数理解析研究所講究録
556巻 1985年 1-16

On the Nested Heap Structure in Smoothsort

野下 浩平　　仲谷 栄伸
Kohei Noshita and Yoshinobu Nakatani
Department of Computer Science
Denkitusin University, Chofu, Tokyo 182

Abstract

An idea for building up the nested heap structure is presented, which leads to an efficient variant of Smoothsort. The worst-case complexity is substantially improved in the sense that the number of comparisons is at most $2n \log_2 n$, which is as many as that of the well-known Heapsort. This smooth algorithm is also able to "smoothly" sort Hertel's examples, for which the original Smoothsort fails to be smooth in terms of the number of inversions. Another variant shares all the basic characteristics with Smoothsort, i.e., it is a smooth algorithm in situ with $O(n \log n)$ comparisons in the worst case. Furthermore, it is also smooth for Hertel's examples. Some other variants with different characteristics are also suggested.

1. Introduction

Smoothsort is a comparison-based sorting algorithm by Dijkstra ([1], [2]), which has the following three basic properties:

(1) The number of comparisons in the worst case is $O(n \log n)$, where n is the number of elements to be sorted. (The base of logarithm is 2.)

(2) It is "smooth" in the sense that, for presorted (i.e., already

sorted) input, the number of comparisons is O(n) and the number of swaps (i.e., interchanges of two elements) is zero.

(3) It is an "in situ" algorithm, i.e., it requires only O(1) memory words of ($\lceil \log n \rceil + c$) bits for the working space, where c is a small integer constant (≥ 1).

We define $\omega = \lceil \log n \rceil + c$ as the word length.

The well-known Heapsort ([3], [7]) is not smooth, but has the other two properties above.

From the complexity viewpoint of practical sorting algorithms, the constant factors of their complexities must be taken into consideration. In this sense, as analysed in detail in this paper, Smoothsort is not enough efficient to replace Heapsort, in spite of its smoothness property. Therefore, it is natural to ask whether the worst-case complexity of Smoothsort can be improved to be competitive with that of Heapsort. This is our first question to be investigated in this paper.

Hertel [4] has pointed out that Smoothsort fails to sort smoothly for some input in terms of the number of inversions included in input. He has shown an example of input for each n, named Hertel's input, which forces Smoothsort to make O(n log n) swaps. Note that Mehlhorn's algorithm [6] makes only O(n log log n) swaps for Hertel's input, although his algorithm requires at least 2n words for the working space. Our second question is whether we can modify Smoothsort to sort Hertel's input smoothly.

This paper presents an idea for building up the nested heap structure [7], and describes two variants of Smoothsort based on this idea. They are more efficient than the original Smoothsort in various senses.

In our variants, we will use familiar balanced binary heaps rather than Leonardo trees [1]. The reason for this will be explained later.

The main results of this paper are now summarised.

The first variant, named **N**, has the following properties.

(1) The number of comparisons in the worst case is at most 2n log n, which is as good as that of Heapsort.

(2) It smoothly sorts not only presorted input but also Hertel's input.

(3) It requires, however, either 3 log n words or 4 log log n words for the working space, depending on whether our machine executes a certain type of bit-searching instruction quickly. In any case, **N** may still be called in situ, at least in less strict sense.

The second variant, named **E**, shares all the three basic properties with Smoothsort, and it also sorts Hertel's input smoothly. The number of comparisons in the worst case, however, is not as good as **N** by some constant factor.

Some other variants are briefly described, and their detailed analysis is left for the future investigation.

While we analyse the total running time, rather than the number of comparisons, of our algorithms including the original Smoothsort, we encounter a delicate problem on the index calculation in the packed working space, which is also discussed in this paper.

2. Smoothsort Revisited

In this section we sketch the algorithm of Smoothsort, and review some complexity results. The reader is assumed to be familiar with [1] and [3] as well as Heapsort [5]. For brevity, let **S** stand for Smoothsort.

