Informatics in Education, 2011, Vol. 10, No. 1, 65–72

A Didactic Analysis of Functional Queues

Christian RINDERKNECHT
Konkuk University
143-701 Seoul Gwangjin-gu Hwayang-dong, Republic of Korea
e-mail: rinderkn@konkuk.ac.kr

Received: January 2011

Abstract. When first introduced to the analysis of algorithms, students are taught how to assess the best and worst cases, whereas the mean and amortized costs are considered advanced topics, usually saved for graduates. When presenting the latter, aggregate analysis is explained first because it is the most intuitive kind of amortized analysis, often involving enumerative combinatorics. We show how the aggregate analysis of functional queues can be carried out accurately and graphically, without combinatorics nor analytical tools like asymptotics, hence making it amenable to undergraduates. Our presentation is independent of any programming language.

Keywords: didactics of informatics, analysis of algorithms, amortized analysis, aggregate analysis, functional queue, functional language, Dyck path, Dyck meander.

1. Analysis of Algorithms

The branch of theoretical computer science devoted to the mathematical study of the efficiency of programs has been pioneered by Donald Knuth, who named it *analysis of algorithms* (Knuth, 1997, 2000; Sedgewick and Flajolet, 1996). Given a function definition, this approach consists basically in three steps: defining a measure on the arguments, which represents their size; defining a measure on time, which abstracts the wall-clock time; expressing the abstract time needed to compute calls to that function in terms of the size of its arguments. This function models the efficiency and is called the *cost* (the lower the cost, the higher the efficiency). For example, when sorting objects, also called keys, by comparing them, the input size is the number of keys and the abstract unit of time is often one comparison, so the cost is the mathematical function which associates the number of objects and the number of comparisons to sort them. Of course, the cost varies depending on the algorithm and it also often depends on the original partial ordering of the keys, so, for a given procedure, the size does not capture all the aspects needed to assess efficiency. This quite naturally leads to consider bounds on the cost: for a given input size, the *minimum cost* is the cost of the configurations leading to the smallest possible cost, or *best case*; the *maximum cost* corresponds to the *worst case*. For example, some sorting algorithms have their worst case when the objects are already sorted, others when they are sorted in reverse order.

Once we obtain bounds on a cost, the question about the *average* or *mean cost* (Vitter and Flajolet, 1990; Knuth, 1997, §1.2.10) arises as well. It is computed by taking the arithmetic mean of the costs for all possible inputs of a given size. Some care is necessary, as there must be a finite number of such inputs. For instance, to assess the mean cost of sorting algorithms based on comparisons, it is usual to assume that the input is a series of n distinct keys and that the sum of the costs is taken over all its permutations and divided by $n!$, the total number of permutations. The uniqueness constraints actually allows the analysis to equivalently, and more simply, consider the permutations of $(1, 2, \ldots, n)$. Some sorting algorithms, like *merge sort* (Knuth, 1998, §5.2.4; Cormen *et al.*, 2009, §2.3) or *insertion sort* (Knuth, 1998, §5.2.1; Cormen *et al.*, 2009, §2.1), have their average cost *asymptotically equivalent* to their maximum cost, that is, for increasingly large numbers of keys, the ratio of the two costs become arbitrarily close to 1. Some others, like *Hoare's sort*, also known as *quicksort* (Knuth, 1998, §5.2.2; Cormen *et al.*, 2009, §7), have the growth rate of their average cost being of a lower magnitude than the maximum, on an asymptotic scale (Graham *et al.*, 1994, §9)

Sorting algorithms can be distinguished depending on whether they operate on the whole series of keys, or key by key. The former are said *off-line*, as keys are not sorted while they are coming in, and the latter are called *on-line*, as the sorting process can be temporally interleaved with the input process. For instance, insertion sort is an on-line algorithm, whereas Hoare's sort is not because it is an instance of the divide-and-conquer strategy that splits the data. This distinction is pertinent in other contexts as well, like with algorithms that are intrinsically *sequential*, instead of allowing some degree of *parallelism*. For instance, a database is updated by a series of atomic requests, but different requests on different parts of the data might be performed in parallel.

