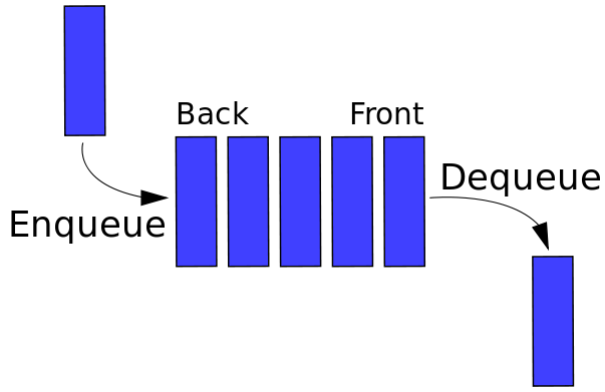
Queue (abstract data type)

In computer science, a **queue** (/ˈkjuː/ *KYEW*) is a particular kind of abstract data type or collection in which the entities in the collection are kept in order and the principle (or only) operations on the collection are the addition of entities to the rear terminal position, known as *enqueue,* and removal of entities from the front terminal position, known as *dequeue*. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed. Often a *peek* or *front* operation is also entered, returning the value of the front element without dequeuing it. A queue is an example of a linear data structure, or more abstractly a sequential collection.

Queue		
Time complexity in big O notation		
Algorithm	**Average**	**Worst case**
Space	O(n)	O(n)
Search	O(n)	O(n)
Insert	O(1)	O(1)
Delete	O(1)	O(1)



Representation of a FIFO (first in, first out) queue

Queues provide services in computer science, transport, and operations research where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer.

Queues are common in computer programs, where they are implemented as data structures coupled with access routines, as an abstract data structure or in object-oriented languages as classes. Common implementations are circular buffers and linked lists.

Contents

Queue implementation

Theoretically, one characteristic of a queue is that it does not have a specific capacity. Regardless of how many elements are already contained, a new element can always be added. It can also be empty, at which point removing an element will be impossible until a new element has been added again.

Fixed length arrays are limited in capacity, but it is not true that items need to be copied towards the head of the queue. The simple trick of turning the array into a closed circle and letting the head and tail drift around endlessly in that circle makes it unnecessary to ever move items stored in the array. If n is the size of the array, then computing indices modulo n will turn the array into a circle. This is still the conceptually simplest way to construct a queue in a high level language, but it does admittedly slow things down a little, because the array indices must be compared to zero and the array size, which is comparable to the time taken to check whether an array index is out of bounds, which some languages do, but this will certainly be the method of choice for a quick and dirty implementation, or for any high level language that does not have pointer syntax. The array size must be declared ahead of time, but some implementations simply double the declared array size when overflow occurs. Most modern languages with objects or pointers can implement or come with libraries for dynamic lists. Such data structures may have not specified fixed capacity limit besides memory constraints. Queue *overflow* results from trying to add an element onto a full queue and queue *underflow* happens when trying to remove an element from an empty queue.

A *bounded queue* is a queue limited to a fixed number of items.[1]

There are several efficient implementations of FIFO queues. An efficient implementation is one that can perform the operations—enqueuing and dequeuing—in O(1) time.

- Linked list
  - A doubly linked list has O(1) insertion and deletion at both ends, so is a natural choice for queues.
  - A regular singly linked list only has efficient insertion and deletion at one end. However, a small modification—keeping a pointer to the *last* node in addition to the first one—will enable it to implement an efficient queue.
- A deque implemented using a modified dynamic array

Queues and programming languages

Queues may be implemented as a separate data type, or may be considered a special case of a double-ended queue (deque) and not implemented separately. For example, Perl and Ruby allow pushing and popping an array from both ends, so one can use **push** and **unshift** functions to enqueue and dequeue a list (or, in reverse, one can use **shift** and **pop**), although in some cases these operations are not efficient.

C++'s Standard Template Library provides a "`queue`" templated class which is restricted to only push/pop operations. Since J2SE5.0, Java's library contains a `Queue (https://docs.oracle.com/javase/10/docs/api/java/util/Queue.html)` interface that specifies queue operations; implementing classes include `LinkedList (https://docs.oracle.com/javase/10/docs/api/java/util/LinkedList.html)` and (since J2SE 1.6) `ArrayDeque (https://docs.oracle.com/javase/10/docs/api/java/util/ArrayDeque.html)`. PHP has an SplQueue (http://www.php.net/manual/en/class.splqueue.php) class and third party libraries like beanstalk'd and Gearman.

Examples

A simple queue implemented in Ruby:

```ruby
class Queue
  def initialize
    @list = Array.new
  end

  def enqueue(element)
    @list << element
  end

  def dequeue
    @list.shift
  end
end
```

Purely functional implementation

Queues can also be implemented as a purely functional data structure.[2] Two versions of the implementation exist. The first one, called **real-time queue**,[3] presented below, 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 its worst-time complexity is $O(n)$ where *n* is the number of elements in the queue.

