An Efficient Construction Algorithm for a Class of Implicit Double-Ended Priority Queues

JINGSEN CHEN

Department of Computer Science, Luleå University, S-971 87 Luleå, Sweden
Email: jingsen@sm.luth.se

Priority queues and double-ended priority queues are fundamental data types in Computer Science, and various data structures have been proposed to implement them. In particular, diamond deques, interval heaps, min-max-pair heaps, and twin-heaps provide implicit structures for double-ended priority queues. Although these heap-like structures are essentially the same when they are presented in an abstract manner, they possess different implementations and thus have different construction algorithms. In this paper, we present a fast algorithm for building these data structures. Our results improve over previously fast known algorithms.

Received April 5, 1995; revised November 14, 1995

1. INTRODUCTION

A *priority queue* is a set of elements on which two basic operations are defined: inserting a new element into the set; and retrieving and deleting the minimum element of the set. A *double-ended priority queue* (*priority deque*, for short) provides insert access and remove access to both the minimum and maximum elements at the same time. They have been useful in many applications [1, 9, 11]. Various data structures have been developed for implementing priority queues and priority deques efficiently. In particular, diamond deques [5], interval heaps [15], min-max-pair heaps [11] and twin-heaps [9] provide implicit structures for mergeable priority deques. These heap-like structures, however, possess different implementations and thus have different construction algorithms. By *mergeable* we mean that the merge operation on the corresponding data structure that implements the queue can be performed in sublinear time. In this paper, we show how these data structures relate to each other and develop a fast algorithm for building the structures.

2. PRELIMINARIES

An implicit data structure describes the structural relationships among the elements by formulas and declarations on the elements' indices; no additional space is needed except for the input data and the size of the input.

The most elegant implicit data structure for implementing priority queues is the heap. A (min-)heap [17] is a binary tree with the following properties. (*i*) It has the *heap shape*: all levels are complete, except possibly the last level where all leaves occupy the leftmost positions; (*ii*) It is *min-ordered*: the key value associated with each node is not smaller than that of its parent. The minimum element is then at the root. A (max-)heap is defined similarly. A heap on n elements can be represented by an array of length n such that the left and right children of an element in array position i are stored in positions $2i$ and $2i+1$, respectively, while its parent is in position $\lfloor \frac{i}{2} \rfloor$. The root of a heap is in position 1.

The simplest way of implementing priority deques would be to build a min-heap and a max-heap simultaneously on the same set of elements. Besides the doubling of space requirement, this implementation may have a worst-case time complexity for priority deque operations twice as high as it should. However, if we decrease the sizes of these two heaps and keep them in a suitable way, then deque operations can be supported efficiently.

More precisely, a twin-heap [9] on n elements is a binary tree with a hole at the position of the root. The left and right subtrees of the root are a min-heap of size $\lceil \frac{n}{2} \rceil$ and a max-heap of size $\lfloor \frac{n}{2} \rfloor$, respectively. Any node in the min-heap is less than the corresponding element in the max-heap. Figure 1 gives a twin-heap on 12 elements and its Hasse diagram. In this paper, we shall use the Hasse diagram [2] of a twin-heap to present the relationships among its elements.

Twin-heaps have the same asymptotic efficiency as heaps: a twin-heap can be constructed in linear time, and both the minimum and maximum elements can be found in constant time and deleted in logarithmic time, while new elements can be inserted in logarithmic time. Moreover, twin-heaps also permit sublinear-time merge operations similar to heaps [12]. Different representations of the twin-heap have been proposed, such as the diamond deque [5], the interval heap [15], and the min-max-pair heap [11]. In Section 4, we will show that these data structures are based on the same idea and present a unifying view of the structures. In what follows, the term *deque* (*structure*) will denote these structures. For the sake of simplicity, we shall assume that all the elements are distinct and drawn from a totally ordered domain.

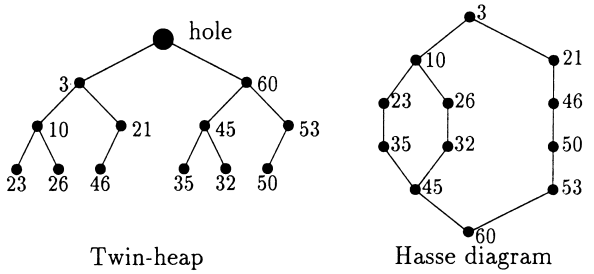

FIGURE 1. A twin-heap and its Hasse diagram.