Sometimes an update is costly because it is delayed due to an imbalance in the data structure that calls for an immediate remediation, but this remediation itself may lead to a state such that subsequent operations are faster than if the costly update had not happen. Therefore, when considering a series of updates, it may be overly pessimistic to cumulate the maximum costs of all the operations considered in isolation. Instead, *amortized analysis* (Okasaki, 1998; Cormen *et al.*, 2009, §17) takes into account the interactions between updates, so a lower maximum bound on the cost is derived. Note that this kind of analysis is inherently different from the average case analysis, as its object is the composition of different functions instead of independent calls to the same function on different inputs. Amortized analysis is a worst case analysis, but of a sequence of updates, not a single one.

For example, consider a counter enumerating the integers from 0 to n in binary by updating an array containing bits (Cormen *et al.*, 2009, §17.1). In the worst case, an increment leads to inverting $\lfloor \lg n \rfloor + 1$ bits, where $\lfloor x \rfloor$ is the greatest integer lower or equal than x and $\lg n$ is the binary logarithm of n, as it is the minimum number of bits required to encode n. The cost of the n increments is thus bounded from above by $n \lg n + n$, but this is too pessimistic, as carry propagation clears a series of rightmost bits to 0, so the next addition will flip only one bit, the following two etc. A counting argument shows that the exact total number of flips is $\sum_{k=0}^{\lfloor \lg n \rfloor} \lfloor n/2^k \rfloor < n \sum_{k \geqslant 0} 1/2^k = 2n$, which is of a lower magnitude than expected. This particular example resorts to a particular kind of amortized analysis called *aggregate analysis*, because it relies on enumerative combinatorics (Martin, 2001) to reach its result (it aggregates positive partial amounts, often in different manners, to obtain the total cost). As such, it is very much suited to teach undergraduates because it can be illustrated with tables and figures. A visually appealing variation on the previous example consists in determining the average number of 1-bits in the binary notation of the integers from 0 to n (Bush, 1940).

Despite its didactic qualities, aggregate analysis is less frequently applied when the data structures are not directly in connection with numeration. We propose to extend its scope by showing a compelling case study on *functional queues*.

2. Functional Queues

A functional queue is a linear data structure that is used in *functional languages*, whose semantics force the programmer to model a *queue* with two *stacks*. Items can be added to a stack, or *pushed*, on only one of its ends, called the *top*. They can be removed, or *popped*, only at the top:

$$\text{Push, Pop (top)} \leftrightsquigarrow \begin{array}{|c|c|c|c|c|}\hline a & b & c & d & e \\ \hline\end{array}$$

A queue is like a stack where items are added, or *enqueued*, at one end, called *rear*, but taken out, or *dequeued*, at the other end, called *front*:

$$\text{Enqueue (rear end)} \rightsquigarrow \begin{array}{|c|c|c|c|c|}\hline a & b & c & d & e \\ \hline\end{array} \rightsquigarrow \text{Dequeue (front end).}$$

Let us implement a queue with two stacks: one for enqueuing, called the *rear stack*, and one for dequeuing, called the *front stack*. The previous queue is equivalent to

$$\text{Enqueue (rear stack)} \rightsquigarrow \begin{array}{|c|c|c|}\hline a & b & c \\ \hline\end{array}\ \begin{array}{|c|c|}\hline d & e \\ \hline\end{array} \rightsquigarrow \text{Dequeue (front stack).}$$

Enqueuing is now pushing on the rear stack and dequeuing is popping on the front stack. In the latter case, if the front stack is empty and the rear stack is not, we swap the stacks and reverse the (new) front stack. Graphically, dequeuing in the configuration $\begin{array}{|c|c|c|}\hline a & b & c \\ \hline\end{array}\ [$ requires first to make $]\ \begin{array}{|c|c|c|}\hline a & b & c \\ \hline\end{array}$ and then dequeue c.

