Quicksort for Linked Lists

Tom Verhoeff[*]

January 1993

Abstract

I present and analyze a version of Quicksort for linked lists without "end"-pointers. On random lists its execution speed compares favorably to that of an efficient list version of Merge sort.

Keywords: Linked Lists, Sorting, Quicksort, Merge sort.

Contents

1 Specification

While working on a small graphics project, I got to a point where linked lists had to be sorted (on depth coordinate for the painter's algorithm [FvDFH90]). Merge sort is very appropriate for sorting lists. Nevertheless, I wondered whether I could also employ Quicksort, since for arrays Quicksort is on the average about 25% faster than Merge sort [Meh84, p. 67]. Knuth [Knu73] considers several sorting algorithms for linked lists, but Quicksort is not among them.

Before embarking on a specification of the problem, let me explain my notation for (finite abstract) lists, adapted from [BW88]. The list consisting of the n elements $a_0, \ldots, a_{n-1}$ (in this order) is denoted by $[a_0, \ldots, a_{n-1}]$. In particular, $[\,]$ denotes the

[*]Faculty of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands; E-mail: wstomv@win.tue.nl

empty list and $[a]$ denotes the list consisting of element a only. The catenation of lists s and t (in this order) is denoted by $s +\!\!+ t$. For element a and list s, I abbreviate $[a] +\!\!+ s$ to $a : s$. An infix dot stands for function application; it binds stronger than catenation. Function # returns the length of its list argument, for example $\#.(s +\!\!+ t) = \#.s + \#.t$. Finally, *sort* is a function from lists to lists such that $sort.s$ consists of the elements of s in ascending order.

type *ElementType* = ... ;
  List = ↑*Cell* ;
  Cell = **record**
    value: *ElementType* ;
    next: *List*
    end { *Cell* } ;

Figure 1: Type definitions to implement lists

The Pascal type definitions in Figure 1 form the basis of an implementation of lists over some element type. For convenience' sake, I assume that *ElementType* is ordered by $<$. For lists s and t over *ElementType*, I write $s < t$ to express that each elements of s precedes all elements of t.

The type *List* represents the set of lists over *ElementType*. In order to express the representation invariant and abstraction function, it is convenient to define some auxiliary operators on *List* (these are adapted from Lex Bijlsma's course notes for Programming 3). for p and q of type *List* and integer n ($n \geq 0$), define

$$
\begin{array}{rcl}
drop.p.n & = & \begin{cases} p & \text{if } p = \textbf{nil} \vee n = 0 \\ drop.(p\uparrow.next).(n-1) & \text{if } p \neq \textbf{nil} \wedge n > 0 \end{cases} \\
p \rightarrow q & = & (\exists\, n : n \geq 0 : drop.p.n = q) \\
p \triangleright q & = & \begin{cases} [\,] & \text{if } p = q \\ (p\uparrow.value) : (p\uparrow.next \triangleright q) & \text{if } p \neq q \end{cases}
\end{array}
$$

Mapping *drop* walks along the *next*-field, dropping an initial segment of the chain. Predicate $p \rightarrow q$ expresses that q is reachable from p. If $p \rightarrow q$ holds then $p \triangleright q$ is the list of values in the chain from p to q.

The representation invariant I and abstraction function $[\![\ ]\!]$ on the type *List* are now defined by

$$
\begin{array}{rcl}
I.s & = & s \rightarrow \textbf{nil} \\
[\![s]\!] & = & s \triangleright \textbf{nil}
\end{array}
$$

Note the following properties of the abstraction function.

$$
\begin{array}{rcl}
[\![p]\!] = [\,] & \equiv & p = \textbf{nil} \\
[\![p]\!] = a : s & \equiv & p\uparrow.value = a \wedge [\![p\uparrow.next]\!] = s
\end{array}
$$

The specification of the sorting problem can be expressed by

function *Quicksort*(*s*: *List*): *List* ;
{ assumes $[\![s]\!] = S$; returns *z*, where $[\![z]\!] = sort.S$ }