3. CONSTRUCTION COMPLEXITY

In this section, we show how to construct deque structures fast. A lower bound of $1.5n$ comparisons for finding both the minimum and maximum in an n-element set [9] applies to our construction problem. Moreover, an information-theoretic argument improves this lower bound to approximately $2.07n$ comparisons [4], and this lower bound holds in the average case as well, where we assume that every possible input is equally likely.

A trivial algorithm for constructing deque structures would be one that first computes the median of the element-set [6, 13] and then builds the min-heap and the max-heap separately by fast known heap construction methods in [7, 10]. Such an algorithm requires $4.625n$ and $3.02n$ comparisons in the worst and average cases, respectively. Improved worst-case upper bounds of $3.5n$ comparisons [5] and $3n$ comparisons [11, 15] have also been established. A natural question, of whether these bounds can be surpassed, arises. We answer this question in the affirmative by giving a construction algorithm that has a worst-case complexity of $2.4083n + \mathcal{O}(\log^2 n)$ comparisons. The result is achieved by first designing fast algorithms for creating deques of smaller sizes, and then using these small structures as building blocks for the deque construction problem. We shall first build a binomial tree [16] on input elements and then convert it into the desired structure; this idea has been applied to the other heap-like structures [7, 14]. First, we show how to build a deque on 14 elements from a binomial tree of size 16 efficiently.

LEMMA 1: *A (min-) binomial tree on* 16 *elements can be transformed into a deque structure of size* 14 *plus one ordered-pair of elements in at most* 11 *comparisons.*

Proof: To establish the deque-ordering on 14 elements from a binomial tree of size 16, we begin with a comparison between x and y. If $x \prec y$, then we have the poset $\mathcal{P}$, else we have $\mathcal{Q}$; see Figure 3.

If $x \prec y$, we construct the desired structure $\mathcal{D}$ from $\mathcal{P}$ by (see Figure 3):

1. Letting the white nodes in $\mathcal{P}$ be the ordered-pair of elements of $\mathcal{D}$;
2. Transforming the black-circle nodes in $\mathcal{P}$ into a deque $\mathcal{D}_6^{(1)}$ of size 6 (the black-circle nodes in $\mathcal{D}$); (Cost: 3 comparisons)
3. Constructing another deque structure $\mathcal{D}_6^{(2)}$ of size 6 on all the square nodes in $\mathcal{P}$; (Cost: 2 comparisons) and
4. Inserting the element z of $\mathcal{P}$ into the partial order generated so far to create a deque $\mathcal{D}$ of size 14. (Cost: 5 comparisons)

Hence, the total number of comparisons for transforming $\mathcal{P}$ into a deque structure of size 14 plus one ordered-pair of elements is at most $3 + 2 + 5 = 10$. For the case when $y \prec x$ (i.e., the case when the poset $\mathcal{Q}$ appears; see Figure 3), we perform similar transformations. In this case, the number of comparisons needed to build the deque $\mathcal{D}_6^{(1)}$ on the black-circle nodes is 4 and only one comparison is needed to construct the deque $\mathcal{D}_6^{(2)}$ on all the square nodes. Hence, the cost is the same as that for $\mathcal{P}$. Therefore, the total cost to build a deque structure of size 14 plus one ordered-pair of elements from a binomial tree of size 16 is at most $1 + 10$. □

Since a binomial tree of size 16 can be built in 15 comparisons, a deque structure and an ordered-pair of elements can be created in at most $11 + 15 = 26$ comparisons. Notice that the ordered-pair of elements in Lemma 1 can be reused to save one comparison. A recursive application of Lemma 1 to large deque structures works. This can be done by first constructing a binomial tree on all the input elements and then applying Lemma 1 recursively. Call a deque of size n a *full* structure if the leaves of the deque occur at the last level only; i.e., $n = 2^{h+1} - 2$ for $h \geqslant 0$.

LEMMA 2: *A full deque of size n can be built in at most* $2.4083n + \mathcal{O}(\log n)$ *comparisons in the worst case.*

Proof: Since a binomial tree of size 2^k can be built in $2^k - 1$ comparisons, we only need to show that the cost, $T(2^k)$, to transform a (min-) binomial tree on 2^k ($k \geqslant 4$) elements into a deque of size $2^k - 2$ is at most 1.4083×2^k comparisons.