As a side note, although our presentation is independent of any programming language, programmers using Erlang (Armstrong, 2007, 2010) may implement enqueuings and dequeuings as in Fig. 1. As a measure of the input, we shall say that the queue

```
enqueue(Item,{Rear,Front}) -> {[Item|Rear],Front}.

dequeue({[Item|Rear],[]})     -> dequeue({[],rcat(Rear,[Item])});
dequeue({Rear,[Item|Front]}) -> {{Rear,Front},Item}.

rcat(          [],To) -> To;
rcat([Item|From],To) -> rcat(From,[Item|To]).
```

Fig. 1. Enqueuing and dequeuing in Erlang.

has size n if the total number of items in both stacks is n. As a measure of time, we shall count as one unit one item movement. Therefore, the cost of enqueuing is $\mathcal{C}_n^{\texttt{enq}} = 1$. The minimum cost for dequeuing is $\mathcal{B}_n^{\texttt{deq}} = 1$, when the front is not empty, so exactly one item moves (out). The maximum cost is $\mathcal{W}_n^{\texttt{deq}} = n+1$, when the front stack is empty and the rear contains n items: these move frontward and then the top moves out.

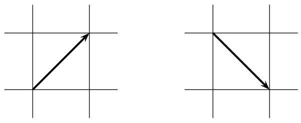
Let $\mathcal{C}_n$ be the cost of a sequence of n updates on a functional queue originally empty. A first attempt at assessing $\mathcal{C}_n$ consists in ignoring any dependence on previous operations and take the maximum cost individually. Since $\mathcal{C}_k^{\texttt{enq}} \leqslant \mathcal{C}_k^{\texttt{deq}}$, we consider a series of n dequeuings in their worst case, that is, with all the items located in the rear stack. Besides, after k updates, there may be k items in the queue, so we draw

$$\mathcal{C}_n \leqslant \sum_{k=1}^{n-1} \mathcal{W}_k^{\texttt{deq}} = \frac{1}{2}(n-1)(n+2) \sim \frac{1}{2}n^2.$$

Actually, this is overly pessimistic and even unrealistic. First, one cannot dequeue on an empty queue, therefore, at any time, the number of enqueuings since the beginning is always greater or equal than the number of dequeuings and the series must start with one enqueuing. Second, when dequeuing with the front being empty, the rear stack is reversed onto the front stack, so its items cannot be reversed again during the next dequeuing, whose cost will be 1. Moreover, as remarked above, $\mathcal{C}_k^{\texttt{enq}} \leqslant \mathcal{C}_k^{\texttt{deq}}$, so the worst case for a series of n operations occurs when the number of dequeuings is maximum, that is, when it is $\lfloor n/2 \rfloor$. If we denote by e the number of enqueuings and by d the number of dequeuings, we have the relationship $n = e + d$ and the two requisites for a worst case become $e = d$ (n even) or $e = d + 1$ (n odd).

Dyck Path ($e = d$). Let us represent graphically the updates as in Fig. 2. Textually, we represent an enqueuing as an opening parenthesis and a dequeuing as a closing parenthesis. For example, `( ( ( ) ( ) ( ( ) ) ) ( ) )` can be represented in Fig. 3 as a *Dyck path*, named in the honor of the logician Walther (von) Dyck (1856–1934). For a broken line to qualify as a Dyck path of length n, it has to start at the origin $(0, 0)$ and end at coordinates $(n, 0)$. In terms of a *Dyck language*, an enqueuing is called a *rise* and a dequeuing is called a *fall*. A rise followed by a fall, that is, `( )`, is called a *peak*. For instance, in Fig. 3, there are four peaks. The number near each rise or fall is the cost incurred by the corresponding operation. The abscissa axis bears the ordinal of each operation.

When $e = d$, the graphics is a Dyck path of length $n = 2e = 2d$. In order to deduce the total cost in this case, we must find a *decomposition* of the path, by which we mean



(a) Enqueue (b) Dequeue

Fig. 2. Graphical representations of operations on queues.

