Double-ended queue

In computer science, a **double-ended queue** (abbreviated to **deque**) is an abstract data type that generalizes a queue, for which elements can be added to or removed from either the front (head) or back (tail).[1] It is also often called a **head-tail linked list**, though properly this refers to a specific data structure *implementation* of a deque (see below).

Contents

Naming conventions

Deque is sometimes written *dequeue*, but this use is generally deprecated in technical literature or technical writing because *dequeue* is also a verb meaning "to remove from a queue". Nevertheless, several libraries and some writers, such as Aho, Hopcroft, and Ullman in their textbook *Data Structures and Algorithms*, spell it *dequeue*. John Mitchell, author of *Concepts in Programming Languages,* also uses this terminology.

Distinctions and sub-types

This differs from the queue abstract data type or *first in first out* list (FIFO), where elements can only be added to one end and removed from the other. This general data class has some possible sub-types:

- An input-restricted deque is one where deletion can be made from both ends, but insertion can be made at one end only.
- An output-restricted deque is one where insertion can be made at both ends, but deletion can be made from one end only.

Both the basic and most common list types in computing, queues and stacks can be considered specializations of deques, and can be implemented using deques.

Operations

The basic operations on a deque are *enqueue* and *dequeue* on either end. Also generally implemented are *peek* operations, which return the value at that end without dequeuing it.

Names vary between languages; major implementations include:

operation	common name(s)	Ada	C++	Java	Perl	PHP	Python	Ruby	JavaScript
insert element at back	inject, snoc, push	`Append`	`push_back`	`offerLast`	`push`	`array_push`	`append`	`push`	`push`
insert element at front	push, cons	`Prepend`	`push_front`	`offerFirst`	`unshift`	`array_unshift`	`appendleft`	`unshift`	`unshift`
remove last element	eject	`Delete_Last`	`pop_back`	`pollLast`	`pop`	`array_pop`	`pop`	`pop`	`pop`
remove first element	pop	`Delete_First`	`pop_front`	`pollFirst`	`shift`	`array_shift`	`popleft`	`shift`	`shift`
examine last element	peek	`Last_Element`	`back`	`peekLast`	`$array[-1]`	`end`	`<obj>[-1]`	`last`	`<obj>[<obj>.length - 1]`
examine first element		`First_Element`	`front`	`peekFirst`	`$array[0]`	`reset`	`<obj>[0]`	`first`	`<obj>[0]`

Implementations

There are at least two common ways to efficiently implement a deque: with a modified dynamic array or with a doubly linked list.

The dynamic array approach uses a variant of a dynamic array that can grow from both ends, sometimes called **array deques**. These array deques have all the properties of a dynamic array, such as constant-time random access, good locality of reference, and inefficient insertion/removal in the middle, with the addition of amortized constant-time insertion/removal at both ends, instead of just one end. Three common implementations include:

- Storing deque contents in a circular buffer, and only resizing when the buffer becomes full. This decreases the frequency of resizings.
- Allocating deque contents from the center of the underlying array, and resizing the underlying array when either end is reached. This approach may require more frequent resizings and waste more space, particularly when elements are only inserted at one end.
- Storing contents in multiple smaller arrays, allocating additional arrays at the beginning or end as needed. Indexing is implemented by keeping a dynamic array containing pointers to each of the smaller arrays.

Purely functional implementation

Double-ended queues can also be implemented as a purely functional data structure.[2] Two versions of the implementation exist. The first one, called '*real-time deque*, is presented below. It allows the queue to be persistent with operations in $O(1)$ worst-case time, but requires lazy lists with memoization. The second one, with no lazy lists nor memoization is presented at the end of the sections. Its amortized time is $O(1)$ if the persistency is not used; but the worst-time complexity of an operation is $O(n)$ where n is the number of elements in the double-ended queue.

Let us recall that, for a list `l`, `|l|` denotes its length, that `NIL` represents an empty list and `CONS(h,t)` represents the list whose head is `h` and whose tail is `t`. The functions `drop(i,l)` and `take(i,l)` return the list `l` without its first `i` elements, and the `i`'s first elements respectively. Or, if `|l| < i`, they return the empty list and `l` respectively.