The algorithm for transforming a binomial tree $\mathcal{B}$ of size 2^k (k is even and $k \geqslant 4$) into a deque $\mathcal{D}$ on $2^k - 2$ elements plus one ordered-pair of elements works as

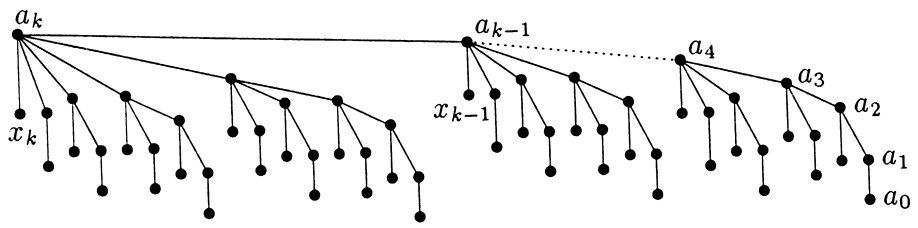

FIGURE 2. Transform binomial trees into full deques.

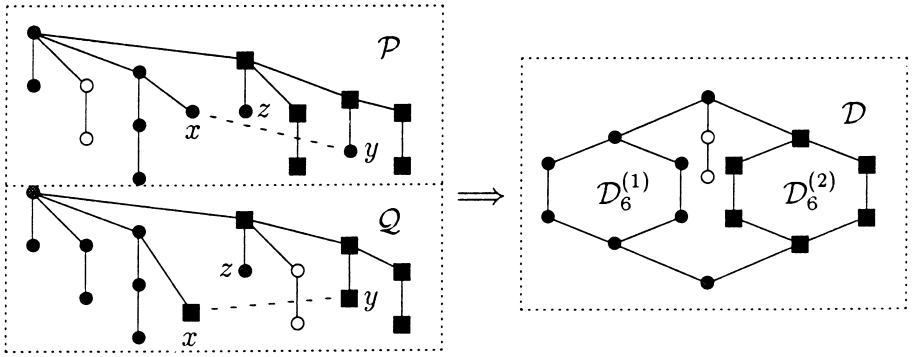

FIGURE 3. Constructing a deque of size 14 via a binomial tree.

follows. Denote the longest chain in the binomial tree by $a_k, a_{k-1}, \ldots, a_2, a_1, a_0$ (see Figure 2).

ALGORITHM 3 *Trans*$(\mathcal{B}, \mathcal{D}, k)$

1. *If $k = 4$, then convert the binomial tree $\mathcal{B}$ rooted at a_k into a deque $\mathcal{D}$ of size $2^k - 2$ plus one ordered-pair of elements, using the method provided in the proof of Lemma 1, and stop.*
2. *If $k > 4$, then call Trans$(\mathcal{B}_{k-1}, \mathcal{D}_{k-1}, k-1)$, where $\mathcal{B}_{k-1}$ is the binomial tree rooted at a_{k-1};*
3. *Combine the ordered-pair of elements produced by the previous step with all the subtrees of a_k (except the pseudo-singleton x_k (see Figure 2) and the deque $\mathcal{D}_{k-1}$ built) to create a binomial tree $\mathcal{B}'_{k-1}$ of size 2^{k-1}; (Cost: $k-2$ comparisons)*
4. *Call Trans$(\mathcal{B}'_{k-1}, \mathcal{D}'_{k-1}, k-1)$;*
5. *Insert the element x_k into the partial order created so far to form a deque $\mathcal{D}$ of size $2^k - 2$ plus one ordered-pair of elements.*

To analyse this algorithm, notice that the algorithm above (except for Step 3) can be understood as first transforming two binomial trees each of size 2^{k-1} into two deques each of size $2^{k-1} - 2$ *in turn* and then inserting an element into the structure in order to produce a deque on $2^k - 2$ elements. Moreover, to insert x_k into the structure, $(k-2)$ comparisons are needed to find the path of maximum children [3, 7], and $\lceil \log(2k-2) \rceil$ comparisons are required to do a binary search on the path of $2k-3$ nodes. Together with Lemma 1, we have then

$$\begin{cases} T(16) = 11 \quad \text{and} \\ T(2^k) = 2 \cdot T(2^{k-1}) + (k - 2 + \lceil \log(2k-2) \rceil) + (k-2) \\ \qquad = 2 \cdot T(2^{k-1}) + 2(k-1) + \lfloor \log(k-2) \rfloor, \end{cases}$$