Our basic data structure of **S** is the ordered set of binary heaps, whose general structure can be seen from an example for n=24 in Figure 1. The words 'left', 'right', 'up' and 'down' are to be understood from this picture. Given n elements as an input in a linear array A[1..n], **S** sorts them in the ascending order in the following three steps.

Step 1: Build the set of heaps in a similar way to the heap-creation step of Heapsort.

Step 2: Sort the set of roots of those heaps by means of the straight-insertion algorithm, which transfers each element from right to left by comparing it with its two sons and the root of the left-adjacent heap. Each straight-inserting process for one element is always followed by the sifting-up process for finding the proper place for the element in question.

Step 3: Repeat the maximum-selection process from right to left, by deleting the rightmost element, and transferring, if any, its two sons one by one toward the left by straight-inserting, followed by sifting-up as in step 2. This step corresponds to the maximum-selection step of Heapsort.

The key for implementing **S** is how to represent the set of indices of roots. Define $m = \lfloor \log(n+1) \rfloor$. A pair of arrays X and H of m (more precisely, m+1) words for each suffices for this purpose, where X and H contain the indices of roots and the height of corresponding heaps, respectively. The height of a heap is understood to be counted from 1; thus the height of a leaf is 1. (In practice, for the values in H, 2 to the power of height is preferred.)

As an alternative method for implementing **S**, in situ, we introduce an array B[1..m] of bits, in place of both X and H. We can assume that B can be packed into a single word of ω bits in the usual sense of the uniform cost criterion on random-access machines. The most basic property of B is that

$B[s] = 1$ iff there is a non-rightmost heap of size 2^s-1.

We also introduce a simple variable t ($1 \leq t \leq m$) as an index of B for indicating the height of the rightmost heap. Given an index of the root of a heap and its height, we can easily find its two sons and the root of the left-adjacent heap in O(1) time. Furthermore, we can also calculate the height of any of two adjacent heaps by way of B. (The timing analysis of this calculation will be discussed in detail later. Note that, if X and H are used, this calculation can be trivially done in O(1) time.)

Now we evaluate the efficiency of **S** by counting the number of comparisons in the asymptotic sense. This number will be denoted by C. (The number of swaps are easily estimated.) We derive an upper bound of C in the worst case. Let C_j stand for the part of C in Step i ($1 \leq i \leq 3$).

$C_1 = 4n.$

This is obvious from the analysis of Heapsort [5].

$C_2 = 3(\log n)^2/2 + (2 \log n)(\log n) = 3.5(\log n)^2.$

The first term is due to the straight-inserting process of length log n. Note that three comparisons are made for determining if an element is to be transferred to the left. The second term comes from the final sifting-up process.

$C_3 = 3n(\log n)/2 + 2n \log n = 3.5n \log n.$

Deriving C_3 needs some analysis, since the number of heaps varies from 1 to log n. We can prove the following. (The proof is left to the reader.)

<u>Proposition</u> The total number of heaps summed up throughout the execution in Step 3 is about n(log n)/2.

Hence we have the first term. Note that the coefficient 3 has been mentioned in C_2. The second term is due to the sifting-up process of at most n elements.

Hence we have the desired upper bound C in the worst case:

$C = 3.5n \log n$.

For presorted input and Hertel's input, the number of comparisons is easily shown to be O(n) and O(n log n) [4], respectively.

The reader who only wants to see the outline of our variants may skip all parts of the timing analysis of the index calculation. Now we begin the first part of this analysis, that lasts until the end of this section.

We introduce the following machine instruction. Let

bitsearch(v,h,k)

be an instruction operating on the value of v of ω bits, which searches v for k-th 1 toward higher bits (left) by starting at h-th bit ($h \geq 1$), and returns the obtained position as a result. We assume that it searches toward the right if $k < 0$. For example,

bitsearch((1010011010),3,3)

yields 8. (A bit number is to be counted from 1 toward the left.)

For the timing analysis, we assume either of the following two

types of machine.

Type I: In the same way as other instructions for basic arithmetic operations on ω bits, this type of machine executes bitsearch in O(1) time.