2 Design

The "standard" Pascal design for *Quicksort* is shown in Figure 2. The problem with this standard approach is that catenation of arbitrary *List* values cannot be done in constant time. Note that another representation of lists, viz. with also a pointer to the last element, would solve this problem. However, in my project the type *List* was given and could not be tinkered with. Anyway, the use of "end"-pointers would not have been an improvement, since either they have to be determined on the fly by walking through the list, or they have to be maintained by all other operations as well.

function *Quicksort*(*s*: *List*): *List* ;
{ assumes $[\![s]\!] = S$; returns *z*, where $[\![z]\!] = sort.S$ }
var *l*, *r*: *List* ;
begin
if $s =$ **nil then** { $[\![s]\!] = [\,]$ }
  Quicksort := *s*
else with $s\uparrow$ **do begin** { $[\![s]\!] = value : [\![next]\!]$ }
  "partition $[\![next]\!]$, using *value* as pivot" ;
  { $sort.S = sort.([\![l]\!] +\!\!+ [value] +\!\!+ [\![r]\!]) \wedge [\![l]\!] < [value] \leq [\![r]\!]$ }
  { hence, $sort.S = sort.[\![l]\!] +\!\!+ [value] +\!\!+ sort.[\![r]\!]$ }
  next := *Quicksort*(*r*) ; { $sort.S = sort.[\![l]\!] +\!\!+ [\![s]\!]$ }
  Quicksort := "catenation of *Quicksort*(*l*) and *s*"
  end { **else with** $s\uparrow$ }
end { *Quicksort* } ;

Figure 2: "Standard" design for Quicksort in Pascal

In a functional style [BW88, p. 154], *Quicksort* can be defined by

$$\begin{array}{rcl} Quicksort.[\,] & = & [\,] \\ Quicksort.(a:u) & = & Quicksort.l +\!\!+ [a] +\!\!+ Quicksort.r \\ & & \textbf{where} \quad l = [x \mid x \leftarrow u; x < a] \\ & & \qquad\quad r = [x \mid x \leftarrow u; a \leq x] \end{array}$$

Here, $[x \mid x \leftarrow u; x < a]$ is the list of elements x drawn from u that are less than a (see [BW88, p. 50]). In functional terms, my goal is to eliminate $+\!\!+$ in favor of the more efficient ':' operator.

The solution is based on the observation that *Quicksort* satisfies

$$Quicksort.s = Quicksort.s +\!\!+ [\,]$$

and, therefore, its specification can be generalized to *Qsort* by defining

$$Qsort.s.t \;=\; Quicksort.s \mathbin{+\!\!+} t$$

Specialization of this definition yields $Quicksort.s = Qsort.s.[\,]$. For *Qsort*, I derive

$$\begin{array}{lcl} Qsort.[\,].t & = & Quicksort.[\,] \mathbin{+\!\!+} t \\ & = & [\,] \mathbin{+\!\!+} t \\ & = & t \end{array}$$

and, assuming $l = [x \mid x \leftarrow u; x < a]$ and $r = [x \mid x \leftarrow u; a \leq x]$,

$$\begin{array}{lcl} Qsort.(a:u).t & = & Quicksort.(a:u) \mathbin{+\!\!+} t \\ & = & Quicksort.l \mathbin{+\!\!+} [a] \mathbin{+\!\!+} Quicksort.r \mathbin{+\!\!+} t \\ & = & Quicksort.l \mathbin{+\!\!+} (a : Quicksort.r \mathbin{+\!\!+} t) \\ & = & Qsort.l.(a : Qsort.r.t) \end{array}$$

Thus, a functional definition for *Qsort* is

$$\begin{array}{lcl} Qsort.[\,].t & = & t \\ Qsort.(a:u).t & = & Qsort.l.(a : Qsort.r.t) \\ & & \textbf{where} \;\; \begin{array}{rcl} l & = & [x \mid x \leftarrow u; x < a] \\ r & = & [x \mid x \leftarrow u; a \leq x] \end{array} \end{array}$$

Figure 3 gives the corresponding Pascal implementation of *Qsort*. I have coded the partition phase explicitly.