which gives

$$T(2^k) = \frac{2^k}{2^{k_0}} T(2^{k_0}) + 2^k \left(\sum_{i=k_0}^{k-1} \frac{i}{2^i} + \frac{1}{4} \sum_{i=k_0-1}^{k-2} \frac{\lfloor \log i \rfloor}{2^i} \right)$$

for $4 \leqslant k_0 < k$. Let $k_0 = 4$. Hence, with a simple computation using the computer algebra system, viz. Maple [8], we know that

$$\begin{aligned} T(2^k) &= \frac{2^k}{16} T(16) + 2^k \left(\sum_{i=4}^{k-1} \frac{i}{2^i} + \frac{1}{4} \sum_{i=3}^{k-2} \frac{\lfloor \log i \rfloor}{2^i} \right) \\ &\leqslant \frac{11}{16} \cdot 2^k + 2^k \left(\sum_{i=4}^{\infty} \frac{i}{2^i} + \frac{1}{4} \sum_{i=3}^{\infty} \frac{\lfloor \log i \rfloor}{2^i} \right) \\ &\leqslant 1.408207 \times 2^k \end{aligned}$$

The result follows. □

Analogous to heap constructions [7], the construction complexity of full deques gives an upper bound on the cost for building deques of arbitrary sizes. In fact, for any deque on n elements, all subdeques hanging off the siblings of the elements lying on the path from the root of the min-heap to the last leaf of the min-heap are full deques, which can be created in at most $2.408207n$ comparisons by applying the above algorithm to each of them. These subdeques can be converted into a deque of size n in $2.408207n + \mathcal{O}(\log^2 n)$ comparisons. This is done by performing merge operations in turn on these full subdeques in a bottom-up fashion, which leads to an $\mathcal{O}(\log^2 n)$ additional term. More precisely, we know that

LEMMA 4: *If the cost to construct a full deque structure of any size m is $f(m) \in \Theta(m)$, then building an n-element deque takes at most $f(n) + \mathcal{O}(\log^2 n)$ comparisons.*

Combining Lemma 2 and Lemma 4 yields

THEOREM 5: *A deque structure of size n can be constructed in at most $2.4083n + \mathcal{O}(\log^2 n)$ comparisons in the worst case.*

As will be shown in the next section, the priority deque structures known in the literature, viz. the diamond deque [5], the interval heap [15], and min-max-pair heap [11], are the same structure as the deque. Hence, Theorem 5 improves over previously known construction algorithms [5, 15, 11].

4. VARIANTS OF TWIN-HEAPS

In this section we show that many of the double-ended priority queues described in the literature are in fact twin-heaps. Notice that twin-heaps can be implemented

as implicit data structures. To see this, let $\mathcal{D}[1..n]$ be an array representing a twin-heap of size n. One way to describe the twin-heap property is as follows. An array $\mathcal{D}$ is a twin-heap if:

- $\mathcal{D}[\lfloor \frac{i}{2} \rfloor] \prec \mathcal{D}[i]$ for $1 < i \leqslant \lceil \frac{n}{2} \rceil$
- $\mathcal{D}[\lfloor \frac{j}{2} \rfloor] \succ \mathcal{D}[j]$ for $\lceil \frac{n}{2} \rceil < j \leqslant n$
- $\mathcal{D}[i] \prec \mathcal{D}[i + \lceil \frac{n}{2} \rceil]$ for $1 \leqslant i \leqslant \lceil \frac{n}{2} \rceil$ (if it exists; otherwise, $\prec \mathcal{D}[\lfloor \frac{i}{2} \rfloor + \lceil \frac{n}{2} \rceil]$)

Hence, $\mathcal{D}[1]$ is the minimum element and $\mathcal{D}[\lceil \frac{n}{2} \rceil + 1]$ is the maximum. This scheme is essentially the same as the one suggested in [9].

Another approach to implement the twin-heap implicitly is to let the entries of $\mathcal{D}$ satisfy: For any odd i, $1 \leqslant i \leqslant n$,

- $\mathcal{D}[i] \prec \min\{\mathcal{D}[2i+1], \mathcal{D}[2i+3]\}$
- $\mathcal{D}[i+1] \succ \max\{\mathcal{D}[2i+2], \mathcal{D}[2i+4]\}$
- $\mathcal{D}[i] \prec \mathcal{D}[i+1]$ if any.