Type II: The instruction bitsearch takes d units of time, where d is the number of bits between the start bit h and the destination bit. (This instruction is probably implemented as a subroutine.)

Now we verify that in **S** the total running time is proportional to the number of comparisons on Type II machines. Note that this trivially holds on Type I machines.

For executing the straight-inserting process, we must search B repeatedly toward the left to obtain the height of left-adjacent roots. In this case, the running time can be bounded by O(log n) in total, because each bit in B is scanned at most once only from right to left. Hence the whole cost of the index calculation is O(n log n) in the worst case, since there is no other process which needs bit-searching in B.

The same analysis is applied to Hertel's input, because they force **S** to make O(n log n) comparisons during the straight-inserting process.

The case of presorted input is obvious, since we need not know the height of the left-adjacent heap.

3. Improving the Worst-Case Complexity

We present a variant **N** of **S**, which has the three properties listed in §1. The basic idea of **N** is to make the heap structure be nested, i.e., we build up another set of heaps from the set of roots of the original heaps. The main point of this idea is that we can replace

the straight-inserting process in the first heaps in Step 3 of **S** by a less costly process in the second heaps. The implementation of this idea and its analysis are not so straightforward as they look, which are the subjects below.

For convenience, let S_1 stand for the set of heaps represented in A (the first heaps), and R_1 for the set of roots of heaps in S_1. We represent R_1 in another array A_2 in the same heap structure as S_1, denoted by S_2. The set of roots of heaps in S_2 will be denoted by R_2.

We represent R_1 in a pair of arrays X_1 and H_1 of O(log n) words, as explained in §2. An array A_2 of log n words suffices to build up S_2, where $A_2[i]$ contains an index j of X_1 (j-th root), rather than its index of A_1 itself. An array AH_2 of log n words is also used, where $AH_2[i]$ is the height of i-th node in S_2. Note that the values of $AH_2[i]$ for all i can be fixed prior to the execution. Another array D_2 of log n words is introduced to represent the inverse permutation of A_2, i.e., $A_2[i] = j$ iff $D_2[j] = i$. This is used to locate where an element in R_1 has been moved in S_2. As usual, R_2 is represented in a pair of arrays X_2 and H_2 of log log n words, since the size of R_2 is about log log n. Note that H_2 is redundant and need not be used, because of AH_2.

For the discussion of the working space, it is important to note that A_2, AH_2 and D_2 can be directly packed into O(log log n) words, since the value of each index contained in those arrays ranges from 1 to log n, which is coded into at most log log n bits. By way of those arrays, for example, when a root in R_2 with index x_2 of X_2 is given, we can calculate the index (of A) of the left-adjacent root

in R_2 in O(1) time.

We are ready to describe the algorithm. The first two steps are exactly the same as S. (The second step may be replaced by a simpler one.) At the beginning of Step 3, D_2, A_2 and X_2 (and H_2) are appropriately initialised. The maximum-selection is repeated in the following way.

Step 3: Let z be the rightmost element in A_2, which is the maximum of all. The index of z in A can be obtained through D_2. In A, z is at the root of some heap w_1 in S_1. The rightmost element v in A is at some node x_2 in some heap w_2 in S_2 (in A_2). Let p_1 and q_1 be, if any, two elements at the left and right sons of v in A, and let p_2 and q_2 be, if any, two elements at the left and right sons of z in A_2, respectively. (See Figure 2.)

Step 3a [in case there exist two sons p_1 and q_1 in A]

If z happens to be the rightmost element in A, let u be p_1. Otherwise, move simultaneously p_1 and v to the nodes z and p_1, respectively ($v \geq p_1$), and sift up w_1 with p_1. Let u be the new element at the root of w_1. (Note that so far no values in A_2 and D_2 have been changed.)

Assign u to the rightmost node z in A_2, and transfer it toward the left by straight-inserting and sifting-up in S_2. Finally, increase S_2 by appending q_1 to the right end of A_2, and transfer it toward the left.