The Pascal implementation of *Quicksort* now boils down to

function *Quicksort*(*s*: *List*): *List* ;
{ assumes $[\![s]\!] = S$; returns *z*, where $[\![z]\!] = sort.S$ }
begin *Quicksort* := *Qsort*(*s*, **nil**) **end** ;

Of course, this list version of Quicksort still has quadratic worst-case time-complexity. Furthermore, it is not stable (in the sense that records with equal values retain their original relative ordering [Knu73, p. 4]), because each partitioning reverses the order (the lists are manipulated as stacks). Quicksort for arrays is, in general, also not stable. Stability can be guaranteed, but at the price of complicating the partitioning phase (also see [Mot81, Weg82], where "end"-pointers are used). In Section 4, I suggest another version of Quicksort for linked lists. That version is stable and also allows limitation of the maximum recursion depth. The latter cannot easily be accomplished in the Pascal version of *Qsort*, unless the compiler implements tail recursion efficiently or allows pointers to static variables. A final remark about *Qsort* is that the head of the list is used as partitioning pivot and that taking another element (for instance, a random element or the median of three random elements as is often done for improvement) would be costly, since lists cannot be indexed efficiently.

```
function Qsort(s, t: List): List ;
{ assumes [[s]] = S ∧ [[t]] = T ; returns z, where [[z]] = sort.S ++ T }
var a: ElementType ; u, l, r, h: List ;
begin
if s = nil then Qsort := t
else with s↑ do begin
   a := value ; u := next ; { [[s]] = a : [[u]] }
   l := nil ; r := nil ;
   { invariant: sort.S = sort.([[l]] ++ [a] ++ [[r]] ++ [[u]]) ∧ [[l]] < [a] ≤ [[r]] }
   while u ≠ nil do with u↑ do { [[u]] = value : [[next]] }
      if value < a then { "l, u := value : l, next" }
         begin h := next ; next := l ; l := u ; u := h end
      else { "r, u := value : r, next" }
         begin h := next ; next := r ; r := u ; u := h end ;
   { sort.S = sort.[[l]] ++ [a] ++ sort.[[r]] }
   next := Qsort(r, t) ; { sort.S = sort.[[l]] ++ [[s]] }
   Qsort := Qsort(l, s)
   end { else with s↑ }
end { Qsort } ;
```

Figure 3: Pascal implementation of *Qsort*

3 Efficiency

It is not guaranteed that the efficiency of the above list version of Quicksort is comparable to that of the array version. One has to be very careful to extrapolate performance figures [Sed78]. Therefore, it is interesting to compare *Qsort* to a (good) list version of Merge sort. In case of the array version, Quicksort is on the average 25% faster than Merge sort [Meh84, p. 67]. For the list versions their execution speed on random lists turns out to be comparable.

I consider the Pascal function for Merge sort given in Figure 4. It first splits the list into two halves by cutting the list roughly in the middle (in the **while**-loop, u is roughly halfway between s and t). A common alternative is to unzip the list into even and odd elements to obtain the two halves, but this turns out to be slightly more expensive and stability is lost. The two halves are then sorted recursively and finally merged to yield the result. The Boolean operators **cor** and **cand** stand for conditional **or** and **and** respectively (also known as short-circuit boolean operators).

The function *Merge* is defined in Figure 5. The precondition of *Merge* expresses that its arguments should be non-intertwined non-empty sorted lists. This is the case for the invocation in *Mergesort* above (however, invoking $Merge(s, s)$ would get into one kind of trouble or another). Of course, the invocation of *Merge* can be unfolded into a loop to obtain a more efficient program (with respect to both speed and memory usage). I leave this to the reader. By the way, the above combination of *Mergesort* and *Merge* forms a

```
function Mergesort(s: List): List ;
{ assumes [[s]] = S ; returns z, where [[z]] = sort.S }
var t, u: List ;
begin
if (s = nil) cor (s↑.next = nil) then Mergesort := s
else begin { #.S ≥ 2 }
   u := s ; t := u↑.next↑.next ;
   { invariant: #.(s ▷ u↑.next) = #.(u↑.next ▷ t) }
   while (t ≠ nil) cand (t↑.next ≠ nil) do
      begin u := u↑.next ; t := t↑.next↑.next end ;
   t := u↑.next ; u↑.next := nil ;
   { S = [[s]] ++ [[t]] ∧ 0 ≤ #.[[t]] − #.[[s]] ≤ 1, hence [[s]] ≠ [ ] ∧ [[t]] ≠ [ ] }
   Mergesort := Merge(Mergesort(s), Mergesort(t))
   end { else }
end { Mergesort } ;
```

Figure 4: Pascal implementation of Merge sort

stable sorting algorithm.