The array $\mathcal{D}$ with this representation is called a diamond deque [5].

Consider now an array $\mathcal{H}[1 \ldots \lceil \frac{n}{2} \rceil]$: Every element $\mathcal{H}[i]$ $(1 \leqslant i \leqslant \lceil \frac{n}{2} \rceil)$ holds two values $\mathcal{H}[i].min$ and $\mathcal{H}[i].max$ with $\mathcal{H}[i].min \prec \mathcal{H}[i].max$, where $\mathcal{H}[\lceil \frac{n}{2} \rceil]$ contains only one value $\mathcal{H}[\lceil \frac{n}{2} \rceil].min$ if n is an odd number. For $1 \leqslant i \leqslant \lceil \frac{n}{2} \rceil$,

- $\mathcal{H}[\lfloor \frac{i}{2} \rfloor].min \prec \mathcal{H}[i].min$
- $\mathcal{H}[\lfloor \frac{i}{2} \rfloor].max \succ \mathcal{H}[i].max$ if any.

Such a representation is called either an interval heap [15] or a min-max-pair heap [11]. Notice that if we expand $\mathcal{H}[1 \ldots \lceil \frac{n}{2} \rceil]$ to an array $\mathcal{D}[1 \ldots n]$ by

$$\mathcal{H}[i].min = \mathcal{D}[2i-1] \quad \text{and}$$

$$\mathcal{H}[i].max = \mathcal{D}[2i] \quad \text{(if any)} \quad \forall\, 1 \leqslant i \leqslant \left\lceil \frac{n}{2} \right\rceil$$

then we obtain a representation of the diamond deque. Application of this data structure to computational geometry can be found in [15].

In summary, all the above structures can be considered the same when they are presented in an abstract manner.

5. CONCLUSIONS

We consider the problem of efficiently constructing implicit data structures for a class of double-ended priority queues. By viewing the structures systematically, we develop a fast construction method, which improves upon the previously best known worst-case upper bounds for solving the problem.

ACKNOWLEDGEMENT

The author would like to thank the referees for very useful comments.

REFERENCES

[1] Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1974) *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts.
[2] Bogart, K. P. (1983) *Introductory Combinatorics*. Pitman Publishing Inc., Boston, Massachusetts.
[3] Carlsson, S. (1987) The deap—A double-ended heap to implement double-ended priority queues. *Information Processing Letters*, **26**, 33–36.
[4] Carlsson, S., Chen, J. and Strothotte, Th. (1989) A note on the construction of the data structure 'Deap'. *Information Processing Letters*, **31**, 315–317.
[5] Chang, S. C. and Du, M. W. (1993) Diamond deque: A simple data structure for priority deques. *Information Processing Letters*, **46**, 231–237.
[6] Floyd, R. W. and Rivest, R. L. (1975) Expected time bounds for selection. *Communications of the ACM*, **18**, 165–172.
[7] Gonnet, G. H. and Munro, J. I. (1986) Heaps on heaps. *SIAM Journal of Computing*, **15**, 964–971.
[8] Heck, A. (1993) *Introduction to Maple*. Springer-Verlag, New York, Inc.
[9] Knuth, D. E. (1973) *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts.
[10] McDiarmid, C. J. H. and Reed, B. A. (1989) Building heaps fast. *Journal of Algorithms*, **10**, 352–365.
[11] Olariu, S., Overstreet, C. M. and Wen, Z. (1991) A mergeable double-ended priority queue. *The Computer Journal*, **34**, 423–427.
[12] Sack, J.-R. and Strothotte, Th. (1985) An algorithm for merging heaps. *Acta Informatica*, **22**, 171–186.
[13] Schönhage, A., Paterson, M. and Pippenger, N. (1976) Finding the median. *Journal of Computer and System Sciences*, **13**, 184–199.
[14] Strothotte, Th., Eriksson, P. and Vallner, S. (1989) A note on constructing min-max heaps. *BIT*, **29**, 251–256.
[15] van Leeuwen, J. and Wood, D. (1993) Interval heaps. *The Computer Journal*, **36**, 209–216.
[16] Vuillemin, J. (1978) A data structure for manipulating priority queues. *Communications of the ACM*, **21**, 309–314.
[17] Williams, J. W. J. (1964) Algorithm 232: Heapsort. *Communications of the ACM*, **7**, 347–348.