Let us recall that, for l a list, $|l|$ denotes its length, that *NIL* represents an empty list and $\mathrm{CONS}(h, t)$ represents the list whose head is *h* and whose tail is *t*.

Real-time queue

The data structure used to implement our queues consists of three linked lists (f, r, s) where *f* is the front of the queue, *r* is the rear of the queue in reverse order. The invariant of the structure is that *s* is the rear of *f* without its $|r|$ first elements, that is $|s| = |f| - |r|$. The tail of the queue $(\mathrm{CONS}(x, f), r, s)$ is then almost (f, r, s) and inserting an element *x* to (f, r, s) is almost $(f, \mathrm{CONS}(x, r), s)$. It is said almost, because in both of those results, $|s| = |f| - |r| + 1$. An auxiliary function aux must then be called for the invariant to be satisfied. Two cases must be considered, depending on whether s is the empty list, in which case $|r| = |f| + 1$, or not. The formal definition is $\mathrm{aux}(f, r, \mathrm{Cons}(\_, s)) = (f, r, s)$ and $\mathrm{aux}(f, r, \mathrm{NIL}) = (f', \mathrm{NIL}, f')$ where f' is *f* followed by *r* reversed.

Let us call $\mathrm{reverse}(f, r)$ the function which returns *f* followed by *r* reversed. Let us furthermore assume that $|r| = |f| + 1$, since it is the case when this function is called. More precisely, we define a lazy function $\mathrm{rotate}(f, r, a)$ which takes as input three list such that $|r| = |f| + 1$, and return the concatenation of *f*, of *r* reversed and of *a*. Then $\mathrm{reverse}(f, r) = \mathrm{rotate}(f, r, \mathrm{NIL})$. The inductive definition of rotate is $\mathrm{rotate}(\mathrm{NIL}, \mathrm{Cons}(y, \mathrm{NIL}), a) = \mathrm{Cons}(y, a)$ and $\mathrm{rotate}(\mathrm{CONS}(x, f), \mathrm{CONS}(y, r), a) = \mathrm{Cons}(x, \mathrm{rotate}(f, r, \mathrm{CONS}(y, a)))$. Its running time is $O(r)$, but, since lazy evaluation is used, the computation is delayed until the results is forced by the computation.

The list *s* in the data structure has two purposes. This list serves as a counter for $|f| - |r|$, indeed, $|f| = |r|$ if and only if *s* is the empty list. This counter allows us to ensure that the rear is never longer than the front list. Furthermore, using *s*, which is a tail of *f*, forces the computation of a part of the (lazy) list *f* during each *tail* and *insert* operation. Therefore, when $|f| = |r|$, the list *f* is totally forced. If it was not the case, the internal representation of *f* could be some append of append of... of append, and forcing would not be a constant time operation anymore.

Amortized queue

Note that, without the lazy part of the implementation, the real-time queue would be a non-persistent implementation of queue in $O(1)$ amortized time. In this case, the list *s* can be replaced by the integer $|f| - |r|$, and the reverse function would be called when s is 0.

See also

- Circular buffer
- Double-ended queue (deque)
- Priority queue
- Queueing theory
- Stack (abstract data type) – the "opposite" of a queue: LIFO (Last In First Out)

References

1. "Queue (Java Platform SE 7)" (http://docs.oracle.com/javase/7/docs/api/java/util/Queue.html). Docs.oracle.com. 2014-03-26. Retrieved 2014-05-22.
2. Okasaki, Chris. "Purely Functional Data Structures" (http://www.cs.cmu.edu/~rwh/theses/okasaki.pdf) (PDF).
3. Hood, Robert; Melville, Robert (November 1981). "Real-time queue operations in pure Lisp". *Information Processing Letters,*. **13** (2). hdl:1813/6273 (https://hdl.handle.net/1813%2F6273).

Further reading

- 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.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 10.1: Stacks and queues, pp. 200–204.
- William Ford, William Topp. *Data Structures with C++ and STL*, Second Edition. Prentice Hall, 2002. ISBN 0-13-085850-1. Chapter 8: Queues and Priority Queues, pp. 386–390.
- Adam Drozdek. *Data Structures and Algorithms in C++*, Third Edition. Thomson Course Technology, 2005. ISBN 0-534-49182-0. Chapter 4: Stacks and Queues, pp. 137–169.

External links

- STL Quick Reference (http://www.halpernwightsoftware.com/stdlib-scratch/quickref.html#containers14)
- VBScript implementation of stack, queue, deque, and Red-Black Tree (https://web.archive.org/web/20110714000645/http://www.ludvikjerabek.com/downloads.html)