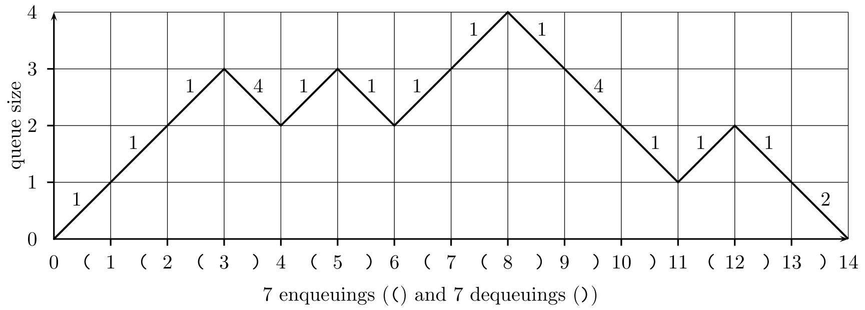


Fig. 3. Dyck path modeling queue operations (cost 21).

to identify patterns whose costs are easy to compute and which make up any path, or to associate any path to another path whose cost is the same but easy to find. Figure 4 shows how the previous path is mapped to an equivalent path only made of a series of isosceles triangles whose bases belong to the abscissa axis. Let us call them *mountains* and their series a *range*. The mapping is simple: after the first fall, if we are back to the abscissa axis, we have a mountain and we proceed with the rest of the path. Otherwise, the next operation is a rise and we exchange it with the first fall after it. This brings us down by 1 and the process resumes until the bottom line is reached. We call this process *rescheduling* because it amounts, in operational terms, to reordering subsequences of operations a posteriori. For instance, Figure 5 displays the rescheduling of Fig. 3. Note that two different paths can be rescheduled into the same path. What makes Fig. 5(c) equivalent to Fig. 5(a) is the invariance of the cost because all operations have cost 1. This always holds because enqueuings always have cost 1 and the dequeuings involved in a rescheduling have cost 1, because they found the front stack non-empty after a peak. We proved that all paths are equivalent to a range with the same cost. Therefore, the maximum cost can be found on ranges alone. Let us note $e_1, e_2, \ldots, e_k$ the maximal subsequences of rises; for example, in Fig. 4, we have $e_1 = 3$, $e_2 = 3$ and $e_3 = 1$. Of course, $e = e_1 + e_2 + \ldots + e_k$. The fall making up the ith peak incurs the cost $\mathcal{W}^{\mathsf{deq}}_{e_i} = e_i{+}1$, due to the front being empty because we started the rises from the abscissa

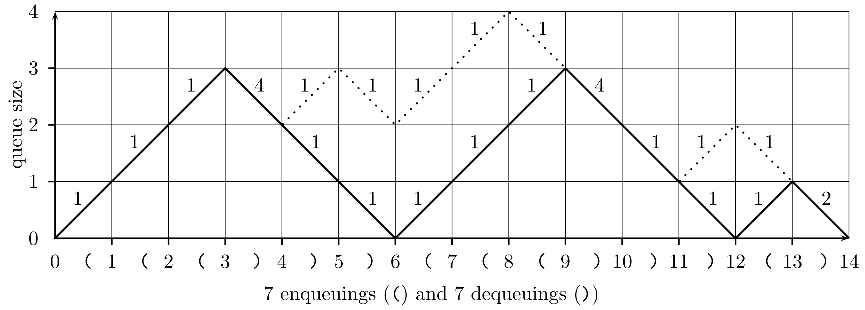
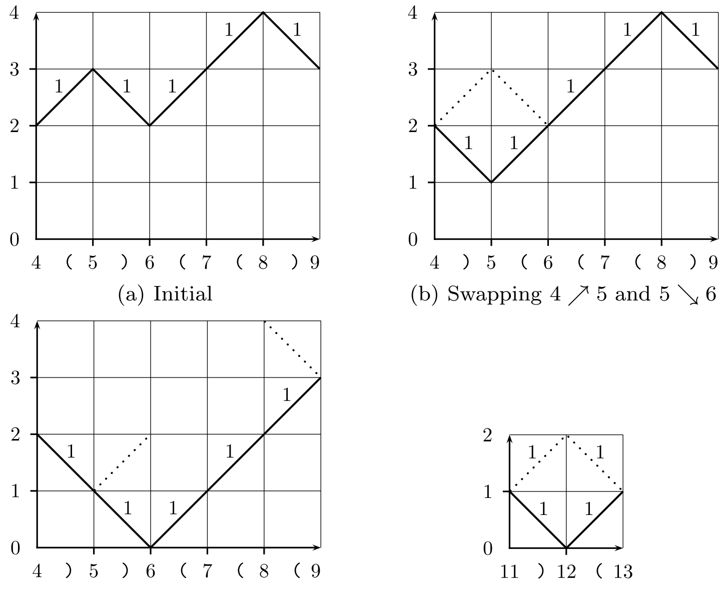


Fig. 4. Dyck path equivalent to Fig. 3.