Step 3b [in case there exist no sons in A]

If there exist p_2 and q_2 in A_2, transfer p_2 and q_2 toward the left. (S_2 has been increased by appending q_2.)

If z is not the rightmost element in A, move v to the node z, and sift up w_1 in A. Let u' be the new element at the

root of w_1. Assign u' to the node x_2 in A_2, and transfer u' toward the root of w_2, and then, if necessary, toward the right along R_2. (AH_2 is useful for calculating indices.)

This is the description of **N**. Note that exchanging z and v in A at the beginning of Step 3a leads to a less efficient implementation.

Now the basic properties are verified.

The total number of comparisons is at most 2n log n, since the sifting-up process in S_1 in Step 3 is invoked only once and the rest can be done in O(log log n) comparisons. The property for presorted input is obvious. For Hertel's input, we have eliminated the costly straight-inserting process along R_1. This implies the desired O(n log log n) comparisons.

Obviously, the working space consists of 2 log n words for X_1 and H_1, plus 4 log log n words for A_2, AH_2, D_2 and X_2. As before, X_1 and H_1 can be replaced by a single word B_1. Hence, we need only 4 log log n words for the working space.

Now we get into the discussion on the index calculation in this packed working space. The index calculation here is this: for a given index x_2 of A_2, calculate the corresponding index x_1 of A. The calculation proceeds as follows:

```
begin
     y := A_2[x]; {y-th root in S_1}
     j := bitsearch(B_1,m+1,-y);
     x_1 := (B_1÷2^(j-1))*2^(j-1)-j {masking off the lower (j-1) bits minus j}
end
```

We can assume that all the operations in this calculation can be done in O(1) time, except bitsearch. Therefore, on Type I machines,

we need not impose any extra cost on the total running time within constant factors. But on Type II machines, the upper bound of, for example, the worst-case running time is at best expressed to be O(n log n log log n), since, during the straight-inserting, transferring and sifting-up processes in S_2, we must count (log n) time per comparison.

(Note that in the packed working space, yet another array AH_1 of log log n words may facilitate the index calculation during straight-inserting in R_2, where $AH_1[i]$ represents the height of the root in R_1 which corresponds to i-th root in R_2.)

4. The Second Variant and Its Extension

We present another method, named **E**, for implementing our basic idea. Here the second set of heaps is directly embedded in A, rather than separately represented in A_2 of **N**.

The properties of **E** are summarised:

(1) O(n log n) comparisons in the worst case,

(2) Smooth for Hertel's input as well as presorted input,

(3) In situ in the strict sense, i.e., O(1) words for the working space.

For implementing **E**, two familiar arrays B_1 for S_1 and B_2 for S_2 suffice for the index calculation. The nested heap structure of **E** is easily understood by studying an example in Figure 3. As seen in Figure 3, if t is the height of a heap w in S_2, there are 2^t-1 heaps of S_1 included in w.

For brevity, we give three essential points for implementing and analysing Step 3 of E, as the first two steps are the same as before.

(1) ordering relation

As R_2 is directly represented in A, the value of any element v

in R_2, is always maintained to be larger than (or equal to) its two sons in R_1 as well as its two sons in R_2 and its left-adjacent root in R_2. Thus, in the straight-inserting process, 5 comparisons are necessary for determining whether the element in question is to be tranferred to the left-adjacent root.

(2) maximum-selection

On deleting the maximum, four elements p_2, q_2 (in R_2) and p_1, q_1 (in R_1) may emerge for readjusting R_2. See Figure 4. As in Step 3a of **N**, p_1 and q_1 are taken up for transferring toward the left. This transfer of p_1 and q_1 is followed by sifting-up in S_2 and, possibly, by another sifting-up in S_1. Note that this final sifting-up process may cost 4 log n comparisons, and it has never occurred in the previous variant **N**.

In the other case that no p_1 and q_1 emerge, which happens n/2 times, p_2 and q_2 are taken up for readjusting R_2, and transferred in a similar way as above.

