deque using the same dynamic array implementation that was used in the dynamic array stack?

Short Exercises

1. Show the state of a deque after the following sequence of operations: [more]

Analysis Exercises

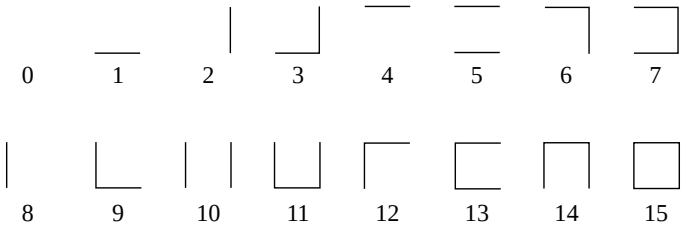
1. A deque can be used as either a stack or a queue. Do you think that it is faster or slower in execution time than either a dynamic array stack or a linked list stack? Can you design an exercise to test your hypothesis? In using a Deque as a stack there are two choices; you can either add and remove from the front, or add and remove from the back. Is there a measurable difference in execution time between these two alternatives?

Programming Assignments

1. Many computer applications present practice drills on a subject, such as arithmetic addition. Often such systems will present the user with a series of problems and keep track of those the user answered incorrectly. At the end of the session, the incorrect problems will then be presented to the user a second time. Implement a practice drill system for addition problems having this behavior.

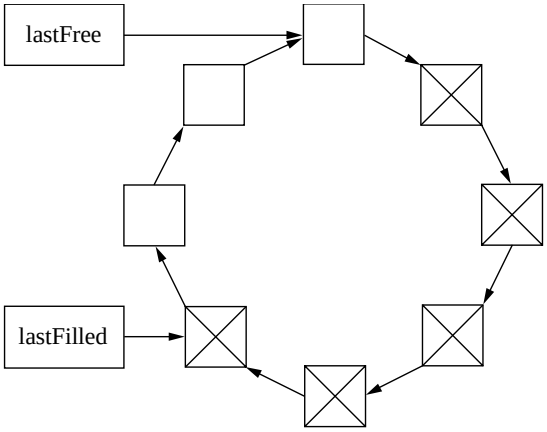
2. FollowMe is a popular video game. The computer display a sequence of values, and then asks the player to reproduce the same sequence. Points are scores if the entire sequence was produced in order. In implementing this game, we can use a queue to store the sequence for the period of time between generation by the computer and response by the player.

3. To create a maze-searching program you first need some way to represent the maze. A simple approach is to use a two-dimensional integer array. The values in this array can represent the type of room, as shown below. The values in this array can then tell you the way in which it is legal to move. Positions in the maze can be represented by (I,j) pairs. Use this technique to write a program that reads a maze description, and prints a sequence of moves that are explored in either a depth-first or breadth-first search of the maze.



4. In Chapter 5 you learned that a *Bag* was a data structure characterized by the following operations: add an element to the collection, test to see if an element is in the collection, and remove an element from the collection. We can build a bag using the same linked list techniques you used for the linked list queue. The add operation is the same as the queue. To test an element, simply loop over the links, examining each in turn. The only difficult operation is remove, since to remove a link you need access to the immediately preceding link. To implement this, one approach is to loop over the links using a pair of pointers. One pointer will reference the current link, while the second will always reference the immediate predecessor (or the sentinel, if the current link is the first element). That way, when you find the link to remove, you simply update the predecessor link. Implement the three bag operations using this approach. Does the use of the sentinel make this code easier to understand than the equivalent code was for the stack version?

5. Another implementation technique for a linked list queue is to link the end of the queue to the front. This is termed a circular buffer. Two pointers then provide access to the last filled position, and the last free position. To place an element into the queue, the last filled pointer is advanced, and the value stored. To remove a value the last free pointer is advanced, and the value returned. The following picture shows this structure. How do you know when the queue is completely filled? What action should you take to increase the size of the structure in this situation? Implement a circular buffer queue based on these ideas.



On The Web

Wikipedia has well written entries for queue, deque, FIFO, as well as on the implementation structures dynamic array, linked list and sentinel node. The NIST *Dictionary of Algorithms and Data Structures* also has a good explanation.