(a) Initial

(b) Swapping 4 ↗ 5 and 5 ↘ 6

(c) Swapping 5 ↗ 6 and 8 ↘ 9

(d) Last one

Fig. 5. Rescheduling of Fig. 3 into Fig. 4.

axis. The next $e_i - 1$ falls have all cost 1, because the front is not empty. For the ith mountain, the cost is thus $e_i+(e_i+1)+(e_i-1) = 3e_i$. Then $\mathcal{C}_n = \sum_{i=1}^{k} 3e_i = 3e = \frac{3}{2}n$, since $n = e + d = 2e$.

Dyck Meander ($e = d + 1$). The other possibility for a worst case is that $e = d + 1$ and the graphics is then a *Dyck meander* whose extremity ends at ordinate $e - d = 1$. An example is given in Fig. 6, where the last operation is a dequeuing. The dotted line delineates the result of applying the rescheduling we used on Dyck paths. Here, the last operation becomes an enqueuing. Another possibility is shown in Fig. 7, where the last operation is left unchanged. The difference between the two examples lies in the fact

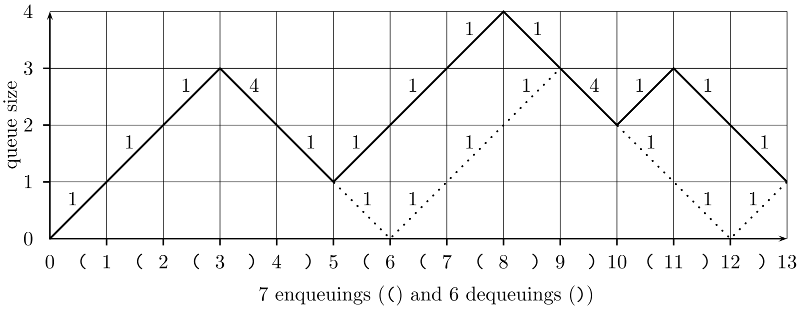


Fig. 6. Dyck meander modeling queue operations (total cost 19).

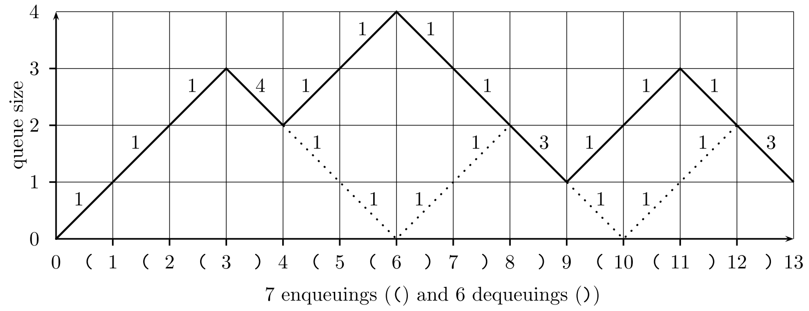


Fig. 7. Dyck meander modeling queue operations (total cost 20).

that the original last dequeuing has, in the former case, a cost of 1 (thus is changed) and, in the latter case, a cost greater than 1 (thus is invariant). The third kind of Dyck meander is one ending with an enqueuing, but because this enqueuing must start from the abscissa axis, this is the same situation as the result of rescheduling a meander ending with a dequeuing with cost 1 (see dotted line in Fig. 6 again). Therefore, we are left to compare the results of rescheduling meanders ending with a dequeuing, that is, we envisage two cases: either a range of $n - 1$ operations followed by an enqueuing or a range of $n - 3$ operations followed by two rises and one fall (of cost 5). In the former case, the total cost is $\mathcal{C}_n = \mathcal{C}_{n-1} + 1 = (3n - 1)/2$, because $n = e + d = 2e - 1$. In the latter case, the cost is $\mathcal{C}_n = \mathcal{C}_{n-3} + 5 = (3n + 1)/2$, which is slightly greater than all the previous costs, so it is the absolute maximum cost.