The total number of comparisons in the worst case is obviously O(n log n). Note that p_1 taken up in the first case may emerge later as either p_2(leaf) or q_2. (It is difficult to analyse how frequently p_2 and q_2 actually emerge, although it is predetermined only by n. We have exhaustively calculated it by computer up to $n=5\ 10^5$. This seems to be strongly suggesting 2.8n log n comparisons in the worst case, but so far we have not been able to prove it in general. In the computing experiments, **E** has made about 15% less comparisons than **S** for random input of n = 16000.)

(3) index calculation

During straight-inserting and sifting-up for transferring an element

in R_2, at most 5 log log n nodes in S_1 are taken up. However, for finding the index of left-adjacent roots in R_2, B must be scanned toward the left, which costs log n time in total on Type II machines. In the worst case, the running time of this index calculation may be comparable with that of the sifting-up process in S_1. Hence, the worst-case complexity is O(n log n) time in total. For Hertel's input, however, the running time is (n log n), while the number of comparisons is O(n log log n). For presorted input, the order of the running time may be n log n, because we need bit-searching in B for the index calculation, for example, for obtaining the index p_2.

In summary, **E** has all the desired properties in terms of the number of comparisons, but, on Type II machines, presorted input as well as Hertel's input requires some extra running time.

As we have seen two methods based on the nested heap structure, it is now easy to generalise our idea to the multiple nested heap structure.

Namely, **N** and **E** are naturally generalised to $\mathbf{N}^k$ and $\mathbf{E}^k$ with k nested heaps, where k is a constant. The number of roots of the top heaps is $O(n \overbrace{\log \ldots \log}^{k} n)$. This may be only of theoretical interest, but useful to preserve the smoothness property for naturally generalised Hertel's input, which force to create the worst-case behaviour in straight-inserting in the top heaps. Furthermore, we can make the depth of nesting to dynamically change with the maximum depth of $O(\log^* n)$, leading to another variant $\mathbf{N}^*$ or $\mathbf{E}^*$. (Taking $\log^* n$ logarithms of n yields 1 or less.) In this case, the worst-case complexity must be multiplied by $\log^* n$.

5. Concluding Remarks

In this paper we have presented several variants of Smoothsort and analysed their complexity. Some computing experiments for random input up to n = 16000 have supported our ideas for speeding up Smoothsort. In these experiments our first variant **N** has shown the best performance in terms of the number of comparisons among all the smooth algorithms in this paper including yet another variant without nested heaps mentioned below. In case n=16000, **N** has made only about 10% more comparisons than Heapsort.

In [1], Dijkstra prefers Leonardo trees to binary heaps, because the number of roots of Leonardo trees are about 20% less than the size of R_1. But, as discussed in our variants, this size is by no means significant — the leading term of the worst-case complexity depends only on the sifting-up process on a heap. Furthermore, as shown in §3, the binary heap structure enables us to calculate indices efficiently.

The original Smoothsort is an elegant algorithm with the three basic properties, but it needs further investigation from the complexity viewpoint, as we have seen an approach based on nested heaps for improving its efficiency. Without nested heaps, the reader may have come up with another simple variant which seems very efficient in the worst case, provided that n is small, say $n \le 2^{20} \doteq 10^6$. Let n be fixed to be 2^{20}. The original Smoothsort makes about $3(\log n)/2 + 2 \log n = 70$ comparisons per one element in Step 3 as seen in 2. If R_1 is represented in a separate array in a similar way to **N** and a modified straight-insertion algorithm is employed, we can prove that this requires only $(3/8)\log n + 2 \log n = 48$ comparisons. This simple variant in situ in less strict sense, however, has not been able to beat our best **N**

for random input of n=16000 in our experiments.

References

[1] Dijkstra, E. W., Smoothsort, an Alternative for Sorting In Situ, Science of Computer Programming, 1 (1982), 223-233; 2 (1982), 85.

[2] Dijkstra, E. W. and A. J. M. van Gasteren, An Introduction to Three Algorithms for Sorting In Situ, Info. Proc. Lett., 15, 3 (1982), 129-134.