A double-ended queue is represented as a sixtuple `lenf,f,sf,lenr,r,sr` where `f` is a linked list which contains the front of the queue of length `lenf`. Similarly, `r` is a linked list which represents the reverse of the rear of the queue, of length `lenr`. Furthermore, it is assured that `|f| <= 2|r|+1` and `|r| <= 2|f|+1` - intuitively, it means that neither the front nor the rear contains more than a third of the list plus one element. Finally, `sf` and `sr` are tails of `f` and of `r`, they allow to schedule the moment where some lazy operations are forced. Note that, when a double-ended queue contains `n` elements in the front list and `n` elements in the rear list, then the inequality invariant remains satisfied after `i` insertions and `d` deletions when `(i+d)/2 <= n`. That is, at most `n/2` operations can happen between each rebalancing.

Intuitively, inserting an element `x` in front of the double-ended queue `lenf, f, sf, lenr, sr` leads almost to the double-ended queue `lenf+1, CONS(x,f), drop(2,sf), lenr, r, drop(2,sr)`, the head and the tail of the double-ended queue `lenf, CONS(x,f), sf, lenr, r, sr` are `x` and almost `lenf-1, f, drop(2,sf), lenr, r, drop(2,sr)` respectively, and the head and the tail of `lenf, NIL, NIL, lenr, CONS(x,NIL), drop(2,sr)` are `x` and `0, NIL, NIL, 0, NIL, NIL` respectively. The function to insert an element in the rear, or to drop the last element of the double-ended queue, are similar to the above function which deal with the front of the double-ended queue. It is said *almost* because, after insertion and after an application of *tail*, the invariant `|r| <= 2|f|+1` may not be satisfied anymore. In this case it is required to rebalance the double-ended queue.

In order to avoid an operation with an `O(n)` costs, the algorithm uses laziness with memoization, and force the rebalancing to be partly done during the following `(|l| + |r|)/2` operations, that is, before the following rebalancing. In order to create the scheduling, some auxiliary lazy functions are required. The function `rotateRev(f,r,a)` returns the list `f`, followed by the list `r`, and followed by the list `a`. It is required in this function that `|r|-2|f|` is 2 or 3. This function is defined by induction as `rotateRev(NIL,r,a)=reverse(r++a)` where `++` is the concatenation operation, and by `rotateRev(CONS(x,f),r,a)=CONS(x,rotateRev(f,drop(2,r),reverse (take(2,r))++a))`. It should be noted that, `rotateRev(f,r,NIL)` returns the list `f` followed by the list `r` reversed. The function `rotateDrop(f,j,r)` which returns `f` followed by (`r` without `j`'s first element) reversed is also required, for `j < |f|`. It is defined by `rotateDrop(f,0,r) == rotateRev(f,r,NIL)`, `rotateDrop(f,1,r) == rotateRev(f,drop(1,r),NIL)` and `rotateDrop(CONS(x,f),j,r) == CONS(x,rotateDrop(f,j-2),drop(2,r))`.

The balancing function can now be defined with

```
fun balance(q as (lenf, f,sf, lenr,r,sr))=
  if lenf > 2*lenr+1 then
    let val i= (left+lenr)div 2
        val j=lenf + lenr -i
        val f'=take(i,f)
        val r'=rotateDrop(r,i,f)
    in (i,f',f',j,r',r')
  else if lenf > 2*lenr+1 then
    let val j= (left+lenr)div 2
        val i=lenf + lenr -j
        val r'=take(i,r)
        val f'=rotateDrop(f,i,r)
    in (i,f',f',j,r',r')
  else q
```

Note that, without the lazy part of the implementation, this would be a non-persistent implementation of queue in `O(1)` amortized time. In this case, the lists `sf` and `sr` can be removed from the representation of the double-ended queue.

Language support

Ada's containers provides the generic packages `Ada.Containers.Vectors` and `Ada.Containers.Doubly_Linked_Lists`, for the dynamic array and linked list implementations, respectively.

C++'s Standard Template Library provides the class templates `std::deque` and `std::list`, for the multiple array and linked list implementations, respectively.

As of Java 6, Java's Collections Framework provides a new `Deque (https://docs.oracle.com/javase/10/docs/api/java/util/Deque.html)` interface that provides the functionality of insertion and removal at both ends. It is implemented by classes such as `ArrayDeque (https://docs.oracle.com/javase/10/docs/api/java/util/ArrayDeque.html)` (also new in Java 6) and `LinkedList (https://docs.oracle.com/javase/10/docs/api/java/util/LinkedList.html)`, providing the dynamic array and linked list implementations, respectively. However, the `ArrayDeque`, contrary to its name, does not support random access.