3. Conclusion

The cost $\mathcal{C}_n$ of a series of n queue updates, starting on an empty queue, is tightly bounded as $n \leqslant \mathcal{C}_n \leqslant (3n+1)/2$, where the lower bound happens when all updates are enqueuings and the upper bound when the queue ends containing one item, located at the front. As a consequence, the *amortized cost* of one operation is $\mathcal{C}_n/n$ and lies between 1 and 2. The average cost of a series of n updates is an open problem.

References

Armstrong, J. (2007), *Programming Erlang*, The Pragmatic Bookshelf.

Armstrong, J. (2010). Erlang. *Communications of the ACM*, 53(9), 68–75.

Bush, L.E. (1940). An asymptotic formula for the average sum of the digits of integers. *The American Mathematical Monthly*, 47(3), 154–156.

Cormen, T., Leiserson, C., Rivest, R., Stein, C. (2009). *Introduction to Algorithms*, 3rd edn. The MIT Press.

Graham, R.L., Knuth, D.E., Patashnik, O. (1994). *Concrete Mathematics*, 2nd edn. Addison-Wesley.

Knuth, D.E. (1997). *Fundamental Algorithms*, Vol. 1 of *The Art of Computer Programming*, 3rd edn. Addison-Wesley.

Knuth, D.E. (1998). *Sorting and Searching*, Vol. 3 of *The Art of Computer Programming*, 2nd edn. Addison-Wesley.

Knuth, D.E. (2000). *Selected Papers on the Analysis of Algorithms*. CSLI Publications.
Martin, G.E. (2001). *Counting: The Art of Enumerative Combinatorics*. Springer.
Okasaki, C. (1998). *Purely Functional Data Structures*. Cambridge University Press, 39–56.
Sedgewick, R., Flajolet, P. (1996). *An Introduction to the Analysis of Algorithms*. Addison-Wesley.
Vitter, J.S., Flajolet, P. (1990). *Algorithms and Complexity*, Vol. A of *Handbook of Theoretical Computer Science*. Elsevier Science, 431–524.

C. Rinderknecht received his MSc from Université Pierre et Marie Curie (Paris, France). His doctoral research at INRIA (French National Institute for Research in Computer Science and Control) dealt with the application of formal methods to telecommunication protocols and was defended in 1998, at Université Pierre et Marie Curie. Since 2005, he is an assistant professor at Konkuk University (Seoul, Republic of Korea), in the Department of Internet and Multimedia Engineering of the College of Information and Telecommunication. He teaches programming languages and the analysis of algorithms. His current research is about didactics of informatics.

Didaktinė funkcinių eilių analizė

Christian RINDERKNECHT

Funkcinė eilė yra tiesinė duomenų struktūra, naudojama funkcinėse kalbose, kuriose programuotojas gali modeliuoti eilę su dviem dėklais. Kai pirmą kartą studentams pristatoma algoritmų analizė, jie yra mokomi, kaip įvertinti geriausius ir blogiausius atvejus, tuo tarpu algoritmų reikšmė ir amortizacijos sąnaudos laikomos sudėtingesnėmis temomis, kurios skirtos absolventams. Dėstant pastarąsias temas, visų pirma paaiškinama bendroji analizė, kadangi tai yra intuityviojo tipo amortizuota analizė, dažnai apimanti išskaičiuojamąją kombinatoriką. Šiame straipsnyje autorius parodo, kaip bendra funkcinių eilių analizė gali būti atlikta tiksliai ir aiškiai, nesinaudojant nei kombinatorika, nei analizės priemonėmis, tokiomis kaip asimptotės. Pabrėžiama, kad ši analizė nepriklauso nuo programavimo kalbos.