[3] Floyd, R. W., Algorithm 242 Treesort 3, Comm. ACM, 7, 12 (1964), 701.

[4] Hertel, S., Smoothsort's Behavior on Presorted Sequences, Info. Proc. Lett., 16, 4 (1983), 165-170.

[5] Knuth, D. E., The Art of Computer Programming, Vol. 3 (Sorting and Searching), Addison-Wesley (1973).

[6] Mehlhorn, K., Sorting Presorted Files, 4th GI Conference, Lecture Notes in Computer Science, 67 (1979), 199-212.

[7] Williams, J. W. J., Algorithms 232 Heapsort, Comm. ACM, 7, 6 (1964), 347-348.

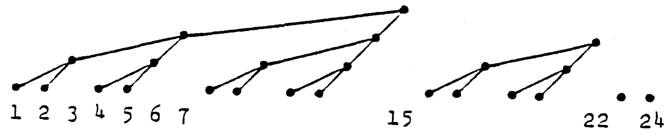

Figure 1. The heap structure in Smoothsort (n=24)

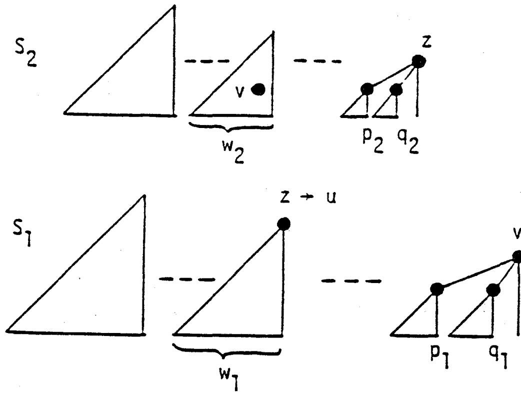

Figure 2. Two sets of heaps in N

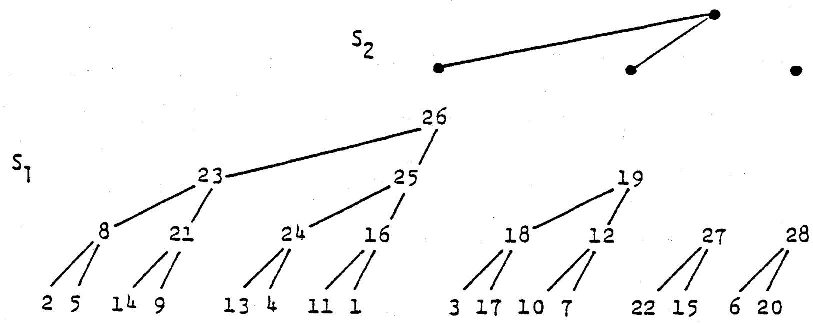

Figure 3. An example of two sets of heaps in E (n=28)

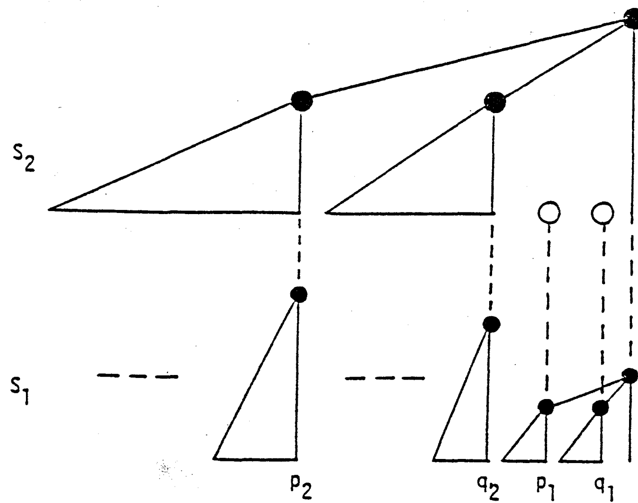

Figure 4. The ordering relation of four sons in E