Perl's arrays have native support for both removing (shift (http://perldoc.perl.org/functions/shift.html) and pop (http://perldoc.perl.org/functions/pop.html)) and adding (unshift (http://perldoc.perl.org/functions/unshift.html) and push (http://perldoc.perl.org/functions/push.html)) elements on both ends.

Python 2.4 introduced the `collections` module with support for deque objects (https://docs.python.org/2.7/library/collections.html#deque-objects). It is implemented using a doubly linked list of fixed-length subarrays.

As of PHP 5.3, PHP's SPL extension contains the 'SplDoublyLinkedList' class that can be used to implement Deque datastructures. Previously to make a Deque structure the array functions array_shift/unshift/pop/push had to be used instead.

GHC's Data.Sequence (https://web.archive.org/web/20091015073943/http://www.haskell.org/ghc/docs/latest/html/libraries/containers/Data-Sequence.html) module implements an efficient, functional deque structure in Haskell. The implementation uses 2–3 finger trees annotated with sizes. There are other (fast) possibilities to implement purely functional (thus also persistent) double queues (most using heavily lazy evaluation).[2][3] Kaplan and Tarjan were the first to implement optimal confluently persistent catenable deques.[4] Their implementation was strictly purely functional in the sense that it did not use lazy evaluation. Okasaki simplified the data structure by using lazy evaluation with a bootstrapped data structure and degrading the performance bounds from worst-case to amortized. Kaplan, Okasaki, and Tarjan produced a simpler, non-bootstrapped, amortized version that can be implemented either using lazy evaluation or more efficiently using mutation in a broader but still restricted fashion. Mihaesau and Tarjan created a simpler (but still highly complex) strictly purely functional implementation of catenable deques, and also a much simpler implementation of strictly purely functional non-catenable deques, both of which have optimal worst-case bounds.

Complexity

- In a doubly-linked list implementation and assuming no allocation/deallocation overhead, the time complexity of all deque operations is O(1). Additionally, the time complexity of insertion or deletion in the middle, given an iterator, is O(1); however, the time complexity of random access by index is O(n).
- In a growing array, the amortized time complexity of all deque operations is O(1). Additionally, the time complexity of random access by index is O(1); but the time complexity of insertion or deletion in the middle is O(n).

Applications

One example where a deque can be used is the *A-Steal* job scheduling algorithm.[5] This algorithm implements task scheduling for several processors. A separate deque with threads to be executed is maintained for each processor. To execute the next thread, the processor gets the first element from the deque (using the "remove first element" deque operation). If the current thread forks, it is put back to the front of the deque ("insert element at front") and a new thread is executed. When one of the processors finishes execution of its own threads (i.e. its deque is empty), it can "steal" a thread from another processor: it gets the last element from the deque of another processor ("remove last element") and executes it. The steal-job scheduling algorithm is used by Intel's Threading Building Blocks (TBB) library for parallel programming.

See also

- Pipe
- Queue
- Priority queue

References

1. Donald Knuth. *The Art of Computer Programming*, Volume 1: *Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4. Section 2.2.1: Stacks, Queues, and Deques, pp. 238–243.
2. http://www.cs.cmu.edu/~rwh/theses/okasaki.pdf C. Okasaki, "Purely Functional Data Structures", September 1996
3. Adam L. Buchsbaum and Robert E. Tarjan. Confluently persistent deques via data structural bootstrapping. Journal of Algorithms, 18(3):513–547, May 1995. (pp. 58, 101, 125)
4. Haim Kaplan and Robert E. Tarjan. Purely functional representations of catenable sorted lists. In ACM Symposium on Theory of Computing, pages 202–211, May 1996. (pp. 4, 82, 84, 124)
5. Eitan Frachtenberg, Uwe Schwiegelshohn (2007). *Job Scheduling Strategies for Parallel Processing: 12th International Workshop, JSSPP 2006*. Springer. ISBN 3-540-71034-5. See p.22.

External links

- Type-safe open source deque implementation at Comprehensive C Archive Network (http://ccodearchive.net/info/deque.html)
- SGI STL Documentation: deque<T, Alloc> (http://www.sgi.com/tech/stl/Deque.html)
- Code Project: An In-Depth Study of the STL Deque Container (http://www.codeproject.com/KB/stl/vector_vs_deque.aspx)
- Deque implementation in C (http://www.martinbroadhurst.com/articles/deque.html)
- VBScript implementation of stack, queue, deque, and Red-Black Tree (http://www.ludvikjerabek.com/downloads.html)
- Multiple implementations of non-catenable deques in Haskell (https://code.google.com/p/deques/source/browse/)