I have compared the execution times on random lists for Quicksort (based on *Qsort*, with singleton lists done directly instead of by partitioning), Merge sort (based on *Mergesort*, with test $s = \mathbf{nil}$ eliminated (because the random lists are non-empty), and unfolded *Merge*), and a list version of insertion sort. The results are presented in Table 1. The execution times are relative to those for applying the identity function. The functions were applied to the same data, by resetting the seed of the random generator before each test run.

```
function Merge(s, t: List): List ;
{ assumes ¬(∃ q : q ≠ nil : s → q ∧ t → q) ∧ [[s]] = S ∧ [[t]] = T ∧ }
{         S ≠ [ ] ∧ T ≠ [ ] ∧ sort.S = S ∧ sort.T = T }
{ returns z, where [[z]] = sort.(S ++ T) }
begin
if s↑.value ≤ t↑.value then with s↑ do { "Merge := value : Merge(next, t)" }
   if next = nil then begin next := t ; Merge := s end
   else begin next := Merge(next, t) ; Merge := s end
else with t↑ do begin { "Merge := value : Merge(s, next)" }
   if next = nil then begin next := s ; Merge := t end
   else begin next := Merge(s, next) ; Merge := t end
end { Merge } ;
```

Figure 5: Pascal implementation of *Merge*

List length	10	100	1000	10,000	100,000
Repeat count	10,000	1000	100	100	1
Quicksort	33	51	70	91	114
Merge sort	42	63	82	102	121
Insertion sort	11	86	846	8459	not run

Table 1: Comparison of execution times (seconds)

The advantage of *Quicksort* is that it contains just one loop (for partitioning), whereas *Mergesort* contains two loops (one for splitting and one for merging). The disadvantage of *Quicksort* is that partitioning may yield two lists whose lengths differ in order of magnitude, whereas *Mergesort* always works with a perfectly 'balanced' split. For random lists, the disadvantage is apparently less important than the advantage of a single loop. The break-even point for *Quicksort* and insertion sort occurs at lists of about fifty elements. For the array versions, the break-even point usually occurs at a much shorter length [Sed78], because array insertion involves data movement in contrast to list insertion.

4 Dummy Head Cells

The implementation of lists by the type *List* has a small disadvantage, viz. that all elements *except possibly the first* (if present) are pointed to from a *next*-field. This may give rise to a case distinction in programs dealing with such lists (this happens, for example, when maintaining stability in *Qsort* or unfolding *Merge* into a loop). The exceptional case can be avoided by prefixing each list with a dummy head cell. This is expressed in the following type definition, and corresponding representation invariant and abstraction function.

$$\begin{aligned} \mathit{HeadedList} &= \mathit{List} \\ I.s &= s \neq \mathbf{nil} \wedge s \rightarrow \mathbf{nil} \\ [\![s]\!] &= s\uparrow.\mathit{next} \triangleright \mathbf{nil} \end{aligned}$$

Each element in a list of type *HeadedList* is pointed to from a *next*-field. In particular, the first element is pointed to by the *next*-field of the dummy head cell.

The version of *Qsort* in Figure 6 is designed in terms of *HeadedList*. Partitioning does not reverse the order and, hence, *QsortH* is stable. Its maximum recursion depth has been limited by recursing only on the shorter of the two lists obtained by partitioning. For that purpose, counters *m* and *n* have been introduced. Observe that the (dummy) head cell of *s* is used as head cell of the "left" partitioning result, and the cell containing the pivot is used as head cell of the "right" partitioning result. Thus, no additional cells are needed.

Table 2 presents some timing results on random headed lists, again relative to applying the identity transformation. Entry 'Ltd. Quicksort' is based on *QsortH* with singleton lists done directly. Entry 'Quicksort' is like 'Ltd. Quicksort' but without limiting the

```
procedure QsortH(s, t: HeadedList) ;
{ assumes [[s]] = S ∧ [[t]] = T ∧ s↑.value = A }
{ establishes [[s]] = sort.S ++ T ∧ s↑.value = A }
var a: ElementType ; r: HeadedList ; sl, rl, u: List ; m, n: Integer ;
begin
while s↑.next ≠ nil do begin
  r := s↑.next ; u := r↑.next ; a := r↑.value ; { [[s]] = a : [[u]] }
  sl := s ; rl := r ; { abbreviate L = s↑.next ▷ sl↑.next and R = r↑.next ▷ rl↑.next }
  m := 0 ; n := 0 ; { m = #.L and n = #.R }
  { invariant: sort.S = sort.(L ++ [a] ++ R ++ [[u]]) ∧ L < [a] ≤ R }
  while u ≠ nil do with u↑ do { [[u]] = value : [[next]] }
    if value < a then { "L, u := L ++ [value], next" }
      begin sl↑.next := u ; sl := u ; m := m + 1 ; u := next end
    else { "R, u := R ++ [value], next" }
      begin rl↑.next := u ; rl := u ; n := n + 1 ; u := next end ;
  sl↑.next := nil ; rl↑.next := nil ;
  { sort.S = sort.[[s]] ++ [a] ++ sort.[[r]] }
  if m < n then begin QsortH(s, r) ; s := r end
  else begin QsortH(r, t) ; t := r end
  end { while } ;
s↑.next := t
end { Qsort } ;
```

Figure 6: Version of *Qsort* in terms of *HeadedList*

maximum recursion depth. Entry 'Merge sort' is adapted from Merge sort in the preceding section. Note that each iteration through the partitioning loop of *QSortH* consists of

List length	10	100	1000	10,000	100,000
Repeat count	10,000	1000	100	100	1
Ltd. Quicksort	29	47	68	90	114
Quicksort	32	49	67	85	105
Merge sort	44	66	87	107	127

Table 2: Comparison of execution times (seconds) for headed lists

two comparisons and four assignments, the same as in *QSort*. This explains why 'Ltd. Quicksort' for headed lists performs almost the same as Quicksort in Table 1, and also why Quicksort in Table 2 performs better (it has one assignment fewer in its partitioning loop). Merge sort in Table 2 performs less well that in Table 1, because only the (dummy) head cell of its argument is available (no pivot), and thus the second half (of the split) must be attached to that same head cell before recursing on it.

5 Concluding Remarks

I have presented a list version of Quicksort for linked lists, called *Qsort*. The lists have no "end"-pointer to the last element. The Pascal program *Qsort* is very short. On random lists it performs well compared to a list version of Merge sort. There are however a few side remarks to be made.

First of all, *Qsort* is not stable. On the other hand, most versions of Quicksort and some versions of Merge sort are also not stable. Secondly, it is harder to implement a version that does not use the head of the list as pivot (but, for instance, a random element). In Pascal it is also harder to limit the maximum recursion depth by putting the "longest" recursive call at the end and transforming it into a loop. However, when using lists with a dummy head cell, it is easy to guarantee stability and to limit the maximum recursion depth.

In the same vein, a list version of Quicksort is feasible that operates in constant-space (i.e. without recursion overhead) [Ver93].

I would like to thank Johan Lukkien and the Eindhoven Algorithm Club led by Anne Kaldewaij for reading and commenting on an earlier version of this note.

References

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, second edition, 1990.

[Knu73] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[Meh84] Kurt Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1984.

[Mot81] Dalia Motzkin. A stable Quicksort. *Software—Practice and Experience*, 11:607–611, June 1981.

[Sed78] Robert Sedgewick. Implementing Quicksort programs. *Communications of the ACM*, 21(10):847–857, October 1978.

[Ver93] Tom Verhoeff. Constant-space Quicksort, January 1993. In preparation.

[Weg82] Lutz M. Wegner. Sorting a linked list with equal keys. *Information Processing Letters*, 15(5):205–208, December 1982.